STM32F1 开发指南

V1.2（精英板-HAL 库版本）

－正点原子精英 STM32F103 开发板教程



修订历史:

版本	日期	修改内容
V1.2	2022/01/22	1,更新为最新版本产品图片 2,增加修订历史记录及更新部分链接

正点原子

正点原子公司名称 ：广州市星翼电子科技有限公司
原子哥在线教学平台：www.yuanzige.com
开源电子网 / 论坛 ：www.openedv.com/forum.php
正点原子官方网站 ：www.alientek.com
正点原子淘宝店铺 ：https://openedv.taobao.com
正点原子 B 站视频 ：https://space.bilibili.com/394620890

电话：020-38271790 传真：020-36773971

请下载原子哥 APP，数千讲视频免费学习，更快更流畅。
请关注正点原子公众号，资料发布更新我们会通知。



扫码下载“原子哥”APP



扫码关注正点原子公众号

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 精英 STM32F103 资源初探 5
1.2 ALIENTEK 精英 STM32F103 资源说明 7
1.2.1 硬件资源说明 7
1.2.2 软件资源说明 11
1.2.3 精英板 IO 引脚分配 12
1.3 ALIENTEK 精英板与战舰板 V3.0 对比 15
第二章 实验平台硬件资源详解 17
2.1 开发板原理图详解 17
2.1.1 MCU 17
2.1.2 引出 IO 口 19
2.1.3 USB 串口/串口 1 选择接口 19
2.1.4 JTAG/SWD 20
2.1.5 LCD 模块接口 20
2.1.6 复位电路 21
2.1.7 启动模式设置接口 21
2.1.8 R485 选择接口 22
2.1.9 RS485 接口 22
2.1.10 CAN/USB 接口 22
2.1.11 EEPROM 23
2.1.12 光敏传感器 23
2.1.13 SPI FLASH 24
2.1.14 温湿度传感器接口 24
2.1.15 红外接收头 25
2.1.16 无线模块接口 25
2.1.17 LED 25
2.1.18 按键 26
2.1.19 TPAD 电容触摸按键 26

2.1.20 OLED/摄像头模块接口 27
2.1.21 有源蜂鸣器 27
2.1.22 SD 卡接口 28
2.1.23 ATK 模块接口 28
2.1.24 AD/DA 组合接口 29
2.1.25 电源 29
2.1.26 电源输入输出接口 30
2.1.27 USB 串口 30
2.2 开发板使用注意事项 31
2.3 STM32F103 学习方法 32
第二篇 软件篇 34
第三章 MDK5 软件入门 35
3.1 MDK5 简介 35
3.2 STM32CubeF1 简介 36
3. 2. 1 库开发与寄存器开发的关系 36
3. 2. 2 STM32CubeF1 固件包介绍 37
3. 2. 3 HAL 库和标准库选择 39
3.3 新建基于 HAL 库的工程模板和工程结构讲解 39
3.3.1 新建基于 HAL 库工程模板 39
3.3.2 工程模板解读 60
3.3.2.1 关键文件介绍 60
3.3.2.2 HAL 库中__weak 修饰符讲解 62
3.3.2.3 Msp 回调函数执行过程解读 63
3.3.2.4 程序执行流程图 64
3.4 程序下载与调试 66
3.4.1 STM32 串口程序下载 66
3.4.2 使用 ST-LINK 下载与调试程序 71
3.5 MDK5 使用技巧 77
3.5.1 文本美化 77
3.5.2 语法检测&代码提示 80
3.5.3 代码编辑/查看技巧 82
3.5.4 其他小技巧 86
第四章 STM32F1 基础知识入门 88

4.1 MDK 下 C 语言基础复习 88
4.1.1 位操作 88
4.1.2 define 宏定义 89
4.1.3# ifdef 和 #if defined 条件编译 89
4.1.4 extern 变量申明 90
4.1.5 typedef 类型别名 91
4.1.6 结构体 91
4.2 STM32F1 系统架构 93
4.3 STM32F103 时钟系统 94
4.3.1 STM32F103 时钟树概述 94
4.3.2 STM32F103 时钟系统配置 96
4.3.3 STM32F1 时钟使能和配置 100
4.4 端口复用和重映射 101
4.5 STM32 NVIC 中断优先级管理 102
4.6 HAL 库中寄存器地址名称映射分析 106
4.7 MDK 中使用 HAL 库快速组织代码技巧 108
第五章 **SYSTEM 文件夹介绍** 113
5.1 delay 文件夹代码介绍 113
5.1.1 操作系统支持宏定义及相关函数 114
5.1.2 delay_init 函数 116
5.1.3 delay_us 函数 117
5.1.4 delay_ms 函数 119
5.1.5 HAL 库延时函数 HAL_Delay 解析 120
5.2 sys 文件夹代码介绍 122
5.2.1 IO 口的位操作实现 122
5.3 usart 文件夹介绍 123
5.3.1 printf 函数支持 124
第三篇 实战篇 125
第六章 跑马灯实验 126
6.1 STM32F1 IO 简介 127
6.2 硬件设计 132
6.3 软件设计 132
6.4 下载验证 140

第七章 蜂鸣器实验 142
7.1 蜂鸣器简介 143
7.2 硬件设计 143
7.3 软件设计 144
7.4 下载验证 146
第八章 按键输入实验 147
8.1 STM32F1 IO 口简介 148
8.2 硬件设计 148
8.3 软件设计 148
8.4 下载验证 152
第九章 串口通信实验 153
9.1 STM32F1 串口简介 154
9.2 硬件设计 159
9.3 软件设计 160
9.4 下载验证 167
第十章 外部中断实验 170
10.1 STM32F1 外部中断简介 171
10.2 硬件设计 173
10.3 软件设计 173
10.4 下载验证 176
第十一章 独立看门狗（IWDG）实验 177
11.1 STM32F1 独立看门狗简介 178
11.2 硬件设计 181
11.3 软件设计 181
11.4 下载验证 182
第十二章 窗口门狗（WWDG）实验 183
12.1 STM32F1 窗口看门狗简介 184
12.2 硬件设计 187
12.3 软件设计 187
12.4 下载验证 189
第十三章 定时器中断实验 190
13.1 STM32F1 通用定时器简介 191

13.2 硬件设计 195
13.3 软件设计 195
13.4 下载验证 198
第十四章 PWM 输出实验 199
14.1 PWM 简介 200
14.2 硬件设计 204
14.3 软件设计 204
14.4 下载验证 207
第十五章 输入捕获实验 208
15.1 输入捕获简介 209
15.2 硬件设计 214
15.3 软件设计 214
15.4 下载验证 218
第十六章 电容触摸按键实验 220
16.1 电容触摸按键简介 221
16.2 硬件设计 222
16.3 软件设计 222
16.4 下载验证 227
第十七章 OLED 显示实验 229
17.1 OLED 简介 230
17.2 硬件设计 236
17.3 软件设计 237
17.4 下载验证 244
第十八章 TFTLCD 显示实验 246
18.1 TFTLCD&FSMC 简介 247
18.1.1 TFTLCD 简介 247
18.1.2 FSMC 简介 252
18.2 硬件设计 261
18.3 软件设计 262
18.4 下载验证 273
第十九章 USMART 调试组件实验 274
19.1 USMART 调试组件简介 275
19.2 硬件设计 278

19.3 软件设计 278
19.4 下载验证 281
第二十章 RTC 实时时钟实验 286
20.1 STM32F1 RTC 时钟简介 287
20.2 硬件设计 295
20.3 软件设计 295
20.4 下载验证 301
第二十一章 待机唤醒实验 303
21.1 STM32F1 待机模式简介 304
21.2 硬件设计 307
21.3 软件设计 307
21.4 下载与测试 310
第二十二章 ADC 实验 311
22.1 STM32F1 ADC 简介 312
22.2 硬件设计 319
22.3 软件设计 319
22.4 下载验证 322
第二十三章 内部温度传感器实验 324
23.1 STM32F1 内部温度传感器简介 325
23.2 硬件设计 325
23.3 软件设计 325
23.4 下载验证 327
第二十四章 光敏传感器实验 328
24.1 光敏传感器简介 329
24.2 硬件设计 329
24.3 软件设计 329
24.4 下载验证 332
第二十五章 DAC 实验 334
25.1 STM32F1 DAC 简介 335
25.2 硬件设计 339
25.3 软件设计 340
25.4 下载验证 343
第二十六章 DMA 实验 345

26.1 STM32F1 DMA 简介 346
26.2 硬件设计 351
26.3 软件设计 352
26.4 下载验证 355
第二十七章 IIC 实验 357
27.1 IIC 简介 358
27.2 硬件设计 358
27.3 软件设计 359
27.4 下载验证 367
第二十八章 SPI 实验 369
28.1 SPI 简介 370
28.2 硬件设计 373
28.3 软件设计 374
28.4 下载验证 380
第二十九章 485 实验 381
29.1 485 简介 382
29.2 硬件设计 383
29.3 软件设计 384
29.4 下载验证 388
第三十章 CAN 通讯实验 390
30.1 CAN 简介 391
30.2 硬件设计 410
30.3 软件设计 411
30.4 下载验证 417
第三十一章 触摸屏实验 420
31.1 触摸屏简介 421
31.1.1 电阻式触摸屏 421
31.1.2 电容式触摸屏 421
31.2 硬件设计 426
31.3 软件设计 427
31.4 下载验证 443
第三十二章 红外遥控实验 445
32.1 红外遥控简介 446

32.2 硬件设计 447
32.3 软件设计 448
32.4 下载验证 453
第三十三章 DS18B20 数字温度传感器实验 455
33.1 DS18B20 简介 456
33.2 硬件设计 457
33.3 软件设计 458
33.4 下载验证 462
第三十四章 DHT11 数字温湿度传感器实验 464
34.1 DHT11 简介 465
34.2 硬件设计 467
34.3 软件设计 467
34.4 下载验证 471
第三十五章 MPU6050 六轴传感器实验 472
35.1 MPU6050 简介 473
35.1.1 MPU6050 基础介绍 473
35.1.2 DMP 使用简介 477
35.2 硬件设计 481
35.3 软件设计 483
35.4 下载验证 491
第三十六章 无线通信实验 494
36.1 NRF24L01 无线模块简介 495
36.2 硬件设计 495
36.3 软件设计 496
36.4 下载验证 505
第三十七章 FLASH 模拟 EEPROM 实验 507
37.1 STM32 FLASH 简介 508
37.2 硬件设计 514
37.3 软件设计 514
37.4 下载验证 518
第三十八章 摄像头实验 520
38.1 OV7725 简介 521
38.2 硬件设计 525

38.3 软件设计 527
38.4 下载验证 536
第三十九章 内存管理实验 539
39.1 内存管理简介 540
39.2 硬件设计 541
39.3 软件设计 541
39.4 下载验证 548
第四十章 SD 卡实验 550
40.1 SDIO 简介 551
40.1.1 SDIO 主要功能及框图 551
40.1.2 SDIO 的时钟 552
40.1.3 SDIO 的命令与响应 552
40.1.4 SDIO 相关寄存器介绍 554
40.1.5 SD 卡初始化流程 559
40.2 硬件设计 561
40.3 软件设计 562
40.4 下载验证 567
第四十一章 FATFS 实验 569
41.1 FATFS 简介 570
41.2 硬件设计 575
41.3 软件设计 575
41.4 下载验证 583
第四十二章 汉字显示实验 585
42.1 汉字显示原理简介 586
42.2 硬件设计 590
42.3 软件设计 590
42.4 下载验证 599
第四十三章 图片显示实验 601
43.1 图片格式简介 602
43.2 硬件设计 603
43.3 软件设计 604
43.4 下载验证 613
第四十四章 照相机实验 615

44.1 BMP 编码简介 616
44.2 硬件设计 618
44.3 软件设计 619
44.4 下载验证 625
第四十五章 手写识别实验 627
45.1 手写识别简介 628
45.2 硬件设计 631
45.3 软件设计 632
45.4 下载验证 637
第四十六章 T9 拼音输入法实验 639
46.1 拼音输入法简介 640
46.2 硬件设计 642
46.3 软件设计 642
46.4 下载验证 650
第四十七章 串口 IAP 实验 652
47.1 IAP 简介 653
47.2 硬件设计 658
47.3 软件设计 659
47.4 下载验证 665
第四十八章 USB 虚拟串口实验 667
48.1 USB 简介 668
48.2 硬件设计 670
48.3 软件设计 671
48.4 下载验证 675
第四十九章 USB 读卡器实验 677
49.1 USB 读卡器简介 678
49.2 硬件设计 678
49.3 软件设计 679
49.4 下载验证 682
第五十章 UCOSII 实验 1-任务调度 684
50.1 UCOSII 简介 685
50.2 硬件设计 688
50.3 软件设计 689

50.4 下载验证 692
50.5 任务删除，挂起和恢复测试 692
第五十一章 UCOSII 实验 2-信号量和邮箱 698
51.1 UCOSII 信号量和邮箱简介 699
51.2 硬件设计 701
51.3 软件设计 701
51.4 下载验证 708
第五十二章 UCOSII 实验 3-消息队列、信号量集和软件定时器 710
52.1 UCOSII 消息队列、信号量集和软件定时器简介 711
52.2 硬件设计 718
52.3 软件设计 718
52.4 下载验证 728

内容简介

本手册将由浅入深，带领大家学习 STM32F103 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，主要介绍本手册硬件平台；2，软件篇，主要介绍 STM32F1 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 48 个实例（绝大部分是直接操作寄存器完成的）带领大家一步步深入了解 STM32F1。

本手册为 ALIENTEK 精英 STM32F103 V1 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，不会有任何警告和错误，另外，源码有我们生成好的 hex 文件，大家只需要通过串口/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。

本手册非常适合广大学生和电子爱好者学习 STM32F103 系列芯片，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。

本手册是 ALIENTEK 精英 STM32F103 开发板的教程，而精英 STM32F103 开发板则是战舰 STM32F103 V1 开发板的精简版，精英板上的所有资源，战舰 STM32F103 V1 开发板上面都有，而且这些相同的资源，就连 IO 的连接关系也是完全一样的，没有任何变化，所以精英板的代码，全部可以在战舰 V3 上面运行，且不用做任何修改。同样，战舰板上的代码，只要精英板有对应资源的，可以直接下载到精英板上直接使用，不用做任何修改。

因此，本教程就是战舰板配套教程《STM32F1 开发指南》-HAL 库版的精简版，绝大部分内容，和战舰板的教程：《STM32F1 开发指南》-HAL 库版完全一样，只有有差异的地方，才会有所不同。

前言

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

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

现在 ST 公司又推出了 STM32F0 系列 Cortex M0 芯片以及 STM32F4/F3 系列 Coretx M4 芯片，这些都已经量产，而且可以比较方便的购买到，本手册，我们只讨论 Cortex M3（Cortex M4 的介绍请看《STM32F4 开发指南.pdf》）。

STM32 的优异性体现在如下几个方面：

1，超低的价格。以 8 位机的价格，得到 32 位机，是 STM32 最大的优势。

2，超多的外设。STM32 拥有包括：FSMC、TIMER、SPI、IIC、USB、CAN、IIS、SDIO、ADC、DAC、RTC、DMA 等众多外设及功能，具有极高的集成度。

3，丰富的型号。STM32 仅 M3 内核就拥有 F100、F101、F102、F103、F105、F107、F207、F217 等 8 个系列上百种型号，具有 QFN、LQFP、BGA 等封装可供选择。同时 STM32 还推出了 STM32L 和 STM32W 等超低功耗和无线应用型的 M3 芯片。

4，优异的实时性能。84 个中断，16 级可编程优先级，并且所有的引脚都可以作为中断输入。

5，杰出的功耗控制。STM32 各个外设都有自己的独立时钟开关，可以通过关闭相应外设的时钟来降低功耗。

6，极低的开发成本。STM32 的开发不需要昂贵的仿真器，只需要一个串口即可下载代码，并且支持 SWD 和 JTAG 两种调试口。SWD 调试可以为你的设计带来跟多的方便，只需要 2 个 IO 口，即可实现仿真调试。

学习 STM32 有两份不错的中文资料：

《STM32 参考手册》中文版 V10.0

《Cortex-M3 权威指南》中文版（宋岩 译）

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

STM32 拥有非常多的寄存器，对于新手来说，直接操作寄存器有一定的难度，所以 ST 官方提供了一套固件库函数，大家不需要再直接操作繁琐的寄存器，而是直接调用固件库函数即可实现操作寄存器的目的。库函数有两种，一是标准库，现在已经不再更新，另一种是 HAL 库，目前官方不断更新和完善。本手册是针对 HAL 库进行教程更新的。当然，我们要了解一些外设的原理，必须对寄存器有一定的了解，这对以后开发和调试也是非常有帮助的，所以在我们手册中我们会保留一些重要寄存器的讲解，但是我们的实例代码基本都是调用 HAL 库来实现的。有关寄存器操作的实例，大家可以参考我们寄存器版本的手册及代码。

本指南将结合《STM32 参考手册》和《Cortex-M3 权威指南》两者的优点，并从固件库级别出发，深入浅出，向读者展示 STM32 的各种功能。总共配有 48 个实例，基本上每个实例在均配有软硬件设计，在介绍完软硬件之后，马上附上实例代码，并带有详细注释及说明，让读者快速理解代码。

这些实例涵盖了 STM32 的绝大部分内部资源，并且提供很多实用级别的程序，如：内存管理、拼音输入法、手写识别、图片解码、IAP 等。所有实例在 MDK5 编译器下编译通过，大家只需下载程序到 ALIENTEK 精英 STM32 开发板，即可验证实验。

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

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

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

第一篇 硬件篇

实践出真知，要想学好 STM32F1，实验平台必不可少！本篇将详细介绍我们用来学习 STM32F1 的硬件平台：ALIENTEK 精英 STM32F103 开发板，通过该篇的介绍，你将了解到我们的学习平台 ALIENTEK 精英 STM32F103 的功能及特点。

为了让读者更好的使用 ALIENTEK 精英 STM32F103，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 精英 STM32F103。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 精英 STM32F103 资源初探；

1.2，ALIENTEK 精英 STM32F103 资源说明；

1.1 ALIENTEK 精英 STM32F103 资源初探

自从 2012 年上市以来，ALIENTEK 战舰 STM32F103 开发板就广受客户好评，并常年稳居淘宝 STM32 系列开发板销量冠军。精英 STM32F103 开发板，则是在最新的战舰 STM32F103 V3 开发板的基础上精简而来，以满足不同客户的需求。下面我们开始介绍精英 STM32F103 开发板。

正点原子精英 STM32F103 V1.0 的资源图如图 1.1.1 所示：

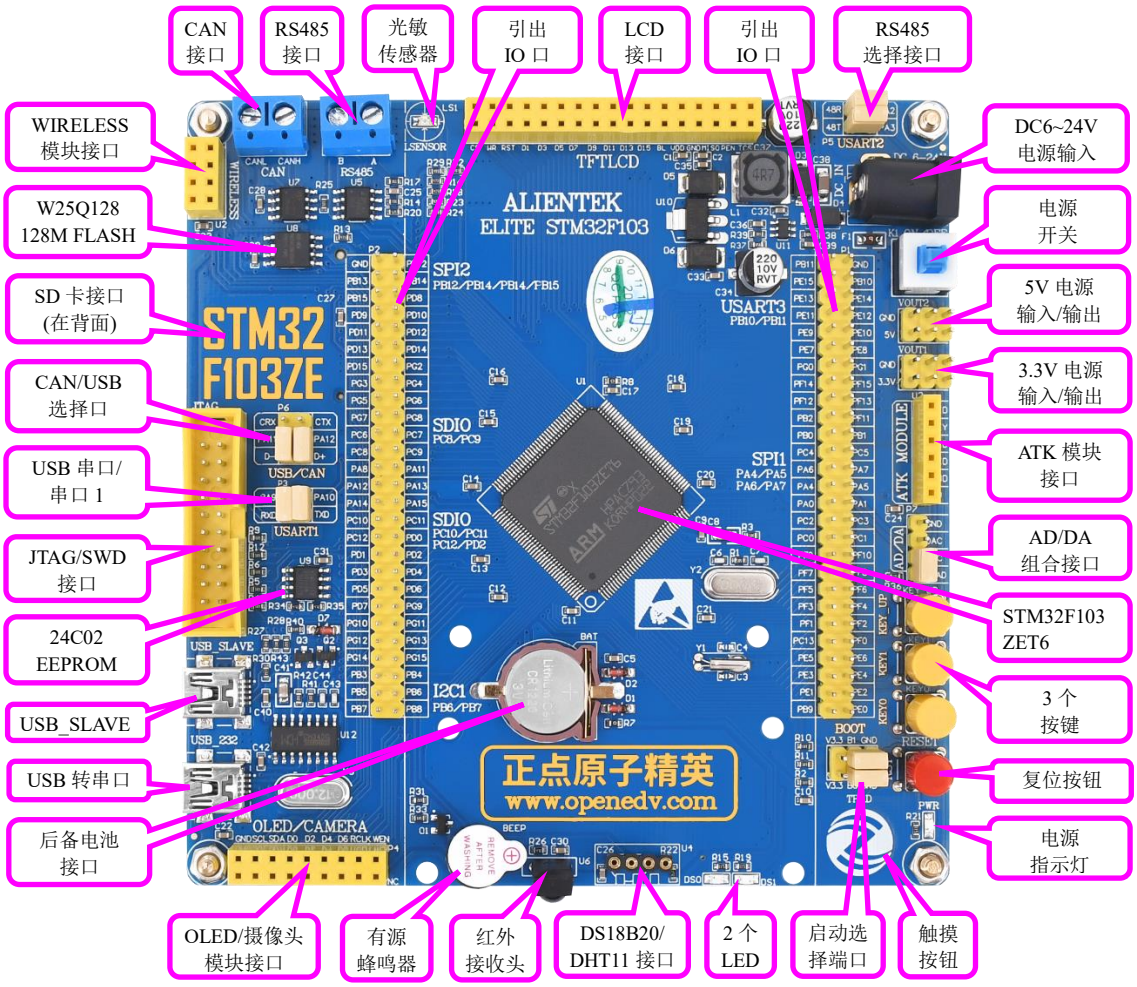


图 1.1.1 精英 STM32F103 资源图

从图 1.1.1 可以看出，ALIENTEK 精英 STM32F103，资源丰富，并充分利用了 STM32F103 的内部资源，基本所有 STM32F103 的内部资源，都可以在此开发板上验证，同时扩充丰富的接口和功能模块，整个开发板小巧精致。

开发板的外形尺寸为 115mm*117mm 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，在战舰 V3 的基础上，将大家可能用不到的资源进行了裁剪，经过多次改进，最终确定了这样的设计。

ALIENTEK 精英 STM32F103 板载资源如下：

- CPU：STM32F103ZET6，LQFP144，FLASH：512K，SRAM：64K；
- 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 个光敏传感器
- 1 个无线模块接口（可接 NRF24L01/RFID 模块等）
- 1 路 CAN 接口，采用 TJA1050 芯片
- 1 路 485 接口，采用 SP3485 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/MPU6050 模块等
- 1 个标准的 2.4/2.8/3.5/4.3/7 寸 LCD 接口，支持触摸屏
- 1 个摄像头模块接口
- 1 个 OLED 模块接口（与摄像头接口共用）
- 1 个 USB 串口，可用于程序下载和代码调试（USMART 调试）
- 1 个 USB SLAVE 接口，用于 USB 通信
- 1 个有源蜂鸣器
- 1 个 RS485 选择接口
- 1 个 CAN/USB 选择接口
- 1 个串口选择接口
- 1 个 SD 卡接口（在板子背面，SDIO 接口）
- 1 个标准的 JTAG/SWD 调试下载口
- 1 组 AD/DA 组合接口（DAC/ADC/ TPAD）
- 1 组 5V 电源供应/接入口
- 1 组 3.3V 电源供应/接入口
- 1 个直流电源输入接口（输入电压范围：6~24V）
- 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
- 3 个功能按钮，其中 KEY_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 精英 STM32F103 的特点包括：

1） 小巧精致。板子尺寸为 11.5cm*11.7cm，接口丰富，布局紧凑。

2） 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。

3） 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。

4） 资源充足。主芯片采用自带 512K 字节 FLASH 的 STM32F103ZET6，并外扩 16M 字节 SPI FLASH，满足大数据存储需求。

5） 人性化设计。各个接口都有丝印标注，且用方框框出，使用起来一目了然；部分常用外设大丝印标出，方便查找；接口位置设计合理，方便顺手。资源搭配合理，物尽其用。

1.2 ALIENTEK 精英 STM32F103 资源说明

资源说明部分，我们将分为三个部分说明：硬件资源说明、软件资源说明和精英板 IO 引脚分配。

1.2.1 硬件资源说明

这里我们详细介绍精英 STM32F103 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. WIRELESS 模块接口

这是开发板板载的无线模块接口（U2），可以外接 NRF24L01/RFID 等无线模块。从而实现无线通信等功能。注意：接 NRF24L01 模块进行无线通信的时候，必须同时有 2 个模块和 2 个板子，才可以测试，单个模块/板子例程是不能测试的。

2. W25Q128 128M FLASH

这是开发板外扩的 SPI FLASH 芯片（U8），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，也可以是 TF 卡+卡套的形式），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. CAN/USB 选择口

这是一个 CAN/USB 的选择接口（P6），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。

5. USB 串口/串口 1

这是 USB 串口同 STM32F103ZET6 的串口 1 进行连接的接口（P3），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 精英 STM32F103 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 精英 STM32F103 开发板上。

6. JTAG/SWD 接口

这是 ALIENTEK 精英 STM32F103 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！

7. 24C02 EEPROM

这是开发板板载的 EEPROM 芯片（U9），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。

8. 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。

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。

9. 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 TTL 串口。同时，此 MiniUSB 接头也是开发板电源的主要提供口。

10. 后备电池接口

这是 STM32 后备区域的供电接口(BAT)，可安装 CR1220 电池（默认安装了），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. 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（P4），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 种外部模块，从而实现相关实验。

12. 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/闹铃等功能。

13. 红外接收头

这是开发板的红外接收头（U6），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。

精英 STM32F103 给大家配备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

14. DS18B20/DHT11 接口

这是开发板的一个复用接口（U4），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。

15. 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。精英 STM32F103 几乎每个实例都使用了 LED 来指示程序的运行状态。

16. 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 2.1.7 小节。

17. 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。

18. 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。

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. 3 个按键

这是开发板板载的 3 个机械式输入按键（KEY0、KEY1 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 2 个是普通按键，可以用于人机交互的输入，这 2 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 和 KEY1 是低电平有效，大家在使用的时候留意一下。

21. STM32F103ZET6

这是开发板的核心芯片（U1），型号为：STM32F103ZET6。该芯片具有 64KB SRAM、512KB FLASH、2 个基本定时器、4 个通用定时器、2 个高级定时器、2 个 DMA 控制器（共 12 个通道）、3 个 SPI、2 个 IIC、5 个串口、1 个 USB、1 个 CAN、3 个 12 位 ADC、1 个 12 位 DAC、1 个 SDIO 接口、1 个 FSMC 接口以及 112 个通用 IO 口。

22. AD/DA 组合接口

这是 1 个由 4 个排针组成的一个组合接口（P7）。可以实现 AD 采集、DA 输出和板载电容触摸按键（TPAD）检测的功能。

23. ATK 模块接口

这是开发板板载的一个 ALIENTEK 通用模块接口（U3），目前可以支持 ALIENTEK 开发的 GPS 模块、蓝牙模块和 MPU6050 模块等，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。

24. 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 精英 STM32F103，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）。

25. 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）。

26. 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。

27. DC6~24V 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 DC6~24V 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。

28. RS485 选择接口

这是开发板板载的 RS485 选择接口（P5），MAX3485 通过这个接口来决定是否连接到 STM32 的串口 2（USART2），当这里断开的时候：串口 2 可以用来做普通串口使用，而 RS485 则可以用来实现 RS485 转 TTL 的功能；当这里接上时：串口 2 连接 MAX3485，就可以实现 RS485 通信。 。

29. 引出 IO 口（共 2 组）

这是开发板 IO 引出端口，总共有 2 组主 IO 引出口：P1 和 P2。它们采用 2*27 排针引出，

总共引出 106 个 IO 口。而 STM32F103ZET6 总共只有 112 个 IO，除去 RTC 晶振占用的 2 个 IO，还剩下 110 个，这 2 组排针，总共引出：106 个 IO，剩下的 4 个 IO 分别通过：P3 和 P5 引出。

30. 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。

31. 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。

32. RS485 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！另外，开发板自带了终端电阻（120Ω）。

33. 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 精英 STM32F103 开发板的硬件资源。接下来，我们将向大家简要介绍一下精英 STM32F103 开发板的软件资源。

精英 STM32F103 开发板提供的标准例程多达 48 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，对初学者来说不那么容易入门。

精英 STM32F103 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	25	CAN 收发实验
2	蜂鸣器实验	26	触摸屏实验
3	按键输入实验	27	红外遥控实验
4	串口实验	28	DS18B20 数字温度传感器实验
5	外部中断实验	29	DHT11 数字温湿度传感器实验
6	独立看门狗实验	30	MPU6050 六轴传感器实验
7	窗口看门狗实验	31	无线通信实验
8	定时器中断实验	32	FLASH 模拟 EEPROM 实验
9	PWM 输出实验	33	摄像头实验
10	输入捕获实验	34	内存管理实验
11	电容触摸按键实验	35	SD 卡实验
12	OLED 显示实验	36	FATFS 实验
13	TFTLCD 显示实验	37	汉字显示实验
14	USMART 调试实验	38	图片显示实验
15	RTC 实验	39	照相机实验

16	待机唤醒实验	40	手写识别实验
17	ADC 实验	41	T9 拼音输入法实验
18	内部温度传感器实验	42	串口 IAP 实验
19	光敏传感器实验	43	USB 虚拟串口实验
20	DAC 实验	44	USB 读卡器实验
21	DMA 实验	45	UCOSII 实验 1-任务调度
22	IIC 实验	46	UCOSII 实验 2-信号量和邮箱
23	SPI 实验	47	UCOSII 实验 3-消息队列、信号量集和软件定时器
24	485 实验	48	综合测试实验

表 1.2.2.1 ALIENTEK 精英 STM32F103 开发板例程表

从上表可以看出，ALIENTEK 精英 STM32F103 的例程基本上涵盖了 STM32F103ZET6 的所有内部资源，并且外扩展了很多有价值的例程，比如：FLASH 模拟 EEPROM 实验、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、综合测试实验等。

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 精英 STM32F103 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 精英 STM32F103 也绝对是一个不错的选择。

1.2.3 精英板 IO 引脚分配

为了让大家更快更好的使用我们的精英 STM32F103 开发板，这里特地将精英板主芯片：STM32F103ZET6 的 IO 资源分配做了一个总表，以便大家查阅。精英 STM32 开发板的 IO 引脚分配总表如表：1.2.3.1 所示：

精英板 IO 资源分配表					
引脚	GPIO	连接资源		独立	连接关系说明
34	PA0	WK_UP		Y	1，按键 KEY_UP 2，可以做待机唤醒脚(WKUP)
35	PA1	STM_ADC	TPAD	Y	ADC 输入引脚，同时做 TPAD 检测脚
36	PA2	USART2_TX	485_RX	Y	RS485 RX 脚(P5 设置)
37	PA3	USART2_RX	485_TX	Y	RS485 TX 脚(P5 设置)
40	PA4	STM_DAC	GBC_KEY	Y	1，DAC_OUT1 输出脚 2，ATK-MODULE 接口的 KEY 引脚
41	PA5			Y	未接任何外设
42	PA6			Y	未接任何外设
43	PA7			Y	未接任何外设
100	PA8	OV_VSYNC		Y	OLED/CAMERA 接口的 VSYNC 脚
101	PA9	USART1_TX		Y	串口 1 TX 脚，默认连接 CH340 的 RX(P3 设置)
102	PA10	USART1_RX		Y	串口 1 RX 脚，默认连接 CH340 的 TX(P3 设置)
103	PA11	USB_D-	CRX	Y	1，USB D-引脚(P6 设置) 2，CRX 引脚(P6 设置)

104	PA12	USB_D+	CTX	Y	1，USB D+引脚(P6 设置) 2，CTX 引脚(P6 设置)
105	PA13	JTMS	SWDIO	N	JTAG/SWD 仿真接口,没接任何外设 注意：如要做普通 IO，需先禁止 JTAG&SWD
109	PA14	JTCK	SWDCLK	N	JTAG/SWD 仿真接口,没接任何外设 注意：如要做普通 IO，需先禁止 JTAG&SWD
110	PA15	JTDI	GBC_LED	N	1，JTAG 仿真口(JTDI) 2，ATK-MODULE 接口的 LED 引脚(使用时，需先禁止 JTAG，才可以当普通 IO 使用)
46	PB0	LCD_BL		Y	TFTLCD 接口背光控制脚
47	PB1	T_SCK		Y	TFTLCD 接口触摸屏 SCK 信号
48	PB2	BOOT1	T_MISO	N	1，BOOT1,启动选择配置引脚(仅上电时用) 2，TFTLCD 接口触摸屏 MISO 信号
133	PB3	JTDO	FIFO_WEN	N	1，JTAG 仿真口(JTDO) 2,OLED/CAMERA 接口 WEN 脚(使用时,需先禁止 JTAG，才可以当普通 IO 使用)
134	PB4	JTRST	FIFO_RCLK	N	1，JTAG 仿真口(JTRST) 2，OLED/CAMERA 接口 RCLK 脚(使用时，需先禁止 JTAG，才可以当普通 IO 使用)
135	PB5	LED0		N	接 DS0 LED 灯(红色)
136	PB6	IIC_SCL		N	接 24C02 的 SCL
137	PB7	IIC_SDA		N	接 24C02 的 SDA
139	PB8	BEEP		N	接蜂鸣器(BEEP)
140	PB9	REMOTE_IN		N	接 HS0038 红外接收头
69	PB10	GBC_RX		Y	接 ATK-MODULE 接口的 RXD 脚
70	PB11	GBC_TX		Y	接 ATK-MODULE 接口的 TXD 脚
73	PB12	F_CS		N	W25Q128 的片选信号
74	PB13	SPI2_SCK		N	W25Q128 和 WIRELESS 接口的 SCK 信号
75	PB14	SPI2_MISO		N	W25Q128 和 WIRELESS 接口的 MISO 信号
76	PB15	SPI2_MOSI		N	W25Q128 和 WIRELESS 接口的 MOSI 信号
26	PC0	OV_D0		Y	OLED/CAMERA 接口的 D0 脚
27	PC1	OV_D1		Y	OLED/CAMERA 接口的 D1 脚
28	PC2	OV_D2		Y	OLED/CAMERA 接口的 D2 脚
29	PC3	OV_D3		Y	OLED/CAMERA 接口的 D3 脚
44	PC4	OV_D4		Y	OLED/CAMERA 接口的 D4 脚
45	PC5	OV_D5		Y	OLED/CAMERA 接口的 D5 脚
96	PC6	OV_D6		Y	OLED/CAMERA 接口的 D6 脚
97	PC7	OV_D7		Y	OLED/CAMERA 接口的 D7 脚
98	PC8	SDIO_D0		N	SD 卡接口的 D0
99	PC9	SDIO_D1		N	SD 卡接口的 D1
111	PC10	SDIO_D2		N	SD 卡接口的 D2
112	PC11	SDIO_D3		N	SD 卡接口的 D3

113	PC12	SDIO_SCK		Y	SD 卡接口的 SCK
7	PC13			N	未接任何外设
8	PC14		RTC 晶振	N	接 32.768K 晶振，不可用做 IO
9	PC15		RTC 晶振	N	接 32.768K 晶振，不可用做 IO
114	PD0	FSMC_D2		Y	FSMC 总线数据线 D2(TFTLCD 接口用)
115	PD1	FSMC_D3		Y	FSMC 总线数据线 D3(TFTLCD 接口用)
116	PD2	SDIO_CMD		N	SD 卡接口的 CMD
117	PD3	OV_SCL		Y	OLED/CAMERA 接口的 SCL 信号
118	PD4	FSMC_NOE		Y	FSMC 总线 NOE(RD)(TFTLCD 接口用)
119	PD5	FSMC_NWE		Y	FSMC 总线 NWE(WR)(TFTLCD 接口用)
122	PD6	FIFO_WRST		Y	OLED/CAMERA 接口的 WRST 信号
123	PD7	RS485_RE		N	接 SP3485 芯片的 RE 引脚
77	PD8	FSMC_D13		Y	FSMC 总线数据线 D13(TFTLCD 接口用)
78	PD9	FSMC_D14		Y	FSMC 总线数据线 D14(TFTLCD 接口用)
79	PD10	FSMC_D15		Y	FSMC 总线数据线 D15(TFTLCD 接口用)
80	PD11			Y	未接任何外设
81	PD12			Y	未接任何外设
82	PD13			Y	未接任何外设
85	PD14	FSMC_D0		Y	FSMC 总线数据线 D0(TFTLCD 接口用)
86	PD15	FSMC_D1		Y	FSMC 总线数据线 D1(TFTLCD 接口用)
141	PE0			Y	未接任何外设
142	PE1			Y	未接任何外设
1	PE2			Y	未接任何外设
2	PE3	KEY1		Y	接按键 KEY1
3	PE4	KEY0		Y	接按键 KEY0
4	PE5	LED1		N	接 DS1 LED 灯(绿色)
5	PE6			Y	未接任何外设
58	PE7	FSMC_D4		Y	FSMC 总线数据线 D4(TFTLCD 接口用)
59	PE8	FSMC_D5		Y	FSMC 总线数据线 D5(TFTLCD 接口用)
60	PE9	FSMC_D6		Y	FSMC 总线数据线 D6(TFTLCD 接口用)
63	PE10	FSMC_D7		Y	FSMC 总线数据线 D7(TFTLCD 接口用)
64	PE11	FSMC_D8		Y	FSMC 总线数据线 D8(TFTLCD 接口用)
65	PE12	FSMC_D9		Y	FSMC 总线数据线 D9(TFTLCD 接口用)
66	PE13	FSMC_D10		Y	FSMC 总线数据线 D10(TFTLCD 接口用)
67	PE14	FSMC_D11		Y	FSMC 总线数据线 D11(TFTLCD 接口用)
68	PE15	FSMC_D12		Y	FSMC 总线数据线 D12(TFTLCD 接口用)
10	PF0			Y	未接任何外设
11	PF1			Y	未接任何外设
12	PF2			Y	未接任何外设
13	PF3			Y	未接任何外设
14	PF4			Y	未接任何外设
15	PF5			Y	未接任何外设

18	PF6			Y	未接任何外设
19	PF7			Y	未接任何外设
20	PF8	LIGHT_SENSOR		N	接光敏传感器(LS1)
21	PF9	T_MOSI		Y	TFTLCD 接口触摸屏 MOSI 信号
22	PF10	T_PEN		Y	TFTLCD 接口触摸屏 PEN 信号
49	PF11	T_CS		Y	TFTLCD 接口触摸屏 CS 信号
50	PF12			Y	未接任何外设
53	PF13			Y	未接任何外设
54	PF14			Y	未接任何外设
55	PF15			Y	未接任何外设
56	PG0	FSMC_A10		Y	FSMC 总线地址线 A10(TFTLCD 接口用)
57	PG1			Y	未接任何外设
87	PG2			Y	未接任何外设
88	PG3			Y	未接任何外设
89	PG4			Y	未接任何外设
90	PG5			Y	未接任何外设
91	PG6	NRF_IRQ		Y	WIRELESS 接口 IRQ 信号
92	PG7	NRF_CS		Y	WIRELESS 接口的 CS 信号
93	PG8	NRF_CE		Y	WIRELESS 接口的 CE 信号
124	PG9			Y	未接任何外设
125	PG10			Y	未接任何外设
126	PG11	1WIRE_DQ		N	单总线接口(U4)数据线，接 DHT11/DS18B20
127	PG12	FSMC_NE4		Y	FSMC 总线的片选信号 4，为 LCD 片选信号
128	PG13	OV_SDA		Y	OLED/CAMERA 接口的 SDA 脚
129	PG14	FIFO_RRST		Y	OLED/CAMERA 接口的 RRST 脚
132	PG15	FIFO_OE		Y	OLED/CAMERA 接口的 OE 脚

表 1.2.3.1 精英板 IO 资源分配总表

表 1.2.3.1 中，引脚栏即 STM32F103ZET6 的引脚编号；GPIO 栏则表示 GPIO；连接资源栏表示了对应 GPIO 所连接到的网络；独立栏，表示该 IO 是否可以完全独立（不接其他任何外设和上下拉电阻）使用，通过一定的方法，可以达到完全独立使用该 IO，Y 表示可做独立 IO，N 表示不可做独立 IO；连接关系栏，则对每个 IO 的连接做了简单的介绍。

该表在：光盘→3，ALIENTEK 精英 STM32F103 开发板原理图 文件夹下有提供 Excel 格式，并注有详细说明和使用建议，大家可以打开该表格的 Excel 版本，详细查看。

1.3 ALIENTEK 精英板与战舰板 V3.0 对比

ALIENTEK 精英 STM32F103 开发板是在战舰 STM32F103 V3.0 开发板的基础上精简而来，去掉了一些大家可能用不到的功能，减少板子尺寸，并节省成本，同时可以提供更多完全开放的 IO 口（未连接任何外设），以满足不同应用场合需求。

精英板与战舰板，主要区别，如表 1.3.1 所示：

编号	对比项	精英板 VS 战舰板 V3		说明
		精英板	战舰板 V3	
1	XM8A51216（外部 SRAM）	无	有	精简

2	10/100M 以太网	无	有	精简
3	VS1053 解码芯片	无	有	精简
4	RS232 串口	无	有 2 个	精简
5	JOYPAD 手柄接口	无	有	精简
6	板载喇叭	无	有	精简
7	亚克力保护板	无	有	精简
8	板载咪头	无	有	精简
9	板载音频接口	无	有	精简
10	普通按键	3 个	4 个	精简
11	参考电压选择端口	无	有	精简
12	PWM DAC 接口	无	有	精简
13	多功能接口	改为 AD/DA 接口	有	精简
14	HT6872 芯片	无	有	精简
15	TDA1308 芯片	无	有	精简
16	DM9000 网络芯片	无	有	精简
17	SP3232 芯片	无	有	精简
18	独立 IO 个数[1]	85 个	35 个	增加

注 1：独立 IO，是指可以达到不受板载其他外设/电阻影响，即可以达到完全浮空状态的 IO 口数目。

表 1.3.1 精英板 VS 战舰板 V3 主要区别表

从表 1.3.1 可以看出，精英 STM32F103 开发板在战舰 STM32F103 开发板 V3 版本的基础上，精简了很多芯片和接口，因此，板子也更加小巧，降低了成本。同时，由于减少了外设数目，所以，精英板独立 IO 的个数就比战舰 V3 要多的多，更方便扩展应用。

精英板和战舰 V3 版，除了表 1.3.1 列出的区别，其他所有资源则几乎都是一模一样，相同资源的 IO 连接关系，也完全是一模一样的。这样，精英板上面的所有例程，都可以直接下载到战舰 V3 版上面使用，不用做任何修改，同样战舰板的代码，只要精英板上面有资源的，也完全是可以直接下载到精英板使用，同样无需做任何修改。这就大大提高了两款板子的互补性和通用性。比如：在功能验证的时候，可以用资源充足的战舰 V3 版来开发，而验证完毕，则可以选择精英板做批量生产，从而节省成本。

第二章 实验平台硬件资源详解

本章，我们将节将向大家详细介绍 ALIENTEK 精英 STM32F103 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：

2.1，开发板原理图详解；

2.2，开发板使用注意事项；

2.3，STM32F103 学习方法；

2.1 开发板原理图详解

2.1.1 MCU

ALIENTEK 精英 STM32 开发板选择的是 STM32F103ZETT6 作为 MCU，该芯片是 STM32F103 里面配置非常强大的了，它拥有的资源包括：64KB SRAM、512KB FLASH、2 个基本定时器、4 个通用定时器、2 个高级定时器、2 个 DMA 控制器（共 12 个通道）、3 个 SPI、2 个 IIC、5 个串口、1 个 USB、1 个 CAN、3 个 12 位 ADC、1 个 12 位 DAC、1 个 SDIO 接口、1 个 FSMC 接口以及 112 个通用 IO 口。该芯片的配置十分强悍，并且还带外部总线（FSMC）可以用来外扩 SRAM 和连接 LCD 等，通过 FSMC 驱动 LCD，可以显著提高 LCD 的刷屏速度，是 STM32F1 家族常用型号里面，最高配置的芯片了，所以我们选择了它作为我们精英板的主芯片。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：

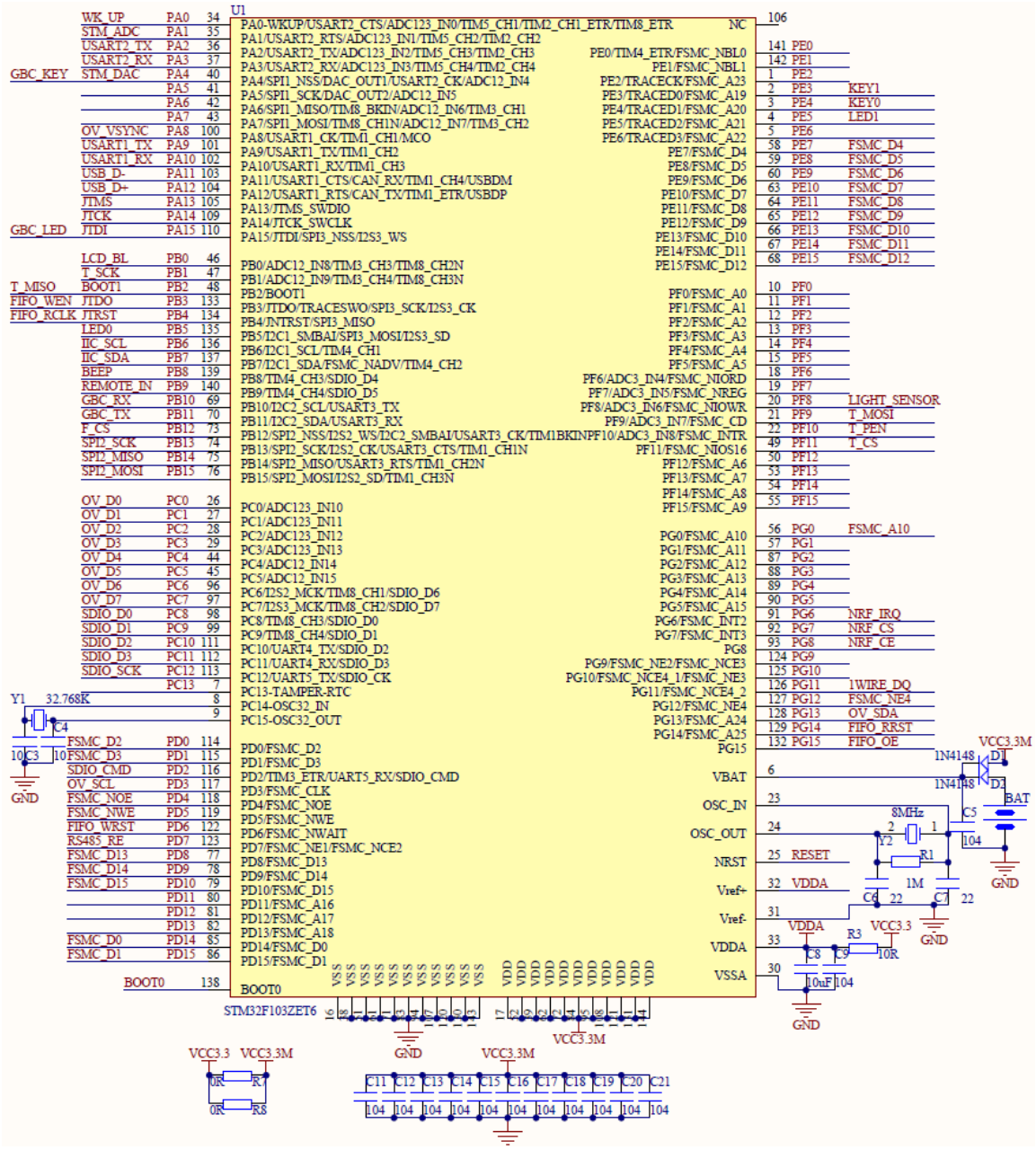


图 2.1.1.1 MCU 部分原理图

上图中 U1 为我们的主芯片：STM32F103ZET6。

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 R7 和 R8 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，开发板模拟部分（ADC/DAC）的参考电压（Vref+和 Vref-）固定接到 3.3V 和 GND，不可设置。

2.1.2 引出 IO 口

ALIENTEK 精英 STM32F103 引出了 STM32F103ZET6 的所有 IO 口，如图 2.1.2.1 所示：

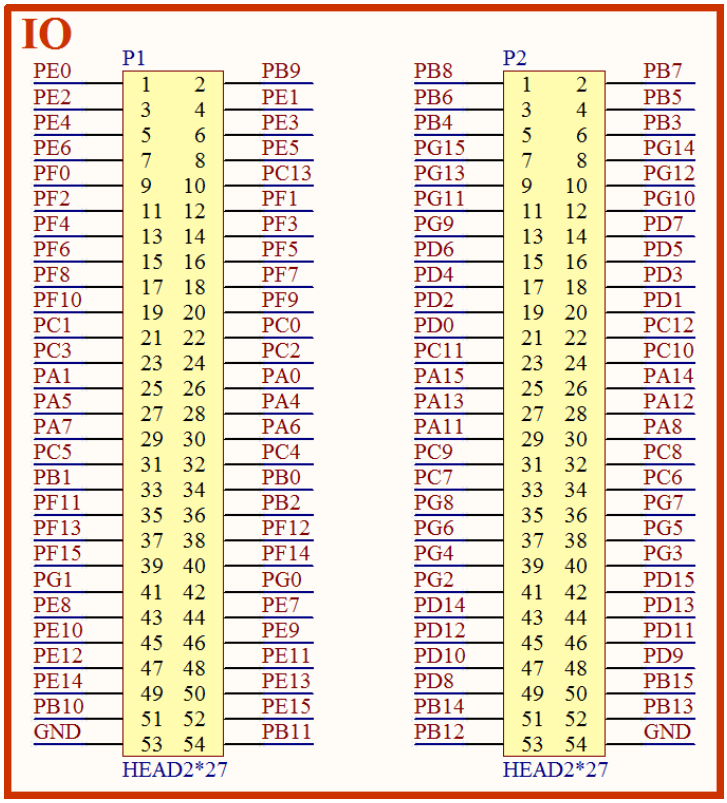


图 2.1.2.1 引出 IO 口

图中 P1 和 P2 为 MCU 主 IO 引出口，这两组排针共引出了 106 个 IO 口，STM32F103ZET6 总共有 112 个 IO，除去 RTC 晶振占用的 2 个，还剩 110 个，这两组主引出排针，总共引出了 106 个 IO，剩下的 4 个 IO 口分别通过：P3（PA9&PA10）和 P5（PA2&PA3）这 2 组排针引出。

2.1.3 USB 串口/串口 1 选择接口

ALIENTEK 精英 STM32F103 板载的 USB 串口和 STM32F103ZET6 的串口是通过 P3 连接起来的，如图 2.1.3.1 所示：

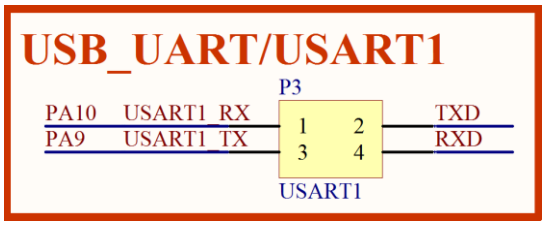


图 2.3.1.1 USB 串口/串口 1 选择接口

图中 TXD/RXD 是相对 CH340G 来说的，也就是 USB 串口的发送和接收脚。而 USART1_RX 和 USART1_TX 则是相对于 STM32F103ZET6 来说的。这样，通过对接，就可以实现 USB 串口和 STM32F103ZET6 的串口通信了。同时，P3 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 TTL 串口用了。

2.1.4 JTAG/SWD

ALIENTEK 精英 STM32F103 板载的标准 20 针 JTAG/SWD 接口电路如图 2.1.4.1 所示：

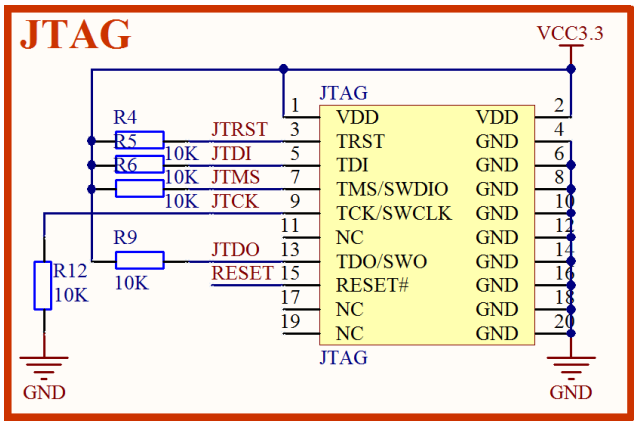


图 2.1.4.1 JTAG/SWD 接口

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 LCD 模块接口

ALIENTEK 精英 STM32F103 板载的 LCD 模块接口电路如图 2.1.5.1 所示：

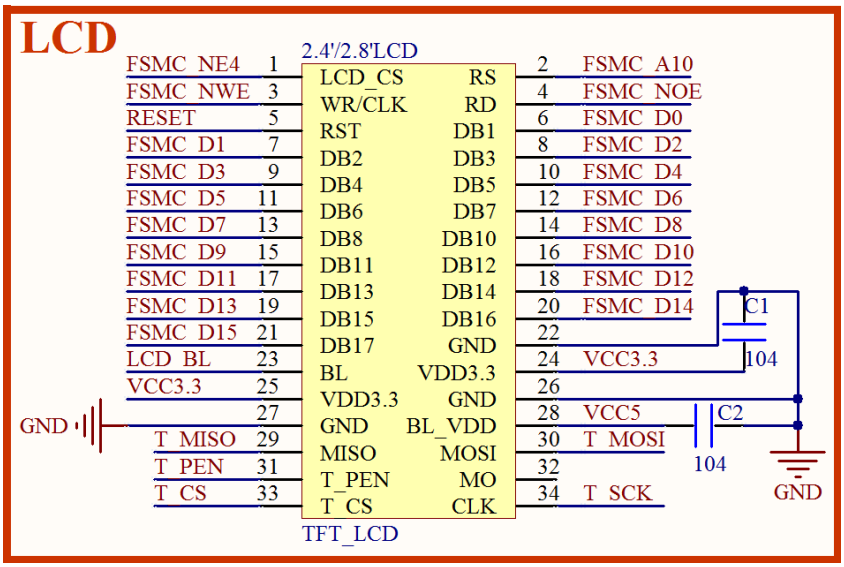


图 2.1.5.1 LCD 模块接口

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 STM32F103ZET6 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_SCK/T_CS 连接在 MCU 的 PB2/PF9/PF10/PB1/PF11 上，这些信号用来实现对液晶触摸屏的控制（支持电阻屏和电容屏）。LCD_BL 连接在 MCU 的 PB0 上，用于控制 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。

2.1.6 复位电路

ALIENTEK 精英 STM32F103 的复位电路如图 2.1.6.1 所示：

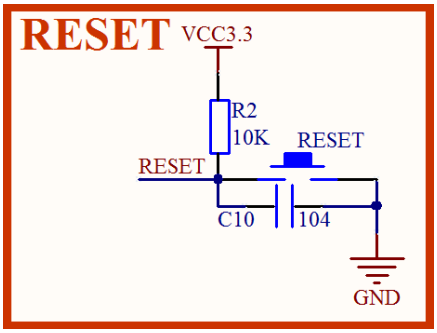


图 2.1.6.1 复位电路

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 R2 和 C10 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。

2.1.7 启动模式设置接口

ALIENTEK 精英 STM32F103 的启动模式设置端口电路如图 2.1.7.1 所示：

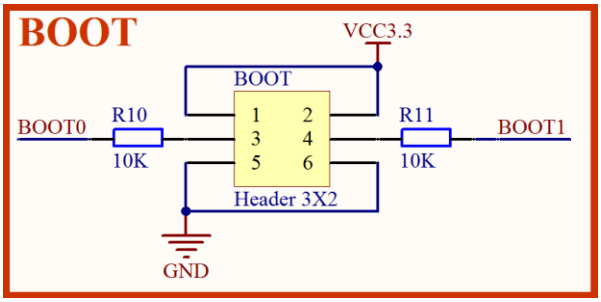


图 2.1.7.1 启动模式设置接口

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 2.1.7.1 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 2.1.7.1 BOOT0、BOOT1 启动模式表

按照表 2. 1.7.1，一般情况下如果我们想用用串口下载代码，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 精英 STM32F103 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。

2.1.8 R485 选择接口

ALIENTEK 精英 STM32F103 板载了 RS485 接口，可以通过 RS485 选择接口，连接在开发板的串口 2 上面，RS485 选择即可原理图如图 2.1.8.1 所示：

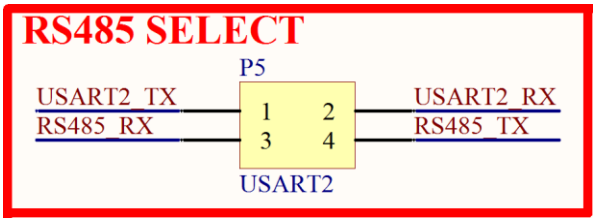


图 2.1.8.1 RS485 选择接口

图中，USART2_RX 和 USART2_TX 分别连接在 STM32F103ZET6 的 PA3 和 PA2 上面，RS485_TX 和 RS485_RX 则分别连接在 SP3485 的 RO 和 DI 引脚。当我们用 2 个跳线帽短接 P5 的 1 和 3、2 和 4，即可实现 RS485 接口连接在 STM32 的串口 2 上面，完成 RS485 通信。当拔了这两个跳线帽的时候，该接口可以实现 2 个功能：1，作为 PA2 和 PA3 的引出 IO 口；2，开发板的 RS485 部分，作为 RS485 转 TTL 模块使用。

2.1.9 RS485 接口

ALIENTEK 精英 STM32F103 板载的 RS485 接口电路如图 2.1.9.1 所示：

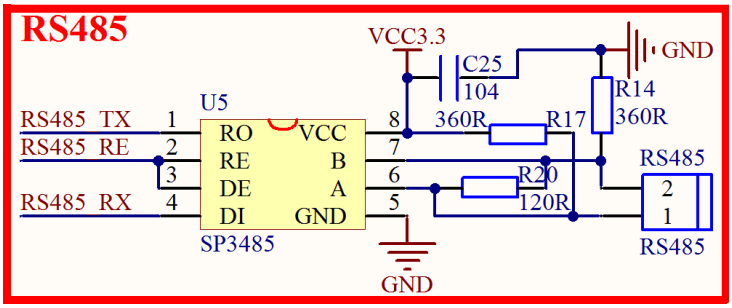


图 2.1.9.1 RS485 接口

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 R20 为终端匹配电阻，而 R14 和 R17，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 P5 上面，通过 P5 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（PD7）上的，该信号用来控制 SP3485 的工作模式（高电平为发送模式，低电平为接收模式）。

2.1.10 CAN/USB 接口

ALIENTEK 精英 STM32F103 板载的 CAN 接口电路以及 STM32 USB 接口电路如图 2.1.10.1 所示：

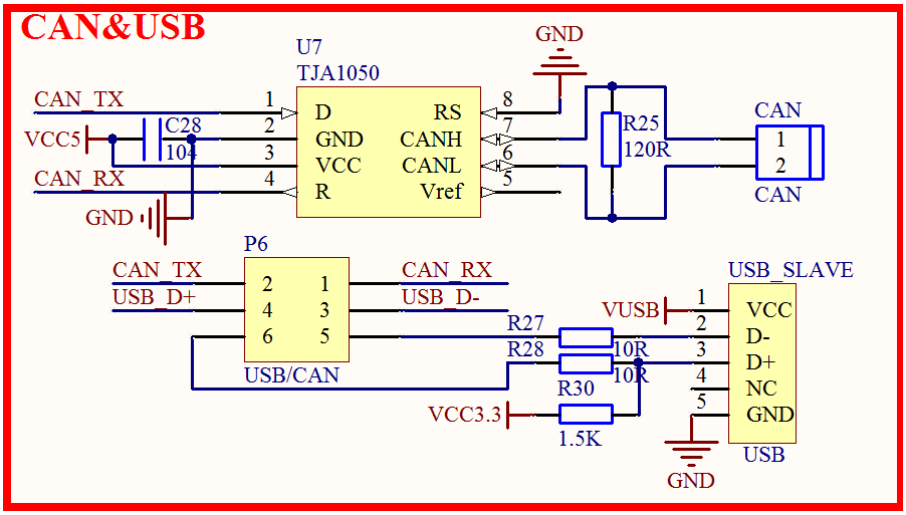


图 2.1.10.1 CAN/USB 接口

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 R25 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 P6 来选择使用 USB 还是 CAN。

USB_SLAVE 可以用来连接电脑，实现 USB 读卡器或 USB 虚拟串口等 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。

2.1.11 EEPROM

ALIENTEK 精英 STM32F103 板载的 EEPROM 电路如图 2.1.11.1 所示：

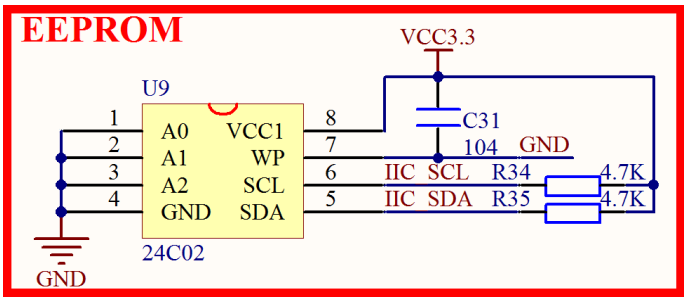


图 2.1.11.1 EEPROM

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了的。当然，你也可以选择换大容量的芯片，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 PB6 上，IIC_SDA 接在 MCU 的 PB7 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。

2.1.12 光敏传感器

ALIENTEK 精英 STM32F103 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 2.1.12.1 所示：

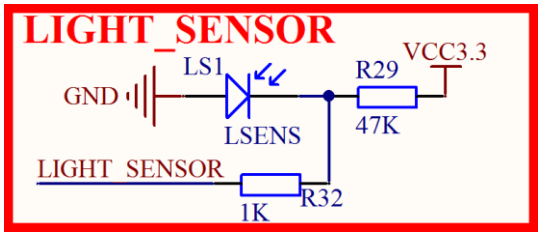


图 2.1.12.1 光敏传感器电路

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_IN6（ADC3 通道 6）上面，即 PF8 引脚。

2.1.13 SPI FLASH

ALIENTEK 精英 STM32F103 板载的 SPI FLASH 电路如图 2.1.13.1 所示：

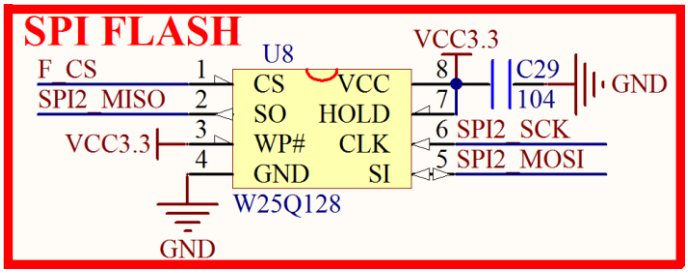


图 2.1.13.1 SPI FLASH 芯片

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（SPI2），通过片选来选择使用某个器件，在使用其中一个器件的时候，请务必禁止另外一个器件的片选信号。

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

2.1.14 温湿度传感器接口

ALIENTEK 精英 STM32F103 板载的温湿度传感器接口电路如图 2.1.14.1 所示：

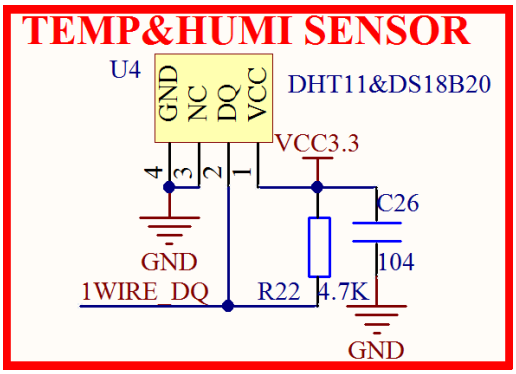


图 2.1.14.1 温湿度传感器接口

该接口（U4）支持 DS18B20/DS1820/DHT11 等单总线数字温湿度传感器。1WIRE_DQ 是传感器的数据线，该信号连接在 MCU 的 PG11 上。

2.1.15 红外接收头

ALIENTEK 精英 STM32F103 板载的红外接收头电路如图 2.1.15.1 所示：

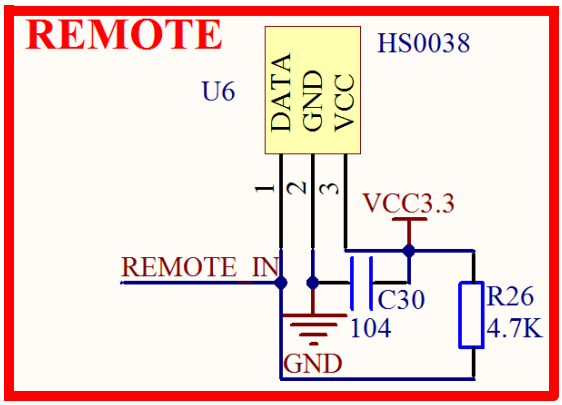


图 2.1.15.1 红外接收头

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

2.1.16 无线模块接口

ALIENTEK 精英 STM32F103 板载的无线模块接口电路如图 2.1.16.1 所示：

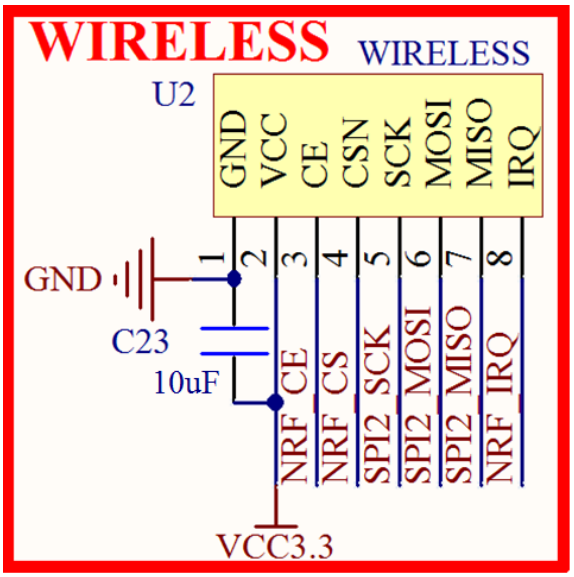


图 2.1.16.1 无线模块接口

该接口用来连接 NRF24L01 或者 RFID 等无线模块，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 PG8/PG7/PG6 上，而另外 3 个 SPI 信号则和 SPI FLASH 共用，接 MCU 的 SPI2。

2.1.17 LED

ALIENTEK 精英 STM32F103 板载总共有 3 个 LED，其原理图如图 2.1.17.1 所示：

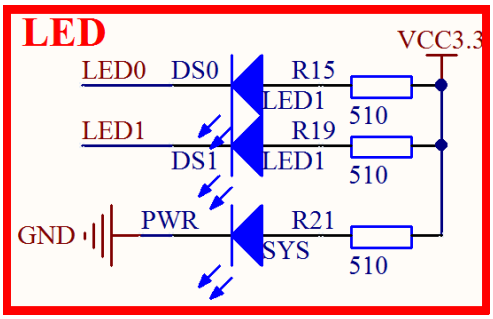


图 2.1.17.1 LED

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

2.1.18 按键

ALIENTEK 精英 STM32F103 板载总共有 3 个输入按键，其原理图如图 2.1.18.1 所示：

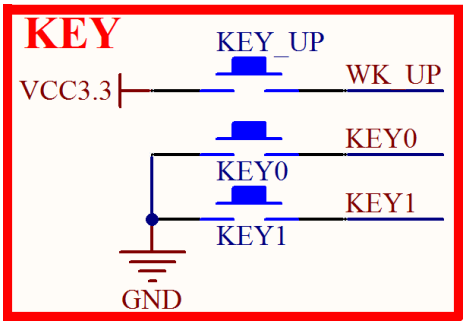


图 2.1.18.1 输入按键

KEY0 和 KEY1 用作普通按键输入，分别连接在 PE4 和 PE3 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。

2.1.19 TPAD 电容触摸按键

ALIENTEK 精英 STM32F103 板载了一个电容触摸按键，其原理图如图 2.1.19.1 所示：

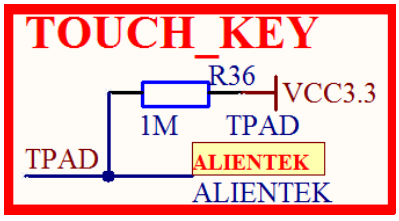


图 2.1.19.1 电容触摸按键

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 AD/DA 组合端口（P7）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 2.1.24 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。

2.1.20 OLED/摄像头模块接口

ALIENTEK 精英 STM32F103 板载了一个 OLED/摄像头模块接口，其原理图如图 2.1.20.1 所示：

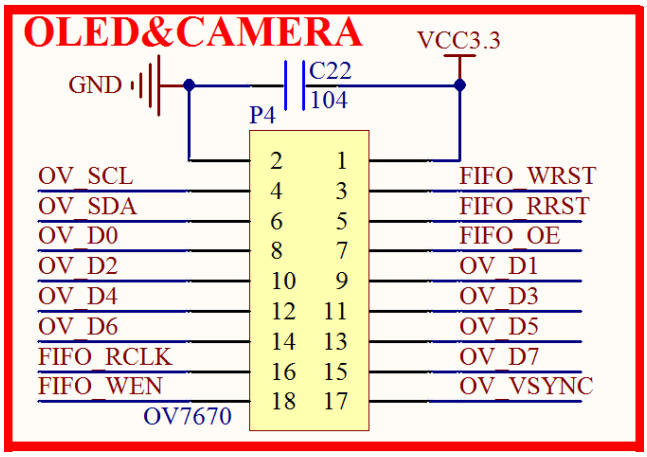


图 2.1.20.1 OLED/摄像头模块接口

图中 P4 是接口可以用来连接 ALIENTEK OLED 模块或者 ALIENTEK 摄像头模块。如果是 OLED 模块，则 FIFO_WEN 和 OV_VSYNC 不需要接（在板上靠左插即可），如果是摄像头模块，则需要用到全部引脚。

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

2.1.21 有源蜂鸣器

ALIENTEK 精英 STM32F103 板载了一个有源蜂鸣器，其原理图如图 2.1.21.1 所示：

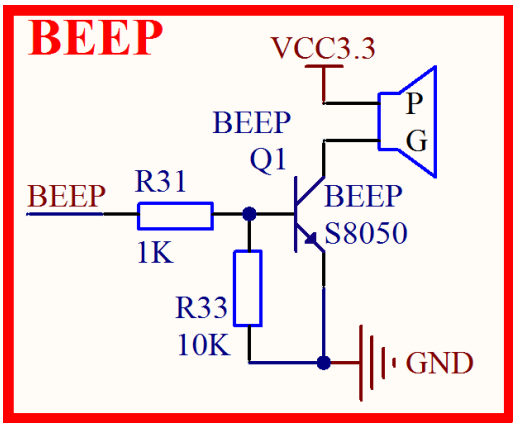


图 2.1.21.1 有源蜂鸣器

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，R33 则是一个下拉电阻，避免 MCU 复位的时候，蜂鸣器可能发声的现象。BEEP 信号直接连接在 MCU 的 PB8 上面，PB8 可以做 PWM 输出，所以大家如果想玩高级点（如：控制蜂鸣器“唱歌”），就可以使用 PWM 来控制蜂鸣器。

2.1.22 SD 卡接口

ALIENTEK 精英 STM32F103 板载了一个 SD 卡（大卡/相机卡）接口，其原理图如图 2.1.22.1 所示：

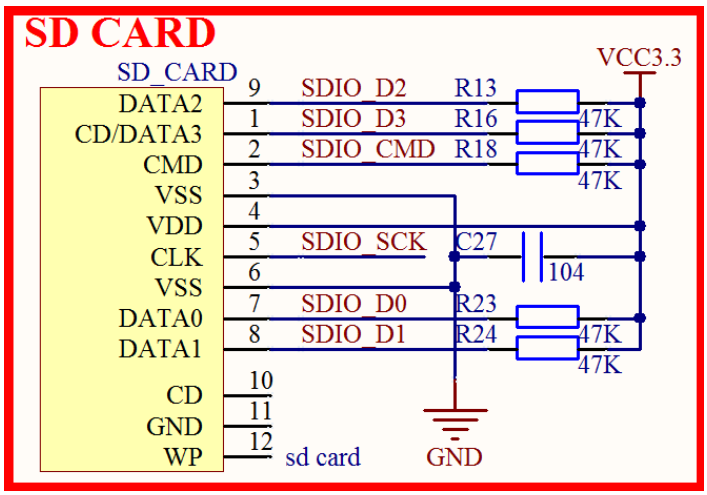


图 2.1.22.1 SD 卡接口

图中 SD_CARD 为 SD 卡接口，该接口在开发板的底面。

SD 卡采用 4 位 SDIO 方式驱动，理论上最大速度可以达到 12MB/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。

2.1.23 ATK 模块接口

ALIENTEK 精英 STM32F103 板载了 ATK 模块接口，其原理图如图 2.1.23.1 所示：

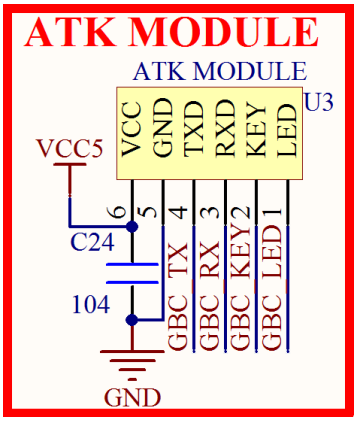


图 2.1.23.1 ATK 模块接口

如图所示，U3 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、MPU6050 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 分别连接在 MCU 的 PB11/PB10（即串口 3）上面，而 GBC_KEY 和 GBC_LED 则分别连接在 MCU 的 PA4 和 PA15 上面。**特别注意：**GBC_KEY 与 STM_DAC

共用 PA4，GBC_LED 和 JTDI 共用 PA15，在使用的时候，要注意这个问题。

2.1.24 AD/DA 组合接口

ALIENTEK 精英 STM32F103 板载了一个 AD/DA 组合接口，即 P7 接口，其原理图如图 2.1.24.1 所示：

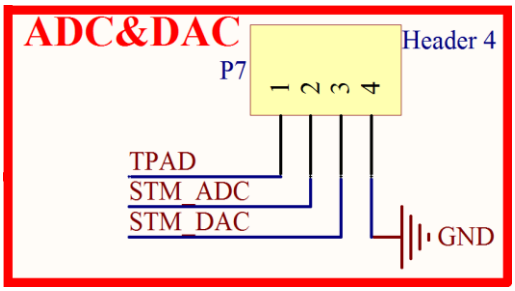


图 2.1.24.1 AD/DA 组合接口

相对于战舰板的多功能接口，精英板的 AD/DA 组合接口功能相对简单一些了。其中 TPAD 为电容触摸按键信号，连接在电容触摸按键上。STM_ADC 和 STM_DAC 则分别连接在 PA1 和 PA4 上，用于 ADC 采集或 DAC 输出。当需要电容触摸按键的时候，我们通过跳线帽短接 TPAD 和 STM_ADC（默认是连接的），就可以实现电容触摸按键（利用定时器的输入捕获）。另外，STM_ADC 还可以用与 AD 采集。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 GBC_KEY 共用 PA4，所以他们不可以同时使用，但是可以分时复用。

2.1.25 电源

ALIENTEK 精英 STM32F103 板载的电源供电部分，其原理图如图 2.1.25.1 所示：

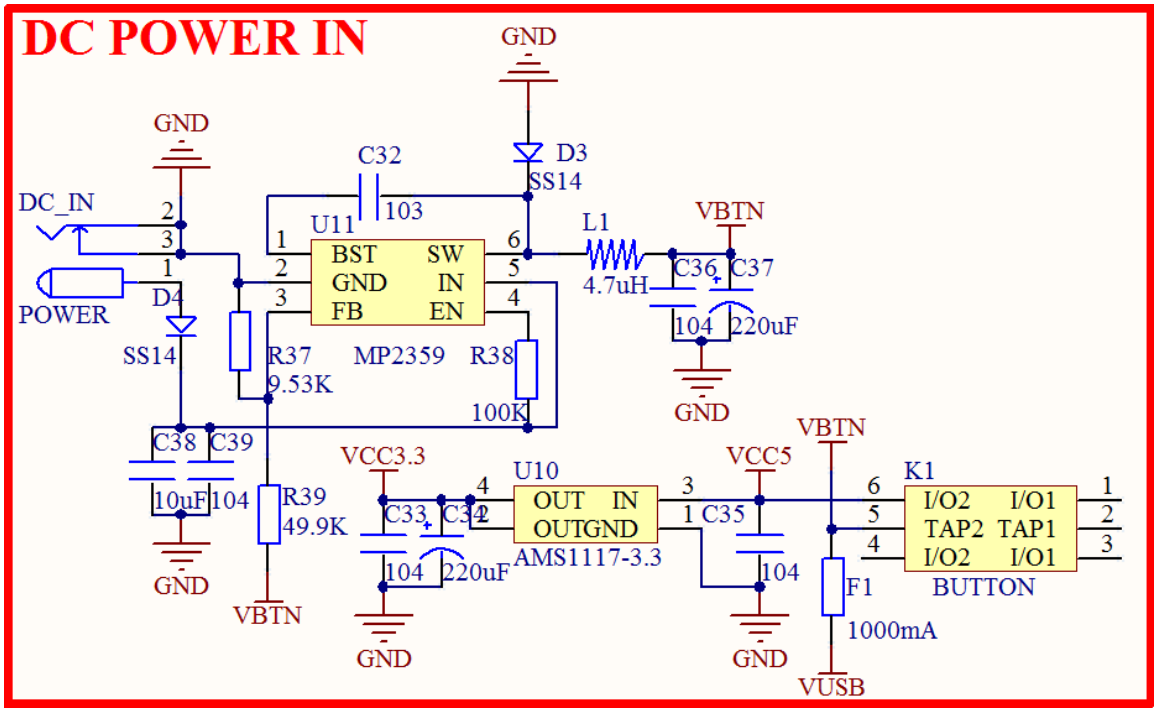


图 2.1.25.1 电源

图中，总共有 2 个稳压芯片：U10/U11，DC_IN 用于外部直流电源输入，范围是 DC6~24V，

输入电压经过 U11 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。U10 为 3.3V 稳压芯片，给开发板提供 3.3V 电源。

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 2.1.27 节进行介绍。

2.1.26 电源输入输出接口

ALIENTEK 精英 STM32F103 板载了两组简单电源输入输出接口，其原理图如图 2.1.26.1 所示：

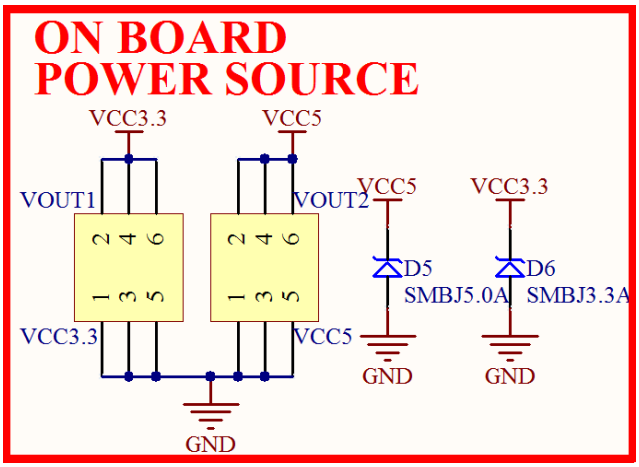


图 2.1.26.1 电源

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。

2.1.27 USB 串口

ALIENTEK 精英 STM32F103 板载了一个 USB 串口，其原理图如图 2.1.27.1 所示：

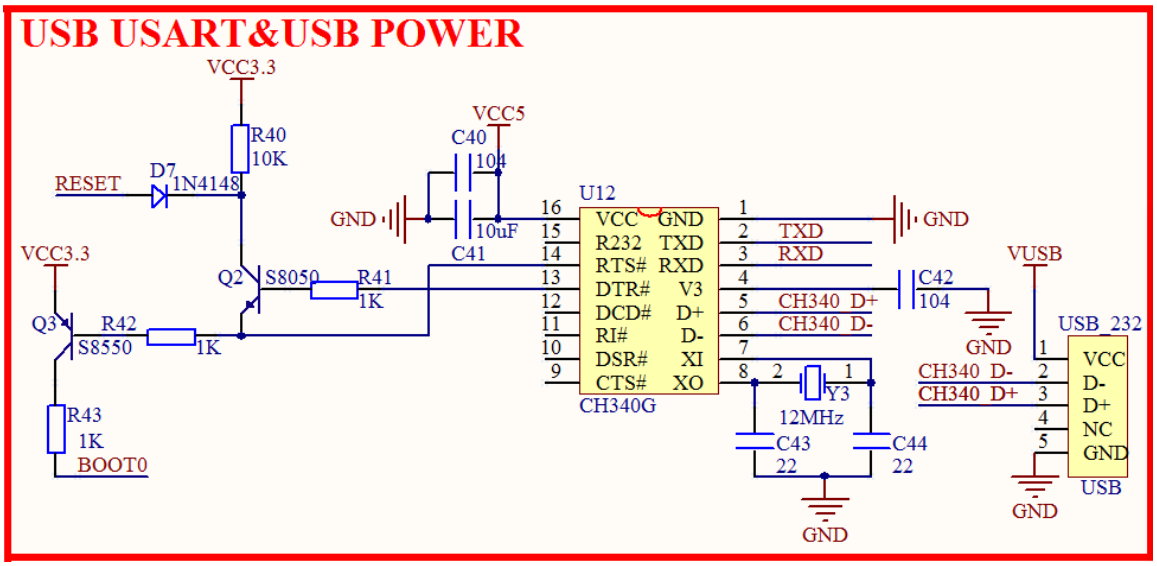


图 2.1.27.1 USB 串口

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 Q2 和 Q3 的组合构成了我们开发板的一键下载电路，**只需要在 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 Q3 导通了，BOOT0 被拉高，即实现设置 BOOT0 为 1，同时 Q2 也会导通，STM32F1 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 STM32F1 的复位引脚，由于 Q2 不再导通，变为高电平，STM32F1 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 STM32F1，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 精英 STM32F103，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、7 寸屏、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/7 寸屏的朋友，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 GBC_LED（ATK MODULE）/ FIFO_WEN（摄像头模块）/ FIFO_RCLK（摄像头模块）占用了，所以在调试这些模块的时候，

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 PB8 就不怎么适合再用做其他输出，因为他接了蜂鸣器，如果你输出高电平就会听到蜂鸣器的叫声了。

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。

至此，本手册的实验平台（ALIENTEK 精英 STM32F103）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 STM32F103 学习方法

STM32 作为目前最热门的 ARM Cortex M3 处理器，正在被越来越多的公司选择使用。学习 STM32 的朋友也越来越多，初学者，可能会认为 STM32 很难学，以前只学过 51，或者甚至连 51 都没学过的，一看到 STM32 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 STM32，还是非常简单的，这里我们总结学习 STM32 的几个要点：

1，一款实用的开发板。

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

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

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

3，掌握方法，勤学慎思。

STM32 不是妖魔鬼怪，不要畏难，STM32 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《STM32 中文参考手册_V10 版》图 8）。

任何单片机，必定是靠时钟驱动的，时钟就是单片机的动力，STM32 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，STM32 这么多寄存器，如何记得啊？回答是：不需要全部记住。我至今也就只记得 STM32 的 IO 口控制这几个寄存器，因为有规律可循，好记。其他的一概不记得。学习 STM32，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 STM32 的过程，只有进一步思考，才能更好的掌握 STM32，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 STM32 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 板块已经有近 4W 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址。

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 STM32F1 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 STM32F1 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 STM32F1 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 STM32F1 的开发流程，并能独立开始 STM32F1 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，下载与调试；

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 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，MDK5 简介；

3.2，新建 MDK5 工程；

3.3，MDK5 使用技巧；

3.1 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：MDK5.14，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 3.1.1 所示：

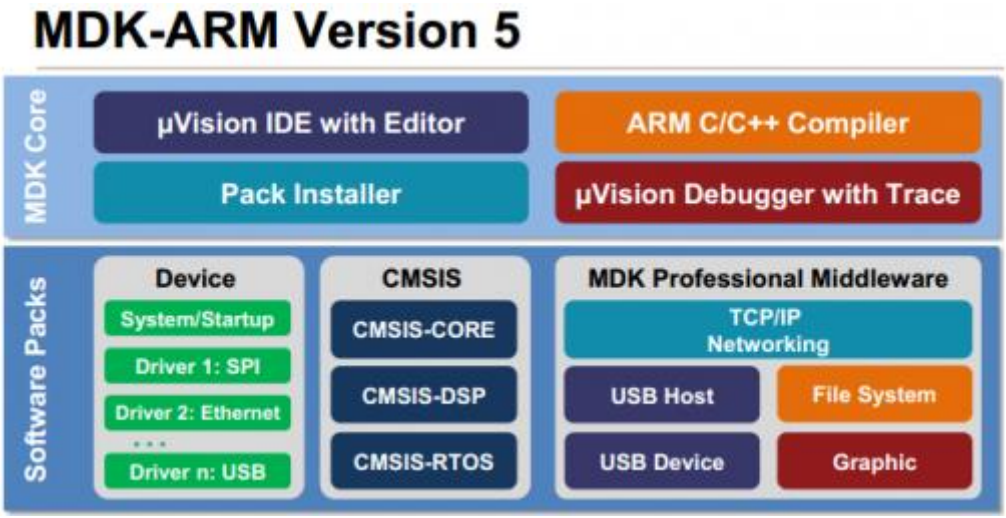


图 3.1.1 MDK5 组成

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 把所有组件到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和设备驱动等组件，但是一般都会包括 CMSIS 组件，大小 350M 左右，相对于 MDK4.70A 的 500 多 M，瘦身

不少，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。

在 MDK5 安装完成后，要让 MDK5 支持 STM32F103 的开发，我们还需要安装 STM32F1 的器件支持包：Keil.STM32F1xx_DFP.1.0.5.pack（STM32F1 的器件包）。这个包以及 MDK5.14 安装软件，我们都已经在开发板光盘提供了，大家自行安装即可。

3.2 STM32CubeF1 简介

STM32Cube 是 ST 提供的一套性能强大的免费开发工具和嵌入式软件模块，能够让开发人员在 STM32 平台上快速、轻松地开发应用。它包含两个关键部分：

1、 图形配置工具 STM32CubeMX。允许用户通过图形化向导来生成 C 语言工程。

2、 嵌入式软件包（STM32Cube 库）。包含完整的 HAL 库（STM32 硬件抽象层 API），配套的中间件（包括 RTOS，USB，TCP/IP 和图形），以及一系列完整的例程。

嵌入式软件包完全兼容 STM32CubeMX。对于图形配置工具 STM32CubeMX 入门使用，由于需要 STM32F1 基础才能入门使用，所以我们安排在后面给大家讲解。本小节，我们主要讲解 STM32Cube 的嵌入式软件包部分。在讲解之前，首先我们来看看库函数和寄存器开发的关系。

3.2.1 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 STM32 固件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->BSRR=0x00000001; //这里是针对 STM32F1 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 HAL 库封装了一个函数：

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
                       GPIO_PinState PinState)
{
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  assert_param(IS_GPIO_PIN_ACTION(PinState));
  if(PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = GPIO_Pin;
  }
```

```
    else
    {
        GPIOx->BSRR = (uint32_t)GPIO_Pin << 16;
    }
}
```

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HAL_GPIO_WritePin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。

3.2.2 STM32CubeF1 固件包介绍

STM32Cube 目前几乎支持 STM32 全系列，本手册，我们讲解的是 STM32F1 的使用，所以我们主要讲解 STM32CubeF1 相关知识。如果大家使用的是其他系列的 STM32 芯片，请到 ST 官网下载对应的 STM32Cube 包即可。完整的 STM32CubeF1 包在我们开发板配套光盘有提供，目录为：**\8，STM32 参考资料\1，STM32CubeF1 固件包**。

接下来我们看看 STM32CubeF1 包目录结构，如下图 3.2.2.1 所示：

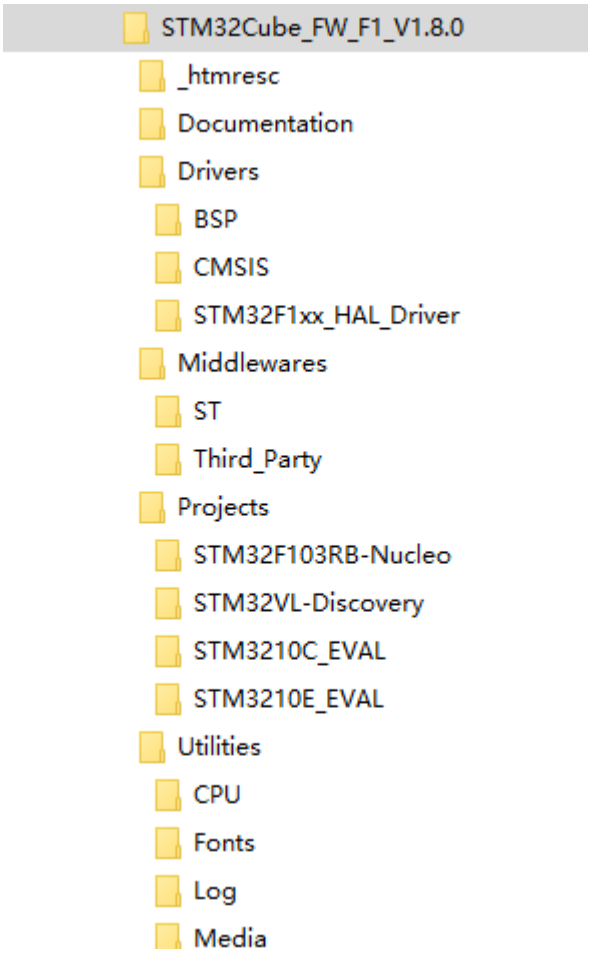


图 3.2.2.1 STM32CubeF1 包目录结构

对于 Documentation 文件夹，里面是一个 STM32CubeF1 的英文说明文档，这里我们就不做过多解释。接下来我们通过几个表格依次来介绍一下 STM32CubeF1 中几个关键的文件夹。

1）Drivers 文件夹。Drivers 文件夹包含 BSP，CMSIS 和 STM32F1xx_HAL_Driver 三个子文件夹。三个子文件夹具体说明请参考下表 3.2.2.2：

Drivers 文件夹	BSP 文件夹	也叫板级支持包，此支持包提供的是直接与硬件打交道的 API，例如触摸屏，LCD，SRAM 以及 EEPROM 等板载硬件资源等驱动。BSP 文件夹下面有多种 ST 官方 Discovery 开发板, Nucleo 开发板以及 EVAL 板的硬件驱动 API 文件，每一种开发板对应一个文件夹。
	CMSIS 文件夹	顾名思义就是符合 CMSIS 标准的软件抽象层组件相关文件。文件夹内部文件比较多。主要包括 DSP 库(DSP_LIB 文件夹)，Cortex-M 内核及其设备文件（Include 文件夹），微控制器专用头文件/启动代码/专用系统文件等(Device 文件夹)。 在我们 3.3 小节讲解新建工程的时候，会使用到这个文件夹内部很多文件，我们会在 3.3 小节对关键文件进行详细讲解。
	STM32F1xx_HAL_Driver 文件夹	这个文件夹非常重要，它包含了所有的 STM32F1xx 系列 HAL 库头文件和源文件，也就是所有底层硬件抽象层 API 声明和定义。它的作用是屏蔽了复杂的硬件寄存器操作，统一了外设的接口函数。该文件夹包含 Src 和 Inc 两个子文件夹，其中 Src 子文件夹存放的是.c 源文件，Inc 子文件夹存放的是与之对应的.h 头文件。每个.c 源文件对应一个.h 头文件。比如 gpio 相关的 API 的声明和定义在文件 stm32f1xx_hal_gpio.h 和 stm32f1xx_hal_gpio.c 中。该文件夹文件在我们新建工程章节都会使用到，我们后面会做详细介绍。

表 3.2.2.2 Drivers 文件夹介绍

2）Middlewares 文件夹。

该文件夹下面有 ST 和 Third_Party 2 个子文件夹。ST 文件夹下面存放的是 STM32 相关的一些文件，包括 STemWin 和 USB 库等。Third_Party 文件夹是第三方中间件，这些中间价都是非常成熟的开源解决方案。具体说明请见下表 3.3.2.3：

Middlewares 文件夹	ST 子文件夹	STemWin 文件夹	STemWin 工具包。Segger 提供。
		STM32_USB_Device_Library 文件夹	USB 从机设备支持包。
		STM32_USB_Host_Library 文件夹	USB 主机设备支持包。
	Third_Party 子文件夹	FatFs 文件夹	FAT 文件系统支持包。采用的 FATFS 文件系统。
		FreeRTOS 文件夹	FreeRTOS 实时系统支持包。
		LwIP 文件夹	LwIP 网络通信协议支持包。

表 3.2.2.3 Middlewares 文件夹介绍

3）Projects 文件夹。

该文件夹存放的是一些可以直接编译的实例工程。每个文件夹对应一个 ST 官方的 Demo 板。里面有很多实例，我们都可以用来参考。这里大家注意，每个工程下面都有一个 MDK-ARM 子文件夹，该子文件夹内部会有名称为 Project.uvprojx 的工程文件，我们只需要点击它就可以在 MDK 中打开工程。

4）Utilities 文件夹。

该文件夹下面是一些其他组件，在项目中使用得不多。有兴趣的同学可以学习一下，这里我们不做过多介绍。

3. 2. 3 HAL 库和标准库选择

ST 先后提供了两套固件库：标准库和 HAL 库。STM32 芯片面市之初只提供了丰富全面的标准库，大大便利了用户程序开发，为广大开发板所推崇，同时也为 ST 积累了大量标准库用户。有过 STM32 基础的同学想必对标准库非常熟悉。我们正点原子目前的所有 STM32F1 开发板以及探索者 STM32F407 开发板都有使用标准库的例程。

大约到 2014 年左右，ST 在标准库的基础上又推出了 HAL 库。实际上，HAL 库和标准库本质上是一样的，都是提供底层硬件操作 API，而且在使用上也是大同小异。有过标准库基础的同学对 HAL 库的使用也很容易入手。个人认为 ST 官方之所以这几年大力推广 HAL 库，是因为 HAL 的结构更加容易整合 STM32Cube，而 STM32CubeMX 是 ST 这几年极力推荐的程序生成开发工具。所以这几年新出的 STM32 芯片，ST 直接只提供 HAL 库。

那么有很多同学不禁要问，我们是使用 HAL 库还是标准库好呢？这里我们想说的是，HAL 库和标准库都非常强大，对于目前标准库支持的芯片采用标准库开发也非常方便实用。大家不需要纠结自己学的是 HAL 库还是标准库，无论使用哪种库，只要理解了 STM32 本质，任何库都是一种工具，使用起来都非常方便。学会了一种库，另外一种库也非常容易上手，程序开发思路转变也非常容易。如果你是一个 STM32 熟手，长期从事 STM32 开发，那么有必要对标准库和 HAL 库都有一定的了解，这样才能在项目开发中得心应手游刃有余。

3.3 新建基于 HAL 库的工程模板和工程结构讲解

在前面的章节我们介绍了 STM32F1xx 官方 HAL 库包的一些知识，这些我们将着重讲解建立基于 HAL 库的工程模板的详细步骤。在新建模板之前之前，首先我们要准备如下资料：

1) HAL 库开发包：STM32Cube_FW_F1_V1.8.0 这是 ST 官网下载的 STM32CubeF1 包完整版，我们光盘目录（压缩包）：
“\8，STM32 参考资料\1，STM32CubeF1 固件包\ en.stm32cubef1.zip”。

2) MDK5.23 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。

3.3.1 新建基于 HAL 库工程模板

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\3，标准例程-HAL 库版本\实验 0-1 Template 工程模板-新建工程章节使用**” 下面，大家在学习新建工程过程中间遇到一些问题，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 4 个子文件夹：CORE ，HALLIB，OBJ 和 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 3.3.1.1.

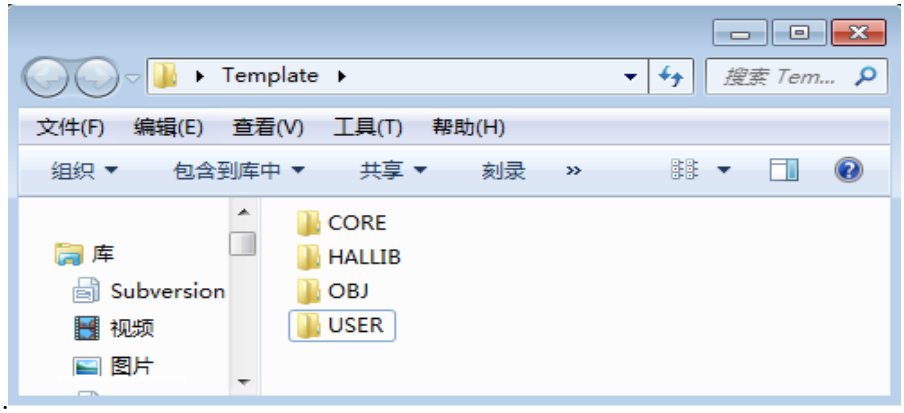


图 3.3.1.1 新建文件夹

2） 接下来，打开 MDK，点击菜单 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，工程取名为 Template 之后点击保存，工程文件就都保存到 USER 文件夹下面。 操作过程如下图 3.3.1.2 和 3.3.1.3 所示：

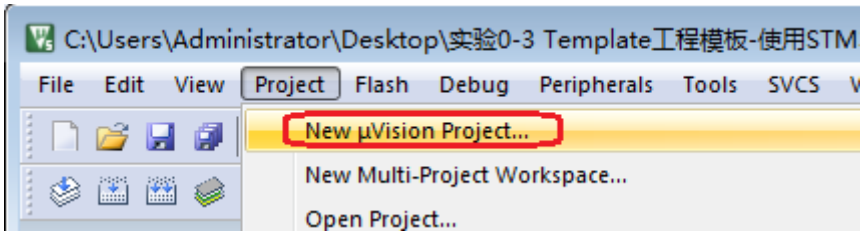


图 3.3.1.2 新建工程

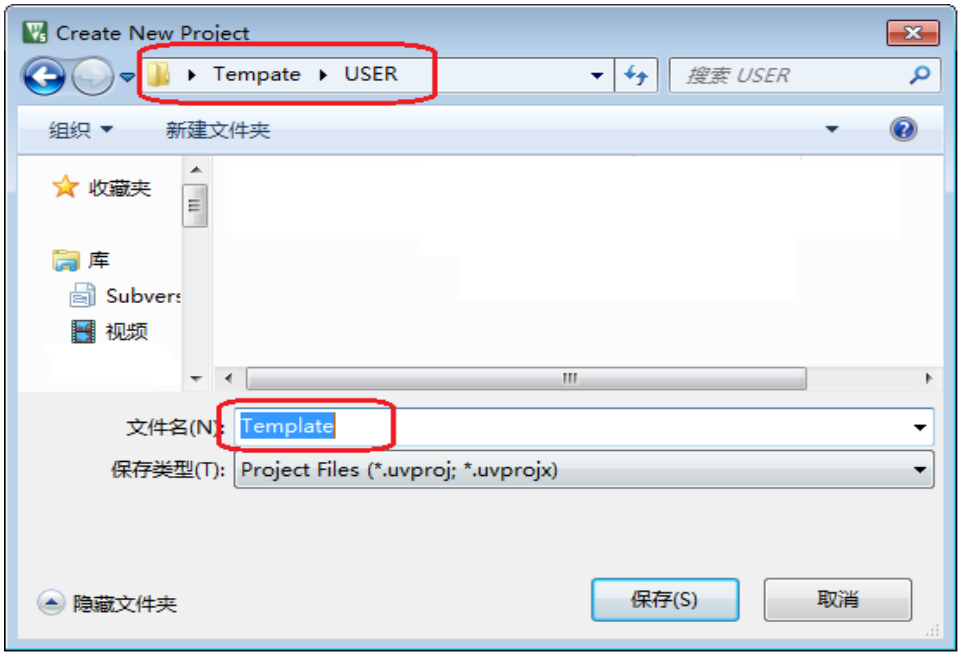


图 3.3.1.3 定义工程名称

接下来会出现一个选择 Device 的界面，就是选择我们的芯片型号，这里我们定位到

STMicroelectronics 下面的 STM32F103ZE（**针对我们的正点原子精英 STM32F103ZET6 板子是这个型号**）。这里我们选择 STMicroelectronics→STM32F1 Series→STM32F103→STM32F103ZE（如果使用的是其他系列的芯片，选择相应的型号就可以了，例如我们的探索者 STM32 开发板是 STM32F407ZG。**特别注意：一定要安装对应的器件 pack 才会显示这些内容**）。

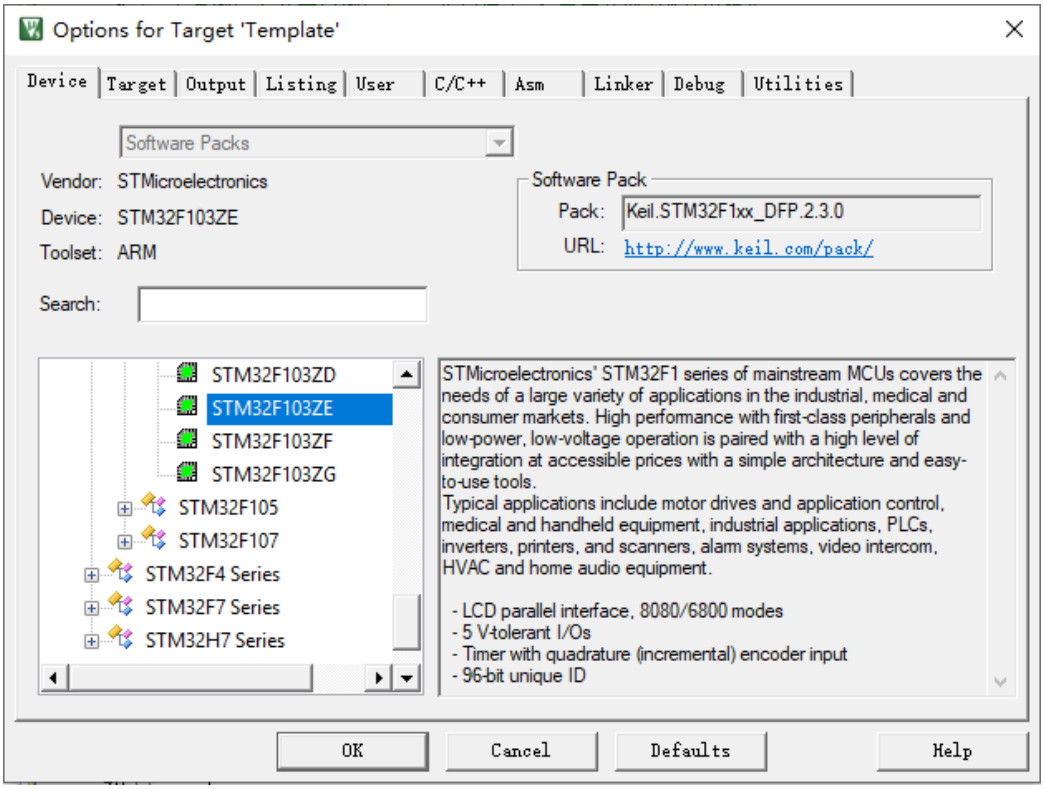


图 3.3.1.4 选择芯片型号

点击 OK，MDK 会弹出 Manage Run-Time Environment 对话框，如图 3.3.1.5 所示：

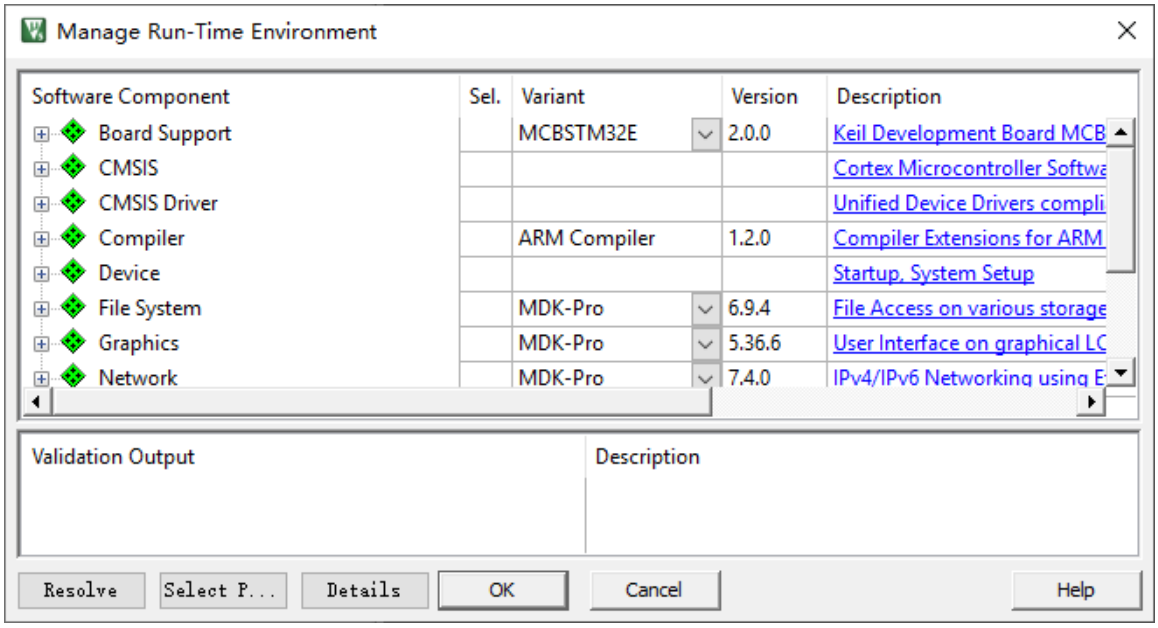


图 3.3.1.5 Manage Run-Time Environment 界面

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 3.3.1.5 所示界面，我们直接点击 Cancel，即可，

得到如图 3.3.1.6 所示界面：

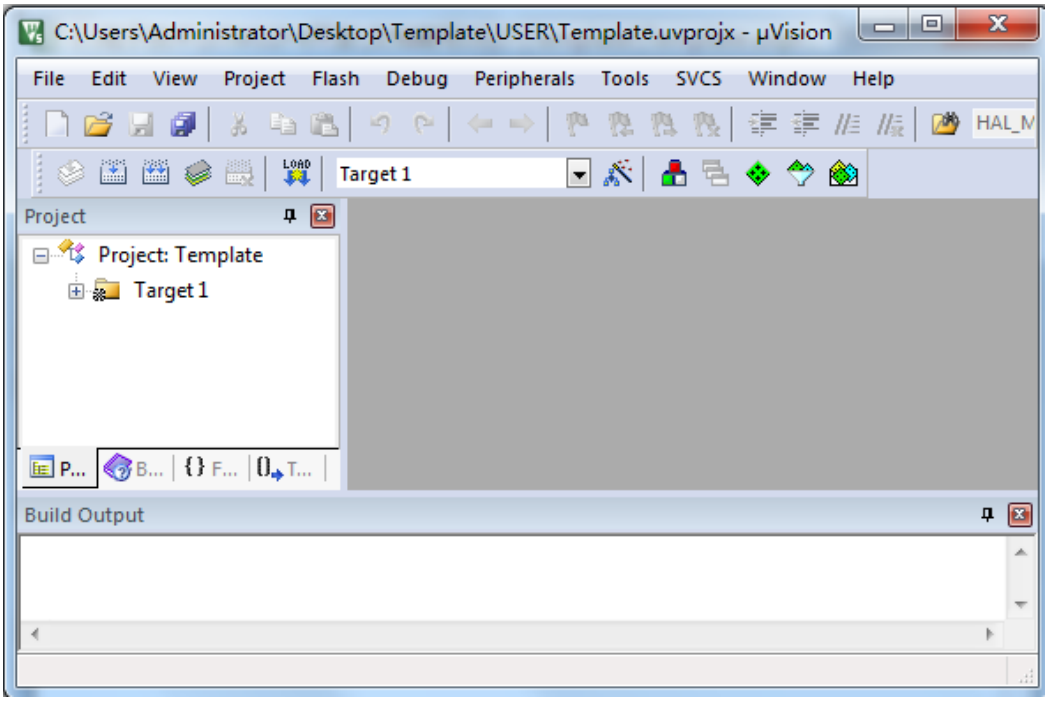


图 3.3.1.6 工程初步建立

3） 现在我们看看 USER 目录下面内容，如下图 3.3.1.7：

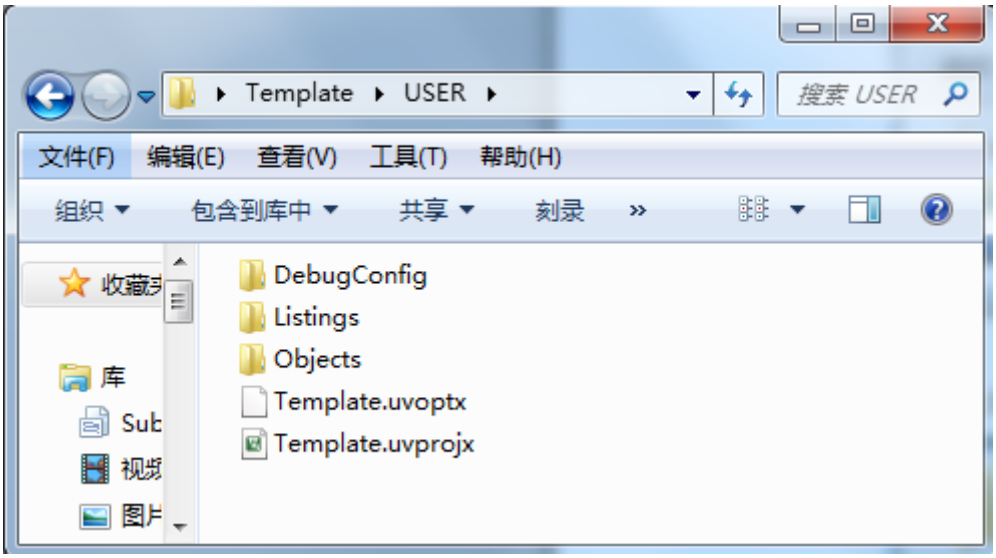


图 3.3.1.7 工程 USER 目录文件

这里我们说明一下， Template.uvprojx 是工程文件，非常关键，不能轻易删除，MDK5.23 生成的工程文件是以.uvprojx 为后缀。DebugConfig，Listings 和 Objects 三个文件夹是 MDK 自动生成的文件夹。其中 DebugConfig 文件夹用于存储一些调试配置文件，Listings 和 Objects 文件夹用来存储 MDK 编译过程的一些中间文件。这里，我们把 Listings 和 Objects 文件夹删除，我们会在下一步骤中新建一个 OBJ 文件夹，用来存放编译中间文件。当然，我们不删除这两个文件夹也没有关系，只是我们不用它而已。

4） 接下来我们将从官方 stm32cubeF1 包里面复制一些我们新建工程需要的关键文件到我们的

工程目录中。首先，我们要将 STM32CubeF1 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 STM32CubeF1 包，定位到我们之前准备好的 HAL 库包的目录:\STM32Cube_FW_F1_V1.8.0\Drivers\STM32F1xx_HAL_Driver 下面，将目录下面的 Src,Inc 文件夹复制到我们刚才建立的 HALLIB 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。操作完成后工程 HALLIB 目录内容如下图 3.3.1.8。

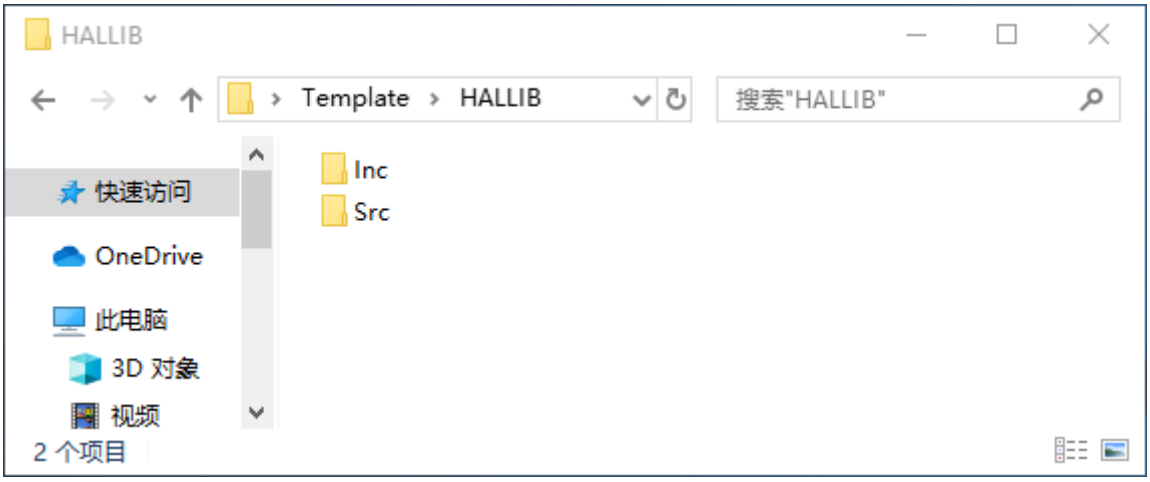


图 3.3.1.8 官方库源码文件夹

5） 接下来，我们要将 STM32CubeF1 包里面相关的启动文件以及一些关键头文件复制到我们的工程目录 CORE 之下。打开 STM32CubeF1 包，定位到目录\STM32Cube_FW_F1_V1.8.0\Drivers\CMSIS\Device\ST\STM32F1xx\Source\Templates\arm 下面，将文件 **startup_stm32f103xe.s** 复制到 CORE 目录下面。然后定位到目录\STM32Cube_FW_F1_V1.8.0\Drivers\CMSIS\Include，将里面的四个头文件：**cmsis_armcc.h，cmsis_armclang.h，cmsis_compiler.h，core_cm3.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 3.3.1.9：

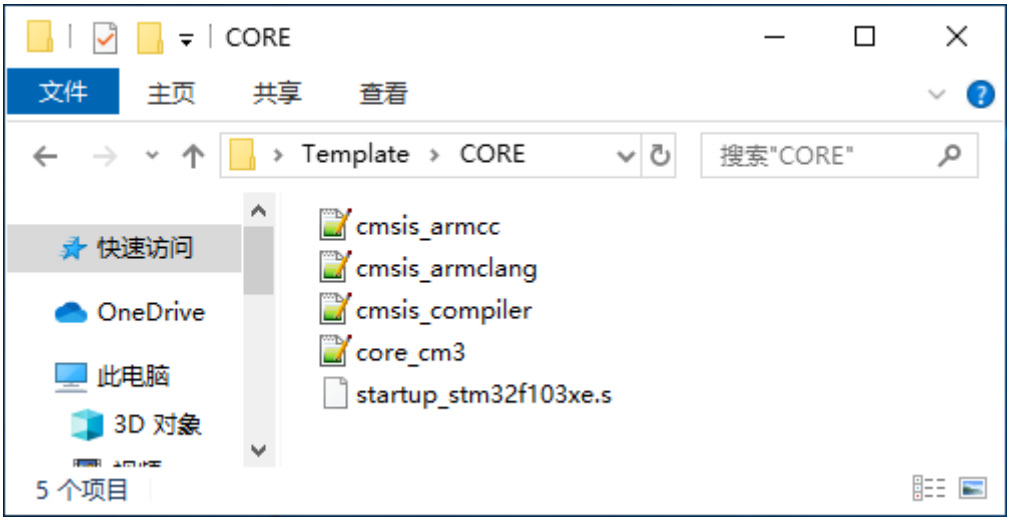


图 3.3.1.9 CORE 文件夹文件

6） 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：\STM32Cube_FW_F1_V1.8.0\Drivers\CMSIS\Device\ST\STM32F1xx\Include 将里面的 3 个文件 stm32f1xx.h，system_stm32f1xx.h 和 stm32f103xe.h 复制到 USER 目录之下。这三个头文件是

STM32F1 工程非常关键的头文件。前面我们介绍 STM32CubeF1 包的时候已经给大家介绍过。然后进入目录\STM32Cube_FW_F1_V1.8.0\Projects\STM3210E_EVAL\Templates 目录下，这个目录下面有好几个文件夹，如下图 3.3.1.10，我们需要从 Src 和 Inc 文件夹下面复制我们需要的文件到 USER 目录。

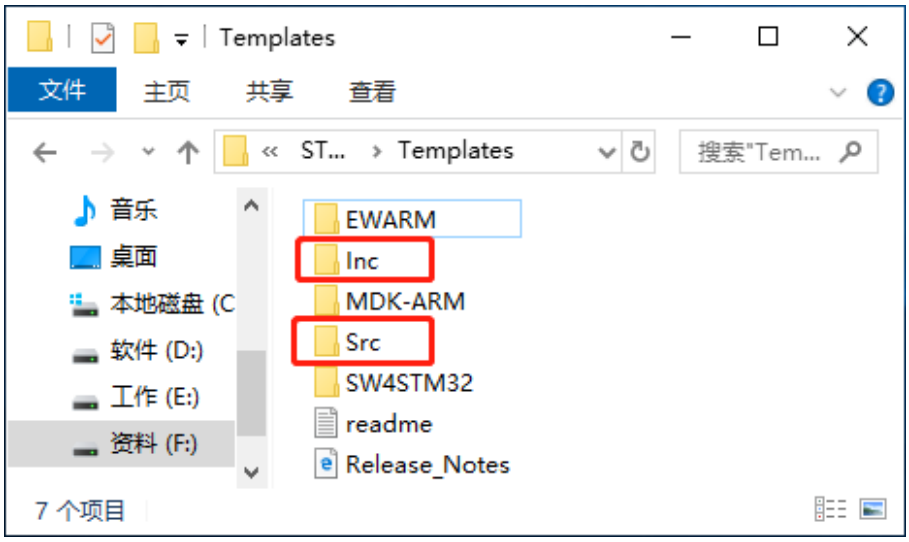


图 3.3.1.10 固件库包 Template 目录下面文件一览

首先我们打开 Inc 目录，将目录下面的 3 个头文件 stm32f1xx_it.h，stm32f1xx_hal_conf.h 和 main.h 全部复制到 USER 目录下面。然后我们打开 Src 目录，将下面的四个源文件 system_stm32f1xx.c，stm32f1xx_it.c, stm32f1xx_hal_msp.c 和 main.c 同样全部复制到 USER 目录下面。相关文件复制到 USER 目录之后 USER 目录文件如下图 3.3.1.11：

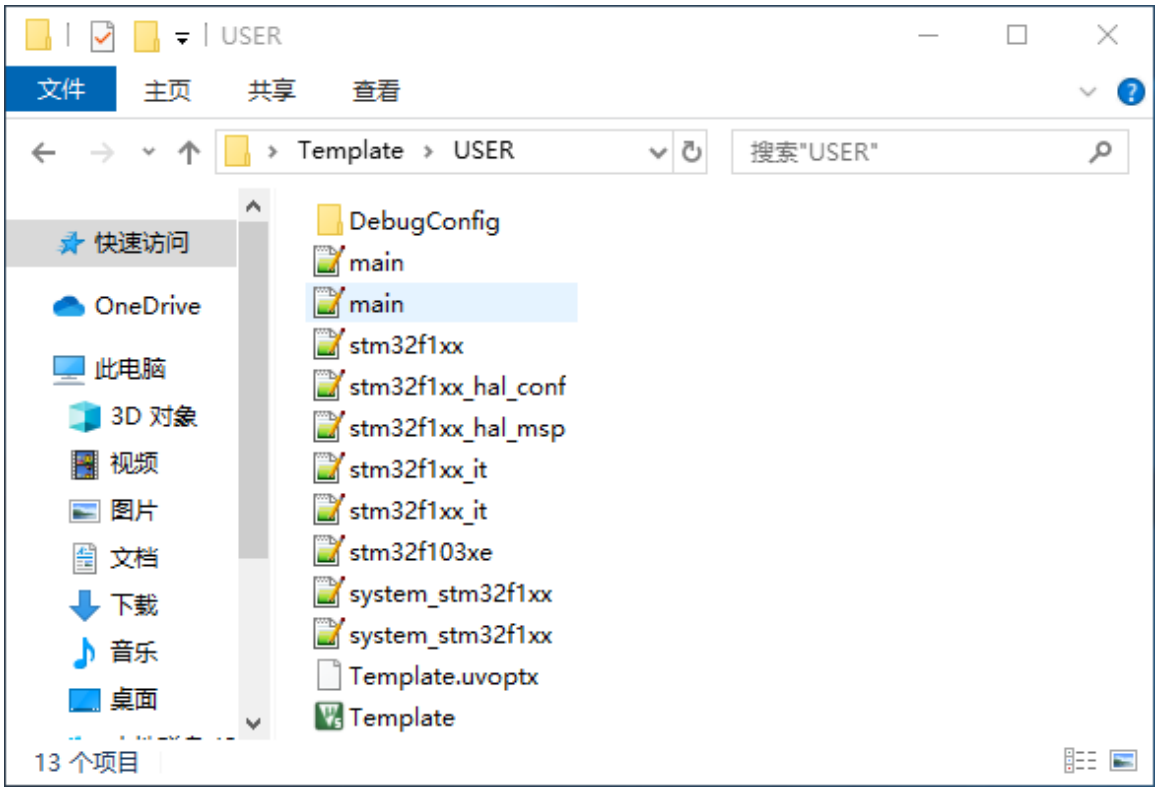


图 3.3.1.11 USER 目录文件浏览

7） 前面 6 个步骤，我们将需要的文件复制到了我们的工程目录下面了。接下来，我们还需要

复制 ALIENTEK 编写的 SYSTEM 文件夹内容到工程目录中。首先，我们需要解释一下，这个 SYSTEM 文件夹内容是 ALIENTEK 为开发板用户编写的一套非常实用的函数库，比如系统时钟初始化，串口打印，延时函数等，这些函数被很多工程师运用到自己的工程项目中。当然，大家也可以根据自己需求决定是否需要 SYSTEM 文件夹，对于 STM32F103 的工程模板，如果没有加入 SYSTEM 文件夹，那么大家需要自己定义系统时钟初始化。**SYSTEM 文件夹对于库函数版本程序和寄存器版本程序是有所区别的，这里我们新建的是 HAL 库工程模板，所以大家从光盘程序源码目录之下的 HAL 库版本的任何一个实验中复制过来即可。**这里我们打开光盘的“**4，程序源码\标准例程-库函数版本\实验 0-1 Template 工程模板-新建工程章节使用**”工程目录，从里面复制 SYSTEM 文件夹到我们的 Template 工程模板根目录即可。操作过程如下图 3.3.1.12 和图 3.3.1.13 所示：

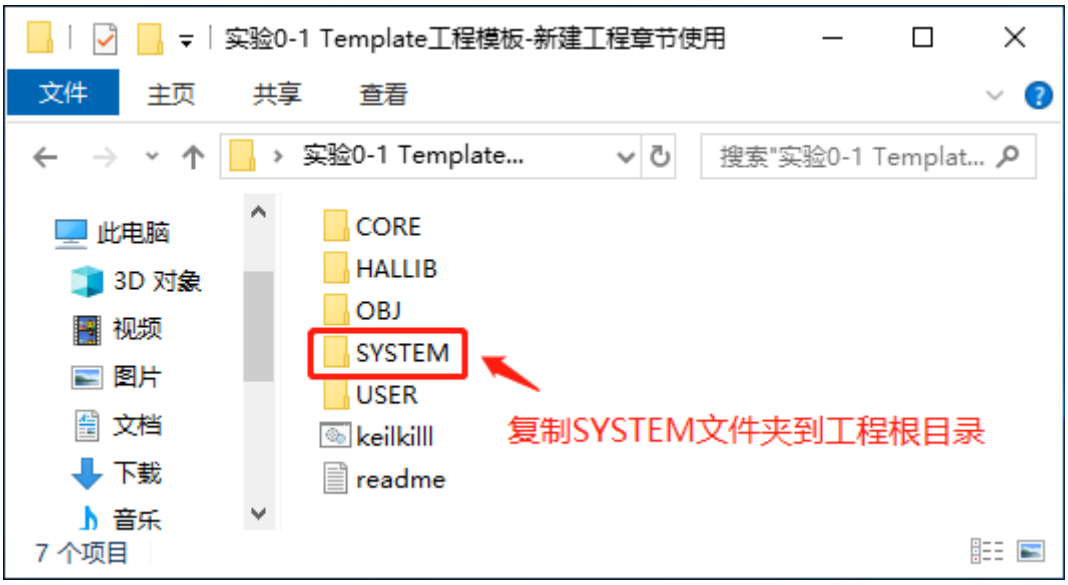


图 3.3.1.12 复制实验 0-1 的 SYSTEM 文件夹到工程根目录

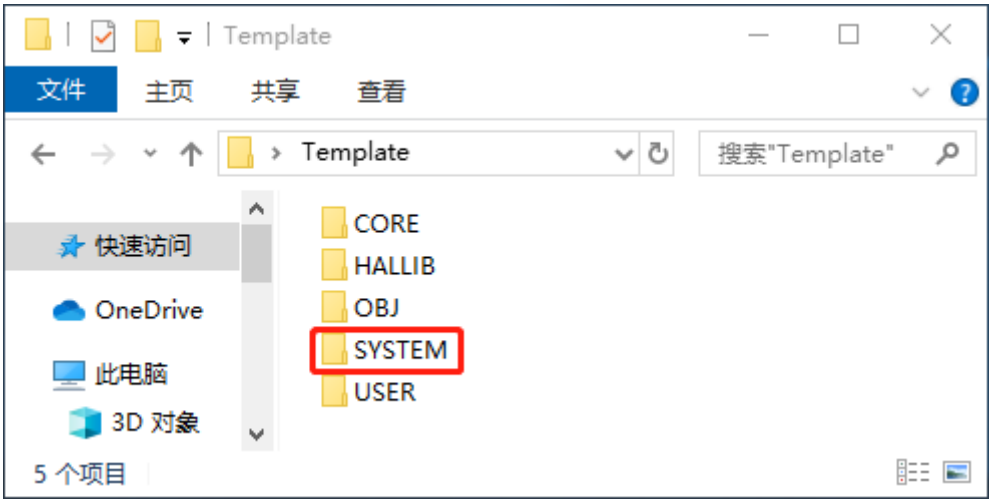


图 3.3.1.13 复制 SYSTEM 文件夹之后的 Template 根目录文件夹结构

到这里，工程模板所需要的所有文件都已经复制进去。接下来，我们将在 MDK 中将这些文件添加到工程。

8） 下面我们将前面复制过来的文件加入我们的工程中。右键点击 Target1，选择 Manage Project Items，如下图 3. 3. 1. 14 所示：

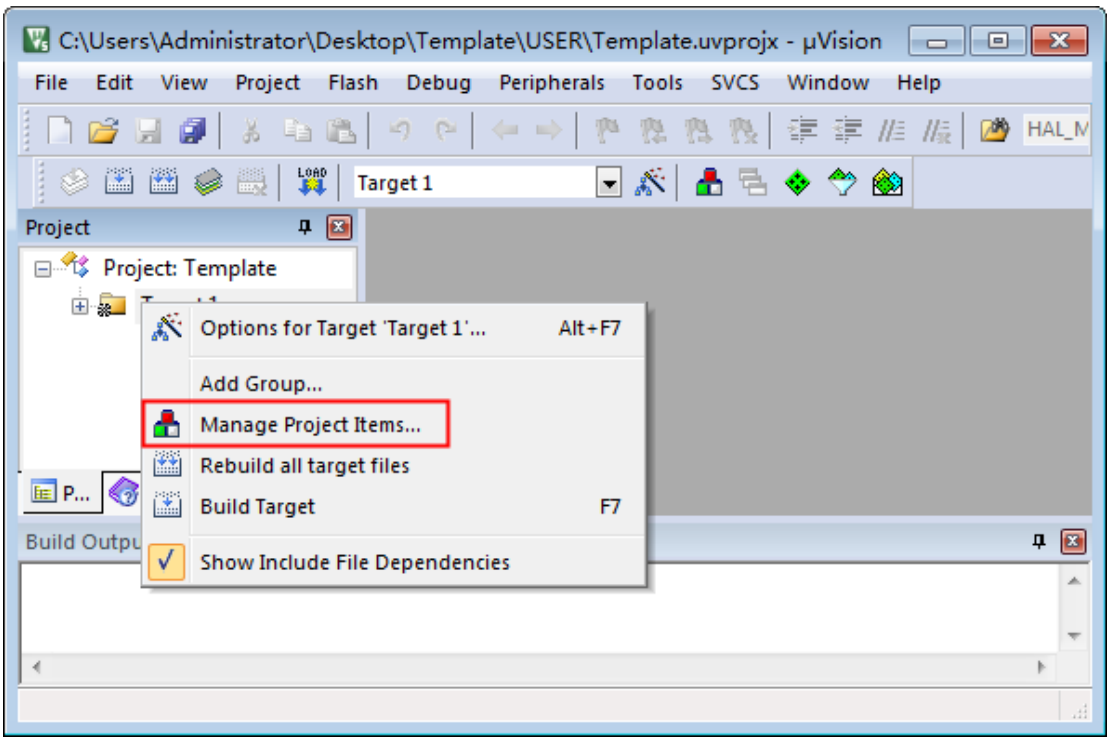


图 3.3.1.14 点击 Management Project Itmes

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，建立四个 Groups：USER，SYSTEM，CORE，和 HALLIB。然后点击 OK，可以看到我们的 Target 名字以及 Groups 情况如下图 3.3.1.15 和图 3.3.1.16 所示:

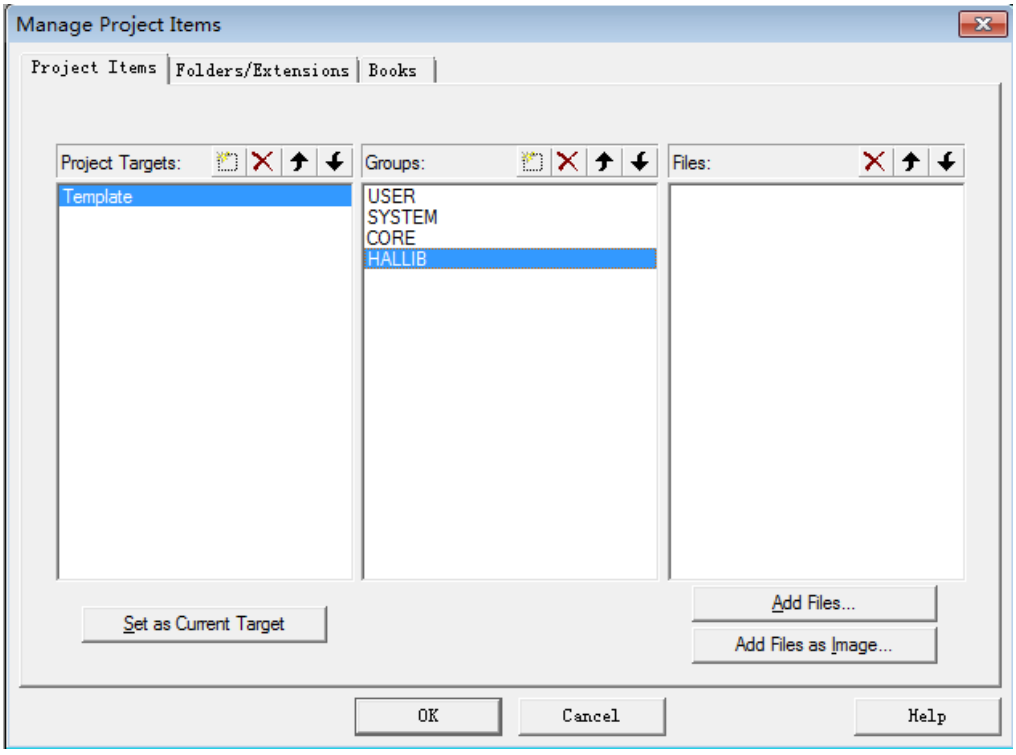


图 3.3.1.15 新建 GROUP

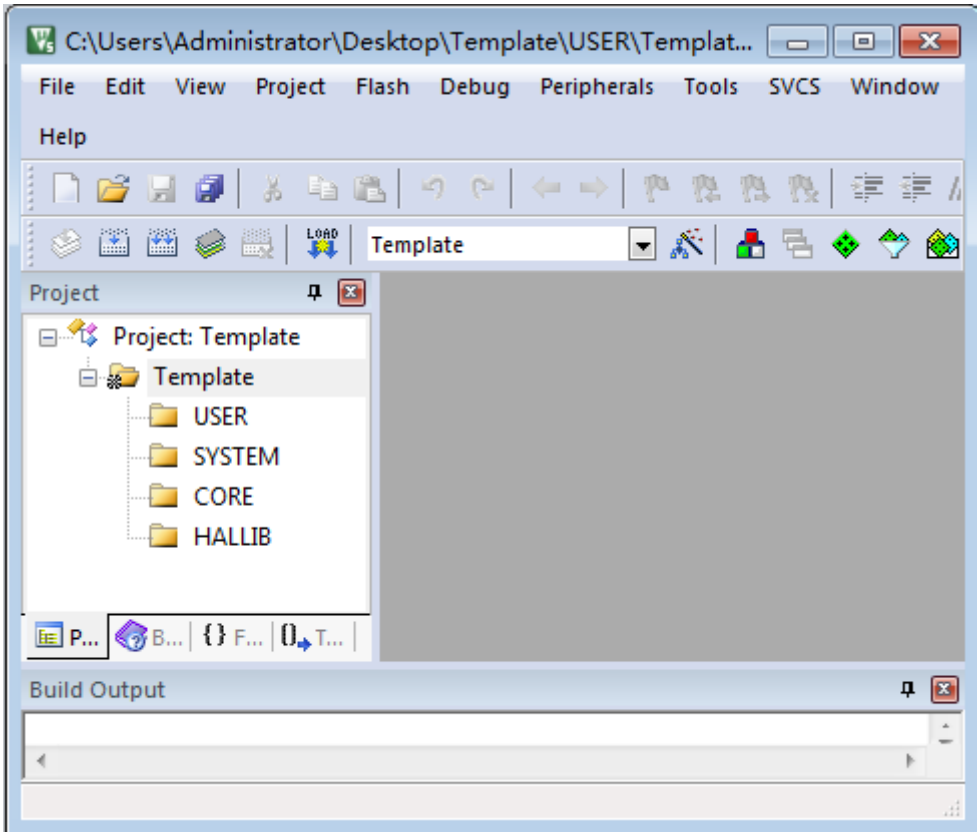


图 3.3.1.16 查看工程 Group 情况

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 Manage Project Items.然后选择需要添加文件的 Group，这里第一步我们选择 HALLIB，然后点击右边的 Add Files,定位到我们刚才建立的目录\HALLIB\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 3.3.1.17。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 stm32f1xx_gpio.c 而其他外设相关的可以不用添加。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。

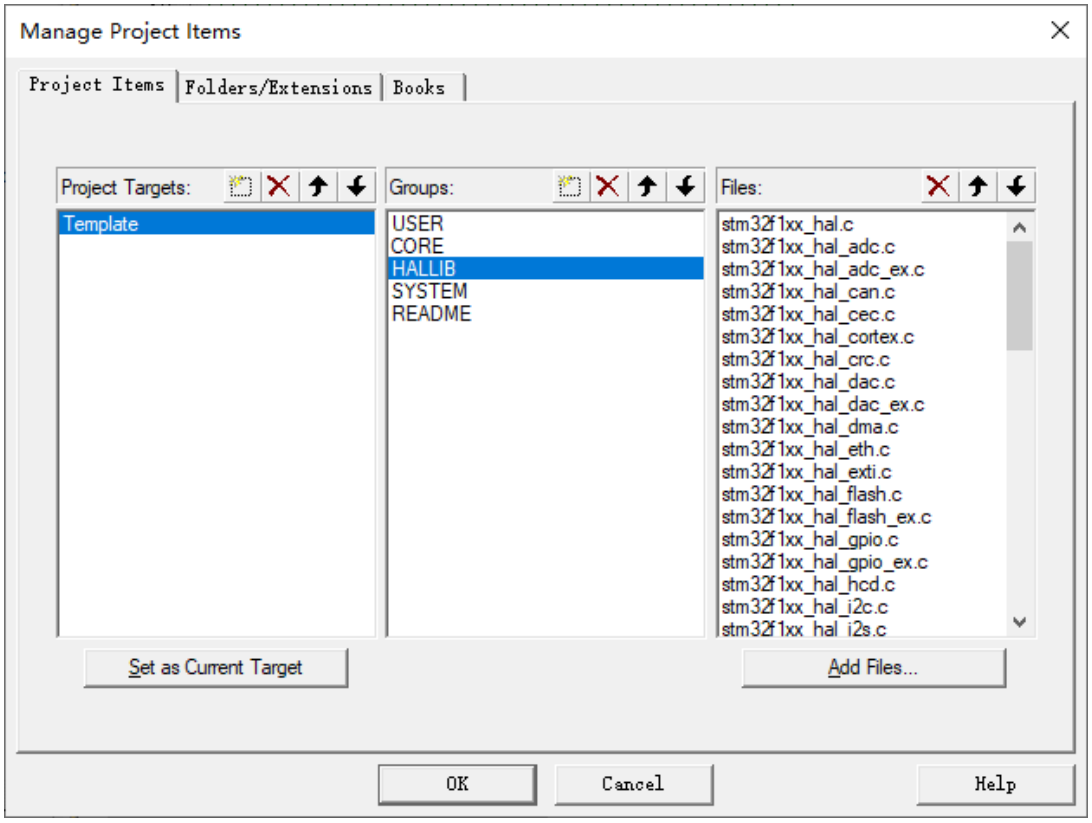


图 3.3.1.17 添加文件到 HALLIB 分组

stm32f1xx_hal_msp_template.c 文件内容是一些空函数，一般也不需要引入。删除某个方法如下图 3.3.1.18 所示：

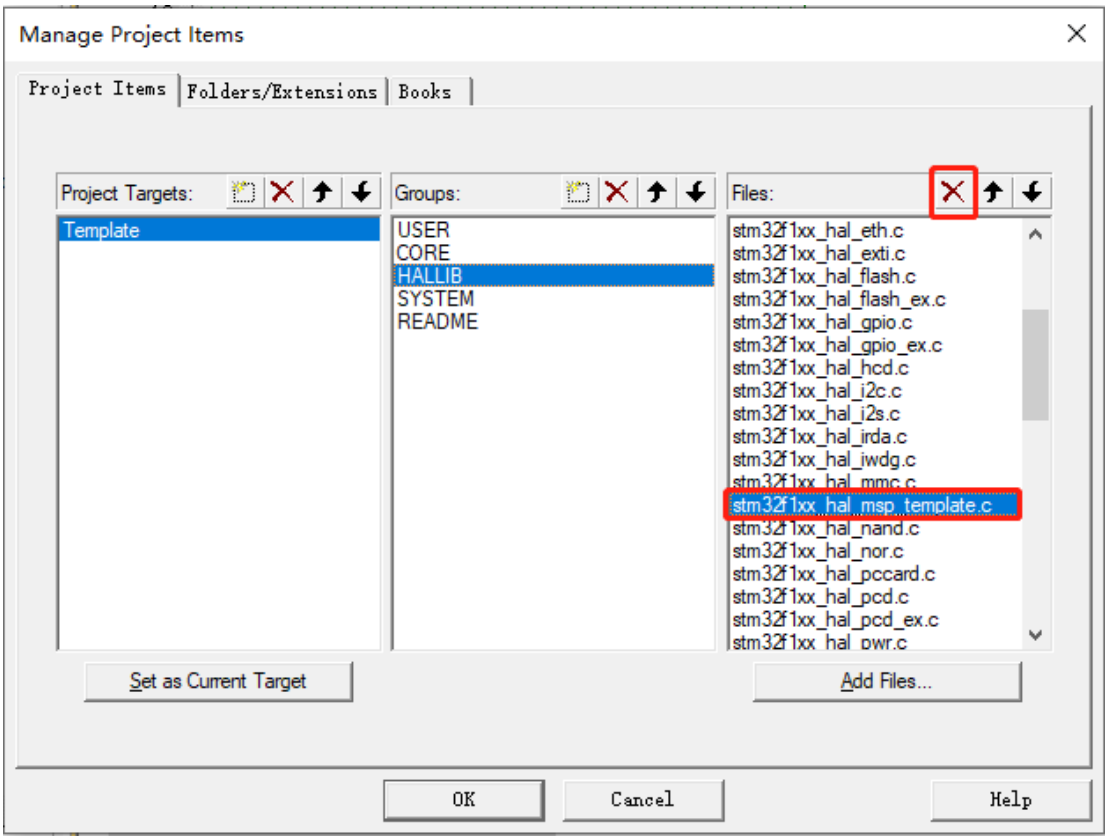


图 3.3.1.18 删掉 HALLIB 分组中不需要的源文件

11） 用上面同样的方法，将 Groups 定位到 CORE，USER 和 SYSTEM 分组之下，添加需要的文件。CORE 分组下面需要添加的文件为一些头文件以及启动文件 **startup_stm32f103xe.s(注意，默认添加的时候文件类型为.c，添加.h 头文件和 startup_stm32f103xe.s 启动文件的时候，你需要选择文件类型为 All files 才能看得到这些文件)**。USER 分组下面需要添加的文件 USER 目录下面所有的.c 文件：main.c，stm32f1xx_hal_msp.c，stm32f1xx_it.c 和 system_stm32f1xx.c 四个文件。 SYSTEM 分组下面需要添加 SYSTEM 文件夹下所有子文件夹内的.c 文件，包括 sys.c，usart.c 和 delay.c 三个源文件。添加完必要的文件到工程之后，最后点击 OK，回到工程主界面。操作过程如下图 3.3.1.19~3.3.1.22：

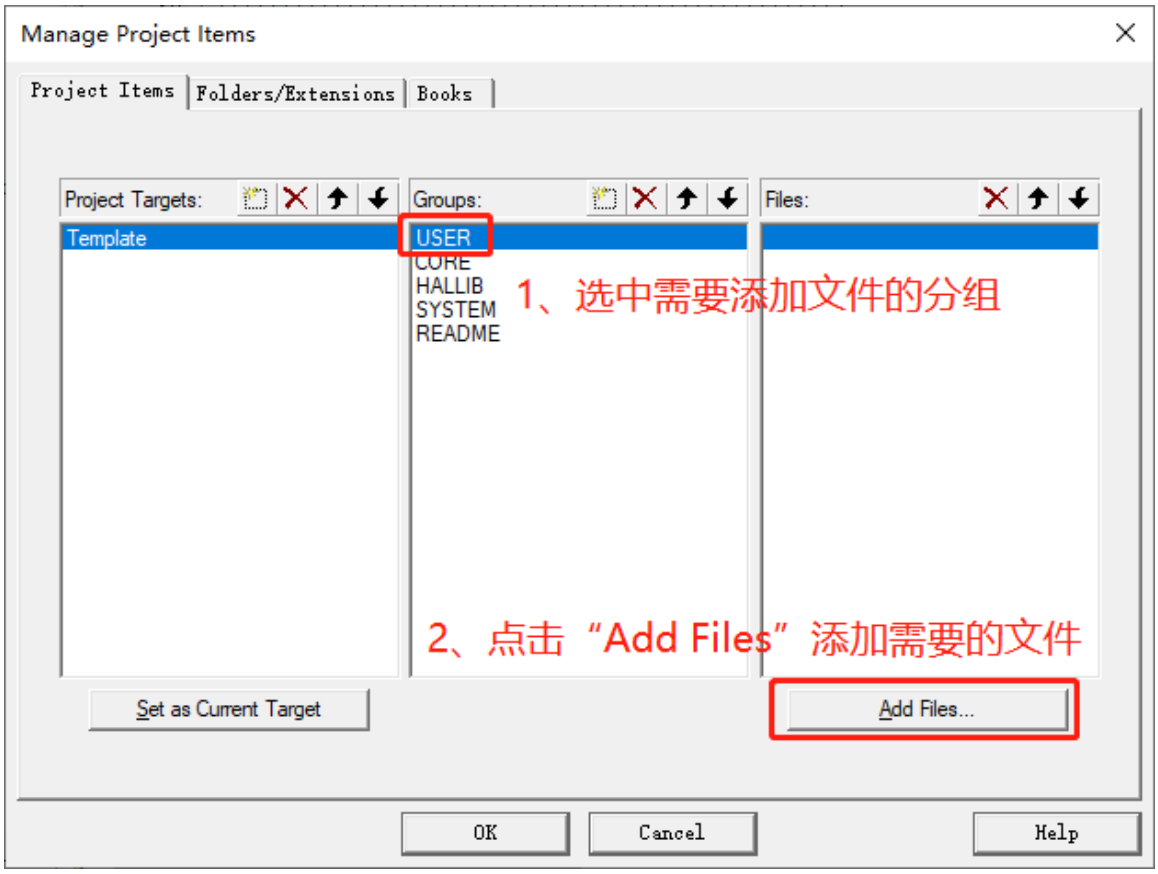


图 3.3.1.19 添加文件到 USER 分组

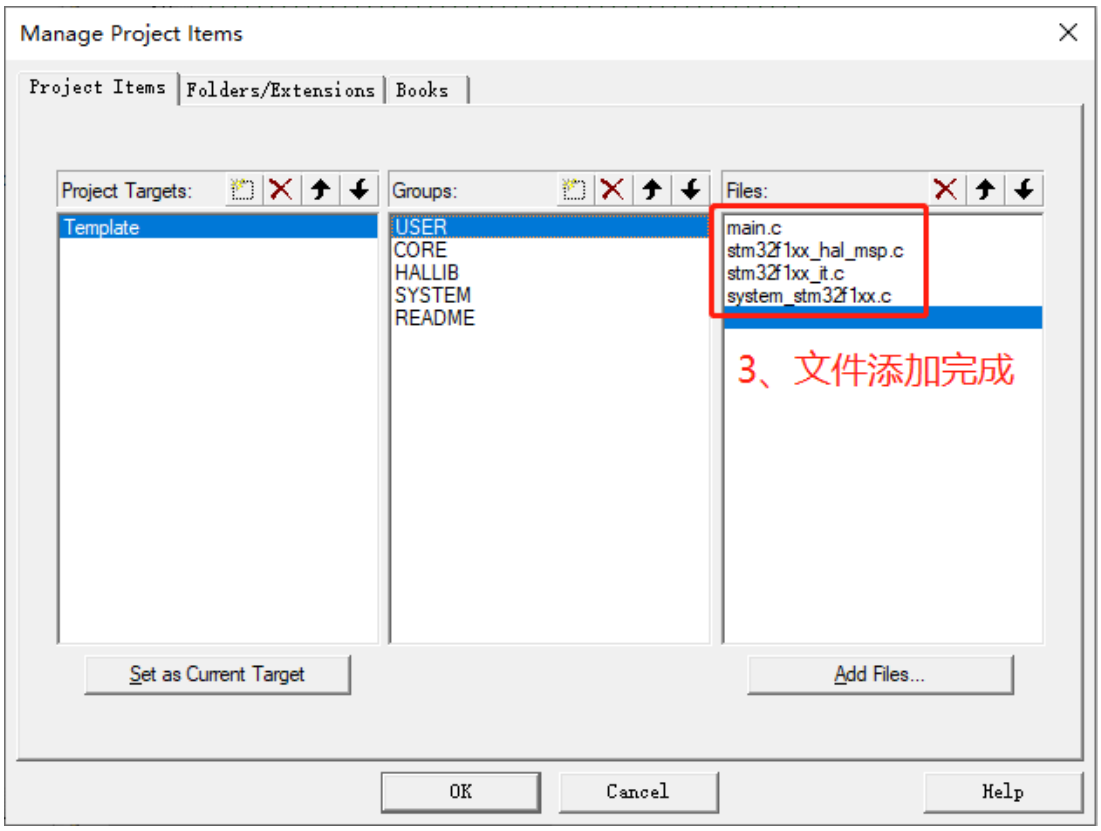


图 3.3.1.20 文件添加到 USER 分组完成

使用同样的方法，选中 CORE 分组，点击 Add Files 按钮，添加需要的文件到 CORE 分组。

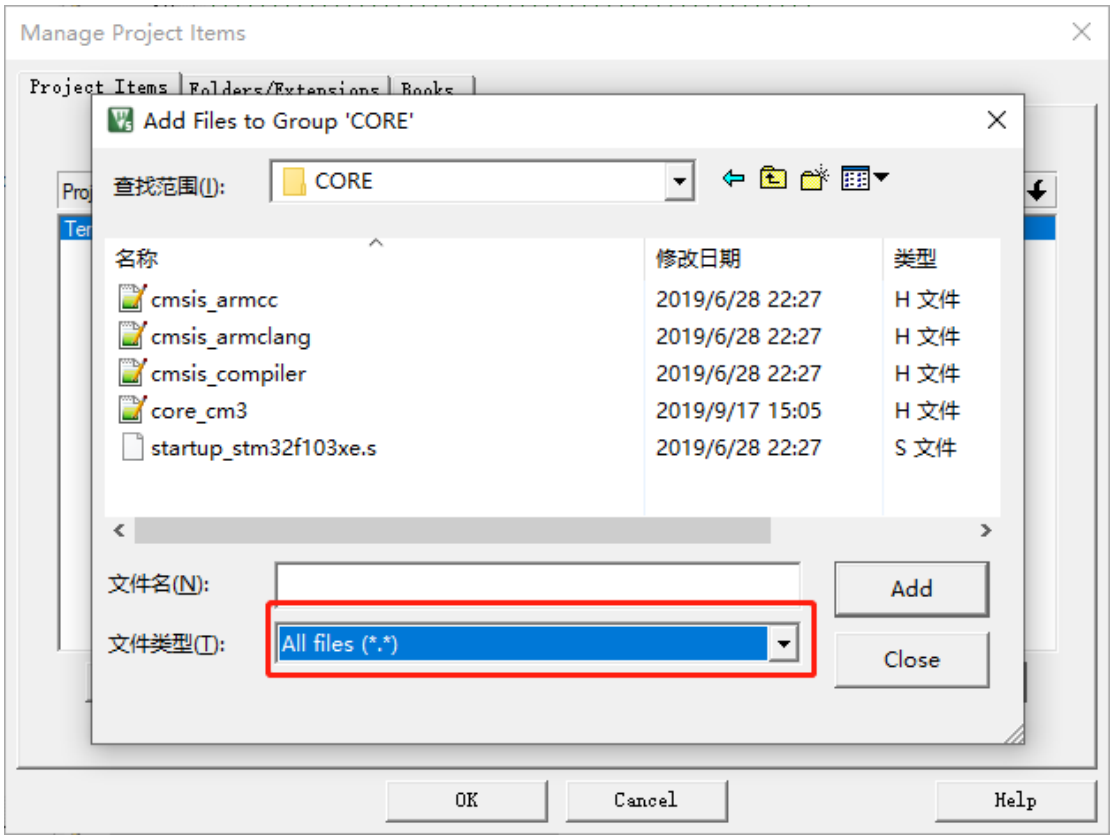


图 3.3.1.21 添加.h 头文件和启动文件到 CORE 分组

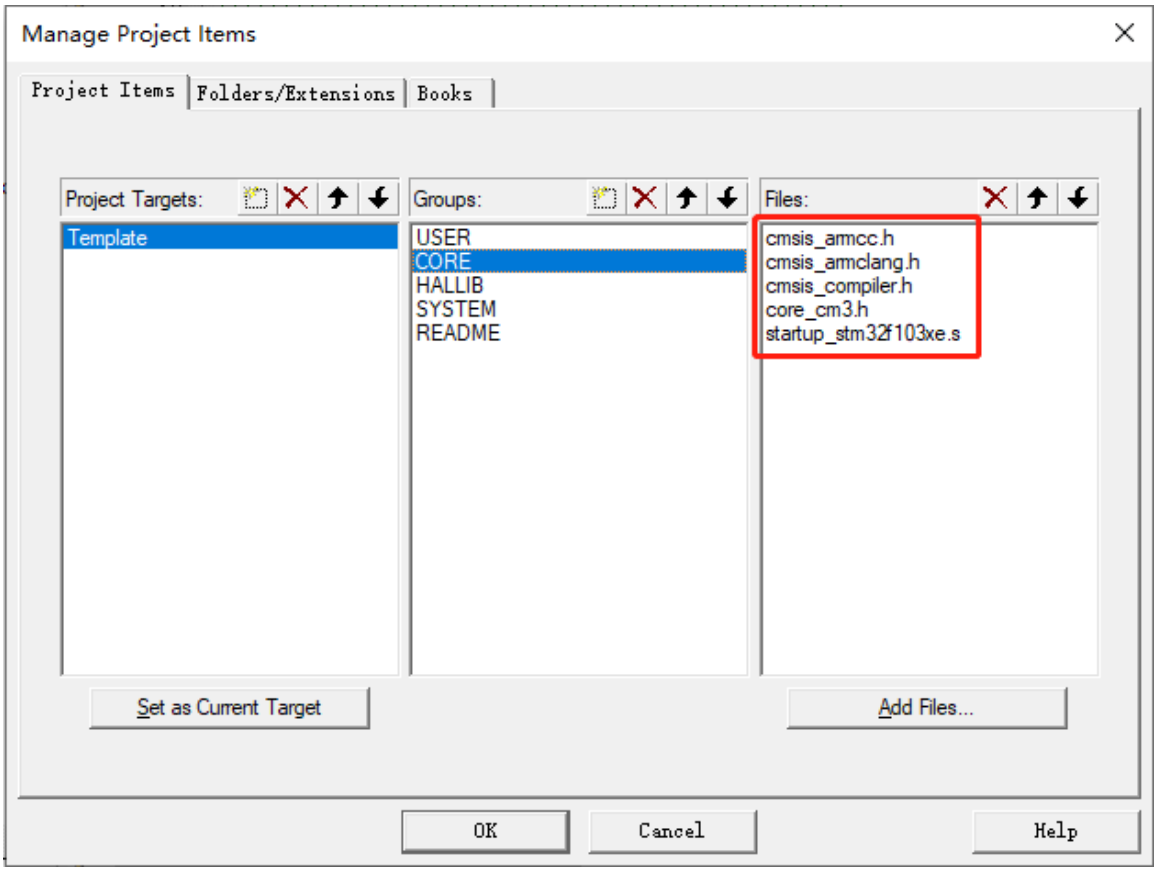


图 3.3.1.22 添加启动文件和头文件到 CORE 分组完成

最后添加文件到 SYSTEM 分组，这里需要注意，SYSTEM 文件夹包含三个子文件夹 sys,delay 和 usart。在添加文件的时候，需要分别定为到三个子文件夹内部，依次添加下面的.c 文件即可。添加完成后如下图 3.3.1.23 所示：

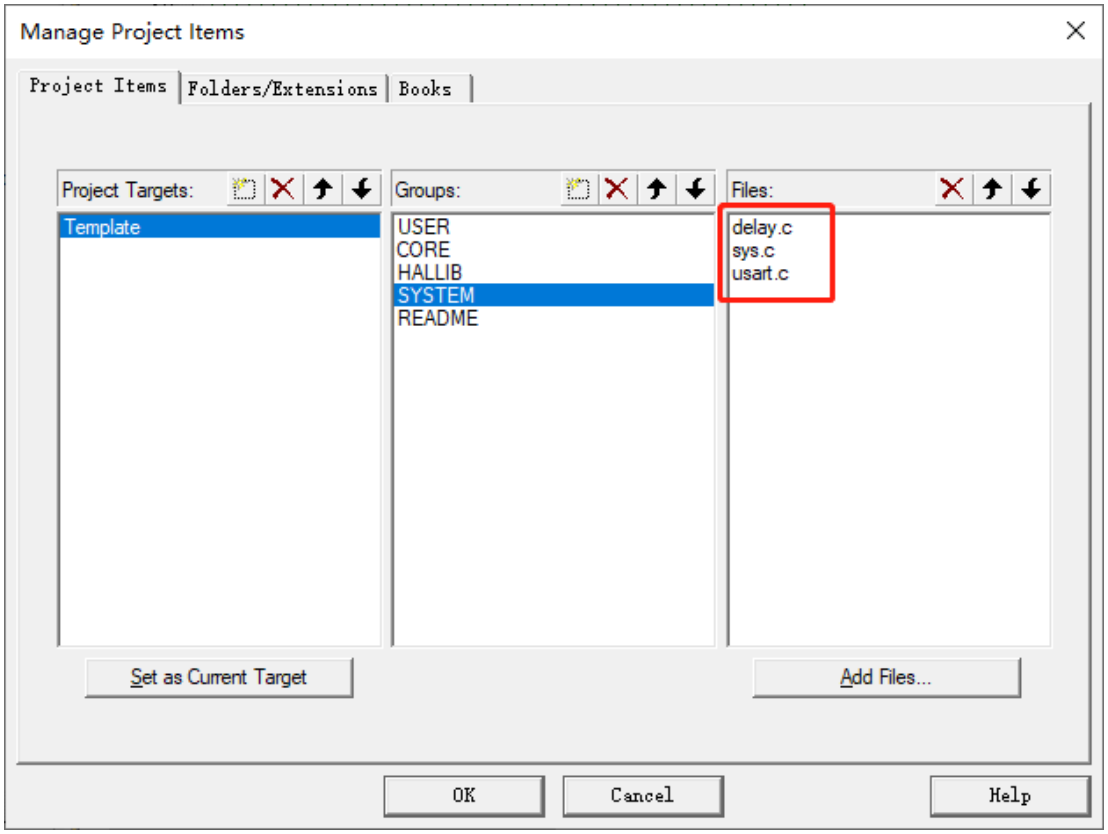


图 3.3.1.23 添加文件到 SYSTEM 分组

添加完所有文件到工程中之后，我们点击 OK 按钮，回到 MDK 工程主界面，如下图 3.3.1.24 所示：

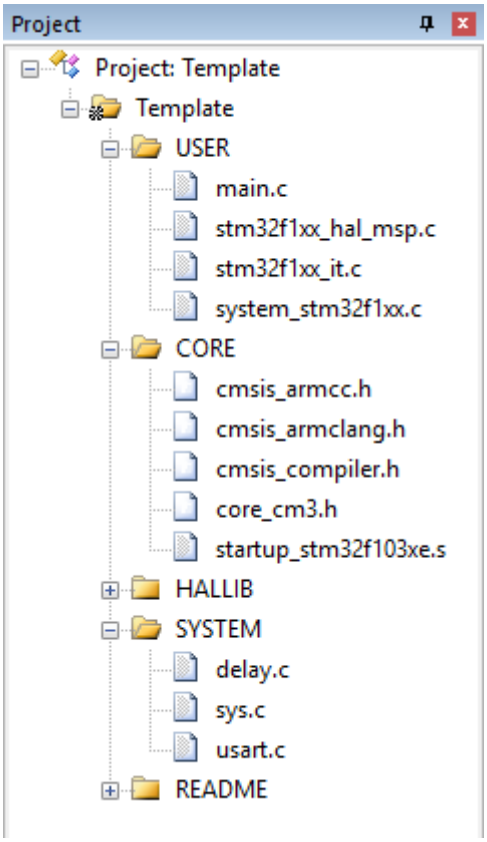


图 3.3.1.24 工程分组情况

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 3.3.1.25 和 3.3.1.26 所示，5 步之后添加相应的头文件路径。

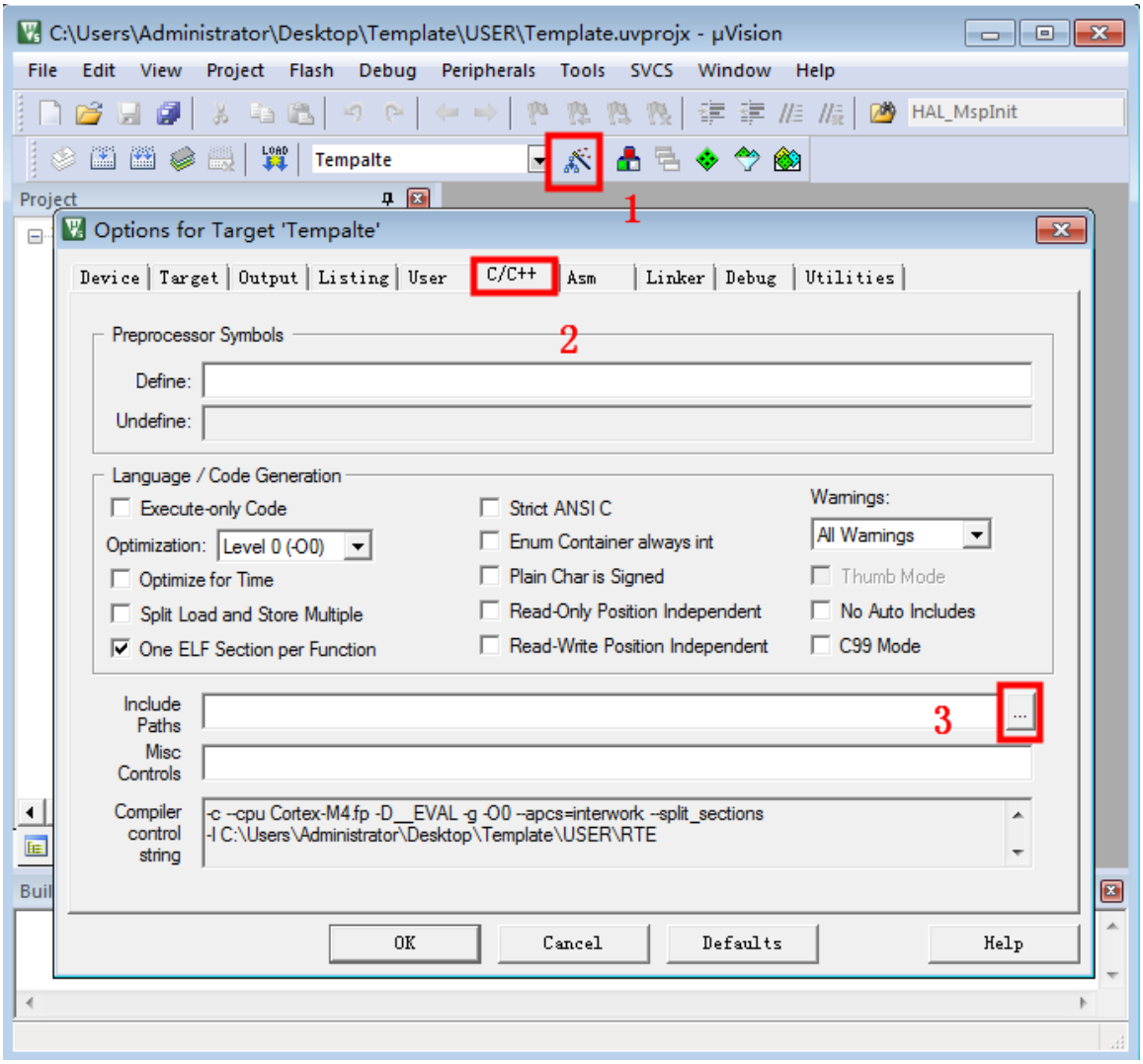


图 3.3.1.25 进入 PATH 配置界面

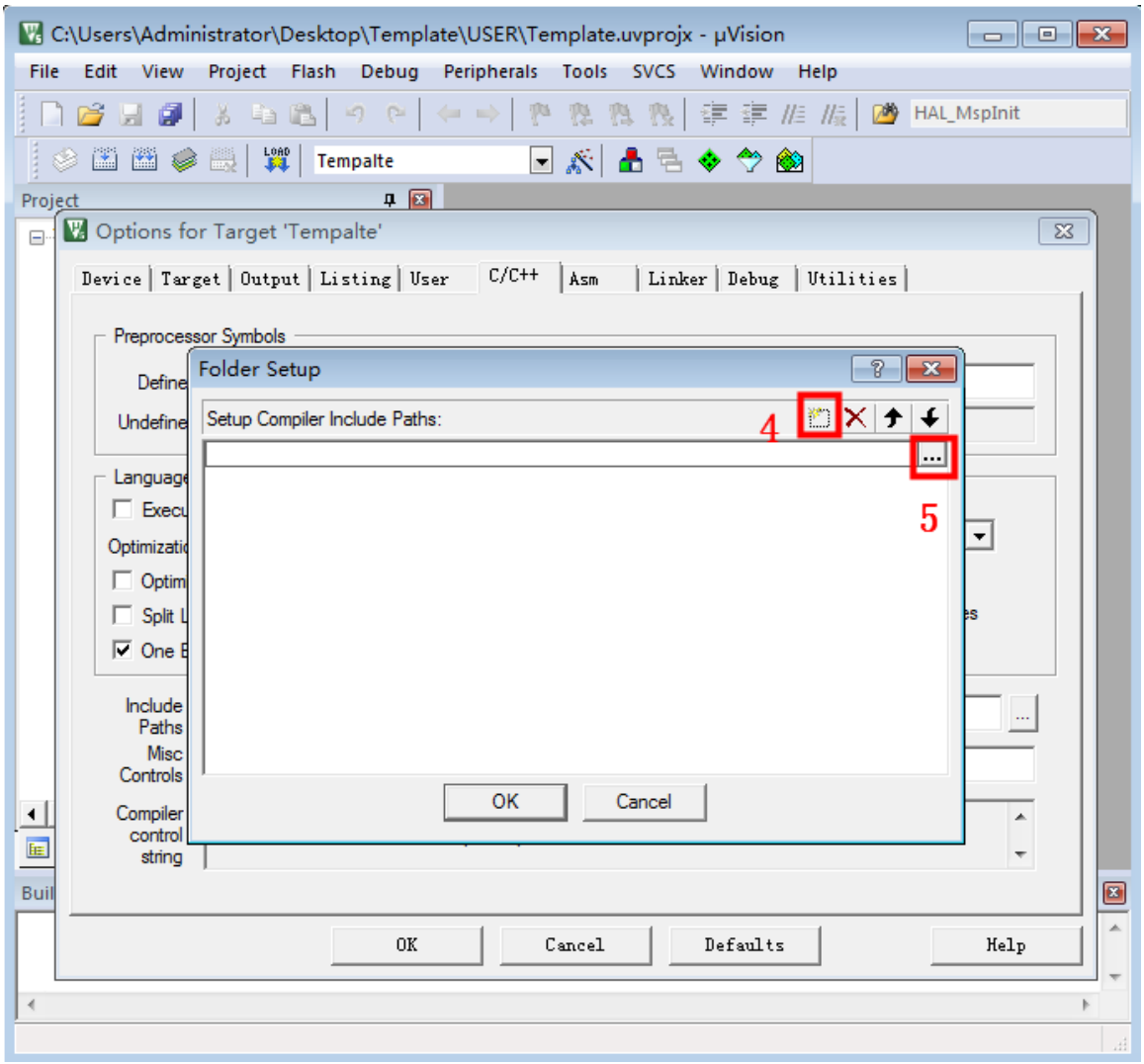


图 3.3.1.26 添加头文件路径到 PATH

这里大家需要注意，这里添加的路径必须添加到头文件所在目录的最后一级。比如在 SYSTEM 文件夹下面有三个子文件夹下面都有.h 头文件，这些头文件在工程中都需要使用到，所以我们必须将这三个子目录都包含进来。这里我们需要添加的头文件路径包括：\CORE， \USER\，\SYSTEM\delay ，\SYSTEM\usart，SYSTEM\sys 以及\HALLIB\Inc。这里还需要提醒大家，HAL 库存放头文件子目录是**\HALLIB\Inc**，不是 **HALLIB\Src**，其次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 3.3.1.27 所示。

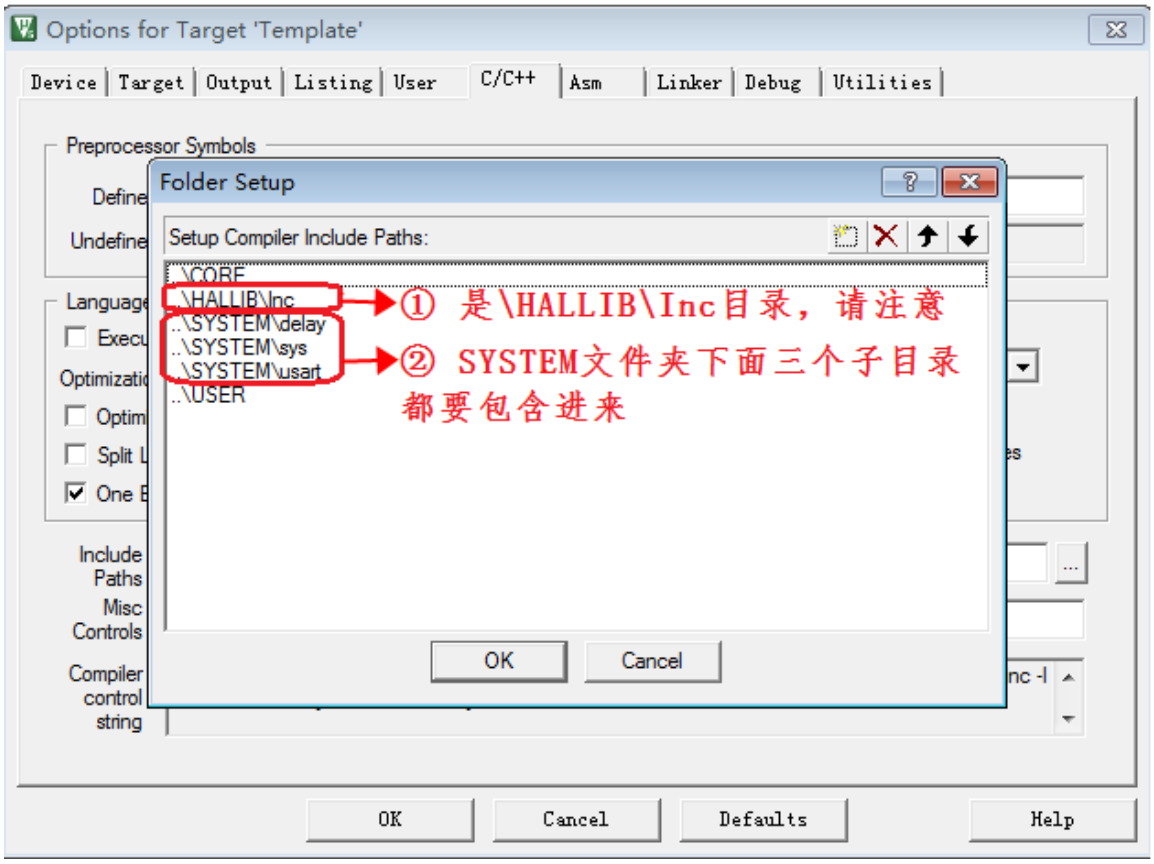


图 3.3.1.27 添加头文件路径

13） 接下来对于 STM32F103 系列的工程，还需要添加全局宏定义标识符，所谓全局宏定义标识符，就是在工程中任何地方都可见。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：USE_HAL_DRIVER,STM32F103xE。注意这里是两个标识符 USE_HAL_DRIVER 和 STM32F103xE，**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 0-1 Template 工程模板-新建工程章节使用**。本步骤操作过程如下图 3.3.1.28 所示：

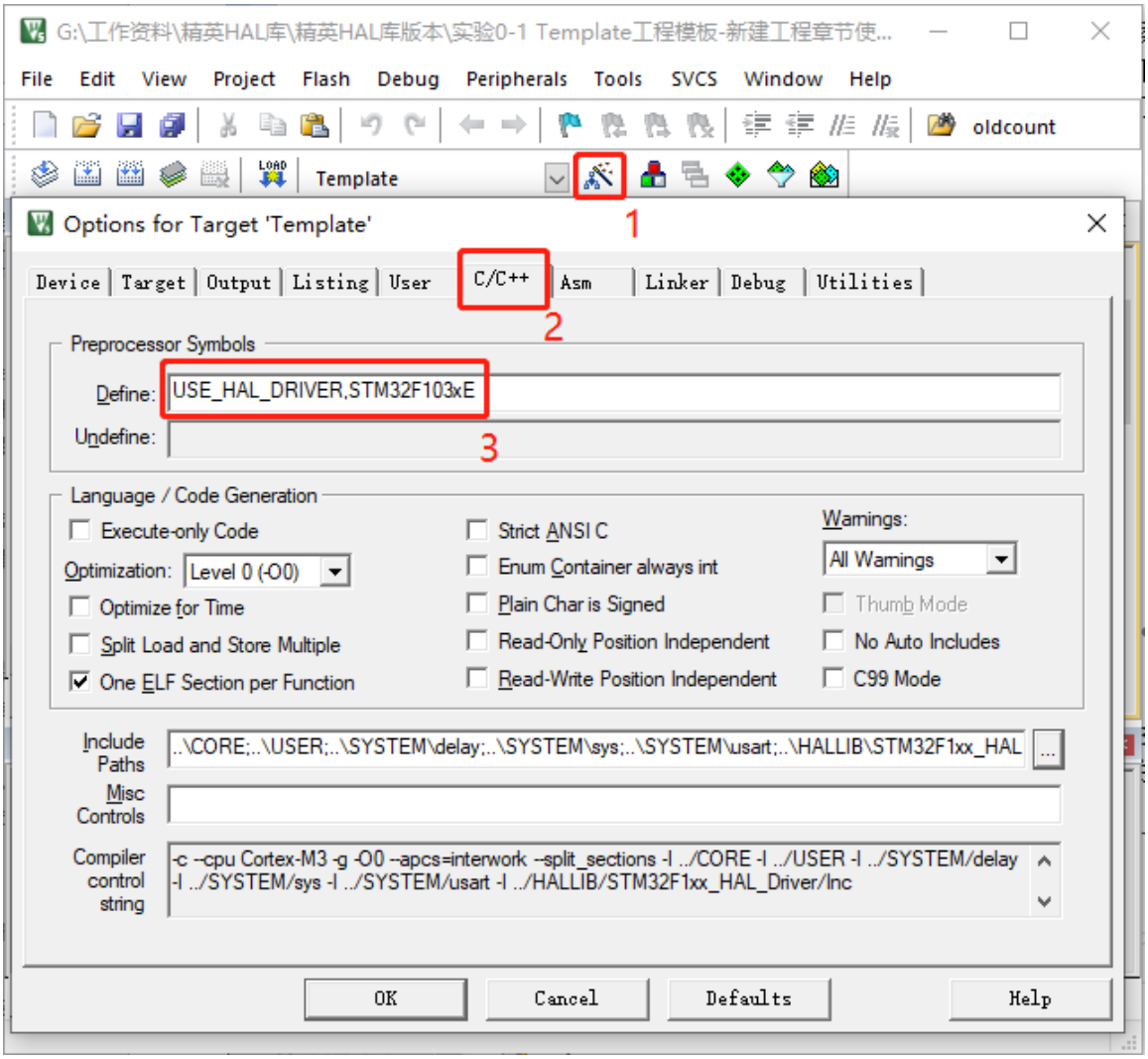


图 3.3.1.28 添加全局宏定义标识符

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编译后存放目录。前面我们讲过，MDK 默认编译后的中间文件存放目录为 USER 目录下面的 Listings 和 Objects 子目录，这里为了和我们 ALIENTEK 工程结构保持一致，我们重新选择存放到目录 OBJ 目录之下。操作方法是点击魔术棒  ，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**，然后依次点击 OK 即可。操作过程如下图 3.3.1.29 和 3.3.1.30 所示：

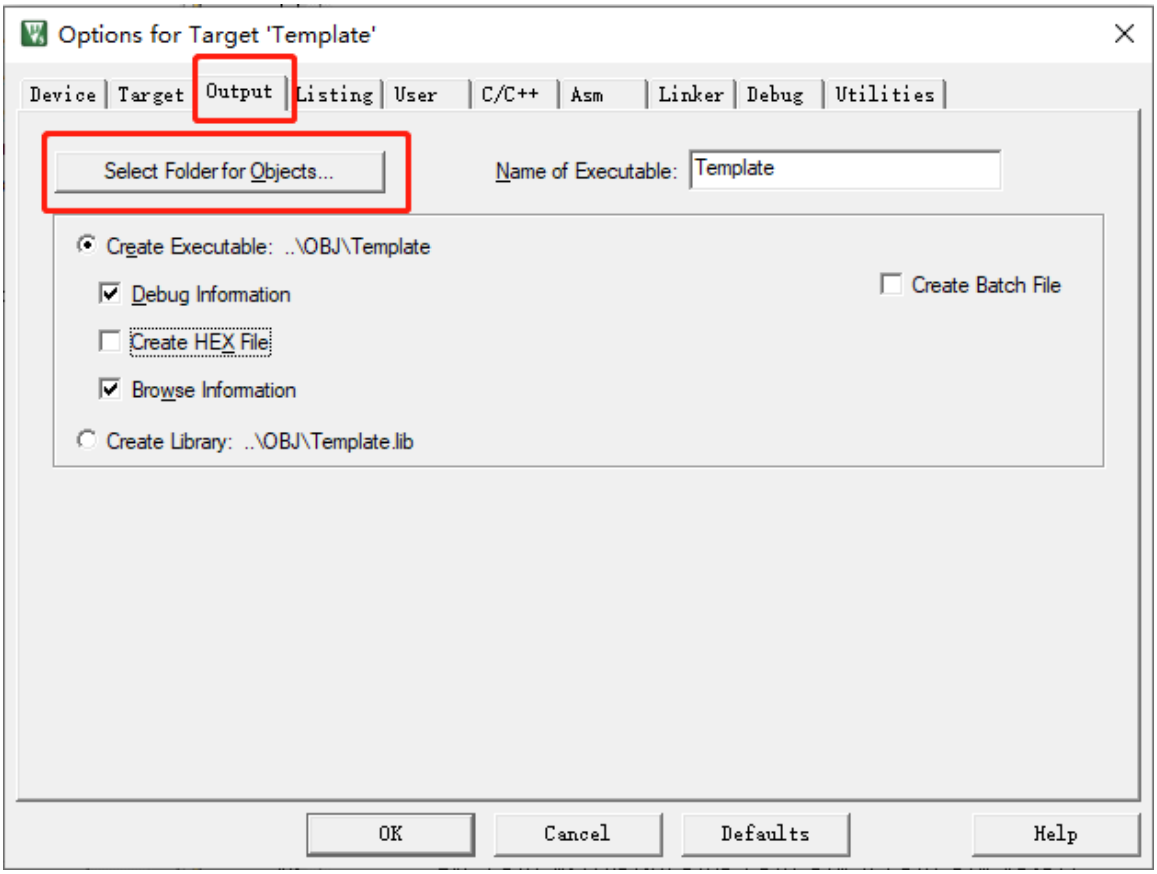


图 3.3.1.29 点击按钮“Select Folder for Objects…”

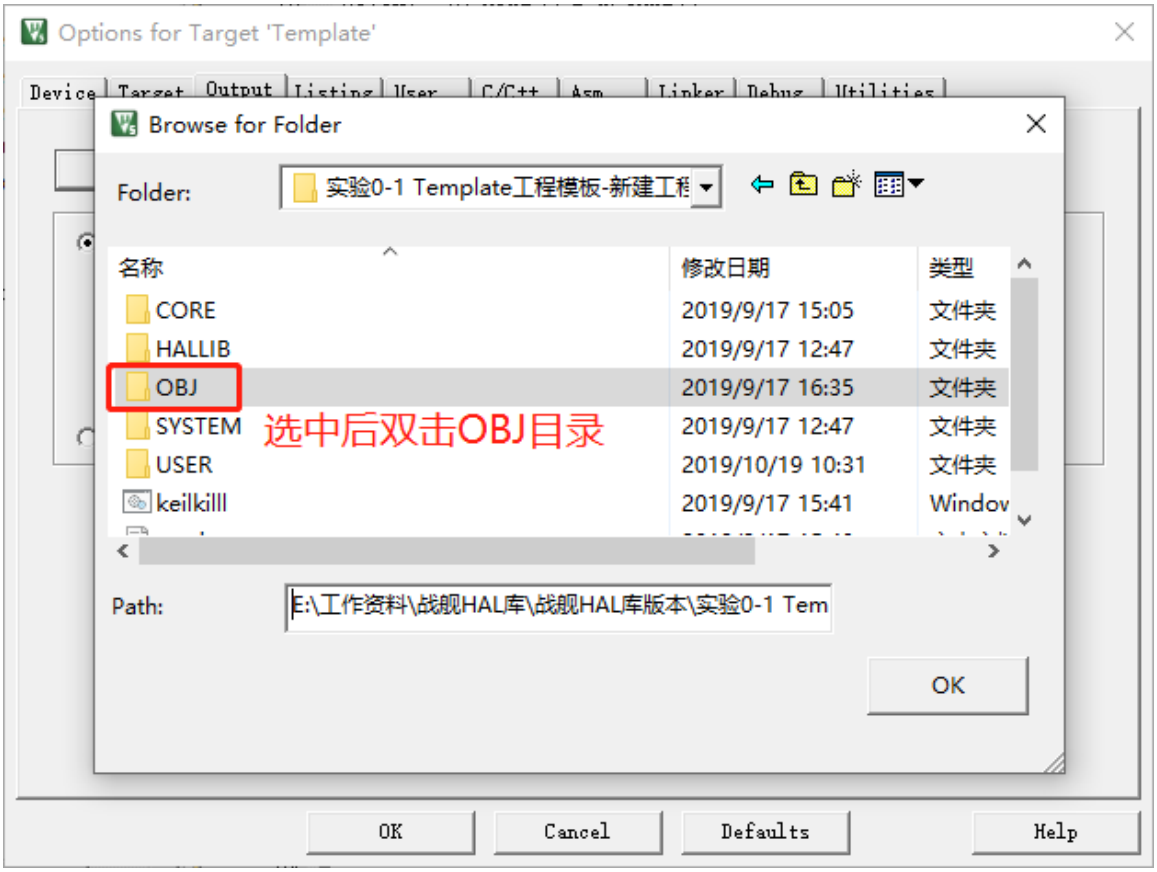


图 3.3.1.30 选择 OBJ 目录为中间文件存放目录

选择完 OBJ 目录为编译中间文件存放目录之后，点击 OK 回到 Output 选项卡。这里我们还要勾上“Create HEX File”选项和 Browse Information 选项。Create HEX File 选项选上是要求编译之后生成 HEX 文件。而 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等。具体操作方法如下图 3.3.1.31 所示：

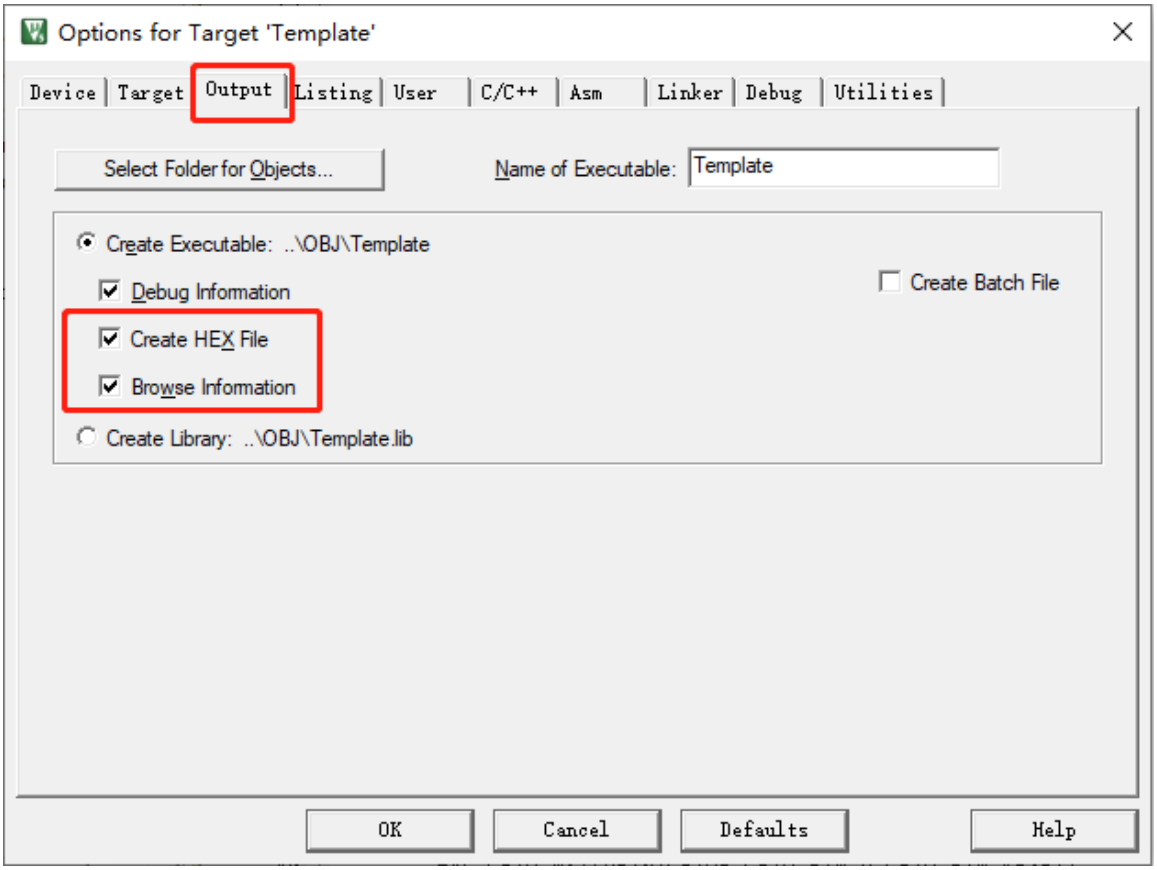


图 3.3.1.31 勾选上 Create HEX file 和 Browse Information 选项

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

```
void Delay(__IO uint32_t nCount);
void Delay(__IO uint32_t nCount)
{
    while(nCount--){}
}

int main(void)
{
  GPIO_InitTypeDef GPIO_Initure;

  HAL_Init();                          //初始化 HAL 库
  Stm32_Clock_Init(RCC_PLL_MUL9);      //设置时钟,72M
 __HAL_RCC_GPIOB_CLK_ENABLE();         //开启 GPIOB 时钟
 __HAL_RCC_GPIOE_CLK_ENABLE();         //开启 GPIOE 时钟
```

```
    GPIO_Initure.Pin=GPIO_PIN_5;                    //PB5
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;  //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);
    GPIO_Initure.Pin=GPIO_PIN_5;                    //PE5
    HAL_GPIO_Init(GPIOE,&GPIO_Initure);
    while(1)
    {
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET);       //PB5 置 1
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_SET);       //PE5 置 1
        Delay(0x7FFFFF);
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_RESET);   //PB5 置 0
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_RESET);   //PE5 置 0
        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 0-1 Template 工程模板-新建工程章节使用 ”**找到我们已经新建好的工程模板 USER 目录下面的 main.c 文件，直接复制过来即可。

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。

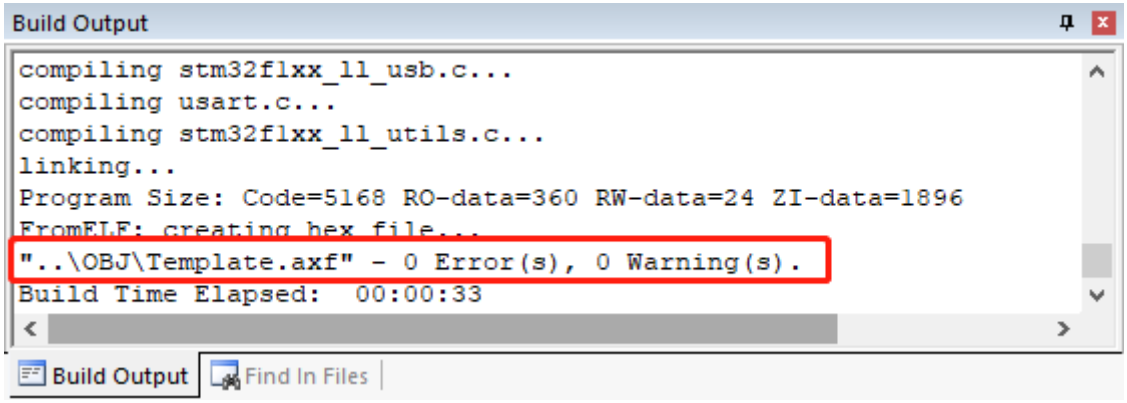


图 3.3.1.32 编译工程

这里大家可能会遇到编译之后会有一个警告，警告的内容是：“warning: #1-D: last line of file ends without a newline”。我们只需要在 main.c 函数结尾加一个回车即可解决，这个是 MDK 自身的 BUG。

17） 到这里，一个基于 HAL 库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 灯不停的闪烁现象。

18） 这里还一个地方需要大家修改一下，那就是关于系统初始化之后的中断优先级分组组号的设置。默认情况下调用 HAL 初始化函数 HAL_Init 之后，会设置分组为组 4，这里我们正点原子所有实验使用的是分组 2，所以我们修改 HAL_Init 函数内部，重新设置分组为组 2 即可。具体方法是:打开 HALLIB 分组之下的 stm32f1xx_hal.c 文件，搜索函数 HAL_Init，找到函数体，

里面默认有这样一行代码：

```
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
```

我们将入口参数 NVIC_PRIORITYGROUP_4 修改为 NVIC_PRIORITYGROUP_2 即可。关于中断优先级分组相关知识请参考本手册 4.5 小节即可。

3.3.2 工程模板解读

上一节，我们新建了一个基于 HAL 库的 STM32F103 工程模板，本节，我们将给大家讲解工程模板中的一些关键文件的作用以及整个工程模板程序运行流程。通过对本节内容的学习，大家将对 STM32F103 工程有一个比较全面的了解，为后面实验学习打下良好的基础。

3.3.2.1 关键文件介绍

在讲解之前我们需要说明一点，任何一个 MDK 工程，不管它有多复杂，无非就是一些.c 源文件和.h 头文件，还有一些类似.s 的启动文件或者 lib 文件等。在工程中，他们通过各种包含关系组织在一起，被我们用户代码最终调用或者引用。所以我们必须了解这些文件的作用以及他们之间的包含关系，从而理解这个工程的运行流程，这样我们才能在项目开发中得心应手。

1) HAL 库关键文件介绍如下表 3.3.2.1 所示：

文件	描述
sm32f1xx_hal.c	包含 HAL 通用 API（比如 HAL_Init,HAL_DeInit,HAL_Delay 等）。
stm32f1xx_hal.h	HAL 的头文件，它应被客户代码所包含。
stm32f1xx_hal_conf.h	HAL 的配置文件，主要用来选择使能何种外设以及一些时钟相关参数设置。其本身应该被客户代码所包含。
stm32f1xx_hal_def.h	包含 HAL 的通用数据类型定义和宏定义

表 3.3.2.1 HAL 库文件介绍

2) stm32f1xx_it.c/stm32f1xx_it.h 文件

这两个文件非常简单，也非常好理解。stm32f1xx_it.h 中主要是一些中断服务函数的申明。stm32f1xx_it.h 中是这些中断服务函数定义，而这些函数定义除了 Systick 中断服务函数 SysTick_Handler 外基本都是空函数，没有任何控制逻辑。一般情况下，我们可以去掉这两个文件，然后把中断服务函数写在工程中的任何一个可见文件中。

3) stm32f1xx.h 头文件

头文件 stm32f1xx.h 内容看似非常少，却非常重要，它是所有 stm32f1 系列的顶层头文件。使用 STM32F1 任何型号的芯片，都需要包含这个头文件。同时，因为 stm32f1 系列芯片型号非常多，ST 为每种芯片型号定义了一个特有的片上外设访问层头文件，比如 STM32F103 系列，ST 定义了一个头文件 stm32f103xx.h，然后 stm32f1xx.h 顶层头文件会根据工程芯片型号，来选择包含对应芯片的片上外设访问层头文件。我们可以打开 stm32f1xx.h 头文件可以看到，里面有如下几行代码：

```
#if defined(STM32F100xB)
  #include "stm32f100xb.h"
  …
#elif defined(STM32F101xE)
  #include "stm32f101xe.h"
…
```

```
#else
#error "Please select first the target STM32F1xx device used in your application
                                                    (in stm32f1xx.h file)"
#endif
```

这几行代码非常好理解，我们以 stm3f103 为例，如果定义了宏定义标识符 STM32F103xx，那么头文件 stm32f1xx.h 将会包含头文件 stm32f103xx.h。实际上，在我们上一节新建工程的时候，我们在 C/C++选项卡里面输入的全局宏定义标识符中就包含了标识符 STM32F103xx（请参考图 3.3.1.28）。所以头文件 stm32f103xx.h 一定会被整个工程所引用。

4) stm32f103xx.h 头文件

根据前面的讲解，stm32f103xx.h 是 stm32f103 系列芯片通用的片上外设访问层头文件，只要我们进行 stm32f103 开发，就必然要使用到该文件。打开该文件我们可以看到里面主要是一些结构体和宏定义标识符。这个文件的主要作用是寄存器定义声明以及封装内存操作。在后面寄存器地址名称映射分析小节我们会给大家详细讲解。

5) system_stm32f1xx.c/system_stm32f1xx.h 文件

头文件 system_stm32f1xx.h 和源文件 system_stm32f1xx.c 主要是声明和定义了系统初始化函数 SystemInit 以及系统时钟更新函数 SystemCoreClockUpdate。SystemInit 函数的作用是进行时钟系统的一些初始化操作以及中断向量表偏移地址设置，但它并没有设置具体的时钟值，这是与标准库的最大区别，在使用标准库的时候，SystemInit 函数会帮我们配置好系统时钟配置相关的各个寄存器。在启动文件 startup_stm32f103xx.s 中会设置系统复位后，直接调用 SystemInit 函数进行系统初始化。SystemCoreClockUpdate 函数是在系统时钟配置进行修改后，调用这个函数来更新全局变量 SystemCoreClock 的值，变量 SystemCoreClock 是一个全局变量，开放这个变量可以方便我们在用户代码中直接使用这个变量来进行一些时钟运算。

6) stm32f1xx_hal_msp.c 文件

MSP，全称为 MCU support package,关于怎么理解 MSP，我们后面在讲解程序运行流程的时候会给大家举例详细讲解，这里大家只需要知道，函数名字中带有 MspInit 的函数，它们的作用是进行 MCU 级别硬件初始化设置，并且它们通常会被上一层的初始化函数所调用，这样做的目的是为了把 MCU 相关的硬件初始化剥夺出来，方便用户代码在不同型号的 MCU 上移植。stm32f1xx_hal_msp.c 文件定义了两个函数 HAL_MspInit 和 HAL_MspDeInit。这两个函数分别被文件 stm32f1xx_hal.c 中的 HAL_Init 和 HAL_DeInit 所调用。HAL_MspInit 函数的主要作用是进行 MCU 相关的硬件初始化操作。例如我们要初始化某些硬件，我们可以硬件相关的初始化配置写在 HAL_MspDeinit 函数中。这样的话，在系统启动后调用了 HAL_Init 之后，会自动调用硬件初始化函数。实际上，我们在工程模板中直接删掉 stm32f1xx_hal_msp.c 文件也不会对程序运行产生任何影响。对于这个文件存在的意义，我们在后面讲解完程序运行流程之后，大家会有更加清晰的理解。

7) startup_stm32f103xe.s 启动文件

STM32 系列所有芯片工程都会有一个.s 启动文件。对于不同型号的 stm32 芯片启动文件也是不一样的。我们的开发板是 STM32F103 系列，所以我们需要使用与之对应的启动文件 startup_stm32f103xe.s。启动文件的作用主要是进行堆栈的初始化，中断向量表以及中断函数定义等。启动文件有一个很重要的作用就是系统复位后引导进入 main 函数。打开启动文件 startup_stm32f103xe.s，可以看到下面几行代码：

```
; Reset handler
Reset_Handler    PROC
```

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

Reset_Handler 在我们系统启动的时候会执行，这几行代码的作用是在系统启动之后，首先调用 SystemInit 函数进行系统初始化，然后引导进入 main 函数执行用户代码。

3.3.2.2 HAL 库中__weak 修饰符讲解

在 HAL 库中，很多回调函数前面使用__weak 修饰符，这里我们有必要给大家讲解__weak 修饰符的作用。

weak 顾名思义是“弱”的意思，所以如果函数名称前面加上__weak 修饰符，我们一般称这个函数为“弱函数”。加上了__weak 修饰符的函数，用户可以在用户文件中重新定义一个同名函数，最终编译器编译的时候，会选择用户定义的函数，如果用户没有重新定义这个函数，那么编译器就会执行__weak 声明的函数，并且编译器不会报错。

这里我给大家举个例子来加深大家的理解。比如我们打开工程模板，找到并打开文件 stm32f1xx_hal.c 文件，里面定义了一个函数 HAL_MspInit，定义如下：

```
__weak void HAL_MspInit(void)
{
}
```

大家可以看出，HAL_MspInit 函数前面有加修饰符__weak。同时，在该文件的前面有定义函数 HAL_Init，并且 HAL_Init 函数中调用了函数 HAL_MspInit。

```
HAL_StatusTypeDef HAL_Init(void)
{

  …//此处省略部分代码

  HAL_MspInit();
  return HAL_OK;
}
```

如果我们没有在工程中其他地方重新定义 HAL_MspInit()函数，那么 HAL_Init 初始化函数执行的时候，会默认执行 stm32f1xx_hal.c 文件中定义的 HAL_MspInit 函数，而这个函数没有任何控制逻辑。如果用户在工程中重新定义函数 HAL_MspInit，那么调用 HAL_Init 之后，会执行用户自己定义的 HAL_MspInit 函数而不会执行 stm32f1xx_hal.c 默认定义的函数。也就是说，表面上我们看到函数 HAL_MspInit 被定义了两次，但是因为有一次定义是弱函数，使用了__weak 修饰符，所以编译器不会报错。

__weak 在回调函数的时候经常用到。这样的好处是，系统默认定义了一个空的回调函数，保证编译器不会报错。同时，如果用户自己要定义用户回调函数，那么只需要重新定义即可，不需要考虑函数重复定义的问题，使用非常方便，在 HAL 库中__weak 关键字被广泛使用。

3.3.2.3 Msp 回调函数执行过程解读

大家先打开我们前面新建的工程模板，搜索 MspInit 字符串可以发现，在我们的工程模板文件中，有 70 多个文件定义或者调用了函数名字中包含 MspInit 字符串的函数，而且函数名字基本遵循 HAL_PPP_MspInit 格式（PPP 代表任意外设）。那么这些函数是怎么被程序调用，又是什么作用呢？下面我们以串口为例进行讲解。

大家打开我们的工程模板 SYSTEM 分组下面的 usart.c 文件可以看到，内部我们定义了两个函数 uart_init 和 HAL_UART_MspInit。我们先来大致看看这两个函数的定义（基于篇幅考虑我们省略部分非关键代码行）：

```
void uart_init(u32 bound)
{

    UART1_Handler.Instance=USART1;                  //USART1
    UART1_Handler.Init.BaudRate=bound;              //波特率
    …//此处省略部分串口 1 参数设置代码
    UART1_Handler.Init.Mode=UART_MODE_TX_RX;            //收发模式
    HAL_UART_Init(&UART1_Handler);      //HAL_UART_Init()会使能 UART1
}

//UART 底层初始化，时钟使能，引脚配置，中断配置
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
    …//此处省略部分代码
    GPIO_Initure.Pin=GPIO_PIN_9;                //PA9
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //复用推挽输出
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化 PA9

    GPIO_Initure.Pin=GPIO_PIN_10;               //PA10
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化 PA10
    …//此处省略部分代码
}
```

用户函数 uart_init 主要作用是设置串口 1 相关参数，包括波特率，停止位，奇偶校验位等，并且最终是通过调用 HAL_UART_Init 函数进行参数设置。而函数 HAL_UART_MspInit 则主要进行串口 GPIO 引脚初始化设置。接下来我们打开 usart_init 函数内部调用的 UART 初始化函数 HAL_UART_Init 可以看到代码如下：

```
HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart)
{
  …//此处省略部分代码
  if(huart->State == HAL_UART_STATE_RESET)//如果串口没有进行过初始化
  {
    huart->Lock = HAL_UNLOCKED;
    HAL_UART_MspInit(huart);
  }
  …//此处省略部分代码
```

```
    return HAL_OK;
}
```

在函数 HAL_UART_Init 内部，通过判断逻辑判断如果串口还没有进行初始话，那么会调用函数 HAL_UART_MspInit 进行相关初始化设置。同时，我们可以看到，在文件 stm32f1xx_hal_uart.c 内部，有定义一个弱函数 HAL_UART_MspInit，内容如下：

```
__weak void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
  UNUSED(huart);
}
```

这里定义的弱函数 HAL_UART_MspInit 是一个空函数，没有任何实际的控制逻辑。根据前面的讲解可知，__weak 修饰符定义的弱函数如果用户自己重新定义了这个函数，那么会优先执行用户定义函数。所以，实际上在函数 HAL_UART_Init 内部调用的 HAL_UART_MspInit()函数，最终执行的是用户在 usart.c 中自定义的 HAL_UART_MspInit()函数。

那么整个串口初始化的过程为：用户函数 usart_init→ HAL_UART_Init→ HAL_UART_MspInit。学到这里有同学会问，为什么串口相关初始化不在 HAL_UART_Init 函数内部一次初始化而还要调用函数 HAL_UART_MspInit()呢？这实际就是 HAL 库的一个优点，它通过开放一个回调函数 HAL_UART_MspInit()，让用户自己去编写与串口相关的 MCU 级别的硬件初始化，而与 MCU 无关的串口参数相关的通用配置则放在 HAL_UART_Init。

我们要初始化一个串口，首先要设置和 MCU 无关的东西，例如波特率，奇偶校验，停止位等，这些参数设置和 MCU 没有任何关系，可以使用 STM32F1，也可以是 STM32F2/F3/F4/F7 上的串口。而一个串口设备它需要一个 MCU 来承载，例如用 STM32F1 来做承载，PA9 做为发送，PA10 做为接收，MSP 就是要初始化 STM32F1 的 PA9,PA10，配置这两个引脚。所以 HAL 驱动方式的初始化流程就是：HAL_USART_Init()—>HAL_USART_MspInit() ，先初始化与 MCU 无关的串口协议，再初始化与 MCU 相关的串口引脚。在 STM32 的 HAL 驱动中 HAL_PPP_MspInit()作为回调，被 HAL_PPP_Init()函数所调用。当我们需要移植程序到 STM32F1 平台的时候，我们只需要修改 HAL_PPP_MspInit 函数内容而不需要修改 HAL_PPP_Init 入口参数内容。

在 STM32 的 HAL 库中，大部分外设都有回调函数 HAL_MspInit，通过对本小节学习，大家对这些回调函数的作用和调用过程会非常熟悉，这里我们就不做一一列举。

3.3.2.4 程序执行流程图

经过前面的讲解，大家对工程模板以及关键文件有了比较详细的了解。接下来我们看看程序执行流程如下图 3.3.2.4.1 所示：

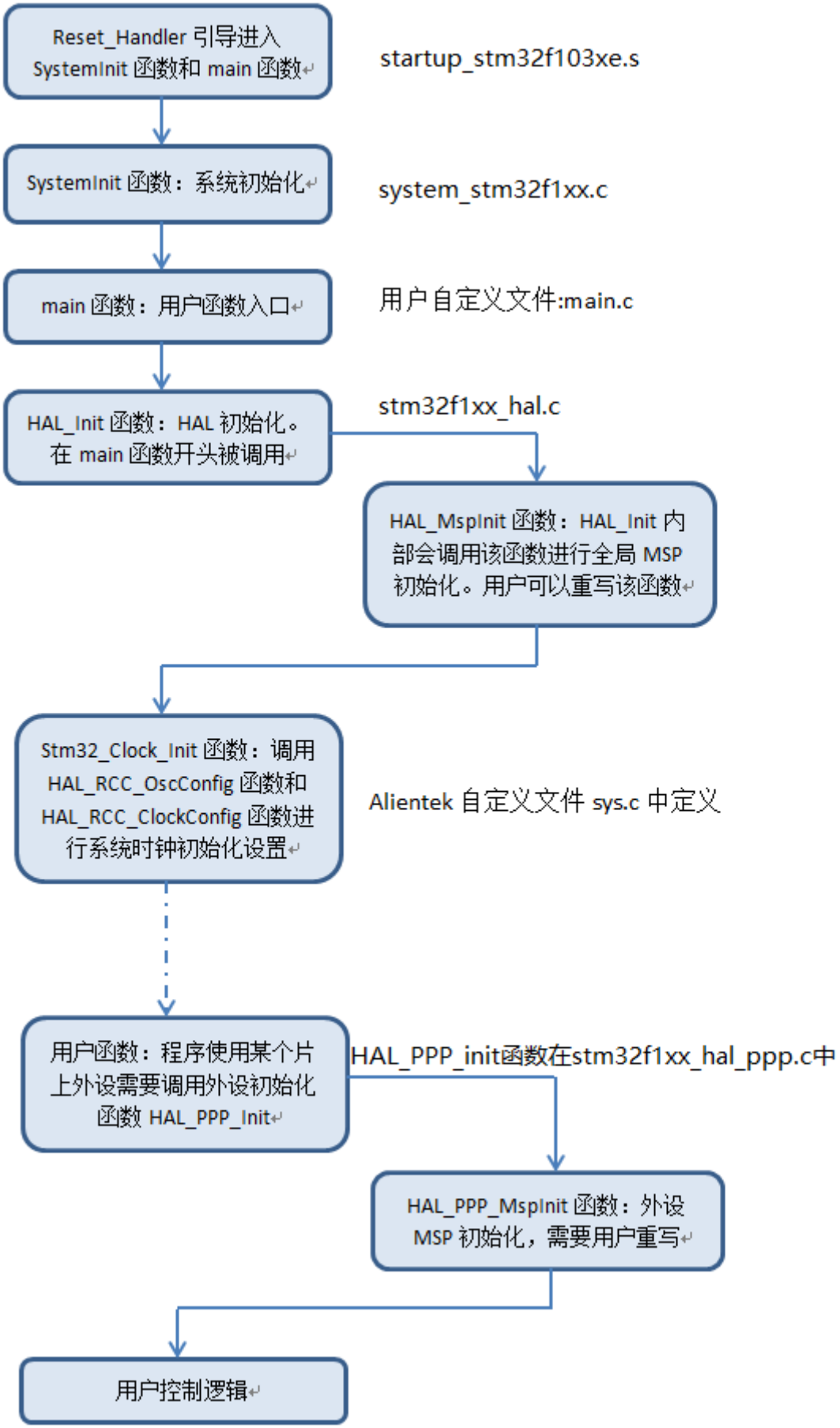

Reset_Handler 引导进入 SystemInit 函数和 main 函数
startup_stm32f103xe.s
SystemInit 函数：系统初始化
system_stm32f1xx.c
main 函数：用户函数入口
用户自定义文件:main.c
HAL_Init 函数：HAL 初始化。在 main 函数开头被调用
stm32f1xx_hal.c
HAL_MspInit 函数：HAL_Init 内部会调用该函数进行全局 MSP 初始化。用户可以重写该函数
Stm32_Clock_Init 函数：调用 HAL_RCC_OscConfig 函数和 HAL_RCC_ClockConfig 函数进行系统时钟初始化设置
Alientek 自定义文件 sys.c 中定义
用户函数：程序使用某个片上外设需要调用外设初始化函数 HAL_PPP_Init
HAL_PPP_init函数在stm32f1xx_hal_ppp.c中
HAL_PPP_MspInit 函数：外设 MSP 初始化，需要用户重写
用户控制逻辑

图 3.3.2.4.1 程序执行流程

从该流程图可以非常清晰的理解整个程序执行流程，这里我们略微讲解一下。启动文件 startup_stm32f103xe.s 中 Reset_Handler 部分会引导先执行 SystemInit 函数，然后再进入 main 函数。在 main 函数内部，一般情况下，我们会把 HAL 初始化函数 HAL_Init 放在最开头部分，然后再进行时钟初始化设置。这些设置完成之后，接下来便是调用外设初始化函数 HAL_PPP_Init 进行外设参数初始化设置，同时重写回调函数 HAL_PPP_MspInit 进行外设 MCU 相关的参数设置。最后编写我们的控制逻辑。关于程序执行流程我们就给大家介绍到这里。

3.4 程序下载与调试

上一节，我们学会了如何在 MDK 下创建 STM32F1 工程。本节，我们将向读者介绍 STM32F1 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、STM32F1 程序下载；2、利用 ST-LINK 对 STM32F1 进行下载与在线调试。

这里大家要注意，为了让大家能够更好的学习调试，我们将 3.3 小节新建的工程模板中的 main.c 文件内容进行了简单的修改。修改后的工程模板在光盘目录：\4，程序源码\2，标准例程-库函数版本\实验 0-2 Template 工程模板-调试章节使用。本小节下载和调试的工程请参考该工程模板。

3.4.1 STM32 串口程序下载

STM32F1 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 STM32F103 下载代码。不过，最简单也是最经济的，就是通过串口给 STM32F103 下载代码。本节，我们将向大家介绍，如何利用串口给 STM32F103（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 战舰 STM32F103 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：

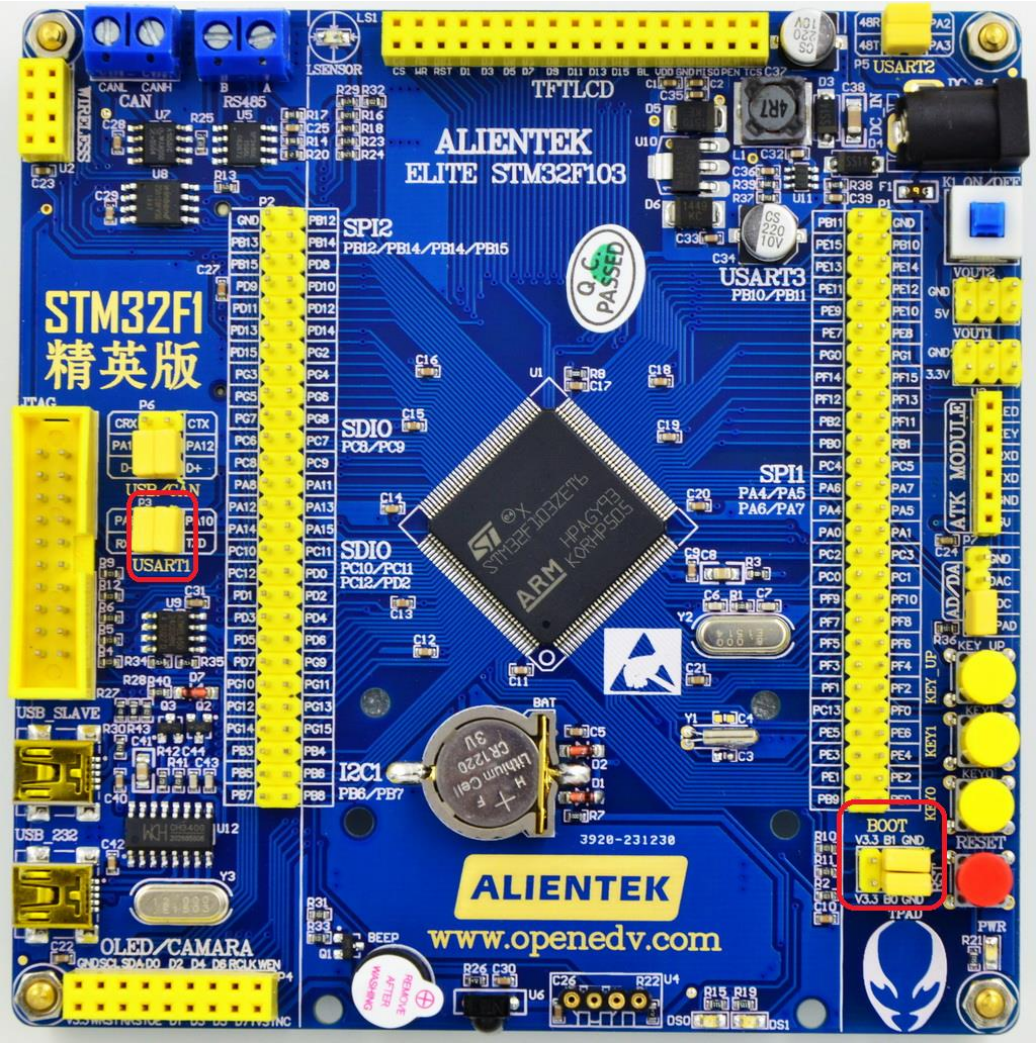


图 3.4.1.1 开发板串口下载跳线设置

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），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：

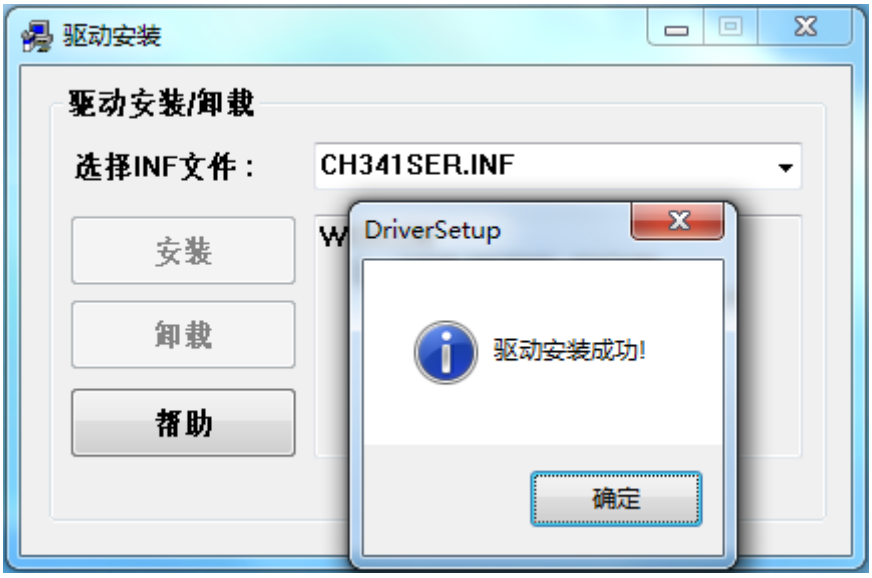


图 3.4.1.2 CH340 驱动安装

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：

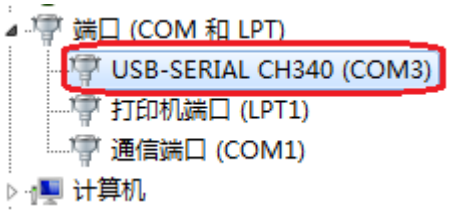


图 3.4.1.3 USB 串口

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，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：

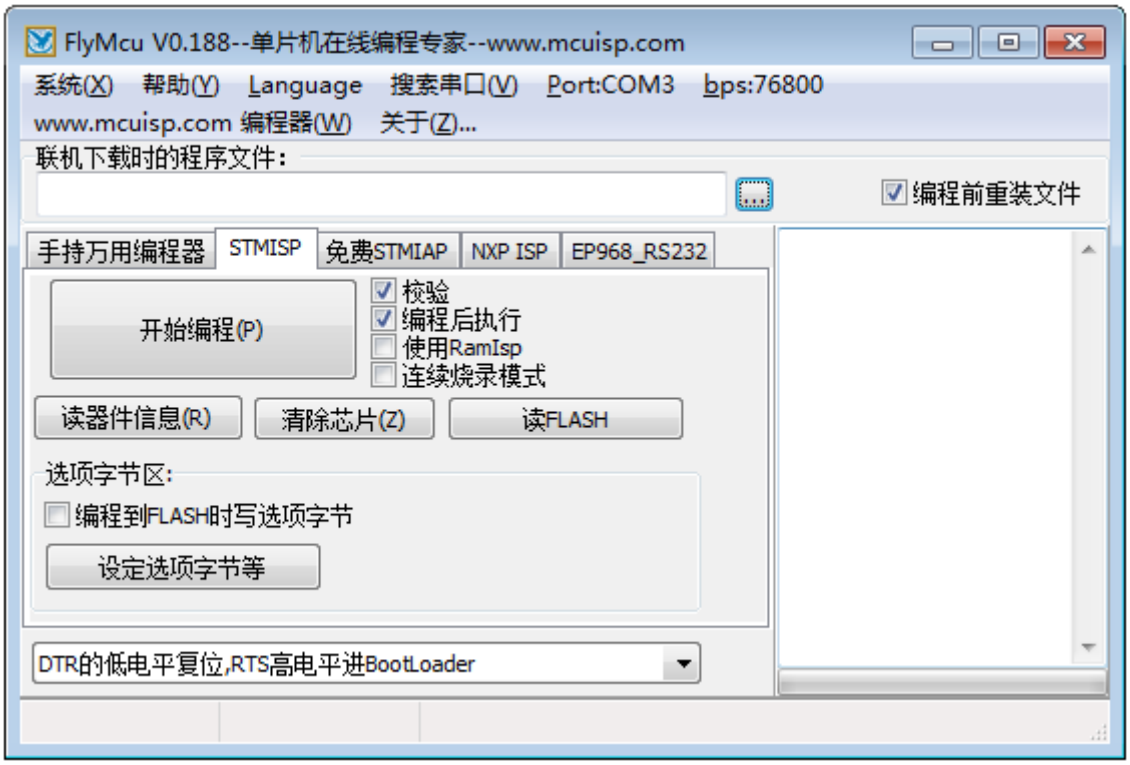


图 3.4.1.4 flymcu 启动界面

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：

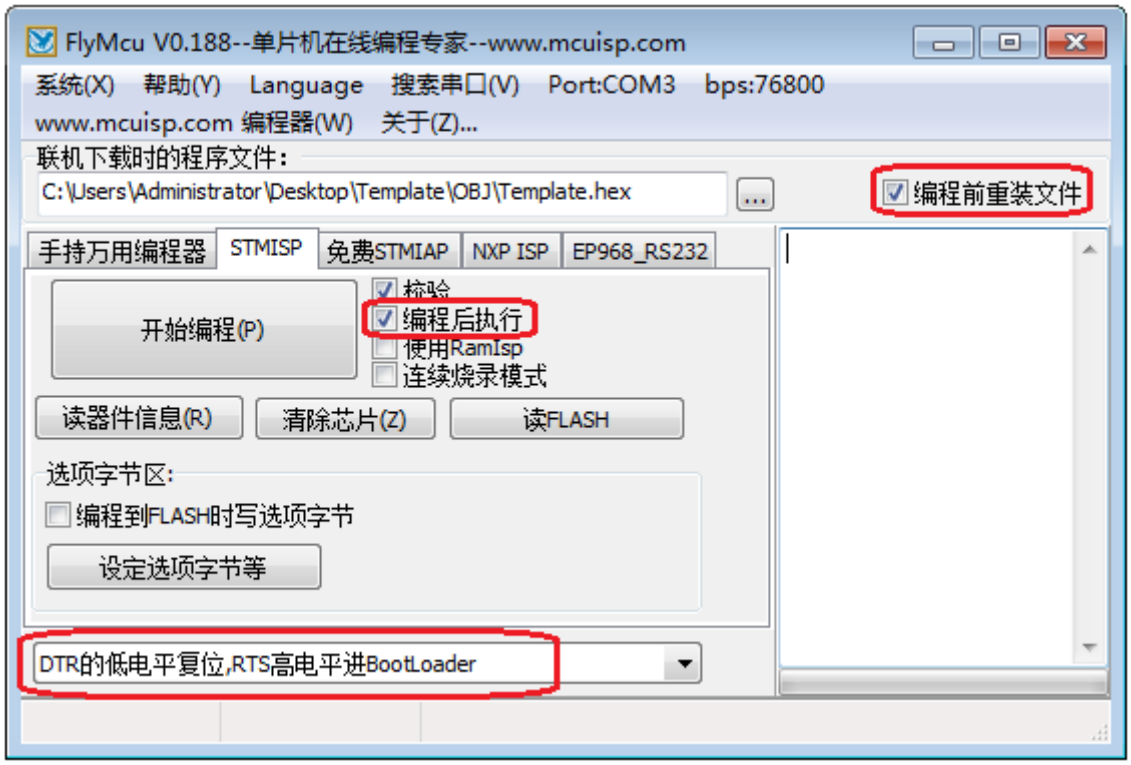


图 3.4.1.5 flymcu 设置

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：

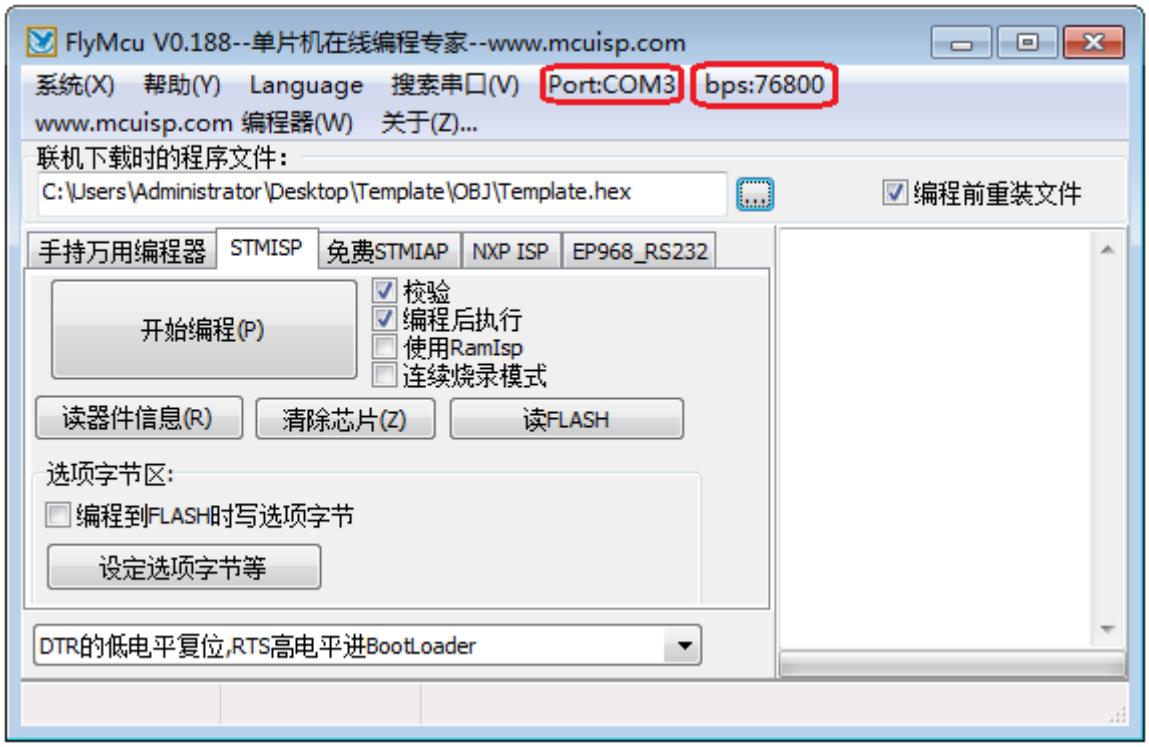


图 3.4.1.6 CH340 虚拟串口

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：

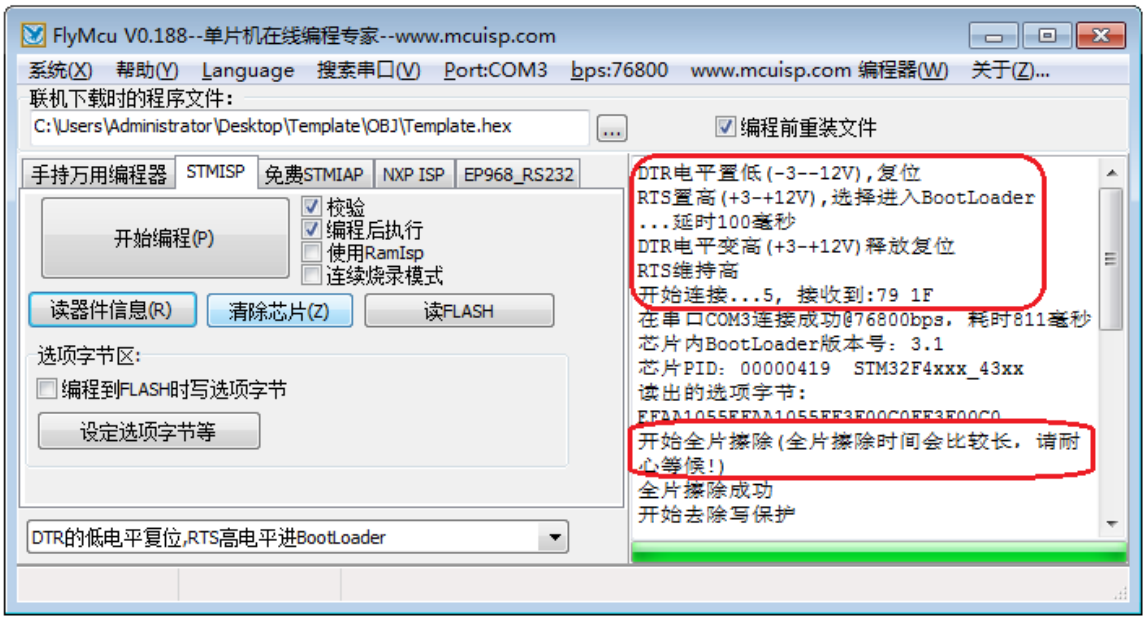


图 3.4.1.7 下载完成

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 STM32F1 的每次下载都需要整片擦除，而 STM32F1 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 ST-LINK 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 ST-LINK 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 精英 STM32F1 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 使用 ST-LINK 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 战舰 STM32 开发板上验证了我们程序的正确性。这个代码比较简单，所以不需要硬件调试，我们直接就一次成功了。可是，如果你的代码工程比较大，难免存在一些 bug，这时，就有必要通过硬件调试来解决问题了。

串口只能下载代码，并不能实时跟踪调试，而利用调试工具，比如 ST-LINK，JLINK 和 ULINK 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 ST-LINK 为例，说说如何在线调试 STM32F103。

ST-LINK 支持 JTAG 和 SWD，同时 STM32F103 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。

ST-LINK 的驱动安装比较简单，我们在这里就不说了，请大家参考：光盘→6，软件资料→1，软件→ST LINK 驱动及教程 文件夹里面的《STLINK 调试补充教程.pdf》自行安装。

在安装 ST-LINK 的驱动之后，我们接上 ST-LINK，并用灰排线连接 ST LINK 和开发板的 JTAG 接口，打开之前 3.3 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 ST-Link Debugger，如图 3.4.2.1 所示：

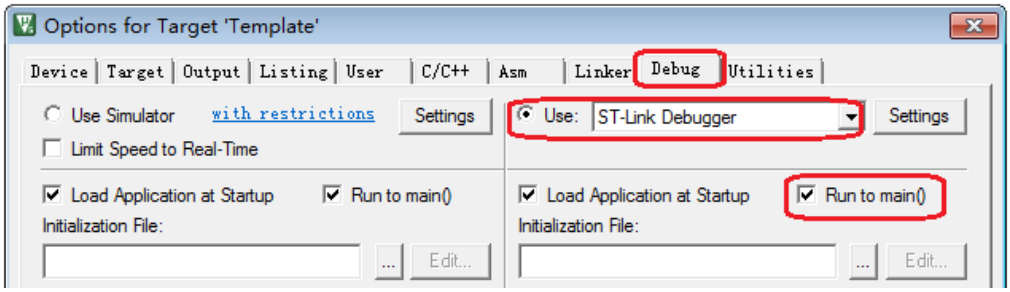


图 3.4.2.1 Debug 选项卡设置

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_stm32f103xe.s 文件的 Reset_Handler，再跳到 main 函数。

然后我们点击 Settings，设置 ST-LINK 的一些参数，如图 3.4.2.2 所示：

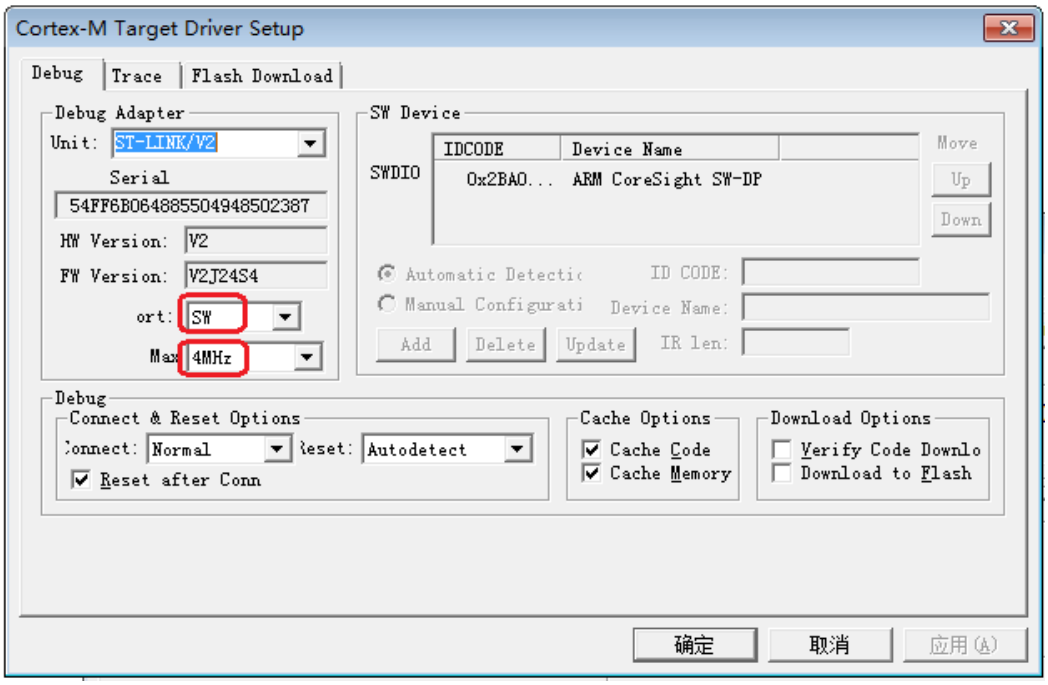


图 3.4.2.2ST-LINK 模式设置

图 3.4.2.2 中，我们使用 ST-LINK 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**。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 我们设置为最大：4Mhz（需要更新固件，否则最大只能到 1.8Mhz），这里，如果你的 USB 数据线比较差，那么可能会出问题，此时，你可以通过降低这里的速率来试试。

单击 OK，完成此部分设置，接下来我们还需要在 Utilities 选项卡里面设置下载时的目标编程器，如图 3.4.2.3 所示：

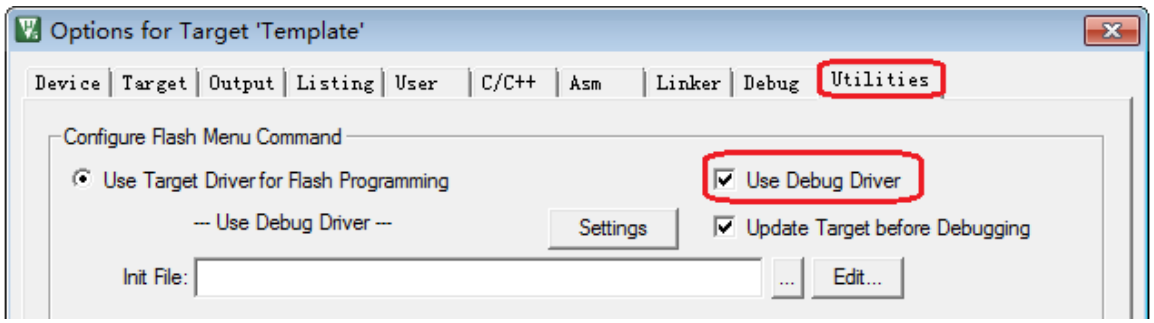


图 3.4.2.3 FLASH 编程器选择

图 3.4.2.3 中，我们直接勾选 Use Debug Driver，即和调试一样，选择 ST-LINK 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：

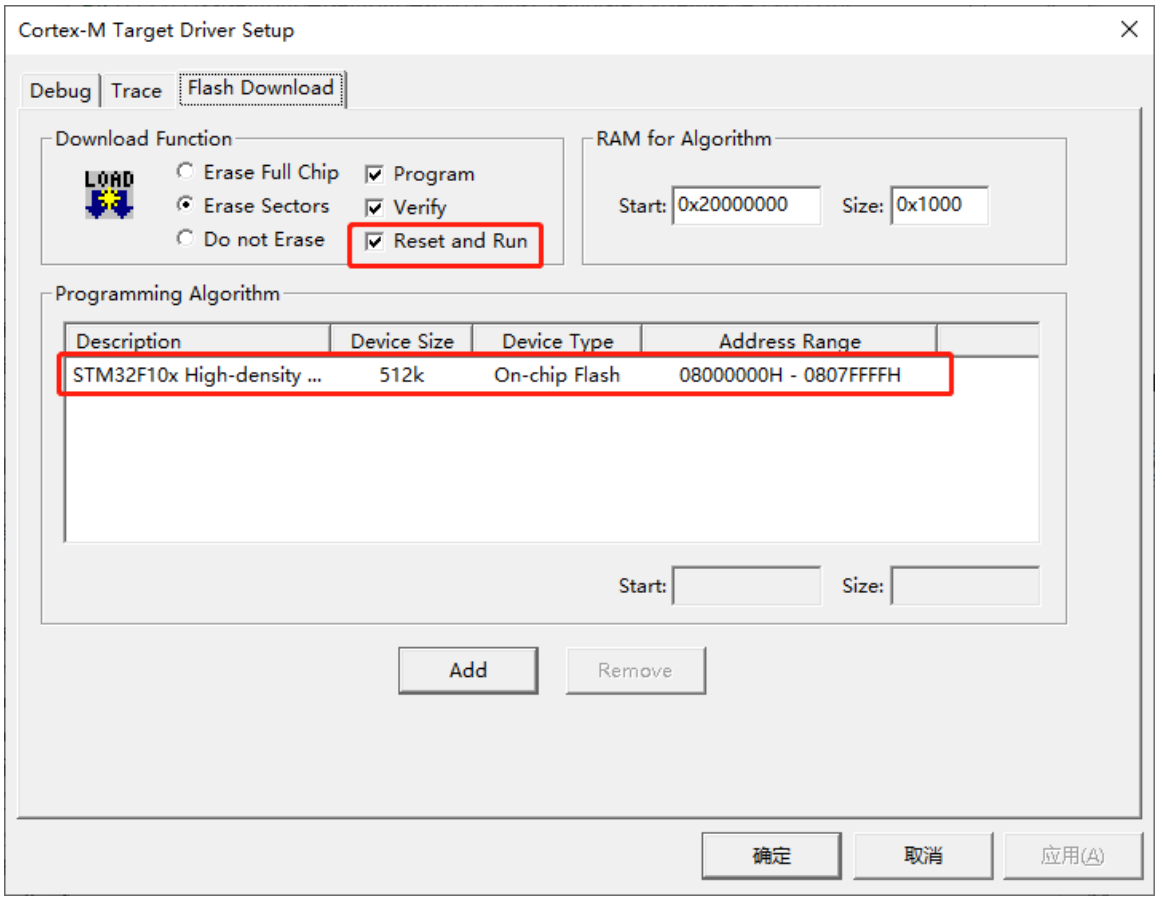


图 3.4.2.4　编程设置

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 STM32F103ZET6，FLASH 容量为 512K 字节，所以 Programming Algorithm 里面默认会有 512K

型号的 STM32F10x FLASH 算法。**特别提醒：**这里的 512K flash 算法，不仅仅针对 512K 容量的 STM32F103，对于小于 512K 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 ST-LINK 下载代码和调试代码。

配置好 ST-LINK 之后，使用 ST-LINK 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：

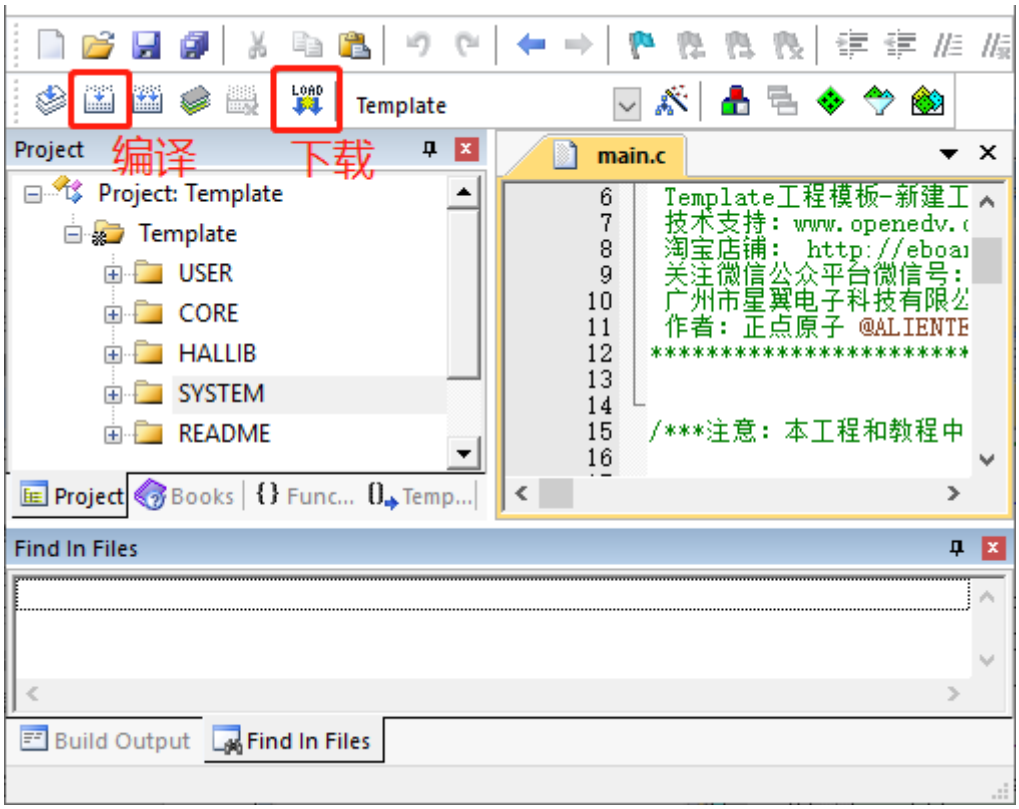


图 3.4.2.5 编译和下载按钮

接下来我们看看用 ST-LINK 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：

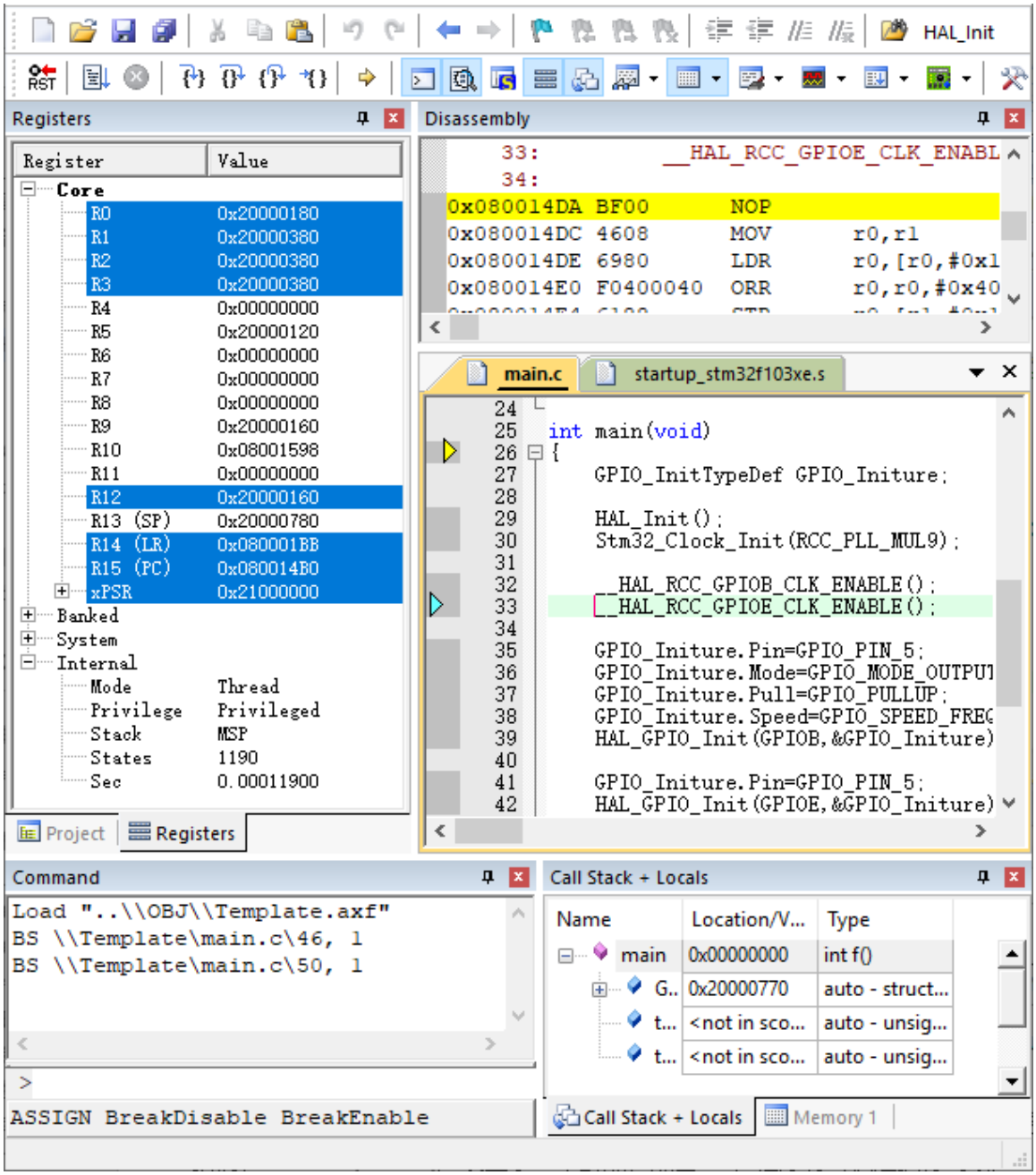


图 3.4.2.6 开始仿真

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(0x7FFFFF)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：

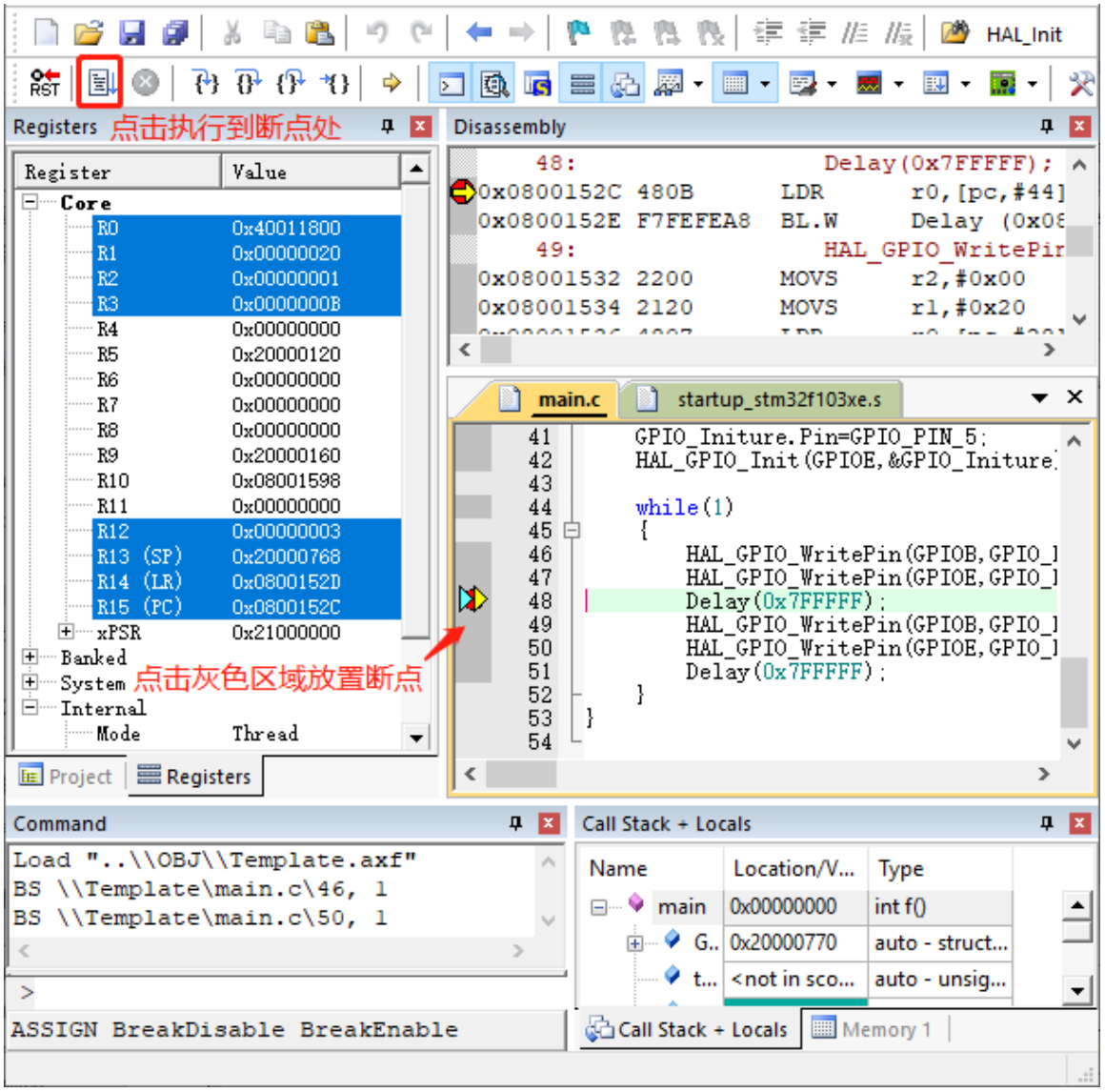


图 3.4.2.7　程序运行到断点处

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，前提是你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，就可以使程序停止下来，进入到单步调试状态。

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

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

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

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

汇编窗口：通过该按钮，就可以查看汇编代码，这对分析程序很有用。

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。

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 3.2.23 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：

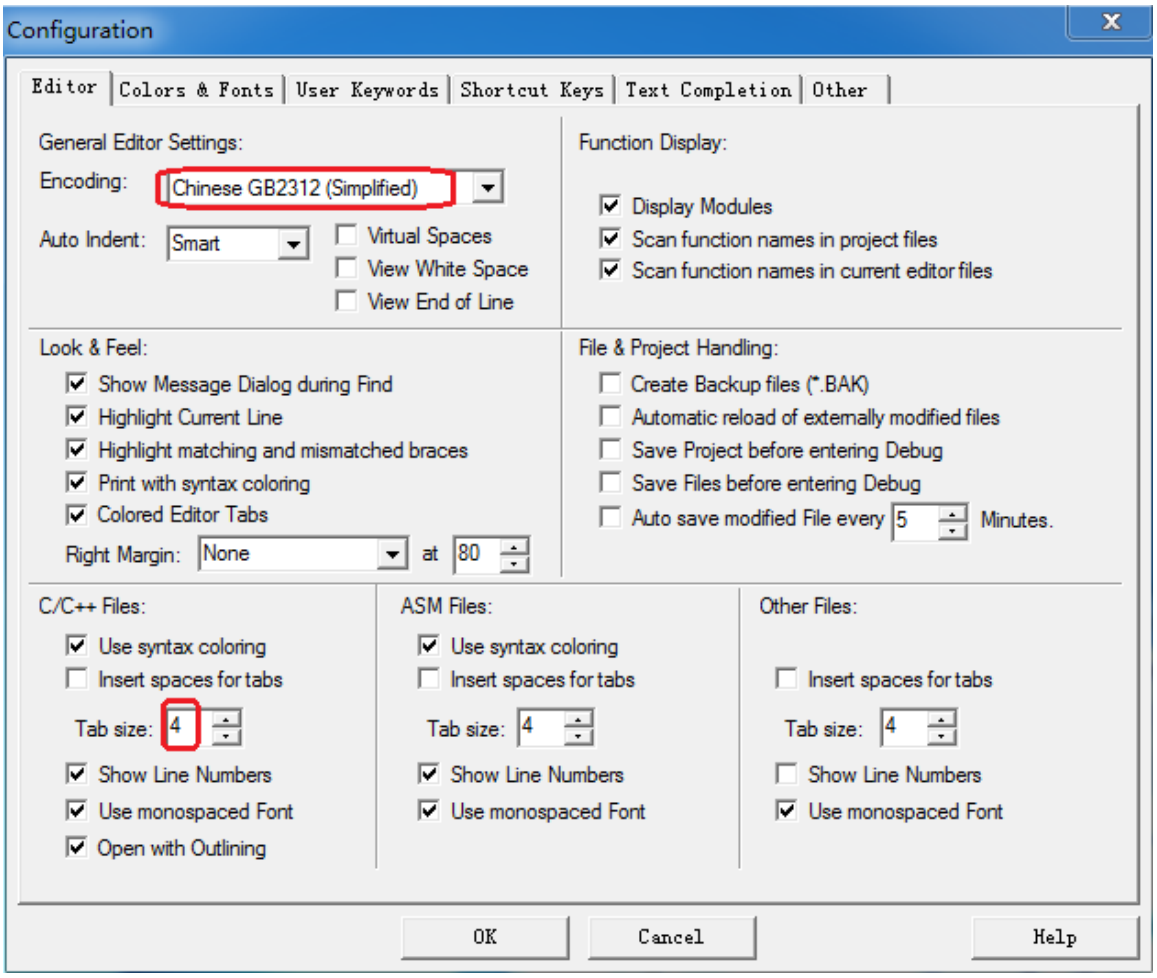


图 3.5.1.1 置对话框

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子体和颜色了。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：

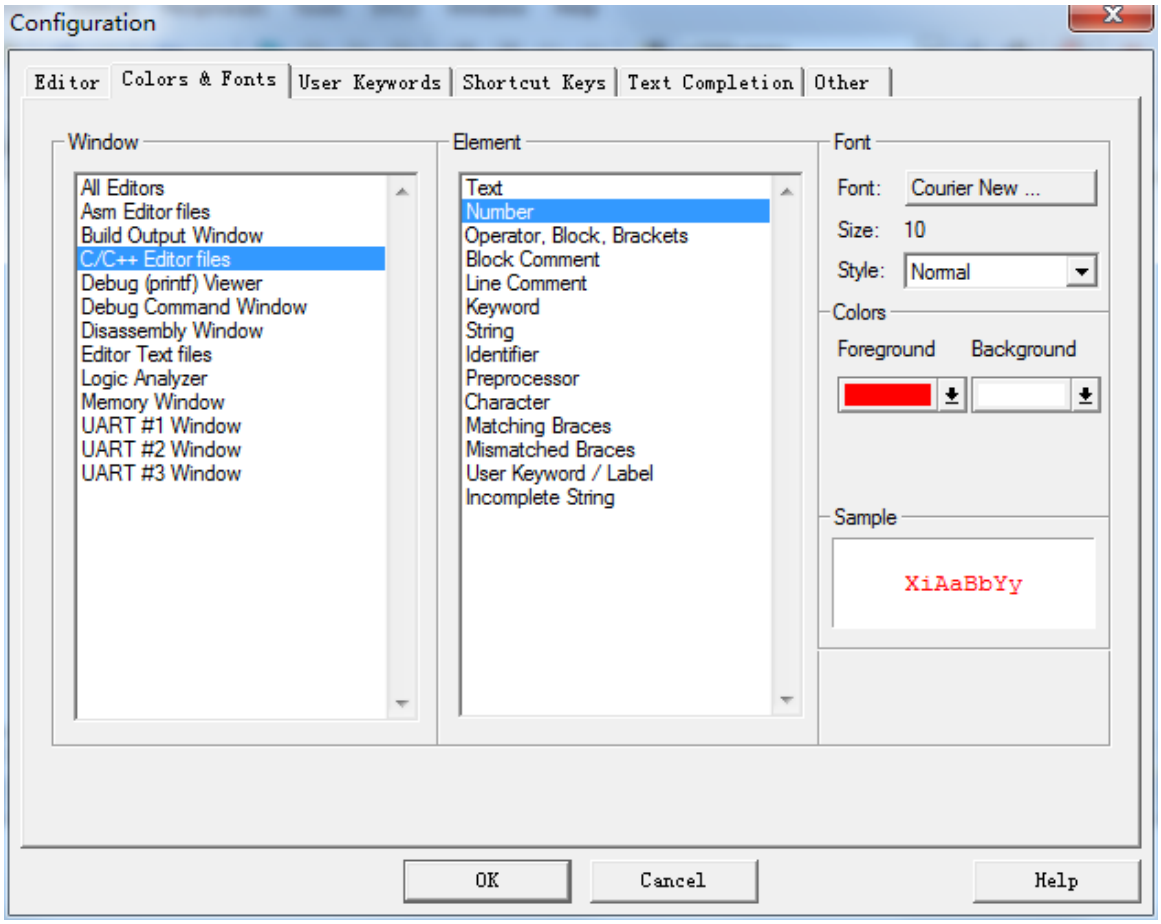


图 3.5.1.2 Colors&Fonts 选项卡

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：

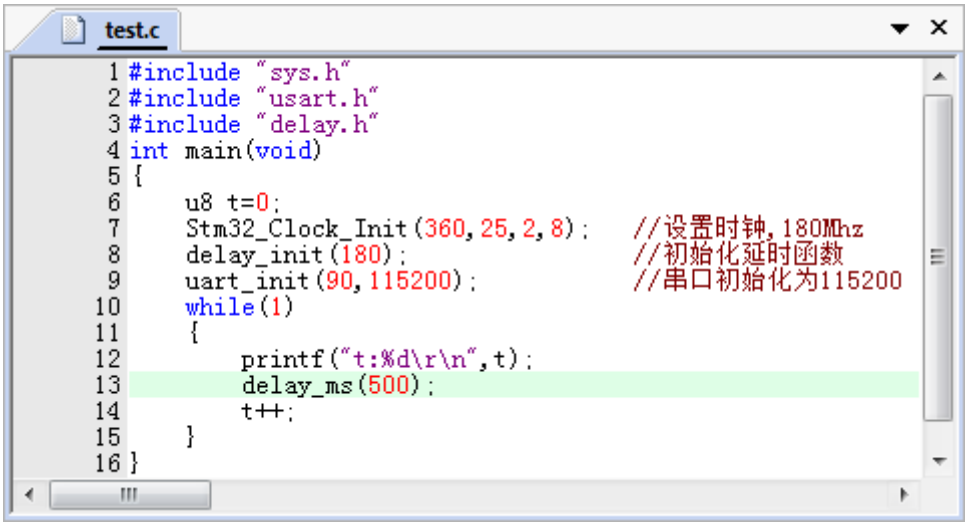


图 3.5.1.3 设置完后显示效果

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 u8，还是黑色的，这是一个用户自定义的关键字，为什么不显示蓝色（假定刚刚已经设置了用户自定义关键字颜色为蓝色）呢？这就又要回到我们刚刚的配置对话框了，单这次我们要选择 User Keywords 选项卡，同样选择：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：

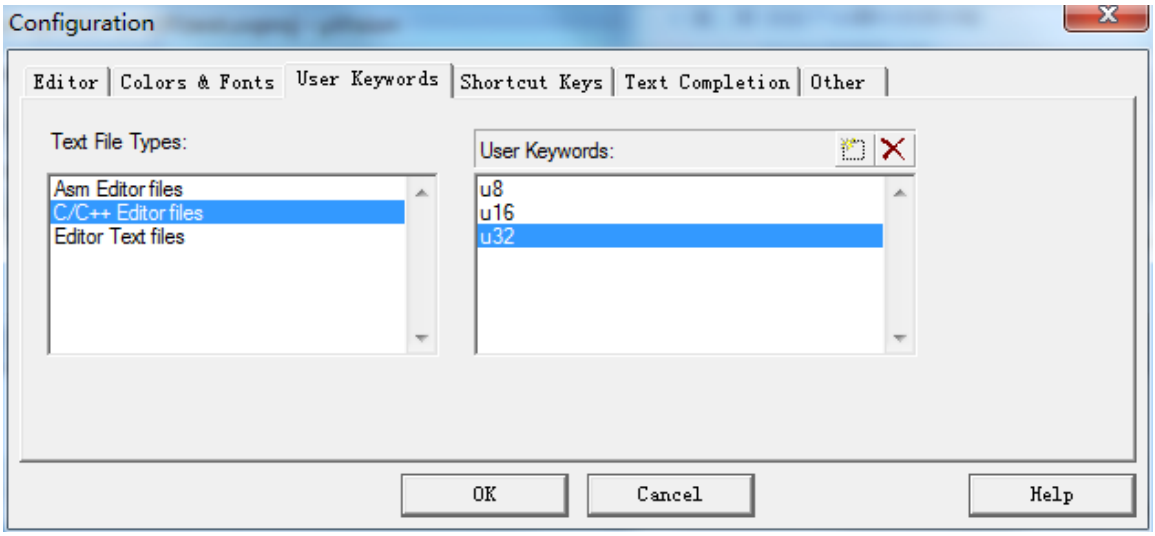


图 3.5.1.4 用户自定义关键字

图 3.5.1.4 中我定义了 u8、u16、u32 等 3 个关键字，这样在以后的代码编辑里面只要出现这三个关键字，肯定就会变成蓝色。点击 OK，再回到主界面，可以看到 u8 变成了蓝色了，如图 3.5.1.5 示：

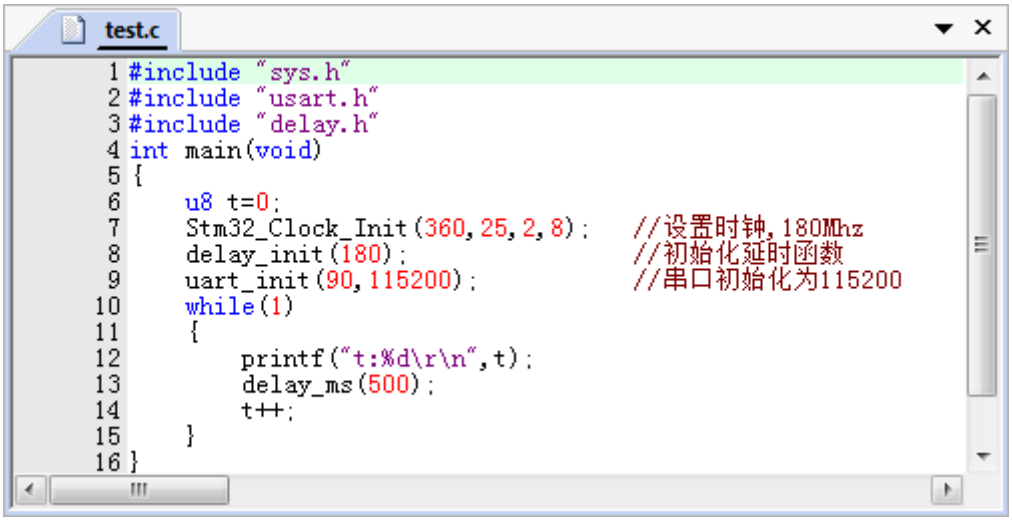


图 3.5.1.5 设置完后显示效果

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

MDK5 支持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：

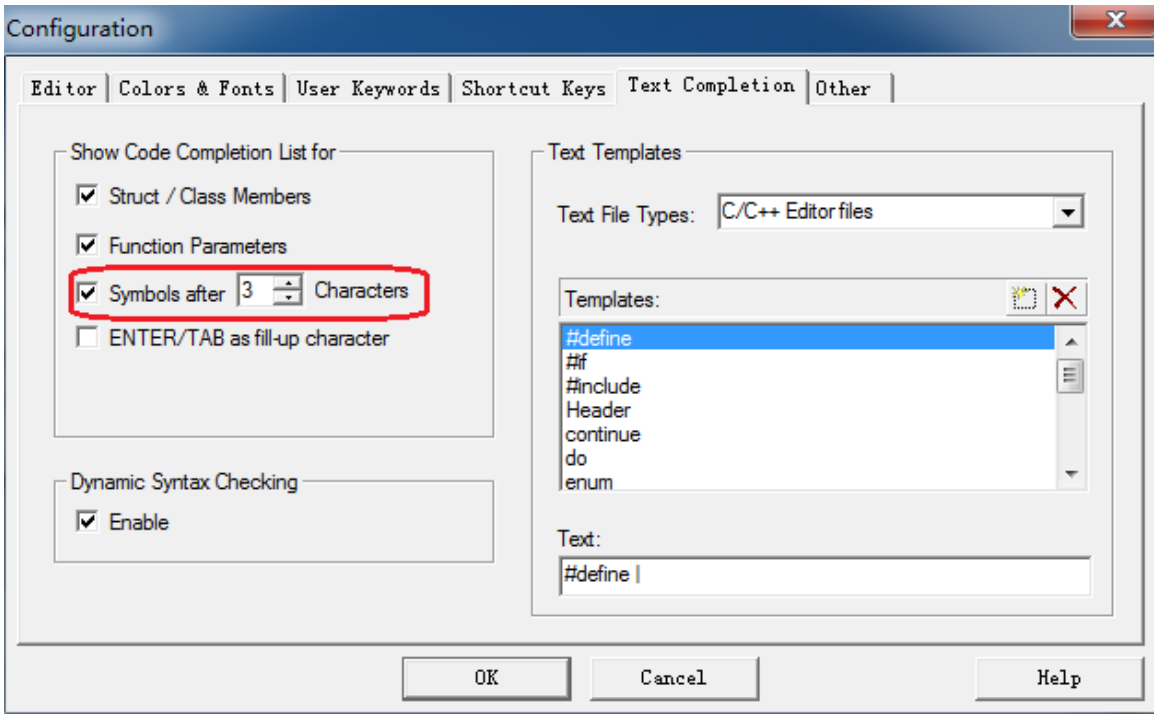


图 3.5.2.1 Text Completion 选项卡设置

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：

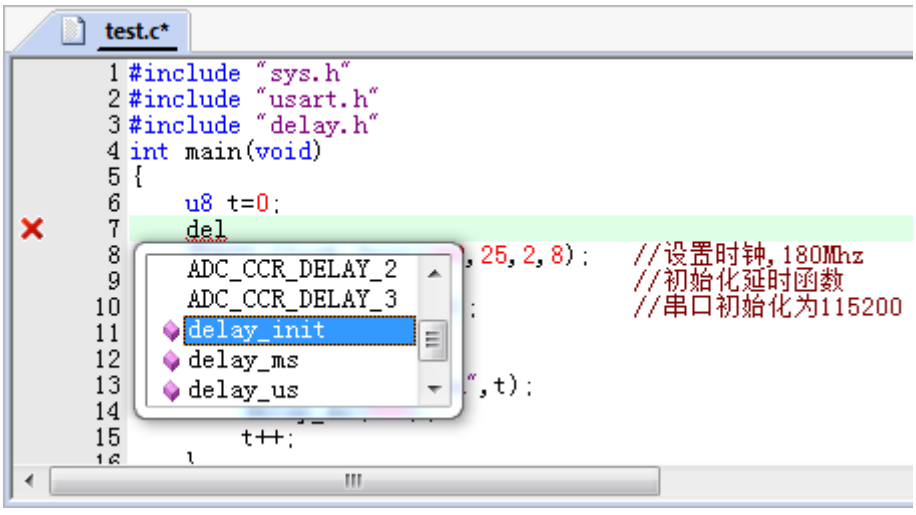


图 3.5.2.2 代码提示

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：

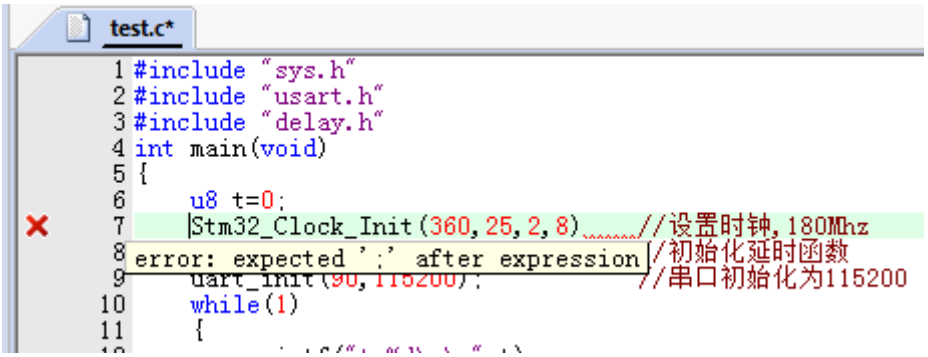


图 3.5.2.3 语法动态检测功能

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/查看技巧

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，如果你经常编写程序，对这个键一定再熟悉不过了。MDK 的 TAB 键还可以支持块操作：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：

```
62  void USART1_IRQHandler(void)
63  {
64  u8 res;
65  #ifdef OS_CRITICAL_METHOD   //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
66  OSIntEnter();
67  #endif
68  if(USART1->SR&(1<<5))//接收到数据
69  {
70  res=USART1->DR;
71  if((USART_RX_STA&0x8000)==0)//接收未完成
72  {
73  if(USART_RX_STA&0x4000)//接收到了0x0d
74  {
75  if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
76  else USART_RX_STA|=0x8000;  //接收完成了
77  }else //还没收到0X0D
78  {
79  if(res==0x0d)USART_RX_STA|=0x4000;
80  else
81  {
82  USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
83  USART_RX_STA++;
84  if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
85  }
86  }
87  }
88  }
89  #ifdef OS_CRITICAL_METHOD   //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
90  OSIntExit();
91  #endif
92  }
```

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：

```
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(USART1->SR&(1<<5))//接收到数据
    {
    res=USART1->DR;
    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_CRITICAL_METHOD   //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
    OSIntExit();
    #endif
}
```

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

```
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(USART1->SR&(1<<5))//接收到数据
    {
        res=USART1->DR;
        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_CRITICAL_METHOD   //如果OS_CRITICAL_METHOD定义了,说明使用ucosII了.
    OSIntExit();
#endif
}
```

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 ：

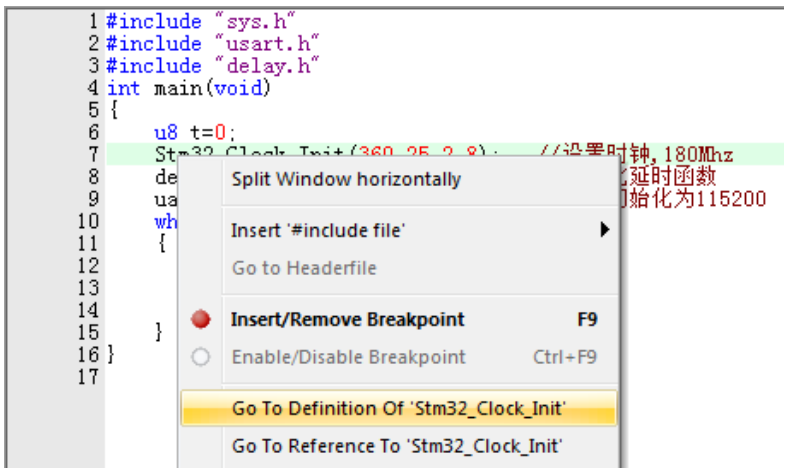


图 3.5.3.4 快速定位

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：

```
//系统时钟初始化函数
//plln:主PLL倍频系数(PLL倍频),取值范围:64~432.
//pllm:主PLL和音频PLL分频系数(PLL之前的分频),取值范围:2~63.
//pllp:系统时钟的主PLL分频系数(PLL之后的分频),取值范围:2,4,6,8.(仅限这4个值!)
//pllq:USB/SDIO/随机数产生器等的主PLL分频系数(PLL之后的分频),取值范围:2~15.
void Stm32_Clock_Init(u32 plln,u32 pllm,u32 pllp,u32 pllq)
{
    RCC->CR|=0x00000001;          //设置HISON,开启内部高速RC振荡
    RCC->CFGR=0x00000000;         //CFGR清零
    RCC->CR&=0xFEF6FFFF;          //HSEON,CSSON,PLLON清零
    RCC->PLLCFGR=0x24003010;      //PLLCFGR恢复复位值
    RCC->CR&=~(1<<18);            //HSEBYP清零,外部晶振不旁路
    RCC->CIR=0x00000000;          //禁止RCC时钟中断
    Sys_Clock_Set(plln,pllm,pllp,pllq);//设置时钟
    //配置向量表
#ifdef  VECT_TAB_RAM
    MY_NVIC_SetVectorTable(1<<29,0x0);
#else
    MY_NVIC_SetVectorTable(0,0x0);
#endif
}
```

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。

很多时候，我们利用 Go to Definition 看完函数/变量的定义后，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：

```
214  //系统时钟初始化函数
215  //plln:主PLL倍频系数(PLL倍频),取值范围:64~432.
216  //pllm:主PLL和音频PLL分频系数(PLL之前的分频),取值范围:2~63.
217  //pllp:系统时钟的主PLL分频系数(PLL之后的分频),取值范围:2,4,6,8.(仅限这4个值!)
218  //pllq:USB/SDIO/随机数产生器等的主PLL分频系数(PLL之后的分频),取值范围:2~15.
219  void Stm32_Clock_Init(u32 plln,u32 pllm,u32 pllp,u32 pllq)
220  {
221      RCC->CR|=0x00000001;          //设置HISON,开启内部高速RC振荡
222      RCC->CFGR=0x00000000;         //CFGR清零
223      RCC->CR&=0xFEF6FFFF;          //HSEON,CSSON,PLLON清零
224      RCC->PLLCFGR=0x24003010;      //PLLCFGR恢复复位值
225      RCC->CR&=~(1<<18);            //HSEBYP清零,外部晶振不旁路
226      RCC->CIR=0x00000000;          //禁止RCC时钟中断
227      Sys_Clock_Set(plln,pllm,pllp,pllq);//设置时钟
228      //配置向量表
229  #ifdef  VECT_TAB_RAM
230      MY_NVIC_SetVectorTable(1<<29,0x0);
231  #else
232      MY_NVIC_SetVectorTable(0,0x0);
233  #endif
234  }
```

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：

```
214  //系统时钟初始化函数
215  //plln:主PLL倍频系数(PLL倍频),取值范围:64~432.
216  //pllm:主PLL和音频PLL分频系数(PLL之前的分频),取值范围:2~63.
217  //pllp:系统时钟的主PLL分频系数(PLL之后的分频),取值范围:2,4,6,8.(仅限这4个值!)
218  //pllq:USB/SDIO/随机数产生器等的主PLL分频系数(PLL之后的分频),取值范围:2~15.
219  void Stm32_Clock_Init(u32 plln,u32 pllm,u32 pllp,u32 pllq)
220  {
221  //  RCC->CR|=0x00000001;          //设置HISON,开启内部高速RC振荡
222  //  RCC->CFGR=0x00000000;         //CFGR清零
223  //  RCC->CR&=0xFEF6FFFF;          //HSEON,CSSON,PLLON清零
224  //  RCC->PLLCFGR=0x24003010;      //PLLCFGR恢复复位值
225  //  RCC->CR&=~(1<<18);            //HSEBYP清零,外部晶振不旁路
226  //  RCC->CIR=0x00000000;          //禁止RCC时钟中断
227  //  Sys_Clock_Set(plln,pllm,pllp,pllq);//设置时钟
228  //  //配置向量表
229  //#ifdef  VECT_TAB_RAM
230  //  MY_NVIC_SetVectorTable(1<<29,0x0);
231  //#else
232  //  MY_NVIC_SetVectorTable(0,0x0);
233  //#endif
234  }
```

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：

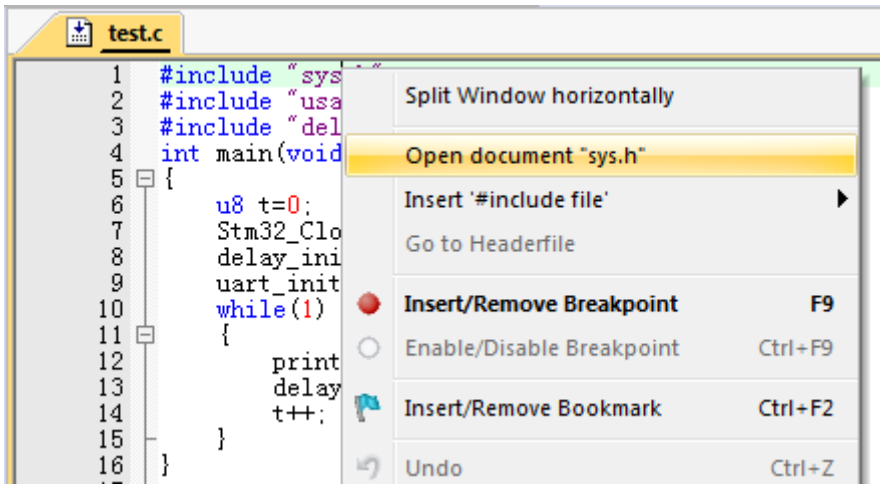


图 3.5.4.1 快速打开头文件

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：

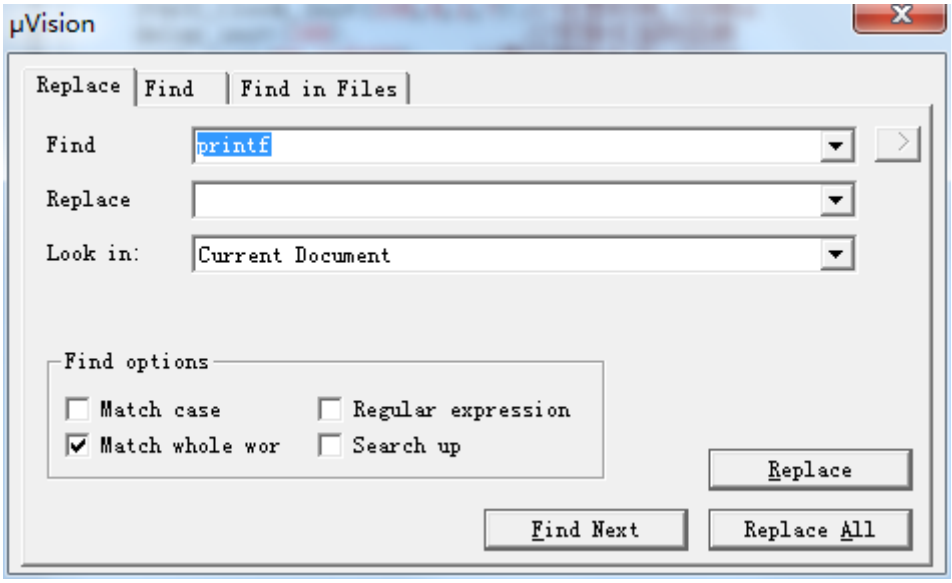


图 3.5.4.2 替换文本

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：

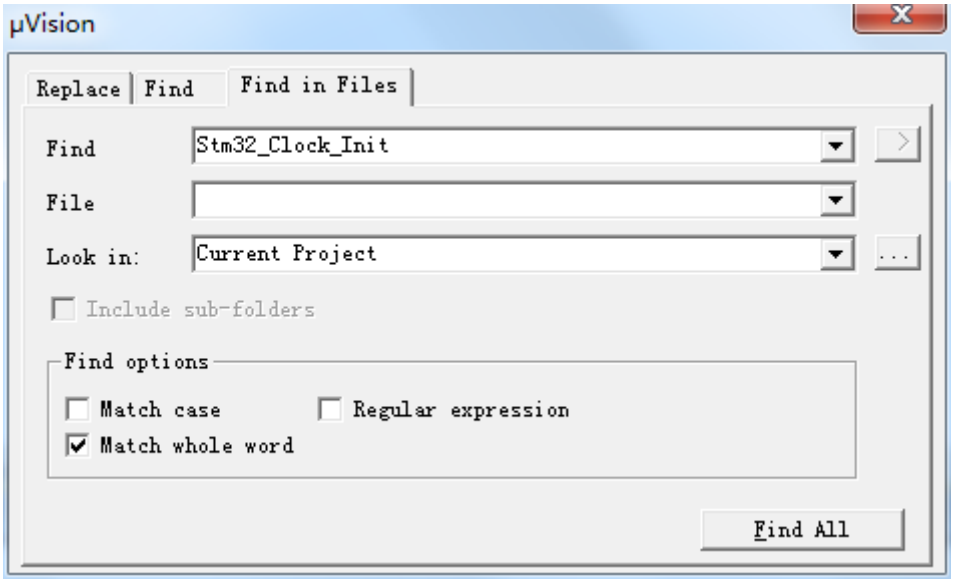


图 3.5.4.3 跨文件查找

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：

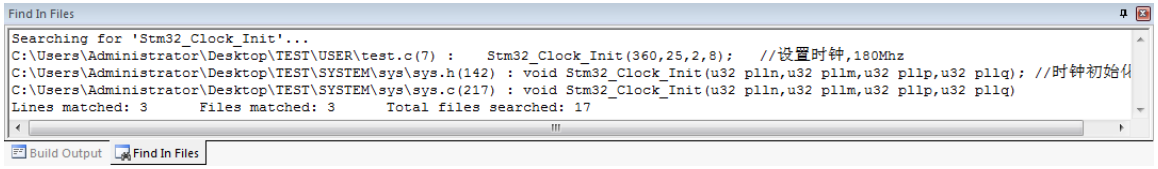


图 3.5.4.4 查找结果

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 STM32F1 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 STM32F1 系统架构
- 4.3 STM32F103 时钟系统
- 4.4 IO 引脚复用器和映射
- 4.5 STM32F1 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->ODR 的状态，可以先对寄存器的值进行&清零操作

```
GPIOA->ODR &=0XFF0F;  //将第 4-7 位清 0
```

然后再与需要设置的值进行|或运算

```
GPIOA->ODR |=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

```
GPIOA->ODR| = 1 << 5;
```

这个操作就是将 ODR 寄存器的第 5 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 5 位设置为 1，其他位的值不变。如果你写成

```
GPIOA->ODR =0x0020;
```

这样的代码可读性非常差同时也不好重用。

3） ~取反操作使用技巧

例如 GPIOA->ODR 寄存器的每一位都用来设置一个 IO 口的输出状态，某个时刻我们希望去设置某一位的值为 0，同时其他位都为 1，简单的作法是直接给寄存器设置一个值：

```
GPIOA->ODR =0xFFF7;
```

这样的作法设置第 3 位为 0，但是这样的写法可读性很差。看看如果我们使用取反操作怎么实现：

```
GPIOA->ODR= (uint16_t)~(1<<3);
```

看这行代码应该很容易明白，我们设置的是 ODR 寄存器的第 3 位为 0，其他位为 1，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 HSI_VALUE    ((uint32_t)16000000)
```

定义标识符 HSI_VALUE 的值为 16000000。这样我们就可以在代码中直接使用标识符 HSI_VALUE，而不用直接使用常量 16000000，同时也很方便我们修改 HSI_VALUE 的值。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 和 #if defined 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_hal_conf.h 这个头文件中会看到这样的语句：

```
#ifdef  HAL_GPIO_MODULE_ENABLED
  #include "stm32f1xx_hal_gpio.h"
#endif
```

这段代码的作用是判断宏定义标识符 HAL_GPIO_MODULE_ENABLED 是否被定义，如果被定义了，那么就引入头文件 stm32f1xx_hal_gpio.h。

对于条件编译，还有个常用的格式，如下：

```
#if defined XXX1
  程序段 1
#elif defined XXX2
  程序段 2
    …
#elif defined XXXn
  程序段 n
    …
#endif
```

这种写法的作用实际跟 ifdef 很相似，不同的是 ifdef 只能在两个选择中判断是否定义，而 if defined 可以在多个选择中判断是否定义。

条件编译也是 c 语言的基础知识，这里就给大家讲解到这里，不懂的大家可以查看在网上搜索相关资料学习。

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 G_TYPE {
 uint32_t Pin;
```

```
    uint32_t Mode;
    uint32_t Speed;
}GPIOA，GPIOB;
```

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 G_TYPE GPIOA,GPIOB;

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 GPIOA 的成员 Mode，方法是：GPIOA. Mode;

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 G_TYPE *GPIOC; //定义结构体指针变量 GPIOC;

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 GPIOC 结构体指针指向的结构体的成员变量 Speed, 方法是：

```
GPIOC-> Speed;
```

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 IO 口。它的初始化状态是由几个属性来决定的，比如模式，速度等。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    HAL_GPIO_Init (uint32_t Pin, uint32_t Mode, uint32_t Speed);
```

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，重新加入上下拉 Pull 这个入口参数。于是我们的定义被修改为：

```
void    HAL_GPIO_Init (uint32_t Pin, uint32_t Mode, uint32_t Speed,uint32_t Pull);
```

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。上面的函数中 Pin，Mode，Speed 和 Pull 这些参数，他们对于 GPIO 而言，是一个有机整体，都是来设置 IO 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 Pin;
  uint32_t Mode;
  uint32_t Pull;
  uint32_t Speed;
}GPIO_InitTypeDef;
```

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

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

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 STM32F1 系统架构

STM32 的系统架构比 51 单片机就要强大很多了。STM32 系统架构的知识可以在《STM32 中文参考手册 V10》的 P25~28 有讲解，这里我们也把这一部分知识抽取出来讲解，是为了大家在学习 STM32 之前对系统架构有一个初步的了解。这里的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，还需要在网上搜索其他资料学习学习。

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

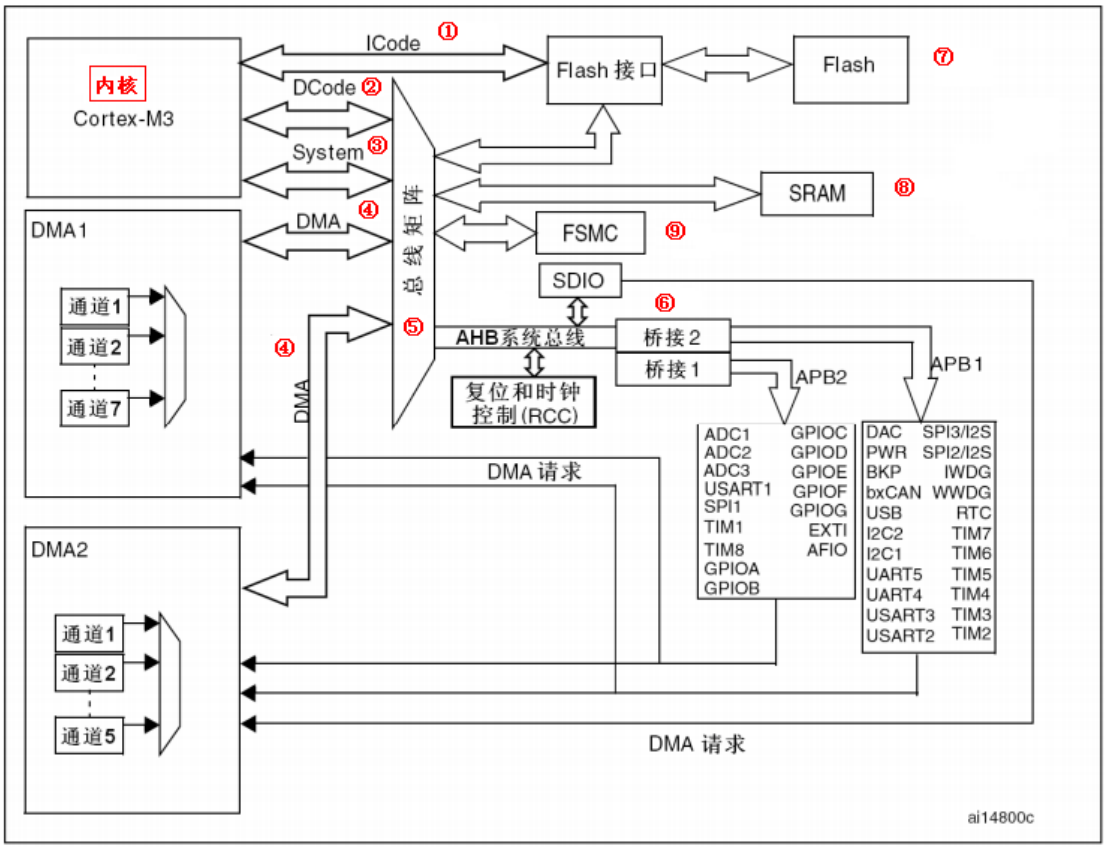


图 4.2.1STM32 系统架构图

STM32 主系统主要由四个驱动单元和四个被动单元构成。

四个驱动单元是：

内核 DCode 总线;

系统总线;

通用 DMA1;

通用 DMA2;

四被动单元是：

AHB 到 APB 的桥：连接所有的 APB 设备；

内部 FlASH 闪存；

内部 SRAM；

FSMC;

下面我们具体讲解一下图中几个总线的知识：

① ICode 总线：该总线将 M3 内核指令总线和闪存指令接口相连，指令的预取在该总线上面完成。

② DCode 总线：该总线将 M3 内核的 DCode 总线与闪存存储器的数据接口相连接，常量加载和调试访问在该总线上面完成。

③ 系统总线：该总线连接 M3 内核的系统总线到总线矩阵，总线矩阵协调内核和 DMA 间访问。

④ DMA 总线：该总线将 DMA 的 AHB 主控接口与总线矩阵相连，总线矩阵协调 CPU 的 DCode 和 DMA 到 SRAM,闪存和外设的访问。

⑤ 总线矩阵：总线矩阵协调内核系统总线和 DMA 主控总线之间的访问仲裁，仲裁利用轮换算法。

⑥ AHB/APB 桥:这两个桥在 AHB 和 2 个 APB 总线间提供同步连接，APB1 操作速度限于 36MHz,APB2 操作速度全速。

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

4.3 STM32F103 时钟系统

STM32F1 时钟系统的知识在《STM32 中文参考手册 V10》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。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 STM32F103 时钟树概述
- 4.3.2 STM32F103 时钟初始化配置
- 4.3.3 STM32F103 时钟使能和配置

4.3.1 STM32F103 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。STM32F103 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 STM32F103 的时钟系统图：

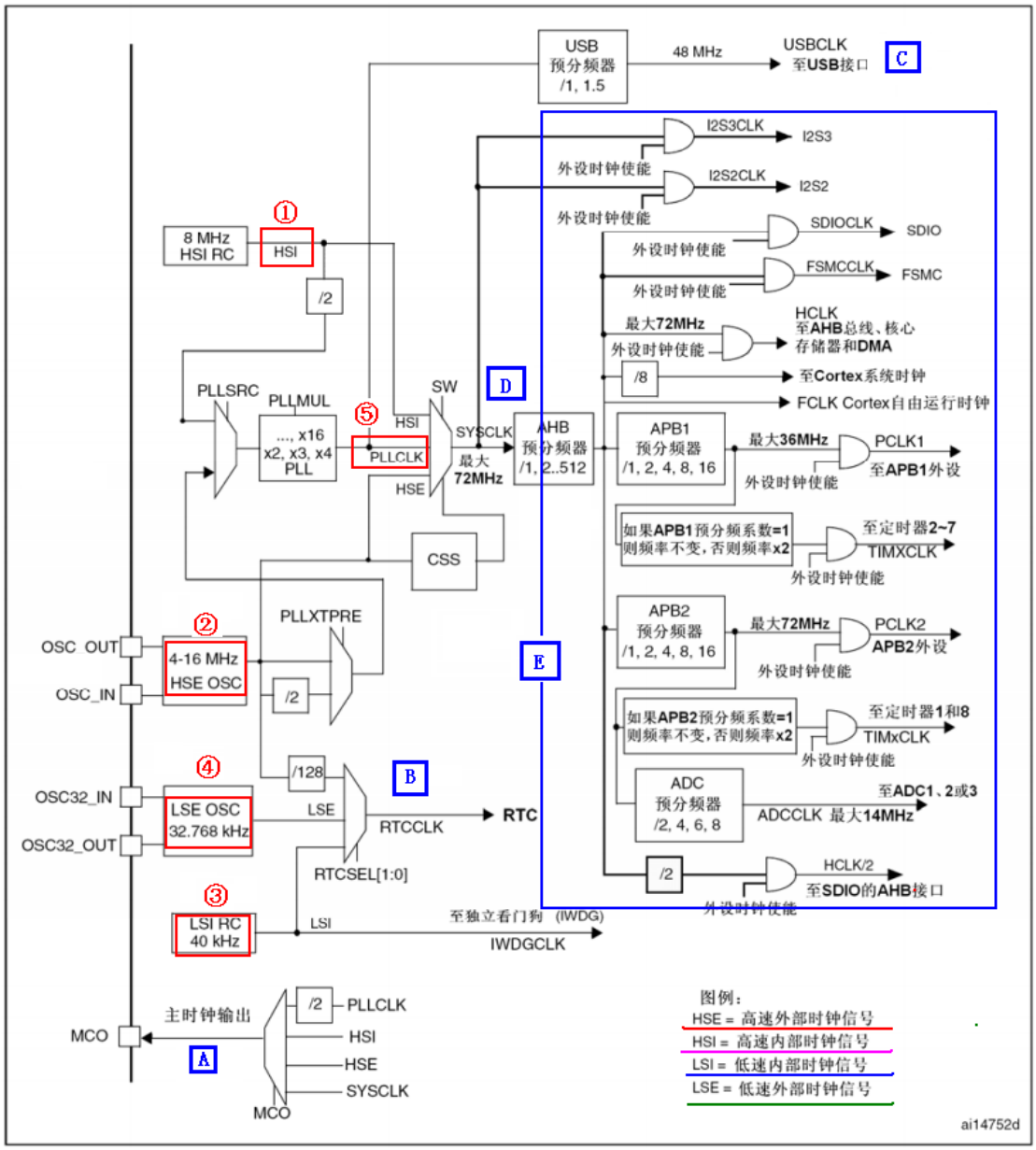


图 4.3.1.1 STM32F103 时钟系统图

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

①、HSI 是高速内部时钟，RC 振荡器，频率为 8MHz。

②、HSE 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz~16MHz。我们的开发板接的是 8M 的晶振。

③、LSI 是低速内部时钟，RC 振荡器，频率为 40kHz。独立看门狗的时钟源只能是 LSI，同时 LSI 还可以作为 RTC 的时钟源。

④、LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体。这个主要是 RTC 的时钟源。

⑤、PLL 为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HSE 或者 HSE/2。倍频可选择为 2~16 倍，但是其输出频率最大不得超过 72MHz。

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

A. MCO 是 STM32 的一个时钟输出 IO(PA8)，它可以选择一个时钟信号输出，可以选择为 PLL 输出的 2 分频、HSI、HSE、或者系统时钟。这个时钟可以用来给外部其他系统提供时钟源。

B. 这里是 RTC 时钟源，从图上可以看出，RTC 的时钟源可以选择 LSI，LSE，以及 HSE 的 128 分频。

C. 从图中可以看出 C 处 USB 的时钟是来自 PLL 时钟源。STM32 中有一个全速功能的 USB 模块，其串行接口引擎需要一个频率为 48MHz 的时钟源。该时钟源只能从 PLL 输出端获取，可以选择为 1.5 分频或者 1 分频，也就是，当需要使用 USB 模块时，PLL 必须使能，并且时钟频率配置为 48MHz 或 72MHz。

D. D 处就是 STM32 的系统时钟 SYSCLK，它是供 STM32 中绝大部分部件工作的时钟源。系统时钟可选择为 PLL 输出、HSI 或者 HSE。系统时钟最大频率为 72MHz，当然你也可以超频，不过一般情况为了系统稳定性是没有必要冒风险去超频的。

E. 这里的 E 处是指其他所有外设了。从时钟图上可以看出，其他所有外设的时钟最终来源都是 SYSCLK。SYSCLK 通过 AHB 分频器分频后送给各模块使用。这些模块包括：

①、AHB 总线、内核、内存和 DMA 使用的 HCLK 时钟。

②、通过 8 分频后送给 Cortex 的系统定时器时钟，也就是 systick 了。

③、直接送给 Cortex 的空闲运行时钟 FCLK。

④、送给 APB1 分频器。APB1 分频器输出一路供 APB1 外设使用(PCLK1，最大频率 36MHz)，另一路送给定时器(Timer)2、3、4 倍频器使用。

⑤、送给 APB2 分频器。APB2 分频器分频输出一路供 APB2 外设使用(PCLK2，最大频率 72MHz)，另一路送给定时器(Timer)1 倍频器使用。

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

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

4.3.2 STM32F103 时钟系统配置

上一小节我们对 STM32F103 时钟树进行了详细讲解，接下来我们来讲解通过 STM32F1 的 HAL 库进行 STM32F103 时钟系统配置步骤。实际上，STM32F1 的时钟系统配置也可以通过图形化配置工具 STM32CubeMX 来配置生成，这里我们讲解初始化代码，是为了让大家对 STM32 时钟系统有更加清晰的理解。

前面我们讲解过，在系统启动之后，程序会先执行 HAL 库定义的 SystemInit 函数，进行系统一些初始化配置。那么我们先来看看 SystemInit 程序：

```
void SystemInit (void)
{
  /* 将 RCC 时钟配置重置为默认重置状态(用于调试)*/
   RCC->CR |= (uint32_t)0x00000001; //打开 HSION 位
  /* 设置 SW, HPRE, PPRE1, PPRE2, ADCPRE 和 MCO 位 */
#if !defined(STM32F105xC) && !defined(STM32F107xC)
  RCC->CFGR &= (uint32_t)0xF8FF0000;
#else
  RCC->CFGR &= (uint32_t)0xF0FF0000;
#endif /* STM32F105xC */
  RCC->CR &= (uint32_t)0xFEF6FFFF;     // 复位 HSEON, CSSON 和 PLLON 位
  RCC->CR &= (uint32_t)0xFFFBFFFF;     // 复位 HSEBYP 位
  RCC->CFGR &= (uint32_t)0xFF80FFFF; //复位 CFGR 寄存器
#if defined(STM32F105xC) || defined(STM32F107xC)
  RCC->CR &= (uint32_t)0xEBFFFFFF;   // 复位 PLL2ON 和 PLL3ON 位
  RCC->CIR = 0x00FF0000;                  // 禁用所有中断并清除挂起位
  RCC->CFGR2 = 0x00000000;               // 重置 CFGR2 注册
#elif defined(STM32F100xB) || defined(STM32F100xE)
   RCC->CIR = 0x009F0000;                // 禁用所有中断并清除挂起位
  RCC->CFGR2 = 0x00000000;              // 重置 CFGR2 注册
#else
  RCC->CIR = 0x009F0000;                // 禁用所有中断并清除挂起位
#endif /* STM32F105xC */
#if  defined(STM32F100xE)  ||  defined(STM32F101xE)  ||  defined(STM32F101xG)  ||
defined(STM32F103xE) || defined(STM32F103xG)
  #ifdef DATA_IN_ExtSRAM
    SystemInit_ExtMemCtl();
  #endif /* DATA_IN_ExtSRAM */
#endif

/* 配置中断向量表地址=基地址+偏移地址 ----------*
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; //内部 SRAM 中的向量表重定位
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; //在内部 FLASH 中的向量表重定位
#endif
```

}从上面代码可以看出，SystemInit 主要做了如下三个方面工作：

1） 复位 RCC 时钟配置为默认复位值（默认开始了 HIS）

2） 外部存储器配置

3） 中断向量表地址配置

HAL 库的 SystemInit 函数并没有像标准库的 SystemInit 函数一样进行时钟的初始化配置。HAL 库的 SystemInit 函数除了打开 HSI 之外，没有任何时钟相关配置，所以使用 HAL 库我们必须编写自己的时钟配置函数。首先我们打开工程模板看看我们在工程 SYSTEM 分组下面定义的 sys.c

文件中的时钟初始化函数 Stm32_Clock_Init 的内容：

```
//时钟系统配置函数
//PLL:选择的倍频数，RCC_PLL_MUL2~RCC_PLL_MUL16
//返回值:0,成功;1,失败
void Stm32_Clock_Init(u32 PLL)
{
    HAL_StatusTypeDef ret = HAL_OK;
    RCC_OscInitTypeDef RCC_OscInitStructure;
    RCC_ClkInitTypeDef RCC_ClkInitStructure;

    RCC_OscInitStructure.OscillatorType=RCC_OSCILLATORTYPE_HSE; //时钟源为 HSE
    RCC_OscInitStructure.HSEState=RCC_HSE_ON;                   //打开 HSE
    RCC_OscInitStructure.HSEPredivValue=RCC_HSE_PREDIV_DIV1;    //HSE 预分频
    RCC_OscInitStructure.PLL.PLLState=RCC_PLL_ON;               //打开 PLL
    RCC_OscInitStructure.PLL.PLLSource=RCC_PLLSOURCE_HSE;
                                                                //PLL 时钟源选择 HSE
    RCC_OscInitStructure.PLL.PLLMUL=PLL;                        //主 PLL 倍频因子
    ret=HAL_RCC_OscConfig(&RCC_OscInitStructure);//初始化
    if(ret!=HAL_OK) while(1);
    //选中 PLL 作为系统时钟源并且配置 HCLK,PCLK1 和 PCLK2
    RCC_ClkInitStructure.ClockType=(RCC_CLOCKTYPE_SYSCLK|
            RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_PCLK1|
            RCC_CLOCKTYPE_PCLK2);
    //设置系统时钟时钟源为 PLL
    RCC_ClkInitStructure.SYSCLKSource=RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStructure.AHBCLKDivider=RCC_SYSCLK_DIV1;     //AHB 分频系数为 1
    RCC_ClkInitStructure.APB1CLKDivider=RCC_HCLK_DIV2;      //APB1 分频系数为 2
    RCC_ClkInitStructure.APB2CLKDivider=RCC_HCLK_DIV1;      //APB2 分频系数为 1
    ret=HAL_RCC_ClockConfig(&RCC_ClkInitStructure,FLASH_LATENCY_2);
                        //同时设置 FLASH 延时周期为 2WS，也就是 3 个 CPU 周期。
    if(ret!=HAL_OK) while(1);
}
```

从函数注释可知，函数 Stm32_Clock_Init 的作用是进行时钟系统配置，除了配置 PLL 相关参数确定 SYSCLK 值之外，还配置了 AHB,APB1 和 APB2 的分频系数，也就是确定了 HCLK，PCLK1 和 PCLK2 的时钟值。

接下来我们看看结构体 RCC_OscInitTypeDef 的定义：

```
typedef struct
{
  uint32_t OscillatorType;    //需要选择配置的振荡器类型
  uint32_t HSEState;          //HSE 状态
  uint32_t HSEPredivValue; // Prediv1 值
  uint32_t LSEState;          //LSE 状态
  uint32_t HSIState;          //HIS 状态
```

```
    uint32_t HSICalibrationValue;  //HIS 校准值
    uint32_t LSIState;             //LSI 状态
    RCC_PLLInitTypeDef PLL;        //PLL 配置
}RCC_OscInitTypeDef;
```

对于这个结构体，前面几个参数主要是用来选择配置的振荡器类型。比如我们要开启 HSE，那么我们会设置 OscillatorType 的值为 RCC_OSCILLATORTYPE_HSE，然后设置 HSEState 的值为 RCC_HSE_ON 开启 HSE。对于其他时钟源 HSI,LSI 和 LSE，配置方法类似。这个结构体还有一个很重要的成员变量是 PLL，它是结构体 RCC_PLLInitTypeDef 类型。它的作用是配置 PLL 相关参数，我们来看看它的定义：

```
typedef struct
{
  uint32_t PLLState;    //PLL 状态
  uint32_t PLLSource;   //PLL 时钟源
  uint32_t PLLMUL;      //PLL VCO 输入时钟的乘法因子
}RCC_PLLInitTypeDef;
```

从 RCC_PLLInitTypeDef;结构体的定义很容易看出该结构体主要用来设置 PLL 时钟源以及相关分频倍频参数。

这个结构体的定义我们就不做过多讲解，接下来我们看看我们的时钟初始化函数 Stm32_Clock_Init 中的配置内容：

```
RCC_OscInitStructure.OscillatorType=RCC_OSCILLATORTYPE_HSE; //时钟源为 HSE
RCC_OscInitStructure.HSEState=RCC_HSE_ON;                   //打开 HSE
RCC_OscInitStructure.HSEPredivValue=RCC_HSE_PREDIV_DIV1;    //HSE 预分频
RCC_OscInitStructure.PLL.PLLState=RCC_PLL_ON;               //打开 PLL
RCC_OscInitStructure.PLL.PLLSource=RCC_PLLSOURCE_HSE;
                                                            //PLL 时钟源选择 HSE
RCC_OscInitStructure.PLL.PLLMUL=PLL;                        //主 PLL 倍频因子
ret=HAL_RCC_OscConfig(&RCC_OscInitStructure);//初始化
```

通过该段函数，我们开启了 HSE 时钟源，同时选择 PLL 时钟源为 HSE，然后把 Stm32_Clock_Init 的唯一的入口参数直接设置作为 PLL 的倍频因子。设置好 PLL 时钟源参数之后，也就是确定了 PLL 的时钟频率，接下来我们就需要设置系统时钟，以及 AHB，APB1 和 APB2 相关参数。

接下来我们来看看步骤 5 中提到的 HAL_RCC_ClockConfig()函数，声明如下：

```
HAL_StatusTypeDef  HAL_RCC_ClockConfig(RCC_ClkInitTypeDef  *RCC_ClkInitStruct,
                                       uint32_t FLatency);
```

该函数有两个入口参数，第一个入口参数 RCC_ClkInitStruct 是结构体 RCC_ClkInitTypeDef 指针类型，用来设置 SYSCLK 时钟源以及 AHB,APB1 和 APB2 的分频系数。第二个入口参数 FLatency 用来设置 FLASH 延迟，这个参数我们放在后面讲解。

RCC_ClkInitTypeDef 结构体类型定义非常简单，这里我们就不列出来，我们来看看 Stm32_Clock_Init 函数中的配置内容：

```
//选中 PLL 作为系统时钟源并且配置 HCLK,PCLK1 和 PCLK2
RCC_ClkInitStructure.ClockType=(RCC_CLOCKTYPE_SYSCLK|
                                RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_PCLK1|
                                RCC_CLOCKTYPE_PCLK2);
```

```
//设置系统时钟时钟源为 PLL
RCC_ClkInitStructure.SYSCLKSource=RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStructure.AHBCLKDivider=RCC_SYSCLK_DIV1;     //AHB 分频系数为 1
RCC_ClkInitStructure.APB1CLKDivider=RCC_HCLK_DIV2;      //APB1 分频系数为 2
RCC_ClkInitStructure.APB2CLKDivider=RCC_HCLK_DIV1;      //APB2 分频系数为 1
ret=HAL_RCC_ClockConfig(&RCC_ClkInitStructure,FLASH_LATENCY_2);
                        //同时设置 FLASH 延时周期为 2WS，也就是 3 个 CPU 周期。
```

第一个参数 ClockType 配置说明我们要配置的是 SYSCLK，HCLK,PCLK1 和 PCLK2 四个时钟。
第二个参数 SYSCLKSource 配置选择系统时钟源为 PLL。
第三个参数 AHBCLKDivider 配置 AHB 分频系数为 1。
第四个参数 APB1CLKDivider 配置 APB1 分频系数为 2。
第五个参数 APB2CLKDivider 配置 APB2 分频系数为 1。

根据我们在主函数中调用 Stm32_Clock_Init(RCC_PLL_MUL9)时候设置的入口参数值，我们可以计算出，PLL 时钟为 PLLCLK=HSE*9 =8MHz*9=72MHz，同时我们选择系统时钟源为 PLL，所以系统时钟 SYSCLK=72MHz。AHB 分频系数为 1，故其频率为 HCLK=SYSCLK/1=72MHz。APB1 分频系数为 2，故其频率为 PCLK1=HCLK/2=36MHz。APB2 分频系数为 1，故其频率为 PCLK2=HCLK/1=72/1=72MHz。最后我们总结一下通过调用函数 Stm32_Clock_Init(RCC_PLL_MUL9)之后的关键时钟频率值：

```
SYSCLK(系统时钟)                    =72MHz
PLL 主时钟                          =72MHz
AHB 总线时钟（HCLK=SYSCLK/1）       =72MHz
APB1 总线时钟（PCLK1=HCLK/2）       =36MHz
APB2 总线时钟（PCLK2=HCLK/1）       =72MHz
```

4.3.3 STM32F1 时钟使能和配置

上一节我们讲解了时钟系统配置步骤。在配置好时钟系统之后，如果我们要使用某些外设，例如 GPIO，ADC 等，我们还要使能这些外设时钟。这里大家必须注意，如果在使用外设之前没有使能外设时钟，这个外设是不可能正常运行的。STM32 的外设时钟使能是在 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《STM32 中文参考手册 V10》6.3 小节查看所有 RCC 相关寄存器的配置。接下来我们来讲解通过 STM32F1 的 HAL 库使能外设时钟的方法。

在 STM32F1 的 HAL 库中，外设时钟使能操作都是在 RCC 相关固件库文件头文件 stm32f1xx_hal_rcc.h 定义的。大家打开 stm32f1xx_hal_rcc.h 头文件可以看到文件中除了少数几个函数声明之外大部分都是宏定义标识符。外设时钟使能在 HAL 库中都是通过宏定义标识符来实现的。首先，我们来看看 GPIOA 的外设时钟使能宏定义标识符：

```
#define __HAL_RCC_GPIOA_CLK_ENABLE()    do { \
                                        __IO uint32_t tmpreg; \
                                        SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);\
                                        tmpreg = READ_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);\
                                        UNUSED(tmpreg); \
                                    } while(0U))
```

这几行代码非常简单，主要是定义了一个宏定义标识符__HAL_RCC_GPIOA_CLK_ENABLE()，它的核心操作是通过下面这行代码实现的：

```
SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);
```

这行代码的作用是，设置寄存器 RCC_APB2ENR 的相关位为 1，至于是哪个位，是由宏定义标识符 RCC_APB2ENR_IOPAEN 的值决定的，而它的值为：

```
#define  RCC_APB2ENR_IOPAEN          ((uint32_t)0x00000001)
```

所以，我们很容易理解上面代码的作用是设置寄存器 RCC->APB2ENR 寄存器的位 2 为 1。我们可以从 STM32F1 的中文参考手册中搜索 APB2ENR 寄存器定义，位 2 的作用是用来使用 GPIOA 时钟。APB2ENR 寄存器的位 2 描述如下：

```
位 2    IOPAEN:IO 端口 A 时钟使能
        由软件置 1 和清零
        0：禁止 IO 端口 A 时钟
        1：使能 IO 端口 A 时钟
```

那么我们只需要在我们的用户程序中调用宏定义标识符__HAL_RCC_GPIOA_CLK_ENABLE()就可以实现 GPIOA 时钟使能。使用方法为：

```
__HAL_RCC_GPIOA_CLK_ENABLE();//使能 GPIOA 时钟
```

对于其他外设，同样都是在 stm32f1xx_hal_rcc.h 头文件中定义，大家只需要找到相关宏定义标识符即可，这里我们列出几个常用使能外设时钟的宏定义标识符使用方法：

```
__HAL_RCC_DMA1_CLK_ENABLE();//使能 DMA1 时钟
__HAL_RCC_USART2_CLK_ENABLE();//使能串口 2 时钟
__HAL_RCC_TIM1_CLK_ENABLE();//使能 TIM1 时钟
```

我们使用外设的时候需要使能外设时钟，如果我们不需要使用某个外设，同样我们可以禁止某个外设时钟。禁止外设时钟使用方法和使能外设时钟非常类似，同样是头文件中定义的宏定义标识符。我们同样以 GPIOA 为例，宏定义标识符为：

```
#define  __HAL_RCC_GPIOA_CLK_DISABLE() \
                    (RCC->APB2ENR &= ~(RCC_APB2ENR_IOPAEN))
```

同样，宏定义标识符__HAL_RCC_GPIOA_CLK_DISABLE()的作用是设置 RCC->APB2ENR 寄存器的位 2 为 0，也就是禁止 GPIOA 时钟。具体使用方法我们这里就不做过多讲解，我们这里同样列出几个常用的禁止外设时钟的宏定义标识符使用方法：

```
__HAL_RCC_DMA1_CLK_DISABLE();//禁止 DMA1 时钟
__HAL_RCC_USART2_CLK_DISABLE();//禁止串口 2 时钟
__HAL_RCC_TIM1_CLK_DISABLE();//禁止 TIM1 时钟
```

关于 STM32F1 的外设时钟使能和禁止方法我们就给大家讲解到这里。

4.4 端口复用和重映射

STM32F1 有很多的内置外设，这些外设的外部引脚都是与 GPIO 复用的。也就是说，一个 GPIO 如果可以复用为内置外设的功能引脚，那么当这个 GPIO 作为内置外设使用的时候，就叫做复用。这部分知识在《STM32 中文参考手册 V10》的 P109，P116~P121 有详细的讲解哪些 GPIO 管脚是可以复用为哪些内置外设的。这里我们就不一一讲解。

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

USART1_TX	PA9
USART1_RX	PA10

图 4.4.1.1 串口 1 复用管脚

接下来我们以串口 1 为例来讲解配置 GPOPA.9,GPIOA.10 口为串口 1 复用功能的一般步骤。

① 首先,我们要使用 IO 复用功能，必须先打开对应的 IO 时钟和复用功能外设时钟，这里我们使用了 GPIOA 以及 USART1,所以我们需要使能 GPIOA 和 USART1 时钟。方法如下：

```
__HAL_RCC_GPIOA_CLK_ENABLE();           //使能 GPIOA 时钟
__HAL_RCC_USART1_CLK_ENABLE();          //使能 USART1 时钟
__HAL_RCC_AFIO_CLK_ENABLE();            //使能辅助功能 IO 时钟
```

② 然后，我们在 GIPOx_MODER 寄存器中将所需 IO（对于串口 1 是 PA9,PA10）配置为复用功能。

③ 最后,我们还需要对 IO 口的其他参数，例如上拉/下拉以及输出速度等进行配置。

上面三步，在我们 HAL 库中是通过 HAL_GPIO_Init 函数来实现的，参考代码如下：

```
GPIO_InitTypeDef GPIO_Initure;

GPIO_Initure.Pin=GPIO_PIN_9;                //PA9
GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //复用推挽输出
GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化 PA9
```

通过上面的配置，PA9 复用为串口 1 的发送引脚。这个时候，PA9 将不再作为普通的 IO 口使用。对于 PA10，配置方法一样，修改 Pin 成员变量值为 PIN_10 即可。

STM32F1 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

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

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

```
typedef struct
{
  __IOM uint32_t ISER[8U];
        uint32_t RESERVED0[24U];
  __IOM uint32_t ICER[8U];
        uint32_t RSERVED1[24U];
  __IOM uint32_t ISPR[8U];
        uint32_t RESERVED2[24U];
  __IOM uint32_t ICPR[8U];
        uint32_t RESERVED3[24U];
  __IOM uint32_t IABR[8U];
       uint32_t RESERVED4[56U];
```

```
    __IOM uint8_t    IP[240U];
          uint32_t RESERVED5[644U];
    __OM    uint32_t STIR;
}   NVIC_Type;;
```

STM32 的中断在这些寄存器的控制下有序的执行的。只有了解这些中断寄存器，才能方便的使用 STM32 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 CM3 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 STM32F103 的可屏蔽中断只有 60 个，所以对我们来说，有用的就是两个（ISER[0]和 ISER[1]），总共可以表示 64 个中断。而 STM32F103 只用了其中的前 60 位。ISER[0]的 bit0~bit31 分别对应中断 0~31。ISER[1]的 bit0~27 对应中断 32~59；这样总共 60 个中断就分别对应上了。你要使能某个中断，必须设置相应的 ISER 位为 1，使该中断被使能(这里仅仅是使能，还要配合中断分组、屏蔽、IO 口映射等设置才算是一个完整的中断设置)。具体每一位对应哪个中断，请参考 stm32f10x.h 里面的第 140 行处（针对编译器 MDK5 来说）。

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。具体为什么这么设计，请看《CM3 权威指南》第 125 页，NVIC 概览一章。

ISPR[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！STM32 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 STM32 只用到了其中的前 60 个。IP[59]~IP[0]分别对应中断 59~0。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子优先级。抢占优先级在前，子优先级在后。而这两个优先级各占几个位又要根据 SCB->AIRCR 中的中断分组设置来决定。

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

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

表 4.5.1 AIRCR 中断分组设置表

通过这个表，我们就可以清楚的看到组 0~4 对应的配置关系，例如组设置为 3，那么此时

所有的 60 个中断，每个中断的中断优先寄存器的高四位中的最高 3 位是抢占优先级，低 1 位是响应优先级。每个中断，你可以设置抢占优先级为 0~7，响应优先级为 1 或 0。抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高。

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

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

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

通过以上介绍，我们熟悉了 STM32F103 中断设置的大致过程。接下来我们介绍如何使用 HAL 库实现以上中断分组设置以及中断优先级管理，使中断配置简单化。NVIC 中断管理相关函数主要在 HAL 库关键文件 stm32f1xx_hal_cortex.c 中定义。

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

```
void HAL_NVIC_SetPriorityGrouping(uint32_t 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需要被调用一次，一旦分组确定就最好不要更改，否则容易造成程序分组混乱。这个函数我们可以找到其函数体内容如下：

```
void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
{
  /* Check the parameters */
  assert_param(IS_NVIC_PRIORITY_GROUP(PriorityGroup));
  /* Set the PRIGROUP[10:8] bits according to the PriorityGroup parameter value */
  NVIC_SetPriorityGrouping(PriorityGroup);
}
```

从函数体以及注释可以看出，这个函数是通过调用函数 NVIC_SetPriorityGrouping 来进行中断优先级分组设置。通过查找(参考 3.5.3 小节 MDK 中“Go to definition of”的使用方法)，我们可以知道函数 NVIC_SetPriorityGrouping 是在文件 core_cm3.h 头文件中定义的。接下来，我们来分析一下函数 NVIC_SetPriorityGrouping 函数定义。定义如下：

```
__STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
{
  uint32_t reg_value;
  uint32_t PriorityGroupTmp = (PriorityGroup & (uint32_t)0x07UL);

  reg_value=   SCB->AIRCR;   /* read old register configuration      */
  reg_value&=~((uint32_t)(SCB_AIRCR_VECTKEY_Msk |SCB_AIRCR_PRIGROUP_Msk));
  reg_value   =   (reg_value|((uint32_t)0x5FAUL << SCB_AIRCR_VECTKEY_Pos) |
                  (PriorityGroupTmp<< SCB_AIRCR_PRIGROUP_Pos) );
  SCB->AIRCR =   reg_value;
}
```

从函数内容可以看出，这个函数主要作用是通过设置 SCB->AIRCR 寄存器的值来设置中断优先

级分组，这在前面寄存器讲解的过程中已经讲到。

关于函数 HAL_NVIC_SetPriorityGrouping 的函数体内容解读我就给大家介绍到这里。接下来我们来看看这个函数的入口参数。大家继续回到函数 HAL_NVIC_SetPriorityGrouping 的定义可以看到，函数的最开头有这样一行函数：

```
assert_param(IS_NVIC_PRIORITY_GROUP(PriorityGroup));
```

其中函数 assert_param 是断言函数，它的作用主要是对入口参数的有效性进行判断。也就是说我们可以通过这个函数知道入口参数在哪些范围内是有效的。而其入口参数通过在 MDK 中双击选中 “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))
```

从这个内容可以看出，当 GROUP 的值为 NVIC_PriorityGroup_0~ NVIC_PriorityGroup_4 的时候，IS_NVIC_PRIORITY_GROUP 的值才为真。这也就是我们上面表 4.5.1 讲解的，分组范围为 0-4，对应的入口参数为宏定义值 NVIC_PriorityGroup_0~ NVIC_PriorityGroup_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：

```
HAL_NVIC_SetPriorityGrouping (NVIC_PriorityGroup_2);
```

这样就确定了中断优先级分组为 2，也就是 2 位抢占优先级，2 位响应优先级，抢占优先级和响应优先级的值的范围均为 0-3。

讲到这里，大家对怎么进行系统的中断优先级分组设置，以及具体的中断优先级设置函数 HAL_NVIC_SetPriorityGrouping 的内部函数实现都有了一个详细的理解。接下来我们来看看在 HAL 库里面，是怎样调用 HAL_NVIC_SetPriorityGrouping 函数进行分组设置的。

打开 stm32f1xx_hal.c 文件可以看到，文件内部定义了 HAL 库初始化函数 HAL_Init，这个函数非常重要，其作用主要是对中断优先级分组，FLASH 以及硬件层进行初始化，我们在 3.1 小节对其进行了比较详细的讲解。这里我们只需要知道，在系统主函数 main 开头部分，我们都会首先调用 HAL_Init 函数进行一些初始化操作。在 HAL_Init 内部，有如下一行代码：

```
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
```

这行代码的作用是把系统中断优先级分组设置为分组 4，这在我们前面已经详细讲解。也就是说，在主函数中调用 HAL_Init 函数之后，在 HAL_Init 函数内部会通过调用我们前面讲解的 HAL_NVIC_SetPriorityGrouping 函数来进行系统中断优先级分组设置。所以，我们要进行中断优先级分组设置，只需要修改 HAL_Init 函数内部的这行代码即可。中断优先级分组的内容我们就给大家讲解到这里。

设置好了系统中断分组，也就是确定了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？官方 HAL 库文件 stm32f1xx_hal_cortex.c 中定义了三个单个中断优先级设置函数。函数如下：

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn,
                          uint32_t PreemptPriority, uint32_t SubPriority);
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

第一个函数 HAL_NVIC_SetPriority 是用来设置单个优先级的抢占优先级和响应优先级的值。第二个函数 HAL_NVIC_EnableIRQ 是用来使能某个中断通道。

第三个函数 HAL_NVIC_DisableIRQ 是用来清除某个中断使能的，也就是中断失能。
这三个函数的使用都非常简单，对于具体的调用方法，大家可以参考我们后面第九章外部中断实验讲解。

这里大家还需要注意，中断优先级分组和中断优先级设置是两个不同的概念。中断优先级分组是用来设置整个系统对于中断分组设置为哪个分组，分组号为 0-4，设置函数为 HAL_NVIC_SetPriorityGrouping，确定了中断优先级分组号，也就确定了系统对于单个中断的抢占优先级和响应优先级设置各占几个位（对应表 4.5.1）。设置好中断优先级分组，确定了分组号之后，接下来我们就是要对单个优先级进行中断优先级设置。也就是这个中断的抢占优先级和响应优先级的值，设置方法就是我们上面讲解的三个函数。

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。设置函数为 HAL_NVIC_PriorityGroupConfig。对于 HAL 库，在文件 stm32f1xx_hal.c 内部定义函数 HAL_Init 中有调用 HAL_NVIC_PriorityGroupConfig 函数进行相关设置，所以我们只需要修改 HAL_Init 内部对中断优先级分组设置即可。

② 设置单个中断的中断优先级别和使能相应中断通道，使用到的函数函数主要为函数 HAL_NVIC_SetPriority 和函数 HAL_NVIC_EnableIRQ。

4.6 HAL 库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 HAL 库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，占用一个内存单元，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 stm32f1xx.h 文件中完成的。我们通过 GPIOA 的几个寄存器的地址来讲解吧。

首先我们可以查看《STM32 中文参考手册 V10》中的寄存器地址映射表(P129)。这里我们选用 GPIOA 为例来讲解。GPIO 寄存器地址映射如下表 4.6.1：

表52　GPIO寄存器地址映像和复位值

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

表 4.6.1 GPIO 寄存器地址映射表

从这个表我们可以看出，GPIOA 的 7 个寄存器都是 32 位的，所以每个寄存器占有 4 个地址，一共占用 28 个地址，地址偏移范围为（000h~01Bh）。这个地址偏移是相对 GPIOA 的基地址而言的。GPIOA 的基地址是怎么算出来的呢？因为 GPIO 都是挂载在 APB2 总线之上，所以它的基地址是由 APB2 总线的基地址+GPIOA 在 APB2 总线上的偏移地址决定的。同理依次类推，我们便可以算出 GPIOA 基地址了。下面我们打开 stm32f103.h 定位到 GPIO_TypeDef 定义处：

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

然后定位到：

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

可以看出，GPIOA 是将 GPIOA_BASE 强制转换为 GPIO_TypeDef 结构体指针，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。然后在 MDK 中双击“GPIOA_BASE”选中之后右键选中“Go to definition of ”，便可以查看 GPIOA_BASE 的宏定义：

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

依次类推，可以找到最顶层：

```
#define APB2PERIPH_BASE         (PERIPH_BASE + 0x10000)
#define PERIPH_BASE             ((uint32_t)0x40000000)
```

所以我们便可以算出 GPIOA 的基地址位：

GPIOA_BASE= 0x40000000+0x10000+0x0800=0x40010800

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

0x4001 2000 - 0x4001 23FF	GPIO端口G
0x4001 2000 - 0x4001 23FF	GPIO端口F
0x4001 1800 - 0x4001 1BFF	GPIO端口E
0x4001 1400 - 0x4001 17FF	GPIO端口D
0x4001 1000 - 0x4001 13FF	GPIO端口C
0X4001 0C00 - 0x4001 0FFF	GPIO端口B
0x4001 0800 - 0x4001 0BFF	GPIO端口A

图 4.6.2 GPIO 存储器地址映射表

同样的道理，我们可以推算出其他外设的基地址。

上面我们已经知道 GPIOA 的基地址，那么那些 GPIOA 的 7 个寄存器的地址又是怎么算出来的呢？？在上面我们讲过 GPIOA 的各个寄存器对于 GPIOA 基地址的偏移地址，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 基地址+寄存器相对 GPIOA 基地址的偏移值

这个偏移值在上面的寄存器地址映像表中可以查到。

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

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

表 4.6.3 GPIOA 各寄存器实际地址表

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

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

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

4.7 MDK 中使用 HAL 库快速组织代码技巧

这一节主要讲解在 MDK 中使用 HAL 库开发的一些小技巧，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

stm32f1xx_hal_gpio.h 头文件中，声明 GPIO 初始化函数为：

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

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，怎么快速组织代码呢？

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，所以我们就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 of…”，如下图 4.7.1：

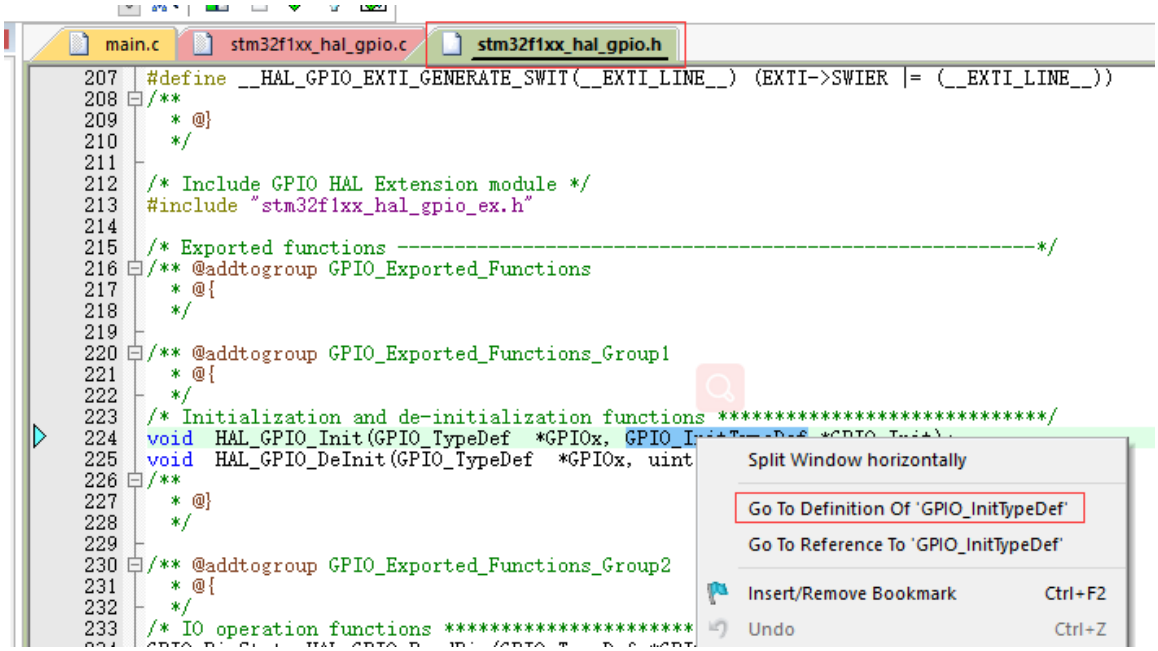


图 4.7.1 查看类型定义方法

于是定位到 stm32f1xx_hal_gpio.h 中 GPIO_InitTypeDef 的定义处：

```
typedef struct
{
  uint32_t Pin;
  uint32_t Mode;
  uint32_t Pull;
  uint32_t Speed;
}GPIO_InitTypeDef;
```

可以看到这个结构体有 4 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（Mode），速度(Speed)以及上下拉（Pull）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 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？

这里我们就回到 HAL_GPIO_Init 声明处，同样双击 HAL_GPIO_Init，右键点击“Go to definition of …”,这样光标定位到 stm32f1xx_hal_gpio.c 文件中的 HAL_GPIO_Init 函数体开始处，我们可以看到在函数中有如下几行：

```
void HAL_GPIO_Init(GPIO_TypeDef  *GPIOx, GPIO_InitTypeDef *GPIO_Init)
{
```

```
    …//此处省略部分代码
    assert_param(IS_GPIO_ALL_INSTANCE(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Init->Pin));
    assert_param(IS_GPIO_MODE(GPIO_Init->Mode));
    …//此处省略部分代码
    assert_param(IS_GPIO_PULL(GPIO_Init->Pull));
  …//此处省略部分代码

  }
```

顾名思义，assert_param 是断言语句，是对函数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，确定入口参数范围。第一行是对第一个参数 GPIOx 进行有效性判断，双击“IS_GPIO_ALL_INSTANCE”右键点击“go to defition of…” 定位到了下面的定义：

```
#define IS_GPIO_ALL_INSTANCE(INSTANCE) (((INSTANCE) == GPIOA) || \
                                        ((INSTANCE) == GPIOB) || \
                                        ((INSTANCE) == GPIOC) || \
                                        ((INSTANCE) == GPIOD) || \
                                        ((INSTANCE) == GPIOE) || \
                                        ((INSTANCE) == GPIOF) || \
                                        ((INSTANCE) == GPIOG))
```

很明显可以看出，GPIOx 的取值规定只允许是 GPIOA~GPIOG。

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

```
#define IS_GPIO_PIN(PIN)       (((((uint32_t)PIN) & GPIO_PIN_MASK ) != 0x00u)
                                && ((((uint32_t)PIN) & ~GPIO_PIN_MASK) == 0x00u))
```

同时，宏定义标识符 GPIO_PIN_MASK 的定义为：

```
#define GPIO_PIN_MASK                   0x0000FFFFu
```

从上面可以看出，PIN 取值只要低 16 位不为 0 即可。这里需要大家注意，因为一组 IO 口只有 16 个 IO，实际上 PIN 的值在这里只有低 16 位有效，所以 PIN 的取值范围为 0x0001~0xFFFF。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，我们不会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，HAL 库会将这些数字的含义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 GPIO_PIN_MASK 宏定义的上面还有数行宏定义：

```
#define GPIO_PIN_0                 ((uint16_t)0x0001)
#define GPIO_PIN_1                 ((uint16_t)0x0002)
#define GPIO_PIN_2                 ((uint16_t)0x0004)
…//此处省略部分定义
#define GPIO_PIN_14                ((uint16_t)0x4000)
#define GPIO_PIN_15                ((uint16_t)0x8000)
#define GPIO_PIN_All               ((uint16_t)0xFFFF)
```

这些宏定义 GPIO_PIN_0 ~ GPIO_PIN_All 就是 HAL 库事先定义好的，我们写代码的时候初始化结构体 成员变量 Pin 的时候入口参数可以是这些宏定义标识符。

同理，对于成员变量 Pull，我们用同样的方法，可以找到其取值范围定义为：

```
#define IS_GPIO_PULL(PULL) (((PULL) == GPIO_NOPULL)\
                    || ((PULL) == GPIO_PULLUP) || \ ((PULL) == GPIO_PULLDOWN))
```

也就是 PULL 的取值范围只能是标识符 GPIO_NOPULL，GPIO_PULLUP 以及 GPIO_PULLDOWN。

对于成员变量 Mode，方法都是一样的，这里基于篇幅考虑我们就不重复讲解。讲到这里，我们基本对 HAL_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_Initure;
GPIO_Initure.Pin=GPIO_PIN_9;                    //PA9
GPIO_Initure.Mode=GPIO_MODE_AF_PP;              //复用推挽输出
GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
HAL_GPIO_Init(GPIOA,&GPIO_Initure);             //初始化 PA9
```

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，Speed 和 Pull 参数值相同，因为这些参数并不能一次定义多种。所以初始化多个具有相同配置的 IO 口的方式可以是如下：

```
GPIO_InitTypeDef    GPIO_Initure;
GPIO_Initure.Pin=GPIO_PIN_9| GPIO_PIN_10| GPIO_PIN_11;     //PA9,PA10,PA11
GPIO_Initure.Mode=GPIO_MODE_AF_PP;     //复用推挽输出
GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化 PA9  ,PA10,PA11
```

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。

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 Pin;           /*!< Specifies the GPIO pins to be configured.
                               This parameter can be any value of @ref GPIO_pins_define */
  uint32_t Mode;          /*!< Specifies the operating mode for the selected pins.
                               This parameter can be a value of @ref GPIO_mode_define */
  uint32_t Pull;          /*!< Specifies the Pull-up or Pull-Down activation for the selected pins.
                               This parameter can be a value of @ref GPIO_pull_define */
  uint32_t Speed;         /*!< Specifies the speed for the selected pins.
                               This parameter can be a value of @ref GPIO_speed_define */
}GPIO_InitTypeDef;
```

从上图的结构体成员后面的注释我们可以看出 Pin 的意思是

```
“Specifies the GPIO pins to be configured.
                        This parameter can be any value of @ref GPIO_pins_define”。
```

从这段注释可以看出 Pin 的取值需要参考注释 GPIO_pins_define，大家可以在 MDK 中搜索注释

GPIO_pins_define，就可以找到上面我们提到的 Pin 的取值范围宏定义。如果要确定详细的信息我们就得去查看手册了。对于去查看手册的哪个地方，你可以在函数 HAL_GPIO_Init ()的函数体中搜索 Pin 关键字，然后查看库函数设置 Pin 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

第三章，我们介绍了如何在上一章，我们介绍了如何在 MDK5.23 下建立 STM32F1 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 STM32F10x 系列的底层核心驱动函数，可以用在 STM32F10x 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 STM32F1 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 7 个函数：

```
void delay_osschedlock(void);
void delay_osschedunlock(void);
void delay_ostimedly(u32 ticks);
void SysTick_Handler(void);
void delay_init(u8 SYSCLK);
void delay_ms(u16 nms);
void delay_us(u32 nus);
```

前面 4 个函数，仅在支持操作系统（OS）的时候，需要用到，而后面三个函数，则不论是否支持 OS 都需要用到。

在介绍这些函数之前，我们先了解一下 delay 延时的编程思想：CM3 内核的处理器，内部包含了一个 SysTick 定时器，SysTick 是一个 24 位的倒计数定时器，当计数到 0 时，将从 RELOAD 寄存器中自动重装载定时初值，开始新一轮计数。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick 在《STM32 中文参考手册》（这里是指 V10.0 版本，下同）里面介绍的很简单，其详细介绍，请参阅《Cortex-M3 权威指南》第 133 页。我们就是利用 STM32 的内部 SysTick 来实现延时的，这样既不占用中断，也不占用系统定时器。

这里我们将介绍的是 ALIENTEK 提供的最新版本的延时函数，该版本的延时函数支持在任意操作系统（OS）下面使用，它可以和操作系统共用 SysTick 定时器。

这里，我们以 UCOSII 为例，介绍如何实现操作系统和我们的 delay 函数共用 SysTick 定时器。首先，我们简单介绍下 UCOSII 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 宏定义设置），比如要求 5ms 一次（即可设置：OS_TICKS_PER_SEC=200），在 STM32 上面，一般是由 SysTick 来提供这个节拍，也就是 SysTick 要设置为 5ms 中断一次，为 ucos 提供时钟节拍，而且这个时钟一般是不能被打断的（否则就不准了）。

因为在 ucos 下 SysTick 不能再被随意更改，如果我们还想利用 SysTick 来做 delay_us 或者 delay_ms 的延时，就必须想点办法了，这里我们利用的是时钟摘取法。以 delay_us 为例，比如 delay_us(50)，在刚进入 delay_us 的时候先计算好这段延时需要等待的 SysTick 计数次数，这里为 50*9（假设系统时钟为 72Mhz，那么 SysTick 每增加 1，就是 1/72us），然后我们就一直统计 SysTick 的计数变化，直到这个值变化了 50*9，一旦检测到变化达到或者超过这个值，就说明延时 50us 时间到了。这样，我们只是抓取 SysTick 计数器的变化，并不需要修改 SysTick 的任何状态，完全不影响 SysTick 作为 UCOS 时钟节拍的功能，这就是实现 delay 和操作系统共用 SysTick 定时器的原理。

下面我们开始介绍这几个函数。

5.1.1 操作系统支持宏定义及相关函数

当需要 delay_ms 和 delay_us 支持操作系统（OS）的时候，我们需要用到 3 个宏定义和 4 个函数，宏定义及函数代码如下：

```
//本例程仅作 UCOSII 和 UCOSIII 的支持,其他 OS,请自行参考着移植
//支持 UCOSII
#ifdef    OS_CRITICAL_METHOD
//OS_CRITICAL_METHOD 定义了,说明要支持 UCOSII
#define delay_osrunning      OSRunning          //OS 是否运行标记,0,不运行;1,在运行
#define delay_ostickspersec  OS_TICKS_PER_SEC   //OS 时钟节拍,即每秒调度次数
#define delay_osintnesting   OSIntNesting       //中断嵌套级别,即中断嵌套次数
#endif

//支持 UCOSIII
#ifdef    CPU_CFG_CRITICAL_METHOD
//CPU_CFG_CRITICAL_METHOD 定义了,说明要支持 UCOSIII
#define delay_osrunning      OSRunning          //OS 是否运行标记,0,不运行;1,在运行
#define delay_ostickspersec  OSCfg_TickRate_Hz  //OS 时钟节拍,即每秒调度次数
#define delay_osintnesting   OSIntNestingCtr    //中断嵌套级别,即中断嵌套次数
#endif

//us 级延时时,关闭任务调度(防止打断 us 级延迟)
void delay_osschedlock(void)
{
#ifdef CPU_CFG_CRITICAL_METHOD   //使用 UCOSIII
    OS_ERR err;
    OSSchedLock(&err);           //UCOSIII 的方式,禁止调度，防止打断 us 延时
#else                            //否则 UCOSII
    OSSchedLock();               //UCOSII 的方式,禁止调度，防止打断 us 延时
#endif
}
//us 级延时时,恢复任务调度
void delay_osschedunlock(void)
{
```

```
#ifdef CPU_CFG_CRITICAL_METHOD   //使用 UCOSIII
    OS_ERR err;
    OSSchedUnlock(&err);         //UCOSIII 的方式,恢复调度
#else                            //否则 UCOSII
    OSSchedUnlock();             //UCOSII 的方式,恢复调度
#endif
}
//调用 OS 自带的延时函数延时
//ticks:延时的节拍数
void delay_ostimedly(u32 ticks)
{
#ifdef CPU_CFG_CRITICAL_METHOD   //使用 UCOSIII 时
    OS_ERR err;
    OSTimeDly(ticks,OS_OPT_TIME_PERIODIC,&err);//UCOSIII 延时采用周期模式
#else
    OSTimeDly(ticks);            //UCOSII 延时
#endif
}
//systick 中断服务函数,使用 ucos 时用到
void SysTick_Handler(void)
{
    if(delay_osrunning==1)                  //OS 开始跑了,才执行正常的调度处理
    {
        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，仅支持 UCOSII 和 UCOSIII，不过，对于其他 OS 的支持，也只需要对以上代码进行简单修改即可实现。

支持 OS 需要用到的三个宏定义（以 UCOSII 为例）即：

```
#define delay_osrunning      OSRunning          //OS 是否运行标记,0,不运行;1,在运行
#define delay_ostickspersec  OS_TICKS_PER_SEC   //OS 时钟节拍,即每秒调度次数
#define delay_osintnesting   OSIntNesting       //中断嵌套级别,即中断嵌套次数
```

宏定义：delay_osrunning，用于标记 OS 是否正在运行，当 OS 已经开始运行时，该宏定义值为 1，当 OS 还未运行时，该宏定义值为 0。

宏定义：delay_ ostickspersec，用于表示 OS 的时钟节拍，即 OS 每秒钟任务调度次数。

宏定义：delay_ osintnesting，用于表示 OS 中断嵌套级别，即中断嵌套次数，每进入一个中断，该值加 1，每退出一个中断，该值减 1。

支持 OS 需要用到的 4 个函数，即：

函数：delay_osschedlock，用于 delay_us 延时，作用是禁止 OS 进行调度，以防打断 us 级延时，导致延时时间不准。

函数：delay_osschedunlock，同样用于 delay_us 延时，作用是在延时结束后恢复 OS 的调度，

继续正常的 OS 任务调度。

函数：delay_ostimedly，则是调用 OS 自带的延时函数，实现延时。该函数的参数为时钟节拍数。

函数：SysTick_Handler，则是 systick 的中断服务函数，该函数为 OS 提供时钟节拍，同时可以引起任务调度。

以上就是 delay_ms 和 delay_us 支持操作系统时，需要实现的 3 个宏定义和 4 个函数。

5.1.2 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，如果需要支持操作系统（OS），只需要在 sys.h 里面，设置 SYSTEM_SUPPORT_OS 宏的值为 1 即可，然后，该函数会根据 delay_ostickspersec 宏的设置，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
//当使用 OS 的时候,此函数会初始化 OS 的时钟节拍
//SYSTICK 的时钟固定为 HCLK 时钟
//SYSCLK:系统时钟频率
void delay_init(u8 SYSCLK)
{
#if SYSTEM_SUPPORT_OS                    //如果需要支持 OS.
    u32 reload;
#endif
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
                                         //SysTick 频率为 HCLK
    fac_us=SYSCLK;                       //不论是否使用 OS,fac_us 都需要使用
#if SYSTEM_SUPPORT_OS                    //如果需要支持 OS.
    reload=SYSCLK;                       //每秒钟的计数次数  单位为 K
    reload*=1000000/delay_ostickspersec;     //根据 delay_ostickspersec 设定溢出时间
            //reload 为 24 位寄存器,最大值:16777216,在 72M 下,约合 0.233s 左右
    fac_ms=1000/delay_ostickspersec;         //代表 OS 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
#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 OS 的时候，只是设置一下 SysTick 的时钟源以及确定 fac_us 和 fac_ms 的值。而如果使用 OS 的时候，则会进行一些不同的配置，这里的条件编译是根据 SYSTEM_SUPPORT_OS 这个宏来确定的，该宏在 sys.h 里面定义。

SysTick 是 MDK 定义了的一个结构体（在 core_m3.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 5.1.2.1 所示：

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

图 5.1.2.1 SysTick->CTRL 寄存器各位定义

SysTick-> LOAD 的定义如图 5.1.2.2 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 5.1.2.2 SysTick->LOAD 寄存器各位定义

SysTick-> VAL 的定义如图 5.1.2.3 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 5.1.2.3 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。

HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);这一句把 SysTick 的时钟选择外部时钟，这里需要注意的是：SysTick 的时钟源自 HCLK，假设我们外部晶振为 8M，然后倍频到 72M，那么 SysTick 的时钟即为 72Mhz，也就是 SysTick 的计数器 VAL 每减 1，就代表时间过了 1/72us。所以 fac_us=SYSCLK;这句话就是计算在 SYSCLK 时钟频率下延时 1us 需要多少个 SysTick 时钟周期。

在不使用 OS 的时候：fac_us，为 us 延时的基数，也就是延时 1us，Systick 定时器需要走过的时钟周期数。 当使用 OS 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（前面已经介绍了）。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 delay_ostickspersec=200，那么 fac_ms 就是 5ms）。

5.1.3 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 OS 和不使用 OS 两个版本，首先是不使用 OS 的时候，实现函数如下：

```
//延时 nus
//nus 为要延时的 us 数.
//nus:0~190887435(最大值即 2^32/fac_us@fac_us=22.5)
void delay_us(u32 nus)
{
    u32 ticks;
```

```
    u32 told,tnow,tcnt=0;
    u32 reload=SysTick->LOAD;                //LOAD 的值
    ticks=nus*fac_us;                        //需要的节拍数
    told=SysTick->VAL;                       //刚进入时的计数器值
    while(1)
    {
        tnow=SysTick->VAL;
        if(tnow!=told)
        {
            if(tnow<told)tcnt+=told-tnow;//这里注意 SYSTICK 是递减的计数器就可以.
            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.
        }
    };
}
```

再来看看使用 OS 的时候，delay_us 的实现函数如下：

```
//延时 nus
//nus:要延时的 us 数.
//nus:0~190887435(最大值即 2^32/fac_us@fac_us=22.5)
void delay_us(u32 nus)
{
    u32 ticks;
    u32 told,tnow,tcnt=0;
    u32 reload=SysTick->LOAD;                //LOAD 的值
    ticks=nus*fac_us;                        //需要的节拍数
    delay_osschedlock();                     //阻止 OS 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;    //注意 SYSTICK 是一个递减的计数器.
            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.
        }
    };
    delay_osschedunlock();                   //恢复 OS 调度
}
```

这里就正是利用了我们前面提到的时钟摘取法，ticks 是延时 nus 需要等待的 SysTick 计数次数（也就是延时时间），told 用于记录最近一次的 SysTick->VAL 值，然后 tnow 则是当前的

SysTick->VAL 值，通过他们的对比累加，实现 SysTick 计数次数的统计，统计值存放在 tcnt 里面，然后通过对比 tcnt 和 ticks，来判断延时是否到达，从而达到不修改 SysTick 实现 nus 的延时。

对于使用 OS 的时候，delay_us 的实现函数和不使用 OS 的时候方法类似，都是使用的时钟摘取法，只不过使用 delay_osschedlock 和 delay_osschedunlock 两个函数，用于调度上锁和解锁，这是为了防止 OS 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。

5.1.4 delay_ms 函数

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

```
//延时 nms
//nms:要延时的 ms 数
void delay_ms(u16 nms)
{
    u32 i;
    for(i=0;i<nms;i++) delay_us(1000);
}
```

该函数其实就是多次调用前面所讲的 delay_us 函数，来实现毫秒级延时的。

再来看看使用 OS 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(delay_osrunning&&delay_osintnesting==0)//如果 OS 已经在跑了,且不是在中断里面
    {
        if(nms>=fac_ms)                         //延时的时间大于 OS 的最少时间周期
        {
            delay_ostimedly(nms/fac_ms);    //OS 延时
        }
        nms%=fac_ms;            //OS 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，delay_osrunning 是 OS 正在运行的标志，delay_osintnesting 则是 OS 中断嵌套次数，必须 delay_osrunning 为真，且 delay_osintnesting 为 0 的时候，才可以调用 OS 自带的延时函数进行延时（可以进行任务调度），delay_ostimedly 函数就是利用 OS 自带的延时函数，实现任务级延时的，其参数代表延时的时钟节拍数（假设 delay_osticksperse c=200，那么 delay_ostimedly (1)，就代表延时 5ms）。

当 OS 还未运行的时候，我们的 delay_ms 就是直接由 delay_us 实现的，OS 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 OS 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 OS 时钟节拍（fac_ms）,当大于这个值的时候，我们就通过调用 OS 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）。

5.1.5 HAL 库延时函数 HAL_Delay 解析

前面我们讲解了 ALIENTEK 提供的使用 Systick 实现延时相关函数。实际上，HAL 库有提供延时函数，只不过它只能实现简单的毫秒级别延时，没有实现 us 级别延时。下面我们列出 HAL 库实现延时相关的函数。首先是功能配置函数：

```
//调用 HAL_SYSTICK_Config 函数配置每隔 1ms 中断一次：文件 stm32f1xx_hal.c 中定义
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
  /* 配置系统在 1ms 的时间基础上有中断*/
  if (HAL_SYSTICK_Config(SystemCoreClock / (1000U / uwTickFreq)) > 0U)
  {
    return HAL_ERROR;
  }
  /* 配置 SysTick IRQ 优先级*/
  if (TickPriority < (1UL << __NVIC_PRIO_BITS))
  {
    HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority, 0U);
    uwTickPrio = TickPriority;
  }
  else
  {
    return HAL_ERROR;
  }
  return HAL_OK;
}

//HAL 库的 SYSTICK 配置函数：文件 stm32f1xx_hal_context.c 中定义
uint32_t HAL_SYSTICK_Config(uint32_t TicksNumb)
{
   return SysTick_Config(TicksNumb);
}

//内核的 Systick 配置函数，配置每隔 ticks 个 systick 周期中断一次
//文件 core_cm3.h 中
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
  ...//此处省略函数定义
}
```

上面三个函数，实际上开放给 HAL 调用的主要是 HAL_InitTick 函数，该函数在 HAL 库初

始化函数 HAL_Init 中会被调用。该函数通过间接调用 SysTick_Config 函数配置 Systick 定时器每隔 1ms 中断一次，永不停歇。

接下来我们来看看延时的逻辑控制代码：

```
//Systick 中断服务函数：文件 stm32f4xx_it.c 中
void SysTick_Handler(void)
{
  HAL_IncTick();
}

//下面代码均在文件 stm32f1xx_hal.c 中
static __IO uint32_t uwTick; //定义计数全局变量

__weak void HAL_IncTick(void)
{
  uwTick += uwTickFreq;
}

__weak uint32_t HAL_GetTick(void) //获取全局变量 uwTick 的值
{
  return uwTick;
}

//开放的 HAL 延时函数，延时 Delay 毫秒
__weak void HAL_Delay(__IO uint32_t Delay)
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(uwTickFreq);
  }
  while ((HAL_GetTick() - tickstart) < wait)
  {
  }
}
```

HAL 库实现延时功能非常简单，首先定义了一个 32 位全局变量 uwTick，在 Systick 中断服务函数 SysTick_Handler 中通过调用 HAL_IncTick 实现 uwTick 值不断增加，也就是每隔 1ms 增加 1。而 HAL_Delay 函数在进入函数之后先记录当前 uwTick 的值，然后不断在循环中读取 uwTick 当前值，进行减运算，得出的就是延时的毫秒数，整个逻辑非常简单也非常清晰。

但是，HAL 库的延时函数有一个局限性，在中断服务函数中使用 HAL_Delay 会引起混乱，因为它是通过中断方式实现，而 Systick 的中断优先级是最低的，所以在中断中运行 HAL_Delay 会导致延时出现严重误差。所以一般情况下，推荐大家使用 ALIENTEK 提供的延时函数库。

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 STM32F1 的 IO 口位操作输入读取宏定义和输出宏定义以及类型别名。sys.c 里面除了定义时钟系统配置函数 Stm32_Clock_Init 外主要是一些汇编函数，对于函数 Stm32_Clock_Init 的讲解请参考本手册 4.3 小节 STM32F103 时钟系统章节内容。本小节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 STM32F1 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。

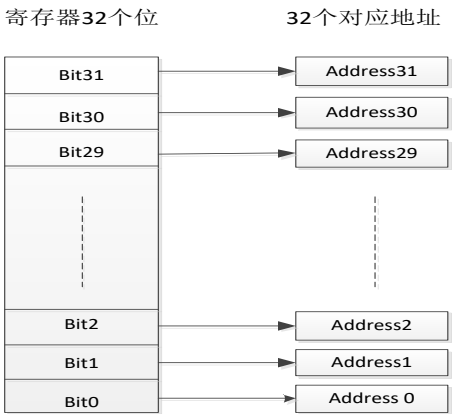


图 5.2.2.1 位带映射图

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).
//IO 口操作宏定义
#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+
                                ((addr &0xFFFFF)<<5)+(bitnum<<2))
#define MEM_ADDR(addr)  *((volatile unsigned long  *)(addr))
#define BIT_ADDR(addr, bitnum)   MEM_ADDR(BITBAND(addr, bitnum))
//IO 口地址映射
#define GPIOA_ODR_Addr    (GPIOA_BASE+12) //0x4001080C
#define GPIOB_ODR_Addr    (GPIOB_BASE+12) //0x40010C0C
```

```
……//省略部分代码
#define GPIOF_ODR_Addr      (GPIOF_BASE+12) //0x40011A0C
#define GPIOG_ODR_Addr      (GPIOG_BASE+12) //0x40011E0C

#define GPIOA_IDR_Addr      (GPIOA_BASE+8) //0x40010808
#define GPIOB_IDR_Addr      (GPIOB_BASE+8) //0x40010C08
……//省略部分代码
#define GPIOF_IDR_Addr      (GPIOF_BASE+8) //0x40011A08
#define GPIOG_IDR_Addr      (GPIOG_BASE+8) //0x40011E08

//IO 口操作,只对单一的 IO 口!
//确保 n 的值小于 16!
#define PAout(n)    BIT_ADDR(GPIOA_ODR_Addr,n)   //输出
#define PAin(n)     BIT_ADDR(GPIOA_IDR_Addr,n)   //输入
#define PBout(n)    BIT_ADDR(GPIOB_ODR_Addr,n)   //输出
#define PBin(n)     BIT_ADDR(GPIOB_IDR_Addr,n)   //输入
……//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_OS       0       //定义系统文件夹是否支持 UCOS
```

SYSTEM_SUPPORT_OS，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

该文件夹下面有 usart.c 和 usarts.h 两个文件。串口相关知识，**我们将在第九章讲解串口实验的时候给大家详细讲解。**本节我们只给大家讲解比较独立的 printf 函数支持相关的知识。

5.3.1 printf 函数支持

printf 函数支持的代码在 usart.c 文件的最上方，在我们初始化和使能串口 1 之后，然后把这段代码加入到工程，便可以通过 printf 函数向串口 1 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，大多情况我们都不需要修改。

代码内容如下：

```
//加入以下代码,支持 printf 函数,而不需要选择 use MicroLIB
#if 1
#pragma import(__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((USART1->SR&0X40)==0);//循环发送,直到发送完毕
    USART1->DR = (u8) ch;
    return ch;}
#endif
```

第三篇 实战篇

经过前两篇的学习，我们对 STM32F1 开发的软件和硬件平台都有了个比较深入的了解了，接下来我们将通过实例，由浅入深，带大家一步步的学习 STM32。

STM32F1 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 STM32F1 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 STM32F1 的各种外设，通过本篇的学习，希望大家能学会 STM32F1 绝大部分外设的使用。

本篇总共分为 48 章，每一章即一个实例，下面就让我们开始精彩的 STM32F1 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 STM32F1 之旅，通过本章的学习，你将了解到 STM32F1 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 精英 STM32F103 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， STM32F1 IO 口简介
6.2， 硬件设计
6.3， 软件设计
6.4， 下载验证

6.1 STM32F1 IO 简介

本章将要实现的是控制 ALIENTEK 精英 STM32 V3 开发板上的两个 LED 实现一个类似跑马灯的效果，该实验的关键在于如何控制 STM32 的 IO 口输出。了解了 STM32 的 IO 口如何输出的，就可以实现跑马灯了。通过这一章的学习，你将初步掌握 STM32 基本 IO 口的使用，而这是迈向 STM32 的第一步。

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

在讲解 STM32F103 的 GPIO 之前，首先打开我们光盘的第一个 HAL 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-HAL 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如下图 6.1.1 所示：

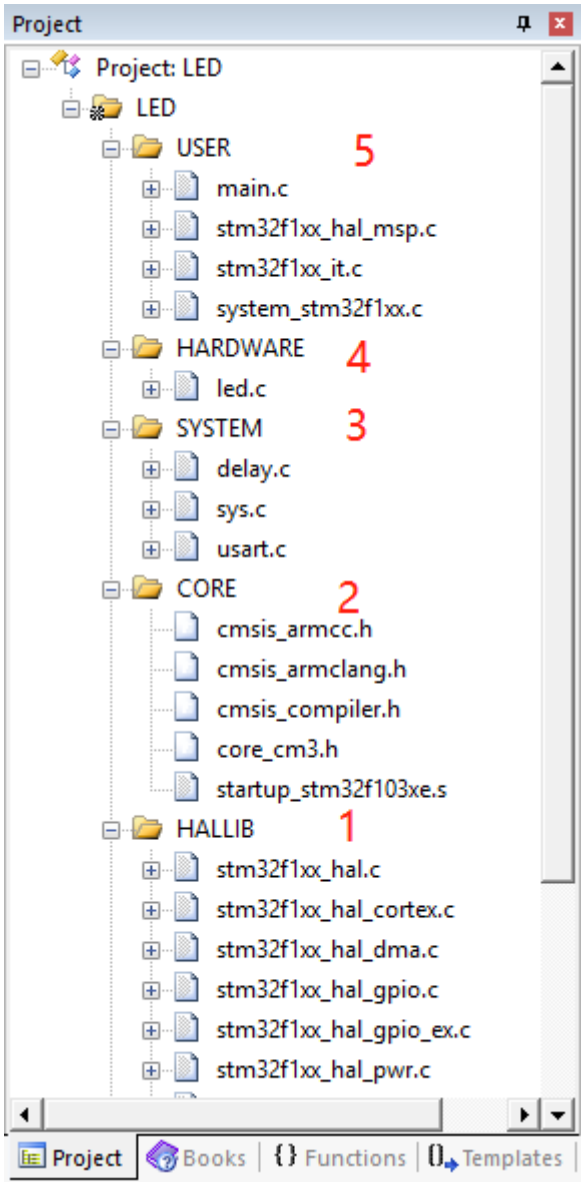


图 6.1.1 跑马灯实验目录结构

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 HALLIB 下面存放的是 ST 官方提供的 HAL 库文件，每一个源文件 stm32f1xx_hal_ppp.c 都对应一个头文件 stm32f1xx_hal_ppp.h。分组内的源文件我们可以根据工程需要添加和删除。

这里对于跑马灯实验，我们需要添加 10 个源文件。

② 组 CORE 下面存放的是固件库必须的核心头文件和启动文件。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，这些代码在第五章都有详细讲解。

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 HALLIB 下面的 HAL 库文件函数实现的，比如 led.c 中函数调用 stm32f1xx_hal_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 下面存放的主要是用户代码。但是 system_stm32f1xx.c 文件用户不需要修改，同时 stm32f1xx_it.c 里面存放的是中断服务函数，这两个文件的作用在 3.3 节有讲解。main.c 函数主要存放的是主函数了。

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 HAL 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 HAL 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基本的了解，这样对以后的学习也很有帮助。

STM32 的 IO 口可以由软件配置成如下 8 种模式：

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

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

STM32 的每个 IO 端口都有 7 个寄存器来控制。他们分别是：配置模式的 2 个 32 位的端口配置寄存器 CRL 和 CRH；2 个 32 位的数据寄存器 IDR 和 ODR；1 个 32 位的置位/复位寄存器 BSRR；一个 16 位的复位寄存器 BRR；1 个 32 位的锁存寄存器 LCKR；这里我们仅介绍常用 的几个寄存器，我们常用的 IO 端口寄存器只有 4 个：CRL、CRH、IDR、ODR。

CRL 和 CRH 控制着每个 IO 口的模式及输出速率。

STM32 的 IO 口位配置表如表 6.1.1 所示：

<table>
<tr><th colspan="2">配置模式</th><th>CNF1</th><th>CNF0</th><th colspan="2">MODE1 MODE0</th><th>PxODR寄存器</th></tr>
<tr><td rowspan="2">通用输出</td><td>推挽式(Push-Pull)</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">01
10
11
见表3.1.2</td><td>0 或 1</td></tr>
<tr><td>开漏(Open-Drain)</td><td>1</td><td>0 或 1</td></tr>
<tr><td rowspan="2">复用功能输出</td><td>推挽式(Push-Pull)</td><td rowspan="2">1</td><td>0</td><td>不使用</td></tr>
<tr><td>开漏(Open-Drain)</td><td>1</td><td>不使用</td></tr>
<tr><td rowspan="4">输入</td><td>模拟输入</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">00</td><td>不使用</td></tr>
<tr><td>浮空输入</td><td>1</td><td>不使用</td></tr>
<tr><td>下拉输入</td><td rowspan="2">1</td><td rowspan="2">0</td><td>0</td></tr>
<tr><td>上拉输入</td><td>1</td></tr>
</table>

表 6.1.1 STM32 的 IO 口位配置表

STM32 输出模式配置如表 6.1.2 所示：

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

表 6.1.2 STM32 输出模式配置表

接下来我们看看端口低配置寄存器 CRL 的描述，如图 6.1.1 所示：

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

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

图 6.1.1 端口低配置寄存器 CRL 各位描述

该寄存器的复位值为 0X4444 4444，从图 6.1.1 可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRL 控制着每组 IO 端口（A~G）的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式（ADC 用）、0X3 表示推挽输出模式（做输出口用，50M 速率）、0X8 表示上/下拉输入模式（做输入口用）、0XB 表示复用输出（使用 IO 口的第二功能，50M 速率）。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8 位输出口。这里我们对 CRH 就不做详细介绍了。

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 HAL 库文件 stm32f1xx_hal_gpio.c 和头文件 stm32f1xx_hal_gpio.h 文件中。

在 HAL 库开发中，初始化 GPIO 是通过 GPIO 初始化函数完成：

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

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~GPIOE。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 HALLIB 组下面的 stm32f1xx_hal_gpio.c 文件，定位到 HAL_GPIO_Init 函数体处，选中结构体“Ctrl + F”全局搜索，可以查看结构体的定义：

```
typedef struct
{
  uint32_t Pin;
  uint32_t Mode;
  uint32_t Pull;
  uint32_t Speed;
}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_Initure.Pin=GPIO_PIN_9|GPIO_PIN_10;          //PF9,10
GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
GPIO_Initure.Pull=GPIO_PULLUP;                    //上拉
GPIO_Initure.Speed= GPIO_SPEED_FREQ_HIGH;   //高速
HAL_GPIO_Init(GPIOF,&GPIO_Initure);
```

上面代码的意思是设置 GPIOF 的第 9 和 10 端口为推挽输出模式，同时速度为 50M，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 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 Mode 是用来设置对应 IO 端口的输出输入端口模式。在 MDK 中是通过宏定义来定义的，我们只需要选择对应的值即可：

```
#define  GPIO_MODE_INPUT                          0x00000000u
#define  GPIO_MODE_OUTPUT_PP                      0x00000001u
#define  GPIO_MODE_OUTPUT_OD                      0x00000011u
……省略部分宏定义
#define  GPIO_MODE_EVT_RISING_FALLING             0x10320000u
```

例如 GPIO_MODE_INPUT 是输入模式，GPIO_MODE_OUTPUT_PP 是推挽输出模式等等，根据实际需求来选择。

第三个参数 Pull 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。同样通过宏定义来定义的：

```
#define  GPIO_NOPULL        0x00000000u
#define  GPIO_PULLUP        0x00000001u
#define  GPIO_PULLDOWN      0x00000002u
```

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

第四个参数 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 中同样是宏定义来定义的：

```
#define  GPIO_SPEED_FREQ_LOW                (GPIO_CRL_MODE0_1)
#define  GPIO_SPEED_FREQ_MEDIUM             (GPIO_CRL_MODE0_0)
#define  GPIO_SPEED_FREQ_HIGH               (GPIO_CRL_MODE0)
```

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。

ODR 是一个端口输出数据寄存器，也只用了低 16 位。该寄存器为可读写，从该寄存器读出来的数据可以用于判断当前 IO 口的输出状态。而向该寄存器写数据，则可以控制某个 IO 口的输出电平。该寄存器的各位描述如图 6.1.3 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:16	保留，始终读为0。
位15:0	**ODRy[15:0]**：端口输出数据(y = 0…15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注：对GPIOx_BSRR(x = A…E)，可以分别地对各个ODR位进行独立的设置/清除。

图 6.1.3 端口输出数据寄存器 ODR 各位描述

在 HAL 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 HAL_GPIO_WritePin 来实现的：

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
GPIO_PinState PinState)
```

使用实例如下：

```
HAL_GPIO_WritePin(GPIOF,GPIO_PIN_9,GPIO_PIN_SET);
```

IDR 是一个端口输入数据寄存器，只用了低 16 位。该寄存器为只读寄存器，并且只能以 16 位的形式读出。该寄存器各位的描述如图 6.1.2 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

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

图 6.1.2 端口输入数据寄存器 IDR 各位描述

要想知道某个 IO 口的状态，你只要读这个寄存器，再看某个位的状态就可以了。使用起来是比较简单的。库函数相关函数为：

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

函数是用来读取一组 IO 口的一个输入电平。比如我们要读取 GPIOF.5 的输入电平，方法为：

```
HAL_GPIO_ReadPin (GPIOF, GPIO_Pin_5);
```

了解了这几个寄存器，我们就可以开始跑马灯实验的真正设计了。关于 IO 口更详细的介

绍，请参考《STM32 中文参考手册》第 105 页 8.1 节。

6.2 硬件设计

本章用到的硬件只有 LED（DS0 和 DS1）。其电路在 ALIENTEK 精英 STM32F103 上默认是已经连接好了的。DS0 接 PB5，DS1 接 PE5。所以在硬件上不需要动任何东西。其连接原理图如图 6.2.1 下：

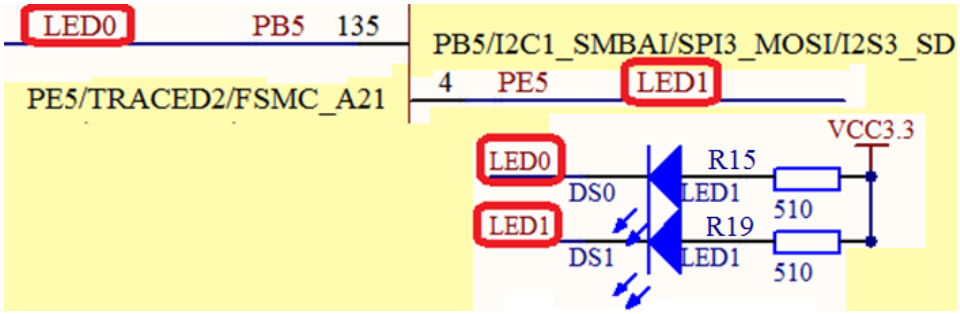


图 6.2.1 LED 与 STM32F1 连接原理图

6.3 软件设计

这是我们学习的第一个实验，所以我会手把手教大家怎么从我们前面讲解的 Template 工程模板一步一步加入 HAL 库以及 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 小节新建的 HAL 库工程模板。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-HAL 库函数版本\实验 0-1 Template 工程模板-新建工程章节使用**”（注意，是直接点击工程下面的 USER 目录下面的 **Tempate.uvprojx**。）。

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

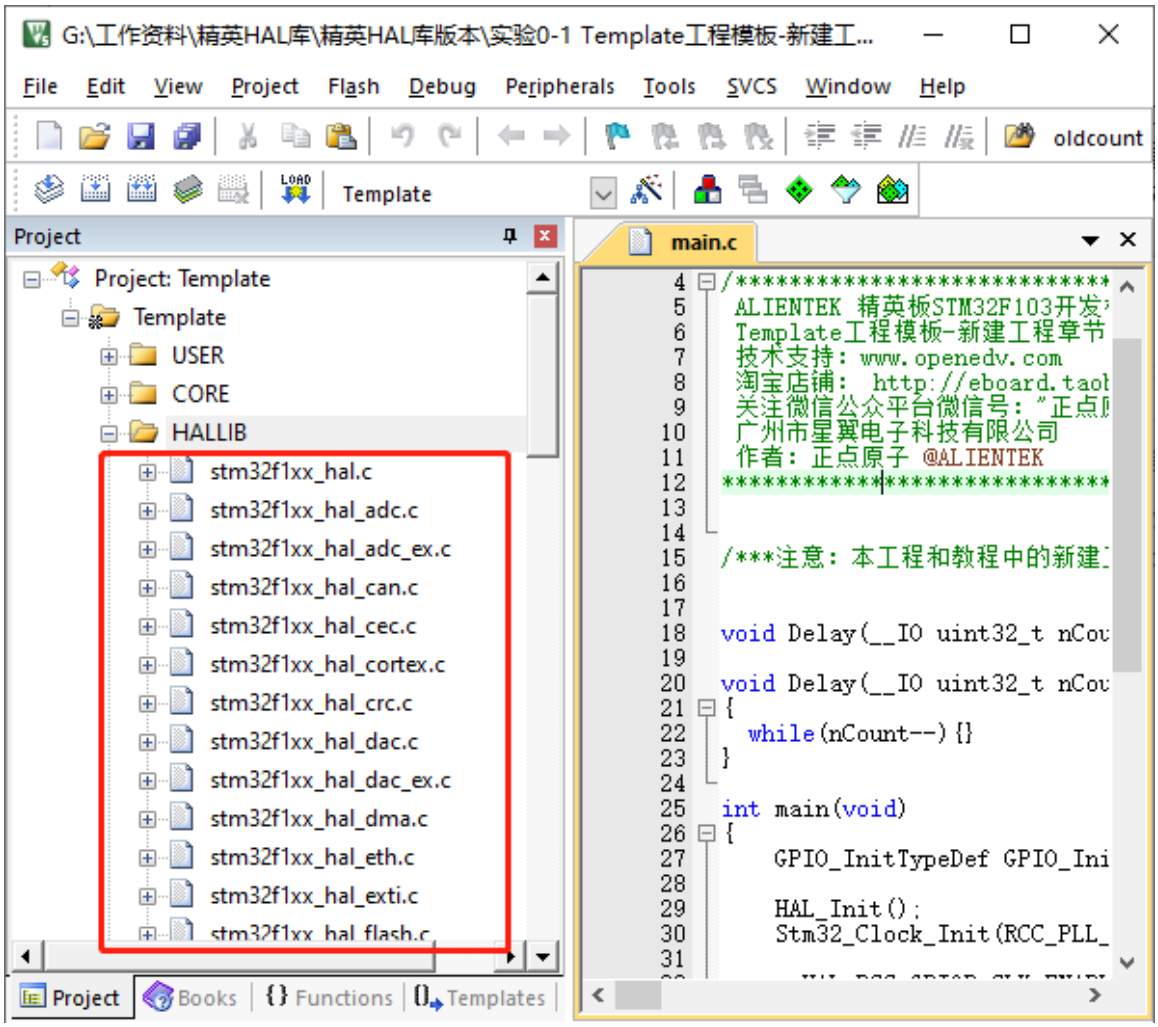


图 6.3.1　Template 模板工程结构

实际上，这些大家可以根据工程需要添加，比如跑马灯实验并没有用到 ADC，我们可以在工程中删掉文件 stm32f1xx_hal_adc.c，这样可以大大减少工程编译时间。跑马灯实验我们一共使用到 HAL 库中 10 个源文件，具体哪 10 个请直接参考我们跑马灯实验工程，其他不用的源文件大家可以直接在工程中删除。在工程的 Manage Project Items 页面，选择要删除文件所在的分组，然后选中文件点击删除按钮即可。具体操作方法如下图 6.3.2 所示：

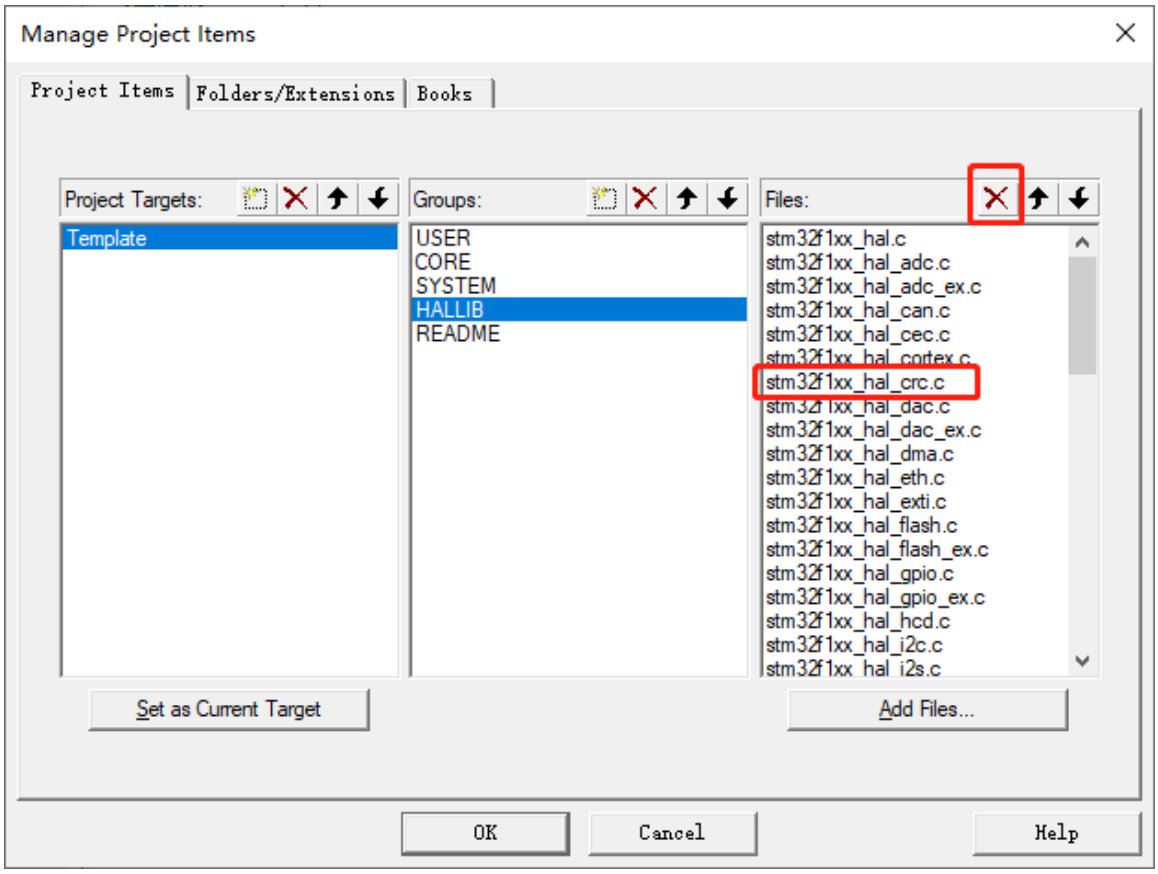


图 6.3.2 删除工程分组中的文件

接下来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 6.3.3 所示：

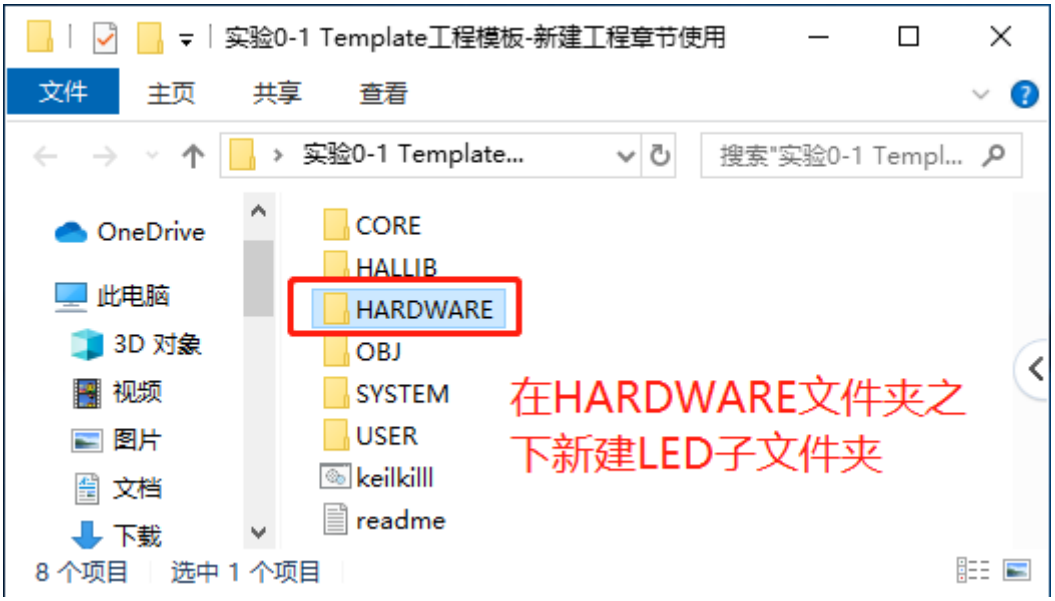


图 6.3.3 新建 HARDWARE 文件夹

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 6.3.4 和 6.3.5：

图 6.3.4 新建文件

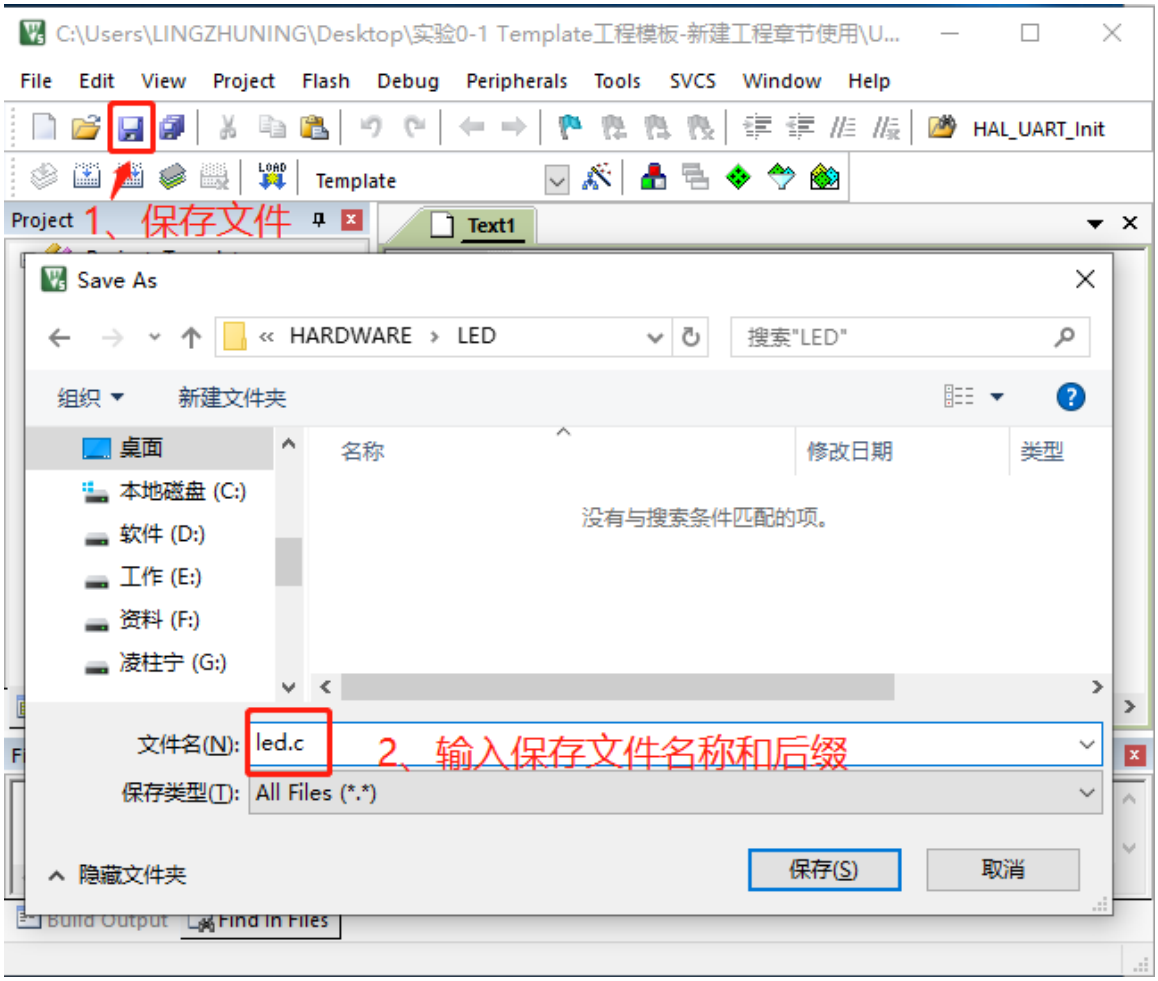


图 6.3.5 保存 led.c

然后在 led.c 文件中输入如下代码(**代码大家可以直接打开我们光盘的实验 1 跑马灯实验，从 led.c 文件内复制过来**)，输入后保存即可：

```
#include "led.h"

//初始化 PB1 为输出.并使能时钟
//LED IO 初始化
void LED_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();           //开启 GPIOB 时钟
    __HAL_RCC_GPIOE_CLK_ENABLE();           //开启 GPIOE 时钟

    GPIO_Initure.Pin=GPIO_PIN_5;            //PB5
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
```

```
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;     //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);

    GPIO_Initure.Pin=GPIO_PIN_5;                 //PE5
    HAL_GPIO_Init(GPIOE,&GPIO_Initure);

    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET);
                                                 //PB5 置 1，默认初始化后灯灭
    HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_SET);
                                                 //PE5 置 1，默认初始化后灯灭}
```

该代码里面就包含了一个函数 void LED_Init(void)，该函数通过调用函数 HAL_GPIO_Init 实现配置 PB5 和 PE5 为推挽输出。关于函数 HAL_GPIO_Init 的使用方法在 6.1 小节有详细讲解。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。使能 GPIOB 时钟方法为：

```
    __HAL_RCC_GPIOB_CLK_ENABLE();                //使能 GPIOB 时钟
```

在设置完时钟之后，LED_Init 调用 HAL_GPIO_Init 函数完成对 PB0 和 PB1 的初始化配置，然后调用函数 HAL_GPIO_WritePin 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 6.1 小节，我们有详细的讲解。

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"

#define LED0 PBout(5)       //LED0
#define LED1 PEout(5)       //LED1

void LED_Init(void);
#endif
```

这段代码里面最关键就是 2 个宏定义：

```
#define LED0 PBout(5)       //LED0
#define LED1 PEout(5)       //LED1
```

这里使用的是位带操作来实现操作某个 IO 口，关于位带操作前面第五章 5.2.1 已经有详细介绍，这里不再多说。需要说明的是，这里同样可以使用 HAL 库操作来实现 IO 口操作。如下：

```
HAL_GPIO_WritePin (GPIOB, GPIO_Pin_5, GPIO_PIN_SET);//PB5=1，等同 LED0=1;
HAL_GPIO_ReadPin(GPIOB, GPIO_Pin_5);                 //读取 PB5 的输入电平
```

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 Project Itmes 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 6.3.6 所示：

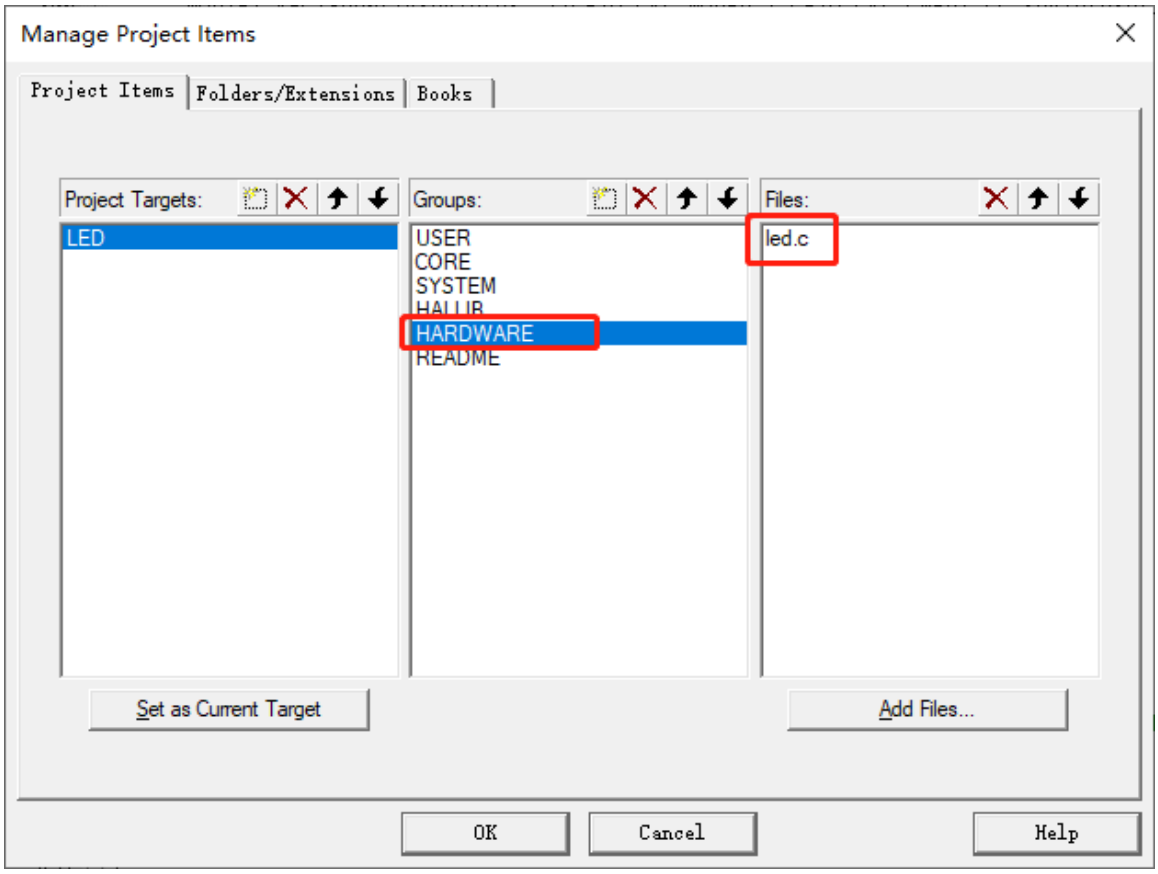


图 6.3.6 给工程新增 HARDWARE 组

单击 OK，回到工程，然后你会发现在 Project Workspace 里面多了一个 HARDWARE 组，在该组下面有一个 led.c 文件。如图 6.3.7 所示：

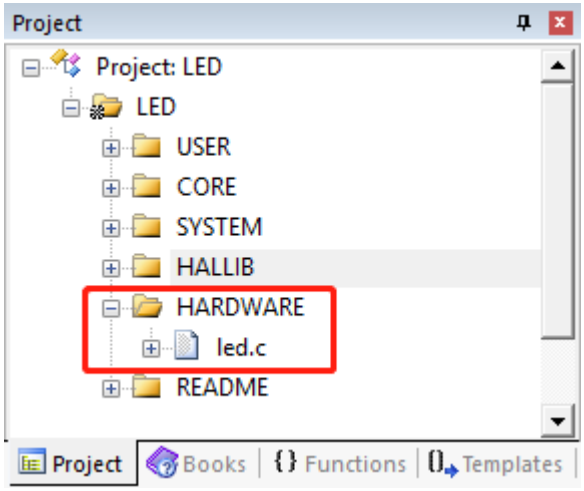


图 6.3.7 工程主界面

然后用之前介绍的方法（在 3.3 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，如下图 6.3.8 所示：

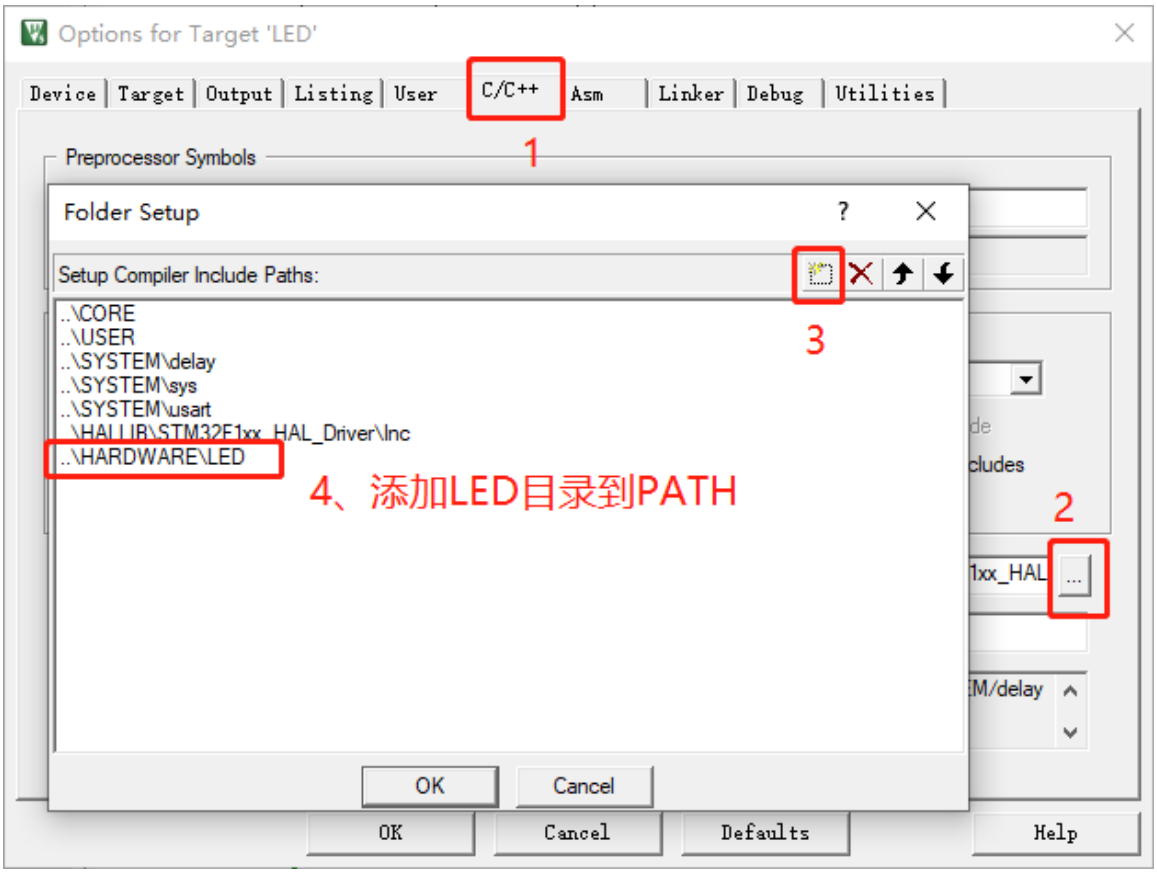


图 6.3.8 添加 LED 目录到 PATH

回到主界面后，修改 main.c 文件内容如下（具体内容请参考跑马灯实验 main.c 文件）：

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
#include "led.h"
int main(void)
{
    HAL_Init();                          //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                      //初始化延时函数
    LED_Init();                          //初始化 LED
    while(1)
    {
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_RESET); //LED0 亮
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_SET);   //LED1 灭
        delay_ms(500);                                      //延时 500ms
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET);   //LED0 灭
        HAL_GPIO_WritePin(GPIOE,GPIO_PIN_5,GPIO_PIN_RESET); //LED1 亮
        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。main()函数非常简单，先调用 HAL_Init 函数初始化 HAL 库，然后调用 Stm32_Clock_Init 进行时钟系统配置，然后调用 delay_init()函数进行延时初始化。接着就是调用 LED_Init()来初始化 PB5 和 PE5 为推挽输出模式，最后在 while 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)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    LED_Init();                             //初始化 LED
    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)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    LED_Init();                             //初始化 LED

    while(1)
    {
      GPIOB->BSRR=GPIO_PIN_5;           //LED0 亮
      GPIOE->BSRR=GPIO_PIN_5<<16;       //LED1 灭
      delay_ms(500);
      GPIOB->BSRR=GPIO_PIN_5<<16;       //LED0 灭
      GPIOE->BSRR=GPIO_PIN_5;           //LED1 亮
      delay_ms(500);                     //延时 500ms
     }
 }
```

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 6.3.9 所示：

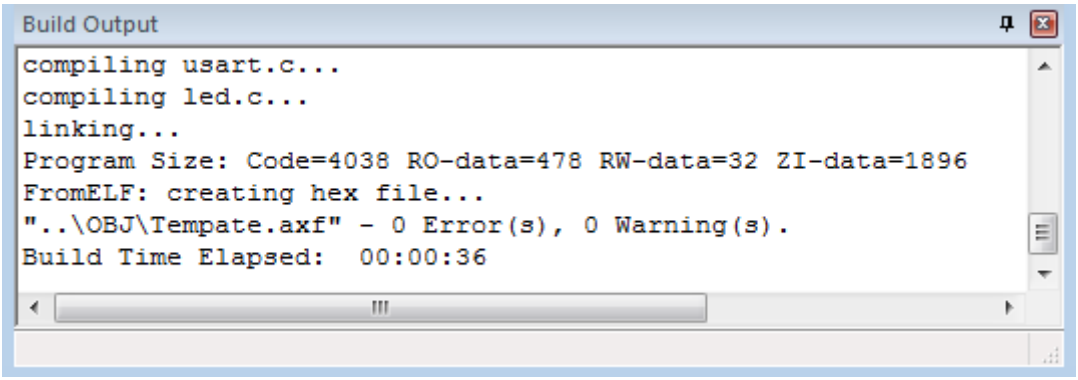


图 6.3.9 编译结果

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：4516 字节（4038+478），所用的 SRAM 大小为：1928 个字节（1896+32）。

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）

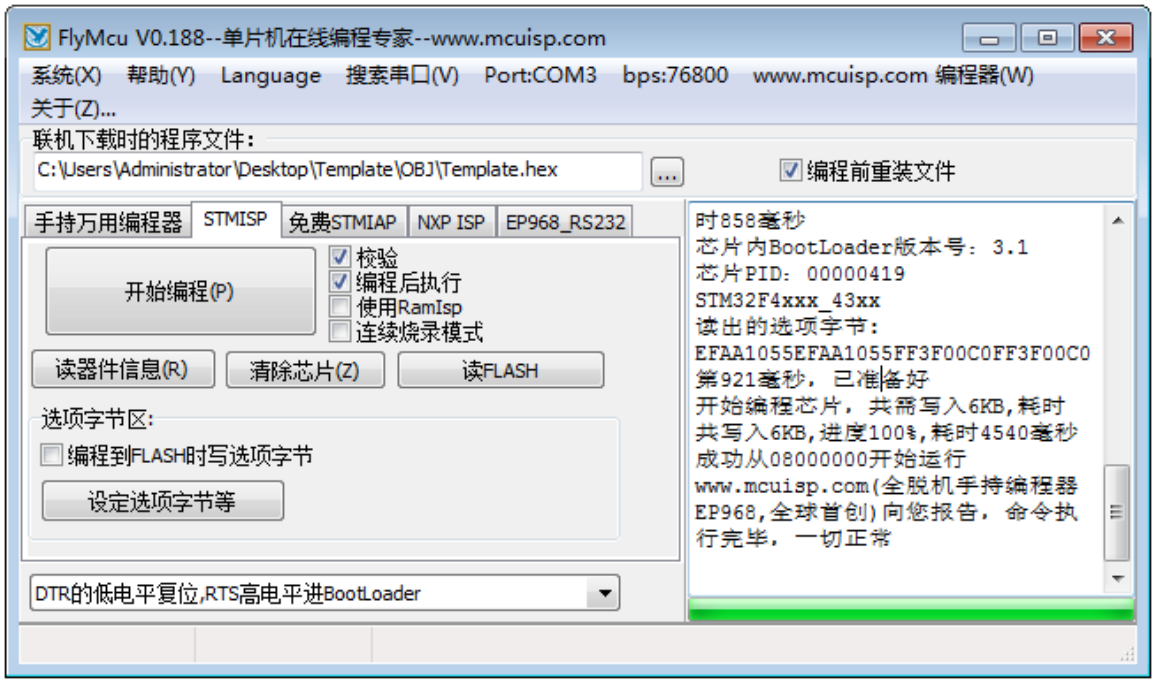
ZI-data：即 Zero Init-data，表示未被初始化的变量(SRAM)

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 ST-LINK，则可以用 ST-LINK 进行在线调试（需要先下载代码），单步查看代码的运行，STM32F1 的在线调试方法介绍请参见 3.4.2 小节。

6.4 下载验证

这里我们使用 flymcu 下载（也可以通过 ST-LINK 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，我们第一章关于使用 HAL 操作 GPIO 口的知识就给大家讲解到这里，本章作为 STM32F1 的入门第一个例子，介绍了 STM32F1 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 STM32F1 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 STM32F1 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 STM32F1 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。

精英 STM32F103 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 STM32F1 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：STM32F1 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，STM32F1 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 STM32F1 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 STM32F1 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 精英 STM32F103 上的蜂鸣器发出：“嘀”…“ 嘀”…的间隔声，进一步熟悉 STM32F1 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 STM32F1 的 PB8 上。如图 7.2.1 所示：

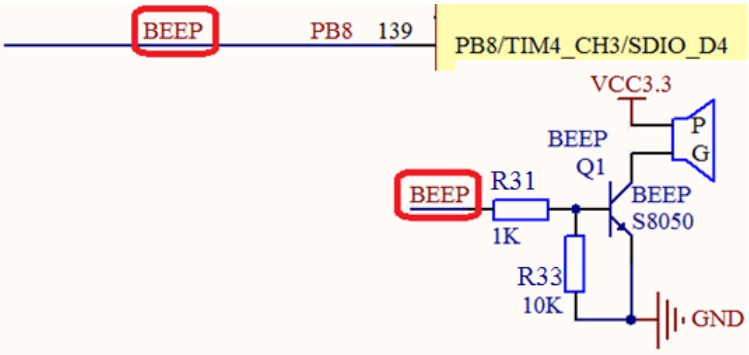


图 7.2.1 蜂鸣器与 STM32F1 连接原理图

图中我们用到一个 NPN 三极管（S8050）来驱动蜂鸣器，R33 主要用于防止蜂鸣器的误发声。当 PB.8 输出高电平的时候，蜂鸣器将发声，当 PB.8 输出低电平的时候，蜂鸣器停止发声。

7.3 软件设计

这里的代码设计，我们还是在之前的基础上继续编写，打开上一章的 TEST 工程，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：

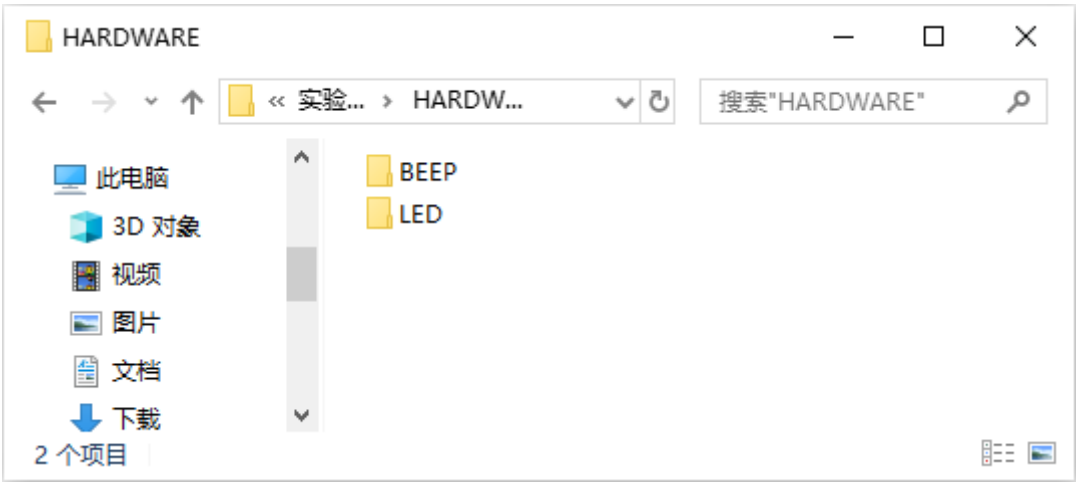


图 7.3.1 在 HARDWARE 下新增 BEEP 文件夹

然后我们打开 USER 文件夹下的 test.uvprojx 工程，按 按钮新建一个文件，然后保存在 HARDWARE→BEEP 文件夹下面，保存为 beep.c。在该文件中输入如下代码：

```
#include "beep.h"
//初始化 PB8 为输出.并使能时钟
//LED IO 初始化
void BEEP_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();              //开启 GPIOB 时钟

    GPIO_Initure.Pin=GPIO_PIN_8;               //PB8
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;     //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;             //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;   //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_8,GPIO_PIN_RESET);
    //蜂鸣器对应引脚 GPIOB8 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 PORTB 的时钟，然后调用 GPIO_Init 函数，配置 PB8 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 PBout(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 PB8 的输出状态。我们只需要令 BEEP=1，就可以让蜂鸣器发声。

将 beep.h 也保存。接着，我们把 beep.c 加入到 HARDWARE 这个组里面，这一次我们通过双击的方式来增加新的.c 文件，双击 HARDWARE，找到 beep.c，加入到 HARDWARE 里面，如图 7.3.2 所示：

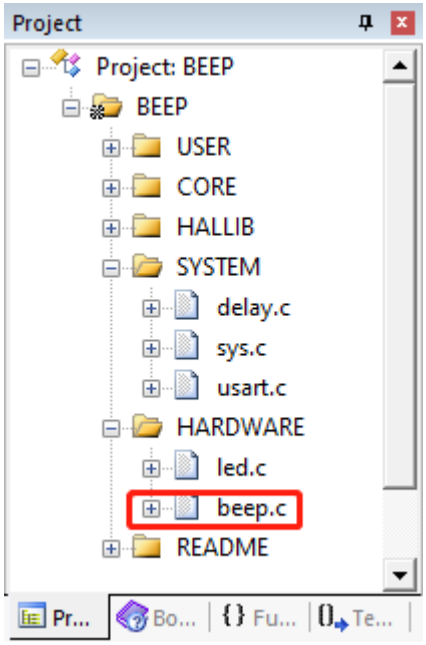


图 7.3.2　将 beep.c 加入 HARDWARE 组下

可以看到 HARDWARE 文件夹里面多了一个 beep.c 的文件，然后还是用老办法（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 "usart.h"
#include "led.h"
#include "beep.h"
int main(void)
{
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    LED_Init();                         //初始化 LED
```

```
    BEEP_Init();                            //初始化蜂鸣器
    while(1)
    {
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_RESET);
                                            //DS0 拉低，亮    等同 LED0=0;
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_8,GPIO_PIN_SET);
                                            //BEEP 引脚拉低，  等同 BEEP=0;
        delay_ms(300);                      //延时 300ms
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET);
                                            //DS0 拉高，灭    等同 LED0=1;
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_8,GPIO_PIN_RESET);
                                           //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 HAL_GPIO_WritePin 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：

```
Build Output
compiling beep.c...
linking...
Program Size: Code=4018 RO-data=442 RW-data=24 ZI-data=1896
FromELF: creating hex file...
"..\OBJ\Template.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:03
```

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 STLINK，则可以用 STLINK 进行在线调试（需要先下载代码），单步查看代码的运行，STM32F1 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 STM32F1 的入门第二个例子，进一步介绍了 STM32F1 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 STM32F1 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 STM32F1 的 IO 口作为输入用。在本章中，我们将利用板载的 3 个按键，来控制板载的两个 LED 的亮灭和蜂鸣器的开关。通过本章的学习，你将了解到 STM32F1 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 STM32F1 IO 口简介
8.2 硬件设计
8.3 软件设计
8.4 下载验证

8.1 STM32F1 IO 口简介

STM32F1 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。STM32F1 的 IO 口做输入使用的时候，是通过读取 IDR 的内容来读取 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 精英 STM32F103 上载有的 3 个按钮（KEY_UP、KEY0 和 KEY1），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；KEY1 控制 DS1，按一次亮，再按一次灭；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） 3 个按键：KEY0、KEY1 和 KEY_UP。

DS0、DS1 以及蜂鸣器和 STM32F1 的连接在上两章都已经分别介绍了，在精英 STM32F103 上的按键 KEY0 连接在 PE4 上、KEY1 连接在 PE3 上、KEY_UP 连接在 PA0 上。如图 8.2.1 所示：

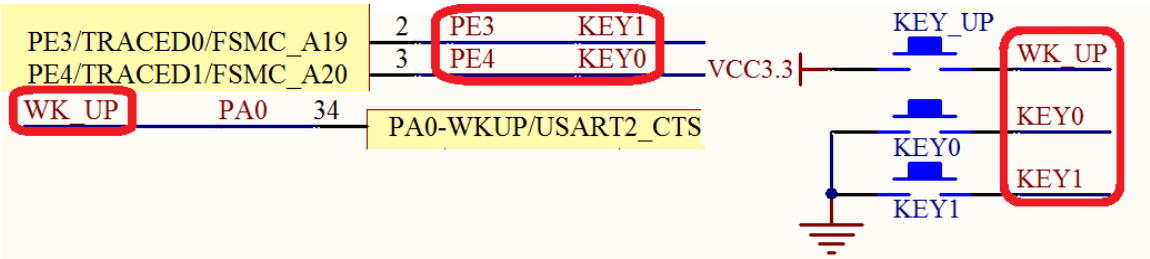


图 8.2.1 按键与 STM32F1 连接原理图

这里需要注意的是：KEY0 和 KEY1 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 STM32F1 内部设置上下拉。

8.3 软件设计

这里的代码设计，我们还是在之前的基础上继续编写，打开上一章的 TEST 工程，然后在 HARDWARE 文件夹下新建一个 KEY 文件夹，用来存放与按键相关的代码。如图 8.3.1 所示：

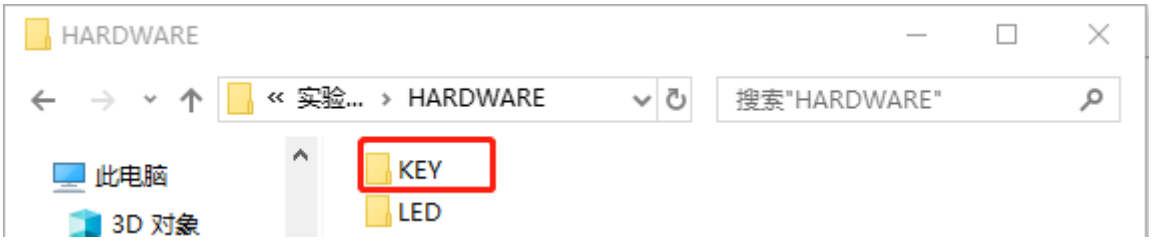


图 8.3.1 在 HARDWARE 下新增 KEY 文件夹

然后我们打开 USER 文件夹下的 TEST.uvprojx 工程，按 按钮新建一个文件，然后保存在 HARDWARE→KEY 文件夹下面，保存为 key.c。在该文件中输入如下代码：

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

```
#include "delay.h"
//按键初始化函数
void KEY_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟
    __HAL_RCC_GPIOE_CLK_ENABLE();           //开启 GPIOE 时钟

    GPIO_Initure.Pin=GPIO_PIN_0;            //PA0
    GPIO_Initure.Mode=GPIO_MODE_INPUT;      //输入
    GPIO_Initure.Pull=GPIO_PULLDOWN;        //下拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    GPIO_Initure.Pin= GPIO_PIN_3|GPIO_PIN_4;  //PE3,4
    GPIO_Initure.Mode=GPIO_MODE_INPUT;      //输入
    GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
    HAL_GPIO_Init(GPIOE,&GPIO_Initure);
}
//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，WKUP 按下 WK_UP
//注意此函数有响应优先级,KEY0>KEY1>WK_UP!!
u8 KEY_Scan(u8 mode)
{
    static u8 key_up=1;     //按键松开标志
    if(mode==1)key_up=1;    //支持连按
    if(key_up&&(KEY0==0||KEY1==0||WK_UP==1))
    {
        delay_ms(10);
        key_up=0;
        if(KEY0==0)         return KEY0_PRES;
        else if(KEY1==0)    return KEY1_PRES;
        else if(WK_UP==1) return WKUP_PRES;
    }else if(KEY0==1&&KEY1==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。

保存 key.c 代码，然后我们按同样的方法，新建一个 key.h 文件，也保存在 KEY 文件夹下面。在 key.h 中输入如下代码：

```
#ifndef __KEY_H
#define __KEY_H
#include "sys.h"
//下面的方式是通过直接操作 HAL 库函数方式读取 IO
//下面的方式是通过直接操作 HAL 库函数方式读取 IO
#define KEY0            HAL_GPIO_ReadPin(GPIOE,GPIO_PIN_4)   //KEY0 按键 PE4
#define KEY1            HAL_GPIO_ReadPin(GPIOE,GPIO_PIN_3)   //KEY1 按键 PE3
#define WK_UP           HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_0)   //WKUP 按键 PA0

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

void KEY_Init(void);
u8 KEY_Scan(u8 mode);
#endif
```

这段代码里面最关键就是 4 个宏定义：

```
#define KEY0            HAL_GPIO_ReadPin(GPIOE,GPIO_PIN_4)   //KEY0 按键 PE4
#define KEY1            HAL_GPIO_ReadPin(GPIOE,GPIO_PIN_3)   //KEY1 按键 PE3
#define WK_UP          HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_0)   //WKUP 按键 PA0
```

这里使用的是位带操作来实现读取某个 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)   //KEY0 按键 PE4
#define KEY1            PEin(3)   //KEY1 按键 PE3
#define WK_UP          PAin(0)   //WKUP 按键 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/WKUP_PRESS 等 4 个宏定义，分别

对应开发板三个按键（KEY0/KEY1/ 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。

将 key.h 也保存一下。接着，我们把 key.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 key.c，加入到 HARDWARE 里面，如图 8.3.2 所示：

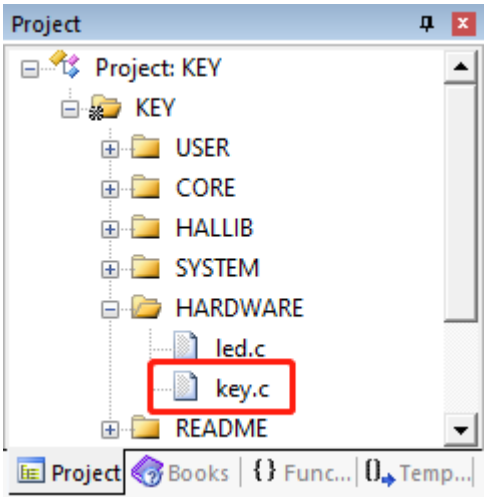


图 8. 3.2　将 key.c 加入 HARDWARE 组下

可以看到 HARDWARE 文件夹里面多了一个 key.c 的文件， 然后还是用老办法把 key.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 "usart.h"
#include "led.h"
#include "key.h"

int main(void)
{
    u8 key;

    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键

    while(1)
    {
        key=KEY_Scan(0);                    //按键扫描
        switch(key)
        {
            case WKUP_PRES:                 //控制 LED0,LED1 互斥点亮
                LED1=!LED1;
```

```
                LED0=!LED1;
                break;
            case KEY1_PRES:         //控制 LED1 翻转
                LED1=!LED1;
                break;
            case KEY0_PRES:         //同时控制 LED0,LED1 翻转
                LED0=!LED0;
                LED1=!LED1;
                break;
        }
        delay_ms(10);
    }
}
```

注意要将 KEY 文件夹加入头文件包含路径，不能少，否则编译的时候会报错。这段代码就实现前面 8.1 节所阐述的功能，相对比较简单。

然后按  ，编译工程，得到结果如图 8.3.3 所示：

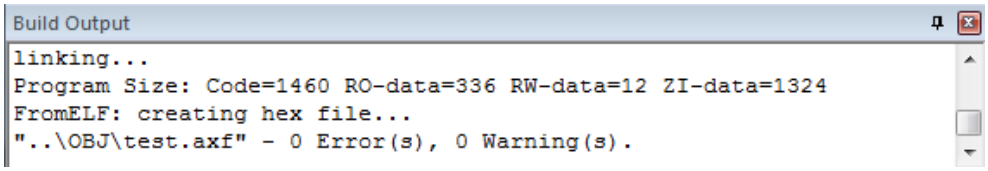


图 8.3.3 编译结果

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。

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 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 STM32F1 的入门第三个例子，介绍了 STM32F1 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

前面两章介绍了 STM32F1 的 IO 口操作。这一章我们将学习 STM32F1 的串口，教大家如何使用 STM32F1 的串口来发送和接收数据。本章将实现如下功能：STM32F1 通过串口和上位机的对话，STM32F1 在收到上位机发过来的字符串后，原原本本的返回给上位机。本章分为如下几个小节：

9.1 STM32F1 串口简介
9.2 硬件设计
9.3 软件设计
9.4 下载验证

9.1 STM32F1 串口简介

串口作为 MCU 的重要外部接口，同时也是软件开发重要的调试手段，其重要性不言而喻。现在基本上所有的 MCU 都会带有串口，STM32 自然也不例外。

STM32 的串口资源相当丰富的，功能也相当强劲。ALIENTEK 精英 STM32 开发板所使用的 STM32F103ZET6 最多可提供 5 路串口，有分数波特率发生器、支持同步单线通信和半双工单线通讯、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范、具有 DMA 等。

5.3 节对串口有过简单的介绍，接下来我们将从寄存器层面，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。精英 STM32 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。

串口最基本的设置，就是波特率的设置。STM32 的串口使用起来还是蛮简单的，只要你开启了串口时钟，并设置相应 IO 口的模式，然后配置一下波特率，数据位长度，奇偶校验位等信息，就可以使用了，详见 5.3.2 节。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寄存器。

1，串口时钟使能。串口作为 STM32 的一个外设，其时钟由外设时钟使能寄存器控制，这里我们使用的串口 1 是在 APB2ENR 寄存器的第 14 位。APB2ENR 寄存器在之前已经介绍过了，这里不再介绍。只是说明一点，就是除了串口 1 的时钟使能在 APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR 寄存器。

2，串口复位。当外设出现异常的时候可以通过复位寄存器里面的对应位设置，实现该外设的复位，然后重新配置这个外设达到让其重新工作的目的。一般在系统刚开始配置外设的时候，都会先执行复位该外设的操作。串口 1 的复位是通过配置 APB2RSTR 寄存器的第 14 位来实现的。APB2RSTR 寄存器的各位描述如图 9.1.1 所示：

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

图 9.1.1 APB2RSTR 寄存器各位描述

从图 9.1.1 可知串口 1 的复位设置位在 APB2RSTR 的第 14 位。通过向该位写 1 复位串口 1，写 0 结束复位。其他串口的复位位在 APB1RSTR 里面。

3，串口波特率设置。在 5.3.2 节，我们已经介绍过了，每个串口都有一个自己独立的波特率寄存器 USART_BRR，通过设置该寄存器就可以达到配置不同波特率的目的。具体实现方法，请参考 5.3.2 节。

4，串口控制。STM32 的每个串口都有 3 个控制寄存器 USART_CR1~3，串口的很多配置都是通过这 3 个寄存器来设置的。这里我们只要用到 USART_CR1 就可以实现我们的功能了，该寄存器的各位描述如图 9.1.2 所示：



图 9.1.2 USART_CR 寄存器各位描述

该寄存器的高 18 位没有用到，低 14 位用于串口的功能设置。UE 为串口使能位，通过该位置 1，以使能串口。M 为字长选择位，当该位为 0 的时候设置串口为 8 个字长外加 n 个停止位，停止位的个数（n）是根据 USART_CR2 的[13:12]位设置来决定的，默认为 0。PCE 为校验使能位，设置为 0，则禁止校验，否则使能校验。PS 为校验位选择，设置为 0 则为偶校验，否则为奇校验。TXIE 为发送缓冲区空中断使能位，设置该位为 1，当 USART_SR 中的 TXE 位为 1 时，将产生串口中断。TCIE 为发送完成中断使能位，设置该位为 1，当 USART_SR 中的 TC 位为 1 时，将产生串口中断。RXNEIE 为接收缓冲区非空中断使能，设置该位为 1，当 USART_SR 中的 ORE 或者 RXNE 位为 1 时，将产生串口中断。TE 为发送使能位，设置为 1，将开启串口的发送功能。RE 为接收使能位，用法同 TE。

其他位的设置，这里就不一一列出来了，大家可以参考《STM32 中文参考手册》第 542 页有详细介绍，在这里我们就不列出来了。

5，数据发送与接收。STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和 RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到数据的时候，也是存在该寄存器内。该寄存器的各位描述如图 9.1.3 所示：



图 9.1.3 USART_DR 寄存器各位描述

可以看出，虽然是一个 32 位寄存器，但是只用了低 9 位（DR[8：0]），其他都是保留。

DR[8：0]为串口数据，包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。

当使能校验位(USART_CR1 中 PCE 位被置位)进行发送时，写到 MSB 的值(根据数据的长度不同，MSB 是第 7 位或者第 8 位)会被后来的校验位取代。

当使能校验位进行接收时，读到的 MSB 位是接收到的校验位。

6，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 9.1.4 所示：



图 9.1.4 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。

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《STM32 中文参考手册》第 516 页至 548 页，通用同步异步收发器一章。

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 USARTxIRQHandler(x 对应串口号)。

接下来我们将着重讲解使用 HAL 库实现串口配置和使用的方法。在 HAL 库中，串口相关的函数和定义主要在文件 stm32f1xx_hal_uart.c 和 stm32f1xx_hal_uart.h 中。接下来我们看看 HAL 库提供的串口相关操作函数。

1) 串口参数初始化（波特率/停止位等），并使能串口。

串口作为 STM32 的一个外设，HAL 库为其配置了串口初始化函数。接下来我们看看串口初始化函数 HAL_UART_Init 相关知识，定义如下：

```
HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart);
```

该函数只有一个入口参数 huart，为 UART_HandleTypeDef 结构体指针类型，我们俗称其为串口句柄，它的使用会贯穿整个串口程序。一般情况下，我们会定义一个 UART_HandleTypeDef 结构体类型全局变量，然后初始化各个成员变量。接下来我们看看结构体 UART_HandleTypeDef 的定义：

```
typedef struct
{
  USART_TypeDef                 *Instance;
  UART_InitTypeDef              Init;
  uint8_t                       *pTxBuffPtr;
  uint16_t                      TxXferSize;
  __IO uint16_t                 TxXferCount;
```

```
    uint8_t                     *pRxBuffPtr;
    uint16_t                    RxXferSize;
    __IO uint16_t               RxXferCount;
    DMA_HandleTypeDef           *hdmatx;
    DMA_HandleTypeDef           *hdmarx;
    HAL_LockTypeDef             Lock;
    __IO HAL_UART_StateTypeDef  gState;
    __IO HAL_UART_StateTypeDef  RxState;
    __IO uint32_t               ErrorCode;
}UART_HandleTypeDef;
```

该结构体成员变量非常多，一般情况下载调用函数 HAL_UART_Init 对串口进行初始化的时候，我们只需要先设置 Instance 和 Init 两个成员变量的值。接下来我们依次解释一下各个成员变量的含义。

Instance 是 USART_TypeDef 结构体指针类型变量，它是执行寄存器基地址，实际上这个基地址 HAL 库已经定义好了，如果是串口 1，取值为 USART1 即可。

Init 是 UART_InitTypeDef 结构体类型变量，它是用来设置串口的各个参数，包括波特率，停止位等，它的使用方法非常简单。UART_InitTypeDef 结构体定义如下：

```
typedef struct
{
    uint32_t BaudRate;          //波特率
    uint32_t WordLength;        //字长
    uint32_t StopBits;          //停止位
    uint32_t Parity;            //奇偶校验
    uint32_t Mode;              //收/发模式设置
    uint32_t HwFlowCtl;         //硬件流设置
    uint32_t OverSampling;      //过采样设置
}UART_InitTypeDef
```

该结构体第一个参数 BaudRate 为串口波特率，波特率可以说是串口最重要的参数了，它用来确定串口通信的速率。第二个参数 WordLength 为字长，可以设置为 8 位字长或者 9 位字长，这里我们设置为 8 位字长数据格式 UART_WORDLENGTH_8B。第三个参数 StopBits 为停止位设置，可以设置为 1 个停止位或者 2 个停止位，这里我们设置为 1 位停止位 UART_STOPBITS_1。第四个参数 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 Mode 为串口模式，可以设置为只收模式，只发模式，或者收发模式。这里我们设置为全双工收发模式。第六个参数 HwFlowCtl 为是否支持硬件流控制，我们设置为无硬件流控制。第七个参数 OverSampling 用来设置过采样为 16 倍还是 8 倍。

pTxBuffPtr，TxXferSize 和 TxXferCount 三个变量分别用来设置串口发送的数据缓存指针，发送的数据量和还剩余的要发送的数据量。而接下来的三个变量 pRxBuffPtr，RxXferSize 和 RxXferCount 则是用来设置接收的数据缓存指针，接收的最大数据量以及还剩余的要接收的数据量。这六个变量是 HAL 库处理中间变量，详细使用方法在我们讲解中断服务函数的时候给大家讲解。

hdmatx 和 hdmarx 是串口 DMA 相关的变量，指向 DMA 句柄，这里我们先不讲解。

其他的三个变量就是一些 HAL 库处理过程状态标志位和串口通信的错误码。

函数 HAL_UART_Init 使用的一般格式为：

```
UART_HandleTypeDef UART1_Handler; //UART 句柄

UART1_Handler.Instance=USART1;                          //USART1
UART1_Handler.Init.BaudRate=115200;                     //波特率
UART1_Handler.Init.WordLength=UART_WORDLENGTH_8B;       //字长为 8 位格式
UART1_Handler.Init.StopBits=UART_STOPBITS_1;            //一个停止位
UART1_Handler.Init.Parity=UART_PARITY_NONE;             //无奇偶校验位
UART1_Handler.Init.HwFlowCtl=UART_HWCONTROL_NONE;       //无硬件流控
UART1_Handler.Init.Mode=UART_MODE_TX_RX;                //收发模式
HAL_UART_Init(&UART1_Handler); //HAL_UART_Init()会使能 UART1
```

这里我们需要说明的是，函数 HAL_UART_Init 内部会调用串口使能函数使能相应串口，所以调用了该函数之后我们就不需要重复使能串口了。当然，HAL 库也提供了具体的串口使能和关闭方法，具体使用方法如下：

```
__HAL_UART_ENABLE(handler);     //使能句柄 handler 指定的串口
__HAL_UART_DISABLE(handler);    //关闭句柄 handler 指定的串口
```

这里还需要提醒大家，串口作为一个重要外设，在调用的初始化函数 HAL_UART_Init 内部，会先调用 MSP 初始化回调函数进行 MCU 相关的初始化，函数为：

```
void HAL_UART_MspInit(UART_HandleTypeDef *huart);
```

我们在程序中，只需要重写该函数即可。一般情况下，该函数内部用来编写 IO 口初始化，时钟使能以及 NVIC 配置。

2）使能串口和 GPIO 口时钟

我们要使用串口，所以我们必须使能串口时钟和使用到的 GPIO 口时钟。例如我们要使用串口 1，所以我们必须使能串口 1 时钟和 GPIOA 时钟（串口 1 使用的是 PA9 和 PA10）。具体方法如下：

```
__HAL_RCC_USART1_CLK_ENABLE();          //使能 USART1 时钟
__HAL_RCC_GPIOA_CLK_ENABLE();           //使能 GPIOA 时钟
```

使能使能相关方法我们在时钟系统相关章节有讲解，操作方法也非常简单，这里我们就不重复讲解。

3）GPIO 口初始化设置（速度，上下拉等）以及复用映射配置

我们在跑马灯实验中讲解过，在 HAL 库中 IO 口初始化参数设置和复用映射配置是在函数 HAL_GPIO_Init 中一次性完成的。这里大家只需要注意，我们要复用 PA9 和 PA10 为串口发送接收相关引脚，我们需要配置 IO 口为复用，同时复用映射到串口 1。配置源码如下：

```
GPIO_Iniure.Pin=GPIO_PIN_9;                 //PA9
GPIO_Iniure.Mode=GPIO_MODE_AF_PP;           //复用推挽输出
GPIO_Iniure.Pull=GPIO_PULLUP;               //上拉
GPIO_Iniure.Speed=GPIO_SPEED_FREQ_HIGH;     //高速
HAL_GPIO_Init(GPIOA,&GPIO_Iniure);          //初始化 PA9

GPIO_Iniure.Pin=GPIO_PIN_10;                //PA10
GPIO_Iniure.Mode=GPIO_MODE_AF_INPUT;        //模式要设置为复用输入模式！
HAL_GPIO_Init(GPIOA,&GPIO_Iniure);          //初始化 PA10
```

4）开启串口相关中断，配置串口中断优先级

HAL 库中定义了一个使能串口中断的标识符__HAL_UART_ENABLE_IT，大家可以把它当

一个函数来使用，具体定义请参考 HAL 库文件 stm32f1xx_hal_uart.h 中该标识符定义。例如我们要使能接收完成中断，方法如下：

```
__HAL_UART_ENABLE_IT(huart,UART_IT_RXNE);          //开启接收完成中断
```

第一个参数为我们步骤 1 讲解的串口句柄，类型为 UART_HandleTypeDef 结构体类型。第二个参数为我们要开启的中断类型值，可选值在头文件 stm32f1xx_hal_uart.h 中有宏定义。

有开启中断就有关闭中断，操作方法为：

```
__HAL_UART_DISABLE_IT(huart,UART_IT_RXNE);         //关闭接收完成中断
```

对于中断优先级配置，方法就非常简单，详细知识情参考 4.5 小节相关知识。参考方法为：

```
HAL_NVIC_EnableIRQ(USART1_IRQn);                   //使能 USART1 中断通道
HAL_NVIC_SetPriority(USART1_IRQn,3,3);             //抢占优先级 3，子优先级 3
```

5） 编写中断服务函数

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们相应的逻辑代码即可。HAL 库实际上对中断处理过程进行了完整的封装，具体内容我们在 8.3 小节通过结合实验源码给大家详细讲解。

6） 串口数据接收和发送

STM32F1 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。HAL 库操作 USART_DR 寄存器发送数据的函数是：

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart,
                                    uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

通过该函数向串口寄存器 USART_DR 写入一个数据。

HAL 库操作 USART_DR 寄存器读取串口接收到的数据的函数是：

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart,
                                   uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

通过该函数可以读取串口接受到的数据。

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 P3 的 RXD 和 TXD 用跳线帽与 PA9 和 PA10 连接起来。如图 9.2.1 所示：

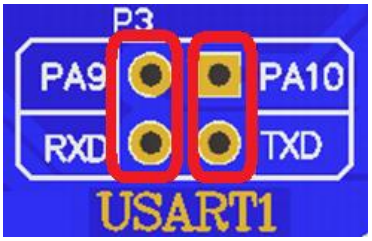


图 9.2.1 硬件连接图示意图

连接上这里之后，我们在硬件上就设置完成了，可以开始软件设计了。

9.3 软件设计

本章的代码设计，比前两章简单很多，因为我们的串口初始化代码和接收代码就是用我们之前介绍的 SYSTEM 文件夹下的串口部分的内容。这里我们对代码部分稍作讲解。

打开串口实验工程，然后在 SYSTEM 组下双击 usart.c，我们就可以看到该文件里面的代码，先介绍 uart_init 函数，该函数代码如下：

```
//初始化 IO 串口 1
//bound:波特率
void uart_init(u32 bound)
{
    //UART 初始化设置
    UART1_Handler.Instance=USART1;                          //USART1
    UART1_Handler.Init.BaudRate=bound;                      //波特率
    UART1_Handler.Init.WordLength=UART_WORDLENGTH_8B; //字长为 8 位数据格式
    UART1_Handler.Init.StopBits=UART_STOPBITS_1;            //一个停止位
    UART1_Handler.Init.Parity=UART_PARITY_NONE;             //无奇偶校验位
    UART1_Handler.Init.HwFlowCtl=UART_HWCONTROL_NONE;   //无硬件流控
    UART1_Handler.Init.Mode=UART_MODE_TX_RX;                //收发模式
    HAL_UART_Init(&UART1_Handler);              //HAL_UART_Init()会使能 UART1
    HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, RXBUFFERSIZE);
    //该函数会开启接收中断：标志位 UART_IT_RXNE，并且设置接收缓冲以
    //及接收缓冲接收最大数据量
}
```

该函数实现的是我们 8.1 小节讲解的步骤 1 的内容。同时这里大家需要注意，最后一行代码调用函数 HAL_UART_Receive_IT，作用是开启接收中断，同时设置接收的缓存区以及接收的数据量，对于这个缓冲我们在后面会给大家讲解它的作用。

串口 MSP 函数 HAL_UART_MspInit 函数我们自定义了其内容，代码如下：

```
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
    //GPIO 端口设置
    GPIO_InitTypeDef GPIO_Initure;
    if(huart->Instance==USART1) //如果是串口 1，进行串口 1 MSP 初始化
    {
        __HAL_RCC_GPIOA_CLK_ENABLE();           //使能 GPIOA 时钟
        __HAL_RCC_USART1_CLK_ENABLE();          //使能 USART1 时钟
        __HAL_RCC_AFIO_CLK_ENABLE();

        GPIO_Initure.Pin=GPIO_PIN_9;            //PA9
        GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //复用推挽输出
        GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
        GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
        HAL_GPIO_Init(GPIOA,&GPIO_Initure);     //初始化 PA9

        GPIO_Initure.Pin=GPIO_PIN_10;           //PA10
```

```
        GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;//模式要设置为复用输入模式！
        HAL_GPIO_Init(GPIOA,&GPIO_Initure);        //初始化 PA10
#if EN_USART1_RX
        HAL_NVIC_EnableIRQ(USART1_IRQn);           //使能 USART1 中断通道
        HAL_NVIC_SetPriority(USART1_IRQn,3,3);     //抢占优先级 3，子优先级 3
#endif
    }
}
```

该函数代码实现的是我们 8.1 小节讲解的步骤 2 到 4 的内容。这里大家需要注意，在该段代码中，通过判断宏定义标识符 EN_USART1_RX 的值来确定是否开启串口中断通道和设置串口 1 中断优先级。标识符 EN_USART1_RX 在头文件 usart.h 中有定义，默认情况下我们设置为 1。

```
#define EN_USART1_RX    1    //使能（1）/禁止（0）串口 1 接收
```

通过上面两个函数，我们就配置了串口相关设置。接下来就是编写中断服务函数 USART1_IRQHandler。而 HAL 库中，对中断服务函数的编写有非常严格的讲究。

首先 HAL 库定义了一个串口中断处理通用函数 HAL_UART_IRQHandler，该函数声明如下：

```
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart);
```

该函数只有一个入口参数就是 UART_HandleTypeDef 结构体指针类型的串口句柄 huart，使用我们在调用 HAL_UART_Init 函数时设置的同一个变量即可。该函数一般在中断服务函数中调用，作为串口中断处理的通用入口。一般调用方法为：

```
void USART1_IRQHandler(void)
{
    HAL_UART_IRQHandler(&UART1_Handler);  //调用 HAL 库中断处理公用函数
    …//中断处理完成后的结束工作
}
```

也就是说，真正的串口中断处理逻辑我们会最终在函数 HAL_UART_IRQHandler 内部执行。而该函数是 HAL 库已经定义好，而且用户一般不能随意修改。这个时候大家会问，那么我们的中断控制逻辑编写在哪里呢？为了把这个问题讲解清楚，我们要来看看函数 HAL_UART_IRQHandler 内部具体实现过程。因为本章实验，我们主要实现的是串口中断接收，也就是每次接收到一个字符后进入中断服务函数来处理。所以我们就以中断接收为例给大家讲解。这里为了篇幅考虑，我们仅仅列出串口中断执行流程中与接收相关的源码。

函数 HAL_UART_IRQHandler 关于串口接收相关源码如下：

```
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
{
  uint32_t isrflags   = READ_REG(huart->Instance->SR);
  uint32_t cr1its     = READ_REG(huart->Instance->CR1);
  uint32_t cr3its     = READ_REG(huart->Instance->CR3);
  uint32_t errorflags = 0x00U;
  uint32_t dmarequest = 0x00U;
  errorflags = (isrflags & (uint32_t)(USART_SR_PE | USART_SR_FE |
                                      USART_SR_ORE | USART_SR_NE));
  if (errorflags == RESET)
  {
```

```
            if (((isrflags & USART_SR_RXNE) != RESET) &&
                                        ((cr1its & USART_CR1_RXNEIE) != RESET))
        {
          UART_Receive_IT(huart);
          return;
        }
      }
    …//此处省略部分代码
    }
```

从代码逻辑可以看出，在函数 HAL_UART_IRQHandler 内部通过判断中断类型是否为接收完成中断，确定是否调用 HAL 另外一个函数 UART_Receive_IT()。函数 UART_Receive_IT()的作用是把每次中断接收到的字符保存在串口句柄的缓存指针 pRxBuffPtr 中，同时每次接收一个字符，其计数器 RxXferCount 减 1，直到接收完成 RxXferSize 个字符之后 RxXferCount 设置为 0，同时调用接收完成回调函数 HAL_UART_RxCpltCallback 进行处理。为了篇幅考虑，这里我们仅列出 UART_Receive_IT()函数调用回调函数 HAL_UART_RxCpltCallback 的处理逻辑，代码如下：

```
static HAL_StatusTypeDef UART_Receive_IT(UART_HandleTypeDef *huart)
{
    ...//此处省略部分代码
    #if (USE_HAL_UART_REGISTER_CALLBACKS == 1)
            huart->RxCpltCallback(huart);
    #else
            HAL_UART_RxCpltCallback(huart);
    #endif
    ...//此处省略部分代码
}
```

最后我们列出串口接收中断的一般流程，如图 8.3.1 所示：

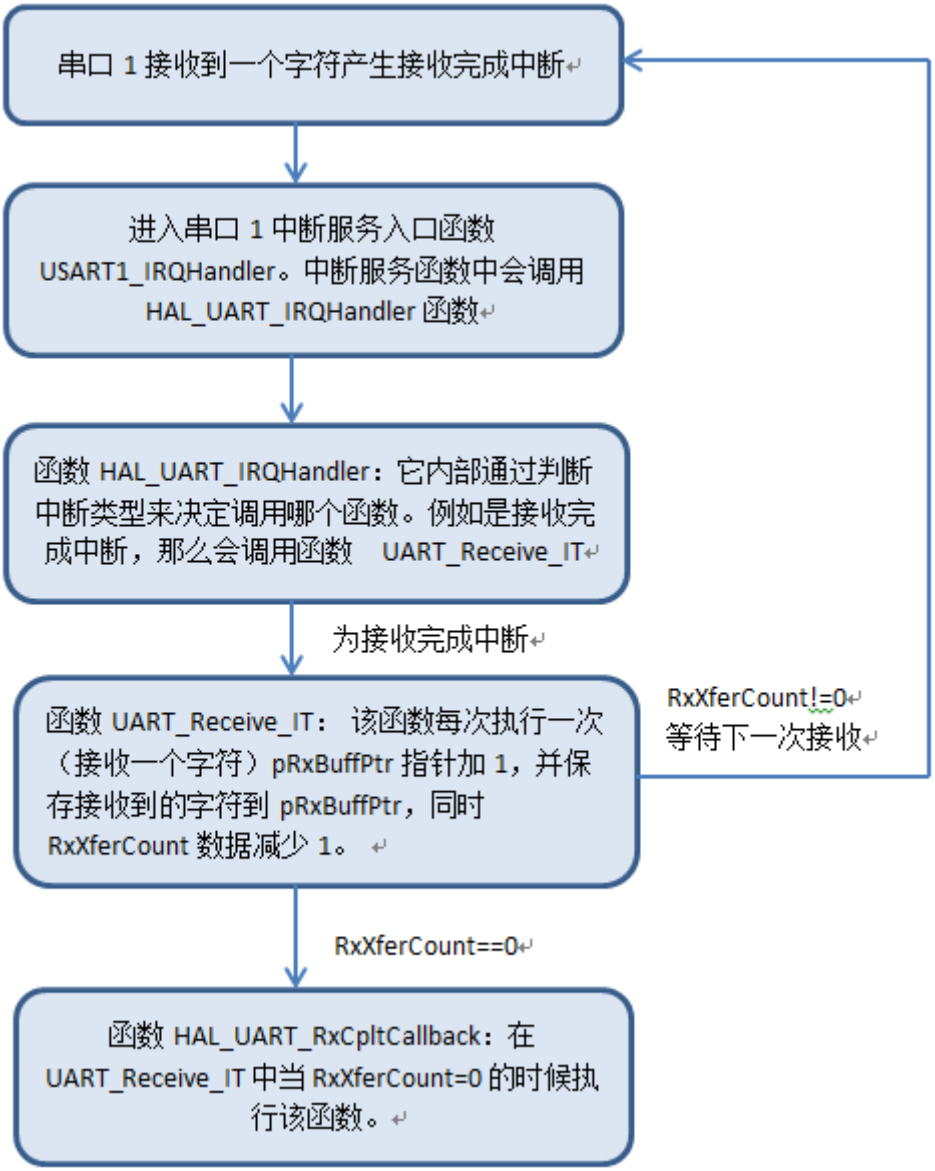


图 8.3.1 串口接收中断执行流程图

这里，我们再把串口接收中断的一般流程进行概括：当接收到一个字符之后，在函数 UART_Receive_IT 中会把数据保存在串口句柄的成员变量 pRxBuffPtr 缓存中，同时 RxXferCount 计数器减 1。如果我们设置 RxXferSize=10,那么当接收到 10 个字符之后，RxXferCount 会由 10 减到 0（RxXferCount 初始值等于 RxXferSize），这个时候再调用接收完成回调函数 HAL_UART_RxCpltCallback 进行处理。接下来我们看看我们的配置。

首先，我们回到用户函数 uart_init 定义可以看到，在 uart_init 函数中调用完 HAL_UART_Init 后我们还调用了 HAL_UART_Receive_IT 开启接收中断，并且初始化串口句柄的缓存相关参数。代码如下：

```
HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, RXBUFFERSIZE);
```

而 aRxBuffer 是我们定义的一个全局数组变量，RXBUFFERSIZE 是我们定义的一个标识符：

```
#define RXBUFFERSIZE    1
u8 aRxBuffer[RXBUFFERSIZE];
```

所以，调用 HAL_UART_Receive_IT 函数后，除了开启接收中断外还确定了每次接收

RXBUFFERSIZE 个字符后标示接收结束从而进入回调函数 HAL_UART_RxCpltCallback 进行相应处理。最后我们看看 HAL_UART_RxCpltCallback 函数定义：

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance==USART1)//如果是串口 1
      if((USART_RX_STA&0x8000)==0)//接收未完成
        {
            if(USART_RX_STA&0x4000)//接收到了 0x0d
            {
                if(aRxBuffer[0]!=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(aRxBuffer[0]==0x0d)USART_RX_STA|=0x4000;
                else
                {
                    USART_RX_BUF[USART_RX_STA&0X3FFF]=aRxBuffer[0] ;
                    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
                }
            }
        }
    }
}
```

因为我们设置了串口句柄成员变量 RxXferSize 为 1，也就是每当串口 1 发生了接收完成中断后（接收到一个字符），就会跳到该函数执行。当串口接受到一个字符后，它会保存在缓存 aRxBuffer 中，由于我们设置了缓存大小为 1，而且 RxXferSize=1，所以每次接受一个字符，回直接保存到 RxXferSize[0]中，我们直接通过读取 RxXferSize[0]的值就是本次接收到的字符。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 8.3.2 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 8.3.2 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。

在函数 USART1_IRQHandler 的结尾还有几行行代码，其中部分代码是超时退出逻辑，关键逻辑代码如下：

```
while (HAL_UART_GetState(&UART1_Handler) != HAL_UART_STATE_READY);
while(HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, 1) != HAL_OK);
```

这两行代码作用非常简单。第一行代码是判断串口是否就绪，如果没有就绪就等待就绪。第二行代码是继续调用 HAL_UART_Receive_IT 函数来开启中断和重新设置 RxXferSize 和 RxXferCount 的初始值为 1，也就是开启新的接收中断。

学到这里大家会发现，HAL 库定义的串口中断逻辑确实非常复杂，并且因为处理过程繁琐所以效率不高。这里我们需要说明的是，在中断服务函数中，大家也可以不用调用 HAL_UART_IRQHandler 函数，而是直接编写自己的中断服务函数。串口实验我们之所以遵循 HAL 库写法， 是为了让大家对 HAL 库有一个更清晰的理解。

如果我们不用中断处理回调函数，那么就不用初始化串口句柄的中断接收缓存，所以我们 HAL_UART_Receive_IT 函数就不用出现在初始化函数 uart_init 中，而是直接在要开启中断的地方通过调用__HAL_UART_ENABLE_IT 单独开启中断即可。如果不用中断回调函数处理，中断服务函数内容为：

```
//串口 1 中断服务程序
void USART1_IRQHandler(void)
{
    u8 Res;
#if SYSTEM_SUPPORT_OS       //使用 OS
    OSIntEnter();
#endif
    if((__HAL_UART_GET_FLAG(&UART1_Handler,UART_FLAG_RXNE)!=RESET))
                            //接收中断(接收到的数据必须是 0x0d 0x0a 结尾)
    {
        HAL_UART_Receive(&UART1_Handler,&Res,1,1000);
        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
                }
            }
        }
    }
    HAL_UART_IRQHandler(&UART1_Handler);
#if SYSTEM_SUPPORT_OS       //使用 OS
    OSIntExit();
#endif
}
```

这段代码逻辑跟上面的中断回调函数类似，只不过这里还需要通过 HAL 库串口接收函数 HAL_UART_Receive 来获取接收到的字符进行相应的处理，这里我们就不做过多讲解。**在我们后面很多实验，为了效率和处理逻辑方便，我们会选择将接收控制逻辑直接编写在中断服务函数内部。**

HAL 库一共提供了 5 个中断处理回调函数：

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);//发送完成回调函数
void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart);//发送完成过半
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);//接收完成回调函数
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart);//接收完成过半
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart);//错误处理回调函数
```

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 len;
    u16 times=0;
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);   //设置时钟,72M
    delay_init(72);                 //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    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");
            HAL_UART_Transmit(&UART1_Handler,(uint8_t*)USART_RX_BUF,len,1000);
                                                    //发送接收到的数据
```

```
            while(__HAL_UART_GET_FLAG(&UART1_Handler,UART_FLAG_TC)!=SET);
                                                    //等待发送结束
            printf("\r\n\r\n");//插入换行
            USART_RX_STA=0;
        }else
        {
            times++;
            if(times%5000==0)
            {
                printf("\r\nALIENTEK 精英 STM32 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);
        }
    }
}
```

这段代码逻辑比较简单，首先判断全局变量 USART_RX_STA 的最高位是否为 1，如果为 1 的话，那么代表前一次数据接收已经完成，接下来就是把我们自定义接收缓冲的数据发送到串口。接下来我们重点以下两句：

```
HAL_UART_Transmit(&UART1_Handler,(uint8_t*)USART_RX_BUF,len,1000);
while(__HAL_UART_GET_FLAG(&UART1_Handler,UART_FLAG_TC)!=SET);
```

第一句，其实就是调用 HAL 串口发送函数 HAL_UART_Transmit 来发送一个字符到串口。第二句呢，就是我们发送一个字节之后之后，要检测这个数据是否已经被发送完成了。

9.4 下载验证

前面两章实例，我们均介绍了软件仿真，仿真的基本技巧也差不多介绍完了，接下来我们将淡化这部分，因为代码都是经过作者检验，并且全部在 ALIENTEK 精英 STM32 V1 开发板上验证了的，有兴趣的朋友可以自己仿真看看。但是这里要说明几点：

1，IO 口复用的，信号在逻辑分析窗口是不能显示出来的（看不到波形），这一点请大家注意。比如串口的输出，SPI，USB，CAN 等。你在仿真的时候在该窗口看不到任何信息。遇到这样的情况，你就不得不准备一个逻辑分析仪，外加一个 ULINK 或者 JLINK 来做在线调试。但一般情况，这些都是有现成的例子，不用这几个东西一般也能编出来。

2，仿真并不能代表实际情况。只能从某些方面给你一些启示，告诉你大方向，不能尽信仿真，当然也不能完全没有仿真。比如上面 IO 口的输出，仿真的时候，其翻转速度可以达到很快，但是实际上 STM32 的 IO 输出就达不到这个速度。

总之，我们要合理的利用仿真，也不能过于依赖仿真。当仿真解决不了了，可以试试在线调试，在线调试一般都可以知道问题在哪个地方，但是问题要怎么解决还是得各位自己动脑筋、找资料了。

把程序下载到到精英 STM32 V1 板子，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，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 及以上系统直接自带了)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，STM32F1 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：

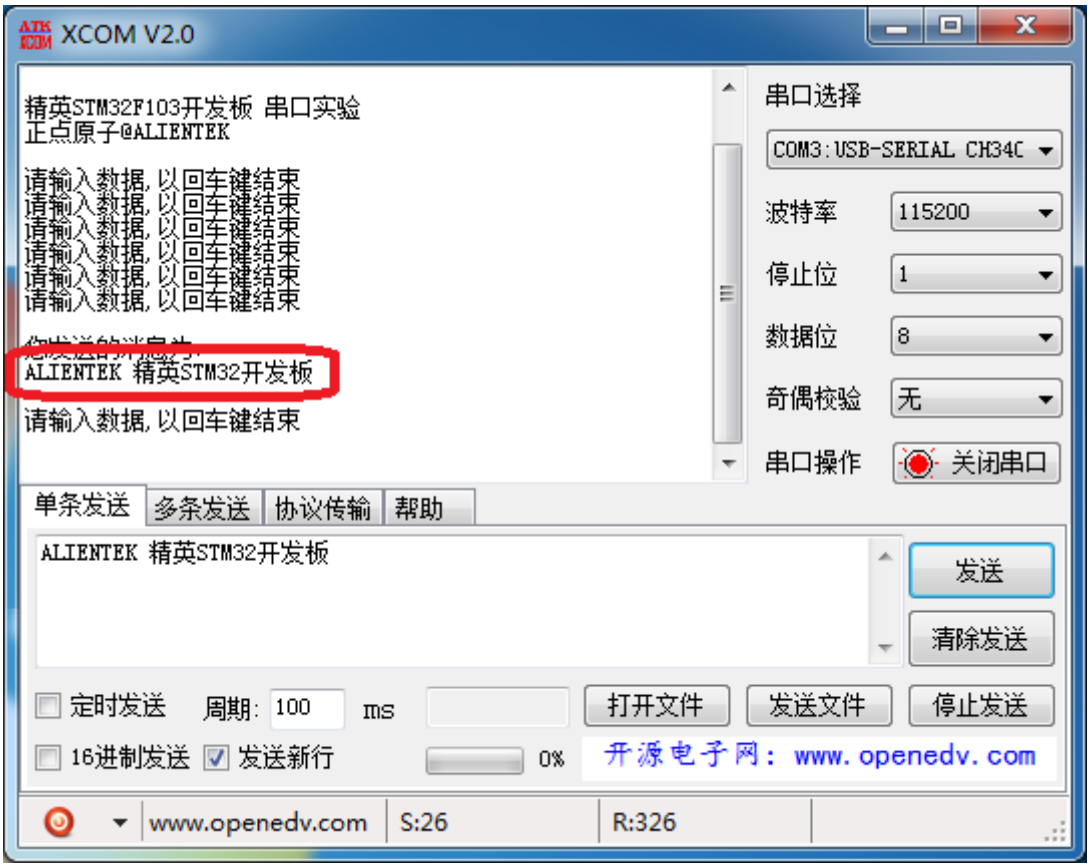


图 9.4.2 发送数据后收到的数据

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 STM32F1 的外部输入中断。在前面几章的学习中，我们掌握了 STM32F1 的 IO 口最基本的操作。本章我们将介绍如何将 STM32F1 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 STM32F1 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 STM32F1 外部中断简介

STM32F1 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 STM32F1 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。

这章的代码主要分布在 HAL 库的 stm32f1xx_hal_exti.h 和 stm32f1xx_hal_exti.c 文件中。

这里我们首先讲解 STM32F1 IO 口中断的一些基础概念。STM32F1 的每个 IO 都可以作为外部中断的中断输入口，这点也是 STM32F1 的强大之处。STM32F103 的中断控制器支持 19 个外部中断/事件请求。每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置。STM32F103 的 19 个外部中断为：

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 唤醒事件。

EXTI 线 19：连接到以太网唤醒事件。

从上面可以看出，STM32F1 供 IO 口使用的中断线只有 16 个，但是 STM32F1 的 IO 口却远远不止 16 个，那么 STM32F1 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。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：

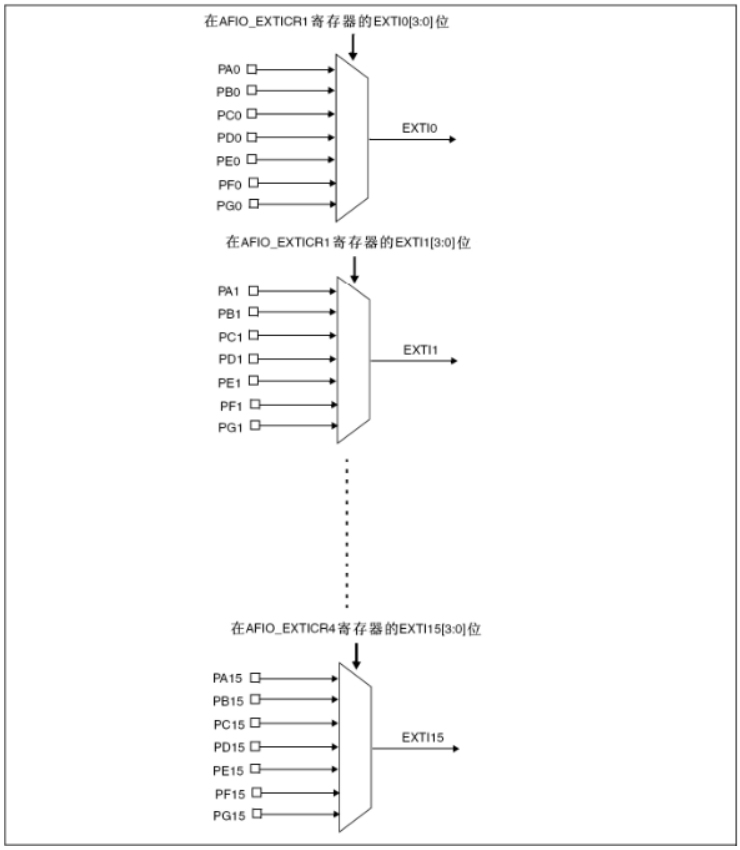


图 10.1.1 GPIO 和中断线的映射关系图

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，具体的操作方法跟我们按键实验是一致的，这里就不做过多讲解。

1） 设置 IO 口模式，触发条件，开启 SYSCFG 时钟，设置 IO 口与中断线的映射关系。

该步骤如果我们使用标准库那么需要多个函数分部实现。而当我们使用 HAL 库的时候，则都是在函数 HAL_GPIO_Init 中一次性完成的。例如我们要设置 PA0 链接中断线 0，并且为上升沿触发，代码为：

```
GPIO_InitTypeDef GPIO_Initure;
GPIO_Initure.Pin=GPIO_PIN_0;                    //PA0
GPIO_Initure.Mode=GPIO_MODE_IT_RISING;          //外部中断，上升沿触发
GPIO_Initure.Pull=GPIO_PULLDOWN;                //默认下拉
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

当我们调用 HAL_GPIO_Init 设置 IO 的 Mode 值为 GPIO_MODE_IT_RISING（外部中断上升沿触发），GPIO_MODE_IT_FALLING（外部中断下降沿触发）或者 GPIO_MODE_IT_RISING_FALLING（外部中断双边沿触发）的时候，该函数内部会通过判断 Mode 的值来开启 SYSCFG 时钟，并且设置 IO 口和中断线的映射关系。

因为我们这里初始化的是 PA0，根据图 10.1.1 可知，调用该函数后中断线 0 会自动连接到 PA0。如果某个时间，我们又同样的方式初始化了 PB0，那么 PA0 与中断线的链接将被清除，而直接链接 PB0 到中断线 0。

2） 配置中断优先级（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 0 的中断优先级并使能外部中断 0 的方法为：

```
HAL_NVIC_SetPriority(EXTI0_IRQn,2,0);          //抢占优先级为 2，子优先级为 0
HAL_NVIC_EnableIRQ(EXTI0_IRQn);                  //使能中断线 2
```

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。

3） 编写中断服务函数。

我们配置完中断优先级之后，接着要做的就是编写中断服务函数。中断服务函数的名字是在 HAL 库中事先有定义的。这里需要说明一下，STM32F1 的 IO 口外部中断函数只有 7 个，分别为：

```
void EXTI0_IRQHandler();
void EXTI1_IRQHandler();
void EXTI2_IRQHandler();
void EXTI3_IRQHandler();
void EXTI4_IRQHandler();
void EXTI9_5_IRQHandler();
void 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。一般情况下，我们可以把中断控制逻辑直接编写在中断服务函数中，但是 HAL 库把中断处理过程进行了简单封装，请看下面步骤 5 讲解。

5） 编写中断处理回调函数 HAL_GPIO_EXTI_Callback

在使用 HAL 库的时候，我们也可以跟使用标准库一样，在中断服务函数中编写控制逻辑。

但是 HAL 库为了用户使用方便，它提供了一个中断通用入口函数 HAL_GPIO_EXTI_IRQHandler，在该函数内部直接调用回调函数 HAL_GPIO_EXTI_Callback。

我们可以看看 HAL_GPIO_EXTI_IRQHandler 函数定义：

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00u)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }
}
```

该函数实现的作用非常简单，就是清除中断标志位，然后调用回调函数 HAL_GPIO_EXTI_Callback()实现控制逻辑。所以我们编写中断控制逻辑将跟串口实验类似，在中断服务函数中直接调用外部中断共用处理函数 HAL_GPIO_EXTI_IRQHandler，然后在回调函数 HAL_GPIO_EXTI_Callback 中通过判断中断是来自哪个 IO 口编写相应的中断服务控制逻辑。

讲到这里，相信大家对 STM32 的 IO 口外部中断已经有了一定的了解。下面我们再总结一下配置 IO 口外部中断的一般步骤：

1）使能 IO 口时钟。

2）调用函数 HAL_GPIO_Init 设置 IO 口模式，触发条件，使能 SYSCFG 时钟以及设置 IO 口与中断线的映射关系。

3）配置中断优先级（NVIC），并使能中断。

4）在中断服务函数中调用外部中断共用入口函数 HAL_GPIO_EXTI_IRQHandler。

5）编写外部中断回调函数 HAL_GPIO_EXTI_Callback。

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，我们要实现同第七章差不多的功能，但是这里我们使用的是中断来检测按键，WK_UP 控制 DS0，按一次亮，再按一次灭；KEY1 控制 DS1，效果同 WK_UP；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

我们直接打开我们的光盘的实验 3 外部中断实验工程，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，并且包含了头文件 exti.h。extic.c 文件代码如下：

```
//外部中断初始化
void EXTI_Init(void)
{
    GPIO_InitTypeDef GPIO_Iniure;
    __HAL_RCC_GPIOA_CLK_ENABLE();                   //开启 GPIOA 时钟
    __HAL_RCC_GPIOE_CLK_ENABLE();                   //开启 GPIOE 时钟
    GPIO_Iniure.Pin=GPIO_PIN_0;                     //PA0
    GPIO_Iniure.Mode=GPIO_MODE_IT_RISING;           //上升沿触发
    GPIO_Iniure.Pull=GPIO_PULLDOWN;
```

```
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
    GPIO_Initure.Pin=GPIO_PIN_3|GPIO_PIN_4;          //PE3,4
    GPIO_Initure.Mode=GPIO_MODE_IT_FALLING;   //下降沿触发
    GPIO_Initure.Pull=GPIO_PULLUP;
    HAL_GPIO_Init(GPIOE,&GPIO_Initure);
    //中断线 0-PA0
    HAL_NVIC_SetPriority(EXTI0_IRQn,2,0);       //抢占优先级为 2，子优先级为 0
    HAL_NVIC_EnableIRQ(EXTI0_IRQn);             //使能中断线 0
    //中断线 3-PE3
    HAL_NVIC_SetPriority(EXTI3_IRQn,2,2);       //抢占优先级为 2，子优先级为 2
    HAL_NVIC_EnableIRQ(EXTI3_IRQn);             //使能中断线 2
    //中断线 4-PE4
    HAL_NVIC_SetPriority(EXTI4_IRQn,2,3);       //抢占优先级为 2，子优先级为 3
    HAL_NVIC_EnableIRQ(EXTI4_IRQn);             //使能中断线 4
}
//中断服务函数
void EXTI0_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);       //调用中断处理公用函数
}
void EXTI3_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_3);       //调用中断处理公用函数
}
void EXTI4_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_4);       //调用中断处理公用函数
}
//中断服务程序中需要做的事情
//在 HAL 库中所有的外部中断服务函数都会调用此函数
//GPIO_Pin:中断引脚号
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    delay_ms(100);          //消抖
    switch(GPIO_Pin)
    {
        case GPIO_PIN_0:
            if(WK_UP==1)
            {
                LED0=!LED0;                     //控制 LED0 反转
            }
            break;
        case GPIO_PIN_3:
```

```
            if(KEY1==0)                         //控制 LED1 反转
            {
                LED1=!LED1;
            }
            break;
        case GPIO_PIN_4:
            if(KEY0==0)
            {
                LED0=!LED0;
                LED1=!LED1;
            }
            break;      }
}
```

exti.c 文件总共包含 6 个函数。外部中断初始化函数 void EXTIX_Init 用来配置 IO 口外部中断相关步骤并使能中断，另一个函数 HAL_GPIO_EXTI_Callback 是外部中断共用回调函数，用来处理所有外部中断真正的控制逻辑。其他 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)，该函数内部主要做了两件事情。首先是调用 IO 口初始化函数 HAL_GPIO_Init 来初始化 IO 口，该函数的配置含义请看 10.1 小节中关于 HAL_GPIO_Init 函数讲解。其次是设置中断优先级并使能中断线。

接下来我们看看外部中断服务函数，一共 4 个。所有的中断服务函数内部都只调用了同样一个函数 HAL_GPIO_EXTI_IRQHandler，该函数是外部中断共用入口函数，函数内部会进行中断标志位清零， 并且调用中断处理共用回调函数 HAL_GPIO_EXTI_Callback。这在 10.1 小节我们也有详细的讲解。

最后是外部中断回调函数 HAL_GPIO_EXTI_Callback，该函数用来编写真正的外部中断控制逻辑。该函数有一个入口参数就是 IO 口序号。所以我们在该函数内部，一般通过判断 IO 口序号值来确定中断是来自哪个 IO 口，也就是哪个中断线，然后编写相应的控制逻辑。所以在该函数内部，我们通过 switch 语句判断 IO 口来源，例如是来自 GPIO_PIN_0，那么一定是来自 PA0，因为中断线一次只能连接一个 IO 口，而四个 IO 口中序号为 0 的 IO 口只有 PA0，所以中断线 0 一定是连接 PA0，也就是外部中断由 PA0 触发。

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)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
```

```
    KEY_Init();                          //初始化按键
    EXTI_Init();                         //初始化外部中断
    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精英 STM32 V1 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 和 KEY_UP 来观察 DS0、DS1 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 STM32F1 的独立看门狗（以下简称 IWDG）。STM32F1 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 STM32F1 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 STM32F1 独立看门狗简介

STM32 的独立看门狗由内部专门的 40Khz 低速时钟驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 40Khz，而是在 30~60Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 40Khz 的频率来计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

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，首先是键值寄存器 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
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEY[15:0]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	保留，始终读为0。
位15:0	**KEY[15:0]**: 键值(只写寄存器，读出值为0x0000) 软件必须以一定的间隔写入0xAAAA，否则，当计数器为0时，看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。 写入0xCCCC，启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。

图 11.1.1 IWDG_KR 寄存器各位描述

在键寄存器(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 0xFFF 递减计数。当计数器计数到末尾 0x000 时，会产生一个复位信号(IWDG_RESET)。 无论何时，只要键寄存器 IWDG_KR 中被写入 0xAAAA， IWDG_RLR 中的值就会被重新加载到计数器中从而避免产生看门狗复位 。

IWDG_PR 和 IWDG_RLR 寄存器具有写保护功能。要修改这两个寄存器的值，必须先向 IWDG_KR 寄存器中写入 0x5555。将其他值写入这个寄存器将会打乱操作顺序，寄存器将重新被保护。重装载操作(即写入 0xAAAA)也会启动写保护功能。

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：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留													PR[2:0]		
													rw	rw	rw

位31:3	保留，始终读为0。
位2:0	**PR[2:0]**: 预分频因子 这些位具有写保护设置，上面已有介绍。 通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子，IWDG_SR寄存器的PVU位必须为0。 000: 预分频因子=4　　100: 预分频因子=64 001: 预分频因子=8　　101: 预分频因子=128 010: 预分频因子=16　　110: 预分频因子=256 011: 预分频因子=32　　111: 预分频因子=256 注意：对此寄存器进行读操作，将从VDD电压域返回预分频值。如果写操作正在进行，则读回的值可能是无效的。因此，只有当IWDG_SR寄存器的PVU位为0时，读出的值才有效。

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。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：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留				RL[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:12	保留，始终读为0。
位11:0	**RL[11:0]**: 看门狗计数器重装载值 这些位具有写保护功能，前面已有介绍。 用于定义看门狗计数器的重装载值，每当向IWDG_KR寄存器写入0xAAAA时，重装载值会被传送到计数器中。随后计数器从这个值开始递减计数。看门狗超时周期可通过此重装载值和时钟预分频值来计算。 只有当IWDG_SR寄存器中的RVU位为0时，才能对此寄存器进行修改。 注：对此寄存器进行读操作，将从VDD电压域返回预分频值。如果写操作正在进行，则读回的值可能是无效的。因此，只有当IWDG_SR寄存器的RVU位为0时，读出的值才有效。

图 11.1.3 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 STM32F1 的独立看门狗。独立看门狗相关的库函数操作函数在文件 stm32f1xx_hal_iwdg.c 和对应的头文件 stm32f1xx_hal_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）

首先我们必须取消 IWDG_PR 和 IWDG_RLR 寄存器的写保护，这样才可以设置寄存器 IWDG_PR 和 IWDG_RLR 的值。取消写保护和设置预分频系数以及重装载值在 HAL 库中是通过函数 HAL_IWDG_Init 实现的。该函数声明为：

```
HAL_StatusTypeDef HAL_IWDG_Init(IWDG_HandleTypeDef *hiwdg);
```

该函数只有一个入口参数 hiwdg，该参数是 IWDG_HandleTypeDef 结构体指针类型。接下来我们看看结构体 IWDG_HandleTypeDef 定义：

```
typedef struct
{
  IWDG_TypeDef                 *Instance;
  IWDG_InitTypeDef              Init;
}IWDG_HandleTypeDef;
```

成员变量 Instance 用来设置看门狗寄存器基地址，实际上在 HAL 库中已经通过标识符定义了，这里对于独立看门狗直接设置为标识符 IWDG 即可。

成员变量 Init 是一个 IWDG_InitTypeDef 结构体类型，该结构体只有 2 个成员变量，分别用来设置独立看门狗的预分频系数和重装载值，定义如下：

```
typedef struct
{
  uint32_t Prescaler;
  uint32_t Reload;
}IWDG_InitTypeDef;
```

成员变量 Lock 是一个锁存变量，该变量在 HAL 库中当操作配置 IWDG 之前设置为锁住 LOCK，当配置操作完成之后设置为 UNLOCK，实际上是一个操作状态标识符。

成员变量 State 也是 HAL 定义的一个过程标识符，用来记录 IWDG 处理状态。

HAL_IWDG_Init 函数使用的一般方法为：

```
IWDG_HandleTypeDef IWDG_Handler;     //独立看门狗句柄
IWDG_Handler.Instance=IWDG;          //独立看门狗
IWDG_Handler.Init.Prescaler=IWDG_PRESCALER_64;  //设置 IWDG 分频系数
IWDG_Handler.Init.Reload=500;  //重装载值
HAL_IWDG_Init(&IWDG_Handler);
```

上面程序的作用是初始化 IWDG，设置分频系数为 64，重装载值为 500。设置完预分频系数和重装载值后，我们就可以知道看门狗的喂狗时间（也就是看门狗溢出时间），该时间的计算方式为：

$$Tout=((4\times 2^{prer})\times rlr)/32$$

其中 Tout 为看门狗溢出时间（单位为 ms）；prer 为看门狗时钟预分频值（IWDG_PR 值），范围为 0~7；rlr 为看门狗的重装载值（IWDG_RLR 的值）；

比如我们设定 prer 值为 4（4 代表的是 64 分频，HAL 库中可以使用宏定义标识符 IWDG_PRESCALER_64），rlr 值为 500，那么就可以得到 Tout=64×500/32=1000ms，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 32Khz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

2）重载计数值喂狗（向 IWDG_KR 写入 0XAAAA）

在 HAL 中重载计数值的函数是 HAL_IWDG_Refresh，该函数声明为：

```
HAL_StatusTypeDef HAL_IWDG_Refresh(IWDG_HandleTypeDef *hiwdg);
```

该函数有一个入口参数为前面讲解的 IWDG_HandleTypeDef 结构体类型指针，它的作用是把值 0xAAAA 写入到 IWDG_KR 寄存器，从而触发计数器重载，即实现独立看门狗的喂狗操作。

3) 启动看门狗(向 IWDG_KR 写入 0XCCCC)

HAL 库函数里面启动独立看门狗的函数是__HAL_IWDG_START：

```
__HAL_IWDG_START(hiwdg);
```

通过上面 3 个步骤，我们就可以启动 STM32F1 的独立看门狗了，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 STM32F1 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 STM32F1 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 HAL 库看门狗支持文件 stm32f1xx_hal_iwdg.h 和 stm32f1xx_hal_iwdg.c 文件。

在 wdg.c 里面输入如下代码：

```
#include "wdg.h"
IWDG_HandleTypeDef IWDG_Handler; //独立看门狗句柄

//初始化独立看门狗
//prer:分频数:IWDG_PRESCALER_4~IWDG_PRESCALER_256
//rlr:自动重装载值,0~0XFFF.
//时间计算(大概):Tout=((4*2^prer)*rlr)/32 (ms).
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG_Handler.Instance=IWDG;
    IWDG_Handler.Init.Prescaler=prer;     //设置 IWDG 分频系数
    IWDG_Handler.Init.Reload=rlr;         //重装载值
    HAL_IWDG_Init(&IWDG_Handler);         //初始化 IWDG,默认会开启独立看门狗
}

//喂独立看门狗
void IWDG_Feed(void)
{
    HAL_IWDG_Refresh(&IWDG_Handler);   //喂狗
}
```

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，分别用来设置与预分频数与重装

寄存器的值的。通过这两个参数，就可以大概知道看门狗复位的时间周期为多少了。其计算方式上面有详细的介绍，这里不再多说了。

void IWDG_Feed(void)函数，该函数用来喂狗，因为 STM32 的喂狗只需要向关键字寄存器写入 0XAAAA 即可,也就是调用库函数 HAL_IWDG_Refresh，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)
{
    HAL_Init();                                 //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                             //初始化延时函数
    uart_init(115200);                          //初始化串口
    LED_Init();                                 //初始化 LED
    KEY_Init();                                 //初始化按键
    delay_ms(100);                             //延时 100ms 再初始化看门狗,LED0 的变化"可见"
    IWDG_Init(IWDG_PRESCALER_64,500);//分频数为 64,重载值为 500,溢出时间为 1s
    LED0=0;
    while(1)
    {
        if(KEY_Scan(0)==WKUP_PRES)      //如果 WK_UP 按下，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同的方式处理），因为以后我们包含的头文件会越来越多，大家想看，可以直接打开光盘相关源码查看。至此，独立看门狗的实验代码，我们就全部编写完了，接着要做的就是下载验证了，看看我们的代码是否真的正确。

11.4 下载验证

在编译成功之后，我们就可以下载代码到精英 STM32 V1 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 STM32F1 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 STM32F1 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 STM32F1 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的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：

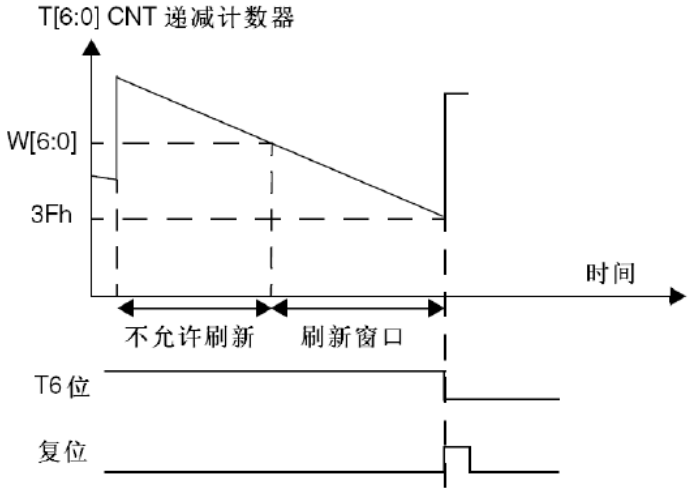


图 12.1.1 窗口看门狗工作示意图

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：

Twwdg=(4096×2^WDGTB×(T[5:0]+1)) /Fpclk1;

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 Fpclk1=36Mhz，那么可以得到最小-最大超时时间表如表 12.1.1 所示：

WDGTB	最小超时值	最大超时值
0	113μs	7.28ms
1	227μs	14.56ms
2	455μs	29.12ms
3	910μs	58.25ms

表 12.1.1 36M 时钟下窗口看门狗的最小最大超时表

接下来，我们介绍窗口看门狗的 3 个寄存器。首先介绍控制寄存器（WWDG_CR），该寄存器的各位描述如图 12.1.2 所示：

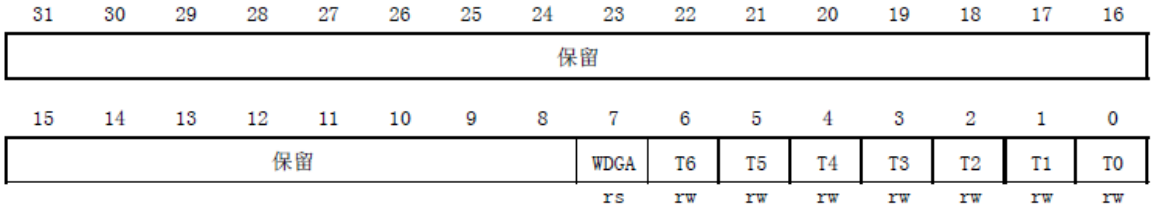
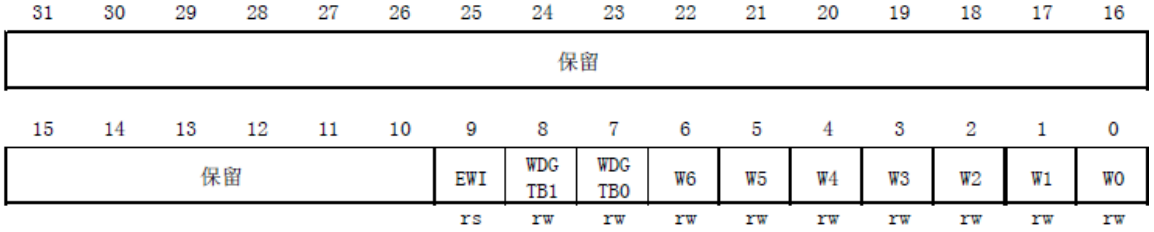


图 12.1.2 WWDG_CR 寄存器各位描述

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^ 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31:8	保留。
位9	**EWI**: 提前唤醒中断 此位若置1，则当计数器值达到40h，即产生中断。 此中断只能由硬件在复位后清除。
位8:7	**WDGTB[1:0]**: 时基 预分频器的时基可根据如下修改： 00: CK计时器时钟(PCLK1除以4096)除以1 01: CK计时器时钟(PCLK1除以4096)除以2 10: CK计时器时钟(PCLK1除以4096)除以4 11: CK计时器时钟(PCLK1除以4096)除以8
位6:0	**W[6:0]**: 7位窗口值 这些位包含了用来与递减计数器进行比较用的窗口值。

图 12.1.3 WWDG_ CFR 寄存器各位描述

该位中的 EWI 是提前唤醒中断，也就是在快要产生复位的前一段时间（T[6:0]=0X40）来提醒我们，需要进行喂狗了，否则将复位！因此，我们一般用该位来设置中断，当窗口看门狗的计数器值减到 0X40 的时候，如果该位设置，并开启了中断，则会产生中断，我们可以在中断里面向 WWDG_CR 重新写入计数器的值，来达到喂狗的目的。注意这里在进入中断后，必须在不大于 1 个窗口看门狗计数周期的时间(在 PCLK1 频率为 36M 且 WDGTB 为 0 的条件下，该时间为 113us）内重新写 WWDG_CR，否则，看门狗将产生复位！

最后我们要介绍的是状态寄存器（WWDG_SR），该寄存器用来记录当前是否有提前唤醒的标志。该寄存器仅有位 0 有效，其他都是保留位。当计数器值达到 40h 时，此位由硬件置 1。它必须通过软件写 0 来清除。对此位写 1 无效。即使中断未被使能，在计数器的值达到 0X40 的时候，此位也会被置 1。

在介绍完了窗口看门狗的寄存器之后，我们介绍要如何启用 STM32F1 的窗口看门狗。这里我们介绍 HAL 中用中断的方式来喂狗的方法，窗口看门狗 HAL 库相关源码和定义分布在文

件 stm32f1xx_hal_wwdg.c 文件和头文件 stm32f1xx_hal_wwdg.h 中。步骤如下：

1）使能 WWDG 时钟

WWDG 不同于 IWDG，IWDG 有自己独立的 40Khz 时钟，不存在使能问题。而 WWDG 使用的是 PCLK1 的时钟，需要先使能时钟。方法是:

```
__HAL_RCC_WWDG_CLK_ENABLE();      //使能窗口看门狗时钟
```

2）设置窗口值,分频数和计数器初始值

在 HAL 库中，这三个值都是通过函数 HAL_WWDG_Init 来设置的。该函数声明如下：

```
HAL_StatusTypeDef HAL_WWDG_Init(WWDG_HandleTypeDef *hwwdg);
```

该函数只有一个入口参数，就是 WWDG_HandleTypeDef 结构体类型指针变量。这里我们来看看 WWDG_HandleTypeDef 结构体定义：

```
typedef struct
{
  WWDG_TypeDef                 *Instance;
  WWDG_InitTypeDef             Init;
}WWDG_HandleTypeDef;
```

该结构体和前面我们讲解的 WWDG_HandleTypeDef 类似，这里我们就主要讲解成员变量 Init，它是 WWDG_InitTypeDef 结构体类型，该结构体定义如下：

```
typedef struct
{
  uint32_t Prescaler;     //预分频系数
  uint32_t Window;        //窗口值
  uint32_t Counter;       //计数器值
  uint32_t EWIMode ; //是否启用 WWDG 早期唤醒中断
}WWDG_InitTypeDef;
```

该结构体有 3 三个成员变量，分别用来设置 WWDG 的预分频系数，窗口之以及计数器值。函数 HAL_WWDG_Init 的使用范例如下：

```
WWDG_HandleTypeDef WWDG_Handler;        //窗口看门狗句柄

WWDG_Handler.Instance=WWDG;   //窗口看门狗
WWDG_Handler.Init.Prescaler=WWDG_PRESCALER_8;//设置分频系数为 8
WWDG_Handler.Init.Window=0X5F;          //设置窗口值 0X5F
WWDG_Handler.Init.Counter=0x7F;         //设置计数器值 0x7F
HAL_WWDG_Init(&WWDG_Handler);     //初始化 WWDG
```

3）使能中断通道并配置优先级（如果开启了 WWDG 中断）

这一步相信大家已经非常熟悉了，我们这里仅仅列出两行实现代码，如下：

```
HAL_NVIC_SetPriority(WWDG_IRQn,2,3);     //抢占优先级 2，子优先级为 3
HAL_NVIC_EnableIRQ(WWDG_IRQn);           //使能窗口看门狗中断
```

这里大家要注意，跟串口一样，HAL 库同样为看门狗提供了 MSP 回调函数 HAL_WWDG_MspInit，一般情况下，步骤 1 和步骤 4 的步骤，是与 MCU 相关的，我们均放在该回调函数中。关于 MSP 回调函数的使用方法，前面多次讲解，这里我们就不累赘了。

4）编写中断服务函数

在最后，还是要编写窗口看门狗的中断服务函数，通过该函数来喂狗，喂狗要快，否则当窗口看门狗计数器值减到 0X3F 的时候，就会引起软复位了。在中断服务函数里面也要将状态

寄存器的 EWIF 位清空。

窗口看门狗中断服务函数为：

```
void WWDG_IRQHandler(void);
```

在 HAL 库中，喂狗函数为：

```
HAL_StatusTypeDef HAL_WWDG_Refresh(WWDG_HandleTypeDef *hwwdg);
```

WWDG 的喂狗操作实际就是往 CR 寄存器重写计数器值。

5)重写窗口看门狗唤醒中断处理回调函数 HAL_WWDG_WakeupCallback

跟串口和外部中断一样，首先，HAL 库定义了一个中断处理共用函数 HAL_WWDG_IRQHandler，我们在 WWDG 中断服务函数中会调用该函数。同时该函数内部，会经过一系列判断，最后调用回调函数 HAL_WWDG_WakeupCallback，所以提前唤醒中断逻辑我们一般些在回调函数 HAL_WWDG_WakeupCallback 中。 回调函数声明为：

```
void HAL_WWDG_EarlyWakeupCallback (WWDG_HandleTypeDef* hwwdg);
```

完成了以上 5 个步骤之后，我们就可以使用 STM32F1 的窗口看门狗了。这一章的实验，我们将通过 DS0 来指示 STM32F1 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 STM32F1 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 STM32F1 的复位情况和窗口看门狗的喂狗情况。

12.3 软件设计

这里，我们在之前的 IWDG 看门狗实例内增添部分代码来实现这个实验，由于我们没有用到按键，所以去掉 HARDWARE 组里面的 key.c 文件（注意，此时 HARDWARE 组仅剩：led.c 和 wdg.c）。首先打开上次的工程，然后在 wdg.c 加入如下代码（之前代码保留）：

```
WWDG_HandleTypeDef WWDG_Handler;        //窗口看门狗句柄
//保存 WWDG 计数器的设置值，默认为最大
//u8 WWDG_CNT=0X7F;
//初始化窗口看门狗
//tr     :T[6:0],计数器值
//wr     :W[6:0],窗口值
//fprer:分频系数（WDGTB）,仅最低 2 位有效
//Fwwdg=PCLK1/(4096*2^fprer). 一般 PCLK1=42Mhz
void WWDG_Init(u8 tr,u8 wr,u32 fprer)
{
    WWDG_Handler.Instance=WWDG;
    WWDG_Handler.Init.Prescaler=fprer;          //设置分频系数
    WWDG_Handler.Init.Window=wr;                //设置窗口值
    WWDG_Handler.Init.Counter=tr;               //设置计数器值
    WWDG_Handler.Init.EWIMode=WWDG_EWI_ENABLE;
                                                //使能窗口看门狗提前唤醒中断
```

```
    HAL_WWDG_Init(&WWDG_Handler);           //初始化 WWDG
}

//WWDG 底层驱动，时钟配置，中断配置
//此函数会被 HAL_WWDG_Init()调用
//hwwdg:窗口看门狗句柄
void HAL_WWDG_MspInit(WWDG_HandleTypeDef *hwwdg)
{
    __HAL_RCC_WWDG_CLK_ENABLE();            //使能窗口看门狗时钟

    HAL_NVIC_SetPriority(WWDG_IRQn,2,3);    //抢占优先级 2，子优先级为 3
    HAL_NVIC_EnableIRQ(WWDG_IRQn);          //使能窗口看门狗中断
}

//窗口看门狗中断服务函数
void WWDG_IRQHandler(void)
{
    HAL_WWDG_IRQHandler(&WWDG_Handler);//调用 WWDG 共用中断处理函数
}

//中断服务函数处理过程
//此函数会被 HAL_WWDG_IRQHandler()调用
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg)
{
    HAL_WWDG_Refresh(&WWDG_Handler);//更新窗口看门狗值
    LED1=!LED1;
}
```

wwdg.c 文件一共包含四个函数。第一个函数 WWDG_Init()实现的是前面讲解的步骤 1 和步骤 3，主要作用是调用函数 HAL_WWDG_Init 设置 WWDG 的分频系数，窗口值和计数器初始值，同时还调用 HAL_WWDG_Start_IT 函数开启看门狗和使能看门狗中断。包括看门狗计数器的值和看门狗比较值等。第二个函数 HAL_WWDG_MspInit 是 WWDG 的 MSP 回调函数，该函数主要作用是使能 WWDG 时钟，以及设置 NVIC，实现的是前面讲解的步骤 2 和 4。第三个函数 WWDG_IRQHandler 也就是中断服务函数，该函数在前面步骤 5 有讲解，一般情况下，在该函数内部会调用中断共用处理函数 HAL_WWDG_IRQHandler。第四个函数 HAL_WWDG_WakeupCallback 是提前唤醒中断回调函数，该函数内部我们主要编写了喂狗操作，以及 LED1 翻转。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 HAL_WWDG_Refresh 函数把该数值放回到 WWDG_CR 上。

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)
{
    HAL_Init();                             //初始化 HAL 库
```

```
    …//此处省略部分初始化代码
    LED0=0;                                 //点亮 LED0
    delay_ms(300);                          //延时 300ms 再初始化看门狗,LED0 的变化"可见"
    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战舰 STM32F1 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

将代码下载到精英 STM32 开发板后，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 8 次左右，和我们预期的一致，说明我们的实验是成功的。

第十三章 定时器中断实验

这一章，我们将向大家介绍如何使用 STM32 的通用定时器，STM32 的定时器功能十分强大，有 TIME1 和 TIME8 等高级定时器，也有 TIME2~TIME5 等通用定时器，还有 TIME6 和 TIME7 等基本定时器。在《STM32 中文参考手册》里面，定时器的介绍占了 1/5 的篇幅，足见其重要性。在本章中，我们将使用 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 STM32F1 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 STM32F1 通用定时器简介

STM32 的通用定时器是一个通过可编程预分频器（PSC）驱动的 16 位自动装载计数器（CNT）构成。STM32 的通用定时器可以被用于：测量输入信号的脉冲长度(输入捕获)或者产生输出波形(输出比较和 PWM)等。 使用定时器预分频器和 RCC 时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。STM32 的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM3 的通用 TIMx (TIM2、TIM3、TIM4 和 TIM5)定时器功能包括：

1)16 位向上、向下、向上/向下自动装载计数器（TIMx_CNT）。

2)16 位可编程(可以实时修改)预分频器(TIMx_PSC)，计数器时钟频率的分频系数为 1～65535 之间的任意数值。

3）4 个独立通道（TIMx_CH1~4），这些通道可以用来作为：

A．输入捕获

B．输出比较

C．PWM 生成(边缘或中间对齐模式)

D．单脉冲模式输出

4）可使用外部信号（TIMx_ETR）控制定时器和定时器互连（可以用 1 个定时器控制另外一个定时器）的同步电路。

5）如下事件发生时产生中断/DMA：

A．更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)

B．触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)

C．输入捕获

D．输出比较

E．支持针对定位的增量(正交)编码器和霍尔传感器电路

F．触发输入作为外部时钟或者按周期的电流管理

由于 STM32 通用定时器比较复杂，这里我们不再多介绍，请大家直接参考《STM32 中文参考手册》第 253 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。

首先是控制寄存器 1（TIMx_CR1），该寄存器的各位描述如图 13.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位0	**CEN**：使能计数器 0：禁止计数器； 1：使能计数器。 注：在软件设置了CEN位后，外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 在单脉冲模式下，当发生更新事件时，CEN被自动清除。

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
保留	TDE	保留	CC4DE	CC3DE	CC2DE	CC1DE	UDE	保留	TIE	保留	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

位0	**UIE**：允许更新中断 (Update interrupt enable) 0：禁止更新中断； 1：允许更新中断。

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]**：预分频器的值 计数器的时钟频率CK_CNT等于f_{CK_PSC}/(PSC[15:0]+1)。 PSC包含了当更新事件产生时装入当前预分频器寄存器的值。

图 13.1.3 TIMx_ PSC 寄存器各位描述

这里，定时器的时钟来源有 4 个：

1）内部时钟（CK_INT）

2）外部时钟模式 1：外部输入脚（TIx）

3）外部时钟模式 2：外部触发输入（ETR）

4）内部触发输入（ITRx）：使用 A 定时器作为 B 定时器的预分频器（A 为 B 提供时钟）。

这些时钟，具体选择哪个可以通过 TIMx_SMCR 寄存器的相关位来设置。这里的 CK_INT 时钟是从 APB1 倍频的来的，除非 APB1 的时钟分频数设置为 1，否则通用定时器 TIMx 的时钟是 APB1 时钟的 2 倍，当 APB1 的时钟不分频的时候，通用定时器 TIMx 的时钟就等于 APB1 的时钟。这里还要注意的就是高级定时器的时钟不是来自 APB1，而是来自 APB2 的。

这里顺带介绍一下 TIMx_CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前定时器的计数值。

接着我们介绍自动重装载寄存器（TIMx_ARR），该寄存器在物理上实际对应着 2 个寄存器。一个是程序员可以直接操作的，另外一个是程序员看不到的，这个看不到的寄存器在《STM32 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 2 者是连通的；而 APRE=1 时，在每一次更新事件（UEV）时，才把预装在寄存器的内容传送到影子寄存器。

自动重装载寄存器的各位描述如图 13.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**ARR[15:0]**: 自动重装载的值 ARR包含了将要装载入实际的自动重装载寄存器的数值。 当自动重装载的值为空时，计数器不工作。

图 13.1.4 TIMx_ ARR 寄存器各位描述

最后，我们要介绍的寄存器是：状态寄存器（TIMx_SR）。该寄存器用来标记当前与定时器相关的各种事件/中断是否发生。该寄存器的各位描述如图 13.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			CC4OF	CC3OF	CC2OF	CC1OF	保留		TIF	保留	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

位0	**UIF**：更新中断标记 (Update interrupt flag) 当产生更新事件时该位由硬件置'1'。它由软件清'0'。 0：无更新事件产生； 1：更新中断等待响应。当寄存器被更新时该位由硬件置'1'： – 若TIMx_CR1寄存器的UDIS=0、URS=0，当TIMx_EGR寄存器的UG=1时产生更新事件(软件对计数器CNT重新初始化)； – 若TIMx_CR1寄存器的UDIS=0、URS=0，当计数器CNT被触发事件重初始化时产生更新事件。(参考同步控制寄存器的说明)

图 13.1.5 TIMx_ SR 寄存器各位描述

TIMx_ SR 寄存器，我们同样只用到了最低位，当计数器 CNT 被重新初始化的时候，产生更新中断标记，通过这个中断标志位，就可以知道产生中断的类型。关于这些位的详细描述，请参考《STM32 中文参考手册》第 282 页。

只要对以上几个寄存器进行简单的设置，我们就可以使用通用定时器了，并且可以产生中断。

这一章，我们将使用定时器产生中断，然后在中断服务函数里面翻转 DS1 上的电平，来指示定时器中断的产生。接下来我们以通用定时器 TIM3 为实例，来说明要经过哪些步骤，才能达到这个要求，并产生中断。这里我们就对每个步骤通过库函数的实现方式来描述。首先要提到的是，定时器相关的库函数主要集中在 HAL 库文件 stm32f1xx_hal_tim.h 和 stm32f1xx_hal_tim.c 文件中。定时器配置步骤如下：

1）TIM3 时钟使能。

HAL 中定时器使能是通过宏定义标识符来实现对相关寄存器操作的，方法如下：

```
__HAL_RCC_TIM3_CLK_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 HAL 库中，定时器的初始化参数是通过定时器初始化函数 HAL_TIM_Base_Init 实现的：

```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim);
```

该函数只有一个入口参数，就是 TIM_HandleTypeDef 类型结构体指针，结构体类型为下面我们看看这个结构体的定义：

```
typedef struct
{
  TIM_TypeDef              *Instance;
  TIM_Base_InitTypeDef     Init;
  HAL_TIM_ActiveChannel    Channel;
  DMA_HandleTypeDef        *hdma[7];
  HAL_LockTypeDef          Lock;
  __IO HAL_TIM_StateTypeDef State;
}TIM_HandleTypeDef;
```

第一个参数 Instance 是寄存器基地址。和串口，看门狗等外设一样，一般外设的初始化结构体定义的第一个成员变量都是寄存器基地址。这在 HAL 中都定义好了，比如要初始化串口 1，那么 Instance 的值设置为 TIM1 即可。

第二个参数 Init 为真正的初始化结构体 TIM_Base_InitTypeDef 类型。该结构体定义如下：

```
typedef struct
{
  uint32_t Prescaler;          //预分频系数
  uint32_t CounterMode;        //计数方式
  uint32_t Period;             //自动装载值 ARR
  uint32_t ClockDivision;      //时钟分频因子
  uint32_t RepetitionCounter;
} TIM_Base_InitTypeDef;
```

该初始化结构体中，参数 Prescaler 是用来设置分频系数的，刚才上面有讲解。参数 CounterMode 是用来设置计数方式，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。参数 Period 是设置自动重载计数周期值。参数 ClockDivision 是用来设置时钟分频因子，也就是定时器时钟频率 CK_INT 与数字滤波器所使用的采样时钟之间的分频比。参数 RepetitionCounter 用来设置重复计数器寄存器的值，用在高级定时器中。

第三个参数 Channel 用来设置活跃通道。前面我们讲解过，每个定时器最多有四个通道可以用来做输出比较，输入捕获等功能之用。这里的 Channel 就是用来设置活跃通道的，取值范围为：HAL_TIM_ACTIVE_CHANNEL_1~ HAL_TIM_ACTIVE_CHANNEL_4。

第四个 hdma 是定时器的 DMA 功能时用到，为了简单起见，我们暂时不讲解太复杂。

第五个参数 Lock 和 State，是状态过程标识符，是 HAL 库用来记录和标志定时器处理过程。

定时器初始化范例如下：

```
TIM_HandleTypeDef TIM3_Handler;                              //定时器句柄
TIM3_Handler.Instance=TIM3;                                  //通用定时器 3
TIM3_Handler.Init.Prescaler= 7199;                           //分频系数
TIM3_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;      //向上计数器
TIM3_Handler.Init.Period=4999;                               //自动装载值
TIM3_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;//时钟分频因子
HAL_TIM_Base_Init(&TIM3_Handler);
```

3）使能定时器更新中断，使能定时器

HAL 库中，使能定时器更新中断和使能定时器两个操作可以在函数 HAL_TIM_Base_Start_IT()中一次完成的，该函数声明如下：

```
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim);
```

该函数非常好理解，只有一个入口参数。调用该定时器之后，会首先调用__HAL_TIM_ENABLE_IT 宏定义使能更新中断，然后调用宏定义__HAL_TIM_ENABLE 使能相应的定时器。这里我们分别列出单独使能/关闭定时器中断和使能/关闭定时器方法：

```
__HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE);//使能句柄指定的定时器更新中断
__HAL_TIM_DISABLE_IT (htim, TIM_IT_UPDATE);//关闭句柄指定的定时器更新中断
__HAL_TIM_ENABLE(htim);//使能句柄 htim 指定的定时器
__HAL_TIM_DISABLE(htim);//关闭句柄 htim 指定的定时器
```

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。之前多次讲解到中断优先级的设置，这里就不重复讲解。

和串口等其他外设一样，HAL 库为定时器初始化定义了回调函数 HAL_TIM_Base_MspInit。

一般情况下，与 MCU 有关的时钟使能，以及中断优先级配置我们都会放在该回调函数内部。函数声明如下：

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim);
```

对于回调函数，这里我们就不做过多讲解，大家只需要重写这个函数即可。

5）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。通常情况下，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。

跟串口一样，对于定时器中断，HAL 库同样为我们封装了处理过程。这里我们以定时器 3 的更新中断为例来讲解。

首先，中断服务函数是不变的，定时器 3 的中断服务函数为：

```
TIM3_IRQHandler();
```

一般情况下我们是在中断服务函数内部编写中断控制逻辑。但是 HAL 库为我们定义了 新的定时器中断共用处理函数 HAL_TIM_IRQHandler，在每个定时器的中断服务函数内部，我们会调用该函数。该函数声明如下：

```
void HAL_TIM_IRQHandler(TIM_HandleTypeDef *htim);
```

而函数 HAL_TIM_IRQHandler 内部，会对相应的中断标志位进行详细判断，判断确定中断来源后，会自动清掉该中断标志位，同时调用不同类型中断的回调函数。所以我们的中断控制逻辑只用编写在中断回调函数中，并且中断回调函数中不需要清中断标志位。

比如定时器更新中断回调函数为：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);
```

跟串口中断回调函数一样，我们只需要重写该函数即可。对于其他类型中断，HAL 库同样提供了几个不同的回调函数，这里我们列出常用的几个回调函数：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);//更新中断
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim);//输出比较
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim);//输入捕获
void HAL_TIM_TriggerCallback(TIM_HandleTypeDef *htim);//触发中断
```

对于这些回调函数的使用方法我们在后面用到的时候会给大家详细讲解。

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，DS0 和 DS1 的电路在前面已经有介绍了。而 TIM3 属于 STM32 的内部资源，只需要软件设置即可正常工作。

13.3 软件设计

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 HAL 库函数文件 stm32f1xx_hal_tim.c 和头文件 stm32f1xx_hal_tim.h。下面我们来看看我

们的 time.c 文件。timer.c 文件代码如下：

```
#include "timer.h"
#include "led.h"
TIM_HandleTypeDef TIM3_Handler;         //定时器句柄

//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_Init(u16 arr,u16 psc)
{
    TIM3_Handler.Instance=TIM3;                                    //通用定时器 3
    TIM3_Handler.Init.Prescaler=psc;                               //分频系数
    TIM3_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;      //向上计数器
    TIM3_Handler.Init.Period=arr;                                  //自动装载值
    TIM3_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;//时钟分频因子
    HAL_TIM_Base_Init(&TIM3_Handler);
    HAL_TIM_Base_Start_IT(&TIM3_Handler);
                                //使能定时器 3 和定时器 3 更新中断：TIM_IT_UPDATE
}

//定时器底册驱动，开启时钟，设置中断优先级
//此函数会被 HAL_TIM_Base_Init()函数调用
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if(htim->Instance==TIM3)
    {
        __HAL_RCC_TIM3_CLK_ENABLE();            //使能 TIM3 时钟
        HAL_NVIC_SetPriority(TIM3_IRQn,1,3);
                                    //设置中断优先级，抢占优先级 1，子优先级 3
        HAL_NVIC_EnableIRQ(TIM3_IRQn);          //开启 ITM3 中断
    }
}

//定时器 3 中断服务函数
void TIM3_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM3_Handler);
}

//回调函数，定时器中断服务函数调用
```

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&TIM3_Handler))
    {
        LED1=!LED1;             //LED1 反转
    }
}
```

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）的取反。

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。该函数的 2 个参数用来设置 TIM3 的溢出时间。因为我们在 Stm32_Clock_Init 函数里面已经初始化 APB1 的时钟为 2 分频，所以 APB1 的时钟为 36M，而从 STM32F1 的内部时钟树图（图 5.2.2.1）得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 的时钟频率将为 APB1 时钟的两倍。因此，TIM3 的时钟为 72M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

```
Tout= ((arr+1)*(psc+1))/Tclk；
```

其中：

Tclk：TIM3 的输入时钟频率（单位为 Mhz）。

Tout：TIM3 溢出时间（单位为 us）。

我们将 timer.c 文件保存，然后加入到 HARDWARE 组下。接下来，在 timer.h 文件里，我们输入如下代码：

```
#ifndef __TIMER_H
#define __TIMER_H
#include "sys.h"
extern TIM_HandleTypeDef TIM3_Handler;          //定时器句柄
void TIM3_Int_Init(u16 arr,u16 psc);
#endif
```

此部分代码十分简单，这里不做介绍。

最后，我们在主程序里面输入如下代码：

```
int main(void)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    TIM3_Init(5000-1,7200-1);               //定时器 3 初始化，定时器时钟为 72M，
    //分频系数为 7200-1，所以定时器 3 的频率为 72M/7200=10K，自动重装载为 5000-1，
    //那么定时器周期就是 500ms

    while(1)
    {
```

```
            LED0=!LED0;
            delay_ms(200);
        }
    }
```

这里的代码和之前大同小异，此段代码对 TIM3 进行初始化之后，进入死循环等待 TIM3 溢出中断，当 TIM3_CNT 的值等于 TIM3_ARR 的值的时候，就会产生 TIM3 的更新中断，然后在中断里面取反 LED1，TIM3_CNT 再从 0 开始计数。

这里定时器定时时长 500ms 是这样计算出来的，定时器的时钟为 72Mhz，分频系数为 7200，所以分频后的计数频率为 72Mhz/7200=10KHz,然后计数到 5000，所以时长为 5000/10000=0.5s，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精英 STM32 V1 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停闪烁（每 400ms 闪烁一次），而 DS1 也是不停的闪烁，但是闪烁时间较 DS0 慢（1s 一次）。

第十四章 PWM 输出实验

上一章，我们介绍了 STM32 的通用定时器 TIM3，用该定时器的中断来控制 DS1 的闪烁，这一章，我们将向大家介绍如何使用 STM32 的 TIM3 来产生 PWM 输出。在本章中，我们将使用 TIM3 的通道 2，并把通道 2 重映射到 PB5，产生 PWM 来控制 DS0 的亮度。本章分为如下几个部分：

14.1 PWM 简介
14.2 硬件设计
14.3 软件设计
14.4 下载验证

14.1 PWM 简介

脉冲宽度调制(PWM)，是英文“Pulse Width Modulation”的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。简单一点，就是对脉冲宽度的控制，PWM 原理如图 14.1.1 所示：

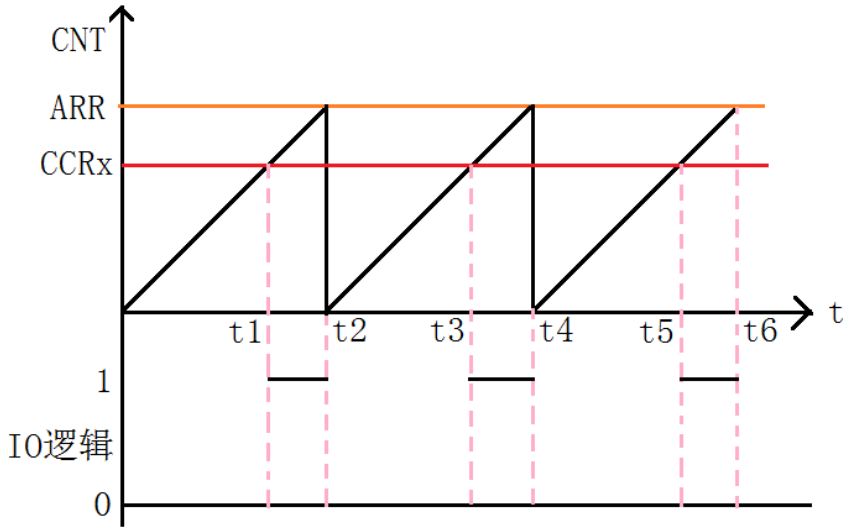


图 14.1.1 PWM 原理示意图

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>=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。

STM32 的定时器除了 TIM6 和 7。其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出，这样，STM32 最多可以同时产生 30 路 PWM 输出！这里我们仅使用 TIM3 的 CH2 产生一路 PWM 输出。如果要产生多路输出，大家可以根据我们的代码稍作修改即可。

要使 STM32 的通用定时器 TIMx 产生 PWM 输出，除了上一章介绍的寄存器外，我们还会用到 3 个寄存器，来控制 PWM 的。这三个寄存器分别是：捕获/比较模式寄存器（TIMx_CCMR1/2）、捕获/比较使能寄存器（TIMx_CCER）、捕获/比较寄存器（TIMx_CCR1~4）。接下来我们简单介绍一下这三个寄存器。

首先是捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器总共有 2 个，TIMx _CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 2，而 TIMx_CCMR2 控制 CH3 和 4。该寄存器的各位描述如图 14.1.2 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	rw

图 14.1.2 TIMx_CCMR1 寄存器各位描述

该寄存器的有些位在不同模式下，功能不一样，所以在图 14.1.2 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《STM32 中文参考手册》第 288 页，14.4.7 一节。这里我们需要说明的是模式设置位 OCxM，此部分由 3 位

组成。总共可以配置成 7 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110/111。这两种 PWM 模式的区别就是输出电平的极性相反。另外 CCxS 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。

接下来，我们介绍捕获/比较使能寄存器（TIMx_CCER），该寄存器控制着各个输入输出通道的开关。该寄存器的各位描述如图 14.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		CC4P	CC4E	保留		CC3P	CC3E	保留		CC2P	CC2E	保留		CC1P	CC1E
		rw	rw			rw	rw			rw	rw			rw	rw

图 14.1.3 TIMx_ CCER 寄存器各位描述

该寄存器比较简单，我们这里只用到了 CC2E 位，该位是输入/捕获 2 输出使能位，要想 PWM 从 IO 口输出，这个位必须设置为 1，所以我们需要设置该位为 1。该寄存器更详细的介绍了，请参考《STM32 中文参考手册》第 292 页，14.4.9 这一节。

最后，我们介绍一下捕获/比较寄存器（TIMx_CCR1~4），该寄存器总共有 4 个，对应 4 个输通道 CH1~4。因为这 4 个寄存器都差不多，我们仅以 TIMx_CCR1 为例介绍，该寄存器的各位描述如图 14.1.4 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的值 **若CC1通道配置为输出：** CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载特性，写入的数值会立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC1端口上产生输出信号。 **若CC1通道配置为输入：** CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。

图 14.1.4 寄存器 TIMx_ CCR1 各位描述

在输出模式下，该寄存器的值与 CNT 的值比较，根据比较结果产生相应动作。利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的输出脉宽了。本章，我们使用的是 TIM3 的通道 2，所以我们需要修改 TIM3_CCR2 以实现脉宽控制 DS0 的亮度。

我们要使用 TIM3 的 CH2 输出 PWM 来控制 DS0 的亮度，但是 TIM3_CH2 默认是接在 PA7 上面的，而我们的 DS0 接在 PB5 上面，如果普通 MCU，可能就只能用飞线把 PA7 飞到 PB5 上来实现了，不过，我们用的是 STM32，它比较高级，可以通过重映射功能，把 TIM3_CH2 映射到 PB5 上。

STM32 的重映射控制是由复用重映射和调试 IO 配置寄存器（AFIO_MAPR）控制的，该寄存器的各位描述如图 14.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留					SWJ_CFG[2:0]			保留			ADC2_ETRGREG_REMAP	ADC2_ETRGINJ_REMAP	ADC1_ETRGREG_REMAP	ADC1_ETRGINJ_REMAP	TIM5CH4_IREMAP
					w	w	w								

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PD01_REMAP	CAN_REMAP[1:0]		TIM4_REMAP	TIM3_REMAP[1:0]		TIM2_REMAP[1:0]		TIM1_REMAP[1:0]		USART3_REMAP[1:0]		USART2_REMAP	USART1_REMAP	I2C1_REMAP	SPI1_REMAP
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 14.1.5 寄存器 AFIO_MAPR 各位描述

我们这里用到的是 TIM3 的重映射，从上图可以看出，TIM3_REMAP 是由[11:10]这 2 个位控制的。TIM3_REMAP[1:0]重映射控制表如表 14.1.1 所示：

复用功能	TIM3_REMAP[1:0] = 00 (没有重映像)	TIM3_REMAP[1:0] = 10 (部分重映像)	TIM3_REMAP[1:0] = 11 (完全重映像)(1)
TIM3_CH1	PA6	PB4	PC6
TIM3_CH2	PA7	PB5	PC7
TIM3_CH3	PB0		PC8
TIM3_CH4	PB1		PC9

表 14.1.1 TIM3_REMAP 重映射控制表

默认条件下，TIM3_REMAP[1:0]为 00，是没有重映射的，所以 TIM3_CH1~TIM3_CH4 分别是接在 PA6、PA7、PB0 和 PB1 上的，而我们想让 TIM3_CH2 映射到 PB5 上，则需要设置 TIM3_REMAP[1:0]=10，即部分重映射，这里需要注意，此时 TIM3_CH1 也被映射到 PB4 上了。

至此，我们把本章要用的几个相关寄存器都介绍完了，本章要实现通过重映射 TIM3_CH2 到 PB5 上，由 TIM3_CH2 输出 PWM 来控制 DS0 的亮度。下面我们介绍配置步骤：

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 stm32f1xx_hal_tim.h 和 stm32f1xx_hal_tim.c 文件中。

1）开启 TIM3 和 GPIO 时钟，配置 PB5 选择复用功能输出。

要使用 TIM3，我们必须先开启 TIM3 的时钟，这点相信大家看了这么多代码，应该明白了。这里我们还要配置复用输出，才可以实现 TIM3_CH2 的 PWM 经过 PB5 输出。HAL 库使能 TIM3 时钟和 GPIO 时钟方法是：

```
__HAL_RCC_TIM3_CLK_ENABLE();                //使能 TIM3 时钟
__HAL_RCC_GPIOB_CLK_ENABLE ();              //开启 GPIOB 时钟
```

接下来便是要配置 PB5 复用映射为 TIM3 的 PWM 输出引脚。关于 IO 口复用映射，在串口通信实验中有详细讲解，主要是通过函数 HAL_GPIO_Init 来实现的：

```
GPIO_InitTypeDef GPIO_Initure;
__HAL_RCC_TIM3_CLK_ENABLE();            //使能定时器 3
__HAL_AFIO_REMAP_TIM3_PARTIAL();        //TIM3 通道引脚部分重映射使能
__HAL_RCC_GPIOB_CLK_ENABLE();           //开启 GPIOB 时钟
GPIO_Initure.Pin=GPIO_PIN_5;            //PB5
GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //复用推挽输出
GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
HAL_GPIO_Init(GPIOB,&GPIO_Initure);
```

2）初始化 TIM3,设置 TIM3 的 ARR 和 PSC 等参数。

根据前面的讲解，初始化定时器的 ARR 和 PSC 等参数是通过函数 HAL_TIM_Base_Init 来实现的，但是这里大家要注意，对于我们使用定时器的 PWM 输出功能时，HAL 库为我们提供了一个独立的定时器初始化函数 HAL_TIM_PWM_Init，该函数声明为：

```
HAL_StatusTypeDef HAL_TIM_PWM_Init(TIM_HandleTypeDef *htim);
```

该函数实现的功能以及使用方法和 HAL_TIM_Base_Init 都是类似的，作用都是初始化定时器的 ARR 和 PSC 等参数。为什么 HAL 库要提供这个函数而不直接让我们使用 HAL_TIM_Base_Init 函数呢？

这是因为 HAL 库为定时器的 PWM 输出定义了单独的 MSP 回调函数 HAL_TIM_PWM_MspInit，也就是说，当我们调用 HAL_TIM_PWM_Init 进行 PWM 初始化之后，

该函数内部会调用 MSP 回调函数 HAL_TIM_PWM_MspInit。而当我们使用 HAL_TIM_Base_Init 初始化定时器参数的时候，它内部调用的回调函数为 HAL_TIM_Base_MspInit，这里大家注意区分。

所以大家一定要注意，使用 HAL_TIM_PWM_Init 初始化定时器时，回调函数为：HAL_TIM_PWM_MspInit，该函数声明为：

```
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim);
```

一般情况下，上面步骤 1 的时钟使能和 IO 口初始化映射都编写在回调函数内部。

3）设置 TIM3_CH2 的 PWM 模式，输出比较极性，比较值等参数。

接下来，我们要设置 TIM3_CH2 为 PWM 模式（默认是冻结的），因为我们的 DS0 是低电平亮，而我们希望当 CCR2 的值小的时候，DS0 就暗，CCR2 值大的时候，DS0 就亮，所以我们要通过配置 TIM3_CCMR1 的相关位来控制 TIM3_CH2 的模式。

在 HAL 库中，PWM 通道设置是通过函数 HAL_TIM_PWM_ConfigChannel 来设置的：

```
HAL_StatusTypeDef HAL_TIM_PWM_ConfigChannel(TIM_HandleTypeDef *htim,
                                            TIM_OC_InitTypeDef* sConfig, uint32_t Channel);
```

第一个参数 htim 是定时器初始化句柄，也就是 TIM_HandleTypeDef 结构体指针类型，这和 HAL_TIM_PWM_Init 函数调用时候参数保存一致即可。

第二个参数 sConfig 是 TIM_OC_InitTypeDef 结构体指针类型，这也是该函数最重要的参数。该参数用来设置 PWM 输出模式，极性，比较值等重要参数。首先我们来看看结构体定义：

```
typedef struct
{
  uint32_t OCMode;        //PWM 模式
  uint32_t Pulse;         //捕获比较值
  uint32_t OCPolarity;    //极性
  uint32_t OCNPolarity;
  uint32_t OCFastMode;    //快速模式
  uint32_t OCIdleState;
  uint32_t OCNIdleState;
} TIM_OC_InitTypeDef;
```

该结构体成员我们重点关注前三个。成员变量 OCMode 用来设置模式，也就是我们前面讲解的 7 种模式，这里我们设置为 PWM 模式 2。成员变量 Pulse 用来设置捕获比较值。成员变量 TIM_OCPolarity 用来设置输出极性是高还是低。其他的参数 TIM_OutputNState，TIM_OCNPolarity，TIM_OCIdleState 和 TIM_OCNIdleState 是高级定时器才用到的。

第三个参数 Channel 用来选择定时器的通道，取值范围为 TIM_CHANNEL_1~TIM_CHANNEL_4。这里我们使用的是定时器 3 的通道 2，所以取值为 TIM_CHANNEL_2 即可。

例如我们要初始化定时器 3 的通道 2 为 PWM 模式 1，输出极性为低，那么实例代码为：

```
TIM_OC_InitTypeDef  TIM3_CH2Handler;     //定时器 3 通道 2 句柄
TIM3_CH2Handler.OCMode=TIM_OCMODE_PWM1; //模式选择 PWM1
TIM3_CH2Handler.Pulse=arr/2;        //设置比较值,此值用来确定占空比，
                                    //默认比较值为自动重装载值的一半,即占空比为 50%
TIM3_CH2Handler.OCPolarity=TIM_OCPOLARITY_LOW; //输出比较极性为低
HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH2Handler,TIM_CHANNEL_2);
//配置 TIM3 通道 2
```

4）使能 TIM3，使能 TIM3 的 CH2 输出。

在完成以上设置了之后，我们需要使能 TIM3。使能 TIM3 的方法前面已经讲解过：

```
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

该函数第二个入口参数 Channel 是用来设置要使能输出的通道号。

对于单独使能定时器的方法，在上一章定时器实验我们已经讲解。实际上，HAL 库也同样提供了单独使能定时器的输出通道函数，函数为：

```
void TIM_CCxChannelCmd(TIM_TypeDef* TIMx, uint32_t Channel, uint32_t ChannelState);
```

5）修改 TIM3_CCR2 来控制占空比。

最后，在经过以上设置之后，PWM 其实已经开始输出了，只是其占空比和频率都是固定的，而我们通过修改比较值 TIM3_CCR2 则可以控制 CH2 的输出占空比。继而控制 DS0 的亮度。HAL 库中并没有提供独立的修改占空比函数，这里我们可以编写这样一个函数如下：

```
//设置 TIM 通道 2 的占空比
//compare:比较值
void TIM_SetTIM3Compare2 (u32 compare)
{
    TIM3->CCR2=compare;
}
```

实际上，因为调用函数 HAL_TIM_PWM_ConfigChanne 进行 PWM 配置的时候可以设置比较值，所以我们也可以直接使用该函数来达到修改占空比的目的:

```
void TIM_SetCompare1(TIM_TypeDef *TIMx,u32 compare)
{
    TIM3_CH2Handler.Pulse=compare;
    HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH2Handler,
                              TIM_CHANNEL_2);
}
```

这种方法因为要调用 HAL_TIM_PWM_ConfigChannel 函数对各种初始化参数进行重新设置，所以大家在使用中一定要注意，例如在实时系统中如果多个线程同时修改初始化结构体相关参数，可能导致结果混乱。

14.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） 定时器 TIM3

这两个前面都有介绍，但是我们这里用到了 TIM3 的部分重映射功能，把 TIM3_CH2 直接映射到了 PB5 上，而通过前面的学习，我们知道 PB5 和 DS0 是直接连接的，所以电路上并没有任何变化。

14.3 软件设计

打开 PWM 输出实验工程可以看到，我们相比上一节，并没有添加其他任何 HAL 库文件，因为 PWM 是使用的定时器资源，所以跟上一讲使用的是同样的 HAL 库文件。同时我们修改了 timer.c 和 timer.h 的内容，删掉了上一章实验源码，直接把 PWM 功能相关函数和定义放在了这两个文件中。

timer.c 源文件代码如下：

```
TIM_HandleTypeDef    TIM3_Handler;          //定时器句柄
```

```
TIM_OC_InitTypeDef  TIM3_CH2Handler;      //定时器 3 通道 2 句柄
//TIM3 PWM 部分初始化
//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
void TIM3_PWM_Init(u16 arr,u16 psc)
{
    TIM3_Handler.Instance=TIM3;                 //定时器 3
    TIM3_Handler.Init.Prescaler=psc;            //定时器分频
    TIM3_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;//向上计数模式
    TIM3_Handler.Init.Period=arr;               //自动重装载值
    TIM3_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_PWM_Init(&TIM3_Handler);    //初始化 PWM

    TIM3_CH2Handler.OCMode=TIM_OCMODE_PWM1; //模式选择 PWM1
    TIM3_CH2Handler.Pulse=arr/2;            //设置比较值,此值用来确定占空比，
                                            //默认比较值为自动重装载值的一半,即占空比为 50%
    TIM3_CH2Handler.OCPolarity=TIM_OCPOLARITY_LOW; //输出比较极性为低
HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH2Handler,TIM_CHANNEL_2);
                                            //配置 TIM3 通道 2

    HAL_TIM_PWM_Start(&TIM3_Handler,TIM_CHANNEL_2);//开启 PWM 通道 2

}

//定时器底册驱动，开启时钟，设置中断优先级
//此函数会被 HAL_TIM_Base_Init()函数调用
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if(htim->Instance==TIM3)
    {
        __HAL_RCC_TIM3_CLK_ENABLE();    //使能 TIM3 时钟
        HAL_NVIC_SetPriority(TIM3_IRQn,1,3);
                        //设置中断优先级，抢占优先级 1，子优先级 3
        HAL_NVIC_EnableIRQ(TIM3_IRQn);      //开启 ITM3 中断
    }
}

//定时器底层驱动，时钟使能，引脚配置
//此函数会被 HAL_TIM_PWM_Init()调用
//htim:定时器句柄
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)
```

```
{
    GPIO_InitTypeDef GPIO_Initure;

    if(htim->Instance==TIM3)
    {
        __HAL_RCC_TIM3_CLK_ENABLE();            //使能定时器 3
        __HAL_AFIO_REMAP_TIM3_PARTIAL();        //TIM3 通道引脚部分重映射使能
        __HAL_RCC_GPIOB_CLK_ENABLE();           //开启 GPIOB 时钟

        GPIO_Initure.Pin=GPIO_PIN_5;            //PB5
        GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //复用推挽输出
        GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
        GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
        HAL_GPIO_Init(GPIOB,&GPIO_Initure);
    }
}

//设置 TIM 通道 2 的占空比
//compare:比较值
void TIM_SetTIM3Compare2(u32 compare)
{
    TIM3->CCR2=compare;
}
```

此部分代码包含三个函数，完全实现了前面 13.1 小节讲解的 5 个配置步骤。第一个函数 TIM3_PWM_Init 实现的是 13.1 小节讲解的步骤 2-4，首先通过调用定时器 HAL 库函数 HAL_TIM_PWM_Init 初始化 TIM3 并设置 TIM3 的 ARR 和 PSC 等参数，其次通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM3_CH4 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM3 以及使能 PWM 通道 TIM3_CH4 输出。第二个函数 HAL_TIM_PWM_MspInit 是 PWM 的 MSP 初始化回调函数，该函数实现的是 13.1 小节步骤 1，主要是使能相应时钟以及初始化定时器通道 TIM3_CH4 对应的 IO 口模式，同时设置复用映射关系。第三个函数 TIM_SetTIM3Compare 4 是用户自定义的设置比较值函数，这在我们 13.1 小节步骤 5 有详细讲解。

接下来，我们看看 main 函数内容如下：

```
int main(void)
{
    u8 dir=1;
    u16 led0pwmval=0;

    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
```

```
    KEY_Init();                          //初始化按键
    TIM3_PWM_Init(500-1,72-1);           //72M/72=1M 的计数频率，自动重装载为 500，
                                         //那么 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_SetTIM3Compare2(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，我们控制 LED0_PWM_VAL 的值从 0 变到 300，然后又从 300 变到 0，如此循环，因此 DS0 的亮度也会跟着从暗变到亮，然后又从亮变到暗。至于这里的值，我们为什么取 300，是因为 PWM 的输出占空比达到这个值的时候，我们的 LED 亮度变化就不大了（虽然最大值可以设置到 499），因此设计过大的值在这里是没必要的。至此，我们的软件设计就完成了。

14.4 下载验证

在完成软件设计之后，将我们将编译好的文件下载到精英 STM32 V1 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 STM32F1 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：

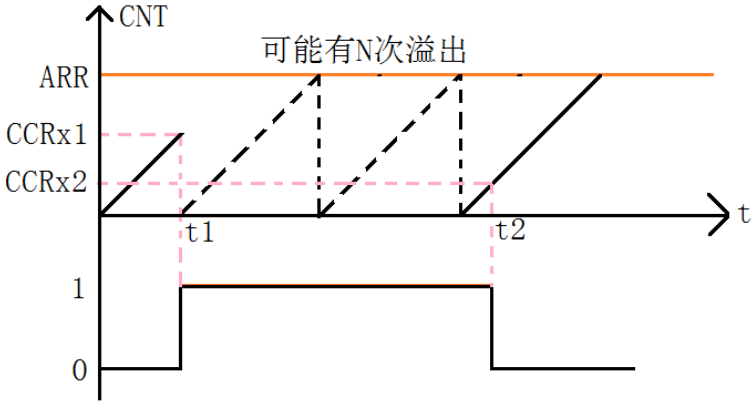


图 15.1.1 输入捕获脉宽测量原理

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 t1~t2 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，t1 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 t2 时刻，又会发生捕获事件，得到此时的 CNT 值，记为 CCRx2。这样，根据定时器的计数频率，我们就可以算出 t1~t2 的时间，从而得到高电平脉宽。

在 t1~t2 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，t1~t2 之间，CNT 计数的次数等于：N*ARR+CCRx2，有了这个计数次数，再乘以 CNT 的计数周期，即可得到 t2-t1 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。

STM32F1 的定时器，除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。STM32F1 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	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《STM32 中文参考手册》290 页，14.4.8 节）。

这里我们用到的是 TIM5 的捕获/比较通道 1，我们重点介绍 TIMx_CCMR1 的[7:0]位（其高 8 位配置类似），TIMx_CCMR1 的[7:0]位详细描述见图 15.1.3 所示：

位7:4	**IC1F[3:0]**：输入捕获1滤波器 (Input capture 1 filter) 这几位定义了TI1输入的采样频率及数字滤波器长度。数字滤波器由一个事件计数器组成，它记录到N个事件后会产生一个输出的跳变： 0000：无滤波器，以f_{DTS}采样　　1000：采样频率$f_{SAMPLING}=f_{DTS}/8$，N=6 0001：采样频率$f_{SAMPLING}=f_{CK_INT}$，N=2　　1001：采样频率$f_{SAMPLING}=f_{DTS}/8$，N=8 0010：采样频率$f_{SAMPLING}=f_{CK_INT}$，N=4　　1010：采样频率$f_{SAMPLING}=f_{DTS}/16$，N=5 0011：采样频率$f_{SAMPLING}=f_{CK_INT}$，N=8　　1011：采样频率$f_{SAMPLING}=f_{DTS}/16$，N=6 0100：采样频率$f_{SAMPLING}=f_{DTS}/2$，N=6　　1100：采样频率$f_{SAMPLING}=f_{DTS}/16$，N=8 0101：采样频率$f_{SAMPLING}=f_{DTS}/2$，N=8　　1101：采样频率$f_{SAMPLING}=f_{DTS}/32$，N=5 0110：采样频率$f_{SAMPLING}=f_{DTS}/4$，N=6　　1110：采样频率$f_{SAMPLING}=f_{DTS}/32$，N=6 0111：采样频率$f_{SAMPLING}=f_{DTS}/4$，N=8　　1111：采样频率$f_{SAMPLING}=f_{DTS}/32$，N=8 注：在现在的芯片版本中，当ICxF[3:0]=1、2或3时，公式中的f_{DTS}由CK_INT替代。
位3:2	**IC1PSC[1:0]**：输入/捕获1预分频器 (Input capture 1 prescaler) 这2位定义了CC1输入(IC1)的预分频系数。 一旦CC1E=’0’(TIMx_CCER寄存器中)，则预分频器复位。 00：无预分频器，捕获输入口上检测到的每一个边沿都触发一次捕获； 01：每2个事件触发一次捕获； 10：每4个事件触发一次捕获； 11：每8个事件触发一次捕获。
位1:0	**CC1S[1:0]**：捕获/比较1选择 (Capture/Compare 1 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00：CC1通道被配置为输出； 01：CC1通道被配置为输入，IC1映射在TI1上； 10：CC1通道被配置为输入，IC1映射在TI2上； 11：CC1通道被配置为输入，IC1映射在TRC上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 **注**：CC1S仅在通道关闭时(TIMx_CCER寄存器的CC1E=’0’)才是可写的。

图 15.1.3 TIMx_CCMR1 [7:0]位详细描述

其中 CC1S[1:0]，这两个位用于 CCR1 的通道配置，这里我们设置 IC1S[1:0]=01，也就是配置 IC1 映射在 TI1 上（关于 IC1，TI1 不明白的，可以看《STM32 中文参考手册》14.2 节的图 98-通用定时器框图），即 CC1 对应 TIMx_CH1。

输入捕获 1 预分频器 IC1PSC[1:0]，这个比较好理解。我们是 1 次边沿就触发 1 次捕获，所以选择 00 就是了。

输入捕获 1 滤波器 IC1F[3:0]，这个用来设置输入采样频率和数字滤波器长度。其中，f_{CK_INT} 是定时器的输入频率（TIMxCLK），一般是 72Mhz，而 f_{DTS} 则是根据 TIMx_CR1 的 CKD[1:0] 的设置来确定的，如果 CKD[1:0]设置为 00，那么 $f_{DTS}=f_{CK\_INT}$。N 值就是滤波长度，举个简单的例子：假设 IC1F[3:0]=0011，并设置 IC1 映射到通道 1 上，且为上升沿触发，那么在捕获

到上升沿的时候，再以f_{CK_INT}的频率，连续采样到 8 次通道 1 的电平，如果都是高电平，则说明确实是一个有效的触发，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**：输入/捕获1输出极性 (Capture/Compare 1 output polarity) **CC1通道配置为输出：** 0：OC1高电平有效 1：OC1低电平有效 **CC1通道配置为输入：** 该位选择是IC1还是IC1的反相信号作为触发或捕获信号。 0：不反相：捕获发生在IC1的上升沿；当用作外部触发器时，IC1不反相。 1：反相：捕获发生在IC1的下降沿；当用作外部触发器时，IC1反相。
位0	**CC1E**：输入/捕获1输出使能 (Capture/Compare 1 output enable) **CC1通道配置为输出：** 0： 关闭－ OC1禁止输出。 1： 开启－ OC1信号输出到对应的输出引脚。 **CC1通道配置为输入：** 该位决定了计数器的值是否能捕获入TIMx_CCR1寄存器。 0：捕获禁止； 0：捕获使能。

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。

开启定时器和 GPIO 时钟的方法和上一章是一样的，这里我们就不做过多讲解。

配置 PA0 为复用功能（AF2）并开启下拉功能也和上一章一样是通过函数 HAL_GPIO_Init

来实现。由于这一步配置过程和上一章几乎没有区别，所以这里我们直接列出配置代码：

```
__HAL_RCC_TIM5_CLK_ENABLE();                //使能 TIM5 时钟
__HAL_RCC_GPIOA_CLK_ENABLE();               //开启 GPIOA 时钟
GPIO_Initure.Pin=GPIO_PIN_0;                //PA0
GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;       //复用推挽输入
GPIO_Initure.Pull=GPIO_PULLDOWN;            //下拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;    //高速
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

2）初始化 TIM5,设置 TIM5 的 ARR 和 PSC。

和上一讲 PWM 输出实验一样，当使用定时器做输入捕获功能时，在 HAL 库中并不使用定时器初始化函数 HAL_TIM_Base_Init 来实现，而是使用输入捕获特定的定时器初始化函数 HAL_TIM_IC_Init。当我们使用函数 HAL_TIM_IC_Init 来初始化定时器的输入捕获功能时，该函数内部会调用输入捕获初始化回调函数 HAL_TIM_IC_MspInit 来初始化与 MCU 无关的步骤。

函数 HAL_TIM_IC_Init 声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_Init(TIM_HandleTypeDef *htim);
```

该函数非常简单，和 HAL_TIM_Base_Init 函数以及函数 HAL_TIM_PWM_Init 使用方法是一模一样的，这里我们就不累赘。

回调函数 HAL_TIM_IC_MspInit 声明如下：

```
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim);
```

该函数使用方法和 PWM 初始化回调函数 HAL_TIM_PWM_MspInit 使用方法一致。一般情况下，输入捕获初始化回调函数中会编写步骤 1 内容，以及后面讲解的 NVIC 配置。

有了 PWM 实验基础知识，这两个函数的使用就非常简单，这里我们列出该步骤程序如下：

```
TIM_HandleTypeDef TIM5_Handler;
TIM5_Handler.Instance=TIM5;                          //通用定时器 5
TIM5_Handler.Init.Prescaler=71;                      //分频系数
TIM5_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;    //向上计数器
TIM5_Handler.Init.Period= 0XFFFFFFFF;                //自动装载值
TIM5_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1; //时钟分频银子
HAL_TIM_IC_Init(&TIM5_Handler);//初始化输入捕获时基参数
```

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。HAL 库是通过 HAL_TIM_IC_ConfigChannel 函数来初始化输入比较参数的：

```
HAL_StatusTypeDef HAL_TIM_IC_ConfigChannel(TIM_HandleTypeDef *htim,
                                           TIM_IC_InitTypeDef* sConfig, uint32_t Channel);
```

该函数有三个参数，第一个参数是定时器初始化结构体指针类型，该参数很好理解。第二个参数是设置要初始化的定时器通道值，取值范围为 TIM_CHANNEL_1~ TIM_CHANNEL_4。接下来我们着重讲解第二个入口参数 sConfig，该参数是 TIM_IC_InitTypeDef 结构体指针类型，它是真正用来初始化定时器通道的捕获参数的。该结构体类型定义为：

```
typedef struct
{
  uint32_t    ICPolarity;
  uint32_t ICSelection;
```

```
    uint32_t ICPrescaler;
    uint32_t ICFilter;
} TIM_IC_InitTypeDef;
```

成员变量 ICPolarity 用来设置输入信号的有效捕获极性，取值范围为：TIM_ICPOLARITY_RISING（上升沿捕获），TIM_ICPOLARITY_FALLING（下降沿捕获）和 TIM_ICPOLARITY_BOTHEDGE（双边沿）捕获。实际上，HAL 还提供了设置输入捕获极性以及清除输入捕获极性设置方法。如下：

```
TIM_RESET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1);//清除极性设置
TIM_SET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1,
                        TIM_ICPOLARITY_FALLING);//定时器 5 通道 1 设置为下降沿捕获
```

成员变量 ICSelection 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。

成员变量 ICPrescaler 用来设置输入捕获分频系数，可以设置为 TIM_ICPSC_DIV1（不分频），TIM_ICPSC_DIV2（2 分频），TIM_ICPSC_DIV4（4 分频）以及 TIM_ICPSC_DIV8（8 分频），本实验需要设置为不分频，所以选值为 TIM_ICPSC_DIV1。

成员变量 ICFilter 用来设置滤波器长度，这里我们不使用滤波器，所以设置为 0。

本实验，我们要设置输入捕获参数为：上升沿捕获，不分频，不滤波，同时 IC1 映射到 TI1(通道 1)上，实例代码如下：

```
TIM_IC_InitTypeDef TIM5_CH1Config;
TIM5_CH1Config.ICPolarity=TIM_ICPOLARITY_RISING;      //上升沿捕获
TIM5_CH1Config.ICSelection=TIM_ICSELECTION_DIRECTTI;//IC1 映射到 TI1 上
TIM5_CH1Config.ICPrescaler=TIM_ICPSC_DIV1;            //配置输入分频，不分频
TIM5_CH1Config.ICFilter=0;                            //配置输入滤波器，不滤波
HAL_TIM_IC_ConfigChannel(&TIM5_Handler,&TIM5_CH1Config,TIM_CHANNEL_1);
```

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。

HAL 库中开启定时器中断方法在定时器中断实验已经讲解，方法为：

```
__HAL_TIM_ENABLE_IT(&TIM5_Handler,TIM_IT_UPDATE);    //使能更新中断
```

实际上，由于本章使用的是定时器的输入捕获功能，HAL 还提供了一个函数同时用来开启定时器的输入捕获通道和使能捕获中断，该函数为：

```
HAL_StatusTypeDef HAL_TIM_IC_Start_IT (TIM_HandleTypeDef *htim, uint32_t Channel);
```

实际上该函数同时还使能了定时器，一个函数具备三个功能。

如果我们不需要开启捕获中断，只是开启输入捕获功能，HAL 库函数为：

```
HAL_StatusTypeDef HAL_TIM_IC_Start (TIM_HandleTypeDef *htim, uint32_t Channel);
```

5）使能定时器（设置 TIM5 的 CR1 寄存器）

在步骤 4 中，如果我们调用了函数 HAL_TIM_IC_Start_IT 来开启输入捕获通道以及输入捕获中断，实际上它同时也开启了相应的定时器。单独的开启定时器的方法为：

```
__HAL_TIM_ENABLE();  //开启定时器方法
```

6）设置 NVIC 中断优先级

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，这里方法跟我们前面讲解一致，这里我们就不累赘了。、

这里大家要注意，一般情况下 NVIC 配置我们都会放在 MSP 回调函数中。对于输入捕获功能，回调函数是我们步骤 2 讲解的函数 HAL_TIM_IC_MspInit。

7）编写中断服务函数

最后编写中断服务函数。定时器 5 中断服务函数为：

```
void TIM5_IRQHandler(void);
```

和定时器中断实验一样，一般情况下，我们都不把中断控制逻辑直接编写在中断服务函数中，因为 HAL 库提供了一个共用的中断处理入口函数 HAL_TIM_IRQHandler，该函数中会对中断来源进行判断然后调用相应的中断处理回调函数。HAL 库提供了多个中断处理回调函数，本章实验，我们要使用到更新中断和捕获中断，所以我们要使用的回调函数为：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);//更新（溢出）中断
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim);//捕获中断
```

我们只需要在我们工程中，重新定义这两个函数，编写中断处理控制逻辑即可。

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 PB5 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，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 HAL 库文件。

接下来我们来看看 timer.c 文件中，我们添加的几个函数的内容：

```
TIM_HandleTypeDef TIM5_Handler;         //定时器 5 句柄
//定时器 5 通道 1 输入捕获配置
//arr：自动重装值(TIM5 是 16 位的!!)
//psc：时钟预分频数
void TIM5_CH1_Cap_Init(u32 arr,u16 psc)
{
    TIM_IC_InitTypeDef TIM5_CH1Config;

    TIM5_Handler.Instance=TIM5;                          //通用定时器 5
    TIM5_Handler.Init.Prescaler=psc;                     //分频系数
    TIM5_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;    //向上计数器
    TIM5_Handler.Init.Period=arr;                        //自动装载值
```

```
    TIM5_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;//时钟分频银子
    HAL_TIM_IC_Init(&TIM5_Handler);                    //初始化输入捕获时基参数

    TIM5_CH1Config.ICPolarity=TIM_ICPOLARITY_RISING;    //上升沿捕获
    TIM5_CH1Config.ICSelection=TIM_ICSELECTION_DIRECTTI;//映射到 TI1 上
    TIM5_CH1Config.ICPrescaler=TIM_ICPSC_DIV1;          //配置输入分频，不分频
    TIM5_CH1Config.ICFilter=0;                          //配置输入滤波器，不滤波
    HAL_TIM_IC_ConfigChannel(&TIM5_Handler,&TIM5_CH1Config,TIM_CHANNEL_1);
                                                        //配置 TIM5 通道 1
    HAL_TIM_IC_Start_IT(&TIM5_Handler,TIM_CHANNEL_1);
                                        //开启 TIM5 的捕获通道 1，并且开启捕获中断
    __HAL_TIM_ENABLE_IT(&TIM5_Handler,TIM_IT_UPDATE);   //使能更新中断
    HAL_NVIC_SetPriority(TIM5_IRQn,2,0);//设置中断优先级，抢占优先级 2，子优先级 0
    HAL_NVIC_EnableIRQ(TIM5_IRQn);  //开启 ITM5 中断通道
}

//定时器 5 底层驱动，时钟使能，引脚配置
//此函数会被 HAL_TIM_IC_Init()调用
//htim:定时器 5 句柄
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_TIM5_CLK_ENABLE();            //使能 TIM5 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_0;            //PA0
    GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;   //复用推挽输入
    GPIO_Initure.Pull=GPIO_PULLDOWN;        //下拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    HAL_NVIC_SetPriority(TIM5_IRQn,2,0);//设置中断优先级，抢占优先级 2，子优先级 0
    HAL_NVIC_EnableIRQ(TIM5_IRQn);  //开启 ITM5 中断通道
}

//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
u8   TIM5CH1_CAPTURE_STA=0;                 //输入捕获状态
u16  TIM5CH1_CAPTURE_VAL;                   //输入捕获值(TIM5 是 16 位)

//定时器 5 中断服务函数
```

```
void TIM5_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM5_Handler);                    //定时器共用处理函数
}
//定时器更新中断（计数溢出）中断处理回调函数，
//该函数在 HAL_TIM_IRQHandler 中会被调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
                                                          //更新中断（溢出）发生时执行
{
    if((TIM5CH1_CAPTURE_STA&0X80)==0)                     //还未成功捕获
    {
        if(TIM5CH1_CAPTURE_STA&0X40)                      //已经捕获到高电平了
        {
            if((TIM5CH1_CAPTURE_STA&0X3F)==0X3F)//高电平太长了
            {
                TIM5CH1_CAPTURE_STA|=0X80;                //标记成功捕获了一次
                TIM5CH1_CAPTURE_VAL=0XFFFF;
            }else TIM5CH1_CAPTURE_STA++;
        }
    }
}

//定时器输入捕获中断处理回调函数，该函数在 HAL_TIM_IRQHandler 中会被调用
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)//捕获中断发生时执行
{
    if((TIM5CH1_CAPTURE_STA&0X80)==0)                     //还未成功捕获
    {
        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=HAL_TIM_ReadCapturedValue
                        (&TIM5_Handler,TIM_CHANNEL_1);//获取当前的捕获值.
            TIM_RESET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1);
                                                          //一定要先清除原来的设置!!
            TIM_SET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1,
                TIM_ICPOLARITY_RISING);//配置 TIM5 通道 1 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
            __HAL_TIM_DISABLE(&TIM5_Handler); //关闭定时器 5
            __HAL_TIM_SET_COUNTER(&TIM5_Handler,0);
```

```
            TIM_RESET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1);
                                                    //一定要先清除原来的设置!!
            TIM_SET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1,
              TIM_ICPOLARITY_FALLING);      //定时器 5 通道 1 设置为下降沿捕获
            __HAL_TIM_ENABLE(&TIM5_Handler);        //使能定时器 5
        }
    }
}
```

此部分代码包含 2 个函数，其中 TIM5_Cap_Init 函数用于 TIM5 通道 1 的输入捕获设置，其设置和我们上面讲的步骤是一样的，这里就不多说，重点来看看第二个函数。

TIM5_IRQHandler 是 TIM5 的中断服务函数，该函数用到了两个全局变量，用于辅助实现高电平捕获。其中 TIM5CH1_CAPTURE_STA，是用来记录捕获状态，该变量类似我们在 usart.c 里面自行定义的 USART_RX_STA 寄存器(其实就是个变量，只是我们把它当成一个寄存器那样来使用)。TIM5CH1_CAPTURE_STA 各位描述如表 15.3.1 所示：

TIM5CH1_CAPTURE_STA		
bit7	bit6	bit5~0
捕获完成标志	捕获到高电平标志	捕获高电平后定时器溢出的次数

表 15.3.1 TIM5CH1_CAPTURE_STA 各位描述

另外一个变量 TIM5CH1_CAPTURE_VAL，则用来记录捕获到下降沿的时候，TIM5_CNT 的值。

现在我们来介绍一下，捕获高电平脉宽的思路：首先，设置 TIM5_CH1 捕获上升沿，这在 TIM5_Cap_Init 函数执行的时候就设置好了，然后等待上升沿中断到来，当捕获到上升沿中断，此时如果 TIM5CH1_CAPTURE_STA 的第 6 位为 0，则表示还没有捕获到新的上升沿，就先把 TIM5CH1_CAPTURE_STA、TIM5CH1_CAPTURE_VAL 和 TIM5->CNT 等清零，然后再设置 TIM5CH1_CAPTURE_STA 的第 6 位为 1，标记捕获到高电平，最后设置为下降沿捕获，等待下降沿到来。如果等待下降沿到来期间，定时器发生了溢出（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;
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                         //初始化延时函数
```

```
    uart_init(115200);                  //初始化串口
    LED_Init();                         //初始化 LED
    TIM3_PWM_Init(500-1,72-1);          //72M/72=1M 的计数频率，自动重装载为 500，
                                        //那么 PWM 频率为 1M/500=2kHZ
    TIM5_CH1_Cap_Init(0XFFFF,72-1);     //以 1Mhz 的频率计数
    while(1)
    {
        delay_ms(10);
        TIM_SetTIM3Compare2(TIM_GetTIM3Capture2()+1);
        if(TIM_GetTIM3Capture2()==300)TIM_SetTIM3Compare2(0);
        if(TIM5CH1_CAPTURE_STA&0X80)        //成功捕获到了一次高电平
        {
            temp=TIM5CH1_CAPTURE_STA&0X3F;
            temp*=65536;                    //溢出时间总和
            temp+=TIM5CH1_CAPTURE_VAL;      //得到总的高电平时间
            printf("HIGH:%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(0XFFFF,72-1)，将 TIM5_CH1 的捕获计数器设计为 1us 计数一次，并设置重装载值为最大，所以我们的捕获时间精度为 1us。

主函数通过 TIM5CH1_CAPTURE_STA 的第 7 位，来判断有没有成功捕获到一次高电平，如果成功捕获，则将高电平时间通过串口输出到电脑。

至此，我们的软件设计就完成了。

15.4 下载验证

在完成软件设计之后，将我们将编译好的文件下载到精英 STM32 V1 上，可以看到 DS0 的状态和上一章差不多，由暗→亮的循环。说明程序已经正常在跑了，我们再打开串口调试助手，选择对应的串口，然后按 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 DS0 亮度

从上图可以看出，其中有 1 次高电平在 100us 以内的，这种就是按键按下时发生的抖动。这就是为什么我们按键输入的时候，一般都需要做防抖处理，防止类似的情况干扰正常输入。大家还可以用杜邦线连接 PA0 和 PB5，看看上一节中我们设置的 PWM 输出的高电平是如何变化的。

第十六章 电容触摸按键实验

上一章，我们介绍了 STM32F1 的输入捕获功能及其使用。这一章，我们将向大家介绍如何通过输入捕获功能，来做一个电容触摸按键。在本章中，我们将用 TIM5 的通道 2（PA1）来做输入捕获，并实现一个简单的电容触摸按键，通过该按键控制 DS1 的亮灭。从本章分为如下几个部分：

16.1 电容触摸按键简介
16.2 硬件设计
16.3 软件设计
16.4 下载验证

16.1 电容触摸按键简介

触摸按键相对于传统的机械按键有寿命长、占用空间少、易于操作等诸多优点。大家看看如今的手机，触摸屏、触摸按键大行其道，而传统的机械按键，正在逐步从手机上面消失。本章，我们将给大家介绍一种简单的触摸按键：电容式触摸按键。

我们将利用精英 STM32F103 上的触摸按键（TPAD），来实现对 DS1 的亮灭控制。这里 TPAD 其实就是精英 STM32F103 上的一小块覆铜区域，实现原理如图 16.1.1 所示：

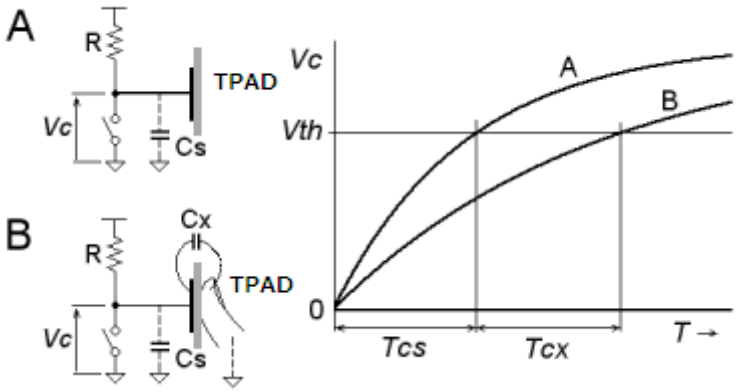


图 16.1.1 电容触摸按键原理

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 STM32F1 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：

Vc=V0*(1-e^(-t/RC))

其中 Vc 为电容电压，V0 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，我们还可以把精英开发板作为一个简单的电容计，直接可以测电容容量了，有兴趣的朋友可以捣鼓下。

在本章中，其实我们只要能够区分 Tcs 和 Tcs+Tcx，就已经可以实现触摸检测了，当充电时间在 Tcs 附近，就可以认为没有触摸，而当充电时间大于 Tcs+Tx 时，就认为有触摸按下（Tx 为检测阀值）。

本章，我们使用 PA1(TIM5_CH2)来检测 TPAD 是否有触摸，在每次检测之前，我们先配置 PA1 为推挽输出，将电容 Cs（或 Cs+Cx）放电，然后配置 PA1 为浮空输入，利用外部上拉电阻给电容 Cs(Cs+Cx)充电，同时开启 TIM5_CH2 的输入捕获，检测上升沿，当检测到上升沿的时候，就认为电容充电完成了，完成一次捕获检测。

在 MCU 每次复位重启的时候，我们执行一次捕获检测（可以认为没触摸），记录此时的值，记为 tpad_default_val，作为判断的依据。在后续的捕获检测，我们就通过与 tpad_default_val 的对比，来判断是不是有触摸发生。

关于输入捕获的配置，在上一章我们已经有详细介绍了，这里我们就不再介绍。至此，电容触摸按键的原理介绍完毕。

16.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0 和 DS1

2） 定时器 TIM5

3） 触摸按键 TPAD

前面两个之前均有介绍，我们需要通过 TIM5_CH2（PA1）采集 TPAD 的信号，所以本实验需要用跳线帽短接多功能端口（P7）的 TPAD 和 ADC，以实现 TPAD 连接到 PA1。如图 16.2.1 所示：

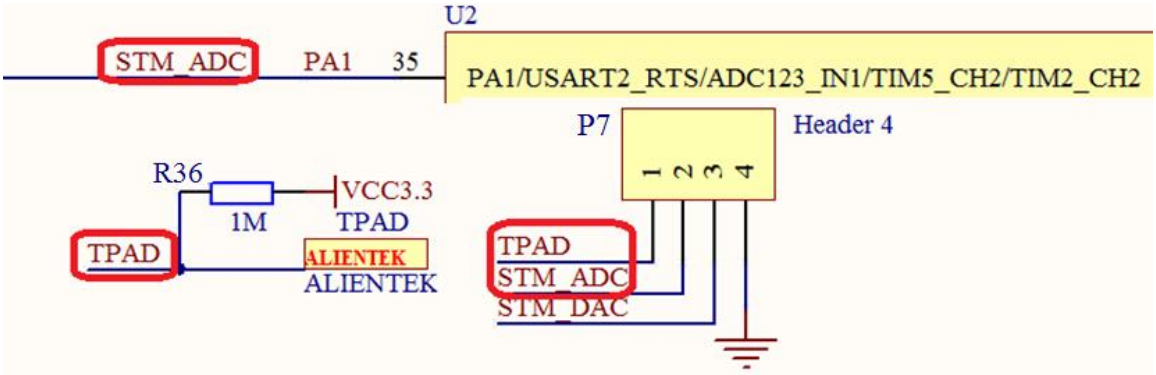


图 16.2.1 TPAD 与 STM32F1 连接原理图

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

软件设计我们在之前的工程上面增加，由于本章我们用不到 timer.c，所以先删掉 timer.c。然后在 HARDWARE 文件夹下新建 TPAD 的文件夹。然后打开 USER 文件夹下的工程，新建一个 tpad.c 的文件和 tpad.h 的头文件，保存在 TAPD 文件夹下，并将 TPAD 文件夹加入头文件包含路径。

我们在 tpad.c 里输入如下代码：

```
#include "tpad.h"
#include "delay.h"
#include "usart.h"
TIM_HandleTypeDef TIM5_Handler;          //定时器 5 句柄
#define TPAD_ARR_MAX_VAL  0XFFFF  //最大的 ARR 值(TIM5 是 16 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];
    u16 temp;
    u8 j,i;
```

```
    TIM5_CH2_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>(vu16)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_Initure;

    GPIO_Initure.Pin=GPIO_PIN_1;                    //PA1
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLDOWN;                //下拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH; //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    HAL_GPIO_WritePin(GPIOA,GPIO_PIN_1,GPIO_PIN_RESET);  //PA1 输出 0，放电
    delay_ms(5);
   __HAL_TIM_CLEAR_FLAG(&TIM5_Handler,TIM_FLAG_CC2|TIM_FLAG_UPDATE);
                                                    //清除标志位
    __HAL_TIM_SET_COUNTER(&TIM5_Handler,0); //计数器值归 0
```

```
    GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;       //推挽复用输入
    GPIO_Initure.Pull=GPIO_NOPULL;              //不带上下拉
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
}

//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(__HAL_TIM_GET_FLAG(&TIM5_Handler,TIM_FLAG_CC2)==RESET)
                                                    //等待捕获上升沿
    {
    if(__HAL_TIM_GET_COUNTER(&TIM5_Handler)>TPAD_ARR_MAX_VAL-500)
    return __HAL_TIM_GET_COUNTER(&TIM5_Handler);//超时了，直接返回 CNT 的值
    };
    return HAL_TIM_ReadCapturedValue(&TIM5_Handler,TIM_CHANNEL_2);
}

//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;
    u16 res=0;
    u8 lcntnum=n*2/3;//至少 2/3*n 的有效个触摸,才算有效
    u8 okcnt=0;
    while(n--)
    {
        temp=TPAD_Get_Val();//得到一次值
        if(temp>(tpad_default_val*5/4))okcnt++;//至少大于默认值的 5/4 才算有效
        if(temp>res)res=temp;
    }
    if(okcnt>=lcntnum)return res;//至少 2/3 的概率,要大于默认值的 5/4 才算有效
    else return 0;
}

//扫描触摸按键
//mode:0,不支持连续触发(按下一次必须松开才能按下一次);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;
u8 TPAD_Scan(u8 mode)
{
    static u8 keyen=0;  //0,可以开始检测;>0,还不能开始检测
    u8 res=0;
    u8 sample=3;        //默认采样次数为 3 次
    u16 rval;
    if(mode)
    {
        sample=6;       //支持连按的时候，设置采样次数为 6 次
        keyen=0;        //支持连按
    }
    rval=TPAD_Get_MaxVal(sample);
    if(rval>(tpad_default_val*4/3)&&rval<(10*tpad_default_val))
      //大于 tpad_default_val+(1/3)*tpad_default_val,且小于 10 倍 tpad_default_val,则有效
    {
        if(keyen==0)res=1; //keyen==0,有效
        //printf("r:%d\r\n",rval);
        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 5 通道 2 输入捕获配置
//arr：自动重装值(TIM2 是 16 位的!!)
//psc：时钟预分频数
void TIM5_CH2_Cap_Init(u32 arr,u16 psc)
{
    TIM_IC_InitTypeDef TIM5_CH2Config;

    TIM5_Handler.Instance=TIM5;                                 //通用定时器 5
    TIM5_Handler.Init.Prescaler=psc;                            //分频
    TIM5_Handler.Init.CounterMode=TIM_COUNTERMODE_UP; //向上计数器
    TIM5_Handler.Init.Period=arr;                               //自动装载值
    TIM5_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_IC_Init(&TIM5_Handler);

    TIM5_CH2Config.ICPolarity=TIM_ICPOLARITY_RISING;    //上升沿捕获
    TIM5_CH2Config.ICSelection=TIM_ICSELECTION_DIRECTTI;//映射到 TI1 上
    TIM5_CH2Config.ICPrescaler=TIM_ICPSC_DIV1;              //配置输入分频，不分频
    TIM5_CH2Config.ICFilter=0;                                  //配置输入滤波器，不滤波
```

```
    HAL_TIM_IC_ConfigChannel(&TIM5_Handler,&TIM5_CH2Config,TIM_CHANNEL_2);
                                                    //配置 TIM5 通道 2
    HAL_TIM_IC_Start(&TIM5_Handler,TIM_CHANNEL_2); //开始捕获 TIM5 的通道 2
}

//定时器 5 底层驱动，时钟使能，引脚配置
//此函数会被 HAL_TIM_IC_Init()调用
//htim:定时器 5 句柄
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_TIM5_CLK_ENABLE();            //使能 TIM5 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_1;            //PA1
    GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;   //推挽复用输入
    GPIO_Initure.Pull=GPIO_NOPULL;          //不带上下拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
}
```

函数 TIM5_CH2_Cap_Init 和上一章输入捕获实验中函数 TIM5_CH1_Cap_Init 的配置过程几乎是一模一样的，不同的是上一章实验开 TIM5_CH1_Cap_Init 函数最后调用的是函数 HAL_TIM_IC_Start_IT，使能输入捕获通道的同事开启了输入捕获中断，而该函数最后调用的函数是 HAL_TIM_IC_Start，只是开启了输入捕获通道，并没有开启输入捕获中断。

函数 HAL_TIM_IC_MspInit 是输入捕获通用 MSP 回调函数，该函数的作用是使能定时器和 GPIO 时钟，配置 GPIO 复用映射关系。该函数功能和输入捕获实验中该函数作用基本类似。

函数 TPAD_Get_Val 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，同时设置通过程序__HAL_TIM_SET_COUNTER(&TIM5_Handler,0)将计数值 TIM5_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。

函数 TPAD_Init 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，用来传递分频系数，其实是为了配置 TIM5_CH2_Cap_Init 的计数周期。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。

函数 TPAD_Scan 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，类似第七章的 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，取最大值和 tpad_default_val*4/3 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，然后取其 4/3 为门限值。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。

函数 TPAD_Reset 顾名思义，是进行一次复位操作。先设置 PA1 输出低电平，电容放电，同时清除中断标志位并且计数器值清零，然后配置 PA1 为复用功能浮空输入，利用外部上拉电

阻给电容 Cs(Cs+Cx)充电，同时开启 TIM5_CH2 的输入捕获。

函数 TPAD_Get_MaxVal 就非常简单了，它通过 n 次调用函数 TPAD_Get_Val 采集捕获值，然后进行比较后获取 n 次采集值中的最大值。

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;

    HAL_Init();                          //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                      //初始化延时函数
    uart_init(115200);                   //初始化串口
    LED_Init();                          //初始化 LED
    TPAD_Init(6);                        //初始化触摸按键

    while(1)
    {
        if(TPAD_Scan(0))   //成功捕获到了一次上升沿(此函数执行时间至少 15ms)
        {
            LED1=!LED1;                  //LED1 取反
        }
        t++;
        if(t==15)
        {
            t=0;
            LED0=!LED0;                  //LED0 取反,提示程序正在运行
        }
        delay_ms(10);
    }
}
```

该 main 函数比较简单，TPAD_Init(6)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精英 STM32 V1 上，可以看到 DS0 慢速闪烁，此时，我们用手指触摸 ALIENTEK 精英 STM32F103 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：

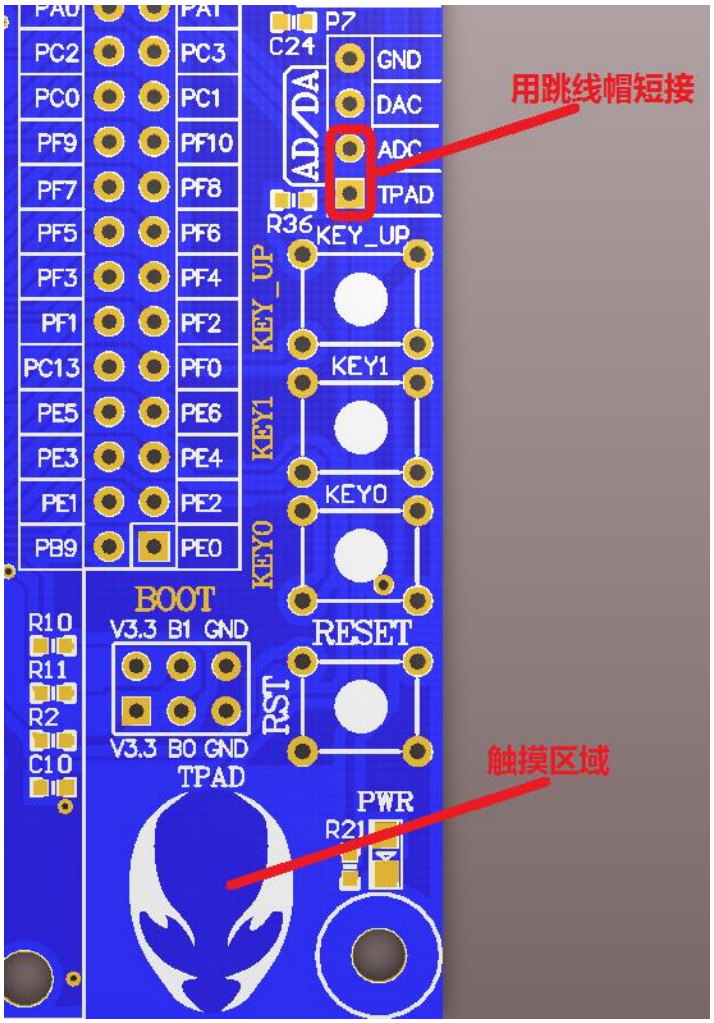


图 16.4.1 触摸区域和跳线帽短接方式示意图

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 150 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，我们将使用精英 STM32F103 V1 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 Electroluminesence 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：

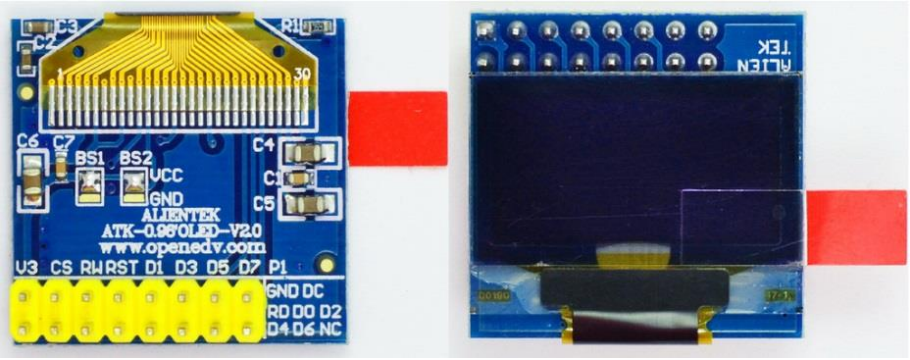


图 17.1.1 ALIENTEK OLED 模块外观图

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：

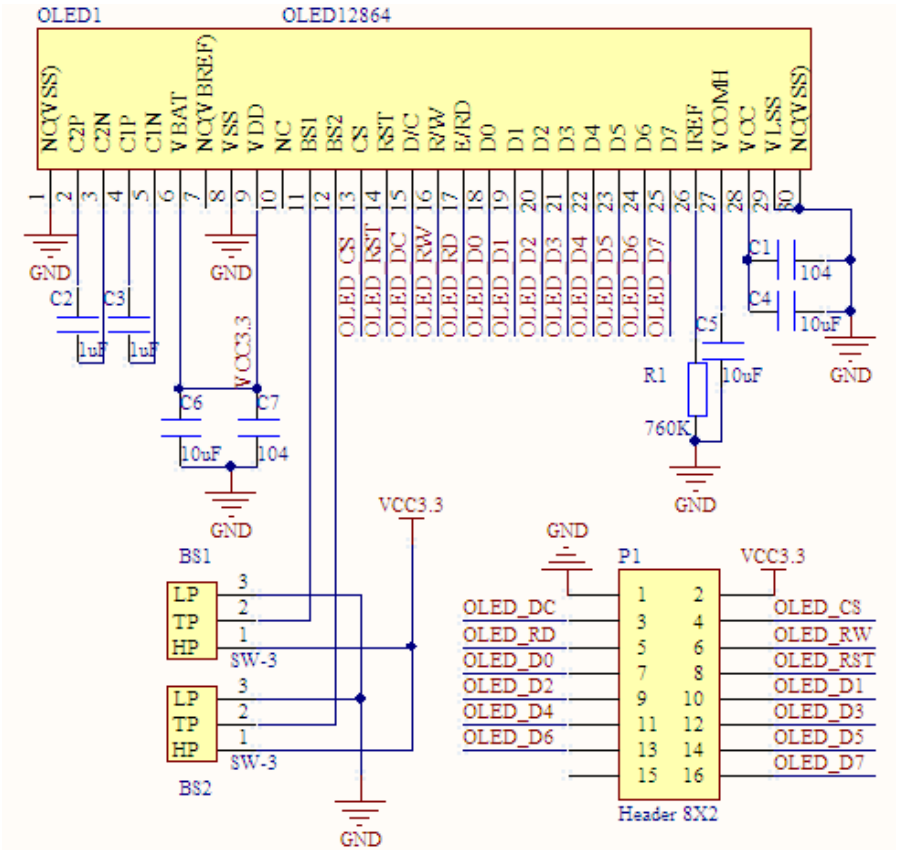


图 17.1.2 ALIENTEK OLED 模块原理图

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 STM32F1 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：

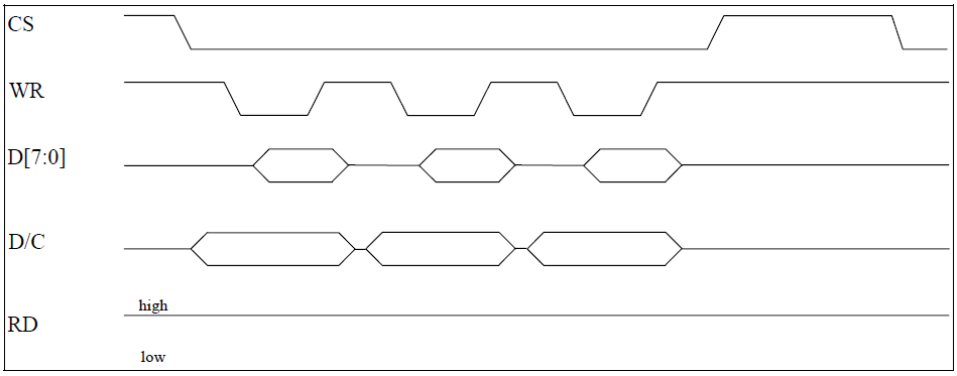


图 17.1.3　8080 并口写时序图

SSD1306 的 8080 并口读时序图如图 17.1.4 所示：

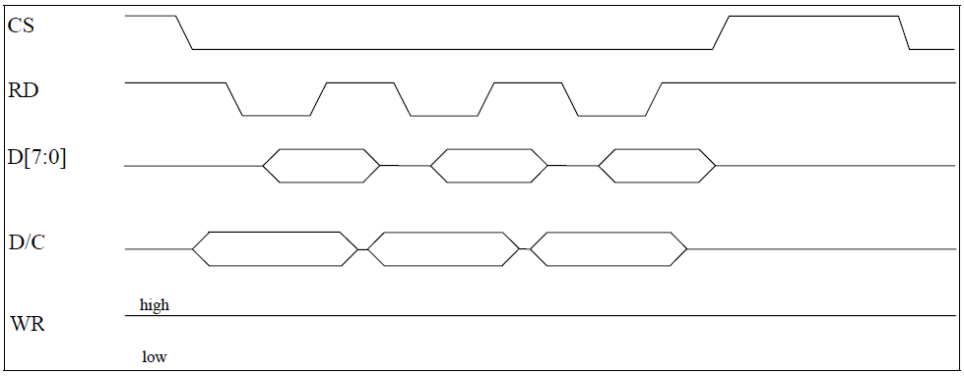


图 17.1.4　8080 并口读时序图

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，才是我们真正要读的数据。

一个典型的读显存的时序图，如图 17.1.5 所示：

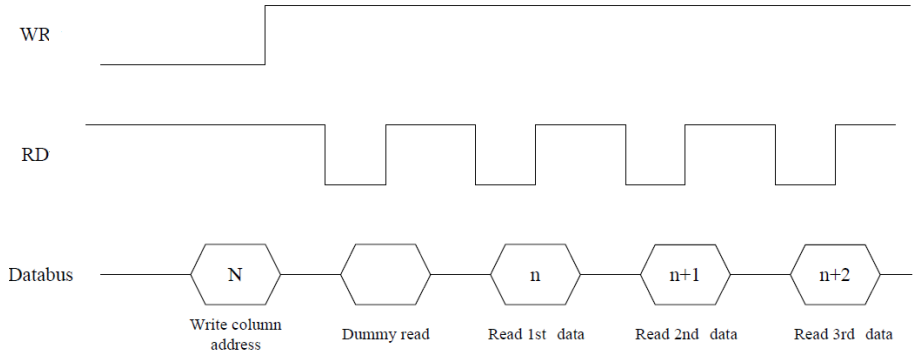


图 17.1.5 读显存时序图

可以看到，在发送了列地址之后，开始读数据，第一个是 Dummy Read，也就是假读，我们从第二个开始，才算是真正有效的数据。

并行接口模式就介绍到这里，我们接下来介绍一下 4 线串行（SPI）方式，4 先串口模式使用的信号线有如下几条：

CS：OLED 片选信号。

RST(RES)：硬复位 OLED。

DC：命令/数据标志（0，读写命令；1，读写数据）。

SCLK：串行时钟线。在 4 线串行模式下，D0 信号线作为串行时钟线 SCLK。

SDIN：串行数据线。在 4 线串行模式下，D1 信号线作为串行数据线 SDIN。

模块的 D2 需要悬空，其他引脚可以接到 GND。在 4 线串行模式下，只能往模块写数据而不能读数据。

在 4 线 SPI 模式下，每个数据长度均为 8 位，在 SCLK 的上升沿，数据从 SDIN 移入到 SSD1306，并且是高位在前的。DC 线还是用作命令/数据的标志线。在 4 线 SPI 模式下，写操作的时序如图 17.1.6 所示：

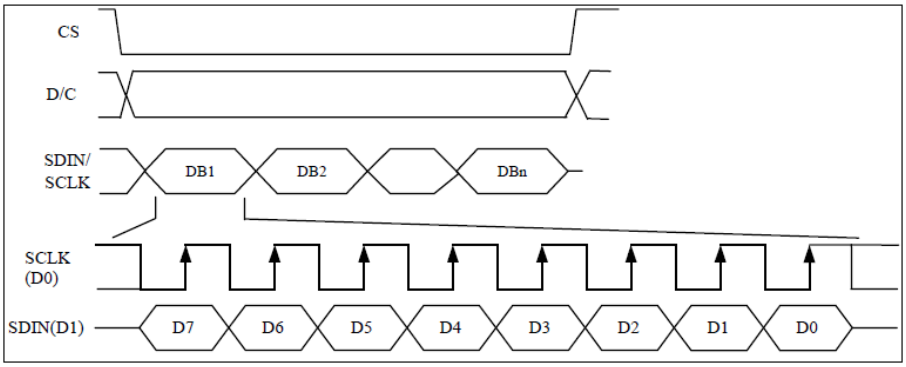


图 17.1.6 4 线 SPI 写操作时序图

4 线串行模式就为大家介绍到这里。其他还有几种模式，在 SSD1306 的数据手册上都有详细的介绍，如果要使用这些方式，请大家参考该手册。

接下来，我们介绍一下模块的显存，SSD1306 的显存总共为 128*64bit 大小，SSD1306 将这些显存分为了 8 页，其对应关系如表 17.1.3 所示：

	列（COL0~127）						
行（COM0~63）	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 STM32F1 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 STM32F1 上的 GRAM（实际上就是 SRAM），在修改完了之后，一次性把 STM32F1 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	D0		
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：

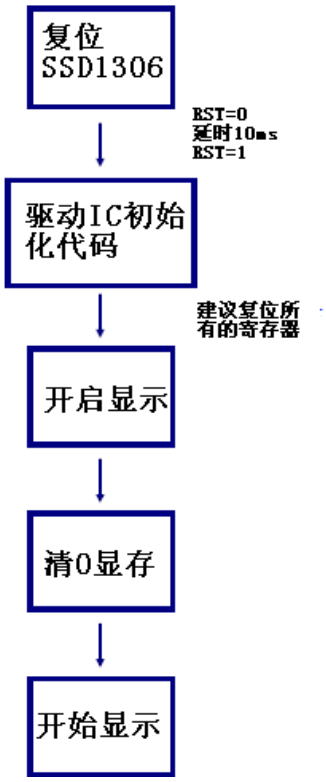


图 17.1.7　SSD1306 初始化框图

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 STM32F1 与 OLED 模块相连接的 IO。

这一步，先将我们与 OLED 模块相连的 IO 口设置为输出，具体使用哪些 IO 口，这里需要根据连接电路以及 OLED 模块所设置的通讯模式来确定。这些将在硬件设计部分向大家介绍。

2）初始化 OLED 模块。

其实这里就是上面的初始化框图的内容，通过对 OLED 相关寄存器的初始化，来启动 OLED 的显示。为后续显示字符和数字做准备。

3）通过函数将字符和数字显示到 OLED 模块上。

这里就是通过我们设计的程序，将要显示的字符送到 OLED 模块就可以了，这些函数将在软件设计部分向大家介绍。

通过以上三步，我们就可以使用 ALIENTEK OLED 模块来显示字符和数字了，在后面我们还将会给大家介绍显示汉字的方法。这一部分就先介绍到这里。

17.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） OLED 模块

OLED 模块的电路在前面已有详细说明了，这里我们介绍 OLED 模块与精英 STM32F103 的连接，开发板底板的 OLED/CAMERA 接口（P4 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：

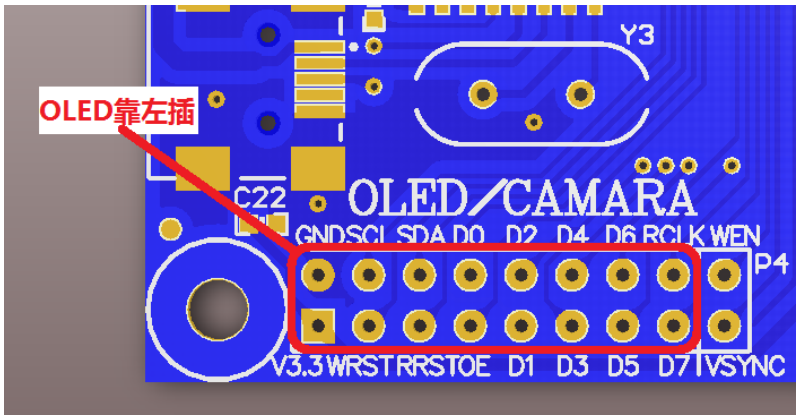


图 17.2.1 OLED 模块与开发板连接示意图

图中圈出来的部分就是连接 OLED 的接口，这里在硬件上，OLED 与精英 STM32F103 的 IO 口对应关系如下：

OLED_CS 对应 PD6;

OLED_RST 对应 PG15;

OLED_RS 对应 PD3;

OLED_WR 对应 PG14;

OLED_RD 对应 PG13;

OLED_D[7：0]对应 PC[7：0];

这些线的连接，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

软件设计我们依旧在之前的工程上面增加，不过没用到电容触摸按键，所以先去掉 tpad.c（注意，此时 HARDWARE 组仅剩：led.c）。然后，在 HARDWARE 文件夹下新建一个 OLED 的文件夹。然后打开 USER 文件夹下的工程，新建一个 oled.c 的文件和 oled.h 的头文件，保存在 OLED 文件夹下，并将 OLED 文件夹加入头文件包含路径。

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_Initure;
    __HAL_RCC_GPIOC_CLK_ENABLE();     //使能 GPIOC 时钟
    __HAL_RCC_GPIOD_CLK_ENABLE();     //使能 GPIOD 时钟
    __HAL_RCC_GPIOG_CLK_ENABLE();     //使能 GPIOG 时钟

    //GPIO 初始化设置
    GPIO_Initure.Pin=GPIO_PIN_3|GPIO_PIN_6;          //PD3、PD6 推挽输出
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;    //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;                   //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;  //快速
    HAL_GPIO_Init(GPIOD,&GPIO_Initure);              //初始化

    //PD3,PD6  输出高
    HAL_GPIO_WritePin(GPIOD,GPIO_PIN_3,GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOD,GPIO_PIN_6,GPIO_PIN_SET);

#if OLED_MODE==1          //使用 8080 并口模式

    //PC0~7 OUT 推挽输出
    GPIO_Initure.Pin=GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3|\
                     GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7;
    HAL_GPIO_Init(GPIOC,&GPIO_Initure);          //初始化

    //PG13,14,15 OUT 推挽输出
    GPIO_Initure.Pin=GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);          //初始化

    OLED_WR=1;
    OLED_RD=1;
```

```
#else                           //使用 4 线 SPI 串口模式

    //GPIO 初始化设置
    GPIO_Initure.Pin=GPIO_PIN_0|GPIO_PIN_1;         //PC0,1 OUT 推挽输出
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;             //高速
    HAL_GPIO_Init(GPIOC,&GPIO_Initure);             //初始化

    //PG15
    GPIO_Initure.Pin=GPIO_PIN_15;
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);//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 STM32F1 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 STM32F1 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—列低地址
        OLED_WR_Byte (0x10,OLED_CMD);        //设置显示位置—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 STM32F1 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  //8080 并口
//向 SSD1306 写入一个字节。
//dat:要写入的数据/命令
//cmd:数据/命令标志 0,表示命令;1,表示数据;
void OLED_WR_Byte(u8 dat,u8 cmd)
{
    DATAOUT(dat);
    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
```

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 SPI 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:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`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：

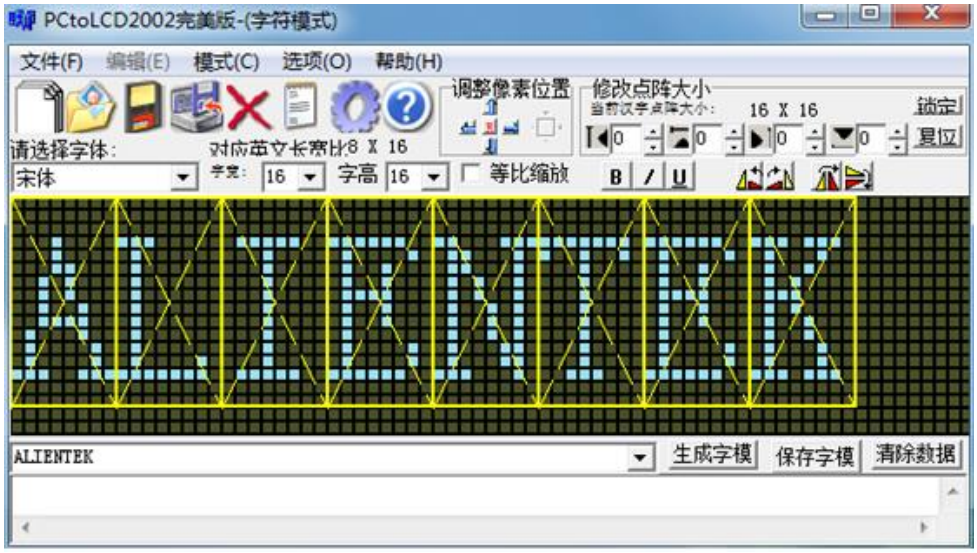


图 17.3.1 PCtoLCD2002 软件界面

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：

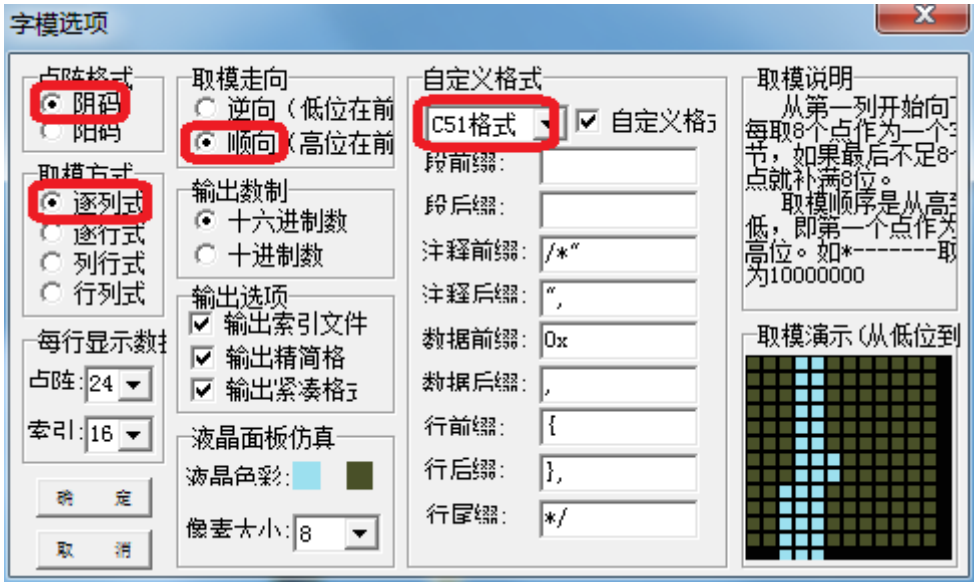


图 17.3.2 设置取模方式

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：

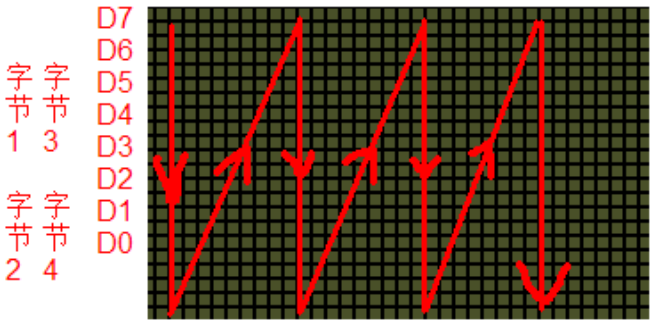


图 17.3.3 取模方式图解

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;
    u8 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
    chr=chr-' ';//得到偏移后的值
    for(t=0;t<csize;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，将 oled.c 保存，然后加入到 HARDWARE 组下。接下来我们在 oled.h 中输入如下代码：

```
#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   PDout(6)
#define OLED_RST PGout(15)
#define OLED_RS   PDout(3)
#define OLED_WR   PGout(14)
#define OLED_RD   PGout(13)
//PC0~7,作为数据线
#define DATAOUT(x) GPIOC->ODR=(GPIOC->ODR&0xff00)|(x&0x00FF); //输出
//使用 4 线串行接口时使用
#define OLED_SCLK PCout(0)
#define OLED_SDIN PCout(1)
#define OLED_CMD   0          //写命令
#define OLED_DATA   1         //写数据
//OLED 控制用函数
void OLED_WR_Byte(u8 dat,u8 cmd);
……                            //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。

保存好 oled.h 之后，我们就可以在主程序里面编写我们的应用层代码了，该部分代码如下：

```
int main(void)
{
    u8 t=0;
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    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 2019/9/17",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(48,48,t,16,1);//显示 ASCII 字符
        OLED_Refresh_Gram();
        t++;
        if(t>'~')t=' ';
        OLED_ShowNum(103,48,t,3,16);//显示 ASCII 字符的码值
        delay_ms(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，我们将使用精英 STM32F103 V1 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 LCD ID。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 STM32F1 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象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/7 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：

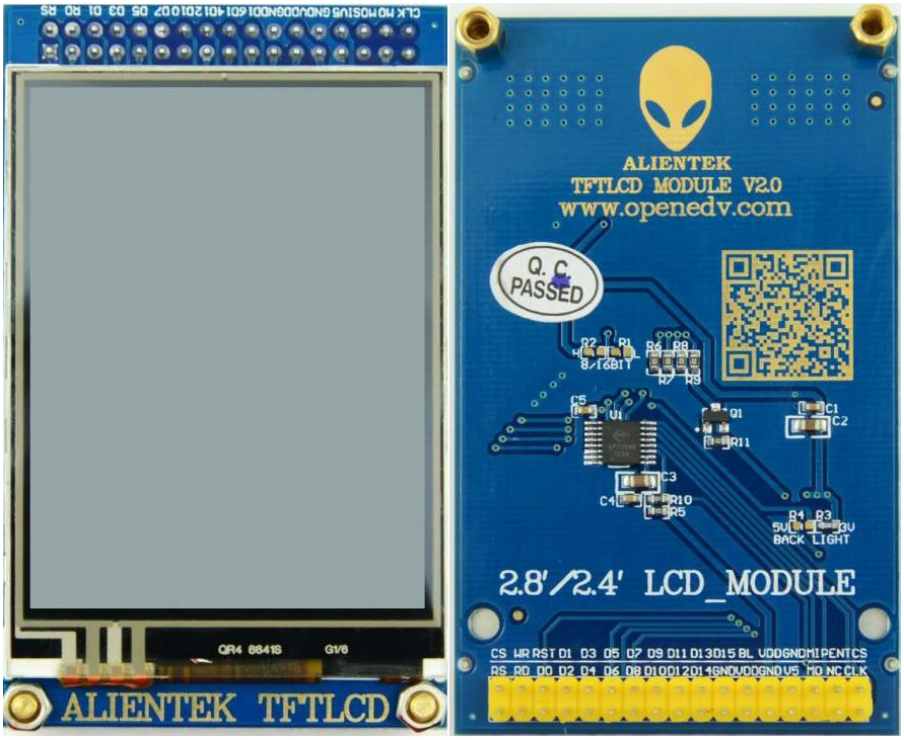


图 18.1.1.1 ALIENTEK 2.8 寸 TFTLCD 外观图

模块原理图如图 18.1.1.2 所示：

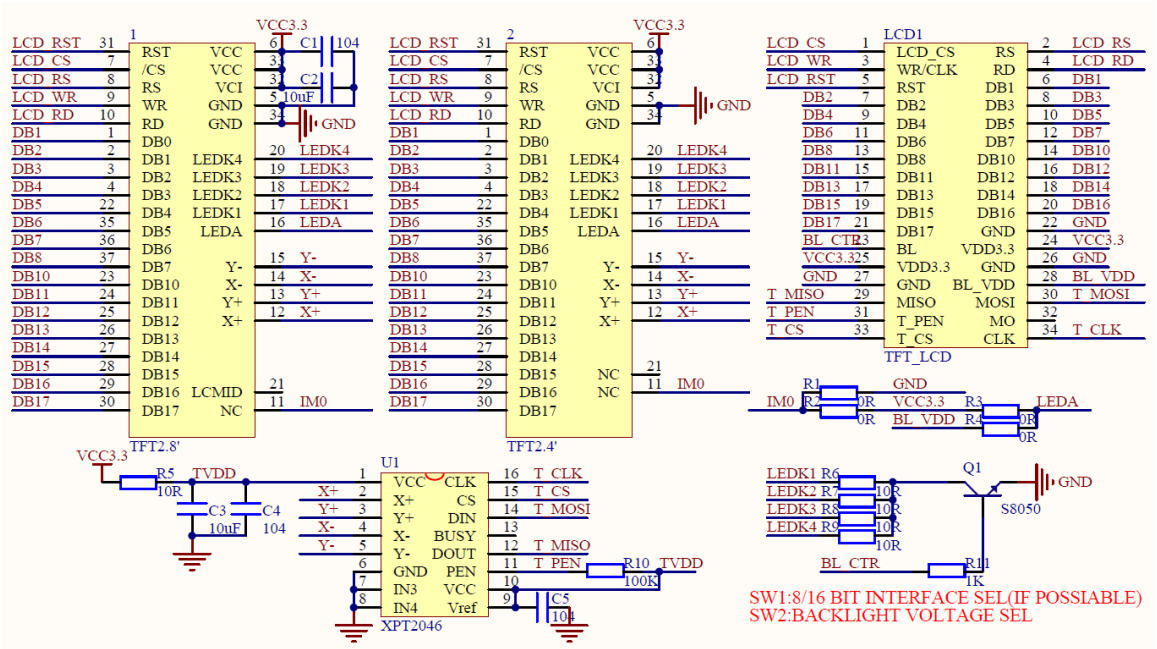


图 18.1.1.2　ALIENTEK 2.8 寸 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接（即接开发板的接口），接口定义如图 18.1.1.3 所示：

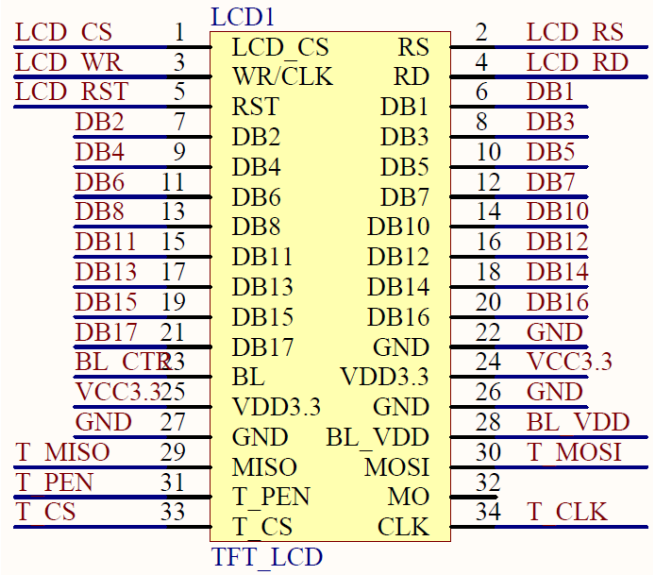


图 18.1.1.3　ALIENTEK 2.8 寸 TFTLCD 模块接口图

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 STM32F1 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 2.8/3.5/4.3/7 寸等不同尺寸的 TFTLCD 模块，其驱动芯片有很多种类型，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	SC
参数 2	1	1	↑	XX	SC7	SC6	SC5	SC4	SC3	SC2	SC1	SC0	
参数 3	1	1	↑	XX	EC15	EC14	EC13	EC12	EC11	EC10	EC9	EC8	EC
参数 4	1	1	↑	XX	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0	

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≤SC/EC≤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	SP
参数 2	1	1	↑	XX	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	
参数 3	1	1	↑	XX	EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	EP
参数 4	1	1	↑	XX	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0	

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≤SP/EP≤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												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：

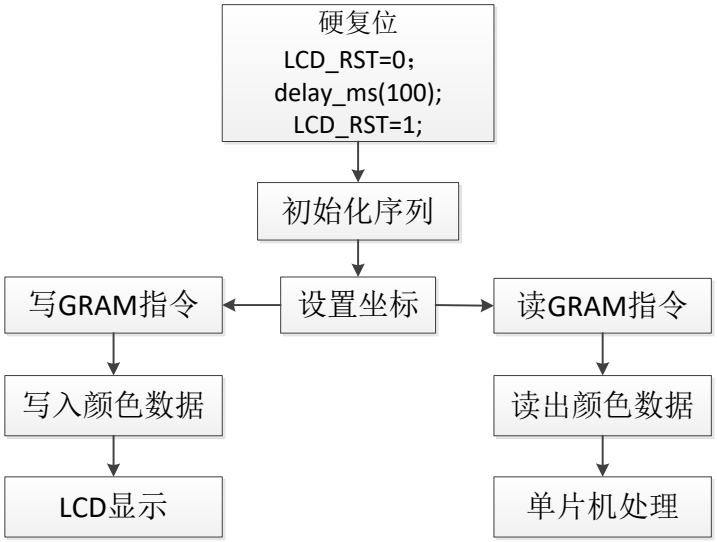


图 18.1.1.5 TFTLCD 使用流程

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（2.8 寸屏模块）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 STM32F1 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，因为精英 STM32F103 的 LCD 接口，将 TFTLCD 的 RST 同 STM32F1 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

大容量，且引脚数目在 100 脚及以上的 STM32F103 芯片都带有 FSMC 接口，ALIENTEK

精英 STM32F103 的主芯片为 STM32F103ZET6，是带有 FSMC 接口的。

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡连接，STM32F1 的 FSMC 接口支持包括 SRAM、NAND FLASH、NOR FLASH 和 PSRAM 等存储器。FSMC 的框图如图 18.1.2.1 所示：

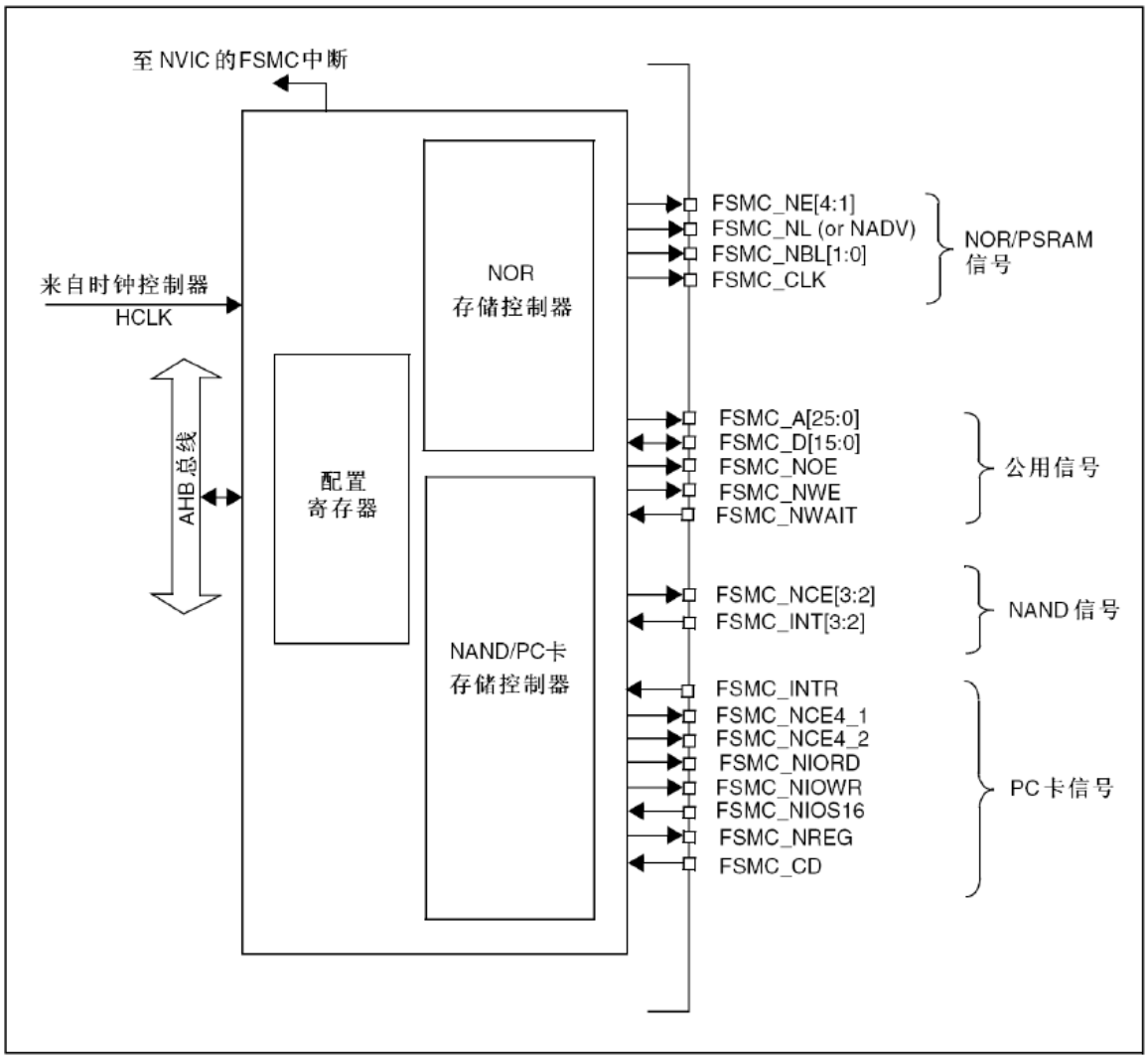


图 18.1.2.1 FSMC 框图

从上图我们可以看出，STM32 的 FSMC 将外部设备分为 3 类：NOR/PSRAM 设备、NAND 设备、PC 卡设备。他们共用地址数据总线等信号，他们具有不同的 CS 以区分不同的设备，比如本章我们用到的 TFTLCD 就是用的 FSMC_NE4 做片选，其实就是将 TFTLCD 当成 SRAM 来控制。

这里我们介绍下为什么可以把 TFTLCD 当成 SRAM 设备用：首先我们了解下外部 SRAM 的连接，外部 SRAM 的控制一般有：地址线（如 A0~A18）、数据线（如 D0~D15）、写信号（WE）、读信号（OE）、片选信号（CS），如果 SRAM 支持字节控制，那么还有 UB/LB 信号。而 TFTLCD 的信号我们在 18.1.1 节有介绍，包括：RS、D0~D15、WR、RD、CS、RST 和 BL 等，其中真正在操作 LCD 的时候需要用到的就只有：RS、D0~D15、WR、RD 和 CS。其操作时序和 SRAM 的控制完全类似，唯一不同就是 TFTLCD 有 RS 信号，但是没有地址信号。

TFTLCD 通过 RS 信号来决定传送的数据是数据还是命令，本质上可以理解为一个地址信号，比如我们把 RS 接在 A0 上面，那么当 FSMC 控制器写地址 0 的时候，会使得 A0 变为 0，

对 TFTLCD 来说，就是写命令。而 FSMC 写地址 1 的时候，A0 将会变为 1，对 TFTLCD 来说，就是写数据了。这样，就把数据和命令区分开了，他们其实就是对应 SRAM 操作的两个连续地址。当然 RS 也可以接在其他地址线上，精英 STM32 开发板是把 RS 连接在 A10 上面的。

STM32F1 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，STM32F1 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：

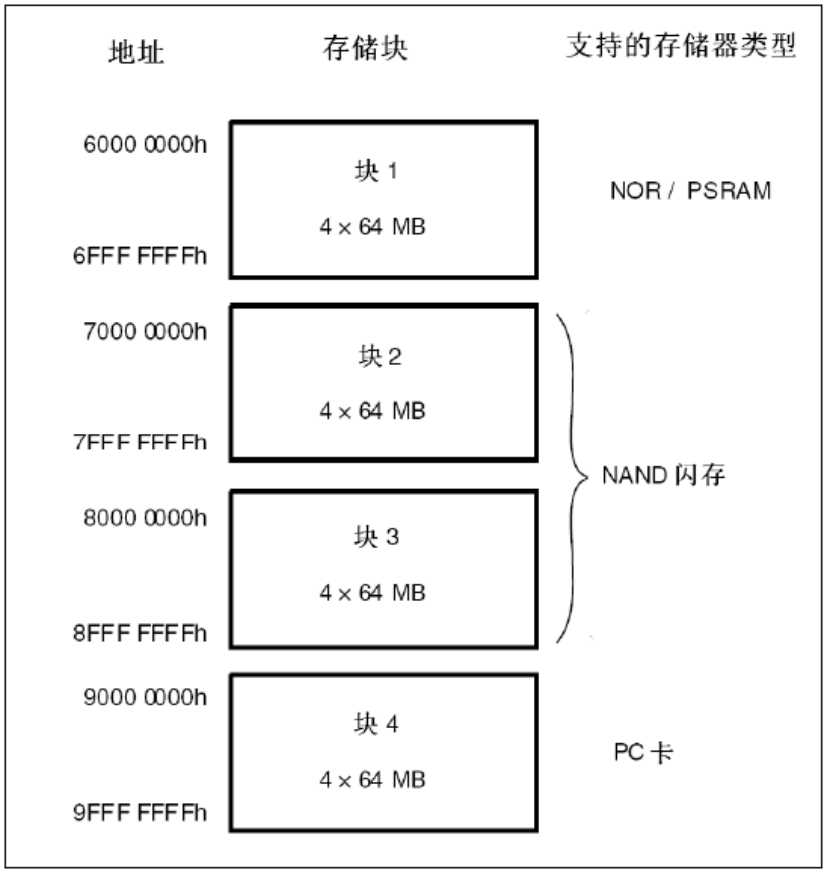


图 18.1.2.2 FSMC 存储块地址映像

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《STM32 中文参考手册》第 19 章（324 页）的相关介绍。

STM32F1 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_A [24:0]。

当 Bank1 接的是 8 位宽度存储器的时候：HADDR[25:0]→ FSMC_A [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，具体来说，比如地址：0X7E，对应二进制是：01111110，此时 FSMC_A6 是 0 而不是 1，因为要右移一位，这里请特别注意。

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。STM32F1 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 FSMC_PIO4
	Bank3	0X8000,0000~ 0X8FFF,FFFF		
	Bank4	0X9000,0000~ 0X9FFF,FFFF	PC Card	

表 18.1.2.2 FSMC 各 Bank 配置寄存器表

对于 NOR FLASH 控制器，主要是通过 FSMC_BCRx、FSMC_BTRx 和 FSMC_BWTRx 寄存器设置（其中 x=1~4，对应 4 个区）。通过这 3 个寄存器，可以设置 FSMC 访问外部存储器的时序参数，拓宽了可选用的外部存储器的速度范围。FSMC 的 NOR FLASH 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要的设置的时间参数有 2 个：

1，HCLK 与 FSMC_CLK 的分频系数(CLKDIV)，可以为 2～16 分频；

2，同步突发访问中获得第 1 个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。FSMC 综合了 SRAM / ROM、PSRAM 和 NOR Flash 产品的信号特点，定义了 4 种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数，如表 18.1.2.3 所列：

时序模型		简单描述	时间参数
异步	Mode1	SRAM/CRAM 时序	DATAST、ADDSET
	ModeA	SRAM/CRAM OE 选通型时序	DATAST、ADDSET
	Mode2/B	NOR FLASH 时序	DATAST、ADDSET
	ModeC	NOR FLASH OE 选通型时序	DATAST、ADDSET
	ModeD	延长地址保持时间的异步时序	DATAST、ADDSET、ADDHLK
同步突发		根据同步时钟 FSMC_CK 读取多个顺序单元的数据	CLKDIV、DATLAT

表 18.1.2.3 NOR FLASH 控制器支持的时序模型

在实际扩展时，根据选用存储器的特征确定时序模型，从而确定各时间参数与存储器读/写周期参数指标之间的计算关系；利用该计算关系和存储芯片数据手册中给定的参数指标，可计算出 FSMC 所需要的各时间参数，从而对时间参数寄存器进行合理的配置。

本章，我们使用异步模式 A（ModeA）方式来控制 TFTLCD，模式 A 的读操作时序如图 18.1.2.3 所示：

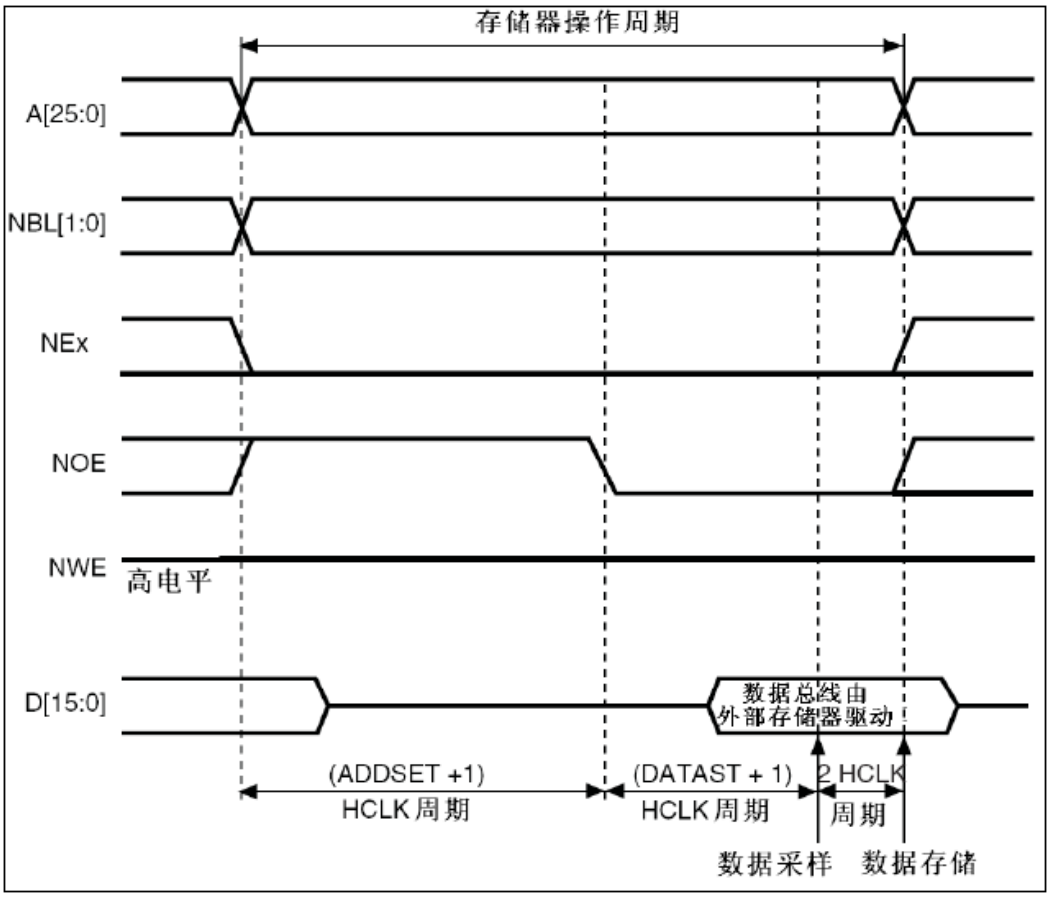


图 18.1.2.3 模式 A 读操作时序图

模式 A 支持独立的读写时序控制，这个对我们驱动 TFTLCD 来说非常有用，因为 TFTLCD 在读的时候，一般比较慢，而在写的时候可以比较快，如果读写用一样的时序，那么只能以读的时序为基准，从而导致写的速度变慢，或者在读数据的时候，重新配置 FSMC 的延时，在读操作完成的时候，再配置回写的时序，这样虽然也不会降低写的速度，但是频繁配置，比较麻烦。而如果有独立的读写时序控制，那么我们只要初始化的时候配置好，之后就不用再配置，既可以满足速度要求，又不需要频繁改配置。

模式 A 的写操作时序如图 18.1.2.4 所示：

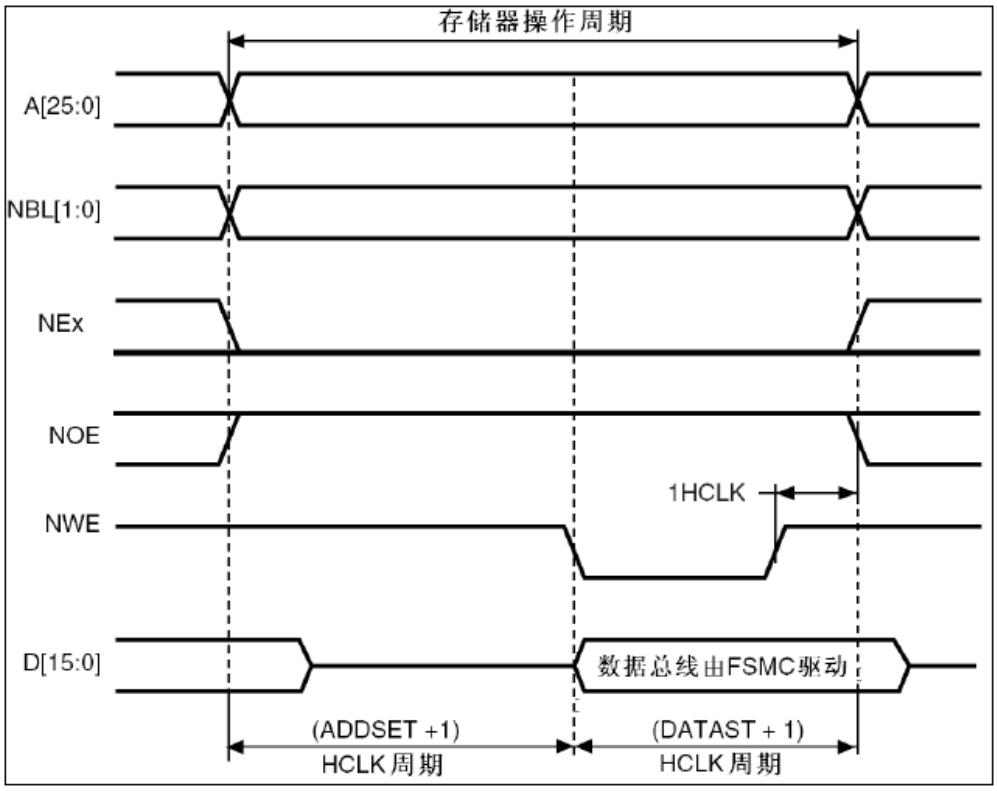


图 18.1.2.4 模式 A 写操作时序

图 18.1.2.3 和图 18.1.2.4 中的 ADDSET 与 DATAST，是通过不同的寄存器设置的，接下来我们讲解一下 Bank1 的几个控制寄存器

首先，我们介绍 SRAM/NOR 闪存片选控制寄存器：FSMC_BCRx（x=1~4），该寄存器各位描述如图 18.1.2.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留												CBURSTRW	保留				EXTMOD	WAITEN	WREN	WAITCFG	WRAPMOD	WAITPOL	BURSTEN	保留	FACCEN	MWID		MTYP		MUXEN	MBKEN
res												rw	res				rw	rw	rw	rw	rw	rw	rw	res	rw	rw		rw		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：

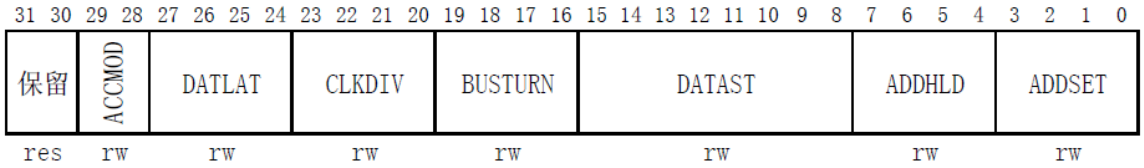


图 18.1.2.6 FSMC_BTRx 寄存器各位描述

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 13.8ns 左右（1/72Mhz），为了兼容其他屏，我们这里设置 DATAST 为 15，也就是 16 个 HCLK 周期，时间大约是 234ns（未计算数据存储的 2 个 HCLK 时间，对 9341 来说超频了，但是实际上是可以正常使用的）。

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，本来这里我们应该设置和 DATAST 一样，但是由于 STM32F103 FSMC 的性能问题，就算设置 ADDSET 为 0，RD 的高电平持续时间也达到了 190ns 以上，所有，我们这里可以设置 ADDSET 为较小的值，本章我们设置 ADDSET 为 1，即 2 个 HCLK 周期，实际 RD 高电平大于 200ns。

最后，我们再来看看 SRAM/NOR 闪写时序寄存器：FSMC_BWTRx（x=1~4），该寄存器各位描述如图 18.1.2.7 所示：

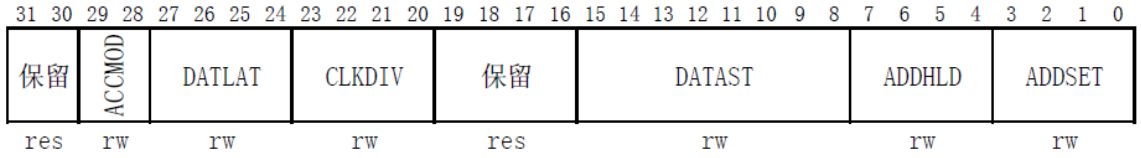


图 18.1.2.7 FSMC_BWTRx 寄存器各位描述

该寄存器在本章用作写操作时序控制寄存器，需要用到的设置同样是：ACCMOD、DATAST 和 ADDSET 这三个设置。这三个设置的方法同 FSMC_BTRx 一模一样，只是这里对应的是写操作的时序，ACCMOD 设置同 FSMC_BTRx 一模一样，同样是选择模式 A，另外 DATAST 和 ADDSET 则对应低电平和高电平持续时间，对 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 3，即 4 个 HCLK 周期，时间约为 55ns（因为 9320 等控制器，这个时间要求比较长，要 50ns）。然后 ADDSET（也存在性能问题）设置为 0，即 1 个 HCLK 周期，实际 WR 高电平时间大于 100ns。

至此，我们对 STM32F1 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 BTCR[8]寄存器组，他们的对应关系如下：

BTCR[0]对应 FSMC_BCR1，BTCR[1]对应 FSMC_BTR1
BTCR[2]对应 FSMC_BCR2，BTCR[3]对应 FSMC_BTR2
BTCR[4]对应 FSMC_BCR3，BTCR[5]对应 FSMC_BTR3
BTCR[6]对应 FSMC_BCR4，BTCR[7]对应 FSMC_BTR4
FSMC_BWTRx 则组合成 BWTR[7]，他们的对应关系如下：
BWTR[0]对应 FSMC_BWTR1，BWTR[2]对应 FSMC_BWTR2，
BWTR[4]对应 FSMC_BWTR3，BWTR[6]对应 FSMC_BWTR4，
BWTR[1]、BWTR[3]和 BWTR[5]保留，没有用到。

通过上面的讲解，通过对 FSMC 相关的寄存器的描述，大家对 FSMC 的原理有了一个初步的认识，如果还不熟悉的朋友，请一定要搜索网络资料理解 FSMC 的原理。只有理解了原理，使用库函数才可以得心应手。那么在库函数中是怎么实现 FSMC 的配置的呢？FSMC_BCRx，FSMC_BTRx 寄存器在库函数是通过什么函数来配置的呢？下面我们来讲解一下使用 FMC 接口驱动 LCD（SRAM）相关的库函数操作过程。

1） 使能 FSMC 和 GPIO 时钟，初始化 IO 口配置，设置映射关系

这个步骤在前面实验已多次讲解。这里我们主要列出 FSMC 时钟使能方法：

```
__HAL_RCC_FSMC_CLK_ENABLE();                    //使能 FSMC 时钟
```

对于 IO 配置，调用函数 HAL_GPIO_Init 配置即可，具体 请参考实验源码。

2） 初始化 FMC 接口读写时序参数，初始化 LCD（SRAM）控制接口

根据前面的讲解，我们把 LCD 当 SRAM 使用，连接在 FMC 接口之上，所以我们要初始化 FMC 读写时序参数以及 LCD 数据接口，也就是初始化三个寄存器 FMC_BCRx，FMC_BTRx 和 FMC_BWTRx。HAL 库提供了 SRAM 初始化函数 HAL_SRAM_Init，该函数声明如下：

```
HAL_StatusTypeDef HAL_SRAM_Init(SRAM_HandleTypeDef *hsram,
                                FSMC_NORSRAM_TimingTypeDef *Timing,
                                FSMC_NORSRAM_TimingTypeDef *ExtTiming);
```

该函数有三个入口参数，首先我们来看看第一个入口参数 hsram，它是 SRAM_HandleTypeDef 结构体指针类型，该参数用来初始化当 FMC 接口当 SRAM 使用时的控制接口参数。结构体 SRAM_HandleTypeDef 定义如下：

```
typedef struct
{
  FMC_NORSRAM_TypeDef                 *Instance;
  FMC_NORSRAM_EXTENDED_TypeDef        *Extended;
  FMC_NORSRAM_InitTypeDef             Init;
  HAL_LockTypeDef                     Lock;
  __IO HAL_SRAM_StateTypeDef          State;
  DMA_HandleTypeDef                   *hdma;
}SRAM_HandleTypeDef;
```

成员变量 Instance 和成员变量 Extended 实际上是用来在指定的时序模型下，寄存器基地址和扩展模式寄存器基地址。这个怎么理解呢，本实验我们使用异步模式 A（ModeA）方式来控制 TFTLCD，使用的存储块是 Bank1，所以寄存器基地址 Instance 我们直接写 FMC_Bank1 即可，当然，HAL 库定义好了宏定义 FSMC_NORSRAM_DEVICE，也就是如果是 SRAM 设备，直接填写这个宏定义标识符即可。因为我们要配置的读写时序是不一样的，也就是我们前面讲解的 FSMC_BCRx 寄存器的 EXTMOD 位我们会配置为 1 允许读写不同的时序，所以我们这里还要指定写操作时序寄存器地址，也就是通过参数 Extended 来指定的，这里我们设置为

FSMC_Bank1E 即可，同样 MDK 定义好了宏定义标识符 FSMC_NORSRAM_EXTENDED_DEVICE，所以这里我们填写这个宏定义标识符也是一样的。对于写时序参数配置，是在函数 HAL_SRAM_Init 的第三个参数 ExtTiming 来配置的，这个我们后面会讲解。

成员变量 Init 是 FSMC_NORSRAM_InitTypeDef 结构体指针类型，改变量才是真正用来设置 SRAM 控制接口参数的。我们接下来看看这个结构体定义：

```
typedef struct
{
  uint32_t NSBank;              //存储区块号
  uint32_t DataAddressMux;      //地址/数据复用使能
  uint32_t MemoryType;          //存储器类型
  uint32_t MemoryDataWidth;     //存储器数据宽度
  uint32_t BurstAccessMode;
  uint32_t WaitSignalPolarity;
  uint32_t WrapMode;
  uint32_t WaitSignalActive;
  uint32_t WriteOperation;      //存储器写使能
  uint32_t WaitSignal;
  uint32_t ExtendedMode;        //是否使能扩展模式
  uint32_t AsynchronousWait;
  uint32_t WriteBurst;
  uint32_t PageSize;
} FSMC_NORSRAM_InitTypeDef;
```

NSBank 用来指定使用到的存储块区号，前面讲过，我们是使用的存储块区号 1，所以选择值为 FSMC_NORSRAM_BANK1。DataAddressMux 用来设置是否使能地址/数据复用,该变量仅对 NOR/PSRAM 有效，所以这里我们选择不使能地址/数据复用值 FSMC_DATA_ADDRESS_MUX_DISABLE 即可。MemoryType 用来设置存储器类型，这里我们把 LCD 当 SRAM 使用，所以设置为 FSMC_MEMORY_TYPE_SRAM 即可。MemoryDataWidth 用来设置存储器数据总线宽度，可选 8 位还是 16 位，这里我们选择 16 位数据宽度 FSMC_NORSRAM_MEM_BUS_WIDTH_16。WriteOperation 用来设置存储器写使能，也就是是否允许写入。毫无疑问我们会进行存储器写操作，所以这里设置为 FSMC_WRITE_OPERATION_ENABLE。ExtendedMode 用来设置是否使能扩展模式，也就是是否允许读写使用不同时序，前面讲解过本实验读写采用不同时序，所以设置值为使能值 FSMC_EXTENDED_MODE_ENABLE。ContinuousClock 用来设置启用/禁止 FSMC 时钟输出到外部存储设备，这里仅当使用 FMC_BCR1 寄存器的时候需要启用，启用值为 FSMC_CONTINUOUS_CLOCK_SYNC_ASYNC。其他参数 WriteBurst，BurstAccessMode，WaitSignalPolarity，WrapMode，WaitSignalActive，WaitSignal，AsynchronousWait 等是用在突发访问和异步时序情况下，这里我们不做过多讲解。

成员变量 Lock 和 State 是 HAL 库处理状态标识变量。这里就不做过多讲解。

成员变量 hdma 在使用 DMA 时候才使用，这里就先不讲解了。

函数 HAL_SRAM_Init 的第一个入口参数就给大家讲解到这里。

接下来看看后面 2 个参数 Timing 和 ExtTiming，它们都是 FSMC_NORSRAM_TimingTypeDef 结构体指针类型，分别用来设置 FMC 接口读和写时序，主要涉及地址建立保持时间，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 Timing 和 ExtTiming 设置了不同的值。

FMC_NORSRAM_TimingTypeDef 结构体定义如下：

```
typedef struct
{
  uint32_t AddressSetupTime;          //地址建立时间
  uint32_t AddressHoldTime;           //地址保持时间
  uint32_t DataSetupTime;             //数据简历时间
  uint32_t BusTurnAroundDuration;     //总线周转阶段的持续时间
  uint32_t CLKDivision;               //CLK 时钟输出信号的周期
  uint32_t DataLatency;               //同步突发 NOR FLASH 的数据延迟
  uint32_t AccessMode;                //异步模式配置
}FSMC_NORSRAM_TimingTypeDef;
```

成员变量 AddressSetupTime 用来设置地址建立时间。AddressHoldTime 用来设置地址保持时间。DataSetupTime 用来设置数据建立时间。BusTurnAroundDuration 用来配置总线周转阶段的持续时间。CLKDivision 用来配置 CLK 时钟输出信号的周期，以 HCLK 周期数表示。DataLatency 用来设置同步突发 NOR FLASH 的数据延迟。AccessMode 用来设置异步模式，取值范围为 FSMC_ACCESS_MODE_A，FSMC_ACCESS_MODE_B, FMC_ACCESS_MODE_C 和 FSMC_ACCESS_MODE_D，这里我们用是异步模式 A，所以取值为 FSMC_ACCESS_MODE_A。

HAL_SRAM_Init 函数各个入口参数含义和配置就给大家讲解到这里。

和其他外设一样，HAL 库也提供了 SRAM 的初始化 MSP 回调函数，函数声明如下：

```
void HAL_SRAM_MspInit(SRAM_HandleTypeDef *hsram) ;
```

关于 MSP 函数的使用方法相信大家已经非常熟悉。该函数内部一般用来使能时钟以及初始化 IO 口这些与 MCU 相关的步骤。

前面我们讲解过，FSMC 接口支持多种存储器，包括 SDRAM，NOR，NAND 和 PC CARD 等。HAL 库为每种支持的存储器类型都定义了一个独立的 HAL 库文件，并且在文件中定义了独立的初始化函数。同时还提供了独立的初始化函数 HAL_SDRAM_Init，这里我们就列出几种存储器的初始化函数：

```
HAL_SDRAM_Init();//SDRAM 初始化函数,省略入口参数
HAL_NOR_Init();//NOR 初始化函数,省略入口参数
HAL_NAND_Init();//NAND 初始化函数,省略入口参数
```

3）存储区使能

实际上，当我们调用了存储器初始化函数之后，相应的使用到的存储区就已经被使能。SRAM 存储区使能方法为：

```
__FSMC_NORSRAM_ENABLE(FMC_Bank1，FMC_NORSRAM_BANK1);
```

18.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） TFTLCD 模块

TFTLCD 模块的电路见图 18.1.1.2，这里我们介绍 TFTLCD 模块与 ALIETEK 精英 STM32F103 的连接，精英 STM32F103 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：

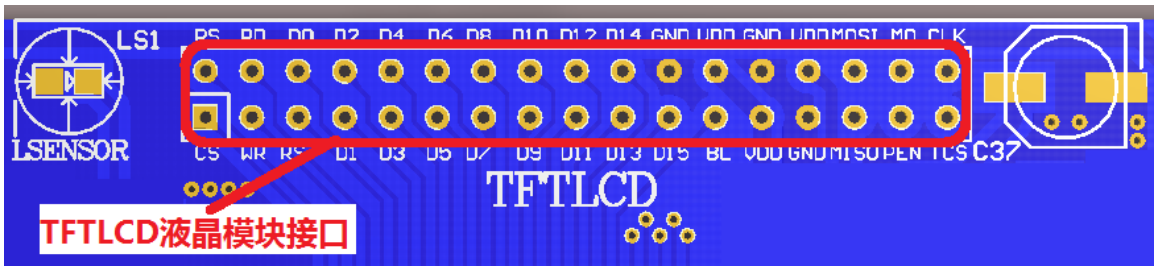


图 18.2.1 TFTLCD 与开发板连接示意图

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 模块与精英 STM32F103 的 IO 口对应关系如下：

LCD_BL(背光控制)对应 PB0;

LCD_CS 对应 PG12 即 FSMC_NE4;

LCD _RS 对应 PG0 即 FSMC_A10;

LCD _WR 对应 PD5 即 FSMC_NWE;

LCD _RD 对应 PD4 即 FSMC_NOE;

LCD _D[15:0]则直接连接在 FSMC_D15~FSMC_D0;

这些线的连接，精英 STM32F103 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（4.3 寸 TFTLCD 模块）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

软件设计我们依旧在之前的工程上面增加，不过没用到 OLED，所以先去掉 oled.c（注意，

此时 HARDWARE 组仅剩：led.c），然后在 HARDWARE 文件夹下新建一个 LCD 的文件夹。然后打开 USER 文件夹下的工程，新建一个 ILI93xx.c 的文件和 lcd.h 的头文件，保存在 LCD 文件夹下，并将 LCD 文件夹加入头文件包含路径。在 ILI93xx.c 里面要输入的代码比较多，我们这里就不贴出来了，只针对几个重要的函数进行讲解。

本实验，我们用到 FSMC 驱动 LCD，通过前面的介绍，我们知道 TFTLCD 的 RS 接在 FSMC 的 A10 上面，CS 接在 FSMC_NE4 上，并且是 16 位数据总线。即我们使用的是 FSMC 存储器 1 的第 4 区，我们定义如下 LCD 操作结构体（在 lcd.h 里面定义）：

```
//LCD 操作结构体
typedef struct
{
    vu16 LCD_REG;
    vu16 LCD_RAM;
} LCD_TypeDef;
//使用 NOR/SRAM 的  Bank1.sector4,地址位 HADDR[27,26]=11 A10 作为数据命令区分线
//注意设置时 STM32 内部会右移一位对其!
#define LCD_BASE        ((u32)(0x6C000000 | 0x000007FE))
#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 0X000007FE，则是 A10 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 A10 为例，7FE 转换成二进制就是：111 1111 1110，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：A10:A0=011 1111 1111，此时 A10 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 7FE+0X02），那么：A10:A0=100 0000 0000，此时 A10 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 0X6C00,07FE，对应 A10 的状态为 0(即 RS=0)，而 LCD-> LCD_RAM 的地址就是 0X6C00,0800（结构体地址自增），对应 A10 的状态为 1（即 RS=1）。

所以，有了这个定义，当我们要往 LCD 写命令/数据的时候，可以这样写：

```
LCD->LCD_REG=CMD;   //写命令
LCD->LCD_RAM=DATA;  //写数据
```

而读的时候反过来操作就可以了，如下所示：

```
CMD= LCD->LCD_REG;   //读 LCD 寄存器
DATA = LCD->LCD_RAM; //读 LCD 数据
```

这其中，CS、WR、RD 和 IO 口方向都是由 FSMC 控制，不需要我们手动设置了。接下来，我们先介绍一下 lcd.h 里面的另一个重要结构体:

```
//LCD 重要参数集
typedef struct
{
    u16 width;          //LCD  宽度
    u16 height;         //LCD  高度
    u16 id;             //LCD ID
    u8  dir;            //横屏还是竖屏控制：0，竖屏；1，横屏。
    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 ILI93xx.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(u16 regval)
{
    LCD->LCD_REG=regval;    //写入要写的寄存器序号
}
//写 LCD 数据
//data:要写入的值
void LCD_WR_DATA(u16 data)
{
    LCD->LCD_RAM=data;
}
//读 LCD 数据
//返回值:读到的值
u16 LCD_RD_DATA(void)
{
    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(u16 LCD_Reg, u16 LCD_RegValue)
{
    LCD->LCD_REG = LCD_Reg;          //写入要写的寄存器序号
    LCD->LCD_RAM = LCD_RegValue;  //写入数据
}
//读寄存器
//LCD_Reg:寄存器地址
//返回值:读到的数据
u16 LCD_ReadReg(u16 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。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。

第八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==0X1963)
    {
        if(lcddev.dir==0)//x 坐标需要变换
        {
            Xpos=lcddev.width-1-Xpos;
            LCD_WR_REG(lcddev.setxcmd);
            LCD_WR_DATA(0);LCD_WR_DATA(0);
```

```
            LCD_WR_DATA(Xpos>>8);LCD_WR_DATA(Xpos&0XFF);
        }else
        {
            LCD_WR_REG(lcddev.setxcmd);
            LCD_WR_DATA(Xpos>>8);LCD_WR_DATA(Xpos&0XFF);
            LCD_WR_DATA((lcddev.width-1)>>8);
            LCD_WR_DATA((lcddev.width-1)&0XFF);
        }
        LCD_WR_REG(lcddev.setycmd);
        LCD_WR_DATA(Ypos>>8);LCD_WR_DATA(Ypos&0XFF);
        LCD_WR_DATA((lcddev.height-1)>>8);LCD_WR_DATA((lcddev.height-1)&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);
    }
}
```

该函数非常重要，该函数实现了将 LCD 的当前操作点设置到指定坐标(x,y)，有了该函数，我们就可以在液晶上任意作图了。这里面的 lcddev.setxcmd、lcddev.setycmd、lcddev.width、lcddev.height 等指令/参数都是在 LCD_Display_Dir 函数里面初始化的，该函数根据 lcddev.id 的不同，执行不同的设置，由于篇幅所限，这里就不列出来介绍了，请大家参考本例程源码。另外，因为 9341/5310/6804/1963/5510 等的设置同其他屏有些不太一样，所以进行了区别对待。

接下来我们介绍第九个函数：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，第十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 320×240×2 个字节来存储颜色值，也就是也需要 150K 字节，这对任何一款单片机来说，都不是一个小数目了。而且我们在图形叠加的时候，可以先读回原来的值，然后写入新的值，在完成叠加后，我们又恢复原来的值。这样在做一些简单菜单的时候，是很有用的。这里我们读取 TFTLCD 模块数据的函数为 LCD_ReadPoint，该函数直接返回读到的 GRAM 值。该函数使用之前要先设置读取的 GRAM 地址，通过 LCD_SetCursor 函数来实现。LCD_ReadPoint 的代码如下：

```
//读取个某点的颜色值
//x,y:坐标
//返回值:此点的颜色
u16 LCD_ReadPoint(u16 x,u16 y)
{
    u16 r=0,g=0,b=0;
    if(x>=lcddev.width||y>=lcddev.height)return 0;    //超过了范围,直接返回
    LCD_SetCursor(x,y);
    if(lcddev.id==0X9341||lcddev.id==0X6804||lcddev.id==0X5310||lcddev.id==0X1963)
    LCD_WR_REG(0X2E);//9341/6804/3510/1963 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(0X22);                          //其他 IC 发送读 GRAM 指令
    if(lcddev.id==0X9320)opt_delay(2);              //FOR 9320,延时 2us
    r=LCD_RD_DATA();                                //dummy Read
    if(lcddev.id==0X1963)return r;                  //1963 直接读就可以
    opt_delay(2);
    r=LCD_RD_DATA();                                //实际坐标颜色
    if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510)//这些 LCD 要分 2 次读出
    {
        opt_delay(2);
        b=LCD_RD_DATA();
        g=r&0XFF;//对于 9341/5310/5510,第一次读取的是 RG 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。

第十一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
    num=num-' ';  //ASCII 字库从空格开始取模，所以-' '即可得到对应字符的字库（点阵）
    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：

```
//初始化 lcd
//该初始化函数可以初始化各种型号的 LCD(详见本.c 文件最前面的描述)
void LCD_Init(void)
{
    GPIO_InitTypeDef GPIO_Iniure;
    FSMC_NORSRAM_TimingTypeDef FSMC_ReadWriteTim;
    FSMC_NORSRAM_TimingTypeDef FSMC_WriteTim;

    __HAL_RCC_GPIOB_CLK_ENABLE();              //开启 GPIOB 时钟
    GPIO_Iniure.Pin=GPIO_PIN_0;                //PB0,背光控制
    GPIO_Iniure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Iniure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Iniure.Speed=GPIO_SPEED_FREQ_HIGH;  //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Iniure);

    TFTSRAM_Handler.Instance=FSMC_NORSRAM_DEVICE;
    TFTSRAM_Handler.Extended=FSMC_NORSRAM_EXTENDED_DEVICE;

    TFTSRAM_Handler.Init.NSBank=FSMC_NORSRAM_BANK4;      //使用 NE4
    TFTSRAM_Handler.Init.DataAddressMux=FSMC_DATA_ADDRESS_MUX_DISABLE;
                                                        //地址/数据线不复用
    TFTSRAM_Handler.Init.MemoryType=FSMC_MEMORY_TYPE_SRAM;  //SRAM
TFTSRAM_Handler.Init.MemoryDataWidth=FSMC_NORSRAM_MEM_BUS_WIDTH_16;
                                                        //16 位数据宽度
TFTSRAM_Handler.Init.BurstAccessMode=FSMC_BURST_ACCESS_MODE_DISABLE;
                    //是否使能突发访问,仅对同步突发存储器有效,此处未用到
    TFTSRAM_Handler.Init.WaitSignalPolarity=FSMC_WAIT_SIGNAL_POLARITY_LOW;
                    //等待信号的极性,仅在突发模式访问下有用
    TFTSRAM_Handler.Init.WaitSignalActive=FSMC_WAIT_TIMING_BEFORE_WS;
                  //存储器是在等待周期之前的一个时钟周期还是等待周期期间使能 NWAIT
    TFTSRAM_Handler.Init.WriteOperation=FSMC_WRITE_OPERATION_ENABLE;
                //存储器写使能
    TFTSRAM_Handler.Init.WaitSignal=FSMC_WAIT_SIGNAL_DISABLE;
              //等待使能位,此处未用到
    TFTSRAM_Handler.Init.ExtendedMode=FSMC_EXTENDED_MODE_ENABLE;
              //读写使用不同的时序
TFTSRAM_Handler.Init.AsynchronousWait=FSMC_ASYNCHRONOUS_WAIT_DISABLE;
            //是否使能同步传输模式下的等待信号,此处未用到
    TFTSRAM_Handler.Init.WriteBurst=FSMC_WRITE_BURST_DISABLE;
              //禁止突发写
    //FMC 读时序控制寄存器
    FSMC_ReadWriteTim.AddressSetupTime=0x06;
```

```
    //地址建立时间（ADDSET）为 7 个 HCLK 13.8ns*7=96.6ns
FSMC_ReadWriteTim.AddressHoldTime=0;
FSMC_ReadWriteTim.DataSetupTime=26;
    //数据保存时间为 27 个 HCLK      =13.8*27=372.6ns
FSMC_ReadWriteTim.AccessMode=FSMC_ACCESS_MODE_A;//模式 A
//FMC 写时序控制寄存器
FSMC_WriteTim.BusTurnAroundDuration=0;//总线周转阶段持续时间为 0，此变量不
    //赋值的话会莫名其妙的自动修改为 4。导致程序运行正常
FSMC_WriteTim.AddressSetupTime=3;
    //地址建立时间（ADDSET）为 4 个 HCLK =55.2ns
FSMC_WriteTim.AddressHoldTime=0;
FSMC_WriteTim.DataSetupTime=0x06;
      //数据保存时间为 13.8ns*7 个 HCLK=96.6ns
FSMC_WriteTim.AccessMode=FSMC_ACCESS_MODE_A;      //模式 A
HAL_SRAM_Init(&TFTSRAM_Handler,&FSMC_ReadWriteTim,&FSMC_WriteTim);
delay_ms(50); // delay 50 ms
//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!=0X5510)             //也不是 NT5510,尝试看看是不是 SSD1963
```

```
                {
                    LCD_WR_REG(0XA1);
                    lcddev.id=LCD_RD_DATA();
                    lcddev.id=LCD_RD_DATA();      //读回 0X57
                    lcddev.id<<=8;
                    lcddev.id|=LCD_RD_DATA();     //读回 0X61
                    if(lcddev.id==0X5761)lcddev.id=0X1963;
                        //SSD1963 读回的 ID 是 5761H,为方便区分,我们强制设置为 1963
                }
            }
        }
        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
    }
    //初始化完成以后,提速
    if(lcddev.id==0X9341||lcddev.id==0X5310||lcddev.id==0X5510||lcddev.id==0X1963)
        {
            //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]|=1<<0;//地址建立时间(ADDSET)为 2 个 HCLK =28ns
            FSMC_Bank1E->BWTR[6]|=1<<8;
             //数据保存时间(DATAST)为 13.8ns*2 个 HCLK=28ns
        }
    LCD_Display_Dir(0);             //默认为竖屏显示
    LCD_LED=1;                      //点亮背光
    LCD_Clear(WHITE);
}
```

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。

SRAM 初始化 MSP 回调函数 HAL_SRAM_MspInit 内容比较简单，主要是进行时钟使能以及 IO 口映射配置，这里就不做过多讲解。

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

    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    LCD_Init();                             //初始化 LCD FSMC 接口
    POINT_COLOR=RED;                        //画笔颜色：红色
    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," ELITE STM32 ^_^");
        LCD_ShowString(30,70,200,16,16,"TFTLCD TEST");
        LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(30,110,200,16,16,lcd_id);        //显示 LCD ID

        LCD_ShowString(30,130,200,12,12," 2019/9/27");
        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

将程序下载到精英 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块的显示如图 18.4.1 所示：

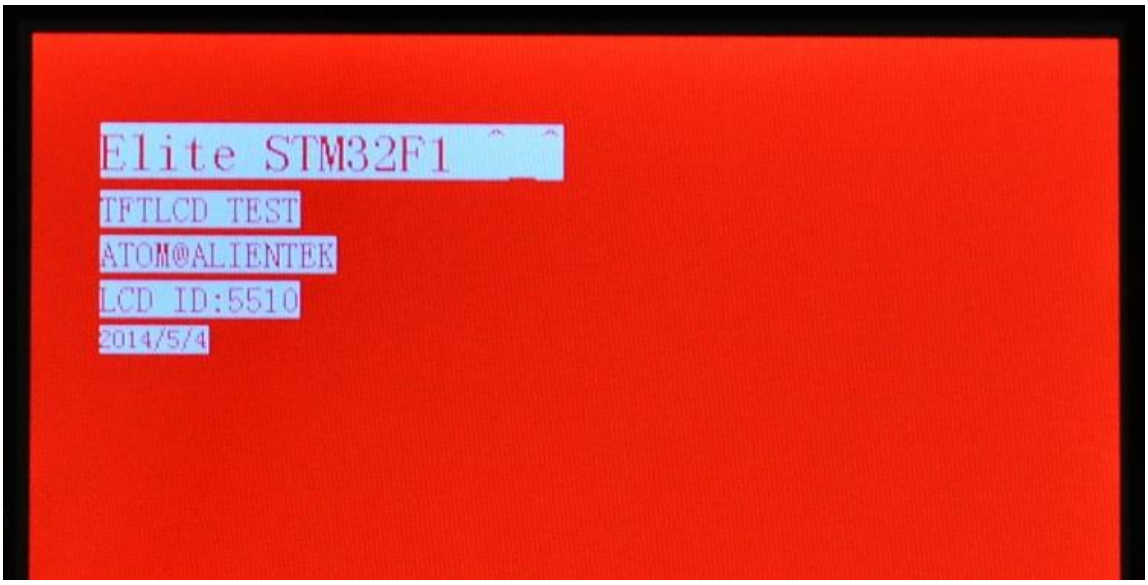


图 18.4.1 TFTLCD 显示效果图

我们可以看到屏幕的背景是不停切换的，同时 DS0 不停的闪烁，证明我们的代码被正确的执行了，达到了我们预期的目的，实现了 TFTLCD 的驱动，以及字符的显示。另外，本例程除了不支持 CPLD 方案的 7 寸屏模块，其余所有的 ALIENTEK TFTLCD 模块都可以支持，直接插上去即可使用。

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：

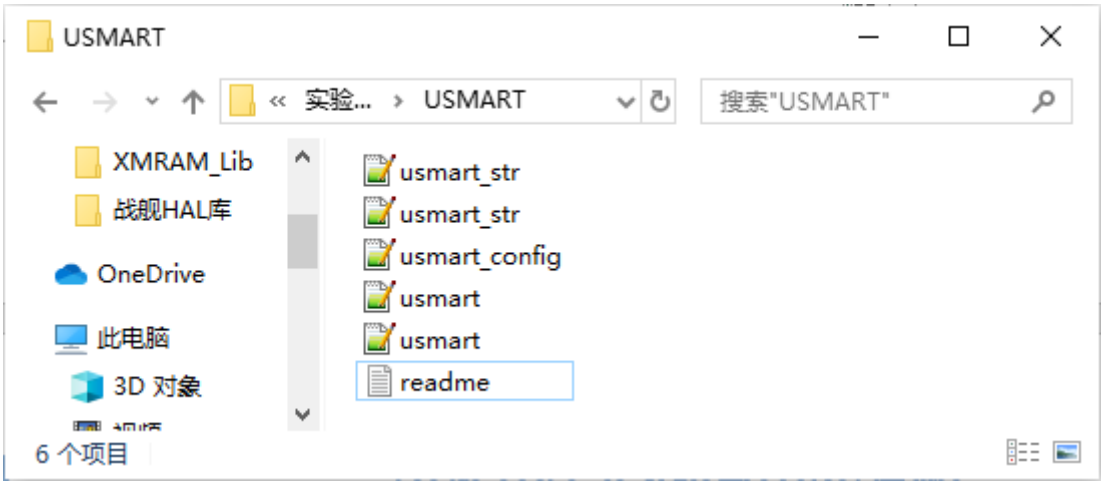


图 19.1.1 USMART 组件代码

其中 redeme.txt 是一个说明文件，不参与编译。其他五个文件，usmart.c 负责与外部互交等。usmat_str.c 主要负责命令和参数解析。usmart_config.c 主要由用户添加需要由 usmart 管理的函数。

usmart.h 和 usmart_str.h 是两个头文件，其中 usmart.h 里面含有几个用户配置宏定义，可以用来配置 usmart 的功能及总参数长度(直接和 SRAM 占用挂钩)、是否使能定时器扫描、是否使用读写函数等。

USMART 的移植，只需要实现 5 个函数。其中 4 个函数都在 usmart.c 里面，另外一个是串口接收函数，必须由用户自己实现，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)
{
    __HAL_TIM_CLEAR_FLAG(&TIM4_Handler,TIM_FLAG_UPDATE);
                                                    //清除中断标志位
    __HAL_TIM_SET_AUTORELOAD(&TIM4_Handler,0XFFFF);//将重装载值设置到最大
    __HAL_TIM_SET_COUNTER(&TIM4_Handler,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(__HAL_TIM_GET_FLAG(&TIM4_Handler,TIM_FLAG_UPDATE)==SET)
                                                //在运行期间,产生了定时器溢出
    {
        usmart_dev.runtime+=0XFFFF;
    }
    usmart_dev.runtime+=__HAL_TIM_GET_COUNTER(&TIM4_Handler);
    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 STM32F1 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*2*0.1ms=13.1 秒。也就是说，如果函数执行时间超过 13.1 秒，那么计时将不准确。

最后一个是 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组大小，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_NOFUNCFIND:
                            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

打开上一章的工程，复制 USMART 文件夹到本工程文件夹下面，如图 19.3.1 所示：



图 19.3.1 复制 USMART 文件夹到工程文件夹下

接着，我们打开工程，并新建 USMART 组，添加 USMART 组件代码，同时把 USMART 文件夹添加到头文件包含路径，在主函数里面加入 include “usmart.h” 如图 19.3.2 所示：

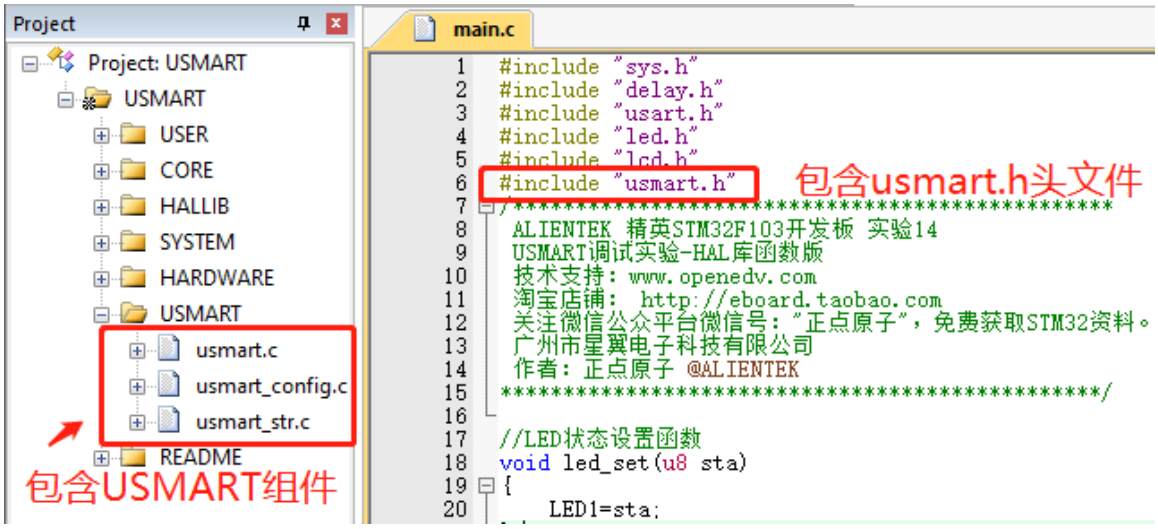


图 19.3.2 添加 USMART 组件代码

由于 USMART 默认提供了 STM32F1 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：

main.c | usmart.c | usmart_config.c | usmart.h

```
#include "usmart.h"
#include "usmart_str.h"
////////////////////////////////用户配置区///////////////////////////////////////////////
//这下面要包含所用到的函数所申明的头文件(用户自己添加)
#include "delay.h"
#include "sys.h"
#include "lcd.h"

extern void led_set(u8 sta);
extern void test_fun(void(*ledset)(u8),u8 sta);
//函数名列表初始化(用户自己添加)
//用户直接在这里输入要执行的函数名及其查找串
struct _m_usmart_nametab usmart_nametab[]=
{
#if USMART_USE_WRFUNS==1    //如果使能了读写操作
    (void*)read_addr,"u32 read_addr(u32 addr)",
    (void*)write_addr,"void write_addr(u32 addr,u32 val)",
#endif
    (void*)delay_ms,"void delay_ms(u16 nms)",
    (void*)delay_us,"void delay_us(u32 nus)",
    (void*)LCD_Clear,"void LCD_Clear(u16 Color)",
    (void*)LCD_Fill,"void LCD_Fill(u16 xsta,u16 ysta,u16 xend,u16 yend,u16 color)",
    (void*)LCD_DrawLine,"void LCD_DrawLine(u16 x1, u16 y1, u16 x2, u16 y2)",
    (void*)LCD_DrawRectangle,"void LCD_DrawRectangle(u16 x1, u16 y1, u16 x2, u16 y2)",
    (void*)LCD_Draw_Circle,"void Draw_Circle(u16 x0,u16 y0,u8 r)",
    (void*)LCD_ShowNum,"void LCD_ShowNum(u16 x,u16 y,u32 num,u8 len,u8 size)",
    (void*)LCD_ShowString,"void LCD_ShowString(u16 x,u16 y,u16 width,u16 height,u8 size,u8 *p)",
    (void*)LCD_Fast_DrawPoint,"void LCD_Fast_DrawPoint(u16 x,u16 y,u16 color)",
    (void*)LCD_ReadPoint,"u16 LCD_ReadPoint(u16 x,u16 y)",
    (void*)LCD_Display_Dir,"void LCD_Display_Dir(u8 dir)",
    (void*)LCD_ShowxNum,"void LCD_ShowxNum(u16 x,u16 y,u32 num,u8 len,u8 size,u8 mode)",

    (void*)led_set,"void led_set(u8 sta)",
    (void*)test_fun,"void test_fun(void(*ledset)(u8),u8 sta)",
};
///////////////////////////////////END///////////////////////////////////////////////////
```

函数所在头文件添加区

用户函数添加区

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：

```
#include "usmart.h"
#include "usmart_str.h"
////////////////////////////////用户配置区///////////////////////////////////////////////
//这下面要包含所用到的函数所申明的头文件(用户自己添加)
#include "delay.h"
#include "sys.h"
#include "lcd.h"

extern void led_set(u8 sta);
extern void test_fun(void(*ledset)(u8),u8 sta);
//函数名列表初始化(用户自己添加)
//用户直接在这里输入要执行的函数名及其查找串
struct _m_usmart_nametab usmart_nametab[]=
{
#if USMART_USE_WRFUNS==1    //如果使能了读写操作
    (void*)read_addr,"u32 read_addr(u32 addr)",
    (void*)write_addr,"void write_addr(u32 addr,u32 val)",
#endif
    (void*)delay_ms,"void delay_ms(u16 nms)",
    (void*)delay_us,"void delay_us(u32 nus)",
    (void*)LCD_Clear,"void LCD_Clear(u16 Color)",
    (void*)LCD_Fill,"void LCD_Fill(u16 xsta,u16 ysta,u16 xend,u16 yend,u16 color)",
    (void*)LCD_DrawLine,"void LCD_DrawLine(u16 x1, u16 y1, u16 x2, u16 y2)",
    (void*)LCD_DrawRectangle,"void LCD_DrawRectangle(u16 x1, u16 y1, u16 x2, u16 y2)",
    (void*)LCD_Draw_Circle,"void Draw_Circle(u16 x0,u16 y0,u8 r)",
    (void*)LCD_ShowNum,"void LCD_ShowNum(u16 x,u16 y,u32 num,u8 len,u8 size)",
    (void*)LCD_ShowString,"void LCD_ShowString(u16 x,u16 y,u16 width,u16 height,u8 size,u8 *p)",
    (void*)LCD_Fast_DrawPoint,"void LCD_Fast_DrawPoint(u16 x,u16 y,u16 color)",
    (void*)LCD_ReadPoint,"u16 LCD_ReadPoint(u16 x,u16 y)",
    (void*)LCD_Display_Dir,"void LCD_Display_Dir(u8 dir)",
    (void*)LCD_ShowxNum,"void LCD_ShowxNum(u16 x,u16 y,u32 num,u8 len,u8 size,u8 mode)",
    (void*)led_set,"void led_set(u8 sta)",
    (void*)test_fun,"void test_fun(void(*ledset)(u8),u8 sta)",
};
```

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 test.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)
{
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    LED_Init();                         //初始化 LED
    LCD_Init();                         //初始化 LCD FSMC 接口
    usmart_dev.init(84);                //初始化 USMART
    POINT_COLOR=RED;                    //画笔颜色：红色
    LCD_ShowString(30,50,200,16,16," ELITE STM32 ^_^");
    LCD_ShowString(30,70,200,16,16,"USMART TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16," 2019/10/8 ");
    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

将程序下载到精英 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时，屏幕上显示了一些字符（就是主函数里面要显示的字符）。

我们打开串口调试助手 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：

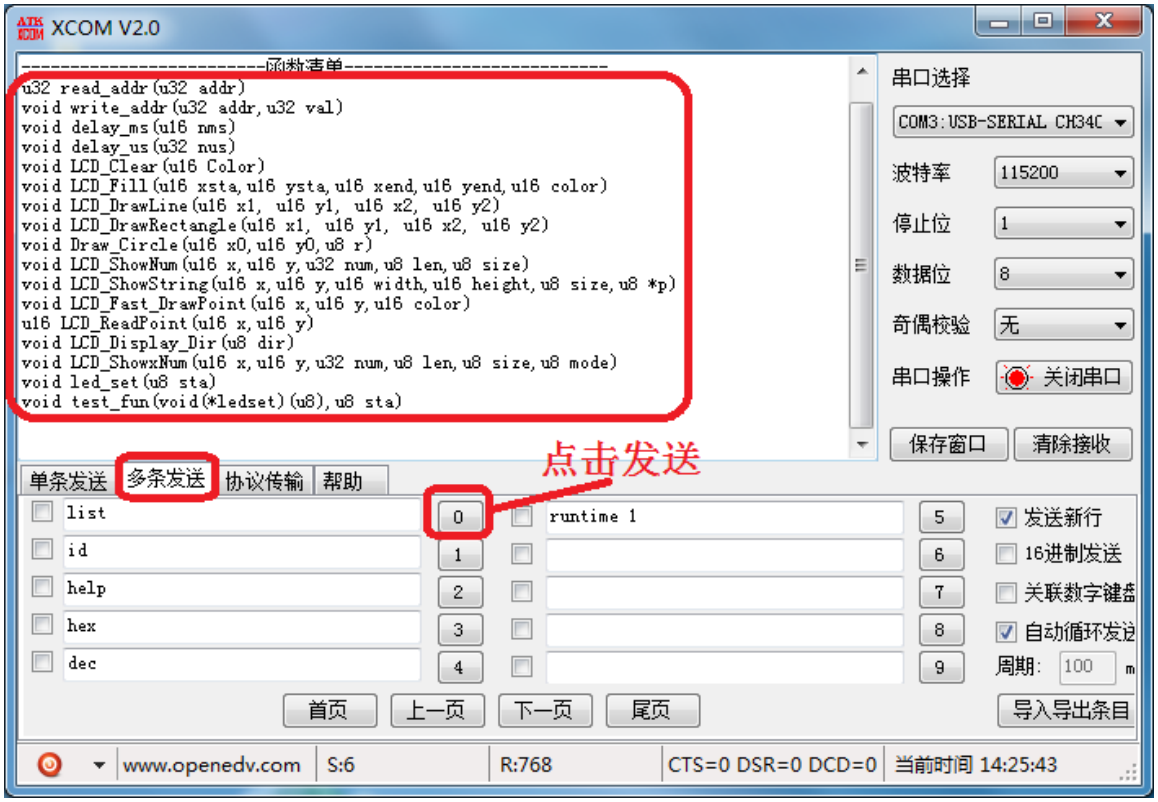


图 19.4.1 驱动串口调试助手

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：

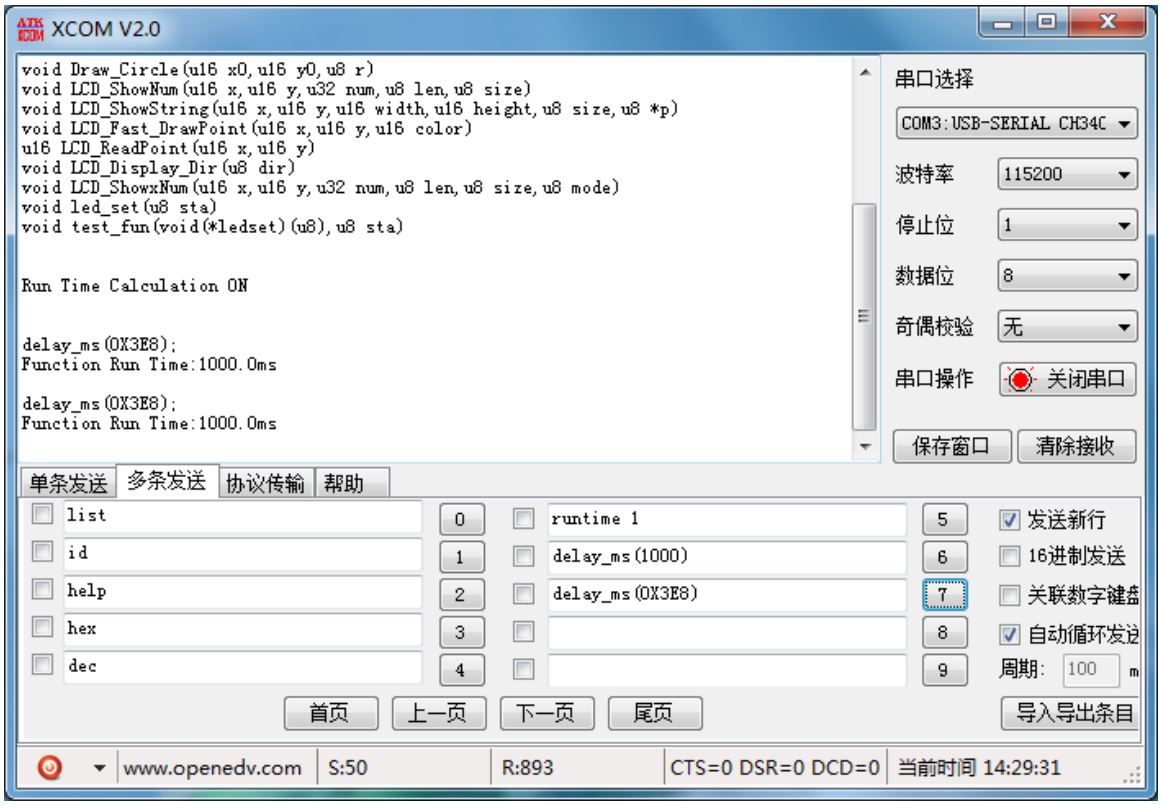


图 19.4.2 串口调用 delay_ms 函数

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：

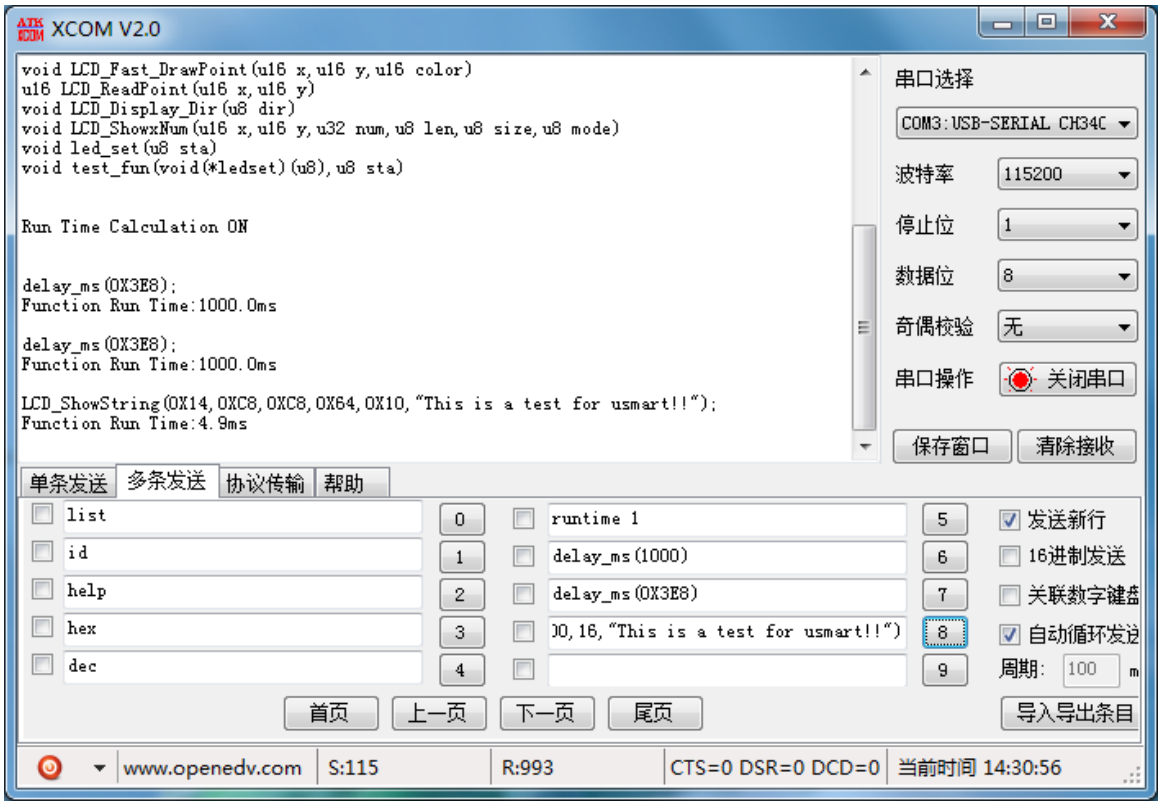


图 19.4.3 串口调用 LCD_ShowString 函数

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(注意：这个地址要以实际串口输出结果为准)，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 STM32F1 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 STM32F1 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 STM32F1 RTC 时钟简介

STM32 的实时时钟（RTC）是一个独立的定时器。STM32 的 RTC 模块拥有一组连续计数的计数器，在相应软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。

RTC 模块和时钟配置系统(RCC_BDCR 寄存器)是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前， 先要取消备份区域（BKP）写保护。

RTC 的简化框图，如图 20.1.1 所示：

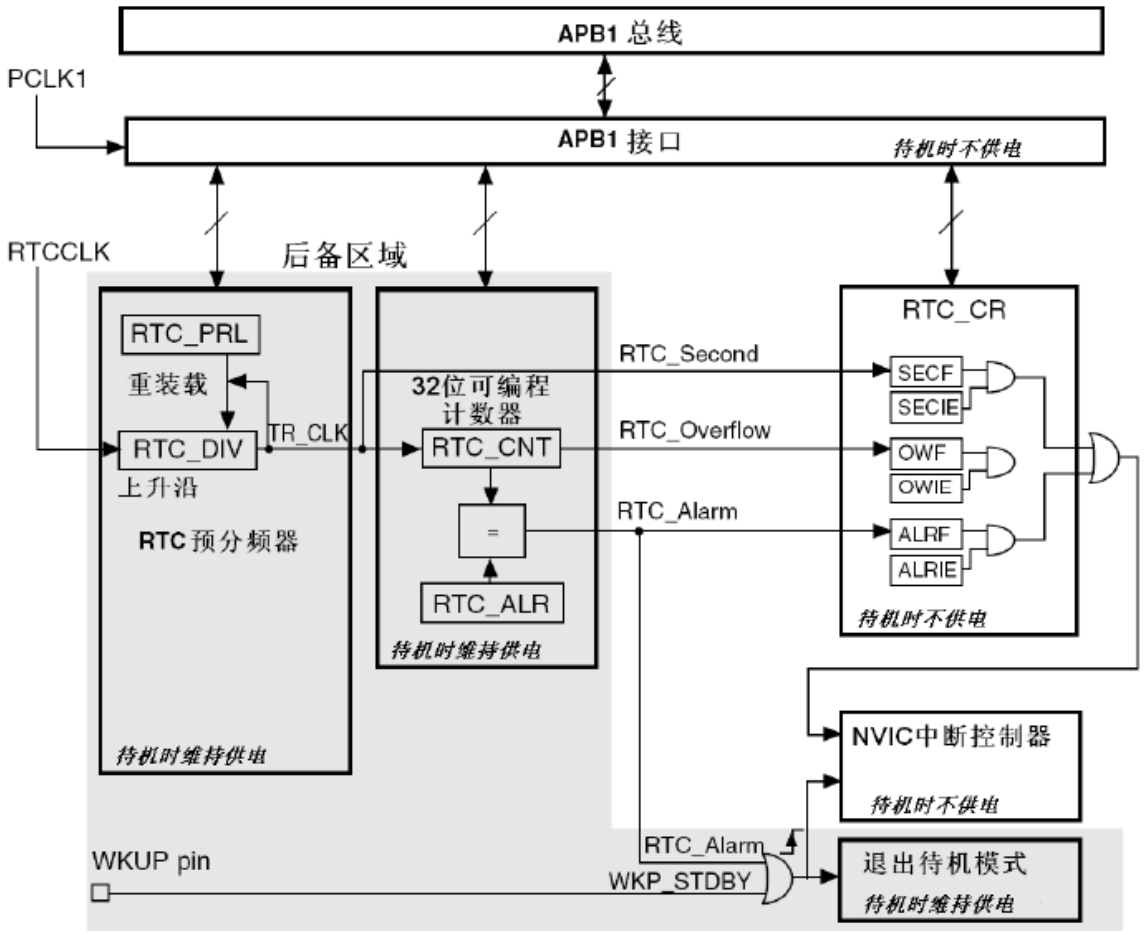


图 20.1.1 RTC 框图

RTC 由两个主要部分组成（参见图 20.1.1），第一部分(APB1 接口)用来和 APB1 总线相连。此单元还包含一组 16 位寄存器，可通过 APB1 总线对其进行读写操作。APB1 接口由 APB1 总线时钟驱动，用来与 APB1 总线连接。

另一部分(RTC 核心)由一组可编程计数器组成，分成两个主要模块。第一个模块是 RTC 的预分频模块，它可编程产生 1 秒的 RTC 时间基准 TR_CLK。RTC 的预分频模块包含了一个 20 位的可编程分频器(RTC 预分频器)。如果在 RTC_CR 寄存器中设置了相应的允许位，则在每个 TR_CLK 周期中 RTC 产生一个中断(秒中断)。第二个模块是一个 32 位的可编程计数器（RTC_CNT），可被初始化为当前的系统时间，一个 32 位的时钟计数器，按秒钟计算，可以记录 4294967296 秒，约合 136 年左右，作为一般应用，这已经是足够了的。

RTC 还有一个闹钟寄存器 RTC_ALR，用于产生闹钟。系统时间按 TR_CLK 周期累加并与

存储在 RTC_ALR 寄存器中的可编程时间相比较，如果 RTC_CR 控制寄存器中设置了相应允许位，比较匹配时（即：RTC_CNT=RTC_ALR 时）将产生一个闹钟中断，从而实现闹钟功能。

RTC 内核完全独立于 RTC APB1 接口，而软件是通过 APB1 接口访问 RTC 的预分频值、计数器值和闹钟值的。但是相关可读寄存器只在 RTC APB1 时钟进行重新同步的 RTC 时钟的上升沿被更新，RTC 标志也是如此。这就意味着，如果 APB1 接口刚刚被开启之后，在第一次的内部寄存器更新之前，从 APB1 上读取的 RTC 寄存器值可能被破坏了（通常读到 0）。因此，若在读取 RTC 寄存器曾经被禁止的 RTC APB1 接口，软件首先必须等待 RTC_CRL 寄存器的 RSF 位（寄存器同步标志位，bit3）被硬件置 1。

接下来，我们介绍一下 RTC 相关的几个寄存器。首先要介绍的是 RTC 的控制寄存器，RTC 总共有 2 个控制寄存器 RTC_CRH 和 RTC_CRL，两个都是 16 位的。RTC_CRH 的各位描如图 20.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留													OWIE	ALRIE	SECIE
													rw	rw	rw

位15:3	保留，被硬件强制为0。
位2	**OWIE**：允许溢出中断位 0：屏蔽(不允许)溢出中断 1：允许溢出中断
位1	**ALRIE**：允许闹钟中断 0：屏蔽(不允许)闹钟中断 1：允许闹钟中断
位0	**SECIE**：允许秒中断 0：屏蔽(不允许)秒中断 1：允许秒中断

图 20.1.2 RTC_CRH 寄存器各位描述

该寄存器用来控制中断的，我们本章将要用到秒钟中断，所以在该寄存器必须设置最低位为 1，以允许秒钟中断。我们再看看 RTC_CRL 寄存器。该寄存器各位描述如图 20.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留										RTOFF	CNF	RSF	OWF	ALRF	SECF
										r	rw	rc w0	rc w0	rc w0	rc w0

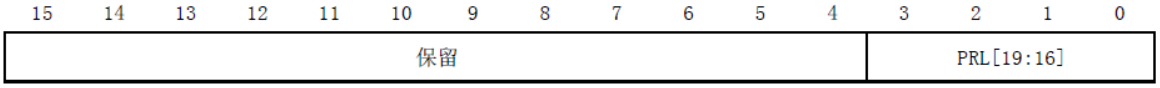
位15:6	保留，被硬件强制为0。
位5	**RTOFF**：RTC操作关闭 RTC模块利用这位来指示对其寄存器进行的最后一次操作的状态，指示操作是否完成。若此位为0，则表示无法对任何的RTC寄存器进行写操作。此位为只读位。 0：上一次对RTC寄存器的写操作仍在进行; 1：上一次对RTC寄存器的写操作已经完成。
位4	**CNF**：配置标志 此位必须由软件置'1'以进入配置模式，从而允许向RTC_CNT、RTC_ALR或RTC_PRL寄存器写入数据。只有当此位在被置'1'并重新由软件清'0'后，才会执行写操作。 0：退出配置模式(开始更新RTC寄存器)； 1：进入配置模式。
位3	**RSF**：寄存器同步标志 每当RTC_CNT寄存器和RTC_DIV寄存器由软件更新或清'0'时，此位由硬件置'1'。在APB1复位后，或APB1时钟停止后，此位必须由软件清'0'。要进行任何的读操作之前，用户程序必须等待这位被硬件置'1'，以确保RTC_CNT、RTC_ALR或RTC_PRL已经被同步。 0：寄存器尚未被同步； 1：寄存器已经被同步。
位2	**OWF**：溢出标志 当32位可编程计数器溢出时，此位由硬件置'1'。如果RTC_CRH寄存器中OWIE=1，则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0：无溢出； 1：32位可编程计数器溢出。
位1	**ALRF**：闹钟标志 当32位可编程计数器达到RTC_ALR寄存器所设置的预定值，此位由硬件置'1'。如果RTC_CRH寄存器中ALRIE=1，则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0：无闹钟； 1：有闹钟。
位0	**SECF**：秒标志 当32位可编程预分频器溢出时，此位由硬件置'1'同时RTC计数器加1。因此，此标志为分辨率可编程的RTC计数器提供一个周期性的信号(通常为1秒)。如果RTC_CRH寄存器中SECIE=1，则产生中断。此位只能由软件清除。对此位写'1'是无效的。 0：秒标志条件不成立； 1：秒标志条件成立。

图 20.1.3 RTC_CRL 寄存器各位描述

本章我们用到的是该寄存器的 0、1、3~5 这几个位，第 0 位是秒钟标志位，我们在进入 RTC 中断的时候，通过判断这位来决定是不是发生了秒钟中断。然后必须通过软件将该位清零（写 0）。第 1 位是闹钟标志位，当 RTC_CNT 的值等于 RTC_ALR 的值时，此位将由硬件置 1（可以判断此位是否为 1 来判定是否产生了闹钟），如果设置了闹钟中断（ALRIE=1），则将产生 RTC 闹钟中断，该位也必须软件写 0 清除。

第 3 位为寄存器同步标志位，我们在修改控制寄存器 RTC_CRH/CRL 之前，必须先判断该位，是否已经同步了，如果没有则等待同步，在没同步的情况下修改 RTC_CRH/CRL 的值是不行的。第 4 位为配置标位，在软件修改 RTC_CNT/RTC_ALR/RTC_PRL 的值的时候，必须先软件置位该位，以允许进入配置模式。第 5 位为 RTC 操作位，该位由硬件操作，软件只读。通过该位可以判断上次对 RTC 寄存器的操作是否完成，如果没有，我们必须等待上一次操作结束才能开始下一次操作。

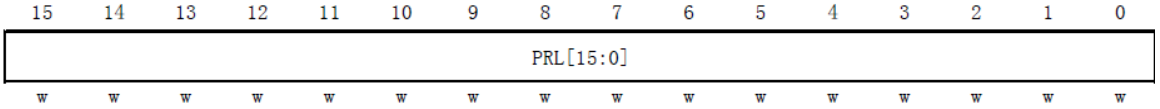
第二个要介绍的寄存器是 RTC 预分频装载寄存器，也有 2 个寄存器组成，RTC_PRLH 和 RTC_PRLL。这两个寄存器用来配置 RTC 时钟的分频数的，比如我们使用外部 32.768K 的晶振作为时钟的输入频率，那么我们要设置这两个寄存器的值为 32767，以得到一秒钟的计数频率。RTC_PRLH 的各位描述如图 20.1.4 所示：



位15:6	保留，被硬件强制为0。
位3:0	**PRL[19:16]**：RTC预分频装载值高位 根据以下公式，这些位用来定义计数器的时钟频率： fTR_CLK = fRTCCLK/(PRL[19:0]+1) 注：不推荐使用0值，否则无法正确的产生RTC中断和标志位。

图 20.1.4 RTC_PRLH 寄存器各位描述

从图 20.1.4 可以看出，RTC_PRLH 只有低四位有效，用来存储 PRL 的 19~16 位。而 PRL 的前 16 位，存放在 RTC_PRLL 里面，寄存器 RTC_PRLL 的各位描述如图 20.1.5 所示：



位15:0	**PRL[15:0]**：RTC预分频装载值低位 根据以下公式，这些位用来定义计数器的时钟频率： $f_{TR\_CLK} = f_{RTCCLK}/(PRL[19:0]+1)$

图 20.1.5 RTC_PRLL 寄存器各位描述

在介绍完这两个寄存器之后，我们介绍 RTC 预分频器余数寄存器，该寄存器也有 2 个寄存器组成 RTC_DIVH 和 RTC_DIVL，这两个寄存器的作用就是用来获得比秒钟更为准确的时钟，比如可以得到 0.1 秒，或者 0.01 秒等。该寄存器的值自减的，用于保存还需要多少时钟周期获得一个秒信号。在一次秒钟更新后，由硬件重新装载。这两个寄存器和 RTC 预分频装载寄存器的各位是一样的，这里我们就不列出来了。

接着要介绍的是 RTC 最重要的寄存器，RTC 计数器寄存器 RTC_CNT。该寄存器由 2 个 16 位的寄存器组成 RTC_CNTH 和 RTC_CNTL，总共 32 位，用来记录秒钟值（TR_CLK=1Hz 的情况下）。一般我们设置时间，就是设置 RTC_CNTH/RTC_CNTL 寄存器的值，假定我们以 1970 年为起始时间，那么当 RTC_CNTH=RTC_CNTL=0 的时候，就代表 1970 年 1 月 1 日 0 时 0 分，这样，我们就可以很方便的根据 RTC_CNT 的值计算当前时间了。反过来，如果要设置时间，则只需要将当前时间的年份减去 1970，然后剩下的时间换算成秒钟，写入 RTC_CNT 即可完成时间设置。

最后，我们再介绍下闹钟寄存器 RTC_ALR。该寄存器同 RTC_CNT 一样，也是由 2 个 16 位的寄存器组 RTC_ALRH 和 RTC_ALRL 组成，总共 32 位，用来记录闹钟时刻，实际上，RTC_ALR 就是一个用于同 RTC_CNT 比较的寄存器，当 RTC_CNT=RTC_ALR 的时候，就说明闹钟时间到了，需要闹铃。因此 RTC_ALR 的设置和读取完全同 RTC_CNT 一模一样。假定我们设置 RTC_CNTH=RTC_CNTL=0，然后设置 RTC_ALRH=0 且 RTC_ALRL=30，然后启动 RTC，那么 30 秒钟后，ALRF 将为 1，表示有闹钟产生，如果开启了闹钟中断（ALRIE=1），那么将产生闹钟中断，这就是 STM32F1 的 RTC 闹钟原理。

因为我们使用到备份寄存器来存储 RTC 的相关信息（我们这里主要用来标记时钟是否已经经过了配置），我们这里顺便介绍一下 STM32 的备份寄存器。

备份寄存器是 42 个 16 位的寄存器（精英开发板就是大容量的），可用来存储 84 个字节的用户应用程序数据。他们处在备份域里，当 VDD 电源被切断，他们仍然由 VBAT 维持供电。即使系统在待机模式下被唤醒，或系统复位或电源复位时，他们也不会被复位。

此外，BKP 控制寄存器用来管理侵入检测和 RTC 校准功能，这里我们不作介绍。

复位后，对备份寄存器和 RTC 的访问被禁止，并且备份域被保护以防止可能存在的意外的写操作。执行以下操作可以使能对备份寄存器和 RTC 的访问：

1）通过设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位来打开电源和后备接口的时钟

2）电源控制寄存器(PWR_CR)的 DBP 位来使能对后备寄存器和 RTC 的访问。

我们一般用 BKP 来存储 RTC 的校验值或者记录一些重要的数据，相当于一个 EEPROM，不过这个 EEPROM 并不是真正的 EEPROM，而是需要电池来维持它的数据。关于 BKP 的详细介绍请看《STM32 中文参考手册》的第 47 页，5.1 一节。

最后，我们还要介绍一下备份区域控制寄存器 RCC_BDCR。该寄存器的个位描述如图 20.1.6 所示：



位31:17	保留，始终读为0。
位16	**BDRST**：备份域软件复位 由软件置'1'或清'0' 0：复位未激活； 1：复位整个备份域。
位15	**RTCEN**：RTC时钟使能 由软件置'1'或清'0' 0：RTC时钟关闭； 1：RTC时钟开启。
位14:10	保留，始终读为0。
位9:8	**RTCSEL[1:0]**：RTC时钟源选择 由软件设置来选择RTC时钟源。一旦RTC时钟源被选定，直到下次后备域被复位，它不能在被改变。可通过设置BDRST位来清除。 00：无时钟； 01：LSE振荡器作为RTC时钟； 10：LSI振荡器作为RTC时钟； 11：HSE振荡器在128分频后作为RTC时钟。
位7:3	保留，始终读为0。
位2	**LSEBYP**：外部低速时钟振荡器旁路 在调试模式下由软件置'1'或清'0'来旁路LSE。只有在外部32kHz振荡器关闭时，才能写入该位 0：LSE时钟未被旁路； 1：LSE时钟被旁路。
位1	**LSERDY**：外部低速LSE就绪 由硬件置'1'或清'0'来指示是否外部32kHz振荡器就绪。在LSEON被清零后，该位需要6个外部低速振荡器的周期才被清零。 0：外部32kHz振荡器未就绪； 1：外部32kHz振荡器就绪。
位0	**LSEON**：外部低速振荡器使能 由软件置'1'或清'0' 0：外部32kHz振荡器关闭； 1：外部32kHz振荡器开启。

图 20.1.6 RCC_ BDCR 寄存器各位描述

RTC 的时钟源选择及使能设置都是通过这个寄存器来实现的，所以我们在 RTC 操作之前先要通过这个寄存器选择 RTC 的时钟源，然后才能开始其他的操作。

寄存器介绍就给大家介绍到这里了，我们下面来看看要经过哪几个步骤的配置才能使 RTC 正常工作。RTC 正常工作的一般配置步骤如下：接下来我们来看看通过 HAL 库配置 RTC 一般配置步骤。RTC 相关的 HAL 库文件为 stm32f1xx_hal_rtc.c 以及头文件 stm32f1xx_hal_rtc.h 中：

1）使能电源时钟和备份区域时钟。

前面已经介绍了，我们要访问 RTC 和备份区域就必须先使能电源时钟和备份区域时钟。这个通过 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。HAL 库设置方法为：

```
__HAL_RCC_PWR_CLK_ENABLE();//使能电源时钟 PWR
```

```
HAL_PWR_EnableBkUpAccess();//取消备份区域写保护
```

2）开启外部低速振荡器 LSE，选择 RTC 时钟，并使能。

配置开启 LSE 的函数为 HAL_RCC_OscConfig，使用方法为：

```
RCC_OscInitStruct.OscillatorType=RCC_OSCILLATORTYPE_LSE;//LSE 配置
RCC_OscInitStruct.PLL.PLLState=RCC_PLL_NONE;
RCC_OscInitStruct.LSEState=RCC_LSE_ON;                    //RTC 使用 LSE
HAL_RCC_OscConfig(&RCC_OscInitStruct);
```

选择 RTC 时钟源为函数为 HAL_RCCEx_PeriphCLKConfig，使用方法为：

```
PeriphClkInitStruct.PeriphClockSelection=RCC_PERIPHCLK_RTC;//外设为 RTC
PeriphClkInitStruct.RTCClockSelection=RCC_RTCCLKSOURCE_LSE;//RTC 时钟源为 LSE
HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);
```

使能 RTC 时钟方法为：

```
__HAL_RCC_RTC_ENABLE();//RTC 时钟使能
```

3）初始化 RTC，设置 RTC 的分频，以及配置 RTC 参数。

在 HAL 中，初始化 RTC 是通过函数 HAL_RTC_Init 实现的，该函数声明为：

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc);
```

同样按照以前的方式，我们来看看 RTC 初始化参数结构体 RTC_HandleTypeDef 定义：

```
typedef struct
{
  RTC_TypeDef                 *Instance;
  RTC_InitTypeDef              Init;
  RTC_DateTypeDef              DateToUpdate;
  HAL_LockTypeDef              Lock;
  __IO HAL_RTCStateTypeDef     State;
}RTC_HandleTypeDef;
```

这里我们着重讲解成员变量 Init 含义，因为它是真正的 RTC 初始化变量，它是 RTC_InitTypeDef 结构体类型，结构体 RTC_InitTypeDef 定义为：

```
typedef struct
{
  uint32_t AsynchPrediv;
  uint32_t OutPut;
}RTC_InitTypeDef;
```

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，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。

OutPut 用来选择要连接到 RTC_ALARM 输出的标志，取值为：RTC_OUTPUT_DISABLE（禁止输出），RTC_OUTPUT_ALARMA（使能闹钟 A 输出），RTC_OUTPUT_ALARMB（使能闹钟 B 输出）和 RTC_OUTPUT_WAKEUP（使能唤醒输出）。

接下来我们看看 RTC 初始化的一般格式：

```
RTC_Handler.Instance=RTC;
RTC_Handler.Init.AsynchPrediv=32767;
HAL_RTC_Init(&RTC_Handler);
```

同样，HAL 库也提供了 RTC 初始化 MSP 函数。函数声明为：

```
void HAL_RTC_MspInit(RTC_HandleTypeDef* hrtc);
```

该函数内部一般存放时钟使能，时钟源选择等操作程序。

4）设置 RTC 的时间。

HAL 库中，设置 RTC 时间的函数为：

```
HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc,
                                  RTC_TimeTypeDef *sTime, uint32_t Format);
```

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 函数的第三个参数 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。

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 Hours;
  uint8_t Minutes;
  uint8_t Seconds;
}RTC_TimeTypeDef;
```

这三个成员变量就比较好理解了，分别用来设置 RTC 时间参数的小时，分钟，秒钟，大家参考前面讲解的 RTC_TR 的位描述即可。HAL_RTC_SetTime 函数参考实例如下：

```
RTC_TimeTypeDef RTC_TimeStructure;
RTC_TimeStructure.Hours=hour;
RTC_TimeStructure.Minutes=min;
RTC_TimeStructure.Seconds=sec;
HAL_RTC_SetTime(&RTC_Handler,&RTC_TimeStructure,RTC_FORMAT_BIN);
```

5）设置 RTC 的日期。

设置 RTC 的日期函数为：

```
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc,
                                  RTC_DateTypeDef *sDate, uint32_t Format);
```

实际上，根据我们前面寄存器的讲解，HAL_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。

该函数有三个入口参数，我们着重讲解第二个入口参数 sData，它是结构体 RTC_DateTypeDef 指针类型变量，结构体 RTC_DateTypeDef 定义如下：

```
typedef struct
{
  uint8_t WeekDay;    //星期几
  uint8_t Month;      //月份
  uint8_t Date;       //日期
  uint8_t Year;       //年份
}RTC_DateTypeDef;
```

结构体一共四个成员变量，这四个成员变量非常好理解，对应的是 RTC_DR 寄存器相关设置位，这里我们就不做过多讲解。

6）获取 RTC 当前日期和时间。

获取当前 RTC 时间的函数为：

```
HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc,
```

```
                                    RTC_TimeTypeDef *sTime, uint32_t Format);
```

获取当前 RTC 日期的函数为：

```
HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef *hrtc,
                                  RTC_DateTypeDef *sDate, uint32_t Format);
```

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 STM32F1 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 精英 STM32F103 标配是有电池的）。

20.3 软件设计

打开上一章的工程，首先在 HARDWARE 文件夹下新建一个 RTC 的文件夹。然后打开 USER 文件夹下的工程，新建一个 rtc.c 的文件和 rtc.h 的头文件，保存在 RTC 文件夹下，并将 RTC 文件夹加入头文件包含路径。

由于篇幅所限，rtc.c 中的代码，我们不全部贴出了，这里针对几个重要的函数，进行简要说明，首先是 RTC_Init，其代码如下：

```
//实时时钟配置
//初始化 RTC 时钟,同时检测时钟是否工作正常
//BKP->DR1 用于保存是否第一次配置的设置
//返回 0:正常
//其他:错误代码
u8 RTC_Init(void)
{
    RTC_Handler.Instance=RTC;
    RTC_Handler.Init.AsynchPrediv=32767;
                        //时钟周期设置(有待观察,看是否跑慢了?)理论值：32767
    if(HAL_RTC_Init(&RTC_Handler)!=HAL_OK) return 1;
  if(HAL_RTCEx_BKUPRead(&RTC_Handler,RTC_BKP_DR1)!=0X5050)//是否第一次配置
    {
      RTC_Set(2017,5,27,17,7,0); //设置日期和时间，2017 年 5 月 27 日，17 点 07 分 0 秒

      HAL_RTCEx_BKUPWrite(&RTC_Handler,RTC_BKP_DR1,0X5050);
                                        //标记已经初始化过了
```

```
            printf("FIRST TIME\n");
        }

        __HAL_RTC_ALARM_ENABLE_IT(&RTC_Handler,RTC_IT_SEC);    //允许秒中断
        __HAL_RTC_ALARM_ENABLE_IT(&RTC_Handler,RTC_IT_ALRA); //允许闹钟中断
        HAL_NVIC_SetPriority(RTC_IRQn,0x01,0x02);       //抢占优先级 1,子优先级 2
        HAL_NVIC_EnableIRQ(RTC_IRQn);
        RTC_Get();//更新时间
        return 0; //ok
    }
```

该函数用来初始化 RTC 时钟，但是只在第一次的时候设置时间，以后如果重新上电/复位都不会再进行时间设置了（前提是备份电池有电），在第一次配置的时候，我们是按照上面介绍的 RTC 初始化步骤来做的，这里就不在多说了。这里设置时间和日期，是通过 RTC_Set 函数来实现的，该函数将在后续进行介绍。这里默认将时间设置为 2017 年 5 月 27 日，17 点 07 分 0 秒。在设置好时间之后，我们向 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，该函数通过判断 RTC 对应 BKR 的值，来决定是不是需要重新设置时间，如果不需要设置，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。

该函数还有返回值，返回值代表此次操作的成功与否，如果返回 0，则代表初始化 RTC 成功，如果返回值非零则代表错误代码了。

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 HAL_RTCEx_BKUPRead(RTC_HandleTypeDef *hrtc, uint32_t BackupRegister);
void HAL_RTCEx_BKUPWrite(RTC_HandleTypeDef *hrtc, uint32_t BackupRegister,
                                                    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。这里我们只是略微点到为止。

介绍完 RTC_Init，我们来介绍一下 RTC_Set 函数，代码如下：

```
//设置时钟
//把输入的时钟转换为秒钟
//以 1970 年 1 月 1 日为基准
//1970~2099 年为合法年份
//返回值:0,成功;其他:错误代码.
//月份数据表
u8 const table_week[12]={0,3,3,6,1,4,6,2,5,0,3,5}; //月修正数据表
//平年的月份日期表
const u8 mon_table[12]={31,28,31,30,31,30,31,31,30,31,30,31};
//syear,smon,sday,hour,min,sec：年月日时分秒
//返回值：设置结果。0，成功；1，失败。
u8 RTC_Set(u16 syear,u8 smon,u8 sday,u8 hour,u8 min,u8 sec)
{
    u16 t;
    u32 seccount=0;
    if(syear<1970||syear>2099)return 1;
```

```
    for(t=1970;t<syear;t++)                         //把所有年份的秒钟相加
    {
        if(Is_Leap_Year(t))seccount+=31622400;      //闰年的秒钟数
        else seccount+=31536000;                    //平年的秒钟数
    }
    smon-=1;
    for(t=0;t<smon;t++)                             //把前面月份的秒钟数相加
    {
        seccount+=(u32)mon_table[t]*86400;          //月份秒钟数相加
        if(Is_Leap_Year(syear)&&t==1)seccount+=86400;//闰年 2 月份增加一天的秒钟数
    }
    seccount+=(u32)(sday-1)*86400;  //把前面日期的秒钟数相加
    seccount+=(u32)hour*3600;       //小时秒钟数
    seccount+=(u32)min*60;          //分钟秒钟数
    seccount+=sec;                  //最后的秒钟加上去
    //设置时钟
    RCC->APB1ENR|=1<<28;            //使能电源时钟
    RCC->APB1ENR|=1<<27;            //使能备份时钟
    PWR->CR|=1<<8;                  //取消备份区写保护
    //上面三步是必须的!
    RTC->CRL|=1<<4;                 //允许配置
    RTC->CNTL=seccount&0xffff;
    RTC->CNTH=seccount>>16;
    RTC->CRL&=~(1<<4);              //配置更新
    while(!(RTC->CRL&(1<<5)));      //等待 RTC 寄存器操作完成
    RTC_Get();                      //设置完之后更新一下数据
    return 0;
}
```

该函数用于设置时间，把我们输入的时间，转换为以 1970 年 1 月 1 日 0 时 0 分 0 秒当做起始时间的秒钟信号，后续的计算都以这个时间为基准的，由于 STM32 的秒钟计数器可以保存 136 年的秒钟数据，这样我们可以计时到 2106 年。

接着，我们介绍 RTC_Alarm_Set 函数，该函数用于设置闹钟时间，同 RTC_Set 函数几乎一模一样，主要区别，就是将：RTC->CNTL 和 RTC->CNTH 换成了 RTC->ALRL 和 RTC->ALRH，用于设置闹钟时间，具体代码请参考本例程源码。

接着，我们介绍一下 RTC_Get 函数，该函数用于获取时间和日期等数据，其代码如下：

```
//得到当前的时间，结果保存在 calendar 结构体里面
//返回值:0,成功;其他:错误代码.
u8 RTC_Get(void)
{
    static u16 daycnt=0;
    u32 timecount=0;
    u32 temp=0;
    u16 temp1=0;
```

```
timecount=RTC->CNTH;      //得到计数器中的值(秒钟数)
timecount<<=16;
timecount+=RTC->CNTL;
temp=timecount/86400;     //得到天数(秒钟数对应的)
if(daycnt!=temp)          //超过一天了
{
    daycnt=temp;
    temp1=1970;           //从 1970 年开始
    while(temp>=365)
    {
        if(Is_Leap_Year(temp1))//是闰年
        {
            if(temp>=366)temp-=366;//闰年的秒钟数
            else break;
        }
        else temp-=365;       //平年
        temp1++;
    }
    calendar.w_year=temp1;    //得到年份
    temp1=0;
    while(temp>=28)           //超过了一个月
    {
        if(Is_Leap_Year(calendar.w_year)&&temp1==1)//当年是不是闰年/2 月份
        {
            if(temp>=29)temp-=29;//闰年的秒钟数
            else break;
        }
        else
        {
            if(temp>=mon_table[temp1])temp-=mon_table[temp1];//平年
            else break;
        }
        temp1++;
    }
    calendar.w_month=temp1+1; //得到月份
    calendar.w_date=temp+1;   //得到日期
}
temp=timecount%86400;         //得到秒钟数
calendar.hour=temp/3600;      //小时
calendar.min=(temp%3600)/60;  //分钟
calendar.sec=(temp%3600)%60;  //秒钟
calendar.week=RTC_Get_Week(calendar.w_year,calendar.w_month,calendar.w_date);
return 0;
```

```
}
```

函数其实就是将存储在秒钟寄存器 RTC->CNTH 和 RTC->CNTL 中的秒钟数据转换为真正的时间和日期。该代码还用到了一个 calendar 的结构体，calendar 是我们在 rtc.h 里面将要定义的一个时间结构体，用来存放时钟的年月日时分秒等信息。因为 STM32 的 RTC 只有秒钟计数器，而年月日，时分秒这些需要我们自己软件计算。我们把计算好的值保存在 calendar 里面，方便其他程序调用。

最后，我们介绍一下秒钟中断服务函数，该函数代码如下：

```
//RTC 时钟中断
//每秒触发一次
void RTC_IRQHandler(void)
{
    if(__HAL_RTC_ALARM_GET_FLAG(&RTC_Handler,RTC_FLAG_SEC)!=RESET)
                                        //秒中断
    {
        __HAL_RTC_ALARM_CLEAR_FLAG(&RTC_Handler,RTC_FLAG_SEC);
                                        //清除秒中断
        RTC_Get();                      //更新时间
        LED1=!LED1;                     //LED1 翻转
    }
    if(__HAL_RTC_ALARM_GET_FLAG(&RTC_Handler,RTC_FLAG_SEC)!=RESET)
                                        //闹钟中断
    {
        __HAL_RTC_ALARM_CLEAR_FLAG(&RTC_Handler,RTC_FLAG_ALRAF);
                                        //清除闹钟中断
        RTC_Get();                      //更新时间
        printf("ALARM A!\r\n");
    }
    __HAL_RTC_ALARM_CLEAR_FLAG(&RTC_Handler,RTC_FLAG_OW);// 清 除 溢 出
}
```

此部分代码比较简单，主要做了 2 个处理，通过 RTC->CRL 的不同位来判断发生的是何种中断，如果是秒钟中断，则执行一次时间的计算，获得最新时间，结果保存在 calendar 结构体里面，因此，我们可以在 calendar 里面读到最新的时间、日期等信息。如果是闹钟中断，则更新时间后，将当前的闹铃时间通过 printf 打印出来，可以在串口调试助手看到当前的闹铃情况。

我们看看 main 函数源码如下：

```
int main(void)
{
    u8 t;
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                 //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    LCD_Init();                     //初始化 LCD FSMC 接口
```

```
usmart_dev.init(84);                    //初始化 USMART
RTC_Init();                             //初始化 RTC

POINT_COLOR=RED;                        //设置字体为红色
LCD_ShowString(30,50,200,16,16," ELITE STM32F103 ^_^");
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,"2019/9/17");
while(RTC_Init())        //RTC 初始化 ，一定要初始化成功
{
    LCD_ShowString(30,130,200,16,16,"RTC ERROR!     ");
    delay_ms(800);
    LCD_ShowString(30,130,200,16,16,"RTC Trying...");
}
//显示时间
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(30,130,200,16,16,"       -   -        ");
LCD_ShowString(30,166,200,16,16,"    :   :    ");
while(1)
{
    if(t!=calendar.sec)
    {
        t=calendar.sec;
        LCD_ShowNum(30,130,calendar.w_year,4,16);

        LCD_ShowNum(70,130,calendar.w_month,2,16);

        LCD_ShowNum(94,130,calendar.w_date,2,16);
        switch(calendar.week)
        {
            case 0:
                LCD_ShowString(30,148,200,16,16,"Sunday      ");
                break;
            case 1:
                LCD_ShowString(30,148,200,16,16,"Monday      ");
                break;
            case 2:
                LCD_ShowString(30,148,200,16,16,"Tuesday     ");
                break;
            case 3:
                LCD_ShowString(30,148,200,16,16,"Wednesday");
                break;
            case 4:
```

```
                    LCD_ShowString(30,148,200,16,16,"Thursday ");
                    break;
                case 5:
                    LCD_ShowString(30,148,200,16,16,"Friday   ");
                    break;
                case 6:
                    LCD_ShowString(30,148,200,16,16,"Saturday ");
                    break;
            }
            LCD_ShowNum(30,166,calendar.hour,2,16);

            LCD_ShowNum(54,166,calendar.min,2,16);

            LCD_ShowNum(78,166,calendar.sec,2,16);
            LED0=!LED0;
        }
        delay_ms(10);
    };
}
```

这部分代码，也比较简单，注意，我们通过

为了方便设置时间，我们在 usmart_config.c 里面，修改 usmart_nametab 如下：

```
struct _m_usmart_nametab usmart_nametab[]=
{
#if USMART_USE_WRFUNS==1    //如果使能了读写操作
    (void*)read_addr,"u32 read_addr(u32 addr)",
    (void*)write_addr,"void write_addr(u32 addr,u32 val)",
#endif
    (void*)delay_ms,"void delay_ms(u16 nms)",
    (void*)delay_us,"void delay_us(u32 nus)",
    (void*)RTC_Set,"u8 RTC_Set(u16 syear,u8 smon,u8 sday,u8 hour,u8 min,u8 sec)  ",
    (void*)RTC_Alarm_Set,"u8 RTC_Alarm_Set(u16 syear,u8 smon,u8 sday,u8 hour,u8 min,
      u8 sec)",
};
```

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 的时间和闹钟了。

至此，RTC 实时时钟的软件设计就完成了，接下来就让我们来检验一下，我们的程序是否正确了。

20.4 下载验证

将程序下载到精英 STM32 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块开始显示时间，实际显示效果如图 20.4.1 所示：

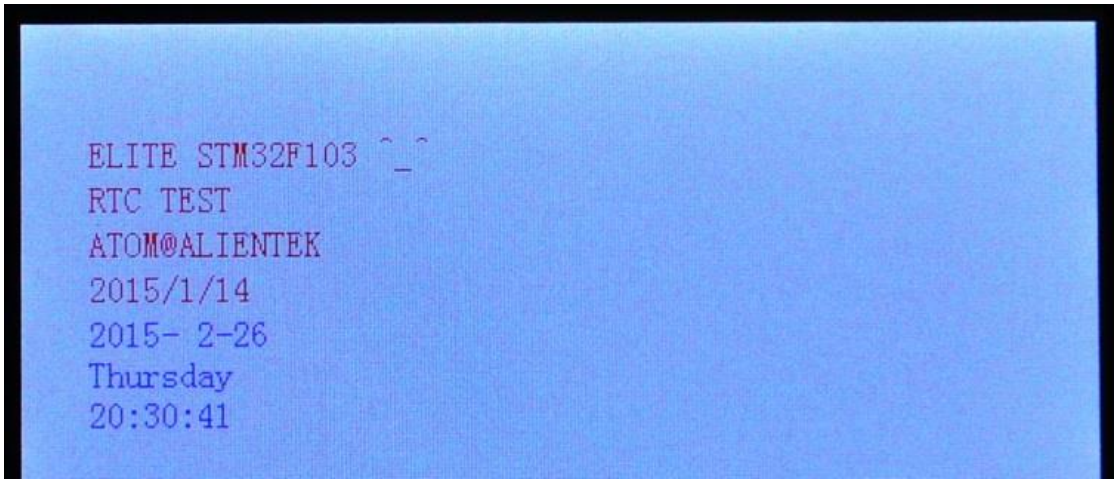


图 20.4.1 RTC 实验测试图

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：

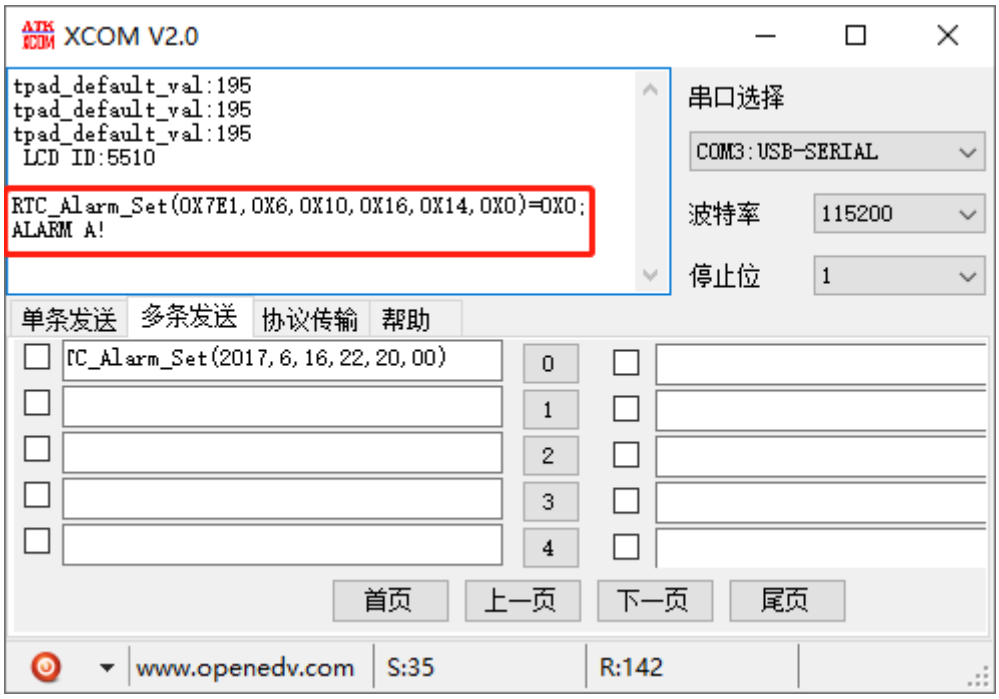


图 20.4.2 通过 USMART 设置时间和日期并测试闹钟 A

可以看到，通过 RTC_Alarm_Set 设置闹钟时间，并等待闹铃时间到了以后，串口返回了 ALARM A!的字符串，说明我们的闹钟运行正常！达到了预期的效果！

第二十一章 待机唤醒实验

本章我们将向大家介绍 STM32F1 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：

21.1 STM32F1 待机模式简介

21.2 硬件设计

21.3 软件设计

21.4 下载验证

21.1 STM32F1 待机模式简介

很多单片机都有低功耗模式，STM32 也不例外。在系统或电源复位以后，微控制器处于运行状态。运行状态下的 HCLK 为 CPU 提供时钟，内核执行程序代码。当 CPU 不需继续运行时，可以利用多个低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗，最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。STM32 的 3 种低功耗模式我们在 5.2.4 节有粗略介绍，这里我们再回顾一下。

STM32 的低功耗模式有 3 种：

1)睡眠模式（CM3 内核停止，外设仍然运行）

2)停止模式（所有时钟都停止）

3)待机模式（1.8V 内核电源关闭）

在运行模式下，我们也可以通过降低系统时钟关闭 APB 和 AHB 总线上未被使用的外设的时钟来降低功耗。三种低功耗模式一览表见表 21.1.1 所示：

模式	进入操作	唤醒	对1.8V区域时钟的影响	对VDD区域时钟的影响	电压调节器
睡眠(SLEEP-NOW或SLEEP-ON-EXIT)	WFI	任一中断	CPU时钟关，对其他时钟和ADC时钟无影响	无	开
	WFE	唤醒事件			
停机	PDDS和LPDS位+SLEEPDEEP位+WFI或WFE	任一外部中断(在外部中断寄存器中设置)	所有使用1.8V的区域的时钟都已关闭，HSI和HSE的振荡器关闭		在低功耗模式下可进行开/关设置(依据电源控制寄存器(PWR_CR)的设定)
待机	PDDS位+SLEEPDEEP位+WFI或WFE	WKUP引脚的上升沿、RTC警告事件、NRST引脚上的外部复位、IWDG复位			关

表 21.1.1 STM32 低功耗一览表

在这三种低功耗模式中，最低功耗的是待机模式，在此模式下，最低只需要 2uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 20uA 左右。最后就是睡眠模式了。用户可以根据自己的需求来决定使用哪种低功耗模式。

本章，我们仅对 STM32 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32 的最低功耗。该模式是在 CM3 深睡眠模式时关闭电压调节器。整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失。仅备份的寄存器和待机电路维持供电。

那么我们如何进入待机模式呢？其实很简单，只要按图 21.1.1 所示的步骤执行就可以了：

待机模式	说明
进入	在以下条件下执行WFI或WFE指令： - 设置Cortex™-M3系统控制寄存器中的SLEEPDEEP位 - 设置电源控制寄存器(PWR_CR)中的PDDS位 - 清除电源控制/状态寄存器(PWR_CSR)中的WUF位被
退出	WKUP引脚的上升沿、RTC闹钟、NRST引脚上外部复位、IWDG复位。
唤醒延时	复位阶段时电压调节器的启动。

图 21.1.1 STM32 进入及退出待机模式的条件

图 21.1.1 还列出了退出待机模式的操作，从图 21.1.1 可知，我们有 4 种方式可以退出待机模式，即当一个外部复位(NRST 引脚)、IWDG 复位、WKUP 引脚上的上升沿或 RTC 闹钟事件发生时，微控制器从待机模式退出。从待机唤醒后，除了电源控制/状态寄存器(PWR_CSR)，所

有寄存器被复位。

从待机模式唤醒后的代码执行等同于复位后的执行(采样启动模式引脚，读取复位向量等)。电源控制/状态寄存器(PWR_CSR)将会指示内核由待机状态退出。

在进入待机模式后，除了复位引脚以及被设置为防侵入或校准输出时的 TAMPER 引脚和被使能的唤醒引脚（WK_UP 脚），其他的 IO 引脚都将处于高阻态。

图 21.1.1 已经清楚的说明了进入待机模式的通用步骤，其中涉及到 2 个寄存器，即电源控制寄存器（PWR_CR）和电源控制/状态寄存器（PWR_CSR）。下面我们介绍一下这两个寄存器：

电源控制寄存器（PWR_CR），该寄存器的各位描述如图 21.1.2 所示：

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

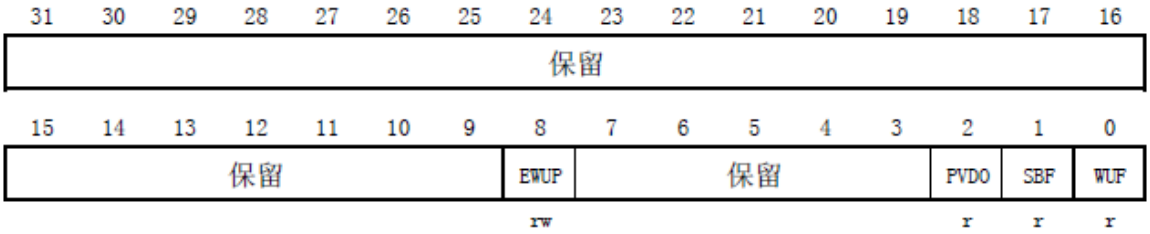
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留							DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
							rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

位 31:9	保留。始终读为0。
位 8	**DBP**：取消后备区域的写保护 在复位后，RTC和后备寄存器处于被保护状态以防意外写入。设置这位允许写入这些寄存器。 0：禁止写入RTC和后备寄存器 1：允许写入RTC和后备寄存器
位 7:5	**PLS[2:0]**：PVD电平选择 这些位用于选择电源电压监测器的电压阀值 000：2.2V　　100：2.6V 001：2.3V　　101：2.7V 010：2.4V　　110：2.8V 011：2.5V　　111：2.9V 注：详细说明参见数据手册中的电气特性部分。
位 4	**PVDE**：电源电压监测器(PVD)使能 0：禁止PVD 1：开启PVD
位 3	**CSBF**：清除待机位 始终读出为0 0：无功效 1：清除SBF待机位(写)
位 2	**CWUF**：清除唤醒位 始终读出为0 0：无功效 1：2个系统时钟周期后清除WUF唤醒位(写)
位 1	**PDDS**：掉电深睡眠 与LPDS位协同操作 0：当CPU进入深睡眠时进入停机模式，调压器的状态由LPDS位控制。 1：CPU进入深睡眠时进入待机模式。
位 0	**LPDS**：深睡眠下的低功耗 PDDS=0时，与PDDS位协同操作 0：在停机模式下电压调压器开启 1：在停机模式下电压调压器处于低功耗模式

图 21.1.2 PWR_CR 寄存器各位描述

这里我们通过设置 PWR_CR 的 PDDS 位，使 CPU 进入深度睡眠时进入待机模式，同时我

们通过 CWUF 位，清除之前的唤醒位。电源控制/状态寄存器（PWR_CSR）的各位描述如图 21.1.3 所示：



位31:9	保留。始终读为0。
位 8	**EWUP**：使能WKUP管脚 0：WKUP管脚为通用I/O。WKUP管脚上的事件不能将CPU从待机模式唤醒 1：WKUP管脚用于将CPU从待机模式唤醒，WKUP管脚被强置为输入下拉的配置(WKUP管脚上的上升沿将系统从待机模式唤醒) 注：在系统复位时清除这一位。
位 7:3	保留。始终读为0。
位 2	**PVDO**：PVD输出 当PVD被PVDE位使能后该位才有效 0：V_{DD}/V_{DDA}高于由PLS[2:0]选定的PVD阀值 1：V_{DD}/V_{DDA}低于由PLS[2:0]选定的PVD阀值 注：在待机模式下PVD被停止。因此，待机模式后或复位后，直到设置PVDE位之前，该位为0。
位 1	**SBF**：待机标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CSBF位清除。 0：系统不在待机模式 1：系统进入待机模式
位 0	**WUF**：唤醒标志 该位由硬件设置，并只能由POR/PDR(上电/掉电复位)或设置电源控制寄存器(PWR_CR)的CWUF位清除。 0：没有发生唤醒事件 1：在WKUP管脚上发生唤醒事件或出现RTC闹钟事件。 注：当WKUP管脚已经是高电平时，在(通过设置EWUP位)使能WKUP管脚时，会检测到一个额外的事件。

图 21.1.3 PWR_ CSR 寄存器各位描述

这里，我们通过设置 PWR_CSR 的 EWUP 位，来使能 WKUP 引脚用于待机模式唤醒。我们还可以从 WUF 来检查是否发生了唤醒事件。不过本章我们并没有用到。

通过以上介绍，我们了解了进入待机模式的方法，以及设置 WK_UP 引脚用于把 STM32 从待机模式唤醒的方法。具体步骤如下：

1）使能 PWR 时钟。

因为要配置 PWR 寄存器，所以必须先使能 PWR 时钟。

在 HAL 库中，使能 PWR 时钟的方法是：

```
__HAL_RCC_PWR_CLK_ENABLE();                    //使能 PWR 时钟
```

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 HAL 库中，设置使能 WK_UP 用于唤醒 CPU 待机模式的函数是：

```
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); //设置 WKUP 用于唤醒
```

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《CM3 权威指南》，第 182 页表 13.1），接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。在库函数中，进行上面三个功能进入待机模式是在函数 HAL_PWR_EnterSTANDBYMode 中实现的：

```
void HAL_PWR_EnterSTANDBYMode(void);
```

4）最后编写 WK_UP 中断服务函数。

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。关于外部中断服务函数以及中断服务回调函数的使用方法请参考外部中断实验，这里我们就不做过多讲解。

通过以上几个步骤的设置，我们就可以使用 STM32F1 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。

21.2 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。

21.3 软件设计

打开待机唤醒实验工程，我们可以发现工程中多了一个 wkup.c 和 wkup.h 文件，相关的用户代码写在这两个文件中。同时，对于待机唤醒功能，我们需要引入 stm32f1xx_hal_pwr.c 和 stm32f1xx_hal_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)
{
    __HAL_RCC_APB2_FORCE_RESET();          //复位所有 IO 口
    __HAL_RCC_PWR_CLK_ENABLE();            //使能 PWR 时钟

    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);                 //清除 Wake_UP 标志
    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);          //设置 WKUP 用于唤醒
    HAL_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;    //记录按下的时间
    LED0=0; //亮灯 DS0
    while(1)
    {
        if(WKUP_KD)
        {
            t++;                //已经按下了
            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
            }
        }else
        {
            LED0=1;
            return 0; //按下不足 3 秒
        }
    }
}

//外部中断线 0 中断服务函数
void EXTI0_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}

//中断线 0 中断处理过程
//此函数会被 HAL_GPIO_EXTI_IRQHandler()调用
//GPIO_Pin:引脚
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin==GPIO_PIN_0)//PA0
    {
        if(Check_WKUP())//关机
        {
            Sys_Enter_Standby();//进入待机模式
        }
    }
}

//PA0 WKUP 唤醒初始化
void WKUP_Init(void)
```

```
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_0;            //PA0
    GPIO_Initure.Mode=GPIO_MODE_IT_RISING;  //中断,上升沿
    GPIO_Initure.Pull=GPIO_PULLDOWN;        //下拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//快速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    //检查是否是正常开机
    if(Check_WKUP()==0)
    {
        Sys_Enter_Standby();//不是开机，进入待机模式
    }

    HAL_NVIC_SetPriority(EXTI0_IRQn,0x02,0x02);//抢占优先级 2，子优先级 2
    HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}
```

该部分代码比较简单，我们在这里说明两点：

1，在 void Sys_Enter_Standby(void)函数里面，我们要在进入待机模式前把所有开启的外设全部关闭，我们这里仅仅复位了所有的 IO 口，使得 IO 口全部为浮空输入。其他外设（比如 ADC 等），大家根据自己所开启的情况进行一一关闭就可，这样才能达到最低功耗！ 然后我们调用 __HAL_RCC_PWR_CLK_ENABLE() 来使能 PWR 时钟，调用函数 HAL_PWR_EnableWakeUpPin() 用来设置 WK_UP 引脚作为唤醒源。最后调用 HAL_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，然后对 GPIOA 初始化为下拉输入，上升沿触发中断，同时初始化 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。接下来程序通过判断 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，外部中断回调函数 HAL_GPIO_EXTI_Callback 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)
{
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    usmart_dev.init(84);                    //初始化 USMART
```

```
    LED_Init();                          //初始化 LED
    LCD_Init();                          //初始化 LCD FSMC 接口
    WKUP_Init();                         //待机唤醒初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    LCD_ShowString(30,70,200,16,16,"WKUP TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2019/9/17");
    while(1)
    {
        LED0=!LED0;
        delay_ms(250);
    } }
```

这里我们先初始化 LED 和 WK_UP 按键（通过 WKUP_Init（）函数初始化），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 KEY_UP 按键，来开机，否则将直接进入待机模式，无任何现象。**

21.4 下载与测试

在代码编译成功之后，下载代码到精英 STM32 V1 上，此时，看不到任何现象，和没下载代码一样，其实这是正常的，在程序下载完之后，开发板检测不到 KEY_UP（即 WK_UP 按键）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。然后，我们长按 KEY_UP 按键 3 秒钟左右（KEY_UP 按下时，DS0 会长亮），可以看到 DS0 开始闪烁，液晶也会显示一些内容。然后再长按 KEY_UP，DS0 会灭掉，液晶灭掉，程序再次进入待机模式。

第二十二章 ADC 实验

本章我们将向大家介绍 STM32F1 的 ADC 功能。在本章中，我们将使用 STM32F1 的 ADC1 通道 1 来采样外部电压值，并在 TFTLCD 模块上显示出来。本章将分为如下几个部分：

22.1 STM32F1 ADC 简介
22.2 硬件设计
22.3 软件设计
22.4 下载验证

22.1 STM32F1 ADC 简介

STM32F103 拥有 1~3 个 ADC（STM32F101/102 系列只有 1 个 ADC），这些 ADC 可以独立使用，也可以使用双重模式（提高采样率）。STM32 的 ADC 是 12 位逐次逼近型的模拟数字转换器。它有 18 个通道，可测量 16 个外部和 2 个内部信号源。各通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阀值。

STM32F103 系列最少都拥有 2 个 ADC，我们选择的 STM32F103ZET 包含有 3 个 ADC。STM32 的 ADC 最大的转换速率为 1Mhz，也就是转换时间为 1us（在 ADCCLK=14M,采样周期为 1.5 个 ADC 时钟下得到），不要让 ADC 的时钟超过 14M，否则将导致结果准确度下降。

STM32 将 ADC 的转换分为 2 个通道组：规则通道组和注入通道组。规则通道相当于你正常运行的程序，而注入通道呢，就相当于中断。在你程序正常执行的时候，中断是可以打断你的执行的。同这个类似，注入通道的转换可以打断规则通道的转换， 在注入通道被转换完成之后，规则通道才得以继续转换。

通过一个形象的例子可以说明：假如你在家里的院子内放了 5 个温度探头，室内放了 3 个温度探头；你需要时刻监视室外温度即可，但偶尔你想看看室内的温度；因此你可以使用规则通道组循环扫描室外的 5 个探头并显示 AD 转换结果，当你想看室内温度时，通过一个按钮启动注入转换组(3 个室内探头)并暂时显示室内温度，当你放开这个按钮后，系统又会回到规则通道组继续检测室外温度。从系统设计上，测量并显示室内温度的过程中断了测量并显示室外温度的过程，但程序设计上可以在初始化阶段分别设置好不同的转换组，系统运行中不必再变更循环转换的配置，从而达到两个任务互不干扰和快速切换的结果。可以设想一下，如果没有规则组和注入组的划分，当你按下按钮后，需要从新配置 AD 循环扫描的通道，然后在释放按钮后需再次配置 AD 循环扫描的通道。

上面的例子因为速度较慢，不能完全体现这样区分(规则通道组和注入通道组)的好处，但在工业应用领域中有很多检测和监视探头需要较快地处理，这样对 AD 转换的分组将简化事件处理的程序并提高事件处理的速度。

STM32 其 ADC 的规则通道组最多包含 16 个转换，而注入通道组最多包含 4 个通道。关于这两个通道组的详细介绍，请参考《STM32 中文参考手册》第 155 页，第 11 章。

STM32 的 ADC 可以进行很多种不同的转换模式，这些模式在《STM32 中文参考手册》的第 11 章也都有详细介绍，我们这里就不在一一列举了。我们本章仅介绍如何使用规则通道的单次转换模式。

STM32 的 ADC 在单次转换模式下，只执行一次转换，该模式可以通过 ADC_CR2 寄存器的 ADON 位（只适用于规则通道）启动，也可以通过外部触发启动（适用于规则通道和注入通道），这是 CONT 位为 0。

以规则通道为例，一旦所选择的通道转换完成，转换结果将被存在 ADC_DR 寄存器中，EOC（转换结束）标志将被置位，如果设置了 EOCIE，则会产生中断。然后 ADC 将停止，直到下次启动。

接下来，我们介绍一下我们执行规则通道的单次转换，需要用到的 ADC 寄存器。第一个要介绍的是 ADC 控制寄存器（ADC_CR1 和 ADC_CR2）。ADC_CR1 的各位描述如图 22.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								AWDEN	AWDENJ	保留		DUALMOD[3:0]			
								rw	rw			rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			DISCENJ	DISCEN	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 22.1.1 ADC_CR1 寄存器各位描述

这里我们不再详细介绍每个位，而是抽出几个我们本章要用到的位进行针对性的介绍，详细的说明及介绍，请参考《STM32 中文参考手册》第 11 章的相关章节。

ADC_CR1 的 SCAN 位，该位用于设置扫描模式，由软件设置和清除，如果设置为 1，则使用扫描模式，如果为 0，则关闭扫描模式。在扫描模式下，由 ADC_SQRx 或 ADC_JSQRx 寄存器选中的通道被转换。如果设置了 EOCIE 或 JEOCIE，只在最后一个通道转换完毕后才会产生 EOC 或 JEOC 中断。

ADC_CR1[19：16]用于设置 ADC 的操作模式，详细的对应关系如图 22.1.2 所示：

位19:16	**DUALMOD[3:0]**：双模式选择 软件使用这些位选择操作模式。 0000：独立模式 0001：混合的同步规则+注入同步模式 0010：混合的同步规则+交替触发模式 0011：混合同步注入+快速交替模式 0100：混合同步注入+慢速交替模式 0101：注入同步模式 0110：规则同步模式 0111：快速交替模式 1000：慢速交替模式 1001：交替触发模式 注： 在ADC2和ADC3中这些位为保留位 在双模式中，改变通道的配置会产生一个重新开始的条件，这将导致同步丢失。建议在进行任何配置改变前关闭双模式。

图 22.1.2 ADC 操作模式

本章我们要使用的是独立模式，所以设置这几位为 0 就可以了。接着我们介绍 ADC_CR2，该寄存器的各位描述如图 22.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								TSVREFE	SWSTART	SWSTARTJ	EXTTRIG	EXTSEL[2:0]			保留
								rw	rw	rw	rw	rw	rw	rw	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JEXTTRIG	JEXTSEL[2:0]			ALIGN	保留		DMA	保留				RSTCAL	CAL	CONT	ADON
rw	rw	rw	rw	rw			rw					rw	rw	rw	rw

图 22.1.3 ADC_CR2 寄存器操作模式

该寄存器我们也只针对性的介绍一些位：ADON 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。CAL 和 RSTCAL 用于 AD 校准。ALIGN 用于设置数据对齐，我们使用右对齐，该位设置为 0。

EXTSEL[2：0]用于选择启动规则转换组转换的外部事件，详细的设置关系如图 22.1.4 所示：

位19:17	**EXTSEL[2:0]**：选择启动规则通道组转换的外部事件 这些位选择用于启动规则通道组转换的外部事件 ADC1和ADC2的触发配置如下 000：定时器1的CC1事件　100：定时器3的TRGO事件 001：定时器1的CC2事件　101：定时器4的CC4事件 010：定时器1的CC3事件　110：EXTI线11/ TIM8_TRGO，仅大容量产品具有TIM8_TRGO功能 011：定时器2的CC2事件　111：SWSTART ADC3的触发配置如下 000：定时器3的CC1事件　100：定时器8的TRGO事件 001：定时器2的CC3事件　101：定时器5的CC1事件 010：定时器1的CC3事件　110：定时器5的CC3事件 011：定时器8的CC1事件　111：SWSTART

图 22.1.4 ADC 选择启动规则转换事件设置

我们这里使用的是软件触发（SWSTART），所以设置这 3 个位为 111。ADC_CR2 的 SWSTART 位用于开始规则通道的转换，我们每次转换（单次转换模式下）都需要向该位写 1。AWDEN 为用于使能温度传感器和 Vrefint。STM32 内部的温度传感器我们将在下一节介绍。

第二个要介绍的是 ADC 采样事件寄存器（ADC_SMPR1 和 ADC_SMPR2），这两个寄存器用于设置通道 0~17 的采样时间，每个通道占用 3 个位。ADC_SMPR1 的各位描述如图 22.1.5 所示：

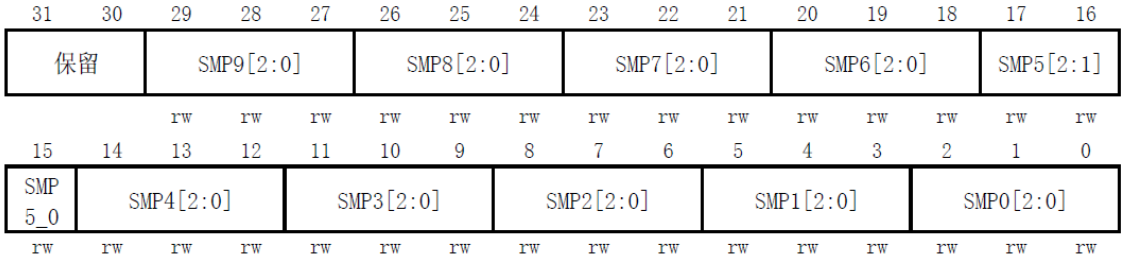
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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31:24	保留。必须保持为0。
位23:0	**SMPx[2:0]**：选择通道x的采样时间 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 000：1.5周期　100：41.5周期 001：7.5周期　101：55.5周期 010：13.5周期　110：71.5周期 011：28.5周期　111：239.5周期 注： – ADC1的模拟输入通道16和通道17在芯片内部分别连到了温度传感器和VREFINT。 – ADC2的模拟输入通道16和通道17在芯片内部连到了VSS。 - ADC3模拟输入通道14，15，16，17与Vss相连

图 22.1.5　ADC_SMPR1 寄存器各位描述

ADC_SMPR2 的各位描述如下图 22.1.6 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		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
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。必须保持为0。
位29:0	**SMPx[2:0]**：选择通道x的采样时间 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 000：1.5周期　　100：41.5周期 001：7.5周期　　101：55.5周期 010：13.5周期　　110：71.5周期 011：28.5周期　　111：239.5周期 注：ADC3模拟输入通道9与Vss相连

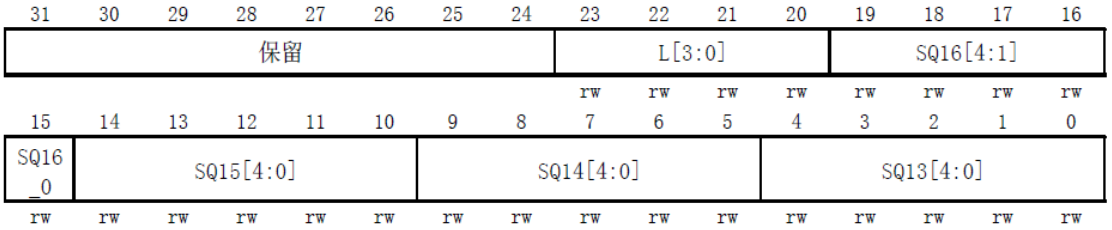
图 22.1.6　ADC_SMPR2 寄存器各位描述

对于每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率。ADC 的转换时间可以由以下公式计算：

Tcovn=采样时间+12.5 个周期

其中：Tcovn 为总转换时间，采样时间是根据每个通道的 SMP 位的设置来决定的。例如，当 ADCCLK=14Mhz 的时候，并设置 1.5 个周期的采样时间，则得到：Tcovn=1.5+12.5=14 个周期=1us。

第三个要介绍的是 ADC 规则序列寄存器（ADC_SQR1~3）,该寄存器总共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC_SQR1，该寄存器的各位描述如图 22.1.7 所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								L[3:0]				SQ16[4:1]			
								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	rw	rw	rw

位31:24	保留。必须保持为0。
位23:20	**L[3:0]**：规则通道序列长度 这些位定义了在规则通道转换序列中转换总数。 0000：1个转换 0001：2个转换 …… 1111：16个转换
位19:15	**SQ16[4:0]**：规则序列中的第16个转换 这些位定义了转换序列中的第16个转换通道的编号(0~17)。
位14:10	**SQ15[4:0]**：规则序列中的第15个转换
位9:5	**SQ14[4:0]**：规则序列中的第14个转换
位4:0	**SQ13[4:0]**：规则序列中的第13个转换

图 22.1.7　ADC_ SQR1 寄存器各位描述

L[3：0]用于存储规则序列的长度，我们这里只用了 1 个，所以设置这几个位的值为 0。其他的 SQ13~16 则存储了规则序列中第 13~16 个通道的编号（0~17）。另外两个规则序列寄存器

同 ADC_SQR1 大同小异，我们这里就不再介绍了，要说明一点的是：我们选择的是单次转换，所以只有一个通道在规则序列里面，这个序列就是 SQ1，通过 ADC_SQR3 的最低 5 位（也就是 SQ1）设置。

第四个要介绍的是 ADC 规则数据寄存器(ADC_DR)。规则序列中的 AD 转化结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。ADC_DR 的各位描述如图 22.1.8：

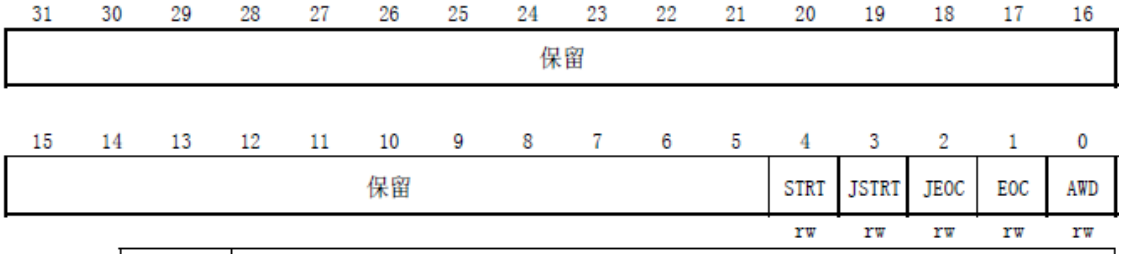
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADC2DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	**ADC2DATA[15:0]**：ADC2转换的数据 - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见10.9：双ADC模式 - 在ADC2中：不用这些位。
位15:0	**DATA[15:0]**：规则转换的数据 这些位为只读，包含了规则通道的转换结果。数据是左或右对齐，如图25和图26所示。

图 22.1.8　ADC_ JDRx 寄存器各位描述

这里要提醒一点的是，该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

最后一个要介绍的 ADC 寄存器为 ADC 状态寄存器（ADC_SR），该寄存器保存了 ADC 转换时的各种状态。该寄存器的各位描述如图 22.1.9 所示：



位31:15	保留。必须保持为0。
位4	**STRT**：规则通道开始位 该位由硬件在规则通道转换开始时设置，由软件清除。 0：规则通道转换未开始 1：规则通道转换已开始
位3	**JSTRT**：注入通道开始位 该位由硬件在注入通道组转换开始时设置，由软件清除。 0：注入通道转换未开始 1：注入通道转换已开始
位2	**JEOC**：注入通道转换结束位 该位由硬件在所有注入通道组转换结束时设置，由软件清除 0：转换未完成 1：转换完成
位1	**EOC**：转换结束位 该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取ADC_DR时清除 0：转换未完成 1：转换完成
位0	**AWD**：模拟看门狗标志位 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置，由软件清除 0：没有发生模拟看门狗事件 1：发生模拟看门狗事件

图 22.1.9　ADC_ SR 寄存器各位描述

这里我们要用到的是 EOC 位，我们通过判断该位来决定是否此次规则通道的 AD 转换已经完成，如果完成我们就从 ADC_DR 中读取转换结果，否则等待转换完成。

通过以上介绍，我们了解了 STM32 的单次转换模式下的相关设置，本章我们使用 ADC1 的通道 1 来进行 AD 转换，这里需要说明一下，使用到的库函数分布在 stm32f1xx_adc.c 文件和 stm32f1xx_adc.h 文件中。下面讲解其详细设置步骤：

1）开启 PA 口时钟和 ADC1 时钟，设置 PA1 为模拟输入。

STM32F103ZET6 的 ADC 通道 1 在 PA1 上，所以，我们先要使能 PORTA 的时钟，然后设置 PA1 为模拟输入。同时我们要把 PA1 复用为 ADC，所以我们要使能 ADC1 时钟。

使能 GPIOA 时钟和 ADC1 时钟都很简单，具体方法为：

```
__HAL_RCC_ADC1_CLK_ENABLE();            //使能 ADC1 时钟
__HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟
```

初始化 GPIOA1 为模拟输入，方法也多次讲解，关键代码为：

```
GPIO_InitTypeDef GPIO_Initure;
GPIO_Initure.Pin=GPIO_PIN_1;            //PA1
GPIO_Initure.Mode=GPIO_MODE_ANALOG;     //模拟
GPIO_Initure.Pull=GPIO_NOPULL;          //不带上下拉
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

2）初始化 ADC，设置 ADC 时钟分频系数，分辨率，模式，扫描方式，对齐方式等信息。

在 HAL 库中，初始化 ADC 是通过函数 HAL_ADC_Init 来实现的，该函数声明为：

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef* hadc);
```

该函数只有一个入口参数 hadc，为 ADC_HandleTypeDef 结构体指针类型，结构体定义为：

```
typedef struct
{
  ADC_TypeDef                 *Instance;     //ADC1/ ADC2/ ADC3
  ADC_InitTypeDef             Init;     //初始化结构体变量
  DMA_HandleTypeDef           *DMA_Handle; //DMA 方式使用
  HAL_LockTypeDef             Lock;
  __IO HAL_ADC_StateTypeDef   State;
  __IO uint32_t               ErrorCode;
}ADC_HandleTypeDef;
```

该结构体定义和其他外设比较类似，我们着重看第二个成员变量 Init 含义，它是结构体 ADC_InitTypeDef 类型，结构体 ADC_InitTypeDef 定义为：

```
typedef struct
{
  uint32_t DataAlign;      //对齐方式：左对齐还是右对齐：ADC_DATAALIGN_RIGHT
  uint32_t ScanConvMode; //扫描模式  DISABLE
  uint32_t ContinuousConvMode;//开启连续转换模式或者单次转换模式  DISABLE
  uint32_t NbrOfConversion;   //规则序列中有多少个转换  1
  uint32_t DiscontinuousConvMode;//不连续采样模式  DISABLE
  uint32_t NbrOfDiscConversion;//不连续采样通道数  0
  uint32_t ExternalTrigConv; //外部触发方式 ADC_SOFTWARE_START
}ADC_InitTypeDef;
```

我们直接把每个成员变量含义注释在结构体定义的后面，请大家仔细阅读上面注释。

这里我们需要说明一下，和其他外设一样，HAL 库同样提供了 ADC 的 MSP 初始化函数，一般情况下，时钟使能和 GPIO 初始化都会放在 MSP 初始化函数中。函数声明为：

```
void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc);
```

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）。

```
HAL_ADC_Start(&ADC1_Handler); //开启 ADC
```

5）配置通道，读取通道 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。

设置规则序列通道以及采样周期的函数是:

```
HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef* hadc,
                                        ADC_ChannelConfTypeDef* sConfig);
```

该函数有两个入口参数，第一个就不用多说了，接下来我们看第二个入口参数 sConfig，它是 ADC_ChannelConfTypeDef 结构体指针类型，结构体定义如下：

```
typedef struct
{
  uint32_t Channel;                //ADC 通道
```

```
    uint32_t Rank;                  //规则通道中的第几个转换
    uint32_t SamplingTime;          //采样时间
}ADC_ChannelConfTypeDef;
```

该结构体有四个成员变量，对于 STM32F1 只用到前面三个。Channel 用来设置 ADC 通道，Rank 用来设置要配置的通道是规则序列中的第几个转换，SamplingTime 用来设置采样时间。使用实例为：

```
ADC1_ChanConf.Channel=ch;                                   //通道
ADC1_ChanConf.Rank=1;                                       //第 1 个序列，序列 1
ADC1_ChanConf.SamplingTime=ADC_SAMPLETIME_239CYCLES_5;      //采样时间
HAL_ADC_ConfigChannel(&ADC1_Handler,&ADC1_ChanConf);        //通道配置
```

配置好通道并且使能 ADC 后，接下来就是读取 ADC 值。这里我们采取的是查询方式读取，所以我们还要等待上一次转换结束。此过程 HAL 库提供了专用函数 HAL_ADC_PollForConversion，函数定义为：

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc,
                                            uint32_t Timeout);
```

等待上一次转换结束之后，接下来就是读取 ADC 值，函数为：

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc);
```

这两个函数的使用方法都比较简单，这里我们就不累赘了。

这里还需要说明一下 ADC 的参考电压，战舰 STM32 开发板使用的是 STM32F103ZET6，该芯片有外部参考电压：Vref-和 Vref+，其中 Vref-必须和 VSSA 连接在一起，而 Vref+的输入范围为：2.4~VDDA。战舰 STM23 开发板通过 P7 端口，设置 Vref-和 Vref+设置参考电压，默认的我们是通过跳线帽将 Vref-接到 GND，Vref+接到 VDDA，参考电压就是 3.3V。如果大家想自己设置其他参考电压，将你的参考电压接在 Vref-和 Vref+上就 OK 了。本章我们的参考电压设置的是 3.3V。

通过以上几个步骤的设置，我们就能正常的使用 STM32 的 ADC1 来执行 AD 转换操作了。

22.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） TFTLCD 模块

3） ADC

4） 杜邦线

前面 2 个均已介绍过，而 ADC 属于 STM32F1 内部资源，实际上我们只需要软件设置就可以正常工作，不过我们需要在外部连接其端口到被测电压上面。本章，我们通过 ADC1 的通道 1（PA1）来读取外部电压值，精英 STM32F103 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 AD/DA 组合接口 P7 的 ADC 插针上（与 PA1 连接），另外一头就接你要测试的电压点（确保该电压不大于 3.3V 即可）。

22.3 软件设计

打开实验工程可以发现，我们在 FWLIB 分组下面新增了 stm32f1xx_hal_adc.c 源文件，同

时会引入对应的头文件 stm32f1xx_hal_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：

```
ADC_HandleTypeDef ADC1_Handler;        //ADC 句柄
//初始化 ADC
//ch: ADC_channels
//通道值  0~16 取值范围为：ADC_CHANNEL_0~ADC_CHANNEL_16
void MY_ADC_Init(void)
{
    RCC_PeriphCLKInitTypeDef ADC_CLKInit;
    ADC_CLKInit.PeriphClockSelection=RCC_PERIPHCLK_ADC;     //ADC 外设时钟
    ADC_CLKInit.AdcClockSelection=RCC_ADCPCLK2_DIV6;
                                                    //分频因子 6 时钟为 72M/6=12MHz
    HAL_RCCEx_PeriphCLKConfig(&ADC_CLKInit);                //设置 ADC 时钟
    ADC1_Handler.Instance=ADC1;
    ADC1_Handler.Init.DataAlign=ADC_DATAALIGN_RIGHT;        //右对齐
    ADC1_Handler.Init.ScanConvMode=DISABLE;                 //非扫描模式
    ADC1_Handler.Init.ContinuousConvMode=DISABLE;           //关闭连续转换
    ADC1_Handler.Init.NbrOfConversion=1;//1 个转换在规则序列中就是只转换规则序列 1
    ADC1_Handler.Init.DiscontinuousConvMode=DISABLE;        //禁止不连续采样模式
    ADC1_Handler.Init.NbrOfDiscConversion=0;                //不连续采样通道数为 0
    ADC1_Handler.Init.ExternalTrigConv=ADC_SOFTWARE_START;          //软件触发
    HAL_ADC_Init(&ADC1_Handler);                                    //初始化
    HAL_ADCEx_Calibration_Start(&ADC1_Handler);                     //校准 ADC
}

//ADC 底层驱动，引脚配置，时钟使能
//此函数会被 HAL_ADC_Init()调用
//hadc:ADC 句柄
void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_ADC1_CLK_ENABLE();            //使能 ADC1 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_1;            //PA1
    GPIO_Initure.Mode=GPIO_MODE_ANALOG;     //模拟
    GPIO_Initure.Pull=GPIO_NOPULL;          //不带上下拉
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
}

//获得 ADC 值
```

```
//ch: 通道值 0~16，取值范围为：ADC_CHANNEL_0~ADC_CHANNEL_16
//返回值:转换结果
u16 Get_Adc(u32 ch)
{
    ADC_ChannelConfTypeDef ADC1_ChanConf;

    ADC1_ChanConf.Channel=ch;                                   //通道
    ADC1_ChanConf.Rank=1;                                       //第 1 个序列，序列 1
    ADC1_ChanConf.SamplingTime=ADC_SAMPLETIME_239CYCLES_5; //采样时间
    HAL_ADC_ConfigChannel(&ADC1_Handler,&ADC1_ChanConf);        //通道配置
    HAL_ADC_Start(&ADC1_Handler);                               //开启 ADC
    HAL_ADC_PollForConversion(&ADC1_Handler,10);                //轮询转换
    return (u16)HAL_ADC_GetValue(&ADC1_Handler);
                                                //返回最近一次 ADC1 规则组的转换结果
}
//获取指定通道的转换值，取 times 次,然后平均
//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(u32 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 1。第二个函数 Get_Adc，用于读取某个通道的 ADC 值，例如我们读取通道 1 上的 ADC 值，就可以通过 Get_Adc（ADC_Channel_1）得到。最后一个函数 Get_Adc_Average，用于多次获取 ADC 值，取平均，用来提高准确度。

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;
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
```

```
    usmart_dev.init(84);                    //初始化 USMART
    LED_Init();                             //初始化 LED
    LCD_Init();                             //初始化 LCD FSMC 接口
    MY_ADC_Init();                          //初始化 ADC1 通道 1

    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2017/5/29");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,130,200,16,16,"ADC1_CH1_VAL:");
    LCD_ShowString(30,150,200,16,16,"ADC1_CH1_VOL:0.000V");//先显示小数点
    while(1)
    {
        adcx=Get_Adc_Average(ADC_CHANNEL_1,20);//获取通道 1 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 整形
        LCD_ShowxNum(134,150,adcx,1,16,0);
                    //显示电压值的整数部分，3.1111 的话，这里就是显示 3
        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，如：0.1111 转换为 111.1，相当于保留三位小数。
        LCD_ShowxNum(150,150,temp,3,16,0X80);
                    //显示小数部分（前面转换为了整形显示），这里显示的就是 111.
        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 1 的值，并显示读到的 ADC 值（数字量），以及其转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。

22.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32F103 上，可以看到 LCD 显示如图 22.4.1 所示：

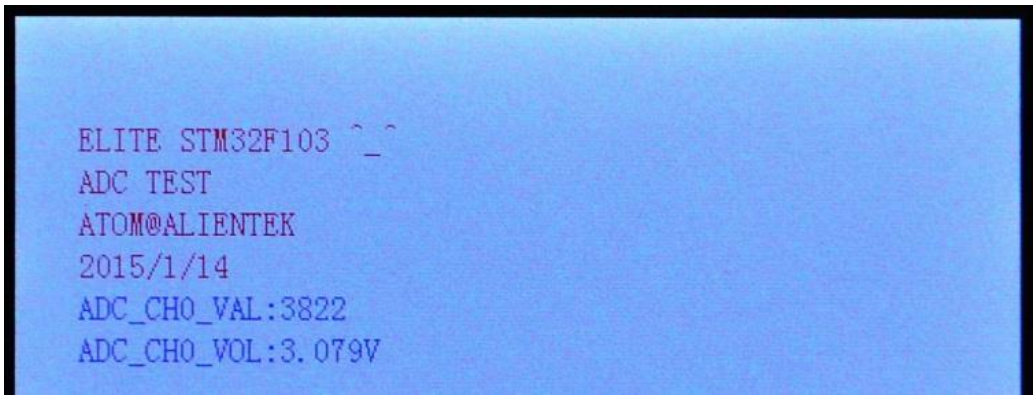


图 22.4.1 ADC 实验测试图

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 R36 的缘故。

同时伴随 DS0 的不停闪烁，提示程序在运行。大家可以试试把杜邦线接到其他地方，看看电压值是否准确？但是一定别接到 5V 上面去，否则可能烧坏 ADC！

通过这一章的学习，我们了解了 STM32F1 ADC 的使用，但这仅仅是 STM32F1 强大的 ADC 功能的一小点应用。STM32F1 的 ADC 在很多地方都可以用到，另外，ADC 的 DMA 功能是很不错的，建议有兴趣的大家深入研究下 STM32F1 的 ADC，相信会给你以后的开发带来方便。

第二十三章 内部温度传感器实验

本章我们将向大家介绍 STM32F1 的内部温度传感器。在本章中，我们将使用 STM32F1 的内部温度传感器来读取温度值，并在 TFTLCD 模块上显示出来。本章分为如下几个部分：

23.1 STM32F1 内部温度传感器简介

23.2 硬件设计

23.3 软件设计

23.4 下载验证

23.1 STM32F1 内部温度传感器简介

STM32F103 有一个内部的温度传感器，可以用来测量 CPU 及周围的温度(TA)。该温度传感器在内部和 ADCx_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值。温度传感器模拟输入推荐采样时间是 17.1μs。 STM32 的内部温度传感器支持的温度范围为：-40~125 度。精度比较差，为±1.5℃左右。

STM32 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部通道就差不多了。关于 ADC 的设置，我们在第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 STM32 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CR2 的 AWDEN 位（bit23）设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32 的内部温度传感器固定的连接在 ADC 的通道 16 上，所以，我们在设置好 ADC 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以计算出当前温度。计算公式如下：

T（℃）={（V25-Vsense）/Avg_Slope}+25

上式中：

V25=Vsense 在 25 度时的数值（典型值为：1.43）。

Avg_Slope=温度与 Vsense 曲线的平均斜率（单位为 mv/℃或 uv/℃）（典型值为 4.3Mv/℃）。

利用以上公式，我们就可以方便的计算出当前温度传感器的温度了。

现在，我们就可以总结一下 STM32 内部温度传感器使用的步骤了，如下：

1）设置 ADC，并开启 ADC_CR2 的 AWDEN 位。

关于如何设置 ADC，上一节已经介绍了，我们采用与上一节一样的设置，这里我们只要增加使能 AWDEN 位这一句就可以了。

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。

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） 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 STM32F1 内部，不需要外部设置，我们只需要软件设置就 OK 了。

23.3 软件设计

打开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个函数 Get_Temprate，该函数内容如下：

```
//得到温度值
//返回值:温度值(扩大了 100 倍,单位:℃.)
short Get_Temprate(void)
```

```
{
    u32 adcx;
    short result;
    double temperate;
    adcx=Get_Adc_Average(ADC_CHANNEL_TEMPSENSOR,20);
                                        //读取内部温度传感器通道,10 次取平均
    temperate=(float)adcx*(3.3/4096);   //电压值
    temperate=(1.43-temperate)/0.0043+25;   //转换为温度值
    result=temperate*=100;              //扩大 100 倍.
    return result;
}
```

该函数非常简单，实际上就是调用上一章实验讲解的 Get_Adc_Average 函数读取 ADC 的值，而 ADC 连接的是内部温度传感器通道 ADC_CHANNEL_TEMPSENSOR。

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;
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    usmart_dev.init(84);                //初始化 USMART
    LED_Init();                         //初始化 LED
    LCD_Init();                         //初始化 LCD FSMC 接口
    MY_ADC_Init();                      //初始化 ADC1 通道 1
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/18");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,140,200,16,16,"TEMPERATE: 00.00C");//在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。

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 精英 STM32F103 上，可以看到 LCD 显示如图 23.4.1 所示：

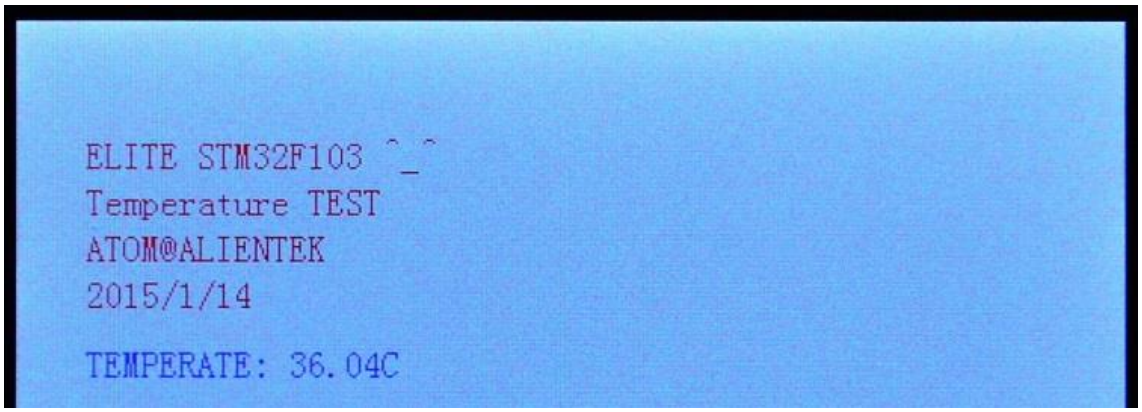


图 23.4.1 内部温度传感器实验测试图

伴随 DS0 的不停闪烁，提示程序在运行。大家可以看看你的温度值与实际是否相符合（因为芯片会发热，而且貌似准确度也不怎么好，所以一般会比实际温度偏高）？

第二十四章 光敏传感器实验

本章我们将向大家介绍精英 STM32F103 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：

24.1 光敏传感器简介
24.2 硬件设计
24.3 软件设计
24.4 下载验证

24.1 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。

精英 STM32F103 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 结中产生电子一空穴对，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 6（PF8）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。

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） 光敏传感器

前三个之前均有介绍，光敏传感器与 STM32F1 的连接如图 24.2.1 所示：

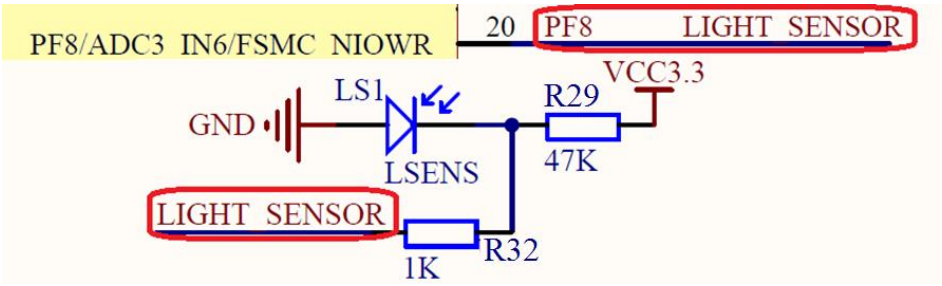


图 24.2.1 光敏传感器与 STM32F1 连接示意图

图中，LS1 是光敏二极管（实物在开发板摄像头接口右侧），R29 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_IN6 通道，读取 LIGHT_SENSOR（PF8）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。

24.3 软件设计

打开本章实验工程可以看到，在 HAL 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 stm32f1xx_hal_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 去测量关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_Initure;
    __HAL_RCC_GPIOF_CLK_ENABLE();           //开启 GPIOF 时钟
    GPIO_Initure.Pin=GPIO_PIN_8;            //PF8
    GPIO_Initure.Mode=GPIO_MODE_ANALOG;     //模拟
    GPIO_Initure.Pull=GPIO_NOPULL;          //不带上下拉
    HAL_GPIO_Init(GPIOF,&GPIO_Initure);
    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_6);    //读取 ADC 值,通道 6
        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 PF8 为模拟输入，然后通过 Adc3_Init 函数初始化 ADC3 的通道 ADC_Channel_6。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_6 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：

```
ADC_HandleTypeDef ADC3_Handler;        //ADC 句柄
//初始化 ADC
//ch: ADC_channels
//通道值  0~16 取值范围为：ADC_CHANNEL_0~ADC_CHANNEL_16
void ADC3_Init(void)
{
    RCC_PeriphCLKInitTypeDef ADC_CLKInit;
```

```
    ADC_CLKInit.PeriphClockSelection=RCC_PERIPHCLK_ADC;   //ADC 外设时钟
    ADC_CLKInit.AdcClockSelection=RCC_ADCPCLK2_DIV6;
                                                  //分频因子 6 时钟为 72M/6=12MHz
    HAL_RCCEx_PeriphCLKConfig(&ADC_CLKInit);              //设置 ADC 时钟

    ADC3_Handler.Instance=ADC3;
    ADC3_Handler.Init.DataAlign=ADC_DATAALIGN_RIGHT;      //右对齐
    ADC3_Handler.Init.ScanConvMode=DISABLE;               //非扫描模式
    ADC3_Handler.Init.ContinuousConvMode=DISABLE;         //关闭连续转换
    ADC3_Handler.Init.NbrOfConversion=1;//1 个转换在规则序列中就是只转换规则序列 1
    ADC3_Handler.Init.DiscontinuousConvMode=DISABLE;      //禁止不连续采样模式
    ADC3_Handler.Init.NbrOfDiscConversion=0;              //不连续采样通道数为 0
    ADC3_Handler.Init.ExternalTrigConv=ADC_SOFTWARE_START;        //软件触发
    HAL_ADC_Init(&ADC3_Handler);                                  //初始化
    HAL_ADCEx_Calibration_Start(&ADC3_Handler);                   //校准 ADC
}

//ADC 底层驱动，引脚配置，时钟使能
//此函数会被 HAL_ADC_Init()调用
//hadc:ADC 句柄
void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)
{
    __HAL_RCC_ADC3_CLK_ENABLE();                  //使能 ADC1 时钟
}

//获得 ADC 值
//ch: 通道值 0~16，取值范围为：ADC_CHANNEL_0~ADC_CHANNEL_16
//返回值:转换结果
u16 Get_Adc3(u32 ch)
{
    ADC_ChannelConfTypeDef ADC1_ChanConf;
    ADC1_ChanConf.Channel=ch;                                     //通道
    ADC1_ChanConf.Rank=1;                                         //第 1 个序列，序列 1
    ADC1_ChanConf.SamplingTime=ADC_SAMPLETIME_239CYCLES_5;
                                                  //采样时间为 239.5 周期
    HAL_ADC_ConfigChannel(&ADC3_Handler,&ADC1_ChanConf);   //通道配置
    HAL_ADC_Start(&ADC3_Handler);                                 //开启 ADC
    HAL_ADC_PollForConversion(&ADC3_Handler,10);                  //轮询转换
    return (u16)HAL_ADC_GetValue(&ADC3_Handler);
                                          //返回最近一次 ADC1 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_CH6 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;

    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    usmart_dev.init(84);                //初始化 USMART
    LED_Init();                         //初始化 LED
    LCD_Init();                         //初始化 LCD FSMC 接口
    Lsens_Init();                       //初始化光敏传感器

    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/18");
    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。

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 精英 STM32F103 上，可以看到 LCD 显示如图 24.4.1 所示：

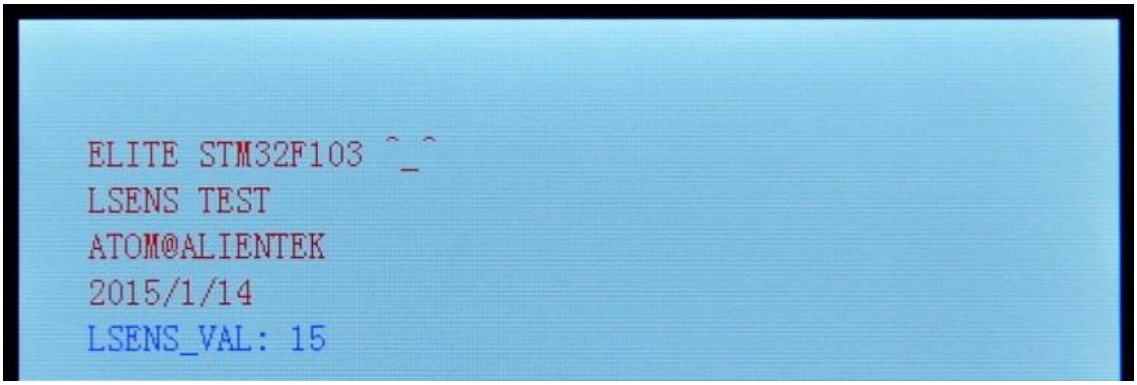


图 24.4.1 光敏传感器实验测试图

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。

第二十五章 DAC 实验

上几章，我们介绍了 STM32F1 的 ADC 使用，本章我们将向大家介绍 STM32F1 的 DAC 功能。在本章中，我们将利用按键（或 USMART）控制 STM32F1 内部 DAC1 来输出电压，通过 ADC1 的通道 1 采集 DAC 的输出电压，在 LCD 模块上面显示 ADC 获取到的电压值以及 DAC 的设定输出电压值等信息。本章将分为如下几个部分：

25.1 STM32F1 DAC 简介
25.2 硬件设计
25.3 软件设计
25.4 下载验证

25.1 STM32F1 DAC 简介

大容量的 STM32F103 具有内部 DAC，精英 STM32 选择的是 STM32F103ZET6 属于大容量产品，所以是带有 DAC 模块的。

STM32 的 DAC 模块(数字/模拟转换模块)是 12 位数字输入，电压输出型的 DAC。DAC 可以配置为 8 位或 12 位模式，也可以与 DMA 控制器配合使用。DAC 工作在 12 位模式时，数据可以设置成左对齐或右对齐。DAC 模块有 2 个输出通道，每个通道都有单独的转换器。在双 DAC 模式下，2 个通道可以独立地进行转换，也可以同时进行转换并同步地更新 2 个通道的输出。DAC 可以通过引脚输入参考电压 VREF+以获得更精确的转换结果。

STM32 的 DAC 模块主要特点有：

① 2 个 DAC 转换器：每个转换器对应 1 个输出通道
② 8 位或者 12 位单调输出
③ 12 位模式下数据左对齐或者右对齐
④ 同步更新功能
⑤ 噪声波形生成
⑥ 三角波形生成
⑦ 双 DAC 通道同时或者分别转换
⑧ 每个通道都有 DMA 功能

单个 DAC 通道的框图如图 25.1.1 所示：

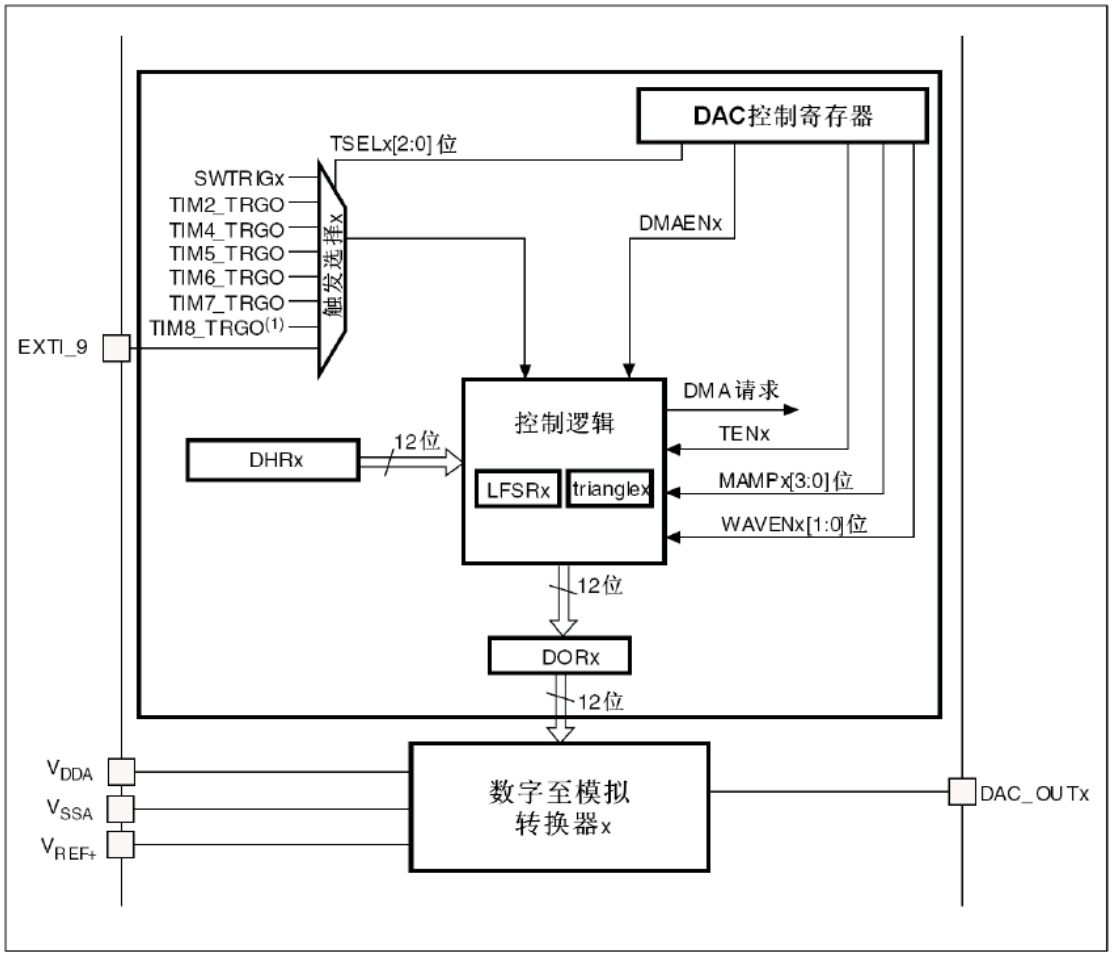


图 25.1.1 DAC 通道模块框图

图中 VDDA 和 VSSA 为 DAC 模块模拟部分的供电，而 Vref+则是 DAC 模块的参考电

压。DAC_OUTx 就是 DAC 的输出通道了（对应 PA4 或者 PA5 引脚）。

从图 25.1.1 可以看出，DAC 输出是受 DORx 寄存器直接控制的，但是我们不能直接往 DORx 寄存器写入数据，而是通过 DHRx 间接的传给 DORx 寄存器，实现对 DAC 输出的控制。前面我们提到，STM32 的 DAC 支持 8/12 位模式，8 位模式的时候是固定的右对齐的，而 12 位模式又可以设置左对齐/右对齐。单 DAC 通道 x，总共有 3 种情况：

① 8 位数据右对齐：用户将数据写入 DAC_DHR8Rx[7:0]位（实际是存入 DHRx[11:4]位）。

② 12 位数据左对齐：用户将数据写入 DAC_DHR12Lx[15:4]位（实际是存入 DHRx[11:0]位）。

③ 12 位数据右对齐：用户将数据写入 DAC_DHR12Rx[11:0]位（实际是存入 DHRx[11:0]位）。

我们本章使用的就是单 DAC 通道 1，采用 12 位右对齐格式，所以采用第③种情况。

如果没有选中硬件触发(寄存器 DAC_CR1 的 TENx 位置’0’)，存入寄存器 DAC_DHRx 的数据会在一个 APB1 时钟周期后自动传至寄存器 DAC_DORx。如果选中硬件触发(寄存器 DAC_CR1 的 TENx 位置’1’)，数据传输在触发发生以后 3 个 APB1 时钟周期后完成。 一旦数据从 DAC_DHRx 寄存器装入 DAC_DORx 寄存器，在经过时间 $t_{SETTLING}$ 之后，输出即有效，这段时间的长短依电源电压和模拟输出负载的不同会有所变化。我们可以从 STM32F103ZET6 的数据手册查到 $t_{SETTLING}$ 的典型值为 3us，最大是 4us。所以 DAC 的转换速度最快是 250K 左右。

本章我们将不使用硬件触发（TEN=0），其转换的时间框图如图 25.1.2 所示：

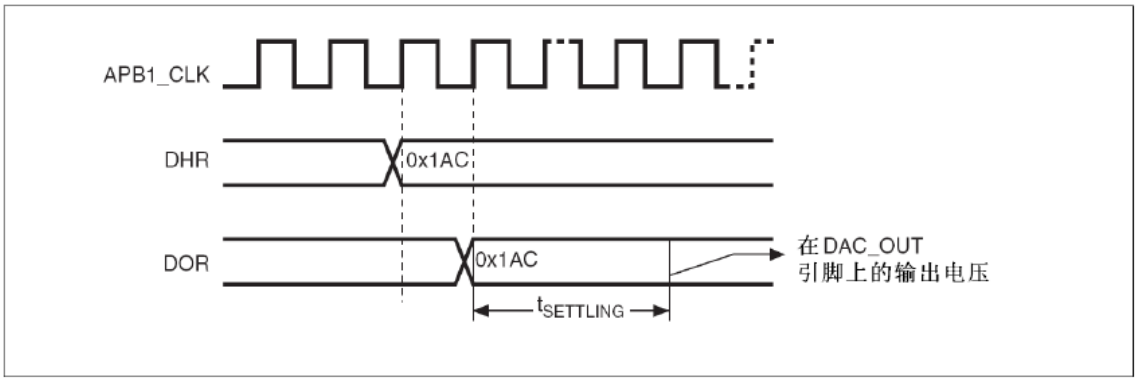


图 25.1.2 TEN=0 时 DAC 模块转换时间框图

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 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 25.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留			DMAEN2	MAMP2[3:0]				WAVE2[2:0]		TSEL2[2:0]			TEN2	BOFF2	EN2
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			DMAEN1	MAMP13:0]				WAVE1[2:0]		TSEL1[2:0]			TEN1	BOFF1	EN1
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 25.1.3 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 25.1.4 所示：

位7:6	**WAVE1[1:0]**：DAC通道1噪声/三角波生成使能 (DAC channel1 noise/triangle wave generation enable) 该2位由软件设置和清除。 00：关闭波形生成； 10：使能噪声波形发生器； 1x：使能三角波发生器。
位5:3	**TSEL1[2:0]**：DAC通道1触发选择 (DAC channel1 trigger selection) 该位用于选择DAC通道1的外部触发事件。 000：TIM6 TRGO事件； 001：对于互联型产品是TIM3 TRGO事件，对于大容量产品是TIM8 TRGO事件； 010：TIM7 TRGO事件； 011：TIM5 TRGO事件； 100：TIM2 TRGO事件； 101：TIM4 TRGO事件； 110：外部中断线9； 111：软件触发。 注意：该位只能在TEN1= 1(DAC通道1触发使能)时设置。
位2	**TEN1**：DAC通道1触发使能 (DAC channel1 trigger enable) 该位由软件设置和清除，用来使能/关闭DAC通道1的触发。 0：关闭DAC通道1触发，写入寄存器DAC_DHRx的数据在1个APB1时钟周期后传入寄存器DAC_DOR1； 1：使能DAC通道1触发，写入寄存器DAC_DHRx的数据在3个APB1时钟周期后传入寄存器DAC_DOR1。 注意：如果选择软件触发，写入寄存器DAC_DHRx的数据只需要1个APB1时钟周期就可以传入寄存器DAC_DOR1。
位1	**BOFF1**：关闭DAC通道1输出缓存 (DAC channel1 output buffer disable) 该位由软件设置和清除，用来使能/关闭DAC通道1的输出缓存。 0：使能DAC通道1输出缓存； 1：关闭DAC通道1输出缓存。
位0	**EN1**：DAC通道1使能 (DAC channel1 enable) 该位由软件设置和清除，用来使能/失能DAC通道1。 0：关闭DAC通道1； 1：使能DAC通道1。

图 25.1.4 寄存器 DAC_CR 低八位详细描述

首先，我们来看 DAC 通道 1 使能位(EN1)，该位用来控制 DAC 通道 1 使能的，本章我们就是用的 DAC 通道 1，所以该位设置为 1。

再看关闭 DAC 通道 1 输出缓存控制位（BOFF1），这里 STM32 的 DAC 输出缓存做的有些不好，如果使能的话，虽然输出能力强一点，但是输出没法到 0，这是个很严重的问题。所以本章我们不使用输出缓存。即设置该位为 1。

DAC 通道 1 触发使能位（TEN1），该位用来控制是否使用触发，里我们不使用触发，所以设置该位为 0。

DAC 通道 1 触发选择位（TSEL1[2:0]），这里我们没用到外部触发，所以设置这几个位为 0 就行了。

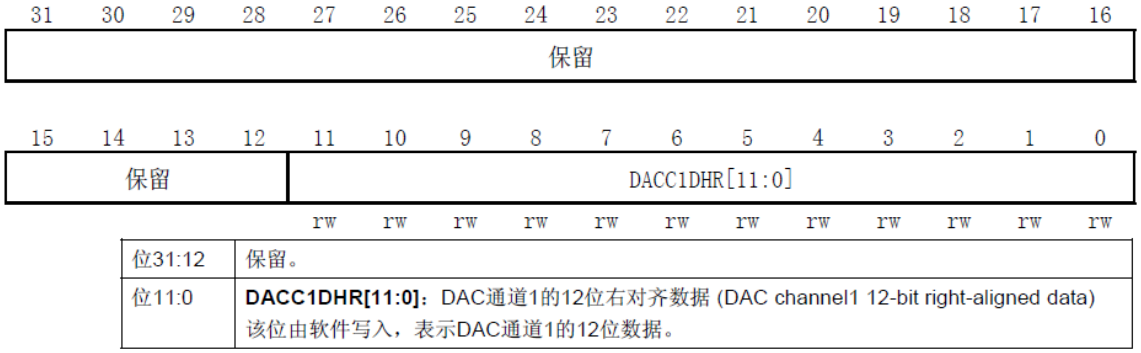
DAC 通道 1 噪声/三角波生成使能位（WAVE1[1:0]），这里我们同样没用到波形发生器，故也设置为 0 即可。

DAC 通道 1 屏蔽/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 25.1.5 所示：



位31:12	保留。
位11:0	**DACC1DHR[11:0]**：DAC通道1的12位右对齐数据 (DAC channel1 12-bit right-aligned data) 该位由软件写入，表示DAC通道1的12位数据。

图 25.1.5 寄存器 DAC_DHR12R1 各位描述

该寄存器用来设置 DAC 输出，通过写入 12 位数据到该寄存器，就可以在 DAC 输出通道 1（PA4）得到我们所要的结果。

通过以上介绍，我们了解了 STM32 实现 DAC 输出的相关设置，本章我们将使用 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 stm32f1xx_dac.c 以及头文件 stm32f1xx_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。

STM32F103ZET6 的 DAC 通道 1 是接在 PA4 上的，所以，我们先要使能 PORTA 的时钟，然后设置 PA4 为模拟输入（虽然是输入，但是 STM32 内部会连接在 DAC 模拟输出上）。程序如下：

```
__HAL_RCC_DAC_CLK_ENABLE();             //使能 DAC 时钟
__HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟
GPIO_Initure.Pin=GPIO_PIN_4;            //PA4
GPIO_Initure.Mode=GPIO_MODE_ANALOG;     //模拟
GPIO_Initure.Pull=GPIO_NOPULL;          //不带上下拉
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

2）初始化 DAC,设置 DAC 的工作模式。

HAL 库中提供了一个 DAC 初始化函数 HAL_DAC_Init，该函数声明如下：

```
HAL_StatusTypeDef HAL_DAC_Init(DAC_HandleTypeDef* hdac);
```

该函数并没有设置任何 DAC 相关寄存器，也就是说没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC，也就是说，为后面 HAL 库操作 DAC 做好准备。它有一个很重要的作用就是在函数内部会调用 DAC 的 MSP 初始化函数 HAL_DAC_MspInit，该函数声明如下：

```
void HAL_DAC_MspInit(DAC_HandleTypeDef* hdac);
```

一般情况下，步骤 1 中的与 MCU 相关的时钟使能和 IO 口配置都放在该函数中实现。

HAL 库提供了一个很重要的 DAC 配置函数 HAL_DAC_ConfigChannel，该函数用来配置 DAC 通道的触发类型以及输出缓冲。该函数声明如下：

```
HAL_StatusTypeDef HAL_DAC_ConfigChannel(DAC_HandleTypeDef* hdac,
```

```
                                        DAC_ChannelConfTypeDef* sConfig, uint32_t Channel);
```

第一个入口参数非常简单，为 DAC 初始化句柄，和 HAL_DAC_Init 保存一致即可。

第三个入口参数 Channel 用来配置 DAC 通道，比如我们使用 PA4，也就是 DAC 通道 1，所以配置值为 DAC_CHANNEL_1 即可。

接下来我们看看第二个入口参数 sConfig，该参数是 DAC_ChannelConfTypeDef 结构体指针类型，结构体 DAC_ChannelConfTypeDef 定义如下：

```
typedef struct
{
  uint32_t DAC_Trigger;          // DAC 触发类型
  uint32_t DAC_OutputBuffer;    //输出缓冲
}DAC_ChannelConfTypeDef;
```

成员变量 DAC_Trigger 用来设置 DAC 触发类型，DAC_OutputBuffer 用来设置输出缓冲，这在我们前面都有讲解。DAC 初始化配置实例代码如下：

```
DAC_HandleTypeDef DAC1_Handler;
DAC_ChannelConfTypeDef DACCH1_Config;
DAC1_Handler.Instance=DAC;
HAL_DAC_Init(&DAC1_Handler); //初始化 DAC
DACCH1_Config.DAC_Trigger=DAC_TRIGGER_NONE;     //不使用触发功能
DACCH1_Config.DAC_OutputBuffer=DAC_OUTPUTBUFFER_DISABLE;
HAL_DAC_ConfigChannel(&DAC1_Handler,&DACCH1_Config,DAC_CHANNEL_1);
```

3）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，HAL 库函数是：

```
HAL_StatusTypeDef HAL_DAC_Start(DAC_HandleTypeDef* hdac, uint32_t Channel);
```

该函数非常简单，第一个参数是 DAC 句柄，第二个用来设置 DAC 通道。

4）设置 DAC 的输出值。

通过前面 3 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，，就可以在 DAC 输出引脚（PA4）得到不同的电压值了，HAL 库函数为：

```
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef* hdac,
                                   uint32_t Channel, uint32_t Alignment, uint32_t Data);
```

该函数从入口参数可以看出，它是配置 DAC 的通道输出值，同时通过第三个入口参数设置对齐方式。

最后，再提醒一下大家，本例程，我们使用的是 3.3V 的参考电压，即 Vref+连接 VDDA。

通过以上几个步骤的设置，我们就能正常的使用 STM32 的 DAC 通道 1 来输出不同的模拟电压了。

25.2 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 25.2.1 所示：

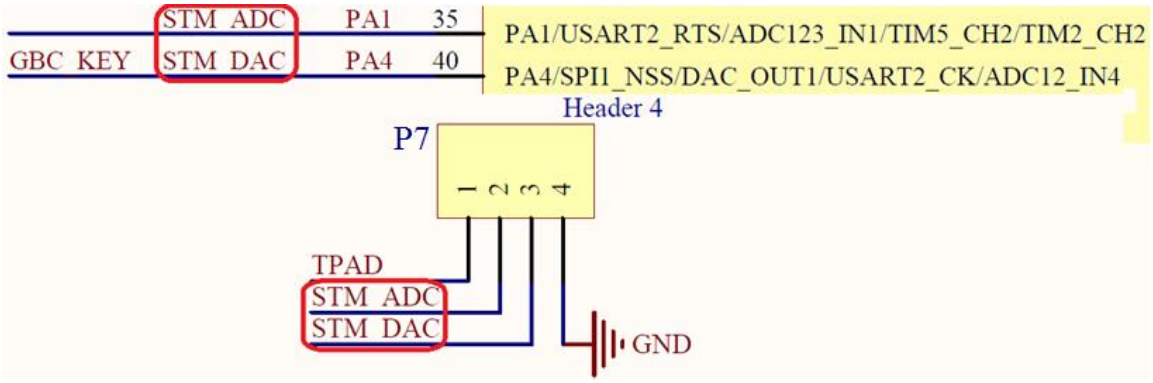


图 25.2.1 ADC、DAC 与 STM32F1 连接原理图

注意：STM_DAC 和 GBC_KEY 共用 PA4，所以，如果您的开发板 ATK MODULE 位置插了其他模块，那么可能影响 DAC 的输出结果，建议在做 DAC 实验的时候，ATK MODULE 位置，不要插任何其他模块。

P7 是 AD/DA 组合接口，我们只需要通过跳线帽短接 P7 的 ADC 和 DAC，就可以开始做本章实验了。如图 25.2.2 所示：

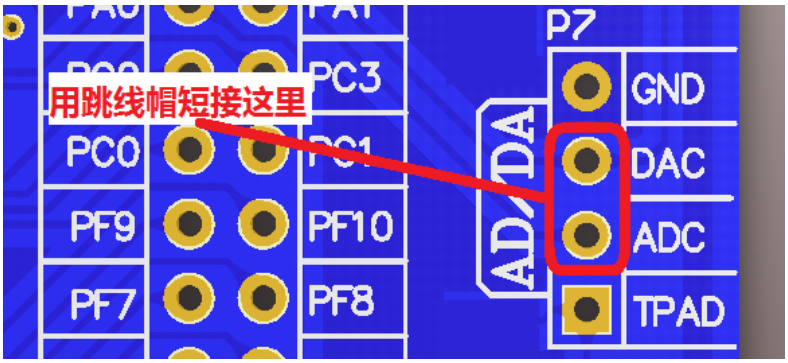


图 25.2.2 硬件连接示意图

25.3 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 stm32f1xx_hal_dac.c 以及包含头文件 stm32f1xx_hal_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_HandleTypeDef DAC1_Handler;//DAC 句柄
//初始化 DAC
void DAC1_Init(void)
{
    DAC_ChannelConfTypeDef DACCH1_Config;
    DAC1_Handler.Instance=DAC;
```

```
        HAL_DAC_Init(&DAC1_Handler);                                    //初始化 DAC
        DACCH1_Config.DAC_Trigger=DAC_TRIGGER_NONE;      //不使用触发功能
        DACCH1_Config.DAC_OutputBuffer=DAC_OUTPUTBUFFER_DISABLE;
                                                                        //DAC1 输出缓冲关闭

HAL_DAC_ConfigChannel(&DAC1_Handler,&DACCH1_Config,DAC_CHANNEL_1);
                                                                        //DAC 通道 1 配置
        HAL_DAC_Start(&DAC1_Handler,DAC_CHANNEL_1);      //开启 DAC 通道 1
    }

    //DAC 底层驱动，时钟配置，引脚 配置
    //此函数会被 HAL_DAC_Init()调用
    //hdac:DAC 句柄
    void HAL_DAC_MspInit(DAC_HandleTypeDef* hdac)
    {
        GPIO_InitTypeDef GPIO_Initure;
        __HAL_RCC_DAC_CLK_ENABLE();                  //使能 DAC 时钟
        __HAL_RCC_GPIOA_CLK_ENABLE();              //开启 GPIOA 时钟
        GPIO_Initure.Pin=GPIO_PIN_4;                  //PA4
        GPIO_Initure.Mode=GPIO_MODE_ANALOG;      //模拟
        GPIO_Initure.Pull=GPIO_NOPULL;                //不带上下拉
        HAL_GPIO_Init(GPIOA,&GPIO_Initure);
    }

    //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;

HAL_DAC_SetValue(&DAC1_Handler,DAC_CHANNEL_1,DAC_ALIGN_12B_R,temp);
                                                        //12 位右对齐数据格式设置 DAC 值
    }
```

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;
u16 dacval=0;
u8 key;
HAL_Init();                                 //初始化 HAL 库
Stm32_Clock_Init(RCC_PLL_MUL9);             //设置时钟,72M
delay_init(72);                             //初始化延时函数
uart_init(115200);                          //初始化串口
usmart_dev.init(84);                        //初始化 USMART
LED_Init();                                 //初始化 LED
KEY_Init();                                 //初始化按键
LCD_Init();                                 //初始化 LCD FSMC 接口
MY_ADC_Init();                              //初始化 ADC1 通道 1
DAC1_Init();                                //初始化 DAC1
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16," ELITE STM32");
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,"2019/9/18");
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");
HAL_DAC_SetValue(&DAC1_Handler,DAC_CHANNEL_1,DAC_ALIGN_12B_R,0);
//初始值为 0
while(1)
{
    t++;
    key=KEY_Scan(0);
    if(key==WKUP_PRES)
    {
        if(dacval<4000)dacval+=200;
        HAL_DAC_SetValue(&DAC1_Handler,DAC_CHANNEL_1,
          DAC_ALIGN_12B_R,dacval);//设置 DAC 值
    }else if(key==2)
    {
        if(dacval>200)dacval-=200;
        else dacval=0;
        HAL_DAC_SetValue(&DAC1_Handler,DAC_CHANNEL_1,
        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=HAL_DAC_GetValue(&DAC1_Handler,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_1,10); //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。

本章，我们还可以利用 USMART 来设置 DAC 的输出电压值，故需要将 Dac1_Set_Vol 函数加入 USMART 控制，方法前面已经有详细的介绍了，大家这里自行添加，或者直接查看我们光盘的源码。

从 main 函数代码可以看出，按键设置输出电压的时候，每次都是以 0.161V 递增或递减的，而通过 USMART 调用 Dac1_Set_Vol 函数，则可以实现任意电平输出控制（当然得在 DAC 可控范围内）。

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 精英 STM32F103 上，可以看到 LCD 显示如图 25.4.1 所示：

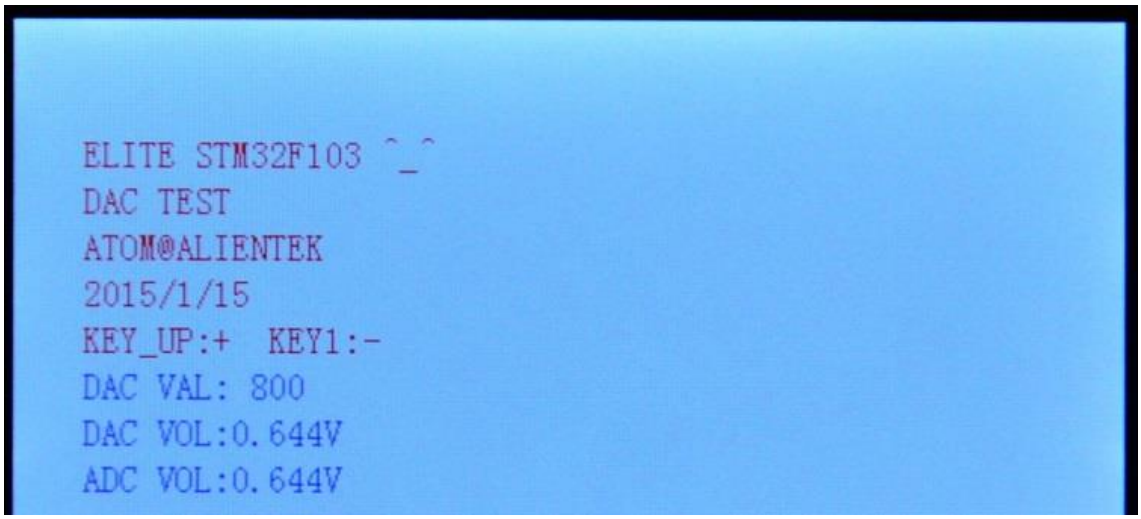


图 25.4.1 DAC 实验测试图

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。

大家可以试试在 USMART 调用 Dac1_Set_Vol 函数，来设置 DAC 通道 1 的输出电压，如图 25.4.2 所示：

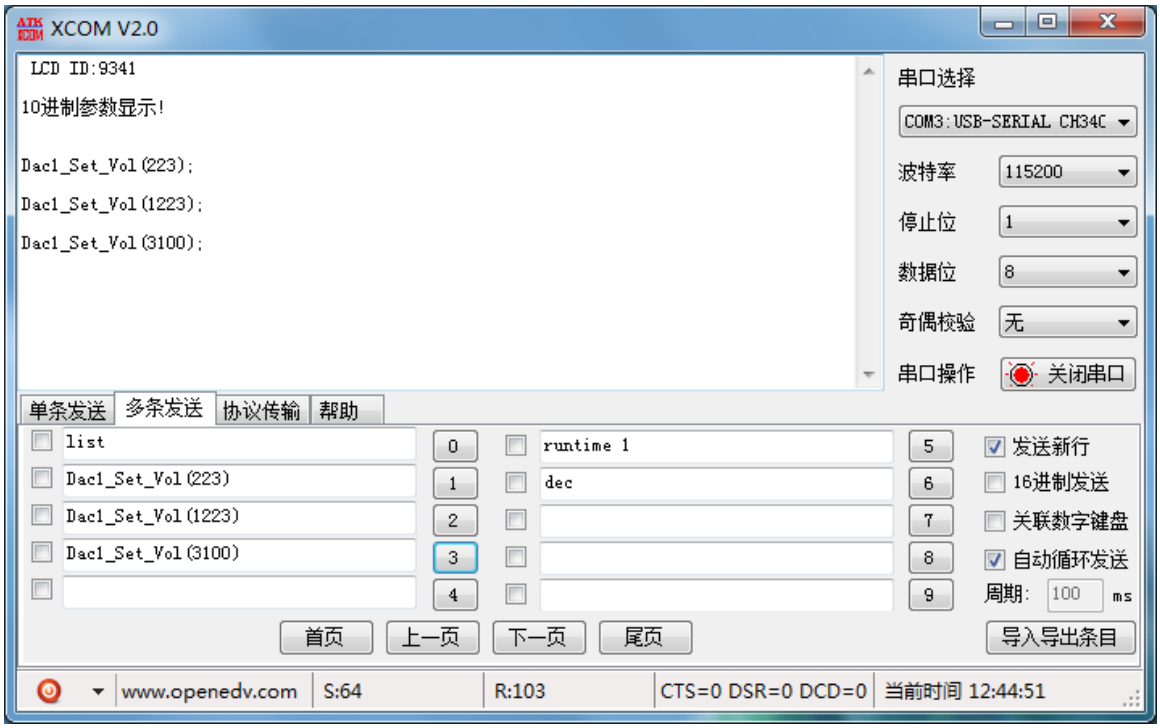


图 25.4.2 通过 usmart 设置 DAC 通道 1 的电压输出

第二十六章 DMA 实验

本章我们将向大家介绍 STM32F1 的 DMA。在本章中，我们将利用 STM32F1 的 DMA 来实现串口数据传送，并在 TFTLCD 模块上显示当前的传送进度。本章分为如下几个部分：

26.1 STM32F1 DMA 简介

26.2 硬件设计

26.3 软件设计

26.4 下载验证

26.1 STM32F1 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。

STM32 最多有 2 个 DMA 控制器（DMA2 仅存在大容量产品中），DMA1 有 7 个通道。DMA2 有 5 个通道。每个通道专门用来管理来自于一个或多个外设对存储器访问的请求。还有一个仲裁起来协调各个 DMA 请求的优先权。

STM32 的 DMA 有以下一些特性：

●每个通道都直接连接专用的硬件 DMA 请求，每个通道都同样支持软件触发。这些功能通过软件来配置。

●在七个请求间的优先权可以通过软件编程设置(共有四级：很高、高、中等和低)，假如在相等优先权时由硬件决定(请求 0 优先于请求 1，依此类推) 。

●独立的源和目标数据区的传输宽度(字节、半字、全字)，模拟打包和拆包的过程。源和目标地址必须按数据传输宽度对齐。

●支持循环的缓冲器管理

●每个通道都有 3 个事件标志(DMA 半传输，DMA 传输完成和 DMA 传输出错)，这 3 个事件标志逻辑或成为一个单独的中断请求。

●存储器和存储器间的传输

●外设和存储器，存储器和外设的传输

●闪存、SRAM、外设的 SRAM、APB1 APB2 和 AHB 外设均可作为访问的源和目标。

●可编程的数据传输数目：最大为 65536

STM32F103ZET6 有两个 DMA 控制器，DMA1 和 DMA2，本章，我们仅针对 DMA1 进行介绍。

从外设（TIMx、ADC、SPIx、I2Cx 和 USARTx）产生的 DMA 请求，通过逻辑或输入到 DMA 控制器，这就意味着同时只能有一个请求有效。外设的 DMA 请求，可以通过设置相应的外设寄存器中的控制位，被独立地开启或关闭。

表 26.1.1 是 DMA1 各通道一览表：

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC	ADC1						
SPI		SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

表 26.1.1 DMA1 个通道一览表

这里解释一下上面说的逻辑或，例如通道 1 的几个 DMA1 请求（ADC1、TIM2_CH3、TIM4_CH1），这几个是通过逻辑或到通道 1 的，这样我们在同一时间，就只能使用其中的一个。其他通道也是类似的。

这里我们要使用的是串口 1 的 DMA 传送，也就是要用到通道 4。接下来，我们介绍一下 DMA

设置相关的几个寄存器。

第一个是 DMA 中断状态寄存器（DMA_ISR）。该寄存器的各位描述如图 26.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5
				r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:28	保留，始终读为0。
位27，23，19，15，11，7，3	**TEIFx**：通道x的传输错误标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有传输错误(TE)； 1：在通道x发生了传输错误(TE)。
位26，22，18，14，10，6，2	**HTIFx**：通道x的半传输标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有半传输事件(HT)； 0：在通道x产生了半传输事件(HT)。
位25，21，17，13，9，5，1	**TCIFx**：通道x的传输完成标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有传输完成事件(TC)； 0：在通道x产生了传输完成事件(TC)。
位24，20，16，12，8，4，0	**GIFx**：通道x的全局中断标志(x = 1 … 7) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有TE、HT或TC事件； 0：在通道x产生了TE、HT或TC事件。

图 26.1.1 DMA_ISR 寄存器各位描述

我们如果开启了 DMA_ISR 中这些中断，在达到条件后就会跳到中断服务函数里面去，即使没开启，我们也可以通过查询这些位来获得当前 DMA 传输的状态。这里我们常用的是 TCIFx，即通道 DMA 传输完成与否的标志。注意此寄存器为只读寄存器，所以在这些位被置位之后，只能通过其他的操作来清除。

第二个是 DMA 中断标志清除寄存器（DMA_IFCR）。该寄存器的各位描述如图 26.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				CTEIF7	CHTIF7	CTCIF7	CGIF7	CTEIF6	CHTIF6	CTCIF6	CGIF6	CTEIF5	CHTIF5	CTCIF5	CGIF5
				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
CTEIF4	CHTIF4	CTCIF4	CGIF4	CTEIF3	CHTIF3	CTCIF3	CGIF3	CTEIF2	CHTIF2	CTCIF2	CGIF2	CTEIF1	CHTIF1	CTCIF1	CGIF1
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留，始终读为0。
位27，23，19，15，11，7，3	**CTEIFx**：清除通道x的传输错误标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 1：清除DMA_ISR寄存器中的对应TEIF标志。
位26，22，18，14，10，6，2	**CHTIFx**：清除通道x的半传输标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应HTIF标志。
位25，21，17，13，9，5，1	**CTCIFx**：清除通道x的传输完成标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应TCIF标志。
位24，20，16，12，8，4，0	**CGIFx**：清除通道x的全局中断标志(x = 1 ... 7) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应的GIF、TEIF、HTIF和TCIF标志。

图 26.1.2 DMA_IFCR 寄存器各位描述

DMA_IFCR 的各位就是用来清除 DMA_ISR 的对应位的，通过写 0 清除。在 DMA_ISR 被置位后，我们必须通过向该位寄存器对应的位写入 0 来清除。

第三个是 DMA 通道 x 配置寄存器（DMA_CCRx）（x=1~7，下同）。该寄存器的我们在这里就不贴出来了，见《STM32 中文参考手册》第 150 页 10.4.3 一节。该寄存器控制着 DMA 的很多相关信息，包括数据宽度、外设及存储器的宽度、通道优先级、增量模式、传输方向、中断允许、使能等都是通过该寄存器来设置的。所以 DMA_CCRx 是 DMA 传输的核心控制寄存器。

第四个是 DMA 通道 x 传输数据量寄存器（DMA_CNDTRx）。这个寄存器控制 DMA 通道 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值会随着传输的进行而减少，当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成了。所以可以通过这个寄存器的值来知道当前 DMA 传输的进度。

第五个是 DMA 通道 x 的外设地址寄存器（DMA_CPARx）。该寄存器用来存储 STM32 外设的地址，比如我们使用串口 1，那么该寄存器必须写入 0x40013804（其实就是&USART1_DR）。如果使用其他外设，就修改成相应外设的地址就行了。

最后一个是 DMA 通道 x 的存储器地址寄存器(DMA_CMARx)，该寄存器和 DMA_CPARx 差不多，但是是用来放存储器的地址的。比如我们使用 SendBuf[5200]数组来做存储器，那么我们在 DMA_CMARx 中写入&SendBuff 就可以了。

DMA 相关寄存器就为大家介绍到这里，此节我们要用到串口 1 的发送，属于 DMA1 的通道 4，接下来我们就介绍下 DMA1 通道 4 的配置步骤：

1）使能 DMA1 时钟。

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。HAL 库方法为：

```
__HAL_RCC_DMA1_CLK_ENABLE();//DMA1 时钟使能
```

2）初始化 DMA1 数据流 4，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 HAL_DMA_Init 函数实现的，该函数声明为：

```
HAL_StatusTypeDef HAL_DMA_Init(DMA_HandleTypeDef *hdma);
```

该函数只有一个 DMA_HandleTypeDef 结构体指针类型入口参数，结构体定义为：

```
typedef struct __DMA_HandleTypeDef
{
    DMA_Stream_TypeDef              *Instance;
    DMA_InitTypeDef                 Init;
    HAL_LockTypeDef                 Lock;
    __IO HAL_DMA_StateTypeDef       State;
    void                            *Parent;
    void                            (* XferCpltCallback)(
    struct __DMA_HandleTypeDef * hdma);
    void                             (* XferHalfCpltCallback)(
    struct __DMA_HandleTypeDef * hdma);
    void                            (* XferM1CpltCallback)(
     struct __DMA_HandleTypeDef * hdma);
    void                            (* XferErrorCallback)(
    struct __DMA_HandleTypeDef * hdma);
    __IO uint32_t                   ErrorCode;
    uint32_t                        StreamBaseAddress;
    uint32_t                        StreamIndex;
}DMA_HandleTypeDef;
```

成员变量 Instance 是用来设置寄存器基地址，例如要设置为 DMA1 的通道 4，那么取值为 DMA1_Channel4。

成员变量 Parent 是 HAL 库处理中间变量，用来指向 DMA 通道外设句柄。

成员变量 XferCpltCallback（传输完成回调函数），XferHalfCpltCallback（半传输完成回调函数），XferM1CpltCallback（Memory1 传输完成回调函数）和 XferErrorCallback（传输错误回调函数）是四个函数指针，用来指向回调函数入口地址。

成员变量 StreamBaseAddress 和 StreamIndex 是数据流基地址和索引号，这个是 HAL 库处理的时候会自动计算，用户无需设置。

其他成员变量 HAL 库处理过程状态标识变量，这里就不做过多讲解。接下来我们着重看看成员变量 Init，它是 DMA_InitTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
    uint32_t Direction;//传输方向，例如存储器到外设 DMA_MEMORY_TO_PERIPH
   uint32_t PeriphInc;//外设（非）增量模式，非增量模式 DMA_PINC_DISABLE
   uint32_t MemInc;//存储器（非）增量模式，增量模式 DMA_MINC_ENABLE
   uint32_t PeriphDataAlignment; //外设数据大小：8/16/32 位。
   uint32_t MemDataAlignment;   //存储器数据大小：8/16/32 位。
   uint32_t Mode;//模式：外设流控模式/循环模式/普通模式
   uint32_t Priority;   //DMA 优先级：低/中/高/非常高
```

```
}DMA_InitTypeDef;
```

该结构体成员变量非常多，但是每个成员变量配置的基本都是 DMA_SxCR 寄存器和 DMA_SxFCR 寄存器的相应位。我们把结构体各个成员变量的含义都通过注释的方式列出来了。例如本实验我们要用到 DMA2_ DMA1_Channel4，把内存中数组的值发送到串口外设发送寄存器 DR，所以方向为存储器到外设 DMA_MEMORY_TO_PERIPH，一个一个字节发送，需要数字索引自动增加，所以是存储器增量模式 DMA_MINC_ENABLE，存储器和外设的字宽都是字节 8 位。具体配置如下：

```
DMA_HandleTypeDef  UART1TxDMA_Handler;          //DMA 句柄
UART1TxDMA_Handler.Instance= DMA1_Channel4;     //通道选择
UART1TxDMA_Handler.Init.Direction=DMA_MEMORY_TO_PERIPH; //存储器到外设
UART1TxDMA_Handler.Init.PeriphInc=DMA_PINC_DISABLE;  //外设非增量模式
UART1TxDMA_Handler.Init.MemInc=DMA_MINC_ENABLE; //存储器增量模式
UART1TxDMA_Handler.Init.PeriphDataAlignment=DMA_PDATAALIGN_BYTE;
                                                 //外设数据长度:8 位
UART1TxDMA_Handler.Init.MemDataAlignment=DMA_MDATAALIGN_BYTE;
                                                 //存储器数据长度:8 位
UART1TxDMA_Handler.Init.Mode=DMA_NORMAL;        //外设普通模式
UART1TxDMA_Handler.Init.Priority=DMA_PRIORITY_MEDIUM; //中等优先级
```

这里大家要注意，HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。例如要使用串口 DMA 发送，所以方式为：

```
__HAL_LINKDMA(&UART1_Handler,hdmatx,UART1TxDMA_Handler);
```

其中 UART1_Handler 是串口初始化句柄，我们在 usart.c 中定义过了。UART1TxDMA_Handler 是 DMA 初始化句柄。hdmatx 是外设句柄结构体的成员变量，在这里实际就是 UART1_Handler 的成员变量。在 HAL 库中，任何一个可以使用 DMA 的外设，它的初始化结构体句柄都会有一个 DMA_HandleTypeDef 指针类型的成员变量，是 HAL 库用来做相关指向的。Hdmatx 就是 DMA_HandleTypeDef 结构体指针类型。

这句话的含义就是把 UART1_Handler 句柄的成员变量 hdmatx 和 DMA 句柄 UART1TxDMA_Handler 连接起来，是纯软件处理，没有任何硬件操作。

这里我们就点到为止，如果大家要详细了解 HAL 库指向关系，请查看本实验宏定义标识符__HAL_LINKDMA 的定义和调用方法，就会很清楚了。

3）使能串口 1 的 DMA 发送

在实验中，开启一次 DMA 传输传输函数如下：

```
//开启一次 DMA 传输
//huart:串口句柄
//pData：传输的数据指针
//Size:传输的数据量
void MYDMA_USART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
{
    HAL_DMA_Start(huart->hdmatx, (u32)pData, (uint32_t)&huart->Instance->DR, Size);
    //开启 DMA 传输
    huart->Instance->CR3 |= USART_CR3_DMAT;//使能串口 DMA 发送
}
```

HAL 库还提供了对串口的 DMA 发送的停止，暂停，继续等操作函数：

```
HAL_StatusTypeDef HAL_UART_DMAStop(UART_HandleTypeDef *huart);      //停止
HAL_StatusTypeDef HAL_UART_DMAPause(UART_HandleTypeDef *huart);     //暂停
HAL_StatusTypeDef HAL_UART_DMAResume(UART_HandleTypeDef *huart);//恢复
```

这些函数使用方法这里我们就不累赘了。

4）使能 DMA1 数据流 4，启动传输。

使能 DMA 数据流的函数为：

```
HAL_StatusTypeDef HAL_DMA_Start(DMA_HandleTypeDef *hdma, uint32_t SrcAddress,
                                uint32_t DstAddress, uint32_t DataLength);
```

这个函数比较好理解，第一个参数是 DMA 句柄，第二个是传输源地址，第三个是传输目标地址，第四个是传输的数据长度。

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，使用的方法是：

```
__HAL_DMA_GET_FLAG(&UART1TxDMA_Handler,DMA_FLAG_TCIF3_7);
```

获取当前传输剩余数据量：

```
__HAL_DMA_GET_COUNTER(&UART1TxDMA_Handler);
```

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。

6）DMA 中断使用方法

DMA 中断对于每个流都有一个中断服务函数，比如 DMA1_Channel4 的中断服务函数为 DMA1_Channel4_IRQHandler。同样，HAL 库也提供了一个通用的 DMA 中断处理函数 HAL_DMA_IRQHandler，在该函数内部，会对 DMA 传输状态进行分析，然后调用相应的中断处理回调函数：

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);//发送完成回调函数
void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart);/发送一半回调函数
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);//接收完成回调函数
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart);//接收一半回调函数
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart);//传输出错回调函数
```

对于串口 DMA 开启，使能数据流，启动传输，这些步骤，如果使用了中断，可以直接调用 HAL 库函数 HAL_USART_Transmit_DMA，该函数声明如下：

```
HAL_StatusTypeDef HAL_USART_Transmit_DMA(USART_HandleTypeDef *husart,
uint8_t *pTxData, uint16_t Size);
```

DMA 相关的库函数我们就讲解到这里，大家可以查看 HAL 库手册详细了解。

26.2 硬件设计

所以本章用到的硬件资源有：

1） 指示灯 DS0

2） KEY0 按键

3） 串口

4） TFTLCD 模块

5） DMA

本章我们将利用外部按键 KEY0 来控制 DMA 的传送，每按一次 KEY0，DMA 就传送一次数据到 USART1，然后在 TFTLCD 模块上显示进度等信息。DS0 还是用来做为程序运行的指示灯。

本章实验需要注意 P3 口的 RXD 和 TXD 是否和 PA9 和 PA10 连接上，如果没有，请先连接。

26.3 软件设计

打开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 stm32f1xx_hal_dma.c，同时引入了 stm32f1xx_hal_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：

```
DMA_HandleTypeDef  UART1TxDMA_Handler;          //DMA 句柄
//DMA1 的各通道配置
//这里的传输形式是固定的,这点要根据不同的情况来修改
//从存储器->外设模式/8 位数据宽度/存储器增量模式
//chx:DMA 通道选择,DMA1_Channel1~DMA1_Channel7
void MYDMA_Config(DMA_Channel_TypeDef *chx)
{
    __HAL_RCC_DMA1_CLK_ENABLE();                //DMA1 时钟使能
    __HAL_LINKDMA(&UART1_Handler,hdmatx,UART1TxDMA_Handler);
                                    //将 DMA 与 USART1 联系起来(发送 DMA)
    //Tx DMA 配置
    UART1TxDMA_Handler.Instance=chx;                        //通道选择
    UART1TxDMA_Handler.Init.Direction=DMA_MEMORY_TO_PERIPH; //存储器到外设
    UART1TxDMA_Handler.Init.PeriphInc=DMA_PINC_DISABLE;  //外设非增量模式
    UART1TxDMA_Handler.Init.MemInc=DMA_MINC_ENABLE;  //存储器增量模式
    UART1TxDMA_Handler.Init.PeriphDataAlignment=DMA_PDATAALIGN_BYTE;
                                                        //外设数据长度:8 位
    UART1TxDMA_Handler.Init.MemDataAlignment=DMA_MDATAALIGN_BYTE;
                                                    //存储器数据长度:8 位
    UART1TxDMA_Handler.Init.Mode=DMA_NORMAL;            //外设普通模式
    UART1TxDMA_Handler.Init.Priority=DMA_PRIORITY_MEDIUM; //中等优先级
    HAL_DMA_DeInit(&UART1TxDMA_Handler);
    HAL_DMA_Init(&UART1TxDMA_Handler);
}

//开启一次 DMA 传输
//huart:串口句柄
//pData：传输的数据指针
//Size:传输的数据量
void MYDMA_USART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
{
    HAL_DMA_Start(huart->hdmatx, (u32)pData, (uint32_t)&huart->Instance->DR, Size);
    //开启 DMA 传输
    huart->Instance->CR3 |= USART_CR3_DMAT;//使能串口 DMA 发送
}
```

该部分代码仅仅 2 个函数，MYDMA_Config 函数，基本上就是按照我们上面介绍的步骤来初始化 DMA 的，该函数在外部只能修改通道、源地址、目标地址和传输数据量等几个参数，更多的其他设置只能在该函数内部修改。

MYDMA_Enable 函数用来产生一次 DMA 传输，该函数每执行一次，DMA 就发送一次。

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：

```
#define SEND_BUF_SIZE 8200
//发送数据长度,最好等于 sizeof(TEXT_TO_SEND)+2 的整数倍.
u8 SendBuff[SEND_BUF_SIZE]; //发送数据缓冲区
const u8 TEXT_TO_SEND[]={"ALIENTEK WarShip STM32F1 DMA  串口实验"};
int main(void)
{
    u16 i;
    u8 t=0;
    u8 j,mask=0;
    float pro=0;
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    usmart_dev.init(84);                    //初始化 USMART
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    LCD_Init();                             //初始化 LCD FSMC 接口
    MYDMA_Config(DMA1_Channel4);            //初始化 DMA1 通道 4
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/18");
    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]=0x0d;
                mask++;
```

```
            }
        }else//复制 TEXT_TO_SEND 语句
        {
            mask=0;
            SendBuff[i]=TEXT_TO_SEND[t];
            t++;
        }
    }
    POINT_COLOR=BLUE;//设置字体为蓝色
    i=0;
     while(1)
    {
        t=KEY_Scan(0);
        if(t==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
            HAL_UART_Transmit_DMA(&UART1_Handler,SendBuff,SEND_BUF_SIZE);
                                                              //启动传输
            //使能串口 1 的 DMA 发送  //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(__HAL_DMA_GET_FLAG(&UART1TxDMA_Handler,DMA_FLAG_TC4))
                //等待 DMA1 通道 4 传输完成
             {
            __HAL_DMA_CLEAR_FLAG(&UART1TxDMA_Handler,DMA_FLAG_TC4);
                //清除 DMA1 通道 4 传输完成标志
                HAL_UART_DMAStop(&UART1_Handler);
                //传输完成以后关闭串口 DMA
                break;
             }
                pro=__HAL_DMA_GET_COUNTER(&UART1TxDMA_Handler);
                //得到当前还剩余多少个数据
                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!");//提示传送完成
        }
```

```
            i++;
            delay_ms(10);
            if(i==20)
            {
                LED0=!LED0;//提示系统正在运行
                i=0;
            }
        }
    }
```

main 函数的流程大致是：先初始化内存 SendBuff 的值，然后通过 KEY0 开启串口 DMA 发送，在发送过程中，通过__HAL_DMA_GET_COUNTER()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。这里还有一点要注意，因为是使用的串口 1 DMA 发送，所以代码中使用 HAL_UART_Transmit_DMA 函数开启串口的 DMA 发送：

至此，DMA 串口传输的软件设计就完成了。

26.4 下载验证

在代码编译成功之后，我们通过串口下载代码到 ALIENTEK 精英 STM32F103 上，可以看到 LCD 显示如图 26.4.1 所示：

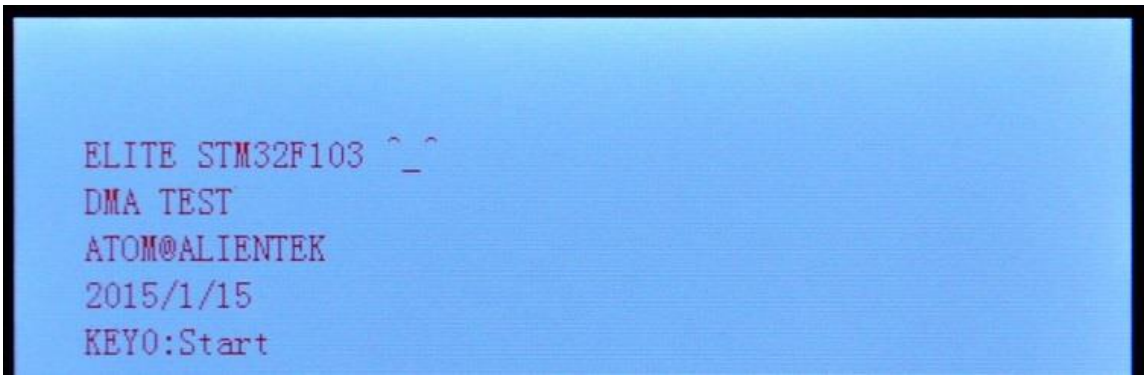


图 26.4.1 DMA 实验测试图

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 26.4.2 所示的内容：

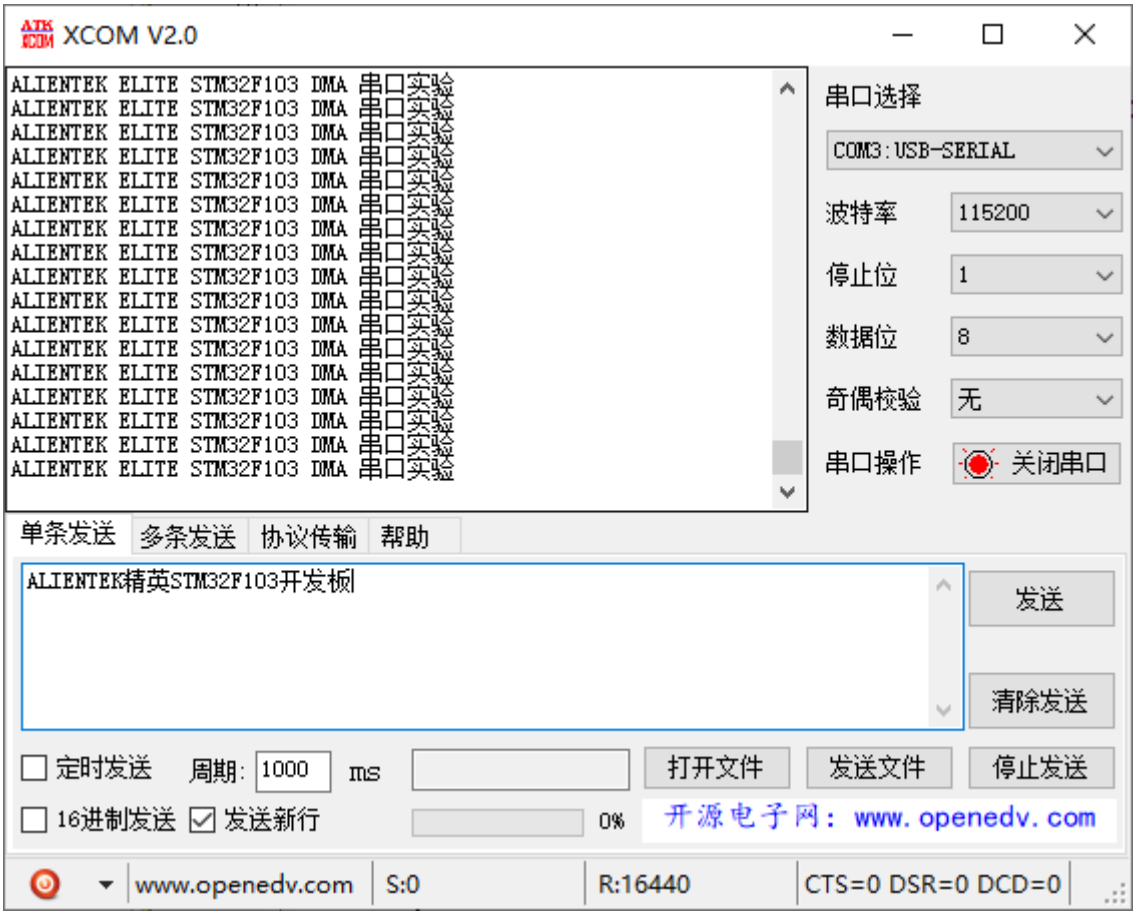
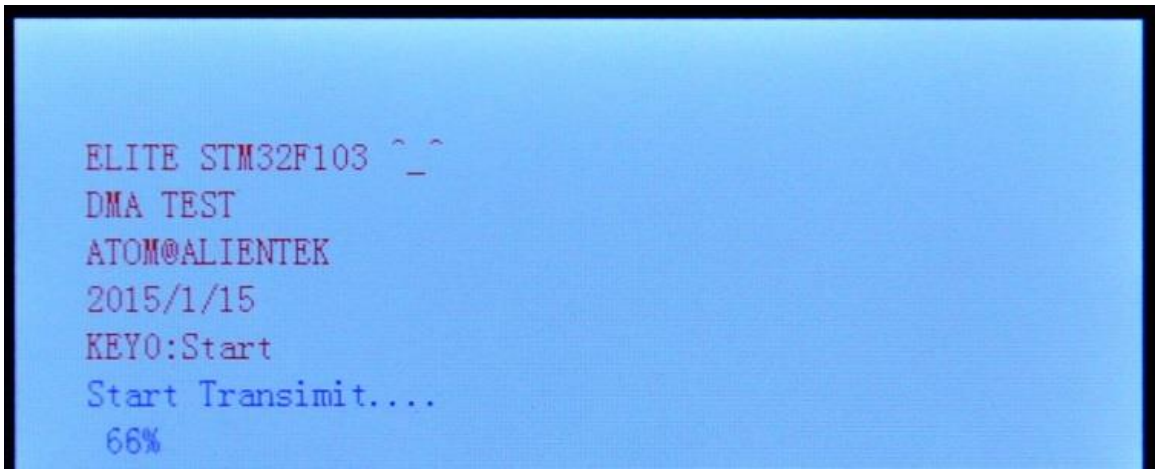


图 26.4.2 串口收到的数据内容

可以看到串口收到了精英 STM32F103 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 26.4.3 所示：



26.4.3 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 STM32F1 的 DMA 使用。DMA 是个非常好的功能，它不但能减轻 CPU 负担，还能提高数据传输速度，合理的应用 DMA，往往能让你的程序设计变得简单。

第二十七章 IIC 实验

本章我们将向大家介绍如何使用 STM32F1 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 STM32F1 的普通 IO 口模拟 IIC 时序，来实现 24C02 的读写，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

27.1 IIC 简介
27.2 硬件设计
27.3 软件设计
27.4 下载验证

27.1 IIC 简介

IIC(Inter－Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，高速 IIC 总线一般可达 400kbps 以上。

I2C 总线在传送数据过程中共有三种类型信号， 它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。

应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

这些信号中，起始信号是必需的，结束信号和应答信号，都可以不要。IIC 总线时序图如图 27.1.1 所示：

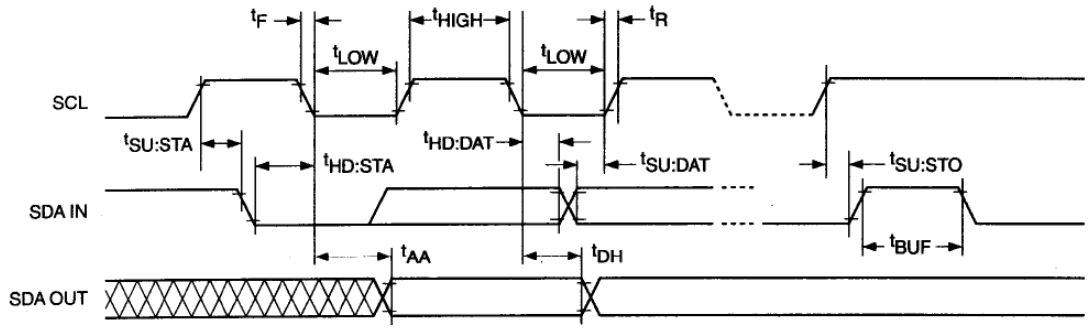


图 27.1.1 IIC 总线时序图

ALIENTEK 精英 STM32F103 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 STM32F1 来实现 24C02 的读写。

目前大部分 MCU 都带有 IIC 总线接口，STM32F1 也不例外。但是这里我们不使用 STM32F1 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 STM32F1 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，这也是我们推荐使用软件模拟 IIC 的另外一个原因。

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。

27.2 硬件设计

本章需要用到的硬件资源有：

1） 指示灯 DS0

2） KEY0 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 STM32F1 的连接，24C02 的 SCL 和 SDA 分别连在 STM32F1 的 PB6 和 PB7 上的，连接关系如图 27.2.1 所示：

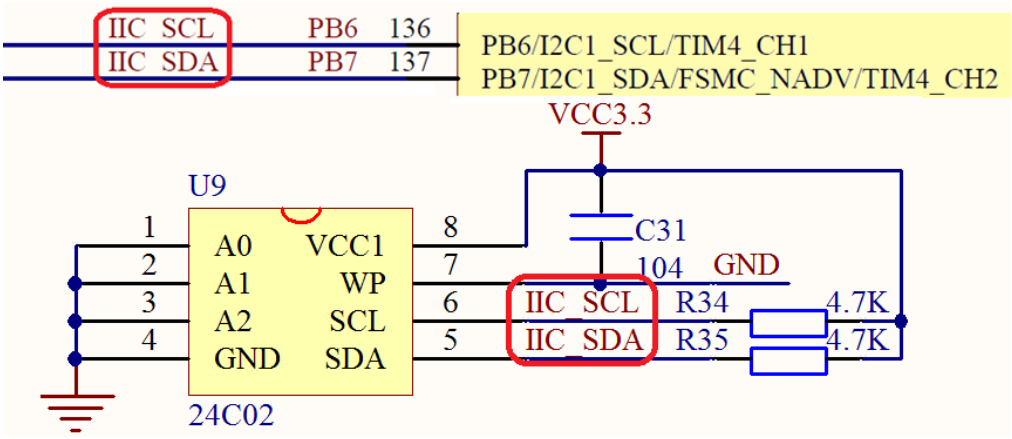


图 27.2.1 STM32F1 与 24C02 连接图

27.3 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 HAL 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：

```
//IIC 初始化
void IIC_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();     //使能 GPIOB 时钟
    //PH4,5 初始化设置
    GPIO_Initure.Pin=GPIO_PIN_6|GPIO_PIN_7;
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);
    IIC_SDA=1;
    IIC_SCL=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->CRL&=0X0FFFFFFF;GPIOB->CRL|=(u32)8<<28;}
                                                    //PB7 输入模式
#define SDA_OUT() {GPIOB->CRL&=0X0FFFFFFF;GPIOB->CRL|=(u32)3<<28;}
                                                    //PB7 输出模式
//IO 操作
#define IIC_SCL    PBout(6) //SCL
#define IIC_SDA    PBout(7) //SDA
#define READ_SDA   PBin(7)  //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);
}
//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[]={"ELITE STM32 IIC TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key;
    u16 i=0;
    u8 datatemp[SIZE];
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    usmart_dev.init(84);                    //初始化 USMART
```

```
    LED_Init();                                 //初始化 LED
    KEY_Init();                                 //初始化按键
    LCD_Init();                                 //初始化 LCD FSMC 接口
    AT24CXX_Init();                             //初始化 IIC
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/18");
    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:   ");//提示传送完成
            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。

最后，我们将 AT24CXX_WriteOneByte 和 AT24CXX_ReadOneByte 函数加入 USMART 控制，这样，我们就可以通过串口调试助手，读写任何一个 24C02 的地址，方便测试。

至此，我们的软件设计部分就结束了。

27.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32F103 上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 27.4.1 所示：

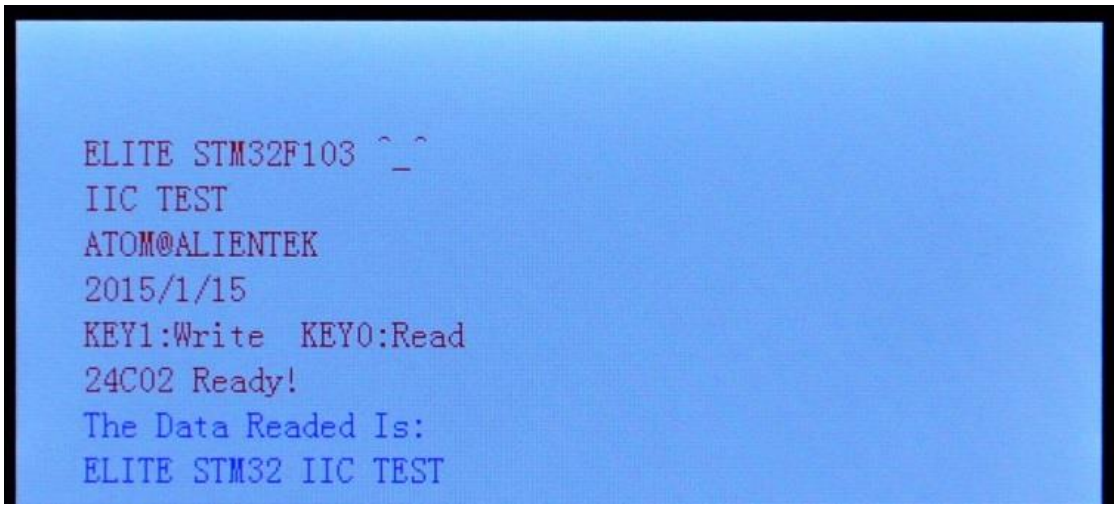


图 27.4.1 IIC 实验程序运行效果图

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 PB6 和 PB7 短接就可以看到报错了。

USMART 测试 24C02 的任意地址（地址范围：0~255）读写如图 23.4.2 所示：



图 27.4.2 USMART 控制 24C02 读写

第二十八章 SPI 实验

本章我们将向大家介绍 STM32F1 的 SPI 功能。在本章中，我们将使用 STM32F1 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：

28.1 SPI 简介
28.2 硬件设计
28.3 软件设计
28.4 下载验证

28.1 SPI 简介

SPI 是英语 Serial Peripheral interface 的缩写，顾名思义就是串行外围设备接口。是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。SPI 接口主要应用在 EEPROM，FLASH，实时时钟，AD 转换器，还有数字信号处理器和数字信号解码器之间。SPI 是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议，STM32F1 也有 SPI 接口。

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。

SPI 主要特点有：可以同时发出和接收串行数据；可以当作主机或从机工作；提供频率可编程时钟；发送结束中断标志；写冲突保护；总线竞争保护等。

SPI 总线四种工作方式 SPI 模块为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性和相位可以进行配置，时钟极性（CPOL）对传输协议没有重大的影响。如果 CPOL=0，串行同步时钟的空闲状态为低电平；如果 CPOL=1，串行同步时钟的空闲状态为高电平。时钟相位（CPHA）能够配置用于选择两种不同的传输协议之一进行数据传输。如果 CPHA=0，在串行同步时钟的第一个跳变沿（上升或下降）数据被采样；如果 CPHA=1，在串行同步时钟的第二个跳变沿（上升或下降）数据被采样。SPI 主模块和与之通信的外设备时钟相位和极性应该一致。

不同时钟相位下的总线数据传输时序如图 28.1.1 所示：

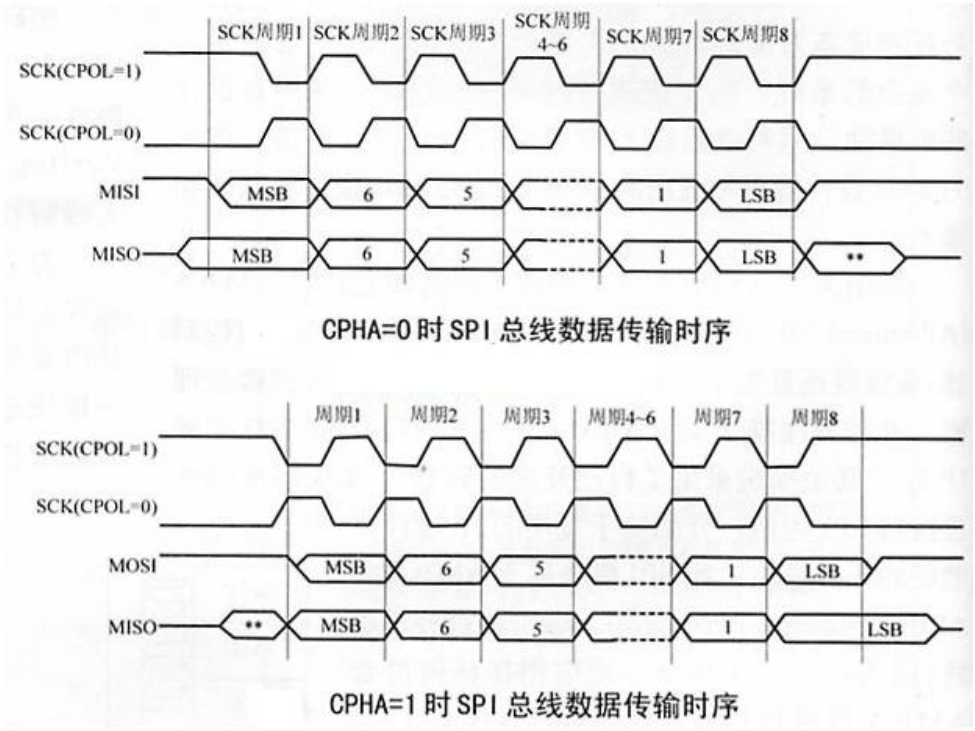


图 28.1.1 不同时钟相位下的总线传输时序（CPHA=0/1）

STM32F1 的 SPI 功能很强大，SPI 时钟最高可以到 18Mhz，支持 DMA，可以配置为 SPI

协议或者 I2S 协议。

本章，我们将使用 STM32 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，STM32 的 SPI 详细介绍请参考《STM32 中文参考手册》第 457 页，23 节。然后我们再简单介绍下 SPI FLASH 芯片。

这节，我们使用 STM32 的 SPI2 的主模式，下面就来看看 SPI2 部分的设置步骤吧。SPI 相关的库函数和定义分布在文件 stm32f1xx_spi.c 以及头文件 stm32f1xx_spi.h 中。STM32 的主模式配置步骤如下：

1）配置相关引脚的复用功能，使能 SPI2 时钟。

我们要用 SPI2，第一步就要使能 SPI2 的时钟，SPI2 的时钟通过 APB1ENR 的第 14 位来设置。其次要设置 SPI2 的相关引脚为复用输出，这样才会连接到 SPI2 上否则这些 IO 口还是默认的状态，也就是标准输入输出口。这里我们使用的是 PB13、14、15 这 3 个（SCK.、MISO、MOSI，CS 使用软件管理方式），所以设置这三个为复用功能 IO。

使能 SPI2 时钟的方法为：

```
__HAL_RCC_SPI2_CLK_ENABLE();            //使能 SPI2 时钟
```

复用 PB13,14,15 为 SPI2 引脚通过 HAL_GPIO_Init 函数实现，代码如下：

```
GPIO_Initure.Pin=GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
GPIO_Initure.Mode=GPIO_MODE_AF_PP;                  //复用推挽输出
GPIO_Initure.Pull=GPIO_PULLUP;                      //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;            //快速
HAL_GPIO_Init(GPIOB,&GPIO_Initure);
```

2）设置 SPI2 工作模式。

这一步全部是通过 SPI2_CR1 来设置，我们设置 SPI2 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 SPI2 的时钟频率（最大 18Mhz），以及数据的格式（MSB 在前还是 LSB 在前）。在 HAL 库中初始化 SPI 的函数为：

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

下面我们来看看 SPI_HandleTypeDef 定义：

```
typedef struct __SPI_HandleTypeDef
{
  SPI_TypeDef                *Instance;          //基地址
  SPI_InitTypeDef            Init;              //初始化接哦固体
  uint8_t                    *pTxBuffPtr;       //发送缓存
  uint16_t                   TxXferSize;        /发送数据大小
  __IO uint16_t              TxXferCount;       //还剩余多少个数据要发送
  uint8_t                    *pRxBuffPtr;       //接收缓存
  uint16_t                   RxXferSize;        //接收数据大小
  __IO uint16_t              RxXferCount;       //还剩余多少个数据要接收
  void                       (*RxISR)(struct __SPI_HandleTypeDef *hspi);
  void                       (*TxISR)(struct __SPI_HandleTypeDef *hspi);
  DMA_HandleTypeDef          *hdmatx;           //DMA 发送句柄
  DMA_HandleTypeDef          *hdmarx;           //DMA 接收句柄
  HAL_LockTypeDef            Lock;
  __IO HAL_SPI_StateTypeDef  State;
  __IO uint32_t              ErrorCode;
```

```
}SPI_HandleTypeDef;
```

该结构体和串口句柄结构体类似，同样有 6 个成员变量和 2 个 DMA_HandleTypeDef 指针类型变量。这几个参数的作用这里我们就不做过多讲解，大家如果对 HAL 库串口通信理解了，那么这些就很好理解。这里我们主要讲解第二个成员变量 Init，它是 SPI_InitTypeDef 结构体类型，该结构体定义如下：

```
typedef struct
{
  uint32_t Mode;         // 模式：主（SPI_MODE_MASTER），从（SPI_MODE_SLAVE）
  uint32_t Direction;    //方式：   只接受模式，单线双向通信数据模式，全双工
  uint32_t DataSize;     //8 位还是 16 位帧格式选择项
  uint32_t CLKPolarity; //时钟极性
  uint32_t CLKPhase;   //时钟相位
  uint32_t NSS;          //SS 信号由硬件（NSS 管脚）还是软件控制
  uint32_t BaudRatePrescaler; //设置 SPI 波特率预分频值
  uint32_t FirstBit;     //起始位是 MSB 还是 LSB
  uint32_t TIMode;       //帧格式  SPI motorola 模式还是 TI 模式
  uint32_t CRCCalculation;  //硬件 CRC 是否使能
  uint32_t CRCPolynomial;  //CRC 多项式
}SPI_InitTypeDef;
```

该结构体个个成员变量的含义我们已经在成员变量后面注释了，请大家参考学习。SPI 初始化实例代码如下：

```
SPI1_Handler.Instance= SPI2;                          // SPI2
SPI1_Handler.Init.Mode=SPI_MODE_MASTER;    //设置 SPI 工作模式，设置为主模式
SPI1_Handler.Init.Direction=SPI_DIRECTION_2LINES;
//设置 SPI 单向或者双向的数据模式:SPI 设置为双线模式
SPI1_Handler.Init.DataSize=SPI_DATASIZE_8BIT;
//设置 SPI 的数据大小:SPI 发送接收 8 位帧结构
SPI1_Handler.Init.CLKPolarity=SPI_POLARITY_HIGH;
//串行同步时钟的空闲状态为高电平
SPI1_Handler.Init.CLKPhase=SPI_PHASE_2EDGE;
//串行同步时钟的第二个跳变沿（上升或下降）数据被采样
SPI1_Handler.Init.NSS=SPI_NSS_SOFT; //NSS 信号由硬件（NSS 管脚）还是软件
//（使用 SSI 位）管理:内部 NSS 信号有 SSI 位控制
SPI1_Handler.Init.BaudRatePrescaler=SPI_BAUDRATEPRESCALER_256;
//定义波特率预分频的值:波特率预分频值为 256
SPI1_Handler.Init.FirstBit=SPI_FIRSTBIT_MSB;
//指定数据传输从 MSB 位还是 LSB 位开始:数据传输从 MSB 位开始
SPI1_Handler.Init.TIMode=SPI_TIMODE_DISABLE;       //关闭 TI 模式
SPI1_Handler.Init.CRCCalculation=SPI_CRCCALCULATION_DISABLE;
//关闭硬件 CRC 校验
SPI1_Handler.Init.CRCPolynomial=7;                    //CRC 值计算的多项式
HAL_SPI_Init(&SPI2_Handler);//初始化
```

同样，HAL 库也提供了 SPI 初始化 MSP 回调函数 HAL_SPI_MspInit，定义如下：

```
void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi);
```

关于回调函数使用，这里我们就不做过多讲解。

3）使能 SPI2。

这一步通过 SPI2_CR1 的 bit6 来设置，以启动 SPI2，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：

```
__HAL_SPI_ENABLE(&SPI2_Handler);                    //使能 SPI2
```

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，HAL 库提供的发送数据函数原型为：

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                   uint16_t Size, uint32_t Timeout);
```

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。

HAL 库提供的接受数据函数原型为：

```
HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                  uint16_t Size, uint32_t Timeout);
```

这个函数也不难理解，从 SPIx 数据寄存器读出接受到的数据。

前面我们讲解了 SPI 通信的原理，因为 SPI 是全双工，发送一个字节的同时接受一个字节，发送和接收同时完成，所以 HAL 也提供了一个发送接收统一函数：

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData,
                                          uint8_t *pRxData, uint16_t Size, uint32_t Timeout);
```

该函数发送一个字节的同时负责接收一个字节。

5）设置 SPI 传输速度

SPI 初始化结构体 SPI_InitTypeDef 有一个成员变量是 BaudRatePrescaler，该成员变量用来设置 SPI 的预分频系数，从而决定了 SPI 的传输速度。但是 HAL 库并没有提供单独的 SPI 分频系数修改函数，如果我们需要在程序中不时的修改速度，那么我们就要通过设置 SPI 的 CR1 寄存器来修改，具体实现方法请参考后面软件设计小节相关函数。

SPI2 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。

28.2 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） KEY0 和 KEY1 按键
3） TFTLCD 模块
4） SPI
5） W25Q128

这里只介绍 W25Q128 与 STM32F1 的连接，板上的 W25Q128 是直接连在 STM32F1 的 SPI2 上的，连接关系如图 28.2.1 所示：

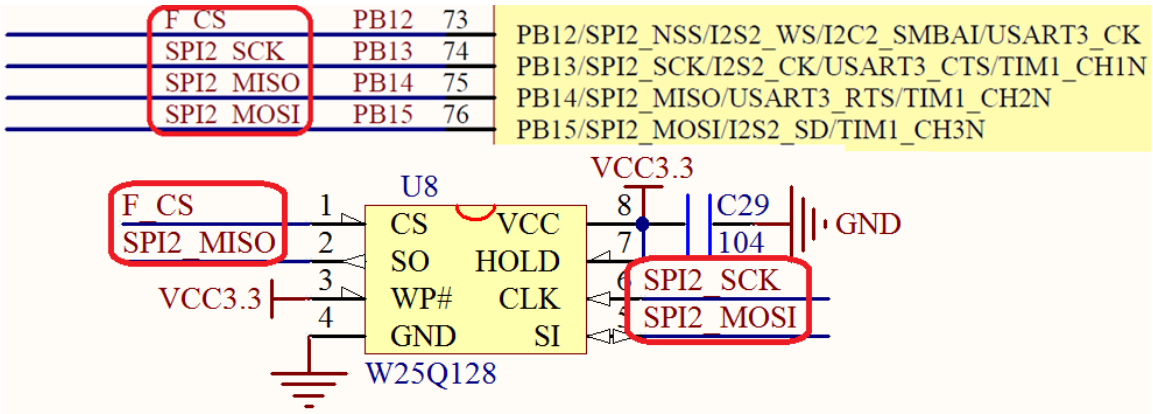


图 28.2.1 STM32F1 与 W25Q128 连接电路图

这里，我们的 F_CS 是连接在 PB12 上面的，另外要特别注意：W25Q128 和 NRF24L01 共用 SPI2，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。

28.3 软件设计

打开我们光盘的 SPI 实验工程，可以看到我们加入了 spi.c,flash.c 文件以及头文件 spi.h 和 flash.h，同时引入了库函数文件 stm32f1xx_hal_spi.c 文件以及头文件 stm32f1xx_hal_spi.h。

打开 spi.c 文件，看到如下代码：

```
SPI_HandleTypeDef SPI2_Handler;  //SPI1 句柄
//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 SPI2_Init(void)
{
    SPI2_Handler.Instance=SPI2;                         //SPI2
    SPI2_Handler.Init.Mode=SPI_MODE_MASTER;  //设置 SPI 工作模式，设置为主模式
    SPI2_Handler.Init.Direction=SPI_DIRECTION_2LINES;
                                        //设置 SPI 单向或者双向的数据模式:SPI 设置为双线模式
    SPI2_Handler.Init.DataSize=SPI_DATASIZE_8BIT;
                                        //设置 SPI 的数据大小:SPI 发送接收 8 位帧结构
    SPI2_Handler.Init.CLKPolarity=SPI_POLARITY_HIGH;
                                       //串行同步时钟的空闲状态为高电平
    SPI2_Handler.Init.CLKPhase=SPI_PHASE_2EDGE;
                                       //串行同步时钟的第二个跳变沿（上升或下降）数据被采样
    SPI2_Handler.Init.NSS=SPI_NSS_SOFT;       //NSS 信号由硬件（NSS 管脚）还是软件
                                       //（使用 SSI 位）管理:内部 NSS 信号有 SSI 位控制
    SPI2_Handler.Init.BaudRatePrescaler=SPI_BAUDRATEPRESCALER_256;
```

```
                                    //定义波特率预分频的值:波特率预分频值为 256
    SPI2_Handler.Init.FirstBit=SPI_FIRSTBIT_MSB;
                                //指定数据传输从 MSB 位还是 LSB 位开始:数据传输从 MSB 位开始
    SPI2_Handler.Init.TIMode=SPI_TIMODE_DISABLE;          //关闭 TI 模式
    SPI2_Handler.Init.CRCCalculation=SPI_CRCCALCULATION_DISABLE;
                                                          //关闭硬件 CRC 校验
    SPI2_Handler.Init.CRCPolynomial=7;                    //CRC 值计算的多项式
    HAL_SPI_Init(&SPI2_Handler);//初始化
    __HAL_SPI_ENABLE(&SPI2_Handler);                      //使能 SPI2
    SPI2_ReadWriteByte(0Xff);                             //启动传输
}

//SPI5 底层驱动，时钟使能，引脚配置
//此函数会被 HAL_SPI_Init()调用
//hspi:SPI 句柄
void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();           //使能 GPIOB 时钟
    __HAL_RCC_SPI2_CLK_ENABLE();            //使能 SPI2 时钟
    //PB13,14,15
    GPIO_Initure.Pin=GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;              //复用推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;        //快速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);
}

//SPI 速度设置函数
//SPI 速度=fAPB1/分频系数
//fAPB1 时钟一般为 42Mhz：
void SPI2_SetSpeed(u8 SPI_BaudRatePrescaler)
{
    assert_param(IS_SPI_BAUDRATE_PRESCALER(SPI_BaudRatePrescaler));//判断有效性
    __HAL_SPI_DISABLE(&SPI2_Handler);               //关闭 SPI
    SPI2_Handler.Instance->CR1&=0XFFC7;             //位 3-5 清零，用来设置波特率
    SPI2_Handler.Instance->CR1|=SPI_BaudRatePrescaler;//设置 SPI 速度
    __HAL_SPI_ENABLE(&SPI2_Handler);                //使能 SPI
}

//SPI1 读写一个字节
//TxData:要写入的字节
//返回值:读取到的字节
```

```
u8 SPI2_ReadWriteByte(u8 TxData)
{
    u8 Rxdata;
    HAL_SPI_TransmitReceive(&SPI2_Handler,&TxData,&Rxdata,1, 1000);
    return Rxdata;                          //返回收到的数据
}
```

此部分代码主要初始化 SPI，这里我们选择的是 SPI2，所以在 SPI2_Init 函数里面，其相关的操作都是针对 SPI2 的，其初始化步骤和我们上面介绍的一样。在初始化之后，我们就可以开始使用 SPI2 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 SPI2_Init 函数里面，把 SPI1 的频率设置成了最低（36Mhz，256 分频）。在外部函数里面，我们通过 SPI2_SetSpeed 来设置 SPI2 的速度，而我们的数据发送和接收则是通过 SPI2_ReadWriteByte 函数来实现的。

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
    SPI2_ReadWriteByte(W25X_ReadData);      //发送读取命令
    if(W25QXX_TYPE==W25Q256) //如果是 W25Q256 的话地址为 4 字节，发送最高 8 位
    {
        SPI2_ReadWriteByte((u8)((ReadAddr)>>24));
    }
    SPI2_ReadWriteByte((u8)((ReadAddr)>>16));   //发送 24bit 地址
    SPI2_ReadWriteByte((u8)((ReadAddr)>>8));
    SPI2_ReadWriteByte((u8)ReadAddr);
    for(i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPI2_ReadWriteByte(0XFF);    //循环读数
    }
    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 W25Q64 的字符串数组
const u8 TEXT_Buffer[]={"ELITE STM32 SPI TEST"};
#define SIZE sizeof(TEXT_Buffer)
int main(void)
{
    u8 key;
    u16 i=0;
    u8 datatemp[SIZE];
    u32 FLASH_SIZE;
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                 //初始化延时函数
    uart_init(115200);              //初始化串口
    usmart_dev.init(84);            //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    LCD_Init();                     //初始化 LCD FSMC 接口
    W25QXX_Init();                  //W25QXX 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"ELITE STM32");
    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,"2019/9/18");
    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 W25Q256
    {
```

```
            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=32*1024*1024;    //FLASH 大小为 32M 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,"W25Q256 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 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:    ");//提示传送完成
                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。

最后，我们将 W25QXX_ReadID 和 W25QXX_Erase_Chip 两个函数加入 usmart 控制，这样我们便可以通过 usmart 调用 W25QXX_ReadID 函数，来读取 SPI FLASH 的 ID，也可以调用 W25QXX_Erase_Chip 函数，实现对整个 SPI FLASH 的擦除（一般情况下不建议擦除整个 SPI

FLASH，因为这将导致字库和综合例程所需的系统文件全部丢失）。

28.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32F103 上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 28.4.1 所示：

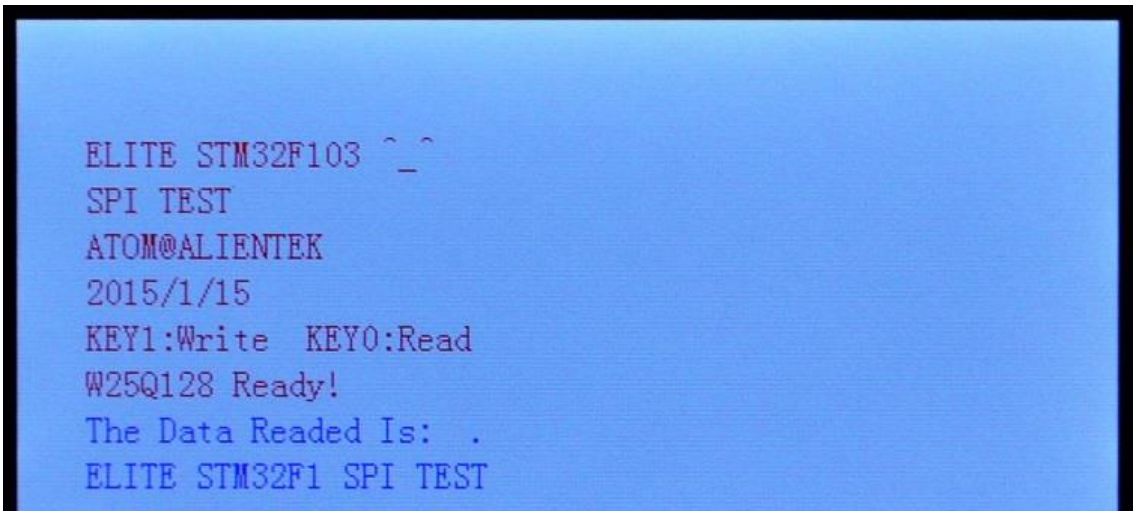


图 28.4.1 SPI 实验程序运行效果图

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 PB13 和 PB14 短接就可以看到报错了。

第二十九章 485 实验

本章我们将向大家介绍如何使用 STM32F1 的串口实现 485 通信（半双工）。在本章中，我们将使用 STM32F1 的串口 2 来实现两块开发板之间的 485 通信，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

29.1 485 简介
29.2 硬件设计
29.3 软件设计
29.4 下载验证

29.1 485 简介

485（一般称作 RS485/EIA-485）是隶属于 OSI 模型物理层的电气特性规定为 2 线，半双工，多点通信的标准。它的电气特性和 RS-232 大不一样。用缆线两端的电压差值来表示传递信号。RS485 仅仅规定了接受端和发送端的电气特性。它没有规定或推荐任何数据协议。

RS485 的特点包括：

1， 接口电平低，不易损坏芯片。RS485 的电气特性：逻辑“1”以两线间的电压差为+(2~6)V 表示；逻辑“0”以两线间的电压差为-(2~6)V 表示。接口信号电平比 RS232 降低了，不易损坏接口电路的芯片，且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。

2， 传输速率高。10 米时，RS485 的数据最高传输速率可达 35Mbps，在 1200m 时，传输速度可达 100Kbps。

3， 抗干扰能力强。RS485 接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。

4，传输距离远，支持节点多。RS485 总线最长可以传输 1200m 以上（速率≤100Kbps）一般最大支持 32 个节点，如果使用特制的 485 芯片，可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

RS485 推荐使用在点对点网络中，比如：线型，总线型网络等，而不能是星型，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 29.1.2 所示：

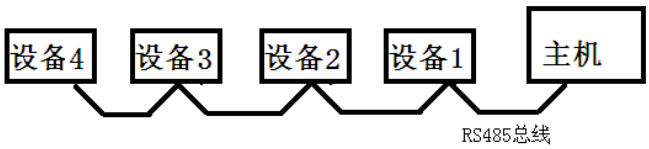


图 29.1.2 RS485 连接

在上面的连接中，如果需要添加匹配电阻，我们一般在总线的起止端加入，也就是主机和设备 4 上面各加一个 120Ω 的匹配电阻。

由于 RS485 具有传输距离远、传输速度快、支持节点多和抗干扰能力更强等特点，所以 RS485 有很广泛的应用。

精英 STM32F103 采用 SP3485 作为收发器，该芯片支持 3.3V 供电，最大传输速度可达 10Mbps，支持多达 32 个节点，并且有输出短路保护。该芯片的框图如图 29.1.2 所示：

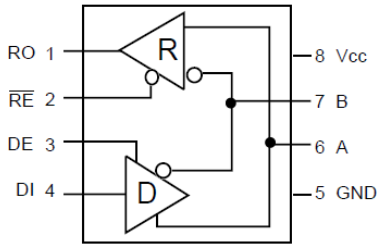


图 29.1.2 SP3485 框图

图中 A、B 总线接口，用于连接 485 总线。RO 是接收输出端，DI 是发送数据收入端，RE 是接收使能信号（低电平有效），DE 是发送使能信号（高电平有效）。

本章，我们通过该芯片连接 STM32F1 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：通过连接两个精英 STM32F103 的 RS485 接口，然后由 KEY0 控制发送，当按下一个开发板的 KEY0 的时候，就发送 5 个数据给另外一个开发板，并在两个开发板上分别显示发送的值和接收到的值。

本章，我们只需要配置好串口 2，就可以实现正常的 485 通信了，串口 2 的配置和串口 1 基本类似，只是串口的时钟来自 APB1，最大频率为 36Mhz。

29.2 硬件设计

本章要用到的硬件资源如下：

1） 指示灯 DS0
2） KEY0 按键
3） TFTLCD 模块
4） 串口 2
5） RS485 收发芯片 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 29.2.1 所示：

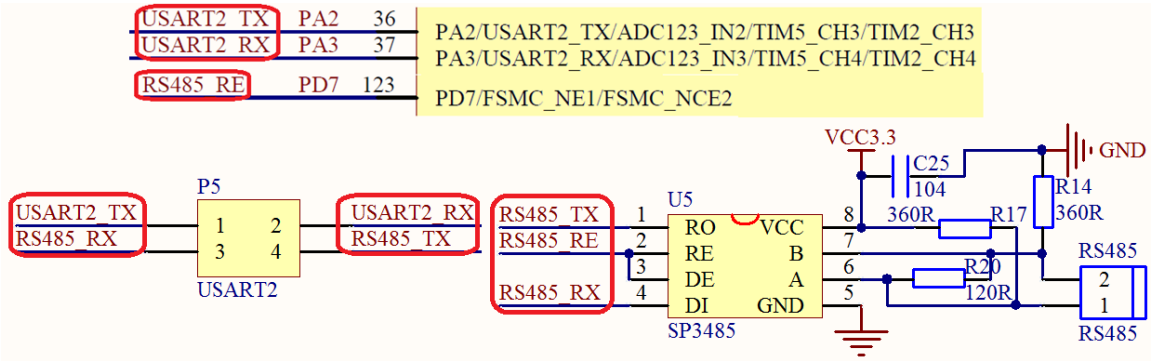


图 29.2.1 STM32F1 与 SP3485 连接电路图

从上图可以看出：STM32F1 的串口 2 通过 P5 端口设置，连接到 SP3485，通过 STM32F1 的 PD7 控制 SP3485 的收发，当 PD7=0 的时候，为接收模式；当 PD7=1 的时候，为发送模式。

另外，图中的 R14 和 R17 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 P5 排针的连接，通过跳线帽将 PA2 和 PA3 分别连接到 48R 和 48T 上面，如图 29.2.2 所示：

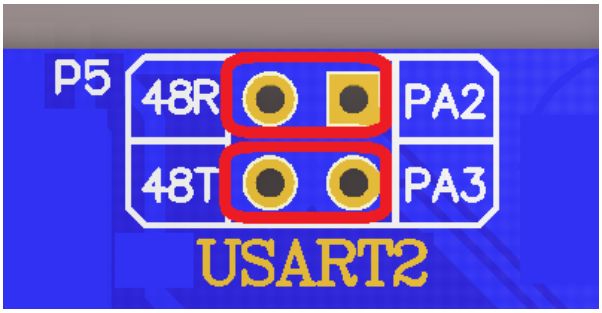


图 29.2.2 硬件连接示意图

最后，我们用 2 根导线将两个开发板 RS485 端子的 A 和 A，B 和 B 连接起来。这里注意不要接反了（A 接 B），接反了会导致通讯异常！！

29.3 软件设计

打开我们的 485 实验例程，可以发现项目中加入了一个 rs485.c 文件以及其头文件 rs485 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 stm32f1xx_hal_usart.c 文件和对应的头文件 stm32f1xx_hal_usart.h。

打开 rs485.c 文件，代码如下：

```
UART_HandleTypeDef USART2_RS485Handler;   //USART2 句柄(用于 RS485)
#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((__HAL_UART_GET_FLAG(&USART2_RS485Handler,UART_FLAG_RXNE)!=RESET))
                                                    //接收中断
    {
        HAL_UART_Receive(&USART2_RS485Handler,&res,1,1000);
        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 端口设置
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOA_CLK_ENABLE();             //使能 GPIOA 时钟
    __HAL_RCC_GPIOD_CLK_ENABLE();             //使能 GPIOD 时钟
    __HAL_RCC_USART2_CLK_ENABLE();            //使能 USART2 时钟
    GPIO_Initure.Pin=GPIO_PIN_2;              //PA2
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;        //复用推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;            //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;  //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);       //初始化 PA2
    GPIO_Initure.Pin=GPIO_PIN_3;              //PA3
    GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;//复用输入
```

```
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化 PA3

    //PD7 推挽输出，485 模式控制
    GPIO_Initure.Pin=GPIO_PIN_7;              //PD7
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
    HAL_GPIO_Init(GPIOD,&GPIO_Initure);

    //USART 初始化设置
    USART2_RS485Handler.Instance=USART2;                    //USART2
    USART2_RS485Handler.Init.BaudRate=bound;                //波特率
    USART2_RS485Handler.Init.WordLength=UART_WORDLENGTH_8B;
                                                            //字长为 8 位数据格式
    USART2_RS485Handler.Init.StopBits=UART_STOPBITS_1;      //一个停止位
    USART2_RS485Handler.Init.Parity=UART_PARITY_NONE;       //无奇偶校验位
    USART2_RS485Handler.Init.HwFlowCtl=UART_HWCONTROL_NONE;//无硬件流控
    USART2_RS485Handler.Init.Mode=UART_MODE_TX_RX;          //收发模式
    HAL_UART_Init(&USART2_RS485Handler);  //HAL_UART_Init()会使能 USART2
  __HAL_UART_DISABLE_IT(&USART2_RS485Handler,UART_IT_TC);
#if EN_USART2_RX
    __HAL_UART_ENABLE_IT(&USART2_RS485Handler,UART_IT_RXNE);
                                                            //开启接收中断
    HAL_NVIC_EnableIRQ(USART2_IRQn);                        //使能 USART1 中断
    HAL_NVIC_SetPriority(USART2_IRQn,3,3);                  //抢占优先级 3，子优先级 3
#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)
{
    RS485_TX_EN=1;              //设置为发送模式
    HAL_UART_Transmit(&USART2_RS485Handler,buf,len,1000);//串口 2 发送数据
    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 PD7 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;
    u8 i=0,t=0;
    u8 cnt=0;
    u8 rs485buf[5];
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    usmart_dev.init(84);                    //初始化 USMART
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    LCD_Init();                             //初始化 LCD FSMC 接口
    RS485_Init(9600);                       //初始化 RS485
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2017/5/30");
        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，我们主要关注下 RS485_Init(36,9600)，这里用的是 36，而不是 72，是因为 APB1 的时钟是 36Mhz，故是 36，而串口 1 的时钟来自 APB2，是 72Mhz 的时钟，所以这里和串口 1 的设置是有点区别的。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。

最后，我们将 RS485_Send_Data 函数加入 USMART 控制，这样，我们就可以通过串口调试助手，随意发送你想要发的数据（字符串形式发送）了，方便大家测试。

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 精英 STM32F103 上（注意要 2 个开发板都下载这个代码哦），得到如图 29.4.1 所示：

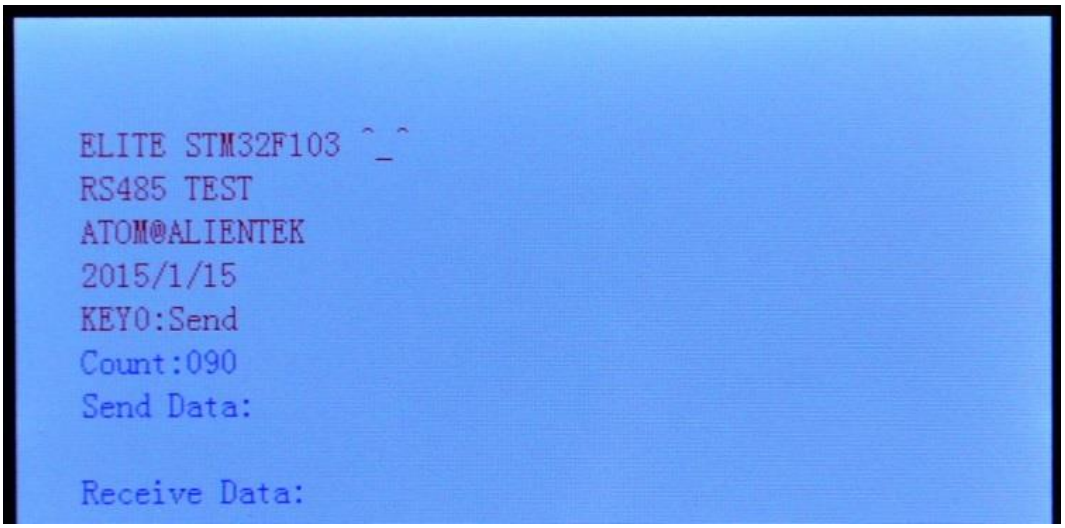


图 29.4.1 程序运行效果图

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 29.4.2 和图 41.4.3 所示：

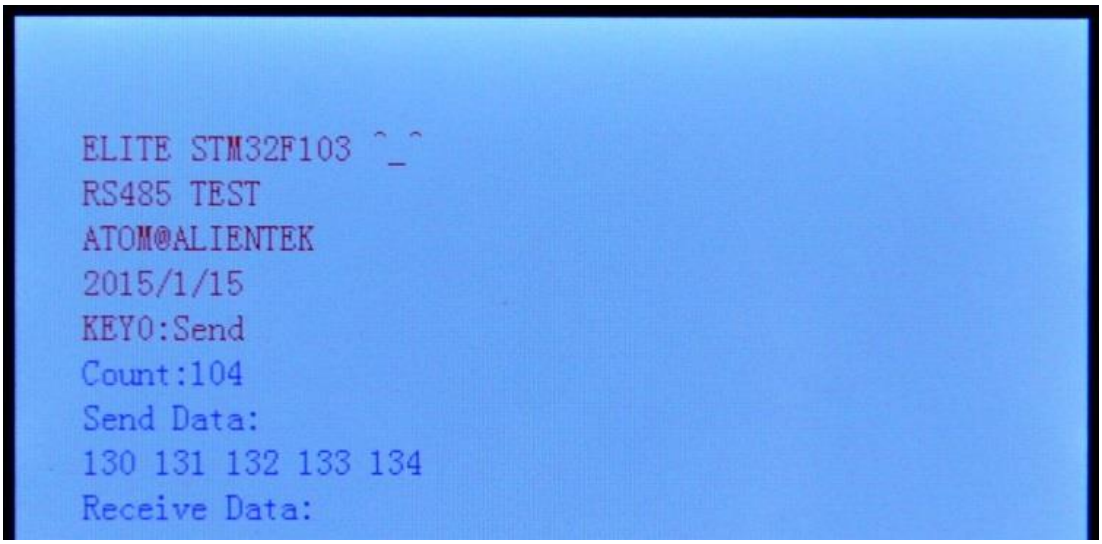


图 29.4.2 RS485 发送数据

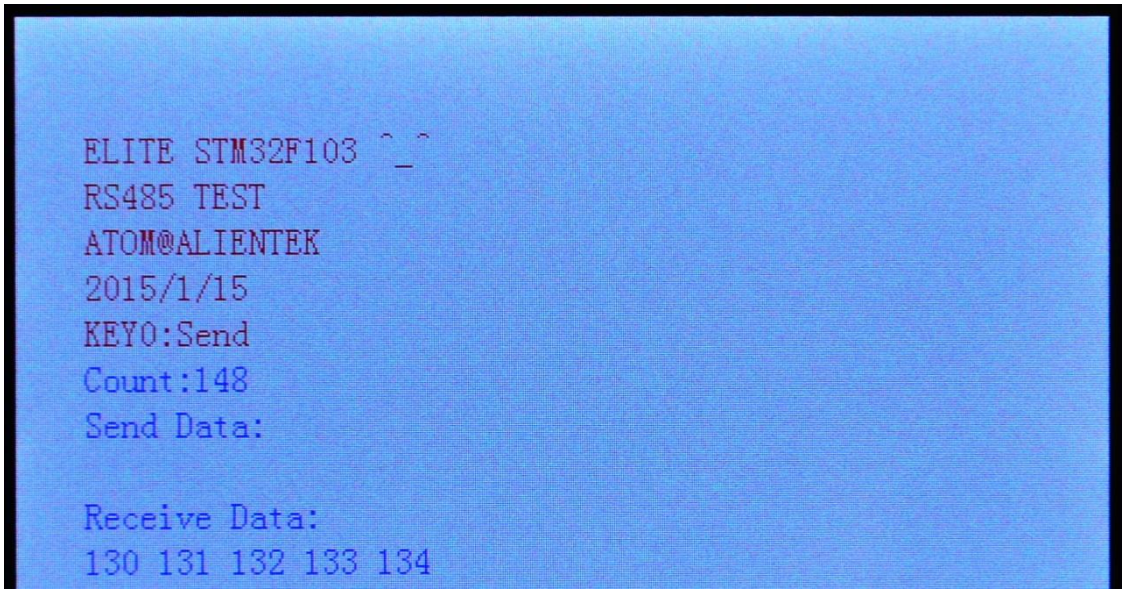


图 29.4.3 RS485 接收数据

图 29.4.2 来自开发板 A，发送了 5 个数据，图 29.4.3 来自开发板 B，接收到了来自开发板 A 的 5 个数据。

另外，利用 USMART 测试的部分，我们这里就不做介绍了，大家可自行验证下。

第三十章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32F1 自带的 CAN 控制器来实现两个开发板之间的 CAN 通讯，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

30.1 CAN 简介
30.2 硬件设计
30.3 软件设计
30.4 下载验证

30.1 CAN 简介

CAN 是 Controller Area Network 的缩写（以下称为 CAN），是 ISO 国际标准化的串行通信协议。在当前的汽车产业中，出于对安全性、舒适性、方便性、低公害、低成本的要求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN，进行大量数据的高速通信”的需要，1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化，现在在欧洲已是汽车网络的标准协议。

现在，CAN 的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。现场总线是当今自动化领域技术发展的热点之一，被誉为自动化领域的计算机局域网。它的出现为分布式控制系统实现各节点之间实时、可靠的数据通信提供了强有力的技术支持。

CAN 控制器根据两根线上的电位差来判断总线电平。总线电平分为显性电平和隐性电平，二者必居其一。发送方通过使总线电平发生变化，将消息发送给接收方。

CAN 协议具有一下特点：

1） **多主控制。**在总线空闲时，所有单元都可以发送消息（多主控制），而两个以上的单元同时开始发送消息时，根据标识符（Identifier 以下称为 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） **系统的柔软性。**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 30.1.1 所示：

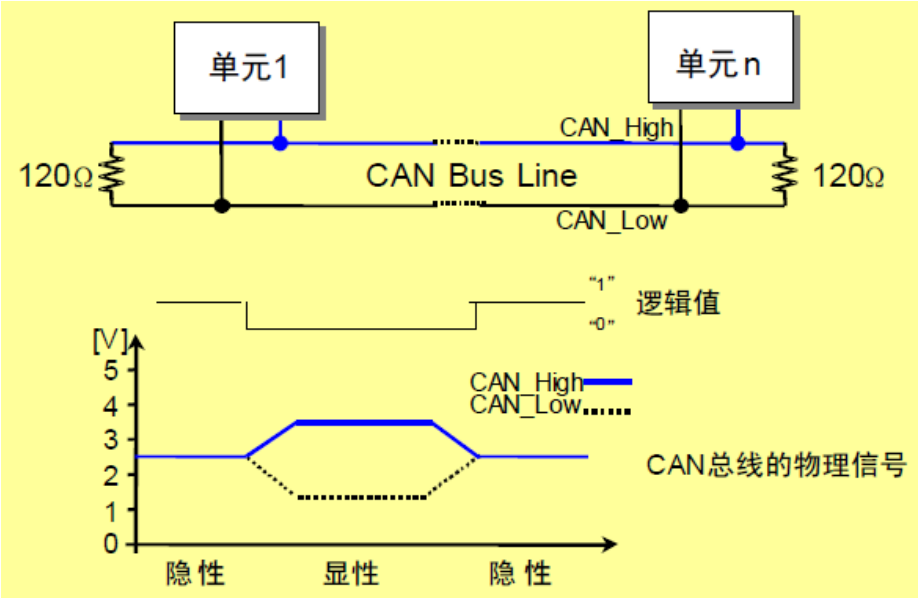


图 30.1.1 ISO11898 物理层特性

从该特性可以看出，显性电平对应逻辑 0，CAN_H 和 CAN_L 之差为 2.5V 左右。而隐性电平对应逻辑 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性电平，总线上才为隐性电平（显性电平比隐性电平更强）。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 30.1.1 所示：

帧类型	帧用途
数据帧	用于发送单元向接收单元传送数据的帧
遥控帧	用于接收单元向具有相同 ID 的发送单元请求数据的帧
错误帧	用于当检测出错误时向其它单元通知错误的帧
过载帧	用于接收单元通知其尚未做好接收准备的帧
间隔帧	用于将数据帧及遥控帧与前面的帧分离开来的帧

表 30.1.1 CAN 协议各种帧及其用途

由于篇幅所限，我们这里仅对数据帧进行详细介绍，数据帧一般由 7 个段构成，即：

（1） 帧起始。表示数据帧开始的段。
（2） 仲裁段。表示该帧优先级的段。
（3） 控制段。表示数据的字节数及保留位的段。
（4） 数据段。数据的内容，一帧可发送 0~8 个字节的数据。
（5） CRC 段。检查帧的传输错误的段。
（6） ACK 段。表示确认正常接收的段。

（7） 帧结束。表示数据帧结束的段。

数据帧的构成如图 30.1.2 所示：

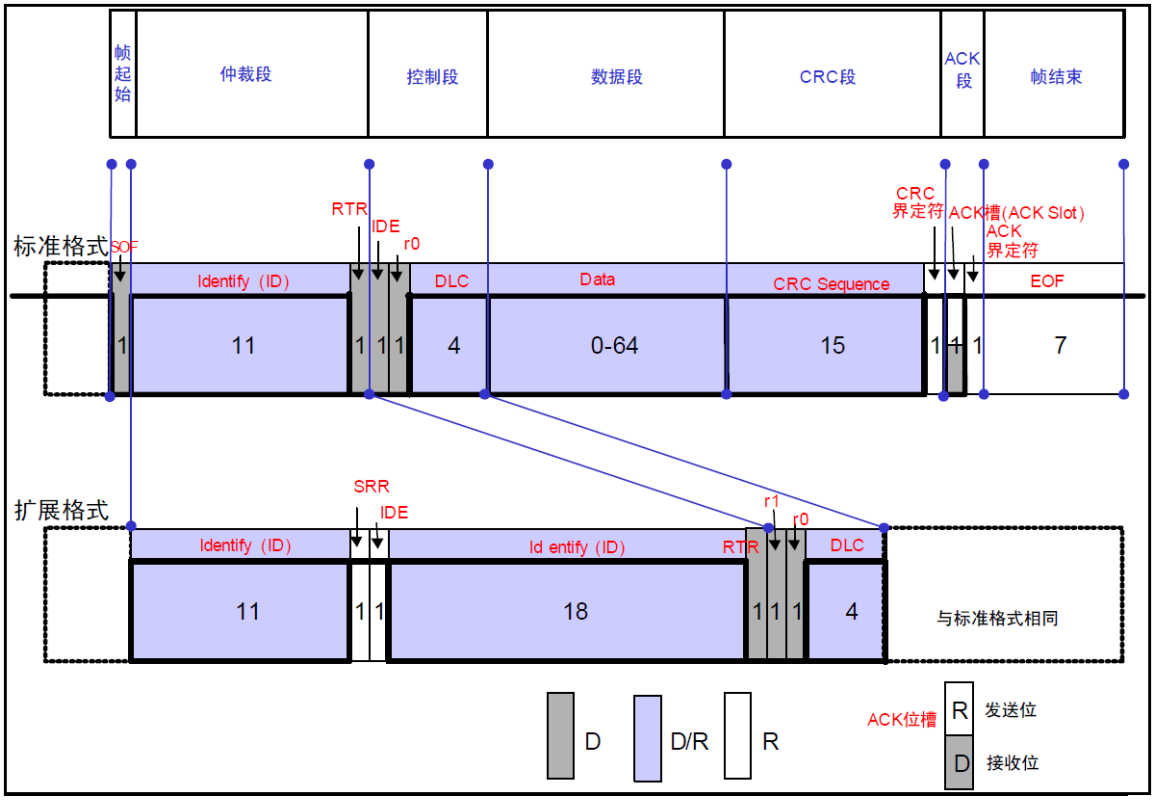


图 30.1.2 数据帧的构成

图中 D 表示显性电平，R 表示隐形电平（下同）。

帧起始，这个比较简单，标准帧和扩展帧都是由 1 个位的显性电平表示帧起始。

仲裁段，表示数据优先级的段，标准帧和扩展帧格式在本段有所区别，如图 30.1.3 所示：

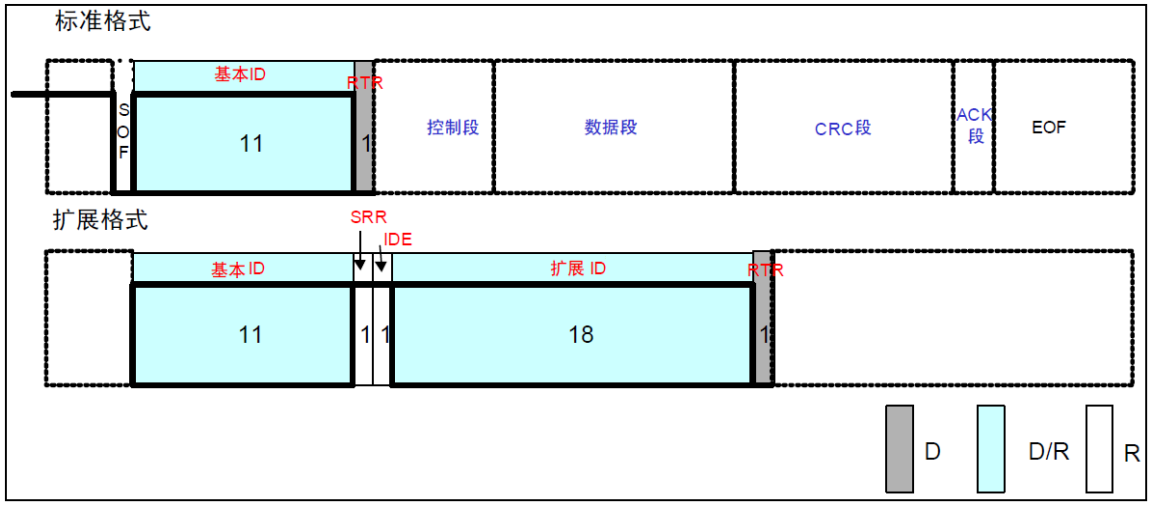


图 30.1.3 数据帧仲裁段构成

标准格式的 ID 有 11 个位。从 ID28 到 ID18 被依次发送。禁止高 7 位都为隐性（禁止设定：ID=1111111XXXX）。扩展格式的 ID 有 29 个位。基本 ID 从 ID28 到 ID18，扩展 ID 由 ID17 到 ID0 表示。基本 ID 和标准格式的 ID 相同。禁止高 7 位都为隐性（禁止设定：基本

ID=1111111XXXX）。

其中 RTR 位用于标识是否是远程帧（0，数据帧；1，远程帧），IDE 位为标识符选择位（0，使用标准标识符；1，使用扩展标识符），SRR 位为代替远程请求位，为隐性位，它代替了标准帧中的 RTR 位。

控制段，由 6 个位构成，表示数据段的字节数。标准帧和扩展帧的控制段稍有不同，如图 30.1.4 所示：

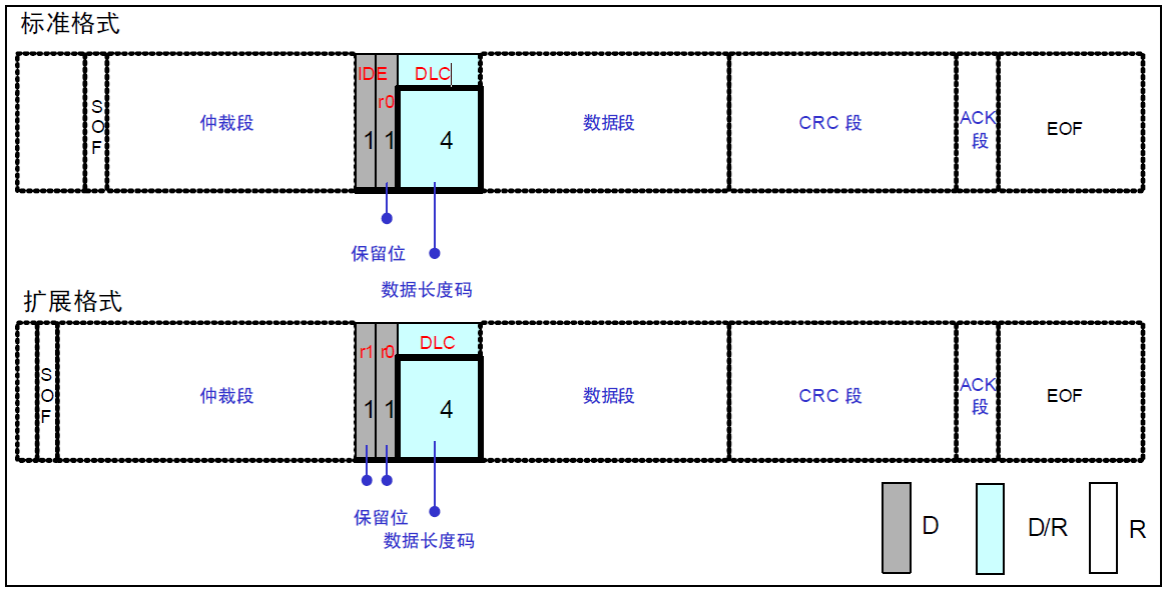


图 30.1.4 数据帧控制段构成

上图中，r0 和 r1 为保留位，必须全部以显性电平发送，但是接收端可以接收显性、隐性及任意组合的电平。DLC 段为数据长度表示段，高位在前，DLC 段有效值为 0~8，但是接收方接收到 9~15 的时候并不认为是错误。

数据段，该段可包含 0~8 个字节的数据。从最高位（MSB）开始输出，标准帧和扩展帧在这个段的定义都是一样的。如图 30.1.5 所示：

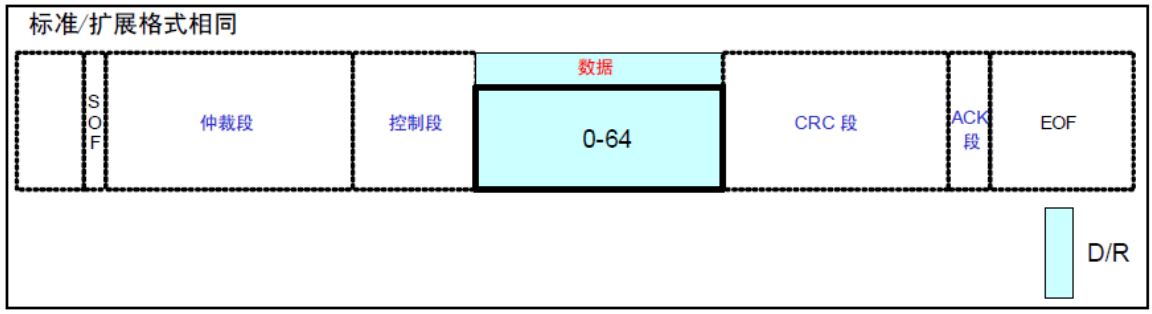


图 30.1.5 数据帧数据段构成

CRC 段，该段用于检查帧传输错误。由 15 个位的 CRC 顺序和 1 个位的 CRC 界定符（用于分隔的位）组成，标准帧和扩展帧在这个段的格式也是相同的。如图 30.1.6 所示：

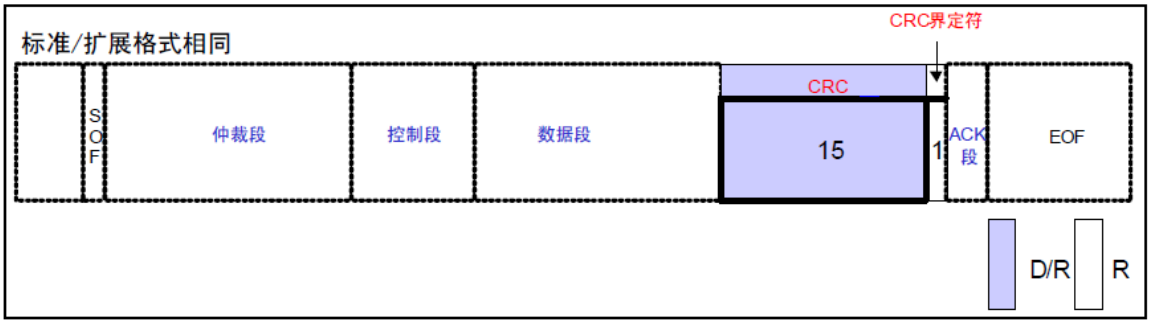


图 30.1.6 数据帧 CRC 段构成

此段 CRC 的值计算范围包括：帧起始、仲裁段、控制段、数据段。接收方以同样的算法计算 CRC 值并进行比较，不一致时会通报错误。

ACK 段，此段用来确认是否正常接收。由 ACK 槽(ACK Slot)和 ACK 界定符 2 个位组成。标准帧和扩展帧在这个段的格式也是相同的。如图 30.1.7 所示：

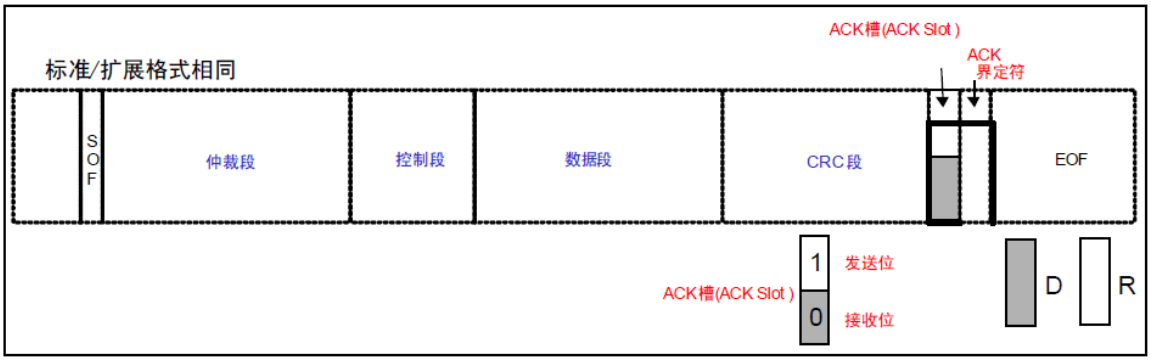


图 30.1.7 数据帧 CRC 段构成

发送单元的 ACK，发送 2 个位的隐性位，而接收到正确消息的单元在 ACK 槽（ACK Slot）发送显性位，通知发送单元正常接收结束，这个过程叫发送 ACK/返回 ACK。发送 ACK 的是在既不处于总线关闭态也不处于休眠态的所有接收单元中，接收到正常消息的单元（发送单元不发送 ACK）。所谓正常消息是指不含填充错误、格式错误、CRC 错误的消息。

帧结束，这个段也比较简单，标准帧和扩展帧在这个段格式一样，由 7 个位的隐性位组成。

至此，数据帧的 7 个段就介绍完了，其他帧的介绍，请大家参考光盘的 CAN 入门书.pdf 相关章节。接下来，我们再来看看 CAN 的位时序。

由发送单元在非同步的情况下发送的每秒钟的位数称为位速率。一个位可分为 4 段。

- 同步段（SS）
- 传播时间段（PTS）
- 相位缓冲段 1（PBS1）
- 相位缓冲段 2（PBS2）

这些段又由可称为 Time Quantum（以下称为 Tq）的最小时间单位构成。

1 位分为 4 个段，每个段又由若干个 Tq 构成，这称为位时序。

1 位由多少个 Tq 构成、每个段又由多少个 Tq 构成等，可以任意设定位时序。通过设定位时序，多个单元可同时采样，也可任意设定采样点。各段的作用和 Tq 数如表 30.1.2 所示：

段名称	段的作用	Tq 数	
同步段 （SS: Synchronization Segment）	多个连接在总线上的单元通过此段实现时序调整，同步进行接收和发送的工作。由隐性电平到显性电平的边沿或由显性电平到隐性电平边沿最好出现在此段中。	1Tq	8～25Tq
传播时间段 （PTS: Propagation Time Segment）	用于吸收网络上的物理延迟的段。 所谓的网络的物理延迟指发送单元的输出延迟、总线上信号的传播延迟、接收单元的输入延迟。 这个段的时间为以上各延迟时间的和的两倍。	1～8Tq	
相位缓冲段 1 (PBS1: Phase Buffer Segment 1)	当信号边沿不能被包含于 SS 段中时，可在此段进行补偿。 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。	1～8Tq	
相位缓冲段 2 (PBS2: Phase Buffer Segment 2)		2～8Tq	
再同步补偿宽度 (SJW: reSynchronization Jump Width)	因时钟频率偏差、传送延迟等，各单元有同步误差。SJW 为补偿此误差的最大值。	1～4Tq	

表 30.1.2 一个位各段及其作用

1 个位的构成如图 30.1.8 所示：

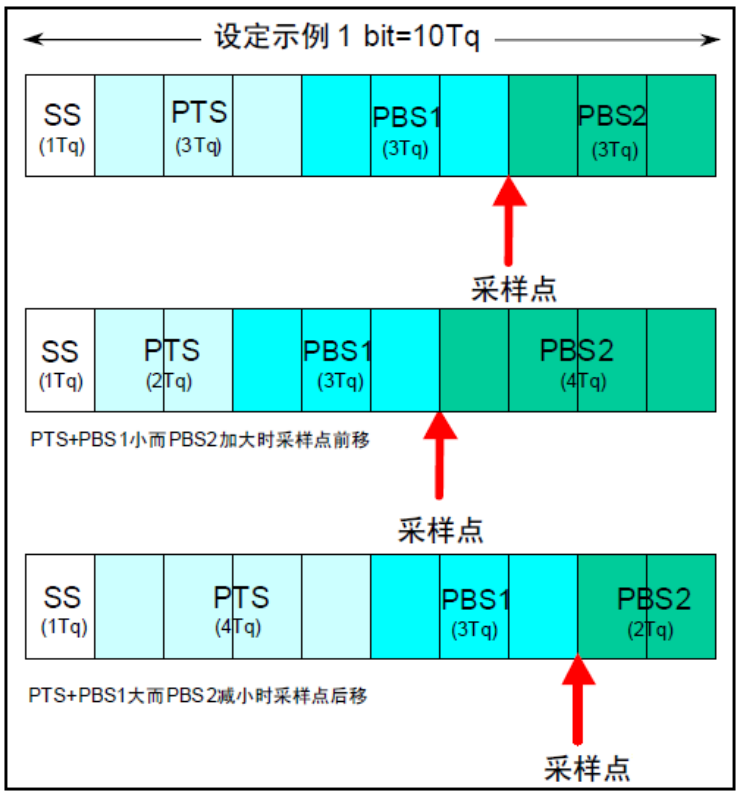


图 30.1.8 一个位的构成

上图的采样点，是指读取总线电平，并将读到的电平作为位值的点。位置在 PBS1 结束处。根据这个位时序，我们就可以计算 CAN 通信的波特率了。具体计算方法，我们等下再介绍，前面提到的 CAN 协议具有仲裁功能，下面我们来看看是如何实现的。

在总线空闲态，最先开始发送消息的单元获得发送权。

当多个单元同时开始发送时，各发送单元从仲裁段的第一位开始进行仲裁。连续输出显性电平最多的单元可继续发送。实现过程，如图 30.1.9 所示：

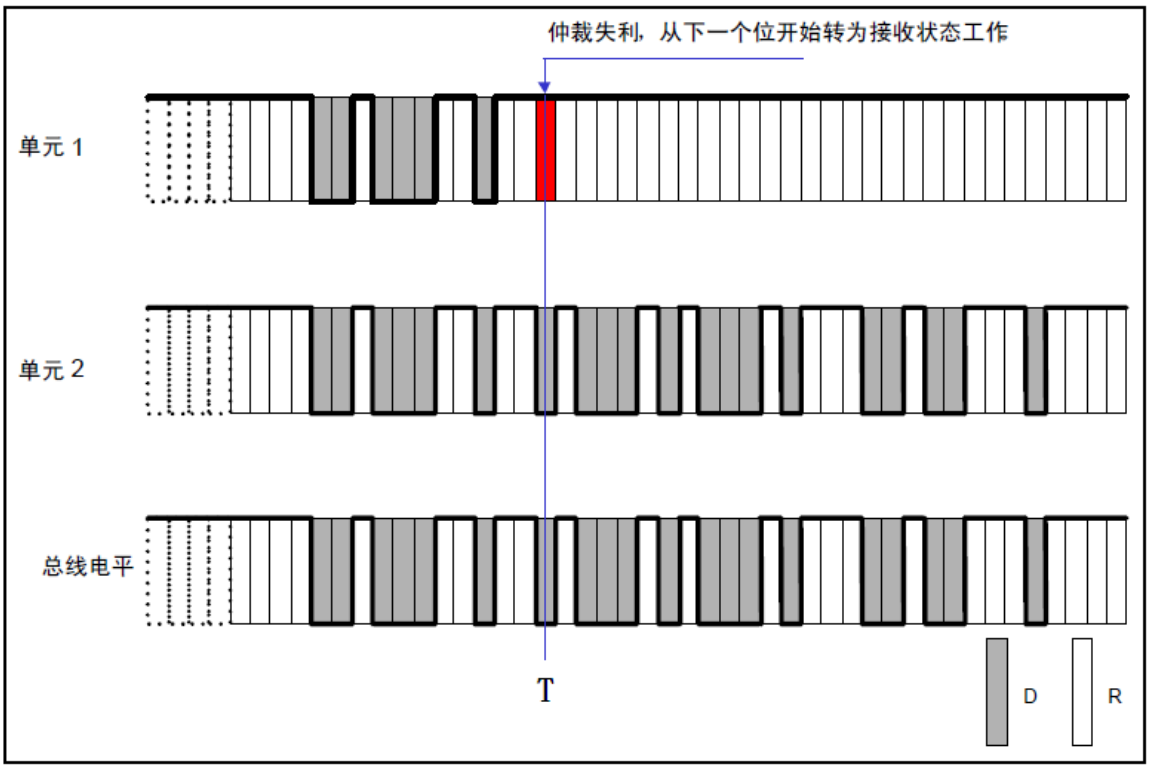


图 30.1.9 CAN 总线仲裁过程

上图中，单元 1 和单元 2 同时开始向总线发送数据，开始部分他们的数据格式是一样的，故无法区分优先级，直到 T 时刻，单元 1 输出隐性电平，而单元 2 输出显性电平，此时单元 1 仲裁失利，立刻转入接收状态工作，不再与单元 2 竞争，而单元 2 则顺利获得总线使用权，继续发送自己的数据。这就实现了仲裁，让连续发送显性电平多的单元获得总线使用权。

通过以上介绍，我们对 CAN 总线有了个大概了解（详细介绍参考光盘的：《CAN 入门书.pdf》），接下来我们介绍下 STM32F1 的 CAN 控制器。

STM32F1 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。

STM32F1 的 bxCAN 的主要特点有：

- 支持 CAN 协议 2.0A 和 2.0B 主动模式
- 波特率最高达 1Mbps
- 支持时间触发通信
- 具有 3 个发送邮箱
- 具有 3 级深度的 2 个接收 FIFO
- 可变的过滤器组（最多 28 个）

在 STM32 互联型产品中，带有 2 个 CAN 控制器，而我们使用的 STM32F103ZET6 属于增强型，不是互联型，只有 1 个 CAN 控制器。双 CAN 的框图如图 30.1.10 所示：

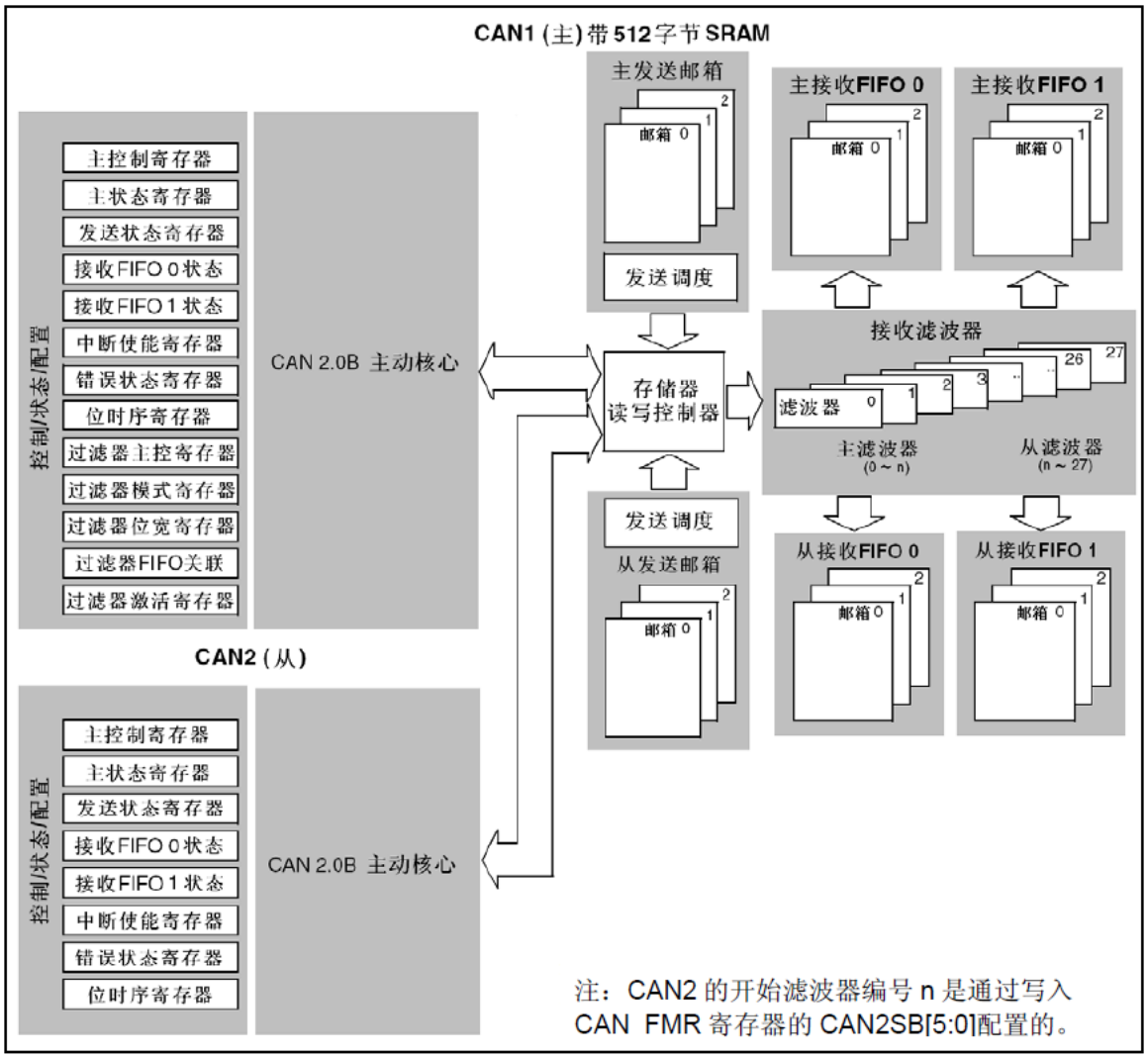


图 30.1.10 双 CAN 框图

从图中可以看出两个 CAN 都分别拥有自己的发送邮箱和接收 FIFO，但是他们共用 28 个滤波器。通过 CAN_FMR 寄存器的设置，可以设置滤波器的分配方式。

STM32 的标识符过滤是一个比较复杂的东东，它的存在减少了 CPU 处理 CAN 通信的开销。STM32 的过滤器组最多有 28 个（互联型），但是 STM32F103ZET6 只有 14 个（增强型），每个滤波器组 x 由 2 个 32 为寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32F1 每个过滤器组的位宽都可以独立配置，以满足应用程序的不同需求。根据位宽的不同，每个过滤器组可提供：

- 1 个 32 位过滤器，包括：STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位
- 2 个 16 位过滤器，包括：STDID[10:0]、IDE、RTR 和 EXTID[17:15]位

此外过滤器可配置为，屏蔽位模式和标识符列表模式。

在屏蔽位模式下，标识符寄存器和屏蔽寄存器一起，指定报文标识符的任何一位，应该按照“必须匹配”或“不用关心”处理。

而在标识符列表模式下，屏蔽寄存器也被当作标识符寄存器用。因此，不是采用一个标识符加一个屏蔽位的方式，而是使用 2 个标识符寄存器。接收报文标识符的每一位都必须跟过滤

器标识符相同。

通过 CAN_FMR 寄存器，可以配置过滤器组的位宽和工作模式，如图 30.1.11 所示：

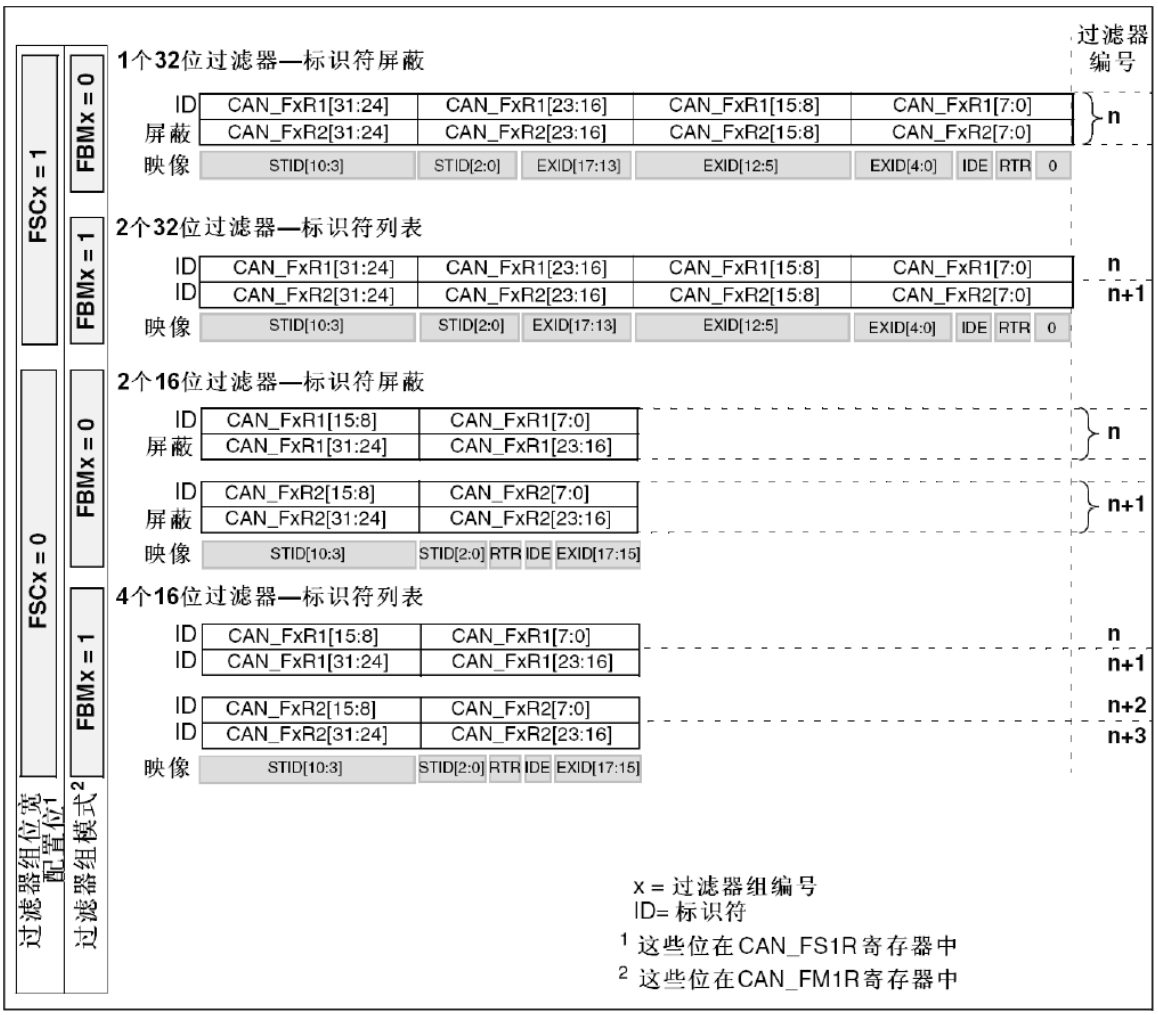


图 30.1.11 过滤器组位宽模式设置

为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。

为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。

应用程序不用的过滤器组，应该保持在禁用状态。

过滤器组中的每个过滤器，都被编号为(叫做过滤器号，图 30.1.11 中的 n)从 0 开始，到某个最大数值一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 为位过滤器-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，即我们希望收到的映像（STID+EXTID+IDE+RTR）最好是：0XFFFF0000。而 0XFF00FF00 就是设置我们需要必须关心的 ID，表示收到的映像，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_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，即收到的映像必须是 0XFFxx00xx，才算是正确的（x 表示不关心）。

关于标识符过滤的详细介绍，请参考《STM32 中文参考手册》的 22.7.4 节（431 页）。接下来，我们看看 STM32 的 CAN 发送和接收的流程。

CAN 发送流程

CAN 发送流程为：程序选择 1 个空置的邮箱（TME=1）→设置标识符（ID），数据长度和发送数据→设置 CAN_TIxR 的 TXRQ 位为 1，请求发送→邮箱挂号（等待成为最高优先级）→预定发送（等待总线空闲）→发送→邮箱空置。整个流程如图 30.1.12 所示：

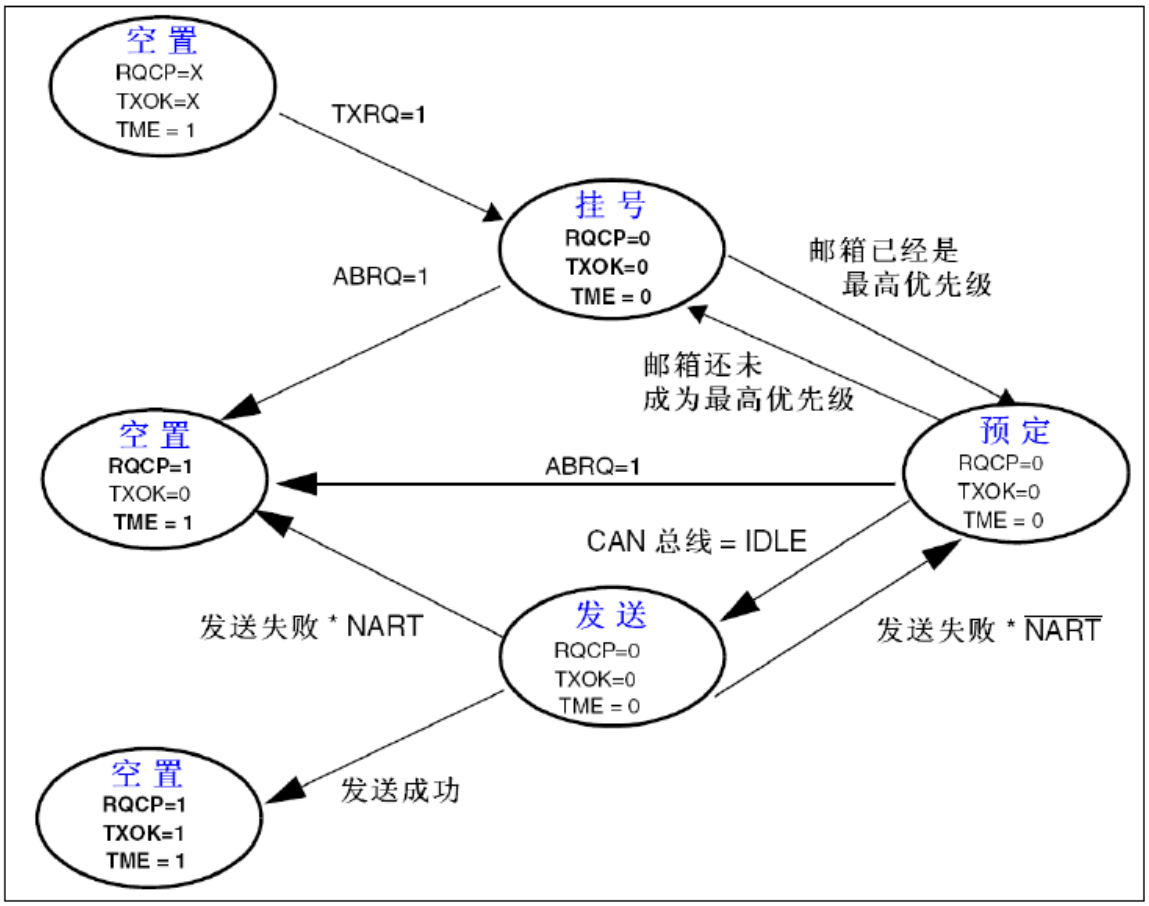


图 30.1.12 发送邮箱

上图中，还包含了很多其他处理，不强制退出发送（ABRQ=1）和发送失败处理等。通过这个流程图，我们大致了解了 CAN 的发送流程，后面的数据发送，我们基本就是按照此流程来走。接下来再看看 CAN 的接收流程。

CAN 接收流程

CAN 接收到的有效报文，被存储在 3 级邮箱深度的 FIFO 中。FIFO 完全由硬件来管理，从而节省了 CPU 的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取 FIFO 输出邮箱，来读取 FIFO 中最先收到的报文。这里的有效报文是指那些正确被接收的（直到 EOF 都没有错误）且通过了标识符过滤的报文。前面我们知道 CAN 的接收有 2 个 FIFO，我们每个滤波器组都可以设置其关联的 FIFO，通过 CAN_FFA1R 的设置，可以将滤波器组关联到 FIFO0/FIFO1。

CAN 接收流程为：FIFO 空→收到有效报文→挂号_1（存入 FIFO 的一个邮箱，这个由硬件控制，我们不需要理会）→收到有效报文→挂号_2→收到有效报文→挂号_3→收到有效报文→溢出。

这个流程里面，我们没有考虑从 FIFO 读出报文的情况，实际情况是：我们必须在 FIFO 溢出之前，读出至少 1 个报文，否则下个报文到来，将导致 FIFO 溢出，从而出现报文丢失。每读出 1 个报文，相应的挂号就减 1，直到 FIFO 空。CAN 接收流程如图 30.1.13 所示：

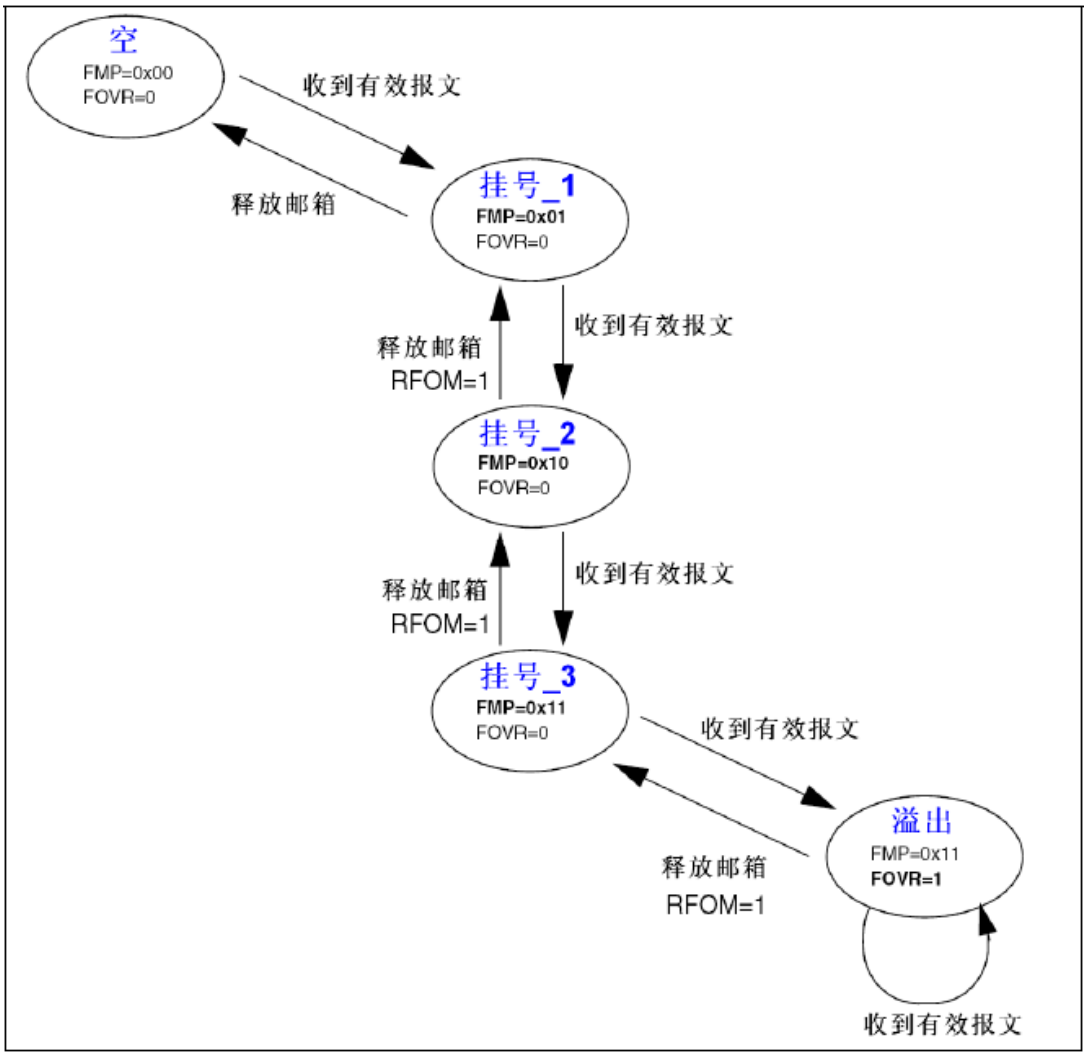


图 30.1.13 FIFO 接收报文

FIFO 接收到的报文数，我们可以通过查询 CAN_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 STM32 的 CAN 位时间特性，STM32 的 CAN 位时间特性和之前我们介绍的，稍有点区别。STM32 把传播时间段和相位缓冲段 1（STM32 称之为时间段 1）合并了，所以 STM32 的 CAN 一个位只有 3 段：同步段（SYNC_SEG）、时间段 1（BS1）和时间段 2（BS2）。STM32 的 BS1 段可以设置为 1~16 个时间单元，刚好等于我们上面介绍的传播时间段和相位缓冲段 1 之和。STM32 的 CAN 位时序如图 30.1.14 所示：

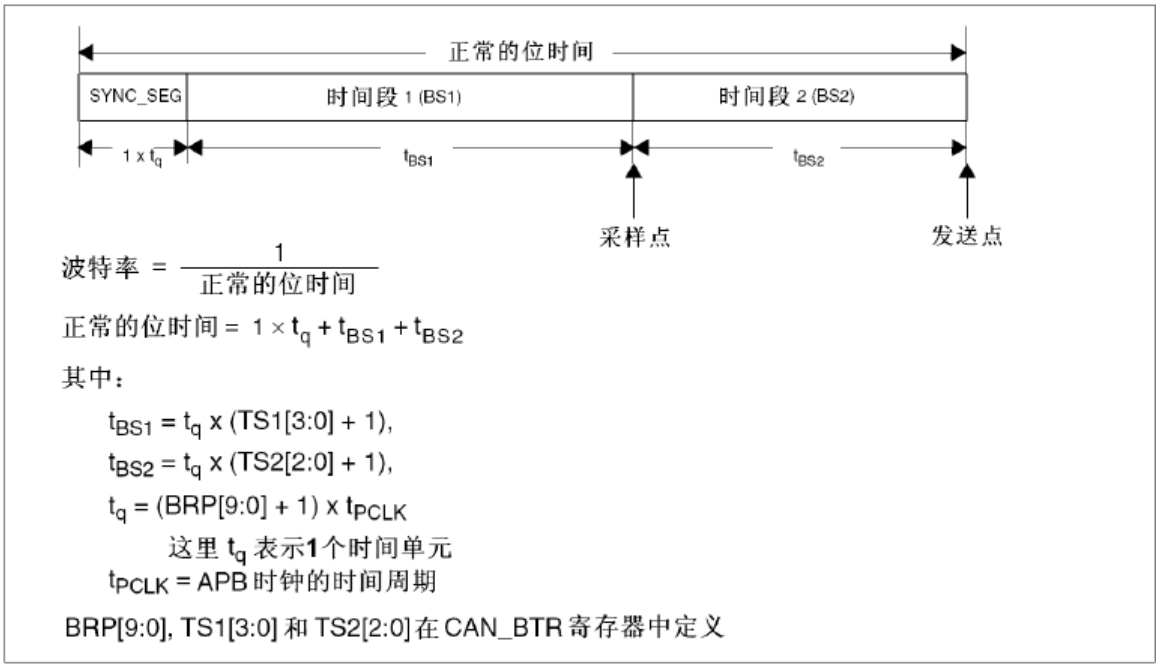


图 30.1.14 STM32 CAN 位时序

图中还给出了 CAN 波特率的计算公式，我们只需要知道 BS1 和 BS2 的设置，以及 APB1 的时钟频率（一般为 36Mhz），就可以方便的计算出波特率。比如设置 TS1=8、TS2=7 和 BRP=3，在 APB1 频率为 36Mhz 的条件下，即可得到 CAN 通信的波特率=36000/[(9+8+1)*4]=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 30.1.15：



图 30.1.15 寄存器 CAN_MCR 各位描述

该寄存器的详细描述，请参考《STM32 中文参考手册》22.9.2 节（439 页），这里我们仅介绍下 INRQ 位，该位用来控制初始化请求。

软件对该位清 0，可使 CAN 从初始化模式进入正常工作模式：当 CAN 在接收引脚检测到连续的 11 个隐性位后，CAN 就达到同步，并为接收和发送数据作好准备了。为此，硬件相应地对 CAN_MSR 寄存器的 INAK 位清’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 30.1.16 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	保留				SJW[1:0]		保留	TS2[2:0]			TS1[3:0]			
rw	rw	res				rw	rw	res	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						BRP[9:0]									
res						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31	**SILM:** 静默模式(用于调试) (Silent mode (debug)) 0: 正常状态； 1: 静默模式。
位30	**LBKM:** 环回模式(用于调试) (Loop back mode (debug)) 0: 禁止环回模式； 1: 允许环回模式。
位29:26	保留位，硬件强制为0。
位25:24	**SJW[1:0]:** 重新同步跳跃宽度 (Resynchronization jump width) 为了重新同步，该位域定义了CAN硬件在每位中可以延长或缩短多少个时间单元的上限。 $t_{RJW} = t_{CAN} \times (SJW[1:0] + 1)$。
位23	保留位，硬件强制为0。
位22:20	**TS2[2:0]:** 时间段2 (Time segment 2) 该位域定义了时间段2占用了多少个时间单元 $t_{BS2} = t_{CAN} \times (TS2[2:0] + 1)$。
位19:16	**TS1[3:0]:** 时间段1 (Time segment 1) 该位域定义了时间段1占用了多少个时间单元 $t_{BS1} = t_{CAN} \times (TS1[3:0] + 1)$ 关于位时间特性的详细信息，请参考22.7.7节位时间特性。
位15:10	保留位，硬件强制其值为0。
位9:0	**BRP[9:0]:** 波特率分频器 (Baud rate prescaler) 该位域定义了时间单元(t_q)的时间长度 $t_q = (BRP[9:0]+1) \times t_{PCLK}$

图 30.1.16 寄存器 CAN_BTR 各位描述

STM32 提供了两种测试模式，环回模式和静默模式，当然他们组合还可以组合成环回静默模式。这里我们简单介绍下环回模式。

在环回模式下，bxCAN 把发送的报文当作接收的报文并保存(如果可以通过接收过滤)在接收邮箱里。也就是环回模式是一个自发自收的模式，如图 30.1.17 所示：

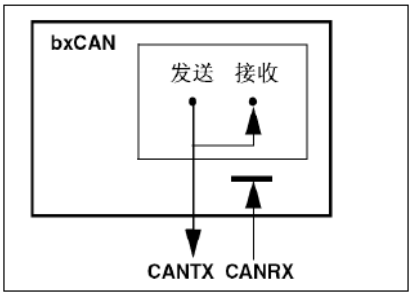


图 30.1.17 CAN 环回模式

环回模式可用于自测试。为了避免外部的影响，在环回模式下 CAN 内核忽略确认错误(在数据/远程帧的确认位时刻，不检测是否有显性位)。在环回模式下，bxCAN 在内部把 Tx 输出回馈到 Rx 输入上，而完全忽略 CANRX 引脚的实际状态。发送的报文可以在 CANTX 引脚上检测到。

第三个，我们介绍 CAN 发送邮箱标识符寄存器（CAN_TIxR）（x=0~3），该寄存器各位描

述如图 30.1.18 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STID[10:0]/EXID[28:18]											EXID[17:13]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXID[12:0]													IDE	RTR	TXRQ
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:21	**STID[10:0]/EXID[28:18]:** 标准标识符或扩展标识符 (Standard identifier or extended identifier) 依据IDE位的内容，这些位或是标准标识符，或是扩展身份标识的高字节。
位20:3	**EXID[17:0]:** 扩展标识符 (Extended identifier) 扩展身份标识的低字节。
位2	**IDE:** 标识符选择 (Identifier extension) 该位决定发送邮箱中报文使用的标识符类型 0：使用标准标识符； 1：使用扩展标识符。
位1	**RTR:** 远程发送请求 (Remote transmission request) 0：数据帧； 1：远程帧。
位0	**TXRQ:** 发送数据请求 (Transmit mailbox request) 由软件对其置'1'，来请求发送邮箱的数据。当数据发送完成，邮箱为空时，硬件对其清'0'。

图 30.1.18 寄存器 CAN_TIxR 各位描述

该寄存器主要用来设置标识符（包括扩展标识符），另外还可以设置帧类型，通过 TXRQ 值 1，来请求邮箱发送。因为有 3 个发送邮箱，所以寄存器 CAN_TIxR 有 3 个。

第四个，我们介绍 CAN 发送邮箱数据长度和时间戳寄存器 (CAN_TDTxR) (x=0~2)，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，这里就不详细介绍了。

第五个，我介绍的是 CAN 发送邮箱低字节数据寄存器 (CAN_TDLxR) (x=0~2)，该寄存器各位描述如图 30.1.19 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:24	**DATA3[7:0]：**数据字节3 (Data byte 3) 报文的数据字节3。
位23:16	**DATA2[7:0]：**数据字节2 (Data byte 2) 报文的数据字节2。
位15:8	**DATA1[7:0]：**数据字节1 (Data byte 1) 报文的数据字节1。
位7:0	**DATA0[7:0]：**数据字节0 (Data byte 0) 报文的数据字节0。 报文包含0到8个字节数据，且从字节0开始。

图 30.1.19 寄存器 CAN_TDLxR 各位描述

该寄存器用来存储将要发送的数据，这里只能存储低 4 个字节，另外还有一个寄存器 CAN_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《STM32 中文参考手册 》22.9.3 节（447 页）。

第七个，我们介绍 CAN 过滤器模式寄存器（CAN_FM1R），该寄存器各位描述如图 30.1.20 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				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
FBM15	FBM14	FBM13	FBM12	FBM11	FBM10	FBM9	FBM8	FBM7	FBM6	FBM5	FBM4	FBM3	FBM2	FBM1	FBM0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留位，硬件强制为0
位13:0	**FBMx：**过滤器模式 (Filter mode) 过滤器组x的工作模式。 0: 过滤器组x的2个32位寄存器工作在标识符屏蔽位模式； 1: 过滤器组x的2个32位寄存器工作在标识符列表模式。 *注：位27:14只出现在互联型产品中，其它产品为保留位。*

图 30.1.20 寄存器 CAN_FM1R 各位描述

该寄存器用于设置各滤波器组的工作模式，对 28 个滤波器组的工作模式，都可以通过该寄存器设置，不过该寄存器必须在过滤器处于初始化模式下（CAN_FMR 的 FINIT 位=1），才可以进行设置。对 STM32F103ZET6 来说，只有[13:0]这 14 个位有效。

第八个，我们介绍 CAN 过滤器位宽寄存器(CAN_FS1R)，该寄存器各位描述如图 30.1.21 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				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
FSC15	FSC14	FSC13	FSC12	FSC11	FSC10	FSC9	FSC8	FSC7	FSC6	FSC5	FSC4	FSC3	FSC2	FSC1	FSC0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留位，硬件强制为0
位13:0	**FSCx：**过滤器位宽设置 (Filter scale configuration) 过滤器组x(13~0)的位宽。 0：过滤器位宽为2个16位； 1：过滤器位宽为单个32位。 *注：位27:14只出现在互联型产品中，其它产品为保留位。*

图 30.1.21 寄存器 CAN_FS1R 各位描述

该寄存器用于设置各滤波器组的位宽，对 28 个滤波器组的位宽设置，都可以通过该寄存器实现。该寄存器也只能在过滤器处于初始化模式下进行设置。对 STM32F103ZET6 来说，同样只有[13:0]这 14 个位有效。

第九个，我们介绍 CAN 过滤器 FIFO 关联寄存器（CAN_FFA1R），该寄存器各位描述如图 30.1.22 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
				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31:14	保留位，硬件强制为0。
位13:0	**FFAx**：过滤器位宽设置 (Filter FIFO assignment for filter x) 报文在通过了某过滤器的过滤后，将被存放到其关联的FIFO中。 0：过滤器被关联到FIFO0； 1：过滤器被关联到FIFO1。 *注：位27:14只出现在互联型产品中，其它产品为保留位。*

图 30.1.22 寄存器 CAN_FFA1R 各位描述

该寄存器设置报文通过滤波器组之后，被存入的 FIFO，如果对应位为 0，则存放到 FIFO0；如果为 1，则存放到 FIFO1。该寄存器也只能在过滤器处于初始化模式下配置。

第十个，我们介绍 CAN 过滤器激活寄存器（CAN_FA1R），该寄存器各位对应滤波器组和前面的几个寄存器类似，这里就不列出了，对对应位置 1，即开启对应的滤波器组；置 0 则关闭该滤波器组。

最后，我们介绍 CAN 的过滤器组 i 的寄存器 x（CAN_FiRx）（互联产品中 i=0~27，其它产品中 i＝0~13；x=1/2）。该寄存器各位描述如图 30.1.23 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FB31	FB30	FB29	FB28	FB27	FB26	FB25	FB24	FB23	FB22	FB21	FB20	FB19	FB18	FB17	FB16
rw	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
FB15	FB14	FB13	FB12	FB11	FB10	FB9	FB8	FB7	FB6	FB5	FB4	FB3	FB2	FB1	FB0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

在所有的配置情况下：

位31:0	**FB[31:0]**：过滤器位 (Filter bits) **标识符模式** 寄存器的每位对应于所期望的标识符的相应位的电平。 0: 期望相应位为显性位； 1: 期望相应位为隐性位。 **屏蔽位模式** 寄存器的每位指示是否对应的标识符寄存器位一定要与期望的标识符的相应位一致。 0: 不关心，该位不用于比较； 1: 必须匹配，到来的标识符位必须与滤波器对应的标识符寄存器位相一致。

图 30.1.23 寄存器 CAN_FiRx 各位描述

每个滤波器组的 CAN_FiRx 都由 2 个 32 位寄存器构成，即：CAN_FiR1 和 CAN_FiR2。根据过滤器位宽和模式的不同设置，这两个寄存器的功能也不尽相同。关于过滤器的映射，功能描述和屏蔽寄存器的关联，请参见图 30.1.11。

关于 CAN 的介绍，就到此结束了。接下来，我们看看本章我们将实现的功能，及 CAN 的配置步骤。

本章，我们通过 KEY_UP 按键（即 WK_UP 案件）选择 CAN 的工作模式（正常模式/环回模式），然后通过 KEY0 控制数据发送，并通过查询的办法，将接收到的数据显示在 LCD 模块

上。如果是环回模式，我们不需要 2 个开发板。如果是正常模式，我们就需要 2 个精英开发板，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：

HAL 库中 CAN 相关的函数在文件 stm32f1xx_hal_can.c 和对应的头文件 stm32f1xx_hal_can.h 中。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，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 为上拉输入（CAN_RX 引脚）PA12 为复用输出（CAN_TX 引脚），并使能 PA 口的时钟。具体配置过程如下：

```
GPIO_InitTypeDef GPIO_Initure;
__HAL_RCC_CAN1_CLK_ENABLE();                //使能 CAN1 时钟
__HAL_RCC_GPIOA_CLK_ENABLE();               //开启 GPIOA 时钟
GPIO_Initure.Pin=GPIO_PIN_12;               //PA12
GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //推挽复用
GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;    //高速
HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化

GPIO_Initure.Pin=GPIO_PIN_11;               //PA11
GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;       //推挽复用
HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化
```

这里需要提醒一下，CAN 发送接受引脚是哪些口，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。

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 HAL_CAN_Init 用来初始化 CAN 的工作模式以及波特率，HAL_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 HAL_CAN_Init 函数的声明：

```
HAL_StatusTypeDef HAL_CAN_Init(CAN_HandleTypeDef *hcan)
```

该函数入口参数只有 hcan 一个，为 CAN_HandleTypeDef 结构体指针类型，接下来我们看看结构体 CAN_HandleTypeDef 定义：

```
typedef struct
{
    CAN_TypeDef                 *Instance;
    CAN_InitTypeDef             Init;
    __IO HAL_CAN_StateTypeDef   State;
```

```
    __IO uint32_t                         ErrorCode;
}CAN_HandleTypeDef;
```

该结构体除了 State ，ErrorCode 两个 HAL 库处理状态过程变量之外，只有两个成员变量需要我们外部设置。

第一个成员变量 Instance 位寄存器基地址，这里我们使用 CAN1，设置为 CAN1 即可。

第二个成员变量 Init，它是 CAN_InitTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
  uint32_t Prescaler;
  uint32_t Mode;
  uint32_t SyncJumpWidth;
  uint32_t TimeSeg1;
  uint32_t TimeSeg2;
  FunctionalState TimeTriggeredMode;
  FunctionalState AutoBusOff;
  FunctionalState AutoWakeUp;
  FunctionalState AutoRetransmission;
  FunctionalState ReceiveFifoLocked;
  FunctionalState TransmitFifoPriority;
} 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 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 Prescaler，SyncJumpWidth;，TimeSeg1 和 TimeSeg2 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_HandleTypeDef    CAN1_Handler;                  //CAN1 句柄
CAN1_Handler.Init.Prescaler=brp;                    //分频系数(Fdiv)为 brp+1
CAN1_Handler.Init.Mode=mode;                        //模式设置
CAN1_Handler.Init.SyncJumpWidth=tsjw;
    //重新同步跳跃宽度(Tsjw)为 tsjw+1 个时间单位 CAN_SJW_1TQ~CAN_SJW_4TQ
CAN1_Handler.Init.TimeSeg1=tbs1;
    //tbs1 范围 CAN_BS1_1TQ~CAN_BS1_16TQ
CAN1_Handler.Init.TimeSeg2=tbs2;
    //tbs2 范围 CAN_BS2_1TQ~CAN_BS2_8TQ
CAN1_Handler.Init.TimeTriggeredMode=DISABLE; //非时间触发通信模式
CAN1_Handler.Init.AutoBusOff=DISABLE;               //软件自动离线管理
CAN1_Handler.Init.AutoWakeUp=DISABLE;
    //睡眠模式通过软件唤醒(清除 CAN->MCR 的 SLEEP 位)
CAN1_Handler.Init.AutoRetransmission=ENABLE;  //禁止报文自动传送
CAN1_Handler.Init.ReceiveFifoLocked=DISABLE;  //报文不锁定,新的覆盖旧的
```

```
CAN1_Handler.Init.TransmitFifoPriority=DISABLE; //优先级由报文标识符决定
if(HAL_CAN_Init(&CAN1_Handler)!=HAL_OK)     //初始化
    return 1;
return 0;
```

HAL 库通用提供了 MSP 初始化回调函数，CAN 回调函数为：

```
void HAL_CAN_MspInit(CAN_HandleTypeDef* hcan);
```

该回调函数一般用来编写时钟使能，IO 初始化以及 NVIC 等配置。

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 HAL 库中，提供了函数 HAL_CAN_ConfigFilter 用来初始化 CAN 的滤波器相关参数。HAL_CAN_ConfigFilter 函数体中，在初始化滤波器之前，会设置 CAN_FMR 寄存器的 FINIT 位为 1 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 位为 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 HAL_CAN_ConfigFilter 函数的声明：

```
HAL_StatusTypeDef HAL_CAN_ConfigFilter(CAN_HandleTypeDef *hcan,
                                       CAN_FilterTypeDef *sFilterConfig)
```

该函数有 2 个入口参数，第一个入口参数 hcan 这里就不多讲了。接下来看看第二个入口参数 sFilterConfig，它是 CAN_FilterTypeDef 结构体指针类型，用来设置滤波器相关参数，结构体 CAN_FilterTypeDef 定义为：

```
typedef struct
{
  uint32_t FilterIdHigh;
  uint32_t FilterIdLow;
  uint32_t FilterMaskIdHigh;
  uint32_t FilterMaskIdLow;
  uint32_t FilterFIFOAssignment;
  uint32_t FilterBank;
  uint32_t FilterMode;
  uint32_t FilterScale;
  uint32_t FilterActivation;
  uint32_t SlaveStartFilterBank;
} CAN_FilterTypeDef;
```

结构体一共有 10 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 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_RX_FIFO0。

第 6 个成员变量 FilterBank 用来设置初始化的过滤器组，取值范围为 0~13 或 0~27。

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 FilterActivation 就很明了了，用来激活该过滤器。
第 10 个成员变量用来设置 CAN 起始存储区。
过滤器初始化参考实例代码：

```
CAN_FilterTypeDef  sFilterConfig;
/*配置 CAN 过滤器*/
sFilterConfig.FilterBank = 0;                       //过滤器 0
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;                //32 位 ID
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;            //32 位 MASK
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;//过滤器 0 关联到 FIFO0
sFilterConfig.FilterActivation = ENABLE;            //激活滤波器 0
sFilterConfig.SlaveStartFilterBank = 14;
//过滤器配置
if (HAL_CAN_ConfigFilter(&CAN1_Handler, &sFilterConfig) != HAL_OK)
{
  while(1){}
}
```

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。HAL 库中提供了发送和接受消息的函数。发送消息的函数是：

```
HAL_StatusTypeDef HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan,
                    CAN_TxHeaderTypeDef *pHeader, uint8_t aData[], uint32_t *pTxMailbox)
```

这个函数比较好理解，入口参数 hcan，为 CAN_HandleTypeDef 结构体指针类型，pHeader 为发送的指针，aData 是待发送数据，pTxMailbox 为发送邮箱指针。

接收消息的函数是：

```
HAL_StatusTypeDef HAL_CAN_GetRxMessage(CAN_HandleTypeDef *hcan,
                    uint32_t RxFifo, CAN_RxHeaderTypeDef *pHeader, uint8_t aData[])
```

第一个入口参数为 CAN 句柄，第二个为 FIFO 号，然后是接收指针及数据存放的地址。我们接受之后，只需要读取 pHeader 便可获取接收数据和相关信息。

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。

30.2 硬件设计

本章要用到的硬件资源如下：

1）指示灯 DS0
2）KEY0 和 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 STM32F1 与 TJA1050 的连接关系，如

图 30.2.1 所示：

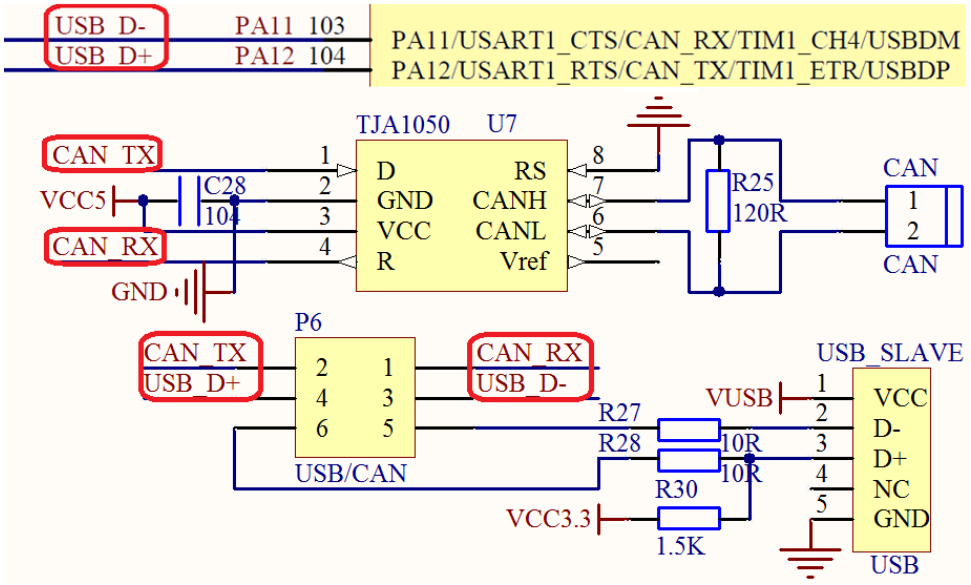


图 30.2.1 STM32F1 与 TJA1050 连接电路图

从上图可以看出：STM32F1 的 CAN 通过 P6 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，在精英 STM32F103 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：CAN 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 P6 排针的连接，通过跳线帽将 PA11 和 PA12 分别连接到 CRX 和 CTX 上面，如图 30.2.2 所示：

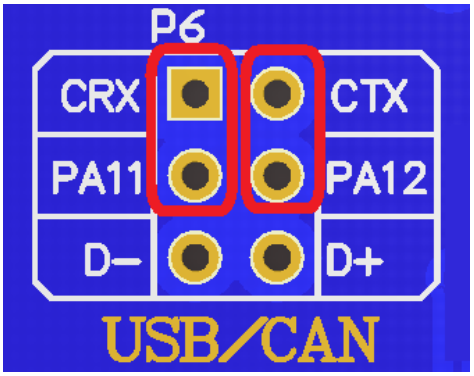


图 30.2.2 硬件连接示意图

最后，我们用 2 根导线将两个开发板 CAN 端子的 CAN_L 和 CAN_L，CAN_H 和 CAN_H 连接起来。这里注意不要接反了（CAN_L 接 CAN_H），接反了会导致通讯异常！！

30.3 软件设计

打开 CAN 通信实验的工程可以看到，我们增加了文件 can.c 以及头文件 can.h，同时 CAN 相关的 HAL 库函数和定义分布在文件 stm32f1xx_hal_can.c 和头文件 stm32f1xx_hal_can.h 中。

打开 can.c 文件，代码如下：

```
CAN_HandleTypeDef     CAN1_Handler;        //CAN1 句柄
CAN_TxHeaderTypeDef   TxHeader;            //发送
CAN_RxHeaderTypeDef   RxHeader;            //接收
```

```
////CAN 初始化
//tsjw:重新同步跳跃时间单元.范围:CAN_SJW_1TQ~CAN_SJW_4TQ
//tbs2:时间段 2 的时间单元.    范围:CAN_BS2_1TQ~CAN_BS2_8TQ;
//tbs1:时间段 1 的时间单元.    范围:CAN_BS1_1TQ~CAN_BS1_16TQ
//brp :波特率分频器.范围:1~1024; tq=(brp)*tpclk1
//波特率=Fpclk1/((tbs1+tbs2+1)*brp); 其中 tbs1 和 tbs2 我们只用关注标识符上标志的序号，
//例如 CAN_BS2_1TQ，我们就认为 tbs2=1 来计算即可。
//mode:CAN_MODE_NORMAL,普通模式;CAN_MODE_LOOPBACK,回环模式;
//Fpclk1 的时钟在初始化的时候设置为 36M,如果设置 CAN1_Mode_Init
//(CAN_SJW_1tq,CAN_BS2_8tq,CAN_BS1_9tq,4,CAN_MODE_LOOPBACK);
//则波特率为:36M/((8+9+1)*4)=500Kbps
//返回值:0,初始化 OK;
//      其他,初始化失败;
u8 CAN1_Mode_Init(u32 tsjw,u32 tbs2,u32 tbs1,u16 brp,u32 mode)
{
    CAN_InitTypeDef        CAN1_InitConf;
    CAN1_Handler.Instance=CAN1;
    CAN1_Handler.Init = CAN1_InitConf;
    CAN1_Handler.Init.Prescaler=brp;                    //分频系数(Fdiv)为 brp+1
    CAN1_Handler.Init.Mode=mode;                        //模式设置
    CAN1_Handler.Init.SyncJumpWidth=tsjw;  //重新同步跳跃宽度(Tsjw)为 tsjw+1 个时间
                                           //单位  CAN_SJW_1TQ~CAN_SJW_4TQ
    CAN1_Handler.Init.TimeSeg1=tbs1;       //tbs1 范围 CAN_BS1_1TQ~CAN_BS1_16TQ
    CAN1_Handler.Init.TimeSeg2=tbs2;       //tbs2 范围 CAN_BS2_1TQ~CAN_BS2_8TQ
    CAN1_Handler.Init.TimeTriggeredMode=DISABLE; //非时间触发通信模式
    CAN1_Handler.Init.AutoBusOff=DISABLE;          //软件自动离线管理
    CAN1_Handler.Init.AutoWakeUp=DISABLE;          //睡眠模式通过软件唤醒(清除
                                                   //CAN->MCR 的 SLEEP 位)
    CAN1_Handler.Init.AutoRetransmission=ENABLE;   //禁止报文自动传送
    CAN1_Handler.Init.ReceiveFifoLocked=DISABLE;   //报文不锁定,新的覆盖旧的
    CAN1_Handler.Init.TransmitFifoPriority=DISABLE; //优先级由报文标识符决定
    if(HAL_CAN_Init(&CAN1_Handler)!=HAL_OK)     //初始化
        return 1;
    return 0;
}

//CAN 底层驱动，引脚配置，时钟配置，中断配置
//此函数会被 HAL_CAN_Init()调用
//hcan:CAN 句柄
void HAL_CAN_MspInit(CAN_HandleTypeDef* hcan)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_CAN1_CLK_ENABLE();                    //使能 CAN1 时钟
```

```
    __HAL_RCC_GPIOA_CLK_ENABLE();                   //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_12;                   //PA12
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;              //推挽复用
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;        //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);             //初始化

    GPIO_Initure.Pin=GPIO_PIN_11;                   //PA11
    GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;           //推挽复用
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);             //初始化
}

void CAN_Config(void)
{
  CAN_FilterTypeDef  sFilterConfig;
  /*配置 CAN 过滤器*/
  sFilterConfig.FilterBank = 0;                     //过滤器 0
  sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
  sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
  sFilterConfig.FilterIdHigh = 0x0000;              //32 位 ID
  sFilterConfig.FilterIdLow = 0x0000;
  sFilterConfig.FilterMaskIdHigh = 0x0000;          //32 位 MASK
  sFilterConfig.FilterMaskIdLow = 0x0000;
  sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;//过滤器 0 关联到 FIFO0
  sFilterConfig.FilterActivation = ENABLE;          //激活滤波器 0
  sFilterConfig.SlaveStartFilterBank = 14;
  //过滤器配置
  if (HAL_CAN_ConfigFilter(&CAN1_Handler, &sFilterConfig) != HAL_OK)
  {
    while(1){}
  }

  //启动 CAN 外围设备
  if (HAL_CAN_Start(&CAN1_Handler) != HAL_OK)
  {
    while(1){}
  }

  //激活可以 RX 通知
  if (HAL_CAN_ActivateNotification(&CAN1_Handler,
          CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK)
  {
```

```
        while(1){}
    }
    /*配置传输过程*/
    TxHeader.StdId = 0x321;
    TxHeader.ExtId = 0x01;
    TxHeader.RTR = CAN_RTR_DATA;
    TxHeader.IDE = CAN_ID_STD;
    TxHeader.DLC = 2;
    TxHeader.TransmitGlobalTime = DISABLE;
}

//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 i=0;
      u32 TxMailbox;
      u8 message[8];
      TxHeader.StdId=0X12;                    //标准标识符
      TxHeader.ExtId=0x12;                    //扩展标识符(29 位)
      TxHeader.IDE=CAN_ID_STD;                //使用标准帧
      TxHeader.RTR=CAN_RTR_DATA;              //数据帧
      TxHeader.DLC=len;
      for(i=0;i<len;i++)
      {
            message[i]=msg[i];
      }
      if(HAL_CAN_AddTxMessage(&CAN1_Handler, &TxHeader, message,
                              &TxMailbox) != HAL_OK)//发送
      {
            return 1;
      }
      while(HAL_CAN_GetTxMailboxesFreeLevel(&CAN1_Handler) != 3) {}
      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;
    u8    RxData[8];
    if(HAL_CAN_GetRxFifoFillLevel(&CAN1_Handler, CAN_RX_FIFO0) != 1)
    {
        return 0xF1;
    }
    if(HAL_CAN_GetRxMessage(&CAN1_Handler, CAN_RX_FIFO0,
                                        &RxHeader, RxData) != HAL_OK)
    {
        return 0xF2;
    }
    for(i=0;i<RxHeader.DLC;i++)
    buf[i]=RxData[i];
    return RxHeader.DLC;
}
```

此部分代码，我们挑其中几个比较重要的函数简单介绍下，首先是：CAN1_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 31.1 节末尾的介绍来初始化的，本章中，我们设计滤波器组 0 工作在 32 位标识符屏蔽模式，从设计值可以看出，该滤波器是不会对任何标识符进行过滤的，因为所有的标识符位都被设置成不需要关心，这样设计，主要是方便大家实验。

第二个函数，Can_Send_Msg 函数。该函数用于 CAN 报文的发送，主要是设置标识符 ID 等信息，写入数据长度和数据，并请求发送，实现一次报文的发送。

第三个函数，Can_Receive_Msg 函数。用来接受数据并且将接受到的数据存放到 buf 中。

最后我们看看主函数，代码如下：

```
int main(void)
{
    u8 key;
    u8 i=0,t=0;
    u8 cnt=0;
    u8 canbuf[8];
    u8 res;
    u8 mode=1;
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    usmart_dev.init(84);                //初始化 USMART
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    LCD_Init();                         //初始化 LCD FSMC 接口
    CAN1_Mode_Init(CAN_SJW_1TQ,CAN_BS2_8TQ,CAN_BS1_9TQ,4,
```

```
                    CAN_MODE_LOOPBACK); //CAN 初始化,波特率 500Kbps
CAN_Config();
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16," ELITE STM32");
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,"2019/9/18");
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;
        if(mode==0)
        CAN1_Mode_Init(CAN_SJW_1TQ,CAN_BS2_8TQ,CAN_BS1_9TQ,4,
                       CAN_MODE_NORMAL);    //回环模式,波特率 500Kbps
        else if(mode==1)
        CAN1_Mode_Init(CAN_SJW_1TQ,CAN_BS2_8TQ,CAN_BS1_9TQ,4,
                       CAN_MODE_LOOPBACK);  //回环模式,波特率 500Kbps
        CAN_Config();
        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,"Nnormal 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 < 9)//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_8TQ,CAN_BS1_9TQ,4,
                CAN_MODE_NORMAL);
```

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。

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 精英 STM32F103 上，得到如图 30.4.1 所示：

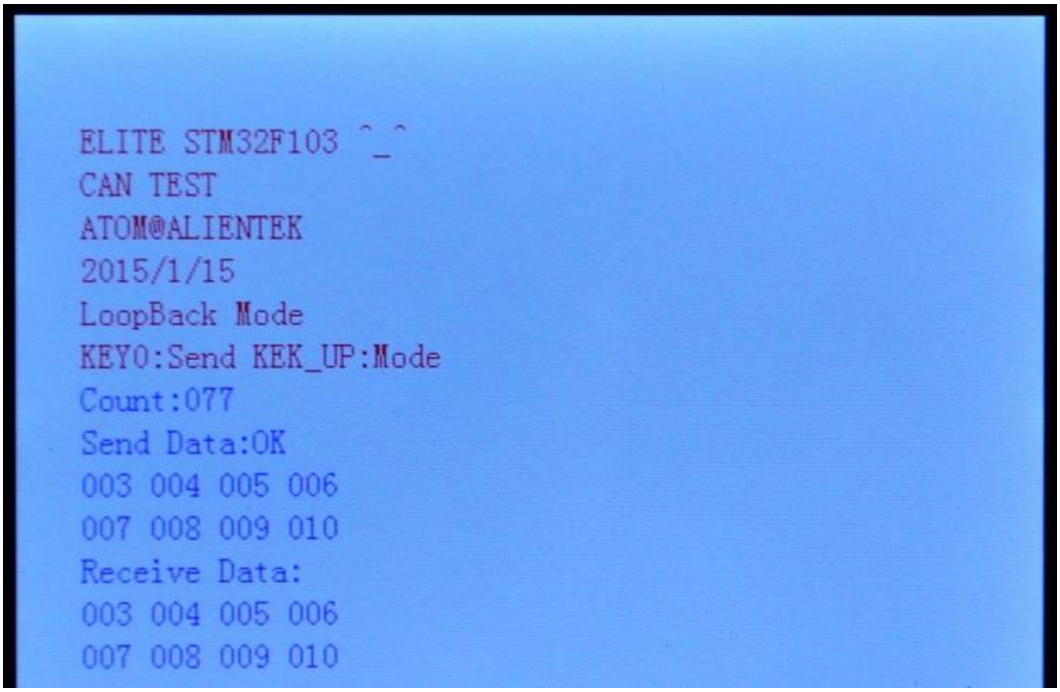


图 30.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。默认我们是设置的环回模式，此时，我们按下 KEY0 就可以在 LCD 模块上面看到自发自收的数据（如上图所示），如果我们选择普通模式（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 30.4.2 和图 30.4.3 所示：

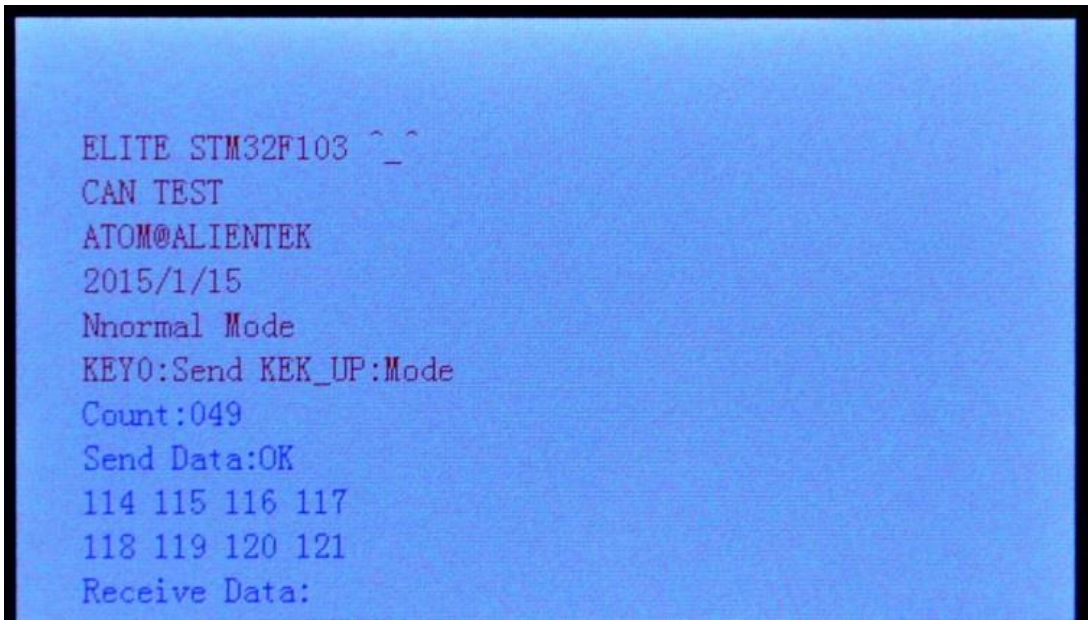


图 30.4.2 CAN 普通模式发送数据

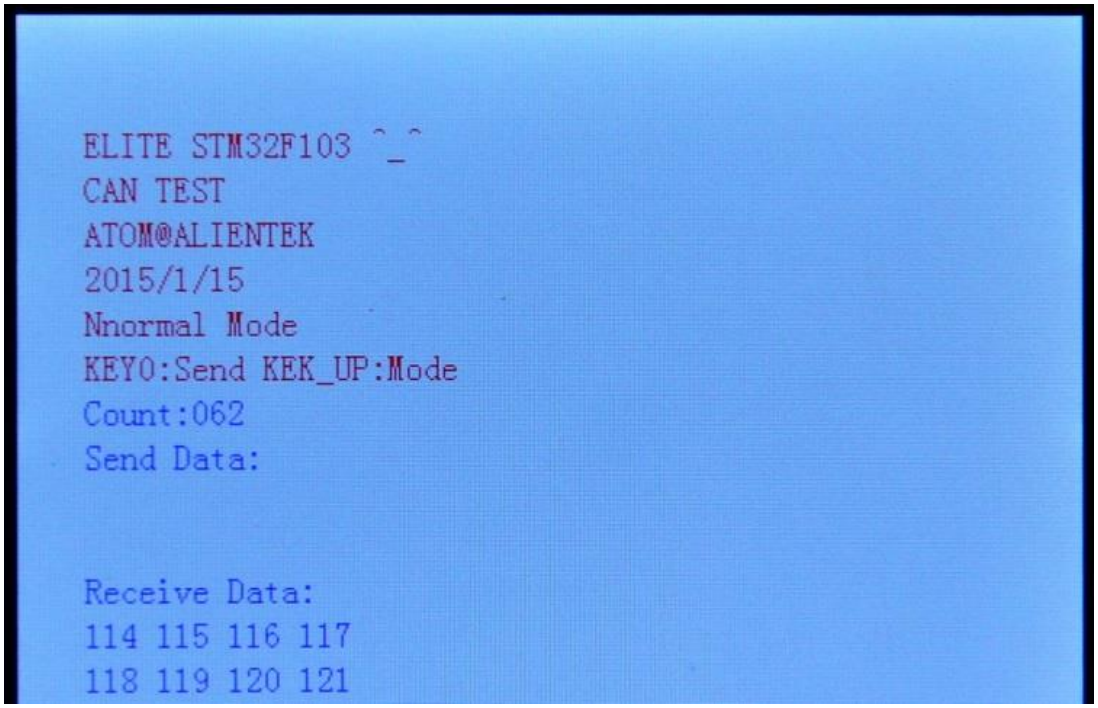


图 30.4.3 CAN 普通模式接收数据

图 30.4.2 来自开发板 A，发送了 8 个数据，图 30.4.3 来自开发板 B，收到了来自开发板 A 的 8 个数据。

另外，利用 USMART 测试的部分，我们这里就不做介绍了，大家可自行验证下。

第三十一章 触摸屏实验

本章，我们将介绍如何使用 STM32F1 来驱动触摸屏，ALIENTEK 精英 STM32F103 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 ALIENTKE 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：

31.1 电阻与电容触摸屏简介
31.2 硬件设计
31.3 软件设计
31.4 下载验证

31.1 触摸屏简介

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。

31.1.1 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℃～+85℃。

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。

31.1.2 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），所以后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 31.1.2.1 所示：

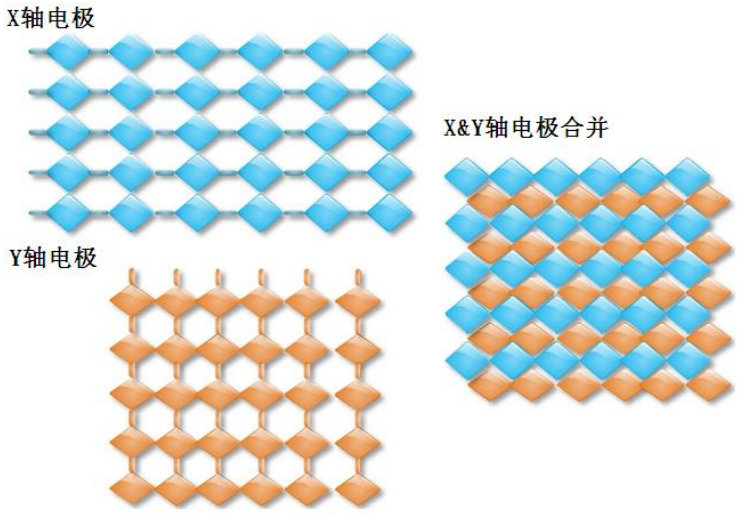


图 31.1.2.1 投射式电容屏电极矩阵示意图

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/FT5206 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，本例程支持 ALIENTEK 的 4.3 寸屏模块和新版的 7 寸屏模块（采用 SSD1963+FT5206 方案），电容触摸驱动 IC，这里只介绍 OTT2001A 和 GT9147，GT811/FT5206 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 STM32F1 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，请参考第二十八章。

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 31.1.2.1 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 31.1.2.1 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 31.1.2.2 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 31.1.2.2 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 31.1.2.3 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 31.1.2.3 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，这里是有多少有效触点值就是多少。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 31.1.2.4 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 坐标高 8 位
0X8152	触点 1 y 坐标低 8 位	0X8153	触点 1 y 坐标高 8 位
0X8154	触点 1 触摸尺寸低 8 位	0X8155	触点 1 触摸尺寸高 8 位

表 31.1.2.4 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。

31.2 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 4.3 寸电容触摸屏，则先读取芯片 ID，判断是不是 GT9147，如果是则执行 GT9147 的初始化代码，如果不是，则执行 OTT2001A 的初始化代码；如果是 7 寸电容触摸屏（仅支持新款 7 寸屏，使用 SSD1963+FT5206 方案），则执行 FT5206 的初始化代码，在初始化电容触摸屏完成后，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 STM32F1 的连接图，在前面都已经介绍了，这里我们只针对 TFTLCD 模块与 STM32F1 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 STM32F1 连接，连接电路图如图 31.2.1 所示：

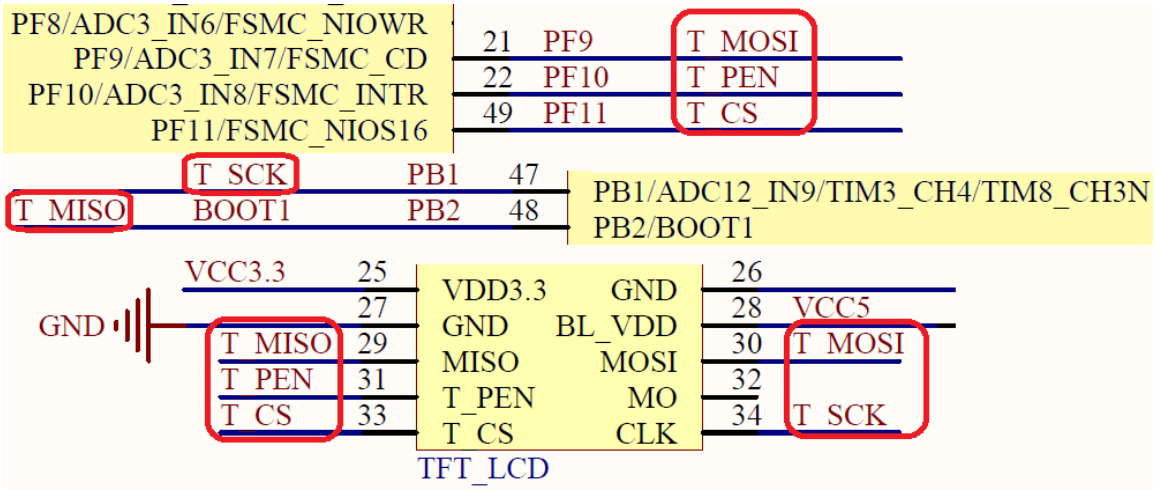


图 31.2.1 触摸屏与 STM32F1 的连接图

从图中可以看出，T_MOSI、T_MISO、T_SCK、T_CS 和 T_PEN 分别连接在 STM32F1 的：PF9、PB2、PB1、PF11 和 PF10 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/FT5206 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/FT5206 的数据，对于 OTT2001A/FT5206 没有用到中断信号（CT_INT），所以同 STM32F1 的连接，最少只需要 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。

31.3 软件设计

打开上一章的工程，由于本章不要用到 USMART 和 CAN 相关代码，所以，先去掉 USMART 相关代码和 can.c（此时 HARDWARE 组剩下：led.c、ILI93xx.c 和 key.c）。不过，本章要用到 24C02，所以还得添加 myiic.c 和 24cxx.c 到 HARDWARE 组下。

然后，在 HARDWARE 文件夹下新建一个 TOUCH 文件夹。然后新建一个 touch.c、touch.h、ctiic.c、ctiic.h、ott2001a.c、ott2001a.h、gt9147.c、gt9147.h、ft5206.c 和 ft5206.h 等十个文件，并保存在 TOUCH 文件夹下，并将这个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，在里面输入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。绝对坐标系统的特点是每一次定位坐标与上一次定位坐标没有关系，每次触摸的数据通过校准转为屏幕上的坐标，不管在什么情况下，触摸屏这套坐标在同一点的输出数据是稳定的。不过由于技术原理的原因，并不能保证同一点触摸每一次采样数据相同，不能保证绝对坐标定位，点不准，这就是触摸屏最怕出现的问题：漂移。对于性能质量好的触摸屏来说，漂移的情况出现并不是很严重。所以很多应用触摸屏的系统启动后，进入应用程序前，先要执行校准程序。 通常应用程序中使用的 LCD 坐标是以像素为单位的。比如说：左上角的坐标是一组非 0 的数值，比如（20，20），而右下角的坐标为（220，300）。这些点的坐标都是以像素为单位的，而从触摸屏中读出的是点的物理坐标，其坐标轴的方向、XY 值的比例因子、偏移量都与 LCD 坐标不同，所以，需要在程序中把物理坐标首先转换为像素坐标，然后再赋给 POS 结构，达到坐标转换的目的。

校正思路：在了解了校正原理之后，我们可以得出下面的一个从物理坐标到像素坐标的转换关系式：

LCDx=xfac*Px+xoff;

LCDy=yfac*Py+yoff;

其中(LCDx,LCDy)是在 LCD 上的像素坐标，（Px,Py）是从触摸屏读到的物理坐标。xfac，yfac 分别是 X 轴方向和 Y 轴方向的比例因子，而 xoff 和 yoff 则是这两个方向的偏移量。

这样我们只要事先在屏幕上面显示 4 个点（这四个点的坐标是已知的），分别按这四个点就可以从触摸屏读到 4 个物理坐标，这样就可以通过待定系数法求出 xfac、yfac、xoff、yoff 这四个参数。我们保存好这四个参数，在以后的使用中，我们把所有得到的物理坐标都按照这个关系式来计算，得到的就是准确的屏幕坐标。达到了触摸屏校准的目的。

TP_Adjust 就是根据上面的原理设计的校准函数，注意该函数里面多次使用了 lcddev.width 和 lcddev.height，用于坐标设置，主要是为了兼容不同尺寸的 LCD（比如 320*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)                   //4.3 寸电容触摸屏
    {
        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 if(lcddev.id==0X1963)              //7 寸电容触摸屏
    {
        FT5206_Init();
        tp_dev.scan=FT5206_Scan;            //扫描函数指向 GT9147 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
    {
        GPIO_InitTypeDef GPIO_Initure;
        __HAL_RCC_GPIOB_CLK_ENABLE();               //开启 GPIOB 时钟
        __HAL_RCC_GPIOF_CLK_ENABLE();               //开启 GPIOF 时钟

        //PB1
        GPIO_Initure.Pin=GPIO_PIN_1;                //PB1
        GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;      //推挽输出
        GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
        GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;    //高速
        HAL_GPIO_Init(GPIOB,&GPIO_Initure);

        //PB2
        GPIO_Initure.Pin=GPIO_PIN_2;                //PB2
        GPIO_Initure.Mode=GPIO_MODE_INPUT;          //上拉输入
```

```
        HAL_GPIO_Init(GPIOB,&GPIO_Initure);

        //PF9,11
        GPIO_Initure.Pin=GPIO_PIN_9|GPIO_PIN_11;  //PF9,11
        GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;    //推挽输出
        HAL_GPIO_Init(GPIOF,&GPIO_Initure);

        //PF10
        GPIO_Initure.Pin=GPIO_PIN_10;             //PF10
        GPIO_Initure.Mode=GPIO_MODE_INPUT;        //输入
        GPIO_Initure.Pull=GPIO_PULLUP;            //上拉
        HAL_GPIO_Init(GPIOF,&GPIO_Initure);

        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_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、OTT2001A_Scan 或 FT5206_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，保存 touch.c 文件，并把该文件加入到 HARDWARE 组下。接下来打开 touch.h 文件，在该文件里面输入如下代码：

```
#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(仅电阻屏需要，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 touchtype;
}_m_tp_dev;
extern _m_tp_dev tp_dev;          //触屏控制器在 touch.c 里面定义
//电阻/电容屏芯片连接引脚
#define PEN          PFin(10)          //PF10 INT
#define DOUT         PBin(2)           //PB2   MISO
#define TDIN         PFout(9)          //PF9   MOSI
#define TCLK         PBout(1)          //PB1   SCLK
#define TCS          PFout(11)         //PF11   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)
u8 TP_Read_XY(u16 *x,u16 *y);         //双方向读取(X+Y)
u8 TP_Read_XY2(u16 *x,u16 *y);        //带加强滤波的双方向坐标读取
void TP_Drow_Touch_Point(u16 x,u16 y,u16 color);//画一个坐标校准点
void TP_Draw_Big_Point(u16 x,u16 y,u16 color);//画一个大点
void TP_Save_Adjdata(void);           //保存校准参数
u8 TP_Get_Adjdata(void);              //读取校准参数
void TP_Adjust(void);                 //触摸屏校准
void TP_Adj_Info_Show(u16 x0,u16 y0,u16 x1,u16 y1,u16 x2,u16 y2,u16 x3,u16 y3,u16 fac)
//电阻屏/电容屏  共用函数
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/变量屏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，与第二十八章的 myiic.c 和 myiic.h 基本一样，这里就不单独介绍了，记得把 ctiic.c 加入 HARDWARE 组下。接下来看看：ott2001a.c，在该文件输入如下代码：

```
//向 OTT2001A 写入一次数据
//reg:起始寄存器地址
//buf:数据缓缓存区
//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_Initure;
    __HAL_RCC_GPIOF_CLK_ENABLE();              //开启 GPIOF 时钟
    GPIO_Initure.Pin=GPIO_PIN_11;              //PF11
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;     //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;             //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;   //高速
    HAL_GPIO_Init(GPIOF,&GPIO_Initure);

    GPIO_Initure.Pin=GPIO_PIN_10;              //PF10
    GPIO_Initure.Mode=GPIO_MODE_INPUT;         //输入
    HAL_GPIO_Init(GPIOF,&GPIO_Initure);

    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];
u8 i=0;
u8 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，则完全是我们在上面介绍的方法，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，保存 ott2001a.c 文件，并把该文件加入到 HARDWARE 组下，ott2001a.h 代码我们这里就不贴出来了，大家参考开发板光盘源码即可。

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 regval=0;
    GPIO_InitTypeDef GPIO_Iniure;
    __HAL_RCC_GPIOF_CLK_ENABLE();              //开启 GPIOF 时钟
    GPIO_Iniure.Pin=GPIO_PIN_11;               //PF11
    GPIO_Iniure.Mode=GPIO_MODE_OUTPUT_PP;      //推挽输出
    GPIO_Iniure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Iniure.Speed=GPIO_SPEED_FREQ_HIGH;    //高速
    HAL_GPIO_Init(GPIOF,&GPIO_Iniure);

    GPIO_Iniure.Pin=GPIO_PIN_10;               //PF10
    GPIO_Iniure.Mode=GPIO_MODE_INPUT;          //输入
    HAL_GPIO_Init(GPIOF,&GPIO_Iniure);

    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 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 CTP_Scan 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&0X80&&((mode&0XF)<6))
        {
            temp=0;
            GT9147_WR_Reg(GT_GSTID_REG,&temp,1);//清标志
        }
        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
            tempsta=tp_dev.sta;             //保存当前的 tp_dev.sta 值
            tp_dev.sta=(~temp)|TP_PRES_DOWN|TP_CATH_PRES;
            tp_dev.x[4]=tp_dev.x[0];        //保存触点 0 的数据
            tp_dev.y[4]=tp_dev.y[0];
            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];
                    }
                    //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]>lcddev.width||tp_dev.y[0]>lcddev.height)//非法数据(坐标超出了)
            {
                if((mode&0XF)>1) //有其他点有数据,复制第二个点的数据到第一个点.
                {
                    tp_dev.x[0]=tp_dev.x[1];
                    tp_dev.y[0]=tp_dev.y[1];
                    t=0;            //触发一次,则会最少连续监测 10 次,从而提高命中率
                }else               //非法数据,则忽略此次数据(还原原来的)
                {
                    tp_dev.x[0]=tp_dev.x[4];
                    tp_dev.y[0]=tp_dev.y[4];
                    mode=0X80;
                    tp_dev.sta=tempsta;//恢复 tp_dev.sta
                }
            }else 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&0X8F)==0X80)//无触摸点按下
    {
        if(tp_dev.sta&TP_PRES_DOWN) //之前是被按下的
        {
            tp_dev.sta&=~(1<<7);        //标记按键松开
        }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，不过这里加入了防错误数据处理。大家看源码即可。

保存 gt9147.c 文件，并把该文件加入到 HARDWARE 组下。另外，ft5206.c 和 ft5206.h 的代码，我们就不再介绍了，请大家参考光盘本例程源码。

最后我们打开 test.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
            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<CT_MAX_TOUCH;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]<16)
                        {
                            Load_Drow_Dialog();//清除
                        }
                    }
                }else lastpos[t][0]=0XFFFF;
            }
            delay_ms(5);i++;
            if(i%20==0)LED0=!LED0;
        }
    }
    int main(void)
    {
        HAL_Init();                             //初始化 HAL 库
        Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
        delay_init(72);                         //初始化延时函数
        uart_init(115200);                      //初始化串口
        LED_Init();                             //初始化 LED
        KEY_Init();                             //初始化按键
        LCD_Init();                             //初始化 LCD
        tp_dev.init();                          //触摸屏初始化
        POINT_COLOR=RED;
        LCD_ShowString(30,50,200,16,16," ELITE STM32");
        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,"2019/9/19");
        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。

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 精英 STM32F103 上，电阻触摸屏测试如图 31.4.1 所示界面：



图 31.4.1 电阻触摸屏测试程序运行效果

图中我们在电阻屏上画了一些内容，右上角的 RST 可以用来清屏，点击该区域，即可清屏重画。另外，按 KEY0 可以进入校准模式，如果发现触摸屏不准，则可以按 KEY0，进入校准，

重新校准一下，即可正常使用。

如果是电容触摸屏，测试界面如图 31.4.2 所示：

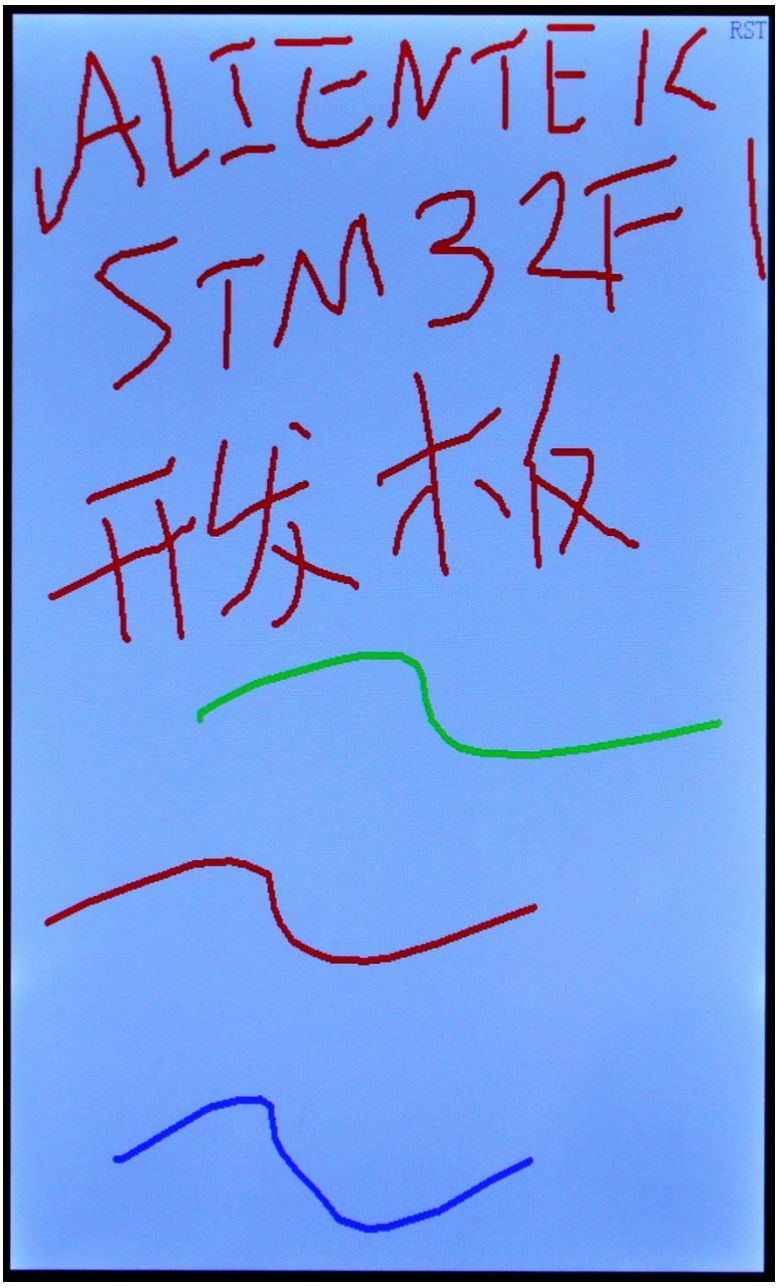


图 31.4.2 电容触摸屏测试界面

图中，同样输入了一些内容。电容屏支持多点触摸，每个点的颜色都不一样，图中的波浪线就是三点触摸画出来的，最多可以 5 点触摸。注意：电容触摸屏支持：ALIENTEK 4.3 寸电容触摸屏模块或者 ALIENTEK 新款 7 寸电容触摸屏模块（SSD1963+FT5206 方案），老款的 7 寸电容触摸屏模块（CPLD+GT811 方案）本例程不支持!!

同样，按右上角的 RST 标志，可以清屏。电容屏无需校准，所以按 KEY0 无效。KEY0 校准仅对电阻屏有效。

第三十二章　红外遥控实验

本章，我们将向大家介绍如何通过 STM32 来解码红外遥控器的信号。ALIENTK 精英 STM32F103 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 STM32F1 的输入捕获功能，解码开发板标配的这个红外遥控器的编码信号，并将解码后的键值 TFTLCD 模块上显示出来。本章分为如下几个部分：

32.1 红外遥控简介
32.2 硬件设计
32.3 软件设计
32.4 下载验证

32.1 红外遥控简介

红外遥控是一种无线、非接触控制技术，具有抗干扰能力强，信息传输可靠，功耗低，成本低，易实现等显著优点，被诸多电子设备特别是家用电器广泛采用，并越来越多的应用到计算机系统中。

由于红外线遥控不具有像无线电遥控那样穿过障碍物去控制被控对象的能力，所以，在设计红外线遥控器时，不必要像无线电遥控器那样，每套(发射器和接收器)要有不同的遥控频率或编码(否则，就会隔墙控制或干扰邻居的家用电器)，所以同类产品的红外线遥控器，可以有相同的遥控频率或编码，而不会出现遥控信号“串门”的情况。这对于大批量生产以及在家用电器上普及红外线遥控提供了极大的方面。由于红外线为不可见光，因此对环境影响很小，再由红外光波动波长远小于无线电波的波长，所以红外线遥控不会影响其他家用电器，也不会影响临近的无线电设备。

红外遥控的编码目前广泛使用的是：NEC Protocol 的 PWM(脉冲宽度调制)和 Philips RC-5 Protocol 的 PPM(脉冲位置调制)。ALIENTEK 精英 STM32F103 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 32.1.1 所示：

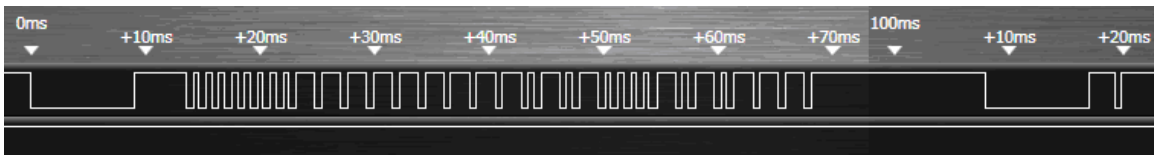


图 32.1.1 按键“▽”所对应的红外波形

从图 32.1.1 中可以看到，其地址码为 0，控制码为 168。可以看到在 100ms 之后，我们还收到了几个脉冲，这是 NEC 码规定的连发码(由 9ms 低电平+2.5m 高电平+0.56ms 低电平+97.94ms 高电平组成)，如果在一帧数据发送完毕之后，按键仍然没有放开，则发射重复码，即连发码，可以通过统计连发码的次数来标记按键按下的长短/次数。

第十五章我们曾经介绍过利用输入捕获来测量高电平的脉宽，本章解码红外遥控信号，刚好可以利用输入捕获的这个功能来实现遥控解码。关于输入捕获的介绍，请参考第十五章的内容。

32.2 硬件设计

本实验采用定时器的输入捕获功能实现红外解码，本章实验功能简介：开机在 LCD 上显示一些信息之后，即进入等待红外触发，如过接收到正确的红外信号，则解码，并在 LCD 上显示键值和所代表的意义，以及按键次数等信息。同样我们也是用 LED0 来指示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0

2） TFTLCD 模块

3） 红外接收头

4） 红外遥控器

前两个，在之前的实例已经介绍过了，遥控器属于外部器件，遥控接收头在板子上，与 MCU 的连接原理图如 32.2.1 所示：

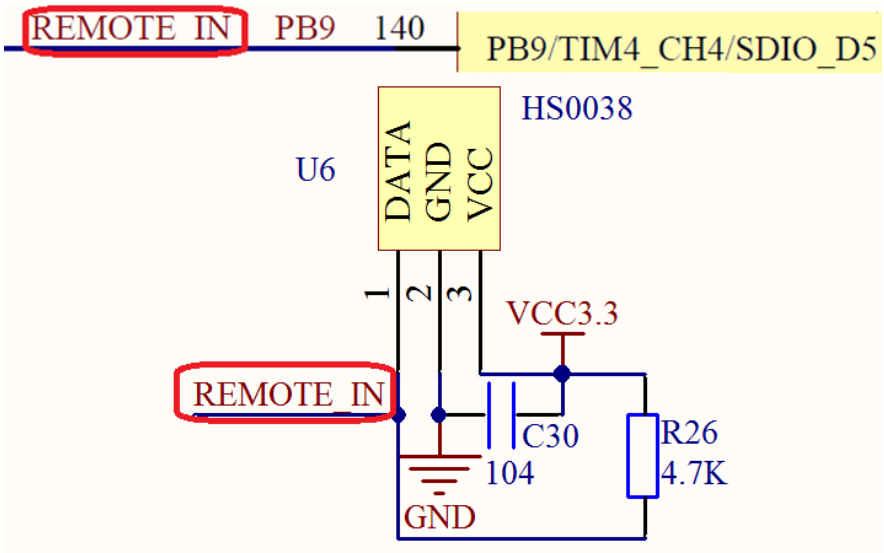


图 32.2.1 红外遥控接收头与 STM32 的连接电路图

红外遥控接收头连接在 STM32 的 PB9（TIM4_CH4）上。硬件上不需要变动，只要程序将 TIM4_CH4 设计为输入捕获，然后将收到的脉冲信号解码就可以了。

开发板配套的红外遥控器外观如图 32.2.2 所示：



图 32.2.2 红外遥控器

32.3 软件设计

打开我们光盘的红外遥控器实验工程，可以看到我们添加了 remote.c 和 remote.h 两个文件，同时因为我们使用的是输入捕获，所以还用到定时器相关的库函数源文件 stm32f1xx_hal_tim.c 和头文件 stm32f1xx_hal_tim.h。

打开 remote.c 文件，代码如下：

```
TIM_HandleTypeDef TIM4_Handler;         //定时器 4 句柄
//红外遥控初始化
//设置 IO 以及 TIM4_CH4 的输入捕获
void Remote_Init(void)
{
    TIM_IC_InitTypeDef TIM4_CH1Config;

    TIM4_Handler.Instance=TIM4;                                    //通用定时器 4
    TIM4_Handler.Init.Prescaler=(72-1);              //预分频器,1M 的计数频率,1us 加 1.
    TIM4_Handler.Init.CounterMode=TIM_COUNTERMODE_UP; //向上计数器
    TIM4_Handler.Init.Period=10000;                                //自动装载值
    TIM4_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_IC_Init(&TIM4_Handler);

    //初始化 TIM1 输入捕获参数
    TIM4_CH1Config.ICPolarity=TIM_ICPOLARITY_RISING;      //上升沿捕获
    TIM4_CH1Config.ICSelection=TIM_ICSELECTION_DIRECTTI;//映射到 TI4 上
    TIM4_CH1Config.ICPrescaler=TIM_ICPSC_DIV1;              //配置输入分频，不分频
    TIM4_CH1Config.ICFilter=0x03;                  //IC4F=0003 8 个定时器时钟周期滤波
   HAL_TIM_IC_ConfigChannel(&TIM4_Handler,&TIM4_CH1Config,TIM_CHANNEL_4);
                                                                   //配置 TIM4 通道 4
    HAL_TIM_IC_Start_IT(&TIM4_Handler,TIM_CHANNEL_4); //开始捕获 TIM4 的通道 4
    __HAL_TIM_ENABLE_IT(&TIM4_Handler,TIM_IT_UPDATE);    //使能更新中断
}

//定时器 1 底层驱动，时钟使能，引脚配置
//此函数会被 HAL_TIM_IC_Init()调用
//htim:定时器 1 句柄
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_TIM4_CLK_ENABLE();                //使能 TIM4 时钟
    __HAL_RCC_GPIOB_CLK_ENABLE();               //开启 GPIOB 时钟
    GPIO_Initure.Pin=GPIO_PIN_9;                //PB9
    GPIO_Initure.Mode=GPIO_MODE_AF_INPUT;       //复用输入
    GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;//高速
```

```
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);
    HAL_NVIC_SetPriority(TIM4_IRQn,1,3);//设置中断优先级，抢占优先级 1，子优先级 3
    HAL_NVIC_EnableIRQ(TIM4_IRQn);          //开启 ITM4 中断
}

//遥控器接收状态
//[7]:收到了引导码标志
//[6]:得到了一个按键的所有信息
//[5]:保留
//[4]:标记上升沿是否已经被捕获
//[3:0]:溢出计时器
u8    RmtSta=0;
u16 Dval;       //下降沿时计数器的值
u32 RmtRec=0;     //红外接收到的数据
u8    RmtCnt=0;     //按键按下的次数

//定时器 4 中端服务函数
void TIM4_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM4_Handler);//定时器共用处理函数
}

//定时器更新（溢出）中断回调函数
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim->Instance==TIM4)
    {
        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
            }
        }
    }
}

//定时器输入捕获中断回调函数
```

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)//捕获中断发生时执行
{
    if(htim->Instance==TIM4)
    {
        if(RDATA)//上升沿捕获
        {
            TIM_RESET_CAPTUREPOLARITY(&TIM4_Handler,TIM_CHANNEL_4);
                //一定要先清除原来的设置!!
            TIM_SET_CAPTUREPOLARITY(&TIM4_Handler,TIM_CHANNEL_4,
             TIM_ICPOLARITY_FALLING);//CC4P=1  设置为下降沿捕获
            __HAL_TIM_SET_COUNTER(&TIM4_Handler,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=HAL_TIM_ReadCapturedValue(&TIM4_Handler,TIM_CHANNEL_4);
                                                    //读取 CCR4 也可以清 CC4IF 标志位
            TIM_RESET_CAPTUREPOLARITY(&TIM4_Handler,TIM_CHANNEL_4);
                                                    //一定要先清除原来的设置!!
            TIM_SET_CAPTUREPOLARITY(&TIM4_Handler,TIM_CHANNEL_4,
                        TIM_ICPOLARITY_RISING);//配置 TIM4 通道 4 上升沿捕获
            if(RmtSta&0X10)                         //完成一次高电平捕获
            {
                if(RmtSta&0X80)//接收到了引导码
                {
                    if(Dval>300&&Dval<800)          //560 为标准值,560us
                    {
                        RmtRec<<=1;                 //左移一位.
                        RmtRec|=0;                  //接收到 0
                    }else if(Dval>1400&&Dval<1800)  //1680 为标准值,1680us
                    {
                        RmtRec<<=1;                 //左移一位.
                        RmtRec|=1;                  //接收到 1
                    }else if(Dval>2200&&Dval<2600)
                      //得到按键键值增加的信息  2500 为标准值 2.5ms
                    {
                        RmtCnt++;                   //按键次数增加 1 次
                        RmtSta&=0XF0;               //清空计时器
                    }
                }else if(Dval>4200&&Dval<4700)      //4500 为标准值 4.5ms
                {
                    RmtSta|=1<<7;                   //标记成功接收到了引导码
                    RmtCnt=0;                       //清除按键次数计数器
                }
```

```
                }
                RmtSta&=~(1<<4);
            }
        }
    }

    //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;
    }
```

我们介绍一下该部分代码，首先是 Remote_Init 函数，该函数用于初始化 IO 口，并配置 TIM4_CH4 为输入捕获，并设置其相关参数。TIM4_IRQHandler 函数是 TIM4 的中断服务函数，在该函数里面，实现对红外信号的高电平脉冲的捕获，同时根据我们之前简介的协议内容来解码 ，该函数用到几个全局变量，用于辅助解码，并存储解码结果。

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，清零定时器的计数器值，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;
    u8 t=0;
    u8 *str=0;
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
    delay_init(72);                 //初始化延时函数
    uart_init(115200);              //初始化串口
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    LCD_Init();                     //初始化 LCD FSMC 接口
    Remote_Init();                  //初始化 红外接收
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/19");
    LCD_ShowString(30,130,200,16,16,"KEYVAL:");
    LCD_ShowString(30,150,200,16,16,"KEYCNT:");
    LCD_ShowString(30,170,200,16,16,"SYMBOL:");
    POINT_COLOR=BLUE;//设置字体为蓝色
    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。

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 精英 STM32F103 上，可以看到 LCD 显示如图 32.4.1 所示的内容：

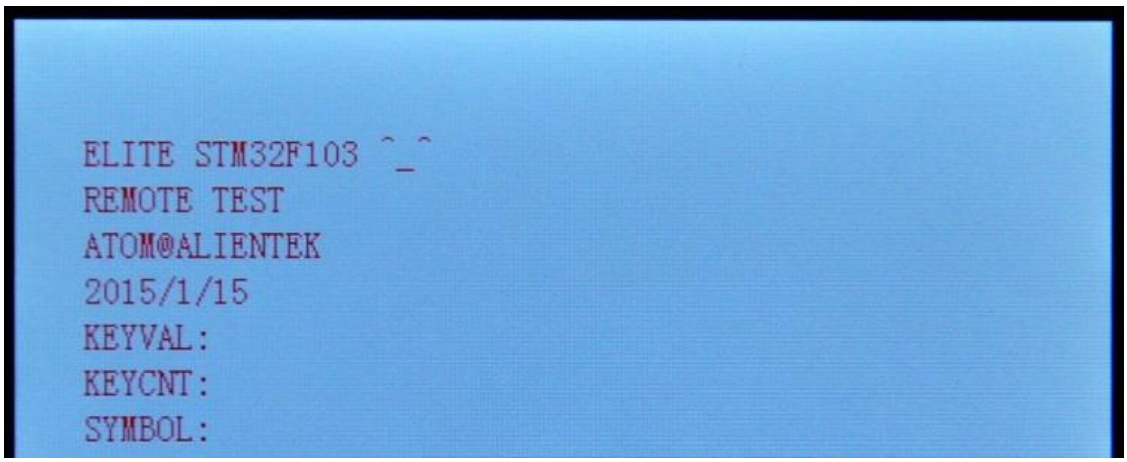


图 32.4.1 程序运行效果图

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 32.4.2 所示：

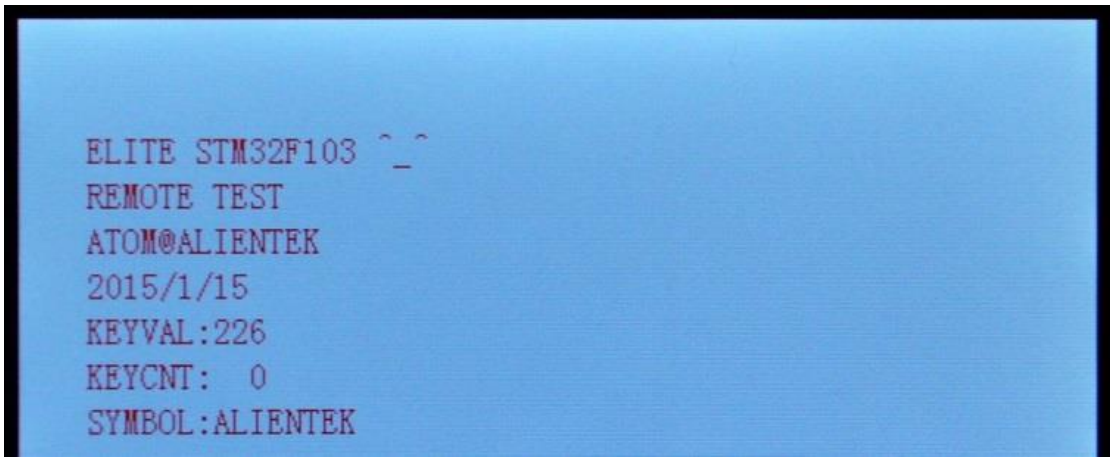


图 32.4.2 解码成功

第三十三章　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：

33.1 DS18B20 简介
33.2 硬件设计
33.3 软件设计
33.4 下载验证

33.1 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℃ ，精度为±0.5℃。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~l2 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 33.1.1 所示：

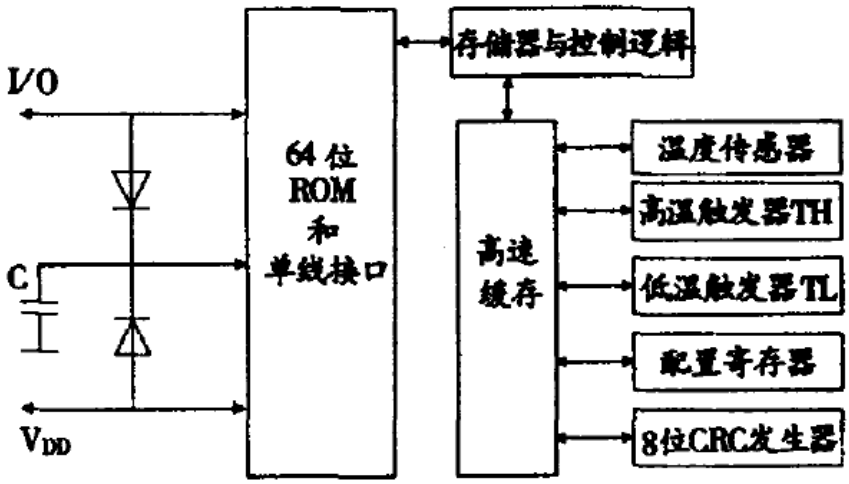


图 33.1.1 DS18B20 内部结构图

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。

33.2 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33.2.1所示：

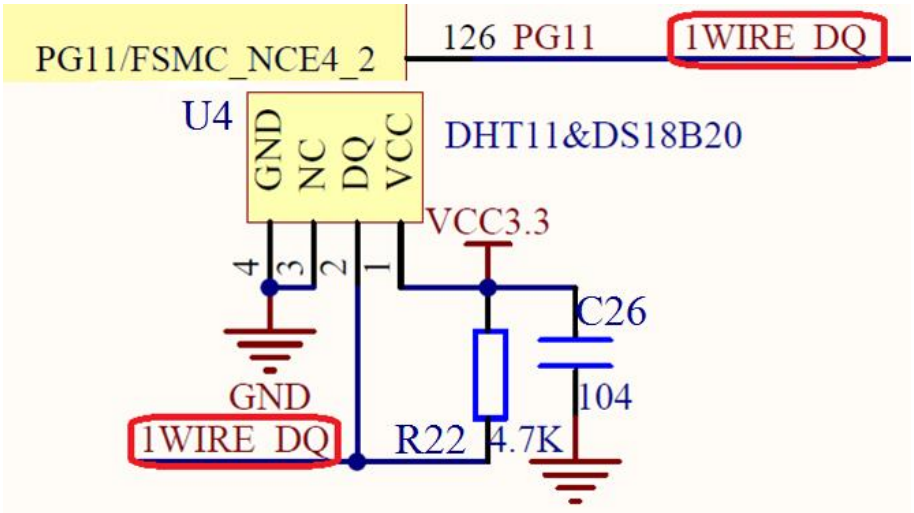


图 33.2.1 DS18B20 接口与 STM32 的连接电路图

从上图可以看出，我们使用的是 STM32 的 PG11 来连接 U4 的 DQ 引脚，图中 U4 为 DHT11（数字温湿度传感器）和 DS18B20 共用的一个接口，DHT11 我们将在下一章介绍。

DS18B20 只用到 U4 的 3 个引脚（U4 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 33.2.2 所示：

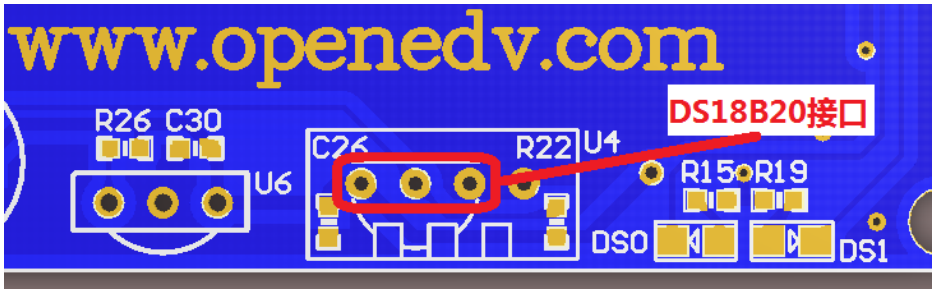


图 33.2.2 DS18B20 连接示意图

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。

33.3 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_Initure;
    __HAL_RCC_GPIOG_CLK_ENABLE();              //开启 GPIOG 时钟

    GPIO_Initure.Pin=GPIO_PIN_11;              //PG11
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;     //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;             //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;   //高速
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);        //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=(float)tem*0.625;             //转换
    if(temp)return tem;               //返回温度值
    else return -tem;
}
```

该部分代码就是根据我们前面介绍的单总线操作时序来读取 DS18B20 的温度值的，DS18B20 的温度通过 DS18B20_Get_Temp 函数读取，该函数的返回值为带符号的短整型数据，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;
    HAL_Init();                       //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);   //设置时钟,72M
    delay_init(72);                   //初始化延时函数
    uart_init(115200);                //初始化串口
    LED_Init();                       //初始化 LED
    KEY_Init();                       //初始化按键
    LCD_Init();                       //初始化 LCD FSMC 接口
    POINT_COLOR=RED;
```

```
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/19");
    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。

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 精英 STM32F1 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 33.4.1 所示：



图 33.4.1 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。

第三十四章　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 STM32F1 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：

34.1 DHT11 简介
34.2 硬件设计
34.3 软件设计
34.4 下载验证

34.1 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℃
- 精度 ：湿度±5%，温度±2℃
- 分辨率 ：湿度 1%，温度 1℃

DHT11 的管脚排列如图 34.1.1 所示：

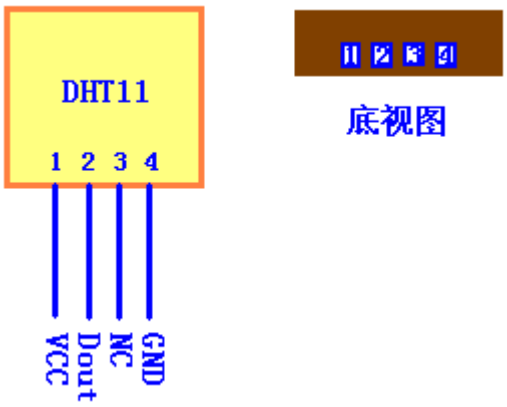


图 34.1.1 DHT11 管脚排列图

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 34.1.2 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 34.1.2 某次读取到 DHT11 的数据

由以上数据就可得到湿度和温度的值，计算方法：

湿度= byte4 . byte3=45.0 (%RH)

温度= byte2 . byte1=28.0 (℃)

校验= byte4+ byte3+ byte2+ byte1=73(=湿度+温度)(校验正确)

可以看出，DHT11 的数据格式是十分简单的，DHT11 和 MCU 的一次通信最大为 3ms 左右，建议主机连续读取时间间隔不要小于 100ms。

下面，我们介绍一下 DHT11 的传输时序。DHT11 的数据发送流程如图 34.1.3 所示：

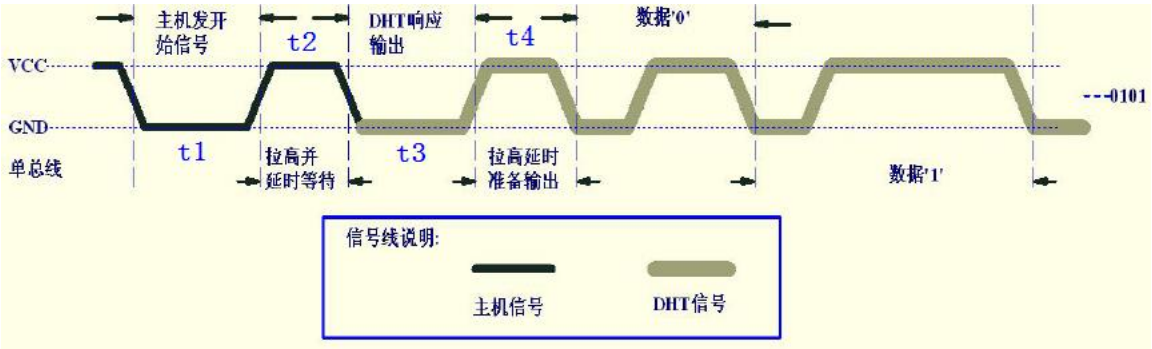


图 34.1.3 DHT11 数据发送流程

首先主机发送开始信号，即：拉低数据线，保持 t1（至少 18ms）时间，然后拉高数据线 t2（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 t3（40~50us）时间，作为响应信号，然后 DHT11 拉高数据线，保持 t4（40~50us）时间后，开始输出数据。

DHT11 输出数字‘0’的时序如图 34.1.4 所示：

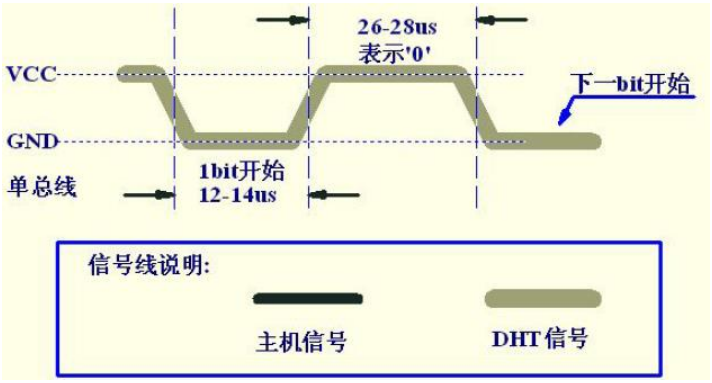


图 34.1.4 DHT11 数字‘0’时序

DHT11 输出数字‘1’的时序如图 34.1.5 所示：

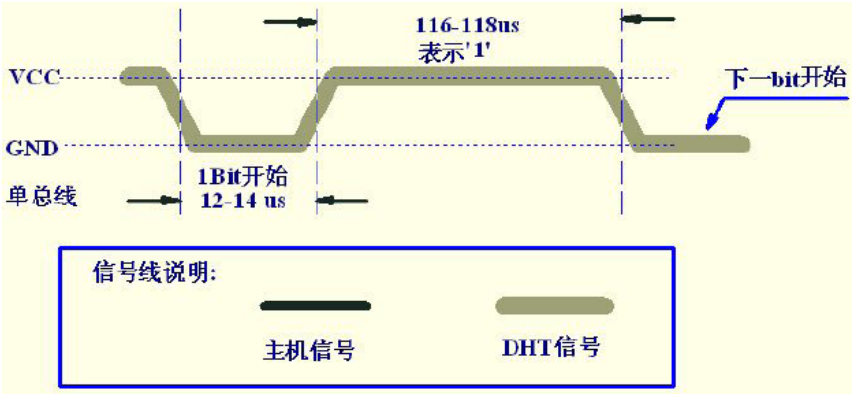


图 34.1.5 DHT11 数字‘1’时序

通过以上了解，我们就可以通过 STM32F1 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。

34.2 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 U4 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 STM32F1 来读取温湿度值了。连接示意图如图 34.2.1 所示：

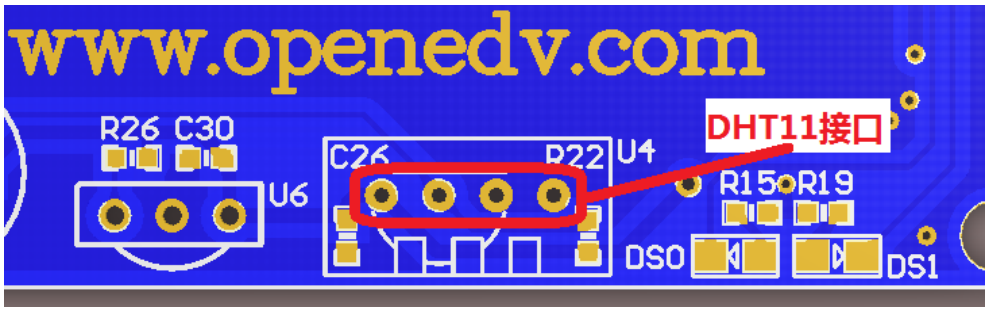


图 34.2.1 DHT11 连接示意图

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，然后然后插入如图所示的四个孔内就可以了。

34.3 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++) buf[i]=DHT11_Read_Byte();//读取 40 位数据
        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_Iniure;
    __HAL_RCC_GPIOG_CLK_ENABLE();           //开启 GPIOG 时钟
    GPIO_Iniure.Pin=GPIO_PIN_11;            //PG11
    GPIO_Iniure.Mode=GPIO_MODE_OUTPUT_PP;   //推挽输出
    GPIO_Iniure.Pull=GPIO_PULLUP;           //上拉
    GPIO_Iniure.Speed=GPIO_SPEED_FREQ_HIGH; //高速
    HAL_GPIO_Init(GPIOG,&GPIO_Iniure);      //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;
    u8 temperature;
    u8 humidity;
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    LCD_Init();                             //初始化 LCD FSMC 接口
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/19");
    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。

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 精英 STM32F1 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 34.4.1 所示：

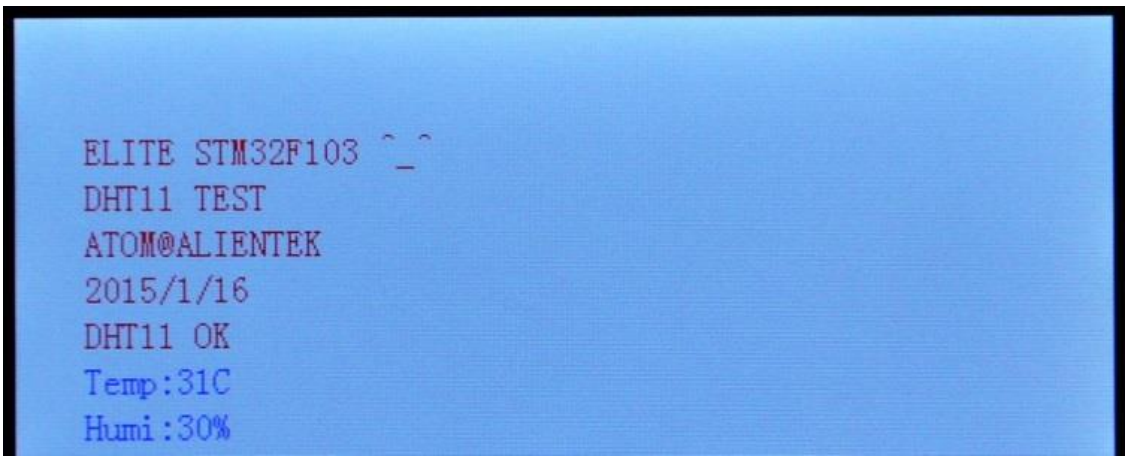


图 34.4.1 DHT11 实验效果图

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？

第三十五章 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 精英 STM32F1 开发板本身并不带 MPU6050 传感器，但是可以通过 ATK MODULE 接口，外扩 ATK-MPU6050 模块来实现本例程。

本章我们将使用 STM32F1 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：

35.1 MPU6050 简介
35.2 硬件设计
35.3 软件设计
35.4 下载验证

35.1 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。

35.1.1 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 35.1.1.1 所示：

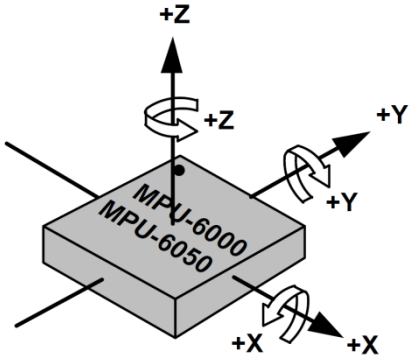


图 35.1.1.1 MPU6050 检测轴及其方向

MPU6050 的内部框图如图 35.1.1.2 所示：

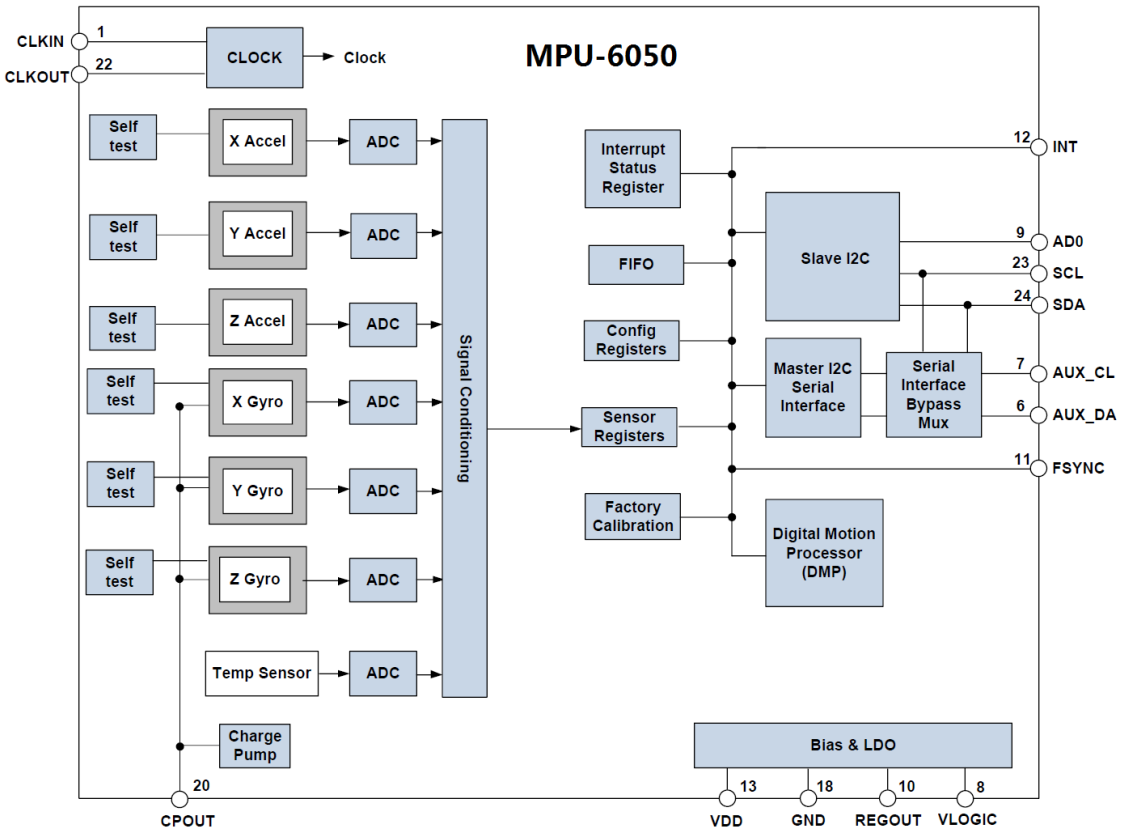


图 35.1.1.2 MPU6050 框图

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）!!

在精英 STM32F1 开发板上，我们通过 PA15 控制 ATK-MPU6050 模块 AD0 接 GND，因而选择 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（第二十七章，IIC 实验），这里就不再细说了。

接下来，我们介绍一下利用 STM32F1 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 STM32F1 通信，所以我们需要先初始化与 MPU6050 连接的 SDA 和 SCL 数据线。

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（0X6B）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 35.1.1.3 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 35.1.1.3 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。SLEEEP 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 35.1.1.1 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

表 35.1.1.1 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 35.1.4 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 35.1.1.4 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=16.4LSB/(°/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 35.1.1.5 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 35.1.1.5 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=16384LSB/g。

接下来，我看看 FIFO 使能寄存器，寄存器地址为：0X23，各位描述如图 35.1.1.6 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 35.1.1.6 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 35.1.1.7 所示：

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
19	25	SMPLRT_DIV[7:0]							

图 35.1.1.7 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 35.1.1.8 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 35.1.1.8 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 35.1.1.2 所示：

DLPF_CFG[2:0]	加速度传感器 Fs=1Khz		角速度传感器（陀螺仪）		
	带宽(Hz)	延迟（ms）	带宽(Hz)	延迟（ms）	Fs(Khz)
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

表 35.1.1.2 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 35.1.1.9 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 35.1.1.9 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，总共由 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：

Temperature = 36.53 + regval/340

其中，Temperature 为计算得到的温度值，单位为℃，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。

35.1.2 DMP 使用简介

经过 35.1.1 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/π，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 STM32F1 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 35.1.2.1 所示：

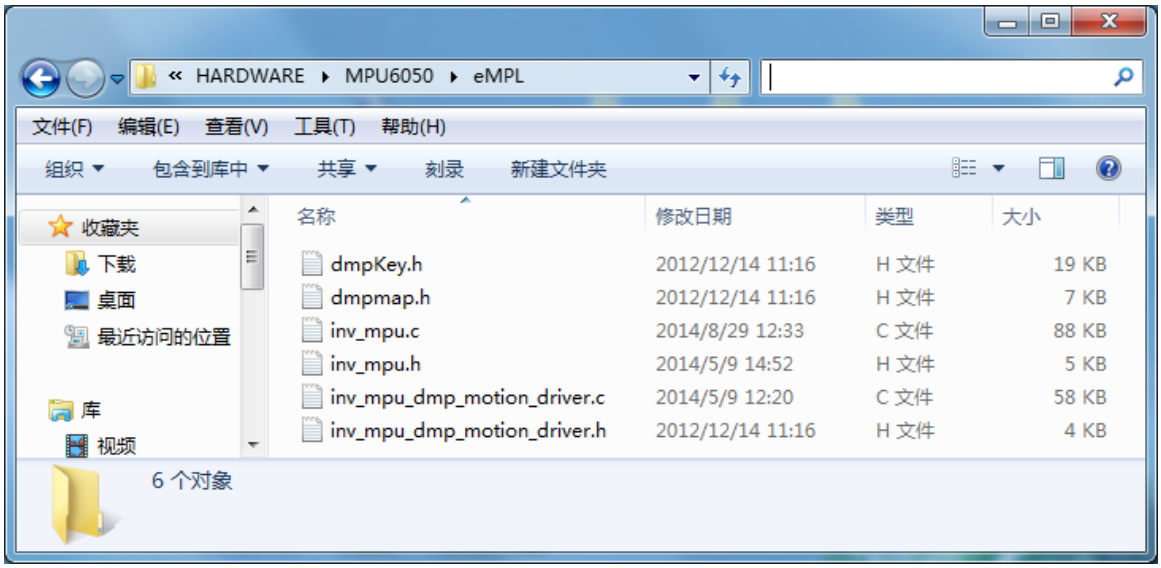


图 35.1.2.1 移植后的驱动库代码

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系列操作，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_stm32f10x_hd.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。

35.2 硬件设计

本实验采用精英 STM32F1 开发板的 ATK MODULE 接口连接 ATK-MPU6050 模块，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） ATK-MPU6050 模块
6） ATK MODULE 接口

前 4 个，在之前的实例已经介绍过了，这里我们介绍下 ATK-MPU6050 模块与精英 STM32F1 开发板的连接。ATK-MPU6050 模块原理图如图 35.2.1 所示：

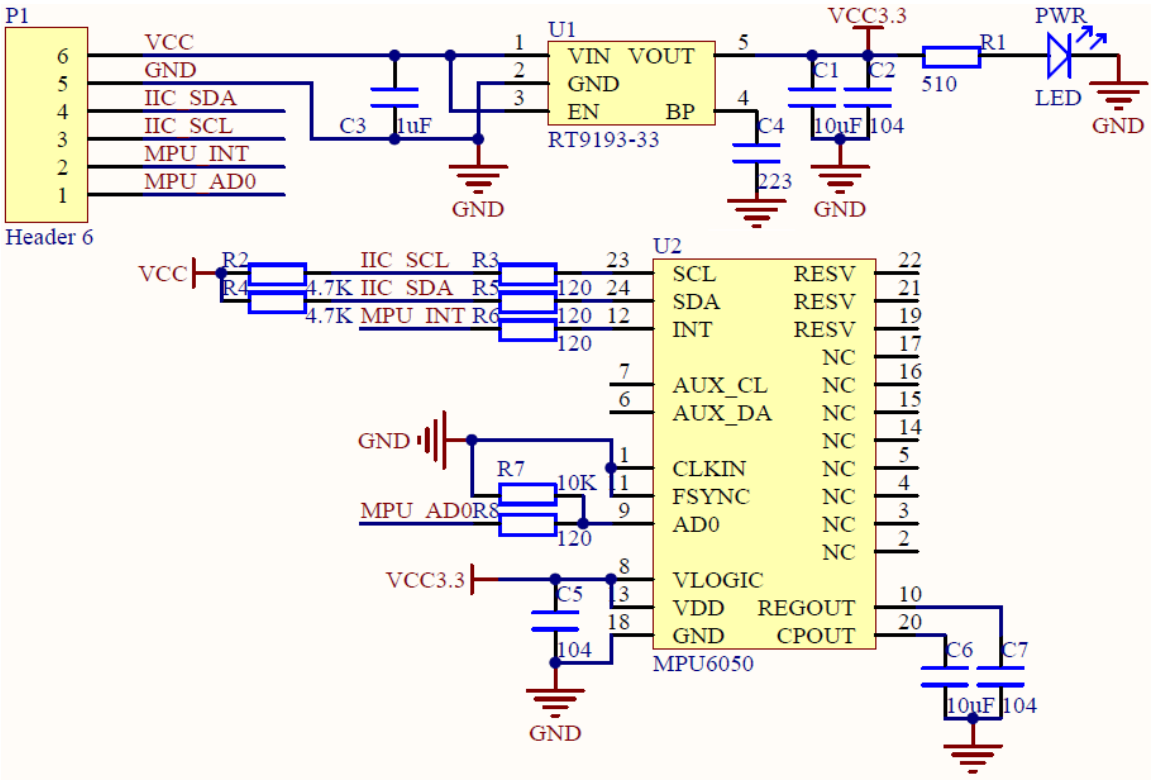


图 35.2.1 ATK-MPU6050 模块原理图

从上图可知，ATK-MPU6050 模块，通过 P1 排针与外部连接，引出了 VCC、GND、IIC_SDA、IIC_SCL、MPU_INT 和 MPU_AD0 等信号，其中，IIC_SDA 和 IIC_SCL 带了 4.7K 上拉电阻，外部可以不用再加上拉电阻了，另外 MPU_AD0 自带了 10K 下拉电阻，当 AD0 悬空时，默认 IIC 地址为（0X68）。模块的 P1 接口可以直接插在精英 STM32F1 开发板的 ATK MODULE 接口上，如图 35.2.2 所示：

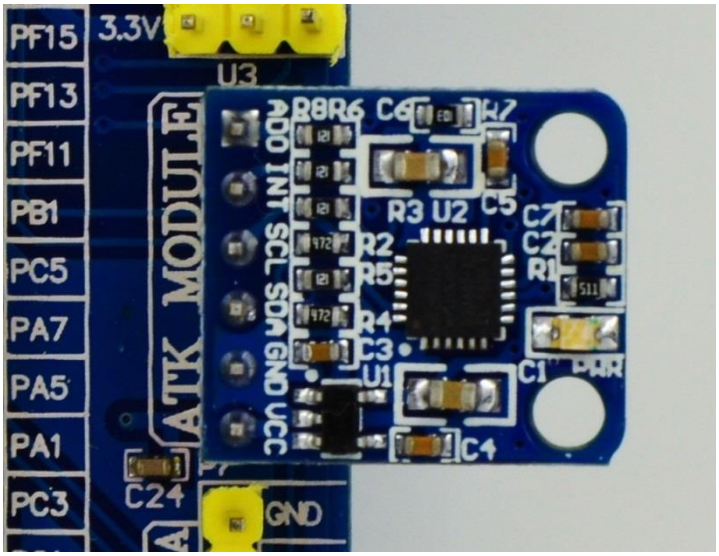


图 35.2.2 ATK-MPU6050 模块与开发板实物连接图

而 ATK MODULE 接口与 MCU 的连接原理图如图 35.2.3 所示：

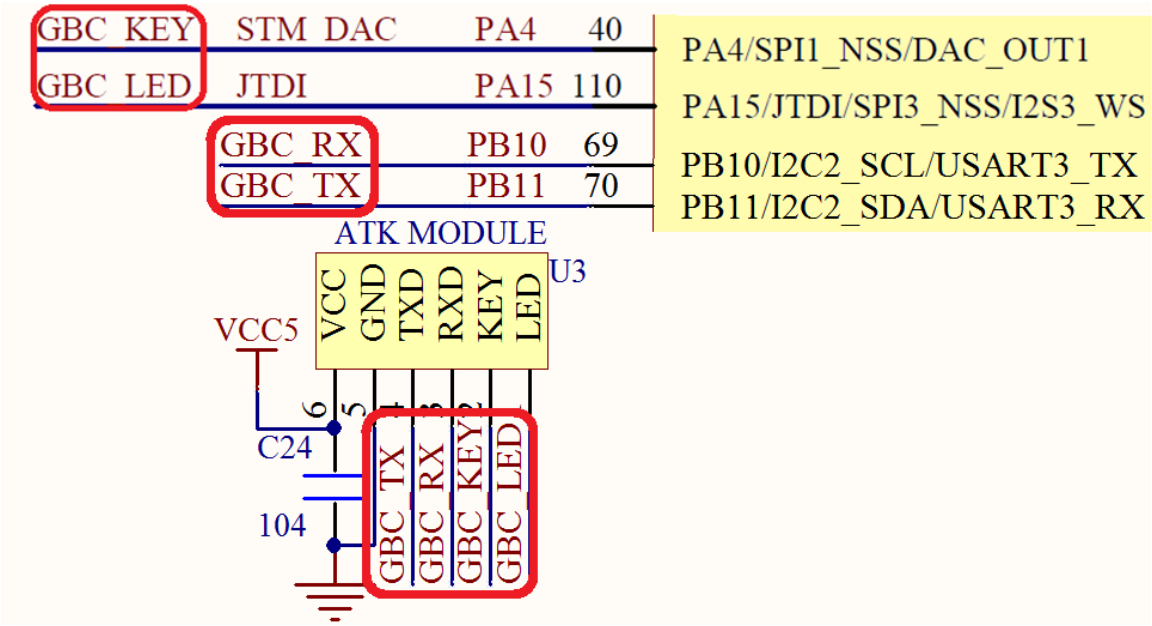


图 35.2.3 ATK MODULE 接口与 STM32F103 的连接原理图

从上图可以看出，ATK MODULE 接口，GBC_TX 接在 MCU 的 PB11，GBC_RX 接在 MCU 的 PB10，GBC_KEY 接在 MCU 的 PA4，GBC_LED 接在 MCU 的 PA15，他们是直接连接在一起的，不需要任何跳线。 特别注意：GBC_KEY 和 GBC_LED 与 STM_DAC 和 JTDI 共用 PA4 和 PA15，他们必须分时复用。

将 ATK-MPU6050 模块插入 ATK-MODULE 接口，那么模块的 IIC_SCL、IIC_SDA、MPU_INT 和 MPU_AD0 分别连接在 STM32 的 PB10、PB11、PA4 和 PA15 上面。不过，本例

程我们并没有用到中断。另外，MPU_AD0 通过 PA15 输出低电平，从而选则 MPU6050 的器件地址是：0X68。

35.3 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;
    GPIO_InitTypeDef GPIO_Iniure;
    __HAL_RCC_GPIOA_CLK_ENABLE();                   //开启 GPIOA 时钟
    GPIO_Iniure.Pin=GPIO_PIN_15;                    //PA15
    GPIO_Iniure.Mode=GPIO_MODE_OUTPUT_PP;           //推挽输出
    GPIO_Iniure.Pull=GPIO_PULLUP;                   //上拉
    GPIO_Iniure.Speed=GPIO_SPEED_FREQ_HIGH;         //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Iniure);
    __HAL_AFIO_REMAP_SWJ_DISABLE();       //禁止 JTAG,从而 PA15 可以做普通
                                          //IO 使用,否则 PA15 不能做普通 IO!!!
    MPU_AD0_CTRL=0;             //控制 MPU6050 的 AD0 脚为低电平,从机地址为:0X68
    MPU_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 轴为参考
```

```
            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，将 mpu6050.c 加入 HARDWARE 组下，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(__HAL_UART_GET_FLAG(&UART1_Handler,UART_FLAG_TC)==RESET){};
                                                    //循环发送,直到发送完毕
    HAL_UART_Transmit(&UART1_Handler,&c,1,1000);
}

//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];
    u8 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;
    usart1_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;
        usart1_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
        HAL_Init();                                 //初始化 HAL 库
        Stm32_Clock_Init(RCC_PLL_MUL9);             //设置时钟,72M
        delay_init(72);                             //初始化延时函数
        uart_init(500000);                          //初始化串口
        LED_Init();                                 //初始化 LED
        KEY_Init();                                 //初始化按键
        LCD_Init();                                 //初始化 LCD FSMC 接口
        MPU_Init();                                 //初始化 MPU6050
        POINT_COLOR=RED;                            //设置字体为红色

        LCD_ShowString(30,50,200,16,16," ELITE STM32");
        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,"2019/9/19");
```

```
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)usart1_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，usart1_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。

35.4 下载验证

本例程测试需要自备 ATK-MPU6050 模块一个。在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32F1 开发板上，可以看到 LCD 显示如图 35.4.1 所示的内容（假定 ATK-MPU6050 模块已经接到精英 STM32 开发板的 ATK MODULE 接口）：

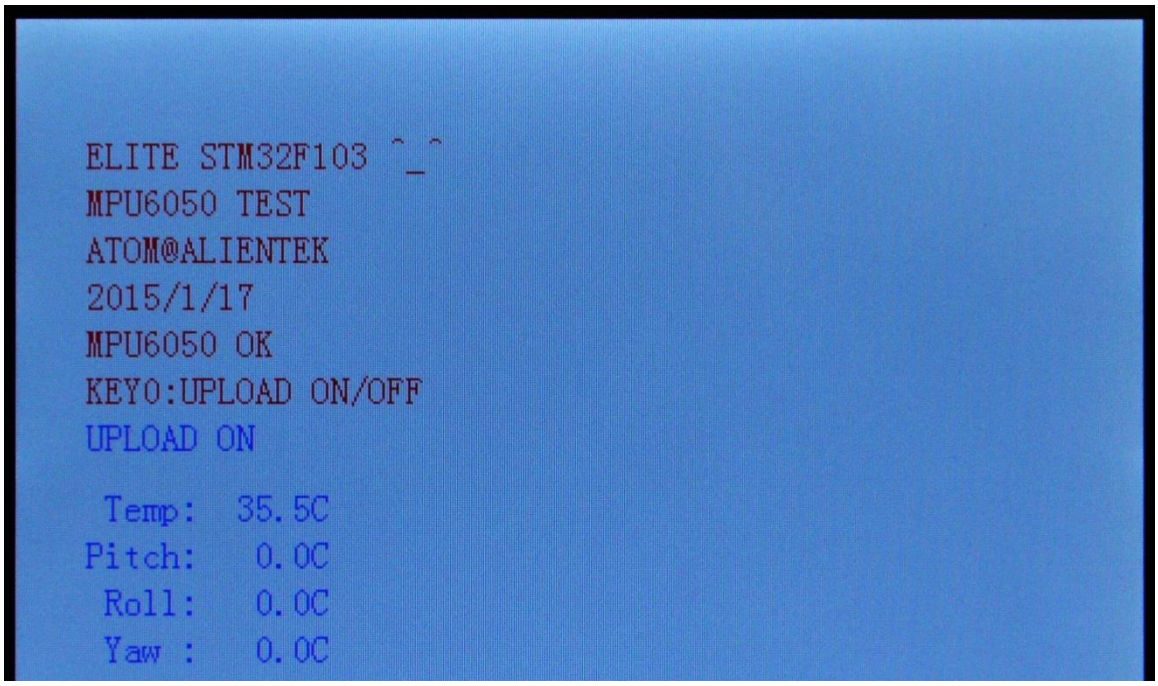


图 35.4.1 程序运行时 LCD 显示内容

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，这个软件，接收 STM32F1 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 35.4.2 和图 35.4.3 所示：



图 35.4.2 传感器数据波形显示

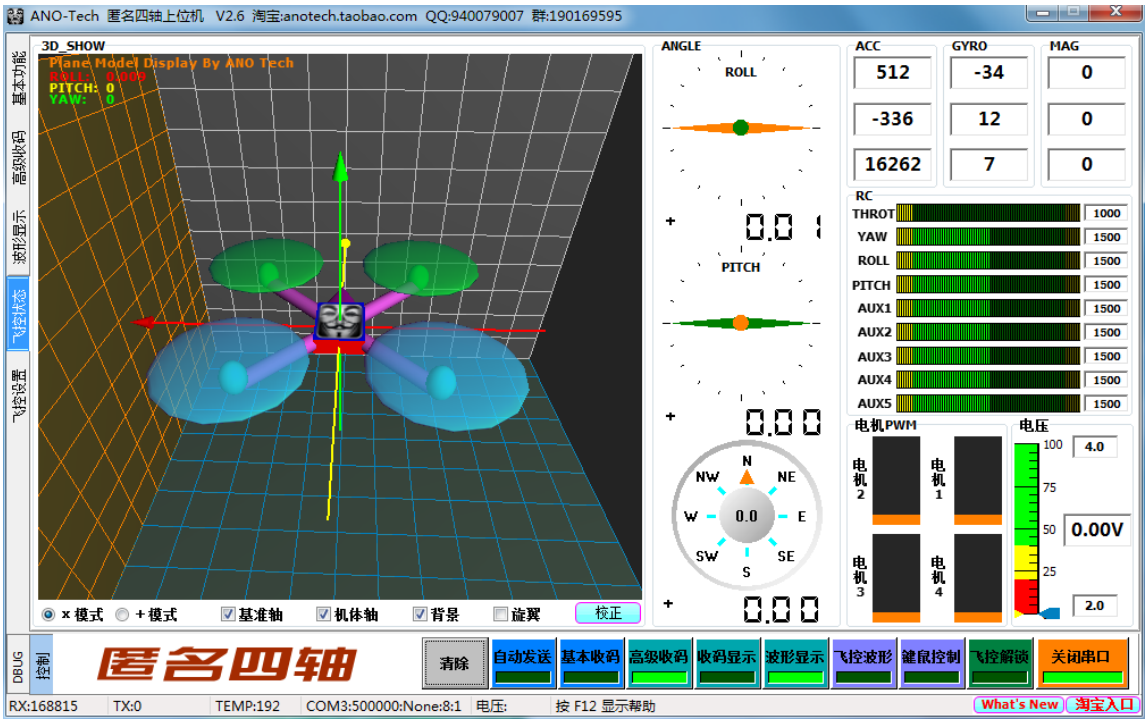


图 35.4.3 飞控状态显示

图 35.4.2 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 35.4.3 则 3D 显示了我们开发板的姿态，通过 usart1_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）。

第三十六章　无线通信实验

ALIENTKE 精英 STM32F1 开发板带有一个无线模块（WIRELESS）接口，采用 8 脚插针方式与开发板连接，可以用来连接 NRF24L01/RFID 等无线模块。本章我们将以 NRF24L01 模块为例向大家介绍如何在 ALIENTEK 精英 STM32F1 开发板上实现无线通信。在本章中，我们将使用两块精英 STM32F1 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：

36.1 NRF24L01 无线模块简介
36.2 硬件设计
36.3 软件设计
36.4 下载验证

36.1 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 36.1.1 所示：

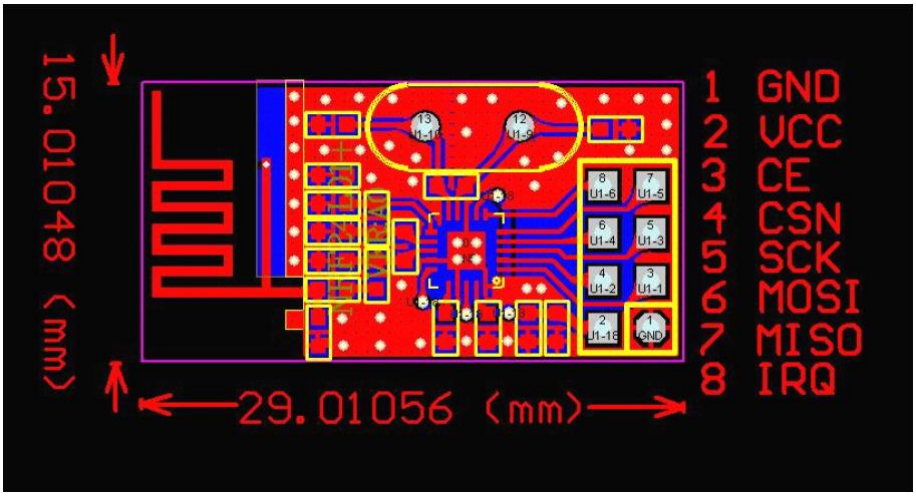


图 36.1.1 NRF24L01 无线模块外观引脚图

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。

36.2 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，通过板载的 WIRELESS 模块接口连接 MCU，这里我们介绍开发板上 WIRELESS 模块接口和 STM32F103 的连接情况，他们的连接关系如图 36.2.1 所示：

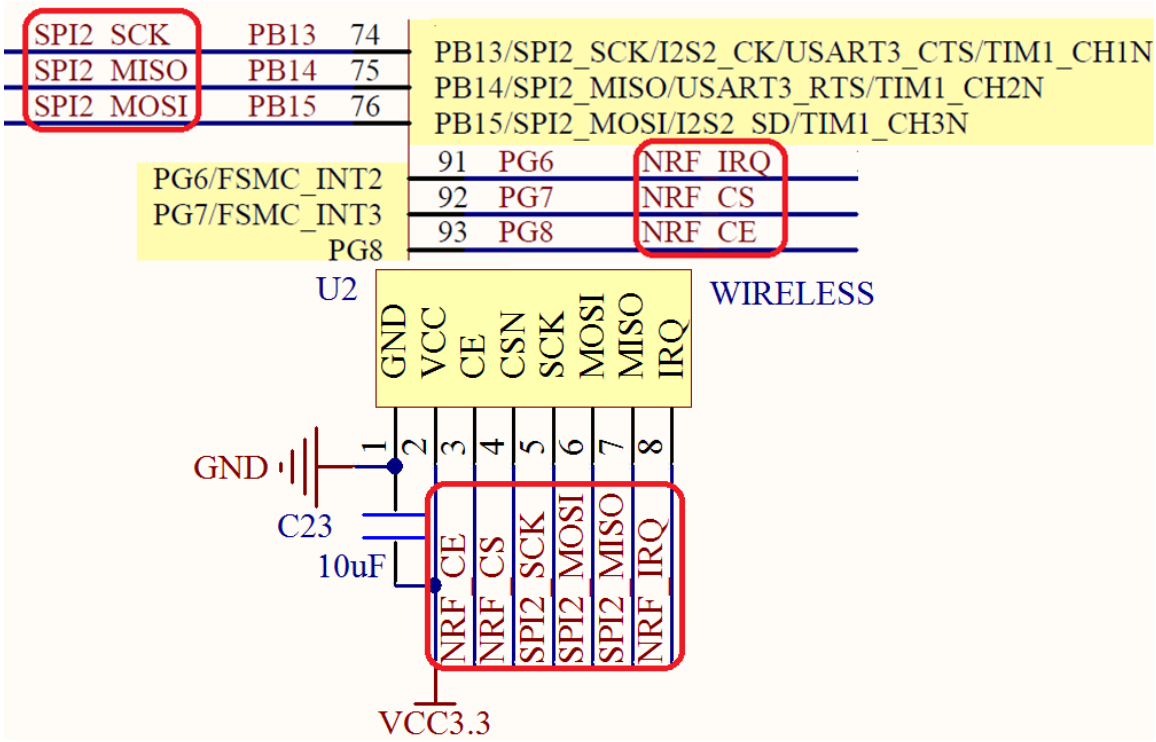


图 36.2.1 NRF24L01 模块接口与 STM32F1 连接原理图

这里WIRELESS模块接口，也是使用的SPI2，和W25Q128共用一个SPI接口，所以在使用的时候，他们分时复用SPI2。本章我们需要把W25Q128的片选信号置高，以防止这个器件对NRF24L01的通信造成干扰。

NRF24L01无线模块和开发板的连接实物图，如图36.2.2所示：

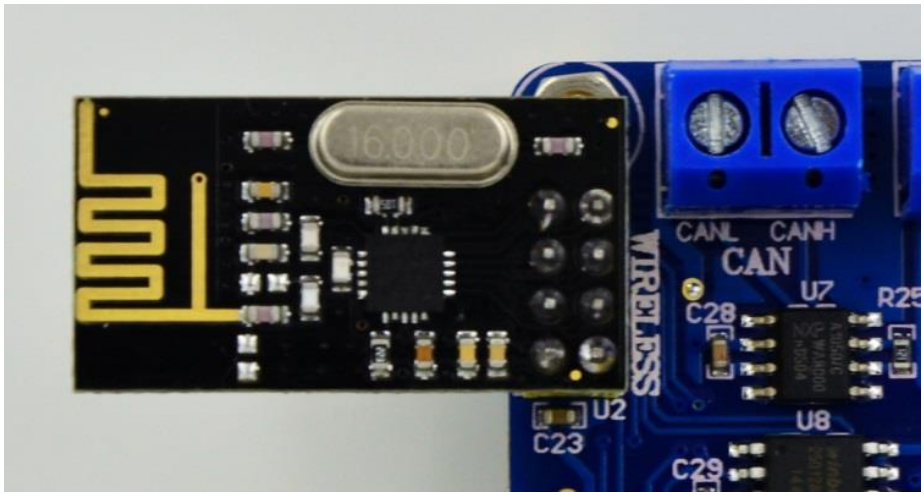


图36.2.2 NRF24L01模块和开发板连接实物图

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精英STM32F1开发板来向大家演示。

36.3 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
//针对 NRF24L01 修改 SPI1 驱动
void NRF24L01_SPI_Init(void)
{
    __HAL_SPI_DISABLE(&SPI2_Handler);                    //先关闭 SPI2
    SPI2_Handler.Init.CLKPolarity=SPI_POLARITY_LOW;
                                                         //串行同步时钟的空闲状态为低电平
    SPI2_Handler.Init.CLKPhase=SPI_PHASE_1EDGE;
                                //串行同步时钟的第 1 个跳变沿（上升或下降）数据被采样
    HAL_SPI_Init(&SPI2_Handler);
    __HAL_SPI_ENABLE(&SPI2_Handler);                     //使能 SPI2
}

//初始化 24L01 的 IO 口
void NRF24L01_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();              //开启 GPIOB 时钟
    __HAL_RCC_GPIOG_CLK_ENABLE();              //开启 GPIOG 时钟

    //GPIOB12 初始化设置:推挽输出
    GPIO_Initure.Pin=GPIO_PIN_12;              //PB12
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;             //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;   //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);        //初始化

    //GPIOG7,8 推挽输出
    GPIO_Initure.Pin=GPIO_PIN_7|GPIO_PIN_8;   //PG7，8
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);        //初始化

    //GPIOG6 上拉输入
    GPIO_Initure.Pin=GPIO_PIN_6;               //PG6
    GPIO_Initure.Mode=GPIO_MODE_INPUT;     //输入
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);        //初始化

    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_12,GPIO_PIN_SET);
                                //PB12 输出 1,防止 SPI FLASH 干扰 NRF 的通信
    SPI2_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;
    SPI2_SetSpeed(SPI_BAUDRATEPRESCALER_8);
    //spi 速度为 10.5Mhz（（24L01 的最大 SPI 时钟为 10Mhz,这里大一点没关系）
    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 =SPI2_ReadWriteByte(reg);  //发送寄存器号
    SPI2_ReadWriteByte(value);      //写入寄存器的值
    NRF24L01_CSN=1;                 //禁止 SPI 传输
    return(status);                 //返回状态值
}
//读取 SPI 寄存器值
//reg:要读的寄存器
u8 NRF24L01_Read_Reg(u8 reg)
{
      u8 reg_val;
    NRF24L01_CSN=0;                 //使能 SPI 传输
    SPI2_ReadWriteByte(reg);        //发送寄存器号
    reg_val=SPI2_ReadWriteByte(0XFF);//读取寄存器内容
    NRF24L01_CSN=1;                 //禁止 SPI 传输
    return(reg_val);                //返回状态值
}
//在指定位置读出指定长度的数据
//reg:寄存器(位置)
//*pBuf:数据指针
//len:数据长度
```

```
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Read_Buf(u8 reg,u8 *pBuf,u8 len)
{
    u8 status,u8_ctr;
    NRF24L01_CSN=0;                 //使能 SPI 传输
    status=SPI2_ReadWriteByte(reg);  //发送寄存器值(位置),并读取状态值
    for(u8_ctr=0;u8_ctr<len;u8_ctr++)pBuf[u8_ctr]=SPI2_ReadWriteByte(0XFF);//读出数据
    NRF24L01_CSN=1;                 //关闭 SPI 传输
    return status;                  //返回读到的状态值
}
//在指定位置写指定长度的数据
//reg:寄存器(位置)
//*pBuf:数据指针
//len:数据长度
//返回值,此次读到的状态寄存器值
u8 NRF24L01_Write_Buf(u8 reg, u8 *pBuf, u8 len)
{
    u8 status,u8_ctr;
    NRF24L01_CSN=0;                 //使能 SPI 传输
    status = SPI2_ReadWriteByte(reg); //发送寄存器值(位置),并读取状态值
    for(u8_ctr=0; u8_ctr<len; u8_ctr++)SPI2_ReadWriteByte(*pBuf++); //写入数据
    NRF24L01_CSN=1;                 //关闭 SPI 传输
    return status;                  //返回读到的状态值
}
//启动 NRF24L01 发送一次数据
//txbuf:待发送数据首地址
//返回值:发送完成状况
u8 NRF24L01_TxPacket(u8 *txbuf)
{
    u8 sta;
    SPI2_SetSpeed(SPI_BAUDRATEPRESCALER_8);
        //spi 速度为 6.75Mhz（24L01 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;
    SPI2_SetSpeed(SPI_BAUDRATEPRESCALER_8);
                                //spi 速度为 6.75Mhz（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 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 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,0x0e);
      //配置基本工作模式的参数;PWR_UP,EN_CRC,16BIT_CRC,接收模式,开启所有中断
    NRF24L01_CE=1;//CE 为高,10us 后启动发送
}
```

此部分代码我们不多介绍，在这里强调一个要注意的地方，在 NRF24L01_Init 函数里面，我们调用了 SPI2_Init()函数，该函数我们在第 29 章曾有提到，在第 29 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：

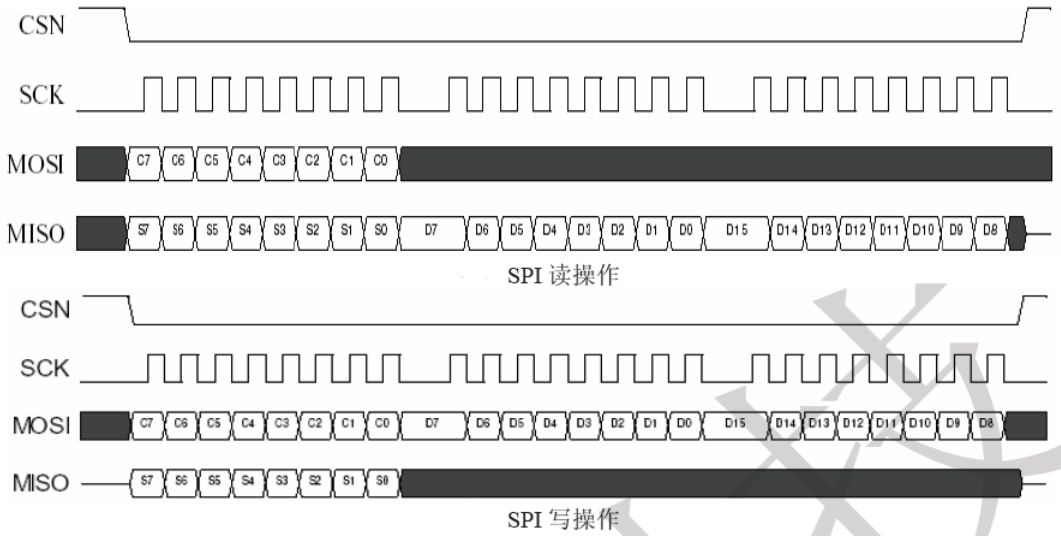


图 38.3.1 NRF24L01 读写操作时序

上图中 Cn 代表指令位，Sn 代表状态寄存器位，Dn 代表数据位。从图中可以看出，SCK 空闲的时候是低电平的，而数据在 SCK 的上升沿被读写。所以，我们需要设置 SPI 的 CPOL 和 CPHA 均为 0，来满足 NRF24L01 对 SPI 操作的要求。所以，我们在 NRF24L01_Init 函数里面又单独添加了将 CPOL 和 CPHA 设置为 0 的函数 NRF24L01_SPI_Init。这里主要是修改了下面两行代码；

```
SPI1_Handler.Init.CLKPolarity=SPI_POLARITY_LOW; //串行同步时钟的空闲状态为低电平
SPI1_Handler.Init.CLKPhase=SPI_PHASE_1EDGE;
//串行同步时钟的第 1 个跳变沿（上升或下降）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(8)    //24L01 片选信号
#define NRF24L01_CSN    PGout(7)    //SPI 片选信号
#define NRF24L01_IRQ    PGin(6)     //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。特别提醒：两个 NRF24L01 模块，互相通信时，他们对应的发射和接收数据宽度必须一致，否则将无法正常通信！

最后我们看看主函数：

```
int main(void)
{
    u8 key,mode; u16 t=0; u8 tmp_buf[33];
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    LCD_Init();                         //初始化 LCD FSMC 接口
    NRF24L01_Init();                    //初始化 NRF24L01
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/19");
    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");//提示
        if(t==200){ LCD_Fill(10,150,230,150+16,WHITE); t=0;}//清除显示
        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){ t=0; LED0=!LED0; }//大约 1s 钟改变一次状态
    };
}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。

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 精英 STM32F1 开发板上，可以看到 LCD 显示如图 36.4.1 所示的内容（假定 NRF24L01 模块已经接上开发板）：



图 36.4.1 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 36.4.2 和图 36.4.3 所示：

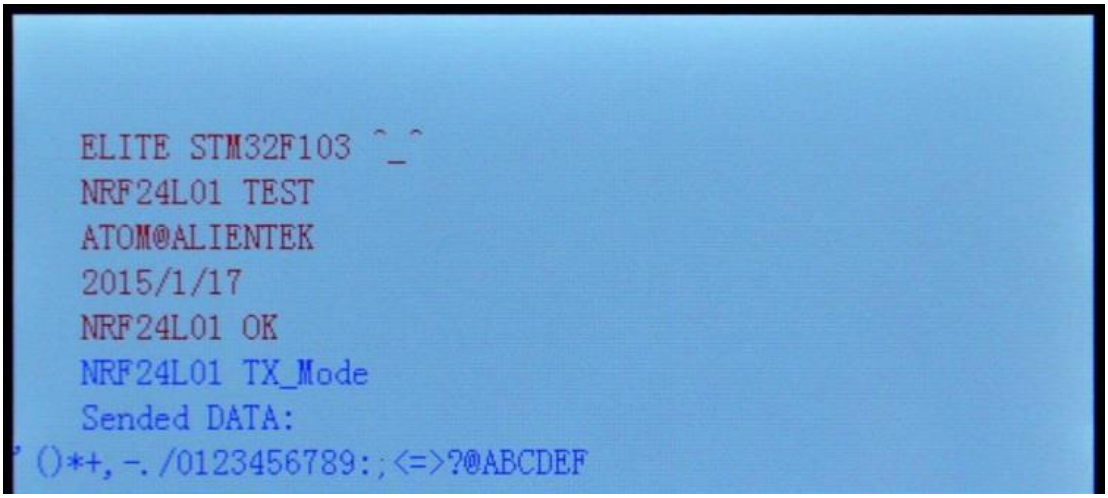


图 36.4.2 开发板 A 发送数据

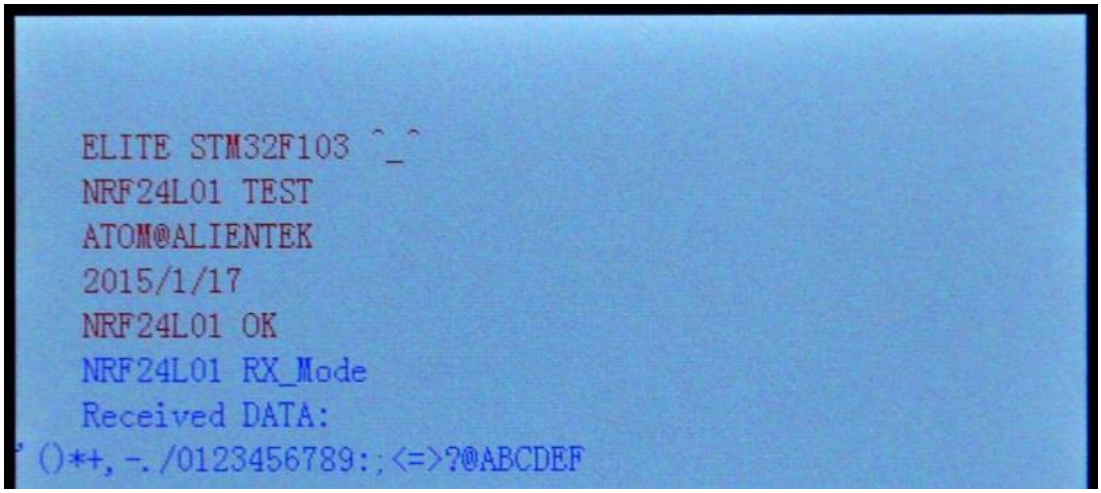


图 36.4.3 开发板 B 接收数据

图 36.4.2 来自开发板 A，工作在发送模式。图 36.4.3 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。

第三十七章 FLASH 模拟 EEPROM 实验

STM32 本身没有自带 EEPROM，但是 STM32 具有 IAP（在应用编程）功能，所以我们可以把它的 FLASH 当成 EEPROM 来使用。本章，我们将利用 STM32 内部的 FLASH 来实现第二十八章类似的效果，不过这次我们是将数据直接存放在 STM32 内部，而不是存放在 W25Q128。本章分为如下几个部分：

37.1 STM32 FLASH 简介
37.2 硬件设计
37.3 软件设计
37.4 下载验证

37.1 STM32 FLASH 简介

不同型号的 STM32，其 FLASH 容量也有所不同，最小的只有 16K 字节，最大的则达到了 1024K 字节。精英 STM32 开发板选择的 STM32F103ZET6 的 FLASH 容量为 512K 字节，属于大容量产品（另外还有中容量和小容量产品），大容量产品的闪存模块组织如图 37.1.1 所示：

块	名称	地址范围	长度(字节)
主存储器	页0	0x0800 0000 – 0x0800 07FF	2K
	页1	0x0800 0800 – 0x0800 0FFF	2K
	页2	0x0800 1000 – 0x0801 17FF	2K
	页3	0x0800 1800 – 0x0801 FFFF	2K
	. . .	. . .	. . .
	页255	0x0807 F800 – 0x0807 FFFF	2K
信息块	启动程序代码	0x1FFF F000 – 0x1FFF F7FF	2K
	用户选择字节	0x1FFF F800 – 0x1FFF F80F	16
闪存存储器接口寄存器	FLASH_ACR	0x4002 2000 – 0x4002 2003	4
	FLASH_KEYR	0x4002 2004 – 0x4002 2007	4
	FLASH_OPTKEYR	0x4002 2008 – 0x4002 200B	4
	FLASH_SR	0x4002 200C – 0x4002 200F	4
	FLASH_CR	0x4002 2010 – 0x4002 2013	4
	FLASH_AR	0x4002 2014 – 0x4002 2017	4
	保留	0x4002 2018 – 0x4002 201B	4
	FLASH_OBR	0x4002 201C – 0x4002 201F	4
	FLASH_WRPR	0x4002 2020 – 0x4002 2023	4

图 37.1.1 大容量产品闪存模块组织

STM32 的闪存模块由：主存储器、信息块和闪存存储器接口寄存器等 3 部分组成。

主存储器，该部分用来存放代码和数据常数（如 const 类型的数据）。对于大容量产品，其被划分为 256 页，每页 2K 字节。注意，小容量和中容量产品则每页只有 1K 字节。从上图可以看出主存储器的起始地址就是 0X08000000， B0、B1 都接 GND 的时候，就是从 0X08000000 开始运行代码的。

信息块，该部分分为 2 个小部分，其中启动程序代码，是用来存储 ST 自带的启动程序，用于串口下载代码，当 B0 接 V3.3，B1 接 GND 的时候，运行的就是这部分代码。用户选择字节，则一般用于配置写保护、读保护等功能，本章不作介绍。

闪存存储器接口寄存器，该部分用于控制闪存读写等，是整个闪存模块的控制机构。

对主存储器和信息块的写入由内嵌的闪存编程/擦除控制器(FPEC)管理；编程与擦除的高电压由内部产生。

在执行闪存写操作时，任何对闪存的读操作都会锁住总线，在写操作完成后读操作才能正确地进行；既在进行写或擦除操作时，不能进行代码或数据的读取操作。

闪存的读取

内置闪存模块可以在通用地址空间直接寻址，任何 32 位数据的读操作都能访问闪存模块的内容并得到相应的数据。读接口在闪存端包含一个读控制器，还包含一个 AHB 接口与 CPU 衔接。这个接口的主要工作是产生读闪存的控制信号并预取 CPU 要求的指令块，预取指令块仅用于在 I-Code 总线上的取指操作，数据常量是通过 D-Code 总线访问的。这两条总线的访问目标是相同的闪存模块，访问 D-Code 将比预取指令优先级高。

这里要特别留意一个闪存等待时间，因为 CPU 运行速度比 FLASH 快得多，STM32F103

的 FLASH 最快访问速度≤24Mhz，如果 CPU 频率超过这个速度，那么必须加入等待时间，比如我们一般使用 72Mhz 的主频，那么 FLASH 等待周期就必须设置为 2，该设置通过 FLASH_ACR 寄存器设置。

例如，我们要从地址 addr，读取一个半字（半字为 16 为，字为 32 位），可以通过如下的语句读取：

```
data=*(vu16*)addr;
```

将 addr 强制转换为 vu16 指针，然后取该指针所指向的地址的值，即得到了 addr 地址的值。类似的，将上面的 vu16 改为 vu8，即可读取指定地址的一个字节。相对 FLASH 读取来说，STM32 FLASH 的写就复杂一点了，下面我们介绍 STM32 闪存的编程和擦除。

闪存的编程和擦除

STM32 的闪存编程是由 FPEC（闪存编程和擦除控制器）模块处理的，这个模块包含 7 个 32 位寄存器，他们分别是：

- FPEC 键寄存器(FLASH_KEYR)
- 选择字节键寄存器(FLASH_OPTKEYR)
- 闪存控制寄存器(FLASH_CR)
- 闪存状态寄存器(FLASH_SR)
- 闪存地址寄存器(FLASH_AR)
- 选择字节寄存器(FLASH_OBR)
- 写保护寄存器(FLASH_WRPR)

其中 FPEC 键寄存器总共有 3 个键值：

RDPRT 键=0X000000A5

KEY1=0X45670123

KEY2=0XCDEF89AB

STM32 复位后，FPEC 模块是被保护的，不能写入 FLASH_CR 寄存器；通过写入特定的序列到 FLASH_KEYR 寄存器可以打开 FPEC 模块（即写入 KEY1 和 KEY2），只有在写保护被解除后，我们才能操作相关寄存器。

STM32 闪存的编程每次必须写入 16 位（不能单纯的写入 8 位数据哦！），当 FLASH_CR 寄存器的 PG 位为’1’时，在一个闪存地址写入一个半字将启动一次编程；写入任何非半字的数据，FPEC 都会产生总线错误。在编程过程中(BSY 位为’1’)，任何读写闪存的操作都会使 CPU 暂停，直到此次闪存编程结束。

同样，STM32 的 FLASH 在编程的时候，也必须要求其写入地址的 FLASH 是被擦除了的（也就是其值必须是 0XFFFF），否则无法写入，在 FLASH_SR 寄存器的 PGERR 位将得到一个警告。

STM23 的 FLASH 编程过程如图 37.1.2 所示：

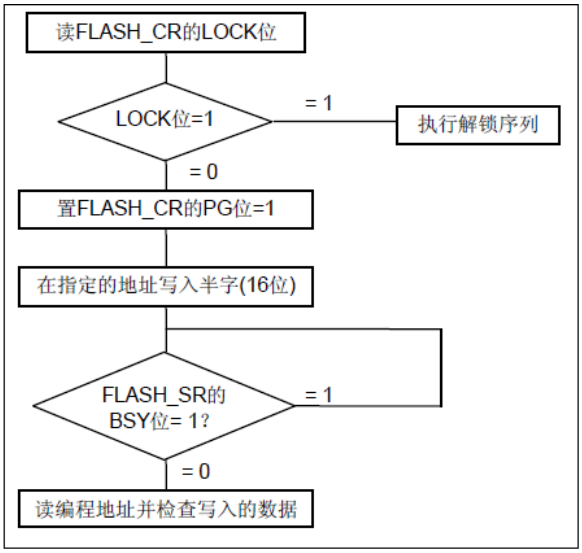


图 37.1.2 STM32 闪存编程过程

从上图可以得到闪存的编程顺序如下：

- 检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁
- 检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的编程操作
- 设置 FLASH_CR 寄存器的 PG 位为’1’
- 在指定的地址写入要编程的半字
- 等待 BSY 位变为’0’
- 读出写入的地址并验证数据

前面提到，我们在 STM32 的 FLASH 编程的时候，要先判断缩写地址是否被擦除了，所以，我们有必要再介绍一下 STM32 的闪存擦除，STM32 的闪存擦除分为两种：页擦除和整片擦除。页擦除过程如图 37.1.3 所示

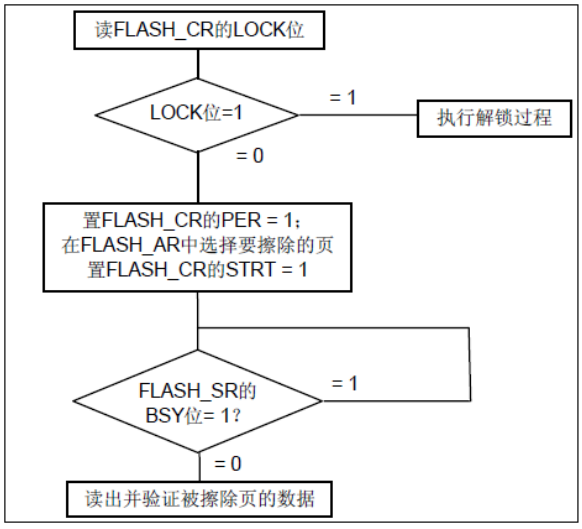


图 37.1.3 STM32 闪存页擦除过程

从上图可以看出，STM32 的页擦除顺序为：

- 检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁
- 检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的闪存操作
- 设置 FLASH_CR 寄存器的 PER 位为’1’

- 用 FLASH_AR 寄存器选择要擦除的页
- 设置 FLASH_CR 寄存器的 STRT 位为’1’
- 等待 BSY 位变为’0’
- 读出被擦除的页并做验证

本章，我们只用到了 STM32 的页擦除功能，整片擦除功能我们在这里就不介绍了。通过以上了解，我们基本上知道了 STM32 闪存的读写所要执行的步骤了，接下来，我们看看与读写相关的寄存器说明。

第一个介绍的是 FPEC 键寄存器：FLASH_KEYR。该寄存器各位描述如图 37.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FKEYR[31:16]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FKEYR[15:0]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

注：　所有这些位是只写的，读出时返回0。

位31~0	**FKEYR**：FPEC键 这些位用于输入FPEC的解锁键。

图 37.1.4 寄存器 FLASH_KEYR 各位描述

该寄存器主要用来解锁 FPEC，必须在该寄存器写入特定的序列（KEY1 和 KEY2）解锁后，才能对 FLASH_CR 寄存器进行写操作。

第二个要介绍的是闪存控制寄存器：FLASH_CR。该寄存器的各位描述如图 37.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
res															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			EOPIE	保留	ERRIE	OPTWRE	保留	LOCK	STRT	OPTER	OPTPG	保留	MER	PER	PG
res			rw	res	rw	rw	res	rw	rw	rw	rw	res	rw	rw	rw

图 37.1.5 寄存器 FLASH_CR 各位描述

该寄存器我们本章只用到了它的 LOCK、STRT、PER 和 PG 等 4 个位。

LOCK 位，该位用于指示 FLASH_CR 寄存器是否被锁住，该位在检测到正确的解锁序列后，硬件将其清零。在一次不成功的解锁操作后，在下次系统复位之前，该位将不再改变。

STRT 位，该位用于开始一次擦除操作。在该位写入 1 ，将执行一次擦除操作。

PER 位，该位用于选择页擦除操作，在页擦除的时候，需要将该位置 1。

PG 位，该位用于选择编程操作，在往 FLASH 写数据的时候，该位需要置 1。

FLASH_CR 的其他位，我们就不在这里介绍了，请大家参考《STM32F10xxx 闪存编程参考手册》第 18 页。

第三个要介绍的是闪存状态寄存器：FLASH_SR。该寄存器各位描述如图 37.1.6 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留										EOP	WRPRT ERR	保留	PGERR	保留	BSY
res										rw	rw	res	rw	res	r

位31~6	保留。必须保持为清除状态'0'
位5	**EOP**：操作结束 当闪存操作(编程/擦除)完成时，硬件设置这位为'1'，写入'1'可以清除这位状态。 *注：每次成功的编程或擦除都会设置EOP状态。*
位4	**WRPRTERR**：写保护错误 试图对写保护的闪存地址编程时，硬件设置这位为'1'，写入'1'可以清除这位状态。
位3	保留。必须保持为清除状态'0'
位2	**PGERR**：编程错误 试图对内容不是'0xFFFF'的地址编程时，硬件设置这位为'1'，写入'1'可以清除这位状态。 *注：进行编程操作之前，必须先清除FLASH_CR寄存器的STRT位。*
位1	保留。必须保持为清除状态'0'
位0	BSY：忙 该位指示闪存操作正在进行。在闪存操作开始时，该位被设置为'1'；在操作结束或发生错误时该位被清除为'0'。

图 37.1.6 寄存器 FLASH_SR 各位描述

该寄存器主要用来指示当前 FPEC 的操作编程状态。

最后，我们再来看看闪存地址寄存器：FLASH_AR。该寄存器各位描述如图 37.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FAR[31:16]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FAR[15:0]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

这些位由硬件修改为当前/最后使用的地址。在页擦除操作中，软件必须修改这个寄存器以指定要擦除的页。

位31~0	**FAR**：闪存地址 当进行编程时选择要编程的地址，当进行页擦除时选择要擦除的页。 *注意：当FLASH_SR中的BSY位为'1'时，不能写这个寄存器。*

图 37.1.7 寄存器 FLASH_AR 各位描述

该寄存器在本章，我们主要用来设置要擦除的页。

关于 STM32 FLASH 的介绍，我们就介绍到这。更详细的介绍，请参考《STM32F10xxx 闪存编程参考手册》。下面我们讲解使用 STM32F1 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源文件 stm32f1xx_hal_flash.c/stm32f1xx_hal_flash_ex.c 以及头文件 stm32f1xx_hal_flash.h/stm32f1xx_hal_flash_ex.h 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）,HAL 库实现很简单：

```
HAL_StatusTypeDef HAL_FLASH_Unlock(void);//解锁函数
```

同样的道理，在对 FLASH 写操作完成之后，我们要锁定 FLASH，使用的 HAL 库函数是：

```
HAL_StatusTypeDef HAL_FLASH_Lock(void);//锁定函数
```

2）写操作函数

HAL 库提供了一个通用的 FLASH 写操作函数 HAL_FLASH_Program，该函数声明如下：

```
HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t Address,
                                    uint64_t Data);//FLASH 写操作函数
```

该函数有三个入口参数。入口参数 TypeProgram 用来区分要写入的数据类型，取值为：FLASH_TYPEPROGRAM_BYTE（字节：8 位），FLASH_TYPEPROGRAM_HALFWORD（半字：16 位），FLASH_TYPEPROGRAM_WORD（字：32 位）和 FLASH_TYPEPROGRAM_DOUBLEWORD（双字：64 位），用户根据写入数据类型选择即可。第二个入口参数 Address 用来设置要写入数据的 FLASH 地址。第三个入口参数 Data 顾名思义就是要写入的数据类型，这个参数默认是 64 位的，如果你要写入小于 64 位的数据比如 16 位，程序会进行类型转换。

3）擦除函数

HAL 库提供的擦除函数在 stm32f1xx_hal_flash_ex.c 中定义。和编程函数一样，HAL 提供了一个通用的基于小区擦除的函数 HAL_FLASHEx_Erase，该函数声明如下：

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
                                    uint32_t *SectorError);
```

该函数有 2 个入口参数，这里我们主要看第一个入口参数 pEraseInit，它是 FLASH_EraseInitTypeDef 结构体指针类型，结构体 FLASH_EraseInitTypeDef 定义如下：

```
typedef struct
{
  uint32_t TypeErase;         //擦除类型
  uint32_t Banks;             //擦除的 Bank 编号
  uint32_t PageAddress;       //擦除页面地址
  uint32_t NbPages;           //擦除的页面数
} FLASH_EraseInitTypeDef;
```

成员变量 TypeErase 用来设置擦除类型，是 Page 擦除还是 BANK 级别的批量擦除，取值为 FLASH_TYPEERASE_PAGES 或者 FLASH_TYPEERASE_MASSERASE，这个比较好理解，如果是一次擦除一个 Bank 下面的所有 Page，那么需要选择 FLASH_TYPEERASE_MASSERASE。成员变量 Banks 用来设置要擦除的 Bank 编号，这个只有设置为批量擦除的时候才有效。成员变量 PageAddress 用来设置要擦除页面的地址。成员变量 NbPages 用来设置要擦除的页面数。

4）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。HAL 库函数为：

```
HAL_StatusTypeDef FLASH_WaitForLastOperation(uint32_t Timeout);
```

该函数在 HAL 库中很多地方用到，比如擦除函数 HAL_FLASHEx_Erase 中在对 FLASH 进行擦除操作后会调用该函数，等待擦除操作完成。

5）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半个字节的函数：

```
//读取指定地址的半字(16 位数据)
//faddr:读地址
//返回值:对应数据.
u16 STMFLASH_ReadHalfWord(u32 faddr)
{
    return *(vu16*)faddr;
}
```

37.2 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（KEY_UP）用来执行写入 FLASH 的操作，另外一个按键（KEY1）用来执行读出操作，在 TFTLCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0

2） KEY_UP 和 KEY1 按键

3） TFTLCD 模块

4） STM32 内部 FLASH

本章需要用到的资源和电路连接，在之前已经全部有介绍过了，接下来我们直接开始软件设计。

37.3 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 stmflash.h。同时我们还引入了 HAL 库 flash 操作文件 stm32f1xx_hal_flash.c 和头文件 stm32f1xx_hal_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.
u16 STMFLASH_ReadHalfWord(u32 faddr)
{
    return *(vu16*)faddr;
}
#if STM32_FLASH_WREN //如果使能了写
//不检查的写入
//WriteAddr:起始地址
//pBuffer:数据指针
//NumToWrite:半字(16 位)数
void STMFLASH_Write_NoCheck(u32 WriteAddr,u16 *pBuffer,u16 NumToWrite)
{
    u16 i;
    for(i=0;i<NumToWrite;i++)
```

```
        {
        HAL_FLASH_Program(FLASH_TYPEPROGRAM_HALFWORD,WriteAddr,pBuffer[i]);
            WriteAddr+=2;//地址增加 2.
        }
    }
    //从指定地址开始写入指定长度的数据
    //WriteAddr:起始地址(此地址必须为 2 的倍数!!)
    //pBuffer:数据指针
    //NumToWrite:半字(16 位)数(就是要写入的 16 位数据的个数.)
    #if STM32_FLASH_SIZE<256
    #define STM_SECTOR_SIZE 1024 //字节
    #else
    #define STM_SECTOR_SIZE      2048
    #endif
    u16 STMFLASH_BUF[STM_SECTOR_SIZE/2];//最多是 2K 字节
    void STMFLASH_Write(u32 WriteAddr,u16 *pBuffer,u16 NumToWrite)
    {
        u32 secpos;         //扇区地址
        u16 secoff;         //扇区内偏移地址(16 位字计算)
        u16 secremain;      //扇区内剩余地址(16 位字计算)
        u16 i;
        u32 offaddr;        //去掉 0X08000000 后的地址
        if(WriteAddr<STM32_FLASH_BASE||(WriteAddr>=(STM32_FLASH_BASE+
                                            1024*STM32_FLASH_SIZE)))return;//非法地址

        HAL_FLASH_Unlock();                             //解锁
        offaddr=WriteAddr-STM32_FLASH_BASE;     //实际偏移地址.
        secpos=offaddr/STM_SECTOR_SIZE;         //扇区地址   0~127 for STM32F103RBT6
        secoff=(offaddr%STM_SECTOR_SIZE)/2;  //在扇区内的偏移(2 个字节为基本单位.)
        secremain=STM_SECTOR_SIZE/2-secoff;  //扇区剩余空间大小
        if(NumToWrite<=secremain)secremain=NumToWrite;//不大于该扇区范围
        while(1)
        {
            STMFLASH_Read(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE,
                          STMFLASH_BUF,STM_SECTOR_SIZE/2);//读出整个扇区的内容
            for(i=0;i<secremain;i++)      //校验数据
            {
                if(STMFLASH_BUF[secoff+i]!=0XFFFF)break;//需要擦除
            }
            if(i<secremain)                         //需要擦除
            {
                FLASH_PageErase(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE);
                                                                //擦除这个扇区
```

```
                FLASH_WaitForLastOperation(FLASH_WAITETIME); //等待上次操作完成
                CLEAR_BIT(FLASH->CR, FLASH_CR_PER);
                //清除 CR 寄存器的 PER 位，此操作应该在 FLASH_PageErase()中完成！
                //但是 HAL 库里面并没有做，应该是 HAL 库 bug！
                for(i=0;i<secremain;i++)//复制
                {
                    STMFLASH_BUF[i+secoff]=pBuffer[i];
                }
            STMFLASH_Write_NoCheck(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE,
             STMFLASH_BUF,STM_SECTOR_SIZE/2);//写入整个扇区
             }else
             {
                FLASH_WaitForLastOperation(FLASH_WAITETIME); //等待上次操作完成
                STMFLASH_Write_NoCheck(WriteAddr,pBuffer,secremain);
                                                //写已经擦除了的,直接写入扇区剩余区间.
             }
             if(NumToWrite==secremain)break;//写入结束了
             else//写入未结束
             {
                secpos++;                   //扇区地址增 1
                secoff=0;                   //偏移位置为 0
                pBuffer+=secremain;         //指针偏移
                WriteAddr+=secremain*2;     //写地址偏移(16 位数据地址,需要*2)
                NumToWrite-=secremain;      //字节(16 位)数递减
                if(NumToWrite>(STM_SECTOR_SIZE/2))secremain=STM_SECTOR_SIZE/2;
                                            //下一个扇区还是写不完
                else secremain=NumToWrite;//下一个扇区可以写完了
             }
        };
        HAL_FLASH_Lock();           //上锁
}
#endif

//从指定地址开始读出指定长度的数据
//ReadAddr:起始地址
//pBuffer:数据指针
//NumToWrite:半字(16 位)数
void STMFLASH_Read(u32 ReadAddr,u16 *pBuffer,u16 NumToRead)
{
    u16 i;
    for(i=0;i<NumToRead;i++)
    {
        pBuffer[i]=STMFLASH_ReadHalfWord(ReadAddr);//读取 2 个字节.
```

```
            ReadAddr+=2;//偏移 2 个字节.
        }
    }

    ////////////////////////////////////////////测试用////////////////////////////////////////
    //WriteAddr:起始地址
    //WriteData:要写入的数据
    void Test_Write(u32 WriteAddr,u16 WriteData)
    {
        STMFLASH_Write(WriteAddr,&WriteData,1);//写入一个字
    }
```

该部分代码，我们重点介绍一下 STMFLASH_Write 函数，该函数用于在 STM32 的指定地址写入指定长度的数据，该函数的实现基本类似第 29 章的 SPI_Flash_Write 函数，不过该函数对写入地址是有要求的，必须保证以下两点：

1， 该地址必须是用户代码区以外的地址。

2， 该地址必须是 2 的倍数。

条件 1 比较好理解，如果把用户代码给卡擦了，可想而知你运行的程序可能就被废了，从而很可能出现死机的情况。条件 2 则是 STM32 FLASH 的要求，每次必须写入 16 位，如果你写的地址不是 2 的倍数，那么写入的数据，可能就不是写在你要写的地址了。

另外，该函数的 STMFLASH_BUF 数组，也是根据所用 STM32 的 FLASH 容量来确定的，战舰 STM32 开发板的 FLASH 是 512K 字节，所以 STM_SECTOR_SIZE 的值为 512，故该数组大小为 2K 字节。其他函数我们就不做介绍了，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[]={"STM32F103 FLASH TEST"};
#define SIZE sizeof(TEXT_Buffer)          //数组长度
#define FLASH_SAVE_ADDR  0X08070000  //设置 FLASH 保存地址(必须为偶数，且
                                     //其值要大于本代码所占用 FLASH 的大小+0X08000000)
int main(void)
{
    u8 key=0;
    u16 i=0;
    u8 datatemp[SIZE];
    HAL_Init();                          //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);      //设置时钟,72M
    delay_init(72);                      //初始化延时函数
    uart_init(115200);                   //初始化串口
    LED_Init();                          //初始化 LED
    KEY_Init();                          //初始化按键
    LCD_Init();                          //初始化 LCD FSMC 接口
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/19");
        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,(u16*)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,(u16*)datatemp,SIZE);
                LCD_ShowString(30,170,200,16,16,"The Data Readed Is:   ");//提示传送完成
                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。

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 精英 STM32 开发板上，通过先按 WK_UP 按键写入数据，然后按 KEY1 读取数据，得到如图 37.4.1 所示：

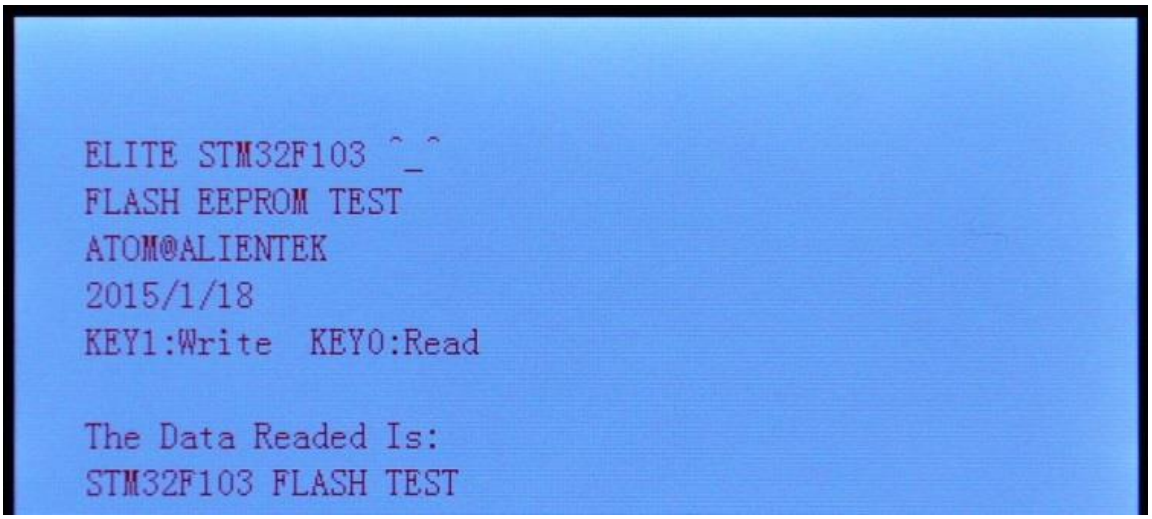


图 37.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。本章的测试，我们还可以借助 USMART，在 USMART 里面添加 STMFLASH_ReadHalfWord 函数，既可以读取任意地址的数据。当然，你也可以将 STMFLASH_Write 稍微改造下，这样就可以在 USMART 里面验证 STM32 FLASH 的读写了。

第三十八章 摄像头实验

ALIENTEK 精英 STM32 开发板板载了一个摄像头接口（P6），该接口可以用来连接 ALIENTEK OV7725 摄像头模块。本章，我们将使用 STM32 驱动 ALIENTEK OV7725 摄像头模块，实现摄像头功能。本章分为如下几个部分：

38.1 OV7725 简介
38.2 硬件设计
38.3 软件设计
38.4 下载验证

38.1 OV7725 简介

OV7725 是 OV（OmniVision）公司生产的一颗 1/4 寸的 CMOS VGA 图像传感器。该传感器体积小、工作电压低（典型电压 3.3V），提供单片 VGA 摄像头和影像处理器的所有功能。通过 SCCB 总线控制，可以输出整帧、子采样、取窗口等方式的各种分辨率，10 位或 8 位影像数据输出。该产品 VGA 图像输出最高可达 60 帧/秒。用户可以完全控制图像质量、数据格式和传输方式。所有图像处理功能过程包括伽玛曲线、白平衡、度、色度等都可以通过 SCCB 接口编程。OmmiVision 图像传感器应用独有的传感器技术，通过减少或消除光学或电子缺陷如固定图案噪声、托尾、浮散等，提高图像质量，得到清晰的稳定的彩色图像。

OV7725 的特点有：

- 高灵敏度、低电压适合嵌入式应用
- 标准的 SCCB 接口，兼容 IIC 接口
- 支持 RawRGB、RGB(GBR4:2:2，RGB565/RGB555/RGB444)，YUV(4:2:2)和 YCbCr（4:2:2）输出格式
- 支持 VGA、QVGA，和从 CIF 到 40*30 的各种尺寸输出
- 支持自动曝光控制、自动增益控制、自动白平衡、自动消除灯光条纹、自动黑电平校准等自动控制功能。同时支持色饱和度、色相、伽马、锐度等设置。
- 支持图像缩放

OV7725 的功能框图如图 38.1.1 所示：

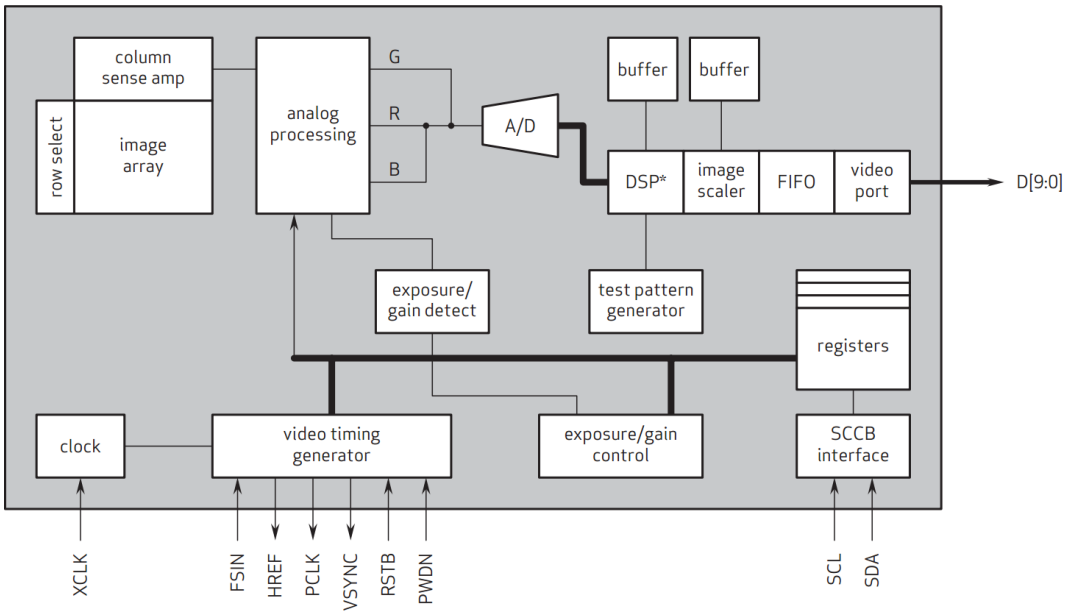


图 38.1.1 OV7725 功能框图

OV7725 传感器包括如下一些功能模块。

1.感光整列（Image Array）

OV7725 总共有 656*488 个像素，其中 640*480 个有效（即有效像素为 30W）。

2.时序发生器（Video Timing Generator）

时序发生器具有的功能包括：整列控制和帧率发生（7 种不同格式输出）、内部信号发生器和分布、帧率时序、自动曝光控制、输出外部时序（VSYNC、HREF/HSYNC 和 PCLK）。

3.模拟信号处理（Analog Processing）

模拟信号处理所有模拟功能，并包括：自动增益（AGC）和自动白平衡（AWB）。

4.A/D 转换（A/D）

原始的信号经过模拟处理器模块之后 ，分 G 和 BR 两路进入一个 10 位的 A/D 转换器，A/D 转换器工作在 12M 频率，与像素频率完全同步（转换的频率和帧率有关）。

除 A/D 转换器外，该模块还有以下三个功能：

- 黑电平校正（BLC)
- U/V 通道延迟
- A/D 范围控制

A/D 范围乘积和 A/D 的范围控制共同设置 A/D 的范围和最大值，允许用户根据应用调整图片的亮度。

5.测试图案发生器（Test Pattern Generator）

测试图案发生器功能包括：八色彩色条图案、渐变至黑白彩色条图案和输出脚移位“1”。

6.数字处理器（DSP）

这个部分控制由原始信号插值到 RGB 信号的过程，并控制一些图像质量：

- 边缘锐化（二维高通滤波器）
- 颜色空间转换（ 原始信号到 RGB 或者 YUV/YCbYCr)
- RGB 色彩矩阵以消除串扰
- 色相和饱和度的控制
- 可编程的伽玛
- 十位到八位数据转换

7.缩放功能（Image Scaler）

这个模块按照预先设置的要求输出数据格式，能将 YUV/RGB 信号从 VGA 缩小到 CIF 以下的任何尺寸。

8.数字视频接口（Digital Video Port）

通过寄存器 COM2[1:0]，调节 IOL/IOH 的驱动电流，以适应用户的负载。

9.SCCB 接口（SCCB Interface）

SCCB 接口控制图像传感器芯片的运行，详细使用方法参照光盘的《OmniVision Technologies Seril Camera Control Bus(SCCB) Specification》这个文档

OV7725 的寄存器通过 SCCB 时序访问并设置，SCCB 时序和 IIC 时序十分类似，在本章我们不做介绍，请大家参考光盘的 SCCB 相关文档。接下来我们介绍一下 OV7725 的图像数据输出格式及时序分析。首先我们简单介绍几个定义：

VGA，即分辨率为 640*480 的输出模式；

QVGA，即分辨率为 320*240 的输出格式；

QQVGA，即分辨率为 160*120 的输出格式；

PCLK，即像素时钟，一个 PCLK 时钟，输出一个像素(或半个像素)。

VSYNC，即帧同步信号。

HREF /HSYNC，即行同步信号。

OV7725 时序分析：

我们的 LCD 数据格式为 RGB565，下面以 OV7725 输出 RGB565 模式分析一下时序。OV7725 输出时序图如下图 38.1.2 所示：

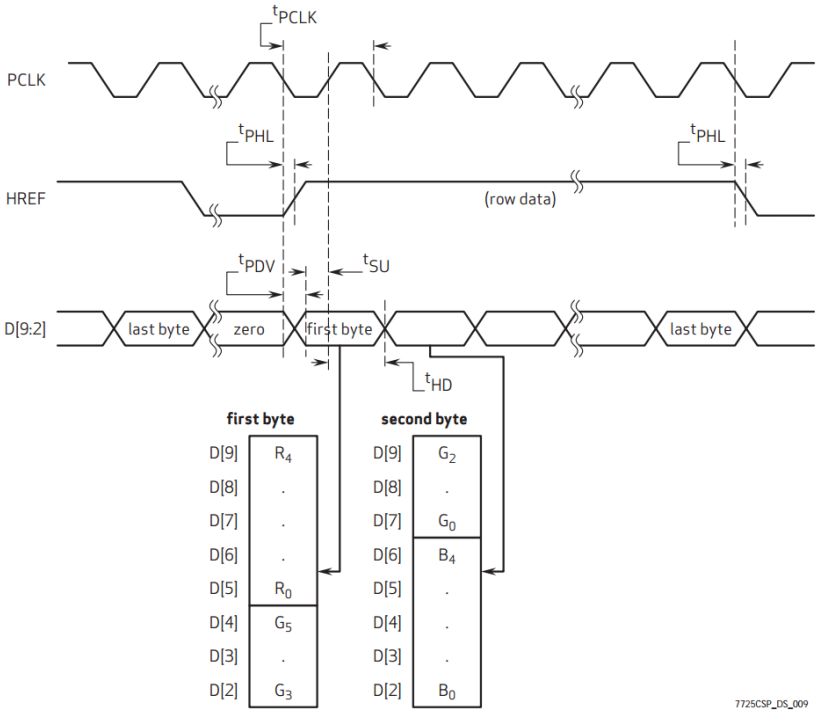


图 38.1.2 OV7725 RGB565 输出时序

从上图可看出，OV7725 的图像数据通过 D[9:2]输出一个字节，first byte 和 second byte 组成一个 16 位 RGB565 数据。时序上，HREF 为高时开始传输一行数据，1 个 PCLK 传输 1 个字节，传输完一行数据最后一个字节（last byte）后 HREF 则变为低。

再来看看 OV7725 帧时序图（VGA 模式），如下图 38.1.3 所示：

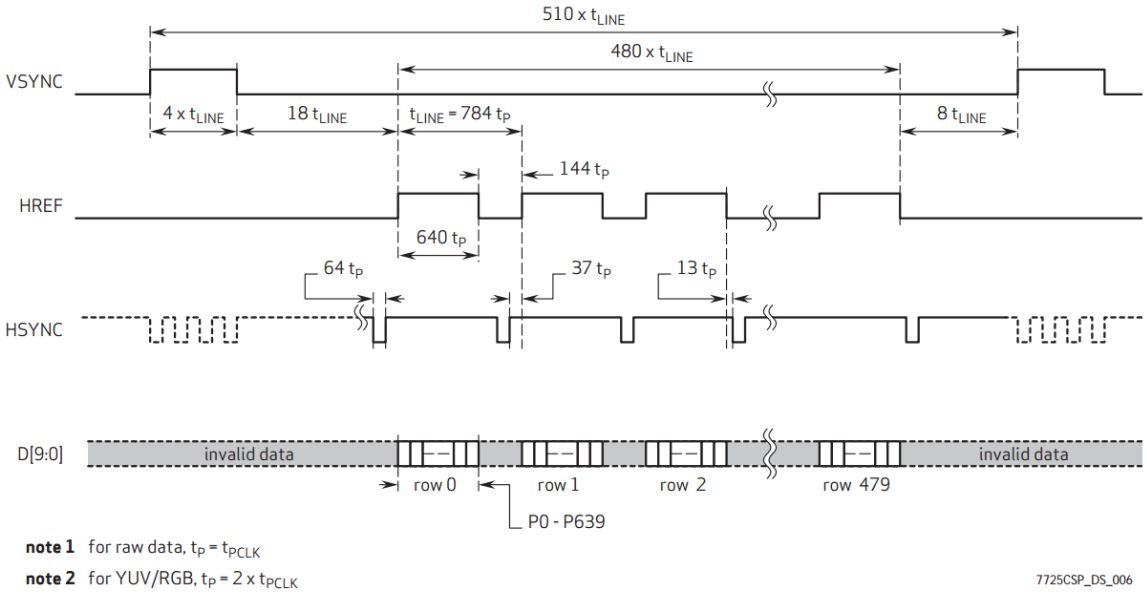


图 38.1.3 OV7725 帧时序

从上图可看出，1 个 HREF 周期由 640t_p 高电平和低电平 144t_p 组成。对于 YUV/RGB 模式，t_P=2×t_{PCLK} 即一个 t_{PCLK} 对应传输一个字节（RGB565 格式传输一行数据的时间 T=640×2t_{PCLK}）。其中 144t_p 是传输一行数据的间隔时间。当传输了 480 个 HREF 周期（480×t_{LINE}）后刚好完成一个 VSYNC（帧）数据传输，等 8 t_{LINE} 后会产生一个 VSYNC 上升沿表示一帧数据传输完成。

程序中我们就可以根据 VSYNC 上升沿来判断一帧图像数据传输完成。注意，图中的 HSYNC 和 HREF 其实是同一个引脚产生的信号，只是在不同场合下面，使用不同的信号方式，我们本章用到的是 HREF。

因为 OV7725 的像素时钟（PCLK）最高可达 24Mhz，我们用 STM32F103ZET6 的 IO 口直接抓取，是非常困难的，也十分占耗 CPU（可以通过降低 PCLK 输出频率，来实现 IO 口抓取，但是不推荐）。所以，本章我们并不是采取直接抓取来自 OV7725 的数据，而是通过 FIFO 读取，ALIENTEK OV7725 摄像头模块自带了一个 FIFO 芯片（AL422B），用于暂存图像数据，有了这个芯片，我们就可以很方便的获取图像数据了，而不再需要单片机具有高速 IO，也不会耗费多少 CPU，可以说，只要是个单片机，都可以通过 ALIENTEK OV7725 摄像头模块实现拍照的功能。

接下来我们介绍一下 ALIENTEK OV7725 摄像头模块。OV7725 模块的外观如图 38.1.4：

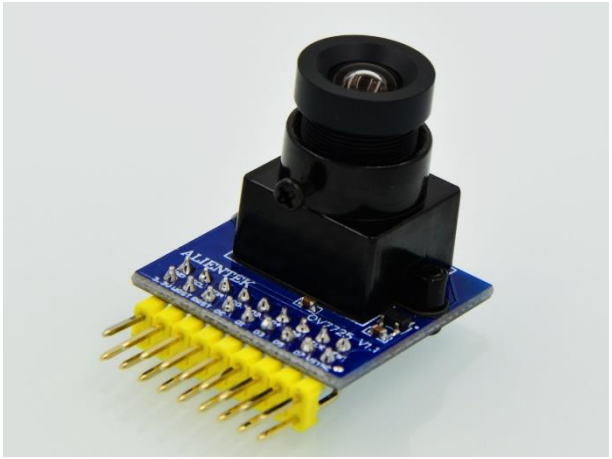


图 38.1.4 ALIENTEK OV7725 摄像头模块外观图

OV7725 模块原理图如下图 38.1.5 所示：

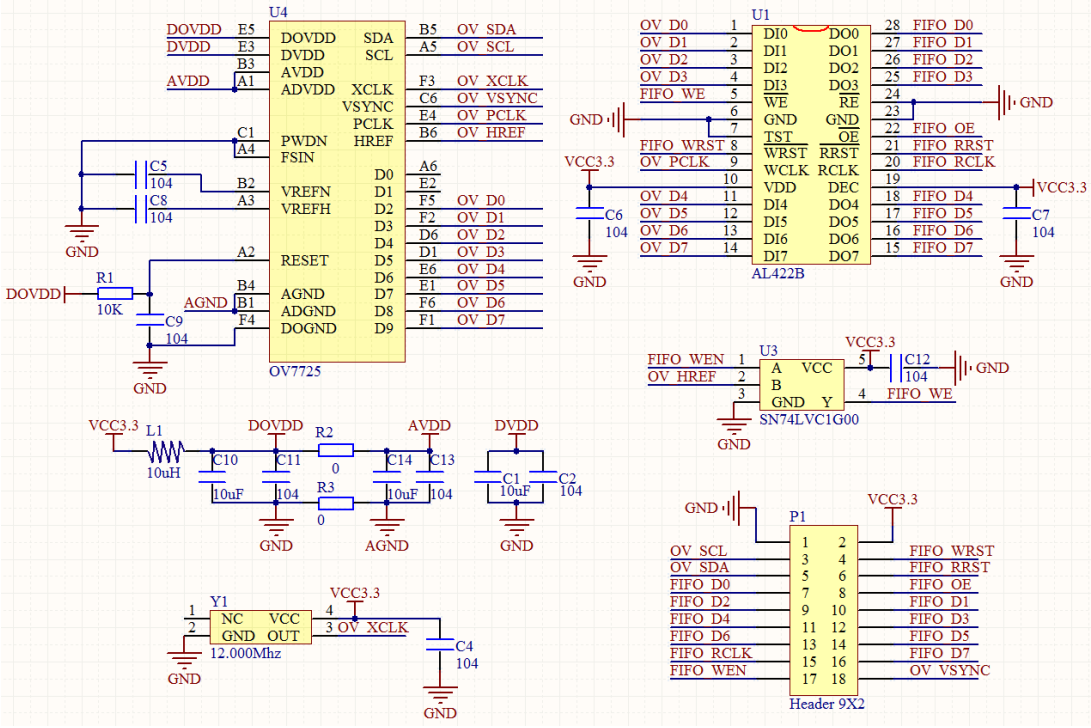


图 38.1.5 ALIENTEK OV7725 摄像头模块原理图

从上图可以看出，ALIENTEK OV7725 摄像头模块自带了有源晶振，用于产生 12M 时钟作为 OV7725 传感器的 XCLK 输入；带有一个 FIFO 芯片（AL422B），该 FIFO 芯片的容量是 384K 字节，足够存储 2 帧 QVGA 的图像数据。模块通过一个 2*9 的双排排针（P1）与外部通信，与外部的通信信号如表 38.1.1 所示：

信号	作用描述	信号	作用描述
VCC3.3	模块供电脚，接 3.3V 电源	FIFO_WEN	FIFO 写使能
GND	模块地线	FIFO_WRST	FIFO 写指针复位
OV_SCL	SCCB 通信时钟信号	FIFO_RRST	FIFO 读指针复位
OV_SDA	SCCB 通信数据信号	FIFO_OE	FIFO 输出使能（片选）
FIFO_D[7:0]	FIFO 输出数据（8 位）	OV_VSYNC	帧同步信号
FIFO_RCLK	读 FIFO 时钟		

表 38.1.1 P1 接口信号描述

下面我们来看看如何使用 ALIENTEK OV7725 摄像头模块（以 QVGA 模式，RGB565 格式为例）。对于该模块，我们只关心两点：1，如何存储图像数据；2，如何读取图像数据。

首先，我们来看如何存储图像数据。

摄像头模块存储图像数据的过程为：等待 OV7725 帧同步信号→FIFO 写指针复位→FIFO 写使能→等待第二个 OV7725 帧同步信号→FIFO 写禁止。通过以上 5 个步骤，我们就可以完成 1 帧图像数据在 AL422B 的存储。注意：FIFO 写禁止操作不是必须的，只有当你想将一帧图片数据存储在 FIFO，并在外部 MCU 读取完这帧图片数据之前，不再采集新的图片数据的时候，才需要进行 FIFO 写禁止。

接下来，我们来看看如何读取图像数据。

在存储完一帧图像以后，我们就可以开始读取图像数据了。读取过程为：FIFO 读指针复位→给 FIFO 读时钟（FIFO_RCLK）→读取第一个像素高字节→给 FIFO 读时钟→读取第一个像素低字节→给 FIFO 读时钟→读取第二个像素高字节→循环读取剩余像素→结束。

可以看出，摄像头模块数据的读取也是十分简单，比如 QVGA 模式，RGB565 格式，我们总共循环读取 320*240*2 次，就可以读取 1 帧图像数据，把这些数据写入 LCD 模块，我们就可以看到摄像头捕捉到的画面了。

了解了数据存储和读取，我们就可以开始设计代码了，本章，我们用一个外部中断，来捕捉帧同步信号（VSYNC），然后在中断里面启动图像数据存储，等待下一次 VSYNC 信号到来，我们就关闭数据存储，然后一帧数据就存储完成了，在主函数里面就可以慢慢的将这一帧数据读出来，放到 LCD 即可显示了，同时开始第二帧数据的存储，如此循环，实现摄像头功能。

本章，我们将使用 ALIENTEK OV7725 摄像头模块的以 QVGA 模式（分辨率 320*240），直接输出显示到我们 LCD 上，或者以 VGA 模式（分辨率 640*320）截取中间 320*240 输出显示。注意：摄像头模块自带的 FIFO（AL422B）是没办法缓存一帧的 VGA 图像的，如果使用 VGA 全屏分辨率输出，那么你必须在 FIFO 写满之前开始读 FIFO 数据，保证数据不被覆盖。OV7725 还可以对输出图像进行各种设置，数据手册和应用笔记详见光盘《OV7725_datasheet.pdf》和《OV7725 Software Application Note.pdf》。对 AL422B 的操作时序，请大家参考 AL422B 的数据手册。

38.2 硬件设计

本章实验功能简介：开机后，检测和初始化摄像头模块。对于 OV7725：初始化成功后需先通过 KEY0 和 KEY1 选择为 QVGA 或 VGA 输出模式，然后 LCD 才会显示拍摄到的画面。

正常显示拍摄画面后我们可以通过 KEY0 设置光照模式、通过 KEY1 设置色饱和度，通过

KEY_UP 设置对比度，通过 TPAD 设置特效（总共 7 种特效）。通过串口，我们可以查看当前的帧率（这里是指 LCD 显示的帧率，而不是指 OV7725 的输出帧率），同时可以借助 USMART 设置 OV7725 的寄存器，方便大家调试。DS0 指示程序运行状态。

本实验用到的硬件资源有：

1） 指示灯 DS0

2） KEY0/KEY1/KEY_UP 和 TPAD 按键

3） 串口

4） TFTLCD 模块

5） OV7725 摄像头模块

摄像头模块接口在 38.1 节已经有详细介绍过，这里我们主要介绍该模块与 ALIETEK 精英 STM32 开发板的连接。

在开发板的左下角的 2*9 的 P6 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 OV7725 或 OV7670 摄像头模块插入这个接口（P6）即可，该接口与 STM32 的连接关系如图 38.3.1 所示：

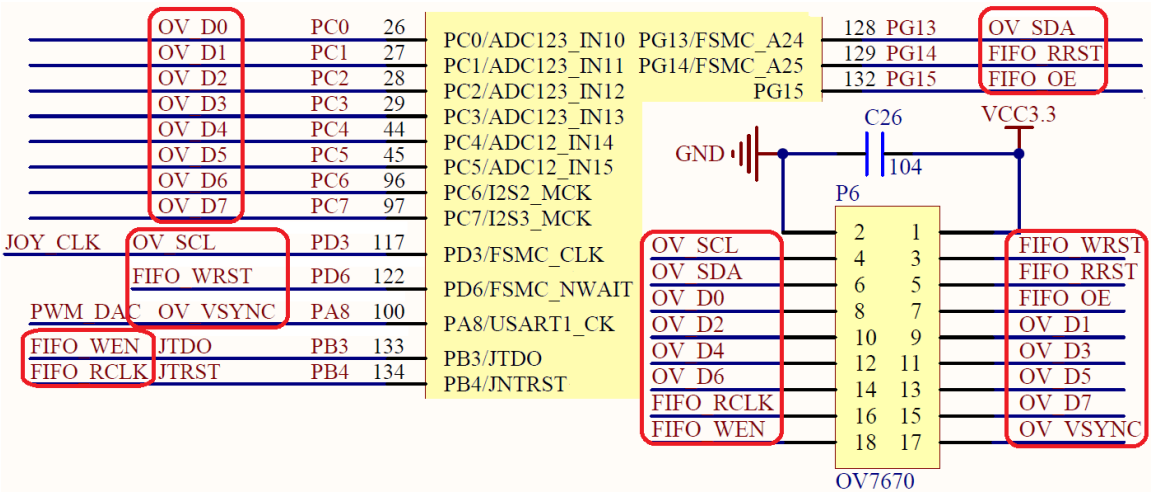


图 38.3.1 摄像头模块接口与 STM32 连接图

从上图可以看出，OV7725 摄像头模块的各信号脚与 STM32 的连接关系为：

OV_SDA 接 PG13；

OV_SCL 接 PD3；

FIFO_RCLK 接 PB4；

FIFO_WEN 接 PB3；

FIFO_WRST 接 PD6；

FIFO_RRST 接 PG14；

FIFO_OE 接 PG15；

OV_VSYNC 接 PA8；

OV_D[7:0]接 PC[7:0]；

这些线的连接，精英 STM32 的内部已经连接好了，我们只需要将摄像头模块插上去就好了。注意，其中有几个信号线和其他外设共用了，OV_SCL 与 JOY_CLK 共用 PD3，所以摄像头和手柄不可以同时使用；FIFO_WEN 和 FIFO_RCLK 则和 JTAG 的信号线 JTDO 和 JTRST 共用了，所以使用摄像头的时候，不能使用 JTAG 模式调试，而应该选择 SW 模式（SW 模式不需要用到 JTDO 和 JTRST）；OV_VSYNC 和 PWM_DAC 共用了 PA8，所以他们也不可以同时使用。

实物连接如图 38.3.2 所示：

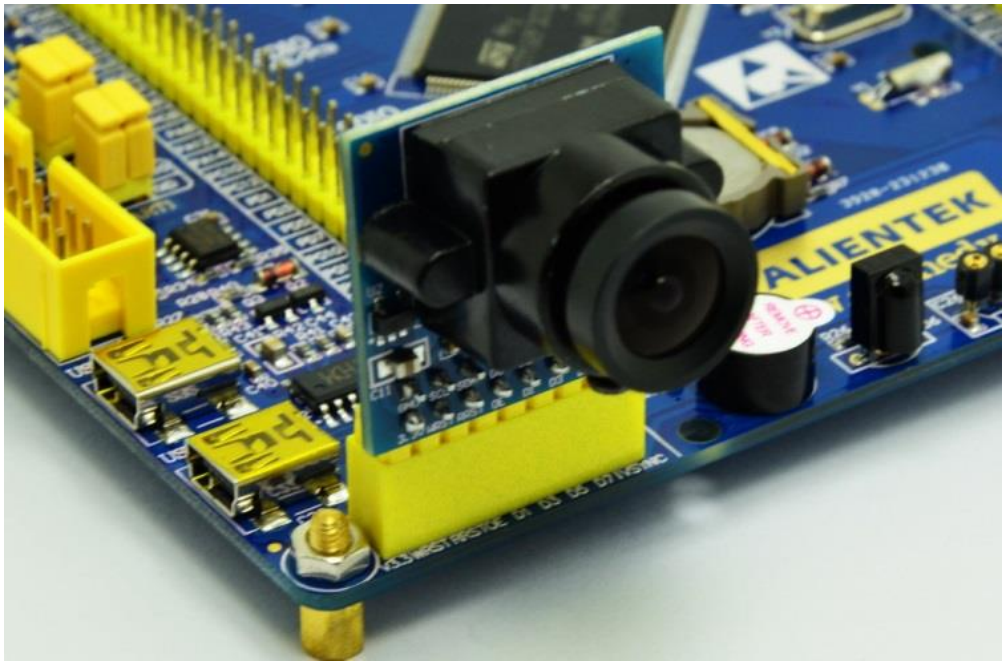


图 38.3.2 摄像头模块与开发板连接实物图

38.3 软件设计

打开上一章的工程，由于本章没用到 FLASH 模拟 EEPROM 相关代码，所以，先去掉 stmflash.c。然后，在 HARDWARE 文件夹下新建 OV7725 的文件夹。OV7725 文件夹如下文件：ov7725.c、ov7725.h、ov7725cfg.h、sccb.c、sccb.h 等 5 个文件。另外，本章还需要用到 exti.c、timer.c 和 tpad.c，所以将这三个.c 文件加入 HARDWARE 组下。

本章新增了多个文件，代码比较多，我们就不一一列出了。首先，我们来看 ov7725.c 里面的 OV7725_Init 函数，该函数代码如下：

```
//初始化 OV7725
//返回 0:成功
//返回其他值:错误代码
u8 OV7725_Init(void)
{
    u16 i=0;
    u16 reg=0;
    //设置 IO
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟
    __HAL_RCC_GPIOB_CLK_ENABLE();           //开启 GPIOB 时钟
    __HAL_RCC_GPIOC_CLK_ENABLE();           //开启 GPIOC 时钟
    __HAL_RCC_GPIOD_CLK_ENABLE();           //开启 GPIOD 时钟
    __HAL_RCC_GPIOG_CLK_ENABLE();           //开启 GPIOG 时钟
    __HAL_RCC_AFIO_CLK_ENABLE();            //开启 AFIO 时钟
    //PA8
    GPIO_Initure.Pin=GPIO_PIN_8;            //PA8
```

```
GPIO_Initure.Mode=GPIO_MODE_INPUT;                  //输入
GPIO_Initure.Pull=GPIO_PULLUP;                      //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;            //高速
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
HAL_GPIO_WritePin(GPIOA,GPIO_PIN_8,GPIO_PIN_SET); //PA8 输出高电平
//PB3,4
GPIO_Initure.Pin=GPIO_PIN_3|GPIO_PIN_4;                     //PB3,4
GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;              //推挽输出
HAL_GPIO_Init(GPIOB,&GPIO_Initure);
HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3,GPIO_PIN_SET); //PB3 输出高电平
HAL_GPIO_WritePin(GPIOB,GPIO_PIN_4,GPIO_PIN_SET); //PB4 输出高电平
//PC0~7,输入
GPIO_Initure.Pin=0xff;
GPIO_Initure.Mode=GPIO_MODE_INPUT;                  //输入
GPIO_Initure.Pull=GPIO_PULLUP;                      //上拉
GPIO_Initure.Speed=GPIO_SPEED_FREQ_HIGH;            //高速
HAL_GPIO_Init(GPIOC,&GPIO_Initure);
//PD6
GPIO_Initure.Pin=GPIO_PIN_6;                        //PD6
GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;              //推挽输出
HAL_GPIO_Init(GPIOD,&GPIO_Initure);
HAL_GPIO_WritePin(GPIOD,GPIO_PIN_6,GPIO_PIN_SET); //PD6 输出高电平
//PG14,15
GPIO_Initure.Pin=GPIO_PIN_14|GPIO_PIN_15;           //PG14,15
GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;              //推挽输出
HAL_GPIO_Init(GPIOG,&GPIO_Initure);
HAL_GPIO_WritePin(GPIOG,GPIO_PIN_14,GPIO_PIN_SET);//PG14 输出高电平
HAL_GPIO_WritePin(GPIOG,GPIO_PIN_15,GPIO_PIN_SET);//PG15 输出高电
__HAL_AFIO_REMAP_SWJ_DISABLE();                     //SWD
SCCB_Init();                                //初始化 SCCB 的 IO 口
if(SCCB_WR_Reg(0x12,0x80))return 1;     //软复位 OV7725
delay_ms(50);
reg=SCCB_RD_Reg(0X1c);                      //读取厂家 ID 高八位
reg<<=8;
reg|=SCCB_RD_Reg(0X1d);                     //读取厂家 ID 低八位
if(reg!=OV7725_MID)
{
    printf("MID:%d\r\n",reg);
    return 1;
}
reg=SCCB_RD_Reg(0X0a);              //读取厂家 ID 高八位
reg<<=8;
reg|=SCCB_RD_Reg(0X0b);             //读取厂家 ID 低八位
```

```
    if(reg!=OV7725_PID)
    {
        printf("HID:%d\r\n",reg);
        return 2;
    }
    //初始化 OV7725,采用 QVGA 分辨率(320*240)
    for(i=0;i<sizeof(ov7725_init_reg_tb1)/sizeof(ov7725_init_reg_tb1[0]);i++)
    {
        SCCB_WR_Reg(ov7725_init_reg_tb1[i][0],ov7725_init_reg_tb1[i][1]);
    }
    return 0x00;    //ok
}
```

此部分代码先初始化 OV7725 相关的 IO 口（包括 SCCB_Init），然后最主要的是完成 OV7725 的寄存器序列初始化。OV7725 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《OV7725 software application note》），本章我们用到的配置序列，存放在 ov7725_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 ov7725cfg.h 里面。

接下来，我们看看 ov7725cfg.h 里面 ov7725_init_reg_tbl 的内容，ov7725cfg.h 文件的代码如下：

```
//初始化寄存器序列及其对应的值
const u8 ov7725_init_reg_tbl[][2]=
{
    /*输出窗口设置*/
    {CLKRC,     0x00}, //clock config
    {COM7,      0x06}, //VGA RGB565
    {HSTART,    0x3f}, //水平起始位置
    ……省略部分设置
    {COM3,      0x50},
    {COM5,      0xf5},
};
```

以上代码，我们省略了很多（全部贴出来太长了），我们大概了解下结构，每个条目的第一个字节为寄存器号（也就是寄存器地址），第二个字节为要设置的值，比如{CLKRC, 0x00}，就表示在 CLKRC（地址宏定义）地址，写入 0X00 这个值。

通过这么一长串寄存器的配置，我们就完成了 OV7725 的初始化，本章我们配置 OV7725 工作在 VGA 模式，RGB565 格式输出。 在完成初始化之后，我们既可以开始读取 OV7725 的数据了。

OV7725 文件夹里面的其他代码我们就不逐个介绍了，请大家参考光盘该例程源码。

因为本章我们还用到了帧率（LCD 显示的帧率）统计和中断处理，所以我们还需要修改 timer.c、timer.h、exti.c 及 exti.h 这几个文件。

在 timer.c 里面，我们新增 TIM6_Int_Init 和 TIM6_IRQHandler 两个函数，用于统计帧率，增加代码如下：

```
TIM_HandleTypeDef TIM6_Handler;         //定时器句柄
//基本定时器 6 中断初始化
```

```
//这里时钟选择为 APB1 的 2 倍，而 APB1 为 36M
//arr：自动重装值。
//psc：时钟预分频数
//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
void TIM6_Init(u16 arr,u16 psc)
{
    TIM6_Handler.Instance=TIM6;                                    //通用定时器 6
    TIM6_Handler.Init.Prescaler=psc;                               //分频系数
    TIM6_Handler.Init.CounterMode=TIM_COUNTERMODE_UP; //向上计数器
    TIM6_Handler.Init.Period=arr;                                  //自动装载值
    TIM6_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;//时钟分频因子
    HAL_TIM_Base_Init(&TIM6_Handler);

    HAL_TIM_Base_Start_IT(&TIM6_Handler);
                            //使能定时器 6 和定时器 6 更新中断：TIM_IT_UPDATE
}

//定时器底册驱动，开启时钟，设置中断优先级
//此函数会被 HAL_TIM_Base_Init()函数调用
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if(htim->Instance==TIM6)
    {
        __HAL_RCC_TIM6_CLK_ENABLE();              //使能 TIM6 时钟
        HAL_NVIC_SetPriority(TIM6_IRQn,1,3);
                            //设置中断优先级，抢占优先级 1，子优先级 3
        HAL_NVIC_EnableIRQ(TIM6_IRQn);            //开启 ITM6 中断
    }
}

//定时器 6 中断服务函数
void TIM6_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM6_Handler);
}

u8 ov_frame;  //统计帧数
//回调函数，定时器中断服务函数调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&TIM6_Handler))
    {
```

```
            printf("frame:%dfps\r\n",ov_frame);    //打印帧率
            ov_frame=0;
        }
    }
```

这里，我们用到基本定时器 TIM6 来统计帧率，也就是 1 秒钟中断一次，打印 ov_frame 的值，ov_frame 用于统计 LCD 帧率。

再在 timer.h 里面添加 TIM6_Int_Init 函数的定义，就完成对 timer.c 和 timer.h 的修改了。

在 exti.c 里面添加 EXTI8_Init 和 EXTI9_5_IRQHandler 函数，用于 OV7725 模块的 FIFO 写控制，exti.c 文件新增部分代码如下：

```
//外部中断初始化
void EXTI8_Init(void)
{
    GPIO_InitTypeDef GPIO_Iniure;
    __HAL_RCC_GPIOA_CLK_ENABLE();                    //开启 GPIOA 时钟
    GPIO_Iniure.Pin=GPIO_PIN_8;                      //PA8
    GPIO_Iniure.Mode=GPIO_MODE_IT_RISING;            //上升沿触发
    GPIO_Iniure.Pull=GPIO_PULLDOWN;
    HAL_GPIO_Init(GPIOA,&GPIO_Iniure);
    //中断线 8-PA8
    HAL_NVIC_SetPriority(EXTI9_5_IRQn,0,0);         //抢占优先级为 2，子优先级为 0
    HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);                //使能中断线 0
}
//中断服务函数
void EXTI9_5_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_8);       //调用中断处理公用函数
}
u8 ov_sta;      //帧中断标记
//中断服务程序中需要做的事情
//在 HAL 库中所有的外部中断服务函数都会调用此函数
//GPIO_Pin:中断引脚号
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(ov_sta==0)
    {
      OV7725_WRST=0;  //复位写指针
      OV7725_WRST=1;
      OV7725_WREN=1;  //允许写入 FIFO
      ov_sta++;       //帧中断加 1
    }else OV7725_WREN=0;//禁止写入 FIFO
}
```

因为 OV7725 的帧同步信号（OV_VSYNC）接在 PA8 上面，所以我们这里配置 PA8 作为中端输入，因为 STM32 的外部中断 5~9 共用一个中端服务函数（EXTI9_5_IRQHandler），所以

在该函数里面，我们需要先判断中断是不是来自中断线 8 的，然后再做处理。

中断处理部分流程：每当帧中断到来后，先判断 ov_sta 的值是否为 0，如果是 0，说明可以往 FIFO 里面写入数据，执行复位 FIFO 写指针，并允许 FIFO 写入，此时，AL422B 将从地址 0 开始，存储新一帧的图像数据。然后设置 ov_sta++即可，标记新的一帧数据正在存储中。如果 ov_sta 不为 0，说明之前存储在 FIFO 里面的一帧数据还未被读取过，直接禁止 FIFO 写入，等待 MCU 读取 FIFO 数据，以免数据覆盖。

然后，STM32 只需要判断 ov_sta 是否大于 0，来读取 FIFO 里面的数据，读完一帧后，设置 ov_sta 为 0，以免重复读取，同时还可以时能 FIFO 新帧的写入。

再在 exti.h 里面添加 EXTI8_Init 函数的定义，就完成对 exti.c 和 exti.h 的修改了。

最后，打开 main.c 文件，代码如下：

```
//由于 OV7725 传感器安装方式原因,OV7725_WINDOW_WIDTH 相当于 LCD 的高度，
//OV7725_WINDOW_HEIGHT 相当于 LCD 的宽度
//注意：此宏定义只对 OV7725 有效
#define  OV7725_WINDOW_WIDTH          320      // <=320
#define  OV7725_WINDOW_HEIGHT         240      // <=240
const u8*LMODE_TBL[6]={"Auto","Sunny","Cloudy","Office","Home","Night"};
                                                //6 种光照模式
const u8*EFFECTS_TBL[7]={"Normal","Negative","B&W","Redish","Greenish",
                         "Bluish","Antique"};  //7 种特效
extern u8 ov_sta;        //在 exit.c 里  面定义
extern u8 ov_frame;      //在 timer.c 里面定义
//更新 LCD 显示(OV7725)
void OV7725_camera_refresh(void)
{
    u32 i,j;
    u16 color;
    if(ov_sta)//有帧中断更新
    {
        LCD_Scan_Dir(U2D_L2R);      //从上到下,从左到右
        LCD_Set_Window((lcddev.width-OV7725_WINDOW_WIDTH)/2,
        (lcddev.height-OV7725_WINDOW_HEIGHT)/2,OV7725_WINDOW_WIDTH,
        OV7725_WINDOW_HEIGHT);//将显示区域设置到屏幕中央
        if(lcddev.id==0X1963)
        LCD_Set_Window((lcddev.width-OV7725_WINDOW_WIDTH)/2,
        (lcddev.height-OV7725_WINDOW_HEIGHT)/2,OV7725_WINDOW_HEIGHT,
        OV7725_WINDOW_WIDTH);//将显示区域设置到屏幕中央
        LCD_WriteRAM_Prepare();     //开始写入 GRAM
        OV7725_RRST=0;              //开始复位读指针
        OV7725_RCK_L;
        OV7725_RCK_H;
        OV7725_RCK_L;
        OV7725_RRST=1;              //复位读指针结束
        OV7725_RCK_H;
```

```
            for(i=0;i<OV7725_WINDOW_HEIGHT;i++)
            {
                for(j=0;j<OV7725_WINDOW_WIDTH;j++)
                {
                    OV7725_RCK_L;
                    color=GPIOC->IDR&0XFF; //读数据
                    OV7725_RCK_H;
                    color<<=8;
                    OV7725_RCK_L;
                    color|=GPIOC->IDR&0XFF; //读数据
                    OV7725_RCK_H;
                    LCD->LCD_RAM=color;
                }
            }
            ov_sta=0;                   //清零帧中断标记
            ov_frame++;
            LCD_Scan_Dir(DFT_SCAN_DIR);     //恢复默认扫描方向
        }
    }
    int main(void)
    {
        u8 key, i=0, tm=0, lightmode=0,effect=0, saturation=0,brightness=0,contrast=0;
        u8 msgbuf[15];//消息缓存区
        HAL_Init();                             //初始化 HAL 库
        Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
        delay_init(72);                         //初始化延时函数
        uart_init(115200);                      //初始化串口
        LED_Init();                             //初始化 LED
        KEY_Init();                             //初始化按键
        LCD_Init();                             //初始化 LCD FSMC 接口
        TPAD_Init(6);                           //触摸按键初始化
        POINT_COLOR=RED;                        //设置字体为红色
        LCD_ShowString(30,50,200,16,16," ELITE STM32");
        LCD_ShowString(30,70,200,16,16,"OV7725 TEST");
        LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(30,110,200,16,16,"2019/9/19");
        LCD_ShowString(30,130,200,16,16,"KEY0:Light Mode");
        LCD_ShowString(30,150,200,16,16,"KEY1:Saturation");
        LCD_ShowString(30,170,200,16,16,"KEY_UP:Contrast");
        LCD_ShowString(30,210,200,16,16,"TPAD:Effects");
        LCD_ShowString(30,230,200,16,16,"OV7725 Init...");
        while(1)//初始化 OV7725
        {
```

```
    if(OV7725_Init()==0)
    {
        LCD_ShowString(30,210,200,16,16,"OV7725 Init OK          ");
        while(1)
        {
            key=KEY_Scan(0);
            if(key==KEY0_PRES)
            {
                OV7725_Window_Set(OV7725_WINDOW_WIDTH,
                        OV7725_WINDOW_HEIGHT,0);//QVGA 模式输出
                break;
            }else if(key==KEY1_PRES)
            {
                OV7725_Window_Set(OV7725_WINDOW_WIDTH,
                        OV7725_WINDOW_HEIGHT,1);//VGA 模式输出
                break;
            }
            i++;
            if(i==100)LCD_ShowString(30,230,210,16,16,"KEY0:QVGA EY1:VGA");
                                        //闪烁显示提示信息
            if(i==200)
            {
                LCD_Fill(30,230,210,250+16,WHITE);
                i=0;
            }
            delay_ms(5);
        }
        OV7725_Light_Mode(lightmode);
        OV7725_Color_Saturation(saturation);
        OV7725_Brightness(brightness);
        OV7725_Contrast(contrast);
        OV7725_Special_Effects(effect);
        OV7725_CS=0;
        break;
    }else
    {
        LCD_ShowString(30,210,200,16,16,"OV7725 Error!!");
        delay_ms(200);
        LCD_Fill(30,210,239,246,WHITE);
        delay_ms(200);
    }
}
TIM6_Init(10000,7199);          //10Khz 计数频率,1 秒钟中断
```

```
EXTI8_Init();                       //使能外部中断 8,捕获帧中断
LCD_Clear(BLACK);
while(1)
{
    key=KEY_Scan(0);//不支持连按
    if(key)
    {
        tm=20;
        switch(key)
        {
            case KEY0_PRES: //灯光模式 Light Mode
                lightmode++;
                if(lightmode>5)lightmode=0;
                OV7725_Light_Mode(lightmode);
                sprintf((char*)msgbuf,"%s",LMODE_TBL[lightmode]);
                break;
            case KEY1_PRES: //饱和度 Saturation
                saturation++;
                if(saturation>4)saturation=-4;
                else if(saturation<-4)saturation=4;
                OV7725_Color_Saturation(saturation);
                sprintf((char*)msgbuf,"Saturation:%d",saturation);
                break;
            case WKUP_PRES:     //对比度 Contrast
                contrast++;
                if(contrast>4)contrast=-4;
                else if(contrast<-4)contrast=4;
                OV7725_Contrast(contrast);
                sprintf((char*)msgbuf,"Contrast:%d",contrast);
                break;
        }
    }
    if(TPAD_Scan(0))//检测到触摸按键
    {
        effect++;
        if(effect>6)effect=0;
        OV7725_Special_Effects(effect);     //设置特效
        sprintf((char*)msgbuf,"%s",EFFECTS_TBL[effect]);
        tm=20;
    }
    OV7725_camera_refresh();        //更新显示
    if(tm)
    {
```

```
                LCD_ShowString((lcddev.width-240)/2+30,(lcddev.height-320)/2+
                                60,200,16,16,msgbuf);
                tm--;
            }
            i++;
            if(i>=15)//DS0 闪烁.
            {
                i=0;
                LED0=!LED0;
            }
        }
    }
```

OV7725_camera_refresh 函数，该函数用于读取摄像头模块自带 FIFO 里面的数据，并显示在 LCD 上面。对于 OV7725 我们可以通过宏定义设置图像窗口输出的大小，设置的宏定义：

```
//注意：此宏定义只对 OV7725 有效
#define  OV7725_WINDOW_WIDTH     320 // <=320
#define  OV7725_WINDOW_HEIGHT    240 // <=240
```

对分辨率大于 320*240 的屏幕，则通过开窗函数（LCD_Set_Window）将显示区域开窗在屏幕的正中央。注意，为了提高 FIFO 读取速度，我们将 FIFO_RCK 的控制，采用快速 IO 控制，关键代码如下（在 ov7725.h 里面）：

```
#define OV7725_RCK_H     GPIOB->BSRR=1<<4   //设置读数据时钟高电平
#define OV7725_RCK_L     GPIOB->BRR=1<<4    //设置读数据时钟低电平
```

OV7725_RCK_H 和 OV7725_RCK_L 就用到了 BSRR 和 BRR 这两个寄存器，以实现快速 IO 设置，从而提高读取速度。

前面提到，我们要用 USMART 来设置摄像头的参数，我们只需要在 usmart_nametab 里面添加 SCCB_WR_Reg 和 SCCB_RD_Reg 这两个函数，就可以轻松调试摄像头了。

最后，为了得到最快的显示速度，我们将 MDK 的代码优化等级设置为-O2 级别（在 C/C++选项卡里面设置），这样 OV7725 的显示帧率可达 23 帧。注意：因为 TPAD_Scan 扫描占用了很多时间（>15ms/次），所以帧率比较慢，屏蔽该函数，可以提高帧率。

38.4 下载验证

本实验我们使用 OV7725 摄像头模块，我们编译代码并下载到 ALIENTEK 精英 STM32 开发板上，得到如图 38.5.1 所示界面：

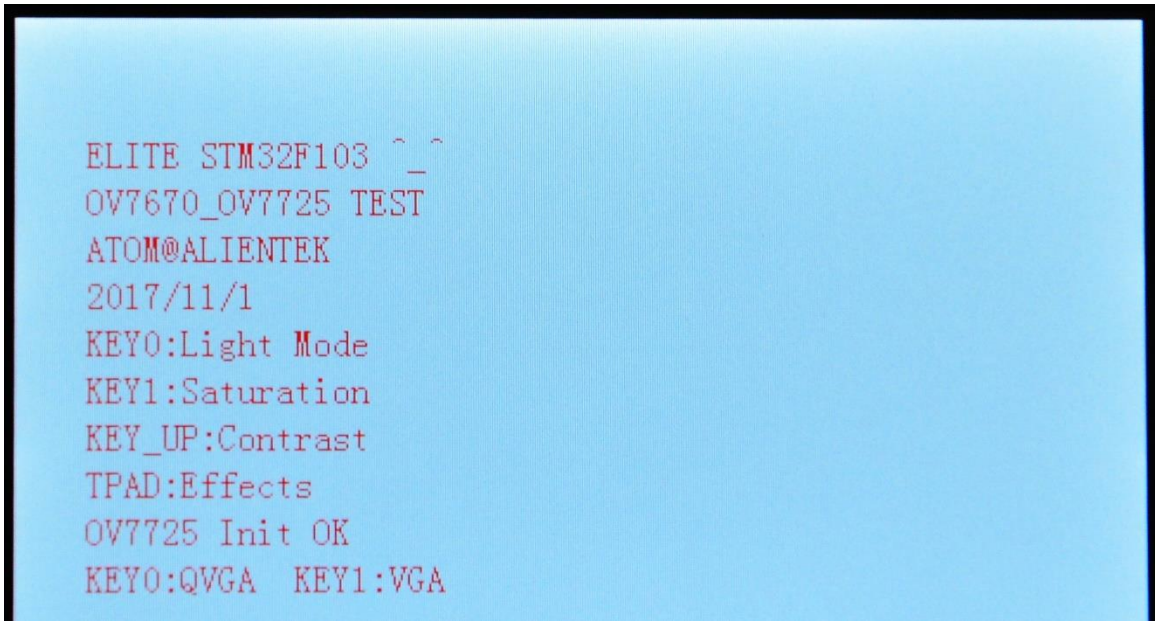


图 38.5.1 程序运行效果图

随后，我们通过按 KEY0 和 KEY1 选择输出模式。当选择 QVGA 模式时，OV7725 直接输出 320*240 分辨图像数据，此模式相对 VGA 模式视角较广但画面没那么清晰细腻。当选择 VGA 模式时，其实是将 640*480 窗口截取中间 320*240 的图像输出，该模式下拍出的图像较为清晰细腻但视角较小。我们按 KEY0 键进入摄像头拍摄界面，进入后我们可以按不同的按键（KEY0、KEY1、KEY_UP、TPAD 等），来设置摄像头的相关参数和模式，得到不同的成像效果。同时，你还可以使用串口，通过 USMART 调用 SCCB_WR_Reg 等函数，来设置 OV7725 的各寄存器，从而达到调试测试的目的，如图 38.5.2 所示：

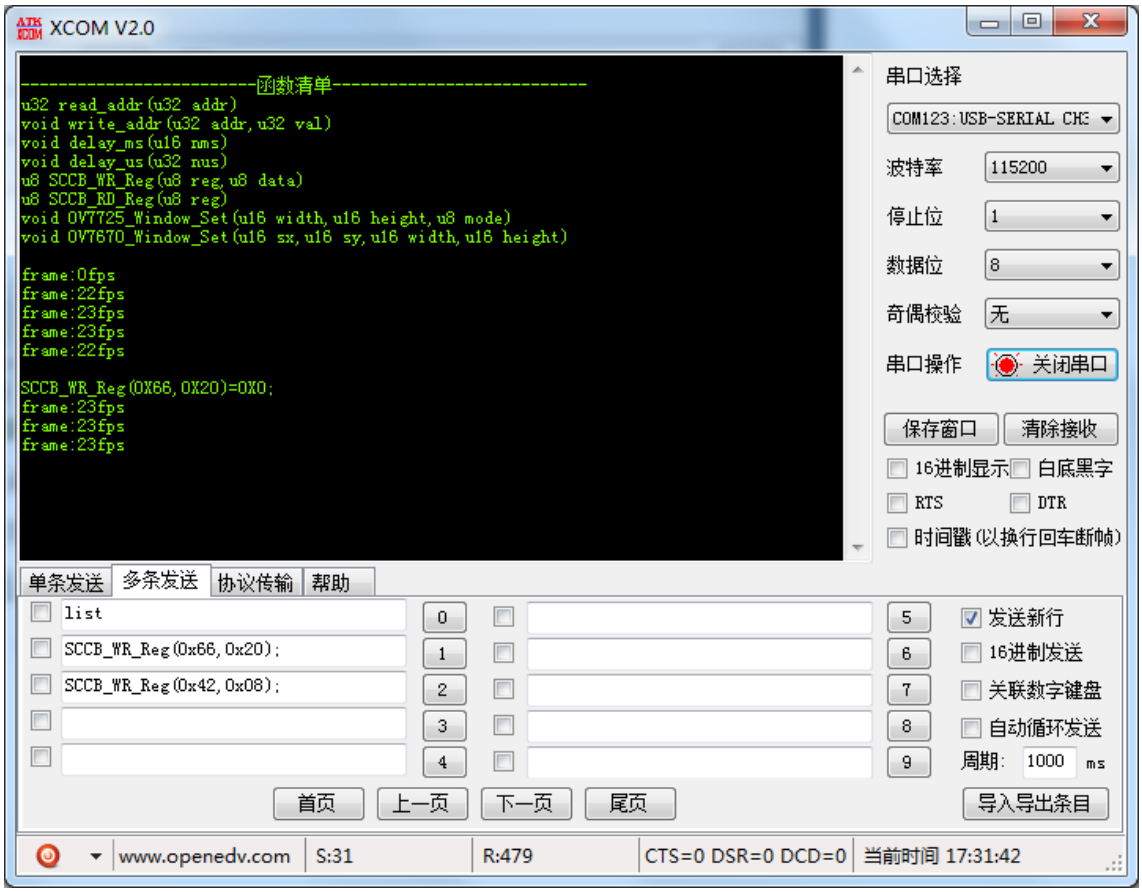


图 38.5.2 USMART 调试 OV7725

从上图还可以看出，LCD 显示帧率为 23 帧左右。图中，我们可以通过 USMART 发送 SCCB_WR_Reg(0X66,0X20)，即可设置 OV7725 输出彩条，方便大家测试。

第三十九章 内存管理实验

内存是单片机内部最宝贵的资源之一，如果都是以数组，变量的方式使用，是十分不方便的。举个简单的例子：要在 lcd 上面实现一个文件浏览界面，肯定需要内存来存储文件名字，首先，文件名字的长度是不定的，而且你不知道有多少个文件名要存储，如果用数组的方式来存储，数组定义大了浪费，定义少了，不够。所以在这样的应用下，使用数组等固定的方式来实现就很头疼了。

但是，如果有内存管理，则可以很方便的实现这种应用，我们可以根据每个文件名的长度来自由申请内存，用完之后，又可以释放内存，其他地方可以继续申请使用，非常方便，这就是内存管理的优势。

本章，我们将学习内存管理，实现对内存的动态管理。本章分为如下几个部分：

39.1 内存管理简介
39.2 硬件设计
39.3 软件设计
39.4 下载验证

39.1 内存管理简介

内存管理，是指软件运行时对计算机内存资源的分配和使用的技术。其最主要的目的是如何高效，快速的分配，并且在适当的时候释放和回收内存资源。内存管理的实现方法有很多种，他们其实最终都是要实现 2 个函数：malloc 和 free；malloc 函数用于内存申请，free 函数用于内存释放。

本章，我们介绍一种比较简单的办法来实现：分块式内存管理。下面我们介绍一下该方法的实现原理，如图 39.1.1 所示：

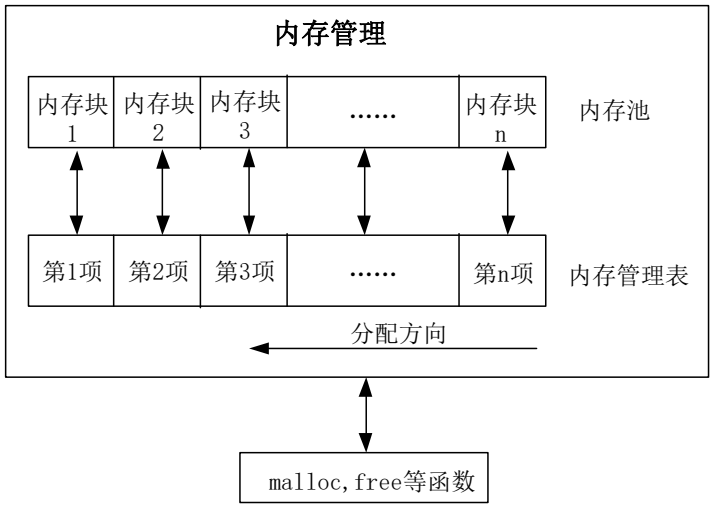


图 39.1.1 分块式内存管理原理

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。

内寸分配方向如图所示，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。

39.2 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 用于释放内存，KEY_UP 用于写数据到申请到的内存里面。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） KEY0/KEY1/KEY_UP 等三个按键

3） 串口

4） TFTLCD 模块

这些我们都已经介绍过，接下来我们开始软件设计。

39.3 软件设计

打开上一章的工程，由于本章没用到 OV7670 和定时器等相关代码，所以，先去掉这些代码（此时 HARDWARE 组下仅剩：led.c、ILI93xx.c 和 key.c 三个文件）。

本章，我们将内存管理部分单独做一个分组，在工程目录下新建一个 MALLOC 的文件夹，然后新建 malloc.c 和 malloc.h 两个文件，将他们保存在 MALLOC 文件夹下。

在 MDK 新建一个 MALLOC 的组，然后将 malloc.c 文件加入到该组，并将 MALLOC 文件夹添加到头文件包含路径。

打开 malloc.c 文件，输入如下代码：

```
//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)));//外部内存池
//内存管理表
u16 mem1mapbase[MEM1_ALLOC_TABLE_SIZE];        //内部 SRAM 内存池 MAP
u16 mem2mapbase[MEM2_ALLOC_TABLE_SIZE] __attribute__((at(0X68000000+
MEM2_MAX_SIZE)));                                          //外部 SRAM 内存池 MAP
//内存管理参数
const u32 memtblsize[SRAMBANK]={MEM1_ALLOC_TABLE_SIZE,
                                MEM2_ALLOC_TABLE_SIZE}; //内存表大小
const u32 memblksize[SRAMBANK]={MEM1_BLOCK_SIZE,MEM2_BLOCK_SIZE};
//内存分块大小
const u32 memsize[SRAMBANK]={MEM1_MAX_SIZE,MEM2_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,              //内存池
    mem1mapbase,mem2mapbase,        //内存管理状态表
    0,0,                            //内存管理未就绪
};
//复制内存
```

```
//*des:目的地址
//*src:源地址
//n:需要复制的内存长度(字节为单位)
void mymemcpy(void *des,void *src,u32 n)
{
    u8 *xdes=des;
    u8 *xsrc=src;
    while(n--)*xdes++=*xsrc++;
}
//设置内存
//*s:内存首地址
//c :要设置的值
//count:需要设置的内存大小(字节为单位)
void mymemset(void *s,u8 c,u32 count)
{
    u8 *xs = s;
    while(count--)*xs++=c;
}
//内存管理初始化
//memx:所属内存块
void 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/memblksize[memx];                     //获取需要分配的连续内存块数
    if(size%memblksize[memx])nmemb++;
    for(offset=memtblsize[memx]-1;offset>=0;offset--)    //搜索整个内存控制区
    {
        if(!mallco_dev.memmap[memx][offset])cmemb++;//连续空内存块数增加
        else cmemb=0;                                //连续内存块清零
        if(cmemb==nmemb)                             //找到了连续 nmemb 个空内存块
        {
            for(i=0;i<nmemb;i++)                     //标注内存块非空
            {
                mallco_dev.memmap[memx][offset+i]=nmemb;
            }
            return (offset*memblksize[memx]);//返回偏移地址
        }
    }
    return 0XFFFFFFFF;//未找到符合分配条件的内存块
}
//释放内存(内部调用)
//memx:所属内存块
//offset:内存地址偏移
//返回值:0,释放成功;1,释放失败;
u8 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);
        return 1;//未初始化
    }
    if(offset<memsize[memx])//偏移在内存池内.
    {
        int index=offset/memblksize[memx];           //偏移所在内存块号码
        int nmemb=mallco_dev.memmap[memx][index]; //内存块数量
        for(i=0;i<nmemb;i++)                         //内存块清零
        {
```

```
                mallco_dev.memmap[memx][index+i]=0;
            }
            return 0;
        }else return 2;//偏移超区了.
}
//释放内存(外部调用)
//memx:所属内存块
//ptr:内存首地址
void myfree(u8 memx,void *ptr)
{
    u32 offset;
    if(ptr==NULL)return;//地址为 0.
    offset=(u32)ptr-(u32)mallco_dev.membase[memx];
    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
        myfree(memx,ptr);                                                  //释放旧内存
        return (void*)((u32)mallco_dev.membase[memx]+offset);  //返回新内存首地址
    }
}
```

这里，我们通过内存管理控制器 mallco_dev 结构体（mallco_dev 结构体见 malloc.h），实现对两个内存池的管理控制。一个是内部 SRAM 内存池，定义为：

```
__align(32) u8 mem1base[MEM1_MAX_SIZE];
```

另外一个是外部 SRAM 内存池，定义为：

```
__align(32) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0X68000000)));
```

其中，MEM1_MAX_SIZE 和 MEM2_MAX_SIZE 为在 malloc.h 里面定义的内存池大小，外部内存池指定地址为 0X68000000，也就是从外部 SRAM 的首地址开始的，内部内存则由编译器自动分配。__align(32)定义内存池为 32 字节对齐，以适应各种不同场合的需求。

特别注意：精英板并没有外扩 SRAM，所以，实际上，外部内存池，在精英板上面，是不能使用的，在精英板上面，只可以使用内部内存池！！！

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 39.1 接所介绍的那样分配和释放内存，不过这两个函数只是内部调用，外部调用我们使用的是 mymalloc 和 myfree 两个函数。其他函数我们就不多介绍了，保存 malloc.c，然后，打开 malloc.h，在该文件里面输入如下代码：

```
#ifndef __MALLOC_H
#define __MALLOC_H
#include "sys.h"
#ifndef NULL
#define NULL 0
#endif
//定义两个内存池
#define SRAMIN          0                       //内部内存池
#define SRAMEX          1                       //外部内存池
#define SRAMBANK        2                       //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               40*1024     //最大管理内存 40K
#define MEM1_ALLOC_TABLE_SIZE       MEM1_MAX_SIZE/MEM1_BLOCK_SIZE
//内存表大小
//mem2 内存参数设定.mem2 的内存池处于外部 SRAM 里面
#define MEM2_BLOCK_SIZE             32          //内存块大小为 32 字节
#define MEM2_MAX_SIZE               1 *32       //精英板没外扩内存,设一个最小值
#define MEM2_ALLOC_TABLE_SIZE       MEM2_MAX_SIZE/MEM2_BLOCK_SIZE
//内存表大小
//内存管理控制器
struct _m_mallco_dev
{
    void (*init)(u8);                           //初始化
    u8 (*perused)(u8);                          //内存使用率
    u8  *membase[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 和 MEM2_BLOCK_SIZE，都是 32 字节。内存池总大小，内部为 40K，外部为 32 字节（**精英板并没有用到外部内存池，但是这里不能设置为 0，否则编译报错！**）。MEM1_ALLOC_TABLE_SIZE 和 MEM2_ALLOC_TABLE_SIZE，则分别代表内存池 1 和 2 的内存管理表大小。

从这里可以看出，如果内存分块越小，那么内存管理表就越大，当分块为 2 字节 1 个块的时候，内存管理表就和内存池一样大了（管理表的每项都是 u16 类型）。显然是不合适的，我们这里取 32 字节，比例为 1:16，内存管理表相对就比较小了。

其他就不多说了，大家自行看代码理解就好。保存此部分代码。最后，打开 main.c 文件，修改代码如下：

```
int main(void)
{
    u8 key; u8 i=0;
    u8 *p=0; u8 *tp=0;
    u8 paddr[18];                    //存放 P Addr:+p 地址的 ASCII 值

    HAL_Init();                      //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);  //设置时钟,72M
    delay_init(72);                  //初始化延时函数
    uart_init(115200);               //初始化串口
    LED_Init();                      //初始化 LED
    KEY_Init();                      //初始化按键
    LCD_Init();                      //初始化 LCD FSMC 接口
    SRAM_Init();                     //初始化外部 SRAM
    my_mem_init(SRAMIN);             //初始化内部内存池
    POINT_COLOR=RED;                 //设置字体为红色
    LCD_ShowString(30,50,200,16,16,"ELITE STM32 ^_^");
    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,"2019/9/19");
    LCD_ShowString(30,130,200,16,16,"KEY0:Malloc  KEY1:Free");
```

```
LCD_ShowString(30,150,200,16,16,"KEY_UP:Write");
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:     %");
while(1)
{
    key=KEY_Scan(0);      //不支持连按
    switch(key)
    {
        case 0:                         //没有按键按下
            break;
        case KEY0_PRES:                 //KEY0 按下
            p=mymalloc(SRAMIN,2048);//申请 2K 字节
            if(p!=NULL)sprintf((char*)p,"Memory Malloc Test%03d",i);
                                        //向 p 写入一些内容
            break;
        case KEY1_PRES:                 //KEY1 按下
            myfree(SRAMIN,p);           //释放内存
            p=0;                        //指向空地址
            break;
        case WKUP_PRES:                 //KEY UP 按下
            if(p!=NULL)
            {
                sprintf((char*)p,"Memory Malloc Test%03d",i);//更新显示内容
                LCD_ShowString(30,250,200,16,16,p);          //显示 P 的内容
            }
            break;
    }
    if(tp!=p)
    {
        tp=p;
        sprintf((char*)paddr,"P Addr:0X%08X",(u32)tp);
        LCD_ShowString(30,230,200,16,16,paddr); //显示 p 的地址
        if(p)LCD_ShowString(30,250,200,16,16,p);//显示 P 的内容
        else LCD_Fill(30,250,239,266,WHITE);     //p=0,清除显示
    }
    delay_ms(10);
    i++;
    if((i%20)==0)//DS0 闪烁.
    {
        LCD_ShowNum(30+96,190,my_mem_perused(SRAMIN),3,16);
                                                  //显示内部内存使用率
        LED0=!LED0;
```

```
            }
        }
    }
```

该部分代码比较简单，主要是对 mymalloc 和 myfree 的应用。不过这里提醒大家，如果对一个指针进行多次内存申请，而之前的申请又没释放，那么将造成“内存泄露”，这是内存管理所不希望发生的，久而久之，可能导致无内存可用的情况！所以，在使用的时候，请大家一定记得，申请的内存在用完以后，一定要释放。

main 函数中，SRAMIN 即在 malloc.h 定义的内部内存标志，表明要操作的对象是内部内存，因为精英板并没有外部内存，所以大家在使用的时候，只可以使用内部内存，**如果试图对外部内存（SRAMEX）进行操作，那么可能引起程序死机**。

另外，本章希望利用 USMART 调试内存管理，所以在 USMART 里面添加了 mymalloc 和 myfree 两个函数，用于测试内存分配和内存释放。大家可以通过 USMART 自行测试。

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 精英 STM32F103 上，得到如图 39.4.1 所示界面：

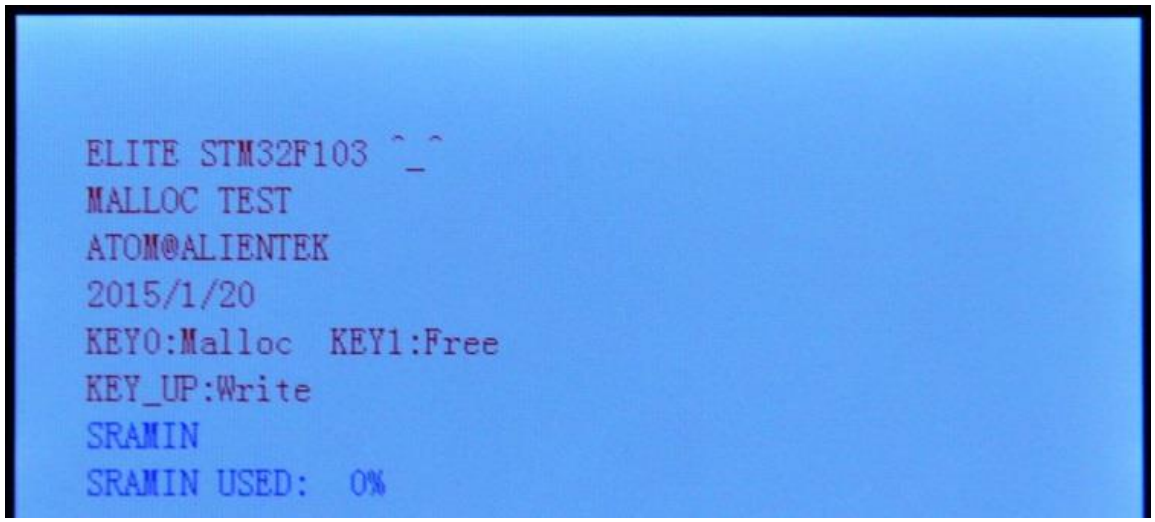


图 39.4.1 程序运行效果图

可以看到，内存的使用率为 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 5%了，同时看到下面提示了指针 p 所指向的地址（其实就是被分配到的内存地址）和内容。多按几次 KEY0，可以看到内存使用率持续上升（注意对比 p 的值，可以发现是递减的，说明是从顶部开始分配内存！），此时如果按下 KEY1，可以发现内存使用率降低了 5%，但是再按 KEY1 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”。

本章，我们还可以借助 USMART，测试内存的分配和释放，有兴趣的朋友可以动手试试。如图 39.4.2 所示：



图 39.4.2 USMART 测试内存管理函数

第四十章　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。

ALIENTKE 精英 STM32F103 自带了标准的 SD 卡接口，使用 STM32F1 自带的 SDIO 接口驱动，4 位模式，最高通信速度可达 24Mhz，最高每秒可传输数据 12M 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 精英 STM32F103 上实现 SD 卡的读取。本章分为如下几个部分：

40.1 SDIO 接口简介
40.2 硬件设计
40.3 软件设计
40.4 下载验证

40.1 SDIO 简介

ALIENTEK 精英 STM32F103 自带 SDIO 接口，本节，我们将简单介绍 STM32F1 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。

40.1.1 SDIO 主要功能及框图

STM32F1 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。
- 数据和命令输出使能信号，用于控制外部双向驱动器。

STM32F1 的 SDIO 控制器包含 2 个部分：SDIO 适配器模块和 AHB 总线接口，其功能框图如图 40.1.1.1 所示：

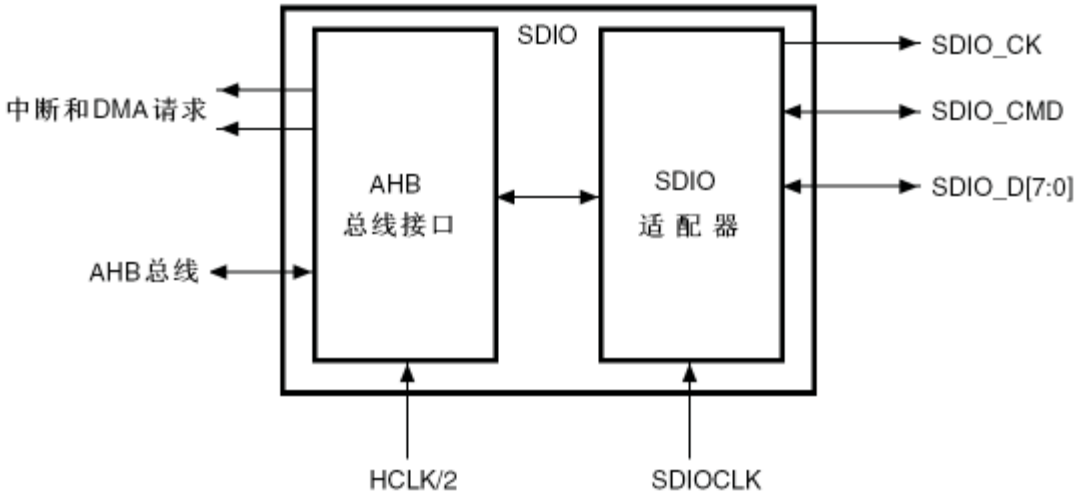


图 40.1.1.1 STM32F1 的 SDIO 控制器功能框图

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)

40.1.2 SDIO 的时钟

从图 40.1.1.1 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，其频率等于 AHB 总线频率（HCLK），并用于产生 SDIO_CK 时钟。

AHB 总线接口时钟（HCLK/2）：该时钟用于驱动 SDIO 的 AHB 总线接口，其频率为 HCLK/2。

前面提到，我们的 SD 卡时钟（SDIO_CK），根据卡的不同，可能有好几个区间，这就涉及到时钟频率的设置，SDIO_CK 与 SDIOCLK 的关系为：

SDIO_CK=SDIOCLK/(2+CLKDIV)

其中，SDIOCLK 为 HCLK，一般是 72Mhz，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）。

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）。

40.1.3 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 40.1.3.1 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 40.1.3.1 SDIO 命令格式

所有的命令都是由 STM32F1 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。STM32F1 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 40.1.3.2 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 40.1.3.2 SDIO 命令格式

长响应的格式如表 40.1.3.3 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 40.1.3.3 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）响应属于短响应，其长度为 48 位，R1 响应的格式如表 40.1.3.4 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 40.1.3.4 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《STM32 中文参考手册》第 20 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 40.1.3.1 所示：

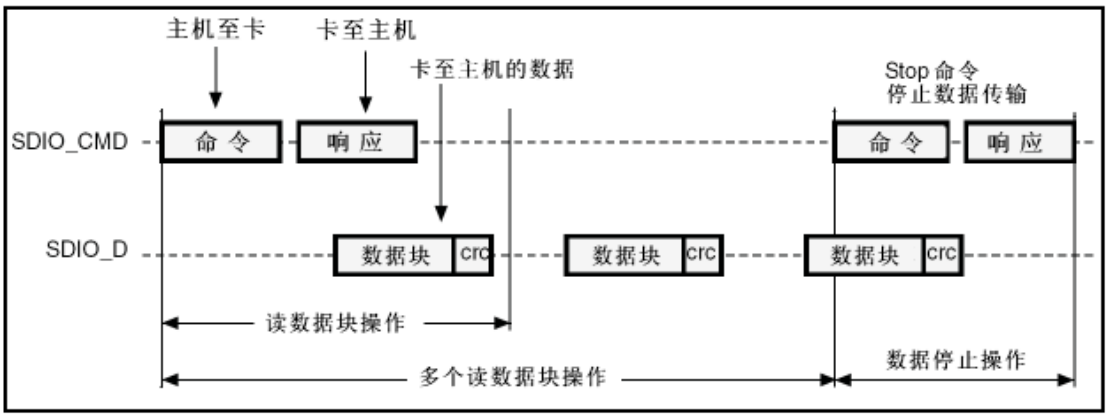


图 40.1.3.1 SDIO（多）数据块读操作

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 40.1.3.2 所示：

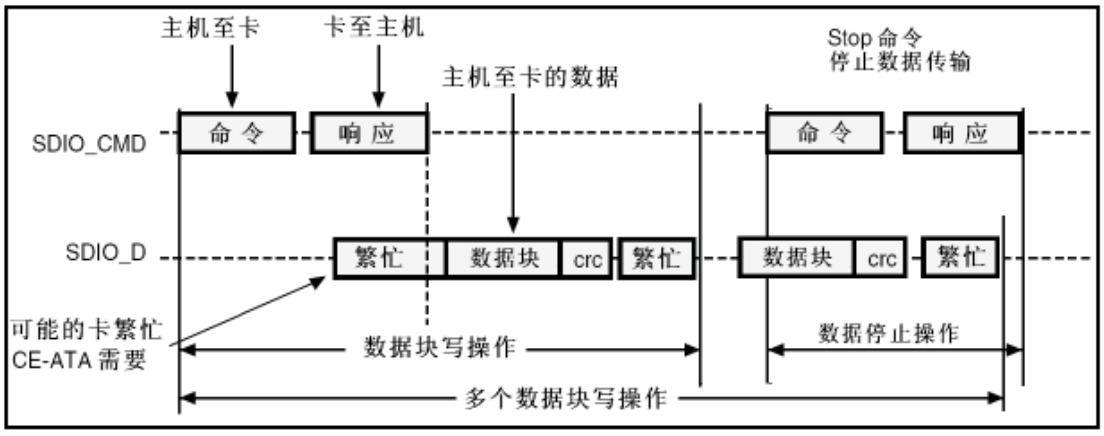


图 40.1.3.2 SDIO（多）数据块写操作

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。

40.1.4 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 40.1.4.1 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
保留																														PWRCTRL	
res																														rw	

位31:2	保留，始终读为0。
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：电源关闭，卡的时钟停止。 01：保留。 10：保留的上电状态。 11：上电状态，卡的时钟开启。

图 40.1.4.1 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 40.1.4.2 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
保留																	HWFC_EN	NEGEDGE	WIDBUS		BYPASS	PWRSAV	CLKEN	CLKDIV							
res																	rw	rw	rw		rw	rw	rw	r/w							

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位总线模式，使用SDIO_D[3:0]。 10：8位总线模式，使用SDIO_D[7:0]。
位10	**BYPASS**：旁路时钟分频器 (Clock divider bypass enable bit) 0：关闭旁路：驱动SDIO_CK输出信号之前，依据CLKDIV数值对SDIOCLK分频。 1：使能旁路：SDIOCLK直接驱动SDIO_CK输出信号。
位9	**PWRSAV**：省电配置位 (Power saving configuration bit) 为了省电，当总线为空闲时，设置PWRSAV位可以关闭SDIO_CK时钟输出。 0：始终输出SDIO_CK。 1：仅在有总线活动时才输出SDIO_CK。
位8	**CLKEN**：时钟使能位 (Clock enable bit) 0：SDIO_CK关闭。 1：SDIO_CK使能。
位7:0	**CLKDIV**：时钟分频系数 (Clock divide factor) 这个域定义了输入时钟(SDIOCLK)与输出时钟(SDIO_CK)间的分频系数： SDIO_CK频率 = SDIOCLK/[CLKDIV + 2]。

图 40.1.4.2 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，设置为 1，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，

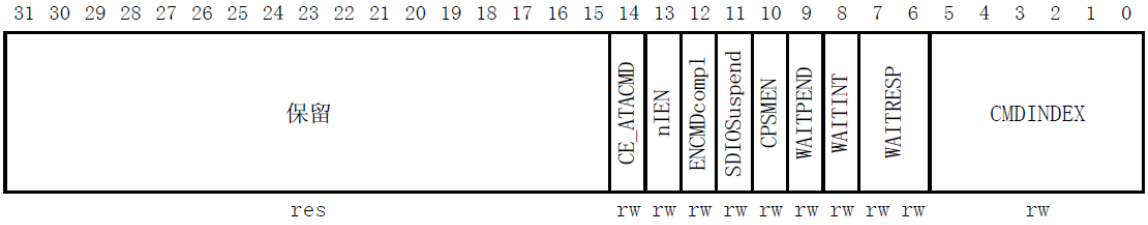
但只有低 6 位有效，比较简单，用于存储最后收到的命令响应中的命令索引。如果传输的命令响应不包含命令索引，则该寄存器的内容不可预知。

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 40.1.4.1 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 40.1.4.1 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 40.1.4.3 所示：



位10	**CPSMEN**：命令通道状态机(CPSM)使能位 (Command path state machine (CPSM) Enable bit) 如果设置该位，则使能CPSM。
位7:6	**WAITRESP**：等待响应位 (Wait for response bits) 这2位指示CPSM是否需要等待响应，如果需要等待响应，则指示响应类型。 00：无响应，期待CMDSENT标志 01：短响应，期待CMDREND或CCRCFAIL标志 10：无响应，期待CMDSENT标志 11：长响应，期待CMDREND或CCRCFAIL标志
位5:0	**CMDINDEX**：命令索引 (Command index) 命令索引是作为命令的一部分发送到卡中。

图 40.1.4.3 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《STM32 中文参考手册》V10 的第 368 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《STM32 中文参考手册》V10 的第 372 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 40.1.4.4 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
保留																				SDIOEN	RWMOD	RWSTOP	RWSTART	DBLOCKSIZE				DMAEN	DTMODE	DTDIR	DTEN
res																				rw	rw	rw	rw	rw				rw	rw	rw	rw

位11	**SDIOEN**：SD I/O使能功能 (SD I/O enable functions) 如果设置了该位，则DPSM执行SD I/O卡特定的操作。
位10	**RWMOD**：读等待模式 (Read wait mode) 0：停止SDIO_CK控制读等待； 1：使用SDIO_D2控制读等待。
位9	**RWSTOP**：读等待停止 (Read wait stop) 0：如果设置了RWSTART，执行读等待； 1：如果设置了RWSTART，停止读等待。
位8	**RWSTART**：读等待开始 (Read wait start) 设置该位开始读等待操作。
位7:4	**DBLOCKSIZE**：数据块长度 (Data block size) 当选择了块数据传输模式，该域定义数据块长度： 0000：块长度 = 2^0 = 1字节；　1000：块长度 = 2^8 = 256字节； 0001：块长度 = 2^1 = 2字节；　1001：块长度 = 2^9 = 512字节； 0010：块长度 = 2^2 = 4字节；　1010：块长度 = 2^{10} = 1024字节； 0011：块长度 = 2^3 = 8字节；　1011：块长度 = 2^{11} = 2048字节； 0100：(十进制4)块长度 = 2^4 = 16字节；　1100：块长度 = 2^{12} = 4096字节； 0101：(十进制5)块长度 = 2^5 = 32字节；　1101：块长度 = 2^{13} = 8192字节； 0110：(十进制6)块长度 = 2^6 = 64字节；　1110：块长度 = 2^{14} = 16384字节； 0111：块长度 = 2^7 = 128字节；　1111：保留。
位3	**DMAEN**：DMA使能位 (DMA enable bit) 0：关闭DMA； 1：使能DMA。
位2	**DTMODE**：数据传输模式 (Data transfer mode selection) 0：块数据传输； 1：流数据传输。
位1	**DTDIR**：数据传输方向 (Data transfer direction selection) 0：控制器至卡； 1：卡至控制器。

图 40.1.4.4 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 40.1.4.5 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
保留								CEATAEND	SDIOIT	RXDAVL	TXDAVL	RXFIFOE	TXFIFOE	RXFIFOF	TXFIFOF	RXFIFOHF	TXFIFOHE	RXACT	TXACT	CMDACT	DBCKEND	STBITERR	DATAEND	CMDSENT	CMDREND	RXOVERR	TXUNDERR	DTIMEOUT	CTIMEOUT	DCRCFAIL	CCRCFAIL
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

位31:24	保留，始终读为0。
位23	**CEATAEND**：在CMD61接收到CE-ATA命令完成信号 (CE-ATA command completion signal received for CMD61)
位22	**SDIOIT**：收到SDIO中断 (SDIO interrupt received)
位21	**RXDVAL**：在接收FIFO中的数据可用 (Data available in receive FIFO)
位20	**TXDVAL**：在发送FIFO中的数据可用 (Data available in transmit FIFO)
位19	**RXFIFOE**：接收FIFO空 (Receive FIFO empty)
位18	**TXFIFOE**：发送FIFO空 (Transmit FIFO empty) 若使用了硬件流控制，当FIFO包含2个字时，TXFIFOE信号变为有效。
位17	**RXFIFOF**：接收FIFO满 (Receive FIFO full) 若使用了硬件流控制，当FIFO还差2个字满时，RXFIFOF信号变为有效。
位16	**TXFIFOF**：发送FIFO满 (Transmit FIFO full)
位15	**RXFIFOHF**：接收FIFO半满 (Receive FIFO half full)：FIFO中至少还有8个字。
位14	**TXFIFOHE**：发送FIFO半空 (Transmit FIFO half empty)：FIFO中至少还可以写入8个字。
位13	**RXACT**：正在接收数据 (Data receive in progress)
位12	**TXACT**：正在发送数据 (Data transmit in progress)
位11	**CMDACT**：正在传输命令 (Command transfer in progress)
位10	**DBCKEND**：已发送/接收数据块(CRC检测成功) (Data block sent/received (CRC check passed))
位9	**STBITERR**：在宽总线模式，没有在所有数据信号上检测到起始位 (Start bit not detected on all data signals in wide bus mode)
位8	**DATAEND**：数据结束(数据计数器，SDIO_DCOUNT = 0) (Data end (data counter, SDIDCOUNT, is zero))
位7	**CMDSENT**：命令已发送(不需要响应) (Command sent (no response required))
位6	**CMDREND**：已接收到响应(CRC检测成功) (Command response)
位5	**RXOVERR**：接收FIFO上溢错误 (Received FIFO overrun error)
位4	**TXUNDERR**：发送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) 命令超时时间是一个固定的值，为64个SDIO_CK时钟周期。
位1	**DCRCFAIL**：已发送/接收数据块(CRC检测失败) (Data block sent/received)
位0	**CCRCFAIL**：已收到命令响应(CRC检测失败) (Command response received)

图 40.1.4.5 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《STM32 中文参考手册》第 20 章相关章节。

40.1.5 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 40.1.5.1 所示：

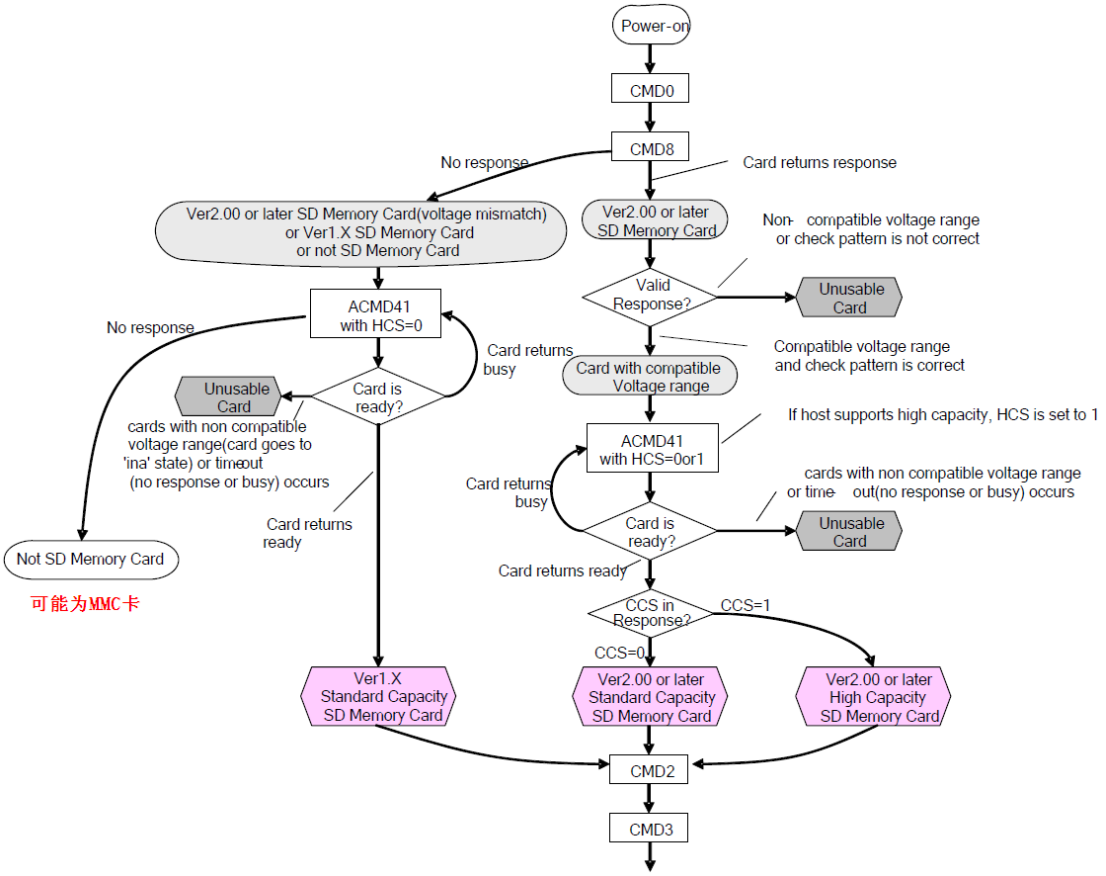


图 40.1.5.1 SD 卡初始化流程

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 40.1.5.1 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 40.1.5.1 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 40.1.5.2 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 40.1.5.2 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 40.1.5.3 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 40.1.5.3 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 40.1.5.4 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 40.1.5.4 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 40.1.5.1 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 40.1.5.5 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 40.1.5.5 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。

40.2 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，这里我们介绍一下精英 STM32F103 板载的 SD 卡接口和 STM32 的连接关系，如图 40.2.1 所示：

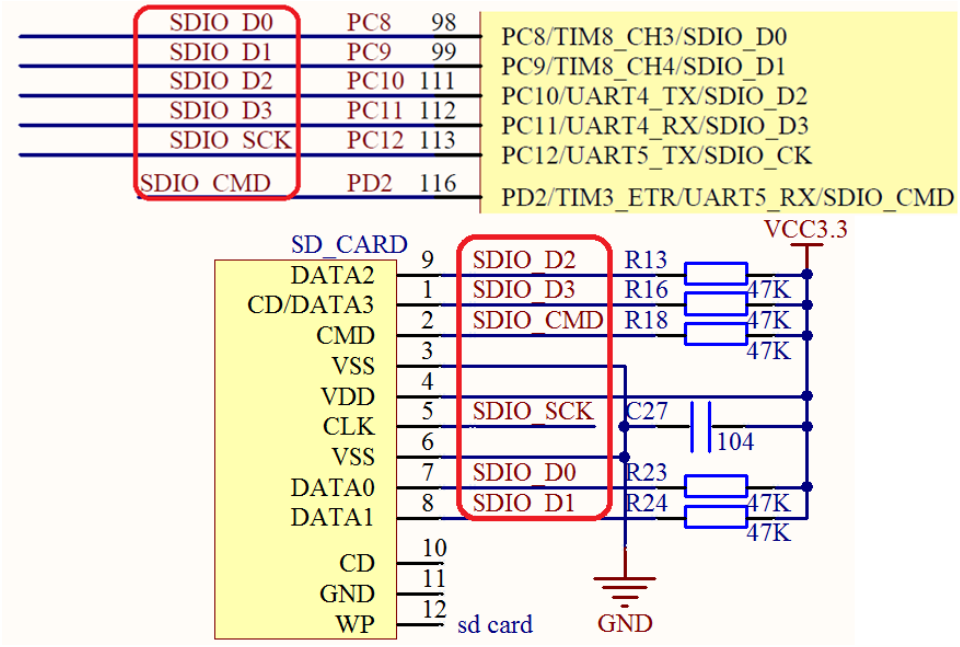


图40.2.1 SD卡接口与STM32F1连接原理图

精英STM32F103的SD卡座（SD_CARD），在PCB背面，SD卡座与STM32F1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。注意，精英板使用的是大的SD卡座，所以一般手机用的TF卡，是不可以直接插精英板用的，但是可以通过TF卡+卡套（需自备）的方式，插精英板上使用。

40.3 软件设计

打开本章实验工程可以看到，我们不但增加了 HAL 库 SDIO 支持文件 stm32f1xx_hal_sdio.c 以及头文件 stm32f1xx_hal_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：

```
//SD 卡初始化
//返回值:0 初始化正确；其他值，初始化错误
u8 SD_Init(void)
{
    u8 SD_Error;
    //初始化时的时钟不能大于 400KHZ
```

```
    SDCARD_Handler.Instance=SDIO;
    SDCARD_Handler.Init.ClockEdge=SDIO_CLOCK_EDGE_RISING;      //上升沿
    SDCARD_Handler.Init.ClockBypass=SDIO_CLOCK_BYPASS_DISABLE;
    //不使用 bypass 模式，直接用 HCLK 进行分频得到 SDIO_CK
    SDCARD_Handler.Init.ClockPowerSave=SDIO_CLOCK_POWER_SAVE_DISABLE;
    //空闲时不关闭时钟电源
    SDCARD_Handler.Init.BusWide=SDIO_BUS_WIDE_1B;                  //1 位数据线
    SDCARD_Handler.Init.HardwareFlowControl=
        SDIO_HARDWARE_FLOW_CONTROL_ENABLE;//开启硬件流控
    SDCARD_Handler.Init.ClockDiv=SDIO_TRANSFER_CLK_DIV;
                                                    //SD 传输时钟频率最大 25MHZ
    SD_Error=HAL_SD_Init(&SDCARD_Handler);
    if(SD_Error!=HAL_OK) return 1;
    SD_Error=HAL_SD_ConfigWideBusOperation(&SDCARD_Handler,
                                        SDIO_BUS_WIDE_4B);//使能宽总线模式
    if(SD_Error!=HAL_OK) return 2;
    return 0;
}
```

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 节有详细介绍了。首先，通过 HAL_SD_Init 函数（该函数我们这里不做介绍，请参考本例程源码），完成 SD 卡的初始化。

限于篇幅，余下函数我们就不一一介绍了，大家可以参本实验考源代码。关于控制命令，如果有不了解的，请参考《SD 卡 2.0 协议.pdf》里面都有非常详细的介绍。

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,u32 cnt)
{
    u8 sta=HAL_OK;
    u32 timeout=SD_TIMEOUT;
    long long lsector=sector;
    INTX_DISABLE();//关闭总中断(POLLING 模式,严禁中断打断 SDIO 读写操作!!!)
    sta=HAL_SD_ReadBlocks(&SDCARD_Handler,
                          (uint8_t*)buf,lsector,cnt,SD_TIMEOUT);//多个 sector 的读操作
    //等待 SD 卡读完
    while(SD_GetCardState()!=SD_TRANSFER_OK)
    {
        if(timeout-- == 0)
        {
```

```
                sta=SD_TRANSFER_BUSY;
            }
        }
        INTX_ENABLE();//开启总中断
        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,u32 cnt)
    {
        u8 sta=HAL_OK;
        u32 timeout=SD_TIMEOUT;
        long long lsector=sector;
        INTX_DISABLE();//关闭总中断(POLLING 模式,严禁中断打断 SDIO 读写操作!!!)
        sta=HAL_SD_WriteBlocks(&SDCARD_Handler,(uint8_t*)buf,lsector,cnt,SD_TIMEOUT);
                                                        //多个 sector 的写操作
        //等待 SD 卡写完
        while(SD_GetCardState()!=SD_TRANSFER_OK)
        {
            if(timeout-- == 0)
            {
                sta=SD_TRANSFER_BUSY;
            }
        }
        INTX_ENABLE();//开启总中断
        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 HAL_SD_ReadBlocks 实现。SD_WriteDisk 用于写数据，通过调用 HAL_SD_WriteBlocks 实现。

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)
    {
        uint64_t CardCap;  //SD 卡容量
        HAL_SD_CardCIDTypeDef SDCard_CID;
        HAL_SD_GetCardCID(&SDCARD_Handler,&SDCard_CID);  //获取 CID
        SD_GetCardInfo(&SDCardInfo);                     //获取 SD 卡信息
```

```
    switch(SDCardInfo.CardType)
    {
        case CARD_SDSC:
        {
            if(SDCardInfo.CardVersion == CARD_V1_X)
                printf("Card Type:SDSC V1\r\n");
            else if(SDCardInfo.CardVersion == CARD_V2_X)
                printf("Card Type:SDSC V2\r\n");
        }
        break;
        case CARD_SDHC_SDXC:printf("Card Type:SDHC\r\n");break;
    }
    CardCap=(uint64_t)(SDCardInfo.LogBlockNbr)*(uint64_t)(SDCardInfo.LogBlockSize);
                                                        //计算 SD 卡容量
    printf("Card ManufacturerID:%d\r\n",SDCard_CID.ManufacturerID);//制造商 ID
    printf("Card RCA:%d\r\n",SDCardInfo.RelCardAdd);            //卡相对地址
    printf("LogBlockNbr:%d \r\n",(u32)(SDCardInfo.LogBlockNbr));    //显示逻辑块数量
    printf("LogBlockSize:%d \r\n",(u32)(SDCardInfo.LogBlockSize));    //显示逻辑块大小
    printf("Card Capacity:%d MB\r\n",(u32)(CardCap>>20));        //显示容量
    printf("Card BlockSize:%d\r\n\r\n",SDCardInfo.BlockSize);    //显示块大小
}

//测试 SD 卡的读取
//从 secaddr 地址开始,读取 seccnt 个扇区的数据
//secaddr:扇区地址
//seccnt:扇区数
void sd_test_read(u32 secaddr,u32 seccnt)
{
    u32 i; u8 *buf; u8 sta=0;
    buf=mymalloc(SRAMEX,seccnt*512); //申请内存
    sta=SD_ReadDisk(buf,secaddr,seccnt);//读取 secaddr 扇区开始的内容
    if(sta==0)
    {
        printf("SECTOR %d DATA:\r\n",secaddr);
        for(i=0;i<seccnt*512;i++)printf("%x ",buf[i]);//打印 secaddr 开始的扇区数据
        printf("\r\nDATA ENDED\r\n");
    }else printf("err:%d\r\n",sta);
    myfree(SRAMEX,buf); //释放内存
}

//测试 SD 卡的写入(慎用,最好写全是 0XFF 的扇区,否则可能损坏 SD 卡.)
//从 secaddr 地址开始,写入 seccnt 个扇区的数据
//secaddr:扇区地址
```

```
//seccnt:扇区数
void sd_test_write(u32 secaddr,u32 seccnt)
{
    u32 i;
    u8 *buf; u8 sta=0;
    buf=mymalloc(SRAMEX,seccnt*512);        //从 SRAM 申请内存
    for(i=0;i<seccnt*512;i++) buf[i]=i*3;   //初始化写入的数据,是 3 的倍数.
    sta=SD_WriteDisk(buf,secaddr,seccnt);//从 secaddr 扇区开始写入 seccnt 个扇区内容
    if(sta==0) printf("Write over!\r\n");
    else printf("err:%d\r\n",sta);
    myfree(SRAMEX,buf);                         //释放内存
}

int main(void)
{
    u8 key;
    u32 sd_size;
    u8 t=0;
    u8 *buf;
    uint64_t CardCap;                           //SD 卡容量
    HAL_Init();                                 //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);             //设置时钟,72M
    delay_init(72);                             //初始化延时函数
    uart_init(115200);                          //初始化串口
    LED_Init();                                 //初始化 LED
    KEY_Init();                                 //初始化按键
    LCD_Init();                                 //初始化 LCD FSMC 接口
    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
    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/9/19");
    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");
        CardCap=(uint64_t)(SDCardInfo.LogBlockNbr)*(uint64_t)(SDCardInfo.LogBlockSize);
                                                        //计算 SD 卡容量
        LCD_ShowNum(30+13*8,170,CardCap>>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]);
                                                        //打印 0 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。

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 精英 STM32F103 上，可以看到 LCD

显示如图 40.4.1 所示的内容（假设 SD 卡已经插上了）：

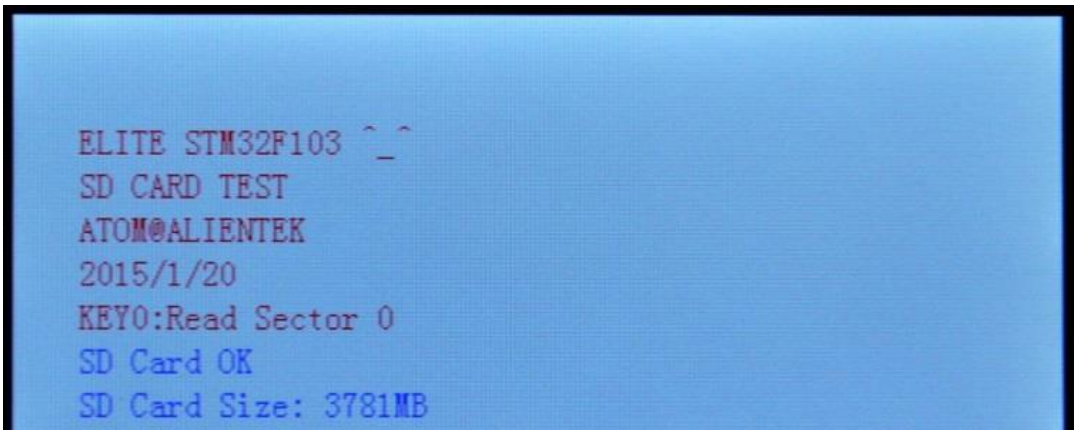


图 40.4.1 程序运行效果图

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 40.4.2 所示：

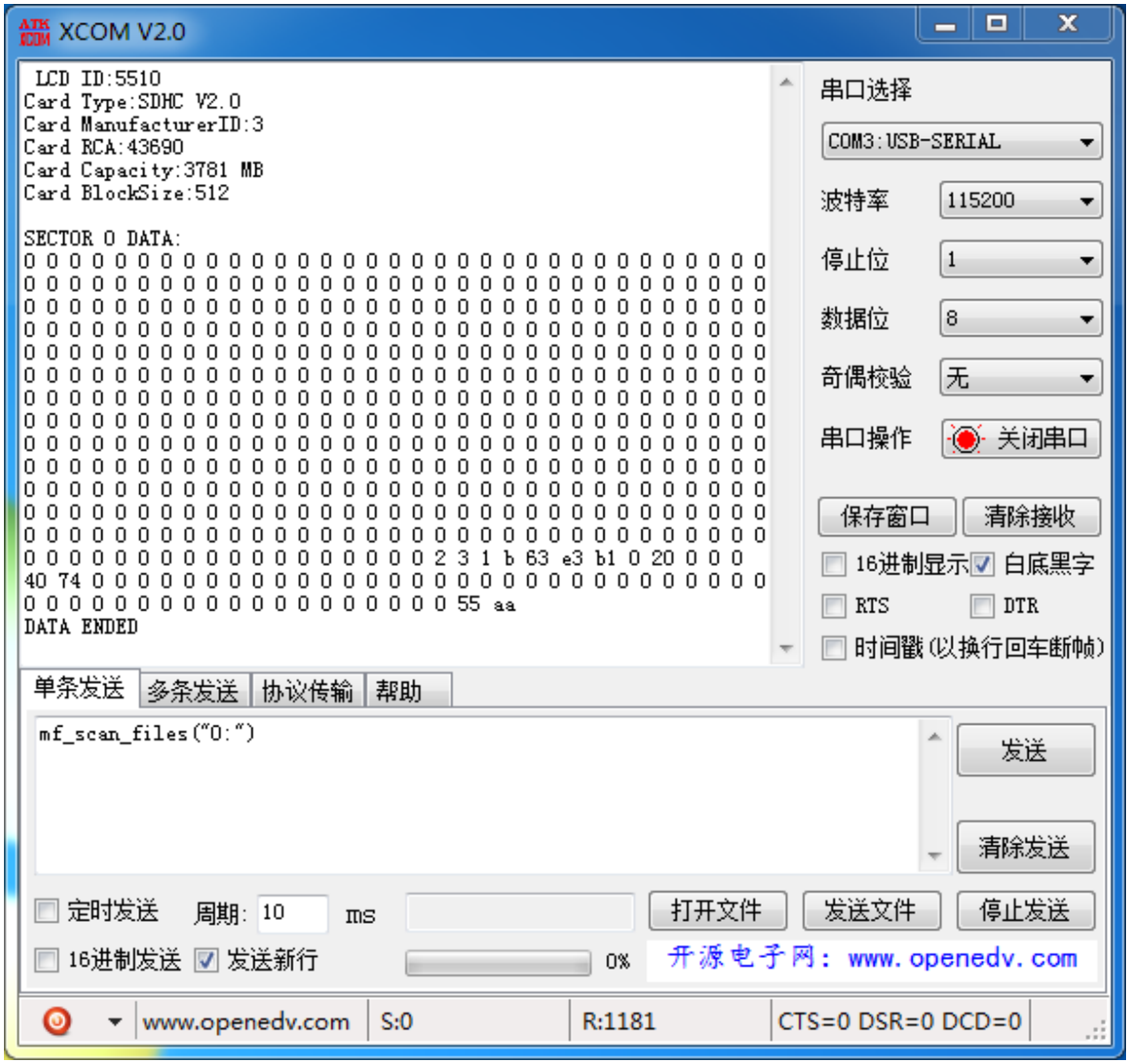


图 40.4.2 串口收到的 SD 卡扇区 0 内容

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 40.4.2 不同而感到惊讶。

第四十一章　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：

41.1 FATFS 简介
41.2 硬件设计
41.3 软件设计
41.4 下载验证

41.1 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 FATl2、FATl6 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 41.1.1 所示：

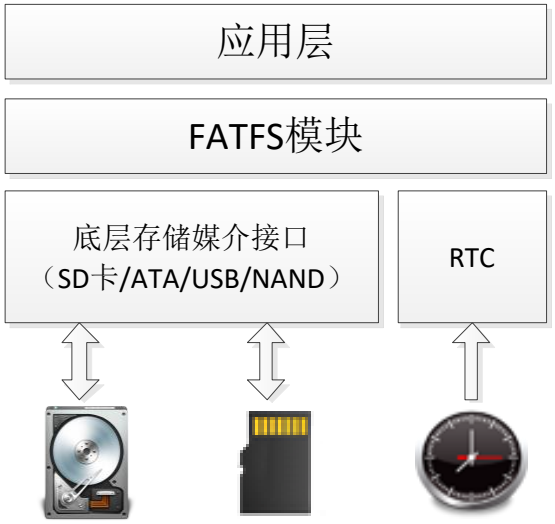


图 41.1.1 FATFS 层次结构图

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 R0.11。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_memalloc 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 41.1.2 所示：

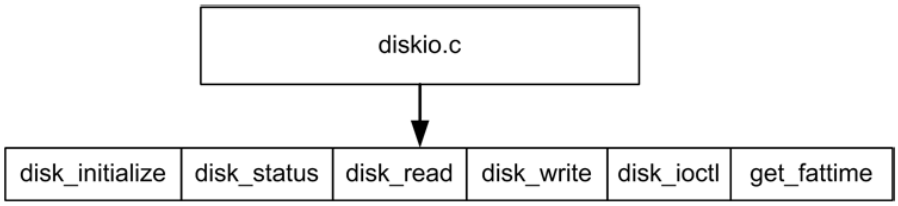


图 41.1.2 diskio 需要实现的函数

通过以下三步，我们即可完成对 FATFS 的移植。

第一步，我们使用的是 MDK5.14 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 41.1.3 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 41.1.3 disk_initialize 函数介绍

第二个函数是 disk_status 函数，该函数介绍如图 41.1.4 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 41.1.4 disk_status 函数介绍

第三个函数是 disk_read 函数，该函数介绍如图 41.1.5 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 41.1.5 disk_read 函数介绍

第四个函数是 disk_write 函数，该函数介绍如图 41.1.6 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	Drive： 指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：指向要写入字节数组的指针， 注：FatFs 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理 SectorNumber： 指定起始扇区的逻辑块（LBA）上的地址 SectorNumber： 指定要写入的扇区数，取值 1~128
返回值	RES_OK(0)： 函数成功 RES_ERROR： 读操作期间产生了任何错误且不能恢复它 RES_WRPRT： 媒体被写保护 RES_PARERR： 非法参数 RES_NOTRDY：磁盘驱动器没有初始化
所在文件	ff.c
注意事项	只读配置中不需要此函数

图 41.1.6 disk_write 函数介绍

第五个函数是 disk_ioctl 函数，该函数介绍如图 41.1.7 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	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
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
注意事项	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

图 41.1.7 disk_ioctl 函数介绍

最后一个函数是 get_fattime 函数，该函数介绍如图 41.1.8 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 41.1.8 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。

41.2 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。

41.3 软件设计

本章，我们将 FATFS 部分单独做一个分组，在工程目录下新建一个 FATFS 的文件夹，然后将 FATFS R0.11 程序包解压到该文件夹下。同时，我们在 FATFS 文件夹里面新建一个 exfuns 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 41.3.1 所示：

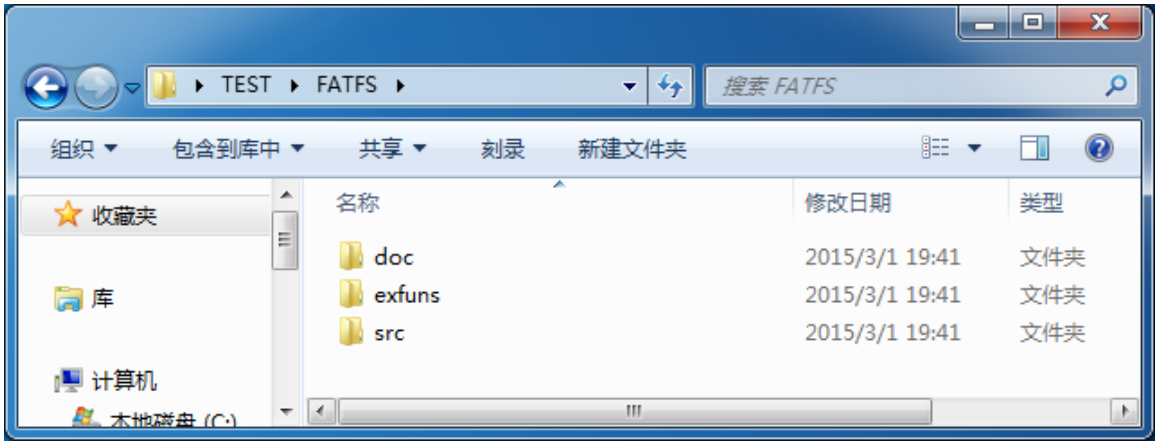


图 41.3.1 FATFS 文件夹子目录

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 3.09M.剩余的,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
//获得磁盘状态
DSTATUS disk_status (
    BYTE pdrv                   /* Physical drive nmuber to identify the drive */
)
{
    return RES_OK;
}
//初始化磁盘
DSTATUS disk_initialize (
    BYTE pdrv                   /* Physical drive nmuber to identify the drive */
)
{
    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
}
//读扇区
//pdrv: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 to identify the drive */
        BYTE *buff,                     /* Data buffer to store read data */
        DWORD sector,                   /* Sector address in LBA */
        UINT count                      /* Number of sectors to read */
)
{
        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);
                while(res)//读出错
                {
                    SD_Init();      //重新初始化 SD 卡
                    res=SD_ReadDisk(buff,sector,count);
                    //printf("sd rd error:%d\r\n",res);
                }
                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
//pdrv: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 to identify the drive */
    const BYTE *buff,   /* Data to be written */
    DWORD sector,       /* Sector address in LBA */
    UINT count          /* Number of sectors to write */
)
{
    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);
            while(res)          //写出错
            {
                SD_Init();      //重新初始化 SD 卡
                res=SD_WriteDisk((u8*)buff,sector,count);
                //printf("sd wr error:%d\r\n",res);
            }
            break;
        case EX_FLASH://外部 flash
            for(;count>0;count--)
            {
                W25QXX_Write((u8*)buff,sector*FLASH_SECTOR_SIZE,
                             FLASH_SECTOR_SIZE);
                sector++;
                buff+=FLASH_SECTOR_SIZE;
            }
            res=0;
            break;
        default: res=1;
    }
```

```
    //处理返回值，将 SPI_SD_driver.c 的返回值转成 ff.c 的返回值
    if(res == 0x00)return RES_OK;
    else return RES_ERROR;
}
#endif
//其他表参数的获得
//pdrv: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.LogBlockSize;
                res = RES_OK;
                break;
            case GET_SECTOR_COUNT:
                *(DWORD*)buff = SDCardInfo.LogBlockNbr;
                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;
                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_memalloc (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 41.1 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_memalloc 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。

另外，diskio.c 里面的函数，直接决定了磁盘编号（盘符/卷标）所对应的具体设备，比如，以上代码中，我们设置 SD_CARD 为 0，EX_FLASH 位为 1，对应到 disk_read/disk_write 函数里面，我们就通过 switch 来判断，到底要操作 SD 卡，还是 SPI FLASH，然后，分别执行对应设备的相关操作。以此实现磁盘编号和磁盘的关联。

保存 diskio.c，然后打开 ffconf.h，修改相关配置，并保存，此部分就不贴代码了，请大家参考本例程源码。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 exfuns 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：exfuns.c、exfuns.h、fattester.c 和 fattester.h。其中 exfuns.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，我们将 exfuns.c 和 fattester.c 加入 FATFS 组下，同时将 exfuns 文件夹加入头文件包含路径。

然后，我们打开 test.c，修改 main 函数如下：

```
int main(void)
{
    u32 total,free;
    u8 t=0; u8 res=0;
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    usmart_dev.init(72);                //初始化 USMART
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    LCD_Init();                         //初始化 LCD FSMC 接口
    SRAM_Init();                        //初始化外部 SRAM
    my_mem_init(SRAMIN);                //初始化内部内存池
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"ELITE STM32 ^_^");
    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,"2019/9/28");
    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
    }
    exfuns_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 Flash 磁盘名：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);//清除显示
        delay_ms(200);
        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。

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 精英 STM32F103 上，可以看到 LCD 显示如图 41.4.1 所示的内容（假定 SD 卡已经插上了）：

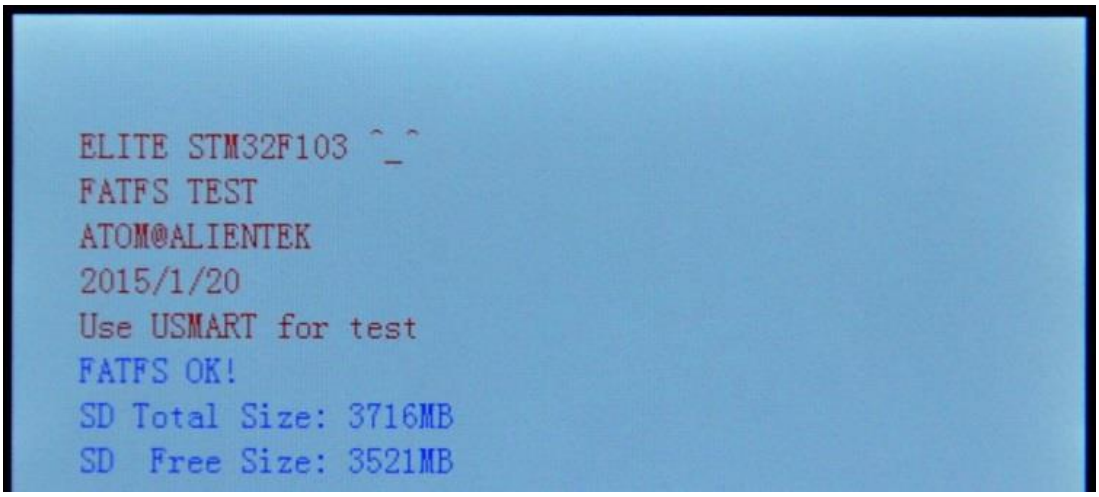


图 41.4.1 程序运行效果图

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 41.4.2 所示：

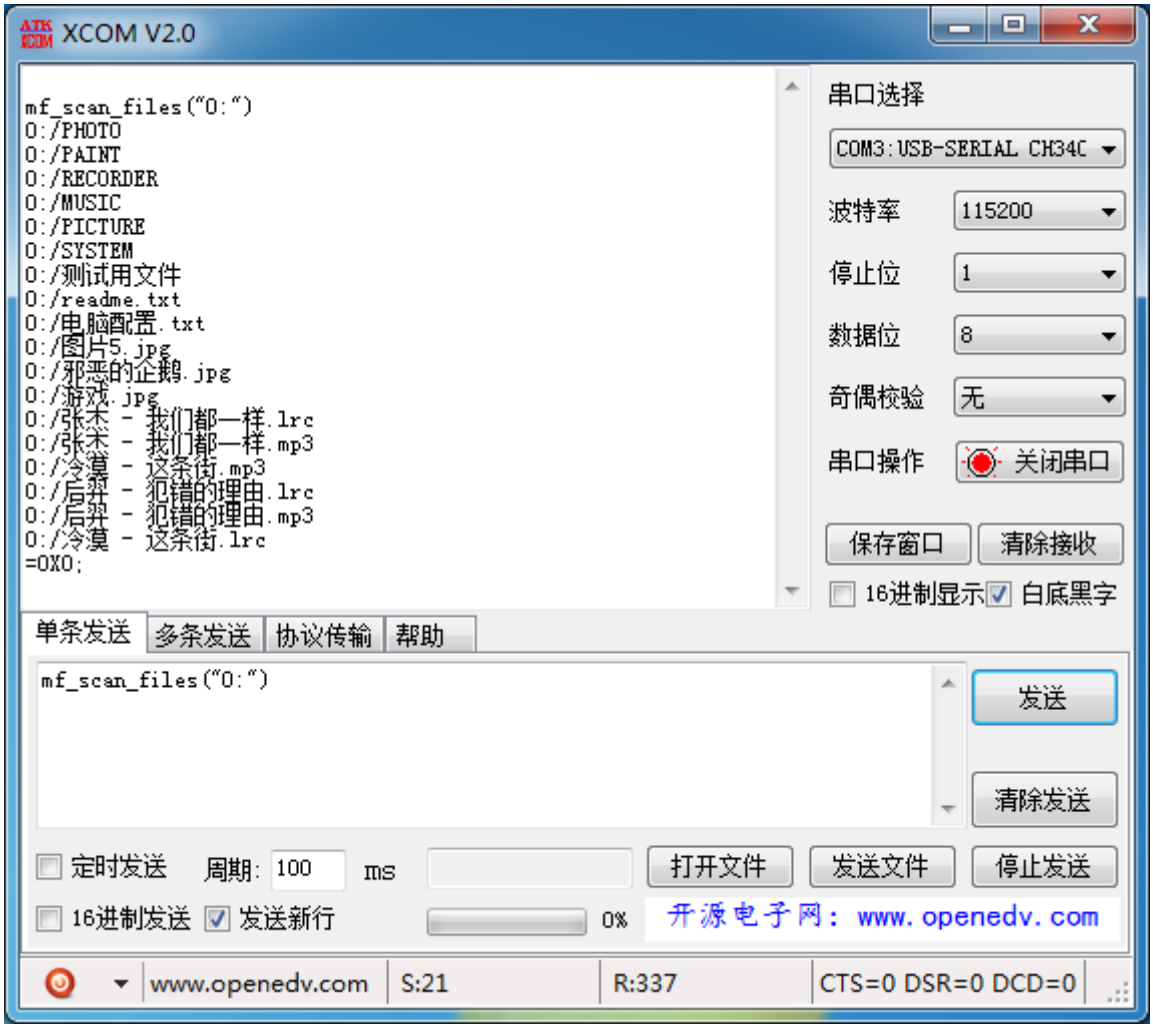


图 41.4.2 扫描 SD 卡根目录所有文件

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。

第四十二章 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 STM32F1 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。STM32F1 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：

42.1 汉字显示原理简介
42.2 硬件设计
42.3 软件设计
42.4 下载验证

42.1 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*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 GBKL<0X7F 时：Hp=((GBKH-0x81)*190+GBKL-0X40)*(size*2)；

当 GBKL>0X80 时：Hp=((GBKH-0x81)*190+GBKL-0X41)*(size*2)；

其中 GBKH、GBKL 分别代表 GBK 的第一个字节和第二个字节(也就是高位和低位)，size 代表汉字字体的大小（比如 16 字体，12 字体等），Hp 则为对应汉字点阵数据在字库里面的起始地址(假设是从 0 开始存放)。

这样我们只要得到了汉字的 GBK 码，就可以显示这个汉字了。从而实现汉字在液晶上的显示。

上一章，我们提到要用 cc936.c，以支持长文件名，但是 cc936.c 文件里面的两个数组太大了（172KB），直接刷在单片机里面，太占用 flash 了，所以我们必须把这两个数组存放在外部 flash。cc936 里面包含的两个数组 oem2uni 和 uni2oem 存放 unicode 和 gbk 的互相转换对照表，这两个数组很大，这里我们利用 ALIENTEK 提供的一个 C 语言数组转 BIN（二进制）的软件：C2B 转换助手 V1.1.exe，将这两个数组转为 BIN 文件，我们将这两个数组拷贝出来存放为一个新的文本文件，假设为 UNIGBK.TXT，然后用 C2B 转换助手打开这个文本文件，如图 42.1.1 所示：



图 42.1.1 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 42.1.2 所示：



图 42.1.2 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 42.1.3 所示：

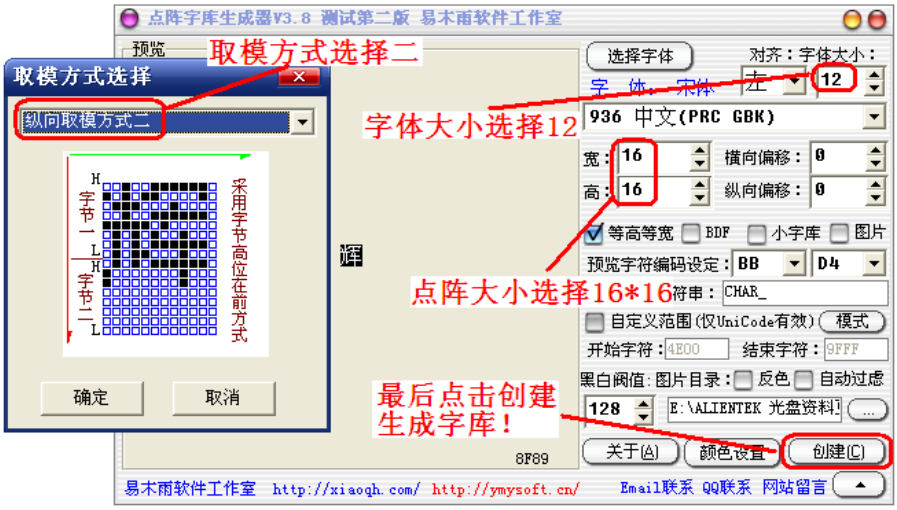


图 42.1.3 生成 GBK16*16 字库的设置方法

注意：电脑端的字体大小与我们生成点阵大小的关系为：

fsize=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（这里是手动修改后缀!!）**。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。

42.2 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。

42.3 软件设计

打开上一章的工程，首先在 HARDWARE 文件夹所在的文件夹下新建一个 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。并将该文件夹加入头文件包含路径。

打开 fontupd.c，在该文件内输入如下代码：

```
//字库区域占用的总扇区数大小(字库信息+ unigbk 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
//精英 STM32F103，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 UNIGBK_PATH="/SYSTEM/FONT/UNIGBK.BIN"; //UNIGBK.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,ungbk;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;
    u16 bread;
    u32 offx=0;
    u8 rval=0;
    fftemp=(FIL*)mymalloc(SRAMIN,sizeof(FIL));  //分配内存
    if(fftemp==NULL)rval=1;
    tempbuf=mymalloc(SRAMIN,4096);                  //分配 4096 个字节空间
    if(tempbuf==NULL)rval=1;
    res=f_open(fftemp,(const TCHAR*)fxpath,FA_READ);
    if(res)rval=2;//打开文件失败
    if(rval==0)
    {
        switch(fx)
        {
            case 0:             //更新 UNIGBK.BIN
                ftinfo.ugbkaddr=FONTINFOADDR+sizeof(ftinfo);//紧跟 UNIGBK 码表
                ftinfo.ugbksize=fftemp->fsize;              //UNIGBK 大小
                flashaddr=ftinfo.ugbkaddr;
                break;
            case 1:
                ftinfo.f12addr=ftinfo.ugbkaddr+ftinfo.ugbksize;  //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;  //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;  //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 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,"
//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;
        u32 *buf;
        u16 i,j;
        FIL *fftemp;
        u8 rval=0;
        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);//读扇区
        for(j=0;j<1024;j++)//校验数据
        {
            if(buf[j]!=0XFFFFFFFF)break;//需要擦除
        }
        if(j!=1024)W25QXX_Erase_Sector((FONTINFOADDR/4096)+i);//要擦除扇区
    }
    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,说明确实是有问题,得更新字库了
        {
            t++;
            W25QXX_Read((u8*)&ftinfo,FONTINFOADDR,sizeof(ftinfo));//ftinfo 结构体数据
            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。

保存该部分代码，并在工程里面新建一个 TEXT 的组，把 fontupd.c 加入到这个组里面，然后打开 fontupd.h 在该文件里面输入如下代码：

```
#ifndef __FONTUPD_H__
#define __FONTUPD_H__
#include <stm32f10x.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 ugbkaddr;       //unigbk 的地址
    u32 ugbksize;       //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 gkb24size;      //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 *fxpath,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。保存此部分代码，然后打开 text.c 文件，在该文件里面输入如下代码：

```
//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 foffset;
    u8 csize=(size/8+((size%8)?1:0))*(size);    //得到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;
    foffset=((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 42.1 节介绍的办法，在外部 flash 查找字库，然后返回对应的字库点阵。Show_Font 函数用于在指定地址显示一个指定大小的汉字，采用的方法和 LCD_ShowChar 所采用的方法一样，都是画点显示，这里就不细说了。保存此部分代码，并把 text.c 文件加入 TEXT 组下。text.h 里面都是一些函数申明，这里我们就不贴出来了，详见光盘本例程源码。

前面提到我们对 cc936.c 文件做了修改，我们将其命名为 mycc936.c，并保存在 exfuns 文件夹下，将工程 FATFS 组下的 cc936.c 删除，然后重新添加 mycc936.c 到 FATFS 组下，mycc936.c 的源码就不贴出来了，其实就是在 cc936.c 的基础上去掉了两个大数组，然后对 ff_convert 进行了修改，详见本例程源码。

最后，我们在 test.c 里面修改 main 函数如下：

```
int main(void)
{
    u32 fontcnt; u8 i,j; u8 key,t;
    u8 fontx[2];                        //gbk 码
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    usmart_dev.init(72);                //初始化 USMART
```

```
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    LCD_Init();                             //初始化 LCD FSMC 接口
    SRAM_Init();                            //初始化外部 SRAM
    my_mem_init(SRAMIN);                    //初始化内部内存池
    exfuns_init();                          //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1);                  //挂载 SD 卡
    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,"ELITE STM32F103 ^_^");
        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,"精英 STM32F103 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," 2019 年 9 月 28 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。

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 精英 STM32F103 上，可以看到 LCD 开始显示汉字及汉字内码，如图 42.4.1 所示：

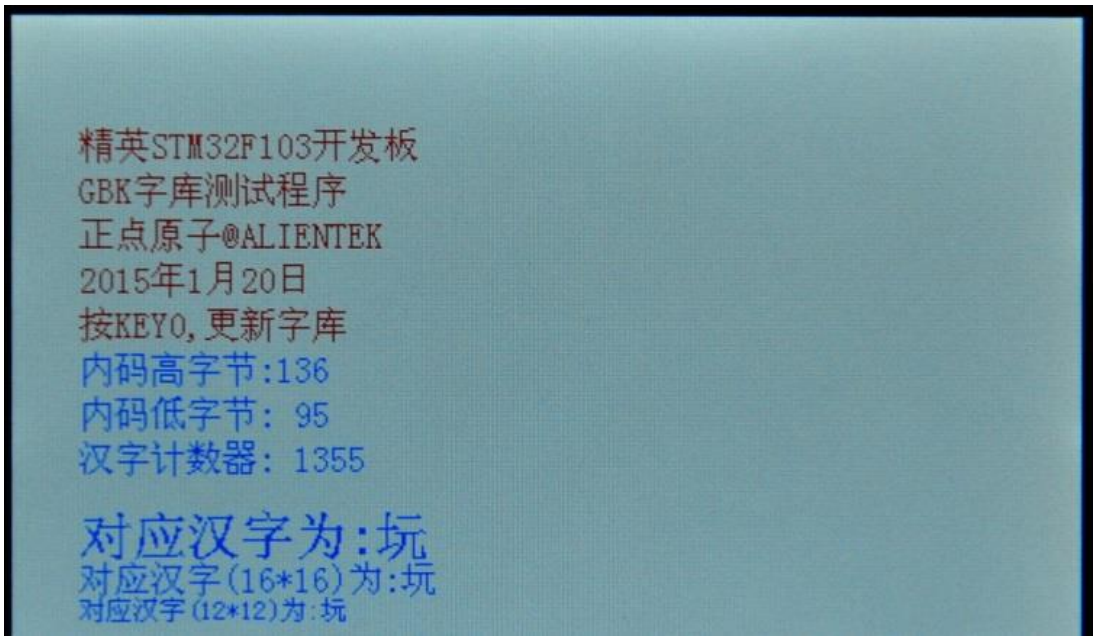


图 42.4.1 汉字显示实验显示效果

一开始就显示汉字，是因为 ALIENTEK 精英 STM32F103 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 42.4.2 所示：

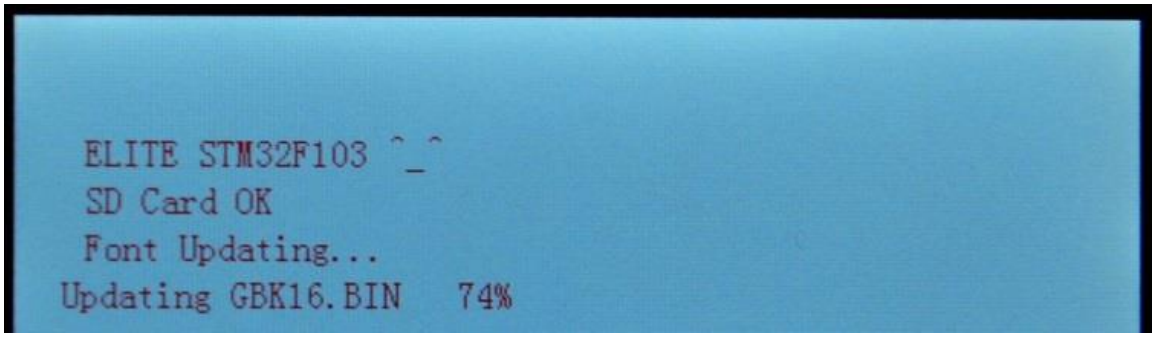


图 42.4.2 汉字字库更新界面

我们还可以通过 USMART 来测试该实验，将 Show_Str 函数加入 USMART 控制（方法前面已经讲了很多次了），就可以通过串口调用该函数，在屏幕上显示任何你想要显示的汉字了，有兴趣的朋友可以测试一下。

第四十三章 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 STM32F1 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：

43.1 图片格式简介
43.2 硬件设计
43.3 软件设计
43.4 下载验证

43.1 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 lbit、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，文件后辍名为“. 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，把其恢复成 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 Walch)压缩算法来存储图像数据，定义了允许用户为图像设置背景的透明(transparency)属性。此外，GIF 文件格式可在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。

43.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY1 可以快速浏览下一张和上一张，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、KEY1 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。

43.3 软件设计

打开上一章的工程，首先在 HARDWARE 文件夹所在的文件夹下新建一个 PICTURE 的文件夹。在该文件夹里面新建 bmp.c、bmp.h、tjpgd.c、tjpgd.h、integer.h、gif.c、gif.h、piclib.c 和 piclib.h 等 9 个文件。并将 PICTURE 文件夹加入头文件包含路径。

其中 bmp.c 和 bmp.h 用于实现对 bmp 文件的解码；tjpgd.c 和 tjpgd.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，在里面输入如下代码：

```
_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*picinfo.ImgWidth)/2;
    picinfo.S_YOFF+=(picinfo.S_Height-temp1*picinfo.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); break;                //解码 bmp
```

```
            case T_JPG:
            case T_JPEG: res=jpg_decode(filename,fast); break;                  //解码 JPG/JPEG
            case T_GIF: res=gif_decode(filename,x,y,width,height); break;  //解码 gif
            default: res=PIC_FORMAT_ERR; break;                                 //非图片格式!!!
        }
        return res;
}
//动态分配内存
void *pic_memalloc (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 exfuns.c 里面实现，具体请参考光盘本例程源码。

最后，pic_memalloc 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。

保存 piclib.c，然后在工程里面新建一个 PICTURE 的分组，将 bmp.c、gif.c、tjpgd.c 和 piclib.c 等 4 个 c 文件加入到 PICTURE 分组下。然后打开 piclib.h，在该文件输入如下代码：

```
#ifndef __PICLIB_H
#define __PICLIB_H
#include "sys.h"
#include "lcd.h"
#include "malloc.h"
#include "ff.h"
```

```
#include "exfuns.h"
#include "bmp.h"
#include "tjpgd.h"
#include "gif.h"
#define PIC_FORMAT_ERR      0x27    //格式错误
#define PIC_SIZE_ERR        0x28    //图片尺寸错误
#define PIC_WINDOW_ERR      0x29    //窗口设定错误
#define PIC_MEM_ERR         0x11    //内存错误
#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,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,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_memalloc (u32 size);     //pic 申请内存
void pic_memfree (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，保存 piclib.h。最后我们在 test.c 文件里面修改代码如下：

```
//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_typetell(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
HAL_Init();                         //初始化 HAL 库
Stm32_Clock_Init(RCC_PLL_MUL9); //设置时钟,72M
delay_init(72);                     //初始化延时函数
uart_init(115200);                  //初始化串口
LED_Init();                         //初始化 LED
KEY_Init();                         //初始化按键
LCD_Init();                         //初始化 LCD FSMC 接口
SRAM_Init();                        //初始化外部 SRAM
my_mem_init(SRAMIN);                //初始化内部内存池
exfuns_init();                      //为 fatfs 相关变量申请内存
f_mount(fs[0],"0:",1);              //挂载 SD 卡
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);//清除显示
    delay_ms(200);
}
Show_Str(30,50,200,16,"ELITE STM32F1 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 KEY1:PREV",16,0);
Show_Str(30,110,200,16,"KEY_UP:PAUSE",16,0);
Show_Str(30,130,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(30,150,200,16,"2019 年 9 月 28 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);//清除显示
    delay_ms(200);
}
totpicnum=pic_get_tnum("0:/PICTURE"); //得到总有效文件数
while(totpicnum==NULL)//图片文件为 0
{
    Show_Str(30,170,240,16,"没有图片文件!",16,0);
    delay_ms(200);
```

```
            LCD_Fill(30,170,240,186,WHITE);//清除显示
            delay_ms(200);
        }
        picfileinfo=(FILINFO*)mymalloc(SRAMIN,sizeof(FILINFO)); //申请内存
        pname=mymalloc(SRAMIN,_MAX_LFN*2+1);                    //为带路径的文件名分配内存
        picoffsettbl=mymalloc(SRAMIN,4*totpicnum);              //申请 4*totpicnum 个字节的内存,用于存放图片索引
        while(!picfileinfo||!pname||!picoffsettbl)              //内存分配出错
        {
            Show_Str(30,170,240,16,"内存分配失败!",16,0);
            delay_ms(200);
            LCD_Fill(30,170,240,186,WHITE);//清除显示
            delay_ms(200);
        }
        //记录索引
        res=f_opendir(&picdir,"0:/PICTURE"); //打开目录
        if(res==FR_OK)
        {
            curindex=0;//当前索引为 0
            while(1)//全部查询一遍
            {
                temp=picdir.dptr;                               //记录当前 dptr 偏移
                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

                res=f_typetell((u8*)picfileinfo->fname);
                if((res&0XF0)==0X50)//取高四位,看看是不是图片文件
                {
                    picoffsettbl[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,picoffsettbl[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
```

```
            strcpy((char*)pname,"0:/PICTURE/");                //复制路径(目录)
            strcat((char*)pname,(const char*)picfileinfo->fname);//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,lcddev.width,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==KEY1_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;        //暂停的时候 LED1 亮.
                }
                if(pause==0)t++;
                delay_ms(10);
            }
            res=0;
        }
        myfree(SRAMIN,picfileinfo);             //释放内存
        myfree(SRAMIN,pname);                   //释放内存
        myfree(SRAMIN,picoffsettbl);            //释放内存
    }
```

此部分除了 mian 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 mian 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。

43.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 精英 STM32F103 上，可以看到 LCD 开始显示图片（假设 SD 卡及文件都准备好了，即：在 SD 卡根目录新建：PICTURE 文件夹，并存放一些图片文件(.bmp/.jpg/.gif)在该文件夹内），如图 43.4.1 所示：



图 43.4.1 图片显示实验显示效果

按 KEY0 和 KEY1 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。

本章，同样可以通过 USMART 来测试该实验，将 ai_load_picfile 函数加入 USMART 控制（方法前面已经讲了很多次了），就可以通过串口调用该函数，在屏幕上任何区域显示任何你想要显示的图片了！同时，可以发送：runtime 1，来开启 USMART 的函数执行时间统计功能，从而获取解码一张图片所需时间，方便验证。

第四十四章 照相机实验

上一章，我们学习了图片解码，本章我们将学习 BMP 编码，结合前面的摄像头实验，实现一个简单的照相机。本章分为如下几个部分：

44.1 BMP 编码简介
44.2 硬件设计
44.3 软件设计
44.4 下载验证

44.1 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 的 32 次方)种颜色。这种位图的结构与 16 位位图结构非常类似，当 biCompression 成员的值是 BI_RGB 时，它也没有调色板，32 位中有 24 位用于存放 RGB 值，顺序是：最高位—保留，红 8 位、绿 8 位、蓝 8 位。这种格式也被成为 888 32 位图。如果 biCompression 成员的值是 BI_BITFIELDS 时，原来调色板的位置将被三个 DWORD 变量占据，成为红、绿、蓝掩码，分别用于描述红、绿、蓝分量在 32 位中所占的位置。在 Windows 95(or 98)中，系统只接受 888 格式，也就是说三个掩码的值将只能是：0xFF0000、

0xFF00、0xFF。而在 NT 系统中，你只要注意使掩码之间不产生重叠就行。（注：这种图像格式比较规整，因为它是 DWORD 对齐的，所以在内存中进行图像处理时可进行汇编级的代码优化（简单））。

通过以上了解，我们对 BMP 有了一个比较深入的了解，本章，我们采用 16 位 BMP 编码（因为我们的 LCD 就是 16 位色的，而且 16 位 BMP 编码比 24 位 BMP 编码更省空间），故我们需要设置 biBitCount 的值为 16，这样得到新的位图信息（BITMAPINFO）结构体：

```
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    u32 RGB_MASK[3];            //调色板用于存放 RGB 掩码.
}BITMAPINFO;
```

其实就是颜色表由 3 个 RGB 掩码代替。最后，我们来看看将 LCD 的显存保存为 BMP 格式的图片文件的步骤：

1）创建 BMP 位图信息，并初始化各个相关信息

这里，我们要设置 BMP 图片的分辨率为 LCD 分辨率、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。

44.2 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，开始检测和初始化 OV7725。对于 OV7725：初始化成功后需先按 KEY0 和 KEY1 选择图像输出模式，选择后即显示拍摄画面。

在正常显示拍摄画面后，按下 KEY0 按键即可进行拍照，此时 DS1 亮，拍照保存成功之后，蜂鸣器会发出“滴”的一声，提示拍照成功，同时 DS1 灭。DS0 还是用于指示程序运行状态。

所要用到的硬件资源如下：

1） 指示灯 DS0 和 DS1
2） KEY0 按键
3） 蜂鸣器
4） 串口
5） TFTLCD 模块
6） SD 卡
7） SPI FLASH

8） OV7725 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。

44.3 软件设计

打开上一章的工程，由于本章要用到摄像头模块、蜂鸣器、外部中断和定时器等外设，所以复制 OV7725 文件和添加 beep.c、exti.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));        // 申请到的内存置零.
    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);
    }
#if 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 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。

保存 bmp.c，然后在 bmp.h 里面添加 bmp_encode 函数的申明，并保存 bmp.h 文件。

剩下的，我们就只需要修改主函数即可了，打开 main.c，修改该文件代码如下：

```
extern u8 ov_sta;          //在 exit.c 里面定义
extern u8 ov_frame;        //在 timer.c 里面定义
//更新 LCD 显示
void camera_refresh(void)
{
    u32 j;
    u16 color;
    if(ov_sta)//有帧中断更新？
    {
        LCD_Scan_Dir(U2D_L2R);          //从上到下,从左到右
        if(lcddev.id==0X1963)LCD_Set_Window((lcddev.width-240)/2,
                    (lcddev.height-320)/2,240,320);//将显示区域设置到屏幕中央
        else if(lcddev.id==0X5510||lcddev.id==0X5310)
        LCD_Set_Window((lcddev.width-320)/2,(lcddev.height-240)/2,320,240);
                                                    //将显示区域设置到屏幕中央
        LCD_WriteRAM_Prepare();         //开始写入 GRAM
        OV7725_RRST=0;                  //开始复位读指针
        OV7725_RCK_L;
        OV7725_RCK_H;
        OV7725_RCK_L;
        OV7725_RRST=1;                  //复位读指针结束
        OV7725_RCK_H;
        for(j=0;j<76800;j++)            //240*320
        {
            OV7725_RCK_L;
            color=GPIOC->IDR&0XFF; //读数据
            OV7725_RCK_H;
            color<<=8;
            OV7725_RCK_L;
            color|=GPIOC->IDR&0XFF; //读数据
            OV7725_RCK_H;
            LCD->LCD_RAM=color;
        }
        ov_sta=0;                       //清零帧中断标记
        ov_frame++;
        LCD_Scan_Dir(DFT_SCAN_DIR);     //恢复默认扫描方向
    }
}
```

```
//文件名自增（避免覆盖）
//组合成:形如"0:PHOTO/PIC13141.bmp"的文件名
void camera_new_pathname(u8 *pname)
{
    u8 res;
    u16 index=0;
    while(index<0XFFFF)
    {
        sprintf((char*)pname,"0:PHOTO/PIC%05d.bmp",index);
        res=f_open(ftemp,(const TCHAR*)pname,FA_READ);//尝试打开这个文件
        if(res==FR_NO_FILE)break;       //该文件名不存在=正是我们需要的.
        index++;
    }
}

int main(void)
{
    u8 res;
    u8 *pname;                          //带路径的文件名
    u8 key;                             //键值
    u8 i;
    u8 sd_ok=1;                         //0,sd 卡不正常;1,SD 卡正常.
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);     //设置时钟,72M
    delay_init(72);                     //初始化延时函数
    uart_init(115200);                  //初始化串口
    usmart_dev.init(72);                //初始化 USMART
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    BEEP_Init();                        //初始化蜂鸣器
    LCD_Init();                         //初始化 LCD FSMC 接口
    SRAM_Init();                        //初始化外部 SRAM
    W25QXX_Init();                      //初始化 W25Q128
    my_mem_init(SRAMIN);                //初始化内部内存池
    my_mem_init(SRAMEX);                //初始化外部内存池
    exfuns_init();                      //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1);              //挂载 SD 卡
    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);//清除显示
    }
    Show_Str(30,50,200,16,"精英 STM32F1 开发板",16,0);
    Show_Str(30,70,200,16,"照相机实验",16,0);
    Show_Str(30,90,200,16,"KEY0:拍照",16,0);
    Show_Str(30,110,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,130,200,16,"2019 年 10 月 9 日",16,0);
    res=f_mkdir("0:/PHOTO");          //创建 PHOTO 文件夹
    if(res!=FR_EXIST&&res!=FR_OK)     //发生了错误
    {
        Show_Str(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;
    }else
    {
        Show_Str(30,150,240,16,"SD 卡正常!",16,0);
        delay_ms(200);
        Show_Str(30,170,240,16,"KEY0:拍照",16,0);
        sd_ok=1;
    }
    pname=mymalloc(SRAMIN,30);  //为带路径的文件名分配 30 个字节的内存
    while(pname==NULL)                //内存分配出错
    {
        Show_Str(30,190,240,16,"内存分配失败!",16,0);
        delay_ms(200);
        LCD_Fill(30,190,240,146,WHITE);//清除显示
        delay_ms(200);
    }
    while(OV7725_Init())//初始化 OV7725
    {
        Show_Str(30,190,240,16,"OV7725  错误!",16,0);
        delay_ms(200);
        LCD_Fill(30,190,239,206,WHITE);
        delay_ms(200);
    }
    Show_Str(30,190,200,16,"OV7725  正常",16,0);
    delay_ms(500);
    TIM6_Init(10000,7199);                  //10Khz 计数频率,1 秒钟中断

    EXTI8_Init();                           //使能定时器捕获
    while(1)
    {
```

```
        key=KEY_Scan(0);
        if(key==KEY0_PRES)
        {
            OV7725_Window_Set(320,240,0);//QVGA 模式输出
            break;
        }else if(key==KEY1_PRES)
        {
            OV7725_Window_Set(320,240,1);//VGA 模式输出
            break;
        }
        i++;
        if(i==100)LCD_ShowString(30,206,200,16,16,"KEY0:QVGA    KEY1:VGA");
                                                        //闪烁显示提示信息
        if(i==200)
        {
            LCD_Fill(30,206,200,250+16,WHITE);
            i=0;
        }
        delay_ms(5);
    }
    OV7725_CS=0;
    LCD_Clear(BLACK);
    while(1)
    {
        key=KEY_Scan(0);//不支持连按
        if(key==KEY0_PRES)
        {
            if(sd_ok)
            {
                LED1=0; //点亮 DS1,提示正在拍照
                camera_new_pathname(pname);//得到文件名
            if(bmp_encode(pname,(lcddev.width-240)/2,(lcddev.height-320)/2,240,320,0))
                                                        //拍照有误
                {
                    Show_Str(40,130,240,12,"写入文件错误!",12,0);
                }else
                {
                    Show_Str(40,130,240,12,"拍照成功!",12,0);
                    Show_Str(40,150,240,12,"保存为:",12,0);
                    Show_Str(40+42,150,240,12,pname,12,0);
                    BEEP=1; //蜂鸣器短叫，提示拍照完成
                    delay_ms(100);
                }
```

```
                }else //提示 SD 卡错误
                {
                    Show_Str(40,130,240,12,"SD 卡错误!",12,0);
                    Show_Str(40,150,240,12,"拍照功能不可用!",12,0);
                }
                BEEP=0;//关闭蜂鸣器
                LED1=1;//关闭 DS1
                delay_ms(1800);//等待 1.8 秒钟
                LCD_Clear(BLACK);
            }else delay_ms(5);
            camera_refresh();//更新显示
            i++;
            if(i==40)//DS0 闪烁.
            {
                i=0;
                LED0=!LED0;
            }
        }
    }
```

此部分代码，和第四十章的代码有点类似，只是这里我们多了一个 camera_new_pathname 函数，用于获取新的 bmp 文件名字（不覆盖旧的）。在 main 函数里面，我们通过 KEY0 按键控制拍照（调用 bmp_encode 函数实现），其他部分我们就不多介绍了，至此照相机实验代码编写完成。

最后，本实验可以通过 USMART 来测试 BMP 编码函数，将 bmp_encode 函数添加到 USMART 管理，即可通过串口自行控制拍照，方便测试。

44.4 下载验证

本实验我们使用 OV7725 摄像头模块，插上 SD 卡，在代码编译成功之后，我们通过下载代码到 ALIENTEK 精英 STM32F103 上，得到如图 44.4.1 所示界面：

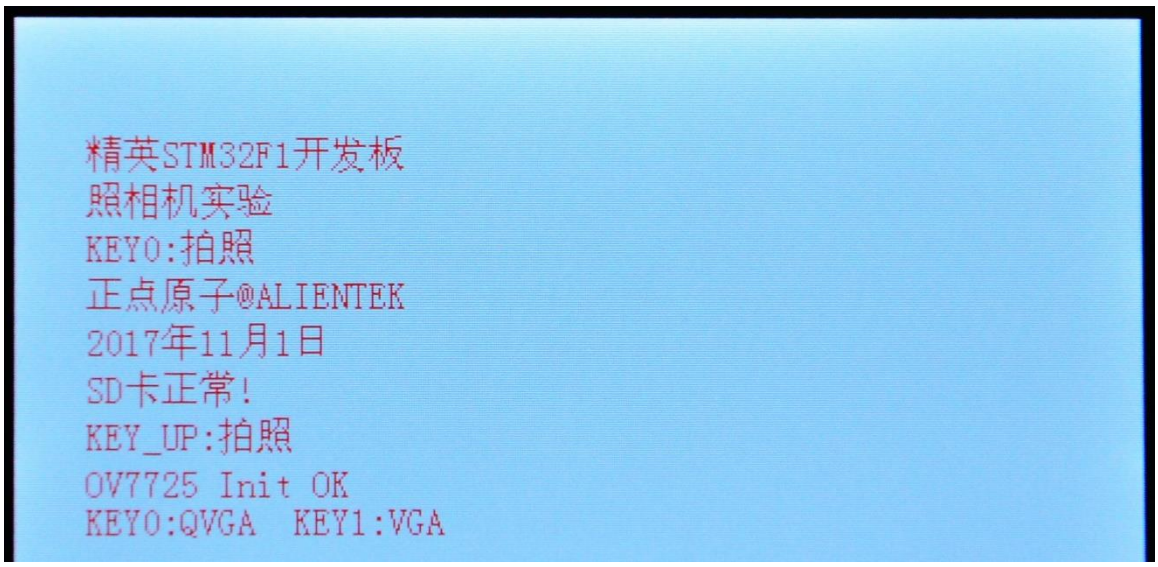


图 44.4.1 程序运行效果图

随后，按 KEY0 键进入监控界面。进入监控界面后，我们就可以按下 KEY0 即可进行拍照了。拍照得到的照片效果如图 44.4.2 所示：



图 44.4.2 拍照样图（bmp 拍照样图）

最后，我们还可以通过 USMART 调用 bmp_encode 函数，实现串口控制拍照，还可以拍成各种尺寸哦（不过必须小于 240*320）！

第四十五章 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 精英 STM32F103 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：

45.1 手写识别简介
45.2 硬件设计
45.3 软件设计
45.4 下载验证

45.1 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 精英 STM32F103 自带的 TFTLCD 触摸屏（2.8/3.5/4.3/7 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 45.1.1 所示：

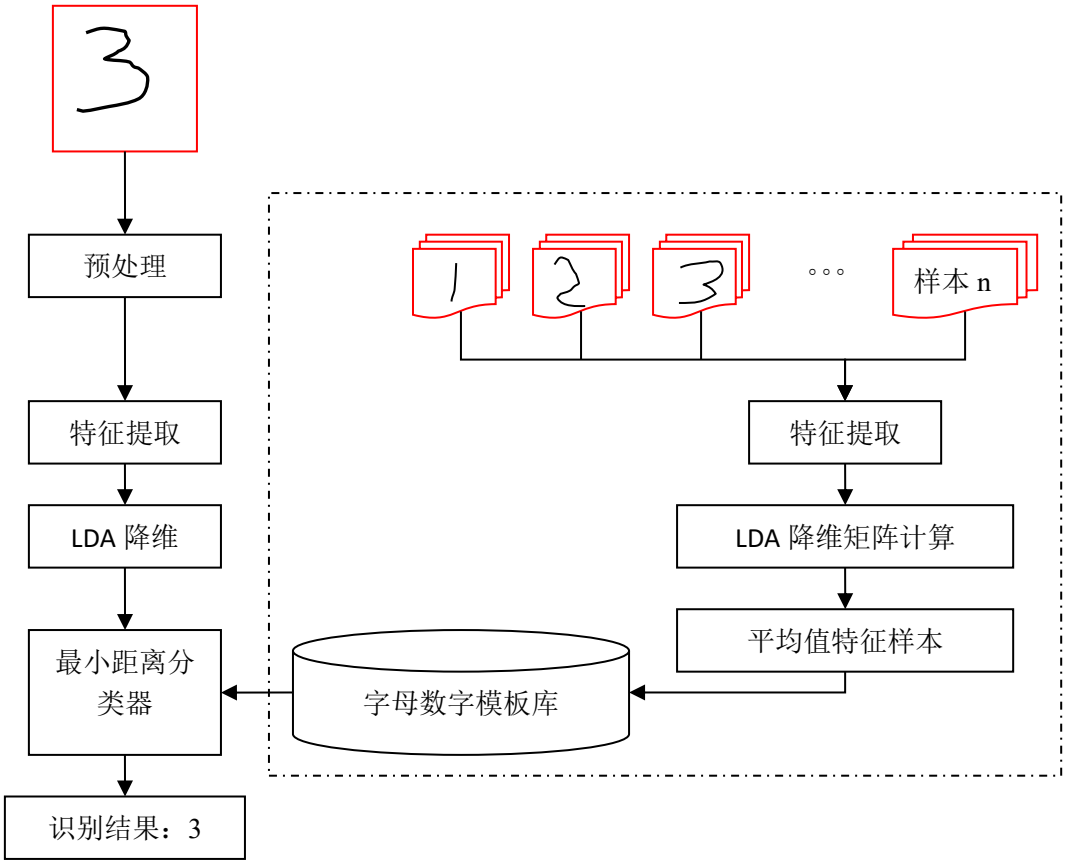


图 45.1.1 字母数字识别系统示意图。

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*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。

$$[d_1, d_2, \cdots, d_{512}] \times \begin{bmatrix} l & \cdots & l \\ \vdots & \ddots & \vdots \\ l & \cdots & l \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_{64} \end{bmatrix}$$

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。

$$output = \arg\min_{i \in [1,62]} \{(f_1 - f_1^i)^2 + (f_2 - f_2^i)^2 + \cdots + (f_{64} - f_{64}^i)^2\}$$

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 种模式：

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。

45.2 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 KEY_UP，可以进入触摸屏校准（仅电阻屏需要校准，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 KEY_UP 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。

45.3 软件设计

本章，我们在汉字显示实验（第四十二章例程）的基础上修改。

打开工程，因为本章需要用到触摸屏，所以得添加：ctiic.c、ft5206.c、ott2001a.c、gt9147.c、touch.c、24cxx.c 和 myiic.c 等 7 个文件到 HARDWARE 组下。

然后，在工程 HARDWARE 文件夹所在的文件夹下新建一个 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 test.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 key; u8 tcnt=0;
        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
        HAL_Init();                             //初始化 HAL 库
        Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
        delay_init(72);                         //初始化延时函数
        uart_init(115200);                      //初始化串口
        LED_Init();                             //初始化 LED
        KEY_Init();                             //初始化按键
        LCD_Init();                             //初始化 LCD FSMC 接口
        SRAM_Init();                            //初始化外部 SRAM
        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(SRAMEX);                    //初始化外部内存池
        exfuns_init();                          //为 fatfs 相关变量申请内存
        f_mount(fs[0],"0:",1);                  //挂载 SD 卡
        f_mount(fs[1],"1:",1);                  //挂载 FLASH.
        alientek_ncr_init();                    //初始化手写识别
        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
        }
```

```
RESTART:
    Show_Str(30,10,200,16,"精英  STM32 开发板",16,0);
    Show_Str(30,30,200,16,"手写识别实验",16,0);
    Show_Str(30,50,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,70,200,16,"KEY0:MODE KEY_UP:Adjust",16,0);
    Show_Str(30,90,200,16,"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==WKUP_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);
                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(30+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 45.1.2 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。

45.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 精英 STM32F103 上，得到，如图 45.4.1 所示：

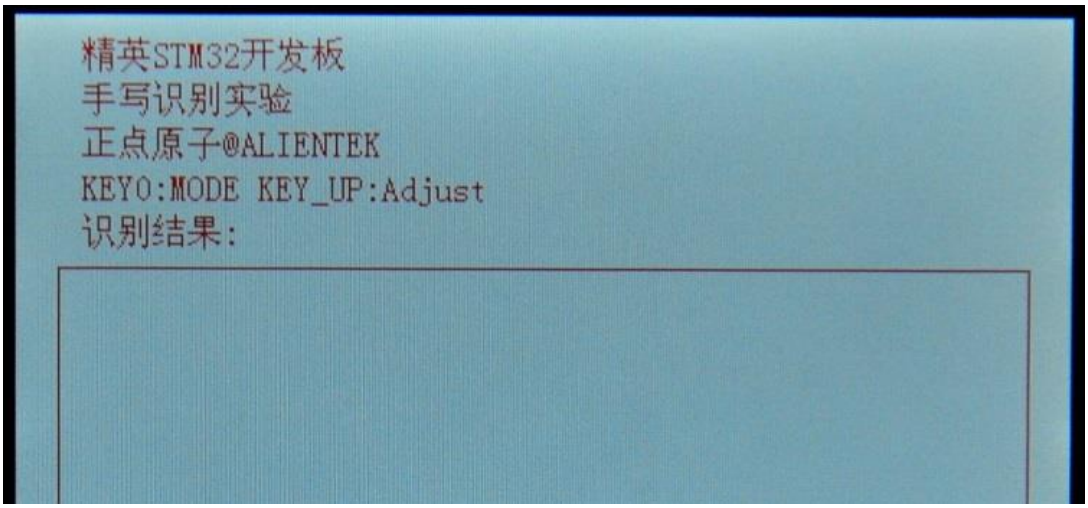


图 45.4.1 手写识别界面

此时，我们在手写区写数字/字母，即可得到识别结果，如图 45.4.2 所示：

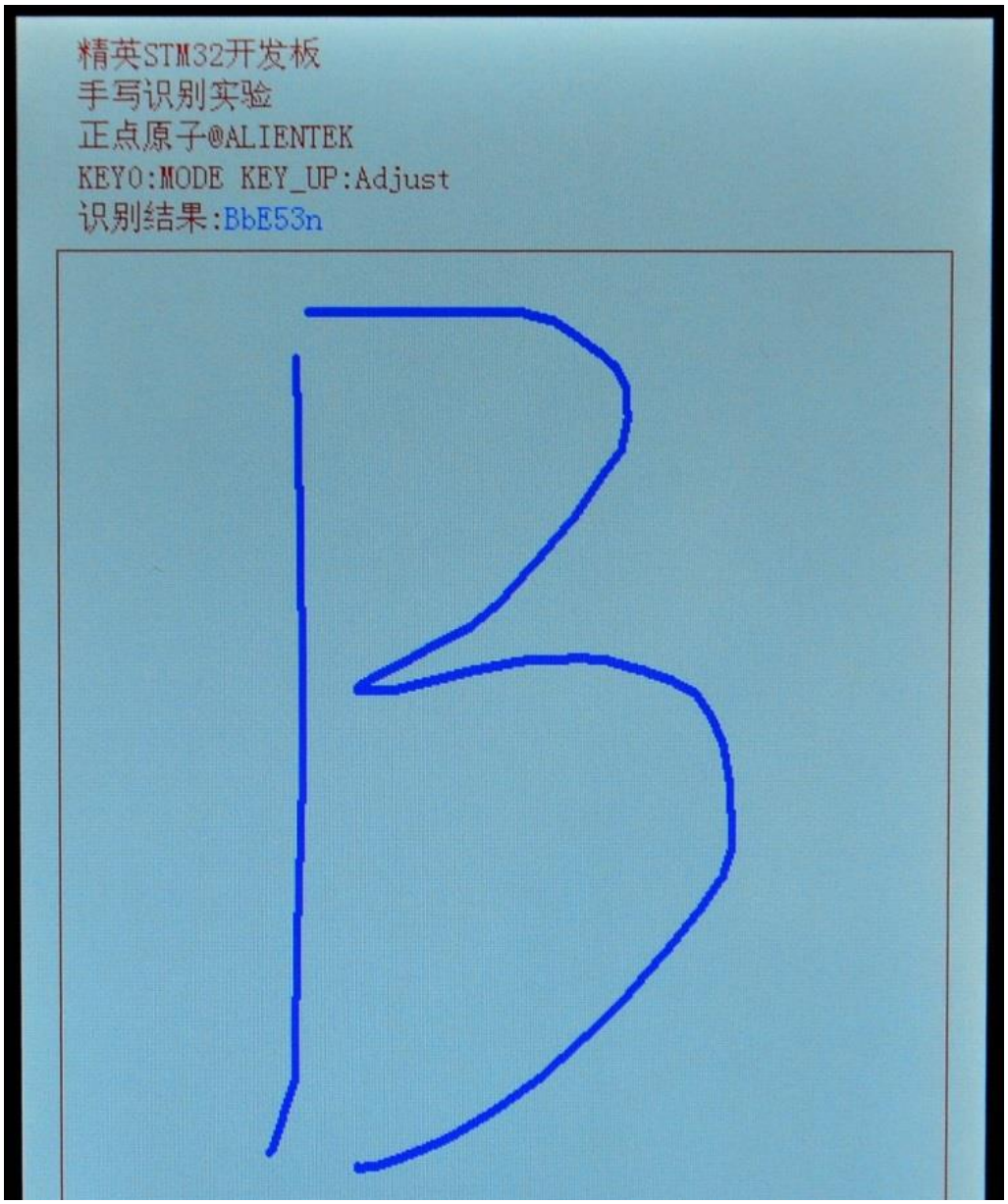


图 45.4.2 手写识别结果

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 KEY_UP 可以进行屏幕校准（仅限电阻屏，电容屏无需校准）。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。

第四十六章 T9 拼音输入法实验

上一章，我们在 ALIENTEK 精英 STM32F103 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 精英 STM32F103 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：

46.1 拼音输入法简介
46.2 硬件设计
46.3 软件设计
46.4 下载验证

46.1 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 46.1.1 所示：

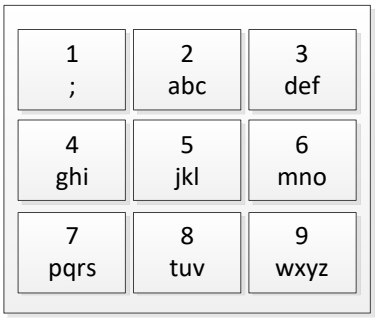


图 46.1.1 手机拼音输入键盘

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     []={"啊阿腌吖锕屙嗄錒呵腌"};
const u8 PY_mb_ai    []={"爱埃挨哎唉哀皑癌蔼矮艾碍隘捱嗳嗌嫒瑷暧砹锿霭"};
const u8 PY_mb_an    []={"安俺按暗岸案鞍氨谙胺埯揞犴庵桉铵鹌黯"};
……此处省略 N 多组合
const u8 PY_mb_zu    []={"足租祖诅阻组卒族俎菹镞"};
const u8 PY_mb_zuan  []={"钻攥纂缵躜"};
const u8 PY_mb_zui   []={"最罪嘴醉蕞觜"};
const u8 PY_mb_zun   []={"尊遵樽鳟撙"};
const u8 PY_mb_zuo   []={"左佐做作坐座昨撮唑柞阼琢嘬怍胙祚砟酢"};
```

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。

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 KEY1 进行选择。按键 KEY0 用于清除一次输入，按键 KEY_UP 用于触摸屏校准。

本实验用到的资源如下：

1） 指示灯 DS0

2） 三个按键（KEY0/KEY1/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。

46.3 软件设计

打开上一章的工程，由于本章用不到手写识别的功能，所以去掉 ATKNCR 组及其源码。同时，本章还要用到 USMART，所以添加 USMART 组件相关代码（方法见：第 19.3 节）。

然后，在 HARDWARE 文件夹所在的文件夹下新建一个 T9INPUT 的文件夹。在该文件夹下面新建 pyinput.c、pyinput.h 和 pymb.h 三个文件，然后在工程里面新建一个 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，在该文件输入如下代码：

```
//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数字串同拼音索引表里面的各个项对比，找出匹配结果，并将完全匹配的项目存放在 matchlist 里面，同时记录匹配数。对于那些没有完全匹配的输入串，则查找与其最佳匹配的项目，并将匹配的长度返回。函数 test_py（代码省略）用于给 usmart 调用，实现串口测试，该函数可有可无，只是在串口测试的时候才用到，如果不使用的话，可以去掉，本章，我们将其加入 usmart 控制，大家可以通过该函数实现串口调试拼音输入法。

其他两个函数，也比较简单了，我们这里就不细说了，保存 pyinput.c，打开 pyinput.h，输入如下代码：

```
#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，我们打开 test.c，输入代码如下：

```
const u8* kbd_tbl[9]={"←","2","3","4","5","6","7","8","9",};//数字表
const u8* kbs_tbl[9]={"DEL","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz",};//字符表
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+kbdysize*3);
    LCD_DrawRectangle(x+kbdxsize,y,x+kbdxsize*2,y+kbdysize*3);
    LCD_DrawRectangle(x,y+kbdysize,x+kbdxsize*3,y+kbdysize*2);
    POINT_COLOR=BLUE;
    for(i=0;i<9;i++)
    {
        Show_Str_Mid(x+(i%3)*kbdxsize,y+4+kbdysize*(i/3),(u8*)kbd_tbl[i],16,kbdxsize);
        Show_Str_Mid(x+(i%3)*kbdxsize,y+kbdysize/2+kbdysize*(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*kbdysize+1,x+j*kbdxsize+kbdxsize-1,y+i*kbdysize+
                    kbdysize-1,GREEN);
    else LCD_Fill(x+j*kbdxsize+1,y+i*kbdysize+1,x+j*kbdxsize+kbdxsize-1,y+i*kbdysize+
                  kbdysize-1,WHITE);
    Show_Str_Mid(x+j*kbdxsize,y+4+kbdysize*i,(u8*)kbd_tbl[keyx],16,kbdxsize);
    Show_Str_Mid(x+j*kbdxsize,y+kbdysize/2+kbdysize*i,(u8*)kbs_tbl[keyx],16,kbdxsize);
}
//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*kbdxsize+kbdxsize)&&tp_dev.x[0]>(x+j*kbdxsize)
                    &&tp_dev.y[0]<(y+i*kbdysize+kbdysize)&&tp_dev.y[0]>(y+i*kbdysize))
                    {
                        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 result_num; u8 cur_index;
    u8 inputstr[7];                         //最大输入 6 个字符+结束符
    u8 inputlen;                            //输入长度
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    KEY_Init();                             //初始化按键
    LCD_Init();                             //初始化 LCD FSMC 接口
    SRAM_Init();                            //初始化外部 SRAM
    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
    exfuns_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);//清除显示
    }
 RESTART:
    POINT_COLOR=RED;
    Show_Str(30,5,200,16,"精英  STM32 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,"KEY_UP:校准   KEY0:清除",16,0);
    Show_Str(30,85,200,16,"   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;kbdysize=43;}//根据 LCD 分辨率设置按键大小
    else if(lcddev.id==0X5510){kbdxsize=140;kbdysize=70;}
    else {kbdxsize=60;kbdysize=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);
                    //重新获取完全匹配字符数
                }
            }else                               //没有任何匹配
            {
                inputlen--;
                inputstr[inputlen]='\0';
            }
        }else
        {
            cur_index=0;
            result_num=0;
        }
        LCD_Fill(30+40,105,30+40+48,110+16,WHITE);      //清除之前的显示
```

```
                LCD_ShowNum(30+144,105,result_num,1,16);  //显示匹配的结果数
                Show_Str(30+40,105,200,16,inputstr,16,0);       //显示有效的数字串
                py_show_result(cur_index);                      //显示第 cur_index 的匹配结果
            }
            key=KEY_Scan(0);
            if(key==WKUP_PRES&&tp_dev.touchtype==0)       //WKUP_PRES 按下&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 46.2 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：kbdxsize 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。

46.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 精英 STM32F103 上，得到，如图 46.4.1 所示：

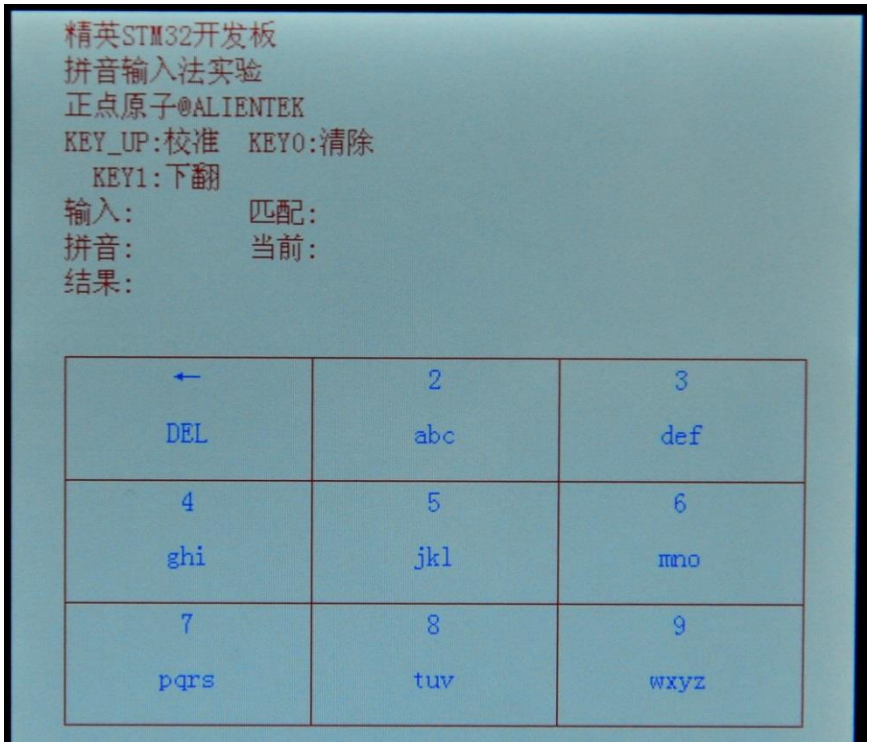


图 46.4.1 汉字输入法界面

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 46.4.2 所示：

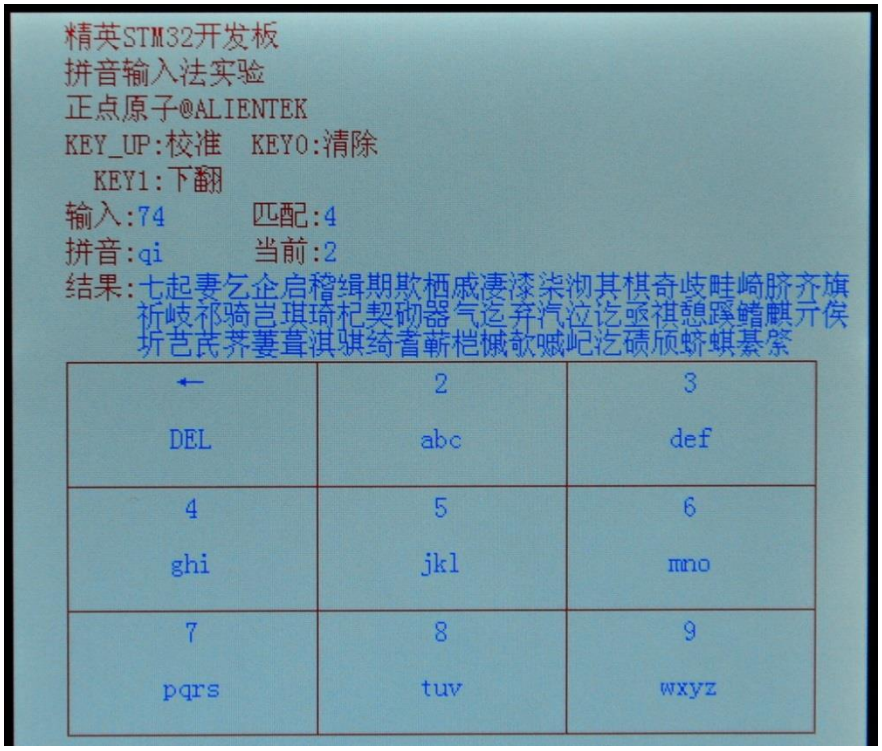


图 46.4.2 实现拼音输入

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 KEY1 来选择拼音。通过按下 KEY0，可以清除当前输入，通过按下 KEY_UP 可以实现触摸屏校准（仅限电阻屏，电容屏无需校准）。

我们还可以通过 USMART 调用 test_py 来实现输入法调试，如图 46.4.3 所示：

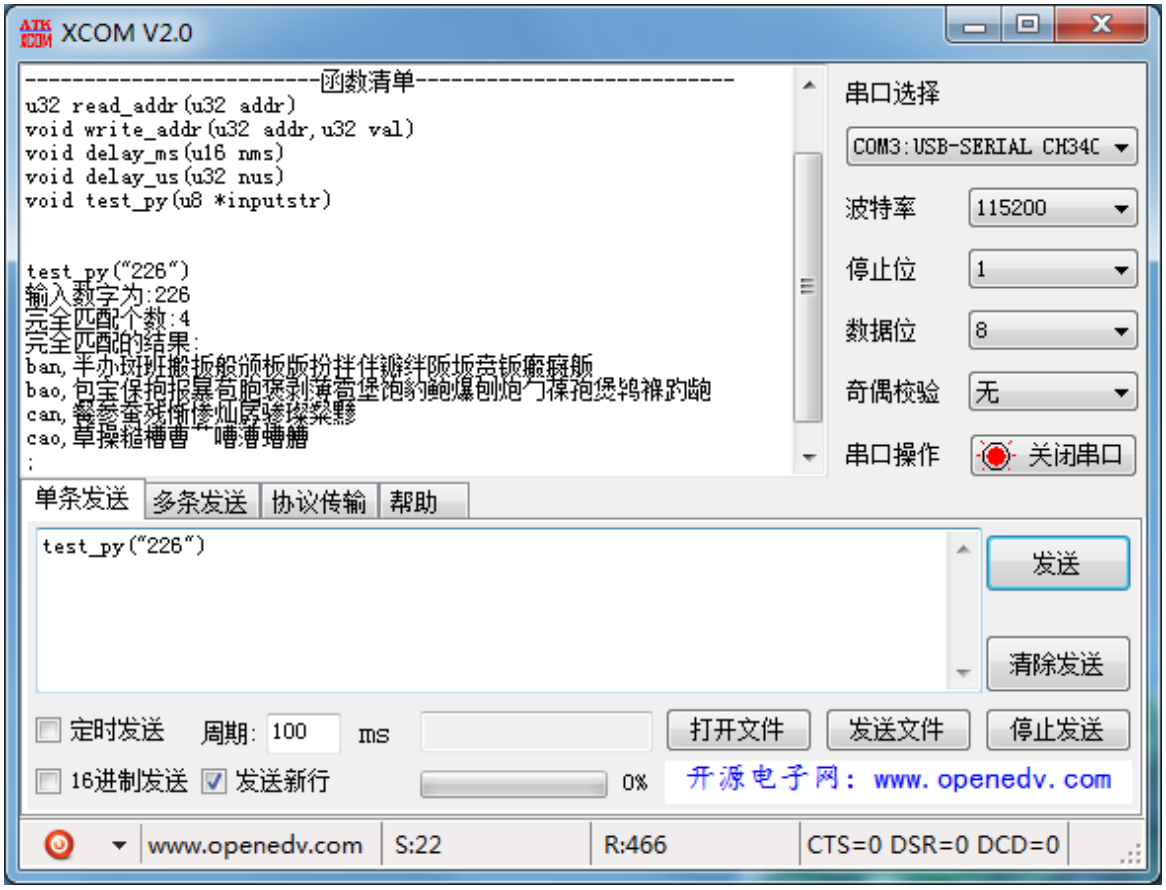


图 46.4.3 USMART 调试 T9 拼音输入法

第四十七章 串口 IAP 实验

IAP，即在应用编程。很多单片机都支持这个功能，STM32F1 也不例外。在之前的 FLASH 模拟 EEPROM 实验里面，我们学习了 STM32F1 的 FLASH 自编程，本章我们将结合 FLASH 自编程的知识，通过 STM32F1 的串口实现一个简单的 IAP 功能本章分为如下几个部分：

47.1 IAP 简介
47.2 硬件设计
47.3 软件设计
47.4 下载验证

47.1 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 STM32F1 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。

STM32F1 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 STM32F1 正常的程序运行流程，如图 47.1.1 所示：

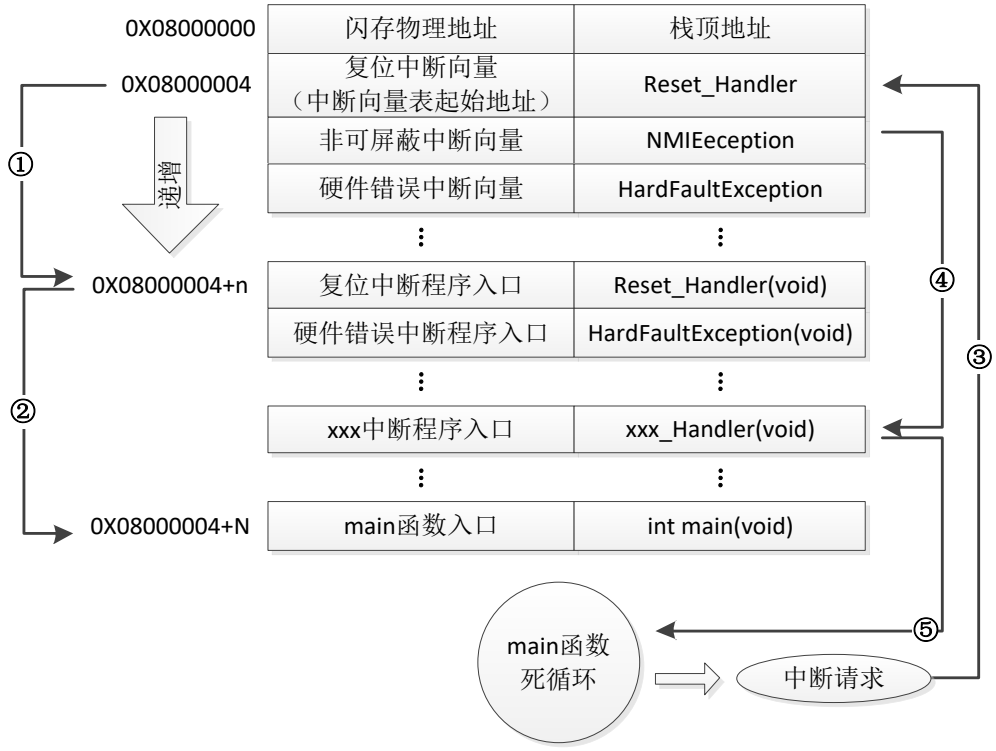


图 47.1.1 STM32F1 正常运行流程图

STM32F1 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 STM32F1 是基于 Cortex-M3 内核的微控制器，其内部通过一张“中断向量表”来响应中断，程序启动后，将首先从“中断向量表”取出复位中断向量执行复位中断程序完成启动，而这张“中断向量表”的起始地址是 0x08000004，当中断来临，STM32F1 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 47.1.1 中，STM32F1 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 STM32F1 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 47.1.2 所示：

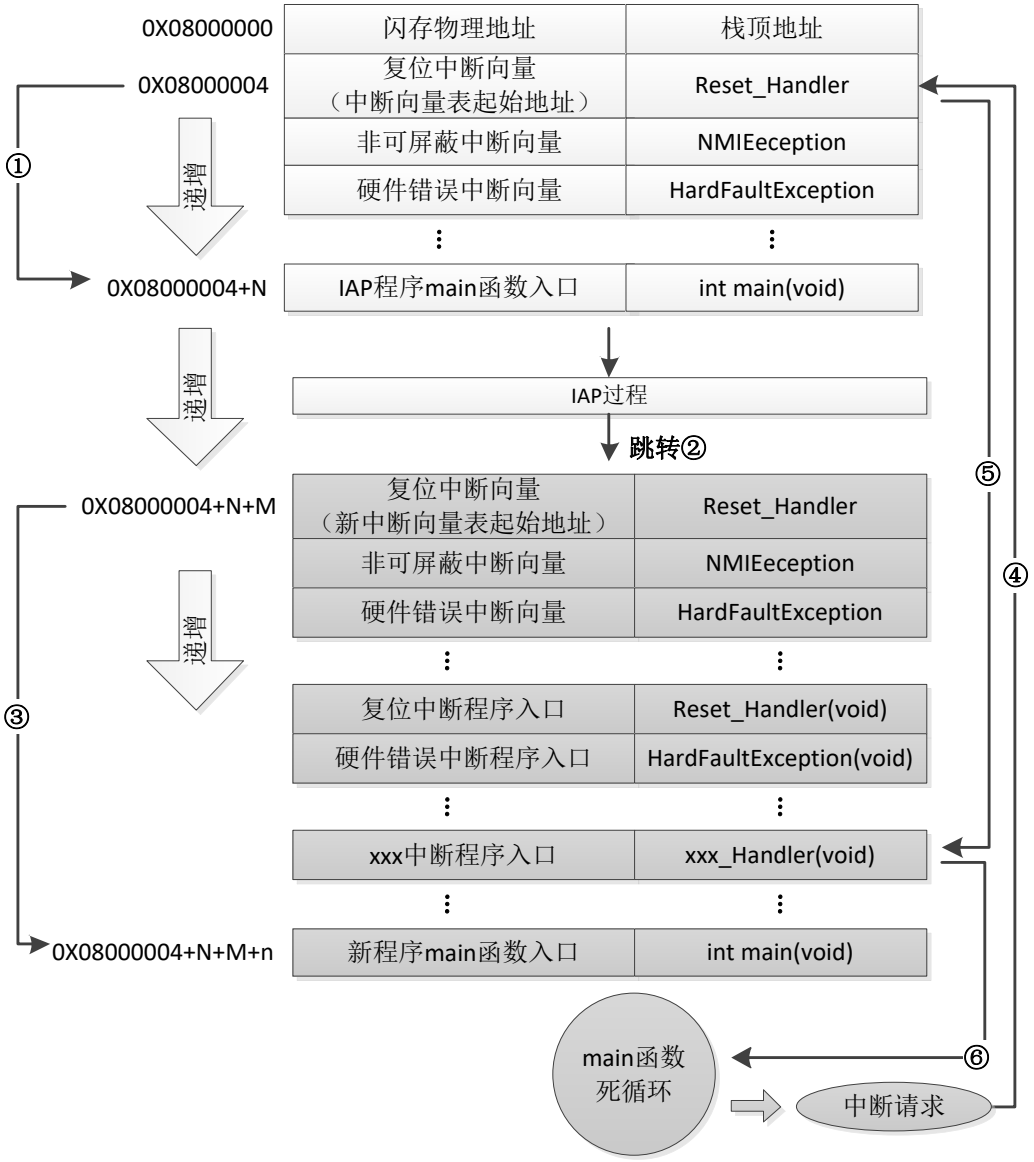


图 47.1.2 加入 IAP 之后程序运行流程图

在图 47.1.2 所示流程中，STM32F1 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 47.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32F1 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 STM32F1 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 47.1.2 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 47.1.3 所示：

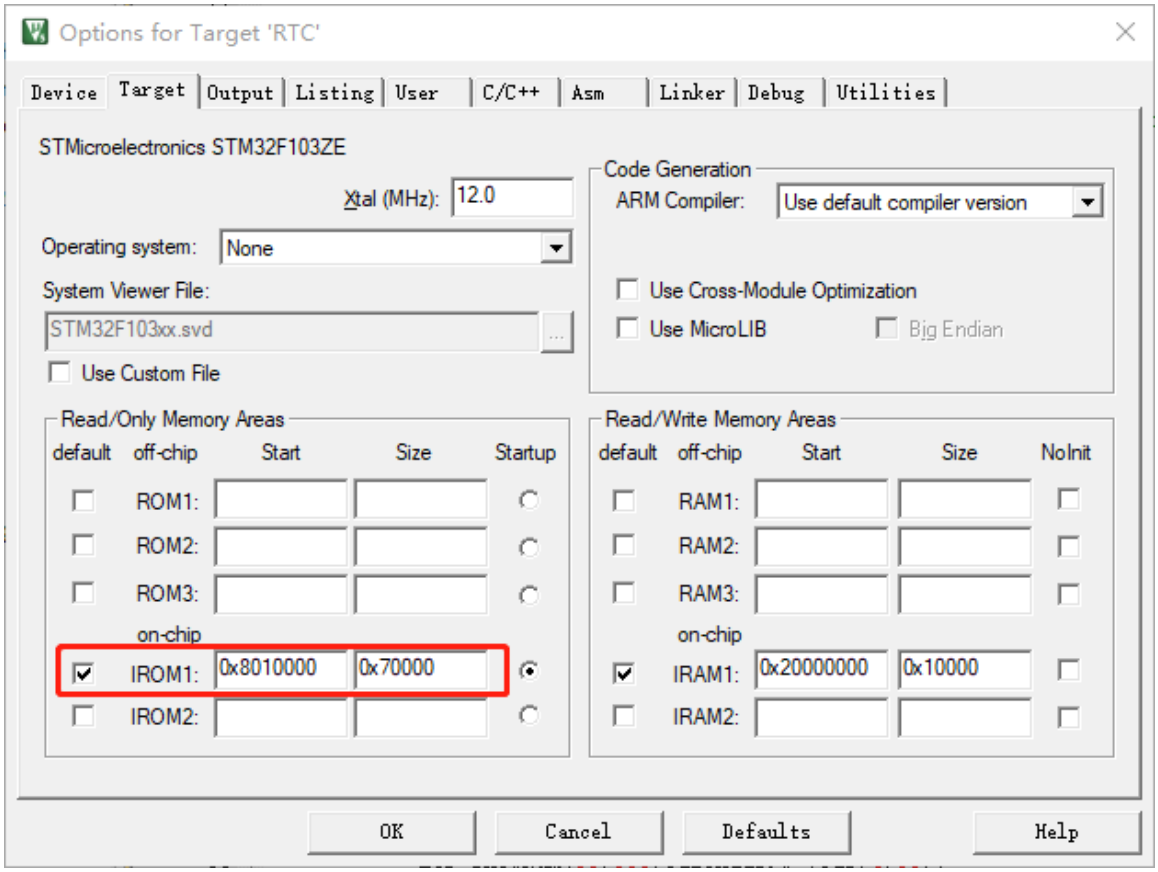


图 47.1.3 FLASH APP Target 选项卡设置

默认的条件下，图中 IROM1 的起始地址（Start）一般为 0X08000000，大小（Size）为 0X80000，即从 0X08000000 开始的 512K 空间为我们的程序存储区。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 0X80000-0X10000=0X70000（448K 字节）大小了。设置好 Start 和 Szie，就完成 APP 程序的起始地址设置。

这里的 64K 字节，需要大家根据 Bootloader 程序大小进行选择，比如我们本章的 Bootloader 程序为 40K 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 47.1.4 所示：

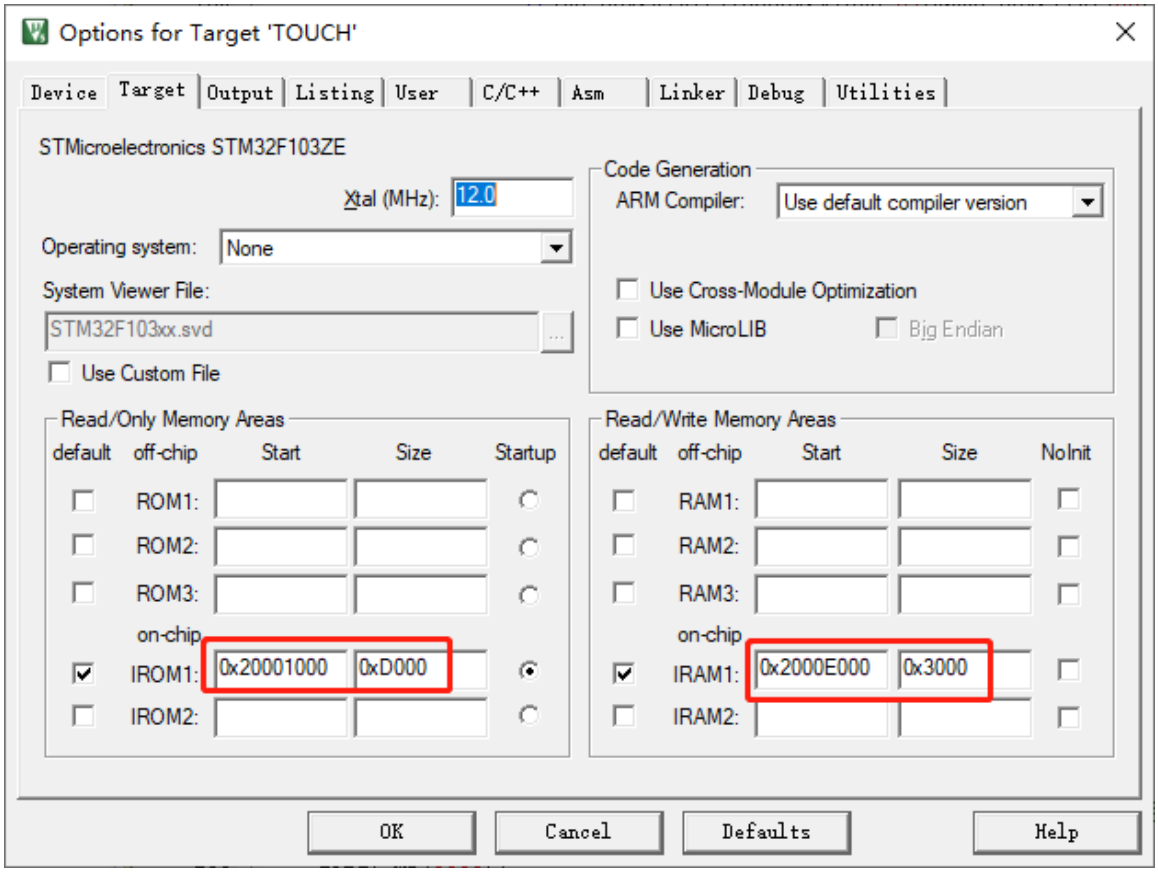


图 47.1.4 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址（Start）定义为：0X20001000，大小为 0XD000（52K 字节），即从地址 0X20000000 偏移 0X1000 开始，存放 APP 代码。因为整个 STM32F103ZET6 的 SRAM 大小为 64K 字节，所以 IRAM1（SRAM）的起始地址变为 0X2001E000，大小只有 0X3000（12K 字节）。这样，整个 STM32F103ZET6 的 SRAM 分配情况为：最开始的 4K 给 Bootloader 程序使用，随后的 52K 存放 APP 程序，最后 8K，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 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 After Build/Rebuild 栏，勾选 Run #1，并写入：C:\Keil_v5\ARM\ARMCC\bin\fromelf.exe --bin -o ..\OBJ\RTC.bin ..\OBJ\RTC.axf，如图 47.1.5 所示：

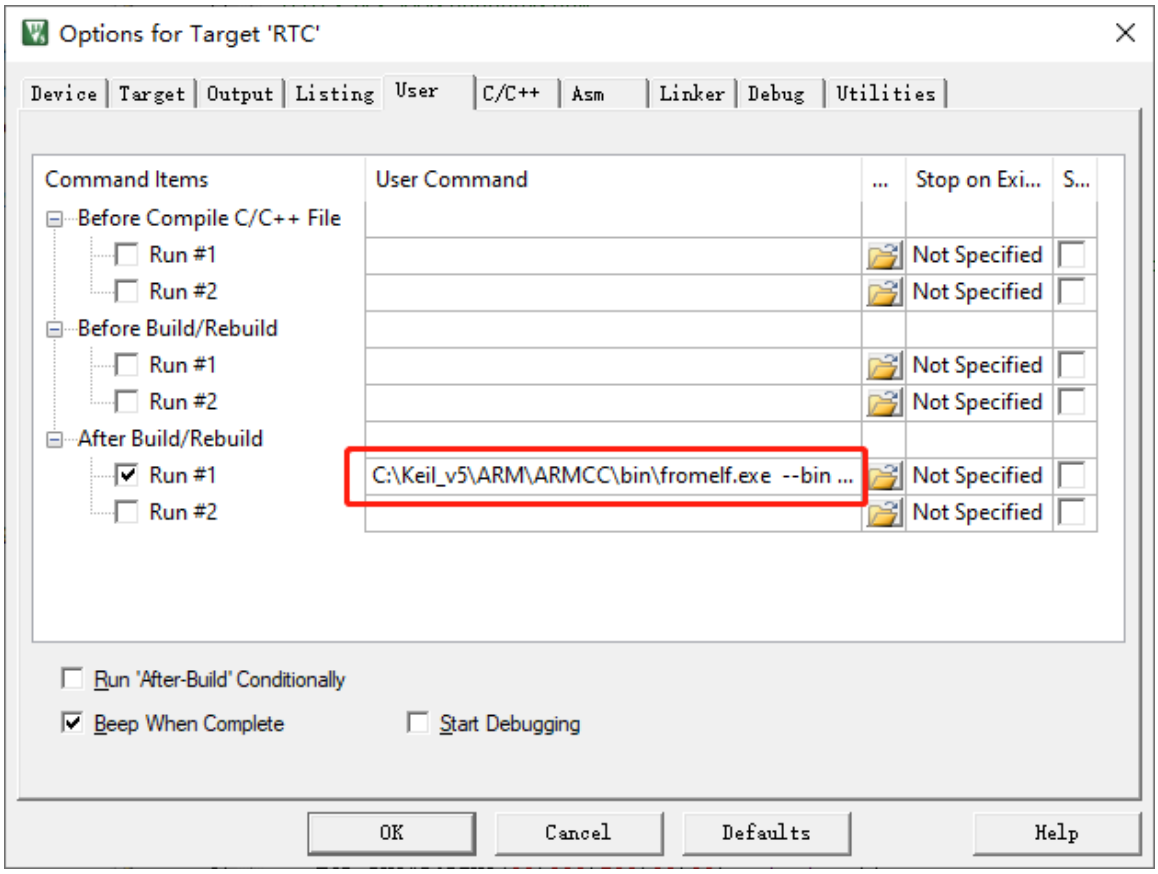


图 47.1.5 MDK 生成.bin 文件设置方法

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 C:\Keil_v5 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 RTC.axf，生成一个 RTC.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。

47.2 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 STM32F1 的 FLASH，之后再按 KEY1 即可以执行这个 FLASH 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） 三个按键（KEY0/KEY1/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。

47.3 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（在第三十二章介绍）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 52.1 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;
u16 iapbuf[1024];
//appxaddr:应用程序的起始地址
//appbuf:应用程序 CODE.
//appsize:应用程序大小(字节).
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 appsize)
{
    u16 t;
    u16 i=0;
    u16 temp;
    u32 fwaddr=appxaddr;//当前写入的地址
    u8 *dfu=appbuf;
    for(t=0;t<appsize;t+=2)
    {
        temp=(u16)dfu[1]<<8;
        temp+=(u16)dfu[0];
        dfu+=2;//偏移 2 个字节
        iapbuf[i++]=temp;
```

```
            if(i==1024)
            {
                i=0;
                STMFLASH_Write(fwaddr,iapbuf,1024);
                fwaddr+=2048;//偏移 2048   16=2*8.所以要乘以 2.
            }
        }
        if(i)STMFLASH_Write(fwaddr,iapbuf,i);    //将最后的一些内容字节写进去.
    }
    //跳转到应用程序段
    //appxaddr:用户代码起始地址.
    void iap_load_app(u32 appxaddr)
    {
        if(((*(vu32*)appxaddr)&0x2FFE0000)==0x20000000)      //检查栈顶地址是否合法.
        {
            jump2app=(iapfun)*(vu32*)(appxaddr+4);
            //用户代码区第二个字为程序开始地址(复位地址)
            MSR_MSP(*(vu32*)appxaddr);
            //初始化 APP 堆栈指针(用户代码区的第一个字用于存放栈顶地址)
            jump2app();                                      //跳转到 APP.
        }
    }
```

该文件总共只有 2 个函数，其中，iap_write_appbin 函数用于将存放在串口接收 buf 里面的 APP 程序写入到 FLASH。iap_load_app 函数，则用于跳转到 APP 程序运行，其参数 appxaddr 为 APP 程序的起始地址，程序先判断栈顶地址是否合法，在得到合法的栈顶地址后，通过 MSR_MSP 函数（该函数在 sys.c 文件）设置栈顶地址，最后通过一个虚拟的函数（jump2app）跳转到 APP 程序执行代码，实现 IAP→APP 的跳转。

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 0X08000000~0X08008FFF 的空间为 Bootloader 使用
void iap_load_app(u32 appxaddr);                        //跳转到 APP 程序执行
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 applen);   //在指定地址开始,写入 bin
#endif
```

这部分代码比较简单，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 55K 字节，也就是串口最大一次可以接收 55K 字节的数据，这也是本 Bootloader 程序所能接收的最大 APP 程序大小。然后新增一个 USART_RX_CNT 的变量，用于记录接收到的文件大小，而 USART_RX_STA 不再使用。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 0x0d
//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;
#if SYSTEM_SUPPORT_OS               //如果 SYSTEM_SUPPORT_OS 为真，则需要支持 OS.
    OSIntEnter();
#endif
    if(USART1->SR&(1<<5))//接收到数据
    {
        res=USART1->DR;
        if(USART_RX_CNT<USART_REC_LEN)
        {
            USART_RX_BUF[USART_RX_CNT]=res;
            USART_RX_CNT++;
        }
    }
#if SYSTEM_SUPPORT_OS       //如果 SYSTEM_SUPPORT_OS 为真，则需要支持 OS.
    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 test.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
    u16 applenth=0;                         //接收到的 app 代码长度
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
```

```
KEY_Init();                                 //初始化按键
LCD_Init();                                 //初始化 LCD FSMC 接口
POINT_COLOR=RED;                            //设置字体为红色
LCD_ShowString(30,50,200,16,16,"ELITE STM32F103 ^_^");
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,"2015/1/27");
LCD_ShowString(30,130,200,16,16,"KEY_UP:Copy APP2FLASH");
LCD_ShowString(30,150,200,16,16,"KEY1:Run FLASH APP");
LCD_ShowString(30,170,200,16,16,"KEY0:Run SRAM 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,则认为接收完成.
        {
            applenth=USART_RX_CNT;
            oldcount=0;
            USART_RX_CNT=0;
            printf("用户程序接收完成!\r\n");
            printf("代码长度:%dBytes\r\n",applenth);
        }else oldcount=USART_RX_CNT;
    }
    t++;
    delay_ms(10);
    if(t==30)
    {
        LED0=!LED0;
        t=0;
        if(clearflag)
        {
            clearflag--;
            if(clearflag==0)LCD_Fill(30,210,240,210+16,WHITE);//清除显示
        }
    }
    key=KEY_Scan(0);
    if(key==WKUP_PRES)
    {
        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)
        {
            printf("开始执行 FLASH 用户代码!!\r\n");
            if(((*(vu32*)(FLASH_APP1_ADDR+4))&0xFF000000)==0x08000000)
            {
                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)
        {
            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），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 47.3.1 所示：

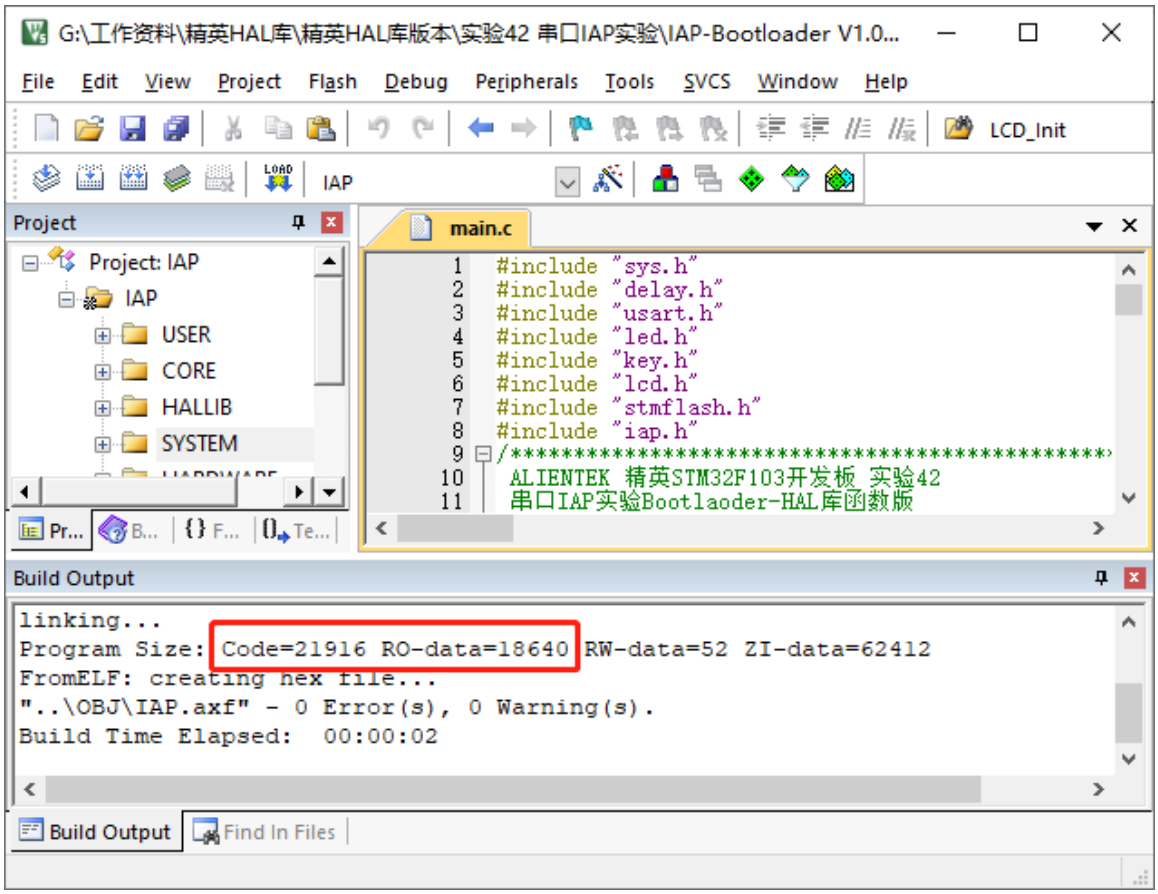


图 47.3.1 Bootloader 工程截图

从上图可以看出，Bootloader 大小为 40K 左右，比较大，主要原因是液晶驱动和 printf 占用了比较多的 flash，如果大家想删减代码，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 47.1 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。

47.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 精英 STM32F103 上，得到，如图 47.4.1 所示：

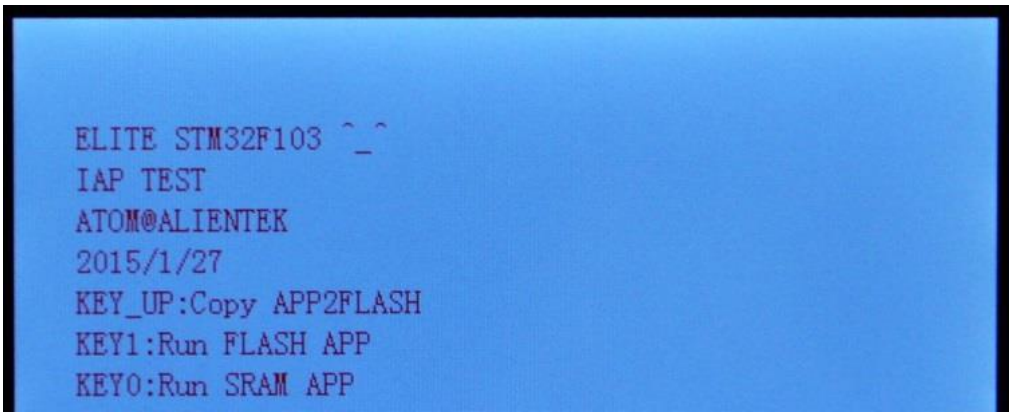


图 47.4.1 IAP 程序界面

此时，我们可以通过串口，发送 FLASH APP 或者 SRAM APP 到精英 STM32F103，如图 47.4.2 所示：



图 47.4.2 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 115200（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 文件发送给精英 STM32F103，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/KEY1 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）。

第四十八章 USB 虚拟串口实验

STM32F103 系列芯片都自带了 USB，不过 STM32F103 的 USB 都只能用来做设备，而不能用作主机。既便如此，对于一般应用来说已经足够了。本章，我们将向大家介绍如何在 ALIENTEK 精英 STM32 开发板上利用 STM32 自生的 USB 功能实现一个虚拟串口。本章分为如下几个部分：

48.1 USB 简介
48.2 硬件设计
48.3 软件设计
48.4 下载验证

48.1 USB 简介

USB ,是英文 Universal Serial BUS（通用串行总线）的缩写，而其中文简称为“通串线，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在 PC 领域的接口技术。USB 接口支持设备的即插即用和热插拔功能。USB 是在 1994 年底由英特尔、康柏、IBM、Microsoft 等多家公司联合提出的。

USB 发展到现在已经有 USB1.0/1.1/2.0/3.0 等多个版本。目前用的最多的就是 USB1.1 和 USB2.0，USB3.0 目前已经开始普及。STM32F103 自带的 USB 符合 USB2.0 规范。

标准 USB 共四根线组成,除 VCC/GND 外,另外为 D+,D-; 这两根数据线采用的是差分电压的方式进行数据传输的。在 USB 主机上，D-和 D+都是接了 15K 的电阻到低的，所以在没有设备接入的时候，D+、D-均是低电平。而在 USB 设备中，如果是高速设备，则会在 D+上接一个 1.5K 的电阻到 VCC，而如果是低速设备，则会在 D-上接一个 1.5K 的电阻到 VCC。这样当设备接入主机的时候，主机就可以判断是否有设备接入，并能判断设备是高速设备还是低速设备。接下来，我们简单介绍一下 STM32 的 USB 控制器。

STM32F103 的 MCU 自带 USB 从控制器，符合 USB 规范的通信连接；PC 主机和微控制器之间的数据传输是通过共享一专用的数据缓冲区来完成的，该数据缓冲区能被 USB 外设直接访问。这块专用数据缓冲区的大小由所使用的端点数目和每个端点最大的数据分组大小所决定，每个端点最大可使用 512 字节缓冲区（专用的 512 字节，和 CAN 共用），最多可用于 16 个单向或 8 个双向端点。USB 模块同 PC 主机通信，根据 USB 规范实现令牌分组的检测，数据发送/接收的处理，和握手分组的处理。整个传输的格式由硬件完成，其中包括 CRC 的生成和校验。

每个端点都有一个缓冲区描述块，描述该端点使用的缓冲区地址、大小和需要传输的字节数。当 USB 模块识别出一个有效的功能/端点的令牌分组时，(如果需要传输数据并且端点已配置)随之发生相关的数据传输。USB 模块通过一个内部的 16 位寄存器实现端口与专用缓冲区的数据交换。在所有的数据传输完成后，如果需要，则根据传输的方向，发送或接收适当的握手分组。在数据传输结束时，USB 模块将触发与端点相关的中断，通过读状态寄存器和/或者利用不同的中断来处理。

USB 的中断映射单元：将可能产生中断的 USB 事件映射到三个不同的 NVIC 请求线上：

1、USB 低优先级中断(通道 20)：可由所有 USB 事件触发(正确传输，USB 复位等)。固件在处理中断前应当首先确定中断源。

2、USB 高优先级中断(通道 19)：仅能由同步和双缓冲批量传输的正确传输事件触发，目的是保证最大的传输速率。

3、USB 唤醒中断(通道 42)：由 USB 挂起模式的唤醒事件触发。

USB 设备框图如图 48.1.1 所示：

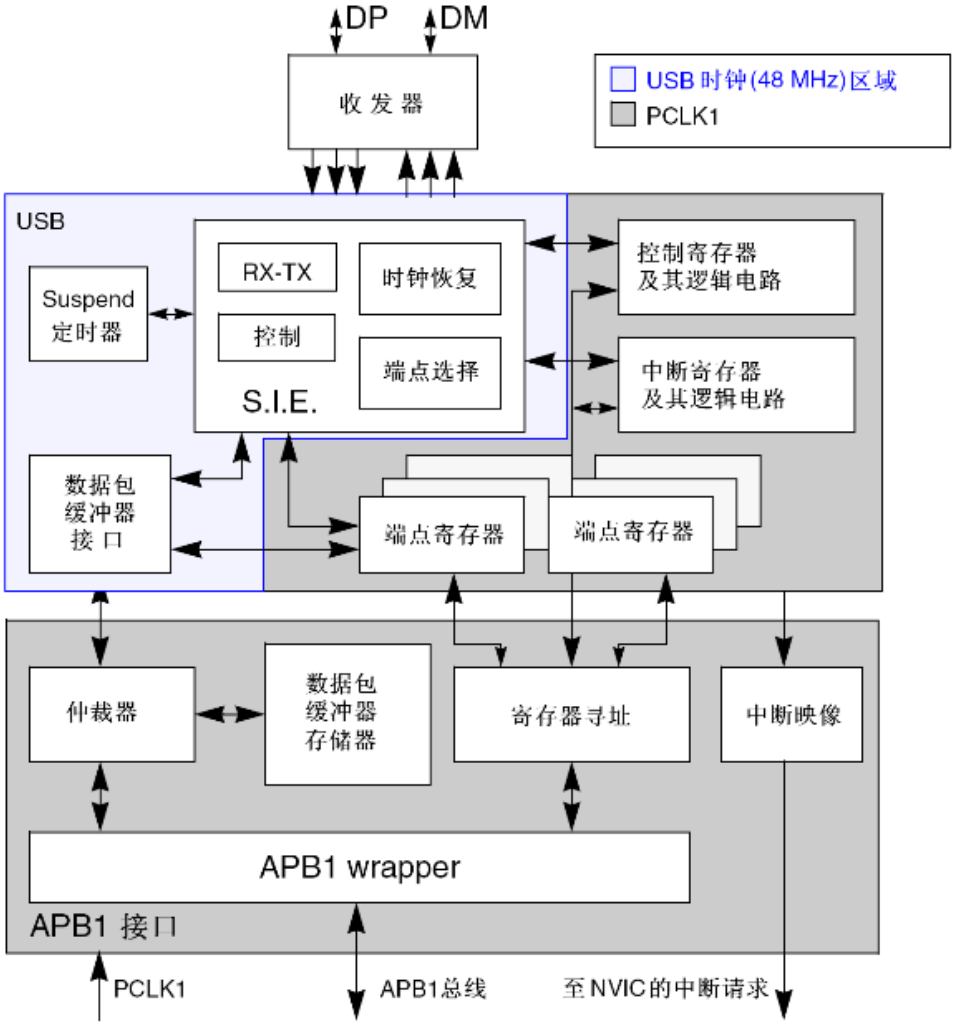


图 48.1.1 USB 设备框图

STM32F1 USB 的其他介绍，请大家参考《STM32 中文参考手册》第 21 章内容，我们这里就不再详细介绍了。

要正常使用 STM32F1 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 提供了我们一套完整的 USB 驱动库，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 驱动库，可以在：https://www.stmicroelectronics.com.cn/zh/microcontrollers-microprocessors/stm32f103.html#tools-software 这里下载。不过，我们已经帮大家下载到开发板光盘：**\8，STM32 参考资料\1，STM32CubeF1 固件包**。打开固件包 en.stm32cubef1\STM32Cube_FW_F1_V1.8.0\Middlewares\ST 目录下可以找到 USB 驱动库，如图 48.1.2 所示：

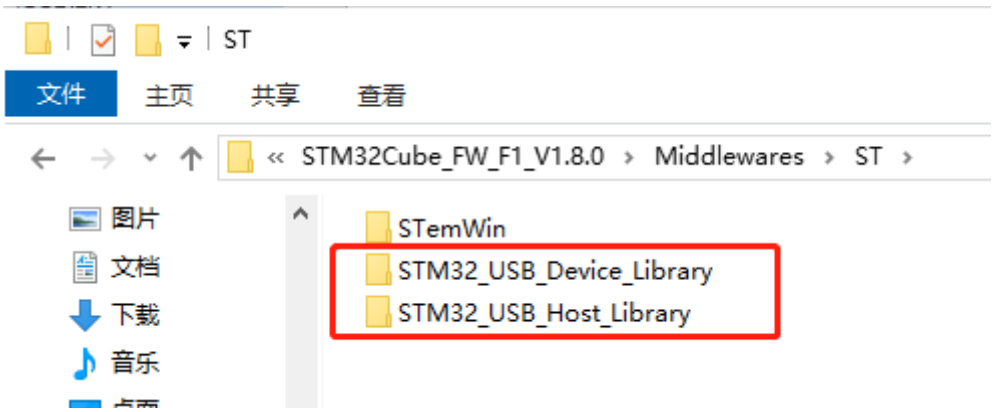


图 48.1.2 ST 提供的 USB 驱动库

USB 相关资料在目录：战舰 V3 资料盘(A 盘) \8，STM32 参考资料\2，STM32 USB 学习资料，专门讲解 USB 库的原理和使用。有了这些资料对我们了解 STM32F103 的 USB 会有不少帮助，尤其在不懂的时候，看看 ST 的例程，会有意想不到的收获。本实验的 USB 部分就是移植 ST 的 Virtual_COM_Port 例程相关部分而来，完成一个 USB 虚拟串口的功能。

48.2 硬件设计

本章实验功能简介：本实验利用 STM32 自带的 USB 功能,连接电脑 USB,虚拟出一个 USB 串口，实现电脑和开发板的数据通信。本例程功能完全同实验 4（串口实验），只不过串口变成了 STM32 的 USB 虚拟串口。当 USB 连接电脑（USB 线插入 USB_SLAVE 接口），开发板将通过 USB 和电脑建立连接虚拟出一个串口（注意：需要先安装:光盘\6，软件资料\1，软件\STM32 USB 虚拟串口驱动\VCP_V1.4.0_Setup.exe 这个驱动软件），USB 和电脑连接成功后，DS1 常亮。

在找到虚拟串口后，即可打开串口调试助手，实现同实验 4 一样的功能，即：STM32 通过 USB 虚拟串口和上位机对话，STM32 在收到上位机发过来的字符串（以回车换行结束）后，原原本本的返回给上位机。下载后，DS0 闪烁，提示程序在运行，同时每隔一定时间，通过 USB 虚拟串口输出一段信息到电脑。

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

前面 3 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 精英 STM32F103 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 48.2.1 所示：

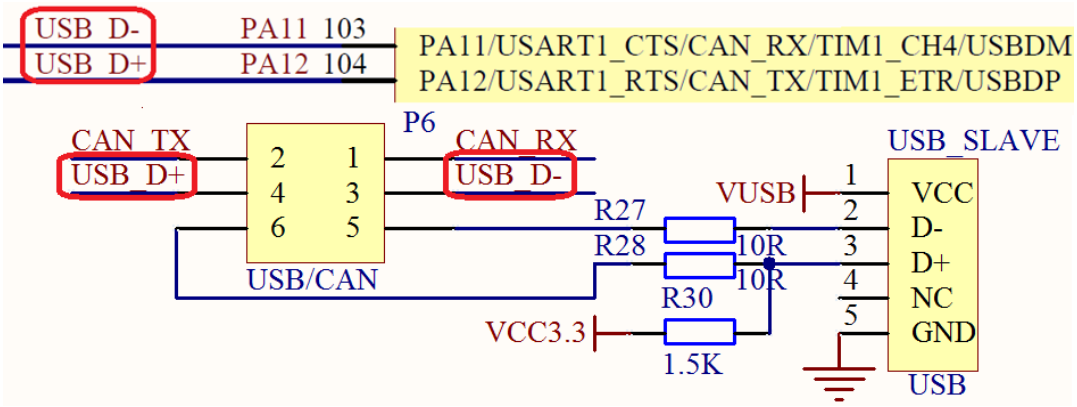


图 48.2.1 MiniUSB 接口与 STM32 的连接电路图

从上图可以看出，USB 座没有直接连接到 STM32F1 上面，而是通过 P6 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 48.2.2 所示：

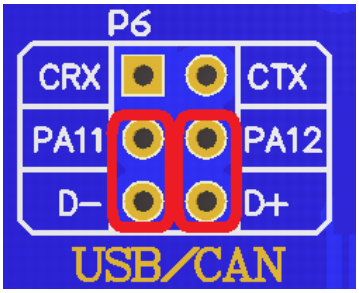


图 48.2.2 硬件连接示意图

用跳线帽短接 PA11 和 D-，PA12 和 D+之后，就完成了 MiniUSB 和 STM32 芯片的连接！！

48.3 软件设计

本章，我们在：实验 13 TFTLCD 显示实验的基础上修改，代码移植自 ST 官方例程，我打开该例程即可知道 USB 相关的代码有哪些，如图 48.3.1 所示：

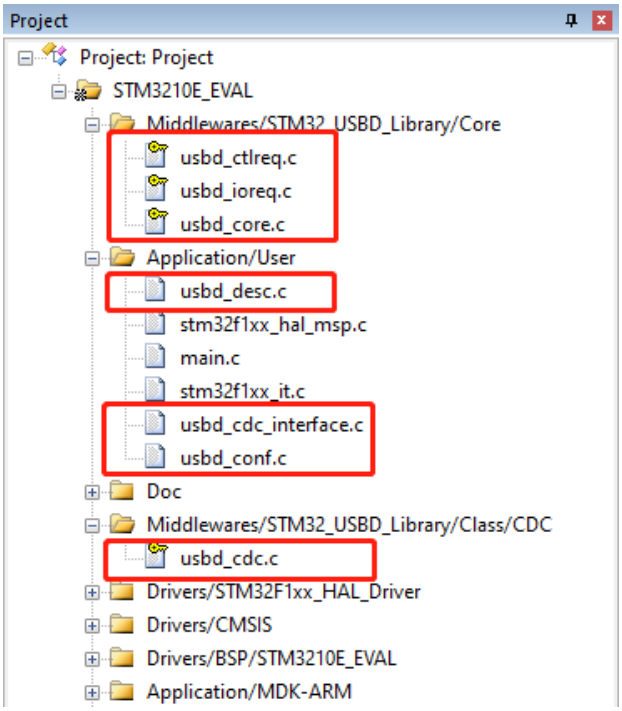


图 48.3.1 ST 官方例程 USB 相关代码

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 13 TFTLCD 显示实验）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：en.stm32cubef1\STM32Cube_FW_F1_V1.8.0\Middlewares\ST 文件夹下的 STM32_USB_Device_Library 文件夹到该文件夹下面。

然后，在 USB 文件夹下，新建 CONFIG 文件夹存放 Virtual COM 实现相关代码，最后 CONFIG 文件夹下的文件如图 48.3.2 所示：

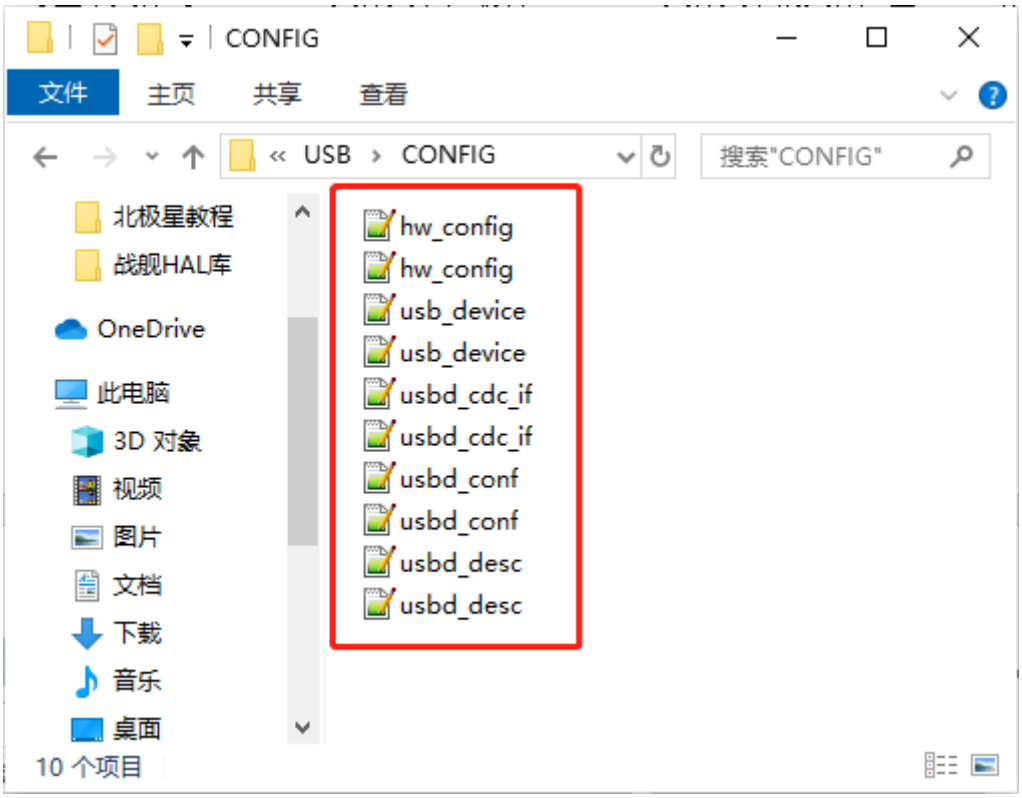


图 48.3.2 CONFIG 文件夹代码

之后，根据 ST 官方 Virtual_COM_Port 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 48.3.3 所示：

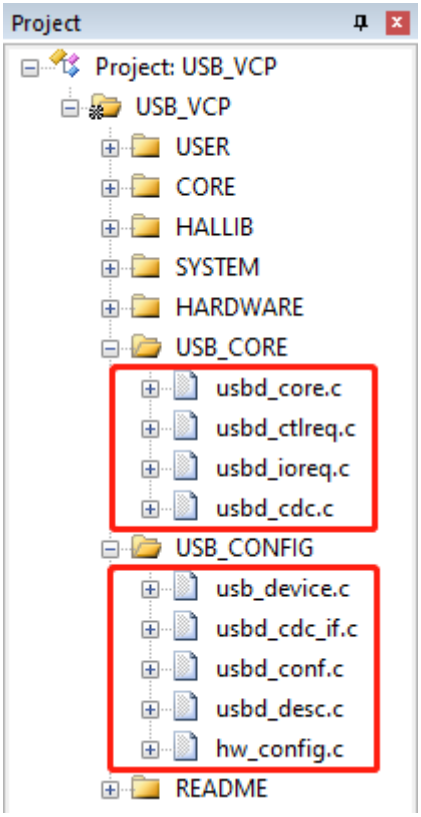


图 48.3.3 添加 USB 驱动等相关代码

USB 相关代码，请大家直接参考例程。注意，本例程有些是经过修改了的，并非完全照搬官方例程。

最后在 main.c 里面，我们修改 main 函数如下：

```
int main(void)
{
    u16 t;
    u16 len;
    u16 times=0;
    u8 usbstatus=0;
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M，9 倍频 8Mhzx9 = 72Mhz
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    LCD_Init();                             //初始化 LCD FSMC 接口
    POINT_COLOR=RED;                        //设置字体为红色
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    LCD_ShowString(30,70,200,16,16,"USB Virtual USART TEST");
    LCD_ShowString(30,90,200,16,16,"JASONZHANG@ALIENTEK");
    LCD_ShowString(30,110,200,16,16," 2019/10/10");
    LCD_ShowString(30,130,200,16,16,"USB Connecting...");//提示 USB 开始连接
    USB_Reset();//USB 断开再重连
```

```
MX_USB_DEVICE_Init();//USB 设备初始化
LCD_ShowString(30,150,200,16,16,"USB inited...");
while(1)
{
    if(usbstatus!=USB_GetStatus())
    {
        usbstatus = USB_GetStatus();//记录新的状态
        if(usbstatus==USBD_STATE_CONFIGURED)
        {
            POINT_COLOR=BLUE;
            LCD_ShowString(30,170,200,16,16,"USB Connected      ");
                                                    //提示 USB 连接成功
            LED1=0;//DS1 亮
        } else
        {
            POINT_COLOR=RED;
            LCD_ShowString(30,170,200,16,16,"USB disConnected ");//提示 USB 断开
            LED1=1;//DS1 灭
        }
    }
    if(USB_USART_RX_STA&0x8000)
    {
        len=USB_USART_RX_STA&0x3FFF;//得到此次接收到的数据长度
        USB_Printf("\r\n 您发送的消息长度为:%d\r\n",len);
        for(t=0; t<len; t++)
        {
            USB_USART_SendData(&USB_USART_RX_BUF[t]);//字符发送
        }
        USB_Printf("\r\n");//插入换行
        USB_USART_RX_STA=0;
    } else
    {
        times++;
        if(times%5000==0)
        {
            USB_Printf("\r\n 战舰 STM32 开发板 USB 虚拟串口实验\r\n");
            USB_Printf("正点原子@ALIENTEK\r\n\r\n");
        }
        if(times%200==0) {
            USB_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);
```

```
            }
        }
    }
```

软件设计部分，就给大家介绍到这里。

48.4 下载验证

本例程的测试，需要在电脑上先安装 ST 提供的 USB 虚拟串口驱动软件，该软件路径：光盘→6，软件资料→1，软件→STM32 USB 虚拟串口驱动→ VCP_V1.4.0_Setup.exe，双击安装即可。

然后，在代码编译成功之后，我们下载代码到精英 STM32 V1 上，然后将 USB 数据线，插入 USB_SLAVE 口，连接电脑和开发板（**注意：不是插 USB_232 端口！**），此时电脑会提示找到新硬件，并自动安装驱动。不过，如果自动安装不成功（有惊叹号），如图 48.4.1 所示：



图 48.4.1 自动安装失败

此时，我们可手动选择驱动（以 WIN7 为例），进行安装，在如图 48.4.1 所示的条目上面，右键→更新驱动程序软件→浏览计算机以查找驱动程序软件→浏览，选择 STM32 虚拟串口的驱动的路径为：C:\Program Files (x86)\STMicroelectronics\Software\Virtual comport driver\WIN7，然后点击下一步，即可完成安装。安装完成后，可以看到设备管理器里面多出了一个 STM32 的虚拟串口，如图 48.4.2 所示：



图 48.4.2 发现 STM32 USB 虚拟串口

如图 48.4.2，STM32 通过 USB 虚拟的串口，被电脑识别了，端口号为：COM5（可变），字符串名字为：STMicroelectronics Virtual COM Port（固定）。此时，开发板的 DS1 常亮，同时，开发板的 LCD 显示 USB Connected，如图 48.4.3 所示：

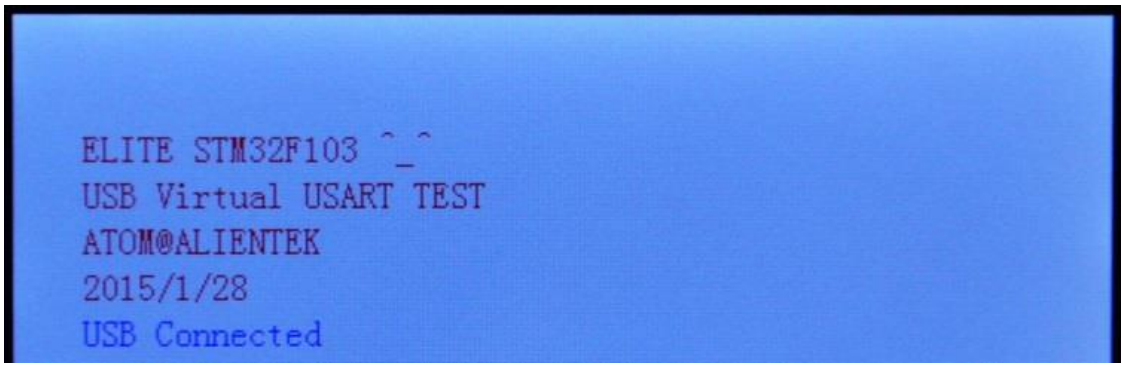


图 48.4.3 USB 虚拟串口连接成功

然后我们打开 XCOM，选择 COM5（需根据自己的电脑识别到的串口号选择），并打开串口（注意：波特率可以随意设置），就可以进行测试了，如图 48.4.4 所示：

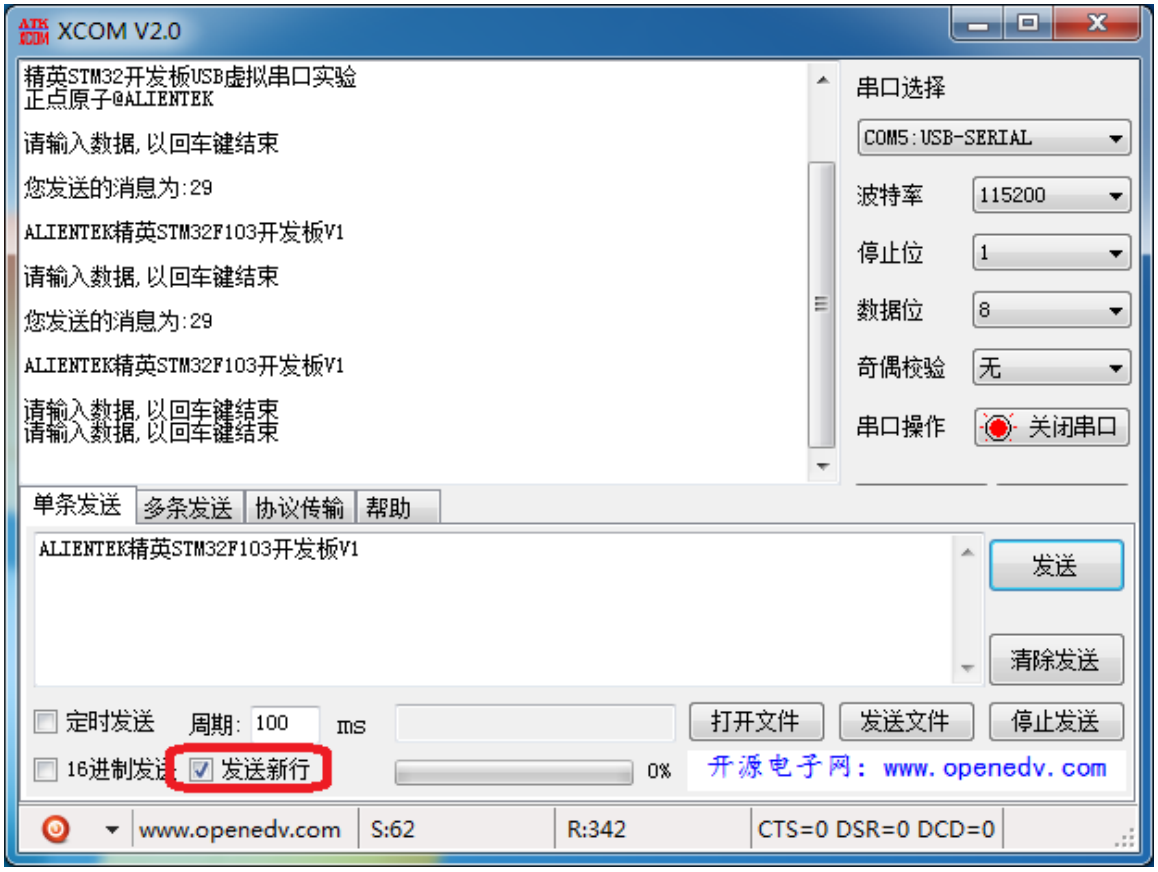


图 48.4.4 STM32 虚拟串口通信测试

可以看到，我们的串口调试助手，收到了来自 STM32 开发板的数据，同时，按发送按钮（串口助手必须勾选：发送新行），也可以收到电脑发送给 STM32 的数据（原样返回），说明我们的实验是成功的。实验现象同第九章完全一样。

至此，USB 虚拟串口实验就完成了，通过本实验，我们就可以利用 STM32 的 USB，直接和电脑进行数据互传了，具有广泛的应用前景。

第四十九章 USB 读卡器实验

上一章我们向大家介绍了如何利用 STM32 的 USB 来做一个 USB 虚拟串口，本章我们将利用 STM32 的 USB 来做一个 USB 读卡器。本章分为如下几个部分：

49.1 USB 读卡器简介
49.2 硬件设计
49.3 软件设计
49.4 下载验证

49.1 USB 读卡器简介

ALIENTEK 精英 STM32 开发板板载了一个 SD 卡插槽，可以用来接入 SD 卡，另外精英 STM32 开发板还板载了一个 16M 字节的 SPI FLASH 芯片，通过 STM32 的 USB 接口，我们可以实现一个简单的 USB 读卡器，来读写 SD 卡和 SPI FLASH。

本章我们还是通过移植官方的 USB Mass_Storage 例程来实现，该例程在：光盘\8，STM32 参考资料\2，STM32 USB 学习资料\ STM32_USB-FS-Device_Lib_V4.0.0\Projects\ Mass_Storage 下可以找到，注意：这里我们同样并非完全照搬官网的例程，有部分代码是被我们修改过的，以支持我们的应用。

USB Mass Storage 类支持两个传输协议：

1）Bulk-Only 传输（BOT）

2）Control/Bulk/Interrupt 传输（CBI）

Mass Storage 类规范定义了两个类规定的请求：Get_Max_LUN 和 Mass Storage Reset，所有的 Mass Storage 类设备都必须支持这两个请求。

Get_Max_LUN（bmRequestType= 10100001b and bRequest= 11111110b）用来确认设备支持的逻辑单元数。Max LUN 的值必须是 0~15。注意：LUN 是从 0 开始的。主机不能向不存在的 LUN 发送 CBW，本章我们定义 Max LUN 的值为 1，即代表 2 个逻辑单元。

Mass Storage Reset（bmRequestType=00100001b and bRequest= 11111111b）用来复位 Mass Storage 设备及其相关接口。

支持 BOT 传输的 Mass Storage 设备接口描述符要求如下：

接口类代码 bInterfaceClass=08h，表示为 Mass Storage 设备；

接口类子代码 bInterfaceSubClass=06h，表示设备支持 SCSI Primary Command-2（SPC-2）；

协议代码 bInterfaceProtocol 有 3 种：0x00、0x01、0x50，前两种需要使用中断传输，最后一种仅使用批量传输（BOT）。

支持 BOT 的设备必须支持最少 3 个 endpoint：Control，Bulk-In 和 Bulk-Out。USB2.0 的规范定义了控制端点 0。Bulk-In 端点用来从设备向主机传送数据（本章用端点 1 实现）。Bulk-Out 端点用来从主机向设备传送数据（本章用端点 2 实现）。

ST 官方的例程是通过 USB 来读写 SD 卡（SDIO 方式）和 NAND FALSH，支持 2 个逻辑单元，我们在官方例程的基础上，只需要修改 SD 驱动部分代码，并将对 NAND FLASH 的操作修改为对 SPI FLASH 的操作。只要这两步完成了，剩下的就比较简单了，对底层磁盘的读写，都是在 mass_mal.c 文件实现的，所以我们只需要修改该函数的 MAL_Init、MAL_Write、MAL_Read 和 MAL_GetStatus 等 4 个函数，使之与我们的 SD 卡和 SPI FLASH 对应起来即可。

49.2 硬件设计

本节实验功能简介：开机的时候先检测 SD 卡和 SPI FLASH 是否存在，如果存在则获取其容量，并显示在 LCD 上面（如果不存在，则报错）。之后开始 USB 配置，在配置成功之后就可以在电脑上发现两个可移动磁盘。我们用 DS1 来指示 USB 正在读写 SD 卡，并在液晶上显示出来，同样我们还是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1

2） 串口

3） TFTLCD 模块

4） USB SLAVE 接口

5） SD 卡

6） SPI FLASH

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。

49.3 软件设计

本章，我们在第四十章（实验 35）的基础上修改，首先打开实验 35 的工程，在 HARDWARE 文件夹所在的文件夹下新建一个 USB 的文件夹，并拷贝官方 USB 驱动库相关代码到该文件夹下即拷贝：en.stm32cubef1\STM32Cube_FW_F1_V1.8.0\Middlewares\ST 文件夹下的 STM32_USB_Device_Library 文件夹到该文件夹下面。

然后，在 USB 文件夹下面新建 CONFIG 文件夹，用来存放 USB 读卡器实现的相关代码，拷贝自：STM32Cube_FW_F1_V1.8.0\Projects\STM3210E_EVAL\Applications\USB_Device\MSC_Standalone 文件夹。注意：部分代码是有修改的，并非完全照抄官方代码，具体代码我们就不细说了（详见光盘本例程源码）。

最后，在 main.c 里面，我们修改 main 函数如下：

```
extern vu8 USB_STATUS_REG;                  //USB 状态
extern vu8 bDeviceState;                    //USB 连接 情况
extern USBD_HandleTypeDef hUsbDeviceFS;
extern SD_HandleTypeDef hsd;                //SD 卡句柄
int main(void)
{
    u8 offline_cnt=0;
    u8 tct=0;
    u8 USB_STA;
    u8 Device_STA;
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);         //设置时钟,72M
    delay_init(72);                         //初始化延时函数
    uart_init(115200);                      //初始化串口
    LED_Init();                             //初始化 LED
    LCD_Init();                             //初始化 LCD FSMC 接口
    W25QXX_Init();                          //初始化 W25Q128
    my_mem_init(SRAMIN);                    //初始化内部内存池
    POINT_COLOR=RED;                        //设置字体为红色
    LCD_ShowString(30,50,200,16,16," ELITE STM32");
    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,"2019/10/10");
    if(SD_Init())//SD 卡初始化
        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");
        SD_GetCardInfo(&SDCardInfo);       //获取 SD 卡信息
```

```
        Mass_Memory_Size[1]=(uint64_t)(SDCardInfo.LogBlockNbr)*
                            (uint64_t)(SDCardInfo.LogBlockSize); //计算 SD 卡容量
        Mass_Block_Size[1] =MSC_MEDIA_PACKET;//因为我们在 Init 里面设置了 SD 卡
                                    //的操作字节为 512 个,所以这里一定是 512 个字节.
                                   // 宏定义名称为 MSC_MEDIA_PACKET
        Mass_Block_Count[1]=Mass_Memory_Size[1]/Mass_Block_Size[1];
        LCD_ShowNum(134,130,Mass_Memory_Size[1]>>20,5,16);    //显示 SD 卡容量
    }
    if(W25QXX_TYPE!=W25Q128)
        LCD_ShowString(30,150,200,16,16,"W25Q128 Error!");   //检测 W25Q128 错误
    else //SPI FLASH  正常
    {
        Mass_Memory_Size[0]=1024*1024*12;//前 12M 字节
        Mass_Block_Size[0] =512;                //设置 SPI FLASH 的操作扇区大小为 512
        Mass_Block_Count[0]=Mass_Memory_Size[0]/Mass_Block_Size[0];
        LCD_ShowString(30,150,200,16,16,"SPI FLASH Size:12MB");
    }
    LCD_ShowString(30,170,200,16,16,"USB Connecting...");//提示 USB 开始连接
    USB_Reset();
    MX_USB_DEVICE_Init();//usb 初始化
    LCD_ShowString(30,190,200,16,16,"USB inited...");
    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
            {
                LCD_ShowString(30,190,200,16,16,"USB Writing...");
                //提示 USB 正在写入数据
            }
            if(USB_STATUS_REG&0x02)//正在读
            {
                LCD_ShowString(30,190,200,16,16,"USB Reading...");
                //提示 USB 正在读出数据
            }
            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(Device_STA!=USB_GetStatus())
        {
            Device_STA = USB_GetStatus();//记录新的状态
            if(Device_STA==USBD_STATE_CONFIGURED)
            {
                POINT_COLOR=BLUE;
                LCD_ShowString(30,170,200,16,16,"USB Connected      ");
                //提示 USB 连接成功
            } else
            {
                POINT_COLOR=RED;
                LCD_ShowString(30,170,200,16,16,"USB disConnected ");//提示 USB 断开
            }
        }
        tct++;
        if(tct==200)
        {
            tct=0;
            LED0=!LED0;//提示系统在运行
            if(USB_STATUS_REG&0x10)
            {
                offline_cnt=0;//USB 连接了,则清除 offline 计数器
                bDeviceState=USBD_STATE_CONFIGURED;
            } else//没有得到轮询
            {
                offline_cnt++;
                if(offline_cnt>10)
                {
                    bDeviceState=USBD_STATE_SUSPENDED;//2s 内没收到在线标记,
                                                      //代表 USB 被拔出了
                }
            }
            USB_STATUS_REG=0;
        }
    }
}
```

在 main 函数里面，我们将 SPI FLASH 的最开始 12M 地址范围用作 SPI FLASH Disk，也就是文件系统管理的范围大小，这个我们在之前的 FATFS 实验也介绍过。

通过此部分代码就可以实现了我们之前在硬件设计部分描述的功能，这里我们用到了一个全局变量 USB_STATUS_REG，用来标记 USB 的相关状态，这样我们就可以在液晶上显示当前

USB 的状态了。

软件设计部分就为大家介绍到这里。

49.4 下载验证

在代码编译成功之后，我们通过下载代码精英 STM32F103 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，要插在开发板的 USB_SLAVE 口！而不是 USB_232 端口，且 P6 必须用跳线帽连接 PA11 和 D-以及 PA12 和 D+!!**），LCD 显示效果如图 49.4.1 所示：

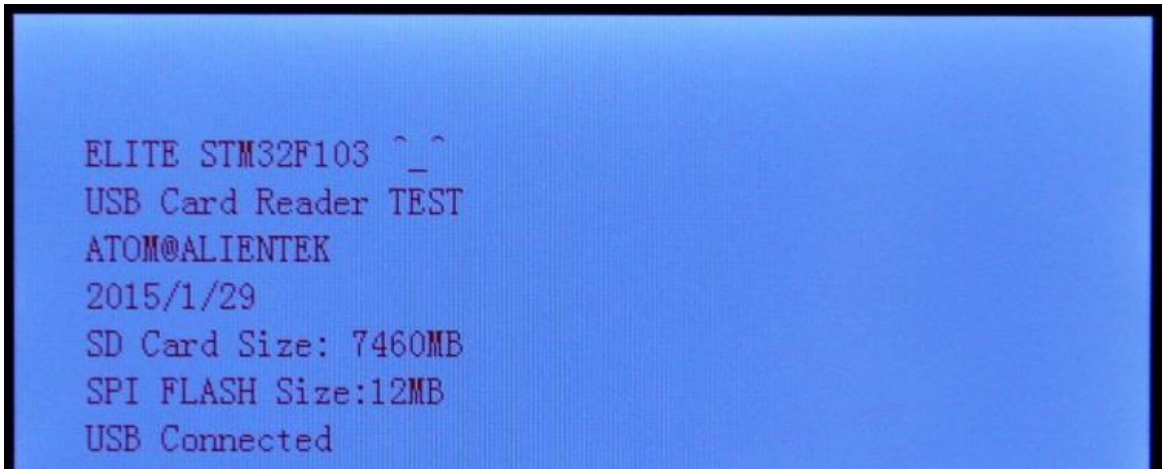


图 49.4.1 USB 连接成功

此时，电脑提示发现新硬件，并自动安装驱动，如图 49.4.2 所示：



图 49.4.2 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 49.4.3 所示：

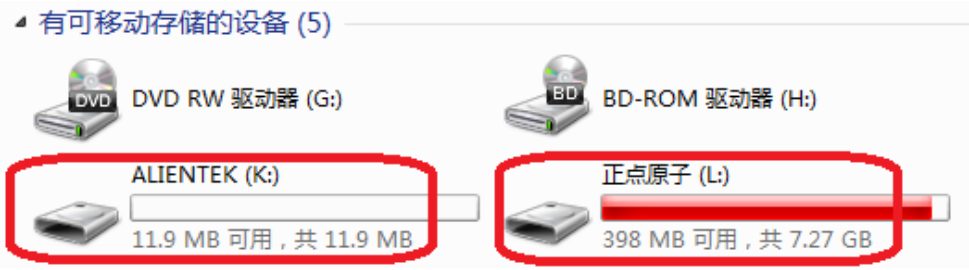


图 49.4.3 电脑找到 USB 读卡器的两个盘符

我们打开设备管理器，在通用串行总线控制器里面可以发现多出了一个 USB Mass Storage Device，同时看到磁盘驱动器里面多了 2 个磁盘，如图 49.4.4 所示：



图 49.4.4 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：

50.1 UCOSII 简介
50.2 硬件设计
50.3 软件设计
50.4 下载验证

50.1 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 嵌人到开发的产品中。UCOSII 具有执行效率高、占用空间小、实时性能优良和可扩展性强等特点， 最小内核可编译至 2KB 。UCOSII 已经移植到了几乎所有知名的 CPU 上。

UCOSII 构思巧妙。结构简洁精练，可读性强，同时又具备了实时操作系统的全部功能，虽然它只是一个内核，但非常适合初次接触嵌入式实时操作系统的朋友，可以说是麻雀虽小，五脏俱全。UCOSII（V2.91 版本）体系结构如图 50.1.1 所示：

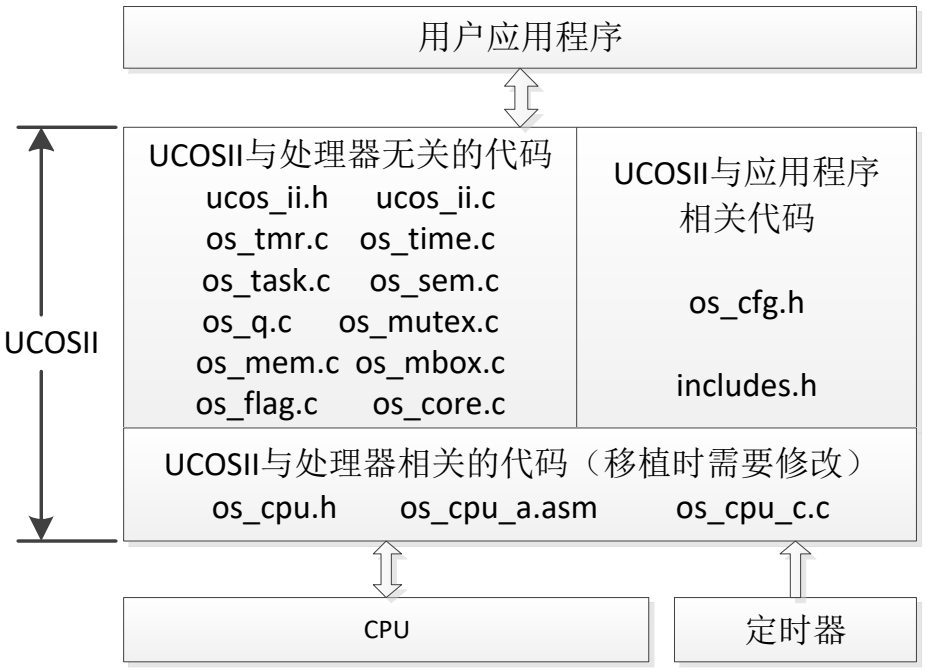


图 50.1.1 UCOSII 体系结构图

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 STM32F1 的 SYSTICK 定时器来提供 UCOSII 时钟节拍。

关于 UCOSII 在 STM32F1 的详细移植过程，请参考光盘资料：《ALIENTEK STM32F1 UCOS 开发手册.pdf》，教程路径：光盘→ STM32F1 UCOS 开发手册.pdf。这里我们就不详细介绍了。

UCOSII 早期版本只支持 64 个任务，但是从 2.80 版本开始，支持任务数提高到 255 个，不过对我们来说一般 64 个任务都是足够多了，一般很难用到这么多个任务。UCOSII 保留了最高 4 个优先级和最低 4 个优先级的总共 8 个任务，用于拓展使用，单实际上，UCOSII 一般只占用了最低 2 个优先级，分别用于空闲任务（倒数第一）和统计任务（倒数第二），所以剩下给我们使用的任务最多可达 255-2=253 个（V2.91）。

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。

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。

UCOSII 的任何任务都是通过一个叫任务控制块（TCB）的东西来控制的，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。

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。

任务的调度其实就是 CPU 运行环境的切换，即：PC 指针、SP 指针和寄存器组等内容的存取过程，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 50.1.2 所示：

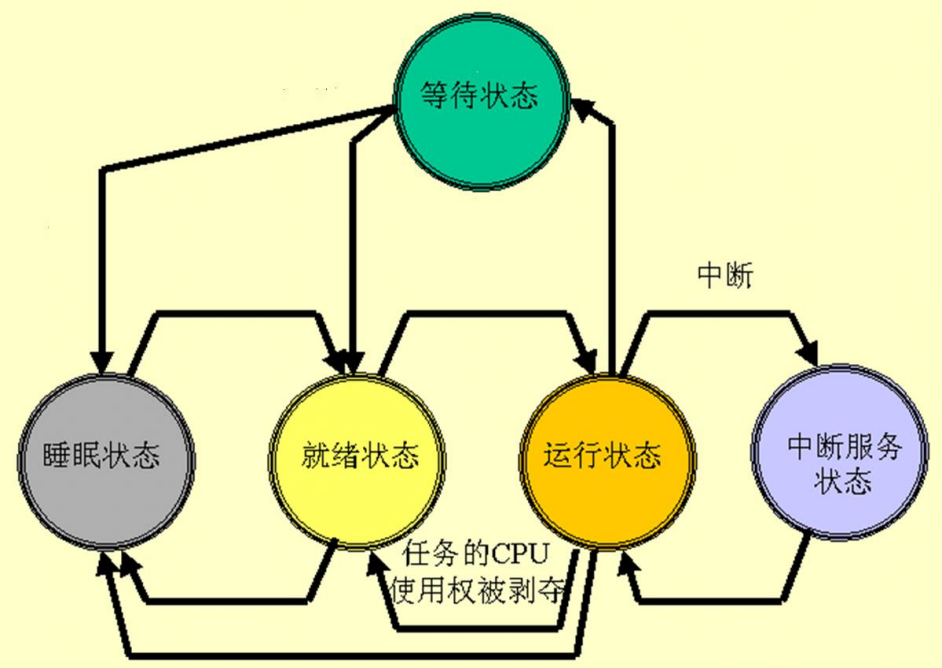


图 50.1.2 UCOSII 任务状态转换关系

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，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)。

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 STM32F1 上面运行 UCOSII 的步骤：

1） 移植 UCOSII

要想 UCOSII 在 STM32F1 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：STM32F1 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_OS 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，在如果堆栈设置小了，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 STM32F1 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。

50.2 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。

50.3 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 50.3.1 所示：

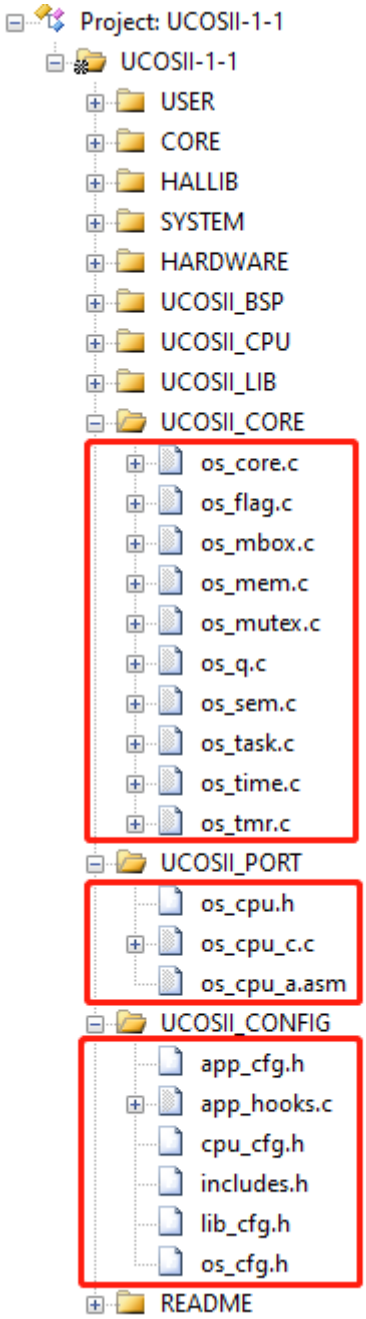


图 50.3.1 添加 UCOSII 源码后的工程

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_OS 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 级的延时在很多时候非常有用：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_OS 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 OS 时用到
void SysTick_Handler(void)
{
    if(delay_osrunning==1)                  //OS 开始跑了,才执行正常的调度处理
    {
        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），注意，这里只有在 OS 开始运行以后（delay_osrunning 为 1），才开始调用 OSTimeTick 等函数。

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)
{
    HAL_Init();                                        //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);                    //设置时钟,72M
    delay_init(72);                                    //初始化延时函数
    LED_Init();                                        //初始化 LED
    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);
        };
    }
```

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。

50.4 下载验证

在代码编译成功之后，我们通过下载代码到精英 STM32 开发板上，可以看到 DS0 一秒钟闪一次，而 DS1 则以固定的频率闪烁，说明两个任务（led0_task 和 led1_task）都已经正常运行了，符合我们预期的设计。

50.5 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 52 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：

```
//START 任务
//设置任务优先级
#define START_TASK_PRIO          10   //开始任务的优先级为最低
//设置任务堆栈大小
#define START_STK_SIZE           128
//任务任务堆栈
OS_STK START_TASK_STK[START_STK_SIZE];
//任务函数
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             128
//任务堆栈
OS_STK LED_TASK_STK[LED_STK_SIZE];
//任务函数
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            128
//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             128
//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)
{
    HAL_Init();                          //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);      //设置时钟,72M
    delay_init(72);                      //初始化延时函数
```

```
    LED_Init();                                 //初始化 LED
    KEY_Init();                                 //初始化按键
    BEEP_Init();                                //初始化蜂鸣器
    OSInit();                                   //UCOS 初始化
    OSTaskCreateExt((void(*)(void*) )start_task,          //任务函数
                    (void*              )0,               //传递给任务函数的参数
                    (OS_STK*            )&START_TASK_STK[START_STK_SIZE-1],
                    //任务堆栈栈顶
                    (INT8U              )START_TASK_PRIO,   //任务优先级
                    (INT16U             )START_TASK_PRIO,
                    //任务 ID，这里设置为和优先级一样
                    (OS_STK*            )&START_TASK_STK[0],   //任务堆栈栈底
                    (INT32U             )START_STK_SIZE,          //任务堆栈大小
                    (void*              )0,                       //用户补充的存储区
                    (INT16U             )OS_TASK_OPT_STK_CHK|
                    OS_TASK_OPT_STK_CLR|OS_TASK_OPT_SAVE_FP);
                    //任务选项,为了保险起见，所有任务都保存浮点寄存器的值
    OSStart(); //开始任务
}
//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    pdata=pdata;
    OSStatInit();   //开启统计任务
    OS_ENTER_CRITICAL();   //进入临界区(关闭中断)
    //LED 任务
    OSTaskCreateExt((void(*)(void*) )led_task,
                    (void*              )0,
                    (OS_STK*            )&LED_TASK_STK[LED_STK_SIZE-1],
                    (INT8U              )LED_TASK_PRIO,
                    (INT16U             )LED_TASK_PRIO,
                    (OS_STK*            )&LED_TASK_STK[0],
                    (INT32U             )LED_STK_SIZE,
                    (void*              )0,
                    (INT16U             )OS_TASK_OPT_STK_CHK|
                    OS_TASK_OPT_STK_CLR|OS_TASK_OPT_SAVE_FP);
    //BEEP 任务
    OSTaskCreateExt((void(*)(void*) )beep_task,
                    (void*              )0,
                    (OS_STK*            )&BEEP_TASK_STK[BEEP_STK_SIZE-1],
                    (INT8U              )BEEP_TASK_PRIO,
                    (INT16U             )BEEP_TASK_PRIO,
```

```
                        (OS_STK*        )&BEEP_TASK_STK[0],
                        (INT32U         )BEEP_STK_SIZE,
                        (void*          )0,
                        (INT16U         )OS_TASK_OPT_STK_CHK|
                        OS_TASK_OPT_STK_CLR|OS_TASK_OPT_SAVE_FP);
    //按键任务
    OSTaskCreateExt((void(*)(void*) )key_task,
                        (void*          )0,
                        (OS_STK*        )&KEY_TASK_STK[KEY_STK_SIZE-1],
                        (INT8U          )KEY_TASK_PRIO,
                        (INT16U         )KEY_TASK_PRIO,
                        (OS_STK*        )&KEY_TASK_STK[0],
                        (INT32U         )KEY_STK_SIZE,
                        (void*          )0,
                        (INT16U         )OS_TASK_OPT_STK_CHK|
                        OS_TASK_OPT_STK_CLR|OS_TASK_OPT_SAVE_FP);
    OS_EXIT_CRITICAL();                 //退出临界区(开中断)
    OSTaskSuspend(START_TASK_PRIO); //挂起开始任务
}
//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 TaskLed
        }
        BEEP=1;                             //打开蜂鸣器
        delay_ms(60);
        BEEP=0;                             //关闭蜂鸣器
        delay_ms(940);
```

```
        }
    }
    //按键扫描任务
    void key_task(void *pdata)
    {
        u8 key;
        while(1)
        {
            key=KEY_Scan(0);
            if(key==KEY0_PRES)
            {
                OSTaskSuspend(LED_TASK_PRIO);     //挂起 LED 任务，LED 停止闪烁
            }
            else if (key==KEY2_PRES)
            {
                OSTaskResume(LED_TASK_PRIO);   //恢复 LED 任务，LED 恢复闪烁
            }
            else if (key==WKUP_PRES)
            {
                OSTaskDelReq(BEEP_TASK_PRIO);
                //发送删除 BEEP 任务请求，任务睡眠，无法恢复
            }
            else if(key==KEY1_PRES)
            {
                //重新创建任务 beep
                OSTaskCreateExt((void(*)(void*) )beep_task,
                                (void*               )0,
                                (OS_STK*             )&
                                BEEP_TASK_STK[BEEP_STK_SIZE-1],
                                (INT8U               )BEEP_TASK_PRIO,
                                (INT16U              )BEEP_TASK_PRIO,
                                (OS_STK*             )&BEEP_TASK_STK[0],
                                (INT32U              )BEEP_STK_SIZE,
                                (void*               )0,
                                (INT16U              )OS_TASK_OPT_STK_CHK|
                                OS_TASK_OPT_STK_CLR|OS_TASK_OPT_SAVE_FP);
            }
            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。

第五十一章 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：

51.1 UCOSII 信号量和邮箱简介
51.2 硬件设计
51.3 软件设计
51.4 下载验证

51.1 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 51.1.1 所示：

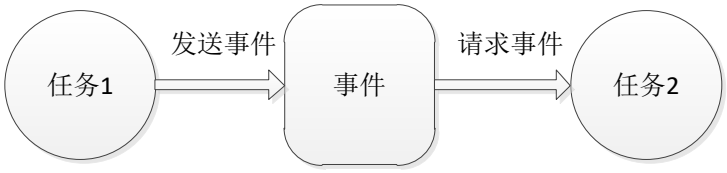


图 51.1.1 两个任务使用事件进行通信的示意图

在图 51.1.1 中任务 1 是发信方，任务 2 是收信方。任务 1 负责把信息发送到事件上，这项操作叫做发送事件。任务 2 通过读取事件操作对事件进行查询：如果有信息则读取，否则等待。读事件操作叫做请求事件。

为了把描述事件的数据结构统一起来，UCOSII 使用叫做事件控制块(ECB)的数据结构来描述诸如信号量、邮箱（消息邮箱）和消息队列这些事件。事件控制块中包含包括等待任务表在内的所有有关事件的数据，事件控制块结构体定义如下：

```
typedef struct
{
    INT8U    OSEventType;              //事件的类型
    INT16U OSEventCnt;                 //信号量计数器
    void *OSEventPtr;                  //消息或消息队列的指针
    INT8U    OSEventGrp;               //等待事件的任务组
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE];//任务等待表
#if OS_EVENT_NAME_EN > 0u
     INT8U     *OSEventName;           //事件名
#endif
} OS_EVENT;
```

信号量

信号量是一类事件。使用信号量的最初目的，是为了给共享资源设立一个标志，该标志表示该共享资源的占用情况。这样，当一个任务在访问共享资源之前，就可以先对这个标志进行查询，从而在了解资源被占用的情况之后，再来决定自己的行为。

信号量可以分为两种：一种是二值型信号量，另外一种是 N 值信号量。

二值型信号量好比家里的座机，任何时候，只能有一个人占用。而 N 值信号量，则好比公共电话亭，可以同时有多个人（N 个）使用。

UCOSII 将二值型信号量称之为也叫互斥型信号量，将 N 值信号量称之为计数型信号量，也就是普通的信号量。本章，我们介绍的是普通信号量，互斥型信号量的介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》5.4 节。

接下来我们看看在 UCOSII 中，与信号量相关的几个函数（未全部列出，下同）。

1） 创建信号量函数

在使用信号量之前，我们必须用函数 OSSemCreate 来创建一个信号量，该函数的原型为：OS_EVENT *OSSemCreate (INT16U cnt)。该函数返回值为已创建的信号量的指针，而参数 cnt 则是信号量计数器（OSEventCnt）的初始值。

2） 请求信号量函数

任务通过调用函数 OSSemPend 请求信号量，该函数原型如下：void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)。其中，参数 pevent 是被请求信号量的指针，timeout 为等待时限，err 为错误信息。

为防止任务因得不到信号量而处于长期的等待状态，函数 OSSemPend 允许用参数 timeout 设置一个等待时间的限制，当任务等待的时间超过 timeout 时可以结束等待状态而进入就绪状态。如果参数 timeout 被设置为 0，则表明任务的等待时间为无限长。

3） 发送信号量函数

任务获得信号量，并在访问共享资源结束以后，必须要释放信号量，释放信号量也叫做发送信号量，发送信号通过 OSSemPost 函数实现 。OSSemPost 函数在对信号量的计数器操作之前，首先要检查是否还有等待该信号量的任务。如果没有，就把信号量计数器 OSEventCnt 加一；如果有，则调用调度器 OS_Sched( )去运行等待任务中优先级别最高的任务。函数 OSSemPost 的原型为：INT8U OSSemPost(OS_EVENT *pevent)。其中，pevent 为信号量指针，该函数在调用成功后，返回值为 OS_ON_ERR，否则会根据具体错误返回 OS_ERR_EVENT_TYPE、OS_SEM_OVF。

4） 删除信号量函数

应用程序如果不需要某个信号量了，那么可以调用函数 OSSemDel 来删除该信号量，该函数的原型为：OS_EVENT *OSSemDel (OS_EVENT *pevent,INT8U opt, INT8U *err)。其中，pevent 为要删除的信号量指针，opt 为删除条件选项，err 为错误信息。

邮箱

在多任务操作系统中，常常需要在任务与任务之间通过传递一个数据（这种数据叫做“消息”）的方式来进行通信。为了达到这个目的，可以在内存中创建一个存储空间作为该数据的缓冲区。如果把这个缓冲区称之为消息缓冲区，这样在任务间传递数据（消息）的最简单办法就是传递消息缓冲区的指针。我们把用来传递消息缓冲区指针的数据结构叫做邮箱（消息邮箱）。

在 UCOSII 中，我们通过事件控制块的 OSEventPrt 来传递消息缓冲区指针，同时使事件控制块的成员 OSEventType 为常数 OS_EVENT_TYPE_MBOX，则该事件控制块就叫做消息邮箱。

接下来我们看看在 UCOSII 中，与消息邮箱相关的几个函数。

1） 创建邮箱函数

创建邮箱通过函数 OSMboxCreate 实现，该函数原型为：OS_EVENT *OSMboxCreate (void *msg)。函数中的参数 msg 为消息的指针，函数的返回值为消息邮箱的指针。

调用函数 OSMboxCreate 需先定义 msg 的初始值。在一般的情况下，这个初始值为 NULL；但也可以事先定义一个邮箱，然后把这个邮箱的指针作为参数传递到函数 OSMboxCreate 中，使之一开始就指向一个邮箱。

2） 向邮箱发送消息函数

任务可以通过调用函数 OSMboxPost 向消息邮箱发送消息，这个函数的原型为：INT8U

OSMboxPost (OS_EVENT *pevent,void *msg)。其中 pevent 为消息邮箱的指针，msg 为消息指针。

3） 请求邮箱函数

当一个任务请求邮箱时需要调用函数 OSMboxPend，这个函数的主要作用就是查看邮箱指针 OSEventPtr 是否为 NULL，如果不是 NULL 就把邮箱中的消息指针返回给调用函数的任务，同时用 OS_NO_ERR 通过函数的参数 err 通知任务获取消息成功；如果邮箱指针 OSEventPtr 是 NULL，则使任务进入等待状态，并引发一次任务调度。

函数 OSMboxPend 的原型为：void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)。其中 pevent 为请求邮箱指针，timeout 为等待时限，err 为错误信息。

4） 查询邮箱状态函数

任务可以通过调用函数 OSMboxQuery 查询邮箱的当前状态。该函数原型为：INT8U OSMboxQuery(OS_EVENT *pevent,OS_MBOX_DATA *pdata)。其中 pevent 为消息邮箱指针，pdata 为存放邮箱信息的结构。

5） 删除邮箱函数

在邮箱不再使用的时候，我们可以通过调用函数 OSMboxDel 来删除一个邮箱，该函数原型为：OS_EVENT *OSMboxDel(OS_EVENT *pevent,INT8U opt,INT8U *err)。其中 pevent 为消息邮箱指针，opt 为删除选项，err 为错误信息。

关于 UCOSII 信号量和邮箱的介绍，就到这里。更详细的介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》第五章。

51.2 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） 3 个按键（KEY0/KEY1/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。

51.3 软件设计

本章，我们在第三十一章实验 （实验 26 ）的基础上修改。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了。

在加入 UCOSII 代码后，我们只需要修改 test.c 函数了，打开 test.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                  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
//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,"Elite STM32F1");
    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");
    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)
{
    HAL_Init();                                 //初始化 HAL 库
    Stm32_Clock_Init(RCC_PLL_MUL9);             //设置时钟,72M
    delay_init(72);                             //初始化延时函数
    uart_init(115200);                          //初始化 USART
    LED_Init();                                 //初始化 LED
    KEY_Init();                                 //初始化按键
    BEEP_Init();                                //初始化蜂鸣器
    LCD_Init();                                 //初始化 LCD
    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
}
//////////////////////////////////////////////////////////////////////////////////////////////
//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;
    u8 semmask=0;
    u8 tcnt=0;
    while(1)
    {
        key=(u32)OSMboxPend(msg_key,10,&err);
        switch(key)
        {
            case 1://控制 DS1
                LED1=!LED1;
                break;
            case 2://发送信号量
                semmask=1;
                OSSemPost(sem_beep);
                break;
            case 3://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 和 touch_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。

51.4 下载验证

在代码编译成功之后，我们通过下载代码到精英 STM32 开发板上，可以看到 LCD 显示界面如图 51.4.1 所示：

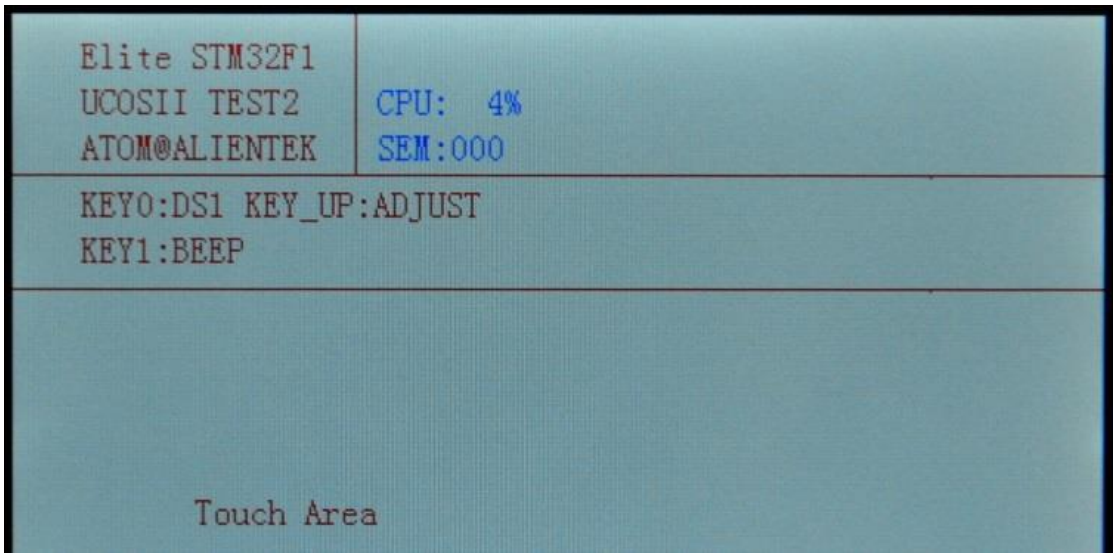


图 51.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率仅为 4%左右。此时通过在触摸区域画图，可以看到 CPU 使用率飙升(50%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 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）。

第五十二章 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：

52.1 UCOSII 消息队列、信号量集和软件定时器简介
52.2 硬件设计
52.3 软件设计
52.4 下载验证

52.1 UCOSII 消息队列、信号量集和软件定时器简介

上一章，我们介绍了信号量和邮箱的使用，本章我们介绍比较复杂消息队列、信号量集以及软件定时器的使用。

消息队列

使用消息队列可以在任务之间传递多条消息。消息队列由三个部分组成：事件控制块、消息队列和消息。当把事件控制块成员 OSEventType 的值置为 OS_EVENT_TYPE_Q 时，该事件控制块描述的就是一个消息队列。

消息队列的数据结构如图 52.1.1 所示。从图中可以看到，消息队列相当于一个共用一个任务等待列表的消息邮箱数组，事件控制块成员 OSEventPtr 指向了一个叫做队列控制块（OS_Q）的结构，该结构管理了一个数组 MsgTbl[]，该数组中的元素都是一些指向消息的指针。

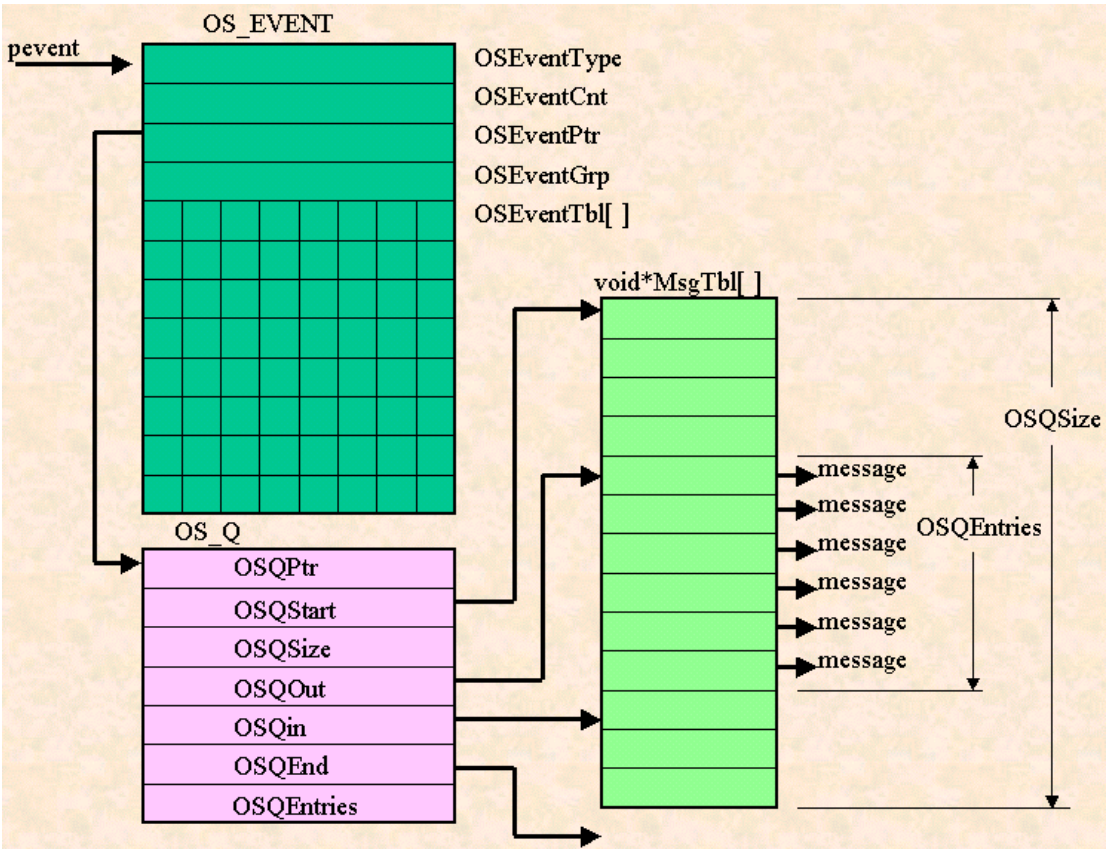


图 52.1.1 消息队列的数据结构

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 52.1.1 所示：

参数	说明
OSQPtr	指向下一个空的队列控制块
OSQSize	数组的长度
OSQEntres	已存放消息指针的元素数目
OSQStart	指向消息指针数组的起始地址
OSQEnd	指向消息指针数组结束单元的下一个单元。它使得数组构成了一个循环的缓冲区
OSQIn	指向插入一条消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元
OSQOut	指向被取出消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元

表 52.1.1 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 52.1.2 所示的循环的队列。

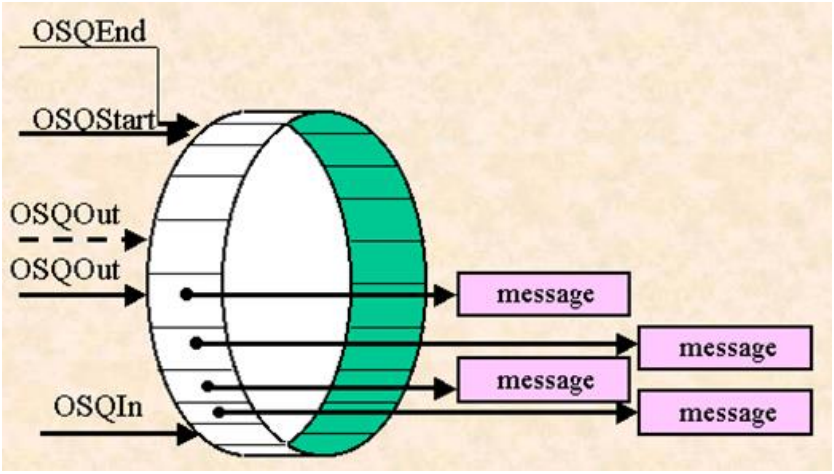


图 52.1.2 消息指针数组构成的环形数据缓冲区

在 UCOSII 初始化时，系统将按文件 os_cfg.h 中的配置常数 OS_MAX_QS 定义 OS_MAX_QS 个队列控制块，并用队列控制块中的指针 OSQPtr 将所有队列控制块链接为链表。由于这时还没有使用它们，故这个链表叫做空队列控制块链表。

接下来我们看看在 UCOSII 中，与消息队列相关的几个函数（未全部列出，下同）。

1） 创建消息队列函数

创建一个消息队列首先需要定义一指针数组，然后把各个消息数据缓冲区的首地址存入这个数组中，然后再调用函数 OSQCreate 来创建消息队列。创建消息队列函数 OSQCreate 的原型为：OS_EVENT *OSQCreate(void**start,INT16U size)。其中，start 为存放消息缓冲区指针数组的地址，size 为该数组大小。该函数的返回值为消息队列指针。

2） 请求消息队列函数

请求消息队列的目的是为了从消息队列中获取消息。任务请求消息队列需要调用函数 OSQPend，该函数原型为：void*OSQPend(OS_EVENT*pevent,INT16U timeout,INT8U *err)。

其中，pevent 为所请求的消息队列的指针，timeout 为任务等待时限，err 为错误信息。

3） 向消息队列发送消息函数

任务可以通过调用函数 OSQPost 或 OSQPostFront 两个函数来向消息队列发送消息。函数 OSQPost 以 FIFO（先进先出）的方式组织消息队列，函数 OSQPostFront 以 LIFO（后进先出）的方式组织消息队列。这两个函数的原型分别为：INT8U OSQPost(OS_EVENT *pevent,void *msg)和 INT8U 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 52.1.3 所示

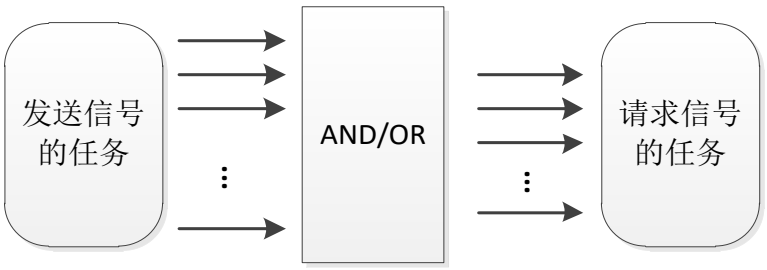


图 52.1.3 信号量集示意图

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 52.1.2 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 52.1.2 OSFlagNodeWaitType 可选值及其意义

OSFlagFlags、OSFlagNodeFlags、OSFlagNodeWaitType 三者的关系如图 52.1.4 所示：

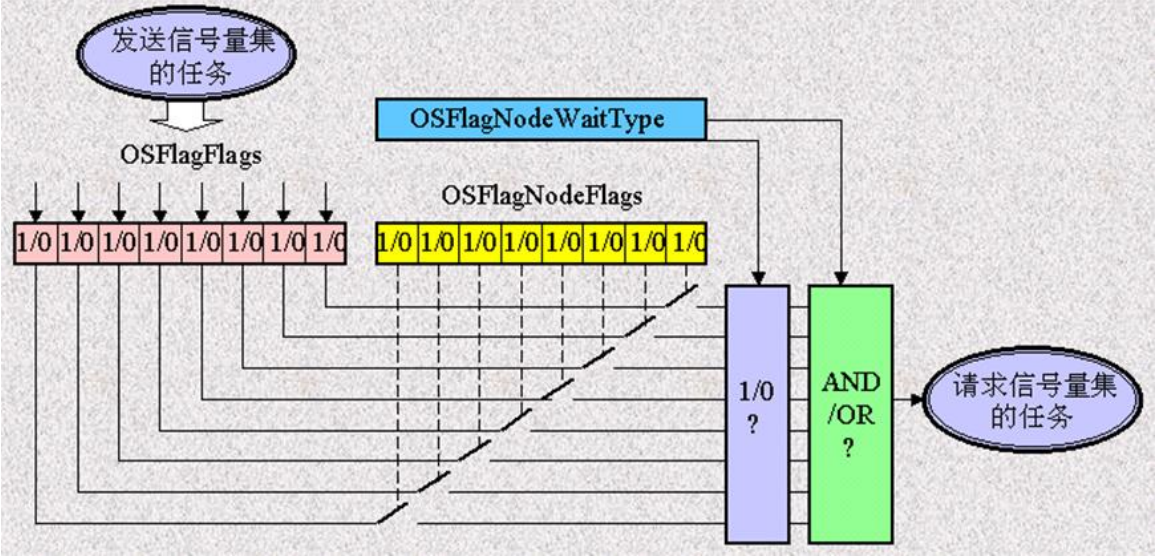


图 52.1.4 标志组与等待任务共同完成信号量集的逻辑运算及控制

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链表头指针。空闲态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针分别指向空闲控制块的前一个和后一个，组织了空闲控制块双向链表。建立定时器时，从这个链表中搜索空闲定时器控制块。

OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]：该数组的每个元素都是已开启定时器的一个分组，元素中记录了指向该分组中第一个定时器控制块的指针，以及定时器控制块的个数。运行态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针同样也组织了所在分组中定时器控制块的双向链表。软件定时器管理所需的数据结构示意图如图 52.1.5 所示：

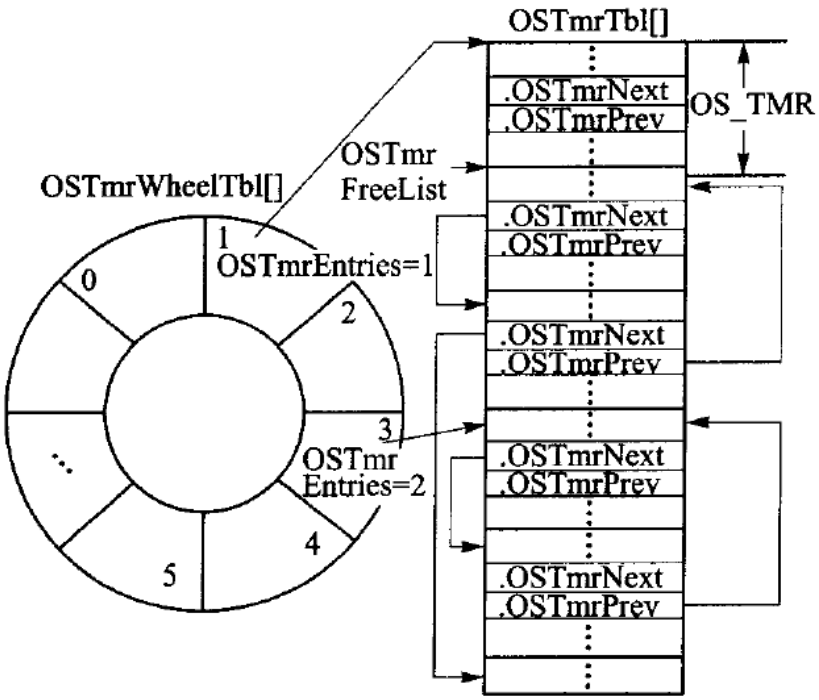


图 52.1.5 软件定时器管理所需的数据结构示意图

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～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～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 52.1.6 所示。

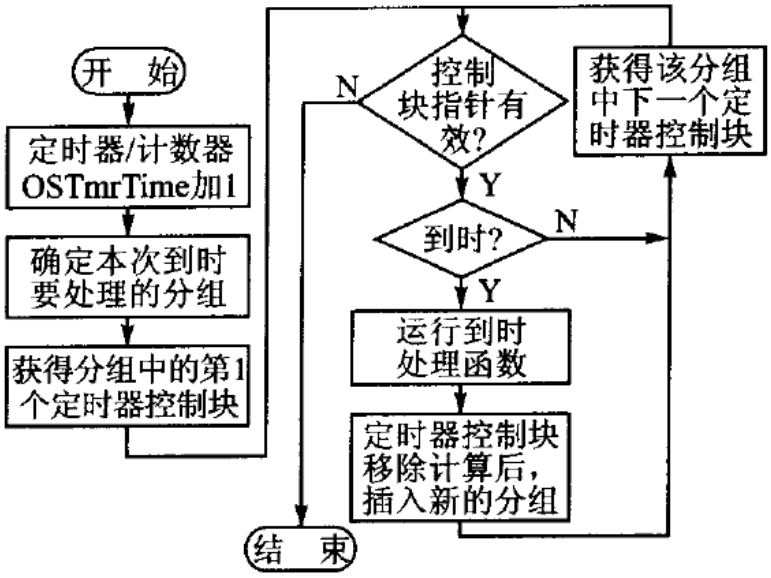


图 52.1.6 软件定时器管理任务流程

当运行完软件定时器的到时处理函数之后，需要进行该定时器控制块在链表中的移除和再插入操作。插入前需要重新计算定时器下次到时时所处的分组。计算公式如下：

定时器下次到时的 OSTmrTime 值(OSTmrMatch)=定时器定时值+当前 OSTmrTime 值

新分组=定时器下次到时的 OSTmrTime 值(OSTmrMatch)%OS_TMR_CFG_WHEEL_SIZE

接下来我们看看在 UCOSII 中，与软件定时器相关的几个函数。

1） 创建软件定时器函数

创建软件定时器通过函数 OSTmrCreate 实现，该函数原型为：OS_TMR *OSTmrCreate (INT32U dly, INT32U period, INT8U opt, OS_TMR_CALLBACK callback,void *callback_arg, INT8U *pname, INT8U *perr)。

dly，用于初始化定时时间，对单次定时（ONE-SHOT 模式）的软件定时器来说，这就是该定时器的定时时间，而对于周期定时（PERIODIC 模式）的软件定时器来说，这是该定时器第一次定时的时间，从第二次开始定时时间变为 period。

period，在周期定时（PERIODIC 模式），该值为软件定时器的周期溢出时间。

opt，用于设置软件定时器工作模式。可以设置的值为：OS_TMR_OPT_ONE_SHOT 或 OS_TMR_OPT_PERIODIC，如果设置为前者，说明是一个单次定时器；设置为后者则表示是周期定时器。

callback，为软件定时器的回调函数，当软件定时器的定时时间到达时，会调用该函数。

callback_arg，回调函数的参数。

pname，为软件定时器的名字。

perr，为错误信息。

软件定时器的回调函数有固定的格式，我们必须按照这个格式编写，软件定时器的回调函数格式为：void (*OS_TMR_CALLBACK)(void *ptmr, void *parg)。其中，函数名我们可以自己随意设置，而 ptmr 这个参数，软件定时器用来传递当前定时器的控制块指针，所以我们一般设置其类型为 OS_TMR*类型，第二个参数（parg）为回调函数的参数，这个就

可以根据自己需要设置了，你也可以不用，但是必须有这个参数。

2） 开启软件定时器函数

任务可以通过调用函数 OSTmrStart 开启某个软件定时器，该函数的原型为：BOOLEAN OSTmrStart (OS_TMR *ptmr, INT8U *perr)。其中 ptmr 为要开启的软件定时器指针，perr 为错误信息。

3） 停止软件定时器函数

任务可以通过调用函数 OSTmrStop 停止某个软件定时器，该函数的原型为：BOOLEAN OSTmrStop (OS_TMR *ptmr,INT8U opt,void *callback_arg,INT8U *perr)。

其中 ptmr 为要停止的软件定时器指针。

opt 为停止选项，可以设置的值及其对应的意义为：

OS_TMR_OPT_NONE，直接停止，不做任何其他处理

OS_TMR_OPT_CALLBACK，停止，用初始化的参数执行一次回调函数

OS_TMR_OPT_CALLBACK_ARG，停止，用新的参数执行一次回调函数

callback_arg，新的回调函数参数。

perr，错误信息。

软件定时器我们就介绍到这。

52.2 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） 3 个机械按键（KEY0/KEY1/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。

52.3 软件设计

本章，我们在第三十九章实验 （实验 33 ）的基础上修改，首先，是 UCOSII 代码的添加，具体方法同第 50 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 test.c 函数了，打开 test.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;
	u8 err;
	static u8 msg_cnt=0;	//msg 编号
	p=mymalloc(SRAMIN,13);	//申请 13 个字节的内存
	if(p)
	{
		sprintf((char*)p,"ALIENTEK %03d",msg_cnt);
		msg_cnt++;
		err=OSQPost(q_msg,p); //发送队列
		if(err!=OS_ERR_NONE)	//发送失败
		{
			myfree(SRAMIN,p);	//释放内存
			OSTmrStop(tmr3,OS_TMR_OPT_NONE,0,&err);	//关闭软件定时器 3
		}
	}
}
//加载主界面
void ucos_load_main_ui(void)
{
	LCD_Clear(WHITE);	//清屏
	POINT_COLOR=RED; //设置字体为红色
	LCD_ShowString(10,10,200,16,16,"Elite STM32F1");
	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");
	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)
    {
        HAL_Init();                                 //初始化 HAL 库
        Stm32_Clock_Init(RCC_PLL_MUL9);             //设置时钟,72M
        delay_init(72);                             //初始化延时函数
        uart_init(115200);                          //初始化 USART
        LED_Init();                                 //初始化 LED
        KEY_Init();                                 //初始化按键
        BEEP_Init();                                //初始化蜂鸣器
        LCD_Init();                                 //初始化 LCD
        TPAD_Init(6);                               //初始化触摸按键
        tp_dev.init();                              //初始化触摸屏
        my_mem_init(SRAMIN);                        //初始化内部内存池
        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://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 4://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,"KEY_UP DOWN");
            if(flags&0X0008)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 test.c 的代码有点多，因为我们创建了 7 个任务，3 个软件定时器及其回调函数，所以，整个代码有点多，我们创建的 7 个任务为：start_task、led_task、touch_task、qmsgshow_task 、flags_task 、main_task 和 key_task，优先级分别是 10 和 7~2，堆栈大小除了 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 52.2 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；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。

52.4 下载验证

在代码编译成功之后，我们通过下载代码到精英 STM32 开发板上，可以看到 LCD 显示界面如图 52.4.1 所示：

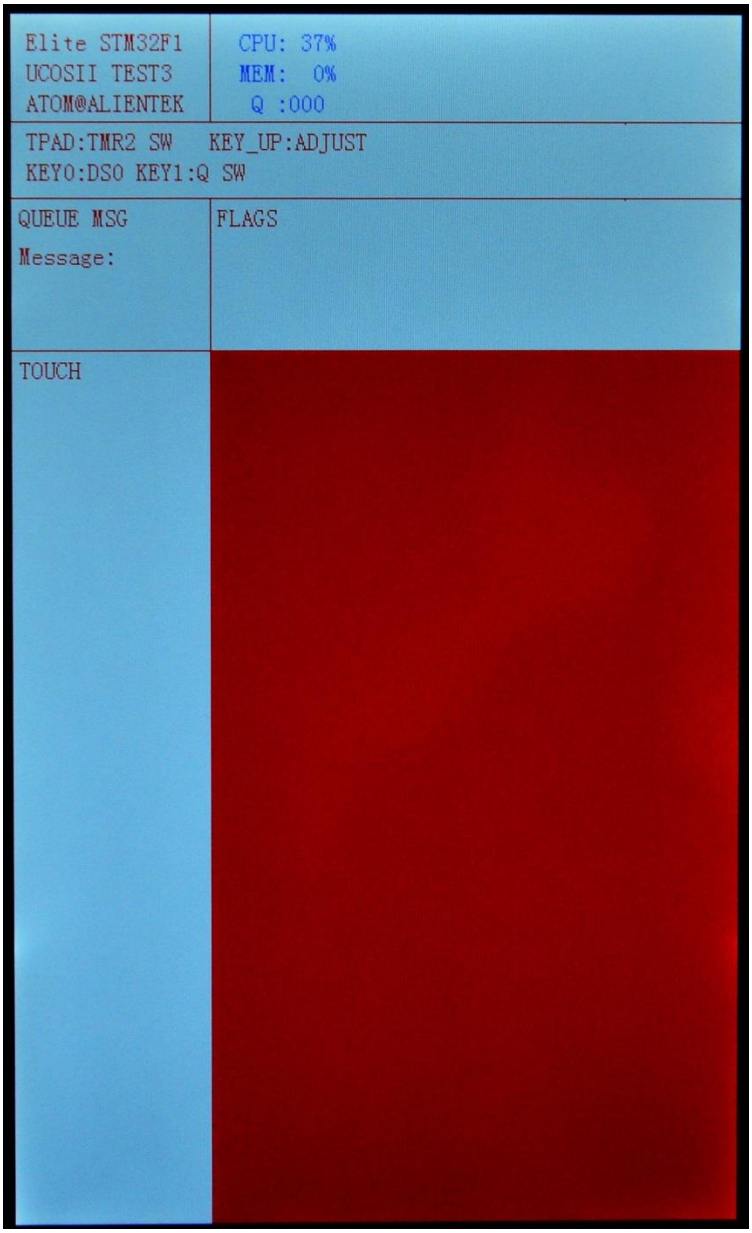


图 52.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率为 40%左右。比上一章多出不少，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 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。