

STM32F4 开发指南

V1.0 – 库函数版本

–ALIENTEK 探索者 STM32F407 开发板教程



官方店铺: <http://eboard.taobao.com>

技术论坛: www.openedv.com

官方网站: www.alientek.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 探索者 STM32F4 开发板资源初探	5
1.2 ALIENTEK 探索者 STM32F4 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	15
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 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	19
2.1.10 RS485 接口	20
2.1.11 CAN/USB 接口	21
2.1.12 EEPROM	21
2.1.13 光敏传感器	22
2.1.14 SPI FLASH	22
2.1.15 六轴加速度传感器	23
2.1.16 温湿度传感器接口	23
2.1.17 红外接收头	24
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 OLED/摄像头模块接口	26
2.1.23 有源蜂鸣器	26

2.1.24 SD 卡接口	27
2.1.25 ATK 模块接口	27
2.1.26 多功能端口	28
2.1.27 以太网接口 (RJ45)	29
2.1.28 I2S 音频编解码器	30
2.1.29 电源	30
2.1.30 电源输入输出接口	31
2.1.31 USB 串口	32
2.2 开发板使用注意事项	32
2.3 STM32F4 学习方法	33
第二篇 软件篇	35
第三章 MDK5 软件入门	36
3.1 STM32 官方标准固件库简介	36
3.1.1 库开发与寄存器开发的关系	36
3.1.2 STM32 固件库与 CMSIS 标准讲解	37
3.1.3 STM32F4 官方库包介绍	38
3.1.3.1 文件夹介绍:	39
3.1.3.2 关键文件介绍:	39
3.2 MDK5 简介	41
3.3 新建基于 STM32F40x 固件库的 MDK5 工程模板	42
3.3.1 MDK5 安装步骤	43
3.3.2 新建工程模板	43
3.4 程序下载与调试	69
3.4.1 STM32 串口程序下载	69
3.4.2 JLINK 下载与调试程序	75
3.5 MDK5 使用技巧	85
3.5.1 文本美化	85
3.5.2 语法检测&代码提示	88
3.5.3 代码编辑技巧	89
3.5.4 其他小技巧	93
第四章 STM32F4 开发基础知识入门	96
4.1 MDK 下 C 语言基础复习	96
4.1.1 位操作	96

4.1.2 define 宏定义	97
4.1.3 ifdef 条件编译	97
4.1.4 extern 变量申明	98
4.1.5 typedef 类型别名	98
4.1.6 结构体	99
4.2 STM32F4 总线架构	101
4.3 STM32F4 时钟系统	102
4.3.1 STM32F4 时钟树概述	102
4.3.2 STM32F4 时钟初始化配置	105
4.3.3 STM32F4 时钟使能和配置	110
4.4 IO 引脚复用器和映射	112
4.5 STM32 NVIC 中断优先级管理	115
4.6 MDK 中寄存器地址名称映射分析	119
4.7 MDK 固件库快速组织代码技巧	121
第五章 SYSTEM 文件夹介绍	127
5.1 delay 文件夹代码介绍	127
5.1.1 delay_init 函数	128
5.1.2 delay_us 函数	129
5.1.3 delay_xms 函数	131
5.1.4 delay_ms 函数	132
5.2 sys 文件夹代码介绍	133
5.2.1 IO 口的位操作实现	133
5.3 usart 文件夹介绍	134
5.3.1 printf 函数支持	135
5.3.2 uart_init 函数	135
5.3.3 USART1_IRQHandler 函数	138
第三篇 实战篇	140
第六章 跑马灯实验	141
6.1 STM32F4 IO 简介	141
6.2 硬件设计	149
6.3 软件设计	149
6.4 下载验证	162
第七章 蜂鸣器实验	164

7.1 蜂鸣器简介	164
7.2 硬件设计	164
7.3 软件设计	165
7.4 下载验证	168
第八章 按键输入实验	169
8.1 STM32F4 IO 口简介	169
8.2 硬件设计	169
8.3 软件设计	169
8.4 下载验证	173
第九章 串口通信实验	174
9.1 STM32F4 串口简介	174
9.2 硬件设计	177
9.3 软件设计	178
9.4 下载验证	180
第十章 外部中断实验	182
10.1 STM32F4 外部中断简介	182
10.2 硬件设计	185
10.3 软件设计	185
10.4 下载验证	189
第十一章 独立看门狗 (IWDG) 实验	190
11.1 STM32F4 独立看门狗简介	190
11.2 硬件设计	192
11.3 软件设计	192
11.4 下载验证	194
第十二章 窗口看门狗 (WWDG) 实验	195
12.1 STM32F4 窗口看门狗简介	195
12.2 硬件设计	197
12.3 软件设计	198
12.4 下载验证	199
第十三章 定时器中断实验	200
13.1 STM32F4 通用定时器简介	200
13.2 硬件设计	204

13.3 软件设计	204
13.4 下载验证	206
第十四章 PWM 输出实验.....	207
14.1 PWM 简介.....	207
14.2 硬件设计	211
14.3 软件设计	211
14.4 下载验证	212
第十五章 输入捕获实验	214
15.1 输入捕获简介	214
15.2 硬件设计	219
15.3 软件设计	219
15.4 下载验证	223
第十六章 电容触摸按键实验	224
16.1 电容触摸按键简介	224
16.2 硬件设计	225
16.3 软件设计	225
16.4 下载验证	230
第十七章 OLED 显示实验	231
17.1 OLED 简介	231
17.2 硬件设计	237
17.3 软件设计	237
17.4 下载验证	245
第十八章 TFTLCD 显示实验	247
18.1 TFTLCD&FSMC 简介	247
18.1.1 TFTLCD 简介	247
18.1.2 FSMC 简介	253
18.2 硬件设计	262
18.3 软件设计	263
18.4 下载验证	275
第十九章 USMART 调试组件实验.....	276
19.1 USMART 调试组件简介	276
19.2 硬件设计	279

19.3 软件设计	280
19.4 下载验证	284
第二十章 RTC 实时时钟实验.....	289
20.1 STM32F4 RTC 时钟简介	289
20.2 硬件设计	299
20.3 软件设计	299
20.4 下载验证	305
第二十一章 硬件随机数实验.....	307
21.1 STM32F4 随机数发生器简介.....	307
21.2 硬件设计	309
21.3 软件设计	309
21.4 下载验证	311
第二十二章 待机唤醒实验	312
22.1 STM32F4 待机模式简介	312
22.2 硬件设计	315
22.3 软件设计	315
22.4 下载与测试	318
第二十三章 ADC 实验.....	319
23.1 STM32F4 ADC 简介	319
23.2 硬件设计	326
23.3 软件设计	327
23.4 下载验证	329
第二十四章 内部温度传感器实验	331
24.1 STM32F4 内部温度传感器简介	331
24.2 硬件设计	331
24.3 软件设计	332
24.4 下载验证	334
第二十五章 光敏传感器实验	335
25.1 光敏传感器简介	335
25.2 硬件设计	335
25.3 软件设计	336
25.4 下载验证	339

第二十六章 DAC 实验.....	340
26.1 STM32F4 DAC 简介	340
26.2 硬件设计	346
26.3 软件设计	347
26.4 下载验证	350
第二十七章 PWM DAC 实验	351
27.1 PWM DAC 简介	351
27.2 硬件设计	353
27.3 软件设计	354
27.4 下载验证	356
第二十八章 DMA 实验.....	358
28.1 STM32F4 DMA 简介	358
28.2 硬件设计	365
28.3 软件设计	366
28.4 下载验证	369
第二十九章 IIC 实验.....	372
29.1 IIC 简介	372
29.2 硬件设计	373
29.3 软件设计	373
29.4 下载验证	381
第三十章 SPI 实验	382
30.1 SPI 简介	382
30.2 硬件设计	386
30.3 软件设计	386
30.4 下载验证	391
第三十一章 485 实验	393
31.1 485 简介	393
31.2 硬件设计	394
31.3 软件设计	396
31.4 下载验证	400
第三十二章 CAN 通讯实验.....	402
32.1 CAN 简介	402

32.2 硬件设计	422
32.3 软件设计	423
32.4 下载验证	427
第三十三章 触摸屏实验	430
33.1 触摸屏简介	430
33.1.1 电阻式触摸屏	430
33.1.2 电容式触摸屏	431
33.2 硬件设计	435
33.3 软件设计	436
33.4 下载验证	451
第三十四章 红外遥控实验	454
34.1 红外遥控简介	454
34.2 硬件设计	455
34.3 软件设计	456
34.4 下载验证	461
第三十五章 DS18B20 数字温度传感器实验	463
35.1 DS18B20 简介	463
35.2 硬件设计	464
35.3 软件设计	465
35.4 下载验证	470
第三十六章 DHT11 数字温湿度传感器实验	471
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 MPU6050 六轴传感器实验	478
37.1 MPU6050 简介	478
37.1.1 MPU6050 基础介绍	478
37.1.2 DMP 使用简介	483
37.2 硬件设计	486
37.3 软件设计	487
37.4 下载验证	494
第三十八章 无线通信实验	496

38.1 NRF24L01 无线模块简介	496
38.2 硬件设计	497
38.3 软件设计	497
38.4 下载验证	506
第三十九章 FLASH 模拟 EEPROM 实验	508
39.1 STM32F4 FLASH 简介	508
39.2 硬件设计	515
39.3 软件设计	515
39.4 下载验证	519
第四十章 摄像头实验	520
40.1 OV2640&DCMI 简介	520
40.1.1 OV2640 简介	520
40.1.2 STM32F4 DCMI 接口简介	525
40.2 硬件设计	531
40.3 软件设计	533
40.4 下载验证	541
第四十一章 外部 SRAM 实验	545
41.1 IS62WV51216 简介	545
41.2 硬件设计	547
41.3 软件设计	547
41.4 下载验证	551
第四十二章 内存管理实验	553
42.1 内存管理简介	553
42.2 硬件设计	554
42.3 软件设计	554
42.4 下载验证	562
第四十三章 SD 卡实验	563
43.1 SDIO 简介	563
43.1.1 SDIO 主要功能及框图	563
43.1.2 SDIO 的时钟	564
43.1.3 SDIO 的命令与响应	565
43.1.4 SDIO 相关寄存器介绍	567
43.1.5 SD 卡初始化流程	572

43.2 硬件设计	574
43.3 软件设计	575
43.4 下载验证	585
第四十四章 FATFS 实验	587
44.1 FATFS 简介	587
44.2 硬件设计	592
44.3 软件设计	592
44.4 下载验证	600
第四十五章 汉字显示实验	602
45.1 汉字显示原理简介	602
45.2 硬件设计	606
45.3 软件设计	606
45.4 下载验证	616
第四十六章 图片显示实验	618
46.1 图片格式简介	618
46.2 硬件设计	619
46.3 软件设计	620
46.4 下载验证	629
第四十七章 照相机实验	630
47.1 BMP&JPEG 编码简介	630
47.1.1 BMP 编码简介	630
47.1.2 JPEG 编码简介	633
47.2 硬件设计	634
47.3 软件设计	634
47.4 下载验证	644
第四十八章 音乐播放器实验	646
48.1 WAV&WM8978&I2S 简介	646
48.1.1 WAV 简介	646
48.1.2 WM8978 简介	648
48.1.3 I2S 简介	651
48.2 硬件设计	658
48.3 软件设计	659
48.4 下载验证	672

第四十九章 录音机实验	674
49.1 I2S 录音简介	674
49.2 硬件设计	676
49.3 软件设计	677
49.4 下载验证	685
第五十章 视频播放器实验	687
50.1 AVI&libjpeg 简介	687
50.1.1 AVI 简介	687
50.1.2 libjpeg 简介	692
50.2 硬件设计	696
50.3 软件设计	696
50.4 下载验证	706
第五十一章 FPU 测试(Julia 分形)实验	710
51.1 FPU&Julia 分形简介	710
51.1.1 FPU 简介	710
51.1.2 Julia 分形简介	711
51.2 硬件设计	712
51.3 软件设计	713
51.4 下载验证	716
第五十二章 DSP 测试实验	718
52.1 DSP 简介与环境搭建	718
52.1.1 STM32F4 DSP 简介	718
52.1.2 DSP 库运行环境搭建	721
52.2 硬件设计	724
52.3 软件设计	724
52.3.1 DSP BasicMath 测试	724
52.3.1 DSP FFT 测试	726
52.4 下载验证	729
第五十三章 手写识别实验	732
53.1 手写识别简介	732
53.2 硬件设计	736
53.3 软件设计	736
53.4 下载验证	741

第五十四章 T9 拼音输入法实验.....	744
54.1 拼音输入法简介	744
54.2 硬件设计	746
54.3 软件设计	746
54.4 下载验证	754
第五十五章 串口 IAP 实验.....	756
55.1 IAP 简介.....	756
55.2 硬件设计	762
55.3 软件设计	762
55.4 下载验证	768
第五十六章 USB 读卡器(Slave)实验.....	770
56.1 USB 简介	770
56.2 硬件设计	773
56.3 软件设计	774
56.4 下载验证	780
第五十七章 USB 声卡(Slave)实验.....	782
57.1 USB 声卡简介	782
57.2 硬件设计	782
57.3 软件设计	782
57.4 下载验证	789
第五十八章 USB U 盘(Host)实验	791
58.1 U 盘简介	791
58.2 硬件设计	791
58.3 软件设计	792
58.4 下载验证	798
第五十九章 USB 鼠标键盘(Host)实验	801
59.1 USB 鼠标键盘简介	801
59.2 硬件设计	801
59.3 软件设计	801
59.4 下载验证	808
第六十章 网络通信实验	810
60.1 STM32F4 以太网以及 TCP/IP LWIP 简介	810

60.1.1 STM32F4 以太网简介	810
60.1.2 TCP/IP LWIP 简介	815
60.2 硬件设计	818
60.3 软件设计	819
60.4 下载验证	822
60.4.1 Web Server 测试.....	824
60.4.2 TCP Server 测试.....	827
60.4.3 TCP Client 测试	828
60.4.4 UDP 测试	830
第六十一章 UCOSII 实验 1-任务调度	832
61.1 UCOSII 简介	832
61.2 硬件设计	837
61.3 软件设计	837
61.4 下载验证	841
58.5 任务删除, 挂起和恢复测试	841
第六十二章 UCOSII 实验 2-信号量和邮箱	845
62.1 UCOSII 信号量和邮箱简介	845
62.2 硬件设计	847
62.3 软件设计	848
62.4 下载验证	854
第六十三章 UCOSII 实验 3-消息队列、信号量集和软件定时器	856
63.1 UCOSII 消息队列、信号量集和软件定时器简介	856
63.2 硬件设计	864
63.3 软件设计	864
63.4 下载验证	874
第六十四章 探索者 STM32F4 开发板综合实验	876
64.1 探索者 STM32F4 开发板综合实验简介	877
64.2 探索者 STM32F4 开发板综合实验详解	877
64.2.1 电子图书	882
64.2.2 数码相框	884
64.2.3 音乐播放	886
64.2.4 视频播放	890
64.2.5 时钟	891

64.2.6 系统设置	892
64.2.7 FC 游戏机	903
64.2.8 记事本	907
64.2.9 运行器	909
64.2.10 手写画笔	910
64.2.11 照相机	913
64.2.12 录音机	917
64.2.13 USB 连接	920
64.2.14 网络通信	921
64.2.15 无线传书	926
64.2.16 计算器	928
64.2.17 拨号	931
64.2.18 应用中心	934
64.2.19 短信	935

内容简介

本手册将由浅入深,带领大家学习 STM32F407 的各个功能,为您开启全新的 STM32 之旅。本手册总共分为三篇: 1, 硬件篇, 主要介绍本手册所讲实例对应的实验平台; 2, 软件篇, 主要介绍 STM32F4 常用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 59 个实例带领大家一步步深入了解 STM32F4。

本手册为 ALIENTEK 探索者 STM32F4 开发板的配套教程, 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/仿真器下载到开发板即可看到实验现象, 亲自体验实验过程。

本手册不仅非常适合广大学生和电子爱好者学习 STM32F4, 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

作为 Cortex M3 市场的最大占有者，ST 公司在 2011 年又推出基于 Cortex M4 内核的 STM32F4 系列产品，相对与 STM32F1/F2 等 Cortex M3 产品，STM32F4 最大的优势，就是新增了硬件 FPU 单元以及 DSP 指令，同时，STM32F4 的主频也提高了很多，达到 168Mhz（可获得 210DMIPS 的处理能力），这使得 STM32F4 尤其适用于需要浮点运算或 DSP 处理的应用，也被称之为：DSC，具有非常广泛的应用前景。

STM32F4 相对于 STM32F1，主要优势如下：

- 1, 更先进的内核。STM32F4 采用 Cortex M4 内核，带 FPU 和 DSP 指令集，而 STM32F1 采用的是 Cortex M3 内核，不带 FPU 和 DSP 指令集。
- 2, 更多的资源。STM32F4 拥有多达 192KB 的片内 SRAM，带摄像头接口（DCMI）、加密处理器（CRYP）、USB 高速 OTG、真随机数发生器、OTP 存储器等。
- 3, 增强的外设功能。对于相同的外设部分，STM32F4 具有更快的模数转换速度、更低的 ADC/DAC 工作电压、32 位定时器、带日历功能的实时时钟（RTC）、IO 复用功能大大增强、4K 字节的电池备份 SRAM 以及更快的 USART 和 SPI 通信速度。
- 4, 更高的性能。STM32F4 最高运行频率可达 168Mhz，而 STM32F1 只能到 72Mhz；STM32F4 拥有 ART 自适应实时加速器，可以达到相当于 FLASH 零等待周期的性能，STM32F1 则需要等待周期；STM32F4 的 FSMC 采用 32 位多重 AHB 总线矩阵，相比 STM32F1 总线访问速度明显提高。
- 5, 更低的功耗。STM32F40x 的功耗为：238uA/Mhz，其中低功耗版本的 STM32F401 更是低到：140uA/Mhz，而 STM32F1 则高达 421uA/Mhz。

STM32F4 家族目前拥有：STM32F40x、STM32F41x、STM32F42x 和 STM32F43x 等几个系列，数十个产品型号，不同型号之间软件和引脚具有良好的兼容性，可方便客户迅速升级产品。其中，STM32F42x/43x 系列带了 LCD 控制器和 SDRAM 接口，对于想要驱动大屏或需要大内存的朋友来说，是个不错的选择。目前 STM32F4 这些芯片型号都已量产，可以方便的购买到，不过目前来说，性价比最高的是 STM32F407，本手册，我们将以 STM32F407 为例，向大家讲解 STM32F4 的学习。

学习 STM32F4 有几份资料经常用到：

- 《STM32F4xx 中文参考手册》
- 《STM32F3 与 F4 系列 Cortex M4 内核编程手册》英文版
- 《Cortex M3 与 M4 权威指南》英文版

其中，最常用的是《STM32F4xx 中文参考手册》，该文档是 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-M4 构架进行多少介绍，读者只能根据自己对书本的理解来编写相关代码，该文档目前已经有中文版本的了，极大的方便了大家的学习。

而《STM32F3 与 F4 系列 Cortex M4 内核编程手册》这个文档，则重点介绍了 Cortex M4 内核的汇编指令及其使用，以及内核相关寄存器（比如：SCB, NVIC, SYSTICK 等寄存器），是《STM32F4xx 中文参考手册》的重要补充，很多在《STM32F4xx 中文参考手册》无法找到的内容，都可以在这里找到答案，不过目前该文档没有中文版本，只有英文版。

最后，《Cortex M3 与 M4 权威指南》这个文档，详细介绍了 Cortex M3 和 Cortex M4 内核

的体系架构，并配有简单实例，对于想深入了解 Cortex M4 内核的朋友，此文档是非常好的参考资料，不过该文档目前只有英文版的，对于英文很犯怵的朋友，可能有点不适应。不过由于 CM3 和 CM4 很多地方都是通用的，所以有的时候，可以参考《Cortex M3 权威指南(中文)》这个文档。

本手册将结合以上三份资料的优点，从库函数级别出发，深入浅出，向读者展示 STM32F4 的各种功能。总共配有 58 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 STM32F4 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、音乐播放、视频播放等。所有实例在 MDK5.11A 编译器下编译通过，大家只需下载程序到 ALIENTEK 探索者 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，本手册都非常适合。尤其对于初学者，本手册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。

本手册的实验平台是 ALIENTEK 探索者 STM32 开发板，有这款开发板的朋友则直接可以拿本手册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 探索者 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（比如 IO 口修改）稍做修改，使之适合你的开发板即可。

作者力求将本手册的内容写好，由于能力有限，手册中但难免会有出错的地方，如果大家发现手册中有什么错误的地方，还请告诉本人一声，本人邮箱：xingyidianzi@foxmail.com，也可以去 www.openedv.com 论坛给我留言，在此先向各位读者表示诚挚的感谢。

第一篇 硬件篇

实践出真知，要想学好 STM32F4，实验平台必不可少！本篇将详细介绍我们用来学习 STM32F4 的硬件平台：ALIENTEK 探索者 STM32F4 开发板，通过该篇的介绍，你将了解到我们的学习平台 ALIENTEK 探索者 STM32F4 开发板的功能及特点。

为了让读者更好的使用 ALIENTEK 探索者 STM32F4 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 探索者 STM32F4 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 探索者 STM32F4 开发板资源初探；
- 1.2, ALIENTEK 探索者 STM32F4 开发板资源说明；

1.1 ALIENTEK 探索者 STM32F4 开发板资源初探

在 ALIENTEK 探索者 STM32F4 开发板之前，ALIENTEK 推出的两款 STM32F1 系列开发板：MinistM32 开发板和战舰 STM32 开发板，常年稳居淘宝销量冠军，累计出货超过 3W 多套。而这款探索者 STM32F4 开发板，则是 ALIENTEK 推出的首款 Cortex M4 开发板，下面我们将开始介绍探索者 STM32F4 开发板。

ALIENTEK 探索者 STM32F4 开发板的资源图如图 1.1.1 所示：

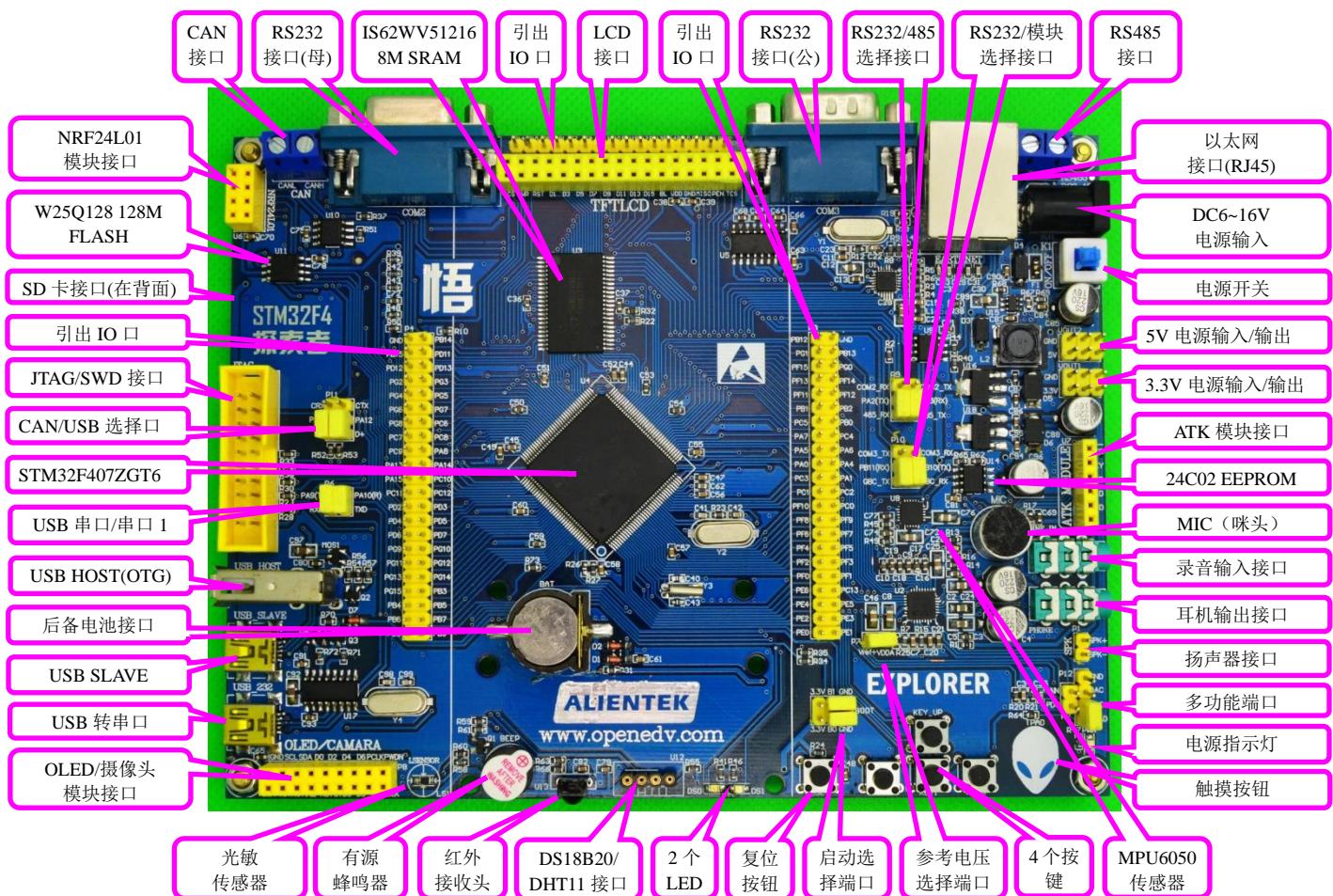


图 1.1.1 探索者 STM32F4 开发板资源图

从图 1.1.1 可以看出，ALIENTEK 探索者 STM32F4 开发板，资源十分丰富，并把 STM32F407 的内部资源发挥到了极致，基本所有 STM32F407 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 121mm*160mm 大小，板子的设计充分考虑了人性化设计，并结合

ALIENTEK 多年的 STM32 开发板设计经验，同时听取了很多网友以及客户的建议，经过多次改进（面市之前，硬件改版超过 5 次，目前面市版本为 V1.5），最终确定了这样的设计。

ALIENTEK 探索者 STM32F4 开发板板载资源如下：

- ◆ CPU: STM32F407ZGT6, LQFP144, FLASH: 1024K, SRAM: 192K;
- ◆ 外扩 SRAM: IS62WV51216, 1M 字节
- ◆ 外扩 SPI FLASH: W25Q128, 16M 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 个六轴（陀螺仪+加速度）传感器芯片, MPU6050
- ◆ 1 个高性能音频编解码芯片, WM8978
- ◆ 1 个 2.4G 无线模块接口，支持 NRF24L01 无线模块
- ◆ 1 路 CAN 接口，采用 TJA1050 芯片
- ◆ 1 路 485 接口，采用 SP3485 芯片
- ◆ 2 路 RS232 串口（一公一母）接口，采用 SP3232 芯片
- ◆ 1 路数字温湿度传感器接口，支持 DS18B20/DHT11 等
- ◆ 1 个 ATK 模块接口，支持 ALIENTEK 蓝牙/GPS 模块
- ◆ 1 个光敏传感器
- ◆ 1 个标准的 2.4/2.8/3.5/4.3/7 寸 LCD 接口，支持电阻/电容触摸屏
- ◆ 1 个摄像头模块接口
- ◆ 1 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 从机通信
- ◆ 1 个 USB HOST(OTG)接口，用于 USB 主机通信
- ◆ 1 个有源蜂鸣器
- ◆ 1 个 RS232/RS485 选择接口
- ◆ 1 个 RS232/模块选择接口
- ◆ 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）
- ◆ 1 个百兆以太网接口（RJ45）
- ◆ 1 个标准的 JTAG/SWD 调试下载口
- ◆ 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 路扬声器输出接口，可接 1W 左右小喇叭
- ◆ 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：DC6~16V）
- ◆ 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 KEY_UP(即 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 探索者 STM32F4 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。主芯片采用自带 1M 字节 FLASH 的 STM32F407ZGT6，并外扩 1M 字节 SRAM 和 16M 字节 FLASH，满足大内存需求和大数据存储。板载高性能音频编解码芯片、六轴传感器、百兆网卡、光敏传感器以及各种接口芯片，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 探索者 STM32F4 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探索者 STM32F4 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. NRF24L01 模块接口

这是开发板板载的 NRF24L01 模块接口 (U6)，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。

2. W25Q128 128M FLASH

这是开发板外扩的 SPI FLASH 芯片 (U11)，容量为 128Mbit，也就是 16M 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 16M 字节还不够用，你可以把数据存放在外部 SD 卡。

3. SD 卡接口

这是开发板板载的一个标准 SD 卡接口 (SD_CARD)，该接口在开发板的背面，采用大 SD 卡接口（即相机卡，TF 卡是不能直接插的，TF 卡得加卡套才行），SDIO 方式驱动，有了这个 SD 卡接口，就可以满足海量数据存储的需求。

4. 引出 IO 口（总共有三处）

这是开发板 IO 引出端口，总共有三组主 IO 引出口：P3、P4 和 P5。其中，P3 和 P4 分别采用 2*22 排针引出，总共引出 86 个 IO 口，P5 采用 1*16 排针，按顺序引出 FSMC_D0~D15 等 16 个 IO 口。而 STM32F407ZGT6 总共只有 112 个 IO，除去 RTC 晶振占用的 2 个 IO，还剩下 110 个，前面三组主引出排针，总共引出：102 个 IO，剩下的分别通过：P6、P9、P10 和 P11 引出。

5. JTAG/SWD 接口

这是 ALIENTEK 探索者 STM32F4 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**强烈建议仿真器使用 SWD 模式！**

6. CAN/USB 选择口

这是一个 CAN/USB 的选择接口 (P11)，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。

7. STM32F407ZGT6

这是开发板的核心芯片 (U4)，型号为：STM32F407ZGT6。该芯片集成 FPU 和 DSP 指令，并具有 192KB SRAM、1024KB FLASH、12 个 16 位定时器、2 个 32 位定时器、2 个 DMA 控制器(共 16 个通道)、3 个 SPI、2 个全双工 I2S、3 个 IIC、6 个串口、2 个 USB(支持 HOST /SLAVE)、2 个 CAN、3 个 12 位 ADC、2 个 12 位 DAC、1 个 RTC (带日历功能)、1 个 SDIO 接口、1 个 FSMC 接口、1 个 10/100M 以太网 MAC 控制器、1 个摄像头接口、1 个硬件随机数生成器、以及 112 个通用 IO 口等。

8. USB 串口/串口 1

这是 USB 串口同 STM32F407ZGT6 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 探索者 STM32F4 开发板当成一个 USB 转 TTL 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 探索者 STM32F4 开发板上。

9. USB HOST(OTG)

这是开发板板载的一个侧插式的 USB-A 座 (USB_HOST)，由于 STM32F4 的 USB 是支持 HOST 的，所以我们可以通过这个 USB-A 座，连接 U 盘/USB 鼠标/USB 键盘等其他 USB 从设备，从而实现 USB 主机功能。不过特别注意，由于 USB HOST 和 USB SLAVE 是共用 PA11 和 PA12，所以两者不可以同时使用。

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 SLAVE

这是开发板板载的一个 MiniUSB 头 (USB_SLAVE)，用于 USB 从机 (SLAVE) 通信，一般用于 STM32 与电脑的 USB 通信。通过此 MiniUSB 头，开发板就可以和电脑进行 USB 通信了。注意：该接口不能和 USB HOST 同时使用。

开发板总共板载了 2 个 MiniUSB 头，一个 (USB_232) 用于 USB 转串口，连接 CH340G 芯片；另外一个 (USB_SLAVE) 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。

12. 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。

13. 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。

14. 光敏传感器

这是开发板板载的一个光敏传感器 (LS1)，通过该传感器，开发板可以感知周围环境光线的变化，从而可以实现类似自动背光控制的应用。

15. 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。

16. 红外接收头

这是开发板的红外接收头 (U13)，可以实现红外遥控功能，通过这个接收头，可以接受市面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。

探索者 STM32F4 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 1.2.1.1 所示：



图 1.2.1.1 红外遥控器

17. DS18B20/DHT11 接口

这是开发板的一个复用接口 (U12)，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。

18. 2 个 LED

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。探索者 STM32F4 开发板几乎每个实例都使用了 LED 来指示程序的运行状态。

19. 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。

20. 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。

21. 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 (VDDA)。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 Vref+ 和 GND 即可。

22. 4 个按键

这是开发板板载的 4 个机械式输入按键 (KEY0、KEY1、KEY2 和 KEY_UP)，其中 KEY_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 KEY_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。

23. MPU6050 传感器

这是开发板板载的一个六轴传感器 (U8)，MPU6050 是一个高性能的六轴传感器，它内部集成 1 个三轴加速度传感器和 1 个三轴陀螺仪，并且带 DMP 功能，该传感器在四轴飞控方面应用非常广泛。所以喜欢玩四轴的朋友，也可以通过我们的开发板进行学习。

24. 触摸按钮

这是开发板板载的一个电容触摸输入按键 (TPAD)，利用电容充放电原理，实现触摸按键检测。

25. 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。

26. 多功能端口

这是 1 个由 6 个排针组成的一个接口 (P2&P12)。不过大家可别小看这 6 个排针，这可是本开发板设计的很巧妙的一个端口 (由 P2 和 P12 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。

27. 扬声器接口

这是开发板预留的一个扬声器接口 (P1)，可以外接 1W (8 Ω) 左右的小喇叭 (喇叭需要自备)，这样使用 WM8978 放音的时候，就可以直接推动喇叭输出音频了。

28. 耳机输出接口

这是开发板板载的音频输出接口 (PHONE)，该接口可以插 3.5mm 的耳机，当 WM8978 放音的时候，就可以通过在该接口插入耳机，欣赏音乐。

29. 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。

30. MIC (咪头)

这是开发板的板载录音输入口 (MIC)，该咪头直接接到 WM8978 的输入上，可以用来实

现录音功能。

31. 24C02 EEPROM

这是开发板板载的 EEPROM 芯片 (U14)，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。

32. ATK 模块接口

这是开发板板载的一个 ALIENTEK 通用模块接口 (U7)，目前可以支持 ALIENTEK 开发的 GPS 模块和蓝牙模块，直接插上对应的模块，就可以进行开发。后续我们将开发更多兼容该接口的其他模块，实现更强大的扩展性能。

33. 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) (VOUT1)，用于给外部提供 3.3V 的电源，也可以用于从外部接 3.3V 的电源给板子供电。

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，有了 ALIENTEK 探索者 STM32F4 开发板，你就可以很方便的拥有一个简单的 3.3V 电源 (USB 供电的时候，最大电流不能超过 500mA，外部供电的时候，最大可达 1000mA)。

34. 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)，该排针用于给外部提供 5V 的电源，也可以用于从外部接 5V 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 5V 电源而苦恼不已，ALIENTEK 充分考虑到了大家需求，有了这组 5V 排针，你就可以很方便的拥有一个简单的 5V 电源 (USB 供电的时候，最大电流不能超过 500mA，外部供电的时候，最大可达 1000mA)。

35. 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。

36. DC6~16V 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。在耗电比较大的情况下，比如用到 4.3 屏/7 寸屏/网口的时候，建议使用外部电源供电，可以提供足够的电流给开发板使用。

37. 以太网接口 (RJ45)

这是开发板板载的网口 (EARTHNET)，可以用来连接网线，实现网络通信功能。该接口使用 STM32F4 内部的 MAC 控制器外加 PHY 芯片，实现 10/100M 网络的支持。

38. RS485 总线接口

这是开发板板载的 RS485 总线接口 (RS485)，通过 2 个端口和外部 485 设备连接。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！

39. RS232/模块选择接口

这是开发板板载的一个 RS232 (COM3) /ATK 模块接口 (U7) 选择接口 (P10)，通过该选择接口，我们可以选择 STM32 的串口 3 连接在 COM3 还是连接在 ATK 模块接口上面，以实现不同的应用需求。这样的设计还有一个好处，就是我们的开发板还可以充当 RS232 到 TTL 串口的转换 (注意，这里的 TTL 高电平是 3.3V)。

40. RS232/485 选择接口

这是开发板板载的 RS232 (COM2) /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（COM2）和 RS485 共用一个串口，通过 P9 来设置当前是使用 RS232（COM2）还是 RS485。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）。

41. RS232 接口（公）

这是开发板板载的一个 RS232 接口（COM3），通过一个标准的 DB9 公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。

42. LCD 接口

这是开发板板载的 LCD 模块接口，该接口兼容 ALIENTEK 全系列 TFTLCD 模块，包括：2.4 寸、2.8 寸、3.5 寸、4.3 寸和 7 寸等 TFTLCD 模块，并且支持电阻/电容触摸功能。

43. IS62WV51216 8M SRAM

这是开发板外扩的 SRAM 芯（U3）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

44. RS232 接口（母）

这是开发板板载的另外一个 RS232 接口（COM2），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

45. CAN 接口

这是开发板板载的 CAN 总线接口（CAN），通过 2 个端口和外部 CAN 总线连接，即 CANH 和 CANL。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！

1.2.2 软件资源说明

上面我们详细介绍了 ALIENTEK 探索者 STM32F4 开发板的硬件资源。接下来，我们将向大家简要介绍一下探索者 STM32F4 开发板的软件资源。

探索者 STM32F4 开发板提供的标准例程多达 59 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（本手册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。

探索者 STM32F4 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	31	DHT11 数字温湿度传感器实验
2	蜂鸣器实验	32	MPU6050 六轴传感器实验
3	按键输入实验	33	无线通信实验
4	串口通信实验	34	FLASH 模拟 EEPROM 实验
5	外部中断实验	35	摄像头实验
6	独立看门狗实验	36	外部 SRAM 实验
7	窗口看门狗实验	37	内存管理实验
8	定时器中断实验	38	SD 卡实验
9	PWM 输出实验	39	FATFS 实验
10	输入捕获实验	40	汉字显示实验
11	电容触摸按键实验	41	图片显示实验

12	OLED 实验	42	照相机实验
13	TFTLCD 实验	43	音乐播放器实验
14	USMART 调试实验	44	录音机实验
15	RTC 实验	45	视频播放器实验
16	硬件随机数实验	46	FPU 测试(Julia 分形)实验
17	待机唤醒实验	47	DSP 测试实验
18	ADC 实验	48	手写识别实验
19	内部温度传感器实验	49	T9 拼音输入法实验
20	光敏传感器实验	50	串口 IAP 实验
21	DAC 实验	51	USB 读卡器(Slave)实验
22	PWM DAC 实验	52	USB 声卡(Slave)实验
23	DMA 实验	53	USB U 盘(Host)实验
24	IIC 实验	54	USB 鼠标键盘(Host)实验
25	SPI 实验	55	网络通信实验
26	485 实验	56	UCOSII 实验 1-任务调度
27	CAN 实验	57	UCOSII 实验 2-信号量和邮箱
28	触摸屏实验	58	UCOSII 实验 3-消息队列、信号量集 和软件定时器
29	红外遥控实验	59	综合测试实验
30	DS18B20 数字温度传感器实验		

表 1.2.2.1 ALIENTEK 探索者 STM32F4 开发板例程表

从上表可以看出，ALIENTEK 探索者 STM32F4 开发板的例程基本上涵盖了 STM32F407ZGT6 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、USMART 调试实验、ucosii 实验、内存管理实验、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 探索者 STM32F4 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 探索者 STM32F4 开发板也绝对是一个不错的选择。

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 探索者 STM32F4 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 探索者 STM32F4 开发板选择的是 STM32F407ZGT6 作为 MCU，该芯片是 STM32F407 里面配置非常强大的了，它拥有的资源包括：集成 FPU 和 DSP 指令，并具有 192KB SRAM、1024KB FLASH、12 个 16 位定时器、2 个 32 位定时器、2 个 DMA 控制器（共 16 个通道）、3 个 SPI、2 个全双工 I2S、3 个 IIC、6 个串口、2 个 USB（支持 HOST /SLAVE）、2 个 CAN、3 个 12 位 ADC、2 个 12 位 DAC、1 个 RTC（带日历功能）、1 个 SDIO 接口、1 个 FSMC 接口、1 个 10/100M 以太网 MAC 控制器、1 个摄像头接口、1 个硬件随机数生成器、以及 112 个通用 IO 口等。该芯片的配置十分强悍，很多功能相对 STM32F1 来说进行了重大改进，比如 FSMC 的速度，F4 刷屏速度可达 3300W 像素/秒，而 F1 的速度则只有 500W 左右。

MCU 部分的原理图如图 2.1.1.1（因为原理图比较大，缩小下来可能有点看不清，请大家打开开发板光盘的原理图进行查看）所示：

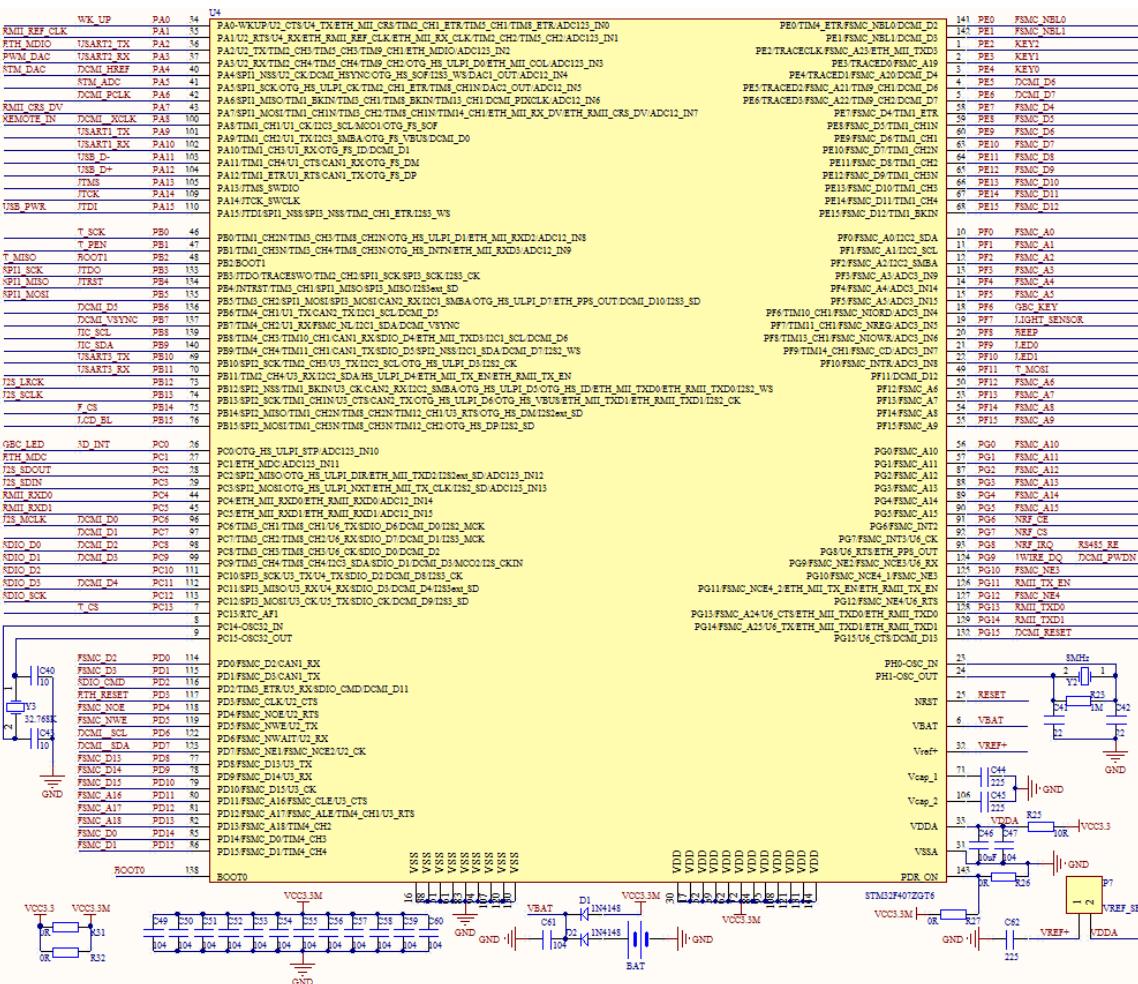


图 2.1.1.1 MCU 部分原理图

上图中 U4 为我们的主芯片：STM32F407ZGT6。

这里主要讲解以下 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 R31 和 R32 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 Vref+ 即可。

2.1.2 引出 IO 口

ALIENTEK 探索者 STM32F4 开发板引出了 STM32F407ZGT6 的所有 IO 口，如图 2.1.2.1 所示：

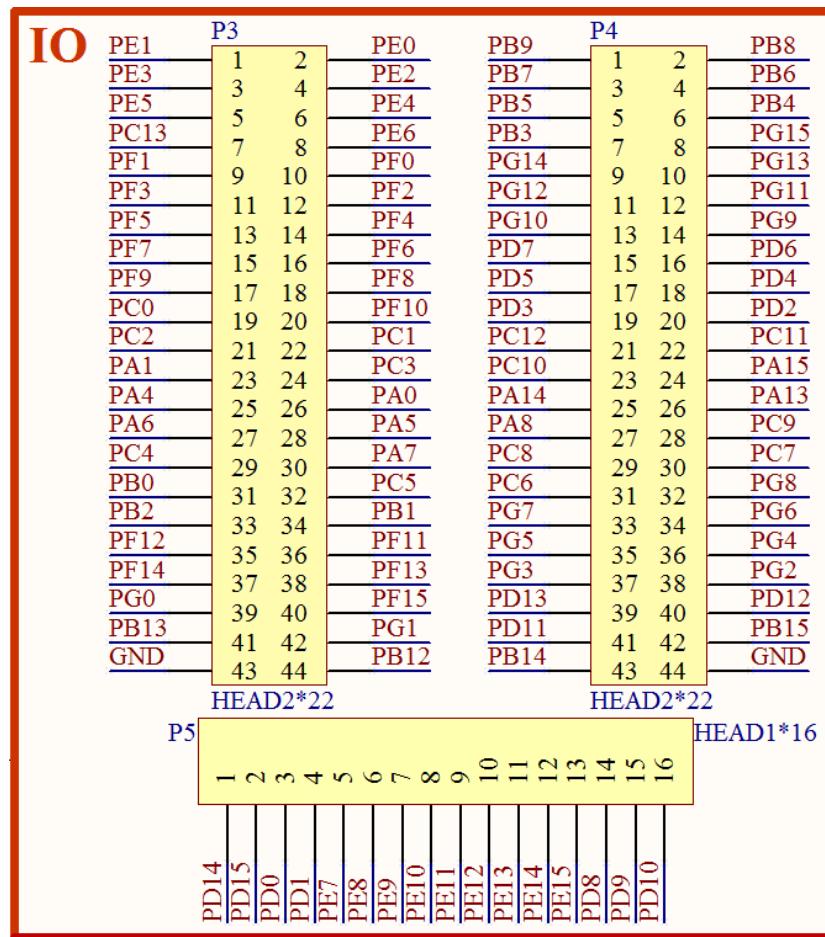


图 2.1.2.1 引出 IO 口

图中 P3、P4 和 P5 为 MCU 主 IO 引出口, 这三组排针共引出了 102 个 IO 口, STM32F407ZGT6 总共有 112 个 IO, 除去 RTC 晶振占用的 2 个, 还剩 110 个, 这三组主引出排针, 总共引出了 102 个 IO, 剩下的 8 个 IO 口分别通过: P6 (PA9&PA10)、P9 (PA2&PA3)、P10 (PB10&PB11) 和 P11 (PA11&PA12) 等 4 组排针引出。

2.1.3 USB 串口/串口 1 选择接口

ALIENTEK 探索者 STM32F4 开发板板载的 USB 串口和 STM32F407ZGT6 的串口是通过 P6 连接起来的, 如图 2.1.3.1 所示:

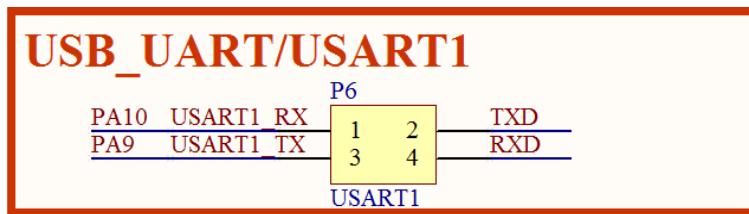


图 2.3.1.1 USB 串口/串口 1 选择接口

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 STM32F407ZGT6 来说的。这样, 通过对接, 就可以实现 USB 串口和 STM32F407ZGT6 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 转串口用了。

2.1.4 JTAG/SWD

ALIENTEK 探索者 STM32F4 开发板板载的标准 20 针 JTAG/SWD 接口电路如图 2.1.4.1 所示：

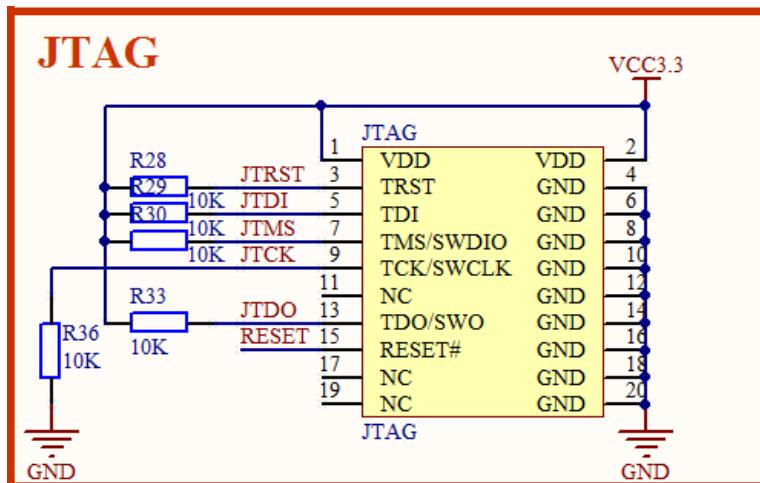


图 2.1.4.1 JTAG/SWD 接口

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。

特别提醒，JTAG 有几个信号线用来接其他外设了，但是 SWD 是完全没有接任何其他外设的，所以在使用的时候，**推荐大家一律使用 SWD 模式!!!**

2.1.5 SRAM

ALIENTEK 探索者 STM32F4 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（特别提醒：地址线可以乱，但是数据线必须一致!!），因此，该原理图对这两个芯片都是可以正常使用的。

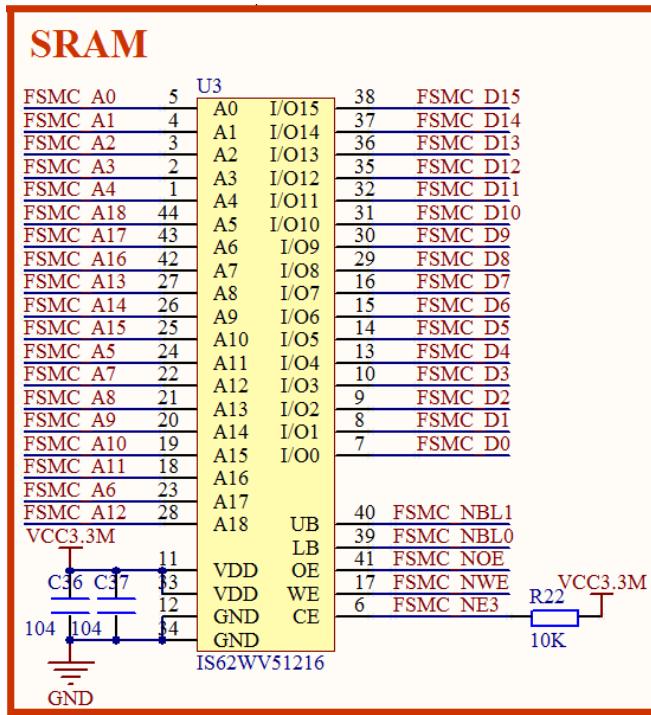


图 2.1.5.1 外扩 SRAM

图中 U3 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(芯片本身有192K字节),从而在需要大内存的场合,探索者STM32F4开发板也可以胜任。

2.1.6 LCD 模块接口

ALIENTEK 探索者 STM32F4 开发板板载的 LCD 模块接口电路如图 2.1.6.1 所示:

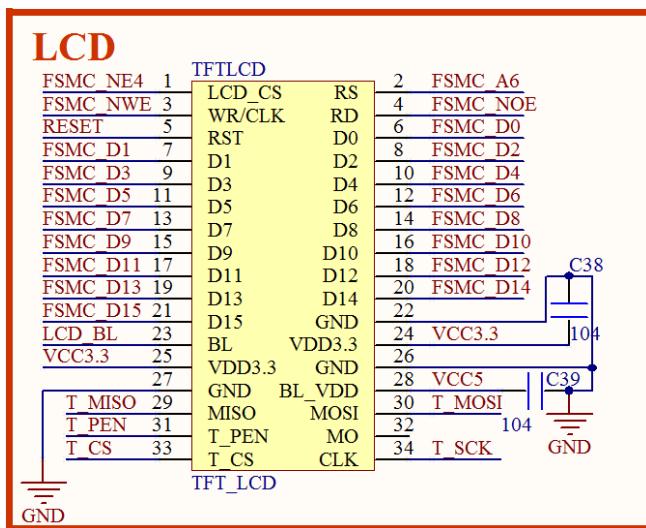


图 2.1.6.1 LCD 模块接口

图中 TFT_LCD 是一个通用的液晶模块接口,支持 ALIENTEK 全系列 TFTLCD 模块,包括:2.4 寸、2.8 寸、3.5 寸、4.3 寸和 7 寸等尺寸的 TFTLCD 模块。LCD 接口连接在 STM32F407ZGT6 的FSMC总线上面,可以显著提高LCD的刷屏速度。

图中的 T_MISO/T_MOSI/T_PEN/T_CS/T_CS 用来实现对液晶触摸屏的控制(支持电阻屏和

电容屏)。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 探索者 STM32F4 开发板的复位电路如图 2.1.7.1 所示:

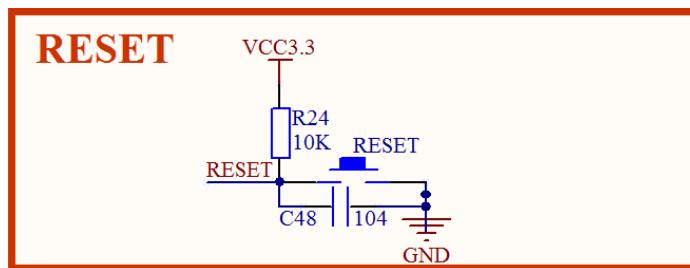


图 2.1.7.1 复位电路

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 R24 和 C48 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 探索者 STM32F4 开发板的启动模式设置端口电路如图 2.1.8.1 所示:

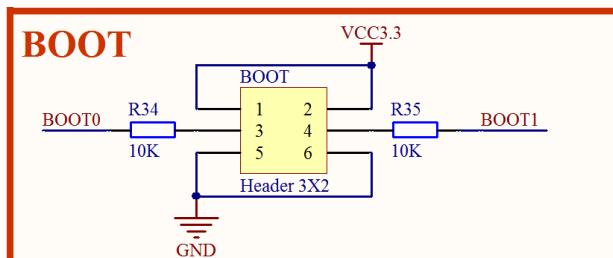


图 2.1.8.1 启动模式设置接口

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 2.1.8.1 BOOT0、BOOT1 启动模式表

按照表 2.1.8.1, 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 探索者 STM32F4 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 探索者 STM32F4 开发板板载了一公一母两个 RS232 接口, 电路原理图如图 2.1.9.1 所示:

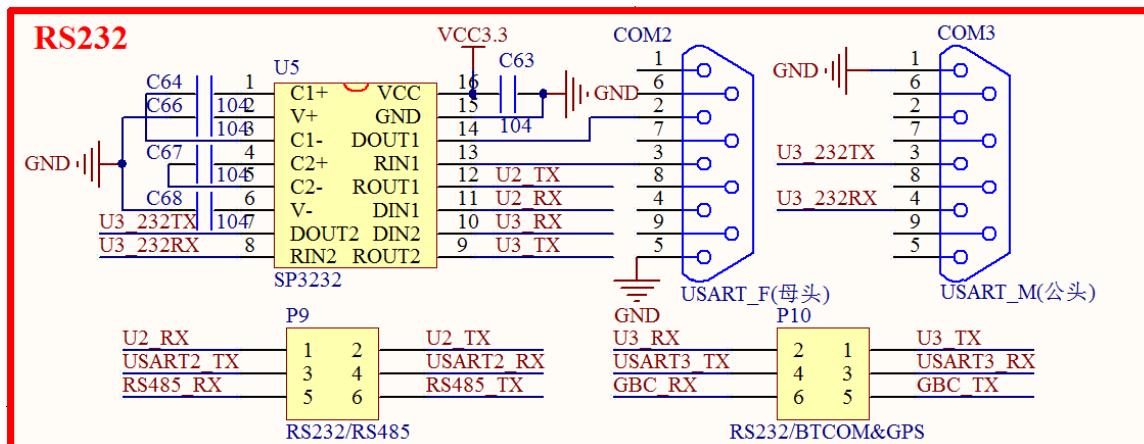


图 2.1.9.1 RS232 串口

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(COM2)/RS485 的选择，P10 用来实现 RS232(COM3)/ATK 模块接口的选择，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(COM2)/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。

而图中的 USART3_TX/USART3_RX 则是连接在 MCU 的串口 3 上（PB10/PB11），所以 RS232(COM3)/ATK 模块接口都是通过串口 3 来实现的。图中 GBC_RX 和 GBC_TX 连接在 ATK 模块接口 U7 上面。

因为 P9/P10 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，通过连接探索者 STM32F4 开发板的 P9/P10 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 探索者 STM32F4 开发板板载的 RS485 接口电路如图 2.1.10.1 所示：

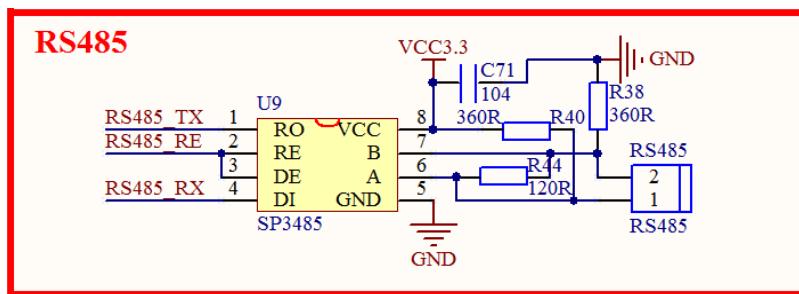


图 2.1.10.1 RS485 接口

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 R44 为终端匹配电阻，而 R38 和 R40，则是两个偏置电阻，以保证静默状态时，485 总线维持逻辑 1。

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（PG8）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）。

另外，**特别注意**：RS485_RX 和 NRF_IRQ 共同接在 PG8 上面，在同时用到这两个外设的时候，需要注意下。

2.1.11 CAN/USB 接口

ALIENTEK 探索者 STM32F4 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：

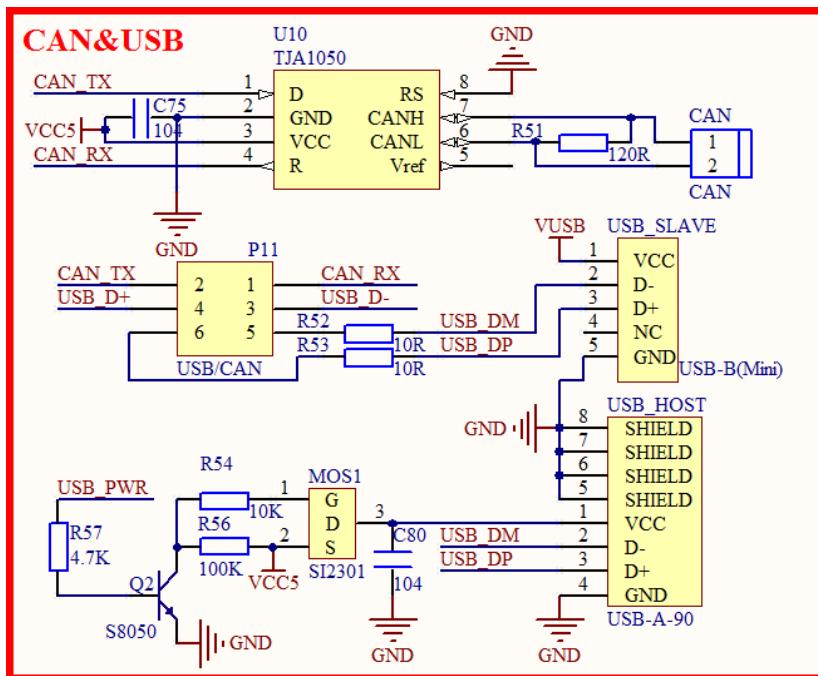


图 2.1.11.1 CAN/USB 接口

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 R51 为终端匹配电阻。

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 P11 来选择使用 USB 还是 CAN。

图中共有 2 个 USB 口：USB_SLAVE 和 USB_HOST，前者是用来做 USB 从机通信的，后者则是用来做 USB 主机通信的。

USB_SLAVE 可以用来连接电脑，实现 USB 读卡器或声卡等 USB 从机实验。另外，该接口还具有供电功能，VUSB 为开发板的 USB 供电电压，通过这个 USB 口，就可以给整个开发板供电了。

USB_HOST 可以用来接如：U 盘、USB 鼠标、USB 键盘和 USB 手柄等设备，实现 USB 主机功能。该接口可以对从设备供电，且供电可控，通过 USB_PWR 控制，该信号连接在 MCU 的 PA15 引脚上与 JTDI 共用 PA15，所以用 JTAG 仿真的时候，USB_PWR 就不受控了，这也是我们**推荐大家使用 SWD 模式**而不用 JTAG 模式的另外一个原因。

2.1.12 EEPROM

ALIENTEK 探索者 STM32F4 开发板板载的 EEPROM 电路如图 2.1.12.1 所示：

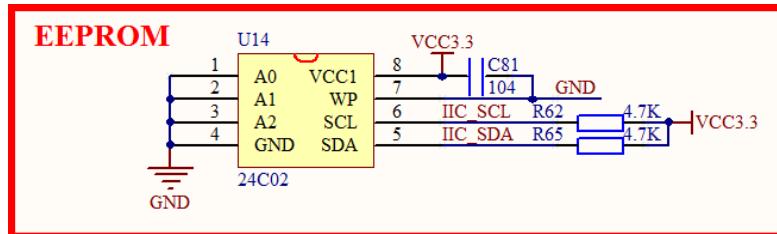


图 2.1.12.1 EEPROM

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 PB8 上，IIC_SDA 接在 MCU 的 PB9 上，这里我们虽然接到了 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、MPU6050 和 WM8978，后续我们将向大家介绍另外两个器件。

2.1.13 光敏传感器

ALIENTEK 探索者 STM32F4 开发板板载了一个光敏传感器，可以用来感应周围光线的变化，该部分电路如图 2.1.13.1 所示：

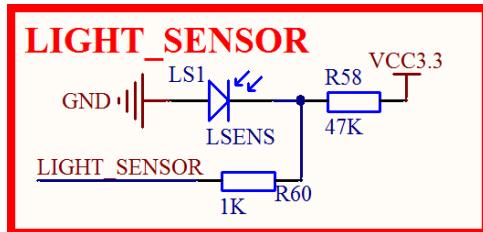


图 2.1.13.1 光敏传感器电路

图中的 LS1 就是光敏传感器，其实就是一个光敏二极管，周围环境越亮，电流越大，反之电流越小，即可等效为一个电阻，环境越亮阻值越小，反之越大，从而通过读取 LIGHT_SENSOR 的电压，即可知道周围环境光线强弱。LIGHT_SENSOR 连接在 MCU 的 ADC3_IN5 (ADC3 通道 5) 上面，即 PF7 引脚。

2.1.14 SPI FLASH

ALIENTEK 探索者 STM32F4 开发板板载的 SPI FLASH 电路如图 2.1.14.1 所示：

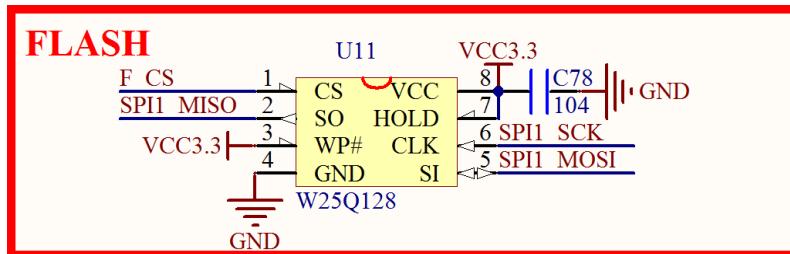


图 2.1.14.1 SPI FLASH 芯片

SPI FLASH 芯片型号为 W25Q128，该芯片的容量为 128Mb，也就是 16M 字节。该芯片和 NRF24L01 共用一个 SPI (SPI1)，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，

请务必禁止另外一个器件的片选信号。

图中 F_CS 连接在 MCU 的 PB14 上, SPI1_SCK/SPI1_MOSI/SPI1_MISO 则分别连接在 MCU 的 PB3/PB5/PB4 上, 其中 PB3/PB4 又是 JTAG 的 JTDO 和 JTRST 信号, 所以在 JTAG 仿真的时候, SPI 就用不了了, 但是用 SWD 仿真, 则不存在任何问题, 所以我们**推荐大家使用 SWD 仿真!**

2.1.15 六轴加速度传感器

ALIENTEK 探索者 STM32F4 开发板板载的六轴加速度传感器电路如图 2.1.15.1 所示:

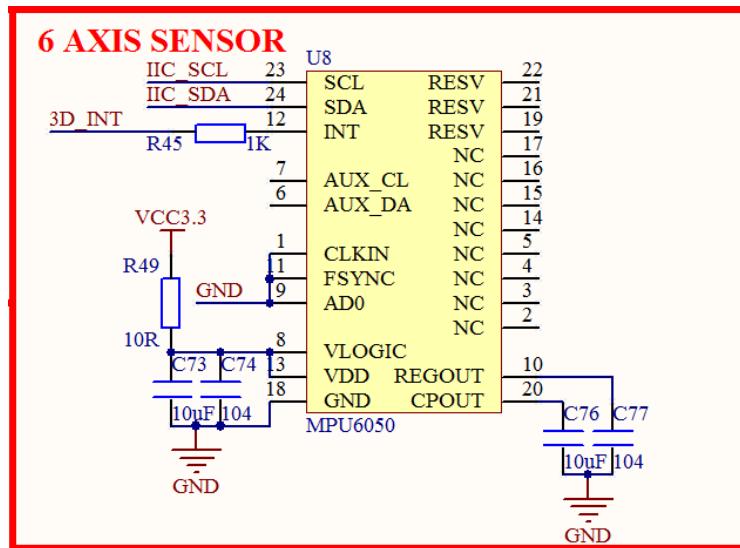


图 2.1.15.1 3D 加速度传感器

六轴加速度传感器芯片型号为: MPU6050, 该芯片内部集成一个三轴加速度传感器和一个三轴陀螺仪, 并且自带 DMP(Digital Motion Processor), 该传感器可以用于四轴飞行器的姿态控制和解算。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 PB8 和 PB9 上, 他们共享一个 IIC 总线。

2.1.16 温湿度传感器接口

ALIENTEK 探索者 STM32F4 开发板板载的温湿度传感器接口电路如图 2.1.16.1 所示:

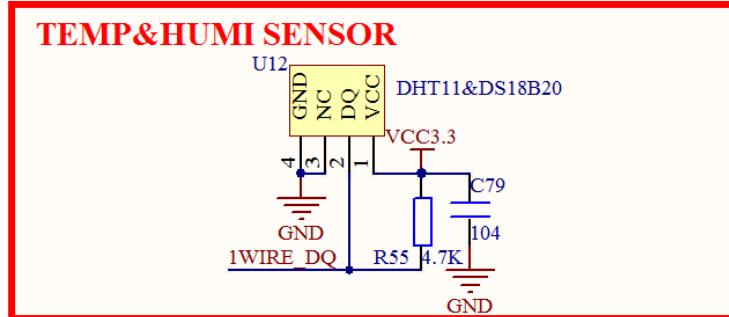


图 2.1.16.1 温湿度传感器接口

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 PG9 上, **特别注意:** 该引脚同时还接到了摄像头模块的 DCMI_PWDN 信号上面, 他们不能同时使用, 但可以分时复用。

2.1.17 红外接收头

ALIENTEK 探索者 STM32F4 开发板板载的红外接收头电路如图 2.1.17.1 所示：

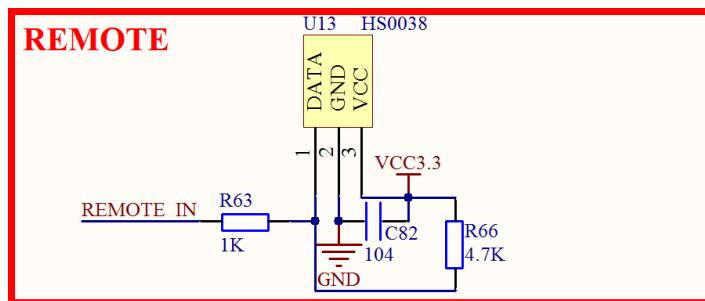


图 2.1.17.1 红外接收头

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 PA8 上。**特别注意：**PA8 同时连接了 DCMI_XCLK，如果要用到 DCMI_XCLK 的时候，HS0038 就不能同时使用了，但可以分时复用。

2.1.18 无线模块接口

ALIENTEK 探索者 STM32F4 开发板板载的无线模块接口电路如图 2.1.18.1 所示：

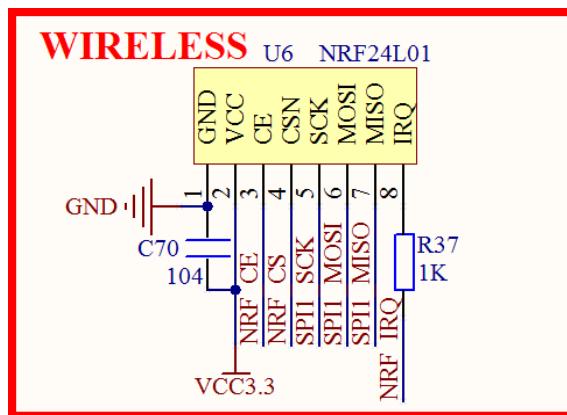


图 2.1.18.1 无线模块接口

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，接 MCU 的 SPI1。这里**需要注意**的是 PG8 还接了 RS485 的 RE 信号，所以在使用 NRF24L01 中断引脚的时候，不能和 RS485 同时使用，不过，如果没用到 NRF24L01 的中断引脚，那么 RS485 和 NRF24L01 模块就可以同时使用了。

2.1.19 LED

ALIENTEK 探索者 STM32F4 开发板板载总共有 3 个 LED，其原理图如图 2.1.19.1 所示：

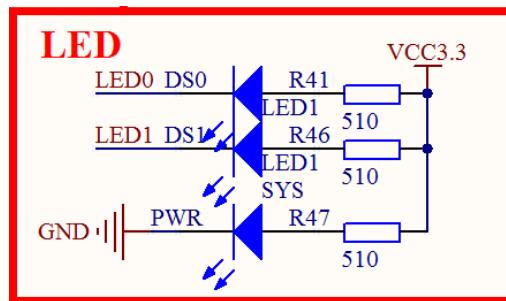


图 2.1.19.1 LED

其中 PWR 是系统电源指示灯，为蓝色。LED0(DS0)和 LED1(DS1)分别接在 PF9 和 PF10 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 探索者 STM32F4 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：

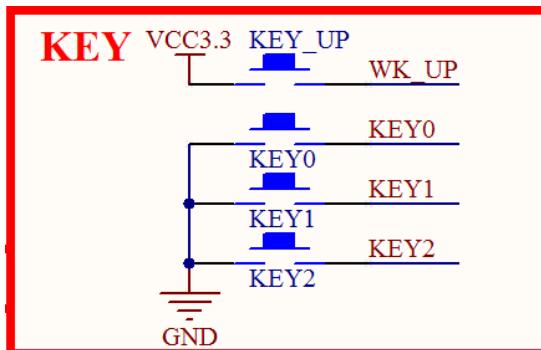


图 2.1.20.1 输入按键

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。

KEY_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。注意：这个按键是高电平触发的。

2.1.21 TPAD 电容触摸按键

ALIENTEK 探索者 STM32F4 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：

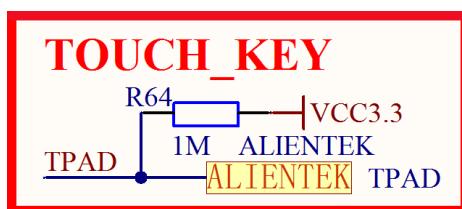


图 2.1.21.1 电容触摸按键

图中 1M 电阻是电容充电电阻，TPAD 并没有直接连接在 MCU 上，而是连接在多功能端口 (P12) 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 2.1.26 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 OLED/摄像头模块接口

ALIENTEK 探索者 STM32F4 开发板板载了一个 OLED/摄像头模块接口，其原理图如图 2.1.22.1 所示：

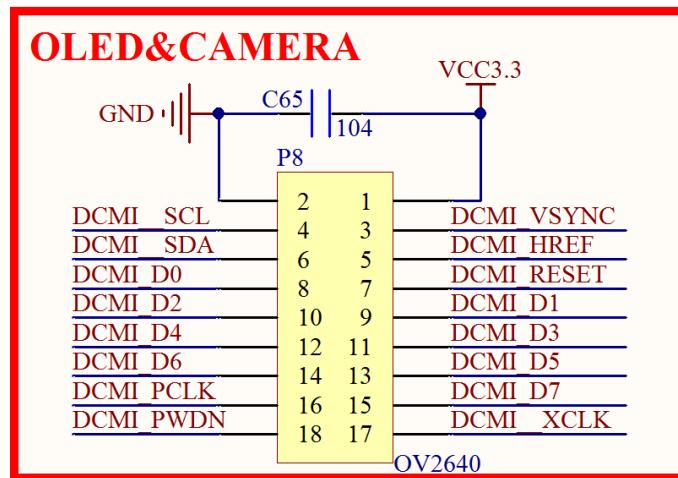


图 2.1.22.1 OLED/摄像头模块接口

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 DCMI_PWDN 和 DCMI_XCLK 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，DCMI_SCL/DCMI_SDA/DCMI_RESET/DCMI_PWDN/DCMI_XCLK 这 5 个信号是不属于 STM32F4 硬件摄像头接口的信号，通过普通 IO 控制即可，分别接在 MCU 的：PD6/PD7/PG15/PG9/PA8 上面。**特别注意：**DCMI_PWDN 和 1WIRE_DQ 信号共用 PG9 这个 IO，所以摄像头和 DS18B20/DHT11 不能同时使用，但是可以分时复用。另外，DCMI_XCLK 和 REMOTE_IN 共用，在用到 DCMI_XCLK 信号的时候，则红外接收和摄像头不可同时使用，不过同样是可以分时复用的。

其他信号全接在 STM32F4 的硬件摄像头接口上，DCMI_VSYNC/DCMI_HREF/DCMI_D0/DCMI_D1/DCMI_D2/DCMI_D3/DCMI_D4/DCMI_D5/DCMI_D6/DCMI_D7/DCMI_PCLK 分别连接在：PB7/PA4/PC6/PC7/PC8/PC9/PC11/PB6/PE5/PE6/PA6 上。**特别注意：**这些信号和 DAC1 输出以及 SD 卡，I2S 音频等有 IO 共用，所以在使用 OLED 模块或摄像头模块的时候，不能和 DAC1 的输出、SD 卡使用和 I2S 音频播放等三个功能同时使用，只能分时复用。

2.1.23 有源蜂鸣器

ALIENTEK 探索者 STM32F4 开发板板载了一个有源蜂鸣器，其原理图如图 2.1.23.1 所示：

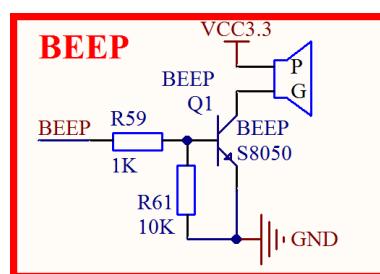


图 2.1.23.1 有源蜂鸣器

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，R61 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 PF8 上面，PF8 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。

2.1.24 SD 卡接口

ALIENTEK 探索者 STM32F4 开发板板载了一个 SD 卡（大卡/相机卡）接口，其原理图如图 2.1.24.1 所示：

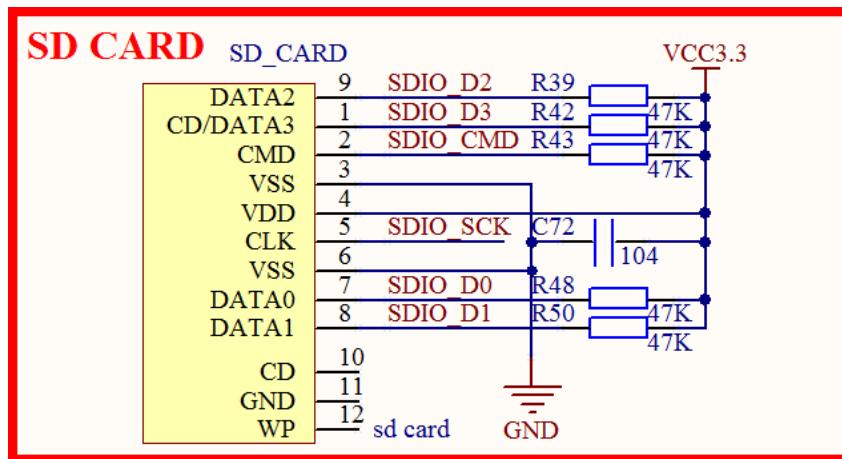


图 2.1.24.1 SD 卡/以太网接口

图中 SD_CARD 为 SD 卡接口，该接口在开发板的底面，这也是探索者 STM32F4 开发板底面唯一的元器件。

SD 卡采用 4 位 SDIO 方式驱动，理论上最大速度可以达到 24MB/S，非常适合需要高速存储的情况。图中：SDIO_D0/SDIO_D1/SDIO_D2/SDIO_D3/SDIO_SCK/SDIO_CMD 分别连接在 MCU 的 PC8/PC9/PC10/PC11/PC12/PD2 上面。**特别注意：**SDIO 和 OLED/摄像头的部分 IO 有共用，所以在使用 OLED 模块或摄像头模块的时候，只能和 SDIO 分时复用，不能同时使用。

2.1.25 ATK 模块接口

ALIENTEK 探索者 STM32F4 开发板板载了 ATK 模块接口，其原理图如图 2.1.25.1 所示：

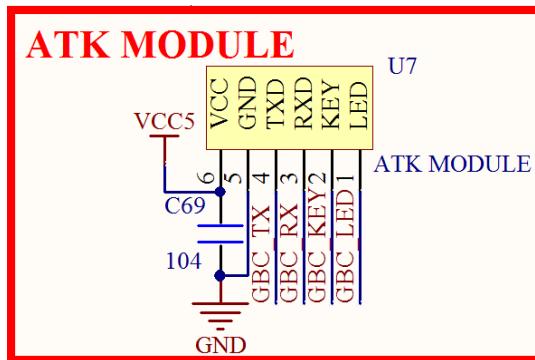


图 2.1.25.1 ATK 模块接口

如图所示，U7 是一个 1*6 的排座，可以用来连接 ALIENTEK 推出的一些模块，比如：蓝牙串口模块、GPS 模块等。有了这个接口，我们连接模块就非常简单，插上即可工作。

图中：GBC_TX/GBC_RX 可通过 P10 排针，选择接入 PB11/PB10（即串口 3），详见 2.1.9

节。而 GBC_KEY 和 GBC_LED 则分别连接在 MCU 的 PF6 和 PC0 上面。**特别注意：**GBC_LED 和 3D_INT 共用 PC0，所以同时使用 ATK 模块接口和 MPU6050 的时候，要注意这个 IO 的设置。

2.1.26 多功能端口

ALIENTEK 探索者 STM32F4 开发板板载的多功能端口，是由 P12 和 P2 构成的一个 6PIN 端口，其原理图如图 2.1.26.1 所示：

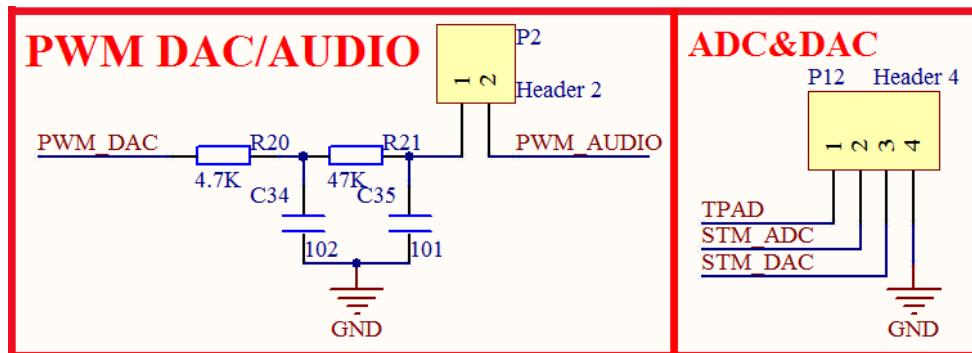


图 2.1.26.1 多功能端口

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 P12，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 PA5 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。**特别注意：**STM_DAC 与摄像头的 DCMI_HREF 共用 PA4，所以他们不可以同时使用，但是可以分时复用。

我们再来看看 P2，PWM_DAC 连接在 MCU 的 PA3，是定时器 2/5 的通道 4 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 WM8978 的 AUX 输入，可通过配置 WM8978，输出到耳机/扬声器。**特别注意：**PWM_DAC 和 USART2_RX 共用 PA3，所以 PWM_DAC 和串口 2 的接收，不可以同时使用，不过，可以分时复用。

单独介绍完了 P12 和 P2，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：

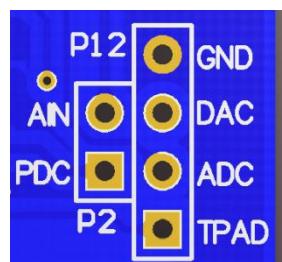


图 2.1.26.2 组合后的多功能端口

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 以太网接口（RJ45）

ALIENTEK 探索者 STM32F4 开发板板载了一个以太网接口(RJ45)，其原理图如图 2.1.27.1 所示：

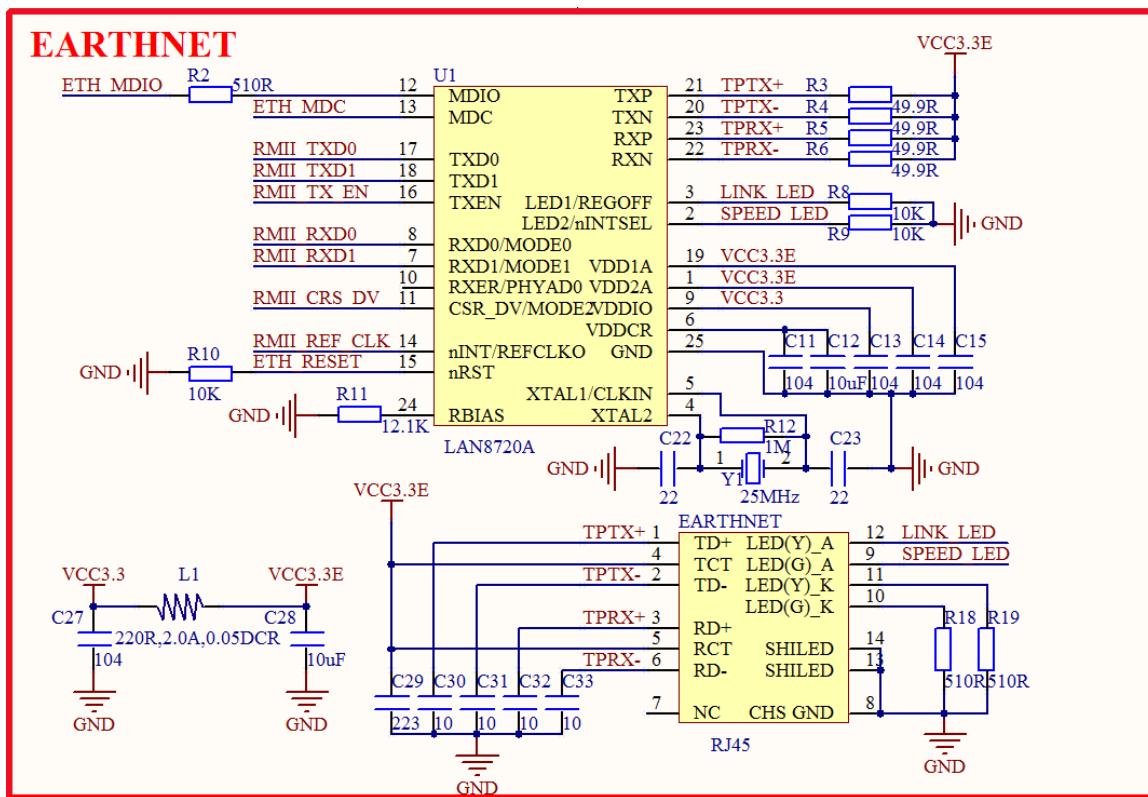


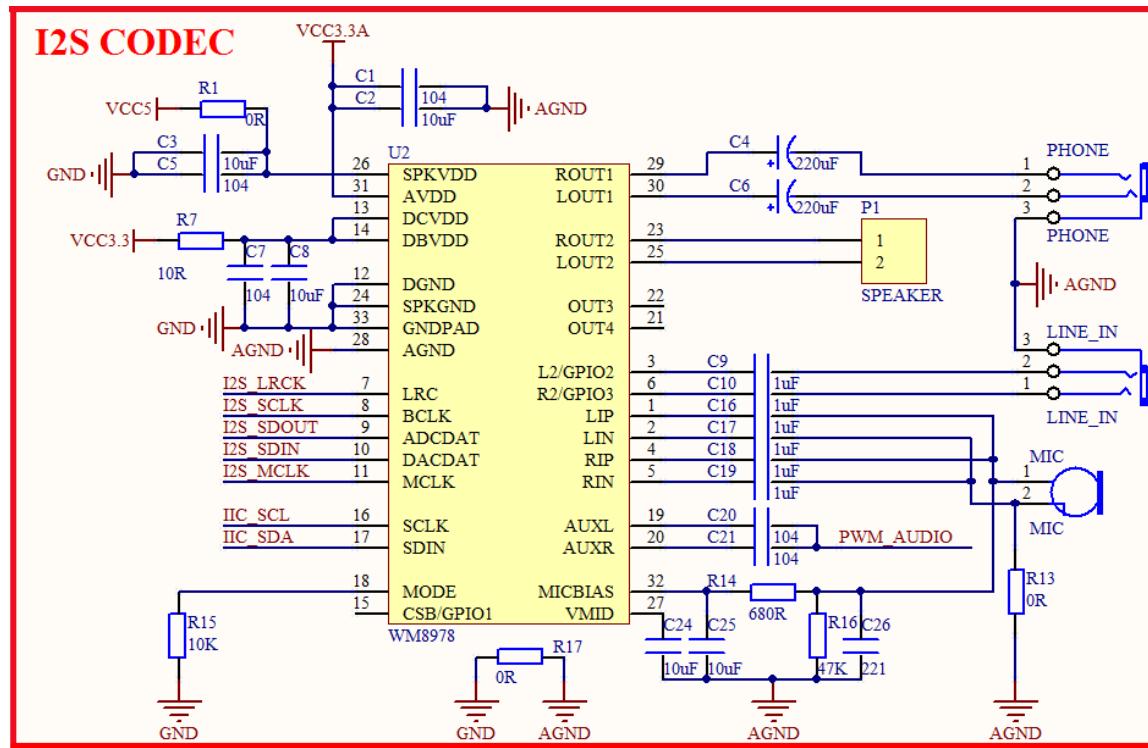
图 2.1.27.1 以太网接口电路

STM32F4 内部自带网络 MAC 控制器，所以只需要外加一个 PHY 芯片，即可实现网络通信功能。这里我们选择的是 LAN8720A 这颗芯片作为 STM32F4 的 PHY 芯片，该芯片采用 RMII 接口与 STM32F4 通信，占用 IO 较少，且支持 auto mdix (即可自动识别交叉/直连网线) 功能。板载一个自带网络变压器的 RJ45 头 (HR91105A)，一起组成一个 10M/100M 自适应网卡。

图中：ETH_MDIO/ETH_MDC/RMII_TXD0/RMII_TXD1/RMII_TX_EN/RMII_RXD0/RMII_RXD1/RMII_CRS_DV/RMII_REF_CLK/ETH_RESET 分别接在 MCU 的：PA2/PC1/PG13/PG14/PG11/PC4/PC5/PA7/PA1/PD3 上。**特别注意：**网络部分 ETH_MDIO 与 USART2_TX 共用 PA2，所以网络和串口 2 的发送，不可以同时使用，但是可以分时复用。

2.1.28 I2S 音频编解码器

ALIENTEK 探索者 STM32F4 开发板板载 WM8978 高性能音频编解码芯片，其原理图如图 2.1.28.1 所示：



2.1.28.1 I2S 音频编解码芯片

WM8978 是一颗低功耗、高性能的立体声多媒体数字信号编解码器。该芯片内部集成了 24 位高性能 DAC&ADC，可以播放最高 192K@24bit 的音频信号，并且自带段 EQ 调节，支持 3D 音效等功能。不仅如此，该芯片还结合了立体声差分麦克风的前置放大与扬声器、耳机和差分、立体声线输出的驱动，减少了应用时必需的外部组件，直接可以驱动耳机（ $16\Omega @40mW$ ）和喇叭（ $8\Omega /0.9W$ ），无需外加功放电路。

图中，P1 是扬声器接口，可以用来外接 1W 左右的扬声器。MIC 是板载的咪头，可用于录音机实验，实现录音。PHONE 是 3.5mm 耳机输出接口，可以用来插耳机。LINE_IN 则是线路输入接口，可以用来外接线路输入，实现立体声录音。

该芯片采用 I2S 接口与 MCU 连接，图中：I2S_LRCK/I2S_SCLK/I2S_SDOUT/I2S_SDIN /I2S_MCLK/IIC_SCL/IIC_SDA 分别接在 MCU 的：PB12/PB13/PC2/PC3/PC6/PB8/PB9 上。**特别注意：**I2S_MCLK 和 DCMI_D0 共用 PC6，所以 I2S 音频播放和 OLED 模块/摄像头模块不可以同时使用。另外，IIC_SCL 和 IIC_SDA 是与 24C02/MPU6050 等共用一个 IIC 接口。

2.1.29 电源

ALIENTEK 探索者 STM32F4 开发板板载的电源供电部分，其原理图如图 2.1.29.1 所示：

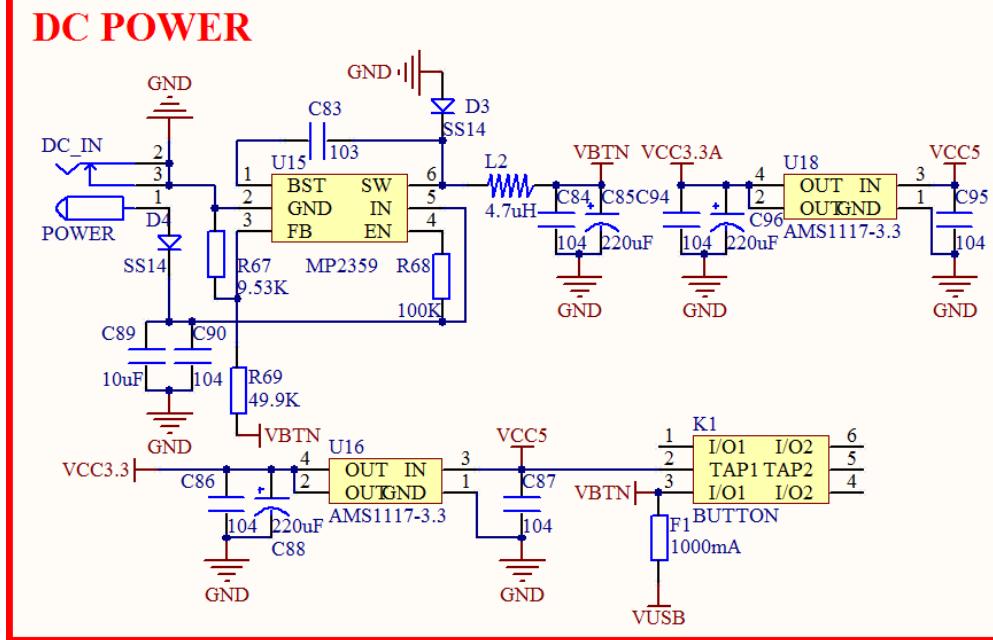


图 2.1.29.1 电源

图中，总共有 3 个稳压芯片：U15/U16/U18，DC_IN 用于外部直流电源输入，经过 U15 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 1000ma 自恢复保险丝，用于保护 USB。U16 和 U18 均为 3.3V 稳压芯片，给开发板提供 3.3V 电源，其中 U16 输出的 3.3V 给数字部分用，U18 输出的 3.3V 给模拟部分（WM8978）使用，分开供电，以得到最佳音质。

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。

2.1.30 电源输入输出接口

ALIENTEK 探索者 STM32F4 开发板板载了两组简单电源输入输出接口，其原理图如图 2.1.30.1 所示：

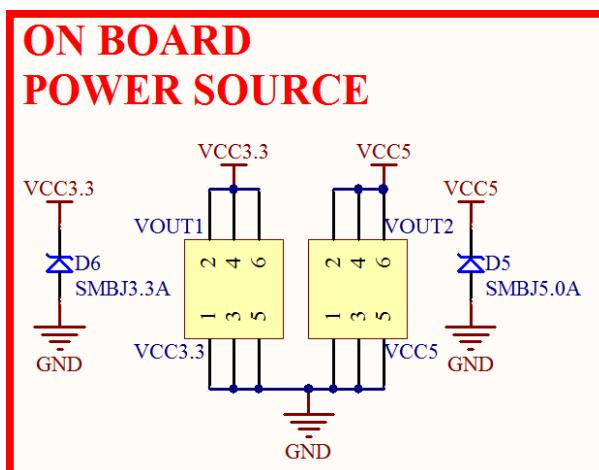


图 2.1.30.1 电源

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 1000ma），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。

图中 D5 和 D6 为 TVS 管，可以有效避免 VOUT 外接电源/负载不稳的时候（尤其是开发板外接电机/继电器/电磁阀等感性负载的时候），对开发板造成的损坏。同时还能一定程度防止外接电源接反，对开发板造成的损坏。

2.1.31 USB 串口

ALIENTEK 探索者 STM32F4 开发板板载了一个 USB 串口，其原理图如图 2.1.31.1 所示：

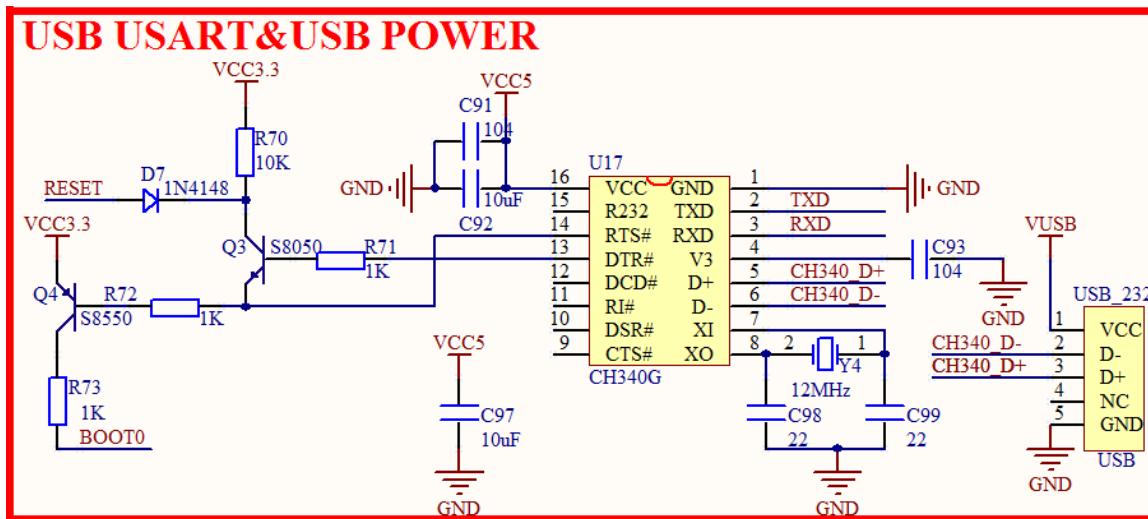


图 2.1.31.1 USB 串口

USB 转串口，我们选择的是 CH340G，是国内芯片公司南京沁恒的产品，稳定性经测试还不错，所以还是多支持下国产。

图中 Q3 和 Q4 的组合构成了我们开发板的一键下载电路，**只需要在 flymcu 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。

一键下载电路的具体实现过程：首先，mcuisp 控制 DTR 输出低电平，则 DTR_N 输出高，然后 RTS 置高，则 RTS_N 输出低，这样 Q4 导通了，BOOT0 被拉高，即实现设置 BOOT0 为 1，同时 Q3 也会导通，STM32F4 的复位脚被拉低，实现复位。然后，延时 100ms 后，mcuisp 控制 DTR 为高电平，则 DTR_N 输出低电平，RTS 维持高电平，则 RTS_N 继续为低电平，此时 STM32F4 的复位引脚，由于 Q3 不再导通，变为高电平，STM32F4 结束复位，但是 BOOT0 还是维持为 1，从而进入 ISP 模式，接着 mcuisp 就可以开始连接 STM32F4，下载代码了，从而实现一键下载。

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 探索者 STM32F4 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, 1 个 USB 供电最多 500mA, 且由于导线电阻存在, 供到开发板的电压, 一般都不会有 5V, 如果使用了很多大负载外设, 比如 4.3 寸屏、网络、摄像头模块等, 那么可能引起 USB 供电不够, 所以如果是使用 4.3 屏的朋友, 或者同时用到多个模块的时候, **建议大家使用一个独立电源供电**。如果没有独立电源, 建议可以同时插 2 个 USB 口, 并插上 JTAG, 这样供电可以更足一些。
- 3, JTAG 接口有几个信号(JTDI/JTDO/JTRST)被 USB_PWR(USB HOST)/SPI1(W25Q128 和 NRF24L01) 占用了, 所以在调试这几个模块的时候, 请大家选择 SWD 模式, 其实 **最好就是一直用 SWD 模式**。
- 4, 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 PF8 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。
- 5, 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。
- 6, 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。
- 7, 开发板的 USB SLAVE 和 USB HOST 共用同一个 USB 口, 所以, 他们不可以同时使用。使用的时候多加注意。

至此, 本手册的实验平台 (ALIENTEK 探索者 STM32F4 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。

2.3 STM32F4 学习方法

为 STM32F4 作为目前最热门的 ARM Cortex M4 处理器, 由于其强大的功能, 可替代 DSP 等特性, 正在被越来越多的公司选择使用。学习 STM32F4 的朋友也越来越多, 初学者, 可能会认为 STM32F4 很难学, 以前可能只学过 51, 或者甚至连 51 都没学过的, 一看到 STM32F4 那么多寄存器, 就懵了。其实, 万事开头难, 只要掌握了方法, 学好 STM32F4, 还是非常简单的, 这里我们总结学习 STM32F4 的几个要点:

1, 一款实用的开发板。

这个是实验的基础, 有个开发板在手, 什么东西都可以直观的看到。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, 三本参考资料, 即《STM32F4xx 中文参考手册》、《STM32F3 与 F4 系列 Cortex M4 内核编程手册》和《Cortex M3 与 M4 权威指南》。

《STM32F4xx 中文参考手册》是 ST 出的官方资料, 有 STM32F4 的详细介绍, 包括了 STM32F4 的各种寄存器定义以及功能等, 是学习 STM32F4 的必备资料之一。而《STM32F3 与 F4 系列 Cortex M4 内核编程手册》则是对《STM32F4xx 中文参考手册》的补充, 很多关于 Cortex M4 内核的介绍(寄存器等), 都可以在这个文档找到答案, 该文档同样是 ST 的官方资料, 专门针对 ST 的 Cortex M4 产品。最后, 《Cortex M3 与 M4 权威指南》则针对 Cortex M4 内核进行了详细介绍, 并配有简单实例, 对于想深入了解 Cortex M4 内核的朋友, 此文档是非常好的参考资料。

3，掌握方法，勤学善悟。

STM32F4 不是妖魔鬼怪，不要畏难，STM32F4 的学习和普通单片机一样，基本方法就是：

- a) 掌握时钟树图（见《STM32F4xx 中文参考手册》图 13）。

任何单片机，必定是靠时钟驱动的，时钟就是单片机的动力，STM32F4 也不例外，通过时钟树，我们可以知道，各种外设的时钟是怎么来的？有什么限制？从而理清思路，方便理解。

- b) 多思考，多动手。

所谓熟能生巧，先要熟，才能巧。如何熟悉？这就要靠大家自己动手，多多练习了，光看/说，是没什么太多用的，很多人问我，STM32F4 这么多寄存器，如何记得啊？回答是：不需要全部记住。学习 STM32F4，不是应试教育，不需要考试，不需要你倒背如流。你只需要知道这些寄存器，在哪个地方，用到的时候，可以迅速查找到，就可以了。完全是可以翻书，可以查资料的，可以抄袭的，不需要死记硬背。掌握学习的方法，远比掌握学习的内容重要的多。

熟悉了之后，就应该进一步思考，也就是所谓的巧了。我们提供了几十个例程，供大家学习，跟着例程走，无非就是熟悉 STM32F4 的过程，只有进一步思考，才能更好的掌握 STM32F4，也即所谓的举一反三。例程是死的，人是活的，所以，可以在例程的基础上，自由发挥，实现更多的其他功能，并总结规律，为以后的学习/使用打下坚实的基础，如此，方能信手拈来。

所以，学习一定要自己动手，光看视频，光看文档，是不行的。举个简单的例子，你看视频，教你如何煮饭，几分钟估计你就觉得学会了。实际上你可以自己测试下，是否真能煮好？

机会总是留给有准备的人，只有平时多做准备，才可能抓住机会。

只要以上三点做好了，学习 STM32F4 基本上就不会有什么太大问题了。如果遇到问题，可以在我们的技术论坛：开源电子网：www.openedv.com 提问，论坛 STM32 板块已经有 3W 多个主题，很多疑问已经有网友提过了，所以可以在论坛先搜索一下，很多时候，就可以直接找到答案了。论坛是一个分享交流的好地方，是一个可以让大家互相学习，互相提高的平台，所以有时间，可以多上去看看。

另外，很多 ST 官方发布的所有资料（芯片文档、用户手册、应用笔记、固件库、勘误手册等），大家都可以在 www.stmcu.org 这个地方下载到。也可以经常关注下，ST 会将最新的资料都放到这个网站上。

第二篇 软件篇

上一篇，我们介绍了本手册的实验平台，本篇我们将详细介绍 STM32F4 的开发软件：MDK5。通过该篇的学习，你将了解到：1、如何在 MDK5 下新建 STM32F4 工程；2、工程的编译；3、MDK5 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一个完整的 STM32F4 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 STM32F4 的开发流程，并能独立开始 STM32F4 的编程和学习。

本篇将分为如下 3 个章节：

- 2.1, MDK5 软件入门；
- 2.2, STM32F4 开发基础知识入门
- 2.3, SYSTEM 文件介绍；

第三章 MDK5 软件入门

本章将向大家介绍 MDK5 软件的使用，通过本章的学习，我们最终将建立一个自己的基于 STM32F40X 系列的 MDK5 工程，同时本章还将向大家介绍 MDK5 软件的一些使用技巧，希望大家在本章之后，能够对 MDK5 这个软件有个比较全面的了解。

本章分为如下个小结：

- 3.1, STM32F4 官方固件库简介
- 3.2, MDK5 简介；
- 3.3, 新建基于 STM32F4 固件库的 MDK5 工程；
- 3.4, 程序下载与调试
- 3.5, MDK5 使用技巧；

3.1 STM32 官方标准固件库简介

ST(意法半导体)为了方便用户开发程序，提供了一套丰富的 STM32F4 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 STM32F4 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。

固件库包光盘路径（是压缩包形式，大家解压即可）：

\8, STM32 参考资料\STM32F4xx 固件库\stm32f4_dsp_stdperiph_lib.zip
同时，大家也可以到我们开源电子网 <http://www.openedv.com> 下载。

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：

GPIOF->BSRRL=0x0001; //这里是针对 STM32F4 系列

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 BSRRL 寄存器实现电平控制，官方库封装了一个函数：

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    GPIOx->BSRRL = GPIO_Pin;
}
```

这个时候你不需要再直接去操作 BSRRL 寄存器了，你只需要知道怎么使用 GPIO_SetBits ()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，了解 STM32 各个外设的运行机制。只有了解了这些原理，你在进行固件库开发过程中才可能得心应手游刃有余。只有了解了原理，你才能做到“知其然知其所以然”，所以大家在学习库函数的同时，别忘了要了解一下寄存器大致配置过程。

3.1.2 STM32 固件库与 CMSIS 标准讲解

前一节我们讲到，STM32F4 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。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-M4 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-M4 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-M4 内核芯片的片上外设也会有很大的区别，比如 STM32F407 和 STM32F429，他们的片上外设就有很大的区别。

既然大家都使用的是 Cortex-M4 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-M4 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 应用程序基本结构如下图 3.1.2.2:

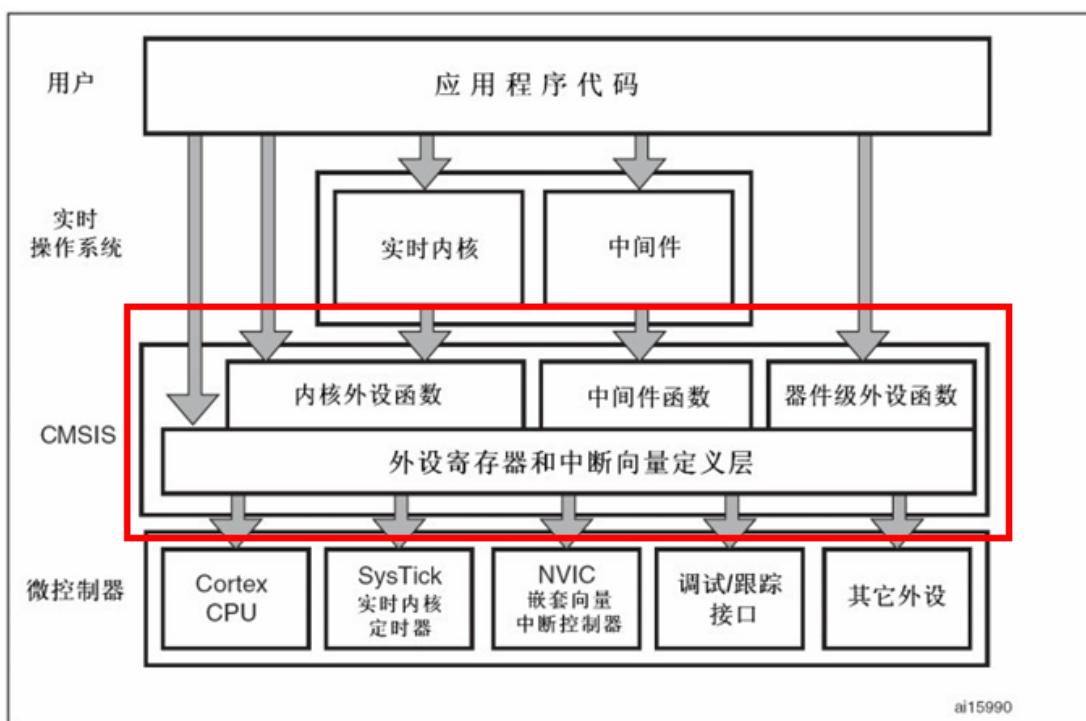


图 3.1.2.2 基于 CMSIS 应用程序基本结构

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。由 ARM 提供，芯片厂商根据需要更新。
- 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 STM32F4 官方库包介绍

这一节内容主要讲解 ST 官方提供的 STM32F4 固件库包的结构。ST 官方提供的固件库完整包可以在官方网站下载，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 V1.4 版本的固件库，大家可以到光盘目录找到其压缩文件：

[\8, STM32 参考资料\STM32F4xx 固件库\stm32f4_dsp_stdperiph_lib.zip](#)

然后解压即可。这在我们论坛有下载。下面看看官方库包的目录结构，如下图 3.1.3.1 和图 3.1.3.2 所示：



图 3.1.3.1 官方库包根目录

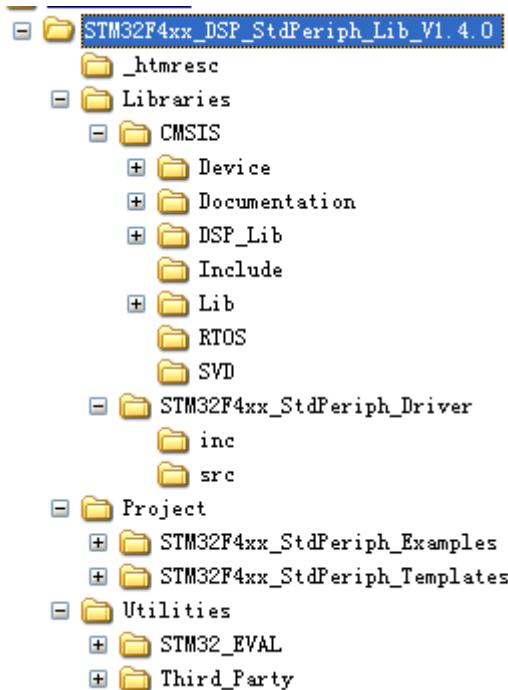


图 3.1.3.2 官方库目录列表

3.1.3.1 文件夹介绍:

Libraries 文件夹下面有 CMSIS 和 STM32F4xx_StdPeriph_Driver 两个目录，这两个目录包含固件核心的所有子文件夹和文件。

CMSIS 文件夹存放的是符合 CMSIS 规范的一些文件。包括 STM32F4 核内外设访问层代码，DSP 软件库，RTOS API，以及 STM32F4 片上外设访问层代码等。我们后面新建工程的时候会从这个文件夹复制一些文件到我们工程。

STM32F4xx_StdPeriph_Driver 放的是 STM32F4 标准外设固件库源码文件和对应的头文件。inc 目录存放的是 stm32f4xx_ppp.h 头文件，无需改动。src 目录下面放的是 stm32f4xx_ppp.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外设的关键文件，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，STM32F4xx_StdPeriph_Examples 文件夹下面存放的的 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。STM32F4xx_StdPeriph_Template 文件夹下面存放的是工程模板。

Utilities 文件夹就是官方评估板的一些对应源码，这个对于本手册学习可以忽略不看。

根目录中还有一个 stm32f4xx_dsp_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:

在介绍一些关键文件之前，首先我们来看看一个基于固件库的 STM32F4 工程需要哪些关键文件，这些文件之间有哪些关联关系。其实这个可以从 ST 提供的英文版的 STM32F4 固件库

说明里面找到。这里讲解的一些知识也是为我们后面章节“3.3.2 新建 STM32F4 工程模板”做铺垫。这些文件它们之间的关系如下图 3.1.3.2.1:

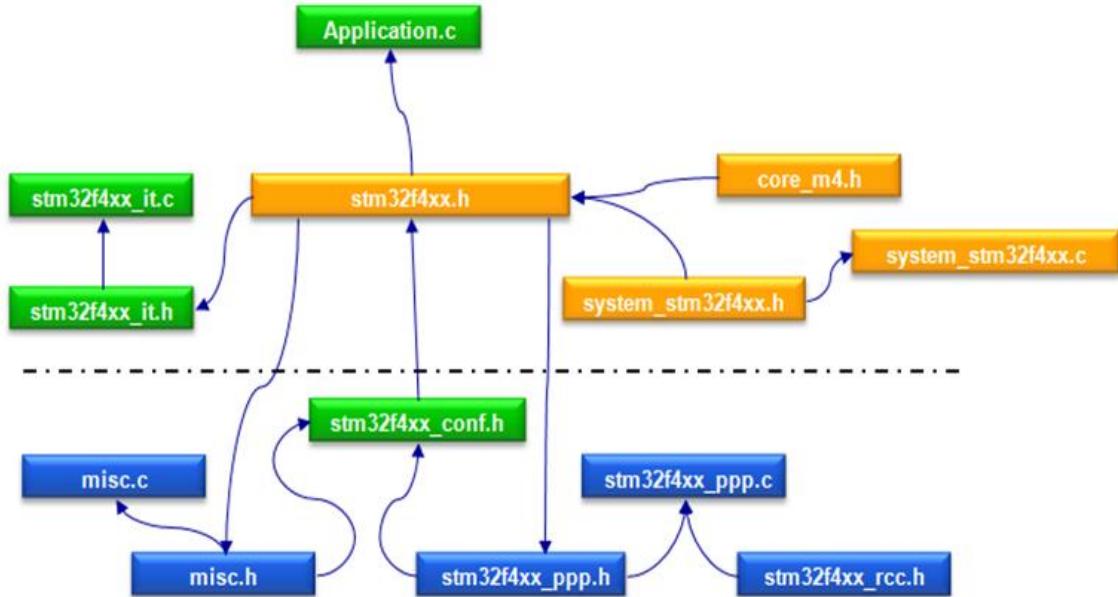


图 3.1.3.2.1 STM32F4 标准外设固件库文件关系图

core_cm4.h 文件位于 \STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Include 目录下面的，这个就是 CMSIS 核心文件，提供进入 M4 内核接口，这是 ARM 公司提供，对所有 CM4 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。

stm32f4xx.h 和 system_stm32f4xx.h 文件存放在文件夹 \STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Include 下面。
system_stm32f4xx.h 是片上外设接入层系统头文件。主要是申明设置系统及总线时钟相关的函数。与其对应的源文件 system_stm32f4xx.c 在目录 \STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Project\STM32F4xx_StdPeriph_Templates 可以找到。这个里面有一个非常重要的 SystemInit() 函数申明，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系统和总线时钟。

stm32f4xx.h 是 STM32F4 片上外设访问层头文件。这个文件就相当重要了，只要你做 STM32F4 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，对于这里是怎样的申明以及怎样将内存操作封装起来的，我们在后面的章节“4.6 MDK 中寄存器地址名称映射分析”中会讲到。同时该文件还包含了一些时钟相关的定义，FPU 和 MPU 单元开启定义，中断相关定义等等。

stm32f4xx_it.c,stm32f4xx_it.h 以及 stm32f4xx_conf.h 等文件，我们可以从 \STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Project\STM32F4xx_StdPeriph_Templates 文件夹中找到。这几个文件我们后面新建工程也有用到。stm32f4xx_it.c 和 stm32f4xx_it.h 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。

stm32f4xx_conf.h 是外设驱动配置文件。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。

对于上图中的 misc.c, misc.h, stm32f4xx_ppp.c, stm32f4xx_ppp.h 以及 stm32f4xx_rcc.c 和

stm32f4xx_rcc.h 文件，这些文件存放在目录 Libraries\STM32F4xx_StdPeriph_Driver。这些文件是 STM32F4 标准的外设库文件。其中 misc.c 和 misc.h 是定义中断优先级分组以及 Systick 定时器相关的函数。stm32f3xx_rcc.c 和 stm32f4xx_rcc.h 是与 RCC 相关的一些操作函数，作用主要是一些时钟的配置和使能。在任何一个 STM32 工程 RCC 相关的源文件和头文件是必须添加的。

对于文件 stm32f4xx_ppp.c 和 stm32f4xx_ppp.h，这就是 stm32F4 标准外设固件库对应的源文件和头文件。包括一些常用外设 GPIO,ADC,USART 等。

文件 Application.c 实际就是说是应用层代码。这个文件名称可以任意取了。我们工程中，直接取名为 main.c。

实际上一个完整的 STM32F4 的工程光有上面这些文件还是不够的。还缺少非常关键的启动文件。STM32F4 的启动文件存放在目录\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm 下面。对于不同型号的 STM32F4 系列对应的启动文件也不一样。我们的开发板是 STM32F407 系列所以我们选择的启动文件为 startup_stm32f40_41xxx.s。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  SystemInit
    IMPORT  __main

    LDR     R0, =SystemInit
    BLX     R0
    LDR     R0, =__main
    BX      R0
ENDP
```

这段代码的作用是在系统复位之后引导进入 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。

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 V1.4 版本固件库的工程模板。

3.2 MDK5 简介

MDK 源自德国的 KEIL 公司，是 RealView MDK 的简称。在全球 MDK 被超过 10 万的嵌入式开发工程师使用。目前最新版本为：MDK5.10，该版本使用 uVision5 IDE 集成开发环境，是目前针对 ARM 处理器，尤其是 Cortex M 内核处理器的最佳开发工具。

MDK5 向后兼容 MDK4 和 MDK3 等，以前的项目同样可以在 MDK5 上进行开发(但是头文件方面得全部自己添加)，MDK5 同时加强了针对 Cortex-M 微控制器开发的支持，并且对传统的开发模式和界面进行升级，MDK5 由两个部分组成：MDK Core 和 Software Packs。其中，Software Packs 可以独立于工具链进行新芯片支持和中间库的升级。如图 3.2.1 所示：

MDK-ARM Version 5

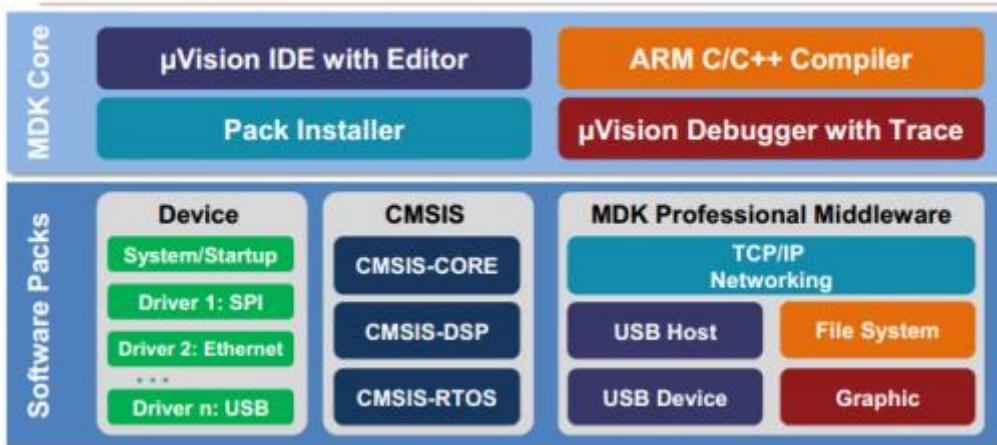


图 3.2.1 MDK5 组成

从上图可以看出，MDK Core 又分成四个部分：uVision IDE with Editor（编辑器），ARM C/C++ Compiler（编译器），Pack Installer（包安装器），uVision Debugger with Trace（调试跟踪器）。uVision IDE 从 MDK4.7 版本开始就加入了代码提示功能和语法动态检测等实用功能，相对于以往的 IDE 改进很大。

Software Packs（包安装器）又分为：Device（芯片支持），CMSIS（ARM Cortex 微控制器软件接口标准）和 Mdidleware（中间库）三个小部分，通过包安装器，我们可以安装最新的组件，从而支持新的器件、提供新的设备驱动库以及最新例程等，加速产品开发进度。

同以往的 MDK 不同，以往的 MDK 把所有组件都包含到了一个安装包里面，显得十分“笨重”，MDK5 则不一样，MDK Core 是一个独立的安装包，它并不包含器件支持、设备驱动、CMSIS 等组件，大小才 300M 左右，相对于 MDK4.70A 的 500 多 M，瘦身明显，MDK5 安装包可以在：<http://www.keil.com/demo/eval/arm.htm> 下载到。而器件支持、设备驱动、CMSIS 等组件，则可以点击 MDK5 的 Build Toolbar 的最后一个图标调出 Pack Installer，来进行各种组件的安装。也可以在 <http://www.keil.com/dd2/pack> 这个地址下载，然后进行安装。

最后，我们学习 STM32F407，还要安装两个包：ARM.CMSIS.4.1.1.pack（用于支持 ST 标准库，也就是所谓的库函数）和 Keil.STM32F4xx_DFP.1.0.8.pack（STM32F4 的器件库）。这两个包以及 MDK5.11a 的安装软件，我们都已经在开发板光盘提供了，跟安装软件在同一级目录，大家在按照 3.3.1 小节的步骤安装 MDK5 之后，点击这两个 pack 即可完成安装。

3.3 新建基于 STM32F40x 固件库的 MDK5 工程模板

在前面的章节我们介绍了 STM32F4 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) V1.4.0 固件库包：STM32F4xx_DSP_StdPeriph_Lib_V1.4.0 这是 ST 官网下载的固件库完整版，我们光盘目录（压缩包）：“\8, STM32 参考资料\STM32F4xx 固件库\stm32f4_dsp_stdperiph_lib.zip”。我们官方论坛开源电子网 www.openedv.com 也有下载。
- 2) MDK5 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\MDK5。
- 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\MDK5\keygen.exe。

3.3.1 MDK5 安装步骤

MDK5 的安装,请参考光盘:“**1, ALIENTEK 探索者 STM32F4 开发板入门资料\MDK5.11a 安装手册.pdf**”,里面详细介绍了 MDK5 的安装方法,本节我们将教大家如何新建一个基于固件库 STM32F4 的 MDK5 工程。这里需要特别说明一下,如果您使用过其他 mdk 或者 keil,请确保新的 mdk5.10 的安装路径跟以前的版本的 mdk 或者 keil 的安装路径不一样,同时安装路径不要包含中文,否则,就会出一些奇怪的错误。

3.3.2 新建工程模板

在新建之前,首先我们要说明一下,这一小节我们新建的工程放在光盘目录,路径为:“**4, 程序源码\标准例程-库函数版本\实验 0 Template 工程模板**”下面,大家在学习新建工程过程中遇到一些问题,可以直接打开这个模板,然后对比学习。

- 1) 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。这是工程的根目录文件夹。然后为了方便我们存放工程需要的一些其他文件,这里我们还新建下面 5 个子文件夹: CORE ,FWLIB,OBJ,SYSTEM,USER。至于这些文件夹名字,实际上是可以任取的,我们这样取名只是为了方便识别。对于这些文件夹用来存放什么文件,我们后面的步骤会一一提到。新建好的目录结构如下图 3.3.2.1.

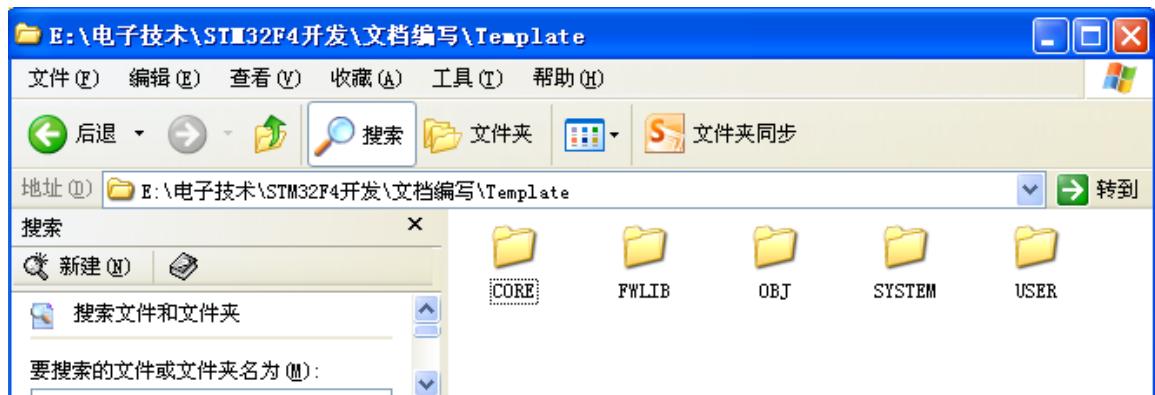


图 3.3.2.1 新建文件夹

- 2) 接下来,打开 Keil,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 之下的 USER 子目录,同时,工程取名为 Template 之后点击保存,我们的工程文件就都保存到 USER 文件夹下面。操作过程如下图:

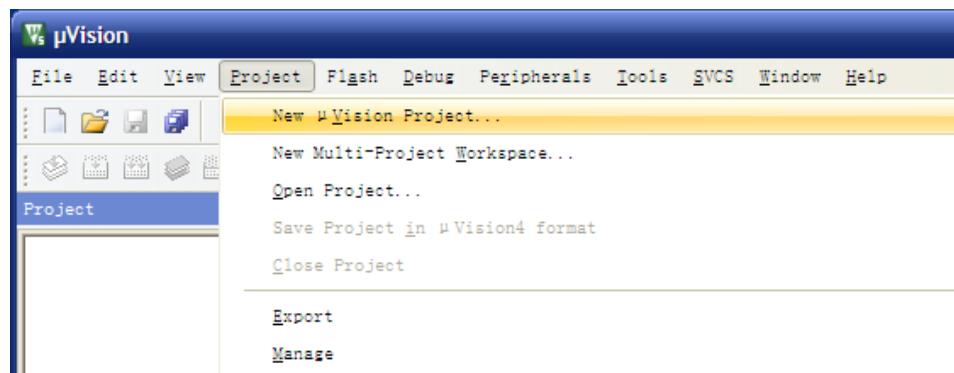


图 3.3.2.2 新建工程

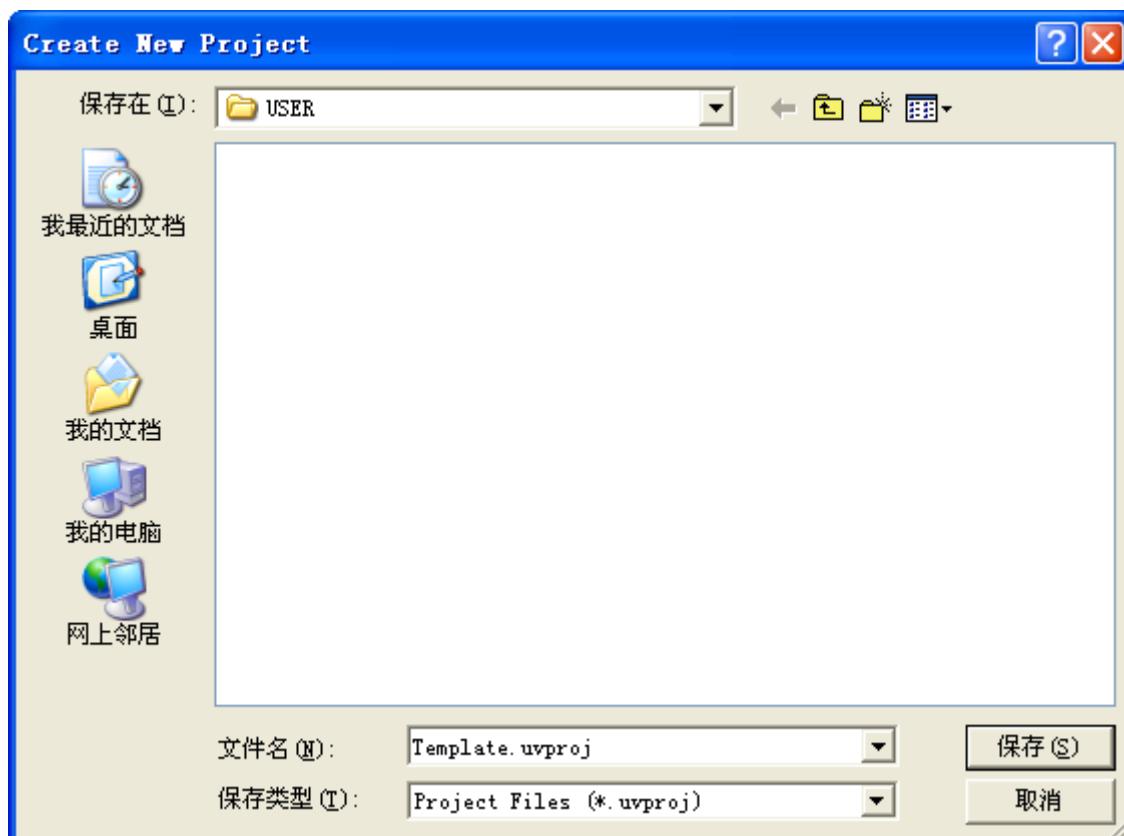


图 3.3.2.3 定义工程名称

- 3) 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 STM32F407ZG(**针对我们的 ExplorerSTM32 板子是这个型号**)。这里我们选择 STMicroelectronics → STM32F4 Series → STM32F407 → STM32F407ZG(如果使用的是其他系列的芯片，选择相应的型号就可以了，例如我们的战舰 STM32 开发板是 STM32F103ZE。特别注意：**一定要安装对应的器件 pack 才会显示这些内容哦!!**)。

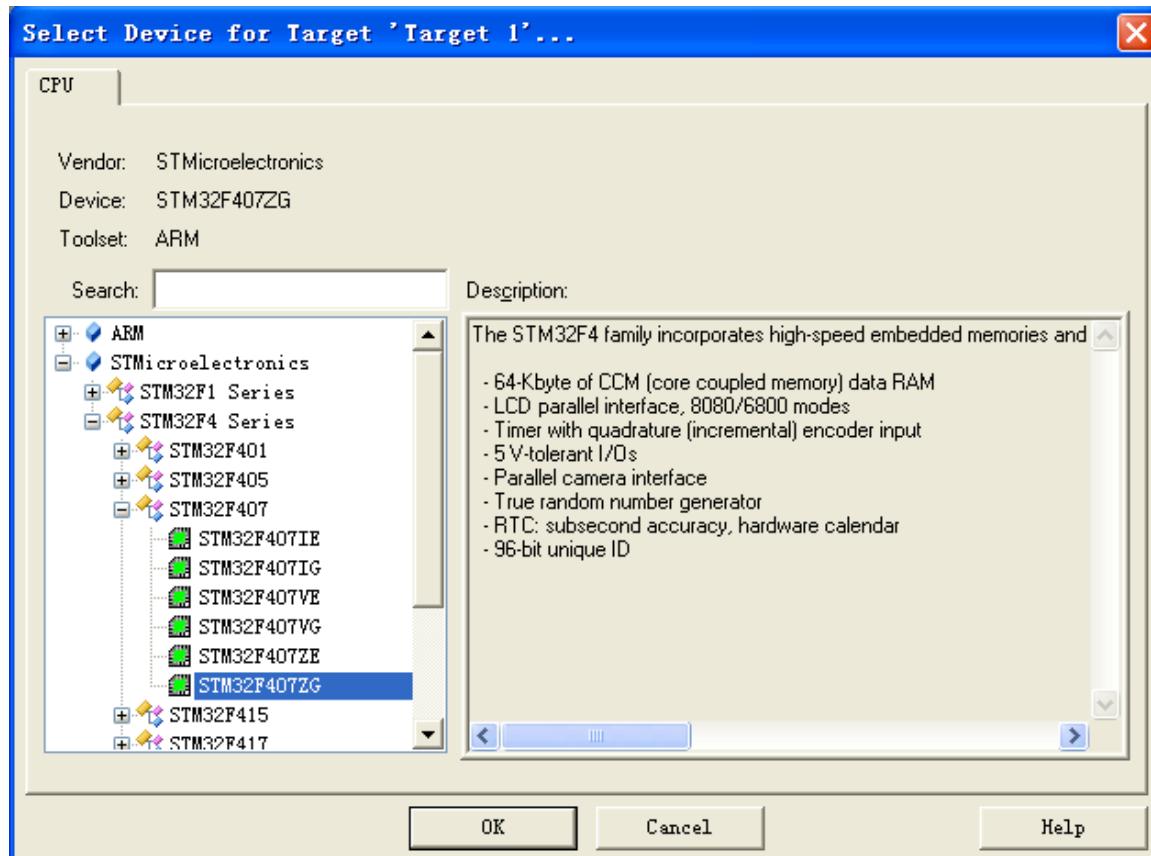


图 3.3.2.4 选择芯片型号

点击 OK，MDK 会弹出 Manage Run-Time Environment 对话框，如图 3.3.2.5 所示：

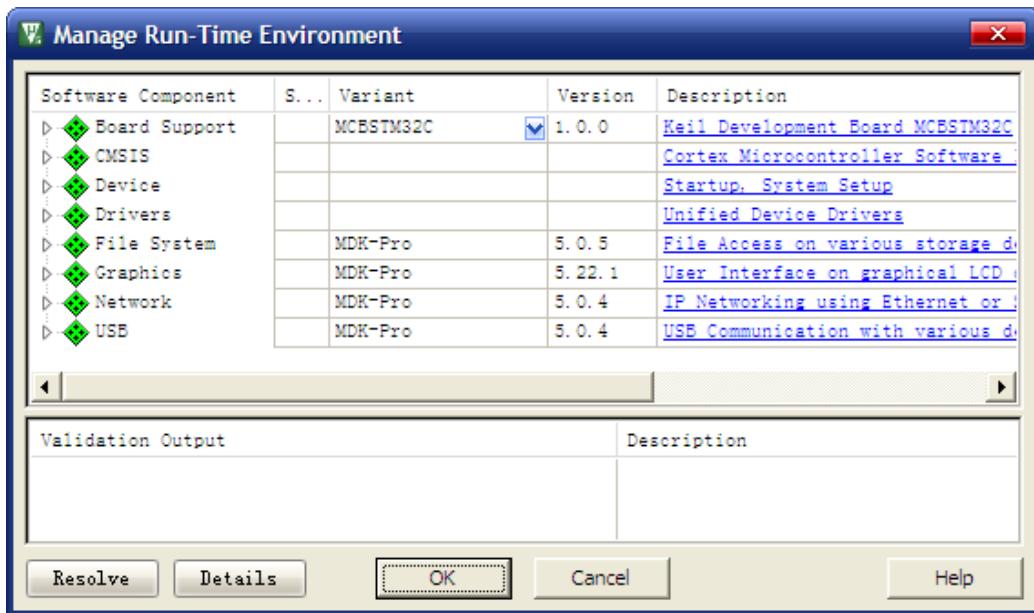


图 3.3.2.5 Manage Run-Time Environment 界面

这是 MDK5 新增的一个功能，在这个界面，我们可以添加自己需要的组件，从而方便构建开发环境，不过这里我们不做介绍。所以在图 3.3.2.5 所示界面，我们直接点击 Cancel，即可，得到如图 3.3.2.6 所示界面：

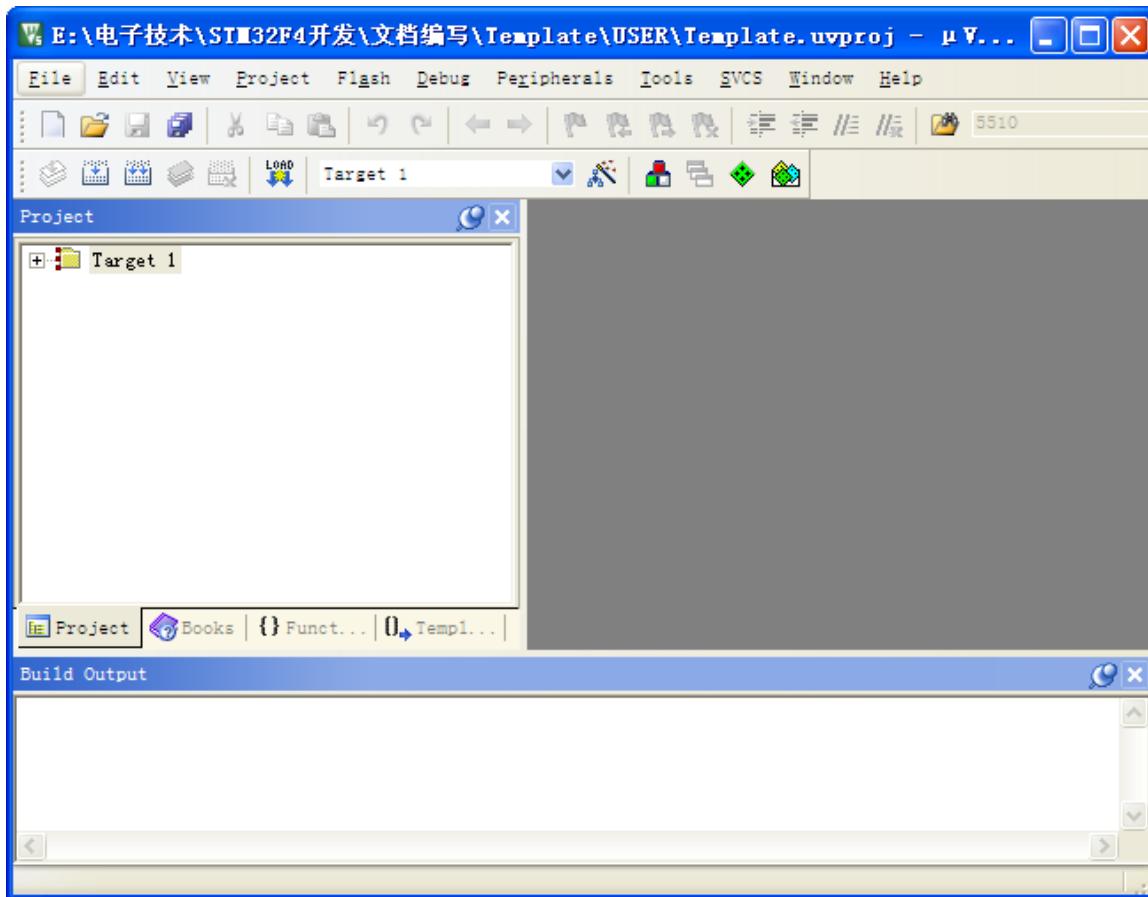


图 3.3.2.6 工程初步建立

- 4) 现在我们看看 USER 目录下面包含 2 个文件，如下图 3.3.2.7：



图 3.3.2.7 工程 USER 目录文件

- 5) 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：

\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\STM32F4xx_StdPeriph_Driver 下面，
将目录下面的 src,inc 文件夹 copy 到我们刚才建立的 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。如下图 3.3.2.8。

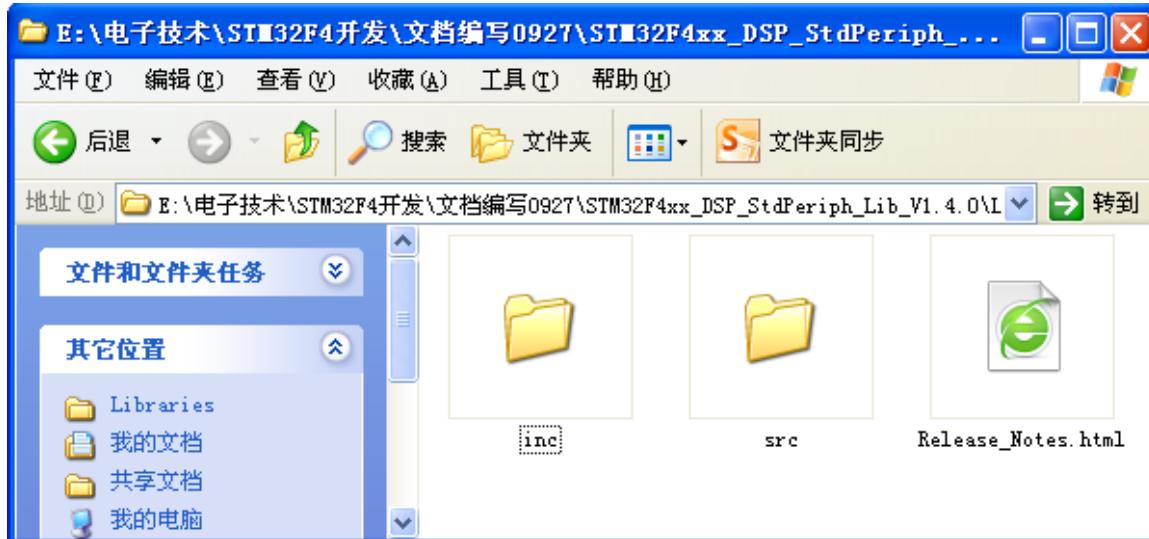


图 3.3.2.8 官方库源码文件夹

6) 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

\STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm 下面，将文件 **startup_stm32f40_41xxx.s** 复制到 CORE 目录下面。然后定位到目录 \STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Includ，将里面的头文件 **core_cm4.h** 和 **core_cm4_simd.h** 同样复制到 CORE 目录下面。现在看看我们的 CORE 文件夹下面的文件，如下图 3.3.2.9：

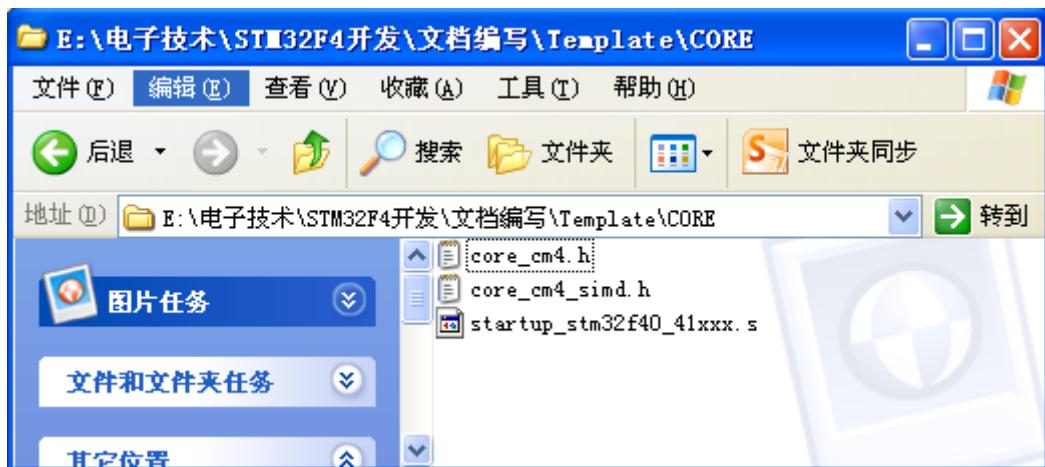


图 3.3.2.9 CORE 文件夹文件

7) 接下来我们要复制工程模板需要的一些其他头文件和源文件到我们工程。首先定位到目录：STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Include 将里面的 2 个头文件 **stm32f4xx.h** 和 **system_stm32f4xx.h** 复制到 USER 目录之下。这两个头文件是 STM32F4 工程非常关键的两个头文件。后面我们讲解相关知识的时候会给大家详细讲解。然后进入目录 \STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Project\STM32F4xx_StdPeriph_Templates，将目录下面的 5 个文件 **main.c**，**stm32f4xx_conf.h**，**stm32f4xx_it.c**，**stm32f4xx_it.h**，

system_stm32f4xx.c 复制到 USER 目录下面。请按下图 3.3.2.10 选中 5 个文件然后复制:

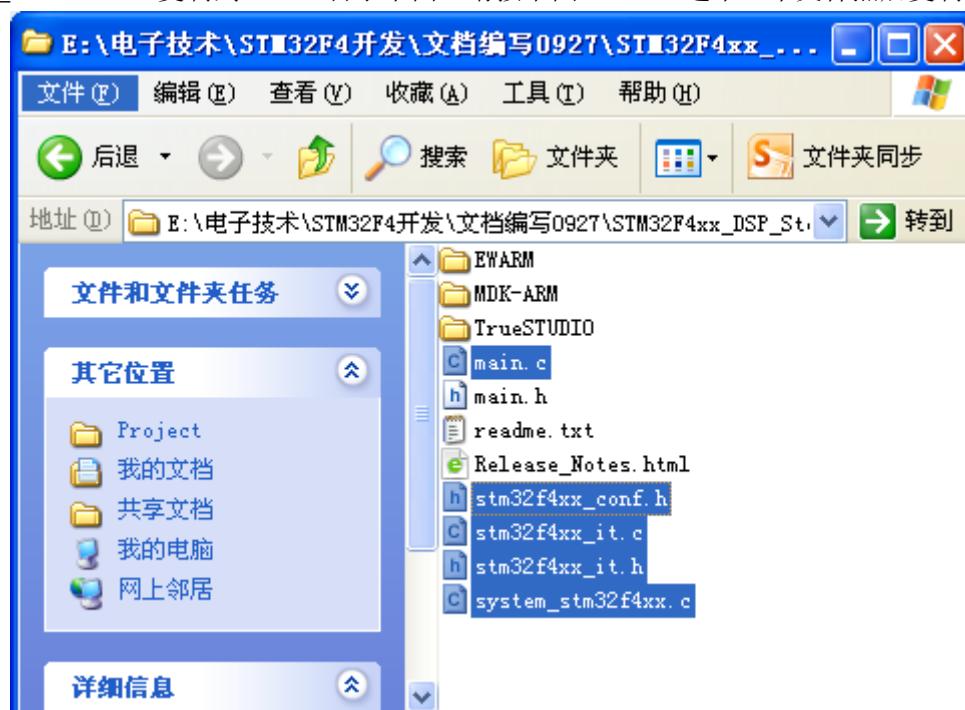


图 3.3.2.10 USER 目录文件浏览

相关文件复制到 USER 目录之后 USER 目录文件如下图 3.3.2.11:

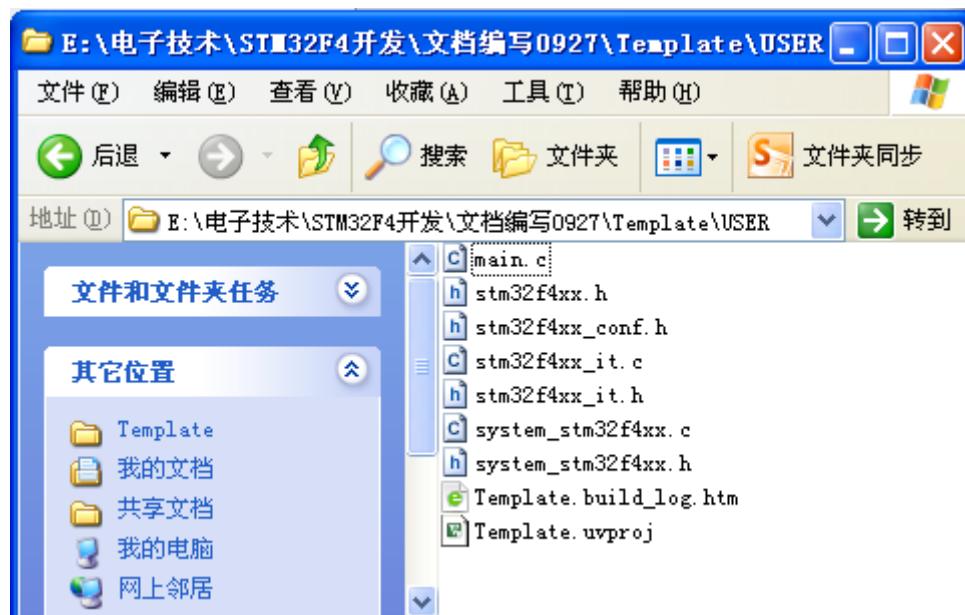


图 3.3.2.11 USER 目录文件浏览

- 8) 前面 7 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，如下图：

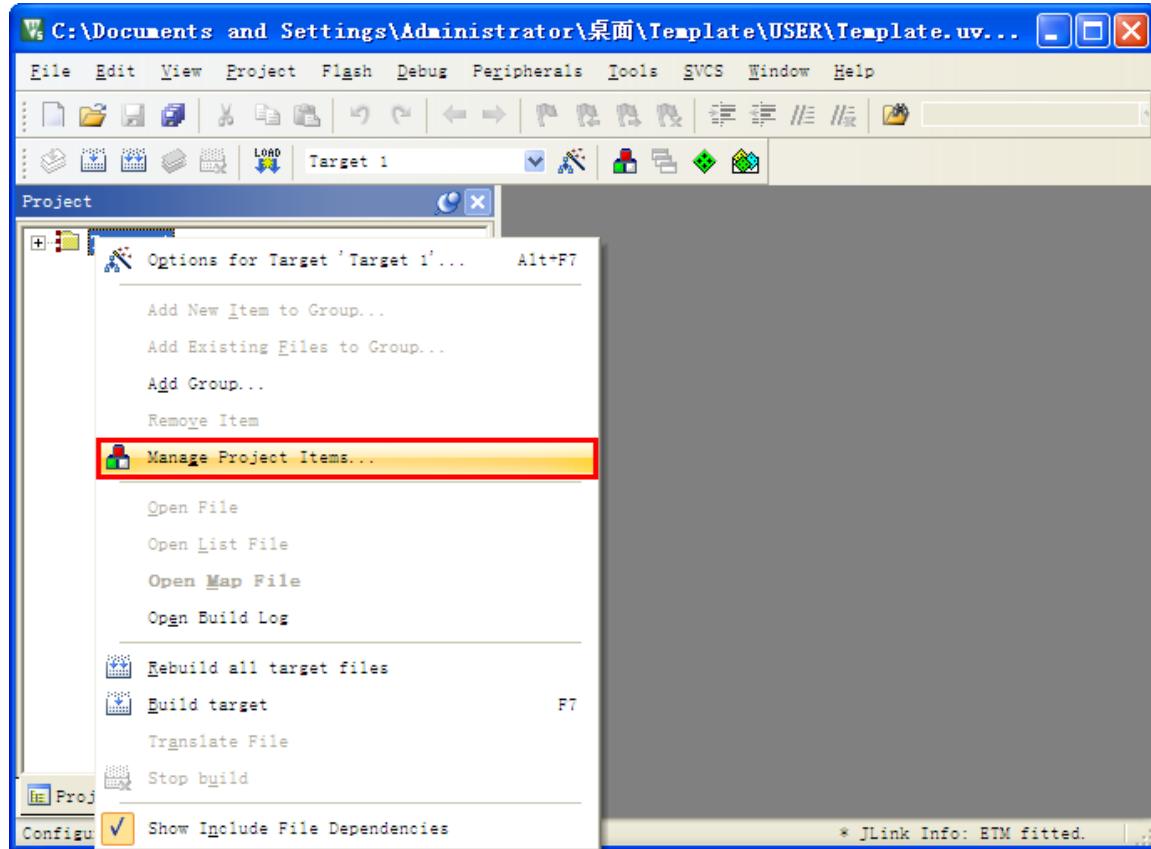


图 3.3.2.12 点击 Management Components

- 9) 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 情况如下图:

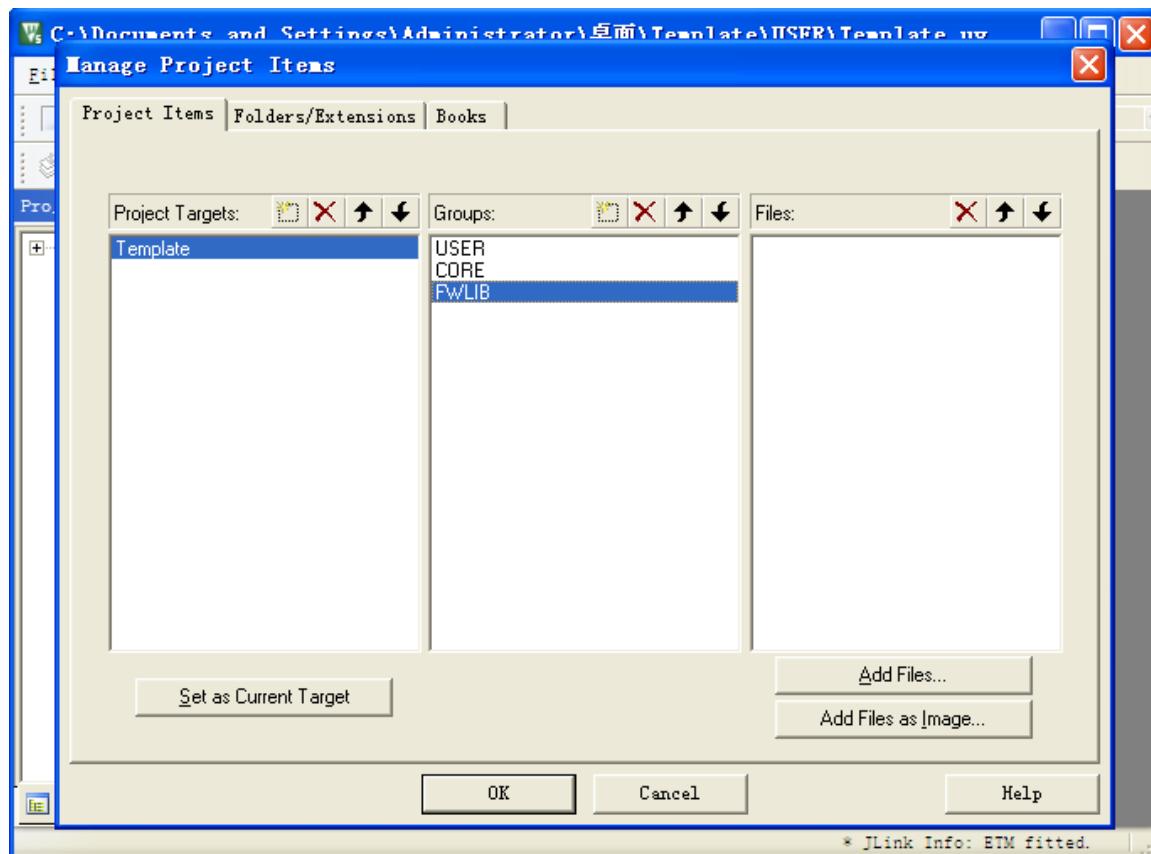


图 3.3.2.13 新建 GROUP

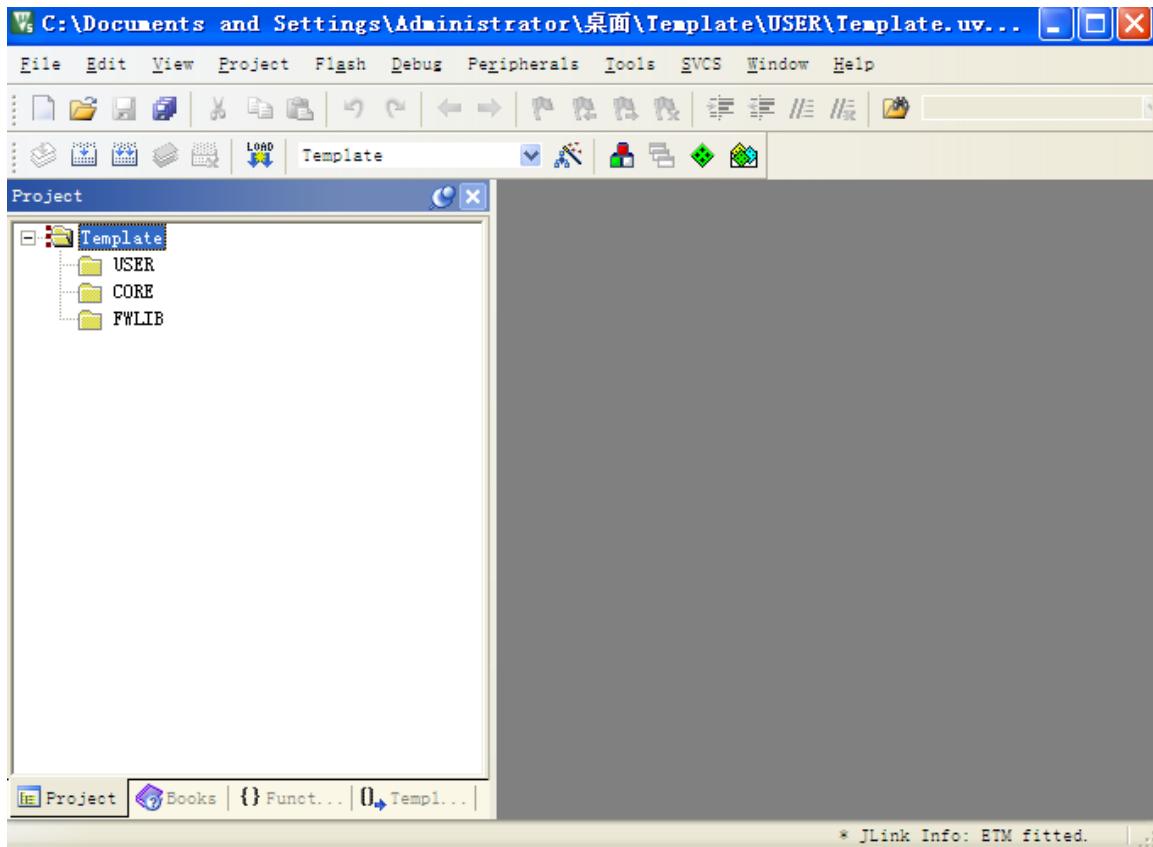


图 3.3.2.14 查看工程 Group 情况

- 10) 下面我们往 Group 里面添加我们需要的文件。我们按照步骤 9 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\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，如下图 3.3.2.15。

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 stm32f4xx_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。

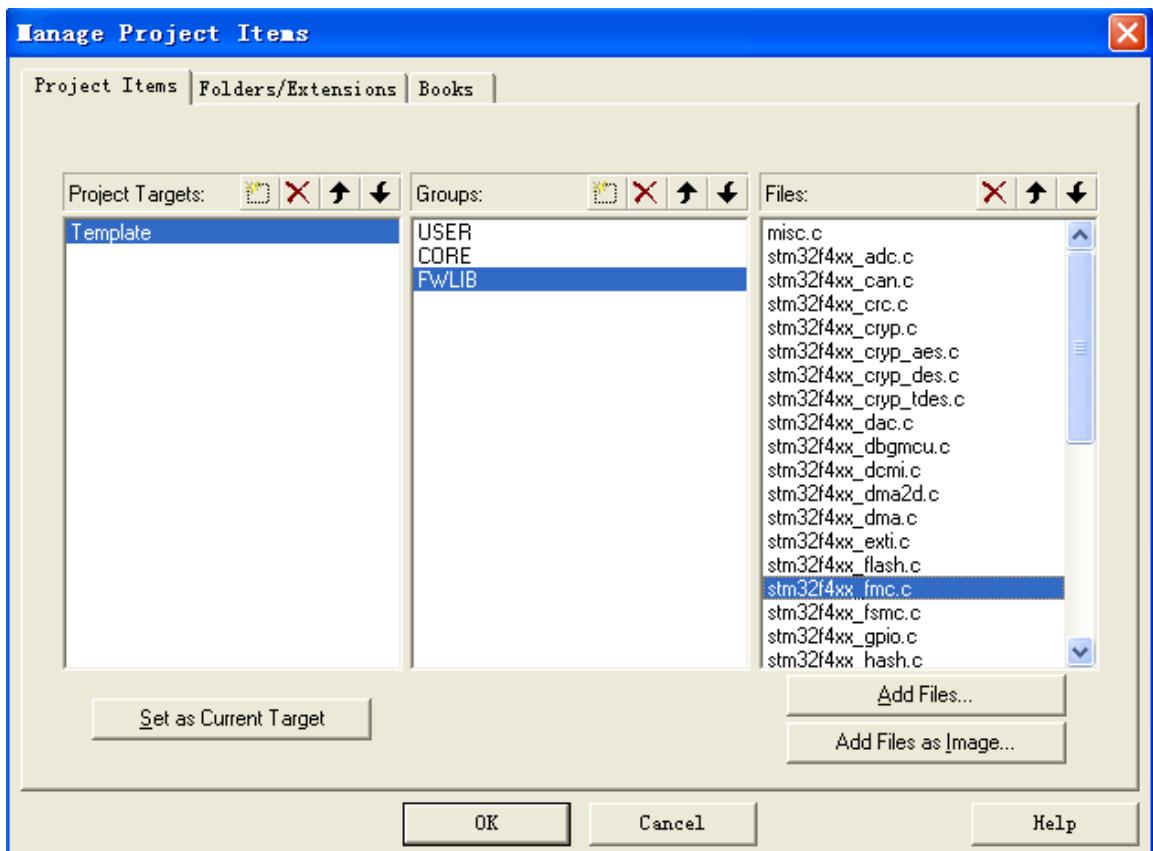


图 3.3.2.15 添加文件到 FWLIB 分组

这里有个文件 **stm32f4xx_fmc.c** 比较特殊。这个文件是 STM32F42 和 STM32F43 系列才用到，所以我们这里要把它删掉（**注意是 stm32f4xx_fmc.c 要删掉，不要删掉 stm32f4xx_fsmc.c**）。如下图 3.3.2.16 所示：

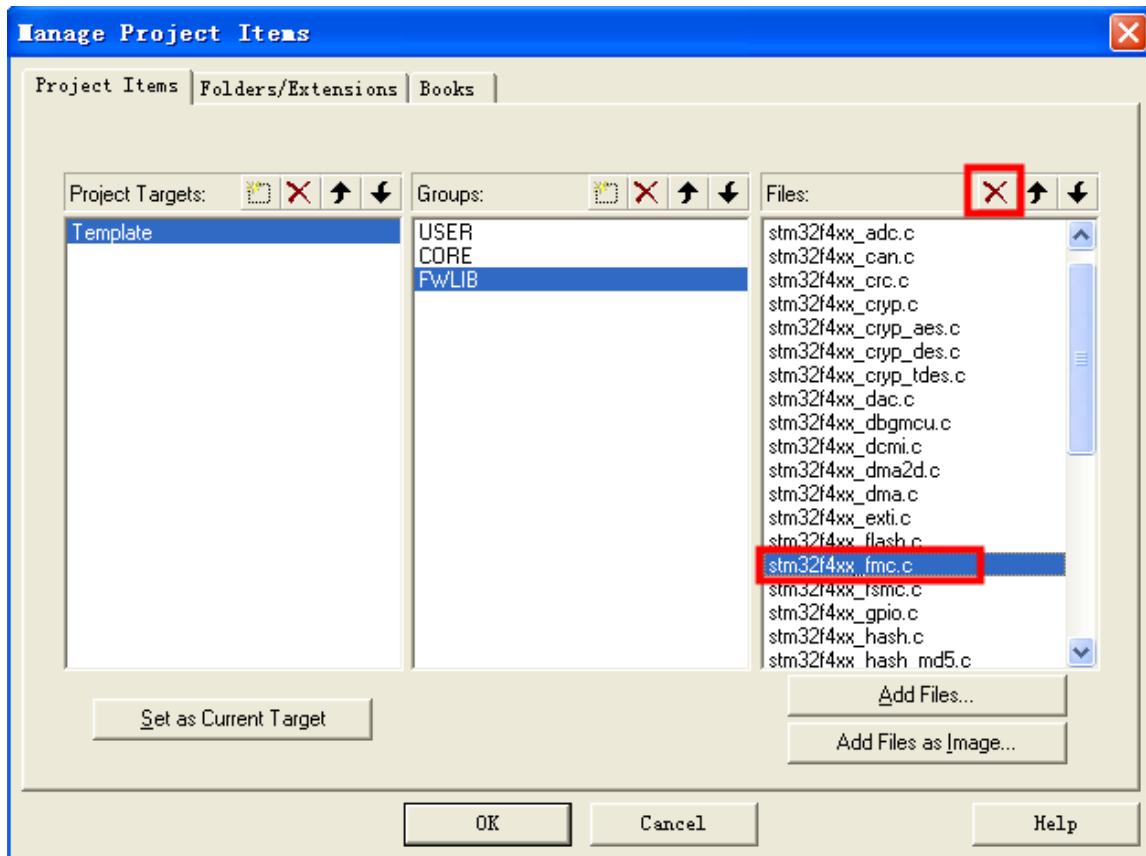


图 3.3.2.16 删掉 stm32f4xx_fmc.c

11) 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 **startup_stm32f40_41xxx.s**(注意，默认添加的时候文件类型为.c,也就是添加 **startup_stm32f40_41xxx.s** 启动文件的时候，你需要选择文件类型为 All files 才能看到这个文件)，USER 目录下面需要添加的文件为 main.c, stm32f4xx_it.c, system_stm32f4xx.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。操作过程如下图 3.3.2.17~3.3.2.20:

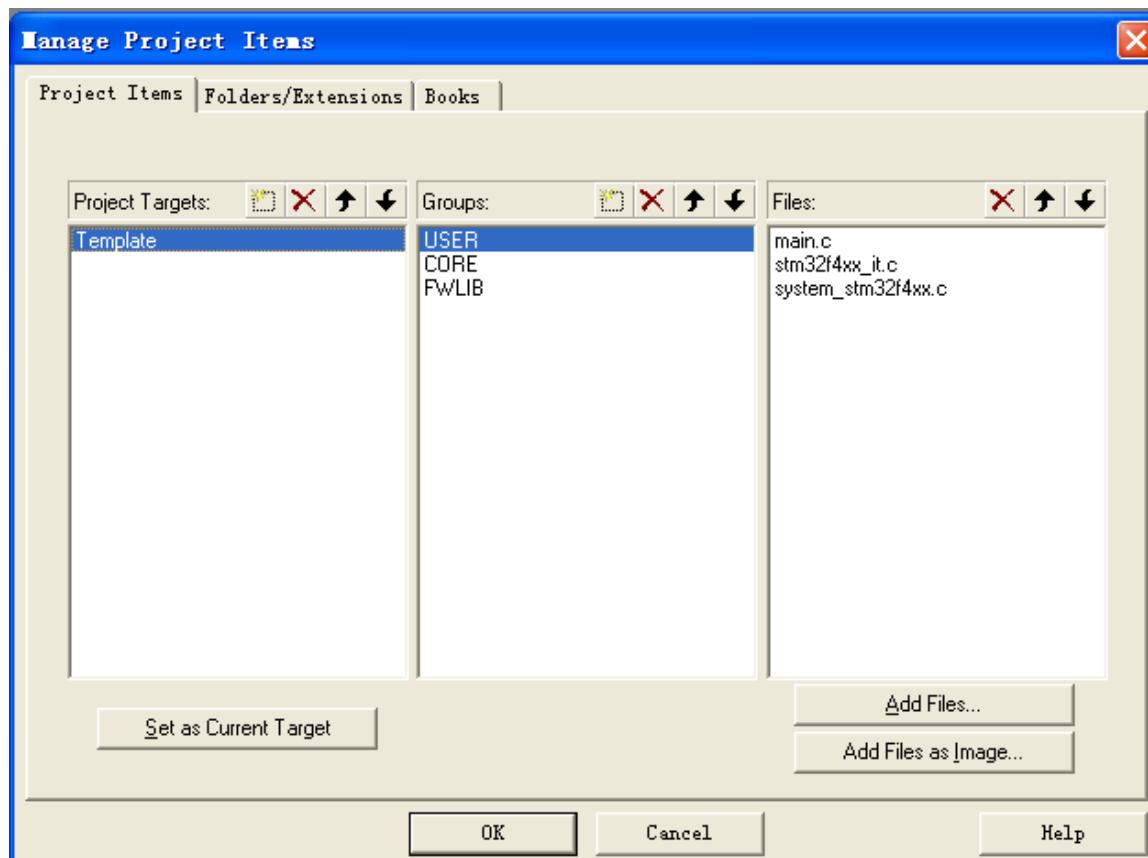


图 3.3.2.17 添加文件到 USER 分组

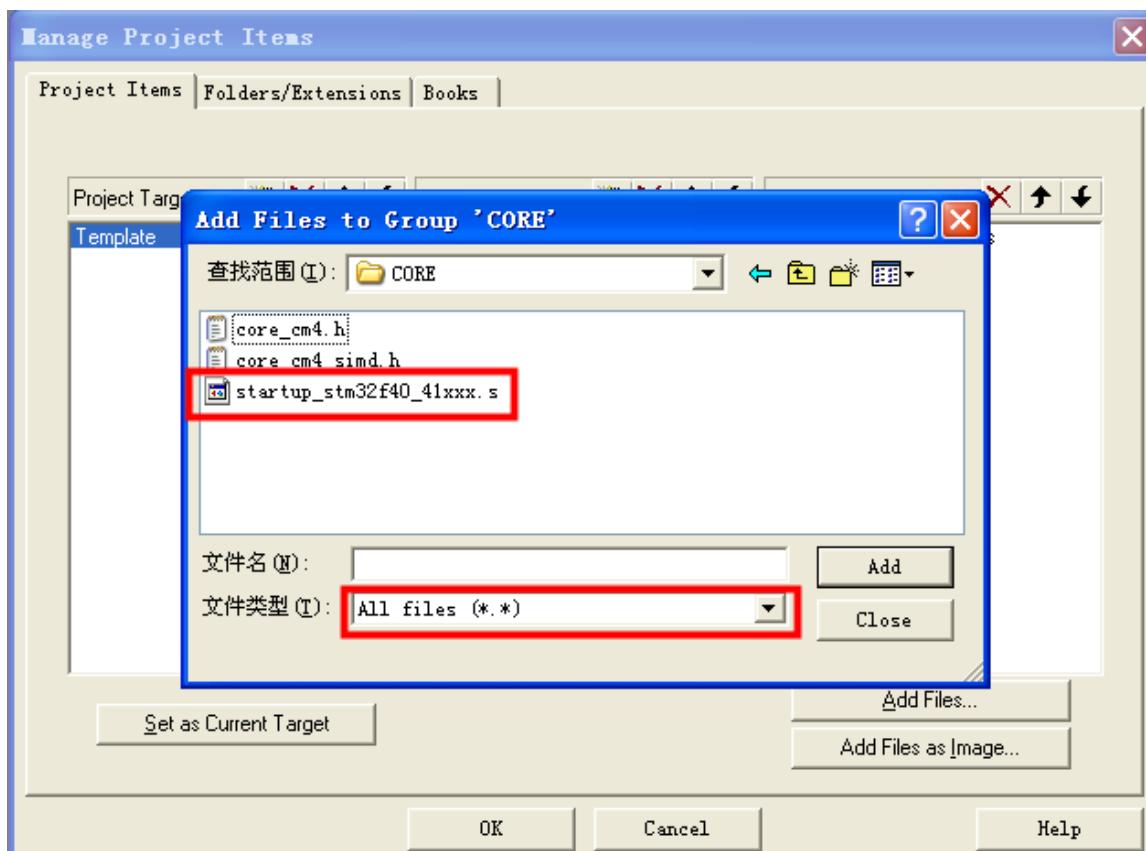


图 3.3.2.18 添加文件 startup_stm32f40_41xxx.s

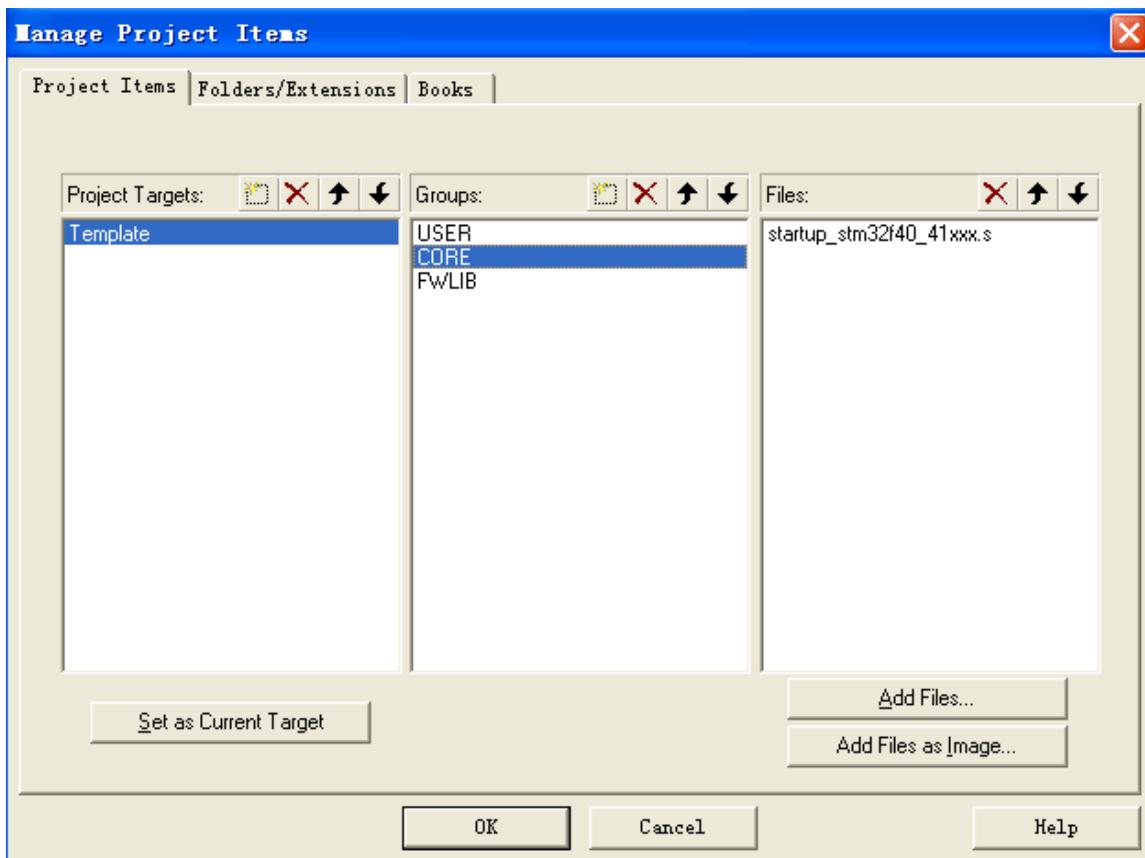


图 3.3.2.19 添加文件到 CORE 分组

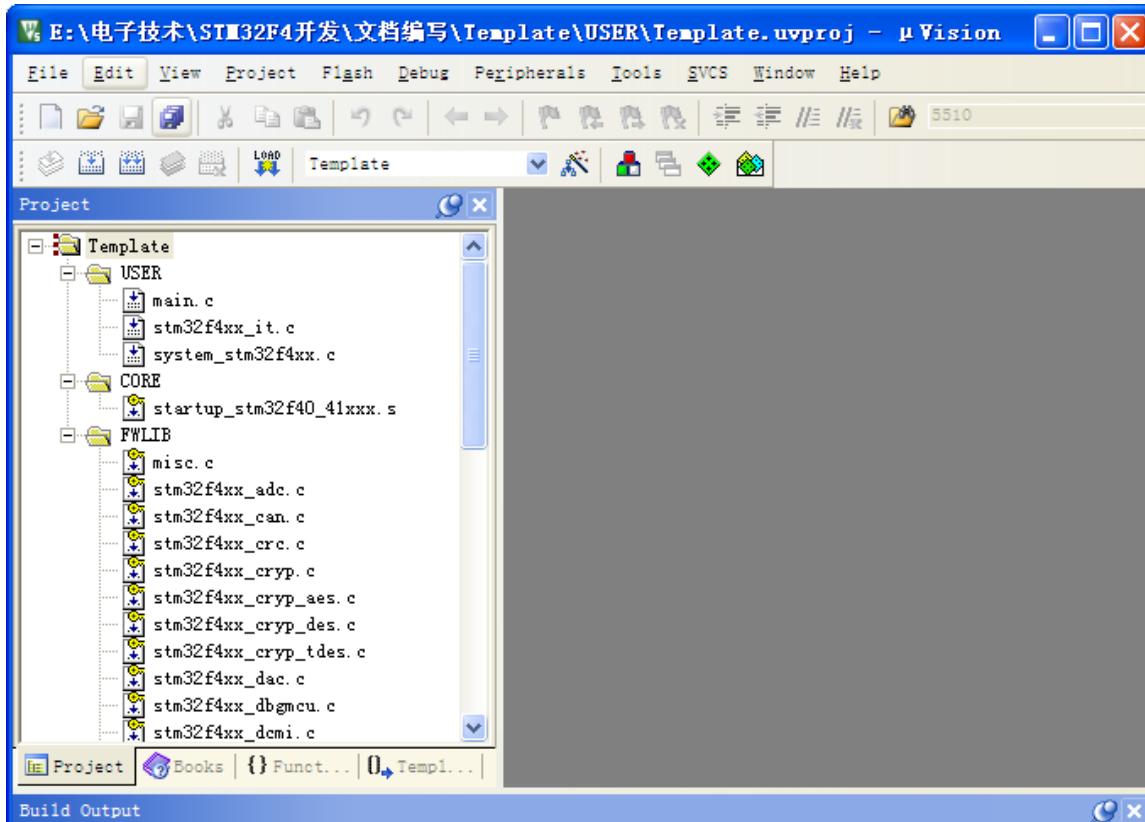


图 3.3.2.20 工程分组情况

- 12) 接下来我们要在 MDK 里面设置头文件存放路径。也就是告诉 MDK 到那些目录下面去寻找包含了的头文件。这一步骤非常重要。**如果没有设置头文件路径，那么工程会出现报错头文件路径找不到。**具体操作如下图，5 步之后添加相应的头文件路径。

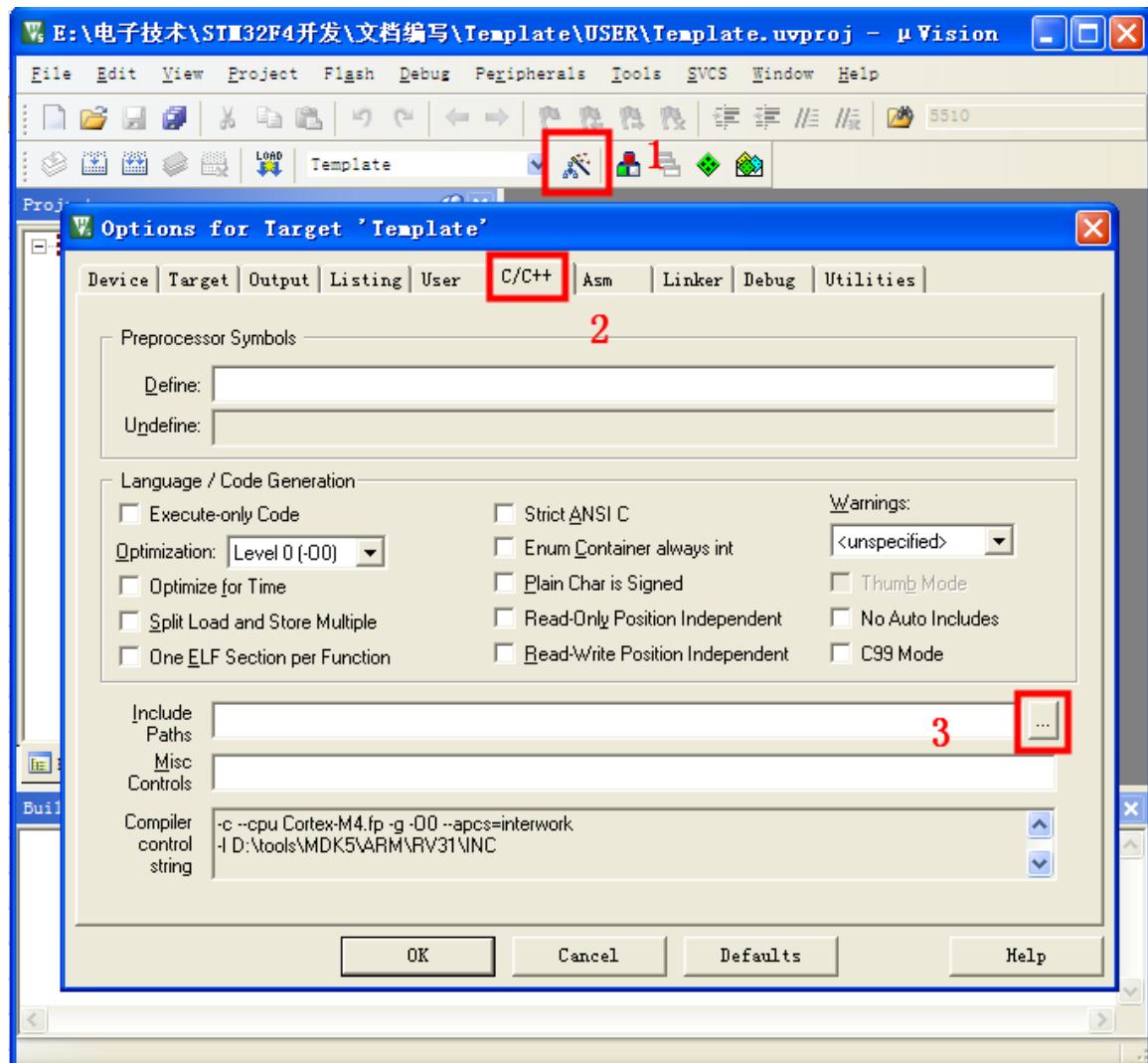


图 3.3.2.21 进入 PATH 配置界面

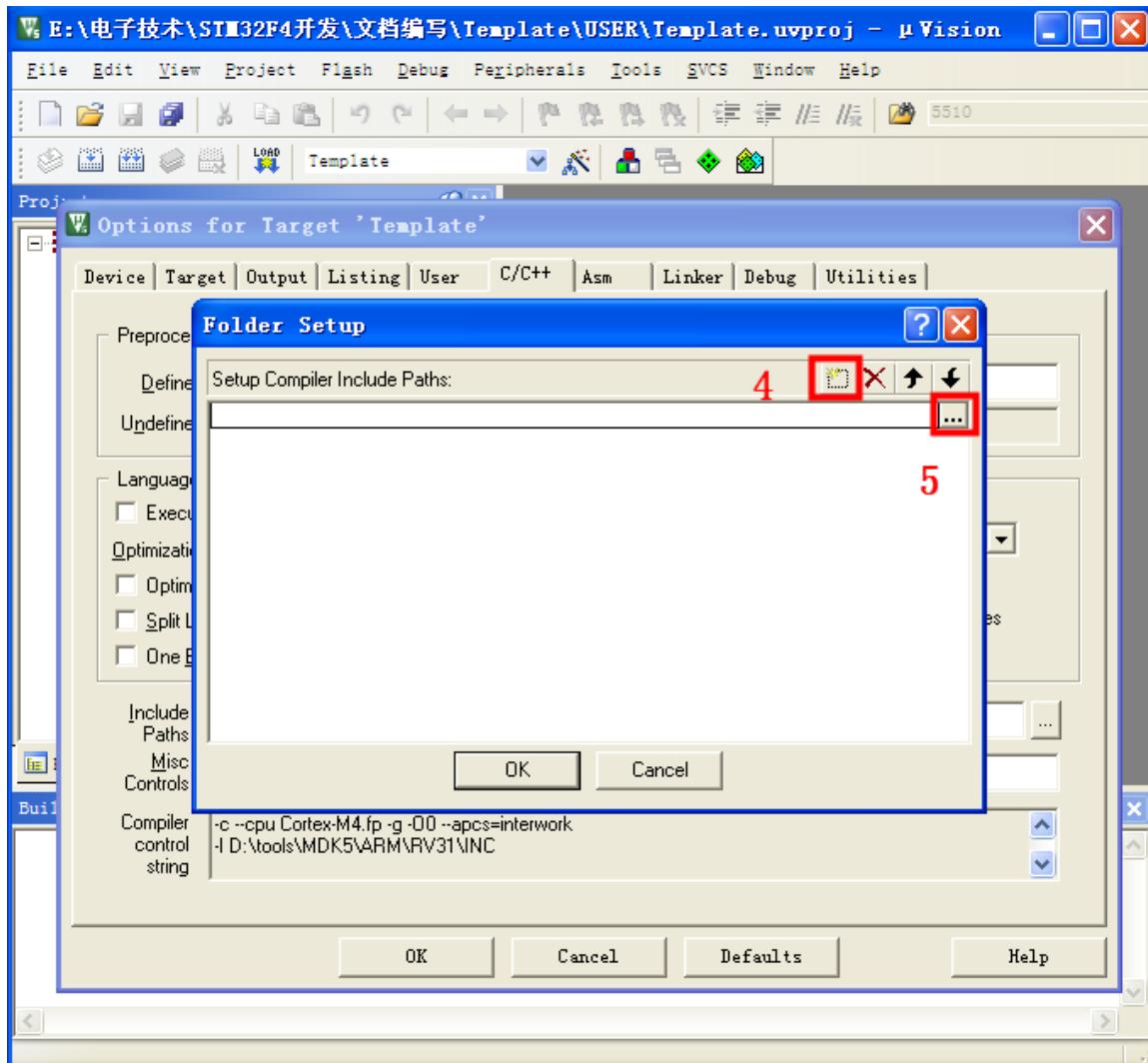


图 3.3.2.22 添加头文件路径到 PATH

这里我们需要添加的头文件路径包括: \CORE, \USER\以及\FWLIB\inc。这里大家务必要仔细, 固件库存放的头文件子目录是**\FWLIB\inc**, 不是 **FWLIB\src**。很多朋友都是这里弄错导致报很多奇怪的错误。添加完成之后如下图 3.3.2.23。

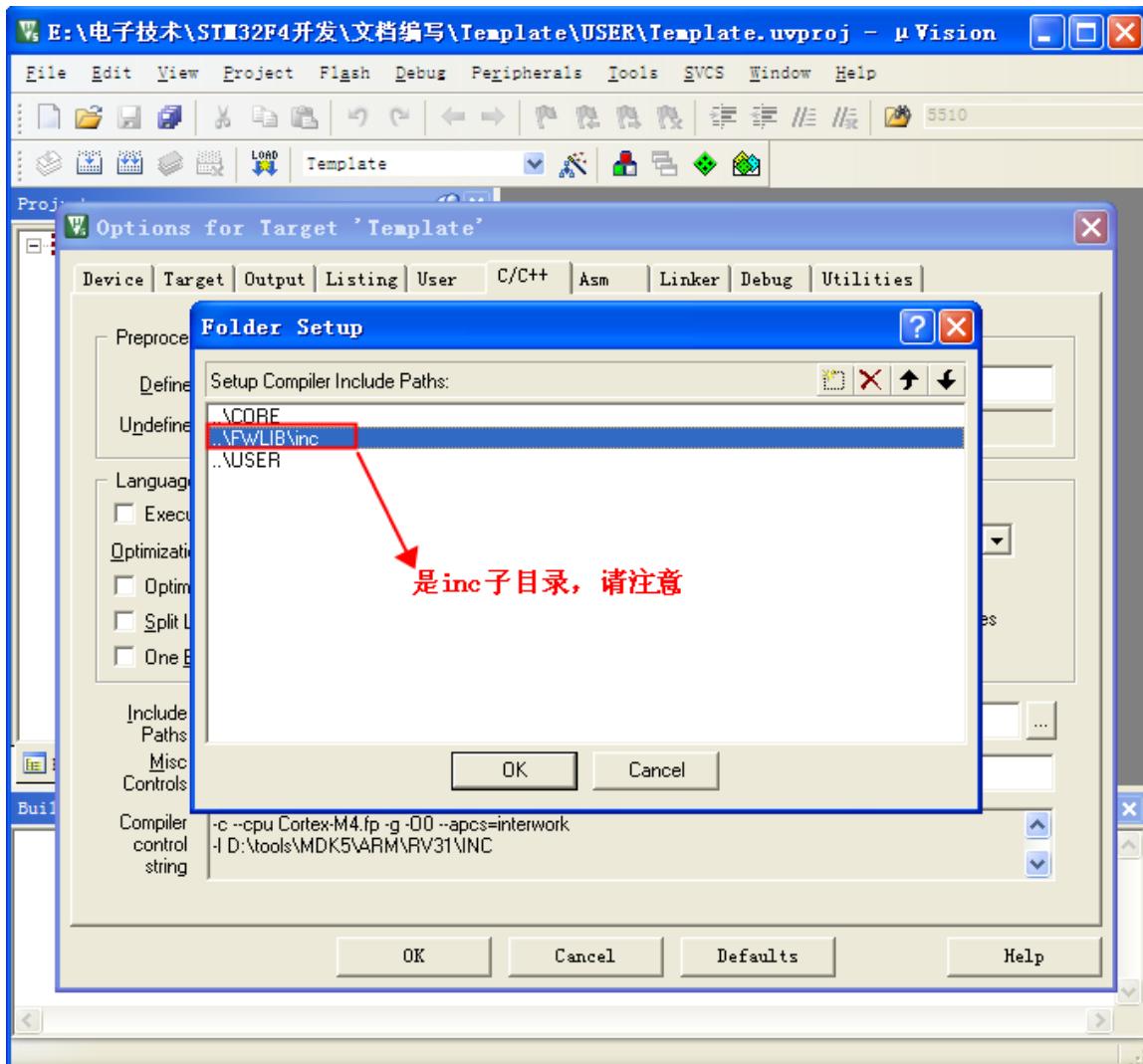


图 3.3.2.23 添加头文件路径

- 13) 接下来对于 STM32F40 系列的工程，还需要添加一个全局宏定义标识符。添加方法是点击魔术棒之后，进入 C/C++选项卡，然后在 Define 输入框连输入：
STM32F40_41xxx,USE_STDPERIPH_DRIVER。注意这里是两个标识符 STM32F40_41xxx 和 USE_STDPERIPH_DRIVER，**他们之间是用逗号隔开的**，请大家注意。
这个字符串大家可以直接打开我们光盘的新建好的工程模板，从里面复制。模板存放目录为：**4. 程序源码\标准例程-库函数版本\实验 0 Template 工程模板**

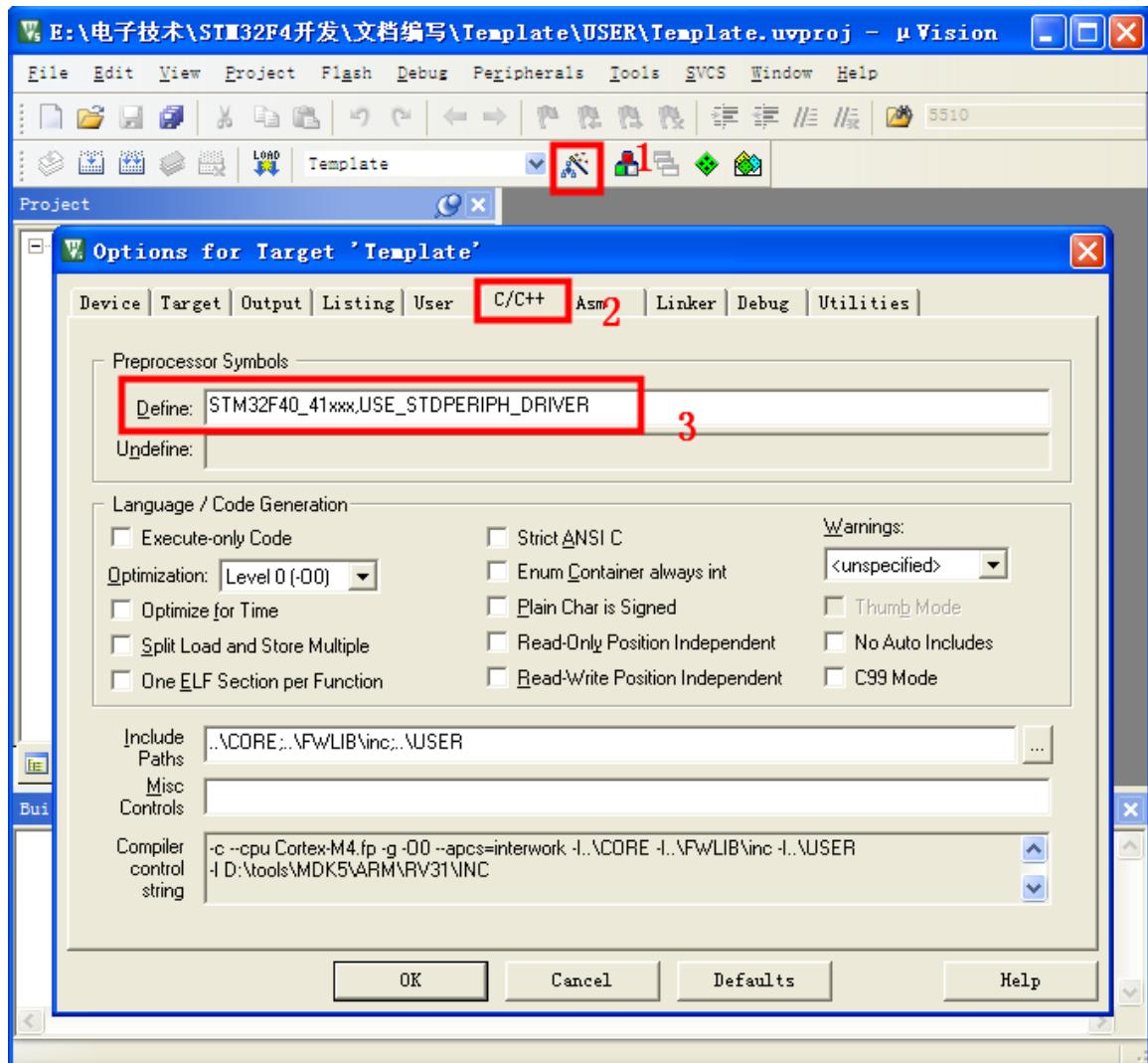


图 3.3.2.24 添加全局宏定义标识符

14) 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**。同时将下方的三个选项框都勾上，操作过程如下图 3.3.2.24：

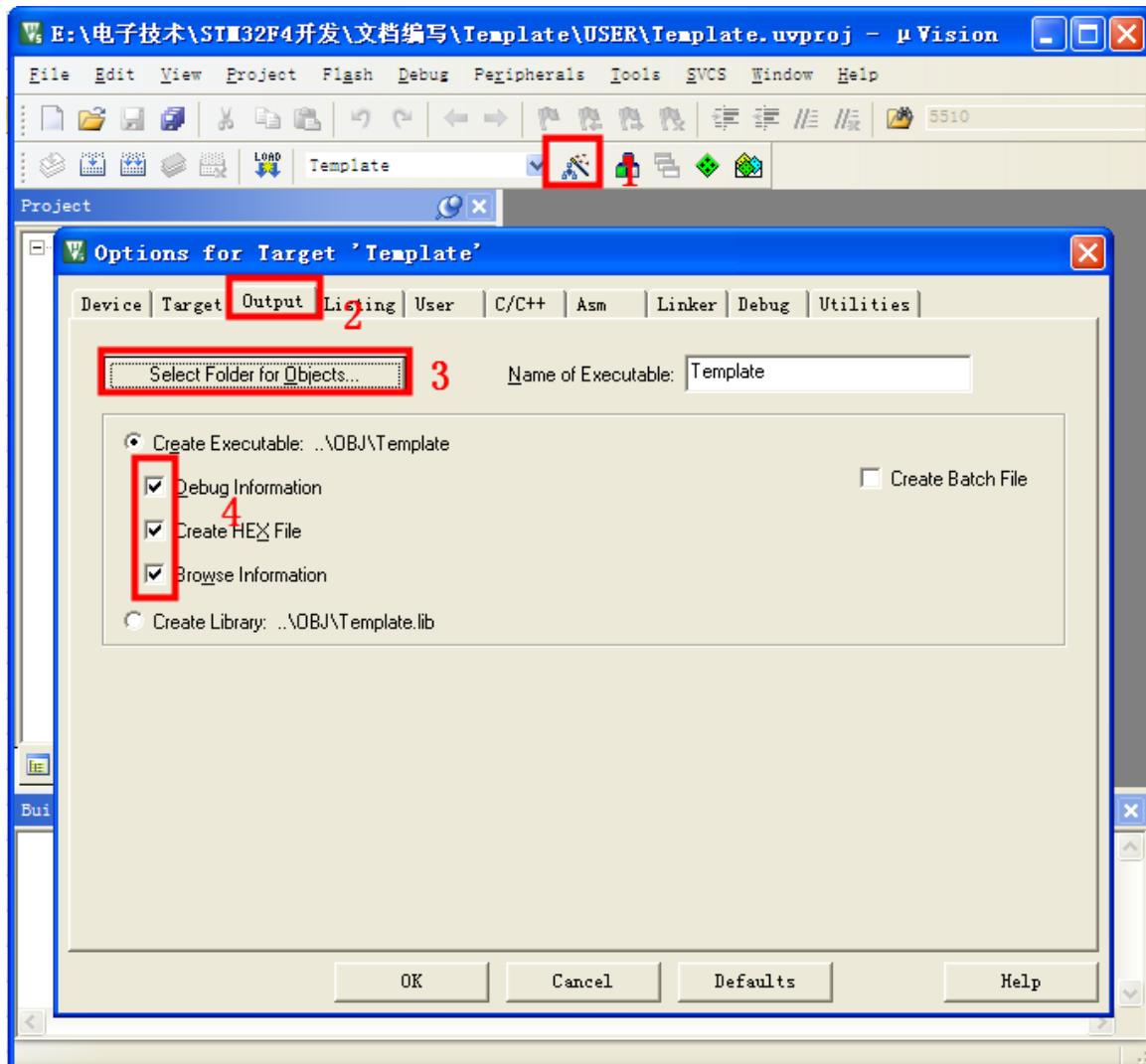


图 3.3.2.24 选择编译中间文件存放目录

这里说明一下步骤 4 的意义。Create HEX File 选项选上是要求编译之后生成 HEX 文件。Browse Information 选项选上是方便我们查看工程中的一些函数变量定义。

- 15) 在编译之前，我们先把 main.c 文件里面的内容替换为如下内容：

```
#include "stm32f4xx.h"

void Delay(__IO uint32_t nCount);
void Delay(__IO uint32_t nCount)
{
    while(nCount--){ }
}
int main(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10;
```

```

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOF, &GPIO_InitStructure);

while(1){
    GPIO_SetBits(GPIOF,GPIO_Pin_9|GPIO_Pin_10);
    Delay(0x7FFFFF);
    GPIO_ResetBits(GPIOF,GPIO_Pin_9|GPIO_Pin_10);
    Delay(0x7FFFFF);
}
}

```

上面这段代码，大家如果不方便自己编写，可以直接打开我们光盘库函数源码目录“**4，程序源码\标准例程-库函数版本\实验 0 Template 工程模板**”找到我们已经新建好的工程模板，工程中有一个 **README.txt** 文件，里面存放了这段代码，直接复制过来即可。

与此同时，我们要将 **USER** 分组下面的 **stm32f4xx_it.c** 文件内容清空。或者删掉其中的 32 行对 **main.h** 头文件的引入以及 144 行 **SysTick_Handler** 函数内容，如下图 3.3.2.25 和图 3.3.2.26：

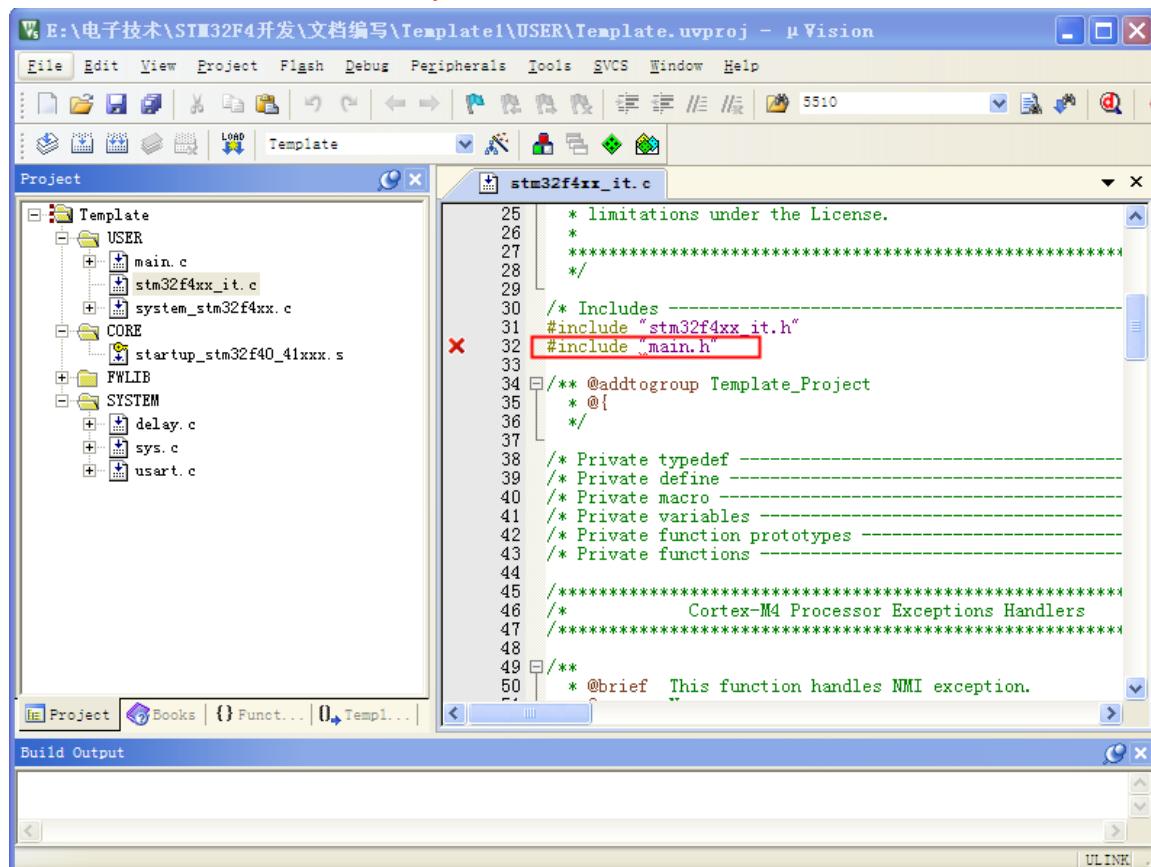


图 3.3.2.25 删掉 **stm32f4xx_it.c** 里面 **main.h** 的引入

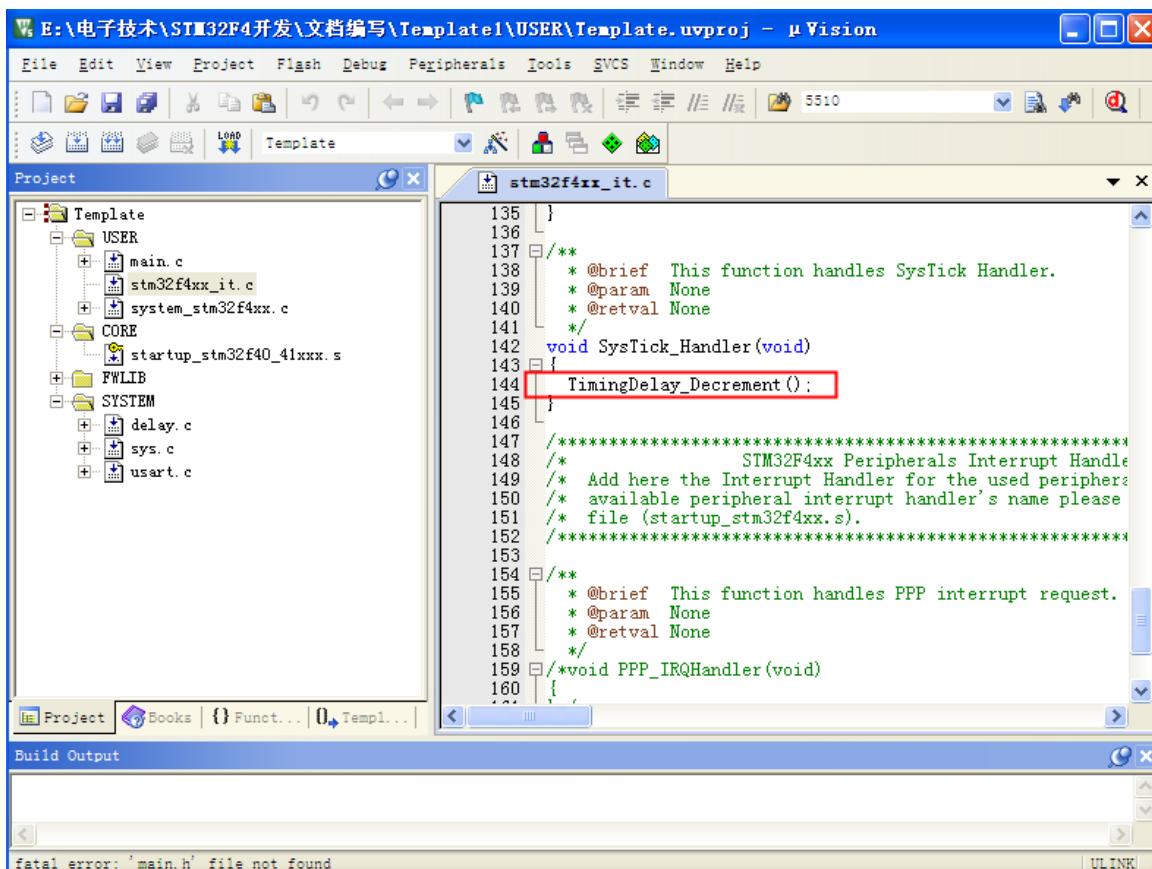


图 3.3.2.26 删掉 stm32f4xx_it.c 里的 SysTick_Handler 函数内容

16) 下面我们点击编译按钮 编译工程，可以看到工程编译通过没有任何错误和警告。

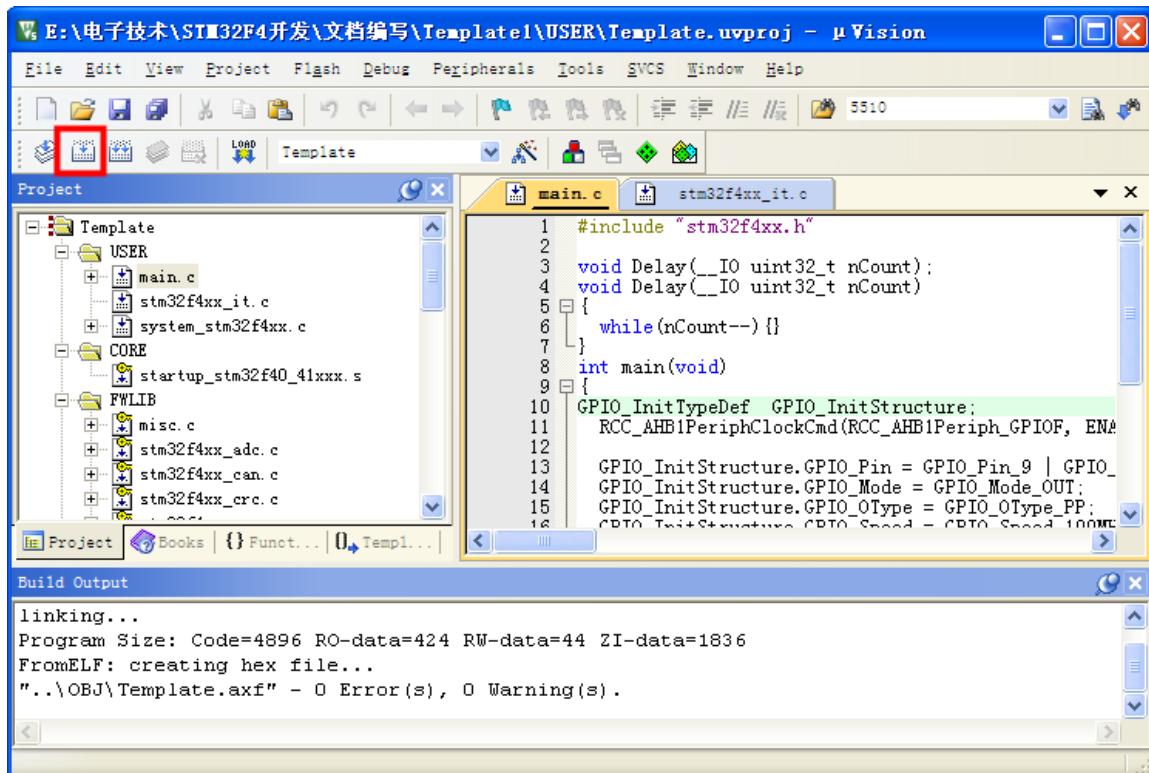


图 3.3.2.27 编译工程

- 17) 到这里，一个基于固件库 V1.4 的工程模板就建立完成，同时在工程的 OBJ 目录下面生成了对应的 hex 文件。大家可以参考后面我们 3.4 小节的内容，将 hex 文件下载到开发板，会发现两个 led 灯不停的闪烁。
- 18) 这里还有个非常重要的关键点，就是系统时钟的配置，这在我们的系统时钟章节 4.3 会详细讲解，这里我们要修改 System_stm32f4xx.c 文件，把 PLL 第一级分频系数 M 修改为 8，这样达到主时钟频率为 168MHz。修改方法如下：

```
***** PLL Parameters *****
#ifndef STM32F40_41xxx
#define STM32F427_437xx
#define STM32F429_439xx
#define STM32F401xx

/* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
#define PLL_M     8
#endif /* STM32F411xE */

#if defined(USE_HSE_BYPASS)
#define PLL_M     8
#endif /* STM32F411xE */
#define PLL_M     16
#endif /* USE_HSE_BYPASS */
#endif
```

PLL_M 这里我们要修改为 8，这样我们的系统时钟就是 168MHz。详细原因我们后面 4.3 小节会讲解。

同时，我们要在 stm32f4xx.h 里面修改外部时钟 HSE_VALUE 值为 8MHz，因为我们的外部高速时钟用的晶振为 8M，具体修改方法如下：

```
#if !defined (HSE_VALUE)
```

```
#define HSE_VALUE      ((uint32_t)8000000) /*!< Value of the External oscillator in Hz */
```

```
#endif /* HSE_VALUE */
```

大家一定要在对应的配置文件中，找到相应的代码行，修改为符合我们硬件的值即可。

- 19) 实际上经过前面 17 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 SYSTEM 文件夹介绍会有详细的讲解，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（**大家一定要注意，打开库函数版本的实验找到 SYSTEM 文件夹，不要用寄存器版本的 SYSTEM 文件夹**），比如我们打开我们的实验 0 Template 工程模板的工程目录如下：

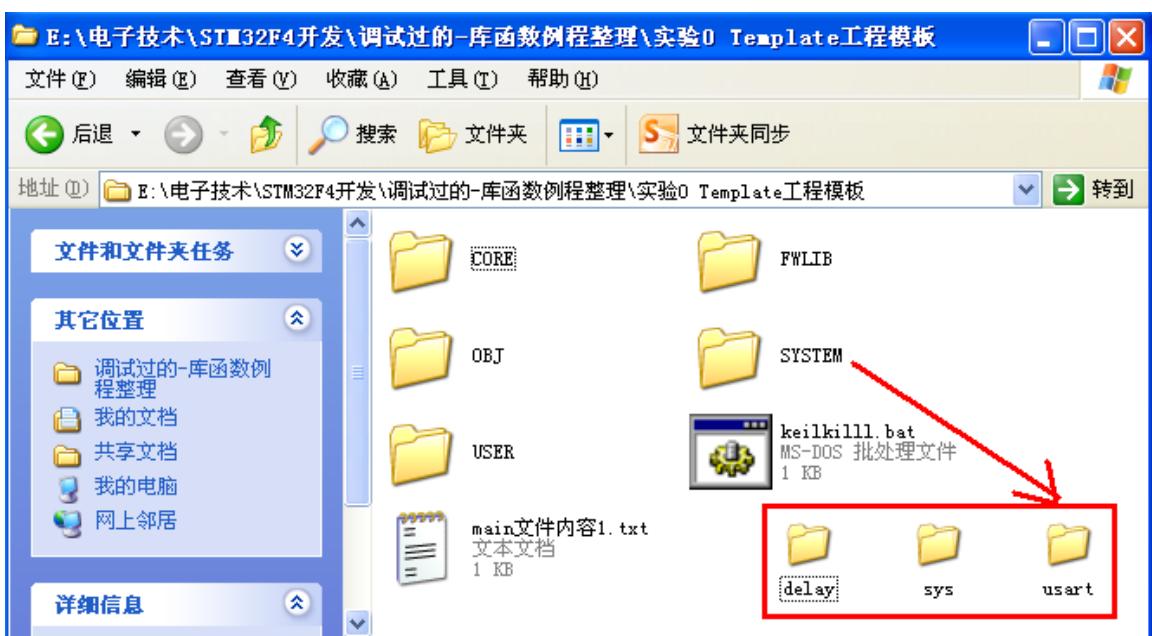


图 3.3.2.28 光盘工程模板根目录

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。**我们将 SYSTEM 文件夹和里面的三个子文件夹复制到我们工程根目录中。如下图。**我们接下来要将这三个目录下面的源文件加入到我们工程，同时将头文件路径加入到 PATH 中，如下图 3.3.2.29 所示：

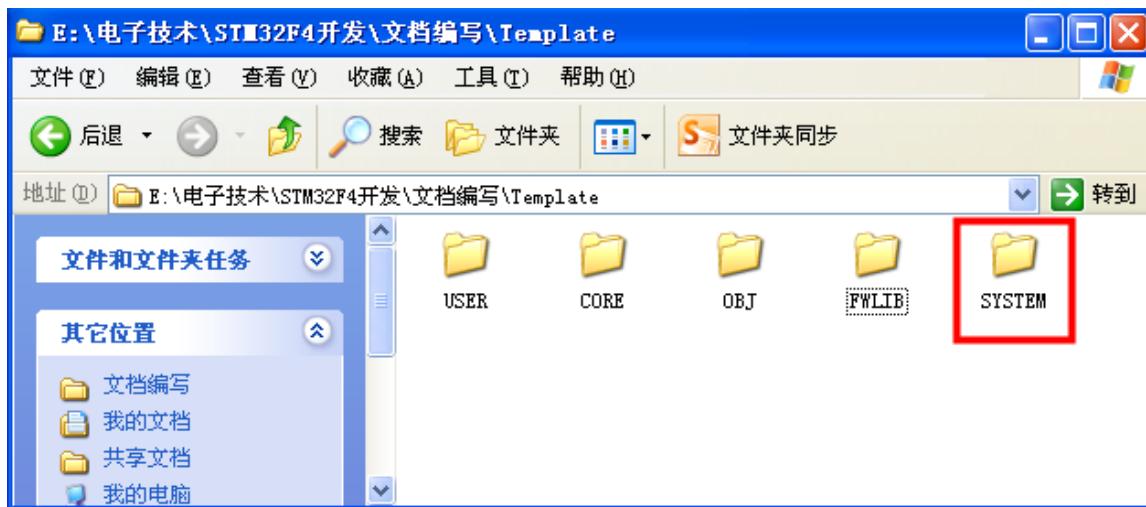


图 3.3.2.29 工程模板根目录

用我们之前讲解步骤 9 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, usart.c，如下图 3.3.2.30 所示：

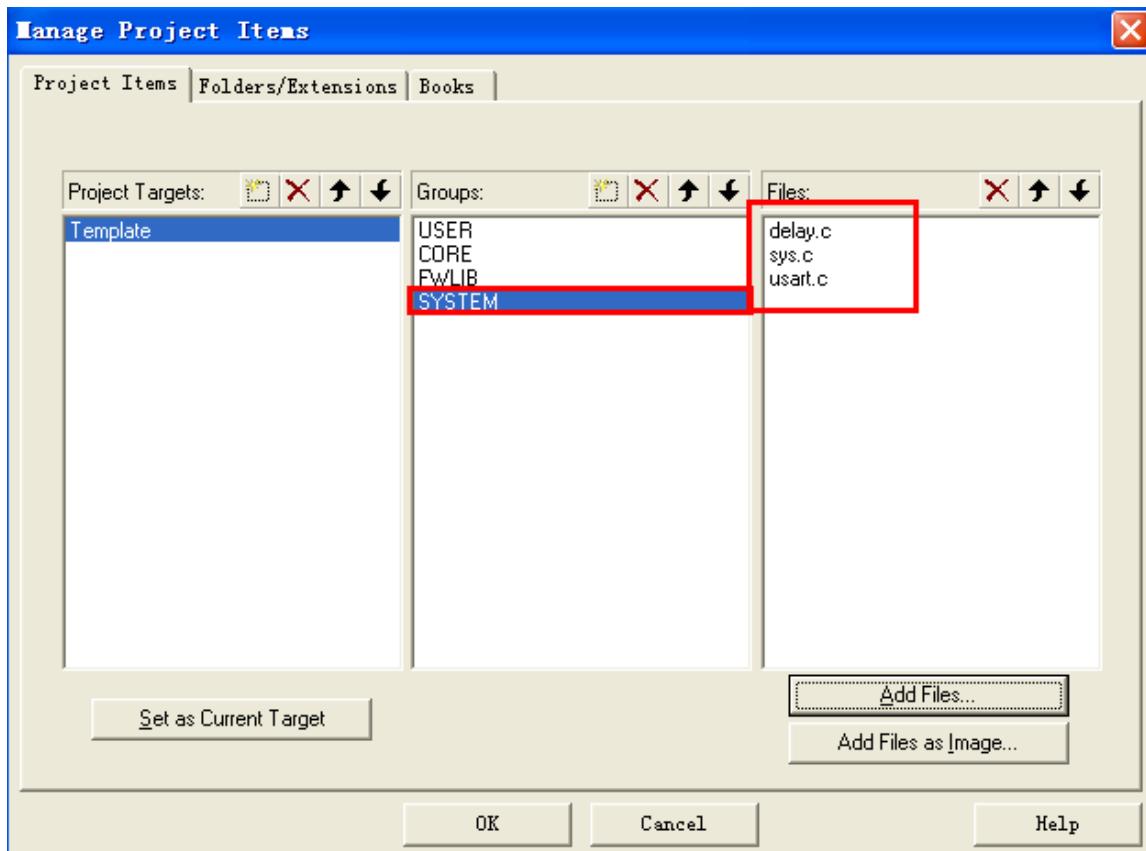


图 3.3.2.30 添加文件到 SYSTEM 分组

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。

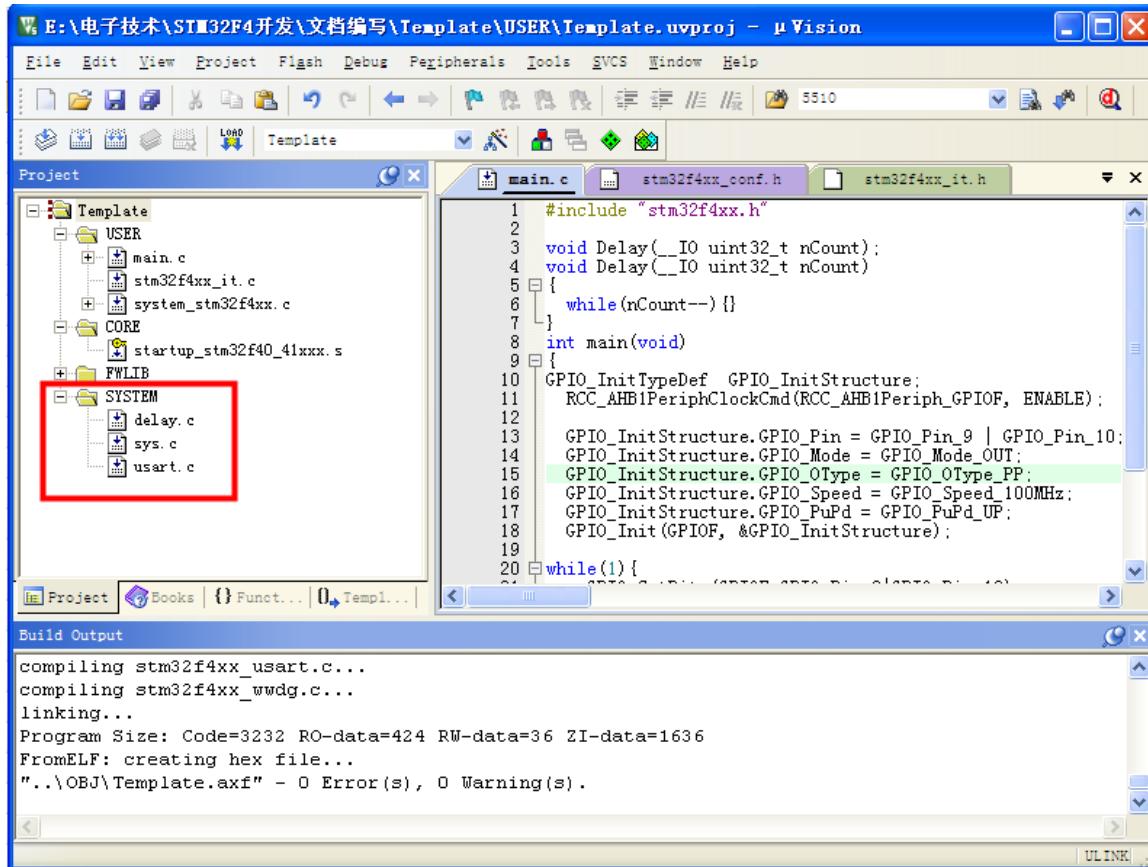


图 3.3.2.31 工程分组情况

接下来我们将对应的三个目录（sys,usart,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 12 即可，加入后的截图如下图 3.3.2.32 所示：

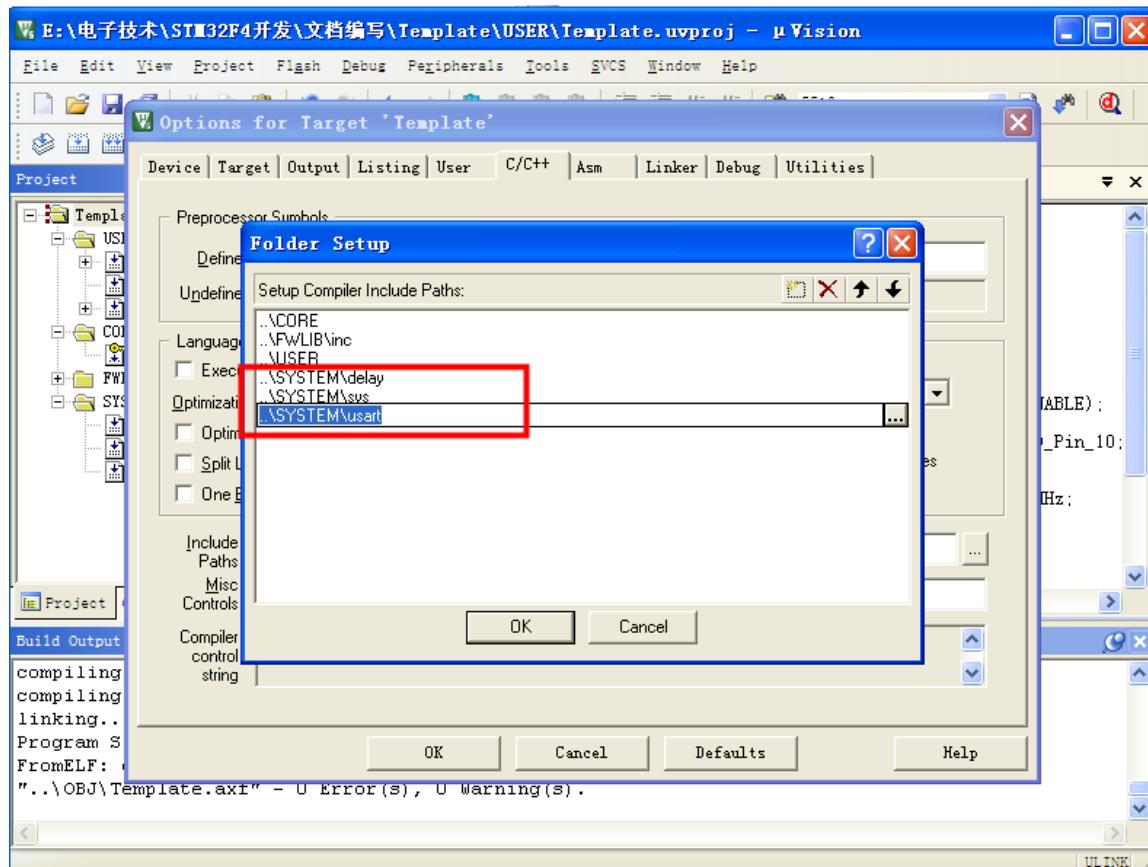


图 3.3.2.32 添加头文件路径到 PATH

最后点击 OK。这样我们的工程模板就彻底完成了。接下来我们修改主函数所在的文件 main.c 的内容，引入 ALIENTEK 提供的系统文件包里面的一些头文件和调用一些函数来测试，修改后的 main.c 文件内容为(下面这段代码大家可以打开我们光盘的模板复制即可):

```
#include "stm32f4xx.h"
#include "uart.h"
#include "delay.h"

int main(void)
{
    u32 t=0;
    uart_init(115200);
    delay_init(84);

    while(1){
        printf("t:%d\r\n",t);
        delay_ms(500);
        t++;
    }
}
```

修改后编译工程，我们会发现没有任何警告。我们建立好的工程模板在我们光盘的实验目录里面有，路径为“光盘目录：“4，程序源码\标准例程-库函数版本\实验 0 Template 工程模板”，

大家可以打开对照一下。完整的工程结构如下图 3.3.2.33:

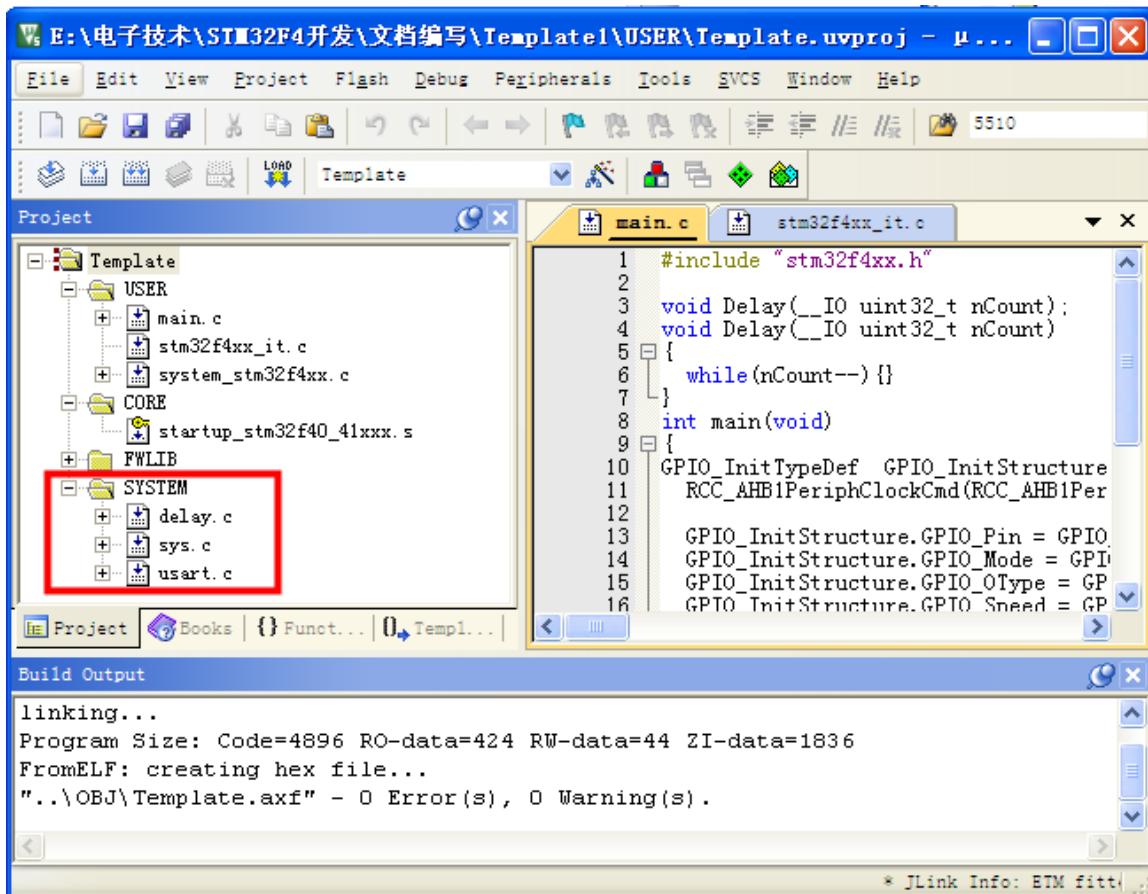


图 3.3.2.33 工程模板结构

工程编译通过之后，接下来我们会教大家怎么对 STM32F4 芯片进行程序下载。

3.4 程序下载与调试

上一节，我们学会了如何在 MDK 下创建 STM32F4 工程。本节，我们将向读者介绍 STM32F4 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、STM32F4 程序下载；2、利用 JLINK 对 STM32F4 进行下载与在线调试。

3.4.1 STM32 串口程序下载

STM32F4 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 STM32F4 下载代码。不过，最简单也是最经济的，就是通过串口给 STM32F4 下载代码。本节，我们将向大家介绍，如何利用串口给 STM32F4（以下简称 STM32）下载代码。

STM32 的串口下载一般是通过串口 1 下载的，本手册的实验平台 ALIENTEK 探索者 STM32F4 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。设置完成如图 3.4.1.1 所示：

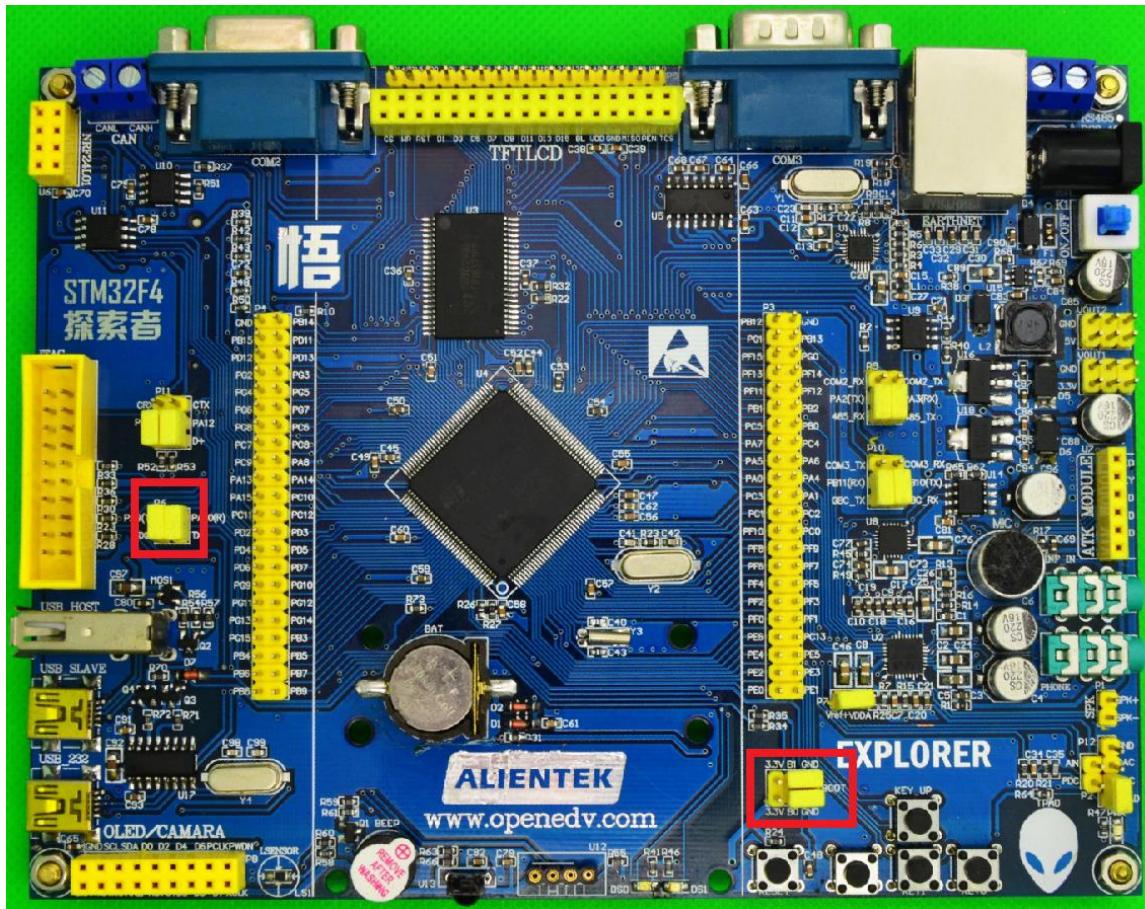


图 3.4.1.1 开发板串口下载跳线设置

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两个步骤：

- 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（flymcu，即 mcuisp 的最新版本），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），则需要先安装 CH340G 的驱动，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 3.4.1.2 所示：

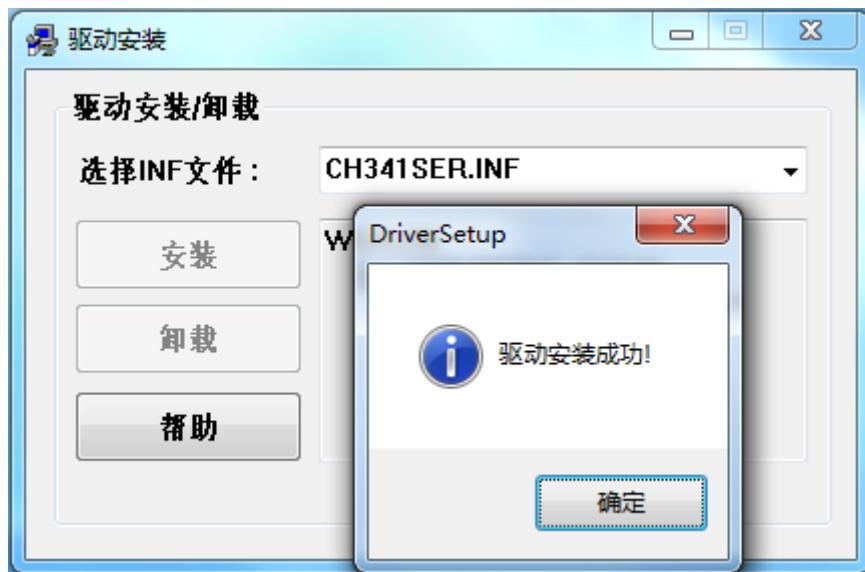


图 3.4.1.2 CH340 驱动安装

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 3.4.1.3 所示：

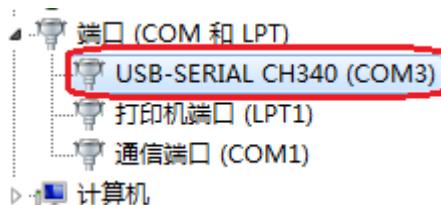


图 3.4.1.3 USB 串口

在图 3.4.1.3 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 flymcu，该软件是 mcuisp 的升级版本(flymcu 新增对 STM32F4 的支持)，由 ALIENTEK 提供部分赞助，mcuisp 作者开发，该软件可以在 www.mcuisp.com 免费下载，本手册的光盘也附带了这个软件，版本为 V0.188。该软件启动界面如图 3.4.1.4 所示：



图 3.4.1.4 flymcu 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 flymcu 软件打开 OBJ 文件夹，找到对应的 hex 文件 Template.hex，打开并进行相应设置后，如图 3.4.1.5 所示：

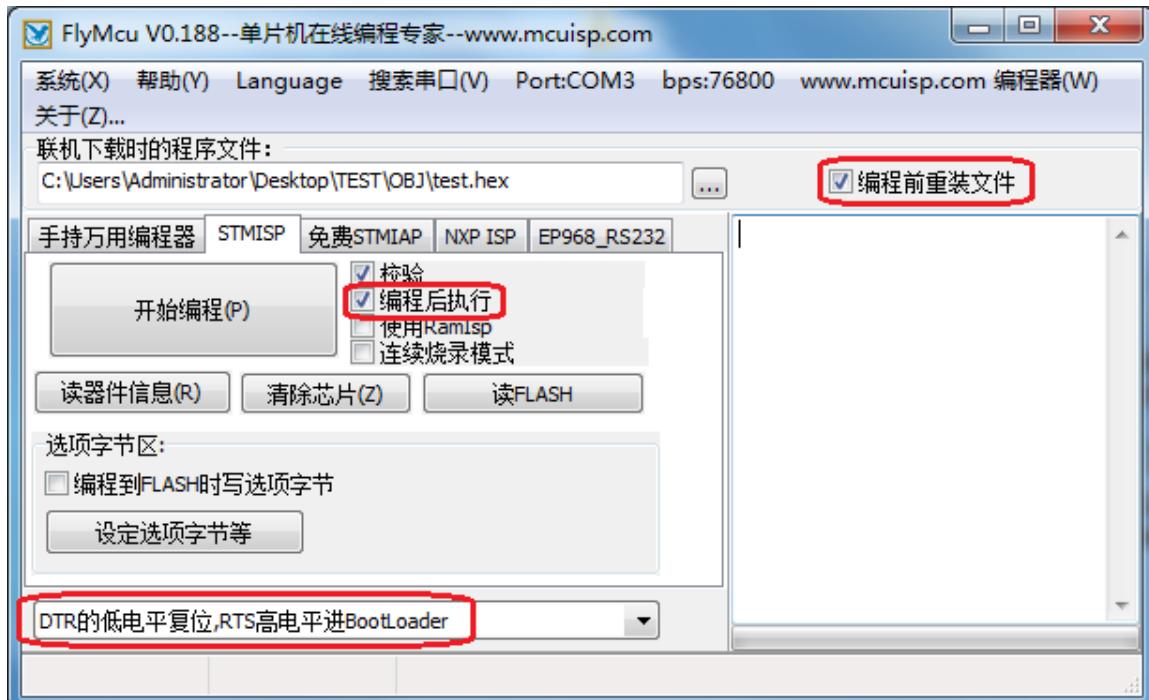


图 3.4.1.5 flymcu 设置

图 3.4.1.5 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，flymcu 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，flymcu 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 flymcu 有智能串口搜索功能。每次打开 flymcu 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择的串口）。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 STM32F4，由于 F4 自带的 bootlaoder 程序对高波特率支持不太好，所以，我们推荐设置波特率为 76800bps，高的波特率将导致极低的下载成功率。找到 CH340 虚拟的串口，如图 3.4.1.6 所示：

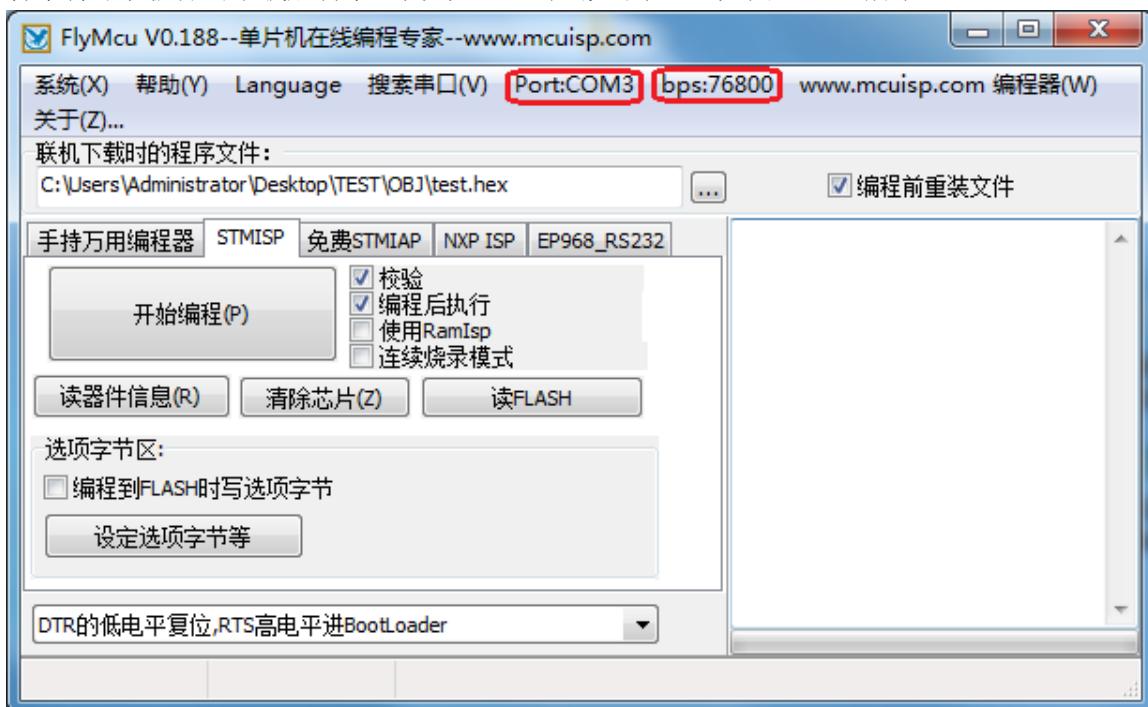


图 3.4.1.6 CH340 虚拟串口

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，波特率设置为 76800。设置好之后，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 3.4.1.7 所示：

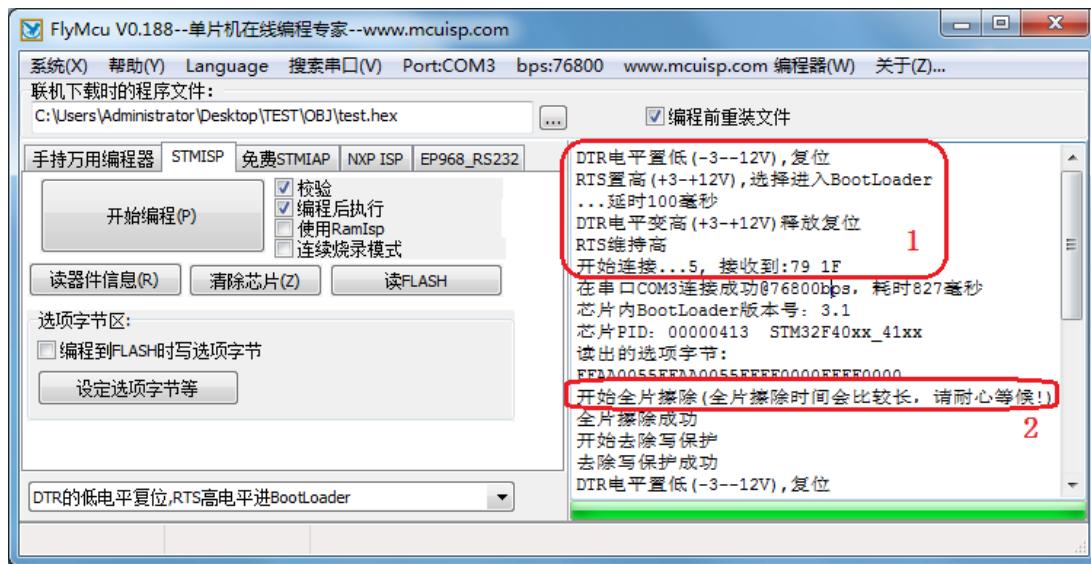


图 3.4.1.7 下载完成

图 3.4.1.7 中, 第 1 个圈, 圈出了 flymcu 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。第 2 个圈这里需要特别注意, 因为 STM32F4 的每次下载都需要整片擦除, 而 STM32F4 的整片擦除是非常慢的 (STM32F1 比较快), 这里的全片擦除, 得**等待几十秒钟**, 才可以执行完成, 请大家耐心等待。但是 JLINK 下载不存在这个问题, 所以, 我们建议, 有条件的话, **最好还是用 JLINK 下载比较快**。

另外, 下载成功后, 会有“共写入 xxxxKB, 耗时 xxxx 毫秒”的提示, 并且从 0X80000000 处开始运行了, 我们打开串口调试助手 (XCOM V2.0, 在光盘→6, 软件资料→软件→串口调试助手里面) 选择 COM3 (得根据你的实际情况选择), 设置波特率为 115200, 会发现从 ALIENTEK 探索者 STM32F4 开发板发回来的信息, 如图 3.4.1.8 所示:

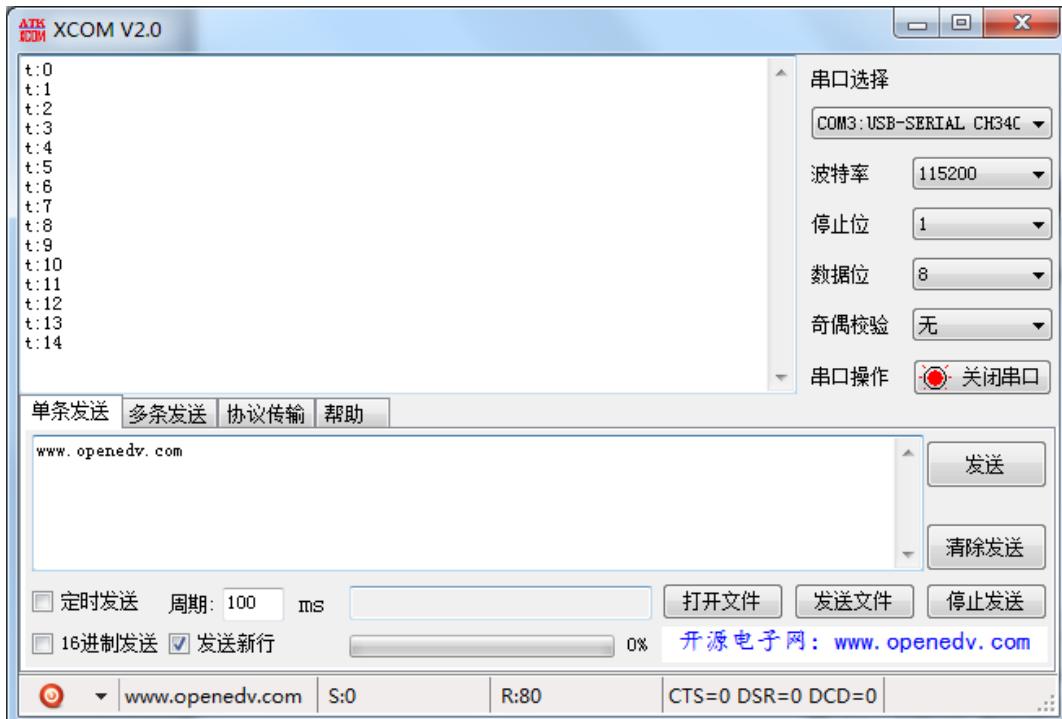


图 3.4.1.8 程序开始运行了

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

3.4.2 JLINK 下载与调试程序

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 探索者 STM32F4 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 STM32F4。

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。

首先，用 JLINK 进行下载与调试，大家要在硬件上，把 JLINK 用 USB 线连接到电脑 usb 和板子的 JTAG 接口上。

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 miniSTM32 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 J-LINK/J-TRACE Cortex，如图 3.4.2.1 所示：

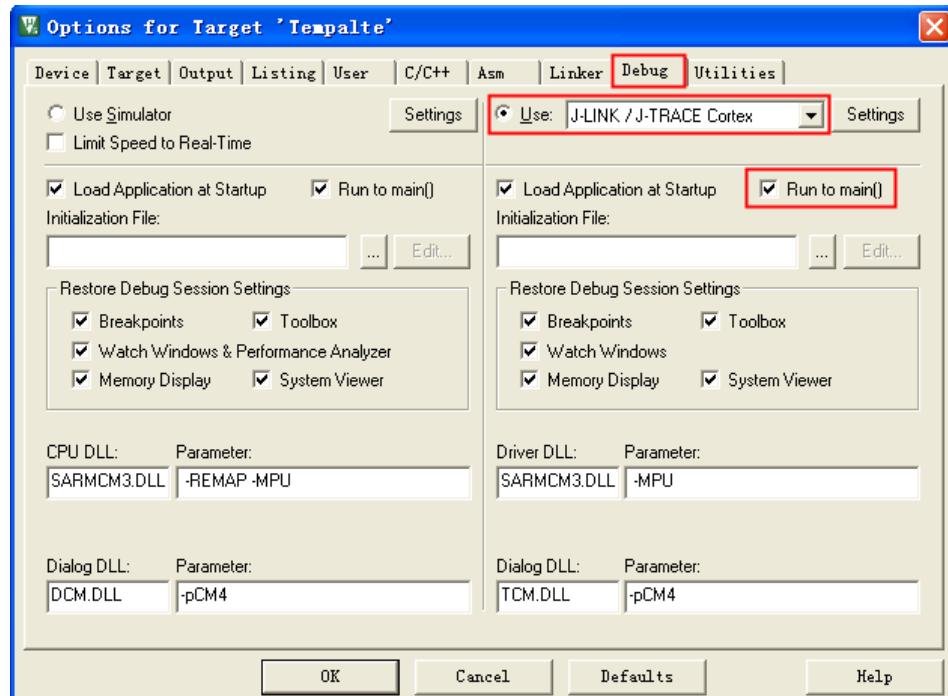


图 3.4.2.1 Debug 选项卡设置

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_stm32f40_41xxx.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 3.4.2.2 所示：

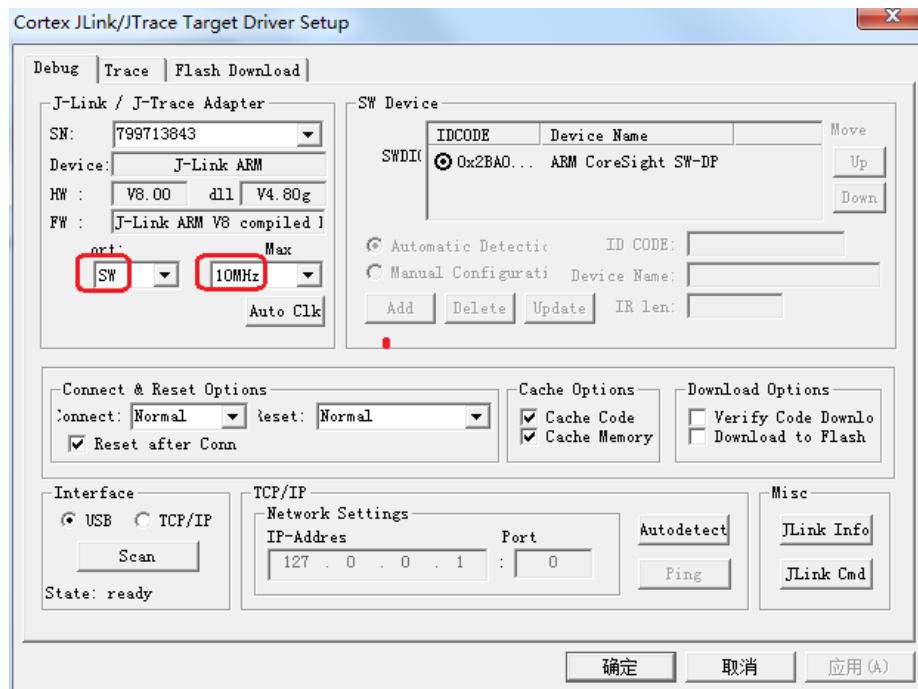


图 3.4.2.2 J-LINK 模式设置

图 3.4.2.2 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 miniSTM32 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 或者 5MHz，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器，如图 3.4.2.3 所示：

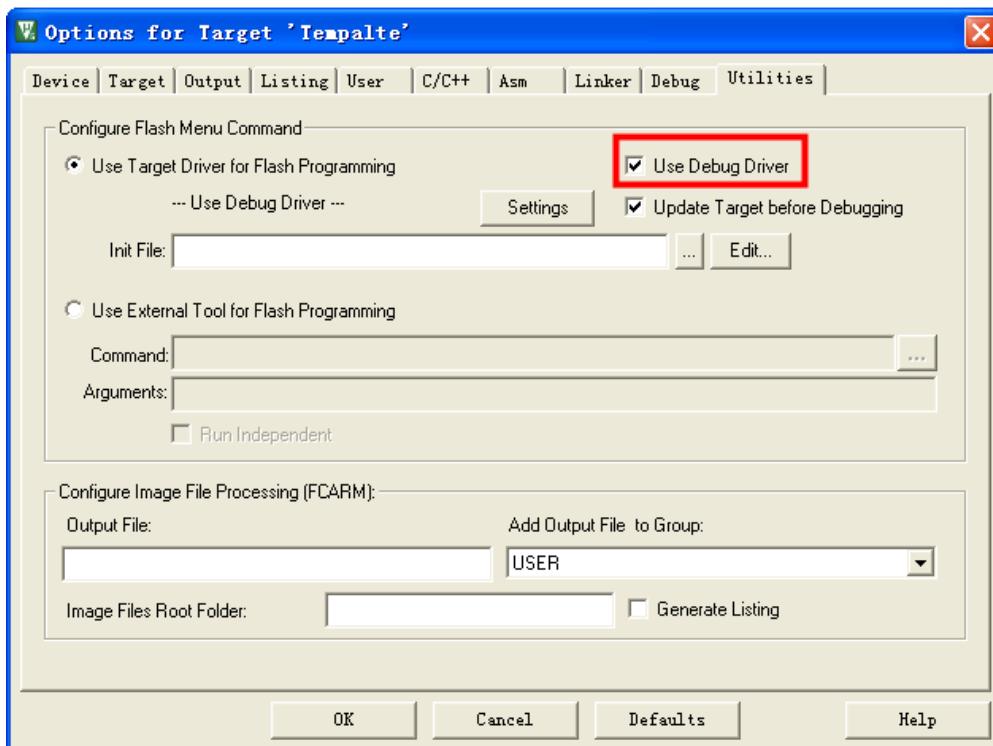


图 3.4.2.3 FLASH 编程器选择

图 3.4.2.3 中，我们直接勾选 Use Debug Driver，即和调试一样，选择 JLINK 来给目标器件的 FLASH 编程，然后点击 Settings，设置如图 3.4.2.4 所示：

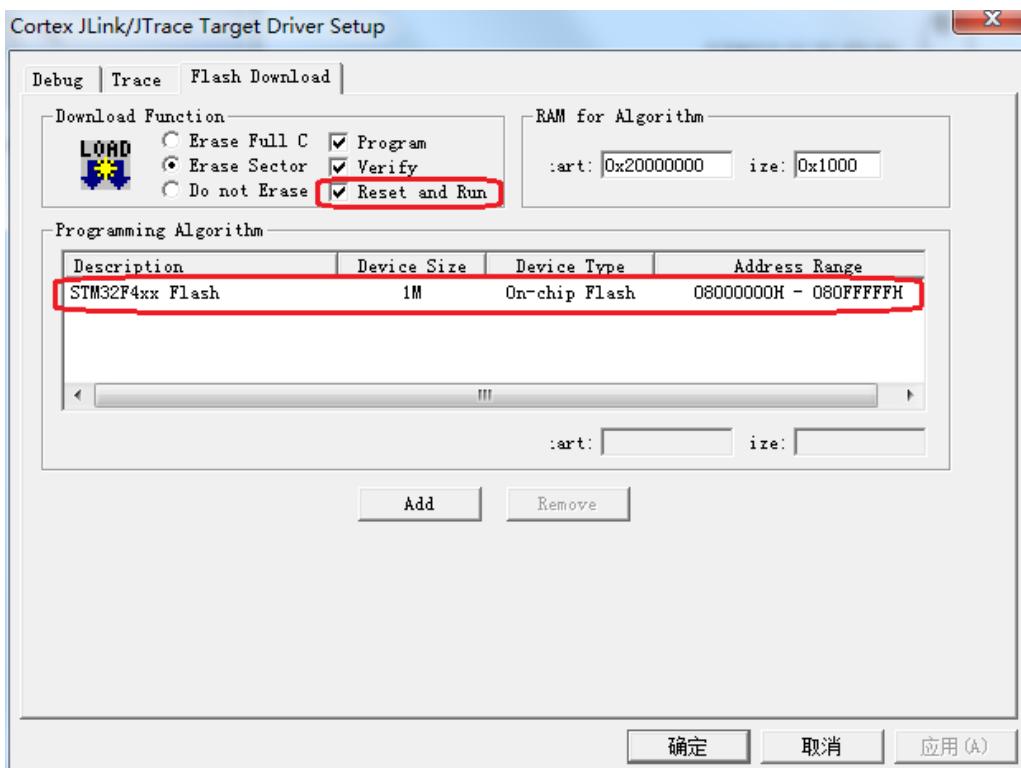


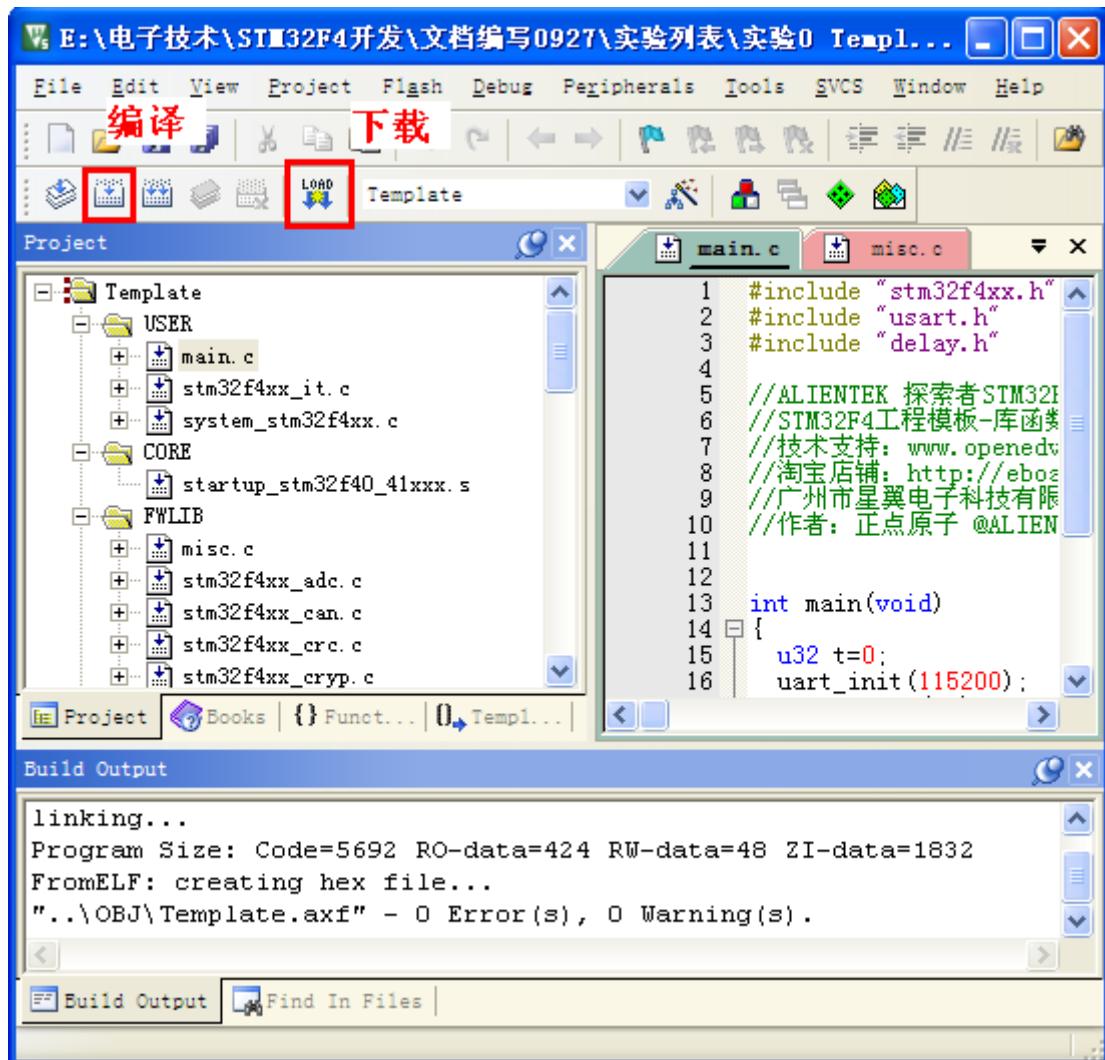
图 3.4.2.4 编程设置

这里 MDK5 会根据我们新建工程时选择的目标器件，自动设置 flash 算法。我们使用的是 STM32F407ZGT6，FLASH 容量为 1M 字节，所以 Programming Algorithm 里面默认会有 1M 型

号的 STM32F4xx FLASH 算法。**特别提醒：**这里的 1M flash 算法，不仅仅针对 1M 容量的 STM32F4，对于小于 1M FLASH 的型号，也是采用这个 flash 算法的。最后，选中 Reset and Run 选项，以实现在编程后自动运行，其他默认设置即可。设置完成之后，如图 3.4.2.4 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。接下来我们就可以通过 JLINK 下载代码和调试代码。

配置好 JLINK 之后，使用 JLINK 下载代码就非常简单，大家只需要点击 LOAD 按钮就可以进行程序下载。下载完成之后程序就可以直接在开发板执行。如图 3.4.2.5：



如图 3.4.2.5

接下来我们看看用 JLINK 进行程序仿真。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 3.4.2.6 所示：

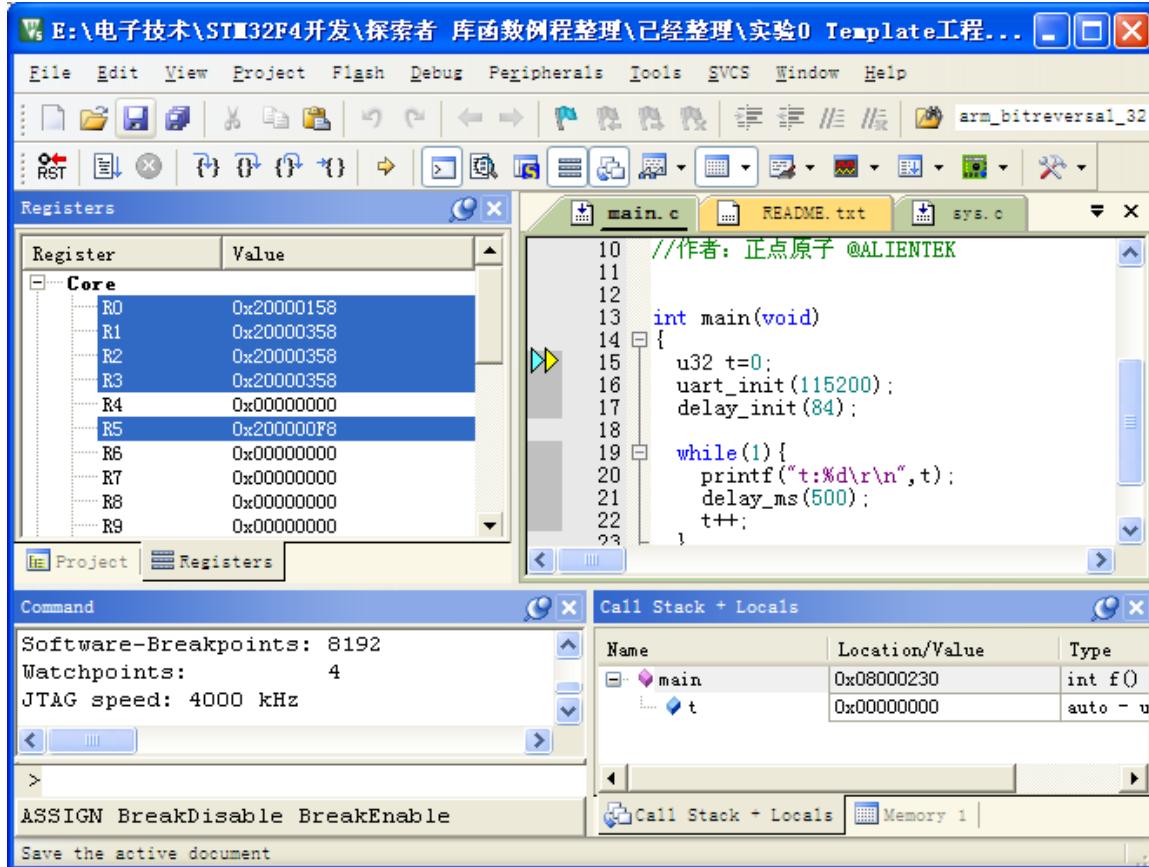


图 3.4.2.6 开始仿真

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 3.4.2.7 所示：

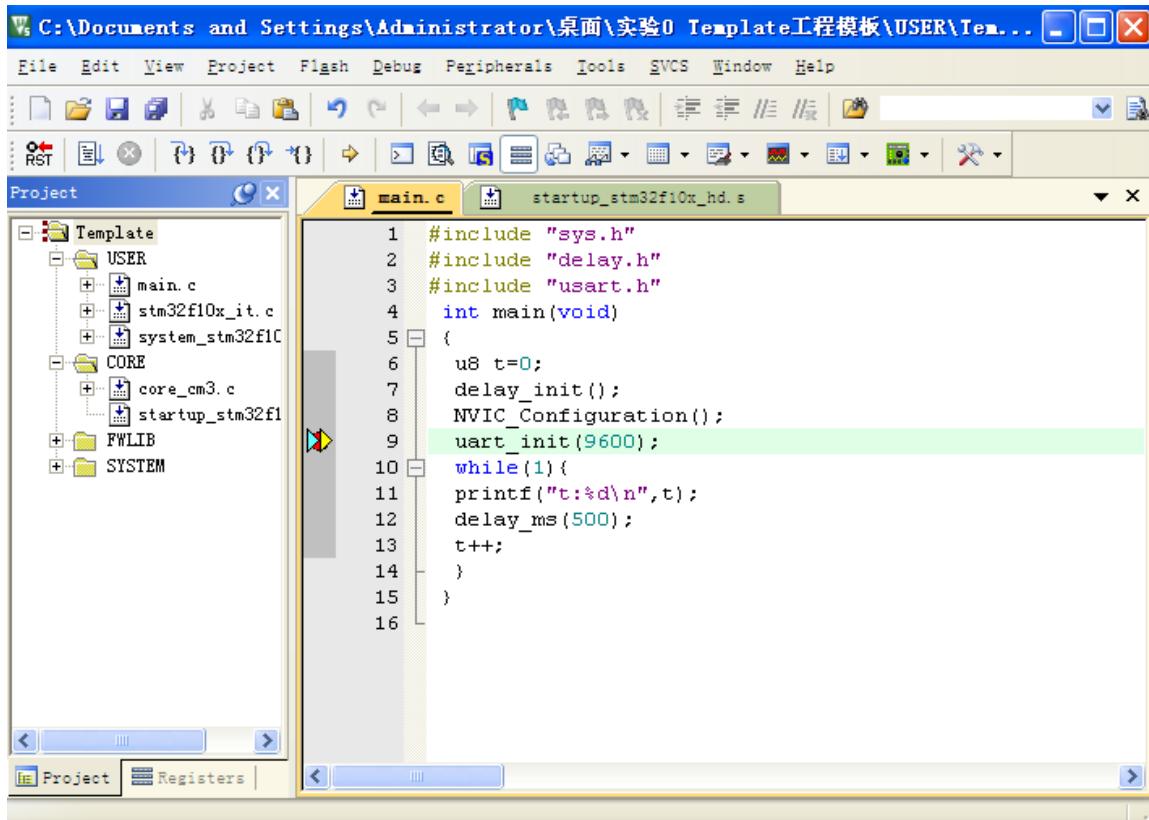


图 3.4.2.7 程序运行到断点处

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。

另外，此时 MDK 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 3.4.2.8 所示：



图 3.4.2.8 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，前提是我在查看的地方设置了断点。

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

执行进去: 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。

堆栈局部变量窗口：通过该按钮，显示 Call Stack+Locals 窗口，显示当前函数的局部变量及其值，方便查看。

观察窗口：MDK5 提供 2 个观察窗口（下拉选择），该按钮按下，会弹出一个显示变量的窗口，输入你所想要观察的变量/表达式，即可查看其值，是很常用的一个调试窗口。

内存查看窗口：MDK5 提供 4 个内存查看窗口（下拉选择），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

串口打印窗口：MDK5 提供 4 个串口打印窗口（下拉选择），该按钮按下，会弹出一个类似串口调试助手界面的窗口，用来显示从串口打印出来的内容。

逻辑分析窗口：该图标下面有 3 个选项（下拉选择），我们一般用第一个，也就是逻辑分析窗口(Logic Analyzer)，点击即可调出该窗口，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。

系统查看窗口：该按钮可以提供各种外设寄存器的查看窗口（通过下拉选择），选择对应外设，即可调出该外设的相关寄存器表，并显示这些寄存器的值，方便查看设置的是否正确。

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。

特别注意：串口打印窗口和逻辑分析窗口仅在软件仿真的时候可用，而 MDK5 对 STM32F4 的软件仿真，基本上不支持（故本教程直接没有对软件仿真进行介绍了），所以，基本上这两个窗口用不着。但是对 STM32F1 的软件仿真，MDK5 是支持的，在 F1 开发的时候，可以用到。

这样，我们在上面的仿真界面里面调出：堆栈局部变量窗口。如图 3.4.2.9 所示：

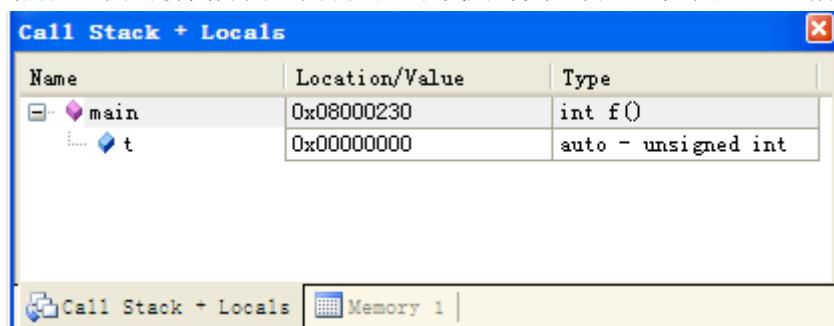
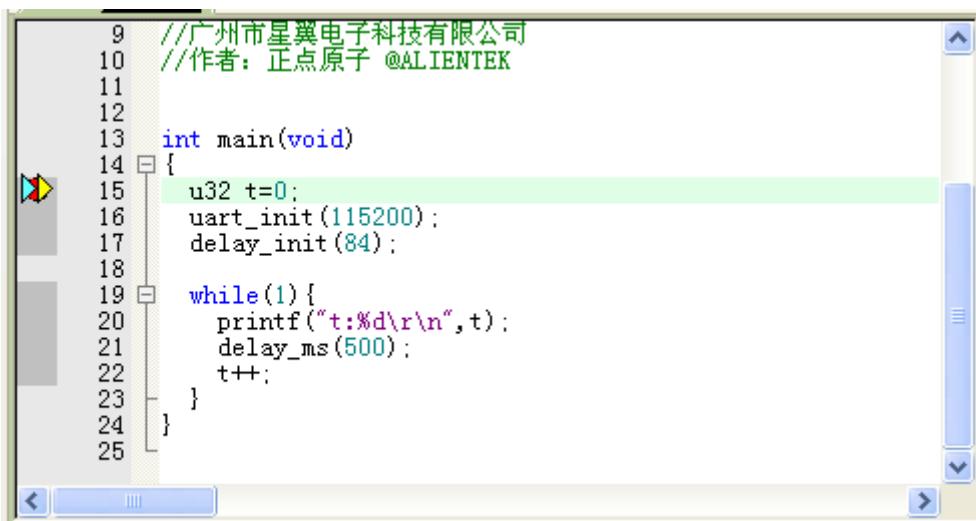


图 3.4.2.9 堆栈局部变量查看窗口

我们把光标放到 main.c 的第 15 行左侧的灰色区域，然后按下鼠标左键，即可放置一个断点（红色的实心点，也可以通过鼠标右键弹出菜单来加入），再次单击则取消。然后点击 ，执行到该断点处，如图 3.4.2.10 所示



```

9 //广州市星翼电子科技有限公司
10 //作者: 正点原子 @ALIENTEK
11
12
13 int main(void)
14 {
15     u32 t=0;
16     uart_init(115200);
17     delay_init(84);
18
19     while(1) {
20         printf("t:%d\r\n",t);
21         delay_ms(500);
22         t++;
23     }
24 }
25

```

图 3.4.2.10 执行到断点处

现在先不忙着往下执行, 点击菜单栏的 Peripherals→System Viewer→USART→USART1。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 3.4.2.11 所示:

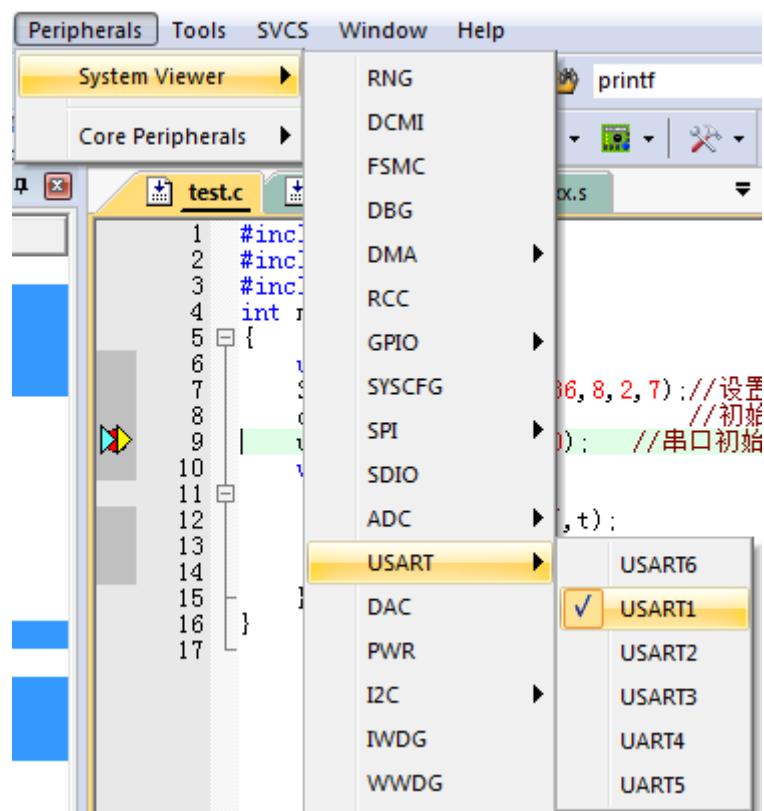


图 3.4.2.11 查看串口 1 相关寄存器

单击 USART1 后会在 IDE 右侧出现一个如图 3.4.2.12 (a) 所示的界面:

The figure consists of two side-by-side windows titled "USART1".

(a) USART1 Settings Before Initialization:

Property	Value
SR	0
DR	0
DR	0x0000
BRR	0
DIV_Mantissa	0x0000
DIV_Fraction	0x00
CR1	0
OVER8	<input type="checkbox"/>
UE	<input type="checkbox"/>
M	<input type="checkbox"/>
WAKE	<input type="checkbox"/>
PCE	<input type="checkbox"/>
PS	<input type="checkbox"/>
PEIE	<input type="checkbox"/>
TXEIE	<input type="checkbox"/>
TCIE	<input type="checkbox"/>
RXNEIE	<input type="checkbox"/>
IDLEIE	<input type="checkbox"/>
TE	<input type="checkbox"/>
RE	<input type="checkbox"/>
RWU	<input type="checkbox"/>
SBK	<input type="checkbox"/>
CR2	0
CR3	0
GTPR	0

(b) USART1 Settings After Initialization:

Property	Value
SR	0x000000C0
DR	0
DR	0x0000
BRR	0x000002D9
DIV_Mantissa	0x002D
DIV_Fraction	0x09
CR1	0x0000202C
OVER8	<input type="checkbox"/>
UE	<input checked="" type="checkbox"/>
M	<input type="checkbox"/>
WAKE	<input type="checkbox"/>
PCE	<input type="checkbox"/>
PS	<input type="checkbox"/>
PEIE	<input type="checkbox"/>
TXEIE	<input type="checkbox"/>
TCIE	<input type="checkbox"/>
RXNEIE	<input checked="" type="checkbox"/>
IDLEIE	<input type="checkbox"/>
TE	<input checked="" type="checkbox"/>
RE	<input checked="" type="checkbox"/>
RWU	<input type="checkbox"/>
SBK	<input type="checkbox"/>
CR2	0
CR3	0
GTPR	0

3.4.2.12 串口 1 各寄存器初始化前后对比

图 3.4.2.12 (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。我们接着单击一下 ，执行完串口初始化函数，得到了如图 3.4.2.12 (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(115200);` 这个函数里面大概执行了哪些操作。

通过图 3.4.2.12 (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。

此时，我们先打开串口调试助手（XCOM V2.0，在光盘→6，软件资料→软件→串口调试助手里面）设置好串口号和波特率，然后我们继续单击 按钮，一步步执行，此时在堆栈局部变量窗口可以看到 t 的值变化，同时在串口调试助手中，也可看到打印出 t 的值，如图 3.4.2.13 和 3.4.2.14 所示：

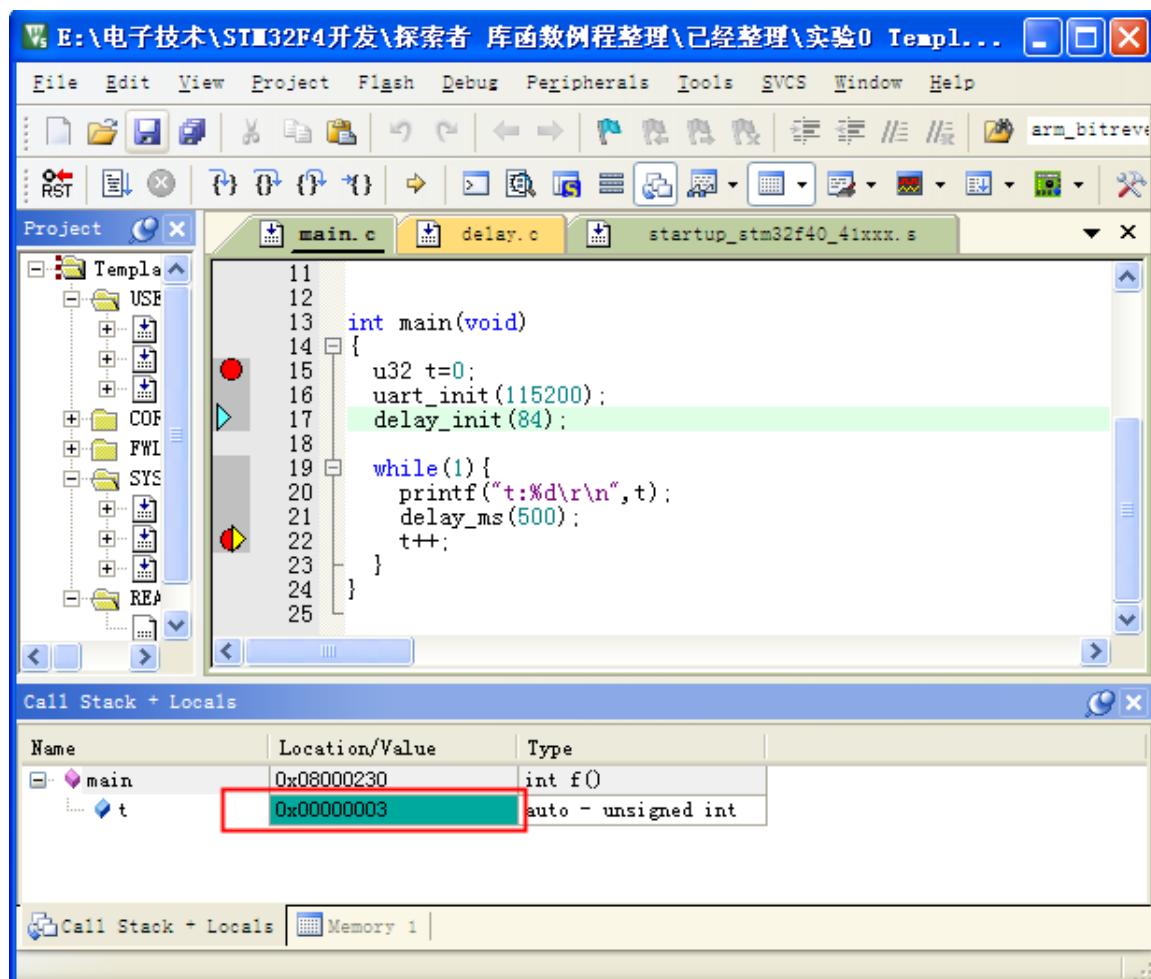


图 3.4.2.13 堆栈局部变量窗口查看 t 的值

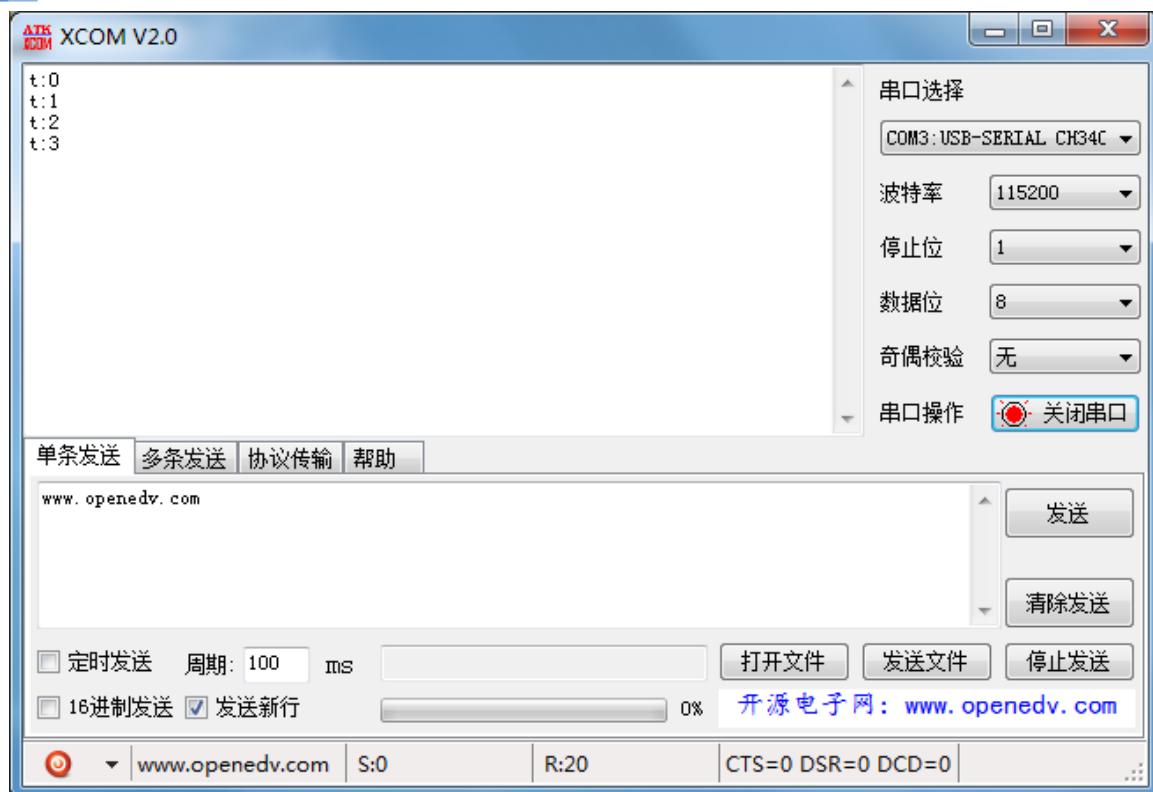


图 3.4.2.14 串口调试助手收到的数据

关于 STM32F4 的硬件调试，我们就介绍到这里，这仅仅是一个简单的 demo 演示，在实际使用中，硬件调试更是大有用处，所以大家一定要好好掌握。

3.5 MDK5 使用技巧

通过前面的学习，我们已经了解了如何在 MDK5 里面建立属于自己的工程。下面，我们将向大家介绍 MDK5 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 MDK5 新建工程的时候看到界面如图 3.2.22 所示，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 (配置对话框) 弹出如图 3.5.1.1 所示界面：

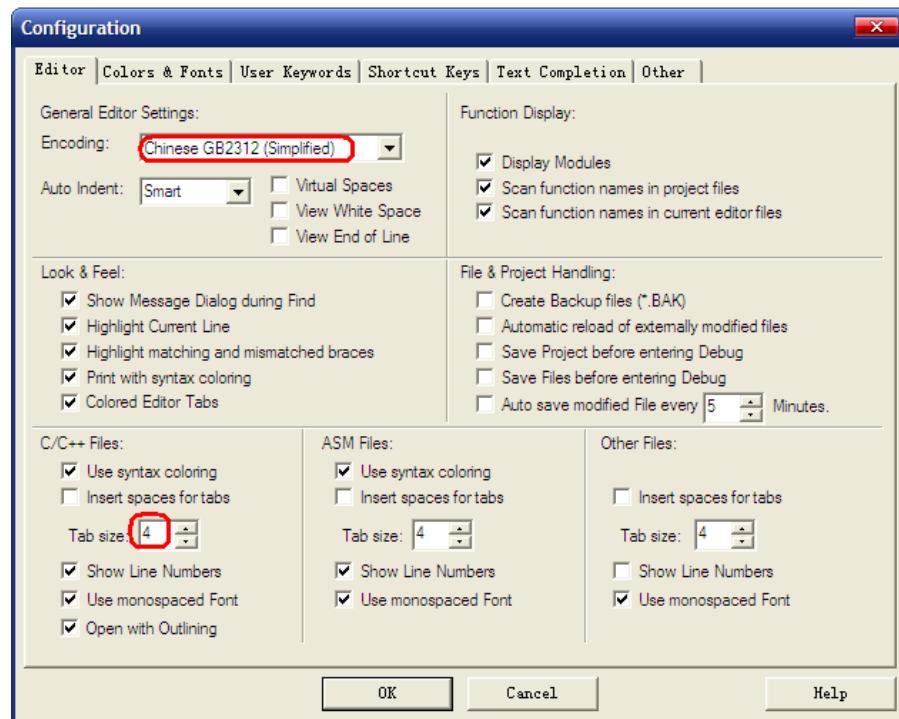


图 3.5.1.1 置对话框

在该对话框中，先设置 Encoding 为:Chinese GB2312(Simplified)，然后设置 Tab size 为: 4。以更好的支持简体中文（否则，拷贝到其他地方的时候，中文可能是一堆的问号），同时 TAB 间隔设置为 4 个单位。然后，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字体和颜色了。由于我们使用的是 C 语言，故在 Window 下面选择:C/C++ Editor Files 在右边就可以看到相应的元素了。如图 3.5.1.2 示：

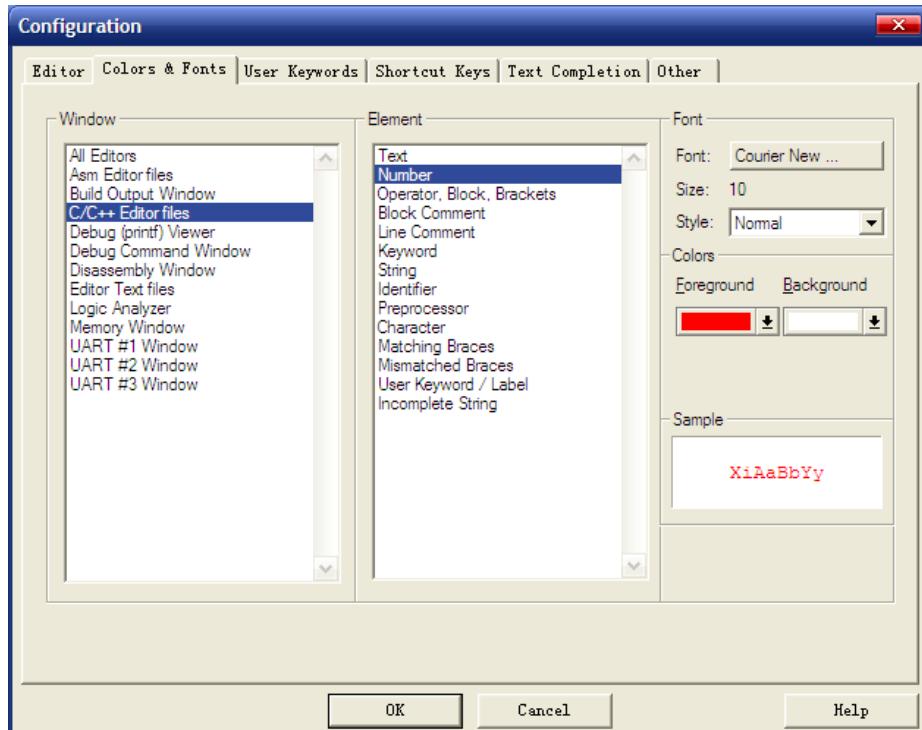


图 3.5.1.2 Colors&Fonts 选项卡

然后点击各个元素修改为你喜欢的颜色（注意双击，且有时候可能需要设置多次才生效，MDK 的 bug），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 3.5.1.3 示：

```

11
12
13 int main(void)
14 {
15     u32 t=0;
16     uart_init(115200);
17     delay_init(84);
18
19     while(1){
20         printf("t:%d\r\n",t);
21         delay_ms(500);
22         t++;
23     }
24 }
25

```

图 3.5.1.3 设置完后显示效果

这就比开始的效果好看一些了。字体大小，则可以直接按住：ctrl+鼠标滚轮，进行放大或者缩小，或者也可以在刚刚的配置界面设置字体大小。

细心的读者可能会发现，上面的代码里面有一个 u32，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：C/C++ Editor 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 3.5.1.4 示：

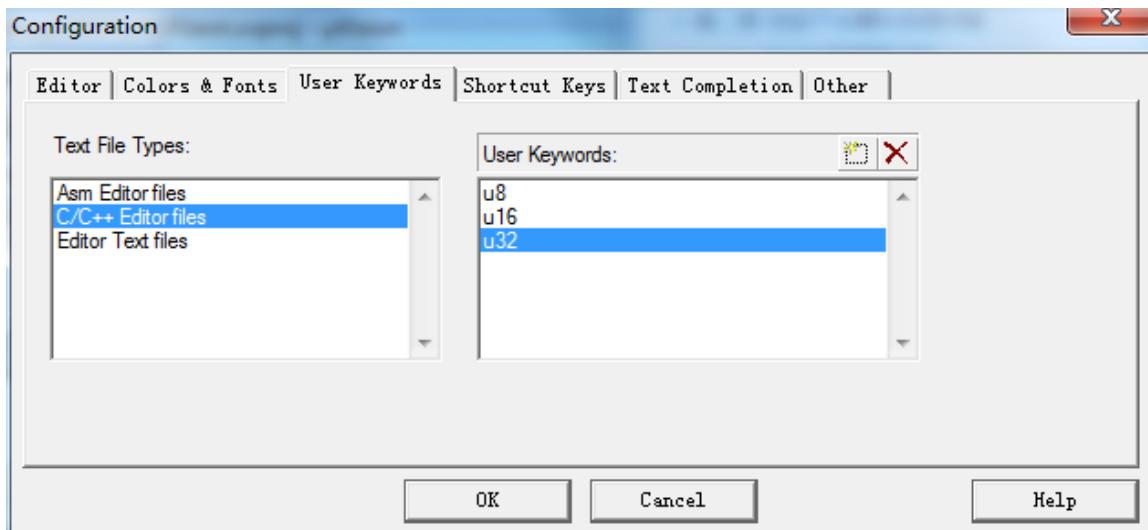


图 3.5.1.4 用户自定义关键字

图 3.5.1.5 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 3.3.1.5 示：

```

11
12
13 int main(void)
14 {
15     u32 t=0;
16     uart_init(115200);
17     delay_init(84);
18
19     while(1){
20         printf("t:%d\r\n",t);
21         delay_ms(500);
22         t++;
23     }
24 }
25

```

图 3.5.1.5 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，比如动态语法检测等，我们将 3.5.2 节介绍。

3.5.2 语法检测&代码提示

MDK4.70 以上的版本，新增了代码提示与动态语法检测功能，使得 MDK 的编辑器越来越好用了，这里我们简单说一下如何设置，同样，点击 ，打开配置对话框，选择 Text Completion 选项卡，如图 3.5.2.1 所示：

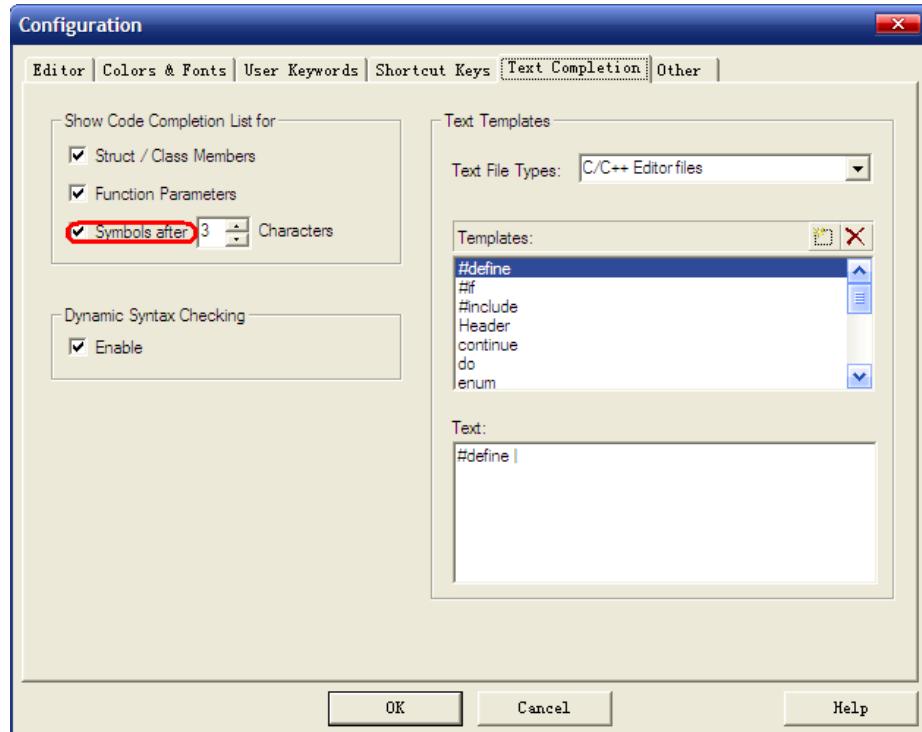


图 3.5.2.1 Text Completion 选项卡设置

Strut/Class Members，用于开启结构体/类成员提示功能。

Function Parameters，用于开启函数参数提示功能。

Symbols after xx characters，用于开启代码提示功能，即在输入多少个字符以后，提示匹配

的内容(比如函数名字、结构体名字、变量名字等),这里默认设置3个字符以后,就开始提示。如图3.5.2.2所示:

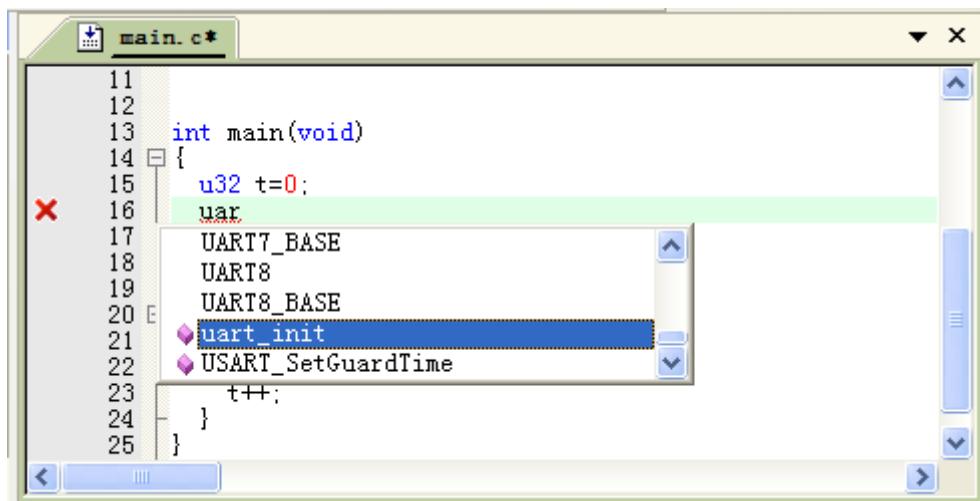


图 3.5.2.2 代码提示

Dynamic Syntax Checking,则用于开启动态语法检测,比如编写的代码存在语法错误的时候,会在对应行前面出现 \times 图标,如出现警告,则会出现 $!$ 图标,将鼠标光标放图标上面,则会提示产生的错误/警告的原因,如图3.5.2.3所示:

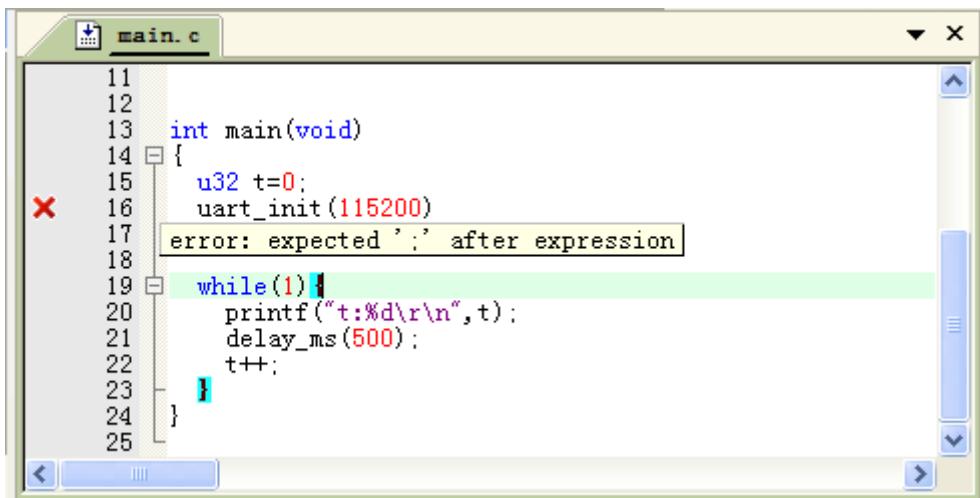


图 3.5.2.3 语法动态检测功能

这几个功能,对我们编写代码很有帮助,可以加快代码编写速度,并且及时发现各种问题。不过这里要提醒大家,语法动态检测这个功能,有的时候会误报(比如sys.c里面,就有很多误报),大家可以不用理会,只要能编译通过(0错误,0警告),这样的语法误报,一般直接忽略即可。

3.5.3 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 3.5.3.1 所示：

```

74 void USART1_IRQHandler(void)
75 {
76     u8 res;
77     #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
78     OSIntEnter();
79     #endif
80     if(USART1->SR&(1<<5))//接收到数据
81     {
82         res=USART1->DR;
83         if((USART_RX_STA&0x8000)==0)//接收未完成
84         {
85             if(USART_RX_STA&0x4000)//接收到了0x0d
86             {
87                 if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
88                 else USART_RX_STA|=0x8000; //接收完成了
89             }else //还没收到0x0d
90             {
91                 if(res==0x0d)USART_RX_STA|=0x4000;
92             }
93         }
94         USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
95         USART_RX_STA++;
96         if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
97     }
98 }
99 }
100 }
101 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
102 OSIntExit();
103 #endif
104 }
```

图 3.5.3.1 头大的代码

图 3.5.3.1 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 3.5.3.2 所示：

```

74 void USART1_IRQHandler(void)
75 {
76     u8 res;
77     #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
78     OSIntEnter();
79     #endif
80     if(USART1->SR&(1<<5))//接收到数据
81     {
82         res=USART1->DR;
83         if((USART_RX_STA&0x8000)==0)//接收未完成
84         {
85             if(USART_RX_STA&0x4000)//接收到了0x0d
86             {
87                 if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
88                 else USART_RX_STA|=0x8000; //接收完成了
89             }else //还没收到0x0d
90             {
91                 if(res==0x0d)USART_RX_STA|=0x4000;
92             }
93         }
94         USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
95         USART_RX_STA++;
96         if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
97     }
98 }
99 }
100 }
101 #ifdef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
102 OSIntExit();
103 #endif
104 }
```

图 3.5.3.2 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 3.5.3.3 所示

```

74 void USART1_IRQHandler(void)
75 {
76     u8 res;
77     #ifndef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
78     OSIntEnter();
79     #endif
80     if(USART1->SR&(1<<5))//接收到数据
81     {
82         res=USART1->DR;
83         if((USART_RX_STA&0x8000)==0)//接收未完成
84         {
85             if(USART_RX_STA&0x4000)//接收到了0x0d
86             {
87                 if(res!=0xa)USART_RX_STA=0;//接收错误,重新开始
88                 else USART_RX_STA|=0x8000; //接收完成了
89             }else //还没收到0x0d
90             {
91                 if(res==0xd)USART_RX_STA|=0x4000;
92                 else
93                 {
94                     USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
95                     USART_RX_STA++;
96                     if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
97                 }
98             }
99         }
100    #ifndef OS_CRITICAL_METHOD //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
101    OSIntExit();
102    #endif
103 }
104 }
```

图 3.5.3.3 修改后的代码

图 3.5.3.3 中的代码相对于图 3.5.3.1 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。只要你把光标放到这个函数/变量（xxx）的上面（xxx 为你想要查看的函数或变量的名字），然后右键，弹出如图 3.5.3.4 所示的菜单栏：

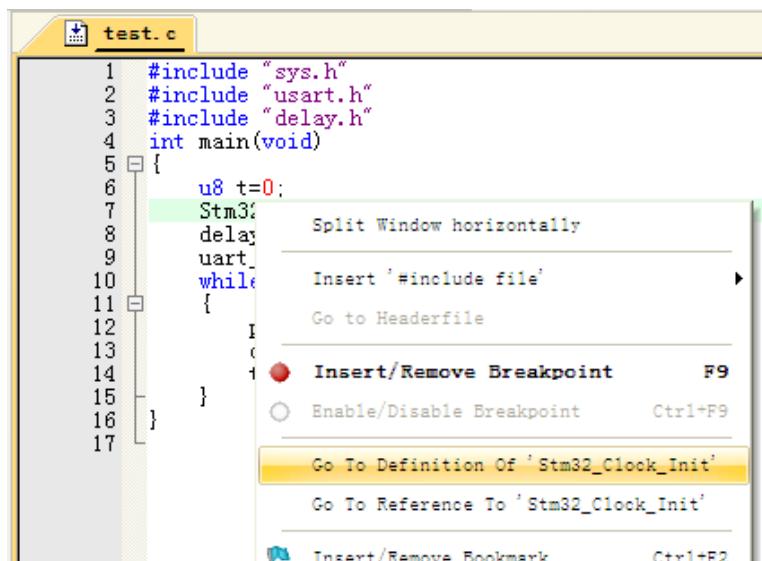


图 3.5.3.4 快速定位

在图 3.5.3.4 中，我们找到 Go to Definition Of ‘STM32_Clock_Init’ 这个地方，然后单击左键就可以快速跳到 STM32_Clock_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 3.5.3.5 所示：

```

175 //系统时钟初始化函数
176 //pll:选择的倍频数，从2开始，最大值为16
177 void Stm32_Clock_Init(u8 PLL)
178 {
179     unsigned char temp=0;
180     MYRCC_DeInit(); //复位并配置向量表
181     RCC->CR|=0x00010000; //外部高速时钟使能HSEON
182     while(!(RCC->CR>>17)); //等待外部时钟就绪
183     RCC->CFGR=0X00000400; //APB1=DIV2;APB2=DIV1;AHB=DIV1;
184     PLL-=2;//抵消2个单位
185     RCC->CFGR|=PLL<<18; //设置PLL值 2^16
186     RCC->CFGR|=1<<16; //PLLSRC ON
187     FLASH->ACR|=0x32; //FLASH 2个延时周期
188
189     RCC->CR|=0x01000000; //PLлон
190     while(!(RCC->CR>>25)); //等待PLL锁定
191     RCC->CFGR|=0x00000002; //PLL作为系统时钟
192     while(temp!=0x02) //等待PLL作为系统时钟设置成功
193     {
194         temp=RCC->CFGR>>2;
195         temp&=0x03;
196     }
197 }
```

图 3.5.3.5 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 ‘STM32_Clock_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 Stm32_Clock_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 3.5.3.6 所示：

```

175 //系统时钟初始化函数
176 //pll:选择的倍频数,从2开始,最大值为16
177 void Stm32_Clock_Init(u8 PLL)
178 {
179     unsigned char temp=0;
180     MYRCC_DeInit();           //复位并配置向量表
181     RCC->CR|=0x00010000;    //外部高速时钟使能HSEON
182     while(!(RCC->CR>>17)); //等待外部时钟就绪
183     RCC->CFGGR=0X00000400; //APB1=DIV2;APB2=DIV1;AHB=DIV1;
184     PLL=2;//抵消2个单位
185     RCC->CFGGR|=PLL<<18; //设置PLL值 2~16
186     RCC->CFGGR|=1<<16;   //PLLSRC ON
187     FLASH->ACR|=0x32;     //FLASH 2个延时周期
188
189     RCC->CR|=0x01000000; //PLLON
190     while(!(RCC->CR>>25)); //等待PLL锁定
191     RCC->CFGGR|=0x00000002; //PLL作为系统时钟
192     while(temp!=0x02)        //等待PLL作为系统时钟设置成功
193     {
194         temp=RCC->CFGGR>>2;
195         temp&=0x03;
196     }
197 }

```

图 3.5.3.6 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 3.5.3.7 所示:

```

175 //系统时钟初始化函数
176 //pll:选择的倍频数,从2开始,最大值为16
177 void Stm32_Clock_Init(u8 PLL)
178 {
179     // unsigned char temp=0;
180     // MYRCC_DeInit();           //复位并配置向量表
181     // RCC->CR|=0x00010000;    //外部高速时钟使能HSEON
182     // while(!(RCC->CR>>17)); //等待外部时钟就绪
183     // RCC->CFGGR=0X00000400; //APB1=DIV2;APB2=DIV1;AHB=DIV1;
184     // PLL=2;//抵消2个单位
185     // RCC->CFGGR|=PLL<<18; //设置PLL值 2~16
186     // RCC->CFGGR|=1<<16;   //PLLSRC ON
187     // FLASH->ACR|=0x32;     //FLASH 2个延时周期
188
189     // RCC->CR|=0x01000000; //PLLON
190     // while(!(RCC->CR>>25)); //等待PLL锁定
191     // RCC->CFGGR|=0x00000002; //PLL作为系统时钟
192     // while(temp!=0x02)        //等待PLL作为系统时钟设置成功
193     //
194     //     temp=RCC->CFGGR>>2;
195     //     temp&=0x03;
196     //
197 }

```

图 3.5.3.7 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。

3.5.4 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 3.5.4.1 所示:

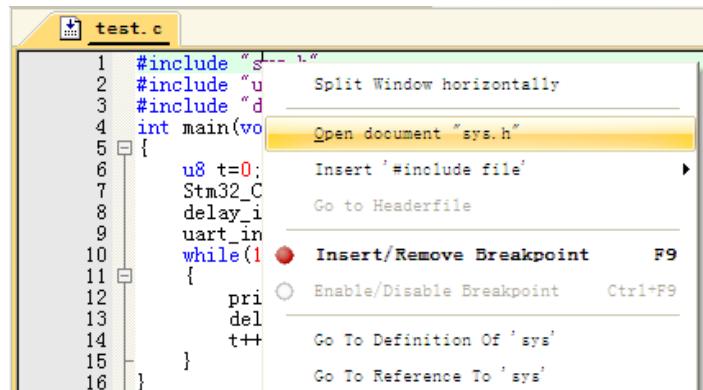


图 3.5.4.1 快速打开头文件

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 3.5.4.2 所示界面：

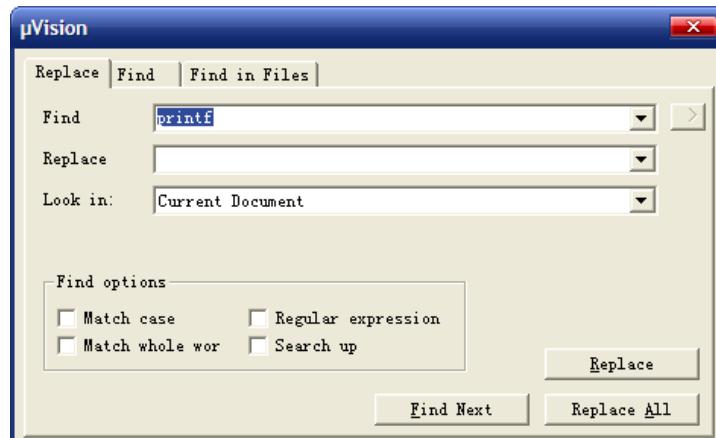


图 3.5.4.2 替换文本

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，这里就不啰嗦了。

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：Stm32_Clock_Init 为例），然后再点击 IDE 上面的 ，弹出如图 3.5.4.3 所示对话框：

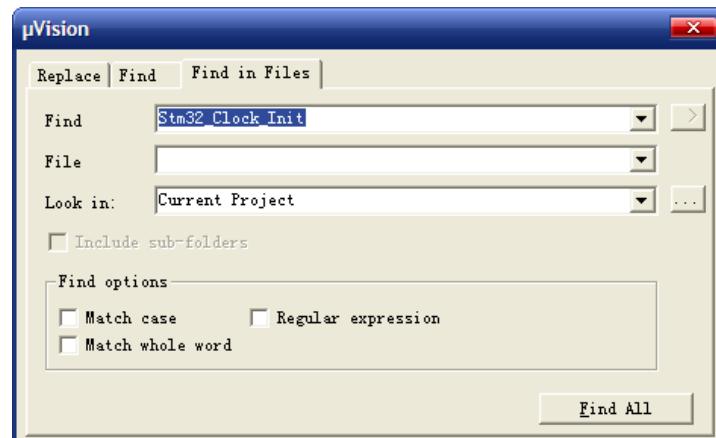
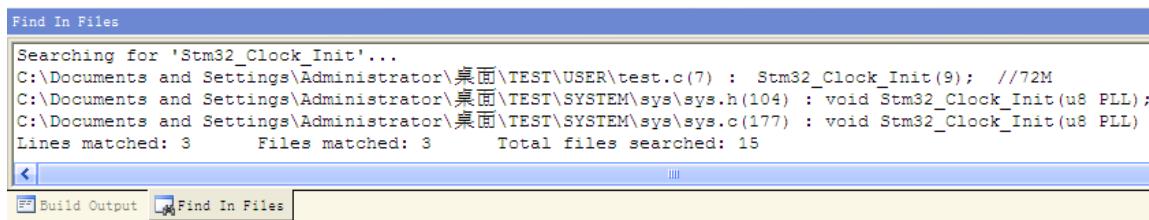


图 3.5.4.3 跨文件查找

点击 Find All, MDK 就会帮你找出所有含有 Stm32_Clock_Init 字段的文件并列出其所在位置, 如图 3.5.4.4 所示:



The screenshot shows the 'Find In Files' search results window. The search term 'Stm32_Clock_Init' is entered. The results list three matching files: test.c, sys.h, and sys.c. The total number of lines matched is 3, files matched is 3, and the total number of files searched is 15.

```
Find In Files
Searching for 'Stm32_Clock_Init'...
C:\Documents and Settings\Administrator\桌面\TEST\USER\test.c(7) : Stm32_Clock_Init(9); //72M
C:\Documents and Settings\Administrator\桌面\TEST\SYSTEM\sys\sys.h(104) : void Stm32_Clock_Init(u8 PLL);
C:\Documents and Settings\Administrator\桌面\TEST\SYSTEM\sys\sys.c(177) : void Stm32_Clock_Init(u8 PLL)
Lines matched: 3      Files matched: 3      Total files searched: 15
```

图 3.5.4.4 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。

第四章 STM32F4 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 STM32F4 系统架构
- 4.3 STM32F4 时钟系统
- 4.4 IO 引脚复用器和映射
- 4.5 STM32F4 NVIC 中断优先级管理
- 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 STM32F4 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 种位操作

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。下面我们想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->BSRRL 的状态，可以先对寄存器的值进行&清零操作

```
GPIOA->BSRRL &=0xFF0F; //将第 4-7 位清 0
```

然后再与需要设置的值进行|或运算

```
GPIOA->BSRRL |=0X0040;//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，我们来看看下面一行代码

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

这个操作就是将 ODR 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->ODR = 0x0030;
```

这样的代码就不好看也不好重用了。

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 PLL_M 8
```

定义标识符 PLL_M 的值为 8。

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stm32f4xx.h这个头文件中经常会看到这样的语句：

```
#if defined (STM32F40_41xxx)
    STM32F40x 系列和 STM32F41x 系列芯片需要的一些变量定义
#endif
```

而(STM32F40_41xxx 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 main.c 的 changeId(void) 函数中使用变量 id，这个时候我们就需要在 main.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 main.c 文件中。看下面 main.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 main.c 中申明变量 id 在外部定义，然后在 main.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 MODER;
    __IO uint32_t OTYPER;
```

...

};

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 MODER;
    __IO uint32_t OTYPER;
    ...
} 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，以及模式等。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 STM32F4 总线架构

STM32F4 的总线架构比 51 单片机就要强大很多了。STM32F4 总线架构的知识可以在《STM32F4XX 中文参考手册》第二章有讲解，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 STM32F4 之前对系统架构有一个初步的了解。这里的内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。如果需要详细深入的了解 STM32 的系统架构，还需要多看看《STM32F4XX 中文参考手册》或者在网上搜索其他相关资料学习。

我们这里所讲的 STM32F4 系统架构主要针对的 STM32F407 系列芯片。首先我们看看 STM32 的总线架构图：

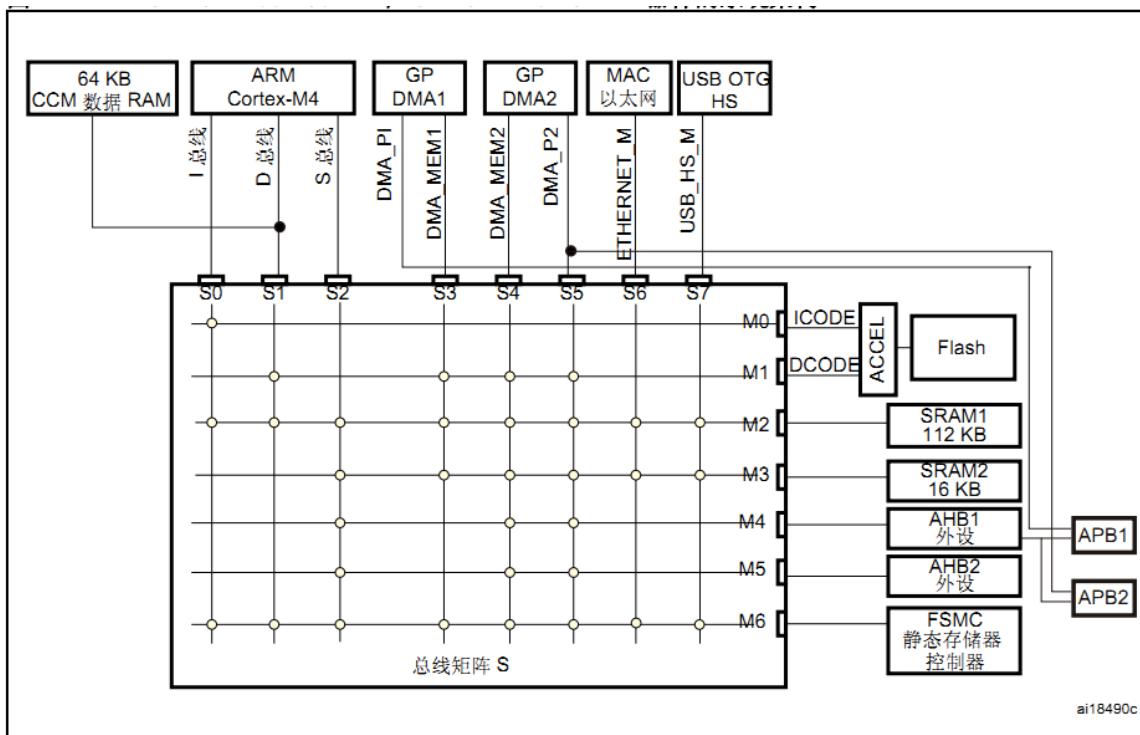


图 4.2.1 STM32F407 系统架构图

主系统由 32 位多层次 AHB 总线矩阵构成。总线矩阵用于主控总线之间的访问仲裁管理。仲裁采取循环调度算法。总线矩阵可实现以下部分互联：

八条主控总线是：

- Cortex-M4 内核 I 总线, D 总线和 S 总线;
- DMA1 存储器总线, DMA2 存储器总线;
- DMA2 外设总线;
- 以太网 DMA 总线;
- USB OTG HS DMA 总线;

七条被控总线：

- 内部 FLASH ICode 总线;
- 内部 FLASH DCode 总线;
- 主要内部 SRAM1(112KB)
- 辅助内部 SRAM2(16KB);
- 辅助内部 SRAM3(64KB) (仅适用 STM32F42xx 和 STM32F43xx 系列器件);

AHB1 外设 和 AHB2 外设;

FSMC

下面我们具体讲解一下图中几个总线的知识。

- ① **I 总线(S0):** 此总线用于将 Cortex-M4 内核的指令总线连接到总线矩阵。内核通过此总线获取指令。此总线访问的对象是包括代码的存储器。
- ② **D 总线(S1):** 此总线用于将 Cortex-M4 数据总线和 64KB CCM 数据 RAM 连接到总线矩阵。内核通过此总线进行立即数加载和调试访问。
- ③ **S 总线(S2):** 此总线用于将 Cortex-M4 内核的系统总线连接到总线矩阵。此总线用于访问位于外设或 SRAM 中的数据。
- ④ **DMA 存储器总线 (S3,S4):** 此总线用于将 DMA 存储器总线主接口连接到总线矩阵。DMA 通过此总线来执行存储器数据的传入和传出。
- ⑤ **DMA 外设总线:** 此总线用于将 DMA 外设主总线接口连接到总线矩阵。DMA 通过此总线访问 AHB 外设或执行存储器之间的数据传输。
- ⑥ **以太网 DMA 总线:** 此总线用于将以太网 DMA 主接口连接到总线矩阵。以太网 DMA 通过此总线向存储器存取数据。
- ⑦ **USB OTG HS DMA 总线(S7):** 此总线用于将 USB OTG HS DMA 主接口连接到总线矩阵。USB OTG HS DMA 通过此总线向存储器加载/存储数据。

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 STM32F4 时钟系统

STM32F4 时钟系统的知识在《STM32F4 中文参考手册》第六章复位和时钟控制章节有非常详细的讲解，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，纯粹根据官方提供的中文参考手册和自己的应用心得来总结的，如有不合理之处望大家谅解。

这部分内容我们分 3 个小节来讲解：

- 4.3.1 STM32F4 时钟树概述
- 4.3.2 STM32F4 时钟初始化配置
- 4.3.3 STM32F4 时钟使能和配置

4.3.1 STM32F4 时钟树概述

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。 STM32F4 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 STM32F4 的时钟系统图：

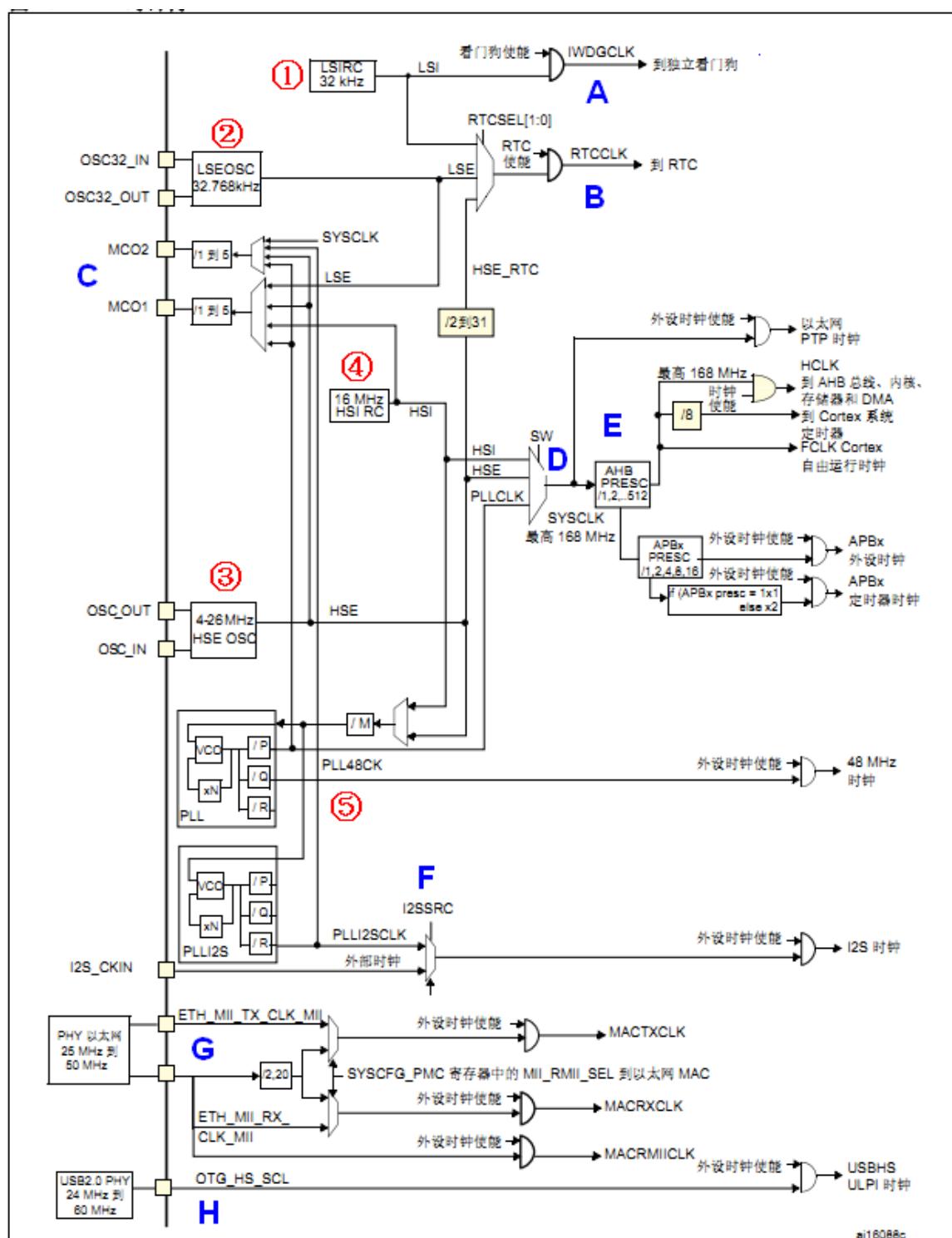


图 4.3.1.1 STM32 时钟系统图

在 STM32F4 中，有 5 个最重要的时钟源，为 HSI、HSE、LSI、LSE、PLL。其中 PLL 实际是分为两个时钟源，分别为主 PLL 和专用 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 HSI，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 STM32F4 的这 5 个时钟源，我们讲解顺序是按图中红图标示的顺序：

- ①、LSI 是低速内部时钟，RC 振荡器，频率为 32kHz 左右。供独立看门狗和自动唤醒单元使用。
 - ②、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。
 - ③、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~26MHz。我们的开发板接的是 8M 的晶振。HSE 也可以直接做为系统时钟或者 PLL 输入。
 - ④、HSI 是高速内部时钟，RC 振荡器，频率为 16MHz。可以直接作为系统时钟或者用作 PLL 输入。
 - ⑤、PLL 为锁相环倍频输出。STM32F4 有两个 PLL：
- 1) 主 PLL(PLL)由 HSE 或者 HSI 提供时钟信号，并具有两个不同的输出时钟。
第一个输出 PLLP 用于生成高速的系统时钟（最高 168MHz）
第二个输出 PLLQ 用于生成 USB OTG FS 的时钟（48MHz），随机数发生器的时钟和 SDIO 时钟。
 - 2) 专用 PLL(PLLI2S)用于生成精确时钟，从而在 I2S 接口实现高品质音频性能。

这里我们着重看看主 PLL 时钟第一个高速时钟输出 PLLP 的计算方法。图 4.3.1.2 是主 PLL 的时钟图。

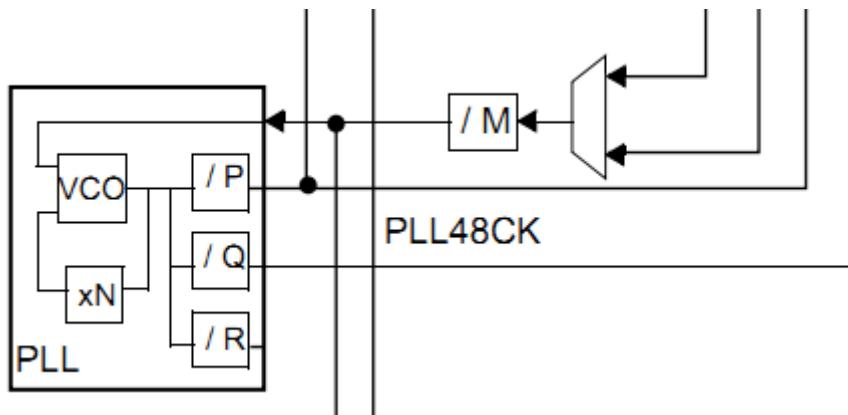


图 4.3.1.2 STM32F4 主 PLL 时钟图

从图 4.3.1.2 可以看出。主 PLL 时钟的时钟源要先经过一个分频系数为 M 的分频器，然后经过倍频系数为 N 的倍频器出来之后的时候还需要经过一个分频系数为 P (第一个输出 PLLP) 或者 Q (第二个输出 PLLQ) 的分频器分频之后，最后才生成最终的主 PLL 时钟。

例如我们的外部晶振选择 8MHz。同时我们设置相应的分频器 M=8，倍频器倍频系数 N=336，分频器分频系数 P=2，那么主 PLL 生成的第一个输出高速时钟 PLLP 为：

$$\text{PLL} = 8\text{MHz} * N / (M * P) = 8\text{MHz} * 336 / (8 * 2) = 168\text{MHz}$$

如果我们选择 HSE 为 PLL 时钟源，同时 SYSCLK 时钟源为 PLL，那么 SYSCLK 时钟为 168MHz。这对于我们后面的实验都是采用这样的配置。

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？这里我们选择一些比较常用的时钟知识来讲解。

图 4.3.1.1 中我们用 A~G 标示我们要讲解的地方。

- A. 这里是看门狗时钟输入。从图中可以看出，看门狗时钟源只能是低速的 LSI 时钟。
- 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 分频后的时钟，HSE 分频系数为 2~31。
- C. 这里是 STM32F4 输出时钟 MCO1 和 MCO2。MCO1 是向芯片的 PA8 引脚输出时钟。它有四个时钟来源分别为：HSI,LSE,HSE 和 PLL 时钟。MCO2 是向芯片的

PC9 输出时钟，它同样有四个时钟来源分别为：HSE,PLL, SYSCLK 以及 PLLI2S 时钟。MCO 输出时钟频率最大不超过 100MHz。

- D. 这里是系统时钟。从图 4.3.1 可以看出，SYSCLK 系统时钟来源有三个方面：HSI,HSE 和 PLL。在我们实际应用中，因为对时钟速度要求都比较高我们才会选用 STM32F4 这种级别的处理器，所以一般情况下，都是采用 PLL 作为 SYSCLK 时钟源。根据前面的计算公式，大家就可以算出你的系统的 SYSCLK 是多少。
- E. 这里我们指的是以太网 PTP 时钟，AHB 时钟，APB2 高速时钟，APB1 低速时钟。这些时钟都是来源于 SYSCLK 系统时钟。其中以太网 PTP 时钟是使用系统时钟。AHB,APB2 和 APB1 时钟是经过 SYSCLK 时钟分频得来。这里大家记住，AHB 最大时钟为 168MHz, APB2 高速时钟最大频率为 84MHz, 而 APB1 低速时钟最大频率为 42MHz。
- F. 这是指 I2S 时钟源。从图 4.3.1 可以看出，I2S 的时钟源来源于 PLLI2S 或者映射到 I2S_CKIN 引脚的外部时钟。I2S 出于音质的考虑，对时钟精度要求很高。探索者 STM32F4 开发板使用的是内部 PLLI2SCLK。
- G. 这是 STM32F4 内部以太网 MAC 时钟的来源。对于 MII 接口来说，必须向外部 PHY 芯片提供 25Mhz 的时钟，这个时钟，可以由 PHY 芯片外接晶振，或者使用 STM32F4 的 MCO 输出来提供。然后，PHY 芯片再给 STM32F4 提供 ETH_MII_TX_CLK 和 ETH_MII_RX_CLK 时钟。对于 RMII 接口来说，外部必须提供 50Mhz 的时钟驱动 PHY 和 STM32F4 的 ETH_RMII_REF_CLK，这个 50Mhz 时钟可以来自 PHY、有源晶振或者 STM32F4 的 MCO。我们的开发板使用的是 RMII 接口，使用 PHY 芯片提供 50Mhz 时钟驱动 STM32F4 的 ETH_RMII_REF_CLK。
- H. 这是指外部 PHY 提供的 USB OTG HS (60MHZ) 时钟。

这里还需要说明一下，Cortex 系统定时器 Systick 的时钟源可以是 AHB 时钟 HCLK 或 HCLK 的 8 分频。具体配置请参考 Systick 定时器配置，我们后面会在 5.1 小节讲解 delay 文件夹代码的时候讲解。

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。

4.3.2 STM32F4 时钟初始化配置

上一小节我们对 STM32F4 时钟树进行了初步的讲解，接下来我们来讲解一下 STM32F4 的系统时钟配置。

STM32F4 时钟系统初始化是在 system_stm32f4xx.c 中的 SystemInit()函数中完成的。对于系统时钟关键寄存器设置主要是在 SystemInit 函数中调用 SetSysClock()函数来设置的。我们可以先看看 SystemInit ()函数体：

```
void SystemInit(void)
{
    /* FPU settings -----*/
    #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
        SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2)); /* set CP10 and CP11 Full Access */
    #endif
}
```

```
/* Reset the RCC clock configuration to the default reset state -----*/
/* Set HSION bit */
RCC->CR |= (uint32_t)0x00000001;
/* Reset CFGR register */
RCC->CFGR = 0x00000000;
/* Reset HSEON, CSSON and PLLON bits */
RCC->CR &= (uint32_t)0xFEF6FFFF;
/* Reset PLLCFG register */
RCC->PLLCFGR = 0x24003010;
/* Reset HSEBYP bit */
RCC->CR &= (uint32_t)0xFFFFBFFF;
/* Disable all interrupts */
RCC->CIR = 0x00000000;

#if defined (DATA_IN_ExtSRAM) || defined (DATA_IN_ExtSDRAM)
    SystemInit_ExtMemCtl();
#endif /* DATA_IN_ExtSRAM || DATA_IN_ExtSDRAM */

/* Configure the System clock source, PLL Multiplier and Divider factors,
   AHB/APBx prescalers and Flash settings -----*/
SetSysClock();
/* Configure the Vector Table location add offset address -----*/
#ifndef 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
}
```

SystemInit 函数开始先进行浮点运算单元设置，然后是复位 PLLCFGR,CFGR 寄存器，同时通过设置 CR 寄存器的 HSI 时钟使能位来打开 HSI 时钟。默认情况下如果 CFGR 寄存器复位，那么是选择 HSI 作为系统时钟，这点大家可以查看 RCC->CFGR 寄存器的位描述最低 2 位可以得知，当低两位配置为 00 的时候（复位之后），会选择 HSI 振荡器为系统时钟。也就是说，调用 SystemInit 函数之后，首先是选择 HSI 作为系统时钟。下面是 RCC->CFGR 寄存器的位 1:0 配置描述（CFGR 寄存器详细描述请参考《STM32F4 中文参考手册》6.3.31CFGR 寄存器配置表）如下表 4.3.2.1：

位 1:0 **SW:** 系统时钟切换 (System clock switch)

由软件置 1 和清零，用于选择系统时钟源。

由硬件置 1，用于在退出停机或待机模式时或者在直接或间接用作系统时钟的 HSE 振荡器发生故障时强制 HSI 的选择。

00: 选择 HSI 振荡器作为系统时钟

01: 选择 HSE 振荡器作为系统时钟

10: 选择 PLL 作为系统时钟

11: 不允许

表 4.3.2.1 RCC->CFGR 寄存器的位 1:0 配置

在设置完相关寄存器后，接下来 SystemInit 函数内部会调用 SetSysClock 函数。这个函数比较长，我们就把函数一些关键代码行截取出来给大家讲解一下。这里我们省略一些宏定义标识符值的判断而直接把针对 STM32F407 比较重要的内容贴出来：

```
static void SetSysClock(void)
{
    __IO uint32_t StartUpCounter = 0, HSEStatus = 0;
    /*使能 HSE*/
    RCC->CR |= ((uint32_t)RCC_CR_HSEON);

    /* 等待 HSE 稳定*/
    do
    {
        HSEStatus = RCC->CR & RCC_CR_HSERDY;
        StartUpCounter++;
    } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));

    if ((RCC->CR & RCC_CR_HSERDY) != RESET)
    {
        HSEStatus = (uint32_t)0x01;
    }
    else
    {
        HSEStatus = (uint32_t)0x00;
    }

    if (HSEStatus == (uint32_t)0x01)
    {
        /* Select regulator voltage output Scale 1 mode */
        RCC->APB1ENR |= RCC_APB1ENR_PWREN;
        PWR->CR |= PWR_CR_VOS;

        /* HCLK = SYSCLK / 1*/
        RCC->CFGR |= RCC_CFGR_HPRE_DIV1;

        /* PCLK2 = HCLK / 2*/
    }
}
```

```
RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;

/* PCLK1 = HCLK / 4*/
RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;
/* PCLK2 = HCLK / 2*/
RCC->CFGR |= RCC_CFGR_PPRE2_DIV1;

/* PCLK1 = HCLK / 4*/
RCC->CFGR |= RCC_CFGR_PPRE1_DIV2;

/* Configure the main PLL */
RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
(RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);

/* 使能主 PLL*/
RCC->CR |= RCC_CR_PLLON;

/* 等待主 PLL 就绪 */
while((RCC->CR & RCC_CR_PLLRDY) == 0)
{
}

/* Configure Flash prefetch, Instruction cache, Data cache and wait state */
FLASH->ACR = FLASH_ACR_PRFTEN | FLASH_ACR_ICEN
|FLASH_ACR_DCEN |FLASH_ACR_LATENCY_5WS;

/* 设置主 PLL 时钟为系统时钟源 */
RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
RCC->CFGR |= RCC_CFGR_SW_PLL;

/* 等待设置稳定（主 PLL 作为系统时钟源）*/
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS ) != RCC_CFGR_SWS_PLL);
{
}
else
{
/* If HSE fails to start-up, the application will have wrong clock
configuration. User can add here some code to deal with this error */
}

}
```

这段代码的大致流程是这样的：先使能外部时钟 HSE，等待 HSE 稳定之后，配置 AHB, APB1, APB2 时钟相关的分频因子，也就是相关外设的时钟。等待这些都配置完成之后，打开主 PLL 时钟，然后设置主 PLL 作为系统时钟 SYSCLK 时钟源。如果 HSE 不能达到就绪状态（比如外部晶振不能稳定或者没有外部晶振），那么依然会是 HSI 作为系统时钟。

在这里要特别提出来，在设置主 PLL 时钟的时候，会要设置一系列的分频系数和倍频系数参数。大家可以从 SetSysClock 函数的这行代码看出：

```
RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) -1) << 16) |
(RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);
```

这些参数是通过宏定义标识符的值来设置的。默认的配置在 System_stm32f4xx.c 文件开头的地方配置。对于我们开发板，我们的设置参数值如下：

```
#define PLL_M      8
#define PLL_Q      7
#define PLL_N      336
#define PLL_P      2
```

所以我们的主 PLL 时钟为：

$$\text{PLL} = 8\text{MHz} * N / (M * P) = 8\text{MHz} * 336 / (8 * 2) = 168\text{MHz}$$

在开发过程中，我们可以通过调整这些值来设置我们的系统时钟。

这里还有个特别需要注意的地方，就是我们还要同步修改 stm32f4xx.h 中宏定义标识符 HSE_VALUE 的值为我们的外部时钟：

```
#if !defined (HSE_VALUE)
#define HSE_VALUE ((uint32_t)8000000) /*!< Value of the External oscillator in Hz */

#endif /* HSE_VALUE */
```

这里默认固件库配置的是 25000000，我们外部时钟为 8MHz，所以我们根据我们硬件情况修改为 8000000 即可。

讲到这里，大家对 SystemInit 函数的流程会有个比较清晰的理解。那么 SystemInit 函数是怎么被系统调用的呢？SystemInit 是整个设置系统时钟的入口函数。这个函数对于我们使用 ST 提供的 STM32F4 固件库的话，会在系统启动之后先执行 main 函数，然后再接着执行 SystemInit 函数实现系统相关时钟的设置。这个过程设置是在启动文件 startup_stm32f40_41xxx.s 中间设置的，我们接下来看看启动文件中这段启动代码：

```
; Reset handler
Reset_Handler    PROC
                EXPORT  Reset_Handler           [WEAK]
IMPORT  SystemInit
IMPORT  __main

                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
ENDP
```

这段代码的作用是在系统复位之后引导进入 main 函数，同时在进入 main 函数之前，首先要调用 SystemInit 系统初始化函数完成系统时钟等相关配置。

最后我们总结一下 SystemInit() 函数中设置的系统时钟大小：

SYSCLK (系统时钟)	=168MHz
AHB 总线时钟(HCLK=SYSCLK)	=168MHz
APB1 总线时钟(PCLK1=SCLK/4)	=42MHz
APB2 总线时钟(PCLK2=SCLK/2)	=84MHz
PLL 主时钟	=168MHz

4.3.3 STM32F4 时钟使能和配置

上小节我们讲解了系统复位之后调用 SystemInit 函数之后相关时钟的默认配置。如果在系统初始化之后，我们还需要修改某些时钟源配置，或者我们要使能相关外设的时钟该怎么设置呢？这些设置实际是在 RCC 相关寄存器中配置的。因为 RCC 相关寄存器非常多，有兴趣的同学可以直接打开《STM32F4 中文参考手册》6.3 小节查看所有 RCC 相关寄存器的配置。所以这里我们不直接讲解寄存器配置，而是通过 STM32F4 标准固件库配置方法给大家讲解。

在 STM32F4 标准固件库里，时钟源的选择以及时钟使能等函数都是在 RCC 相关固件库文件 stm32f4xx_rcc.h 和 stm32f4xx_rcc.c 中声明和定义的。大家打开 stm32f4xx_rcc.h 文件可以看到文件开头有很多宏定义标识符，然后是一系列时钟配置和时钟使能函数申明。这些函数大致可以归结为三类，一类是外设时钟使能函数，一类是时钟源和分频因子配置函数，还有一类是外设复位函数。当然还有几个获取时钟源配置的函数。下面我们以几种常见的操作来简要介绍一下这些库函数的使用。

首先是时钟使能函数。时钟使能相关函数包括外设设置使能和时钟源使能两类。首先我们来看看外设时钟使能相关的函数：

```
void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
void RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState);
void RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState);
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState);
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

这里主要有 5 个外设时钟使能函数。5 个函数分别用来使能 5 个总线下挂载的外设时钟，这些总线分别为：AHB1 总线，AHB2 总线，AHB3 总线，APB1 总线以及 APB2 总线。要使能某个外设，调用对应的总线外设时钟使能函数即可。

这里我们要特别说明一下，STM32F4 的外设在使用之前，必须对时钟进行使能，如果没有使能时钟，那么外设是无法正常工作的。对于哪个外设是挂载在哪个总线之下，虽然我们也可以查手册查询到，但是这里如果大家使用的是库函数的话，实际上是没有必要去查询手册的，这里我们给大家介绍一个小技巧。

比如我们要使能 GPIOA，我们只需要在 stm32f4xx_rcc.h 头文件里面搜索 GPIOA，就可以搜索到对应的时钟使能函数的第一个入口参数为 RCC_AHB1Periph_GPIOA，从这个宏定义标识符一眼就可以看出，GPIOA 是挂载在 AHB1 下面。同理，对于串口 1 我们可以搜索 USART1，找到标识符为 RCC_APB2Periph_USART1，那么很容易知道串口 1 是挂载在 APB2 之下。这个知识在我们后面的“4.7 快速组织代码技巧”小节也有讲解，这里顺带提一下。

如果我们要使能 GPIOA，那么我们可以在头文件 stm32f4xx_rcc.h 里面查看到宏定义标识符 RCC_AHB1Periph_GPIOA，顾名思义 GPIOA 是挂载在 AHB1 总线之下，所以，我们调用 AHB1 总线下外设时钟使能函数 RCC_AHB1PeriphClockCmd 即可。具体调用方式入如下：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 GPIOA 时钟
```

同理，如果我们要使能串口 1 的时钟，那么我们调用的函数为：

```
void RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
```

具体的调用方法是：

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

还有一类时钟使能函数是时钟源使能函数，前面我们已经讲解过 STM32F4 有 5 大类时钟源。这里我们列出来几种重要的时钟源使能函数：

```
void RCC_HSICmd(FunctionalState NewState);
void RCC_LSICmd(FunctionalState NewState);
void RCC_PLLCmd(FunctionalState NewState);
void RCC_PLLI2SCmd(FunctionalState NewState);
void RCC_PLLSAICmd(FunctionalState NewState);
void RCC_RTCCLKCmd(FunctionalState NewState);
```

这些函数是用来使能相应的时钟源。比如我们要使能 PLL 时钟，那么调用的函数为：

```
void RCC_PLLCmd(FunctionalState NewState);
```

具体调用方法如下：

```
RCC_PLLCmd(ENABLE);
```

我们要使能相应的时钟源，调用对应的函数即可。

接下来我们要讲解的是第二类时钟功能函数：时钟源选择和分频因子配置函数。这些函数是用来选择相应的时钟源以及配置相应的时钟分频系数。比如我们之前讲解过系统时钟 SYSCLK，我们可以选择 HSI,HSE 以及 PLL 三个中的一个时钟源为系统时钟。那么到底选择哪一个，这是可以配置的。下面我们列举几种时钟源配置函数：

```
void RCC_LSEConfig(uint8_t RCC_LSE);
void RCC_SYSCLKConfig(uint32_t RCC_SYSCLKSource);
void RCC_HCLKConfig(uint32_t RCC_SYSCLK);
void RCC_PCLK1Config(uint32_t RCC_HCLK);
void RCC_PCLK2Config(uint32_t RCC_HCLK);
void RCC_RTCCLKConfig(uint32_t RCC_RTCCLKSource);
void RCC_PLLConfig(uint32_t RCC_PLLSource, uint32_t PLLM,
                    uint32_t PLLN, uint32_t PLLP, uint32_t PLLQ);
```

比如我们要设置系统时钟源为 HSI，那么我们可以调用系统时钟源配置函数：

```
void RCC_HCLKConfig(uint32_t RCC_SYSCLK);
```

具体配置方法如下：

```
RCC_HCLKConfig(RCC_SYSCLKSource_HSI);//配置时钟源为 HSI
```

又如我们要设置 APB1 总线时钟为 HCLK 的 2 分频，也就是设置分频因子为 2 分频，那么如果我们要使能 HSI，那么调用的函数为：

```
void RCC_PCLK1Config(uint32_t RCC_HCLK);
```

具体配置方法如下：

```
RCC_PCLK1Config(RCC_HCLK_Div2);
```

接下来我们看看第三类外设复位函数。如下：

```
void RCC_AHB1PeriphResetCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
void RCC_AHB2PeriphResetCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState);
void RCC_AHB3PeriphResetCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState);
void RCC_APB1PeriphResetCmd(uint32_t RCC_APB1Periph, FunctionalState NewState);
```

```
void RCC_APB2PeriphResetCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

这类函数跟前面讲解的外设时钟函数使用方法基本一致，不同的是一个是用来使能外设时钟，一个是用来复位对应的外设。这里大家在调用函数的时候一定不要混淆。

对于这些时钟操作函数，我们就不一一列举出来，大家可以打开 RCC 对应的文件仔细了解。

4.4 IO 引脚复用器和映射

STM32F4 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《STM32F4 中文参考手册》第七章和芯片数据手册有详细的讲解哪些 GPIO 管脚是可以复用为哪些内置外设。

对于本小节知识，STM32F4 中文参考手册讲解比较详细，我们同样会从中抽取重要的知识点罗列出来。同时，我们会以串口使用为例给大家讲解具体的引脚复用的配置。

STM32F4 系列微控制器 IO 引脚通过一个复用器连接到内置外设或模块。该复用器一次只允许一个外设的复用功能（AF）连接到对应的 IO 口。这样可以确保共用同一个 IO 引脚的外设之间不会发生冲突。

每个 IO 引脚都有一个复用器，该复用器采用 16 路复用功能输入（AF0 到 AF15），可通过 GPIOx_AFRL（针对引脚 0~7）和 GPIOx_AFRH（针对引脚 8~15）寄存器对这些输入进行配置，每四位控制一路复用：

- 1) 完成复位后，所有 IO 都会连接到系统的复用功能 0 (AF0)。
- 2) 外设的复用功能映射到 AF1 到 AF13。
- 3) Cortex-M4 EVENTOUT 映射到 AF15。

复用器示意图如下图 4.4.1：

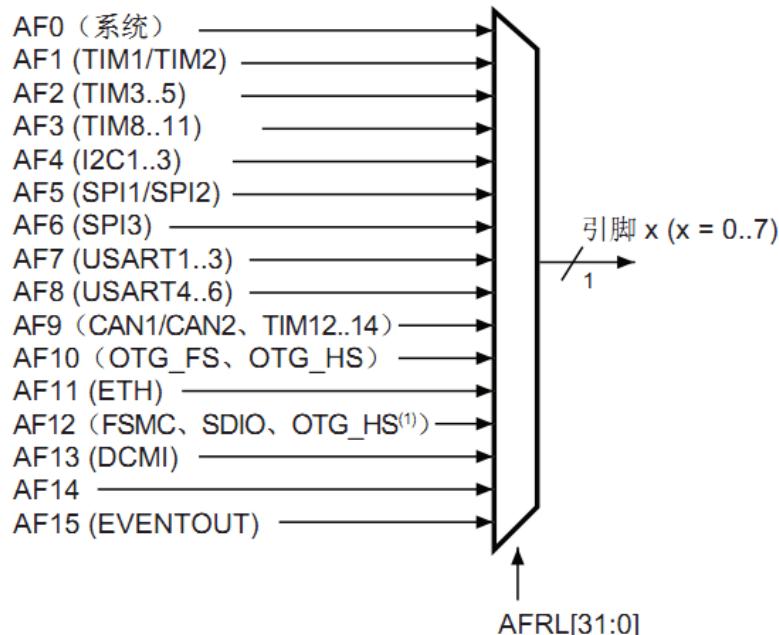


图 4.4.1 复用器示意图

接下来，我们简单说明一下这个图要如何看，举个例子，探索者 STM32F407 开发板的原理图上 PC11 的原理图如图 4.4.2 所示：

SDIO_D3 DCMI_D4 PC11_112 | PC11/SPI3_MISO/U3_RX/U4_RX/SDIO_D3/DCMI_D4/I2S3ext_SD

图 4.4.2 探索者 STM32F407 开发板 PC11 原理图

如上图所示，PC11 可以作为 SPI3_MISO/U3_RX/U4_RX/SDIO_D3/DCMI_D4/I2S3ext_SD 等复用功能输出，这么多复用功能，如果这些外设都开启了，那么对 STM32F1 来说，那就可能乱套了，外设之间可互相干扰，但是 STM32F4，由于有复用功能选择功能，可以让 PC11 仅连接到某个特定的外设，因此不存在互相干扰的情况。

上图 4.4.1 是针对引脚 0-7，对于引脚 8-15，控制寄存器为 GPIOx_AFRH。从图中可以看出。当需要使用复用功能的时候，我们配置相应的寄存器 GPIOx_AFRL 或者 GPIOx_AFRH，让对应引脚通过复用器连接到对应的复用功能外设。这里我们列出 GPIOx_AFRL 寄存器的描述，GPIOx_AFRH 的作用跟 GPIOx_AFRL 类似，只不过 GPIOx_AFRH 控制的是一组 I/O 口的高八位，GPIOx_AFRL 控制的是一组 I/O 口的低八位。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw												

位 31:0 **AFRLy**: 端口 x 位 y 的复用功能选择 (Alternate function selection for port x bit y) (y = 0..7)
这些位通过软件写入，用于配置复用功能 I/O。

AFRLy 选择:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

图 4.4.3 GPIOx_AFRL 寄存器位描述

从表中可以看出，32 位寄存器 GPIOx_AFRL 每四个位控制一个 I/O 口，所以每个寄存器控制 $32/4=8$ 个 I/O 口。寄存器对应四位的值配置决定这个 I/O 映射到哪个复用功能 AF。

在微控制器完成复位后，所有 I/O 口都会连接到系统复用功能 0 (AF0)。这里大家需要注意，对于系统复用功能 AF0，我们将 I/O 口连接到 AF0 之后，还要根据所用功能进行配置：

- 1) JTAG/SWD: 在器件复位之后，会将这些功能引脚指定为专用引脚。也就是说，这些引脚在复位后默认就是 JTAG/SWD 功能。如果我们要作为 GPIO 来使用，就需要对对应的 I/O 口复用器进行配置。
- 2) RTC_REFIN: 此引脚在系统复位之后要使用的话要配置为浮空输入模式。
- 3) MC01 和 MC02: 这些引脚在系统复位之后要使用的话要配置为复用功能模式。

对于外设复用功能的配置，**除了 ADC 和 DAC 要将 I/O 配置为模拟通道之外其他外设功能一律要配置为复用功能模式**，这个配置是在 I/O 口对应的 GPIOx_MODER 寄存器中配置的。同时要配置 GPIOx_AFRH 或者 GPIOx_AFRL 寄存器，将 I/O 口通过复用器连接到所需要的复用功能对应的 AFx。

不是每个 I/O 口都可以复用为任意复用功能外设。到底哪些 I/O 可以复用为相关外设呢？这在芯片对应的数据手册(请参考光盘目录：)上面会有详细的表格列出来。对于 STM32F407，数据手册里面的 Table 9. Alternate function mapping 表格列出了所有的端口 AF 映射表，因为

表格比较大，所以这里只列出 PORTA 的几个端口为例方便大家理解：

	PA0	PA5	PA8	PA9	PA10
AF0			MC01		
AF1	TIM2_CH1_ETR	TIM2_CH1_ETR	TIM1_CH1	TIM1_CH2	TIM1_CH3
AF2	TIM5_CH1				
AF3	TIM8_ETR	TIM8_CH1N			
AF4			I2C3_SCL	I2C3_SMBA	
AF5					
AF6					
AF7	USART2_CTS	SPI1_SCK	USART1_CK	USART1_TX	USART1_RX
AF8	UART4_TX				
AF9					
AF10		OTG_HS_ULPI_CK	OTG_FS_SOF		OTG_FS_ID
AF11	ETH_MII CRS				
AF12					
AF13				DCMI_D0	DCMI_D1
AF14					
AF15	EVENTOUT	EVENTOUT	EVENTOUT	EVENTOUT	EVENTOUT

表 4.4.4 PORTA 部分端口 AF 映射表

从表 4.4.4 可以看出，PA9 连接 AF7 可以复用为串口 1 的发送引脚 USART1_TX，PA10 连接 AF7 可以复用为串口 2 的接受引脚 USART1_RX。

接下来我们以串口 1 为例来讲解怎么配置 GPIOA.9, GPIOA.10 口为串口 1 复用功能。

1) 首先，我们要使用 I0 复用功能外设，必须先打开对应的 I0 时钟和复用功能外设时钟。

```
/*使能 GPIOA 时钟*/
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE);
/*使能 USART1 时钟*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE);
```

这里需要说明一下，官方库提供了五个打开 GPIO 和外设时钟的函数分别为：

```
void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
void RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState);
void RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState);
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState);
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

这五个函数分别用来打开相应的总线下 GPIO 和外设时钟。比如我们的串口 1 是挂载在 APB2 总线之下，所以我们调用对应的 APB2 总线下外设时钟使能函数 RCC_APB2PeriphClockCmd 来使能串口 1 时钟。对于其他外设我们调用相应的函数即可。具体库函数要怎么快速找到对应的外设使能函数，大家可以参考我们接下来的 4.7 小节快速组织代码技巧，我们有详细的举例说明。

2) 其次，我们在 GPIOx_MODER 寄存器中将所需 I0 (对于串口 1 是 PA9, PA10) 配置为复用功能 (ADC 和 DAC 设置为模拟通道)。

3) 再次，我们还需要对 I0 口的其他参数，例如类型，上拉/下拉以及输出速度。

上面两步，在我们库函数中是通过 GPIO_Init 函数来实现的，参考代码如下：

```
/*GPIOA9 与 GPIOA10 初始化*/
```

```

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 PA9, PA10

```

4) 最后，我们配置 GPIOx_AFRL 或者 GPIOx_AFRH 寄存器，将 IO 连接到所需的 AFx。

这些步骤对于我们使用库函数来操作的话，是调用的 GPIO_PinAFConfig 函数来实现的。具体操作代码如下：

```

/*PA9 连接 AF7, 复用为 USART1_TX */
GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
/* PA10 连接 AF7, 复用为 USART1_RX */
GPIO_PinAFConfig(GPIOA, GPIO_PinSource10, GPIO_AF_USART1);

```

对于函数 GPIO_PinAFConfig 函数，入口第一个第二个参数很好理解，可以确定是哪个 IO，对于第三个参数，实际上我们确定了这个 IO 到底是复用为哪种功能之后，这个参数也很好选择，因为可选的参数在 stm32f4xx_gpio.h 列出来非常详细，如下

```

#define IS_GPIO_AF(AF) (((AF) == GPIO_AF_RTC_50Hz) || ((AF) == GPIO_AF_TIM14) || \
                     ((AF) == GPIO_AF_MCO) || ((AF) == GPIO_AF_TAMPER) || \
                     ((AF) == GPIO_AF_SWJ) || ((AF) == GPIO_AF_TRACE) || \
                     ((AF) == GPIO_AF_TIM1) || ((AF) == GPIO_AF_TIM2) || \
                     ((AF) == GPIO_AF_TIM3) || ((AF) == GPIO_AF_TIM4) || \
                     ((AF) == GPIO_AF_TIM5) || ((AF) == GPIO_AF_TIM8) || \
                     ((AF) == GPIO_AF_I2C1) || ((AF) == GPIO_AF_I2C2) || \
                     ((AF) == GPIO_AF_I2C3) || ((AF) == GPIO_AF_SPI1) || \
                     ((AF) == GPIO_AF_SPI2) || ((AF) == GPIO_AF_TIM13) || \
                     ((AF) == GPIO_AF_SPI3) || ((AF) == GPIO_AF_TIM14) || \
                     ((AF) == GPIO_AF_USART1) || ((AF) == GPIO_AF_USART2) || \
                     ((AF) == GPIO_AF_USART3) || ((AF) == GPIO_AF_UART4) || \
                     ((AF) == GPIO_AF_UART5) || ((AF) == GPIO_AF_USART6) || \
                     ((AF) == GPIO_AF_CAN1) || ((AF) == GPIO_AF_CAN2) || \
                     ((AF) == GPIO_AF_OTG_FS) || ((AF) == GPIO_AF_OTG_HS) || \
                     ((AF) == GPIO_AF_ETH) || ((AF) == GPIO_AF_OTG_HS_FS) || \
                     ((AF) == GPIO_AF_SDIO) || ((AF) == GPIO_AF_DCMI) || \
                     ((AF) == GPIO_AF_EVENTOUT) || ((AF) == GPIO_AF_FSMC))

```

参考这些宏定义标识符，能很快找到函数的入口参数。

ST32F4 的端口复用和映射就给大家讲解到这里，希望大家课余结合相关实验工程和手册巩固本小节知识。

4.5 STM32 NVIC 中断优先级管理

CM4 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 STM32F4 并没有使用 CM4 内核的全部东西，而是只用了它的一部分。

STM32F40xx/STM32F41xx 总共有 92 个中断，STM32F42xx/STM32F43xx 则总共有 96 个中断，以下仅以 STM32F40xx/41xx 为例讲解。

STM32F40xx/STM32F41xx 的 92 个中断里面，包括 10 个内核中断和 82 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 82 个可屏蔽中断。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
{
    __IO uint32_t ISER[8];           /*!< Interrupt Set Enable Register */
    uint32_t RESERVED0[24];
    __IO uint32_t ICER[8];           /*!< Interrupt Clear Enable Register */
    uint32_t RESERVED1[24];
    __IO uint32_t ISPR[8];           /*!< Interrupt Set Pending Register */
    uint32_t RESERVED2[24];
    __IO uint32_t ICPR[8];           /*!< Interrupt Clear Pending Register */
    uint32_t RESERVED3[24];
    __IO uint32_t IABR[8];           /*!< Interrupt Active bit Register */
    uint32_t RESERVED4[56];
    __IO uint8_t  IP[240];           /*!< Interrupt Priority Register, 8Bit wide */
    uint32_t RESERVED5[644];
    __O  uint32_t STIR;             /*!< Software Trigger Interrupt Register */
} NVIC_Type;
```

STM32F4 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，才能方便的使用 STM32F4 的中断。下面重点介绍这几个寄存器：

ISER[8]: ISER 全称是：Interrupt Set-Enable Registers，这是一个中断使能寄存器组。上面说了 CM4 内核支持 256 个中断，这里用 8 个 32 位寄存器来控制，每个位控制一个中断。但是 STM32F4 的可屏蔽中断最多只有 82 个，所以对我们来说，有用的就是三个 (ISER[0~2])，总共可以表示 96 个中断。而 STM32F4 只用了其中的前 82 个。ISER[0]的 bit0~31 分别对应中断 0~31；ISER[1]的 bit0~32 对应中断 32~63；ISER[2]的 bit0~17 对应中断 64~81；这样总共 82 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 stm32f4xx.h 里面的第 188 行处。

ICER[8]: 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。

ISPR[8]: 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[8]: 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[8]: 全称是：Interrupt Active Bit Registers，是一个中断激活标志位寄存器组。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。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。

IP[240]: 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！STM32F4 的中断分组与这个寄存器组密切相关。IP 寄存器组由 240 个 8bit 的寄存器组成，每个可屏蔽中断占用 8bit，这样总共可以表示 240 个可屏蔽中断。而 STM32F4 只用到了其中的 82 个。IP[81]~IP[0]分别对应中断 81~0。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响应优先级。抢占优先级在前，响应优先级在后。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 STM32F4 的中断分组：STM32F4 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 4.5.1AIRCR 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 82 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_WKUP 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 STM32F4 中断设置的大致过程。接下来我们介绍如何使用函数实现以上中断设置，使得我们以后的中断设置简单化。

通过以上介绍，我们熟悉了 STM32F4 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 stm32f4xx.h 中定义的枚举类型 IRQn 的成员变量中可以找到每个中断对应的名字。例如串口 1 对应 USART1_IRQn。

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

NVIC_IRQChannelSubPriority: 定义这个中断的响应优先级别。

NVIC_IRQChannelCmd: 该中断通道是否使能。

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

```
NVIC_InitTypeDef NVIC_InitStruct;
NVIC_InitStruct.NVIC_IRQChannel = USART1_IRQn; //串口 1 中断
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 1; // 抢占优先级为 1
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 2; // 响应优先级位 2
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道使能
NVIC_Init(&NVIC_InitStruct); //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响应优先级的分配位数。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，下面我们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 stm32f4xx.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《STM32F4 中文参考手册》中的寄存器地址映射表(P193)。这里我们选用 GPIOA 为例来讲解。GPIOA 寄存器地址映射如下表 4.6.1：

偏移	寄存器
0x00	GPIOA_MODER
0x04	GPIOA_OTYPER
0x08	GPIOA_OSPEEDER
0x0C	GPIOA_PUPDR
0x10	GPIOA_IDR
0x14	GPIOA_ODR
0x18	GPIOA_BSRR
0x1c	GPIOA_LCKR
0x20	GPIOA_AFRL
0x24	GPIOA_AFRH

表 4.6.1 GIPOA 寄存器地址偏移表

从这个表我们可以看出，因为 GIPO 寄存器都是 32 位，所以每组 GPIO 的 10 个寄存器中，每个寄存器占有 4 个地址，一共占用 40 个地址，地址偏移范围为 (0x00~0x24)。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 AHB1 总线之上，所以它的基址是由 AHB1 总线的基址+GPIOA 在 AHB1 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。下面我们打开 stm32f4xx.h 定位到 GPIO_TypeDef 定义处：

```
typedef struct
{
    __IO uint32_t MODER;
    __IO uint32_t OTYPER;
    __IO uint32_t OSPEEDER;
```

```

__IO uint32_t PUPDR;
__IO uint32_t IDR;
__IO uint32_t ODR;
__IO uint16_t BSRRL;
__IO uint16_t BSRRH;
__IO uint32_t LCKR;
__IO uint32_t AFR[2];
} 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 (AHB1PERIPH_BASE + 0x0000)
```

依次类推，可以找到最顶层：

```
#define AHB1PERIPH_BASE (PERIPH_BASE + 0x00020000)
```

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

所以我们便可以算出 GPIOA 的基址地址：

```
GPIOA_BASE= 0x40000000+0x00020000+0x0000=0x40020000
```

下面我们再跟《STM32F 中文参考手册》比较一下看看 GPIOA 的基址地址是不是 0x40020000。截图 P53 存储器映射表我们可以看到，GPIOA 的起始地址也就是基址地址确实是 0x40020000：

0x4002 2000 - 0x4002 23FF	GPIOI
0x4002 1C00 - 0x4002 1FFF	GPIOH
0x4002 1800 - 0x4002 1BFF	GPIOG
0x4002 1400 - 0x4002 17FF	GPIOF
0x4002 1000 - 0x4002 13FF	GPIOE
0x4002 0C00 - 0x4002 0FFF	GPIOD
0x4002 0800 - 0x4002 0BFF	GPIOC
0x4002 0400 - 0x4002 07FF	GPIOB
0x4002 0000 - 0x4002 03FF	GPIOA

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地址。

上面我们已经知道 GPIOA 的基址地址，那么那些 GPIOA 的 10 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 基址地址的偏移地址，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 基址地址+寄存器相对 GPIOA 基址地址的偏移值

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？这里涉及到结构体成员变量地址对齐方式方面的知识，这方面的知识大家可以在网上查看相关资料复习一下，这里我们不做详细讲解。在我们定义好地址对齐方式之后，每个成员变量对应的地址就可以根据其基址地址来计算。对于结构体类型 GPIO_TypeDef，他的所有成员变量都是 32 位，成

员变量地址具有连续性。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_MODER	0x00	0x40020000+0x00
GPIOA_OTYPER	0x04	0x40020000+0x04
GPIOA_OSPEEDER	0x08	0x40020000+0x08
GPIOA_PUPDR	0x0C	0x40020000+0x0c
GPIOA_IDR	0x10	0x40020000+0x10
GPIOA_ODR	0x14	0x40010800+0x14
GPIOA_BSRR	0x18	0x40020000+0x18
GPIOA_LCKR	0x1c	0x40020000+0x1c
GPIOA_AFRL	0x20	0x40020000+0x20
GPIOA_AFRH	0x24	0x40020000+0x24

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->BSRR=value;就是设置地址为 0x40020000 +0x18 (BSRR 偏移量)=0x40020018 的寄存器 BSRR 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 stm32f4xx_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：

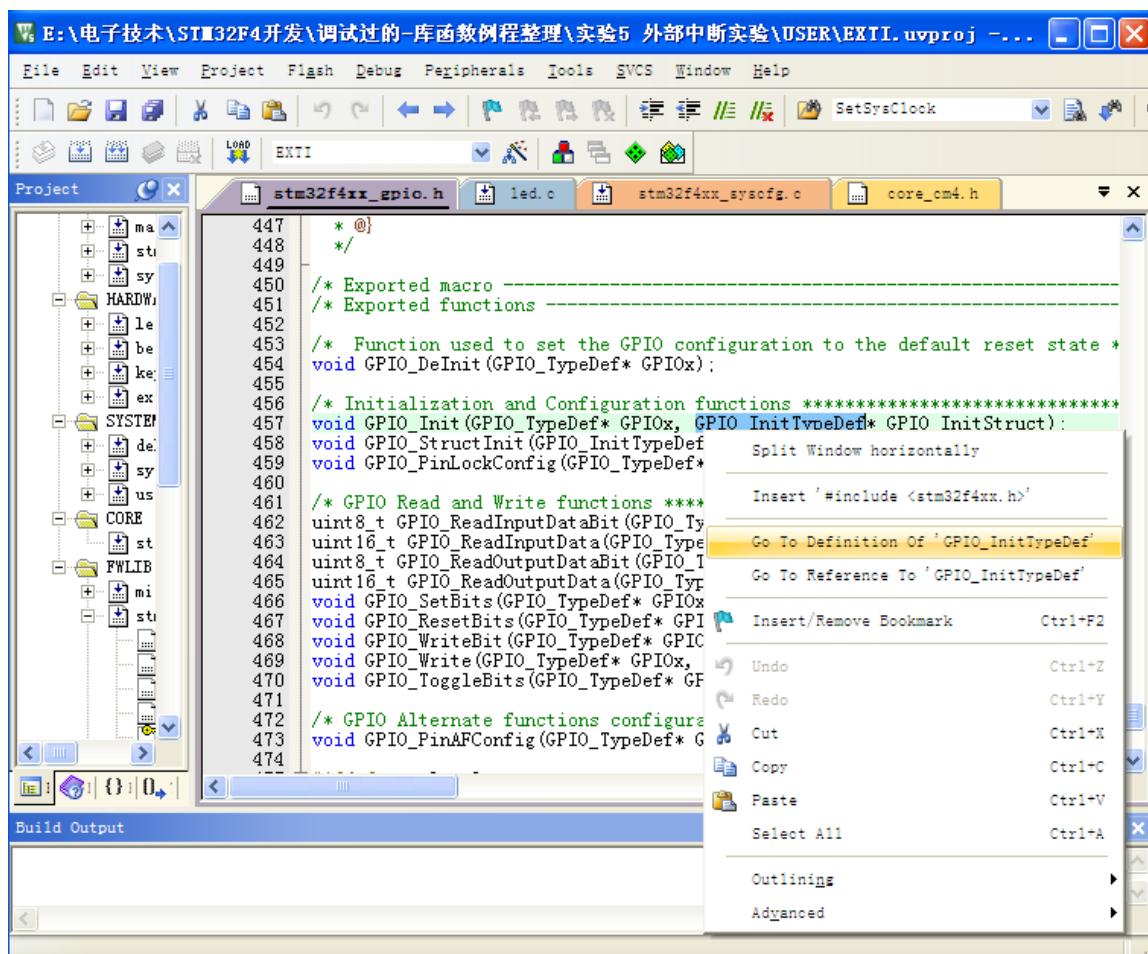


图 4.7.1 查看类型定义方法

于是定位到 `stm32f4xx_gpio.h` 中 `GPIO_InitTypeDef` 的定义处:

```
typedef struct
{
    uint32_t GPIO_Pin;
    GPIOMode_TypeDef GPIO_Mode;
    GPIOSpeed_TypeDef GPIO_Speed;
    GPIOOTYPE_TypeDef GPIO_OType;
    GPIOPuPd_TypeDef GPIO_PuPd;
}GPIO_InitTypeDef;
```

可以看到这个结构体有 5 个成员变量，这也告诉我们一个信息，一个 GPIO 口的状态是由模式 (`GPIO_Mode`)，速度(`GPIO_Speed`)，输出类型 (`GPIO_OType`) 以及上下来属性 (`GPIO_PuPd`) 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 `stm32f4xx_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_PIN(GPIO_InitStruct->GPIO_Pin));
    assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
    assert_param(IS_GPIO_PUPD(GPIO_InitStruct->GPIO_PuPd));
    .....
    assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
    .....
    assert_param(IS_GPIO_OTYPE(GPIO_InitStruct->GPIO_OType));
    .....
}

```

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) || \
                                    ((PERIPH) == GPIOH) || \
                                    ((PERIPH) == GPIOI) || \
                                    ((PERIPH) == GPIOJ) || \
                                    ((PERIPH) == GPIOK))
```

很明显可以看出，GPIOx 的取值规定只允许是 GPIOA~GPIOK。

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_IN    = 0x00, /*!< GPIO Input Mode */
    GPIO_Mode_OUT   = 0x01, /*!< GPIO Output Mode */
    GPIO_Mode_AF    = 0x02, /*!< GPIO Alternate function Mode */
    GPIO_Mode_AN    = 0x03 /*!< GPIO Analog Mode */

}GPIOMode_TypeDef;

#define IS_GPIO_MODE(MODE) (((MODE) == GPIO_Mode_IN)  ||
                           ((MODE) == GPIO_Mode_OUT) || \
                           ((MODE) == GPIO_Mode_AF)|| \
                           ((MODE) == GPIO_Mode_AN))
```

所以 GPIO_InitStruct->GPIO_Mode 成员的取值范围只能是上面定义的 4 种。这 4 种模式是通过

一个枚举类型组织在一起的。

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_9 ;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化
```

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_9 | GPIO_Pin_10| GPIO_Pin_11;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOF, &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 stm32f4xx.h 文件里面我们可以看到如下的宏定义:

```

#define RCC_AHB1Periph_GPIOA ((uint32_t)0x00000001)
#define RCC_AHB1Periph_GPIOB ((uint32_t)0x00000002)
#define RCC_AHB1Periph_GPIOC ((uint32_t)0x00000004)

#define RCC_AHB2Periph_DCMI ((uint32_t)0x00000001)
#define RCC_AHB2Periph_CRYP ((uint32_t)0x00000010)
#define RCC_AHB2Periph_HASH ((uint32_t)0x00000020)
#define RCC_AHB2Periph RNG ((uint32_t)0x00000040)

#define RCC_APB1Periph_TIM2 ((uint32_t)0x00000001)
#define RCC_APB1Periph_TIM3 ((uint32_t)0x00000002)
#define RCC_APB1Periph_TIM4 ((uint32_t)0x00000004)
#define RCC_APB2Periph_TIM1 ((uint32_t)0x00000001)
#define RCC_APB2Periph_TIM8 ((uint32_t)0x00000002)
#define RCC_APB2Periph_USART1 ((uint32_t)0x00000010)

#define RCC_AHB3Periph_FSMC ((uint32_t)0x00000001)

```

从上图定义的标识符名称可以很明显的看出 GPIOA~GPIOC 是挂载在 AHB1 下面, TIM2~TIM4 是挂载在 APB1 下面, TIM1 和 TIM8 是挂载在 APB2 下面。所以在使能 GPIO 的时候记住要调用的是 `RCC_AHB1PeriphClockCmd()` 函数使能, 在使能 TIM2 的时候调用的是 `RCC_APB1PeriphResetCmd()` 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
{
    uint32_t GPIO_Pin; /*!< Specifies the GPIO pins to be configured.
                        This parameter can be any value of @ref GPIO_pins_define */
    GPIOMode_TypeDef GPIO_Mode; /*!< Specifies the operating mode for the selected pins.
                                This parameter can be a value of @ref GPIOMode_TypeDef */
    GPIOSpeed_TypeDef GPIO_Speed; /*!< Specifies the speed for the selected pins.
                                */
}

```

```
This parameter can be a value of @ref GPIOSpeed_TypeDef */  
GPIOOType_TypeDef GPIO_OType; /*!< Specifies the operating output type for the  
selected pins. This parameter can be a value of @ref GPIOOType_TypeDef */  
GPIOPuPd_TypeDef GPIO_PuPd; /*!< Specifies the operating Pull-up/Pull down for the  
selected pins. This parameter can be a value of @ref GPIOPuPd_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。对于去查看手册的那个地方，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哪个位，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

上一章，我们介绍了如何在上一章，我们介绍了如何在 MDK5.11a 下建立 STM32F4 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，是 STM32F4xx 系列的底层核心驱动函数，可以用在 STM32F4xx 系列的各个型号上面，方便大家快速构建自己的工程。

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。通过这 3 个 c 文件，可以快速的给任何一款 STM32F4 构建最基本的框架。使用起来是很方便的。

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，来快速构建工程，并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 4 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_xms(u16 nms);
void delay_us(u32 nus);
```

下面分别介绍这四个函数，在介绍之前，我们先了解一下编程思想：CM4 内核的处理和 CM3 一样，内部都包含了一个 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《STM32xx 中文参考手册》里面基本没有介绍，其详细介绍，请参阅《STM32F3 与 F4 系列 Cortex M4 内核编程手册》第 230 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。

ALIENTEK 提供的这个最新版本延时函数，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*21（假设系统时钟为 168Mhz，因为 systick 的频率为系统时钟频率的 1/8，那么 systick 每增加 1，就是 1/21us），然后我们就一直统计 systick 的计数变化，直到这个值变化了 50*21，一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.
#ifdef 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.
#ifdef 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《STM32 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 core_m4.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=HCLK/8 1=HCLK
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 HCLK/8, 也就是 CPU 时钟频率的 1/8。假设我们外部晶振为 8M, 然后倍频到 168M, 那么 SysTick 的时钟即为 21Mhz, 也就是 SysTick 的计数器 VAL 每减 1, 就代表时间过了 1/21us。

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 位整型数据, fac_ms 为 16 位整型数据。Systick 的时钟来自系统时钟 8 分频, 正因为如此, 系统时钟如果不是 8 的倍数(不能被 8 整除), 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。
//注意:nus 的值,不要大于 798915us
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:要延时的 us 数.
void delay_us(u32 nus)
{
    u32 ticks, 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;//这里注意一下 SYSTICK 是一个递减
            //的计数器就可以了.
            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 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 $2^{32}/\text{fac_us}$ ，在 168M 主频下，最大延时，大概是 204 秒。

5.1.3 delay_xms 函数

该函数仅在没用到 ucosii 的时候使用，用来延时指定的 ms，其参数 nms 为要延时的毫秒数。该函数代码如下：

```

//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 168M 条件下,nms<=798ms
void delay_xms(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： $nms \leq 0xfffff * 8 * 1000 / \text{SYSCLK}$ 计算。 SYSCLK 单位为 Hz， nms 的单位为 ms。如果时钟为 168M，那么 nms 的最大值为 798ms。超过这个值，建议通过多次调用 delay_xms 实现，否则就会导致延时不准确。

很显然，仅仅提供 delay_xms 函数，是不够用的，很多时候，我们延时都是大于 798ms 的，所以需要再做一个 delay_ms 函数，下面将介绍该函数。

5.1.4 delay_ms 函数

该函数同 delay_xms 一样，也是用来延时指定的 ms 的，其参数 nms 为要延时的毫秒数。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 nms ,nms:0~65535
void delay_ms(u16 nms)
{
    u8 repeat=nms/540;//这里用 540,是考虑到某些客户可能超频使用,
                      //比如超频到 248M 的时候,delay_xms 最大只能延时 541ms 左右
    u16 remain=nms%540;
    while(repeat)
    {
        delay_xms(540); repeat--;
    }
    if(remain)delay_xms(remain);
}
```

该函数其实就是多次调用前面所讲的 delay_xms 函数，来实现毫秒级延时的。注意下，这里以 540ms 为周期是考虑到 MCU 超频使用的情况。

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,nms:0~65535
void delay_ms(u16 nms)
{
    if(OSRunning==OS_TRUE&&OSLockNesting==0)//os 在跑了? &&OSLockNesting==0?
    {
        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（达到 204 秒）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 STM32F4 的 IO 口输入读取宏定义和输出宏定义。sys.c 里面主要是一些汇编函数。下面我们主要向大家介绍 sys.h 头文件里面的 IO 口位操作。

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 STM32F4 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 GPIO 的 ODR 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。

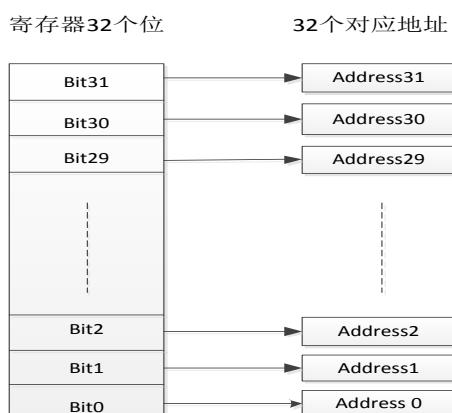


图 5.2.2.1 位带映射图

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:

```
//位带操作,实现 51 类似的 GPIO 控制功能
//具体实现思想,参考<<CM3 权威指南>>第五章(87 页~92 页).M4 同 M3 类似,
//只是寄存器地址变了.

//IO 口操作宏定义
#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+20) //0x40020014
```

```

#define GPIOB_ODR_Addr      (GPIOB_BASE+20) //0x40020414
.....//省略部分代码
#define GPIOH_ODR_Addr      (GPIOH_BASE+20) //0x40021C14
#define GPIOI_ODR_Addr      (GPIOI_BASE+20) //0x40022014

#define GPIOA_IDR_Addr      (GPIOA_BASE+16) //0x40020010
#define GPIOB_IDR_Addr      (GPIOB_BASE+16) //0x40020410
.....//省略部分代码
#define GPIOH_IDR_Addr      (GPIOH_BASE+16) //0x40021C10
#define GPIOI_IDR_Addr      (GPIOI_BASE+16) //0x40022010

//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
.....//省略部分代码
#define PHout(n)    BIT_ADDR(GPIOH_ODR_Addr,n) //输出
#define PHin(n)     BIT_ADDR(GPIOH_IDR_Addr,n) //输入
#define PIout(n)    BIT_ADDR(GPIOI_ODR_Addr,n) //输出
#define PIin(n)     BIT_ADDR(GPIOI_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 个位是否等于 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）。

5.3 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 或其他的串口，只要对代码稍作修改就可以了。usart.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.c 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。这段代码如果要修改一般也只是用来改变 printf 函数针对的串口号，大多情况下我们都不需要修改。

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 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。uart_init 函数代码如下：

```
//初始化 IO 串口 1
//bound:波特率
void uart_init(u32 bound){
    //GPIO 端口设置
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
```

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE);  
                                //使能 GPIOA 时钟  
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE);  
                                //使能 USART1 时钟  
  
GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1);  
                                //GPIOA9 复用为 USART1  
GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1);  
                                //GPIOA10 复用为 USART1  
  
//USART1 PA.9 PA.10  
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //GPIOA9 与 GPIOA10  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉  
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA9, PA10  
  
//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  
USART_Cmd(USART1, ENABLE); //使能串口  
USART_ClearFlag(USART1, USART_FLAG_TC);  
#if EN_USART1_RX  
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);//开启中断  
  
//Usart1 NVIC 配置  
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;  
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=3;//抢占优先级 3  
NVIC_InitStructure.NVIC_IRQChannelSubPriority =3; //响应优先级 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、  
#endif  
}
```

下面我们一一分析一下这段初始化代码。首先是时钟使能代码：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 GPIOA 时钟
```

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE); //使能 USART1 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，然后使能复用功能外设时钟。

然后我们要配置相应的引脚复用器映射。这里我们调用函数为：

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1); //PA9 复用为 USART1
```

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1); //PA10 复用为 USART1
```

把 PA9 和 PA10 复用为串口 1。

接下来我们要初始化相应的 GPIO 端口模式 (GPIO_Mode) 为复用功能。配置方法如下：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //GPIOA9 与 GPIOA10
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
```

```
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
```

```
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
```

```
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
```

```
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 PA9, PA10
```

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响应优先级的值：

```
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn; //Usart1 中断配置
```

```
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3; //抢占优先级 3
```

```
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //响应优先级 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_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_stm32f40_41xxx.s` 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。
```

`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。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。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 0x0d 0x0a 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 0x0d  
            {  
                if(Res!=0x0a)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==0x0d)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  
}  
#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_TICKS_PER_SEC ,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 STM32F4 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 STM32F4。

STM32F4 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 STM32F4 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 STM32F4 的各种外设，通过本篇的学习，希望大家能学会 STM32F4 绝大部分外设的使用。

本篇总共分为 59 章，每一章即一个实例，下面就让我们开始精彩的 STM32F4 之旅。

第六章 跑马灯实验

任何一个单片机，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 STM32F4 之旅，通过本章的学习，你将了解到 STM32F4 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 探索者 STM32F4 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, STM32F4 IO 口简介
- 6.2, 硬件设计
- 6.3, 软件设计
- 6.4, 下载验证

6.1 STM32F4 IO 简介

本章将要实现的是控制 ALIENTEK 探索者 STM32F4 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 STM32F4 的 IO 口输出。了解了 STM32F4 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 STM32F4 基本 IO 口的使用，而这是迈向 STM32F4 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 STM32F4 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-库函数版本\实验 1 跑马灯\USER\LED.uvproj”），可以看到我们的实验工程目录：

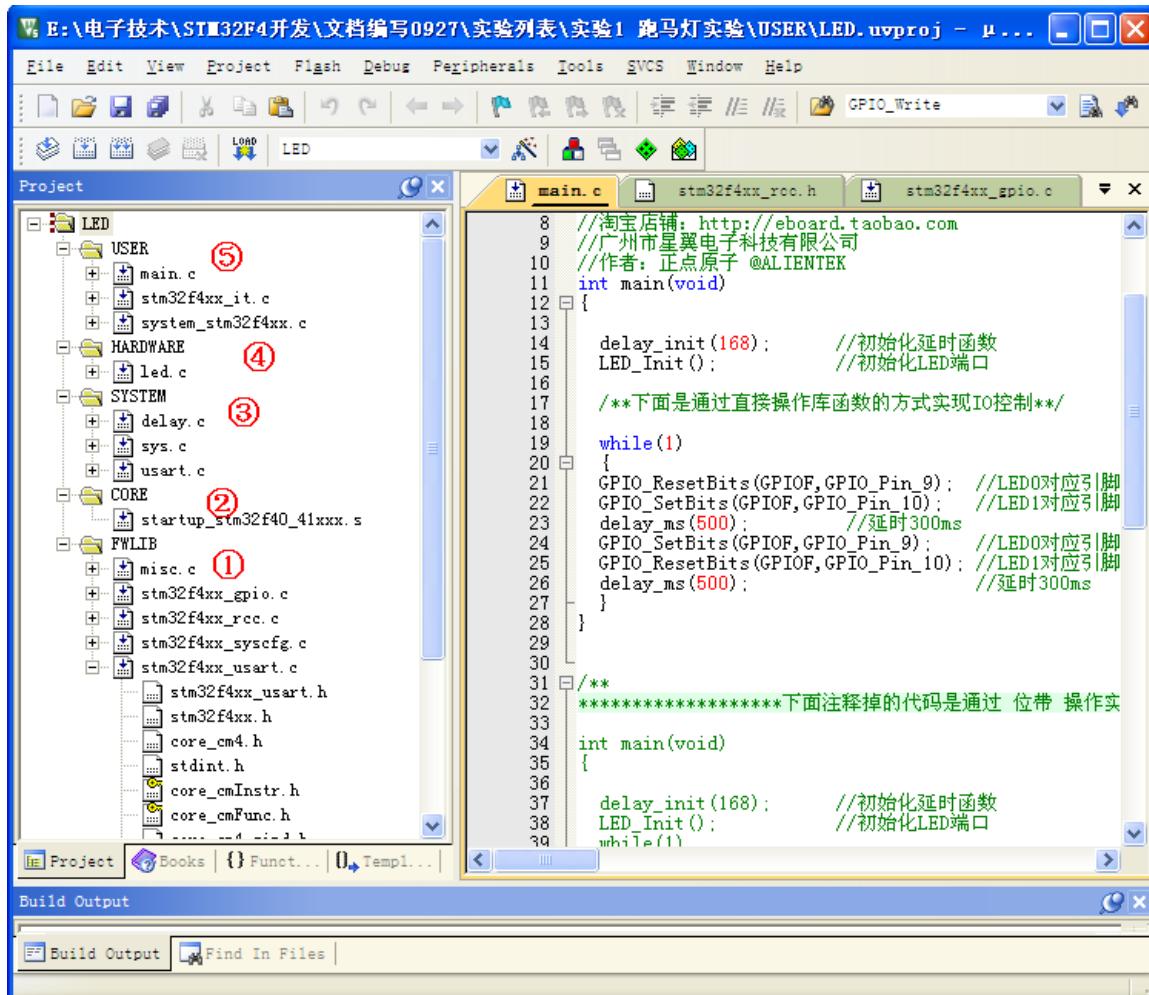


图 6.1.1 跑马灯实验目录结构

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，每一个源文件 stm32f4xx_ppp.c 都对应一个头文件 stm32f4xx_ppp.h。分组内的文件我们可以根据工程需要添加和删除，但是一定要注意如果你引入了某个源文件，一定要在头文件 stm32f4xx_conf.h 文件中确保对应的头文件也已经添加。比如我们跑马灯实验，我们只添加了 5 个源文件，那么对应的头文件我们必须确保在 stm32f4xx_conf.h 内也包含进来，否则工程会报错。
- 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。大家可以根据自己的芯片型号选择对应的启动文件。
- 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 stm32f4xx_gpio.c 内定义的函数对 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 下面存放的主要时用户代码。但是 system_stm32f4xx.c 文件用户不需要修改，同时 stm32f4xx_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。

工程分组情况我们就讲解到这里，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 stm32f4xx_gpio.h 和 stm32f4xx_gpio.c 中。

相对于 STM32F1 来说，STM32F4 的 GPIO 设置显得更为复杂，也更加灵活，尤其是复用功能部分，比 STM32F1 改进了很多，使用起来更加方便。

STM32F4 每组通用 I/O 端口包括 4 个 32 位配置寄存器(MODER、OTYPER、OSPEEDR 和 PUPDR)、2 个 32 位数据寄存器(IDR 和 ODR)、1 个 32 位位置位/复位寄存器(BSRR)、1 个 32 位锁定寄存器(LCKR) 和 2 个 32 位复用功能选择寄存器(AFRH 和 AFRL) 等。

这样，STM32F4 每组 IO 有 10 个 32 位寄存器控制，其中常用的有 4 个配置寄存器+2 个数据寄存器+2 个复用功能选择寄存器，共 8 个，如果在使用的时候，每次都直接操作寄存器配置 IO，代码会比较多，也不容易记住，所以我们在讲解寄存器的同时会讲解是用库函数配置 IO 的方法。

同 STM32F1 一样，STM32F4 的 IO 可以由软件配置成如下 8 种模式中的任何一种：

- 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、开漏式复用功能

关于这些模式的介绍及应用场景，我们这里就不详细介绍了，感兴趣的朋友，可以看看这个帖子了解下：<http://www.openedv.com/posts/list/32730.htm>。接下来我们详细介绍 IO 配置常用的 8 个寄存器：MODER、OTYPER、OSPEEDR、PUPDR、ODR、IDR、AFRH 和 AFRL。同时讲解对应的库函数配置方法。

首先看 MODER 寄存器，该寄存器是 GPIO 端口模式控制寄存器，用于控制 GPIOx (STM32F4 最多有 9 组 IO，分别用大写字母表示，即 x=A/B/C/D/E/F/G/H/I，下同) 的工作模式，该寄存器各位描述如表表 6.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw										

MODERy[1:0]: 端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 方向模式。

- 00: 输入（复位状态）
- 01: 通用输出模式
- 10: 复用功能模式
- 11: 模拟模式

表 6.1.1 GPIOx MODER 寄存器各位描述

该寄存器各位在复位后，一般都是 0 (个别不是 0，比如 JTAG 占用的几个 IO 口)，也就是默认条件下一般是输入状态的。每组 IO 下有 16 个 IO 口，该寄存器共 32 位，每 2 个位控制 1

个 IO，不同设置所对应的模式见表 5.2.5.1 描述。

然后看 OTYPER 寄存器，该寄存器用于控制 GPIOx 的输出类型，该寄存器各位描述见表表 6.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 保留，必须保持复位值。

位 15:0 **OTy[1:0]**: 端口 x 配置位 (Port x configuration bits) ($y = 0..15$)

这些位通过软件写入，用于配置 I/O 端口的输出类型。

0: 输出推挽 (复位状态)

1: 输出开漏

表 6.1.2 GPIOx OTYPER 寄存器各位描述

该寄存器仅用于输出模式，在输入模式（MODER[1:0]=00/11 时）下不起作用。该寄存器低 16 位有效，每一个位控制一个 IO 口，复位后，该寄存器值均为 0。

然后看 OSPEEDR 寄存器，该寄存器用于控制 GPIOx 的输出速度，该寄存器各位描述见表表 6.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]	OSPEEDR14[1:0]	OSPEEDR13[1:0]	OSPEEDR12[1:0]	OSPEEDR11[1:0]	OSPEEDR10[1:0]	OSPEEDR9[1:0]	OSPEEDR8[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
OSPEEDR7[1:0]	OSPEEDR6[1:0]	OSPEEDR5[1:0]	OSPEEDR4[1:0]	OSPEEDR3[1:0]	OSPEEDR2[1:0]	OSPEEDR1[1:0]	OSPEEDR0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

OSPEEDRy[1:0]: 端口 x 配置位 (Port x configuration bits) ($y = 0..15$)

这些位通过软件写入，用于配置 I/O 输出速度。

00: 2 MHz (低速)

01: 25 MHz (中速)

10: 50 MHz (快速)

11: 30 pF 时为 100 MHz (高速) (15 pF 时为 80 MHz 输出 (最大速度))

表 6.1.3 GPIOx OSPEEDR 寄存器各位描述

该寄存器也仅用于输出模式，在输入模式（MODER[1:0]=00/11 时）下不起作用。该寄存器每 2 个位控制一个 IO 口，复位后，该寄存器值一般为 0。

然后看 PUPDR 寄存器，该寄存器用于控制 GPIOx 的上拉/下拉，该寄存器各位描述见表表 6.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]	PUPDR14[1:0]	PUPDR13[1:0]	PUPDR12[1:0]	PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]	PUPDR6[1:0]	PUPDR5[1:0]	PUPDR4[1:0]	PUPDR3[1:0]	PUPDR2[1:0]	PUPDR1[1:0]	PUPDR0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

PUPDRy[1:0]: 端口 x 配置位 (Port x configuration bits) ($y = 0..15$)

这些位通过软件写入，用于配置 I/O 上拉或下拉。

00: 无上拉或下拉

01: 上拉

10: 下拉

11: 保留

表 6.1.4 GPIOx PUPDR 寄存器各位描述

该寄存器每 2 个位控制一个 IO 口，用于设置上下拉，这里提醒大家，STM32F1 是通过 ODR 寄存器控制上下拉的，而 STM32F4 则由单独的寄存器 PUPDR 控制上下拉，使用起来更加灵活。复位后，该寄存器值一般为 0。

前面，我们讲解了 4 个重要的配置寄存器。顾名思义，配置寄存器就是用来配置 GPIO 的相关模式和状态，接下来我们讲解怎么在库函数初始化 GPIO 的配置。

GPIO 相关的函数和定义分布在固件库文件 `stm32f4xx_gpio.c` 和头文件 `stm32f4xx_gpio.h` 文件中。

在固件库开发中，操作四个配置寄存器初始化 GPIO 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需要初始化的 GPIO 对应的 GPIO 组，取值范围为 GPIOA~GPIOK。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 `stm32f4xx_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
{
    uint32_t GPIO_Pin;
    GPIO_Mode_TypeDef GPIO_Mode;
    GPIO_Speed_TypeDef GPIO_Speed;
    GPIO_OType_TypeDef GPIO_OType;
    GPIO_PuPd_TypeDef GPIO_PuPd;
}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_9//GPIOF9
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOF, &GPIO_InitStructure); //初始化 GPIO
```

上面代码的意思是设置 GPIOF 的第 9 个端口为推挽输出模式，同时速度为 100M，上拉。

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，这个很好理解；第二个成员变量 `GPIO_Mode` 是用来设置对应 IO 端口的输出输入端口模式，这个值实际就是配置我们前面讲解的 `GPIOx` 的 MODER 寄存器的值。在 MDK 中是通过一个枚举类型定义的，我们只需要选择对应的值即可：

```
typedef enum
{
    GPIO_Mode_IN    = 0x00, /*!< GPIO Input Mode */
    GPIO_Mode_OUT   = 0x01, /*!< GPIO Output Mode */
    GPIO_Mode_AF    = 0x02, /*!< GPIO Alternate function Mode */
    GPIO_Mode_AN    = 0x03  /*!< GPIO Analog Mode */
}GPIOMode_TypeDef;
```

GPIO_Mode_IN 是用来设置为复位状态的输入， GPIO_Mode_OUT 是通用输出模式， GPIO_Mode_AF 是复用功能模式， GPIO_Mode_AN 是模拟输入模式。

第三个参数 GPIO_Speed 是 IO 口输出速度设置，有四个可选值。实际上这就是配置的 GPIO 对应的 OSPEEDR 寄存器的值。在 MDK 中同样通过枚举类型定义：

```
typedef enum
{
    GPIO_Low_Speed      = 0x00, /*!< Low speed      */
    GPIO_Medium_Speed   = 0x01, /*!< Medium speed   */
    GPIO_Fast_Speed     = 0x02, /*!< Fast speed     */
    GPIO_High_Speed     = 0x03  /*!< High speed     */

}GPIOSpeed_TypeDef;

/* Add legacy definition */
#define GPIO_Speed_2MHz     GPIO_Low_Speed
#define GPIO_Speed_25MHz    GPIO_Medium_Speed
#define GPIO_Speed_50MHz    GPIO_Fast_Speed
#define GPIO_Speed_100MHz   GPIO_High_Speed
```

这里需要说明一下，实际我们的输入可以是 GPIOSpeed_TypeDef 枚举类型中 GPIO_High_Speed 枚举类型值，也可以是 GPIO_Speed_100MHz 这样的值，实际上 GPIO_Speed_100MHz 就是通过 define 宏定义标识符定义出来的，它跟 GPIO_High_Speed 是等同的。

第四个参数 GPIO_OType 是 GPIO 的输出类型设置，实际上是配置的 GPIO 的 OTYPER 寄存器的值。在 MDK 中同样通过枚举类型定义：

```
typedef enum
{
    GPIO_OType_PP = 0x00,
    GPIO_OType_OD = 0x01
}GPIOOType_TypeDef;
```

如果需要设置为输出推挽模式，那么选择值 GPIO_OType_PP，如果需要设置为输出开漏模式，那么设置值为 GPIO_OType_OD。

第五个参数 GPIO_PuPd 用来设置 IO 口的上下拉，实际上就是设置 GPIO 的 PUPDR 寄存器的值。同样通过一个枚举类型列出：

```
typedef enum
{
    GPIO_PuPd_NOPULL = 0x00,
    GPIO_PuPd_UP      = 0x01,
    GPIO_PuPd_DOWN    = 0x02
}GPIOPuPd_TypeDef;
```

这三个值的意思很好理解，GPIO_PuPd_NOPULL 为不使用上下拉，GPIO_PuPd_UP 为上拉，GPIO_PuPd_DOWN 为下拉。我们根据我们需要设置相应的值即可。

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。

看完了 GPIO 的参数配置寄存器，接下来我们看看 GPIO 输入输出电平控制相关的寄存器。

首先我们看 ODR 寄存器，该寄存器用于控制 GPIOx 的输出，该寄存器各位描述见表 6.1.5 所示：

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

位 31:16 保留，必须保持复位值。

位 15:0 **ODRy[15:0]**: 端口输出数据 (Port output data) ($y = 0..15$)
这些位可通过软件读取和写入。

表 6.1.5 GPIOx ODR 寄存器各位描述

该寄存器用于设置某个 IO 输出低电平($ODRy=0$)还是高电平($ODRy=1$)，该寄存器也仅在输出模式下有效，在输入模式（MODER[1:0]=00/11 时）下不起作用。

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。

使用实例如下：

```
GPIO_Write(GPIOA,0x0000);
```

大部分情况下，设置 IO 口我们都不用这个函数，后面我们会讲解我们常用的设置 IO 口电平的函数。

同时读 ODR 寄存器还可以读出 IO 口的输出状态，库函数为：

```
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

这两个函数功能类似，只不过前面是用来一次读取一组 IO 口所有 IO 口输出状态，后面的函数用来一次读取一组 IO 口中一个或者几个 IO 口的输出状态。

接下来我们看看 IDR 寄存器，该寄存器用于读取 GPIOx 的输入，该寄存器各位描述见表 6.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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，必须保持复位值。

位 15:0 **IDRy[15:0]**: 端口输入数据 (Port input data) ($y = 0..15$)
这些位为只读形式，只能在字模式下访问。它们包含相应 I/O 端口的输入值。

表 6.1.6 GPIOx IDR 寄存器各位描述

该寄存器用于读取某个 IO 的电平，如果对应的位为 0($IDRy=0$)，则说明该 IO 输入的是低电平，如果是 1($IDRy=1$)，则表示输入的是高电平。库函数相关函数为：

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

前面的函数是用来读取一组 IO 口的一个或者几个 IO 口输入电平，后面的函数用来一次读取一

组 IO 口所有 IO 口的输入电平。比如我们要读取 GPIOF.5 的输入电平，方法为：

```
GPIO_ReadInputDataBit(GPIOF, GPIO_Pin_5);
```

接下来我们看看 32 位位置位/复位寄存器 (BSRR)，顾名思义，这个寄存器是用来置位或者复位 IO 口，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。寄存器描述如下：

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**: 端口 x 复位位 y (Port x reset bit y) (y = 0..15)

这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。

0: 不会对相应的 ODRx 位执行任何操作

1: 对相应的 ODRx 位进行复位

注意：如果同时对 BSx 和 BRx 置位，则 BSx 的优先级更高。

位 15:0 **BSy**: 端口 x 置位位 y (Port x set bit y) (y = 0..15)

这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。

0: 不会对相应的 ODRx 位执行任何操作

1: 对相应的 ODRx 位进行置位

表 6.1.7 BSRR 寄存器各位描述

对于低 16 位 (0-15)，我们往相应的位写 1，那么对应的 IO 口会输出高电平，往相应的位写 0，对 IO 口没有任何影响。高 16 位 (16-31) 作用刚好相反，对相应的位写 1 会输出低电平，写 0 没有任何影响。也就是说，对于 BSRR 寄存器，你写 0 的话，对 IO 口电平是没有任何影响的。我们要设置某个 IO 口电平，只需要为相关位设置为 1 即可。而 ODR 寄存器，我们要设置某个 IO 口电平，我们首先需要读出来 ODR 寄存器的值，然后对整个 ODR 寄存器重新赋值来打到设置某个或者某些 IO 口的目的，而 BSRR 寄存器，我们就不需要先读，而是直接设置。

BSRR 寄存器使用方法如下：

```
GPIOA->BSRR=1<<1; //设置 GPIOA.1 为高电平
```

```
GPIOA->BSRR=1<<(16+1) //设置 GPIOA.1 为低电平;
```

库函数操作 BSRR 寄存器来设置 IO 电平的函数为：

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

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

函数 GPIO_SetBits 用来设置一组 IO 口中的一个或者多个 IO 口为高电平。GPIO_ResetBits 用来设置一组 IO 口中一个或者多个 IO 口为低电平。比如我们要设置 GPIOB.5 输出高，方法为：

```
GPIO_SetBits(GPIOB,GPIO_Pin_5);//GPIOB.5 输出高
```

设置 GPIOB.5 输出低电平，方法为：

```
GPIO_ResetBits(GPIOB,GPIO_Pin_5);//GPIOB.5 输出低
```

最后我们来看看 2 个 32 位复用功能选择寄存器 (AFRH 和 AFRL)，这两个寄存器是用来设置 IO 口的复用功能的。关于这两个寄存器的配置以及相关库函数的使用，在我们前面章节 4.4 IO 引脚复用和映射有详细讲解，这里我们就不做过多的说明。

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：

- 使能 IO 口时钟。调用函数为 RCC_AHB1PeriphClockCmd ()。

- 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 STM32F4 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 探索者 STM32F4 开发板上默认是已经连接好了的。DS0 接 PF9，DS1 接 PF10。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：

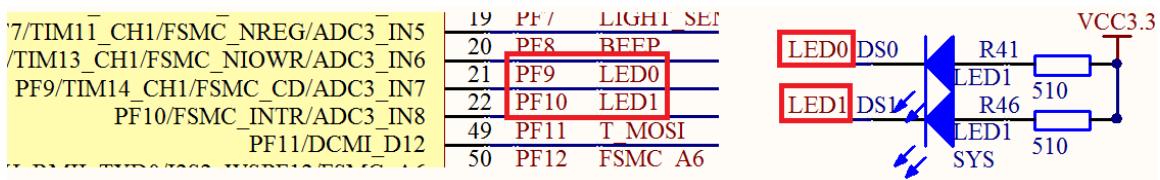


图 6.2.1 LED 与 STM32F4 连接原理图

6.3 软件设计

这是我们的第一个实验，所以我教大家怎么从我们前面讲解的 Template 工程一步一步加入我们的固件库以及我们的 led 相关的驱动函数到我们工程，使之跟我们光盘的跑马灯实验工程一模一样。首先大家打开我们 3.3.2 小节新建的库函数版本工程模板。如果您还没有新建，也可以直接打开我们光盘已经新建好了的工程模板，路径为：“**|4, 程序源码\标准例程-库函数版本\实验 0 Template 工程模板**”。注意，是直接点击工程下面的 USER 目录下面的 **Template.uvproj**。

大家可以看到，我们模板里面的 FWLIB 分组下面，我们引入了所有的固件库源文件和对应的头文件，如下图 6.3.1：

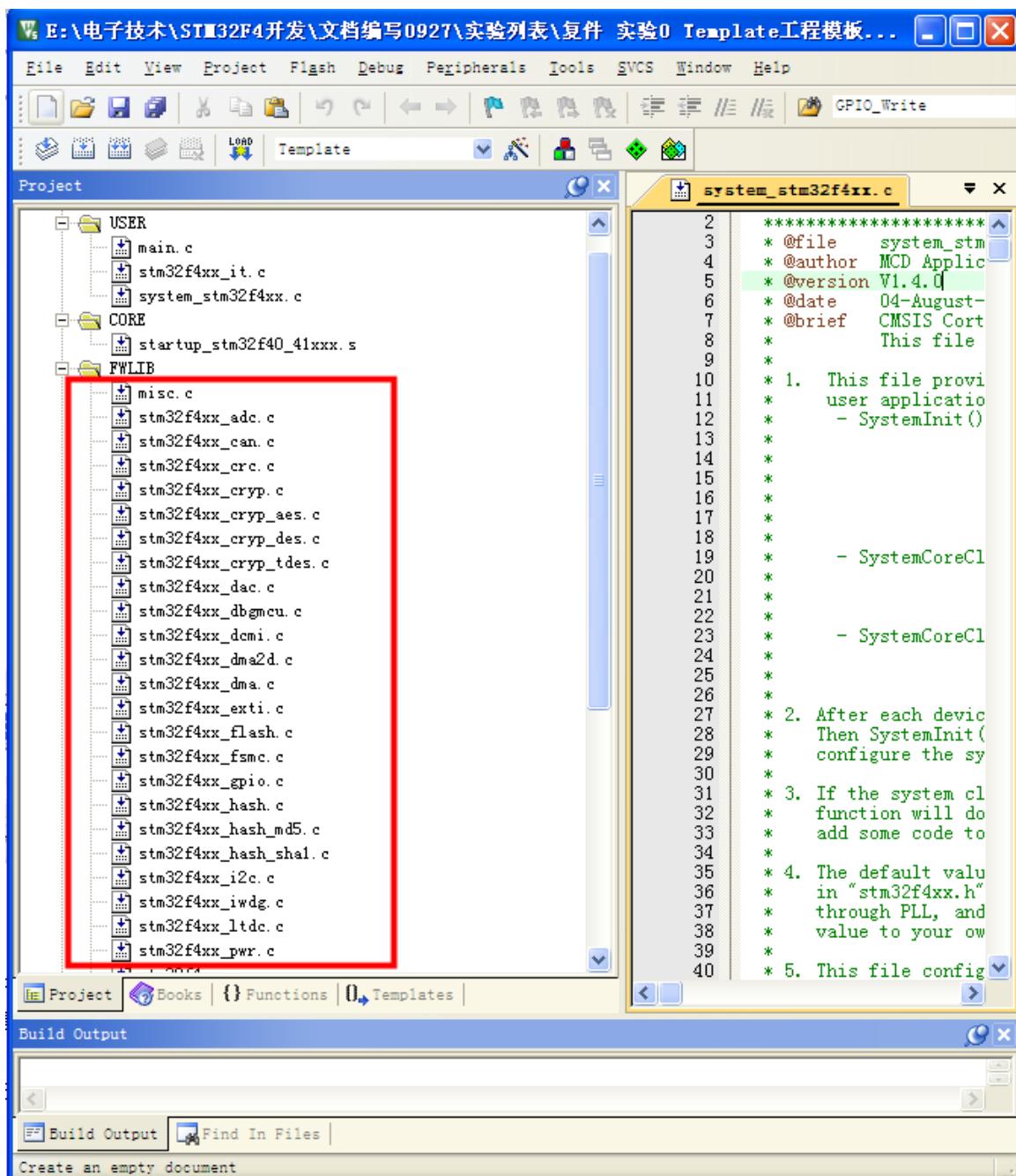


图 6.3.1 Template 模板工程结构

实际上，这些大家可以根据工程需要添加，比如我们跑马灯实验，我们并没有用到 ADC，自然我们可以去掉 `stm32f4xx_adc.c`，这样可以减少工程编译时间。

跑马灯实验我们主要用到的固件库文件是：

```

stm32f4xx_gpio.c /stm32f4xx_gpio.h
stm32f4xx_rcc.c/stm32f4xx_rcc.h
misc.c/ misc.h
stm32f4xx_usart.c/stm32f4xx_usart.h
stm32f4xx_syscfg.c/stm32f4xx_syscfg.h

```

其中 `stm32f4xx_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其源文件 `stm32f4xx_rcc.c` 中。`stm32f4xx_usart.h` 和 `misc.h` 头文件和对应的源文件在我们 `SYSTEM` 文件夹中都需要使用到，所以每个实验都会引用。`stm32f4xx_syscfg.h` 和对应的源文件虽然本实验也没有用到，但是后面很多实验都要使用到，所以我们不妨也添加进来。

在 `stm32f4xx_conf.h` 文件里面，这些头文件默认都是打开的，实际我们可以不用理。当然我们也可以注释掉其他不用的头文件，但是如果你引入了某个源文件，一定不能不包含对应的头文件：

```
#include "stm32f4xx_gpio.h"
#include "stm32f4xx_rcc.h"
#include "stm32f4xx_usart.h"
#include "stm32f4xx_syscfg.h"
#include "misc.h"
```

接下来，我们讲解怎样去掉多余的其他的源文件，方法如下图，右击 `Template`，选择“**Manage project Items**”，进入这个选项卡：

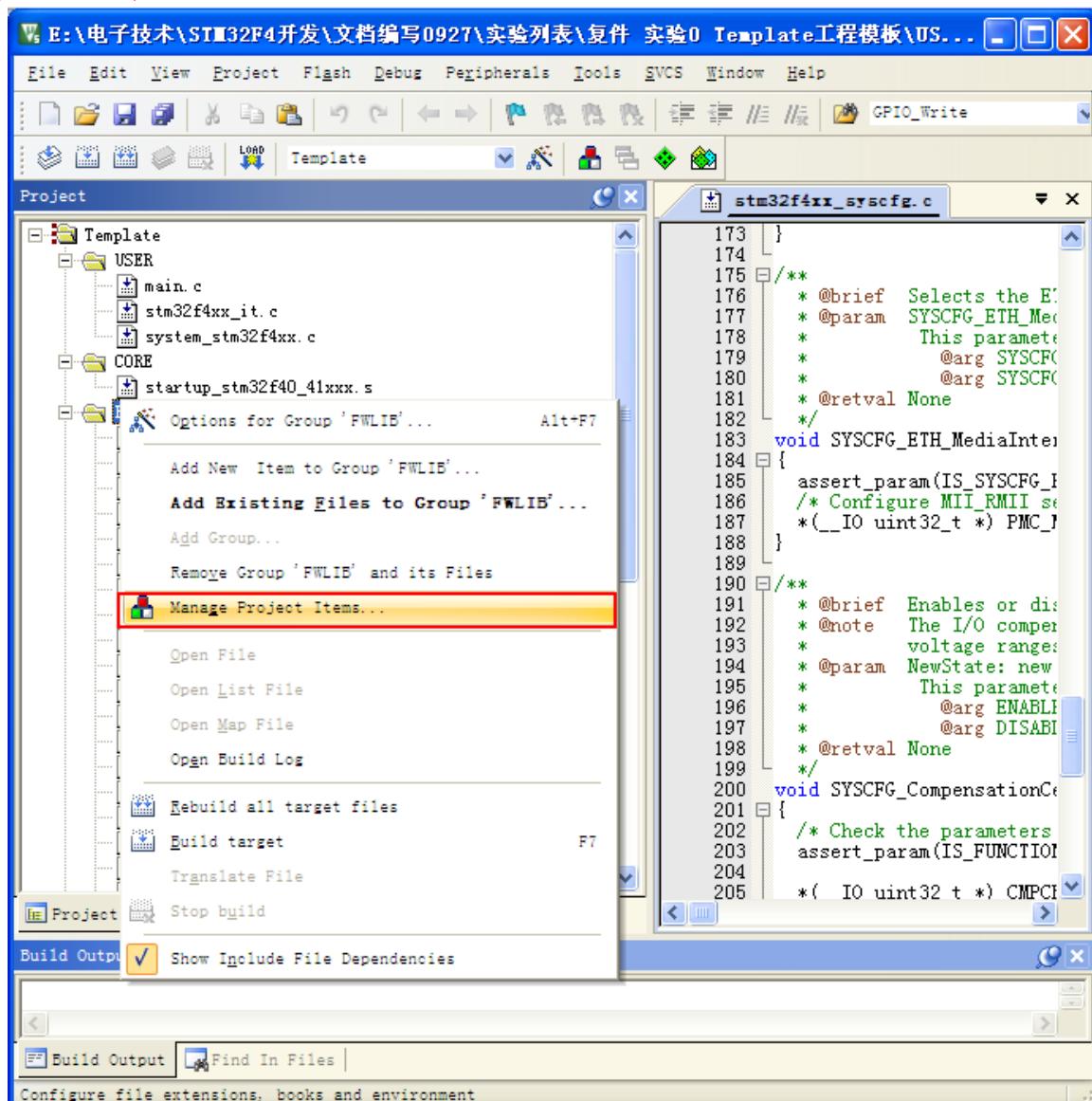


图 6.3.2

我们选中“FWLIB”分组，然后选中不需要的源文件点击删除按钮删掉，留下下图中我们使用到的五个源文件，然后点击 OK：

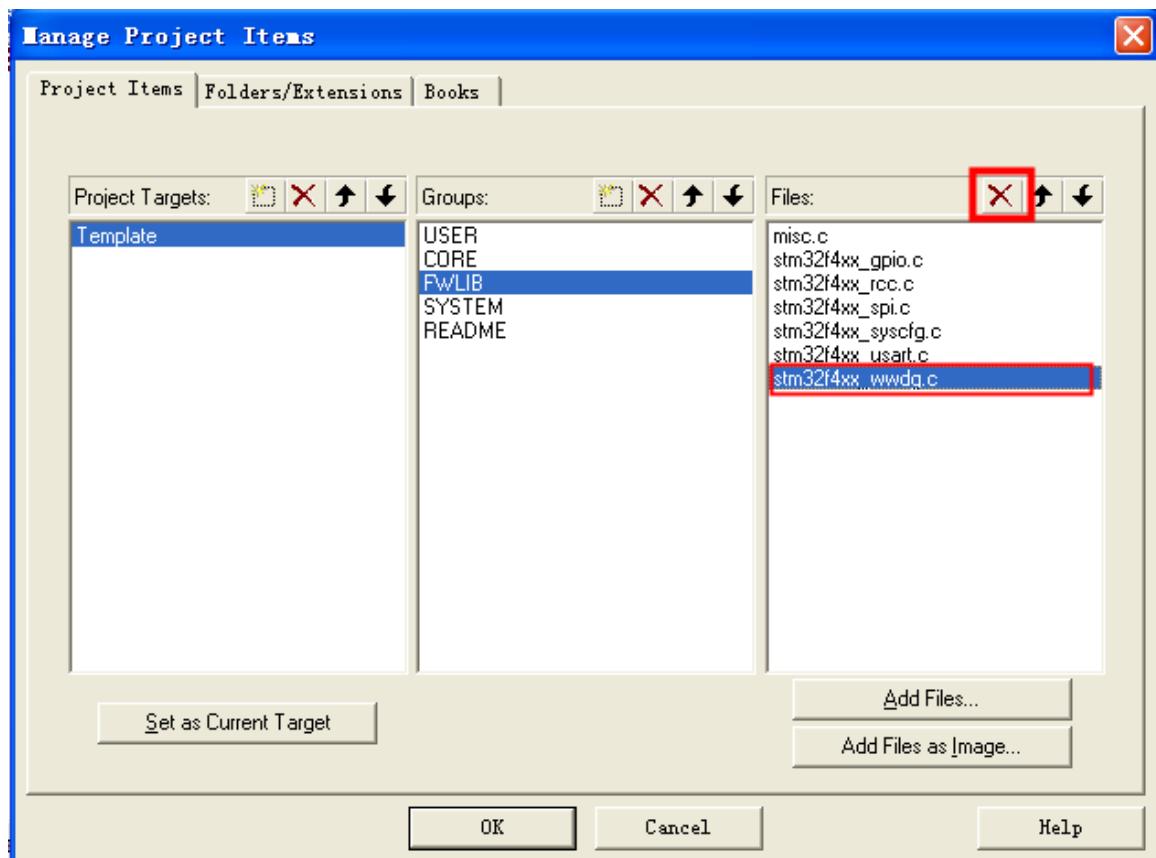


图 6.3.3

这样我们的工程 FWLIB 下面只剩下五个源文件：

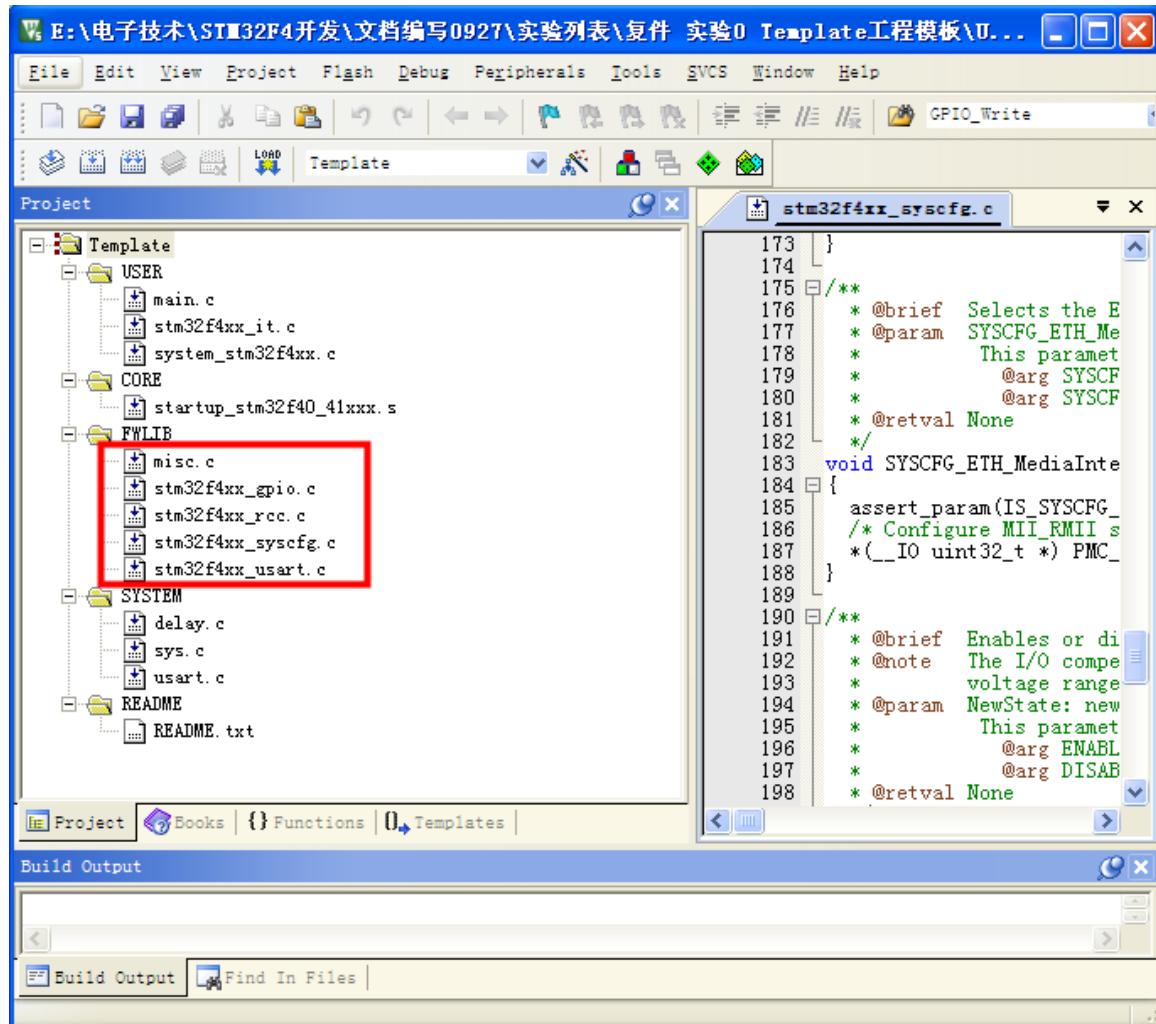


图 6.3.4

然后我们进入我们工程的目录，在工程根目录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 6.3.5 所示：

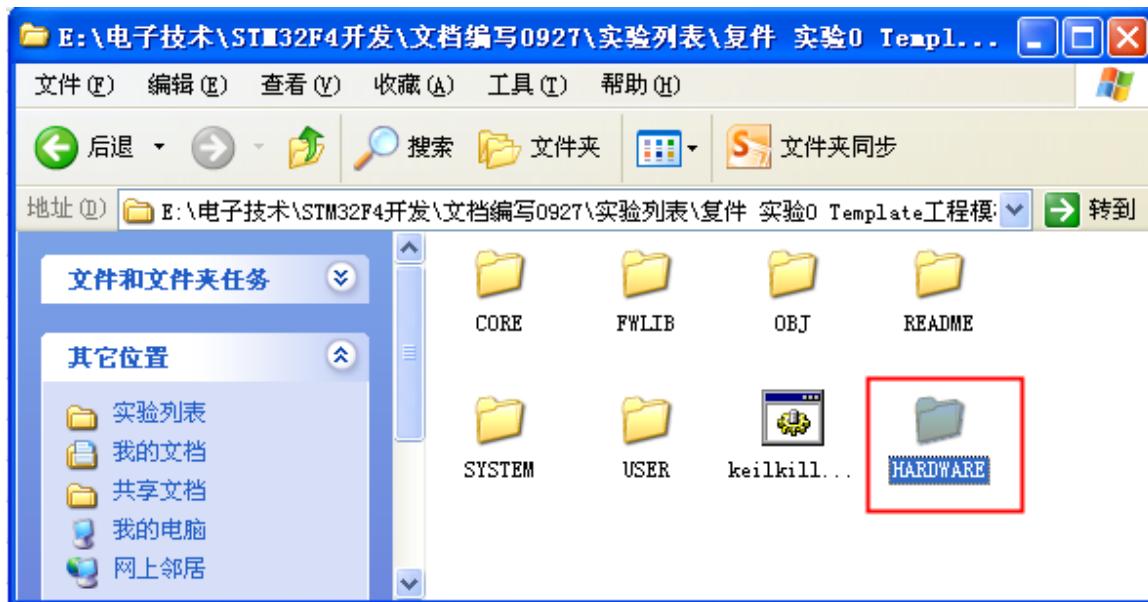


图 6.3.5 新建 HARDWARE 文件夹

接下来，我们回到我们的工程(如果是使用的上面新建的工程模板，那么就是 Template.uvproj，大家可以将其重命名为 LED.uvproj)，按 按钮新建一个文件，然后按 保存在 HARDWARE->LED 文件夹下面，保存为 led.c，操作步骤如下图：

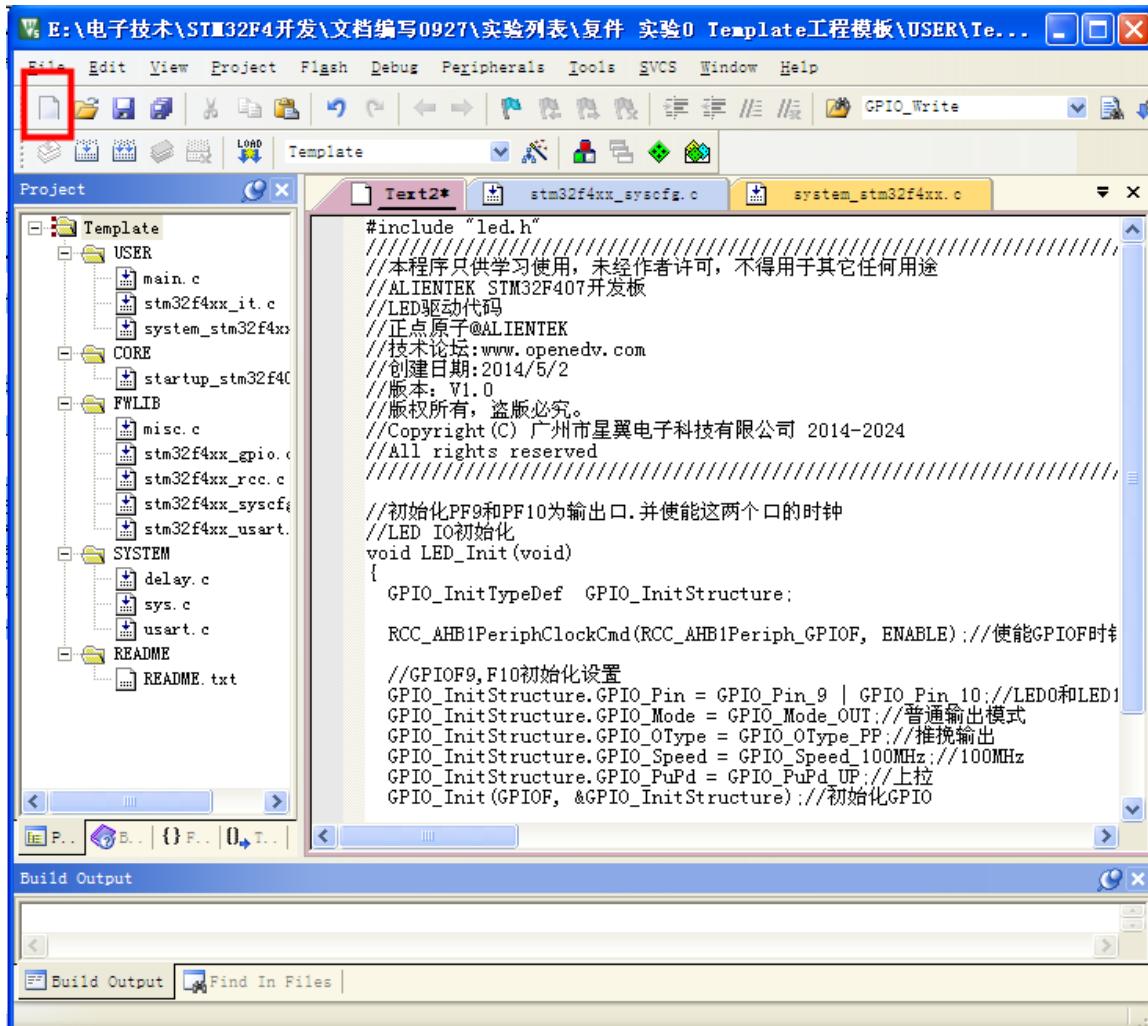


图 6.3.6 新建文件

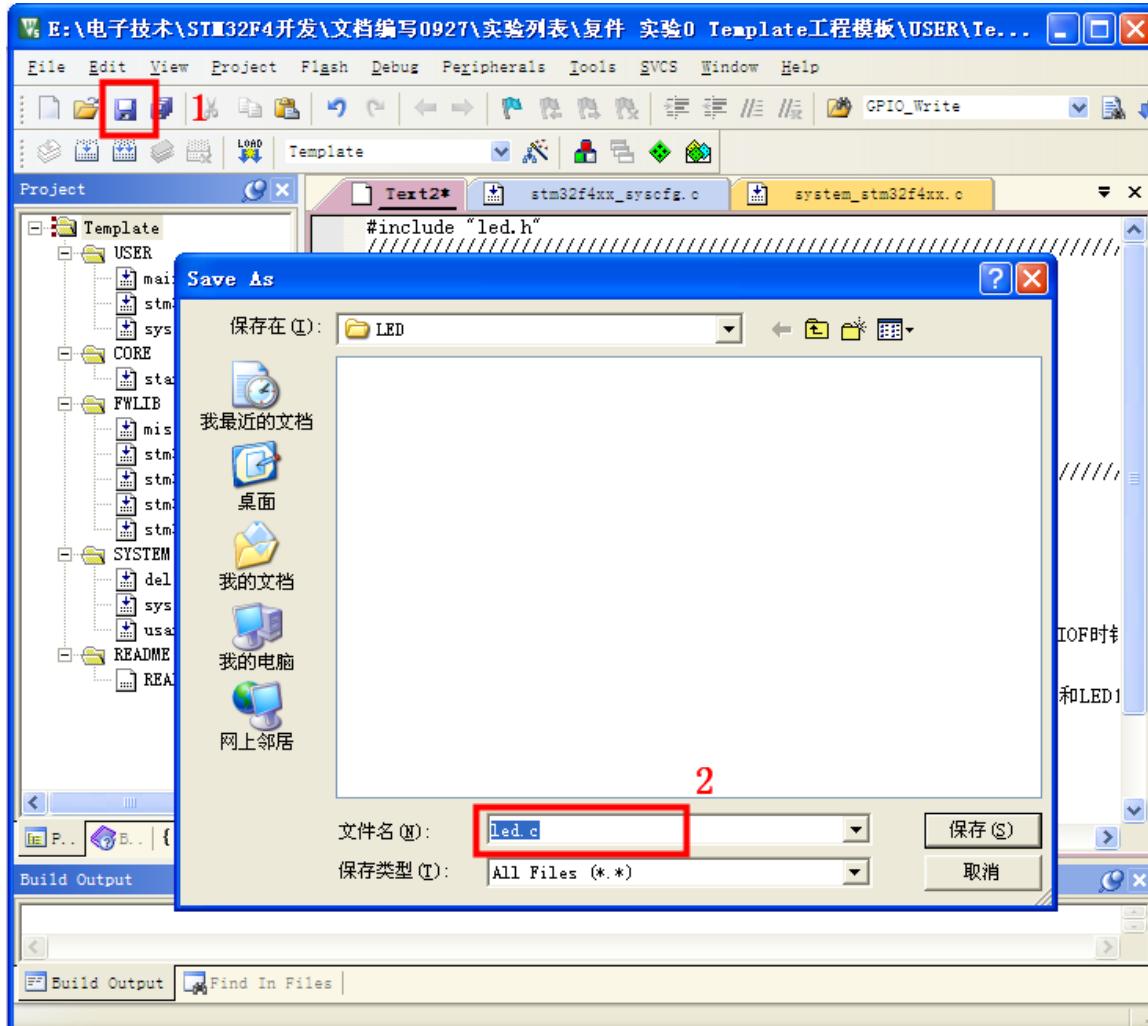


图 6.3.7 保存 led.c

然后在 lcd.c 文件中输入如下代码(**代码大家可以打开我们光盘的跑马灯实验，从相应的文件中复制过来**)，输入后保存即可：

```
#include "led.h"
//初始化 PF9 和 PF10 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_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE); //使能 GPIOF 时钟
    //GPIOF9,F10 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //LED0 和 LED1 对应 IO 口
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //普通输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOF, &GPIO_InitStructure); //初始化 GPIO
```

```
GPIO_SetBits(GPIOF,GPIO_Pin_9 | GPIO_Pin_10); //GPIOF9,F10 设置高, 灯灭
```

```
}
```

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 PF9 和 PF10 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 AHB1 总线上的外设，在固件库中对挂载在 AHB1 总线上的外设时钟使能是通过函数 RCC_AHB1PeriphClockCmd () 来实现的。对于这个入口参数设置，在我们前面的“4.7 快速组织代码”章节已经讲解很清楚了。看看我们的代码：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE); //使能 GPIOF 时钟  
这行代码的作用是使能 AHB1 总线上的 GPIOF 时钟。
```

在设置完时钟之后，LED_Init 调用 GPIO_Init 函数完成对 PF9 和 PF10 的初始化配置，然后调用函数 GPIO_SetBits 控制 LED0 和 LED1 输出 1 (LED 灭)。至此，两个 LED 的初始化完毕。这样就完成了对这两个 IO 口的初始化。这段代码的具体含义，大家可以看前面一小节，我们有详细的讲解。初始化函数代码如下：

```
//GPIOF9,F10 初始化设置  
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //LED0 和 LED1 对应 IO 口  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //普通输出模式  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉  
GPIO_Init(GPIOF, &GPIO_InitStructure); //初始化 GPIO
```

```
GPIO_SetBits(GPIOF,GPIO_Pin_9 | GPIO_Pin_10); //GPIOF9,F10 设置高, 灯灭
```

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 PFout(9) // DS0  
#define LED1 PFout(10) // 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 PFout(9) // DS0 PF9  
#define LED1 PFout(10) // DS1 PF10
```

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(GPIOF, GPIO_Pin_9); //设置 GPIOF.9 输出 1,等同 LED0=1;  
GPIO_ResetBits(GPIOF, GPIO_Pin_9); //设置 GPIOF.9 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 6.3.8 所示：

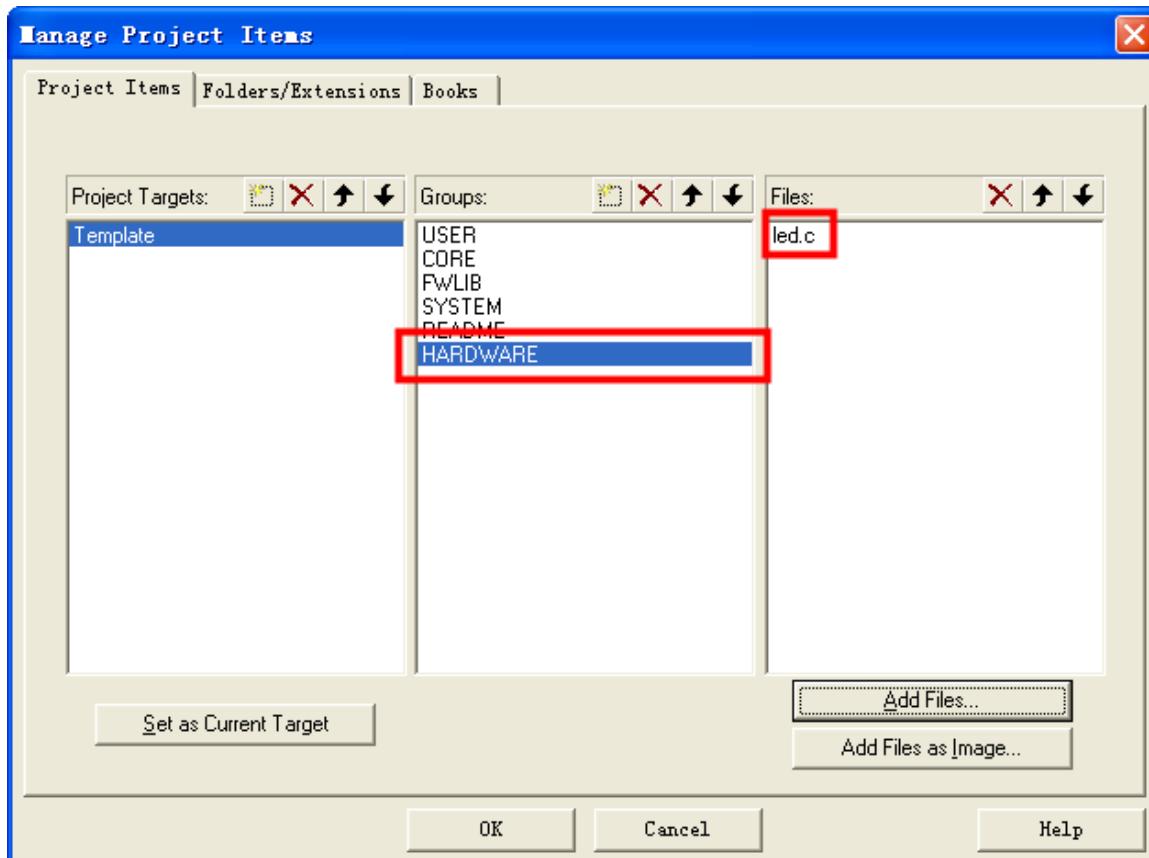


图 6.3.8 给工程新增 HARDWARE 组

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，在该组下面有一个 led.c 的文件。如图 6.3.9 所示：

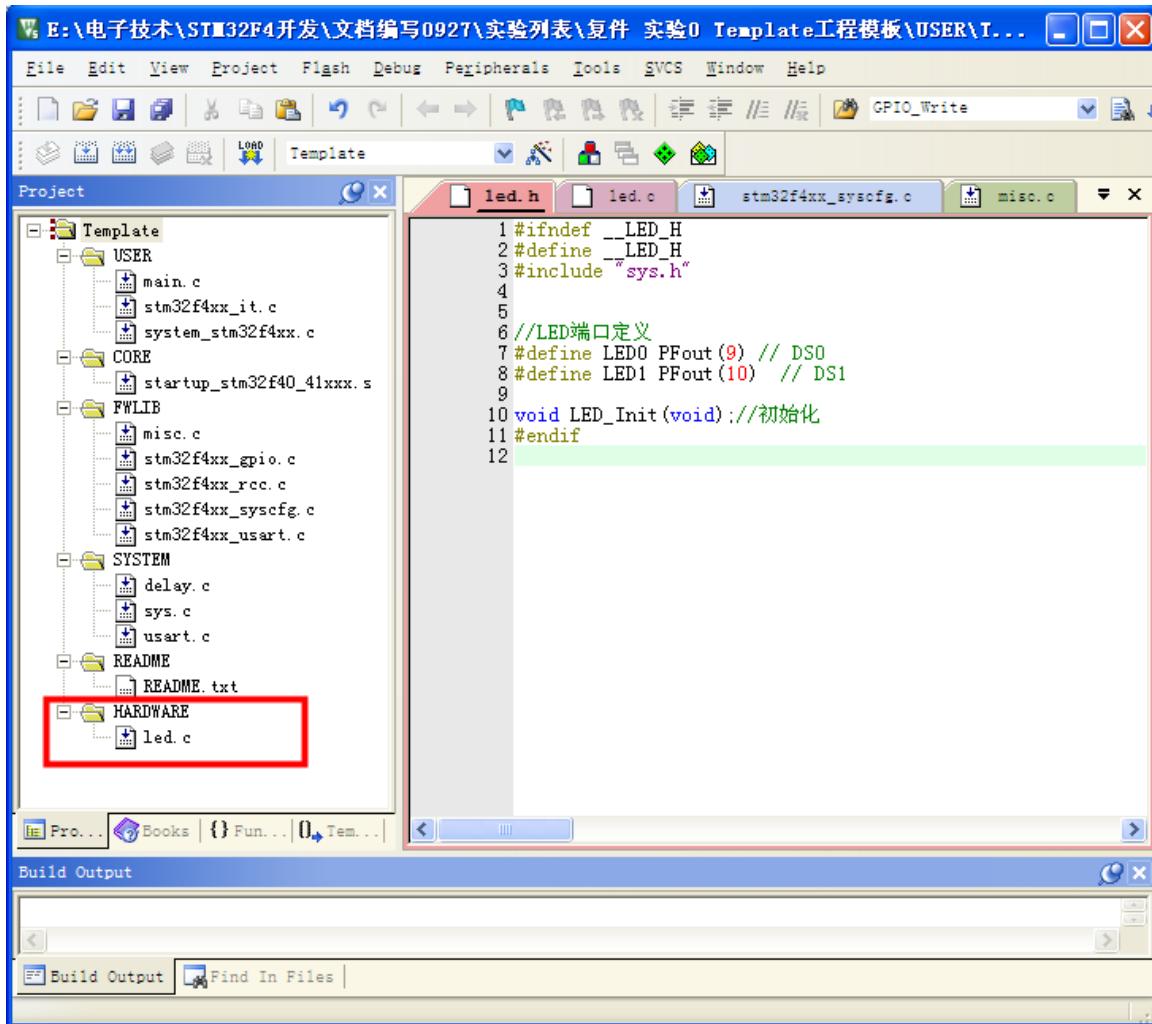


图 6.3.9 新增 HARDWARE 组

然后用之前介绍的方法（在 3.3.2 节介绍的）将 led.h 头文件的路径加入到工程里面，然后点击 OK 回到主界面。

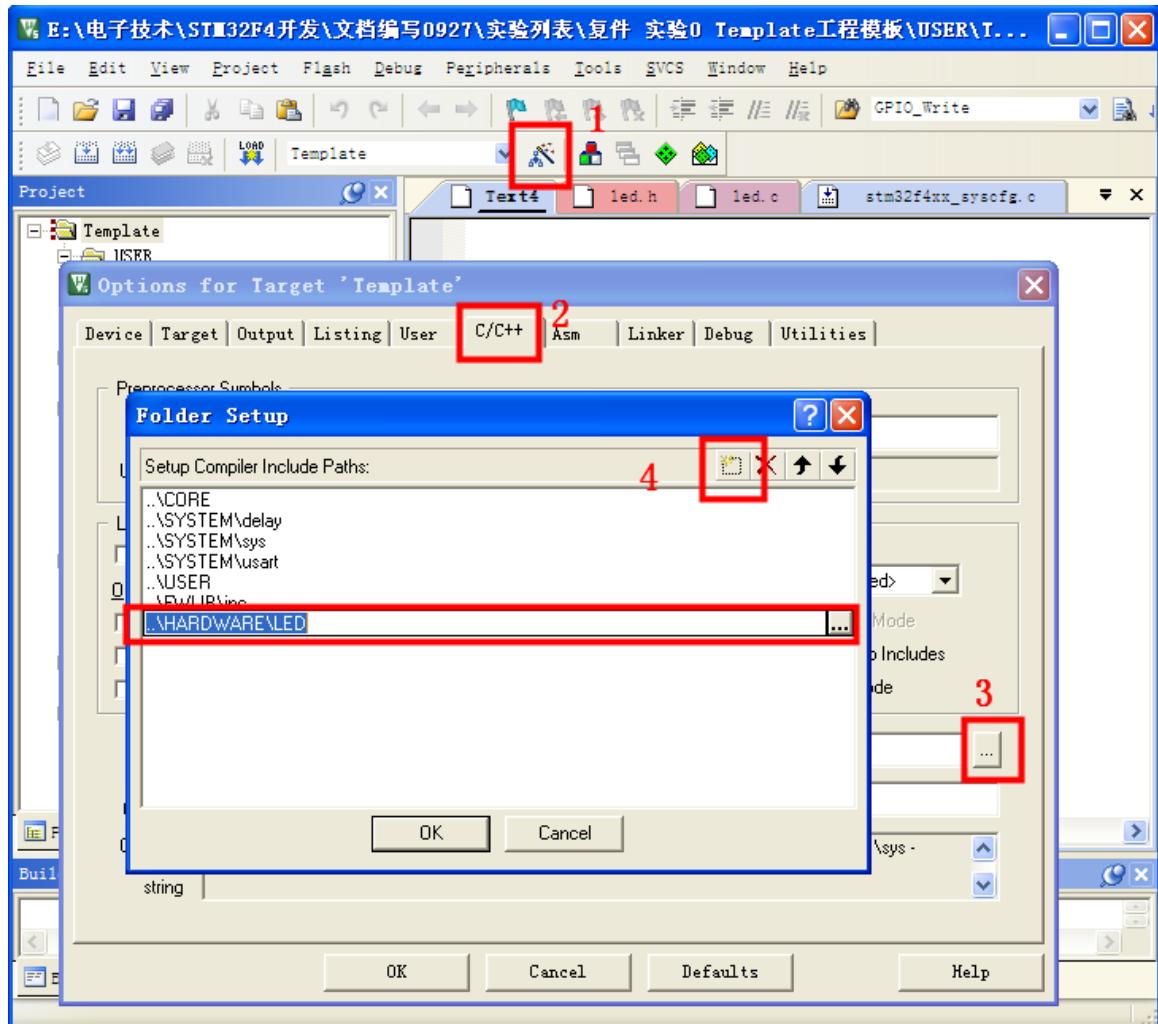


图 6.3.10 添加 LED 目录到 PATH

回到主界面后，在 main 函数里面编写如下代码：

```
#include "sys.h"
#include "delay.h"
#include "uart.h"
#include "led.h"

int main(void)
{
    delay_init(168);           // 初始化延时函数
    LED_Init();                // 初始化 LED 端口

    /**下面是通过直接操作库函数的方式实现 IO 控制**/
    while(1)
    {
        GPIO_ResetBits(GPIOF,GPIO_Pin_9); //LED0 对应引脚 GPIOF.9 拉低，亮 等同 LED0=0;
        GPIO_SetBits(GPIOF,GPIO_Pin_10); //LED1 对应引脚 GPIOF.10 拉高，灭 等同 LED1=1;
        delay_ms(500);                 //延时 500ms
    }
}
```

```

GPIO_SetBits(GPIOF,GPIO_Pin_9);      //LED0 对应引脚 GPIOF.0 拉高，灭 等同 LED0=1;
GPIO_ResetBits(GPIOF,GPIO_Pin_10); //LED1 对应引脚 GPIOF.10 拉低，亮 等同 LED1=0;
delay_ms(500);                  //延时 500ms
}

}

```

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，在固件库中，系统在启动的时候会调用 system_stm32f4xx.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 GPIOF.9 和 GPIOF.10 为输出。最后在死循环里面实现 LED0 和 LED1 交替闪烁，间隔为 500ms。

上面是通过库函数来实现的 IO 操作，我们也可以修改 main() 函数，直接通过位带操作达到同样的效果，大家不妨试试。位带操作的代码如下：

```

int main(void)
{
    delay_init(168);          //初始化延时函数
    LED_Init();               //初始化 LED 端口
    while(1)
    {
        LED0=0;                //LED0 亮
        LED1=1;                //LED1 灭
        delay_ms(500);
        LED0=1;                //LED0 灭
        LED1=0;                //LED1 亮
        delay_ms(500);
    }
}

```

当然我们也可以通过直接操作相关寄存器的方法来设置 IO，我们只需要将主函数修改为如下内容：

```

int main(void)
{
    delay_init(168);          //初始化延时函数
    LED_Init();               //初始化 LED 端口
    while(1)
    {
        GPIOF->BSRRH=GPIO_Pin_9;//LED0 亮
        GPIOF->BSRRL=GPIO_Pin_10;//LED1 灭
        delay_ms(500);
        GPIOF->BSRRL=GPIO_Pin_9;//LED0 灭
        GPIOF->BSRRH=GPIO_Pin_10;//LED1 亮
    }
}

```

```
    delay_ms(500);
}
}
```

将主函数替换为上面代码，然后重新执行，可以看到，结果跟库函数操作和位带操作一样的效果。大家可以对比一下。这个代码在我们跑马灯实验的 main.c 文件中有注释掉，大家可以替换试试。

然后按 ，编译工程，得到结果如图 6.3.11 所示：

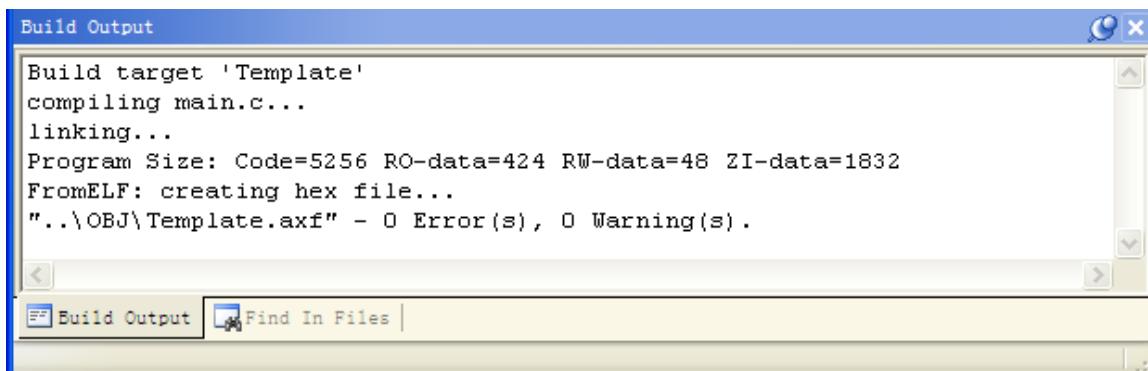


图 6.3.11 编译结果

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：5678 字节（5256+424），所用的 SRAM 大小为：1880 个字节（1832+48）。

这里我们解释一下，编译结果里面的几个数据的意义：

Code: 表示程序所占用 FLASH 的大小 (FLASH)。

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

RW-data: 即 Read Write-data，表示已被初始化的变量 (SRAM)

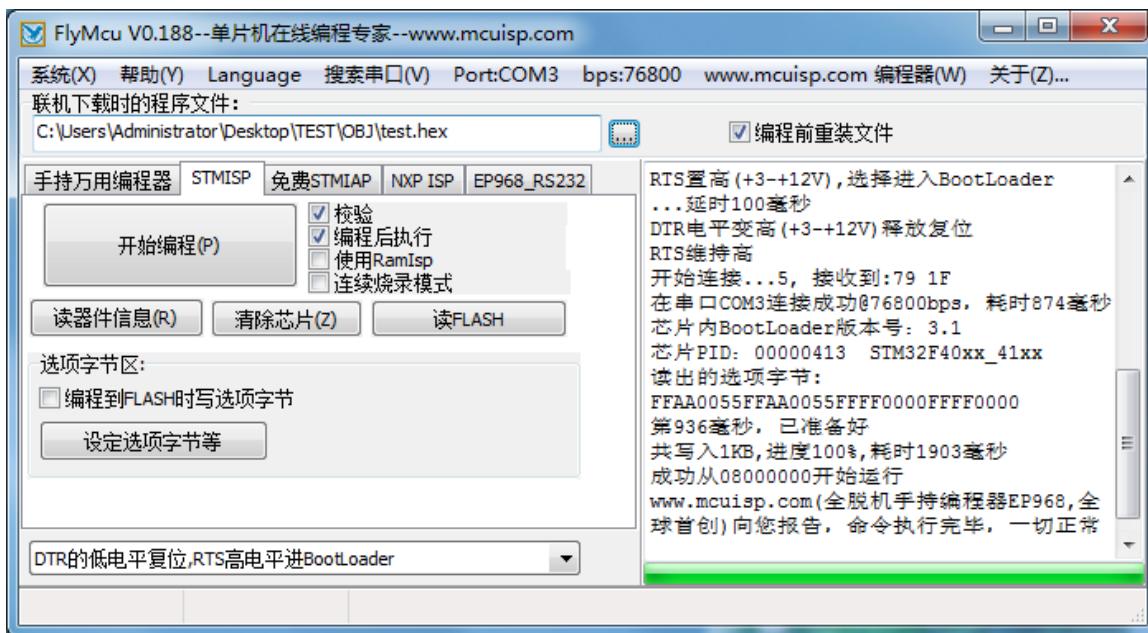
ZI-data: 即 Zero Init-data，表示未被初始化的变量(SRAM)

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，大家就可以下载验证了。如果有 JLINK，则可以用 jlink 进行在线调试（需要先下载代码），单步查看代码的运行，STM32F4 的在线调试方法介绍，参见：3.4.2 节。

6.4 下载验证

这里我们使用 flymcu 下载(也可以通过 JLINK 等仿真器下载，具体方法请参考 3.4.2 小节)，如图 6.4.1 所示：



6.4.1 利用 flymcu 下载代码

下载完之后，运行结果如图 6.4.2 所示，LED0 和 LED1 循环闪烁：

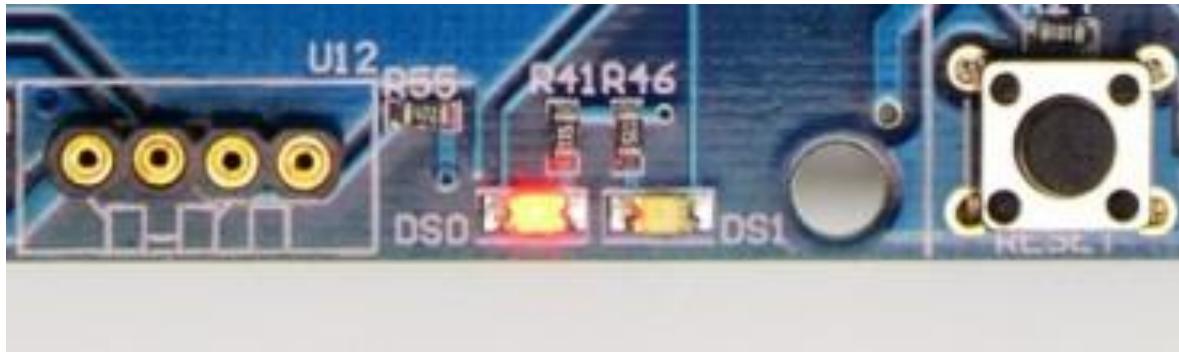


图 6.4.2 程序运行结果

至此，我们的第一章的学习就结束了，本章作为 STM32F4 的入门第一个例子，介绍了 STM32F4 的 IO 口的使用及注意事项，同时巩固了前面的学习，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 STM32F4 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 STM32F4 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 STM32F4 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 下载验证

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。

探索者 STM32F4 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 STM32F4 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：STM32F4 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，STM32F4 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 STM32F4 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 STM32F4 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 探索者 STM32F4 开发板上的蜂鸣器发出：“嘀”…“嘀”…的间隔声，进一步熟悉 STM32F4 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 STM32F4 的 PF8 上。如图 7.2.1 所示：

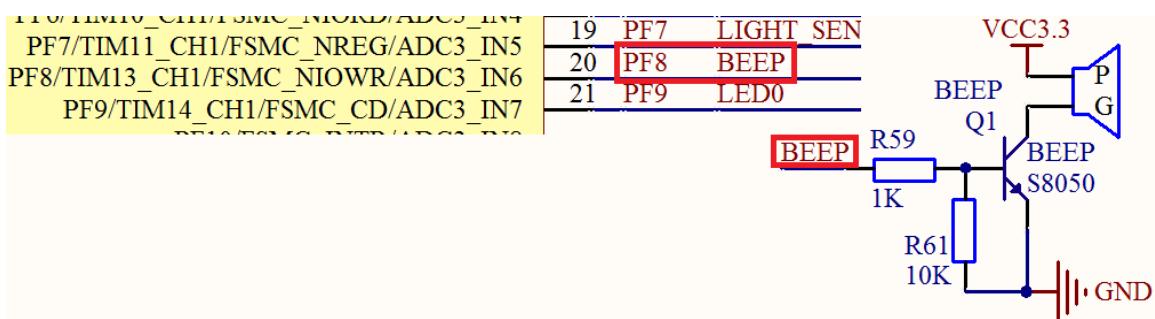


图 7.2.1 蜂鸣器与 STM32F4 连接原理图

图中我们用到一个 NPN 三极管 (S8050) 来驱动蜂鸣器，R61 主要用于防止蜂鸣器的误发声。当 PF.8 输出高电平的时候，蜂鸣器将发声，当 PF.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.uvproj 重命名为 BEEP.uvproj，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.uvproj 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 PF8 为输出口
//BEEP IO 初始化
void BEEP_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE); //使能 GPIOF 时钟
```

```
//初始化蜂鸣器对应引脚 GPIOF8
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;//下拉
GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化 GPIO

GPIO_ResetBits(GPIOF,GPIO_Pin_8); //蜂鸣器对应引脚 GPIOF8 拉低,
}
```

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 PORTF 的时钟, 然后调用 GPIO_Init 函数, 配置 PF8 为推挽输出。IO 口的初始化跟上一讲跑马灯实验非常类似, 这里我们就不做过多讲解。

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"
//LED 端口定义
#define BEEP_PFout(8) // 蜂鸣器控制 IO
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 PF8 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:

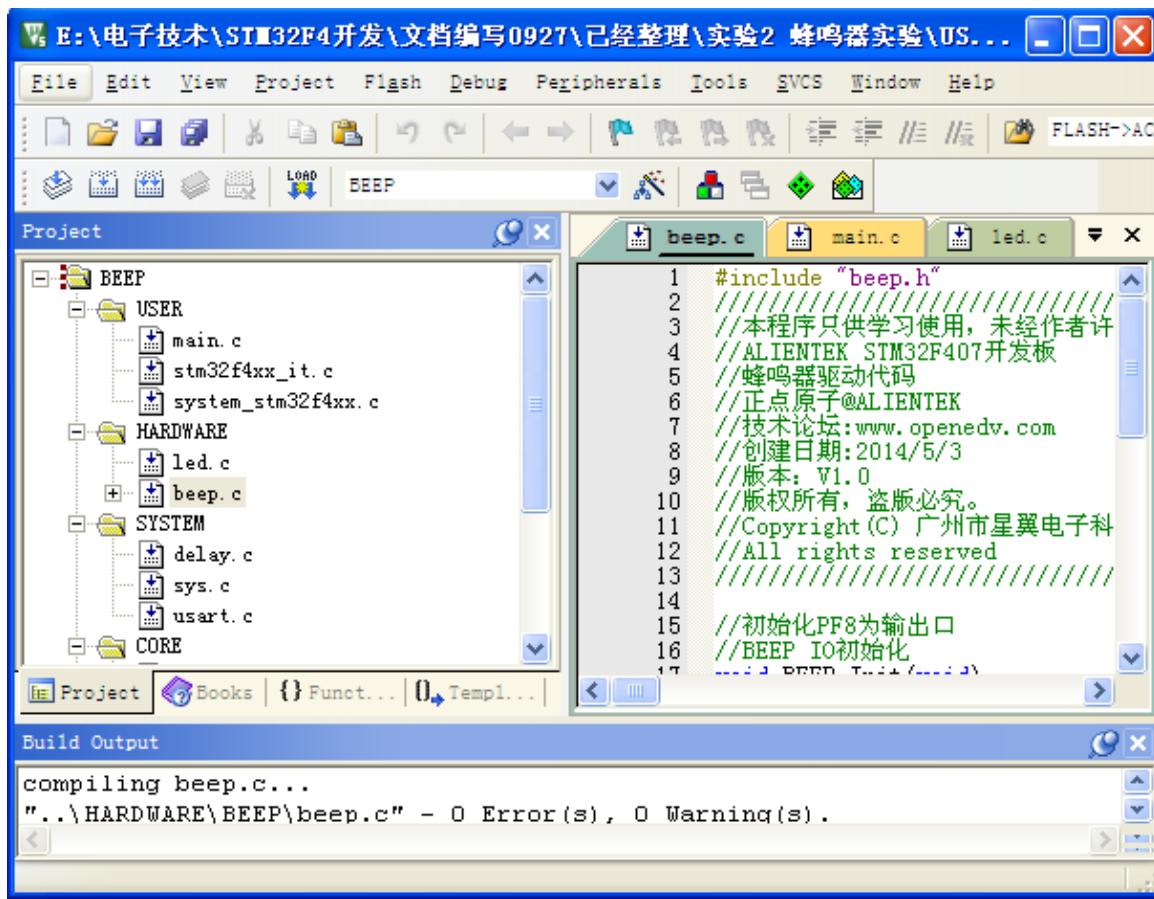


图 7.3.2 将 beep.c 加入 HARDWARE 组下

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（头文件包含路径，见 3.3.2 节或者上一讲的图 6.3.11）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 "uart.h"
#include "led.h"
#include "beep.h"
int main(void)
{
    delay_init(168);           // 初始化延时函数
    LED_Init();                // 初始化 LED 端口
    BEEP_Init();               // 初始化蜂鸣器端口

    while(1)
    {
        GPIO_ResetBits(GPIOF,GPIO_Pin_9); // DS0 拉低，亮 等同 LED0=0;
        GPIO_ResetBits(GPIOF,GPIO_Pin_8); // BEEP 引脚拉低， 等同 BEEP=0;
        delay_ms(300);                 // 延时 300ms
        GPIO_SetBits(GPIOF,GPIO_Pin_9); // DS0 拉高，灭 等同 LED0=1;
    }
}
```

```
GPIO_SetBits(GPIOF,GPIO_Pin_8); //BEEP 引脚拉高， 等同 BEEP=1;  
delay_ms(300); //延时 300ms  
}  
}
```

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。这段代码就是通过库函数 GPIO_ResetBits 和 GPIO_SetBits 两个函数实现前面 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。对于这两个函数的使用，在我们第六章跑马灯实验，我们已经做了非常详细的讲解，大家可以翻过去仔细学习。

然后按 ，编译工程，得到结果如图 7.3.3 所示：

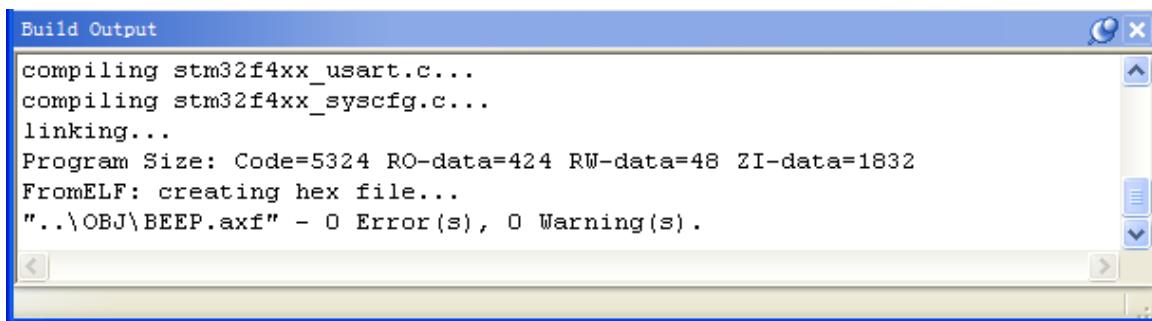


图 7.3.3 编译结果

可以看到没有错误，也没有警告。接下来，大家就可以下载验证了。如果有 JLINK，则可以用 jlink 进行在线调试（需要先下载代码），单步查看代码的运行，STM32F4 的在线调试方法介绍，参见：3.4.2 节。

7.4 下载验证

同样，我们通过 flymcu 下载代码，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 STM32F4 的入门第二个例子，进一步介绍了 STM32F4 的 IO 作为输出口的使用方法，同时巩固了前面知识的学习。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 STM32F4 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 STM32F4 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 STM32F4 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 STM32F4 IO 口简介
- 8.2 硬件设计
- 8.3 软件设计
- 8.4 下载验证

8.1 STM32F4 IO 口简介

STM32F4 的 IO 口在上两章已经有了比较详细的介绍，这里我们不再多说。STM32F4 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 探索者 STM32F4 开发板上载有的 4 个按钮（KEY_UP、KEY0、KEY1 和 KEY2），来控制板上的 2 个 LED（DS0 和 DS1）和蜂鸣器，其中 KEY_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 STM32F4 的连接在上两章都已经分别介绍了，在探索者 STM32F4 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、KEY_UP 连接在 PA0 上。如图 8.2.1 所示：

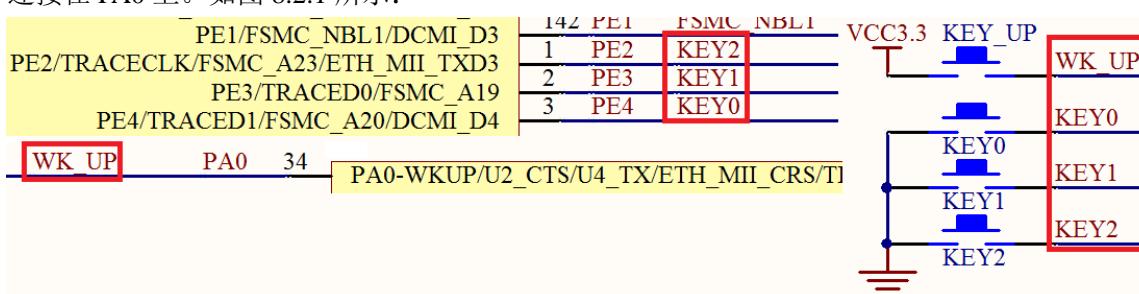


图 8.2.1 按键与 STM32F4 连接原理图

这里需要注意的是：KEY0、KEY1 和 KEY2 是低电平有效的，而 KEY_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 STM32F4 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。工程中添加相关文件的方法在我们前面两个实验已经讲解非常详细。

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，关键代码如下：

```
#include "key.h"
#include "delay.h"

//按键初始化函数
void KEY_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOE,
                           ENABLE);//使能 GPIOA,GPIOE 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4;
    //KEY0 KEY1 KEY2 对应引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//普通输入模式
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100M
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOE, &GPIO_InitStructure);//初始化 GPIOE2,3,4

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;//WK_UP 对应引脚 PA0
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN ;//下拉
    GPIO_Init(GPIOA, &GPIO_InitStructure);//初始化 GPIOA0

}

//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, WKUP 按下 WK_UP
//注意此函数有响应优先级,KEY0>KEY1>KEY2>WK_UP!!
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||WK_UP==1))
    {
        delay_ms(10);//去抖动
        key_up=0;
        if(KEY0==0)return 1;
        else if(KEY1==0)return 2;
        else if(KEY2==0)return 3;
        else if(WK_UP==1)return 4;
    }else if(KEY0==1&&KEY1==1&&KEY2==1&&WK_UP==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。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 KEY_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"

```

```

/*下面的方式是通过直接操作库函数方式读取 IO*/
#define KEY0      GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_4) //PE4
#define KEY1      GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3) //PE3
#define KEY2      GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2) //PE2
#define WK_UP    GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) //PA0

#define KEY0_PRES 1
#define KEY1_PRES 2
#define KEY2_PRES 3
#define WKUP_PRES 4

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) //PE4
#define KEY1      GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_3) //PE3
#define KEY2      GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_2) //PE2
#define WK_UP    GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0) //PA0

```

这里使用的是调用库函数来实现读取某个 IO 口的 1 个位的。同输出一样，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)    //P32
#define WK_UP     PAin(0)   //PA0
```

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 KEY0_PRES / KEY1_PRES/ KEY2_PRES/WKUP_PRES 等 4 个宏定义，分别对应开发板四个按键（KEY0/KEY1/KEY2/ KEY_UP）按键按下时 KEY_Scan 返回的值。通过宏定义的方式判断返回值，方便大家记忆和使用。

最后，我们看看 main.c 里面编写的主函数代码如下：

```
#include "sys.h"
#include "delay.h"
#include "uart.h"
#include "led.h"
#include "beep.h"
#include "key.h"

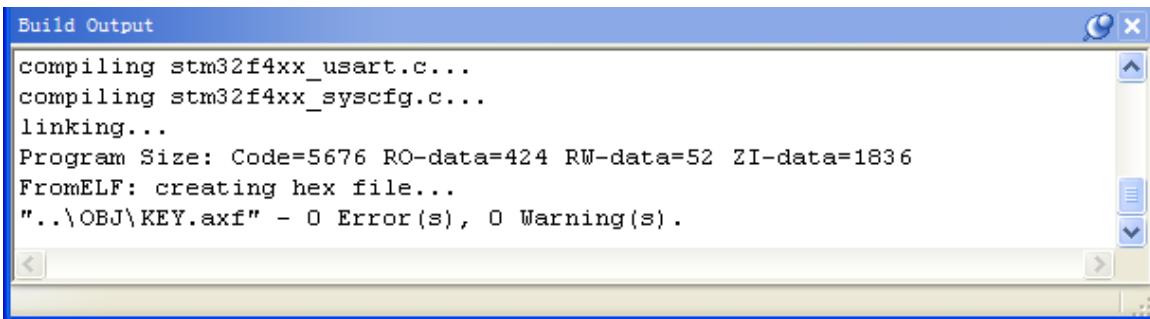
int main(void)
{
    u8 key;           //保存键值
    delay_init(168); //初始化延时函数
    LED_Init();       //初始化 LED 端口
    BEEP_Init();      //初始化蜂鸣器端口
    KEY_Init();       //初始化与按键连接的硬件接口
    LED0=0;           //先点亮红灯
    while(1)
    {
        key=KEY_Scan(0); //得到键值
        if(key)
        {
            switch(key)
            {
                case WKUP_PRES: //控制蜂鸣器
                    BEEP=!BEEP;
                    break;
                case KEY0_PRES: //控制 LED0 翻转
                    LED0=!LED0;
                    break;
                case KEY1_PRES: //控制 LED1 翻转
                    LED1=!LED1;
                    break;
                case KEY2_PRES: //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。

最后按 ，编译工程，得到结果如图 8.3.1 所示：



```
Build Output
compiling stm32f4xx_usart.c...
compiling stm32f4xx_syscfg.c...
linking...
Program Size: Code=5676 RO-data=424 RW-data=52 ZI-data=1836
FromELF: creating hex file...
"\..\OBJ\KEY.axf" - 0 Error(s), 0 Warning(s).
```

图 8.3.1 编译结果

可以看到没有错误，也没有警告。接下来，大家就可以下载验证了。如果有 JLINK，则可以用 jlink 进行在线调试（需要先下载代码），单步查看代码的运行，STM32F4 的在线调试方法介绍，参见：3.4.2 节。

8.4 下载验证

同样，我们还是通过 flymcu 下载代码，在下载完之后，我们可以按 KEY0、KEY1、KEY2 和 KEY_UP 来看看 DS0 和 DS1 以及蜂鸣器的变化，是否和我们预期的结果一致？

至此，我们的本章的学习就结束了。本章，作为 STM32F4 的入门第三个例子，介绍了 STM32F4 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 串口通信实验

前面三章介绍了 STM32F4 的 IO 口操作。这一章我们将学习 STM32F4 的串口，教大家如何使用 STM32F4 的串口来发送和接收数据。本章将实现如下功能：STM32F4 通过串口和上位机的对话，STM32F4 在收到上位机发过来的字符串后，原原本本地返回给上位机。本章分为如下几个小节：

- 9.1 STM32F4 串口简介
- 9.2 硬件设计
- 9.3 软件设计
- 9.4 下载验证

9.1 STM32F4 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。

STM32F4 的串口资源相当丰富的，功能也相当强劲。ALIENTEK 探索者 STM32F4 开发板所使用的 STM32F407ZGT6 最多可提供 6 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。探索者 STM32F4 开发板板载了 1 个 USB 串口和 2 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. 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，然后使能相应的外设时钟，同时要把 GPIO 模式设置为复用。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) 设置引脚复用器映射：调用 GPIO_PinAFConfig 函数。
- 3) GPIO 初始化设置：要设置模式为复用功能。
- 4) 串口参数初始化：设置波特率，字长，奇偶校验等参数。
- 5) 开启中断并且初始化 NVIC，使能中断（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：函数名格式为 USARTx_IRQHandler(x 对应串口号)。

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 `stm32f4xx_usart.h` 和 `stm32f4xx_usart.c` 文件中。

1) 串口时钟和 GPIO 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,ENABLE); //使能 USART1 时钟
```

GPIO 时钟使能，就非常简单，因为我们使用的是串口 1，串口 1 对应着芯片引脚 PA9,PA10，

所以这里我们只需要使能 GPIOA 时钟即可：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 GPIOA 时钟
```

2) 设置引脚复用器映射

引脚复用器映射配置方法在我们 4.4 小节讲解非常清晰，调用函数为：

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1); //PA9 复用为 USART1
```

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1); //PA10 复用为 USART1
```

因为串口使用到 PA9,PA10，所以我们要把 PA9 和 PA10 都映射到串口 1。所以这里我们要调用两次函数。

对于 GPIO_PinAFConfig 函数的第一个和第二个参数很好理解，就是设置对应的 IO 口，如果是 PA9 那么第一个参数是 GPIOA, 第二个参数就是 GPIO_PinSource9。第二个参数，实际我们不需要去记忆，只需要根据我们 4.7 小节讲解的快速组织代码技巧里面，去相应的配置文件找到外设对应的 AF 配置宏定义标识符即可，串口 1 为 GPIO_AF_USART1。

3) GPIO 端口模式设置：PA9 和 PA10 要设置为复用功能。

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //GPIOA9 与 GPIOA10
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
```

```
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
```

```
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
```

```
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
```

```
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA9, PA10
```

4) 串口参数初始化：设置波特率，字长，奇偶校验等参数

串口初始化是调用函数 USART_Init 来实现的，具体设置方法如下：

```
USART_InitStructureUSART_BaudRate = bound; //一般设置为 9600;
```

```
USART_InitStructureUSART_WordLength = USART_WordLength_8b; //字长为 8 位数据格式
```

```
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;
```

```
USART_InitStructureUSART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式
```

```
USART_Init(USART1, &USART_InitStructure); //初始化串口
```

5) 使能串口

使能串口调用函数 USART_Cmd 来实现，具体使能串口 1 方法如下：

```
USART_Cmd(USART1, ENABLE); //使能串口
```

6) 串口数据发送与接收。

STM32F4 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。

7) 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		

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_TC	((uint16_t)0x0626)
#define USART_IT_RXNE	((uint16_t)0x0525)
.....//(省略部分代码)	
#define USART_IT_NE	((uint16_t)0x0260)
#define USART_IT_FE	((uint16_t)0x0160)

8) 开启中断并且初始化 NVIC，使能相应中断

这一步如果我们要开启串口中断才需要配置 NVIC 中断优先级分组。通过调用函数 NVIC_Init 来设置。

```
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=3;//抢占优先级 3
NVIC_InitStructure.NVIC_IRQChannelSubPriority =3; //响应优先级 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、
```

同时，我们还需要使能相应中断，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);
```

这里还要特别提醒，因为我们实验开启了串口中断，所以我们在系统初始化的时候需要先设置系统的中断优先级分组，我们是在我们 main 函数开头设置的，代码如下：

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
```

我们设置分组为 2，也就是 2 位抢占优先级，2 位响应优先级。

9) 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。

10) 中断服务函数

串口 1 中断服务函数为：

```
void USART1_IRQHandler(void);
```

当发生中断的时候，程序就会执行中断服务函数。然后我们在中断服务函数中编写我们相应的逻辑代码即可。

通过以上一些寄存器的操作外加一下 IO 口的配置，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《STM32F4XX 中文参考手册》第 676 页至 720 页，通用同步异步收发器这一章节。

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：

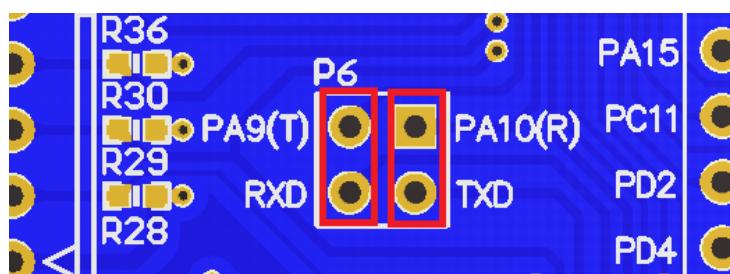


图 9.2.1 硬件连接图示意图

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：

```
void uart_init(u32 bound)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    //GPIOA 和 USART1 时钟使能①
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE); //使能 USART1 时钟

    //USART_DeInit(USART1); //复位串口 1 ②
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1); //PA9 复用为 USART1
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1); //PA10 复用为 USART1

    //USART1_TX PA.9 PA.10 ③
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10; //GPIOA9 与 GPIOA10
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA9, PA10

    //USART 初始化设置 ④
    USART_InitStructureUSART_BaudRate = bound; //一般设置为 9600;
    USART_InitStructureUSART_WordLength = USART_WordLength_8b; //字长为 8 位数据格式
    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;
    USART_InitStructureUSART_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 //NVIC 设置，使能中断 ⑤
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //开启中断
    //Usart1 NVIC 配置
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; //抢占优先级 2
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2; //响应优先级 2
#endif
}
```

```

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、
#endif
}
USART_Cmd(USART1, ENABLE); //使能串口 ⑥

```

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 设置引脚复用器映射
- ③ GPIO 端口初始化设置
- ④ 串口参数初始化
- ⑤ 初始化 NVIC 并且开启中断
- ⑥ 使能串口

这里需要注意一点，因为我们使用到了串口的中断接收，必须在 `uart.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。

串口 1 的中断服务函数 `USART1_IRQHandler`，在 5.3.3 已经有详细介绍了，这里我们就不在介绍了。

介绍完了这两个函数，我们回到 `main.c`，对于 `main.c` 前面引入的头文件为了篇幅考虑，我们后面的实验不再列出，详情请参考我们实验代码即可。主函数代码如下：

```

int main(void)
{
    u8 t,len; u16 times=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为 115200
    LED_Init(); //初始化与 LED 连接的硬件接口
    LED0=0; //先点亮红灯
    while(1)
    {
        if(USART_RX_STA&0x8000)
        {
            len=USART_RX_STA&0x3fff;//得到此次接收到的数据长度
            printf("\r\n 您发送的消息为:\r\n");
            for(t=0;t<len;t++)
            {
                USART1->DR=USART_RX_BUF[t];
                while((USART1->SR&0X40)==0); //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\nALIENTEK 探索者 STM32F407 开发板 串口实验\r\n");
            printf("正点原子@ALIENTEK\r\n\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，开头部分我们先调用 NVIC_PriorityGroupConfig 函数设置系统的中断优先级分组。然后调用 uart_init 函数，设置波特率为 115200。接下来我们重点看下以下两句：

```

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 9.3.1 所示：

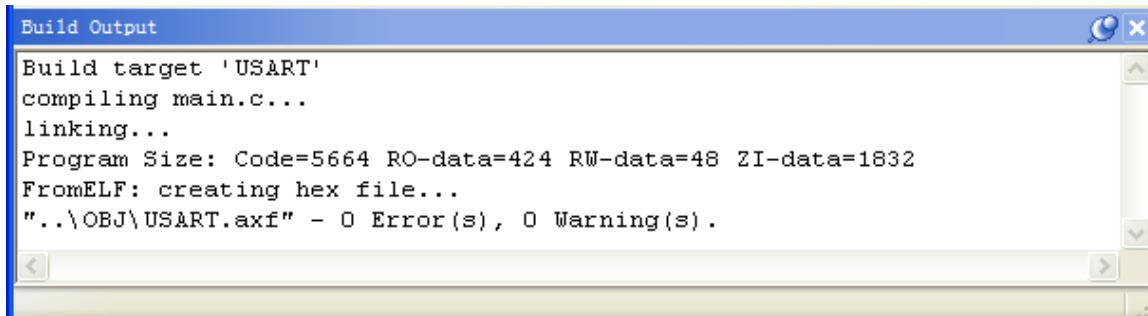


图 9.3.1 编译结果

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

我们把程序下载到探索者 STM32F4 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，串口调试助手，我们用 XCOM V2.0，该软件在光盘有提供，且无需安装，直接可以运行，但是需要你的电脑安装有.NET Framework 4.0(WIN7 直接自带了)或以上版本的环境才可以，该软件的详细介绍请看：<http://www.openedv.com/posts/list/22994.htm> 这个帖子。

接着我们打开 XCOM V2.0，设置串口为开发板的 USB 转串口 (CH340 虚拟串口，得根据你自己的电脑选择，我的电脑是 COM3，另外，请注意：**波特率是 115200**)，可以看到如图 9.4.1 所示信息：

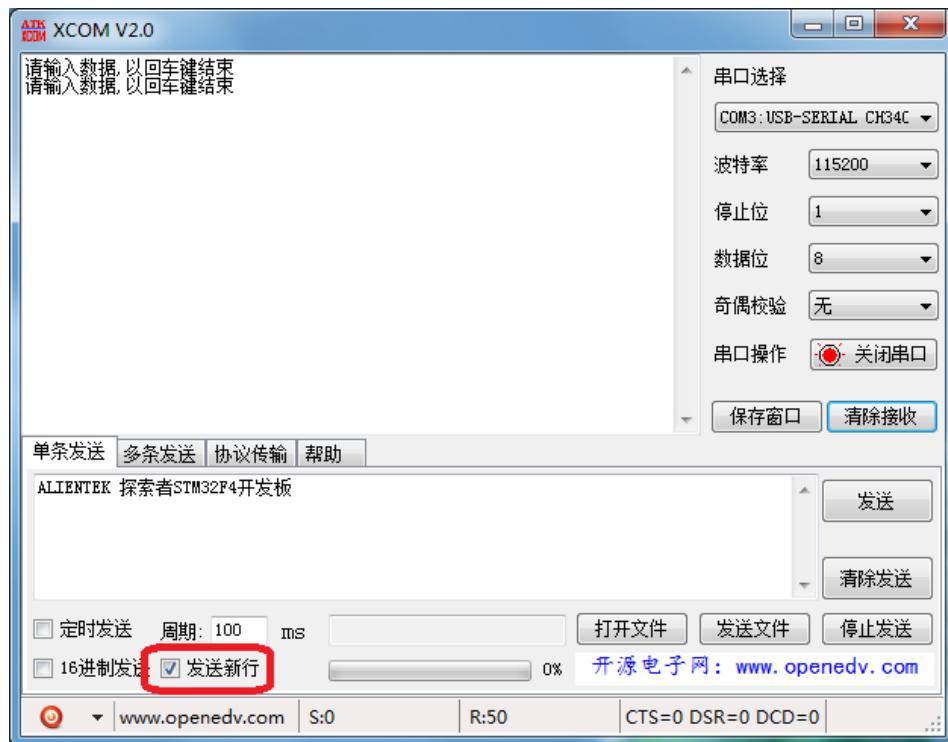


图 9.4.1 串口调试助手收到的信息

从图 9.4.1 可以看出, STM32F4 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 XCOM 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, XCOM 都会自动多发一个回车(0X0D+0X0A)。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:

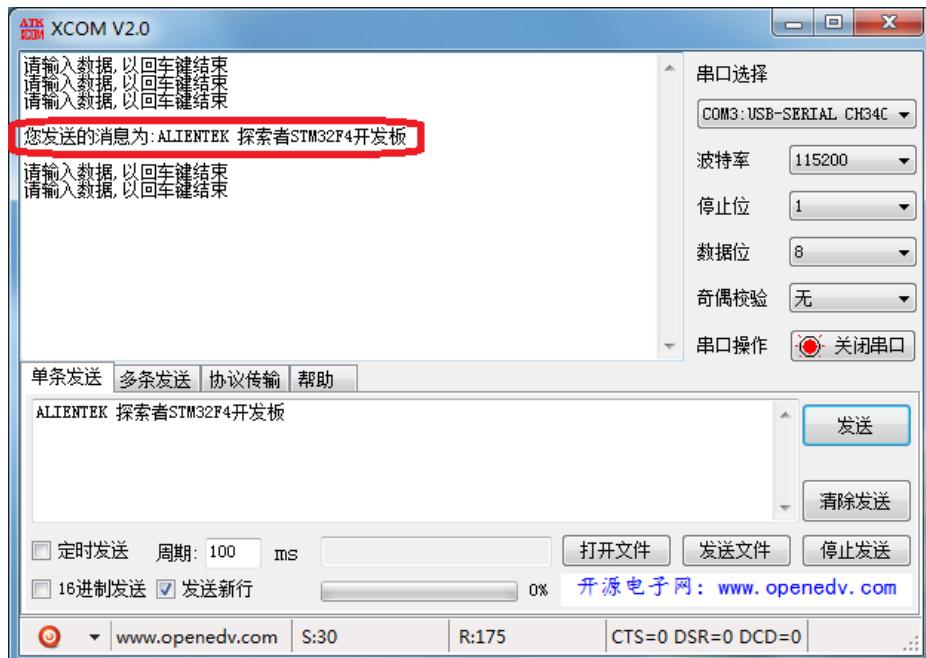


图 9.4.2 发送数据后收到的数据

可以看到, 我们发送的消息被发送回来了 (图中圈圈内)。大家可以试试, 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 STM32F4 的外部输入中断。在前面几章的学习中，我们掌握了 STM32F4 的 IO 口最基本的操作。本章我们将介绍如何将 STM32F4 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 STM32F4 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 STM32F4 外部中断简介

STM32F4 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 STM32F4 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 `stm32f4xx_exti.h` 和 `stm32f4xx_exti.c` 文件中。

这里我们首先介绍 STM32F4 IO 口中断的一些基础概念。STM32F4 的每个 IO 都可以作为外部中断的中断输入口，这点也是 STM32F4 的强大之处。STM32F407 的中断控制器支持 22 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。STM32F407 的 22 个外部中断为：

- EXTI 线 0~15：对应外部 IO 口的输入中断。
- EXTI 线 16：连接到 PVD 输出。
- EXTI 线 17：连接到 RTC 闹钟事件。
- EXTI 线 18：连接到 USB OTG FS 唤醒事件。
- EXTI 线 19：连接到以太网唤醒事件。
- EXTI 线 20：连接到 USB OTG HS(在 FS 中配置)唤醒事件。
- EXTI 线 21：连接到 RTC 入侵和时间戳事件。
- EXTI 线 22：连接到 RTC 唤醒事件。

从上面可以看出，STM32F4 供 IO 口使用的中断线只有 16 个，但是 STM32F4 的 IO 口却远远不止 16 个，那么 STM32F4 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,H,I)分别对应中断线 0~15。这样每个中断线对应了最多 9 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,GPIOH.0,GPIOI.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：

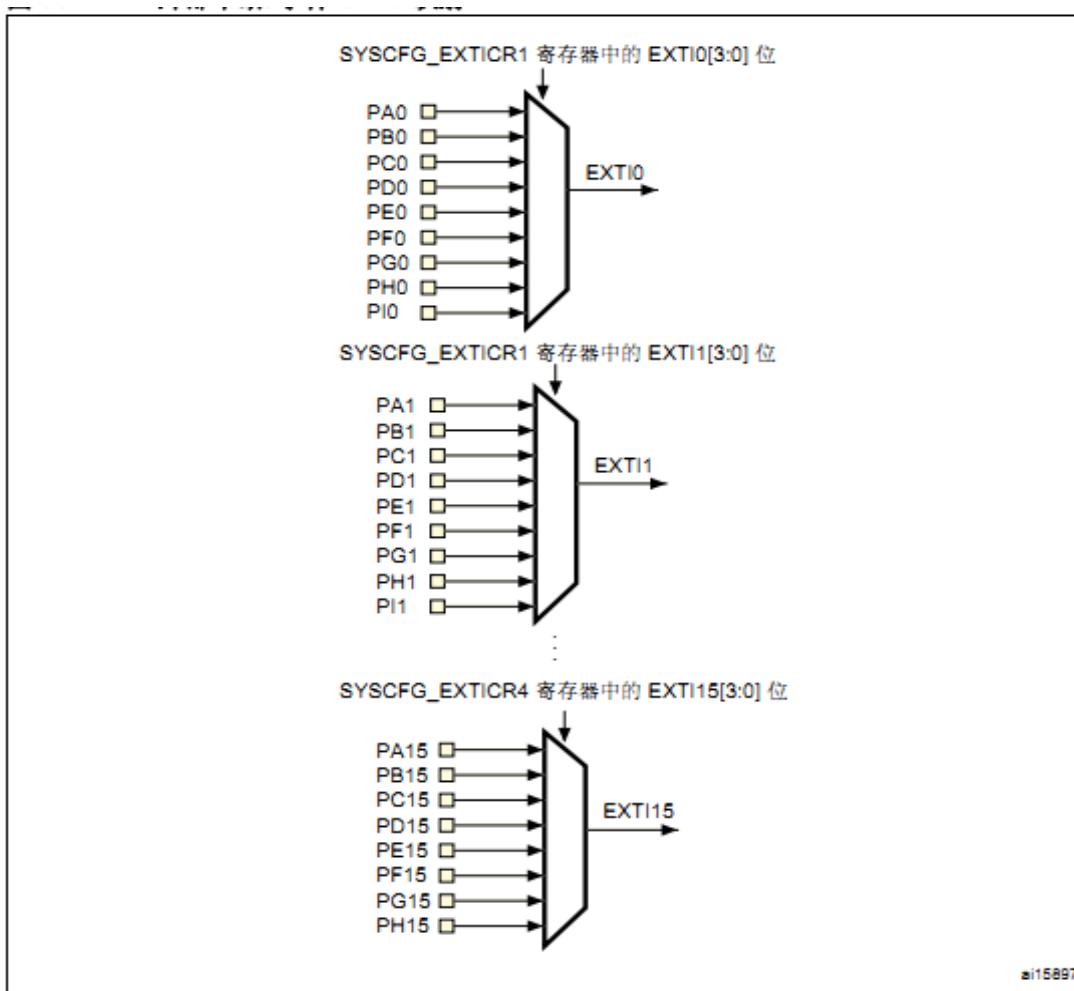


图 10.1.1 GPIO 和中断线的映射关系图

接下来我们讲解使用库函数配置外部中断的步骤。

1) 使能 IO 口时钟，初始化 IO 口为输入

首先，我们要使用 IO 口作为中断输入，所以我们要使能相应的 IO 口时钟，以及初始化相应的 IO 口为输入模式，具体的使用方法跟我们按键实验是一致的。这里就不做过多讲解。

2) 开启 SYSCFG 时钟，设置 IO 口与中断线的映射关系。

接下来，我们要配置 GPIO 与中断线的映射关系，那么我们首先需要打开 SYSCFG 时钟。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE); //使能 SYSCFG 时钟
```

这里大家一定要注意，只要我们使用到外部中断，就必须打开 SYSCFG 时钟。

接下来，我们配置 GPIO 与中断线的映射关系。在库函数中，配置 GPIO 与中断线的映射关系的函数 SYSCFG_EXTILineConfig ()来实现的：

```
void SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex);
```

该函数将 GPIO 端口与中断线映射起来，使用范例是：

```
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
```

将中断线 0 与 GPIOA 映射起来，那么很显然是 GPIOA.0 与 EXTI1 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。

3) 初始化线上中断，设置触发条件等。

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);      //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，这里就不再讲解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，对于我们的外部中断，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。

4) 配置中断分组 (NVIC)，并使能中断。

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;        //响应优先级 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。

5) 编写中断服务函数。

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，STM32F4 的 IO 口外部中断函数只有 7 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 EXTI3_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解。下面我们再总结一下使用 IO 口外部中断的一般步骤：

- 1) 使能 IO 口时钟，初始化 IO 口为输入。
- 2) 使能 SYSCFG 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 KEY_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 5 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 stm32f4xx_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；

extic.c 代码如下：

```
//外部中断 0 服务程序
void EXTI0_IRQHandler(void)
{
    delay_ms(10); //消抖
    if(WK_UP==1)
    {
        BEEP=!BEEP; //蜂鸣器翻转
    }
    EXTI_ClearITPendingBit(EXTI_Line0); //清除 LINE0 上的中断标志位
}

//外部中断 2 服务程序
void EXTI2_IRQHandler(void)
{
    delay_ms(10); //消抖
    if(KEY2==0)
    {
        LED0=!LED0;
    }
    EXTI_ClearITPendingBit(EXTI_Line2); //清除 LINE2 上的中断标志位
}

//外部中断 3 服务程序
void EXTI3_IRQHandler(void)
{
    delay_ms(10); //消抖
    if(KEY1==0)
    {
        LED1=!LED1;
    }
    EXTI_ClearITPendingBit(EXTI_Line3); //清除 LINE3 上的中断标志位
}

//外部中断 4 服务程序
void EXTI4_IRQHandler(void)
{
    delay_ms(10); //消抖
    if(KEY0==0)
    {
        LED0=!LED0;
        LED1=!LED1;
    }
    EXTI_ClearITPendingBit(EXTI_Line4); //清除 LINE4 上的中断标志位
}

//外部中断初始化程序
```

```
//初始化 PE2~4,PA0 为中断输入.
void EXTIx_Init(void)
{
    NVIC_InitTypeDef      NVIC_InitStructure;
    EXTI_InitTypeDef     EXTI_InitStructure;

    KEY_Init(); //按键对应的 IO 口初始化
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE); //使能 SYSCFG 时钟

    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource2); //PE2 连接线 2
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource3); //PE3 连接线 3
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource4); //PE4 连接线 4
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0); //PA0 连接线 0

    /* 配置 EXTI_Line0 */
    EXTI_InitStructure.EXTI_Line = EXTI_Line0; //LINE0
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //中断事件
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; //上升沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE; //使能 LINE0
    EXTI_Init(&EXTI_InitStructure); /


    /* 配置 EXTI_Line2,3,4 */
    EXTI_InitStructure.EXTI_Line = EXTI_Line2 | EXTI_Line3 | EXTI_Line4;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //中断事件
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE; //中断线使能
    EXTI_Init(&EXTI_InitStructure); //配置

    NVIC_InitStructure.NVIC IRQChannel = EXTI0 IRQn; //外部中断 0
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x00; //抢占优先级 0
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x02; //响应优先级 2
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; //使能外部中断通道
    NVIC_Init(&NVIC_InitStructure); //配置 NVIC

    NVIC_InitStructure.NVIC IRQChannel = EXTI2 IRQn; //外部中断 2
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x03; //抢占优先级 3
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x02; //响应优先级 2
    NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; //使能外部中断通道
    NVIC_Init(&NVIC_InitStructure); //配置 NVIC

    NVIC_InitStructure.NVIC IRQChannel = EXTI3 IRQn; //外部中断 3
    NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x02; //抢占优先级 2
    NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x02; //响应优先级 2
```

```

NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;//使能外部中断通道
NVIC_Init(&NVIC_InitStructure);//配置 NVIC

NVIC_InitStructure.NVIC IRQChannel = EXTI4 IRQn;//外部中断 4
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x01;//抢占优先级 1
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x02;//响应优先级 2
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;//使能外部中断通道
NVIC_Init(&NVIC_InitStructure);//配置 NVIC
}

```

exti.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 KEY_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；下面我们分别介绍这几个函数。

首先是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，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 SYSCFG 时钟。接着调用函数 SYSCFG_EXTILineConfig 配置中断线和 GPIO 的映射关系，然后初始化中断线和配置中断优先级。需要说明的是因为我们的 KEY_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 KEY_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，把按键的抢占优先级设置成一样，而响应优先级不同，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 KEY_UP 的中断服务函数 void EXTI0_IRQHandler(void)，该函数代码比较简单，先延时 10ms 以消抖，再检测 KEY_UP 是否还是为高电平，如果是，则执行此次操作（翻转蜂鸣器控制信号），如果不是，则直接跳过，在最后有一句 EXTI_ClearITPendingBit(EXTI_Line0);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 KEY2 的中断服务函数和 KEY_UP 按键的十分相似，我们就不逐个介绍了。

这里向大家重申一下，STM32F4 的外部中断 0~4 都有单独的中断服务函数，但是从 5 开始，他们就没有单独的服务函数了，而是多个中断共用一个服务函数，比如外部中断 5~9 的中断服务函数为：void EXTI9_5_IRQHandler(void)，类似的，void EXTI15_10_IRQHandler(void)就是外部中断 10~15 的中断服务函数。另外，STM32F4 所有中断服务函数的名字，都已经在 startup_stm32f40_41xx.s 里面定义好了，如果有不知道的，去这个文件里面找就可以了。

exti.h 头文件里面主要是一个函数申明，比较简单，这里不做过多讲解。

接下来我们看看主函数，main 函数代码如下：

```

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //串口初始化
    LED_Init(); //初始化 LED 端口
    BEEP_Init(); //初始化蜂鸣器端口
}

```

```

EXTIX_Init(); //初始化外部中断输入
LED0=0; //先点亮红灯
while(1)
{
    printf("OK\r\n"); //打印 OK 提示程序运行
    delay_ms(1000); //每隔 1s 打印一次
}
}

```

该部分代码很简单，先设置系统优先级分组，延时函数以及串口等外设。然后在初始化完中断后，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探索者 STM32F4 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 KEY_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 STM32F4 的独立看门狗(以下简称 IWDG)。STM32F4 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 KEY_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 STM32F4 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 STM32F4 独立看门狗简介

STM32F4 的独立看门狗由内部专门的 32Khz 低速时钟 (LSI) 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 32Khz，而是在 15~47Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 32Khz 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。

独立看门狗有几个寄存器与我们这节相关，我们分别介绍这几个寄存器，首先是关键字寄存器 IWDG_KR，该寄存器的各位描述如图 11.1.1 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
Reserved															KEY[15:0]																
															W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	

位 31:16 保留，必须保持复位值。

位 15:0 **KEY[15:0]**: 键值 (Key value) (只写位，读为 0000h)

必须每隔一段时间便通过软件对这些位写入键值 AAAAh，否则当计数器计数到 0 时，看门狗会产生复位。

写入键值 5555h 可使能对 IWDG_PR 和 IWDG_RLR 寄存器的访问

写入键值 CCCCh 可启动看门狗（选中硬件看门狗选项的情况除外）

图 11.1.1 IWDG_KR 寄存器各位描述

在关键字寄存器(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 0x0000 时，会产生一个复位信号(IWDG_RESET)。无论何时，只要关键字寄存器 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。

接下来，我们介绍预分频寄存器 (IWDG_PR)，该寄存器用来设置看门狗时钟的分频系数，最低为 4，最高位 256，该寄存器是一个 32 位的寄存器，但是我们只用了最低 3 位，其他都是保留位。预分频寄存器各位定义如图 11.1.2 所示：

位 31:3 保留，必须保持复位值。

位 2:0 PR[2:0]: 预分频器 (Prescaler divider)

这些位受写访问保护，通过软件设置这些位来选择计数器时钟的预分频因子。若要更改预分频器的分频系数，IWDG_SR 的 PVU 位必须为 0。

000: 4 分频	100: 64 分频
001: 8 分频	101: 128 分频
010: 16 分频	110: 256 分频
011: 32 分频	111: 256 分频

注意：读取该寄存器会返回 VDD 电压域的预分频器值。如果正在对该寄存器执行写操作，则读取的值可能不是最新的/有效的。因此，只有在 IWDG_SR 寄存器中的 PVU 位为 0 时，从寄存器读取的值才有效。

图 11.1.2 IWDG PR 寄存器各位描述

在介绍完 IWDG_PR 之后，我们介绍一下重装载寄存器 IWDG_RLR。该寄存器用来保存重装载到计数器中的值。该寄存器也是一个 32 位寄存器，但是只有低 12 位是有效的，该寄存器的各位描述如图 11.1.3 所示：

位 31:12 保留，必须保持复位值。

位 11:0 RL[11:0]: 看门狗计数器重载值 (Watchdog counter reload value)

这些位受写访问保护，请参考之前介绍。这个值由软件设置，每次对 IWDR_KR 寄存器写入值 AAAAh 时，这个值就会重装载到看门狗计数器中。之后，看门狗计数器便从该装载的值开始递减计数。超时周期由该值和时钟预分频器共同决定。

若要更改重载值，IWDG_SR 中的 RVU 位必须为 0。

注意：读取该寄存器会返回 VDD 电压域的重载值。如果正在对该寄存器执行写操作，则读取的值可能不是最新的/有效的。因此，只有在 IWDG_SR 寄存器中的 RVU 位为 0 时，从寄存器读取的值才有效。

图 11.1.3 IWDG_RLR 重装载寄存器各位描述

只要对以上三个寄存器进行相应的设置，我们就可以启动 STM32F4 的独立看门狗。独立看门狗相关的库函数操作函数在文件 `stm32f4xx_iwdg.c` 和对应的头文件 `stm32f4xx_iwdg.h` 中。接下来我们讲解一下通过库函数来配置独立看门狗的步骤：

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

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 IW
```

设置看门狗的重装裁值的函数是：

设置首尾列的空表视图的函数是：

设置好看门狗的分频系数 和重挂载值就可以知道看门狗的

设置好看门狗的分频系数 `per` 和重表软值就可以知道看门狗的看狗时间（也就是看门狗溢出时间），该时间的计算方式为：

Tout=((4×Z'prer) ×flr)/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（`TWDG_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 STM32F4 的看门狗。注意 IWDG 在一旦启用, 就不能再被关闭! 想要关闭, 只能重启, 并且重启之后不能打开 IWDG, 否则问题依旧, 所以在这里提醒大家, 如果不用 IWDG 的话, 就不要去打开它, 免得麻烦。

通过上面 4 个步骤, 我们就可以启动 STM32F4 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 STM32F4 的独立看门狗。

在配置看门狗后, DS0 将常亮, 如果 KEY_UP 按键按下, 就喂狗, 只要 KEY_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) KEY_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 STM32F4 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 KEY_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文件 iwdg.c, 同时引入了头文件 iwdg.h。同样的道理, 我们要加入固件库看门狗支持文件 stm32f4xx_iwdg.h 和 stm32f4xx_iwdg.c 文件。

iwdg.c 里面的代码如下:

```
#include "iwdg.h"
//初始化独立看门狗
//prer:分频数:0~7(只有低 3 位有效!)    rlr:自动重装载值,0~0xFFFF.
//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)/32 (ms).
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable); //取消寄存器写保护
    IWDG_SetPrescaler(prer); //设置 IWDG 分频系数
    IWDG_SetReload(rlr); //设置 IWDG 装载值
    IWDG_ReloadCounter(); //reload
    IWDG_Enable(); //使能看门狗
}
//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来初始化独立看门狗的。该函数有 2 个参数，分别用来设置预分频数与重装载寄存器的值的。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 的喂狗只需要向关键字寄存器写入 0XAAAA 即可，也就是调用库函数 IWDG_ReloadCounter()，所以这个函数也是很简单的。

iwdg.h 内容比较简单，主要是一些函数申明，这里我们忽略不讲解。

接下来我们看看主函数，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 KEY_UP 有按键，则喂狗，否则等待 IWDG 复位的到来。该部分代码如下：

```

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    LED_Init(); //初始化 LED 端口
    KEY_Init(); //初始化按键
    delay_ms(100); //延时 100ms
    IWDG_Init(4,500); //与分频数为 64,重载值为 500,溢出时间为 1s
    LED0=0; //先点亮红灯
    while(1)
    {
        if(KEY_Scan(0)==WKUP_PRES)//如果 WK_UP 按下,则喂狗
        {
            IWDG_Feed(); //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似的方式处理），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。

11.4 下载验证

在编译成功之后，我们就可以下载代码到探索者 STM32F4 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 KEY_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 STM32F4 的另外一个看门狗，窗口看门狗（以下简称 WWDG）。在本章中，我们将使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 STM32F4 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 STM32F4 窗口看门狗简介

窗口看门狗 (WWDG) 通常被用来监测由外部干扰或不可预见的逻辑条件造成应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位(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：

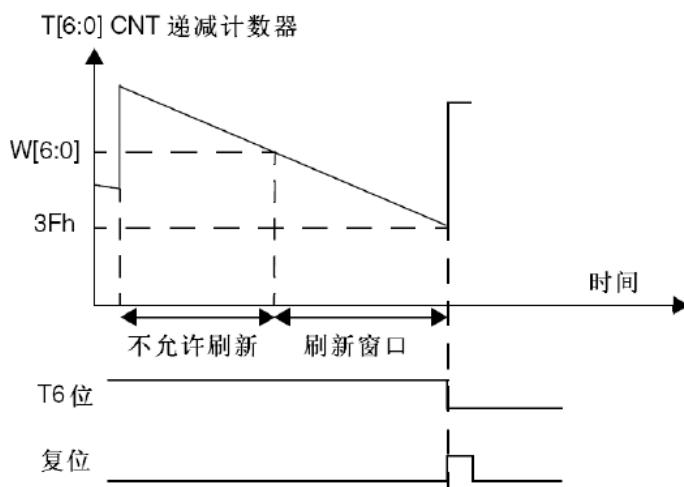


图 12.1.1 窗口看门狗工作示意图

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：WWDG 超时时间（单位为 ms）

Fpclk1：APB1 的时钟频率（单位为 Khz）

WDGTB：WWDG 的预分频系数

T[5:0]：窗口看门狗的计数器低 6 位

根据上面的公式，假设 $F_{PCLK1}=42MHz$ ，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 (μs) $T[5:0] = 0x00$	最大超时 (ms) $T[5:0] = 0x3F$
0	97.52	6.24
1	195.05	12.48
2	390.10	24.97
3	780.19	49.93

表 12.1.1 42M 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：

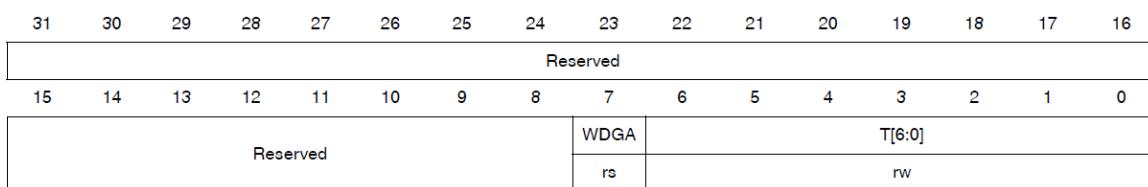
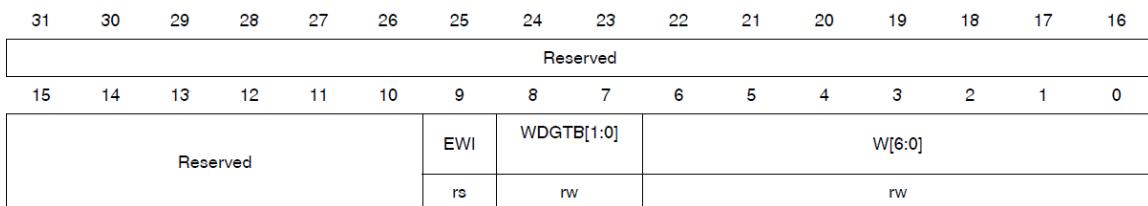


图 12.1.2 WWDG_CR 寄存器各位描述

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 (4096×2^8 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：



位 31:10 保留，必须保持复位值。

位 9 EWI：提前唤醒中断 (Early wakeup interrupt)

置 1 后，只要计数器值达到 0x40 就会产生中断。此中断只有在复位后才由硬件清零。

位 8:7 WDGTB[1:0]：定时器时基 (Timer base)

可按如下方式修改预分频器的时基：

00: CK 计数器时钟 (PCLK1 div 4096) 分频器 1

01: CK 计数器时钟 (PCLK1 div 4096) 分频器 2

10:CK 计数器时钟 (PCLK1 div 4096) 分频器 4

11:CK 计数器时钟 (PCLK1 div 4096) 分频器 8

位 6:0 W[6:0]: 7 位窗口值 (7-bit window value)

这些位包含用于与递减计数器进行比较的窗口值。

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 42M 且 WDGTR 为 0 的条件下，该时间为 97.52us) 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 STM32F4 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 `stm32f4xx_wwdg.c` 文件和头文件 `stm32f4xx_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 32Khz 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。

这里还需要说明一下，库函数还提供了一个独立的设置计数器值的函数为：

```
void WWDG_SetCounter(uint8_t Counter);
```

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 4 个步骤之后，我们就可以使用 STM32F4 的窗口看门狗了。这一章的实验，我们将通过 DS0 来指示 STM32F4 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 STM32F4 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 STM32F4 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，我们增加了窗口看门狗相关的库函数支持文件 stm32f4xx_wwdg.c 和 stm32f4xx_wwdg.h, 同时新建 wwdg.c 和对应的头文件 wwdg.h 用来编写窗口看门狗相关的函数代码。

接下来我们看看 wwdg.c 文件内容：

```
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). 一般 PCLK1=42Mhz  
void WWDG_Init(u8 tr,u8 wr,u32 fprer)  
{  
    NVIC_InitTypeDef NVIC_InitStruct;  
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG,ENABLE);  
    //使能窗口看门狗时钟  
    WWDG_CNT=tr&WWDG_CNT;    //初始化 WWDG_CNT.  
    WWDG_SetPrescaler(fprer); //设置分频值  
    WWDG_SetWindowValue(wr); //设置窗口值  
    WWDG_SetCounter(WWDG_CNT); //设置计数值  
    WWDG_Enable(WWDG_CNT); //开启看门狗  
  
    NVIC_InitStruct.NVIC_IRQChannel=WWDG_IRQn; //窗口看门狗中断  
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority=0x02; //抢占优先级为 2  
    NVIC_InitStruct.NVIC_IRQChannelSubPriority=0x03; //响应优先级为 3  
    NVIC_InitStruct.NVIC_IRQChannelCmd=ENABLE; //使能窗口看门狗  
    NVIC_Init(&NVIC_InitStruct);  
  
    WWDG_ClearFlag(); //清除提前唤醒中断标志位  
    WWDG_EnableIT(); //开启提前唤醒中断  
}  
  
//窗口看门狗中断服务程序  
void WWDG_IRQHandler(void)  
{  
    WWDG_SetCounter (WWDG_CNT); //重设窗口看门狗值  
    WWDG_ClearFlag(); //清除提前唤醒中断标志位  
    LED1=!LED1;  
}
```

wwdg.c 文件一共包含两个函数。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 5 个步骤的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，就又通过 WWDG_SetCounter 函数把该数值放回到 WWDG_CR 上。

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状况。

wwdg.h 头文件内容比较简单，这里我们就不做过多讲解。

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)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 设置系统中断优先级分组 2
    delay_init(168); // 初始化延时函数
    LED_Init(); // 初始化 LED 端口
    KEY_Init(); // 初始化按键
    LED0=0; // 点亮 LED0
    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; // 熄灭 LED 灯
    }
}
```

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探索者 STM32F4 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

将代码下载到探索者 STM32F4 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 20 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 STM32F4 的通用定时器，STM32F4 的定时器功能十分强大，有 TIME1 和 TIME8 等高级定时器，也有 TIME2~TIME5，TIM9~TIM14 等通用定时器，还有 TIME6 和 TIME7 等基本定时器，总共达 14 个定时器之多。在本章中，我们将使用 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 STM32F4 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 STM32F4 通用定时器简介

STM32F4 的通用定时器包含一个 16 位或 32 位自动重载计数器 (CNT)，该计数器由可编程预分频器 (PSC) 驱动。STM32F4 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。STM32F4 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~TIM5 和 TIM9~TIM14) 定时器功能包括：

- 1) 16 位/32 位(仅 TIM2 和 TIM5)向上、向下、向上/向下自动装载计数器 (TIMx_CNT)，注意：TIM9~TIM14 只支持向上（递增）计数方式。
- 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，TIM9~TIM14 最多 2 个通道)，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)，注意：TIM9~TIM14 不支持中间对齐模式
 - 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 (TIM9~TIM14 不支持 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 (TIM9~TIM14 不支持)
 - F. 触发输入作为外部时钟或者按周期的电流管理 (TIM9~TIM14 不支持)

由于 STM32F4 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《STM32F4xx 中文参考手册》第 392 页，通用定时器一章。下面我们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（以下均以 TIM2~TIM5 的寄存器介绍，TIM9~TIM14 的略有区别，具体请看《STM32F4xx 中文参考手册》对应章节）。

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
Reserved						CKD[1:0]	ARPE	CMS		DIR	OPM	URS	UDIS	CEN	
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 0 **CEN**: 计数器使能 (Counter enable)

- 0: 禁止计数器
- 1: 使能计数器

注意: 只有事先通过软件将 **CEN** 位置 1, 才可以使用外部时钟、门控模式和编码器模式。而触发模式可通过硬件自动将 **CEN** 位置 1。

在单脉冲模式下, 当发生更新事件时会自动将 **CEN** 位清零。

图 13.1.1 TIMx_CR1 寄存器各位描述

在本实验中, 我们只用到了 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。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	
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res.	CC4IE	CC3IE	CC2IE	CC1IE	UIE	
			rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw	

位 0 **UIE**: 更新中断使能 (Update interrupt enable)

- 0: 禁止更新中断
- 1: 使能更新中断

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]**: 预分频器值 (Prescaler value)

计数器时钟频率 CK_CNT 等于 $f_{CK_PSC} / (PSC[15:0] + 1)$ 。

PSC 包含在每次发生更新事件时要装载到实际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), 仅适用于 TIM2、TIM3、TIM4
- 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 (一般都不会是 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以及 TIM9~TIM11 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

《STM32F4xx 中文参考手册》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，此时两者是连通的；而 APRE=1 时，在每一次更新事件（UEV）时，才把预装载寄存器（ARR）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]**: 自动重载值 (Auto-reload value)

ARR 为要装载到实际自动重载寄存器的值。

当自动重载值为空时，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
Reserved	CC4OF	CC3OF	CC2OF	CC1OF	Reserved	TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF			
	rc_w0	rc_w0	rc_w0	rc_w0		rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0			

位 0 **UIF**: 更新中断标志 (Update interrupt flag)

- 该位在发生更新事件时通过硬件置 1。但需要通过软件清零。
0: 未发生更新。
1: 更新中断挂起。该位在以下情况下更新寄存器时由硬件置 1:
 - 上溢或下溢（对于 TIM2 到 TIM5）以及当 TIMx_CR1 寄存器中 UDIS = 0 时。
 - TIMx_CR1 寄存器中的 URS = 0 且 UDIS = 0，并且由软件使用 TIMx_EGR 寄存器中的 UG 位重新初始化 CNT 时。

TIMx_CR1 寄存器中的 URS=0 且 UDIS=0，并且 CNT 由触发事件重新初始化

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《STM32F4xx 中文参考手册》第 429 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 stm32f4xx_tim.h 和 stm32f4xx_tim.c 文件中。定时器配置步骤如下：

1) TIM3 时钟使能。

TIM3 是挂载在 APB1 之下，所以我们通过 APB1 总线下的使能函数来使能 TIM3。调用的函数是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE); //使能 TIM3 时钟
```

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 PF10 上的，这个前面已经有介绍了。而 TIM3 属于 STM32F4 的内部资源，只需要软件设置即可正常工作。

13.3 软件设计

打开我们光盘实验 8 定时器中断实验可以看到，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 stm32f4xx_tim.c 和头文件 stm32f4xx_tim.h。下面我们来看看我们的 time.c 文件。timer.c 文件代码如下：

```
//通用定时器 3 中断初始化
//arr: 自动重装值。 psc: 时钟预分频数
//定时器溢出时间计算方法:Tout=((arr+1)*(psc+1))/Ft us.
//Ft=定时器工作频率,单位:Mhz
//这里使用的是定时器 3!
void TIM3_Int_Init(u16 arr,u16 psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
```

```

RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE); //①使能 TIM3 时钟

TIM_TimeBaseInitStructure.TIM_Period = arr; //自动重装载值
TIM_TimeBaseInitStructure.TIM_Prescaler=psc; //定时器分频
TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;

TIM_TimeBaseInit(TIM3,&TIM_TimeBaseInitStructure); // ②初始化定时器 TIM3

TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE); //③允许定时器 3 更新中断

NVIC_InitStructure.NVIC_IRQChannel=TIM3_IRQn; //定时器 3 中断
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0x01; //抢占优先级 1
NVIC_InitStructure.NVIC_IRQChannelSubPriority=0x03; //响应优先级 3
NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
NVIC_Init(&NVIC_InitStructure); // ④初始化 NVIC

TIM_Cmd(TIM3,ENABLE); //⑤使能定时器 3
}

//定时器 3 中断服务函数
void TIM3_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM3,TIM_IT_Update)==SET) //溢出中断
    {
        LED1=!LED1;
    }
    TIM_ClearITPendingBit(TIM3,TIM_IT_Update); //清除中断标志位
}

```

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，则执行 LED1 (DS1) 的翻转。

TIM3_Int_Init 函数就是执行我们上面介绍的那 5 个步骤，使得 TIM3 开始工作，并开启中断。这里我们分别用标号①~⑤来标注定时器初始化的五个步骤。该函数的 2 个参数用来设置 TIM3 的溢出时间。因为系统初始化 SystemInit 函数里面已经初始化 APB1 的时钟为 4 分频，所以 APB1 的时钟为 42M，而从 STM32F4 的内部时钟树图（图 4.3.1.1）得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 以及 TIM12~14 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 以及 TIM12~14 的时钟频率将为 APB1 时钟的两倍。因此，TIM3 的时钟为 84M，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 头文件内容比较简单，这里我们就不做讲解。

最后，我们看看主函数代码如下：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    LED_Init(); //初始化 LED 端口
    TIM3_Int_Init(5000-1,8400-1); //定时器时钟 84M, 分频系数 8400, 所以 84M/8400=10Khz
                                //的计数频率, 计数 5000 次为 500ms
    while(1)
    {
        LED0=!LED0;
        delay_ms(200); //延时 200ms
    };
}
```

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。

这里定时器定时长 500ms 是这样计算出来的，定时器的时钟为 84Mhz，分频系数为 8400，所以分频后的计数频率为 $84\text{Mhz}/8400=10\text{KHz}$ ，然后计数到 5000，所以时长为 $5000/10000=0.5\text{s}$ ，也就是 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探索者 STM32F4 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 STM32F4 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 STM32F4 的 TIM3 来产生 PWM 输出。在本章中，我们将使用 TIM14 的通道 1 来产生 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，PWM 原理如图 14.1.1 所示：

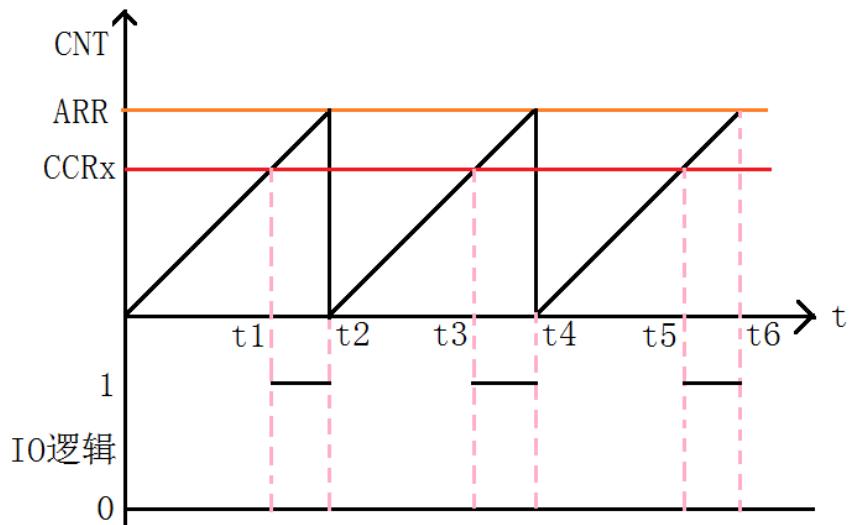


图 14.1.1 PWM 原理示意图

图 14.1.1 就是一个简单的 PWM 原理示意图。图中，我们假定定时器工作在向上计数 PWM 模式，且当 $CNT < CCRx$ 时，输出 0，当 $CNT \geq CCRx$ 时输出 1。那么就可以得到如上的 PWM 示意图：当 CNT 值小于 CCRx 的时候，IO 输出低电平(0)，当 CNT 值大于等于 CCRx 的时候，IO 输出高电平(1)，当 CNT 达到 ARR 值的时候，重新归零，然后重新向上计数，依次循环。改变 CCRx 的值，就可以改变 PWM 输出的占空比，改变 ARR 的值，就可以改变 PWM 输出的频率，这就是 PWM 输出的原理。

STM32F4 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！这里我们仅使用 TIM14 的 CH1 产生一路 PWM 输出。

要使 STM32F4 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)，该寄存器一般有 2 个：TIMx_CCMR1

和 TIMx_CCMR2，不过 TIM14 只有一个。TIMx_CCMR1 控制 CH1 和 2，而 TIMx_CCMR2 控制 CH3 和 4。以下我们将以 TIM14 为例进行介绍。TIM14_CCMR1 寄存器各位描述如图 14.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]		
Reserved								IC1F[3:0]			IC1PSC[1:0]				
								rw	rw	rw	rw	rw	rw	rw	rw

图 14.1.2 TIM14_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 14.1.2 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《STM32F4xx 中文参考手册》第 476 页，16.6.4 节。这里我们需要说明的是模式设置位 OC1M，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。另外 CC1S 用于设置通道的方向（输入/输出）默认设置为 0，就是设置通道作为输出使用。注意：这里是因为我们的 TIM14 只有 1 个通道，所以才只有第八位有效，高八位无效，其他有多个通道的定时器，高八位也是有效的，具体请参考《STM32F4xx 中文参考手册》对应定时器的寄存器描述。

接下来，我们介绍 TIM14 的捕获/比较使能寄存器 (TIM14_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 14.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								CC1NP		Res.	CC1P		CC1E		
								rw			rw		rw		rw

图 14.1.3 TIM14_CCER 寄存器各位描述

该寄存器比较简单，我们这里只用到了 CC1E 位，该位是输入/捕获 1 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《STM32F4xx 中文参考手册》第 478 页，16.6.5 这一节。同样，因为 TIM14 只有 1 个通道，所以才只有低四位有效，如果是其他定时器，该寄存器的其他位也可能有效。

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 个通道 CH1~4。不过 TIM14 只有一个，即：TIM14_CCR1，该寄存器的各位描述如图 14.1.4 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 值 (Capture/Compare 1 value)

如果通道 CC1 配置为输出：

CCR1 为要装载到实际捕获/比较 1 寄存器的值（预装载值）。

如果没有通过 TIMx_CCMR 寄存器中的 OC1PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只在发生更新事件时生效（拷贝到实际起作用的捕获/比较寄存器）。实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC1 输出上发出信号的值。

如果通道 CC1 配置为输入：

CCR1 为上一个输入捕获 1 事件 (IC1) 发生时的计数器值。

图 14.1.4 寄存器 TIM14_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。

如果是通用定时器，则配置以上三个寄存器就够了，但是如果是高级定时器，则还需要配置：刹车和死区寄存器 (TIMx_BDTR)，该寄存器各位描述如图 14.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15 **MOE**: 主输出使能 (Main output enable)

只要断路输入变为有效状态，此位便由硬件异步清零。此位由软件置 1，也可根据 AOE 位状态自动置 1。此位仅对配置为输出的通道有效。

0: OC 和 OCN 输出禁止或被强制为空闲状态。

1: 如果 OC 和 OCN 输出的相应使能位 (TIMx_CCER 寄存器中的 CCxE 和 CCxNE 位) 均置 1，则使能 OC 和 OCN 输出。

图 14.1.5 寄存器 TIMx_BDTR 各位描述

该寄存器，我们只需要关注最高位：MOE 位，要想高级定时器的 PWM 正常输出，则必须设置 MOE 位为 1，否则不会有输出。注意：通用定时器不需要配置这个。其他位我们这里就不详细介绍，请参考《STM32F4xx 中文参考手册》第 386 页，14.4.18 这一节。

本章，我们使用的是 TIM14 的通道 1，所以我们需要修改 TIM14_CCR1 以实现脉宽控制 DS0 的亮度。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 TIM14_CH1 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 实际跟上一章节一样使用的是定时器的功能，所以相关的函数设置同样在库函数文件 stm32f4xx_tim.h 和 stm32f4xx_tim.c 文件中。

1) 开启 TIM14 和 GPIO 时钟，配置 PF9 选择复用功能 AF9 (TIM14) 输出。

要使用 TIM14，我们必须先开启 TIM14 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 PF9 为复用 (AF9) 输出，才可以实现 TIM14_CH1 的 PWM 经过 PF9 输出。库函数使能 TIM14 时钟的方法是：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM14,ENABLE); //TIM14 时钟使能  
这在前面章节已经提到过。当然，这里我们还要使能 GPIOF 的时钟。然后我们要配置 PF9 引脚映射至 AF9，复用为定时器 14，调用的函数为：
```

```
GPIO_PinAFConfig(GPIOF,GPIO_PinSource9,GPIO_AF_TIM14); //GPIOF9 复用为定时器 14  
这个方法跟我们串口实验讲解一样，调用的同一个函数，至于函数的使用，在我们的 4.4 小节有详细的讲解。最后设置 PF9 为复用功能输出这里我们只列出 GPIO 初始化为复用功能的一行代码：
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
```

这里还需要说明一下，对于定时器通道的引脚关系，大家可以查看 STM32F4 对应的数据手册，比如我们 PWM 实验，我们使用的是定时器 14 的通道 1，对应的引脚 PF9 可以从数据手册表中查看：

PF9	I/O	FT	(4)	TIM14_CH1 / FSMC_CD/ EVENTOUT	ADC3_IN7
-----	-----	----	-----	----------------------------------	----------

2) 初始化 TIM14，设置 TIM14 的 ARR 和 PSC 等参数。

在开启了 TIM14 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:TDDTS = 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
3) 设置 TIM14 CH1 的 PWM 模式，使能 TIM14 的 CH1 输出。

```

接下来，我们要设置 TIM14_CH1 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 CCR1 的值小的时候，DS0 就暗，CCR1 值大的时候，DS0 就亮，所以我们要通过配置 TIM14_CCMR1 的相关位来控制 TIM14_CH1 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 1，所以使用的函数是 TIM_OC1Init()。

```
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
```

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_OCInitTypeDef 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_PWM1; //选择模式 PWM
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //输出极性低
TIM_OC1Init(TIM14, &TIM_OCInitStructure); //根据 T 指定的参数初始化外设 TIM1 4OC1

```

4) 使能 TIM14。

在完成以上设置了之后，我们需要使能 TIM14。使能 TIM14 的方法前面已经讲解过：

```
TIM_Cmd(TIM14, ENABLE); //使能 TIM14
```

5) 修改 TIM14_CCR1 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 TIM14_CCR1 则可以控制 CH1 的输出占空比。继而控制 DS0 的亮度。

在库函数中，修改 TIM14_CCR1 占空比的函数是：

```
void TIM_SetCompare1(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 5 个步骤，我们就可以控制 TIM14 的 CH1 输出 PWM 波了。这里特别提醒一下大家，高级定时器虽然和通用定时器类似，但是高级定时器要想输出 PWM，必须还要设置一个 MOE 位(TIMx_BDTR 的第 15 位)，以使能主输出，否则不会输出 PWM。库函数设置的函数为：

```
void TIM_CtrlPWMOutputs(TIM_TypeDef* TIMx, FunctionalState NewState)
```

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 TIM14

这两个我们前面都已经介绍了，因为 TIM14_CH1 可以通过 PF9 输出 PWM，而 DS0 就是直接节在 PF9 上面的，所以电路上并没有任何变化。

14.3 软件设计

打开实验 9 PWM 输出实验代码可以看到，我们相比上一节，并没有添加其他任何固件库文件，而是添加了我们编写的 PWM 配置文件 pwm.c 和 pwm.h。

pwm.c 源文件代码如下：

```
//TIM14 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 TIM14_PWM_Init(u32 arr,u32 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM14,ENABLE); //TIM14 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE); //使能 PORTF 时钟

    GPIO_PinAFConfig(GPIOF,GPIO_PinSource9,GPIO_AF_TIM14); //GF9 复用为 TIM14

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;           //GPIOF9
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;         //复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;   //速度 50MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;       //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;         //上拉
    GPIO_Init(GPIOF,&GPIO_InitStructure);               //初始化 PF9

    TIM_TimeBaseStructure.TIM_Prescaler=psc; //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseStructure.TIM_Period=arr; //自动重装载值
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
```

```
TIM_TimeBaseInit(TIM14,&TIM_TimeBaseStructure); //初始化定时器 14
```

```
//初始化 TIM14 Channel1 PWM 模式
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; //PWM 调制模式 1
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //输出极性低
TIM_OC1Init(TIM14, &TIM_OCInitStructure); //初始化外设 TIM1 4OC1
TIM_OC1PreloadConfig(TIM14, TIM_OCPreload_Enable); //使能预装载寄存器
TIM_ARRPreloadConfig(TIM14,ENABLE); //ARPE 使能
TIM_Cmd(TIM14, ENABLE); //使能 TIM14
}
```

此部分代码包含了上面介绍的 PWM 输出设置的前 5 个步骤。这里我们关于 TIM14 的设置就不再说了。

接下来，我们看看主程序里面的 main 函数如下：

```
int main(void)
{
    u16 led0pwmval=0;
    u8 dir=1;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    TIM14_PWM_Init(500-1,84-1); //定时器时钟为 84M，分频系数为 84，所以计数频率
                                //为 84M/84=1Mhz,重装载值 500，所以 PWM 频率为 1M/500=2Khz.
    while(1)
    {
        delay_ms(10);
        if(dir)led0pwmval++; //dir==1 led0pwmval 递增
        else led0pwmval--; //dir==0 led0pwmval 递减
        if(led0pwmval>300)dir=0; //led0pwmval 到达 300 后，方向为递减
        if(led0pwmval==0)dir=1; //led0pwmval 递减到 0 后，方向改为递增

        TIM_SetCompare1(TIM14,led0pwmval); //修改比较值，修改占空比
    }
}
```

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 的亮度也会跟着信号的占空比变化从暗变到亮，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 499），因此设计过大的值在这里是没有必要的。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探索者 STM32F4 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：



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 STM32F4 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 KEY_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。我们以测量脉宽为例，用一个简图来说明输入捕获的原理，如图 15.1.1 所示：

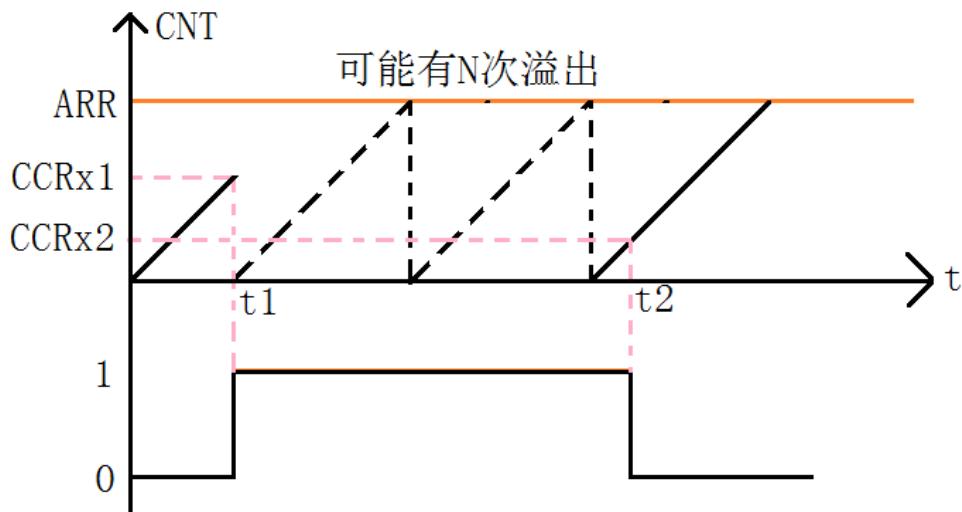


图 15.1.1 输入捕获脉宽测量原理

如图 15.1.1 所示，就是输入捕获测量高电平脉宽的原理，假定定时器工作在向上计数模式，图中 $t_1 \sim t_2$ 时间，就是我们需要测量的高电平时间。测量方法如下：首先设置定时器通道 x 为上升沿捕获，这样， t_1 时刻，就会捕获到当前的 CNT 值，然后立即清零 CNT，并设置通道 x 为下降沿捕获，这样到 t_2 时刻，又会发生捕获事件，得到此时的 CNT 值，记为 $CCRx_2$ 。这样，根据定时器的计数频率，我们就可以算出 $t_1 \sim t_2$ 的时间，从而得到高电平脉宽。

在 $t_1 \sim t_2$ 之间，可能产生 N 次定时器溢出，这就要求我们对定时器溢出，做处理，防止高电平太长，导致数据不准确。如图 15.1.1 所示， $t_1 \sim t_2$ 之间，CNT 计数的次数等于： $N * ARR + CCRx_2$ ，有了这个计数次数，再乘以 CNT 的计数周期，即可得到 $t_2 - t_1$ 的时间长度，即高电平持续时间。输入捕获的原理，我们就介绍到这。

STM32F4 的定时器，除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。STM32F4 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，捕获原理如图 15.1.1 所示，这里我们就不再多说了。

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 15.1.2 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	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw

图 15.1.2 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 15.1.2 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《STM32F4xx 中文参考手册》435 页, 15.4.8 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 (其高 8 位配置类似), TIMx_CMMR1 的[7:0]位详细描述见图 15.1.3 所示:

位 7:4 IC1F: 输入捕获 1 滤波器 (Input capture 1 filter)

此位域可定义 TI1 输入的采样频率和适用于 TI1 的数字滤波器带宽。数字滤波器由事件计数器组成, 每 N 个事件才视为一个有效边沿:

0000: 无滤波器, 按 f_{DTS} 频率进行采样	1000: $f_{SAMPLING}=f_{DTS}/8, N=6$
0001: $f_{SAMPLING}=f_{CK_INT}, N=2$	1001: $f_{SAMPLING}=f_{DTS}/8, N=8$
0010: $f_{SAMPLING}=f_{CK_INT}, N=4$	1010: $f_{SAMPLING}=f_{DTS}/16, N=5$
0011: $f_{SAMPLING}=f_{CK_INT}, N=8$	1011: $f_{SAMPLING}=f_{DTS}/16, N=6$
0100: $f_{SAMPLING}=f_{DTS}/2, N=6$	1100: $f_{SAMPLING}=f_{DTS}/16, N=8$
0101: $f_{SAMPLING}=f_{DTS}/2, N=8$	1101: $f_{SAMPLING}=f_{DTS}/32, N=5$
0110: $f_{SAMPLING}=f_{DTS}/4, N=6$	1110: $f_{SAMPLING}=f_{DTS}/32, N=6$
0111: $f_{SAMPLING}=f_{DTS}/4, N=8$	1111: $f_{SAMPLING}=f_{DTS}/32, N=8$

注意: 在当前硅版本中, 当 $ICxF[3:0]=1, 2$ 或 3 时, 将用 CK_INT 代替公式中的 f_{DTS} 。

位 3:2 IC1PSC: 输入捕获 1 预分频器 (Input capture 1 prescaler)

此位域定义 CC1 输入 (IC1) 的预分频比。

只要 $CC1E=0$ (TIMx_CCER 寄存器), 预分频器便立即复位。

00: 无预分频器, 捕获输入上每检测到一个边沿便执行捕获

01: 每发生 2 个事件便执行一次捕获

10: 每发生 4 个事件便执行一次捕获

11: 每发生 8 个事件便执行一次捕获

位 1:0 CC1S: 捕获/比较 1 选择 (Capture/Compare 1 selection)

此位域定义通道方向 (输入/输出) 以及所使用的输入。

00: CC1 通道配置为输出

01: CC1 通道配置为输入, IC1 映射到 TI1 上

10: CC1 通道配置为输入, IC1 映射到 TI2 上

11: CC1 通道配置为输入, IC1 映射到 TRC 上。此模式仅在通过 TS 位 (TIMx_SMCR 寄存器) 选择内部触发输入时有效

注意: 仅当通道关闭时 (TIMx_CCER 中的 $CC1E=0$), 才可向 CC1S 位写入数据。

图 15.1.3 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《STM32F4xx 中文参考手册》393 页的图 119-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 (TIMx_CLK)，一般为 84Mhz/168Mhz (看该定时器在那个总线上)，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 14.1.3（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 15.1.4 所示：

位 1 CC1P：捕获/比较 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：

CC1NP/CC1P 位可针对触发或捕获操作选择 TI1FP1 和 TI2FP1 的极性。

00：非反相/上升沿触发

电路对 TIxFP1 上升沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式或编码器模式下执行触发操作）。

01：反相/下降沿触发

电路对 TIxFP1 下降沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 反相（在门控模式或编码器模式下执行触发操作）。

10：保留，不使用此配置。

11：非反相/上升沿和下降沿均触发

电路对 TIxFP1 上升沿和下降沿都敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式下执行触发操作）。编码器模式下不得使用此配置。

位 0 CC1E：捕获/比较 1 输出使能 (Capture/Compare 1 output enable)。

CC1 通道配置为输出：

0：关闭——OC1 未激活

1：开启——在相应输出引脚上输出 OC1 信号

CC1 通道配置为输入：

此位决定了是否可以实际将计数器值捕获到输入捕获/比较寄存器 1 (TIMx_CCR1) 中。

0：禁止捕获

1：使能捕获

图 15.1.4 TIMx_CCER 最低 2 位描述

所以，要使能输入捕获，必须设置 CC1E=1，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(注意, 对于脉宽太长的情况, 还要计算定时器溢出的次数)。

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。下面我们介绍库函数配置上述功能输入捕获的步骤:

1) 开启 TIM5 时钟, 配置 PA0 为复用功能 (AF2), 并开启下拉电阻。

要使用 TIM5, 我们必须先开启 TIM5 的时钟。同时我们要捕获 TIM5_CH1 上面的高电平脉宽, 所以先配置 PA0 为带下拉的复用功能, 同时, 为了让 PA0 的复用功能选择连接到 TIM5, 所以设置 PA0 的复用功能为 AF2, 即连接到 TIM5 上面。

开启 TIM5 时钟的方法为:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,ENABLE); //TIM5 时钟使能
```

当然, 这里我们也要开启 PA0 对应的 GPIO 的时钟。

配置 PA0 为复用功能, 所以我们首先要设置 PA0 引脚映射 AF2, 方法为:

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource0,GPIO_AF_TIM5); //GPIOF9 复用位定时器 14
```

最后, 我们还要初始化 GPIO 的模式为复用功能, 同时这里我们还要设置为开启下拉。方法为:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //GPIOA0
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA0
```

跟上一讲 PWM 输出类似, 这里我们使用的是定时器 5 的通道 1, 所以我们从 STM32F4 对应的数据手册可以查看到对应的 IO 口为 PA0:

PA0/WKUP (PA0)	I/O	FT	(5)	USART2_CTS/ UART4_TX/ ETH_MII_CRS / TIM2_CH1_ETR/ TIM5_CH1/TIM8_ETR/ EVENTOUT	ADC123_IN0/WKUP ⁽⁴⁾
-------------------	-----	----	-----	---	--------------------------------

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_TimeBaseStructure.TIM_Prescaler=psc; //定时器分频
TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
TIM_TimeBaseStructure.TIM_Period=arr; //自动重装载值
TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
TIM_TimeBaseInit(TIM5,&TIM_TimeBaseStructure); //初始化 TIM5
```

3) 设置 TIM5 的输入捕获参数, 开启输入捕获。

TIM5_CCMR1 寄存器控制着输入捕获 1 和 2 的模式, 包括映射关系, 滤波和分频等。这里

我们需要设置通道 1 为输入模式，且 IC1 映射到 TI1(通道 1)上面，并且不使用滤波（提高响应速度）器。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; //通道
    uint16_t TIM_ICPolarity; //捕获极性
    uint16_t TIM_ICSelection; //映射
    uint16_t TIM_ICPrescaler; //分频系数
    uint16_t TIM_ICFilter; //滤波器长度
} 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_ICPolarit 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：

```
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，不过，由于 STM32F4 的 TIM5 是 32 位定时器，假设计数周期为 1us，那么需要 4294 秒才会溢出一次，这基本上是不可能的。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) 设置中断优先级，编写中断服务函数

因为我们要使用到中断，所以我们在系统初始化之后，需要先设置中断优先级分组，这里方法跟我们前面讲解一致，调用 NVIC_PriorityGroupConfig()函数即可，我们系统默认设置都是分组 2。设置中断优先级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。设置优先级完成后，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
在我们实验的中断服务函数中，我们还使用到了一个设置计数器值的函数：
TIM_SetCounter(TIM5,0);
```

上面语句的意思是将 TIM5 的计数值设置为 0。这个相信是比较容易理解的。

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，同时因为还用到了串口输出结果，所以还需要配置一下串口。

15.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY_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 KEY_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 PF9 和 PA0，来测量 PWM 输出的高电平脉宽。

15.3 软件设计

相比上一章讲解的 PWM 实验，这里我们将相应的驱动文件名称由 pwm.c 和 pwm.h 改为 timer.c 和 timer.h，然后我们在 timer.c 和 timer.h 中主要是添加了输入捕获初始化函数 TIM5_CH1_Cap_Init 以及中断服务函数 TIM5_IRQHandler。对于输入捕获，我们也是使用的定时器相关的操作，所以相比上一实验，我们并没有添加其他任何固件库文件。

接下来我们来看看 timer.c 文件中，我们添加的两个函数的内容：

```
TIM_ICInitTypeDef TIM5_ICInitStructure;
//定时器 5 通道 1 输入捕获配置
//arr: 自动重装值(TIM2,TIM5 是 32 位的!!)          psc: 时钟预分频数
void TIM5_CH1_Cap_Init(u32 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
```

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,ENABLE); //TIM5 时钟使能
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PORTA 时钟

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //GPIOA0
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA0

GPIO_PinAFConfig(GPIOA,GPIO_PinSource0,GPIO_AF_TIM5); //PA0 复用位定时器 5

TIM_TimeBaseStructure.TIM_Prescaler=psc; //定时器分频
TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
TIM_TimeBaseStructure.TIM_Period=arr; //自动重装载值
TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
TIM_TimeBaseInit(TIM5,&TIM_TimeBaseStructure);

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 输入捕获参数

TIM_ITConfig(TIM5,TIM_IT_Update|TIM_IT_CC1,ENABLE);//允许更新和捕获中断
TIM_Cmd(TIM5,ENABLE ); //使能定时器 5

NVIC_InitStructure.NVIC_IRQChannel = TIM5_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=2;//抢占优先级 2
NVIC_InitStructure.NVIC_IRQChannelSubPriority =0;//响应优先级 0
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、

}

//捕获状态
//[7]:0,没有成功的捕获;1,成功捕获到一次.
//[6]:0,还没捕获到低电平;1,已经捕获到低电平了.
//[5:0]:捕获低电平后溢出的次数(对于 32 位定时器来说,1us 计数器加 1,溢出时间:4294 秒)
u8  TIM5CH1_CAPTURE_STA=0; //输入捕获状态
u32 TIM5CH1_CAPTURE_VAL;//输入捕获值(TIM2/TIM5 是 32 位)
//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)//溢出
    {
        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=0xFFFFFFFF;
            }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高电平脉宽
            TIM5CH1_CAPTURE_VAL=TIM_GetCapture1(TIM5); //获取当前的捕获值.
            TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Rising); //设置上升沿捕获
        }else                                //还未开始,第一次捕获上升沿
        {
            TIM5CH1_CAPTURE_STA=0;//清空
            TIM5CH1_CAPTURE_VAL=0;
            TIM5CH1_CAPTURE_STA|=0X40;//标记捕获到了上升沿
            TIM_Cmd(TIM5,ENABLE ); //使能定时器 5
            TIM_SetCounter(TIM5,0); //计数器清空
            TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Falling); //设置下降沿捕获
            TIM_Cmd(TIM5,ENABLE ); //使能定时器 5
        }
    }
}
TIM_ClearITPendingBit(TIM5, TIM_IT_CC1|TIM_IT_Update); //清除中断标志位
}

```

此部分代码包含两个函数，其中 `TIM5_CH1_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，特别注意：`TIM5` 是 32 位定时器，所以 `arr` 是 `u32` 类型的。接下来，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 `uart.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` 和计数器值 `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(对 32 位定时器来说, 很难溢出), 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。

`timer.h` 头文件内容比较简单, 主要是函数申明, 这里我们不做过多讲解。

接下来, 我们看看 `main` 函数内容:

```
extern u8  TIM5CH1_CAPTURE_STA;          //输入捕获状态
extern u32   TIM5CH1_CAPTURE_VAL;//输入捕获值
int main(void)
{
    long long temp=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200);//初始化串口波特率为 115200
    TIM14_PWM_Init(500-1,84-1);
    //84M/84=1Mhz 的计数频率计数到 500,频率为 1M/500=2Khz
    TIM5_CH1_Cap_Init(0xFFFFFFFF,84-1);//以 84M/84=1Mhz 的频率计数
    while(1)
    {
        delay_ms(10);
        TIM_SetCompare1(TIM14,TIM_GetCapture1(TIM14)+1);
        if(TIM_GetCapture1(TIM14)==300)TIM_SetCompare1(TIM14,0);
        if(TIM5CH1_CAPTURE_STA&0X80)//成功捕获到了一次高电平
        {
            temp=TIM5CH1_CAPTURE_STA&0X3F;
            temp*=0xFFFFFFFF;           //溢出时间总和
            temp+=TIM5CH1_CAPTURE_VAL; //得到总的高电平时间
            printf("HIGH:%lld 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(0xFFFFFFFF,84-1)，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以达到不让定时器溢出的作用（溢出时间为 $2^{32}-1$ us），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探索者 STM32F4 开发板上，可以看到 DSO 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 KEY_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：

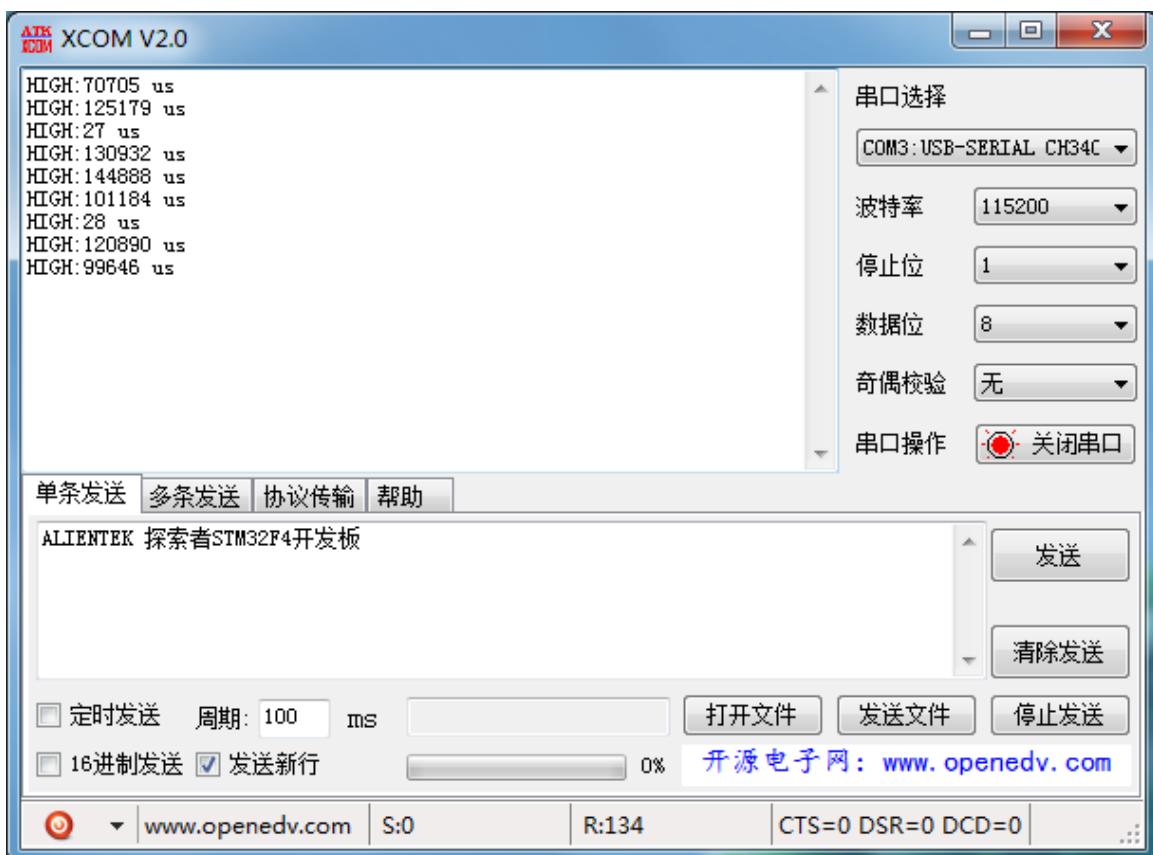


图 15.4.1 PWM 控制 DSO 亮度

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 PF9，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 STM32F4 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 TIM2 的通道 1 (PA5) 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。

我们将利用探索者 STM32F4 开发板上的触摸按键 (TPAD)，来实现对 DS1 的亮灭控制。这里 TPAD 其实就是探索者 STM32F4 开发板上的一小块覆铜区域，实现原理如图 16.1.1 所示：

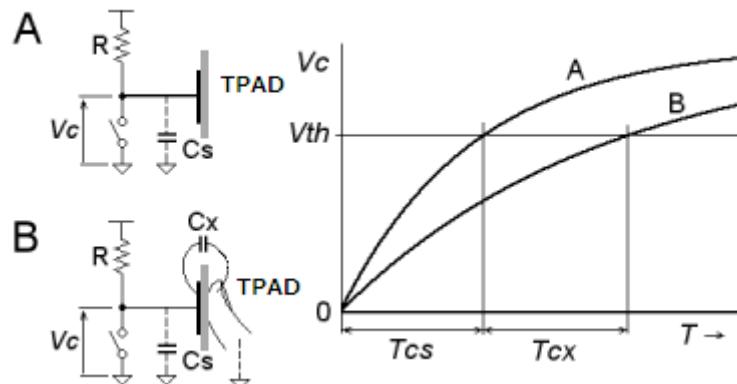


图 16.1.1 电容触摸按键原理

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 STM32F4 的 IO 代替）。

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：

$$V_c = V_0 * (1 - e^{-(t/RC)})$$

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，我们还可以把探索者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 T_{cs} 附近，就可以认为没有触摸，而当充电时间大于 $T_{cs}+T_x$ 时，就认为有触摸按下（ T_x 为检测阈值）。

本章，我们使用 PA5(TIM2_CH1) 来检测 TPAD 是否有触摸，在每次检测之前，我们先配置 PA5 为推挽输出，将电容 C_s （或 C_s+C_x ）放电，然后配置 PA5 为浮空输入，利用外部上拉电阻给电容 $C_s(C_s+C_x)$ 充电，同时开启 TIM2_CH1 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，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 TIM2
- 3) 触摸按键 TPAD

前面两个之前均有介绍，我们需要通过 TIM2_CH1 (PA5) 采集 TPAD 的信号，所以本实验需要用跳线帽短接多功能端口 (P12) 的 TPAD 和 ADC，以实现 TPAD 连接到 PA5。如图 16.2.1 所示：

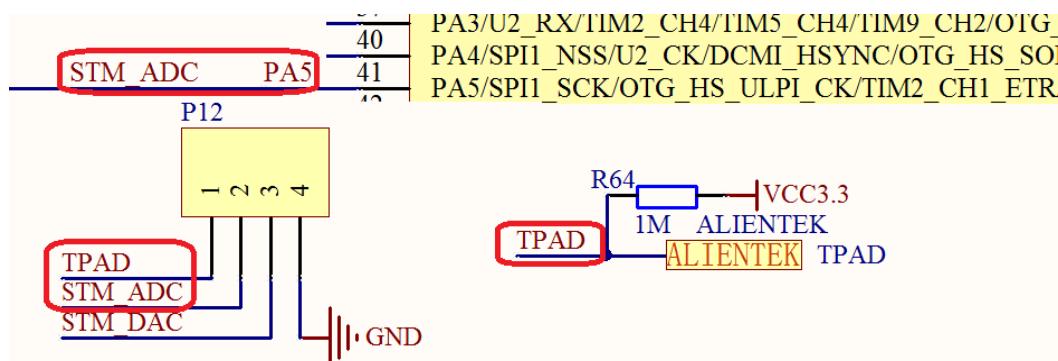


图 16.2.1 TPAD 与 STM32F4 连接原理图

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

软件设计方面，相比上一实验，我们删掉了 timer.c 和 timer.h 文件，同时新建了 tpad.c 和 tpad.h 文件。因为 tpad 我们也是使用的定时器输入捕获来实现，所以我们相比上个实验并没有增加任何库函数相关的文件。

接下来我们看看 tpad.c 文件代码：

```
#define TPAD_ARR_MAX_VAL 0xFFFFFFFF //最大的 ARR 值(TIM2 是 32 位定时器)
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.
//psc:分频系数,越小,灵敏度越高.
//返回值:0,初始化成功;1,初始化失败
```

```
u8 TPAD_Init(u8 psc)
{
    u16 buf[10],temp;  u8 j,i;
    TIM2_CH1_Cap_Init(TPAD_ARR_MAX_VAL,psc-1); //设置分频系数
    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);
    if(tpad_default_val>TPAD_ARR_MAX_VAL/2) return 1;
    //初始化遇到超过 TPAD_ARR_MAX_VAL/2 的数值,不正常!
    return 0;
}
//复位一次
//释放电容电量, 并清除定时器的计数值
void TPAD_Reset(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //PA5
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
    GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA5

    GPIO_ResetBits(GPIOA,GPIO_Pin_5); //输出 0,放电
    delay_ms(5);
    TIM_ClearITPendingBit(TIM2, TIM_IT_CC1|TIM_IT_Update); //清除中断标志
    TIM_SetCounter(TIM2,0); //归 0
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //PA5
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//速度 100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;//不带上下拉
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA5

}

//得到定时器捕获值
//如果超时,则直接返回定时器的计数值.
//返回值: 捕获值/计数值 (超时的情况下返回)
u16 TPAD_Get_Val(void)
{
    TPAD_Reset();
    while(TIM_GetFlagStatus(TIM2, TIM_IT_CC1) == RESET)//等待捕获上升沿
    {
        if(TIM_GetCounter(TIM2)>TPAD_ARR_MAX_VAL-500)return
            TIM_GetCounter(TIM2);//超时了,直接返回 CNT 的值
    };
    return TIM_GetCapture1(TIM2);
}

//读取 n 次,取最大值
//n: 连续获取的次数
//返回值: n 次读数里面读到的最大读数值
u16 TPAD_Get_MaxVal(u8 n)
{
    u16 temp=0,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(按每一次必须松开才能按下一步);
//      1,支持连续触发(可以一直按下)
//返回值:0,没有按下;1,有按下;
#define TPAD_GATE_VAL 100 //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,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)&&rval<(10*tpad_default_val))
    //大于 tpad_default_val+TPAD_GATE_VAL,且小于 10 倍 tpad_default_val,则有效
{
    if((keyen==0)&&(rval>(tpad_default_val+TPAD_GATE_VAL)))    res=1;
    //大于 tpad_default_val+TPAD_GATE_VAL,有效
    keyen=3;          //至少要再过 3 次之后才能按键有效
}
if(keyen)keyen--;
return res;
}

//定时器 2 通道 2 输入捕获配置
//arr: 自动重装值
//psc: 时钟预分频数
void TIM2_CH1_Cap_Init(u32 arr,u16 psc)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_ICInitTypeDef  TIM2_ICInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE); //TIM2 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PORTA 时钟

    GPIO_PinAFConfig(GPIOA,GPIO_PinSource5,GPIO_AF_TIM2); //PA5 复用位定时器 2

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //GPIOA5
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//速度 100MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;//不带上下拉
    GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA5

    //初始化 TIM2
    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;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数
```

```

TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure); // 初始化定时器 2
//初始化通道 1
TIM2_ICInitStructure.TIM_Channel = TIM_Channel_1; //选择输入端 IC1 映射到 TIM2
TIM2_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; //上升沿捕获
TIM2_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
TIM2_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频
TIM2_ICInitStructure.TIM_ICFilter = 0x00;//IC2F=0000 配置输入滤波器 不滤波
TIM_ICInit(TIM2, &TIM2_ICInitStructure);//初始化 TIM2 IC1

TIM_Cmd(TIM2,ENABLE ); //使能定时器 2
}

```

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：TIM2_CH1_Cap_Init、TPAD_Get_Val、TPAD_Init 和 TPAD_Scan。

首先介绍 TIM2_CH1_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 TIM2 上一章是 TIM5。通过该函数的设置，我们将可以捕获 PA5 上的上升沿，同样 TIM2 也是 32 位定时器。

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，同时设置通过调用函数 TIM_SetCounter(TIM2,0) 将计数值 TIM2_CNT 设置为 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 TIM2_CH1_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 100。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。

tpad.h 头文件部分代码比较简单，这里不做介绍。

接下来我们看看主函数代码如下：

```

int main(void)
{
    u8 t=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    TPAD_Init(8); //初始化触摸按键,以 84/4=21Mhz 频率计数
}

```

```

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(8)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。注意在修改 main 函数之后，还需要在 main.c 里面添加 tpad.h 头文件，否则会报错哦。

这里还要提醒一下大家，不要把 uart_init(115200);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探索者 STM32F4 开发板上，可以看到 DS0 慢速闪烁，此时，我们用手指触摸 ALIENTEK 探索者 STM32F4 开发板上的 TPAD（右下角的白色头像），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：

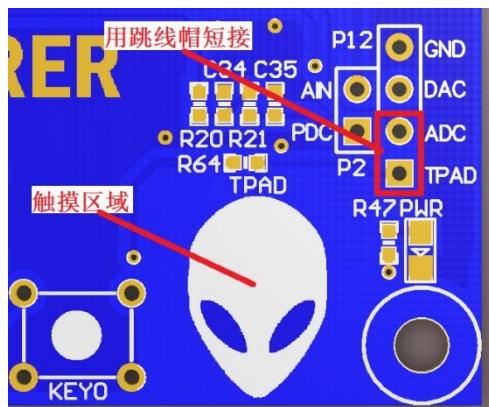


图 16.4.1 触摸区域和跳线帽短接方式示意图

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 250 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，我们将使用探索者 STM32F4 开发板上的 OLED 模块接口，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 4 种接口包括：6800、8080 两种并行接口方式、4 线 SPI 接口方式以及 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 4 种模式通过模块的 BS1 和 BS2 设置，BS1 和 BS2 的设置与模块接口模式的关系如表 17.1.1 所示：

接口方式	4 线 SPI	IIC	8 位 6800	8 位 8080
BS1	0	1	0	1
BS2	0	0	1	1

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：

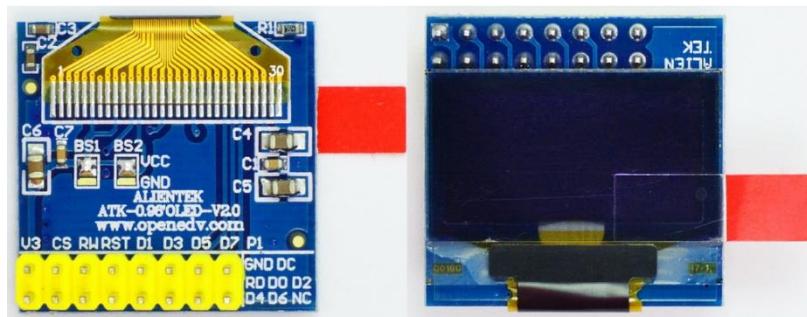


图 17.1.1 ALIENTEK OLED 模块外观图

ALIENTEK OLED 模块默认设置是：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 BS1 和 BS2 的设置。

模块的原理图如图 17.1.2 所示：

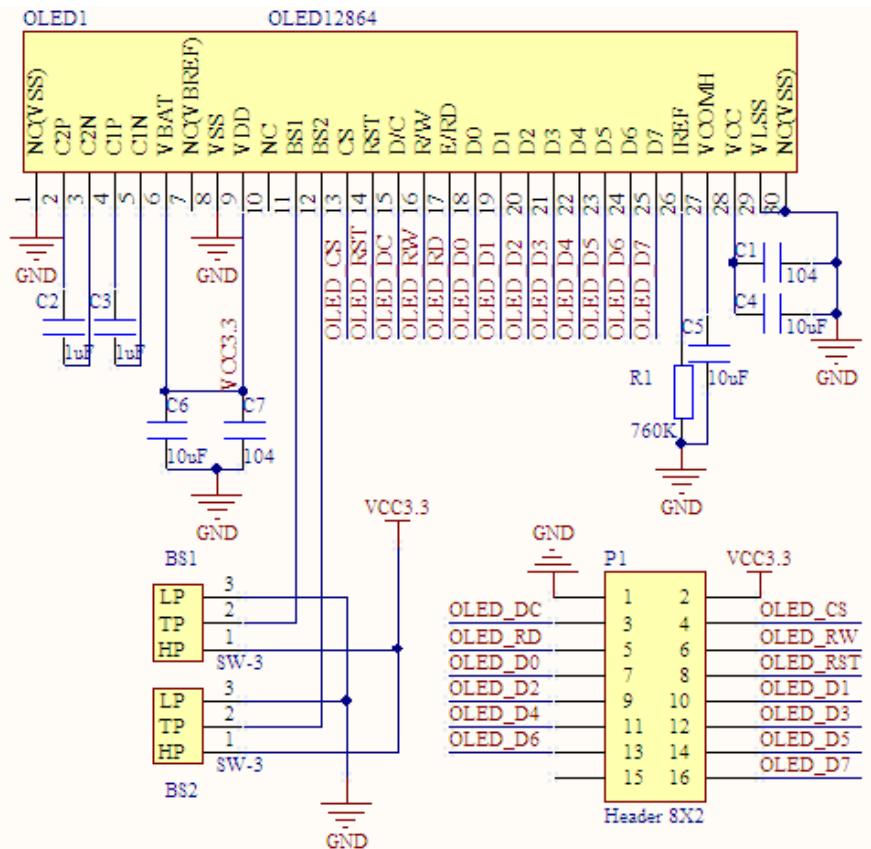


图 17.1.2 ALIENTEK OLED 模块原理图

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 STM32F4 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两种方式与 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：

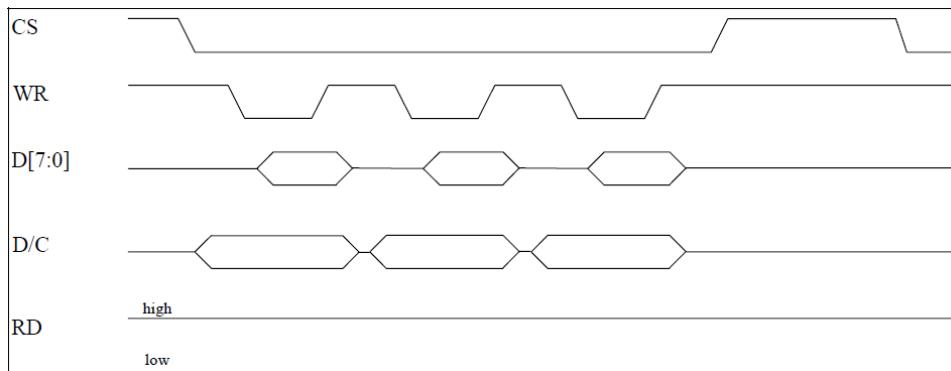


图 17.1.3 8080 并口写时序图

SSD1306 的 8080 并口读时序图如图 17.1.4 所示：

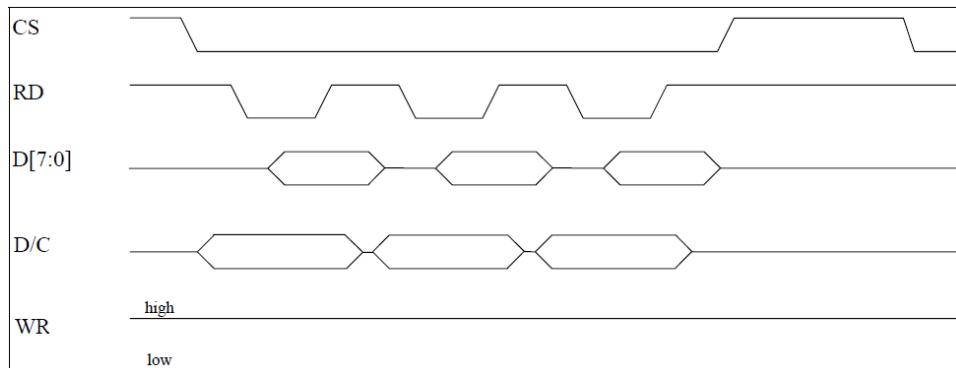


图 17.1.4 8080 并口读时序图

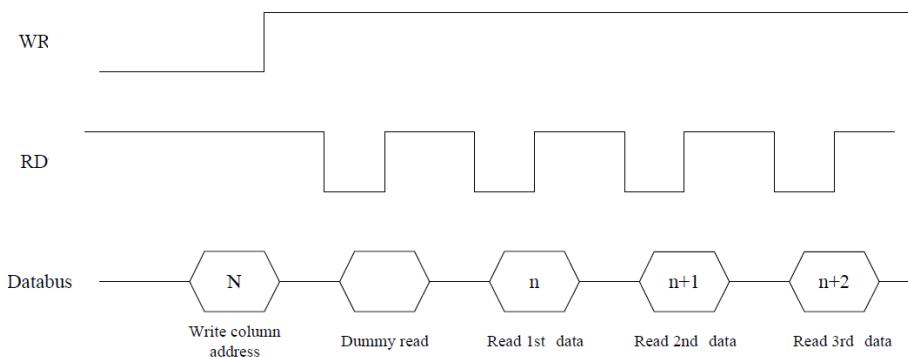
SSD1306 的 8080 接口方式下，控制脚的信号状态所对应的功能如表 17.1.2：

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：



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：

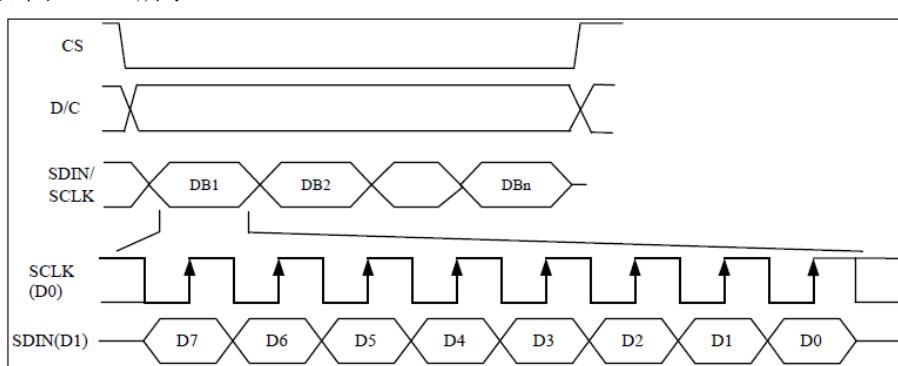


图 17.1.6 4 线 SPI 写操作时序图

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 (COL0~127)						
	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 4 线 SPI 模式/IIC 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 STM32F4 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 STM32F4 上的 GRAM (实际上就是 SRAM)，在修改完了之后，一次性把 STM32F4 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	设置列地址低四位	设置 8 位起始列地址的低四位
5	10~1F	0	0	0	0	X3	X2	X1	X0	设置列地址高四位	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:

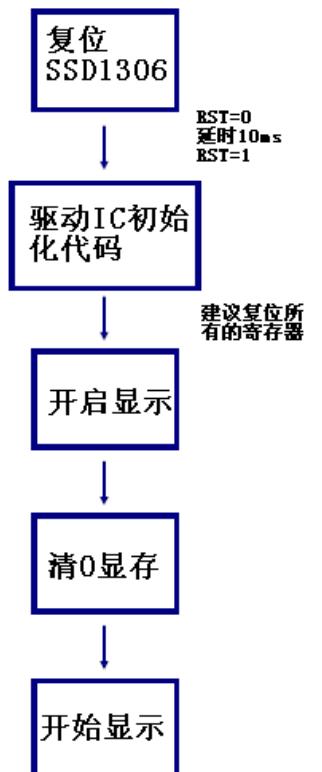


图 17.1.7 SSD1306 初始化框图

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 STM32F4 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) 通过函数将字符串和数字显示到 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 模块的电路在前面已有详细说明了，这里我们介绍 OLED 模块与探索者 STM32F4 开发板的连接，开发板底板的 LCD 接口和 ALIENTEK OLED 模块直接可以对插（**靠左插！**），连接如图 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 与探索者 STM32F4 开发板的 IO 口对应关系如下：

- OLED_CS 对应 DCMI_VSYNC，即：PB7;
- OLED_RS 对应 DCMI_SCL，即：PD6;
- OLED_WR 对应 DCMI_HREF，即：PA4;
- OLED_RD 对应 DCMI_SDA，即：PD7;
- OLED_RST 对应 DCMI_RESET，即：PG15;
- OLED_D[7:0] 对应 DCMI_D[7:0]，即：PE6/PE5/PB6/PC11/PC9/PC8/PC7/PC6;

这些线的连接，开发板的内部已经连接好了，我们只需要将 OLED 模块插上去就好了，注意，这里的 OLED_D[7:0] 因为不是接的连续的 IO，所以得用拼凑的方式去组合一下，后续会介绍。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

本实验，我们新建了 oled.c 和 oled.h 文件。这两个文件用来存放 OLED 相关的驱动函数以

及文件申明等。

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，所以我们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_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOB
                           |RCC_AHB1Periph_GPIOC|RCC_AHB1Periph_GPIOD|RCC_AHB1Periph_GPIOE
                           |RCC_AHB1Periph_GPIOG, ENABLE); //使能 PORTA~E,PORTG 时钟
#if OLED_MODE==1           //使用 8080 并口模式

    //GPIO 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //普通输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7 ;
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化

    GPIO_InitStructure.GPIO_Pin =
        GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_11;
    GPIO_Init(GPIOC, &GPIO_InitStructure); //初始化

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7;
    GPIO_Init(GPIOD, &GPIO_InitStructure); //初始化

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_5;
    GPIO_Init(GPIOE, &GPIO_InitStructure); //初始化

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
    GPIO_Init(GPIOG, &GPIO_InitStructure); //初始化
    OLED_WR=1;
    OLED_RD=1;
#else
    //使用 4 线 SPI 串口模式

```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7;
GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_Init(GPIOD, &GPIO_InitStructure);//初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
GPIO_Init(GPIOG, &GPIO_InitStructure);//初始化  OLED_SDIN=1;
OLED_SCLK=1;
#endif
OLED_CS=1;  OLED_RS=1;    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();
}

```

接着,要介绍的是 OLED_Refresh_Gram 函数。我们在 STM32F4 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 STM32F4 内部的 GRAM 就可以了,然后通过 OLED_Refresh_Gram 函数把 GRAM 一次刷新到 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 STM32F4 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
//通过拼凑的方法向 OLED 输出一个 8 位数据
//data:要输出的数据
void OLED_Data_Out(u8 data)
{
    u16 dat=data&0X0F;
    GPIOC->ODR&=~(0XF<<6);//清空 6~9
    GPIOC->ODR|=dat<<6; //D[3:0]-->PC[9:6]
    PCout(11)=(data>>4)&0X01;//D4
    PBout(6)=(data>>5)&0X01; //D5
    PEout(5)=(data>>6)&0X01; //D6
    PEout(6)=(data>>7)&0X01; //D7
}
//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)
{
    OLED_Data_Out(dat);
    OLED_RS=cmd;
    OLED_CS=0;    OLED_WR=0;
    OLED_WR=1;    OLED_CS=1;    OLED_RS=1;
}
#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;
    OLED_RS=cmd; //写命令
    OLED_CS=0;
    for(i=0;i<8;i++)
    {
        OLED_SCLK=0;
        if(dat&0x80)OLED_SDIN=1;
        else OLED_SDIN=0;
        OLED_SCLK=1;dat<<=1;
    }
    OLED_CS=1; OLED_RS=1;
}
#endif

```

首先，我们看 OLED_Data_Out 函数，这就是我们前面说的，因为 OLED 的 D0~D7 不是接的连续 IO，所以必须将数据，拆分到各个 IO，以实现一次完整的数据传输，该函数就是根据我们 OLED_D[7:0]具体连接的 IO，对数据进行拆分，然后输出给对应位的各个 IO，实现并口数据输出。这种方式会降低并口速度，但是我们 OLED 模块，是单色的，数据量不是很大，所以这种方式也不会造成视觉上的影响，大家可以放心使用，但是如果 TFTLCD，就不推荐了。

然后，看 OLED_WR_Byte 函数，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。

接下来，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：

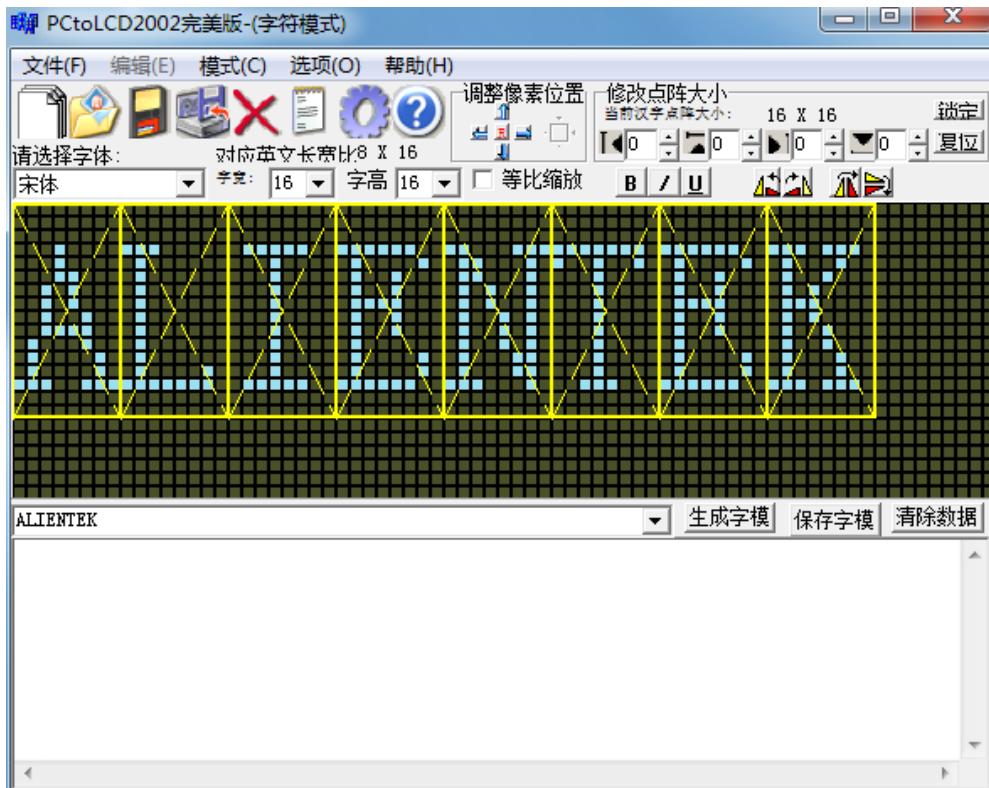


图 17.3.1 PCtoLCD2002 软件界面

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：

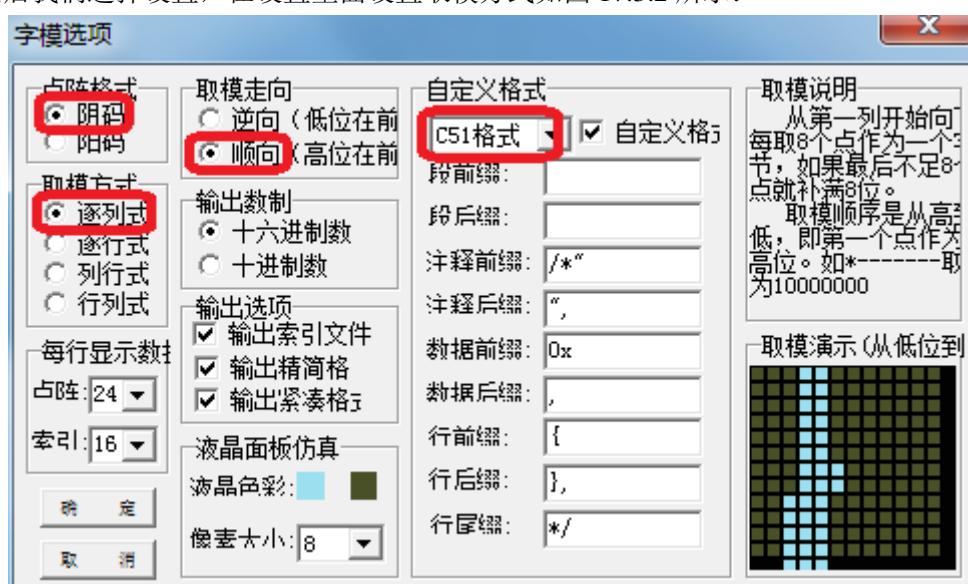


图 17.3.2 设置取模方式

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：

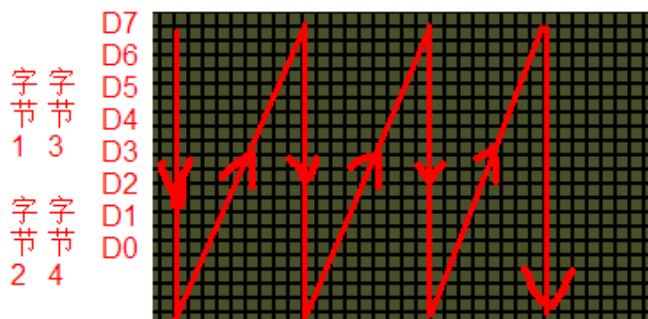


图 17.3.3 取模方式图解

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、16*8 和 24*12 大小取模出来（对应汉字大小为 12*12、16*16 和 24*24，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，每个 24*12 的字符占用 36 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：

```
//在指定位置显示一个字符,包括部分字符
//x:0~127 y:0~63
//mode:0,反白显示;1,正常显示
//size:选择字体 12/16/24
void OLED_ShowChar(u8 x,u8 y,u8 chr,u8 size,u8 mode)
{
    u8 temp,t,t1,y0=y;
```

```

u8 cszie=(size/8+((size%8)?1:0))*(size/2); //得到字体一个字符对应点阵集所占的字节数
chr=chr-' '; //得到偏移后的值
for(t=0;t<cszie;t++)
{
    if(size==12)temp=asc2_1206[chr][t]; //调用 1206 字体
    else if(size==16)temp=asc2_1608[chr][t]; //调用 1608 字体
    else if(size==24)temp=asc2_2412[chr][t]; //调用 2412 字体
    else return; //没有的字库
    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(先 `y++`)，从左到右(再 `x++`)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，接下来我们看看 `oled.h` 代码：

```

#ifndef __OLED_H
#define __OLED_H
#include "sys.h"
#include "stdlib.h"
//OLED 模式设置
//0: 4 线串行模式 (模块的 BS1, BS2 均接 GND)
//1: 并行 8080 模式 (模块的 BS1, BS2 均接 VCC)
#define OLED_MODE 1
//-----OLED 端口定义-----
#define OLED_CS      PBout(7)
#define OLED_RST     PGout(15)
#define OLED_RS      PDout(6)
#define OLED_WR      PAout(4)
#define OLED_RD      PDout(7)
//使用 4 线串行接口时使用
#define OLED_SCLK   PCout(6)

```

```
#define OLED_SDIN  PCout(7)
#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);
....          //忽略部分函数声明
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 BS1 和 BS2 的设置（目前代码仅支持 8080 和 4 线 SPI）来确定 OLED_MODE 的值。

最后我们来看看主函数代码：

```
int main(void)
{
    u8 t=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    OLED_Init(); //初始化 OLED
    OLED_ShowString(0,0,"ALIENTEK",24);
    OLED_ShowString(0,24, "0.96' OLED TEST",16);
    OLED_ShowString(0,40, "ATOM 2014/5/4",12);
    OLED_ShowString(0,52,"ASCII:",12);
    OLED_ShowString(64,52,"CODE:",12);
    OLED_Refresh_Gram(); //更新显示到 OLED
    t=' ';
    while(1)
    {
        OLED_ShowChar(36,52,t,12,1); //显示 ASCII 字符
        OLED_ShowNum(94,52,t,3,12); //显示 ASCII 字符的码值
        OLED_Refresh_Gram(); //更新显示到 OLED
        t++;
        if(t> '~')t=' ';
        delay_ms(500); 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。然后我们编译此工程，直到编译成功为止。

17.4 下载验证

将代码下载到开发板后，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：

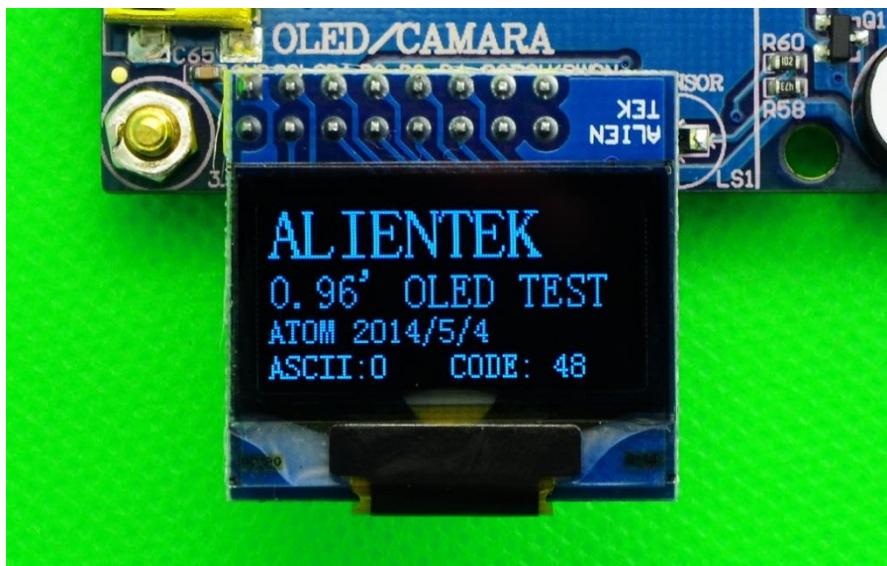


图 17.4.1 OLED 显示效果

图中 OLED 显示了三种尺寸的字符：24*12（ALIENTEK）、16*8（0.96’ OLED TEST）和 12*6（剩下的内容）。说明我们的实验是成功的，实现了三种不同尺寸 ASCII 字符的显示，在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，我们将使用探索者 STM32F4 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 & FSMC 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 STM32F4 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' /4.3' /7' 5 种大小的屏幕可选。
- 2, 320×240 的分辨率 (3.5' 分辨率为:320*480, 4.3' 和 7' 分辨率为: 800*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(其他 3.5 寸/4.3 寸等 LCD 方法类似，请参考 2.8 的即可)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：

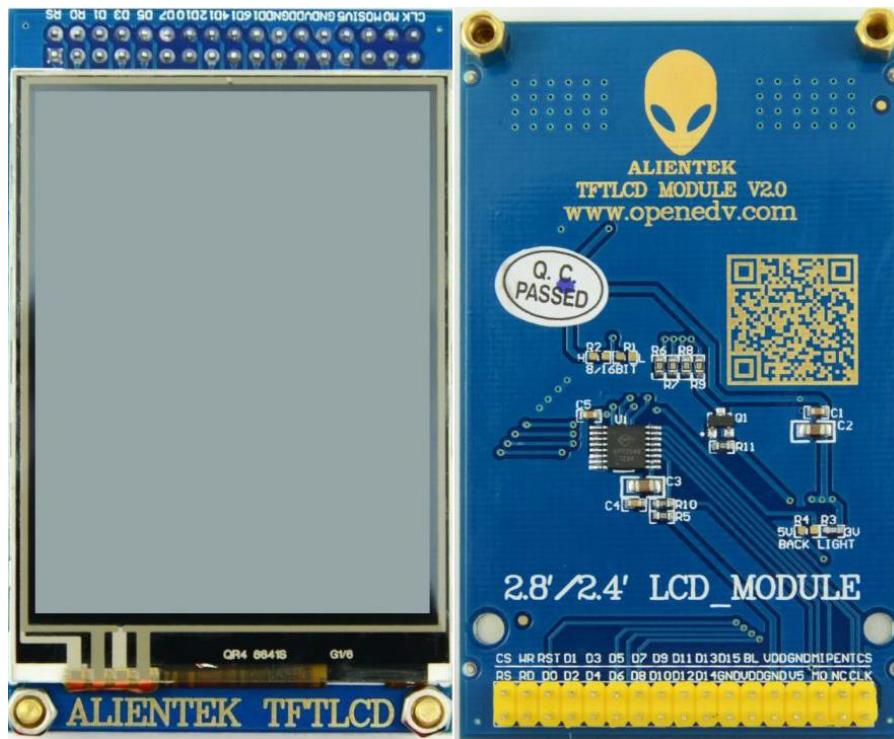


图 18.1.1.1 ALIENTEK 2.8 寸 TFTLCD 外观图

模块原理图如图 18.1.1.2 所示：

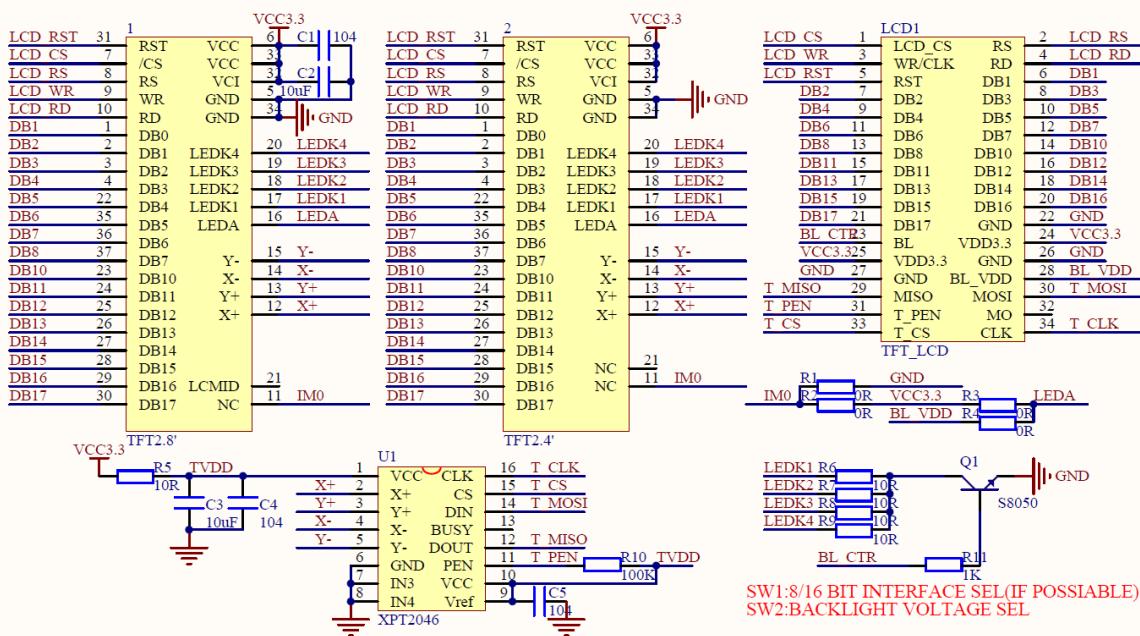


图 18.1.1.2 ALIENTEK 2.8 寸 TFTLCD 模块原理图

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：

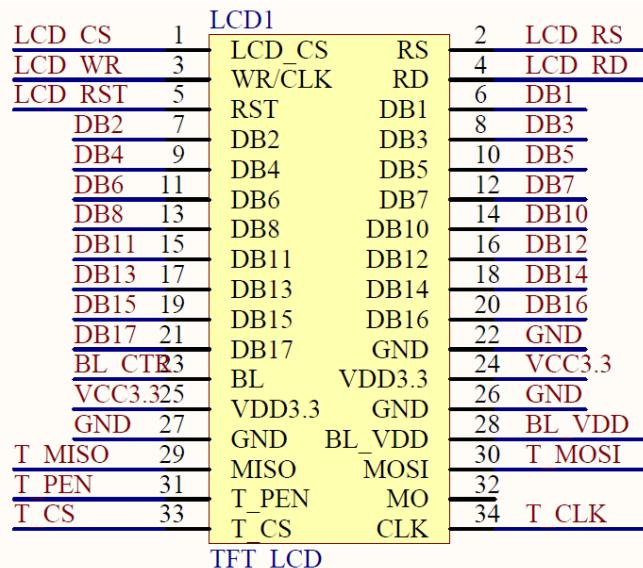


图 18.1.1.3 ALIENTEK 2.8寸 TFTLCD 模块接口图

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 18.1.1.3 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 STM32F4 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，其驱动芯片有很多类型，比如有：ILI9341/ILI9325/RM68042/RM68021/ILI9320/ILI9328/LGDP4531/LGDP4535/SPFD5408/SSD1289/1505/B505/C505/NT35310/NT35510 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 ILI9341 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。

ILI9341 液晶控制器自带显存，其显存总大小为 172800 (240*320*18/8)，即 18 位模式（26 万色）下的显存量。在 16 位模式下，ILI9341 采用 RGB565 格式存储颜色数据，此时 ILI9341 的 18 位数据线与 MCU 的 16 位数据线以及 LCD GRAM 的对应关系如图 18.1.1.4 所示：

9341总线	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MCU数据 (16位)	D15	D14	D13	D12	D11	NC	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	NC
LCD GRAM (16位)	R[4]	R[3]	R[2]	R[1]	R[0]	NC	G[5]	G[4]	G[3]	G[2]	G[1]	G[0]	B[4]	B[3]	B[2]	B[1]	B[0]	NC

图 18.1.1.4 16 位数据与显存对应关系图

从图中可以看出, ILI9341 在 16 位模式下面, 数据线有用的是: D17~D13 和 D11~D1, D0 和 D12 没有用到, 实际上在我们 LCD 模块里面, ILI9341 的 D0 和 D12 压根就没有引出来, 这样, ILI9341 的 D17~D13 和 D11~D1 对应 MCU 的 D15~D0。

这样 MCU 的 16 位数据, 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。另外, 特别注意 ILI9341 所有的指令都是 8 位的 (高 8 位无效), 且参数除了读写 GRAM 的时候是 16 位, 其他操作参数, 都是 8 位的, 这个和 ILI9320 等驱动器不一样, 必须加以注意。

接下来, 我们介绍一下 ILI9341 的几个重要命令, 因为 ILI9341 的命令很多, 我们这里就不全部介绍了, 有兴趣的大家可以找到 ILI9341 的 datasheet 看看。里面对这些命令有详细的介绍。我们将介绍: 0XD3, 0X36, 0X2A, 0X2B, 0X2C, 0X2E 等 6 条指令。

首先来看指令: 0XD3, 这个是读 ID4 指令, 用于读取 LCD 控制器的 ID, 该指令如表 18.1.1.1 所示:

顺序	控制			各位描述									HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	1	1	0	1	0	0	1	1	D3H
参数 1	1	↑	1	XX	X	X	X	X	X	X	X	X	X
参数 2	1	↑	1	XX	0	0	0	0	0	0	0	0	00H
参数 3	1	↑	1	XX	1	0	0	1	0	0	1	1	93H
参数 4	1	↑	1	XX	0	1	0	0	0	0	0	1	41H

表 18.1.1.1 0XD3 指令描述

从上表可以看出, 0XD3 指令后面跟了 4 个参数, 最后 2 个参数, 读出来是 0X93 和 0X41, 刚好是我们控制器 ILI9341 的数字部分, 从而, 通过该指令, 即可判别所用的 LCD 驱动器是什么型号, 这样, 我们的代码, 就可以根据控制器的型号去执行对应驱动 IC 的初始化代码, 从而兼容不同驱动 IC 的屏, 使得一个代码支持多款 LCD。

接下来看指令: 0X36, 这是存储访问控制指令, 可以控制 ILI9341 存储器的读写方向, 简单的说, 就是在连续写 GRAM 的时候, 可以控制 GRAM 指针的增长方向, 从而控制显示方式 (读 GRAM 也是一样)。该指令如表 18.1.1.2 所示:

顺序	控制			各位描述									HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	1	0	1	1	0	36H
参数	1	1	↑	XX	MY	MX	MV	ML	BGR	MH	0	0	0

表 18.1.1.2 0X36 指令描述

从上表可以看出, 0X36 指令后面, 紧跟一个参数, 这里我们主要关注: MY、MX、MV 这三个位, 通过这三个位的设置, 我们可以控制整个 ILI9341 的全部扫描方向, 如表 18.1.1.3 所示:

控制位			效果			
MY	MX	MV	LCD 扫描方向 (GRAM 自增方式)			
0	0	0	从左到右, 从上到下			
1	0	0	从左到右, 从下到上			
0	1	0	从右到左, 从上到下			
1	1	0	从右到左, 从下到上			

0	0	1	从上到下, 从左到右
0	1	1	从上到下, 从右到左
1	0	1	从下到上, 从左到右
1	1	1	从下到上, 从右到左

表 18.1.1.3 MY、MX、MV 设置与 LCD 扫描方向关系表

这样, 我们在利用 ILI9341 显示内容的时候, 就有很大灵活性了, 比如显示 BMP 图片, BMP 解码数据, 就是从图片的左下角开始, 慢慢显示到右上角, 如果设置 LCD 扫描方向为从左到右, 从下到上, 那么我们只需要设置一次坐标, 然后就不停的往 LCD 填充颜色数据即可, 这样可以大大提高显示速度。

接下来看指令: 0X2A, 这是列地址设置指令, 在从左到右, 从上到下的扫描方式(默认)下面, 该指令用于设置横坐标(x坐标), 该指令如表 18.1.1.4 所示:

顺序	控制			各位描述									HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	0	1	0	1	0	2AH
参数 1	1	1	↑	XX	SC15	SC14	SC13	SC12	SC11	SC10	SC9	SC8	
参数 2	1	1	↑	XX	SC7	SC6	SC5	SC4	SC3	SC2	SC1	SC0	SC
参数 3	1	1	↑	XX	EC15	EC14	EC13	EC12	EC11	EC10	EC9	EC8	
参数 4	1	1	↑	XX	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0	EC

表 18.1.1.4 0X2A 指令描述

在默认扫描方式时, 该指令用于设置 x 坐标, 该指令带有 4 个参数, 实际上是 2 个坐标值: SC 和 EC, 即列地址的起始值和结束值, SC 必须小于等于 EC, 且 $0 \leq SC/EC \leq 239$ 。一般在设置 x 坐标的时候, 我们只需要带 2 个参数即可, 也就是设置 SC 即可, 因为如果 EC 没有变化, 我们只需要设置一次即可(在初始化 ILI9341 的时候设置), 从而提高速度。

与 0X2A 指令类似, 指令: 0X2B, 是页地址设置指令, 在从左到右, 从上到下的扫描方式(默认)下面, 该指令用于设置纵坐标(y坐标)。该指令如表 18.1.1.5 所示:

顺序	控制			各位描述									HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	0	1	0	1	0	2BH
参数 1	1	1	↑	XX	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	
参数 2	1	1	↑	XX	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SP
参数 3	1	1	↑	XX	EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	
参数 4	1	1	↑	XX	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0	EP

表 18.1.1.5 0X2B 指令描述

在默认扫描方式时, 该指令用于设置 y 坐标, 该指令带有 4 个参数, 实际上是 2 个坐标值: SP 和 EP, 即页地址的起始值和结束值, SP 必须小于等于 EP, 且 $0 \leq SP/EP \leq 319$ 。一般在设置 y 坐标的时候, 我们只需要带 2 个参数即可, 也就是设置 SP 即可, 因为如果 EP 没有变化, 我们只需要设置一次即可(在初始化 ILI9341 的时候设置), 从而提高速度。

接下来看指令: 0X2C, 该指令是写 GRAM 指令, 在发送该指令之后, 我们便可以往 LCD 的 GRAM 里面写入颜色数据了, 该指令支持连续写, 指令描述如表 18.1.1.6 所示:

顺序	控制			各位描述									HEX
	RS	RD	WR	D15~D8	D7	D6	D5	D4	D3	D2	D1	D0	
指令	0	1	↑	XX	0	0	1	0	1	1	0	0	2CH

参数 1	1	1	↑	D1[15: 0]										XX
.....	1	1	↑	D2[15: 0]										XX
参数 n	1	1	↑	Dn[15: 0]										XX

表 18.1.1.6 0X2C 指令描述

从上表可知，在收到指令 0X2C 之后，数据有效位宽变为 16 位，我们可以连续写入 LCD GRAM 值，而 GRAM 的地址将根据 MY/MX/MV 设置的扫描方向进行自增。例如：假设设置的是从左到右，从上到下的扫描方式，那么设置好起始坐标（通过 SC, SP 设置）后，每写入一个颜色值，GRAM 地址将会自动自增 1 (SC++), 如果碰到 EC，则回到 SC，同时 SP++, 一直到坐标：EC, EP 结束，其间无需再次设置的坐标，从而大大提高写入速度。

最后，来看看指令：0X2E，该指令是读 GRAM 指令，用于读取 ILI9341 的显存 (GRAM)，该指令在 ILI9341 的数据手册上面的描述是有误的，真实的输出情况如表 18.1.1.7 所示：

顺序	控制			各位描述											HEX		
	RS	RD	WR	D15~D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0		
指令	0	1	↑	XX				0	0	1	0	1	1	1	0	2EH	
参数 1	1	↑	1	XX				G1[5:0]				XX				dummy	
参数 2	1	↑	1	R1[4:0]	XX				G1[5:0]				XX				R1G1
参数 3	1	↑	1	B1[4:0]	XX				R2[4:0]				XX				B1R2
参数 4	1	↑	1	G2[5:0]	XX				B2[4:0]				XX				G2B2
参数 5	1	↑	1	R3[4:0]	XX				G3[5:0]				XX				R3G3
参数 N	1	↑	1	按以上规律输出													

表 18.1.1.7 0X2E 指令描述

该指令用于读取 GRAM，如表 18.1.1.7 所示，ILI9341 在收到该指令后，第一次输出的是 dummy 数据，也就是无效的数据，第二次开始，读取到的才是有效的 GRAM 数据（从坐标：SC, SP 开始），输出规律为：每个颜色分量占 8 个位，一次输出 2 个颜色分量。比如：第一次输出是 R1G1，随后的规律为：B1R2→G2B2→R3G3→B3R4→G4B4→R5G5... 以此类推。如果我们只需要读取一个点的颜色值，那么只需要接收到参数 3 即可，如果要连续读取（利用 GRAM 地址自增，方法同上），那么就按照上述规律去接收颜色数据。

以上，就是操作 ILI9341 常用的几个指令，通过这几个指令，我们便可以很好的控制 ILI9341 显示我们所要显示的内容了。

一般 TFTLCD 模块的使用流程如图 18.1.1.5：

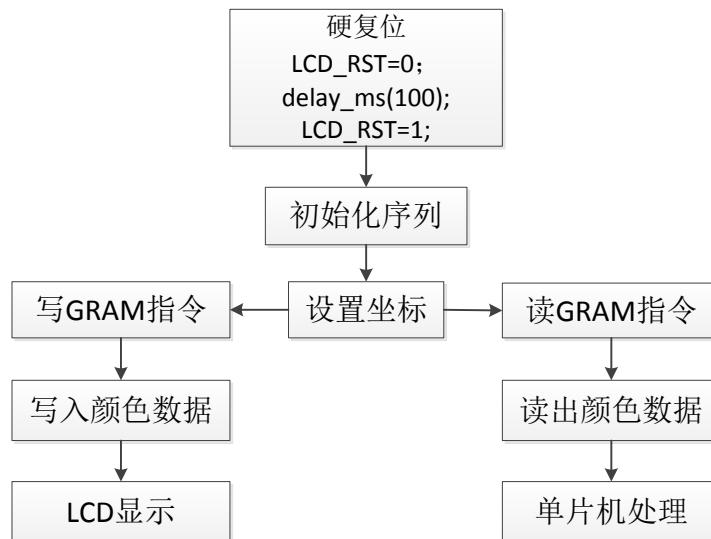


图 18.1.1.5 TFTLCD 使用流程

任何 LCD，使用流程都可以简单的用以上流程图表示。其中硬复位和初始化序列，只需要执行一次即可。而画点流程就是：设置坐标→写 GRAM 指令→写入颜色数据，然后在 LCD 上面，我们就可以看到对应的点显示我们写入的颜色了。读点流程为：设置坐标→读 GRAM 指令→读取颜色数据，这样就可以获取到对应点的颜色数据了。

以上只是最简单的操作，也是最常用的操作，有了这些操作，一般就可以正常使用 TFTLCD 了。接下来我们将该模块用来来显示字符和数字，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 STM32F4 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。

即图 18.1.1.5 的初始化序列，这里我们没有硬复位 LCD，因为探索者 STM32F4 开发板的 LCD 接口，将 TFTLCD 的 RST 同 STM32F4 的 RESET 连接在一起了，只要按下开发板的 RESET 键，就会对 LCD 进行硬复位。初始化序列，就是向 LCD 控制器写入一系列的设置值（比如伽马校准），这些初始化序列一般 LCD 供应商会提供给客户，我们直接使用这些序列即可，不需要深入研究。在初始化之后，LCD 才可以正常使用。

3) 通过函数将字符和数字显示到 TFTLCD 模块上。

这一步则通过图 18.1.1.5 左侧的流程，即：设置坐标→写 GRAM 指令→写 GRAM 来实现，但是这个步骤，只是一个点的处理，我们要显示字符/数字，就必须要多次使用这个步骤，从而达到显示字符/数字的目的，所以需要设计一个函数来实现数字/字符的显示，之后调用该函数，就可以实现数字/字符的显示了。

18.1.2 FSMC 简介

STM32F407 或 STM32F417 系列芯片都带有 FSMC 接口，ALIENTEK 探索者 STM32F4 开发板的主芯片为 STM32F407ZGT6，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，STM32F4 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：

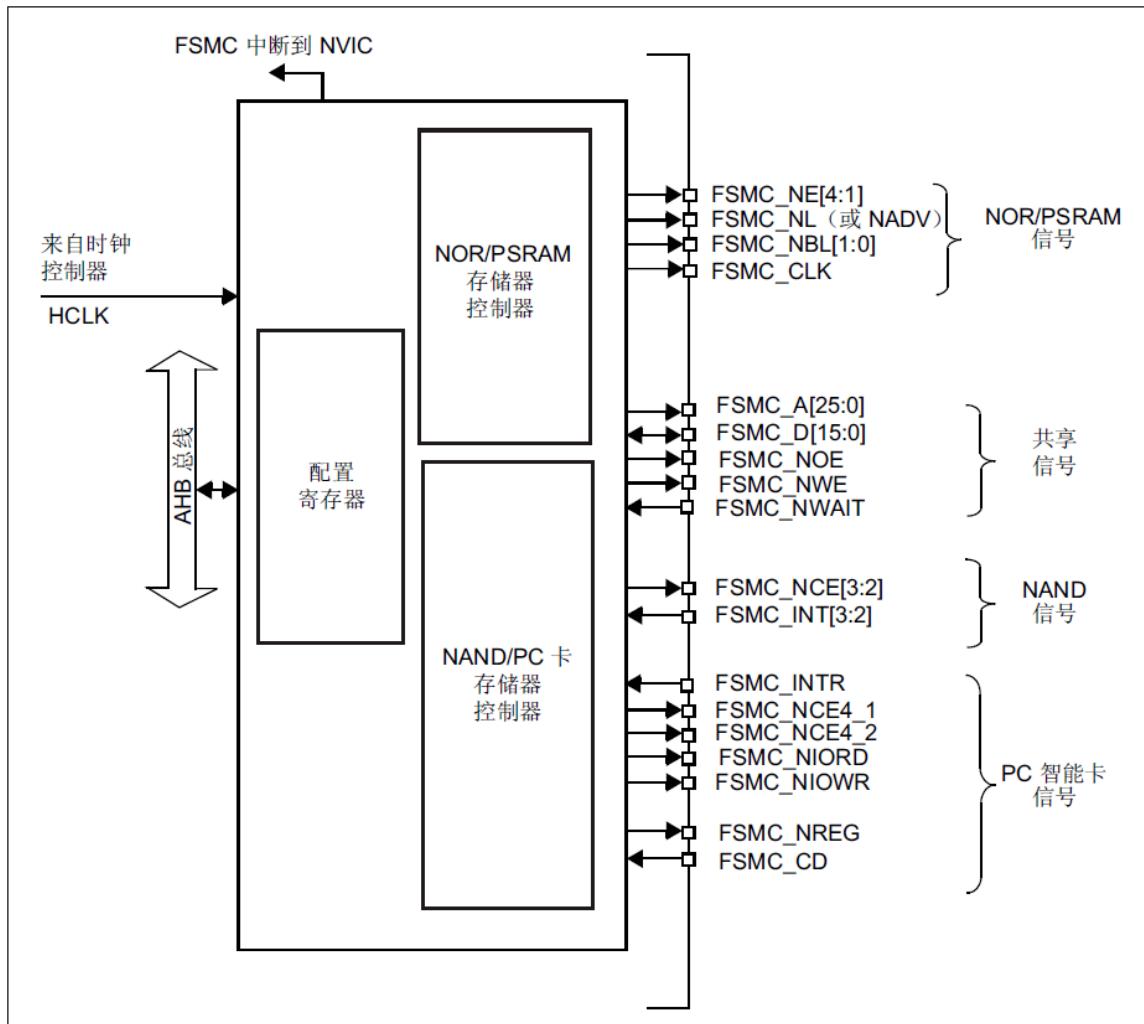


图 18.1.2.1 FSMC 框图

从上图我们可以看出，STM32F4 的 FSMC 将外部设备分为 2 类：NOR/PSRAM 设备、NAND/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，探索者 STM32F4 开发板是把 RS 连接在 A6 上面的。

STM32F4 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，STM32F4

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：

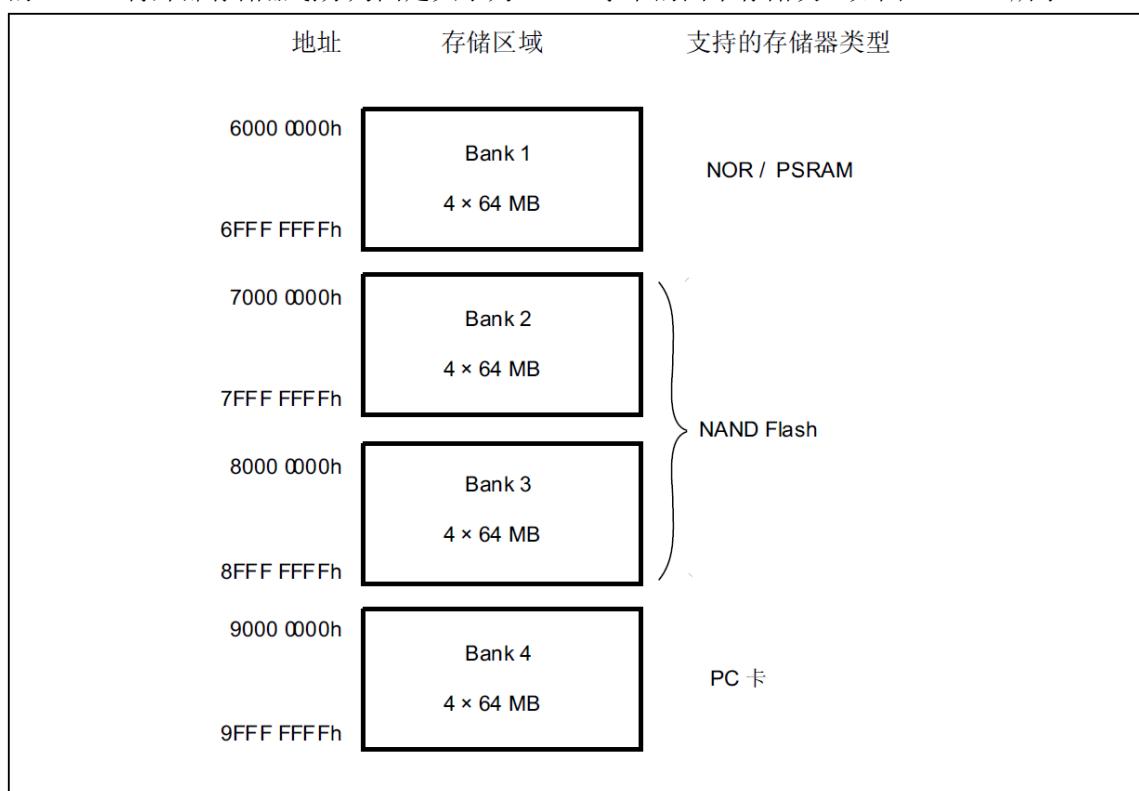


图 18.1.2.2 FSMC 存储块地址映像

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《STM32F4xx 中文参考手册》第 32 章（1191 页）的相关介绍。

STM32F4 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。STM32F4 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 FSMC_PMEM2/3/4 FSMC_PATT2/3/4
	Bank3	0X8000, 0000~0X8FFF, FFFF		FSMC_PI04
	Bank4	0X9000, 0000~0X9FFF, FFFF	PC Card	FSMC_ECCR2/3

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
	ModeA	SRAM/CRAM OE 选通型时序
	Mode2/B	NOR FLASH 时序
	ModeC	NOR FLASH OE 选通型时序
	ModeD	延长地址保持时间的异步时序
同步突发	根据同步时钟 FSMC_CK 读取 多个顺序单元的数据	CLKDIV、DATLAT

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：

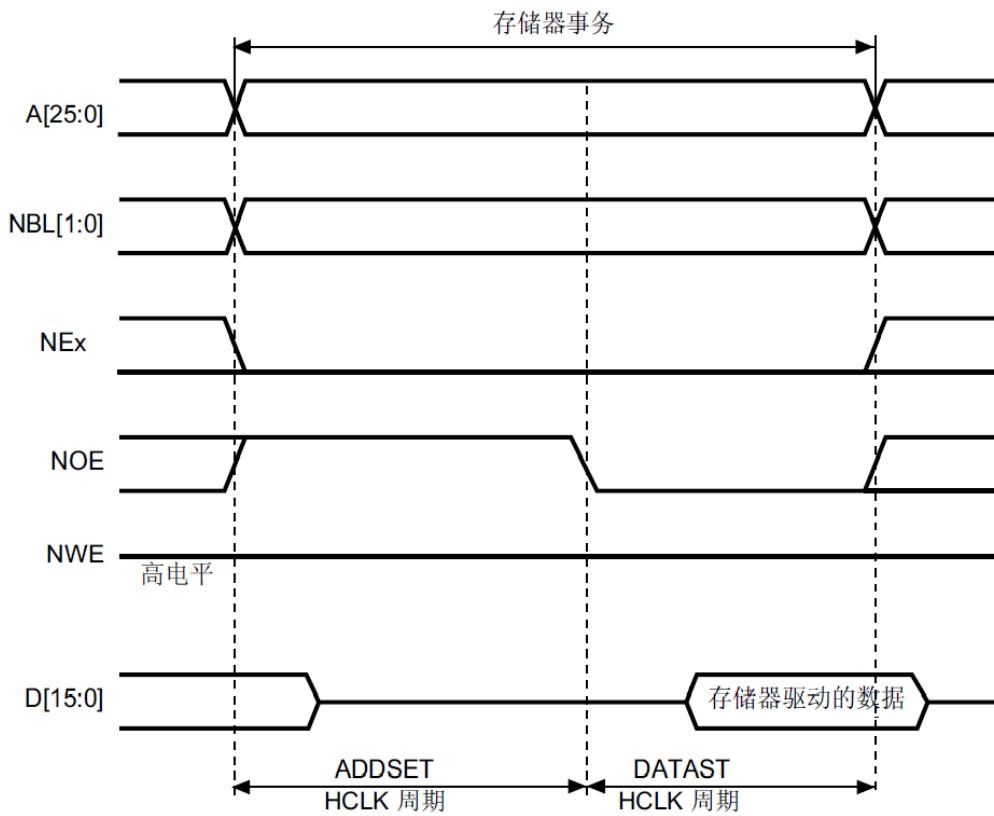


图 18.1.2.3 模式 A 读操作时序图

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：

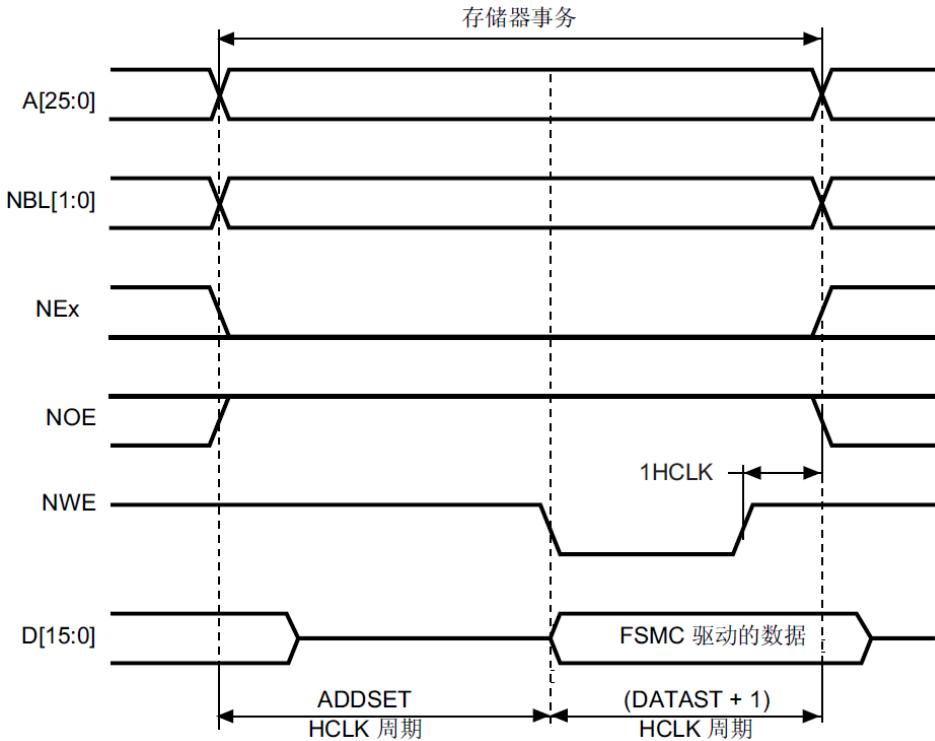


图 18.1.2.3 和图 18.1.2.4 中的 ADDSET 与 DATAST，是通过不同的寄存器设置的，接下来我们讲解一下 Bank1 的几个控制寄存器

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
Reserved																															

CBURSTRW Reserved ASCYCWAIT EXTMOD WAITEN WREN WAITCFG WRAPMOD WAITPOL BURSTEN Reserved FACCEN MWID MTYP MUXEN MBKEN

rw rw

图 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：

	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
Reserved	ACCMOD		DATLAT				CLKDIV				BUSTURN				DATAST								ADDHLD				ADDSET					
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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 个 HCLK 时钟周期, 最大为 255 个 HCLK 周期。对 ILI9341 来说, 其实就是 RD 低电平持续时间, 一般为 355ns。而一个 HCLK 时钟周期为 6ns 左右 (1/168Mhz), 为了兼容其他屏, 我们这里设置 DATAST 为 60, 也就是 60 个 HCLK 周期, 时间大约是 360ns。

ADDSET[3:0]: 地址建立时间。其建立时间为: ADDSET 个 HCLK 周期, 最大为 15 个 HCLK 周期。对 ILI9341 来说, 这里相当于 RD 高电平持续时间, 为 90ns, 我们设置 ADDSET 为 15, 即 $15 \times 6 = 90$ ns。

最后, 我们再来看看 SRAM/NOR 闪写时序寄存器: FSMC_BWTRx ($x=1\sim 4$), 该寄存器各位描述如图 18.1.2.7 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
Res.	ACCMOD		DATLAT				CLKDIV				BUSTURN				DATAST								ADDHLD				ADDSET					
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	

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 ILI9341 来说, 这两个时间只需要 15ns 就够了, 比读操作快得多。所以我们这里设置 DATAST 为 2, 即 3 个 HCLK 周期, 时间约为 18ns。然后 ADDSET 设置为 3, 即 3 个 HCLK 周期, 时间为 18ns。

至此, 我们对 STM32F4 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_BCRx 和 FSMC_BTRx, 组合成 BTCSR[8]寄存器组, 他们的对应关系如下:

BTCSR[0]对应 FSMC_BCR1, BTCSR[1]对应 FSMC_BTR1

BTCSR[2]对应 FSMC_BCR2, BTCSR[3]对应 FSMC_BTR2

BTCSR[4]对应 FSMC_BCR3, BTCSR[5]对应 FSMC_BTR3

BTCSR[6]对应 FSMC_BCR4, BTCSR[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

SMC_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`，所以使用的是第一个函数。

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 ALIENTEK 探索者 STM32F4 开发板的连接，探索者 STM32F4 开发板底板的 LCD 接口和 ALIENTEK TFTLCD 模块直接可以对插，连接关系如图 18.2.1 所示：

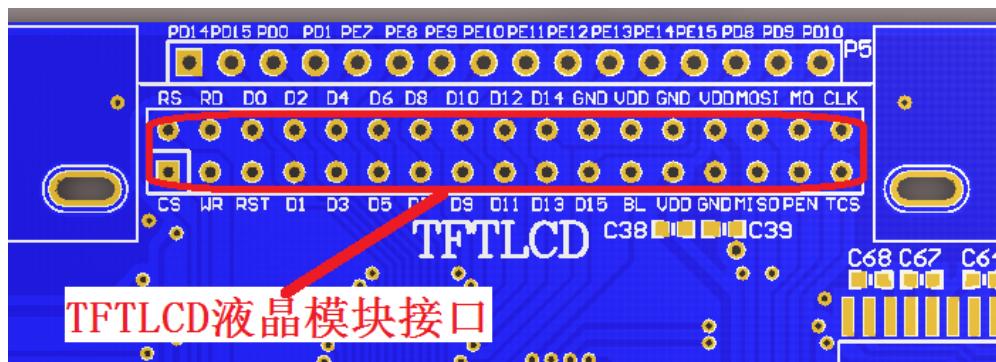


图 18.2.1 TFTLCD 与开发板连接示意图

图 18.2.1 中圈出来的部分就是连接 TFTLCD 模块的接口，液晶模块直接插上去即可。

在硬件上，TFTLCD 模块与探索者 STM32F4 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 PF12 即 FSMC_A6;
- 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，探索者 STM32F4 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 stm32f4xx_fsmc.c 文件和头文件 stm32f4xx_fsmc.h 中。所以我们在工程中要引入 stm32f4xx_fsmc.c 源文件。

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-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 A6 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 A6 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 | 0x0000007E))
#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 0X0000007E，则是 A6 的偏移量，这里很多朋友不理解这个偏移量的概念，简单说明下：以 A6 为例，7E 转换成二进制就是：1111110，而 16 位数据时，地址右移一位对齐，那么实际对应到地址引脚的时候，就是：A6:A0=0111111，此时 A6 是 0，但是如

果 16 位地址再加 1 (注意: 对应到 8 位地址是加 2, 即 $7E+0X02$), 那么: A6:A0=1000000, 此时 A6 就是 1 了, 即实现了对 RS 的 0 和 1 的控制。

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 0X6C00,007E, 对应 A6 的状态为 0(即 RS=0), 而 LCD-> LCD_RAM 的地址就是 0X6C00,0080 (结构体地址自增), 对应 A6 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。
```

```
    u16 wramcmd;         //开始写 gram 指令
```

```
    u16 setxcmd;          //设置 x 坐标指令
```

```
    u16 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十几个字节的内存, 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 7 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(vu16 regval)
```

```
{   regval=regval;      //使用-O2 优化的时候,必须插入的延时
```

```
    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(vu16 data)
```

```
{   data=data;          //使用-O2 优化的时候,必须插入的延时
```

```
    LCD->LCD_RAM=data;
```

```
}
```

```
//读 LCD 数据
//返回值:读到的值
u16 LCD_RD_DATA(void)
{
    vu16 ram;           //防止被优化
    ram=LCD->LCD_RAM;
    return ram;
}

//写寄存器
//LCD_Reg:寄存器地址
//LCD_RegValue:要写入的数据
void LCD_WriteReg(vu16 LCD_Reg, vu16 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(vu16 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;
}

//LCD 写 GRAM
//RGB_Code:颜色值
void LCD_WriteRAM(u16 RGB_Code)
{
    LCD->LCD_RAM = RGB_Code;//写十六位 GRAM
}
```

因为 FSMC 自动控制了 WR/RD/CS 等这些信号，所以这 7 个函数实现起来都非常简单，我们就不多说，注意，上面有几个函数，我们添加了一些对 MDK - O2 优化的支持，去掉的话，在-O2 优化的时候会出问题。这些函数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||lcddev.id==0X5310)
    {
```

```
LCD_WR_REG(lcddev.setxcmd);
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);
}else if(lcddev.id==0X6804)
{
    if(lcddev.dir==1)Xpos=lcddev.width-1-Xpos;//横屏时处理
    LCD_WR_REG(lcddev.setxcmd);
    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);
}else if(lcddev.id==0X5510)
{
    LCD_WR_REG(lcddev.setxcmd);
    LCD_WR_DATA(Xpos>>8);
    LCD_WR_REG(lcddev.setxcmd+1);
    LCD_WR_DATA(Xpos&0XFF);
    LCD_WR_REG(lcddev.setycmd);
    LCD_WR_DATA(Ypos>>8);
    LCD_WR_REG(lcddev.setycmd+1);
    LCD_WR_DATA(Ypos&0XFF);
}
}else
{
    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 9341/5310/6804/5510 等的设置同其他屏有些不太一样，所以进行了区别对待。

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_Prepare(); //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)
{
    vu16 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||lcddev.id==0X6804||lcddev.id==0X5310)LCD_WR_REG(0X2E);
    //9341/6804/3510 发送读 GRAM 指令
    else if(lcddev.id==0X5510)LCD_WR_REG(0X2E00); //5510 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
    LCD_RD_DATA();                                //dummy Read
    opt_delay(2);
    r=LCD_RD_DATA();                            //实际坐标颜色
    if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510)
    {   //9341/NT35310/NT35510 要分 2 次读出

        opt_delay(2);
        b=LCD_RD_DATA();
        g=r&0xFF;//9341/5310/5510 等,第一次读取的是 RG 的值,R 在前,G 在后,各占 8 位
        g<<=8;
    }
    if(lcddev.id==0X9325||lcddev.id==0X4535||lcddev.id==0X4531||lcddev.id==0XB505||
       lcddev.id==0XC505) return r; //这几种 IC 直接返回颜色值
    else if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510) return (((r>>11)<<11)
        |((g>>10)<<5)|(b>>11)); //ILI9341/NT35310/NT35510 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 12/16/24
//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;
    u8 csize=(size/8+((size%8)?1:0))*(size/2); //得到字体一个字符对应点阵集所占的字节数
    //设置窗口
    num=num-' '; //得到偏移后的值
    for(t=0;t<csize;t++)
    {
        if(size==12)temp=asc2_1206[num][t]; //调用 1206 字体
        else if(size==16)temp=asc2_1608[num][t]; //调用 1608 字体
        else if(size==24)temp=asc2_2412[num][t]; //调用 2412 字体
        else return; //没有的字库
        for(t1=0;t1<8;t1++)
        {
            if(temp&0x80)LCD_Fast_DrawPoint(x,y,POINT_COLOR);
            else if(mode==0)LCD_Fast_DrawPoint(x,y,BACK_COLOR);
            temp<<=1;
            y++;
            if(y>=lcddev.height)return; //超区域了
            if((y-y0)==size)
            {
                y=y0; x++;
                if(x>=lcddev.width)return; //超区域了
                break;
            }
        }
    }
}
```

在 LCD_ShowChar 函数里面，我们采用快速画点函数 LCD_Fast_DrawPoint 来画点显示字符，该函数同 LCD_DrawPoint 一样，只是带了颜色参数，且减少了函数调用的时间，详见本例

程源码。该代码中我们用到了三个字符集点阵数据数组 `asc2_2412`、`asc2_1206` 和 `asc2_1608`，这几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：

```
void LCD_Init(void)
{
    vu32 i=0;
    GPIO_InitTypeDef  GPIO_InitStructure;
    FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
    FSMC_NORSRAMTimingInitTypeDef  readWriteTiming;
    FSMC_NORSRAMTimingInitTypeDef  writeTiming;

    //① GPIO,FSMC 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOD
                           |RCC_AHB1Periph_GPIOE|RCC_AHB1Periph_GPIOF|RCC_AHB1Periph_GPIOG,
                           ENABLE); //使能 PD,PE,PF,PG 时钟
    RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC,ENABLE); //使能 FSMC 时钟

    //② GPIO 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15; //PB15 推挽输出,控制背光
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //普通输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化 //PB15 推挽输出,控制背光

    GPIO_InitStructure.GPIO_Pin = (3<<0)|(3<<4)|(7<<8)|(3<<14);
    //PD0,1,4,5,8,9,10,14,15 AF OUT
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOD, &GPIO_InitStructure); //初始化

    GPIO_InitStructure.GPIO_Pin = (0X1FF<<7); //PE7~15,AF OUT
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOE, &GPIO_InitStructure); //初始化
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;//PF12,FSMC_A6  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉  
GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;//PF12,FSMC_A6  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉  
GPIO_Init(GPIOG, &GPIO_InitStructure);//初始化
```

//③ 引脚复用映射设置

```
GPIO_PinAFConfig(GPIOD,GPIO_PinSource0,GPIO_AF_FSMC);//PD0,AF12  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource1,GPIO_AF_FSMC);//PD1,AF12  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource4,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource5,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource8,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource9,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource10,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource14,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOD,GPIO_PinSource15,GPIO_AF_FSMC);//PD15,AF12
```

```
GPIO_PinAFConfig(GPIOE,GPIO_PinSource7,GPIO_AF_FSMC);//PE7,AF12  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource8,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource9,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource10,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource11,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource12,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource13,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource14,GPIO_AF_FSMC);  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource15,GPIO_AF_FSMC);//PE15,AF12
```

```
GPIO_PinAFConfig(GPIOF,GPIO_PinSource12,GPIO_AF_FSMC);//PF12,AF12  
GPIO_PinAFConfig(GPIOG,GPIO_PinSource12,GPIO_AF_FSMC);
```

//④FSMC 初始化

```
readWriteTiming.FSMC_AddressSetupTime = 0XF; //地址建立时间为 16 个 HCLK  
readWriteTiming.FSMC_AddressHoldTime = 0x00; //地址保持时间模式 A 未用到  
readWriteTiming.FSMC_DataSetupTime = 24;//数据保存时间为 25 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 =8; //地址建立时间（ADDSET）为 8 个 HCLK
writeTiming.FSMC_AddressHoldTime = 0x00; //地址保持时间
writeTiming.FSMC_DataSetupTime = 8; //数据保存时间为 6ns*9 个 HCLK=54ns
writeTiming.FSMC_BusTurnAroundDuration = 0x00;
writeTiming.FSMC_CLKDivision = 0x00;
writeTiming.FSMC_DataLatency = 0x00;
writeTiming.FSMC_AccessMode = FSMC_AccessMode_A; //模式 A

FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM4;
//这里我们使用 NE4 , 也就对应 BTCR[6],[7]。
FSMC_NORSRAMInitStructure.FSMC_DataAddressMux
    =FSMC_DataAddressMux_Disable; // 不复用数据地址
FSMC_NORSRAMInitStructure.FSMC_MemoryType =FSMC_MemoryType_SRAM;
// FSMC_MemoryType_SRAM;
FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth
    = FSMC_MemoryDataWidth_16b;//存储器数据宽度为 16bit
FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode
    =FSMC_BurstAccessMode_Disable;// FSMC_BurstAccessMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity
    =FSMC_WaitSignalPolarity_Low;
FSMC_NORSRAMInitStructure.FSMC_AsyncronousWait
    =FSMC_AsyncronousWait_Disable;
FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive
    =FSMC_WaitSignalActive_BeforeWaitState;
FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
//存储器写使能
FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Enable;
// 读写使用不同的时序
FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &readWriteTiming;
//读写时序
FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &writeTiming; //写时序

FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure); //初始化 FSMC 配置

//⑤使能 FSMC
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM4, ENABLE); // 使能 BANK1

```

```
delay_ms(50); // delay 50 ms
lcddev.id = LCD_ReadReg(0x0000);
//⑥不同的 LCD 驱动器不同的初始化设置
if(lcddev.id<0xFF||lcddev.id==0xFFFF||lcddev.id==0X9300)
    //ID 不正确,新增 0X9300 判断, 因为 9341 在未被复位的情况下会被读成 9300
{
    //尝试 9341 ID 的读取
    LCD_WR_REG(0XD3);
    lcddev.id=LCD_RD_DATA();      //dummy read
    lcddev.id=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);
        lcddev.id=LCD_RD_DATA();//dummy read
        lcddev.id=LCD_RD_DATA();//读回 0X01
        lcddev.id=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!=0X6804)      //也不是 6804,尝试看看是不是 NT35310
        {
            LCD_WR_REG(0XD4);
            lcddev.id=LCD_RD_DATA();    //dummy read
            lcddev.id=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
            if(lcddev.id!=0X5310)      //也不是 NT35310,尝试看看是不是 NT35510
            {
                LCD_WR_REG(0XDA00);
                lcddev.id=LCD_RD_DATA();//读回 0X00
                LCD_WR_REG(0XDB00);
                lcddev.id=LCD_RD_DATA();//读回 0X80
                lcddev.id<<=8;
                LCD_WR_REG(0XDC00);
                lcddev.id|=LCD_RD_DATA();//读回 0X00
                if(lcddev.id==0x8000)lcddev.id=0x5510;
                //NT35510 读回的 ID 是 8000H,为方便区分,我们强制设置为 5510
            }
        }
    }
}
```

```

        }
    }

if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510)
{
    //如果是这三个 IC,则设置 WR 时序为最快
    //重新配置写时序控制寄存器的时序
    FSMC_Bank1E->BWTR[6]&=~(0XF<<0); //地址建立时间(ADDSET)清零
    FSMC_Bank1E->BWTR[6]&=~(0XF<<8); //数据保存时间清零
    FSMC_Bank1E->BWTR[6]|=3<<0;         //地址建立时间为 3 个 HCLK =18ns
    FSMC_Bank1E->BWTR[6]|=2<<8;          //数据保存时间为 6ns*3 个 HCLK=18ns
}else if(lcddev.id==0X6804||lcddev.id==0XC505)//6804/C505 速度上不去,得降低
{
    //重新配置写时序控制寄存器的时序
    FSMC_Bank1E->BWTR[6]&=~(0XF<<0); //地址建立时间(ADDSET)清零
    FSMC_Bank1E->BWTR[6]&=~(0XF<<8); //数据保存时间清零
    FSMC_Bank1E->BWTR[6]|=10<<0;        //地址建立时间为 10 个 HCLK =60ns
    FSMC_Bank1E->BWTR[6]|=12<<8;         //数据保存时间为 6ns*13 个 HCLK=78ns
}
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==0xXXXX)     //其他 LCD 初始化代码
{
    .....//其他 LCD 驱动 IC, 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①~⑥在代码中标注：

- ① GPIO,FSMC 使能。
- ② GPIO 初始化：GPIO_Init()函数。
- ③ 设置引脚复用映射。
- ④ FSMC 初始化：FSMC_NORSRAMInit()函数。
- ⑤ FSMC 使能：FSMC_NORSRAMCmd()函数。
- ⑥ 不同的 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。

LCD 驱动相关的函数就给大家讲解到这里。接下来，我们看看主函数代码如下：

```
int main(void)
{
    u8 x=0;
    u8 lcd_id[12];           //存放 LCD ID 字符串
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200

    LED_Init();             //初始化 LED
    LCD_Init();             //初始化 LCD FSMC 接口
    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(30,40,210,24,24,"Explorer STM32F4");
        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(30,130,200,12,12,"2014/5/4");
        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，字体大小包括 24*12、16*8 和 12*6 等三种，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。

另外特别注意：`uart_init` 函数，不能去掉，因为在 `LCD_Init` 函数里面调用了 `printf`，所以一旦你去掉这个初始化，就会死机了！实际上，只要你的代码有用到 `printf`，就必须初始化串口，否则都会死机，即停在 `usart.c` 里面的 `fputc` 函数，出不来。

在编译通过之后，我们开始下载验证代码。

18.4 下载验证

将程序下载到探索者 STM32F4 开发板后，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，支持负数）、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 V3.2。

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，支持负数）、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：

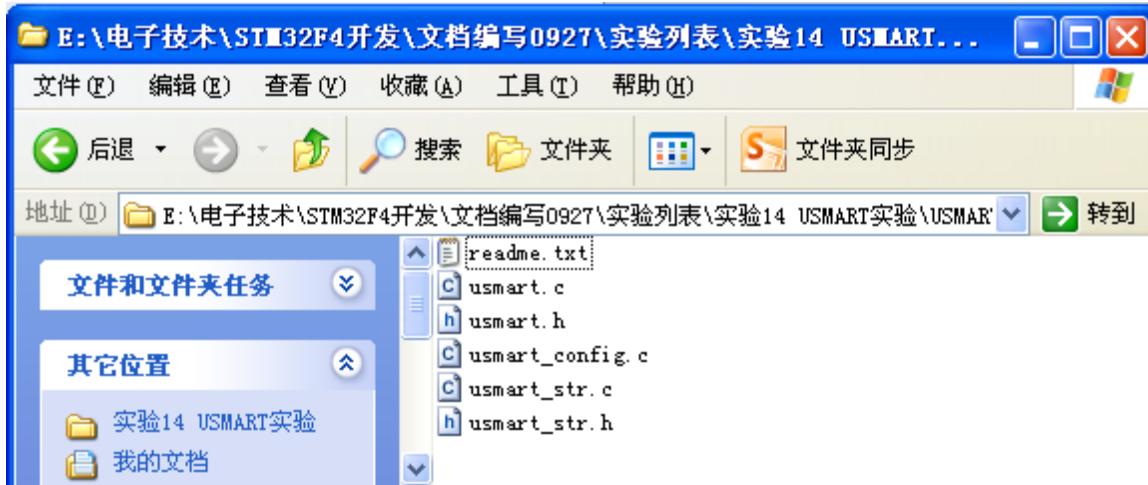


图 19.1.1 USMART 组件代码

其中 `readme.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
    Timer4_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。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，就初始化定时器中断，并在中断里面调用 `usmart_scan` 函数。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)
{
    TIM_ClearFlag(TIM4,TIM_FLAG_Update);//清除中断标志位
    TIM_SetAutoreload(TIM4,0xFFFF);//将重装载值设置到最大
    TIM_SetCounter(TIM4,0); //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(TIM_GetFlagStatus(TIM4,TIM_FLAG_Update)==SET)//在运行期间,产生了定时器溢出
{
    usmart_dev.runtime+=0xFFFF;
}
usmart_dev.runtime+=TIM_GetCounter(TIM4);
return usmart_dev.runtime; //返回计数值
}
```

这里我们利用定时器 4 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 STM32F4 的定时器 4，该定时器是 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： $65535 \times 2 \times 0.1\text{ms} = 13.1\text{ 秒}$ 。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.
//非 ALIENTEK 开发板用户,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

软件设计我们在上一章实验的基础上添加 USMART 组件相关的支持。打开上一章 LCD 显示实验工程，复制 USMART 文件夹（该文件夹可以在：光盘→标准例程-库函数版本→实验 14 USMART 调试组件实验 里面找到）到 LCD 工程文件夹下面，如图 19.3.1 所示：

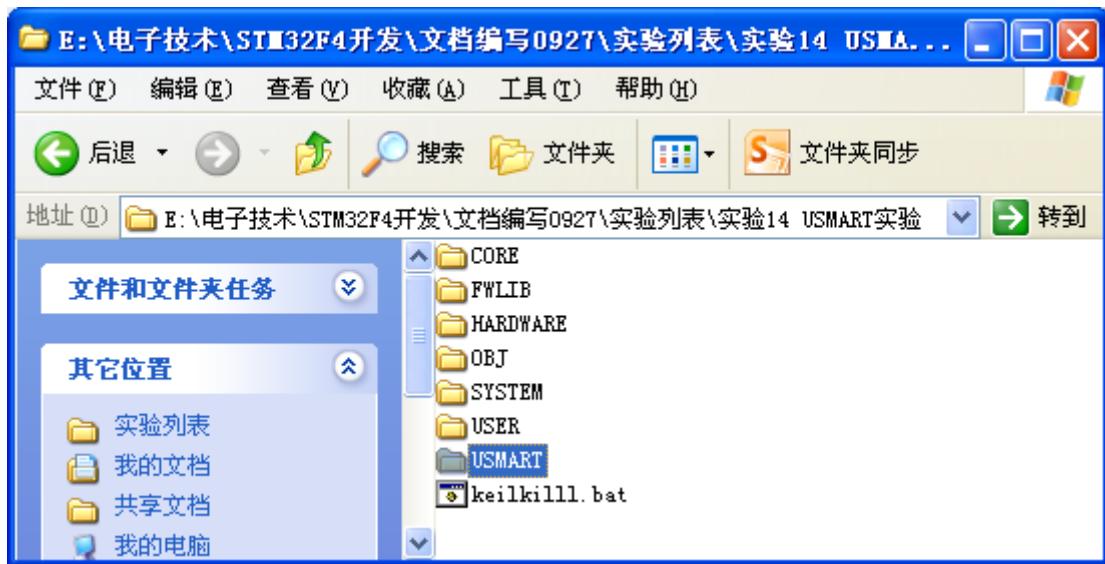


图 19.3.1 复制 USMART 文件夹到工程文件夹下

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：

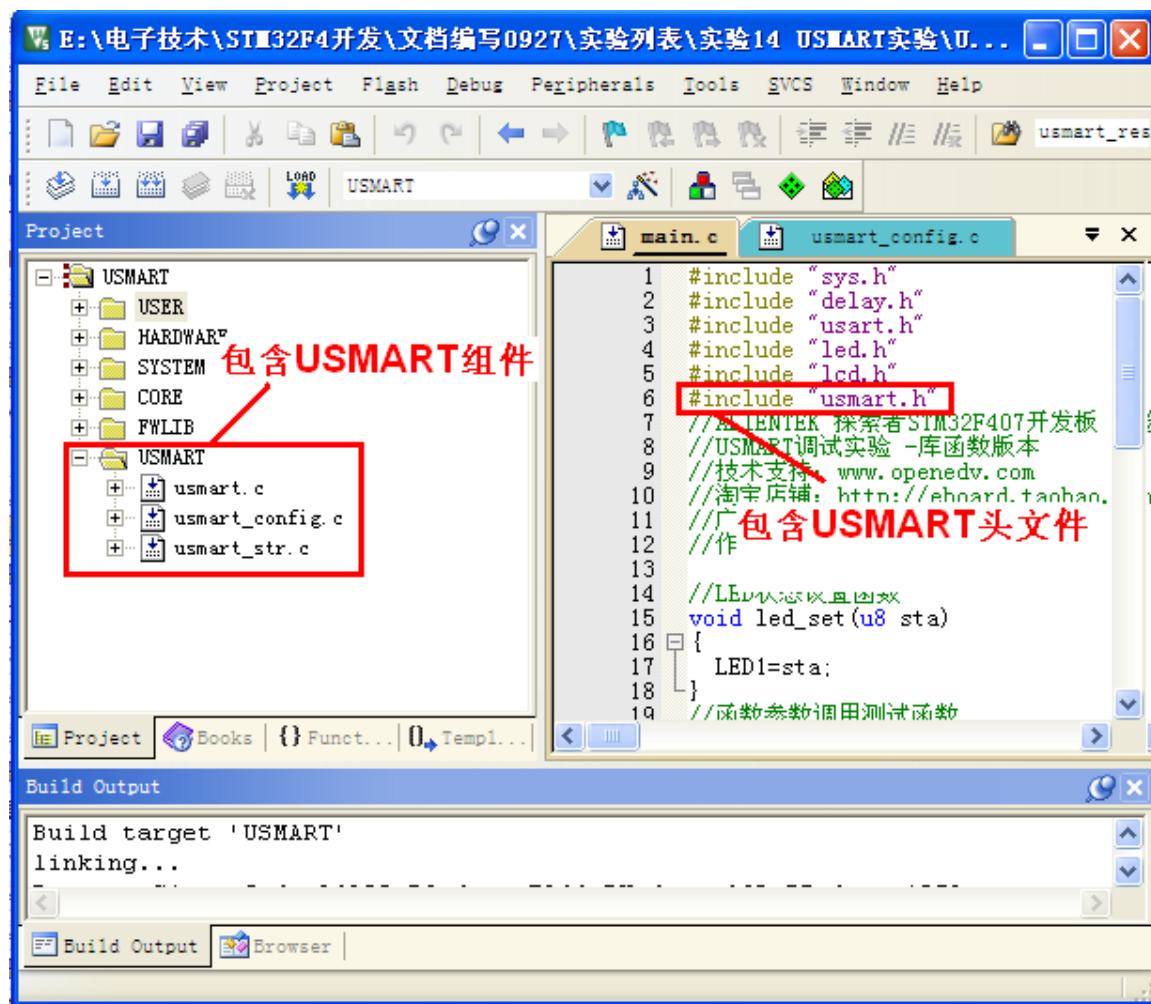


图 19.3.2 添加 USMART 组件代码

由于 USMART 默认提供了 STM32F4 的 TIM4 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 TIM4 的设置, 通过 TIM4 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:

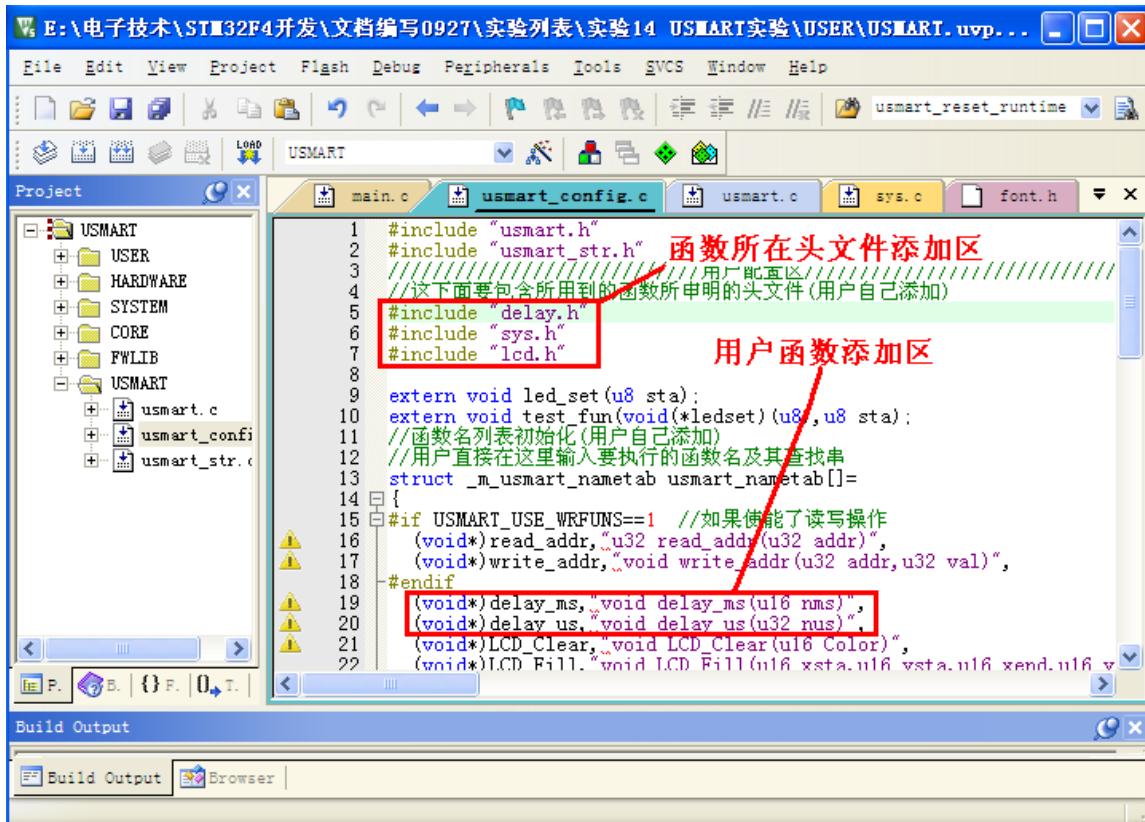


图 19.3.3 添加需要被 USMART 调用的函数

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：

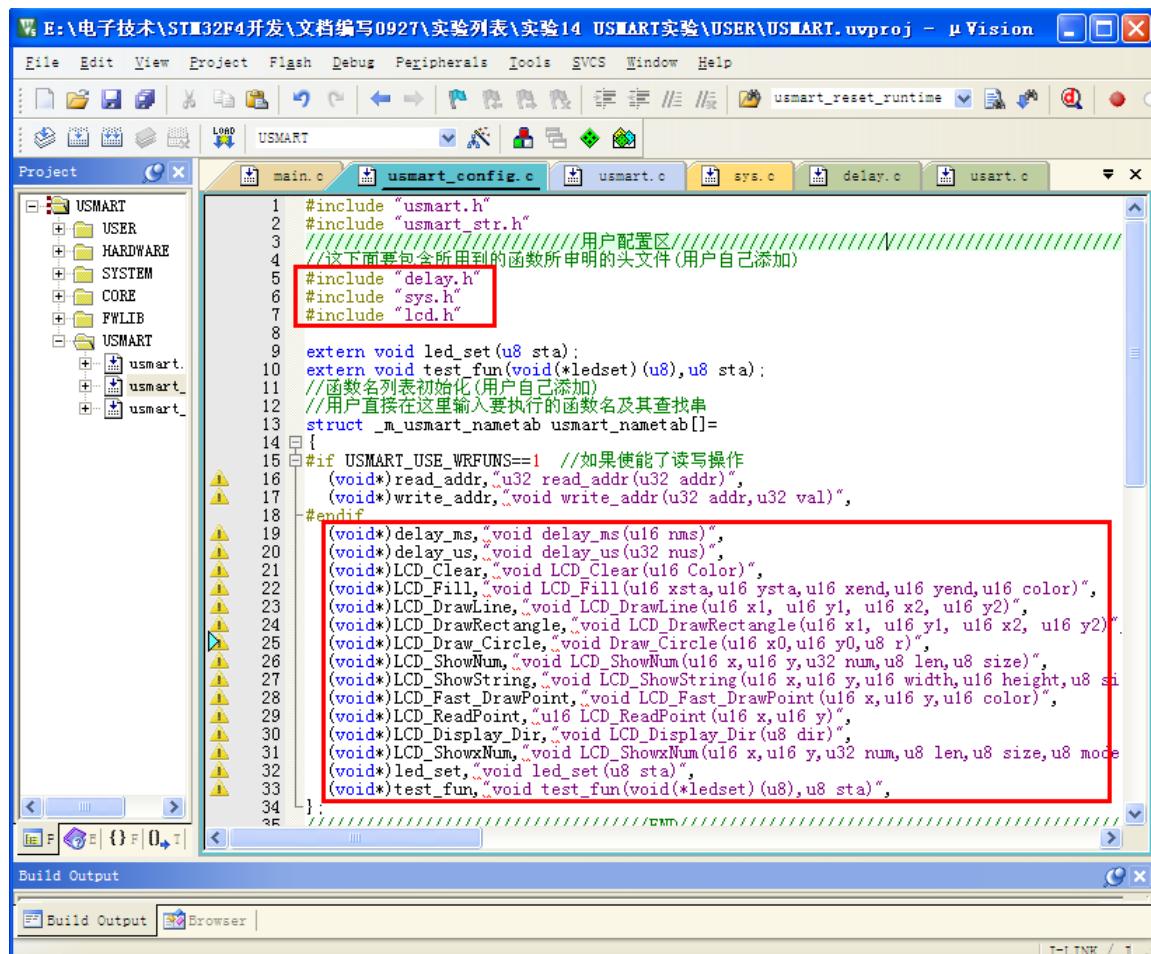


图 19.3.4 添加函数后

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)
{
```

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
```

```
delay_init(168);          //初始化延时函数
uart_init(115200);        //初始化串口波特率为 115200
usmart_dev.init(84);      //初始化 USMART
LED_Init();                //初始化 LED
LCD_Init();                //初始化 LCD
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
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,"2014/5/5");
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

将程序下载到探索者 STM32F4 开发板后，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 XCOM，选择正确的串口号→多条发送→勾选发送新行（即发送回车键）选项，然后发送 list 指令，即可打印所有 usmart 可调用函数。如下图所示：

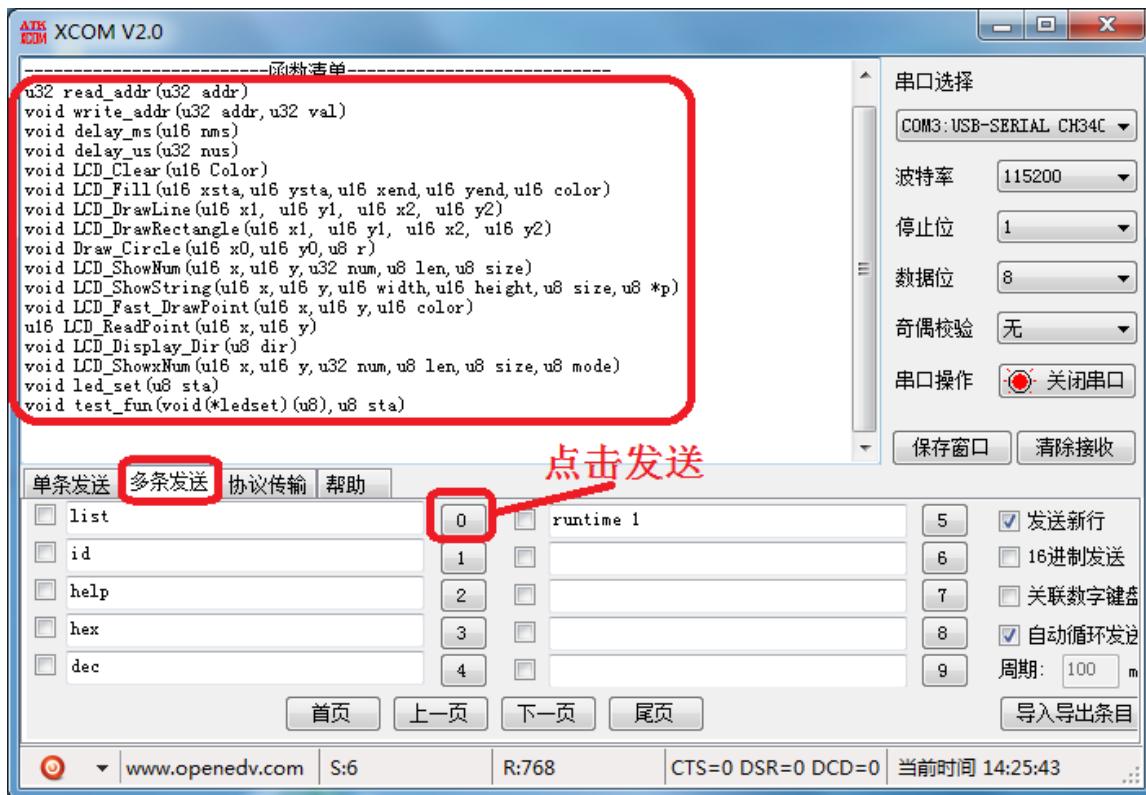


图 19.4.1 驱动串口调试助手

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：

上图中 list、id、help、hex、dec 和 runtime 都属于 usmart 自带的系统命令，点击后方的数字按钮，即可发送对应的指令。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 19.4.1 所示。

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 led_set 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。

help（或者 ‘?’ 也可以），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 19.4.2 所示：

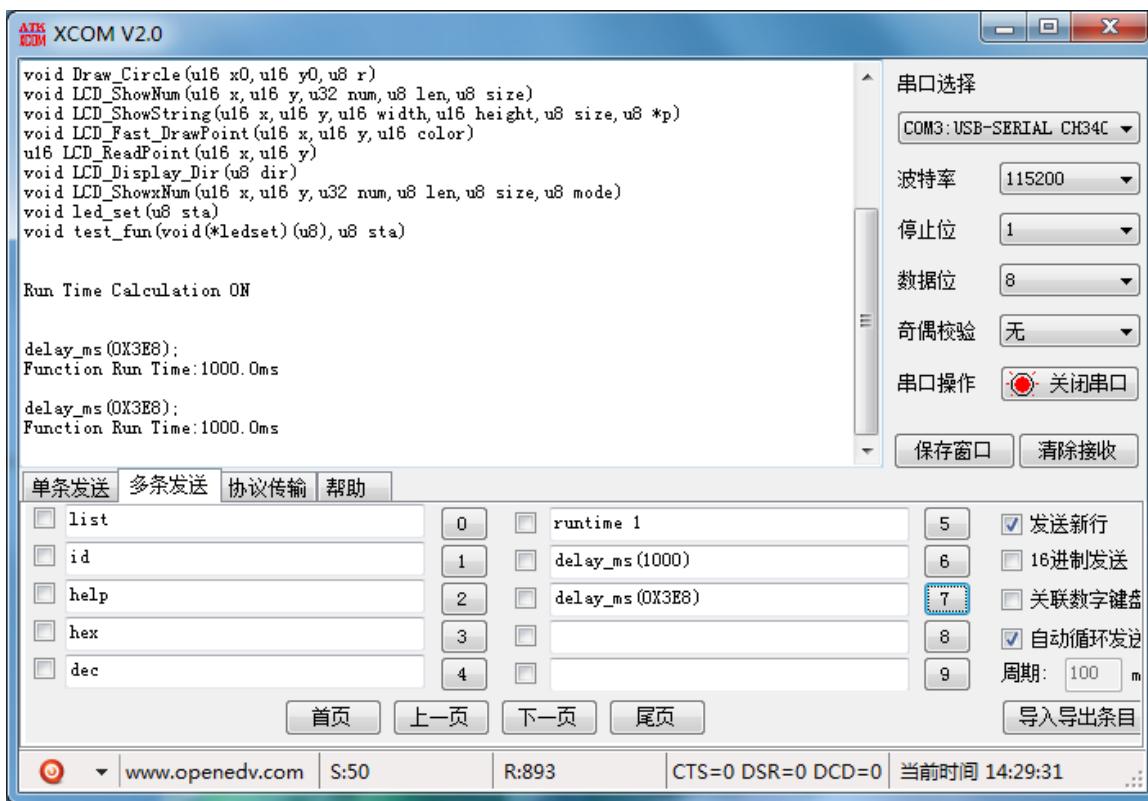


图 19.4.2 串口调用 delay_ms 函数

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 19.4.3 所示：

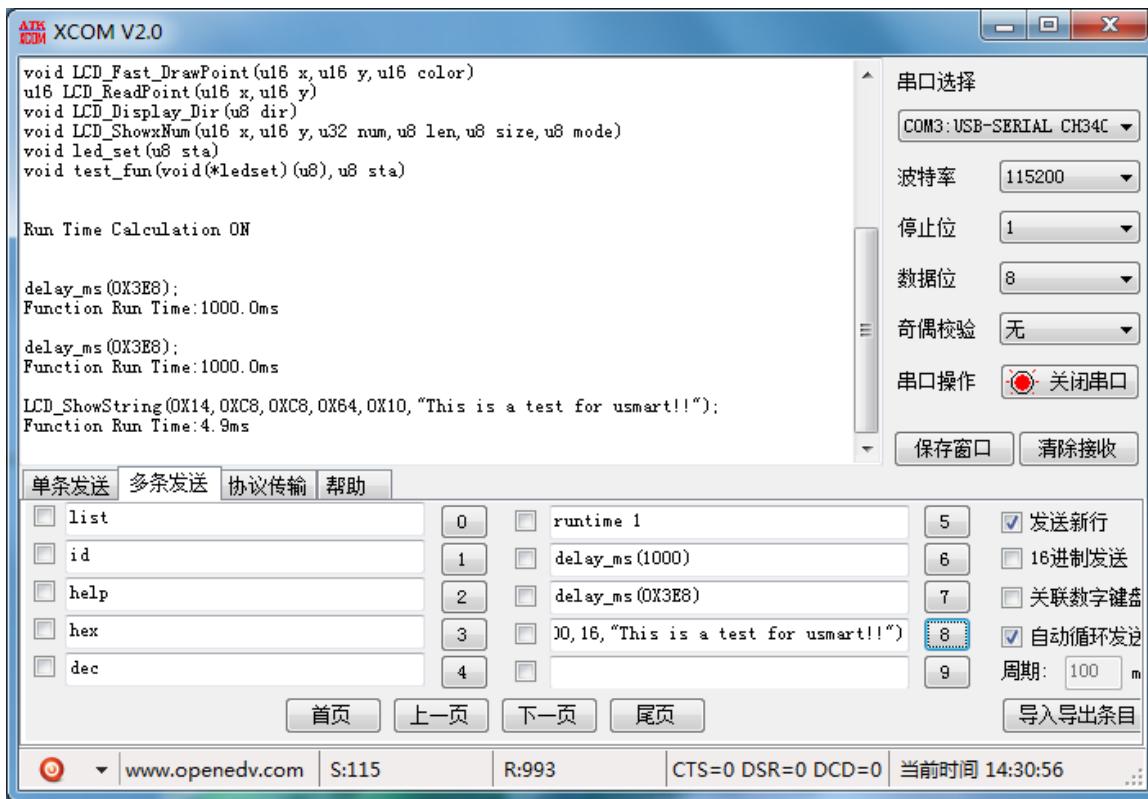


图 19.4.3 串口调用 LCD_ShowString 函数

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：0X080052C9，所以，我们在串口输入： test_fun(0X080052C9,0)，就可以控制 DS1 亮了。如图 19.4.4 所示：

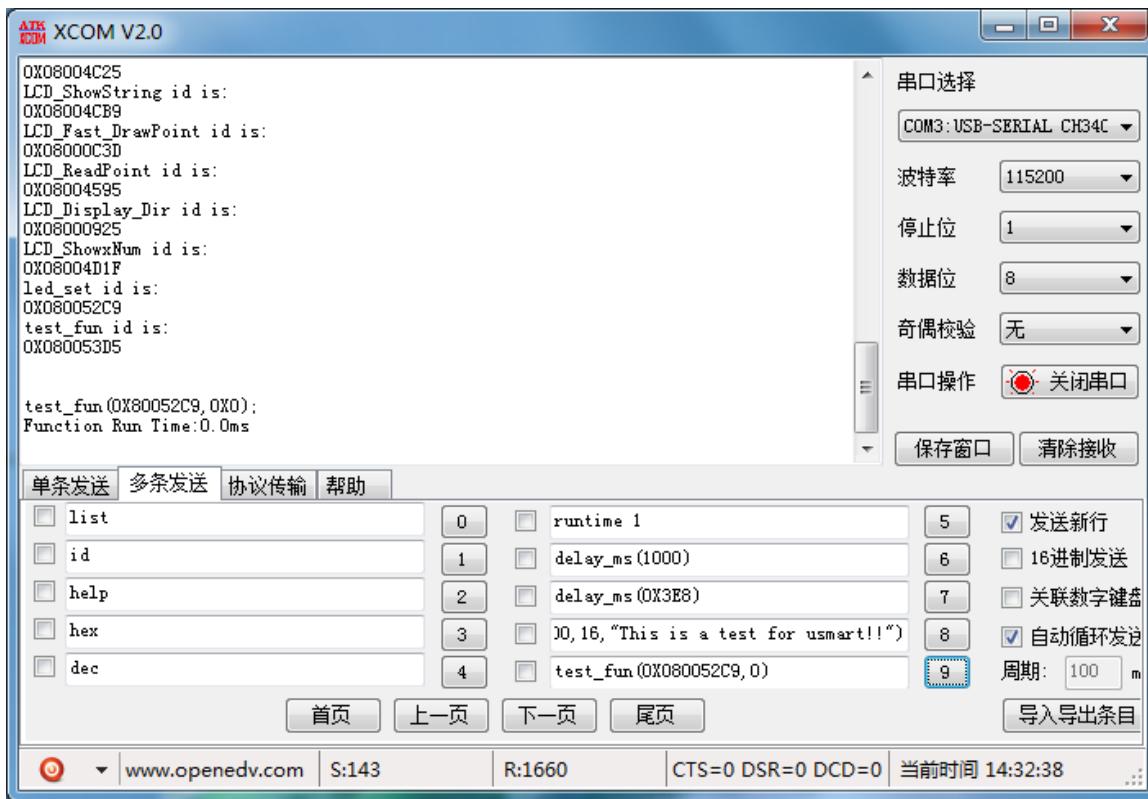


图 19.4.4 串口调用 test_fun 函数

在开发板上，我们可以看到，收到串口发送的 test_fun(0X080052C9,0)后，开发板的 DS1 亮了，然后大家可以通过发送 test_fun(0X080052C9,1)，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 STM32F4 的内部实时时钟（RTC）。在本章中，我们将使用 TFTLCD 模块来显示日期和时间，实现一个简单的实时时钟，并可以设置闹铃。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 STM32F4 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 STM32F4 RTC 时钟简介

STM32F4 的实时时钟（RTC）相对于 STM32F1 来说，改进了不少，带了日历功能了，STM32F4 的 RTC，是一个独立的 BCD 定时器/计数器。RTC 提供一个日历时钟（包含年月日时分秒信息）、两个可编程闹钟（ALARM A 和 ALARM B）中断，以及一个具有中断功能的周期性可编程唤醒标志。RTC 还包含用于管理低功耗模式的自动唤醒单元。

两个 32 位寄存器（TR 和 DR）包含二进码十进数格式（BCD）的秒、分钟、小时（12 或 24 小时制）、星期、日期、月份和年份。此外，还可提供二进制格式的亚秒值。

STM32F4 的 RTC 可以自动将月份的天数补偿为 28、29（闰年）、30 和 31 天。并且还可以进行夏令时 补偿。

RTC 模块和时钟配置是在后备区域，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，只要后备区域供电正常，那么 RTC 将可以一直运行。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：

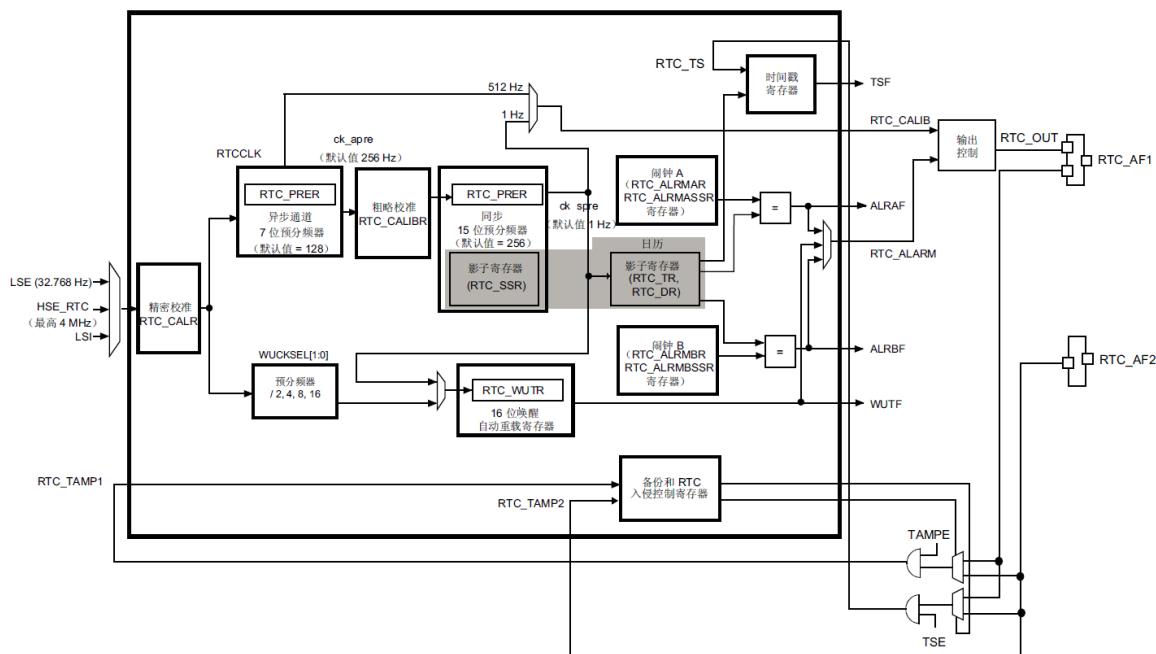


图 20.1.1 RTC 框图

此图可以从 STM32F4 中文参考手册 RTC 章节中找到（图 222：RTC 框图）。本章我们用到

RTC 时钟和日历，并且用到闹钟功能。接下来简单介绍下 STM32F4 RTC 时钟的使用。

1. 时钟和分频

首先，我们看 STM32F4 的 RTC 时钟分频。STM32F4 的 RTC 时钟源（RTCCLK）通过时钟控制器，可以从 LSE 时钟、LSI 时钟以及 HSE 时钟三者中选择（通过 RCC_BDCR 寄存器选择）。一般我们选择 LSE，即外部 32.768Khz 晶振作为时钟源(RTCCLK)，而 RTC 时钟核心，要求提供 1Hz 的时钟，所以，我们要设置 RTC 的可编程预分配器。STM32F4 的可编程预分配器（RTC_PRER）分为 2 个部分：

- 1, 一个通过 RTC_PRER 寄存器的 PREDIV_A 位配置的 7 位异步预分频器。
- 2, 一个通过 RTC_PRER 寄存器的 PREDIV_S 位配置的 15 位同步预分频器。

图 20.1.1 中，`ck_spre` 的时钟可由如下计算公式计算：

$$F_{CK_SPRE} = Fr_{CCLK} / [(PREDIV_S + 1) * (PREDIV_A + 1)]$$

其中，`Fck_spre` 即可用于更新日历时间等信息。PREDIV_A 和 PREDIV_S 为 RTC 的异步和同步分频器。且推荐设置 7 位异步预分频器（PREDIV_A）的值较大，以最大程度降低功耗。要设置为 32768 分频，我们只需要设置：`PREDIV_A=0X7F`，即 128 分频；`PREDIV_S=0xFF`，即 256 分频，即可得到 1Hz 的 `Fck_spre`。

另外，图 20.1.1 中，`ck_apre` 可作为 RTC 亚秒递减计数器（RTC_SSR）的时钟，`Fck_apre` 的计算公式如下：

$$F_{CK_APRE} = Fr_{CCLK} / (PREDIV_A + 1)$$

当 RTC_SSR 寄存器递减到 0 的时候，会使用 PREDIV_S 的值重新装载 PREDIV_S。而 PREDIV_S 一般为 255，这样，我们得到亚秒时间的精度是：1/256 秒，即 3.9ms 左右，有了这个亚秒寄存器 RTC_SSR，就可以得到更加精确的时间数据。

2. 日历时间（RTC_TR）和日期（RTC_DR）寄存器

STM32F4 的 RTC 日历时间（RTC_TR）和日期（RTC_DR）寄存器，用于存储时间和日期（也可以用于设置时间和日期），可以通过与 PCLK1（APB1 时钟）同步的影子寄存器来访问，这些时间和日期寄存器也可以直接访问，这样可避免等待同步的持续时间。

每隔 2 个 RTCCLK 周期，当前日历值便会复制到影子寄存器，并置位 RTC_ISR 寄存器的 RSF 位。我们可以读取 RTC_TR 和 RTC_DR 来得到当前时间和日期信息，不过需要注意的是：时间和日期都是以 BCD 码的格式存储的，读出来要转换一下，才可以得到十进制的数据。

3. 可编程闹钟

STM32F4 提供两个可编程闹钟：闹钟 A(ALARM_A)和闹钟 B(ALARM_B)。通过 RTC_CR 寄存器的 ALRAE 和 ALRBE 位置 1 来使能可编程闹钟功能。当日历的亚秒、秒、分、小时、日期分别与闹钟寄存器 RTC_ALRMSSR/RTC_ALRMAR 和 RTC_ALRMBSSR/RTC_ALRMBR 中的值匹配时，则可以产生闹钟（需要适当配置）。本章我们将利用闹钟 A 产生闹铃，即设置 RTC_ALRMSSR 和 RTC_ALRMAR 即可。

4. 周期性自动唤醒

STM32F4 的 RTC 不带秒钟中断了，但是多了一个周期性自动唤醒功能。周期性唤醒功能，由一个 16 位可编程自动重载递减计数器（RTC_WUTR）生成，可用于周期性中断/唤醒。

我们可以通过 RTC_CR 寄存器中的 WUTE 位设置使能此唤醒功能。

唤醒定时器的时钟输入可以是：2、4、8 或 16 分频的 RTC 时钟(RTCCLK)，也可以是 `ck_spre` 时钟（一般为 1Hz）。

当选择 RTCCLK(假定 LSE 是：32.768 kHz)作为输入时钟时，可配置的唤醒中断周期介于 122us（因为 RTCCLK/2 时，RTC_WUTR 不能设置为 0）和 32 s 之间，分辨率最低为：61us。

当选择 `ck_spre` (1Hz) 作为输入时钟时，可得到的唤醒时间为 1s 到 36h 左右，分辨率为 1

秒。并且这个 1s~36h 的可编程时间范围分为两部分：

当 WUCKSEL[2:1]=10 时为： 1s 到 18h。

当 WUCKSEL[2:1]=11 时约为： 18h 到 36h。

在后一种情况下，会将 2^{16} 添加到 16 位计数器当前值（即扩展到 17 位，相当于最高位用 WUCKSEL [1] 替代）。

初始化完成后，定时器开始递减计数。在低功耗模式下使能唤醒功能时，递减计数保持有效。此外，当计数器计数到 0 时，RTC_ISR 寄存器的 WUTF 标志会置 1，并且唤醒寄存器会使用其重载值（RTC_WUTR 寄存器值）自动重载，之后必须用软件清零 WUTF 标志。

通过将 RTC_CR 寄存器中的 WUTIE 位置 1 来使能周期性唤醒中断时，可以使 STM32F4 退出低功耗模式。系统复位以及低功耗模式（睡眠、停机和待机）对唤醒定时器没有任何影响，它仍然可以正常工作，故唤醒定时器，可以用于周期性唤醒 STM32F4。

接下来，我们看看本章我们要用到的 RTC 部分寄存器，首先是 RTC 时间寄存器：RTC_TR，该寄存器各位描述如图 20.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								PM	HT[1:0]		HU[3:0]				
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserv ed	MNT[2:0]			MNU[3:0]				Reserv ed	ST[2:0]		SU[3:0]				
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

位 31-24 保留

位 23 保留，必须保持复位值。

位 22 PM: AM/PM 符号 (AM/PM notation)

0: AM 或 24 小时制

1: PM

位 21:20 HT[1:0]: 小时的十位 (BCD 格式) (Hour tens in BCD format)

位 16:16 HU[3:0]: 小时的个位 (BCD 格式) (Hour units in BCD format)

位 15 保留，必须保持复位值。

位 14:12 MNT[2:0]: 分钟的十位 (BCD 格式) (Minute tens in BCD format)

位 11:8 MNU[3:0]: 分钟的个位 (BCD 格式) (Minute units in BCD format)

位 7 保留，必须保持复位值。

位 6:4 ST[2:0]: 秒的十位 (BCD 格式) (Second tens in BCD format)

位 3:0 SU[3:0]: 秒的个位 (BCD 格式) (Second units in BCD format)

图 20.1.2 RTC_TR 寄存器各位描述

这个寄存器比较简单，注意数据保存是 BCD 格式的，读取之后需要稍加转换，才是十进制的时分秒等数据，在初始化模式下，对该寄存器进行写操作，可以设置时间。

然后看 RTC 日期寄存器：RTC_DR，该寄存器各位描述如图 20.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								YT[3:0]				YU[3:0]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WDU[2:0]			MT	MU[3:0]				Reserved	DT[1:0]		DU[3:0]				
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

位 31-24 保留

位 23:20 YT[3:0]: 年份的十位 (BCD 格式) (Year tens in BCD format)

位 19:16 YU[3:0]: 年份的个位 (BCD 格式) (Year units in BCD format)

位 15:13 WDU[2:0]: 星期几的个位 (Week day units)

000: 禁止

001: 星期一

...

111: 星期日

位 12 MT: 月份的十位 (BCD 格式) (Month tens in BCD format)

位 11:8 MU: 月份的个位 (BCD 格式) (Month units in BCD format)

位 7:6 保留, 必须保持复位值。

位 5:4 DT[1:0]: 日期的十位 (BCD 格式) (Date tens in BCD format)

位 3:0 DU[3:0]: 日期的个位 (BCD 格式) (Date units in BCD format)

图 20.1.3 RTC_DR 寄存器各位描述

同样, 该寄存器的数据采用 BCD 码格式 (如不熟悉 BCD, 百度即可), 其他的就比较简单了。同样, 在初始化模式下, 对该寄存器进行写操作, 可以设置日期。

接下来, 看 RTC 亚秒寄存器: RTC_SSR, 该寄存器各位描述如图 20.1.4 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
SS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留

位 15:0 SS: 亚秒值 (Sub second value)

SS[15:0] 是同步预分频器计数器的值。此亚秒值可根据以下公式得出:

$$\text{亚秒值} = (\text{PREDIV_S} - \text{SS}) / (\text{PREDIV_S} + 1)$$

注意: 仅当执行平移操作之后, SS 才能大于 PREDIV_S。在这种情况下, 正确的时间/日期比 RTC_TR/RTC_DR 所指示的时间/日期慢一秒钟。

图 20.1.4 RTC_SSR 寄存器各位描述

该寄存器可用于获取更加精确的 RTC 时间。不过, 在本章没有用到, 如果需要精确时间的地方, 大家可以使用该寄存器。

接下来看 RTC 控制寄存器: RTC_CR, 该寄存器各位描述如图 20.1.5 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								COE	OSEL[1:0]		POL	COSEL	BKP	SUB1H	ADD1H
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSIE	WUTIE	ALRBE	ALRAIE	TSE	WUTE	ALRBE	ALRAE	DCE	FMT	BYPS_HAD	REFCKON	TSEDGE	WUCKSEL[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.5 RTC_CR 寄存器各位描述

该寄存器我们不详细介绍每个位了，重点介绍几个要用到的：WUTIE，ALRAIE 是唤醒定时器中断和闹钟 A 中断使能位，本章要用到，设置为 1 即可。WUTE 和 ALRAE，则是唤醒定时器和闹钟 A 定时器使能位，同样设置为 1，开启。Fmt 为小时格式选择位，我们设置为 0，选择 24 小时制。最后 WUCKSEL[2:0]，用于唤醒时钟选择，这个前面已经有介绍了，我们这里就不多说了，RTC_CR 寄存器的详细介绍，请看《STM32F4xx 中文参考手册》第 23.6.3 节。

接下来看 RTC 初始化和状态寄存器：RTC_ISR，该寄存器各位描述如图 20.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															RECAL PF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TAMP 2F	TAMP 1F	TSOVF	TSF	WUTF	ALRBF	ALRAF	INIT	INITF	RSF	INITS	SHPF	WUT WF	ALRB WF	ALRA WF
	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rw	r	rc_w0	r	rc_w0	r	r	r

图 20.1.6 RTC_ISR 寄存器各位描述

该寄存器中，WUTF、ALRBF 和 ALRAF，分别是唤醒定时器闹钟 B 和闹钟 A 的中断标志位，当对应事件产生时，这些标志位被置 1，如果设置了中断，则会进入中断服务函数，这些位通过软件写 0 清除；INIT 为初始化模式控制位，要初始化 RTC 时，必须先设置 INIT=1；INITF 为初始化标志位，当设置 INIT 为 1 以后，要等待 INITF 为 1，才可以更新时间、日期和预分频寄存器等；RSF 位为寄存器同步标志，仅在该位为 1 时，表示日历影子寄存器已同步，可以正确读取 RTC_TR/RTC_DR 寄存器的值了；WUTWF、ALRBWF 和 ALRAWF 分别是唤醒定时器、闹钟 B 和闹钟 A 的写标志，只有在这些位为 1 的时候，才可以更新对应的内容，比如：要设置闹钟 A 的 ALRMAR 和 ALRMASSR，则必须先等待 ALRAWF 为 1，才可以设置。

接下来看 RTC 预分频寄存器：RTC_PRER，该寄存器各位描述如图 20.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															PREDIV_A[6:0]
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PREDIV_S[14:0]															
Res.	rw														

位 31:24 保留

位 23 保留，必须保持复位值。

位 22:16 PREDIV_A[6:0]: 异步预分频系数 (Asynchronous prescaler factor)

下面是异步分频系数的公式：

$$\text{ck_apre} \text{ 频率} = \text{RTCKL} \text{ 频率}/(\text{PREDIV_A}+1)$$

注意：PREDIV_A [6:0]= 000000 为禁用值。

位 15 保留，必须保持复位值。

位 14:0 PREDIV_S[14:0]: 同步预分频系数 (Synchronous prescaler factor)

下面是同步分频系数的公式：

$$\text{ck_spre} \text{ 频率} = \text{ck_apre} \text{ 频率}/(\text{PREDIV_S}+1)$$

图 20.1.7 RTC_PRER 寄存器各位描述

该寄存器用于 RTC 的分频，我们在之前也有讲过，这里就不多说了。该寄存器的配置，必须在初始化模式 (INITF=1) 下，才可以进行。

接下来看 RTC 唤醒定时器寄存器：RTC_WUTR，该寄存器各位描述如图 20.1.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WUT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 保留

位 15:0 **WUT[15:0]**: 唤醒自动重载值位 (Wakeup auto-reload value bit)

当使能唤醒定时器时 (WUTE 置 1)，每 (WUT[15:0] + 1) 个 ck_wut 周期将 WUTF 标志置 1 一次。ck_wut 周期通过 RTC_CR 寄存器的 WUCKSEL[2:0] 位进行选择。

当 WUCKSEL[2] = 1 时，唤醒定时器变为 17 位，WUCKSEL[1] 等效为 WUT[16]，即要重载到定时器的最高有效位。

注意：WUTF 第一次置 1 发生在 WUTE 置 1 之后 (WUT+1) 个 ck_wut 周期。禁止在 WUCKSEL[2:0]=011(RTCCLK/2) 时将 WUT[15:0] 设置为 0x0000。

图 20.1.8 RTC_WUTR 寄存器各位描述

该寄存器用于设置自动唤醒重装载值，可用于设置唤醒周期。该寄存器的配置，必须等待 RTC_ISR 的 WUTWF 为 1 才可以进行。

接下来看 RTC 闹钟 A 器寄存器：RTC_ALRMAR，该寄存器各位描述如图 20.1.9 所示：

位 31 MSK4: 闹钟 A 日期掩码 (Alarm A date mask)

- 0: 如果日期/日匹配, 则闹钟 A 置 1
1: 在闹钟 A 比较中, 日期/日无关

位 30 WDSEL: 星期几选择 (Week day selection)

- 0: DU[3:0] 代表日期的个位
1: DU[3:0] 代表星期几。DT[1:0] 为无关位。

位 29:28 DT[1:0]: 日期的十位 (BCD 格式) (Date tens in BCD format)。

位 27:24 DU[3:0]: 日期的个位或日 (BCD 格式) (Date units or day in BCD format)。

位 23 MSK3: 闹钟 A 小时掩码 (Alarm A hours mask)

- 0: 如果小时匹配，则闹钟 A 置 1
1: 在闹钟 A 比较中，小时无关

位 22 PM: AM/PM 符号 (AM/PM notation)

- 0: AM 或 24 小时制
1: PM

位 21:20 HT[1:0]: 小时的十位 (BCD 格式) (Hour tens in BCD format)。

位 19:16 HU[3:0]: 小时的个位 (BCD 格式) (Hour units in BCD format)。

位 15 MSK2: 闹钟 A 分钟掩码 (Alarm A minutes mask)

- 0: 如果分钟匹配，则闹钟 A 置 1
1: 在闹钟 A 比较中，分钟无关

位 14:12 MNT[2:0]: 分钟的十位 (BCD 格式) (Minute tens in BCD format)。

位 11:8 MNU[3:0]: 分钟的个位 (BCD 格式) (Minute units in BCD format)。

位 7 MSK1: 闹钟 A 秒掩码 (Alarm A seconds mask)

- 0: 如果秒匹配, 则闹钟 A 置 1
1: 在闹钟 A 比较中, 秒无关

位 6:4 ST[2:0]: 秒的十位 (BCD 格式) (Second tens in BCD format)。

位 3:0 SU[3:0]: 秒的个位 (BCD 格式) (Second units in BCD format)。

图 20.1.9 RTC_ALRMAR 寄存器各位描述

该寄存器用于设置闹铃 A，当 WDSEL 选择 1 时，使用星期制闹铃，本章我们选择星期制闹铃。该寄存器的配置，必须等待 RTC_ISR 的 ALRAWF 为 1 才可以进行。另外，还有 RTC_ALRMASSR 寄存器，该寄存器我们这里就不再介绍了，大家参考《STM32F4xx 中文数据手册》第 23.6.19 节。

接下来我们看 RTC 写保护寄存器：RTC_WPR，该寄存器比较简单，低八位有效。上电后，所有 RTC 寄存器都受到写保护（RTC_ISR[13:8]、RTC_TAFCR 和 RTC_BKPxR 除外），必须依次写入：0XCA、0X53 两关键字到 RTC_WPR 寄存器，才可以解锁。写一个错误的关键字将再次激活 RTC 的寄存器写保护。

接下来，我们介绍下 RTC 备份寄存器：RTC_BKPxR，该寄存器组总共有 20 个，每个寄存器是 32 位的，可以存储 80 个字节的用户数据，这些寄存器在备份域中实现，可在 VDD 电源关闭时通过 VBAT 保持上电状态。备份寄存器不会在系统复位或电源复位时复位，也不会在 MCU 从待机模式唤醒时复位。

复位后，对 RTC 和 RTC 备份寄存器的写访问被禁止，执行以下操作可以使能对 RTC 及

RTC 备份寄存器的写访问：

- 1) 通过设置寄存器 RCC_APB1ENR 的 PWREN 位来打开电源接口时钟
- 2) 电源控制寄存器(PWR_CR)的 DBP 位来使能对 RTC 及 RTC 备份寄存器的访问。

我们可以用 BKP 来存储一些重要的数据，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。

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 20.1.10 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved															BDRST	
															rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
RTCEN	Reserved				RTCSEL[1:0]		Reserved				LSEBYP	LSERDY	LSEON			
rw					rw	rw					rw	r	rw			

位 31:17 保留，必须保持复位值。

位 16 **BDRST**: 备份域软件复位 (Backup domain software reset)

由软件置 1 和清零。

- 0: 复位未激活
- 1: 复位整个备份域

注意: BKPSRAM 不受此复位影响，只能在 Flash 保护级别从级别 1 更改为级别 0 时复位 BKPSRAM。

位 15 **RTCEN**: RTC 时钟使能 (RTC clock enable)

由软件置 1 和清零。

- 0: RTC 时钟禁止
- 1: RTC 时钟使能

位 14:10 保留，必须保持复位值。

位 9:8 **RTCSEL[1:0]**: RTC 时钟源选择 (RTC clock source selection)

由软件置 1，用于选择 RTC 的时钟源。选择 RTC 时钟源后，除非备份域复位，否则其不可再更改。可使用 BDRST 位对其进行复位。

- 00: 无时钟
- 01: LSE 振荡器时钟用作 RTC 时钟
- 10: LSI 振荡器时钟用作 RTC 时钟
- 11: 由可编程预分频器分频的 HSE 振荡器时钟（通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 RTCPRE[4:0] 位选择）用作 RTC 时钟

位 7:3 保留，必须保持复位值。

位 2 **LSEBYP**: 外部低速振荡器旁路 (External low-speed oscillator bypass)

由软件置 1 和清零，用于旁路调试模式下的振荡器。只有在禁止 LSE 时钟后才能写入该位。

- 0: 不旁路 LSE 振荡器
- 1: 旁路 LSE 振荡器

位 1 **LSERDY**: 外部低速振荡器就绪 (External low-speed oscillator ready)

由硬件置 1 和清零，用于指示外部 32 kHz 振荡器已稳定。在 LSEON 位被清零后，LSERDY 将在 6 个外部低速振荡器时钟周期后转为低电平。

- 0: LSE 时钟未就绪
- 1: LSE 时钟就绪

位 0 **LSEON**: 外部低速振荡器使能 (External low-speed oscillator enable)

- 由软件置 1 和清零。
- 0: LSE 时钟关闭
- 1: LSE 时钟开启

图 20.1.10 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 操作之前首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。

RTC 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。接下来我们来看看通过库函数配置 RTC 一般配置步骤。RTC 相关的库函数文件为 `stm32f4xx_rtc.c` 以及头文件 `stm32f4xx_rtc.h` 中：

1) 使能电源时钟，并使能 RTC 及 RTC 后备寄存器写访问。

前面已经介绍了，我们要访问 RTC 和 RTC 备份区域就必须先使能电源时钟，然后使能 RTC 即后备区域访问。电源时钟使能，通过 `RCC_APB1ENR` 寄存器来设置；RTC 及 RTC 备份寄存器的写访问，通过 `PWR_CR` 寄存器的 `DBP` 位设置。库函数设置方法为：

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); //使能 PWR 时钟  
PWR_BackupAccessCmd(ENABLE); //使能后备寄存器访问
```

2) 开启外部低速振荡器，选择 RTC 时钟，并使能。

这个步骤，只需要在 RTC 初始化的时候执行一次即可，不需要每次上电都执行，这些操作都是通过 `RCC_BDCR` 寄存器来实现的。

开启 LSE 的库函数为：

```
RCC_LSEConfig(RCC_LSE_ON); //LSE 开启
```

同时，选择 RTC 时钟源以及使能时钟函数为：

```
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //设选择 LSE 作为 RTC 时钟  
RCC_RTCCLKCmd(ENABLE); //使能 RTC 时钟
```

3) 初始化 RTC，设置 RTC 的分频，以及配置 RTC 参数。

在库函数中，初始化 RTC 是通过函数 `RTC_Init` 实现的：

```
ErrorStatus RTC_Init(RTC_InitTypeDef* RTC_InitStruct);
```

同样按照以前的方式，我们来看看 RTC 初始化参数结构体 `RTC_InitTypeDef` 定义：

```
typedef struct  
{  
    uint32_t RTC_HourFormat;  
    uint32_t RTC_AsynchPrediv;  
    uint32_t RTC_SynchPrediv;  
}RTC_InitTypeDef;
```

结构体一共只有三个成员变量，我们逐一来看看：

参数 `RTC_HourFormat` 用来设置 RTC 的时间格式，也就是我们前面寄存器讲解的设置 CR 寄存器的 `FMT` 位。如果设置为 24 小时格式参数值可选择 `RTC_HourFormat_24`，12 小时格式，参数值可以选择 `RTC_HourFormat_24`。

参数 `RTC_AsynchPrediv` 用来设置 RTC 的异步预分频系数，也就是设置 RTC_PRER 寄存器的 `PREDIV_A` 相关位。同时，因为异步预分频系数是 7 位，所以最大值为 `0x7F`，不能超过这个值。

参数 `RTC_SynchPrediv` 用来设置 RTC 的同步预分频系数，也就是设置 RTC_PRER 寄存器的 `PREDIV_S` 相关位。同时，因为同步预分频系数也是 15 位，所以最大值为 `0x7FFF`，不能超过这个值。

最后关于 `RTC_Init` 函数我们还要指出，在设置 RTC 相关参数之前，会先取消 RTC 写保护，这个操作通过向寄存器 `RTC_WPR` 写入 `0xCA` 和 `0x53` 两个数据实现。所以 `RTC_Init` 函数体开头会有下面两行代码用来取消 RTC 写保护：

```
RTC->WPR = 0xCA;  
RTC->WPR = 0x53;
```

在取消写保护之后，我们要对 `RTC_PRER`、`RTC_TR` 和 `RTC_DR` 等寄存器的写操作，必须先进入 RTC 初始化模式，才可以进行，库函数中进入初始化模式的函数为：

```
ErrorStatus RTC_EnterInitMode(void);
```

进入初始化模式之后，RTC_init 函数才去设置 RTC->CR 以及 RTC->PRER 寄存器的值。在设置完值之后，我们还要退出初始化模式，函数为：

```
void RTC_ExitInitMode(void)
```

最后再开启 RTC 写保护，往 RTC_WPR 寄存器写入值 0xFF 即可。

4) 设置 RTC 的时间。

库函数中，设置 RTC 时间的函数为：

```
ErrorStatus RTC_SetTime(uint32_t RTC_Format, RTC_TimeTypeDef* RTC_TimeStruct);
```

实际上，根据我们前面寄存器的讲解，RTC_SetTime 函数是用来设置时间寄存器 RTC_TR 的相关位的值。

RTC_SetTime 函数的第一个参数 RTC_Format,用来设置输入的时间格式为 BIN 格式还是 BCD 格式，可选值为 RTC_Format_BIN 和 RTC_Format_BCD。因为 RTC_DR 的数据必须是 BCD 格式，所以如果您设置为 RTC_Format_BIN，那么在函数体内部会调用函数 RTC_ByteToBcd2 将参数转换为 BCD 格式。这里还是比较理解的。

我们接下来看看第二个初始化参数结构体 RTC_TimeTypeDef 的定义：

```
typedef struct
```

```
{
```

```
    uint8_t RTC_Hours;  
    uint8_t RTC_Minutes;  
    uint8_t RTC_Seconds;  
    uint8_t RTC_H12;
```

```
}RTC_TimeTypeDef;
```

这四个的参数真的就比较好理解了，分别用来设置 RTC 时间参数的小时，分钟，秒钟，以及 AM/PM 符号，大家参考前面讲解的 RTC_TR 的位描述即可。

5) 设置 RTC 的日期。

设置 RTC 的日期函数为：

```
ErrorStatus RTC_SetDate(uint32_t RTC_Format, RTC_DateTypeDef* RTC_DateStruct);
```

实际上，根据我们前面寄存器的讲解，RTC_SetDate 设置日期函数是用来设置日期寄存器 RTC_DR 的相关位的值。

第一个参数 RTC_Format,跟函数 RTC_SetTime 的第一个入口参数是一样的，用来设置输入日期格式。

接下来我们看看第二个日期初始化参数结构体 RTC_DateTypeDef 的定义：

```
typedef struct
```

```
{
```

```
    uint8_t RTC_WeekDay;  
    uint8_t RTC_Month;  
    uint8_t RTC_Date;  
    uint8_t RTC_Year;
```

```
}RTC_DateTypeDef;
```

这四个参数也很好理解，分别用来设置日期的星期几，月份，日期，年份。这个大家可以参考我们前面讲解的 RTC_DR 寄存器的位描述来理解。

6) 获取 RTC 当前日期和时间。

获取当前 RTC 时间的函数为：

```
void RTC_GetTime(uint32_t RTC_Format, RTC_TimeTypeDef* RTC_TimeStruct);
```

获取当前 RTC 日期的函数为：

```
void RTC_GetDate(uint32_t RTC_Format, RTC_DateTypeDef* RTC_DateStruct);
```

这两个函数非常简单，实际就是读取 RTC_TR 寄存器和 RTC_DR 寄存器的时间和日期的值，然后将值存放到相应的结构体中。

通过以上 6 个步骤，我们就完成了对 RTC 的配置，RTC 即可正常工作，而且这些操作不是每次上电都必须执行的，可以视情况而定。当然，我们还需要设置时间、日期、唤醒中断、闹钟等，这些将在后面介绍。

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 STM32F4 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 探索者 STM32F4 开发板标配是有电池的）。

20.3 软件设计

打开本章实验工程可以看到，我们先在 FWLIB 下面引入了 RTC 支持的库函数文件 stm32f4xx_rtc.c。然后我们在 HARDWARE 文件夹下新建了一个 rtc.c 的文件和 rtc.h 的头文件，同时将这两个文件引入我们的工程 HARDWARE 分组下。

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 My_RTC_Init，其代码如下：

```
u8 My_RTC_Init(void)
{
    RTC_InitTypeDef RTC_InitStructure;
    u16 retry=0X1FFF;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); //使能 PWR 时钟
    PWR_BackupAccessCmd(ENABLE); //使能后备寄存器访问
    if(RTC_ReadBackupRegister(RTC_BKP_DR0)!=0x5050)//是否第一次配置？
    {
        RCC_LSEConfig(RCC_LSE_ON); //LSE 开启
        while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
            //检查指定的 RCC 标志位设置与否，等待低速晶振就绪
        {
            retry++;
            delay_ms(10);
        }
        if(retry==0) return 1; //LSE 开启失败。
    }
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); //选择 LSE 作为 RTC 时钟
}
```

```

RCC_RTCCLKCmd(ENABLE); //使能 RTC 时钟

RTC_InitStructure.RTC_AsynchPrediv = 0x7F;//RTC 异步分频系数(1~0X7F)
RTC_InitStructure.RTC_SynchPrediv = 0xFF;//RTC 同步分频系数(0~7FFF)
RTC_InitStructure.RTC_HourFormat = RTC_HourFormat_24;//24 小时格式
RTC_Init(&RTC_InitStructure);//初始化 RTC 参数

RTC_Set_Time(23,59,56,RTC_H12_AM); //设置时间
RTC_Set_Date(14,5,5,1); //设置日期
RTC_WriteBackupRegister(RTC_BKP_DR0,0x5050);//标记已经初始化过了
}

return 0;
}

```

该函数用来初始化 RTC 配置以及日期和时钟，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。

这里设置时间和日期，分别是通过 RTC_Set_Time 和 RTC_Set_Date 函数来实现的，这两个函数实际就是调用库函数里面的 RTC_SetTime 函数和 RTC_SetDate 函数来实现，这里我们之所以要写两个这样的函数，目的是为了我们的 USMART 来调用，方便直接通过 USMART 来设置时间和日期。

这里默认将时间设置为 14 年 5 月 5 日星期 1，23 点 59 分 56 秒。在设置好时间之后，我们调用函数 RTC_WriteBackupRegister 向 RTC 的 BKR 寄存器（地址 0）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，该函数通过调用函数 RTC_ReadBackupRegister 判断 RTC 对应 BKR 地址的值，来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，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。

这里我们来看看读备份区域和写备份区域寄存器的两个函数为：

```

uint32_t RTC_ReadBackupRegister(uint32_t RTC_BKP_DR);
void RTC_WriteBackupRegister(uint32_t RTC_BKP_DR, uint32_t Data);

```

这两个函数的使用方法就非常简单，分别用来读和写 BKR 寄存器的值。这里我们只是略点到为止。

接着，我们介绍一下 RTC_Set_AlarmA 函数，该函数代码如下：

```

//设置闹钟时间(按星期闹铃,24 小时制)
//week:星期几(1~7) @ref RTC_Alarm_Definitions
//hour,min,sec:小时,分钟,秒钟
void RTC_Set_AlarmA(u8 week,u8 hour,u8 min,u8 sec)
{
    EXTI_InitTypeDef    EXTI_InitStructure;
    RTC_AlarmTypeDef   RTC_AlarmTypeInitStructure;
    RTC_TimeTypeDef    RTC_TimeTypeInitStructure;

    RTC_AlarmCmd(RTC_Alarm_A,DISABLE);//关闭闹钟 A
}

```

```

RTC_TimeTypeInitStructure.RTC_Hours=hour;//小时
RTC_TimeTypeInitStructure.RTC_Minutes=min;//分钟
RTC_TimeTypeInitStructure.RTC_Seconds=sec;//秒
RTC_TimeTypeInitStructure.RTC_H12=RTC_H12_AM;

RTC_AlarmTypeInitStructure.RTC_AlarmDateWeekDay=week;//星期
RTC_AlarmTypeInitStructure.RTC_AlarmDateWeekDaySel
    =RTC_AlarmDateWeekDaySel_WeekDay;//按星期闹
RTC_AlarmTypeInitStructure.RTC_AlarmMask=RTC_AlarmMask_None;
                                //精确匹配星期，时分秒
RTC_AlarmTypeInitStructure.RTC_AlarmTime=RTC_TimeTypeInitStructure;
RTC_SetAlarm(RTC_Format_BIN,RTC_Alarm_A,&RTC_AlarmTypeInitStructure);

RTC_ClearITPendingBit(RTC_IT_ALRA);//清除 RTC 钟 A 的标志
EXTI_ClearITPendingBit(EXTI_Line17);//清除 LINE17 上的中断标志位
RTC_ITConfig(RTC_IT_ALRA,ENABLE);//开启闹钟 A 中断
RTC_AlarmCmd(RTC_Alarm_A,ENABLE);//开启闹钟 A

EXTI_InitStructure.EXTI_Line = EXTI_Line17;//LINE17
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;//中断事件
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;//上升沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;//使能 LINE17
EXTI_Init(&EXTI_InitStructure);//配置

NVIC_InitStructure.NVIC_IRQChannel = RTC_Alarm IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;//抢占优先级 1
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;//响应优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能外部中断通道
NVIC_Init(&NVIC_InitStructure);//配置
}

```

该函数用于设置闹钟 A，也就是设置 ALRMAR 和 ALRMASSR 寄存器的值，来设置闹钟时间，这里库函数中用来设置闹钟的函数为：

```

void RTC_SetAlarm(uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmTypeDef*
RTC_AlarmStruct);

```

第一个参数 RTC_Format 用来设置格式，这里前面我们讲解过，就不做过多讲解。

第二个参数 RTC_Alarm 用来设置是闹钟 A 还是闹钟 B，我们使用的是闹钟 A，所以值为 RTC_Alarm_A。

第三个参数就是我们用来设置闹钟参数的结构体指针。接下来我们看看 RTC_AlarmTypeDef 结构体的定义：

```

typedef struct
{
    RTC_TimeTypeDef RTC_AlarmTime;
    uint32_t RTC_AlarmMask;
}

```

```

    uint32_t RTC_AlarmDateWeekDaySel;
    uint8_t RTC_AlarmDateWeekDay;
}RTC_AlarmTypeDef;

```

结构体的第一个成员变量为 RTC_TimeTypeDef 类型的成员变量 RTC_AlarmTime，这个是用来设置闹钟时间的，RTC_TimeTypeDef 结构体成员变量的含义我们在之前已经讲解，这里我们就不做过多讲解。

第二个参数 RTC_AlarmMask，使用来设置闹钟时间掩码，也就是在我们第一个参数设置的时间中（包括后面参数 RTC_AlarmDateWeekDay 设置的星期几/哪一天），哪些是无关的。比如我们设置闹钟时间为每天的 10 点 10 分 10 秒，那么我们可以选择值 RTC_AlarmMask_DateWeekDay，也就是我们不关心是星期几/每月哪一天。这里我们选择为 RTC_AlarmMask_None，也就是精确匹配时间，所有的时分秒以及星期几/(或者每月哪一天)都要精确匹配。

第三个参数 RTC_AlarmDateWeekDaySel，用来选择是闹钟是按日期还是按星期。比如我们选择 RTC_AlarmDateWeekDaySel_WeekDay 那么闹钟就是按星期。如果我们选择 RTC_AlarmDateWeekDaySel_Date 那么闹钟就是按日期。这与后面第四个参数是有关联的，我们在后面第四个参数讲解。

第四个参数 RTC_AlarmDateWeekDay 用来设置闹钟的日期或者星期几。比如我们第三个参数 RTC_AlarmDateWeekDaySel 设置了值为 RTC_AlarmDateWeekDaySel_WeekDay,也就是按星期，那么参数 RTC_AlarmDateWeekDay 的取值范围就为星期一~星期天，也就是 RTC_Weekday_Monday~RTC_Weekday_Sunday。如果第三个参数 RTC_AlarmDateWeekDaySel 设置值为 RTC_AlarmDateWeekDaySel_Date，那么它的取值范围就为日期值，0~31。

调用函数 RTC_SetAlarm 设置闹钟 A 的参数之后，最后，开启闹钟 A 中断（连接在外部中断线 17），并设置中断分组。当 RTC 的时间和闹钟 A 设置的时间完全匹配时，将产生闹钟中断。

接着，我们介绍一下 RTC_Set_WakeUp 函数，该函数代码如下：

```

//周期性唤醒定时器设置
//wksel: @ref RTC_Wakeup_Timer_Definitions
//cnt:自动重装载值.减到 0,产生中断.
void RTC_Set_WakeUp(u32 wksel,u16 cnt)
{
    EXTI_InitTypeDef    EXTI_InitStructure;
    RTC_WakeUpCmd(DISABLE); //关闭 WAKE UP
    RTC_WakeUpClockConfig(wksel); //唤醒时钟选择
    RTC_SetWakeUpCounter(cnt); //设置 WAKE UP 自动重装载寄存器
    RTC_ClearITPendingBit(RTC_IT_WUT); //清除 RTC WAKE UP 的标志
    EXTI_ClearITPendingBit(EXTI_Line22); //清除 LINE22 上的中断标志位
    RTC_ITConfig(RTC_IT_WUT,ENABLE); //开启 WAKE UP 定时器中断
    RTC_WakeUpCmd(ENABLE); //开启 WAKE UP 定时器

    EXTI_InitStructure.EXTI_Line = EXTI_Line22; //LINE22
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //中断事件
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; //上升沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE; //使能 LINE22
    EXTI_Init(&EXTI_InitStructure); //配置
}

```

```

NVIC_InitStructure.NVIC IRQChannel = RTC_WKUP_IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x02;//抢占优先级 1
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x02;//响应优先级 2
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;//使能外部中断通道
NVIC_Init(&NVIC_InitStructure);//配置
}

```

该函数用于设置 RTC 周期性唤醒定时器，步骤同 RTC_Set_AlarmA 级别一样，只是周期性唤醒中断，连接在外部中断线 22。

有了中断设置函数，就必定有中断服务函数，接下来看这两个中断的中断服务函数，代码如下：

```

//RTC 闹钟中断服务函数
void RTC_Alarm_IRQHandler(void)
{
    if(RTC_GetFlagStatus(RTC_FLAG_ALRAF)==SET)//ALARM A 中断?
    {
        RTC_ClearFlag(RTC_FLAG_ALRAF); //清除中断标志
        printf("ALARM A!\r\n");
    }
    EXTI_ClearITPendingBit(EXTI_Line17); //清除中断线 17 的中断标志
}

```

```

//RTC WAKE UP 中断服务函数
void RTC_WKUP_IRQHandler(void)
{
    if(RTC_GetFlagStatus(RTC_FLAG_WUTF)==SET)//WK_UP 中断?
    {
        RTC_ClearFlag(RTC_FLAG_WUTF); //清除中断标志
        LED1=!LED1;
    }
    EXTI_ClearITPendingBit(EXTI_Line22); //清除中断线 22 的中断标志
}

```

其中，RTC_Alarm_IRQHandler 函数用于闹钟中断，该函数先判断中断类型，然后执行对应操作，每当闹钟 A 闹铃时，会从串口打印一个：ALARM A!的字符串。RTC_WKUP_IRQHandler 函数用于 RTC 自动唤醒定时器中断，先判断中断类型，然后对 LED1 取反操作，可以通过观察 LED1 的状态来查看 RTC 自动唤醒中断的情况。

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。rtc.h 头文件中主要是一些函数声明，我们就不多说了，有些函数在这里没有介绍，请大家参考本例程源码。

最后我们看看 main 函数源码如下：

```

int main(void)
{
    RTC_TimeTypeDef RTC_TimeStruct;

```

```
RTC_DateTypeDef RTC_DateStruct;

u8 tbuf[40];
u8 t=0;
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
delay_init(168); //初始化延时函数
uart_init(115200); //初始化串口波特率为 115200
usmart_dev.init(84); //初始化 USMART
LED_Init(); //初始化 LED
LCD_Init(); //初始化 LCD
My_RTC_Init(); //初始化 RTC

RTC_Set_WakeUp(RTC_WakeUpClock_CK_SPRE_16bits,0);//WAKE UP 每秒一次中
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"RTC TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/5");
while(1)
{
    t++;
    if((t%10)==0) //每 100ms 更新一次显示数据
    {
        RTC_GetTime(RTC_Format_BIN,&RTC_TimeStruct);
        sprintf((char*)tbuf,"Time:%02d:%02d:%02d",RTC_TimeStruct.RTC_Hours,
                RTC_TimeStruct.RTC_Minutes,RTC_TimeStruct.RTC_Seconds);
        LCD_ShowString(30,140,210,16,16,tbuf);
        RTC_GetDate(RTC_Format_BIN, &RTC_DateStruct);

        sprintf((char*)tbuf,"Date:20%02d-%02d-%02d",RTC_DateStruct.RTC_Year,
                RTC_DateStruct.RTC_Month,RTC_DateStruct.RTC_Date);
        LCD_ShowString(30,160,210,16,16,tbuf);
        sprintf((char*)tbuf,"Week:%d",RTC_DateStruct.RTC_WeekDay);
        LCD_ShowString(30,180,210,16,16,tbuf);
    }
    if((t%20)==0)LED0=!LED0; //每 200ms,翻转一次 LED0
    delay_ms(10);
}
}
```

这部分代码，也比较简单，注意，我们通过

RTC_Set_WakeUp(RTC_WakeUpClock_CK_SPRE_16bits,0);设置 RTC 周期性自动唤醒周期为 1 秒钟，类似于 STM32F1 的秒钟中断。然后，在 main 函数不断的读取 RTC 的时间和日期（每 100ms 一次），并显示在 LCD 上面。

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*)RTC_Set_Time,"u8 RTC_Set_Time(u8 hour,u8 min,u8 sec,u8 ampm)",  
        (void*)RTC_Set_Date,"u8 RTC_Set_Date(u8 year,u8 month,u8 date,u8 week)",  
        (void*)RTC_Set_AlarmA,"void RTC_Set_AlarmA(u8 week,u8 hour,u8 min,u8 sec)",  
        (void*)RTC_Set_WakeUp,"void RTC_Set_WakeUp(u8 wksel,u16 cnt)",  
};
```

将 RTC 的一些相关函数加入了 usmart，这样通过串口就可以直接设置 RTC 时间、日期、闹钟 A、周期性唤醒和备份寄存器读写等操作。

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

将程序下载到探索者 STM32F4 开发板后，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：

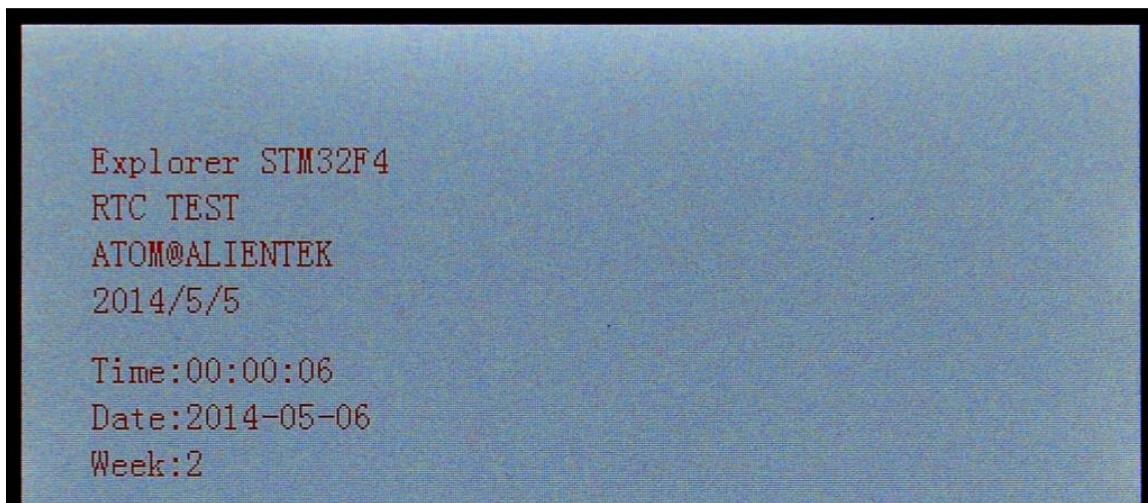


图 20.4.1 RTC 实验测试图

如果时间和日期不正确，可以利用上一章介绍的 usmart 工具，通过串口来设置，并且可以设置闹钟时间等，如图 20.4.2 所示：

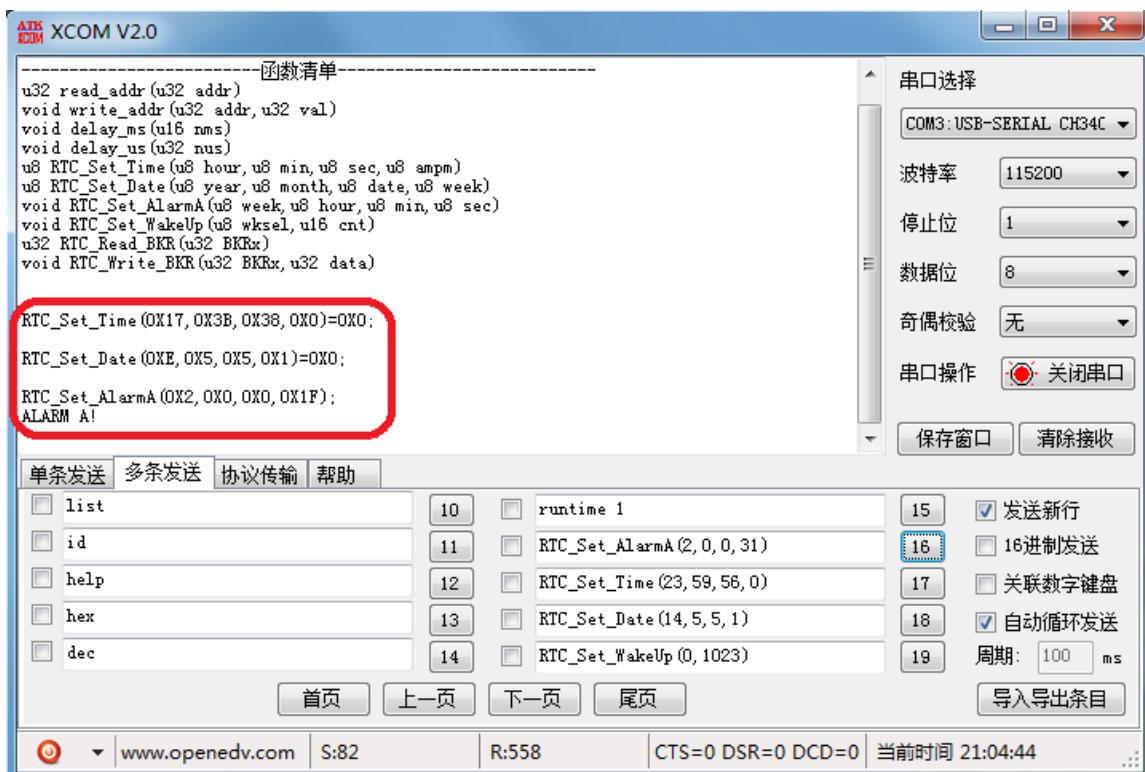


图 20.4.2 通过 USMART 设置时间和日期并测试闹钟 A

可以看到，设置闹钟 A 后，串口返回了 ALARM A!字符串，说明我们的闹钟 A 代码正常运行了！

第二十一章 硬件随机数实验

本章我们将向大家介绍 STM32F4 的硬件随机数发生器。在本章中，我们将使用 KEY0 按键来获取硬件随机数，并且将获取到的随机数值显示在 LCD 上面，同时，使用 DS0 指示程序运行状态。本章将分为如下几个部分：

- 21.1 STM32F4 随机数发生器简介
- 21.2 硬件设计
- 21.3 软件设计
- 21.4 下载验证

21.1 STM32F4 随机数发生器简介

STM32F4 自带了硬件随机数发生器 (RNG)，RNG 处理器是一个以连续模拟噪声为基础的随机数发生器，在主机读数时提供一个 32 位的随机数。STM32F4 的随机数发生器框图如图 21.1.1 所示：

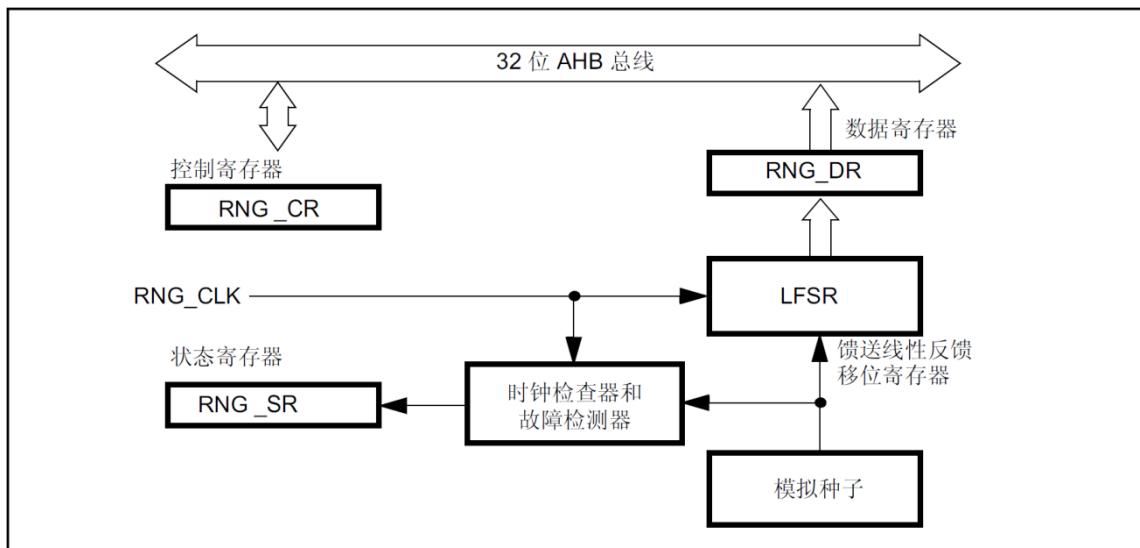


图 21.1.1 随机数发生器(RNG)框图

STM32F4 的随机数发生器 (RNG) 采用模拟电路实现。此电路产生馈入线性反馈移位寄存器 (RNG_LFSR) 的种子，用于生成 32 位随机数。

该模拟电路由几个环形振荡器组成，振荡器的输出进行异或运算以产生种子。RNG_LFSR 由专用时钟 (PLL48CLK) 按恒定频率提供时钟信息，因此随机数质量与 HCLK 频率无关。当将大量种子引入 RNG_LFSR 后，RNG_LFSR 的内容会传入数据寄存器 (RNG_DR)。

同时，系统会监视模拟种子和专用时钟 PLL48CLK，当种子上出现异常序列，或 PLL48CLK 时钟频率过低时，可以由 RNG_SR 寄存器的对应位读取到，如果设置了中断，则在检测到错误时，还可以产生中断。

接下来，我们介绍下 STM32F4 随机数发生器 (RNG) 的几个寄存器。

首先是 RNG 控制寄存器：RNG_CR，该寄存器各位描述如图 21.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												IE	RNGEN	Reserved	
												rw	rw		

位 31:4 保留，必须保持复位值

位 3 **IE**: 中断使能 (Interrupt enable)

0: 禁止 RNG 中断。

1: 使能 RNG 中断。只要 RNG_SR 寄存器中 DRDY=1 或 SEIS=1 或 CEIS=1，就会挂起中断。

位 2 **RNGEN**: 随机数发生器使能 (Random number generator enable)

0: 禁止随机数发生器。

1: 使能随机数发生器。

位 1:0 保留，必须保持复位值

图 21.1.2 RNG_CR 寄存器各位描述

该寄存器只有 bit2 和 bit3 有效，用于使能随机数发生器和中断。我们一般不用中断，所以只需要设置 bit2 为 1，使能随机数发生器即可。

然后，我们看看 RNG 状态寄存器：RNG_SR，该寄存器各位描述如图 21.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16		
Reserved																	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Reserved												SEIS	CEIS	Reserved	SECS	CECS	DRDY

图 21.1.3 RNG_SR 寄存器各位描述

该寄存器我们仅关心最低位 (DRDY 位)，该位用于表示 RNG_DR 寄存器包含的随机数数据是否有效，如果该位为 1，则说明 RNG_DR 的数据是有效的，可以读取出来了。读 RNG_DR 后，该位自动清零。

最后，我们看看 RNG 数据寄存器：RNG_DR，该寄存器各位描述如图 21.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RNDDATA															
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
RNDDATA															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:0 **RNDDATA**: 随机数据 (Random data)

32 位随机数据。

图 21.1.4 RNG_DR 寄存器各位描述

在 RNG_SR 的 DRDY 位置位后，我们就可以读取该寄存器获得 32 位随机数值。此寄存器在最多 40 个 PLL48CK 时钟周期后，又可以提供新的随机数值。

至此，随机数发生器的寄存器，我们就介绍完了。接下来，我们看看要使用库函数操作随机数发生器，应该如何设置。

首先，我们要说明的是，库函数中随机数发生器相关的操作在文件 `stm32f4xx_rng.c` 和对应的头文件 `stm32f4xx_rng.h` 中。所以我们实验工程必须引入这两个文件。

随机数发生器操作步骤如下：

1) 使能随机数发生器时钟。

要使用随机数发生器，必须先使能其时钟。随机数发生器时钟来自 PLL48CK，通过 AHB2ENR 寄存器使能。所以我们调用使能 AHB2 总线外设时钟的函数使能 RNG 时钟即可：

```
RCC_AHB2PeriphClockCmd(RCC_AHB2Periph RNG, ENABLE); //开启 RNG 时钟
```

2) 使能随机数发生器。

这个就是通过 RNG_CR 寄存器的最低位设置为 1，使能随机数发生器。当然，如果需要用到中断，你快还可以使能 RNG 中断。本章我们不用中断。库函数中使能随机数发生器的方法为：

```
RNG_Cmd(ENABLE); //使能 RNG
```

3) 判断 DRDY 位，读取随机数值。

经过前面两个步骤，我们就可以读取随机数值了，不过每次读取之前，必须先判断 RNG_SR 寄存器的 DRDY 位，如果该位为 1，则可以读取 RNG_DR 得到随机数值，如果不为 1，则需要等待。

在库函数中，获取随机数发生器状态的函数为：

```
FlagStatus RNG_GetFlagStatus(uint8_t RNG_FLAG);
```

库函数中，判断数据是否有效的入口参数为 RNG_FLAG_DRDY，所以等待就绪的方法为：

```
while(RNG_GetFlagStatus(RNG_FLAG_DRDY)==RESET);
```

判断数据有效后，然后我们读取随机数发生器产生的随机数即可，调用函数为：

```
uint32_t RNG_GetRandomNumber(void);
```

通过以上几个步骤的设置，我们就可以使用 STM32F4 的随机数发生器（RNG）了。本章，我们将实现如下功能：通过 KEY0 获取随机数，并将获取到的随机数显示在 LCD 上面，通过 DS0 指示程序运行状态。

21.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) 串口
- 3) KEY0 按键
- 4) 随机数发生器(RNG)
- 5) TFTLCD 模块

这些资源，我们都已经介绍了，硬件连接上面也不需要任何变动，插上 TFTLCD 模块即可。

21.3 软件设计

打开本章的实验工程可以看到，我们在 FWLIB 下面添加了随机数发生器支持库函数 `stm32f4xx_rng.c` 和对应的头文件 `stm32f4xx_rng.h`。同时我们编写的随机数发生器相关的函数在新增的文件 `rng.c` 中。

接下来我们看看 `rng.c` 源文件内容：

```
//初始化 RNG
//返回值:0,成功;1,失败
u8 RNG_Init(void)
{
    u16 retry=0;
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph RNG, ENABLE); //开启 RNG 时钟
    RNG_Cmd(ENABLE); //使能 RNG
    while(RNG_GetFlagStatus(RNG_FLAG_DRDY)==RESET&&retry<10000)//等待就绪
    {
        retry++; delay_us(100);
    }
}
```

```
        }
        if(retry>=10000) return 1;//随机数产生器工作不正常
        return 0;
    }
    //得到随机数
    //返回值:获取到的随机数
    u32 RNG_Get_RandomNum(void)
    {
        while(RNG_GetFlagStatus(RNG_FLAG_DRDY)==RESET); //等待随机数就绪
        return RNG_GetRandomNumber();      }
    //生成[min,max]范围的随机数
    int RNG_Get_RandomRange(int min,int max)
    {
        return RNG_Get_RandomNum()% (max-min+1) +min;
    }
```

该部分总共 3 个函数，其中：RNG_Init 用于初始化随机数发生器；RNG_Get_RandomNum 用于读取随机数值；RNG_Get_RandomRange 用于读取一个特定范围内的随机数，实际上也是调用的前一个函数 RNG_Get_RandomNum 来实现的。这些函数的实现方法都比较好理解。

rng.h 头文件内容就主要是三个函数申明，比较简单，这里我们就不做讲解。

最后我们看看 main.c 文件内容：

```
int main(void)
{
    u32 random; u8 t=0,key;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    KEY_Init(); //按键初始化
    LCD_Init(); //初始化液晶接口
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"RNG TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/5");
    while(RNG_Init()) //初始化随机数发生器
    {
        LCD_ShowString(30,130,200,16,16,"RNG Error! ");
        delay_ms(200);
        LCD_ShowString(30,130,200,16,16,"RNG Trying... ");
    }
    LCD_ShowString(30,130,200,16,16,"RNG Ready!   ");
    LCD_ShowString(30,150,200,16,16,"KEY0:Get Random Num");
    LCD_ShowString(30,180,200,16,16,"Random Num:"');
```

```
POINT_COLOR=BLUE;
while(1)
{
    delay_ms(10);
    key=KEY_Scan(0);
    if(key==KEY0_PRES)
    {
        random=RNG_Get_RandomNum(); //获得随机数
        LCD_ShowNum(30+8*11,180,random,10,16); //显示随机数
    }
    if((t%20)==0)
    {
        LED0=!LED0; //每 200ms,翻转一次 LED0
        random=RNG_Get_RandomRange(0,9); //获取[0,9]区间的随机数
        LCD_ShowNum(30+8*22,210,random,2,16); //显示随机数
    }
    delay_ms(10); t++;
}
```

该部分代码也比较简单，在所有外设初始化成功后，进入死循环，等待按键按下，如果 KEY0 按下，则调用 RNG_Get_RandomNum 函数，读取随机数值，并将读到的随机数显示在 LCD 上面。每隔 200ms 获取一次区间[0,9]的随机数，并实时显示在液晶上。同时 DS0，周期性闪烁，400ms 闪烁一次。这就实现了前面我们所说的功能。

至此，本实验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。

21.4 下载验证

将程序下载到探索者 STM32F4 开发板后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时每隔 200ms，获取一次区间[0,9]的随机数，实时显示在液晶上。然后我们也可以按下 KEY0，就可以在屏幕上看到获取到的随机数，如图 21.4.1 所示：

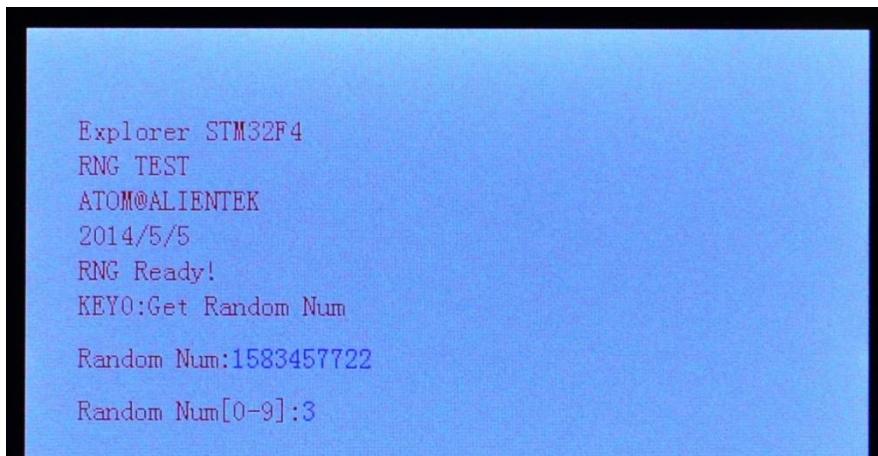


图 21.4.1 获取随机数成功

第二十二章 待机唤醒实验

本章我们将向大家介绍 STM32F4 的待机唤醒功能。在本章中，我们将使用 KEY_UP 按键来实现唤醒和进入待机模式的功能，然后使用 DS0 指示状态。本章将分为如下几个部分：

22.1 STM32F4 待机模式简介

22.2 硬件设计

22.3 软件设计

22.4 下载验证

22.1 STM32F4 待机模式简介

很多单片机都有低功耗模式，STM32F4 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。STM32F4 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。

STM32F4 提供了 3 种低功耗模式，以达到不同层次的降低功耗的目的，这三种模式如下：

- 1) 睡眠模式（CM4 内核停止工作，外设仍在运行）；
- 2) 停止模式（所有的时钟都停止）；
- 3) 待机模式；

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 22.1.1 所示：

模式名称	进入	唤醒	对 1.2 V 域时钟的影响	对 V _{DD} 域时钟的影响	调压器
睡眠 (立即休眠或退出时休眠)	WFI	任意中断	CPU CLK 关闭 对其它时钟或模拟时钟源无影响	无	开启
	WFE	唤醒事件			
停止	PDDS 和 LPDS 位 + SLEEPDEEP 位 + WFI 或 WFE	任意 EXTI 线（在 EXTI 寄存器中配置，内部线和外部线）	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	开启或处于低功耗模式（取决于用于 STM32F405xx/07xx 和 STM32F415xx/17xx 的 PWR 电源控制寄存器 (PWR_CR) 和用于 STM32F42xxx 和 STM32F43xxx 的 PWR 电源控制寄存器 (PWR_CR)）
待机	PDDS 位 + SLEEPDEEP 位 + WFI 或 WFE	WKUP 引脚上升沿、RTC 闹钟（闹钟 A 或闹钟 B）、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位、IWDG 复位	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	关闭

表 22.1.1 STM32F4 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 2.2uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 350uA 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 STM32F4 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32F4 的最低功耗。该模式是在 CM4 深睡眠模式时关闭电压调节器。整个 1.2V 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。除备份域（RTC 寄存器、RTC 备份寄存器和备份 SRAM）和待机电路中的寄存器外，SRAM 和寄存器内容都将丢失。

那么我们如何进入待机模式呢？其实很简单，只要按图 22.1.1 所示的步骤执行就可以了：

待机模式	说明
进入模式	WFI (等待中断) 或 WFE (等待事件)，且： <ul style="list-style-type: none"> - 将 Cortex™-M4F 系统控制寄存器中的 SLEEPDEEP 位置 1 - 将电源控制寄存器 (PWR_CR) 中的 PDDS 位置 1 - 将电源控制/状态寄存器 (PWR_CSR) 中的 WUF 位清零 - 将与所选唤醒源 (RTC 闹钟 A、RTC 闹钟 B、RTC 唤醒、RTC 入侵或 RTC 时间戳标志) 对应的 RTC 标志清零
退出模式	WKUP 引脚上升沿、RTC 闹钟 (闹钟 A 和闹钟 B)、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位 和 IWDG 复位。
唤醒延迟	复位阶段。

图 22.1.1 STM32F4 进入及退出待机模式的条件

图 22.1.1 还列出了退出待机模式的操作，从图 22.1.1 可知，我们有多种方式可以退出待机模式，包括：WKUP 引脚的上升沿、RTC 闹钟、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、外部复位(NRST 引脚)、IWDG 复位等，微控制器从待机模式退出。

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、RTC_AF1 引脚 (PC13) (如果针对入侵、时间戳、RTC 闹钟输出或 RTC 时钟校准输出进行了配置) 和 WK_UP (PA0) (如果使能了) 等引脚外，其他所有 IO 引脚都将处于高阻态。

图 22.1.1 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 22.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	VOS	Reserved		FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS		
	rw			rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw		

位 2 CWUF：将唤醒标志清零 (Clear wakeup flag)

此位始终读为 0。

0: 无操作

1: 写 1 操作 2 个系统时钟周期后将 WUF 唤醒标志清零

位 1 PDDS：深度睡眠掉电 (Power-down deepsleep)

此位由软件置 1 和清零。与 LPDS 位结合使用。

0: 器件在 CPU 进入深度睡眠时进入停止模式。调压器状态取决于 LPDS 位。

1: 器件在 CPU 进入深度睡眠时进入待机模式。

图 22.1.2 PWR_CR 寄存器各位描述

该寄存器我们只关心 bit1 和 bit2 这两个位，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 22.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
Res.															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	VOS RDY	Reserved			BRE	EWUP	Reserved Res.			BRR	PVDO	SBF	WUF		
	r				rw	rw				r	r	r	r		

位 8 **EWUP**: 使能 WKUP 引脚 (Enable WKUP pin)

此位由软件置 1 和清零。

0: WKUP 引脚用作通用 I/O。WKUP 引脚上的事件不会把器件从待机模式唤醒。

1: WKUP 用于从待机模式唤醒器件并被强制配置成输入下拉 (WKUP 引脚出现上升沿时从待机模式唤醒系统)。

注意: 此位通过系统复位进行复位。

位 0 **WUF**: 唤醒标志 (Wakeup flag)

此位由硬件置 1, 清零则只能通过 POR/PDR (上电复位/掉电复位) 或将 PWR_CR 寄存器中的 CWUF 位置 1 来实现。

0: 未发生唤醒事件

1: 收到唤醒事件, 可能来自 WKUP 引脚、RTC 闹钟 (闹钟 A 和闹钟 B)、RTC 入侵事件、RTC 时间戳事件或 RTC 唤醒事件。

注意: 如果使能 WKUP 引脚 (将 EWUP 位置 1) 时 WKUP 引脚已为高电平, 系统将检测到另一唤醒事件。

图 22.1.3 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。关于 PWR_CR 和 PWR_CSR 这两个寄存器的详细描述, 请看《STM32F4xx 中文参考手册》第 5.4.1 节和 5.4.3 节。

对于使能了 RTC 闹钟中断或 RTC 周期性唤醒等中断的时候, 进入待机模式前, 必须按如下操作处理:

- 1, 禁止 RTC 中断 (ALRAIE、ALRBIE、WUTIE、TAMPIE 和 TSIE 等)。
- 2, 清零对应中断标志位。
- 3, 清除 PWR 唤醒(WUF)标志 (通过设置 PWR_CR 的 CWUF 位实现)。
- 4, 重新使能 RTC 对应中断。
- 5, 进入低功耗模式。

在有用到 RTC 相关中断的时候, 必须按以上步骤执行之后, 才可以进入待机模式, 这个大家一定要注意, 否则可能无法唤醒。详情请参考《STM32F4xx 中文参考手册》第 5.3.6 节。

通过以上介绍, 我们了解了进入待机模式的方法, 以及设置 KEY_UP 引脚用于把 STM32F4 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 (详见《STM32F3 与 F4 系列 Cortex M4 内核编程手册》, 第 214 页 4.4.6 节), 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 STM32F4 的待机模式了，并且可以通过 KEY_UP 来唤醒 CPU，我们最终要实现这样一个功能：通过长按（3 秒）KEY_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。

22.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY_UP 按键
- 3) TFTLCD 模块

本章，我们使用了 KEY_UP 按键用于唤醒和进入待机模式。然后通过 DS0 和 TFTLCD 模块来指示程序是否在运行。这几个硬件的连接前面均有介绍。

22.3 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 stm32f4xx_pwr.c 和 stm32f4xx_pwr.h 文件。

打开 wkup.c，可以看到如下关键代码：

```
//系统进入待机模式
void Sys_Enter_Standby(void)
{
    RCC_AHB1PeriphResetCmd(0X01FF,ENABLE); //复位所有 IO 口
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE); //使能 PWR 时钟
    PWR_ClearFlag(PWR_FLAG_WU); //清除 Wake-up 标志
    PWR_WakeUpPinCmd(ENABLE); //设置 WKUP 用于唤醒
    PWR_EnterSTANDBYMode(); //进入待机模式
}

//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_Sleep(); //进入待机模式
    }
}

//PA0 WKUP 唤醒初始化
void WKUP_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;
    EXTI_InitTypeDef EXTI_InitStruct;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE); //使能 SYSCFG 时钟

    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0; //PA0
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN; //输入模式
    GPIO_InitStruct.GPIO_OType = GPIO_OType_OD;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
}
```

```

GPIO_Init(GPIOA, &GPIO_InitStructure); // 初始化
    // (检查是否是正常开机)
if(Check_WKUP() == 0)
{
    Sys_Enter_Sleep(); // 不是开机, 进入待机模式
}
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0); // PA0 连接到线 0

EXTI_InitStructure.EXTI_Line = EXTI_Line0; // LINE0
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; // 中断事件
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; // 上升沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE; // 使能 LINE0
EXTI_Init(&EXTI_InitStructure); // 配置

NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn; // 外部中断 0
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02; // 抢占优先级 2
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02; // 响应优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // 使能外部中断通道
NVIC_Init(&NVIC_InitStructure); // 配置 NVIC
}

```

该部分代码比较简单，我们在这里说明两点：

1，在 void Sys_Enter_Sleep(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就可，这样才能达到最低功耗！然后我们调用函数 RCC_APB1PeriphClockCmd 来使能 PWR 时钟，调用函数 PWR_WakeUpPinCmd 用来设置 WK_UP 引脚作为唤醒源。最后调用 PWR_EnterSTANDBYMode 函数进入待机模式。

2，在 void WKUP_Init(void) 函数里面，我们首先要使能 GPIOA 时钟，同时因为我们要使用到外部中断，所以必须先使能 SYSCFG 时钟。然后对 GPIOA 初始化位下拉输入。同时调用函数 SYSCFG_EXTILineConfig 配置 GPIOA.0 连接到中断线 0。最后初始化 EXTI 中断线以及 NVIC 中断优先级。这上面的步骤实际上跟我们之前的外部中断实验知识是一样的，所以不理解的地方大家可以翻到外部中断实验章节看看。在上面初始化的过程中，我们还先判断 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 秒开机**，才能看到 DS0 闪烁。

3，在中断服务函数 EXTI0_IRQHandler 内，我们通过调用函数 Check_WKUP 来判断 WK_UP 按下的时间长短，来决定是否进入待机模式，如果按下时间超过 3 秒，则进入待机，否则退出中断。

wkup.h 部分代码比较简单，我们就不多说了。最后我们看看 main 函数内容如下：

```

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 设置系统中断优先级分组 2
    delay_init(168); // 初始化延时函数
    uart_init(115200); // 初始化串口波特率为 115200
}

```

```
LED_Init();          //初始化 LED
WKUP_Init();         //待机唤醒初始化
LCD_Init();          //液晶初始化
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
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,"2014/5/6");
LCD_ShowString(30,130,200,16,16,"WK_UP:Stanby/WK_UP");
while(1)
{
    LED0=!LED0;  delay_ms(250);//延时 250ms
}
}
```

这里我们先初始化 LED 和 WK_UP 按键（通过 WKUP_Init() 函数初始化），如果检测到有长按 WK_UP 按键 3 秒以上，则开机，并执行 LCD 初始化，在 LCD 上面显示一些内容，如果没有长按，则在 WKUP_Init 里面，调用 Sys_Enter_Standby 函数，直接进入待机模式了。

开机后，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，如果按下时间超过 3 秒，则进入待机，否则退出中断，继续执行 main 函数的死循环等待，同时不停的取反 LED0，让红灯闪烁。

代码部分就介绍到这里，大家记住[下载代码后，一定要长按 WK_UP 按键，来开机，否则将直接进入待机模式，无任何现象。](#)

22.4 下载与测试

在代码编译成功之后，下载代码到探索者 STM32F4 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，开发板检测不到 WK_UP 的持续按下（3 秒以上），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，液晶也会显示一些内容。然后再长按 WK_UP，DS0 会灭掉，液晶灭掉，程序再次进入待机模式。

第二十三章 ADC 实验

本章我们将向大家介绍 STM32F4 的 ADC 功能。在本章中，我们将使用 STM32F4 的 ADC1 通道 5 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：

- 23.1 STM32F4 ADC 简介
- 23.2 硬件设计
- 23.3 软件设计
- 23.4 下载验证

23.1 STM32F4 ADC 简介

STM32F4xx 系列一般都有 3 个 ADC，这些 ADC 可以独立使用，也可以使用双重/三重模式（提高采样率）。STM32F4 的 ADC 是 12 位逐次逼近型的模拟数字转换器。它有 19 个通道，可测量 16 个外部源、2 个内部源和 Vbat 通道的信号。这些通道的 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/低阀值。

STM32F407ZGT6 包含有 3 个 ADC。STM32F4 的 ADC 最大的转换速率为 2.4Mhz，也就是转换时间为 1us (在 ADCCLK=36M,采样周期为 3 个 ADC 时钟下得到)，不要让 ADC 的时钟超过 36M，否则将导致结果准确度下降。

STM32F4 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。

STM32F4 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《STM32F4xx 中文参考手册》第 250 页，第 11.3.3 节。

STM32F4 的 ADC 可以进行很多种不同的转换模式，这些模式在《STM32F4xx 中文参考手册》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。

STM32F4 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），这时 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 23.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				OVRIE	RES		AWDEN	JAWDEN	Reserved						
				rw	rw	rw	rw	rw							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]		JDISCEN	DISC EN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]					
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 23.1.1 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《STM32F4xx 中文参考手册》第 11.13.2 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 JEDECIE 中断。

ADC_CR1[25:24]用于设置 ADC 的分辨率，详细的对应关系如图 23.1.2 所示：

位 25:24 RES[1:0]: 分辨率 (Resolution)

通过软件写入这些位可选择转换的分辨率。

- 00: 12 位 (15 ADCCLK 周期)
- 01: 10 位 (13 ADCCLK 周期)
- 10: 8 位 (11 ADCCLK 周期)
- 11: 6 位 (9 ADCCLK 周期)

图 23.1.2 ADC 分辨率选择

本章我们使用 12 位分辨率，所以设置这两个位为 0 就可以了。接着我们介绍 ADC_CR2，该寄存器的各位描述如图 23.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
reserved	SWST ART		EXTEN		EXTSEL[3:0]				reserved	JSWST ART	JEXTEN		JEXTSEL[3:0]			
	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	
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON	
				rw	rw	rw	rw							rw	rw	

图 23.1.3 ADC_CR2 寄存器各位描述

该寄存器我们也只针对性的介绍一些位：ADON 位用于开关 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。ALIGN 用于设置数据对齐，我们使用右对齐，该位设置为 0。

EXTEN[1:0]用于规则通道的外部触发使能设置，详细的设置关系如图 23.1.4 所示：

位 29:28 **EXTEN:** 规则通道的外部触发使能 (External trigger enable for regular channels)

通过软件将这些位置 1 和清零可选择外部触发极性和使能规则组的触发。

00: 禁止触发检测

01: 上升沿上的触发检测

10: 下降沿上的触发检测

11: 上升沿和下降沿上的触发检测

图 23.1.4 ADC 规则通道外部触发使能设置

我们这里使用的是软件触发，即不使用外部触发，所以设置这 2 个位为 0 即可。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。

第二个要介绍的是 ADC 通用控制寄存器 (ADC_CCR)，该寄存器各位描述如图 23.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved		ADCPRE			
rw	rw							rw	rw			rw	rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA[1:0]			DDS	Res.	DELAY[3:0]				Reserved		MULTI[4:0]				
rw	rw	rw			rw	rw	rw	rw			rw	rw	rw	rw	rw

图 23.1.5 ADC_CCR 寄存器各位描述

该寄存器我们也只针对性的介绍一些位：TSVREFE 位是内部温度传感器和 Vrefint 通道使能位，内部温度传感器我们将在下一章介绍，这里我们直接设置为 0。ADCPRE[1:0]用于设置 ADC 输入时钟分频，00~11 分别对应 2/4/6/8 分频，STM32F4 的 ADC 最大工作频率是 36Mhz，而 ADC 时钟(ADCCLK)来自 APB2，APB2 频率一般是 84Mhz，所以我们一般设置 ADCPRE=01，即 4 分频，这样得到 ADCCLK 频率为 21Mhz。MULTI[4:0]用于多重 ADC 模式选择，详细的设置关系如图 23.1.6 所示：

位 4:0 **MULTI[4:0]:** 多重 ADC 模式选择 (Multi ADC mode selection)

通过软件写入这些位可选择操作模式。

所有 ADC 均独立：

00000: 独立模式

00001 到 01001: 双重模式，ADC1 和 ADC2 一起工作，ADC3 独立

00001: 规则同时 + 注入同时组合模式

00010: 规则同时 + 交替触发组合模式

00011: Reserved

00101: 仅注入同时模式

00110: 仅规则同时模式

仅交错模式

01001: 仅交替触发模式

10001 到 11001: 三重模式：ADC1、ADC2 和 ADC3 一起工作

10001: 规则同时 + 注入同时组合模式

10010: 规则同时 + 交替触发组合模式

10011: Reserved

10101: 仅注入同时模式

10110: 仅规则同时模式

仅交错模式

11001: 仅交替触发模式

其它所有组合均需保留且不允许编程

图 23.1.6 多重 ADC 模式选择设置

本章我们仅用了 ADC1 (独立模式)，并没用到多重 ADC 模式，所以设置这 5 个位为 0 即

可。

第三个要介绍的是 ADC 采样时间寄存器 (ADC_SMPR1 和 ADC_SMPR2), 这两个寄存器用于设置通道 0~18 的采样时间, 每个通道占用 3 个位。ADC_SMPR1 的各位描述如图 23.1.7 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]		
				rw	rw	rw									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_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 31:27 保留, 必须保持复位值。

位 26:0 **SMPx[2:0]**: 通道 X 采样时间选择 (Channel x sampling time selection)

通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间, 通道选择位必须保持不变。

注意: 000: 3 个周期	100: 84 个周期
001: 15 个周期	101: 112 个周期
010: 28 个周期	110: 144 个周期
011: 56 个周期	111: 480 个周期

图 23.1.7 ADC_SMPR1 寄存器各位描述

ADC_SMPR2 的各位描述如下图 23.1.8 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]		
				rw	rw	rw									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_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, 必须保持复位值。

位 29:0 **SMPx[2:0]**: 通道 X 采样时间选择 (Channel x sampling time selection)

通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间, 通道选择位必须保持不变。

注意: 000: 3 个周期	100: 84 个周期
001: 15 个周期	101: 112 个周期
010: 28 个周期	110: 144 个周期
011: 56 个周期	111: 480 个周期

图 23.1.8 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:

$$T_{covn} = \text{采样时间} + 12 \text{ 个周期}$$

其中: T_{covn} 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 $\text{ADCCLK}=21\text{Mhz}$ 的时候, 并设置 3 个周期的采样时间, 则得到: $T_{covn}=3+12=15$ 个周期 = $0.71\mu\text{s}$ 。

第四个要介绍的是 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 23.1.9 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										L[3:0]		SQ16[4:1]			
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
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

位 31:24 保留，必须保持复位值。

位 23:20 L[3:0]: 规则通道序列长度 (Regular channel sequence length)

通过软件写入这些位可定义规则通道转换序列中的转换总数。

0000: 1 次转换

0001: 2 次转换

...

1111: 16 次转换

位 19:15 SQ16[4:0]: 规则序列中的第十六次转换 (16th conversion in regular sequence)

通过软件写入这些位，并将通道编号 (0~18) 分配为转换序列中的第十六次转换。

位 14:10 SQ15[4:0]: 规则序列中的第十五次转换 (15th conversion in regular sequence)

位 9:5 SQ14[4:0]: 规则序列中的第十四次转换 (14th conversion in regular sequence)

位 4:0 SQ13[4:0]: 规则序列中的第十三次转换 (13th conversion in regular sequence)

图 23.1.9 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 (0~18)。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，至于 SQ1 里面哪个通道，完全由用户自己设置，通过 ADC_SQR3 的最低 5 位（也就是 SQ1）设置。

第五个要介绍的是 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 23.1.10:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留，必须保持复位值。

位 15:0 DATA[15:0]: 规则数据 (Regular data)

这些位为只读。它们包括来自规则通道的转换结果。数据有左对齐和右对齐两种方式。

图 23.1.10 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 23.1.11 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															
OVR	STRT	JSTRT	JEOC	EOC	AWD										
rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0										

图 23.1.11 ADC_SR 寄存器各位描述

这里我们仅介绍将要用到的是 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，如果该位为 1，则表示转换完成了，就可以从 ADC_DR 中读取转换结果，否则等待转换完成。

至此，本章要用到的 ADC 相关寄存器全部介绍完毕了，对于未介绍的部分，请大家参考《STM32F4xx 中文参考手册》第 11 章相关章节。通过以上介绍，我们了解了 STM32F4 的单次转换模式下的相关设置，接下来我们介绍使用库函数来设置 ADC1 的通道 5 来进行 AD 转换的步骤，这里需要说明一下，使用到的库函数分布在 stm32f4xx_adc.c 文件和 stm32f4xx_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 PA5 为模拟输入。

STM32F407ZGT6 的 ADC1 通道 5 在 PA5 上，所以，我们先要使能 GPIOA 的时钟，然后设置 PA5 为模拟输入。同时我们要把 PA5 复用为 ADC，所以我们要使能 ADC1 时钟。

这里特别要提醒，对于 IO 口复用为 ADC 我们要设置模式为模拟输入，而不是复用功能，也不需要调用 GPIO_PinAFConfig 函数来设置引脚映射关系。

使能 GPIOA 时钟和 ADC1 时钟都很简单，具体方法为：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 时钟  
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); //使能 ADC1 时钟
```

初始化 GPIOA5 为模拟输入，方法也多次讲解，关键代码为：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //模拟输入
```

这里需要说明一下，ADC 的通道与引脚的对应关系在 STM32F4 的数据手册可以查到，我们这里使用 ADC1 的通道 5，在数据手册中的表格为：

PA5	I/O	TTa	(4)	SPI1_SCK/ OTG_HS_ULPI_CK / TIM2_CH1_ETR/ TIM8_CH1N/ EVENTOUT	ADC12_IN5/DAC_OU T2
-----	-----	-----	-----	---	------------------------

表 23.1.12 ADC1 通道 5 对应引脚查看表

这里我们把 ADC1~ADC3 的引脚与通道对应关系列出来，16 个外部源的对应关系如下表：

通道号	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	PF3
通道 10	PC0	PC0	PC0
通道 11	PC1	PC1	PC1
通道 12	PC2	PC2	PC2
通道 13	PC13	PC13	PC13
通道 14	PC4	PC4	PF4

通道 15	PC5	PC5	PF5
-------	-----	-----	-----

表 23.1.13 ADC1~ADC3 引脚对应关系表

2) 设置 ADC 的通用控制寄存器 CCR，配置 ADC 输入时钟分频，模式为独立模式等。

在库函数中，初始化 CCR 寄存器是通过调用 ADC_CommonInit 来实现的：

```
void ADC_CommonInit(ADC_CommonInitTypeDef* ADC_CommonInitStruct)
```

这里我们不再逐处初始化结构体成员变量，而是直接看实例。初始化实例为：

```
ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;//独立模式  
ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;  
ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;  
ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div4;  
ADC_CommonInit(&ADC_CommonInitStructure);//初始化
```

第一个参数 ADC_Mode 用来设置是独立模式还是多重模式，这里我们选择独立模式。

第二个参数 ADC_TwoSamplingDelay 用来设置两个采样阶段之间的延迟周期数。这个比较好理解。取值范围为：ADC_TwoSamplingDelay_5Cycles~ADC_TwoSamplingDelay_20Cycles。

第三个参数 ADC_DMAAccessMode 是 DMA 模式禁止或者使能相应 DMA 模式。

第四个参数 ADC_Prescaler 用来设置 ADC 预分频器。这个参数非常重要，这里我们设置分频系数为 4 分频 ADC_Prescaler_Div4，保证 ADC1 的时钟频率不超过 36MHz。

3) 初始化 ADC1 参数，设置 ADC1 的转换分辨率，转换方式，对齐方式，以及规则序列等相关信息。

在设置完通用控制参数之后，我们就可以开始 ADC1 的相关参数配置了，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。具体的使用函数为：

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

初始化实例为：

```
ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;//12 位模式  
ADC_InitStruct.ADC_ScanConvMode = DISABLE;//非扫描模式  
ADC_InitStruct.ADC_ContinuousConvMode = DISABLE;//关闭连续转换  
ADC_InitStruct.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;  
//禁止触发检测，使用软件触发  
ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right;//右对齐  
ADC_InitStruct.ADC_NbrOfConversion = 1;//1 个转换在规则序列中  
ADC_Init(ADC1, &ADC_InitStruct);//ADC 初始化
```

第一个参数 ADC_Resolution 用来设置 ADC 转换分辨率。取值范围为：ADC_Resolution_6b，ADC_Resolution_8b，ADC_Resolution_10b 和 ADC_Resolution_12b。

第二个参数 ADC_ScanConvMode 用来设置是否打开扫描模式。这里我们设置单次转换所以不打开扫描模式，值为 DISABLE。

第三个参数 ADC_ContinuousConvMode 用来设置是单次转换模式还是连续转换模式，这里我们是单次，所以关闭连续转换模式，值为 DISABLE。

第三个参数 ADC_ExternalTrigConvEdge 用来设置外部通道的触发使能和检测方式。这里我们直接禁止触发检测，使用软件触发。还可以设置为上升沿触发检测，下降沿触发检测以及上升沿和下降沿都触发检测。

第四个参数 ADC_DataAlign 用来设置数据对齐方式。取值范围为右对齐 ADC_DataAlign_Right 和左对齐 ADC_DataAlign_Left。

第五个参数 ADC_NbrOfConversion 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。

实际上还有个参数 ADC_ExternalTrigConv 是用来为规则组选择外部事件。因为我们前面配置的是软件触发，所以这里我们可以不用配置。如果选择其他触发方式方式，这里需要配置。

4) 开启 AD 转换器。

在设置完了以上信息后，我们就开启 AD 转换器了（通过 ADC_CR2 寄存器控制）。

```
ADC_Cmd(ADC1, ENABLE); //开启 AD 转换器
```

5) 读取 ADC 值。

在上面的步骤完成后，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，然后启动 ADC 转换。在转换结束后，读取转换结果值值就是了。

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 480，所以设置为：

```
ADC-RegularChannelConfig(ADC1, ADC_Channel_5, 1, ADC_SampleTime_480Cycles );
```

软件开启 ADC 转换的方法是：

```
ADC_SoftwareStartConvCmd(ADC1); //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 ADC1 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，探索者 STM32F4 开发板使用的是 STM32F407ZGT6，该芯片只有 Vref+参考电压引脚，Vref+的输入范围为：1.8~VDDA。探索者 STM32F4 开发板通过 P7 端口，来设置 Vref+的参考电压，默认的我们是通过跳线帽将 ref+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+上就 OK 了（注意要共地）。另外，对于还有 Vref-引脚的 STM32F4 芯片，直接就近将 Vref-接 VSSA 就可以了。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 STM32F4 的 ADC1 来执行 AD 转换操作了。

23.2 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 STM32F4 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 5 (PA5) 来读取外部电压值，探索者 STM32F4 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 P12 的 ADC 插针上（与 PA5 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）。

23.3 软件设计

打开实验工程可以发现，我们在 FWLIB 分组下面新增了 stm32f4xx_adc.c 源文件，同时会引入对应的头文件 stm32f4xx_adc.h。ADC 相关的库函数和宏定义都分布在这两个文件中。同时，我们在 HARDWARE 分组下面新建了 adc.c，也引入了对应的头文件 adc.h。这两个文件是我们编写的 adc 相关的初始化函数和操作函数。

打开 adc.c，代码如下：

```
//初始化 ADC
//这里我们仅以规则通道为例
void Adc_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    ADC_CommonInitTypeDef ADC_CommonInitStruct;
    ADC_InitTypeDef      ADC_InitStruct;

    //①开启 ADC 和 GPIO 相关时钟和初始化 GPIO
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); //使能 ADC1 时钟

    //先初始化 ADC1 通道 5 IO 口
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_5; //PA5 通道 5
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AN; //模拟输入
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL; //不带上下拉
    GPIO_Init(GPIOA, &GPIO_InitStruct); //初始化

    RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1, ENABLE); //ADC1 复位
    RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1, DISABLE); //复位结束

    //②初始化通用配置
    ADC_CommonInitStruct.ADC_Mode = ADC_Mode_Independent; //独立模式
    ADC_CommonInitStruct.ADC_TwoSamplingDelay =
        ADC_TwoSamplingDelay_5Cycles; //两个采样阶段之间的延迟 5 个时钟
    ADC_CommonInitStruct.ADC_DMAAccessMode =
        ADC_DMAAccessMode_Disabled; //DMA 失能
    ADC_CommonInitStruct.ADC_Prescaler = ADC_Prescaler_Div4; //预分频 4 分频。
    //ADCCLK=PCLK2/4=84/4=21Mhz, ADC 时钟最好不要超过 36Mhz
    ADC_CommonInit(&ADC_CommonInitStruct); //初始化

    //③初始化 ADC1 相关参数
```

```

ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;//12 位模式
ADC_InitStructure.ADC_ScanConvMode = DISABLE;//非扫描模式
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;//关闭连续转换
ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
                                //禁止触发检测，使用软件触发
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//右对齐
ADC_InitStructure.ADC_NbrOfConversion = 1;//1 个转换在规则序列中
ADC_Init(ADC1, &ADC_InitStructure);//ADC 初始化

//④开启 ADC 转换
ADC_Cmd(ADC1, ENABLE);//开启 AD 转换器
}

//获得 ADC 值
//ch:通道值 0~16: ch: @ref ADC_channels
//返回值:转换结果
u16 Get_Adc(u8 ch)
{
    //设置指定 ADC 的规则组通道，一个序列，采样时间
    ADC-RegularChannelConfig(ADC1, ch, 1, ADC_SampleTime_480Cycles );
    ADC_SoftwareStartConv(ADC1);      //使能指定的 ADC1 的软件转换启动功能
    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
}
//获取通道 ch 的转换值，取 times 次,然后平均
//ch:通道编号  times:获取次数
//返回值:通道 ch 的 times 次转换结果平均值
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，我们用标号①~④标示出来步骤。这里我们仅开通了 1 个通道，即通道 5。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 5 上的 ADC 值，就可以通过 Get_Adc(ADC_Channel_5) 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。

头文件 adc.h 代码比较简单，主要是三个函数申明。接下来我们看看 main 函数内容：

```

int main(void)
{
    u16 adcx;  float temp;

```

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
delay_init(168); //初始化延时函数
uart_init(115200); //初始化串口波特率为 115200
LED_Init(); //初始化 LED
LCD_Init(); //初始化 LCD 接口
Adc_Init(); //初始化 ADC
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"ADC TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/6");
POINT_COLOR=BLUE; //设置字体为蓝色
LCD_ShowString(30,130,200,16,16,"ADC1_CH5_VAL:");
LCD_ShowString(30,150,200,16,16,"ADC1_CH5_VOL:0.000V"); //这里显示了小数点
while(1)
{
    adcx=Get_Adc_Average(ADC_Channel_5,20); //获取通道 5 的转换值，20 次取平均
    LCD_ShowxNum(134,130,adcx,4,16,0); //显示 ADCC 采样后的原始值
    temp=(float)adcx*(3.3/4096); //获取计算后的带小数的实际电压值，比如 3.1111
    adcx=temp; //赋值整数部分给 adcx 变量，因为 adcx 为 u16 整型
    LCD_ShowxNum(134,150,adcx,1,16,0); //显示电压值的整数部分
    temp-=adcx; //把已经显示的整数部分去掉，留下小数部分，比如 3.1111-3=0.1111
    temp*=1000; //小数部分乘以 1000，例如：0.1111 就转换为 111.1，保留三位小数。
    LCD_ShowxNum(150,150,temp,3,16,0X80); //显示小数部分
    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 5 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。这里关于最后的 ADC 值的显示我们说明一下，首先我们在液晶固定位置显示了小数点，然后后面计算步骤中，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就在液晶上面显示转换结果的整数和小数部分。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 23.4.1 所示：

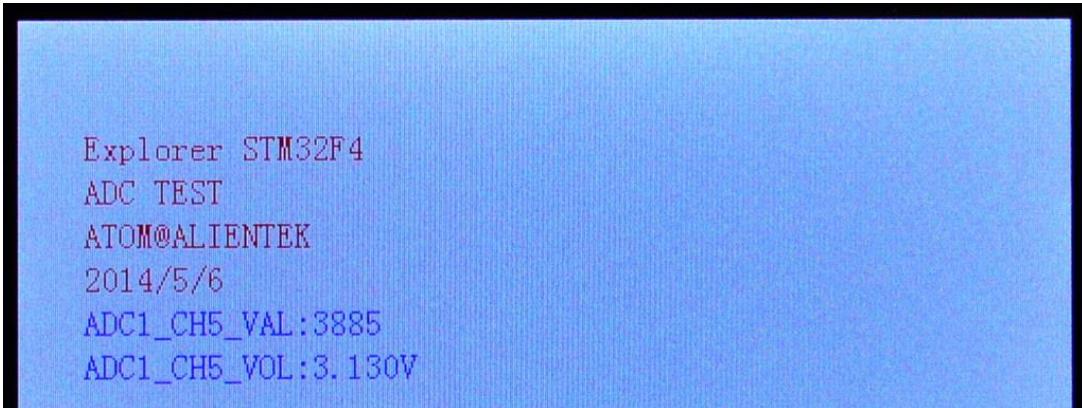


图 23.4.1 ADC 实验测试图

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 R64 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！

特别注意：STM32F4 的 ADC 精度貌似不怎么好，ADC 引脚直接接 GND，都可以读到十几的数值，相比 STM32F103 来说，要差了一些，在使用的时候，请大家注意下这个问题。

通过这一章的学习，我们了解了 STM32F4 ADC 的使用，但这仅仅是 STM32F4 强大的 ADC 功能的一小点应用。STM32F4 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 STM32F4 的 ADC，相信会给你以后的开发带来方便。

第二十四章 内部温度传感器实验

本章我们将向大家介绍 STM32F4 的内部温度传感器。在本章中，我们将使用 STM32F4 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：

- 24.1 STM32F4 内部温度传感器简介
- 24.2 硬件设计
- 24.3 软件设计
- 24.4 下载验证

24.1 STM32F4 内部温度传感器简介

STM32F4 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 ADC1_IN16 (STM32F40xx/F41xx 系列) 或 ADC1_IN18 (STM32F42xx/F43xx 系列) 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。STM32F4 的内部温度传感器支持的温度范围为：-40~125 度。精度为±1.5℃左右。

STM32F4 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部温度传感器通道就差不多了。关于 ADC 的设置，我们在上一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 STM32F4 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CCR 的 TSVREFE 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，STM32F407ZGT6 的内部温度传感器固定的连接在 ADC1 的通道 16 上，所以，我们在设置好 ADC1 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：

$$T (\text{ }^{\circ}\text{C}) = \{ (V_{\text{sense}} - V_{25}) / \text{Avg_Slope} \} + 25$$

上式中：

V_{25} = V_{sense} 在 25 度时的数值 (典型值为：0.76)。

Avg_Slope =温度与 V_{sense} 曲线的平均斜率 (单位为 mv/°C 或 uv/°C) (典型值为 2.5mV/°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 STM32F4 内部温度传感器使用的步骤了，如下：

1) 设置 ADC1，开启内部温度传感器。

关于如何设置 ADC1，上一章已经介绍了，我们采用与上一章一样的设置，这里我们只要增加使能内部温度传感器这一句就可以了。方法为：

```
ADC_TempSensorVrefintCmd(ENABLE); //使能内部温度传感器
```

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了。具体方法跟上一讲是一样的。

24.2 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) 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 STM32F4 内部，不需要外部设置，我们只需要软件设置就 OK 了。

24.3 软件设计

打开本章实验工程中可以看到，我们并没有增加任何文件，而是在 adc.c 文件修改和添加了一些函数，adc.c 文件中 Adc_Init 函数内容如下：

```
void Adc_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_CommonInitTypeDef ADC_CommonInitStructure;
    ADC_InitTypeDef      ADC_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); //使能 ADC1 时钟

    //先初始化 IO 口
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //模拟输入
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; // 下拉
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化

    RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1, ENABLE); //ADC1 复位
    RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1, DISABLE); //复位结束
```

ADC_TempSensorVrefintCmd(ENABLE); //使能内部温度传感器

```
ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent; //独立模式
ADC_CommonInitStructure.ADC_TwoSamplingDelay =
                                ADC_TwoSamplingDelay_5Cycles;
ADC_CommonInitStructure.ADC_DMAAccessMode =
                                ADC_DMAAccessMode_Disabled; /
ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div4;
ADC_CommonInit(&ADC_CommonInitStructure);
```

```
ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b; //12 位模式
ADC_InitStructure.ADC_ScanConvMode = DISABLE; //非扫描模式
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //右对齐
```

```
ADC_InitStructure.ADC_NbrOfConversion = 1;//1 个转换在规则序列中
ADC_Init(ADC1, &ADC_InitStructure);
```

```
ADC_Cmd(ADC1, ENABLE);//开启 AD 转换器
}
```

这部分代码与上一章的 Adc_Init 代码几乎一摸一样，我们仅仅在里面增加了如下一句代码：

```
ADC_TempSensorVrefintCmd(ENABLE);//使能内部温度传感器
```

这句我们就是使能内部温度传感器。然后在 adc.c 里面添加了获取温度函数：Get_Temprate，该函数代码如下：

```
//得到温度值
//返回值:温度值(扩大了 100 倍,单位:°C.)
short Get_Temprate(void)
{
    u32 adcx; short result;
    double temperate;
    adcx=Get_Adc_Average(ADC_Channel_16,20); //读取通道 16,20 次取平均
    temperate=(float)adcx*(3.3/4096); //电压值
    temperate=(temperate-0.76)/0.0025+25; //转换为温度值
    result=temperate*=100; //扩大 100 倍.
    return result;
}
```

该函数读取 ADC_Channel_16 通道（即通道 16）采集到的电压值，并根据前面的计算公式，计算出当前温度，然后，返回扩大了 100 倍的温度值。

adc.h 代码比较简单，我们就不多说了。接下来，我们看看 main 函数如下：

```
int main(void)
{
    short temp;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //液晶初始化
    Adc_Init(); //内部温度传感器 ADC 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"Temperature TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/6");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,140,200,16,16,"TEMPERATE: 00.00C");//固定位置显示小数点
```

```
while(1)
{
    temp=Get_Temprate(); //得到温度值
    if(temp<0)
    {
        temp=-temp;
        LCD_ShowString(30+10*8,140,16,16,16,"-"); //显示负号
    }else LCD_ShowString(30+10*8,140,16,16,16," "); //无符号

    LCD_ShowxNum(30+11*8,140,temp/100,2,16,0); //显示整数部分
    LCD_ShowxNum(30+14*8,140,temp%100,2,16,0); //显示小数部分

    LED0=!LED0; delay_ms(250);
}
}
```

这里同上一章的主函数也大同小异，这里，我们通过 Get_Temprate 函数读取温度值，并通过 TFTLCD 模块显示出来。

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。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 24.4.1 所示：

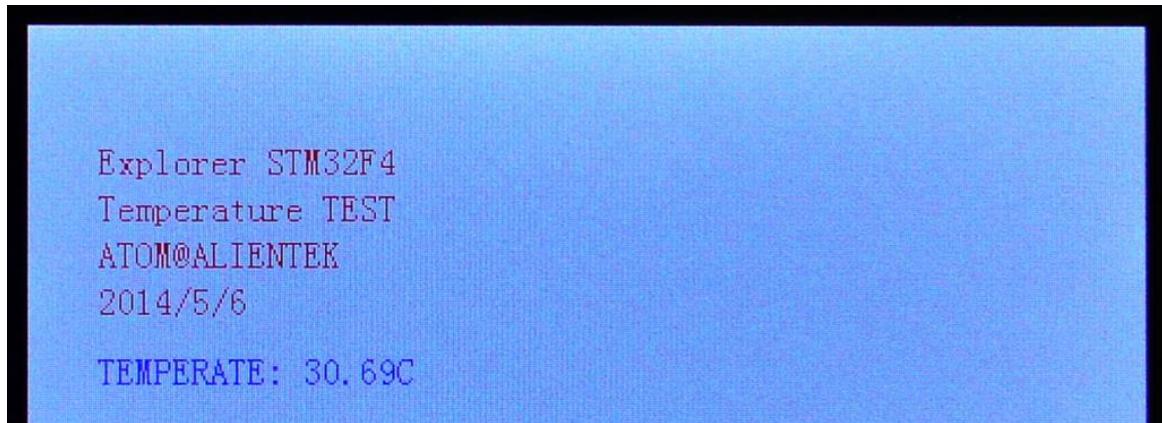


图 24.4.1 内部温度传感器实验测试图

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，而且貌似准确度也不怎么好，所以一般会比实际温度偏高）？

第二十五章 光敏传感器实验

本章我们将向大家介绍探索者 STM32F4 开发板自带的一个光敏传感器，本章还是要用到 ADC 采集，通过 ADC 采集电压，获取光敏传感器的电阻变化，从而得出环境光线的变化，并在 TFTLCD 上面显示出来。本章将分为如下几个部分：

- 25.1 光敏传感器简介
- 25.2 硬件设计
- 25.3 软件设计
- 25.4 下载验证

25.1 光敏传感器简介

光敏传感器是最常见的传感器之一，它的种类繁多，主要有：光电管、光电倍增管、光敏电阻、光敏三极管、太阳能电池、红外线传感器、紫外线传感器、光纤式光电传感器、色彩传感器、CCD 和 CMOS 图像传感器等。光传感器是目前产量最多、应用最广的传感器之一，它在自动控制和非电量电测技术中占有非常重要的地位。

光敏传感器是利用光敏元件将光信号转换为电信号的传感器，它的敏感波长在可见光波长附近，包括红外线波长和紫外线波长。光传感器不只局限于对光的探测，它还可以作为探测元件组成其他传感器，对许多非电量进行检测，只要将这些非电量转换为光信号的变化即可。

探索者 STM32F4 开发板板载了一个光敏二极管（光敏电阻），作为光敏传感器，它对光的变化非常敏感。光敏二极管也叫光电二极管。光敏二极管与半导体二极管在结构上是类似的，其管芯是一个具有光敏特征的 PN 结，具有单向导电性，因此工作时需加上反向电压。无光照时，有很小的饱和反向漏电流，即暗电流，此时光敏二极管截止。当受到光照时，饱和反向漏电流大大增加，形成光电流，它随入射光强度的变化而变化。当光线照射 PN 结时，可以使 PN 结中产生电子—空穴对，使少数载流子的密度增加。这些载流子在反向电压下漂移，使反向电流增加。因此可以利用光照强弱来改变电路中的电流。

利用这个电流变化，我们串接一个电阻，就可以转换成电压的变化，从而通过 ADC 读取电压值，判断外部光线的强弱。

本章，我们利用 ADC3 的通道 5 (PF7) 来读取光敏二极管电压的变化，从而得到环境光线的变化，并将得到的光线强度，显示在 TFTLCD 上面。关于 ADC 的介绍，前面两章已经有详细介绍了，这里我们就不再细说了。

25.2 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) 光敏传感器

前三个之前均有介绍，光敏传感器与 STM32F4 的连接如图 25.2.1 所示：

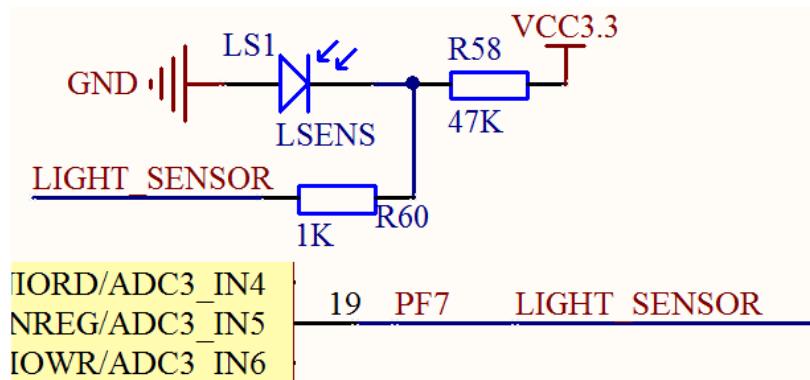


图 25.2.1 光敏传感器与 STM32F4 连接示意图

图中，LS1 是光敏二极管（实物在开发板摄像头接口右侧），R58 为其提供反向电压，当环境光线变化时，LS1 两端的电压也会随之改变，从而通过 ADC3_IN5 通道，读取 LIGHT_SENSOR (PF7) 上面的电压，即可得到环境光线的强弱。光线越强，电压越低，光线越暗，电压越高。

25.3 软件设计

打开本章实验工程可以看到，在固件库文件中，我们跟上一讲的实验是一样的，添加了 adc 相关的库函数文件 `stm32f4xx_adc.c` 和对应头文件的支持。同时，我们在 HARDWARE 分组下新建了 `adc3.c` 和 `lsens.c` 源文件，以及包含了它们对应的头文件。因为本实验我们主要是使用 ADC3 去测量光敏二极管的电压变化，所以大部分知识我们在前面 ADC 实验部分都有所讲解，这里我们就略带而过。打开 `lsens.c`，代码如下：

```
//初始化光敏传感器
void Lsens_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE); //使能 GPIOF 时钟

    //先初始化 ADC3 通道 7IO 口
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7; //PA7 通道 7
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //模拟输入
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; //不带上下拉
    GPIO_Init(GPIOF, &GPIO_InitStructure); //初始化
    Adc3_Init(); //初始化 ADC3
}

//读取 Light Sens 的值
//0~100:0,最暗;100,最亮
u8 Lsens_Get_Val(void)
{
    u32 temp_val=0;
    u8 t;
    for(t=0;t<LSENS_READ_TIMES;t++)

```

```

{
    temp_val+=Get_Adc3(ADC_Channel_5); //读取 ADC 值,通道 5
    delay_ms(5);
}
temp_val/=LSENS_READ_TIMES;//得到平均值
if(temp_val>4000)temp_val=4000;
return (u8)(100-(temp_val/40));
}

```

这里就 2 个函数，其中：Lsens_Init 用于初始化光敏传感器，其实就是初始化 PF7 为模拟输入，然后通过 Adc3_Init 函数初始化 ADC3 的通道 ADC_Channel_5。Lsens_Get_Val 函数用于获取当前光照强度，该函数通过 Get_Adc3 得到通道 ADC_Channel_5 转换的电压值，经过简单量化后，处理成 0~100 的光强值。0 对应最暗，100 对应最亮。

接下来我们看看 adc3.c 源文件代码：

```

//初始化 ADC3
//这里我们仅以规则通道为例

void Adc3_Init(void)
{
    ADC_CommonInitTypeDef ADC_CommonInitStructure;
    ADC_InitTypeDef      ADC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE); //使能 ADC3 时钟

    RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC3,ENABLE); //ADC3 复位
    RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC3,DISABLE); //复位结束

    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;//独立模式
    ADC_CommonInitStructure.ADC_TwoSamplingDelay =
        ADC_TwoSamplingDelay_5Cycles;//两个采样阶段之间的延迟 5 个时钟
    ADC_CommonInitStructure.ADC_DMAAccessMode =
        ADC_DMAAccessMode_Disabled;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div4;//预分频 4 分频。
    ADC_CommonInit(&ADC_CommonInitStructure);//初始化

    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;//12 位模式
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;//非扫描模式
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;//关闭连续转换
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
                                //禁止触发检测，使用软件触发
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//右对齐
    ADC_InitStructure.ADC_NbrOfConversion = 1;//1 个转换在规则序列中
}

```

```

ADC_Init(ADC3, &ADC_InitStructure); //ADC 初始化
ADC_Cmd(ADC3, ENABLE); //开启 AD 转换器
}
//获得 ADC 值
//ch:通道值 0~16 ADC_Channel_0~ADC_Channel_16
//返回值:转换结果
u16 Get_AdC3(u8 ch)
{
    //设置指定 ADC 的规则组通道, 一个序列, 采样时间
    ADC-RegularChannelConfig(ADC3, ch, 1, ADC_SampleTime_480Cycles );
    ADC_SoftwareStartConv(ADC3); //使能指定的 ADC3 的软件转换启动功能
    while(!ADC_GetFlagStatus(ADC3, ADC_FLAG_EOC )); //等待转换结束
    return ADC_GetConversionValue(ADC3); //返回最近一次 ADC3 规则组的转换结果
}

```

这里, Adc3_Init 函数几乎和 ADC_Init 函数一模一样, 这里我们设置了 ADC3_CH5 的相关参数, 但是没有设置对应 IO 为模拟输入, 因为这个在 Lsens_Init 函数已经实现。Get_AdC3 用于获取 ADC3 某个通道的转换结果。

因为我们前面对 ADC 有了详细的讲解, 所以本章实验源码部分讲解就比较简单。接下来我们看看主函数:

```

int main(void)
{
    u8 adcx;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //初始化 LCD
    Lsens_Init(); //初始化光敏传感器
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"LSENS TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/7");
    POINT_COLOR=BLUE; //设置字体为蓝色
    LCD_ShowString(30,130,200,16,16,"LSENS_VAL:");
    while(1)
    {
        adcx=Lsens_Get_Val();
        LCD_ShowxNum(30+10*8,130,adcx,3,16,0); //显示 ADC 的值
        LED0=!LED0;
        delay_ms(250);
    }
}

```

{}

此部分代码也比较简单，初始化各个外设之后，进入死循环，通过 Lsens_Get_Val 获取光敏传感器得到的光强值（0~100），并显示在 TFTLCD 上面。

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。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 25.4.1 所示：

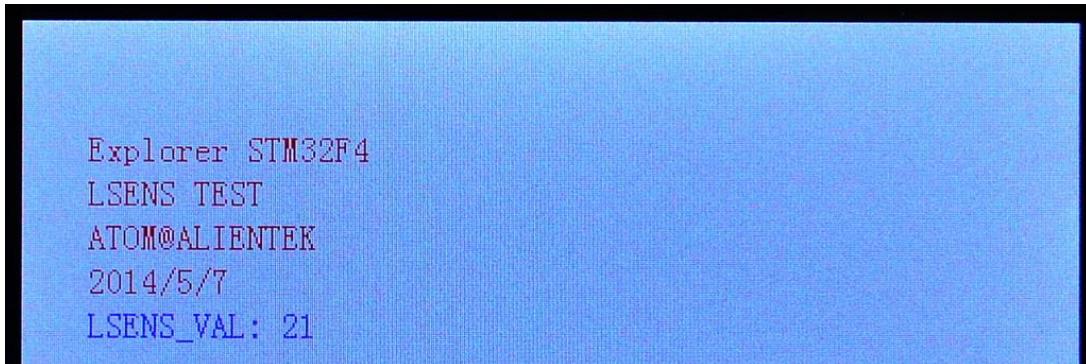


图 25.4.1 光敏传感器实验测试图

伴随 DS0 的不停闪烁，提示程序在运行。此时，我们可以通过给 LS1 不同的光照强度，来观察 LSENS_VAL 值的变化，光照越强，该值越大，光照越弱，该值越小。

第二十六章 DAC 实验

上几章，我们介绍了 STM32F4 的 ADC 使用，本章我们将向大家介绍 STM32F4 的 DAC 功能。在本章中，我们将利用按键（或 USMART）控制 STM32F4 内部 DAC1 来输出电压，通过 ADC1 的通道 5 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：

26.1 STM32F4 DAC 简介

26.2 硬件设计

26.3 软件设计

26.4 下载验证

26.1 STM32F4 DAC 简介

STM32F4 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+（通 ADC 共用）以获得更精确的转换结果。

STM32F4 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 26.1.1 所示：

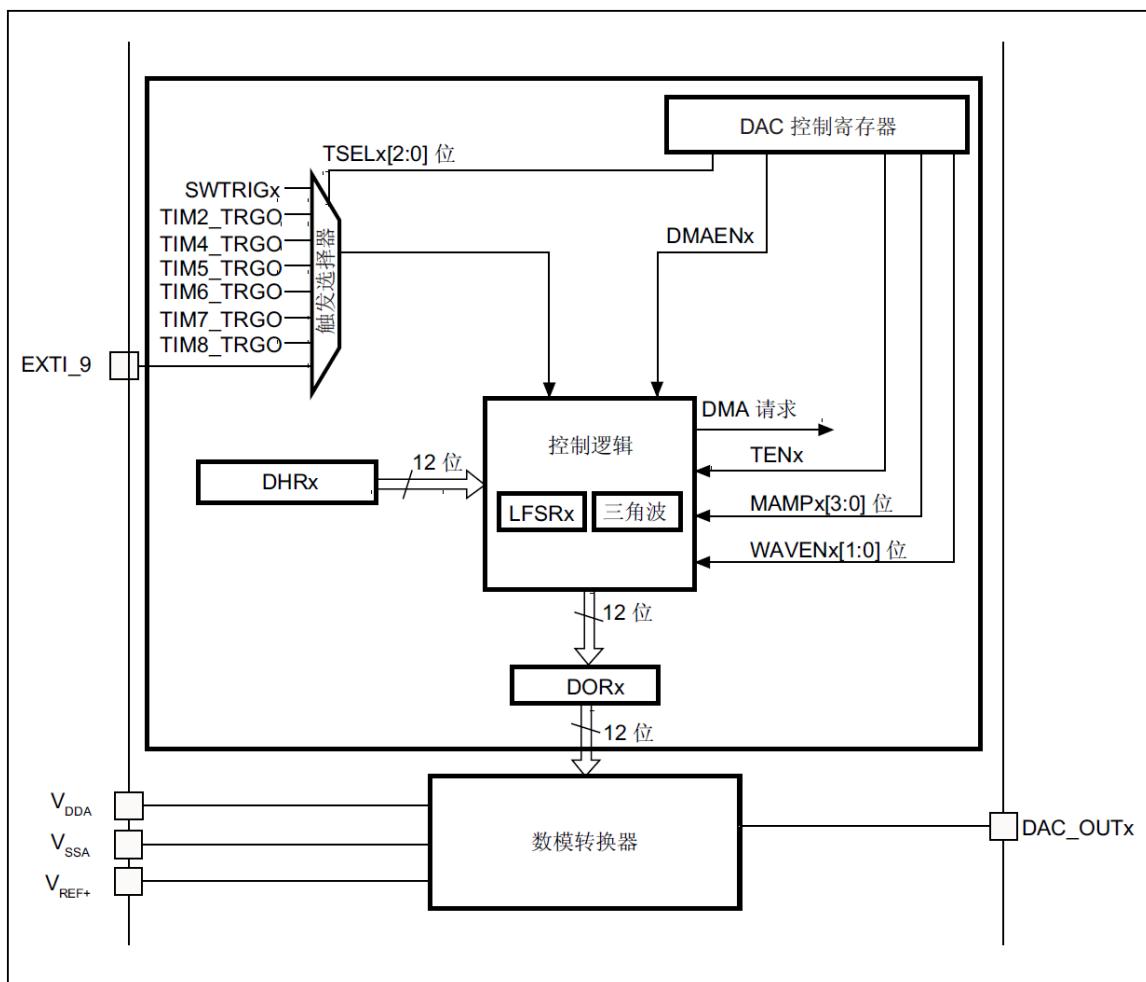


图 26.1.1 DAC 通道模块框图

图中 V_{DDA} 和 V_{SSA} 为 DAC 模块模拟部分的供电，而 V_{ref+} 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 26.1.1 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，STM32F4 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 STM32F407ZGT6 的数据手册查到 $t_{SETTLING}$ 的典型值为 3us，最大是 6us。所以 DAC 的转换速度最快是 333K 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 26.1.2 所示：

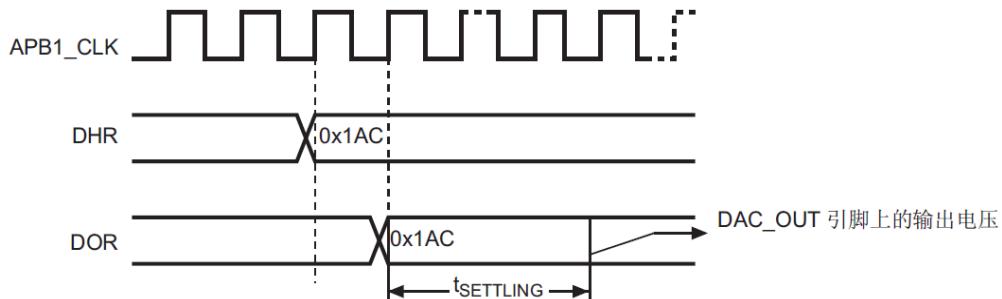


图 26.1.2 TEN=0 时 DAC 模块转换时间框图

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 26.1.3 所示：

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	DMAU DRIE2	DMA EN2	MAMP2[3:0]				WAVE2[1:0]		TSEL2[2:0]			TEN2	BOFF2	EN2		
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
Reserved	DMAU DRIE1	DMA EN1	MAMP1[3:0]				WAVE1[1:0]		TSEL1[2:0]			TEN1	BOFF1	EN1		
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 26.1.3 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 26.1.4 所示：

位 7:6 WAVE1[1:0]: DAC 1 通道噪声/三角波生成使能 (DAC channel1 noise/triangle wave generation enable)

这些位将由软件置 1 和清零。

- 00: 禁止生成波
- 01: 使能生成噪声波
- 1x: 使能生成三角波

注意: 只在位 **TEN1 = 1** (使能 DAC 1 通道触发) 时使用。

位 5:3 TSEL1[2:0]: DAC 1 通道触发器选择 (DAC channel1 trigger selection)

这些位用于选择 DAC 1 通道的外部触发事件。

- | | |
|--------------------|--------------------|
| 000: 定时器 6 TRGO 事件 | 100: 定时器 2 TRGO 事件 |
| 001: 定时器 8 TRGO 事件 | 101: 定时器 4 TRGO 事件 |
| 010: 定时器 7 TRGO 事件 | 110: 外部中断线 9 |
| 011: 定时器 5 TRGO 事件 | 111: 软件触发 |

注意: 只在位 **TEN1 = 1** (使能 DAC 1 通道触发) 时使用。

位 2 TEN1: DAC 1 通道触发使能 (DAC channel1 trigger enable)

此位由软件置 1 和清零, 以使能/禁止 DAC 1 通道触发。

- 0: 禁止 DAC 1 通道触发, 写入 **DAC_DHRx** 寄存器的数据在一个 APB1 时钟周期之后转移到 **DAC_DOR1** 寄存器
- 1: 使能 DAC 1 通道触发, **DAC_DHRx** 寄存器的数据在三个 APB1 时钟周期之后转移到 **DAC_DOR1** 寄存器

注意: 如果选择软件触发, **DAC_DHRx** 寄存器的内容只需一个 APB1 时钟周期即可转移到 **DAC_DOR1** 寄存器。

位 1 BOFF1: DAC 1 通道输出缓冲器禁止 (DAC channel1 output buffer disable)

此位由软件置 1 和清零, 以使能/禁止 DAC 1 通道输出缓冲器。

- 0: 使能 DAC 1 通道输出缓冲器
- 1: 禁止 DAC 1 通道输出缓冲器

位 0 EN1: DAC 1 通道使能 (DAC channel1 enable)

此位由软件置 1 和清零, 以使能/禁止 DAC 1 通道。

- 0: 禁止 DAC 1 通道
- 1: 使能 DAC 1 通道

图 26.1.4 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 STM32F4 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/复制选择器 (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 26.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DACC1DHR[11:0]															
Reserved	rw														

位 31:12 保留，必须保持复位值。

位 11:0 **DACC1DHR[11:0]**: DAC 1 通道 12 位右对齐数据 (DAC channel1 12-bit right-aligned data)
这些位由软件写入，用于为 DAC 1 通道指定 12 位数据。

图 26.1.5 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 STM32F4 实现 DAC 输出的相关设置，本章我们将使用 DAC 模块的通道 1 来输出模拟电压。这里我们用到的库函数以及相关定义分布在文件 stm32f4xx_dac.c 以及头文件 stm32f4xx_dac.h 中。实现上面功能的详细设置步骤如下：

1) 开启 PA 口时钟，设置 PA4 为模拟输入。

STM32F407ZGT6 的 DAC 通道 1 是接在 PA4 上的，所以，我们先要使能 GPIOA 的时钟，然后设置 PA4 为模拟输入。

这里需要特别说明一下，虽然 DAC 引脚设置为输入，但是 STM32F4 内部会连接在 DAC 模拟输出上，这在我们引脚复用映射章节有讲解。

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //模拟输入
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化
```

对于 DAC 通道与引脚对应关系，这在 STM32F4 的数据手册引脚表上有列出，如下图：

PA4	I/O	TTa	(4)	SPI1_NSS / SPI3_NSS / USART2_CK / DCMI_HSYNC / OTG_HS_SOF / I2S3_WS / EVENTOUT	ADC12_IN4 DAC_OUT1
PA5	I/O	TTa	(4)	SPI1_SCK / OTG_HS_ULPI_CK / TIM2_CH1_ETR / TIM8_CH1N / EVENTOUT	ADC12_IN5 / DAC_OUT2

图 26.1.6 DAC 通道引脚对应关系

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。STM32F4 的 DAC 模块时钟是由

APB1 提供的，所以我们先要在通过调用函数 RCC_APB1PeriphClockCmd 来使能 DAC1 时钟。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE); //使能 DAC 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 DAC 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 DAC 通道 1
```

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。设置 DHR12R1 的库函数是：

```
DAC_SetChannel1Data(DAC_Align_12b_R, 0); //12 位右对齐数据格式设置 DAC 值
```

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 对应通道最后一次转换的数值，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 STM32F4 的 DAC 通道 1 来输出不同的模拟电压了。

26.2 硬件设计

本章用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY_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 26.2.1 所示：

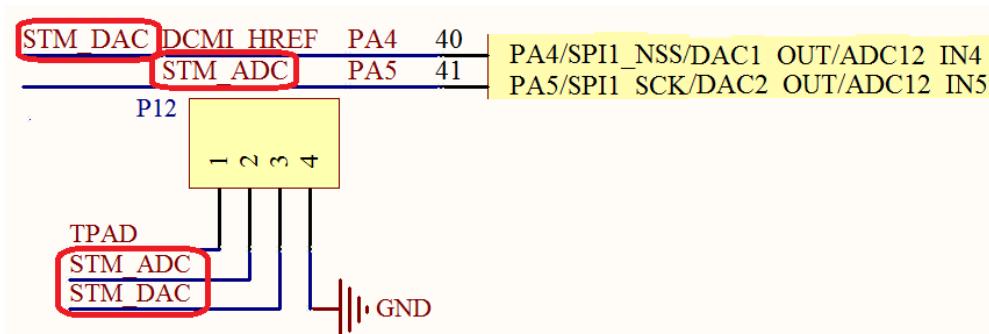


图 26.2.1 ADC、DAC 与 STM32F4 连接原理图

P12 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 26.2.2 所示：

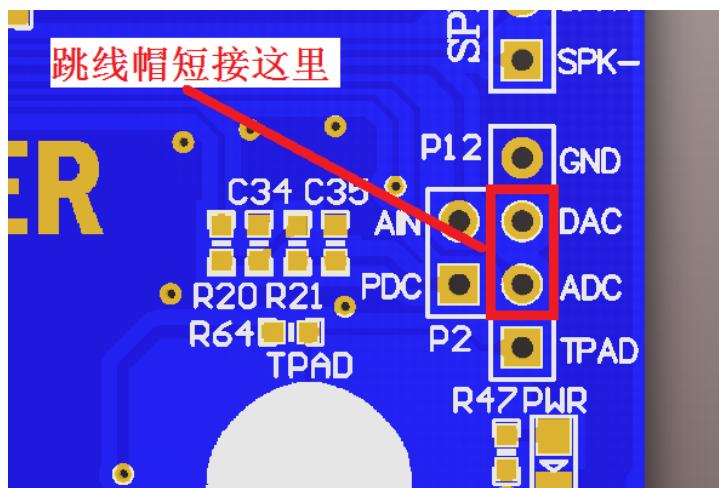


图 26.2.2 硬件连接示意图

26.3 软件设计

打开本章实验工程可以发现，我们相比 ADC 实验，在库函数中主要是添加了 dac 支持的相关文件 `stm32f4xx_dac.c` 以及包含头文件 `stm32f4xx_dac.h`。同时我们在 HARDWARE 分组下面新建了 `dac.c` 源文件以及包含对应的头文件 `dac.h`。这两个文件用来存放我们编写的 ADC 相关函数和定义。打开 `dac.c`，代码如下：

```

//DAC 通道 1 输出初始化
void Dac1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    DAC_InitTypeDef DAC_InitType;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_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;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN; //模拟输入
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
    GPIO_Init(GPIOA, &GPIO_InitStructure); //①初始化 GPIO

    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;
                                         //屏蔽、幅值设置
    DAC_InitType.DAC_OutputBuffer = DAC_OutputBuffer_Disable; //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 DAC 通道 1
}

```

```
DAC_SetChannel1Data(DAC_Align_12b_R, 0); //⑤12 位右对齐数据格式
}
//设置通道 1 输出电压
//vol:0~3300,代表 0~3.3V
void Dac1_Set_Vol(u16 vol)
{
    double temp=vol;
    temp/=1000;
    temp=temp*4096/3.3;
    DAC_SetChannel1Data(DAC_Align_12b_R,temp);//12 位右对齐数据格式}
```

此部分代码就 2 个函数，Dac1_Init 函数用于初始化 DAC 通道 1。这里基本上是按我们上面的步骤来初始化的，我们用序号①~⑤已经标示这些步骤。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，实际就是将电压值转换为 DAC 输入值。

其他头文件代码就比较简单，这里我们不做过多讲解，接下来我们来看看主函数代码：

```
int main(void)
{
    u16 adcx;
    float temp;
    u8 t=0, key;
    u16 dacval=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    Adc_Init(); //adc 初始化
    KEY_Init(); //按键初始化
    Dac1_Init(); //DAC 通道 1 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"DAC TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/6");
    LCD_ShowString(30,130,200,16,16,"WK_UP:+ KEY1:-");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,150,200,16,16,"DAC VAL:");
    LCD_ShowString(30,170,200,16,16,"DAC VOL:0.000V");
    LCD_ShowString(30,190,200,16,16,"ADC VOL:0.000V");

    DAC_SetChannel1Data(DAC_Align_12b_R,dacval);//初始值为 0
    while(1)
```

```

{
    t++;
    key=KEY_Scan(0);
    if(key==WKUP_PRES)
    {
        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==KEY1_PRES||key==WKUP_PRES)
        //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(94,150,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(94,170,temp,1,16,0);          //显示电压值整数部分
        temp-=adcx;
        temp*=1000;
        LCD_ShowxNum(110,170,temp,3,16,0X80);      //显示电压值的小数部分
        adcx=Get_Adc_Average(ADC_Channel_5,10);    //得到 ADC 转换值
        temp=(float)adcx*(3.3/4096);                //得到 ADC 电压值
        adcx=temp;
        LCD_ShowxNum(94,190,temp,1,16,0);          //显示电压值整数部分
        temp-=adcx;
        temp*=1000;
        LCD_ShowxNum(110,190,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 KEY_UP (WKUP 按键) 和 KEY1 (也就是上下键) 来实现对 DAC 输出的幅值控制。按下 KEY_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。

26.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 26.4.1 所示：

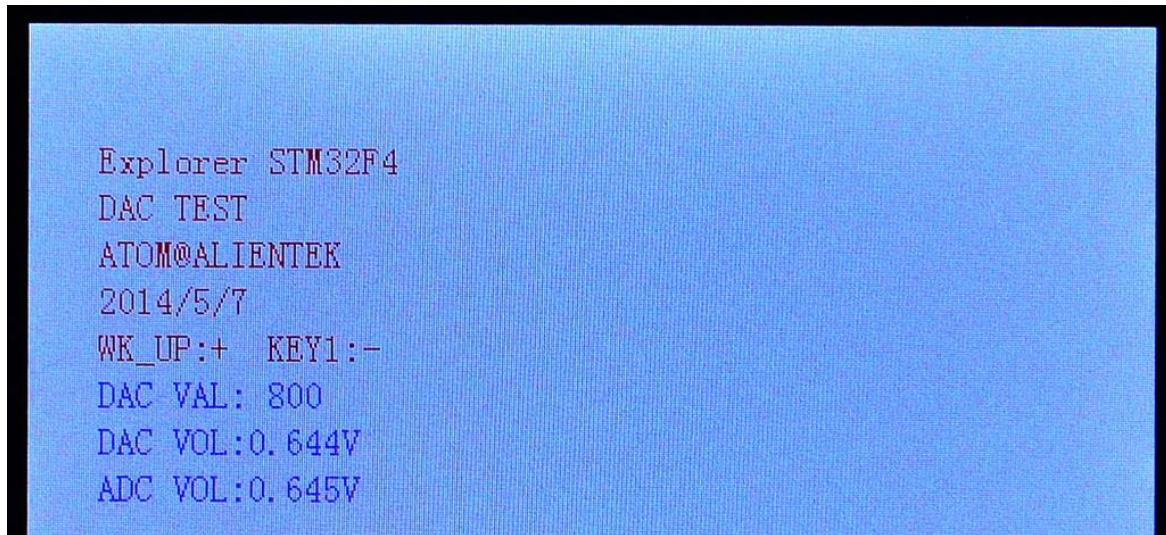


图 26.4.1 DAC 实验测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。此时，我们通过按 KEY_UP 按键，可以看到输出电压增大，按 KEY1 则变小。

第二十七章 PWM DAC 实验

上一章，我们介绍了 STM32F4 自带 DAC 模块的使用，但有时候，可能两个 DAC 不够用，此时，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使用 STM32F4 的 PWM 来设计一个 DAC。我们将使用按键（或 USMART）控制 STM32F4 的 PWM 输出，从而控制 PWM DAC 的输出电压，通过 ADC1 的通道 5 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：

27.1 PWM DAC 简介

27.2 硬件设计

27.3 软件设计

27.4 下载验证

27.1 PWM DAC 简介

有时候，STM32F4 自带的 2 路 DAC 可能不够用，需要多路 DAC，外扩 DAC 成本又会高不少。此时，我们可以利用 STM32F4 的 PWM+简单的 RC 滤波来实现 DAC 输出，从而节省成本。在精度要求不是很高的时候，PWM+RC 滤波的 DAC 输出方式，是一种非常廉价的解决方案。

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 27.1.1 所示：

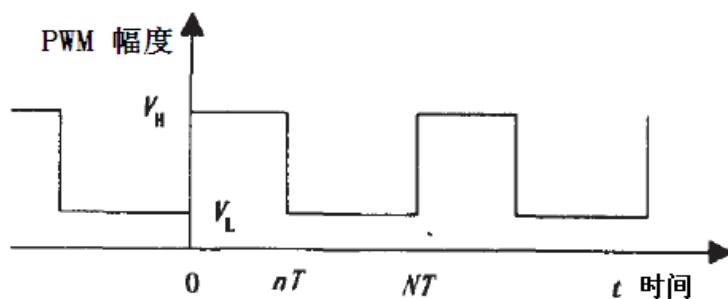


图 27.1.1 实际电路典型 PWM 波形

图 27.1.1 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 STM32F4 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 STM32F4 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 STM32F4 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：

$$\begin{aligned}
 f(t) = & \left[\frac{n}{N} (V_h - V_l) + V_l \right] + \\
 & 2 \frac{V_h - V_l}{\pi} \sin\left(\frac{n}{N}\pi\right) \cos\left(\frac{2\pi}{NT}t - \frac{n\pi}{N}k\right) + \quad ② \\
 & \sum_{k=2}^{\infty} 2 \frac{V_h - V_l}{k\pi} \left| \sin\left(\frac{n\pi}{N}k\right) \right| \left| \cos\left(\frac{2\pi}{NT}kt - \frac{n\pi}{N}k\right) \right|
 \end{aligned}$$

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STM32F4的定时器大部分都是16位的(TIM2和TIM5是32位)，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 $-20\lg(2.1/0.01289)=-44dB$ 的衰减。

STM32F4的定时器最快的计数频率是168Mhz，某些定时器只能到84M，所以我们以84M频率为例介绍，8为分辨率的时候，PWM频率为 $84M/256=328.125Khz$ 。如果是1阶RC滤波，则要求截止频率2.07Khz，如果为2阶RC滤波，则要求截止频率为26.14Khz。

探索者STM32F4开发板的PWM DAC输出采用二阶RC滤波，该部分原理图如图27.1.2所示：

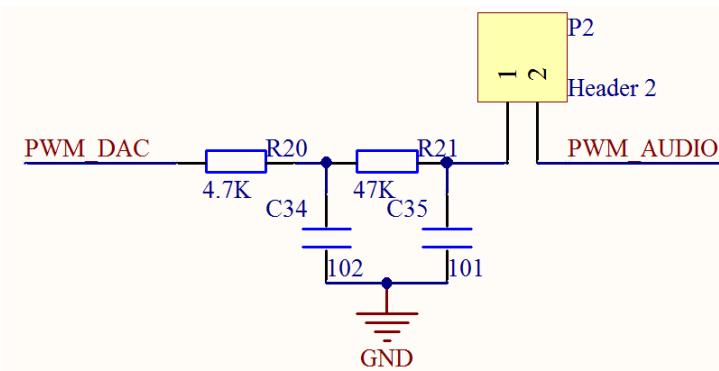


图 27.1.2 PWM DAC 二阶 RC 滤波原理图

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 $R28*C37=R29*C38=RC$ 。根据这个公式，我们计算出图27.1.2的截止频率为：33.8Khz超过了26.14Khz，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 左右。

PWM DAC 的原理部分，就为大家介绍到这里。

27.2 硬件设计

本章用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY_UP 和 KEY1 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) ADC
- 6) PWM DAC

本章，我们使用 STM32F4 的 TIM9_CH2(PA3)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 5 (PA5) 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 27.2.1 所示：

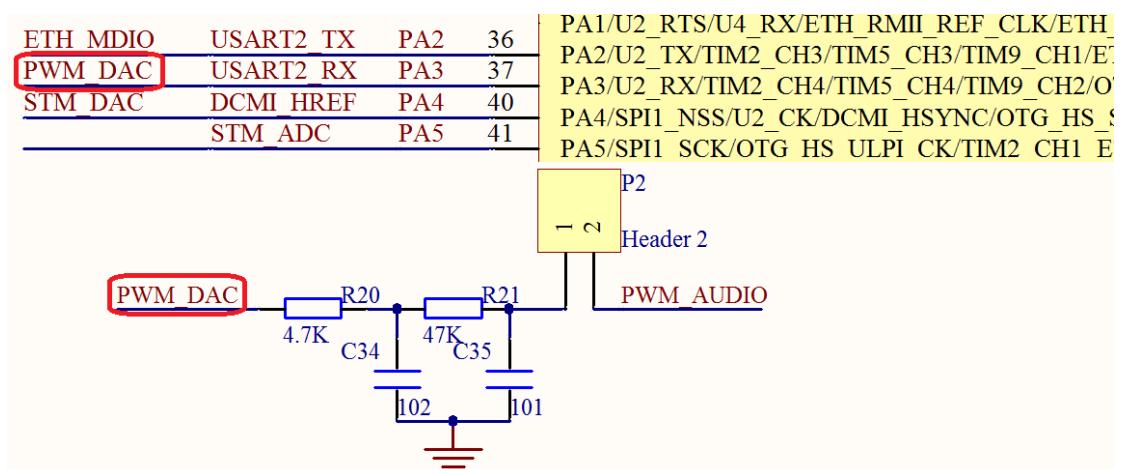


图 27.2.1 PWM DAC 原理图

从上图可知 PWM_DAC 的连接关系，但是这里有个特别需要注意的地方：因为 PWM_DAC 和 USART2_RX 共用了 PA3 引脚，所以在做本例程的时候，必须拔了 P9 上面 PA3(RX)的跳线帽（左侧跳线帽），否则会影响 PWM 转换结果!!!

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 27.2.2 所示：

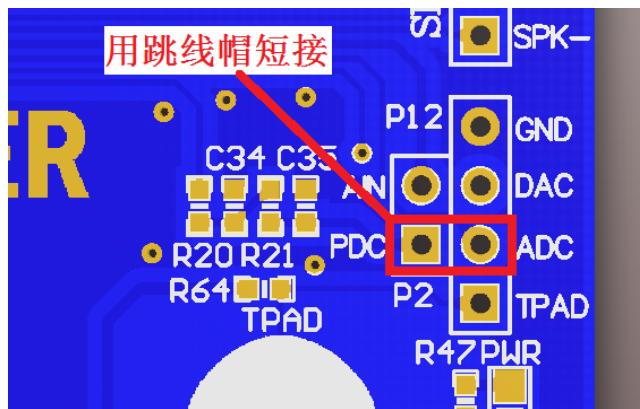


图 27.2.2 硬件连接示意图

27.3 软件设计

打开本章的实验工程可以看到，我们本章并没有增加其他新的库函数文件支持。主要是使用了 adc 和定时器相关的库函数支持。因为我们是使用定时器产生 PWM 信号作为 PWM DAC 的输入信号经过二阶 RC 滤波从而产生一定幅度模拟信号，所以我们需要添加定时器相关的库函数支持。在 HARDWARE 分组下，我们新建了 pwmdac.c 源文件和对应的头文件用来初始化定时器 9 的 PWM。接下来我们看看 pwmdac.c 源文件内容：

```

void TIM9_CH2_PWM_Init(u16 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCIInitTypeDef  TIM_OCInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM9,ENABLE); //TIM9 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PA 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3; //GPIOA3
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA3

    GPIO_PinAFConfig(GPIOA,GPIO_PinSource3,GPIO_AF_TIM9);
                                //PA3 复用位定时器 9 AF3
    TIM_TimeBaseStructure.TIM_Prescaler=psc; //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseStructure.TIM_Period=arr; //自动重装载值
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
    TIM_TimeBaseInit(TIM9,&TIM_TimeBaseStructure); //初始化定时器 9
}

```

```

//初始化 TIM14 Channel1 PWM 模式
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
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_OCInitStructure.TIM_Pulse=0;
TIM_OC2Init(TIM9, &TIM_OCInitStructure); //初始化外设 TIM9 OC2

TIM_OC2PreloadConfig(TIM9, TIM_OCPreload_Enable); //使能预装载寄存器
TIM_ARRPreloadConfig(TIM9,ENABLE); //ARPE 使能
TIM_Cmd(TIM9, ENABLE); //使能 TIM9
}

```

该函数用来初始化 TIM9_CH2 的 PWM 输出 (PA3)，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。

pwmdac.h 头文件内容主要是函数申明，这里不做过多讲解。

接下来我们看看主函数内容：

```

int main(void)
{
    u16 adcx, pwmval=0;
    float temp;
    u8 t=0,key;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    Adc_Init(); //adc 初始化
    KEY_Init(); //按键初始化
    TIM9_CH2_PWM_Init(255,0); //TIM4 PWM 初始化, Fpwm=168M/256=656.25Khz.
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"PWM DAC TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/6");
    LCD_ShowString(30,130,200,16,16,"WK_UP:+ KEY1:-");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,150,200,16,16,"DAC VAL:");
    LCD_ShowString(30,170,200,16,16,"DAC VOL:0.000V");
    LCD_ShowString(30,190,200,16,16,"ADC VOL:0.000V");
    TIM_SetCompare2(TIM9,pwmval); //初始值
    while(1)
    {
        t++;

```

```
key=KEY_Scan(0);
if(key==4)
{
    if(pwmval<250)pwmval+=10;
    TIM_SetCompare2(TIM9,pwmval); //输出
}else if(key==2)
{
    if(pwmval>10)pwmval-=10;
    else pwmval=0;
    TIM_SetCompare2(TIM9,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_GetCapture2(TIM9);;
    LCD_ShowxNum(94,150,adcx,3,16,0); //显示 DAC 寄存器值
    temp=(float)adcx*(3.3/256); //得到 DAC 电压值
    adcx=temp;
    LCD_ShowxNum(94,170,temp,1,16,0); //显示电压值整数部分
    temp-=adcx; temp*=1000;
    LCD_ShowxNum(110,170,temp,3,16,0x80); //显示电压值的小数部分
    adcx=Get_Adc_Average(ADC_Channel_5,20); //得到 ADC 转换值
    temp=(float)adcx*(3.3/4096); //得到 ADC 电压值
    adcx=temp;
    LCD_ShowxNum(94,190,temp,1,16,0); //显示电压值整数部分
    temp-=adcx; temp*=1000;
    LCD_ShowxNum(110,190,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 KEY_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 KEY_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。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 27.4.1 所示：

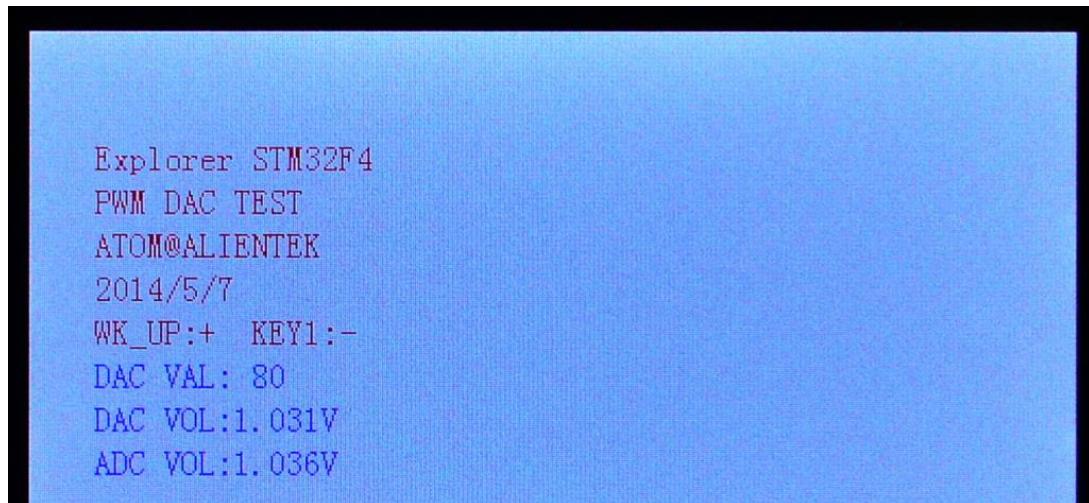


图 27.4.1 PWM DAC 实验测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。此时，我们通过按 KEY_UP 按键，可以看到输出电压增大，按 KEY1 则变小。**特别提醒：**此时 PA3 不能接其他任何外设，如果没有拔了 P9 排针上面 PA3 的跳线帽，那么 PWM DAC 将有很大误差！

第二十八章 DMA 实验

本章我们将向大家介绍 STM32F4 的 DMA。在本章中，我们将利用 STM32F4 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：

- 28.1 STM32F4 DMA 简介
- 28.2 硬件设计
- 28.3 软件设计
- 28.4 下载验证

28.1 STM32F4 DMA 简介

DMA，全称为：Direct Memory Access，即直接存储器访问。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。

STM32F4 最多有 2 个 DMA 控制器（DMA1 和 DMA2），共 16 个数据流（每个控制器 8 个），每一个 DMA 控制器都用于管理一个或多个外设的存储器访问请求。每个数据流总共可以有多达 8 个通道（或称请求）。每个数据流通道都有一个仲裁器，用于处理 DMA 请求间的优先级。

STM32F4 的 DMA 有以下一些特性：

- 双 AHB 主总线架构，一个用于存储器访问，另一个用于外设访问
- 仅支持 32 位访问的 AHB 从编程接口
- 每个 DMA 控制器有 8 个数据流，每个数据流有多达 8 个通道（或称请求）
- 每个数据流有单独的四级 32 位先进先出存储器缓冲区(FIFO)，可用于 FIFO 模式或直接模式。
 - 通过硬件可以将每个数据流配置为：
 - 1, 支持外设到存储器、存储器到外设和存储器到存储器传输的常规通道
 - 2, 支持在存储器方双缓冲的双缓冲区通道
 - 8 个数据流中的每一个都连接到专用硬件 DMA 通道（请求）
 - DMA 数据流请求之间的优先级可用软件编程（4 个级别：非常高、高、中、低），在软件优先级相同的情况下可以通过硬件决定优先级（例如，请求 0 的优先级高于请求 1）
 - 每个数据流也支持通过软件触发存储器到存储器的传输（仅限 DMA2 控制器）
 - 可供每个数据流选择的通道请求多达 8 个。此选择可由软件配置，允许几个外设启动 DMA 请求
- 要传输的数据项的数目可以由 DMA 控制器或外设管理：
 - 1, DMA 流控制器：要传输的数据项的数目是 1 到 65535，可用软件编程
 - 2, 外设流控制器：要传输的数据项的数目未知并由源或目标外设控制，这些外设通过硬件发出传输结束的信号
- 独立的源和目标传输宽度（字节、半字、字）：源和目标的数据宽度不相等时，DMA 自动封装/解封必要的传输数据来优化带宽。这个特性仅在 FIFO 模式下可用。
- 对源和目标的增量或非增量寻址
- 支持 4 个、8 个和 16 个节拍的增量突发传输。突发增量的大小可由软件配置，通常等于外设 FIFO 大小的一半
- 每个数据流都支持循环缓冲区管理
- 5 个事件标志（DMA 半传输、DMA 传输完成、DMA 传输错误、DMA FIFO 错误、

直接模式错误），进行逻辑或运算，从而产生每个数据流的单个中断请求

STM32F4 有两个 DMA 控制器，DMA1 和 DMA2，本章，我们仅针对 DMA2 进行介绍。STM32F4 的 DMA 控制器框图如图 28.1.1 所示：

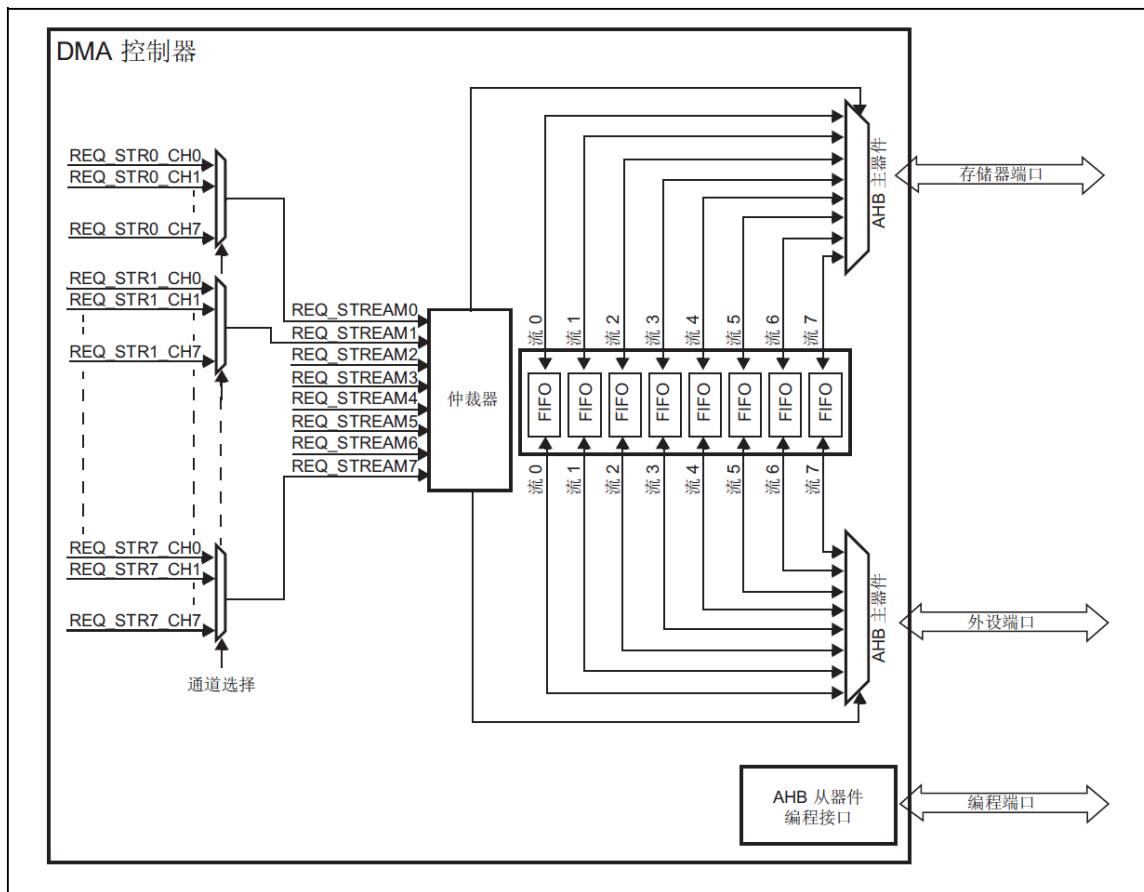


图 28.1.1 DMA 控制器框图

DMA 控制器执行直接存储器传输：因为采用 AHB 主总线，它可以控制 AHB 总线矩阵来启动 AHB 事务。它可以执行下列事务：

- 1, 外设到存储器的传输
- 3, 存储器到外设的传输
- 3, 存储器到存储器的传输

这里特别注意一下，存储器到存储器需要外设接口可以访问存储器，而仅 DMA2 的外设接口可以访问存储器，所以仅 DMA2 控制器支持存储器到存储器的传输，DMA1 不支持。

图 28.1.1 中数据流的多通道选择，是通过 DMA_SxCR 寄存器控制的，如图 28.1.2 所示：

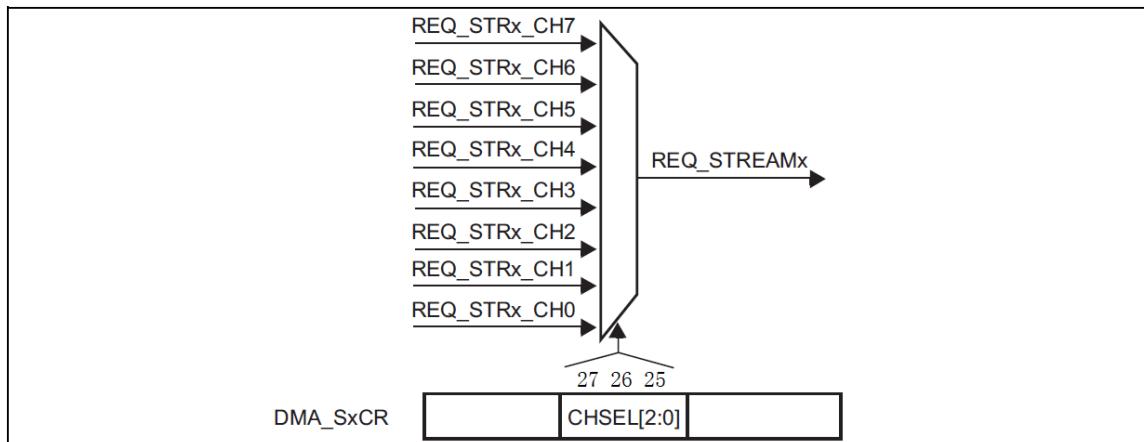


图 28.1.2 DMA 数据流通道选择

从上图可以看出，DMA_SxCR 控制数据流到底使用哪一个通道，每个数据流有 8 个通道可供选择，每次只能选择其中一个通道进行 DMA 传输。接下来，我们看看 DMA2 的各数据流通道映射表，如表 28.1.1 所示：

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道 1		DCMI	ADC2	ADC2		SPI6_TX ⁽¹⁾	SPI6_RX ⁽¹⁾	DCMI
通道 2	ADC3	ADC3		SPI5_RX ⁽¹⁾	SPI5_TX ⁽¹⁾	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道 4	SPI4_RX ⁽¹⁾	SPI4_TX ⁽¹⁾	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道 5		USART6_RX	USART6_RX	SPI4_RX ⁽¹⁾	SPI4_TX ⁽¹⁾		USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
通道 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX ⁽¹⁾	SPI5_TX ⁽¹⁾	TIM8_CH4 TIM8_TRIG TIM8_COM

1. 这些请求在 STM32F42xxx 和 STM32F43xxx 上可用。

表 28.1.1 DMA2 各数据流通道映射表

上表就列出了 DMA2 所有可能的选择情况，来总共 64 种组合，比如本章我们要实现串口 1 的 DMA 发送，即 USART1_TX，就必须选择 DMA2 的数据流 7，通道 4，来进行 DMA 传输。这里注意一下，有的外设（比如 USART1_RX）可能有多个通道可以选择，大家随意选择一个就可以了。

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，该寄存器总共有 2 个：DMA_LISR 和 DMA_HISR，每个寄存器管理 4 数据流（总共 8 个），DMA_LISR 寄存器用于管理数据流 0~3，而 DMA_HISR 用于管理数据流 4~7。这两个寄存器各位描述都完全一模一样，只是管理的数据流不一样。

这里，我们仅以 DMA_LISR 寄存器为例进行介绍，DMA_LISR 各位描述如图 28.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
					TCIF3	HTIF3	TEIF3	DMEIF3		FEIF3	TCIF2	HTIF2	TEIF2	DMEIF2		
r	r	r	r	r	r	r	r		r	r	r	r	r		FEIF2	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
				Reserved	TCIF1	HTIF1	TEIF1	DMEIF1		FEIF1	TCIFO	HTIFO	TEIFO	DMEIFO		FEIFO
r	r	r	r	r	r	r	r		r	r	r	r	r		r	

位 31:28、15:12 保留，必须保持复位值。

位 27、21、11、5 **TCIFx**: 数据流 x 传输完成中断标志 (Stream x transfer complete interrupt flag) ($x = 3..0$)

此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。

0: 数据流 x 上无传输完成事件 1: 数据流 x 上发生传输完成事件

位 26、20、10、4 **HTIFx**: 数据流 x 半传输中断标志 (Stream x half transfer interrupt flag) ($x=3..0$)

此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。

0: 数据流 x 上无半传输事件 1: 数据流 x 上发生半传输事件

位 25、19、9、3 **TEIFx**: 数据流 x 传输错误中断标志 (Stream x transfer error interrupt flag) ($x=3..0$)

此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。

0: 数据流 x 上无传输错误 1: 数据流 x 上发生传输错误

位 24、18、8、2 **DMEIFx**: 数据流 x 直接模式错误中断标志 (Stream x direct mode error interrupt flag) ($x=3..0$)

此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。

0: 数据流 x 上无直接模式错误 1: 数据流 x 上发生直接模式错误

位 23、17、7、1 保留，必须保持复位值。

位 22、16、6、0 **FEIFx**: 数据流 x FIFO 错误中断标志 (Stream x FIFO error interrupt flag) ($x=3..0$)

此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。

0: 数据流 x 上无 FIFO 错误事件 1: 数据流 x 上发生 FIFO 错误事件

图 28.1.3 DMA_LISR 寄存器各位描述

如果开启了 DMA_LISR 中这些位对应的中断，则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 位，即数据流 x 的 DMA 传输完成与否标志。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。DMA_HISR 寄存器各位描述与 DMA_LISR 寄存器各位描述完全一样，只是对应数据流 4~7，这里我们就不列出来了。

第二个是 DMA 中断标志清除寄存器，该寄存器同样有 2 个：DMA_LIFCR 和 DMA_HIFCR，同样是每个寄存器控制 4 个数据流，DMA_LIFCR 寄存器用于管理数据流 0~3，而 DMA_HIFCR 用于管理数据流 4~7。这两个寄存器各位描述都完全一模一样，只是管理的数据流不一样。

这里，我们仅以 DMA_LIFCR 寄存器为例进行介绍，DMA_LIFCR 各位描述如图 28.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved	CTCIF3	CHTIF3	CTEIF3	CDMEIF3	Reserved	CFEIF3	CTCIF2	CHTIF2	CTEIF2	CDMEIF2	Reserved	CFEIF2	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	
Reserved	CTCIF1	CHTIF1	CTEIF1	CDMEIF1	Reserved	CFEIF1	CTCIF0	CHTIF0	CTEIF0	CDMEIF0	Reserved	CFEIF0	W	W	W	W
	W	W	W	W		W	W	W	W	W		W				

位 31:28、15:12 保留，必须保持复位值。

位 27、21、11、5 **CTCIFx**: 数据流 x 传输完成中断标志清零 (Stream x clear transfer complete interrupt flag) ($x = 3..0$)

将 1 写入此位时，DMA_LISR 寄存器中相应的 TCIFx 标志将清零

位 26、20、10、4 **CHTIFx**: 数据流 x 半传输中断标志清零 (Stream x clear half transfer interrupt flag) ($x = 3..0$)

将 1 写入此位时，DMA_LISR 寄存器中相应的 HTIFx 标志将清零

位 25、19、9、3 **CTEIFx**: 数据流 x 传输错误中断标志清零 (Stream x clear transfer error interrupt flag) ($x = 3..0$)

将 1 写入此位时，DMA_LISR 寄存器中相应的 TEIFx 标志将清零

位 24、18、8、2 **CDMEIFx**: 数据流 x 直接模式错误中断标志清零 (Stream x clear direct mode error interrupt flag) ($x = 3..0$)

将 1 写入此位时，DMA_LISR 寄存器中相应的 DMEIFx 标志将清零

位 23、17、7、1 保留，必须保持复位值。

位 22、16、6、0 **CFEIFx**: 数据流 x FIFO 错误中断标志清零 (Stream x clear FIFO error interrupt flag) ($x = 3..0$)

将 1 写入此位时，DMA_LISR 寄存器中相应的 CFEIFx 标志将清零

图 28. 1. 4 DMA_LIFCR 寄存器各位描述

DMA_LIFCR 的各位就是用来清除 DMA_LISR 的对应位的，通过写 1 清除。在 DMA_LISR 被置位后，我们必须通过向该位寄存器对应的位写入 1 来清除。DMA_HIFCR 的使用同 DMA_LIFCR 类似，这里就不做介绍了。

第三个是 DMA 数据流 x 配置寄存器 (DMA_SxCR) ($x=0^7$, 下同)。该寄存器的我们在这里就不贴出来了，见《STM32F4xx 中文参考手册》第 223 页 9.5.5 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、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_SxCR 是 DMA 传输的核心控制寄存器。

第四个是 DMA 数据流 x 数据项数寄存器 (DMA_SxNDTR)。这个寄存器控制 DMA 数据流 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。特别注意，这里是数据项数目，而不是指的字节数。比如设置数据位宽为 16 位，那么传输一次（一个项）就是 2 个字节。

第五个是 DMA 数据流 x 的外设地址寄存器 (DMA_SxPAR)。该寄存器用来存储 STM32F4 外设的地址，比如我们使用串口 1，那么该寄存器必须写入 0x40011004（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 数据流 x 的存储器地址寄存器，由于 STM32F4 的 DMA 支持双缓存，所以存储器地址寄存器有两个：DMA_SxMOAR 和 DMA_SxM1AR，其中 DMA_SxM1AR 仅在双缓冲模式下，才有效。本章我们没用到双缓冲模式，所以存储器地址寄存器就是：DMA_SxMOAR，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[8200] 数组来做存储器，那么我们在 DMA_SxMOAR 中写入&SendBuff 就可以了。

DMA 相关寄存器就为大家介绍到这里，关于这些寄存器的详细描述，请参考《STM32F4xx 中文参考手册》第 9.5 节。本章我们要用到串口 1 的发送，属于 DMA2 的数据流 7，通道 4，接下来我们就介绍下使用库函数的配置步骤和方法。首先这里我们需要指出的是，DMA 相关的库函数支持在文件 stm32f4xx_dma.c 以及对应的头文件 stm32f4xx_dac.h 中。具体步骤如下：

1) 使能 DMA2 时钟，并等待数据流可配置。

DMA 的时钟使能是通过 AHB1ENR 寄存器来控制的，这里我们要先使能时钟，才可以配置 DMA 相关寄存器。所以先要使能 DMA2 的时钟。另外，要对配置寄存器 (DMA_SxCR) 进行设置，必须先等待其最低位为 0 (也就是 DMA 传输禁止了)，才可以进行配置。

库函数使能 DMA2 时钟的方法为：

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE); //DMA2 时钟使能
```

等待 DMA 可配置，也就是等待 DMA_SxCR 寄存器最低位为 0 的方法为：

```
while (DMA_GetCmdStatus(DMA_Streamx) != DISABLE) {} //等待 DMA 可配置
```

2) 初始化 DMA2 数据流 7，包括配置通道，外设地址，存储器地址，传输数据量等。

DMA 的某个数据流各种配置参数初始化是通过 DMA_Init 函数实现的：

```
void DMA_Init(DMA_Stream_TypeDef* DMAy_Streamx, DMA_InitTypeDef* DMA_InitStruct);
```

函数的第一个参数是指定初始化的 DMA 的数据流编号，这个很容易理解。入口参数范围为：DMAx_Stream0~DMAx_Stream7 (x=1, 2)。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_Channel;
    uint32_t DMA_PeripheralBaseAddr;
    uint32_t DMA_Memory0BaseAddr;
    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_FIFOMode;
    uint32_t DMA_FIFOThreshold;
    uint32_t DMA_MemoryBurst;
    uint32_t DMA_PeripheralBurst;
} 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_Channel 用来设置 DMA 数据流对应的通道。前面我们已经讲解过，可供每个数据流选择的通道请求多达 8 个，取值范围为：DMA_Channel_0~DMA_Channel_7。

第二个参数 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。

第三个参数 DMA_Memory0BaseAddr 为内存基地址，也就是我们存放 DMA 传输数据的内存地址。

第四个参数 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。

第五个参数 DMA_BufferSize 设置一次传输数据量的大小，这个很容易理解。

第六个参数 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；

第七个参数 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。

第八个参数 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。

第九个参数 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。

第十个参数 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。

第十一个参数 DMA_Priority 是用来设置 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。优先级可以随便设置，因为我们只有一个数据流被开启了。假设有多个数据流开启（最多 8 个），那么就要设置优先级了，DMA 仲裁器将根据这些优先级的设置来决定先执行那个数据流的 DMA。优先级越高的，越早执行，当优先级相同的时候，根据硬件上的编号来决定哪个先执行（编号越小越优先）。

第十二个参数 DMA_FIFOMode 用来设置是否开启 FIFO 模式。这里我们不开启所以选择 DMA_FIFOMode_Disable。

第十三个参数 DMA_FIFOThreshold 用来选择 FIFO 阈值。根据前面讲解可以为 FIFO 容量的 1/4, 1/2, 3/4 以及 1 倍。这里我们实际并没有开启 FIFO 模式，所以可以不关心。

第四个参数 DMA_MemoryBurst 用来配置存储器突发传输配置。可以选择为 4 个节拍的增量突发传输 DMA_MemoryBurst_INC4, 8 个节拍的增量突发传输 DMA_MemoryBurst_INC8, 16 个节拍的增量突发传输 DMA_MemoryBurst_INC16 以及单次传输 DMA_MemoryBurst_Single。

第十五个参数 DMA_PeripheralBurst 用来配置外设突发传输配置。跟前面一个参数 DMA_MemoryBurst 作用类似，只不过一个针对的是存储器，一个是外设。这里我们选择单次传输 DMA_PeripheralBurst_Single。

参数含义我们就给大家讲解到这里，具体详细配置，大家可以参考中文参考手册相关寄存器配置可以更加详细的了解含义。接下来我们给出上面场景的实例代码：

```
/* 配置 DMA Stream */
DMA_InitStructure.DMA_Channel = chx; //通道选择
DMA_InitStructure.DMA_PeripheralBaseAddr = par;//DMA 外设地址
DMA_InitStructure.DMA_Memory0BaseAddr = mar;//DMA 存储器 0 地址
DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral;//存储器到外设模式
DMA_InitStructure.DMA_BufferSize = ndtr;//数据传输量
```

```

DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
                                         //外设非增量模式
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; //存储器增量模式
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
                                         //外设数据长度:8位
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
                                         //存储器数据长度:8位
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal; // 使用普通模式
DMA_InitStructure.DMA_Priority = DMA_Priority_Medium; //中等优先级
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single; //单次传输
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
                                         //外设突发单次传输
DMA_Init(DMA_Streamx, &DMA_InitStructure); //初始化 DMA Stream

```

3) 使能串口 1 的 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); //使能串口 1 的 DMA 发送
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 DMA2 数据流 7，启动传输。

使能 DMA 数据流的函数为：

```
void DMA_Cmd(DMA_Stream_TypeDef* DMAy_Streamx, FunctionalState NewState)
```

使能 DMA2_Stream7，启动传输的方法为：

```
DMA_Cmd(DMA2_Stream7, ENABLE);
```

通过以上 4 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 数据流 7 传输是否完成，方法是：

```
DMA_GetFlagStatus(DMA2_Stream7, DMA_FLAG_TCIF7);
```

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_Stream_TypeDef* DMAy_Streamx);
```

比如我们要获取 DMA 数据流 7 还有多少个数据没有传输，方法是：

```
DMA_GetCurrDataCounter(DMA1_Channel4);
```

同样，我们也可以设置对应的 DMA 数据流传输的数据量大小，函数为：

```
void DMA_SetCurrDataCounter(DMA_Stream_TypeDef* DMAy_Streamx, uint16_t Counter);
```

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。

28.2 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。

28.3 软件设计

打开本章的实验工程可以看到，我们在 FWLIB 分组下面增加了 DMA 支持文件 stm32f4xx_dma.c，同时引入了 stm32f4xx_dma.h 头文件支持。在 HARDWARE 分组下面我们新增了 dma.c 以及对应头文件 dma.h 用来存放 dma 相关的函数和定义。

打开 dma.c 文件，代码如下：

```
//DMAx 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_Streamx:DMA 数据流, DMA1_Stream0~7/DMA2_Stream0~7
//chx:DMA 通道选择, @ref DMA_channel1 DMA_Channel1_0~DMA_Channel1_7
//par:外设地址 mar:存储器地址 ndtr:数据传输量
void MYDMA_Config(DMA_Stream_TypeDef *DMA_Streamx, u32 chx, u32 par, u32 mar, u16 ndtr)
{
    DMA_InitTypeDef DMA_InitStruct;
    if((u32)DMA_Streamx > (u32)DMA2) //得到当前 stream 是属于 DMA2 还是 DMA1
    {
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE); //DMA2 时钟使能
    } else
    {
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE); //DMA1 时钟使能
    }
    DMA_DeInit(DMA_Streamx);
    while (DMA_GetCmdStatus(DMA_Streamx) != DISABLE) {} //等待 DMA 可配置
    /* 配置 DMA Stream */
    DMA_InitStruct.DMA_Channel = chx; //通道选择
    DMA_InitStruct.DMA_PeripheralBaseAddr = par; //DMA 外设地址
    DMA_InitStruct.DMA_Memory0BaseAddr = mar; //DMA 存储器 0 地址
    DMA_InitStruct.DMA_DIR = DMA_DIR_MemoryToPeripheral; //存储器到外设模式
    DMA_InitStruct.DMA_BufferSize = ndtr; //数据传输量
    DMA_InitStruct.DMA_PeripheralInc = DMA_PeripheralInc_Disable; //外设非增量模式
    DMA_InitStruct.DMA_MemoryInc = DMA_MemoryInc_Enable; //存储器增量模式
    DMA_InitStruct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte; //外设数据长度:8 位
    DMA_InitStruct.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte; //存储器数据长度:8 位
```

```

DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;// 使用普通模式
DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;//中等优先级
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;//FIFO 模式禁止
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;//FIFO 阈值
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
                                         //存储器突发单次传输
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
                                         //外设突发单次传输
DMA_Init(DMA_Streamx, &DMA_InitStructure); //初始化 DMA Stream
}
//开启一次 DMA 传输
//DMA_Streamx:DMA 数据流, DMA1_Stream0~7/DMA2_Stream0~7
//ndtr:数据传输量
void MYDMA_Enable(DMA_Stream_TypeDef *DMA_Streamx, u16 ndtr)
{
    DMA_Cmd(DMA_Streamx, DISABLE);           //关闭 DMA 传输
    while (DMA_GetCmdStatus(DMA_Streamx) != DISABLE) {} //确保 DMA 可以被设置
    DMA_SetCurrDataCounter(DMA_Streamx, ndtr); //数据传输量
    DMA_Cmd(DMA_Streamx, ENABLE);           //开启 DMA 传输
}

```

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，该函数是一个通用的 DMA 配置函数，DMA1/DMA2 的所有通道，都可以利用该函数配置，不过有些固定参数可能要适当修改（比如位宽，传输方向等）。该函数在外部只能修改 DMA 及数据流编号、通道号、外设地址、存储器地址(SxMOAR) 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 数据流。对照前面的配置步骤的详细讲解看看这部分代码即可。

dma.h 头文件内容比较简单，主要是函数申明，这里我们不细说。

接下来我们看看那 main 函数如下：

```

/*发送数据长度,最好等于 sizeof(TEXT_TO_SEND)+2 的整数倍.*/
#define SEND_BUF_SIZE 8200
u8 SendBuff[SEND_BUF_SIZE]; //发送数据缓冲区
const u8 TEXT_TO_SEND[]={"ALIENTEK Explorer STM32F4 DMA 串口实验"};

int main(void)
{
    u16 i;
    u8 t=0, j, mask=0;
    float pro=0;//进度
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200

    LED_Init(); //初始化 LED

```

```
LCD_Init();           //LCD 初始化
KEY_Init();           //按键初始化
/*DMA2, STEAM7, CH4, 外设为串口 1, 存储器为 SendBuff, 长度为:SEND_BUF_SIZE.*/
MYDMA_Config(DMA2_Stream7, DMA_Channel1_4, (u32)&USART1->DR, (u32)SendBuff,
              SEND_BUF_SIZE);

POINT_COLOR=RED;
LCD_ShowString(30, 50, 200, 16, 16, "Explorer STM32F4");
LCD_ShowString(30, 70, 200, 16, 16, "DMA TEST");
LCD_ShowString(30, 90, 200, 16, 16, "ATOM@ALIENTEK");
LCD_ShowString(30, 110, 200, 16, 16, "2014/5/6");
LCD_ShowString(30, 130, 200, 16, 16, "KEY0:Start");
POINT_COLOR=BLUE;//设置字体为蓝色
//显示提示信息
j=sizeof(TEXT_TO_SEND);
for(i=0;i<SEND_BUF_SIZE;i++)//填充 ASCII 字符集数据
{
    if(t>=j)//加入换行符
    {
        if(mask)
        {
            SendBuff[i]=0x0a;t=0;
        }else
        {
            SendBuff[i]=0xd;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==KEY0_PRES) //KEY0 按下
    {
        printf("\r\nDMA DATA:\r\n");
        LCD_ShowString(30, 150, 200, 16, 16, "Start Transimit....");
        LCD_ShowString(30, 170, 200, 16, 16, "%");//显示百分号
        USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE); //使能串口 1 的 DMA 发送
        MYDMA_Enable(DMA2_Stream7, SEND_BUF_SIZE); //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(DMA2_Stream7, DMA_FLAG_TCIF7) !=RESET)
        //等待 DMA2_Stream7 传输完成
    {
        DMA_ClearFlag(DMA2_Stream7, DMA_FLAG_TCIF7); //清传输完成标志
        break;
    }
    pro=DMA_GetCurrDataCounter(DMA2_Stream7); //得到当前剩余数据数
    pro=1-pro/SEND_BUF_SIZE; //得到百分比
    pro*=100; //扩大 100 倍
    LCD_ShowNum(30, 170, pro, 3, 16);
}
LCD_ShowNum(30, 170, 100, 3, 16); //显示 100%
LCD_ShowString(30, 150, 200, 16, 16, "Transimit Finished!");
}
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。
```

28.4 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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 28.4.1 所示：

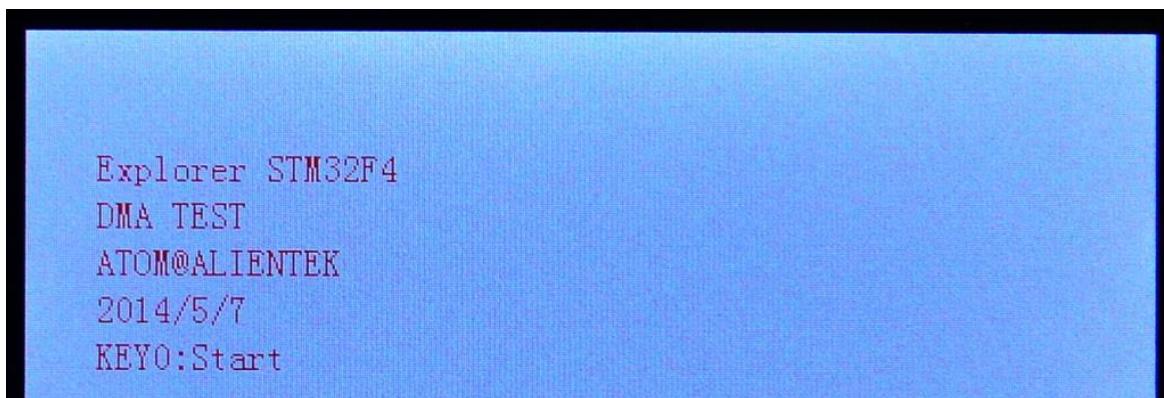


图 28.4.1 DMA 实验测试图

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 28.4.2 所示的内容：

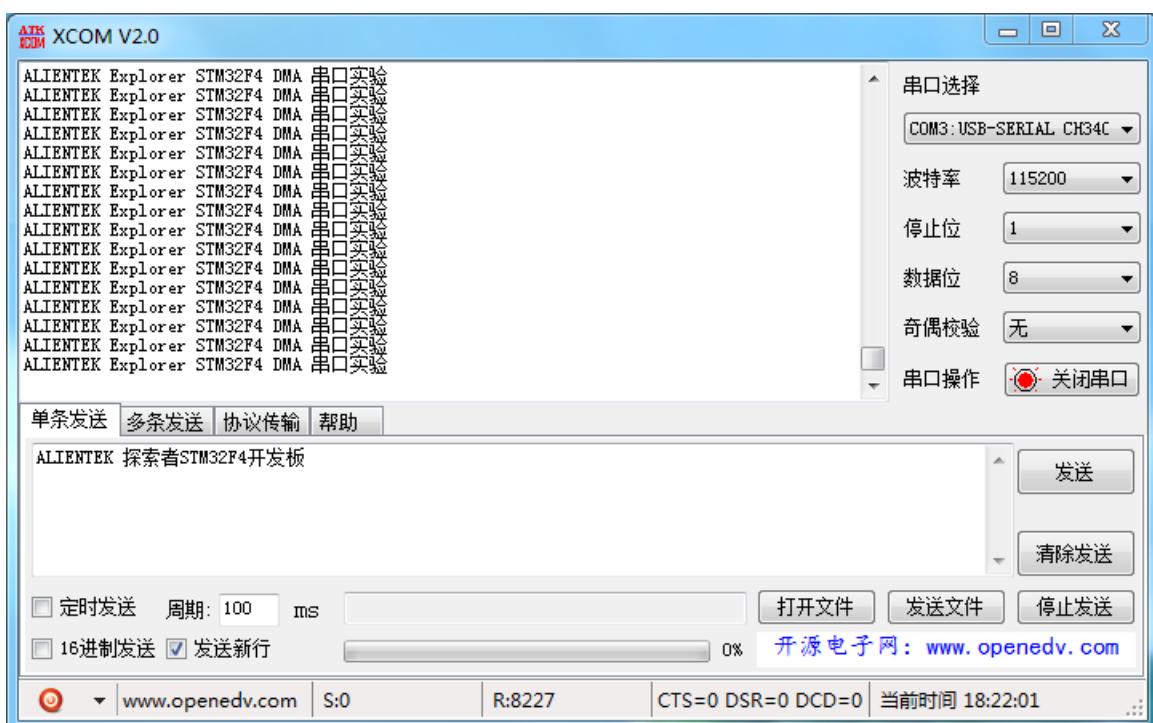
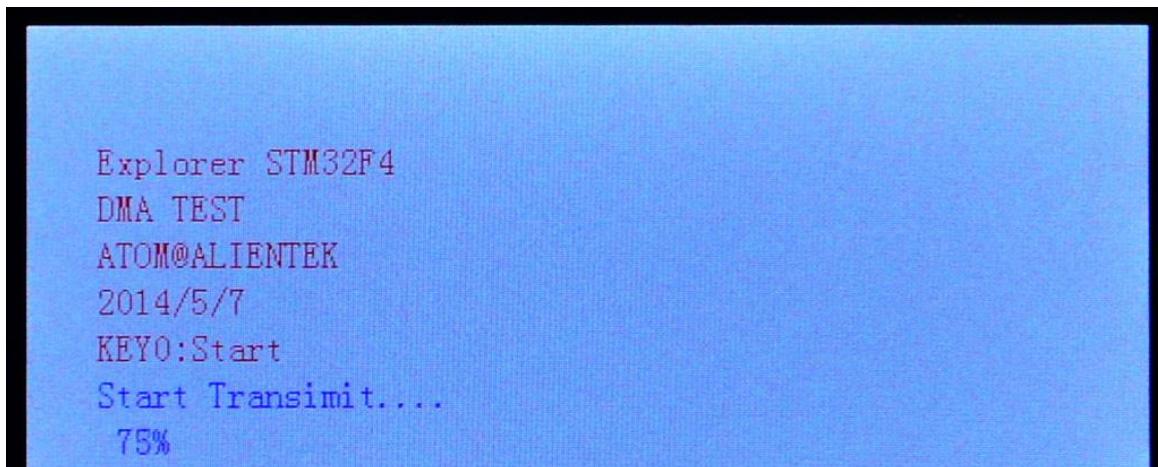


图 28.4.2 串口收到的数据内容

可以看到串口收到了探索者 STM32F4 开发板发送过来的数据，同时可以看到 TFTLCD 上显示了进度等信息，如图 28.4.3 所示：



28.4.3 DMA 串口数据传输中

至此，我们整个 DMA 实验就结束了，希望大家通过本章的学习，掌握 STM32F4 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。

第二十九章 IIC 实验

本章我们将向大家介绍如何使用 STM32F4 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，我们将使用 STM32F4 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：

- 29.1 IIC 简介
- 29.2 硬件设计
- 29.3 软件设计
- 29.4 下载验证

29.1 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 29.1.1 所示：

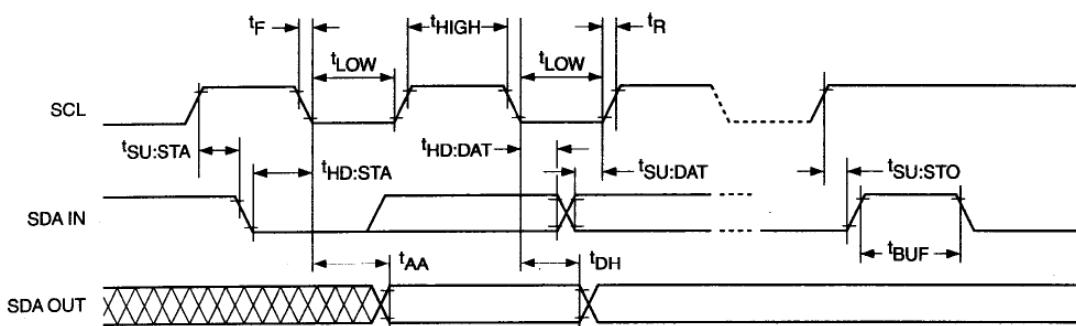


图 29.1.1 IIC 总线时序图

ALIENTEK 探索者 STM32F4 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 STM32F4 来实现 24C02 的读写。

目前大部分 MCU 都带有 IIC 总线接口，STM32F4 也不例外。但是这里我们不使用 STM32F4 的硬件 IIC 来读写 24C02，而是通过软件模拟。ST 为了规避飞利浦 IIC 专利问题，将 STM32 的硬件 IIC 设计的比较复杂，而且稳定性不怎么好，所以这里我们不推荐使用。有兴趣的读者可以研究一下 STM32F4 的硬件 IIC。

用软件模拟 IIC，最大的好处就是方便移植，同一个代码兼容所有 MCU，任何一个单片机

只要有 IO 口，就可以很快的移植过去，而且不需要特定的 IO 口。而硬件 IIC，则换一款 MCU，基本上就得重新搞一次，移植是比较麻烦的。

本章实验功能简介：开机的时候先检测 24C02 是否存在，然后在主循环里面检测两个按键，其中 1 个按键（KEY1）用来执行写入 24C02 的操作，另外一个按键（KEY0）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。

29.2 硬件设计

本章需要用到的硬件资源有：

- 1) 指示灯 DS0
- 2) KEY_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 STM32F4 的连接，24C02 的 SCL 和 SDA 分别连在 STM32F4 的 PB8 和 PB9 上的，连接关系如图 29.2.1 所示：

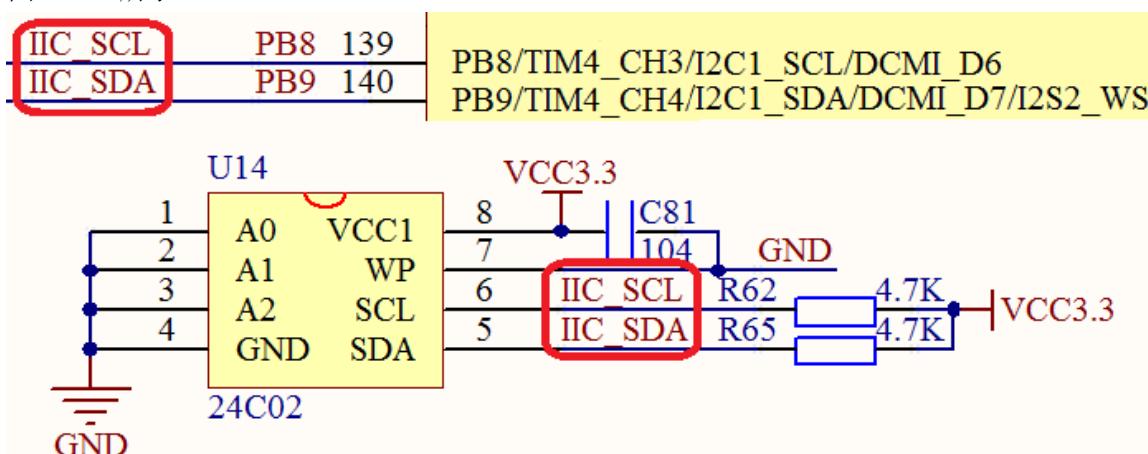


图 29.2.1 STM32F4 与 24C02 连接图

29.3 软件设计

打开本章的实验工程可以看到，我们并没有在 FWLIB 分组之下添加新的固件库文件支持，因为我们是通过 GPIO 来模拟 IIC。我们新增了 myiic.c 文件用来存放 iic 底层驱动。新增了 24cxx.c 文件用来存放 24C02 的底层驱动。

打开 myiic.c 文件，代码如下：

```
//初始化 IIC
void IIC_Init(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE); //使能 GPIOB 时钟

    //GPIOB8,B9 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //普通输出模式
```

```
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化
IIC_SCL=1;
IIC_SDA=1;
}
//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 form high to low
    delay_us(4);
    IIC_SCL=0;//钳住 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。

打开 myiic.h 头文件可以看到，我们除了函数申明之外，还定义了几个宏定义标识符：

```

//IO 方向设置
#define SDA_IN() {GPIOB->MODER&=~(3<<(9*2));GPIOB->MODER|=0<<9*2;}
                                         //PB9 输入模式
#define SDA_OUT() {GPIOB->MODER&=~(3<<(9*2));GPIOB->MODER|=1<<9*2;}
                                         //PB9 输出模式

```

```

//IO 操作函数
#define IIC_SCL    PBout(8) //SCL
#define IIC_SDA    PBout(9) //SDA
#define READ_SDA   PBin(9)  //输入 SDA

```

该部分代码的 SDA_IN() 和 SDA_OUT() 分别用于设置 IIC_SDA 接口为输入和输出，如果这两句代码看不懂，请好好温习下 IO 口的使用。其他几个宏定义就是我们通过位带实现 IO 口操作。

接下来我们看看 24cxx.c 源文件代码代码：

```

//初始化 IIC 接口
void AT24CXX_Init(void)
{
    IIC_Init();//IIC 初始化
}
//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); //EEPROM 写入过程比较慢，需等待一点时间，再写下一次
}
//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，我们看看主函数代码：

```
//要写入到 24c02 的字符串数组
const u8 TEXT_Buffer[]={"Explorer STM32F4 IIC TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key;
    u16 i=0;
    u8 datatemp[SIZE];
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    AT24CXX_Init(); //IIC 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"IIC TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
```

```
LCD_ShowString(30,110,200,16,16,"2014/5/6");
LCD_ShowString(30,130,200,16,16,"KEY1:Write  KEY0:Read"); //显示提示信息

while(AT24CXX_Check())//检测不到 24c02
{
    LCD_ShowString(30,150,200,16,16,"24C02 Check Failed!");
    delay_ms(500);
    LCD_ShowString(30,150,200,16,16,"Please Check!      ");
    delay_ms(500);
    LED0=!LED0;//DS0 闪烁
}
LCD_ShowString(30,150,200,16,16,"24C02 Ready!");
POINT_COLOR=BLUE;//设置字体为蓝色
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY1_PRES)//KEY1 按下,写入 24C02
    {
        LCD_Fill(0,170,239,319,WHITE);//清除半屏
        LCD_ShowString(30,170,200,16,16,"Start Write 24C02....");
        AT24CXX_Write(0,(u8*)TEXT_Buffer,SIZE);
        LCD_ShowString(30,170,200,16,16,"24C02 Write Finished");//提示传送完成
    }
    if(key==KEY0_PRES)//KEY0 按下,读取字符串并显示
    {
        LCD_ShowString(30,170,200,16,16,"Start Read 24C02.... ");
        AT24CXX_Read(0,datatemp,SIZE);
        LCD_ShowString(30,170,200,16,16,"The Data Readed Is:   ");
        LCD_ShowString(30,190,200,16,16,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 KEY1 按键来控制 24C02 的写入，通过另外一个按键 KEY0 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。

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 探索者 STM32F4 开发板上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 29.4.1 所示：

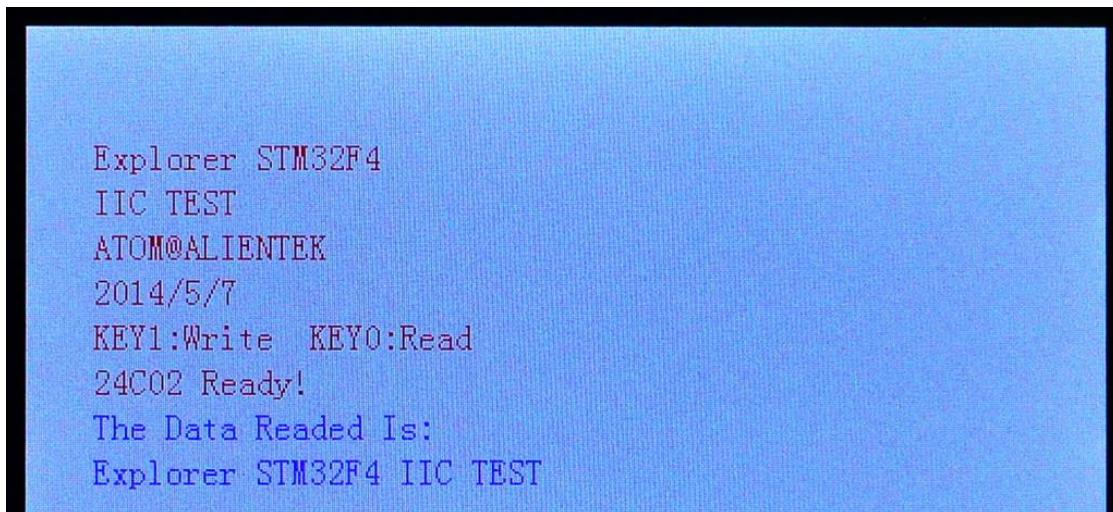


图 29.4.1 IIC 实验程序运行效果图

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 PB8 和 PB9 短接就可以看到报错了。

第三十章 SPI 实验

本章我们将向大家介绍 STM32F4 的 SPI 功能。在本章中，我们将使用 STM32F4 自带的 SPI 来实现对外部 FLASH (W25Q128) 的读写，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

- 30.1 SPI 简介
- 30.2 硬件设计
- 30.3 软件设计
- 30.4 下载验证

30.1 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，STM32F4 也有 SPI 接口。下面我们看看 SPI 的内部简明图（图 30.1.1）：

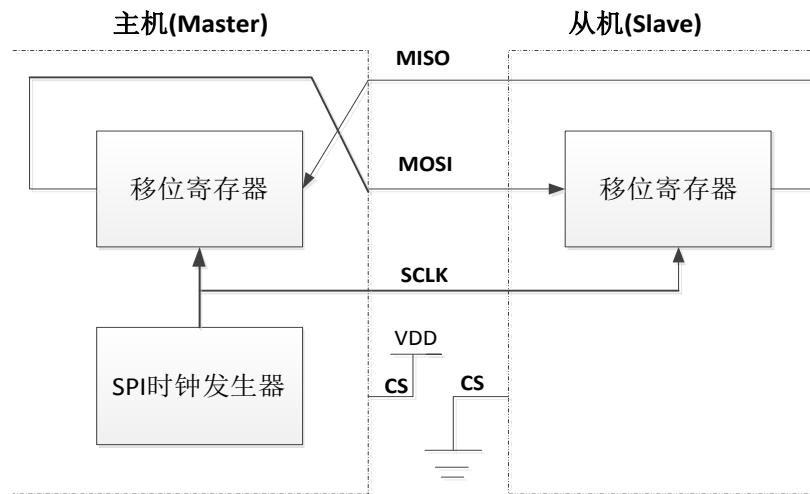


图 30.1.1 SPI 内部结构简明图

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 30.1.2 所示：

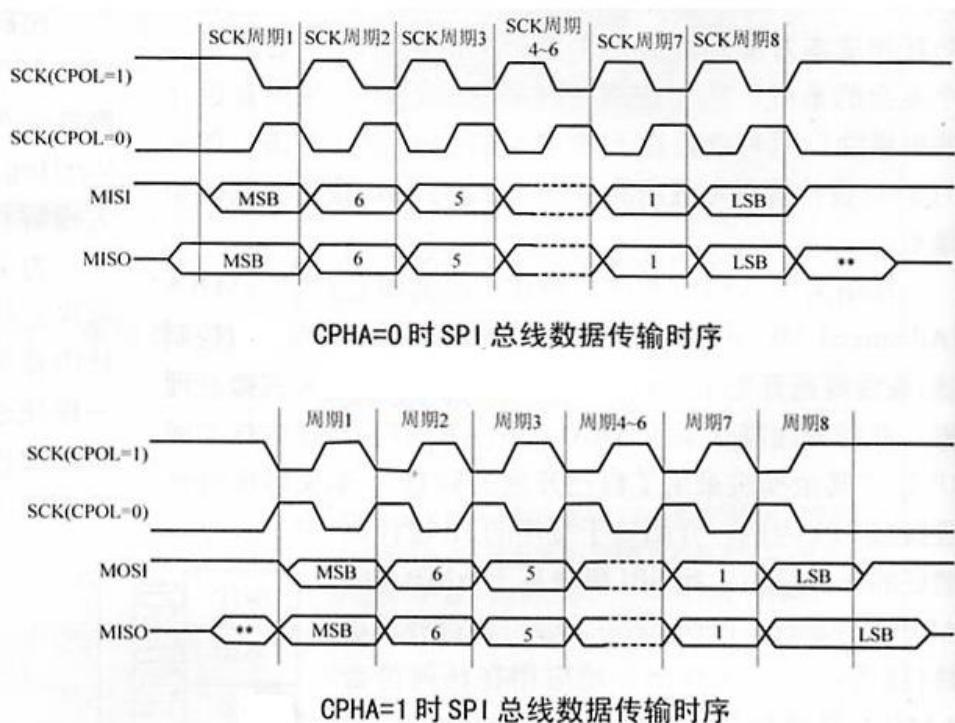


图 30.1.2 不同时钟相位下的总线传输时序 (CPHA=0/1)

STM32F4 的 SPI 功能很强大，SPI 时钟最高可以到 37.5Mhz，支持 DMA，可以配置为 SPI 协议或者 I2S 协议（支持全双工 I2S）。

本章，我们将使用 STM32F4 的 SPI 来读取外部 SPI FLASH 芯片（W25Q128），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，STM32F4 的 SPI 详细介绍请参考《STM32F4xx 中文参考手册》第 721 页，27 节。然后我们再介绍下 SPI FLASH 芯片。

这节，我们使用 STM32F4 的 SPI1 的主模式，下面就来看看 SPI1 部分的设置步骤吧。SPI 相关的库函数和定义分布在文件 `stm32f4xx_spi.c` 以及头文件 `stm32f4xx_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 SPI1 时钟。

我们要用 SPI1，第一步就要使能 SPI1 的时钟，SPI1 的时钟通过 APB2ENR 的第 12 位来设置。其次要设置 SPI1 的相关引脚为复用(AF5)输出，这样才会连接到 SPI1 上。这里我们使用的是 PB3、4、5 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，复用功能为 AF5。

使能 SPI1 时钟的方法为：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE); //使能 SPI1 时钟
```

复用 PB3,PB4,PB5 为 SPI1 引脚的方法为：

```
GPIO_PinAFConfig(GPIOB,GPIO_PinSource3,GPIO_AF_SPI1); //PB3 复用为 SPI1
GPIO_PinAFConfig(GPIOB,GPIO_PinSource4,GPIO_AF_SPI1); //PB4 复用为 SPI1
GPIO_PinAFConfig(GPIOB,GPIO_PinSource5,GPIO_AF_SPI1); //PB5 复用为 SPI1
```

同时我们要设置相应的引脚模式为复用功能模式：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
```

2) 初始化 SPI1,设置 SPI1 工作模式等。

这一步全部是通过 SPI1_CR1 来设置，我们设置 SPI1 为主机模式，设置数据格式为 8 位，然后通过 CPOL 和 CPHA 位来设置 SCK 时钟极性及采样方式。并设置 SPI1 的时钟频率（最大 37.5Mhz），以及数据的格式（MSB 在前还是 LSB 在前）。在库函数中初始化 SPI 的函数为：

```
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 SPI1。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，接下来我们简单讲解一下：

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 2 分频到 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 $84M/256=328.125KHz$ 。

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 SPI1。

这一步通过 SPI1_CR1 的 bit6 来设置，以启动 SPI1，在启动之后，我们就可以开始 SPI 通讯了。库函数使能 SPI1 的方法为：

```
SPI_Cmd(SPI1, ENABLE); //使能 SPI1 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 数据寄存器读出接受到的数据。

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(SPI1, SPI_I2S_FLAG_RXNE);
```

SPI1 的使用就介绍到这里，接下来介绍一下 W25Q128。W25Q128 是华邦公司推出的大容量 SPI FLASH 产品，W25Q128 的容量为 128Mb，该系列还有 W25Q80/16/32/64 等。ALIENTEK 所选择的 W25Q128 容量为 128Mb，也就是 16M 字节。

W25Q128 将 16M 的容量分为 256 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。W25Q128 的最小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 W25Q128 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。

W25Q128 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，W25Q128 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 W25Q128 的介绍，请参考 W25Q128 的 DATASHEET。

30.2 硬件设计

本章实验功能简介：开机的时候先检测 W25Q128 是否存在，然后在主循环里面检测两个按键，其中 1 个按键（KEY1）用来执行写入 W25Q128 的操作，另外一个按键（KEY0）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) KEY_UP 和 KEY1 按键
- 3) TFTLCD 模块
- 4) SPI
- 5) W25Q128

这里只介绍 W25Q128 与 STM32F4 的连接，板上的 W25Q128 是直接连在 STM32F4 的 SPI1 上的，连接关系如图 30.2.1 所示：

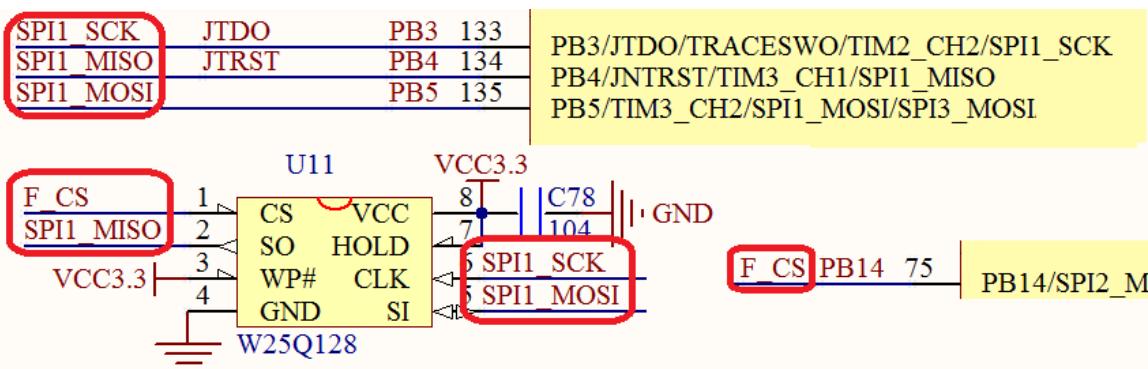


图 30.2.1 STM32F4 与 W25Q128 连接电路图

这里，我们的 F_CS 是连接在 PB14 上面的，另外要特别注意：W25Q128 和 NRF24L01 共用 SPI1，所以这两个器件在使用的时候，必须分时复用（通过片选控制）才行。

30.3 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 stm32f4xx_spi.c 文件以及头文件 stm32f4xx_spi.h。

打开 spi.c 文件，看到如下代码：

```
//以下是 SPI 模块的初始化代码，配置成主机模式
//SPI 口初始化
//这里针是对 SPI1 的初始化
void SPI1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef SPI_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE); //使能 SPI1 时钟

    //GPIOFB3,4,5 初始化设置：复用功能输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5;//PB3~5
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOB, &GPIO_InitStructure);// 初始化

//配置引脚复用映射
GPIO_PinAFConfig(GPIOB,GPIO_PinSource3,GPIO_AF_SPI1); //PB3 复用为 SPI1
GPIO_PinAFConfig(GPIOB,GPIO_PinSource4,GPIO_AF_SPI1); //PB4 复用为 SPI1
GPIO_PinAFConfig(GPIOB,GPIO_PinSource5,GPIO_AF_SPI1); //PB5 复用为 SPI1

//这里只针对 SPI 口初始化
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SPI1,ENABLE);//复位 SPI1
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SPI1,DISABLE);//停止复位 SPI1

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:主 SPI
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b; //设置 SPI 的数据大小: 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; //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(SPI1, &SPI_InitStructure); //根据指定的参数初始化外设 SPIx 寄存器

SPI_Cmd(SPI1, ENABLE); //使能 SPI1
SPI1_ReadWriteByte(0xff); //启动传输
}

//SPI1 速度设置函数
//SPI 速度=fAPB2/分频系数
//入口参数范围: @ref SPI_BaudRate_Prescaler
//SPI_BaudRatePrescaler_2~SPI_BaudRatePrescaler_256
//fAPB2 时钟一般为 84Mhz:
void SPI1_SetSpeed(u8 SPI_BaudRatePrescaler)
{
    assert_param(IS_SPI_BAUDRATE_PRESCALER(SPI_BaudRatePrescaler)); //判断有效性
    SPI1->CR1&=0xFFC7;//位 3-5 清零, 用来设置波特率
    SPI1->CR1|=SPI_BaudRatePrescaler; //设置 SPI1 速度
    SPI_Cmd(SPI1,ENABLE); //使能 SPI1
}
//SPI1 读写一个字节
//TxData:要写入的字节
```

```
//返回值:读取到的字节
u8 SPI1_ReadWriteByte(u8 TxData)
{
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET){} //等待发送区空
    SPI_I2S_SendData(SPI1, TxData); //通过外设 SPIx 发送一个 byte 数据
    while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET){} //等待接收完
    return SPI_I2S_ReceiveData(SPI1); //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 SPI1，所以在 SPI1_Init 函数里面，其相关的操作都是针对 SPI1 的，其初始化步骤和我们上面介绍的一样。在初始化之后，我们就可以开始使用 SPI1 了，这里特别注意，SPI 初始化函数的最后有一个启动传输，这句话最大的作用就是维持 MOSI 为高电平，而且这句话也不是必须的，可以去掉。

在 SPI1_Init 函数里面，把 SPI1 的频率设置成了最低（84Mhz，256 分频）。在外部函数里面，我们通过 SPI1_SetSpeed 来设置 SPI1 的速度，而我们的数据发送和接收则是通过 SPI1_ReadWriteByte 函数来实现的。

接下来我们来看看 w25qxx.c 文件内容。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 W25QXX_Read 函数，该函数用于从 W25Q128 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 W25QXX_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead)
{
    u16 i;
    W25QXX_CS=0;                                //使能器件
    SPI1_ReadWriteByte(W25X_ReadData);           //发送读取命令
    SPI1_ReadWriteByte((u8)((ReadAddr)>>16)); //发送 24bit 地址
    SPI1_ReadWriteByte((u8)((ReadAddr)>>8));
    SPI1_ReadWriteByte((u8)ReadAddr);
    for(i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPI1_ReadWriteByte(0xFF); //循环读数
    }
    W25QXX_CS=1;
}
```

由于 W25Q128 支持以任意地址（但是不能超过 W25Q128 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 W25Q128 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 W25QXX_Write 这个函数，该函数的作用与 W25QXX_Flash_Read 的作用类似，不过是用来写数据到 W25Q128 里面的，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 W25QXX_BUFFER[4096];
void W25QXX_Write(u8* pBuffer,u32 WriteAddr,u16 NumByteToWrite)
{
    u32 secpos;
    u16 secoff; u16 secremain; u16 i;
    u8 * W25QXX_BUF;
    W25QXX_BUF=W25QXX_BUFFER;
    secpos=WriteAddr/4096;//扇区地址
    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)
    {
        W25QXX_Read(W25QXX_BUF,secpos*4096,4096);//读出整个扇区的内容
        for(i=0;i<secremain;i++)//校验数据
        {
            if(W25QXX_BUF[secoff+i]!=0xFF)break;//需要擦除
        }
        if(i<secremain)//需要擦除
        {
            W25QXX_Erase_Sector(secpos);//擦除这个扇区
            for(i=0;i<secremain;i++)          //复制
            {
                W25QXX_BUF[i+secoff]=pBuffer[i];
            }
            W25QXX_Write_NoCheck(W25QXX_BUF,secpos*4096,4096);//写入整个扇区
        }else W25QXX_Write_NoCheck(pBuffer,WriteAddr,secremain);//已擦除的,直接写
        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 W25Q128 的任意地址开始写入任意长度（必须不超过 W25Q128 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，如果不要，则直接写入数据即可，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。这里我们还定义了一个 W25QXX_BUFFER 的全局变量，用于擦除时缓存扇区内的数据。

其他的代码就比较简单了，我们这里不介绍了。对于头文件 w25qxx.h，这里面就定义了一些与 W25Q128 操作相关的命令和函数（部分省略了），这些命令在 W25Q128 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。

最后，我们看看 main 函数，代码如下：

```

//要写入到 W25Q128 的字符串数组
const u8 TEXT_Buffer[]{"Explorer STM32F4 SPI TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key, datatemp[SIZE];
    u16 i=0;
    u32 FLASH_SIZE;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    W25QXX_Init(); //W25QXX 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"SPI TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/7");
    LCD_ShowString(30,130,200,16,16,"KEY1:Write KEY0:Read"); //显示提示信息
    while(W25QXX_ReadID()!=W25Q128) //检测不到 W25Q128
    {
        LCD_ShowString(30,150,200,16,16,"W25Q128 Check Failed!");
        delay_ms(500);
        LCD_ShowString(30,150,200,16,16,"Please Check! ");
    }
}

```

```
delay_ms(500);
LED0=!LED0;           //DS0 闪烁
}
LCD_ShowString(30,150,200,16,16,"W25Q128 Ready!");
FLASH_SIZE=128*1024*1024; //FLASH 大小为 2M 字节
POINT_COLOR=BLUE;          //设置字体为蓝色
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY1_PRES)//KEY1 按下,写入 W25Q128
    {
        LCD_Fill(0,170,239,319,WHITE);//清除半屏
        LCD_ShowString(30,170,200,16,16,"Start Write W25Q128....");
        W25QXX_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(30,170,200,16,16,"W25Q128 Write Finished");//提示完成
    }
    if(key==KEY0_PRES)//KEY0 按下,读取字符串并显示
    {
        LCD_ShowString(30,170,200,16,16,"Start Read W25Q128.... ");
        W25QXX_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(30,170,200,16,16,"The Data Readed Is:   ");
        LCD_ShowString(30,190,200,16,16,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。

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 探索者 STM32F4 开发板上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 30.4.1 所示：

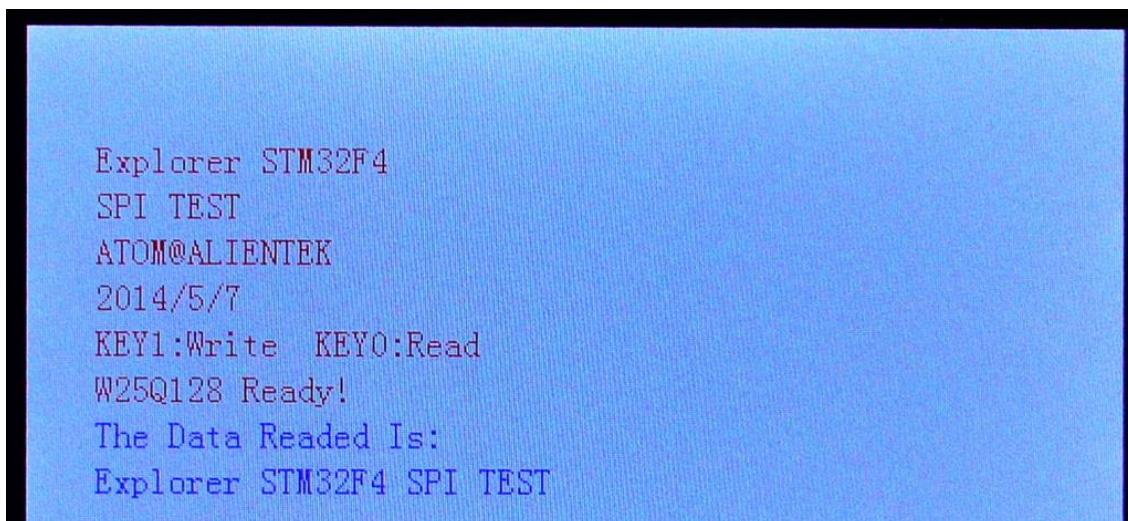


图 30.4.1 SPI 实验程序运行效果图

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 W25Q128 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 PB4 和 PB5 短接就可以看到报错了。

第三十一章 485 实验

本章我们将向大家介绍如何使用 STM32F4 的串口实现 485 通信（半双工）。在本章中，我们将使用 STM32F4 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：

- 31.1 485 简介
- 31.2 硬件设计
- 31.3 软件设计
- 31.4 下载验证

31.1 485 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 $+ (2\sim 6)V$ 表示；逻辑“0”以两线间的电压差为 $- (2\sim 6)V$ 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 $\leq 100Kbps$) 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 个终端匹配电阻，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 31.1.2 所示：

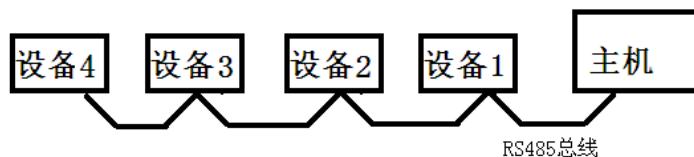


图 31.1.2 RS485 连接

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。

探索者 STM32F4 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 31.1.2 所示：

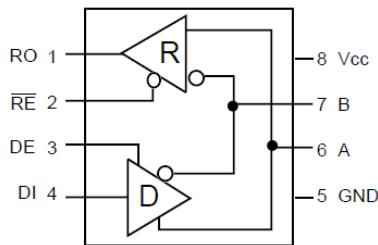


图 31.1.2 SP3485 框图

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 STM32F4 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：通过连接两个探索者 STM32F4 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 42Mhz。

31.2 硬件设计

本章要用到的硬件资源如下：

- 1) 指示灯 DSO
- 2) KEY0 按键
- 3) TFTLCD 模块
- 4) 串口 2
- 5) RS485 收发芯片 SP3485

前面 3 个之前都已经详细介绍过了，这里我们介绍 SP3485 和串口 2 的连接关系，如图 31.2.1 所示：

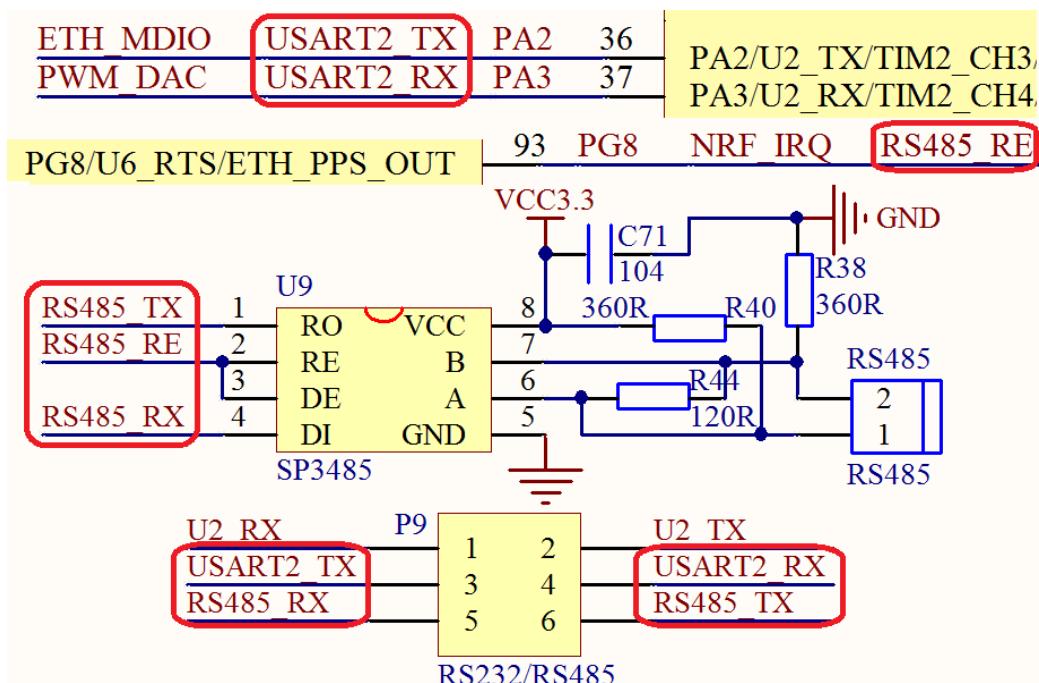


图 31.2.1 STM32F4 与 SP3485 连接电路图

从上图可以看出：STM32F4 的串口 2 通过 P9 端口设置，连接到 SP3485，通过 STM32F4 的 PG8 控制 SP3485 的收发，当 PG8=0 的时候，为接收模式；当 PG8=1 的时候，为发送模式。这里需要注意，PA2，PA3 和 ETH_MDIO 和 PWM_DAC 有共用 IO，所以在使用的时候，注意分时复用，不能同时使用。另外 RS485_RX 信号，也和 NRF_IRQ 共用 PG8，所以他们也不可以同时使用，只能分时复用。

另外，图中的 R38 和 R40 是两个偏置电阻，用来保证总线空闲时，A、B 之间的电压差都会大于 200mV（逻辑 1）。从而避免因总线空闲时，A、B 压差不定，引起逻辑错乱，可能出现的乱码。

然后，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 485_TX 和 485_RX 上面，如图 31.2.2 所示：

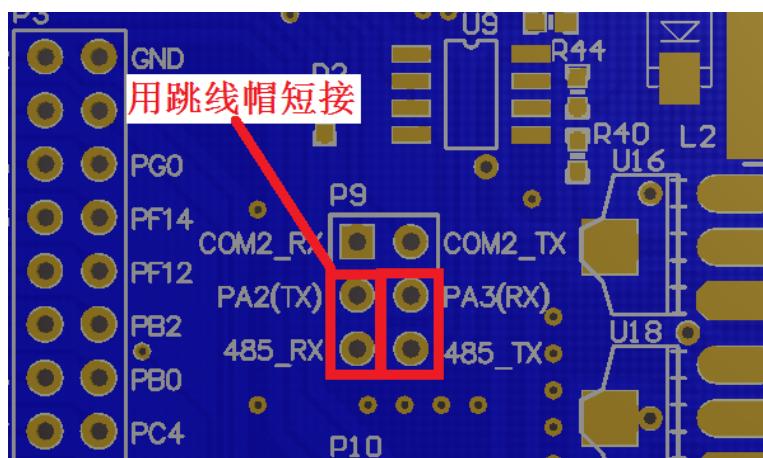


图 31.2.2 硬件连接示意图

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！！

31.3 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 通信因为底层用的是串口 2，所以需要引入库函数 stm32f4xx_usart.c 文件和对应的头文件 stm32f4xx_usart.h。

打开 rs485.c 文件，代码如下：

```
#if 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 USART2->DR
        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_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE); //使能 PA 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE); //使能 USART2 时钟

    //串口 2 引脚复用映射
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource2,GPIO_AF_USART2); //PA2 复用为 USART2
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource3,GPIO_AF_USART2); //PA3 复用为 USART2

    //USART2
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3; //GPIOA2 与 GPIOA3
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
```

```
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA2, PA3

//PG8 推挽输出, 485 模式控制
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8; //GPIOG8
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOG,&GPIO_InitStructure); //初始化 PG8

//USART2 初始化设置
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
USART_Init(USART2, &USART_InitStructure); //初始化串口 2

USART_Cmd(USART2, ENABLE); //使能串口 2
USART_ClearFlag(USART2, USART_FLAG_TC);

#if EN_USART2_RX
    USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);//开启接受中断

    //Usart2 NVIC 配置
    NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=3;//抢占优先级 3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority =3; //响应优先级 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、
#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++)       //循环发送数据
    {
        while(USART_GetFlagStatus(USART2,USART_FLAG_TC)==RESET); //等待发送结束
        USART_SendData(USART2,buf[t]); //发送数据
    }
    while(USART_GetFlagStatus(USART2,USART_FLAG_TC)==RESET); //等待发送结束
    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 PG8 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 文件中，我们通过下面一行代码打开了接受中断：

```
#define EN_USART2_RX 1      //0,不接收;1,接收.
```

其他内容就是一些函数什么，所以这里我们不细说。

接下来，我们来看看主函数代码：

```
int main(void)
{
    u8 key, i=0,t=0, cnt=0, rs485buf[5];
```

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
delay_init(168); //初始化延时函数
uart_init(115200); //初始化串口波特率为 115200
LED_Init(); //初始化 LED
LCD_Init(); //LCD 初始化
KEY_Init(); //按键初始化
RS485_Init(9600); //初始化 RS485 串口 2
POINT_COLOR=RED; //设置字体为红色
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"RS485 TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/7");
LCD_ShowString(30,130,200,16,16,"KEY0:Send"); //显示提示信息
POINT_COLOR=BLUE; //设置字体为蓝色
LCD_ShowString(30,150,200,16,16,"Count:"); //显示当前计数值
LCD_ShowString(30,170,200,16,16,"Send Data:"); //提示发送的数据
LCD_ShowString(30,210,200,16,16,"Receive Data:"); //提示接收到的数据
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY0_PRES)//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(30+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(30+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(30+48,150,cnt,3,16,0X80); //显示数据
    }
}
```

```
    }
}
}
```

此部分代码，我们通过函数 RS485_Init(9600)，初始化串口 2 的波特率为 9600。cnt 是一个累加数，一旦 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。

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 探索者 STM32F4 开发板上（注意要 2 个开发板都下载这个代码哦），得到如图 31.4.1 所示：

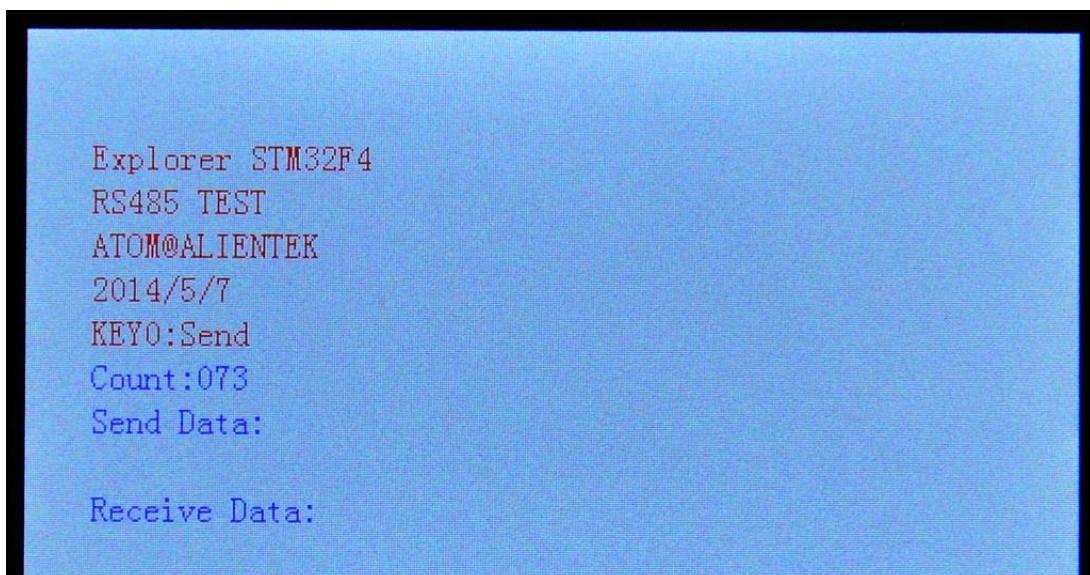


图 31.4.1 程序运行效果图

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 31.4.2 和图 31.4.3 所示：

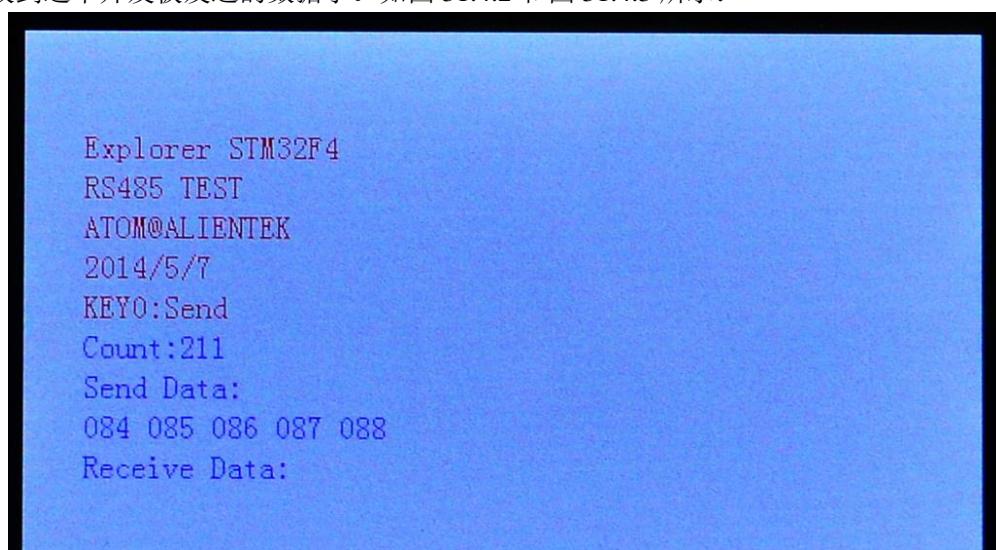


图 31.4.2 RS485 发送数据

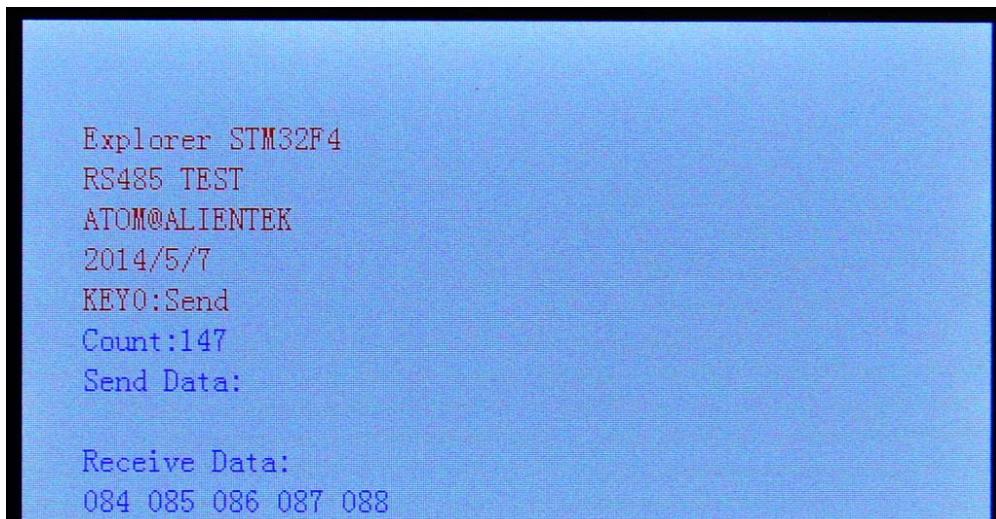


图 31.4.3 RS485 接收数据

图 31.4.2 来自开发板 A，发送了 5 个数据，图 31.4.3 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。

第三十二章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32F4 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：

- 32.1 CAN 简介
- 32.2 硬件设计
- 32.3 软件设计
- 32.4 下载验证

32.1 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 以下称为 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) **系统的柔韧性。**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 500Kbps 的通信速率，使用的是 ISO11898 标准，该标准的物理层特征如图 32.1.1 所示：

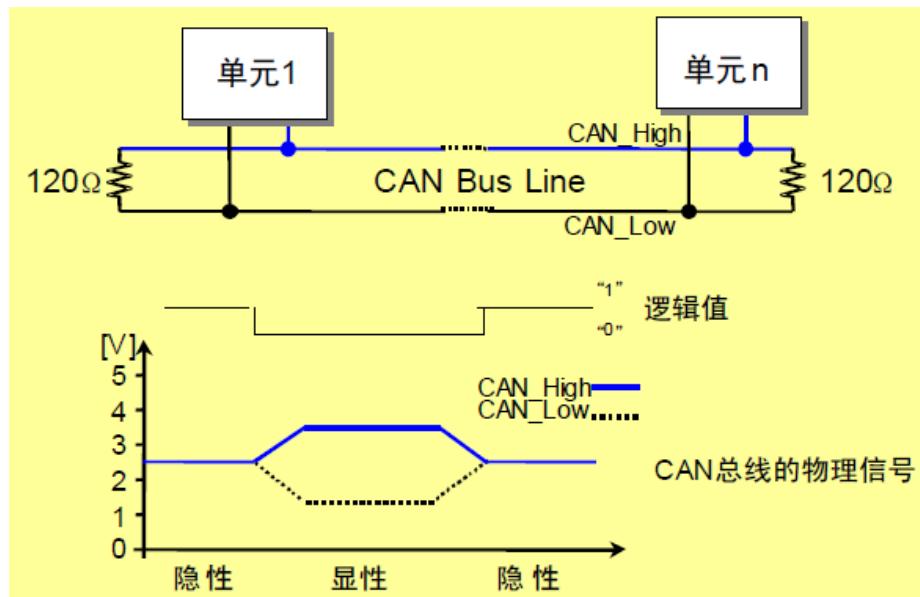


图 32.1.1 ISO11898 物理层特性

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
- 遥控帧
- 错误帧
- 过载帧
- 间隔帧

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 32.1.1 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 32.1.1 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 32.1.2 所示：

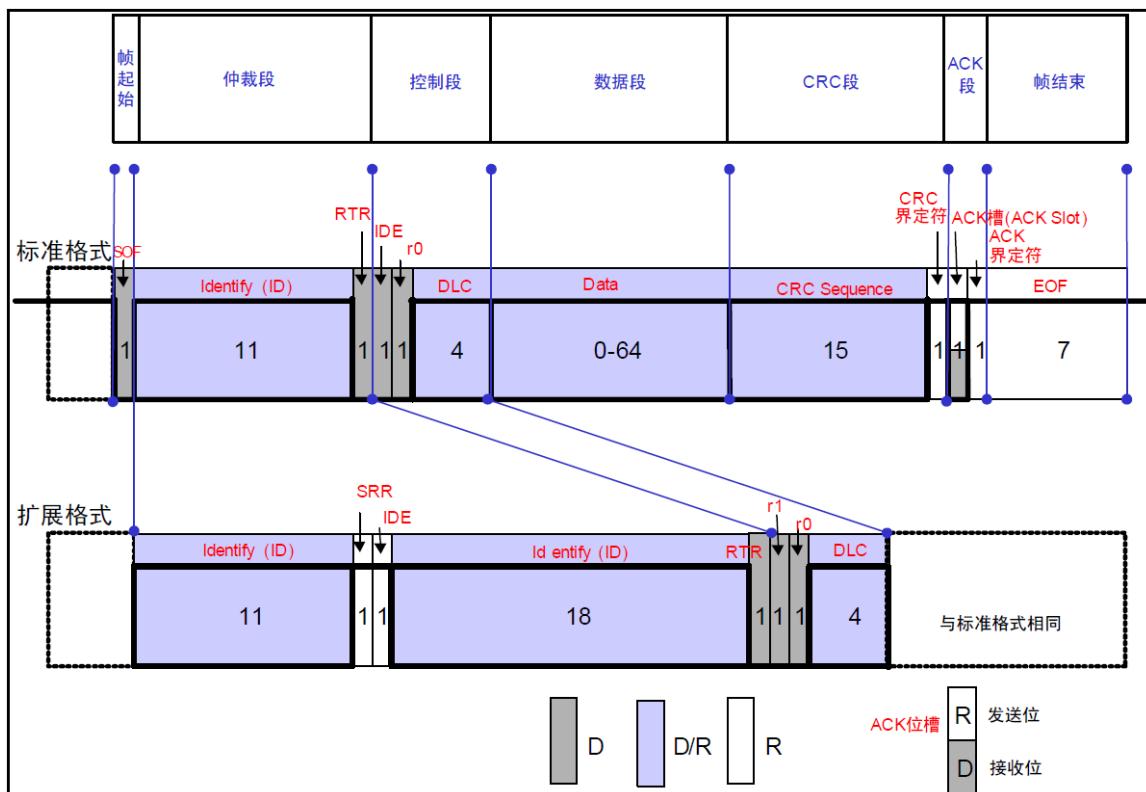


图 32.1.2 数据帧的构成

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 32.1.3 所示：

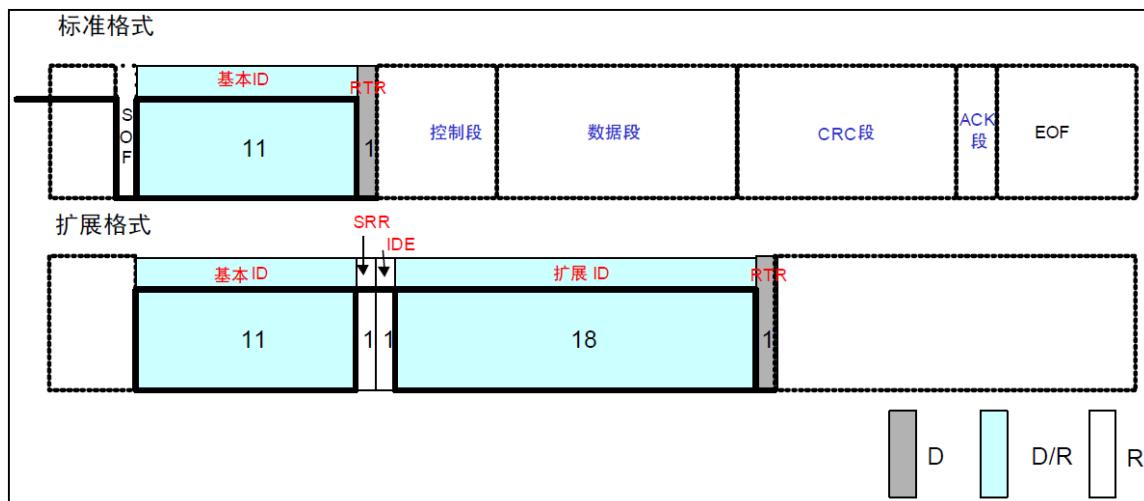


图 32.1.3 数据帧仲裁段构成

标准格式的 ID 有 11 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 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 32.1.4 所示：

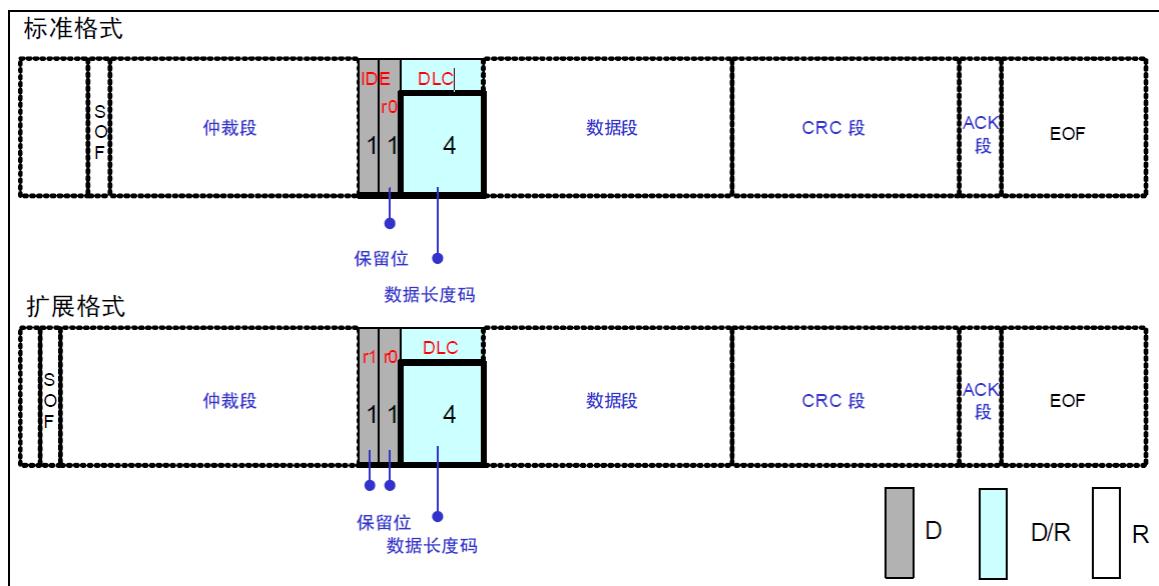


图 32.1.4 数据帧控制段构成

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 32.1.5 所示：

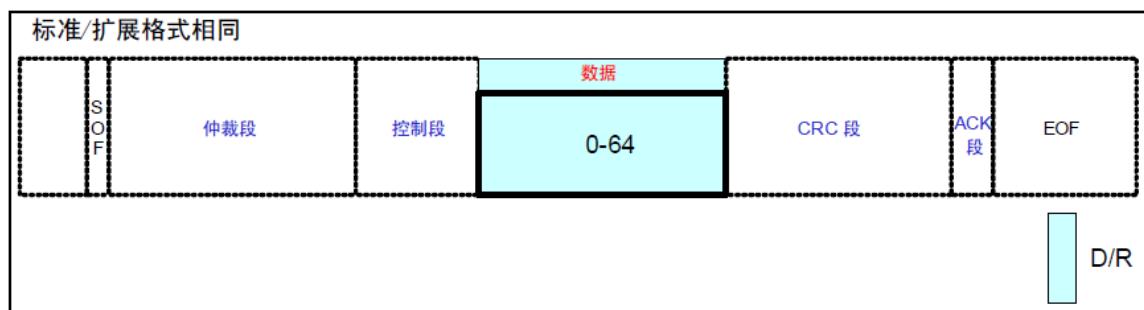


图 32.1.5 数据帧数据段构成

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 32.1.6 所示：

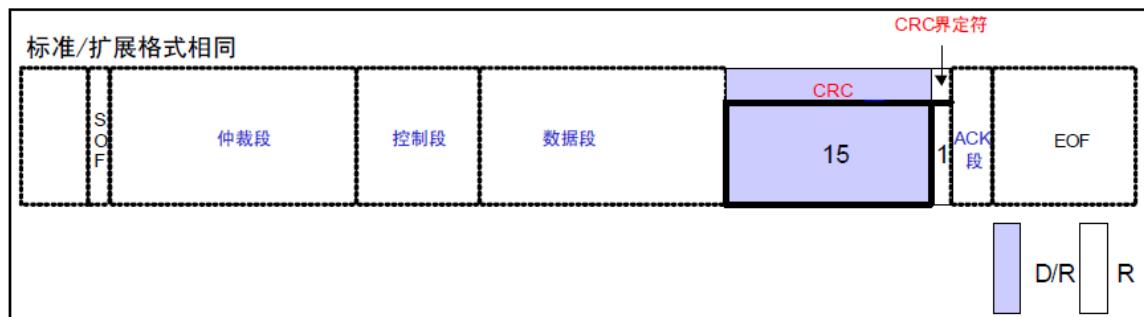


图 32.1.6 数据帧 CRC 段构成

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 32.1.7 所示：

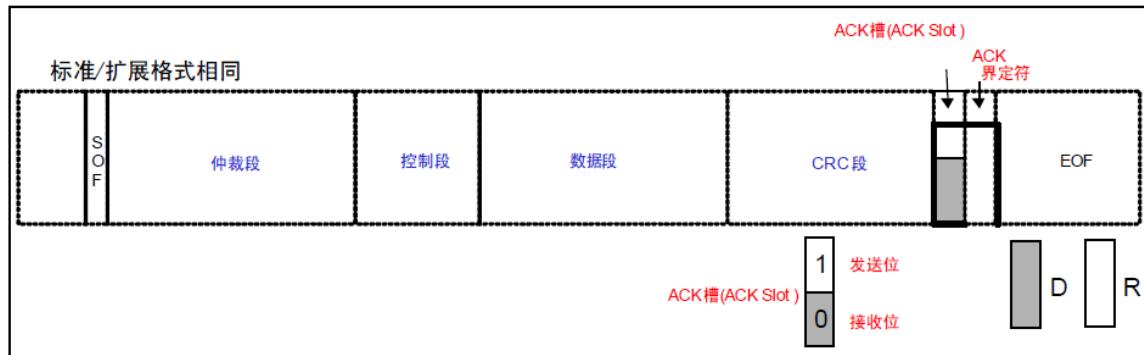


图 32.1.7 数据帧 CRC 段构成

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 32.1.2 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 32.1.2 一个位各段及其作用

1 个位的构成如图 32.1.8 所示：

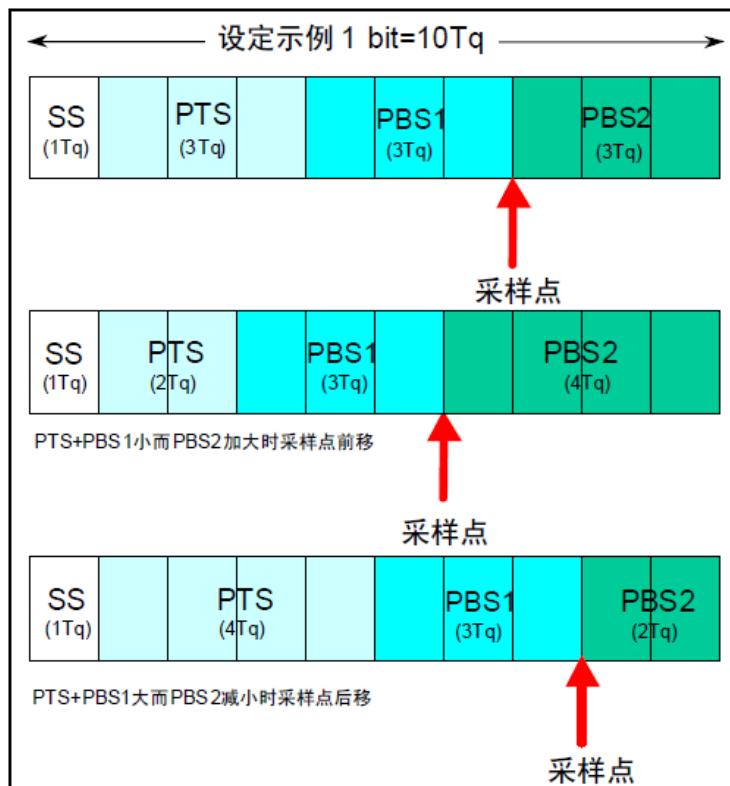


图 32.1.8 一个位的构成

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 32.1.9 所示：

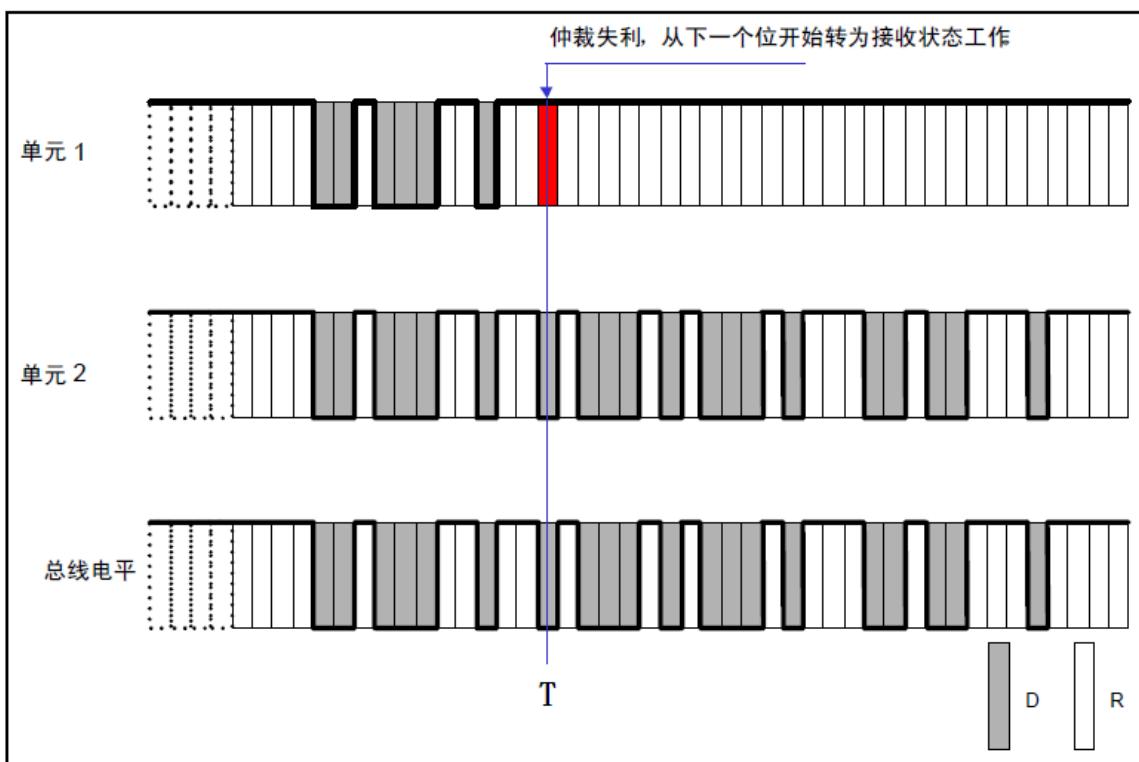


图 32.1.9 CAN 总线仲裁过程

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 STM32F4 的 CAN 控制器。

STM32F4 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。

STM32F4 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（28 个， CAN1 和 CAN2 共享）

在 STM32F407ZGT6 中，带有 2 个 CAN 控制器，而我们本章只用了 1 个 CAN，即 CAN1。双 CAN 的框图如图 32.1.10 所示：

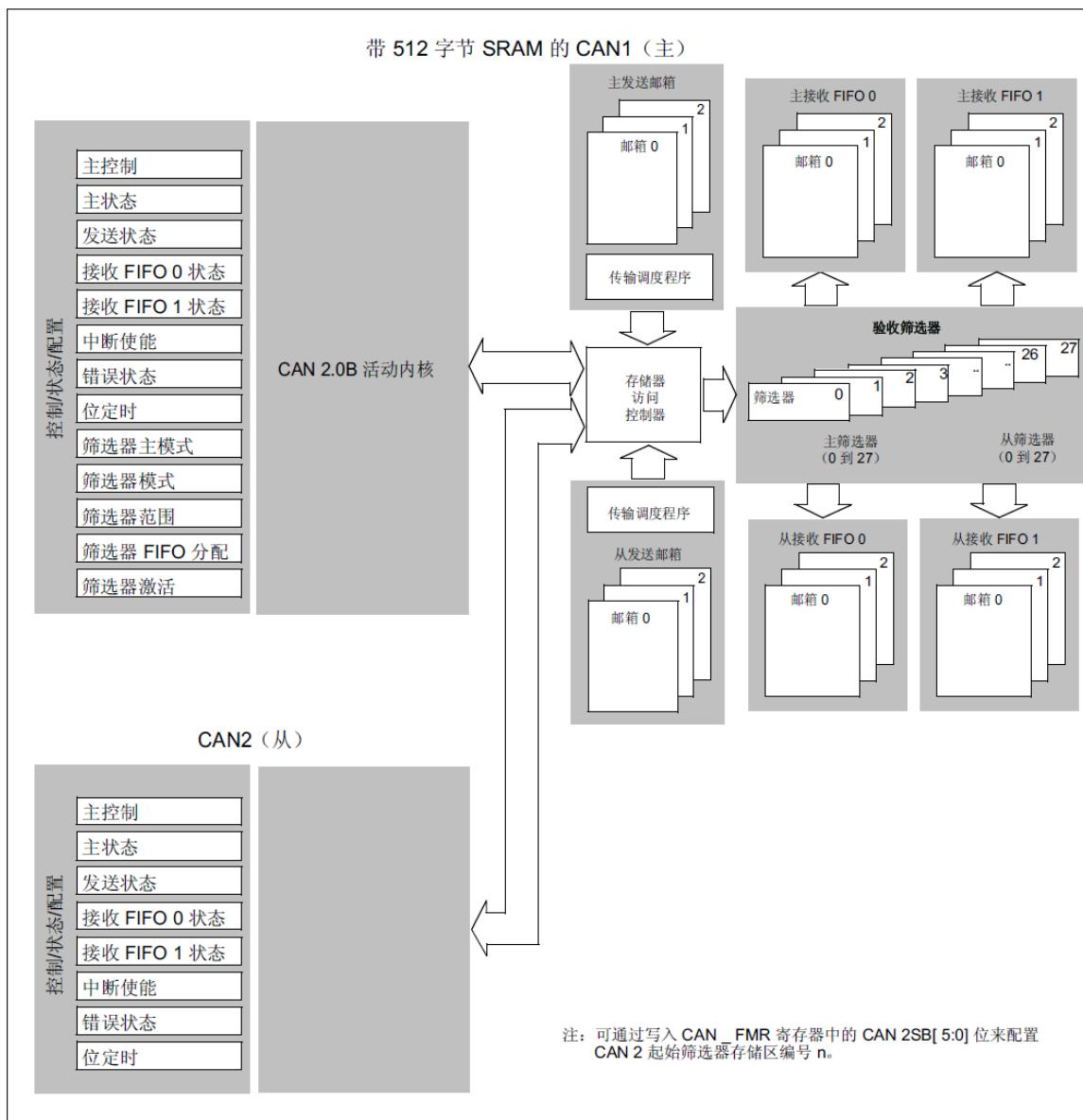


图 32.1.10 双 CAN 框图

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。

STM32F4 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。STM32F4 的过滤器（也称筛选器）组最多有 28 个，每个过滤器组 x 由 2 个 32 位寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32F4 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 32.1.11 所示：

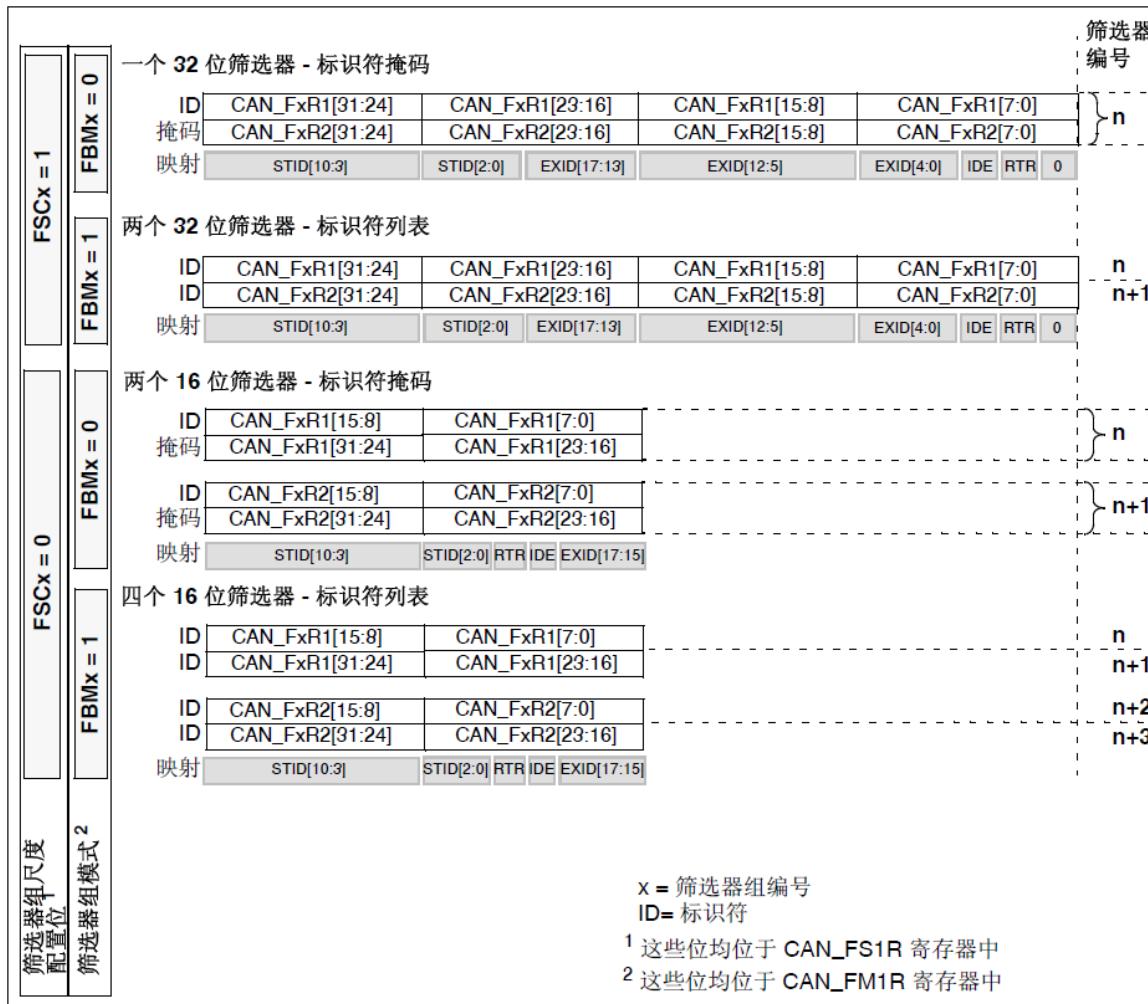


图 32.1.11 过滤器组位宽模式设置

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 32.1.11 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_F0R1=0xFFFF0000, CAN_F0R2=0xFF00FF00。其中存放到 CAN_F0R1 的值就是期望收到的 ID，即我们希望收到的 ID (STID+EXTID+IDE+RTR) 最好是：0xFFFF0000。而 0xFF00FF00 就是设置我们需要关心的 ID，表示收到的 ID，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_F0R1 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 ID 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《STM32F4xx 中文参考手册》的 24.7.4 节 (616 页)。接下来，我们看看 STM32F4 的 CAN 发送和接收的流程。

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 32.1.12 所示：

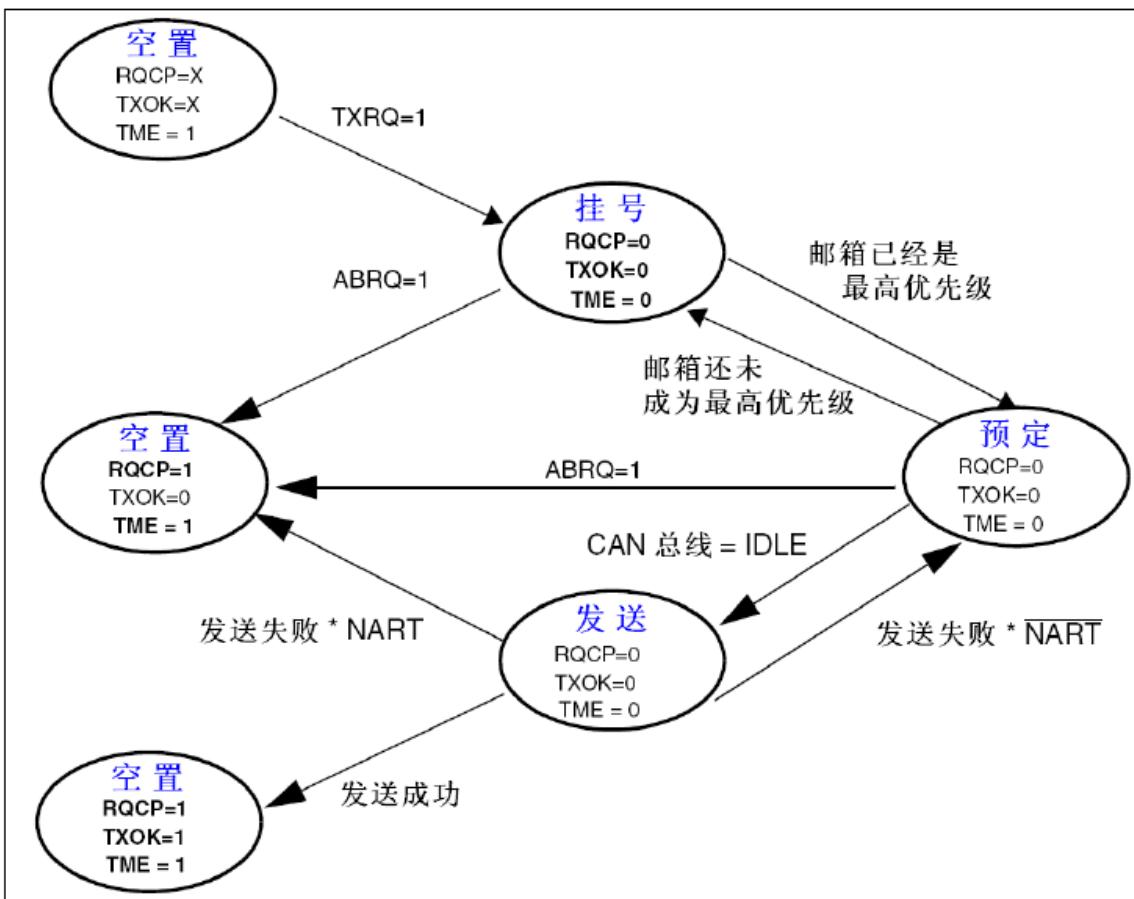


图 32.1.12 发送邮箱

上图中，还包含了很多其他处理，终止发送（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。

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 32.1.13 所示：

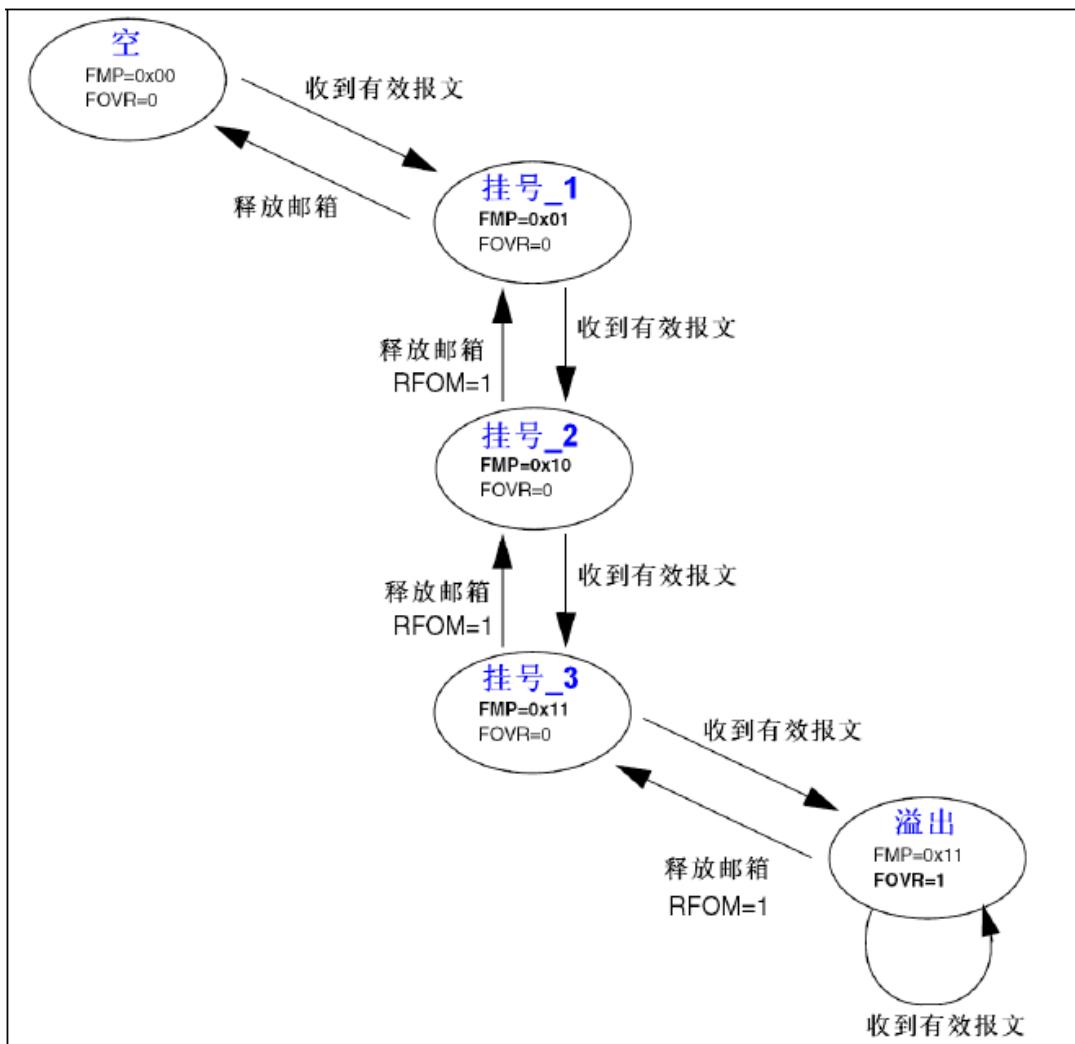


图 32.1.13 FIFO 接收报文

FIFO 接收到的报文数，我们可以通过查询 CAN_RFxR 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 STM32F4 的 CAN 位时间特性，STM32F4 的 CAN 位时间特性和之前我们介绍的，稍有点区别。STM32F4 把传播时间段和相位缓冲段 1 (STM32F4 称之为时间段 1) 合并了，所以 STM32F4 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)。STM32F4 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。STM32F4 的 CAN 位时序如图 32.1.14 所示：

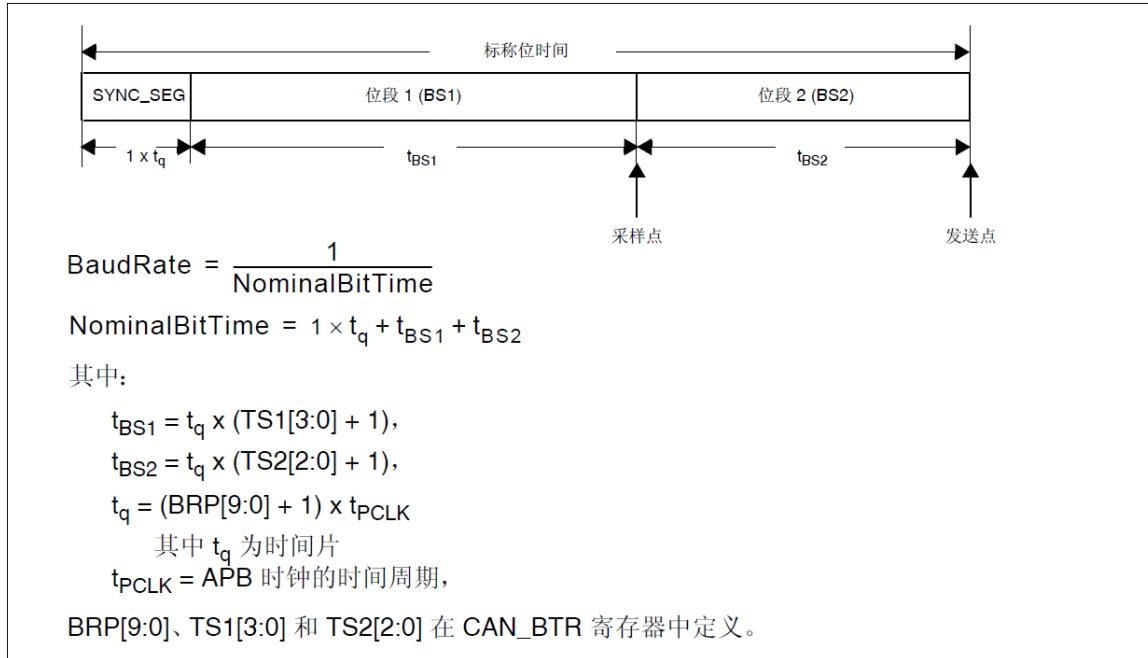


图 32.1.14 STM32F4 CAN 位时序

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 42Mhz)，就可以方便的计算出波特率。比如设置 TS1=6、TS2=5 和 BRP=5，在 APB1 频率为 42Mhz 的条件下，即可得到 CAN 通信的波特率=42000/[(7+6+1)*6]=500Kbps。

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 32.1.15：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved														DBF	
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESET	Reserved						TTCM	ABOM	AWUM	NART	RFLM	TXFP	SLEEP	INRQ	
rs							rw	rw	rw						

图 32.1.15 寄存器 CAN_MCR 各位描述

该寄存器的详细描述，请参考《STM32F4xx 中文参考手册》24.9.2 节 (625 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 位清‘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 32.1.16 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	Reserved				SJW[1:0]	Res.	TS2[2:0]			TS1[3:0]				
						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
Reserved				BRP[9:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31 **SILM:** 静默模式 (调试) (Silent mode (debug))

0: 正常工作

1: 静默模式

位 30 **LBKM:** 环回模式 (调试) (Loop back mode (debug))

0: 禁止环回模式

1: 使能环回模式

位 29:26 保留, 必须保持复位值。

位 25:24 **SJW[1:0]:** 再同步跳转宽度 (Resynchronization jump width)

这些位定义 CAN 硬件在执行再同步时最多可以将位加长或缩短的时间片数目。

$$t_{RJW} = t_{CAN} \times (SJW[1:0] + 1)$$

位 23 保留, 必须保持复位值。

位 22:20 **TS2[2:0]:** 时间段 2 (Time segment 2)

这些位定义时间段 2 中的时间片数目。

$$t_{BS2} = t_{CAN} \times (TS2[2:0] + 1)$$

位 19:16 **TS1[3:0]:** 时间段 1 (Time segment 1)

这些位定义时间段 1 中的时间片数目。

$$t_{BS1} = t_{CAN} \times (TS1[3:0] + 1)$$

位 15:10 保留, 必须保持复位值。

位 9:0 **BRP[9:0]:** 波特率预分频器 (Baud rate prescaler)

这些位定义一个时间片的长度。

$$t_q = (BRP[9:0]+1) \times t_{PCLK}$$

图 32.1.16 寄存器 CAN_BTR 各位描述

STM32F4 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 32.1.17 所示:

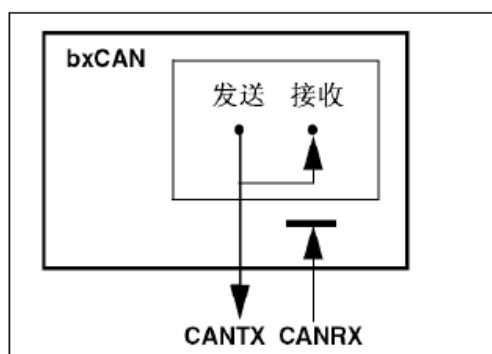


图 32.1.17 CAN 环回模式

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=0~3), 该寄存器各位描

述如图 32.1.18 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)
标准标识符或扩展标识符的 MSB (取决于 IDE 位的值)。

位 20:3 **EXID[17:0]**: 扩展标识符 (Extended identifier)

扩展标识符的 LSB。

位 2 **IDE**: 标识符扩展 (Identifier extension)

此位用于定义邮箱中消息的标识符类型。

0: 标准标识符。

1: 扩展标识符。

位 1 **RTR**: 远程发送请求 (Remote transmission request)

0: 数据帧

1: 遥控帧

位 0 **TXRQ**: 发送邮箱请求 (Transmit mailbox request)

由软件置 1, 用于请求发送相应邮箱的内容。

邮箱变为空后, 此位由硬件清零。

图 32.1.18 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=0~2), 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。

第五个, 我介绍的是 CAN 发送邮箱低字节数据寄存器 (CAN_TDLxR) (x=0~2), 该寄存器各位描述如图 32.1.19 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)

消息的数据字节 3。

位 23:16 **DATA2[7:0]**: 数据字节 2 (Data byte 2)

消息的数据字节 2。

位 15:8 **DATA1[7:0]**: 数据字节 1 (Data byte 1)

消息的数据字节 1。

位 7:0 **DATA0[7:0]**: 数据字节 0 (Data byte 0)

消息的数据字节 0。

一条消息可以包含 0 到 8 个数据字节, 从字节 0 开始。

图 32.1.19 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_TDhxR, 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_TDhxR 的各位描述同 CAN_TDLxR 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_RDHxR) 分别和发送邮箱的：CAN_TDTxR、CAN_TDLxR 以及 CAN_TDhxR 类似，这里我们就不单独一一介绍了。详细介绍，请参考《STM32F4xx 中文参考手册》24.9.3 节 (635 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 32.1.20 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				rw											
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，必须保持复位值。

位 27:0 **FBMx**: 筛选器模式 (Filter mode)

筛选器 x 的寄存器的模式

0: 筛选器存储区 x 的两个 32 位寄存器处于标识符屏蔽模式。

1: 筛选器存储区 x 的两个 32 位寄存器处于标识符列表模式。

图 32.1.20 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。

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 32.1.21 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				rw											
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，必须保持复位值。

位 27:0 **FSCx**: 筛选器尺度配置 (Filter scale configuration)

这些位定义了筛选器 13-0 的尺度配置。

0: 双 16 位尺度配置

1: 单 32 位尺度配置

图 32.1.21 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。

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 32.1.22 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
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
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 31:28 保留，必须保持复位值。

位 27:0 **FFAx**: 筛选器 x 的筛选器 FIFO 分配 (Filter FIFO assignment for filter x)

通过此筛选器的消息将存储在指定的 FIFO 中。

0: 筛选器分配到 FIFO 0

1: 筛选器分配到 FIFO 1

图 32.1.22 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) (i=0~27; x=1/2)。该寄存器各位描述如图 32.1.23 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															

位 31:0 **FB[31:0]**: 筛选器位 (Filter bits)

标识符

寄存器的每一位用于指定预期标识符相应位的级别。

0: 需要显性位

1: 需要隐性位

掩码

寄存器的每一位用于指定相关标识符寄存器的位是否必须与预期标识符的相应位匹配。

0: 无关，不使用此位进行比较。

1: 必须匹配，传入标识符的此位必须与筛选器相应标识符寄存器中指定的级别相同。

图 32.1.23 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 32.1.11。

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 KEY_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，我们用一个开发板即可测试。如果是正常模式，我们就需要 2 个探索者 STM32F4 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：

1) 配置相关引脚的复用功能 (AF9)，使能 CAN 时钟。

我们要用 CAN，第一步就要使能 CAN 的时钟，CAN 的时钟通过 APB1ENR 的第 25 位来设置。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 (CAN1_RX) 和 PA12 (CAN1_TX) 为复用功能 (AF9)，并使能 PA 口的时钟。具体配置过程如下：

```
//使能相关时钟
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PORTA 时钟
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); //使能 CAN1 时钟
//初始化 GPIO
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 PA11,PA12

//引脚复用映射配置
GPIO_PinAFConfig(GPIOA, GPIO_PinSource11, GPIO_AF_CAN1); //PA11 复用为 CAN1
GPIO_PinAFConfig(GPIOA, GPIO_PinSource12, GPIO_AF_CAN1); //PA12 复用为 CAN1
```

这里需要提醒一下，CAN 发送接受引脚是哪些 IO 口，可以在中文参考手册引脚表里面查找。

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;
CAN_InitStructure.CAN_SJW=CAN_SJW_1tq;//重新同步跳跃宽度为 1 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() 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

```

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 是用来存放接受到的消息信息。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, 也比较容易理解。

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。

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。

32.2 硬件设计

本章要用到的硬件资源如下：

- 1) 指示灯 DS0
- 2) KEY0 和 KEY_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 STM32F4 与 TJA1050 连接关系，如图 32.2.1 所示：

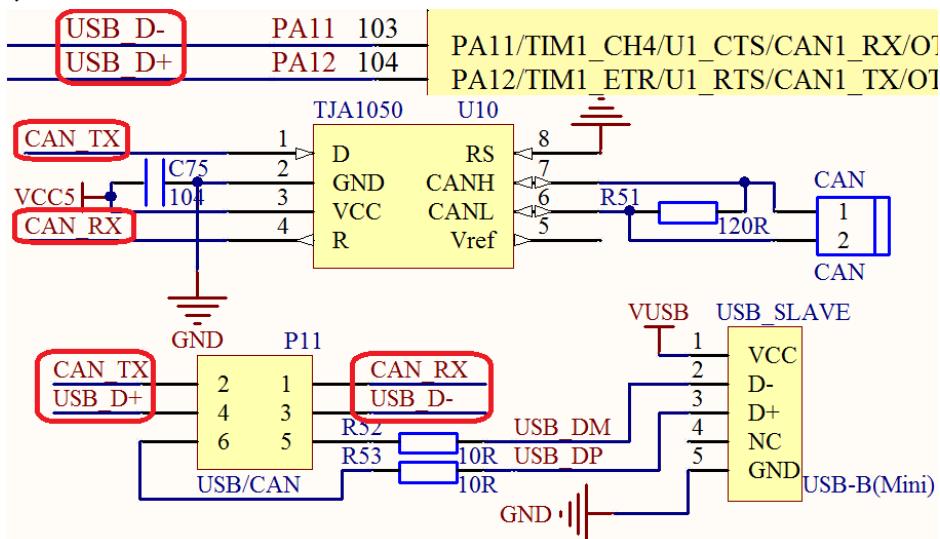


图 32.2.1 STM32F4 与 TJA1050 连接电路图

从上图可以看出：STM32F4 的 CAN 通过 P11 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，在探索者 STM32F4 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。另外，需要注意：CAN1 和 USB 共用了 PA11 和 PA12，所以他们不能同时使用。

这里还要注意，我们要设置好开发板上 P11 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 32.2.2 所示：

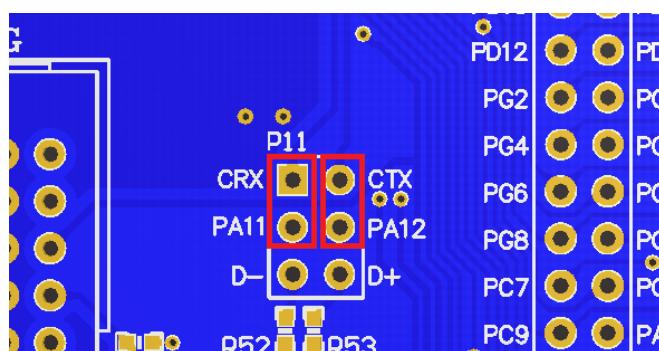


图 32.2.2 硬件连接示意图

最后，我们用 2 根导线将两个开发板 CAN 端子的 CAN_L 和 CAN_H, CAN_L 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！！

32.3 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 stm32f4xx_can.c 和头文件 stm32f4xx_can.h 中。

打开 can.c 文件，代码如下：

```
//CAN 初始化
//tsjw:重新同步跳跃时间单元. @ref CAN_synchronisation_jump_width
//tbs2:时间段 2 的时间单元.    @ref CAN_time_quantum_in_bit_segment_2
//tbs1:时间段 1 的时间单元.    @ref CAN_time_quantum_in_bit_segment_1
//brp :波特率分频器.范围:1~1024;(实际要加 1,也就是 1~1024) tq=(brp)*tpclk1
//mode: @ref CAN_operating_mode
u8 CAN1_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 CAN1_RX0_INT_ENABLE
    NVIC_InitTypeDef    NVIC_InitStructure;
#endif
    //使能相关时钟
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 PORTA 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE); //使能 CAN1 时钟
    //初始化 GPIO
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11|GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 PA11,PA12

    //引脚复用映射配置
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource11, GPIO_AF_CAN1); //PA11 复用为 CAN1
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource12, GPIO_AF_CAN1); //PA12 复用为 CAN1

    //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
    CAN_InitStructure.CAN_SJW=tsjw; //重新同步跳跃宽度
```

```
CAN_InitStructure.CAN_BS1=tbs1; //Tbs1 范围 CAN_BS1_1tq ~CAN_BS1_16tq
CAN_InitStructure.CAN_BS2=tbs2;//Tbs2 范围 CAN_BS2_1tq ~ CAN_BS2_8tq
CAN_InitStructure.CAN_Prescaler=brp; //分频系数(Fdiv)为 brp+1
CAN_Init(CAN1, &CAN_InitStructure); // 初始化 CAN1

//配置过滤器
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;
CAN_FilterInitStructure.CAN_FilterActivation=ENABLE; //激活过滤器 0
CAN_FilterInit(&CAN_FilterInitStructure);//滤波器初始化

#if CAN1_RX0_INT_ENABLE
    CAN_ITConfig(CAN1,CAN_IT_FMP0,ENABLE);//FIFO0 消息挂号中断允许.
    NVIC_InitStructure.NVIC_IRQChannel = 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 CAN1_RX0_INT_ENABLE //使能 RX0 中断
//中断服务函数
void CAN1_RX0_IRQHandler(void)
{
    CanRxMsg RxMessage;
    int i=0;
    CAN_Receive(CAN1, 0, &RxMessage);
    for(i=0;i<8;i++)
        printf("rdbuf[%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 CAN1_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++)
        TxMessage.Data[i]=msg[i];          // 第一帧信息
    mbox= CAN_Transmit(CAN1, &TxMessage);
    i=0;
    while((CAN_TransmitStatus(CAN1, mbox)==CAN_TxStatus_Failed)&&(i<0xFFFF))i++;
    if(i>=0XFFF)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 CAN1_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<RxMessage.DLC;i++)
        buf[i]=RxMessage.Data[i];
    return RxMessage.DLC;//返回接收到的数据长度
}

```

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 32.1 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收到的数据存放到 buf 中。

can.c 里面，还包含了中断接收的配置，通过 can.h 的 CAN1_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。

can.h 头文件中，CAN1_RX0_INT_ENABLE 用于设置是否使能中断接收，本章我们不用中断接收，故设置为 0。最后我们看看主函数，代码如下：

```
int main(void)
```

```
{  
    u8 key, i=0,t=0.cnt=0,u8 canbuf[8],res;  
    u8 mode=1;//CAN 工作模式;0,普通模式;1,环回模式  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2  
    delay_init(168); //初始化延时函数  
    uart_init(115200); //初始化串口波特率为 115200  
    LED_Init(); //初始化 LED  
    LCD_Init(); //LCD 初始化  
    KEY_Init(); //按键初始化  
    CAN1_Mode_Init(CAN_SJW_1tq,CAN_BS2_6tq,CAN_BS1_7tq,6,  
        CAN_Mode_LoopBack); //CAN 初始化环回模式,波特率 500Kbps  
    POINT_COLOR=RED;//设置字体为红色  
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");  
    LCD_ShowString(30,70,200,16,16,"CAN TEST");  
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(30,110,200,16,16,"2014/5/7");  
    LCD_ShowString(30,130,200,16,16,"LoopBack Mode");  
    LCD_ShowString(30,150,200,16,16,"KEY0:Send WK_UP:Mode");//显示提示信息  
    POINT_COLOR=BLUE;//设置字体为蓝色  
    LCD_ShowString(30,170,200,16,16,"Count:"); //显示当前计数值  
    LCD_ShowString(30,190,200,16,16,"Send Data:"); //提示发送的数据  
    LCD_ShowString(30,250,200,16,16,"Receive Data:");//提示接收到的数据  
    while(1)  
    {  
        key=KEY_Scan(0);  
        if(key==KEY0_PRES)//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(30+i*32,210,canbuf[i],3,16,0X80); //显示数据  
                else LCD_ShowxNum(30+(i-4)*32,230,canbuf[i],3,16,0X80); //显示数据  
            }  
            res=CAN1_Send_Msg(canbuf,8);//发送 8 个字节  
            if(res)LCD_ShowString(30+80,190,200,16,16,"Failed"); //提示发送失败  
            else LCD_ShowString(30+80,190,200,16,16,"OK "); //提示发送成功  
        }else if(key==WKUP_PRES)//WK_UP 按下, 改变 CAN 的工作模式  
        {  
            mode=!mode;  
            CAN1_Mode_Init(CAN_SJW_1tq,CAN_BS2_6tq,CAN_BS1_7tq,6,mode);  
                //CAN 普通模式初始化,普通模式,波特率 500Kbps  
            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(30,130,200,16,16,"Normal Mode ");
}else //回环模式,一个开发板就可以测试了.
{
    LCD_ShowString(30,130,200,16,16,"LoopBack Mode");
}
POINT_COLOR=BLUE;//设置字体为蓝色
}
key=CAN1_Receive_Msg(canbuf);
if(key)//接收到有数据
{
    LCD_Fill(30,270,160,310,WHITE);//清除之前的显示
    for(i=0;i<key;i++)
    {
        if(i<4)LCD_ShowxNum(30+i*32,270,canbuf[i],3,16,0X80); //显示数据
        else LCD_ShowxNum(30+(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(30+48,170,cnt,3,16,0X80); //显示数据
    }
}
}
此部分代码，我们主要关注下 CAN1_Mode_Init 初始化代码：
```

```
CAN1_Mode_Init(CAN_SJW_1tq,CAN_BS2_6tq,CAN_BS1_7tq,6,mode);
```

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 500Kbps。mode 参数用于设置 CAN 的工作模式（普通模式/环回模式），通过 KEY_UP 按键，可以随时切换模式。cnt 是一个累加数，一旦 KEY0 按下，就以这个数位基准连续发送 8 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。

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 探索者 STM32F4 开发板上，得到如图 32.4.1 所示：

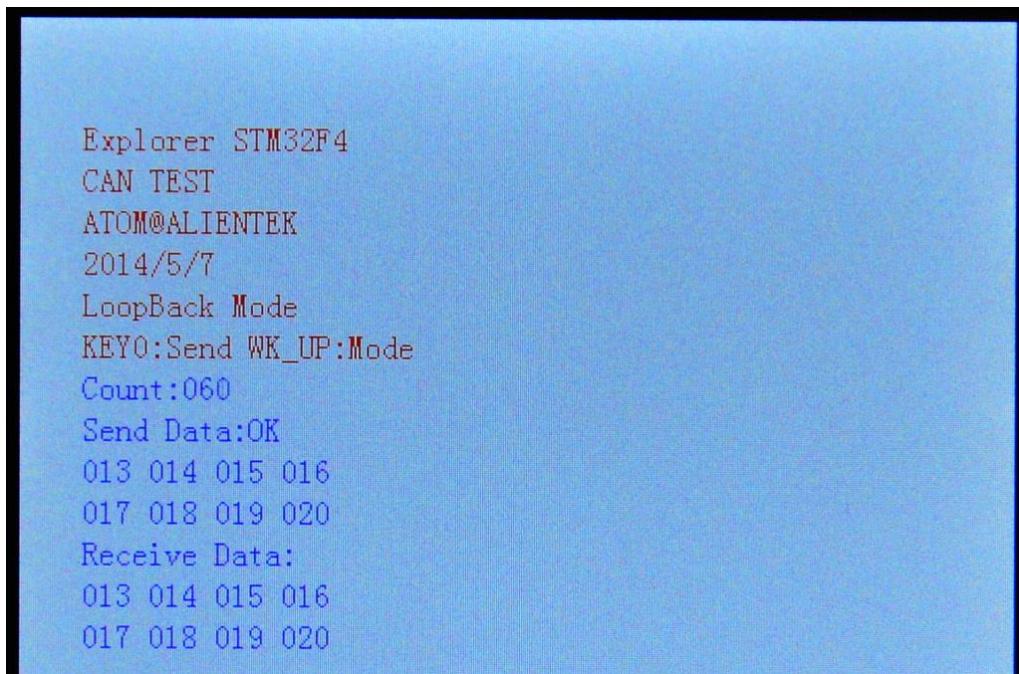


图 32.4.1 程序运行效果图

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），如果我们选择普通模式（通过 KEY_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 32.4.2 和图 32.4.3 所示：

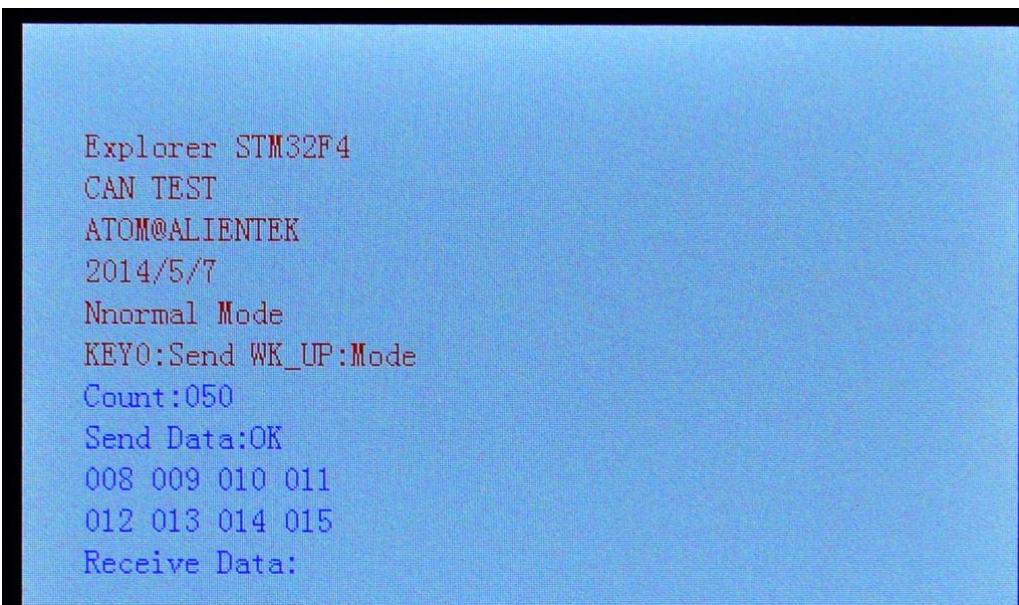


图 32.4.2 CAN 普通模式发送数据

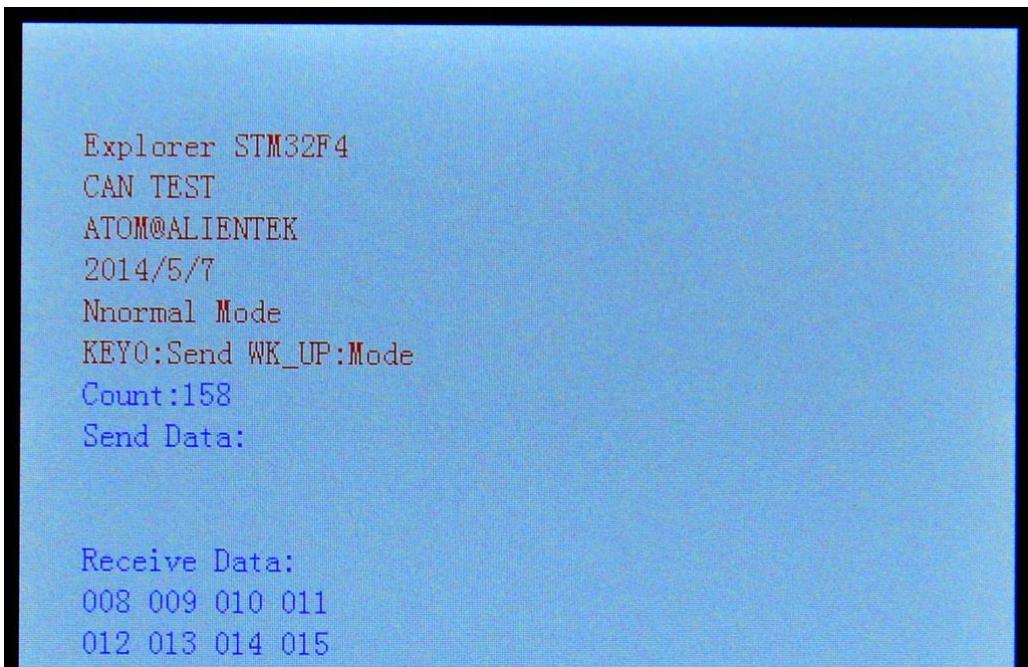


图 32.4.3 CAN 普通模式接收数据

图 32.4.2 来自开发板 A，发送了 8 个数据，图 32.4.3 来自开发板 B，收到了来自开发板 A 的 8 个数据。

第三十三章 触摸屏实验

本章，我们将介绍如何使用 STM32F4 来驱动触摸屏，ALIENTEK 探索者 STM32F4 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（包括电阻触摸与电容触摸），实现触摸屏驱动，最终实现一个手写板的功能。本章分为如下几个部分：

- 33.1 电阻与电容触摸屏简介
- 33.2 硬件设计
- 33.3 软件设计
- 33.4 下载验证

33.1 触摸屏简介

目前最常用的触摸屏有两种：电阻式触摸屏与电容式触摸屏。下面，我们来分别介绍。

33.1.1 电阻式触摸屏

在 Iphone 面世之前，几乎清一色的都是使用电阻式触摸屏，电阻式触摸屏利用压力感应进行触点检测控制，需要直接应力接触，通过检测电阻来定位触摸位置。

ALIENTEK 2.4/2.8/3.5 寸 TFTLCD 模块自带的触摸屏都属于电阻式触摸屏，下面简单介绍一下电阻式触摸屏的原理。

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。

电阻触摸屏的优点：精度高、价格便宜、抗干扰能力强、稳定性好。

电阻触摸屏的缺点：容易被划伤、透光性不太好、不支持多点触摸。

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。

电阻式触摸屏就介绍到这里。

33.1.2 电容式触摸屏

现在几乎所有智能手机，包括平板电脑都是采用电容屏作为触摸屏，电容屏是利用人体感应进行触点检测控制，不需要直接接触或只需要轻微接触，通过检测感应电流来定位触摸坐标。

ALIENTEK 4.3/7 寸 TFTLCD 模块自带的触摸屏采用的是电容式触摸屏，下面简单介绍下电容式触摸屏的原理。

电容式触摸屏主要分为两种：

1、表面电容式电容触摸屏。

表面电容式触摸屏技术是利用 ITO(铟锡氧化物，是一种透明的导电材料)导电膜，通过电场感应方式感测屏幕表面的触摸行为进行。但是表面电容式触摸屏有一些局限性，它只能识别一个手指或者一次触摸。

2、投射式电容触摸屏。

投射电容式触摸屏是传感器利用触摸屏电极发射出静电场线。一般用于投射电容传感技术的电容类型有两种：自我电容和交互电容。

自我电容又称绝对电容，是最广为采用的一种方法，自我电容通常是指扫描电极与地构成的电容。在玻璃表面有用 ITO 制成的横向与纵向的扫描电极，这些电极和地之间就构成一个电容的两极。当用手或触摸笔触摸的时候就会并联一个电容到电路中去，从而使在该条扫描线上的总体的电容量有所改变。在扫描的时候，控制 IC 依次扫描纵向和横向电极，并根据扫描前后的电容变化来确定触摸点坐标位置。笔记本电脑触摸输入板就是采用的这种方式，笔记本电脑的输入板采用 X*Y 的传感电极阵列形成一个传感格子，当手指靠近触摸输入板时，在手指和传感电极之间产生一个小量电荷。采用特定的运算法则处理来自行、列传感器的信号来确定手指的位置。

交互电容又叫做跨越电容，它是在玻璃表面的横向和纵向的 ITO 电极的交叉处形成电容。交互电容的扫描方式就是扫描每个交叉处的电容变化，来判定触摸点的位置。当触摸的时候就会影响到相邻电极的耦合，从而改变交叉处的电容量，交互电容的扫面方法可以侦测到每个交叉点的电容值和触摸后电容变化，因而它需要的扫描时间与自我电容的扫描方式相比要长一些，需要扫描检测 X*Y 根电极。目前智能手机/平板电脑等的触摸屏，都是采用交互电容技术。

ALIENTEK 所选择的电容触摸屏，也是采用的是投射式电容屏（交互电容类型），所以下面仅以投射式电容屏作为介绍。

透射式电容触摸屏采用纵横两列电极组成感应矩阵，来感应触摸。以两个交叉的电极矩阵，即： X 轴电极和 Y 轴电极，来检测每一格感应单元的电容变化，如图 33.1.2.1 所示：

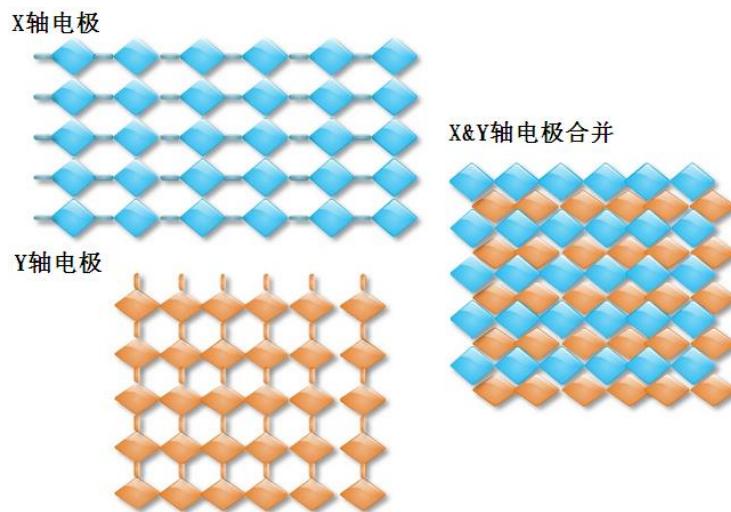


图 33.1.2.1 投射式电容屏电极矩阵示意图

示意图中的电极，实际是透明的，这里是为了方便大家理解。图中，X、Y 轴的透明电极电容屏的精度、分辨率与 X、Y 轴的通道数有关，通道数越多，精度越高。以上就是电容触摸屏的基本原理，接下来看看电容触摸屏的优缺点：

电容触摸屏的优点：手感好、无需校准、支持多点触摸、透光性好。

电容触摸屏的缺点：成本高、精度不高、抗干扰能力差。

这里特别提醒大家电容触摸屏对工作环境的要求是比较高的，在潮湿、多尘、高低温环境下面，都是不适合使用电容屏的。

电容触摸屏一般都需要一个驱动 IC 来检测电容触摸，且一般是通过 IIC 接口输出触摸数据的。ALIENTEK 7' TFTLCD 模块的电容触摸屏，采用的是 15*10 的驱动结构（10 个感应通道，15 个驱动通道），采用的是 GT811 做为驱动 IC。ALIENTEK 4.3' TFTLCD 模块有两种成触摸屏：1，使用 OTT2001A 作为驱动 IC，采用 13*8 的驱动结构（8 个感应通道，13 个驱动通道）；2，使用 GT9147 作为驱动 IC，采用 17*10 的驱动结构（10 个感应通道，17 个驱动通道）。

这两个模块都只支持最多 5 点触摸，在本例程，仅支持 ALIENTEK 4.3 寸 TFTLCD 电容触摸屏模块，所以这里介绍仅 OTT2001A 和 GT9147，GT811 的驱动方法同这两款 IC 是类似的，大家可以参考着学习即可。

OTT2001A 是台湾旭曜科技生产的一颗电容触摸屏驱动 IC，最多支持 208 个通道。支持 SPI/IIC 接口，在 ALIENTEK 4.3' TFTLCD 电容触摸屏上，OTT2001A 只用了 104 个通道，采用 IIC 接口。IIC 接口模式下，该驱动 IC 与 STM32F4 的连接仅需要 4 根线：SDA、SCL、RST 和 INT，SDA 和 SCL 是 IIC 通信用的，RST 是复位脚（低电平有效），INT 是中断输出信号，关于 IIC 我们就不详细介绍了，请参考第二十九章。

OTT2001A 的器件地址为 0X59（不含最低位，换算成读写命令则是读：0XB3，写：0XB2），接下来，介绍一下 OTT2001A 的几个重要的寄存器。

1，手势 ID 寄存器

手势 ID 寄存器（00H）用于告诉 MCU，哪些点有效，哪些点无效，从而读取对应的数据，该寄存器各位描述如表 33.1.2.1 所示：

手势 ID 寄存器（00H）				
位	BIT8	BIT6	BIT5	BIT4
说明	保留	保留	保留	0, (X1, Y1) 无效 1, (X1, Y1) 有效

位	BIT3	BIT2	BIT1	BIT0
说 明	0, (X4, Y4) 无效 1, (X4, Y4) 有效	0, (X3, Y3) 无效 1, (X3, Y3) 有效	0, (X2, Y2) 无效 1, (X2, Y2) 有效	0, (X1, Y1) 无效 1, (X1, Y1) 有效

表 33.1.2.1 手势 ID 寄存器

OTT2001A 支持最多 5 点触摸，所以表中只有 5 个位用来表示对应点坐标是否有效，其余位为保留位（读为 0），通过读取该寄存器，我们可以知道哪些点有数据，哪些点无数据，如果读到的全是 0，则说明没有任何触摸。

2. 传感器控制寄存器 (ODH)

传感器控制寄存器 (ODH)，该寄存器也是 8 位，仅最高位有效，其他位都是保留，当最高位为 1 的时候，打开传感器(开始检测)，当最高位设置为 0 的时候，关闭传感器(停止检测)。

3. 坐标数据寄存器 (共 20 个)

坐标数据寄存器总共有 20 个，每个坐标占用 4 个寄存器，坐标寄存器与坐标的对应关系如表 33.1.2.2 所示：

寄存器编号	01H	02H	03H	04H
坐标 1	X1[15:8]	X1[7:0]	Y1[15:8]	Y1[7:0]
寄存器编号	05H	06H	07H	08H
坐标 2	X2[15:8]	X2[7:0]	Y2[15:8]	Y2[7:0]
寄存器编号	10H	11H	12H	13H
坐标 3	X3[15:8]	X3[7:0]	Y3[15:8]	Y3[7:0]
寄存器编号	14H	15H	16H	17H
坐标 4	X4[15:8]	X4[7:0]	Y4[15:8]	Y4[7:0]
寄存器编号	18H	19H	1AH	1BH
坐标 5	X5[15:8]	X5[7:0]	Y5[15:8]	Y5[7:0]

表 33.1.2.2 坐标寄存器与坐标对应表

从表中可以看出，每个坐标的值，可以通过 4 个寄存器读出，比如读取坐标 1 (X1, Y1)，我们则可以读取 01H~04H，就可以知道当前坐标 1 的具体数值了，这里我们也可以只发送寄存器 01，然后连续读取 4 个字节，也可以正常读取坐标 1，寄存器地址会自动增加，从而提高读取速度。

OTT2001A 相关寄存器的介绍就介绍到这里，更详细的资料，请参考：OTT2001A IIC 协议指导.pdf 这个文档。OTT2001A 只需要经过简单的初始化就可以正常使用了，初始化流程：复位→延时 100ms→释放复位→设置传感器控制寄存器的最高位位 1，开启传感器检查。就可以正常使用了。

另外，OTT2001A 有两个地方需要特别注意一下：

- 1, OTT2001A 的寄存器是 8 位的，但是发送的时候要发送 16 位（高八位有效），才可以正常使用。
- 2, OTT2001A 的输出坐标，默认是以：X 坐标最大值是 2700，Y 坐标最大值是 1500 的分辨率输出的，也就是输出范围为：X: 0~2700, Y: 0~1500；MCU 在读取到坐标后，必须根据 LCD 分辨率做一个换算，才能得到真实的 LCD 坐标。

下面我们简单介绍下 GT9147，该芯片是深圳汇顶科技研发的一颗电容触摸屏驱动 IC，支持 100Hz 触点扫描频率，支持 5 点触摸，支持 18*10 个检测通道，适合小于 4.5 寸的电容触摸屏使用。

和 OTT2001A 一样，GT9147 与 MCU 连接也是通过 4 根线：SDA、SCL、RST 和 INT。不过，GT9147 的 IIC 地址，可以是 0X14 或者 0X5D，当复位结束后的 5ms 内，如果 INT 是高电

平，则使用 0X14 作为地址，否则使用 0X5D 作为地址，具体的设置过程，请看：GT9147 数据手册.pdf 这个文档。本章我们使用 0X14 作为器件地址（不含最低位，换算成读写命令则是读：0X29，写：0X28），接下来，介绍一下 GT9147 的几个重要的寄存器。

1，控制命令寄存器（0X8040）

该寄存器可以写入不同值，实现不同的控制，我们一般使用 0 和 2 这两个值，写入 2，即可软复位 GT9147，在硬复位之后，一般要往该寄存器写 2，实行软复位。然后，写入 0，即可正常读取坐标数据（并且会结束软复位）。

2，配置寄存器组（0X8047~0X8100）

这里共 186 个寄存器，用于配置 GT9147 的各个参数，这些配置一般由厂家提供给我们（一个数组），所以我们只需要将厂家给我们的配置，写入到这些寄存器里面，即可完成 GT9147 的配置。由于 GT9147 可以保存配置信息（可写入内部 FLASH，从而不需要每次上电都更新配置），我们有几点注意的地方提醒大家：1，0X8047 寄存器用于指示配置文件版本号，程序写入的版本号，必须大于等于 GT9147 本地保存的版本号，才可以更新配置。2，0X80FF 寄存器用于存储校验和，使得 0X8047~0X80FF 之间所有数据之和为 0。3，0X8100 用于控制是否将配置保存在本地，写 0，则不保存配置，写 1 则保存配置。

3，产品 ID 寄存器（0X8140~0X8143）

这里总共由 4 个寄存器组成，用于保存产品 ID，对于 GT9147，这 4 个寄存器读出来就是：9，1，4，7 四个字符（ASCII 码格式）。因此，我们可以通过这 4 个寄存器的值，来判断驱动 IC 的型号，从而判断是 OTT2001A 还是 GT9147，以便执行不同的初始化。

4，状态寄存器（0X814E）

该寄存器各位描述如表 33.1.2.3 所示：

寄存器	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0X814E	buffer 状态	大点	接近 有效	按键	有效触点个数			

表 33.1.2.3 状态寄存器各位描述

这里，我们仅关心最高位和最低 4 位，最高位用于表示 buffer 状态，如果有数据（坐标/按键），buffer 就会是 1，最低 4 位用于表示有效触点的个数，范围是：0~5，0，表示没有触摸，5 表示有 5 点触摸。这和前面 OTT2001A 的表示方法稍微有点区别，OTT2001A 是每个位表示一个触点，这里是有效触点值是多少。最后，该寄存器在每次读取后，如果 bit7 有效，则必须写 0，清除这个位，否则不会输出下一次数据！！这个要特别注意！！！

5，坐标数据寄存器（共 30 个）

这里共分成 5 组（5 个点），每组 6 个寄存器存储数据，以触点 1 的坐标数据寄存器组为例，如表 33.1.2.4 所示：

寄存器	bit7~0	寄存器	bit7~0
0X8150	触点 1 x 坐标低 8 位	0X8151	触点 1 x 坐标低高位
0X8152	触点 1 y 坐标低 8 位	0X8153	触点 1 y 坐标低高位
0X8154	触点 1 触摸尺寸低 8 位	0X8155	触点 1 触摸尺寸高 8 位

表 33.1.2.4 触点 1 坐标寄存器组描述

我们一般只用到触点的 x, y 坐标，所以只需要读取 0X8150~0X8153 的数据，组合即可得到触点坐标。其他 4 组分别是：0X8158、0X8160、0X8168 和 0X8170 等开头的 16 个寄存器组成，分别针对触点 2~4 的坐标。同样 GT9147 也支持寄存器地址自增，我们只需要发送寄存器组的首地址，然后连续读取即可，GT9147 会自动地址自增，从而提高读取速度。

GT9147 相关寄存器的介绍就介绍到这里，更详细的资料，请参考：GT9147 编程指南.pdf 这个文档。

GT9147 只需要经过简单的初始化就可以正常使用了，初始化流程：硬复位 → 延时 10ms → 结束硬复位 → 设置 IIC 地址 → 延时 100ms → 软复位 → 更新配置（需要时）→ 结束软复位。此时 GT9147 即可正常使用了。

然后，我们不停的查询 0X814E 寄存器，判断是否有有效触点，如果有，则读取坐标数据寄存器，得到触点坐标，特别注意，如果 0X814E 读到的值最高位为 1，就必须对该位写 0，否则无法读到下一次坐标数据。

电容式触摸屏部分，就介绍到这里。

33.2 硬件设计

本章实验功能简介：开机的时候先初始化 LCD，读取 LCD ID，随后，根据 LCD ID 判断是电阻触摸屏还是电容触摸屏，如果是电阻触摸屏，则先读取 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，校准过后再进入电阻触摸屏测试程序，如果已经校准了，就直接进入电阻触摸屏测试程序。

如果是电容触摸屏，则先读取芯片 ID，判断是不是 GT9147，如果是则执行 GT9147 初始化代码，如果不是，则执行 OTT2001A 的初始化代码，初始化电容触摸屏，随后进入电容触摸屏测试程序（电容触摸屏无需校准！！）。

电阻触摸屏测试程序和电容触摸屏测试程序基本一样，只是电容触摸屏支持最多 5 点同时触摸，电阻触摸屏只支持一点触摸，其他一模一样。测试界面的右上角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。使用电阻触摸屏的时候，可以通过按 KEY0 来实现强制触摸屏校准，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（带电阻/电容式触摸屏）
- 4) 24C02

所有这些资源与 STM32F4 的连接图，在前面都已经介绍了，这里我们只针对 TFTLCD 模块与 STM32F4 的连接端口再说明一下，TFTLCD 模块的触摸屏（电阻触摸屏）总共有 5 跟线与 STM32F4 连接，连接电路图如图 33.2.1 所示：

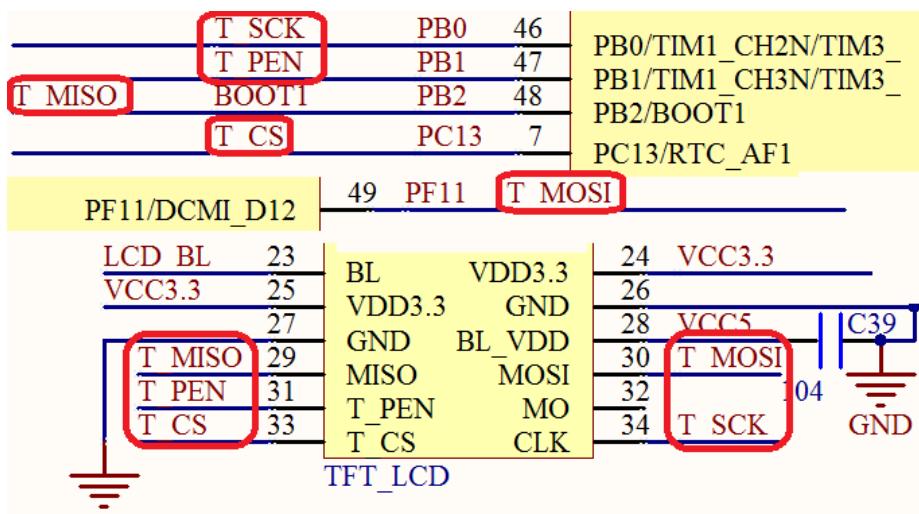


图 33.2.1 触摸屏与 STM32F4 的连接图

从图中可以看出, T_MOSI、T_MISO、T_SCK、T_CS 和 T_PEN 分别连接在 STM32F4 的: PF11、PB2、PB0、PC13 和 PB1 上。

如果是电容式触摸屏, 我们的接口和电阻式触摸屏一样 (上图右侧接口), 只是没有用到五根线了, 而是四根线, 分别是: T_PEN(CT_INT)、T_CS(CT_RST)、T_CLK(CT_SCL) 和 T_MOSI(CT_SDA)。其中: CT_INT、CT_RST、CT_SCL 和 CT_SDA 分别是 OTT2001A/GT9147 的: 中断输出信号、复位信号, IIC 的 SCL 和 SDA 信号。这里, 我们用查询的方式读取 OTT2001A/GT9147 的数据, 对于 OTT2001A 没有用到中断信号 (CT_INT), 所以同 STM32F4 的连接, 只需要 3 根线即可, 不过 GT9147 还需要用到 CT_INT 做 IIC 地址设定, 所以需要 4 根线连接。

33.3 软件设计

打开本章实验工程目录可以看到, 我们在 HARDWARE 文件夹下新建了一个 TOUCH 文件夹, 然后新建了 touch.c、touch.h、ctiic.c、ctiic.h、ott2001a.c、ott2001a.h、gt9147.c 和 gt9147.h 等八个文件用来存放触摸屏相关的代码。同时引入这些源文件到工程 HARDWARE 分组之下, 并将 TOUCH 文件夹加入头文件包含路径。其中, touch.c 和 touch.h 是电阻触摸屏部分的代码, 顺带兼电容触摸屏的管理控制, 其他则是电容触摸屏部分的代码。

打开 touch.c 文件, 里面主要是与触摸屏相关的代码 (主要是电阻触摸屏的代码), 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, 该函数专门用于从电阻式触摸屏控制 IC 读取坐标的值 (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);
    //前后两次采样在+-50 内
    if(((x2<=x1&&x1<x2+ERR_RANGE)||((x1<=x2&&x2<x1+ERR_RANGE))
    &&((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; u32 tem1,tem2;
    u16 d1,d2; u16 outtime=0;
    double fac;
    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 {CMD_RDX=0XD0;CMD_RDY=0X90;}
//X,Y 方向与屏幕相同
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。绝对坐标系统的优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：

$$\begin{aligned} \text{LCDx} &= \text{xfac} * \text{Px} + \text{xoff}; \\ \text{LCDy} &= \text{yfac} * \text{Py} + \text{yoff}; \end{aligned}$$

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*240、480*320 和 800*480 的屏都可以兼容)。

接下来看看触摸屏初始化函数：TP_Init，该函数根据 LCD 的 ID (即 lcddev.id) 判别是电阻屏还是电容屏，执行不同的初始化，该函数代码如下：

```
//触摸屏初始化
//返回值:0,没有进行校准 1,进行过校准
u8 TP_Init(void)
{
    if(lcddev.id==0X5510)      //电容触摸屏
    {
        if(GT9147_Init()==0)  //是 GT9147?
        {
            tp_dev.scan=GT9147_Scan; //扫描函数指向 GT9147 触摸屏扫描
        }
        else
        {
            OTT2001A_Init();
            tp_dev.scan=OTT2001A_Scan;//扫描函数指向 OTT2001A 触摸屏扫描
        }
        tp_dev.touchtype|=0X80;    //电容屏
        tp_dev.touchtype|=lcddev.dir&0X01;//横屏还是竖屏
        return 0;
    }
    else
    {
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC|
                               RCC_AHB1Periph_GPIOF, ENABLE); //使能 GPIOB,C,F 时钟
        //GPIOB1,2 初始化设置
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;//PB1/2 设置为上拉输入
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//输入模式
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
        GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;//PB0 设置为推挽输出
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
        GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;//PC13 设置为推挽输出
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
        GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;//PF11 设置推挽输出
```

```

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出模式
GPIO_Init(GPIOF, &GPIO_InitStructure);//初始化
TP_Read_XY(&tp_dev.x[0],&tp_dev.y[0]);//第一次读取初始化
AT24CXX_Init(); //初始化 24CXX
if(TP_Get_Adjdata())return 0;//已经校准
else //未校准?
{
    LCD_Clear(WHITE); //清屏
    TP_Adjust(); //屏幕校准
    TP_Save_Adjdata();
}
TP_Get_Adjdata();
}
return 1;
}

```

该函数比较简单，重点说一下：tp_dev.scan，这个结构体函数指针，默认是指向 TP_Scan 的，如果是电阻屏则用默认的即可，如果是电容屏，则指向新的扫描函数 GT9147_Scan 或 OTT2001A_Scan（根据芯片 ID 判断到底指向那个），执行电容触摸屏的扫描函数，这两个函数在后续会介绍。

其他的函数我们这里就不多介绍了，接下来打开 touch.h 文件，代码如下：

```

#define TP_PRES_DOWN 0x80 //触屏被按下
#define TP_CATH_PRES 0x40 //有按键按下了
#define CT_MAX_TOUCH 5 //电容屏支持的点数,固定为 5 点
//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 x[CT_MAX_TOUCH]; //当前坐标
    u16 y[CT_MAX_TOUCH]; //电容屏有最多 5 组坐标,电阻屏则用 x[0],y[0]代表: 此次
                         //扫描时触屏的坐标,用 x[4],y[4]存储第一次按下时的坐标.
    u8 sta; //笔的状态
             //b7:按下 1/松开 0;
             //b6:0,没有按键按下;1,有按键按下.
             //b5:保留
             //b4~b0:电容触摸屏按下的点数(0,表示未按下,1 表示按下)
////////////////////////////触摸屏校准参数(电容屏不需要校准)///////////////////
    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.

```

```

//b0:0,竖屏(适合左右为 X 坐标,上下为 Y 坐标的 TP)
//    1,横屏(适合左右为 Y 坐标,上下为 X 坐标的 TP)
//b1~6:保留.
//b7:0,电阻屏
//    1,电容屏
    u8 toucheType;
}__m_tp_dev;
extern __m_tp_dev tp_dev;          //触屏控制器在 touch.c 里面定义
//电阻屏芯片连接引脚
#define PEN          PBIn(1)      //T_PEN
#define DOUT         PBIn(2)      //T_MISO
#define TDIN         PFout(11)    //T_MOSI
#define TCLK         PBout(0)     //T_SCK
#define TCS          PCout(13)    //T_CS
//电阻屏函数
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)
.....//(省略部分代码)
u8 TP_Scan(u8 tp);               //扫描
u8 TP_Init(void);                //初始化
#endif

```

上述代码，我们重点看看`_m_tp_dev` 结构体，改结构体用于管理和记录触摸屏（包括电阻触摸屏与电容触摸屏）相关信息。通过结构体，在使用的时候，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，这种设计简化了接口，且方便管理和维护，大家可以效仿一下。

`ctiic.c` 和 `ctiic.h` 是电容触摸屏的 IIC 接口部分代码，与第二十九章的 `myiic.c` 和 `myiic.h` 基本一样，这里就不单独介绍了。接下来看看文件 `ott2001a.c` 代码如下：

```

//向 OTT2001A 写入一次数据
//reg:起始寄存器地址
//buf:数据缓存区
//len:写数据长度
//返回值:0,成功;1,失败.
u8 OTT2001A_WR_Reg(u16 reg,u8 *buf,u8 len)
{
    u8 i; u8 ret=0;
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_WR);CT_IIC_Wait_Ack(); //发送写命令
    CT_IIC_Send_Byte(reg>>8); CT_IIC_Wait_Ack();      //发送高 8 位地址
    CT_IIC_Send_Byte(reg&0xFF); CT_IIC_Wait_Ack();      //发送低 8 位地址
    for(i=0;i<len;i++)
    {
        CT_IIC_Send_Byte(buf[i]); ret=CT_IIC_Wait_Ack(); //发数据
    }
}

```

```
        if(ret)break;
    }
    CT_IIC_Stop(); //产生一个停止条件
    return ret;
}

//从 OTT2001A 读出一次数据
//reg:起始寄存器地址
//buf:数据缓存区
//len:读数据长度
void OTT2001A_RD_Reg(u16 reg,u8 *buf,u8 len)
{
    u8 i;
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_WR); CT_IIC_Wait_Ack(); //发送写命令
    CT_IIC_Send_Byte(reg>>8); CT_IIC_Wait_Ack(); //发送高 8 位地址
    CT_IIC_Send_Byte(reg&0xFF); CT_IIC_Wait_Ack(); //发送低 8 位地址
    CT_IIC_Start();
    CT_IIC_Send_Byte(OTT_CMD_RD); CT_IIC_Wait_Ack(); //发送读命令
    for(i=0;i<len;i++) buf[i]=CT_IIC_Read_Byte(i==(len-1)?0:1); //发数据
    CT_IIC_Stop(); //产生一个停止条件
}
//传感器打开/关闭操作
//cmd:1,打开传感器;0,关闭传感器
void OTT2001A_SensorControl(u8 cmd)
{
    u8 regval=0X00;
    if(cmd)regval=0X80;
    OTT2001A_WR_Reg(OTT_CTRL_REG,&regval,1);
}

//初始化触摸屏
//返回值:0,初始化成功;1,初始化失败
u8 OTT2001A_Init(void)
{
    u8 regval=0;
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC,
                           ENABLE); //使能 GPIOB,C 时钟
    //GPIOB1 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1; //PB1 设置为上拉输入
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN; //输入模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
}
```

```
GPIO_Init(GPIOB, &GPIO_InitStructure); // 初始化  
  
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13; // PC13 设置为推挽输出  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; // 输出模式  
GPIO_Init(GPIOC, &GPIO_InitStructure); // 初始化  
  
CT_IIC_Init(); // 初始化电容屏的 I2C 总线  
OTT_RST=0; // 复位  
delay_ms(100);  
OTT_RST=1; // 释放复位  
delay_ms(100);  
OTT2001A_SensorControl(1); // 打开传感器  
OTT2001A_RD_Reg(OTT_CTRL_REG, &regval, 1); // 读取传感器运行寄存器的值来判断  
 // I2C 通信是否正常  
printf("CTP ID:%x\r\n", regval);  
if (regval == 0x80) return 0;  
return 1;  
}  
const u16 OTT_TPX_TBL[5] = {OTT_TP1_REG, OTT_TP2_REG, OTT_TP3_REG, OTT_TP4  
 _REG, OTT_TP5_REG};  
// 扫描触摸屏(采用查询方式)  
// mode: 0, 正常扫描.  
// 返回值: 当前触屏状态.  
// 0, 触屏无触摸; 1, 触屏有触摸  
u8 OTT2001A_Scan(u8 mode)  
{  
    u8 buf[4], i=0, res=0;  
    static u8 t=0; // 控制查询间隔, 从而降低 CPU 占用率  
    t++;  
    if ((t%10)==0 || t<10) // 空闲时, 每进入 10 次, 才检测 1 次, 从而节省 CPU 使用率  
    {  
        OTT2001A_RD_Reg(OTT_GSTID_REG, &mode, 1); // 读取触摸点的状态  
        if (mode&0X1F)  
        {  
            tp_dev.sta=(mode&0X1F)|TP_PRES_DOWN|TP_CATH_PRES;  
            for (i=0; i<5; i++)  
            {  
                if (tp_dev.sta&(1<<i)) // 触摸有效?  
                {  
                    OTT2001A_RD_Reg(OTT_TPX_TBL[i], buf, 4); // 读取 XY 坐标值  
                    if (tp_dev.touchtype&0X01) // 横屏  
                    {  
                        tp_dev.y[i]=(((u16)buf[2]<<8)+buf[3])*OTT_SCAL_Y;
```

```

        tp_dev.x[i]=800-(((u16)buf[0]<<8)+buf[1])*OTT_SCAL_X;
    }else
    {
        tp_dev.x[i]=((u16)buf[2]<<8)+buf[3])*OTT_SCAL_Y;
        tp_dev.y[i]=((u16)buf[0]<<8)+buf[1])*OTT_SCAL_X;
    }
    //printf("x[%d]:%d,y[%d]:%d\r\n",i,tp_dev.x[i],i,tp_dev.y[i]);
}
res=1;
if(tp_dev.x[0]==0 && tp_dev.y[0]==0)mode=0; //数据全 0,则忽略此次数据
t=0;      //触发一次,则会最少连续监测 10 次,从而提高命中率
}
}
if((mode&0X1F)==0)//无触摸点按下
{
    if(tp_dev.sta&TP_PRES_DOWN) tp_dev.sta&=~(1<<7); //之前是按下, 标记松开
    else //之前就没有被按下
    {
        tp_dev.x[0]=0xffff;  tp_dev.y[0]=0xffff;
        tp_dev.sta&=0XE0; //清除点有效标记
    }
}
if(t>240)t=10;//重新从 10 开始计数
return res;
}

```

此部分总共 5 个函数，其中 OTT2001A_WR_Reg 和 OTT2001A_RD_Reg 分别用于读写 OTT2001A 芯片，这里特别注意寄存器地址是 16 位的，与 OTT2001A 手册介绍的是有出入的，必须 16 位才能正常操作。另外，重点介绍下 OTT2001A_Scan 函数，OTT2001A_Scan 函数用于扫描电容触摸屏是否有按键按下，由于我们不是用的中断方式来读取 OTT2001A 的数据的，而是采用查询的方式，所以这里使用了一个静态变量来提高效率，当无触摸的时候，尽量减少对 CPU 的占用，当有触摸的时候，又保证能迅速检测到。至于对 OTT2001A 数据的读取，则完全是在上面介绍的方法，先读取手势 ID 寄存器 (OTT_GSTID_REG)，判断是不是有有效数据，如果有，则读取，否则直接忽略，继续后面的处理。

其他的函数我们这里就不多介绍了，接下来看下 gt9147.c 里面的代码，这里我们仅介绍 GT9147_Init 和 GT9147_Scan 两个函数，代码如下：

```

//初始化 GT9147 触摸屏
//返回值:0,初始化成功;1,初始化失败
u8 GT9147_Init(void)
{
    u8 temp[5];
    GPIO_InitTypeDef  GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC,

```

```
ENABLE); //使能 GPIOB,C 时钟
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1; //PB1 设置为上拉输入
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN; //输入模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13; //PC13 设置为推挽输出
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //输出模式
GPIO_Init(GPIOC, &GPIO_InitStructure); //初始化
CT_IIC_Init(); //初始化电容屏的 I2C 总线
GT_RST=0; delay_ms(10); //复位
GT_RST=1; delay_ms(10); //释放复位
GPIO_Set(GPIOB, PIN1, GPIO_MODE_IN, 0, 0, GPIO_PUPD_NONE); //PB1 浮空输入
delay_ms(100);
GT9147_RD_Reg(GT_PID_REG, temp, 4); //读取产品 ID
temp[4]=0;
printf("CTP ID:%s\r\n", temp); //打印 ID
if(strcmp((char*)temp, "9147") == 0) //ID==9147
{
    temp[0]=0X02;
    GT9147_WR_Reg(GT_CTRL_REG, temp, 1); //软复位 GT9147
    GT9147_RD_Reg(GT_CFGS_REG, temp, 1); //读取 GT_CFGS_REG 寄存器
    if(temp[0]<0X60) //默认版本比较低,需要更新 flash 配置
    {
        printf("Default Ver:%d\r\n", temp[0]);
        GT9147_Send_Cfg(1); //更新并保存配置
    }
    delay_ms(10);
    temp[0]=0X00;
    GT9147_WR_Reg(GT_CTRL_REG, temp, 1); //结束复位
    return 0;
}
return 1;
}
```

```
const
```

u16

```
GT9147_TPX_TBL[5]={GT_TP1_REG, GT_TP2_REG, GT_TP3_REG, GT_TP4_REG,
GT_TP5_REG};
```

//扫描触摸屏(采用查询方式)

//mode:0,正常扫描.

//返回值:当前触屏状态.

//0,触屏无触摸;1,触屏有触摸

```
u8 GT9147_Scan(u8 mode)
{
    u8 buf[4]; u8 i=0; u8 res=0; u8 temp;
    static u8 t=0;//控制查询间隔,从而降低 CPU 占用率
    t++;
    if((t%10)==0||t<10)//空闲时,每进入 10 次,函数才检测 1 次,从而节省 CPU 使用率
    {
        GT9147_RD_Reg(GT_GSTID_REG,&mode,1);//读取触摸点的状态
        if((mode&0XF)&&((mode&0XF)<6))
        {
            temp=0xFF<<(mode&0XF);//将点的个数转换为 1 的位数,匹配 tp_dev.sta 定义
            tp_dev.sta=(~temp)|TP_PRES_DOWN|TP_CATH_PRES;
            for(i=0;i<5;i++)
            {
                if(tp_dev.sta&(1<<i)) //触摸有效?
                {
                    GT9147_RD_Reg(GT9147_TPX_TBL[i],buf,4); //读取 XY 坐标值
                    if(tp_dev.touchtype&0X01)//横屏
                    {
                        tp_dev.y[i]=((u16)buf[1]<<8)+buf[0];
                        tp_dev.x[i]=800-(((u16)buf[3]<<8)+buf[2]);
                    }
                    else
                    {
                        tp_dev.x[i]=((u16)buf[1]<<8)+buf[0];
                        tp_dev.y[i]=((u16)buf[3]<<8)+buf[2];
                    }
                }
            }
            res=1;
            if(tp_dev.x[0]==0 && tp_dev.y[0]==0)mode=0; //数据全 0,则忽略此次数据
            t=0; //触发一次,则会最少连续监测 10 次,从而提高命中率
        }
        if(mode&0X80&&((mode&0XF)<6)) //清标志?
        { temp=0; GT9147_WR_Reg(GT_GSTID_REG,&temp,1);}
    }
    if((mode&0X8F)==0X80)//无触摸点按下
    {
        if(tp_dev.sta&TP_PRES_DOWN) tp_dev.sta&=~(1<<7);//之前是按下, 标记松开
        else //之前就没有被按下
        {
            tp_dev.x[0]=0xffff;
            tp_dev.y[0]=0xffff;
            tp_dev.sta&=0XE0;//清除点有效标记
        }
    }
}
```

```
        }
    }
if(t>240)t=10;//重新从 10 开始计数
return res;
}
```

以上代码，GT9147_Init 用于初始化 GT9147，该函数通过读取 0X8140~0X8143 这 4 个寄存器，并判断是否是：“9147”，来确定是不是 GT9147 芯片，在读取到正确的 ID 后，软复位 GT9147，然后根据当前芯片版本号，确定是否需要更新配置，通过 GT9147_Send_Cfg 函数，发送配置信息（一个数组），配置完后，结束软复位，即完成 GT9147 初始化。GT9147_Scan 函数，用于读取触摸屏坐标数据，这个和前面的 OTT2001A_Scan 大同小异，大家看源码即可。

最后我们打开 main.c，修改部分代码，这里就不全部贴出来了，仅介绍三个重要的函数：

```
//5 个触控点的颜色(电容触摸屏用)
const u16 POINT_COLOR_TBL[5]={RED,GREEN,BLUE,BROWN,GRED};

//电阻触摸屏测试函数
void rtp_test(void)
{
    u8 key; u8 i=0;
    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[0]<lcddev.width&&tp_dev.y[0]<lcddev.height)
            {
                if(tp_dev.x[0]>(lcddev.width-24)&&tp_dev.y[0]<16)Load_Drow_Dialog();
                else TP_Draw_Big_Point(tp_dev.x[0],tp_dev.y[0],RED);/画图
            }
        }else delay_ms(10);      //没有按键按下的时候
        if(key==KEY0_PRES) //KEY0 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==0)LED0=!LED0;
    }
}

//电容触摸屏测试函数
void ctp_test(void)
{
```

```
u8 t=0; u8 i=0;
u16 lastpos[5][2];      //最后一次的数据
while(1)
{
    tp_dev.scan(0);
    for(t=0;t<5;t++)
    {
        if((tp_dev.sta)&(1<<t))
        {
            if(tp_dev.x[t]<lcddev.width&&tp_dev.y[t]<lcddev.height)
            {
                if(lastpos[t][0]==0xFFFF)
                {
                    lastpos[t][0] = tp_dev.x[t];
                    lastpos[t][1] = tp_dev.y[t];
                }
                lcd_draw_bline(lastpos[t][0],lastpos[t][1],tp_dev.x[t],tp_dev.y[t],2,
                               POINT_COLOR_TBL[t]);//画线
                lastpos[t][0]=tp_dev.x[t];
                lastpos[t][1]=tp_dev.y[t];
                if(tp_dev.x[t]>(lcddev.width-24)&&tp_dev.y[t]<20)
                {
                    Load_Drow_Dialog();//清除
                }
            }
            }else lastpos[t][0]=0xFFFF;
        }
        delay_ms(5);i++;
        if(i%20==0)LED0=!LED0;
    }
}
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    tp_dev.init(); //触摸屏初始化
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"TOUCH TEST");
```

```
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/7");
if(tp_dev.touchtype!=0XFF)LCD_ShowString(30,130,200,16,16,"Press KEY0 to Adjust");
                                         //电阻屏才显示
delay_ms(1500);
Load_Drow_Dialog();

if(tp_dev.touchtype&0X80)ctp_test();//电容屏测试
else rtp_test();                  //电阻屏测试
}
```

下面分别介绍一下这三个函数。

`rtp_test`, 该函数用于电阻触摸屏的测试, 该函数代码比较简单, 就是扫描按键和触摸屏, 如果触摸屏有按下, 则在触摸屏上面划线, 如果按中“RST”区域, 则执行清屏。如果按键 KEY0 按下, 则执行触摸屏校准。

`ctp_test`, 该函数用于电容触摸屏的测试, 由于我们采用 `tp_dev.sta` 来标记当前按下的触摸屏点数, 所以判断是否有电容触摸屏按下, 也就是判断 `tp_dev.sta` 的最低 5 位, 如果有数据, 则划线, 如果没数据则忽略, 且 5 个点划线的颜色各不一样, 方便区分。另外, 电容触摸屏不需要校准, 所以没有校准程序。

`main` 函数, 则比较简单, 初始化相关外设, 然后根据触摸屏类型, 去选择执行 `ctp_test` 还是 `rtp_test`。

软件部分就介绍到这里, 接下来看看下载验证。

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 探索者 STM32F4 开发板上, 电阻触摸屏得到如图 33.4.1 所示界面 (左侧画图界面, 右侧是校准界面):



图 33.4.1 电阻触摸屏测试程序运行效果

左侧的图片，表示已经校准过了，并且可以在屏幕触摸画图了。右侧的图片则是校准界面程序界面，用于校准触摸屏用（可以按 KEY0 进入校准）。

如果是电容触摸屏，测试界面如图 33.4.2 所示：

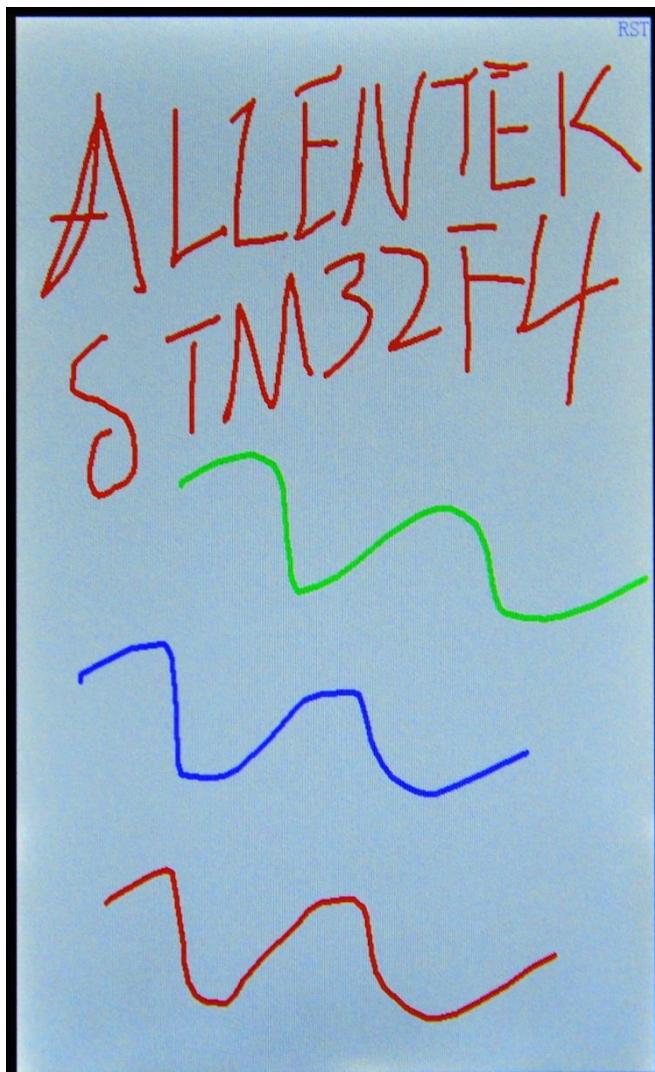


图 33.4.2 电容触摸屏测试界面

左侧是单点触摸效果图，右侧是多点触摸（图为 3 点，最大支持 5 点）效果图。

第三十四章 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。ALIENTEK 探索者 STM32F4 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 STM32F4 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：

- 34.1 红外遥控简介
- 34.2 硬件设计
- 34.3 软件设计
- 34.4 下载验证

34.1 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 探索者 STM32F4 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：同步码头、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“▽”按下时，从红外接收头端收到的波形如图 34.1.1 所示：



图 34.1.1 按键“▽”所对应的红外波形

从图 34.1.1 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。

34.2 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
- 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 34.2.1 所示：

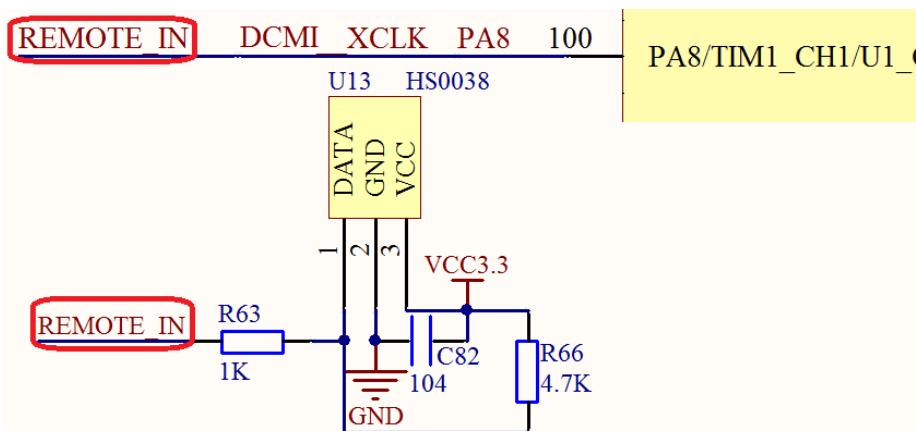


图 34.2.1 红外遥控接收头与 STM32 的连接电路图

红外遥控接收头连接在 STM32 的 PA8 (TIM1_CH1) 上。硬件上不需要变动，只要程序将 TIM1_CH1 设计为输入捕获，然后将收到的脉冲信号解码就可以了。不过需要注意：REMOTE_IN 和 DCMI_XCLK 共用了 PA8，所以他们不可以同时使用。

开发板配套的红外遥控器外观如图 34.2.2 所示：



图 34.2.2 红外遥控器

34.3 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，所以还用到定时器相关的库函数源文件 stm32f4xx_tim.c 和头文件 stm32f4xx_tim.h。

打开 remote.c 文件，代码如下：

```
//红外遥控初始化
//设置 IO 以及 TIM2_CH1 的输入捕获
void Remote_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_ICInitTypeDef TIM1_ICInitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE); //TIM1 时钟使能

    //GPIOA8 复用功能,上拉
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
    GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化

    GPIO_PinAFConfig(GPIOA, GPIO_PinSource8, GPIO_AF_TIM1); //GPIOA8 复用为 TIM1

    TIM_TimeBaseStructure.TIM_Prescaler = 167; //预分频器,1M 的计数频率,1us 加 1.
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式
```

```

TIM_TimeBaseStructure.TIM_Period=10000; //设定计数器自动重装值 最大 10ms 溢出
TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
TIM_TimeBaseInit(TIM1,&TIM_TimeBaseStructure);

//初始化 TIM1 输入捕获参数
TIM1_ICInitStructure.TIM_Channel = TIM_Channel_1; //选择输入端 IC1 映射到 TI1 上
TIM1_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; //上升沿捕获
TIM1_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //映射到 TI1 上
TIM1_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频,不分频
TIM1_ICInitStructure.TIM_ICFilter = 0x03;//IC1F=0003 8 个定时器时钟周期滤波
TIM_ICInit(TIM1, &TIM1_ICInitStructure); //初始化定时器 2 输入捕获通道

TIM_ITConfig(TIM1,TIM_IT_Update|TIM_IT_CC1,ENABLE); //允许更新中断捕获中断
TIM_Cmd(TIM1,ENABLE ); //使能定时器 1

NVIC_InitStructure.NVIC_IRQChannel = TIM1_CC_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=1; //抢占优先级 1
NVIC_InitStructure.NVIC_IRQChannelSubPriority =3; //响应优先级 3
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure); //初始化 NVIC 寄存器

NVIC_InitStructure.NVIC_IRQChannel = TIM1_UP_TIM10_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=1; //抢占优先级 3
NVIC_InitStructure.NVIC_IRQChannelSubPriority =2; //响应优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure); //初始化 NVIC 寄存器
}

//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 1 溢出中断
void TIM1_UP_TIM10_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM1,TIM_IT_Update)==SET) //溢出中断
    {
        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;//标记已经完成一次按键的键值信息采集
    if((RmtSta&0X0F)<14)RmtSta++;
    else
    {
        RmtSta&=~(1<<7);//清空引导标识
        RmtSta&=0XF0; //清空计数器
    }
}
TIM_ClearITPendingBit(TIM1,TIM_IT_Update); //清除中断标志位
}

//定时器 1 输入捕获中断服务程序
void TIM1_CC_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM1,TIM_IT_CC1)==SET) //处理捕获(CC1IE)中断
    {
        if(RDATA)//上升沿捕获
        {
            TIM_OC1PolarityConfig(TIM1,TIM_ICPolarity_Falling);//下降沿捕获
            TIM_SetCounter(TIM1,0); //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_GetCapture1(TIM1);//读取 CCR1 也可以清 CC1IF 标志位
            TIM_OC1PolarityConfig(TIM1,TIM_ICPolarity_Rising); //设置为上升沿捕获
            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;      //标记成功接收到到了引导码
        RmtCnt=0;           //清除按键次数计数器
    }
}
RmtSta&=~(1<<4);
}

}

TIM_ClearITPendingBit(TIM1,TIM_IT_CC1); //清除中断标志位
}

//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 4 个函数，首先是 Remote_Init 函数，该函数用于初始化 IO 口，并配置 TIM1_CH1 为输入捕获，并设置其相关参数。由于 TIM1 是高级定时器，它有 2 个中断服务函数：

TIM1_UP_TIM10_IRQHandler 用于处理 TIM1 的溢出事件，在本例程里面，该函数用来处理 TIM1 溢出，并标用于标记键值获取完成。每一次红外按键解码，都必须通过定时器溢出事件来标记完成。

TIM1_CC_IRQHandler 用于处理 TIM1 的输入捕获事件，在本例程里面，该函数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。

这两个函数用到几个全局变量，用于辅助解码，并存储解码结果。

这里简单介绍一下高电平捕获思路：首先输入捕获设置的是捕获上升沿，在上升沿捕获到以后，立即设置输入捕获模式为捕获下降沿（以便捕获本次高电平），然后，清零定时器的计数值，并标记捕获到上升沿。当下降沿到来时，再次进入捕获中断服务函数，立即更改输入捕获模式为捕获上升沿（以便捕获下一次高电平），然后处理此次捕获到的高电平。

最后是 Remote_Scan 函数，该函用来扫描解码结果，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。

接下来我们打开 remote.h 文件，可以看到下面一行代码：

```
#define REMOTE_ID 0
```

这里的 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。remote.h 其他是一些函数的声明，我们不做过多讲解，最后我们看看主函数代码如下：

```
int main(void)
{
    u8 key, t=0, *str=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init();
    Remote_Init(); //红外接收初始化
    POINT_COLOR=RED; //设置字体为红色
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"REMOTE TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/7");
    LCD_ShowString(30,130,200,16,16,"KEYVAL:");
    LCD_ShowString(30,150,200,16,16,"KEYCNT:");
    LCD_ShowString(30,170,200,16,16,"SYMBOL:");

    while(1)
    {
        key=Remote_Scan();
        if(key)
        {
            LCD_ShowNum(86,130,key,3,16); //显示键值
            LCD_ShowNum(86,150,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(86,170,116+8*8,170+16,WHITE); //清楚之前的显示
LCD_ShowString(86,170,200,16,16,str); //显示 SYMBOL
}else delay_ms(10);
t++;
if(t==20)
{
    t=0;
    LED0=!LED0;
}
}

main 函数代码比较简单，主要是通过 Remote_Scan 函数获得红外遥控输入的数据(键值)，然后显示在 LCD 上面。
```

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 34.4.1 所示的内容：

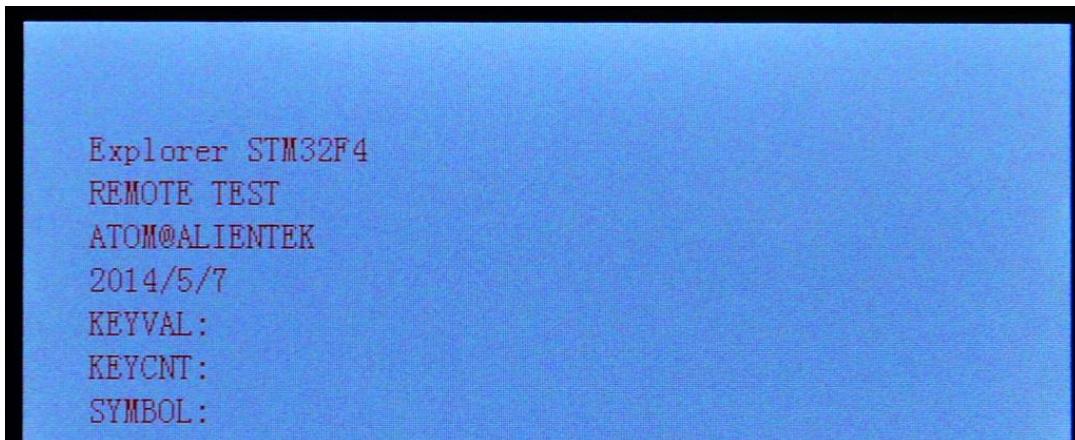


图 34.4.1 程序运行效果图

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 34.4.2 所示：

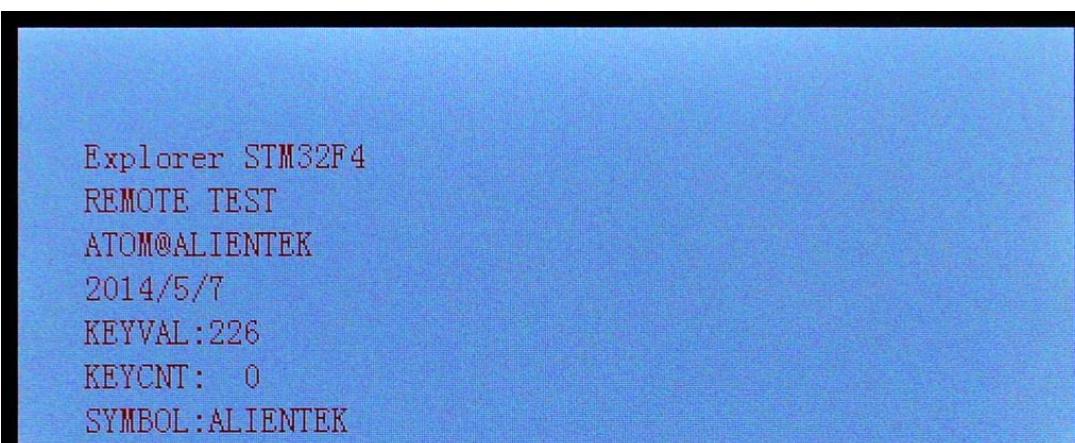


图 34.4.2 解码成功

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：

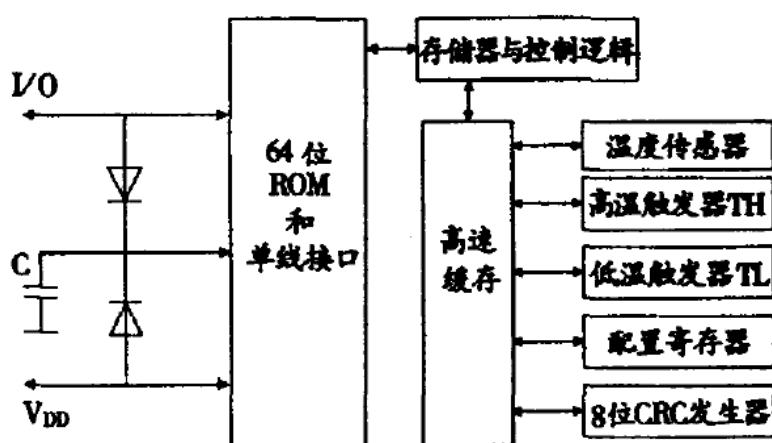


图 35.1.1 DS18B20 内部结构图

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 DS18B20。

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 us,，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 us，并进入接收模式(Rx)。接着 DS18B20 拉低总线 60~240 us,，以产生低电平应答脉冲，

若为低电平，再延时 480 us。

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 60us,，且在 2 次独立的写时序之间至少需要 1us 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 2us,，然后释放总线，延时 60us。写 0 时序：主机输出低电平，延时 60us,，然后释放总线，延时 2us.

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 (0XBE) →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 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，我们仅介绍开发板上 DS18B20 接口和 STM32 的连接电路，如图35.2.1所示：

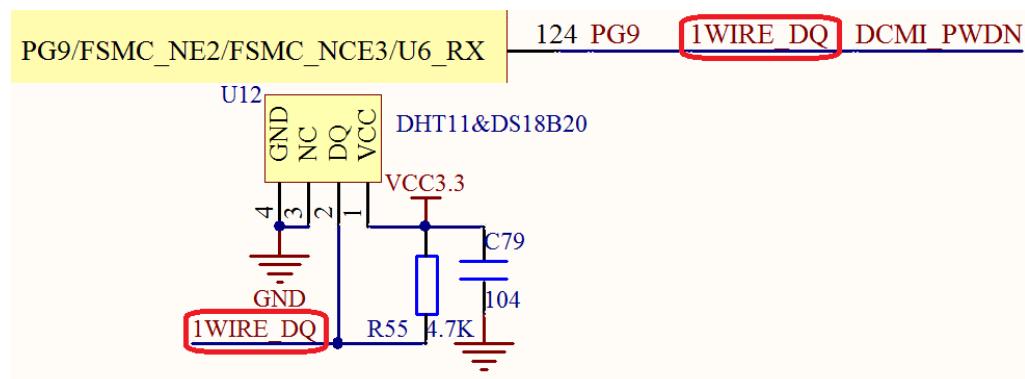


图 35.2.1 DS18B20 接口与 STM32 的连接电路图

从上图可以看出,我们使用的是 STM32 的 PG9 来连接 U12 的 DQ 引脚,图中 U12 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。这里, 1WIRE_DQ 和 DCMI_PWDN 是共用 PG9 的, 所以他们不能同时使用。

DS18B20 只用到 U12 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:

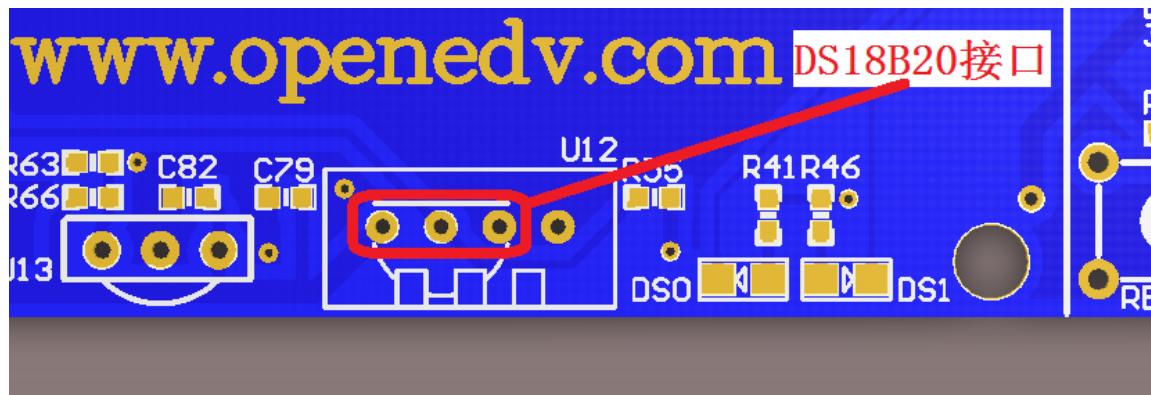


图 35.2.2 DS18B20 连接示意图

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:

```
//复位 DS18B20
void DS18B20_Rst(void)
{
    DS18B20_IO_OUT(); //SET PG11 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 PG11 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)
{
    u8 data;
    DS18B20_IO_OUT();//SET PG11 OUTPUT
    DS18B20_DQ_OUT=0;
    delay_us(2);
    DS18B20_DQ_OUT=1;
    DS18B20_IO_IN();//SET PG11 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)
{
    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 PG11 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)
{
    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_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);//使能 GPIOG 时钟
    //GPIOG9
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//50MHz
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOG, &GPIO_InitStructure);//初始化

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=(double)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型数据, 返回值的范围为-550~1250, 其实就是温度值扩大了 10 倍。

接下来我们打开 ds18b20.h, 可以看到跟 IIC 实验代码很类似, 这里我们不做过多讲解。接下来我们看看主函数代码:

```
int main(void)
```

```
{  
    u8 t=0;  
    short temperature;  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2  
    delay_init(168); //初始化延时函数  
    uart_init(115200); //初始化串口波特率为 115200  
    LED_Init(); //初始化 LED  
    LCD_Init();  
    POINT_COLOR=RED; //设置字体为红色  
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");  
    LCD_ShowString(30,70,200,16,16,"DS18B20 TEST");  
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(30,110,200,16,16,"2014/5/7");  
    while(DS18B20_Init()) //DS18B20 初始化  
    {  
        LCD_ShowString(30,130,200,16,16,"DS18B20 Error");  
        delay_ms(200);  
        LCD_Fill(30,130,239,130+16,WHITE);  
        delay_ms(200);  
    }  
    LCD_ShowString(30,130,200,16,16,"DS18B20 OK");  
    POINT_COLOR=BLUE; //设置字体为蓝色  
    LCD_ShowString(30,150,200,16,16,"Temp: . C");  
    while(1)  
    {  
        if(t%10==0) //每 100ms 读取一次  
        {  
            temperature=DS18B20_Get_Temp();  
            if(temperature<0)  
            {  
                LCD_ShowChar(30+40,150,'-',16,0); //显示负号  
                temperature=-temperature; //转为正数  
            }else LCD_ShowChar(30+40,150,' ',16,0); //去掉负号  
            LCD_ShowNum(30+40+8,150,temperature/10,2,16); //显示正数部分  
            LCD_ShowNum(30+40+32,150,temperature%10,1,16); //显示小数部分  
        }  
        delay_ms(10); t++;  
        if(t==20)  
        {  
            t=0; LED0=!LED0;  
        }  
    }  
}
```

{

主函数代码比较简单，一系列硬件初始化后，在循环中调用 DS18B20_Get_Temp 函数获取温度值，然后显示在 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 探索者 STM32F4 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 实验效果图

该程序还可以读取并显示负温度值的，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 STM32F4 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：

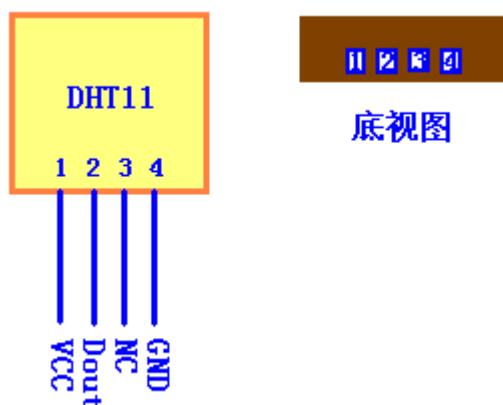


图 36.1.1 DHT11 管脚排列图

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：

$$\text{湿度} = \text{byte4} . \text{byte3} = 45.0 (\%RH)$$

$$\text{温度} = \text{byte2} . \text{byte1} = 28.0 (\text{ }^{\circ}\text{C})$$

$$\text{校验} = \text{byte4} + \text{byte3} + \text{byte2} + \text{byte1} = 73 (= \text{湿度} + \text{温度}) (\text{校验正确})$$

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：

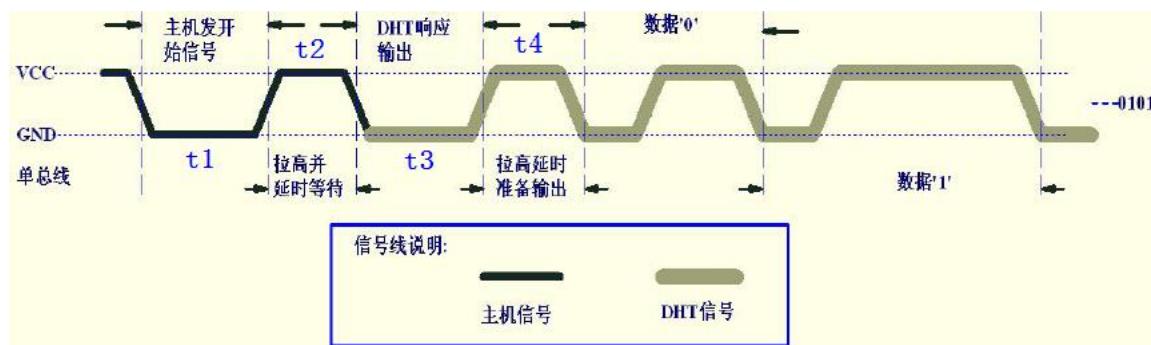


图 36.1.3 DHT11 数据发送流程

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：

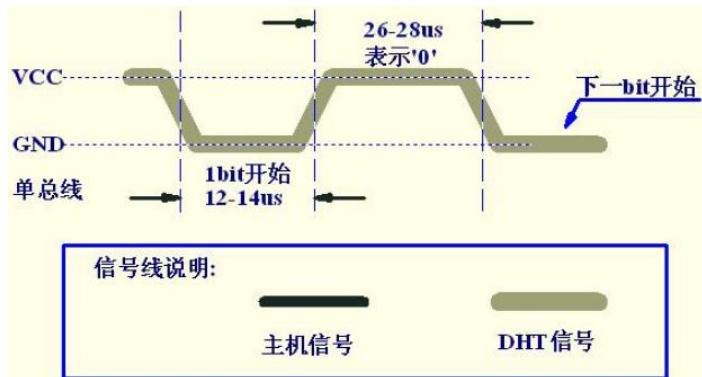


图 36.1.4 DHT11 数字 ‘0’ 时序

DHT11 输出数字 ‘1’ 的时序如图 36.1.5 所示：

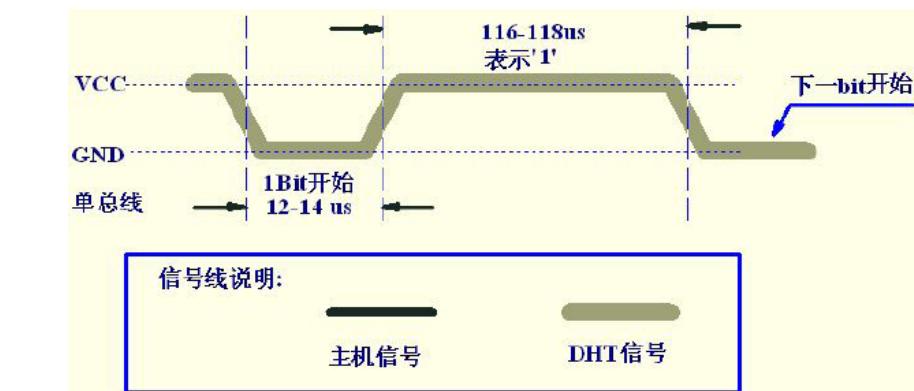


图 36.1.5 DHT11 数字 ‘1’ 时序

通过以上了解，我们就可以通过 STM32F4 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 温湿度传感器

这些我们都已经介绍过了，DHT11 和 DS18B20 的接口是共用一个的，不过 DHT11 有 4 条腿，需要把 U12 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 STM32F4 来读取温湿度值了。连接示意图如图 36.2.1 所示：

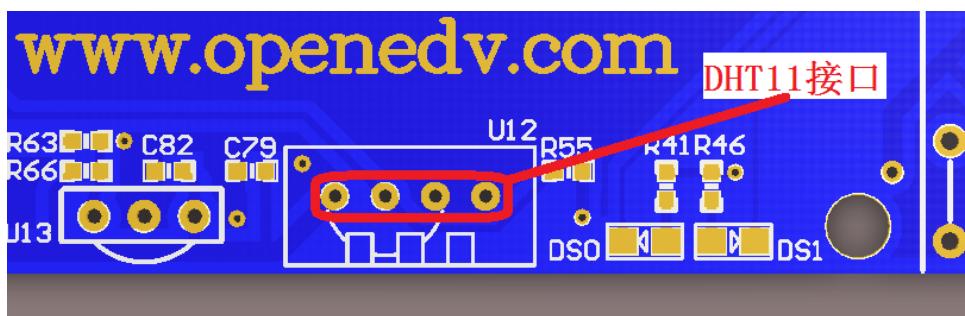


图 36.2.1 DHT11 连接示意图

这里要注意，将 DHT11 贴有字的一面朝内，而有很多孔的一面(网面)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：

```
//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_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE); //使能 GPIOG 时钟  
  
    //GPIOF9,F10 初始化设置  
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 ;  
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //普通输出模式  
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出  
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //50MHz  
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉  
    GPIO_Init(GPIOG, &GPIO_InitStructure); //初始化  
    DHT11_Rst();  
    return DHT11_Check();  
}
```

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 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,temperature,humidity;  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2  
    delay_init(168); //初始化延时函数  
    uart_init(115200); //初始化串口波特率为 115200  
    LED_Init(); //初始化 LED  
    LCD_Init(); //LCD 初始化  
    POINT_COLOR=RED; //设置字体为红色  
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");  
    LCD_ShowString(30,70,200,16,16,"DHT11 TEST");  
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(30,110,200,16,16,"2014/5/7");  
    while(DHT11_Init()) //DHT11 初始化  
    {  
        LCD_ShowString(30,130,200,16,16,"DHT11 Error");  
        delay_ms(200);  
        LCD_Fill(30,130,239,130+16,WHITE);  
        delay_ms(200);  
    }  
    LCD_ShowString(30,130,200,16,16,"DHT11 OK");  
    POINT_COLOR=BLUE; //设置字体为蓝色  
    LCD_ShowString(30,150,200,16,16,"Temp: C");  
    LCD_ShowString(30,170,200,16,16,"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(30+40,150,temperature,2,16);        //显示温度
        LCD_ShowNum(30+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 探索者 STM32F4 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 实验效果图

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 MPU6050 六轴传感器实验

本章，我们介绍当下最流行的一款六轴（三轴加速度+三轴角速度(陀螺仪)）传感器：MPU6050，该传感器广泛用于四轴、平衡车和空中鼠标等设计，具有非常广泛的应用范围。ALIENTEK 探索者 STM32F4 开发板自带了 MPU6050 传感器。本章我们将使用 STM32F4 来驱动 MPU6050，读取其原始数据，并利用其自带的 DMP 实现姿态解算，结合匿名四轴上位机软件和 LCD 显示，教大家如何使用这款功能强大的六轴传感器。本章分为如下几个部分：

37.1 MPU6050 简介

37.2 硬件设计

37.3 软件设计

37.4 下载验证

37.1 MPU6050 简介

本节，我们将分 2 个部分介绍：1，MPU6050 基础介绍。2，DMP 使用简介。

37.1.1 MPU6050 基础介绍

MPU6050 是 InvenSense 公司推出的全球首款整合性 6 轴运动处理组件，相较于多组件方案，免除了组合陀螺仪与加速器时之轴间差的问题，减少了安装空间。

MPU6050 内部整合了 3 轴陀螺仪和 3 轴加速度传感器，并且含有一个第二 IIC 接口，可用于连接外部磁力传感器，并利用自带的数字运动处理器（DMP: Digital Motion Processor）硬件加速引擎，通过主 IIC 接口，向应用端输出完整的 9 轴融合演算数据。有了 DMP，我们可以使用 InvenSense 公司提供的运动处理资料库，非常方便的实现姿态解算，降低了运动处理运算对操作系统的负荷，同时大大降低了开发难度。

MPU6050 的特点包括：

- ① 以数字形式输出 6 轴或 9 轴（需外接磁传感器）的旋转矩阵、四元数(quaternion)、欧拉角格式(Euler Angle forma)的融合演算数据（需 DMP 支持）
- ② 具有 131 LSBs/° /sec 敏感度与全格感测范围为±250、±500、±1000 与±2000° /sec 的 3 轴角速度感测器(陀螺仪)
- ③ 集成可程序控制，范围为±2g、±4g、±8g 和±16g 的 3 轴加速度传感器
- ④ 移除加速器与陀螺仪轴间敏感度，降低设定给予的影响与感测器的飘移
- ⑤ 自带数字运动处理(DMP: Digital Motion Processing)引擎可减少 MCU 复杂的融合演算数据、感测器同步化、姿势感应等的负荷
- ⑥ 内建运作时间偏差与磁力感测器校正演算技术，免除了客户须另外进行校正的需求
- ⑦ 自带一个数字温度传感器
- ⑧ 带数字输入同步引脚(Sync pin)支持视频电子影相稳定技术与 GPS
- ⑨ 可程序控制的中断(interrupt)，支持姿势识别、摇摄、画面放大缩小、滚动、快速下降中断、high-G 中断、零动作感应、触击感应、摇动感应功能
- ⑩ VDD 供电电压为 2.5V±5%、3.0V±5%、3.3V±5%；VLOGIC 可低至 1.8V± 5%
- ⑪ 陀螺仪工作电流：5mA，陀螺仪待机电流：5uA；加速器工作电流：500uA，加速器省电模式电流：40uA@10Hz
- ⑫ 自带 1024 字节 FIFO，有助于降低系统功耗

(13) 高达 400Khz 的 IIC 通信接口

(14) 超小封装尺寸: 4x4x0.9mm (QFN)

MPU6050 传感器的检测轴如图 37.1.1.1 所示:

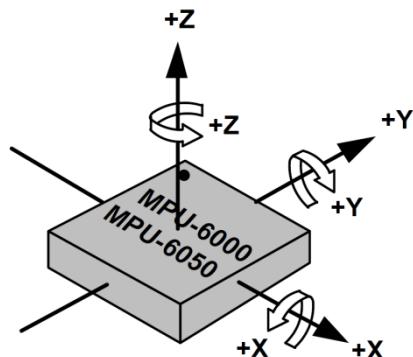


图 37.1.1.1 MPU6050 检测轴及其方向

MPU6050 的内部框图如图 37.1.1.2 所示:

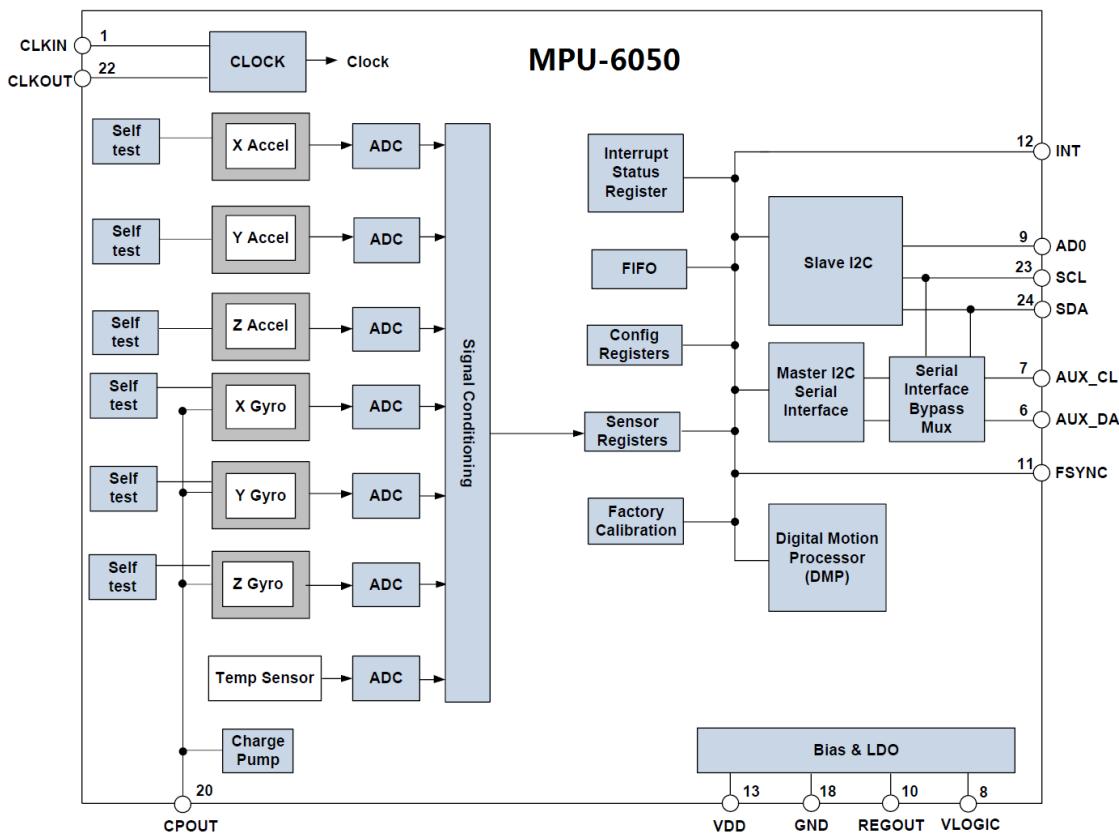


图 37.1.1.2 MPU6050 框图

其中, SCL 和 SDA 是连接 MCU 的 IIC 接口, MCU 通过这个 IIC 接口来控制 MPU6050, 另外还有一个 IIC 接口: AUX_CL 和 AUX_DA, 这个接口可用来连接外部从设备, 比如磁传感器, 这样就可以组成一个九轴传感器。VLOGIC 是 IO 口电压, 该引脚最低可以到 1.8V, 我们一般直接接 VDD 即可。AD0 是从 IIC 接口 (接 MCU) 的地址控制引脚, 该引脚控制 IIC 地址的最低位。如果接 GND, 则 MPU6050 的 IIC 地址是: 0X68, 如果接 VDD, 则是 0X69, 注意: 这里的地址是不包含数据传输的最低位的 (最低位用来表示读写)!!

在探索者 STM32F4 开发板上, AD0 是接 GND 的, 所以 MPU6050 的 IIC 地址是 0X68 (不含最低位), IIC 通信的时序我们在之前已经介绍过 (第二十九章, IIC 实验), 这里就不再细说

了。

接下来，我们介绍一下利用 STM32F4 读取 MPU6050 的加速度和角度传感器数据（非中断方式），需要哪些初始化步骤：

1) 初始化 IIC 接口

MPU6050 采用 IIC 与 STM32F4 通信，所以我们需要先初始化与 MPU6050 连接的 SDA 和 SCL 数据线。这个在前面的 IIC 实验章节已经介绍过了，这里 MPU6050 与 24C02 共用一个 IIC，所以初始化 IIC 完全一模一样。

2) 复位 MPU6050

这一步让 MPU6050 内部所有寄存器恢复默认值，通过对电源管理寄存器 1 (0X6B) 的 bit7 写 1 实现。复位后，电源管理寄存器 1 恢复默认值(0X40)，然后必须设置该寄存器为 0X00，以唤醒 MPU6050，进入正常工作状态。

3) 设置角速度传感器（陀螺仪）和加速度传感器的满量程范围

这一步，我们设置两个传感器的满量程范围(FSR)，分别通过陀螺仪配置寄存器 (0X1B) 和加速度传感器配置寄存器 (0X1C) 设置。我们一般设置陀螺仪的满量程范围为±2000dps，加速度传感器的满量程范围为±2g。

4) 设置其他参数

这里，我们还需要配置的参数有：关闭中断、关闭 AUX IIC 接口、禁止 FIFO、设置陀螺仪采样率和设置数字低通滤波器(DLPF)等。本章我们不用中断方式读取数据，所以关闭中断，然后也没用到 AUX IIC 接口外接其他传感器，所以也关闭这个接口。分别通过中断使能寄存器 (0X38) 和用户控制寄存器 (0X6A) 控制。MPU6050 可以使用 FIFO 存储传感器数据，不过本章我们没有用到，所以关闭所有 FIFO 通道，这个通过 FIFO 使能寄存器 (0X23) 控制，默认都是 0(即禁止 FIFO)，所以用默认值就可以了。陀螺仪采样率通过采样率分频寄存器(0X19) 控制，这个采样率我们一般设置为 50 即可。数字低通滤波器(DLPF)则通过配置寄存器 (0X1A) 设置，一般设置 DLPF 为带宽的 1/2 即可。

5) 配置系统时钟源并使能角速度传感器和加速度传感器

系统时钟源同样是通过电源管理寄存器 1 (0X1B) 来设置，该寄存器的最低三位用于设置系统时钟源选择，默认值是 0 (内部 8M RC 震荡)，不过我们一般设置为 1，选择 x 轴陀螺 PLL 作为时钟源，以获得更高精度的时钟。同时，使能角速度传感器和加速度传感器，这两个操作通过电源管理寄存器 2 (0X6C) 来设置，设置对应位为 0 即可开启。

至此，MPU6050 的初始化就完成了，可以正常工作了（其他未设置的寄存器全部采用默认值即可），接下来，我们就可以读取相关寄存器，得到加速度传感器、角速度传感器和温度传感器的数据了。不过，我们先简单介绍几个重要的寄存器。

首先，我们介绍电源管理寄存器 1，该寄存器地址为 0X6B，各位描述如图 37.1.1.3 所示：

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6B	107	DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		

图 37.1.1.3 电源管理寄存器 1 各位描述

其中，DEVICE_RESET 位用来控制复位，设置为 1，复位 MPU6050，复位结束后，MPU 硬件自动清零该位。SLEEP 位用于控制 MPU6050 的工作模式，复位后，该位为 1，即进入了睡眠模式（低功耗），所以我们要清零该位，以进入正常工作模式。TEMP_DIS 用于设置是否使能温度传感器，设置为 0，则使能。最后 CLKSEL[2:0]用于选择系统时钟源，选择关系如表 37.1.1.1 所示：

CLKSEL[2:0]	时钟源
-------------	-----

000	内部 8M RC 晶振
001	PLL, 使用 X 轴陀螺作为参考
010	PLL, 使用 Y 轴陀螺作为参考
011	PLL, 使用 Z 轴陀螺作为参考
100	PLL, 使用外部 32.768Khz 作为参考
101	PLL, 使用外部 19.2Mhz 作为参考
110	保留
111	关闭时钟, 保持时序产生电路复位状态

图 37.1.1.1 CLKSEL 选择列表

默认是使用内部 8M RC 晶振的, 精度不高, 所以我们一般选择 X/Y/Z 轴陀螺作为参考的 PLL 作为时钟源, 一般设置 CLKSEL=001 即可。

接着, 我们看陀螺仪配置寄存器, 该寄存器地址为: 0X1B, 各位描述如图 37.1.4 所示:

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]	-	-	-	-

图 37.1.1.4 陀螺仪配置寄存器各位描述

该寄存器我们只关心 FS_SEL[1:0]这两个位, 用于设置陀螺仪的满量程范围: 0, ±250° /S; 1, ±500° /S; 2, ±1000° /S; 3, ±2000° /S; 我们一般设置为 3, 即±2000° /S, 因为陀螺仪的 ADC 为 16 位分辨率, 所以得到灵敏度为: $65536/4000=16.4\text{LSB}/(\text{°}/\text{S})$ 。

接下来, 我们看加速度传感器配置寄存器, 寄存器地址为: 0X1C, 各位描述如图 37.1.1.5 所示:

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1C	28	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]	-	-	-	-

图 37.1.1.5 加速度传感器配置寄存器各位描述

该寄存器我们只关心 AFS_SEL[1:0]这两个位, 用于设置加速度传感器的满量程范围: 0, ±2g; 1, ±4g; 2, ±8g; 3, ±16g; 我们一般设置为 0, 即±2g, 因为加速度传感器的 ADC 也是 16 位, 所以得到灵敏度为: $65536/4=16384\text{LSB/g}$ 。

接下来, 我看看 FIFO 使能寄存器, 寄存器地址为: 0X1C, 各位描述如图 37.1.1.6 所示:

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
23	35	TEMP_FIFO_EN	XG_FIFO_EN	YG_FIFO_EN	ZG_FIFO_EN	ACCEL_FIFO_EN	SLV2_FIFO_EN	SLV1_FIFO_EN	SLV0_FIFO_EN

图 37.1.1.6 FIFO 使能寄存器各位描述

该寄存器用于控制 FIFO 使能, 在简单读取传感器数据的时候, 可以不用 FIFO, 设置对应位为 0 即可禁止 FIFO, 设置为 1, 则使能 FIFO。注意加速度传感器的 3 个轴, 全由 1 个位 (ACCEL_FIFO_EN) 控制, 只要该位置 1, 则加速度传感器的三个通道都开启 FIFO 了。

接下来, 我们看陀螺仪采样率分频寄存器, 寄存器地址为: 0X19, 各位描述如图 37.1.1.7 所示:

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
19	25	SMPLRT_DIV[7:0]							

图 37.1.1.7 陀螺仪采样率分频寄存器各位描述

该寄存器用于设置 MPU6050 的陀螺仪采样频率, 计算公式为:

采样频率 = 陀螺仪输出频率 / (1+SMPLRT_DIV)

这里陀螺仪的输出频率，是 1Khz 或者 8Khz，与数字低通滤波器（DLPF）的设置有关，当 DLPF_CFG=0/7 的时候，频率为 8Khz，其他情况是 1Khz。而且 DLPF 滤波频率一般设置为采样率的一半。采样率，我们假定设置为 50Hz，那么 SMPLRT_DIV=1000/50-1=19。

接下来，我们看配置寄存器，寄存器地址为：0X1A，各位描述如图 37.1.1.8 所示：

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	-	-	EXT_SYNC_SET[2:0]	DLPF_CFG[2:0]				

图 37.1.1.8 配置寄存器各位描述

这里，我们主要关心数字低通滤波器（DLPF）的设置位，即：DLPF_CFG[2:0]，加速度计和陀螺仪，都是根据这三个位的配置进行过滤的。DLPF_CFG 不同配置对应的过滤情况如表 37.1.2 所示：

DLPF_CFG[2:0]	加速度传感器		角速度传感器 (陀螺仪)		
	Fs=1Khz	带宽 (Hz)	延迟 (ms)	带宽 (Hz)	延迟 (ms)
000	260	0	256	0.98	8
001	184	2.0	188	1.9	1
010	94	3.0	98	2.8	1
011	44	4.9	42	4.8	1
100	21	8.5	20	8.3	1
101	10	13.8	10	13.4	1
110	5	19.0	5	18.6	1
111	保留		保留		8

图 37.1.1.2 DLPF_CFG 配置表

这里的加速度传感器，输出速率（Fs）固定是 1Khz，而角速度传感器的输出速率（Fs），则根据 DLPF_CFG 的配置有所不同。一般我们设置角速度传感器的带宽为其采样率的一半，如前面所说的，如果设置采样率为 50Hz，那么带宽就应该设置为 25Hz，取近似值 20Hz，就应该设置 DLPF_CFG=100。

接下来，我们看电源管理寄存器 2，寄存器地址为：0X6C，各位描述如图 37.1.1.9 所示：

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6C	108	LP_WAKE_CTRL[1:0]	STBY_XA	STBY_YA	STBY_ZA	STBY_XG	STBY_YG	STBY_ZG	

图 37.1.1.9 电源管理寄存器 2 各位描述

该寄存器的 LP_WAKE_CTRL 用于控制低功耗时的唤醒频率，本章用不到。剩下的 6 位，分别控制加速度和陀螺仪的 x/y/z 轴是否进入待机模式，这里我们全部都不进入待机模式，所以全部设置为 0 即可。

接下来，我们看看陀螺仪数据输出寄存器，总共有 6 个寄存器组成，地址为：0X43~0X48，通过读取这 6 个寄存器，就可以读到陀螺仪 x/y/z 轴的值，比如 x 轴的数据，可以通过读取 0X43（高 8 位）和 0X44（低 8 位）寄存器得到，其他轴以此类推。

同样，加速度传感器数据输出寄存器，也有 8 个，地址为：0X3B~0X40，通过读取这 8 个寄存器，就可以读到加速度传感器 x/y/z 轴的值，比如读 x 轴的数据，可以通过读取 0X3B（高 8 位）和 0X3C（低 8 位）寄存器得到，其他轴以此类推。

最后，温度传感器的值，可以通过读取 0X41（高 8 位）和 0X42（低 8 位）寄存器得到，温度换算公式为：

$$\text{Temperature} = 36.53 + \text{regval}/340$$

其中，Temperature 为计算得到的温度值，单位为°C，regval 为从 0X41 和 0X42 读到的温度传感器值。

关于 MPU6050 的基础介绍，我们就介绍到这。MPU6050 的详细资料和相关寄存器介绍，请参考光盘：7，硬件资料→MPU6050 资料→MPU-6000 and MPU-6050 Product Specification.pdf 和 MPU-6000 and MPU-6050 Register Map and Descriptions.pdf 这两个文档，另外该目录还提供了部分 MPU6050 的中文资料，供大家参考学习。

37.1.2 DMP 使用简介

经过 37.1.1 节的介绍，我们可以读出 MPU6050 的加速度传感器和角速度传感器的原始数据。不过这些原始数据，对想搞四轴之类的初学者来说，用处不大，我们期望得到的是姿态数据，也就是欧拉角：航向角（yaw）、横滚角（roll）和俯仰角（pitch）。有了这三个角，我们就可以得到当前四轴的姿态，这才是我们想要的结果。

要得到欧拉角数据，就得利用我们的原始数据，进行姿态融合解算，这个比较复杂，知识点比较多，初学者 不易掌握。而 MPU6050 自带了数字运动处理器，即 DMP，并且，InvenSense 提供了一个 MPU6050 的嵌入式运动驱动库，结合 MPU6050 的 DMP，可以将我们的原始数据，直接转换成四元数输出，而得到四元数之后，就可以很方便的计算出欧拉角，从而得到 yaw、roll 和 pitch。

使用内置的 DMP，大大简化了四轴的代码设计，且 MCU 不用进行姿态解算过程，大大降低了 MCU 的负担，从而有更多的时间去处理其他事件，提高系统实时性。

使用 MPU6050 的 DMP 输出的四元数是 q30 格式的，也就是浮点数放大了 2 的 30 次方倍。在换算成欧拉角之前，必须先将其转换为浮点数，也就是除以 2 的 30 次方，然后再进行计算，计算公式为：

```

q0=quat[0] / q30; //q30 格式转换为浮点数
q1=quat[1] / q30;
q2=quat[2] / q30;
q3=quat[3] / q30;
//计算得到俯仰角/横滚角/航向角
pitch=asin(-2 * q1 * q3 + 2 * q0 * q2)* 57.3; //俯仰角
roll=atan2(2 * q2 * q3 + 2 * q0 * q1, -2 * q1 * q1 - 2 * q2 * q2 + 1)* 57.3; //横滚角
yaw=atan2(2*(q1*q2 + q0*q3),q0*q0+q1*q1-q2*q2-q3*q3) * 57.3; //航向角

```

其中 quat[0]~ quat[3] 是 MPU6050 的 DMP 解算后的四元数，q30 格式，所以要除以一个 2 的 30 次方，其中 q30 是一个常量：1073741824，即 2 的 30 次方，然后带入公式，计算出欧拉角。上述计算公式的 57.3 是弧度转换为角度，即 $180/\pi$ ，这样得到的结果就是以度（°）为单位的。关于四元数与欧拉角的公式推导，这里我们不进行讲解，感兴趣的朋友，可以自行查阅相关资料学习。

InvenSense 提供的 MPU6050 运动驱动库是基于 MSP430 的，我们需要将其移植一下，才可以用到 STM32F4 上面，官方原版驱动在光盘：7，硬件资料→MPU6050 资料→DMP 资料→Embedded_MotionDriver_5.1.rar，这就是官方原版的驱动，代码比较多，不过官方提供了两个资料供大家学习：Embedded Motion Driver V5.1.1 API 说明.pdf 和 Embedded Motion Driver V5.1.1 教程.pdf，这两个文件都在 DMP 资料文件夹里面，大家可以阅读这两个文件，来熟悉官

方驱动库的使用。

官方 DMP 驱动库移植起来,还是比较简单的,主要是实现这 4 个函数:i2c_write,i2c_read, delay_ms 和 get_ms, 具体细节, 我们就不详细介绍了, 移植后的驱动代码, 我们放在本例程 →HARDWARE→MPU6050→eMPL 文件夹内, 总共 6 个文件, 如图 37.1.2.1 所示:

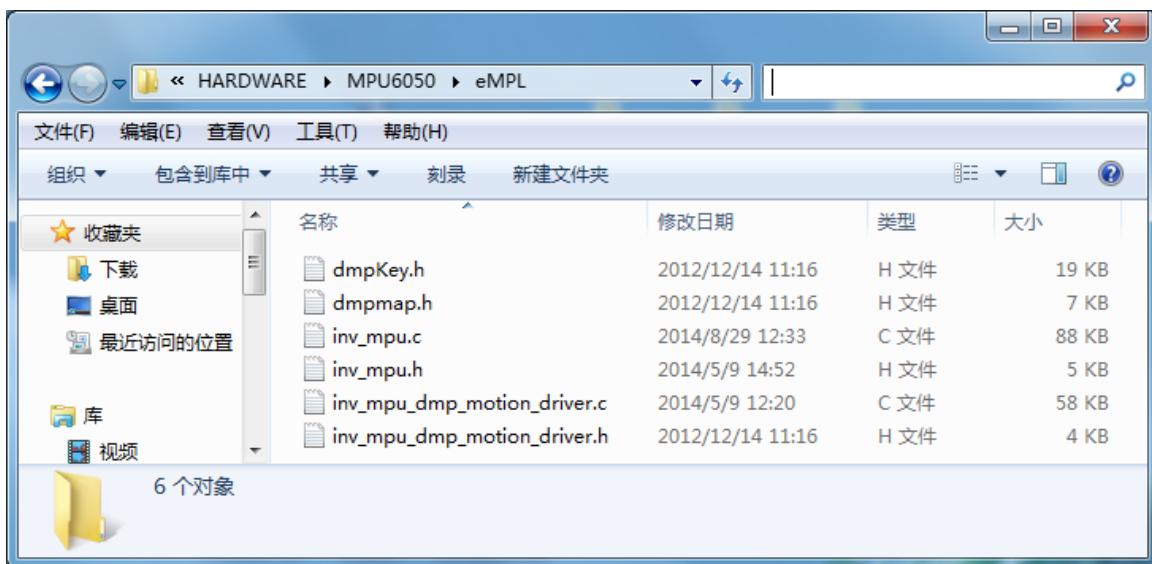


图 37.1.2.1 移植后的驱动库代码

该驱动库, 重点就是两个 c 文件: inv_mpu.c 和 inv_mpu_dmp_motion_driver.c。其中我们在 inv_mpu.c 添加了几个函数, 方便我们使用, 重点是两个函数: mpu_dmp_init 和 mpu_dmp_get_data 这两个函数, 这里我们简单介绍下这两个函数。

mpu_dmp_init, 是 MPU6050 DMP 初始化函数, 该函数代码如下:

```
//mpu6050,dmp 初始化
//返回值:0,正常
//    其他,失败
u8 mpu_dmp_init(void)
{
    u8 res=0;
    IIC_Init();          //初始化 IIC 总线
    if(mpu_init()==0)   //初始化 MPU6050
    {
        res=mpu_set_sensors(INV_XYZ_GYRO|INV_XYZ_ACCEL); //设置需要的传感器
        if(res)return 1;
        res=mpu_configure_fifo(INV_XYZ_GYRO|INV_XYZ_ACCEL); //设置 FIFO
        if(res)return 2;
        res=mpu_set_sample_rate(DEFAULT_MPU_HZ);      //设置采样率
        if(res)return 3;
        res=dmp_load_motion_driver_firmware();           //加载 dmp 固件
        if(res)return 4;
        res=dmp_set_orientation(inv_orientation_matrix_to_scalar(gyro_orientation));
        //设置陀螺仪方向
        if(res)return 5;
    }
}
```

```

res=dmp_enable_feature(DMP_FEATURE_6X_LP_QUAT|DMP_FEATURE_TAP|
DMP_FEATURE_ANDROID_ORIENT|DMP_FEATURE_SEND_RAW_ACCEL|
DMP_FEATURE_SEND_CAL_GYRO|DMP_FEATURE_GYRO_CAL);
//设置 dmp 功能
if(res) return 6;
res=dmp_set_fifo_rate(DEFAULT_MPU_HZ);//设置 DMP 输出速率(最大 200Hz)
if(res) return 7;
res=run_self_test(); //自检
if(res) return 8;
res=mpu_set_dmp_state(1); //使能 DMP
if(res) return 9;
}
return 0;
}

```

此函数首先通过 IIC_Init(需外部提供)初始化与 MPU6050 连接的 IIC 接口，然后调用 mpu_init 函数，初始化 MPU6050，之后就是设置 DMP 所用传感器、FIFO、采样率和加载固件等一些列操作，在所有操作都正常之后，最后通过 mpu_set_dmp_state(1)使能 DMP 功能，在使能成功以后，我们便可以通过 mpu_dmp_get_data 来读取姿态解算后的数据了。

mpu_dmp_get_data 函数代码如下：

```

//得到 dmp 处理后的数据(注意,本函数需要比较多堆栈,局部变量有点多)
//pitch:俯仰角 精度:0.1° 范围:-90.0° <---> +90.0°
//roll:横滚角 精度:0.1° 范围:-180.0° <---> +180.0°
//yaw:航向角 精度:0.1° 范围:-180.0° <---> +180.0°
//返回值:0,正常 其他,失败
u8 mpu_dmp_get_data(float *pitch,float *roll,float *yaw)
{
    float q0=1.0f,q1=0.0f,q2=0.0f,q3=0.0f;
    unsigned long sensor_timestamp;
    short gyro[3], accel[3], sensors;
    unsigned char more;
    long quat[4];
    if(dmp_read_fifo(gyro, accel, quat, &sensor_timestamp, &sensors,&more))return 1;
    if(sensors&INV_WXYZ_QUAT)
    {
        q0 = quat[0] / q30; //q30 格式转换为浮点数
        q1 = quat[1] / q30;
        q2 = quat[2] / q30;
        q3 = quat[3] / q30;
        //计算得到俯仰角/横滚角/航向角
        *pitch = asin(-2 * q1 * q3 + 2 * q0 * q2)* 57.3;// pitch
        *roll = atan2(2 * q2 * q3 + 2 * q0 * q1, -2 * q1 * q1 - 2 * q2 * q2 + 1)* 57.3;// roll
        *yaw= atan2(2*(q1*q2 + q0*q3),q0*q0+q1*q1-q2*q2-q3*q3) * 57.3;//yaw
    }else return 2;
}

```

```
    return 0;  
}
```

此函数用于得到 DMP 姿态解算后的俯仰角、横滚角和航向角。不过本函数局部变量有点多，大家在使用的时候，如果死机，那么请设置堆栈大一点(在 startup_stm32f40_41xxx.s 里面设置，默认是 400)。这里就用到了我们前面介绍的四元数转欧拉角公式，将 dmp_read_fifo 函数读到的 q30 格式四元数转换成欧拉角。

利用这两个函数，我们就可以读取到姿态解算后的欧拉角，使用非常方便。DMP 部分，我们就介绍到这。

37.2 硬件设计

本实验采用 STM32F4 的 3 个普通 IO 连接 MPU6050，本章实验功能简介：程序先初始化 MPU6050 等外设，然后利用 DMP 库，初始化 MPU6050 及使能 DMP，最后，在死循环里面不停读取：温度传感器、加速度传感器、陀螺仪、DMP 姿态解算后的欧拉角等数据，通过串口上报给上位机（温度不上报），利用上位机软件（ANO_Tech 匿名四轴上位机_V2.6.exe），可以实时显示 MPU6050 的传感器状态曲线，并显示 3D 姿态，可以通过 KEY0 按键开启/关闭数据上传功能。同时，在 LCD 模块上面显示温度和欧拉角等信息。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 按键
 - 3) TFTLCD 模块
 - 4) 串口
 - 5) MPU6050

前 4 个，在之前的实例已经介绍过了，这里我们仅介绍 MPU6050 与探索者 STM32F4 开发板的连接。该接口与 MCU 的连接原理图如 37.2.1 所示：

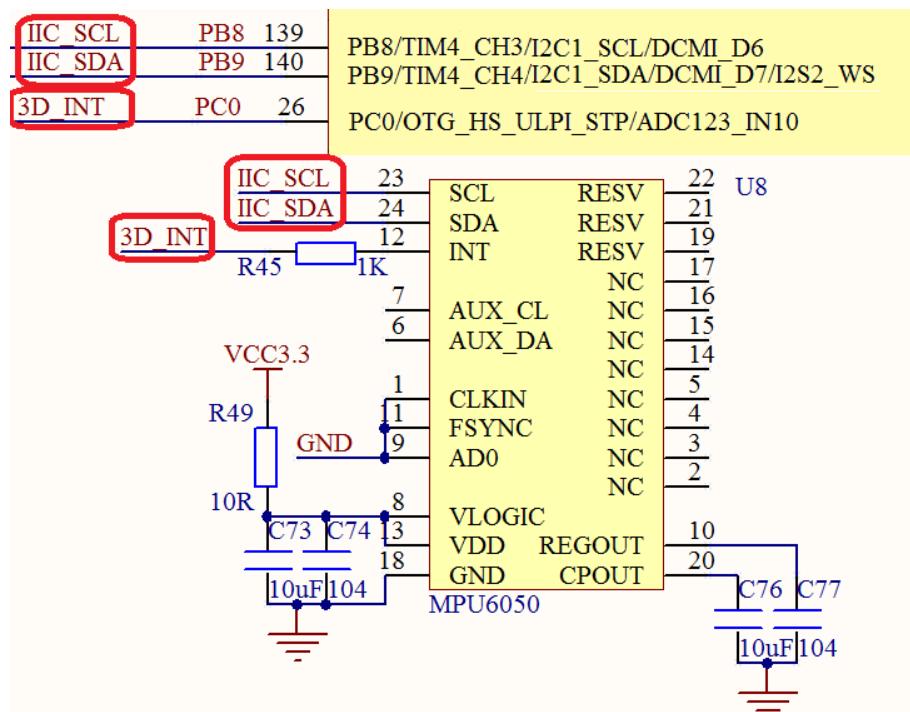


图 37.2.1 MPU6050 与 STM32F4 的连接电路图

从上图可以看出, MPU6050 通过三根线与 STM32F4 开发板连接, 其中 IIC 总线时和 24C02 以及 WM8978 共用, 接在 PB8 和 PB9 上面。MPU6050 的中断输出, 连接在 STM32F4 的 PC0 脚, 不过本例程我们并没有用到中断。另外, AD0 接的 GND, 所以 MPU6050 的器件地址是: 0X68。

37.3 软件设计

打开本章工程可以看到, 首先我们在工程中 HARDWARE 分组下首先添加了 IIC 支持的底层驱动文件 myiic.c 和源文件 myiic.h, 因为我们的 mpu6050 通信接口是 IIC。同时我们还增加了 mpu6050.c 源文件和对应的头文件 mpu6050 用来编写 mpu6050 相关的底层驱动。最后我们还添加了 DMP 驱动库代码到我们实验工程, DMP 驱动库代码包含 inv_mpu.c 和 inv_mpu_dmp_motion_driver.c 两个源文件, 以及几个头文件。

由于 mpu6050.c 里面代码比较多, 这里我们就不全部列出来了, 仅介绍几个重要的函数。

首先是: **MPU_Init**, 该函数代码如下:

```
//初始化 MPU6050
//返回值:0,成功      其他,错误代码
u8 MPU_Init(void)
{
    u8 res;
    IIC_Init(); //初始化 IIC 总线
    MPU_Write_Byte(MPU_PWR_MGMT1_REG, 0X80); //复位 MPU6050
    delay_ms(100);
    MPU_Write_Byte(MPU_PWR_MGMT1_REG, 0X00); //唤醒 MPU6050
    MPU_Set_Gyro_Fsr(3); //陀螺仪传感器,±2000dps
    MPU_Set_Accel_Fsr(0); //加速度传感器,±2g
    MPU_Set_Rate(50); //设置采样率 50Hz
    MPU_Write_Byte(MPU_INT_EN_REG, 0X00); //关闭所有中断
    MPU_Write_Byte(MPU_USER_CTRL_REG, 0X00); //I2C 主模式关闭
    MPU_Write_Byte(MPU_FIFO_EN_REG, 0X00); //关闭 FIFO
    MPU_Write_Byte(MPU_INTBP_CFG_REG, 0X80); //INT 引脚低电平有效
    res=MPU_Read_Byte(MPU_DEVICE_ID_REG);
    if(res==MPU_ADDR) //器件 ID 正确
    {
        MPU_Write_Byte(MPU_PWR_MGMT1_REG, 0X01); //设置 CLKSEL,PLL X 轴参考
        MPU_Write_Byte(MPU_PWR_MGMT2_REG, 0X00); //加速度与陀螺仪都工作
        MPU_Set_Rate(50); //设置采样率为 50Hz
    }else return 1;
    return 0;
}
```

该函数就是按我们在 37.1.1 节介绍的方法, 对 MPU6050 进行初始化, 该函数执行成功后, 便可以读取传感器数据了。

然后再看 **MPU_Get_Temperature**、**MPU_Get_Gyroscope** 和 **MPU_Get_Accelerometer** 等三个函数, 源码如下:

```
//得到温度值
```

```
//返回值:温度值(扩大了 100 倍)
short MPU_Get_Temperature(void)
{
    u8 buf[2];
    short raw; float temp;
    MPU_Read_Len(MPU_ADDR,MPU_TEMP_OUTH_REG,2,buf);
    raw=((u16)buf[0]<<8)|buf[1];
    temp=36.53+((double)raw)/340;
    return temp*100;;
}

//得到陀螺仪值(原始值)
//gx,gy,gz:陀螺仪 x,y,z 轴的原始读数(带符号)
//返回值:0,成功
//      其他,错误代码
u8 MPU_Get_Gyroscope(short *gx,short *gy,short *gz)
{
    u8 buf[6],res;
    res=MPU_Read_Len(MPU_ADDR,MPU_GYRO_XOUTH_REG,6,buf);
    if(res==0)
    {
        *gx=((u16)buf[0]<<8)|buf[1];
        *gy=((u16)buf[2]<<8)|buf[3];
        *gz=((u16)buf[4]<<8)|buf[5];
    }
    return res;;
}

//得到加速度值(原始值)
//gx,gy,gz:陀螺仪 x,y,z 轴的原始读数(带符号)
//返回值:0,成功
//      其他,错误代码
u8 MPU_Get_Accelerometer(short *ax,short *ay,short *az)
{
    u8 buf[6],res;
    res=MPU_Read_Len(MPU_ADDR,MPU_ACCEL_XOUTH_REG,6,buf);
    if(res==0)
    {
        *ax=((u16)buf[0]<<8)|buf[1];
        *ay=((u16)buf[2]<<8)|buf[3];
        *az=((u16)buf[4]<<8)|buf[5];
    }
    return res;;
}
```

其中 `MPU_Get_Temperature` 用于获取 MPU6050 自带温度传感器的温度值，然后

MPU_Get_Gyroscope 和 MPU_Get_Accelerometer 分别用于读取陀螺仪和加速度传感器的原始数据。

最后看 MPU_Write_Len 和 MPU_Read_Len 这两个函数，代码如下：

```
//IIC 连续写
//addr:器件地址    reg:寄存器地址
//len:写入长度      buf:数据区
//返回值:0,正常     其他,错误代码
u8 MPU_Write_Len(u8 addr,u8 reg,u8 len,u8 *buf)
{
    u8 i;
    IIC_Start();
    IIC_Send_Byte((addr<<1)|0); //发送器件地址+写命令
    if(IIC_Wait_Ack()){IIC_Stop();return 1;} //等待应答
    IIC_Send_Byte(reg); //写寄存器地址
    IIC_Wait_Ack(); //等待应答
    for(i=0;i<len;i++)
    {
        IIC_Send_Byte(buf[i]); //发送数据
        if(IIC_Wait_Ack()) {IIC_Stop();return 1;} //等待 ACK
    }
    IIC_Stop();
    return 0;
}
//IIC 连续读
//addr:器件地址      reg:要读取的寄存器地址
//len:要读取的长度    buf:读取到的数据存储区
//返回值:0,正常       其他,错误代码
u8 MPU_Read_Len(u8 addr,u8 reg,u8 len,u8 *buf)
{
    IIC_Start();
    IIC_Send_Byte((addr<<1)|0); //发送器件地址+写命令
    if(IIC_Wait_Ack()){ IIC_Stop();return 1;} //等待应答
    IIC_Send_Byte(reg); //写寄存器地址
    IIC_Wait_Ack(); //等待应答
    IIC_Start();
    IIC_Send_Byte((addr<<1)|1); //发送器件地址+读命令
    IIC_Wait_Ack(); //等待应答
    while(len)
    {
        if(len==1)*buf=IIC_Read_Byte(0); //读数据,发送 nACK
        else *buf=IIC_Read_Byte(1); //读数据,发送 ACK
        len--; buf++;
    }
}
```

```
IIC_Stop(); //产生一个停止条件  
return 0;  
}
```

MPU_Write_Len 用于指定器件和地址，连续写数据，可用于实现 DMP 部分的：i2c_write 函数。而 **MPU_Read_Len** 用于指定器件和地址，连续读数据，可用于实现 DMP 部分的：i2c_read 函数。DMP 移植部分的 4 个函数，这里就实现了 2 个，剩下的 delay_ms 就直接采用我们 delay.c 里面的 delay_ms 实现，get_ms 则直接提供一个空函数即可。

关于 mpu6050.c 我们就介绍到这，另外 mpu6050.h 的代码，我们这里就不再贴出了，大家看光盘源码即可。

最后看看 main.c 代码如下：

```
//串口 1 发送 1 个字符  
//c:要发送的字符  
void usart1_send_char(u8 c)  
{  
    while(USART_GetFlagStatus(USART1,USART_FLAG_TC)==RESET);  
    USART_SendData(USART1,c);  
}  
//传送数据给匿名四轴上位机软件(V2.6 版本)  
//fun:功能字. 0XA0~0XAF  
//data:数据缓存区,最多 28 字节!!  
//len: data 区有效数据个数  
void usart1_niming_report(u8 fun,u8*data,u8 len)  
{  
    u8 send_buf[32],i;  
    if(len>28)return; //最多 28 字节数据  
    send_buf[len+3]=0;//校验数置零  
    send_buf[0]=0X88;//帧头  
    send_buf[1]=fun; //功能字  
    send_buf[2]=len; //数据长度  
    for(i=0;i<len;i++)send_buf[3+i]=data[i]; //复制数据  
    for(i=0;i<len+3;i++)send_buf[len+3]+=send_buf[i]; //计算校验和  
    for(i=0;i<len+4;i++)usart1_send_char(send_buf[i]); //发送数据到串口 1  
}  
//发送加速度传感器数据和陀螺仪数据  
//aacx,aacy,aacz:x,y,z 三个方向上面的加速度值  
//gyrox,gyroy,gyroz:x,y,z 三个方向上面的陀螺仪值  
void mpu6050_send_data(short aacx,short aacy,short aacz,short gyrox,short gyroy,short gyroz)  
{  
    u8 tbuf[12];  
    tbuf[0]=(aacx>>8)&0xFF; tbuf[1]=aacx&0xFF;  
    tbuf[2]=(aacy>>8)&0xFF; tbuf[3]=aacy&0xFF;  
    tbuf[4]=(aacz>>8)&0xFF; tbuf[5]=aacz&0xFF;  
    tbuf[6]=(gyrox>>8)&0xFF; tbuf[7]=gyrox&0xFF;
```

```
tbuf[8]=(gyroy>>8)&0XFF; tbuf[9]=gyroy&0XFF;
tbuf[10]=(gyroz>>8)&0XFF; tbuf[11]=gyroz&0XFF;
uart1_niming_report(0XA1,tbuf,12);//自定义帧,0XA1
}

//通过串口 1 上报结算后的姿态数据给电脑
//aacx,aacy,aacz:x,y,z 三个方向上面的加速度值
//gyrox,gyroy,gyroz:x,y,z 三个方向上面的陀螺仪值
//roll:横滚角.单位 0.01 度。 -18000 -> 18000 对应 -180.00 -> 180.00 度
//pitch:俯仰角.单位 0.01 度。 -9000 - 9000 对应 -90.00 -> 90.00 度
//yaw:航向角.单位为 0.1 度 0 -> 3600 对应 0 -> 360.0 度
void usart1_report_imu(short aacx,short aacy,short aacz,short gyrox,short gyroy,
                        short gyroz,short roll,short pitch,short yaw)
{
    u8 tbuf[28];
    u8 i;
    for(i=0;i<28;i++)tbuf[i]=0;//清 0
    tbuf[0]=(aacx>>8)&0XFF; tbuf[1]=aacx&0XFF;
    tbuf[2]=(aacy>>8)&0XFF; tbuf[3]=aacy&0XFF;
    tbuf[4]=(aacz>>8)&0XFF; tbuf[5]=aacz&0XFF;
    tbuf[6]=(gyrox>>8)&0XFF; tbuf[7]=gyrox&0XFF;
    tbuf[8]=(gyroy>>8)&0XFF; tbuf[9]=gyroy&0XFF;
    tbuf[10]=(gyroz>>8)&0XFF; tbuf[11]=gyroz&0XFF;
    tbuf[18]=(roll>>8)&0XFF; tbuf[19]=roll&0XFF;
    tbuf[20]=(pitch>>8)&0XFF; tbuf[21]=pitch&0XFF;
    tbuf[22]=(yaw>>8)&0XFF; tbuf[23]=yaw&0XFF;
    uart1_niming_report(0XAF,tbuf,28);//飞控显示帧,0XAF
}

int main(void)
{
    u8 t=0,report=1; //默认开启上报
    u8 key;
    float pitch,roll,yaw; //欧拉角
    short aacx,aacy,aacz; //加速度传感器原始数据
    short gyrox,gyroy,gyroz;//陀螺仪原始数据
    short temp; //温度
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(500000); //初始化串口波特率为 500000
    LED_Init(); //初始化 LED
    KEY_Init(); //初始化按键
    LCD_Init(); //LCD 初始化
    MPU_Init(); //初始化 MPU6050
```

```
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"MPU6050 TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/9");
while(mpu_dmp_init())
{
    LCD_ShowString(30,130,200,16,16,"MPU6050 Error"); delay_ms(200);
    LCD_Fill(30,130,239,130+16,WHITE); delay_ms(200);
}
LCD_ShowString(30,130,200,16,16,"MPU6050 OK");
LCD_ShowString(30,150,200,16,16,"KEY0:UPLOAD ON/OFF");
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(30,170,200,16,16,"UPLOAD ON ");
LCD_ShowString(30,200,200,16,16," Temp: . C");
LCD_ShowString(30,220,200,16,16,"Pitch: . C");
LCD_ShowString(30,240,200,16,16," Roll: . C");
LCD_ShowString(30,260,200,16,16," Yaw : . C");
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY0_PRES)
    {
        report=!report;
        if(report)LCD_ShowString(30,170,200,16,16,"UPLOAD ON ");
        else LCD_ShowString(30,170,200,16,16,"UPLOAD OFF");
    }
    if(mpu_dmp_get_data(&pitch,&roll,&yaw)==0)
    {
        temp=MPU_Get_Temperature(); //得到温度值
        MPU_Get_Accelerometer(&aacx,&aacy,&aacz); //得到加速度传感器数据
        MPU_Get_Gyroscope(&gyrox,&gyroy,&gyroz); //得到陀螺仪数据
        if(report)mpu6050_send_data(aacx,aacy,aacz,gyrox,gyroy,gyroz);
        //用自定义帧发送加速度和陀螺仪原始数据
        if(report)uart1_report_imu(aacx,aacy,aacz,gyrox,gyroy,gyroz,(int)(roll*100),
                                    (int)(pitch*100),(int)(yaw*10));
        if((t%10)==0)
        {
            if(temp<0)
            {
                LCD_ShowChar(30+48,200,'-',16,0); //显示负号
                temp=-temp; //转为正数
            }else LCD_ShowChar(30+48,200,' ',16,0); //去掉负号
            LCD_ShowNum(30+48+8,200,temp/100,3,16); //显示整数部分
        }
    }
}
```

```

LCD_ShowNum(30+48+40,200,temp%10,1,16);      //显示小数部分
temp=pitch*10;
if(temp<0)
{
    LCD>ShowChar(30+48,220,'-',16,0);      //显示负号
    temp=-temp;      //转为正数
}else LCD>ShowChar(30+48,220,' ',16,0);      //去掉负号
LCD>ShowNum(30+48+8,220,temp/10,3,16);      //显示整数部分
LCD>ShowNum(30+48+40,220,temp%10,1,16);      //显示小数部分
temp=roll*10;
if(temp<0)
{
    LCD>ShowChar(30+48,240,'-',16,0);      //显示负号
    temp=-temp;      //转为正数
}else LCD>ShowChar(30+48,240,' ',16,0);      //去掉负号
LCD>ShowNum(30+48+8,240,temp/10,3,16);      //显示整数部分
LCD>ShowNum(30+48+40,240,temp%10,1,16);      //显示小数部分
temp=yaw*10;
if(temp<0)
{
    LCD>ShowChar(30+48,260,'-',16,0);      //显示负号
    temp=-temp;      //转为正数
}else LCD>ShowChar(30+48,260,' ',16,0);      //去掉负号
LCD>ShowNum(30+48+8,260,temp/10,3,16);      //显示整数部分
LCD>ShowNum(30+48+40,260,temp%10,1,16);      //显示小数部分
t=0; LED0=!LED0;//LED 闪烁
}
}
t++;
}
}

```

此部分代码除了 main 函数，还有几个函数，用于上报数据给上位机软件，利用上位机软件显示传感器波形，以及 3D 姿态显示，有助于更好的调试 MPU6050。上位机软件使用：ANO_Tech 匿名四轴上位机_V2.6.exe，该软件在：开发板光盘→6，软件资料→软件→匿名四轴上位机 文件夹里面可以找到，该软件的使用方法，见该文件夹下的 README.txt，这里我们不做介绍。其中，uart1_niming_report 函数用于将数据打包、计算校验和，然后上报给匿名四轴上位机软件。mpu6050_send_data 函数用于上报加速度和陀螺仪的原始数据，可用于波形显示传感器数据，通过 A1 自定义帧发送。而 usart1_report_imu 函数，则用于上报飞控显示帧，可以实时 3D 显示 MPU6050 的姿态，传感器数据等。

这里，main 函数是比较简单的，大家看代码即可，不过需要注意的是，为了高速上传数据，**这里我们将串口 1 的波特率设置为 500Kbps 了**，测试的时候要注意下。

最后，我们将 MPU_Write_Byte、MPU_Read_Byte 和 MPU_Get_Temperature 等三个函数加入 USMART 控制，这样，我们就可以通过串口调试助手，改写和读取 MPU6050 的寄存器数据

了，并可以读取温度传感器的值，方便大家调试（注意在 USMART 调试的时候，最好通过按 KEY0，先关闭数据上传功能，否则会受到很多乱码，妨碍调试）。

至此，我们的软件设计部分就结束了。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 37.4.1 所示的内容：

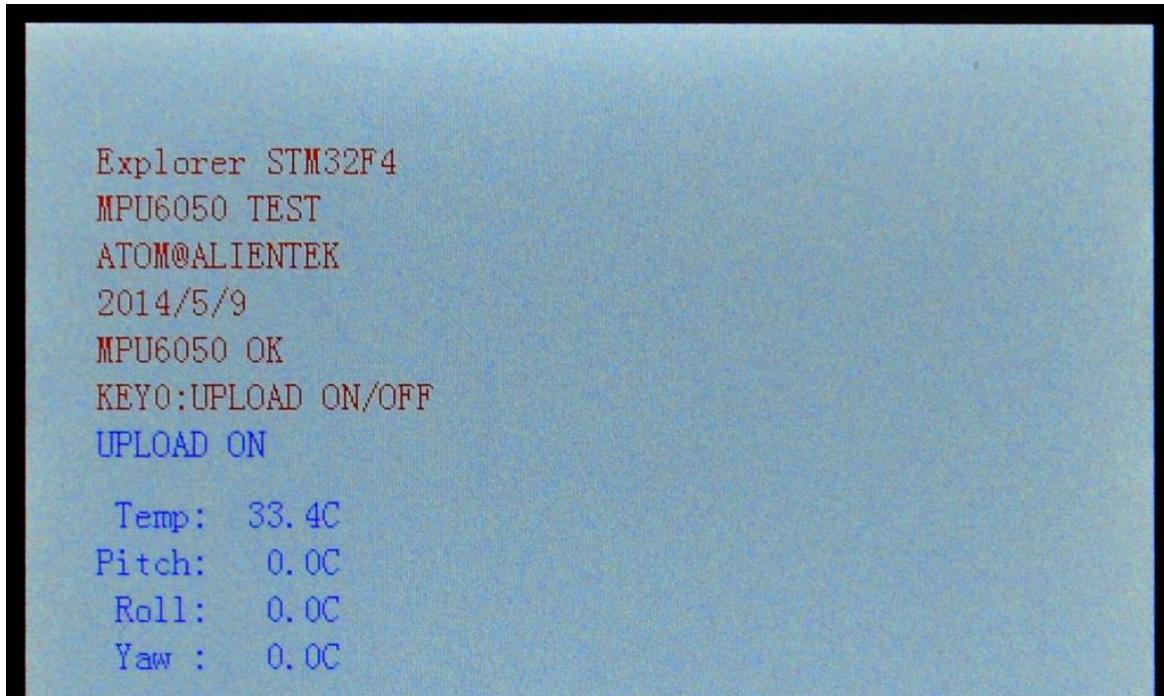


图 37.4.1 程序运行时 LCD 显示内容

屏幕显示了 MPU6050 的温度、俯仰角（pitch）、横滚角（roll）和航向角（yaw）的数值。然后，我们可以晃动开发板，看看各角度的变化。

另外，通过按 KEY0 可以开启或关闭数据上报，开启状态下，我们可以打开：ANO_Tech 匿名四轴上位机_V2.6.exe（该软件双击后，会弹出一个蓝色的小界面，直接关闭即可。然后才会进入主界面），这个软件，接收 STM32F4 上传的数据，从而图形化显示传感器数据以及飞行姿态，如图 37.4.2 和图 37.4.3 所示：



图 37.4.2 传感器数据波形显示

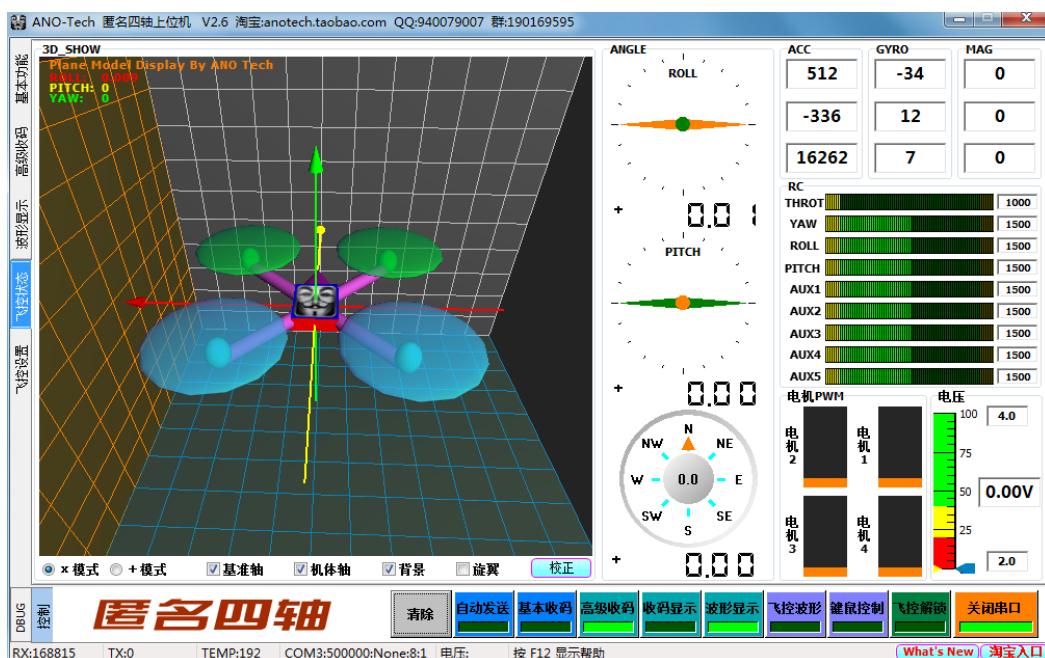


图 37.4.3 飞控状态显示

图 37.4.2 就是波形化显示我们通过 `mpu6050_send_data` 函数发送的数据,采用 A1 功能帧发送,总共 6 条线 (Series1~6) 显示波形,全部来自 A1 功能帧, int16 数据格式, Series1~6 分别代表: 加速度传感器 x/y/z 和角速度传感器 (陀螺仪) x/y/z 方向的原始数据。

图图 37.4.3 则 3D 显示了我们开发板的姿态,通过 `uart1_report_imu` 函数发送的数据显示,采用飞控显示帧格 (AF) 式上传,同时还显示了加速度陀螺仪等传感器的原始数据。

最后,我们还可以用 USMART 读写 MPU6050 的任何寄存器,来调试代码,这里我们就不做演示了,大家自己测试即可。最后,建议大家用 USMART 调试的时候,先按 KEY0 关闭数据上传功能,否则会收到很多乱码!!, 注意波特率设置为: 500Kbps (设置方法: XCOM 在关闭串口状态下,选择自定义波特率,然后输入: 500000, 再打开串口就可以了)。

第三十八章 无线通信实验

ALIENTEK 探索者 STM32F4 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 探索者 STM32F4 开发板上实现无线通信。在本章中，我们将使用两块探索者 STM32F4 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：

- 38.1 NRF24L01 无线模块简介
- 38.2 硬件设计
- 38.3 软件设计
- 38.4 下载验证

38.1 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 38.1.1 所示：

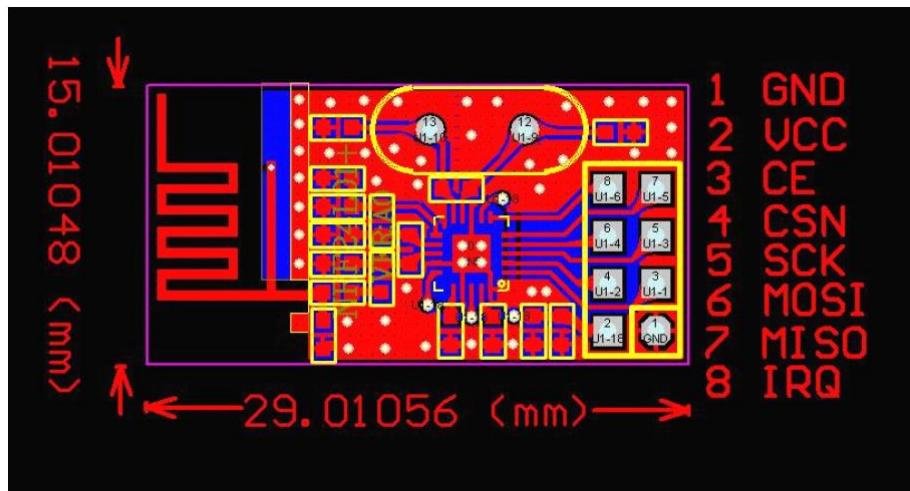


图 38.1.1 NRF24L01 无线模块外观引脚图

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。

38.2 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 模块

NRF24L01 模块属于外部模块，这里我们仅介绍开发板上 NRF24L01 模块接口和 STM32F4 的连接情况，他们的连接关系如图 38.2.1 所示：

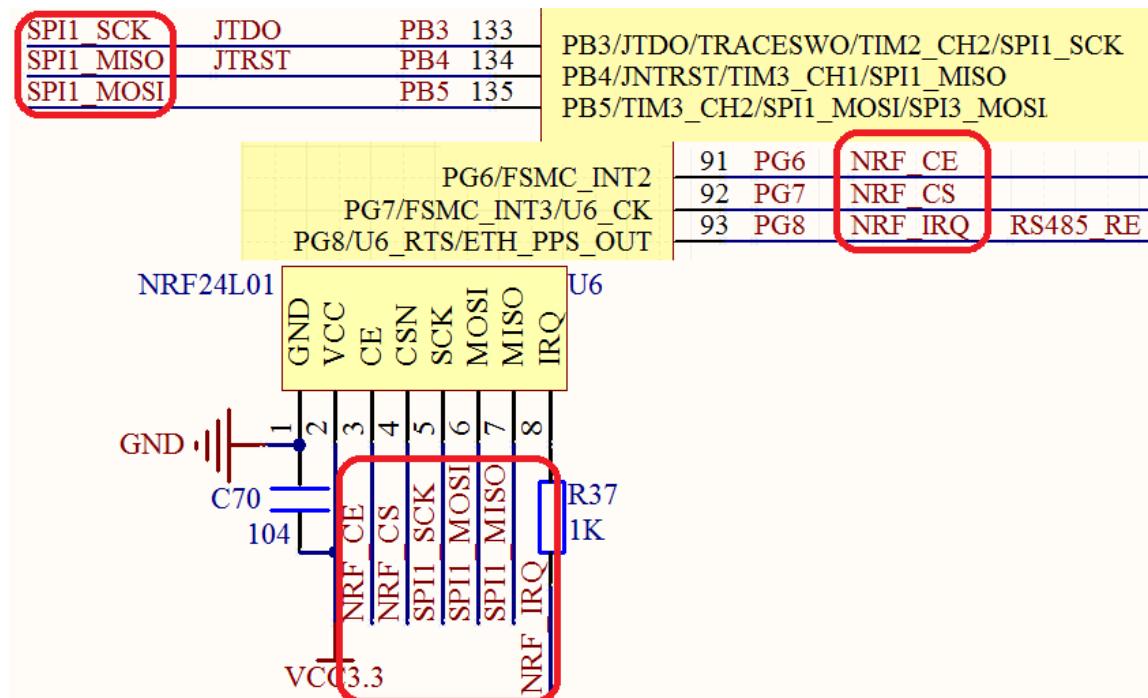


图 38.2.1 NRF24L01 模块接口与 STM32F4 连接原理图

这里NRF24L01也是使用的SPI1，和W25Q128共用一个SPI接口，所以在使用的时候，他们分时复用SPI1。本章我们需要把W25Q128的片选信号置高，以防止这个器件对NRF24L01的通信造成干扰。另外，NRF_IRQ和RS485_RE共用了PG8，所以，他们不能同时使用，不过我们一般用不到NRF_IRQ这个信号，因此，RS485和NRF一般也可以同时使用。

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探索者STM32F4开发板来向大家演示。

38.3 软件设计

打开本章实验工程可以看到，我们在工程中添加了 spi 底层驱动函数，因为 NRF24L01 是 SPI 通信接口。同时，我们增加了 24l01.c 源文件以及包含了对应的头文件用来编写 NRF24L01 底层驱动函数。

打开 24l01.c 文件，代码如下：

```
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
void NRF24L01_SPI_Init(void)
{
    SPI_InitTypeDef SPI_InitStructure;
    SPI_Cmd(SPI1, DISABLE); //失能 SPI 外设
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex; //全双工
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master; //工作模式：主 SPI
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;// 8 位帧结构
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; //空闲状态为低电平
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;//第 1 个跳变沿数据被采样
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft; //NSS 信号软件管理
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256;
    //预分频值为 256
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;//数据传输从 MSB 位开始
    SPI_InitStructure.SPI_CRCPolynomial = 7; //CRC 值计算的多项式
    SPI_Init(SPI1, &SPI_InitStructure); //初始化外设 SPIx 寄存器

    SPI_Cmd(SPI1, ENABLE); //使能 SPI 外设
}

//初始化 24L01 的 IO 口
void NRF24L01_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOG,
                           ENABLE); //使能 GPIOB, G 时钟
    //GPIOB14 初始化设置：推挽输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure); //初始化 PB14
    //GPIOG6, 7 推挽输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOG, &GPIO_InitStructure); //初始化 PG6, 7
    //GPIOG.8 上拉输入
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;//输入
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOG, &GPIO_InitStructure);//初始化 PG8

GPIO_SetBits(GPIOB, GPIO_Pin_14);//PB14 输出 1, 防止 SPI FLASH 干扰 NRF 的通信

SPI1_Init();           //初始化 SPI1
NRF24L01_SPI_Init();//针对 NRF 的特点修改 SPI 的设置
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;
    SPI1_SetSpeed(SPI_BaudRatePrescaler_8);
    NRF24L01_Write_Buf(NRF_WRITE_REG+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 =SPI1_ReadWriteByte(reg); //发送寄存器号
    SPI1_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
    SPI1_ReadWriteByte(reg); //发送寄存器号
```

```
reg_val=SPI1_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=SPI1_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]=SPI1_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 = SPI1_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++) SPI1_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;
    SPI1_SetSpeed(SPI_BaudRatePrescaler_8); //s24L01 的最大 SPI 时钟为 10Mhz
    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(NRF_WRITE_REG+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_TX, 0xff); //清除 TX 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 *rxbuf)
{
    u8 sta;
    SPI1_SetSpeed(SPI_BaudRatePrescaler_8); //24L01 的最大 SPI 时钟为 10Mhz
    sta=NRF24L01_Read_Reg(STATUS); //读取状态寄存器的值
    NRF24L01_Write_Reg(NRF_WRITE_REG+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(NRF_WRITE_REG+RX_ADDR_P0, (u8*)RX_ADDRESS, RX_ADR_WIDTH); //写 RX 节点地址
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_AA, 0x01); //使能通道 0 的自动应答
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_RXADDR, 0x01); //使能通道 0 的接收地址
    NRF24L01_Write_Reg(NRF_WRITE_REG+RF_CH, 40); //设置 RF 通信频率
    NRF24L01_Write_Reg(NRF_WRITE_REG+RX_PW_P0, RX_PLOAD_WIDTH); //选择通道 0 的有效数据宽度
    NRF24L01_Write_Reg(NRF_WRITE_REG+RF_SETUP, 0x0f);
}
```

```
//设置 TX 发射参数, 0db 增益, 2Mbps, 低噪声增益开启
NRF24L01_Write_Reg(NRF_WRITE_REG+CONFIG, 0x0f);
    //配置基本工作模式的参数;PWR_UP, EN_CRC, 16BIT_CRC, 接收模式
    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(NRF_WRITE_REG+TX_ADDR, (u8*) TX_ADDRESS, TX_ADR_WIDTH);
        //写 TX 节点地址
    NRF24L01_Write_Buf(NRF_WRITE_REG+RX_ADDR_P0, (u8*) RX_ADDRESS, RX_ADR_WIDTH);
        //设置 TX 节点地址, 主要为了使能 ACK
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_AA, 0x01);      //使能通道 0 的自动应答
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_RXADDR, 0x01); //使能通道 0 的接收地址
    NRF24L01_Write_Reg(NRF_WRITE_REG+SETUP_RETR, 0x1a);
        //设置自动重发间隔时间:500us + 86us;最大自动重发次数:10 次
    NRF24L01_Write_Reg(NRF_WRITE_REG+RF_CH, 40);        //设置 RF 通道为 40
    NRF24L01_Write_Reg(NRF_WRITE_REG+RF_SETUP, 0x0f);
        //设置 TX 发射参数, 0db 增益, 2Mbps, 低噪声增益开启
    NRF24L01_Write_Reg(NRF_WRITE_REG+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 SPI1_Init() 函数，该函数我们在第三十章曾有提到，在第三十章的设置里面，SCK 空闲时为高，但是 NRF24L01 的 SPI 通信时序如图 38.3.1 所示：

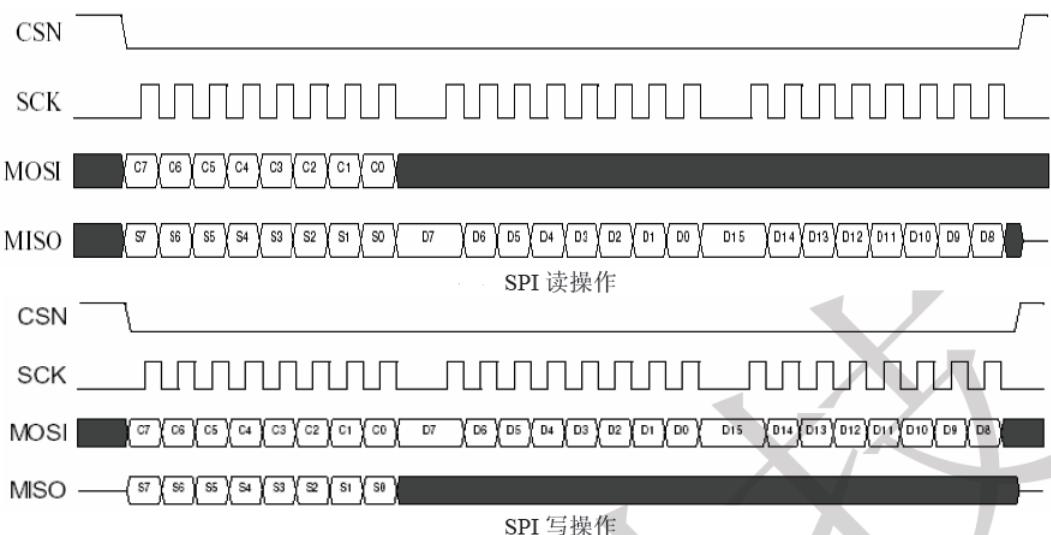


图 38.3.1 NRF24L01 读写操作时序

上图中 C_n 代表指令位, S_n 代表状态寄存器位, D_n 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 的函数 NRF24L01_SPI_Init。这里主要是修改了下面两行代码;

```
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low; //空闲状态为低电平
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;//第 1 个跳变沿数据被采样
```

接下来我们看看 24101.h 头文件部分内容:

```
#ifndef __24L01_H
#define __24L01_H
#include "sys.h"

//NRF24L01 寄存器操作命令
#define READ_REG      0x00          //读配置寄存器,低 5 位为寄存器地址
.....//省略部分定义
#define FIFO_STATUS   0x17          //FIFO 状态寄存器;bit0,RX FIFO 寄存器空标志;
//bit1,RX FIFO 满标志;bit2,3,保留 bit4,TX FIFO 空标志;bit5,TX FIFO 满标志;
//bit6,1, 循环发送上一数据包.0,不循环;
//24L01 操作线
#define NRF24L01_CE    PGout(6)    //24L01 片选信号
#define NRF24L01_CSN   PGout(7)    //SPI 片选信号
#define NRF24L01_IRQ    PGin(8)    //IRQ 主机数据输入

//24L01 发送接收数据宽度定义
#define TX_ADR_WIDTH   5           //5 字节的地址宽度
#define RX_ADR_WIDTH   5           //5 字节的地址宽度
#define TX_PLOAD_WIDTH 32          //32 字节的用户数据宽度
#define RX_PLOAD_WIDTH 32          //32 字节的用户数据宽度

void NRF24L01_Init(void); //初始化
.....//省略部分函数申明
u8 NRF24L01_RxPacket(u8 *rxbuf); //接收一个包的数据
```

```
#endif
```

部分代码，主要定义了一些 24L01 的命令字（这里我们省略了一部分），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：

```
//要写入到 W25Q16 的字符串数组
const u8 TEXT_Buffer[]={"Explorer STM32F4 SPI TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key,mode, tmp_buf[33];
    u16 t=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    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(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"NRF24L01 TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/9");
    while(NRF24L01_Check())
    {
        LCD_ShowString(30,130,200,16,16,"NRF24L01 Error");
        delay_ms(200);
        LCD_Fill(30,130,239,130+16,WHITE);
        delay_ms(200);
    }
    LCD_ShowString(30,130,200,16,16,"NRF24L01 OK");
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY0_PRES)
        {
            mode=0; break;
        }else if(key==KEY1_PRES)
        {
            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(30,150,200,16,16,"NRF24L01 RX_Mode");
    LCD_ShowString(30,170,200,16,16,"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	lcddev.width-1,32,16,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(30,150,200,16,16,"NRF24L01 TX_Mode");
    NRF24L01_TX_Mode();
    mode=' ';//从空格键开始
    while(1)
    {
        if(NRF24L01_TxPacket(tmp_buf)==TX_OK)
        {
            LCD_ShowString(30,170,239,32,16,"Sended DATA:");
            LCD_ShowString(0,190	lcddev.width-1,32,16,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_Fill(0,170	lcddev.width,170+16*3,WHITE);//清空显示
    LCD_ShowString(30,170	lcddev.width-1,32,16,"Send Failed ");
}
LED0=!LED0;delay_ms(1500);
};
}
}
```

以上代码，我们就实现了 38.2 节所介绍的功能，程序运行时先通过 NRF24L01_Check 函数检测 NRF24L01 是否存在，如果存在，则让用户选择发送模式 (KEY1) 还是接收模式 (KEY0)，在确定模式之后，设置 NRF24L01 的工作模式，然后执行相应的数据发送/接收处理。

至此，我们整个实验的软件设计就完成了。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 38.4.1 所示的内容（默认 NRF24L01 已经接上了）：

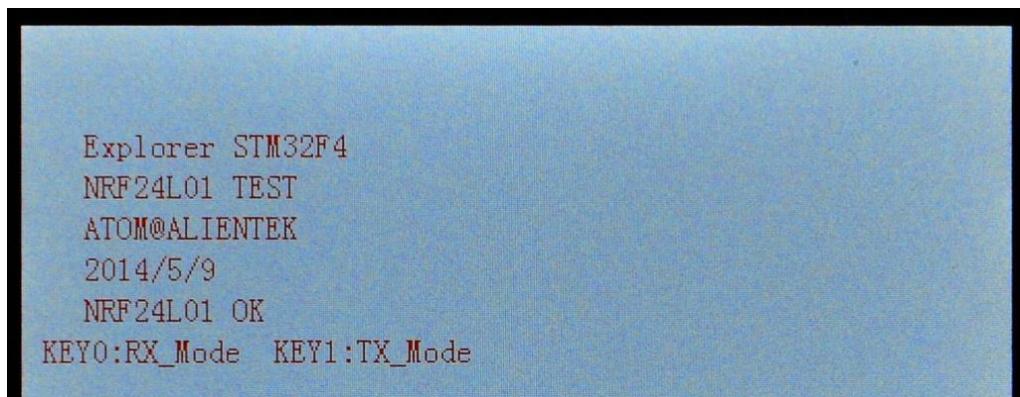


图 38.4.1 选择工作模式界面

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 38.4.2 和图 38.4.3 所示：

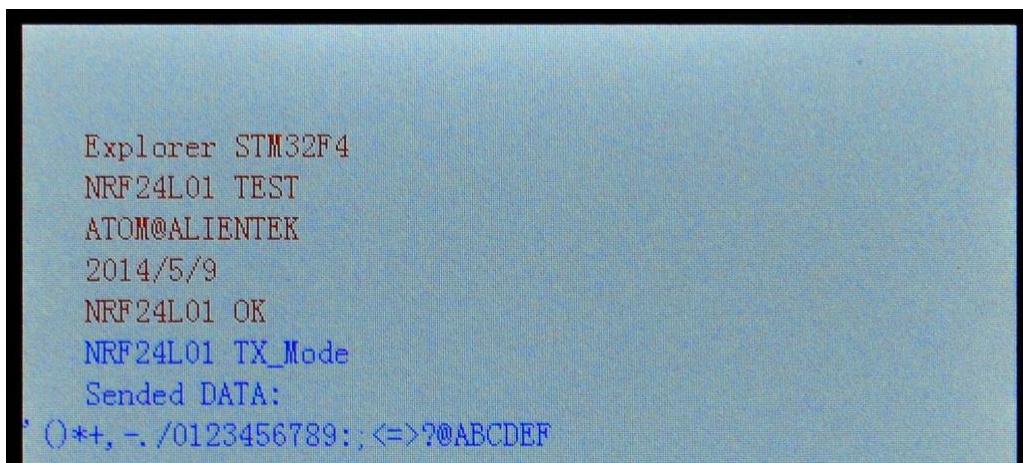


图 38.4.2 开发板 A 发送数据

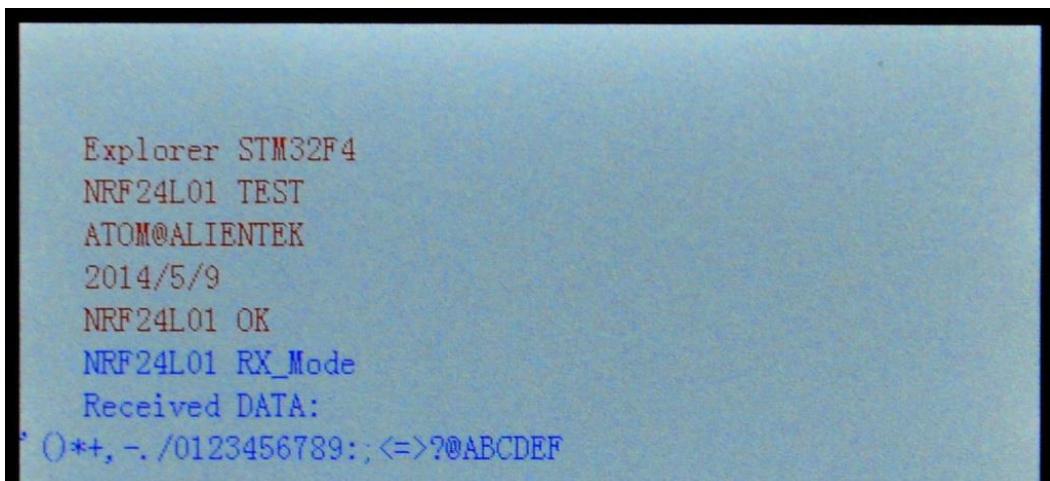


图 38.4.3 开发板 B 接收数据

图 38.4.2 来自开发板 A，工作在发送模式。图 38.4.3 来自开发板 B，工作在接收模式，A 发送，B 接收。可以看到收发数据是一致的，说明实验成功。

第三十九章 FLASH 模拟 EEPROM 实验

STM32F4 本身没有自带 EEPROM，但是 STM32F4 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 STM32F4 内部的 FLASH 来实现第三十章实验类似的效果，不过这次我们是将数据直接存放在 STM32F4 内部，而不是存放在 W25Q128。本章分为如下几个部分：

39.1 STM32F4 FLASH 简介

39.2 硬件设计

39.3 软件设计

39.4 下载验证

39.1 STM32F4 FLASH 简介

不同型号的 STM32F40xx/41xx，其 FLASH 容量也有所不同，最小的只有 128K 字节，最大的则达到了 1024K 字节。探索者 STM32F4 开发板选择的 STM32F407ZGT6 的 FLASH 容量为 1024K 字节，STM32F40xx/41xx 的闪存模块组织如图 39.1.1 所示：

块	名称	块基址	大小
主存储器	扇区 0	0x0800 0000 - 0x0800 3FFF	16 KB
	扇区 1	0x0800 4000 - 0x0800 7FFF	16 KB
	扇区 2	0x0800 8000 - 0x0800 BFFF	16 KB
	扇区 3	0x0800 C000 - 0x0800 FFFF	16 KB
	扇区 4	0x0801 0000 - 0x0801 FFFF	64 KB
	扇区 5	0x0802 0000 - 0x0803 FFFF	128 KB
	扇区 6	0x0804 0000 - 0x0805 FFFF	128 KB
	.	.	.
	.	.	.
	.	.	.
	扇区 11	0x080E 0000 - 0x080F FFFF	128 KB
	系统存储器	0x1FFF 0000 - 0x1FFF 77FF	30 KB
OTP 区域	OTP 区域	0x1FFF 7800 - 0x1FFF 7A0F	528 字节
	选项字节	0x1FFF C000 - 0x1FFF C00F	16 字节

图 39.1.1 大容量产品闪存模块组织

STM32F4 的闪存模块由：主存储器、系统存储器、OTP 区域和选项字节等 4 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）。分为 12 个扇区，前 4 个扇区为 16KB 大小，然后扇区 4 是 64KB 大小，扇区 5~11 是 128K 大小，不同容量的 STM32F4，拥有的扇区数不一样，比如我们的 STM32F407ZGT6，则拥有全部 12 个扇区。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。

系统存储器，这个主要用来存放 STM32F4 的 bootloader 代码，此代码是出厂的时候就固化在 STM32F4 里面了，专门来给主存储器下载代码的。当 B0 接 V3.3，B1 接 GND 的时候，从

该存储器启动（即进入串口下载模式）。

OTP 区域，即一次性可编程区域，共 528 字节，被分成两个部分，前面 512 字节（32 字节为 1 块，分成 16 块），可以用来存储一些用户数据（一次性的，写完一次，永远不可以擦除!!），后面 16 字节，用于锁定对应块。

选项字节，用于配置读保护、BOR 级别、软件/硬件看门狗以及器件处于待机或停止模式下的复位。

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。

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

STM32F4 可通过内部的 I-Code 指令总线或 D-Code 数据总线访问内置闪存模块，本章我们主要讲解数据读写，即通过 D-Code 数据总线来访问内部闪存模块。为了准确读取 Flash 数据，必须根据 CPU 时钟 (HCLK) 频率和器件电源电压在 Flash 存取控制寄存器 (FLASH_ACR) 中正确地设置等待周期数 (LATENCY)。当电源电压低于 2.1V 时，必须关闭预取缓冲器。Flash 等待周期与 CPU 时钟频率之间的对应关系，如表 39.1.1 所示：

等待周期 (WS) (LATENCY)	HCLK (MHz)			
	电压范围 2.7 V - 3.6 V	电压范围 2.4 V - 2.7 V	电压范围 2.1 V - 2.4 V	电压范围 1.8 V - 2.1 V 预取关闭
0 WS (1 个 CPU 周期)	0 < HCLK ≤ 30	0 < HCLK ≤ 24	0 < HCLK ≤ 22	0 < HCLK ≤ 20
1 WS (2 个 CPU 周期)	30 < HCLK ≤ 60	24 < HCLK ≤ 48	22 < HCLK ≤ 44	20 < HCLK ≤ 40
2 WS (3 个 CPU 周期)	60 < HCLK ≤ 90	48 < HCLK ≤ 72	44 < HCLK ≤ 66	40 < HCLK ≤ 60
3 WS (4 个 CPU 周期)	90 < HCLK ≤ 120	72 < HCLK ≤ 96	66 < HCLK ≤ 88	60 < HCLK ≤ 80
4 WS (5 个 CPU 周期)	120 < HCLK ≤ 150	96 < HCLK ≤ 120	88 < HCLK ≤ 110	80 < HCLK ≤ 100
5 WS (6 个 CPU 周期)	150 < HCLK ≤ 168	120 < HCLK ≤ 144	110 < HCLK ≤ 132	100 < HCLK ≤ 120
6 WS (7 个 CPU 周期)		144 < HCLK ≤ 168	132 < HCLK ≤ 154	120 < HCLK ≤ 140
7 WS (8 个 CPU 周期)			154 < HCLK ≤ 168	140 < HCLK ≤ 160

表 39.1.1 CPU 时钟 (HCLK) 频率对应的 FLASH 等待周期表

等待周期通过 FLASH_ACR 寄存器的 LATENCY[2:0]三个位设置。系统复位后，CPU 时钟频率为内部 16M RC 振荡器，LATENCY 默认是 0，即 1 个等待周期。供电电压，我们一般是 3.3V，所以，在我们设置 168Mhz 频率作为 CPU 时钟之前，必须先设置 LATENCY 为 5，否则 FLASH 读写可能出错，导致死机。

正常工作时 (168Mhz)，虽然 FLASH 需要 6 个 CPU 等待周期，但是由于 STM32F4 具有自适应实时存储器加速器(ART Accelerator)，通过指令缓存存储器，预取指令，实现相当于 0 FLASH 等待的运行速度。关于自适应实时存储器加速器的详细介绍，请大家参考《STM32F4xx 中文参考手册》3.4.2 节。

STM32F4 的 FLASH 读取是很简单的。例如，我们要从地址 addr，读取一个字（字节为 8

位，半字为 16 位，字为 32 位），可以通过如下的语句读取：

```
data=*(vu32*)addr;
```

将 addr 强制转换为 vu32 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 vu32 改为 vu16，即可读取指定地址的一个半字。相对 FLASH 读取来说，STM32F4 FLASH 的写就复杂一点了，下面我们介绍 STM32F4 闪存的编程和擦除。

闪存的编程和擦除

执行任何 Flash 编程操作（擦除或编程）时，CPU 时钟频率 (HCLK)不能低于 1 MHz。如果在 Flash 操作期间发生器件复位，无法保证 Flash 中的内容。

在对 STM32F4 的 Flash 执行写入或擦除操作期间，任何读取 Flash 的尝试都会导致总线阻塞。只有在完成编程操作后，才能正确处理读操作。这意味着，写/擦除操作进行期间不能从 Flash 中执行代码或数据获取操作。

STM32F4 的闪存编程由 6 个 32 位寄存器控制，他们分别是：

- FLASH 访问控制寄存器(FLASH_ACR)
- FLASH 秘钥寄存器(FLASH_KEYR)
- FLASH 选项秘钥寄存器(FLASH_OPTKEYR)
- FLASH 状态寄存器(FLASH_SR)
- FLASH 控制寄存器(FLASH_CR)
- FLASH 选项控制寄存器(FLASH_OPTCR)

STM32F4 复位后，FLASH 编程操作是被保护的，不能写入 FLASH_CR 寄存器；通过写入特定的序列（0X45670123 和 0XCDEF89AB）到 FLASH_KEYR 寄存器才可解除写保护，只有在写保护被解除后，我们才能操作相关寄存器。

FLASH_CR 的解锁序列为：

- 1, 写 0X45670123 到 FLASH_KEYR
- 2, 写 0XCDEF89AB 到 FLASH_KEYR

通过这两个步骤，即可解锁 FLASH_CR，如果写入错误，那么 FLASH_CR 将被锁定，直到下次复位后才可以再次解锁。

STM32F4 闪存的编程位数可以通过 FLASH_CR 的 PSIZE 字段配置，PSIZE 的设置必须和电源电压匹配，见表：39.1.2：

	电压范围 2.7 - 3.6 V (使用外部 V_{PP})	电压范围 2.7 - 3.6 V	电压范围 2.4 - 2.7 V	电压范围 2.1 - 2.4 V	电压范围 1.8 V - 2.1 V
并行位数	x64	x32	x16	x8	
PSIZE(1:0)	11	10	01	00	

表 39.1.2 编程/擦除并行位数与电压关系表

由于我们开发板用的电压是 3.3V，所以 PSIZE 必须设置为 10，即 32 位并行位数。擦除或者编程，都必须以 32 位为基础进行。

STM32F4 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 0xFFFFFFFF），否则无法写入。STM32F4 的标准编程步骤如下：

- 1, 检查 FLASH_SR 中的 BSY 位，确保当前未执行任何 FLASH 操作。
- 2, 将 FLASH_CR 寄存器中的 PG 位置 1，激活 FLASH 编程。
- 3, 针对所需存储器地址（主存储器块或 OTP 区域内）执行数据写入操作：
 - 并行位数为 x8 时按字节写入 (PSIZE=00)
 - 并行位数为 x16 时按半字写入 (PSIZE=01)

- 并行位数为 x32 时按字写入 (PSIZE=02)
 - 并行位数为 x64 时按双字写入 (PSIZE=03)
- 4, 等待 BSY 位清零, 完成一次编程。

按以上四步操作, 就可以完成一次 FLASH 编程。不过有几点要注意: 1, 编程前, 要确保要写如地址的 FLASH 已经擦除。2, 要先解锁 (否则不能操作 FLASH_CR)。3, 编程操作对 OPT 区域也有效, 方法一模一样。

我们在 STM32F4 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 STM32F4 的闪存擦除, STM32F4 的闪存擦除分为两种: 扇区擦除和整片擦除。

扇区擦除步骤如下:

- 1, 检查 FLASH_CR 的 LOCK 是否解锁, 如果没有则先解锁
- 2, 检查 FLASH_SR 寄存器中的 BSY 位, 确保当前未执行任何 FLASH 操作
- 3, 在 FLASH_CR 寄存器中, 将 SER 位置 1, 并从主存储块的 12 个扇区中选择要擦除的扇区 (SNB)
- 4, 将 FLASH_CR 寄存器中的 STRT 位置 1, 触发擦除操作
- 5, 等待 BSY 位清零

经过以上五步, 就可以擦除某个扇区。本章, 我们只用到了 STM32F4 的扇区擦除功能, 整片擦除功能我们在这里就不介绍了, 想了解的朋友可以看《STM32F4xx 中文参考手册》第 3.5.3 节。

通过以上了解, 我们基本上知道了 STM32F4 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 FLASH 访问控制寄存器: FLASH_ACR。该寄存器各位描述如图 39.1.2 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	DCRST	ICRST	DCEN	ICEN	PRFTEN		Reserved		LATENCY						
	rw	w	rw	rw	rw				rw	rw	rw				

位 31:11 保留，必须保持清零。

位 12 **DCRST**: 数据缓存复位 (Data cache reset)

- 0: 数据缓存不复位
- 1: 数据缓存复位

只有在关闭数据缓存时才能在该位中写入值。

位 11 **ICRST**: 指令缓存复位 (Instruction cache reset)

- 0: 指令缓存不复位
- 1: 指令缓存复位

只有在关闭指令缓存时才能在该位中写入值。

位 10 **DCEN**: 数据缓存使能 (Data cache enable)

- 0: 关闭数据缓存
- 1: 使能数据缓存

位 9 **ICEN**: 指令缓存使能 (Instruction cache enable)

- 0: 关闭指令缓存
- 1: 使能指令缓存

位 8 **PRFTEN**: 预取使能 (Prefetch enable)

- 0: 关闭预取
- 1: 使能预取

位 7:3 保留，必须保持清零。

位 2:0 **LATENCY**: 延迟 (Latency)

这些位表示 CPU 时钟周期与 Flash 访问时间之比。

- | | |
|-------------|-------------|
| 000: 零等待周期 | 100: 四个等待周期 |
| 001: 一个等待周期 | 101: 五个等待周期 |
| 010: 两个等待周期 | 110: 六个等待周期 |
| 011: 三个等待周期 | 111: 七个等待周期 |

图 39.1.2 FLASH_ACR 寄存器各位描述

这里，我们重点看 LATENCY[2:0]这三个位，这三个位，必须根据我们 MCU 的工作电压和频率，来进行正确的设置，否则，可能死机，设置规则见表 39.1.1。其他 DCEN、ICEN 和 PRFTEN 这三个位也比较重要，为了达到最佳性能，这三个位我们一般都设置为 1 即可。

第二个介绍的是 FLASH 秘钥寄存器:FLASH_KEYR。该寄存器各位描述如图 39.1.3 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEY[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
KEY[15:0]															
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

位 31:0 **FKEYR**: FPEC 密钥 (FPEC key)

要将 FLASH_CR 寄存器解锁并允许对其执行编程/擦除操作，必须顺序编程以下值：

- a) KEY1 = 0x45670123
- b) KEY2 = 0xCDEF89AB

图 39.1.3 FLASH_KEYR 寄存器各位描述

该寄存器主要用来解锁 FLASH_CR，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。

第三个要介绍的是 FLASH 控制寄存器: FLASH_CR。该寄存器的各位描述如图 39.1.4 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LOCK rs	Reserved			ERRIE	EOPIE	Reserved			Reserved			STRT rs			
	rw	rw													
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			PSIZE[1:0]		Res.	SNB[3:0]			MER	SER	PG				
			rw	rw		rw	rw	rw	rw	rw	rw				

图 39.1.4 FLASH_CR 寄存器各位描述

该寄存器我们本章只用到了它的 LOCK、STRT、PSIZE[1:0]、SNB[3:0]、SER 和 PG 等位。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。

PSIZE[1:0]位，用于设置编程宽度，3.3V 时，我们设置 PSIZE =2 即可。

SNB[3:0]位，这 4 个位用于选择要擦除的扇区编号，取值范围为 0~11。

SER 位，该位用于选择扇区擦除操作，在扇区擦除的时候，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《STM32F4xx 中文参考手册》第 3.8.5 节。

最后要介绍的是 FLASH 状态寄存器：FLASH_SR。该寄存器各位描述如图 39.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved												BSY			
												r			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			PGSERR		PGPERR	PGAERR	WRPERR	Reserved	OPERR		EOP				
			rc_w1	rc_w1	rc_w1	rc_w1	rc_w1								

图 39.1.5 FLASH_SR 寄存器各位描述

该寄存器我们主要用了其 BSY 位，当该位为 1 时，表示正在执行 FLASH 操作。当该位为 0 时，表示当前未执行任何 FLASH 操作。

关于 STM32F4 FLASH 的介绍，我们就介绍到这。更详细的介绍，请参考《STM32F4xx 中文参考手册》第三章。下面我们讲解使用 STM32F4 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 stm32f4xx_flash.c 以及 stm32f4xx_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

固件库提供了四个 FLASH 写函数：

```
FLASH_Status FLASH_ProgramDoubleWord(uint32_t Address, uint64_t Data);
FLASH_Status FLASH_ProgramWord(uint32_t Address, uint32_t Data);
FLASH_Status FLASH_ProgramHalfWord(uint32_t Address, uint16_t Data);
FLASH_Status FLASH_ProgramByte(uint32_t Address, uint8_t Data);
```

这几个函数从名字上面还是比较好理解意思，分别为写入双字，字，半字，字节的函数。这些函数的内部实现过程，实际就是按照我们 39.1 讲解的编程步骤来实现的。有兴趣的同学可以进入函数体看看，这样会加深理解。

3) 擦除函数

固件库提供四个 FLASH 擦除函数：

```
FLASH_Status FLASH_EraseSector(uint32_t FLASH_Sector, uint8_t VoltageRange);
FLASH_Status FLASH_EraseAllSectors(uint8_t VoltageRange);
FLASH_Status FLASH_EraseAllBank1Sectors(uint8_t VoltageRange);
FLASH_Status FLASH_EraseAllBank2Sectors(uint8_t VoltageRange);
```

对于前面两个函数比较好理解，一个是用来擦除某个 Sector，一个使用来擦除全部的 sectors。对于第三个和第四个函数，这里的话主要是针对 STM32F42X 系列和 STM32F43X 系列芯片而言的，因为它们将所有的 sectors 分为两个 bank。所以这两个函数用来擦除 2 个 bank 下的 sectors 的。第一个参数取值范围在固件库有相关宏定义标识符已经定义好，为 FLASH_Sector_0~FLASH_Sector_11（对于我们使用的 STM32F407 最大是 FLASH_Sector_11），对于这些函数的第二个参数，我们这里电源电压范围是 3.3V，所以选择 VoltageRange_3 即可。

4) 获取 FLASH 状态

获取 FLASH 状态主要调用的函数是：

```
FLASH_Status FLASH_GetStatus(void);
```

返回值是通过枚举类型定义的：

```
typedef enum
{
    FLASH_BUSY = 1, //操作忙
    FLASH_ERROR_RD, //读保护错误
    FLASH_ERROR_PGS, //编程顺序错误
    FLASH_ERROR_PGP, //编程并行位数错误
    FLASH_ERROR_PGA, //编程对齐错误
    FLASH_ERROR_WRP, //写保护错误
    FLASH_ERROR_PROGRAM, //编程错误
    FLASH_ERROR_OPERATION, //操作错误
    FLASH_COMPLETE //操作结束
}FLASH_Status;
```

从这里面我们可以看到 FLASH 操作的几个状态。

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(void)
```

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 指定地址的数据的函数固件库并没有给出来，这里我们提供从指定地址一个读取一个字的函数：

```
u32 STMFLASH_ReadWord(u32 faddr)
{
    return *(vu32*)faddr;
}
```

7) 写选项字节操作

固件库还提供了一些列选项字节区域操作函数，这里因为本实验没有用到选项字节区域操作，这里我们就不做过多讲解，有兴趣的同学可以了解一下。

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（KEY1）用来执行写入 FLASH 的操作，另外一个按键（KEY0）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) KEY1 和 KEY0 按键
- 3) TFTLCD 模块
- 4) STM32F4 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 stm32f4xx_flash.c 和头文件 stm32f4xx_flash.h。

打开 stmflash.c 文件，代码如下：

```
//读取指定地址的半字(16位数据)
//faddr:读地址
//返回值:对应数据.
u32 STMFLASH_ReadWord(u32 faddr)
{
    return *(vu32*)faddr;
}

//获取某个地址所在的 flash 扇区
//addr:flash 地址
//返回值:0~11,即 addr 所在的扇区
uint16_t STMFLASH_GetFlashSector(u32 addr)
{
    if(addr<ADDR_FLASH_SECTOR_1)return FLASH_Sector_0;
    else if(addr<ADDR_FLASH_SECTOR_2)return FLASH_Sector_1;
    else if(addr<ADDR_FLASH_SECTOR_3)return FLASH_Sector_2;
    else if(addr<ADDR_FLASH_SECTOR_4)return FLASH_Sector_3;
    else if(addr<ADDR_FLASH_SECTOR_5)return FLASH_Sector_4;
    else if(addr<ADDR_FLASH_SECTOR_6)return FLASH_Sector_5;
    else if(addr<ADDR_FLASH_SECTOR_7)return FLASH_Sector_6;
    else if(addr<ADDR_FLASH_SECTOR_8)return FLASH_Sector_7;
    else if(addr<ADDR_FLASH_SECTOR_9)return FLASH_Sector_8;
    else if(addr<ADDR_FLASH_SECTOR_10)return FLASH_Sector_9;
```

```
else if(addr<ADDR_FLASH_SECTOR_11) return FLASH_Sector_10;
return FLASH_Sector_11;
}

//从指定地址开始写入指定长度的数据
//特别注意:因为 STM32F4 的扇区实在太大,没办法本地保存扇区数据,所以本函数
// 写地址如果非 0xFF,那么会先擦除整个扇区且不保存扇区数据.所以
// 写非 0xFF 的地址,将导致整个扇区数据丢失.建议写之前确保扇区里
// 没有重要数据,最好是整个扇区先擦除了,然后慢慢往后写.
//该函数对 OTP 区域也有效!可以用来写 OTP 区!
//OTP 区域地址范围:0X1FFF7800~0X1FFF7A0F
//WriteAddr:起始地址(此地址必须为 4 的倍数!!)
//pBuffer:数据指针
//NumToWrite:字(32 位)数(就是要写入的 32 位数据的个数.)
void STMFLASH_Write(u32 WriteAddr,u32 *pBuffer,u32 NumToWrite)
{
    FLASH_Status status = FLASH_COMPLETE;
    u32 addrx=0;
    u32 endaddr=0;
    if(WriteAddr<STM32_FLASH_BASE||WriteAddr%4) return; //非法地址
    FLASH_Unlock(); //解锁
    FLASH_DataCacheCmd(DISABLE); //FLASH 擦除期间,必须禁止数据缓存

    addrx=WriteAddr; //写入的起始地址
    endaddr=WriteAddr+NumToWrite*4; //写入的结束地址
    if(addrx<0X1FFF0000) //只有主存储区,才需要执行擦除操作!!
    {
        while(addrx<endaddr) //扫清一切障碍.(对非 FFFFFFFF 的地方,先擦除)
        {
            if(STMFLASH_ReadWord(addrx)!=0xFFFFFFFF)
                //有非 0xFFFFFFFF 的地方,要擦除这个扇区
            {
                status=FLASH_EraseSector(STMFLASH_GetFlashSector(addrx),VoltageRange_3);
                //VCC=2.7~3.6V 之间!!
                if(status!=FLASH_COMPLETE) break; //发生错误了
            }else addrx+=4;
        }
    }
    if(status==FLASH_COMPLETE)
    {
        while(WriteAddr<endaddr)//写数据
        {
            if(FLASH_ProgramWord(WriteAddr,*pBuffer)!=FLASH_COMPLETE)//写入数据
            {

```

```
        break; //写入异常
    }
    WriteAddr+=4; pBuffer++;
}
}

FLASH_DataCacheCmd(ENABLE); //FLASH 擦除结束,开启数据缓存
FLASH_Lock();//上锁
}

//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:字(4 位)数
void STMFLASH_Read(u32 ReadAddr,u32 *pBuffer,u32 NumToRead)
{
    u32 i;
    for(i=0;i<NumToRead;i++)
    {
        pBuffer[i]=STMFLASH_ReadWord(ReadAddr);//读取 4 个字节.
        ReadAddr+=4;//偏移 4 个字节.
    }
}

///////////////////////////////测试用/////////////////////////////
//WriteAddr:起始地址
//WriteData:要写入的数据
void Test_Write(u32 WriteAddr,u32 WriteData)
{
    STMFLASH_Write(WriteAddr,&WriteData,1); //写入一个字
}
```

该部分代码，我们重点介绍一下 STMFLASH_Write 函数，该函数用于在 STM32F4 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 30 章的 W25QXX_Flash_Write 函数，不过该函数使用的时候，有几个要注意要注意：

- 1, 写入地址必须是用户代码区以外的地址。
- 2, 写入地址必须是 4 的倍数。

第 1 点比较好理解，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。不过，因为 STM32F4 的扇区都比较大（最少 16K，大的 128K），所以本函数不缓存要擦除的扇区内容，也就是如果要擦除，那么就是整个扇区擦除，所以建议大家使用该函数的时候，写入地址定位到用户代码占用扇区以外的扇区，比较保险。

第 2 点则是每次必须写入 32 位，即 4 字节，所以地址必须是 4 的倍数。

关于 STMFLASH_GetFlashSector 函数，这个就比较好理解了，根据地址确定其 sector 编号。其他函数我们就不做介绍了。

对于头文件 stmflash.h，这里面有一点提一下，就是我们定义了从 ADDR_FLASH_SECTOR_0~ADDR_FLASH_SECTOR_11 等一系列宏定义标识符，实际上这些

标识符的值就是对应的 sector 的起始地址值，相信也比较好理解。

最后我们打开 main.c 文件，代码如下：

```
//要写入到 STM32 FLASH 的字符串数组
const u8 TEXT_Buffer[]={"STM32 FLASH TEST"};
#define TEXT_LENGTH sizeof(TEXT_Buffer) //数组长度
#define SIZE TEXT_LENGTH/4+((TEXT_LENGTH%4)?1:0)

/*设置 FLASH 保存地址(必须为偶数，且所在扇区,要大于本代码所占用到的扇区.否则,
 *写操作的时候,可能会导致擦除整个扇区,从而引起部分程序丢失.引起死机.*/
#define FLASH_SAVE_ADDR 0X0800C004

int main(void)
{
    u8 key=0, datatemp[SIZE];
    u16 i=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    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(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"FLASH EEPROM TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/9");
    LCD_ShowString(30,130,200,16,16,"KEY1:Write KEY0:Read");
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY1_PRES) //KEY1 按下,写入 STM32 FLASH
        {
            LCD_Fill(0,170,239,319,WHITE);//清除半屏
            LCD_ShowString(30,170,200,16,16,"Start Write FLASH....");
            STMFLASH_Write(FLASH_SAVE_ADDR,(u32*)TEXT_Buffer,SIZE);
            LCD_ShowString(30,170,200,16,16,"FLASH Write Finished");//提示传送完成
        }
        if(key==KEY0_PRES) //KEY0 按下,读取字符串并显示
        {
            LCD_ShowString(30,170,200,16,16,"Start Read FLASH.... ");
            STMFLASH_Read(FLASH_SAVE_ADDR,(u32*)datatemp,SIZE);
            LCD_ShowString(30,170,200,16,16,"The Data Readed Is: ");
            LCD_ShowString(30,190,200,16,16,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;
}
}
```

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 探索者 STM32F4 开发板上，伴随 DS0 的不停闪烁，提示程序在运行，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 39.4.1 所示：

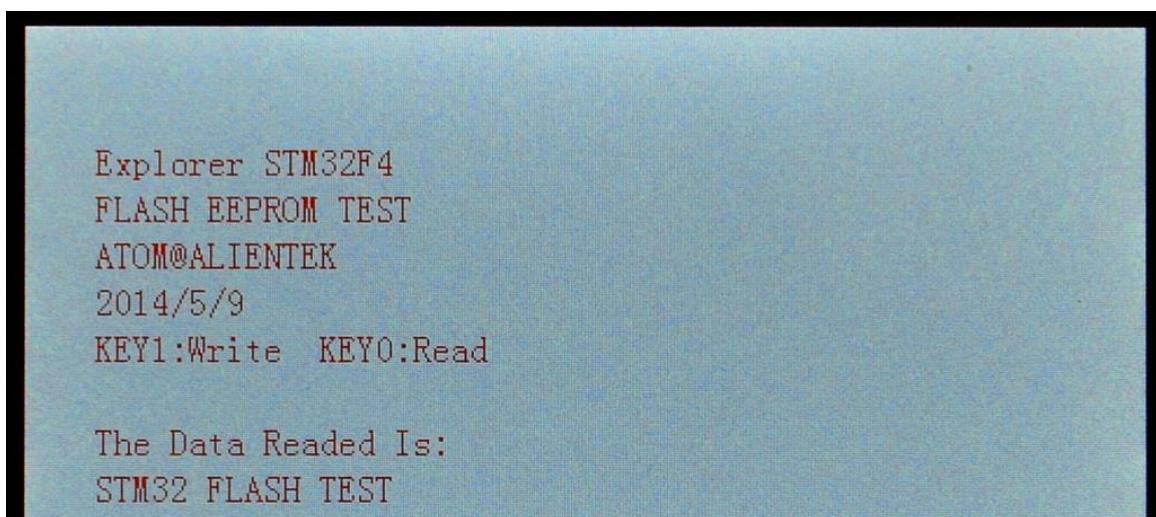


图 39.4.1 程序运行效果图

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，调用：TMFLASH_ReadWord 和 Test_Write 函数，大家可以测试下 OTP 区域的读写，注意：OTP 区域，最后 16 字节，不要乱写！！是用于锁定 OTP 数据块的的！！

另外，OTP 的一次性可编程，也并不像字面意思那样，只能写一次。而是要理解成：只能写 0，不能写 1。举个例子，你在地址：0X1FFF7808，第一次写入 0X12345678。读出来，发现是对的，和你写入的一样。而当你在这个地址，再次写入：0X12345673 的时候，再读出来，变成了：0X12345670，不是第一次写入的值，也不是第二次写入的值，而是两次写入值相与的值，说明第二次也发生了写操作。所以，要理解成：只能写 0，不能写 1。

第四十章 摄像头实验

ALIENTEK 探索者 STM32F4 开发板具有 DCMI 接口，并板载了一个摄像头接口（P8），该接口可以用来连接 ALIENTEK OV2640 等摄像头模块。本章，我们将使用 STM32 驱动 ALIENTEK OV2640 摄像头模块，实现摄像头功能。本章分为如下几个部分：

40.1 OV2640&DCMI 简介

40.2 硬件设计

40.3 软件设计

40.4 下载验证

40.1 OV2640&DCMI 简介

本节将分为两个部分，分别介绍 OV2640 和 STM32F4 的 DCMI 接口。

40.1.1 OV2640 简介

OV2640 是 OV (OmniVision) 公司生产的一颗 1/4 寸的 CMOS UXGA (1632*1232) 图像传感器。该传感器体积小、工作电压低，提供单片 UXGA 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、缩放和取窗口等方式的各种分辨率 8/10 位影像数据。该产品 UXGA 图像最高达到 15 帧/秒 (SVGA 可达 30 帧，CIF 可达 60 帧)。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、对比度、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、拖尾、浮散等，提高图像质量，得到清晰的稳定的彩色图像。

OV2640 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(RGB565/RGB555)、GRB422、YUV(422/420)和 YCbCr (422) 输出格式
- 支持 UXGA、SXGA、SVGA 以及按比例缩小到从 SXGA 到 40*30 的任何尺寸
- 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、平移和窗口设置
- 支持图像压缩，即可输出 JPEG 图像数据
- 自带嵌入式微处理器

OV2640 的功能框图如图 40.1.1.1 所示：

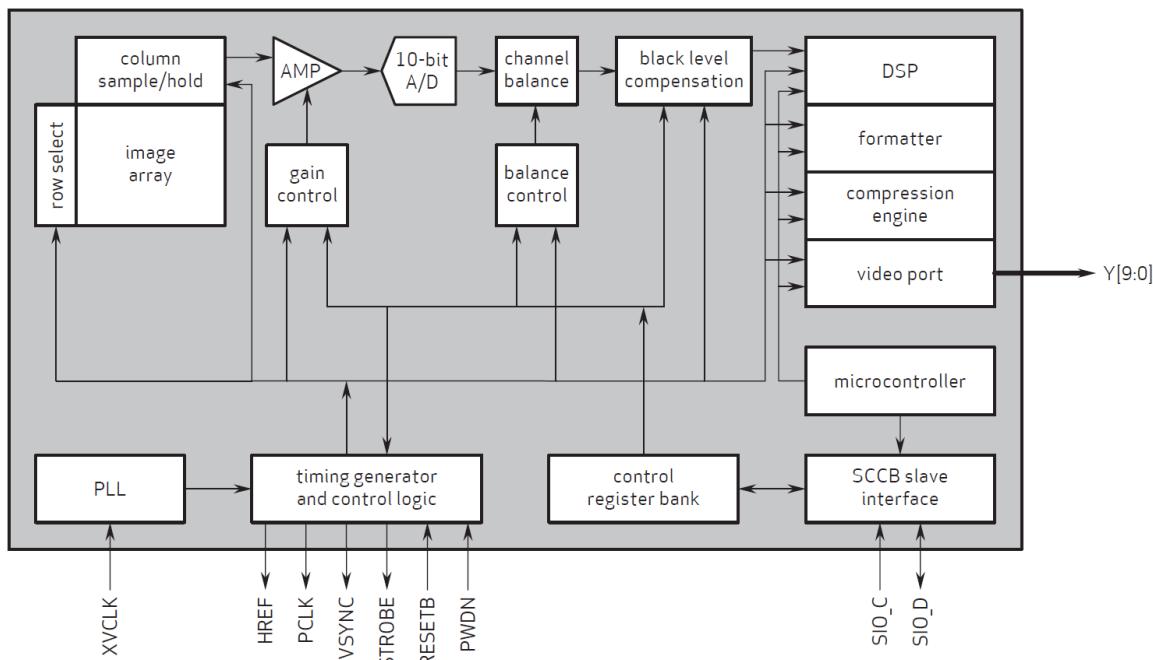


图 40.1.1.1 OV2640 功能框图

OV2640 传感器包括如下一些功能模块。

1. 感光整列 (Image Array)

OV2640 总共有 1632×1232 个像素，最大输出尺寸为 UXGA (1600×1200)，即 200W 像素。

2. 模拟信号处理 (Analog Processing)

模拟信号处理所有模拟功能，并包括：模拟放大 (AMP)、增益控制、通道平衡和平衡控制等。

3.10 位 A/D 转换 (A/D)

原始的信号经过模拟放大后，分 G 和 BR 两路进入一个 10 位的 A/D 转换器，A/D 转换器工作频率高达 20M，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，该模块还有黑电平校正 (BLC) 功能。

4. 数字信号处理器 (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/YCbCr)
- 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

5. 输出格式模块 (Output Formatter)

该模块按设定优先级控制图像的所有输出数据及其格式。

6. 压缩引擎 (Compression Engine)

压缩引擎框图如图 40.1.1.2 所示：

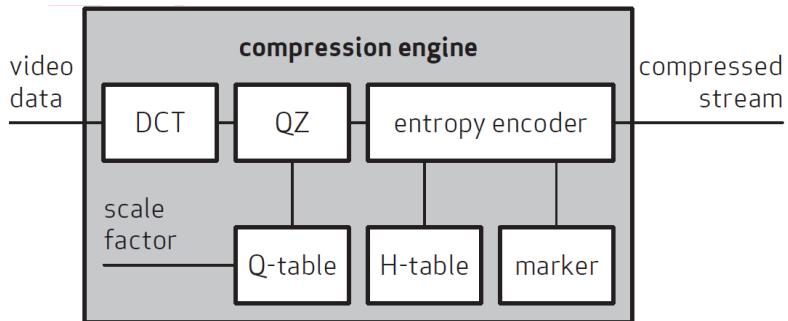


图 40.1.1.2 压缩引擎框图

从图可以看出，压缩引擎主要包括三部分：DCT、QZ 和 entropy encoder（熵编码器），将原始的数据流，压缩成 jpeg 数据输出。

7.微处理器 (Microcontroller)

OV2640 自带了一个 8 位微处理器，该处理器有 512 字节 SRAM，4KB 的 ROM，它提供一个灵活的主机到控制系统的指令接口，同时也具有细调图像质量的功能。

8.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 Seril Camera Control Bus(SCCB) Specification》这个文档

9.数字视频接口 (Digital Video Port)

OV2640 拥有一个 10 位数字视频接口(支持 8 位接法)，其 MSB 和 LSB 可以程序设置先后顺序，ALIENTEK OV2640 模块采用默认的 8 位连接方式，如图 40.1.1.3 所示：

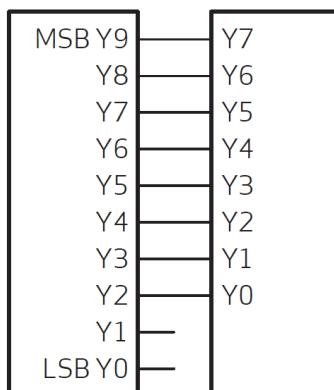


图 40.1.1.3 OV2640 默认 8 位连接方式

OV2640 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《OmniVision Technologies Seril Camera Control Bus(SCCB) Specification》这个文档。

接下来，我们介绍一下 OV2640 的传感器窗口设置、图像尺寸设置、图像窗口设置和图像输出大小设置，这几个设置与我们的正常使用密切相关，有必要了解一下。其中，除了传感器窗口设置是直接针对传感器阵列的设置，其他都是 DSP 部分的设置了，接下来我们一个个介绍。

传感器窗口设置，该功能允许用户设置整个传感器区域 (1632*1220) 的感兴趣部分，也就是在传感器里面开窗，开窗范围从 2*2~1632*1220 都可以设置，不过要求这个窗口必须大于等于随后设置的图像尺寸。传感器窗口设置，通过：0X03/0X19/0X1A/0X07/0X17/0X18 等寄存器设置，寄存器定义请看 OV2640_DS(1.6).pdf 这个文档（下同）。

图像尺寸设置，也就是 DSP 输出（最终输出到 LCD 的）图像的最大尺寸，该尺寸要小于等于前面我们传感器窗口设置所设定的窗口尺寸。图像尺寸通过：0XC0/0XC1/0X8C 等寄存器

设置。

图像窗口设置，这里起始和前面的传感器窗口设置类似，只是这个窗口是在我们前面设置的图像尺寸里面，再一次设置窗口大小，该窗口必须小于等于前面设置的图像尺寸。该窗口设置后的图像范围，将用于输出到外部。图像窗口设置通过：0X51/0X52/0X53/0X54/0X55/0X57 等寄存器设置。

图像输出大小设置，这是最终输出到外部的图像尺寸。该设置将图像窗口设置所决定的窗口大小，通过内部 DSP 处理，缩放成我们输出到外部的图像大小。该设置将会对图像进行缩放处理，如果设置的图像输出大小不等于图像窗口设置图像大小，那么图像就会被缩放处理，只有这两者设置一样大的时候，输出比例才是 1: 1 的。

因为 OmniVision 公司公开的文档，对这些设置实在是没有详细介绍。只能从他们提供的初始化代码（还得去 linux 源码里面移植过来）里面去分析规律，所以，这几个设置，都是作者根据 OV2640 的调试经验，以及相关文档总结出来的，不保证百分比正确，如有错误，还请大家指正。

以上几个设置，光看文字可能不太清楚，这里我们画一个简图有助于大家理解，如图 40.1.1.4 所示：

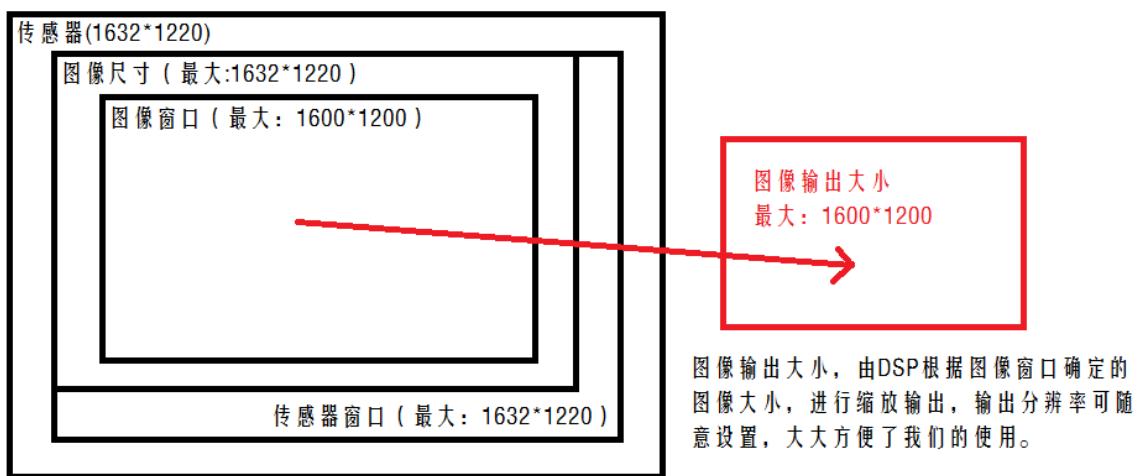


图 40.1.1.4 OV2640 图像窗口设置简图

上图，最终红色框所示的图像输出大小，才是 OV2640 输出给外部的图像尺寸，也就是显示在 LCD 上面的图像大小。当图像输出大小与图像窗口不等时，会进行缩放处理，在 LCD 上面看到的图像将会变形。

最后，我们介绍一下 OV2640 的图像数据输出格式。首先我们简单介绍一些定义：

UXGA，即分辨率位 1600*1200 的输出格式，类似的还有：SXGA(1280*1024)、WXGA+(1440*900)、XVGA(1280*960)、WXGA(1280*800)、XGA(1024*768)、SVGA(800*600)、VGA(640*480)、CIF(352*288)、WQVGA(400*240)、QCIF(176*144)和 QQVGA(160*120)等。

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。

OV2640 的图像数据输出（通过 Y[9:0]）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 40.1.1.5 所示：

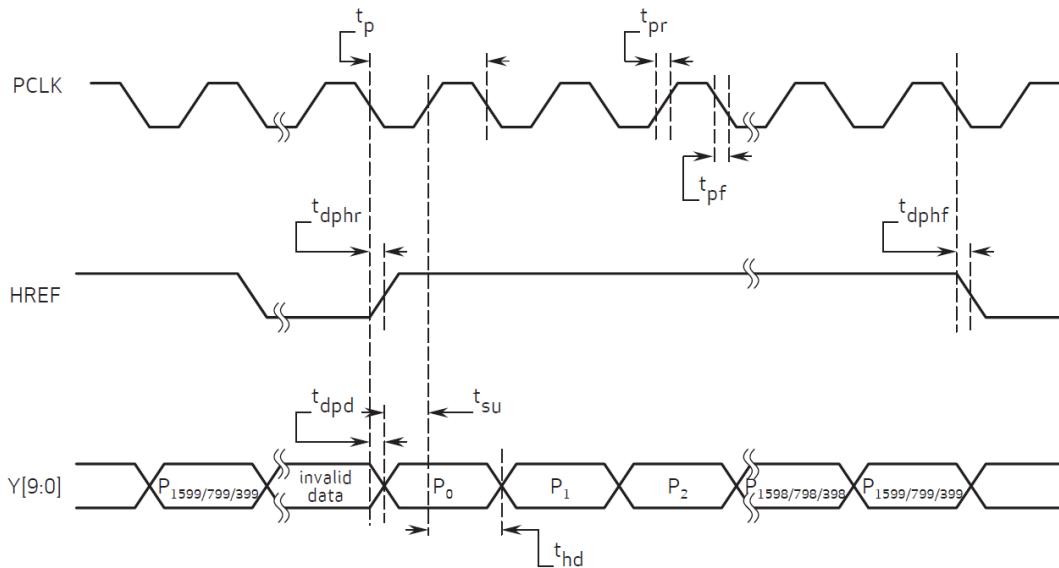


图 40.1.1.5 OV2640 行输出时序

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 8 位/10 位数据。我们采用 8 位接口，所以每个 PCLK 输出 1 个字节，且在 RGB/YUV 输出格式下，每个 $t_p=2$ 个 Tpclk，如果是 Raw 格式，则一个 $t_p=1$ 个 Tpclk。比如我们采用 UXGA 时序，RGB565 格式输出，每 2 个字节组成一个像素的颜色（高低字节顺序可通过 0XDA 寄存器设置），这样每行输出总共有 $1600*2$ 个 PCLK 周期，输出 $1600*2$ 个字节。

再来看看帧时序（UXGA 模式），如图 40.1.1.6 所示：

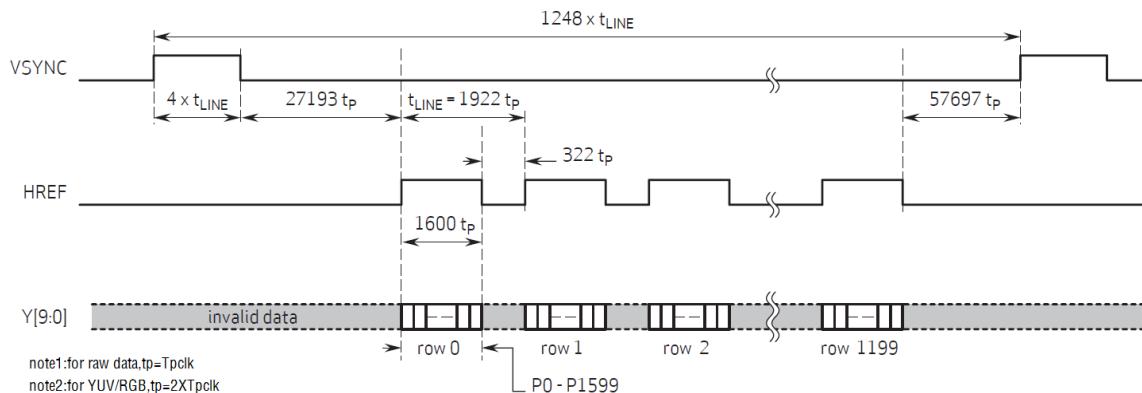


图 40.1.1.6 OV2640 帧时序

上图清楚的表示了 OV2640 在 UXGA 模式下的数据输出。我们按照这个时序去读取 OV2640 的数据，就可以得到图像数据。

最后说一下 OV2640 的图像数据格式，我们一般用 2 种输出方式：RGB565 和 JPEG。当输出 RGB565 格式数据的时候，时序完全就是上面两幅图介绍的关系。以满足不同需要。而当输出数据是 JPEG 数据的时候，同样也是这种方式输出（所以数据读取方法一模一样），不过 PCLK 数目大大减少了，且不连续，输出的数据是压缩后的 JPEG 数据，输出的 JPEG 数据以：0xFF,0xD8 开头，以 0xFF,0xD9 结尾，且在 0xFF,0xD8 之前，或者 0xFF,0xD9 之后，会有不定数量的其他数据存在（一般是 0），这些数据我们直接忽略即可，将得到的 0xFF,0xD8~0xFF,0xD9 之间的数据，保存为.jpg/jpeg 文件，就可以直接在电脑上打开看到图像了。

OV2640 自带的 JPEG 输出功能，大大减少了图像的数据量，使得其在网络摄像头、无线视

频传输等方面具有很大的优势。OV2640 我们就介绍到这。

40.1.2 STM32F4 DCMI 接口简介

STM32F4 自带了一个数字摄像头 (DCMI) 接口，该接口是一个同步并行接口，能够接收外部 8 位、10 位、12 位或 14 位 CMOS 摄像头模块发出的高速数据流。可支持不同的数据格式：YCbCr4:2:2/RGB565 逐行视频和压缩数据 (JPEG)。

STM32F4 DCMI 接口特点：

- 8 位、10 位、12 位或 14 位并行接口
- 内嵌码/外部行同步和帧同步
- 连续模式或快照模式
- 裁剪功能
- 支持以下数据格式：
 - 1, 8/10/12/14 位逐行视频：单色或原始拜尔 (Bayer) 格式
 - 2, YCbCr 4:2:2 逐行视频
 - 3, RGB 565 逐行视频
 - 4, 压缩数据：JPEG

DCMI 接口包括如下一些信号：

- 1, 数据输入 (D[0:13])，用于接摄像头的数据输出，接 OV2640 我们只用了 8 位数据。
- 2, 水平同步 (行同步) 输入 (HSYNC)，用于接摄像头的 HSYNC/HREF 信号。
- 3, 垂直同步 (场同步) 输入 (VSYNC)，用于接摄像头的 VSYNC 信号。
- 4, 像素时钟输入 (PIXCLK)，用于接摄像头的 PCLK 信号。

DCMI 接口是一个同步并行接口，可接收高速（可达 54 MB/s）数据流。该接口包含多达 14 条数据线(D13-D0)和一条像素时钟线(PIXCLK)。像素时钟的极性可以编程，因此可以在像素时钟的上升沿或下降沿捕获数据。

DCMI 接收到的摄像头数据被放到一个 32 位数据寄存器(DCMI_DR)中，然后通过通用 DMA 进行传输。图像缓冲区由 DMA 管理，而不是由摄像头接口管理。

从摄像头接收的数据可以按行/帧来组织（原始 YUV/RGB/拜尔模式），也可以是一系列 JPEG 图像。要使能 JPEG 图像接收，必须将 JPEG 位 (DCMI_CR 寄存器的位 3) 置 1。

数据流可由可选的 HSYNC (水平同步) 信号和 VSYNC (垂直同步) 信号硬件同步，或者通过数据流中嵌入的同步码同步。

STM32F4 DCMI 接口的框图如图 40.1.2.1 所示：

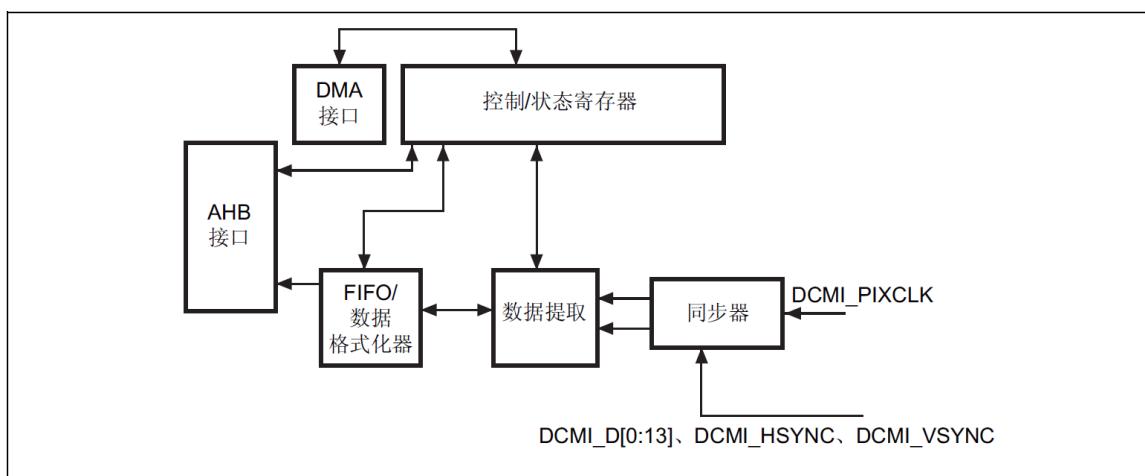


图 40.1.2.1 DCMI 接口框图

DCMI 接口的数据与 PIXCLK (即 PCLK) 保持同步，并根据像素时钟的极性在像素时钟上升沿/下降沿发生变化。HSYNC (HREF) 信号指示行的开始/结束，VSYNC 信号指示帧的开始/结束。DCMI 信号波形如图 40.1.2.2 所示：

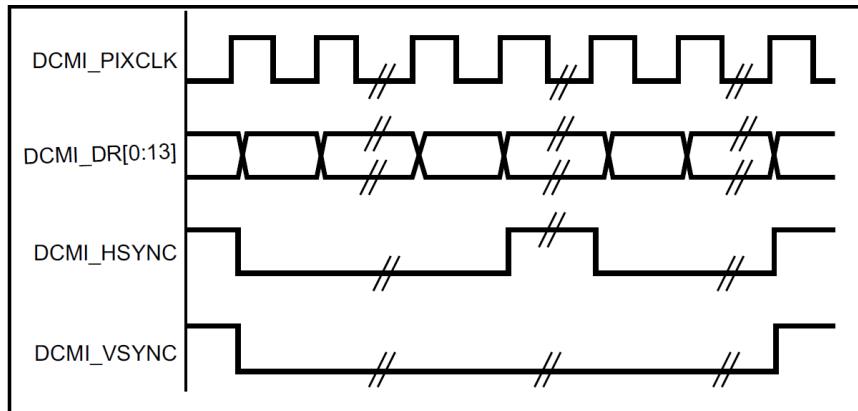


图 40.1.2.2 DCMI 信号波形

上图中，对应设置为：DCMI_PIXCLK 的捕获沿为下降沿，DCMI_HSYNC 和 DCMI_VSYNC 的有效状态为 1，注意，这里的有效状态实际上对应的是指示数据在并行接口上无效时，HSYNC/VSYNC 引脚上面的引脚电平。

本章我们用到 DCMI 的 8 位数据宽度，通过设置 DCMI_CR 中的 EDM[1:0]=00 设置。此时 DCMI_D0~D7 有效，DCMI_D8~D13 上的数据则忽略，这个时候，每次需要 4 个像素时钟来捕获一个 32 位数据。捕获的第一个数据存放在 32 位字的 LSB 位置，第四个数据存放在 32 位字的 MSB 位置，捕获数据字节在 32 位字中的排布如表 40.1.2.1 所示：

字节地址	31:24	23:16	15:8	7:0
0	D _{n+3} [7:0]	D _{n+2} [7:0]	D _{n+1} [7:0]	D _n [7:0]
4	D _{n+7} [7:0]	D _{n+6} [7:0]	D _{n+5} [7:0]	D _{n+4} [7:0]

表 40.1.2.1 8 位捕获数据在 32 位字中的排布

从表 40.1.2.1 可以看出，STM32F4 的 DCMI 接口，接收的数据是低字节在前，高字节在后的，所以，要求摄像头输出数据也是低字节在前，高字节在后才可以，否则就还得程序上处理字节顺序，会比较麻烦。

DCMI 接口支持 DMA 传输，当 DCMI_CR 寄存器中的 CAPTURE 位置 1 时，激活 DMA 接口。摄像头接口每次在其寄存器中收到一个完整的 32 位数据块时，都将触发一个 DMA 请求。

DCMI 接口支持两种同步方式：内嵌码同步和硬件（HSYNC 和 VSYNC）同步。我们简单介绍下硬件同步，详细介绍请参考《STM32F4xx 中文数据手册》第 13.5.3 节。

硬件同步模式下将使用两个同步信号（HSYNC/VSYNC）。根据摄像头模块/模式的不同，可能在水平/垂直同步期间内发送数据。由于系统会忽略 HSYNC/VSYNC 信号有效电平期间内接收的所有数据，HSYNC/VSYNC 信号相当于消隐信号。

为了正确地将图像传输到 DMA/RAM 缓冲区，数据传输将与 VSYNC 信号同步。选择硬件同步模式并启用捕获（DCMI_CR 中的 CAPTURE 位置 1）时，数据传输将与 VSYNC 信号的无效电平同步（开始下一帧时）。之后传输便可以连续执行，由 DMA 将连续帧传输到多个连续的缓冲区或一个具有循环特性的缓冲区。为了允许 DMA 管理连续帧，每一帧结束时都将激活 VSIF（垂直同步中断标志，即帧中断），我们可以利用这个帧中断来判断是否有一帧数据采集完成，方便处理数据。

DCMI 接口的捕获模式支持：快照模式和连续采集模式。一般我们使用连续采集模式，通过 DCMI_CR 中的 CM 位设置。另外，DCMI 接口还支持实现了 4 个字深度的 FIFO，配有一个简单的 FIFO 控制器，每次摄像头接口从 AHB 读取数据时读指针递增，每次摄像头接口向 FIFO 写入数据时写指针递增。因为没有溢出保护，如果数据传输率超过 AHB 接口能够承受的速率，FIFO 中的数据就会被覆盖。如果同步信号出错，或者 FIFO 发生溢出，FIFO 将复位，DCMI 接口将等待新的数据帧开始。

关于 DCMI 接口的其他特性，我们这里就不再介绍了，请大家参考《STM32F4xx 中文参考手册》第 13 章相关内容。

本章，我们将使用 STM32F407ZGT6 的 DCMI 接口连接 ALIENTEK OV2640 摄像头模块，该模块采用 8 位数据输出接口，自带 24M 有源晶振，无需外部提供时钟，采用百万高清镜头，单独 3.3V 供电即可正常使用。

ALIENTEK OV2640 摄像头模块外观如图 40.1.2.3 所示：



图 40.1.2.3 ALIENTEK OV2640 摄像头模块外观图

模块原理图如图 40.1.2.4 所示：

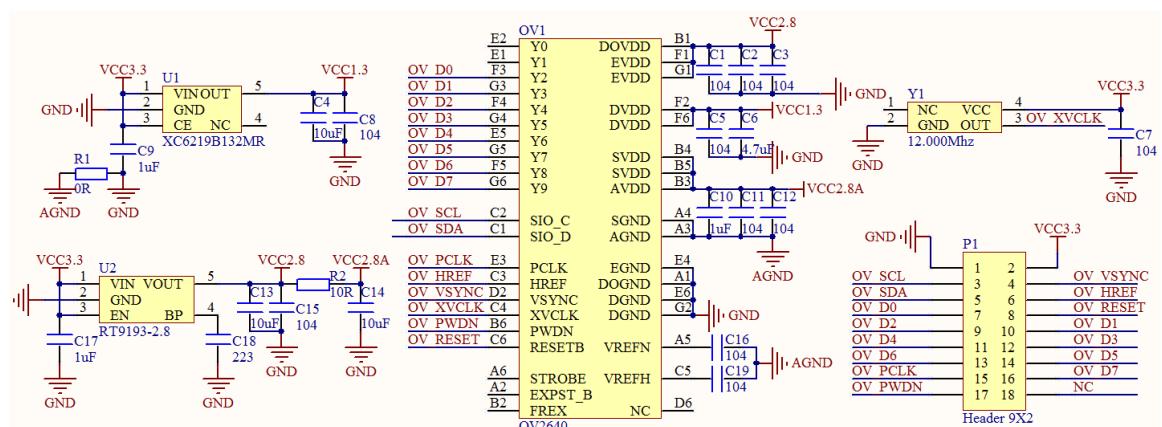


图 40.1.2.4 ALIENTEK OV2640 摄像头模块原理图

从上图可以看出，ALIENTEK OV2640 摄像头模块自带了有源晶振，用于产生 24M 时钟作为 OV2640 的 XVCLK 输入。同时自带了稳压芯片，用于提供 OV2640 稳定的 2.8V 和 1.3V 工作电压，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 40.1.2.2 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	OV_PCLK	像素时钟输出
GND	模块地线	OV_PWDN	掉电使能(高有效)
OV_SCL	SCCB 通信时钟信号	OV_VSYNC	帧同步信号输出
OV_SDA	SCCB 通信数据信号	OV_HREF	行同步信号输出
OV_D[7:0]	8 位数据输出	OV_RESET	复位信号(低有效)

表 40.1.2.2 OV2640 模块信号及其作用描述

本章，我们将 OV2640 默认配置为 UXGA 输出，也就是 1600*1200 的分辨率，输出信号设置为：VSYNC 高电平有效，HREF 高电平有效，输出数据在 PCLK 的下降沿输出（即上升沿的时候，MCU 才可以采集）。这样，STM32F4 的 DCMI 接口就必须设置为：VSYNC 低电平有效、Hsync 低电平有效和 PIXCLK 上升沿有效，这些设置都是通过 DCMI_CR 寄存器控制的，该寄存器描述如图 40.1.2.5 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

图 40.1.2.5 DCMI_CR 寄存器各位描述

ENABLE，该位用于设置是否使能 DCMI，不过，在使能之前，必须将其他配置设置好。

FCRC[1:0]，这两个位用于帧率控制，我们捕获所有帧，所以设置为 00 即可。

VSPOL，该位用于设置垂直同步极性，也就是 VSYNC 引脚上面，数据无效时的电平状态，根据前面说所，我们应该设置为 0。

HSPOL，该位用于设置水平同步极性，也就是 Hsync 引脚上面，数据无效时的电平状态，同样应该设置为 0。

PCKPOL，该位用于设置像素时钟极性，我们用上升沿捕获，所以设置为 1。

CM，该位用于设置捕获模式，我们用连续采集模式，所以设置为 0 即可。

CAPTURE，该位用于使能捕获，我们设置为 1。该位使能后，将激活 DMA，DCMI 等待第一帧开始，然后生成 DMA 请求将收到的数据传输到目标存储器中。注意：该位必须在 DCMI 的其他配置（包括 DMA）都设置好了之后，才设置！！

DCMI_CR 寄存器的其他位，我们就不介绍了，另外 DCMI 的其他寄存器这里也不再介绍，请大家参考《STM32F4xx 中文参考手册》第 13.8 节。

最后，我们来看下怎么用库函数实现 DCMI 驱动 OV2640 的步骤：

1) 配置 OV2640 控制引脚，并配置 OV2640 工作模式。

在启动 DCMI 之前，我们先设置好 OV2640。OV2640 通过 OV_SCL 和 OV_SDA 进行寄存器配置，同时还有 OV_PWDN/OV_RESET 等信号，我们也要配置对应 IO 状态，先设置 OV_PWDN=0，退出掉电模式，然后拉低 OV_RESET 复位 OV2640，之后再设置 OV_RESET 为 1，结束复位，然后就是对 OV2640 的大把寄存器进行配置了，这里我们配置成 UXGA 输出。然后，可以根据我们的需要，设置成 RGB565 输出模式，还是 JPEG 输出模式。

学习到这里，怎么使能相关 IO 口时钟以及配置相关模式这里我们就不再讲解，大家可以打开我们实验源码查看，OV2640 的初始化配置以及相关操作函数在我们实验的 ov2640.c 源文件中，其中初始化在 OV2640_Init 函数中，大家可以打开看看相关步骤。

2) 配置相关引脚的模式和复用功能 (AF13)，使能时钟。

OV2640 配置好之后，再设置 DCMI 接口与摄像头模块连接的 IO 口，使能 IO 和 DCMI 时

钟，然后设置相关 IO 口为复用功能模式，复用功能选择 AF13(DCMI 复用)。

使能 DCMI 时钟的方法为：

```
RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_DCMI,ENABLE); //使能 DCMI 时钟
```

关于相关 IO 口设置复用功能的方法之前已经多次讲解，这里我们就只贴出关键代码片段：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能输出
```

接下来我们就是设置复用映射，方法为：

```
GPIO_PinAFConfig(GPIOA,GPIO_PinSource4,GPIO_AF_DCMI); //PA4,DCMI_HSYNC  
GPIO_PinAFConfig(GPIOA,GPIO_PinSource6,GPIO_AF_DCMI); //PA6, DCMI_PCLK  
GPIO_PinAFConfig(GPIOB,GPIO_PinSource7,GPIO_AF_DCMI); //PB7, DCMI_VSYNC  
.....//省略部分 IO 映射代码  
GPIO_PinAFConfig(GPIOB,GPIO_PinSource6,GPIO_AF_DCMI); //PB6,DCMI_D5  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource5,GPIO_AF_DCMI); //PE5,DCMI_D6  
GPIO_PinAFConfig(GPIOE,GPIO_PinSource6,GPIO_AF_DCMI); //PE6,DCMI_D7
```

3) 配置 DCMI 相关设置。

这一步，主要通过 DCMI_CR 寄存器设置，包括 VSPOL/HSPOL/PCKPOL/数据宽度等重要参数，都在这一步设置，同时我们也开启帧中断，编写 DCMI 中断服务函数，方便进行数据处理（尤其是 JPEG 模式的时候）。不过对于 CAPTURE 位，我们等待 DMA 配置好之后再设置，另外对于 OV2640 输出的 JPEG 数据，我们也不使用 DCMI 的 JPEG 数据模式（实测设置不设置都一样），而是采用正常模式，直接采集。

DCMI 相关寄存器的配置是通过函数 DCMI_Init 来实现的。接下来我们看看函数申明：

```
void DCMI_Init(DCMI_InitTypeDef* DCMI_InitStruct);
```

同样，我们来看看结构体 DCMI_InitTypeDef 的定义：

```
typedef struct  
{  
    uint16_t DCMI_CaptureMode;  
    uint16_t DCMI_SynchroMode;  
    uint16_t DCMI_PCKPolarity;  
    uint16_t DCMI_VSPolarity;  
    uint16_t DCMI_HSPolarity;  
    uint16_t DCMI_CaptureRate;  
    uint16_t DCMI_ExtendedDataMode;  
} DCMI_InitTypeDef;
```

结构体 DCMI_InitTypeDef 一共有 7 个成员变量，接下来我们来看看每个成员变量的含义：

第一个参数 DCMI_CaptureMode 是用来设置捕获模式为连续捕获模式还是快照模式。我们实验采取的是连续捕获模式值 DCMI_CaptureMode_Continuous，也就是通过 DMA 连续传输数据到目标存储区。

第二个参数 DCMI_SynchroMode 用来选择同步方式为硬件同步还是内嵌码同步。如果选择硬件同步值 DCMI_SynchroMode_Hardware，那么数据捕获由 HSYNC/VSYNC 信号同步，如果选择内嵌码同步方式值 DCMI_SynchroMode_EMBEDDED，那么数据捕获由数据流中嵌入的同步码同步。

第三个参数 DCMI_PCKPolarity 用来设置像素时钟极性为上升沿有效还是下降沿有效。我们实验使用的是上升沿有效，所以值为 DCMI_PCKPolarity_Rising。

第四个参数 DCMI_VSPolarity 用来设置垂直同步极性 VSYNC 为低电平有效还是高电平有

效。也就是 VSYNC 引脚上面，数据无效时的电平状态。我们设置为 VSYNC 低电平有效。所以值为 DCMI_VSPolarity_Low。

第五个参数 DCMI_HSPolarity 用来设置水平同步极性为高电平有效还是低电平有效，也就是 HSYNC 引脚上面，数据无效时的电平状态。我们设置为 HSYNC 低电平有效。所以值为 DCMI_HSPolarity_Low。

第六个参数 DCMI_CaptureRate 用来设置帧捕获率。如果设置为值 DCMI_CaptureRate_All_Frame，也就是全帧捕获，设置为 DCMI_CaptureRate_1of2_Frame，也就 2 帧捕获一帧，设置为 DCMI_CaptureRate_1of4_Frame，也就是 4 帧捕获一帧。

第七个参数 DCMI_ExtendedDataMode 用来设置扩展数据模式。可以设置为每个像素时钟捕获 8 位，10 位，12 位以及 14 位数据。这里我们设置为 8 位值 DCMI_ExtendedDataMode_8b。

DCMI 初始化实例如下：

```
DCMI_InitTypeDef DCMI_InitStructure;
DCMI_InitStructure.DCMI_CaptureMode=DCMI_CaptureMode_Continuous;//连续模式
DCMI_InitStructure.DCMI_CaptureRate=DCMI_CaptureRate_All_Frame;//全帧捕获
DCMI_InitStructure.DCMI_ExtendedDataMode= DCMI_ExtendedDataMode_8b;//8 位格式
DCMI_InitStructure.DCMI_HSPolarity= DCMI_HSPolarity_Low;//HSYNC 低电平有效
DCMI_InitStructure.DCMI_PCKPolarity= DCMI_PCKPolarity_Rising;//PCLK 上升沿有效
DCMI_InitStructure.DCMI_SynchroMode= DCMI_SynchroMode_Hardware;//硬件同步
DCMI_InitStructure.DCMI_VSPolarity=DCMI_VSPolarity_Low;//VSYNC 低电平有效
DCMI_Init(&DCMI_InitStructure);//初始化 DCMI
```

4) 配置 DMA。

本章采用连续模式采集，并将采集到的数据输出到 LCD（RGB565 模式）或内存（JPEG 模式），所以源地址都是 DCMI_DR，而目的地址可能是 LCD->RAM 或者 SRAM 的地址。DCMI 的 DMA 传输采用的是 DMA2 数据流 1 的通道 1 来实现的，关于 DMA 的介绍，请大家参考前面的 DMA 实验章节。这里我们列出本章我们的 DMA 配置源码如下：

```
DMA_InitTypeDef DMA_InitStructure;
DMA_InitStructure.DMA_Channel = DMA_Channel_1; //通道 1 DCMI 通道
DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&DCMI->DR;//外设地址为:DCMI->DR
DMA_InitStructure.DMA_Memory0BaseAddr = (u32)&LCD->LCD_RAM;//存储器 0 地址
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//外设到存储器模式
DMA_InitStructure.DMA_BufferSize = 1;//数据传输量
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
DMA_InitStructure.DMA_PeripheralDataSize=DMA_PeripheralDataSize_Word;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
DMA_InitStructure.DMA_Priority = DMA_Priority_High;//高优先级
DMA_InitStructure.DMA_FIFOMode=DMA_FIFOMode_Enable; //FIFO 模式
DMA_InitStructure.DMA_FIFOThreshold=DMA_FIFOThreshold_Full;//使用全 FIFO
DMA_InitStructure.DMA_MemoryBurst=DMA_MemoryBurst_Single;//外设突发单次传输
DMA_InitStructure.DMA_PeripheralBurst=DMA_PeripheralBurst_Single;
DMA_Init(DMA2_Stream1, &DMA_InitStructure);//初始化 DMA Stream
```

5) 设置 OV2640 的图像输出大小，使能 DCMI 捕获。

图像输出大小设置，分两种情况：在 RGB565 模式下，我们根据 LCD 的尺寸，设置输出图像大小，以实现全屏显示（图像可能因缩放而变形）；在 JPEG 模式下，我们可以自由设置输出图像大小（可不缩放）；最后，开启 DCMI 捕获，即可正常工作了。开启 DCMI 捕获的方法为：

```
DCMI_CaptureCmd(ENABLE); //DCMI 捕获使能
```

设置过程就给大家讲解到这里。

40.2 硬件设计

并可选择 RGB565 和 JPEG 两种输出格式，当使用 RGB565 时，输出图像将经过缩放处理（完全由 OV2640 的 DSP 控制），显示在 LCD 上面。当使用 JPEG 数据输出的时候，我们将收到的 JPEG 数据，通过串口 2（115200 波特率），送给电脑，并利用电脑端上位机软件，显示接收到的图片。

本章实验功能简介：开机后，初始化摄像头模块（OV2640），如果初始化成功，则提示选择模式：RGB565 模式，或者 JPEG 模式。KEY0 用于选择 RGB565 模式，KEY1 用于选择 JPEG 模式。

当使用 RGB565 时，输出图像（固定为：UXGA）将经过缩放处理（完全由 OV2640 的 DSP 控制），显示在 LCD 上面。我们可以通过 KEY_UP 按键选择：1:1 显示，即不缩放，图片不变形，但是显示区域小（液晶分辨率大小），或者缩放显示，即将 1600*1200 的图像压缩到液晶分辨率尺寸显示，图片变形，但是显示了整个图片内容。通过 KEY0 按键，可以设置对比度；KEY1 按键，可以设置饱和度；KEY2 按键，可以设置特效。

当使用 JPEG 模式时，图像可以设置任意尺寸（QQVGA~UXGA），采集到的 JPEG 数据将先存放到 STM32F4 的内存里面，每当采集到一帧数据，就会关闭 DMA 传输，然后将采集到的数据发送到串口 2（此时可以通过上位机软件（串口摄像头.exe）接收，并显示图片），之后再重新启动 DMA 传输。我们可以通过 KEY_UP 设置输出图片的尺寸（QQVGA~UXGA）。通过 KEY0 按键，可以设置对比度；KEY1 按键，可以设置饱和度；KEY2 按键，可以设置特效。

同时可以通过串口 1，借助 USMART 设置/读取 OV2640 的寄存器，方便大家调试。DS0 指示程序运行状态，DS1 用于指示帧中断。

本实验用到的硬件资源有：

- 1) 指示灯 DS0 和 DS1
- 2) 4 个按键
- 3) 串口 1 和串口 2
- 4) TFTLCD 模块
- 5) OV2640 摄像头模块

这些资源，基本上都介绍过了，这里我们用到串口 2 来传输 JPEG 数据给上位机，其配置同串口 1 几乎一模一样，支持串口 2 的时钟来自 APB1，频率为 42Mhz。

我们重点介绍下探索者 STM32F4 开发板的摄像头接口与 ALIENTEK OV2640 摄像头模块的连接。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 OV2640 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 40.2.1 所示：

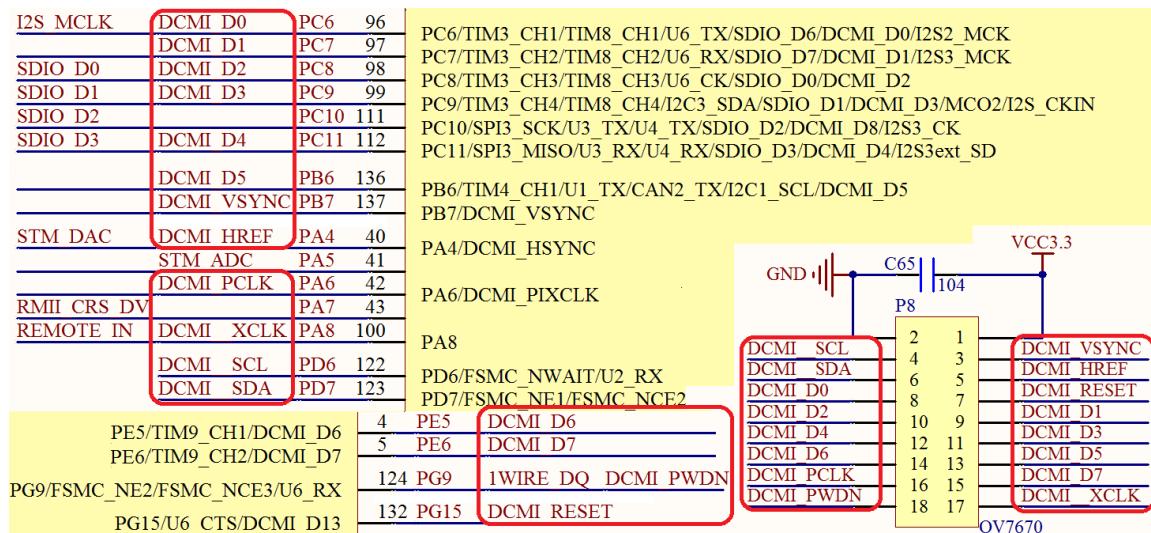


图 40.2.1 摄像头模块接口与 STM32 连接图

从上图可以看出，OV2640 摄像头模块的各信号脚与 STM32 的连接关系为：

- DCMI_VSYNC 接 PB7;
- DCMI_HREF 接 PA4;
- DCMI_PCLK 接 PA6;
- DCMI_SCL 接 PD6;
- DCMI_SDA 接 PD7;
- DCMI_RESET 接 PG15;
- DCMI_PWDN 接 PG9;
- DCMI_XCLK 接 PA8 (本章未用到)；
- DCMI_D[7:0]接 PE6/PE5/PB6/PC11/PC9/PC8/PC7/PC6;

这些线的连接，探索者 STM32F4 开发板的内部已经连接好了，我们只需要将 OV2640 摄像头模块插上去就好了。**特别注意：**DCMI 摄像头接口和 I2S 接口、DAC、SDIO 以及 1WIRE_DQ 等有冲突，使用的时候，必须分时复用才可以，不可同时使用。实物连接如图 40.2.2 所示：

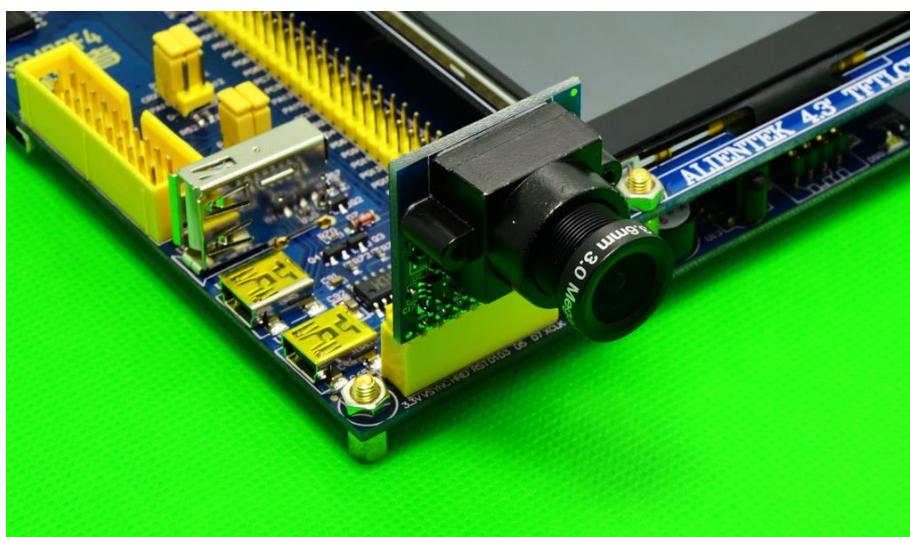


图 40.2.2 OV2640 摄像头模块与开发板连接实物图

40.3 软件设计

打开本章实验工程目录，我们在 HARDWARE 文件夹下新建了 OV2640、DCMI 和 USART2 三个文件夹。在 OV2640 文件夹下新建了 ov2640.c、sccb.c、ov2640.h、sccb.h、ov2640cfg.h 等 5 个文件，并同时在工程中将这个文件夹加入头文件包含路径。在 DCMI 文件夹下新建了 dcmi.c 和 dcmi.h 两个文件，也将这个文件夹加入头文件包含路径。在 USART2 文件夹下新建了：uart2.c 和 usart2.h 两个文件，同时将这个文件夹加入头文件包含路径。

DCMI 接口相关的库函数分布在 stm32f4xx_dcmi.c 文件以及对应的头文件 stm32f4xx_dcmi.h 中。我们工程引入了这两个文件。

本章总共新增了 9 个文件，代码比较多，我们就不一一列出了，仅挑几个重要的地方进行讲解。首先，我们来看 ov2640.c 里面的 OV2640_Init 函数，该函数代码如下：

```
//初始化 OV2640
//配置完以后，默认输出是 1600*1200 尺寸的图片!!
//返回值:0,成功 其他,错误代码
u8 OV2640_Init(void)
{
    u16 i=0, reg;
    //设置 IO
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);
    //GPIOG9, GPIOG15 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_15;//PG9,15 推挽输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //推挽输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOG, &GPIO_InitStructure);//初始化

    OV2640_PWDN=0; delay_ms(10); //POWER ON
    OV2640_RST=0; delay_ms(10); //复位 OV2640
    OV2640_RST=1; //结束复位
    SCCB_Init(); //初始化 SCCB 的 IO 口
    SCCB_WR_Reg(OV2640_DSP_RA_DLMT, 0x01); //操作 sensor 寄存器
    SCCB_WR_Reg(OV2640_SENSOR_COM7, 0x80); //软复位 OV2640
    delay_ms(50);
    reg=SCCB_RD_Reg(OV2640_SENSOR_MIDH); //读取厂家 ID 高八位
    reg<<=8;
    reg|=SCCB_RD_Reg(OV2640_SENSOR_MIDL); //读取厂家 ID 低八位
    if(reg!=OV2640_MID)
    {
        printf("MID:%d\r\n",reg);
        return 1;
    }
}
```

```

reg=SCCB_RD_Reg(OV2640_SENSOR_PIDH);      //读取厂家 ID 高八位
reg<=8;
reg|=SCCB_RD_Reg(OV2640_SENSOR_PIDL);    //读取厂家 ID 低八位
if(reg!=OV2640_PID)
{
    printf("HID:%d\r\n",reg);
}
//初始化 OV2640,采用 SXGA 分辨率(1600*1200)
for(i=0;i<sizeof.ov2640_uxga_init_reg_tbl)/2;i++)
{
    SCCB_WR_Reg(OV2640_uxga_init_reg_tbl[i][0],OV2640_uxga_init_reg_tbl[i][1]);
}
return 0x00; //ok
}

```

此部分代码先初始化 OV2640 相关的 IO 口(包括 SCCB_Init)，然后最主要的是完成 OV2640 的寄存器序列初始化。OV2640 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《OV2640 Software Application Notes 1.03》)，本章我们用到的配置序列，存放在 ov2640_uxga_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 ov2640cfg.h 里面。

另外，在 ov2640.c 里面，还有几个函数比较重要，这里贴代码了，只介绍功能：

OV2640_Window_Set 函数，该函数用于设置传感器输出窗口；

OV2640_ImageSize_Set 函数，用于设置图像尺寸；

OV2640_ImageWin_Set 函数，用于设置图像窗口大小；

OV2640_OutSize_Set 函数，用于设置图像输出大小；

这就是我们在 40.1.1 节所介绍的 4 个设置，他们共同决定了图像的输出。接下来，我们看看 ov2640cfg.h 里面 ov2640_uxga_init_reg_tbl 的内容，ov2640cfg.h 文件的代码如下：

```

//OV2640 UXGA 初始化寄存器序列表
//此模式下帧率为 15 帧
//UXGA(1600*1200)
const u8 ov2640_uxga_init_reg_tbl[][2]=
{
    0xff, 0x00,
    .....//省略部分代码
    0x05, 0x00,
};

//OV2640 SVGA 初始化寄存器序列表
//此模式下,帧率可以达到 30 帧
//SVGA 800*600
const u8 ov2640_svga_init_reg_tbl[][2]=
{
    0xff, 0x00,
    .....//省略部分代码
    0x05, 0x00,
};

```

```

};

const u8 ov2640_yuv422_reg_tbl[][2]=
{
    0xFF, 0x00,
    .....//省略部分代码
    0x00, 0x00,
};

const u8 ov2640_jpeg_reg_tbl[][2]=
{
    0xff, 0x01,
    .....//省略部分代码
    0xe0, 0x00,
};

const u8 ov2640_rgb565_reg_tbl[][2]=
{
    0xFF, 0x00,
    .....//省略部分代码
    0xe0, 0x00,
};

```

以上代码，我们省略了很多（全部贴出来太长了），里面总共有 5 个数组。我们大概了解下数组结构，每个数组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{0xff, 0x01}，就表示在 0Xff 地址，写入 0X01 这个值。

五个数组里面 ov2640_uxga_init_reg_tbl 和 ov2640_svga_init_reg_tbl，分别用于配置 OV2640 输出 UXGA 和 SVGA 分辨率的图像，我们只用了 ov2640_uxga_init_reg_tbl 这个数组，完成对 OV2640 的初始化(设置为 UXGA)。最后 OV2640 要输出数据是 RGB565 还是 JPEG，就得通过其他数组设置，输出 RGB565 时，通过一个数组：ov2640_rgb565_reg_tbl 设置即可；输出 JPEG 时，则要通过 ov2640_yuv422_reg_tbl 和 ov2640_jpeg_reg_tbl 两个数组设置。

接下来，我们看看 dcmi.c 里面的代码，如下：

```

u8 ov_frame=0;                      //帧率
extern void jpeg_data_process(void); //JPEG 数据处理函数
//DCMI 中断服务函数
void DCMI_IRQHandler(void)
{
    if(DCMI_GetITStatus(DCMI_IT_FRAME)==SET)//捕获到一帧图像
    {
        jpeg_data_process(); //jpeg 数据处理
        DCMI_ClearITPendingBit(DCMI_IT_FRAME);//清除帧中断
        LED1=!LED1;
        ov_frame++;
    }
}
//DMA_Memory0BaseAddr:存储器地址 将要存储摄像头数据的内存地址
//DMA_BufferSize:存储器长度      0~65535

```

```
//DMA_MemoryDataSize:存储器位宽
//DMA_MemoryInc:存储器增长方式 @defgroup DMA_memory_incremented_mode
void DCMI_DMA_Init(u32 DMA_Memory0BaseAddr,
                    u16 DMA_BufferSize,u32 DMA_MemoryDataSize,u32 DMA_MemoryInc)
{
    DMA_InitTypeDef DMA_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2,ENABLE);//DMA2 时钟使能
    DMA_DeInit(DMA2_Stream1);
    while (DMA_GetCmdStatus(DMA2_Stream1) != DISABLE){} //等待 DMA 可配置
    /* 配置 DMA Stream */
    DMA_InitStructure.DMA_Channel = DMA_Channel_1; //通道 1 DCMI 通道
    DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&DCMI->DR;//外设地址
    DMA_InitStructure.DMA_Memory0BaseAddr = DMA_Memory0BaseAddr;
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//外设到存储器模式
    DMA_InitStructure.DMA_BufferSize = DMA_BufferSize;//数据传输量
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc;//存储器增量模式
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;// 32 位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize;//存储器数据长度
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;//高优先级
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Enable; //FIFO 模式
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;//使用全 FIFO
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
    DMA_Init(DMA2_Stream1, &DMA_InitStructure); //初始化 DMA Stream
}
//DCMI 初始化
void My_DCMI_Init (void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOB
                           |RCC_AHB1Periph_GPIOC|RCC_AHB1Periph_GPIOE, ENABLE); //使能 PABCE 时钟
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_DCMI,ENABLE); //使能 DCMI 时钟
    //PA4/6 初始化设置
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4|GPIO_Pin_6;//PA4/6 复用功能输出
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
}
```

```
GPIO_Init(GPIOA, &GPIO_InitStructure); // 初始化
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7|GPIO_Pin_6; // PB6/7 复用功能输出
```

```
GPIO_Init(GPIOB, &GPIO_InitStructure); // 初始化
```

```
GPIO_InitStructure.GPIO_Pin =
```

```
GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_11; // 复用功能输出
```

```
GPIO_Init(GPIOC, &GPIO_InitStructure); // 初始化
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6; // PE5/6 复用功能输出
```

```
GPIO_Init(GPIOE, &GPIO_InitStructure); // 初始化
```

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource4, GPIO_AF_DCMI); // PA4, DCMI_HSYNC
```

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_DCMI); // PA6, DCMI_PCLK
```

```
GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_DCMI); // PB7, DCMI_VSYNC
```

```
GPIO_PinAFConfig(GPIOC, GPIO_PinSource6, GPIO_AF_DCMI); // PC6, DCMI_D0
```

```
GPIO_PinAFConfig(GPIOC, GPIO_PinSource7, GPIO_AF_DCMI); // PC7, DCMI_D1
```

```
GPIO_PinAFConfig(GPIOC, GPIO_PinSource8, GPIO_AF_DCMI); // PC8, DCMI_D2
```

```
GPIO_PinAFConfig(GPIOC, GPIO_PinSource9, GPIO_AF_DCMI); // PC9, DCMI_D3
```

```
GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_DCMI); // PC11, DCMI_D4
```

```
GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_DCMI); // PB6, DCMI_D5
```

```
GPIO_PinAFConfig(GPIOE, GPIO_PinSource5, GPIO_AF_DCMI); // PE5, DCMI_D6
```

```
GPIO_PinAFConfig(GPIOE, GPIO_PinSource6, GPIO_AF_DCMI); // PE6, DCMI_D7
```

```
DCMI_DeInit(); // 清除原来的设置
```

```
DCMI_InitStructure.DCMI_CaptureMode = DCMI_CaptureMode_Continuous; // 连续模式
```

```
DCMI_InitStructure.DCMI_CaptureRate = DCMI_CaptureRate_All_Frame; // 全帧捕获
```

```
DCMI_InitStructure.DCMI_ExtendedDataMode = DCMI_ExtendedDataMode_8b; // 8 位
```

```
DCMI_InitStructure.DCMI_HSPolarity = DCMI_HSPolarity_Low; // HSYNC 低电平有效
```

```
DCMI_InitStructure.DCMI_PCKPolarity = DCMI_PCKPolarity_Rising; // PCL 上升沿有效
```

```
DCMI_InitStructure.DCMI_SynchroMode = DCMI_SynchroMode_Hardware; // 硬件同步
```

```
DCMI_InitStructure.DCMI_VSPolarity = DCMI_VSPolarity_Low; // VSYNC 低电平有效
```

```
DCMI_Init(&DCMI_InitStructure); // DCMI 初始化
```

```
DCMI_ITConfig(DCMI_IT_FRAME, ENABLE); // 开启帧中断
```

```
DCMI_Cmd(ENABLE); // DCMI 使能
```

```
NVIC_InitStructure.NVIC_IRQChannel = DCMI_IRQn;
```

```
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // 抢占优先级 1
```

```
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2; // 响应优先级 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、 }
```

```
// DCMI, 启动传输
```

```

void DCMI_Start(void)
{
    LCD_SetCursor(0,0);
    LCD_WriteRAM_Prep();
    DMA_Cmd(DMA2_Stream1, ENABLE); //开启 DMA2, Stream1
    DCMI_CaptureCmd(ENABLE); //DCMI 捕获使能
}

//DCMI,关闭传输
void DCMI_Stop(void)
{
    DCMI_CaptureCmd(DISABLE); //DCMI 捕获使能关闭
    while(DCMI->CR&0X01); //等待传输结束
    DMA_Cmd(DMA2_Stream1, DISABLE); //关闭 DMA2, Stream1
}

```

其中：DCMI_IRQHandler 函数，用于处理帧中断，可以实现帧率统计（需要定时器支持）和 JPEG 数据处理等。DCMI_DMA_Init 函数，则用于配置 DCMI 的 DMA 传输，其外设地址固定为：DCMI->DR，而存储器地址可变（LCD 或者 SRAM）。DMA 被配置为循环模式，一旦开启，DMA 将不停的循环传输数据。My_DCMI_Init 函数用于初始化 STM32F4 的 DCMI 接口，这是根据在 40.1.2 节提到的配置步骤进行配置的。最后，DCMI_Start 和 DCMI_Stop 两个函数，用于开启或停止 DCMI 接口。

其他部分代码我们就不再细说了，请大家参考光盘本例程源码库函数版本（实验 35 摄像头实验）。

最后，打开 main.c 文件，代码如下：

```

u8 ov2640_mode=0; //工作模式:0,RGB565 模式;1,JPEG 模式
#define jpeg_buf_size 31*1024 //定义 JPEG 数据缓存 jpeg_buf 的大小(*4 字节)
__align(4) u32 jpeg_buf[jpeg_buf_size]; //JPEG 数据缓存 buf
volatile u32 jpeg_data_len=0; //buf 中的 JPEG 有效数据长度
volatile u8 jpeg_data_ok=0; //JPEG 数据采集完成标志
//0,数据没有采集完;
//1,数据采集完了,但是还没处理;
//2,数据已经处理完成了,可以开始下一帧接收

//JPEG 尺寸支持列表
const u16 jpeg_img_size_tbl[][2]=
{
    160,120, //QQVGA
    .....//省略部分代码
    1600,1200, //UXGA
};

const u8*EFFECTS_TBL[7]={"Normal","Negative","B&W","Redish","Greenish","Bluish",
"Antique"}; //7 种特效
const u8*JPEG_SIZE_TBL[13]={"QQVGA","QCIF","QVGA","WQVGA","CIF","VGA",
"SVGA","XGA","WXGA","XVGA","WXGA+","SXGA","UXGA"};//JPEG 图片 13 种尺寸
//处理 JPEG 数据

```

//当采集完一帧 JPEG 数据后,调用此函数,切换 JPEG BUF.开始下一帧采集.

```
void jpeg_data_process(void)
{
    if.ov2640_mode)//只有在 JPEG 格式下,才需要做处理.
    {
        if(jpeg_data_ok==0) //jpeg 数据还未采集完?
        {
            DMA2_Stream1->CR&=~(1<<0); //停止当前传输
            while(DMA2_Stream1->CR&0X01); //等待 DMA2_Stream1 可配置
            jpeg_data_len=jpeg_buf_size-DMA2_Stream1->NDTR;//得到此次数据的长度
            jpeg_data_ok=1; //标记 JPEG 数据采集完接成,等待其他函数处理
        }
        if(jpeg_data_ok==2) //上一次的 jpeg 数据已经被处理了
        {
            DMA2_Stream1->NDTR=jpeg_buf_size; //传输长度为 jpeg_buf_size*4 字节
            DMA2_Stream1->CR|=1<<0; //重新传输
            jpeg_data_ok=0; //标记数据未采集
        }
    }
}

//JPEG 测试
//JPEG 数据,通过串口 2 发送给电脑.
void jpeg_test(void)
{
    u32 i; u8 *p; u8 key;
    u8 effect=0,saturation=2,contrast=2;
    u8 size=2; //默认是 QVGA 320*240 尺寸
    u8 msgbuf[15]; //消息缓存区
    .....//省略部分代码
    LCD_ShowString(30,180,200,16,16,msgbuf);//显示当前 JPEG 分辨率
    OV2640_JPEG_Mode();//JPEG 模式
    DCMI_Init(); //DCMI 配置
    DCMI_DMA_Init((u32)&jpeg_buf,jpeg_buf_size,2,1);//DCMI DMA 配置
    OV2640_OutSize_Set(jpeg_img_size_tbl[size][0],jpeg_img_size_tbl[size][1]); //设置尺寸
    DCMI_Start(); //启动传输
    while(1)
    {
        if(jpeg_data_ok==1)//已经采集完一帧图像了
        {
            p=(u8*)jpeg_buf;
            LCD_ShowString(30,210,210,16,16,"Sending JPEG data..."); //提示在传输数据
            for(i=0;i<jpeg_data_len*4;i++) //dma 传输 1 次等于 4 字节,所以乘以 4.
            {

```

```
while((USART2->SR&0X40)==0);//循环发送,直到发送完毕
    USART2->DR=p[i];
    key=KEY_Scan(0);
    if(key)break;
}
if(key)//有按键按下,需要处理
{
    .....//省略部分代码
}else LCD_ShowString(30,210,210,16,16,"Send data complete!!");//提示结束
jpeg_data_ok=2; //标记 jpeg 数据处理完了,可以让 DMA 去采集下一帧了.
}
}
}

//RGB565 测试
//RGB 数据直接显示在 LCD 上面
void rgb565_test(void)
{
    u8 key; u8 effect=0,saturation=2,contrast=2;
    u8 scale=1; //默认是全尺寸缩放
    u8 msgbuf[15]; //消息缓存区
    .....//省略部分代码
    LCD_ShowString(30,170,200,16,16,"KEY_UP:FullSize/Scale");
    OV2640_RGB565_Mode(); //RGB565 模式
    DCMI_Init(); //DCMI 配置
    DCMI_DMA_Init((u32)&LCD->LCD_RAM,1,1,0); //DCMI DMA 配置
    OV2640_OutSize_Set(lcddev.width,lcddev.height);
    DCMI_Start(); //启动传输
    while(1)
    {
        key=KEY_Scan(0);
        if(key)
        {
            .....//省略部分代码
        }
        delay_ms(10);
    }
}

int main(void)
{
    u8 key; u8 t;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    usart2_init(42,115200); //初始化串口 2 波特率为 115200
```

```

LED_Init();           //初始化 LED
LCD_Init();          //LCD 初始化
KEY_Init();          //按键初始化
TIM3_Int_Init(10000-1,8400-1); //10Khz 计数,1 秒钟中断一次
POINT_COLOR=RED; //设置字体为红色
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"OV2640 TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/14");
while(OV2640_Init()) //初始化 OV2640
{
    LCD_ShowString(30,130,240,16,16,"OV2640 ERROR"); delay_ms(200);
    LCD_Fill(30,130,239,170,WHITE); delay_ms(200);
    LED0=!LED0;
}
LCD_ShowString(30,130,200,16,16,"OV2640 OK");
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY0_PRES){ov2640_mode=0; break;} //RGB565 模式
    else if(key==KEY1_PRES) {ov2640_mode=1; break;} //JPEG 模式
    t++;
    if(t==100)LCD_ShowString(30,150,230,16,16,"KEY0:RGB565 KEY1:JPEG");
    if(t==200) {LCD_Fill(30,150,210,150+16,WHITE); t=0; LED0=!LED0;}
    delay_ms(5);
}
if(ov2640_mode)jpeg_test(); //JPEG 测试
else rgb565_test(); //RGB565 测试
}

```

这部分代码比较长，我们省略了一些内容。详细的代码，请大家参考光盘本例程源码。注意，这里定义了一个非常大的数组 jpeg_buf (124KB)，用来存储 JPEG 数据，因为 1600*1200 大小的 jpeg 图片，有可能大于 120K，所以必须将这个数组尽量设置大一点。

在 main.c 里面，总共有 4 个函数：jpeg_data_process、jpeg_test、rgb565_test 和 main 函数。其中，jpeg_data_process 函数用于处理 JPEG 数据的接收，在 DCMI_IRQHandler 函数里面被调用，通过一个 jpeg_data_ok 变量与 jpeg_test 函数共同控制 JPEG 的数据传送。jpeg_test 函数将 OV2640 设置为 JPEG 模式，该函数接收 OV2640 的 JPEG 数据，并通过串口 2 发送给上位机。rgb565_test 函数将 OV2640 设置为 RGB565 模式，并将接收到的数据，直接传送给 LCD，处理过程完全由硬件实现，CPU 完全不用理会。最后，main 函数就相对简单了，大家看代码即可。

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 等相关函数，就可以轻松调试摄像头了。

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 探索者 STM32F4 开发板上，在

OV2640 初始化成功后，屏幕提示选择模式，此时我们可以按 KEY0，进入 RGB565 模式测试，也可以按 KEY1，进入 JPEG 模式测试。

当按 KEY0 后，选择 RGB565 模式，LCD 满屏显示压缩放后的图像（有变形），如图 40.4.1 所示：



图 40.4.1 RGB565 模式测试图片

此时，可以按 KEY_UP 切换为 1:1 显示（不变形）。同时还可以通过 KEY0 按键，设置对比度；KEY1 按键，设置饱和度；KEY2，可以设置特效。

当按 KEY1 后，选择 JPEG 模式，此时屏幕显示 JPEG 数据传输进程，如图 40.4.2 所示：

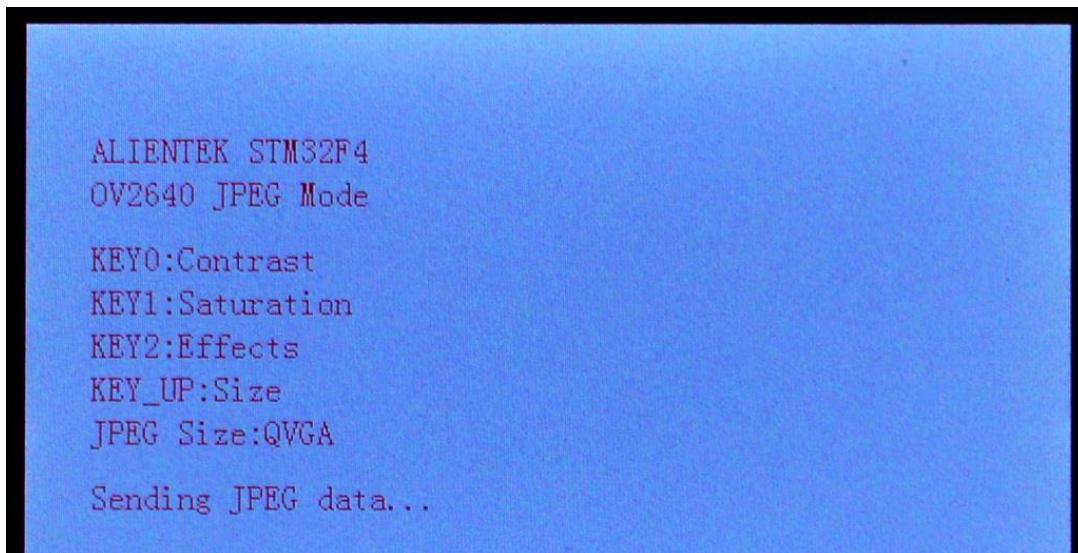


图 40.4.2 JPEG 模式测试图

默认条件下，图像分辨率是 QVGA(320*240)的，硬件上：我们需要一根 RS232 串口线连接开发板的 COM2（注意要用跳线帽将 P9 的：COM2_RX 连接在 PA2(TX)）。如果没有 RS232 线，也可以借助我们开发板板载的 USB 转串口实现（有 2 个办法：1，改代码；2，杜邦线连接 P9 的 PA2(TX)和 P6 的 RX）。

我们打开上位机软件：串口摄像头.exe（路径：光盘→\6，软件资料→软件→串口摄像头软件→串口摄像头.exe），选择正确的串口，然后波特率设置为 115200，打开即可收到下位机传过来的图片了，如图 40.4.3 所示：

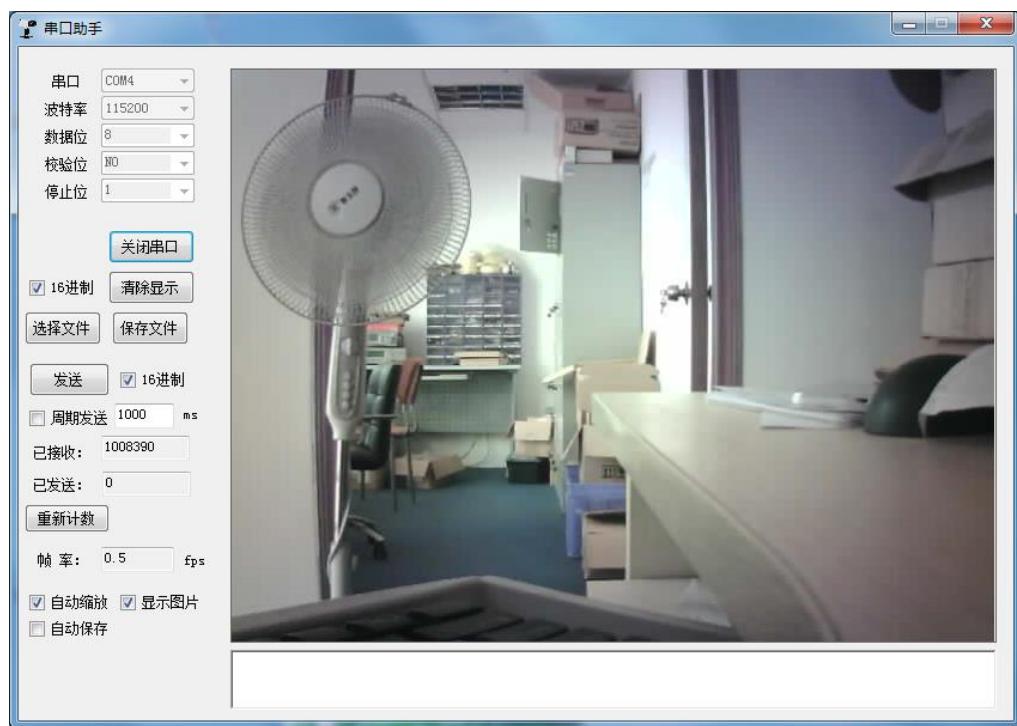


图 40.4.3 串口摄像头软件接收并显示 JPEG 图片

我们可以通过 KEY_UP 设置输出图像的尺寸 (QQVGA~UXGA)。通过 KEY0 按键，设置对比度；KEY1 按键，设置饱和度；KEY2 按键，设置特效。

同时，你还可以在串口，通过 USMART 调用 SCCB_WR_Reg 等函数，来设置 OV2640 的各寄存器，达到调试测试 OV2640 的目的，如图 40.4.4 所示：

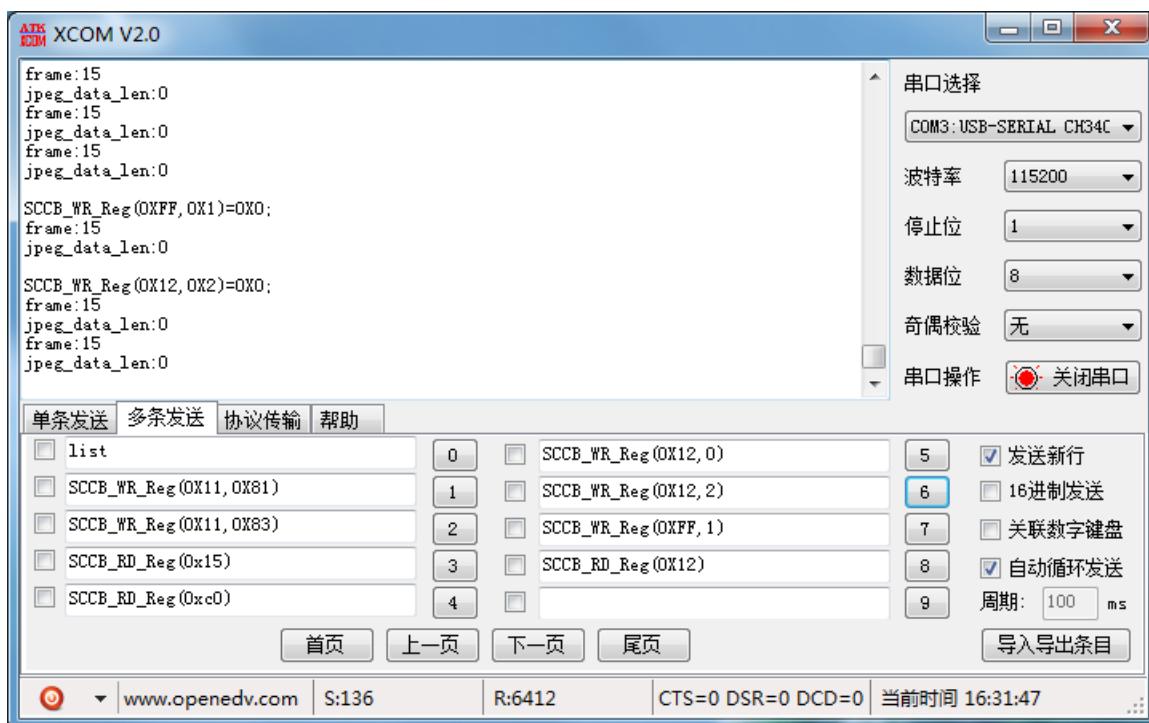


图 40.4.4 USMART 调试 OV2640

从上图还可以看出，帧率为 15 帧，这和我们前面介绍的 OV2640 在 UXGA 模式，输出帧率是 15 帧是一致的。

第四十一章 外部 SRAM 实验

STM32F407ZGT6 自带了 192K 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，STM32F4 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，所以探索者 STM32F4 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 STM32F4 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：

41.1 IS62WV51216 简介

41.2 硬件设计

41.3 软件设计

41.4 下载验证

41.1 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 41.1.1 所示：

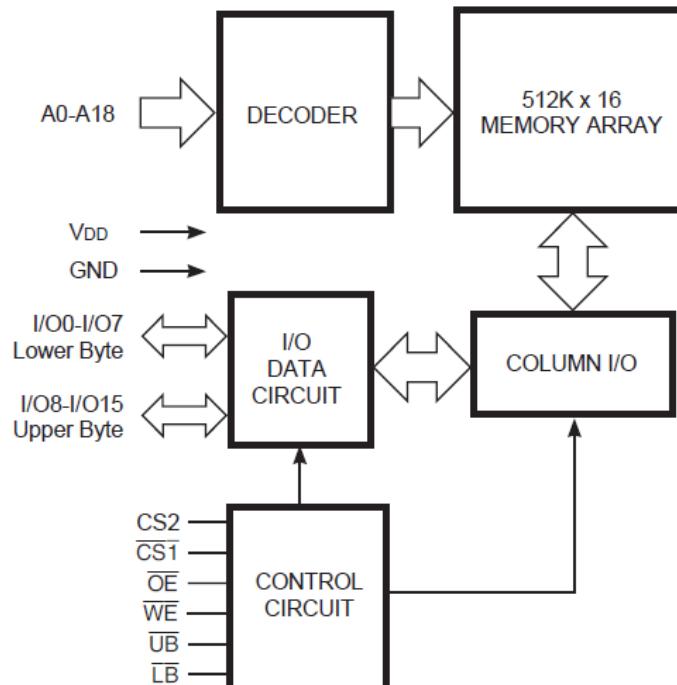


图 41.1.1 IS62WV51216 功能框图

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；

探索者 STM32F4 开发板使用的是 TSOP44 封装的 IS62WV51216 芯片，该芯片直接接在 STM32F4 的 FSMC 上，IS62WV51216 原理图如图 41.1.2 所示：

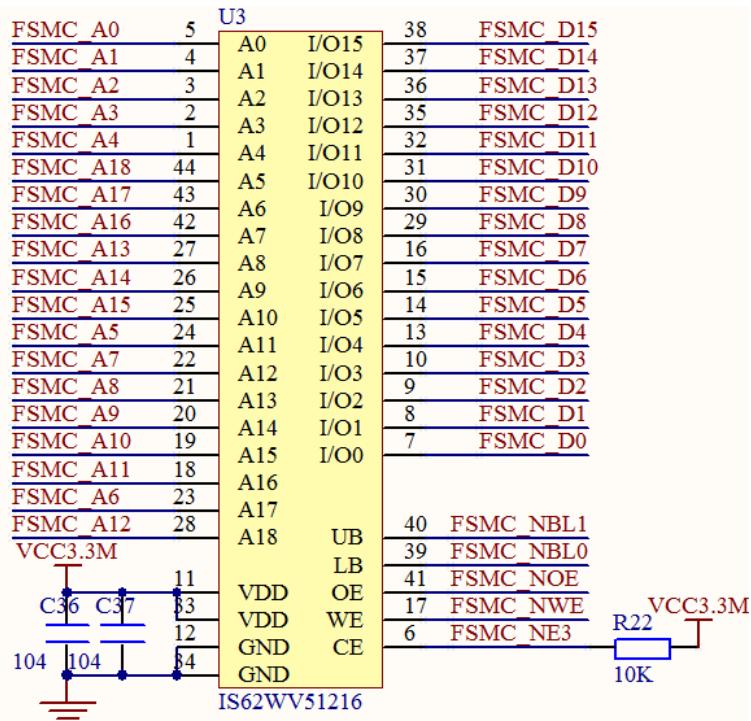


图 41.1.2 IS62WV51216 原理图

从原理图可以看出，IS62WV51216 同 STM32F4 的连接关系：

A[0:18]接 FSMC_A[0:18]（不过顺序错乱了）

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

上面的连接关系，IS62WV51216 的 A[0:18]并不是按顺序连接 STM32F4 的 FSMC_A[0:18]，不过这并不影响我们正常使用外部 SRAM，因为地址具有唯一性。所以，只要地址线不和数据线混淆，就可以正常使用外部 SRAM。这样设计的好处，就是可以方便我们的 PCB 布线。

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《STM32F4xx 中文参考手册》。

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 时钟的方法前面 LCD 实验已经讲解过，方法为：

```
RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC,ENABLE); //使能 FSMC 时钟
```

配置 IO 口为复用输出的关键行代码为：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用输出
```

关于引脚复用映射配置，这在 LCD 实验章节也讲解非常详细，调用函数为：

```
void GPIO_PinAFConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF);
```

针对每个复用引脚调用这个函数即可，例如 GPIOD.0 引脚复用映射配置方法为：

```
GPIO_PinAFConfig(GPIOD,GPIO_PinSource0,GPIO_AF_FSMC); //PD0,AF12
```

2) 设置 FSMC BANK1 区域 3 的相关寄存器。

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。这个是通过调用函数 `FSMC_NORSRAMInit` 来实现的，函数原型为：

```
void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct);
```

3) 使能 BANK1 区域 3。

最后，只需要通过 `FSMC_BCR` 寄存器使能 BANK1，区域 3 即可。使能方法为：

```
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM3, ENABLE); //使能 BANK3
```

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`。

41.2 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 `KEY0` 按键，即测试外部 SRAM 容量大小并显示在 LCD 上。按下 `KEY1` 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) `KEY0` 和 `KEY1` 按键
- 3) 串口
- 4) TFTLCD 模块
- 5) IS62WV51216

这些我们都已经介绍过 (IS62WV51216 与 STM32F4 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。

41.3 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 `stm32f4xx_fsmc.c` 和 `stm32f4xx_fsmc.h` 文件。

打开 `sram.c` 文件，代码如下：

```
//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)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    FSMC_NORSRAMInitTypeDef FSMC_NORSRAMInitStructure;
    FSMC_NORSRAMTimingInitTypeDef readWriteTiming;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOD|
                           RCC_AHB1Periph_GPIOE|RCC_AHB1Periph_GPIOF|RCC_AHB1Periph_GPIOG,
                           ENABLE);//使能 PD,PE,PF,PG 时钟
    RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC,ENABLE);//使能 FSMC 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;//PB15 推挽输出,控制背光
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化 //PB15 推挽输出,控制背光

    GPIO_InitStructure.GPIO_Pin = (3<<0)|(3<<4)|(0xFF<<8);//PD0,1,4,5,8~15 AF OUT
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOD, &GPIO_InitStructure);//初始化

    .....//省略部分 GPIO 初始化设置

    GPIO_PinAFConfig(GPIOD,GPIO_PinSource0,GPIO_AF_FSMC);//PD0,AF12
    GPIO_PinAFConfig(GPIOD,GPIO_PinSource1,GPIO_AF_FSMC);//PD1,AF12
    .....//省略部分 GPIO AF 映射设置
    GPIO_PinAFConfig(GPIOG,GPIO_PinSource5,GPIO_AF_FSMC);
    GPIO_PinAFConfig(GPIOG,GPIO_PinSource10,GPIO_AF_FSMC);

    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 = 0x08; //数据保持时间为 9 个 HCLK
    readWriteTiming.FSMC_BusTurnAroundDuration = 0x00;
    readWriteTiming.FSMC_CLKDivision = 0x00;
    readWriteTiming.FSMC_DataLatency = 0x00;
```

```
readWriteTiming.FSMC_AccessMode = FSMC_AccessMode_A; //模式 A

FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM3;// NE3
FSMC_NORSRAMInitStructure.FSMC_DataAddressMux =
    FSMC_DataAddressMux_Disable;
FSMC_NORSRAMInitStructure.FSMC_MemoryType=FSMC_MemoryType_SRAM;
FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth =
    FSMC_MemoryDataWidth_16b;//存储器数据宽度为 16bit
FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode =
    FSMC_BurstAccessMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity =
    FSMC_WaitSignalPolarity_Low;
FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait=
    FSMC_AsynchronousWait_Disable;
FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive =
    FSMC_WaitSignalActive_BeforeWaitState;
FSMC_NORSRAMInitStructure.FSMC_WriteOperation =
    FSMC_WriteOperation_Enable;//存储器写使能
FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
FSMC_NORSRAMInitStructure.FSMC_ExtendedMode =
    FSMC_ExtendedMode_Disable; // 读写使用相同的时序
FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &readWriteTiming;
FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct =
    &readWriteTiming; //读写同样时序

FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure); //初始化 FSMC 配置
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM3, ENABLE); // 使能 BANK1 区域 3
}
//在指定地址(WriteAddr+Bank1_SRAM3_ADDR)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++;
        pBuffer++;
    }
}
```

//在指定地址((WriteAddr+Bank1_SRAM3_ADDR))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++;
```

```
}
```

```
}
```

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; 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 (字节数)。

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。

头文件 sram.h 内容比较简洁, 主要是一些函数申明, 这里我们不做过多讲解。

最后我们来看看 main.c 文件代码如下:

```
u32 testsram[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;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
```

```
delay_init(168); //初始化延时函数
uart_init(115200); //初始化串口波特率为 115200
LED_Init(); //初始化 LED
LCD_Init(); //LCD 初始化
KEY_Init(); //按键初始化
FSMC_SRAM_Init(); //初始化外部 SRAM
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"SRAM TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/5/14");
LCD_ShowString(30,130,200,16,16,"KEY0:Test Sram");
LCD_ShowString(30,150,200,16,16,"KEY1: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==KEY0_PRES)fsmc_sram_test(60,170);//测试 SRAM 容量
    else if(key==KEY1_PRES)//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。

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 探索者 STM32F4 开发板上，得到如图 41 在代码编译成功之后，我们通过下载代码到 ALIENTEK 探索者 STM32F4 开发板上，

得到如图 41.4.1 所示界面：

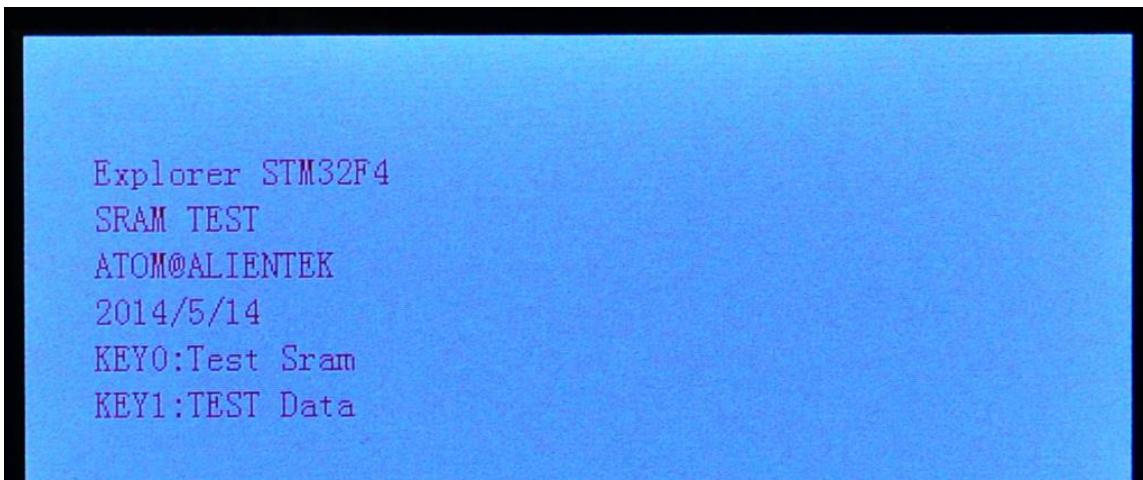


图 41.4.1 程序运行效果图

此时，我们按下 KEY0，就可以在 LCD 上看到内存测试的画面，同样，按下 KEY1，就可以看到 LCD 显示存放在数组 testsram 里面的测试数据，如图 41.4.2 所示：

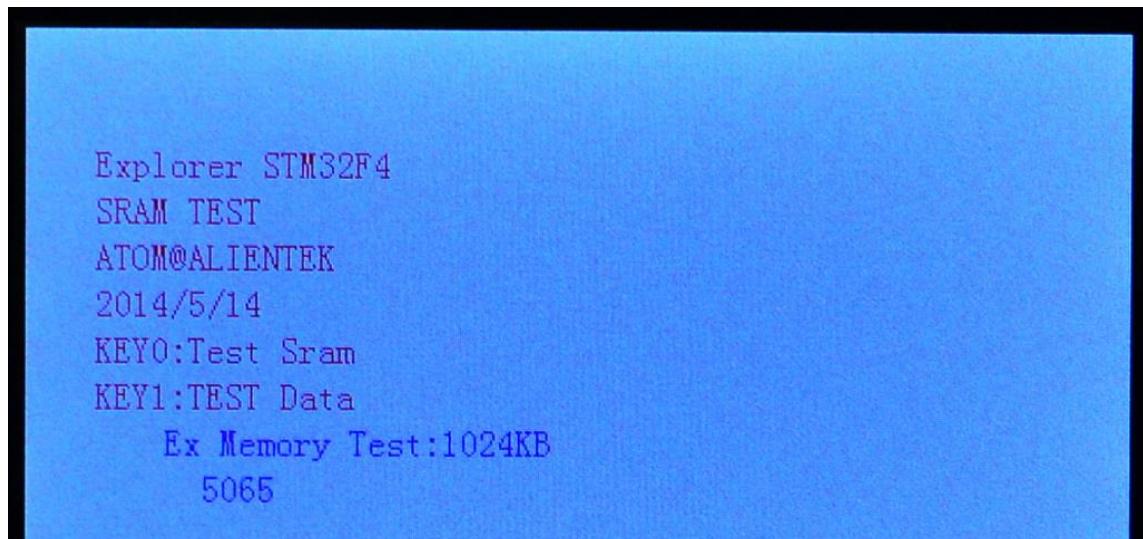


图 41.4.2 外部 SRAM 测试界面

第四十二章 内存管理实验

上一章，我们学会了使用 STM32F4 驱动外部 SRAM，以扩展 STM32F4 的内存，加上 STM32F4 本身自带的 192K 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。本章分为以下几个部分：

- 42.1 内存管理简介
- 42.2 硬件设计
- 42.3 软件设计
- 42.4 下载验证

42.1 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 42.1.1 所示：

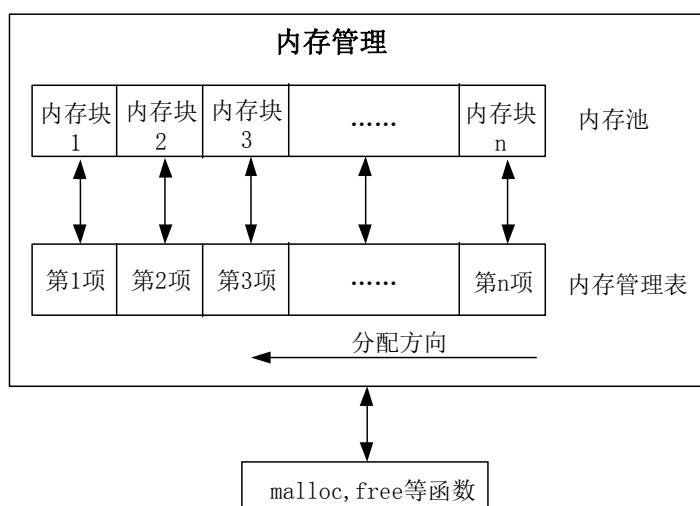


图 42.1.1 分块式内存管理原理

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。

42.2 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。KEY_UP 用于切换操作内存区（内部 SRAM 内存/外部 SRAM 内存/内部 CCM 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。

42.3 软件设计

打开本章实验工程可以看到，我们新增了 MALLOC 分组，同时在分组中新建了文件 malloc.c 以及头文件 malloc.h。内存管理相关的函数和定义主要是在这两个文件中。

打开 malloc.c 文件，代码如下：

```
//内存池(32 字节对齐)
__align(32) u8 mem1base[MEM1_MAX_SIZE]; //内部 SRAM 内存池
__align(32) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0X68000000)));
//外部 SRAM 内存池
__align(32) u8 mem3base[MEM3_MAX_SIZE] __attribute__((at(0X10000000)));
//内部 CCM 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
u16 mem3mapbase[MEM3_ALLOC_TABLE_SIZE] __attribute__((at(0X10000000
+MEM3_MAX_SIZE)));    //内部 CCM 内存池 MAP
//内存管理参数
```

```
const u32 memtblsize[SRAMBANK]={MEM1_ALLOC_TABLE_SIZE,
MEM2_ALLOC_TABLE_SIZE, MEM3_ALLOC_TABLE_SIZE}; //内存表大小
const u32 memblksize[SRAMBANK]={MEM1_BLOCK_SIZE, MEM2_BLOCK_SIZE,
MEM3_BLOCK_SIZE}; //内存分块大小
const u32 memsize[SRAMBANK]={MEM1_MAX_SIZE, MEM2_MAX_SIZE,
MEM3_MAX_SIZE}; //内存总大小
//内存管理控制器
struct _m_mallco_dev mallco_dev=
{
    my_mem_init, //内存初始化
    my_mem_perused, //内存使用率
    mem1base,mem2base,mem3base, //内存池
    mem1mapbase,mem2mapbase,mem3mapbase, //内存管理状态表
    0,0,0, //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 my_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 my_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 my_mem_malloc(u8 memx,u32 size)
{
    signed long offset=0;
    u32 nmemb; //需要的内存块数
    u32 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/membblksize[memx]; //获取需要分配的连续内存块数
    if(size%membblksize[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*membblksize[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 my_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);      //初始化内存池
    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++) mallco_dev.memmap[memx][index+i]=0;//内存块清零
    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];
    my_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=my_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=my_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），实现对三个内存池的管理控制。为甚

首先，是内部 SRAM 内存池，定义为：

```
__align(32) u8 mem1base[MEM1_MAX_SIZE];
```

然后，是外部 SRAM 内存池，定义为：

```
__align(32) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0X68000000)));
```

最后，是内部 CCM 内存池，定义为：

```
__align(32) u8 mem3base[MEM3_MAX_SIZE] __attribute__((at(0X10000000)));
```

这里之所以要定义成 3 个，是因为这三个内存区域的地址都不一样，STM32F4 内部内存分为两大块：1，普通内存（又分为主要内存和辅助内存，地址从：0X2000 0000 开始，共 128KB），这部分内存任何外设都可以访问。2，CCM 内存（地址从：0X1000 0000 开始，共 64KB），这部分内存仅 CPU 可以访问，DMA 之类的不可以直接访问，使用时得特别注意！！

而外部 SRAM，地址是从 0X6800 0000 开始的，共 1024KB。所以，这样总共有 3 部分内存，而内存池必须是连续的内存空间，才可以，这样 3 个内存区域，就有 3 个内存池，因此，分成了 3 块来管理。

其中，MEM1_MAX_SIZE、MEM2_MAX_SIZE 和 MEM3_MAX_SIZE 为在 malloc.h 里面定义的内存池大小，外部 SRAM 内存池指定地址为 0X6800 0000，也就是从外部 SRAM 的首地址开始的，CCM 内存池从 0X1000 0000 开始，同样是从 CCM 内存的首地址开始的。但是，内部 SRAM 内存池的首地址则由编译器自动分配。`__align(32)` 定义内存池为 32 字节对齐，以适应各种不同场合的需求。

此部分代码的核心函数为：my_mem_malloc 和 my_mem_free，分别用于内存申请和内存释放。思路就是我们在 42.1 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，代码如下：

```

#ifndef __MALLOC_H
#define __MALLOC_H
#include "stm32f4xx.h"#ifndef NULL
#define NULL 0
#endif
//定义三个内存池
#define SRAMIN      0      //内部内存池
#define SRAMEX     1      //外部内存池
#define SRAMCCM   2      //CCM 内存池(此部分 SRAM 仅仅 CPU 可以访问!!!)

```

```
#define SRAMBANK 3           //定义支持的 SRAM 块数.  
//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        100*1024     //最大管理内存 100K  
#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  
#define MEM2_BLOCK_SIZE      32           //内存块大小为 32 字节  
#define MEM2_MAX_SIZE        960 *1024    //最大管理内存 960K  
#define MEM2_ALLOC_TABLE_SIZE MEM2_MAX_SIZE/MEM2_BLOCK_SIZE  
//内存表大小  
//mem3 内存参数设定.mem3 处于 CCM,用于管理 CCM(特别注意,这部分 SRAM,仅 CPU 可  
//以访问!!)  
#define MEM3_BLOCK_SIZE      32           //内存块大小为 32 字节  
#define MEM3_MAX_SIZE        60 *1024     //最大管理内存 60K  
#define MEM3_ALLOC_TABLE_SIZE MEM3_MAX_SIZE/MEM3_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[SRAMBANK];     //内存池 管理 SRAMBANK 个区域的内存  
    u16 *memmap[SRAMBANK];     //内存管理状态表  
    u8 memrdy[SRAMBANK];       //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 my_mem_init(u8 memx);           //内存管理初始化函数(外/内部调用)  
u32 my_mem_malloc(u8 memx,u32 size); //内存分配(内部调用)  
u8 my_mem_free(u8 memx,u32 offset); //内存释放(内部调用)  
u8 my_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、MEM2_BLOCK_SIZE 和 MEM3_BLOCK_SIZE，都是 32 字节。内存池总大小，内部 SRAM 内存池大小为 100K，外部 SRAM 内存池大小为 960K，内部 CCM 内存池大小为 60K。

MEM1_ALLOC_TABLE_SIZE、MEM2_ALLOC_TABLE_SIZE 和 MEM3_ALLOC_TABLE_

SIZE，则分别代表内存池 1、2 和 3 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。接下来我们看看主函数代码：

```
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
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200);      //初始化串口波特率为 115200
    LED_Init();              //初始化 LED
    LCD_Init();              //LCD 初始化
    KEY_Init();              //按键初始化
    FSMC_SRAM_Init();        //初始化外部 SRAM
    my_mem_init(SRAMIN);     //初始化内部内存池
    my_mem_init(SRAMEX);     //初始化外部内存池
    my_mem_init(SRAMCCM);   //初始化 CCM 内存池
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"MALLOC TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/15");
    LCD_ShowString(30,130,200,16,16,"KEY0:Malloc KEY2:Free");
    LCD_ShowString(30,150,200,16,16,"KEY_UP:SRAMx KEY1:Read");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,170,200,16,16,"SRAMIN");
    LCD_ShowString(30,190,200,16,16,"SRAMIN USED: %");
    LCD_ShowString(30,210,200,16,16,"SRAMEX USED: %");
    LCD_ShowString(30,230,200,16,16,"SRAMCCM USED: %");
    while(1)
    {
        key=KEY_Scan(0);//不支持连接
        switch(key)
        {
            case 0://没有按键按下
                break;
            case KEY0_PRES: //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 写入内容
                break;
        }
    }
}
```

```
case KEY1_PRES: //KEY1 按下
    if(p!=NULL)
    {
        sprintf((char*)p,"Memory Malloc Test%03d",i);//更新显示内容
        LCD_ShowString(30,270,200,16,16,p); //显示 P 的内容
    }
    break;
case KEY2_PRES: //KEY2 按下
    myfree(sramx,p); //释放内存
    p=0; //指向空地址
    break;
case WKUP_PRES: //KEY UP 按下
    sramx++;
    if(sramx>2)sramx=0;
    if(sramx==0)LCD_ShowString(30,170,200,16,16,"SRAMIN ");
    else if(sramx==1)LCD_ShowString(30,170,200,16,16,"SRAMEX ");
    else LCD_ShowString(30,170,200,16,16,"SRAMCCM");
    break;
}
if(tp!=p)
{
    tp=p;
    sprintf((char*)paddr,"P Addr:0X%08X",(u32)tp);
    LCD_ShowString(30,250,200,16,16,paddr); //显示 p 的地址
    if(p)LCD_ShowString(30,270,200,16,16,p); //显示 P 的内容
    else LCD_Fill(30,270,239,266,WHITE); //p=0,清除显示
}
delay_ms(10);
i++;
if((i%20)==0)//DS0 闪烁.
{
    LCD_ShowNum(30+104,190,my_mem_perused(SRAMIN),3,16); //显示使用率
    LCD_ShowNum(30+104,210,my_mem_perused(SRAMEX),3,16); //显示使用率
    LCD_ShowNum(30+104,230,my_mem_perused(SRAMCCM),3,16); //显示使用率
    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。

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 探索者 STM32F4 开发板上，得到如图 42.4.1 所示界面：

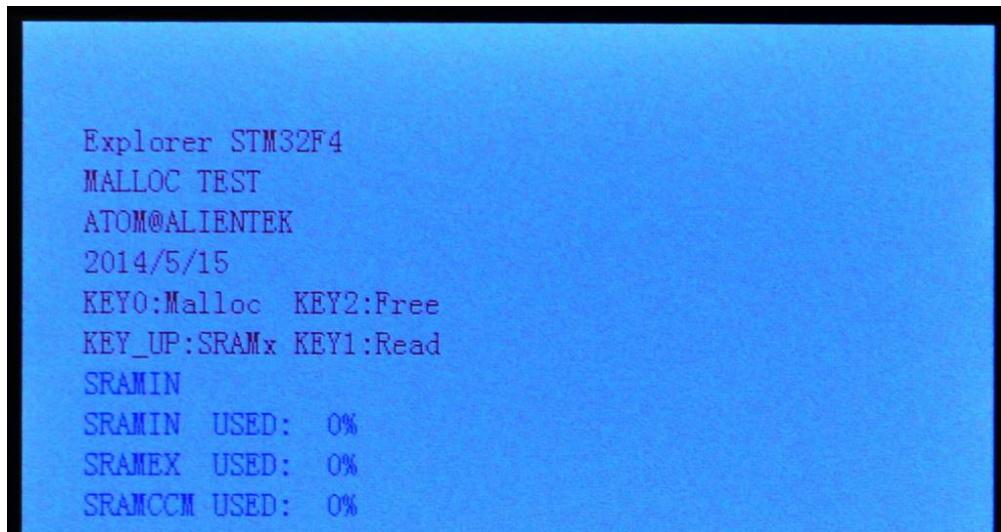


图 42.4.1 程序运行效果图

可以看到，所有内存的使用率均为 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 SRAM 内存被使用 2% 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 2%，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 SRAM 内存/外部 SRAM 内存/内部 CCM 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。

第四十三章 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），支持 SPI/SDIO 驱动，而且有多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。

只需要少数几个 IO 口即可外扩一个高达 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。

ALIENTEK 探索者 STM32F4 开发板自带了标准的 SD 卡接口，使用 STM32F4 自带的 SDIO 接口驱动，4 位模式，最高通信速度可达 48Mhz（分频器旁路时），最高每秒可传输数据 24M 字节，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 探索者 STM32F4 开发板上实现 SD 卡的读取。本章分为如下几个部分：

43.1 SDIO 接口简介

43.2 硬件设计

43.3 软件设计

43.4 下载验证

43.1 SDIO 简介

ALIENTEK 探索者 STM32F4 开发板自带 SDIO 接口，本节，我们将简单介绍 STM32F4 的 SDIO 接口，包括：主要功能及框图、时钟、命令与响应和相关寄存器简介等，最后，我们将介绍 SD 卡的初始化流程。

43.1.1 SDIO 主要功能及框图

STM32F4 的 SDIO 控制器支持多媒体卡（MMC 卡）、SD 存储卡、SD I/O 卡和 CE-ATA 设备等。SDIO 的主要功能如下：

- 与多媒体卡系统规格书版本 4.2 全兼容。支持三种不同的数据总线模式：1 位（默认）、4 位和 8 位。
- 与较早的多媒体卡系统规格版本全兼容（向前兼容）。
- 与 SD 存储卡规格版本 2.0 全兼容。
- 与 SD I/O 卡规格版本 2.0 全兼容：支持良种不同的数据总线模式：1 位（默认）和 4 位。
- 完全支持 CE-ATA 功能（与 CE-ATA 数字协议版本 1.1 全兼容）。8 位总线模式下数据传输速率可达 48MHz（分频器旁路时）。
- 数据和命令输出使能信号，用于控制外部双向驱动器。

STM32F4 的 SDIO 控制器包含 2 个部分：SDIO 适配器模块和 APB2 总线接口，其功能框图如图 43.1.1.1 所示：

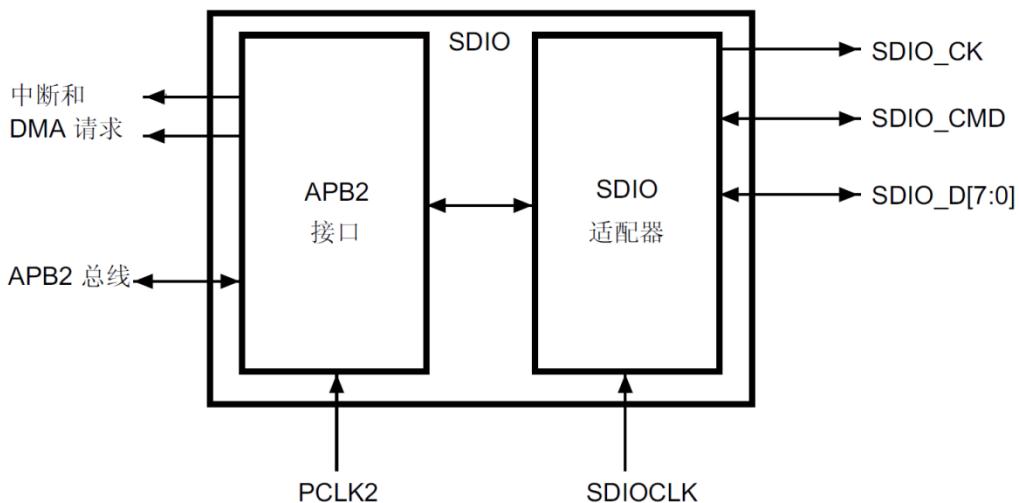


图 43.1.1.1 STM32F4 的 SDIO 控制器功能框图

复位后默认情况下 SDIO_D0 用于数据传输。初始化后主机可以改变数据总线的宽度（通过 ACMD6 命令设置）。

如果一个多媒体卡接到了总线上，则 SDIO_D0、SDIO_D[3:0]或 SDIO_D[7:0]可以用于数据传输。MMC 版本 V3.31 和之前版本的协议只支持 1 位数据线，所以只能用 SDIO_D0（为了通用性考虑，在程序里面我们只要检测到是 MMC 卡就设置为 1 位总线数据）。

如果一个 SD 或 SD I/O 卡接到了总线上，可以通过主机配置数据传输使用 SDIO_D0 或 SDIO_D[3:0]。所有的数据线都工作在推挽模式。

SDIO_CMD 有两种操作模式：

- ① 用于初始化时的开路模式(仅用于 MMC 版本 V3.31 或之前版本)
- ② 用于命令传输的推挽模式(SD/SD I/O 卡和 MMC V4.2 在初始化时也使用推挽驱动)

43.1.2 SDIO 的时钟

从图 43.1.1.1 我们可以看到 SDIO 总共有 3 个时钟，分别是：

卡时钟 (SDIO_CK): 每个时钟周期在命令和数据线上传输 1 位命令或数据。对于多媒体卡 V3.31 协议，时钟频率可以在 0MHz 至 20MHz 间变化；对于多媒体卡 V4.0/4.2 协议，时钟频率可以在 0MHz 至 48MHz 间变化；对于 SD 或 SD I/O 卡，时钟频率可以在 0MHz 至 25MHz 间变化。

SDIO 适配器时钟 (SDIOCLK): 该时钟用于驱动 SDIO 适配器，来自 PLL48CK，一般为 48Mhz，并用于产生 SDIO_CK 时钟。

APB2 总线接口时钟(PCLK2): 该时钟用于驱动 SDIO 的 APB2 总线接口，其频率为 HCLK/2，一般为 84Mhz。

前面提到，我们的 SD 卡时钟 (SDIO_CK)，根据卡的不同，可能有几个区间，这就涉及到时钟频率的设置，SDIO_CK 与 SDIOCLK 的关系（时钟分频器不旁路时）为：

$$\text{SDIO_CK} = \text{SDIOCLK} / (2 + \text{CLKDIV})$$

其中，SDIOCLK 为 PLL48CK，一般是 48Mhz，而 CLKDIV 则是分配系数，可以通过 SDIO 的 SDIO_CLKCR 寄存器进行设置（确保 SDIO_CK 不超过卡的最大操作频率）。注意，以上公式，是时钟分频器不旁路时的计算公式，当时钟分频器旁路时，SDIO_CK 直接等于 SDIOCLK。

这里要提醒大家，在 SD 卡刚刚初始化的时候，其时钟频率 (SDIO_CK) 是不能超过 400Khz 的，否则可能无法完成初始化。在初始化以后，就可以设置时钟频率到最大了（但不可超过 SD

卡的最大操作时钟频率)。

43.1.3 SDIO 的命令与响应

SDIO 的命令分为应用相关命令(ACMD)和通用命令(CMD)两部分,应用相关命令(ACMD)的发送,必须先发送通用命令(CMD55),然后才能发送应用相关命令(ACMD)。

SDIO 的所有命令和响应都是通过 SDIO_CMD 引脚传输的,任何命令的长度都是固定为 48 位,SDIO 的命令格式如表 43.1.3.1 所示:

位的位置	宽度	值	说明
47	1	0	起始位
46	1	1	传输位
[45:40]	6	-	命令索引
[39:8]	32	-	参数
[7:1]	7	-	CRC7
0	1	1	结束位

表 43.1.3.1 SDIO 命令格式

所有的命令都是由 STM32F4 发出,其中开始位、传输位、CRC7 和结束位由 SDIO 硬件控制,我们需要设置的就只有命令索引和参数部分。其中命令索引(如 CMD0, CMD1 之类的)在 SDIO_CMD 寄存器里面设置,命令参数则由寄存器 SDIO_ARG 设置。

一般情况下,选中的 SD 卡在接收到命令之后,都会回复一个应答(注意 CMD0 是无应答的),这个应答我们称之为响应,响应也是在 CMD 线上串行传输的。STM32F4 的 SDIO 控制器支持 2 种响应类型,即:短响应(48 位)和长响应(136 位),这两种响应类型都带 CRC 错误检测(注意不带 CRC 的响应应该忽略 CRC 错误标志,如 CMD1 的响应)。

短响应的格式如表 43.1.3.2 所示:

位的位置	宽度	值	说明
47	1	0	起始位
46	1	0	传输位
[45:40]	6	-	命令索引
[39:8]	32	-	参数
[7:1]	7	-	CRC7 (或 1111111)
0	1	1	结束位

表 43.1.3.2 SDIO 命令格式

长响应的格式如表 43.1.3.3 所示:

位的位置	宽度	值	说明
135	1	0	起始位
134	1	0	传输位
[133:128]	6	111111	保留
[127:1]	127	-	CID 或 CSD (包括内部 CRC7)
0	1	1	结束位

表 43.1.3.3 SDIO 命令格式

同样，硬件为我们滤除了开始位、传输位、CRC7 以及结束位等信息，对于短响应，命令索引存放在 SDIO_RESPCMD 寄存器，参数则存放在 SDIO_RESP1 寄存器里面。对于长响应，则仅留 CID/CSD 位域，存放在 SDIO_RESP1~SDIO_RESP4 等 4 个寄存器。

SD 存储卡总共有 5 类响应 (R1、R2、R3、R6、R7)，我们这里以 R1 为例简单介绍一下。R1 (普通响应命令) 响应输入短响应，其长度为 48 位，R1 响应的格式如表 43.1.3.4 所示：

位的位置	宽度 (位)	值	说明
47	1	0	起始位
46	1	0	传输位
[45:40]	6	X	命令索引
[39:8]	32	X	卡状态
[7:1]	7	X	CRC7
0	1	1	结束位

表 43.1.3.4 R1 响应格式

在收到 R1 响应后，我们可以从 SDIO_RESPCMD 寄存器和 SDIO_RESP1 寄存器分别读出命令索引和卡状态信息。关于其他响应的介绍，请大家参考光盘：《SD 卡 2.0 协议.pdf》或《STM32F4xx 中文参考手册》第 28 章。

最后，我们看看数据在 SDIO 控制器与 SD 卡之间的传输。对于 SDI/SDIO 存储器，数据是以数据块的形式传输的，而对于 MMC 卡，数据是以数据块或者数据流的形式传输。本节我们只考虑数据块形式的数据传输。

SDIO (多) 数据块读操作，如图 43.1.3.1 所示：

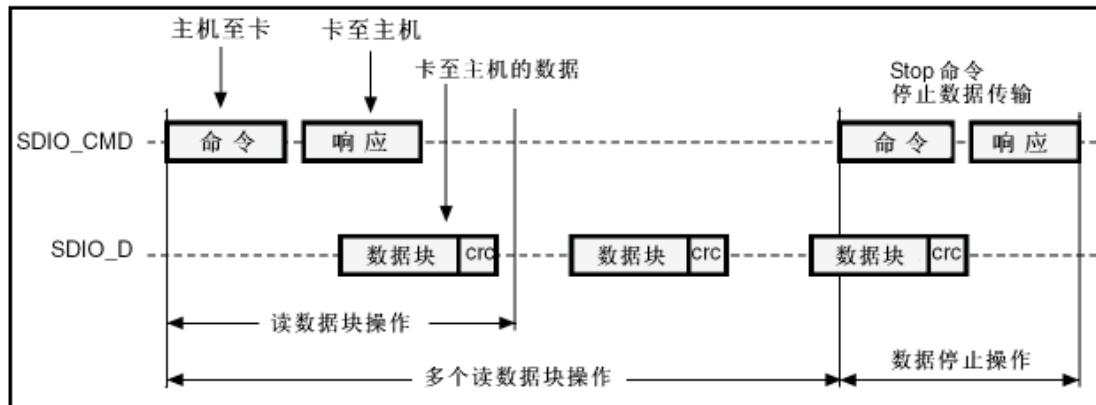


图 43.1.3.1 SDIO (多) 数据块读操作

从上图，我们可以看出，从机在收到主机相关命令后，开始发送数据块给主机，所有数据块都带有 CRC 校验值（CRC 由 SDIO 硬件自动处理），单个数据块读的时候，在收到 1 个数据块以后即可以停止了，不需要发送停止命令（CMD12）。但是多块数据读的时候，SD 卡将一直发送数据给主机，直到接到主机发送的 STOP 命令（CMD12）。

SDIO（多）数据块写操作，如图 43.1.3.2 所示：

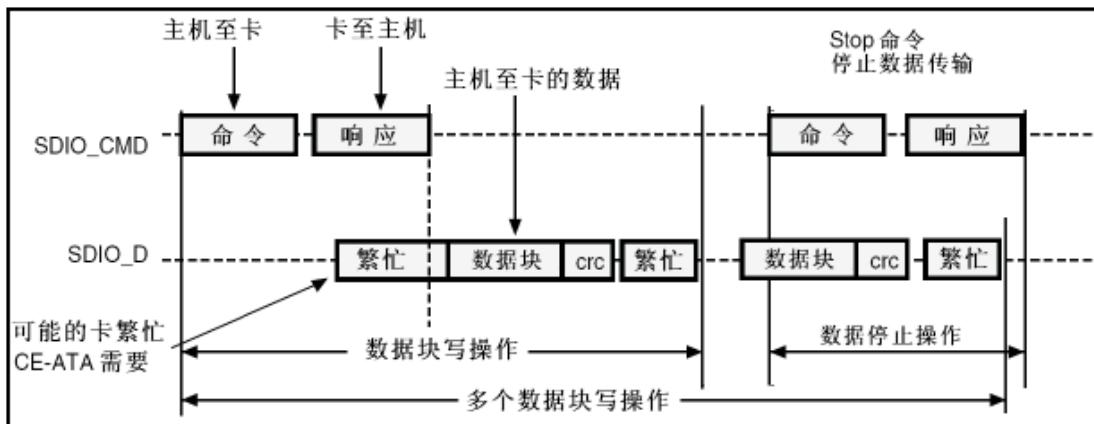


图 43.1.3.2 SDIO (多) 数据块写操作

数据块写操作同数据块读操作基本类似，只是数据块写的时候，多了一个繁忙判断，新的数据块必须在 SD 卡非繁忙的时候发送。这里的繁忙信号由 SD 卡拉低 SDIO_D0，以表示繁忙，SDIO 硬件自动控制，不需要我们软件处理。

SDIO 的命令与响应就为大家介绍到这里。

43.1.4 SDIO 相关寄存器介绍

第一个，我们来看 SDIO 电源控制寄存器（SDIO_POWER），该寄存器定义如图 43.1.4.1 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
Reserved																														PWRCTRL	
																														rw	rw

位 31:2 保留，必须保持复位值

位 1:0 PWRCTRL：电源控制位（Power supply control bits）。

这些位用于定义卡时钟的当前功能状态：

- | | |
|------------------|----------------|
| 00: 掉电；停止为卡提供时钟。 | 10: 保留，上电 |
| 01: 保留 | 11: 通电；为卡提供时钟。 |

图 43.1.4.1 SDIO_POWER 寄存器位定义

该寄存器复位值为 0，所以 SDIO 的电源是关闭的，我们要启用 SDIO，第一步就是要设置该寄存器最低 2 个位均为 1，让 SDIO 上电，开启卡时钟。

第二个，我们看 SDIO 时钟控制寄存器（SDIO_CLKCR），该寄存器主要用于设置 SDIO_CK 的分配系数，开关等，并可以设置 SDIO 的数据位宽，该寄存器的定义如图 43.1.4.2 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

位 12:11 **WIDBUS:** 宽总线模式使能位 (Wide bus mode enable bit)

- 00: 默认总线模式: 使用 SDIO_D0
- 01: 4 位宽总线模式: 使用 SDIO_D[3:0]
- 10: 8 位宽总线模式: 使用 SDIO_D[7:0]

位 10 **BYPASS:** 时钟分频器旁路使能位 (Clock divider bypass enable bit)

- 0: 禁止旁路: 在驱动 SDIO_CK 输出信号前, 根据 CLKDIV 值对 SDIOCLK 进行分频。
- 1: 使能旁路: SDIOCLK 直接驱动 SDIO_CK 输出信号。

位 8 **CLKEN:** 时钟使能位 (Clock enable bit)

- 0: 禁止 SDIO_CK
- 1: 使能 SDIO_CK

位 7:0 **CLKDIV:** 时钟分频系数 (Clock divide factor)

该字段定义输入时钟 (SDIOCLK) 与输出时钟 (SDIO_CK) 之间的分频系数:
SDIO_CK 频率 = SDIOCLK / [CLKDIV + 2]。

图 43.1.4.2 SDIO_CLKCR 寄存器位定义

上图仅列出了部分我们要用到的位设置, WIDBUS 用于设置 SDIO 总线位宽, 正常使用的时候, 设置为 1, 即 4 位宽度。BYPASS 用于设置分频器是否旁路, 我们一般要使用分频器, 所以这里设置为 0, 禁止旁路。CLKEN 则用于设置是否使能 SDIO_CK, 我们设置为 1。最后, CLKDIV, 则用于控制 SDIO_CK 的分频, 一般设置为 0, 即可得到 24Mhz 的 SDIO_CK 频率。

第三个, 我们要介绍的是 SDIO 参数制寄存器 (SDIO_ARG), 该寄存器比较简单, 就是一个 32 位寄存器, 用于存储命令参数, 不过需要注意的是, 必须在写命令之前先写这个参数寄存器!

第四个, 我们要介绍的是 SDIO 命令响应寄存器 (SDIO_RESPCMD), 该寄存器为 32 位, 但只有低 6 位有效, 比较简单, 用于存储最后收到的命令响应中的命令索引。如果传输的命令响应不包含命令索引, 则该寄存器的内容不可预知。

第五个, 我们要介绍的是 SDIO 响应寄存器组 (SDIO_RESP1~SDIO_RESP4), 该寄存器组总共由 4 个 32 位寄存器组成, 用于存放接收到的卡响应部分信息。如果收到短响应, 则数据存放在 SDIO_RESP1 寄存器里面, 其他三个寄存器没有用到。而如果收到长响应, 则依次存放在 SDIO_RESP1~SDIO_RESP4 里面, 如表 43.1.4.1 所示:

寄存器	短响应	长响应
SDIO_RESP1	卡状态[31:0]	卡状态 [127:96]
SDIO_RESP2	未使用	卡状态 [95:64]
SDIO_RESP3	未使用	卡状态 [63:32]
SDIO_RESP4	未使用	卡状态 [31:1]0b

表 43.1.4.1 响应类型和 SDIO_RESPx 寄存器

第七个, 我们介绍 SDIO 命令寄存器 (SDIO_CMD), 该寄存器各位定义如图 43.1.4.3 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

位 10 **CPSMEN**: 命令路径状态机 (CPSM) 使能位 (Command path state machine (CPSM) Enable bit)
如果此位置 1，则使能 CPSM。

位 7:6 **WAITRESP**: 等待响应位 (Wait for response bits)

这些位用于配置 CPSM 是否等待响应，如果等待，将等待哪种类型的响应。

- 00: 无响应，但 CMDSENT 标志除外
- 01: 短响应，但 CMDREND 或 CCRCFAIL 标志除外
- 10: 无响应，但 CMDSENT 标志除外
- 11: 长响应，但 CMDREND 或 CCRCFAIL 标志除外

位 5:0 **CMDINDEX**: 命令索引 (Command index)

命令索引作为命令消息的一部分发送给卡。

图 43.1.4.3 SDIO_CMD 寄存器位定义

图中只列出了部分位的描述，其中低 6 位为命令索引，也就是我们要发送的命令索引号（比如发送 CMD1，其值为 1，索引就设置为 1）。位[7:6]，用于设置等待响应位，用于指示 CPSM 是否需要等待，以及等待类型等。这里的 CPSM，即命令通道状态机，我们就不详细介绍了，请参阅《STM32F4xx 中文参考手册》第 776 页，有详细介绍。命令通道状态机我们一般都是开启的，所以位 10 要设置为 1。

第八个，我们要介绍的是 SDIO 数据定时器寄存器 (SDIO_DTIMER)，该寄存器用于存储以卡总线时钟 (SDIO_CK) 为周期的数据超时时间，一个计数器将从 SDIO_DTIMER 寄存器加载数值，并在数据通道状态机(DPSM)进入 Wait_R 或繁忙状态时进行递减计数，当 DPSM 处在这些状态时，如果计数器减为 0，则设置超时标志。这里的 DPSM，即数据通道状态机，类似 CPSM，详细请参考《STM32F4xx 中文参考手册》第 780 页。注意：在写入数据控制寄存器，进行数据传输之前，必须先写入该寄存器(SDIO_DTIMER)和数据长度寄存器(SDIO_DLEN)！

第九个，我们要介绍的是 SDIO 数据长度寄存器 (SDIO_DLEN)，该寄存器低 25 位有效，用于设置需要传输的数据字节长度。对于块数据传输，该寄存器的数值，必须是数据块长度（通过 SDIO_DCTRL 设置）的倍数。

第十个，我们要介绍的是 SDIO 数据控制寄存器 (SDIO_DCTRL)，该寄存器各位定义如图 43.1.4.4 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
																				SDIOEN	RWMOD	RWSTOP	RWSTART	DBLOCKSIZE				DMAEN	DTMODE	DTDIR	DTEN
																				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 11 **SDIOEN:** SD I/O 使能功能 (SD I/O enable functions)

如果将该位置 1，则 DPSM 执行特定于 SD I/O 卡的操作。

位 10 **RWMOD:** 读取等待模式 (Read wait mode)

- 0: 通过停止 SDIO_D2 进行读取等待控制
- 1: 使用 SDIO_CK 进行读取等待控制

位 9 **RWSTOP:** 读取等待停止 (Read wait stop)

- 0: 如果将 RWSTART 位置 1，则读取等待正在进行中
- 1: 如果将 RWSTART 位置 1，则使能读取等待停止

位 8 **RWSTART:** 读取等待开始 (Read wait start)

如果将该位置 1，则读取等待操作开始。

位 7:4 **DBLOCKSIZE:** 数据块大小 (Data block size)

定义在选择了块数据传输模式时数据块的长度:

0000: (十进制数 0) 块长度 = $2^0 = 1$ 字节	1000: (十进制数 8) 块长度 = $2^8 = 256$ 字节
0001: (十进制数 1) 块长度 = $2^1 = 2$ 字节	1001: (十进制数 9) 块长度 = $2^9 = 512$ 字节
0010: (十进制数 2) 块长度 = $2^2 = 4$ 字节	1010: (十进制数 10) 块长度 = $2^{10} = 1024$ 字节
0011: (十进制数 3) 块长度 = $2^3 = 8$ 字节	1011: (十进制数 11) 块长度 = $2^{11} = 2048$ 字节
0100: (十进制数 4) 块长度 = $2^4 = 16$ 字节	1100: (十进制数 12) 块长度 = $2^{12} = 4096$ 字节
0101: (十进制数 5) 块长度 = $2^5 = 32$ 字节	1101: (十进制数 13) 块长度 = $2^{13} = 8192$ 字节
0110: (十进制数 6) 块长度 = $2^6 = 64$ 字节	1110: (十进制数 14) 块长度 = $2^{14} = 16384$ 字节
0111: (十进制数 7) 块长度 = $2^7 = 128$ 字节	1111: (十进制数 15) 保留

位 3 **DMAEN:** DMA 使能位 (DMA enable bit)

- 0: 禁止 DMA。
- 1: 使能 DMA。

位 2 **DTMODE:** 数据传输模式选择 (Data transfer mode selection)

- 0: 块数据传输
- 1: 流或 SDIO 多字节数据传输

位 1 **DTDIR:** 数据传输方向选择 (Data transfer direction selection)

- 0: 从控制器到卡。
- 1: 从卡到控制器。

位 0 **DTEN:** 数据传输使能位 (Data transfer enabled bit)

如果 1 写入到 DTEN 位，则数据传输开始。根据方向位 DTDIR，如果在传输开始时立即将 RW 置 1 开始，则 DPSM 变为 Wait_S 状态、Wait_R 状态或读取等待状态。在数据传输结束后不需要将使能位清零，但必须更新 SDIO_DCTRL 以使能新的数据传输

图 43.1.4.4 SDIO_DCTRL 寄存器位定义

该寄存器，用于控制数据通道状态机 (DPSM)，包括数据传输使能、传输方向、传输模式、DMA 使能、数据块长度等信息，都是通过该寄存器设置。我们需要根据自己的实际情况，来配置该寄存器，才可正常实现数据收发。

接下来，我们介绍几个位定义十分类似的寄存器，他们是：状态寄存器 (SDIO_STA)、清除中断寄存器 (SDIO_ICR) 和中断屏蔽寄存器 (SDIO_MASK)，这三个寄存器每个位的定义都相同，只是功能各有不同。所以可以一起介绍，以状态寄存器 (SDIO_STA) 为例，该寄存器各位定义如图 43.1.4.5 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
Reserved																																
Res.		r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r		

位 23 **CEATAEND**: 针对 CMD61 收到了 CE-ATA 命令完成信号

位 22 **SDIOIT**: 收到了 SDIO 中断 (SDIO interrupt received)

位 21 **RXDAVL**: 接收 FIFO 中有数据可用 (Data available in receive FIFO)

位 20 **TXDAVL**: 传输 FIFO 中有数据可用 (Data available in transmit FIFO)

位 19 **RXFIFOE**: 接收 FIFO 为空 (Receive FIFO empty)

位 18 **TXFIFOE**: 发送 FIFO 为空 (Transmit FIFO empty)

如果使能了硬件流控制，则 TXFIFOE 信号在 FIFO 包含 2 个字时激活。

位 17 **RXFIFOF**: 接收 FIFO 已满 (Receive FIFO full)

如果使能了硬件流控制，则 RXFIFOF 信号在 FIFO 差 2 个字便变满之前激活。

位 16 **TXFIFOF**: 传输 FIFO 已满 (Transmit FIFO full)

位 15 **RXFIFOHF**: 接收 FIFO 半满: FIFO 中至少有 8 个字

位 14 **TXFIFOHE**: 传输 FIFO 半空: 至少可以写入 8 个字到 FIFO

位 13 **RXACT**: 数据接收正在进行中 (Data receive in progress)

位 12 **TXACT**: 数据传输正在进行中 (Data transmit in progress)

位 11 **CMDACT**: 命令传输正在进行中 (Command transfer in progress)

位 10 **DBCKEND**: 已发送/接收数据块 (CRC 校验通过)

位 9 **STBITERR**: 在宽总线模式下，并非在所有数据信号上都检测到了起始位

位 8 **DATAEND**: 数据结束 (数据计数器 SDIDCOUNT 为零)

位 7 **CMDSENT**: 命令已发送 (不需要响应) (Command sent (no response required))

位 6 **CMDREND**: 已接收命令响应 (CRC 校验通过)

位 5 **RXOVERR**: 收到了 FIFO 上溢错误 (Received FIFO overrun error)

位 4 **TXUNDERR**: 传输 FIFO 下溢错误 (Transmit FIFO underrun error)

位 3 **DTIMEOUT**: 数据超时 (Data timeout)

位 2 **CTIMEOUT**: 命令响应超时 (Command response timeout)

命令超时周期为固定值 64 个 SDIO_CK 时钟周期。

位 1 **DCRCFAIL**: 已发送/接收数据块 (CRC 校验失败)

位 0 **CCRCFAIL**: 已接收命令响应 (CRC 校验失败)

图 43.1.4.5 SDIO_STA 寄存器位定义

状态寄存器可以用来查询 SDIO 控制器的当前状态，以便处理各种事务。比如 SDIO_STA 的位 2 表示命令响应超时，说明 SDIO 的命令响应出了问题。我们通过设置 SDIO_ICR 的位 2 则可以清除这个超时标志，而设置 SDIO_MASK 的位 2，则可以开启命令响应超时中断，设置为 0 关闭。其他位我们就不一一介绍了，请大家自行学习。

最后，我们向大家介绍 SDIO 的数据 FIFO 寄存器 (SDIO_FIFO)，数据 FIFO 寄存器包括接收和发送 FIFO，他们由一组连续的 32 个地址上的 32 个寄存器组成，CPU 可以使用 FIFO 读写多个操作数。例如我们要从 SD 卡读数据，就必须读 SDIO_FIFO 寄存器，要写数据到 SD 卡，则要写 SDIO_FIFO 寄存器。SDIO 将这 32 个地址分为 16 个一组，发送接收各占一半。而我们每次读写的时候，最多就是读取发送 FIFO 或写入接收 FIFO 的一半大小的数据，也就是 8 个字 (32 个字节)，**这里特别提醒，我们操作 SDIO_FIFO (不论读出还是写入) 必须是以 4 字节对齐的内存进行操作，否则将导致出错！**

至此，SDIO 的相关寄存器介绍，我们就介绍完了。还有几个不常用的寄存器，我们没有

介绍到, 请大家参考《STM32F4xx 中文参考手册》第 28 章相关章节。

43.1.5 SD 卡初始化流程

最后, 我们来看看 SD 卡的初始化流程, 要实现 SDIO 驱动 SD 卡, 最重要的步骤就是 SD 卡的初始化, 只要 SD 卡初始化完成了, 那么剩下的 (读写操作) 就简单了, 所以我们这里重点介绍 SD 卡的初始化。从 SD 卡 2.0 协议 (见光盘资料) 文档, 我们得到 SD 卡初始化流程图如图 43.1.5.1 所示:

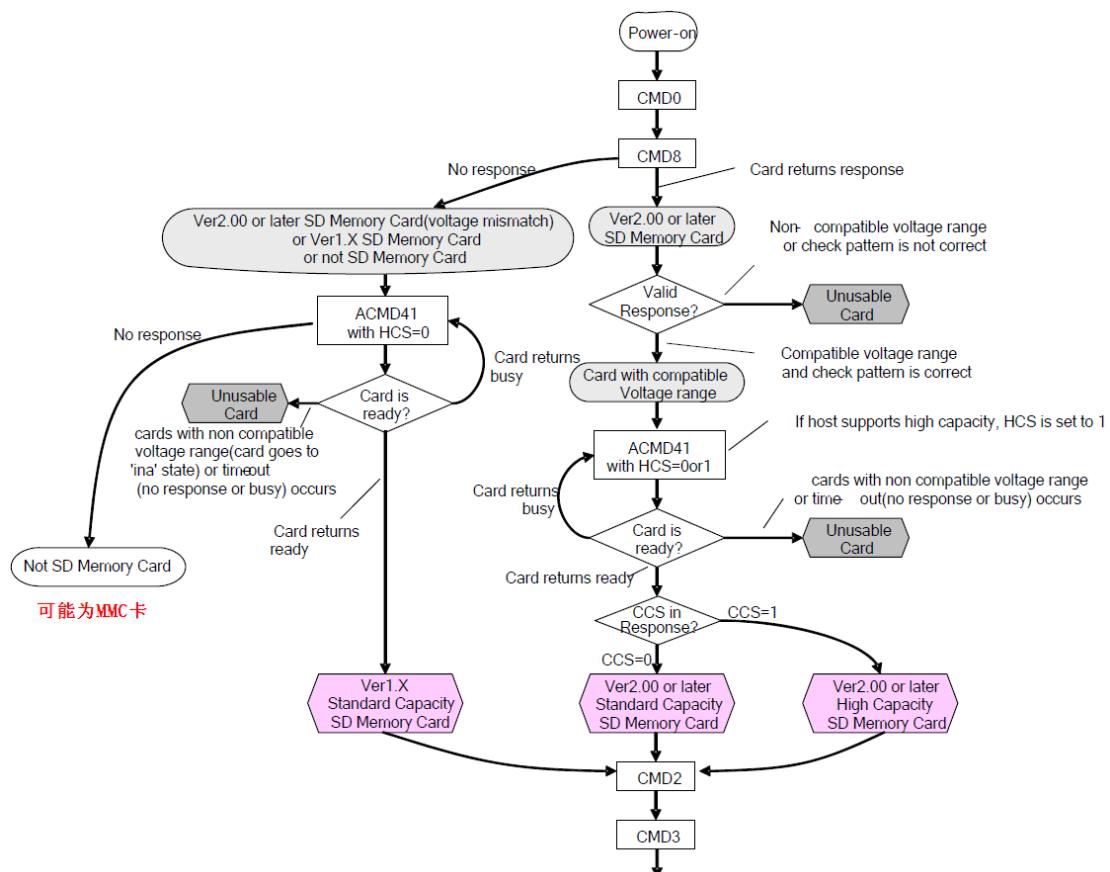


图 43.1.5.1 SD 卡初始化流程

从图中, 我们看到, 不管什么卡 (这里我们将卡分为 4 类: SD2.0 高容量卡 (SDHC, 最大 32G), SD2.0 标准容量卡 (SDSC, 最大 2G), SD1.x 卡和 MMC 卡), 首先我们要执行的是卡上电 (需要设置 SDIO_POWER[1:0]=11), 上电后发送 CMD0, 对卡进行软复位, 之后发送 CMD8 命令, 用于区分 SD 卡 2.0, 只有 2.0 及以后的卡才支持 CMD8 命令, MMC 卡和 V1.x 的卡, 是不支持该命令的。CMD8 的格式如表 43.1.5.1 所示:

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	1
Value	'0'	'1'	'001000'	'00000h'	x	x	x	'1'
Description	start bit	transmission bit	command index	reserved bits	voltage supplied (VHS)	check pattern	CRC7	end bit

表 43.1.5.1 CMD8 命令格式

这里, 我们需要在发送 CMD8 的时候, 通过其带的参数我们可以设置 VHS 位, 以告诉 SD

卡，主机的供电情况，VHS 位定义如表 43.1.5.2 所示：

Voltage Supplied	Value Definition
0000b	Not Defined
0001b	2.7-3.6V
0010b	Reserved for Low Voltage Range
0100b	Reserved
1000b	Reserved
Others	Not Defined

表 43.1.5.2 VHS 位定义

这里我们使用参数 0X1AA，即告诉 SD 卡，主机供电为 2.7~3.6V 之间，如果 SD 卡支持 CMD8，且支持该电压范围，则会通过 CMD8 的响应 (R7) 将参数部分原本返回给主机，如果不支持 CMD8，或者不支持这个电压范围，则不响应。

在发送 CMD8 后，发送 ACMD41（注意发送 ACMD41 之前要先发送 CMD55），来进一步确认卡的操作电压范围，并通过 HCS 位来告诉 SD 卡，主机是不是支持高容量卡 (SDHC)。ACMD41 的命令格式如表 43.1.5.3 所示：

ACMD INDEX	type	argument	resp	abbreviation	command description
ACMD41	bcr	[31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] V _{DD} Voltage Window(OCR[23:0])	R3	SD_SEND_OP_COND	Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30].

表 43.1.5.3 ACMD41 命令格式

ACMD41 得到的响应(R3)包含 SD 卡 OCR 寄存器内容，OCR 寄存器内容定义如表 43.1.5.4 所示：

OCR bit position	OCR Fields Definition
0-6	reserved
7	Reserved for Low Voltage Range
8-14	reserved
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-29	reserved
30	Card Capacity Status (CCS) ¹
31	Card power up status bit (busy) ²

VDD Voltage Window

1) This bit is valid only when the card power up status bit is set.

2) This bit is set to LOW if the card has not finished the power up routine.

表 43.1.5.4 OCR 寄存器定义

对于支持 CMD8 指令的卡，主机通过 ACMD41 的参数设置 HCS 位为 1，来告诉 SD 卡主

机支 SDHC 卡，如果设置为 0，则表示主机不支持 SDHC 卡，SDHC 卡如果接收到 HCS 为 0，则永远不会反回卡就绪状态。对于不支持 CMD8 的卡，HCS 位设置为 0 即可。

SD 卡在接收到 ACMD41 后，返回 OCR 寄存器内容，如果是 2.0 的卡，主机可以通过判断 OCR 的 CCS 位来判断是 SDHC 还是 SDSC；如果是 1.x 的卡，则忽略该位。OCR 寄存器的最后一个位用于告诉主机 SD 卡是否上电完成，如果上电完成，该位将会被置 1。

对于 MMC 卡，则不支持 ACMD41，不响应 CMD55，对 MMC 卡，我们只需要在发送 CMD0 后，在发送 CMD1（作用同 ACMD41），检查 MMC 卡的 OCR 寄存器，实现 MMC 卡的初始化。

至此，我们便实现了对 SD 卡的类型区分，图 43.1.5.1 中，最后发送了 CMD2 和 CMD3 命令，用于获得卡 CID 寄存器数据和卡相对地址（RCA）。

CMD2，用于获得 CID 寄存器的数据，CID 寄存器数据各位定义如表 43.1.5.5 所示：

Name	Field	Width	CID-slice
Manufacturer ID	MID	8	[127:120]
OEM/Application ID	OID	16	[119:104]
Product name	PNM	40	[103:64]
Product revision	PRV	8	[63:56]
Product serial number	PSN	32	[55:24]
reserved	--	4	[23:20]
Manufacturing date	MDT	12	[19:8]
CRC7 checksum	CRC	7	[7:1]
not used, always 1	-	1	[0:0]

表 43.1.5.5 卡 CID 寄存器位定义

SD 卡在收到 CMD2 后，将返回 R2 长响应（136 位），其中包含 128 位有效数据（CID 寄存器内容），存放在 SDIO_RESP1~4 等 4 个寄存器里面。通过读取这四个寄存器，就可以获得 SD 卡的 CID 信息。

CMD3，用于设置卡相对地址（RCA，必须为非 0），对于 SD 卡（非 MMC 卡），在收到 CMD3 后，将返回一个新的 RCA 给主机，方便主机寻址。RCA 的存在允许一个 SDIO 接口挂多个 SD 卡，通过 RCA 来区分主机要操作的是哪个卡。而对于 MMC 卡，则不是由 SD 卡自动返回 RCA，而是主机主动设置 MMC 卡的 RCA，即通过 CMD3 带参数（高 16 位用于 RCA 设置），实现 RCA 设置。同样 MMC 卡也支持一个 SDIO 接口挂多个 MMC 卡，不同于 SD 卡的是所有的 RCA 都是由主机主动设置的，而 SD 卡的 RCA 则是 SD 卡发给主机的。

在获得卡 RCA 之后，我们便可以发送 CMD9（带 RCA 参数），获得 SD 卡的 CSD 寄存器内容，从 CSD 寄存器，我们可以得到 SD 卡的容量和扇区大小等十分重要的信息。CSD 寄存器我们在这里就不详细介绍了，关于 CSD 寄存器的详细介绍，请大家参考《SD 卡 2.0 协议.pdf》。

至此，我们的 SD 卡初始化基本就结束了，最后通过 CMD7 命令，选中我们要操作的 SD 卡，即可开始对 SD 卡的读写操作了，SD 卡的其他命令和参数，我们这里就不再介绍了，请大家参考《SD 卡 2.0 协议.pdf》，里面有非常详细的介绍。

43.2 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

前面四部分，在之前的实例已经介绍过了，这里我们介绍一下探索者 STM32F4 开发板板载的 SD 卡接口和 STM32F4 的连接关系，如图 43.2.1 所示：

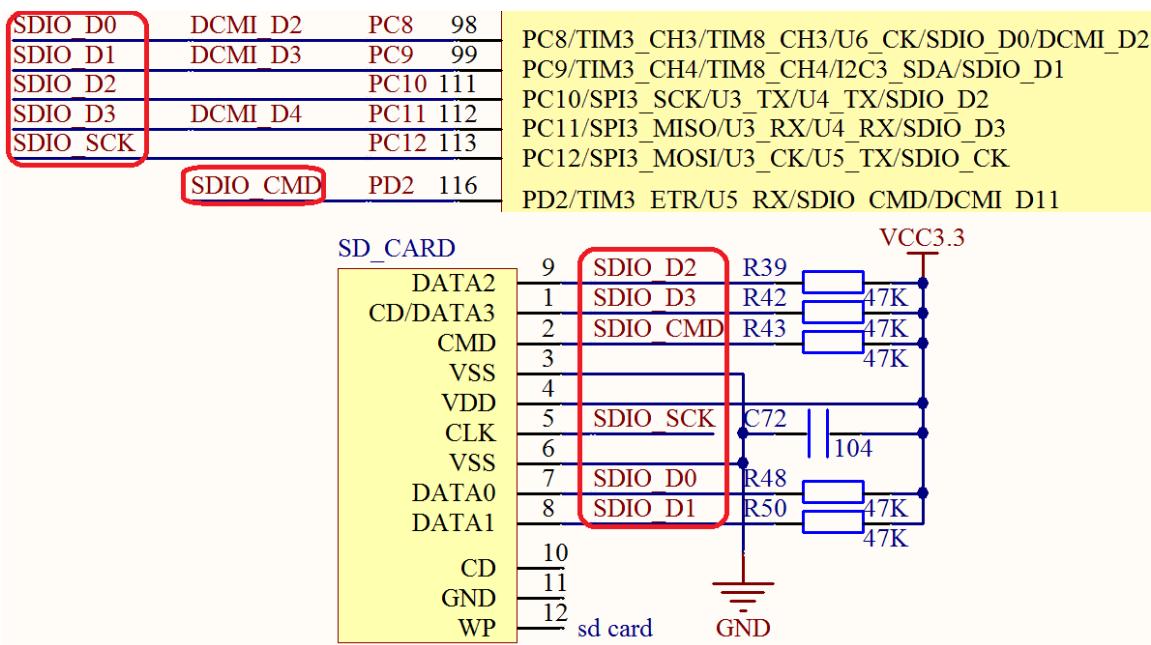


图 43.2.1 SD 卡接口与 STM32F4 连接原理图

探索者 STM32F4 开发板的 SD 卡座 (SD_CARD)，在 PCB 背面，SD 卡座与 STM32F4 的连接在开发板上是直接连接在一起的，硬件上不需要任何改动。

43.3 软件设计

打开本章实验工程可以看到，我们不但增加了固件库 SDIO 支持文件 `stm32f4xx_sdio.c` 以及头文件 `stm32f4xx_sdio.h`，同时，我们还新增了 SD 卡的 SDIO 支持文件 `sdio_sdcard.c` 以及头文件 `sdio_sdcard.h`。由于 `sdio_sdcard.c` 里面代码比较多，由于篇幅限制，我们不贴出所有代码，仅介绍几个重要的函数，第一个是 `SD_Init` 函数，该函数源码如下：

```
// 初始化 SD 卡
// 返回值: 错误代码;(0,无错误)
SD_Error SD_Init(void)
{
    SD_Error errorstatus=SD_OK;
    GPIO_InitTypeDef GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC|RCC_AHB1Periph_GPIOD|
                           RCC_AHB1Periph_DMA2, ENABLE); //使能 GPIOC,GPIOD DMA2 时钟
```

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SDIO, ENABLE); //SDIO 时钟使能
```

```
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SDIO, ENABLE); //SDIO 复位
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|  
    GPIO_Pin_11|GPIO_Pin_12 //PC8,9,10,11,12 复用功能输出  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //复用功能  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //100M  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉  
GPIO_Init(GPIOC, &GPIO_InitStructure); // PC8,9,10,11,12 复用功能输出
```

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;  
GPIO_Init(GPIOD, &GPIO_InitStructure); //PD2 复用功能输出
```

```
//引脚复用映射设置  
GPIO_PinAFConfig(GPIOC, GPIO_PinSource8, GPIO_AF_SDIO); //PC8,AF12  
GPIO_PinAFConfig(GPIOC, GPIO_PinSource9, GPIO_AF_SDIO);  
GPIO_PinAFConfig(GPIOC, GPIO_PinSource10, GPIO_AF_SDIO);  
GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_SDIO);  
GPIO_PinAFConfig(GPIOC, GPIO_PinSource12, GPIO_AF_SDIO);  
GPIO_PinAFConfig(GPIOD, GPIO_PinSource2, GPIO_AF_SDIO);
```

```
RCC_APB2PeriphResetCmd(RCC_APB2Periph_SDIO, DISABLE); //SDIO 结束复位  
SDIO_Register_Deinit(); //SDIO 外设寄存器设置为默认值
```

```
NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn;  
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //抢占优先级 3  
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; //响应优先级 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、
```

```
errorstatus=SD_PowerON(); //SD 卡上电
```

```
if(errorstatus==SD_OK)  
    errorstatus=SD_InitializeCards(); //初始化 SD 卡
```

```
if(errorstatus==SD_OK)  
    errorstatus=SD_GetCardInfo(&SDCardInfo); //获取卡信息  
if(errorstatus==SD_OK)  
    errorstatus=SD_SelectDeselect((u32)(SDCardInfo.RCA<<16)); //选中 SD 卡  
if(errorstatus==SD_OK)
```

```

errorstatus=SD_EnableWideBusOperation(SDIO_BusWide_4b);
           //4 位宽度,如果是 MMC 卡,则不能用 4 位模式
if((errorstatus==SD_OK)|(SDIO_MULTIMEDIA_CARD==CardType))
{
    //设置时钟频率,SDIO 时钟计算公式:SDIO_CK 时钟=SDIOCLK/[clkdiv+2];
    //其中,SDIOCLK 固定为 48Mhz
    SDIO_Clock_Set(SDIO_TRANSFER_CLK_DIV);
    //errorstatus=SD_SetDeviceMode(SD_DMA_MODE); //设置为 DMA 模式
    errorstatus=SD_SetDeviceMode(SD_POLLING_MODE); //设置为查询模式
}
return errorstatus;
}

```

该函数先实现 SDIO 时钟及相关 IO 口的初始化，然后开始 SD 卡的初始化流程，这个过程在 43.1.5 节有详细介绍。首先，通过 SD_PowerON 函数（该函数我们这里不做介绍，请参考本例程源码），我们将完成 SD 卡的上电，并获得 SD 卡的类型（SDHC/SDSC/SDV1.x/MMC），然后，调用 SD_InitializeCards 函数，完成 SD 卡的初始化，该函数代码如下：

```

//初始化所有的卡,并让卡进入就绪状态
//返回值:错误代码
SD_Error SD_InitializeCards(void)
{
    SD_Error errorstatus=SD_OK;
    u16 rca = 0x01;

    if (SDIO_GetPowerState() == SDIO_PowerState_OFF) //检查电源状态,确保为上电状态
    {
        errorstatus = SD_REQUEST_NOT_APPLICABLE;
        return(errorstatus);
    }

    if(SDIO_SECURE_DIGITAL_IO_CARD!=CardType)//非 SECURE_DIGITAL_IO_CARD
    {
        SDIO_CmdInitStructure.SDIO_Argument = 0x0; //发送 CMD2,取得 CID,长响应
        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ALL_SEND_CID;
        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
        SDIO_SendCommand(&SDIO_CmdInitStructure); //发送 CMD2,取得 CID,长响应

        errorstatus=CmdResp2Error();           //等待 R2 响应
    }

    if(errorstatus!=SD_OK) return errorstatus;      //响应错误
}

```

```
CID_Tab[0]=SDIO->RESP1;
CID_Tab[1]=SDIO->RESP2;
CID_Tab[2]=SDIO->RESP3;
CID_Tab[3]=SDIO->RESP4;
}
if((SDIO_STD_CAPACITY_SD_CARD_V1_1==CardType) ||
(SDIO_STD_CAPACITY_SD_CARD_V2_0==CardType) ||
(SDIO_SECURE_DIGITAL_IO_COMBO_CARD==CardType) ||
(SDIO_HIGH_CAPACITY_SD_CARD==CardType))//判断卡类型
{
    SDIO_CmdInitStructure.SDIO_Argument = 0x00;//发送 CMD3,短响应
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR; //cmd3
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r6
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure); //发送 CMD3,短响应

    errorstatus=CmdResp6Error(SD_CMD_SET_REL_ADDR,&rca);//等待 R6 响应

    if(errorstatus!=SD_OK) return errorstatus; //响应错误
}
if (SDIO_MULTIMEDIA_CARD==CardType)
{
    SDIO_CmdInitStructure.SDIO_Argument = (u32)(rca<<16);//发送 CMD3,短响应
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR; //cmd3
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r6
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure); //发送 CMD3,短响应
    errorstatus=CmdResp2Error(); //等待 R2 响应
    if(errorstatus!=SD_OK) return errorstatus; //响应错误
}
if (SDIO_SECURE_DIGITAL_IO_CARD!=CardType)//非 SECURE_DIGITAL_IO_CARD
{
    RCA = rca;

    SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16);
    //发送 CMD9+卡 RCA,取得 CSD,长响应
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
```

```

SDIO_SendCommand(&SDIO_CmdInitStructure);

errorstatus=CmdResp2Error();           //等待 R2 响应
if(errorstatus!=SD_OK) return errorstatus; //响应错误

CSD_Tab[0]=SDIO->RESP1;
CSD_Tab[1]=SDIO->RESP2;
CSD_Tab[2]=SDIO->RESP3;
CSD_Tab[3]=SDIO->RESP4;

}

return SD_OK;//卡初始化成功
}

```

SD_InitializeCards 函数主要发送 CMD2 和 CMD3，获得 CID 寄存器内容和 SD 卡的相对地址（RCA），并通过 CMD9，获取 CSD 寄存器内容。到这里，实际上 SD 卡的初始化就已经完成了。

随后，SD_Init 函数又通过调用 SD_GetCardInfo 函数，获取 SD 卡相关信息，之后调用 SD_SelectDeselect 函数，选择要操作的卡（CMD7+RCA），通过 SD_EnableWideBusOperation 函数设置 SDIO 的数据位宽为 4 位（但 MMC 卡只能支持 1 位模式！）。最后设置 SDIO_CK 时钟的频率，并设置工作模式（DMA/轮询）。

接下来，我们看看 SD 卡读块函数：SD_ReadBlock，该函数用于从 SD 卡指定地址读出一个块（扇区）数据，该函数代码如下：

```

//SD 卡读取一个块
//buf:读数据缓存区(必须 4 字节对齐!!)
//addr:读取地址
//blksize:块大小
SD_Error SD_ReadBlock(u8 *buf,long long addr,u16 blksize)
{
    SD_Error errorstatus=SD_OK;
    u8 power;
    u32 count=0,*tempbuff=(u32*)buf;//转换为 u32 指针
    u32 timeout=SDIO_DATATIMEOUT;
    if(NULL==buf) return SD_INVALID_PARAMETER;
    SDIO->DCTRL=0x0; //数据控制寄存器清零(关 DMA)
    if(CardType==SDIO_HIGH_CAPACITY_SD_CARD)//大容量卡
    {
        blksize=512;
        addr>>=9;
    }
    SDIO_DataInitStructure.SDIO_DataBlockSize= 0;//清除 DPSM 状态机配置
    SDIO_DataInitStructure.SDIO_DataLength= 0 ;
    SDIO_DataInitStructure.SDIO_DataTimeOut=SD_DATATIMEOUT ;
    SDIO_DataInitStructure.SDIO_DPSM=SDIO_DPSM_Enable;
    SDIO_DataInitStructure.SDIO_TransferDir=SDIO_TransferDir_ToCard;

```

```
SDIO_DataInitStructure.SDIO_TransferMode=SDIO_TransferMode_Block;
SDIO_DataConfig(&SDIO_DataInitStructure);

if(SDIO->RESP1&SD_CARD_LOCKED)return SD_LOCK_UNLOCK_FAILED;//卡锁了
if((blksize>0)&&(blksize<=2048)&&((blksize&(blksize-1))==0))
{
    power=convert_from_bytes_to_power_of_two(blksize);

    SDIO_CmdInitStructure.SDIO_Argument = blksize;
        //发送 CMD16+设置数据长度为 blksize,短响应
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    errorstatus=CmdResp1Error(SD_CMD_SET_BLOCKLEN);    //等待 R1 响应
    if(errorstatus!=SD_OK)return errorstatus;          //响应错误
}else return SD_INVALID_PARAMETER;

SDIO_DataInitStructure.SDIO_DataBlockSize= power<<4; //清除 DPSM 状态机配置
SDIO_DataInitStructure.SDIO_DataLength= blksize ;
SDIO_DataInitStructure.SDIO_DataTimeOut=SD_DATATIMEOUT ;
SDIO_DataInitStructure.SDIO_DPSM=SDIO_DPSM_Enable;
SDIO_DataInitStructure.SDIO_TransferDir=SDIO_TransferDir_ToSDIO;
SDIO_DataInitStructure.SDIO_TransferMode=SDIO_TransferMode_Block;
SDIO_DataConfig(&SDIO_DataInitStructure);

SDIO_CmdInitStructure.SDIO_Argument = addr;
        //发送 CMD17+从 addr 地址出读取数据,短响应
SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_READ_SINGLE_BLOCK;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);

errorstatus=CmdResp1Error(SD_CMD_READ_SINGLE_BLOCK);//等待 R1 响应
if(errorstatus!=SD_OK)return errorstatus;          //响应错误
if(DeviceMode==SD_POLLING_MODE)                 //查询模式,轮询数据
{
    INTX_DISABLE();//关闭总中断(POLLING 模式,严禁中断打断 SDIO 读写操作!!!)
    while(!(SDIO->STA&((1<<5)|(1<<1)|(1<<3)|(1<<10)|(1<<9))))
        //无上溢/CRC/超时/完成(标志)/起始位错误
}
```

```
{  
    if(SDIO_GetFlagStatus(SDIO_FLAG_RXFIFOHF) != RESET)  
        //接收区半满,表示至少存了 8 个字  
    {  
        for(count=0;count<8;count++)  
            //循环读取数据  
            { *(tempbuff+count)=SDIO->FIFO;  
            }  
        tempbuff+=8;  
        timeout=0X7FFF;  
    }else //处理超时  
    {  
        if(timeout==0)return SD_DATA_TIMEOUT;  
        timeout--;  
    }  
}  
if(SDIO_GetFlagStatus(SDIO_FLAG_DTIMEOUT) != RESET)//数据超时错误  
{  
    SDIO_ClearFlag(SDIO_FLAG_DTIMEOUT); //清错误标志  
    return SD_DATA_TIMEOUT;  
}  
else if (SDIO_GetFlagStatus(SDIO_FLAG_DCRCFAIL) != RESET)//数据块 CRC 错误  
{  
    SDIO_ClearFlag(SDIO_FLAG_DCRCFAIL); //清错误标志  
    return SD_CRC_FAIL;  
}  
else if(SDIO_GetFlagStatus(SDIO_FLAG_RXOVERR) != RESET) //接收 fifo 上溢错误  
{  
    SDIO_ClearFlag(SDIO_FLAG_RXOVERR); //清错误标志  
    return SD_RX_OVERRUN;  
}  
else if(SDIO_GetFlagStatus(SDIO_FLAG_STBITERR) != RESET) //接收起始位错误  
{  
    SDIO_ClearFlag(SDIO_FLAG_STBITERR); //清错误标志  
    return SD_START_BIT_ERR;  
}  
while(SDIO_GetFlagStatus(SDIO_FLAG_RXDAVL) != RESET)//FIFO 还存在可用数据  
{  
    *tempbuff=SDIO->FIFO; //循环读取数据  
    tempbuff++;  
}  
INTX_ENABLE(); //开启总中断  
SDIO_ClearFlag(SDIO_STATIC_FLAGS); //清除所有标记  
}  
else if(DeviceMode==SD_DMA_MODE)  
{  
    TransferError=SD_OK;  
    StopCondition=0; //单块读,不需要发送停止传输指令  
    TransferEnd=0; //传输结束标志置位,在中断服务置 1  
    SDIO->MASK|=(1<<1)|(1<<3)|(1<<8)|(1<<5)|(1<<9); //配置需要的中断  
    SDIO->DCTRL|=1<<3; //SDIO DMA 使能  
    SD_DMA_Config((u32*)buf,blksize,DMA_DIR_PeripheralToMemory);  
    while((DMA_GetFlagStatus(DMA2_Stream3,DMA_FLAG_TCIF3)==RESET)&&(
```

```

        TransferEnd==0)&&(TransferError==SD_OK)&&timeout)timeout--;//等待传输完成
        if(timeout==0)return SD_DATA_TIMEOUT;//超时
        if(TransferError!=SD_OK)errorstatus=TransferError;
    }
    return errorstatus;
}

```

该函数先发送 CMD16，用于设置块大小，然后配置 SDIO 控制器读数据的长度，这里我们用到函数 convert_from_bytes_to_power_of_two 求出 blksize 以 2 为底的指数，用于 SDIO 读数据长度设置。然后发送 CMD17（带地址参数 addr），从指定地址读取一块数据。最后，根据我们所设置的模式（查询模式/DMA 模式），从 SDIO_FIFO 读出数据。

该函数有两个注意的地方：1，addr 参数类型为 long long，以支持大于 4G 的卡，否则操作大于 4G 的卡，可能有问题!!! 2，轮询方式，读写 FIFO 时，严禁任何中断打断，否则可能导致读写数据出错!! 所以使用了 INTX_DISABLE 函数关闭总中断，在 FIFO 读写操作结束后，才打开总中断（INTX_ENABLE 函数设置）。

SD_ReadBlock 函数，就介绍到这里，另外，还有三个底层读写函数：SD_ReadMultiBlocks，用于多块读；SD_WriteBlock，用于单块写；SD_WriteMultiBlocks，用于多块写；**再次提醒：无论哪个函数，其数据 buf 的地址都必须是 4 字节对齐的！**限于篇幅，余下 3 个函数我们就不一一介绍了，大家可以参看实验考源代码。关于控制命令，如果有不了解的，请参考《SD 卡 2.0 协议.pdf》里面有非常详细的介绍。

最后，我们来看看 SDIO 与文件系统的两个接口函数：SD_ReadDisk 和 SD_WriteDisk，这两个函数的代码如下：

```

//读 SD 卡
//buf:读数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 SD_ReadDisk(u8*buf,u32 sector,u8 cnt)
{
    u8 sta=SD_OK;
    long long lsector=sector;
    u8 n;
    if(CardType!=SDIO_STD_CAPACITY_SD_CARD_V1_1)lsector<<=9;
    if((u32)buf%4!=0)
    {
        for(n=0;n<cnt;n++)
        {
            sta=SD_ReadBlock(SDIO_DATA_BUFFER,lsector+512*n,512);//单扇区读操作
            memcpy(buf,SDIO_DATA_BUFFER,512);
            buf+=512;
        }
    }else
    {
        if(cnt==1)sta=SD_ReadBlock(buf,lsector,512);      //单个 sector 的读操作
    }
}

```

```

        else sta=SD_ReadMultiBlocks(buf,lsector,512,cnt);//多个 sector
    }
    return sta;
}
//写 SD 卡
//buf:写数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 SD_WriteDisk(u8*buf,u32 sector,u8 cnt)
{
    u8 sta=SD_OK;
    u8 n;
    long long lsector=sector;
    if(CardType!=SDIO_STD_CAPACITY_SD_CARD_V1_1)lsector<<=9;
    if((u32)buf%4!=0)
    {
        for(n=0;n<cnt;n++)
        {
            memcpy(SDIO_DATA_BUFFER,buf,512);
            sta=SD_WriteBlock(SDIO_DATA_BUFFER,lsector+512*n,512);//单扇区写
            buf+=512;
        }
    }else
    {
        if(cnt==1)sta=SD_WriteBlock(buf,lsector,512);      //单个 sector 的写操作
        else sta=SD_WriteMultiBlocks(buf,lsector,512,cnt); //多个 sector
    }
    return sta;
}

```

这两个函数在下一章(FATFS 实验)将会用到的,这里提前给大家介绍下,其中 SD_ReadDisk 用于读数据,通过调用 SD_ReadBlock 和 SD_ReadMultiBlocks 实现。SD_WriteDisk 用于写数据,通过调用 SD_WriteBlock 和 SD_WriteMultiBlocks 实现。注意,因为 FATFS 提供给 SD_ReadDisk 或者 SD_WriteDisk 的数据缓存区地址不一定是 4 字节对齐的,所以我们在这两个函数里面做了 4 字节对齐判断,如果不是 4 字节对齐的,则通过一个 4 字节对齐缓存(SDIO_DATA_BUFFER)作为数据过度,以确保传递给底层读写函数的 buf 是 4 字节对齐的。

sdio_sdcard.c 的内容,我们就介绍到这里, sdio_sdcard.h 我们就不做介绍了,请大家参考本例程源码。接下来,打开 main.c 文件,代码如下:

```

//通过串口打印 SD 卡相关信息
void show_sdcard_info(void)
{
    switch(SDCardInfo.CardType)
    {

```

```
case SDIO_STD_CAPACITY_SD_CARD_V1_1:  
    printf("Card Type:SDSC V1.1\r\n");break;  
case SDIO_STD_CAPACITY_SD_CARD_V2_0:  
    printf("Card Type:SDSC V2.0\r\n");break;  
case SDIO_HIGH_CAPACITY_SD_CARD:  
    printf("Card Type:SDHC V2.0\r\n");break;  
case SDIO_MULTIMEDIA_CARD:  
    printf("Card Type:MMC Card\r\n");break;  
}  
printf("Card ManufacturerID:%d\r\n",SDCardInfo.SD_cid.ManufacturerID);//制造商 ID  
printf("Card RCA:%d\r\n",SDCardInfo.RCA);//卡相对地址  
printf("Card Capacity:%d MB\r\n", (u32)(SDCardInfo.CardCapacity>>20));//显示容量  
printf("Card BlockSize:%d\r\n\r\n",SDCardInfo.CardBlockSize); //显示块大小  
}  
int main(void)  
{  
    u8 key; u8 t=0; u8 *buf;  
    u32 sd_size;  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2  
    delay_init(168); //初始化延时函数  
    uart_init(115200); //初始化串口波特率为 115200  
    LED_Init(); //初始化 LED  
    LCD_Init(); //LCD 初始化  
    KEY_Init(); //按键初始化  
    my_mem_init(SRAMIN); //初始化内部内存池  
    my_mem_init(SRAMCCM); //初始化 CCM 内存池  
    POINT_COLOR=RED;//设置字体为红色  
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");  
    LCD_ShowString(30,70,200,16,16,"SD CARD TEST");  
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(30,110,200,16,16,"2014/5/15");  
    LCD_ShowString(30,130,200,16,16,"KEY0:Read Sector 0");  
    while(SD_Init())//检测不到 SD 卡  
    {  
        LCD_ShowString(30,150,200,16,16,"SD Card Error!"); delay_ms(500);  
        LCD_ShowString(30,150,200,16,16,"Please Check! "); delay_ms(500);  
        LED0=!LED0;//DS0 闪烁  
    }  
    show_sdcard_info(); //打印 SD 卡相关信息  
    POINT_COLOR=BLUE; //设置字体为蓝色  
    //检测 SD 卡成功  
    LCD_ShowString(30,150,200,16,16,"SD Card OK      ");  
    LCD_ShowString(30,170,200,16,16,"SD Card Size:      MB");
```

```
LCD_ShowNum(30+13*8,170,SDCardInfo.CardCapacity>>20,5,16); //显示 SD 卡容量
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY0_PRES)//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(30,190,200,16,16,"USART1 Sending Data... ");
            printf("SECTOR 0 DATA:\r\n");
            for(sd_size=0;sd_size<512;sd_size++)printf("%x ",buf[sd_size]); //扇区数据
            printf("\r\nDATA ENDED\r\n");
            LCD_ShowString(30,190,200,16,16,"USART1 Send Data Over!");
        }
        myfree(0,buf); //释放内存
    }
    t++;
    delay_ms(10);
    if(t==20) { LED0=!LED0; t=0; }
}
}
```

这里总共 2 个函数, show_sdcard_info 函数用于从串口输出 SD 卡相关信息。而 main 函数, 则先初始化 SD 卡, 初始化成功, 则调用 show_sdcard_info 函数, 输出 SD 卡相关信息, 并在 LCD 上面显示 SD 卡容量。然后进入死循环, 如果有按键 KEY0 按下, 则通过 SD_ReadDisk 读取 SD 卡的扇区 0 (物理磁盘, 扇区 0), 并将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。

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 探索者 STM32F4 开发板上, 可以看到 LCD 显示如图 43.4.1 所示的内容 (假设 SD 卡已经插上了):

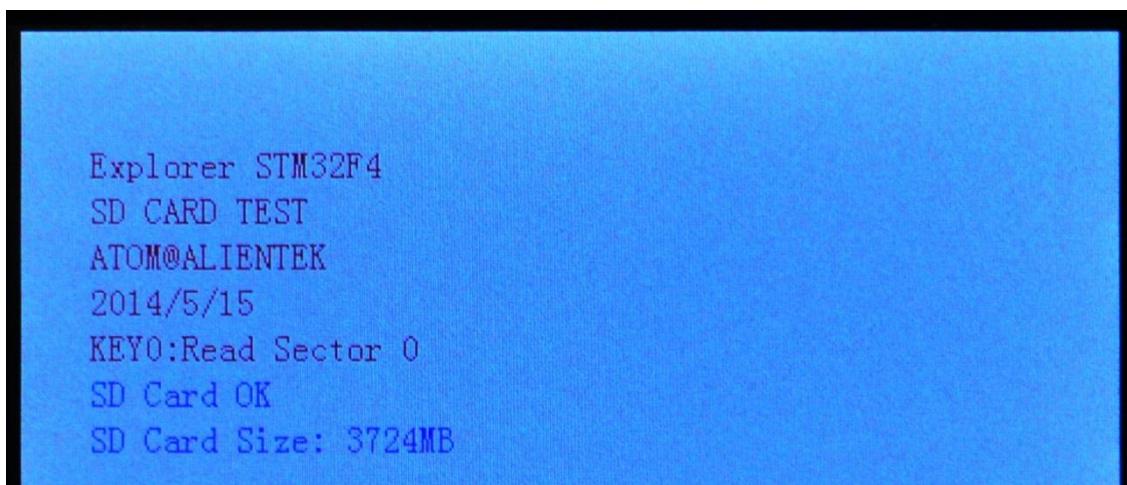


图 43.4.1 程序运行效果图

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 43.4.2 所示：

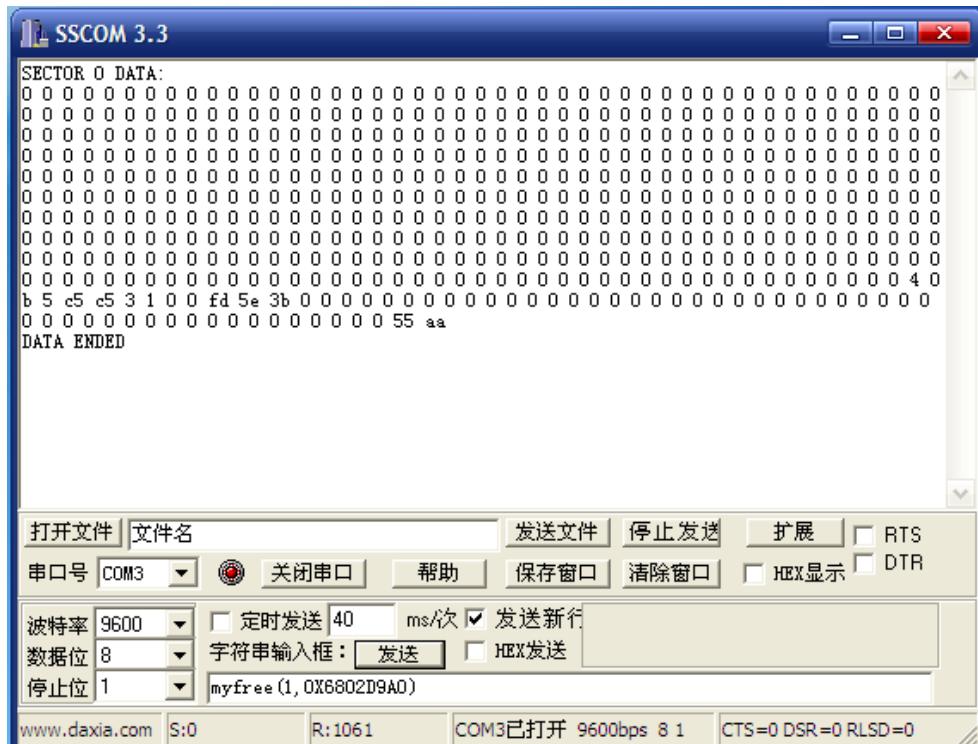


图 43.4.2 串口收到的 SD 卡扇区 0 内容

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 43.4.2 不同而感到惊讶。

第四十四章 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：

- 44.1 FATFS 简介
- 44.2 硬件设计
- 44.3 软件设计
- 44.4 下载验证

44.1 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 44.1.1 所示：



图 44.1.1 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 R0.10b。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) _USE_LABEL。这个用来设置是否支持磁盘盘符（磁盘名字）读取与设置。我们设置为 1，使能，就可以通过相关函数读取或者设置磁盘的名字了。

7) _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）。

8) _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。

9) _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。

10) _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 integer.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 44.1.2 所示：

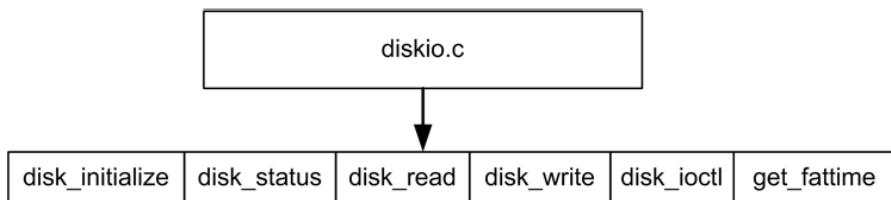


图 44.1.2 diskio 需要实现的函数

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 MDK5.11a 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 10 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 44.1.3 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 44.1.3 disk_initialize 函数介绍

第二个函数是 disk_status 函数，该函数介绍如图 44.1.4 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 44.1.4 disk_status 函数介绍

第三个函数是 disk_read 函数, 该函数介绍如图 44.1.5 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	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
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 44.1.5 disk_read 函数介绍

第四个函数是 disk_write 函数, 该函数介绍如图 44.1.6 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 SectorCount: 指定要写入的扇区数, 取值 1~128</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_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</p>
所在文件	ff.c
注意事项	只读配置中不需要此函数

图 44.1.6 disk_write 函数介绍

第五个函数是 disk_ioctl 函数, 该函数介绍如图 44.1.7 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 44.1.7 disk_ioctl 函数介绍

最后一个函数是 get_fattime 函数, 该函数介绍如图 44.1.8 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	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)
所在文件	ff.c
注意事项	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

图 44.1.8 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。

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，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
- 5) 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。

44.3 软件设计

打开本章实验目录可以看到，我们在工程目录下新建了一个 FATFS 的文件夹，然后将 FATFS R0.10b 程序包解压到该文件夹下。同时，我们在 FATFS 文件夹里面新建了一个 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 44.3.1 所示：



图 44.3.1 FATFS 文件夹子目录

然后打开我们实验工程可以看到，我们新建了 FATFS 分组，将必要的源文件添加到了 FATFS 分组之下。打开 diskio.c，代码如下：

```
#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 W25Q128
//前 12M 字节给 fatfs 用,12M 字节后,用于存放字库,字库占用 3.09M. 剩余部分,
//给客户自己用
u16 FLASH_SECTOR_COUNT=2048*12; //W25Q1218,前 12M 字节给 FATFS 占用
#define FLASH_BLOCK_SIZE 8 //每个 BLOCK 有 8 个扇区
//初始化磁盘
DSTATUS disk_initialize (
    BYTE pdrv /* Physical drive nmuber (0..) */
)
{
    u8 res=0;
    switch(pdrv)
    {
        case SD_CARD://SD 卡
            res=SD_Init(); //SD 卡初始化
            break;
        case EX_FLASH://外部 flash
            W25QXX_Init();
            FLASH_SECTOR_COUNT=2048*12; //W25Q1218,前 12M 字节给 FATFS 占用
            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 pdrv      /* 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 pdrv,          /* Physical drive nmuber (0..) */
    BYTE *buff,         /* Data buffer to store read data */
    DWORD sector,       /* Sector address (LBA) */
    UINT count          /* Number of sectors to read (1..128) */
)
{
    u8 res=0;
    if (!count) return RES_PARERR; //count 不能等于 0, 否则返回参数错误
    switch(pdrv)
    {
        case SD_CARD://SD 卡
            res=SD_ReadDisk(buff,sector,count);
            break;
        case EX_FLASH://外部 flash
            for(;count>0;count--)
            {
                W25QXX_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 _USE_WRITE
DRESULT disk_write (
    BYTE pdrv,          /* Physical drive nmuber (0..) */
    const BYTE *buff,   /* Data to be written */
    DWORD sector,       /* Sector address (LBA) */
    UINT count          /* Number of sectors to write (1..128) */
)
{
    u8 res=0;
    if (!count) return RES_PARERR; //count 不能等于 0, 否则返回参数错误
    switch(pdrv)
    {
        case SD_CARD://SD 卡
            res=SD_WriteDisk((u8*)buff,sector,count);
            break;
        case EX_FLASH://外部 flash
            for(;count>0;count--)
            {
                W25QXX_Write((u8*)buff,sector*FLASH_SECTOR_SIZE,FLASH_SECT
                    OR_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
//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
#if _USE_IOCTL
```

```
DRESULT disk_ioctl (
    BYTE pdrv,          /* Physical drive nmuber (0..) */
    BYTE cmd,           /* Control code */
    void *buff          /* Buffer to send/receive control data */
)
{
    DRESULT res;
    if(pdrv==SD_CARD)//SD 卡
    {
        switch(cmd)
        {
            case CTRL_SYNC:
                res = RES_OK;
                break;
            case GET_SECTOR_SIZE:
                *(DWORD*)buff = 512;
                res = RES_OK;
                break;
            case GET_BLOCK_SIZE:
                *(WORD*)buff = SDCardInfo.CardBlockSize;
                res = RES_OK;
                break;
            case GET_SECTOR_COUNT:
                *(DWORD*)buff = SDCardInfo.CardCapacity/512;
                res = RES_OK;
                break;
            default:
                res = RES_PARERR;
                break;
        }
    }else if(pdrv==EX_FLASH) //外部 FLASH
    {
        switch(cmd)
        {
            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;
                break;
        }
    }
}
```

```
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;
}

#endif

//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_memfree (void* mf)
{
    myfree(SRAMIN,mf);
}
```

该函数实现了我们 44.1 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_memfree 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。

打开 ffconf.h 可以看到，我们根据前面讲解修改了相关配置，此部分就不贴代码了，请大家参考本例程源码。另外，cc936.c 主要提供 UNICODE 到 GBK 以及 GBK 到 UNICODE 的码表转换，里面就是两个大数组，并提供一个 ff_convert 的转换函数，供 UNICODE 和 GBK 码互换，这个在中文长文件名支持的时候，必须用到！

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，请大家参考本例程源码，

最后，我们打开 main. c， main 函数如下：

```
int main(void)
{
    u32 total,free;
    u8 t=0; u8 res=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    usmart_dev.init(84); //初始化 USMART
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    W25QXX_Init(); //初始化 W25Q128
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"FATFS TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/5/15");
    LCD_ShowString(30,130,200,16,16,"Use USMART for test");
    while(SD_Init())//检测不到 SD 卡
    {
        LCD_ShowString(30,150,200,16,16,"SD Card Error!"); delay_ms(500);
        LCD_ShowString(30,150,200,16,16,"Please Check! "); delay_ms(500);
        LED0=!LED0;//DS0 闪烁
    }
    exfun_init(); //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1); //挂载 SD 卡
    res=f_mount(fs[1],"1:",1); //挂载 FLASH.
    if(res==0X0D)//FLASH 磁盘,FAT 文件系统错误,重新格式化 FLASH
    {
        LCD_ShowString(30,150,200,16,16,"Flash Disk Formatting...");//格式化 FLASH
        res=f_mkfs("1:",1,4096);//格式化 FLASH,1,盘符;1,不需要引导区,8 个扇区为 1 个簇
        if(res==0)
        {
            f_setlabel((const TCHAR *)"1:ALIENTEK");//设置磁盘的名字为： ALIENTEK
        }
    }
}
```

```

LCD_ShowString(30,150,200,16,16,"Flash Disk Format Finish");//格式化完成
}else LCD_ShowString(30,150,200,16,16,"Flash Disk Format Error ");//格式化失败
delay_ms(1000);
}
LCD_Fill(30,150,240,150+16,WHITE); //清除显示
while(exf_getfree("0",&total,&free)) //得到 SD 卡的总容量和剩余容量
{
    LCD_ShowString(30,150,200,16,16,"SD Card Fatfs Error!"); delay_ms(200);
    LCD_Fill(30,150,240,150+16,WHITE); delay_ms(200); //清除显示
    LED0=!LED0;//DS0 闪烁
}
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(30,150,200,16,16,"FATFS OK!");
LCD_ShowString(30,170,200,16,16,"SD Total Size:      MB");
LCD_ShowString(30,190,200,16,16,"SD  Free Size:     MB");
LCD_ShowNum(30+8*14,170,total>>10,5,16); //显示 SD 卡总容量 MB
LCD_ShowNum(30+8*14,190,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* path,u8 mt)",
(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_closedir,"u8 mf_closedir(void)",
(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* path,u8 mode,u16 au)",
(void*)mf_unlink,"u8 mf_unlink(u8 *pname)",

```

```
(void*)mf_rename,"u8 mf_rename(u8 *oldname,u8* newname)",
(void*)mf_getlabel,"void mf_getlabel(u8 *path)",
(void*)mf_setlabel,"void mf_setlabel(u8 *path)",
(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。

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 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 44.4.1 所示的内容（假定 SD 卡已经插上了）：

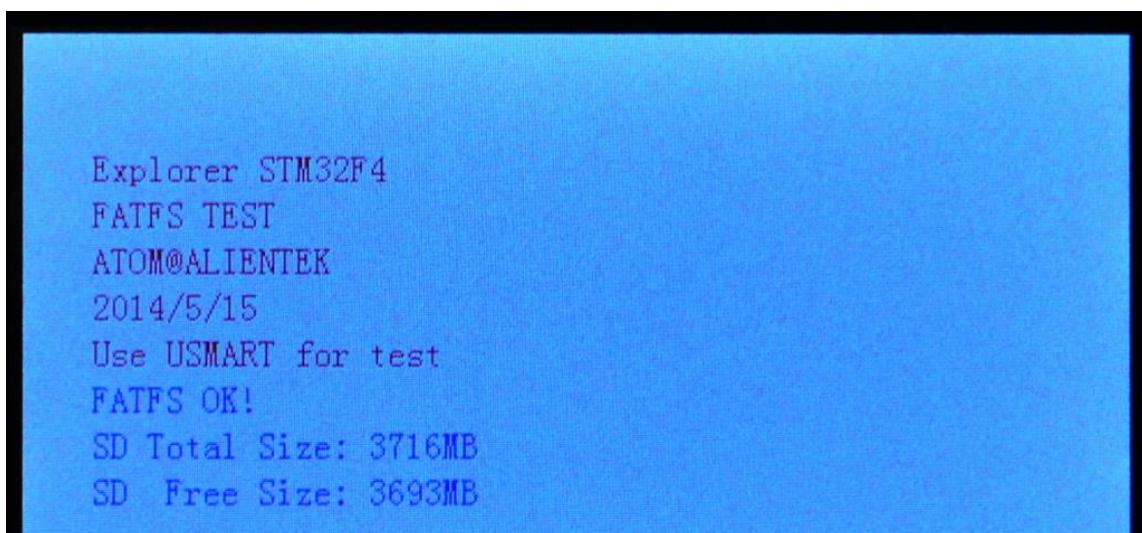


图 44.4.1 程序运行效果图

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 44.4.2 所示：

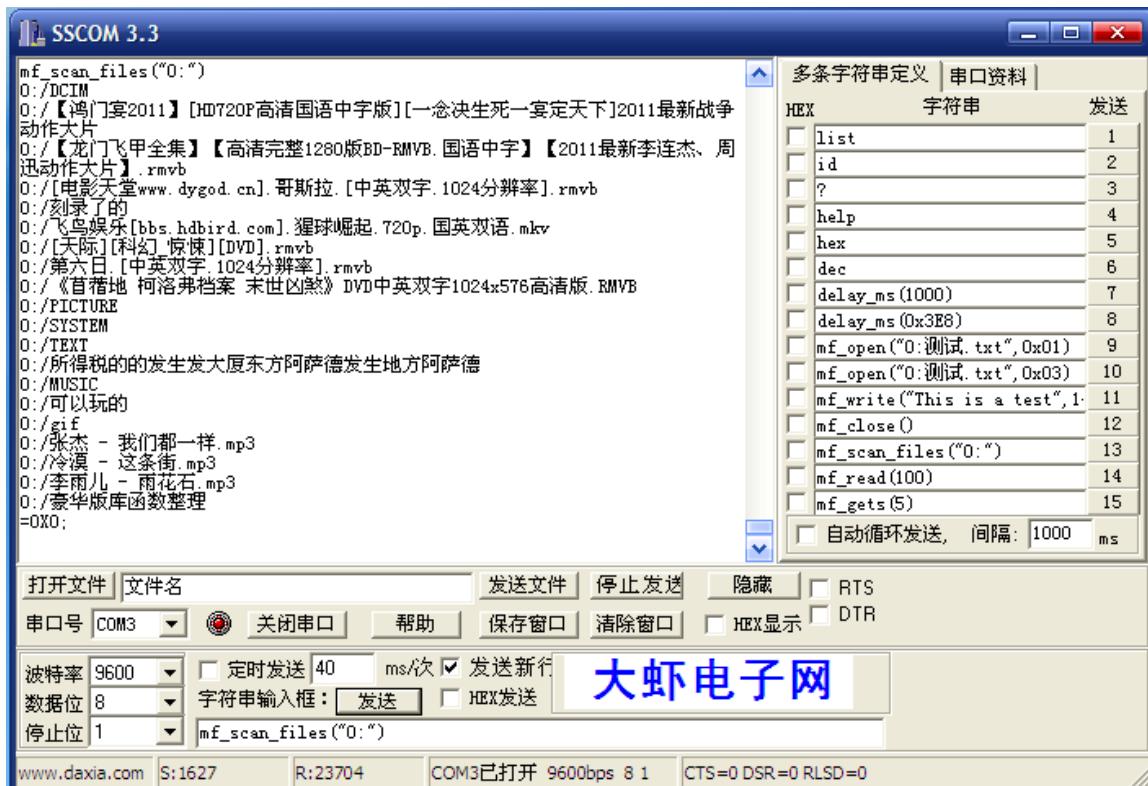


图 44.4.2 扫描 SD 卡根目录所有文件

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。

第四十五章 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 STM32F4 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。STM32F4 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：

- 45.1 汉字显示原理简介
- 45.2 硬件设计
- 45.3 软件设计
- 45.4 下载验证

45.1 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。

本实例我们将制作三个 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 W25Q128 里，这样，W25Q128 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 $\text{GBK}_L < 0X7F$ 时： $H_p = ((\text{GBK}_H - 0x81) * 190 + \text{GBK}_L - 0X40) * (\text{size} * 2);$

当 $\text{GBK}_L > 0X80$ 时： $H_p = ((\text{GBK}_H - 0x81) * 190 + \text{GBK}_L - 0X41) * (\text{size} * 2);$

其中 GBK_H 、 GBK_L 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）， H_p 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 45.1.1 所示：



图 45.1.1 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;
            W25QXX_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 45.1.2 所示：



图 45.1.2 点阵字库生成器默认界面

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，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(.DZK 文件)。具体设置如图 45.1.3 所示：

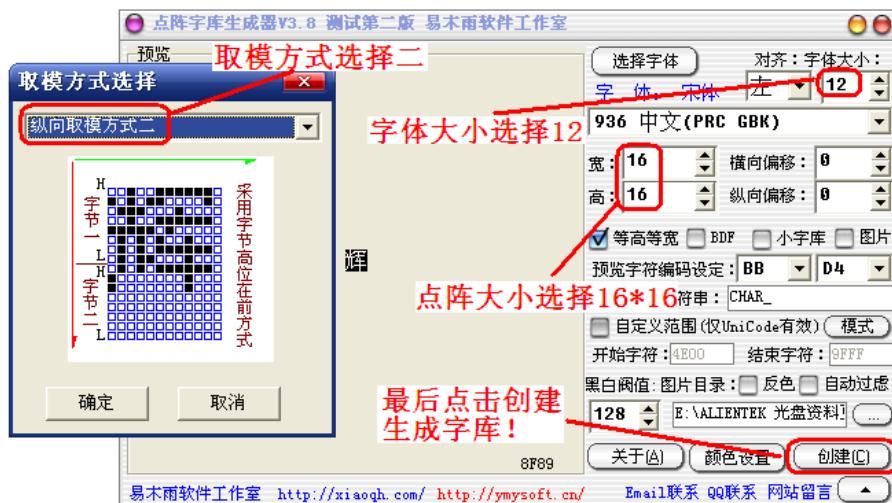


图 45.1.3 生成 GBK16*16 字库的设置方法

注意：电脑端的字体大小与我们生成点阵大小的关系为：

$$\text{fsize} = \text{dsize} * 6 / 8$$

其中，fsize 是电脑端字体大小，dsize 是点阵大小（12、16、24 等）。所以 16*16 点阵大小对应的是 12 字体。

生成完以后，我们把文件名和后缀改成：GBK16.FON。同样的方法，生成 12*12 的点阵库（GBK12.FON）和 24*24 的点阵库（GBK24.FON），总共制作 3 个字库。

另外，该软件还可以生成其他很多字库，字体也可选，大家可以根据自己的需要按照上面的方法生成即可。该软件的详细介绍请看软件自带的《点阵字库生成器说明书》，关于汉字显示原理，我们就介绍到这。

45.2 硬件设计

本章实验功能简介：开机的时候先检测 W25Q128 中是否已经存在字库，如果存在，则按次序显示汉字（三种字体都显示）。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、GBK16.FON 和 GBK24.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。通过按按键 KEY0，可以强制更新字库。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) KEY0 按键
- 3) 串口
- 4) TFTLCD 模块
- 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。

45.3 软件设计

打开本章实验目录可以看到，首先在工程根目录文件夹下面新建了一个 TEXT 的文件夹。在 TEXT 文件夹下新建 fontupd.c、fontupd.h、text.c、text.h 这 4 个文件。同时，我们在实验工

程中新建了 TEXT 分组，将新建的源文件加入到了分组之下，并将头文件包含路径加入到了工程的 PATH 中。

打开 fontupd.c，代码如下：

```
//字库区域占用的总扇区数大小(字库信息+unibk 表+3 个字库=3238700 字节,约占 791
//个 W25QXX 扇区)
#define FONTSECSIZE      791
//字库存放起始地址
#define FONTINFOADDR    1024*1024*12
//探索者 STM32F4 开发板，是从 12M 地址以后开始存放字库,前面 12M 被 fatfs 占用了.
//12M 以后紧跟 3 个字库+UNIGBK.BIN,总大小 3.09M,被字库占用了,不能动!
//15.10M 以后,用户可以自由使用.建议用最后的 100K 字节比较好.
//用来保存字库基本信息，地址，大小等
_font_info ftinfo;
//字库存放在磁盘中的路径
u8*const GBK24_PATH="/SYSTEM/FONT/GBK24.FON"; //GBK24 的存放位置
u8*const GBK16_PATH="/SYSTEM/FONT/GBK16.FON"; //GBK16 的存放位置
u8*const GBK12_PATH="/SYSTEM/FONT/GBK12.FON"; //GBK12 的存放位置
u8*const UNIBK_PATH="/SYSTEM/FONT/UNIBK.BIN"; //UNIBK.BIN 的存放位置
//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)
{
    .....//此处省略代码
}
//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;3,gbk24;
//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; u8 rval=0;
    u16 bread; u32 offx=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*)xpath,FA_READ);
if(res)rval=2;//打开文件失败
if(rval==0)
{
    switch(fx)
    {
        case 0:           //更新 UNIGBK.BIN
            ftinfo.ugbkaddr=FONTINFOADDR+sizeof(ftinfo);
            //信息头之后，紧跟 UNIGBK 转换码表
            ftinfo.ugbksize=fftemp->fsize;           //UNIGBK 大小
            flashaddr=ftinfo.ugbkaddr;
            break;
        case 1:
            ftinfo.f12addr=ftinfo.ugbkaddr+ftinfo.ugbksize;
            //UNIGBK 之后，紧跟 GBK12 字库
            ftinfo.gbk12size=fftemp->fsize;          //GBK12 字库大小
            flashaddr=ftinfo.f12addr;                  //GBK12 的起始地址
            break;
        case 2:
            ftinfo.f16addr=ftinfo.f12addr+ftinfo.gbk12size;
            //GBK12 之后，紧跟 GBK16 字库
            ftinfo.gbk16size=fftemp->fsize;          //GBK16 字库大小
            flashaddr=ftinfo.f16addr;                  //GBK16 的起始地址
            break;
        case 3:
            ftinfo.f24addr=ftinfo.f16addr+ftinfo.gbk16size;
            //GBK16 之后，紧跟 GBK24 字库
            ftinfo.gkb24size=fftemp->fsize;          //GBK24 字库大小
            flashaddr=ftinfo.f24addr;                  //GBK24 的起始地址
            break;
    }
    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
        W25QXX_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,GBK24 一起更新
//x,y:提示信息的显示地址
//size:字体大小
//src:字库来源磁盘."0:",SD 卡;"1:",FLASH 盘,"2:",U 盘.
//提示信息字体大小
//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 *pname; u8 res=0; u8 rval=0;
    u32 *buf; u16 i,j;
    FIL *fftemp;
    res=0xFF;
    ftinfo.fontok=0xFF;
    pname=mymalloc(SRAMIN,100); //申请 100 字节内存
    buf=mymalloc(SRAMIN,4096); //申请 4K 字节内存
    fftemp=(FIL*)mymalloc(SRAMIN,sizeof(FIL)); //分配内存
    if(buf==NULL||pname==NULL||fftemp==NULL)
    {
        myfree(SRAMIN,fftemp);
        myfree(SRAMIN,pname);
        myfree(SRAMIN,buf);
        return 5; //内存申请失败
    }
    //先查找文件是否正常
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)UNIGBK_PATH);
    res=f_open(fftemp,(const TCHAR*)pname,FA_READ);
    if(res)rval|=1<<4;//打开文件失败
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)GBK12_PATH);
    res=f_open(fftemp,(const TCHAR*)pname,FA_READ);
    if(res)rval|=1<<5;//打开文件失败
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)GBK16_PATH);
    res=f_open(fftemp,(const TCHAR*)pname,FA_READ);
    if(res)rval|=1<<6;//打开文件失败
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)GBK24_PATH);
    res=f_open(fftemp,(const TCHAR*)pname,FA_READ);
```

```
if(res)rval|=1<<7;//打开文件失败
myfree(SRAMIN,fftemp);//释放内存
if(rval==0)//字库文件都存在.
{
    LCD_ShowString(x,y,240,320,size,"Erasing sectors...");//提示正在擦除扇区
    for(i=0;i<FONTSECSIZE;i++) //先擦除字库区域,提高写入速度
    {
        fupd_prog(x+20*size/2,y,size,FONTSECSIZE,i);//进度显示
        W25QXX_Read((u8*)buf,((FONTINFOADDR/4096)+i)*4096,4096);
        //读出整个扇区的内容
        for(j=0;j<1024;j++) if(buf[j]!=0xFFFFFFFF)break;//校验数据, 是否需要擦除
        if(j!=1024)W25QXX_Erase_Sector((FONTINFOADDR/4096)+i);//擦除扇区
    }
    myfree(SRAMIN,buf);
    LCD_ShowString(x,y,240,320,size,"Updating UNIGBK.BIN");
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)UNIGBK_PATH);
    res=updata_fontx(x+20*size/2,y,size,pname,0); //更新 UNIGBK.BIN
    if(res){myfree(SRAMIN,pname);return 1;}
    LCD_ShowString(x,y,240,320,size,"Updating GBK12.BIN ");
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)GBK12_PATH);
    res=updata_fontx(x+20*size/2,y,size,pname,1); //更新 GBK12.FON
    if(res){myfree(SRAMIN,pname);return 2;}
    LCD_ShowString(x,y,240,320,size,"Updating GBK16.BIN ");
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)GBK16_PATH);
    res=updata_fontx(x+20*size/2,y,size,pname,2); //更新 GBK16.FON
    if(res){myfree(SRAMIN,pname);return 3;}
    LCD_ShowString(x,y,240,320,size,"Updating GBK24.BIN ");
    strcpy((char*)pname,(char*)src); //copy src 内容到 pname
    strcat((char*)pname,(char*)GBK24_PATH);
    res=updata_fontx(x+20*size/2,y,size,pname,3); //更新 GBK24.FON
    if(res){myfree(SRAMIN,pname);return 4;}
    //全部更新好了
    ftinfo.fontok=0XAA;
    W25QXX_Write((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo)); //保存字库信息
}
myfree(SRAMIN,pname);//释放内存
myfree(SRAMIN,buf);
return rval;//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)
{
    u8 t=0;
    W25QXX_Init();
    while(t<10)//连续读取 10 次,都是错误,说明确实是存在问题,得更新字库了
    {
        t++;
        W25QXX_Read((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo));//读 ftinfo 结构体
        if(ftinfo.fontok==0XAA)break;
        delay_ms(20);
    }
    if(ftinfo.fontok!=0XAA)return 1;
    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 W25Q128 的前 12M 字节给 FATFS 管理（用做本地磁盘），随后，紧跟字库结构体、UNIGBK.bin、和三个字库，这部分内容首地址是：(1024*12)*1024，大小约 3.09M，最后 W25Q128 还剩下约 0.9M 给用户自己用。

接下来我们打开 fontupd.h 文件代码如下：

```

#ifndef __FONTUPD_H__
#define __FONTUPD_H__
#include <stm32f4xx.h>

//字体信息保存地址,占 33 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 ubkaddr;         //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 gbk16size;       //gbk16 的大小
    u32 f24addr;         //gbk24 地址
    u32 gbk24size;       //gbk24 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 33 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。

接下来打开 text.c 文件，代码如下：

```
//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/8+((size%8)?1:0))*(size) bytes 大小  
//size:字体大小  
void Get_HzMat(unsigned char *code,unsigned char *mat,u8 size)  
{  
    unsigned char qh,ql;  
    unsigned char i;  
    unsigned long fooffset;  
    u8 csize=(size/8+((size%8)?1:0))*(size); //得到字体一个字符对应点阵集所占的字节数  
    qh=*code;  
    ql=*(++code);  
    if(qh<0x81||ql<0x40||ql==0xff||qh==0xff)//非 常用汉字  
    {  
        for(i=0;i<csize;i++) *mat++=0x00; //填充满格  
        return; //结束访问  
    }  
    if(ql<0x7f)ql-=0x40; //注意!  
    else ql-=0x41;  
    qh-=0x81;  
    fooffset=((unsigned long)190*qh+ql)*csize; //得到字库中的字节偏移量  
    switch(size)  
    {  
        case 12:W25QXX_Read(mat,foffset+ftinfo.f12addr,csize); break;  
        case 16:W25QXX_Read(mat,foffset+ftinfo.f16addr,csize);break;  
        case 24:W25QXX_Read(mat,foffset+ftinfo.f24addr,csize);break;  
    }  
}  
//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[72];  
    u8 csize=(size/8+((size%8)?1:0))*(size); //得到字体一个字符对应点阵集所占的字节数  
    if(size!=12&&size!=16&&size!=24) return; //不支持的 size  
    Get_HzMat(font,dzk,size); //得到相应大小的点阵数据  
    for(t=0;t<csize;t++)  
    {  
        temp=dzk[t]; //得到点阵数据  
        for(t1=0;t1<8;t1++)  
        {  
            if(temp&0x80)LCD_Fast_DrawPoint(x,y,POINT_COLOR);  
            else if(mode==0)LCD_Fast_DrawPoint(x,y,BACK_COLOR);  
            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 45.1 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。

text.h 头文件是一些函数申明，我们这里不细说了。

前面提到我们对 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 删掉，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，详见本例程源码。

最后，我们看看 main 函数如下：

```
int main(void)
{
    u32 fontcnt; u8 i,j; u8 key,t;
    u8 fontx[2];//gbk 码
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    W25QXX_Init(); //初始化 W25Q128
    usmart_dev.init(168); //初始化 USMART
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    exfun_init(); //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1); //挂载 SD 卡
    f_mount(fs[1],"1:",1); //挂载 FLASH.
    while(font_init()) //检查字库
    {
        UPD:
        LCD_Clear(WHITE); //清屏
        POINT_COLOR=RED; //设置字体为红色
        LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
        while(SD_Init()) //检测 SD 卡
        {
            LCD_ShowString(30,70,200,16,16,"SD Card Failed!"); delay_ms(200);
            LCD_Fill(30,70,200+30,70+16,WHITE); delay_ms(200);
        }
        LCD_ShowString(30,70,200,16,16,"SD Card OK");
        LCD_ShowString(30,90,200,16,16,"Font Updating...");
        key=update_font(20,110,16,"0:");//更新字库
        while(key)//更新失败
        {
            LCD_ShowString(30,110,200,16,16,"Font Update Failed!"); delay_ms(200);
            LCD_Fill(20,110,200+20,110+16,WHITE); delay_ms(200);
        }
        LCD_ShowString(30,110,200,16,16,"Font Update Success! ");
        delay_ms(1500);
    }
}
```

```
LCD_Clear(WHITE);//清屏
}

POINT_COLOR=RED;
Show_Str(30,50,200,16,"探索者 STM32F407 开发板",16,0);
Show_Str(30,70,200,16,"GBK 字库测试程序",16,0);
Show_Str(30,90,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(30,110,200,16,"2014 年 5 月 15 日",16,0);
Show_Str(30,130,200,16,"按 KEY0,更新字库",16,0);
POINT_COLOR=BLUE;
Show_Str(30,150,200,16,"内码高字节:",16,0);
Show_Str(30,170,200,16,"内码低字节:",16,0);
Show_Str(30,190,200,16,"汉字计数器:",16,0);
Show_Str(30,220,200,24,"对应汉字为:",24,0);
Show_Str(30,244,200,16,"对应汉字(16*16)为:",16,0);
Show_Str(30,260,200,12,"对应汉字(12*12)为:",12,0);
while(1)
{
    fontcnt=0;
    for(i=0x81;i<0xff;i++)
    {
        fontx[0]=i;
        LCD_ShowNum(118,150,i,3,16);      //显示内码高字节
        for(j=0x40;j<0xfe;j++)
        {
            if(j==0x7f)continue;
            fontcnt++;
            LCD_ShowNum(118,170,j,3,16); //显示内码低字节
            LCD_ShowNum(118,190,fontcnt,5,16); //汉字计数显示
            fontx[1]=j;
            Show_Font(30+132,220,fontx,24,0);
            Show_Font(30+144,244,fontx,16,0);
            Show_Font(30+108,260,fontx,12,0);
            t=200;
            while(t--)//延时,同时扫描按键
            {
                delay_ms(1);
                key=KEY_Scan(0);
                if(key==KEY0_PRES)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 45.3.1 所示：

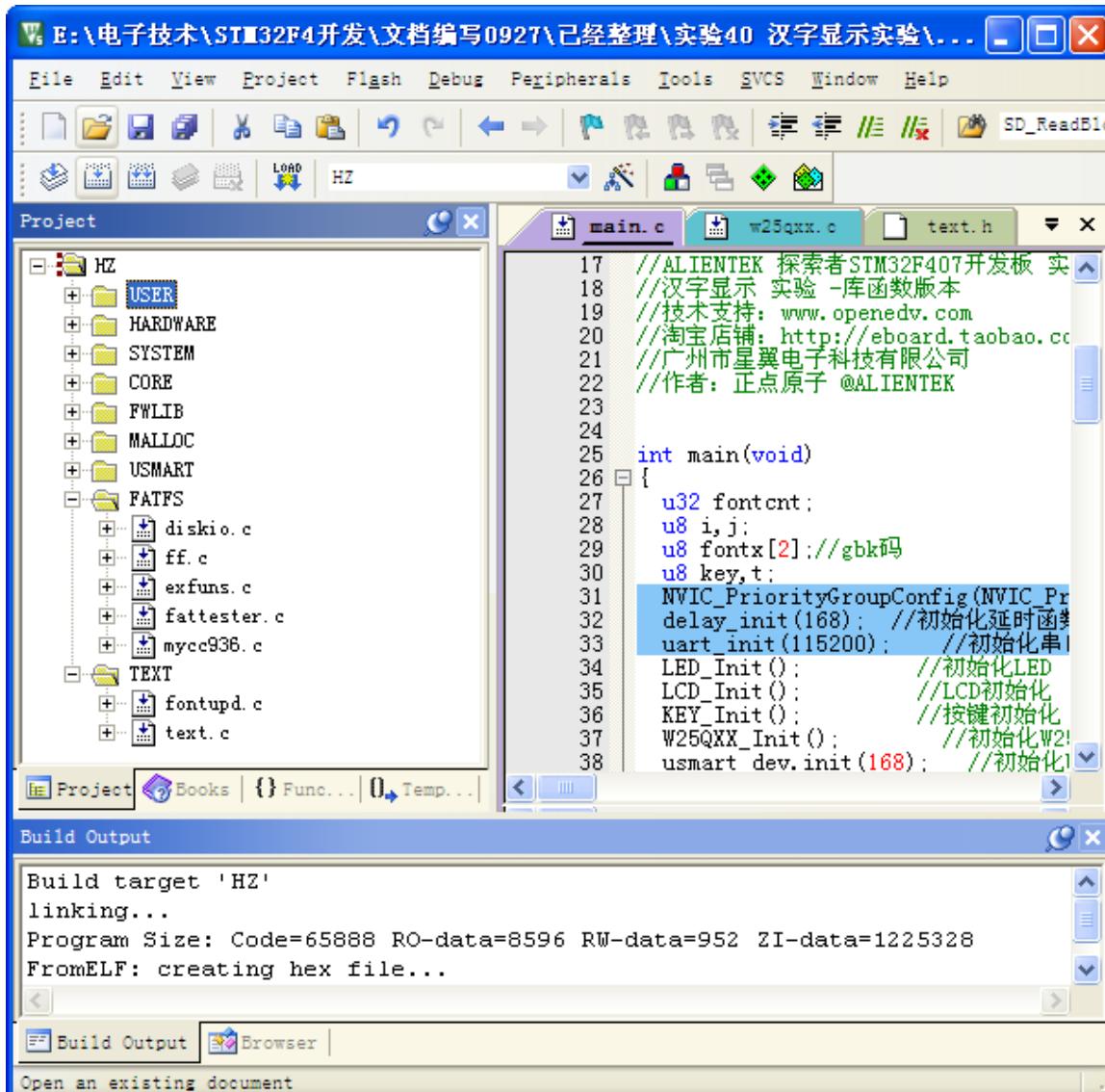


图 45.3.1 工程建成截图

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 探索者 STM32F4 开发板上，可以看到 LCD 开始显示汉字及汉字内码，如图 45.4.1 所示：

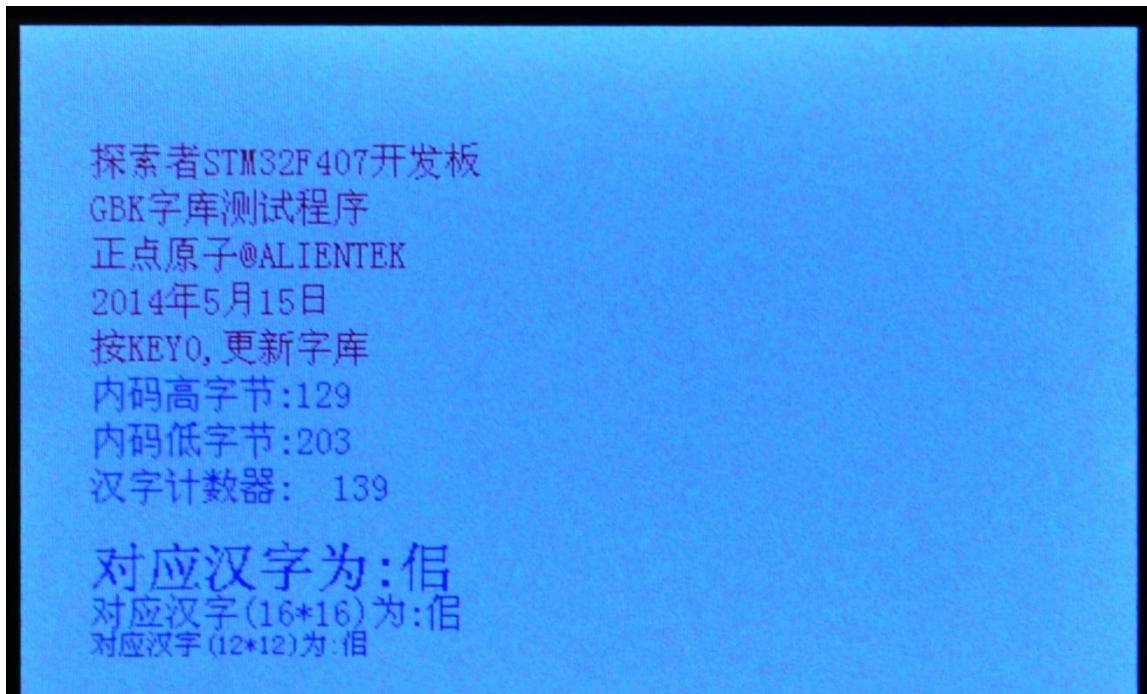


图 45.4.1 汉字显示实验显示效果

一开始就显示汉字，是因为 ALIENTEK 探索者 STM32F4 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 W25Q128 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：光盘\5，SD 卡根目录文件 文件夹下面的 SYSTEM 文件夹拷贝到 SD 卡根目录下，插入开发板，并按复位，之后，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 45.4.2 所示：

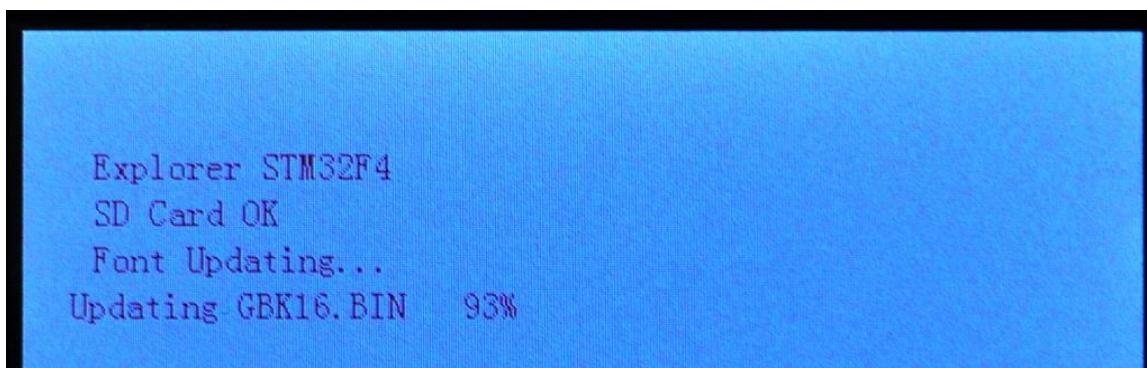


图 45.4.2 汉字字库更新界面

第四十六章 图片显示实验

在开发产品的时候，很多时候，我们都会用到图片解码，在本章中，我们将向大家介绍如何通过 STM32F4 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：

- 46.1 图片格式简介
- 46.2 硬件设计
- 46.3 软件设计
- 46.4 下载验证

46.1 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\图片编解码\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 Welch)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。

46.2 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 KEY2 可以快速浏览下一张和上一张，KEY_UP 按键用于暂停/继续播放，DS1 用于指示当前是否处于暂停状态。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 和 DS1
- 2) KEY0、KEY2 和 KEY_UP 三个按键
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡
- 6) 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。

46.3 软件设计

打开本章实验工程目录可以看到，我们在工程根目录下面新建了一个 PICTURE 文件夹。在该文件夹里面新建了 bmp.c、bmp.h、tjpd.c、tjpd.h、integer.h、gif.c、gif.h、piclib.c 和 piclib.h 等 9 个文件。打开实验工程可以看到，我们在工程中新建了 PICTURE 分组，添加了相关源文件到工程，同时将 PICTURE 文件夹加入头文件包含路径。

对于这些文件，其中 bmp.c 和 bmp.h 用于实现对 bmp 文件的解码；tjpd.c 和 tjpd.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，所以我们在这里不贴出来，请大家参考光盘本例程的源码，我们打开 piclib.c，代码如下：

```
_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);
}
//填充颜色
//x,y:起始坐标
//width, height: 宽度和高度。
//color: 颜色数组
void piclib_fill_color(u16 x,u16 y,u16 width,u16 height,u16 *color)
{
    LCD_Color_Fill(x,y,x+width-1,y+height-1,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,仅 BMP 需要
    pic_phy.draw_point=LCD_Fast_DrawPoint; //画点函数实现
    pic_phy.fill=LCD_Fill;                //填充函数实现,仅 GIF 需要
    pic_phy.draw_hline=piclib_draw_hline;   //画线函数实现,仅 GIF 需要
}
```

```
pic_phy.fillcolor=piclib_fill_color;           //颜色填充函数实现,仅 TJPGD 需要
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;
    //Convert to 32bit |----GGGGGG----RRRRR-----BBBBB|
    src2=((src<<16)|src)&0x07E0F81F;
    dst2=((dst<<16)|dst)&0x07E0F81F;
    //Perform blending R:G:B with alpha in range 0..32
    //Note that the reason that alpha may not exceed 32 is that there are only
    //5bits of space between each R:G:B value, any higher value will overflow
    //into the next component and deliver ugly result.
    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
picinfo.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.
//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)>picinfo.lcdwidth)return PIC_WINDOW_ERR; //x 坐标超范围了.
    if((y+height)>picinfo.lcdheight)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;
    }
    if(pic_phy.fillcolor==NULL)fast=0;//颜色填充函数未实现,不能快速显示
    //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);           //解码 bmp
        break;
    case T_JPG:
    case T_JPEG:
        res=jpg_decode(filename,fast);         //解码 JPG/JPEG
        break;
    case T_GIF:
        res=gif_decode(filename,x,y,width,height); //解码 gif
        break;
    default:
        res=PIC_FORMAT_ERR;                  //非图片格式!!!
        break;
}
return res;
}
//动态分配内存
void *pic_malloc (u32 size)
{
    return (void*)mymalloc(SRAMIN,size);
}
//释放内存
void pic_memfree (void* mf)
{
    myfree(SRAMIN,mf);
}
```

此段代码总共 9 个函数，其中，`piclib_draw_hline` 和 `piclib_fill_color` 函数因为 LCD 驱动代码没有提供，所以在这里单独实现，如果 LCD 驱动代码有提供，则直接用 LCD 提供的即可。

`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（图片分辨率小于 LCD 分辨率）`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，具体请参考光盘本例程源码。

最后，`pic_malloc` 和 `pic_memfree` 分别用于图片解码时需要用到的内存申请和释放，通过调用 `mymalloc` 和 `myfreee` 来实现。

接下来我们看看头文件 `piclib.h` 代码如下：

```
#ifndef __PICLIB_H
#define __PICLIB_H
.....//圣罗头文件引入
#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
#ifndef TRUE
#define TRUE     1
#endif
#ifndef FALSE
#define FALSE    0
#endif
//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);
    //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);
    //void draw_hline(u16 x0,u16 y0,u16 len,u16 color) 画水平线函数
    void(*fillcolor)(u16,u16,u16,u16* );
    //void piclib_fill_color(u16 x,u16 y,u16 width,u16 height,u16 *color) 颜色填充
}_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_fill_color(u16 x,u16 y,u16 width,u16 height,u16 *color);
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,u8 fast);//智能画图
void *pic_malloc (u32 size);   //pic 申请内存
void pic_free (void* mf);       //pic 释放内存
#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，相信大家很容易明白。最后我们看看 main.c 文件内容如下：

```

//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_tgettell(fn);
            if((res&0XF0)==0X50) rval++;//取高四位,是否图片文件？是则加 1
        }
    }
}

```

```
return rval;
}

int main(void)
{
    u8 res; u8 t; u16 temp;
    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
    u16 *picindextbl;      //图片索引表
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init();           //初始化 LED
    usmart_dev.init(84); //初始化 USMART
    LCD_Init();           //LCD 初始化
    KEY_Init();            //按键初始化
    W25QXX_Init();        //初始化 W25Q128
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    exfun_init();          //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1); //挂载 SD 卡
    f_mount(fs[1],"1:",1); //挂载 FLASH.
    POINT_COLOR=RED;
    while(font_init()) //检查字库
    {
        LCD_ShowString(30,50,200,16,16,"Font Error!"); delay_ms(200);
        LCD_Fill(30,50,240,66,WHITE); delay_ms(200); //清除显示
    }
    Show_Str(30,50,200,16,"Explorer STM32F4 开发板",16,0);
    Show_Str(30,70,200,16,"图片显示程序",16,0);
    Show_Str(30,90,200,16,"KEY0:NEXT KEY2:PREV",16,0);
    Show_Str(30,110,200,16,"WK_UP:PAUSE",16,0);
    Show_Str(30,130,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,150,200,16,"2014 年 5 月 15 日",16,0);
    while(f_opendir(&picdir,"0:/PICTURE"))//打开图片文件夹
    {
        Show_Str(30,170,240,16,"PICTURE 文件夹错误!",16,0); delay_ms(200);
        LCD_Fill(30,170,240,186,WHITE); delay_ms(200); //清除显示
    }
}
```

```
}

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(30,170,240,16,"没有图片文件!",16,0); delay_ms(200);
    LCD_Fill(30,170,240,186,WHITE); delay_ms(200);//清除显示
}
picfileinfo.lfsize=_MAX_LFN*2+1; //长文件名最大长度
picfileinfo.lfname=mymalloc(SRAMIN,picfileinfo.lfsize); //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); //申请内存,用于存放图片索引
while(picfileinfo.lfname==NULL||pname==NULL||picindextbl==NULL)//分配出错
{
    Show_Str(30,170,240,16,"内存分配失败!",16,0); delay_ms(200);
    LCD_Fill(30,170,240,186,WHITE); delay_ms(200);//清除显示
}
//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_ttype(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(30,170,240,16,"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/");
strcat((char*)pname,(const char*)fn);           //复制路径(目录)
//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=1;                         //模拟一次按下 KEY0
    if((t%20)==0)LED0=!LED0;//LED0 闪烁,提示程序正在运行.
    if(key==KEY2_PRES)                      //上一张
    {
        if(curindex)curindex--;
        else curindex=totpicnum-1;
        break;
    }else if(key==KEY0_PRES)//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==WKUP_PRES) { pause=!pause; LED1=!pause;}//暂停?
    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。

46.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 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 46.4.1 所示：



图 46.4.1 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，KEY_UP 按键可以暂停自动播放，同时 DS1 亮，指示处于暂停状态，再按一次 KEY_UP 则继续播放。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。

第四十七章 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&JPEG 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：

- 47.1 BMP&JPEG 编码简介
- 47.2 硬件设计
- 47.3 软件设计
- 47.4 下载验证

47.1 BMP&JPEG 编码简介

本章，我们要实现的照相机，支持 BMP 图片格式的照片和 JPEG 图片格式的照片，这里简单介绍一下这两种图片格式的编码。

47.1.1 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像素。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。每个像素用 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 $4294967296(2 \text{ 的 } 32 \text{ 次方})$ 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：最高位一保留，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、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。

47.1.2 JPEG 编码简介

JPEG (Joint Photographic Experts Group) 是一个由 ISO 和 IEC 两个组织机构联合组成的一个专家组，负责制定静态的数字图像数据压缩编码标准，这个专家组开发的算法称为 JPEG 算法，并且成为国际上通用的标准，因此又称为 JPEG 标准。JPEG 是一个适用范围很广的静态图像数据压缩标准，既可用于灰度图像又可用于彩色图像。

JPEG 专家组开发了两种基本的压缩算法，一种是采用以离散余弦变换（Discrete Cosine Transform, DCT）为基础的有损压缩算法，另一种是采用以预测技术为基础的无损压缩算法。使用有损压缩算法时，在压缩比为 25:1 的情况下，压缩后还原得到的图像与原始图像相比较，非图像专家难于找出它们之间的区别，因此得到了广泛的应用。

JPEG 压缩是有损压缩，它利用了人的视角系统的特性，使用量化和无损压缩编码相结合来去掉视角的冗余信息和数据本身的冗余信息。

JPEG 压缩编码分为三个步骤：

1) 使用正向离散余弦变换（Forward Discrete Cosine Transform, FDCT）把空间域表示的图转换成频率域表示的图。

2) 使用加权函数对 DCT 系数进行量化，这个加权函数对于人的视觉系统是最佳的。

3) 使用霍夫曼可变字长编码器对量化系数进行编码。

这里我们不详细介绍 JPEG 压缩的过程了，大家可以自行查找相关资料。我们本章要实现的 JPEG 拍照，并不需要自己压缩图像，因为我们使用的 ALIENTEK OV2640 摄像头模块，直接就可以输出压缩后的 JPEG 数据，我们完全不需要理会压缩过程，所以本章我们实现 JPEG 拍照的关键，在于准确接收 OV2640 摄像头模块发送过来的编码数据，然后将这些数据保存为.jpg 文件，就可以实现 JPEG 拍照了。

在第四十章，我们定义了一个很大的数组 jpeg_buf (124KB) 来存储 JPEG 图像数据，但本章，我们要用到内存管理，其他地方也要用到一些数组，所以，肯定无法再定义这么大的数组了。并且这个数组不能使用外部 SRAM (实测：DCMI 接口使用 DMA 直接传输 JPEG 数据到外部 SRAM 会出现数据丢失，所以 DMA 接收 JPEG 数据只能用内部 SRAM)，所以，我们本章将使用 DMA 的双缓冲机制来读取，DMA 双缓冲读取 JPEG 数据框图如图 47.1.2.1 所示：

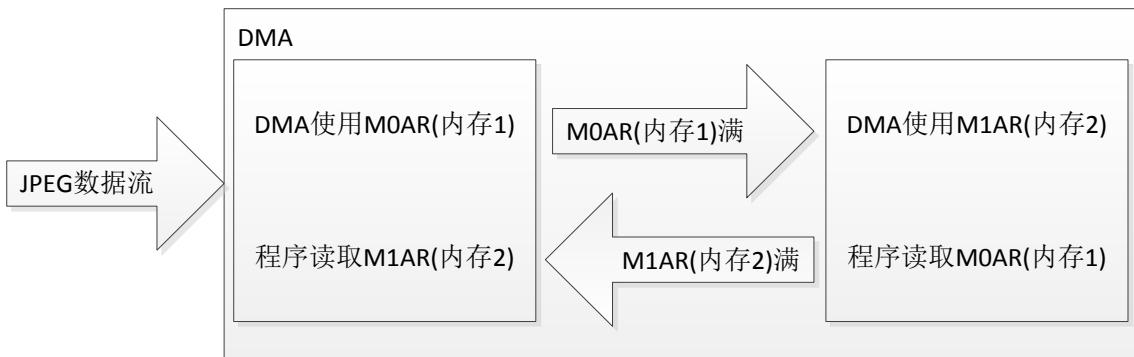


图 47.1.2.1 DMA 双缓冲读取 JPEG 数据原理框图

DMA 接收来自 OV2640 的 JPEG 数据流，首先使用 M0AR (内存 1) 来存储，当 M0AR 满了以后，自动切换到 M1AR (内存 2)，同时程序读取 M0AR (内存 1) 的数据到外部 SRAM；当 M1AR 满了以后，又切回 M0AR，同时程序读取 M1AR (内存 2) 的数据到外部 SRAM；依次循环(此时的数据处理，是通过 DMA 传输完成中断实现的，在中断里面处理)，直到帧中断，结束一帧数据的采集，读取剩余数据到外部 SRAM，完成一次 JPEG 数据的采集。

这里，M0AR，M1AR 所指向的内存，必须是内部内存，不过由于采用了双缓冲机制，我

们就不必定义一个很大的数组，一次性接收所有 JPEG 数据了，而是可以分批次接收，数组可以定义的比较小。

最后，将存储在外部 SRAM 的 jpeg 数据，保存为.jpg/.jpeg 存放在 SD 卡，就完成了一次 JPEG 拍照。

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 OV2640，在初始化成功之后，就一直在屏幕显示 OV2640 拍到的内容。当按下 KEY_UP 按键的时候，可以选择缩放，还是 1:1 显示，默认缩放。按下 KEY0，可以拍 bmp 图片照片（分辨率为：LCD 分辨率）。按下 KEY1 可以拍 JPEG 图片照片（分辨率为 UXGA，即 1600*1200）。拍照保存成功之后，蜂鸣器会发出“滴”的一声，提示拍照成功。DS0 还是用于指示程序运行状态，DS1 用于提示 DCMI 帧中断。

所要用到的硬件资源如下：

- 1) 指示灯 DS0 和 DS1
- 2) KEY0、KEY1 和 KEY_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。需要注意的是：SD 卡与 DCMI 接口有部分 IO 共用，所以他们不能同时使用，必须分时复用，本章，这部分共用 IO 我们只有在拍照保存的时候，才切换为 SD 卡使用，其他时间，都是被 DCMI 占用的。

47.3 软件设计

打开本章实验工程，由于本章要用到 OV2640、蜂鸣器、外部 SRAM 和定时器等外设，所以，先添加了 dcmi.c、sccb.c、ov2640.c、beep.c、sram.c 和 timer.c 等文件到 HARDWARE 组下。

然后，我们来看下 PICTURE 组下的 bmp.c 文件里面的 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,需要利用原来的调色板位置增加掩码.这里我们增加了掩码.
//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,仅创建新文件;1,如果存在文件,则覆盖该文件.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; u8 res=0;
    u16 bmpheadsize; //bmp 头大小
```

```

BITMAPINFO hbmp;           //bmp 头
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*)pic_memalloc(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 *)pic_memalloc(sizeof(FIL)); //开辟 FIL 字节的内存区域
    if(f_bmp==NULL){pic_memfree(databuf); return PIC_MEM_ERR; } //内存申请失败.
#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)); //置零清空申请到的内存.
    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
pic_memfree(databuf); pic_memfree(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 47.1.1 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,保存个为 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。

在 jpeg 拍照的时候,我们使用了双缓冲机制,且用到了 DMA 传输完成中断,这里我们需要修改 dcmi.c 里面的 DCMI_DMA_Init 函数,并添加 DMA 传输完成中断服务函数,代码如下:

```

//DCMI DMA 配置
//memaddr:存储器地址    将要存储摄像头数据的内存地址(也可以是外设地址)
//DMA_BufferSize:存储器长度    0~65535
//DMA_MemoryDataSize:存储器位宽
//DMA_MemoryInc:存储器增长方式 @defgroup DMA_memory_incremented_mode
void DCMI_DMA_Init(u32 DMA_Memory0BaseAddr,u32 DMA_Memory1BaseAddr,
                    u16 DMA_BufferSize,u32 DMA_MemoryDataSize,u32 DMA_MemoryInc)
{
    DMA_InitTypeDef  DMA_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2,ENABLE);//DMA2 时钟使能
    DMA_DeInit(DMA2_Stream1);//等待 DMA2_Stream1
    while (DMA_GetCmdStatus(DMA2_Stream1) != DISABLE){}//等待可配置
}

```

```

/* 配置 DMA Stream */
DMA_InitStructure.DMA_Channel = DMA_Channel_1; //通道 1 DCMI 通道
DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&DCMI->DR;//外设地址
DMA_InitStructure.DMA_Memory0BaseAddr = DMA_Memory0BaseAddr;//存储器 0 地址
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//外设到存储器模式
DMA_InitStructure.DMA_BufferSize = DMA_BufferSize;//数据传输量
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc;//存储器增量模式
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize;//存储器数据长度
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
DMA_InitStructure.DMA_Priority = DMA_Priority_High;//高优先级
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Enable; //FIFO 模式
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;//使用全 FIFO
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;//外设突发单次传输
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
DMA_Init(DMA2_Stream1, &DMA_InitStructure);//初始化 DMA Stream

if(DMA_Memory1BaseAddr)
{
    DMA_DoubleBufferModeCmd(DMA2_Stream1,ENABLE); //双缓冲模式
    DMA_MemoryTargetConfig(DMA2_Stream1,DMA_Memory1BaseAddr,
                           DMA_Memory_1); //配置目标地址 1
    DMA_ITConfig(DMA2_Stream1,DMA_IT_TC,ENABLE); //开启传输完成中断
    NVIC_InitStructure.NVIC_IRQChannel = DMA2_Stream1 IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0; //抢占优先级 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority =0; //响应优先级 0
    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、
}
}

void (*dcmi_rx_callback)(void); //DCMI DMA 接收回调函数
//DMA2_Stream1 中断服务函数(仅双缓冲模式会用到)
void DMA2_Stream1_IRQHandler(void)
{
    if(DMA_GetFlagStatus(DMA2_Stream1,DMA_FLAG_TCIF1)==SET)
        //DMA2_Stream1,传输完成标志
    {
        DMA_ClearFlag(DMA2_Stream1,DMA_FLAG_TCIF1); //清除传输完成中断
        dcmi_rx_callback(); //执行摄像头接收回调函数,读取数据等操作在这里处理
    }
}

```

这里, DCMI_DMA_Init 函数, 和第四十章相比增加了一个参数: DMA_Memory1BaseAddr,

用于设置 DMA 的第二个内存地址，同时根据该值是否为非 0，来判断是否需要使用双缓冲机制，只有在双缓冲机制下，才开启 DMA 传输完成中断。DMA2_Stream1_IRQHandler 函数，是 DMA 传输完成中断，里面通过 dcmi_rx_callback 回调函数(函数指针，指向 jpeg_dcni_rx_callback 函数)，及时将满了的内存 (M0AR 或 M1AR) 数据读取到外部 SRAM。

最后我们来看看 main.c 文件源码：

```

u8 ov2640_mode=0;           //工作模式:0,RGB565 模式;1,JPEG 模式
#define jpeg_dma_bufsize 5*1024 //定义 JPEG DMA 接收时数据缓存大小(*4 字节)
volatile u32 jpeg_data_len=0; //buf 中的 JPEG 有效数据长度(*4 字节)
volatile u8 jpeg_data_ok=0;   //JPEG 数据采集完成标志
                            //0,数据没有采集完;
                            //1,数据采集完了,但是还没处理;
                            //2,数据已经处理完成了,可以开始下一帧接收
u32 *jpeg_buf0;            //JPEG 数据缓存 buf,通过 malloc 申请内存
u32 *jpeg_buf1;            //JPEG 数据缓存 buf,通过 malloc 申请内存
u32 *jpeg_data_buf;        //JPEG 数据缓存 buf,通过 malloc 申请内存
//处理 JPEG 数据
//当采集完一帧 JPEG 数据后,调用此函数,切换 JPEG BUF.开始下一帧采集.
void jpeg_data_process(void)
{
    u16 i; u16 rlen;//剩余数据长度
    u32 *pbuf;
    if(ov2640_mode)//只有在 JPEG 格式下,才需要做处理.
    {
        if(jpeg_data_ok==0) //jpeg 数据还未采集完?
        {
            DMA_Cmd(DMA2_Stream1,DISABLE); //停止当前传输
            while(DMA_GetCmdStatus(DMA2_Stream1) != DISABLE);//等待可配置
            rlen=jpeg_dma_bufsize- DMA_GetCurrDataCounter(DMA2_Stream1)
                ;//得到剩余数据长度
            pbuf=jpeg_data_buf+jpeg_data_len;//偏移到有效数据末尾,继续添加
            if(DMA2_Stream1->CR&(1<<19))for(i=0;i<rlen;i++)pbuf[i]=jpeg_buf1[i];
            //读取 buf1 里面的剩余数据
            else for(i=0;i<rlen;i++)pbuf[i]=jpeg_buf0[i];//读取 buf0 里面的剩余数据
            jpeg_data_len+=rlen; //加上剩余长度
            jpeg_data_ok=1; //标记 JPEG 数据采集完按成,等待其他函数处理
        }
        if(jpeg_data_ok==2) //上一次的 jpeg 数据已经被处理了
        {
            DMA_SetCurrDataCounter(DMA2_Stream1,jpeg_dma_bufsize);
                //传输长度为 jpeg_buf_size*4 字节
            DMA_Cmd(DMA2_Stream1,ENABLE);//重新传输
            jpeg_data_ok=0; //标记数据未采集
            jpeg_data_len=0; //数据重新开始
        }
    }
}

```

```
        }
    }
}

//jpeg 数据接收回调函数
void jpeg_dcmi_rx_callback(void)
{
    u16 i; u32 *pbuf;
    pbuf=jpeg_data_buf+jpeg_data_len;//偏移到有效数据末尾
    if(DMA2_Stream1->CR&(1<<19))//buf0 已满,正常处理 buf1
    {
        for(i=0;i<jpeg_dma_bufsize;i++)pbuf[i]=jpeg_buf0[i];//读取 buf0 里面的数据
        jpeg_data_len+=jpeg_dma_bufsize;//偏移
    }else //buf1 已满,正常处理 buf0
    {
        for(i=0;i<jpeg_dma_bufsize;i++)pbuf[i]=jpeg_buf1[i];//读取 buf1 里面的数据
        jpeg_data_len+=jpeg_dma_bufsize;//偏移
    }
}
//切换为 OV2640 模式 (GPIOC8/9/11 切换为 DCMI 接口)
void sw_ov2640_mode(void)
{
    OV2640_PWDN=0;//OV2640 Power Up
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource8,GPIO_AF_DCMI);
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource9,GPIO_AF_DCMI);
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource11,GPIO_AF_DCMI);
}
//切换为 SD 卡模式 (GPIOC8/9/11 切换为 SDIO 接口)
void sw_sdcard_mode(void)
{
    OV2640_PWDN=1;//OV2640 Power Down
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource8,GPIO_AF_SDIO); //PC8,AF12
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource9,GPIO_AF_SDIO); //PC9,AF12
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource11,GPIO_AF_SDIO);
} //文件名自增 (避免覆盖)
//mode:0,创建.bmp 文件;1,创建.jpg 文件.
//bmp 组合成:形如"0:PHOTO/PIC13141.bmp"的文件名
//jpg 组合成:形如"0:PHOTO/PIC13141.jpg"的文件名
void camera_new_pathname(u8 *pname,u8 mode)
{
    u8 res; u16 index=0;
    while(index<0xFFFF)
    {
        if(mode==0)sprintf((char*)pname,"0:PHOTO/PIC%05d.bmp",index);
    }
}
```

```
else sprintf((char*)pname,"0:PHOTO/PIC%05d.jpg",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++;
}
//OV2640 拍照 jpg 图片
//返回值:0,成功
// 其他,错误代码
u8 ov2640_jpg_photo(u8 *pname)
{
    FIL* f_jpg; u8* pbuf;
    u8 res=0; u32 bwr; u16 i;
    f_jpg=(FIL *)mymalloc(SRAMIN,sizeof(FIL)); //开辟 FIL 字节的内存区域
    if(f_jpg==NULL)return 0XFF; //内存申请失败.
    ov2640_mode=1;
    sw_ov2640_mode(); //切换为 OV2640 模式
    dcmi_rx_callback=jpeg_dcmi_rx_callback;//回调函数
    DCMI_DMA_Init((u32)jpeg_buf0,(u32)jpeg_buf1,jpeg_dma_bufsize,
        DMA_MemoryDataSize_Word,DMA_MemoryInc_Enable); //双缓冲模式
    OV2640_JPEG_Mode(); //切换为 JPEG 模式
    OV2640_ImageWin_Set(0,0,1600,1200);
    OV2640_OutSize_Set(1600,1200); //拍照尺寸为 1600*1200
    DCMI_Start(); //启动传输
    while(jpeg_data_ok!=1); //等待第一帧图片采集完
    jpeg_data_ok=2; //忽略本帧图片,启动下一帧采集
    while(jpeg_data_ok!=1); //等待第二帧图片采集完,第二帧,才保存到 SD 卡去.
    DCMI_Stop(); //停止 DMA 搬运
    ov2640_mode=0;
    sw_sdcard_mode(); //切换为 SD 卡模式
    res=f_open(f_jpg,(const TCHAR*)pname,FA_WRITE|FA_CREATE_NEW);
    //模式 0,或者尝试打开失败,则创建新文件
    if(res==0)
    {
        printf("jpeg data size:%d\r\n",jpeg_data_len*4); //串口打印 JPEG 文件大小
        pbuf=(u8*)jpeg_data_buf;
        for(i=0;i<jpeg_data_len*4;i++) //查找 0xFF,0xD8
        {
            if((pbuf[i]==0xFF)&&(pbuf[i+1]==0xD8))break;
        }
        if(i==jpeg_data_len*4)res=0xFD; //没找到 0xFF,0xD8
        else//找到了
        {

```

```
    pbuf+=i;//偏移到 0XFF,0XD8 处
    res=f_write(f_jpg,pbuf,jpeg_data_len*4-i,&bwr);
    if(bwr!=(jpeg_data_len*4-i))res=0XFE;
}
jpeg_data_len=0;
f_close(f_jpg);
sw_ov2640_mode(); //切换为 OV2640 模式
OV2640_RGB565_Mode(); //RGB565 模式
DCMI_DMA_Init((u32)&LCD->LCD_RAM,0,1,DMA_MemoryDataSize_HalfWord,
               DMA_MemoryInc_Disable); //DCMI DMA 配置
myfree(SRAMIN,f_jpg);
return res;
}
int main(void)
{
    u8 res; u8 i;
    u8 *pname; //带路径的文件名
    u8 key; //键值
    u8 sd_ok=1; //0,SD 卡不正常;1,SD 卡正常.
    u8 scale=1; //默认是全尺寸缩放
    u8 msgbuf[15]; //消息缓存区
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    .....//省略部分代码
    while(font_init()) //检查字库
    {
        LCD_ShowString(30,50,200,16,16,"Font Error!"); delay_ms(200);
        LCD_Fill(30,50,240,66,WHITE); delay_ms(200); //清除显示
    }
    .....//省略部分代码
    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(30,150,240,16,"SD 卡错误!",16,0); delay_ms(200);
        Show_Str(30,170,240,16,"拍照功能将不可用!",16,0); sd_ok=0;
    }
    jpeg_buf0=mymalloc(SRAMIN,jpeg_dma_bufsize*4); //为 jpeg dma 接收申请内存
    jpeg_buf1=mymalloc(SRAMIN,jpeg_dma_bufsize*4); //为 jpeg dma 接收申请内存
    jpeg_data_buf=mymalloc(SRAMEX,300*1024); //为 jpeg 文件申请内存(最大 300KB)
    pname=mymalloc(SRAMIN,30); //为带路径的文件名分配 30 个字节的内存
    while(pname==NULL||!jpeg_buf0||!jpeg_buf1||!jpeg_data_buf) //内存分配出错
    {
        Show_Str(30,190,240,16,"内存分配失败!",16,0); delay_ms(200);
    }
```

```
LCD_Fill(30,190,240,146,WHITE); delay_ms(200); //清除显示
}
while(OV2640_Init())//初始化 OV2640
{
    Show_Str(30,190,240,16,"OV2640 错误!",16,0); delay_ms(200);
    LCD_Fill(30,190,239,206,WHITE); delay_ms(200);
}
Show_Str(30,190,200,16,"OV2640 正常",16,0);
delay_ms(2000);
OV2640_RGB565_Mode(); //JPEG 模式
My_DCMI_Init(); //DCMI 配置
DCMI_DMA_Init((u32)&LCD->LCD_RAM,0,1,DMA_MemoryDataSize_HalfWord,
                DMA_MemoryInc_Disable); //DCMI DMA 配置
OV2640_OutSize_Set(lcddev.width,lcddev.height);
DCMI_Start(); //启动传输
while(1)
{
    key=KEY_Scan(0); //不支持连接
    if(key)
    {
        DCMI_Stop(); //停止显示
        if(key==WKUP_PRES) //缩放 or 1:1 显示
        {
            scale=!scale;
            .....//省略部分代码
        }
        else if(sd_ok)//SD 卡正常才可以拍照
        {
            sw_sdcard_mode(); //切换为 SD 卡模式
            if(key==KEY0_PRES) //BMP 拍照
            {
                camera_new_pathname(pname,0); //得到文件名
                res=bmp_encode(pname,0,0	lcddev.width,lcddev.height,0);
            }
            else if(key==KEY1_PRES)//JPG 拍照
            {
                camera_new_pathname(pname,1); //得到文件名
                res=ov2640_jpg_photo(pname);
                if(scale==0)
                {
                    OV2640_ImageWin_Set((1600-lcddev.width)/2,(1200-lcddev.height)/2,lcddev.width,lcddev.height); //1:1 真实尺寸
                    OV2640_OutSize_Set(lcddev.width,lcddev.height);
                }
                else OV2640_ImageWin_Set(0,0,1600,1200); //全尺寸缩放
                OV2640_OutSize_Set(lcddev.width,lcddev.height);
            }
        }
    }
}
```

```

        }
        sw_ov2640_mode(); //切换为 OV2640 模式
        if(res) Show_Str(30,130,240,16,"写入文件错误!",16,0);//拍照有误
        else
        {
            Show_Str(30,130,240,16,"拍照成功!",16,0);
            Show_Str(30,150,240,16,"保存为:",16,0);
            Show_Str(30+42,150,240,16,pname,16,0);
            BEEP=1; delay_ms(100); //蜂鸣器短叫, 提示拍照完成
        }
    }else //提示 SD 卡错误
    {
        Show_Str(30,130,240,16,"SD 卡错误!",16,0);
        Show_Str(30,150,240,16,"拍照功能不可用!",16,0);
    }
    BEEP=0; //关闭蜂鸣器
    if(key!=WKUP_PRES)delay_ms(1800); //非尺寸切换,等待 1.8 秒钟
    DCMI_Start(); //停止显示
}
delay_ms(10); i++;
if(i==20) { i=0; LED0=!LED0; } //DS0 闪烁.
}
}

```

此部分代码有点多，main 函数里面我们省略了部分代码，以节省篇幅。接下来分别介绍下这些函数。

`jpeg_data_process` 函数，该函数在 DCMI 帧中断里面被调用，当 JPEG 拍照时，该函数用于将最后接收到的 JPEG 数据拷贝到外部 SRAM (`jpeg_data_buf`) 里面，并标记 JPEG 数据采集完成，同时该函数也可以启动下一次 JPEG 采集。

`jpeg_dcmi_rx_callback` 函数，该函数是 DCMI 的 DMA 传输完成中断，循环读取 M0AR 和 M1AR 的数据，存放到外部 SRAM (`jpeg_data_buf`) 里面。

`sw_ov2640_mode` 和 `sw_sdcard_mode` 函数，用于设置 SDIO 和 DCMI 共用的几个 IO 到底归哪个外设使用，以实现分时复用。

`camera_new_pathname` 函数，则用根据拍照类型 (bmp/jpg) 创建新文件名，保证文件名和 SD 卡原有的照片不重复（避免覆盖）。

`ov2640_jpg_photo` 函数，则用于 jpg 拍照，该函数设置 OV2640 为 JPEG 输出，然后设置分辨率为 1600*1200。为了确保数据完整，jpg 拍照选择的是第二帧数据（第一帧可能不完整，直接丢弃）。拍照完成后，偏移到 JPEG 数据起始标识 (0xFF,0xD8)，然后保存到 SD 卡。注意，这里得先切换为 SD 卡模式，才能进行数据保存，拍照完再切回 OV2640 模式。

`bmp` 拍照则比较简单，直接在 main 函数里面调用 `bmp_encode` 函数实现，这里我们就不再细说了，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。

47.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 探索者 STM32F4 开发板上，得到如图 47.4.1 所示界面：

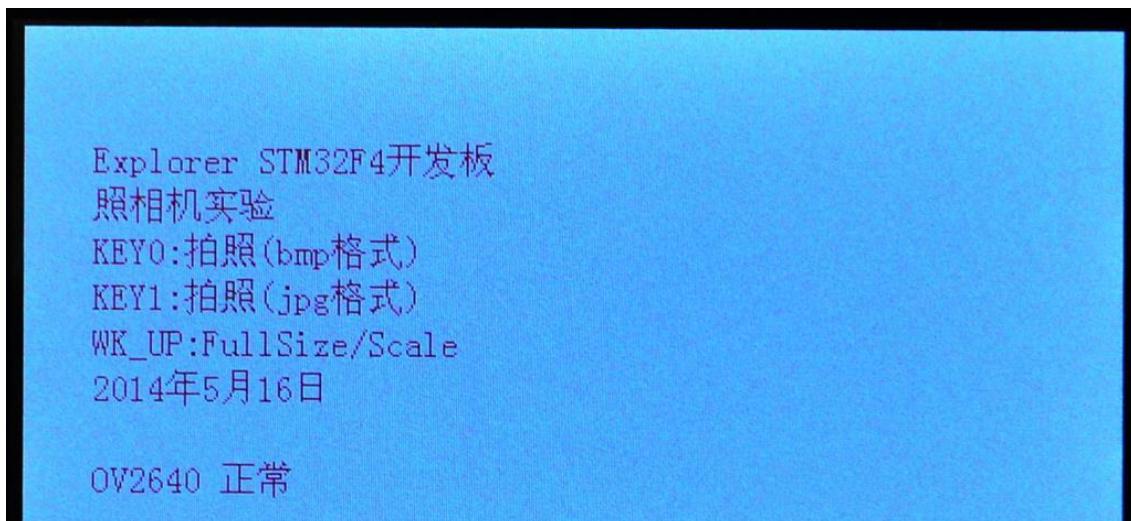


图 47.4.1 程序运行效果图

随后，进入监控界面。此时，我们可以按下 KEY0 和 KEY1，即可进行 bmp/jpg 拍照。拍照得到的照片效果如图 47.4.2 和图 47.4.3 所示：



图 47.4.2 拍照样图 (bmp 拍照样图)



图 47.4.3 拍照片样图 (jpg 拍照片样图)

最后，我们还可以通过 USMART 调用 `bmp_encode` 函数，实现串口控制 `bmp` 拍照，还可以拍成各种尺寸哦（不过必须 \leqslant LCD 分辨率）！

第四十八章 音乐播放器实验

ALIENTEK 探索者 STM32F4 开发板拥有全双工 I2S，且外扩了一颗 HIFI 级 CODEC 芯片：WM8978G，支持最高 192K 24BIT 的音频播放，并且支持录音（下一章介绍）。本章，我们将利用探索者 STM32F4 开发板实现一个简单的音乐播放器（仅支持 WAV 播放）。本章分为如下几个部：

- 48.1 WAV&WM8978&I2S 简介
- 48.2 硬件设计
- 48.3 软件设计
- 48.4 下载验证

48.1 WAV&WM8978&I2S 简介

本章新知识点比较多，包括：WAV、WM8978 和 I2S 等三个知识点。下面我们将分别向大家介绍。

48.1.1 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！

WAV 一般采用线性 PCM（脉冲编码调制）编码，本章，我们也主要讨论 PCM 的播放，因为这个最简单。

WAV 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 48.1.1.1 所示：



图 48.1.1.1 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。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;

```

ChunkSize 后紧接着就是 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 48.1.1.1 所示的几种形式：

单声道	取样 1	取样 2	取样 3	取样 4	取样 5	取样 6
-----	------	------	------	------	------	------

8 位量化	声道 0					
双声道	取样 1		取样 2		取样 3	
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		取样 3
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)	声道 0 (低字节)	声道 0 (高字节)
双声道	取样 1				取样 2	
16 位量化	声道 0 (低字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (高字节)	声道 0 (低字节)	声道 0 (高字节)
单声道	取样 1			取样 2		
24 位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)
双声道	取样 1					
24 位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (中字节)	声道 1 (高字节)

表 48.1.1.1 WAVE 文件数据采样格式

本章，我们播放的音频支持：16 位和 24 位，立体声，所以每个取样为 4/6 个字节，低字节在前，高字节在后。在得到这些 wav 数据以后，通过 I2S 丢给 WM8978，就可以欣赏音乐了。

48.1.2 WM8978 简介

WM8978 是欧胜 (Wolfson) 推出的一款全功能音频处理器。它带有一个 HI-FI 级数字信号处理内核，支持增强 3D 硬件环绕音效，以及 5 频段的硬件均衡器，可以有效改善音质；并有一个可编程的陷波滤波器，用以去除屏幕开、切换等噪音。

WM8978 同样集成了对麦克风的支持，以及用于一个强悍的扬声器功放，可提供高达 900mW 的高质量音响效果扬声器功率。

一个数字回放限制器可防止扬声器声音过载。WM8978 进一步提升了耳机放大器输出功率，在推动 16 欧姆耳机的时候，每声道最大输出功率高达 40 毫瓦！可以连接市面上绝大多数适合随身听的高端 HI-FI 耳机。

WM8988 的主要特性有：

- I2S 接口，支持最高 192K, 24bit 音频播放
- DAC 信噪比 98dB; ADC 信噪比 90dB
- 支持无电容耳机驱动（提供 40mW@16 Ω 的输出能力）
- 支持扬声器输出（提供 0.9W@8 Ω 的驱动能力）
- 支持立体声差分输入/麦克风输入
- 支持左右声道音量独立调节
- 支持 3D 效果，支持 5 路 EQ 调节

WM8978 的控制通过 I2S 接口（即数字音频接口）同 MCU 进行音频数据传输（支持音频接收和发送），通过两线（MODE=0，即 IIC 接口）或三线（MODE=1）接口进行配置。WM8978 的 I2S 接口，由 4 个引脚组成：

- 1, ADCDAT:ADC 数据输出
- 2, DACDAT: DAC 数据输入
- 3, LRC: 数据左/右对齐时钟

4, BCLK: 位时钟, 用于同步

WM8978 可作为 I2S 主机, 输出 LRC 和 BLCK 时钟, 不过我们一般使用 WM8978 作为从机, 接收 LRC 和 BLCK。另外, WM8978 的 I2S 接口支持 5 种不同的音频数据模式: 左 (MSB) 对齐标准、右 (LSB) 对齐标准、飞利浦 (I2S) 标准、DSP 模式 A 和 DSP 模式 B。本章, 我们用飞利浦标准来传输 I2S 数据。

飞利浦 (I2S) 标准模式, 数据在跟随 LRC 传输的 BCLK 的第二个上升沿时传输 MSB, 其他位一直到 LSB 按顺序传输。传输依赖于字长、BCLK 频率和采样率, 在每个采样的 LSB 和下一个采样的 MSB 之间都应该有未用的 BCLK 周期。飞利浦标准模式的 I2S 数据传输协议如图 48.1.2.1 所示:

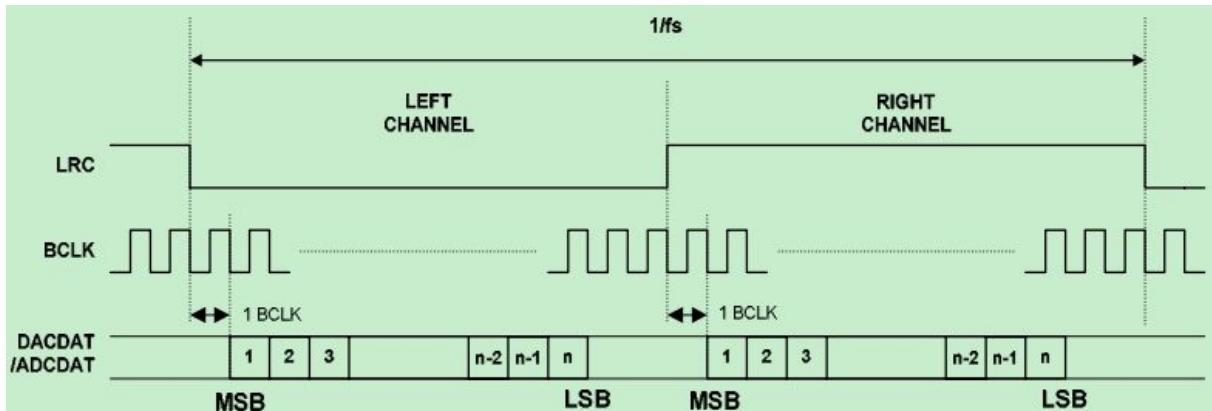


图 48.1.2.1 飞利浦标准模式 I2S 数据传输图

图中, f_s 即音频信号的采样率, 比如 44.1Khz, 因此可以知道, LRC 的频率就是音频信号的采样率。另外, WM8978 还需要一个 MCLK, 本章我们采用 STM32F4 为其提供 MCLK 时钟, MCLK 的频率必须等于 $256f_s$, 也就是音频采样率的 256 倍。

WM8978 的框图如图 48.1.2.2 所示:

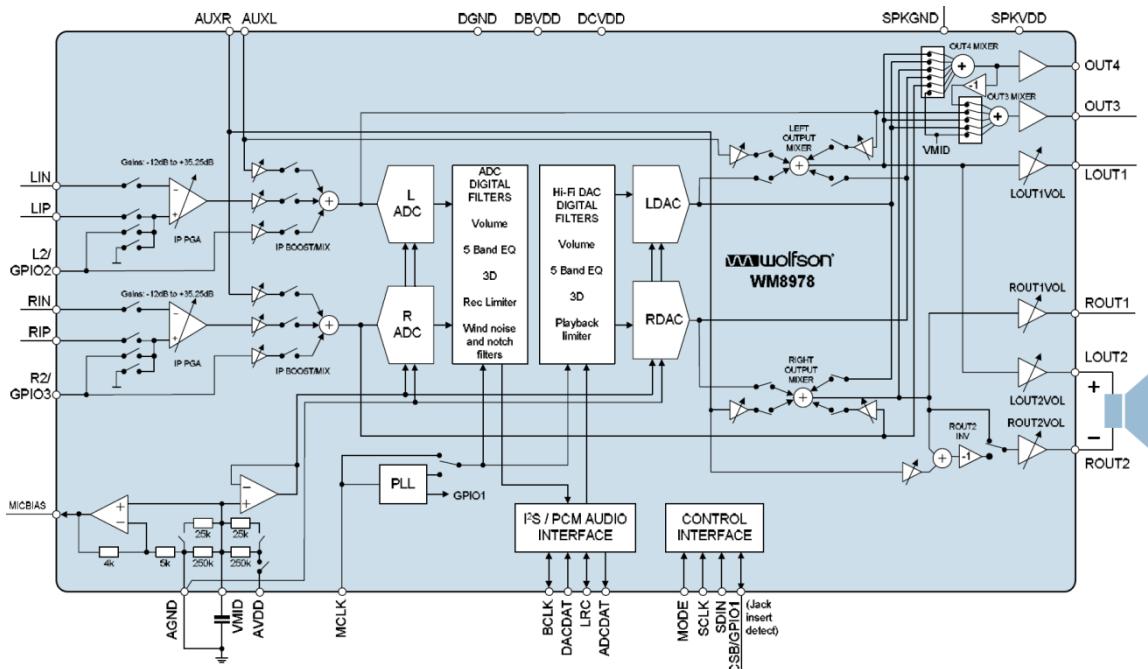


图 48.1.2.2 WM8978 框图

从上图可以看出, WM8978 内部有很多的模拟开关, 用来选择通道, 同时还有很多调节器,

用来设置增益和音量。

本章，我们通过 IIC 接口 (MODE=0) 连接 WM8978，不过 WM8978 的 IIC 接口比较特殊：1，只支持写，不支持读数据；2，寄存器长度为 7 位，数据长度为 9 位。3，寄存器字节的最低位用于传输数据的最高位（也就是 9 位数据的最高位，7 位寄存器的最低位）。WM8978 的 IIC 地址固定为：0X1A。关于 WM8978 的 IIC 详细介绍，请看其数据手册第 77 页。

这里我们简单介绍一下要正常使用 WM8978 来播放音乐，应该执行哪些配置。

1，寄存器 R0 (00h)，该寄存器用于控制 WM8978 的软复位，写任意值到该寄存器地址，即可实现软复位 WM8978。

2，寄存器 R1 (01h)，该寄存器主要要设置 BIASEN (bit3)，该位设置为 1，模拟部分的放大器才会工作，才可以听到声音。

3，寄存器 R2 (02h)，该寄存器要设置 ROUT1EN(bit8), LOUT1EN(bit7) 和 SLEEP(bit6) 等三个位，ROUT1EN 和 LOUT1EN，设置为 1，使能耳机输出，SLEEP 设置为 0，进入正常工作模式。

4，寄存器 R3 (03h)，该寄存器要设置 LOUT2EN(bit6), ROUT2EN(bit5), RMIXER(bit3), LMIXER(bit2), DACENR(bit1) 和 DACENL(bit0) 等 6 个位。LOUT2EN 和 ROUT2EN，设置为 1，使能喇叭输出；LMIXER 和 RMIXER 设置为 1，使能左右声道混合器；DACENL 和 DACENR 则是使能左右声道的 DAC 了，必须设置为 1。

5，寄存器 R4 (04h)，该寄存器要设置 WL(bit6:5) 和 FMT(bit4:3) 等 4 个位。WL(bit6:5) 用于设置字长(即设置音频数据有效位数)，00 表示 16 位音频，10 表示 24 位音频；FMT(bit4:3) 用于设置 I2S 音频数据格式(模式)，我们一般设置为 10，表示 I2S 格式，即飞利浦模式。

6，寄存器 R6 (06h)，该寄存器我们直接全部设置为 0 即可，设置 MCLK 和 BCLK 都来自外部，即由 STM32F4 提供。

7，寄存器 R10 (0Ah)，该寄存器我们要设置 SOFTMUTE(bit6) 和 DACOSR128(bit3) 等两个位，SOFTMUTE 设置为 0，关闭软件静音；DACOSR128 设置为 1，DAC 得到最好的 SNR。

8，寄存器 R43 (2Bh)，该寄存器我们只需要设置 INVROUT2 为 1 即可，反转 ROUT2 输出，更好的驱动喇叭。

9，寄存器 R49 (31h)，该寄存器我们要设置 SPKBOOST(bit2) 和 TSDEN(bit1) 这两个位。SPKBOOST 用于设置喇叭的增益，我们默认设置为 0 就好了 (gain=-1)，如想获得更大的声音，设置为 1 (gain=+1.5) 即可；TSDEN 用于设置过热保护，设置为 1 (开启) 即可。

10，寄存器 R50 (32h) 和 R51 (33h)，这两个寄存器设置类似，一个用于设置左声道 (R50)，另外一个用于设置右声道 (R51)。我们只需要设置这两个寄存器的最低位为 1 即可，将左右声道的 DAC 输出接入左右声道混合器里面，才能在耳机/喇叭听到音乐。

11，寄存器 R52 (34h) 和 R53 (35h)，这两个寄存器用于设置耳机音量，同样一个用于设置左声道 (R52)，另外一个用于设置右声道 (R53)。这两个寄存器的最高位 (HPVU) 用于设置是否更新左右声道的音量，最低 6 位用于设置左右声道的音量，我们可以先设置好两个寄存器的音量值，最后设置其中一个寄存器最高位为 1，即可更新音量设置。

12，寄存器 R54 (36h) 和 R55 (37h)，这两个寄存器用于设置喇叭音量，同 R52, R53 设置一模一样，这里就不细说了。

以上，就是我们用 WM8978 播放音乐时的设置，按照以上所述，对各个寄存器进行相应的配置，即可使用 WM8978 正常播放音乐了。还有其他一些 3D 设置，EQ 设置等，我们这里就不再介绍了，大家参考 WM8978 的数据手册自行研究下即可。

48.1.3 I2S 简介

I2S(Inter IC Sound)总线，又称集成电路内置音频总线，是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专责于音频设备之间的数据传输，广泛应用于各种多媒体系统。它采用了沿独立的导线传输时钟与数据信号的设计，通过将数据和时钟信号分离，避免了因时差诱发的失真，为用户节省了购买抵抗音频抖动的专业设备的费用。

STM32F4 自带了 2 个全双工 I2S 接口，其特点包括：

- 支持全双工/半双工通信
- 主持主/从模式设置
- 8 位可编程线性预分频器，可实现精确的音频采样频率(8~192Khz)
- 支持 16 位/24 位/32 位数据格式
- 数据包帧固定为 16 位（仅 16 位数据帧）或 32 位（可容纳 16/24/32 位数据帧）
- 可编程时钟极性
- 支持 MSB 对齐（左对齐）、LSB 对齐（右对齐）、飞利浦标准和 PCM 标准等 I2S 协议
- 支持 DMA 数据传输（16 位宽）
- 数据方向固定位 MSB 在前
- 支持主时钟输出（固定为 $256 \times f_s$, f_s 即音频采样率）

STM32F4 的 I2S 框图如图 48.1.3.1 所示：

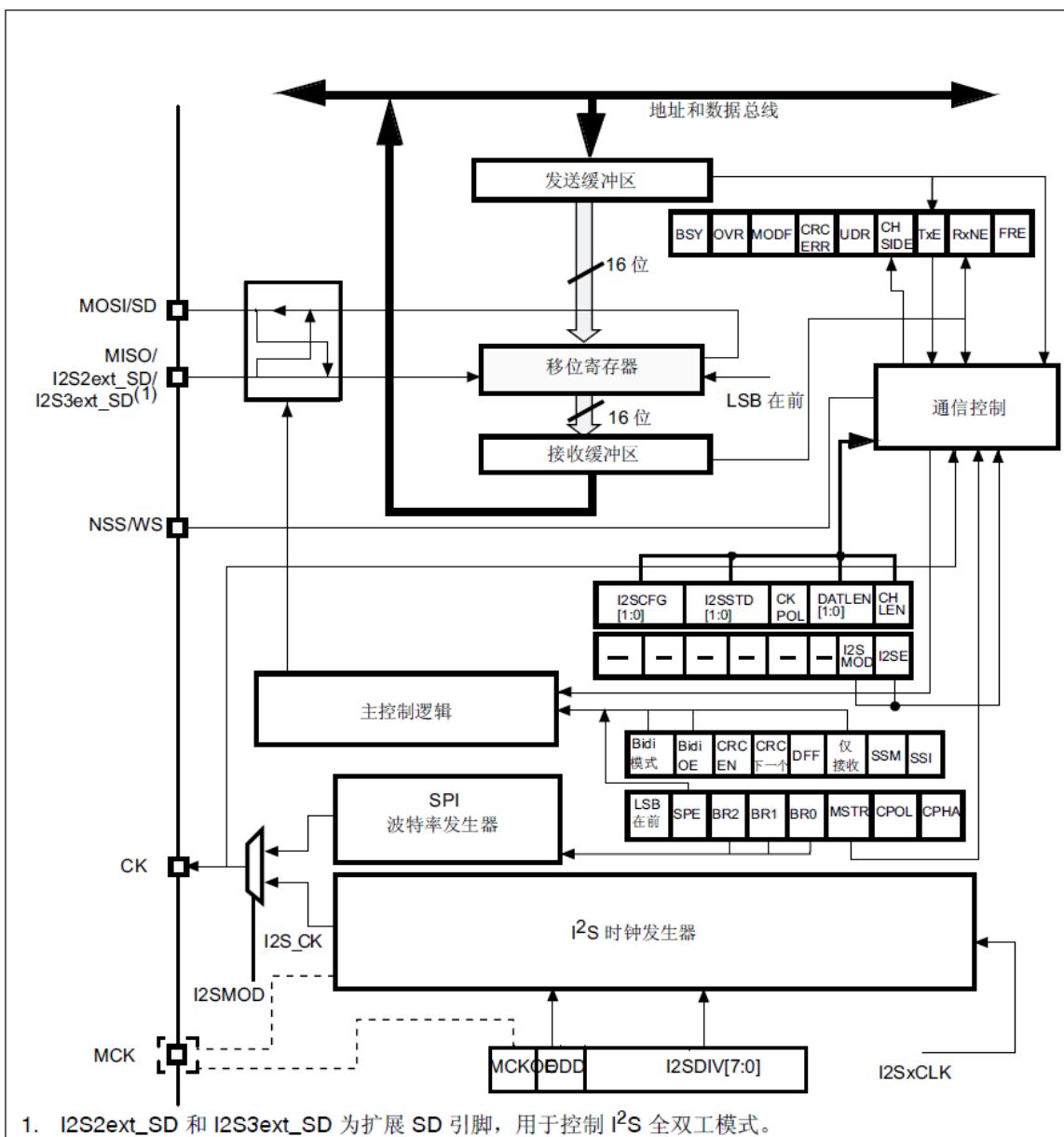


图 48.1.3.1 I2S 框图

STM32F4 的 I2S 是与 SPI 部分共用的, 通过设置 SPI_I2SCFGR 寄存器的 I2SMOD 位即可开启 I2S 功能, I2S 接口使用了几乎与 SPI 相同的引脚、标志和中断。

I2S 用到的信号有:

1, SD: 串行数据 (映射到 MOSI 引脚), 用于发送或接收两个时分复用的数据通道上的数据 (仅半双工模式)。

2, WS: 字选择 (映射到 NSS 引脚), 即帧时钟, 用于切换左右声道的数据。WS 频率等于音频信号采样率 (fs)。

3, CK: 串行时钟 (映射到 SCK 引脚), 即位时钟, 是主模式下的串行时钟输出以及从模式下的串行时钟输入。CK 频率=WS 频率 (fs) *2*16 (16 位宽), 如果是 32 位宽, 则是: CK 频率=WS 频率 (fs) *2*32 (32 位宽)

4, I2S2ext_SD 和 I2S3ext_SD: 用于控制 I2S 全双工模式的附加引脚(映射到 MISO 引脚)。

5, MCK: 即主时钟输出, 当 I2S 配置为主模式 (并且 SPI_I2SPR 寄存器中的 MCKOE 位

置 1) 时, 使用此时钟, 该时钟输出频率 $256 \times fs$, fs 即音频信号采样频率 (fs)。

为支持 I2S 全双工模式, 除了 I2S2 和 I2S3, 还可以使用两个额外的 I2S, 它们称为扩展 I2S (I2S2_ext, I2S3_ext), 如图 48.1.3.2:

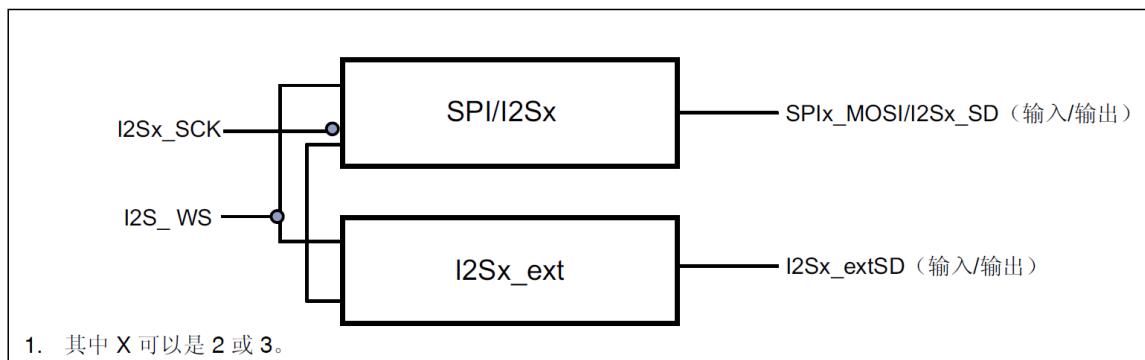


图 48.1.3.2 I2S 全双工框图

因此, 第一个 I2S 全双工接口基于 I2S2 和 I2S2_ext, 第二个基于 I2S3 和 I2S3_ext。注意: I2S2_ext 和 I2S3_ext 仅用于全双工模式。

I2Sx 可以在主模式下工作。因此:

- 1, 只有 I2Sx 可在半双工模式下输出 SCK 和 WS
- 2, 只有 I2Sx 可在全双工模式下向 I2S2_ext 和 I2S3_ext 提供 SCK 和 WS。

扩展 I2S (I2Sx_ext)只能用于全双工模式。I2Sx_ext 始终在从模式下工作。I2Sx 和 I2Sx_ext 均可用于发送和接收。

STM32F4 的 I2S 支持 4 种数据和帧格式组合, 分别是: 1, 将 16 位数据封装在 16 位帧中; 2, 将 16 位数据封装在 32 位帧中; 3, 将 24 位数据封装在 32 位帧中; 4, 将 32 位数据封装在 32 位帧中。

将 16 位数据封装在 32 位帧中时, 前 16 位(MSB)为有效位, 16 位 LSB 被强制清零, 无需任何软件操作或 DMA 请求 (只需一个读/写操作)。如果应用程序首选 DMA, 则 24 位和 32 位数据帧需要对 SPI_DR 执行两次 CPU 读取或写入操作, 或者需要两次 DMA 操作。24 位的数据帧, 硬件会将 8 位非有效位扩展到带有 0 位的 32 位。

对于所有数据格式和通信标准而言, 始终会先发送最高有效位 (MSB 优先)。

STM32F4 的 I2S 支持: MSB 对齐 (左对齐) 标准、LSB 对齐 (右对齐) 标准、飞利浦标准和 PCM 标准等 4 种音频标准, 本章我们用飞利浦标准, 仅针对该标准进行介绍, 其他的请大家参考《STM32F4xx 中文参考手册》第 27.4 节。

I2S 飞利浦标准, 使用 WS 信号来指示当前正在发送的数据所属的通道。该信号从当前通道数据的第一个位(MSB)之前的一个时钟开始有效。发送方在时钟信号(CK)的下降沿改变数据, 接收方在上升沿读取数据。WS 信号也在 CK 的下降沿变化。这和我们 48.1.2 节介绍的是一样的。

本章我们使用 16 位/24 位数据格式, 16 位时采用扩展帧格式 (即将 16 位数据封装在 32 位帧中), 以 24 位帧为例, I2S 波形 (飞利浦标准) 如图 48.1.3.3 所示:

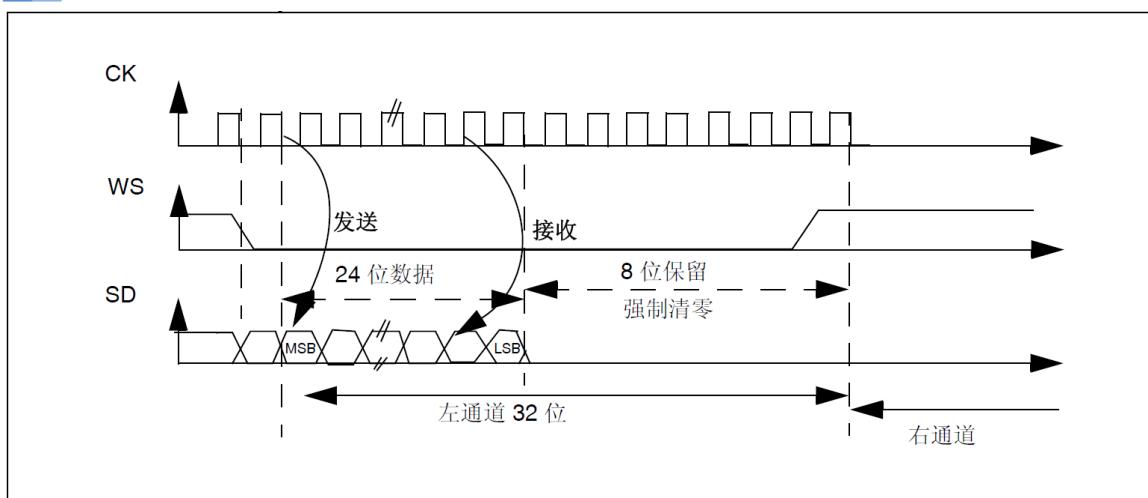


图 48.1.3.3 I2S 飞利浦标准 24 位帧格式波形

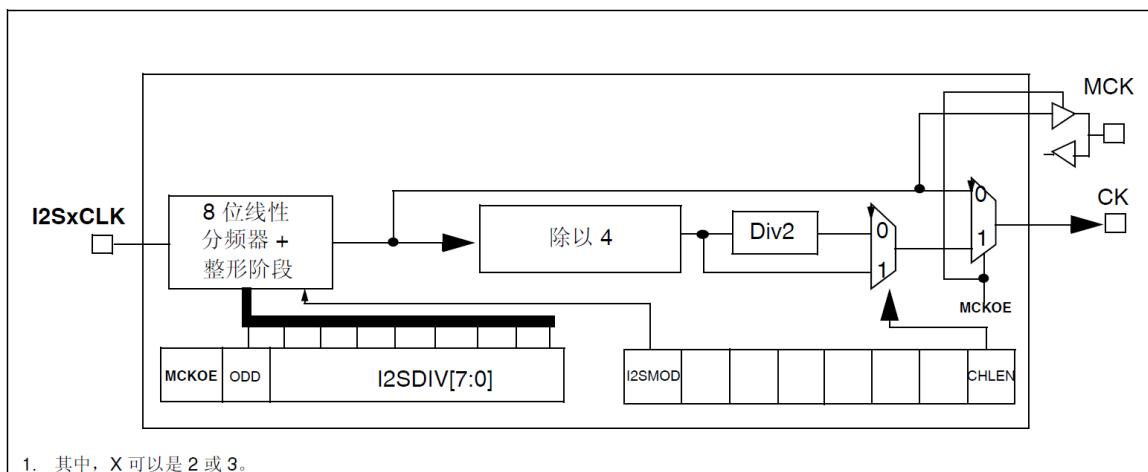
这个图和图 48.1.2.1 是一样的时序，在 24 位模式下数据传输，需要对 SPI_DR 执行两次读取或写入操作。比如我们要发送 0X8EAA33 这个数据，就要分两次写入 SPI_DR，第一次写入：0X8EAA，第二次写入 0X33xx（xx 可以为任意数值），这样就把 0X8EAA33 发送出去了。

顺便说一下 SD 卡读取到的 24 位 WAV 数据流，是低字节在前，高字节在后的，比如，我们读到一个声道的数据（24bit），存储在 buf[3] 里面，那么要通过 SPI_DR 发送这个 24 位数据，过程如下：

```
SPI_DR=((u16)buf[2]<<8)+buf[1];
SPI_DR=(u16)buf[0]<<8;
```

这样，第一次发送高 16 位数据，第二次发送低 8 位数据，完成一次 24bit 数据的发送。

接下来，我们介绍下 STM32F4 的 I2S 时钟发生器，其架构如图 48.1.3.4 所示：



1. 其中，X 可以是 2 或 3。

图 48.1.3.4 I2S 时钟发生器架构

图中 I2SxCLK 可以来自 PLLI2S 输出（通过 R 系数分频）或者来自外部时钟（I2S_CKIN 引脚），一般我们使用前者作为 I2SxCLK 输入时钟。

一般我们需要根据音频采样率（fs，即 CK 的频率）来计算各个分频器的值，常用的音频采样率有：22.05Khz、44.1Khz、48Khz、96Khz、196Khz 等。

根据是否使能 MCK 输出，fs 频率的计算公式有 2 种情况。不过，本章只考虑 MCK 输出使能时的情况，当 MCK 输出使能时，fs 频率计算公式如下：

$$fs=I2SxCLK/[256*(2*I2SDIV+ODD)]$$

其中：I2SxCLK=(HSE/pllm)*PLLI2SN/PLLI2SR。HSE 我们是 8Mhz，而 pllm 在系统时钟初始化就确定了，是 8，这样结合以上 2 式，可得计算公式如下：

$$fs = (1000 * PLLI2SN / PLLI2SR) / [256 * (2 * I2SDIV + ODD)]$$

fs 单位是：Khz。其中：PLLI2SN 取值范围：192~432；PLLI2SR 取值范围：2~7；I2SDIV 取值范围：2~255；ODD 取值范围：0/1。根据以上约束条件，我们便可以根据 fs 来设置各个系数的值了，不过很多时候，并不能取得和 fs 一模一样的频率，只能近似等于 fs，比如 44.1Khz 采样率，我们设置 PLLI2SN=271，PLLI2SR=2，I2SDIV=6，ODD=0，得到 fs=44.108073Khz，误差为：0.0183%。晶振频率决定了有时无法通过分频得到我们所要的 fs，所以，某些 fs 如果要实现 0 误差，大家必须得选用外部时钟才可以。

如果要通过程序去计算这些系数的值，是比较麻烦的，所以，我们事先计算好常用 fs 对应的系数值，建立一个表，这样，用的时候，只需要查表取值就可以了，大大简化了代码，常用 fs 对应系数表如下：

```
//表格式:采样率/10,PLLI2SN,PLLI2SR,I2SDIV,ODD
const u16 I2S_PSC_TBL[][5]=
{
    {800,256,5,12,1},      //8Khz 采样率
    {1102,429,4,19,0},     //11.025Khz 采样率
    {1600,213,2,13,0},     //16Khz 采样率
    {2205,429,4,9,1},      //22.05Khz 采样率
    {3200,213,2,6,1},      //32Khz 采样率
    {4410,271,2,6,0},      //44.1Khz 采样率
    {4800,258,3,3,1},      //48Khz 采样率
    {8820,316,2,3,1},      //88.2Khz 采样率
    {9600,344,2,3,1},      //96Khz 采样率
    {17640,361,2,2,0},     //176.4Khz 采样率
    {19200,393,2,2,0},     //192Khz 采样率
};
```

有了上面的 fs-系数对应表，我们可以很方便的完成 I2S 的时钟配置。

接下来，我们看看本章需要用到的一些相关寄存器。

首先，是 SPI_I2S 配置寄存器：SPI_I2SCFGR，该寄存器各位描述如图 48.1.3.5 所示：

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				I2SMOD	I2SE	I2SCFG		PCMSYNC	Reserved	I2SSTD		CKPOL	DATLEN		CHLEN	
				rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	

图 48.1.3.5 寄存器 SPI_I2SCFGR 各位描述

I2SMOD 位，设置为 1，选择 I2S 模式，注意，必须在 I2S/SPI 禁止的时候，设置该位。

I2SE 位，设置为 1，使能 I2S 外设，该位必须在 I2SMOD 位设置之后再设置。

I2SCFG[1:0]位，这两个位用于配置 I2S 模式，设置为 10，选择主模式（发送）。

I2SSTD[1:0]位，这两个位用于选择 I2S 标准，设置为 00，选择飞利浦模式。

CKPOL 位，用于设置空闲时时钟电平，设置为 0，空闲时时钟低电平。

DATLEN[1:0]位，用于设置数据长度，00，表示 16 位数据；01 表示 24 位数据。

CHLEN 位，用于设置通道长度，即帧长度，0，表示 16 位；1，表示 32 位。

第二个是 SPI_I2S 预分配器寄存器：SPI_I2SPR，该寄存器各位描述如图 48.1.3.6 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved					MCKOE	ODD	I2SDIV									
					rw	rw										

位 15:10 保留，必须保持复位值。

位 9 **MCKOE**: 主时钟输出使能 (Master clock output enable)

0: 禁止主时钟输出

1: 使能主时钟输出

注意: 应在 \mathcal{P}_S 禁止时配置此位。只有在 \mathcal{P}_S 为主模式时，才会使用此位。不适用于 SPI 模式。

位 8 **ODD**: 预分频器的奇数因子 (Odd factor for the prescaler)

0: 实际分频值为 = I2SDIV * 2

1: 实际分频值为 = (I2SDIV * 2)+1

注意: 应在 \mathcal{P}_S 禁止时配置此位。只有在 \mathcal{P}_S 为主模式时，才会使用此位。

位 7:0 **I2SDIV**: I2S 线性预分频器 (I2S Linear prescaler)

I2SDIV [7:0] = 0 或 I2SDIV [7:0] = 1 为禁用值。

注意: 应在 \mathcal{P}_S 禁止时配置这些位。只有在 \mathcal{P}_S 为主模式时，才会使用此位。

图 48.1.3.6 寄存器 SPI_I2SPR 各位描述

本章我们设置 MCKOE 为 1，开启 MCK 输出，ODD 和 I2SDIV 则根据不同的 fs，查表进行设置。

第三个是 PLLI2S 配置寄存器: RCC_PLLI2SCFGR，该寄存器各位描述如图 48.1.3.7 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserv ed	PLLI2S R2	PLLI2S R1	PLLI2S R0	Reserved											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Reserv ed	PLLI2SN 8	PLLI2SN 7	PLLI2SN 6	PLLI2SN 5	PLLI2SN 4	PLLI2SN 3	PLLI2SN 2	PLLI2SN 1	PLLI2SN 0	Reserved					
	rw														

图 48.1.3.7 寄存器 RCC_PLLI2SCFGR 各位描述

该寄存器用于配置 PLLI2SR 和 PLLI2SN 两个系数，PLLI2SR 的取值范围是: 2~7, PLLI2SN 的取值范围是: 192~432。同样，这两个也是根据 fs 的值来设置的。

此外，还要用到 SPI_CR2 寄存器的 bit1 位，设置 I2S TX DMA 数据传输，SPI_DR 寄存器用于传输数据，本章用 DMA 来传输，所以直接设置 DMA 的外设地址位 SPI_DR 即可。

最后，我们看看要通过 STM32F4 的 I2S，驱动 WM8978 播放音乐的简要步骤。这里需要说明一下，I2S 相关的库函数申明和定义跟 SPI 是同文件的，在 stm32f4xx_spi.c 以及头文件 stm32f4xx_spi.h 中。具体步骤如下：

1) 初始化 WM8978

这个过程就是在 48.1.2 节最后那十几个寄存器的配置，包括软复位、DAC 设置、输出设置和音量设置等。在我们实验工程中是在文件 wm8978.c 中，大家可以打开实验工程参考。

2) 初始化 I2S

此过程主要设置 SPI_I2SCFGR 寄存器，设置 I2S 模式、I2S 标准、时钟空闲电平和数据帧长等，最后开启 I2S TX DMA，使能 I2S 外设。

在库函数中初始化 I2S 调用的函数为：

```
void I2S_Init(SPI_TypeDef* SPIx, I2S_InitTypeDef* I2S_InitStruct);
```

第一个参数比较好理解，我们来着重看下第二个参数，这里我们主要讲解结构体 I2S_InitTypeDef 各个成员变量的含义。结构体 I2S_InitTypeDef 的定义为：

```
typedef struct
```

```
{
    uint16_t I2S_Mode;
    uint16_t I2S_Standard;
    uint16_t I2S_DataFormat;
    uint16_t I2S_MCLKOutput;
    uint32_t I2S_AudioFreq;
    uint16_t I2S_CPOL;
}I2S_InitTypeDef;
```

第一个参数用来设置 I2S 的模式，也就是设置 SPI_I2SCFGR 寄存器的 I2SCFG 相关位。可以配置为主模式发送 I2S_Mode_MasterTx，主模式接受 I2S_Mode_MasterRx，从模式发送 I2S_Mode_SlaveTx 以及从模式接受 I2S_Mode_SlaveRx 四种模式。

第二个参数 I2S_Standard 用来设置 I2S 标准，这个前面已经讲解过。可以设置为：飞利浦标准 I2S_Standard_Philips, MSB 对齐标准 I2S_Standard_MSB, LSB 对齐标准 I2S_Standard_LSB 以及 PCM 标准 I2S_Standard_PCMShort。

第三个参数 I2S_DataFormat 用来设置 I2S 的数据通信格式。这里实际包含设置 SPI_I2SCFGR 寄存器的 HCLEN 位（通道长度）以及 DATLEN 位（传输的数据长度）。当我们设置为 16 位标准格式 I2S_DataFormat_16b 的时候，实际上传输的数据长度为 16 位，通道长度为 16 位。当我们设置为其他值的时候，通道长度都为 32 位。

第四个参数 I2S_MCLKOutput 用来设置是否使能主时钟输出。我们实验会使能主时钟输出。

第五个参数 I2S_AudioFreq 用来设置 I2S 频率。实际根据输入的频率值，会来计算 SPI 预分频寄存器 SPI_I2SPR 的预分频奇数因子以及 I2S 线性预分频器的值。这里支持 10 中频率：

#define I2S_AudioFreq_192k	((uint32_t)192000)
#define I2S_AudioFreq_96k	((uint32_t)96000)
#define I2S_AudioFreq_48k	((uint32_t)48000)
#define I2S_AudioFreq_44k	((uint32_t)44100)
#define I2S_AudioFreq_32k	((uint32_t)32000)
#define I2S_AudioFreq_22k	((uint32_t)22050)
#define I2S_AudioFreq_16k	((uint32_t)16000)
#define I2S_AudioFreq_11k	((uint32_t)11025)
#define I2S_AudioFreq_8k	((uint32_t)8000)
#define I2S_AudioFreq_Default	((uint32_t)2)

第六个参数 I2S_CPOL 用来设置空闲状态时钟电平，这个比较好理解。取值为高电平 I2S_CPOL_High 以及低电平 I2S_CPOL_Low。

3) 解析 WAV 文件，获取音频信号采样率和位数并设置 I2S 时钟分频器

这里，要先解析 WAV 文件，取得音频信号的采样率 (fs) 和位数 (16 位或 32 位)，根据这两个参数，来设置 I2S 的时钟分频，这里我们用前面介绍的查表法来设置即可。这是我们单独写了一个设置频率的函数为 I2S2_SampleRate_Set，我们后面程序章节会讲解。

4) 设置 DMA

I2S 播放音频的时候，一般都是通过 DMA 来传输数据的，所以必须配置 DMA，本章我们用 I2S2，其 TX 是使用的 DMA1 数据流 4 的通道 0 来传输的。并且，STM32F4 的 DMA 具有双缓冲机制，这样可以提高效率，大大方便了我们的数据传输，本章将 DMA1 数据流 4 设置为：双缓冲循环模式，外设和存储器都是 16 位宽，并开启 DMA 传输完成中断（方便填充数据）。DMA 具体配置过程请参考我们光盘工程代码，前面 DMA 实验我们已经讲解过 DMA 相关配置

过程。

5) 编写 DMA 传输完成中断服务函数

为了方便填充音频数据，我们使用 DMA 传输完成中断，每当一个缓冲数据发送完后，硬件自动切换为下一个缓冲，同时进入中断服务函数，填充数据到发送完的这个缓冲。过程如图 48.1.3.8 所示：

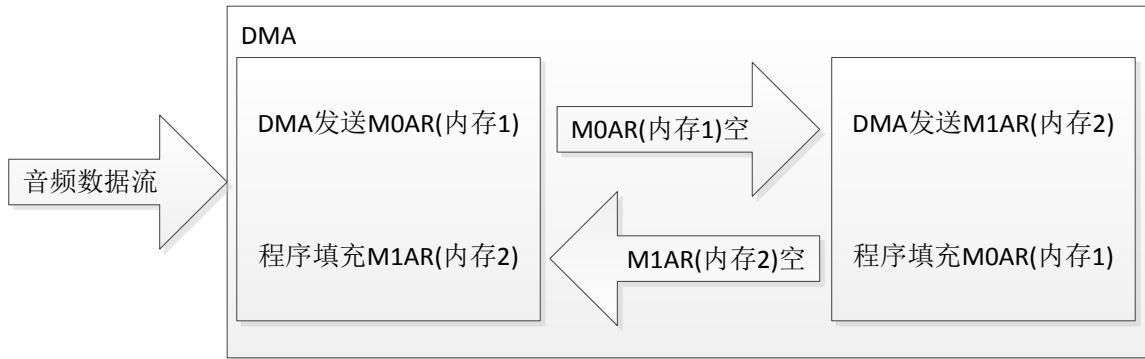


图 48.1.3.8 DMA 双缓冲发送音频数据流框图

6) 开启 DMA 传输，填充数据

最后，我们就只需要开启 DMA 传输，然后及时填充 WAV 数据到 DMA 的两个缓存区即可。此时，就可以在 WM8978 的耳机和喇叭通道听到所播放音乐了。操作方法为：

```
DMA_Cmd(DMA1_Stream4,ENABLE); //开启 DMA TX 传输,开始播放
```

48.2 硬件设计

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，则开始循环播放 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（仅支持 wav 格式）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，KEY_UP 用来控制暂停/继续播放。DS0 还是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) 三个按键 (KEY_UP/KEY0/KEY1)
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡
- 6) SPI FLASH
- 7) WM8978
- 8) I2S2

这些硬件我们都已经介绍过了，不过 WM8978 和 STM32F4 的连接，还没有介绍，连接如图 48.2.1 所示：

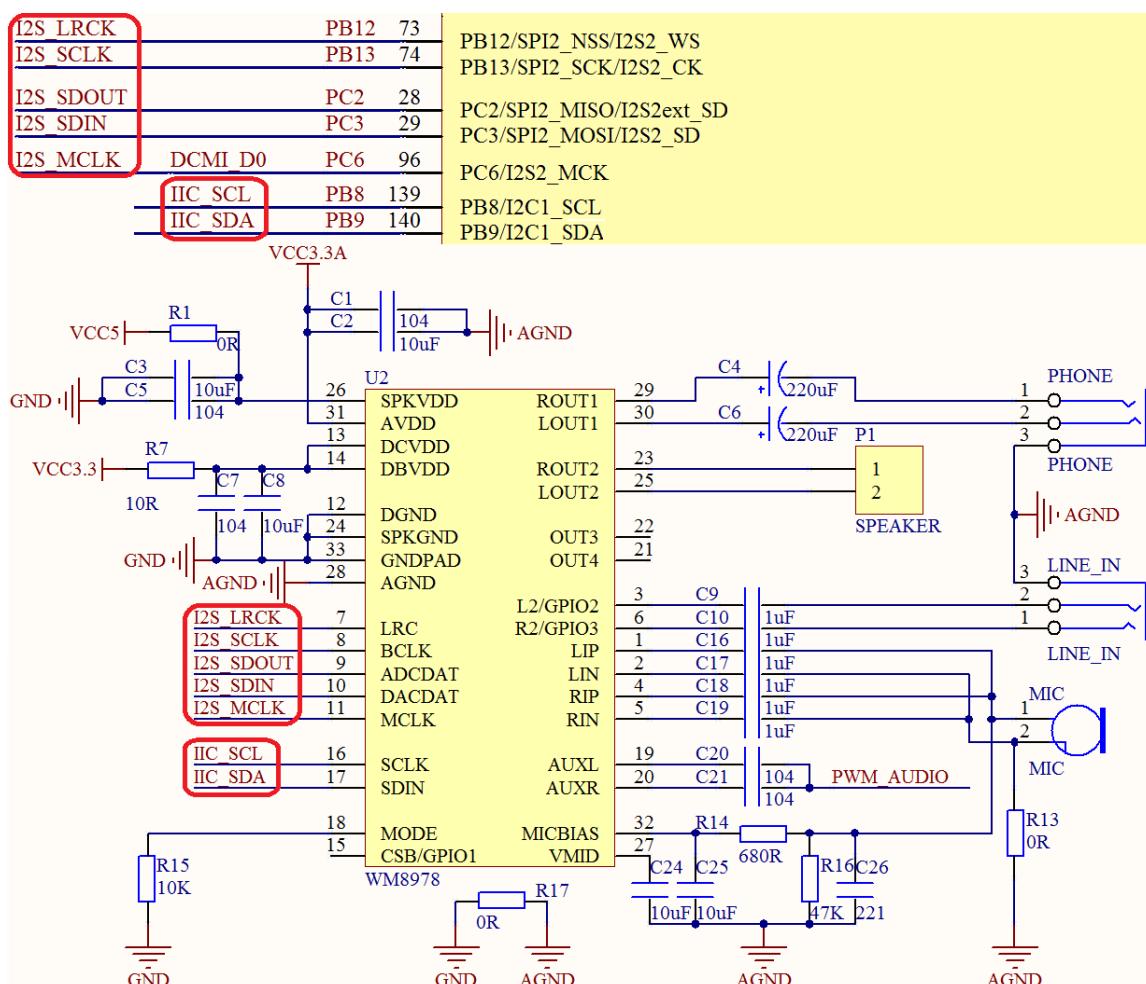


图 48.2.1 WM8978 与 STM32F4 连接原理图

图中，PHONE 接口，可以用来插耳机，P1 接口，可以外接喇叭（1W@8Ω，需自备）。硬件上，IIC 接口和 24C02，MPU6050 等共用，另外 I2S_MCLK 和 DCMI_D0 共用，所以 I2S 和 DCMI 不可以同时使用。

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 wav 歌曲在 MUSIC 文件夹下）和一个耳机（或喇叭），分别插入 SD 卡接口和耳机接口（喇叭接 P1 接口），然后下载本实验就可以通过耳机来听歌了。

48.3 软件设计

打开本章实验工程目录可以看到，我们在工程根目录文件夹下新建 APP 和 AUDIOCODEC 两个文件夹。在 APP 文件夹里面新建了 audioplay.c 和 audioplay.h 两个文件。在 AUDIOCODEC 文件夹里面新建了 wav 文件夹，然后在其中新建了 wavplay.c 和 wavplay.h 两个文件。同时，我们把相关的源文件引入工程相应分组，同时将 APP 和 wav 文件夹加入头文件包含路径。

然后，我们在 HARDWARE 文件夹下新建了 WM8978 和 I2S 两个文件夹，在 WM8978 文件夹里面新建了 wm8978.c 和 wm8978.h 两个文件，在 I2S 文件夹里面新建了 i2s.c 和 i2s.h 两个文件。最后将 wm8978.c 和 i2s.c 添加到工程 HARDWARE 组下。同时相应的头文件加入到 PATH 中。

本章代码比较多，我们就不全部贴出来给大家介绍了，这里仅挑一些重点函数给大家介绍下。首先是 i2s.c 里面，重点函数代码如下：

```
//I2S2 初始化
//参数 I2S_Standard: @ref SPI_I2S_Standard I2S 标准,
//参数 I2S_Mode: @ref SPI_I2S_Mode
//参数 I2S_Clock_Polarity      @ref SPI_I2S_Clock_Polarity:
//参数 I2S_DataFormat: @ref SPI_I2S_Data_Format :
void I2S2_Init(u16 I2S_Standard,u16 I2S_Mode,u16 I2S_Clock_Polarity,u16 I2S_DataFormat)
{
    I2S_InitTypeDef I2S_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE); //使能 SPI2 时钟
    RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI2, ENABLE); //复位 SPI2
    RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI2, DISABLE); //结束复位

    I2S_InitStructure.I2S_Mode=I2S_Mode;//IIS 模式
    I2S_InitStructure.I2S_Standard=I2S_Standard;//IIS 标准
    I2S_InitStructure.I2S_DataFormat=I2S_DataFormat;//IIS 数据长度
    I2S_InitStructure.I2S_MCLKOutput=I2S_MCLKOutput_Disable;//主时钟输出禁止
    I2S_InitStructure.I2S_AudioFreq=I2S_AudioFreq_Default;//IIS 频率设置
    I2S_InitStructure.I2S_CPOL=I2S_Clock_Polarity;//空闲状态时钟电平
    I2S_Init(SPI2,&I2S_InitStructure);//初始化 IIS

    SPI_I2S_DMAReq(SPI2,SPI_I2S_DMAReq_Tx,ENABLE); //SPI2 TX DMA 请求使能.
    I2S_Cmd(SPI2,ENABLE); //SPI2 I2S EN 使能.
} //采样率计算公式:Fs=I2SxCLK/[256*(2*I2SDIV+ODD)]
//I2SxCLK=(HSE/pllm)*PLLI2SN/PLLI2SR
//一般 HSE=8Mhz
//pllm:在 Sys_Clock_Set 设置的时候确定, 一般是 8
//PLLI2SN:一般是 192~432
//PLLI2SR:2~7
//I2SDIV:2~255
//ODD:0/1
//I2S 分频系数表@pllm=8,HSE=8Mhz,即 vco 输入频率为 1Mhz
//表格式:采样率/10,PLLI2SN,PLLI2SR,I2SDIV,ODD
const u16 I2S_PSC_TBL[][5]=
{
    .....//省略部分代码, 见 48.1.3 节介绍
};
//设置 IIS 的采样率(@MCKEN)
//samplerate:采样率,单位:Hz
//返回值:0,设置成功;1,无法设置.
u8 I2S2_SampleRate_Set(u32 samplerate)
{
    u8 i=0;
```

```
u32 tempreg=0;
samplerate/=10;//缩小 10 倍

for(i=0;i<(sizeof(I2S_PSC_TBL)/10);i++)//看看改采样率是否可以支持
{
    if(samplerate==I2S_PSC_TBL[i][0])break;
}

RCC_PLLI2SCmd(DISABLE);//先关闭 PLLI2S
if(i==(sizeof(I2S_PSC_TBL)/10))return 1;//搜遍了也找不到
RCC_PLLI2SConfig((u32)I2S_PSC_TBL[i][1],(u32)I2S_PSC_TBL[i][2]);
    //设置 I2SxCLK 的频率(x=2) 设置 PLLI2SN PLLI2SR
RCC->CR|=1<<26;           //开启 I2S 时钟
while((RCC->CR&1<<27)==0); //等待 I2S 时钟开启成功.
tempreg=I2S_PSC_TBL[i][3]<<0;//设置 I2SDIV
tempreg|=I2S_PSC_TBL[i][3]<<8; //设置 ODD 位
tempreg|=1<<9;             //使能 MCKOE 位,输出 MCK
SPI2->I2SPR=tempreg;        //设置 I2SPR 寄存器
return 0;
}

//I2S2 TX DMA 配置
//设置为双缓冲模式,并开启 DMA 传输完成中断
//buf0:M0AR 地址.
//buf1:M1AR 地址.
//num:每次传输数据量
void I2S2_TX_DMA_Init(u8* buf0,u8 *buf1,u16 num)
{
    NVIC_InitTypeDef    NVIC_InitStructure;
    DMA_InitTypeDef    DMA_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);//DMA1 时钟使能

    DMA_DeInit(DMA1_Stream4);
    while (DMA_GetCmdStatus(DMA1_Stream4) != DISABLE){}//等待可配置

    /* 配置 DMA Stream */
    DMA_InitStructure.DMA_Channel = DMA_Channel_0; //通道 0 SPI2_TX 通道
    DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&SPI2->DR;//外设地址
    DMA_InitStructure.DMA_Memory0BaseAddr = (u32)buf0;//DMA 存储器 0 地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral;//存储器到外设模式
    DMA_InitStructure.DMA_BufferSize = num;//数据传输量
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
```

//外设数据长度:16 位

```

DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
                                         //存储器数据长度: 16 位
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
DMA_InitStructure.DMA_Priority = DMA_Priority_High;//高优先级
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable; //不使用 FIFO 模式
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_1QuarterFull;
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
                                         //外设突发单次传输
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
                                         //存储器突发单次传输
DMA_Init(DMA1_Stream4, &DMA_InitStructure);//初始化 DMA Stream

DMA_DoubleBufferModeConfig(DMA1_Stream4,(u32)buf1,DMA_Memory_0);
                                         //双缓冲模式配置
DMA_DoubleBufferModeCmd(DMA1_Stream4,ENABLE);//双缓冲模式开启
DMA_ITConfig(DMA1_Stream4,DMA_IT_TC,ENABLE);//开启传输完成中断

NVIC_InitStructure.NVIC IRQChannel = DMA1_Stream4 IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x00;//抢占优先级 0
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x00;//响应优先级 0
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;//使能外部中断通道
NVIC_Init(&NVIC_InitStructure);//配置 }

//I2S DMA 回调函数指针
void (*i2s_tx_callback)(void);    //TX 回调函数
//DMA1_Stream4 中断服务函数
void DMA1_Stream4_IRQHandler(void)
{
    if(DMA_GetITStatus(DMA1_Stream4,DMA_IT_TCIF4)==SET)//传输完成标志
    {
        DMA_ClearITPendingBit(DMA1_Stream4,DMA_IT_TCIF4);
        i2s_tx_callback(); //执行回调函数,读取数据等操作在这里面处理
    }
}

```

其中，I2S2_Init 完成 I2S2 的初始化，通过 4 个参数设置 I2S2 的详细配置信息。另外一个函数：I2S2_SampleRate_Set，则是用前面介绍的查表法，根据音频采样率来设置 I2S 的时钟部分。函数 I2S2_TX_DMA_Init，用于设置 I2S2 的 DMA 发送，使用双缓冲循环模式，发送数据给 WM8978，并开启了发送完成中断。而 DMA1_Stream4_IRQHandler 函数，则是 DMA1 数据流 4 发送完成中断的服务函数，该函数调用 i2s_tx_callback 函数（函数指针，使用前需指向特定函数）实现 DMA 数据填充。在 i2s.c 里面，还有 2 个函数：I2S_Play_Start 和 I2S_Play_Stop，用于开启和关闭 DMA 传输，这里我们没贴出来了，请大家参考光盘本例程源码。

再来看 wm8978.c 里面的几个函数，代码如下：

```
//WM8978 初始化
```

```
//返回值:0,初始化正常
//    其他,错误代码
u8 WM8978_Init(void)
{
    u8 res;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC,
                           ENABLE); //使能外设 GPIOB,GPIOC 时钟

    //PB12/13 复用功能输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化

    //PC2/PC3/PC6 复用功能输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3|GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
    GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化

    GPIO_PinAFConfig(GPIOB,GPIO_PinSource12,GPIO_AF_SPI2);//PB12, I2S_LRCK
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource13,GPIO_AF_SPI2);//PB13 I2S_SCLK
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource3,GPIO_AF_SPI2);//PC3,I2S_DACDATA
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource6,GPIO_AF_SPI2);//PC6,AF5 I2S_MCK
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource2,GPIO_AF6_SPI2);//PC2,I2S_ADCDATA
    IIC_Init(); //初始化 IIC 接口
    res=WM8978_Write_Reg(0,0); //软复位 WM8978
    if(res) return 1; //发送指令失败,WM8978 异常
    //以下为通用设置
    WM8978_Write_Reg(1,0X1B); //R1,MICEN 设置为 1(MIC 使能),BIASEN 设置为 1
                                //(模拟器工作),VMIDSEL[1:0]设置为:11(5K)
    WM8978_Write_Reg(2,0X1B0); //R2,ROUT1,LOUT1 输出使能(耳机可以工作)
                                //,BOOSTENR,BOOSTENL 使能
    WM8978_Write_Reg(3,0X6C); //R3,LOUT2,ROUT2,喇叭输出,RMIX,LMIX 使能
    WM8978_Write_Reg(6,0); //R6,MCLK 由外部提供
    WM8978_Write_Reg(43,1<<4); //R43,INVROUT2 反向,驱动喇叭
    WM8978_Write_Reg(47,1<<8); //R47 设置,PGABOOSTL,左通道 MIC 获得 20 倍增益
    WM8978_Write_Reg(48,1<<8); //R48 设置,PGABOOSTR,右通道 MIC 获得 20 倍增益
```

```
WM8978_Write_Reg(49,1<<1); //R49,TSDEN,开启过热保护
WM8978_Write_Reg(10,1<<3); //R10,SOFTMUTE 关闭,128x 采样,最佳 SNR
WM8978_Write_Reg(14,1<<3); //R14,ADC 128x 采样率
return 0;
}
//WM8978 DAC/ADC 配置
//adcen:adc 使能(1)/关闭(0)
//dacen:dac 使能(1)/关闭(0)
void WM8978_ADDA_Cfg(u8 dacen,u8 adcen)
{
    u16 regval;
    regval=WM8978_Read_Reg(3); //读取 R3
    if(dacen)regval|=3<<0; //R3 最低 2 个位设置为 1,开启 DACR&DACL
    else regval&=~(3<<0); //R3 最低 2 个位清零,关闭 DACR&DACL.
    WM8978_Write_Reg(3,regval); //设置 R3
    regval=WM8978_Read_Reg(2); //读取 R2
    if(adcen)regval|=3<<0; //R2 最低 2 个位设置为 1,开启 ADCR&ADCL
    else regval&=~(3<<0); //R2 最低 2 个位清零,关闭 ADCR&ADCL.
    WM8978_Write_Reg(2,regval); //设置 R2
}
//WM8978 输出配置
//dacen:DAC 输出(放音)开启(1)/关闭(0)
//bpsen:Bypass 输出(录音,包括 MIC,LINE IN,AUX 等)开启(1)/关闭(0)
void WM8978_Output_Cfg(u8 dacen,u8 bpsen)
{
    u16 regval=0;
    if(dacen)regval|=1<<0; //DAC 输出使能
    if(bpsen)
    {
        regval|=1<<1; //BYPASS 使能
        regval|=5<<2; //0dB 增益
    }
    WM8978_Write_Reg(50,regval);//R50 设置
    WM8978_Write_Reg(51,regval);//R51 设置
}
//设置 I2S 工作模式
//fmt:0,LSB(右对齐);1,MSB(左对齐);2,飞利浦标准 I2S;3,PCM/DSP;
//len:0,16 位;1,20 位;2,24 位;3,32 位;
void WM8978_I2S_Cfg(u8 fmt,u8 len)
{
    fmt&=0X03;
    len&=0X03;//限定范围
    WM8978_Write_Reg(4,(fmt<<3)|(len<<5)); //R4,WM8978 工作模式设置
```

}

以上代码 WM8978_Init 用于初始化 WM8978，这里只是通用配置（ADC&DAC），初始化之后，并不能正常播放音乐，还需要通过 WM8978_ADDA_Cfg 函数，使能 DAC，然后通过 WM8978_Output_Cfg 选择 DAC 输出，通过 WM8978_I2S_Cfg 配置 I2S 工作模式，最后设置音量才可以接收 I2S 音频数据，实现音乐播放。这里设置音量、EQ、音效等函数，没有贴出了，请大家参考光盘本例程源码。

接下来，看看 wavplay.c 里面的几个函数，代码如下：

```
__wavctrl wavctrl;      //WAV 控制结构体
vu8 wavtransferend=0;  //i2s 传输完成标志
vu8 wavwitchbuf=0;    //i2sbufx 指示标志
//WAV 解析初始化
//fname:文件路径+文件名
//wavx:wav 信息存放结构体指针
//返回值:0,成功;1,打开文件失败;2,非 WAV 文件;3,DATA 区域未找到.
u8 wav_decode_init(u8* fname,__wavctrl* wavx)
{
    FIL*ftemp; u32 br=0;
    u8 *buf; u8 res=0;
    ChunkRIFF *riff; ChunkFMT *fmt;
    ChunkFACT *fact; ChunkDATA *data;
    ftemp=(FIL*)mymalloc(SRAMIN,sizeof(FIL));
    buf=mymalloc(SRAMIN,512);
    if(ftemp&&buf)   //内存申请成功
    {
        res=f_open(ftemp,(TCHAR*)fname,FA_READ);//打开文件
        if(res==FR_OK)
        {
            f_read(ftemp,buf,512,&br); //读取 512 字节在数据
            riff=(ChunkRIFF *)buf;    //获取 RIFF 块
            if(riff->Format==0X45564157)//是 WAV 文件
            {
                fmt=(ChunkFMT *)(buf+12); //获取 FMT 块
                fact=(ChunkFACT *)(buf+12+8+fmt->ChunkSize); //读取 FACT 块
                if(fact->ChunkID==0X74636166||fact->ChunkID==0X5453494C)
                    wavx->datastart=12+8+fmt->ChunkSize+8+fact->ChunkSize;
                //具有 fact/LIST 块的时候(未测试)
                else wavx->datastart=12+8+fmt->ChunkSize;
                data=(ChunkDATA *)(buf+wavx->datastart); //读取 DATA 块
                if(data->ChunkID==0X61746164)//解析成功!
                {
                    wavx->audioformat=fmt->AudioFormat; //音频格式
                    wavx->nchannels=fmt->NumOfChannels; //通道数
                    wavx->samplerate=fmt->SampleRate; //采样率
                }
            }
        }
    }
}
```

```
wavx->bitrate=fmt->ByteRate*8;           //得到位速
wavx->blockalign=fmt->BlockAlign;          //块对齐
wavx->bps=fmt->BitsPerSample;              //位数,16/24/32 位
wavx->datasize=data->ChunkSize;           //数据块大小
wavx->datastart=wavx->datastart+8;         //数据流开始的地方.

}else res=3;//data 区域未找到.
}else res=2;//非 wav 文件
}else res=1;//打开文件错误
}
f_close(ftemp);
myfree(SRAMIN,ftemp); myfree(SRAMIN,buf); //释放内存
return 0;
}

//填充 buf
//buf:数据区
//size:填充数据量
//bits:位数(16/24)
//返回值:读到的数据个数
u32 wav_bufffill(u8 *buf,u16 size,u8 bits)
{
    u16 readlen=0; u32 bread;
    u16 i; u8 *p;
    if(bits==24)//24bit 音频,需要处理一下
    {
        readlen=(size/4)*3;                      //此次要读取的字节数
        f_read(audiodev.file,audiodev.tbuf,readlen,(UINT*)&bread); //读取数据
        p=audiodev.tbuf;
        for(i=0;i<size;)
        {
            buf[i++]=p[1]; buf[i]=p[2];
            i+=2; buf[i++]=p[0];
            p+=3;
        }
        bread=(bread*4)/3;           //填充后的大小.
    }else
    {
        f_read(audiodev.file,buf,size,(UINT*)&bread);//16bit 音频,直接读取数据
        if(bread<size) for(i=bread;i<size-bread;i++)buf[i]=0;//不够数据了,补充 0
    }
    return bread;
}
//WAV 播放时,I2S DMA 传输回调函数
void wav_i2s_dma_tx_callback(void)
```

```
{  
    u16 i;  
    if(DMA1_Stream4->CR&(1<<19))  
    {  
        wavwitchbuf=0;  
        if((audiodev.status&0X01)==0) //暂停  
            for(i=0;i<WAV_I2S_TX_DMA_BUFSIZE;i++)audiodev.i2sbuf1[i]=0;//填 0  
    }else  
    {  
        wavwitchbuf=1;  
        if((audiodev.status&0X01)==0) //暂停  
            for(i=0;i<WAV_I2S_TX_DMA_BUFSIZE;i++)audiodev.i2sbuf2[i]=0;//填 0  
    }  
    wavtransferend=1;  
}  
//播放某个 WAV 文件  
//fname:wav 文件路径.  
//返回值:  
//KEY0_PRES:下一曲  
//KEY1_PRES:上一曲  
//其他:错误  
u8 wav_play_song(u8* fname)  
{  
    u8 key; u8 t=0; u8 res; u32 fillnum;  
    audiodev.file=(FIL*)mymalloc(SRAMIN,sizeof(FIL));  
    audiodev.i2sbuf1=mymalloc(SRAMIN,WAV_I2S_TX_DMA_BUFSIZE);  
    audiodev.i2sbuf2=mymalloc(SRAMIN,WAV_I2S_TX_DMA_BUFSIZE);  
    audiodev.tbuf=mymalloc(SRAMIN,WAV_I2S_TX_DMA_BUFSIZE);  
    if(audiodev.file&&audiodev.i2sbuf1&&audiodev.i2sbuf2&&audiodev.tbuf)  
    {  
        res=wav_decode_init(fname,&wavctrl);//得到文件的信息  
        if(res==0)//解析文件成功  
        {  
            if(wavctrl.bps==16)  
            {  
                WM8978_I2S_Cfg(2,0);//飞利浦标准,16 位数据长度  
                I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,  
                          I2S_DataFormat_16bextended);  
                //飞利浦标准,主机发送,时钟低电平,16 位扩展帧长度  
            }else if(wavctrl.bps==24)  
            {  
                WM8978_I2S_Cfg(2,2);//飞利浦标准,24 位数据长度  
                I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,
```

```
I2S_DataFormat_24b); //飞利浦标准,主机发送,时钟低,24位扩展帧长度
}
I2S2_SampleRate_Set(wavctrl.samplerate); //设置采样率
I2S2_TX_DMA_Init(audiodev.i2sbuf1,audiodev.i2sbuf2,
                    WAV_I2S_TX_DMA_BUFSIZE/2); //配置 TX DMA
i2s_tx_callback=wav_i2s_dma_tx_callback; //回调函数指 wav_i2s_dma_callback
audio_stop();
res=f_open(audiodev.file,(TCHAR*)fname,FA_READ); //打开文件
if(res==0)
{
    f_lseek(audiodev.file, wavctrl.datastart); //跳过文件头
    fillnum=wav_bufffill(audiodev.i2sbuf1,WAV_I2S_TX_DMA_BUFSIZE,
                          wavctrl.bps);
    fillnum=wav_bufffill(audiodev.i2sbuf2,WAV_I2S_TX_DMA_BUFSIZE,
                          wavctrl.bps);
    audio_start();
    while(res==0)
    {
        while(wavtransferend==0); //等待 wav 传输完成;
        wavtransferend=0;
        if(fillnum!=WAV_I2S_TX_DMA_BUFSIZE) //播放结束?
        { res=KEY0_PRES; break; }
        if(wavwitchbuf)fillnum=wav_bufffill(audiodev.i2sbuf2,
                                              WAV_I2S_TX_DMA_BUFSIZE,wavctrl.bps); //填充 buf2
        else fillnum=wav_bufffill(audiodev.i2sbuf1,
                                   WAV_I2S_TX_DMA_BUFSIZE,wavctrl.bps); //填充 buf1
        while(1)
        {
            key=KEY_Scan(0);
            if(key==WKUP_PRES) //暂停
            {
                if(audiodev.status&0X01)audiodev.status&=~(1<<0);
                else audiodev.status|=0X01;
            }
            if(key==KEY2_PRES||key==KEY0_PRES) //下一曲/上一曲
            { res=key; break; }
            wav_get_curtme(audiodev.file,&wavctrl); //得到播放和总时间
            audio_msg_show(wavctrl.totsec,wavctrl.cursec,wavctrl.bitrate);
            t++;
            if(t==20) { t=0; LED0=!LED0; }
            if((audiodev.status&0X01)==0)delay_ms(10);
            else break;
        }
    }
}
```

```

        }
        audio_stop();
    }else res=0xFF;
}else res=0xFF;
}else res=0xFF;
myfree(SRAMIN,audiodev.tbuf); myfree(SRAMIN,audiodev.file);           //释放内存
myfree(SRAMIN,audiodev.i2sbuf1); myfree(SRAMIN,audiodev.i2sbuf2); //释放内存
return res;
}

```

以上, wav_decode_init 函数, 用来对 wav 文件进行解析, 得到 wav 的详细信息 (音频采样率, 位数, 数据流起始位置等); wav_buffill 函数, 用 f_read 读取数据, 填充数据到 buf 里面, 注意 24 位音频的时候, 读出的数据需要经过转换后才填充到 buf; wav_i2s_dma_tx_callback 函数, 则是 DMA 发送完成的回调函数 (i2s_tx_callback 函数指针指向该函数), 这里面, 我们并没有对数据进行填充处理 (暂停时进行了填 0 处理), 而是采用 2 个标志量: wavtransferend 和 wavwitchbuf, 来告诉 wav_play_song 函数是否传输完成, 以及应该填充哪个数据 buf (i2sbuf1 或 i2sbuf2);

最后, wav_play_song 函数, 是播放 WAV 的最终执行函数, 该函数解析完 WAV 文件后, 设置 WM8978 和 I2S 的参数 (采样率, 位数等), 并开启 DMA, 然后不停填充数据, 实现 WAV 播放, 该函数还进行了按键扫描控制, 实现上下取切换和暂停/播放等操作。该函数通过判断 wavtransferend 是否为 1 来处理是否应该填充数据, 而到底填充到哪个 buf(i2sbuf1 或 i2sbuf2), 则是通过 wavwitchbuf 标志来确定的, 当 wavwitchbuf=0 时, 说明 DMA 正在使用 i2sbuf2, 程序应该填充 i2sbuf1; 当 wavwitchbuf=1 时, 说明 DMA 正在使用 i2sbuf1, 程序应该填充 i2sbuf2;

接下来, 看看 audioplay.c 里面的几个函数, 代码如下:

```

//播放音乐
void audio_play(void)
{
    u8 res; u8 key;u16 temp;
    DIR wavdir;          //目录
    FILINFO wavfileinfo; //文件信息
    u8 *fn;              //长文件名
    u8 *pname;            //带路径的文件名
    u16 totwavnum;        //音乐文件总数
    u16 curindex;         //图片当前索引
    u16 *wavindextbl;     //音乐索引表
    WM8978_ADDA_Cfg(1,0); //开启 DAC
    WM8978_Input_Cfg(0,0,0); //关闭输入通道
    WM8978_Output_Cfg(1,0); //开启 DAC 输出
    while(f_opendir(&wavdir,"0:/MUSIC"))//打开音乐文件夹
    {
        Show_Str(60,190,240,16,"MUSIC 文件夹错误!",16,0); delay_ms(200);
        LCD_Fill(60,190,240,206,WHITE); delay_ms(200); //清除显示
    }
    totwavnum=audio_get_tnum("0:/MUSIC"); //得到总有效文件数
}

```

```
while(totwavnum==NULL)//音乐文件总数为0
{
    Show_Str(60,190,240,16,"没有音乐文件!",16,0); delay_ms(200);
    LCD_Fill(60,190,240,146,WHITE); delay_ms(200); //清除显示
}
wavfileinfo.lfsize=_MAX_LFN*2+1; //长文件名最大长度
wavfileinfo.lfname=mymalloc(SRAMIN,wavfileinfo.lfsize);//为长文件缓存区分配内存
pname=mymalloc(SRAMIN,wavfileinfo.lfsize); //为带路径的文件名分配内存
wavindextbl=mymalloc(SRAMIN,2*totwavnum); //申请内存,用于存放音乐文件索引
while(wavfileinfo.lfname==NULL||pname==NULL||wavindextbl==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); delay_ms(200); //清除显示
}
//记录索引
res=f_opendir(&wavdir,"0:/MUSIC");//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=wavdir.index; //记录当前 index
        res=f_readdir(&wavdir,&wavfileinfo); //读取目录下的一个文件
        if(res!=FR_OK||wavfileinfo.fname[0]==0)break; //错误了/到末尾了,退出
        fn=(u8*)(*wavfileinfo.lfname?wavfileinfo.lfname:wavfileinfo.fname);
        res=f_ttypeell(fn);
        if((res&0XF0)==0X40)//取高四位,看看是不是音乐文件
        {
            wavindextbl[curindex]=temp;//记录索引
            curindex++;
        }
    }
}
curindex=0; //从 0 开始显示
res=f_opendir(&wavdir,(const TCHAR*)"0:/MUSIC");//打开目录
while(res==FR_OK)//打开成功
{
    dir_sdi(&wavdir,wavindextbl[curindex]); //改变当前目录索引
    res=f_readdir(&wavdir,&wavfileinfo); //读取目录下的一个文件
    if(res!=FR_OK||wavfileinfo.fname[0]==0)break; //错误了/到末尾了,退出
    fn=(u8*)(*wavfileinfo.lfname?wavfileinfo.lfname:wavfileinfo.fname);
    strcpy((char*)pname,"0:/MUSIC/");
    strcat((char*)pname,(const char*)fn); //复制路径(目录)
    strcat((char*)pname,(const char*)fn); //将文件名接在后面
}
```

```

LCD_Fill(60,190,240,190+16,WHITE);           //清除之前的显示
Show_Str(60,190,240-60,16,fn,16,0);          //显示歌曲名字
audio_index_show(curindex+1,totwavnum);
key=audio_play_song(pname);                  //播放这个音频文件
if(key==KEY2_PRES)      //上一曲
{
    if(curindex)curindex--;
    else curindex=totwavnum-1;
}else if(key==KEY0_PRES)//下一曲
{
    curindex++;
    if(curindex>=totwavnum)curindex=0;//到末尾的时候,自动从头开始
}else break; //产生了错误
}
myfree(SRAMIN,wavfileinfo.lfname); //释放内存
myfree(SRAMIN,pname);           //释放内存
myfree(SRAMIN,wavindextbl);     //释放内存
}
//播放某个音频文件
u8 audio_play_song(u8* fname)
{
    u8 res;
    res=f_typetell(fname);
    switch(res)
    {
        case T_WAV:
            res=wav_play_song(fname); break;
        default://其他文件,自动跳转到下一曲
            printf("can't play:%s\r\n",fname);
            res=KEY0_PRES; break;
    }
    return res;
}

```

这里，audio_play 函数在 main 函数里面被调用，该函数首先设置 WM8978 相关配置，然后查找 SD 卡里面的 MUSIC 文件夹，并统计该文件夹里面总共有多少音频文件（统计包括：WAV/MP3/APE/FLAC 等），然后，该函数调用 audio_play_song 函数，按顺序播放这些音频文件。

在 audio_play_song 函数里面，通过判断文件类型，调用不同的解码函数，本章，只支持 WAV 文件，通过 wav_play_song 函数实现 WAV 解码。其他格式：MP3/APE/FLAC 等，在综合实验我们会实现其解码函数，大家可以参考综合实验代码，这里就不做介绍了。

最后，我们看看主函数代码：

```

int main(void)
{

```

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
delay_init(168); //初始化延时函数
uart_init(115200); //初始化串口波特率为 115200
LED_Init(); //初始化 LED
usmart_dev.init(84); //初始化 USMART
LCD_Init(); //LCD 初始化
KEY_Init(); //按键初始化
W25QXX_Init(); //初始化 W25Q128
WM8978_Init(); //初始化 WM8978
WM8978_HPvol_Set(40,40); //耳机音量设置
WM8978_SPKvol_Set(50); //喇叭音量设置

my_mem_init(SRAMIN); //初始化内部内存池
my_mem_init(SRAMCCM); //初始化 CCM 内存池
exfun_init(); //为 fatfs 相关变量申请内存
f_mount(fs[0],"0:",1); //挂载 SD 卡
POINT_COLOR=RED;
while(font_init()) //检查字库
{
    LCD_ShowString(30,50,200,16,16,"Font Error!");
    delay_ms(200);
    LCD_Fill(30,50,240,66,WHITE); //清除显示
    delay_ms(200);
}
POINT_COLOR=RED;
Show_Str(60,50,200,16,"Explorer STM32F4 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,"正点原子@ALIENTEK",16,0);
Show_Str(60,110,200,16,"2014 年 5 月 24 日",16,0);
Show_Str(60,130,200,16,"KEY0:NEXT KEY2:PREV",16,0);
Show_Str(60,150,200,16,"KEY_UP:PAUSE/PLAY",16,0);
while(1)
{
    audio_play();
}
}
```

该函数就相对简单了，在初始化各个外设后，通过 `audio_play` 函数，开始音频播放。软件部分就介绍到这里，其他未贴出代码，请参考光盘本例程源码。

48.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 开发板上，程序先执行字库检测，然后当检测到 SD 卡根目录的 MUSIC 文件夹有有效音频文件（WAV 格式音频）的时候，就开始自动播放歌曲了，如图 48.4.1 所示：

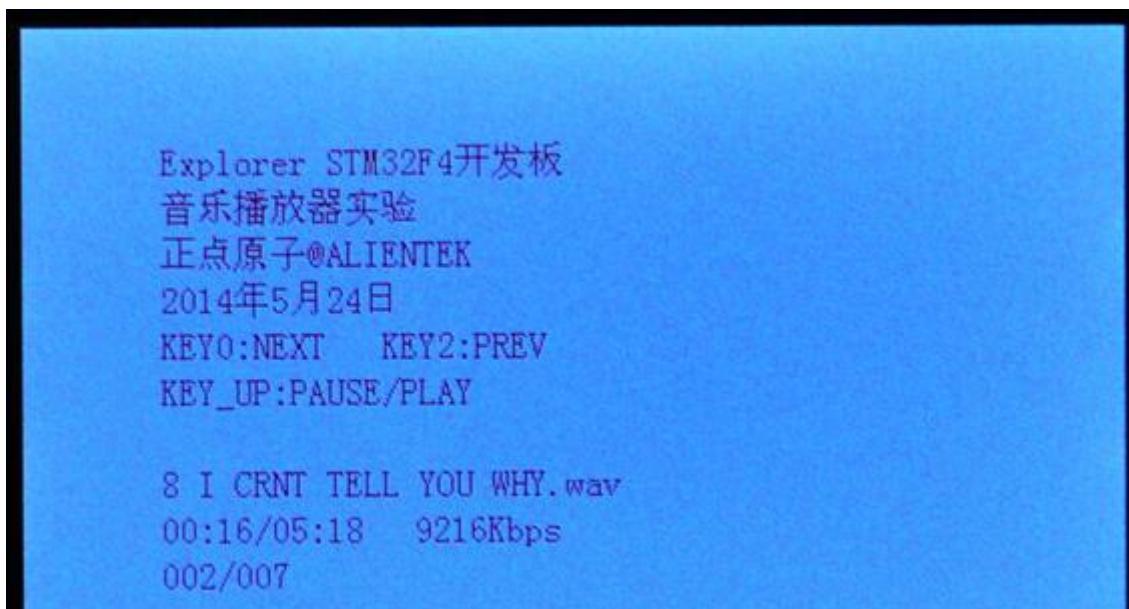


图 48.4.1 MP3 播放中

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。

只要我们在开发板的 PHONE 端子插入耳机（或者在 P1 接口插入喇叭），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 KEY_UP 控制暂停和继续播放。

本实验，我们还可以通过 USMART 来测试 WM8978 的其他功能，通过将 wm8978.c 里面的部分函数加入 USMART 管理，我们可以很方便的设置 wm8978 的各种参数（音量、3D、EQ 等都可以设置），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音乐播放器了，虽然只支持 WAV 文件，但是大家可以在基础上，增加其他音频格式解码器（可参考综合实验），便可实现其他音频格式解码了。

第四十九章 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：

- 49.1 I2S 录音简介
- 49.2 硬件设计
- 49.3 软件设计
- 49.4 下载验证

49.1 I2S 录音简介

本章涉及的知识点基本上在上一章都有介绍。本章要实现 WAV 录音，还是和上一章一样，要了解：WAV 文件格式、WM8978 和 I2S。WAV 文件格式，我们在上一章已经做了详细介绍，这里就不作介绍了。

ALIENTEK 探索者 STM32F4 开发板将板载的一个 MIC 分别接入到了 WM8978 的 2 个差分输入通道（LIP/LIN 和 RIP/RIN，原理图见：图 48.2.1）。代码上，我们采用立体 WAV 声录音，不过，左右声道的音源都是一样的，录音出来的 WAV 文件，听起来就是个单声道效果。

WM8978 上一章也做了比较详细的介绍，本章我们主要看一下要进行 MIC 录音，WM8978 的配置步骤：

- 1, 寄存器 R0 (00h)，该寄存器用于控制 WM8978 的软复位，写任意值到该寄存器地址，即可实现软复位 WM8978。
- 2, 寄存器 R1 (01h)，该寄存器主要设置 MICBEN(bit4)和 BIASEN(bit3)两个位为 1，开启麦克风(MIC)偏置，以及使能模拟部分放大器。
- 3, 寄存器 R2 (02h)，该寄存器要设置 SLEEP(bit6)、INPGAENR(bit3)、INPGAENL(bit2)、ADCENR(bit1)和 ADCENL(bit0)等五个位。SLEEP 设置为 0，进入正常工作模式；INPGAENR 和 INPGAENL 设置为 1，使能 IP PGA 放大器；ADCENL 和 ADCENR 设置为 1，使能左右通道 ADC。
- 4, 寄存器 R4 (04h)，该寄存器要设置 WL(bit6:5)和 FMT(bit4:3)等 4 个位。WL(bit6:5)用于设置字长(即设置音频数据有效位数)，00 表示 16 位音频，10 表示 24 位音频；FMT(bit4:3)用于设置 I2S 音频数据格式(模式)，我们一般设置为 10，表示 I2S 格式，即飞利浦模式。
- 5, 寄存器 R6 (06h)，该寄存器我们直接全部设置为 0 即可，设置 MCLK 和 BCLK 都来自外部，即由 STM32F4 提供。
- 6, 寄存器 R14 (0Eh)，该寄存器要设置 ADCOSR128(bit3)为 1，ADC 得到最好的 SNR。
- 7, 寄存器 R44 (2Ch)，该寄存器我们要设置 LIP2INPPGA(bit0)、LIN2INPPGA(bit1)、RIP2INPPGA(bit4)和 RIN2INPPGA(bit5)等 4 个位，将这 4 个位都设置为 1，将左右通道差分输入接入 IN PGA。
ADCOSR128(bit3)为 1，ADC 得到最好的 SNR。
- 8, 寄存器 R45 (2Dh) 和 R46 (2Eh)，这两个寄存器用于设置 PGA 增益(调节麦克风增益)，一个用于设置左通道(R45)，另外一个用于设置右通道(R46)。这两个寄存器的最高位(INPPGAUPDATE)用于设置是否更新左右通道的增益，最低 6 位用于设置左右通道的增益，我们可以先设置好两个寄存器的增益，最后设置其中一个寄存器最高位为 1，即可更新增益设置。
- 9, 寄存器 R47 (2Fh) 和 R48 (30h)，这两个寄存器也类似，我们只关心其最高位(bit8)，

都设置为 1，可以让左右通道的 MIC 各获得 20dB 的增益。

10，寄存器 R49 (31h)，该寄存器我们要设置 TSDEN(bit1)这个位，设置为 1，开启过热保护。

以上，就是我们用 WM8978 录音时的设置，按照以上所述，对各个寄存器进行相应的配置，即可使用 WM8978 正常录音了。不过我们本章还要用到播放录音的功能，WM8978 的播放配置在 48.1.2 节已经介绍过了，请大家参考这个章节。

上一章我们向大家介绍了 STM32F4 的 I2S 放音，通过上一章的了解，我们知道：STM32F4 的全双工需要用到扩展的 I2Sx_ext (x=2/3)，和 I2Sx 组成全双工 I2S。在全双工模式下，I2Sx 向 I2Sx_ext 提供 CK 和 WS 时钟信号。

本章我们必须向 WM8978 提供 WS，CK 和 MCK 等时钟，同时又要录音，所以只能使用全双工模式。主 I2Sx 循环发送数据 0X0000，给 WM8978，以产生 CK、WS 和 MCK 等信号，从 I2Sx_ext，则接收来自 WM8978 的 ADC 数据 (I2Sxext_SD)，并保存到 SD 卡，实现录音。

本章我们还是采用 I2S2 的全双工模式来录音，I2S2 的相关寄存器，我们在上一章已经介绍的差不多了。至于 I2S2ext 的寄存器，则有一套和 I2S2 一样的寄存器，不过仅仅少数几个对我们有用，他们是：I2S2ext_I2SCFGR、I2S2ext_CR2 和 I2S2ext_DR，这三个寄存器对应为的功能和描述，完全同 I2S2。寄存器描述，我们这里就不再介绍了，大家可以看前面章节，也可以看《STM32F4xx 中文参考手册》第 27.5 节。

最后，我们看看要通过 STM32F4 的 I2S，驱动 WM8978 实现 WAV 录音的简要步骤。这一章用到的硬件部分知识点实际上一章节已经讲解，这里我们主要讲解一下步骤：

1) 初始化 WM8978

这个过程就是前面所讲的 WM8978 MIC 录音配置步骤，让 WM8978 的 ADC 以及其模拟部分工作起来。

2) 初始化 I2S2 和 I2S2ext

本章要用到 I2S2 的全双工模式，所以，I2S2 和 I2S2ext 都需要配置，其中 I2S2 配置为主机发送模式，I2S2ext 设置为从机接收模式。他们的其他配置 (I2S 标准、时钟空闲电平和数据帧长) 基本一样，只是一个是发送一个是接收，且都要使能 DMA。同时，还需要设置音频采样率，不过这个只需要设置 I2S2 即可，还是通过上一章介绍的查表法设置。

3) 设置发送和接收 DMA

放音和录音都是采用 DMA 传输数据的，本章放音起始就是个幌子，不过也得设置 DMA (使用 DMA1 数据流 4 的通道 0)，配置同上一章一模一样，不过不需要开启 DMA 传输完成中断。对于录音，则使用的是 DMA1 数据流 3 的通道 3 实现的 DMA 数据接收，我们需要配置 DMA1 的数据流 3，本章将 DMA1 数据流 3 设置为：双缓冲循环模式，外设和存储器都是 16 位宽，并开启 DMA 传输完成中断 (方便接收数据)。

4) 编写接收通道 DMA 传输完成中断服务函数

为了方便接收音频数据，我们使用 DMA 传输完成中断，每当一个缓冲接数据满了，硬件自动切换为下一个缓冲，同时进入中断服务函数，将已满缓冲的数据写入 SD 卡的 wav 文件。过程如图 49.1.1 所示：

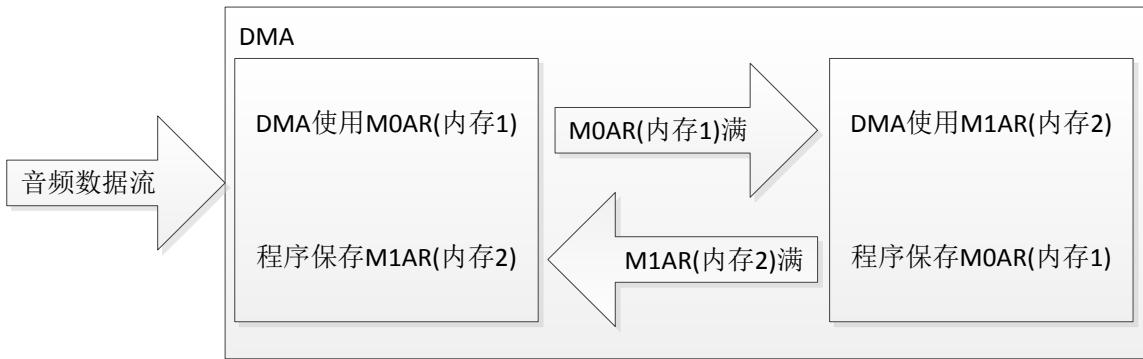


图 49.1.1 DMA 双缓冲接收音频数据流框图

5) 创建 WAV 文件，并保存 wav 头

前面 4 步完成，其实就可以开始读取音频数据了，不过在录音之前，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读取到的的 PCM 音频数据。

6) 开启 DMA 传输，接收数据

然后，我们就只需要开启 DMA 传输，然后及时将 I2S2ext 读到的数据写入到 SD 卡之前新建的 wav 文件里面，就可以实现录音了。

7) 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。

49.2 硬件设计

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，再检测 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，即进入录音模式（包括配置 WM8978 和 I2S 等），此时可以在耳机（或喇叭）听到采集到的音频。`KEY0` 用于开始/暂停录音，`KEY2` 用于保存并停止录音，`KEY_UP` 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、码率以及录音时间等，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），在完成一段录音后，我们可以通过按 `KEY_UP` 按键，来试听刚刚的录音。`DS0` 用于提示程序正在运行，`DS1` 用于提示是否处于暂停录音状态。

本实验用到的资源如下：

- 1) 指示灯 DS0, DS1
- 2) 三个按键 (KEY_UP/KEY0/KEY2)
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡
- 6) SPI FLASH
- 7) WM8978
- 8) I2S2

这些前面都已介绍过。本实验，大家需要准备 1 个 SD 卡和一个耳机（或喇叭），分别插入 SD 卡接口和耳机接口（喇叭接 P1 接口），然后下载本实验就可以实现一个简单的录音机了。

49.3 软件设计

打开本章实验工程可以看到，相比上一章实验，我们主要在 APP 分组下面新增了 recorder.c 和 recorder.h 两个文件。

因为 recorder.c 代码比较多，我们这里仅介绍其中几个重要的函数，代码如下：

```

u8 *i2srecbuf1; //录音 buf1
u8 *i2srecbuf2; //录音 buf2
FIL* f_rec=0; //录音文件
u32 wavsize; //wav 数据大小(字节数,不包括文件头!!)
u8 rec_sta=0; //录音状态
                //#[7]:0,没有开启录音;1,已经开启录音;
                //#[6:1]:保留
                //#[0]:0,正在录音;1,暂停录音;
//录音 I2S_DMA 接收中断服务函数.在中断里面写入数据
void rec_i2s_dma_rx_callback(void)
{
    u16 bw; u8 res;
    if(rec_sta==0X80)//录音模式
    {
        if(DMA1_Stream3->CR&(1<<19))
        {
            res=f_write(f_rec,i2srecbuf1,I2S_RX_DMA_BUF_SIZE,(UINT*)&bw); //写文件
            if(res) printf("write error:%d\r\n",res);
        }
        else
        {
            res=f_write(f_rec,i2srecbuf2,I2S_RX_DMA_BUF_SIZE,(UINT*)&bw); //写文件
            if(res) printf("write error:%d\r\n",res);
        }
        wavsize+=I2S_RX_DMA_BUF_SIZE;
    }
}
const u16 i2splaybuf[2]={0X0000,0X0000}; //2 个 16 位数据,用于录音时 I2S2 主机循环发送.
//进入 PCM 录音模式
void recoder_enter_rec_mode(void)
{
    WM8978_ADDA_Cfg(0,1); //开启 ADC
    WM8978_Input_Cfg(1,1,0); //开启输入通道(MIC&LINE IN)
    WM8978_Output_Cfg(0,1); //开启 BYPASS 输出
    WM8978_MIC_Gain(46); //MIC 增益设置

    WM8978_I2S_Cfg(2,0); //飞利浦标准,16 位数据长度
    I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,
              I2S_DataFormat_16b); //飞利浦标准,主机发送,时钟低电平有效,16 位帧长度

```

```
I2S2ext_Init(I2S_Standard_Phillips,I2S_Mode_SlaveRx,I2S_CPOL_Low,
I2S_DataFormat_16b); //飞利浦标准,从机接收,时钟低电平有效,16位帧长度
I2S2_SetSampleRate_Set(16000); //设置采样率
I2S2_TX_DMA_Init((u8*)&i2splaybuf[0],(u8*)&i2splaybuf[1],1); //配置 TX DMA
DMA1_Stream4->CR&= ~(1<<4); //关闭传输完成中断(这里不用中断送数据)
I2S2ext_RX_DMA_Init(i2srecbuf1,i2srecbuf2,I2S_RX_DMA_BUFSIZE/2); //RX DMA
i2s_rx_callback=rec_i2s_dma_rx_callback; //回调函数指 wav_i2s_dma_callback
I2S_Play_Start(); //开始 I2S 数据发送(主机)
I2S_Rec_Start(); //开始 I2S 数据接收(从机)
recoder_remindmsg_show(0);
} //进入 PCM 放音模式
void recoder_enter_play_mode(void)
{
    WM8978_ADDA_Cfg(1,0); //开启 DAC
    WM8978_Input_Cfg(0,0,0); //关闭输入通道(MIC&LINE IN)
    WM8978_Output_Cfg(1,0); //开启 DAC 输出
    WM8978_MIC_Gain(0); //MIC 增益设置为 0
    I2S_Play_Stop(); //停止时钟发送
    I2S_Rec_Stop(); //停止录音
    recoder_remindmsg_show(1);
}
//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=2; //双声道
    wavhead->fmt.SampleRate=16000; //16Khz 采样率 采样速率
    wavhead->fmt.ByteRate=wavhead->fmt.SampleRate*4;
    //字节速率=采样率*通道数*(ADC 位数/8)
    wavhead->fmt.BlockAlign=4; //块大小=通道数*(ADC 位数/8)
    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
}
//WAV 录音
void wav_recorder(void)
{
    u8 res; u8 key; u8 rval=0;
```

```
_WaveHeader *wavhead=0;
DIR recdir;          //目录
u8 *pname=0;
u8 timecnt=0;       //计时器
u32 recsec=0;        //录音时间
while(f_opendir(&recdir,"0:/RECODER"))//打开录音文件夹
{
    Show_Str(30,230,240,16,"RECODER 文件夹错误!",16,0); delay_ms(200);
    LCD_Fill(30,230,240,246,WHITE); delay_ms(200);//清除显示
    f_mkdir("0:/RECODER");                      //创建该目录
}
i2srecbuf1=mymalloc(SRAMIN,I2S_RX_DMA_BUF_SIZE);//I2S 录音内存 1 申请
i2srecbuf2=mymalloc(SRAMIN,I2S_RX_DMA_BUF_SIZE);//I2S 录音内存 2 申请
f_rec=(FIL *)mymalloc(SRAMIN,sizeof(FIL));      //开辟 FIL 字节的内存区域
wavhead=(_WaveHeader*)mymalloc(SRAMIN,sizeof(_WaveHeader));//开辟内存
pname=mymalloc(SRAMIN,30);//申请 30 字节内存
if(!i2srecbuf1||!i2srecbuf2 ||!f_rec ||!wavhead ||!pname)rval=1;
if(rval==0)
{
    recoder_enter_rec_mode(); //进入录音模式,此时耳机可以听到咪头采集到的音频
    pname[0]=0;               //pname 没有任何文件名
    while(rval==0)
    {
        key=KEY_Scan(0);
        switch(key)
        {
            case KEY2_PRES: //STOP&SAVE
                if(rec_sto&0X80)//有录音
                {
                    rec_sto=0;    //关闭录音
                    wavhead->riff.ChunkSize=wavsize+36;   //整个文件的大小-8;
                    wavhead->data.ChunkSize=wavsize;        //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);
                    wavsize=0;
                }
                rec_sto=0; recsec=0;
                LED1=1;           //关闭 DS1
                LCD_Fill(30,190,lcddev.width,lcddev.height,WHITE); //清除显示
                break;
            case KEY0_PRES: //REC/PAUSE

```

```
if(rec_sta&0X01) rec_sta&=0XFE;//原来是暂停,继续录音
else if(rec_sta&0X80) rec_sta|=0X01;//已经在录音了,暂停
else//还没开始录音
{
    recsec=0;
    recoder_new.pathname(pname);           //得到新的名字
    Show_Str(30,190, lcddev.width,16,"录制:",16,0);
    Show_Str(30+40,190, lcddev.width,16, pname+11,16,0); //显示名字
    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
        recoder_msg_show(0,0);
        rec_sta|=0X80;    //开始录音
    }
}
if(rec_sta&0X01)LED1=0; //提示正在暂停
else LED1=1;
break;

case WKUP_PRES: //播放最近一段录音
if(rec_sta!=0X80)//没有在录音
{
    if(pname[0])//如果触摸按键被按下,且 pname 不为空
    {
        Show_Str(30,190, lcddev.width,16,"播放:",16,0);
        Show_Str(30+40,190, lcddev.width,16, pname+11,16,0); //名字
        recoder_enter_play_mode(); //进入播放模式
        audio_play_song(pname); //播放 pname
        LCD_Fill(30,190, lcddev.width, lcddev.height,WHITE); //清除
        recoder_enter_rec_mode(); //重新进入录音模式
    }
}
break;
}

delay_ms(5);
timecnt++;
```

```

if((timecnt%20)==0)LED0=!LED0;//DS0 闪烁
if(recsec!=(wavsize/wavhead->fmt.ByteRate)) //录音时间显示
{
    LED0=!LED0;//DS0 闪烁
    recsec=wavsize/wavhead->fmt.ByteRate; //录音时间
    recoder_msg_show(recsec,wavhead->fmt.SampleRate*wavhead->fmt.
                      NumOfChannels*wavhead->fmt.BitsPerSample);//显示码率
}
}

myfree(SRAMIN,i2srecbuf1); //释放内存
myfree(SRAMIN,i2srecbuf2); //释放内存
myfree(SRAMIN,f_rec); //释放内存
myfree(SRAMIN,wavhead); //释放内存
myfree(SRAMIN,pname); //释放内存
}

```

这里总共 5 个函数，其中：rec_i2s_dma_rx_callback 函数，用于 I2S2ext 的 DMA 接收完成中断回调函数（通过 i2s_rx_callback 指向该函数实现），在该函数里面，实现音频数据的保存。recoder_enter_rec_mode 函数，用于设置 WM8978 和 I2S 进入录音模式（开始录音时用到）。recoder_enter_play_mode 函数，则用于设置 WM8978 和 I2S 进入播放模式（录音回放时用到）。recoder_wav_init 函数，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 16Khz 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 结构体就是由上一章（48.1.1 节）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，wav_recorder 函数，实现了我们在硬件设计时介绍的功能（开始/暂停录音、保存录音文件、播放最近一次录音等）。该函数使用上一章实现的 audio_play_song 函数，来播放最近一次录音。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。

然后，我们在 i2s.c 里面也增加了几个函数，如下：

```

//I2S2ext 配置
//参数 I2S_Standard: @ref SPI_I2S_Standard I2S 标准,
//参数 I2S_Mode: @ref SPI_I2S_Mode 模式
//参数 I2S_Clock_Polarity @ref SPI_I2S_Clock_Polarity:
//参数 I2S_DataFormat: @ref SPI_I2S_Data_Format
Void I2S2ext_Init(u16 I2S_Standard,u16 I2S_Mode,

```

```

        u16 I2S_Clock_Polarity,u16 I2S_DataFormat)
{
    I2S_InitTypeDef I2S2ext_InitStructure;

    I2S2ext_InitStructure.I2S_Mode=I2S_Mode^(1<<8);//IIS 模式
    I2S2ext_InitStructure.I2S_Standard=I2S_Standard;//IIS 标准
    I2S2ext_InitStructure.I2S_DataFormat=I2S_DataFormat;//IIS 数据长度
    I2S2ext_InitStructure.I2S_MCLKOutput=I2S_MCLKOutput_Disable;//主时钟输出禁止
    I2S2ext_InitStructure.I2S_AudioFreq=I2S_AudioFreq_Default;//IIS 频率设置
    I2S2ext_InitStructure.I2S_CPOL=I2S_Clock_Polarity;//空闲状态时钟电平

    I2S_FullDuplexConfig(I2S2ext,&I2S2ext_InitStructure);//初始化 I2S2ext 配置

    SPI_I2S_DMACmd(I2S2ext,SPI_I2S_DMAReq_Rx,ENABLE);//I2S2ext DMA 请求使能.

    I2S_Cmd(I2S2ext,ENABLE);           //I2S2ext I2S EN 使能.
}

//I2S2ext RX DMA 配置
//设置为双缓冲模式,并开启 DMA 传输完成中断
//buf0:M0AR 地址.
//buf1:M1AR 地址.
//num:每次传输数据量
void I2S2ext_RX_DMA_Init(u8* buf0,u8 *buf1,u16 num)
{
    NVIC_InitTypeDef    NVIC_InitStructure;
    DMA_InitTypeDef    DMA_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);//DMA1 时钟使能

    DMA_DeInit(DMA1_Stream3);
    while (DMA_GetCmdStatus(DMA1_Stream3) != DISABLE){ } //等待可配置

    DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_FEIF3|DMA_IT_DMEIF3|
                           DMA_IT_TEIF3|DMA_IT_HTIF3|DMA_IT_TCIF3);
                           //清空 DMA1_Stream3 上所有中断标志

/* 配置 DMA Stream */
    DMA_InitStructure.DMA_Channel = DMA_Channel_3; //通道 3 I2S2ext_RX 通道
    DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&I2S2ext->DR;//外设地址
    DMA_InitStructure.DMA_Memory0BaseAddr = (u32)buf0;//DMA 存储器 0 地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//外设到存储器模式
    DMA_InitStructure.DMA_BufferSize = num;//数据传输量
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
}

```

```

DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;//中等优先级
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable; //不使用 FIFO 模式
DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_1QuarterFull;
DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
DMA_Init(DMA1_Stream3, &DMA_InitStructure);//初始化 DMA Stream

DMA_DoubleBufferModeConfig(DMA1_Stream3,(u32)buf1,DMA_Memory_0);
//双缓冲模式配置
DMA_DoubleBufferModeCmd(DMA1_Stream3,ENABLE);//双缓冲模式开启
DMA_ITConfig(DMA1_Stream3,DMA_IT_TC,ENABLE);//开启传输完成中断

NVIC_InitStructure.NVIC IRQChannel = DMA1_Stream3 IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0x00;//抢占优先级 0
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0x01;//响应优先级 1
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;//使能外部中断通道
NVIC_Init(&NVIC_InitStructure);//配置
}

void (*i2s_rx_callback)(void); //RX 回调函数
//DMA1_Stream3 中断服务函数
void DMA1_Stream3_IRQHandler(void)
{
    if(DMA_GetITStatus(DMA1_Stream3,DMA_IT_TCIF3)==SET)//传输完成标志
    {
        DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_TCIF3); //清除传输完成中断
        i2s_rx_callback(); //执行回调函数,读取数据等操作在这里面处理
    }
}
//I2S 开始录音
void I2S_Rec_Start(void)
{
    DMA_Cmd(DMA1_Stream3,ENABLE);//开启 DMA TX 传输,开始录音
}
//关闭 I2S 录音
void I2S_Rec_Stop(void)
{
    DMA_Cmd(DMA1_Stream3,DISABLE);//关闭 DMA,结束录音
}

```

这里也是 5 个函数， I2S2ext _Init 函数完成 I2S2ext 的初始化，通过 4 个参数设置 I2S2ext

的详细配置信息。I2S2ext_RX_DMA_Init 函数，用于设置 I2S2ext 的 DMA 接收，使用双缓冲循环模式，接收来自 WM8978 的数据，并开启了传输完成中断。而 DMA1_Stream3_IRQHandler 函数，则是 DMA1 数据流 3 传输完成中断的服务函数，该函数调用 i2s_rx_callback 函数（函数指针，使用前需指向特定函数）实现 DMA 数据接收保存。最后，I2S_Rec_Start 和 I2S_Rec_Stop，用于开启和关闭 DMA 传输。

其他代码，我们就不再介绍了，请大家参考开发板光盘本例程源码。最后，我们看看主函数代码：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); // 设置系统中断优先级分组 2
    delay_init(168); // 初始化延时函数
    uart_init(115200); // 初始化串口波特率为 115200
    LED_Init(); // 初始化 LED
    usmart_dev.init(84); // 初始化 USMART
    LCD_Init(); // LCD 初始化
    KEY_Init(); // 按键初始化
    W25QXX_Init(); // 初始化 W25Q128
    WM8978_Init(); // 初始化 WM8978
    WM8978_HPvol_Set(40,40); // 耳机音量设置
    WM8978_SPKvol_Set(50); // 喇叭音量设置

    my_mem_init(SRAMIN); // 初始化内部内存池
    my_mem_init(SRAMCCM); // 初始化 CCM 内存池
    exfuns_init(); // 为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1); // 挂载 SD 卡
    POINT_COLOR=RED;
    while(font_init()) // 检查字库
    {
        LCD_ShowString(30,40,200,16,16,"Font Error!");
        delay_ms(200);
        LCD_Fill(30,40,240,66,WHITE); // 清除显示
        delay_ms(200);
    }
    POINT_COLOR=RED;
    Show_Str(30,40,200,16,"Explorer STM32 开发板",16,0);
    Show_Str(30,60,200,16,"录音机实验",16,0);
    Show_Str(30,80,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,100,200,16,"2014 年 6 月 6 日",16,0);
    while(1)
    {
        wav_recorder();
    }
}
```

该函数代码同上一章的 main 函数代码几乎一样，十分简单，我们就不再多说了。

至此，本实验的软件设计部分结束。

49.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 开发板上，程序先检测字库，然后检测 SD 卡的 RECORDER 文件夹，一切顺利通过之后，进入录音模式，得到，如图 49.4.1 所示：

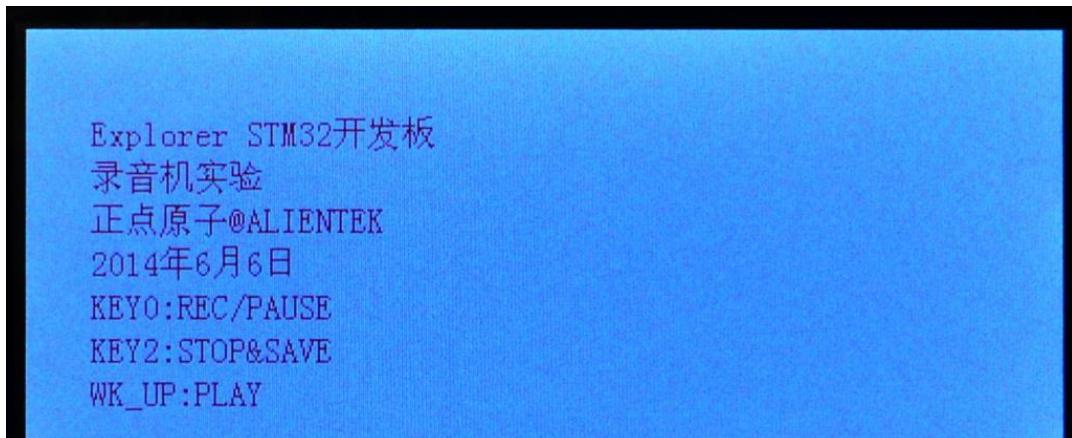


图 49.4.1 录音机界面

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、码率以及录音时长，如图 49.4.2 所示：

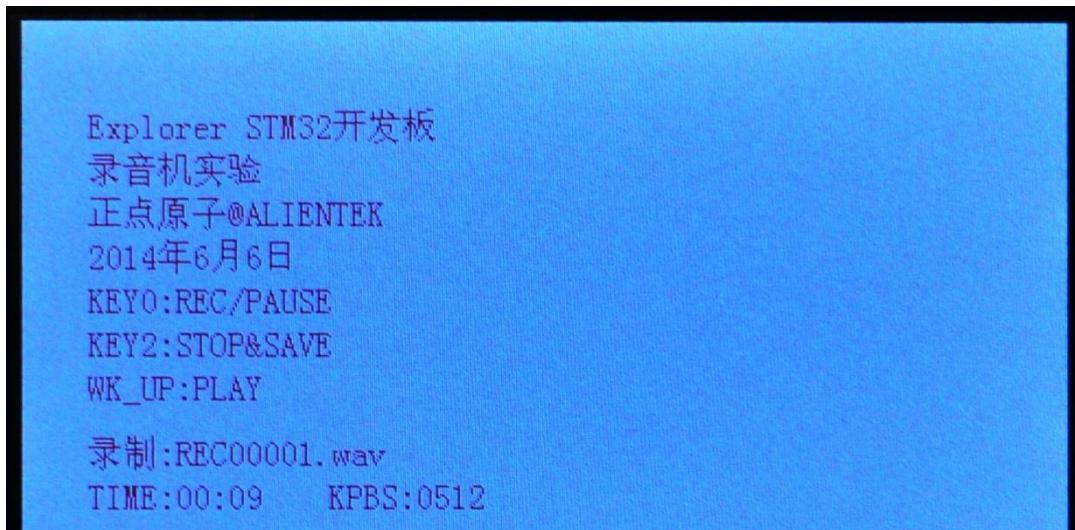


图 49.4.2 录音进行中

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。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 KEY_UP 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 49.4.3 所示：



图 49.4.3 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。

第五十章 视频播放器实验

STM32F4 的处理能力，不仅可以软解码音频，还可以用来播放视频！本章，我们将使用探索者 STM32F4 开发板来播放 AVI 视频，本章我们将实现一个简单的视频播放器，实现 AVI 视频播放。本章分为如下几个部分：

- 50.1 AVI&libjpeg 简介
- 50.2 硬件设计
- 50.3 软件设计
- 50.4 下载验证

50.1 AVI&libjpeg 简介

本章，我们使用 libjpeg（由 IJG 提供），来实现 MJPG 编码的 AVI 格式视频播放，我们先来简单介绍一下 AVI 和 libjpeg。

50.1.1 AVI 简介

AVI 是音频视频交错(Audio Video Interleaved)的英文缩写，它是微软开发的一种符合 RIFF 文件规范的数字音频与视频文件格式，原先用于 Microsoft Video for Windows (简称 VFW)环境，现在已被多数操作系统直接支持。

AVI 格式允许视频和音频交错在一起同步播放，支持 256 色和 RLE 压缩，但 AVI 文件并未限定压缩标准，AVI 仅仅是一个容器，用不同压缩算法生成的 AVI 文件，必须使用相应的解压缩算法才能播放出来。比如本章，我们使用的 AVI，其音频数据采用 16 位线性 PCM 格式（未压缩），而视频数据，则采用 MJPG 编码方式。

在介绍 AVI 文件前，我们要先来看看 RIFF 文件结构。AVI 文件采用的是 RIFF 文件结构方式，RIFF（Resource Interchange File Format，资源互换文件格式）是微软定义的一种用于管理 WINDOWS 环境中多媒体数据的文件格式，波形音频 WAVE，MIDI 和数字视频 AVI 都采用这种格式存储。构造 RIFF 文件的基本单元叫做数据块（Chunk），每个数据块包含 3 个部分，

- 1、4 字节的数据块标记（或者叫做数据块的 ID）
- 2、数据块的大小
- 3、数据

整个 RIFF 文件可以看成一个数据块，其数据块 ID 为 RIFF，称为 RIFF 块。一个 RIFF 文件中只允许存在一个 RIFF 块。RIFF 块中包含一系列的子块，其中有一种子块的 ID 为"LIST"，称为 LIST 块，LIST 块中可以再包含一系列的子块，但除了 LIST 块外的其他所有的子块都不能再包含子块。

RIFF 和 LIST 块分别比普通的数据块多一个被称为形式类型 (Form Type) 和列表类型 (List Type) 的数据域，其组成如下：

- 1、4 字节的数据块标记 (Chunk ID)
- 2、数据块的大小
- 3、4 字节的形式类型或者列表类型 (ID)
- 4、数据

下面我们看看 AVI 文件的结构。AVI 文件是目前使用的最复杂的 RIFF 文件，它能同时存储同步表现的音频视频数据。AVI 的 RIFF 块的形式类型 (Form Type) 是 AVI，它一般包含 3

个子块，如下所述：

- 1、信息块，一个 ID 为"hdrl"的 LIST 块，定义 AVI 文件的数据格式。
 - 2、数据块，一个 ID 为 "movi"的 LIST 块，包含 AVI 的音视频序列数据。
 - 3、索引块，ID 为"idxl"的子块，定义"movi"LIST 块的索引数据，是可选块（不一定有）。
- 接下来，我们详细介绍下 AVI 文件的各子块构造，AVI 文件的结构如图 50.1.1.1 所示：

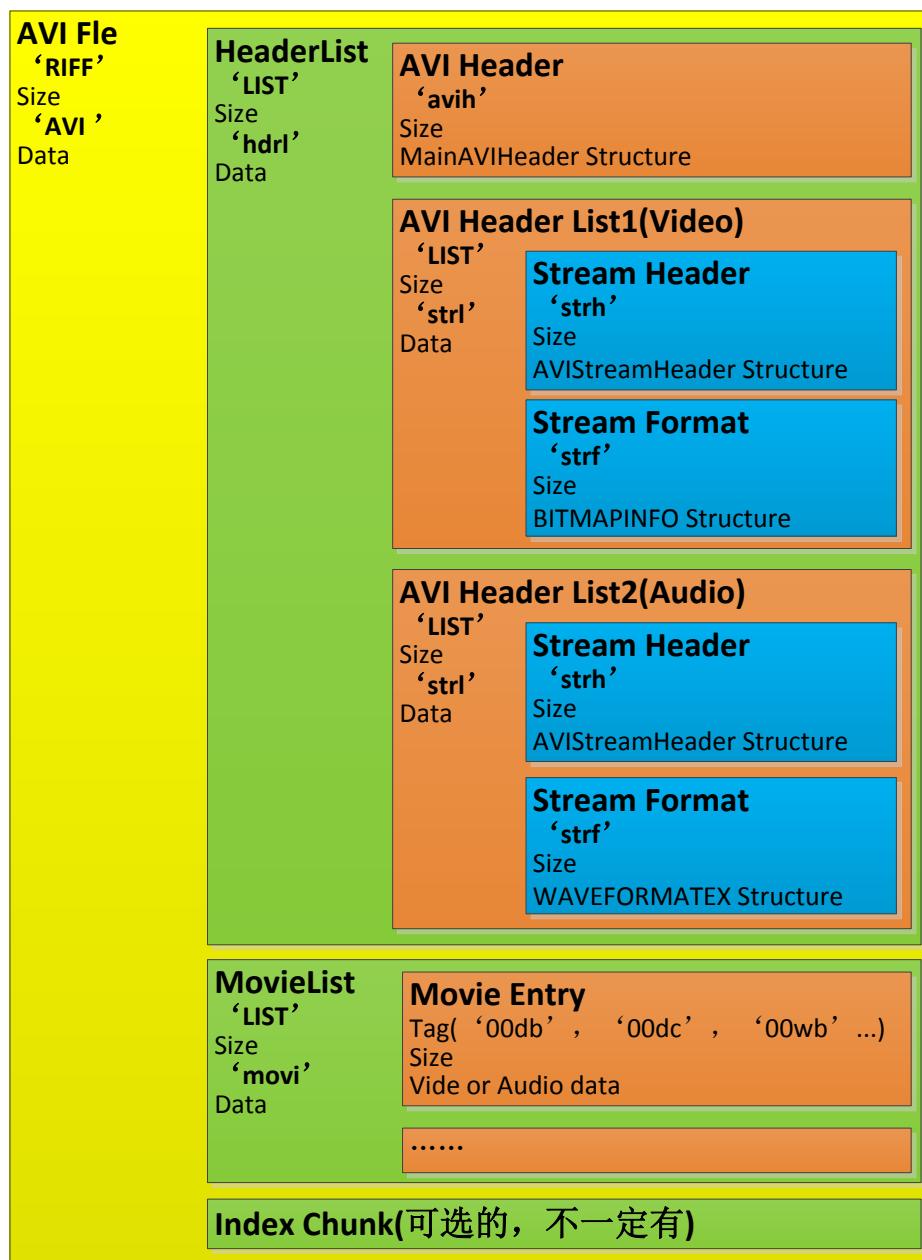


图 50.1.1.1 AVI 文件结构图

从上图可以看出(注意‘AVI’，是带了一个空格的)，AVI 文件，由：信息块(HeaderList)、数据块(MovieList) 和索引块(Index Chunk) 等三部分组成，下面，我们分别介绍这几个部分。

1，信息块 (HeaderList)

信息块，即 ID 为 “hdrl” 的 LIST 块，它包含文件的通用信息，定义数据格式，所用的压缩算法等参数等。hdrl 块还包括了一系列的字块，首先是：avih 块，用于记录 AVI 的全局信息，

比如数据流的数量，视频图像的宽度和高度等信息，avih 块(结构体都有把 BlockID 和 BlockSize 包含进来，下同) 的定义如下：

```
//avih 子块信息
typedef struct
{
    u32 BlockID;           //块标志:avih==0X61766968
    u32 BlockSize;         //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    u32 SecPerFrame;       //视频帧间隔时间(单位为 us)
    u32 MaxByteSec;        //最大数据传输率,字节/秒
    u32 PaddingGranularity; //数据填充的粒度
    u32 Flags;             //AVI 文件的全局标记, 比如是否含有索引块等
    u32 TotalFrame;         //文件总帧数
    u32 InitFrames;        //为交互格式指定初始帧数 (非交互格式应该指定为 0)
    u32 Streams;           //包含的数据流种类个数,通常为 2
    u32 RefBufSize;         //建议读取本文件的缓存大小 (应能容纳最大的块)
    u32 Width;              //图像宽
    u32 Height;             //图像高
    u32 Reserved[4];        //保留
}AVIH_HEADER;
```

这里有很多我们要用到的信息，比如 SecPerFrame，通过该参数，我们可以知道每秒钟的帧率，也就知道了每秒钟需要解码多少帧图片，才能正常播放。TotalFrame 告诉我们整个视频有多少帧，结合 SecPerFrame 参数，就可以很方便计算整个视频的时间了。Streams 告诉我们数据流的种类数，一般是 2，即包含视频数据流和音频数据流。

在 avih 块之后，是一个或者多个 strl 子列表，文件中有多少种数据流(即前面的 Streams)，就有多少个 strl 子列表。每个 strl 子列表，至少包括一个 strh(Stream Header)块和一个 strf(Stream Format) 块，还有一个可选的 strn (Stream Name) 块 (未列出)。注意：strl 子列表出现的顺序与媒体流的编号 (比如：00dc，前面的 00，即媒体流编号 00) 是对应的，比如第一个 strl 子列表说明的是第一个流 (Stream 0)，假设是视频流，则表征视频数据块的四字符码为“00dc”，第二个 strl 子列表说明的是第二个流 (Stream 1)，假设是音频流，则表征音频数据块的四字符码为“01dw”，以此类推。

先看 strh 子块，该块用于说明这个流的头信息，定义如下：

```
//strh 流头子块信息(strh∈strl)
typedef struct
{
    u32 BlockID;           //块标志:strh==0X73747268
    u32 BlockSize;          //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    u32 StreamType;         //数据流种类, vids(0X73646976):视频;auds(0X73647561):音频
    u32 Handler;            //指定流的处理器,对于音视频来说即解码器,如 MJPG/H264 等.
    u32 Flags;              //标记: 是否允许这个流输出? 调色板是否变化?
    u16 Priority;           //流的优先级 (当有多个同类型的流时优先级最高的为默认流)
    u16 Language;           //音频的语言代号
    u32 InitFrames;          //为交互格式指定初始帧数
    u32 Scale;               //数据量, 视频每桢的大小或者音频的采样大小
```

```

u32 Rate;           //Scale/Rate=每秒采样数
u32 Start;         //数据流开始播放的位置, 单位为 Scale
u32 Length;        //数据流的数据量, 单位为 Scale
u32 RefBufSize;    //建议使用的缓冲区大小
u32 Quality;       //解压缩质量参数, 值越大, 质量越好
u32 SampleSize;    //音频的样本大小
struct             //视频帧所占的矩形
{
    short Left;
    short Top;
    short Right;
    short Bottom;
}Frame;
}STRH_HEADER;

```

这里面, 对我们最有用的即 StreamType 和 Handler 这两个参数了, StreamType 用于告诉我们此 strl 描述的是音频流 (“auds”), 还是视频流 (“vids”)。而 Handler 则告诉我们所使用的解码器, 比如 MJPG/H264 等 (实际以 strf 块为准)。

然后是 strf 子块, 不过 strf 字块, 需要根据 strh 字块的类型而定。

如果 strh 子块是视频数据流 (StreamType= “vids”), 则 strf 子块的内容定义如下:

```

//BMP 结构体
typedef struct
{
    u32 BmpSize;          //bmp 结构体大小,包含(BmpSize 在内)
    long Width;           //图像宽
    long Height;          //图像高
    u16 Planes;           //平面数, 必须为 1
    u16 BitCount;         //像素位数,0X0018 表示 24 位
    u32 Compression;      //压缩类型, 比如:MJPG/H264 等
    u32 SizeImage;        //图像大小
    long XpixPerMeter;    //水平分辨率
    long YpixPerMeter;    //垂直分辨率
    u32 ClrUsed;          //实际使用了调色板中的颜色数,压缩格式中不使用
    u32 ClrImportant;     //重要的颜色
}BMP_HEADER;
//颜色表
typedef struct
{
    u8 rgbBlue;           //蓝色的亮度(值范围为 0-255)
    u8 rgbGreen;          //绿色的亮度(值范围为 0-255)
    u8 rgbRed;            //红色的亮度(值范围为 0-255)
    u8 rgbReserved;       //保留, 必须为 0
}AVIRGBQUAD;
//对于 strh,如果是视频流,strf(流格式)使 STRH_BMPHEADER 块

```

```

typedef struct
{
    u32 BlockID;          //块标志,strf==0X73747266
    u32 BlockSize;        //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    BMP_HEADER bmiHeader; //位图信息头
    AVIRGBQUAD bmColors[1]; //颜色表
}STRF_BMPHEADER;

```

这里有 3 个结构体，strf 子块完整内容即：STRF_BMPHEADER 结构体，不过对我们有用的信息，都存放在 BMP_HEADER 结构体里面，本结构体对视频数据的解码起决定性的作用，它告诉我们视频的分辨率（Width 和 Height），以及视频所用的编码器（Compression），因此它决定了视频的解码。本章例程仅支持解码视频分辨率小于屏幕分辨率，且编解码器必须是 MJPG 的视频格式。

如果 strh 子块是音频数据流（StreamType=“auds”），则 strf 子块的内容定义如下：

```
//对于 strh,如果是音频流,strf(流格式)使 STRF_WAVHEADER 块
```

```

typedef struct
{
    u32 BlockID;          //块标志,strf==0X73747266
    u32 BlockSize;        //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    u16 FormatTag;        //格式标志:0X0001=PCM,0X0055=MP3...
    u16 Channels;         //声道数,一般为 2,表示立体声
    u32 SampleRate;        //音频采样率
    u32 BaudRate;         //波特率
    u16 BlockAlign;        //数据块对齐标志
    u16 Size;              //该结构大小
}STRF_WAVHEADER;

```

本结构体对音频数据解码起决定性的作用，他告诉我们音频信号的编码方式（FormatTag）、声道数（Channels）和采样率（SampleRate）等重要信息。本章例程仅支持 PCM 格式（FormatTag=0X0001）的音频数据解码。

2. 数据块（MovieList）

信息块，即 ID 为“movi”的 LIST 块，它包含 AVI 的音视频序列数据，是这个 AVI 文件的主体部分。音视频数据块交错的嵌入在“movi”LIST 块里面，通过标准类型码进行区分，标准类型码有如下 4 种：

- 1, “##db”（非压缩视频帧）、
- 2, “##dc”（压缩视频帧）、
- 3, “##pc”（改用新的调色板）、
- 4, “##wb”（音频帧）。

其中##是编号，得根据我们的数据流顺序来确定，也就是前面的 str1 块。比如，如果第一个 str1 块是视频数据，那么对于压缩的视频帧，标准类型码就是：00dc。第二个 str1 块是音频数据，那么对于音频帧，标准类型码就是：01wb。

紧跟着标准类型码的是 4 个字节的数据长度（不包含类型码和长度参数本身，也就是总长度必须要加 8 才对），该长度必须是偶数，如果读到为奇数，则加 1 即可。我们读数据的时候，一般一次性要读完一个标准类型码所表征的数据，方便解码。

3. 索引块 (Index Chunk)

最后，紧跟在 ‘hdr1’ 列表和 ‘movi’ 列表之后的，就是 AVI 文件可选的索引块。这个索引块为 AVI 文件中每一个媒体数据块进行索引，并且记录它们在文件中的偏移（可能相对于 ‘movi’ 列表，也可能相对于 AVI 文件开头）。本章我们用不到索引块，这里就不详细介绍。

关于 AVI 文件，我们就介绍到这，有兴趣的朋友，可以再看看光盘：6，软件资料→AVI 学习资料 里面的相关文档。

50.1.2 libjpeg 简介

libjpeg 是一个完全用 C 语言编写的库，包含了被广泛使用的 JPEG 解码、JPEG 编码和其他的 JPEG 功能的实现。这个库由 IJG 组织 (Independent JPEG Group (独立 JPEG 小组)) 提供，并维护。libjpeg，目前最新版本为 v9a，可以在：<http://www.ijg.org> 这个网站下载到。libjpeg 具有稳定、兼容性强和解码速度较快等优点。

本章，我们使用 libjpeg v9a 来实现 MJPG 数据流的解码，MJPG 数据流，其实就是一张张的 JPEG 图片拼起来的图片视频流，只要能快速解码 JPEG 图片，就可以实现视频播放。

前面的图片显示实验我们使用了 TJPJD 来做 JPEG 解码，大家可能会问，为什么不直接用 TJPJD 来解码呢？原因就是 TJPJD 的特点就是：占用资源少，但是解码速度慢。在 STM32F4 上，同样一张 320*240 的 JPG 图片，用 TJPJD 来解码，得 120 多 ms，而用 libjpeg，则只需要 50ms 左右即可完成解码，明显速度上 libjpeg 要快不少，使得解码视频成为可能。实际上，经过我们优化后的 libjpeg，使用 STM32F4，在不超频的情况下，可以流畅播放 480*272@10 帧的 MJPG 视频（带音频）。

篇幅所限，关于 libjpeg 的移植，我们这里就不介绍了，请大家参考光盘源码。关于 libjpeg 的移植和使用，其实在下载的 libjpeg 源码里面，就有很多介绍，大家重点可以看：readme.txt、filelist.txt、install.txt 和 libjpeg.txt 等。

本节我们主要讲解一下如何使用 libjpeg 来实现一个 jpeg 图片的解码，这个在 libjpeg 源码里面：example.c，这个文件里面有简单的示范代码，在 libjpeg.txt 里面也有相关内容介绍。这里我们简要的给大家介绍一下，example.c 里面的标准解码流程如下（示例代码）：

```
//错误结构体
struct my_error_mgr
{
    struct jpeg_error_mgr pub; // jpeg_error_mgr 结构体，里面有很多错误处理函数
    jmp_buf setjmp_buffer; //返回给函数调用者
};

typedef struct my_error_mgr * my_error_ptr;
//JPEG 解码错误处理函数
METHODDEF(void) my_error_exit (j_common_ptr cinfo)
{
    my_error_ptr myerr = (my_error_ptr) cinfo->err; //指向 cinfo->err
    (*cinfo->err->output_message) (cinfo); //显示错误信息
    longjmp(myerr->setjmp_buffer, 1); //跳转到 setjmp 处
}

//JPEG 解码函数
GLOBAL(int) read_JPEG_file (char * filename)
```

```
{  
    struct jpeg_decompress_struct cinfo;  
    struct my_error_mgr jerr;      //错误处理结构体  
    FILE * infile;                //输入源文件  
    JSAMPARRAY buffer;             //输出缓存  
    int row_stride;               /* physical row width in output buffer */  
    if ((infile = fopen(filename, "rb")) == NULL)//尝试打开文件  
    {  
        fprintf(stderr, "can't open %s\n", filename);  
        return 0;  
    }  
    //第一步，设置错误管理，初始化 JPEG 解码对象  
    cinfo.err = jpeg_std_error(&jerr.pub);          //建立 JPEG 错误处理流程  
    jerr.pub.error_exit = my_error_exit;           //处理函数指向 my_error_exit  
    if (setjmp(jerr.setjmp_buffer)) //建立 my_error_exit 函数使用的返回上下文，当其他地方  
                                    //调用 longjmp 函数时，可以返回到这里进行错误处理  
    {  
        jpeg_destroy_decompress(&cinfo); //释放解码对象资源  
        fclose(infile); //关闭文件  
        return 0;  
    }  
    jpeg_create_decompress(&cinfo); //初始化解码对象 cinfo  
    //第二步，指定数据源（比如一个文件）  
    jpeg_stdio_src(&cinfo, infile);  
    //第三步，读取文件参数（通过 jpeg_read_header 函数）  
    (void) jpeg_read_header(&cinfo, TRUE); //可以忽略此返回值  
    //第四步，设置解码参数（这里使用 jpeg_read_header 确定的默认参数），故无处理。  
    //第五步，开始解码  
    (void) jpeg_start_decompress(&cinfo); //还是忽略返回值  
    //在读取数据之前，可以做一些处理，比如设定 LCD 窗口，设定 LCD 起始坐标等  
    row_stride = cinfo.output_width * cinfo.output_components; //确定一样有多少个样本  
    //确保 buffer 至少可以保存一行的样本数据，为其申请内存  
    buffer = (*cinfo.mem->alloc_sarray) ((j_common_ptr) &cinfo, JPOOL_IMAGE,  
                                         row_stride, 1);  
    //第六步，循环读取数据  
    while (cinfo.output_scanline < cinfo.output_height) //每次读一样，直到读完整个文件  
    {  
        (void) jpeg_read_scanlines(&cinfo, buffer, 1); //解码一行数据  
        put_scanline_someplace(buffer[0], row_stride); //将解码后的数据输出到某处  
    }  
    //第七步，结束解码  
    (void) jpeg_finish_decompress(&cinfo); //结束解码，忽略返回值  
    //第八步，释放解码对象资源
```

```

jpeg_destroy_decompress(&cinfo); //释放解码时申请的资源（大把内存）
fclose(infile); //关闭文件
return 1; //结束
}

```

以上代码，将一个 jpeg 解码分成了 8 个步骤，我们结合本例程代码简单讲解下这几个步骤。不过，我们先来看一下一个很重要的结构体数据类型：struct jpeg_decompress_struct，定义成 cinfo 变量，该变量保存着 jpeg 数据的详细信息，也保存着解码之后输出数据的详细信息。一般情况下，每次调用 libjpeg 库 API 的时候都需要把这个变量作为第一个参数传入。另外用户也可以通过修改该变量来修改 libjpeg 行为，比如输出数据格式，libjpeg 库可用的最大内存等等。

不过，在 STM32F4 里面使用，可不能按以示例代码这么来定义 cinfo 和 jerr 结构体，因为单片机堆栈有限，cinfo 和 jerr 都比较大（均超过 400 字节），很容易出现堆栈溢出的情况。在开发板源码，使用的是全局变量，而且用的是指针，通过内存管理分配。

接下来，开始看解码步骤，第一步是分配，并初始化解码对象结构体。这里做了两件事：1，错误管理，2，初始化解码对象。首先，错误管理使用 setjmp 和 longjmp 机制（不懂请百度）来实现类似 C++ 的异常处理功能，外部代码可以调用 longjmp 来跳转到 setjmp 位置，执行错误管理（释放内存，关闭文件等）。这里注册了一个 my_error_exit 函数，来执行错误退出处理，在本例程代码，还实现了一个函数：my_emit_message，输出警告信息，方便调试代码。然后，初始化解码对象 cinfo，就是通过 jpeg_create_decompress 函数实现。

第二步，指定数据源。示例代码用的是 jpeg_stdio_src 函数。本章代码，我们用另外一个函数实现：

```

//初始化 jpeg 解码数据源
static void jpeg_filerw_src_init(j_decompress_ptr cinfo)
{
    if (cinfo->src == NULL) /* first time for this JPEG object? */
    {
        cinfo->src = (struct jpeg_source_mgr *) (*cinfo->mem->alloc_small)((j_common_ptr)
            cinfo, JPOOL_PERMANENT, sizeof(struct jpeg_source_mgr));
    }
    cinfo->src->init_source = init_source;
    cinfo->src->fill_input_buffer = fill_input_buffer;
    cinfo->src->skip_input_data = skip_input_data;
    cinfo->src->resync_to_restart = jpeg_resync_to_restart; /* use default method */
    cinfo->src->term_source = term_source;
    cinfo->src->bytes_in_buffer = 0; /* forces fill_input_buffer on first read */
    cinfo->src->next_input_byte = NULL; /* until buffer loaded */
}

```

该函数里面，设置了 cinfo->src 的各个函数指针，用于获取外部数据。这里面重点是两个函数：fill_input_buffer 和 skip_input_data，前者用于填充数据给 libjpeg，后者用于跳过一定字节的数据。这两个函数请看本例程源码（在 mjpeg.c 里面）。

第三步，读取文件参数。通过 jpeg_read_header 函数实现，该函数将读取 JPEG 的很多参数，必须在解码前调用。

第四步，设置解码参数，示例代码没有做任何设置（使用默认值）。本章代码则做了设置，如下：

```
cinfo->dct_method = JDCT_IFAST;
cinfo->do_fancy_upsampling = 0;
```

这里，我们设置了使用快速整型 DCT 和 do_fancy_upsampling 的值为假 (0)，以提高解码速度。

第五步，开始解码。示例代码首先调用 jpeg_start_decompress 函数，然后计算样本输出 buffer 大小，并为其申请内存，为后续读取解码后的数据做准备。不过本章例程，我们为了提高速度，没有做这些处理了，我们直接修改底层函数：h2v1_merged_upsample 和 h2v2_merged_upsample (在 jdmerge.c 里面)，将输出的 RGB 数据直接转换成 RGB565，送给 LCD。然后，为了正确的输出到 LCD，我们在 jpeg_start_decompress 函数之后，加入如下代码：

```
LCD_Set_Window(imgoffx,imgoffy,cinfo->output_width,cinfo->output_height);
LCD_WriteRAM_Prepote(); //开始写入 GRAM
```

这两个函数，先设置好开窗大小 (即 jpeg 图片尺寸)，然后就发送准备写入 GRAM 指令。后续解码的时候，直接就在 h2v1_merged_upsample 和 h2v2_merged_upsample 里面丢数据给 LCD，实现 jpeg 解码输出到 LCD。

第六步，循环读取数据。通过 jpeg_read_scanlines 函数，循环解码并读取 jpeg 图片数据，实现 jpeg 解码。示例代码通过 put_scanline_somewhere 函数，输出到某个地方 (如 lcd，文件等)，本章例程则直接解码的时候就输出到 LCD 了，所以仅剩 jpeg_read_scanlines 函数，循环调用即可实现 jpeg → LCD 的操作。

第七步，解码结束。解码完成后，通过 jpeg_finish_decompress 函数，结束 jpeg 解码。

第八步，释放解码对象资源。在所有操作完成后，通过 jpeg_destroy_decompress，释放解码过程中用到的资源 (比如释放内存)。

这样，我们就完成了一张 jpeg 图片的解码。上面，我们简要列出了本章例程与 example.c 的异同，详细的代码，请大家参考光盘本例程源码 mjpeg.c.libjpeg 的使用，我们就介绍到这里。

最后，我们看看要实现 avi 视频文件的播放，主要有哪些步骤，如下：

1) 初始化各外设

要解码视频，相关外设肯定要先初始化好，比如：SDIO (驱动 SD 卡用)、I2S、DMA、WM8978、LCD 和按键等。这些具体初始化过程，在前面的例程都有介绍，大同小异，这里就不再细说了。

2) 读取 AVI 文件，并解析

要解码，得先读取 avi 文件，按 50.1.1 节的介绍，读取出音视频关键信息，音频参数：编码方式、采样率、位数和音频流类型码(01wb/00wb)等；视频参数：编码方式、帧间隔、图片尺寸和视频流类型码(00dc/01dc)等；共同的：数据流起始地址。有了这些参数，我们便可以初始化音视频解码，为后续解码做好准备。

3) 根据解析结果，设置相关参数

根据第 2 步解析的结果，设置 I2S 的音频采样率和位数，同时要让视频显示在 LCD 中间区域，得根据图片尺寸，设置 LCD 开窗时 x，y 方向的偏移量。

4) 读取数据流，开始解码

前面三步完成，就可以正式开始播放视频了。读取视频流数据 (movi 块)，根据类型码，执行音频/视频解码。对于音频数据(01wb/00wb)，本例程只支持未压缩的 PCM 数据，所以，直接填充到 DMA 缓冲区即可，由 DMA 循环发送给 WM8978，播放音频。对于视频数据(00dc/01dc)，本例程只支持 MJPG，通过 libjpeg 解码，所以将视频数据按前面所说的几个步骤解码即可。然后，利用定时器来控制帧间隔，以正常速度播放视频，从而实现音视频解码。

5) 解码完成，释放资源

最后在文件读取完后(或者出错了),需要释放申请的内存、恢复 LCD 窗口、关闭定时器、停止 I2S 播放音乐和关闭文件等一系列操作,等待下一次解码。

50.2 硬件设计

本章实验功能简介:开机后,先初始化各外设,然后检测字库是否存在,如果检测无问题,则开始播放 SD 卡 VIDEO 文件夹里面的视频 (.avi 格式)。注意:1, 在 SD 卡根目录必须建立一个 VIDEO 文件夹,并存放 AVI 视频(仅支持 MJPG 视频,音频必须是 PCM,且视频分辨率必须小于等于屏幕分辨率)在里面。2, 我们所需要的视频,可以通过: 狸窝全能视频转换器,转换后得到,具体步骤后续会讲到(50.4 节)。

视频播放时,LCD 上还会显示视频名字、当前视频编号、总视频数、声道数、音频采样率、帧率、播放时间和总时间等信息。KEY0 用于选择下一个视频,KEY2 用于选择上一个视频,KEY_UP 可以快进,KEY1 可以快退。DS0 还是用于指示程序运行状态(仅字库错误时)。

本实验用到的资源如下:

- 1) 指示灯 DS0
- 2) 4 个按键 (KEY_UP/KEY0/KEY1/KEY2)
- 3) 串口
- 4) TFTLCD 模块
- 5) SD 卡
- 6) SPI FLASH
- 7) WM8978
- 8) I2S2

这些前面都已介绍过。本实验,大家需要准备 1 个 SD 卡和一个耳机(或喇叭),分别插入 SD 卡接口和耳机接口(喇叭接 P1 接口),然后下载本实验就可以看视频了!

50.3 软件设计

打开本章实验工程目录可以看到,我们在工程根目录新建 MJPEG 文件夹,在该文件夹里面新建了 JPEG 文件夹,存放 libjpeg v9a 的相关代码,同时,在 MJPEG 文件夹里面新建了 avi.c、avi.h、mpeg.c 和 mjpeg.h 四个文件。然后,工程里面,新建了 MJPEG 分组,将需要用到的相关.c 文件添加到该分组下面,并将 MJPEG 和 JPEG 两个文件夹加入头文件包含路径。

我们还在 APP 文件夹下面新建了 videoplayer.c 和 videoplayer.h 两个文件,然后将 videoplayer.c 加入到工程的 APP 组下。

整个工程代码有点多,我们看看本实验新添加进来的代码,有哪些,如图 50.3.1 所示:

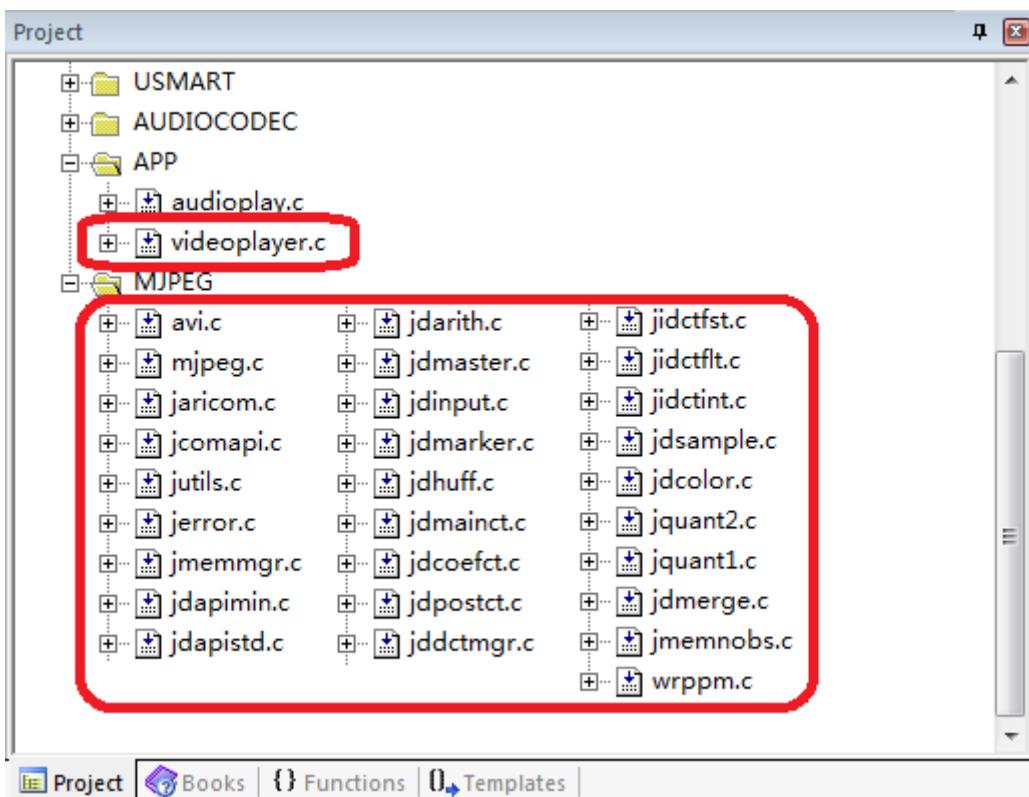


图 50.3.1 本实验新增代码

可见，本工程新增的代码是比较的多的，主要是 libjpeg 需要的文件挺多的。这里我们挑一部分重要代码给大家讲解。

首先是 avi.c 里面的几个函数，代码如下：

```

AVI_INFO avix;           //avi 文件相关信息
u8*const AVI_VIDS_FLAG_TBL[2]={"00dc","01dc"};//视频编码标志字符串,00dc/01dc
u8*const AVI_AUDS_FLAG_TBL[2]={"00wb","01wb"};//音频编码标志字符串,00wb/01wb
//avi 解码初始化
//buf:输入缓冲区
//size:缓冲区大小
//返回值:AVI_OK,avi 文件解析成功
//      其他,错误代码
AVISTATUS avi_init(u8 *buf,u16 size)
{
    u16 offset; u8 *tbuf;
    AVISTATUS res=AVI_OK;
    AVI_HEADER *aviheader; LIST_HEADER *listheader;
    AVIH_HEADER *avihheader;STRH_HEADER *strhheader;
    STRF_BMPHEADER *bmpheader;STRF_WAVHEADER *wavheader;
    tbuf=buf;
    aviheader=(AVI_HEADER*)buf;
    if(avihheader->RiffID!=AVI_RIFF_ID) return AVI_RIFF_ERR;      //RIFF ID 错误
    if(avihheader->AviID!=AVI_AVIS_ID) return AVI_AVIS_ERR;      //AVI ID 错误
}

```

```

buf+=sizeof(AVI_HEADER); //偏移
listheader=(LIST_HEADER*)(buf);
if(listheader->ListID!=AVI_LIST_ID) return AVI_LIST_ERR; //LIST ID 错误
if(listheader->ListType!=AVI_HDRL_ID) return AVI_HDRL_ERR; //HDRL ID 错误
buf+=sizeof(LIST_HEADER); //偏移
avihheader=(AVIH_HEADER*)(buf);
if(avihheader->BlockID!=AVI_AVIH_ID) return AVI_AVIH_ERR; //AVIH ID 错误
avix.SecPerFrame=avihheader->SecPerFrame; //得到帧间隔时间
avix.TotalFrame=avihheader->TotalFrame; //得到总帧数
buf+=avihheader->BlockSize+8; //偏移
listheader=(LIST_HEADER*)(buf);
if(listheader->ListID!=AVI_LIST_ID) return AVI_LIST_ERR; //LIST ID 错误
if(listheader->ListType!=AVI_STRL_ID) return AVI_STRL_ERR; //STRL ID 错误
strhheader=(STRH_HEADER*)(buf+12);
if(strhheader->BlockID!=AVI_STRH_ID) return AVI_STRH_ERR; //STRH ID 错误
if(strhheader->StreamType==AVI_VIDS_STREAM) ///视频帧在前
{
    if(strhheader->Handler!=AVI_FORMAT_MJPG) return AVI_FORMAT_ERR;//不支持
    avix.VideoFLAG=(u8*)AVI_VIDS_FLAG_TBL[0]; //视频流标记 "00dc"
    avix.AudioFLAG=(u8*)AVI_AUDS_FLAG_TBL[1]; //音频流标记 "01wb"
    bmpheader=(STRF_BMPHEADER*)(buf+12+strhheader->BlockSize+8); //strf
    if(bmpheader->BlockID!=AVI_STRF_ID) return AVI_STRF_ERR; //STRF ID 错误
    avix.Width=bmpheader->bmiHeader.Width;
    avix.Height=bmpheader->bmiHeader.Height;
    buf+=listheader->BlockSize+8; //偏移
    listheader=(LIST_HEADER*)(buf);
    if(listheader->ListID!=AVI_LIST_ID)//是不含有音频帧的视频文件
    {
        avix.SampleRate=0; //音频采样率
        avix.Channels=0; //音频通道数
        avix.AudioType=0; //音频格式
    }else
    {
        if(listheader->ListType!=AVI_STRL_ID) return AVI_STRL_ERR; //STRL ID 错误
        strhheader=(STRH_HEADER*)(buf+12);
        if(strhheader->BlockID!=AVI_STRH_ID) return AVI_STRH_ERR; //STRH 错误
        if(strhheader->StreamType!=AVI_AUDS_STREAM)
            return AVI_FORMAT_ERR; //格式错误
        wavheader=(STRF_WAVHEADER*)(buf+12+strhheader->BlockSize+8); //strf
        if(wavheader->BlockID!=AVI_STRF_ID) return AVI_STRF_ERR; //STRF 错误
        avix.SampleRate=wavheader->SampleRate; //音频采样率
        avix.Channels=wavheader->Channels; //音频通道数
        avix.AudioType=wavheader->FormatTag; //音频格式
    }
}

```

```

    }

}else if(strhheader->StreamType==AVI_AUDS_STREAM) //音频帧在前
{
    avix.VideoFLAG=(u8*)AVI_VIDS_FLAG_TBL[1]; //视频流标记 "01dc"
    avix.AudioFLAG=(u8*)AVI_AUDS_FLAG_TBL[0]; //音频流标记 "00wb"
    wavheader=(STRF_WAVHEADER*)(buf+12+strhheader->BlockSize+8); //strf
    if(wavheader->BlockID!=AVI_STRF_ID) return AVI_STRF_ERR; //STRF ID 错误
    avix.SampleRate=wavheader->SampleRate; //音频采样率
    avix.Channels=wavheader->Channels; //音频通道数
    avix.AudioType=wavheader->FormatTag; //音频格式
    buf+=listheader->BlockSize+8; //偏移
    listheader=(LIST_HEADER*)(buf);
    if(listheader->ListID!=AVI_LIST_ID) return AVI_LIST_ERR; //LIST ID 错误
    if(listheader->ListType!=AVI_STRL_ID) return AVI_STRL_ERR; //STRL ID 错误
    strhheader=(STRH_HEADER*)(buf+12);
    if(strhheader->BlockID!=AVI_STRH_ID) return AVI_STRH_ERR; //STRH ID 错误
    if(strhheader->StreamType!=AVI_VIDS_STREAM) return AVI_FORMAT_ERR;
    bmpheader=(STRF_BMPHEADER*)(buf+12+strhheader->BlockSize+8); //strf
    if(bmpheader->BlockID!=AVI_STRF_ID) return AVI_STRF_ERR; //STRF ID 错误
    if(bmpheader->bmiHeader.Compression!=AVI_FORMAT_MJPG)
        return AVI_FORMAT_ERR; //格式错误
    avix.Width=bmpheader->bmiHeader.Width;
    avix.Height=bmpheader->bmiHeader.Height;
}

offset=avi_srarch_id(tbuf,size,"movi"); //查找 movi ID
if(offset==0) return AVI_MOVI_ERR; //MOVI ID 错误
if(avix.SampleRate)//有音频流,才查找
{
    tbuf+=offset;
    offset=avi_srarch_id(tbuf,size,avix.AudioFLAG); //查找音频流标记
    if(offset==0) return AVI_STREAM_ERR; //流错误
    tbuf+=offset+4;
    avix.AudioBufSize=*((u16*)tbuf); //得到音频流 buf 大小.
}
return res;
}

//查找 ID
//buf:待查缓存区
//size:缓存大小
//id:要查找的 id,必须是 4 字节长度
//返回值:0,查找失败,其他:movi ID 偏移量
u16 avi_srarch_id(u8* buf,u16 size,u8 *id)
{

```

```

u16 i;
size-=4;
for(i=0;i<size;i++)
{
    if(buff[i]==id[0])
        if(buff[i+1]==id[1])
            if(buff[i+2]==id[2])
                if(buff[i+3]==id[3])return i;//找到"id"所在的位置
}
return 0;
}

//得到 stream 流信息
//buf:流开始地址(必须是 01wb/00wb/01dc/00dc 开头)
AVISTATUS avi_get_streaminfo(u8* buf)
{
    avix.StreamID=MAKEWORD(buf+2);      //得到流类型
    avix.StreamSize=MAKEDWORD(buf+4);   //得到流大小
    if(avix.StreamSize%2)avix.StreamSize++; //奇数加 1(avix.StreamSize,必须是偶数)
    if(avix.StreamID==AVI_VIDS_FLAG||avix.StreamID==AVI_AUDS_FLAG)
        return AVI_OK;
    return AVI_STREAM_ERR;
}

```

这里三个函数，其中 `avi_ini` 用于解析 AVI 文件，获取音视频流数据的详细信息，为后续解码做准备。而 `avi_srarch_id` 用于查找某个 ID，可以是 4 个字节长度的 ID，比如 00dc，01wb，movi 之类的，在解析数据以及快进快退的时候，有用到。`avi_get_streaminfo` 函数，则是用来获取当前数据流信息，重点是取得流类型和流大小，方便解码和读取下一个数据流。

接下来，我们看 `mjpeg.c` 里面的几个函数，代码如下：

```

//mjpeg 解码初始化
//offx,offy:x,y 方向的偏移
//返回值:0,成功; 1,失败
u8 mjpegdec_init(u16 offx,u16 offy)
{
    cinfo=mymalloc(SRAMCCM,sizeof(struct jpeg_decompress_struct));
    jerr=mymalloc(SRAMCCM,sizeof(struct my_error_mgr));
    jmembuf=mymalloc(SRAMCCM,MJPEG_MAX_MALLOC_SIZE);//解码内存池申请
    if(cinfo==0||jerr==0||jmembuf==0){ mjpegdec_free();return 1;}
    //保存图像在 x,y 方向的偏移量
    imgoffx=offx; imgoffy=offy;
    return 0;
}
//mjpeg 结束,释放内存
void mjpegdec_free(void)
{

```

```
myfree(SRAMCCM,cinfo);
myfree(SRAMCCM,jerr);
myfree(SRAMCCM,jmembuf);

}

//解码一副 JPEG 图片
//buf:jpeg 数据流数组    bsize:数组大小
//返回值:0,成功          其他,错误
u8 mjpegdec_decode(u8* buf,u32 bsize)
{
    JSAMPARRAY buffer;
    if(bsize==0) return 1;
    jpegbuf=buf; jbufsize=bsize;
    jmempos=0;//MJEPG 解码,重新从 0 开始分配内存
    cinfo->err=jpeg_std_error(&jerr->pub);
    jerr->pub.error_exit = my_error_exit;
    jerr->pub.emit_message = my_emit_message;
    //if(bsize>20*1024)printf("s:%d\r\n",bsize);
    if (setjmp(jerr->setjmp_buffer)) //错误处理
    {
        jpeg_abort_decompress(cinfo);
        jpeg_destroy_decompress(cinfo);
        return 2;
    }
    jpeg_create_decompress(cinfo);
    jpeg_filerw_src_init(cinfo);
    jpeg_read_header(cinfo, TRUE);
    cinfo->dct_method = JDCT_IFAST;
    cinfo->do_fancy_upsampling = 0;
    jpeg_start_decompress(cinfo);
    LCD_Set_Window(imgoffx,imgoffy,cinfo->output_width,cinfo->output_height);
    LCD_WriteRAM_Prepae();           //开始写入 GRAM
    while (cinfo->output_scanline < cinfo->output_height)
    {
        jpeg_read_scanlines(cinfo, buffer, 1);
    }
    LCD_Set_Window(0,0,lcddev.width,lcddev.height);//恢复窗口
    jpeg_finish_decompress(cinfo);
    jpeg_destroy_decompress(cinfo);
    return 0;
}
```

其中，`mjpegdec_init` 函数，用于初始化 jpeg 解码，主要是申请内存，然后确定视频在液晶上面的偏移（以让视频显示在 LCD 中央）。`mjpegdec_free` 函数，用于释放内存，解码结束后调用。`mjpegdec_decode` 函数，是解码 jpeg 的主要函数，通过前面 50.1.2 节介绍的步骤进行解

码，该函数的参数 buf 指向内存里面的一帧 jpeg 数据，bsize 就是数据大小。

接下来，我们看 videoplayer.c 里面 video_play_mjpeg 函数，代码如下：

```
//播放一个 mjpeg 文件
//pname:文件名
//返回值： KEY0_PRES:下一曲 KEY1_PRES:上一曲
//其他:错误
u8 video_play_mjpeg(u8 *pname)
{
    u8* framebuf; //视频解码 buf
    u8* pbuf;      //buf 指针
    FIL *favi;
    u8 res=0; u16 offset=0;
    u32 nr;u8 key; u8 i2ssavebuf;
    i2sbuf[0]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
    i2sbuf[1]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
    i2sbuf[2]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
    i2sbuf[3]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
    framebuf=mymalloc(SRAMIN,AVI_VIDEO_BUF_SIZE); //申请视频 buf
    favi=(FIL*)mymalloc(SRAMIN,sizeof(FIL));           //申请 favi 内存
    memset(i2sbuf[0],0,AVI_AUDIO_BUF_SIZE);
    memset(i2sbuf[1],0,AVI_AUDIO_BUF_SIZE);
    memset(i2sbuf[2],0,AVI_AUDIO_BUF_SIZE);
    memset(i2sbuf[3],0,AVI_AUDIO_BUF_SIZE);
    if(i2sbuf[3]==NULL||framebuf==NULL||favi==NULL) res=0xFF;
    while(res==0)
    {
        res=f_open(favi,(char *)pname,FA_READ);
        if(res==0)
        {
            pbuf=framebuf;
            res=f_read(favi,pbuf,AVI_VIDEO_BUF_SIZE,&nr); //开始读取
            if(res) {printf("fread error:%d\r\n",res);break;}
            //开始 avi 解析
            res=avi_init(pbuf,AVI_VIDEO_BUF_SIZE); //avi 解析
            if(res){ printf("avi err:%d\r\n",res); break;}
            video_info_show(&avix);
            TIM6_Int_Init(avix.SecPerFrame/100-1,8400-1); //10Khz 计数频率,加 1 是 100us
            offset=avi_srarch_id(pbuf,AVI_VIDEO_BUF_SIZE,"movi");//寻找 movi ID
            avi_get_streaminfo(pbuf+offset+4); //获取流信息
            f_lseek(favi,offset+12); //跳过标志 ID,读地址偏移到流数据开始处
            res=mjpegdec_init((lcddev.width-avix.Width)/2, 110+(lcddev.height-110-
                avix.Height)/2); //初始化 JPG 解码
        //JPG 解码初始化
    }
}
```

```
if(avix.SampleRate)          //有音频信息,才初始化
{
    WM8978_I2S_Cfg(2,0); //飞利浦标准,16 位数据长度

    I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,
              I2S_DataFormat_16bextended);
              //飞利浦标准,主机发送,时钟低有效,16 位扩展帧
    I2S2_SampleRate_Set(avix.SampleRate); //设置采样率
    I2S2_TX_DMA_Init(i2sbuf[1],i2sbuf[2],avix.AudioBufSize/2); //配置 DMA
    i2s_tx_callback=audio_i2s_dma_callback; //回调函数 I2S_DMA_Callback
    i2splaybuf=0; i2ssavebuf=0;
    I2S_Play_Start(); //开启 I2S 播放
}
while(1)//播放循环
{
    if(avix.StreamID==AVI_VIDS_FLAG)//视频流
    {
        pbuf=framebuf;
        f_read(favi,pbuf,avix.StreamSize+8,&nr); //读整帧+下个数据流 ID
        res=mjpegdec_decode(pbuf,avix.StreamSize);
        if(res) printf("decode error!\r\n");
        while(frameup==0); //等待时间到达(在 TIM6 的中断里面设置为 1)
        frameup=0; //标志清零
        frame++;
    }
    else //音频流
    {
        video_time_show(favi,&avix); //显示当前播放时间
        i2ssavebuf++;
        if(i2ssavebuf>3)i2ssavebuf=0;
        do
        {
            nr=i2splaybuf;
            if(nr)nr--;
            else nr=3;
        }while(i2ssavebuf==nr); //碰撞等待.
        f_read(favi,i2sbuf[i2ssavebuf],avix.StreamSize+8,&nr); //填充 i2sbuf
        pbuf=i2sbuf[i2ssavebuf];
    }
    key=KEY_Scan(0);
    if(key==KEY0_PRES||key==KEY2_PRES) { res=key; break; } //切换
    else if(key==KEY1_PRES||key==WKUP_PRES)
    {
        I2S_Play_Stop(); //关闭音频
    }
}
```

```

        video_seek(favi,&avix,framebuf);
        pbuf=framebuf;
        I2S_Play_Start(); //开启 DMA 播放
    }
    if(avi_get_streaminfo(pbuf+avix.StreamSize))//读取下一帧 流标志
    {
        printf("frame error \r\n");
        res=KEY0_PRES;
        break;
    }
}
I2S_Play_Stop(); //关闭音频
TIM6->CR1&=~(1<<0); //关闭定时器 6
LCD_Set_Window(0,0	lcddev.width, lcddev.height); //恢复窗口
mjpegdec_free(); //释放内存
f_close(favi);
}
}
myfree(SRAMIN,i2sbuf[0]); myfree(SRAMIN,i2sbuf[1]);
myfree(SRAMIN,i2sbuf[2]); myfree(SRAMIN,i2sbuf[3]);
myfree(SRAMIN,framebuf); myfree(SRAMIN,favi);
return res;
}

```

该函数用来播放一个 avi 视频文件 (mjpg 编码), 解码过程就是根据前面我们在 50.1.2 节最后所介绍的步骤进行, 不过在这里, 我们的音频播放用了 4 个 buf, 以提高解码的流畅度。

最后, 我们看看主函数代码:

```

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200

    LED_Init(); //初始化 LED
    usmart_dev.init(84); //初始化 USMART
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    W25QXX_Init(); //初始化 W25Q128
    WM8978_Init(); //初始化 WM8978

    WM8978_ADDA_Cfg(1,0); //开启 DAC
    WM8978_Input_Cfg(0,0,0); //关闭输入通道
    WM8978_Output_Cfg(1,0); //开启 DAC 输出
}

```

```
WM8978_HPvol_Set(40,40);
WM8978_SPKvol_Set(60);
TIM3_Int_Init(10000-1,8400-1); //10Khz 计数,1 秒钟中断一次

my_mem_init(SRAMIN);          //初始化内部内存池
my_mem_init(SRAMCCM);         //初始化 CCM 内存池
exfun_init();                  //为 fatfs 相关变量申请内存
f_mount(fs[0],"0:",1);        //挂载 SD 卡
POINT_COLOR=RED;
while(font_init())             //检查字库
{
    LCD_ShowString(30,50,200,16,16,"Font Error!");
    delay_ms(200);
    LCD_Fill(30,50,240,66,WHITE); //清除显示
    delay_ms(200);
    LED0=!LED0;
}
POINT_COLOR=RED;
Show_Str(60,50,200,16,"Explorer STM32 开发板",16,0);
Show_Str(60,70,200,16,"视频播放器实验",16,0);
Show_Str(60,90,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,110,200,16,"2014 年 7 月 1 日",16,0);
Show_Str(60,130,200,16,"KEY0:NEXT    KEY2:PREV",16,0);
Show_Str(60,150,200,16,"KEY_UP:FF    KEY1: REW",16,0);
delay_ms(1500);
while(1)
{
    video_play();
}
}
```

该函数代码同上一章的 main 函数代码几乎一样，十分简单，我们就不再多说了。

最后，因为视频解码需要用到比较多的堆栈，所以需要修改 startup_stm32f40_41xxx.s 里面的堆栈大小，将原来的 0x00000400 设置为 0x00000800，如下：

```
Stack_Size      EQU      0x00000800
```

同时，为了提高速度，我们对编译器进行设置，选择使用-O2 优化，从而优化代码，提高速度（但调试效果不好，建议调试时设置为-O0），编译器设置如图 50.3.2 所示：

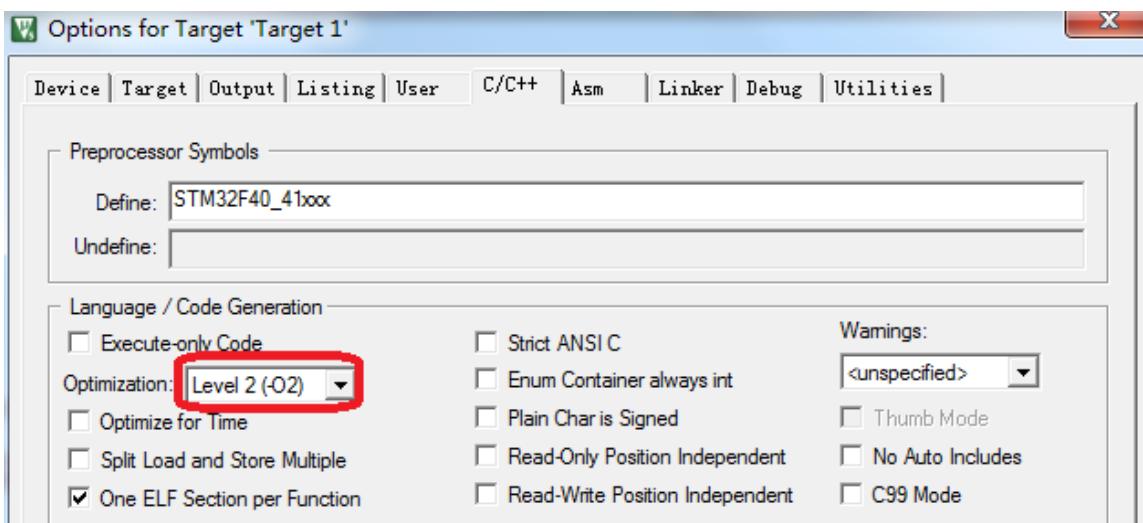


图 50.3.2 编译器优化设置

设置完后，重新编译即可。至此，本实验的软件设计部分结束。

50.4 下载验证

本章，我们例程仅支持 MJPG 编码的 avi 格式视频，且音频必须是 PCM 格式，另外视频分辨率不能大于 LCD 分辨率。要满足这些要求，现成的 avi 文件是很难找到的，所以我们需要用软件，将通用视频（任何视频都可以）转换为我们需要的格式，这里我们通过：狸窝全能视频转换器，这款软件来实现（路径：光盘：6，软件资料→软件→视频转换软件→狸窝全能视频转换器.exe）。安装完后，打开，然后进行相关设置，软件设置如图 50.4.1 和 50.4.2 所示：



图 50.4.1 软件启动界面和设置



图 50.4.2 高级设置

首先，如图 50.4.1 所示，点击 1 处，添加视频，找到你要转换的视频，添加进来。有的视频可能有独立字幕，比如我们打开的这个视频就有，所以在 2 处选择下字幕（如果没有的，可以忽略此步）。然后在 3 处，点击▼图标，选择预制方案：AVI-Audio-Video Interleaved (*.avi)，即生成 .avi 文件，然后点击 4 处的高级设置按钮，进入 50.4.2 所示的界面，设置详细参数如下：

视频编码器：选择 MJPEG。本例程仅支持 MJPG 视频解码，所以选择这个编码器。

视频尺寸：480x272。这里得根据所用 LCD 分辨率来选择，我们用 480*800 的 4.3 寸电容屏模块，所以，这里最大可以设置：480x272。PS：如果是 2.8 屏，最大宽度只能是 240）。

比特率：1000。这里设置越大，视频质量越好，解码就越慢（可能会卡），我们设置为 1000，可以得到比较好的视频质量，同时也不怎么会卡。

帧率：10。即每秒钟 10 帧，对于 480*272 的视频，本例程最高就只能播放 10 帧左右的视频，如果要想提高帧率，有几个办法：1，降低分辨率；2，降低比特率；3，降低音频采样率。

音频编码器：PCMS16LE。本例程只支持 PCM 音频，所以选择音频编码器为这个。

采样率：这里设置为 11025，即 11.025Khz 的采样率。这里越高，声音质量越好，不过，转换后的文件就越大，而且视频可能会卡。

其他设置，采用默认的即可。设置完以后，点击确定，即可完成设置。

点击图 50.4.1 的 5 处的文件夹图标，设置转换后视频的输出路径，这里我们设置到了桌面，这样转换后的视频，会保存在桌面。最后，点击图中 6 处的按钮，即可开始转换了，如图 50.4.3 所示：



图 50.4.3 正在转换

等转换完成后，将转换后的.avi 文件，拷贝到 SD 卡→VIDEO 文件夹下，然后插入开发板的 SD 卡接口，就可以开始测试本章例程了。

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 开发板上，程序先检测字库，然后检测 SD 卡的 VIDEO 文件夹，并查找 avi 视频文件，在找到有效视频文件后，便开始播放视频，如图 50.4.4 所示：



图 50.4.4 视频播放中

可以看到，屏幕显示了文件名、索引、声道数、采样率、帧率和播放时间等参数。然后，我们按 KEY0/KEY2，可以切换到下一个/上一个视频，按 KEY_UP/KEY1，可以快进/快退。

至此，本例程介绍就结束了。本实验，我们在 ALIENTEK STM32F4 探索者开发板上实现了视频播放，体现了 STM32F4 强大的处理能力。

附本实验测试结果（视频比特率：1000，音频均为：11025，立体声）

对 240*160/240*180 分辨率，可达 30 帧

对 320*240 分辨率，可达 20 帧

对 480*272 分辨率，可达 10 帧

最后提醒大家，转换的视频分辨率，一定要根据自己的 LCD 设置，不能超过 LCD 的尺寸!!

否则无法播放（可能只听到声音，看不到图像）。

第五十一章 FPU 测试(Julia 分形)实验

本章，我们将向大家介绍如何开启 STM32F4 的硬件 FPU，并对比使用硬件 FPU 和不使用硬件 FPU 的速度差别，以体现硬件 FPU 的优势。本章分为如下几个部：

- 51.1 FPU&Julia 分形简介
- 51.2 硬件设计
- 51.3 软件设计
- 51.4 下载验证

51.1 FPU&Julia 分形简介

本节将分别介绍 STM32F4 的 FPU 和 Julia 分形。

51.1.1 FPU 简介

FPU 即浮点运算单元（Float Point Unit）。浮点运算，对于定点 CPU（没有 FPU 的 CPU）来说必须要按照 IEEE-754 标准的算法来完成运算，是相当耗费时间的。而对于有 FPU 的 CPU 来说，浮点运算则只是几条指令的事情，速度相当快。

STM32F4 属于 Cortex M4F 架构，带有 32 位单精度硬件 FPU，支持浮点指令集，相对于 Cortex M0 和 Cortex M3 等，高出数十倍甚至上百倍的运算性能。

STM32F4 硬件上要开启 FPU 是很简单的，通过一个叫：协处理器控制寄存器（CPACR）的寄存器设置即可开启 STM32F4 的硬件 FPU，该寄存器各位描述如图 51.1.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16		
Reserved								CP11	CP10	Reserved							
								rw	rw								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Reserved																	

图 51.1.1.1 协处理器控制寄存器（CPACR）各位描述

这里我们就是要设置 CP11 和 CP10 这 4 个位，复位后，这 4 个位的值都为 0，此时禁止访问协处理器（禁止了硬件 FPU），我们将这 4 个位都设置为 1，即可完全访问协处理器（开启硬件 FPU），此时便可以使用 STM32F4 内置的硬件 FPU 了。CPACR 寄存器这 4 个位的设置，我们在 system_stm32f4xx_c 文件里面开启，代码如下：

```
void SystemInit(void)
{
    /* FPU settings -----*/
    #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
        SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2)); /* set CP10 and CP11 Full Access */
    #endif
    .....//省略部分代码
}
```

此部分代码是系统初始化函数的部分内容，功能就是设置 CPACR 寄存器的 20~23 位为 1，以开启 STM32F4 的硬件 FPU 功能。从程序可以看出，只要我们定义了全局宏定义标识符 __FPU_PRESENT 以及 __FPU_USED 为 1，那么就可以开启硬件 FPU。其中宏定义标识符 __FPU_PRESENT 用来确定处理器是否带 FPU 功能，标识符 __FPU_USED 用来确定是否开启

FPU 功能。

实际上，因为 F4 是带 FPU 功能的，所以在我们的 `stm32f4xx.h` 头文件里面，我们默认是定义了 `_FPU_PRESENT` 为 1。大家可以打开文件搜索即可找到下面一行代码：

```
#define _FPU_PRESENT 1
```

但是，仅仅只是说明处理器有 FPU 是不够的，我们还需要开启 FPU 功能。开启 FPU 有两种方法，第一种是直接在头文件 `STM32f4xx.h` 中定义宏定义标识符 `_FPU_USED` 的值为 1。也可以直接在 MDK 编译器上面设置，我们在 MDK5 编译器里面，点击  按钮，然后在 Target 选项卡里面，设置 Floating Point Hardware 为 Use FPU，如图 51.1.1.2 所示：

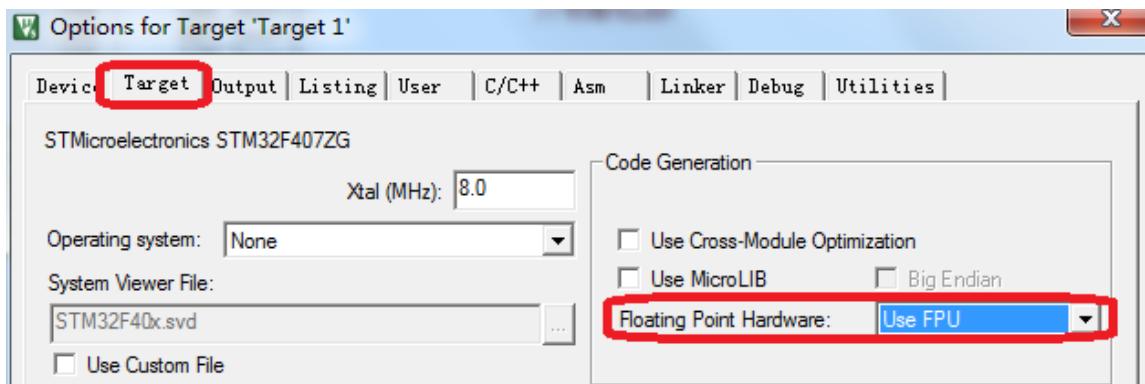


图 51.1.1.2 编译器开启硬件 FPU 选型

经过这个设置，编译器会自动加入标识符 `_FPU_USED` 为 1。这样遇到浮点运算就会使用硬件 FPU 相关指令，执行浮点运算，从而大大减少计算时间。

最后，总结下 STM32F4 硬件 FPU 使用的要点：

- 1, 设置 CPACR 寄存器 bit20~23 为 1，使能硬件 FPU。
- 2, MDK 编译器 Code Generation 里面设置：Use FPU。

经过这两步设置，我们的编写的浮点运算代码，即可使用 STM32F4 的硬件 FPU 了，可以大大加快浮点运算速度。

51.1.2 Julia 分形简介

Julia 分形即 Julia 集，它最早由法国数学家 Gaston Julia 发现，因此命名为 Julia（朱利亚）集。Julia 集合的生成算法非常简单：对于复平面的每个点，我们计算一个定义序列的发散速度。该序列的 Julia 集计算公式为：

$$z_{n+1} = z_n^2 + c$$

针对复平面的每个 $x + i.y$ 点，我们用 $c = c_x + i.c_y$ 计算该序列：

$$\begin{aligned} x_{n+1} + i.y_{n+1} &= x_n^2 - y_n^2 + 2.i.x_n.y_n + c_x + i.c_y \\ x_{n+1} &= x_n^2 - y_n^2 + c_x \quad \text{且} \quad y_{n+1} = 2.x_n.y_n + c_y \end{aligned}$$

一旦计算出的复值超出给定圆的范围（数值大小大于圆半径），序列便会发散，达到此限时完成的迭代次数与该点相关。随后将该值转换为颜色，以图形方式显示复平面上各个点的分散速度。

经过给定的迭代次数后，若产生的复值保持在圆范围内，则计算过程停止，并且序列也不发散，本例程生成 Julia 分形图片的代码如下：

```
#define ITERATION 128 //迭代次数
#define REAL_CONSTANT 0.285f //实部常量
```

```
#define      IMG_CONSTANT      0.01f      //虚部常量

//产生 Julia 分形图形
//size_x,size_y:屏幕 x,y 方向的尺寸
//offset_x,offset_y:屏幕 x,y 方向的偏移
//zoom:缩放因子
void GenerateJulia_fpu(u16 size_x,u16 size_y,u16 offset_x,u16 offset_y,u16 zoom)
{
    u8 i; u16 x,y;
    float tmp1,tmp2;
    float num_real,num_img;
    float radius;
    for(y=0;y<size_y;y++)
    {
        for(x=0;x<size_x;x++)
        {
            num_real=y-offset_y;
            num_real=num_real/zoom;
            num_img=x-offset_x;
            num_img=num_img/zoom;
            i=0;
            radius=0;
            while((i<ITERATION-1)&&(radius<4))
            {
                tmp1=num_real*num_real;
                tmp2=num_img*num_img;
                num_img=2*num_real*num_img+IMG_CONSTANT;
                num_real=tmp1-tmp2+REAL_CONSTANT;
                radius=tmp1+tmp2;
                i++;
            }
            LCD->LCD_RAM=color_map[i];//绘制到屏幕
        }
    }
}
```

这种算法非常有效地展示了 FPU 的优势：无需修改代码，只需在编译阶段激活或禁止 FPU（在 MDK Code Generation 里面设置：Use FPU/Not Used），即可测试使用硬件 FPU 和不使用硬件 FPU 的差距。

51.2 硬件设计

本章实验功能简介：开机后，根据迭代次数生成颜色表（RGB565），然后计算 Julia 分形，并显示到 LCD 上面。同时，程序开启了定时器 3，用于统计一帧所要的时间（ms），在一帧 Julia 分形图片显示完成后，程序会显示运行时间、当前是否使用 FPU 和缩放因子（zoom）等

信息，方便观察对比。**KEY0/KEY2** 用于调节缩放因子，**KEY_UP** 用于设置自动缩放，还是手动缩放。**DS0** 用于提示程序运行状况。

本实验用到的资源如下：

- 1, 指示灯 DS0
 - 2, 三个按键 (KEY_UP/KEY0/KEY2)
 - 3, 串口
 - 4, TFTLCD 模块
- 这些前面都已介绍过。

51.3 软件设计

本章代码，分成两个工程：

- 1, 实验 46_1 FPU 测试(Julia 分形)实验_开启硬件 FPU
- 2, 实验 46_2 FPU 测试(Julia 分形)实验_关闭硬件 FPU

这两个工程的代码一模一样，只是前者使用硬件 FPU 计算 Julia 分形集 (MDK 参考图 51.1.1.2 设置 Use FPU)，后者使用 IEEE-754 标准计算 Julia 分形集 (MDK 设置参考图 51.1.1.2 设置不使用 FPU)。由于两个工程代码一模一样，我们这里仅介绍其中一个：实验 46_1 FPU 测试(Julia 分形)实验_开启硬件 FPU。

本章代码，我们在 TFTLCD 显示实验的基础上修改，打开 TFTLCD 显示实验的工程，由于要统计帧时间和按键设置，所以在 HARDWARE 组下加入 timer.c 和 key.c 两个文件。

本章不需要添加其他.c 文件，所有代码均在 main.c 里面实现，整个代码如下：

```
//FPU 模式提示
#ifndef __FPU_USED__ == 1
#define SCORE_FPU_MODE "FPU On"
#else
#define SCORE_FPU_MODE "FPU Off"
#endif

#define ITERATION 128 //迭代次数
#define REAL_CONSTANT 0.285f //实部常量
#define IMG_CONSTANT 0.01f //虚部常量

//颜色表
u16 color_map[ITERATION];
//缩放因子列表
const u16 zoom_ratio[] =
{
    120, 110, 100, 150, 200, 275, 350, 450,
    600, 800, 1000, 1200, 1500, 2000, 1500,
    1200, 1000, 800, 600, 450, 350, 275, 200,
    150, 100, 110,
};

//初始化颜色表
//clut:颜色表指针
void InitCLUT(u16 * clut)
{
```

```
u32 i=0x00;
u16 red=0,green=0,blue=0;
for(i=0;i<ITERATION;i++)//产生颜色表
{
    //产生 RGB 颜色值
    red=(i*8*256/ITERATION)%256;
    green=(i*6*256/ITERATION)%256;
    blue=(i*4*256 /ITERATION)%256;
    //将 RGB888,转换为 RGB565
    red=red>>3;
    red=red<<11;
    green=green>>2;
    green=green<<5;
    blue=blue>>3;
    clut[i]=red+green+blue;
}
}

//产生 Julia 分形图形
//size_x,size_y:屏幕 x,y 方向的尺寸
//offset_x,offset_y:屏幕 x,y 方向的偏移
//zoom:缩放因子
void GenerateJulia_fpu(u16 size_x,u16 size_y,u16 offset_x,u16 offset_y,u16 zoom)
{
    .....//代码省略, 详见 51.1.2 节
}

u8 timeout;
int main(void)
{
    u8 key; u8 i=0; u8 autorun=0; u8 buf[50];
    float time;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    KEY_Init(); //初始化按键
    LCD_Init(); //初始化 LCD
    TIM3_Int_Init(65535,8400-1);//10Khz 计数频率,最大计时 6.5 秒超出
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"FPU TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/7/2");
    LCD_ShowString(30,130,200,16,16,"KEY0:+      KEY2:-"); //显示提示信息
```

```
LCD_ShowString(30,150,200,16,16,"KEY_UP:AUTO/MANUL"); //显示提示信息
delay_ms(1200);
POINT_COLOR=BLUE; //设置字体为蓝色
InitCLUT(color_map); //初始化颜色表
while(1)
{
    key=KEY_Scan(0);
    switch(key)
    {
        case KEY0_PRES:
            i++;
            if(i>sizeof(zoom_ratio)/2-1)i=0;//限制范围
            break;
        case KEY2_PRES:
            if(i)i--;
            else i[sizeof(zoom_ratio)/2-1];
            break;
        case WKUP_PRES: autorun=!autorun; break;//自动/手动
    }
    if(autorun==1)//自动时,自动设置缩放因子
    {
        i++;
        if(i>sizeof(zoom_ratio)/2-1)i=0;//限制范围
    }
    LCD_Set_Window(0,0	lcddev.width, lcddev.height);//设置窗口
    LCD_WriteRAM_Prepare();
    TIM3->CNT=0;//重设 TIM3 定时器的计数器值
    timeout=0;
    GenerateJulia_fpu(lcddev.width, lcddev.height, lcddev.width/2, lcddev.height/2,
                       zoom_ratio[i]);
    time=TIM3->CNT+(u32)timeout*65536;
    sprintf((char*)buf,"%s: zoom:%d  runtime:%0.1fms\r\n",SCORE_FPU_MODE,
            zoom_ratio[i],time/10);
    LCD_ShowString(5, lcddev.height-5-12, lcddev.width-5, 12, 12, buf);//显示运行情况
    printf("%s",buf);//输出到串口
    LED0=!LED0;
}
}
```

这里面，总共 3 个函数：InitCLUT、GenerateJulia_fpu 和 main 函数。

InitCLUT 函数，该函数用于初始化颜色表，该函数根据迭代次数 (ITERATION) 计算出颜色表，这些颜色值将显示在 TFTLCD 上。

GenerateJulia_fpu 函数，该函数根据给定的条件计算 Julia 分形集，当迭代次数大于等于 ITERATION 或者半径大于等于 4 时，结束迭代，并在 TFTLCD 上面显示迭代次数对应的颜色

值，从而得到漂亮的 Julia 分形图。我们可以通过修改 REAL_CONSTANT 和 IMG_CONSTANT 这两个常量的值来得到不同的 Julia 分形图。

main 函数，完成我们在 51.2 节所介绍的实验功能，代码比较简单。这里我们用到一个缩放因子表：zoom_ratio，里面存储了一些不同的缩放因子，方便演示效果。

最后，为了提高速度，同上一章一样，我们在 MDK 里面选择使用-O2 优化，优化代码速度，本例程代码就介绍到这里。

再次提醒大家：本例程两个代码（实验 46_1 和实验 46_2）程序是完全一模一样的，他们的区别就是 MDK→Options for Target ‘Target1’ →Target 选项卡→Floating Point Hardware 的设置不一样，当设置 Use FPU 时，使用硬件 FPU；当设置 Not Used 时，不使用硬件 FPU。分别下载这两个代码，通过屏幕显示的 runtime 时间，即可看出速度上的区别。

51.4 下载验证

代码编译成功之后，下载本例程任意一个代码（这里以 46_1 为例）到 ALIENTEK 探索者 STM32F4 开发板上，可以看到 LCD 显示 Julia 分形图，并显示相关参数，如图 51.4.1 所示：

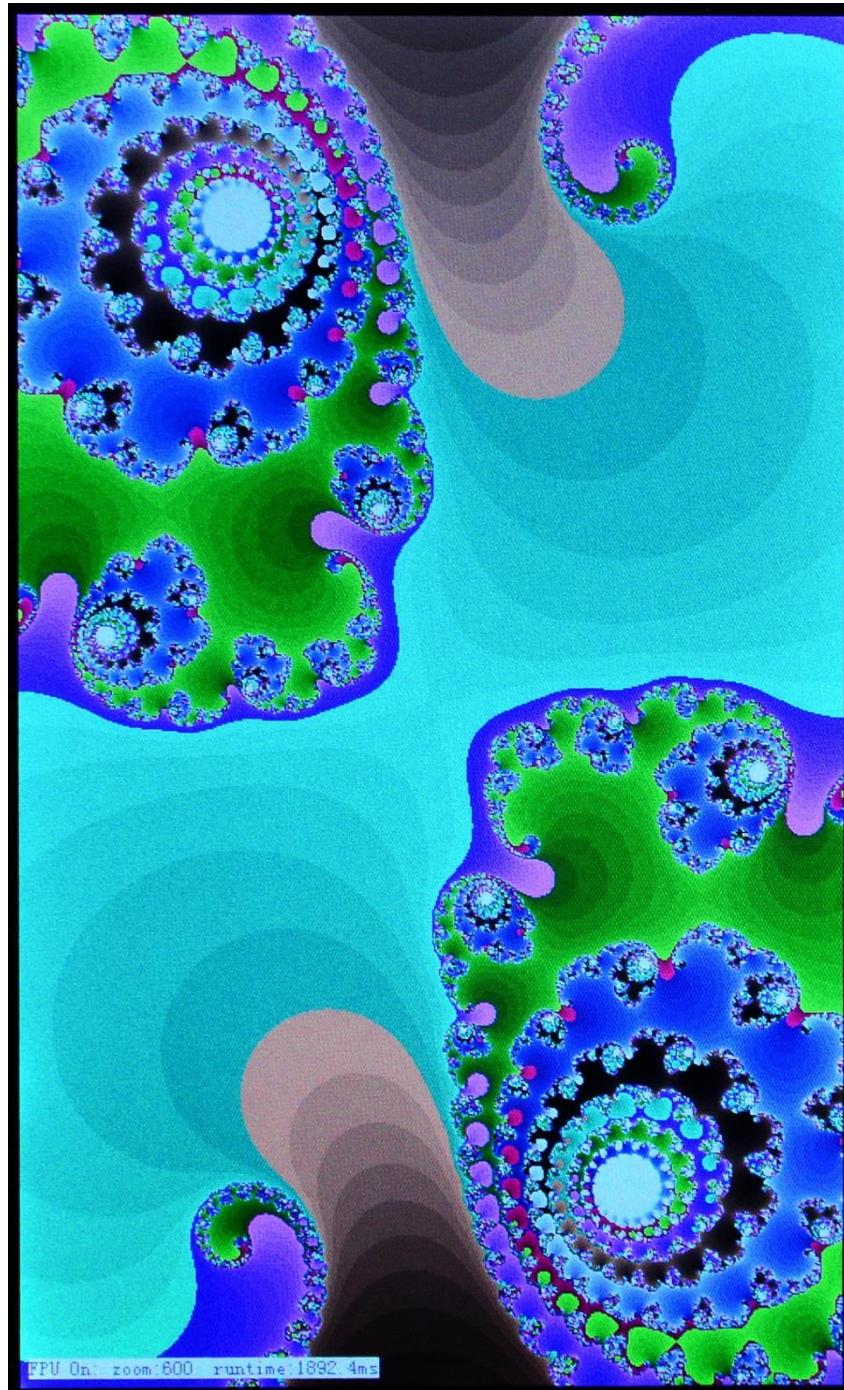


图 51.4.1 Julia 分形显示效果

实验 46_1 是开启了硬件 FPU 的，所以显示 Julia 分形图片速度比较快。如果下载实验 46_2，同样的缩放因子，会比实验 46_1 慢 9 倍左右，这与 ST 官方给出的 17 倍有点差距，这是因为我们没有选择：Use MicroLIB（还是在 Target 选项卡设置），如果都勾选这个，则会发现：使用硬件 FPU 的例程（实验 46_1）时间基本没变化，而不使用硬件 FPU 的例程（实验 46_2）则速度变慢了很多，这样，两者相差差不多就是 17 倍了。

因此可以看出，使用硬件 FPU 和不使用硬件 FPU 对比，同样的条件下，快了近 10 倍，充分体现了 STM32F4 硬件 FPU 的优势。

第五十二章 DSP 测试实验

上一章，我们在 ALIENTEK 探索者 STM32F4 开发板上测试了 STM32F4 的硬件 FPU。STM32F4 除了集成硬件 FPU 外，还支持多种 DSP 指令集。同时 ST 还提供了一整套 DSP 库方便我们工程中开发应用。

本章，我们将指导大家入门 STM32F4 的 DSP，手把手教大家搭建 DSP 库测试环境，同时通过对 DSP 库中的几个基本数学功能函数和 FFT 快速傅里叶变换函数的测试，让大家对 STM32F4 的 DSP 库有个基本的了解。本章分为如下几个部分：

52.1 DSP 简介与环境搭建

52.2 硬件设计

52.3 软件设计

52.4 下载验证

52.1 DSP 简介与环境搭建

本节将分两个部分：1，STM32F4 DSP 简介；2，DSP 库运行环境搭建

52.1.1 STM32F4 DSP 简介

STM32F4 采用 Cortex-M4 内核，相比 Cortex-M3 系列除了内置硬件 FPU 单元，在数字信号处理方面还增加了 DSP 指令集，支持诸如单周期乘加指令（MAC），优化的单指令多数据指令（SIMD），饱和算数等多种数字信号处理指令集。相比 Cortex-M3，Cortex-M4 在数字信号处理能力方面得到了大大的提升。Cortex-M4 执行所有的 DSP 指令集都可以在单周期内完成，而 Cortex-M3 需要多个指令和多个周期才能完成同样的功能。

接下来我们来看看 Cortex-M4 的两个 DSP 指令：MAC 指令（32 位乘法累加）和 SIMD 指令。

32 位乘法累加（MAC）单元包括新的指令集，能够在单周期内完成一个 $32 \times 32 + 64 \rightarrow 64$ 的操作或两个 16×16 的操作，其计算能力，如表 52.1.1.1 所示：

计算	指令	周期
$16 \times 16 = 32$	SMULBB, SMULBT, SMULTB, SMULTT	1
$16 \times 16 + 32 = 32$	SMLABB, SMLABT, SMLATB, SMLATT	1
$16 \times 16 + 64 = 64$	SMLALBB, SMLALBT, SMLALTB, SMLALTT	1
$16 \times 32 = 32$	SMULWB, SMULWT	1
$(16 \times 32) + 32 = 32$	SMLAWB, SMLAWT	1
$(16 \times 16) \pm (16 \times 16) = 32$	SMUAD, SMUADX, SMUSD, SMUSDX	1
$(16 \times 16) \pm (16 \times 16) + 32 = 32$	SMLAD, SMLADX, SMLSD, SMLSX	1
$(16 \times 16) \pm (16 \times 16) + 64 = 64$	SMLALD, SMLALDX, SMLSX, SMLSX	1
$32 \times 32 = 32$	MUL	1
$32 \pm (32 \times 32) = 32$	MLA, MLS	1
$32 \times 32 = 64$	SMULL, UMULL	1
$(32 \times 32) + 64 = 64$	SMLAL, UMLAL	1
$(32 \times 32) + 32 + 32 = 64$	UMAAL	1
$2 \pm (32 \times 32) = 32$ (上)	SMMLA, SMMLAR, SMMLS, SMMLSR	1
$(32 \times 32) = 32$ (上)	SMMLU, SMMLUR	1

图 52.1.1.1 32 位乘法累加 (MAC) 单元的计算能力

Cortex-M4 支持 SIMD 指令集, 这在 Cortex-M3/M0 系列是不可用的。上述表中的指令, 有的属于 SIMD 指令。与硬件乘法器一起工作 (MAC), 使所有这些指令都能在单个周期内执行。受益于 SIMD 指令的支持, Cortex-M4 处理器能在单周期内完成高达 $32 \times 32 + 64 \rightarrow 64$ 的运算, 为其他任务释放处理器的带宽, 而不是被乘法和加法消耗运算资源。

比如一个比较复杂的运算: 两个 16×16 乘法加上一个 32 位加法, 如图 52.1.1.2 所示:

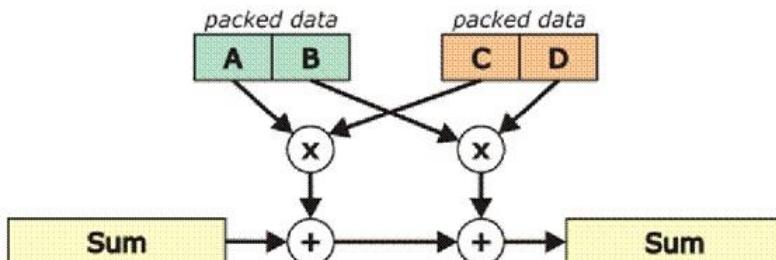


图 52.1.1.2 SUM 运算过程

以上图片所示的运算, 即: $SUM = SUM + (A * C) + (B * D)$, 在 STM32F4 上面, 可以被编译成由一条单周期指令完成。

上面我们简单的介绍了 Cortex-M4 的 DSP 指令, 接下来我们来介绍一下 STM32F4 的 DSP 库。

STM32F4 的 DSP 库源码和测试实例在 ST 提供的标准库: stm32f4_DSP_stdperiph.lib.zip 里面就有 (该文件可以在: <http://www.st.com/web/en/catalog/tools/FM147/CL1794/SC961/SS1743/PF257901> 下载, 文件名: STSW-STM32065), 该文件在: 光盘→8, STM32 参考资料→STM32F4xx 固件库 文件夹里面, 解压该文件, 即可找到 ST 提供的 DSP 库, 详细路径为: 光盘→8, STM32 参考资料→STM32F4xx 固件库→STM32F4xx_DSP_StdPeriph_Lib_V1.4.0→Libraries→CMSIS→DSP_Lib, 该文件夹下目录结构如图 52.1.1.3 所示:

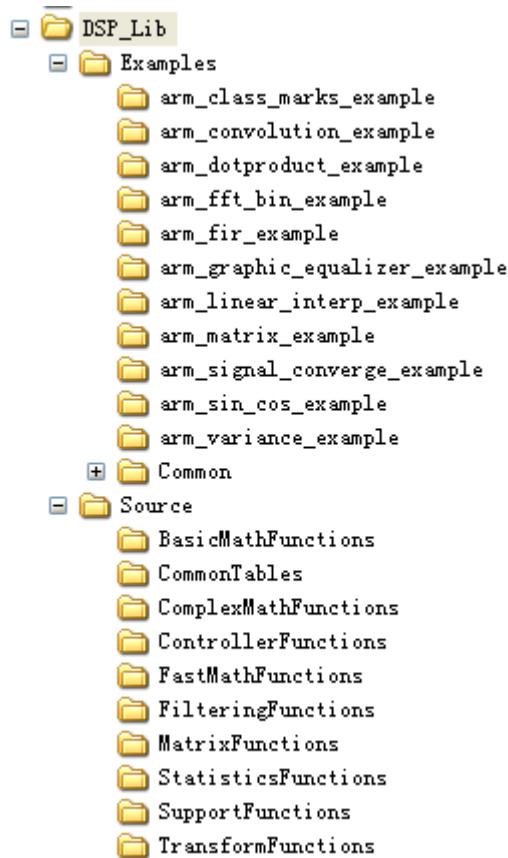


图 52.1.1.3 DSP_Lib 目录结构

DSP_Lib 源码包的 Source 文件夹是所有 DSP 库的源码，Examples 文件夹是相对应的一些测试实例。这些测试实例都是带 main 函数的，也就是拿到工程中可以直接使用。接下来我们一一讲解一下 Source 源码文件夹下面的子文件夹包含的 DSP 库的功能。

BasicMathFunctions

基本数学函数：提供浮点数的各种基本运算函数，如向量加减乘除等运算。

CommonTables

arm_common_tables.c 文件提供位翻转或相关参数表。

ComplexMathFunctions

复杂数学功能，如向量处理，求模运算的。

ControllerFunctions

控制功能函数。包括正弦余弦，PID 电机控制，矢量 Clarke 变换，矢量 Clarke 逆变换等。

FastMathFunctions

快速数学功能函数。提供了一种快速的近似正弦，余弦和平方根等相比 CMSIS 计算库要快的数学函数。

FilteringFunctions

滤波函数功能，主要为 FIR 和 LMS（最小均方根）等滤波函数。

MatrixFunctions

矩阵处理函数。包括矩阵加法、矩阵初始化、矩阵反、矩阵乘法、矩阵规模、矩阵减法、矩阵转置等函数。

StatisticsFunctions

统计功能函数。如求平均值、最大值、最小值、计算均方根 RMS、计算方差/标准差等。

SupportFunctions

支持功能函数，如数据拷贝，Q 格式和浮点格式相互转换，Q 任意格式相互转换。

TransformFunctions

变换功能。包括复数 FFT (CFFT) /复数 FFT 逆运算 (CIFFT)、实数 FFT (RFFT) /实数 FFT 逆运算 (RIFFT)、和 DCT (离散余弦变换) 和配套的初始化函数。

所有这些 DSP 库代码合在一起是比较大的，因此，ST 为我们提了.lib 格式的文件，方便使用。这些.lib 文件就是由 Source 文件夹下的源码编译生成的，如果想看某个函数的源码，大家可以在 Source 文件夹下面查找。.lib 格式文件路径：光盘→8，STM32 参考资料→STM32F4xx 固件库→STM32F4xx_DSP_StdPeriph_Lib_V1.4.0→Libraries→CMSIS→Lib→ARM，总共有 8 个.lib 文件，如下：

- ① arm_cortexM0b_math.lib (Cortex-M0 大端模式)
- ② arm_cortexM0l_math.lib (Cortex-M0 小端模式)
- ③ arm_cortexM3b_math.lib (Cortex-M3 大端模式)
- ④ arm_cortexM3l_math.lib (Cortex-M3 小端模式)
- ⑤ arm_cortexM4b_math.lib (Cortex-M4 大端模式)
- ⑥ arm_cortexM4bf_math.lib (Cortex-M4 小端模式)
- ⑦ arm_cortexM4l_math.lib (浮点 Cortex-M4 大端模式)
- ⑧ arm_cortexM4lf_math.lib (浮点 Cortex-M4 小端模式)

我们得根据所用 MCU 内核类型以及端模式来选择符合要求的.lib 文件，本章我们所用的 STM32F4 属于 CortexM4F 内核，小端模式，应选择：arm_cortexM4lf_math.lib(浮点 Cortex-M4 小端模式)。

对于 DSP_Lib 的子文件夹 Examples 下面存放的文件，是 ST 官方提供的一些 DSP 测试代码，提供简短的测试程序，方便上手，有兴趣的朋友可以根据需要自行测试。

52.1.2 DSP 库运行环境搭建

本节我们将讲解怎么搭建 DSP 库运行环境，只要运行环境搭建好了，使用 DSP 库里面的函数来做相关处理就非常简单了。本节，我们将以上一章例程（实验 46_1）为基础，搭建 DSP 运行环境。

在 MDK 里面搭建 STM32F4 的 DSP 运行环境(使用.lib 方式)是很简单的，分为 3 个步骤：

1，添加文件。

首先，我们在例程工程目录下新建：DSP_LIB 文件夹，存放我们将要添加的文件：arm_cortexM4lf_math.lib 和相关头文件，如图 52.1.2.1 所示：

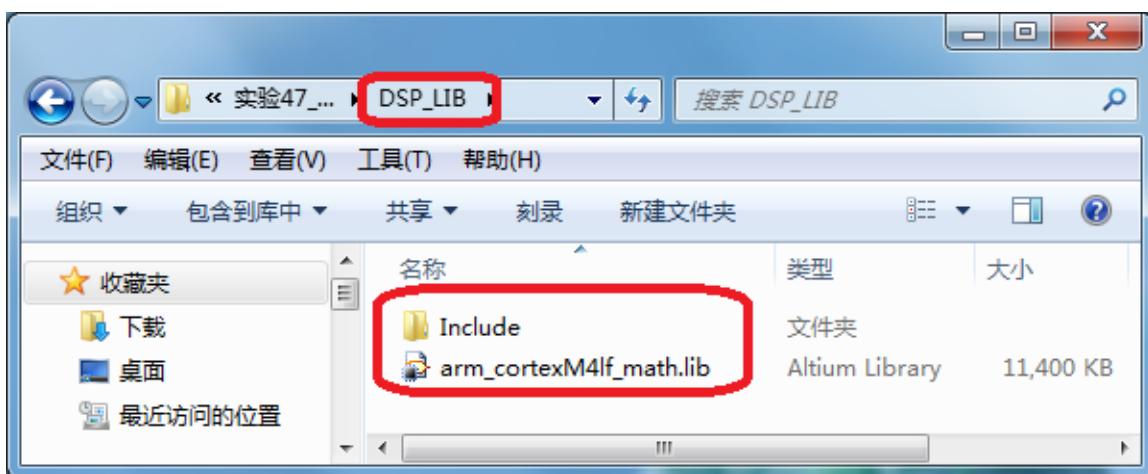


图 52.1.2.1 DSP_LIB 文件夹添加文件

其中 arm_cortexM4lf_math.lib 的由来，在 52.1.1 节已经介绍过了。Include 文件夹，则是直接拷贝：STM32F4xx_DSP_StdPeriph_Lib_V1.4.0→Libraries→CMSIS→Include 这个 Include 文件夹，里面包含了我们可能要用到的相关头文件。

然后，打开工程，新建 DSP_LIB 分组，并将 arm_cortexM4lf_math.lib 添加到工程里面，如图 52.1.2.2 所示：

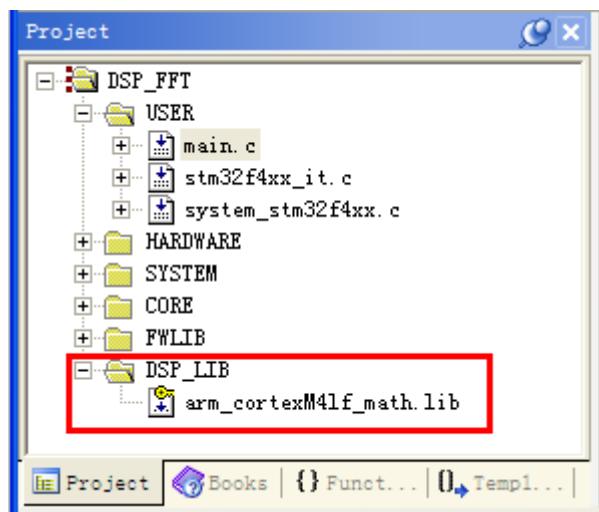


图 52.1.2.2 添加.lib 文件

这样，添加文件就结束了（就添加了一个.lib 文件）。

2. 添加头文件包含路径

添加好.lib 文件后，我们要添加头文件包含路径，将第一步拷贝的 Include 文件夹和 DSP_LIB 文件夹，加入头文件包含路径，如图 52.1.2.3 所示：

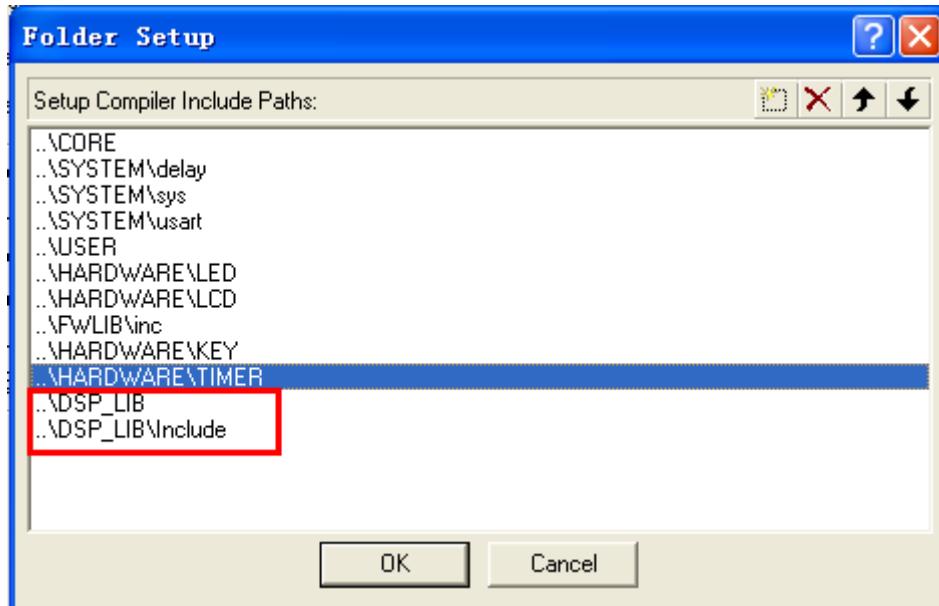


图 52.1.2.3 添加相关头文件包含路径

3. 添加全局宏定义

最后，为了使用 DSP 库的所有功能，我们还需要添加几个全局宏定义：

- 1, __FPU_USED
- 2, __FPU_PRESENT
- 3, ARM_MATH_CM4
- 4, __CC_ARM
- 5, ARM_MATH_MATRIX_CHECK
- 6, ARM_MATH_ROUNDING

添加方法：点击 → C/C++ 选项卡，然后在 Define 里面进行设置，如图 52.1.2.4 所示：

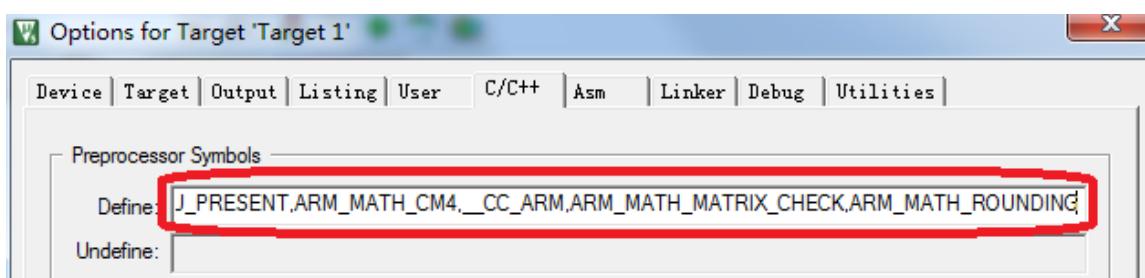


图 52.1.2.4 DSP 库支持全局宏定义设置

这里，两个宏之间用“,”隔开。并且，上面的全局宏里面，我们没有添加__FPU_USED，因为这个宏定义在 Target 选项卡设置 Code Generation 的时候(上一章有介绍)，选择了：Use FPU (如果没有设置 Use FPU，则必须设置!!)，故 MDK 会自动添加这个全局宏，因此不需要我们手动添加了。同时__FPU_PRESENT 全局宏我们 FPU 实验已经讲解，这个宏定义在 stm32f4xx.h 头文件里面已经定义。这样，在 Define 处要输入的所有宏为：STM32F40_41xxx,USE_STDPERIPH_DRIVER,ARM_MATH_CM4,__CC_ARM,ARM_MATH_MATRIX_CHECK,ARM_MATH_ROUNDING 共 6 个。

至此，STM32F4 的 DSP 库运行环境就搭建完成了。

特别注意，为了方便调试，本章例程我们将 MDK 的优化设置为-O0 优化，以得到最好的

调试效果。

52.2 硬件设计

本例程包含 2 个源码：实验 47_1 DSP BasicMath 测试和实验 47_2 DSP FFT 测试，他们除了 main.c 里面内容不一样外，其他源码完全一模一样（包括 MDK 配置）。

实验 47_1 DSP BasicMath 测试 实验功能简介：测试 STM32F4 的 DSP 库基础数学函数：arm_cos_f32 和 arm_sin_f32 和标准库基础数学函数：cosf 和 sinf 的速度差别，并在 LCD 屏幕上面显示两者计算所用时间，DS0 用于提示程序正在运行。

实验 47_2 DSP FFT 测试 实验功能简介：测试 STM32F4 的 DSP 库的 FFT 函数，程序运行后，自动生成 1024 点测试序列，然后，每当 KEY0 按下后，调用 DSP 库的 FFT 算法（基 4 法）执行 FFT 运算，在 LCD 屏幕上面显示运算时间，同时将 FFT 结果输出到串口，DS0 用于提示程序正在运行。

本实验用到的资源如下：

- 1, 指示灯 DS0
- 2, KEY0 按键
- 3, 串口
- 4, TFTLCD 模块

这些前面都已介绍过。

52.3 软件设计

本章代码，分成两个工程：1，实验 47_1 DSP BasicMath 测试；2，实验 47_2 DSP FFT 测试，接下来我们分别介绍。

52.3.1 DSP BasicMath 测试

这是我们使用 STM32F4 的 DSP 库进行基础数学函数测试的一个例程。使用大家耳熟能详的公式进行计算：

$$\sin(x)^2 + \cos(x)^2 = 1$$

这里我们用到的就是 sin 和 cos 函数，不过实现方式不同。MDK 的标准库（math.h）提供给我们：sin、cos、sinf 和 cosf 等 4 个函数，带 f 的表示单精度浮点型运算，即 float 型，而不带 f 的表示双精度浮点型，即 double。

STM32F4 的 DSP 库，则提供我们另外两个函数：arm_sin_f32 和 arm_cos_f32（注意：需要添加：arm_math.h 头文件才可使用!!!），这两个函数也是单精度浮点型的，用法同 sinf 和 cosf 一模一样。

本例程就是测试：arm_sin_f32& arm_cos_f32 同 sinf&cosf 的速度差别。

因为 52.1.2 节已经搭建好 DSP 库运行环境了，所以我们这里直接只需要修改 main.c 里面的代码即可，main.c 代码如下：

```
#include "math.h"
#include "arm_math.h"
#define DELTA 0.000001f      //误差值
//sin cos 测试    angle:起始角度   times:运算次数
//mode:0,不使用 DSP 库;1,使用 DSP 库
//返回值: 0,成功;0xFF,出错
```

```
u8 sin_cos_test(float angle,u32 times,u8 mode)
{
    float sinx,cosx;
    float result;
    u32 i=0;
    if(mode==0)
    {
        for(i=0;i<times;i++)
        {
            cosx=cosf(angle);           //不使用 DSP 优化的 sin, cos 函数
            sinx=sinf(angle);
            result=sinx*sinx+cosx*cosx; //计算结果应该等于 1
            result=fabsf(result-1.0f); //对比与 1 的差值
            if(result>DELTA) return 0XFF;//判断失败
            angle+=0.001f;             //角度自增
        }
    }
    else
    {
        for(i=0;i<times;i++)
        {
            cosx=arm_cos_f32(angle); //使用 DSP 优化的 sin, cos 函数
            sinx=arm_sin_f32(angle);
            result=sinx*sinx+cosx*cosx; //计算结果应该等于 1
            result=fabsf(result-1.0f); //对比与 1 的差值
            if(result>DELTA) return 0XFF;//判断失败
            angle+=0.001f;             //角度自增
        }
    }
    return 0;//任务完成
}
u8 timeout;//定时器溢出次数
int main(void)
{
    float time;
    u8 buf[50]; u8 res;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init();          //初始化 LED
    KEY_Init();          //初始化按键
    LCD_Init();          //初始化 LCD
    TIM3_Int_Init(65535,8400-1); //10Khz 计数频率,最大计时 6.5 秒超出
    POINT_COLOR=RED;
```

```

LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"DSP BasicMath TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/7/2");
LCD_ShowString(30,150,200,16,16," No DSP runtime:"); //显示提示信息
LCD_ShowString(30,190,200,16,16,"Use DSP runtime:"); //显示提示信息
POINT_COLOR=BLUE; //设置字体为蓝色
while(1)
{
    LCD_Fill(30+16*8,150	lcddev.width-1,60,WHITE); //清除原来现实
    //不使用 DSP 优化
    TIM_SetCounter(TIM3,0);//重设 TIM3 定时器的计数器值
    timeout=0;
    res=sin_cos_test(PI/6,200000,0);
    time=TIM_GetCounter(TIM3)+(u32)timeout*65536;
    sprintf((char*)buf,"%0.1fms\r\n",time/10);
    if(res==0)LCD_ShowString(30+16*8,150,100,16,16,buf); //显示运行时间
    else LCD_ShowString(30+16*8,150,100,16,16,"error! "); //显示当前运行情况
    //使用 DSP 优化
    TIM_SetCounter(TIM3,0);//重设 TIM3 定时器的计数器值
    timeout=0;
    res=sin_cos_test(PI/6,200000,1);
    time= TIM_GetCounter(TIM3)+(u32)timeout*65536;
    sprintf((char*)buf,"%0.1fms\r\n",time/10);
    if(res==0)LCD_ShowString(30+16*8,190,100,16,16,buf); //显示运行时间
    else LCD_ShowString(30+16*8,190,100,16,16,"error! "); //显示错误
    LED0=!LED0;
}
}

```

这里包括 2 个函数：sin_cos_test 和 main 函数，sin_cos_test 函数用于根据给定参数，执行 $\sin(x)^2 + \cos(x)^2 = 1$

的计算。计算完后，计算结果同给定的误差值（DELTA）对比，如果不大于误差值，则认为计算成功，否则计算失败。该函数可以根据给定的模式参数(mode)来决定使用哪个基础数学函数执行运算，从而得出对比。

main 函数则比较简单，这里我们通过定时器 3 来统计 sin_cos_test 运行时间，从而得出对比数据。主循环里面，每次循环都会两次调用 sin_cos_test 函数，首先采用不使用 DSP 库方式计算，然后采用使用 DSP 库方式计算，并得出两次计算的时间，显示在 LCD 上面。

DSP 基础数学函数测试的程序设计就讲解到这里。

52.3.1 DSP FFT 测试

这是我们使用 STM32F4 的 DSP 库进行 FFT 函数测试的一个例程。

首先，我们简单介绍下 FFT：FFT 即快速傅里叶变换，可以将一个时域信号变换到频域。因为有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征

了，这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。简而言之，FFT 就是将一个信号从时域变换到频域方便我们分析处理。

在实际应用中，一般的处理过程是先对一个信号在时域进行采集，比如我们通过 ADC，按照一定大小采样频率 F 去采集信号，采集 N 个点，那么通过对这 N 个点进行 FFT 运算，就可以得到这个信号的频谱特性。

这里还涉及到一个采样定理的概念：在进行模拟/数字信号的转换过程中，当采样频率 F 大于信号中最高频率 f_{max} 的 2 倍时($F > 2 * f_{max}$)，采样之后的数字信号完整地保留了原始信号中的信息，采样定理又称奈奎斯特定理。举个简单的例子：比如我们正常人发声，频率范围一般在 8KHz 以内，那么我们要通过采样之后的数据来恢复声音，我们的采样频率必须为 8KHz 的 2 倍以上，也就是必须大于 16KHz 才行。

模拟信号经过 ADC 采样之后，就变成了数字信号，采样得到的数字信号，就可以做 FFT 变换了。N 个采样点数据，在经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。

假设采样频率为 F，对一个信号采样，采样点数为 N，那么 FFT 之后结果就是一个 N 点的复数，每一个点就对应着一个频率点（以基波频率为单位递增），这个点的模值 ($\sqrt{(\text{实部}^2 + \text{虚部}^2)}$) 就是该频点频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 $N/2$ 倍，而第一个点就是直流分量，它的模值就是直流分量的 N 倍。

这里还有个基波频率，也叫频率分辨率的概念，就是如果我们按照 F 的采样频率去采集一个信号，一共采集 N 个点，那么基波频率（频率分辨率）就是 $f_k = F/N$ 。这样，第 n 个点对应信号频率为： $F*(n-1)/N$ ；其中 $n \geq 1$ ，当 $n=1$ 时为直流分量。

关于 FFT 我们就介绍到这。

如果我们要自己实现 FFT 算法，对于不懂数字信号处理的朋友来说，是比较难的，不过，ST 提供的 STM32F4 DSP 库里面有 FFT 函数给我们调用，因此我们只需要知道如何使用这些函数，就可以迅速的完成 FFT 计算，而不需要自己学习数字信号处理，去编写代码了，大大方便了我们的开发。

STM32F4 的 DSP 库里面，提供了定点和浮点 FFT 实现方式，并且有基 4 的也有基 2 的，大家可以根据需要自由选择实现方式。注意：对于基 4 的 FFT 输入点数必须是 4^n ，而基 2 的 FFT 输入点数则必须是 2^n ，并且基 4 的 FFT 算法要比基 2 的快。

本章我们将采用 DSP 库里面的基 4 浮点 FFT 算法来实现 FFT 变换，并计算每个点的模值，所用到的函数有：

```
arm_status arm_cfft_radix4_init_f32(
    arm_cfft_radix4_instance_f32 * S,
    uint16_t fftLen,uint8_t ifftFlag,uint8_t bitReverseFlag)
void arm_cfft_radix4_f32(const arm_cfft_radix4_instance_f32 * S,float32_t * pSrc)
void arm_cmplx_mag_f32(float32_t * pSrc,float32_t * pDst,uint32_t numSamples)
```

第一个函数 arm_cfft_radix4_init_f32，用于初始化 FFT 运算相关参数，其中：fftLen 用于指定 FFT 长度 (16/64/256/1024/4096)，本章设置为 1024；ifftFlag 用于指定是傅里叶变换(0)还是反傅里叶变换(1)，本章设置为 0；bitReverseFlag 用于设置是否按位取反，本章设置为 1；最后，所有这些参数存储在一个 arm_cfft_radix4_instance_f32 结构体指针 S 里面。

第二个函数 arm_cfft_radix4_f32 就是执行基 4 浮点 FFT 运算的，pSrc 传入采集到的输入信号数据（实部+虚部形式），同时 FFT 变换后的数据，也按顺序存放在 pSrc 里面，pSrc 必须大

于等于 2 倍 fftLen 长度。另外，S 结构体指针参数是先由 arm_cfft_radix4_init_f32 函数设置好，然后传入该函数的。

第三个函数 arm_cmplx_mag_f32 用于计算复数模值，可以对 FFT 变换后的结果数据，执行取模操作。pSrc 为复数输入数组（大小为 2*numSamples）指针，指向 FFT 变换后的结果；pDst 为输出数组（大小为 numSamples）指针，存储取模后的值；numSamples 就是总共有多少个数据需要取模。

通过这三个函数，我们便可以完成 FFT 计算，并取模值。本节例程（实验 47_2 DSP FFT 测试）同样是在 52.1.2 节已经搭建好 DSP 库运行环境上面修改代码，只需要修改 main.c 里面的代码即可，本例程 main.c 代码如下：

```
#include "math.h"
#include "arm_math.h"
#define FFT_LENGTH      1024      //FFT 长度，默认是 1024 点 FFT
float fft_inputbuf[FFT_LENGTH*2]; //FFT 输入数组
float fft_outputbuf[FFT_LENGTH]; //FFT 输出数组
u8 timeout;//定时器溢出次数
int main(void)
{
    arm_cfft_radix4_instance_f32 scfft;
    u8 key,t=0;    float time;
    u8 buf[50];    u16 i;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    KEY_Init(); //初始化按键
    LCD_Init(); //初始化 LCD
    TIM3_Int_Init(65535,84-1); //1Mhz 计数频率,最大计时 65ms 左右超出
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"DSP FFT TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/7/2");
    LCD_ShowString(30,130,200,16,16,"KEY0:Run FFT"); //显示提示信息
    LCD_ShowString(30,160,200,16,16,"FFT runtime:"); //显示 FFT 执行时间
    POINT_COLOR=BLUE; //设置字体为蓝色
    arm_cfft_radix4_init_f32(&scfft,FFT_LENGTH,0,1); //初始化 scfft 结构体,设定 FFT 参数
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY0_PRES)
        {
            for(i=0;i<FFT_LENGTH;i++)//生成信号序列
            {
                buf[i]=sin((float)i*3.1415926/100);
            }
            arm_cfft_radix4_f32(&scfft,fft_inputbuf,fft_outputbuf,0);
            for(i=0;i<FFT_LENGTH;i++)
            {
                time+=fft_outputbuf[i];
            }
            LCD_ShowString(30,180,200,16,16,"FFT runtime:");
            LCD_ShowString(30,200,200,16,16,(time/100));
        }
    }
}
```

```

fft_inputbuf[2*i]=100+
    10*arm_sin_f32(2*PI*i/FFT_LENGTH)+  

    30*arm_sin_f32(2*PI*i*4/FFT_LENGTH)+  

    50*arm_cos_f32(2*PI*i*8/FFT_LENGTH); //实部  

fft_inputbuf[2*i+1]=0;//虚部全部为 0
}  

TIM_SetCounter(TIM3,0);//重设 TIM3 定时器的计数器值  

timeout=0;  

arm_cfft_radix4_f32(&scfft,fft_inputbuf); //FFT 计算 (基 4)  

time= TIM_GetCounter(TIM3)+(u32)timeout*65536; //计算所用时间  

sprintf((char*)buf,"%0.3fms\r\n",time/1000);  

LCD_ShowString(30+12*8,160,100,16,16,buf); //显示运行时间  

arm_cmplx_mag_f32(fft_inputbuf,fft_outputbuf,FFT_LENGTH); //取模得幅值  

printf("\r\n%d point FFT runtime:%0.3fms\r\n",FFT_LENGTH,time/1000);  

printf("FFT Result:\r\n");
for(i=0;i<FFT_LENGTH;i++)
{
    printf("fft_outputbuf[%d]:%f\r\n",i,fft_outputbuf[i]);
}
}while delay_ms(10);
t++;
if((t%10)==0)LED0=!LED0;
}
}

```

以上代码只有一个 main 函数，里面通过我们前面介绍的三个函数：arm_cfft_radix4_init_f32、arm_cfft_radix4_f32 和 arm_cmplx_mag_f32 来执行 FFT 变换并取模值。每当按下 KEY0 就会重新生成一个输入信号序列，并执行一次 FFT 计算，将 arm_cfft_radix4_f32 所用时间统计出来，显示在 LCD 屏幕上面，同时将取模后的模值通过串口打印出来。

这里，我们在程序上生成了一个输入信号序列用于测试，输入信号序列表达式：

```

fft_inputbuf[2*i]=100+
    10*arm_sin_f32(2*PI*i/FFT_LENGTH)+  

    30*arm_sin_f32(2*PI*i*4/FFT_LENGTH)+  

    50*arm_cos_f32(2*PI*i*8/FFT_LENGTH); //实部

```

通过该表达式我们可知，信号的直流分量为 100，外加 2 个正弦信号和一个余弦信号，其幅值分别为 10、30 和 50。

关于输出结果分析，请看 52.4 节，软件设计我们就介绍到这里。

52.4 下载验证

代码编译成功之后，便可以下载到我们的探索者 STM32F4 开发板上验证了。

对于实验 47_1 DSP BasicMath 测试，下载后，可以在屏幕看到两种实现方式的速度差别，如图 52.4.1 所示：

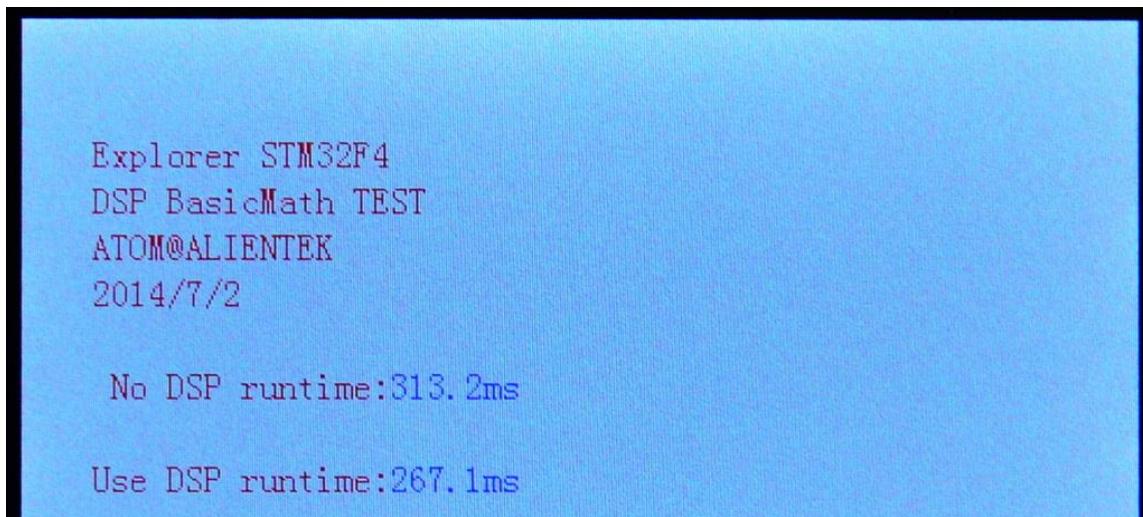


图 52.4.1 使用 DSP 库和不使用 DSP 库的基础数学函数速度对比

从图中可以看出，使用 DSP 库的基础数学函数计算所用时间比不使用 DSP 库的短，使用 STM32F4 的 DSP 库，速度上面比传统的实现方式提升了约 17%。

对于实验 47_2 DSP FFT 测试，下载后，屏幕显示提示信息，然后我们按下 KEY0 就可以看到 FFT 运算所耗时间，如图 52.4.2 所示：

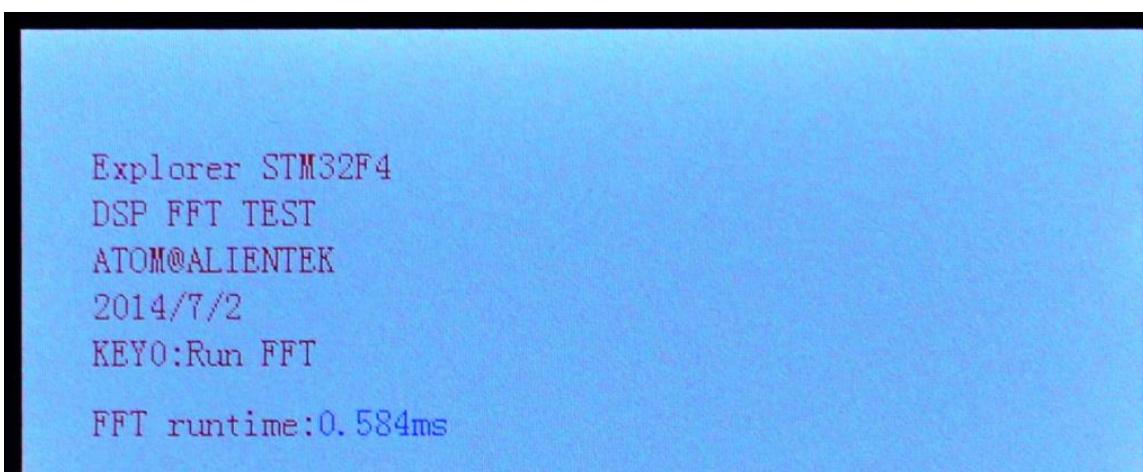


图 52.4.2 FFT 测试界面

可以看到，STM32F4 采用基 4 法计算 1024 个浮点数的 FFT，只用了 0.584ms，速度是相当快的了。同时，可以在串口看到 FFT 变换取模后的各频点模值，如图 52.4.3 所示：



图 52.4.3 FFT 变换后个频点模值

查看所有数据，会发现：第 0、1、4、8、1016、1020、1023 这 7 个点的值比较大，其他点的值都很小，接下来我们就简单分析一下这些数据。

由于 FFT 变换后的结果具有对称性，所以，实际上有用的数据，只有前半部分，后半部分和前半部分是对称关系，比如 1 和 1023，4 和 1020,8 和 1016 等，就是对称关系，因此我们只需要分析前半部分数据即可。这样，就只有第 0、1、4、8 这四个点，比较大，重点分析。

假设我们采样频率为 1024Hz，那么总共采集 1024 个点，频率分辨率就是 1Hz，对应到频谱上面，两个点之间的间隔就是 1Hz。因此，上面我们生成的三个叠加信号： $10\sin(2\pi i \cdot 1/1024) + 30\sin(2\pi i \cdot 4/1024) + 50\cos(2\pi i \cdot 8/1024)$ ，频率分别是：1Hz、4Hz 和 8Hz。

对于上述 4 个值比较大的点，结合 52.3.1 节的知识，很容易分析得出：第 0 点，即直流分量，其 FFT 变换后的模值应该是原始信号幅值的 N 倍， $N=1024$ ，所以值是 $100 \cdot 1024 = 102400$ ，与理论完全一样，然后其他点，模值应该是原始信号幅值的 $N/2$ 倍，即 $10 \cdot 512, 30 \cdot 512, 50 \cdot 512$ ，而我们计算结果是：5119.999023、15360、256000，除了第 1 个点，稍微有点点误差（说明精度上有损失），其他同理论值完全一致。

DSP 测试实验，我们就讲解到这里，DSP 库的其他测试实例，大家可以自行研究下，我们这里就不再介绍了。

第五十三章 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 探索者 STM32F4 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分：

- 53.1 手写识别简介
- 53.2 硬件设计
- 53.3 软件设计
- 53.4 下载验证

53.1 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 探索者 STM32F4 开发板自带的 TFTLCD 触摸屏（2.8/3.5/4.3 寸），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 53.1.1 所示：

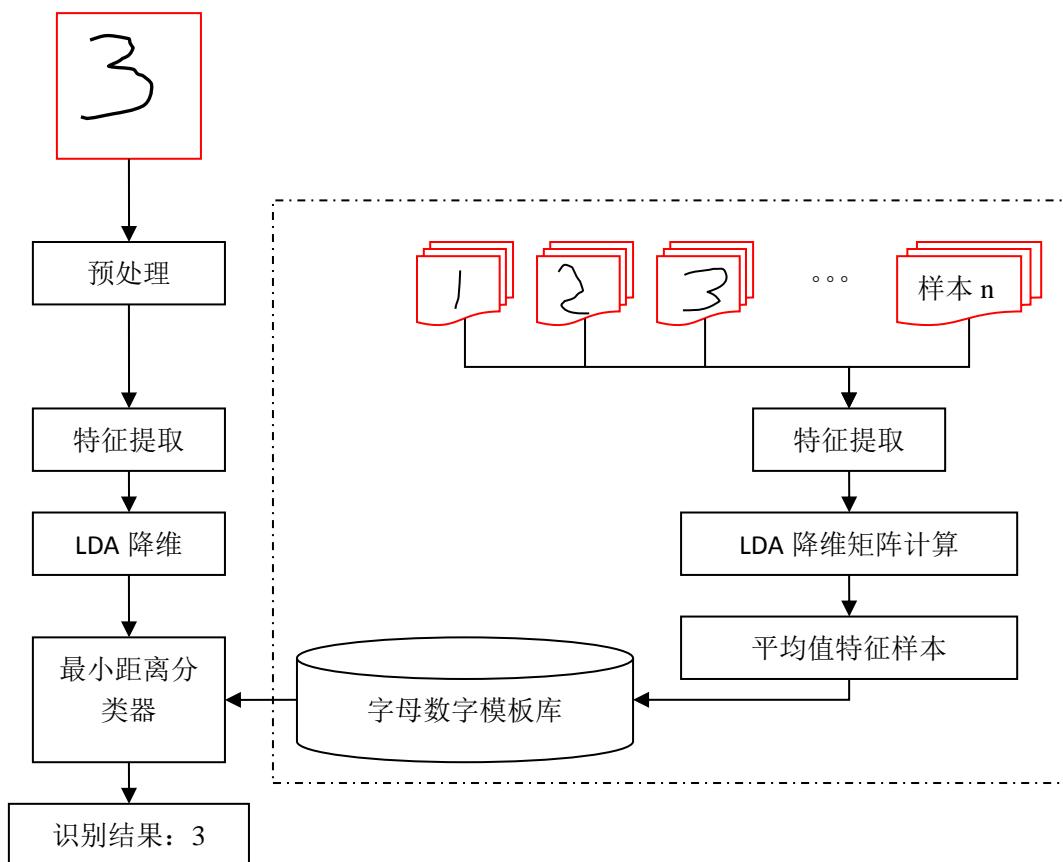


图 53.1.1 字母数字识别系统示意图。

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，计算量和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，即是假设所有样本服从高斯分布（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先将经过预处理后的 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 $64 \times 8 = 512$ 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。

$$\begin{bmatrix} d_1, d_2, \dots, d_{512} \end{bmatrix} \times \begin{bmatrix} l & \cdots & l \\ \vdots & \ddots & \vdots \\ l & \cdots & l \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_{64} \end{bmatrix}$$

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。

53.2 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。

53.3 软件设计

打开本章实验工程目录可以看到，我们在工程根目录文件夹下新建一个 ATKNCR 的文件夹。将 ALIETENK 提供的手写识别库文件（ATKNCR_M_V2.0.lib、ATKNCR_N_V2.0.lib、atk_ncr.c

和 atk_ncr.h 这四个个文件，在光盘→4，程序源码→5，ATKNCR(数字字母手写识别库) 文件夹里面) 拷贝到该文件夹下，然后在工程里面新建一个 ATKNCR 的组，将 atk_ncr.c 和 ATKNCR_M_V2.0.lib 加入到该组下面(这里我们使用内存管理版本的识别库)。最后，将 ATKNCR 文件夹加入头文件包含路径。

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 里面修改代码如下：

```
//最大记录的轨迹点数
atk_ncr_point READ_BUF[200];
//画水平线
//x0,y0:坐标 len:线长度 color:颜色
void gui_draw_hline(u16 x0,u16 y0,u16 len,u16 color)
{
    if(len==0)return;
    LCD_Fill(x0,y0,x0+len-1,y0,color);
}

//画实心圆
//x0,y0:坐标 r:半径 color:颜色
void gui_fill_circle(u16 x0,u16 y0,u16 r,u16 color)
{
    u32 i;
    u32 imax = ((u32)r*707)/1000+1;
    u32 sqmax = (u32)r*(u32)r+(u32)r/2;
    u32 x=r;
    gui_draw_hline(x0-r,y0,2*r,color);
    for (i=1;i<=imax;i++)
    {
        if ((i*i+x*x)>sqmax)// draw lines from outside
        {
            if (x>imax)
            {
                gui_draw_hline (x0-i+1,y0+x,2*(i-1),color);
                gui_draw_hline (x0-i+1,y0-x,2*(i-1),color);
            }
            x--;
        }
        // draw lines from inside (center)
        gui_draw_hline(x0-x,y0+i,2*x,color);
        gui_draw_hline(x0-x,y0-i,2*x,color);
    }
}
//两个数之差的绝对值
//x1,x2: 需取差值的两个数
//返回值: |x1-x2|
```

```
u16 my_abs(u16 x1,u16 x2)
{
    if(x1>x2) return x1-x2;
    else return x2-x1;
}

//画一条粗线
//(x1,y1),(x2,y2):线条的起始坐标
//size: 线条的粗细程度
//color: 线条的颜色
void lcd_draw_bline(u16 x1, u16 y1, u16 x2, u16 y2,u8 size,u16 color)
{
    u16 t;
    int xerr=0,yerr=0,delta_x,delta_y,distance;
    int incx,incy,uRow,uCol;
    if(x1<size|| x2<size||y1<size|| y2<size)return;
    delta_x=x2-x1; //计算坐标增量
    delta_y=y2-y1;
    uRow=x1; uCol=y1;
    if(delta_x>0)incx=1; //设置单步方向
    else if(delta_x==0)incx=0;//垂直线
    else {incx=-1;delta_x=-delta_x;}
    if(delta_y>0)incy=1;
    else if(delta_y==0)incy=0;//水平线
    else {incy=-1;delta_y=-delta_y;}
    if( delta_x>delta_y)distance=delta_x; //选取基本增量坐标轴
    else distance=delta_y;
    for(t=0;t<=distance+1;t++)//画线输出
    {
        gui_fill_circle(uRow,uCol,size,color);//画点
        xerr+=delta_x ; yerr+=delta_y ;
        if(xerr>distance){ xerr-=distance; uRow+=incx;}
        if(yerr>distance){ yerr-=distance;uCol+=incy;}
    }
}
int main(void)
{
    u8 i=0; u8 tcnt=0; u8 key;u8 res[10];
    u16 pcnt=0;     u8 mode=4;      //默认是混合模式
    u16 lastpos[2]; //最后一次的数据
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
```

```
LCD_Init();           //LCD 初始化
KEY_Init();           //按键初始化
W25QXX_Init();       //初始化 W25Q128
tp_dev.init();        //初始化触摸屏
my_mem_init(SRAMIN); //初始化内部内存池
my_mem_init(SRAMCCM); //初始化 CCM 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:
POINT_COLOR=RED;
Show_Str(60,10,200,16,"探索者 STM32F407 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,lcddev.width-20,lcddev.height-5);
POINT_COLOR=BLUE;
Show_Str(96,207,200,16,"手写区",16,0);
tcnt=100; tcnt=100;
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY2_PRES&&(tp_dev.touchtype&0X80)==0)
    {
        TP_Adjust(); //屏幕校准
        LCD_Clear(WHITE);
        goto RESTART; //重新加载界面
    }
    if(key==KEY0_PRES)
    {
        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[0]<(lcddev.width-20-2)&&tp_dev.x[0]>=(20+2))&&(tp_dev.y[0]
        <(lcddev.height-5-2)&&tp_dev.y[0]>=(115+2)))
    {
        if(lastpos[0]==0xFFFF) { lastpos[0]=tp_dev.x[0]; lastpos[1]=tp_dev.y[0];}
        lcd_draw_bline(lastpos[0],lastpos[1],tp_dev.x[0],tp_dev.y[0],2,BLUE);//画线
        lastpos[0]=tp_dev.x[0]; lastpos[1]=tp_dev.y[0];
        if(pcnt<200)//总点数少于 200
        {
            if(pcnt)
            {
                if((READ_BUF[pcnt-1].y!=tp_dev.y[0])&&(READ_BUF[pcnt-1]
                    .x!=tp_dev.x[0]))//x,y 不相等
                {
                    READ_BUF[pcnt].x=tp_dev.x[0];
                    READ_BUF[pcnt].y=tp_dev.y[0];
                    pcnt++;
                }
            }
            else
            {
                READ_BUF[pcnt].x=tp_dev.x[0];
                READ_BUF[pcnt].y=tp_dev.y[0];
                pcnt++;
            }
        }
    }
}
}else //按键松开了
{
    lastpos[0]=0xFFFF;
    tcnt++;delay_ms(10);
    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,lcddev.width-20-1,lcddev.height-5-1,WHITE);
}
}
if(i==30) {i=0; LED0=!LED0;}
}
}
```

这里代码看上去比较多，其实很多都是为 lcd_draw_bline 函数服务的，lcd_draw_bline 函数用于实现画指定粗细的直线，以得到较好的画线效果。而 main 函数，则实现 53.1.2 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重复的点阵（即相邻的坐标不相等）。这样可以避免重复数据，而重复的点阵数据对识别是没有帮助的。

至此，本实验的软件设计部分结束。

53.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 开发板上，得到，如图 53.4.1 所示：

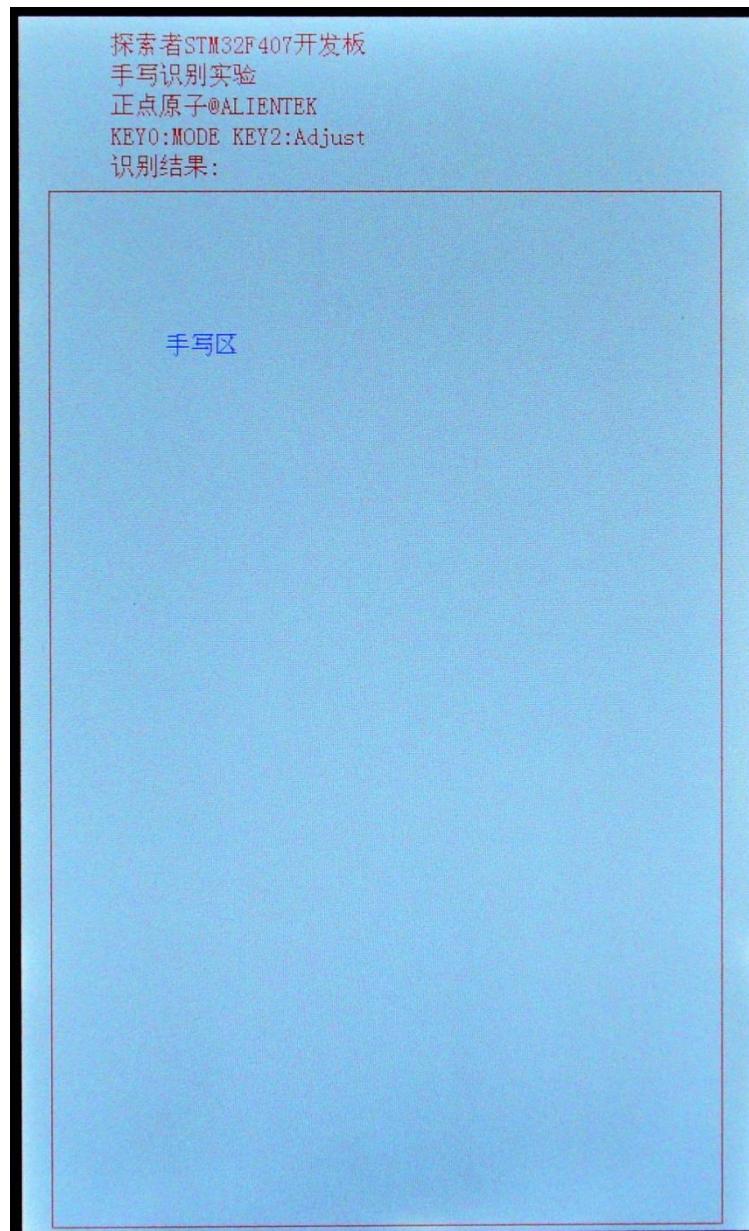


图 53.4.1 手写识别界面

此时，我们在手写区写数字/字母，即可得到识别结果，如图 53.4.2 所示：

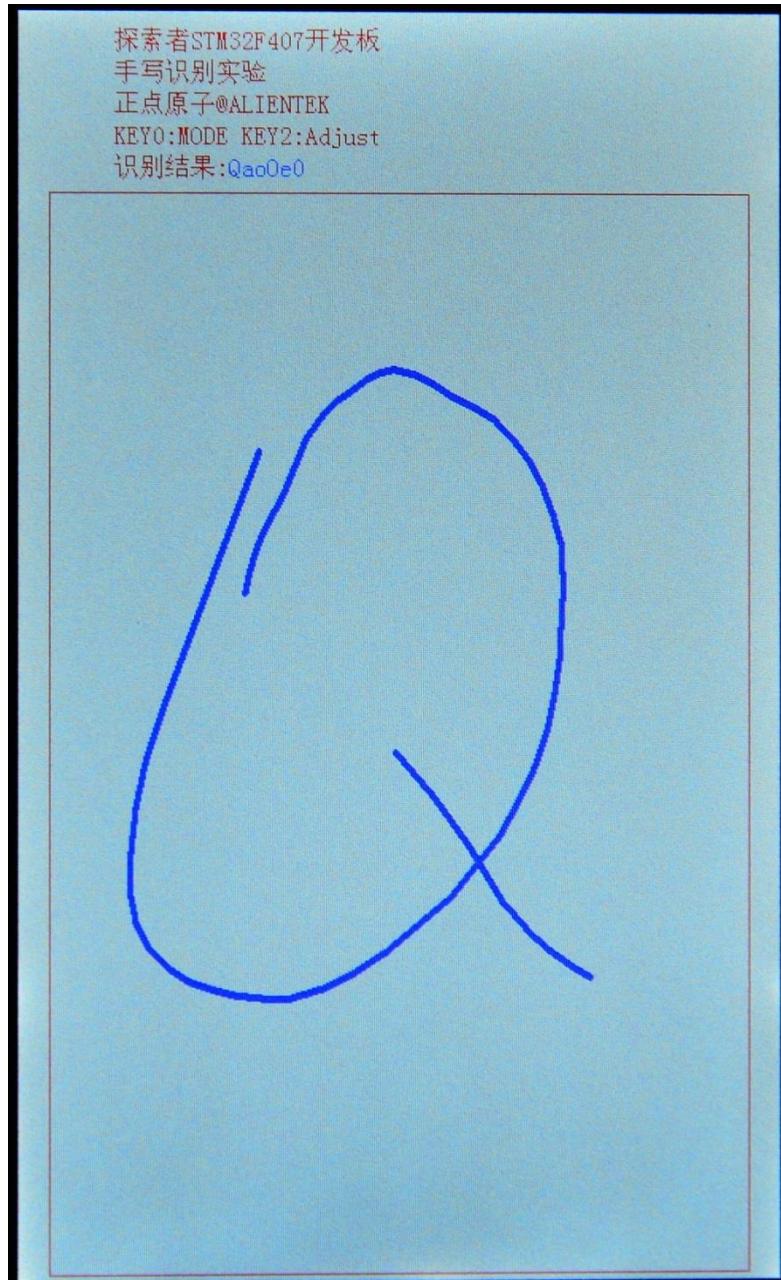


图 53.4.2 手写识别结果

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。

第五十四章 T9 拼音输入法实验

上一章，我们在 ALIENTEK 探索者 STM32F4 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 探索者 STM32F4 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分：

- 54.1 拼音输入法简介
- 54.2 硬件设计
- 54.3 软件设计
- 54.4 下载验证

54.1 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 54.1.1 所示：



图 54.1.1 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词，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      []={"啊阿唵吖锕屁嘎锕呵唵"};
const u8 PY_mb_ai     []={"爱埃挨哎唉哀皑癌蔼矮艾碍隘捱嗳嗌媛嗳破锿鬻"};
const u8 PY_mb_an     []={"安俺按暗岸案鞍氨谙胺埯指犴庵桉铵鵠黯"};
.....此处省略 N 多组合
const u8 PY_mb_zu     []={"足租祖沮阻组卒族俎菹餼"};
const u8 PY_mb_zuan   []={"钻攢纂缵蹠"};
const u8 PY_mb_zui    []={"最罪嘴醉蕞觜"};
const u8 PY_mb_zun    []={"尊遵樽撙撙"};
const u8 PY_mb_zuo    []={"左佐做作坐座昨撮咤柞阵琢嘬怍胙祚痄酢"};

```

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。

54.2 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 KEY_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/KEY_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。

54.3 软件设计

打开本章实验工程可以看到，我们在根目录文件夹下新建了一个 T9INPUT 的文件夹。在该文件夹下面新建了 pyinput.c、pyinput.h 和 pymb.h 三个文件，然后在工程里面新建一个 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，代码如下：

```
//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=0;//最佳匹配
    u16 pyindex_len=0;
    u16 i=0;
    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字符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，保存 `pyinput.c`，打开 `pyinput.h`，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
```

保存 pyinput.h。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，我们看看主函数代码：

```
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
u16 kbdxsize; //虚拟键盘按键宽度
u16 kbdysize; //虚拟键盘按键高度
//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+kbdxsize*3,y+kbysize*3);
    LCD_DrawRectangle(x+kbdxsize,y,x+kbdxsize*2,y+kbysize*3);
    LCD_DrawRectangle(x,y+kbysize,x+kbdxsize*3,y+kbysize*2);
    POINT_COLOR=BLUE;
    for(i=0;i<9;i++)
    {
        Show_Str_Mid(x+(i%3)*kbdxsize,y+4+kbysize*(i/3),(u8*)kbd_tbl[i],16,kbdxsize);
        Show_Str_Mid(x+(i%3)*kbdxsize,y+kbysize/2+kbysize*(i/3),(u8*)kbs_tbl[i],
                      16,kbdxsize);
    }
}
//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*kbdxsize+1,y+i*kbysize+1,x+j*kbdxsize+kbdxsize-1,y+i*kbysize+
                     kbysize-1,WHITE);
    else LCD_Fill(x+j*kbdxsize+1,y+i*kbysize+1,x+j*kbdxsize+kbdxsize-1,y+i*kbysize+
                  kbysize-1,WHITE);
    Show_Str_Mid(x+j*kbdxsize,y+4+kbysize*i,(u8*)kbd_tbl[keyx],16,kbdxsize);
```

```
Show_Str_Mid(x+j*kbdxsize,y+kbysize/2+kbysize*i,(u8*)kbs_tbl[keyx],16,kbdsiz
}
//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; u8 key=0;
    static u8 key_x=0;//0,没有任何按键按下； 1~9， 1~9 号按键按下
    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[0]<(x+j*kbdsiz
                tp_dev.y[0]<(y+i*kbysize+kbysize)&&tp_dev.y[0]>(y+i*kbysize))
                {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,lcddev.width,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,lcddev.width-70,48,t9.pymb[index-1]->pymb,16,0); //显示汉字
    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 key; u8 cur_index; u8 result_num;
    u8 inputstr[7]; //最大输入 6 个字符+结束符
    u8 inputlen; //输入长度
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    usmart_dev.init(84); //初始化 USMART
    LCD_Init(); //LCD 初始化
    KEY_Init(); //按键初始化
    W25QXX_Init(); //初始化 W25Q128
    tp_dev.init(); //初始化触摸屏
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 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(30,5,200,16,"探索者 STM32F407 开发板",16,0);
    Show_Str(30,25,200,16,"拼音输入法实验",16,0);
    Show_Str(30,45,200,16,"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);
    if(lcddev.id==0X5310){kbdxsize=86;kbdsiz=43;} //根据 LCD 分辨率设置按键大小
    else if(lcddev.id==0X5510){kbdxsize=140;kbdsiz=70;}
    else {kbdxsize=60;kbdsiz=40;}
    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&0X7F;//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
            }
        }
        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
}
```

```
        }
        key=KEY_Scan(0);
        if(key==KEY2_PRES&&tp_dev.touchtype==0)//KEY2 按下,且是电阻屏
        {
            tp_dev.adjust();
            LCD_Clear(WHITE);
            goto RESTART;
        }
        if(result_num)//存在匹配的结果
        {
            switch(key)
            {
                case WKUP_PRES://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 KEY1_PRES://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 KEY0_PRES://清除输入
                    LCD_Fill(30+40,145	lcddev.width,145+48,WHITE);//清除之前的显示
                    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 54.2 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。注意：kbdsiz 和 kbdysize 代表虚拟键盘按键宽度和高度，程序根据 LCD 分辨率不同而自动设置这两个参数，以达到较好的输入效果。

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。

54.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 开发板上，得到，如图 54.4.1 所示：

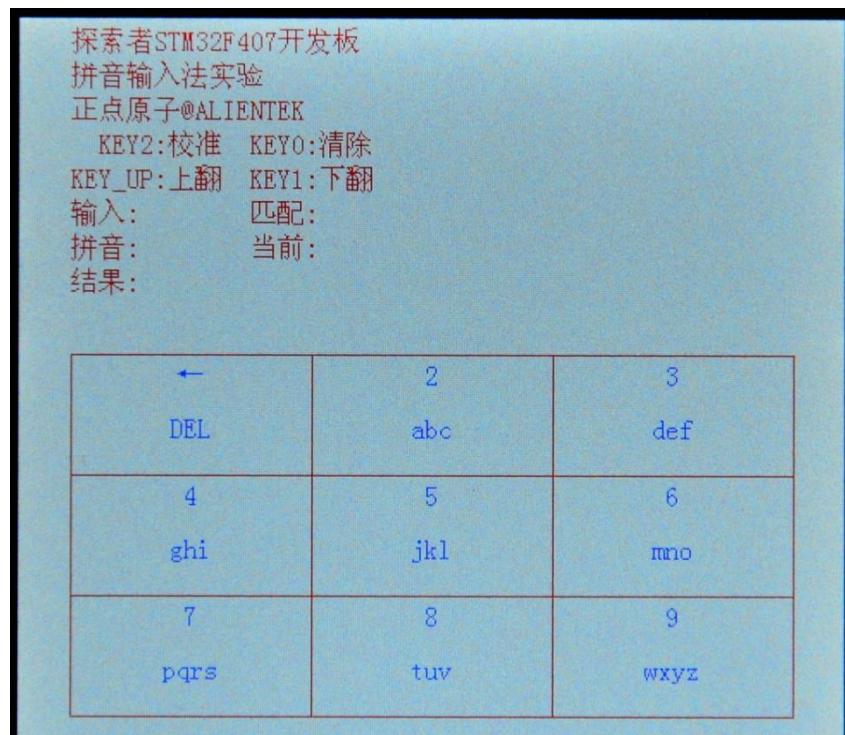


图 54.4.1 汉字输入法界面

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 54.4.2 所示：

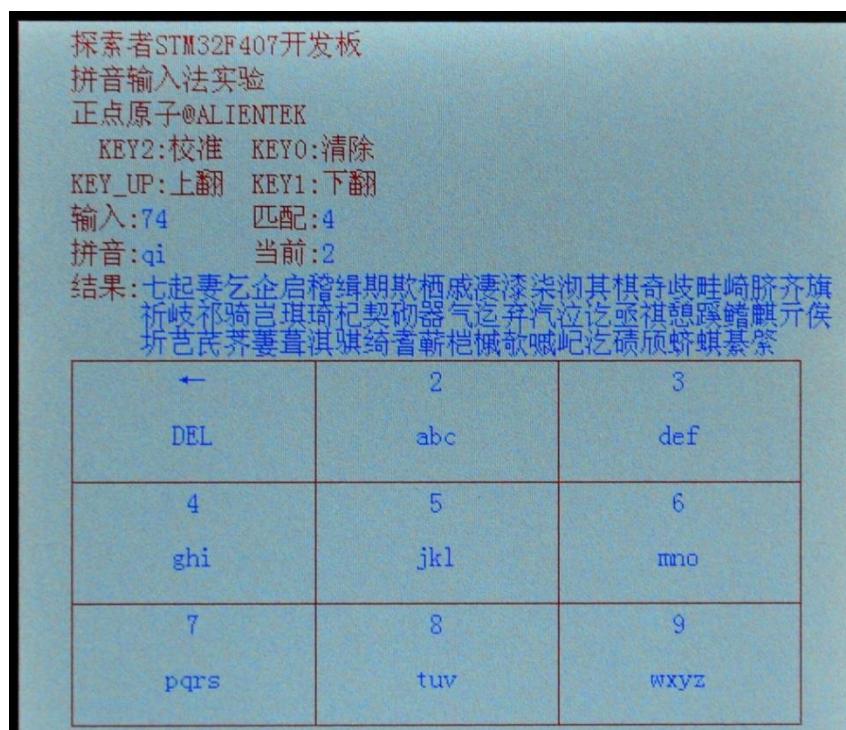


图 54.4.2 实现拼音输入

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 KEY_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 54.4.3 所示：



图 54.4.3 USMART 调试 T9 拼音输入法

第五十五章 串口 IAP 实验

IAP，即在应用编程。很多单片机都支持这个功能，STM32F4 也不例外。在之前的 FLASH 模拟 EEPROM 实验里面，我们学习了 STM32F4 的 FLASH 自编程，本章我们将结合 FLASH 自编程的知识，通过 STM32F4 的串口实现一个简单的 IAP 功能。本章分为如下几个部分：

55.1 IAP 简介

55.2 硬件设计

55.3 软件设计

55.4 下载验证

55.1 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 STM32F4 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。

STM32F4 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 STM32F4 正常的程序运行流程，如图 55.1.1 所示：

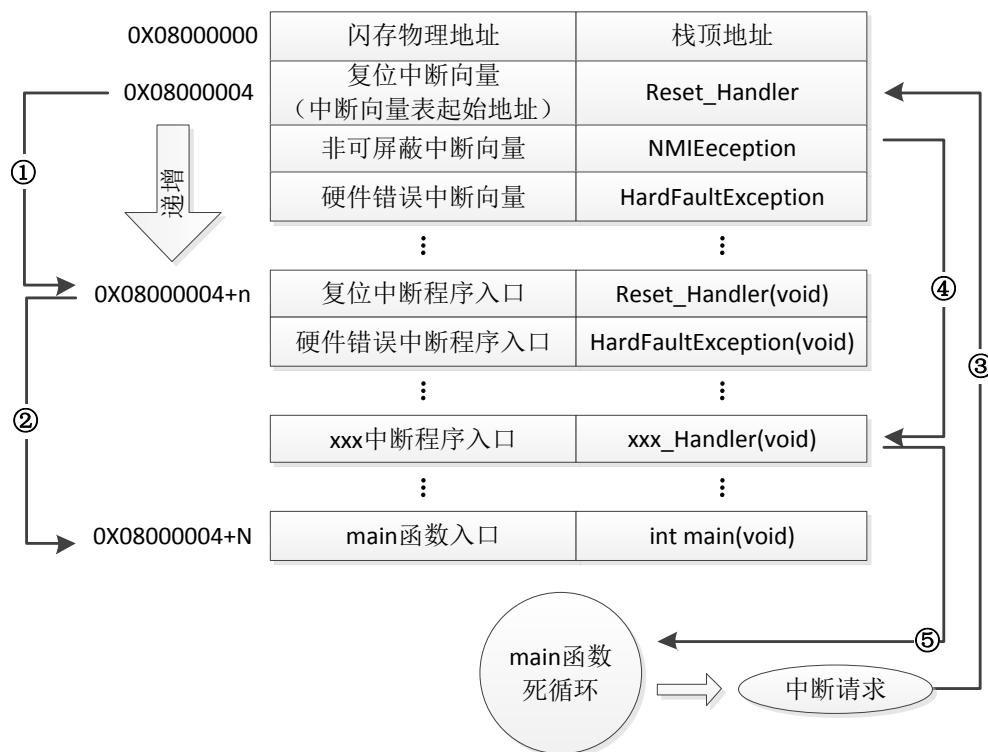


图 55.1.1 STM32F4 正常运行流程图

STM32F4 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 STM32F4 是基于 Cortex-M4 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，STM32F4 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 55.1.1 中，STM32F4 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 STM32F4 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 55.1.2 所示：

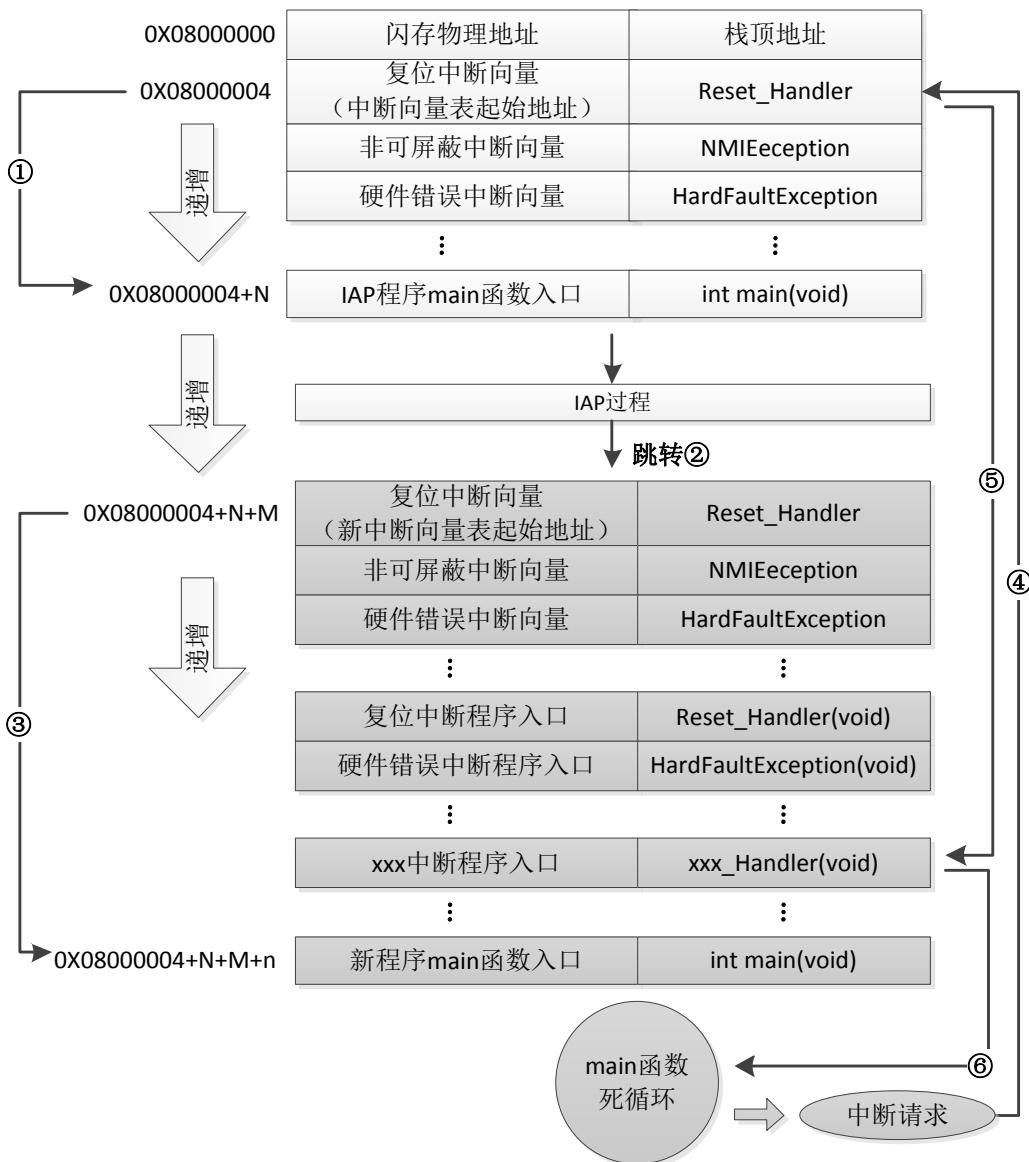


图 55.1.2 加入 IAP 之后程序运行流程图

在图 55.1.2 所示流程中，STM32F4 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 55.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32F4 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 STM32F4 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, 另外一个位 SRAM 的 APP, 图 55.1.2 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 55.1.3 所示:

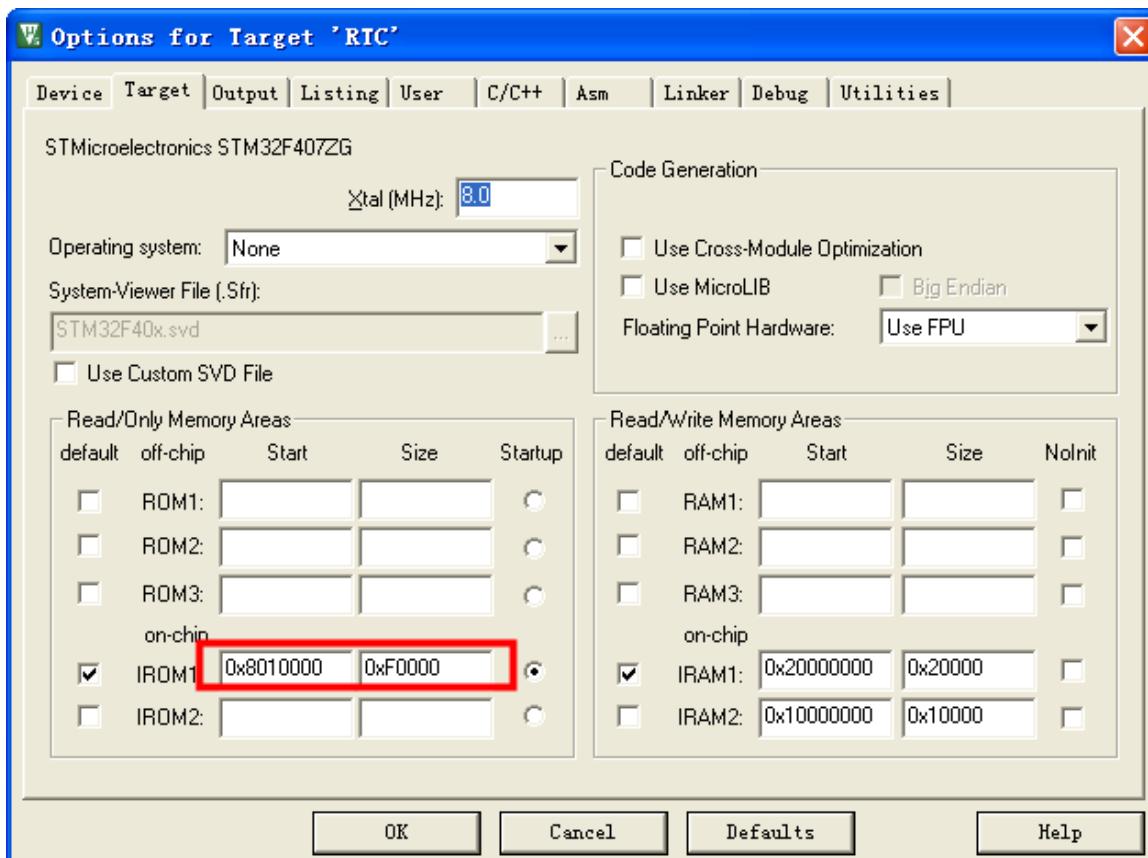


图 55.1.3 FLASH APP Target 选项卡设置

默认的条件下, 图中 IROM1 的起始地址(Start)一般为 0X08000000, 大小(Size)为 0X100000, 即从 0X08000000 开始的 1024K 空间为我们的程序存储区。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 0X100000-0X10000=0XF0000 (960K 字节) 大小了。设置好 Start 和 Size, 就完成 APP 程序的起始地址设置。

这里的 64K 字节, 需要大家根据 Bootloader 程序大小进行选择, 比如我们本章的 Bootloader 程序为 36K 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.openedv.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 55.1.4 所示:

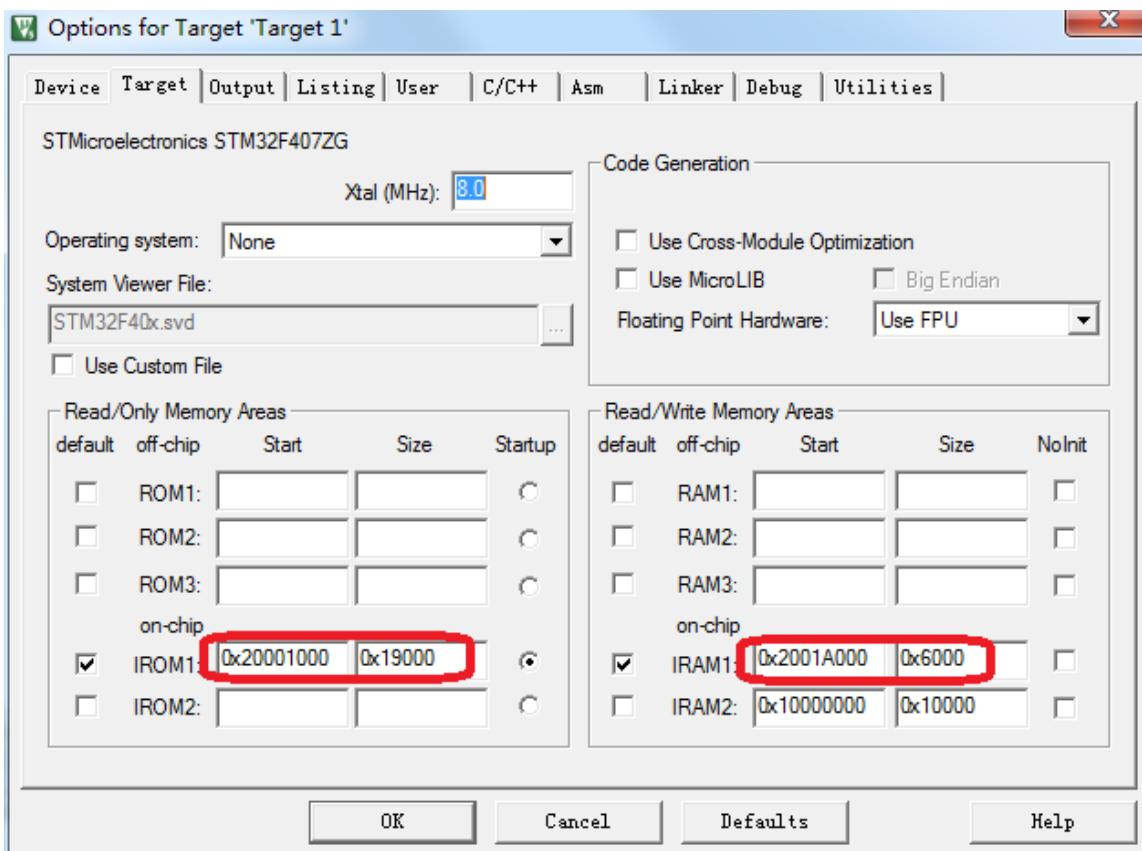


图 55.1.4 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址 (Start) 定义为: 0X20001000，大小为 0X19000 (100K 字节)，即从地址 0X20000000 偏移 0X1000 开始，存放 APP 代码。因为整个 STM32F407ZGT6 的 SRAM 大小 (不算 CCM) 为 128K 字节，所以 IRAM1 (SRAM) 的起始地址变为 0X2001A000，大小只有 0X6000 (24K 字节)。这样，整个 STM32F407ZGT6 的 SRAM (不含 CCM) 分配情况为：最开始的 4K 给 Bootloader 程序使用，随后的 100K 存放 APP 程序，最后 24K，用作 APP 程序的内存。这个分配关系大家可以根据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。默认的情况下 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 SystemInit 函数

里面修改 SCB->VTOR 的值。当然为了尽可能不修改系统级别文件，我们可以也可以在 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.doc》。

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，并写入：D:\MDK5.11A\ARM\ARMCC\bin\fromelf.exe --bin -o ..\OBJ\TEST.bin ..\OBJ\TEST.axf，如图 55.1.5 所示：

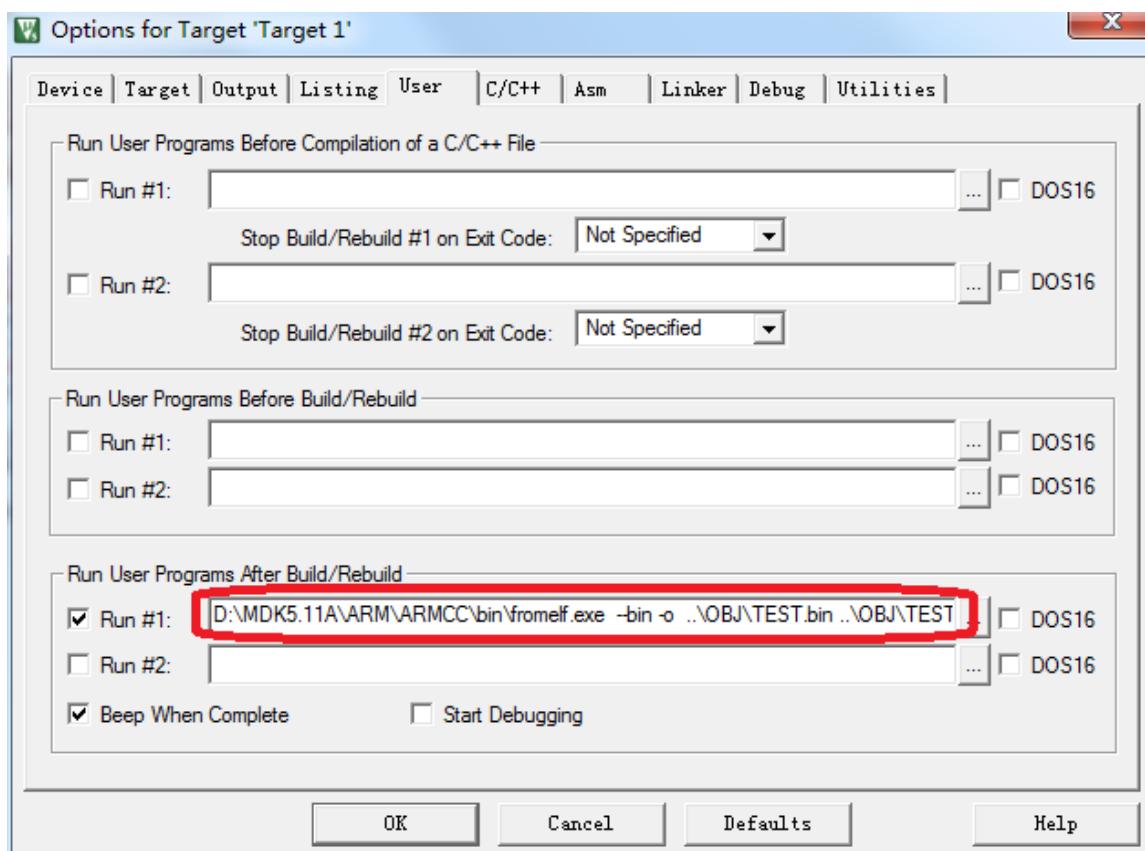


图 55.1.5 MDK 生成.bin 文件设置方法

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:\MDK5.11A 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，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。

最后再来看看 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，我们只需要设置 APP 程序的起始地址，和存储空间大小即可。而对于在 SRAM 里面运行的 APP 程序，我们还需要设置 SRAM 的起始地址和大小。无论哪种 APP 程序，都需要确保 APP 程序的大小和所占 SRAM 大小不超过我们的设置范围。

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。

55.2 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 KEY_UP 按键，将串口接收到的 APP 程序存放到 STM32F4 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/KEY_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。

55.3 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（在第三十三章介绍）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 55.1 节的介绍，自行理解。本章软件设计仅针对 Bootloader 程序。

复制第十八章的工程（即实验 13），作为本章的工程模版（命名为：IAP Bootloader V1.0），并复制第三十九章实验（FLASH 模拟 EEPROM 实验）的 STMFLASH 文件夹到本工程的 HARDWARE 文件夹下，打开本实验工程，并将 STMFLASH 文件夹内的 stmflash.c 加入到 HARDWARE 组下，同时将 STMFLASH 加入头文件包含路径。

在 HARDWARE 文件夹所在的文件夹下新建一个 IAP 的文件夹，并在该文件夹下新建 iap.c 和 iap.h 两个文件。然后在工程里面新建一个 IAP 的组，将 iap.c 加入到该组下面。最后，将 IAP 文件夹加入头文件包含路径。

打开 iap.c，输入如下代码：

```
iapfun jump2app;
u32 iapbuf[512]; //2K 字节缓存
//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)
{
    u32 t; u16 i=0; u32 temp;
    u32 fwaddr=appxaddr;//当前写入的地址
    u8 *dfu=appbuf;
    for(t=0;t<appsize;t+=4)
    {
        temp=(u32)dfu[3]<<24;
        temp|=(u32)dfu[2]<<16;
        temp|=(u32)dfu[1]<<8;
        temp|=(u32)dfu[0];
        dfu+=4;//偏移 4 个字节
        iapbuf[i++]=temp;
        if(i==512)
        {
            i=0;
            STMFLASH_Write(fwaddr,iapbuf,512);
            fwaddr+=2048;//偏移 2048 512*4=2048
        }
    }
    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。

保存 iap.c，打开 iap.h 输入如下代码：

```
#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 0x08010000  
//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(共 64KB)  
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，保存 iap.h。本章，我们是通过串口接收 APP 程序的，我们将 usart.c 和 usart.h 做了稍微修改，在 usart.h 中，我们定义 USART_REC_LEN 为 120K 字节，也就是串口最大一次可以接收 120K 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。在 usart.c 里面，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  
u32 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)  
    {  
        Res =USART_ReceiveData(USART1); //读取接收到的数据  
  
        if(USART_RX_CNT<USART_REC_LEN)  
        {  
            USART_RX_BUF[USART_RX_CNT]=Res;  
            USART_RX_CNT++;  
        }  
    } #ifdef OS_CRITICAL_METHOD
```

```
//如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
```

```
OSIntExit();  
#endif  
}
```

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。

改完 usart.c 和 usart.h 之后, 我们在 main.c 修改 main 函数如下:

```
int main(void)  
{  
    u8 t; u8 key; u8 clearflag=0;  
    u16 oldcount=0; //老的串口接收数据值  
    u32 applength=0; //接收到的 app 代码长度  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2  
    delay_init(168); //初始化延时函数  
    uart_init(460800); //初始化串口波特率为 115200  
    LED_Init(); //初始化 LED  
    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(30,50,200,16,16,"Explorer STM32F4");  
    LCD_ShowString(30,70,200,16,16,"IAP TEST");  
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(30,110,200,16,16,"2014/7/21");  
    LCD_ShowString(30,130,200,16,16,"KEY_UP:Copy APP2FLASH");  
    LCD_ShowString(30,150,200,16,16,"KEY1:Erase SRAM APP");  
    LCD_ShowString(30,170,200,16,16,"KEY0:Run SRAM APP");  
    LCD_ShowString(30,190,200,16,16,"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,没收到数据,认为本次接收完成.  
            {  
                applength=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",applength);  
            }else oldcount=USART_RX_CNT;
```

```
        }
        t++;
        delay_ms(10);
        if(t==30)
        {
            LED0=!LED0;  t=0;
            if(clearflag)
            {
                clearflag--;
                if(clearflag==0)LCD_Fill(30,210,240,210+16,WHITE);//清除显示
            }
        }
        key=KEY_Scan(0);
        if(key==WKUP_PRES) //WK_UP 按键按下
        {
            if(applenth)
            {
                printf("开始更新固件...\r\n");
                LCD_ShowString(30,210,200,16,16,"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(30,210,200,16,16,"Copy APP Successed!!");
                    printf("固件更新完成!\r\n");
                }
            }
            else
            {
                LCD_ShowString(30,210,200,16,16,"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(30,210,200,16,16,"No APP!");
        }
        clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
    }
    if(key==KEY1_PRES) //KEY1 按下
    {
        if(applenth)
        {
            printf("固件清除完成!\r\n");
        }
    }
}
```

```
LCD_ShowString(30,210,200,16,16,"APP Erase Successed!");
applenth=0;
}
{
    printf("没有可以清除的固件!\r\n");
    LCD_ShowString(30,210,200,16,16,"No APP!");
}
clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}
if(key==KEY2_PRES) //KEY2 按下
{
    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(30,210,200,16,16,"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==KEY0_PRES) //KEY0 按下
{
    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(30,210,200,16,16,"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），实际应用时，可以尽量精简代码来得到最小的 IAP。本章例程我们仅作演示用，所以不对代码做任何精简，最后得到工程截图如图 55.3.1 所示：

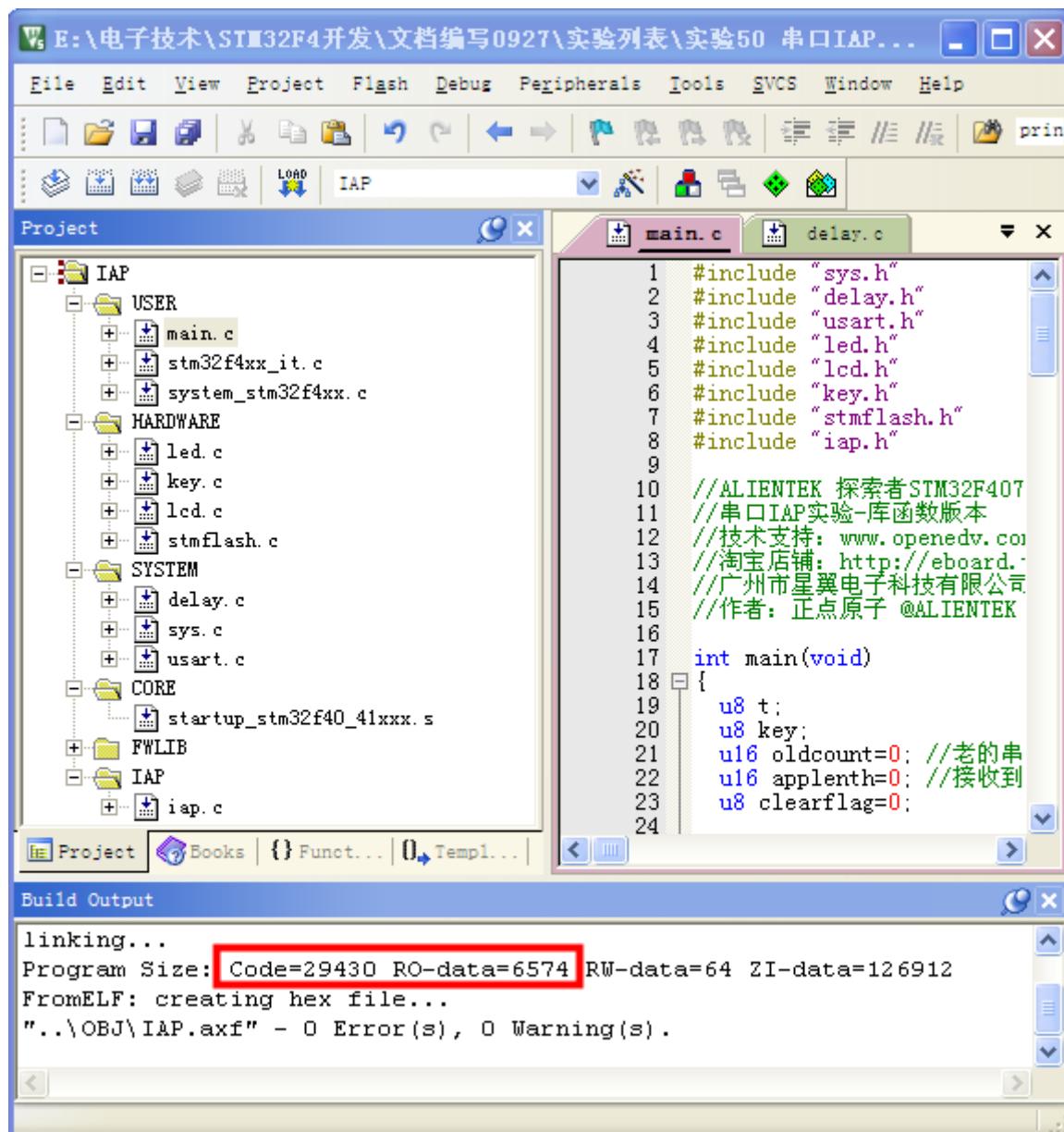


图 55.3.1 Bootloader 工程截图

从上图可以看出，Bootloader 大小为 36K 左右，比较大，主要是因为液晶驱动和 printf 占用了比较多的 flash，如果大家想删减代码，可以去掉不用的 LCD 部分代码和 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，根据 55.1 节的介绍，大家自行修改都比较简单，我们这里就不介绍了，不过要提醒大家：FLASH APP 的起始地址必须是 0X08010000，而 SRAM APP 的起始地址必须是 0X20001000。

55.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 探索者 STM32F4 开发板上，得到，如图 55.4.1 所示：

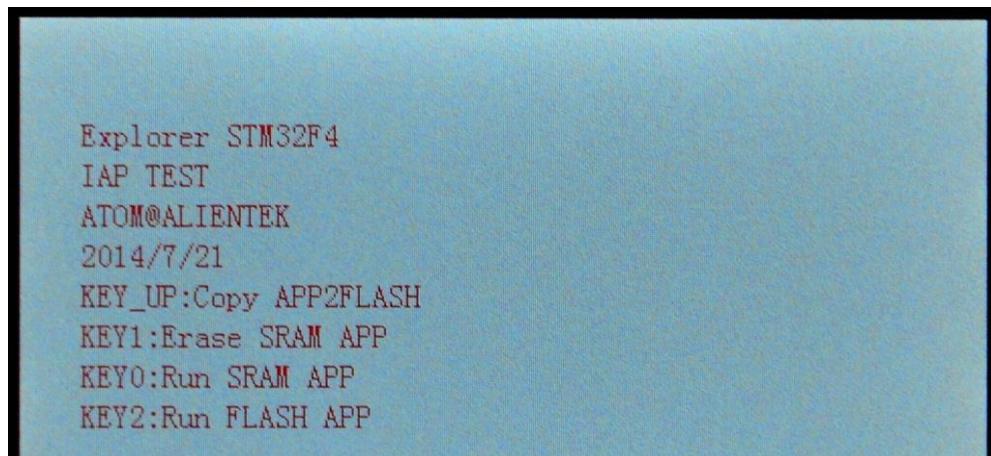


图 55.4.1 IAP 程序界面

此时，我们可以通过串口，发送 FLASH APP 或者 SRAM APP 到探索者 STM32F4 开发板，如图 55.4.2 所示：

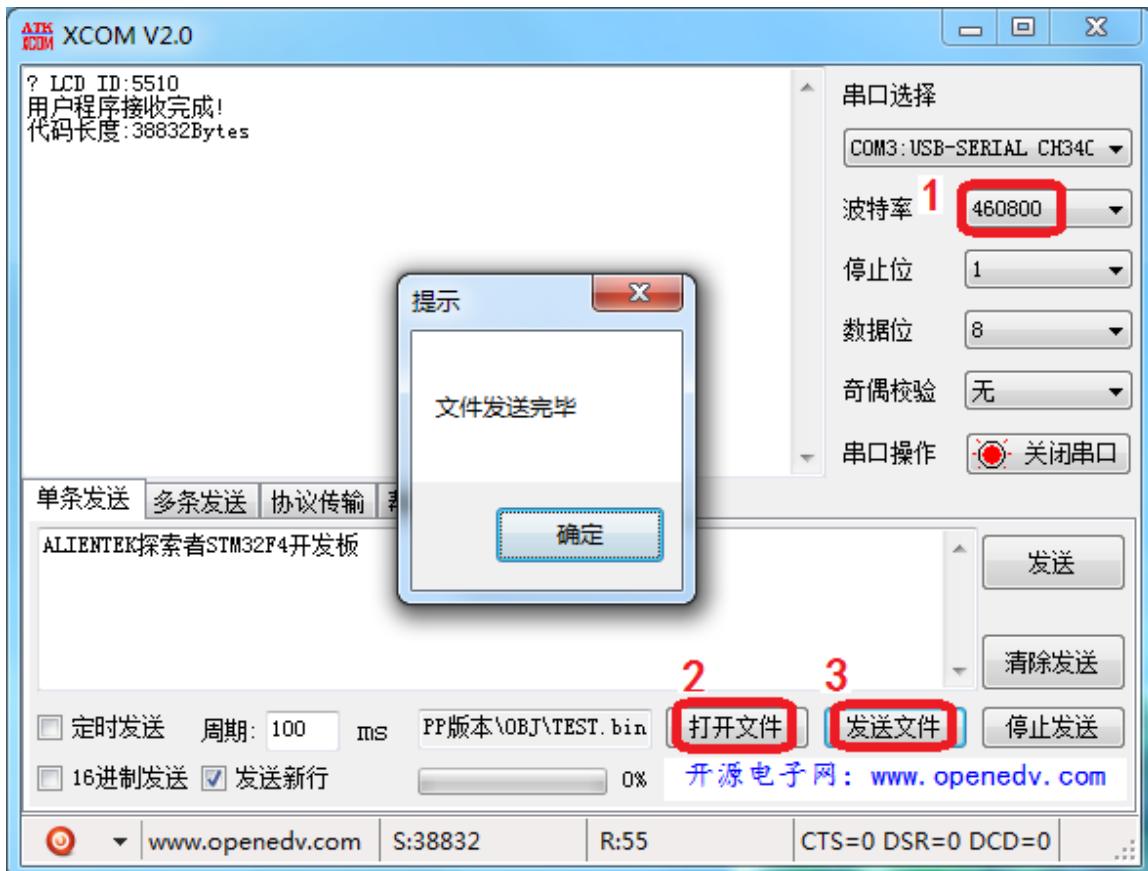


图 55.4.2 串口发送 APP 程序界面

首先找到开发板 USB 转串口的串口号，打开串口（我电脑是 COM3），然后设置波特率为 460800（图中标号 1 所示），然后，点击打开文件按钮（如图标号 2 所示），找到 APP 程序生成的.bin 文件（注意：文件类型得选择所有文件！！默认是只打开 txt 文件的），最后点击发送文件（图中标号 3 所示），将.bin 文件发送给探索者 STM32F4 开发板，发送完成后，XCOM 会提示文件发送完毕。

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 KEY_UP 将其存入对应 FLASH 区域）。

第五十六章 USB 读卡器(Slave)实验

STM32F407 系列芯片都自带了 USB OTG FS 和 USB OTG HS (HS 需要外扩高速 PHY 芯片实现, 速度可达 480Mbps), 支持 USB Host 和 USB Device, 探索者 STM32F4 开发板没有外扩高速 PHY 芯片, 仅支持 USB OTG FS (FS, 即全速, 12Mbps), 所有 USB 相关例程, 均使用 USB OTG FS 实现。

本章, 我们将向大家介绍如何利用 USB OTG FS 在 ALIENTEK 探索者 STM32F4 开发板实现一个 USB 读卡器。本章分为如下几个部分:

56.1 USB 简介

56.2 硬件设计

56.3 软件设计

56.4 下载验证

56.1 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。STM32F407 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+和 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 的电阻到地的, 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。

STM32F407 系列芯片自带 USB OTG FS (全速) 和 USB OTG HS (高速), 其中 HS 需要外扩高速 PHY 芯片实现, 我们这里不做介绍。

STM32F407 的 USB OTG FS 是一款双角色设备 (DRD) 控制器, 同时支持从机功能和主机功能, 完全符合 USB 2.0 规范的 On-The-Go 补充标准。此外, 该控制器也可配置为“仅主机”模式或“仅从机” 模式, 完全符合 USB 2.0 规范。在主机模式下, OTG FS 支持全速 (FS, 12 Mb/s) 和低速 (LS, 1.5 Mb/s) 收发器, 而从机模式下则仅支持全速 (FS, 12 Mb/s) 收发器。OTG FS 同时支持 HNP 和 SRP。

STM32F407 的 USB OTG FS 主要特性可分为三类: 通用特性、主机模式特性和从机模式特性。

1, 通用特性

- 经 USB-IF 认证, 符合通用串行总线规范第 2.0 版
- 集成全速 PHY, 且完全支持定义在标准规范 OTG 补充第 1.3 版中的 OTG 协议
 - 1, 支持 A-B 器件识别 (ID 线)
 - 2, 支持主机协商协议(HNP)和会话请求协议(SRP)
 - 3, 允许主机关闭 VBUS 以便在 OTG 应用中节省电池电量
 - 4, 支持通过内部比较器对 VBUS 电平采取监控

5, 支持主机到从机的角色动态切换

➤ 可通过软件配置为以下角色:

1, 具有 SRP 功能的 USB FS 从机 (B 器件)

2, 具有 SRP 功能的 USB FS/LS 主机 (A 器件)

3, USB On-The-Go 全速双角色设备

➤ 支持 FS SOF 和 LS Keep-alive 令牌

1, SOF 脉冲可通过 PAD 输出

2, SOF 脉冲从内部连接到定时器 2 (TIM2)

3, 可配置的帧周期

3, 可配置的帧结束中断

➤ 具有省电功能, 例如在 USB 挂起期间停止系统、关闭数字模块时钟、对 PHY 和 DFIFO 电源加以管理

➤ 具有采用高级 FIFO 控制的 1.25 KB 专用 RAM

1, 可将 RAM 空间划分为不同 FIFO, 以便灵活有效地使用 RAM

2, 每个 FIFO 可存储多个数据包

3, 动态分配存储区

4, FIFO 大小可配置为非 2 的幂次方值, 以便连续使用存储单元

➤ 一帧之内可以无需要应用程序干预, 以达到最大 USB 带宽

2, 主机 (Host) 模式特性

➤ 通过外部电荷泵生成 VBUS 电压。

➤ 多达 8 个主机通道 (管道): 每个通道都可以动态实现重新配置, 可支持任何类型的 USB 传输。

➤ 内置硬件调度器可:

1, 在周期性硬件队列中存储多达 8 个中断加同步传输请求

2, 在非周期性硬件队列中存储多达 8 个控制加批量传输请求

➤ 管理一个共享 RX FIFO、一个周期性 TX FIFO 和一个非周期性 TX FIFO, 以有效使用 USB 数据 RAM。

3, 从机 (Slave/Device) 模式特性

➤ 1 个双向控制端点 0

➤ 3 个 IN 端点 (EP), 可配置为支持批量传输、中断传输或同步传输

➤ 3 个 OUT 端点 (EP), 可配置为支持批量传输、中断传输或同步传输

➤ 管理一个共享 Rx FIFO 和一个 Tx-OUT FIFO, 以高效使用 USB 数据 RAM

➤ 管理多达 4 个专用 Tx-IN FIFO (分别用于每个使能的 IN EP), 降低应用程序负载支持软断开功能。

STM32F407 USB OTG FS 框图如图 56.1.1 所示:

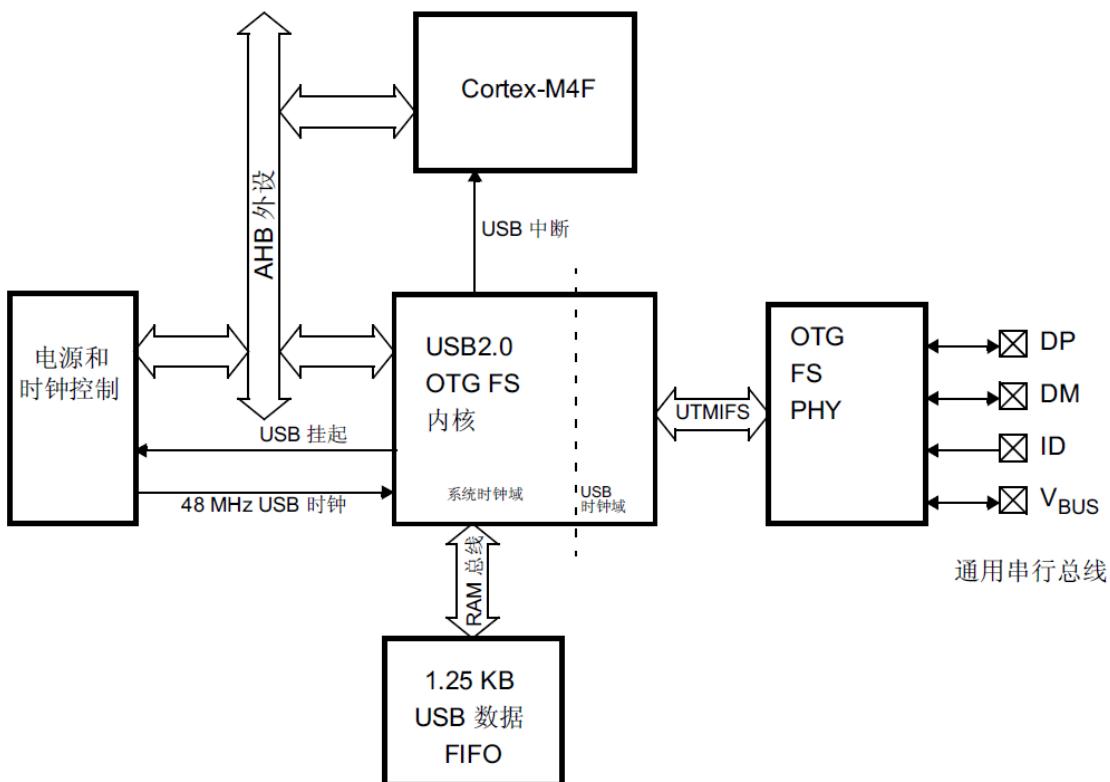


图 56.1.1 USB OTG 框图

对于 USB OTG FS 功能模块，STM32F4 通过 AHB 总线访问（AHB 频率必须大于 14.2Mhz），其中 48Mhz 的 USB 时钟，是来自时钟树图里面的 PLL48CK（和 SDIO 共用）。

STM32F4 USB OTG FS 的其他介绍，请大家参考《STM32F4xx 中文参考手册》第 30 章内容，我们这里就不再详细介绍了。

要正常使用 STM32F4 的 USB，就得编写 USB 驱动，而整个 USB 通信的详细过程是很复杂的，本书篇幅有限，不可能在这里详细介绍，有兴趣的朋友可以去看看电脑圈圈的《圈圈教你玩 USB》这本书，该书对 USB 通信有详细讲解。如果要我们自己编写 USB 驱动，那是一件相当困难的事情，尤其对于从没了解过 USB 的人来说，基本上不花个一两年时间学习，是没法搞定的。不过，ST 提供了我们一个完整的 USB OTG 驱动库（包括主机和设备），通过这个库，我们可以很方便的实现我们所要的功能，而不需要详细了解 USB 的整个驱动，大大缩短了我们的开发时间和精力。

ST 提供的 USB OTG 库，可以在：<http://www.stmcu.org/download/index.php?act=ziliao&id=150> 这里下载到 (UM1021)。不过，我们已经帮大家下载到开发板光盘：8，STM32 参考资料 → STM32 USB 学习资料，文件名：stm32_f105-07_f2_f4_usb-host-device_lib.zip。该库包含了 STM32F4 USB 主机 (Host) 和从机 (Device) 驱动库，并提供了 10 个例程供我们参考，如图 56.1.2 所示：

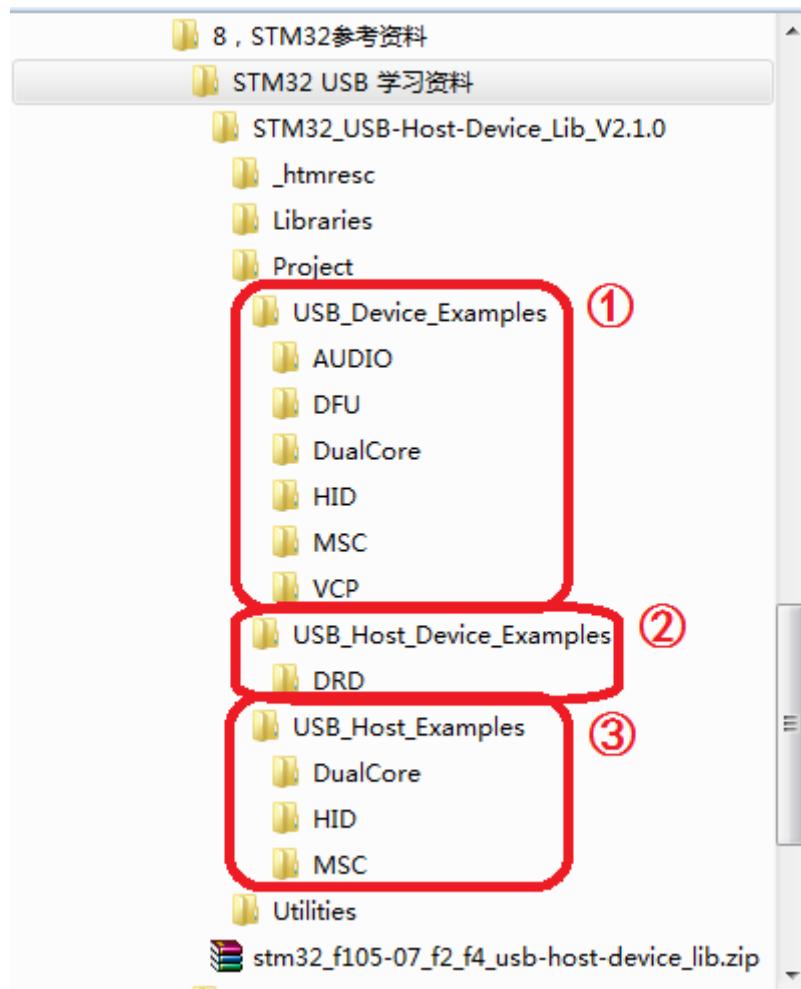


图 56.1.2 ST 提供的 USB OTG 例程

如图 56.1.2 所示, ST 提供了 3 类例程: ①即设备类 (Device, 即 Slave)、②主从一体类 (Host_Device) 和③主机类 (Host), 总共 10 个例程。整个 USB OTG 库还有一个说明文档: CD00289278.pdf (在光盘有提供), 即 UM1021, 该文档详细介绍了 USB OTG 库的各个组成部分以及所提供的例程使用方法, 有兴趣学习 USB 的朋友, 这个文档是必须仔细看的。

这 10 个例程, 虽然都是基于官方 EVAL 板的, 但是很容易移植到我们的探索者 STM32F407 开发板上, 本章我们就是移植: STM32_USB-Host-Device_Lib_V2.1.0\Project\USB_Device_Examples\MSC 这个例程, 以实现 USB 读卡器功能。

56.2 硬件设计

本章实验功能简介: 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, 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 SLAVE 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 SLAVE 连接口。ALIENTEK 探索者 STM32F4 开发板采用的是 5PIN 的 MiniUSB 接头, 用来和电脑的 USB 相连接, 连接电路如图 56.2.1 所示:

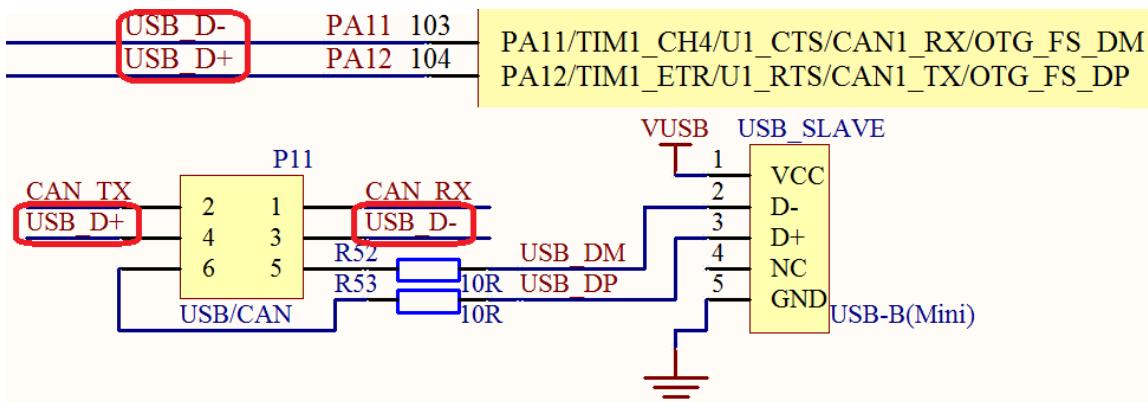


图 56.2.1 MiniUSB 接口与 STM32 的连接电路图

从上图可以看出, USB 座没有直接连接到 STM32F4 上面, 而是通过 P11 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 56.2.2 所示:

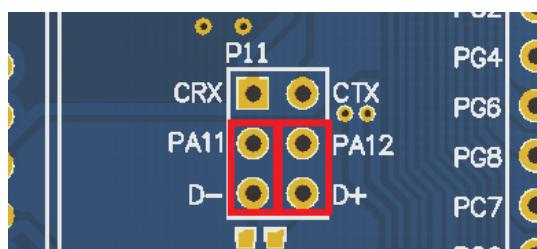


图 56.2.2 硬件连接示意图

不过这个 MiniUSB 座和 USB-A 座 (USB_HOST) 是共用 D+ 和 D- 的, 所以他们不能同时使用。这个在使用的时候, 要特别注意!! 本实验测试时, USB_HOST 不能插入任何 USB 设备!

56.3 软件设计

本章, 我们在: 实验 38 SD 卡实验 的基础上修改, 代码移植自 ST 官方例程: STM32_USB-Host-Device_Lib_V2.1.0\Project\USB_Device_Examples\MSC, 我打开该例程即可知道 USB 相关的代码有哪些, 如图 56.3.1 所示:

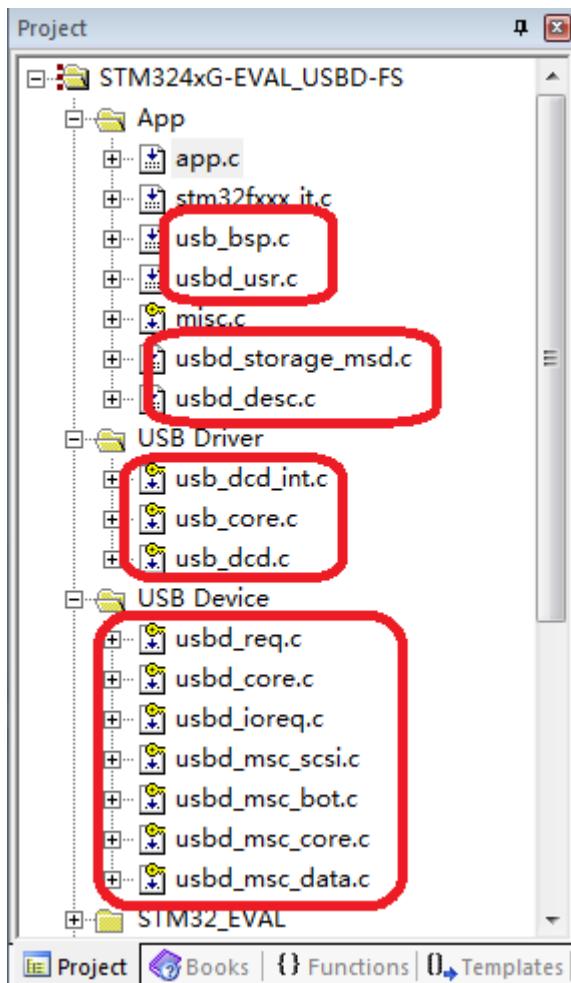


图 56.3.1 ST 官方例程 USB 相关代码

有了这个官方例程做指引，我们就知道具体需要哪些文件，从而实现本章例程。

首先，在本章例程（即实验 38 SD 卡实验）的工程文件夹下面，新建 USB 文件夹，并拷贝官方 USB 驱动库相关代码到该文件夹下，即拷贝：光盘→8, STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.1.0→Libraries 文件夹下的 STM32_USB_Device_Libray、STM32_USB_HOST_Library 和 STM32_USB_OTG_Driver 等三个文件夹的源码到该文件夹下面。

然后，在 USB 文件夹下，新建 USB_APP 文件夹存放 MSC 实现相关代码，即：STM32_USB-Host-Device_Lib_V2.1.0→Project→USB_Device_Examples→MSC→src 下的部分代码：usb_bsp.c、usbd_storage_msdf.c、usbd_desc.c 和 usbd_usr.c 等 4 个.c 文件，同时拷贝 STM32_USB-Host-Device_Lib_V2.1.0→Project→USB_Device_Examples→MSC→inc 下面的：usb_conf.h、usbd_conf.h 和 usbd_desc.h 等三个文件到 USB_APP 文件夹下，最后 USB_APP 文件夹下的文件如图 56.3.2 所示：

名称	修改日期	类型	大小
usb_bsp.c	2014/9/2 13:57	C 文件	3 KB
usb_conf.h	2014/6/12 16:22	H 文件	10 KB
usbd_conf.h	2014/9/1 18:46	H 文件	3 KB
usbd_desc.c	2014/6/12 17:16	C 文件	9 KB
usbd_desc.h	2012/3/22 15:44	H 文件	4 KB
usbd_storage_msdu.c	2014/9/1 18:41	C 文件	5 KB
usbd_usr.c	2014/9/1 21:20	C 文件	3 KB

图 56.3.2 USB_APP 代码

之后，根据 ST 官方 MSC 例程，在我们本章例程的基础上新建分组添加相关代码，具体细节，这里就不详细介绍了，添加好之后，如图 56.3.3 所示：

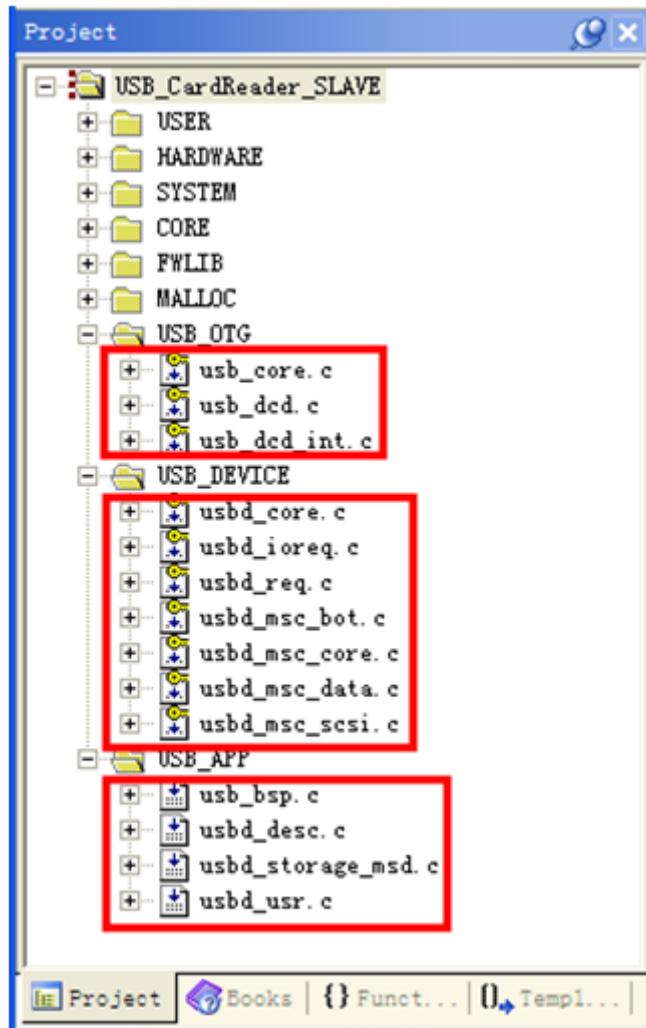


图 56.3.3 添加 USB 驱动等相关代码

移植时，我们重点要修改的就是 USB_APP 文件夹下面的代码。其他代码（USB_OTG 和

USB_DEVICE 文件夹下的代码) 一般不用修改。

usb_bsp.c 提供了几个 USB 库需要用到的底层初始化函数, 包括: IO 设置、中断设置、VBUS 配置以及延时函数等, 需要我们自己实现。USB Device (Slave) 和 USB Host 共用这个.c 文件。

usbd_desc.c 提供了 USB 设备类的描述符, 直接决定了 USB 设备的类型、断点、接口、字符串、制造商等重要信息。这个里面的内容, 我们一般不用修改, 直接用官方的即可。注意, 这里: usbd_desc.c 里面的: usbd 即 device 类, 同样: usbh 即 host 类, 所以通过文件名我们可以很容易区分该文件是用在 device 还是 host, 而只有 usb 字样的那就是 device 和 host 可以共用的。

usbd_usr.c 提供用户应用层接口函数, 即 USB 设备类的一些回调函数, 当 USB 状态机处理完不同事务的时候, 会调用这些回调函数, 我们通过这些回调函数, 就可以知道 USB 当前状态, 比如: 是否枚举成功了? 是否连接上了? 是否断开了? 等, 根据这些状态, 用户应用程序可以执行不同操作, 完成特定功能。

usbd_storage_msdc.c 提供一些磁盘操作函数, 包括支持的磁盘个数, 以及每个磁盘的初始化和读写等函数。本章我们设置了 2 个磁盘: SD 卡和 SPI FLASH。

以上 4 个.c 文件里面的函数, 基本上都是以回调函数的形式, 被 USB 驱动库调用的。这些代码的具体修改过程, 我们这里不详细介绍, 请大家参考光盘本例程源码, 这里只提几个重点地方讲解下:

1, 要使用 USB OTG FS, 必须在 MDK 编译器的全局宏定义里面, 定义: USE_USB_OTG_FS 宏, 如图 56.3.4 所示:

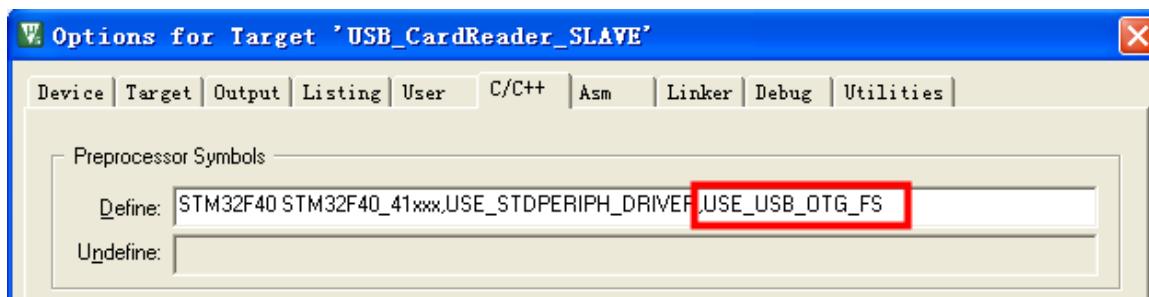


图 56.3.4 定义全局宏 USE_USB_OTG_FS

2, 因为探索者 STM32F407 开发板没有用到 VUSB 电压检测, 所以要在 usb_conf.h 里面, 将宏定义: #define VBUS_SENSING_ENABLED, 屏蔽掉。

3, 通过修改 usbd_conf.h 里面的 MSC_MEDIA_PACKET 定义值大小, 可以一定程度提高 USB 读写速度 (越大越快), 本例程我们设置 12*1024, 也就是 12K 大小。

4, 官方例程不支持大于 4G 的 SD 卡, 得修改 usbd_msc_scsi.c 里面的 SCSI_blk_addr 类型为 uint64_t, 才可以支持大于 4G 的卡, 官方默认是 uint32_t, 最大只能支持 4G 卡。注意: usbd_msc_scsi.c 文件, 是只读的, 得先修改属性, 去掉只读属性, 才可以更改。

以上 4 点, 就是我们移植的时候需要特别注意的, 其他我们就不详细介绍了 (USB 相关源码解释, 请参考: CD00289278.pdf 这个文档), 最后修改 main.c 里面代码如下:

```
USB_OTG_CORE_HANDLE USB_OTG_dev;
extern vu8 USB_STATUS_REG;      //USB 状态
extern vu8 bDeviceState;        //USB 连接 情况
int main(void)
{
    u8 offline_cnt=0; u8 tct=0;
```

```
u8 Divece_STA; u8 USB_STA;
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
delay_init(168); //初始化延时函数
uart_init(115200); //初始化串口波特率为 115200
LED_Init(); //初始化 LED
LCD_Init(); //LCD 初始化
KEY_Init(); //按键初始化
W25QXX_Init(); //初始化 W25Q128
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,70,200,16,16,"USB Card Reader TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2014/7/21");
if(SD_Init())LCD_ShowString(30,130,200,16,16,"SD Card Error!"); //检测 SD 卡错误
else //SD 卡正常
{
    LCD_ShowString(30,130,200,16,16,"SD Card Size:      MB");
    LCD_ShowNum(134,130,SDCardInfo.CardCapacity>>20,5,16); //显示 SD 卡容量
}
if(W25QXX_ReadID()!=W25Q128)
LCD_ShowString(30,130,200,16,16,"W25Q128 Error!"); //检测 W25Q128 错误
else LCD_ShowString(30,150,200,16,16,"SPI FLASH Size:12MB"); //SPI FLASH 正常
LCD_ShowString(30,170,200,16,16,"USB Connecting..."); //提示正在建立连接
USBD_Init(&USB_OTG_dev,USB_OTG_FS_CORE_ID,&USR_desc,&USBD_MSC_cb,
&USR_cb);
delay_ms(1800);
while(1)
{
    delay_ms(1);
    if(USB_STA!=USB_STATUS_REG)//状态改变了
    {
        LCD_Fill(30,190,240,190+16,WHITE); //清除显示
        if(USB_STATUS_REG&0x01)//正在写
        {
            LED1=0;
            LCD_ShowString(30,190,200,16,16,"USB Writing..."); //USB 正在写数据
        }
        if(USB_STATUS_REG&0x02)//正在读
        {
            LED1=0;
            LCD_ShowString(30,190,200,16,16,"USB Reading..."); //USB 正在读数据
        }
        if(USB_STATUS_REG&0x04)
```

```

LCD_ShowString(30,210,200,16,16,"USB Write Err");//提示写入错误
else LCD_Fill(30,210,240,210+16,WHITE);//清除显示
if(USB_STATUS_REG&0x08)
LCD_ShowString(30,230,200,16,16,"USB Read Err");//提示读出错误
else LCD_Fill(30,230,240,230+16,WHITE);//清除显示
USB_STA=USB_STATUS_REG;//记录最后的状态
}
if(Divece_STA!=bDeviceState)
{
    if(bDeviceState==1)LCD_ShowString(30,170,200,16,16,"USB Connected ");
    else LCD_ShowString(30,170,200,16,16,"USB DisConnected");//USB 被拔出了
    Divece_STA=bDeviceState;
}
tct++;
if(tct==200)
{
    tct=0; LED1=1;
    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=1;
    }else//没有得到轮询
    {
        offline_cnt++;
        if(offline_cnt>10)bDeviceState=0;//2s 内没收到在线标记,则 USB 被拔出了
    }
    USB_STATUS_REG=0;
}
};

}

```

其中，USB_OTG_CORE_HANDLE 是一个全局结构体类型，用于存储 USB 通信中 USB 内核需要使用的各种变量、状态和缓存等，任何 USB 通信（不论主机，还是从机），我们都必须定义这么一个结构体以实现 USB 通信，这里定义成：USB_OTG_dev。

然后，USB 初始化非常简单，只需要调用 USBD_Init 函数即可，顾名思义，该函数是 USB 设备类初始化函数，本章的 USB 读卡器属于 USB 设备类，所以使用该函数。该函数初始化了 USB 设备类处理的各种回调函数，以便 USB 驱动库调用。执行完该函数以后，USB 就启动了，所有 USB 事务，都是通过 USB 中断触发，并由 USB 驱动库自动处理。USB 中断服务函数在 usbd_usr.c 里面：

```

//USB OTG 中断服务函数 处理所有 USB 中断
void OTG_FS_IRQHandler(void)
{
    USBD_OTG_ISR_Handler(&USB_OTG_dev);
}

```

}

该函数调用 USBD_OTG_ISR_Handler 函数来处理各种 USB 中断请求。因此在 main 函数里面，我们的处理过程就非常简单，main 函数里面通过两个全局状态变量（USB_STATUS_REG 和 bDeviceState），来判断 USB 状态，并在 LCD 上面显示相关提示信息。

USB_STATUS_REG 在 usbd_storage_msd.c 里面定义的一个全局变量，不同的位表示不同状态，用来指示当前 USB 的读写等操作状态。

bDeviceState 是在 usbd_usr.c 里面定义的一个全局变量，0 表示 USB 还没有连接；1 表示 USB 已经连接。

软件设计部分，就给大家介绍到这里。

56.4 下载验证

在代码编译成功之后，我们下载到探索者 STM32F4 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_SLAVE 口！不是 USB_232 端口！另外，USB_HOST 接口，也不要插入任何设备，否则会干扰！！），LCD 显示效果如图 56.4.1 所示：

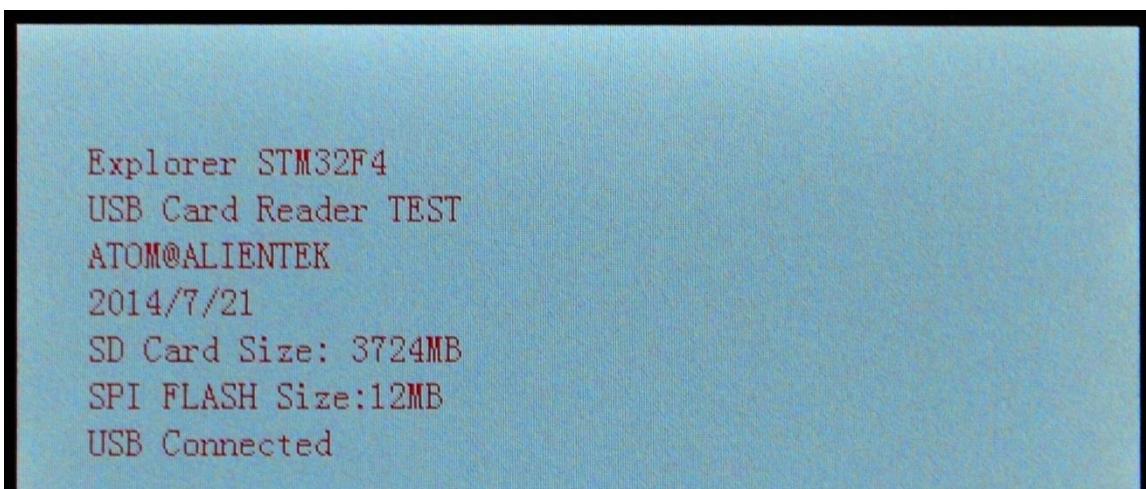


图 56.4.1 USB 连接成功

此时，电脑提示发现新硬件，并开始自动安装驱动，如图 56.4.2 所示：

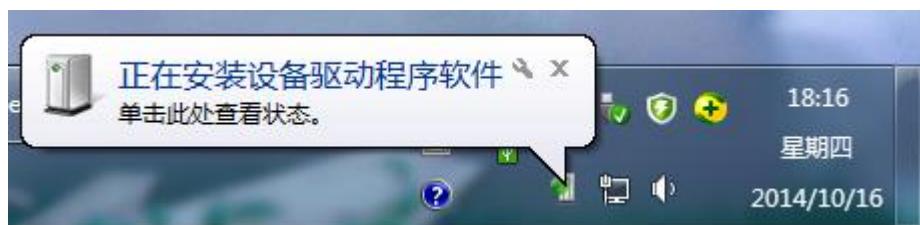


图 56.4.2 USB 读卡器被电脑找到

等 USB 配置成功后，DS1 不亮，DS0 闪烁，并且在电脑上可以看到我们的磁盘，如图 56.4.3 所示：

▲ 有可移动存储的设备 (4)

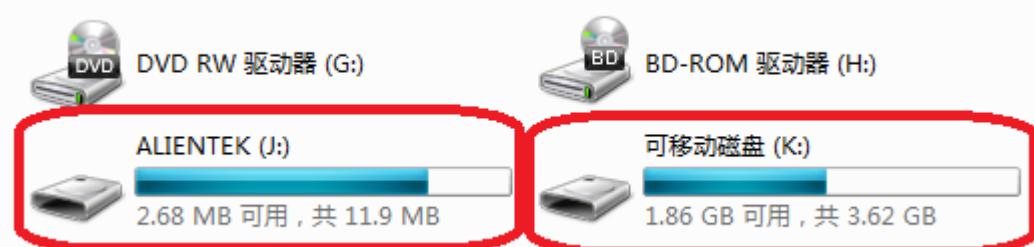


图 56.4.3 电脑找到 USB 读卡器的两个盘符

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 大容量存储设备，同时看到磁盘驱动器里面多了 2 个磁盘，如图 56.4.4 所示：

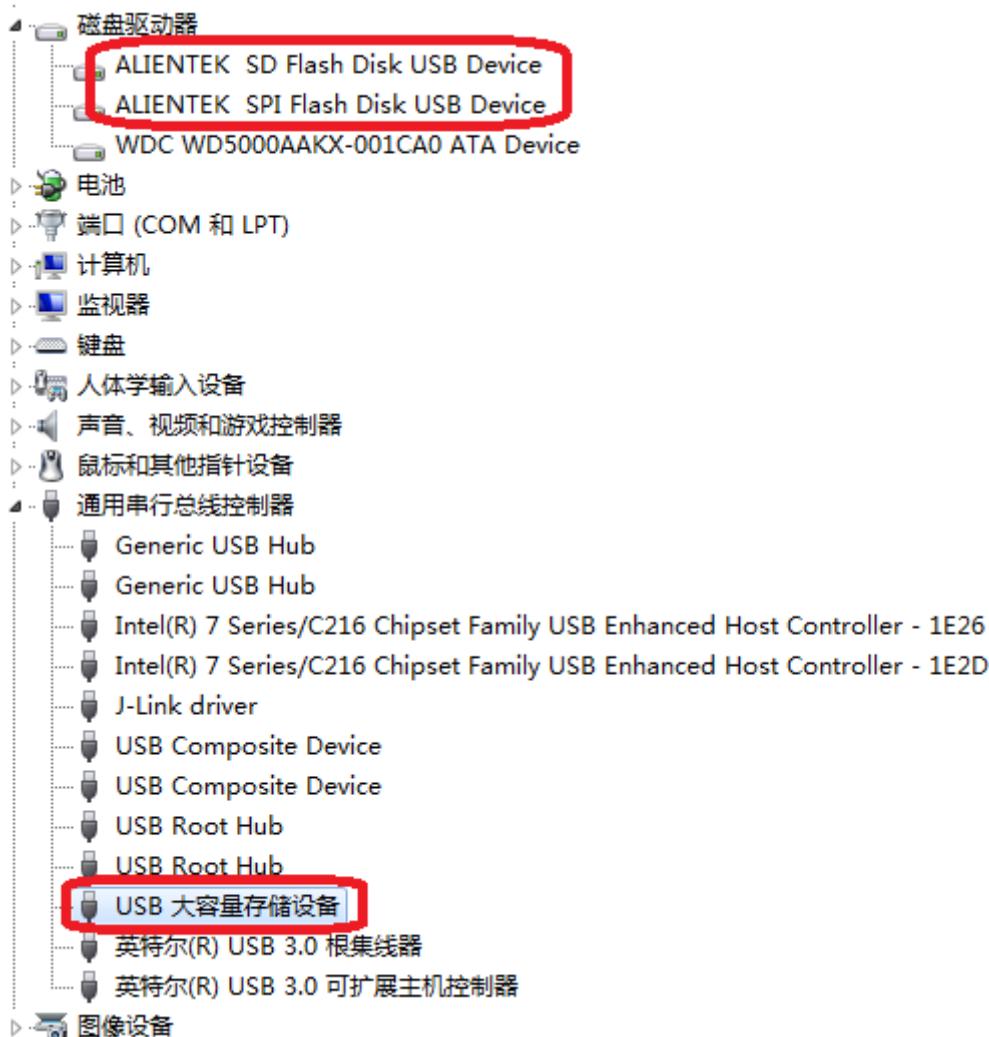


图 56.4.4 通过设备管理器查看磁盘驱动器

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！！

第五十七章 USB 声卡(Slave)实验

上一章我们向大家介绍了如何利用 STM32F4 的 USB 接口来做一个 USB 读卡器,本章我们将利用 STM32F4 的 USB 来做一个声卡。本章分为以下几个部分:

- 57.1 USB 声卡简介
- 57.2 硬件设计
- 57.3 软件设计
- 57.4 下载验证

57.1 USB 声卡简介

ALIENTEK 探索者 STM32F4 开发板板载了一颗高性能 CODEC 芯片: WM8978, 我们可以利用 STM32F4 的 IIS, 控制 WM8978 播放音乐, 同样, 如果结合 STM32F4 的 USB 功能, 就可以实现一个 USB 声卡。

同上一章一样, 我们直接移植官方的 USB AUDIO 例程, 官方例程路径: 8, STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.1.0→Project→USB_Device_Examples→AUDIO, 该例程采用 USB 同步传输来传输音频数据流并且支持某些控制命令(比如静音控制), 例程仅支持 USB FS 模式(不支持 HS), 同时例程不需要特殊的驱动支持, 大多数操作系统直接就可以识别。

57.2 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(或者喇叭接 P1 (SPK) 端子也行), 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 SLAVE 接口
- 5) WM8978

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。这里再次提醒大家, P11 的连接, 要通过跳线帽连接 PA11 和 D-以及 PA12 和 D+。

57.3 软件设计

本章, 我们在第四十八章实验 (实验 43) 的基础上修改, 先打开实验 43 的工程, 在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹, 同上一章一样, 对照官方 AUDIO 例子, 将相关文件拷贝到 USB 文件夹下。

然后, 我们在工程里面去掉一些不必要的代码, 并添加 USB 相关代码, 最终得到如图 57.3.1 所示的工程:

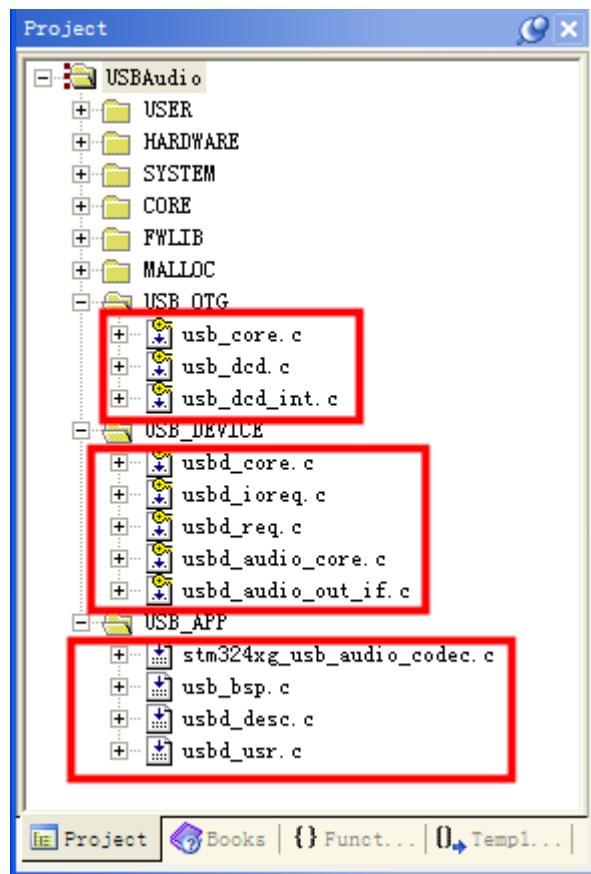


图 57.3.1 USB 声卡工程截图

可以看到，USB 部分代码，同上一章的在结构上是一模一样的，只是.c 文件稍微有些变化。同样，我们移植需要修改的代码，就是 USB_APP 里面的这四个.c 文件了。

其中 usb_bsp.c 和 usbd_usr.c 的代码，和上一章基本一样，可以用上一章的代码直接替换即可正常使用。

usb_desc.c 代码，同上一章不一样，上一章描述符是大容量存储设备，本章变成了 USB 声卡了，所以直接用 ST 官方的就行。

最后 stm324xg_usb_audio_codec.c，这里面的代码，是重点要修改的，该文件是配合 USB 声卡的 CS43L22 底层驱动相关代码，官方 STM32F4xG 的板子，用的是 CS43L22，而我们用的是 WM8978，所以这里面代码要大改，修改后代码如下：

```

u8 volume=0;                                //当前音量
vu8 audiostatus=0;                           //bit0:0,暂停播放;1,继续播放
vu8 i2splaybuf=0;                            //即将播放的音频帧缓冲编号
vu8 i2ssavebuf=0;                            //当前保存到的音频缓冲编号
#define AUDIO_BUF_NUM    100      //由于采用的是 USB 同步传输数据播放
//而 STM32 IIS 的速度和 USB 传送过来数据的速度存在差异,
//比如在 48Khz 下,实际 IIS 是低
//于 48Khz(47.991Khz)的,所以电脑送过来的数据流,会比 STM32 播放速度快,缓冲区写位置
//追上播放位置(i2ssavebuf==i2splaybuf)时,就会出现混叠.设置尽量大的 AUDIO_BUF_NUM
//值,可以尽量减少混叠次数.
u8 *i2sbuf[AUDIO_BUF_NUM];
//音频缓冲帧,占用内存数=AUDIO_BUF_NUM*AUDIO_OUT_PACKET 字节

```

```
//音频数据 I2S DMA 传输回调函数
void audio_i2s_dma_callback(void)
{
    if((i2splaybuf==i2ssavebuf)&&audiostatus==0) I2S_Play_Stop();
    else
    {
        i2splaybuf++; //指向下一个 buf
        if(i2splaybuf>(AUDIO_BUF_NUM-1))i2splaybuf=0; //溢出
        if(DMA1_Stream4->CR&(1<<19))
            DMA_MemoryTargetConfig(DMA1_Stream4,(u32)i2sbuf[i2splaybuf],DMA_Memory_0);
        else
            DMA_MemoryTargetConfig(DMA1_Stream4,(u32)i2sbuf[i2splaybuf],DMA_Memory_1);
    }
}

//配置音频接口
//OutputDevice:输出设备选择,未用到.
//Volume:音量大小,0~100
//AudioFreq:音频采样率
uint32_t EVAL_AUDIO_Init(uint16_t OutputDevice, uint8_t Volume, uint32_t AudioFreq)
{
    u16 t=0;
    for(t=0;t<AUDIO_BUF_NUM;t++) //内存申请
    {
        i2sbuf[t]=mymalloc(SRAMIN,AUDIO_OUT_PACKET);
    }
    if(i2sbuf[AUDIO_BUF_NUM-1]==NULL) //内存申请失败
    {
        printf("Malloc Error!\r\n");
        for(t=0;t<AUDIO_BUF_NUM;t++)myfree(SRAMIN,i2sbuf[t]);
        return 1;
    }
    I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,
              I2S_DataFormat_16bextended);
    //飞利浦标准,主机发送,时钟低电平有效,16 位扩展帧长度
    I2S2_SampleRate_Set(AudioFreq); //设置采样率
    EVAL_AUDIO_VolumeCtl(Volume); //设置音量
    I2S2_TX_DMA_Init(i2sbuf[0],i2sbuf[1],AUDIO_OUT_PACKET/2);
    i2s_tx_callback=audio_i2s_dma_callback; //回调函数指 wav_i2s_dma_callback
    I2S_Play_Start(); //开启 DMA
    printf("EVAL_AUDIO_Init:%d,%d\r\n",Volume,AudioFreq);
    return 0;
}
//开始播放音频数据
```

```
// pBuffer:音频数据流首地址指针
//Size:数据流大小(单位:字节)
uint32_t EVAL_AUDIO_Play(uint16_t* pBuffer, uint32_t Size)
{
    printf("EVAL_AUDIO_Play:%x,%d\r\n",pBuffer,Size);
    return 0;
}

//暂停/恢复音频流播放
//Cmd:0,暂停播放;1,恢复播放
//Addr:音频数据流缓存首地址
//Size:音频数据流大小(单位:half word,也就是 2 个字节
//返回值:0,成功
//      其他,设置失败
uint32_t EVAL_AUDIO_PauseResume(uint32_t Cmd, uint32_t Addr, uint32_t Size)
{
    u16 i;u8 *p=(u8*)Addr;
    if(Cmd==AUDIO_PAUSE) audiostatus=0;
    else
    {
        audiostatus=1;
        i2ssavebuf++;
        if(i2ssavebuf>(AUDIO_BUF_NUM-1))i2ssavebuf=0;
        for(i=0;i<AUDIO_OUT_PACKET;i++) i2sbuf[i2ssavebuf][i]=p[i];//拷贝数据
        I2S_Play_Start(); //开启 DMA
    }
    return 0;
}

//停止播放
//Option:控制参数,1/2,详见:CODEC_PDWN_HW 定义
//返回值:0,成功
//      其他,设置失败
uint32_t EVAL_AUDIO_Stop(uint32_t Option)
{
    printf("EVAL_AUDIO_Stop:%d\r\n",Option);
    audiostatus=0;return 0;
}

//音量设置
//Volume:0~100
//返回值:0,成功
//      其他,设置失败
uint32_t EVAL_AUDIO_VolumeCtl(uint8_t Volume)
{
    volume=Volume;
```

```
WM8978_HPvol_Set(volume*0.63,volume*0.63);
WM8978_SPKvol_Set(volume*0.63);
return 0;
}

//静音控制
//Cmd:0,正常
//    1,静音
//返回值:0,正常
//    其他,错误代码
uint32_t EVAL_AUDIO_Mute(uint32_t Cmd)
{
    if(Cmd==AUDIO_MUTE_ON) {
        WM8978_HPvol_Set(0,0); WM8978_SPKvol_Set(0);}
    else
    {
        WM8978_HPvol_Set(volume*0.63,volume*0.63);
        WM8978_SPKvol_Set(volume*0.63);
    }
    return 0;
}

//播放音频数据流
//Addr:音频数据流缓存首地址
//Size:音频数据流大小(单位:half word,也就是 2 个字节)
void Audio_MAL_Play(uint32_t Addr, uint32_t Size)
{
    u16 i;  u8 t=i2ssavebuf;
    u8 *p=(u8*)Addr;
    u8 curplay=i2splaybuf; //目前正在播放的缓存帧编号
    if(curplay)curplay--;
    else curplay=AUDIO_BUF_NUM-1;
    audiostatus=1;
    t++;
    if(t>(AUDIO_BUF_NUM-1))t=0;
    if(t==curplay)      //写缓存碰上了目前正在播放的帧,跳到下一帧
    {
        t++;
        if(t>(AUDIO_BUF_NUM-1))t=0;
        printf("bad position:%d\r\n",t);
    }
    i2ssavebuf=t;
    for(i=0;i<Size*2;i++) i2sbuf[i2ssavebuf][i]=p[i];//拷贝数据
    I2S_Play_Start();      //开启 DMA
}
```

这里特别说明一下，USB AUDIO 我们使用的是 USB 同步数据传输，音频采样率固定为：48Khz（通过 USBD_AUDIO_FREQ 设置，在 usbd_conf.h 里面），这样，USB 传输过来的数据都是 48Khz 的音频数据流，STM32F4 必须以同样的频率传输数据给 IIS，以同步播放音乐。

但是，STM32F4 我们采用的是内部 8M 时钟倍频后分频作为 IIS 时钟的，在使能主时钟（MCK）输出的时候，只能以 47.991Khz 频率播放，稍微有点误差，这样，导致 USB 送过来的数据，会比传输给 IIS 的数据快一点点，如果不做处理，就很容易产生数据混叠，产生噪音。

因此，我们这里提供了一个简单的解决办法：建立一个类似 FIFO 结构的缓冲数组，USB 传输过来的数据全部存放在这些数组里面，同时通过 IIS DMA 双缓冲机制，播放这些数组里面的音频数据，当混叠发生时（USB 传过来的数据，赶上 IIS 播放的数据了），直接越过当前正在播放的数组，继续保存。这样，虽然会导致一些数据丢失（混叠时），但是避免了混叠，保证了良好的播放效果（听不到噪音），同时，数组个数越多，效果就越好（越不容易混叠）。

以上代码 AUDIO_BUF_NUM 就是我们定义的 FIFO 结构数组的大小，越大，效果越好，这里我们定义成 100，每个数组的大小由音频采样率和位数决定，计算公式为：

$$(\text{USBD_AUDIO_FREQ} * 2 * 2) / 1000$$

单位为字节，其中 USBD_AUDIO_FREQ 即音频采样率：48Khz，这样，每个数组大小就是 192 字节。100 个数组，我们总共用了 19200 字节。

audio_i2s_dma_callback 函数是 IIS 播放音频的回调函数，完成 IIS 数据流的发送，其他函数则基本都是在 usbd_audio_out_if.c 里面被调用，这里就不再详细介绍。

最后在 main.c 里面，我们修改 main 函数如下：

```
USB_OTG_CORE_HANDLE USB_OTG_dev;
extern vu8 bDeviceState;           //USB 连接 情况
extern u8 volume;                 //音量(可通过按键设置)
int main(void)
{
    u8 key; u8 t=0;u8 Divece_STA=0XFF;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init();          //初始化 LED
    usmart_dev.init(84); //初始化 USMART
    LCD_Init();          //LCD 初始化
    KEY_Init();          //按键初始化
    WM8978_Init();       //初始化 WM8978
    WM8978_ADDA_Cfg(1,0); //开启 DAC
    WM8978_Input_Cfg(0,0,0); //关闭输入通道
    WM8978_Output_Cfg(1,0); //开启 DAC 输出
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,70,200,16,16,"USB Sound Card TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2014/7/22");
```

```
LCD_ShowString(30,130,200,16,16,"KEY2:Vol- KEY0:vol+");  
POINT_COLOR=BLUE;//设置字体为蓝色  
LCD_ShowString(30,160,200,16,16,"VOLUME:"); //音量显示  
LCD_ShowxNum(30+56,160,DEFAULT_VOLUME,3,16,0X80);//显示音量  
LCD_ShowString(30,180,200,16,16,"USB Connecting...");//提示正在建立连接  
USBD_Init(&USB_OTG_dev,USB_OTG_FS_CORE_ID,&USR_desc,&AUDIO_cb,  
          &USR_cb);  
while(1)  
{  
    key=KEY_Scan(1);//支持连接  
    if(key)  
    {  
        if(key==KEY0_PRES) //KEY0 按下,音量增加  
        {  
            volume++;  
            if(volume>100)volume=100;  
        }else if(key==KEY2_PRES)//KEY2 按下,音量减少  
        {  
            if(volume)volume--;  
            else volume=0;  
        }  
        EVAL_AUDIO_VolumeCtl(volume);  
        LCD_ShowxNum(30+56,160,volume,3,16,0X80);//显示音量  
        delay_ms(20);  
    }  
    if(Divece_STA!=bDeviceState)//状态改变了  
    {  
        if(bDeviceState==1)  
        {  
            LED1=0;  
            LCD_ShowString(30,180,200,16,16,"USB Connected "); //连接建立  
        }else  
        {  
            LED1=1;  
            LCD_ShowString(30,180,200,16,16,"USB DisConnected "); //连接失败  
        }  
        Divece_STA=bDeviceState;  
    }  
    delay_ms(20); t++;  
    if(t>10) {t=0; LED0=!LED0;}  
}  
}
```

此部分代码比较简单，同上一章一样定义了 USB_OTG_dev 结构体，然后通过 USBD_Init 初始化 USB，不过本章实现的是 USB 声卡功能。本章我们保留了原例程（实验 43）的 USMART 部分，同样可以通过串口 1 设置 WM8978 相关参数。

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。

57.4 下载验证

在代码编译成功之后，我们通过下载代码到探索者 STM32F4 开发板上，在 USB 配置成功后（注意：USB 数据线，要插在 USB_SLAVE 端口！不是 USB_232 端口！另外，USB_HOST 接口不要插任何外设！），LCD 显示效果如图 57.4.1 所示：

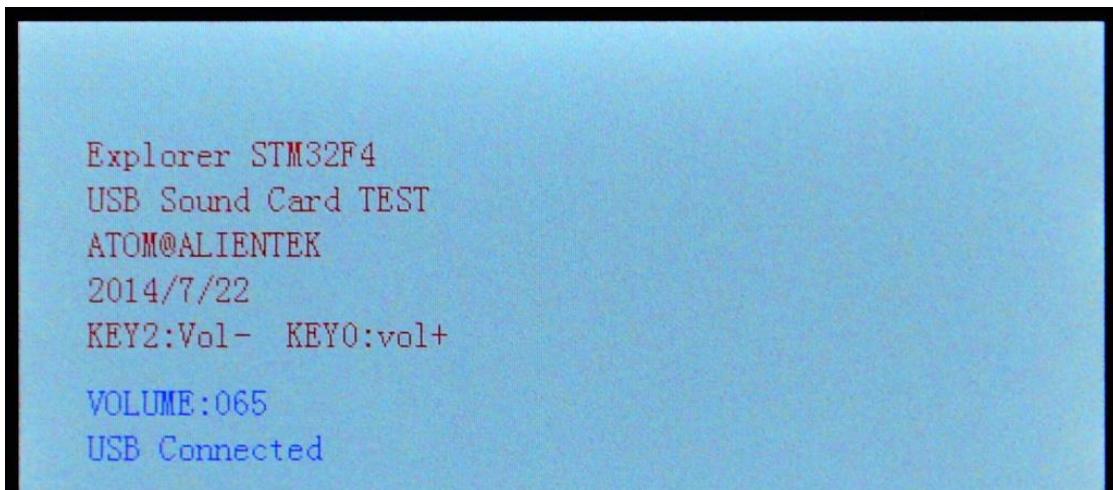


图 57.4.1 USB 连接成功

此时，电脑提示发现新硬件，并自动完成驱动安装，如图 57.4.2 所示：



图 57.4.2 电脑找到 ALIENTEK USB 声卡

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 ALIENTEK STM32F407 USB AUDIO 设备，如图 57.4.3 所示：

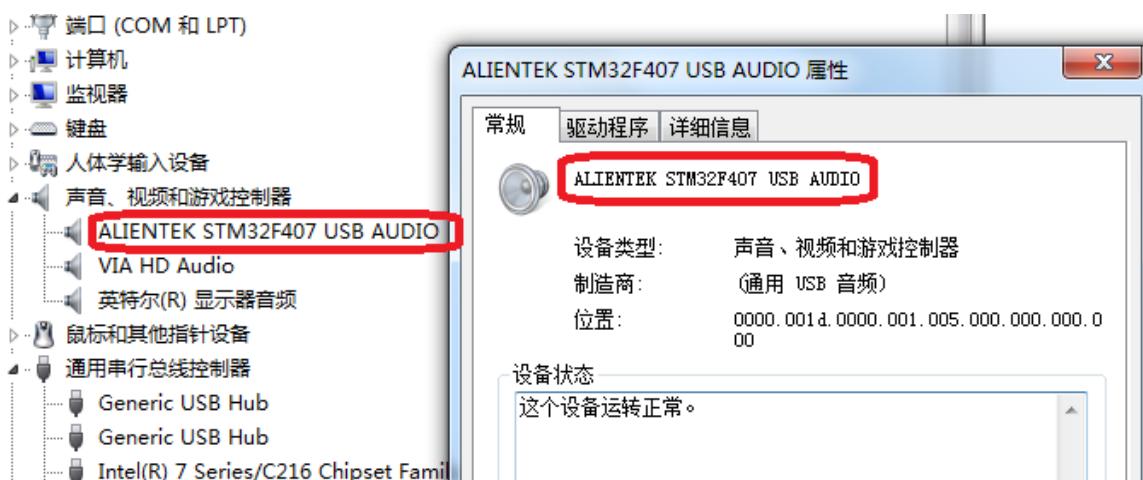


图 57.4.3 设备管理器找到 ALIENTEK USB 声卡

此时，电脑的所有音频输出都被切换到 USB 声卡输出，将耳机插入探索者 STM32F4 开发板的 PHONE 端口（或者接喇叭到 P1 端子（SPK）），即可听到来自电脑的声音。

通过按键 KEY0/KEY2 可以增大/减少音量，默认音量设置的是 65，大家可以自己调节（范围：0~100）。

第五十八章 USB U 盘(Host)实验

前面两章,我们介绍了 STM32F407 的 USB SLAVE 应用,本章我们介绍 STM32F407 的 USB HOST 应用,即通过 USB HOST 功能,实现读写 U 盘/读卡器等大容量 USB 存储设备。本章分为如下几个部分:

58.1 U 盘简介

58.2 硬件设计

58.3 软件设计

58.4 下载验证

58.1 U 盘简介

U 盘,全称 USB 闪存盘,英文名“USB flash disk”。它是一种使用 USB 接口的无需物理驱动器的微型高容量移动存储产品,通过 USB 接口与主机连接,实现即插即用,是最常用的移动存储设备之一。

STM32F4 的 USB OTG FS 支持 U 盘,并且 ST 官方提供了 USB HOST 大容量存储设备(MSC)例程,ST 官方例程路径:光盘 → 8, STM32 参考资料 → STM32 USB 学习资料 → STM32_USB-Host-Device_Lib_V2.1.0 → Project → USB_Host_Examples → MSC。本章代码,我们就要移植该例程到探索者 STM32F4 开发板上,以通过 STM32F4 的 USB HOST 接口,读写 U 盘或 SD 卡读卡器等设备。

58.2 硬件设计

本章实验功能简介:开机后,检测字库,然后初始化 USB HOST,并不断轮询。当检测并识别 U 盘后,在 LCD 上面显示 U 盘总容量和剩余容量,此时便可以通过 USMART 调用 FATFS 相关函数,来测试 U 盘数据的读写了,方法同 FATFS 实验一模一样。当 U 盘没插入的时候,DS0 闪烁,提示程序运行,当 U 盘插入后,DS1 闪烁,提示可以通过 USMART 测试了。

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 (非必须)
- 5) SPI FLASH
- 6) USB HOST 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 HOST 连接口。

ALIENTEK 探索者 STM32F4 开发板的 USB HOST 接口采用的是侧式 USB-A 座,它和 USB SLAVE 的 5PIN MiniUSB 接头是共用 USB_DM 和 USB_DP 信号的,所以 USB HOST 和 USB SLAVE 不能同时使用。

USB HOST 同 STM32F4 的连接原理图,如图 58.2.1 所示:

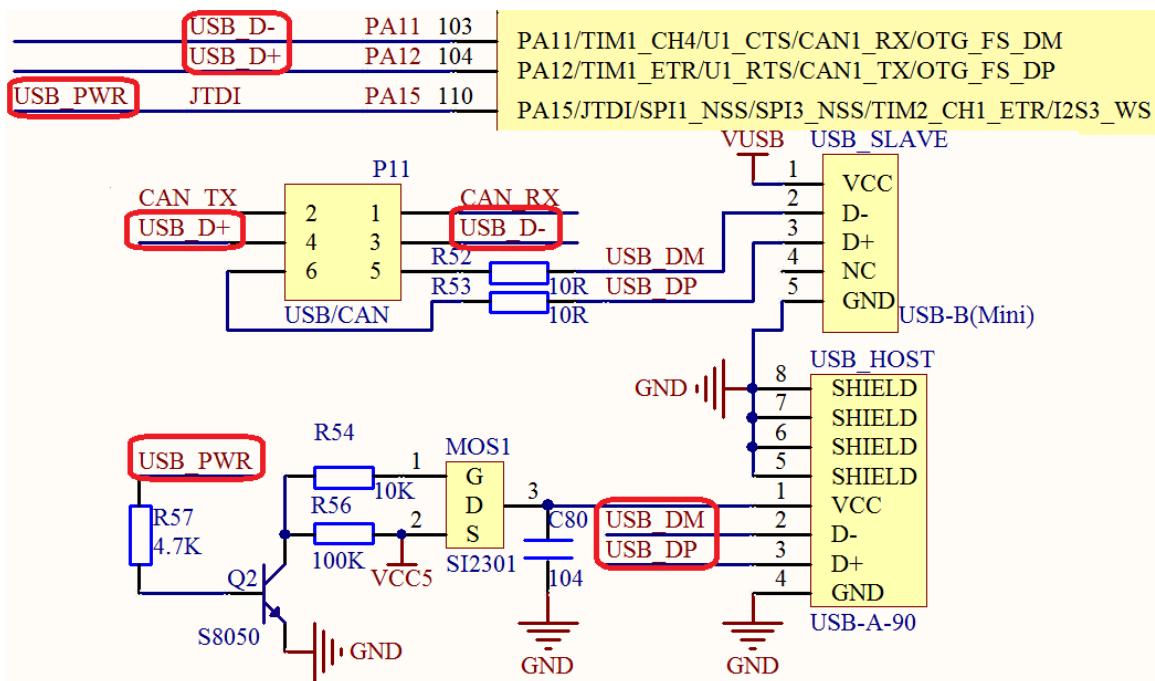


图 58.2.1 USB HOST 接口与 STM32F4 的连接原理图

从上图可以看出，USB_HOST 和 USB_SLAVE 共用 USB_DM/DP 信号，通过 P11 连接到 STM32F4。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 58.2.2 所示：

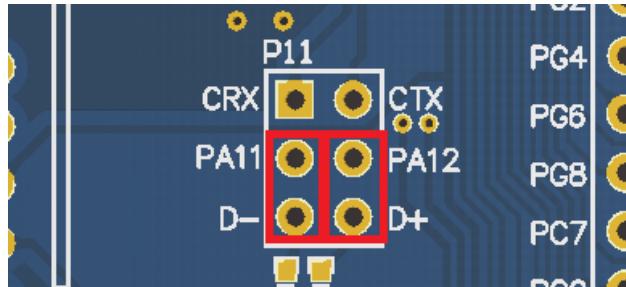


图 58.2.2 硬件连接示意图

图 58.2.1 中，我们还有一个 USB_PWR 的控制信号，用于控制给 USB 设备供电，该信号连接在 PA15 上面，和 JTAG 的 JTDX 信号共用，所以我们建议大家使用 SWD 模式调试，这样 PA15 就解放了，可以用于 USB_PWR 的控制。

使用 USB HOST 驱动外部 USB 设备的时候，必须要先控制 USB_PWR 输出 1，给外部设备供电，之后才可以识别到外部设备！

58.3 软件设计

本章，我们在：实验 41 图片显示实验 的基础上修改，代码移植自 ST 官方例程：STM32_USB-Host-Device_Lib_V2.1.0\Project\USB_Host_Examples\MSC，我打开该例程即可知道 USB 相关的代码有哪些，如图 58.3.1 所示：

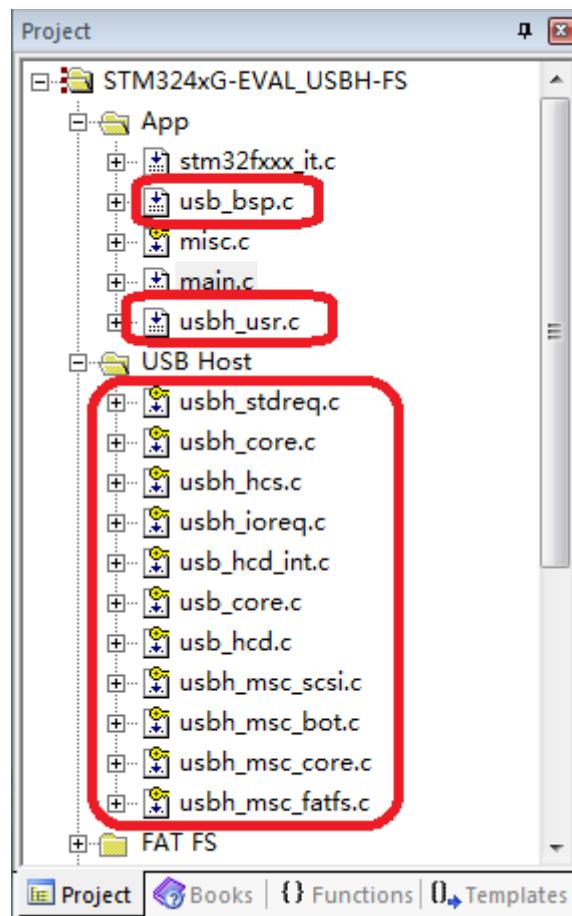


图 58.3.1 ST 官方例程 USB 相关代码

有了这个官方例程做指引，我们就知道具体需要哪些文件，从而实现本章例程。

从上图可以看出，这里并没有像第五十六章图 56.3.1 那样，区分不同分组，而是都放到 USB_Host 组下，看起来有点小乱，我们移植的时候，还是以 56 章的方式，分不同分组添加代码，方便阅读和管理。

这里面 usbh_msc_fatfs.c，是为了支持 fatfs 而写的一些底层接口函数，我们例程就直接放到 diskio.c 里面了，方便统一管理。

本例程的具体移植步骤，我们这里就不一一介绍了，最终移植好之后的工程截图，如图 58.3.2 所示：

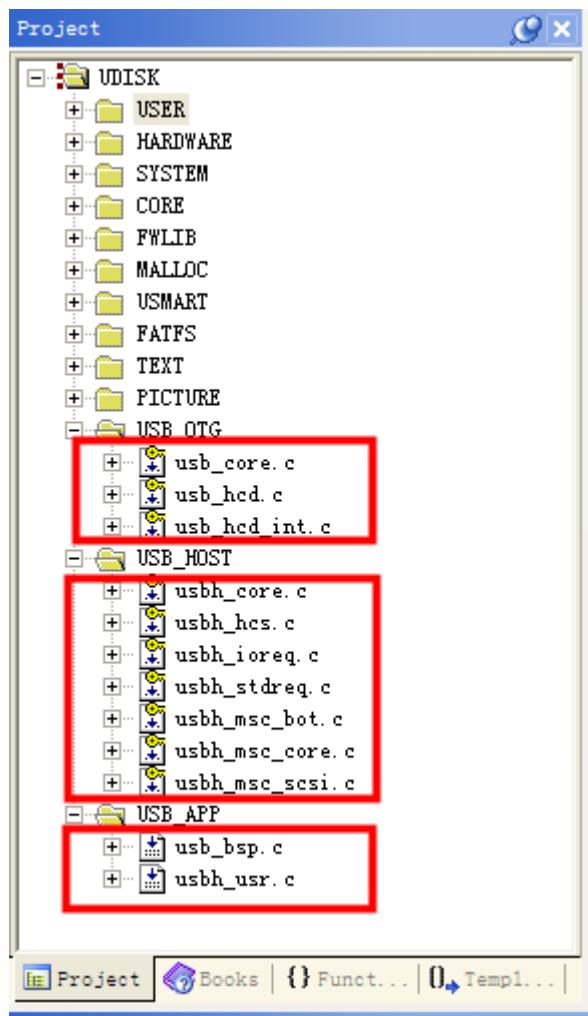


图 58.3.2 添加 USB 驱动等相关代码

移植时，我们重点要修改的就是 USB_APP 文件夹下面的代码。其他代码（USB_OTG 和 USB_HOST 文件夹下的代码）一般不用修改。

usb_bsp.c 的代码，和上一章的一样，可以用上一章的代码直接替换即可正常使用。

usbbh_usr.c 提供用户应用层接口函数，相比前两章例程，USB HOST 通信的回调函数更多一些，这里重点介绍 3 个函数，代码如下：

```

extern u8 USH_User_App(void);      //用户测试主程序
//USB HOST MSC 类用户应用程序
int USBH_USR_MSC_Application(void)
{
    u8 res=0;
    switch(AppState)
    {
        case USH_USR_FS_INIT://初始化文件系统
            printf("开始执行用户程序!!!\r\n");
            AppState=USH_USR_FS_TEST;
            break;
        case USH_USR_FS_TEST: //执行 USB OTG 测试主程序
    }
}

```

```
res=USH_User_App(); //用户主程序
res=0;
if(res)AppState=USH_USR_FS_INIT;
break;
default:break;
}
return res;
}

//用户定义函数,实现 fatfs diskio 的接口函数
extern USBH_HOST          USB_Host;
//读 U 盘
//buf:读数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 USBH_UDISK_Read(u8* buf,u32 sector,u32 cnt)
{
    u8 res=1;
    if(HCD_IsDeviceConnected(&USB_OTG_Core)&&AppState==USH_USR_FS_TEST)
        //连接还存在,且是 APP 测试状态
    {
        do
        {
            res=USBH_MSC_Read10(&USB_OTG_Core,buf,sector,512*cnt);
            USBH_MSC_HandleBOTXfer(&USB_OTG_Core ,&USB_Host);
            if(!HCD_IsDeviceConnected(&USB_OTG_Core))
            {
                res=1;//读写错误
                break;
            };
        }while(res==USBH_MSC_BUSY);
    }else res=1;
    if(res==USBH_MSC_OK)res=0;
    return res;
}
//写 U 盘
//buf:写数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 USBH_UDISK_Write(u8* buf,u32 sector,u32 cnt)
{
    u8 res=1;
```

```

if(HCD_IsDeviceConnected(&USB_OTG_Core)&&AppState==USH_USR_FS_TEST)
//连接还存在,且是 APP 测试状态
{
    do
    {
        res=USBH_MSC_Write10(&USB_OTG_Core,buf,sector,512*cnt);
        USBH_MSC_HandleBOTXfer(&USB_OTG_Core ,&USB_Host);
        if(!HCD_IsDeviceConnected(&USB_OTG_Core))
        {
            res=1;//读写错误
            break;
        };
    }while(res==USBH_MSC_BUSY);
}else res=1;
if(res==USBH_MSC_OK)res=0;
return res;
}

```

其中，`USBH_USR_MSC_Application` 函数通过状态机的方式，处理相关事务，执行到这个函数，说明 U 盘已经被成功识别了，此时用户可以执行一些自己想要做的事情，比如读取 U 盘文件什么的，这里我们直接进入到 `USH_User_App` 函数，执行各种处理，后续会介绍该函数。

`USBH_UDISK_Read` 和 `USBH_UDISK_Write` 这两个函数，用于 U 盘读写，从指定扇区地址读写指定个数的扇区数据，这两个函数，再配合 `fatfs`，即可实现对 U 盘的文件读写访问。

其他代码，我们就不详细讲解了，请大家参考光盘本例程源码，最后修改 `main.c` 里面代码如下：

```

USBH_HOST USB_Host;
USB_OTG_CORE_HANDLE USB_OTG_Core;
//用户测试主程序
//返回值:0,正常
//      1,有问题
u8 USH_User_App(void)
{
    u32 total,free;u8 res=0;
    Show_Str(30,140,200,16,"设备连接成功!",16,0);
    f_mount(fs[2],"2:",1); //重新挂载 U 盘
    res=exf_getfree("2:",&total,&free);
    if(res==0)
    {
        POINT_COLOR=BLUE;//设置字体为蓝色
        LCD_ShowString(30,160,200,16,16,"FATFS OK!");
        LCD_ShowString(30,180,200,16,16,"U Disk Total Size:      MB");
        LCD_ShowString(30,200,200,16,16,"U Disk  Free Size:      MB");
        LCD_ShowNum(174,180,total>>10,5,16); //显示 U 盘总容量 MB
        LCD_ShowNum(174,200,free>>10,5,16);
    }
}

```

```
        }
        while(HCD_IsDeviceConnected(&USB_OTG_Core))//设备连接成功，死循环
        {
            LED1=!LED1;
            delay_ms(200);
        }
        f_mount(0,"2:",1); //卸载 U 盘
        POINT_COLOR=RED;//设置字体为红色
        Show_Str(30,140,200,16,"设备连接中...",16,0);
        LCD_Fill(30,160,239,220,WHITE);
        return res;
    }
int main(void)
{
    u8 t;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化与 LED 连接的硬件接口
    KEY_Init(); //按键
    LCD_Init(); //初始化 LCD
    W25QXX_Init(); //SPI FLASH 初始化
    usmart_dev.init(84); //初始化 USMART
    my_mem_init(SRAMIN);//初始化内部内存池
    exfun_init(); //为 fatfs 相关变量申请内存
    piclib_init(); //初始化画图
    f_mount(fs[0],"0:",1); //挂载 SD 卡
    f_mount(fs[1],"1:",1); //挂载外部 SPI FLASH 盘
    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); delay_ms(200);//清除显示
    }
    Show_Str(30,50,200,16,"探索者 STM32F407 开发板",16,0);
    Show_Str(30,70,200,16,"USB U 盘实验",16,0);
    Show_Str(30,90,200,16,"2014 年 7 月 22 日",16,0);
    Show_Str(30,110,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,140,200,16,"设备连接中...",16,0);
    //初始化 USB 主机
    USBH_Init(&USB_OTG_Core,USB_OTG_FS_CORE_ID,&USB_Host,&USBH_MSC_cb
              ,&USR_Callbacks);
    while(1)
```

```
{  
    USBH_Process(&USB_OTG_Core, &USB_Host);  
    delay_ms(1);  
    t++;  
    if(t==200){ LED0=!LED0; t=0; }  
}  
}
```

相比 USB SLAVE 例程，我们这里多了一个 USB_HOST 的结构体定义：USB_Host，用于存储主机相关状态。所以，使用 USB 主机的时候，需要两个结构体：USB_OTG_CORE_HANDLE 和 USB_HOST。

然后，USB 初始化，使用的是 USBH_Init，用于 USB 主机初始化，包括对 USB 硬件和 USB 驱动库的初始化。如果是：USB SLAVE 通信，在只需要调用 USBD_Init 函数即可，不过 USB HOST 则还需要调用另外一个函数 USBH_Process，该函数用于实现 USB 主机通信的核心状态机处理，该函数必须在主函数里面，被循环调用，而且调用频率得比较快才行（越快越好），以便及时处理各种事务。注意，USBH_Process 函数仅在 U 盘识别阶段，需要频繁反复调用，但是当 U 盘被识别后，剩下的操作（U 盘读写），都可以由 USB 中断处理。

以上代码，main 函数十分简单，就不多做介绍了，这里主要看看 USH_User_App 函数，该函数前面有提到，是在 USBH_USR_MSC_Application 函数里面被调用，用于实现 U 盘插入后，用户想要实现的功能，一旦进入到该函数，即表示 U 盘已经成功识别了，所以，函数里面提示设备连接成功，挂载 U 盘（U 盘盘符为 2，0 代表 SD 卡，1 代表 SPI FLASH）并读取 U 盘总容量和剩余容量，显示在 LCD 上面，然后，进入死循环，只要 USB 连接一直存在，则一直死循环，同时控制 LED1 闪烁，提示 U 盘已经准备好了。

当 U 盘拔出来后，卸载 U 盘，然后再次提示设备连接中，会到 main 函数死循环，等待 U 盘再次连上。

最后，我们需要将 FATFS 相关测试函数(mf_open/ mf_close 等函数)，加入 USMART 管理，这里同第四十四章（FATFS 实验）一模一样，可以参考第四十四章的方法操作。

软件设计部分，就给大家介绍到这里。

58.4 下载验证

在代码编译成功之后，我们下载到探索者 STM32F4 开发板上，然后在 USB_HOST 端子插入 U 盘/读卡器（带卡），**注意：此时 USB SLAVE 口不要插 USB 线到电脑，否则会干扰！！**

等 U 盘成功识别后，便可以看到 LCD 显示 U 盘容量等信息，如图 58.4.1 所示：

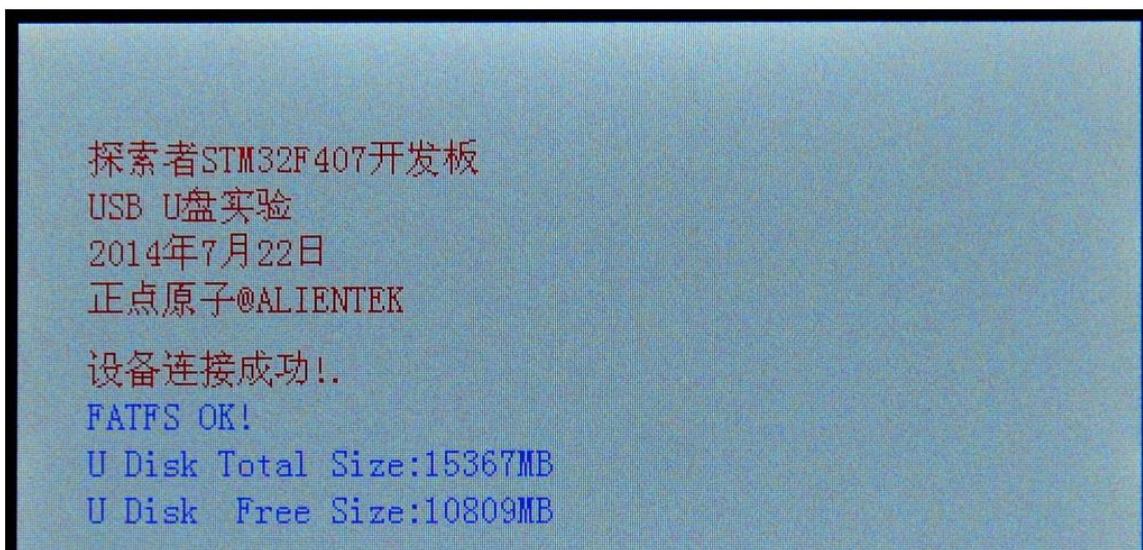


图 58.4.1 U 盘识别成功

此时，我们便可以通过 USMART 来测试 U 盘读写了，如图 58.4.2 和图 58.4.3 所示：

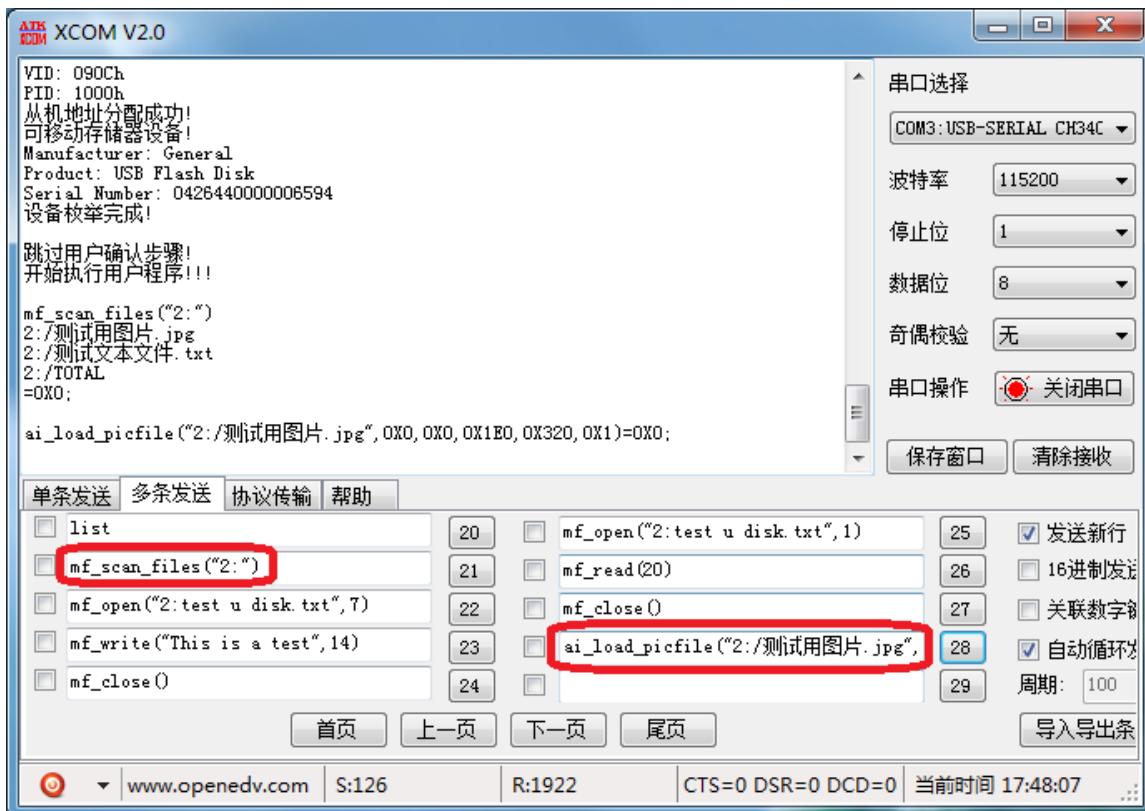


图 58.4.2 测试读取 U 盘读取

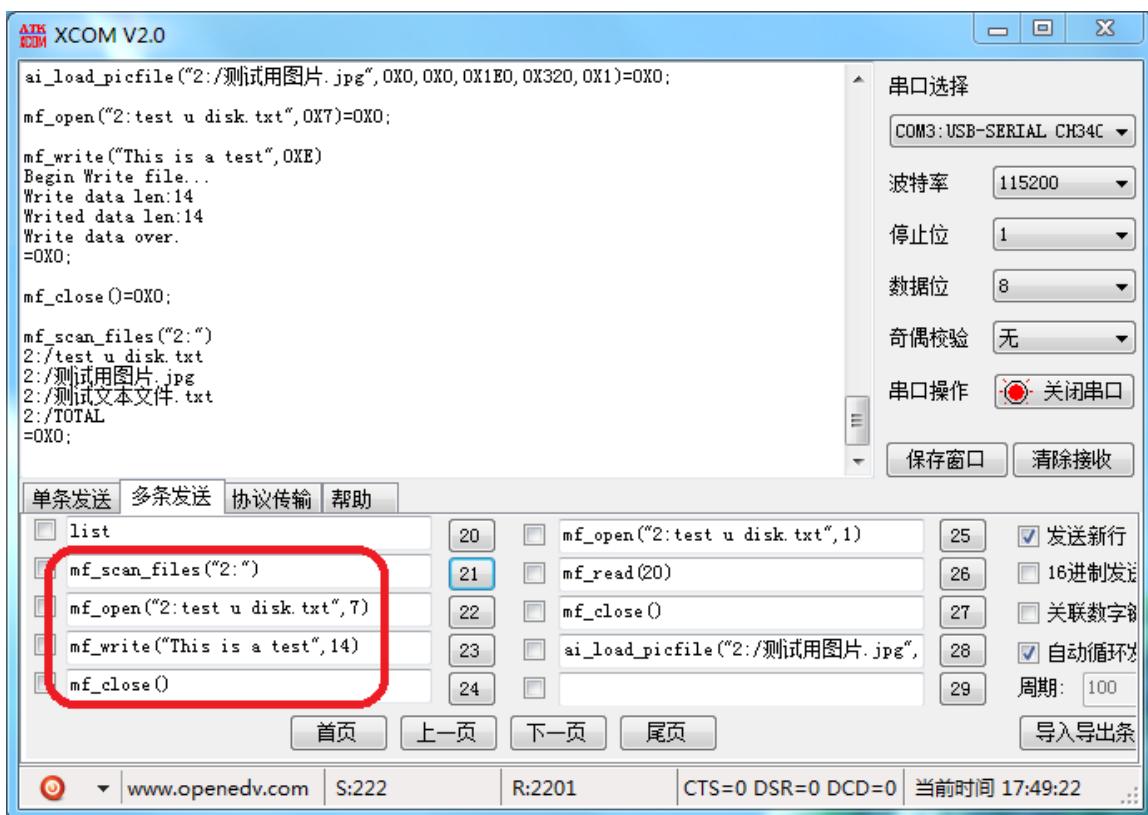


图 58.4.3 测试 U 盘写入

图 58.4.2 通过发送: `mf_scan_files("2:")`, 扫描 U 盘根目录所有文件, 然后通过 `ai_load_picfile("2:/测试用图片.jpg",0,0,480,800,1)`, 解码图片, 并显示在 LCD 上面。说明读 U 盘是没问题的。

图 58.4.3 通过发送: `mf_open("2:test u disk.txt",7)`, 在 U 盘根目录创建 `test u disk.txt` 这个文件, 然后发送: `mf_write("This is a test",14)`, 写入 `This is a test` 到这个文件里面, 然后发送: `mf_close()`, 关闭文件, 完成一次文件创建。最后, 发送: `mf_scan_files("2:")`, 扫描 U 盘根目录文件, 发现比图 58.4.2 所示多出了一个 `test u disk.txt` 的文件, 说明 U 盘写入成功。

这样, 就完成了本实验的设计目的: 实现 U 盘的读写操作。最后, 大家还可以调用其他函数, 实现相关功能测试, 这里就不给大家一一演示了, 测试方法同: FATFS 实验(第四十四章)。

第五十九章 USB 鼠标键盘(Host)实验

上一章我们向大家介绍了如何利用 STM32F4 的 USB HOST 接口来驱动 U 盘，本章，我们将利用 STM32F4 的 USB HOST 来驱动 USB 鼠标/键盘。本章分为如下几个部分：

- 59.1 USB 鼠标键盘简介
- 59.2 硬件设计
- 59.3 软件设计
- 59.4 下载验证

59.1 USB 鼠标键盘简介

传统的鼠标和键盘是采用 PS/2 接口和电脑通信的，但是现在 PS/2 接口在电脑上逐渐消失，所以现在越来越多的鼠标键盘采用的是 USB 接口，而不是 PS/2 接口的了。

USB 鼠标键盘属于 USB HID 设备。USB HID 即：Human Interface Device（人机交互设备）的缩写，键盘、鼠标与游戏杆等都属于此类设备。不过 HID 设备并不一定要有人机接口，只要符合 HID 类别规范的设备都是 HID 设备。关于 USB HID 的知识，我们这里就不详细介绍了，请大家自行百度学习。

本章，我们同上一章一样，我们直接移植官方的 USB HID 例程，官方例程路径：光盘→\8，STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.1.0→Project→USB_Host_Examples→HID，该例程支持 USB 鼠标和键盘等 USB HID 设备，本章我们将移植这个例程到探索者 STM32F407 开发板上。

59.2 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，然后初始化 USB HOST，并不断轮询。当检测到 USB 鼠标/键盘的插入后，显示设备类型，并显示设备输入数据，

如果是 USB 鼠标：将显示鼠标移动的坐标（X，Y 坐标），滚轮滚动数值（Z 坐标）以及按键（左中右）。

如果是 USB 键盘：将显示键盘输入的数字/字母等内容（不是所有按键都支持，部分按键没有做解码支持，比如 F1~F12）。

最后，还是用 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) 串口
- 3) TFTLCD 模块
- 4) USB HOST 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。这里再次提醒大家，P11 的连接，要通过跳线帽连接 PA11 和 D- 以及 PA12 和 D+。

59.3 软件设计

本章，我们在第十八章实验（实验 13 TFTLCD 显示实验）的基础上修改，先打开实验 13 的工程，在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹，对照官方 HID 例子，将相关文件拷贝到 USB 文件夹下。

然后，我们在工程里面添加 USB HID 相关代码，最终得到如图 59.3.1 所示的工程：

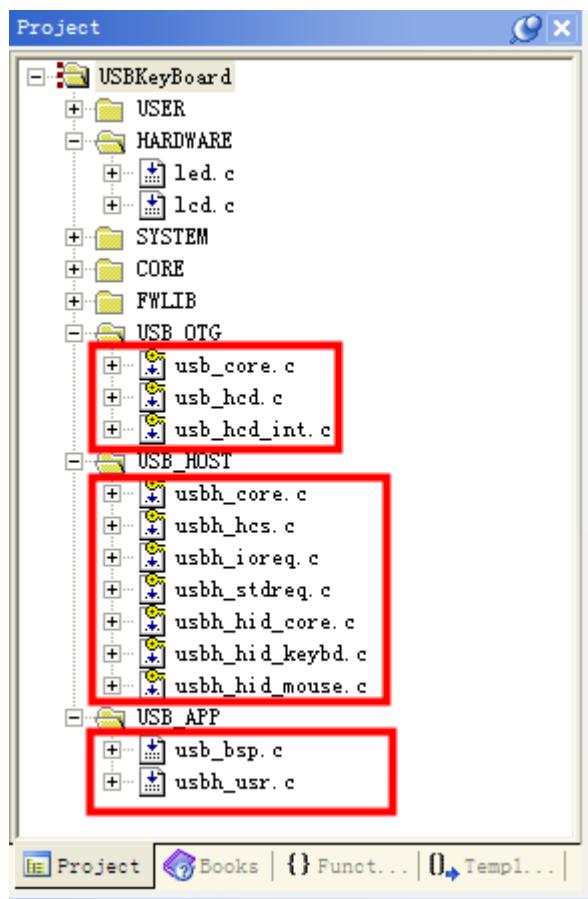


图 59.3.1 USB 鼠标键盘工程截图

可以看到，USB 部分代码，同上一章的在结构上是一模一样的，只是.c 文件稍微有些变化。同样，我们移植需要修改的代码，就是 USB_APP 里面的这两个.c 文件了。

其中 usb_bsp.c 的代码，和之前的章节一模一样，可以用上一章的代码直接替换即可正常使用。

usbh_usr.c 里面的代码，则有所变化，重点代码如下：

```
//下面两个函数,为 ALIENTEK 添加,以防止 USB 死机
//USB 枚举状态死机检测,防止 USB 枚举失败导致的死机
//phost:USB_HOST 结构体指针
//返回值:0,没有死机
//      1,死机了,外部必须重新启动 USB 连接.
u8 USBH_Check_EnumDead(USBH_HOST *phost)
{
    static u16 errcnt=0;
    //这个状态,如果持续存在,则说明 USB 死机了.
    if(phost->gState==HOST_CTRL_XFER&&(phost->EnumState==ENUM_IDLE||
        phost->EnumState==ENUM_GET_FULL_DEV_DESC))
    {
        errcnt++;
        if(errcnt>2000)//死机了
```

```
{  
    errcnt=0;  
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_OTG_FS,ENABLE);  
        //USB OTG FS 复位  
    delay_ms(5);  
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_OTG_FS,DISABLE);  
        //复位结束  
    return 1;  
}  
}  
}else errcnt=0;  
return 0;  
}  
//USB HID 通信死机检测,防止 USB 通信死机(暂时仅针对:DTERR,即 Data toggle error)  
//pcore:USB_OTG_Core_dev_HANDLE 结构体指针  
//phidm:HID_Machine_TypeDef 结构体指针  
//返回值:0,没有死机  
//      1,死机了,外部必须重新启动 USB 连接.  
u8 USBH_Check_HIDCommDead(USB_OTG_CORE_HANDLE *pcore,  
                            HID_Machine_TypeDef *phidm)  
{  
    if(pcore->host.HC_Status[phidm->hc_num_in]==HC_DATATGLERR)//DTERR 错误  
    {  
        return 1;  
    }  
    return 0;  
}  
//USB 键盘鼠标数据处理  
//鼠标初始化  
void USR_MOUSE_Init(void)  
{  
    USBH_Msg_Show(2);      //USB 鼠标  
    USB_FIRST_PLUGIN_FLAG=1;//标记第一次插入  
}  
//键盘初始化  
void USR_KEYBRD_Init(void)  
{  
    USBH_Msg_Show(1);      //USB 键盘  
    USB_FIRST_PLUGIN_FLAG=1;//标记第一次插入  
}  
//临时数组,用于存放鼠标坐标/键盘输入内容(4.3 屏,最大可以输入 2016 字节)  
__align(4) u8 tbuf[2017];  
//USB 鼠标数据处理  
//data:USB 鼠标数据结构体指针
```

```
void USR_MOUSE_ProcessData(HID_MOUSE_Data_TypeDef *data)
{
    static signed short x,y,z;
    if(USB_FIRST_PLUGIN_FLAG)//第一次插入,将数据清零
    {
        USB_FIRST_PLUGIN_FLAG=0;
        x=y=z=0;
    }
    x+=(signed char)data->x;
    if(x>9999)x=9999;
    if(x<-9999)x=-9999;
    y+=(signed char)data->y;
    if(y>9999)y=9999;
    if(y<-9999)y=-9999;
    z+=(signed char)data->z;
    if(z>9999)z=9999;
    if(z<-9999)z=-9999;
    POINT_COLOR=BLUE;
    sprintf((char*)tbuf,"BUTTON:");
    if(data->button&0X01)	strcat((char*)tbuf,"LEFT");
    if((data->button&0X02)==0X02)	strcat((char*)tbuf,"RIGHT");
    else if((data->button&0X03)==0X03)	strcat((char*)tbuf,"+RIGHT");
    if((data->button&0X07)==0X04)	strcat((char*)tbuf,"MID");
    else if((data->button&0X07)>0X04)	strcat((char*)tbuf,"+MID");
    LCD_Fill(30+56,180,lcddev.width,180+16,WHITE);
    LCD_ShowString(30,180,210,16,16,tbuf);
    sprintf((char*)tbuf,"X POS:%05d",x);
    LCD_ShowString(30,200,200,16,16,tbuf);
    sprintf((char*)tbuf,"Y POS:%05d",y);
    LCD_ShowString(30,220,200,16,16,tbuf);
    sprintf((char*)tbuf,"Z POS:%05d",z);
    LCD_ShowString(30,240,200,16,16,tbuf);
}

//USB 键盘数据处理
//data:USB 键盘数据结构体指针
void USR_KEYBRD_ProcessData (uint8_t data)
{
    static u16 pos,endx,endy,maxinputchar;
    u8 buf[4];
    if(USB_FIRST_PLUGIN_FLAG)//第一次插入,将数据清零
    {
        USB_FIRST_PLUGIN_FLAG=0;
        endx=((lcddev.width-30)/8)*8+30; //得到 endx 值
    }
}
```

```

endy=((lcddev.height-220)/16)*16+220; //得到 endy 值
maxinputchar=((lcddev.width-30)/8);
maxinputchar*=(lcddev.height-220)/16;//当前 LCD 最大可以显示的字符数.
pos=0;
}
POINT_COLOR=BLUE;
sprintf((char*)buf,"%02X",data);
LCD_ShowString(30+56,180,200,16,16,buf);//显示键值
if(data>=' '&&data<='~')
{
    tbuf[pos++]=data;
    tbuf[pos]=0; //添加结束符.
    if(pos>maxinputchar)pos=maxinputchar;//最大输入这么多
}else if(data==0X0D) //退格键
{
    if(pos)pos--;
    tbuf[pos]=0; //添加结束符.
}
if(pos<=maxinputchar) //没有超过显示区
{
    LCD_Fill(30,220,endx,endy,WHITE);
    LCD_ShowString(30,220,endx-30,endy-220,16,tbuf);
}
}

```

ST 官方的 USB HID 例程，仅仅是能用，很多地方还要改善，比如识别率低，容易死机（枚举/通信都可能死机）等问题，这里：USBH_Check_EnumeDead 和 USBH_Check_HIDCommDead 这两个函数，就是我们针对官方 HID 例程现有 bug 做出的改进处理，通过这两个函数，可以检测枚举/通信是否正常，当出现异常时，直接重启 USB 内核，重新连接设备，这样可以防止死机造成的程序无响应情况。

另外，为了提高对鼠标键盘的识别率和兼容性，对 usbh_hid_core.c 里面的两处代码进行了修改：

1， USBH_HID_ClassRequest 函数，修改代码（351 行）为：

```

classReqStatus = USBH_Set_Idle (pdev, pphost, 100, 0); //这里 duration 官方设置的是 0,修改为
//100,提高兼容性

```

2， USBH_Set_Idle 函数，修改代码（542 行）为：

```

phost->Control.setup.b.wLength.w = 100; //官方的这里设置的是 0,导致部分鼠标无法识别,
//这里修改为 100 以后,识别率明显提高.

```

以上两处地方，官方默认值都是设置的 0，我们修改为 100 后，可以明显提高 USB 鼠标/键盘的识别率，兼容性好很多。

再回到 usbh_usr.c，USR_MOUSE_Init 和 USR_MOUSE_ProcessData 用于处理鼠标数据，这两个函数在 usbh_hid_mouse.c 里面被调用，USR_MOUSE_Init 在鼠标初始化的时候被调用，而 USR_MOUSE_ProcessData 函数，则在鼠标初始化成功，轮询数据的时候调用，处理鼠标数据，该函数将得到的鼠标数据显示在 LCD 上面。

同样，USR_KEYBRD_Init 和 USR_KEYBRD_ProcessData 用于处理键盘数据，这两个函数在 usbh_hid_keybd.c 里面被调用，USR_KEYBRD_Init 在键盘初始化的时候被调用，而 USR_KEYBRD_ProcessData 函数，则在键盘初始化成功，轮询数据的时候调用，处理键盘数据，该函数将键盘输入的字符显示在 LCD 上面。

其他代码，我们就不再介绍了，请大家参考开发板光盘本例程源码。

最后，来看看 main.c 里面的代码，如下：

```
USBH_HOST USB_Host;
USB_OTG_CORE_HANDLE USB_OTG_Core_dev;
extern HID_Machine_TypeDef HID_Machine;
//USB 信息显示
//msgx:0,USB 无连接 1,USB 键盘
//      2,USB 鼠标      3,不支持的 USB 设备
void USBH_Msg_Show(u8 msgx)
{
    POINT_COLOR=RED;
    switch(msgx)
    {
        case 0: //USB 无连接
            LCD_ShowString(30,130,200,16,16,"USB Connecting...");
            LCD_Fill(0,150,lcddev.width,lcddev.height,WHITE);
            break;
        case 1: //USB 键盘
            LCD_ShowString(30,130,200,16,16,"USB Connected      ");
            LCD_ShowString(30,150,200,16,16,"USB KeyBoard");
            LCD_ShowString(30,180,210,16,16,"KEYVAL:");
            LCD_ShowString(30,200,210,16,16,"INPUT STRING:");
            break;
        case 2: //USB 鼠标
            LCD_ShowString(30,130,200,16,16,"USB Connected      ");
            LCD_ShowString(30,150,200,16,16,"USB Mouse");
            LCD_ShowString(30,180,210,16,16,"BUTTON:");
            LCD_ShowString(30,200,210,16,16,"X POS:");
            LCD_ShowString(30,220,210,16,16,"Y POS:");
            LCD_ShowString(30,240,210,16,16,"Z POS:");
            break;
        case 3: //不支持的 USB 设备
            LCD_ShowString(30,130,200,16,16,"USB Connected      ");
            LCD_ShowString(30,150,200,16,16,"Unknow Device");
            break;
    }
}
//HID 重新连接
void USBH_HID_Reconnect(void)
```

```
{  
    //关闭之前的连接  
    USBH_DeInit(&USB_OTG_Core_dev,&USB_Host); //复位 USB HOST  
    USB_OTG_StopHost(&USB_OTG_Core_dev);      //停止 USBhost  
    if(USB_Host.usr_cb->DeviceDisconnected)    //存在,才禁止  
    {  
        USB_Host.usr_cb->DeviceDisconnected(); //关闭 USB 连接  
        USBH_DeInit(&USB_OTG_Core_dev, &USB_Host);  
        USB_Host.usr_cb->DeInit();  
        USB_Host.class_cb->DeInit(&USB_OTG_Core_dev,&USB_Host.device_prop);  
    }  
    USB_OTG_DisableGlobalInt(&USB_OTG_Core_dev); //关闭所有中断  
    //重新复位 USB  
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_OTG_FS,ENABLE); //复位  
    delay_ms(5);  
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_OTG_FS,DISABLE); //复位结束  
    memset(&USB_OTG_Core_dev,0,sizeof(USB_OTG_CORE_HANDLE));  
    memset(&USB_Host,0,sizeof(USB_Host));  
    //重新连接 USB HID 设备  
    USBH_Init(&USB_OTG_Core_dev,USB_OTG_FS_CORE_ID,&USB_Host,&HID_cb,  
              &USR_Callbacks);  
}  
  
int main(void)  
{  
    u32 t;  
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2  
    delay_init(168); //初始化延时函数  
    uart_init(115200); //初始化串口波特率为 115200  
    LED_Init(); //初始化 LED  
    LCD_Init(); //初始化 LCD  
    POINT_COLOR=RED;  
    LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");  
    LCD_ShowString(30,70,200,16,16,"USB MOUSE/KEYBOARD TEST");  
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");  
    LCD_ShowString(30,110,200,16,16,"2014/7/23");  
    LCD_ShowString(30,130,200,16,16,"USB Connecting...");  
    //初始化 USB 主机  
    USBH_Init(&USB_OTG_Core_dev,USB_OTG_FS_CORE_ID,&USB_Host,&HID_cb,  
              &USR_Callbacks);  
    while(1)  
    {  
        USBH_Process(&USB_OTG_Core_dev, &USB_Host);  
        if(bDeviceState==1)//连接建立了
```

```
{  
    if(USBH_Check_HIDCommDead(&USB_OTG_Core_dev,&HID_Machine))  
        //检测 USB HID 通信,是否还正常?  
    {  
        USBH_HID_Reconnect(); //重连  
    }  
  
}  
}  
}  
}  
}  
}  
}  
}
```

这里总共三个函数：USBH_Msg_Show 用于显示一些提示信息，在 usbh_usr.c 里面被相关函数调用。USBH_HID_Reconnect 则用于 USB HID 重新连接，当发现枚举/通信死机的时候，调用该函数实现 USB 复位重启，以重新连接；最后，main 函数就比较简单了，处理方式和上一章几乎一样，只是多了一些通信死机处理。

软件设计部分就给大家介绍到这里。

59.4 下载验证

在代码编译成功之后，我们下载到探索者 STM32F4 开发板上，然后在 USB_HOST 端子插入 USB 鼠标/键盘，注意：**此时 USB SLAVE 口不要插 USB 线到电脑，否则会干扰!!**

等 USB 鼠标/键盘成功识别后，便可以看到 LCD 显示 USB Connected，并显示设备类型：USB Mouse 或者 USB KeyBoard，同时也会显示输入的数据，如图 59.4.1 和图 59.4.2 所示：

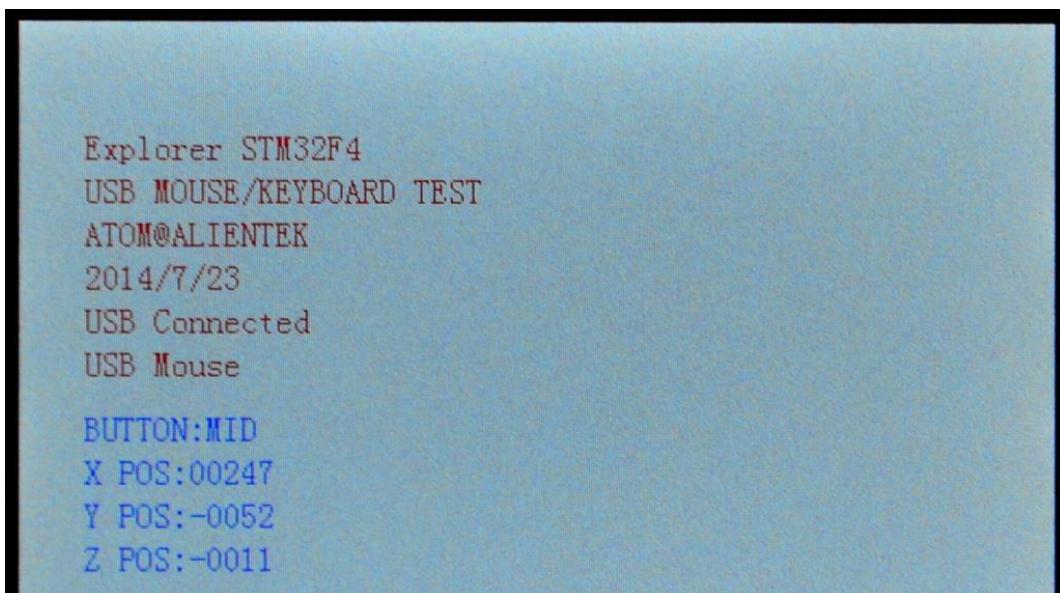


图 59.4.1 USB 鼠标测试

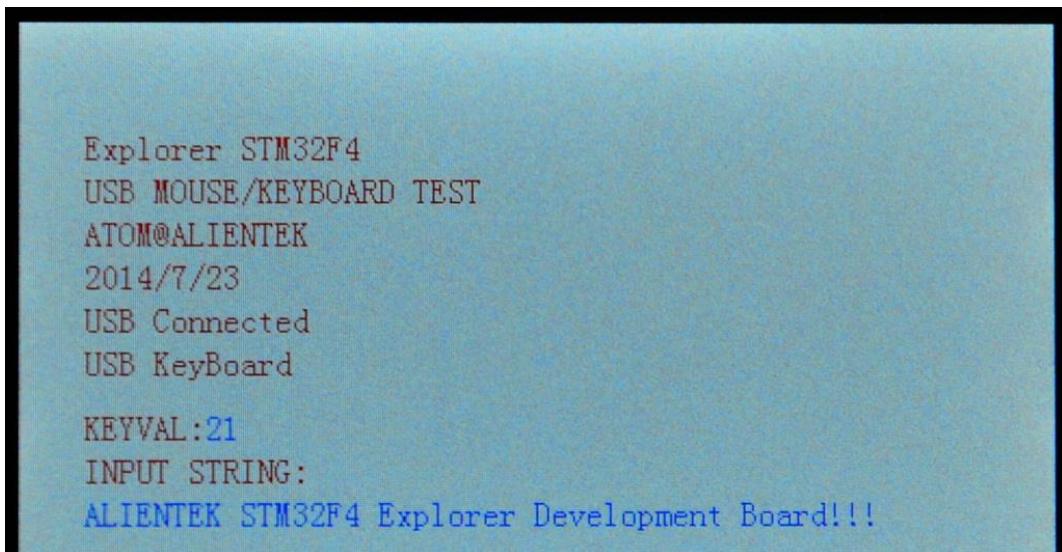


图 59.4.2 USB 键盘测试

其中，图 59.4.1 是 USB 鼠标测试界面，图 59.4.2 是 USB 键盘测试界面。

最后，特别提醒大家，由于例程的 HID 内核，只处理了第一个接口描述符，所以对于 USB 符合设备，只能识别第一个描述符所代表的设备。体现到实际使用中，就是：USB 无线鼠标，一般是无法使用（被识别为键盘），而 USB 无线键盘，可以使用，因为键盘在第一个描述符，鼠标在第二个描述符。

如果想支持 USB 无线鼠标，可以通过修改 usbh_hid_core.c 里面的 USBH_HID_InterfaceInit 函数来支持。

第六十章 网络通信实验

本章，我们将向大家介绍探索者 STM32F4 开发板的网口及其使用。本章，我们将使用 ALIENTEK 探索者 STM32F4 开发板自带的网口和 LWIP 实现：TCP 服务器、TCP 客户端、UDP 以及 WEB 服务器等四个功能。本章分为如下几个部分：

- 60.1 STM32F4 以太网以及 TCP/IP LWIP 简介
- 60.2 硬件设计
- 60.3 软件设计
- 60.4 下载验证

60.1 STM32F4 以太网以及 TCP/IP LWIP 简介

本章，我们需要用到 STM32F4 的以太网控制器和 LWIP TCP/IP 协议栈。接下来分别介绍这两个部分。

60.1.1 STM32F4 以太网简介

STM32F407 芯片自带以太网模块，该模块包括带专用 DMA 控制器的 MAC 802.3（介质访问控制）控制器，支持介质独立接口（MII）和简化介质独立接口（RMII），并自带了一个用于外部 PHY 通信的 SMI 接口，通过一组配置寄存器，用户可以为 MAC 控制器和 DMA 控制器选择所需模式和功能。

STM32F4 自带以太网模块特点包括：

- 支持外部 PHY 接口，实现 10M/100Mbit/s 的数据传输速率
- 通过符合 IEEE802.3 的 MII/RMII 接口与外部以太网 PHY 进行通信
- 支持全双工和半双工操作
- 可编程帧长度，支持高达 16KB 巨型帧
- 可编程帧间隔（40~96 位时间，以 8 为步长）
- 支持多种灵活的地址过滤模式
- 通过 SMI（MDIO）接口配置和管理 PHY 设备
- 支持以太网时间戳（参见 IEEE1588-2008），提供 64 位时间戳
- 提供接收和发送两组 FIFO。
- 支持 DMA

STM32F4 以太网功能框图如图 60.1.1.1 所示：

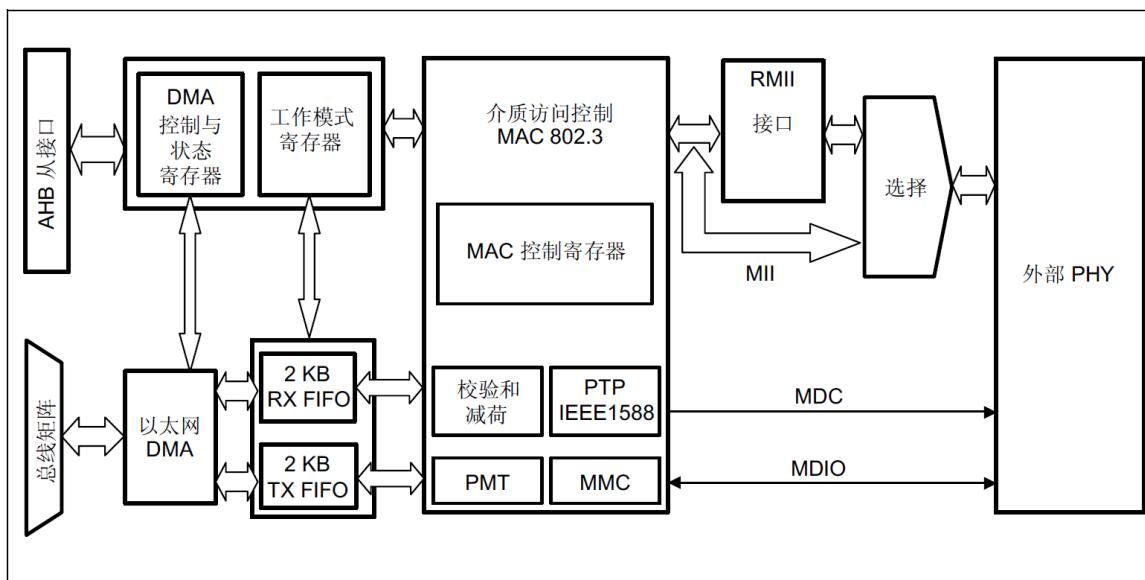


图 60.1.1.1 STM32F4 以太网框图

从上图可以看出，STM32F4 是必须外接 PHY 芯片，才可以完成以太网通信的，外部 PHY 芯片可以通过 MII/RMII 接口与 STM32F4 内部 MAC 连接，并且支持 SMI (MDIO&MDC) 接口配置外部以太网 PHY 芯片。

接下来分别介绍 SMI/MII/RMII 接口和外部 PHY 芯片。

SMI 接口，即站管理接口，该接口允许应用程序通过 2 条线：时钟(MDC)和数据线(MDIO)访问任意 PHY 寄存器。该接口支持访问多达 32 个 PHY，应用程序可以从 32 个 PHY 中选择一个 PHY，然后从任意 PHY 包含的 32 个寄存器中选择一个寄存器，发送控制数据或接收状态信息。任意给定时间内只能对一个 PHY 中的一个寄存器进行寻址。

MII 接口，即介质独立接口，用于 MAC 层与 PHY 层进行数据传输。STM32F407 通过 MII 与 PHY 层芯片的连接如图 60.1.1.2 所示。

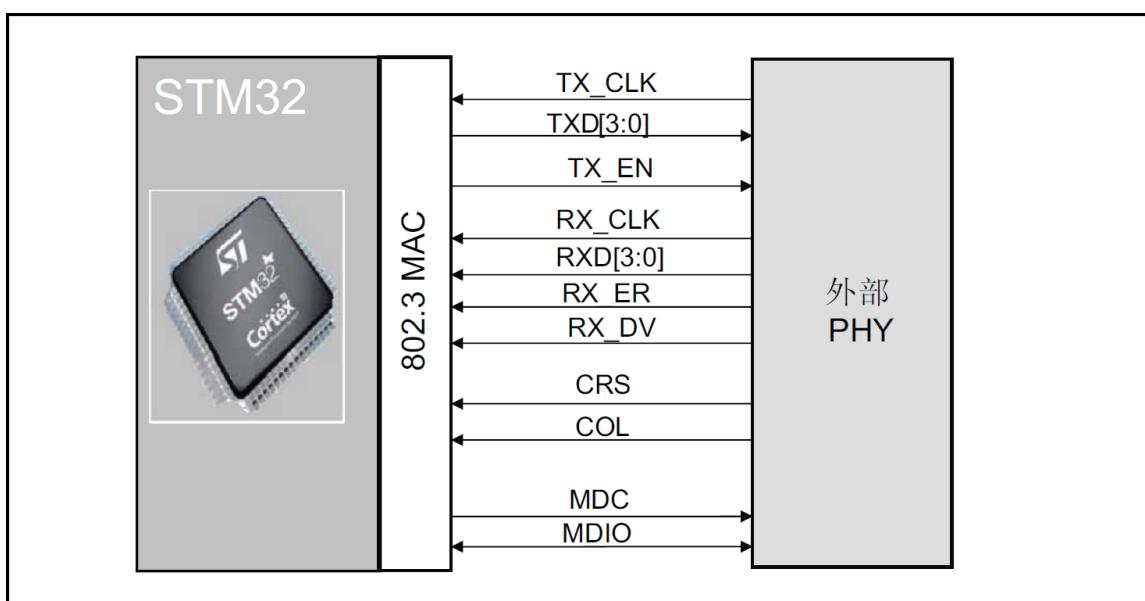


图 60.1.1.2 MII 接口信号

- MII_TX_CLK: 连续时钟信号。该信号提供进行 TX 数据传输时的参考时序。标称频率为：速率为 10 Mbit/s 时为 2.5 MHz；速率为 100 Mbit/s 时为 25 MHz。

- MII_RX_CLK: 连续时钟信号。该信号提供进行 RX 数据传输时的参考时序。标称频率为：速率 10 Mbit/s 时为 2.5 MHz；速率 100 Mbit/s 时为 25 MHz。
- MII_TX_EN: 发送使能信号。
- MII_TXD[3:0]: 数据发送信号。该信号是 4 个一组的数据信号，
- MII_CRS: 载波侦听信号。
- MII_COL: 冲突检测信号。
- MII_RXD[3:0]: 数据接收信号。该信号是 4 个一组的数据信号
- MII_RX_DV: 接收数据有效信号。
- MII_RX_ER: 接收错误信号。该信号必须保持一个或多个周期(MII_RX_CLK)，从而向 MAC 子层指示在帧的某处检测到错误。

RMII 接口，即精简介质独立接口，该接口降低了在 10/100 Mbit/s 下微控制器以太网外设与外部 PHY 间的引脚数。根据 IEEE 802.3u 标准，MII 包括 16 个数据和控制信号的引脚。RMII 规范将引脚数减少为 7 个。

RMII 接口是 MAC 和 PHY 之间的实例化对象。这有助于将 MAC 的 MII 转换为 RMII。RMII 具有以下特性：

- 1, 支持 10Mbit/s 和 100Mbit/s 的运行速率
- 2, 参考时钟必须是 50 MHz
- 3, 相同的参考时钟必须从外部提供给 MAC 和外部以太网 PHY
- 4, 它提供了独立的 2 位宽（双位）的发送和接收数据路径

STM32F407 通过 RMII 接口与 PHY 层芯片的连接如图 60.1.1.3 所示：

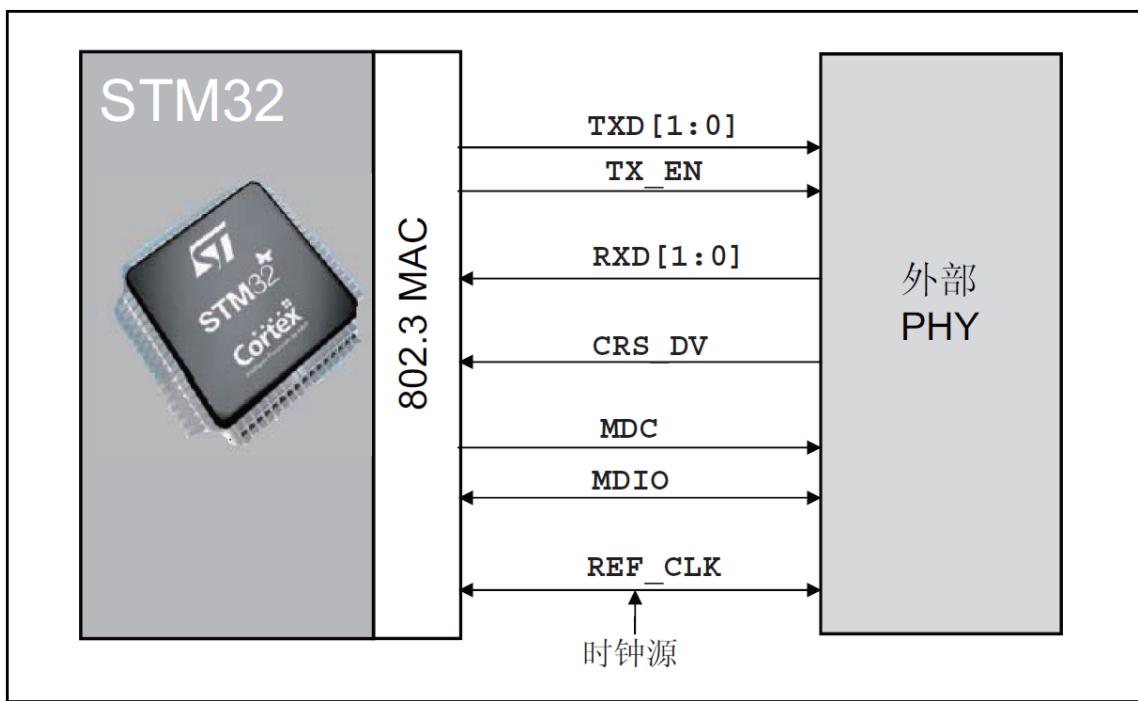


图 60.1.1.3 RMII 接口信号

从上图可以看出 RMII 相比 MII，引脚数量精简了不少。注意，图中的 REF_CLK 信号，是 RMII 和外部 PHY 共用的 50Mhz 参考时钟，必须由外部提供，比如有源晶振，或者 STM32F4 的 MCO 输出。不过有些 PHY 芯片可以自己产生 50Mhz 参考时钟，同时提供给 STM32F4，这样也是可以的。

本章我们采用 RMII 接口和外部 PHY 芯片连接，实现网络通信功能，探索者 STM32F4 开发板使用的是 LAN8720A 作为 PHY 芯片。接下来，我们简单介绍一下 LAN8720A 这个芯片。

LAN8720A 是低功耗的 10/100M 以太网 PHY 层芯片，I/O 引脚电压符合 IEEE802.3-2005 标准，支持通过 RMII 接口与以太网 MAC 层通信，内置 10-BASE-T/100BASE-TX 全双工传输模块，支持 10Mbps 和 100Mbps。

LAN8720A 可以通过自协商的方式与目的主机最佳的连接方式(速度和双工模式)，支持 HP Auto-MDIX 自动翻转功能，无需更换网线即可将连接更改为直连或交叉连接。LAN8720A 的主要特点如下：

- 高性能的 10/100M 以太网传输模块
- 支持 RMII 接口以减少引脚数
- 支持全双工和半双工模式
- 两个状态 LED 输出
- 可以使用 25M 晶振以降低成本
- 支持自协商模式
- 支持 HP Auto-MDIX 自动翻转功能
- 支持 SMI 串行管理接口
- 支持 MAC 接口

LAN8720A 功能框图如图 60.1.1.4 所示。

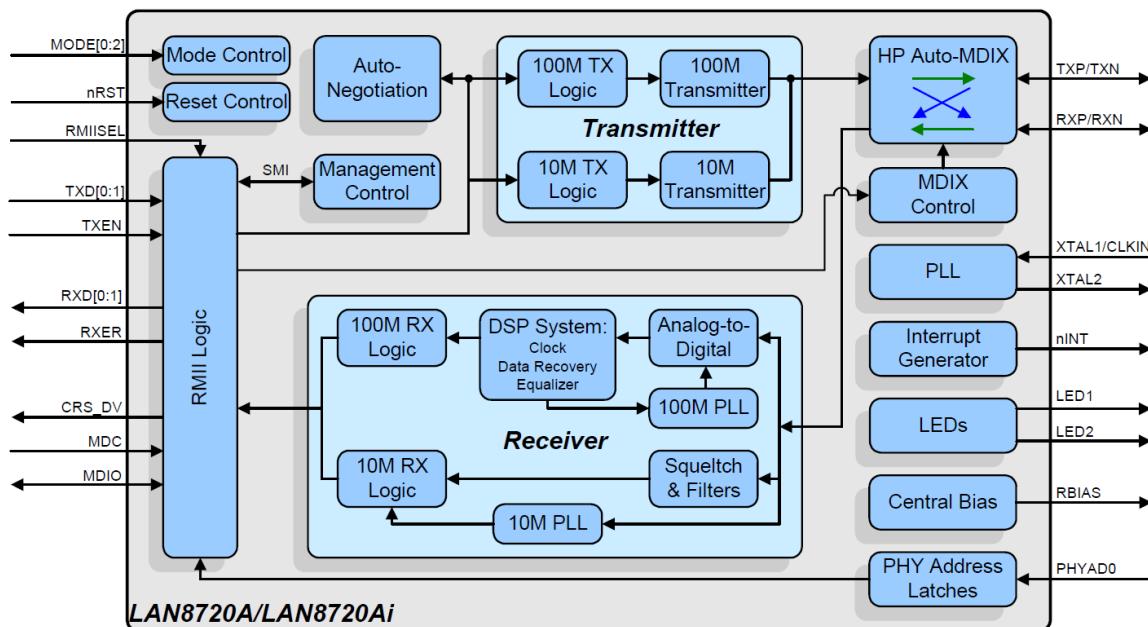


图 60.1.1.4 LAN8720A 功能框图

LAN8720A 的引脚数是比较少的，因此，很多引脚具有多个功能。这里，我们介绍几个重要的设置。

1, PHY 芯片地址设置

LAN8720A 可以通过 PHYAD0 引脚来配置，该引脚与 RXER 引脚复用，芯片内部自带下拉电阻，当硬复位结束后，LAN8720A 会读取该引脚电平，作为器件的 SMI 地址，接下拉电阻时（浮空也可以，因为芯片内部自带了下拉电阻），设置 SMI 地址为 0，当外接上拉电阻后，可以设置为 1。本章我们采用的是该引脚浮空，即设置 LAN8720 地址为 0。

2, nINT/REFCLKO 引脚功能配置

nINT/REFCLKO 引脚可以用作中断输出，或者参考时钟输出。通过 LED2 (nINTSEL) 引脚设置，LED2 引脚的值在芯片复位后，被 LAN8720A 读取，当该引脚接上拉电阻（或浮空，内置上拉电阻），那么正常工作后，nINT/REFCLKO 引脚将作为中断输出引脚（选中 REF_CLK IN 模式）。当该引脚接下拉电阻时，正常工作后，nINT/REFCLKO 引脚将作为参考时钟输出（选中 REF_CLK OUT 模式）。

在 REF_CLK IN 模式，外部必须提供 50Mhz 参考时钟给 LAN8720A 的 XTAL1 (CLKIN) 引脚。

在 REF_CLK OUT 模式，LAN8720A 可以外接 25Mhz 石英晶振，通过内部倍频到 50Mhz，然后通过 REFCLKO 引脚，输出 50Mhz 参考时钟给 MAC 控制器。这种方式，可以降低 BOM 成本。

本章，我们设置 nINT/REFCLKO 引脚为参考时钟输出（REF_CLK OUT 模式），用于给 STM32F4 的 RMII 提供 50Mhz 参考时钟。

3. 1.2V 内部稳压器配置

LAN8720A 需要 1.2V 电压给 VDDCR 供电，不过芯片内部集成了 1.2V 稳压器，可以通过 LED1(REGOFF)来配置是否使用内部稳压器，当不使用内部稳压器的时候，必须外部提供 1.2V 电压给 VDDCR 引脚。这里我们使用内部稳压器，所以我们在 LED1 接下拉电阻（浮空也行，内置了下拉电阻），以控制开启内部 1.2V 稳压器。

最后，我们来看下 LAN8720A 同我们探索者 STM32F4 开发板的连接关系，如图 60.1.1.5 所示：

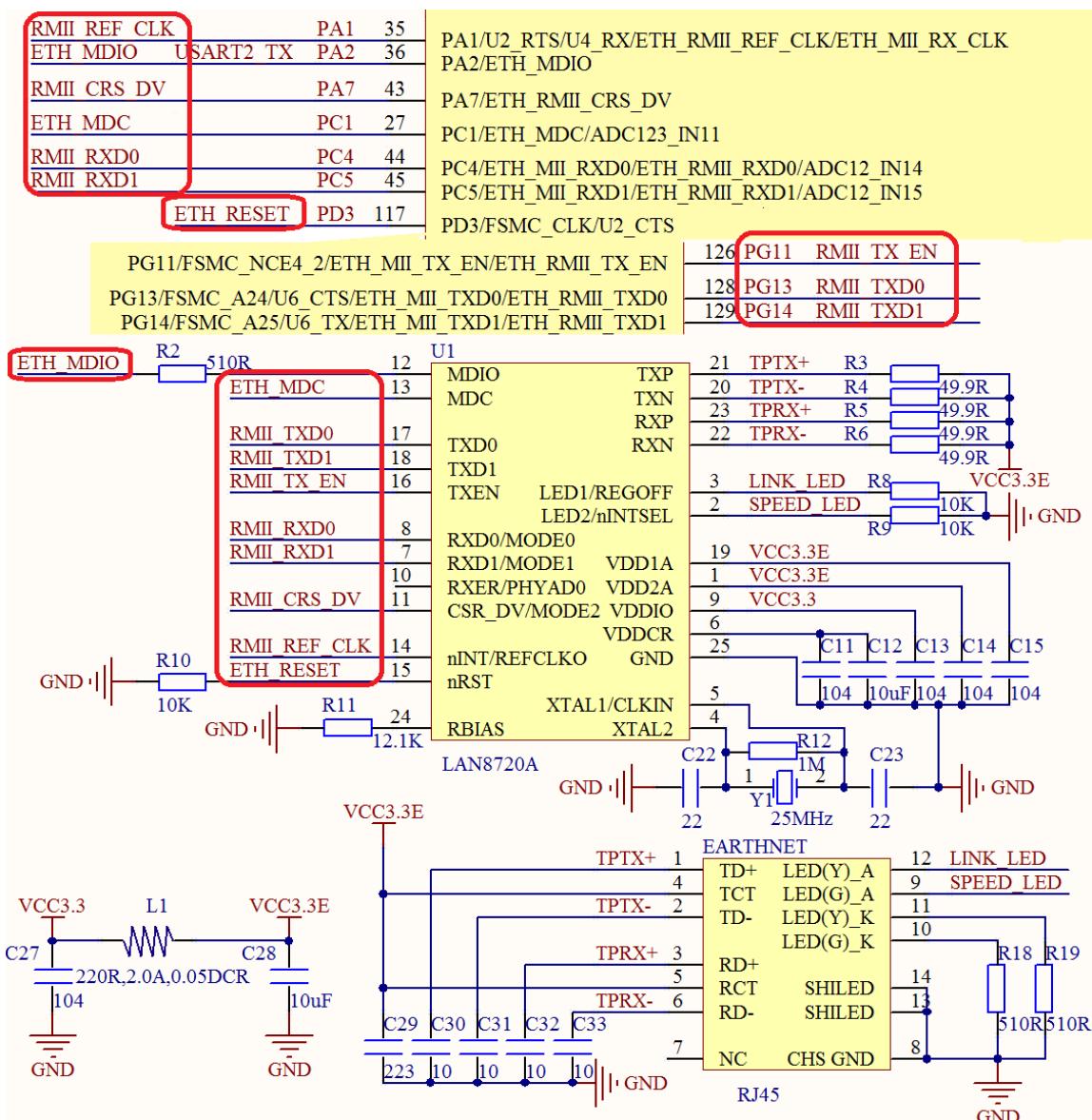


图 60.1.1.5 LAN8720A 与 STM32F407ZGT6 连接原理图

从上图可以看出，LAN8720A 总共通过 10 跟线同 STM32F4 连接，注意：MDIO 同串口 2 的 TX 信号有共用，所以串口 2 和以太网功能不能同时使用，使用时需要注意这个问题。

60.1.2 TCP/IP LWIP 简介

1. TCP/IP 简介

TCP/IP 中文名为传输控制协议/因特网互联协议，又名网络通讯协议，是 Internet 最基本的协议、Internet 国际互联网络的基础，由网络层的 IP 协议和传输层的 TCP 协议组成。TCP/IP 定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了 4 层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。通俗而言：TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而 IP 是给因特网的每一台联网设备规定一个地址。

TCP/IP 协议不是 TCP 和 IP 这两个协议的合称，而是指因特网整个 TCP/IP 协议族。从协议分层模型方面来讲，TCP/IP 由四个层次组成：网络接口层、网络层、传输层、应用层。OSI 是

传统的开放式系统互连参考模型,该模型将 TCP/IP 分为七层: 物理层、数据链路层(网络接口层)、网络层(网络层)、传输层(传输层)、会话层、表示层和应用层(应用层)。TCP/IP 模型与 OSI 模型对比如表 60.1.2.1 所示。

编 号	OSI 模型	TCP/IP 模型
1	应用层	应用层
2	表示层	
3	会话层	
4	传输层	传输层
5	网络层	互联层
6	数据链路层	链路层
7	物理层	

表 60.1.2.1 TCP/IP 模型与 OSI 模型对比

具体一点理解,本例程中的: PHY 层芯片 LAN8720A 相当于物理层, STM32F407 自带的 MAC 层相当于数据链路层,而 LWIP 提供的就是网络层、传输层的功能,应用层是需要用户自己根据自己想要的功能去实现的。

2, LWIP 简介

LWIP 是瑞典计算机科学院(SICS)的 Adam Dunkels 等开发的一个小型开源的 TCP/IP 协议栈,是 TCP/IP 的一种实现方式。LWIP 是轻量级 IP 协议,有无操作系统的支持都可以运行, LWIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用,它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行,这使 LWIP 协议栈适合在低端的嵌入式系统中使用。目前 LWIP 的最新版本是 1.4.1。本教程采用的就是 1.4.1 版本的 LWIP。

关于 LWIP 的详细信息大家可以去 <http://savannah.nongnu.org/projects/lwip/> 这个网站去查阅,LWIP 的主要特性如下:

- ARP 协议, 以太网地址解析协议;
- IP 协议, 包括 IPv4 和 IPv6, 支持 IP 分片与重装, 支持多网络接口下数据转发;
- ICMP 协议, 用于网络调试与维护;
- IGMP 协议, 用于网络组管理, 可以实现多播数据的接收;
- UDP 协议, 用户数据报协议;
- TCP 协议, 支持 TCP 拥塞控制, RTT 估计, 快速恢复与重传等;
- 提供三种用户编程接口方式: raw/callback API、sequential API、BSD-style socket API;
- DNS, 域名解析;
- SNMP, 简单网络管理协议;
- DHCP, 动态主机配置协议;
- AUTOIP, IP 地址自动配置;
- PPP, 点对点协议, 支持 PPPoE

我们从 LWIP 官网上下载 LWIP 1.4.1 版本, 打开后如图 60.1.2.1 所示。

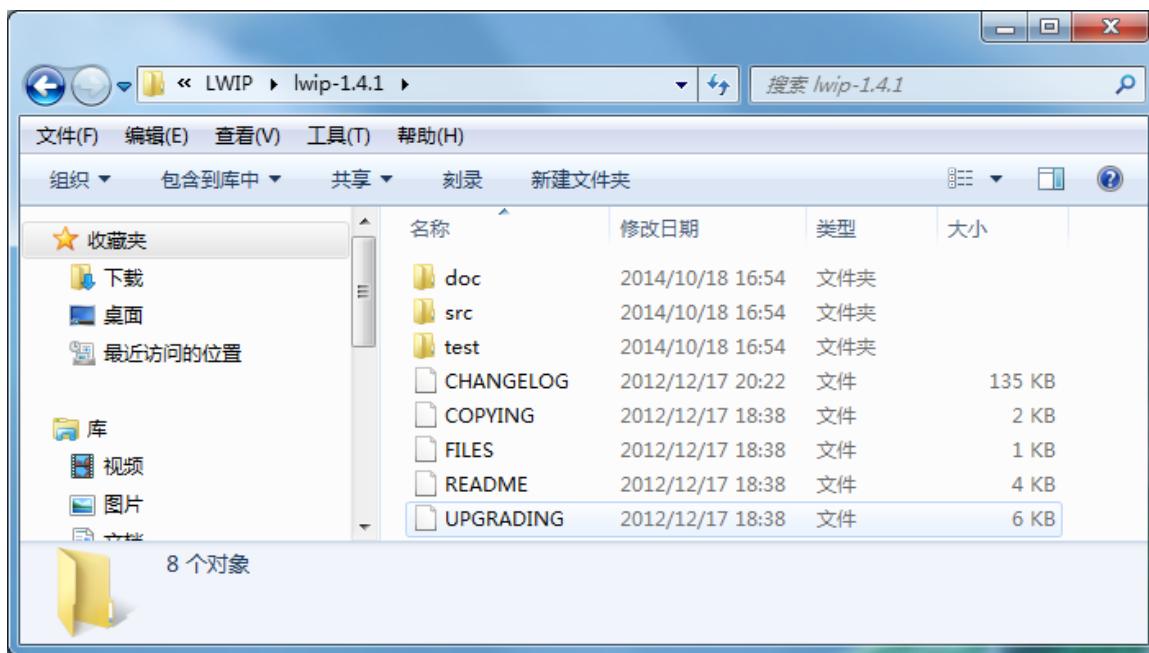


图 60.1.2.1 LWIP 1.4.1 源码内容

打开从官网上下载下来的 LWIP1.4.1 其中包括 doc, src 和 test 三个文件夹和 5 个其他文件。doc 文件夹下包含了几个与协议栈使用相关的文本文档, doc 文件夹里面有两个比较重要的文档:rawapi.txt 和 sys_arch.txt。

rawapi.txt 告诉读者怎么使用 raw/callback API 进行编程, sys_arch.txt 包含了移植说明, 在移植的时候会用到。src 文件夹是我们的重点, 里面包含了 LWIP 的源码。test 是 LWIP 提供的一些测试程序, 方便大家使用 LWIP。打开 src 源码文件夹, 如图 60.1.2.2 所示:

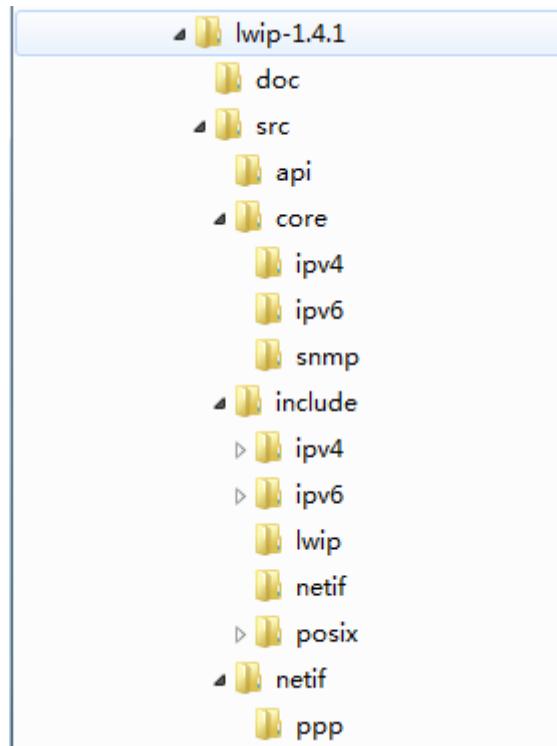


图 60.1.2.2 LWIP src 文件夹内容

src 文件夹由 4 个文件夹组成: api、core、include、netif 四个文件夹。api 文件夹里面是 LWIP

的 sequential API(Netconn)和 socket API 两种接口函数的源码，要使用这两种 API 需要操作系统支持。core 文件夹是 LWIP 内核源码，实现了各种协议支持，include 文件夹里面是 LWIP 使用到的头文件，netif 文件夹里面是与网络底层接口有关的文件。

关于 LWIP 的移植，请参考：ALIENTEK STM32F4 LWIP 使用教程.pdf（文档路径：光盘→6，软件资料→LWIP 学习资料）第一章，该文档详细介绍了 LWIP 在 STM32F4 上面的移植。这里我们就不详细介绍了。

60.2 硬件设计

本节实验功能简介：开机后，程序初始化 LWIP，包括：初始化 LAN8720A、申请内存、开启 DHCP 服务、添加并打开网卡，然后等待 DHCP 获取 IP 成功，当 DHCP 获取成功后，将在 LCD 屏幕上显示 DHCP 得到的 IP 地址，如果 DHCP 获取失败，那么将使用静态 IP（固定为：192.168.1.30），然后开启 Web Server 服务，并进入主循环，等待按键输入选择需要测试的功能：

KEY0 按键，用于选择 TCP Server 测试功能。

KEY1 按键，用于选择 TCP Client 测试功能

KEY2 按键，用于选择 UDP 测试功能

TCP Server 测试的时候，直接使用 DHCP 获取到的 IP（DHCP 失败，则使用静态 IP）作为服务器地址，端口号固定为：8088。在电脑端，可以使用网络调试助手（TCP Client 模式）连接开发板，连接成功后，屏幕显示连接上的 Client 的 IP 地址，此时便可以互相发送数据了。按 KEY0 发送数据给电脑，电脑端发送过来的数据将会显示在 LCD 屏幕上。按 KEY_UP 可以退出 TCP Server 测试。

TCP Client 测试的时候，先通过 KEY0/KEY2 来设置远端 IP 地址（Server 的 IP），端口号固定为：8087。设置好之后，通过 KEY_UP 确认，随后，开发板会不断尝试连接到所设置的远端 IP 地址（端口：8087），此时我们需要在电脑端使用网络调试助手(TCP Server 模式)，设置端口为：8087，开启 TCP Server 服务，等待开发板连接。当连接成功后，测试方法同 TCP Server 测试的方法一样。

UDP 测试的时候，同 TCP Client 测试几乎一模一样，先通过 KEY0/KEY2 设置远端 IP 地址（电脑端的 IP），端口号固定为：8089，然后按 KEY_UP 确认。电脑端使用网络调试助手（UDP 模式），设置端口为：8089，开启 UDP 服务。不过对于 UDP 通信，我们得先按开发板 KEY0，发送一次数据给电脑，随后才可以电脑发送数据给开发板，实现数据互发。按 KEY_UP 可以退出 UDP 测试。

Web Server 的测试相对简单，只需要在浏览器端输入开发板的 IP 地址（DHCP 获取到的 IP 地址或者 DHCP 失败时使用的静态 IP 地址），即可登录一个 Web 界面，在 Web 界面，可以实现对 DS1(LED1)的控制、蜂鸣器的控制、查看 ADC1 通道 5 的值、内部温度传感器温度值以及查看 RTC 时间和日期等。

DS0 用于提示程序正在运行。

本例程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/KEY_UP)
- 3) 串口
- 4) TFTLCD 模块
- 5) ETH (STM32F4 自带以太网功能)
- 6) LAN8720A

这几个部分我们都已经详细介绍过了。本实验测试，需自备网线一根，路由器一个。

60.3 软件设计

本章，我们综合了《STM32F4 LWIP 开发手册.pdf》这个文档里面的 4 个 LWIP 基础例程：UDP 实验、TCP 客户端（TCP Client）实验、TCP 服务器（TCP Server）实验和 Web Server 实验。这些实验测试代码在工程 LWIP→lwip_app 文件夹下，如图 60.3.1 所示：

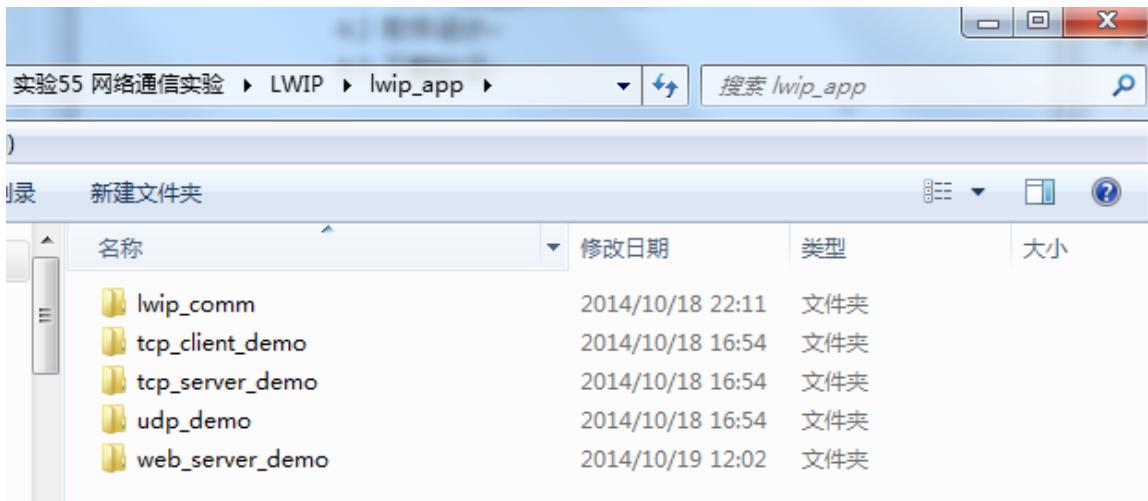


图 60.3.1 LWIP 文件夹内容

这里面总共 4 个文件夹：lwip_comm 文件夹，存放了 ALIENTEK 提供的 LWIP 扩展支持代码，方便使用和配置 LWIP，其他四个文件夹，则分别存放了 TCP Client、TCP Server、UDP 和 Web Server 测试 demo 程序。这里我们就不详细介绍这些内容了，详细的介绍，请参考：《STM32F4 LWIP 开发手册.pdf》这个文档。本例程工程结构如图 60.3.2 所示：

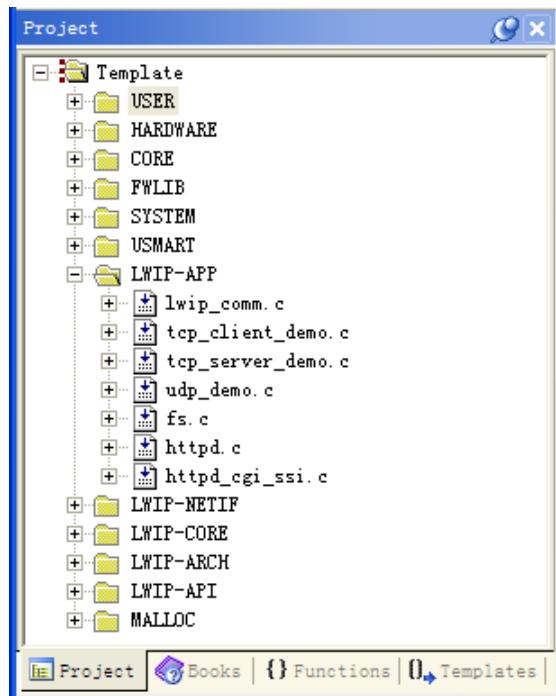


图 60.3.2 例程工程结构体

本章例程所实现的功能，全部由 LWIP_APP 组下的几个.c 文件实现，这些文件的具体介绍

在：ALIENTEK STM32F4 LWIP 使用教程.pdf 里面，请大家参考该文档学习。

其他部分代码我们就不详细介绍了，最后，我们来看看 main.c 里面的代码，如下：

```
//加载 UI
//mode:
//bit0:0,不加载;1,加载前半部分 UI
//bit1:0,不加载;1,加载后半部分 UI
void lwip_test_ui(u8 mode)
{
    u8 speed; u8 buf[30];
    POINT_COLOR=RED;
    if(mode&1<<0)
    {
        LCD_Fill(30,30	lcddev.width,110,WHITE); //清除显示
        LCD_ShowString(30,30,200,16,16,"Explorer STM32F4");
        LCD_ShowString(30,50,200,16,16,"Ethernet lwIP Test");
        LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(30,90,200,16,16,"2014/8/15");
    }
    if(mode&1<<1)
    {
        LCD_Fill(30,110	lcddev.width, lcddev.height,WHITE);//清除显示
        LCD_ShowString(30,110,200,16,16,"lwIP Init Successed");
        if(lwipdev.dhcpstatus==2)sprintf((char*)buf,"DHCP IP:%d.%d.%d.%d",lwipdev.ip[0],
                                         lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);//IP 地址
        else sprintf((char*)buf,"Static IP:%d.%d.%d.%d",lwipdev.ip[0],lwipdev.ip[1],
                     lwipdev.ip[2],lwipdev.ip[3]);//打印静态 IP 地址
        LCD_ShowString(30,130,210,16,buf);
        speed=LAN8720_Get_Speed();//得到网速
        if(speed&1<<1)LCD_ShowString(30,150,200,16,16,"Ethernet Speed:100M");
        else LCD_ShowString(30,150,200,16,16,"Ethernet Speed:10M");
        LCD_ShowString(30,170,200,16,16,"KEY0:TCP Server Test");
        LCD_ShowString(30,190,200,16,16,"KEY1:TCP Client Test");
        LCD_ShowString(30,210,200,16,16,"KEY2:UDP Test");
    }
}
int main(void)
{
    u8 t; u8 key;
    delay_init();           //延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    uart_init(115200);      //串口波特率设置
    usmart_dev.init(84);    //初始化 USMART
    LED_Init();             //LED 初始化
```

```
KEY_Init();           //按键初始化
LCD_Init();          //LCD 初始化
BEEP_Init();         //蜂鸣器初始化
RTC_Init();          //RTC 初始化
Adc_Init();          //ADC 初始化
TIM3_Int_Init(100-1,8400-1); //10khz 的频率,计数 100 为 10ms
my_mem_init(SRAMIN); //初始化内部内存池
my_mem_init(SRAMCCM); //初始化 CCM 内存池
POINT_COLOR=RED;     //红色字体
lwip_test_ui(1);     //加载前半部分 UI
//先初始化 lwIP(包括 LAN8720A 初始化),此时必须插上网线,否则初始化会失败!!
LCD_ShowString(30,110,200,16,16,"lwIP Init...");
while(lwip_comm_init()!=0)
{
    LCD_ShowString(30,110,200,16,16,"lwIP Init failed!");
    delay_ms(1200);
    LCD_Fill(30,110,230,110+16,WHITE); //清除显示
    LCD_ShowString(30,110,200,16,16,"Retrying...");
}
LCD_ShowString(30,110,200,16,16,"lwIP Init Successed");
//等待 DHCP 获取
LCD_ShowString(30,130,200,16,16,"DHCP IP configing...");
while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF)) //等待 DHCP 成功/超时
{
    lwip_periodic_handle();
}
lwip_test_ui(2); //加载后半部分 UI
httpd_init(); //HTTP 初始化(默认开启 websever)
while(1)
{
    key=KEY_Scan(0);
    switch(key)
    {
        case KEY0_PRES://TCP Server 模式
            tcp_server_test();
            lwip_test_ui(3); //重新加载 UI
            break;
        case KEY1_PRES://TCP Client 模式
            tcp_client_test();
            lwip_test_ui(3); //重新加载 UI
            break;
        case KEY2_PRES://UDP 模式
            udp_demo_test();
    }
}
```

```
lwip_test_ui(3); //重新加载 UI
break;
}
lwip_periodic_handle();
delay_ms(2);
t++;
if(t==100)LCD_ShowString(30,230,200,16,16,"Please choose a mode!");
if(t==200)
{
    t=0;
    LCD_Fill(30,230,230,230+16,WHITE); //清除显示
    LED0=!LED0;
}
}
```

这里，我们开启了定时器 3，来给 LWIP 提供时钟，然后通过 lwip_comm_init 函数，初始化 LWIP，该函数处理包括：初始化 STM32F4 的以太网外设、初始化 LAN8720A、分配内存、使能 DHCP、添加并打开网卡等操作。

这里特别注意：因为我们配置 STM32F4 的网卡使用自动协商功能(双工模式和连接速度)，如果协商过程中遇到问题，则会进行多次重试，需要等待很久，而且如果协商失败，那么直接返回错误，导致 LWIP 初始化失败，因此一定要插上网线，然后 LWIP 才能初始化成功，否则肯定会初始化失败，而这个失败，不是硬件问题，是因为你没插网线的缘故!!!

在 LWIP 初始化成功后，进入 DHCP 获取 IP 状态，当 DHCP 获取成功后，显示开发板获取到的 IP 地址，然后开启 HTTP 服务。此时可以在浏览器输入开发板 IP 地址，登录 Web 控制界面，进行 Web Server 测试。

在主循环里面，我们可以通过按键选择：TCP Server 测试、TCP Client 测试和 UDP 测试等测试项目，主循环还调用了 lwip_periodic_handle 函数，周期性处理 LWIP 事务。

软件设计部分就为大家介绍到这里。

60.4 下载验证

在开始测试之前，我们先用网线（需自备）将开发板和电脑连接起来。

对于有路由器的用户，直接用网线连接路由器，同时电脑也连接路由器，即可完成电脑与开发板的连接设置。

对于没有路由器的用户，则直接用网线连接电脑的网口，然后设置电脑的本地连接属性，如图 60.4.1 所示：

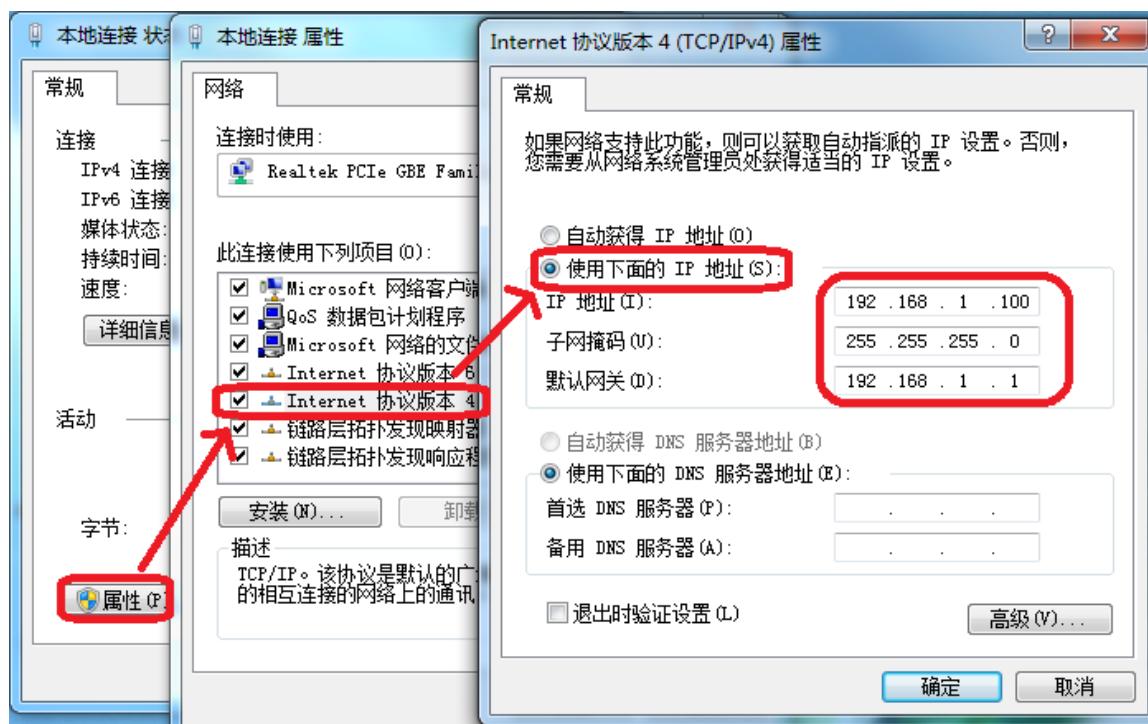


图 60.4.1 开发板与电脑直连时电脑本地连接属性设置

这里，我们设置 IPV4 的属性，设置 IP 地址为：192.168.1.100（100 是可以随意设置的，但是不能是 30 和 1）；子网掩码：255.255.255.0；网关：192.168.1.1；DNS 部分可以不用设置。

设置完后，点击确定，即可完成电脑端设置，这样开发板和电脑就可以通过互相通信了。

然后，在代码编译成功之后，我们通过下载代码到探索者 STM32F4 开发板上（这里我们以路由器连接方式介绍，下同，且假设 DHCP 获取 IP 成功），LCD 显示如图 60.4.2 所示界面：

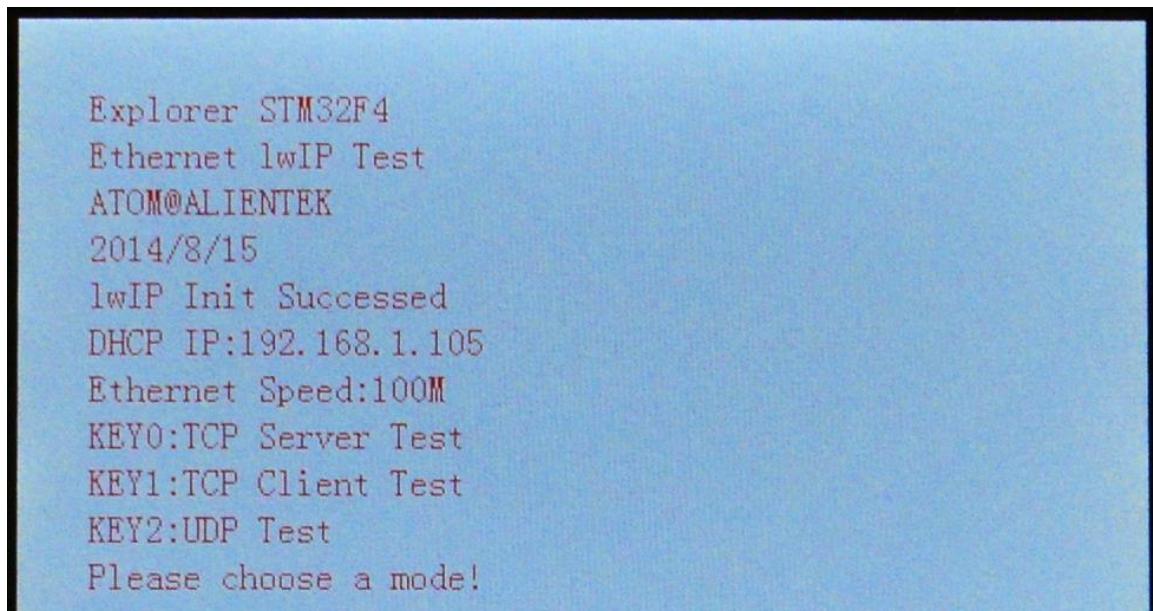


图 60.4.2 DHCP 获取 IP 成功

此时屏幕提示选择测试模式，可以选择 TCP Server、TCP Client 和 UDP 三项测试。不过，我们先来看看网络连接是否正常。从 60.4.2 可以看到，我们开发板通过 DHCP 获取到的 IP 地址为：192.168.1.105，因此，我们在电脑上先来 ping 一下这个 IP，看看能否 ping 通，以检查连

接是否正常 (Start→运行→CMD)，如图 60.4.3 所示：



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 <c> 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>ping 192.168.1.105

正在 Ping 192.168.1.105 具有 32 字节的数据:
来自 192.168.1.105 的回复: 字节=32 时间<1ms TTL=255

192.168.1.105 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (<0% 丢失),
往返行程的估计时间<以毫秒为单位>:
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms

C:\Users\Administrator>
```

图 60.4.3 ping 开发板 IP 地址

可以看到开发板所显示的 IP 地址，是可以 ping 通的，说明我们的开发板和电脑连接正常，可以开始后续测试了。

60.4.1 Web Server 测试

这个测试不需要任何操作来开启，开发板在获取 IP 成功（也可以使用静态 IP）后，即开启了 Web Server 功能。我们在浏览器输入：192.168.1.105（开发板显示的 IP 地址），即可进入一个 Web 界面，如图 60.4.1.1 所示：



图 60.4.1.1 Web Server 测试网页

该界面总共有 5 个子页面：主页、LED/BEEP 控制、ADC/内部温度传感器、RTC 实时时钟和联系我们等。登录 Web 时默认打开的是主页面，介绍了我们探索者 STM32F4 开发板的一些资源和特点和 LWIP 的一些简介。

点击：LED/BEEP 控制，进入该子页面，即可对开发板板载的 DS0（LED1）和蜂鸣器进行控制，如图 60.4.1.2 所示：



图 60.4.1.2 LED/BEEP 控制页面

此时，选择 ON，然后点击 SEND 按钮，即可点亮 LED1 或者打开蜂鸣器。同样，发送 OFF 即可关闭 LED1 或蜂鸣器。

点击：ADC/内部温度传感器，进入该子页面，会显示 ADC1 通道 5 的值和 STM32 内部温度传感器所测得的温度，如图 60.4.1.3 所示：



图 60.4.1.3 ADC/内部温度传感器测试页面

ADC1_CH5 是我们开发板多功能接口 ADC 的输入通道，默认连接在 TPAD 上，TPAD 带有上拉电阻，所以这里显示 3V 多，大家可以将 ADC 接其他地方来测量电压。同时，该界面还显示了内部温度传感器采集到的温度值。该界面每个一秒钟刷新一次。

点击：RTC 实时时钟，进入该子页面，会显示 STM32 内部 RTC 的时间和日期，如图 60.4.1.4 所示：



图 60.4.1.4 RTC 实时时钟测试页面

此界面显示了探索者 STM32F4 自带的 RTC 实时时钟的当前时间和日期等参数，每隔 1 秒钟刷新一次。

最后，点击联系我们，即可进入到 ALIENTEK 官方店铺，这里就不再介绍了。

60.4.2 TCP Server 测试

在提示界面，按 KEY0 即可进入 TCP Server 测试，此时，开发板作为 TCP Server。此时，LCD 屏幕上显示 Server IP 地址(就是开发板的 IP 地址)，Server 端口固定为：8088。如图 60.4.2.1 所示：

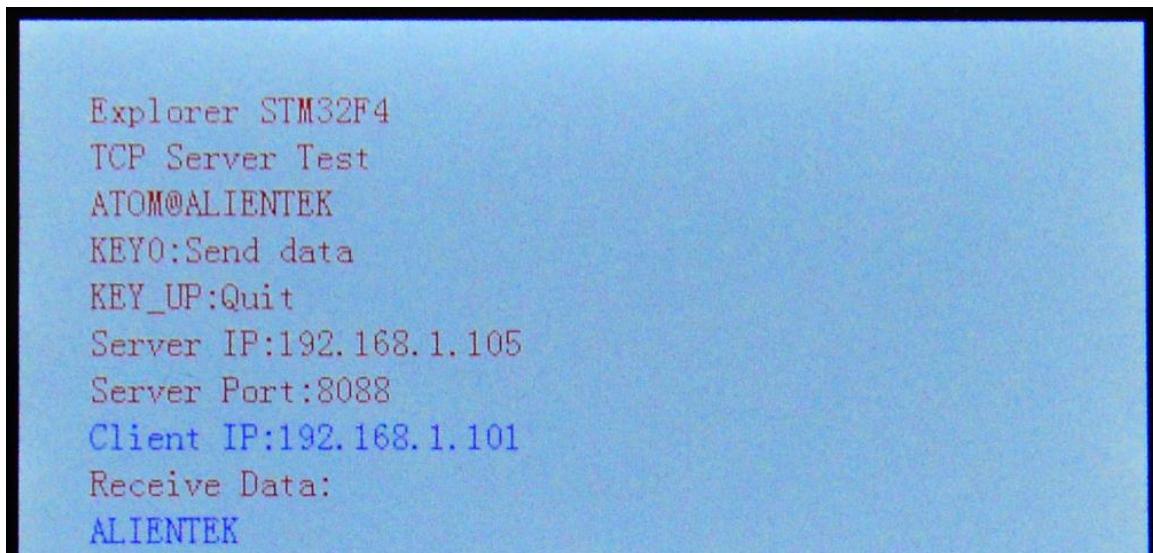


图 60.4.2.1 TCP Sever 测试界面

图中显示了 Server IP 地址是 192.168.1.105，Server 端口号是：8088。上位机配合我们测试，需要用到一个网络调试助手的软件，该软件在光盘→ 6，软件资料→软件→网络调试助手→网络调试助手 V3.8.exe。

我们在电脑端打开网络调试助手，设置协议类型为：TCP Client，服务器 IP 地址为：192.168.1.105，服务器端口号为：8088，然后点击连接，即可连上开发板的 TCP Sever，此时，开发板的液晶显示：Client IP:192.168.1.101（电脑的 IP 地址），如图 60.4.2.1 所示，而网络调试助手端则显示连接成功，如图 60.4.2.2 所示：

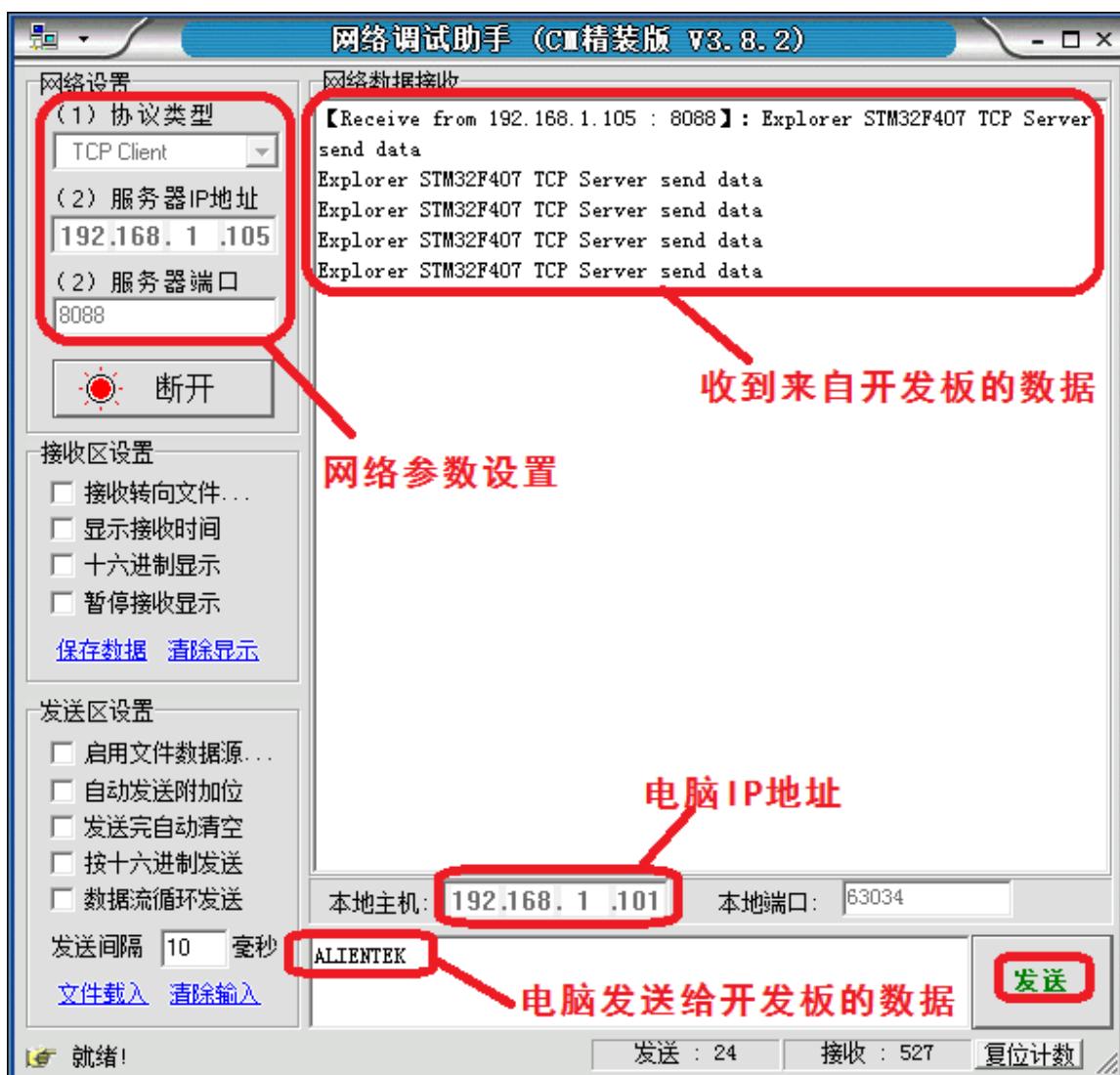


图 60.4.2.2 电脑端网络调试助手 TCP Client 测试界面

按开发板的 KEY0 按键，即可发送数据给电脑。同样，电脑端输入数据，也可以通过网络调试助手发送给开发板。如图 60.4.2.1 和图 60.4.2.2 所示。按 KEY_UP 按键，可以退出 TCP Server 测试，返回选择界面。

60.4.3 TCP Client 测试

在提示界面，按 KEY1 即可进入 TCP Client 测试，此时，先进入一个远端 IP 设置界面，也就是 Client 要去连接的 Server 端的 IP 地址。通过 KEY0/KEY2 可以设置 IP 地址，通过 60.4.3.2 节的测试，我们知道电脑的 IP 是 192.168.1.101，所以我们这里设置 Client 要连接的远端 IP 为 192.168.1.101，如图 60.4.3.1 所示：

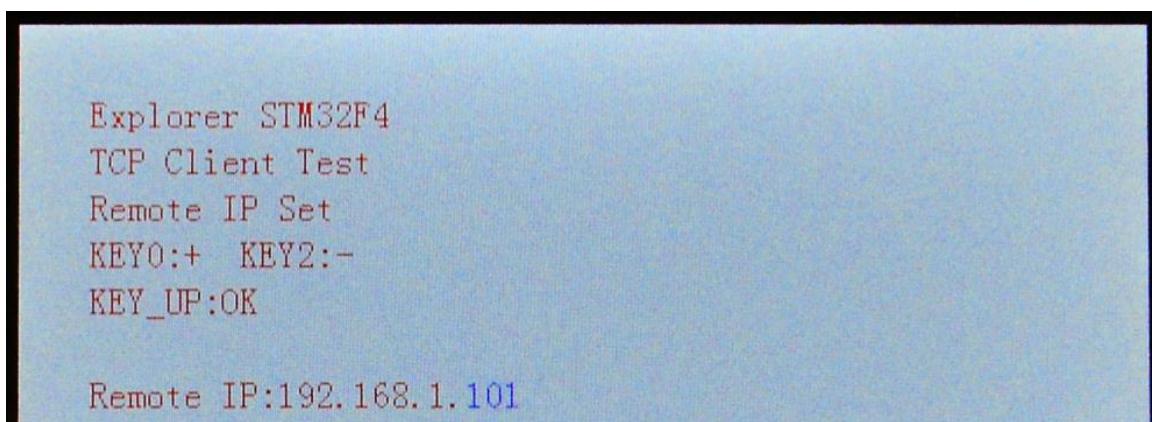


图 60.4.3.1 远端 IP 地址设置

设置好之后，按 KEY_UP，确认，进入 TCP Client 测试界面。开始的时候，屏幕显示 Disconnected。然后我们在电脑端打开网络调试助手，设置协议类型为：TCP Server，本地 IP 地址为：192.168.1.101（电脑 IP），本地端口号为：8087，然后点击连接，开启电脑端的 TCP Server 服务，如图 60.4.3.2 所示：

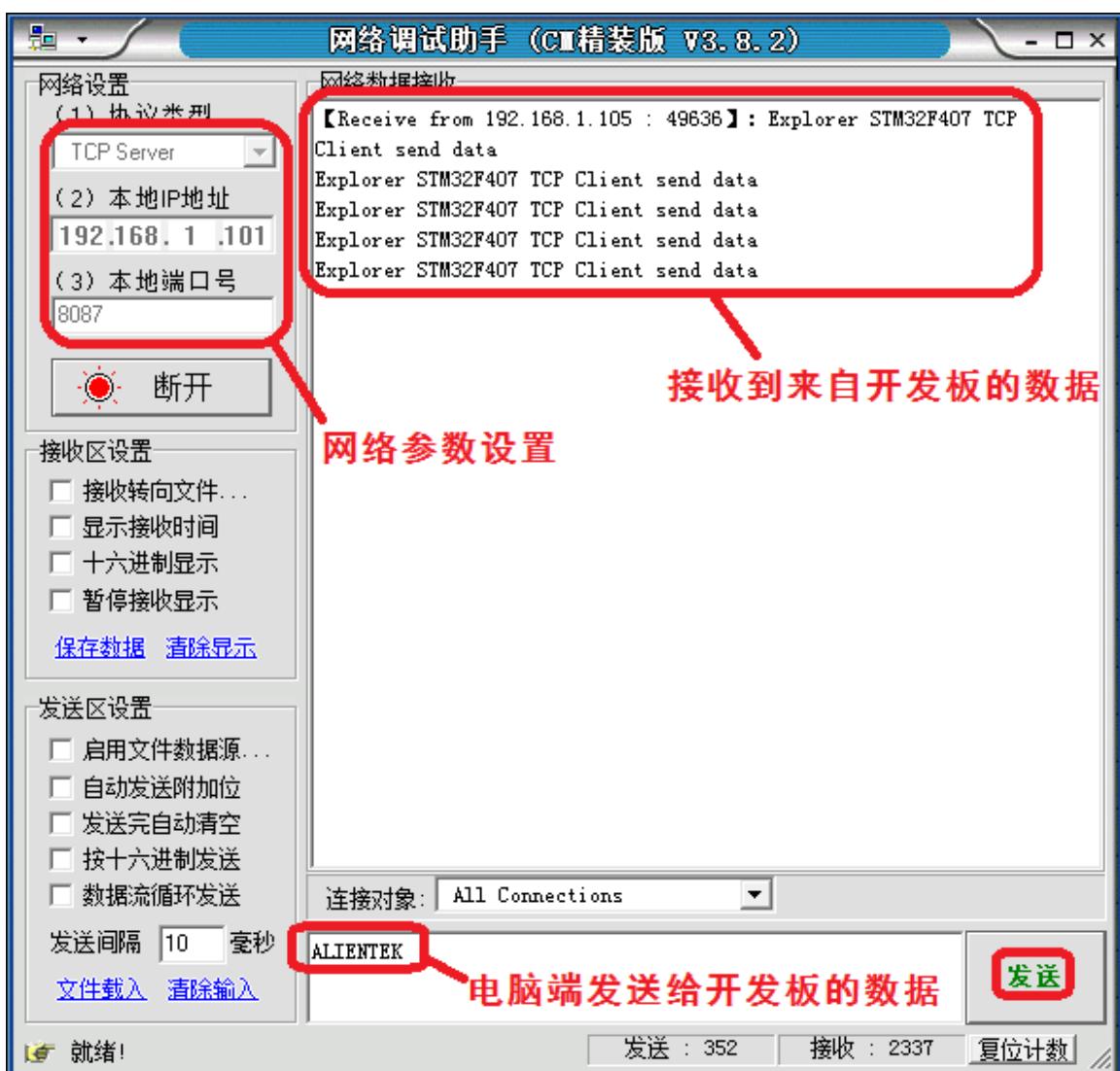


图 60.4.3.2 电脑端网络调试助手 TCP Server 测试界面

在电脑端开启 Server 后，稍等片刻，开发板的 LCD 即显示 Connected，如图 60.4.3.3 所示：

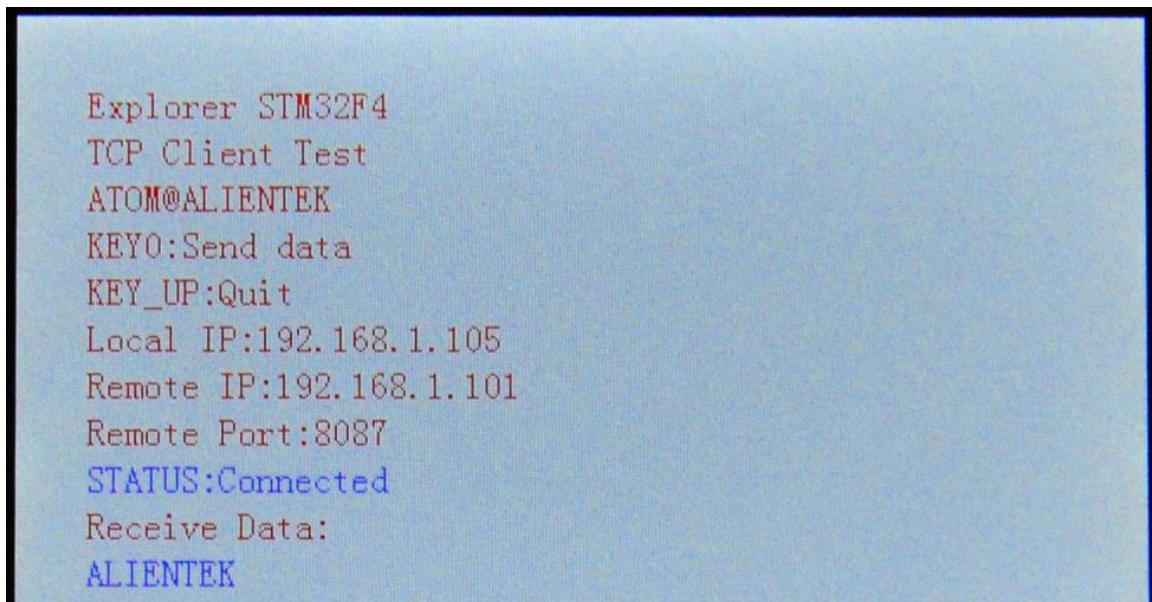


图 60.4.3.3 TCP Client 测试界面

在连接成功后，电脑和开发板即可互发数据，同样开发板还是按 KEY0 发送数据给电脑，测试结果如图 60.4.3.2 和图 60.4.3.3 所示。按 KEY_UP 按键，可以退出 TCP Client 测试，返回选择界面。

60.4.4 UDP 测试

在提示界面，按 KEY2 即可进入 UDP 测试，UDP 测试同 TCP Client 测试一样，要先设置远端 IP 地址，设置好之后，进入 UDP 测试界面，如图 60.4.4.1 所示：

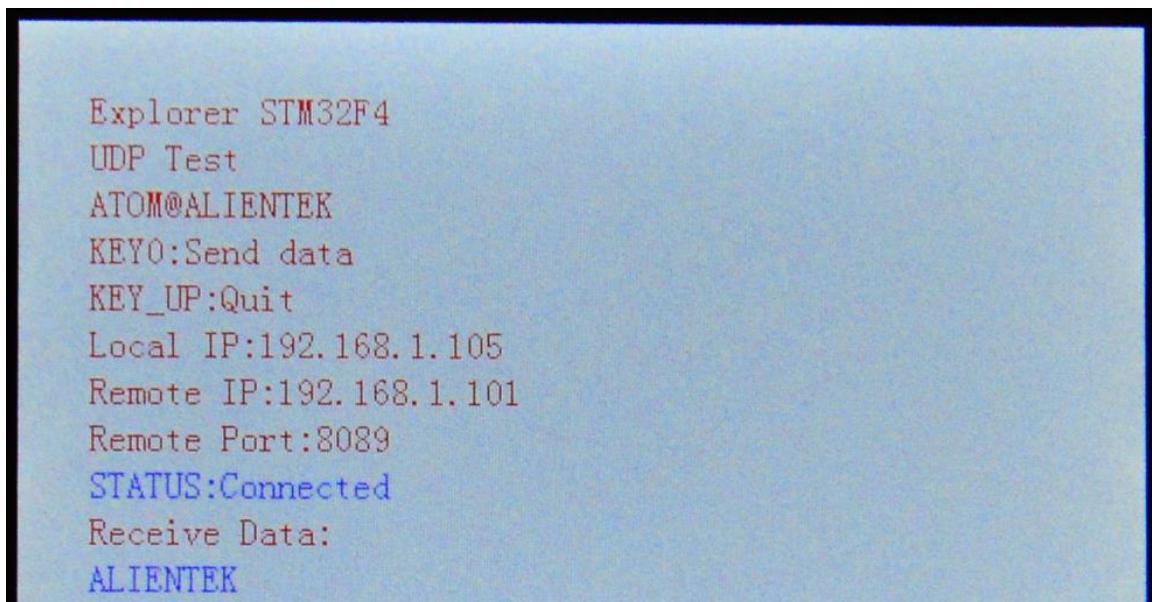


图 60.4.4.1 UDP 测试界面

可以看到，UDP 测试时我们要连接的端口号为：8089，所以网络调试助手需要设置端口号为：8089。另外，UDP 不是基于连接的传输协议，所以，这里直接就显示 Connected 了。在电

电脑端打开网络调试助手，设置协议类型为：UDP，本地 IP 地址为：192.168.1.101（电脑 IP），本地端口号为：8089，然后点击连接，开启电脑端的 UDP 服务，如图 60.4.4.2 所示：

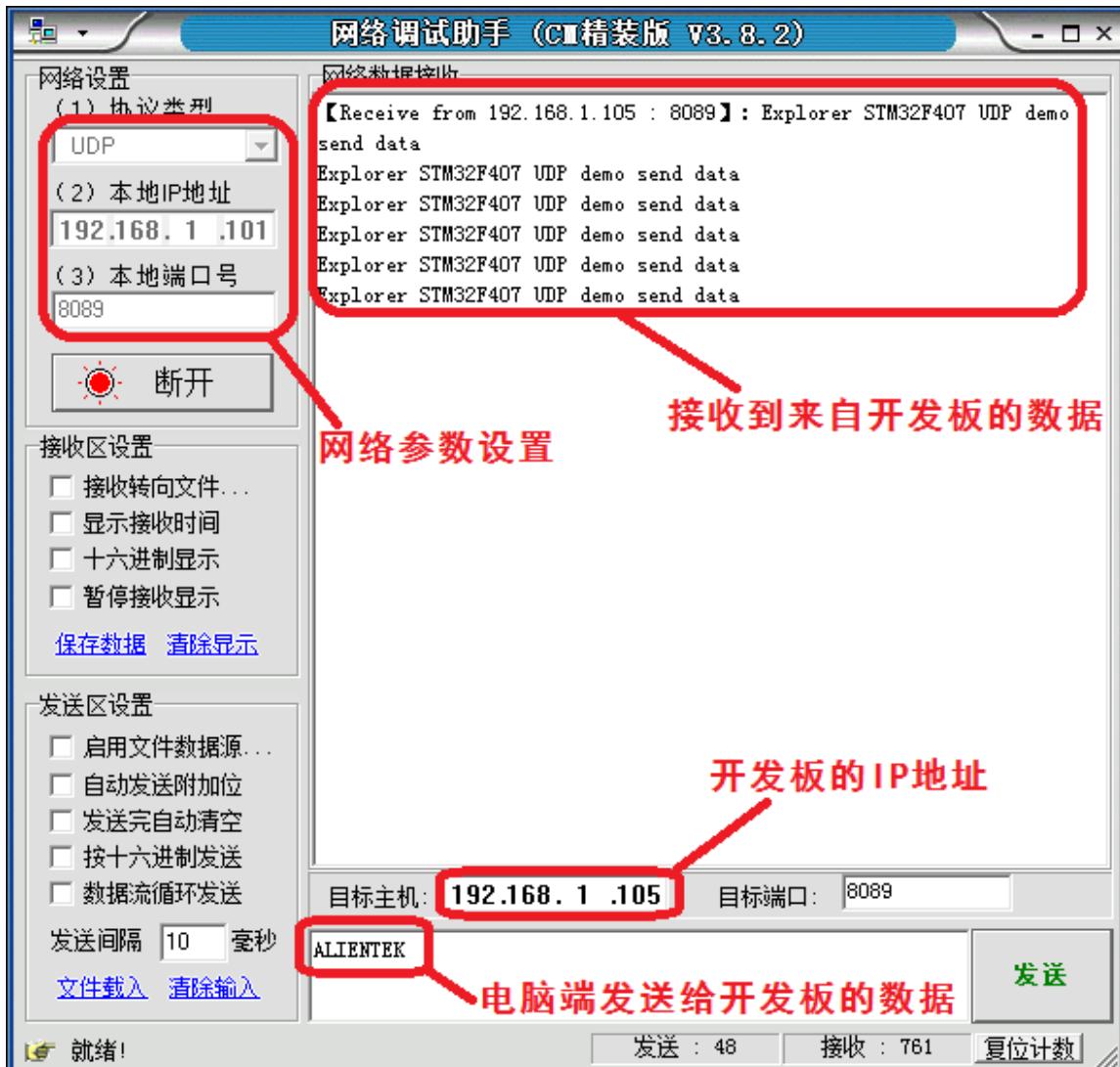


图 60.4.4.2 电脑端网络调试助手 UDP 测试界面

然后，我们先按开发板的 KEY0，发送一次数据给电脑端网络调试助手，这样电脑端网络调试助手便可以识别出开发板的 IP 地址，然后就可以互相发送数据了。按 KEY_UP 按键，可以退出 UDP 测试，返回选择界面。

第六十一章 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：

- 61.1 UCOSII 简介
- 61.2 硬件设计
- 61.3 软件设计
- 61.4 下载验证

61.1 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。

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 61.1.1 所示：

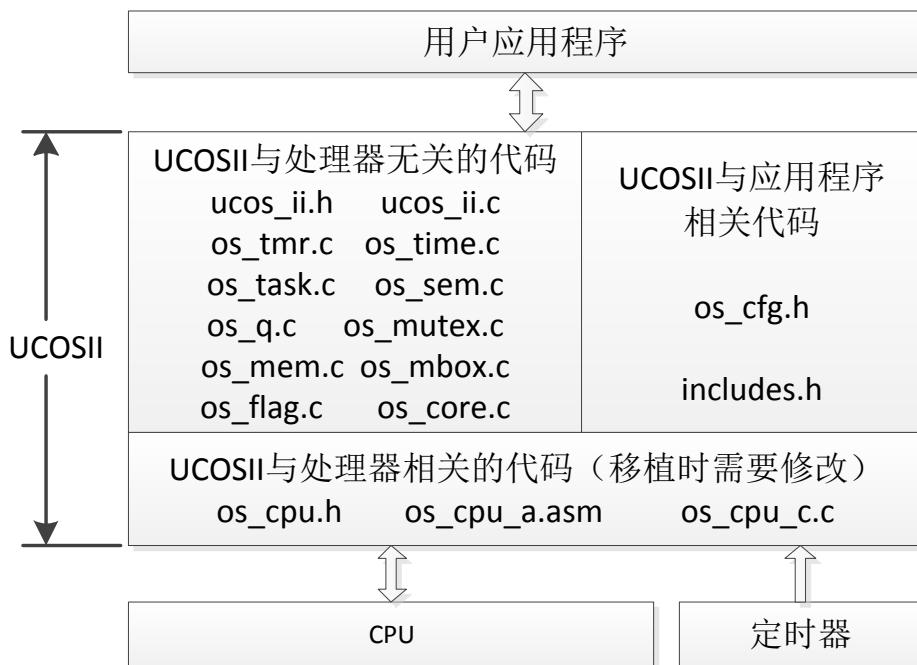


图 61.1.1 UCOSII 体系结构图

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 STM32F4 的 SYSTICK 定时器来提供 UCOSII 时钟节拍。

关于 UCOSII 在 STM32F4 的详细移植过程, 请参考光盘资料: 《ALIENTEK STM32F4 UCOSII 移植教程.pdf》, 教程路径: 光盘 → 6, 软件资料 → UCOS 学习资料 → ALIENTEK STM32F4 UCOSII 移植教程.pdf。这里我们就不详细介绍。

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)。

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,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：音乐播放器实验，在音乐播放的时候，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，音频可能出现停顿（尤其是采样率高的时候），这主要是歌词显示占用太长时间，导致 IIS 数据无法及时填充而停顿。而如果用 UCOSII 来处理，那么我们可以分 2 个任务，音乐播放一个任务（优先级高），歌词显示一个任务（优先级低）。这样，由于音乐播放任务的优先级高于歌词显示任务，音乐播放任务可以打断歌词显示任务，从而及时给 IIS 填充数据，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（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 61.1.2 所示：

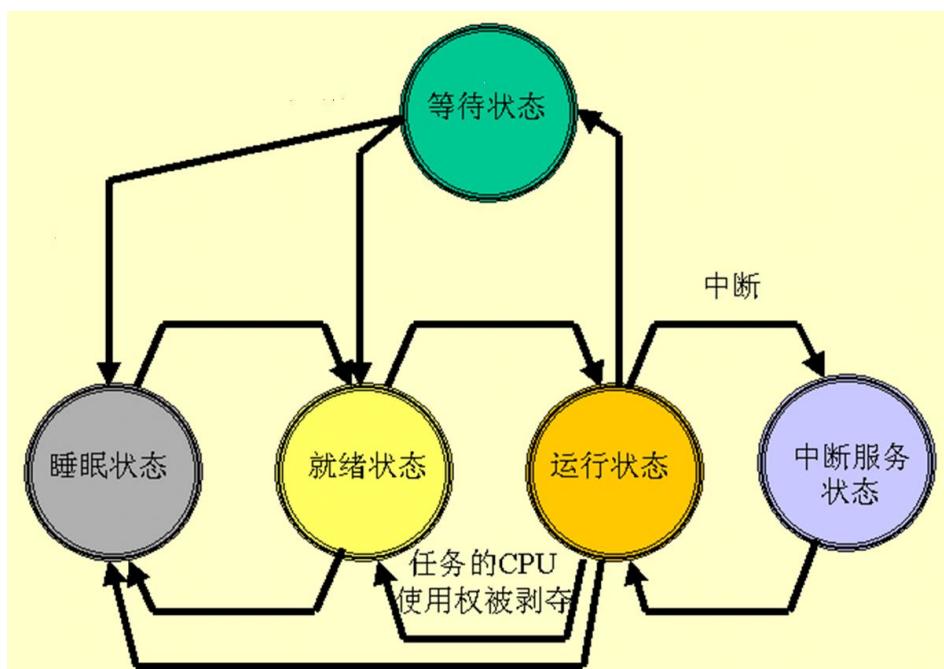


图 61.1.2 UCOSII 任务状态转换关系

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：OSTaskCreate 和 OSTaskCreateExt，我们一般用 OSTaskCreate 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 STM32F4 上面运行 UCOSII 的步骤：

1) 移植 UCOSII

要想 UCOSII 在 STM32F4 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（移植过程参考光盘：STM32F4 UCOS 开发手册.pdf）。

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，通过调用 OSTaskCreate 函数创建我们的任务。

4) 启动 UCOSII

调用 OSStart，启动 UCOSII。

通过以上 4 个步骤，UCOSII 就开始在 STM32F4 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。

61.2 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

这个我们在前面已经介绍过了。

61.3 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 61.3.1 所示：

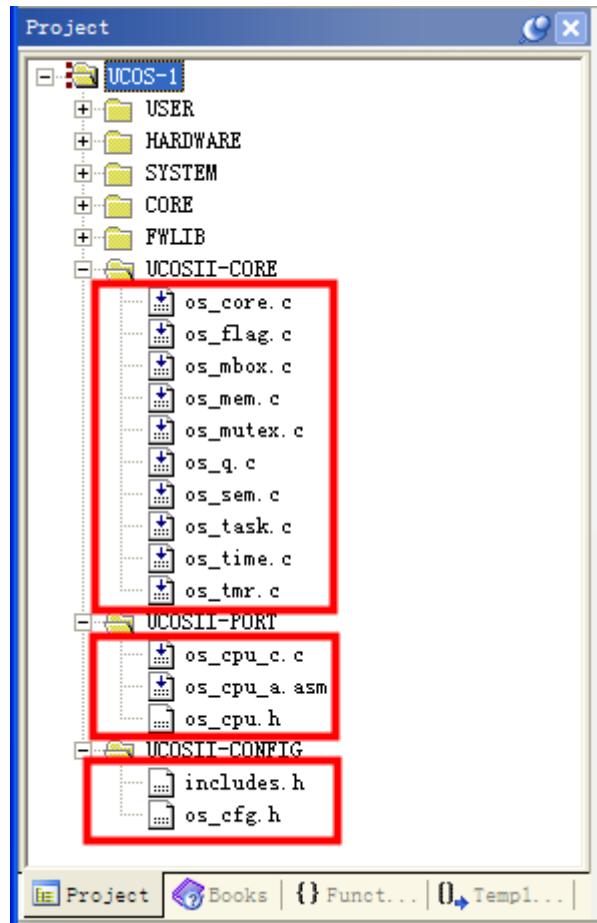


图 61.3.1 添加 UCOSII 源码后的工程

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了，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`，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

//LED0 任务
#define LED0_TASK_PRIO           7       //设置任务优先级
#define LED0_STK_SIZE             64      //设置任务堆栈大小
OS_STK LED0_TASK_STK[LED0_STK_SIZE]; //任务堆栈
void led0_task(void *pdata);          //任务函数

//LED1 任务
#define LED1_TASK_PRIO           6       //设置任务优先级
#define LED1_STK_SIZE             64      //设置任务堆栈大小
OS_STK LED1_TASK_STK[LED1_STK_SIZE]; //任务堆栈
void led1_task(void *pdata);          //任务函数

//main 函数
int main(void)
{
    delay_init(168);           //初始化延时函数
    LED_Init();                //初始化 LED 时钟
    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();                 //启动 UCOSII
}

//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。该部分代码我们创建了 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。

61.4 下载验证

在代码编译成功之后，我们通过下载代码到探索者 STM32F4 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 56 UCOSII 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：

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(168);           //初始化延时函数
    uart_init(115200);
    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 KEY1 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 KEY0 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 KEY2，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。

第六十二章 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：

- 62.1 UCOSII 信号量和邮箱简介
- 62.2 硬件设计
- 62.3 软件设计
- 62.4 下载验证

62.1 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 62.1.1 所示：



图 62.1.1 两个任务使用事件进行通信的示意图

在图 62.1.1 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，发送信号量通过 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。

62.2 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/KEY_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。

62.3 软件设计

本章，我们在第三十三章实验（实验 28）的基础上修改。首先，是 UCOSII 代码的添加，具体方法同上一章一模一样，本章就不再详细介绍。

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

//触摸屏任务
#define TOUCH_TASK_PRIO          7       //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

//LED 任务
#define LED_TASK_PRIO             6       //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

//蜂鸣器任务
#define BEEP_TASK_PRIO            5       //设置任务优先级
#define BEEP_STK_SIZE              64      //设置任务堆栈大小
OS_STK BEEP_TASK_STK[BEEP_STK_SIZE]; //任务堆栈
void beep_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

//按键扫描任务
#define KEY_TASK_PRIO              3      //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,"Explorer 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:DS1 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)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init();          //初始化 LED
    BEEP_Init();         //蜂鸣器初始化
    KEY_Init();          //按键初始化
    LCD_Init();          //LCD 初始化
    tp_dev.init();        //触摸屏初始化
    ucos_load_main_ui(); //加载主界面
    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();           //启动 UCOSII
}
/////////////////////////////
//以下代码为支持画粗线而编写，如不需要，可以去掉。
//画水平线
//x0,y0:坐标
```

```
//len:线长度
//color:颜色
void gui_draw_hline(u16 x0,u16 y0,u16 len,u16 color)
{
    if(len==0)return;
    LCD_Fill(x0,y0,x0+len-1,y0,color);
}

//画实心圆
//x0,y0:坐标
//r:半径
//color:颜色
void gui_fill_circle(u16 x0,u16 y0,u16 r,u16 color)
{
    u32 i;
    u32 imax = ((u32)r*707)/1000+1;
    u32 sqmax = (u32)r*(u32)r+(u32)r/2;
    u32 x=r;
    gui_draw_hline(x0-r,y0,2*r,color);
    for (i=1;i<=imax;i++)
    {
        if ((i*i+x*x)>sqmax)// draw lines from outside
        {
            if (x>imax)
            {
                gui_draw_hline (x0-i+1,y0+x,2*(i-1),color);
                gui_draw_hline (x0-i+1,y0-x,2*(i-1),color);
            }
            x--;
        }
        // draw lines from inside (center)
        gui_draw_hline(x0-x,y0+i,2*x,color);
        gui_draw_hline(x0-x,y0-i,2*x,color);
    }
}

//两个数之差的绝对值
//x1,x2: 需取差值的两个数
//返回值: |x1-x2|
u16 my_abs(u16 x1,u16 x2)
{
    if(x1>x2)return x1-x2;
    else return x2-x1;
}

//画一条粗线
```

```
//(x1,y1),(x2,y2):线条的起始坐标
//size: 线条的粗细程度
//color: 线条的颜色
void lcd_draw_bline(u16 x1, u16 y1, u16 x2, u16 y2,u8 size,u16 color)
{
    u16 t;
    int xerr=0,yerr=0,delta_x,delta_y,distance;
    int incx,incy,uRow,uCol;
    if(x1<size|| x2<size||y1<size|| y2<size)return;
    delta_x=x2-x1; //计算坐标增量
    delta_y=y2-y1;
    uRow=x1; uCol=y1;
    if(delta_x>0)incx=1; //设置单步方向
    else if(delta_x==0)incx=0;//垂直线
    else {incx=-1;delta_x=-delta_x;}
    if(delta_y>0)incy=1;
    else if(delta_y==0)incy=0;//水平线
    else{incy=-1;delta_y=-delta_y;}
    if( delta_x>delta_y)distance=delta_x; //选取基本增量坐标轴
    else distance=delta_y;
    for(t=0;t<=distance+1;t++)//画线输出
    {
        gui_fill_circle(uRow,uCol,size,color);//画点
        xerr+=delta_x ; yerr+=delta_y ;
        if(xerr>distance) { xerr-=distance; uRow+=incx; }
        if(yerr>distance) { yerr-=distance; uCol+=incy; }
    }
}
////////////////////////////////////////////////////////////////
//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(touch_task,(void *)0,(OS_STK*)&TOUCH_TASK_STK[TOUCH_STK_SIZE-1],TOUCH_TASK_PRIO);
    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(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) //LED0 亮  
        {  
            t=0;LED0=0;  
        }  
    }  
}  
//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)  
{  
    u32 cpu_sr;  
    u16 lastpos[2]; //最后一次的数据  
    while(1)  
    {  
        tp_dev.scan(0);  
        if(tp_dev.sta&TP_PRES_DOWN) //触摸屏被按下  
        {
```

```
if(tp_dev.x[0]<lcddev.width&&tp_dev.y[0]<lcddev.height&&tp_dev.y[0]>120)
{
    if(lastpos[0]==0xFFFF)
    {
        lastpos[0]=tp_dev.x[0];
        lastpos[1]=tp_dev.y[0];
    }
    OS_ENTER_CRITICAL();//进入临界段,防止打断 LCD 操作,导致乱序.
    lcd_draw_bline(lastpos[0],lastpos[1],tp_dev.x[0],tp_dev.y[0],2,RED);//画线
    OS_EXIT_CRITICAL();
    lastpos[0]=tp_dev.x[0];
    lastpos[1]=tp_dev.y[0];
}
}else lastpos[0]=0xFFFF;//没有触摸
delay_ms(5);
}

//主任务
void main_task(void *pdata)
{
    u32 key=0;
    u8 err,semmask=0,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
                if((tp_dev.touchtype&0X80)==0)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。

62.4 下载验证

在代码编译成功之后，我们通过下载代码到探索者 STM32F4 开发板上，可以看到 LCD 显

示界面如图 62.4.1 所示：

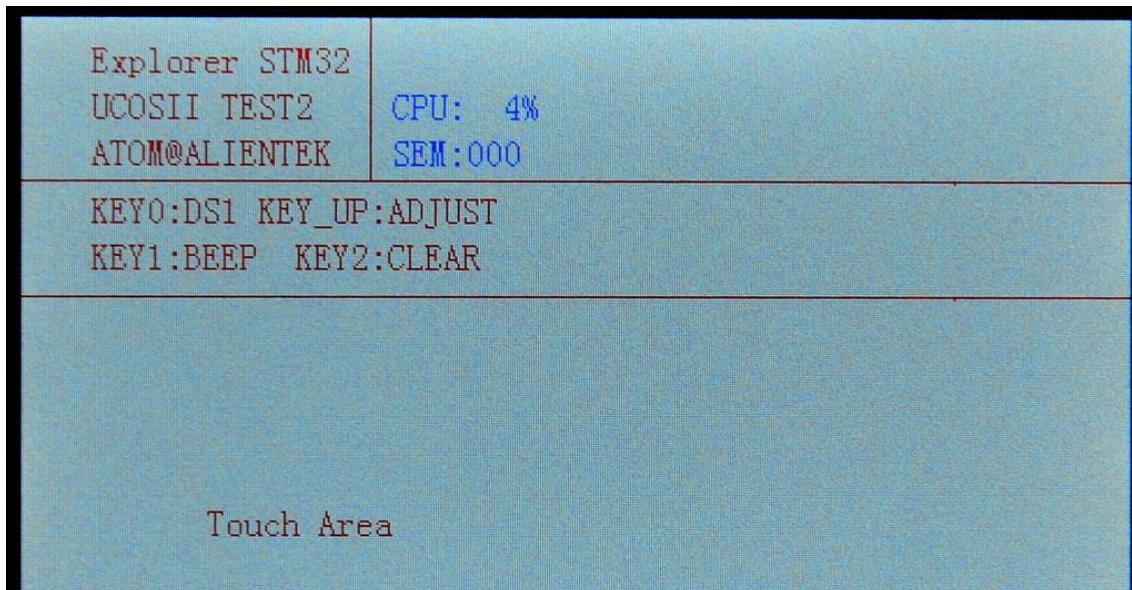


图 62.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率仅为 2%。此时通过在触摸区域画图，可以看到 CPU 使用率飙升(20%多)，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 KEY_UP 可以进入校准程序，进行触摸屏校准（注意，电容触摸屏不需要校准，所以如果是电容屏，按 KEY_UP，就相当于清屏一次的效果，不会进行校准）。

第六十三章 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：

- 63.1 UCOSII 消息队列、信号量集和软件定时器简介
- 63.2 硬件设计
- 63.3 软件设计
- 63.4 下载验证

63.1 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 OS_EventType 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 63.1.1 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 OS_EventPtr 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。

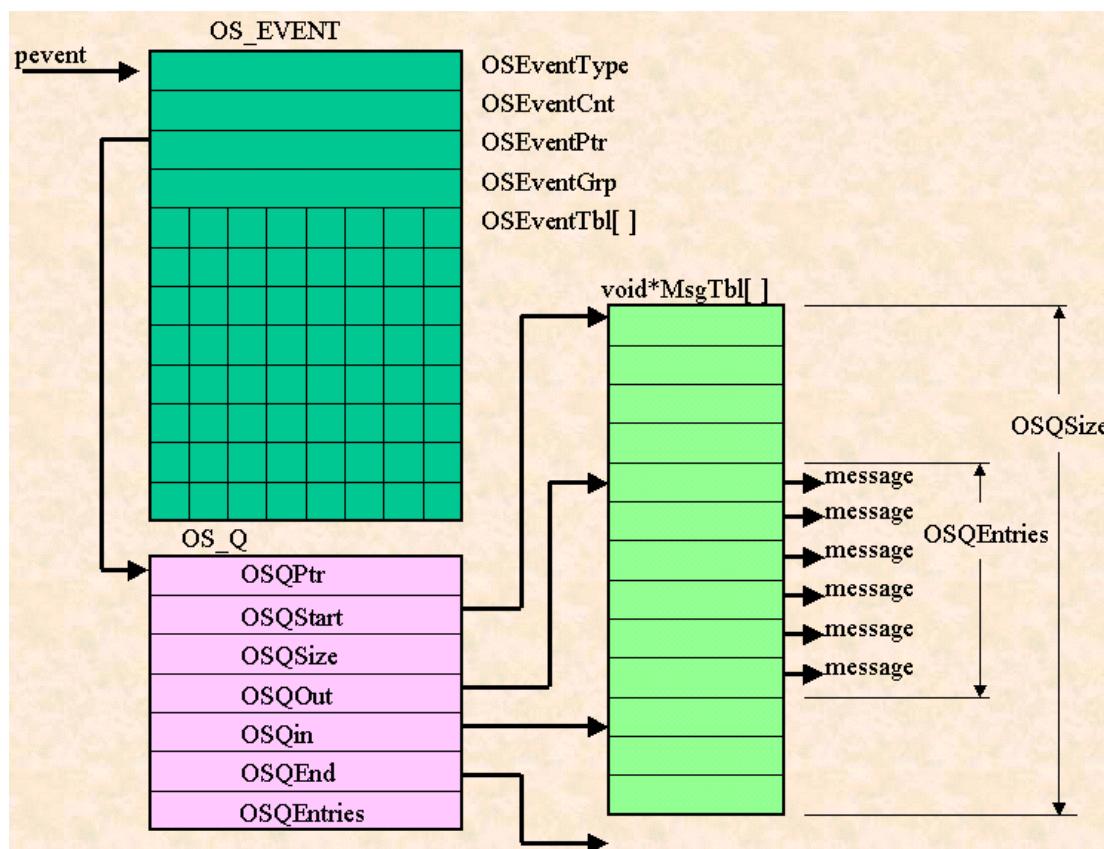


图 63.1.1 消息队列的数据结构

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 63.1.1 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
OSQEntries	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 63.1.1 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 63.1.2 所示的循环的队列。

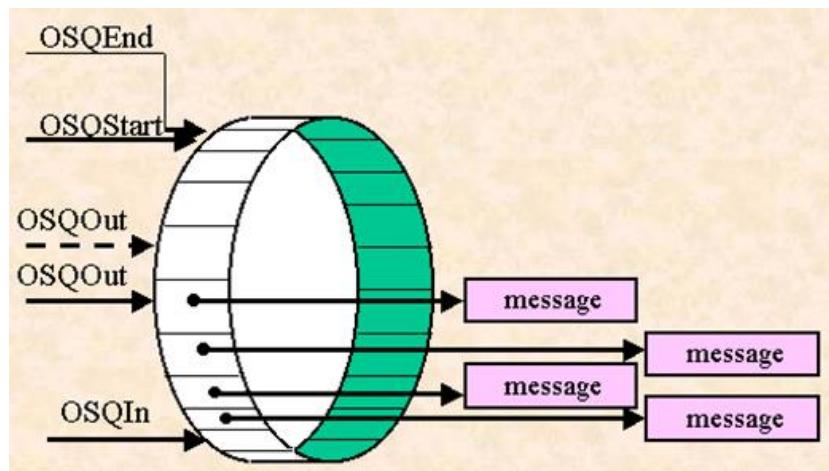


图 63.1.2 消息指针数组构成的环形数据缓冲区

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);
```

```
INT8U OSQPostFront (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 63.1.3 所示

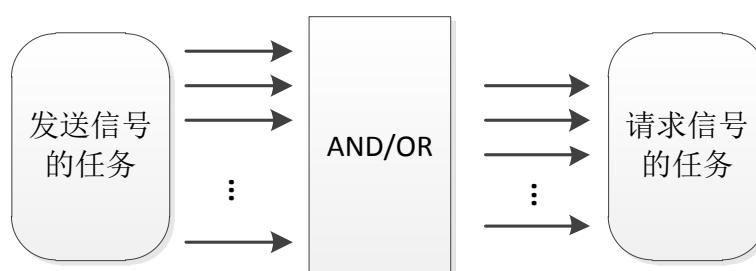


图 63.1.3 信号量集示意图

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 63.1.2 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 63.1.2 OSFlagNodeWaitType 可选值及其意义

OSFlagFlags、OSFlagNodeFlags、OSFlagNodeWaitType 三者的关系如图 63.1.4 所示：

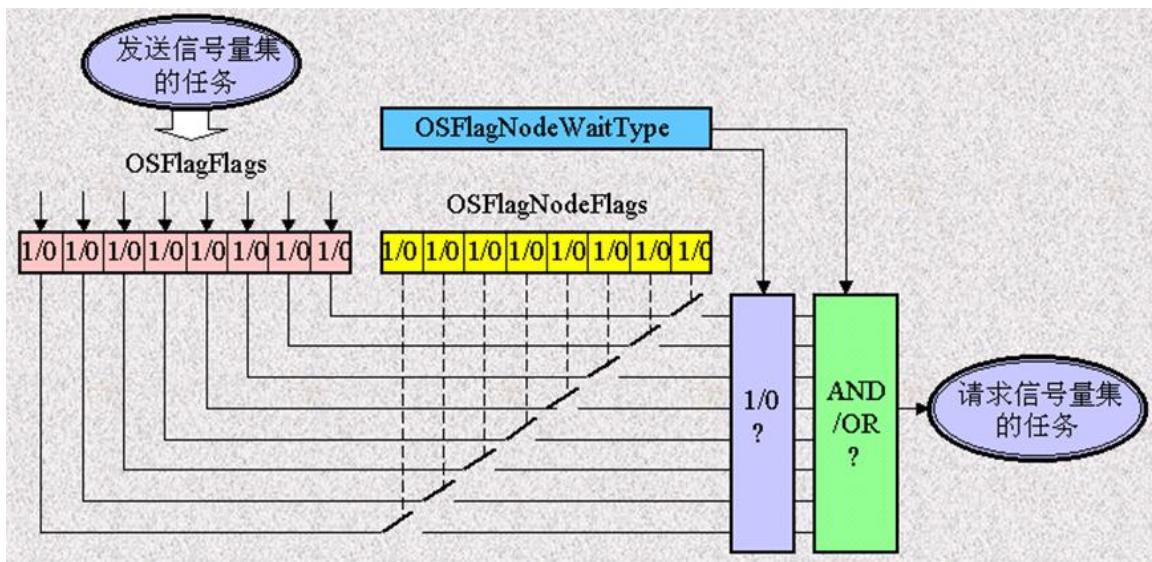


图 63.1.4 标志组与等待任务共同完成信号量集的逻辑运算及控制

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 63.1.5 所示:

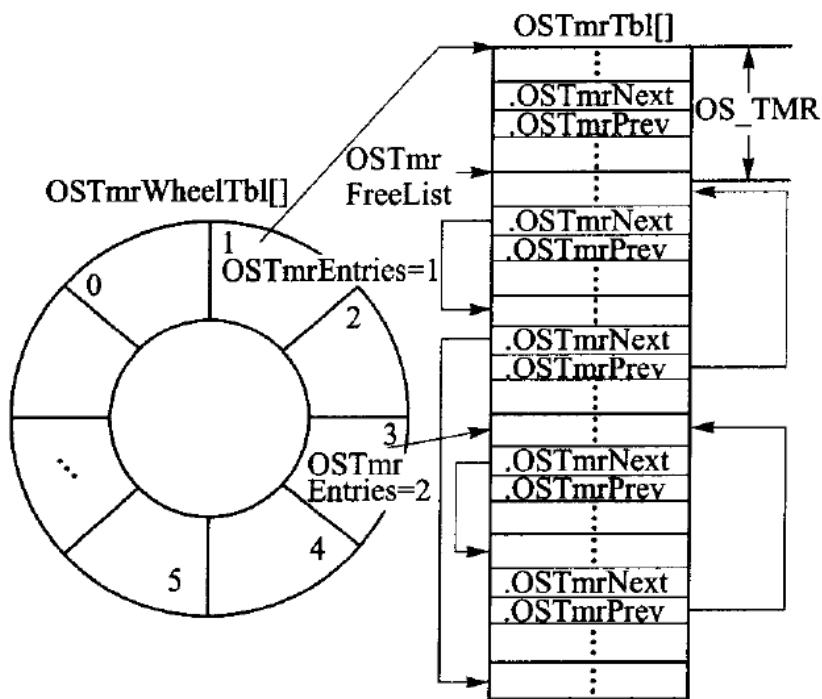


图 63.1.5 软件定时器管理所需的数据结构示意图

`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 63.1.6 所示。

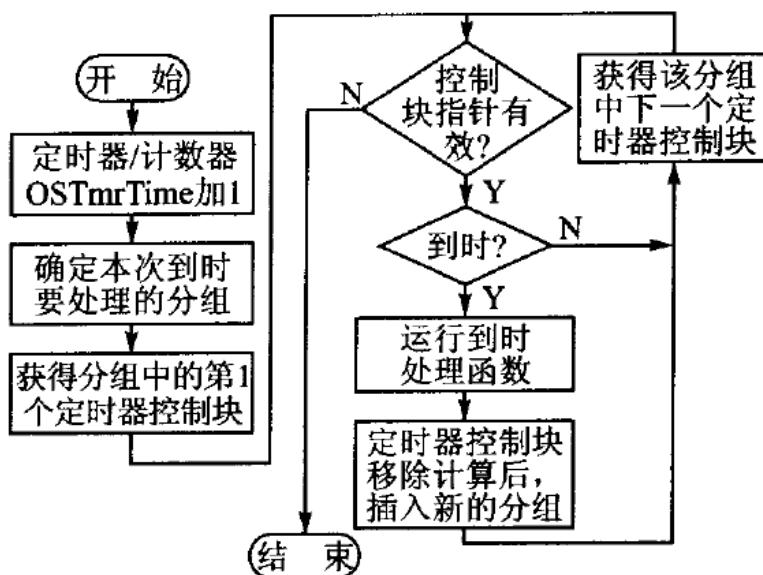


图 63.1.6 软件定时器管理任务流程

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 OSTmrMatch 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:

```
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。

63.2 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/KEY_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。

63.3 软件设计

本章, 我们在第四十二章实验 (实验 37) 的基础上修改, 首先, 是 UCOSII 代码的添加, 具体方法同第 61 章一模一样, 本章就不再详细介绍了。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 文件了，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               128   //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             128  //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                128   //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                  128   //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,my_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(131,221,lcddev.width-1,lcddev.height-1,RED); break;
        case 1: LCD_Fill(131,221,lcddev.width-1,lcddev.height-1,GREEN); break;
        case 2: LCD_Fill(131,221,lcddev.width-1,lcddev.height-1,BLUE); break;
        case 3: LCD_Fill(131,221,lcddev.width-1,lcddev.height-1,MAGENTA); break;
        case 4: LCD_Fill(131,221,lcddev.width-1,lcddev.height-1,GBLUE); break;
        case 5: LCD_Fill(131,221,lcddev.width-1,lcddev.height-1,YELLOW); break;
        case 6: LCD_Fill(131,221,lcddev.width-1,lcddev.height-1,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,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,"Explorer 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(130,0,130,70);
    LCD_DrawLine(0,120	lcddev.width,120);
    LCD_DrawLine(0,220	lcddev.width,220);
    LCD_DrawLine(130,120,130, 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+130,125,240,16,16,"FLAGS");//信号量集
    LCD_ShowString(5,225,240,16,16,"TOUCH"); //触摸屏
    LCD_ShowString(5+130,225,240,16,16,"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)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //初始化延时函数
    uart_init(115200); //初始化串口波特率为 115200
    LED_Init(); //初始化 LED
    LCD_Init(); //初始化 LCD
    BEEP_Init(); //蜂鸣器初始化
    KEY_Init(); //按键初始化
    TPAD_Init(8); //初始化 TPAD
    my_mem_init(SRAMIN); //初始化内部内存池
    tp_dev.init(); //初始化触摸屏
    ucos_load_main_ui(); //加载主界面
    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(); //启动 UCOSII
}

//////////////////////////////画水平线
//x0,y0:坐标
//len:线长度
//color:颜色
void gui_draw_hline(u16 x0,u16 y0,u16 len,u16 color)
{
    if(len==0) return;
    LCD_Fill(x0,y0,x0+len-1,y0,color);
}

//////////////////////////////画实心圆
//x0,y0:坐标
//r:半径
//color:颜色
void gui_fill_circle(u16 x0,u16 y0,u16 r,u16 color)
{
    u32 i;
    u32 imax = ((u32)r*707)/1000+1;
    u32 sqmax = (u32)r*(u32)r+(u32)r/2;
    u32 x=r;
    gui_draw_hline(x0-r,y0,2*r,color);
    for (i=1;i<=imax;i++)
    {
        if ((i*i+x*x)>sqmax)// draw lines from outside
```

```
{  
    if (x>imax)  
    {  
        gui_draw_hline (x0-i+1,y0+x,2*(i-1),color);  
        gui_draw_hline (x0-i+1,y0-x,2*(i-1),color);  
    }  
    x--;  
}  
// draw lines from inside (center)  
gui_draw_hline(x0-x,y0+i,2*x,color);  
gui_draw_hline(x0-x,y0-i,2*x,color);  
}  
}  
//两个数之差的绝对值  
//x1,x2: 需取差值的两个数  
//返回值: |x1-x2|  
u16 my_abs(u16 x1,u16 x2)  
{  
    if(x1>x2) return x1-x2;  
    else return x2-x1;  
}  
//画一条粗线  
//(x1,y1),(x2,y2):线条的起始坐标  
//size: 线条的粗细程度  
//color: 线条的颜色  
void lcd_draw_bline(u16 x1, u16 y1, u16 x2, u16 y2,u8 size,u16 color)  
{  
    u16 t;  
    int xerr=0,yerr=0,delta_x,delta_y,distance;  
    int incx,incy,uRow,uCol;  
    if(x1<size|| x2<size||y1<size|| y2<size)return;  
    delta_x=x2-x1; //计算坐标增量  
    delta_y=y2-y1;  
    uRow=x1; uCol=y1;  
    if(delta_x>0)incx=1; //设置单步方向  
    else if(delta_x==0)incx=0;//垂直线  
    else {incx=-1;delta_x=-delta_x;}  
    if(delta_y>0)incy=1;  
    else if(delta_y==0)incy=0;//水平线  
    else {incy=-1;delta_y=-delta_y;}  
    if( delta_x>delta_y)distance=delta_x; //选取基本增量坐标轴  
    else distance=delta_y;  
    for(t=0;t<=distance+1;t++)//画线输出
```

```
{  
    gui_fill_circle(uRow,uCol,size,color);//画点  
    xerr+=delta_x ; yerr+=delta_y ;  
    if(xerr>distance) { xerr-=distance; uRow+=incx; }  
    if(yerr>distance) { yerr-=distance; uCol+=incy; }  
}  
}  
/////////////////////////////  
//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) //LED0 亮  
    }
```

```
{  
    t=0;LED0=0;  
}  
}  
}  
//触摸屏任务  
void touch_task(void *pdata)  
{  
    u32 cpu_sr;  
    u16 lastpos[2];      //最后一次的数据  
    while(1)  
    {  
        tp_dev.scan(0);  
        if(tp_dev.sta&TP_PRES_DOWN)      //触摸屏被按下  
        {  
            if(tp_dev.x[0]<(130-1)&&tp_dev.y[0]<lcddev.height&&tp_dev.y[0]>(220+1))  
            {  
                if(lastpos[0]==0xFFFF)  
                {  
                    lastpos[0]=tp_dev.x[0];  
                    lastpos[1]=tp_dev.y[0];  
                }  
                OS_ENTER_CRITICAL();//进入临界段,防止打断 LCD 操作,导致乱序.  
                lcd_draw_bline(lastpos[0],lastpos[1],tp_dev.x[0],tp_dev.y[0],2,RED);//画线  
                OS_EXIT_CRITICAL();  
                lastpos[0]=tp_dev.x[0];  
                lastpos[1]=tp_dev.y[0];  
            }  
        }  
        else lastpos[0]=0xFFFF;//没有触摸按下的时候  
        delay_ms(5);  
    }  
}  
//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://控制 DS1
                LED1=!LED1;
                break;
            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://清除
                LCD_Fill(0,221,129,lcddev.height,WHITE);
                break;
            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 2
if((tp_dev.touchtype&0X80)==0)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");//提示关闭了
}
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
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)
    {
        key=KEY_Scan(0);
        if(key==0)
        {
            if(TPAD_Scan(0))key=5;
        }
        if(key)OSMboxPost(msg_key,(void*)key);//发送消息
        delay_ms(10);
    }
}
```

本章 main.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 start_task 和 led_task 是 64，其他都是 128。

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 63.2 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；KEY_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。

63.4 下载验证

在代码编译成功之后，我们通过下载代码到探索者 STM32F4 开发板上，可以看到 LCD 显示界面如图 63.4.1 所示：

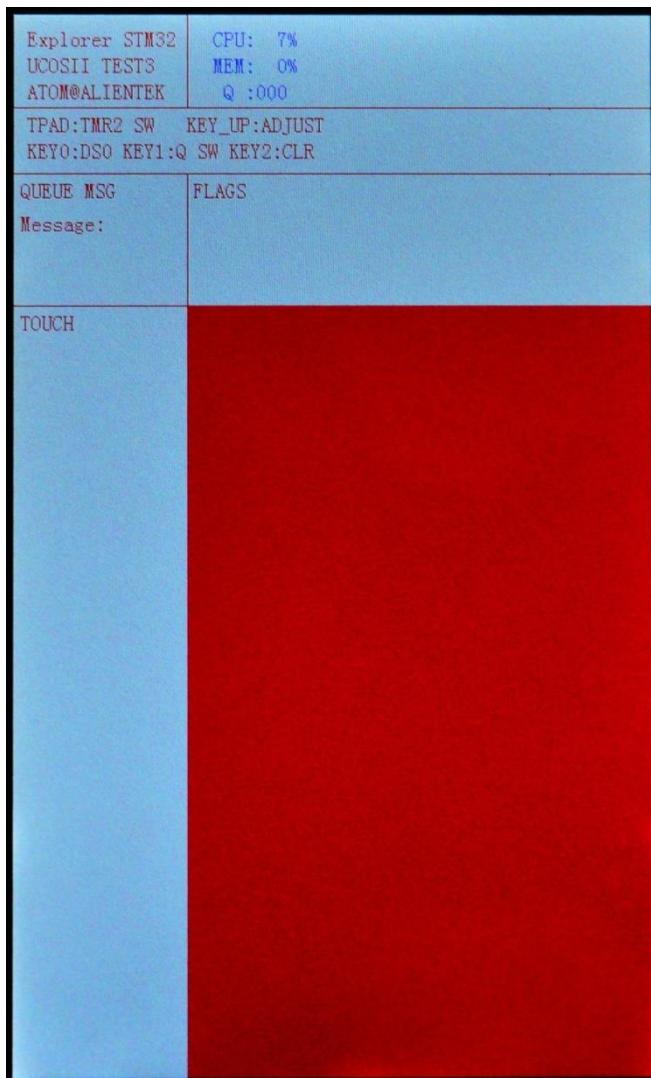


图 63.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率为 10% 左右。比上一章多出很多，这主要是 key_task 里面增加不停的刷屏（tmr2）操作导致的。

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 KEY_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。

第六十四章 探索者 STM32F4 开发板综合实验

前面已经给大家讲了 58 个实例了，本章将设计一个综合实例，作为本手册的最后一个实验，该实验向大家展示了 STM32F4 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)、UCOSII、内存管理、图片解码、视频解码 (AVI)、音频解码 (软解 WAV/MPE/APE/FLAC)、文件系统、USB (主机和从机)、IAP、LWIP (TCP/UDP/Web Server)、陀螺仪 (MPU6050)、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：

- 64.1 探索者 STM32F4 开发板综合实验简介
- 64.2 探索者 STM32F4 开发板综合实验详解

64.1 探索者 STM32F4 开发板综合实验简介

探索者 STM32F4 开发板是 ALIENTEK 的第三款 STM32 开发板（之前有 MiniSTM32 和战舰 STM32 开发板），也是第一款基于 STM32F4 的开发板，其性能比 STM32F1 强不少，该开发板的出现，为大家提供了一个更强大的 STM32 开发板平台。

探索者 STM32F4 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。

探索者 STM32F4 开发板的功能在战舰 STM32 开发板的基础上进行了扩展，功能更强大了。

探索者 STM32F4 开发板综合实验总共有 19 大功能，分别是：电子图书、数码相框、音乐播放、视频播放、时钟、系统设置、FC 游戏机、记事本、运行器、手写画笔、照相机、录音机、USB 连接、网络通信、无线传书、计算器、拨号、应用中心和短信。

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/.wav/.ape/.flac 等 4 种常见音频文件的播放，全部软解码实现。

视频播放，支持.avi 格式（MJPEG 编码）的视频播放（带音频），也是软解码实现。

时钟，支持温度、时间、日期、星期的显示，同时具有指针式时钟显示。

系统设置，整个综合实验的设置。

FC 游戏机，支持绝大部分 NES 游戏(.nes)，支持 USB 手柄/键盘控制，带声音，超 InfoNES。

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≤120K）。

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 (.bmp/.jpg 格式，需摄像头模块支持），并支持成像效果设置。

录音机，支持 wav 文件格式的录音（8~48Khz/16 位立体声录音），支持 AGC 设置。

USB 连接，支持和电脑连接读写 SD 卡/SPI FLASH 的内容。

网络通信，LWIP，支持 10/100M 自适应，支持 DHCP，支持 UDP/TCP/Web Server 测试。

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。

拨号，支持拨打电话（需要 GSM 模块支持）。

应用中心，可扩展 16 个应用程序，我们实现了其中 2 个（红外遥控&陀螺仪），其他预留。

短信，支持短信读取、发送、删除等操作（需要 GSM 模块支持）。

以上，就是综合实验的 19 个功能简介，涉及到的内容包括：GUI（ALIENTEK 编写，非 ucGUI）、UCOSII、内存管理、图片解码、音频解码、视频解码、文件系统、USB(主机&从机)、IAP、LWIP（TCP/UDP/Web Server）、陀螺仪（MPU6050）、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 19 个功能。

64.2 探索者 STM32F4 开发板综合实验详解

要测试探索者 STM32F4 开发板综合实验的全部功能，大家得自备 1 个 SD 卡、1 个 U 盘、1 根网线、一个耳机/喇叭、1 个 GSM 模块和 1 个 ALIENTEK OV2640 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、不能拨号、不能收发短信等。除了这几个，其他功能基本都可以正常运行。

预备知识：

- 1, 系统支持： ALIENTEK 2.8 寸电阻屏、 ALIENTEK 3.5 寸电阻屏和 ALIENTEK 4.3 寸电容屏，自动识别。
- 2, 系统针对不同分辨率的屏幕，不同界面，会采用不同的字体和图标，以达到最佳效果。
- 3, 系统主界面，对于 2.8 寸和 3.5 寸液晶模块，将会有 2 页，通过滑动切换。每页 8 个图标+底部 3 个固定图标，总共 19 个。对于 4.3 寸液晶模块，直接就是 1 页，4.3 寸屏不支持滑动。
- 4, 系统测试有可能需要比较大电流（4.3 屏、网络、外接喇叭）供电，强烈建议使用外部电源供电。
- 5, 系统要用到 USB 通信（接 U 盘/USB 手柄/USB 键盘），请将开发板 P11 端子的 D+ 和 D- 分别连接到 PA12 和 PA11。

有了以上预备知识，我们先来看看探索者 STM32F4 开发板综合实验的启动界面，启动界面如图 64.2.1 所示：

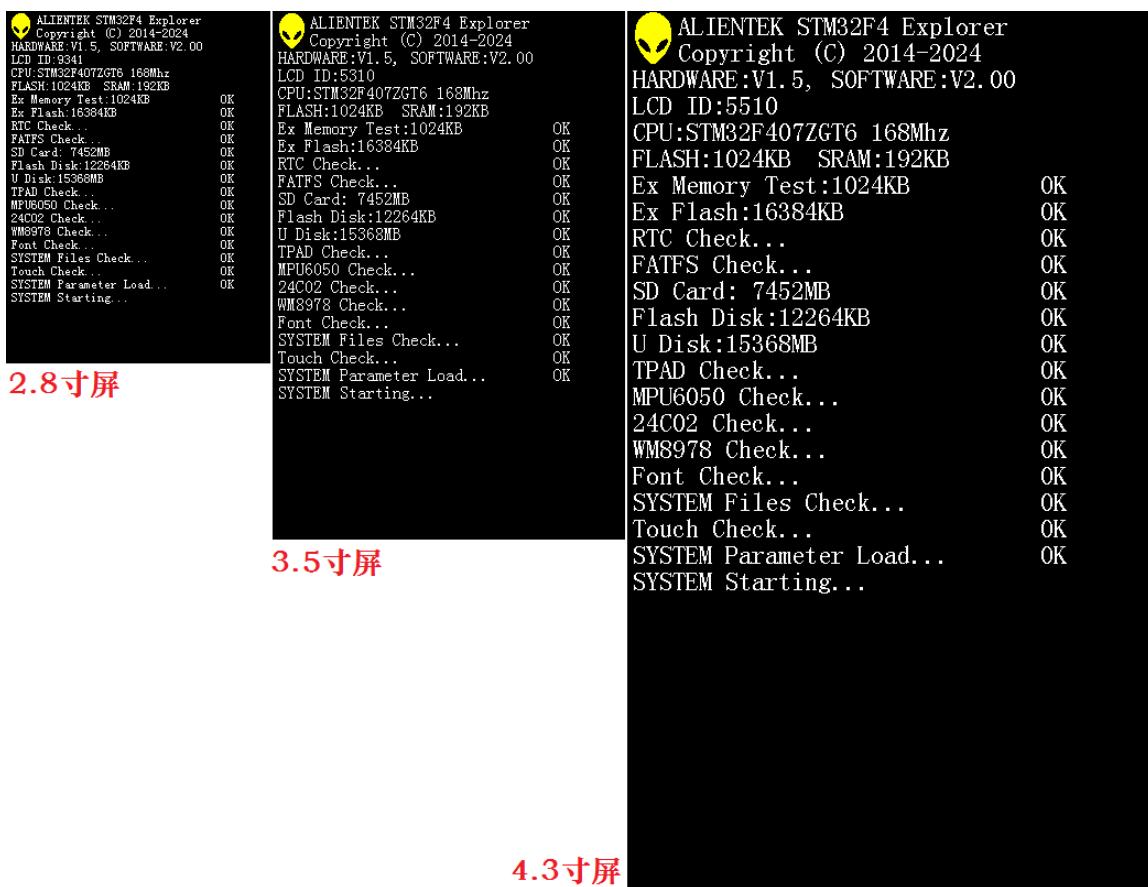


图 64.2.1 综合实验启动界面

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！

图 64.2.1 总共有 3 个截图拼成，分别代表 2.8 寸、3.5 寸和 4.3 寸屏模块，显示内容都一样，但是图标大小和文字大小各不相同。图片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 RTC 和文件系统 (FATFS)，然后显示 SD 卡容量、FLASH Disk 容量（注意 FLASH Disk 就是指 SPI FLASH，因为我们划分了 12M 空间给 FATFS 管理，所以 FLASH Disk 的容量为 12264KB）和 U 盘容量。

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/U 盘，其容量显示 0，并提示 ERROR，不过系统还是会继续启动，因为没有 SD 卡/U 盘系统还是可以启动的（前提是 SPI FLASH (W25Q128) 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(W25Q128)里面的，如这些文件被破坏了，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/U 盘，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 (W25Q128) 里面分割 12M 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**（仅电阻屏支持）。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 在系统启动时，移植按着 KEY2 不放（加载到 FLASH 容量时），可以选择是否擦除所有文件（清空 SPI FLASH），当需要重新更新的时候，建议先用此方法擦除，再更新。
- ⑦ 本系统用到触摸按键 TPAD 做返回（类似手机的 HOME 键，**TPAD 在开发板右下角，白色的骷髅头丝印，该区域是触摸按键区域!! 手指轻轻一摸，即可返回**），所以请确保多功能端口 P12 的 ADC 和 TPAD 用跳线帽短接！
- ⑧ 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）。
- ⑨ 如果插入了 U 盘，且没有插入 SD 卡，那么在执行文件保存的时候，会在 U 盘创建和 SD 卡一样的文件夹（TEXT、RECORDER、PAINT 和 PHOTO 等），然后将文件保存到 U 盘（如果插入 SD 卡，默认就存 SD 卡了！）。

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 64.2.2~64.2.4 所示：



2.8寸主界面1

2.8寸主界面2

图 64.2.2 综合实验系统主界面(2.8 屏版本)



3.5寸主界面1

3.5寸主界面2

图 64.2.3 综合实验系统主界面(3.5 屏版本)



图 64.2.4 综合实验系统主界面(4.3 屏版本)

从上面三张图可以看出，2.8 屏和 3.5 屏主界面有 2 个页面（滑动切换），而 4.3 屏的只有 1 个页面（不支持滑动），总共是 19 个图标。每个图标代表一大功能，主界面顶部具有状态栏，显示 GSM 模块信号质量、运营商、SD 卡状态、U 盘状态、CPU 使用率和时间等信息。

注意：GSM 模块信号质量和运营商，必须是接了 ATK-SIM900A GSM 模块后，才可能正常显示的，否则信号质量显示灰色，运营商显示：无移动网。

ALIENTEK ATK-SIM900A GSM 模块与开发板的连接有两种方式：

1，通过 RS232 串口线连接。将开发板配套的 RS232 串口线，连接 GSM 模块的 RS232 串口和开发板的 COM3 即可完成连接，注意：开发板 P10 端子的 COM3_TX 与 PB11(RX)和 COM3_RX 与 PB10(TX)必须用跳线帽短接!!

2，通过杜邦线连接。将开发板 P10 端子的两个跳线帽拔了，然后用杜邦线将开发板的 PB11(RX)接 GSM 模块的 STXD 脚，将开发板的 PB10(TX)接 GSM 模块的 SRXD 脚。最后共地，开发板上随便找一个 GND 用杜邦线和 GSM 模块的 GND 连接起来。

只有 GSM 模块连接好，SIM 卡正常，且长按 GSM 模块的 PWR_KEY 开机后，开发板才会显示如图 64.2.4 所示的信号质量和运营商，才可以进行拨号和短信功能测试!! 如果不显示信号质量和运营商，请检查是否有 GSM 模块，或者 GSM 模块是否工作正常！

会到主界面，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，探索者 STM32F4 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。

我们可以通过点击任何一个图标，选中，然后再次点击，即可进入该图标的功能。接下来，我们主要以 4.3 屏为例，给大家讲解综合实验。2.8 屏和 3.5 屏操作基本一模一样，下面就不再分别贴图了。

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 就是探索者 STM32F4 开发板上的一个触摸按键，即右下角的白色骷髅头!!

在介绍完系统启动之后，我们开始介绍各个功能。

64.2.1 电子图书

双击主界面的电子图书图标，进入如图 64.2.1.1 所示的文件浏览界面：



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/WIN7上打开我的电脑),可以看到我们有3个盘,磁盘名字分别是:正点原子、ALIENTEK和广州星翼。正点原子是我们SD卡的卷标(即磁盘名字),ALIENTEK是板载SPI FLASH磁盘的卷标,广州星翼是我们插入的一个U盘的卷标。注意:如果没有插入SD卡和U盘,则只会显示ALIENTEK这一个卷标。我们可以选择任何一个磁盘打开,并浏览里面的内容。

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,选中一个文件夹,双击打开得到如图64.2.1.2所示界面:

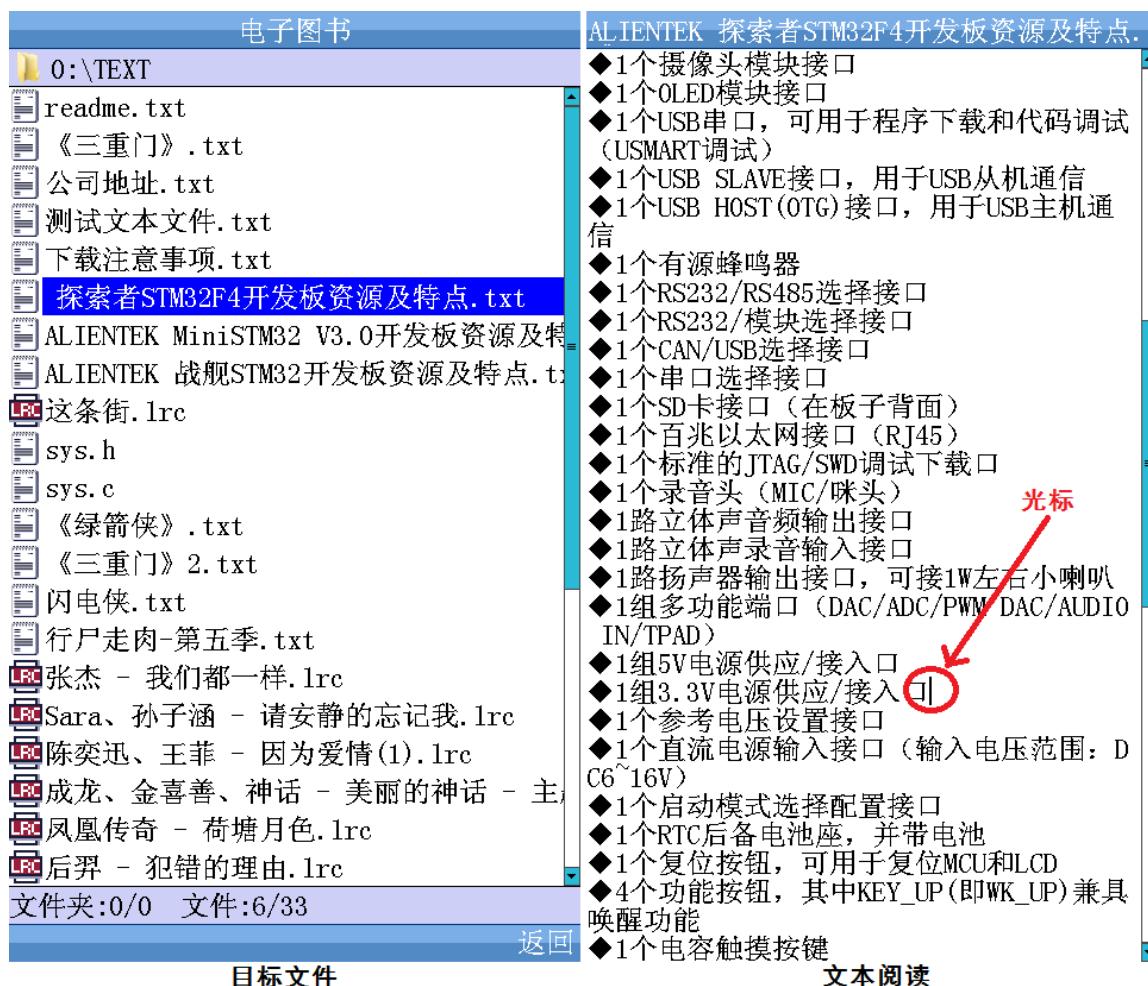


图 64.2.1.2 目标文件和文本阅读

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 txt 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 960KB（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。

64.2.2 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 64.2.1 节差不多，我们找到存放图片的文件夹，如图 64.2.2.1 所示：



图 64.2.2.1 文件浏览和图片播放

左侧是文件浏览的界面,可以看到在 PICTURE 文件夹下总共有 27 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 64.2.2.2 所示:



图 64.2.2.2 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 LCD 分辨率以内的文件（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）区域切换到下一张图片；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，同样，通过按 TPAD 按钮，可以返回文件浏览状态。

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。

64.2.3 音乐播放

探索者 STM32F407 开发板综合实验的音乐播放器性能非常强悍，可作为 HIFI 播放器使用。支持常见的无损音乐(wav/flac/ape)播放，具体性能如下：

wav 文件：支持最高 192K@24bit 播放。

mp3 文件：全码率支持

flac 文件：支持最高 192K@16bit，或者 96K@24bit 播放

ape 文件：最高支持 96K@16bit（LEVEL1 压缩）播放

注意：如果是播放 U 盘的音乐，建议不要播放采样率太高的音频，否则可能导致播放不正常（建议在 48K 及以下比较好）。

双击主界面的音乐播放图标，进入文件浏览界面，这个和 64.2.1 节差不多，只是这里我们浏览的文件变为了.wav/.mp3/.flac/.ape 等音频文件，我们找到存放音频文件的文件夹，如图 64.2.3.1 所示：



图 64.2.3.1 文件浏览和 wav 格式播放

左侧是文件浏览的界面，可以看到在音乐文件 文件夹下总共有 33 个音频文件，包括 wav/mp3/flac/ape 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、采样率、位数、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（注意：ape 格式不支持），点击声音进度条，可以设置音量。上图为正在播放 wav 文件，当然我们还可以播放其他音频格式，如图 64.2.3.2 所示：



图 64.2.3.2 mp3 格式播放和 flac 格式播放

图 64.2.3.2 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 5 段 EQ、3D 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（当然后缀是不同的，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。歌词采用多行显示，中间为当前正在演唱的歌词（粉红色字体显示），上下分别有预览歌词（白色字体显示），如果正在演唱的歌词太长，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。

本音乐播放器支持多种无损音频格式播放，前面介绍了 wav 和 flac。Wav 和 flac 是支持 24bit 播放的，不过 ape 则只支持 16bit 播放。最后，看看 ape 文件的播放，如图 64.2.3.3 所示：



图 64.2.3.3 ape 格式播放

注意，需要外接耳机（插入 PHONE 端子）或者喇叭（接 SPK，P1 端子）才可以欣赏音乐哦!!! 耳机和喇叭需自备。

64.2.4 视频播放

探索者 STM32F407 开发板的综合实验支持视频播放（带声音），软解码 avi 文件，实现视频播放。支持的视频格式为：.avi，视频必须使用 MJPEG 压缩，音频采用线性 PCM 编码（无压缩）。视频分辨率必须小于等于屏幕分辨率，对于综合实验来说：2.8 寸屏，最大支持 240*164 分辨率的视频；3.5 寸屏，最大支持 320*296 分辨率的视频；4.3 寸屏，最大支持 480*550 分辨率的视频。**特别提醒：一般网络下载的视频文件（.avi/.rmvb/.mkv/.mp4 等），本播放器不支持，必须通过软件转换（狸窝全能视频转换器），才可以，详细转换方法，见 50.4 节。**

双击主界面的视频播放图标，进入文件浏览界面，这个和 64.2.1 节差不多，只是这里我们浏览的文件变为了.avi 的视频文件，我们找到存放视频文件的文件夹，如图 64.2.4.1 所示：



图 64.2.4.1 文件浏览和 avi 视频播放

左侧是文件浏览的界面，可以看到在视频文件夹下总共有 21 个视频文件。右侧图片则是我们视频播放器的主界面，该界面显示了当前播放视频的名字、播放进度、播放时长、总时长、音频采样率、视频帧率、视频分辨率、音量、当前文件编号、总文件数等信息。下方的 5 个按键分别是：目录、上一个视频、暂停/播放、下一个视频、返回。点击视频播放进度条，可以直接设置视频播放位置，点击声音进度条，可以设置音量。

视频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 5 段 EQ、3D 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。

我们可以通过按目录按钮，来选择其他视频文件；按返回按键（或 TPAD）则可以返回主

界面。视频播放不支持后台播放，所以一旦退出到文件浏览或者主界面，则停止视频播放。

在图 64.2.4.1 中，右侧图片播放的是 480*272 的视频，帧率为 10 帧/秒。对于小分辨率的视频，帧率可以更快一些，比如 320*240 的可以去到 25 帧，240*160 的可以去到 30 帧，如图 64.2.4.2 所示：

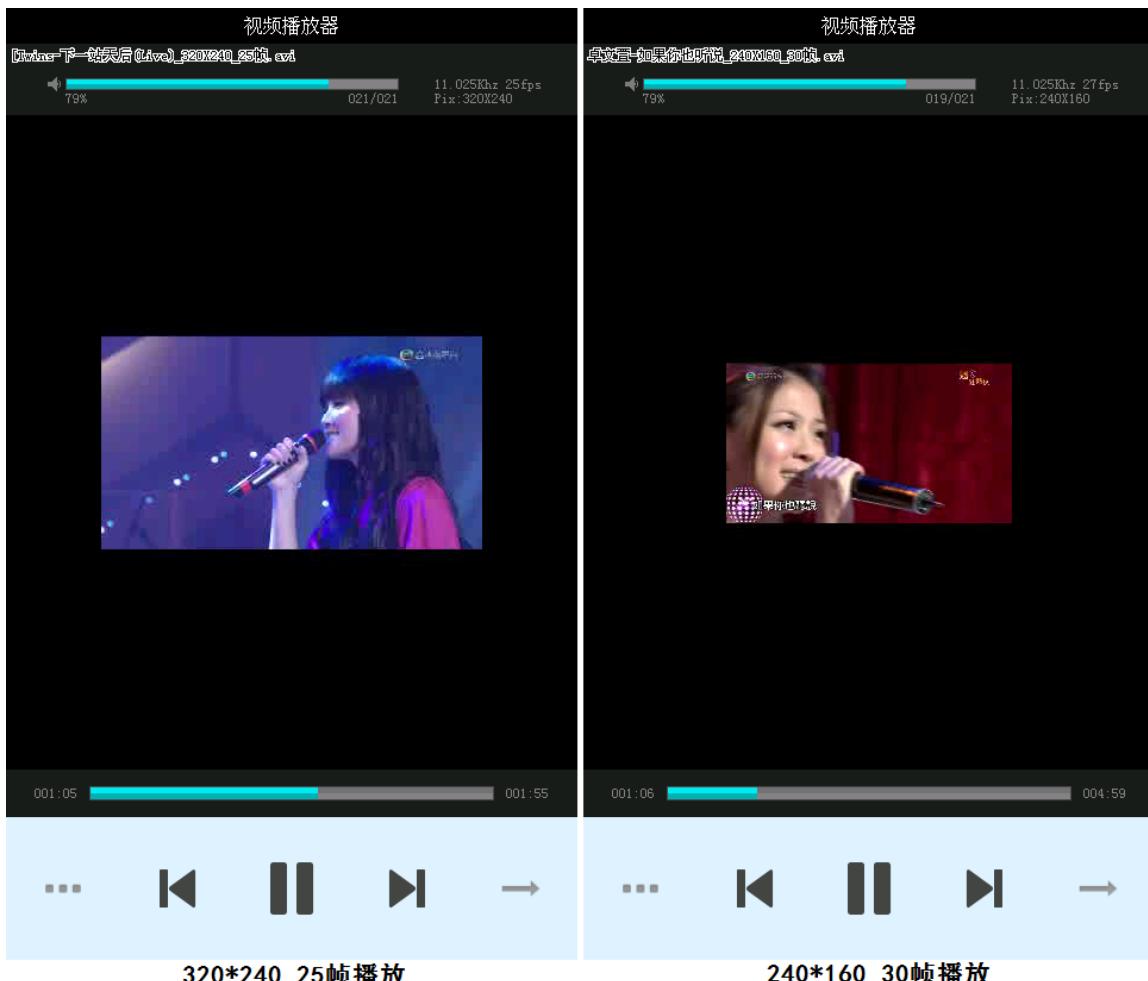


图 64.2.4.2 视频播放

注意，需要外接耳机（插入 PHONE 端子）或者喇叭（接 SPK，P1 端子）才可以听到视频的声音哦!!! 耳机和喇叭需自备。

64.2.5 时钟

双击主界面的时钟图标，进入时钟界面，如图 64.2.5.1 所示：



图 64.2.5.1 时钟界面

图 64.2.5.1 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 DS18B20，启用 MPU6050 的内部温度传感器，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，有一个指针式时钟显示。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。

同样，按 TPAD 可以返回主界面。

64.2.6 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 64.2.6.1 所示：



图 64.2.6.1 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 22 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、音乐播放器模式设置、视频播放器模式设置、WM8978 音量&3D 设置、WM8978 音效 1 设置、WM8978 音效 2 设置、WM8978 音效 3 设置、WM8978 音效 4 设置、WM8978 音效 5 设置、背光设置、屏幕校准、系统文件更新、恢复默认设置、系统信息、系统状态、关于。通过这 22 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 64.2.6.1 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 64.2.6.2 所示：



图 64.2.6.2 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 64.2.6.3 所示：



闹钟开关设置

闹钟铃声设置

图 64.2.6.3 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22号）是周三，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17:31分，而上图我们设置的闹钟时间是17:40，所以时间还不相等，故不闹铃，当时间来到17:40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64.2.6.4所示：



图 64.2.6.4 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（17:45）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 64.2.6.5 所示：

系統設置	SYSTEM SET
1. 時間設置	1. TIME SET
2. 日期設置	2. DATE SET
3. 鬧鐘時間設置	3. ALARM TIME SET
4. 鬧鐘開關設置	4. ALARM ON/OFF SET
5. 鬧鐘鈴聲設置	5. ALARM RING SET
6. 語言設置	6. LANGUAGE SET
7. 數碼相框設置	7. DIGITAL PHOTO FRAME SET
8. 音樂播放器模式設置	8. AUDIO PLAYER MODE SET
9. 視頻播放器模式設置	9. VIDEO PLAYER MODE SET
10. WM8978音量&3D設置	10. WM8978 VOL&3D SET
11. WM8978音效1設置	11. WM8978 EQ1 SET
12. WM8978音效2設置	12. WM8978 EQ2 SET
13. WM8978音效3設置	13. WM8978 EQ3 SET
14. WM8978音效4設置	14. WM8978 EQ4 SET
15. WM8978音效5設置	15. WM8978 EQ5 SET
16. 背光設置	16. BACKLIGHT SET
17. 屏幕校準	17. TOUCH SCREEN ADJUST
18. 系統文件更新	18. SYSTEM FILE UPDATE
19. 恢復默認設置	19. RESTORE DEFAULT SET
20. 系統信息	20. SYSTEM INFORMATION
21. 系統狀態	21. SYSTEM STATUS
22. 關於	22. ABOUT

確定

返回

OK

BACK

繁體中文

English

图 64.2.6.5 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音乐播放器模式设置，如图 64.2.6.6 所示：



图 64.2.6.6 数码相框设置和音乐播放器模式设置

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。而音乐播放器的三个播放模式，则通过右侧的界面进行设置。接下来看看视频播放器模式设置 WM8978 音量&3D 设置，如图 64.2.6.7 所示：



视频播放器模式设置

WM8978音量&3D设置

图 64.2.6.7 视频播放器模式设置和 WM8978 音量&3D 设置

上图中，左侧的界面可以设置视频播放器的模式：全部循环、随机播放、单曲循环，默认是全部循环模式，这个设置和音乐播放器的模式一模一样。右侧是 WM8978 音量&3D 效果设置，可以设置 WM8978 的音量（在此处修改后，将写入 EEPROM 保存，而如果在音频/视频播放的时候修改音量，是不会保存在 EEPROM 的），同时还可以设置 3D 效果，值越大，3D 效果越强，默认是 0，即关闭 3D 效果。

下面我们看看 WM8978 的音效设置和背光设置，如图 64.2.6.8 所示：



WM8978音效1设置

背光设置

图 64.2.6.8 WM8978 音效设置和背光设置

上图中，左侧的界面用于设置 WM8978 的音效 1，WM8978 总共支持 5 段 EQ 设置，分别对应设置界面的：音效 1~音效 5 设置，这里我们仅以音效 1 设置为例，其他的类似。图中的中心频率，即增益对应的频点，默认是 80Hz，可以通过中心频率滚动条设置（80Hz~175Hz），而增益默认设置的是 0，也可以通过滚动条设置（范围：-12dB~+12dB）。大家可以根据自己的喜好，来设置音效以达到最好的音乐享受。**注意：系统设置里面对 WM8978 的设置，都是立即生效的，并且会保存在 EEPROM 里面，当这里设置好之后，其他用到 WM8978 来播放音乐的程序（音乐播放、视频播放和 NES 游戏等），都会共用这些设置。**

右侧的图片用于设置 LCD 背光，背光通过 PWM 控制。当设置为 0 的时候，启动自动(auto)背光控制，其他值则是固定的背光亮度，值越大越亮。自动背光的时候，通过板载的光敏传感器（在摄像头座右侧，LS1）采集环境光强，自动调整背光。

第 17 项，屏幕校准，这里因为我们用的是 4.3 寸电容触摸屏为例讲解的，电容屏不需要校准，所以这个设置对 4.3 屏模块无效。如果是电阻屏，点击该项则可以进入屏幕校准，根据提示完成校准即可。

接下来，我们看看系统文件更新，如图 64.2.6.9 所示：



图 64.2.6.9 系统文件更新

上图中，左侧是双击系统文件更新提示，这里的系统文件是指 SYSTEM 文件夹里面除字库文件外的所有内容。探索者 STM32F4 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 W25Q128）里面，这样，我们所有的系统资源都可以从 W25Q128 里面获得，从而正常启动。

SYSTEM 文件夹目前是包含 151 个文件，总大小为 2.74MB。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 64.2.6.9 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/U 盘的 SYSTEM 文件夹（拷贝自光盘：5，SD 卡根目录文件 里面的 SYSTEM 文件夹），拷贝到 FLASH Disk 里面。这里有个前提，就是你的 SD 卡/U 盘必须有这个 SYSTEM 文件夹！更新时界面如图 64.2.6.9 右侧图片所示，该界面显示了当前更新的文件夹以及文件和进度等信息。

接下来，我们看看恢复默认设置和系统信息，如图 64.2.6.10 所示：

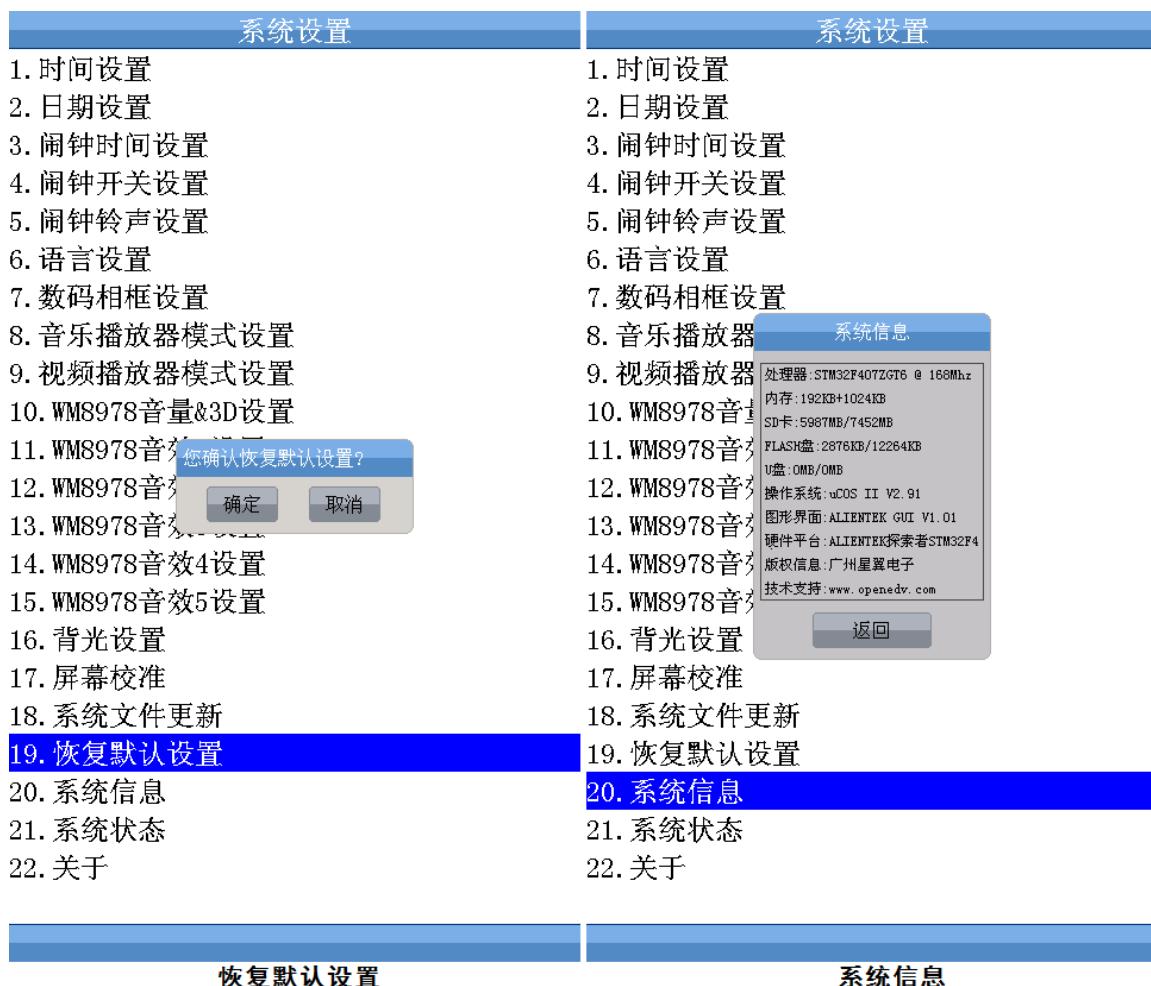


图 64.2.6.10 恢复默认设置和系统信息

上图左侧图片为恢复默认设置功能确认界面，当选择确定后，系统将恢复默认设置，除了RTC时间日期以外的所有设置，都将恢复默认值，方便大家在设置乱以后，恢复正常。

上图右侧图片为系统信息界面，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 64.2.6.11 所示：

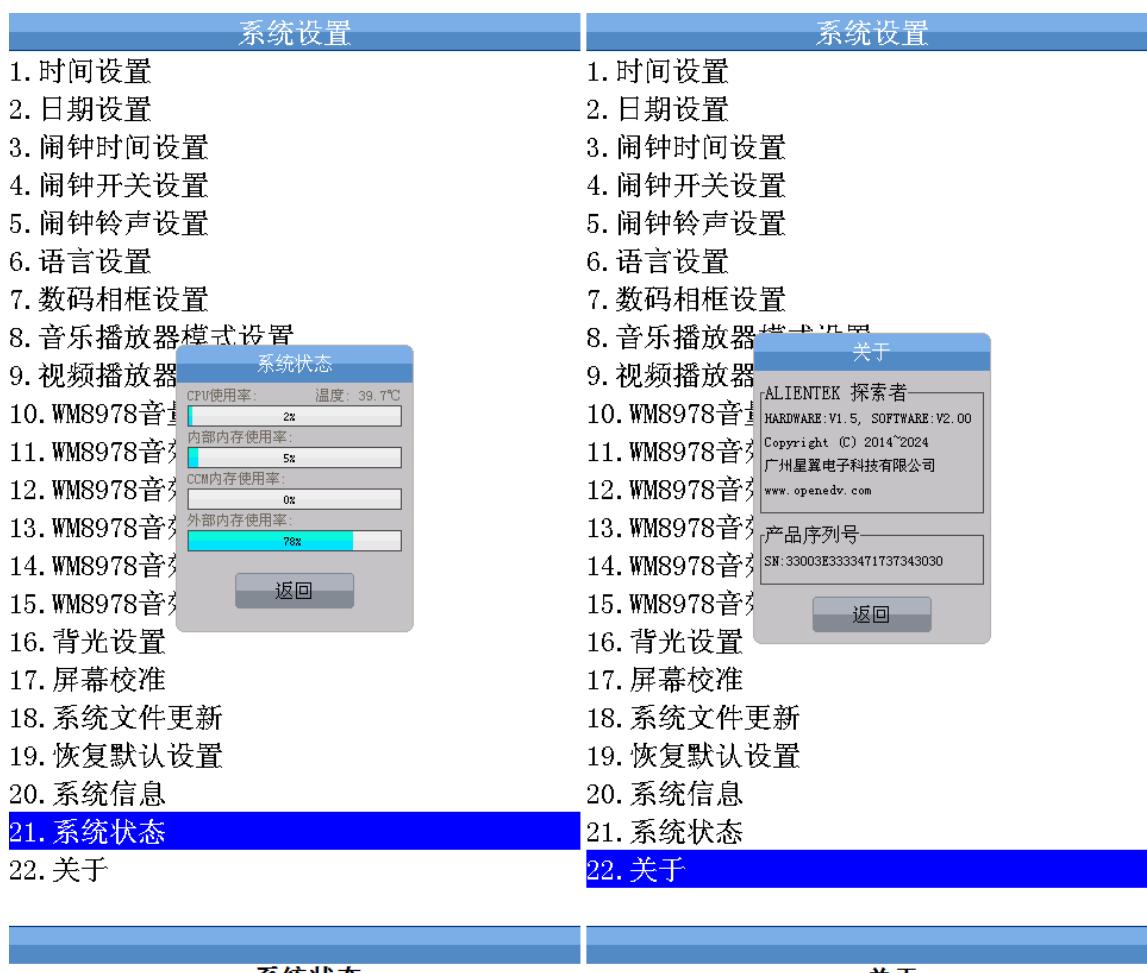


图 64.2.6.11 系统状态和关于界面

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。。

右侧的图片显示了探索者 STM32F4 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。

64.2.7 FC 游戏机

探索者 STM32F4 开发板综合实验移植了一个非常强大的 NES 模拟器，核心部分采用汇编实现，效率极高，支持音频输出，支持 MAP，支持绝大部分 NES 游戏的运行。综合性能超过 infoNES。该模拟器由开源电子网(www.openedv.com)论坛网友:ye781205 编写，然后 ALIENTEK 移植到了探索者 STM32F4 开发板上，即 FC 游戏机。

该 FC 游戏机特点如下：

- 1, 支持 MAP，可运行绝大部分小于 960K 的 NES 游戏。
- 2, 支持 USB 手柄（目前已测试了迪龙 PU201、PU401、PU701、蓝觉 L600、酷孩 USB 无振动手柄等）。
- 3, 支持 USB 键盘输入，可双人游戏。键盘按键与手柄功能对应关系如表 64.2.7.1 所示。
- 4, 支持声音输出（音效远超 infoNES）。
- 5, 支持全速运行（60 帧），在 4.3 寸屏会放大 4 倍处理（480*480 分辨率）。

标准 FC 手柄功能	A	A 连击	B	B 连击	上	下	左	右	Select	Start
对应 USB 键盘按键										
手柄 1 (主)	K	I	J	U	W	S	A	D	空格	回车
手柄 2 (从)	3	6	2	5	↑	↓	←	→		

表 64.2.7.1 USB 键盘按键与手柄功能对应关系

上表中的 3、6、2、5 等数字，是指小键盘的数字按键。另外，需要注意：玩 NES 游戏的时候，USB_SLAVE 不要插电脑，否则无法识别 USB 键盘/USB 手柄!!!

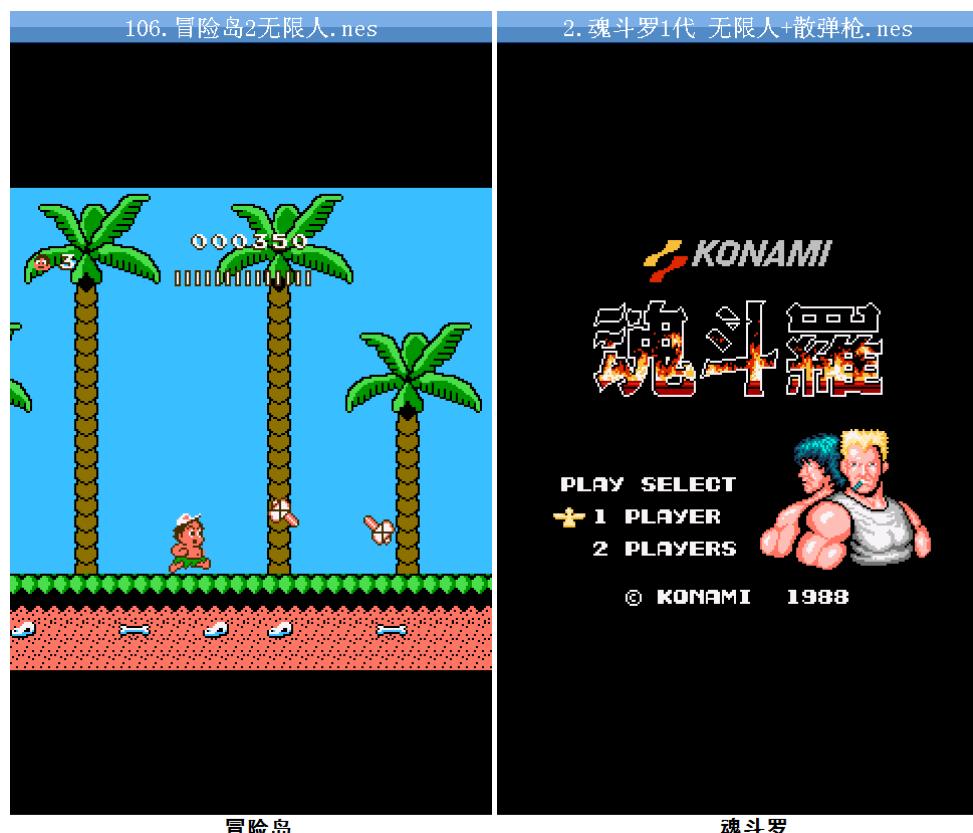
双击主界面的系统设置图标，如果当时插着 U 盘，屏幕将提示：请先拔掉 U 盘。然后，会提示：请插入 USB 手柄/键盘。然后进入 nes 文件浏览界面，如图 64.2.7.1 所示：



图 64.2.7.1 文件浏览和超级玛丽游戏

在检测到 USB 手柄/键盘插入后，屏幕会提示：检测到 USB 手柄/键盘。这个检测是一直开启的，只要插入 USB 手柄/键盘，就会被系统检测到。

上图中，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！右侧的图片为经典的超级玛丽游戏界面，当然还可以玩很多其他经典游戏，如下面的图片所示：



冒险岛

魂斗罗

图 64.2.7.2 冒险岛和魂斗罗



三木童子

双截龙

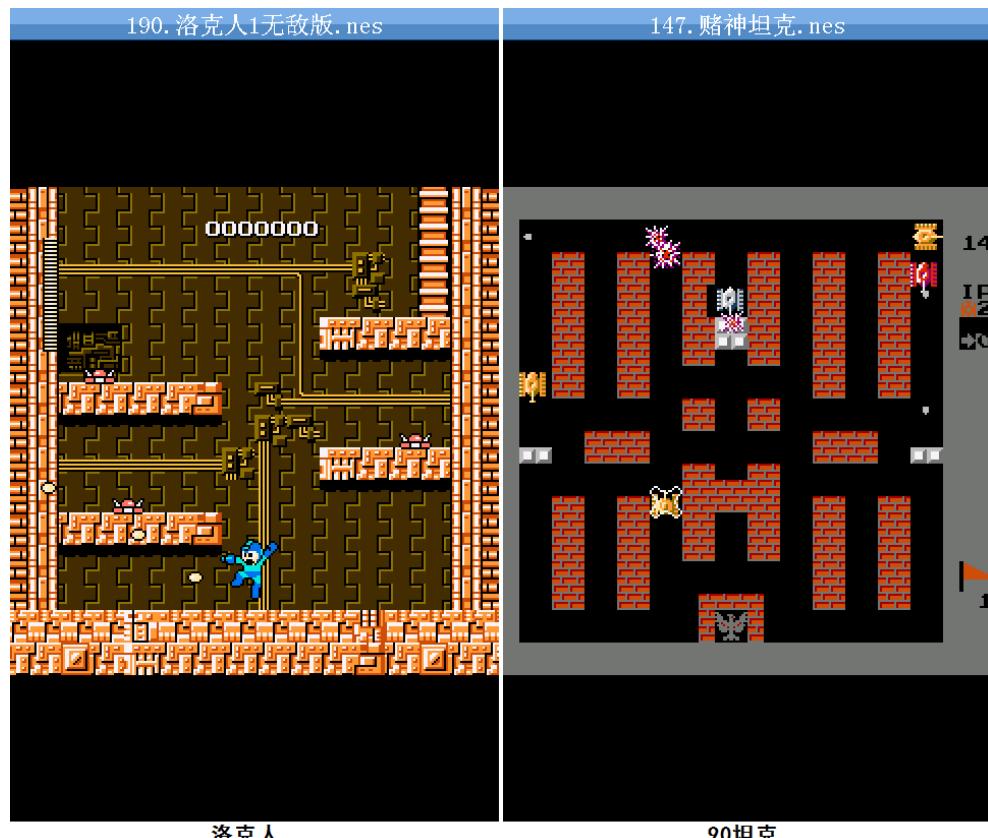
图 64.2.7.3 三木童子和双截龙



炸弹人

SD快打

图 64.2.7.4 炸弹人和 SD 快打



洛克人

90坦克

图 64.2.7.5 洛克人和 90 坦克

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, 在探索者 STM32F4 开发板上, 大家可以回味一下当年的经典了。

64.2.8 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 64.2.8.1 所示:



图 64.2.8.1 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, 如图 64.2.8.2 和图 64.2.8.3 所示:



图 64.2.8.2 中文输入和标点符号输入



图 64.2.8.3 英文输入和手写识别输入

其中，中文输入就是我们前面 T9 拼音输入法实验的具体运用，而手写识别的输入界面，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 64.2.8.4 所示：

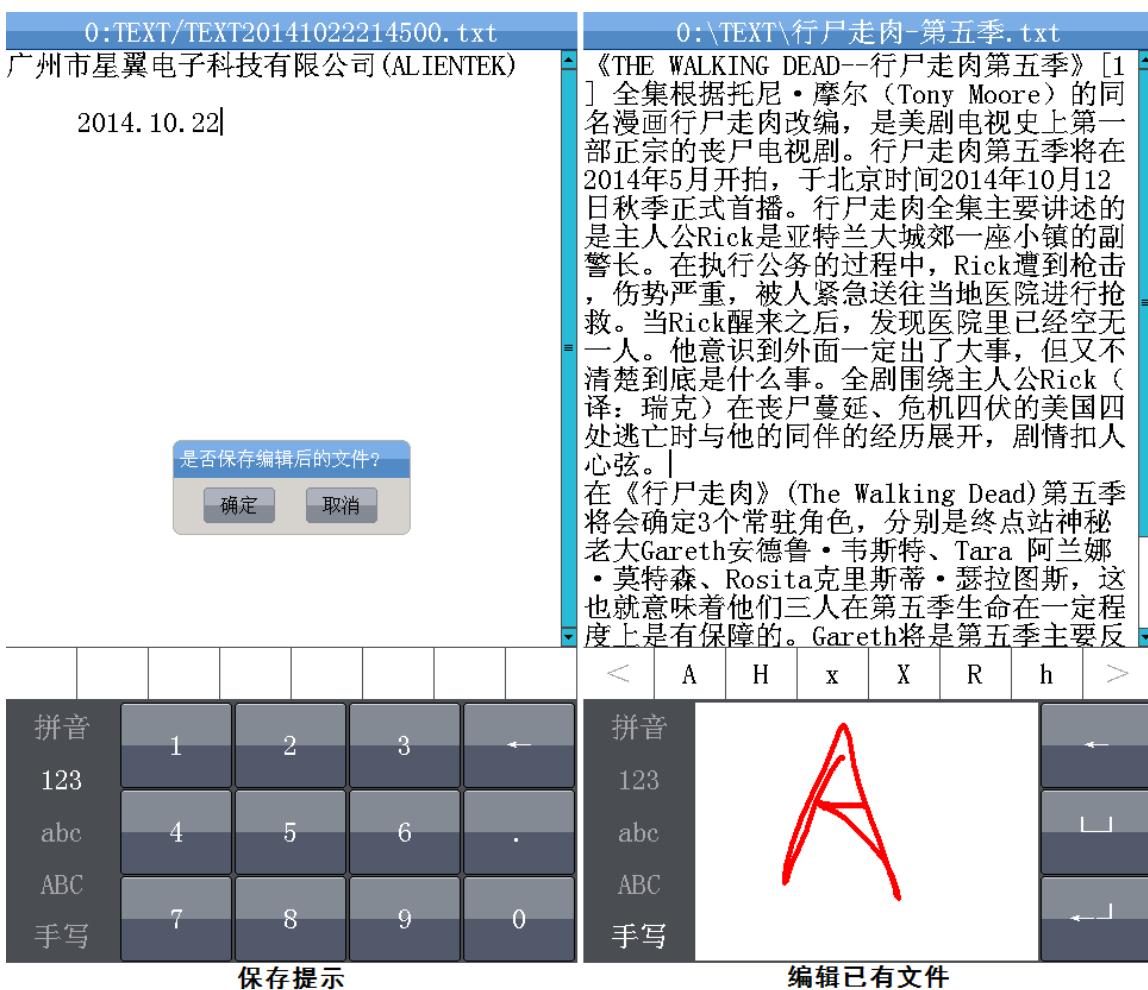


图 64.2.8.4 保存提示和编辑已有文件

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/U 盘根目录的 TEXT 文件夹里面。右侧图片为打开已有文件进行编辑的界面，这样我们就可以在探索者 STM32F4 开发板上编辑.txt/.h/.c/.lrc 文件了。

64.2.9 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 64.2.9.1 所示：

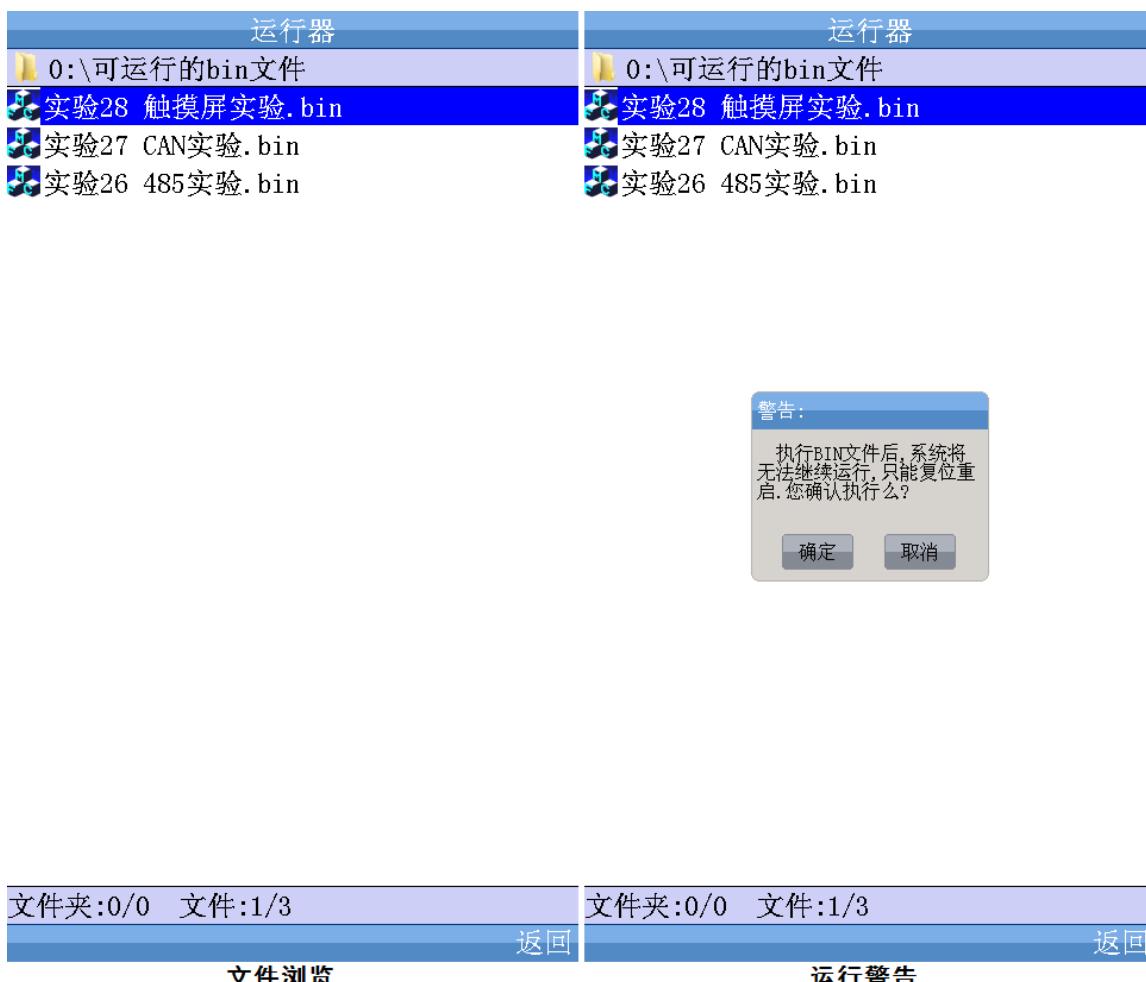


图 64.2.9.1 文件浏览和运行警告

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 3 个，这些全部来自我们的标准例程对应实验。本运行器支持 120K 字节以内的程序运行（FLASH+SRAM 总共不超过 120K），很多例程都可以生成 SRAM 版本的 bin 文件，在运行器里面运行，这里我们仅提供了 3 个实验的.bin 文件以供大家测试。**SRAM 版本.bin 文件的生成办法，请参考串口 IAP 实验这个章节，里面有详细介绍。**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。

64.2.10 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 64.2.10.1 所示：



图 64.2.10.1 模式选择和新建画笔

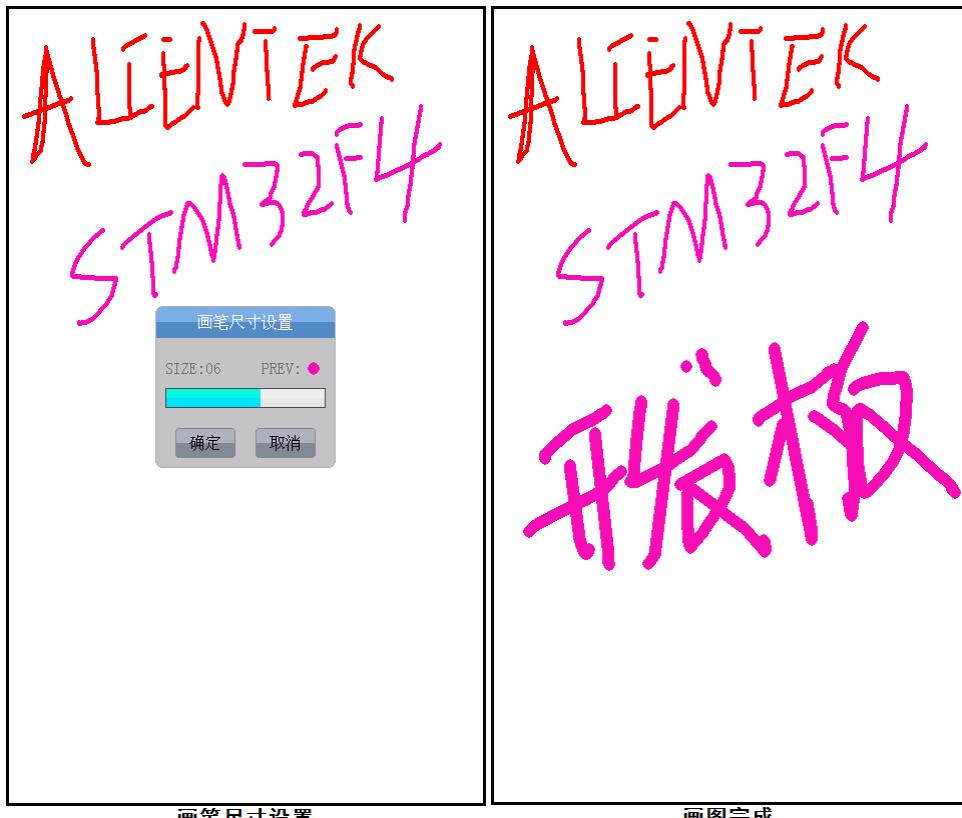
上图中，左侧图片为我们双击手写画笔后，弹出的模式选择界面，我们可以选择新建画笔，建立一个新的文件；也可以选择打开一个已有的位图进行编辑。右侧的图片为我们新建画笔后输入的内容，默认画笔为最小尺寸，颜色为红色。画笔的颜色和尺寸是可以设置的，按 KEY_UP 按键，则弹出画笔设置对话框，然后，可以对画笔颜色和画笔尺寸进行设置。如图 64.2.10.2 和图 64.2.10.3 所示：



画笔设置

画笔颜色设置

图 64.2.10.2 画笔设置和画笔颜色设置



画笔尺寸设置

画图完成

图 64.2.10.3 画笔尺寸设置和完成后的画图

图 64.2.10.2 中，左侧的图片为按 KEY_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，左侧的正方形区域为预览区。

图 64.2.10.3 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 64.2.10.4 所示：



图 64.2.10.4 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，则新的画图文件将会被保存在 SD 卡/U 盘的 PAINT 文件夹里面，命名方式是以当前系统的时间命名的，如 PAINT20141023113650.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。

64.2.11 照相机

本照相机支持 ALIENTEK OV2640 这款 200W 像素的 CMOS 摄像头模块，本照相机的特点有：

- 1, 支持 BMP 拍照（拍下的 bmp 分辨率为 LCD 分辨率），按 KEY2 拍 BMP 照片。
- 2, 支持 JPG 拍照（默认是 1600*1200 像素(UXGA)，可按 KEY_UP 设置拍照分辨率），按

KEY0 拍 JPG 照片。

3, 屏幕显示可以是全景(缩放)或者 1:1 显示(无缩放), 默认是全景, 通过 KEY1 切换。

4, 支持各种参数设置, 包括: 场景、特效、曝光、亮度、色度和对比度等。

双击主界面的照相机图标, 首先初始化 OV2640 摄像头模块, 如图 64.2.11.1 所示:

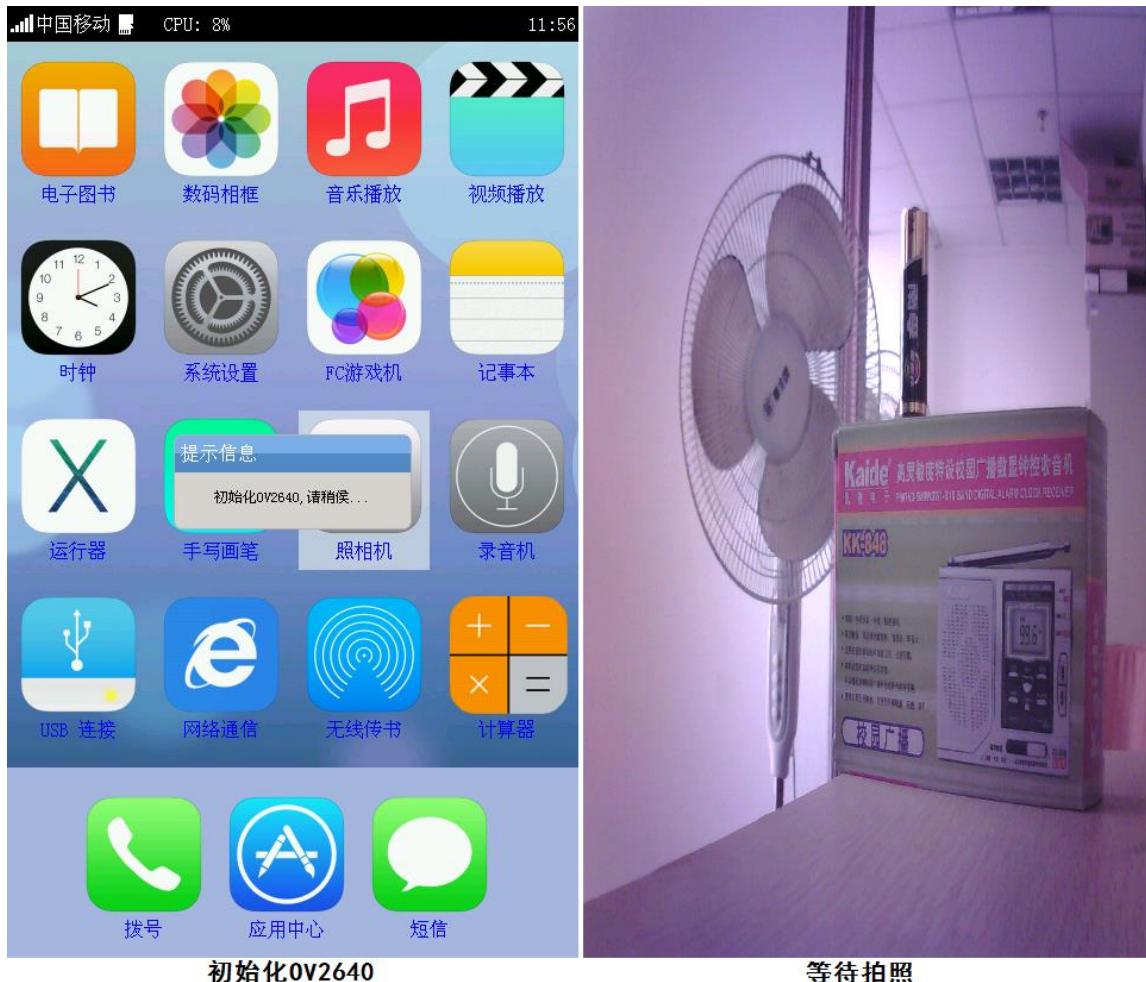


图 64.2.11.1 初始化 OV2640 和等待拍照

在初始化 OV2640 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 64.2.11.2~64.2.11.5 所示:



图 64.2.11.2 相机设置和优场景设置

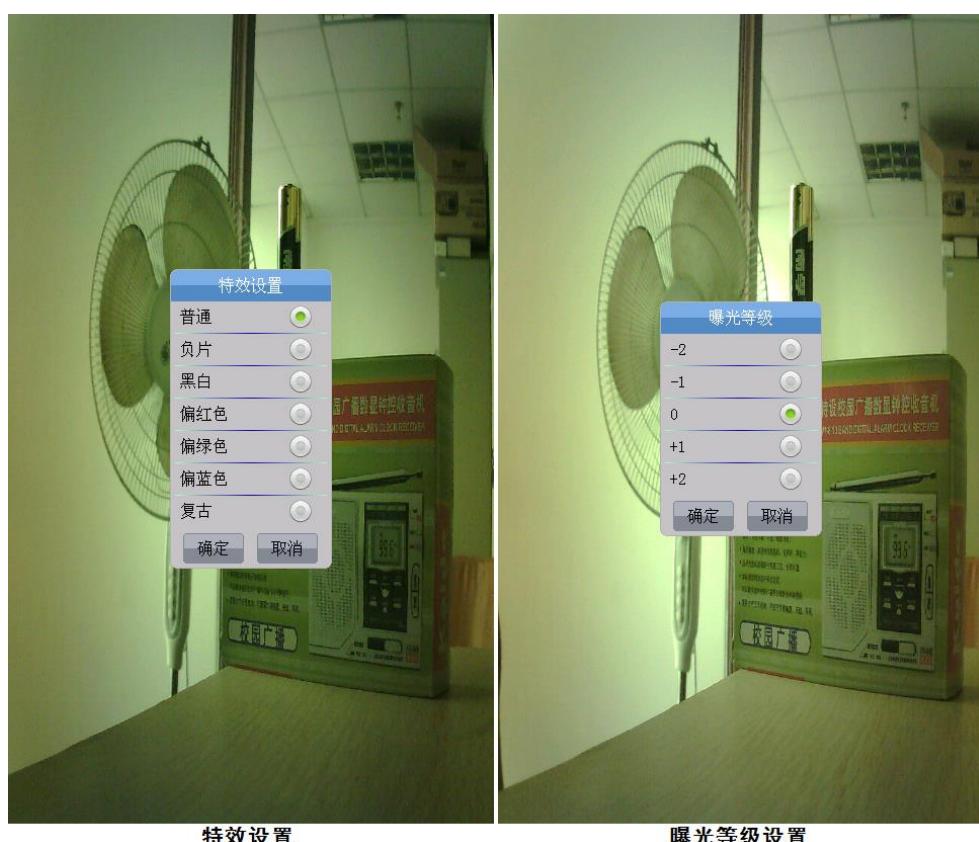


图 64.2.11.3 特效设置和曝光等级设置

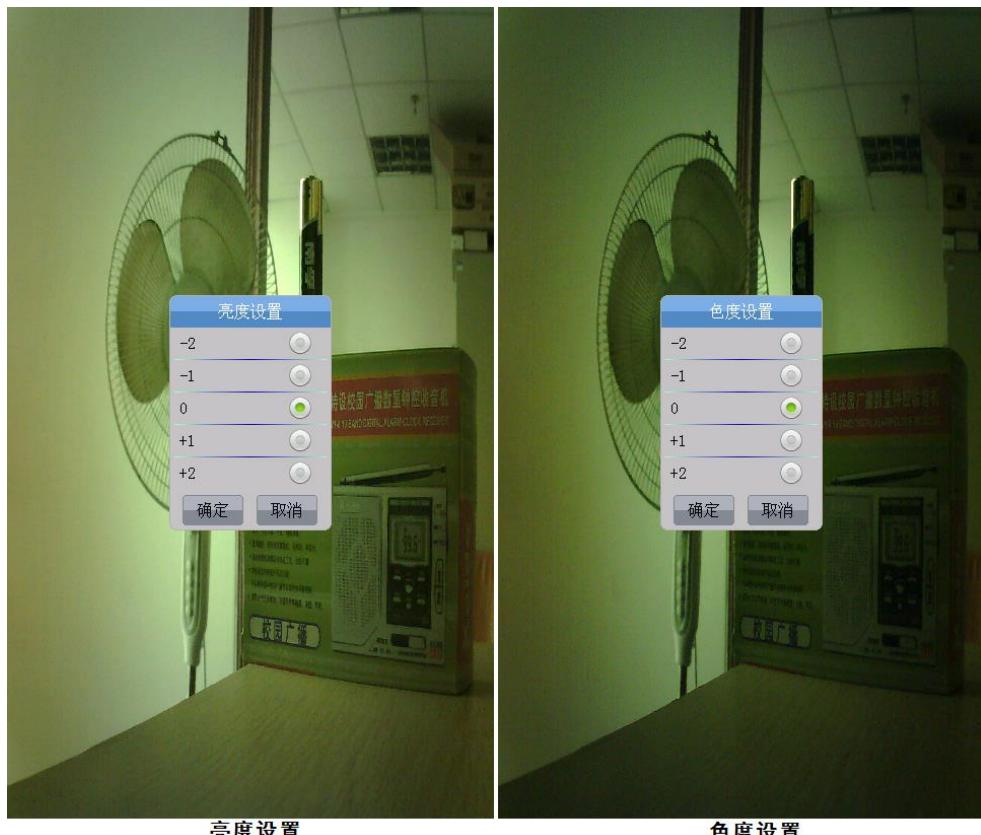


图 64.2.11.4 亮度设置和色度设置

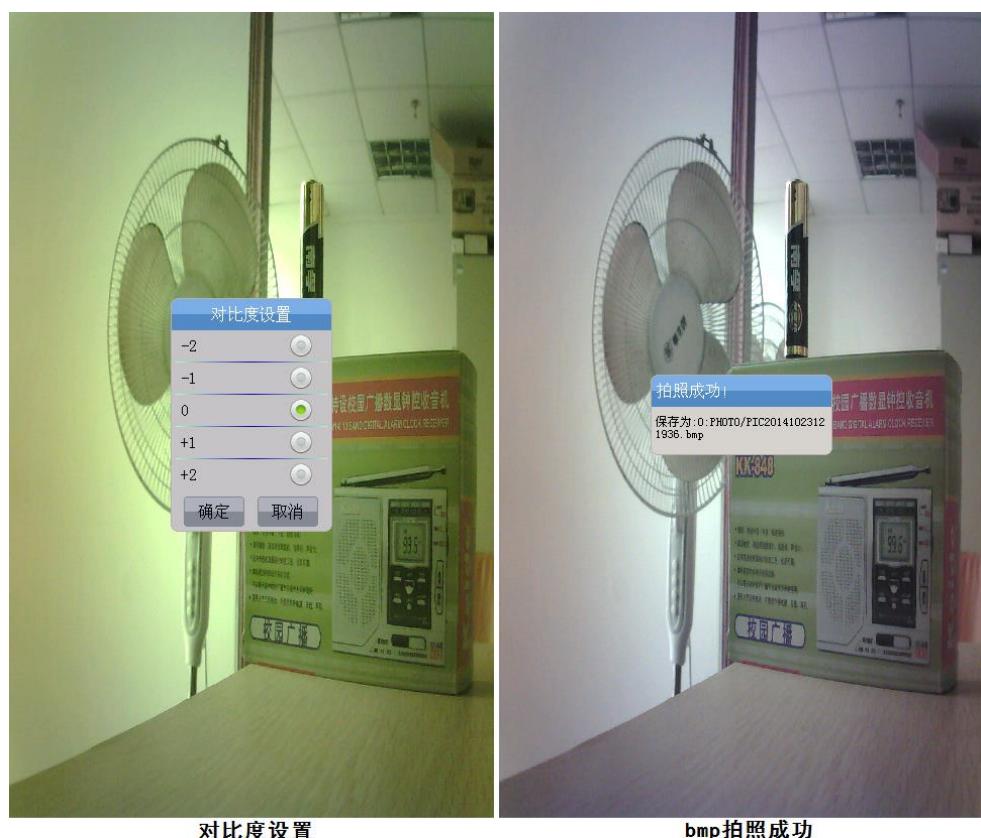


图 64.2.11.5 对比度设置和拍照

以上设置，和手机类似，这里就不一一和大家解释了。我们按 KEY0 按键，即可实现 JPG 拍照，JPG 照片的尺寸，默认是 1600*1200 分辨率，可通过 KEY_UP 设置其他分辨率。按 KEY2 按键即可实现 BMP 拍照，BMP 照片的尺寸就是 LCD 的分辨率。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/U 盘的 PHOTO 文件夹。如果你没有插入 SD 卡/U 盘，拍照时会提示“创建文件失败，请检查！”的提示信息。

按 KEY1 按键，可以实现 1:1 显示（显示区域小，但是图像无压缩，不变形），或者全尺寸显示（显示整个摄像头拍照区域，但是图像有压缩，会变形）。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。

最后，看看本程序拍到的 JPG 照片样张，如图 64.2.11.6 所示：



图 64.2.11.6 JPG 照片样图(UXGA)

64.2.12 录音机

ALIENTEK 探索者 STM32F4 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/U 盘。录音文件为 WAV 文件，格式为：立体声（但是左声道数据=右声道数据）、16 位、8Khz~48Khz 采样率可设置。

双击主界面的录音机图标，进入录音机主界面，如图 64.2.12.1 左侧图片所示，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采样率和 MIC（咪头，这里称之为麦克风）增益，通过点击左下角的选项按钮，弹出录音设置对话框，可以设置采样率和 MIC 增益，如图 64.2.12.1 右侧图片所示。

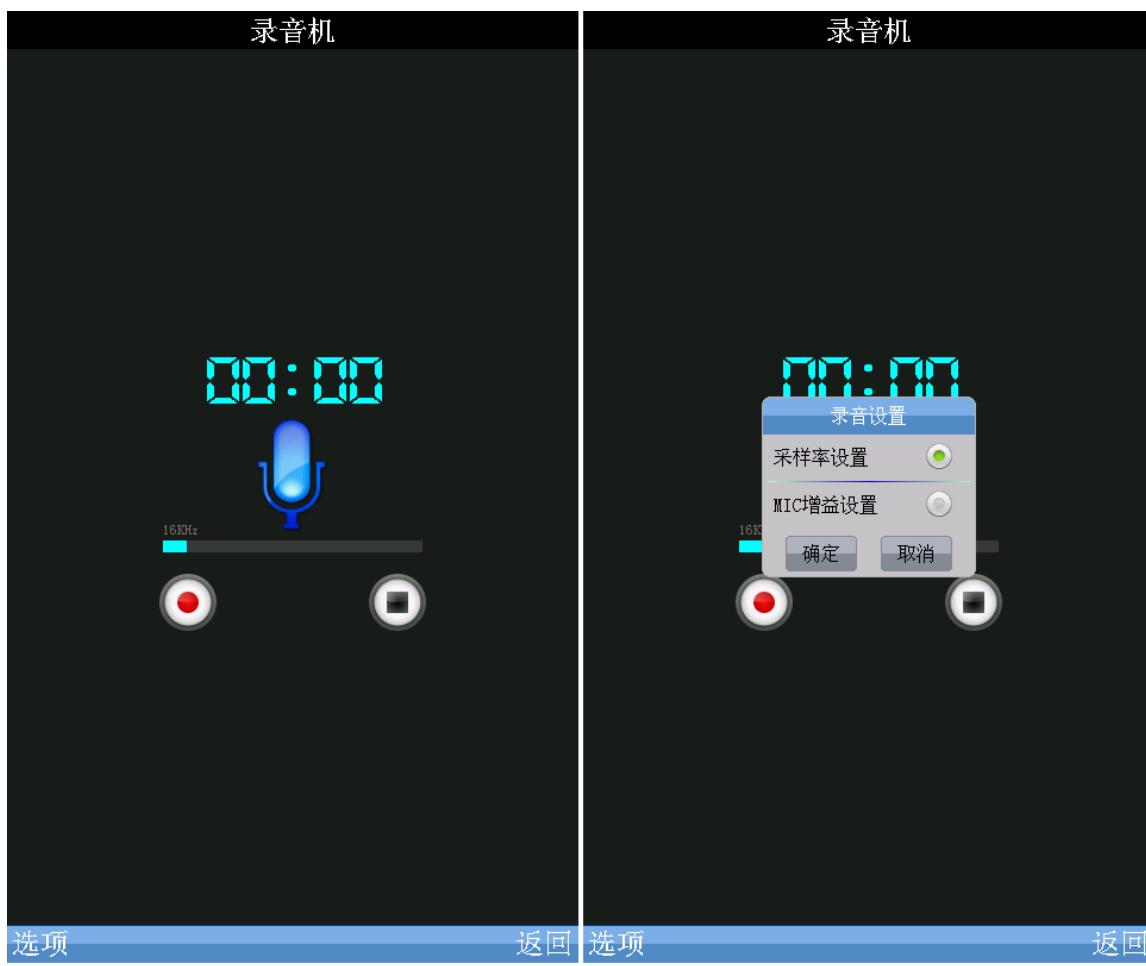


图 64.2.12.1 录音机主界面和麦克风增益设置

采样率设置，可以设置录音时的采样率，支持：8Khz、16Khz、32Khz、44.1Khz 和 48Khz 等几种采样率。默认是 16Khz。注意：采样率必须在没有开始录音之前进行设置才行，否则会提示：请先停止录音。

MIC 增益设置，支持从-12dB~32.25dB 的设置范围，以 0.75dB 步进。默认是 22.5dB 增益，大家可以根据自己需要设置合适的 MIC 增益，以达到最好效果。注意：如果外接了喇叭或者音箱，当增益太大的时候，会产生啸叫，这个时候，请将增益设置小一点。

采样率设置和 MIC 增益设置，如图 64.2.12.2 所示：



图 64.2.12.2 采样率设置和麦克风增益设置

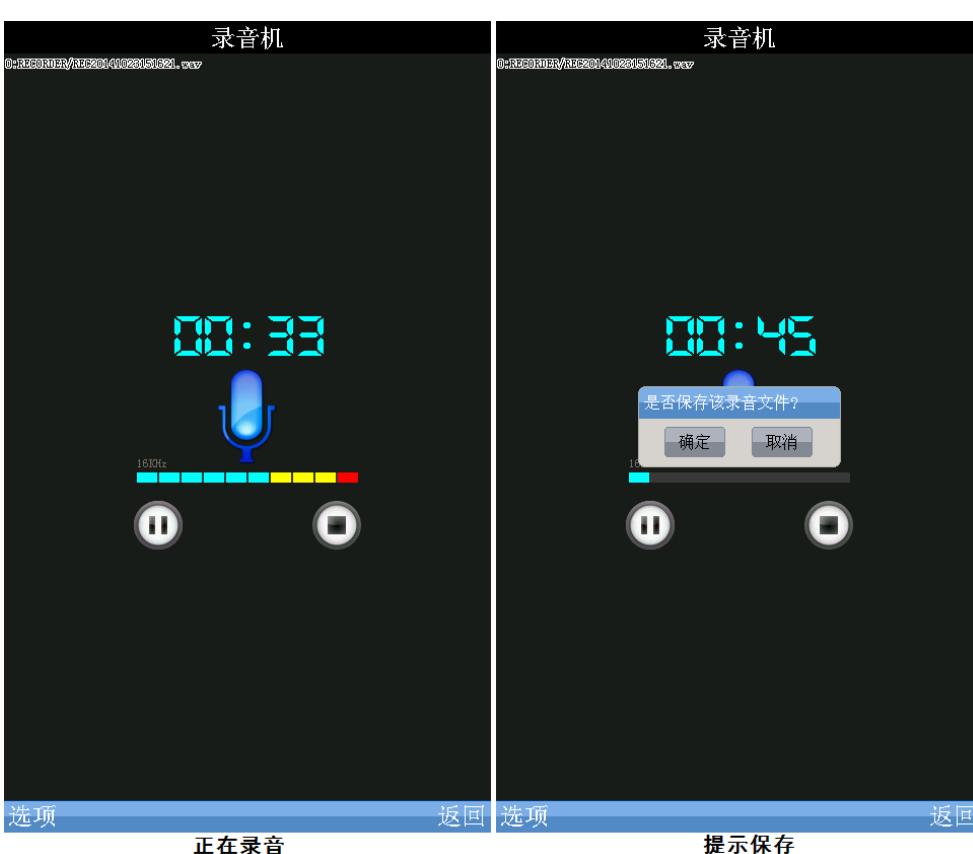


图 64.2.12.3 录音进行中和提示保存

在图 64.2.12.3 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 SD 卡/U 盘根目录的 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。

64.2.13 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 64.2.13.1 所示：

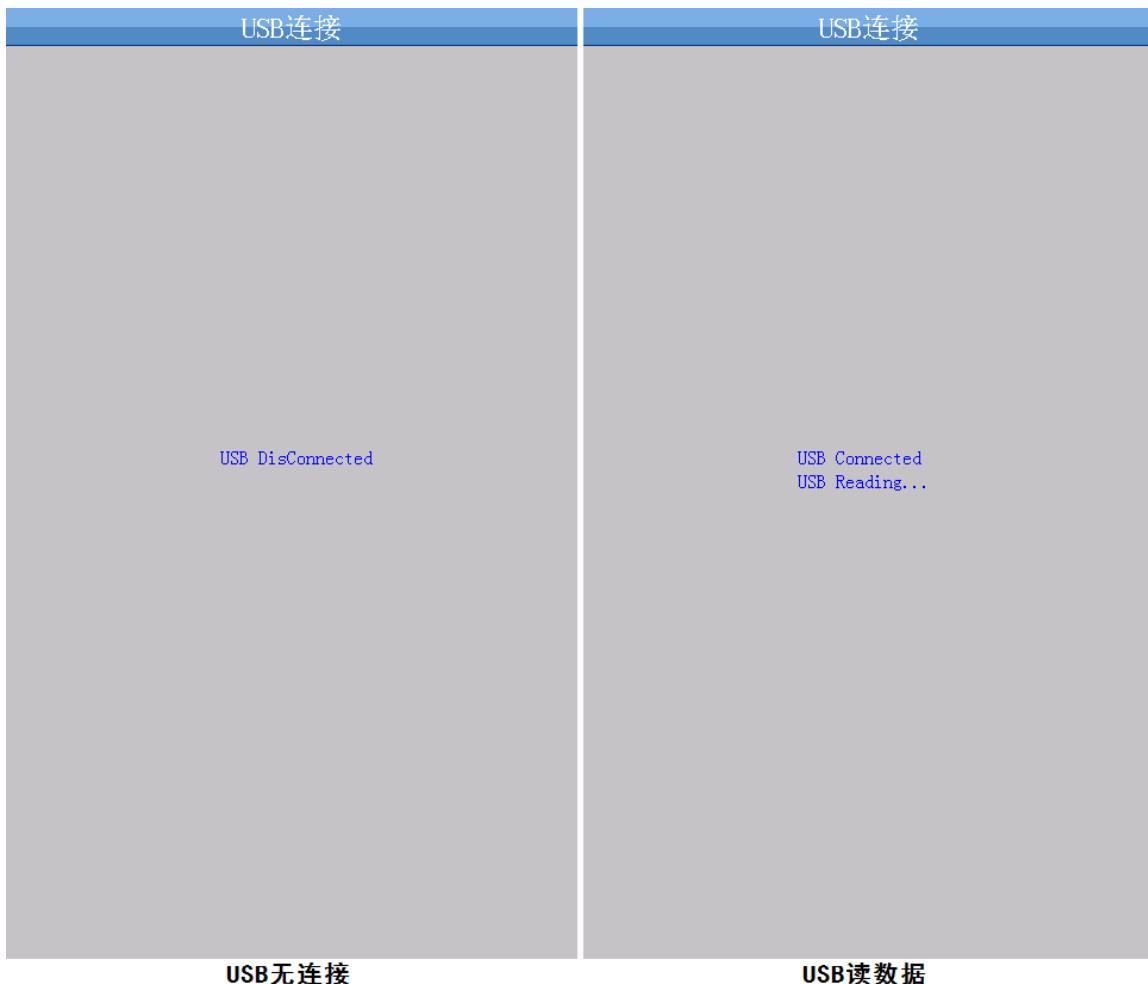


图 64.2.13.1 USB 无连接和 USB 读数据

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：此时 USB_HOST 不能插任何 USB 设备！然后，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，并自动安装驱动（如果是第一次连接的话），如图 64.2.13.2 所示：

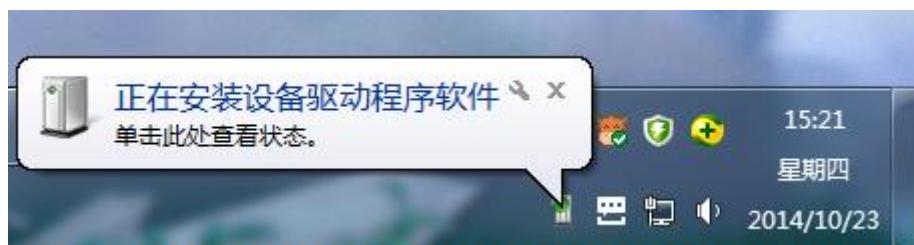


图 64.2.13.2 电脑发现新硬件

此时，我们打开我的电脑，即可找到可移动磁盘，如果有 SD 卡插入，那么会显示 2 个磁盘：ALIENTEK 磁盘和 SD 卡磁盘。如果没有 SD 卡插入，则只显示 ALIENTEK 磁盘。这里的 ALIENTEK 磁盘即开发板板载的 SPI 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 W25Q128）。

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。

64.2.14 网络通信

探索者 STM32F4 开发板板载了一个 10M/100M 自适应以太网接口，可以实现网络通信。本系统的网络通信，就是给大家演示开发板的网络通信功能。

本系统自带的网络通信具有如下特点：

- 1, 使用 LWIP 作为 TCP/IP 协议栈。
- 2, 支持 DHCP，当 DHCP 失败时，使用静态 IP（静态 IP 地址为：192.168.1.30）。
- 3, 自适应网线（支持交叉和直连网线）。
- 4, 支持 Web Server 测试，可通过浏览器，控制开发板的 DS1 和蜂鸣器等。
- 5, 支持 TCP Client、TCP Server 和 UDP 等测试。

特别注意：本测试，必须在开发板网口接入网线，并连接正常后，才可以进行测试。也就是必须用网线连接开发板和电脑/路由器，才可以进行测试，否则，系统会提示网卡初始化失败，从而退出测试!!!

双击主界面的网络通信图标，开始网卡初始化，在网卡初始化成功后，开始 DHCP 获取 IP 地址，在 DHCP 成功后，进入网络通信主界面，如图 64.2.14.1 所示：



图 64.2.14.1 初始化网卡和网络通信主界面

上图中，左侧图片显示正在初始化网卡，此时，我们必须在开发板的网口插入网线，并连接电脑或者路由器，才可以初始化网卡成功。初始化网卡成功后，则开始 DHCP 获取 IP 地址：

1，开发板连接到路由器，此种方式，DHCP 一般可以成功获取 IP 地址。

2，如果是直接连接电脑，那么 DHCP 肯定失败，最终会使用静态 IP 地址：192.168.1.30，此时需要设置电脑 IP 地址为 192.168.1.XXX，其中 XXX 可以由用户自己随意设置（但是不能是 1 和 30）。

这里我们以连接路由器为例，DHCP 获取成功后，进入网络通信主界面，如图 64.2.14.1 右侧图片所示，该界面显示了开发板网卡的详细设置和参数，包括：本机 MAC 地址、本机 IP 地址（DHCP 获取）、网关和网速等。

能进入网络通信主界面，说明开发板与路由器的连接已经正常了，可以在电脑端 ping 屏幕显示的 IP 地址，即可查看网络是否连接正常，如图 64.2.14.2 所示：



```

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 © 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>ping 192.168.1.119

正在 Ping 192.168.1.119 具有 32 字节的数据:
来自 192.168.1.119 的回复: 字节=32 时间<1ms TTL=255

192.168.1.119 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间<以毫秒为单位>:
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms

C:\Users\Administrator>

```

图 64.2.14.2 ping 192.168.1.119 成功

从上图可以看出, ping 开发板 IP 地址是成功的, 说明网络连接正常了。然后在浏览器输入开发板的 IP 地址, 即可登录 Web Server 的主界面, 如图 64.2.14.3 所示:



图 64.2.14.3 Web Server 主界面

在上图所示的 Web Server 主界面, 可以实现对开发板 LED 和蜂鸣器的控制, 以及读取 ADC1 通道 5 的值、温度传感器值、RTC 时间和日期等。**特别注意: ADC1_CH5 所在通道刚好是 TPAD**

的输入通道，所以，本测试读取 ADC1_CH5 的电压值，是不正确的，因为 IO 被 TPAD 占用了!!! 如需要正确读取，请参考网络通信实验（第六十章）。

点击图 64.2.14.1 右侧图片所示的开始测试，即可进入：TCP Server、TCP Client 和 UDP 的测试界面，如图 64.2.14.4 所示：



图 64.2.14.4 TCP&UDP 测试界面和协议选择

从图 64.2.14.4 左侧图片可以看出：最顶部，显示了 TCP Server 的本机 IP 地址和口号，其中 IP 地址是白色，表示不可以设置；口号是绿色，表示可以设置，设办法：触摸点击该区域，看到光标闪烁后，即可输入数字进行设置。随后，显示 TX, RX, 和协议等三个信息，分别代表发送接收的数据量，和当前所使用的协议类型。

然后就是一个接收区和发送区，分别用于显示接收到的数据，和发送的数据。这个同电脑端的网络调试助手一样。右边的协议选择按钮，可以选择不同协议（如图 64.2.14.4 右侧图片所示），该选择按钮只有在连接断开的时候，才有效。连接按钮，用于启动连接（TCP Server/TCP Client 和 UDP 等），连接结果会有对话框提示（成功或者失败）。清除接收按钮，则可以清除接收区的所有数据，同时清除 RX 和 TX 计数器。发送按钮，则用于发送数据，没按一次，发送区的数据就发送一次。

进入 TCP&UDP 测试后，协议默认选择的是 TCP Server，可以通过协议选择按钮，选择不同协议（TCP Server、TCP Client 和 UDP），如上图右侧图片所示。

我们使用默认的 TCP Server 端口号（8088），然后点击链接，提示：连接成功后，即可开启开发板的 TCP Server 服务，然后在电脑端，打开网络调试助手，设置正确的网络参数后（如

不懂设置方法, 请参考第六十章 网络通信实验对应测试部分, 下同!!), 即可连接上开发板的 TCP Server, 并互相通信, 如图 64.2.14.5 所示:



图 64.2.14.5 TCP Server 测试和 TCP Client 参数设置

上图中, 左侧图片为 TCP Server 测试界面, 可以看到, 我们收到来自电脑的数据, 接收数据时, 首先会在接收区提示收到的数据来自何处(即电脑端的 IP 地址, 本例可以看出, 电脑 IP 地址为: 192.168.1.114), 随后才是电脑端发送过来的数据。同时, 我们也发送了一些数据给电脑(电脑端网络调试助手可以看到, 这里就不截图出来了)。此时, 协议选择按钮变为了灰色, 处于无效状态。只有在断开连接后, 才可以选择新的协议。

TCP Client 的测试如图 64.2.14.5 中右侧图片所示, 我们选择了 TCP Client 协议之后, 顶部 IP 变为了目标 IP, 也就是 TCP Client 要连接的 IP 地址, 我们可以根据自己需要设置对应的 IP 地址和端口号。这里, 我们设置为: 192.168.1.114, 端口为: 8087。注意: 这里的 IP 和端口号, 要根据自己电脑的实际情况修改。随后需要在电脑端开启网络调试助手, 并设置正确的网络参数(主要是端口号和 IP 地址), 然后开启 TCP Server 服务。

TCP Client 测试必须等电脑端 TCP Server 服务开启后, 我们才可以在开发板点击连接, 并连接成功, 否则肯定是连接失败的。连接成功后, TCP Client 的测试如图 64.2.14.6 所示:



图 64.2.14.6 TCP Client 测试和 UDP 测试

上图中，左侧为 TCP Client 测试界面，右侧为 UDP 测试界面，这两个测试界面和 TCP Server 差不多，请参考前面的介绍。

UDP 的测试，同 TCP Client 基本一样，也是线设置目标 IP 和端口号，然后按连接，进行测试。UDP 测试的时候，一般需要开发板先发送一次数据给电脑端的网络调试助手，随后才可以实现数据互发。UDP 测试界面如图 64.2.14.6 右侧图片所示。注意：UDP 不是基于可靠连接的通信，所以程序提示连接成功的时候，仅仅是个提示作用，并不是说就一定连接上了目标 IP 和端口号，因此，不一定能成功发送数据给对方，这个在使用的时候，要注意。

网络通信就为大家介绍到这里。

64.2.15 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 个探索者 STM32F4 开发板（也可以一个探索板与战舰板或 Mini 板搭配用，不过都要刷综合实验!!）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 64.2.15.1 所示：



图 64.2.15.1 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 64.2.15.2 所示：

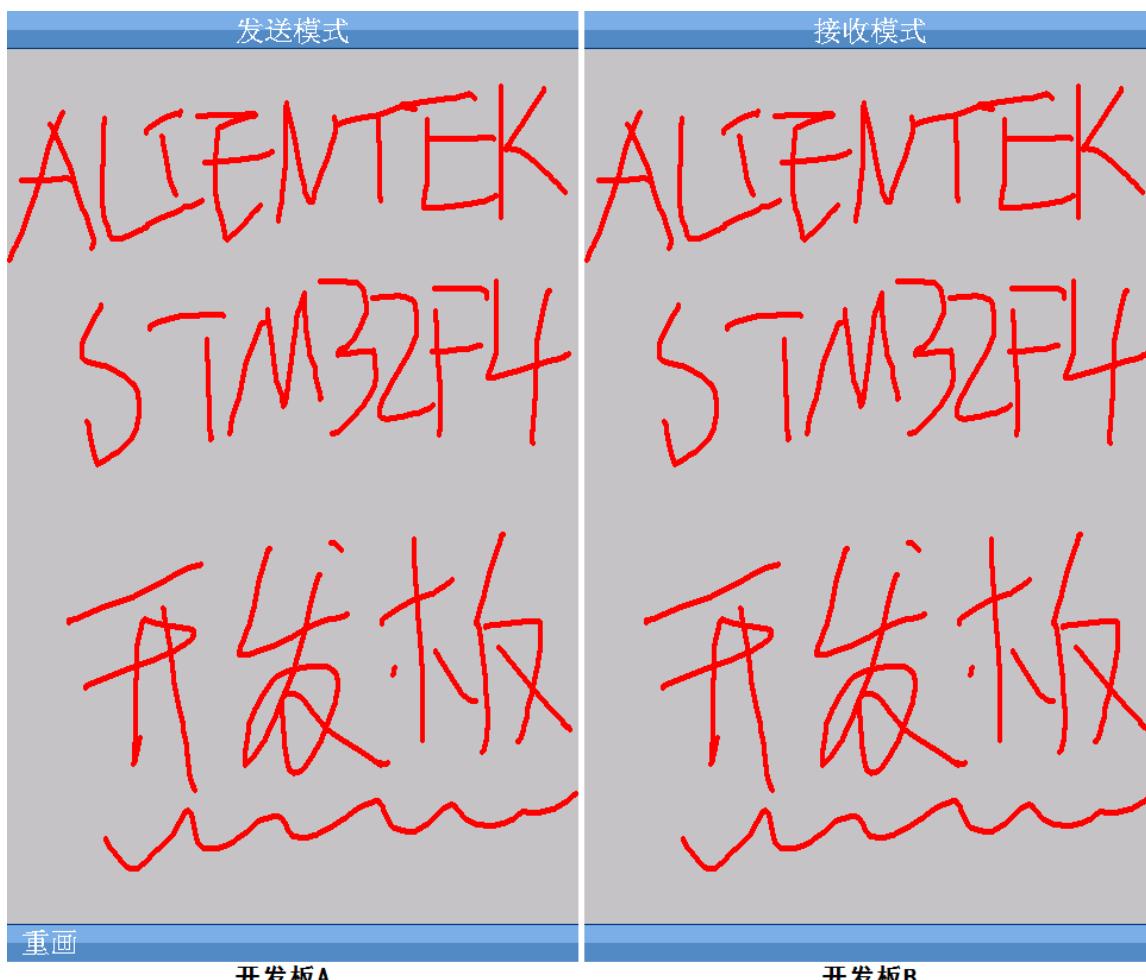


图 64.2.15.2 在开发板 A 输入的内容完整的显示在开发板 B 上

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。

64.2.16 计算器

探索者 STM32F4 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 64.2.16.1 所示：

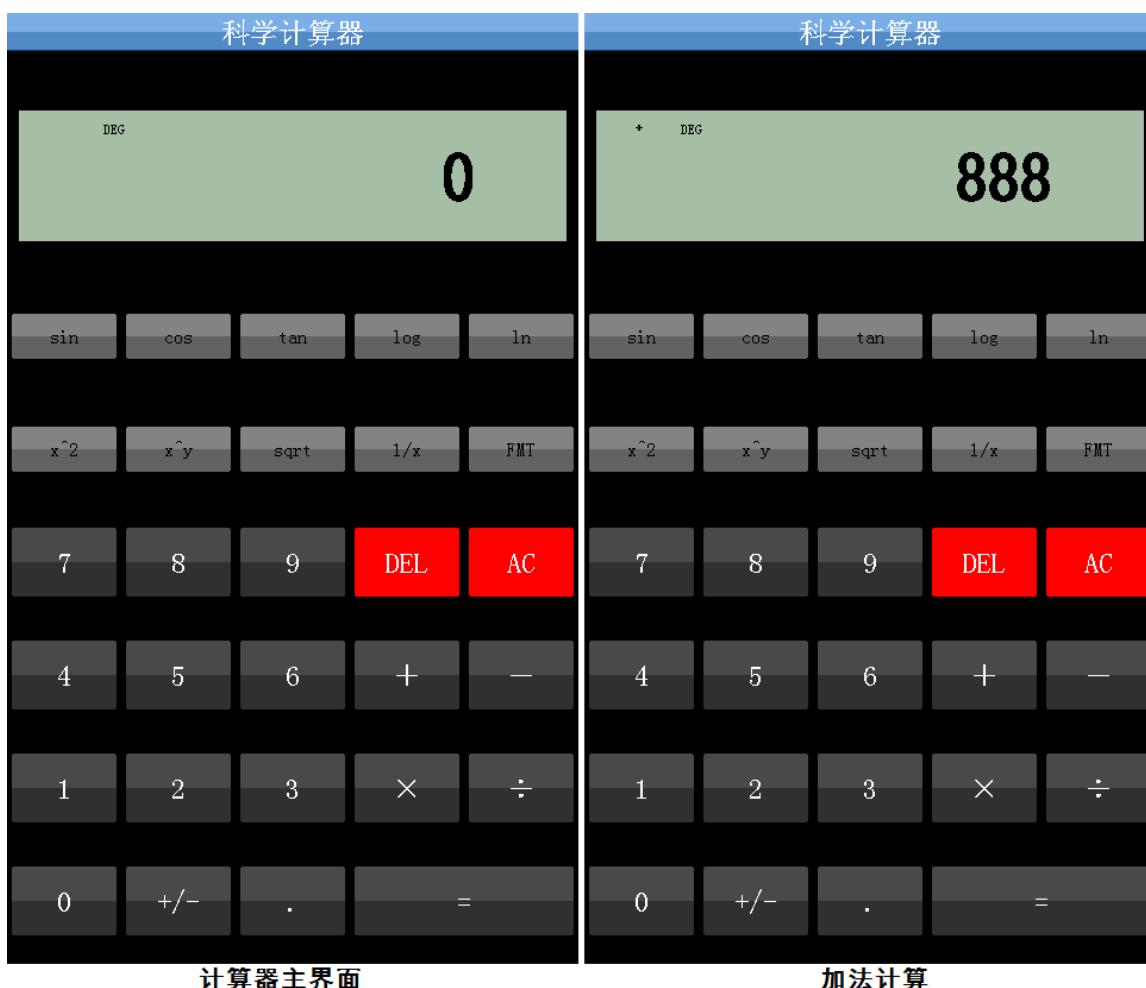


图 64.2.16.1 计算器主界面和加法计算

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 64.2.16.2 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 64.2.16.3 所示：

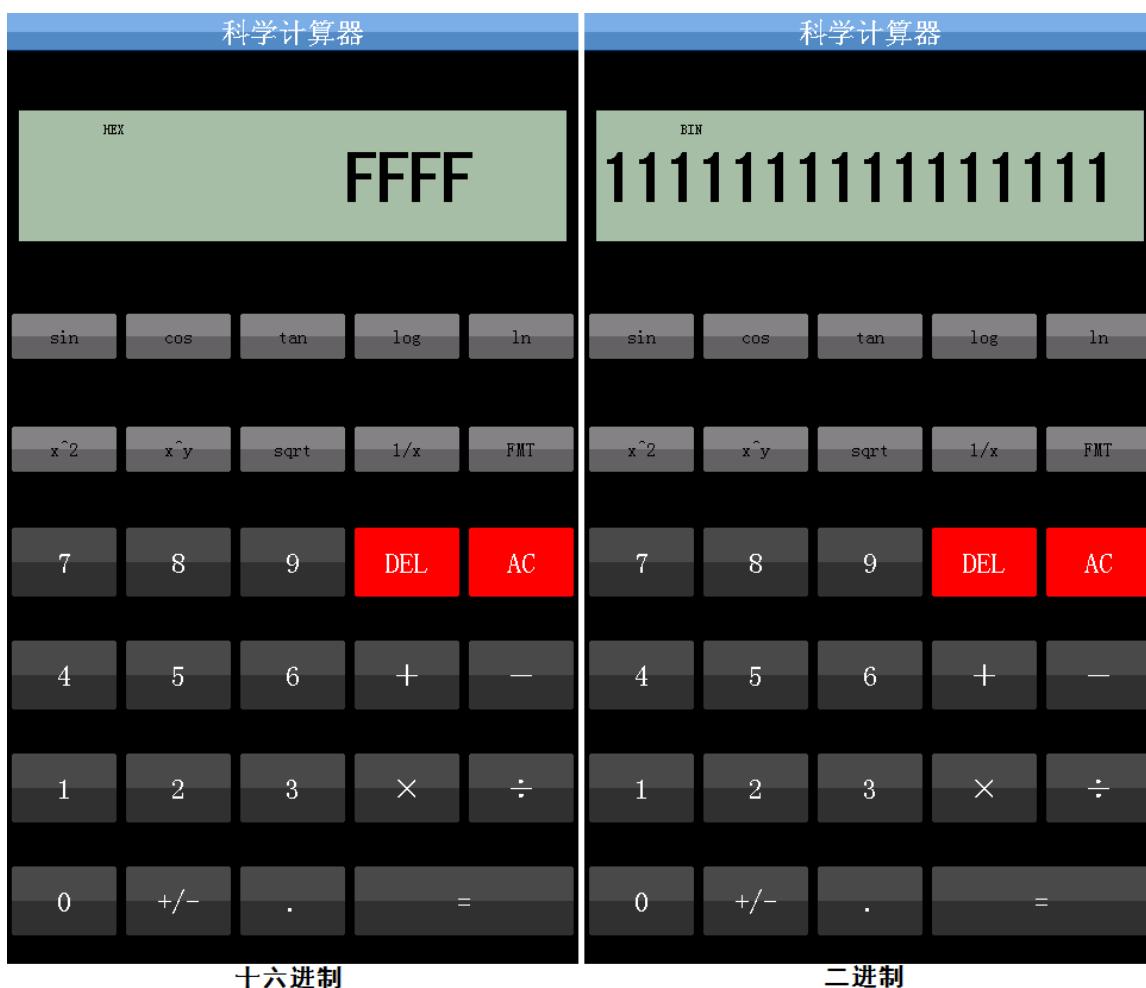


图 64.2.16.3 格式转换

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。

64.2.17 拨号

本综合实验支持拨打和接听电话，不过需要 ALIENTEK ATK-SIM900A GSM 模块的支持，所以本功能的测试，请先确保有 GSM 模块，并连接成功（详见 64.2 节开头部分）。

双击主界面的拨号图标，进入拨号界面，如图 64.2.17.1 所示：

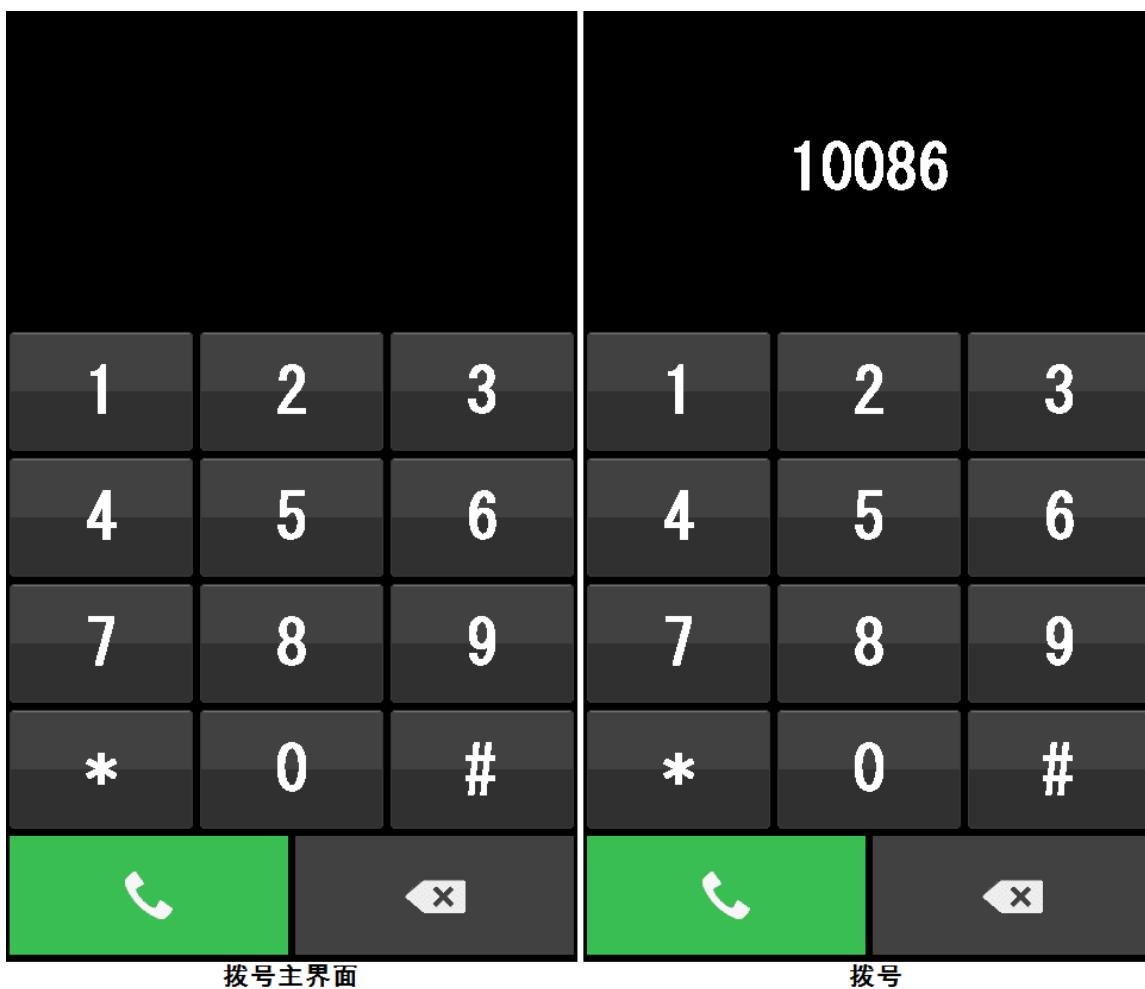


图 64.2.17.1 拨号主界面和拨号

上图中左侧图片就是拨号主界面，这个和手机拨号是一样的。右侧是我们输入的拨号号码，点击拨号图标，即可进行拨号。如图 64.2.17.2 所示。



图 64.2.17.2 拨号中和通话中

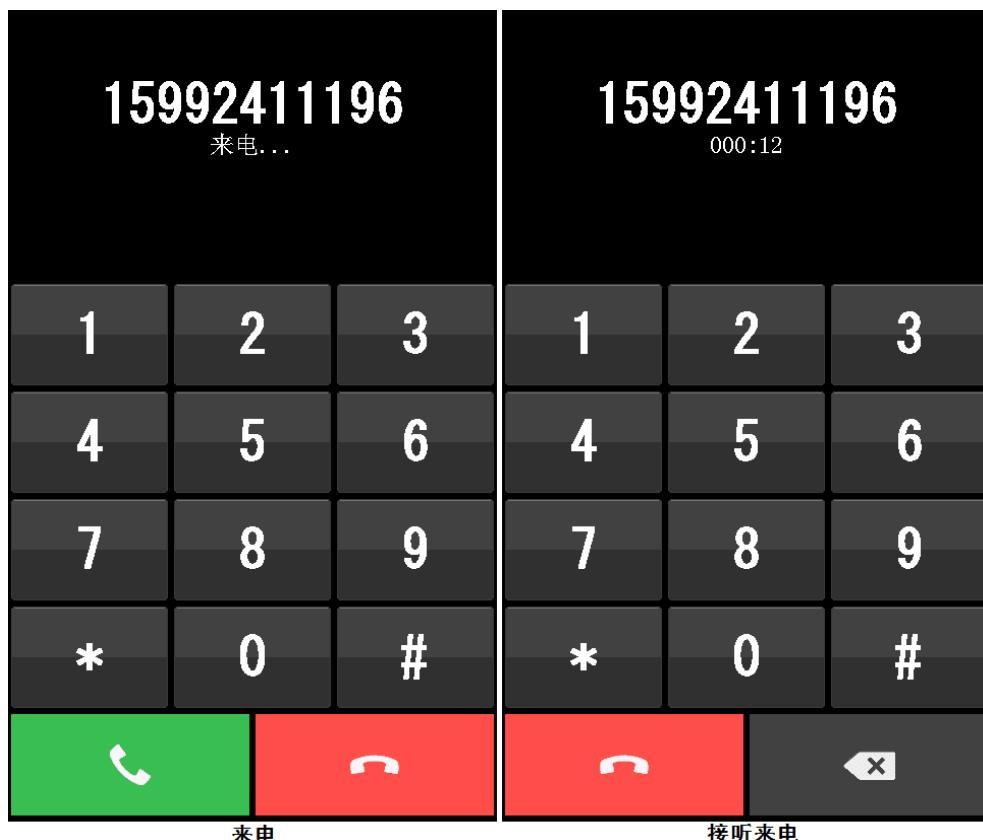


图 64.2.17.3 来电和接听来电

图 64.2.17.2 为拨号和拨通后的通话界面。图 64.2.17.3 为来电和接听来电后的通话界面，此时蜂鸣器会发出“滴、滴”的提示声，提示有电话呼入。其他的操作和我们智能手机基本一模一样，就无需多说了。

注意，在通话状态，如果按 TPAD，则会挂断电话，结束通话。

64.2.18 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 64.2.18.1 所示：

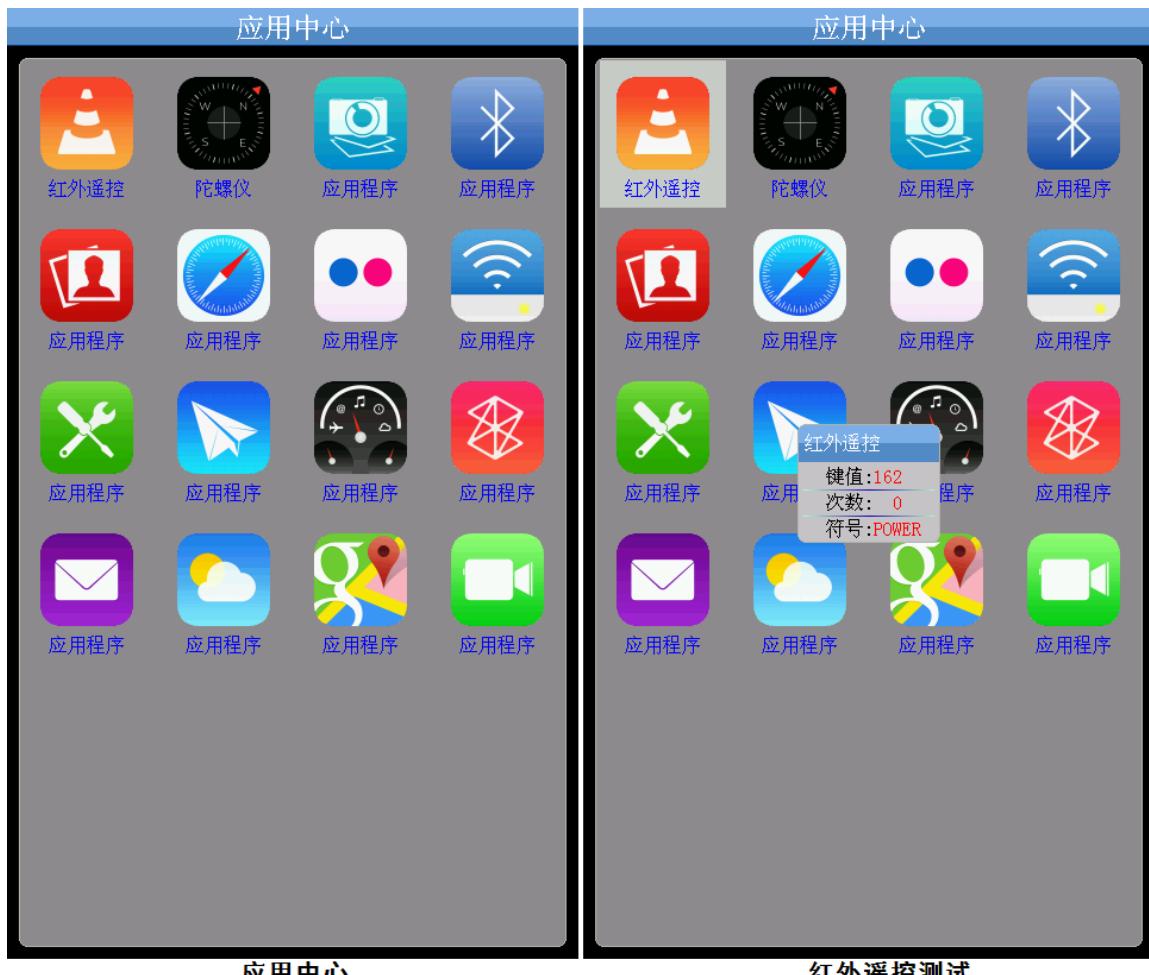


图 64.2.18.1 应用中心和红外遥控测试

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，我们仅实现了前两个：红外遥控和陀螺仪功能。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，此时，我们将红外遥控对准探索者 STM32F4 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 64.2.18.1 右侧图片所示。

按 TPAD 可以退出红外遥控功能，返回应用中心主界面，然后按第二个图标，即可进入陀螺仪测试界面，如图 64.2.18.2 所示：

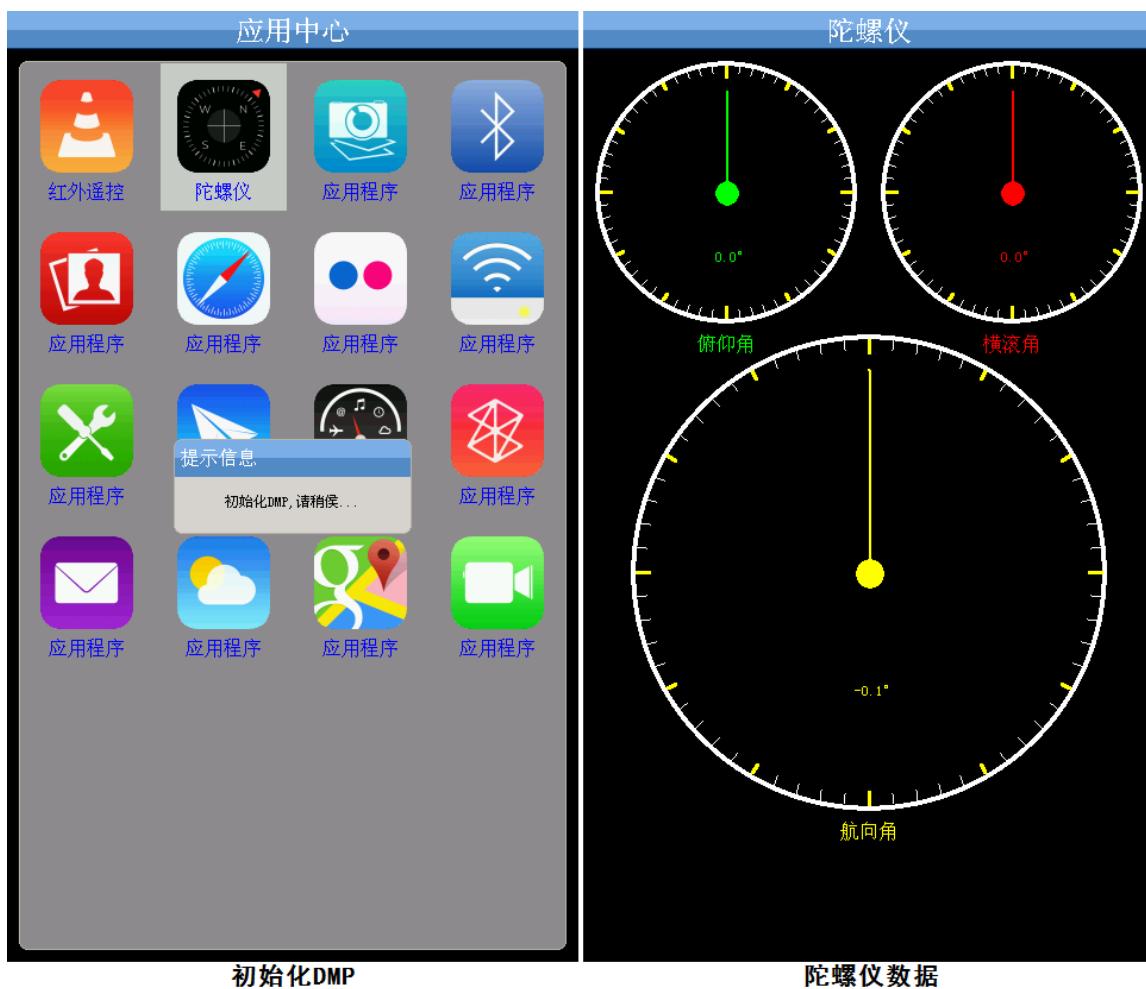


图 64.2.18.2 初始化 DMP 和陀螺仪测试

这里的陀螺仪，起始就是指开发板板载的 MPU6050 传感器，这里我们应用 MPU6050 独有的 DMP 功能，来实现姿态解算，得出欧拉角：俯仰角、横滚角和航向角。

上图中，左侧图片显示了正在初始化 MPU6050 的 DMP，在初始化成功后，显示姿态解算后的俯仰角、横滚角和航向角。此时，我们调整开发板的位置，就可以得到不同的俯仰角、横滚角和航向角。

按 TPAD 返回应用中心主界面，再按 TPAD 返回系统主界面。

64.2.19 短信

双击主界面的短信图标，开始读取 GSM 模块 SIM 卡中的短信，如图 64.2.19.1 所示：



图 64.2.19.1 短信读取中和读取到的短信

上图中，左侧显示了短信读取的进程，当所有短信读取完成后，显示读取到的短信，如图右侧图片所示。可以看出，SIM 卡中总共有 4 条短信，其中：前面有黑色实心圆标志的代表是未读的短信，前面有空心圆的表示读取过了的短信。

当短信内容大于一行宽度后，会采用走字的方式显示，起到预览的作用。在该界面，按 TPAD 可以返回系统主界面。

点击左下角的选型按钮，可以选择对短信的操作，如图 64.2.19.2 所示：



图 64.2.19.2 短信操作和新建信息

上图中，左侧图片显示的短信操作总共有三个操作：新建信息、阅读信息和删除信息。其中阅读信息也可以在读取到的短信界面，直接双击短信条目进行阅读。

右侧图片显示了新建信息的界面，新建信息是，收件人是可以编辑的，我们先输入收件人，比如 1008611，然后就可以进行对话了。如图 64.2.19.3 所示：



图 64.2.19.3 短信对话和重新回到读取到的短信界面

图中左侧的短信对话界面，有点类似现在手机的短信方式，收发双方的内容都显示在一个区域，可以通过滚动条拖动查看。图中是我们发数字给 1008611，以及 1008611 对我们做出的应答。每当一条短信发送成功后，蜂鸣器会有“滴”的一声短叫，提示发送成功。如果接受到新的短信，蜂鸣器会有“滴、滴”的两声短叫，然后新收到的短信（收件人发过来的）会实时添加到我们的对话中来。

图中，右侧图片是我们同 1008611 进行短信对话之后，按 TPAD，返回到读取到的短信界面。从图中可以看出，短信条数比图 64.2.19.1 中多了很多，说明收到了很多短信，且都已经自动添加到读取到的短信中来了。

短信的其他操作，就不详细介绍了，和手机基本一模一样，大家自己摸索下就可以了。

至此，整个探索者 STM32F4 开发板的综合测试实验就介绍完了。此代码在战舰 STM32 开发板的基础上进行修改而来，相比战舰板的综合实验，功能更多，更强悍。其中，参考了不少网友的代码，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。

综合实验整个代码编译后大小为 680K 左右 (-O2 优化后)，代码量是非常的大，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！

正点原子

2014-10-23

于广州

