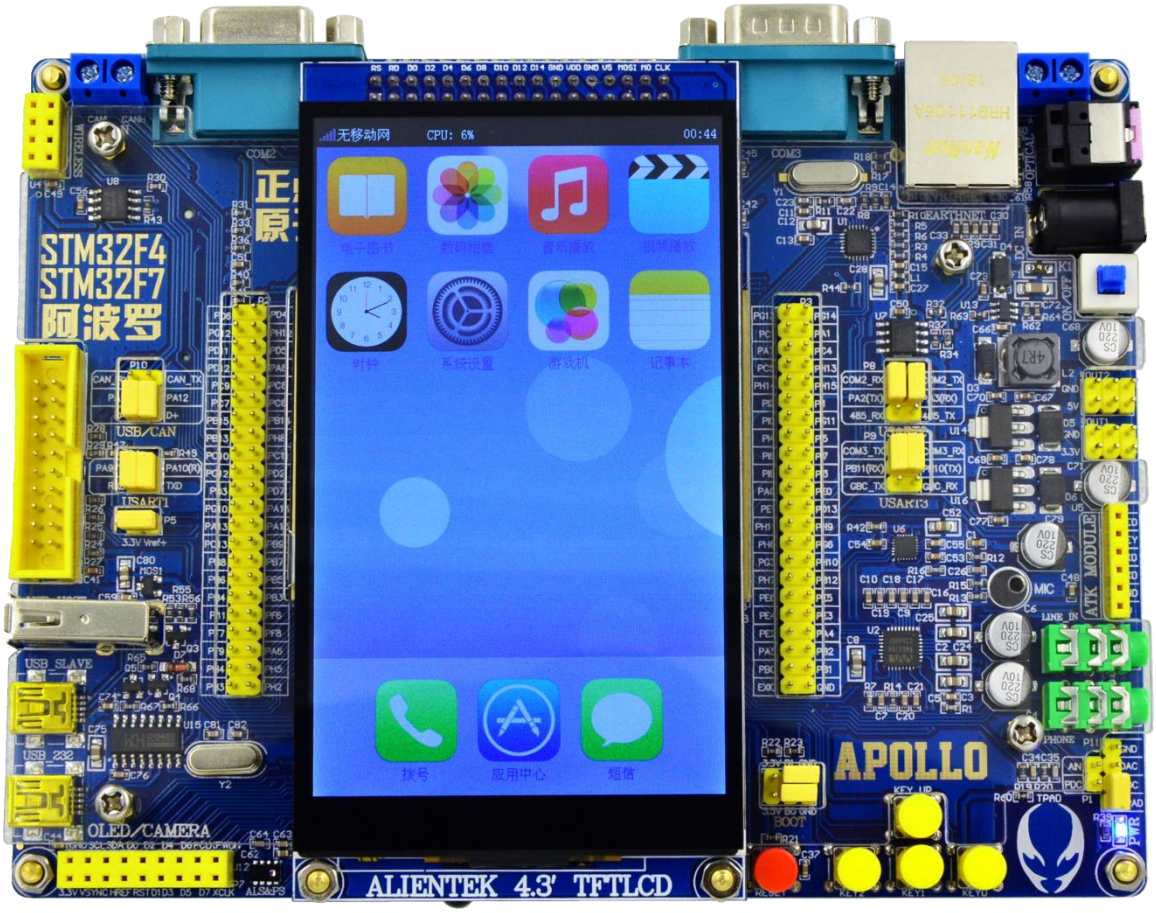
STM32F7 开发指南

V1.0 - HAL 库版本

-ALIENTEK 阿波罗 STM32F767 开发板教程



广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛（开源电子网）：www.openedv.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com　　**QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友 情 提 示

如果您想及时免费获取“正点原子”最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。

关注方法：

(1)微信“扫一扫”，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入“正点原子”→关注

(3)微信→添加朋友→输入“alientek_stm32”→关注

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 阿波罗 STM32F4/F7 开发板资源初探 5
1.1.1 阿波罗 STM32 开发板底板资源 5
1.1.2 STM32F767 核心板资源 7
1.2 ALIENTEK 阿波罗 STM32F767 开发板资源说明 8
1.2.1 硬件资源说明 8
1.2.2 软件资源说明 14
1.2.3 阿波罗 IO 引脚分配 15
第二章 实验平台硬件资源详解 21
2.1 开发板底板原理图详解 21
2.1.1 核心板接口 21
2.1.2 引出 IO 口 21
2.1.3 USB 串口/串口 1 选择接口 22
2.1.4 JTAG/SWD 22
2.1.5 参考电压选择端口 23
2.1.6 LCD 模块接口 23
2.1.7 复位电路 24
2.1.8 启动模式设置接口 24
2.1.9 VBAT 供电接口 25
2.1.10 RS232 串口 26
2.1.11 RS485 接口 26
2.1.12 CAN/USB 接口 27
2.1.13 光环境传感器 27
2.1.14 IIC IO 扩展 28
2.1.15 九轴传感器 28
2.1.16 温湿度传感器接口 29
2.1.17 红外接收头 29
2.1.18 WIRELESS 模块接口 30
2.1.19 LED 30
2.1.20 按键 31

2.1.21 TPAD 电容触摸按键 31
2.1.22 OLED/摄像头模块接口 31
2.1.23 有源蜂鸣器 32
2.1.24 SD 卡接口 33
2.1.25 ATK 模块接口 33
2.1.26 多功能端口 33
2.1.27 光纤输入接口 35
2.1.28 以太网接口（RJ45） 35
2.1.29 I2S 音频编解码器 36
2.1.30 电源 37
2.1.31 电源输入输出接口 37
2.1.32 USB 串口 38
2.2 STM32F767 核心板原理图详解 38
2.2.1 MCU 38
2.2.2 底板接口 40
2.2.3 SWD 调试接口 40
2.2.4 SDRAM 41
2.2.5 NAND FLASH 41
2.2.6 SPI FLASH 42
2.2.7 EEPROM 42
2.2.8 RGB LCD 接口 43
2.2.9 串口 43
2.2.10 Micro USB 接口 44
2.2.11 按键 44
2.2.12 LED 44
2.2.13 电源 45
2.3 开发板使用注意事项 45
2.3 STM32F767 学习方法 46
第二篇 软件篇 48
第三章 软件入门 49
3.1 MDK5 简介与安装 49
3.2 STM32CubeF7 简介 50
3.2.1 库开发与寄存器开发的关系 50

3. 2. 2 STM32CubeF7 固件包介绍 51
3.3 新建基于 HAL 库的工程模板和工程结构讲解 54
3.3.1 新建基于 HAL 库工程模板 55
3.3.2 工程模板解读 82
3.3.2.1 关键文件介绍 82
3.3.2.2 HAL 库中__weak 修饰符讲解 85
3.3.2.3 Msp 回调函数执行过程解读 86
3.3.2.4 程序执行流程图 88
3.4 程序下载与调试 90
3.4.1 STM32F7 程序下载 90
3.4.2 STM32F7 在线调试 94
3.5 MDK5 使用技巧 99
3.5.1 文本美化 100
3.5.2 语法检测&代码提示 102
3.5.3 代码编辑技巧 104
3.5.4 其他小技巧 108
第四章 STM32F7 基础知识入门 110
4.1 MDK 下 C 语言基础复习 110
4.1.1 位操作 110
4.1.2 define 宏定义 111
4.1.3# ifdef 和 #if defined 条件编译 111
4.1.4 extern 变量申明 112
4.1.5 typedef 类型别名 113
4.1.6 结构体 113
4.2 STM32F7 总线架构 115
4.3 STM32F7 时钟系统 118
4.3.1 STM32F7 时钟树概述 118
4.3.2 STM32F7 时钟系统配置 122
4.3.3 STM32F7 时钟使能和配置 127
4.4 IO 引脚复用器和映射 128
4.5 STM32 NVIC 中断优先级管理 132
4.6 HAL 库中寄存器地址名称映射分析 136
4.7 MDK 中使用 HAL 库快速组织代码技巧 138

4.8 手把手教你入门 STM32CubeMX 图形配置工具 143
4.8.1 STM32CubeMX 简介 143
4.8.2 STM32CubeMX 运行环境搭建 144
4.8.3 使用 STM32CubeMX 工具配置工程模板 148
4.8.3.1 工程初步建立和保存 148
4.8.3.2 RCC 设置 152
4.8.3.3 时钟系统（时钟树）配置 154
4.8.3.4 GPIO 功能引脚配置 156
4.8.3.5 Cortex-M7 内核基本配置 159
4.8.3.6 生成工程源码 161
4.8.3.7 编写用户程序 163
第五章 SYSTEM 文件夹介绍 167
5.1 delay 文件夹代码介绍 167
5.1.1 操作系统支持宏定义及相关函数 168
5.1.2 delay_init 函数 170
5.1.3 delay_us 函数 171
5.1.4 delay_ms 函数 172
5.1.5 HAL 库延时函数 HAL_Delay 解析 173
5.2 sys 文件夹代码介绍 175
5.2.1 Cache 使能函数 175
5.3 usart 文件夹介绍 176
5.3.1 printf 函数支持 176
第三篇 实战篇 177
第六章 跑马灯实验 178
6.1 STM32F7 IO 简介 178
6.2 硬件设计 185
6.3 软件设计 185
6.4 下载验证 194
6.5 STM32CubeMX 配置 IO 口输入 195
第七章 按键输入实验 199
7.1 STM32F7 IO 口简介 199
7.2 硬件设计 199
7.3 软件设计 199

7.4 下载验证 203
7.5 STM32CubeMX 配置 IO 口输出 203
第八章 串口通信实验 206
8.1 STM32F7 串口简介 206
8.2 硬件设计 211
8.3 软件设计 211
8.4 下载验证 218
8.5 STM32CubeMX 配置串口 219
第九章 外部中断实验 227
9.1 STM32F7 外部中断简介 227
9.2 硬件设计 230
9.3 软件设计 230
9.4 下载验证 234
9.5 STM32CubeMX 配置外部中断 234
第十章 独立看门狗（IWDG）实验 237
10.1 STM32F7 独立看门狗简介 237
10.2 硬件设计 240
10.3 软件设计 240
10.4 下载验证 242
10.5 STM32CubeMX 配置 IWDG 242
第十一章 窗口门狗（WWDG）实验 244
11.1 STM32F7 窗口看门狗简介 244
11.2 硬件设计 247
11.3 软件设计 247
11.4 下载验证 249
11.5 STM32CubeMX 配置 WWDG 249
第十二章 定时器中断实验 252
12.1 STM32F7 通用定时器简介 252
12.2 硬件设计 257
12.3 软件设计 257
12.4 下载验证 259
12.5 STM32CubeMX 配置定时器更新中断功能 259
第十三章 PWM 输出实验 261

13.1 PWM 简介 261
13.2 硬件设计 266
13.3 软件设计 266
13.4 下载验证 268
13.5 STM32CubeMX 配置定时器 PWM 输出功能 268
第十四章 输入捕获实验 271
14.1 输入捕获简介 271
14.2 硬件设计 277
14.3 软件设计 277
14.4 下载验证 282
14.5 STM32CubeMX 配置定时器输入捕获功能 282
第十五章 电容触摸按键实验 284
15.1 电容触摸按键简介 284
15.2 硬件设计 285
15.3 软件设计 285
15.4 下载验证 290
第十六章 OLED 显示实验 292
16.1 OLED 简介 292
16.2 硬件设计 298
16.3 软件设计 298
16.4 下载验证 306
第十七章 内存保护（MPU）实验 307
17.1 MPU 简介 307
17.2 硬件设计 312
17.3 软件设计 312
17.4 下载验证 315
第十八章 TFTLCD（MCU 屏）实验 317
18.1 TFTLCD&FMC 简介 317
18.1.1 TFTLCD 简介 317
18.2 硬件设计 332
18.3 软件设计 333
18.4 下载验证 343
18.5 STM32CubeMX 配置 FMC（SRAM） 344

第十九章 SDRAM 实验.....346
19.1 SDRAM 简介.....346
19.1.1 SDRAM 简介.....346
19.1.2 FMC SDRAM 接口简介.....355
19.2 硬件设计.....363
19.3 软件设计.....363
19.4 下载验证.....368
19.5 STM32CubeMX 配置 FMC（SDRAM）.....370
第二十章 LTDC LCD（RGB 屏）实验.....372
20.1 RGBLCD&LTDC 简介.....372
20.1.1 RGBLCD 简介.....372
20.1.2 LTDC 简介.....375
20.1.3 DMA2D 简介.....387
20.2 硬件设计.....392
20.3 软件设计.....393
20.4 下载验证.....405
第二十一章 USMART 调试组件实验.....406
21.1 USMART 调试组件简介.....406
21.2 硬件设计.....410
21.3 软件设计.....410
21.4 下载验证.....414
第二十二章 RTC 实时时钟实验.....419
22.1 STM32F767 RTC 时钟简介.....419
22.2 硬件设计.....430
22.3 软件设计.....430
22.4 下载验证.....435
第二十三章 硬件随机数实验.....437
23.1 STM32F767 随机数发生器简介.....437
23.2 硬件设计.....439
23.3 软件设计.....439
23.4 下载验证.....442
第二十四章 待机唤醒实验.....443
24.1 STM32F767 待机模式简介.....443

24.2 硬件设计 446
24.3 软件设计 446
24.4 下载与测试 450
第二十五章 ADC 实验 451
25.1 STM32F767 ADC 简介 451
25.2 硬件设计 458
25.3 软件设计 459
25.4 下载验证 462
第二十六章 内部温度传感器实验 463
26.1 STM32F767 内部温度传感器简介 463
26.2 硬件设计 463
26.3 软件设计 464
26.4 下载验证 465
第二十七章 DAC 实验 466
27.1 STM32F767 DAC 简介 466
27.2 硬件设计 472
27.3 软件设计 473
27.4 下载验证 475
第二十八章 PWM DAC 实验 477
28.1 PWM DAC 简介 477
28.2 硬件设计 479
28.3 软件设计 480
28.4 下载验证 483
第二十九章 DMA 实验 485
29.1 STM32F767 DMA 简介 485
29.2 硬件设计 492
29.3 软件设计 493
29.4 下载验证 496
第三十章 IIC 实验 498
30.1 IIC 简介 498
30.2 硬件设计 499
30.3 软件设计 499
30.4 下载验证 507

第三十一章 IO 扩展实验 508
31.1 PCF8574/AT8574 简介 508
31.2 硬件设计 510
31.3 软件设计 511
31.4 下载验证 514
第三十二章 光环境传感器实验 516
32.1 AP3216C 简介 516
32.2 硬件设计 519
32.3 软件设计 520
32.4 下载验证 523
第三十三章 QSPI 实验 524
33.1 QSPI 简介 524
33.1.1 QSPI 接口简介 524
33.1.2 W25Q256 简介 532
33.2 硬件设计 534
33.3 软件设计 535
33.4 下载验证 542
第三十四章 485 实验 544
34.1 485 简介 544
34.2 硬件设计 545
34.3 软件设计 546
34.4 下载验证 550
第三十五章 CAN 通讯实验 553
35.1 CAN 简介 553
35.2 硬件设计 573
35.3 软件设计 574
35.4 下载验证 580
第三十六章 触摸屏实验 582
36.1 触摸屏简介 582
36.1.1 电阻式触摸屏 582
36.1.2 电容式触摸屏 583
36.2 硬件设计 586
36.3 软件设计 587

36.4 下载验证 602
第三十七章 红外遥控实验 605
37.1 红外遥控简介 605
37.2 硬件设计 606
37.3 软件设计 607
37.4 下载验证 612
第三十八章 DS18B20 数字温度传感器实验 614
38.1 DS18B20 简介 614
38.2 硬件设计 615
38.3 软件设计 616
38.4 下载验证 621
第三十九章 DHT11 数字温湿度传感器实验 622
39.1 DHT11 简介 622
39.2 硬件设计 624
39.3 软件设计 625
39.4 下载验证 629
第四十章 MPU9250 九轴传感器实验 630
40.1 MPU9250 简介 630
40.1.1 MPU9250 基础介绍 630
40.1.2 DMP 使用简介 636
40.2 硬件设计 639
40.3 软件设计 640
40.4 下载验证 649
第四十一章 无线通信实验 652
41.1 SPI&NRF24L01 无线模块简介 652
41.1.1 SPI 接口简介 652
41.1.2 NRF24L01 无线模块简介 656
41.2 硬件设计 657
41.3 软件设计 658
41.4 下载验证 667
第四十二章 FLASH 模拟 EEPROM 实验 669
42.1 STM32F767 FLASH 简介 669
42.2 硬件设计 675

42.3 软件设计 675
42.4 下载验证 679
第四十三章 摄像头实验 681
43.1 OV5640&DCMI 简介 681
43.1.1 OV5640 简介 681
43.1.2 STM32F767 DCMI 接口简介 686
43.2 硬件设计 693
43.3 软件设计 695
43.4 下载验证 709
第四十四章 内存管理实验 712
44.1 内存管理简介 712
44.2 硬件设计 713
44.3 软件设计 713
44.4 下载验证 721
第四十五章 SD 卡实验 723
45.1.2 SDMMC 的时钟 723
45.1.3 SDMMC 的命令与响应 723
45.1.4 SDMMC 相关寄存器介绍 726
45.1.5 SD 卡初始化流程 731
45.2 硬件设计 733
45.3 软件设计 734
45.4 下载验证 741
第四十六章 NAND FLASH 实验 743
46.1.1 NAND FLASH 简介 743
46.1.2 FTL 简介 750
46.1.3 FMC NAND FLASH 接口简介 753
46.2 硬件设计 758
46.3 软件设计 758
46.4 下载验证 773
第四十七章 FATFS 实验 775
47.1 FATFS 简介 775
47.2 硬件设计 779
47.3 软件设计 780

47.4 下载验证 789
第四十八章 汉字显示实验 791
48.1 汉字显示原理简介 791
48.2 硬件设计 796
48.3 软件设计 796
48.4 下载验证 807
第四十九章 图片显示实验 809
49.1 图片格式简介 809
49.2 硬件设计 810
49.3 软件设计 811
49.4 下载验证 821
第五十章 硬件 JPEG 解码实验 822
50.1 硬件 JPEG 编解码器简介 822
50.2 硬件设计 827
50.3 软件设计 828
50.4 下载验证 845
第五十一章 照相机实验 847
51.1 BMP&JPEG 编码简介 847
51.1.1 BMP 编码简介 847
51.1.2 JPEG 编码简介 850
51.2 硬件设计 851
51.3 软件设计 851
51.4 下载验证 863
第五十二章 音乐播放器实验 866
52.1 WAV&WM8978&SAI 简介 866
52.1.1 WAV 简介 866
52.1.2 WM8978 简介 868
52.1.3 SAI 简介 870
52.2 硬件设计 880
52.3 软件设计 881
52.4 下载验证 896
第五十三章 录音机实验 898
53.1 SAI 录音简介 898

53.2 硬件设计 900
53.3 软件设计 901
53.4 下载验证 910
第五十四章 SPDIF(光纤音频)实验 913
54.1 SPDIF 简介 913
54.2 硬件设计 922
54.3 软件设计 923
54.4 下载验证 932
第五十五章 视频播放器实验 933
555.1 AVI 简介 933
55.2 硬件设计 938
55.3 软件设计 939
55.4 下载验证 951
第五十六章 FPU 测试(Julia 分形)实验 956
56.1 FPU&Julia 分形简介 956
56.1.1 FPU 简介 956
56.1.2 Julia 分形简介 957
56.2 硬件设计 959
56.3 软件设计 959
56.4 下载验证 962
第五十七章 DSP 测试实验 964
57.1 DSP 简介与环境搭建 964
57.1.1 STM32F7 DSP 简介 964
57.1.2 DSP 库运行环境搭建 967
57.2 硬件设计 969
57.3 软件设计 969
57.3.1 DSP BasicMath 测试 969
57.3.1 DSP FFT 测试 972
57.4 下载验证 975
第五十八章 手写识别实验 977
58.1 手写识别简介 977
58.2 硬件设计 981
58.3 软件设计 981

58.4 下载验证 985
第五十九章 T9 拼音输入法实验 987
59.1 拼音输入法简介 987
59.2 硬件设计 989
59.3 软件设计 989
59.4 下载验证 996
第六十章 串口 IAP 实验 999
60.1 IAP 简介 999
60.2 硬件设计 1005
60.3 软件设计 1005
60.4 下载验证 1011
第六十一章 USB 读卡器(Slave)实验 1013
61.1 USB 简介 1013
61.2 硬件设计 1016
61.3 软件设计 1017
61.4 下载验证 1026
第六十二章 USB 声卡(Slave)实验 1028
62.1 USB 声卡简介 1028
62.2 硬件设计 1028
62.3 软件设计 1028
62.4 下载验证 1036
第六十三章 USB 虚拟串口(Slave)实验 1038
63.1 USB 虚拟串口简介 1038
63.2 硬件设计 1038
63.3 软件设计 1038
63.4 下载验证 1045
第六十四章 USB U 盘(Host)实验 1048
64.1 U 盘简介 1048
64.2 硬件设计 1048
64.3 软件设计 1049
64.4 下载验证 1056
第六十五章 USB 鼠标键盘(Host)实验 1058

65.1 USB 鼠标键盘简介 1058
65.2 硬件设计 1058
65.3 软件设计 1058
65.4 下载验证 1066
第六十六章 网络通信实验 1068
66.1 STM32F767 以太网以及 TCP/IP LWIP 简介 1068
66.1.1 STM32F767 以太网简介 1068
66.1.2 TCP/IP LWIP 简介 1073
66.2 硬件设计 1075
66.3 软件设计 1076
66.4 下载验证 1080
66.4.1 Web Server 测试 1082
66.4.2 TCP Server 测试 1084
66.4.3 TCP Client 测试 1085
66.4.4 UDP 测试 1087
第六十七章 UCOSII 实验 1-任务调度 1089
67.1 UCOSII 简介 1089
67.2 硬件设计 1095
67.3 软件设计 1095
67.4 下载验证 1099
67.5 任务删除，挂起和恢复测试 1099
第六十八章 UCOSII 实验 2-信号量和邮箱 1104
68.1 UCOSII 信号量和邮箱简介 1104
68.2 硬件设计 1106
68.3 软件设计 1107
68.4 下载验证 1113
第六十九章 UCOSII 实验 3-消息队列、信号量集和软件定时器 1114
69.1 UCOSII 消息队列、信号量集和软件定时器简介 1114
69.2 硬件设计 1122
69.3 软件设计 1122
69.4 下载验证 1131

内容简介

本手册将由浅入深，带领大家学习 STM32F7 的各个功能，为您开启 STM32F7 的学习之旅。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 STM32F7 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 64 个实例（都是通过操作 HAL 库完成的）带领大家一步步深入了解 STM32F7。

本手册为 ALIENTEK 阿波罗 STM32F7 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。

本手册不仅非常适合广大学生和电子爱好者学习 STM32F7，其大量的实验以及详细的解说，也是公司产品开发的不二参考。

前言

作为 Cortex M 系列通用处理器市场的最大占有者，STM32 以其优异的性能、超高的性价比、丰富的本地化教程，迅速占领了市场。ST 公司自 2007 年推出第一款 STM32 以来，先后推出了 STM32F0/F1/F2/F3/F4/F7 等系列产品，涵盖了 Cortex M0/M3/M4/M7 等内核，总出货量超过 18 亿颗，是 ARM 公司 Cortex M 系列内核的霸主。

STM32F7 系列，是 ST 推出的基于 ARM Cortex M7 内核的处理器，采用 6 级流水线，性能高达 5 CoreMark/MHz，在 200MHz 工作频率下测试数据高达 1000 CoreMarks，远超此前性能最高的 STM32F4（Cortex M4 内核）系列，DSP 性能超过 STM32F4 的两倍。

STM32F76x 系列（包括：STM32F765/767/768/769 等），主要有如下优势：

1， 更先进的内核，采用 Cortex M7 内核，具有 16KB 指令/数据 Cache，采用 ST 独有的自适应实时加速技术（ART Accelerator），性能高达 5 CoreMark/MHz。

2， 更丰富的外设，拥有高达 512KB 的片内 SRAM，并且支持 SDRAM、带 TFTLCD 控制器、带图形加速器（Chorme ART）、带摄像头接口（DCMI）、带硬件 JPEG 编解码器、带 QSPI 接口、带 SAI&I2S 音频接口、带 SPDIF RX 接口、USB 高速 OTG、真随机数发生器、OTP 存储器等。

3， 更高的性能，STM32F767 最高运行频率可达 216Mhz，具有 6 级流水线，带有指令和数据 Cache，大大提高了性能，性能大概是 STM32F4 的两倍。而且 STM32F76x 自带了双精度硬件浮点单元（DFFPU），在做 DSP 处理的时候，具有更好的性能。

STM32F76x 系列，自带了 LCD 控制器和 SDRAM 接口，对于想要驱动大屏或需要大内存的朋友来说，是个非常不错的选择，更重要的是集成了硬件 JPEG 编解码器，可以秒解 JPEG 图片，做界面的时候，可以大大提高加载速度，并且，可以实现视频播放。本手册，我们将以 STM32F767 为例，向大家讲解 STM32F767 的学习。

学习 STM32F767 有几份资料经常用到：

《STM32F7 中文参考手册》

《STM32F7xx 参考手册》英文版

《STM32F7 编程手册》

其中，最常用的是《STM32F7 中文参考手册》，该文档是 ST 官方针对 STM32F74x/75x 的一份中文参考资料，里面有绝大部分寄存器的详细描述，内容详实，方便大家编写代码，不过没有实例，也没有对 Cortex-M7 构架进行多少介绍，读者只能根据自己对书本的理解来编写相关代码。另外，对 STM32F767 特有的部分外设（比如硬件 JPEG 编解码器、DFSDM 等），我们则必须参考《STM32F7xx 参考手册》英文版来学习。

而《STM32F7 编程手册》这个文档，则重点介绍了 Cortex M7 内核的汇编指令及其使用，以及内核相关寄存器（比如：SCB，NVIC，SYSTICK 等寄存器），是《STM32F7 中文参考手册》的重要补充，很多在《STM32F7 中文参考手册》无法找到的内容，都可以在这里找到答案，不过目前该文档没有中文版本，只有英文版。

本手册将结合以上三份资料的优点，从库函数级别出发，深入浅出，向读者展示 STM32F7 的各种功能。总共配有 64 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 STM32F7 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、NAND FLASH FTL、拼音输入法、手写识别、图片解码、IAP 等。所有实例在 MDK5.21A 编译器下编译通过，大家只需下载程序到 ALIENTEK 阿波罗 STM32 开发板，即可验证实验。

不管你是一个 STM32 初学者，还是一个老手，本手册都非常适合。尤其对于初学者，本手册将手把手的教你如何使用 MDK，包括新建工程、编译、仿真、下载调试等一系列步骤，让你轻松上手。

本手册的实验平台是 ALIENTEK 阿波罗 STM32F7 开发板，有这款开发板的朋友则直接可以拿本手册配套的光盘上的例程在开发板上运行、验证。而没有这款开发板而又想要的朋友，可以上淘宝购买。当然你如果有了一款自己的开发板，而又不想再买，也是可以的，只要你的板子上有 ALIENTEK 阿波罗 STM32 开发板上的相同资源（需要实验用到的），代码一般都是可以通用的，你需要做的就只是把底层的驱动函数（比如 IO 口修改）稍做修改，使之适合你的开发板即可。

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

第一篇 硬件篇

实践出真知，要想学好 STM32F7，实验平台必不可少！本篇将详细介绍我们用来学习 STM32F7 的硬件平台：ALIENTEK 阿波罗 STM32F7 开发板，通过该篇的介绍，你将了解到我们的学习平台 ALIENTEK 阿波罗 STM32F7 开发板的功能及特点。

为了让读者更好的使用 ALIENTEK 阿波罗 STM32F7 开发板，本篇还介绍了开发板的一些使用注意事项，请读者在使用开发板的时候一定要注意。

本篇将分为如下两章：

1，实验平台简介；

2，实验平台硬件资源详解；

第一章 实验平台简介

本章，主要向大家简要介绍我们的实验平台：ALIENTEK 阿波罗 STM32F4/F7 开发板。通过本章的学习，你将对我们后面使用的实验平台有个大概了解，为后面的学习做铺垫。

本章将分为如下两节：

1.1，ALIENTEK 阿波罗 STM32F4/F7 开发板资源初探；

1.2，ALIENTEK 阿波罗 STM32F4/F7 开发板资源说明；

1.1 ALIENTEK 阿波罗 STM32F4/F7 开发板资源初探

ALIENTEK 之前总共推出过四款开发板：mini 板、精英板、战舰板和探索板，前三款均为 STM32F1 系列开发板，探索板为 STM32F407 开发板，这几款开发板常年稳居淘宝销量冠军，累计出货超过 8 万套。而这款阿波罗开发板，则是 ALIENTEK 推出的第二款 Cortex M4（F429）开发板和第一款 Cortex M7（F767）开发板，阿波罗开发板采用核心板+底板的形式，当使用 STM32F767 的核心板时，它就是一款 STM32F767 开发板，当使用 STM32F767 核心板时，它就是一款 STM32F767 开发板。接下来我们分别介绍阿波罗 STM32 开发板的底板和核心板。

1.1.1 阿波罗 STM32 开发板底板资源

首先，我们来看阿波罗 STM32 开发板的底板资源图，如图 1.1.1.1 所示：

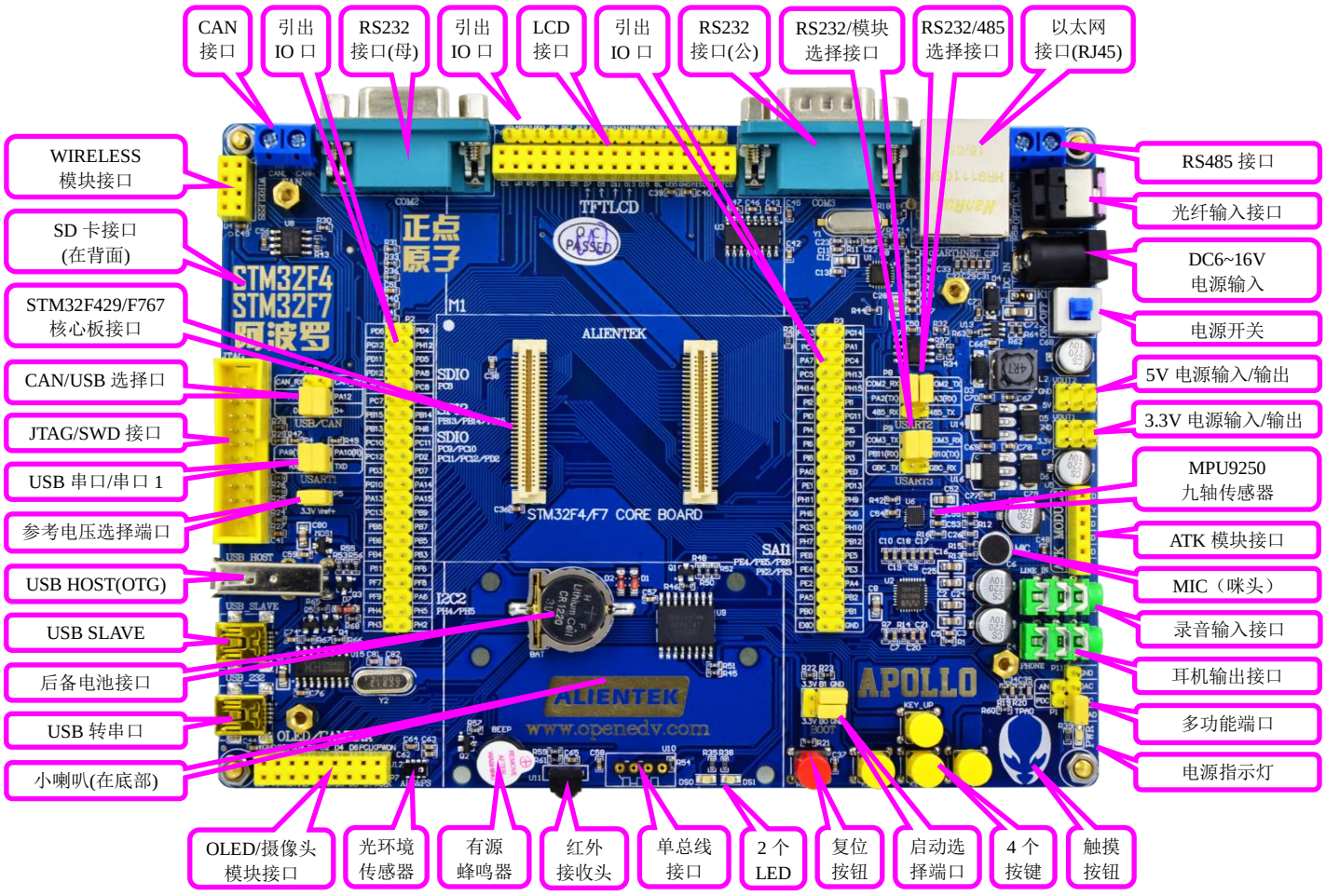


图 1.1.1.1 阿波罗 STM32 开发板底板资源图

从图 1.1.1.1 可以看出，阿波罗 STM32 开发板底板，资源十分丰富，把 STM32F429/F767 的内部资源发挥到了极致，基本所有 STM32F429/F767 的内部资源，都可以在此开发板上验证，同时扩充丰富的接口和功能模块，整个开发板显得十分大气。

开发板的外形尺寸为 121mm*160mm 大小，板子的设计充分考虑了人性化设计，并结合 ALIENTEK 多年的 STM32 开发板设计经验，经过多次改进，最终确定了这样的设计。

ALIENTEK 阿波罗 STM32 开发板底板板载资源如下：

- 1 个核心板接口，支持 STM32F429/F767 等核心板
- 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 个九轴（陀螺仪+加速度+磁力计）传感器芯片，MPU9250
- 1 个高性能音频编解码芯片，WM8978
- 1 个无线模块接口，支持 NRF24L01 无线模块
- 1 路光纤输入接口（音频，仅 F7 支持）
- 1 路 CAN 接口，采用 TJA1050 芯片
- 1 路 485 接口，采用 SP3485 芯片
- 2 路 RS232 串口（一公一母）接口，采用 SP3232 芯片
- 1 路单总线接口，支持 DS18B20/DHT11 等单总线传感器
- 1 个 ATK 模块接口，支持 ALIENTEK 蓝牙/GPS/MPU6050/RGB 灯模块
- 1 个光环境传感器（光照、距离、红外三合一）
- 1 个标准的 2.4/2.8/3.5/4.3/7 寸 LCD 接口，支持电阻/电容触摸屏
- 1 个摄像头模块接口
- 1 个 OLED 模块接口
- 1 个 USB 串口，可用于程序下载和代码调试（USMART 调试）
- 1 个 USB SLAVE 接口，用于 USB 从机通信
- 1 个 USB HOST(OTG)接口，用于 USB 主机通信
- 1 个有源蜂鸣器
- 1 个 RS232/RS485 选择接口
- 1 个 RS232/模块选择接口
- 1 个 CAN/USB 选择接口
- 1 个串口选择接口
- 1 个 SD 卡接口（在板子背面）
- 1 个百兆以太网接口（RJ45）
- 1 个标准的 JTAG/SWD 调试下载口
- 1 个录音头（MIC/咪头）
- 1 路立体声音频输出接口
- 1 路立体声录音输入接口
- 1 个小扬声器（在板子背面）
- 1 组多功能端口（DAC/ADC/PWM DAC/AUDIO IN/TPAD）
- 1 组 5V 电源供应/接入口
- 1 组 3.3V 电源供应/接入口
- 1 个参考电压设置接口
- 1 个直流电源输入接口（输入电压范围：DC6~24V）

- 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
- 4 个功能按钮，其中 KEY_UP(即 WK_UP)兼具唤醒功能
- 1 个电容触摸按键
- 1 个电源开关，控制整个板的电源
- 独创的一键下载功能
- 引出 110 个 IO 口

ALIENTEK 阿波罗 STM32 开发板底板的特点包括：

1） 接口丰富。板子提供十来种标准接口，可以方便的进行各种外设的实验和开发。

2） 设计灵活。我们采用核心板+底板形式，一款底板可以学习多款 MCU，减少重复投资；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 110 个 IO 口，极大的方便大家扩展及使用。板载一键下载功能，可避免频繁设置 B0、B1 的麻烦，仅通过 1 根 USB 线即可实现 STM32 的开发。

3） 资源丰富。板载高性能音频编解码芯片、九轴传感器、百兆网卡、光环境传感器以及各种接口芯片，满足各种应用需求。

4） 人性化设计。各个接口都有丝印标注，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。

1.1.2 STM32F767 核心板资源

接下来，我们来看 STM32F767 核心板资源图，如图 1.1.2.1 所示：

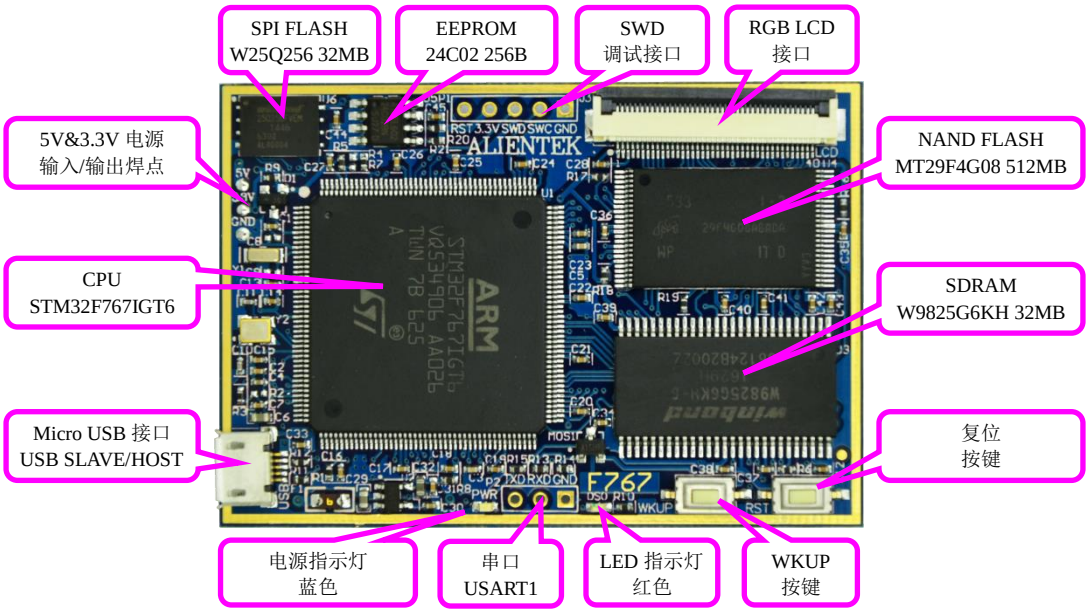


图 1.1.2.1 STM32F767 核心板资源图

从图 1.1.2.1 可以看出，STM32F767 核心板的板载资源十分丰富，可以满足各种应用的需求，完全可以独立使用。整个核心板的外形尺寸为 65mm*45mm 大小，非常小巧，并且，采用了贴片板对板连接器，使得其可以很方便的应用在各种项目上。

ALIENTEK STM32F767 核心板板载资源如下：

- CPU：STM32F767IGT6，LQFP176，FLASH：1024KB，SRAM：512KB

- 外扩 SDRAM：W9825G6KH，32M 字节
- 外扩 NAND FLASH：MT29F4G08，512M 字节
- 外扩 SPI FLASH：W25Q256，32M 字节
- 外扩 EEPROM：24C02，256 字节
- 2 个板对板接口（在底部），引出 110 个 IO，方便接入各种底板
- 1 个 5V&3.3V 焊点，支持外接电源或输出电源给外部
- 1 个 Micro USB 接口，可作 USB SLAVE/HOST(OTG)使用
- 1 个电源指示灯（蓝色）
- 1 个状态指示灯（红色）
- 1 个 TTL 串口（USART1）
- 1 个复位按钮，可用于复位 MCU 和 LCD
- 1 个功能按钮，WKUP，可以用作 MCU 唤醒
- 1 个 RGB LCD 接口，支持 RGB 接口的 LCD 屏（RGB565 格式）
- 1 个 SWD 调试接口

ALIENTEK STM32F767 核心板的特点包括：

1） 体积小巧。核心板仅 65mm*45mm 大小，方便使用到各种项目里面。

2） 接口丰富。核心板自带了串口、SWD 调试接口、RGB LCD 屏接口、USB 接口和 3.3V&5V 电源接口等，并通过板对板接口，引出了 110 个 IO 口，满足各种应用需求。

3） 资源丰富。核心板板载：32MB SDRAM、32MB SPI FLASH、512MB NAND FLASH 和 EEPROM 等存储器，可以满足各种应用需求。

4） 性能稳定。核心板采用 4 层板设计，单独地层、电源层，且关键信号采用等长线走线，保证运行稳定、可靠。

5） 人性化设计。各个接口都有丝印标注，使用起来一目了然；接口位置设计合理，方便顺手。

1.2 ALIENTEK 阿波罗 STM32F767 开发板资源说明

资源说明部分，我们将分为两个部分说明：硬件资源说明和软件资源说明。

1.2.1 硬件资源说明

这里我们首先详细介绍阿波罗 STM32F767 开发板的各个部分，包括底板和核心板两部分（图 1.1.1.1 和图 1.1.2.1 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。 首先，我们来看底板的资源说明：

1. WIRELESS 模块接口

这是开发板板载的无线模块接口（U4），可以插入 NRF24L01 模块/WIFI 模块等无线模块，从而实现无线通信功能。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. SD 卡接口

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

3. STM32F429/F767 核心板接口

这是开发板底板上面的核心板接口，由 2 个 2*30 的贴片板对板接线端子（3710F 母座）组成，可以用来插 ALIENTEK 的 STM32F429 核心板/STM32F767 核心板等，从而学习

STM32F429/STM32F767 等芯片，达到一个开发板，学习多款 MCU 的目的，减少重复投资。

4. CAN/USB 选择口

这是一个 CAN/USB 的选择接口（P10），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. JTAG/SWD 接口

这是开发板板载的 20 针标准 JTAG 调试口（JTAG），该 JTAG 口直接可以和 ULINK、JLINK（**V9 或者以上版本**）或者 STLINK 等调试器（仿真器）连接，同时由于 STM32 支持 SWD 调试，这个 JTAG 口也可以用 SWD 模式来连接。

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

6. USB 串口/串口 1

这是 USB 串口同 STM32 的串口 1 进行连接的接口（P4），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，你可以把阿波罗 STM32 开发板当成一个 USB 转 TTL 串口，来和其他板子通信，而其他板子的串口，也可以方便地接到开发板上。

7. 参考电压选择端口（核心板指示灯控制口）

这是 STM32 的参考电压选择端口（P5），我们默认是接开发板的 3.3V（VDDA）。如果大家想设置其他参考电压，只需要把你的参考电压源接到 Vref+和 GND 即可 。**特别注意：P5 还有控制核心板指示灯亮灭的功能，当 P5 的 Vref+接 3.3V 的时候（默认），核心板的所有指示灯，都停止工作。当 Vref+悬空的时候，核心板的指示灯才正常工作。**

8. USB HOST(OTG)

这是开发板板载的一个侧插式的 USB-A 座（USB_HOST），由于 STM32F4/F7 的 USB 是支持 HOST 的，所以我们可以通过这个 USB-A 座，连接 U 盘/USB 鼠标/USB 键盘等其他 USB 从设备，从而实现 USB 主机功能。不过特别注意，由于 USB HOST 和 USB SLAVE 是共用 PA11 和 PA12，所以两者不可以同时使用。

9. USB SLAVE

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

开发板总共板载了两个 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。

10. 后备电池接口

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

11. USB 转串口

这是开发板板载的另外一个 MiniUSB 头（USB_232），用于 USB 连接 CH340G 芯片，从而

实现 USB 转串口。同时，此 MiniUSB 接头也是开发板的电源提供口。

12. 小喇叭

这是开发板自带的一个 8Ω 2W 的小喇叭，安装在开发板的背面，并带了一个小音腔，可以用来播放音频。该喇叭由 WM8978 直接驱动，最大输出功率可达 0.9W。

13. OLED/摄像头模块接口

这是开发板板载的一个 OLED/摄像头模块接口（P7），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，可以分别连接多个外部模块，从而实现相关实验。

14. 光环境传感器

这是开发板板载的一个光环境三合一传感器（U12），它可以作为：环境光传感器、近距离（接近）传感器和红外传感器。通过该传感器，开发板可以感知周围环境光线的变化，接近距离等，从而可以实现类似手机的自动背光控制。

15. 有源蜂鸣器

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

16. 红外接收头

这是开发板的红外接收头（U11），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。

阿波罗 STM32 开发板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

17. 单总线接口

这是开发板的一个单总线接口（U10），该接口由 4 个镀金排孔组成，可以用来接 DS18B20/DS1820 等单总线数字温度传感器。也可以用来接 DHT11 这样的单总线数字温湿度传感器。实现一个接口，多个功能。不用的时候，大家可以拆下上面的传感器，放到其他地方去用，使用上是十分方便灵活的。

18. 2 个 LED

这是开发板板载的两个 LED 灯（DS0 和 DS1），DS0 是红色的，DS1 是绿色的，主要是方便大家识别。两个 LED，一般的应用足够了，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。阿波罗 STM32 开发板几乎每个实例都使用了 LED 来指示程序的运行状态。

19. 复位按钮

这是开发板板载的复位按键（RESET），用于复位 STM32，还具有复位液晶的功能，因为

液晶模块的复位引脚和 STM32 的复位引脚是连接在一起的，当按下该键的时候，STM32 和液晶一并被复位。

20. 启动选择端口

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

21. 4 个按键

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

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

22. 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。

23. 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。

24. 多功能端口

这是 1 个由 6 个排针组成的一个接口（P1&P11）。不过大家可别小看这 6 个排针，这可是本开发板设计的很巧妙的一个端口（由 P1 和 P11 组成），这组端口通过组合可以实现的功能有：ADC 采集、DAC 输出、PWM DAC 输出、外部音频输入、电容触摸按键、DAC 音频、PWM DAC 音频、DAC ADC 自测等，所有这些，你只需要 1 个跳线帽的设置，就可以逐一实现。

25. 耳机输出接口

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

26. 录音输入接口

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

27. MIC（咪头）

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

28. ATK 模块接口

这是开发板板载的一个 ALIENTEK 通用模块接口（U5），目前可以支持 ALIENTEK 开发的 GPS 模块、蓝牙模块、MPU6050 模块和全彩 RGB 灯模块等，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。

29. MPU9250 九轴传感器

这是开发板板载的一个九轴传感器（U6），MPU9250 是一个高性能的九轴传感器，内部集成 1 个三轴加速度传感器、1 个三轴陀螺仪和 1 个三轴磁力传感器，并且带 MPL 功能，该传感器在四轴飞控方面应用非常广泛。所以喜欢玩四轴的朋友，也可通过本开发板进行学习。

30. 3.3V 电源输入/输出

这是开发板板载的一组 3.3V 电源输入输出排针（2*3）（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，有了阿波罗 STM32 开发板，你就可以很方便的拥有一个简单的 3.3V 电源（最大电流不能超过 500mA）。

31. 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）。

32. 电源开关

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

33. DC6~16V 电源输入

这是开发板板载的一个外部电源输入口（DC_IN），采用标准的直流电源插座。开发板板载了 DC-DC 芯片（MP2359），用于给开发板提供高效、稳定的 5V 电源。由于采用了 DC-DC 芯片，所以开发板的供电范围十分宽，大家可以很方便的找到合适的电源（只要输出范围在 DC6~16V 的基本都可以）来给开发板供电。在耗电比较大的情况下，比如用到 4.3 屏/7 寸屏/网口的时候，建议使用外部电源供电，可以提供足够的电流给开发板使用。

34. 光纤输入接口

这是开发板板载的音频光纤输入接口（OPTICAL），可以接收光纤传递过来的数字音频信号。**注意：此接口仅在使用 STM32F7 核心板的时候才有用，STM32F429 核心板无法使用。**

35. 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！

36. 以太网接口（RJ45）

这是开发板板载的网口（EARTHNET），可以用来连接网线，实现网络通信功能。该接口使用 STM32 内部的 MAC 控制器外加 PHY 芯片，实现 10/100M 网络的支持。

37. RS232/485 选择接口

这是开发板板载的 RS232（COM2）/485 选择接口（P8），因为 RS485 基本上就是一个半双工的串口，为了节约 IO，我们把 RS232（COM2）和 RS485 共用一个串口，通过 P9 来设置当前是使用 RS232（COM2）还是 RS485。这样的设计还有一个好处。就是我们的开发板既可以充当 RS232 到 TTL 串口的转换，又可以充当 RS485 到 TTL485 的转换。（注意，这里的 TTL 高电平是 3.3V）

38. RS232/模块选择接口

这是开发板板载的一个 RS232（COM3）/ATK 模块接口（U5）选择接口（P9），通过该选择接口，我们可以选择 STM32 的串口 3 连接在 COM3 还是连接在 ATK 模块接口上面，以实现不同的应用需求。该接口，同样也可以充当 RS232 到 TTL 串口的转换。

39. RS232 接口（公）

这是开发板板载的一个 RS232 接口（COM3），通过一个标准的 DB9 公头和外部的串口连接。通过这个接口，我们可以连接带有串口的电脑或者其他设备，实现串口通信。

40. 引出 IO 口（总共有三处）

这是开发板 IO 引出端口，总共有三组主 IO 引出口：P2、P3 和 P6。其中，P2 和 P3 分别采用 2*22 排针引出，总共引出 86 个 IO 口，P6 采用 1*16 排针，按顺序引出 FSMC_D0~D15

等 16 个 IO 口。另外，还通过：P4、P8、P9 和 P10 引出 8 个 IO，总共引出 110 个 IO 口。

41. LCD 接口

这是开发板板载的 LCD 模块接口（16 位 80 并口），兼容 ALIENTEK 全系列 LCD 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。

42. RS232 接口（母）

这是开发板板载的另外一个 RS232 接口（COM2），通过一个标准的 DB9 母头和外部的串口连接。通过这个接口，我们可以连接带有串口的电脑或者其他设备，实现串口通信

43. 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！

接下来，我们来看 STM32F767 核心板的资源说明：

1. 5V&3.3V 电源

这里实际上由 3 个焊点组成：5V、3.3V、GND。通过这三个焊点，我们可以给核心板提供电源，也可以由核心板给外部提供电源（3.3V 对外供电时，电流不要超过 300mA）。方便应用到各种场景中去。

2. CPU

这是核心板的 CPU（U1），型号为：STM32F767IGT6。该芯片采用六级流水线，自带指令和数据 Cache、集成 JPEG 编解码器、集成双精度硬件浮点计算单元（DPFPU）和 DSP 指令，并具有 512KB SRAM、1024KB FLASH、13 个 16 位定时器、2 个 32 位定时器、2 个 DMA 控制器（共 16 个通道）、6 个 SPI、1 个 QSPI 接口、3 个全双工 I2S、2 个 SAI、4 个 IIC、8 个串口、2 个 USB（支持 HOST /SLAVE）、3 个 CAN、3 个 12 位 ADC、2 个 12 位 DAC、1 个 SPDIF RX 接口、1 个 RTC（带日历功能）、2 个 SDMMC 接口、1 个 FMC 接口、1 个 TFTLCD 控制器（LTDC）、1 个 10/100M 以太网 MAC 控制器、1 个摄像头接口、1 个硬件随机数生成器、以及 140 个通用 IO 口等。

3. Micro USB 接口

这是核心板的 USB 接口（USB），采用 Micro USB 接口，和手机数据线通用，此接口既可以作为 USB SLAVE 使用，也可以做 USB HOST(OTG)使用，当作为 HOST 使用的时候，需要外接一根 USB OTG 线。同时，这个接口也是核心板电源的主要提供口（单独使用核心板时）。

4. 电源指示灯

这是核心板自带的液晶电源指示灯（PWR），为蓝色。当核心板正常供电时，此 LED 会亮。不过，该 LED 默认受 VREF+控制，当 VREF+悬空时，才正常工作，当 VREF+接 3.3V 时，则一直关闭。想要 LED 不受 VREF+控制，把核心板的 R13 拆了即可。**注意，当核心板插在底板上时，可以通过拔掉底板上 P5 的跳线帽，即可实现 VREF+悬空，从而指示灯亮。**

5. 串口

这是核心板引出的串口 1（USART1），可用于串口通信。注意：排针默认没有焊接，需要自行焊接。

6. LED 指示灯

这是核心板自带的一个状态指示灯（DS0），红色，可以表示程序运行状态，该指示灯与底板上的 DS0 共用一个 IO。同样，当 VREF+悬空时，才正常工作，受限条件同电源指示灯。

7. WKUP 按键

这是核心板板载的一个功能按键（WKUP），并且具有唤醒功能，该按键和底板上的 KEY_UP 共用一个 IO 口（PA0），该按键也是高电平有效。

8. 复位按键

这是核心板板载的复位按键（RST），用于复位 STM32，另外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。此按键和底板上的复位按键功能完全一样。

9. SDRAM

这是核心板外扩的 SDRAM 芯（U3）片，型号为：W9825G6KH，容量为 32M 字节，轻松应对各种大内存需求场景，比如 GUI 设计、算法设计、大数据处理等。

10. NAND FLASH

这是核心板外扩的 NAND FLASH 芯（U4）片，型号为：MT29F4G08，容量为 512M 字节，可以实现大数据存储，满足各种应用需求。另外，大家可以自行更换更大容量的 NAND FLASH，满足项目需要。

11. RGB LCD 接口

这是核心板自带的 RGB LCD 接口（LCD），可以连接各种 ALIENTEK 的 RGB LCD 屏模块，并且支持触摸屏（电阻/电容屏都可以）。为了节省 IO 口，采用的是 RGB565 格式，虽然降低了颜色深度，但是节省了 IO，且 RGB565 格式，程序上更通用一些。

12. SWD 接口

这是核心板自带的调试接口（SWD），可以用于代码下载和仿真调试。采用 SWD 接口，只需最少 3 根线（SWD、SWC 和 GND），即可实现代码下载和仿真调试。注意：排针默认没有焊接，需要自行焊接。

13. EEPROM

这是核心板板载的 EEPROM 芯片（U5），型号为：24C02，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。

14. SPI FLASH

这是核心板外扩的 SPI FLASH 芯片（U6），型号为：W25Q256，容量为 256Mbit，即 32M 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。

最后，STM32F767 核心板的接口，是在底部，通过两个 2*30 的板对板端子（3710M 公座）组成，总共引出了 110 个 IO，通过这个接口，可以实现与阿波罗 STM32 开发板的对接。

1.2.2 软件资源说明

上面我们详细介绍了 ALIENTEK 阿波罗 STM32F767 开发板的硬件资源。接下来，我们将向大家简要介绍一下阿波罗 STM32F767 开发板的软件资源。

阿波罗 STM32F767 开发板提供的标准例程多达 65 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（本手册以库函数版本作为介绍）。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。

阿波罗 STM32F767 开发板的例程列表如表 1.2.2.1 所示：

编号	实验名字	编号	实验名字
1	跑马灯实验	33	DS18B20 数字温度传感器实验
2	按键输入实验	34	DHT11 数字温湿度传感器实验
3	串口通信实验	35	MPU9250 九轴传感器实验
4	外部中断实验	36	无线通信实验

5	独立看门狗实验	37	FLASH 模拟 EEPROM 实验
6	窗口看门狗实验	38	摄像头实验
7	定时器中断实验	39	内存管理实验
8	PWM 输出实验	40	SD 卡实验
9	输入捕获实验	41	NAND FLASH 实验
10	电容触摸按键实验	42	FATFS 实验
11	OLED 实验	43	汉字显示实验
12	内存保护（MPU）实验	44	图片显示实验
13	TFTLCD（MCU 屏）实验	45	硬件 JPEG 解码实验
14	SDRAM 实验	46	照相机实验
15	LTDC LCD（RGB 屏）实验	47	音乐播放器实验
16	USMART 调试实验	48	录音机实验
17	RTC 实验	49	SPDIF(光纤音频)实验
18	硬件随机数实验	50	视频播放器实验
19	待机唤醒实验	51	FPU 测试(Julia 分形)实验
20	ADC 实验	52	DSP 测试实验
21	内部温度传感器实验	53	手写识别实验
22	DAC 实验	54	T9 拼音输入法实验
23	PWM DAC 实验	55	串口 IAP 实验
24	DMA 实验	56	USB 读卡器(Slave)实验
25	IIC 实验	57	USB 声卡(Slave)实验
26	IO 扩展实验	58	USB 虚拟串口(Slave)实验
27	光环境传感器实验	59	USB U 盘(Host)实验
28	QSPI 实验	60	USB 鼠标键盘(Host)实验
29	485 实验	61	网络通信实验
30	CAN 实验	62	UCOSII 实验 1-任务调度
31	触摸屏实验	63	UCOSII 实验 2-信号量和邮箱
32	红外遥控实验	64	UCOSII 实验 3-消息队列、信号量集和软件定时器

表 1.2.2.1 ALIENTEK 阿波罗 STM32F767 开发板例程表

从上表可以看出，ALIENTEK 阿波罗 STM32F767 开发板的例程基本上涵盖了 STM32F767IGT6 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 阿波罗 STM32F767 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 阿波罗 STM32F767 开发板也绝对是一个不错的选择。

1.2.3 阿波罗 IO 引脚分配

为了让大家更快更好的使用我们的阿波罗 STM32F767 开发板，这里特地将阿波罗开发板主芯片：STM32F767IGT6 的 IO 资源分配做了一个总表，以便大家查阅。阿波罗 IO 引脚分配

总表如表 1.2.3.1 所示：

阿波罗 STM32F767 开发板 IO 资源分配表					
引脚	GPIO	连接资源		独立	连接关系说明
40	PA0	WK_UP		Y	1，按键 KEY_UP 2，可以做待机唤醒脚(WKUP)
41	PA1	RMII_REF_CLK		N	接 LAN8720 的 REFCLKO 脚
42	PA2	USART2_TX /RS485_RX	ETH_MDIO	N	1，RS232 串口 2(COM2)RX 脚(P8 设置) 2，RS485 RX 脚(P8 设置) 3，LAN8720 的 MDIO 脚
47	PA3	USART2_RX /RS485_TX	PWM_DAC	N	1，RS232 串口 2(COM2)TX 脚(P8 设置) 2，RS485 TX 脚(P8 设置) 3，PWM_DAC 输出脚
50	PA4	STM_DAC	GBC_LED	Y	1，DAC_OUT1 输出脚 2，ATK-MODULE 接口的 LED 引脚
51	PA5	STM_ADC		Y	ADC 输入引脚，同时做 TPAD 检测脚
52	PA6		DCMI_PCLK	Y	OLED/CAMERA 接口的 PCLK 脚
53	PA7	RMII_CRS_DV		N	接 LAN8720 的 CRS_DV 脚
119	PA8	REMOTE_IN	DCMI_XCLK	N	1，接 HS0038 红外接收头 2，OLED/CAMERA 接口的 XCLK 脚
120	PA9	USART1_TX		Y	串口 1 TX 脚,默认连接 CH340 的 RX(P4 设置)
121	PA10	USART1_RX		Y	串口 1 RX 脚,默认连接 CH340 的 TX(P4 设置)
122	PA11	USB_D-	CAN_RX	Y	1，USB D-引脚(P10 设置) 2，CAN_RX 引脚(P10 设置)
123	PA12	USB_D+	CAN_TX	Y	1，USB D+引脚(P10 设置) 2，CAN_TX 引脚(P10 设置)
124	PA13	JTMS	SWDIO	N	JTAG/SWD 仿真接口,没接任何外设
137	PA14	JTCK	SWDCLK	N	JTAG/SWD 仿真接口,没接任何外设
138	PA15	JTDI	DCMI_RESET	N	1，JTAG 仿真口(JTDI) 2，OLED/CAMERA 接口的 RESET 脚
56	PB0	LED1		N	接 DS1 LED 灯(绿色)
57	PB1	LED0		N	接 DS0 LED 灯(红色)
58	PB2	QSPI_BK1_CLK		N	QSPI 时钟信号，接 W25Q256
161	PB3	JTDO	DCMI_SDA	N	1，JTAG 仿真口(JTDO) 2，OLED/CAMERA 接口的 SDA 脚
162	PB4	JTRST	DCMI_SCL	N	1，JTAG 仿真口(JTRST) 2，OLED/CAMERA 接口的 SCL 脚
163	PB5	LCD_BL		Y	TFTLCD 接口背光控制脚
164	PB6	QSPI_BK1_NCS		N	QSPI 片选信号，接 W25Q256
165	PB7		DCMI_VSYNC	Y	OLED/CAMERA 接口的 VSYNC 脚
167	PB8		DCMI_D6	Y	OLED/CAMERA 接口的 D6 脚
168	PB9		DCMI_D7	Y	OLED/CAMERA 接口的 D7 脚

79	PB10	USART3_TX		Y	1，RS232 串口 3(COM3)RX 脚(P9 设置) 2，ATK-MODULE 接口的 RXD 脚(P9 设置)
80	PB11	USART3_RX	RMII_TX_EN	N	1，RS232 串口 3(COM3)TX 脚(P9 设置) 2，ATK-MODULE 接口的 TXD 脚(P9 设置) 3，接 LAN8720 的 TXEN 脚
92	PB12	IIC_INT	1WIRE_DQ	N	1，接 PCF8574 的 INT 脚 2，接单总线接口(U10),即 DHT11/DS18B20
93	PB13	SPI2_SCK		Y	接 WIRELESS 接口的 SCK 信号
94	PB14	SPI2_MISO		Y	接 WIRELESS 接口的 MISO 信号
95	PB15	SPI2_MOSI		Y	接 WIRELESS 接口的 MOSI 信号
32	PC0	FMC_SDNWE		N	接 SDRAM 的 WE 脚
33	PC1	ETH_MDC		N	接 LAN8720 的 MDC 脚
34	PC2	FMC_SDNE0		N	接 SDRAM 的 CS 脚
35	PC3	FMC_SDCKE0		N	接 SDRAM 的 CKE 脚
54	PC4	RMII_RXD0		N	接 LAN8720 的 RXD0 脚
55	PC5	RMII_RXD1		N	接 LAN8720 的 RXD1 脚
115	PC6		DCMI_D0	Y	OLED/CAMERA 接口的 D0 脚
116	PC7		DCMI_D1	Y	OLED/CAMERA 接口的 D1 脚
117	PC8	SDIO_D0	DCMI_D2	N	1，SD 卡接口的 D0 2，OLED/CAMERA 接口的 D2 脚
118	PC9	SDIO_D1	DCMI_D3	N	1，SD 卡接口的 D1 2，OLED/CAMERA 接口的 D3 脚
139	PC10	SDIO_D2		N	SD 卡接口的 D2
140	PC11	SDIO_D3	DCMI_D4	N	1，SD 卡接口的 D3 2，OLED/CAMERA 接口的 D4 脚
141	PC12	SDIO_SCK		Y	SD 卡接口的 SCK
8	PC13	KEY2		Y	接按键 KEY2
9	PC14		RTC 晶振	N	接 32.768K 晶振，不可用做 IO
10	PC15		RTC 晶振	N	接 32.768K 晶振，不可用做 IO
142	PD0	FMC_D2		N	FMC 总线数据线 D2(LCD/SDRAM/NAND 共用)
143	PD1	FMC_D3		N	FMC 总线数据线 D3(LCD/SDRAM/NAND 共用)
144	PD2	SDIO_CMD		N	SD 卡接口的 CMD
145	PD3		DCMI_D5	Y	OLED/CAMERA 接口的 D5 脚
146	PD4	FMC_NOE		N	FMC 总线 NOE(RD)(LCD/NAND 共用)
147	PD5	FMC_NWE		N	FMC 总线 NWE(WR)(LCD/NAND 共用)
150	PD6	FMC_NWAIT		N	FMC 总线 NWAIT(NAND 用)
151	PD7	FMC_NE1		Y	FMC 总线的片选信号 1，为 TFTLCD 片选信号
96	PD8	FMC_D13		N	FMC 总线数据线 D13(LCD/SDRAM 共用)
97	PD9	FMC_D14		N	FMC 总线数据线 D14(LCD/SDRAM 共用)
98	PD10	FMC_D15		N	FMC 总线数据线 D15(LCD/SDRAM 共用)
99	PD11	FMC_A16_CLE		N	FMC 总线地址线 A16_CLE(NAND 专用)
100	PD12	FMC_A17_ALE		N	FMC 总线地址线 A17_ALE(NAND 专用)

101	PD13	FMC_A18		N	FMC 总线地址线 A18(LCD 专用)
104	PD14	FMC_D0		N	FMC 总线数据线 D0(LCD/SDRAM/NAND 共用)
105	PD15	FMC_D1		N	FMC 总线数据线 D1(LCD/SDRAM/NAND 共用)
169	PE0	FMC_NBL0		N	FMC 总线 NBL0(SDRAM 专用)
170	PE1	FMC_NBL1		N	FMC 总线 NBL1(SDRAM 专用)
1	PE2	SAI1_MCLKA		N	WM8978 的 MCLK 信号
2	PE3	SAI1_SDB		N	WM8978 的 ADCDAT 信号
3	PE4	SAI1_FSA		N	WM8978 的 LRC 信号
4	PE5	SAI1_SCKA		N	WM8978 的 BCLK 信号
5	PE6	SAI1_SDA		N	WM8978 的 DACDAT 信号
68	PE7	FMC_D4		N	FMC 总线数据线 D4(LCD/SDRAM/NAND 共用)
69	PE8	FMC_D5		N	FMC 总线数据线 D5(LCD/SDRAM/NAND 共用)
70	PE9	FMC_D6		N	FMC 总线数据线 D6(LCD/SDRAM/NAND 共用)
73	PE10	FMC_D7		N	FMC 总线数据线 D7(LCD/SDRAM/NAND 共用)
74	PE11	FMC_D8		N	FMC 总线数据线 D8(LCD/SDRAM 共用)
75	PE12	FMC_D9		N	FMC 总线数据线 D9(LCD/SDRAM 共用)
76	PE13	FMC_D10		N	FMC 总线数据线 D10(LCD/SDRAM 共用)
77	PE14	FMC_D11		N	FMC 总线数据线 D11(LCD/SDRAM 共用)
78	PE15	FMC_D12		N	FMC 总线数据线 D12(LCD/SDRAM 共用)
16	PF0	FMC_A0		N	FMC 总线地址线 A0(SDRAM 专用)
17	PF1	FMC_A1		N	FMC 总线地址线 A1(SDRAM 专用)
18	PF2	FMC_A2		N	FMC 总线地址线 A2(SDRAM 专用)
19	PF3	FMC_A3		N	FMC 总线地址线 A3(SDRAM 专用)
20	PF4	FMC_A4		N	FMC 总线地址线 A4(SDRAM 专用)
21	PF5	FMC_A5		N	FMC 总线地址线 A5(SDRAM 专用)
24	PF6	QSPI_BK1_IO3		N	QSPI 数据线，接 W25Q256
25	PF7	QSPI_BK1_IO2		N	QSPI 数据线，接 W25Q256
26	PF8	QSPI_BK1_IO1		N	QSPI 数据线，接 W25Q256
27	PF9	QSPI_BK1_IO0		N	QSPI 数据线，接 W25Q256
28	PF10		LCD_DE	Y	RGBLCD 接口的 DE 信号
59	PF11	FMC_SDNRAS		N	接 SDRAM 的 RAS 脚
60	PF12	FMC_A6		N	FMC 总线地址线 A6(SDRAM 专用)
63	PF13	FMC_A7		N	FMC 总线地址线 A7(SDRAM 专用)
64	PF14	FMC_A8		N	FMC 总线地址线 A8(SDRAM 专用)
65	PF15	FMC_A9		N	FMC 总线地址线 A9(SDRAM 专用)
66	PG0	FMC_A10		N	FMC 总线地址线 A10(SDRAM 专用)
67	PG1	FMC_A11		N	FMC 总线地址线 A11(SDRAM 专用)
106	PG2	FMC_A12		N	FMC 总线地址线 A12(SDRAM 专用)
107	PG3	T_MISO		Y	TFTLCD/RGBLCD 接口触摸屏 MOSI 信号
108	PG4	FMC_BA0		N	接 SDRAM 的 BA0 脚
109	PG5	FMC_BA1		N	接 SDRAM 的 BA1 脚
110	PG6		LCD_R7	Y	RGBLCD 接口的 R7 数据线

111	PG7		LCD_CLK	Y	RGBLCD 接口的 CLK 信号
112	PG8	FMC_SDCLK		N	接 SDRAM 的 CLK 脚
152	PG9	FMC_NCE3		N	FMC 总线的片选信号 3，为 NAND 片选信号
153	PG10	NRF_CS		Y	WIRELESS 接口 CS 信号
154	PG11		LCD_B3	Y	RGBLCD 接口的 B3 数据线
155	PG12	NRF_CE	SPDIF_RX	N	1，WIRELESS 接口 CE 信号 2，SPDIF 输入引脚（仅 F7 芯片支持）
156	PG13	RMII_TXD0		N	接 LAN8720 的 TXD0 脚
157	PG14	RMII_TXD1		N	接 LAN8720 的 TXD1 脚
160	PG15	FMC_SDNCAS		N	接 SDRAM 的 CAS 脚
29	PH0		系统晶振	N	接 25M 外部晶振
30	PH1		系统晶振	N	接 25M 外部晶振
43	PH2	KEY1		Y	接按键 KEY1
44	PH3	KEY0		Y	接按键 KEY0
45	PH4	IIC_SCL		N	接 24C02、PCF8574、MPU9250、AP3216C 和 WM8978 的 SCL
46	PH5	IIC_SDA		N	接 24C02、PCF8574、MPU9250、AP3216C 和 WM8978 的 SDA
83	PH6	T_SCK		Y	TFTLCD/RGBLCD 接口触摸屏 SCK 信号
84	PH7	T_PEN		Y	TFTLCD/RGBLCD 接口触摸屏 PEN 信号
85	PH8		DCMI_HREF	Y	OLED/CAMERA 接口的 HREF 脚
86	PH9		LCD_R3	Y	RGBLCD 接口的 R3 数据线
87	PH10		LCD_R4	Y	RGBLCD 接口的 R4 数据线
88	PH11		LCD_R5	Y	RGBLCD 接口的 R5 数据线
89	PH12		LCD_R6	Y	RGBLCD 接口的 R6 数据线
128	PH13		LCD_G2	Y	RGBLCD 接口的 G2 数据线
129	PH14		LCD_G3	Y	RGBLCD 接口的 G3 数据线
130	PH15		LCD_G4	Y	RGBLCD 接口的 G4 数据线
131	PI0		LCD_G5	Y	RGBLCD 接口的 G5 数据线
132	PI1		LCD_G6	Y	RGBLCD 接口的 G6 数据线
133	PI2		LCD_G7	Y	RGBLCD 接口的 G7 数据线
134	PI3	T_MOSI		Y	TFTLCD/RGBLCD 接口触摸屏 PEN 信号
173	PI4		LCD_B4	Y	RGBLCD 接口的 B4 数据线
174	PI5		LCD_B5	Y	RGBLCD 接口的 B5 数据线
175	PI6		LCD_B6	Y	RGBLCD 接口的 B6 数据线
176	PI7		LCD_B7	Y	RGBLCD 接口的 B7 数据线
7	PI8	T_CS		Y	TFTLCD/RGBLCD 接口触摸屏 CS 信号
11	PI9		LCD_VSYNC	Y	RGBLCD 接口的 VSYNC 信号线
12	PI10		LCD_HSYNC	Y	RGBLCD 接口的 HSYNC 信号线
13	PI11	NRF_IRQ	GBC_KEY	Y	1，WIRELESS 接口 IRQ 信号 2，ATK-MODULE 接口的 KEY 引脚

表 1.2.3.1 阿波罗 IO 资源分配总表

表 1.2.3.1 中，引脚栏即 STM32F767IGT6 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 阿波罗 STM32F767 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。

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 阿波罗 STM32F7 开发板各部分（包括底板和核心板）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，开发板底板原理图详解；

2.2，STM32F7 核心板原理图详解

2.3，开发板使用注意事项；

2.4，STM32F7 学习方法；

2.1 开发板底板原理图详解

2.1.1 核心板接口

阿波罗 STM32F767 开发板采用底板+核心板的形式，使得一块底板，可以学习多款 MCU，提高资源利用率，从而降低学习成本。阿波罗 STM32 开发板底板采用 2 个 2*30 的 3710F（母座）板对板连接器来同核心板连接，接插非常方便，底板上面的核心板接口原理图如图 2.1.1.1 所示：

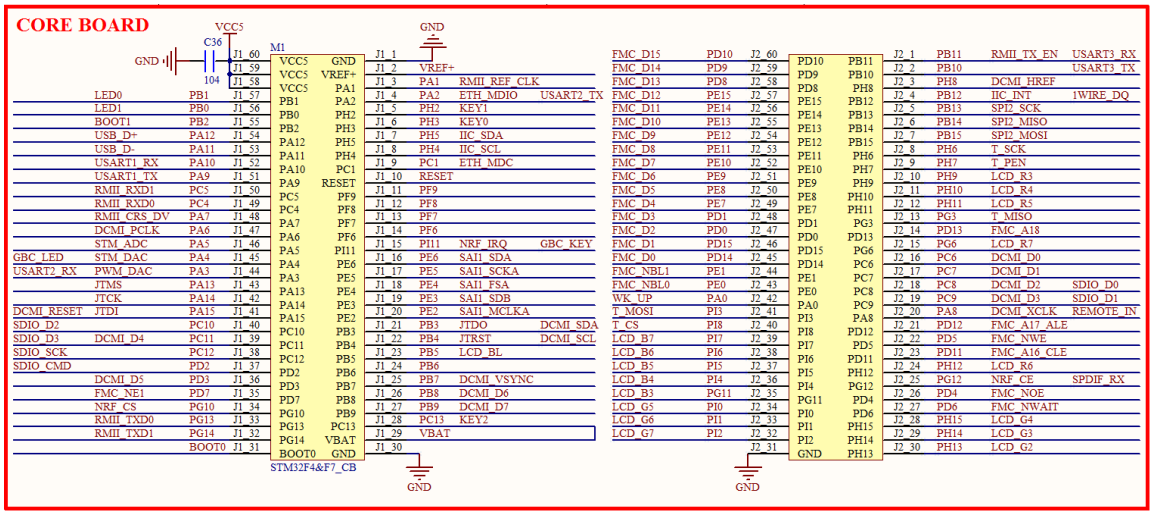


图 2.1.1.1 底板核心板接口部分原理图

图中的 M1 就是底板上的核心板接口，由 2 个 2*30PIN 的 3710F 板对板母座组成，总共引出了核心板上面 110 个 IO 口，另外，还有：6 根电源线（VCC/GND 各占 3 根）、BOOT1、VBAT、RESET 和 VREF+。

2.1.2 引出 IO 口

阿波罗 STM32F767 开发板底板上面，总共引出了 STM32F767IGT6 的 110 个 IO 口，如图 2.1.2.1 所示：

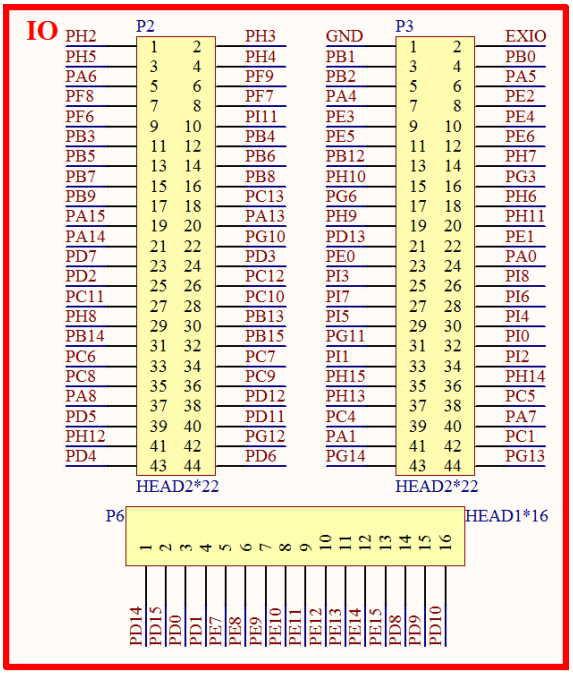


图 2.1.2.1 引出 IO 口

图中 P2、P3 和 P6 为 MCU 主 IO 引出口，这三组排针共引出了 102 个 IO 口，另外，通过：P4（PA9&PA10）、P8（PA2&PA3）、P9（PB10&PB11）和 P10（PA11&PA12）等 4 组排针引出 8 个 IO 口，这样底板上总共引出了 110 个 IO。STM32F767IGT6 总共有 140 个 IO，剩下的 30 个 IO，主要用在了晶振、SDRAM、RGBLCD 等常用外设上面，不太适合再引出来做其他用，所以，我们就没有引出来了。

2.1.3 USB 串口/串口 1 选择接口

阿波罗 STM32F767 开发板板载的 USB 串口和 STM32F767IGT6 的串口是通过 P4 连接起来的，如图 2.1.3.1 所示：

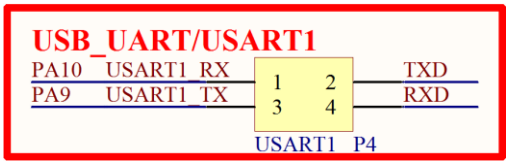


图 2.1.3.1 USB 串口/串口 1 选择接口

图中 TXD/RXD 是相对 CH340G 来说的，也就是 USB 串口的发送和接受脚。而 USART1_RX 和 USART1_TX 则是相对于 STM32F767IGT6 来说的。这样，通过对接，就可以实现 USB 串口和 STM32F767IGT6 的串口通信了。同时，P4 是 PA9 和 PA10 的引出口。

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

2.1.4 JTAG/SWD

阿波罗 STM32F767 开发板板载的标准 20 针 JTAG/SWD 接口电路如图 2.1.4.1 所示：

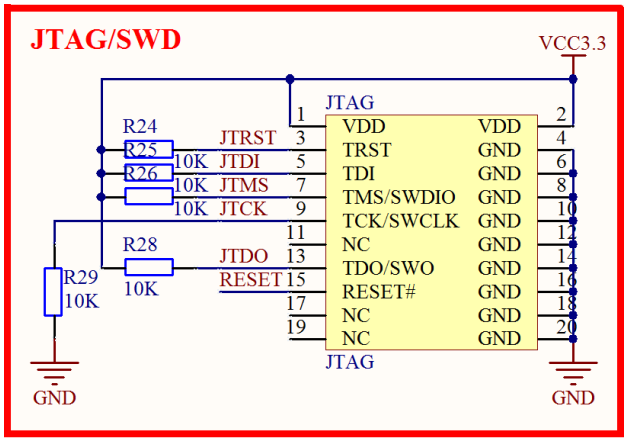


图 2.1.4.1 JTAG/SWD 接口

这里，我们采用的是标准的 JTAG 接法（支持 SWD），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，也能调试。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(**必须是 V9 或者以上版本**)、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 参考电压选择端口

阿波罗 STM32F767 开发板板载了一个参考电压选择口，如图 2.1.5.1 所示：

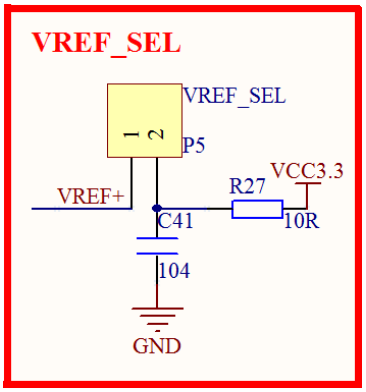


图 2.1.5.1 参考电压选择端口

图中 VREF_SEL 默认用跳线帽连接 1&2 脚，从而 VREF+=3.3V，即 STM32 芯片的 ADC/DAC 参考电压，默认是 3.3V 的。如果大家想用自己的参考电压，则把你的参考电压接入 VREF+即可（注意，还要共地）。

特别注意：该接口还是控制核心板 LED 的总开关，当 VREF+接 3.3V 时（插跳线帽），核心板所有 LED（PWR&DS0）都不工作，当 VREF+悬空时（拔掉跳线帽），核心板所有 LED 都正常工作。如不想让此接口控制核心板的 LED，那么请拆除核心板的 R14 电阻即可。

2.1.6 LCD 模块接口

阿波罗 STM32F767 开发板板载的 LCD 模块接口电路如图 2.1.6.1 所示：

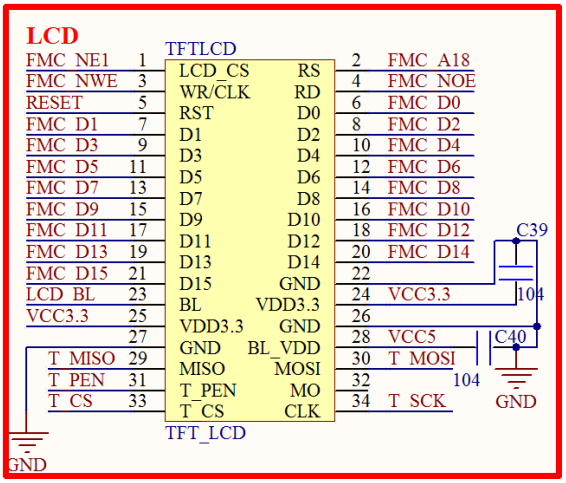


图 2.1.6.1 LCD 模块接口

图中 TFT_LCD 是一个通用的液晶模块接口，采用 16 位 80 并口，也称作 MCU 屏接口，仅支持 MCU 接口的液晶（不支持 RGB 接口的液晶）， ALIENTEK 的 MCU 接口 TFTLCD 模块有：2.4 寸、2.8 寸、3.5 寸、4.3 寸和 7 寸等尺寸。LCD 接口连接在 STM32F767IGT6 的 FSMC 总线上面，可以显著提高 LCD 的刷屏速度。

图中的 T_MISO/T_MOSI/T_PEN/T_CS/T_SCK 连接在 MCU 的 PG3/PI3/PH7/PI8/PH6 上，用于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 PB5 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。**特别注意：**该接口核心板上的 RGBLCD（RGB 屏）接口，共用触摸屏和背光信号线，所以他们不能同时都使用触摸屏！！！

2.1.7 复位电路

阿波罗 STM32F767 开发板的复位电路如图 2.1.7.1 所示：

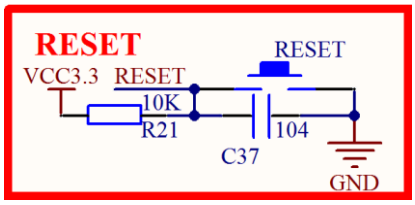


图 2.1.7.1 复位电路

因为 STM32 是低电平复位的，所以我们设计的电路也是低电平复位的，这里的 R21 和 C37 构成了上电复位电路。同时，开发板把 LCD 接口的复位引脚也接在 RESET 上，这样这个复位按钮不仅可以用来复位 MCU，还可以复位 LCD。

2.1.8 启动模式设置接口

阿波罗 STM32F767 开发板的启动模式设置端口电路如图 2.1.8.1 所示：

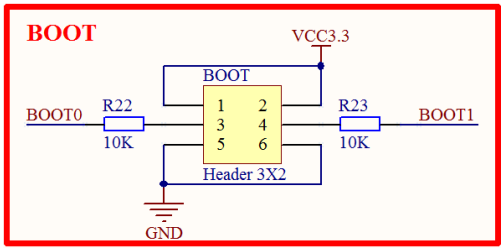


图 2.1.8.1 启动模式设置接口

在 STM32F7 系列的芯片上，图中的 BOOT0 和 BOOT1 只有 BOOT0 有效，对应 STM32F7 芯片的 B00T 引脚。STM32F7 的启动模式（也称自举模式），如表 2.1.8.1 所示：

启动模式选择		启动地址
B00T0	启动地址选项字节	
0	BOOT_ADD0[15:0]	由用户选项字节 BOOT_ADD0[15:0]决定启动地址,ST 出厂默认的启动地址为:0X0020 0000 的 ITCM 上的 Flash
1	BOOT_ADD1[15:0]	由用户选项字节 BOOT_ADD1[15:0]决定启动地址,ST 出厂默认的启动地址为:0X0010 0000 的 ITCM 上的 Flash
BOOT_ADDx = 0x0000：从 ITCM RAM(0x0000 0000)启动 BOOT_ADDx = 0x0040：从系统存储器(0x0010 0000)启动 BOOT_ADDx = 0x0080：从 ITCM 接口上的 FLASH (0x0020 0000)启动 BOOT_ADDx = 0x2000：从 AXIM 接口上的 FLASH (0x0800 0000)启动 BOOT_ADDx = 0x8000：从 DTCM RAM(0x2000 0000)启动 BOOT_ADDx = 0x8004：从 SRAM1(0x2001 0000)启动 BOOT_ADDx = 0x8013：从 SRAM2(0x2004 C000)启动 x=0/1，出厂时：BOOT_ADD0=0X0080；BOOT_ADD1=0X0040；		

表 2.1.8.1 启动模式选择表

按照表 2. 1.8.1，一般情况下我们设置 B00T0 为低电平即可，默认情况下系统通过 ITCM 总线接口访问 FLASH（地址从 0X0020 0000 开始）。

这里需要注意两点：

1，STM32F7 虽然也支持串口下载（B00T0=1，从系统存储器启动），但目前没有比较好的支持 STM32F7 的串口下载软件，所以，大家必须自备 ST LINK V2 仿真器一个，用来下载和调试代码。

2，STM32F7 实际上只有一个 FLASH 存储器，但是有两条访问路径：ITCM 和 AXIM，他们访问 FLASH 的地址映射是不一样的，ITCM 是从 0X0020 0000 开始的 1MB 访问空间，AXIM 则是从 0X0800 0000 开始的 1MB 访问空间。我们通过 MDK 将代码下载到 0X0020 0000 还是 0X0800 0000 都是可以正常运行的，因为实际上只有一个 FLASH，只是地址映射不一样而已。我们在 MDK 里面，一般设置 FLASH 地址为 0X0800 0000。

2.1.9 VBAT 供电接口

阿波罗 STM32F767 开发板的 VBAT 供电电路如图 2.1.9.1 所示：

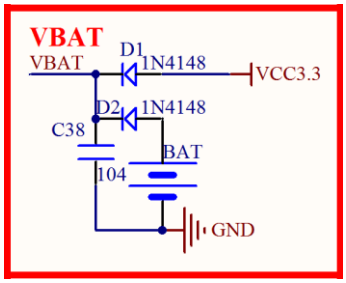


图 2.1.9.1 启动模式设置接口

上图的 VBAT 接 MCU 的 VBAT 脚，从而给核心板的后备区域供电，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。

2.1.10 RS232 串口

阿波罗 STM32F767 开发板板载了一公一母两个 RS232 接口，电路原理图如图 2.1.10.1 所示：

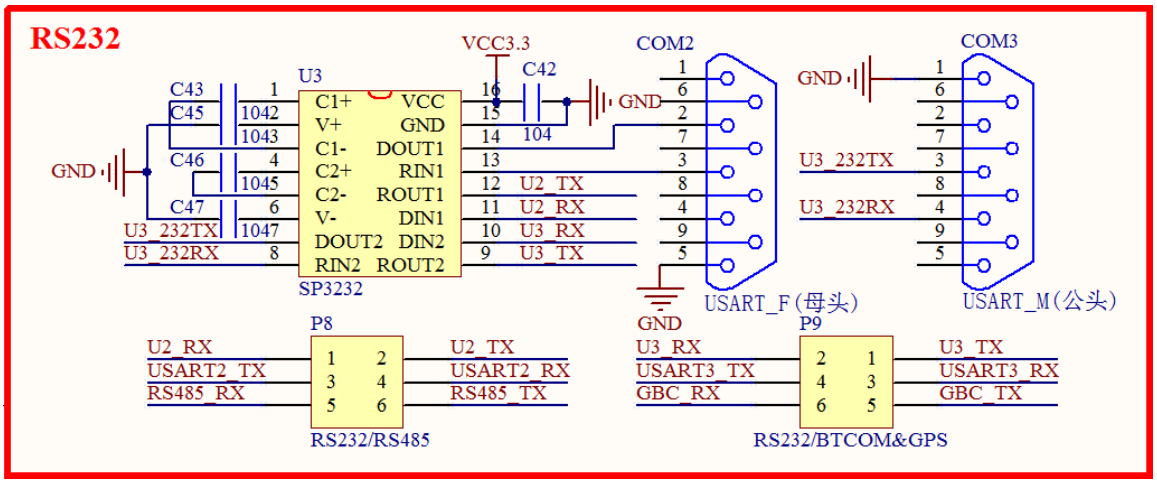


图 2.1.10.1 RS232 串口

因为 RS232 电平不能直接连接到 STM32，所以需要一个电平转换芯片。这里我们选择的是 SP3232（也可以用 MAX3232）来做电平转接，同时图中的 P8 用来实现 RS232(COM2)/RS485 的选择，P9 用来实现 RS232(COM3)/ATK 模块接口的选择，以满足不同实验的需要。

图中 USART2_TX/USART2_RX 连接在 MCU 的串口 2 上（PA2/PA3），所以这里的 RS232(COM2)/RS485 都是通过串口 2 来实现的。图中 RS485_TX 和 RS485_RX 信号接在 SP3485 的 DI 和 RO 信号上。

而图中的 USART3_TX/USART3_RX 则是连接在 MCU 的串口 3 上（PB10/PB11），所以 RS232(COM3)/ATK 模块接口都是通过串口 3 来实现的。图中 GBC_RX 和 GBC_TX 连接在 ATK 模块接口 U5 上面。

因为 P8/P9 的存在，其实还带来另外一个好处，就是我们可以把开发板变成一个 RS232 电平转换器，或者 RS485 电平转换器，比如你买的核心板，可能没有板载 RS485/RS232 接口，通过连接我们开发板的 P8/P9 端口，就可以让你的核心板拥有 RS232/RS485 的功能。

2.1.11 RS485 接口

阿波罗 STM32F767 开发板板载的 RS485 接口电路如图 2.1.11.1 所示：

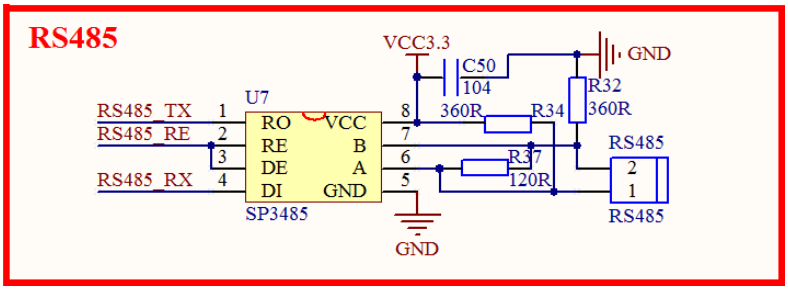


图 2.1.11.1 RS485 接口

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 R37 为终端匹配电阻，而 R34 和 R32，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 P8 上面，通过 P8 跳线来选择是否连接在 MCU 上面，RS485_RE 则是连接在 PCF8574（IIC IO 扩展芯片）的 P6 引脚上的，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）。

2.1.12 CAN/USB 接口

ALIENTEK 阿波罗 STM32F767 开发板板载的 CAN 接口电路以及 STM32 USB 接口电路如图 2.1.12.1 所示：

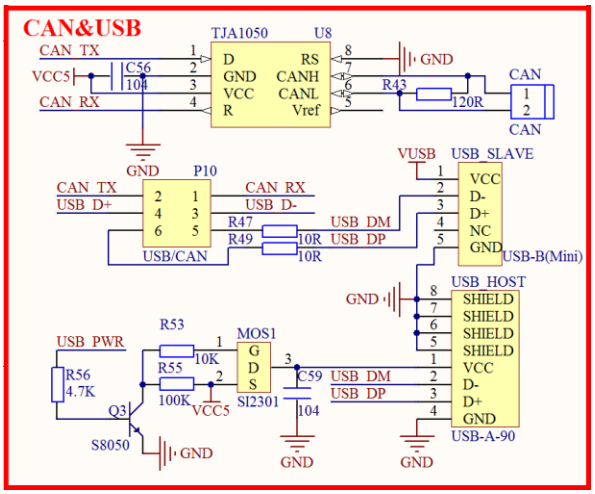


图 2.1.12.1 CAN/USB 接口

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 R43 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 P10 来选择使用 USB 还是 CAN。

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

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

USB HOST 可以用来接如：U 盘、USB 鼠标、USB 键盘和 USB 手柄等设备，实现 USB 主机功能。该接口可以对从设备供电，供电受 USB_PWR 控制。USB_PWR 信号连接在 PCF8574（IIC IO 扩展芯片）的 P3 引脚上。

2.1.13 光环境传感器

阿波罗 STM32F767 开发板板载了一个光环境传感器，可以用来感应周围光线强度、接近距离和红外线强度等，该部分电路如图 2.1.13.1 所示：

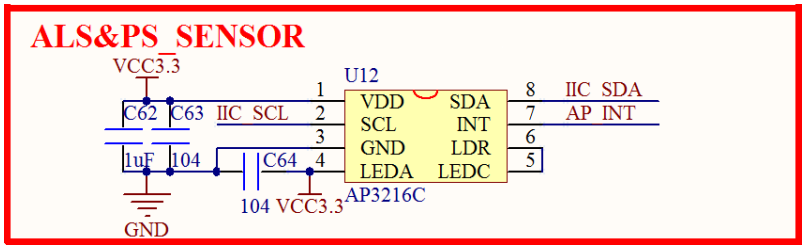


图 2.1.13.1 光环境传感器电路

图中的 U12 就是光环境传感器：AP3216C，它集成了光照强度、近距离、红外三个传感器功能于一身，被广泛应用于各种智能手机。该芯片采用 IIC 接口，IIC_SCL 和 IIC_SDA 分别连接 PH4 和 PH5 上，AP_INT 是其中断输出脚，连接在 PCF8574（IIC IO 扩展芯片）的 P1 引脚上。

2.1.14 IIC IO 扩展

阿波罗 STM32F767 开发板板载了一个 IIC IO 扩展芯片，电路如图 2.1.14.1 所示：

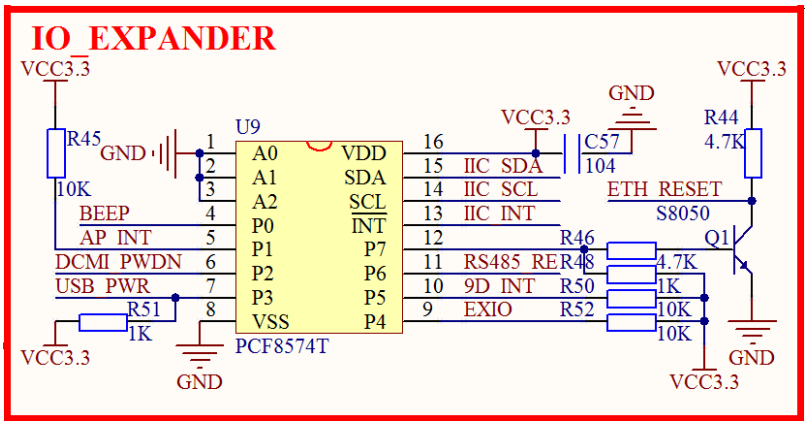


图 2.1.14.1 IIC IO 扩展芯片

IIC IO 扩展芯片型号为：PCF8574/AT8574（这两个芯片完全互相兼容，可互相替换），该芯片通过 IIC 接口，可以扩展出 8 个 IO。这里我们利用扩展的 IO 连接了：蜂鸣器（BEEP）、光环境传感器（AP_INT）、OLED/CAMERA 接口（DCMI_PWDN）、USB HOST 接口（USB_PWR）、九轴传感器（9D_INT）、RS485 接口（RS485_RE）和网络接口（ETH_RESET）等。多余的一个扩展 I（EXIO）O，通过 P3 排针引出。

同 AP3216C 一样，该芯片的 IIC_SCL 和 IIC_SDA 同样是挂在 PH4 和 PH5 上，他们共享一个 IIC 总线。IIC_INT 连接在 PB12 上，**特别注意：**PB12 还连接了单总线接口的 1WIRE_DQ 信号，所以，单总线接口和 IIC_INT 不能同时使用。

2.1.15 九轴传感器

阿波罗 STM32F767 开发板板载了一个九轴传感器，电路如图 2.1.15.1 所示：

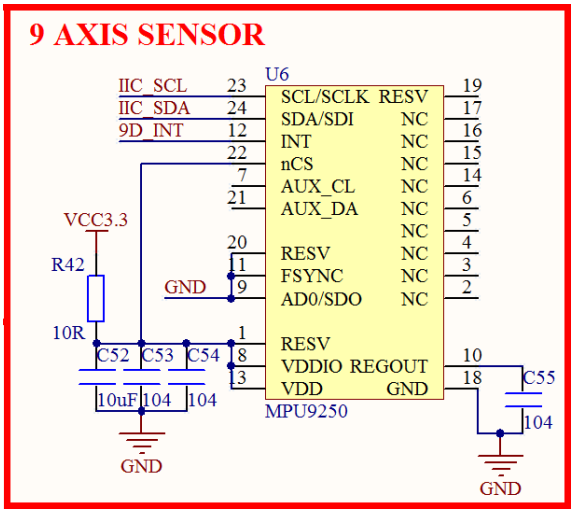


图 2.1.15.1 3D 加速度传感器

九轴传感器芯片型号为：MPU9250，该芯片内部集成了：三轴加速度传感器、三轴陀螺仪和三轴磁力计。并且自带 DMP(Digital Motion Processor)，支持 MPL，该传感器可以用于四轴飞行器的姿态控制和解算。这里我们使用 IIC 接口来访问。

同 AP3216C 一样，该芯片的 IIC_SCL 和 IIC_SDA 同样是挂在 PH4 和 PH5 上，他们共享一个 IIC 总线。9D_INT 是其中断输出脚，连接在 PCF8574（IIC IO 扩展芯片）的 P5 引脚上。

2.1.16 温湿度传感器接口

阿波罗 STM32F767 开发板板载了一个温湿度传感器接口，电路如图 2.1.16.1 所示：

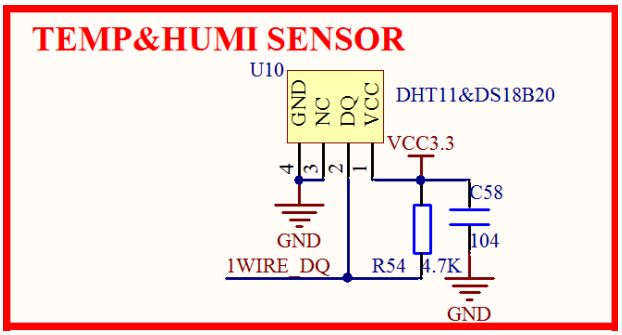


图 2.1.16.1 温湿度传感器接口

该接口支持 DS18B20/DS1820/DHT11 等单总线数字温湿度传感器。1WIRE_DQ 是传感器的数据线，该信号连接在 MCU 的 PB12 上，**特别注意：**该引脚同时还接了 IIC_INT 信号，所以，单总线接口和 IIC_INT，不能同时使用，但可以分时复用。

2.1.17 红外接收头

阿波罗 STM32F767 开发板板载了一个红外接收头，电路如图 2.1.17.1 所示：

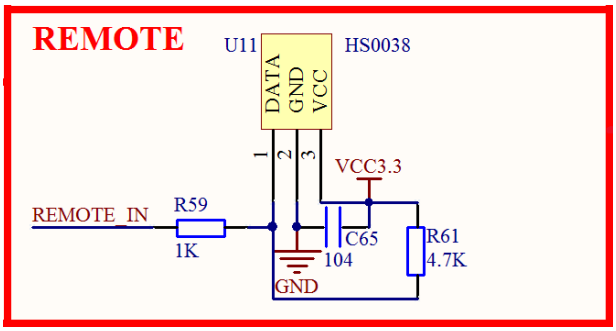


图 2.1.17.1 红外接收头

HS0038 是一个通用的红外接收头，几乎可以接收市面上所有红外遥控器的信号，有了它，就可以用红外遥控器来控制开发板了。REMOTE_IN 为红外接收头的输出信号，该信号连接在 MCU 的 PA8 上。**特别注意：**PA8 同时连接了 DCMI_XCLK，如果要用到 DCMI_XCLK 的时候，HS0038 就不能同时使用了，但可以分时复用。

2.1.18 WIRELESS 模块接口

阿波罗 STM32F767 开发板板载了一个 WIRELESS 模块接口，电路如图 2.1.18.1 所示：

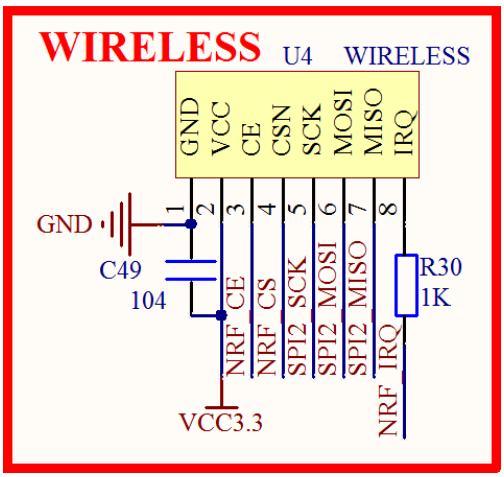


图 2.1.18.1 无线模块接口

该接口用来连接 NRF24L01、SPI WIFI 模块等无线模块，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）。

NRF_CE/NRF_CS/NRF_IRQ 连接在 MCU 的 PG12/PG10/PI11 上，而另外 3 个 SPI 信号则接 MCU 的 SPI2（PB13/PB14/PB15）。这里**需要注意**的是 PI11 还接了 ATK-MODULE 接口的 KEY 信号（GBC_KEY），所以在使用 WIRELESS 中断引脚的时候，不能和 ATK-MODULE 接口同时使用，不过，如果没用到 WIRELESS 的中断引脚，那么 ATK-MODUL 接口和 WIRELESS 模块就可以同时使用了。另外，PG12 同时还连接了光纤输入信号（SPDIF_RX），所以，光纤输入和 WIRELESS 接口，也不能同时使用。

2.1.19 LED

阿波罗 STM32F767 开发板板载总共有 3 个 LED，其原理图如图 2.1.19.1 所示：

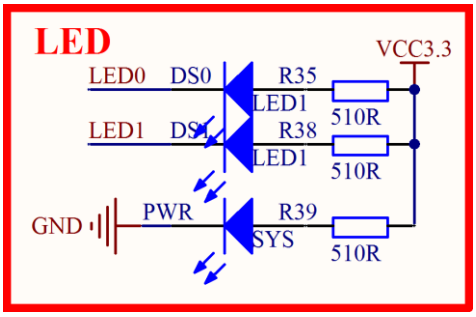


图 2.1.19.1 LED

其中 PWR 是系统电源指示灯，为蓝色。LED0(DS0)和 LED1(DS1)分别接在 PB1 和 PB0。为了方便大家判断，我们选择了 DS0 为红色的 LED，DS1 为绿色的 LED。

2.1.20 按键

阿波罗 STM32F767 开发板板载总共有 4 个输入按键，其原理图如图 2.1.20.1 所示：

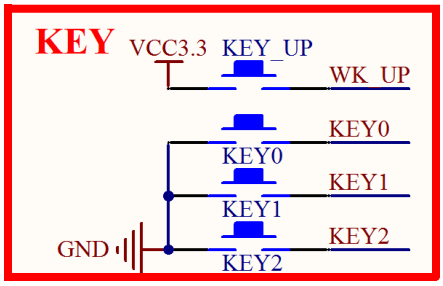


图 2.1.20.1 输入按键

KEY0、KEY1 和 KEY2 用作普通按键输入，分别连接在 PH3、PH2 和 PC13 上，这里并没有使用外部上拉电阻，但是 STM32 的 IO 作为输入的时候，可以设置上下拉电阻，所以我们使用 STM32 的内部上拉电阻来为按键提供上拉。

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

2.1.21 TPAD 电容触摸按键

阿波罗 STM32F767 开发板板载了一个电容触摸按键，其原理图如图 2.1.21.1 所示：

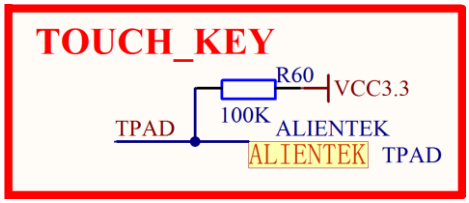


图 2.1.21.1 电容触摸按键

图中 1M 电阻是电容充电电阻，TPAD 并没有直接连接在 MCU 上，而是连接在多功能端口（P11）上面，通过跳线帽来选择是否连接到 STM32。多功能端口，我们将在 2.1.26 节介绍。

电容触摸按键的原理我们将在后续的实战篇里面介绍。

2.1.22 OLED/摄像头模块接口

阿波罗 STM32F767 开发板板载了一个 OLED/摄像头模块接口，连接在 MCU 的硬件摄像

头接口（DCMI）上面，其原理图如图 2.1.22.1 所示：

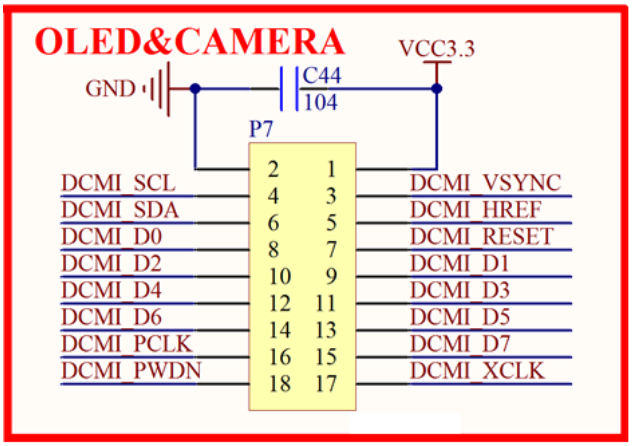


图 2.1.22.1 OLED/摄像头模块接口

图中 P7 是接口可以用来连接 ALIENTEK OLED 模块或者 ALIENTEK 摄像头模块。如果是 OLED 模块，则 DCMI_PWDN 和 DCMI_XCLK 不需要接（在板上靠左插即可），如果是摄像头模块，则需要用到全部引脚。

其中，DCMI_SCL/DCMI_SDA/DCMI_RESET/DCMI_XCLK/DCMI_PWDN 这 5 个信号是不属于 STM32F767 硬件摄像头接口的信号，通过普通 IO 控制即可，前四根线分别接在 MCU 的：PB4/PB3/PA15/PA8 上面，DCMI_PWDN 则连接在 PCF8574（IIC IO 扩展芯片）的 P2 引脚上。

特别注意：DCMI_SCL、DCMI_SDA 和 DCMI_RESET 和 JTAG 接口共用 IO，所以，使用摄像头的时候，不能用 JTAG 调试/下载代码，但是 SWD 模式调试不受影响，这也是为什么我们**极力推荐使用 SWD 模式**。另外，DCMI_XCLK 和 REMOTE_IN 共用 IO，他们不可以同时使用，不过可以分时复用。

其他信号全接 MCU 的硬件摄像头接口：DCMI_VSYNC/DCMI_HREF/DCMI_D0/DCMI_D1/ DCMI_D2/ DCMI_D3/ DCMI_D4/ DCMI_D5/ DCMI_D6/ DCMI_D7/ DCMI_PCLK 分别连接在：PB7/PH8/PC6/PC7/PC8/PC9/PC11/PD3/PB8/PB9/PA6 上。**特别注意：**这些信号和 SD 卡有 IO 共用，所以在使用 OLED 模块或摄像头模块的时候，不能和 SD 卡同时使用，只能分时复用。

2.1.23 有源蜂鸣器

阿波罗 STM32F767 开发板板载了一个有源蜂鸣器，其原理图如图 2.1.23.1 所示：

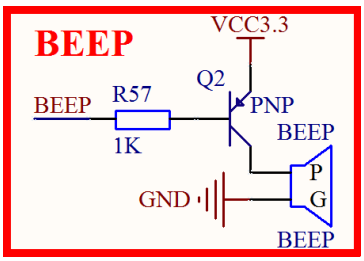


图 2.1.23.1 有源蜂鸣器

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

BEEP 信号直接连接在 PCF8574（IIC IO 扩展芯片）的 P0 引脚上，需要通过 IIC 控制 PCF8574，间接控制蜂鸣器开关。

2.1.24 SD 卡接口

阿波罗 STM32F767 开发板板载了一个 SD 卡（大卡）接口，其原理图如图 2.1.24.1 所示：

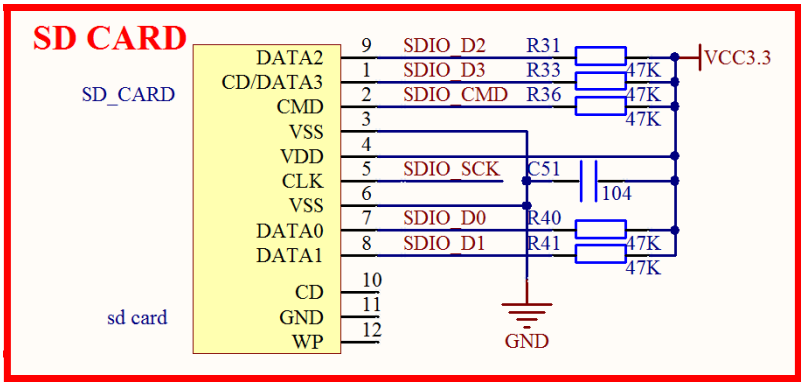


图 2.1.24.1 SD 卡/以太网接口

图中 SD_CARD 为 SD 卡接口，该接口在开发板的底面，这也是阿波罗 STM32F767 开发板底面唯一的元器件。

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

2.1.25 ATK 模块接口

阿波罗 STM32F767 开发板板载了 ATK 模块接口，其原理图如图 2.1.25.1 所示：

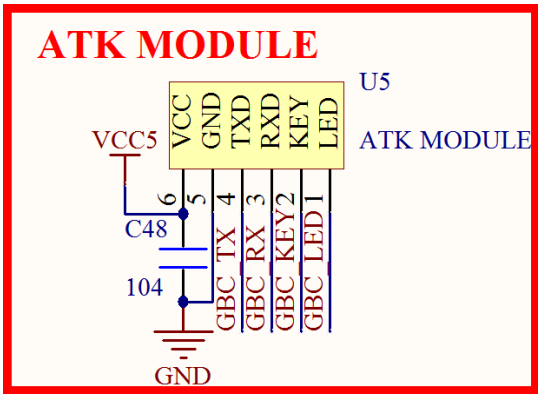


图 2.1.25.1 ATK 模块接口

如图所示，U5 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、WIFI 模块和 RGB 彩灯模块等。有了这个接口，我们连接模块就非常简单，插上即可工作。

图中：GBC_TX/GBC_RX 可通过 P9 排针，选择接入 PB11/PB10（即串口 3），详见 2.1.10 节。而 GBC_KEY 和 GBC_LED 则分别连接在 MCU 的 PI11 和 PA4 上面。**特别注意：**GBC_LED 和 STM_DAC 共用 PA4，GBC_KEY 和 NRF_IRQ 共用 PI11，在使用的时候，注意分时复用。

2.1.26 多功能端口

阿波罗 STM32F767 开发板板载的多功能端口，是由 P1 和 P11 构成的一个 6PIN 端口，其

原理图如图 2.1.26.1 所示：

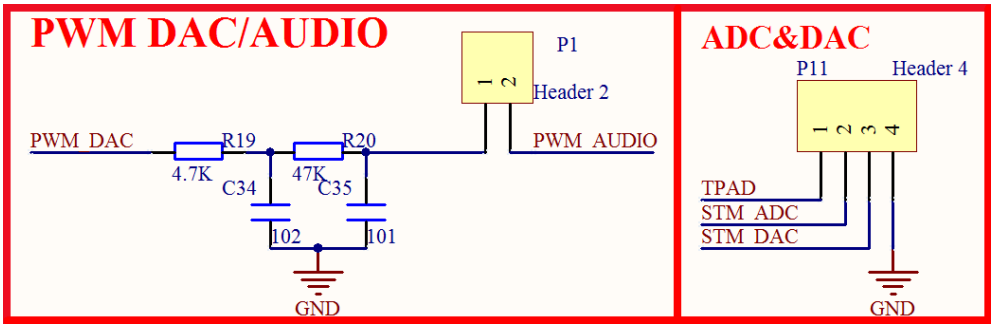


图 2.1.26.1 多功能端口

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

首先介绍右侧的 P11，其中 TPAD 为电容触摸按键信号，连接在电容触摸按键上。STM_ADC 和 STM_DAC 则分别连接在 PA5 和 PA4 上，用于 ADC 采集或 DAC 输出。当需要电容触摸按键的时候，我们通过跳线帽短接 TPAD 和 STM_ADC，就可以实现电容触摸按键（利用定时器的输入捕获）。STM_DAC 信号则既可以用作 DAC 输出，也可以用作 ADC 输入，因为 STM32 的该管脚同时具有这两个复用功能。**特别注意：**STM_DAC 与 ATK-MODULE 接口的 GBC_LED 共用 PA4，所以他们不可以同时使用，但是可以分时复用。

我们再来看看 P1，PWM_DAC 连接在 MCU 的 PA3，是定时器 2/5 的通道 4 输出，后面跟一个二阶 RC 滤波电路，其截止频率为 33.8Khz。经过这个滤波电路，MCU 输出的方波就变为直流信号了。PWM_AUDIO 是一个音频输入通道，它连接到 WM8978 的 AUX 输入，可通过配置 WM8978，输出到耳机/扬声器。**特别注意：**PWM_DAC 和 USART2_RX 共用 PA3，所以 PWM_DAC 和串口 2 的接收，不可以同时使用，不过，可以分时复用。

单独介绍完了 P11 和 P1 我们再来看看他们组合在一起的多功能端口，如图 2.1.26.2 所示：

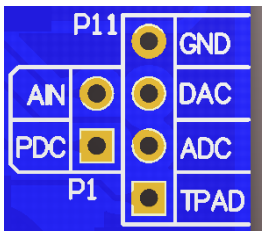


图 2.1.26.2 组合后的多功能端口

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

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

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

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

2.1.27 光纤输入接口

阿波罗 STM32F767 开发板底板板载了一个光纤输入接口，其原理图如图 2.1.27.1 所示：

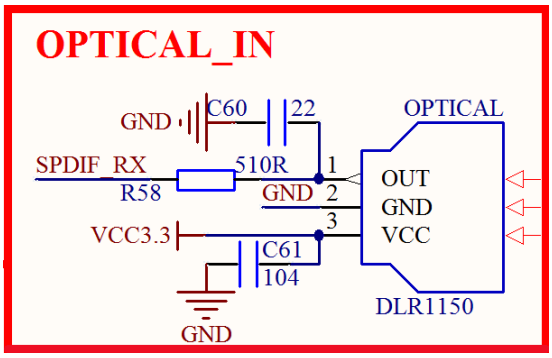


图 2.1.27.1 光纤输入接口

图中，光纤输入采用的是 DLR1150，输出信号经过 SPIDIF_RX 传输给 MCU，SPIDIF_RX 连接在 MCU 的 PG12 上面。**特别注意：**SPIDIF_RX 和 NRF_CE 共用 PG12，所以光纤接口和 WIRELESS 接口，不可以同时使用，不过，可以分时复用。

2.1.28 以太网接口（RJ45）

阿波罗 STM32F767 开发板板载了一个以太网接口(RJ45)，其原理图如图 2.1.28.1 所示：

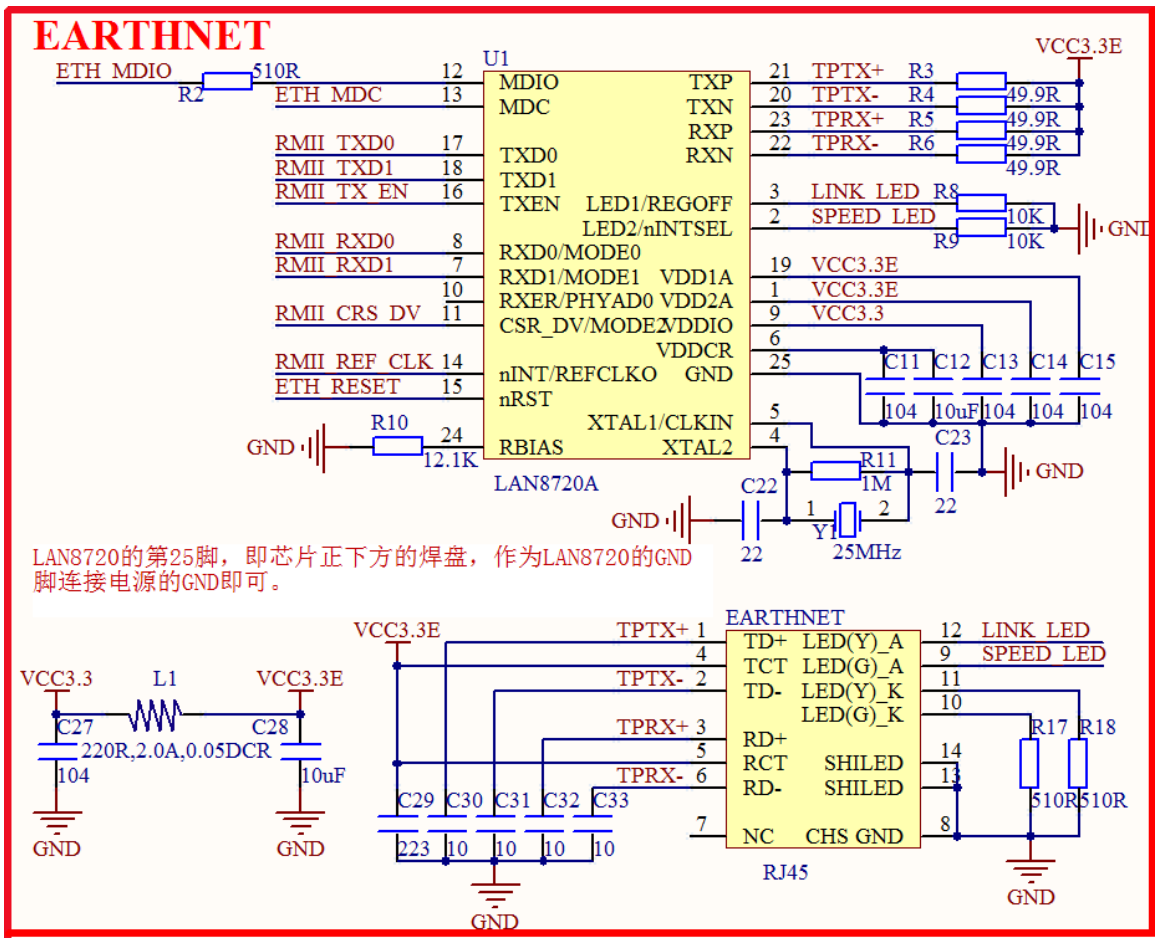


图 2.1. 28.1 以太网接口电路

STM32F767 内部自带网络 MAC 控制器，所以只需要外加一个 PHY 芯片，即可实现网络通信功能。这里我们选择的是 LAN8720A 这颗芯片作为 STM32F767 的 PHY 芯片，该芯片采用 RMII 接口与 STM32F767 通信，占用 IO 较少，且支持 auto mdix（即可自动识别交叉/直连网线）功能。板载一个自带网络变压器的 RJ45 头（HR91105A），一起组成一个 10M/100M 自适应网卡。

图中：ETH_MDIO/ETH_MDC/RMII_TXD0/RMII_TXD1/RMII_TX_EN/RMII_RXD0/RMII_RXD1/RMII_CRS_DV/RMII_REF_CLK 分别接在 MCU 的：PA2/PC1/PG13/PG14/PB11/PC4/PC5/PA7/PA1 上，ETH_RESET 则连接在 PCF8574（IIC IO 扩展芯片）的 P7 引脚上（有三极管反向）。**特别注意：**网络部分 ETH_MDIO 与 USART2_TX 共用 PA2，ETH_TX_EN 和 USART3_RX 共用 PB11，所以网络和串口 2 的发送以及串口 3 的接收，不可以同时使用，但是可以分时复用。

2.1.29 I2S 音频编解码器

阿波罗 STM32F767 开发板板载 WM8978 高性能音频编解码芯片，其原理图如图 2.1.29.1 所示：

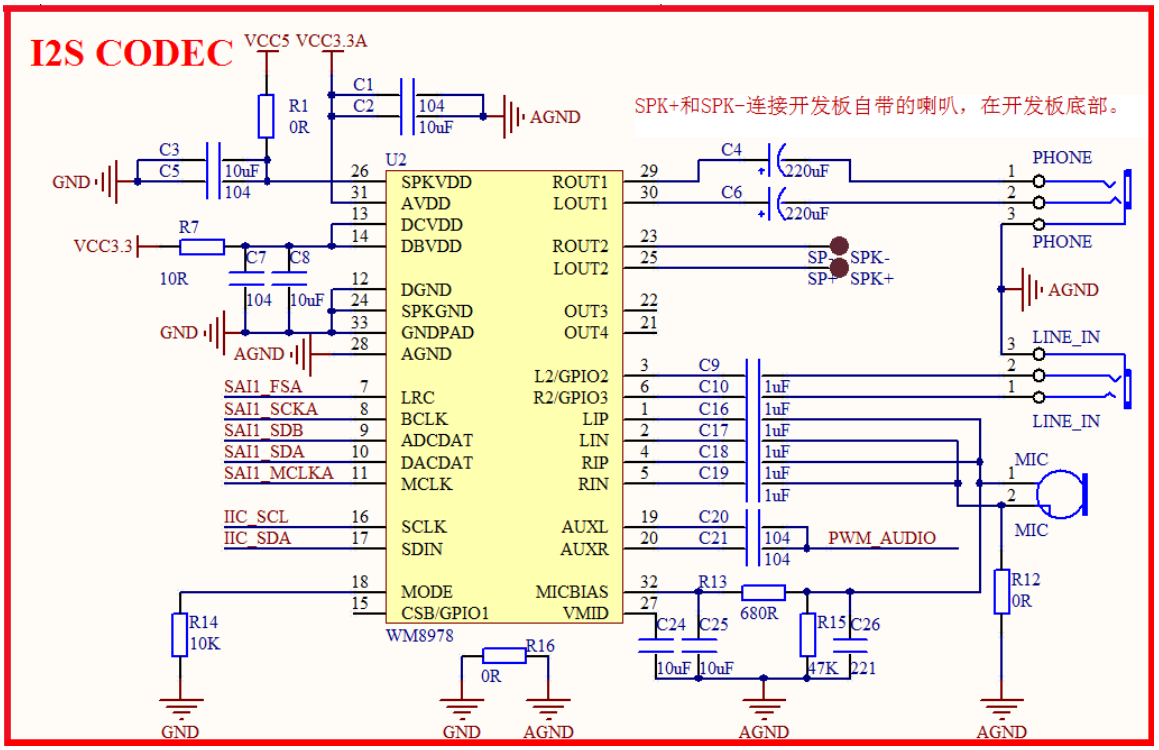


图 2.1.29.1 I2S 音频编解码芯片

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

图中，SPK-和 SPK+连接了一个板载的 8Ω 2W 小喇叭（在开发板背面）。MIC 是板载的咪头，可用于录音机实验，实现录音。PHONE 是 3.5mm 耳机输出接口，可以用来插耳机。LINE_IN 则是线路输入接口，可以用来外接线路输入，实现立体声录音。

该芯片采用 I2S 与 MCU 的 SAI 接口连接（SAI 支持 I2S），图中：SAI1_FSA/SAI1_SCKA

SAI1_SDB/SAI1_SDA/SAI1_MCLKA 分别接在 MCU 的：PE4/PE5/PE3/PE6/PE2 上。IIC_SCL 和 IIC_SDA 是与 AP3216C 等共用一个 IIC 接口。

2.1.30 电源

阿波罗 STM32F767 开发板板载的电源供电部分，其原理图如图 2.1.30.1 所示：

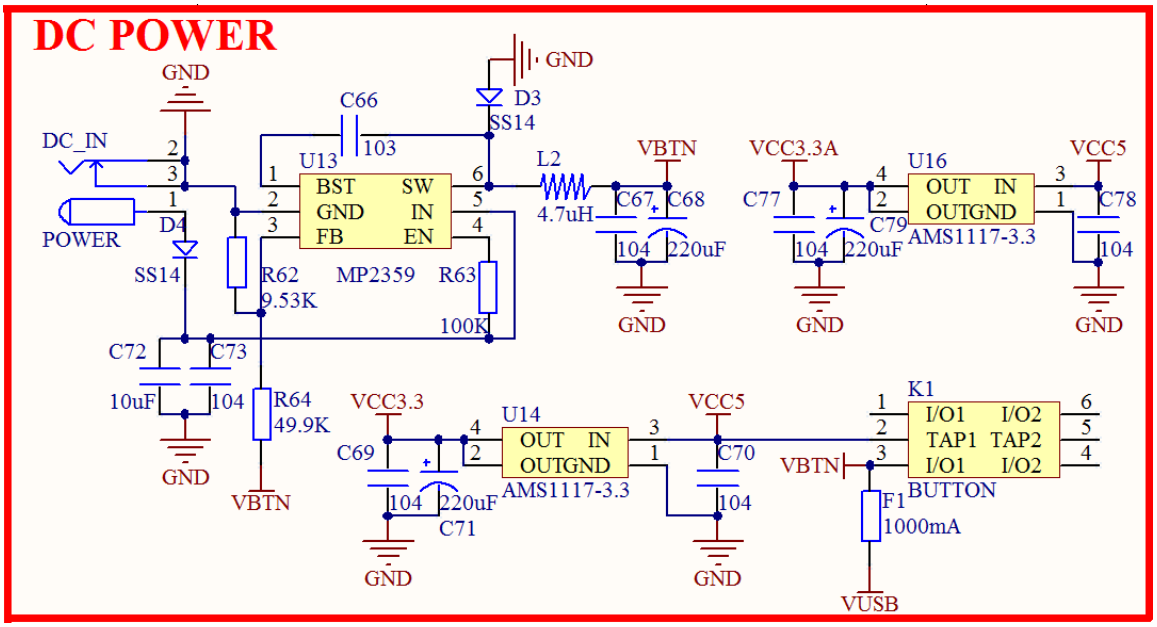


图 2.1. 30.1 电源

图中，总共有 3 个稳压芯片：U13/U14/U16，DC_IN 用于外部直流电源输入，经过 U13 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。U14 和 U16 均为 3.3V 稳压芯片，给开发板提供 3.3V 电源，其中 U14 输出的 3.3V 给数字部分用，U16 输出的 3.3V 给模拟部分（WM8978）使用，分开供电，以得到最佳音质。

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

2.1.31 电源输入输出接口

阿波罗 STM32F767 开发板板载了两组简单电源输入输出接口，其原理图如图 2.1.31.1 所示：

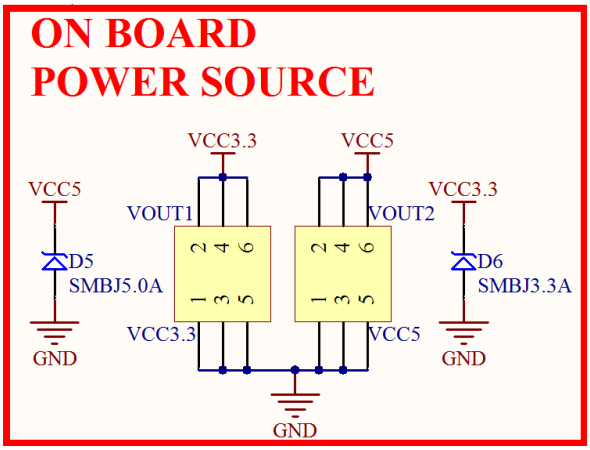


图 2.1.31.1 电源

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。

2.1.32 USB 串口

阿波罗 STM32F767 开发板板载了一个 USB 串口，其原理图如图 2.1.32.1 所示：

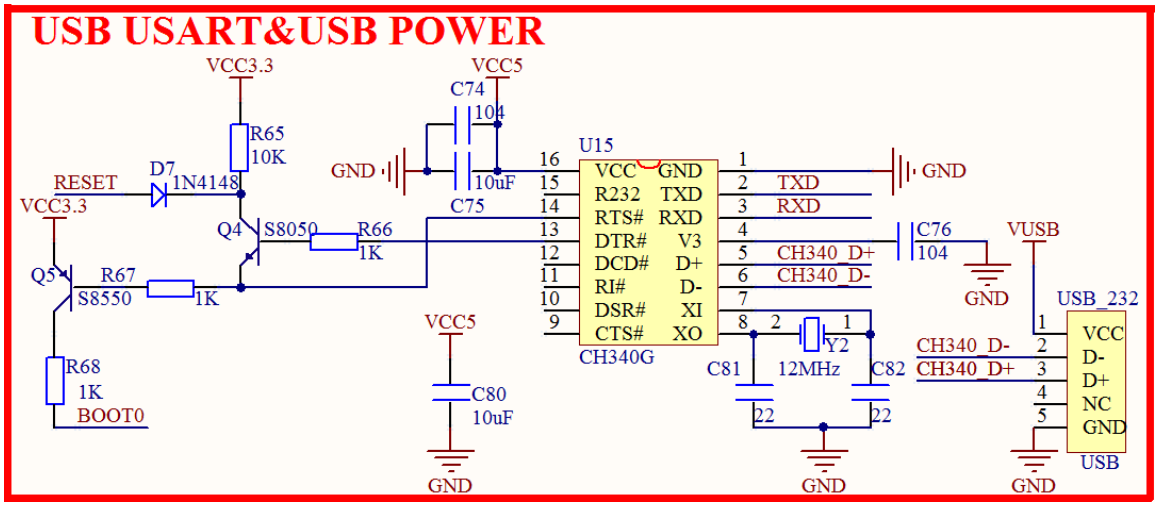


图 2.1.32.1 USB 串口

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 Q4 和 Q5 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 Q5 导通了，BOOT0 被拉高，即实现设置 BOOT0 为 1，同时 Q4 也会导通，STM32F767 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 STM32F767 的复位引脚，由于 Q4 不再导通，变为高电平，STM32F767 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 STM32F767，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 STM32F767 核心板原理图详解

2.2.1 MCU

阿波罗 STM32 开发板配套的 STM32F767 核心板，采用 STM32F767IGT6 作为 MCU，该芯片采用六级流水线，自带指令和数据 Cache、集成 JPEG 编解码器、集成双精度硬件浮点计算

单元（DPFPU）和 DSP 指令，并具有 512KB SRAM、1024KB FLASH、13 个 16 位定时器、2 个 32 位定时器、2 个 DMA 控制器（共 16 个通道）、6 个 SPI、1 个 QSPI 接口、3 个全双工 I2S、2 个 SAI、4 个 IIC、8 个串口、2 个 USB（支持 HOST /SLAVE）、3 个 CAN、3 个 12 位 ADC、2 个 12 位 DAC、1 个 SPDIF RX 接口、1 个 RTC（带日历功能）、2 个 SDMMC 接口、1 个 FMC 接口、1 个 TFTLCD 控制器（LTDC）、1 个 10/100M 以太网 MAC 控制器、1 个摄像头接口、1 个硬件随机数生成器、以及 140 个通用 IO 口等，芯片主频高达 216Mhz，轻松应对各种应用。

MCU 部分的原理图如图 2.2.1.1 和图 2.2.1.2（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 2.2.1.1 MCU 部分原理图-A

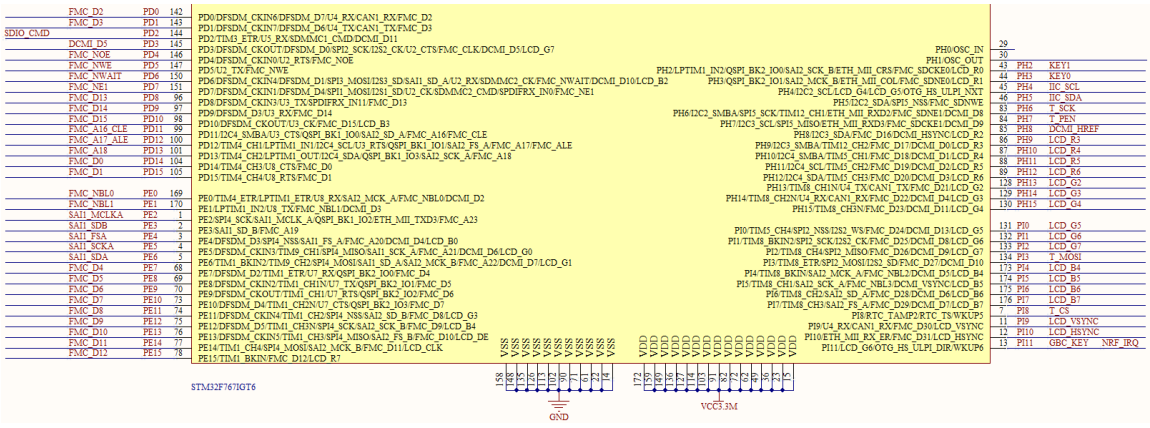


图 2.2.1.2 MCU 部分原理图-B

上图中 U1 为我们的主芯片：STM32F767IGT6。

这里主要讲解以下 4 个地方（有部分原理图未贴出，请参考光盘完整原理图查看）：

1，后备区域供电脚 VBAT 脚的供电采用 CR1220 纽扣电池（在底板上）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，原理图中的 R8 和 R9 用隔离 MCU 部分和外部的电源，这样的设计主要是考虑了后期维护，如果 3.3V 电源短路，可以断开这两个电阻，来确定是 MCU 部分短路，还是外部短路，

有助于生产和维修。当然大家在自己的设计上，这两个电阻是完全可以去掉的。

3，图中 VREF+是 MCU AD/DA 的参考电压，引出到底板。R2 默认不焊接，这样 VREF+由底板提供（底板的 P5 排针）。另外，VREF+还具有控制核心板 LED 总开关的功能，我们将在后续介绍。

4，PDR_ON 引脚，用于复位控制等，一般接 VCC 即可。在我们核心板上，默认通过 R5 电阻连接到 VCC3.3V。

2.2.2 底板接口

STM32F767 核心板采用 2 个 2*30 的 3710M（公座）板对板连接器来同底板连接(在核心板底面)，接插非常方便，核心板上面的底板接口原理图如图 2.2.2.1 所示：

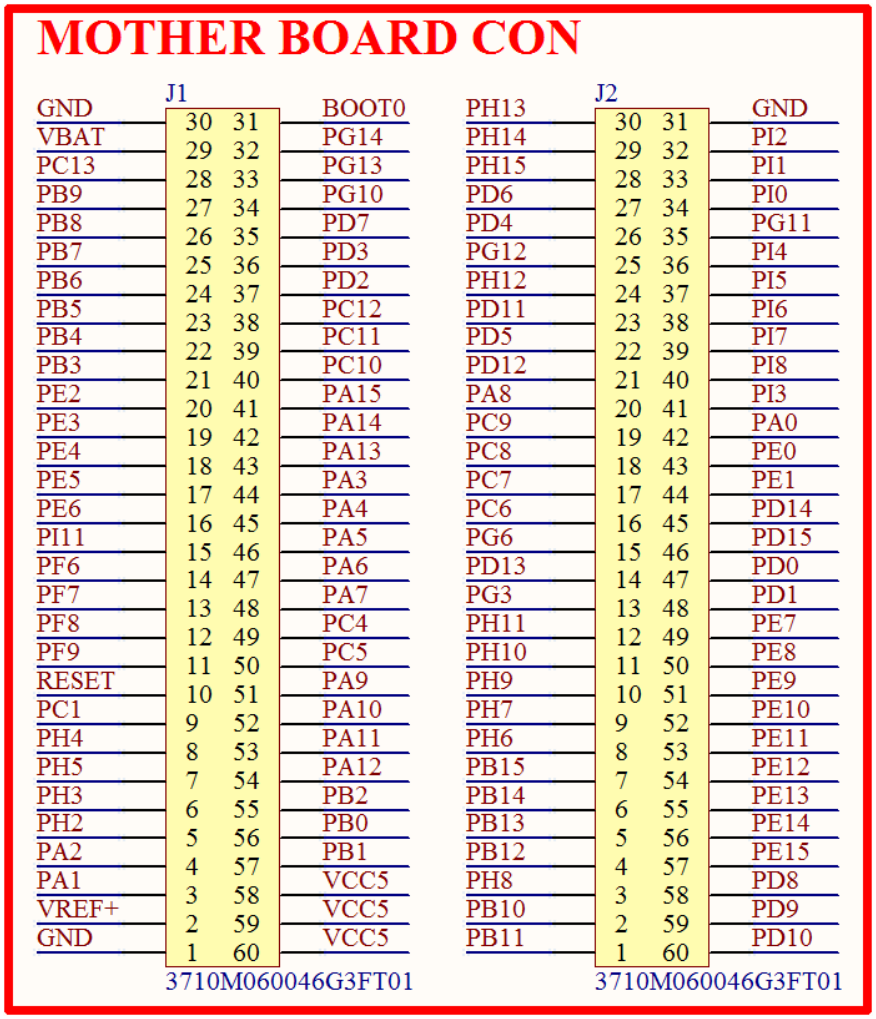


图 2.2.2.1 底板接口

图中，J1 和 J2 是 2 个 2*30 的板对板公座（3710M），和底板的接插非常方便，方便大家嵌入自己的项目中去。该接口总共引出 110 个 IO 口，另外，还有 3 根电源线、3 根地线、VBAT、RESET、BOOT0 和 VREF+。

2.2.3 SWD 调试接口

STM32F767 核心板板载了一个 SWD 调试接口，只需要最少 3 根线（GND、SWDCLK 和 SWDIO），即可实现代码调试和下载，SWD 接口原理图如图 2.2.3.1 所示：

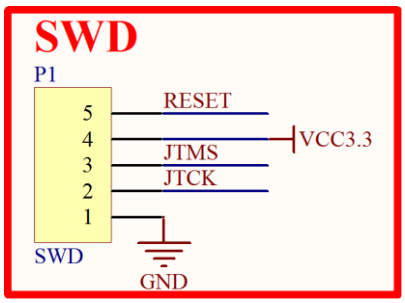
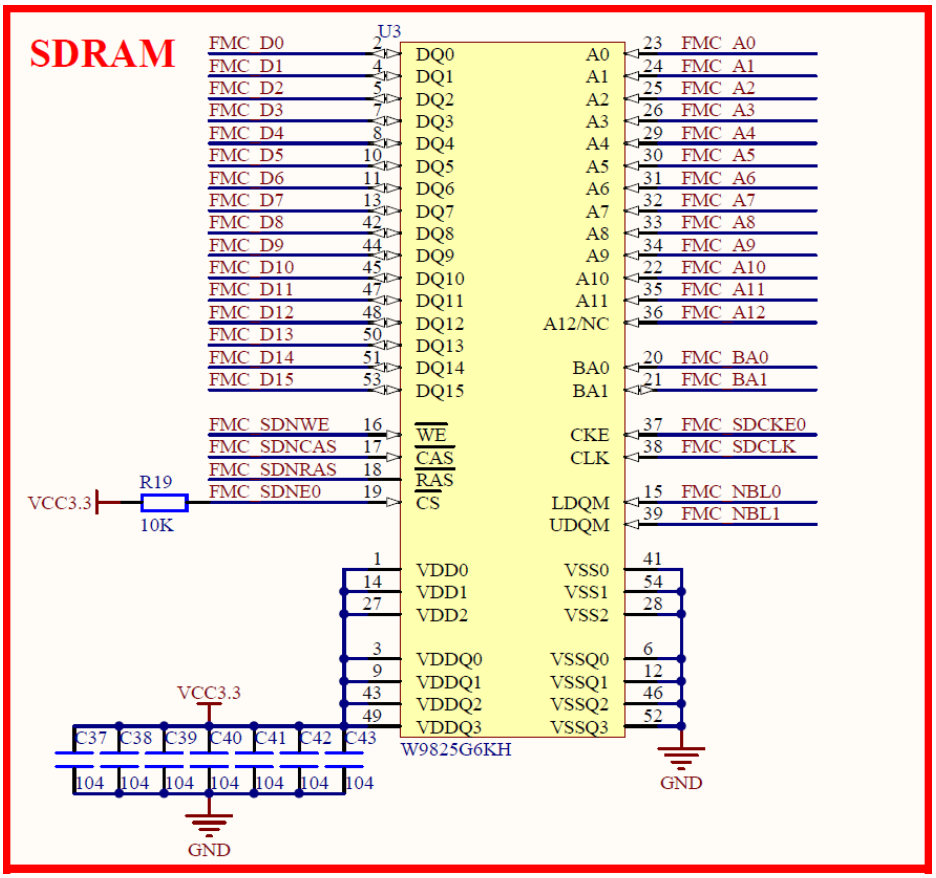


图 2.2.3.1 SWD 调试接口

图中 P1 就是一个 6P 的排针（默认没有焊接，需自行焊接），引出了 SWD 的信号线（JTMS=SWDIO，JTCK=SWDCLK）、RESET、电源和地。将这几个引脚正确连接 ST LINK、JLINK（**V9 或者以上版本**）或 ULINK 等仿真器对应的引脚，就可以对核心板进行仿真调试了。

2.2.4 SDRAM

STM32F767 核心板板载了 SDRAM，此部分电路如图 2.2.4.1 所示：

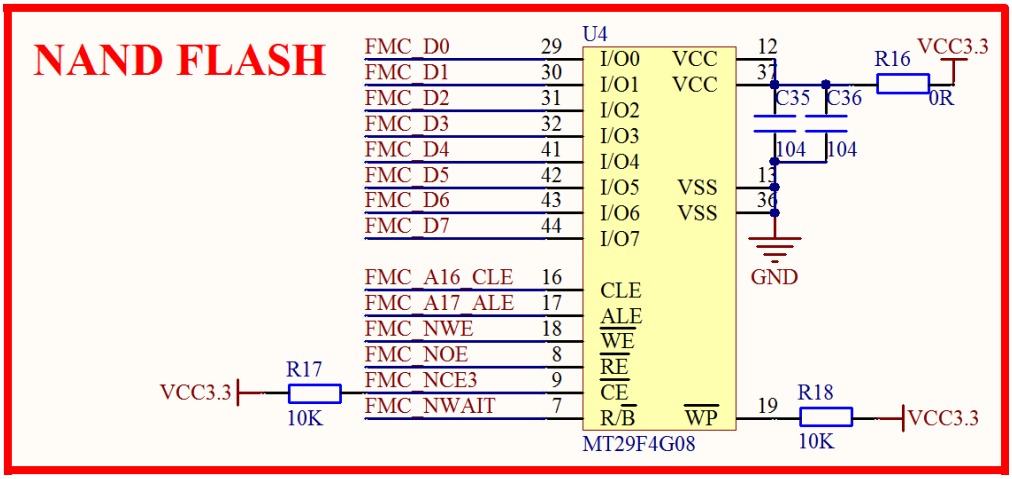


2.2.4.1 SDRAM

图中，U3 就是 SDRAM 芯片，型号为：W9825G6KH，容量为 32M 字节。该芯片挂在 STM32F767 的 FMC 接口上，有了这颗芯片，大大扩展了 STM32 的内存（本身只有 256KB），在各种大内存需求场合，ALIENTEK 这款 STM32F767 核心板，都可以从容面对。

2.2.5 NAND FLASH

STM32F767 核心板板载了 NAND FLASH，此部分电路如图 2.2.5.1 所示：

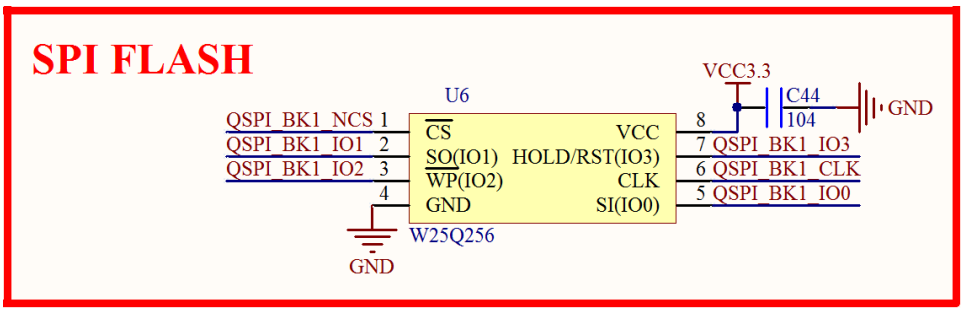


2.2.5.1 NAND FLASH

图中，U4 就是 NAND FLASH 芯片，型号为：MT29F4G08，容量为 512M 字节。该芯片同样是挂在 STM32F767 的 FMC 接口上，有了这颗芯片，大大扩展了 STM32 的存储空间，可以实现海量数据存储。另外，如果大家觉得 512M 不够用，还可以更换其他更大容量的 NAND FLASH 芯片，硬件上，接口是完全兼容的。

2.2.6 SPI FLASH

STM32F767 核心板板载了 SPI FLASH，此部分电路如图 2.2.6.1 所示：



2.2.6.1 SPI FLASH

图中，U6 就是 SPI FLASH 芯片，支持 QSPI 接口，型号为：W25Q256，容量为 32M 字节，可以用来存放字库、启动文件等重要的数据。这里我们采用 STM32F7 的 QSPI 接口连接，使得访问速度大大提高。

2.2.7 EEPROM

STM32F767 核心板板载了 EEPROM，此部分电路如图 2.2.7.1 所示：

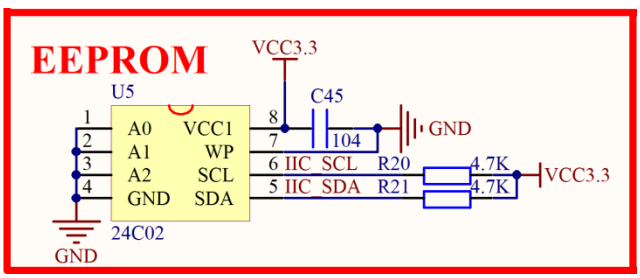


图 2.2.7.1 EEPROM

图中，U5 就是 EEPROM 芯片，型号为：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 PH4 上，IIC_SDA 接在 MCU 的 PH5 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 非常不好用！请慎用。IIC_SCL/IIC_SDA 总线上总共挂了 5 个器件：24C02、AP3216C、PCF8574、MPU9250 和 WM8978（后面四个在之前已经介绍过）。

2.2.8 RGB LCD 接口

STM32F767 核心板板载了 RGB LCD 接口，此部分电路如图 2.2.8.1 所示：

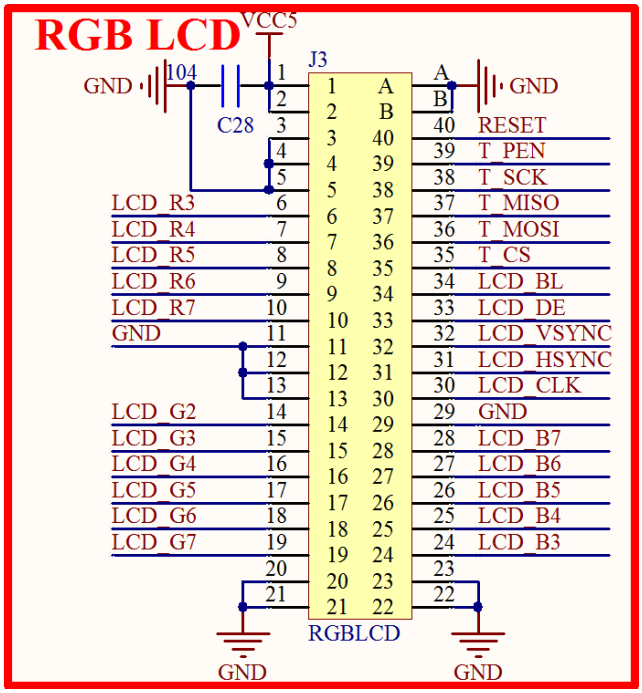


图 2.2.8.1 RGB LCD 接口

图中，J3（RGBLCD）就是 RGB LCD 接口，采用 RGB565 数据格式，并支持触摸屏（支持电阻屏和电容屏）。该接口仅支持 RGB 接口的液晶（不支持 MCU 接口的液晶），目前 ALIENTEK 的 RGB 接口 LCD 模块有：4.3 寸（ID:4342，480*272）和 7 寸(ID:7084，800*480 和 ID:7016，1024*600)等尺寸可选。

图中的 T_MISO/T_MOSI/T_PEN/T_CS/T_SCK 连接在 MCU 的 PG3/PI3/PH7/PI8/PH6 上，用于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 PB5 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。**特别注意：**该接口底板上的 LCD（MCU 屏）模块接口，共用触摸屏和背光信号线，所以他们不能同时都使用触摸屏！！！

2.2.9 串口

STM32F767 核心板板载了一个 TTL 串口，引出了 MCU 的串口 1（USART1），此部分电路如图 2.2.9.1 所示：

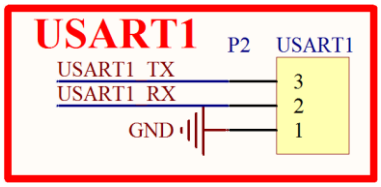


图 2.2.9.1 串口

图中，P2 就是核心板引出的串口 1（USART1）接口，通过 3 个排针引出（默认没有焊接，需自行焊接），USART1_TX 和 USART1_RX 分别连接在 MCU 的 PA9 和 PA10 上面。注意：它和底板上的 P4 是连接在一起的。

2.2.10 Micro USB 接口

STM32F767 核心板板载了一个 Micro USB 接口，此部分电路如图 2.2.10.1 所示：

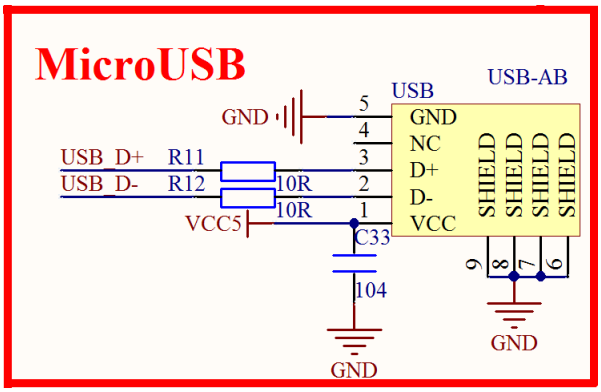


图 2.2.10.1 Micro USB 接口

图中，USB-AB 就是一个 Micro USB 座，可以用来连接电脑，做从机（SLAVE），也可以通过外接 USB OTG 线连接 U 盘/USB 鼠标/USB 键盘和 USB 手柄等，做主机（HOST）。USB_D-和 USB_D+分别连接在 MCU 的 PA11 和 PA12 上面，它们和底板上的 P10 是连接在一起的。

同时，该接口也可以用于给核心板提供电源。

2.2.11 按键

STM32F767 核心板板载了一个功能按键：WK_UP，此部分电路如图 2.2.11.1 所示：

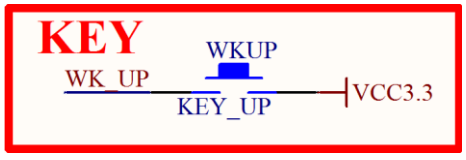


图 2.2.11.1 功能按键

图中，KEY_UP 按键，连接在 MCU 的 PA0，高电平有效，可以用来实现按键输入，也可以用作 MCU 的唤醒（WKUP）。注意：它和底板上的 KEY_UP 是连接在一起的。

2.2.12 LED

STM32F767 核心板板载了 2 个 LED，此部分电路如图 2.2.12.1 所示：

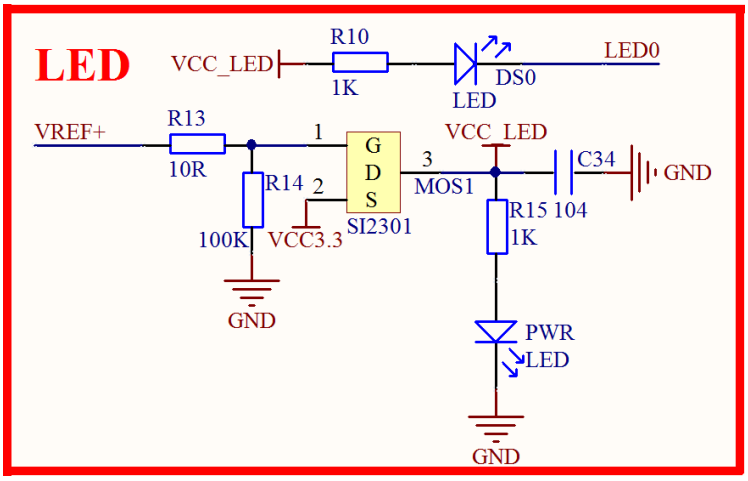


图 2.2.12.1 2 个 LED

图中，PWR 是电源指示灯（蓝色），用于指示核心板的供电状态；DS0 是功能指示灯（红色），LED0 连接在 MCU 的 PB1，可以用于指示程序运行状态。这两个 LED 的工作状态，都受 VREF+的控制，**当 VREF+悬空的时候（核心板单独工作或拔了底板 P5 的跳线帽），PWR 和 DS0 都正常工作，当 VREF+接 3.3V 的时候（底板的 P5 跳线帽短接），PWR 和 DS0 都关闭**。如果不想 PWR 和 DS0 受 VREF+控制（一直工作），去掉 R13 即可。

2.2.13 电源

STM32F767 核心板板载的电源供电部分，原理图如图：2.2.13.1 所示：

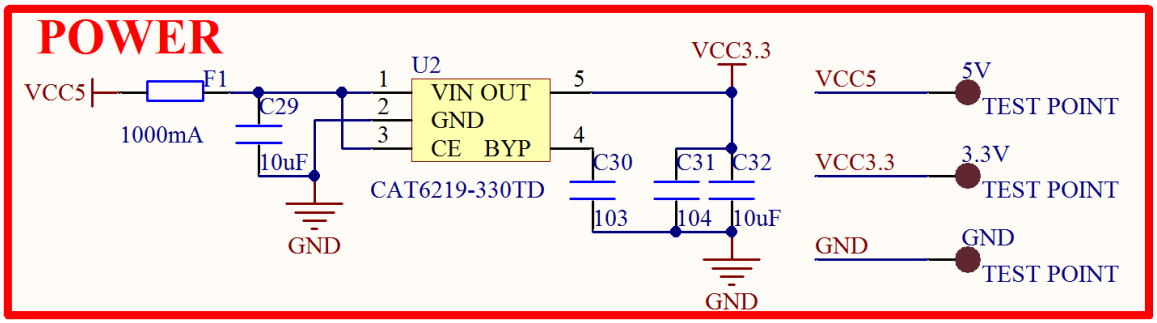


图 2.2.13.1 电源

图中，U2 是稳压芯片，将 5V 转换为 3.3V，整个核心板的 3.3V 电源，都来自此芯片。F1 是自恢复保险丝，可以起到过流保护的作用。右侧的 5V、3.3V 和 GND 等三个 TEST_POINT 是在核心板流出的 3 个焊盘，可以给开发板供电，或者从开发板取电。

2.3 开发板使用注意事项

为了让大家更好的使用 ALIENTEK 阿波罗 STM32F767 开发板，我们在这里总结该开发板使用的时候尤其要注意的一些问题，希望大家在使用的时候多多注意，以减少不必要的问题。

1， 如果 USB_232 接口连接了电脑，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 3~5 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， 核心板上的 PWR 和 DS0 两个 LED 是受 VREF+控制的，所以，当底板上的 P5 跳线帽连接时（默认就是连接的，短接 VREF+和 3.3V），核心板的 PWR 和 DS0 是一直关闭的（不会亮）！如果想要核心板的 PWR 和 DS0 受控，拔了 P5 的跳线帽即可！

3， 1 个 USB 供电最多 500mA，且由于导线电阻存在，供到开发板的电压，一般都不会有 5V，如果使用了很多大负载外设，比如 4.3 寸屏、网络、摄像头模块等，那么可能引起 USB 供电不够，所以如果是使用 4.3 屏的朋友，或者同时用到多个模块的时候，**建议大家使用一个独立电源供电**。 如果没有独立电源，建议可以同时插 2 个 USB 口，并插上 JTAG，这样供电可以更足一些。

4， JTAG 接口有几个信号（JTDI/JTDO/JTRST）被 OLED/CAMERA 接口占用了，所以在调试这个接口的时候，请大家选择 SWD 模式，其实**最好就是一直用 SWD 模式。**

5， 当你想使用某个 IO 口用作其他用处的时候，请先看看开发板的原理图，该 IO 口是否有连接在开发板的某个外设上，如果有，该外设的这个信号是否会对你的使用造成干扰，先确定无干扰，再使用这个 IO。比如 PA8 就不太适合做输入 IO，因为 REMOTE_IN 连接在这个 IO 上面，可能会对输入检测造成影响。

6， 开发板上的跳线帽比较多，大家在使用某个功能的时候，要先查查这个是否需要设置跳线帽，以免浪费时间。

7， 当液晶显示白屏的时候，请先检查液晶模块是否插好（拔下来重新插试试），如果还不行，可以通过串口看看 LCD ID 是否正常，再做进一步的分析。

8， 开发板的 USB SLAVE 和 USB HOST 共用同一个 USB 口，所以，他们不可以同时使用。使用的时候多加注意。

至此，本手册的实验平台（ALIENTEK 阿波罗 STM32F767 开发板）的硬件部分就介绍完了，了解了整个硬件对我们后面的学习会有很大帮助，有助于理解后面的代码，在编写软件的时候，可以事半功倍，希望大家细读！另外 ALIENTEK 开发板的其他资料及教程更新，都可以在技术论坛 www.openedv.com 下载到，大家可以经常去这个论坛获取更新的信息。

2.3 STM32F767 学习方法

STM32F7 系列是目前最强大的 ARM Cortex M7 处理器，由于其强大的功能，可替代 DSP 等特性，具有非常广泛的应用前景。初学者可能会认为 STM32F767 很难学，以前可能只学过 51，或者甚至连 51 都没学过的，一看到 STM32F767 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 STM32F767，还是非常简单的，这里我们总结学习 STM32F767 的几个要点：

1，一款实用的开发板。

这个是实验的基础，有个开发板在手，什么东西都可以直观的看到。但开发板不宜多，多了的话连自己都不知道该学哪个了，觉得这个也还可以，那个也不错，那就这个学半天，那个学半天，结果学个四不像。倒不如从一而终，学完一个在学另外一个。

2，三本参考资料，即《STM32F7 中文参考手册》、《STM32F7xx 参考手册》和《STM32F7 编程手册》。

《STM32F7 中文参考手册》和《STM32F7xx 参考手册》都是 STM32F7 系列的参考手册，前者是中文翻译版本，仅针对 STM32F74x/75x 系列，后者则是英文版本，针对 STM32F76x/77x 系列，这两本手册详细介绍了 STM32F7 的各种寄存器定义以及外设的使用说明等，是学习 STM32F767 的必备资料。而《STM327 编程手册》则是对《STM32F7 中文参考手册》的补充，很多关于 Cortex M7 内核的介绍（寄存器等），都可以在这个文档找到答案，该文档同样是 ST 的官方资料，专门针对 ST 的 Cortex M7 产品。结合这三本参考资料，就可以比较好的学习 STM32F7 了。

3，掌握方法，勤学善悟。

STM32F767 不是妖魔鬼怪，不要畏难，STM32F767 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《STM32F7 中文参考手册》图 12）。

任何单片机，必定是靠时钟驱动的，时钟就是单片机的动力，STM32F767 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。学习 STM32F767，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 STM32F767 的过程，只有进一步思考，才能更好的掌握 STM32F767，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 STM32F767 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 6W 多个主题，很多疑问已经有网友提过了，所以可以在论坛先搜索一下，很多时候，就可以直接找到答案了。论坛是一个分享交流的好地方，是一个可以让大家互相学习，互相提高的平台，所以有时间，可以多上去看看。

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

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 STM32F7 的开发软件：MDK5。通过该篇的学习，你将了解到：1、STM32CubeF7 和 HAL 库；2、如何在 MDK5 下新建基于 HAL 库的 STM32F7 工程；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 STM32F7 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 STM32F7 的开发流程，并能独立开始 STM32F7 的编程和学习。

本篇将分为如下 3 个章节：

2.1，软件入门；

2.2，STM32F7 基础知识入门

2.3，SYSTEM 文件介绍；

第三章 软件入门

本章将向大家介绍 MDK5 软件和 STM32CubeF7，通过本章的学习，我们最终将建立一个基于 HAL 库的的 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 简介与安装

3.2，STM32CubeF7 简介

3.3，新建基于 HAL 库的工程模板和工程结构讲解

3.4，程序下载与调试

3.5，MDK5 使用技巧；

3.1 MDK5 简介与安装

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：MDK5.21，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：

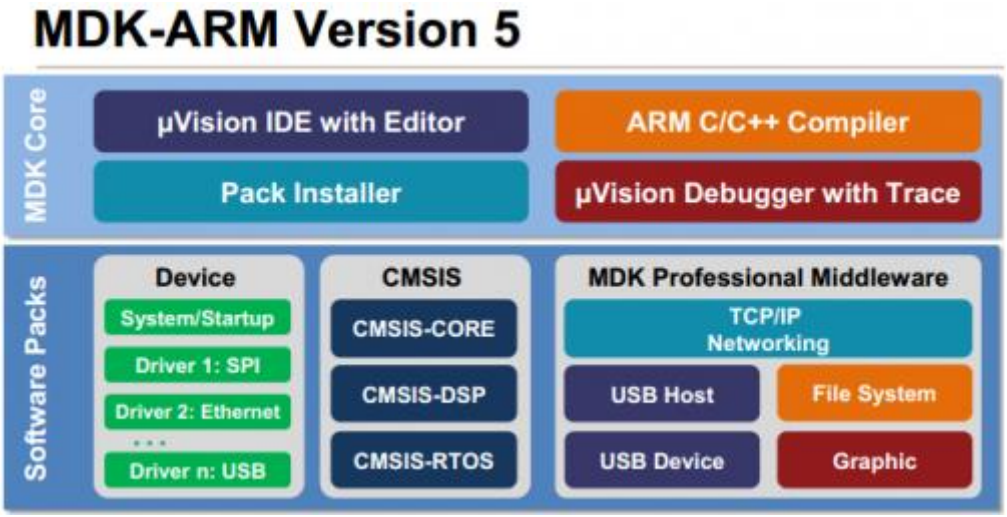


图 3.1.1 MDK5 组成

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。

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。具体安装步骤请参考光盘教程“**/1，ALIENTEK 阿波罗 STM32F7 开发板入门资料/MDK5.21 安装手册.pdf**”即可。

在 MDK5 安装完成后，要让 MDK5 支持 STM32F7 的开发，还要安装 STM32F7 的器件支持包：Keil.STM32F7xx_DFP.2.7.0.pack（STM32F7 系列的器件包）。这个包以及 MDK5.21 安装软件，我们都已经在开发板光盘提供了，跟安装软件在同一级目录。

3.2 STM32CubeF7 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 STM32F7 基础才能入门使用，所以我们安排在后面 4.8 小节给大家讲解。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 STM32F7 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)
{
  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 STM32CubeF7 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 STM32F7 的使用，所以我们主要讲解 STM32CubeF7 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 STM32CubeF7 包在我们开发板配套光盘有提供，目录为：**\8，STM32 参考资料\1，STM32CubeF7 固件包**。

接下来我们看看 STM32CubeF7 包目录结构，如下图 3.2.2.1 所示：

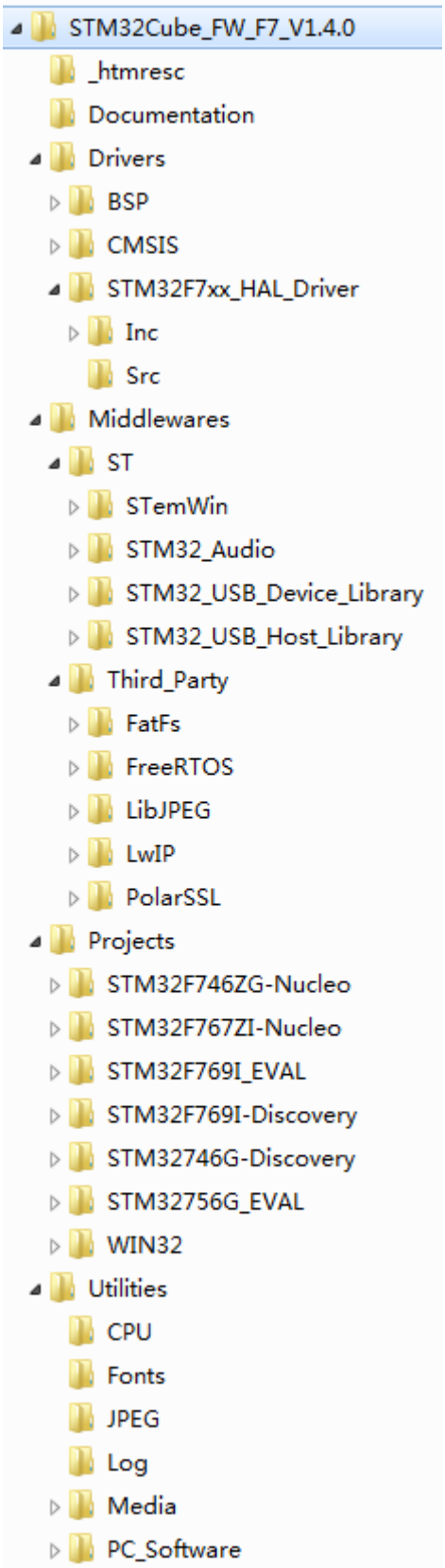


图 3.2.2.1 STM32CubeF7 包目录结构

对于 Documentation 文件夹，里面是一个 STM32CubeF7 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 STM32CubeF7 中几个关键的文件夹。

1）Drivers 文件夹。Drivers 文件夹包含 BSP，CMSIS 和 STM32F7xx_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。
	STM32F7xx_HAL_Driver 文件夹	这个文件夹非常重要，它包含了所有的 STM32F7xx 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。源文件名称基本遵循 stm32f7xx_hal_ppp.c 定义格式，头文件名称基本遵循 stm32f7xx_hal_ppp.h 定义格式。比如 gpio 相关的 API 的声明和定义在文件 stm32f7xx_hal_gpio.h 和 stm32f7xx_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_Audio 文件夹	
		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。

		LibJPEG 文件夹	基于 C 语言的 JPEG 图形解码支持包。
		LwIP 文件夹	LwIP 网络通信协议支持包。
		PolarSSL 文件夹	SSL/TLS 安全层解决方案支持包，基于开源的 PolarSSL。

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。比如我们要查看 STM32F767 相关工程，所以我们直接打开子文件夹 STM32F767ZI-Nucleo 即可。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。例如我们打开\Projects\STM32F767ZI-Nucleo 文件夹，内容如下图 3.2.2.4：

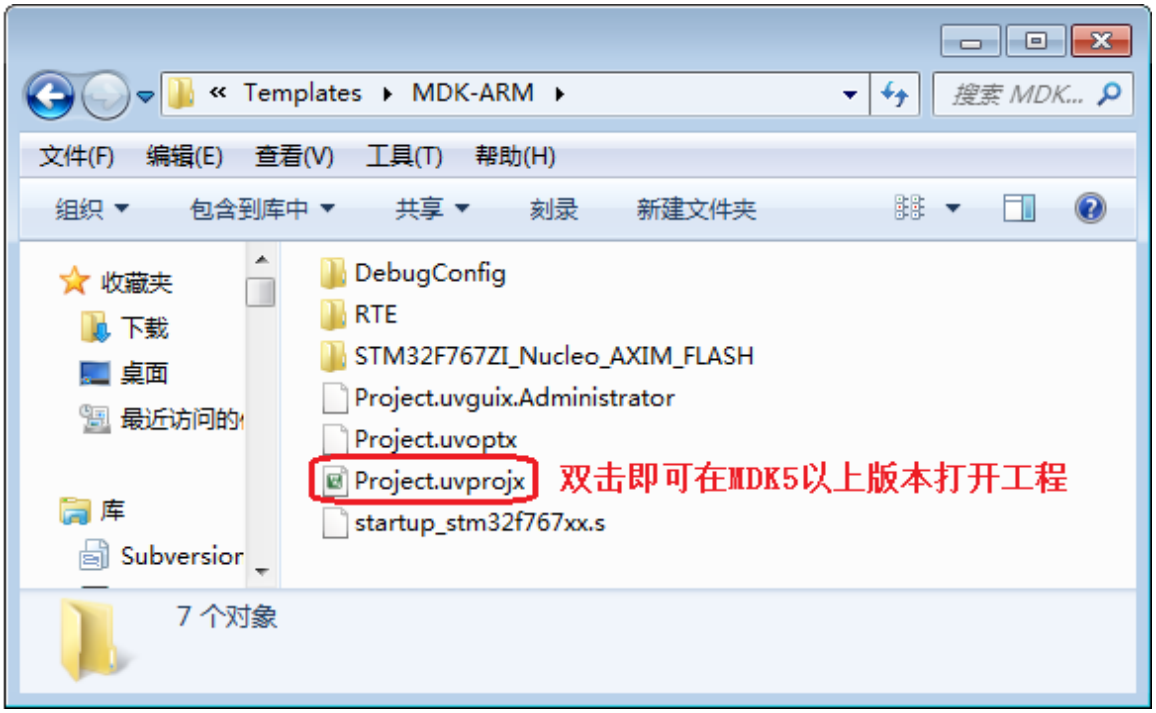


图 3.2.2.4 Templates 工程中 MDK-ARM 文件夹内容

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。

3.3 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 STM32F7xx 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。**实际上，我们可以使用 ST 官方的 STM32CubeMX 图形工具生成一个工程模板，这里之所以我们还要手把手教大家新建一个模板，是为了让大家对工程新建和运行过程有一个深入的理解，这样在日后的开发中遇到任何问题都可以得心应手的解决，STM32CubeMX 工具的使用我们在后面的 4.8 小节会详细讲解**。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_F7_V1.4.0 这是 ST 官网下载的 STM32CubeF7 包完

整版，我们光盘目录（压缩包）：

“\8，STM32 参考资料\1，STM32CubeF7 固件包”。

我们官方论坛开源电子网帖子 http://www.openedv.com/thread-80566-1-1.html 中也有下载。

2) MDK5.21 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\MDK5.21。

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\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**” 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：

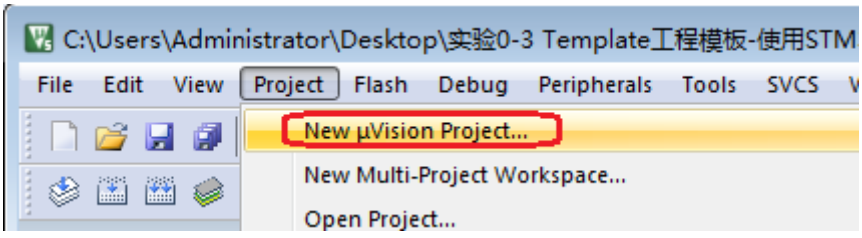


图 3. 3. 1. 2 新建工程

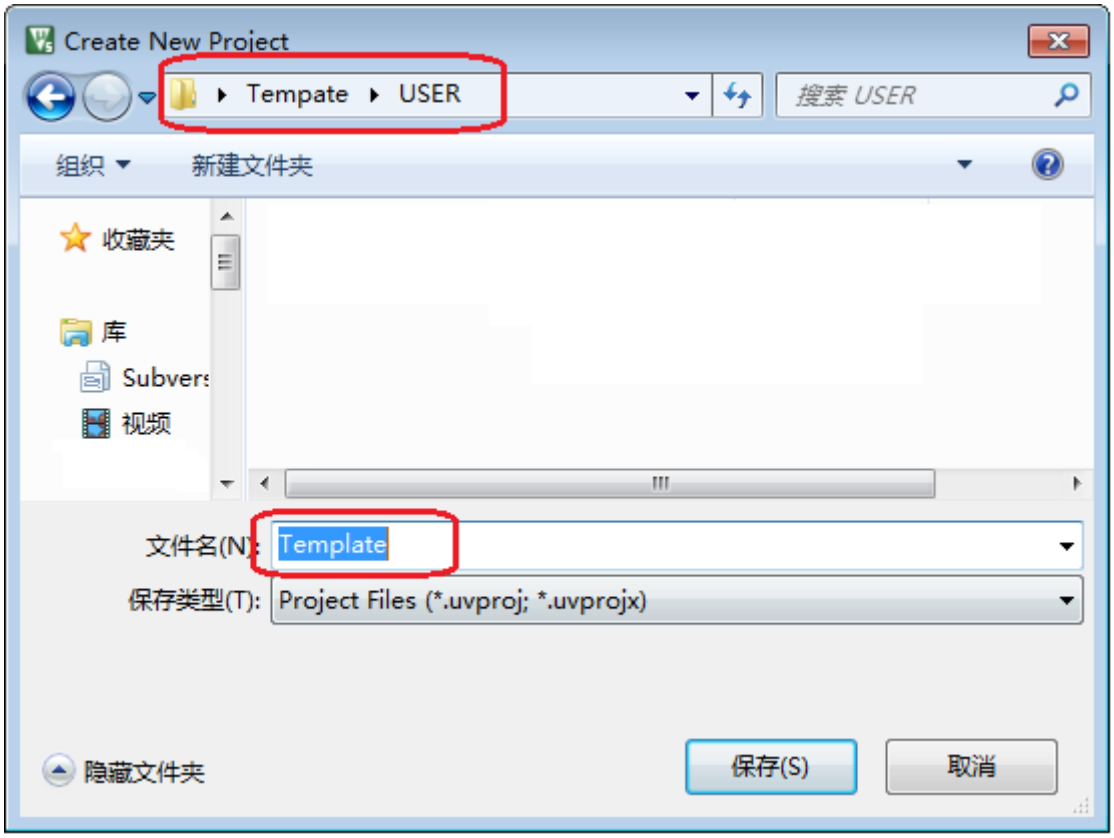


图 3.3.1.3 定义工程名称

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，大家根据自己使用的芯片型号依次选择即可。如果阿波罗 STM32F 开发板使用的是 STM32F767IGT 芯片，那么依次选择 STMicroelectronics→STM32F7 Series→STM32F767→STM32F767IG→STM23F767IGTx（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**）。

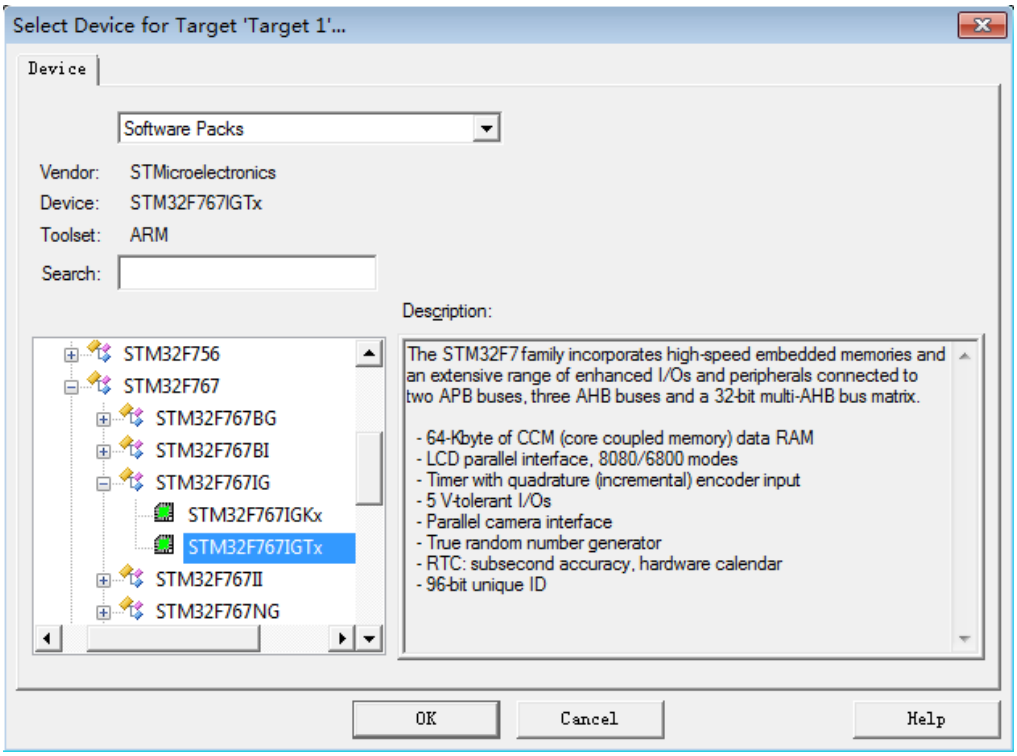


图 3.3.1.4 选择芯片型号

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

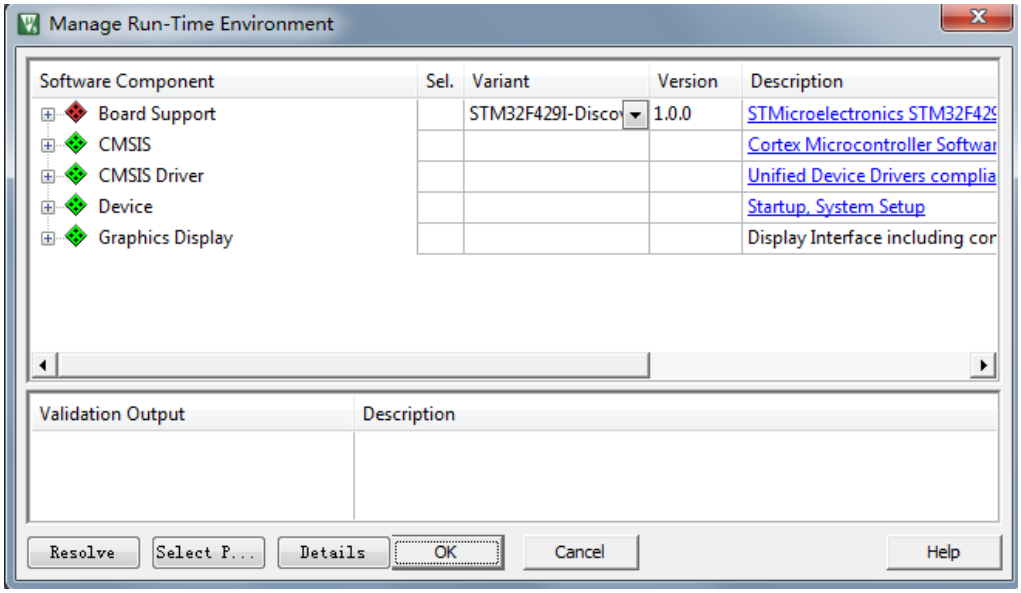


图 3.3.1.5 Manage Run-Time Environment 界面

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：

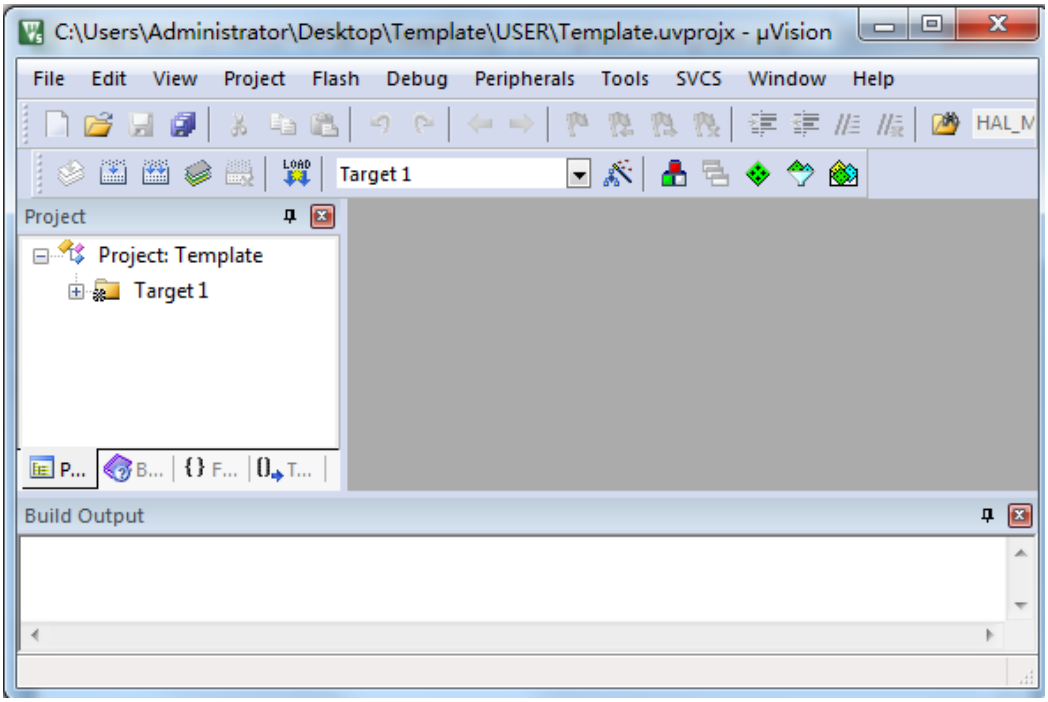


图 3.3.1.6 工程初步建立

3） 现在我们看看 USER 目录下面内容，如下图 3.3.1.7：

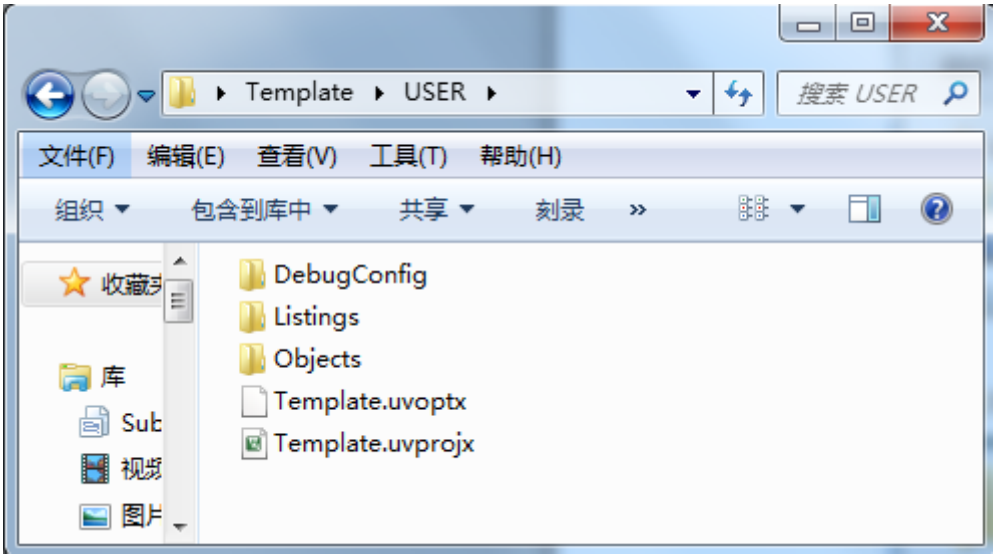


图 3.3.1.7 工程 USER 目录文件

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，MDK5.21 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 STM32CubeF7 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 STM32CubeF7 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 STM32CubeF7 包，定位到我们之前准备好的 HAL 库包的目录:\STM32Cube_FW_F7_V1.4.0\Drivers\STM32F7xx_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。

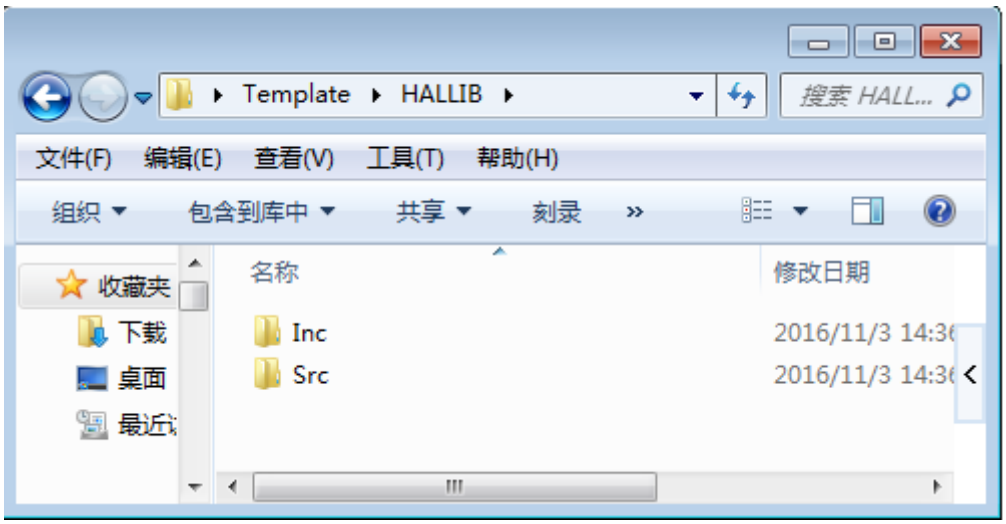


图 3.3.1.8 官方库源码文件夹

5） 接下来，我们要将 STM32CubeF7 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 STM32CubeF7 包，定位到目录\STM32Cube_FW_F7_V1.4.0\Drivers\CMSIS\Device\ST\STM32F7xx\Source\Templates\arm 下面，将文件 **startup_stm32f767xx.s** 复制到 CORE 目录下面。然后定位到目录\STM32Cube_FW_F7_V1.4.0\Drivers\CMSIS\Include，将里面的五个头文件：**cmsis_armcc.h，core_cm7.h，core_cmFunc.h，core_cmInstr.h ，core_cmSimd.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：

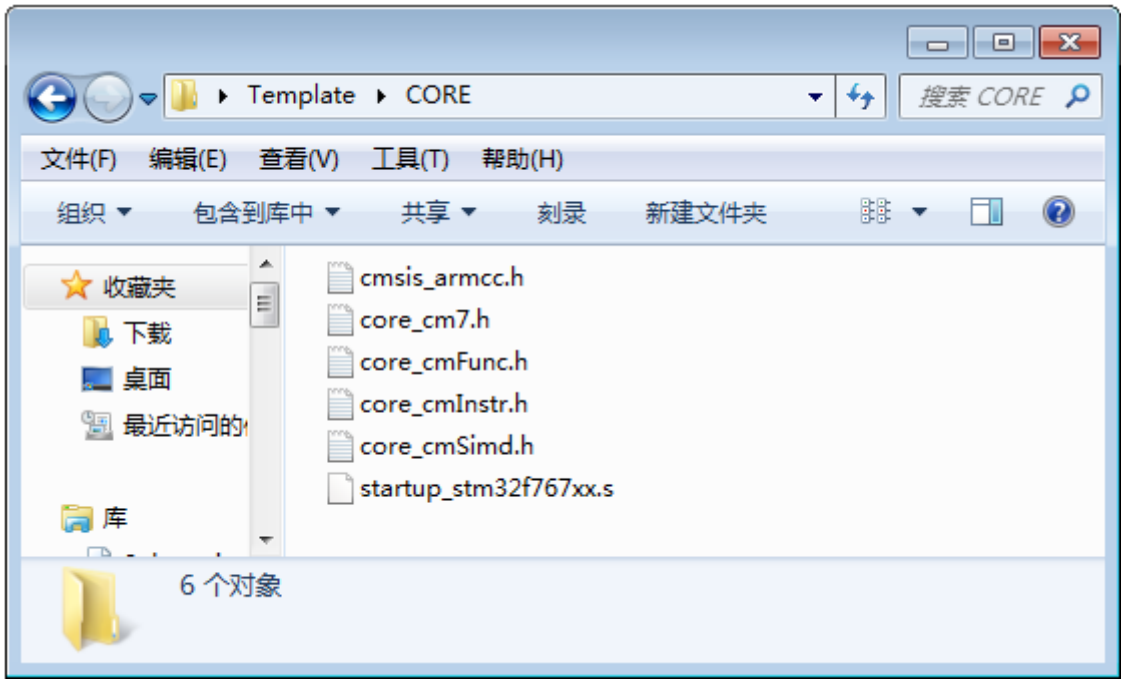


图 3.3.1.9 CORE 文件夹文件

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_F7_V1.4.0\Drivers\CMSIS\Device\ST\STM32F7xx\Include 将里面的 3 个文件 stm32f7xx.h，system_stm32f7xx.h 和 stm32f767xx.h 复制到 USER 目录之下。这三个头文件是 STM32F7 工程非常关键的头文件，前面我们介绍 STM32CubeF7 包的时候已经给大家介绍过。然后进入目录\STM32Cube_FW_F7_V1.4.0\Projects\STM32F767ZI-Nucleo\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。

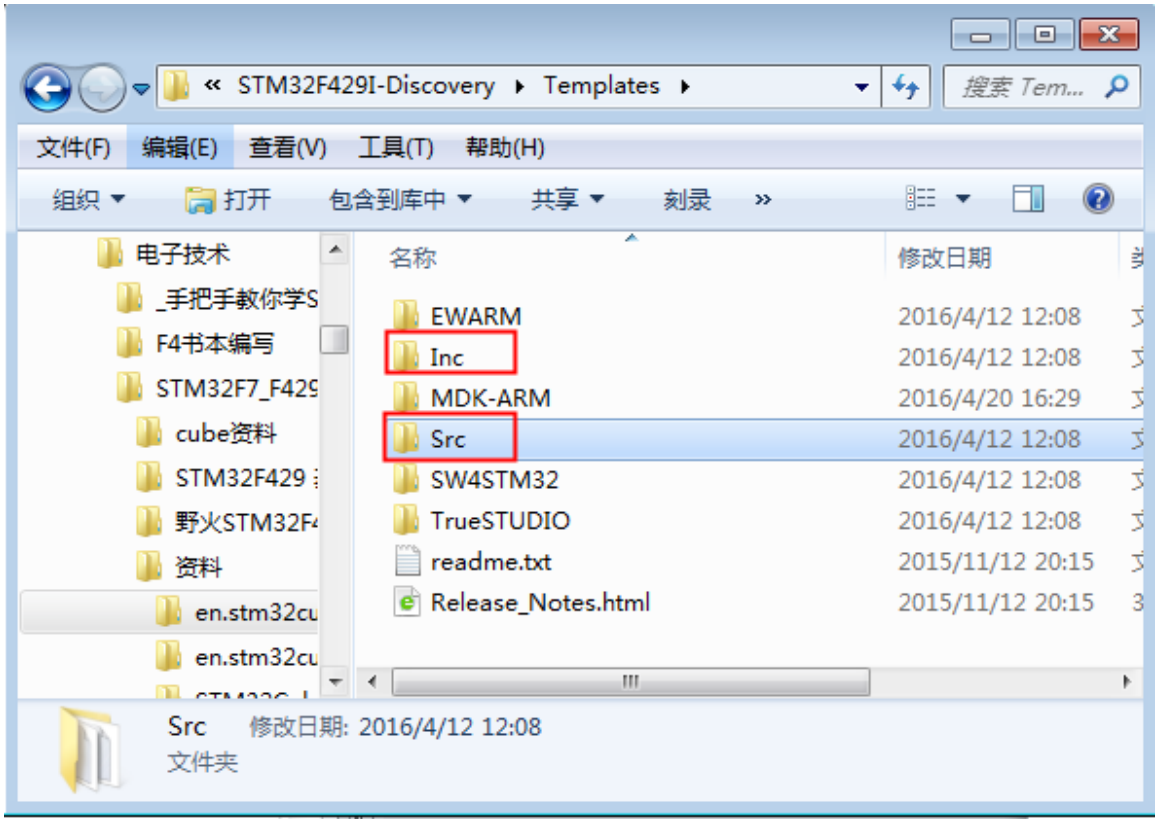


图 3.3.1.10 固件库包 Template 目录下面文件一览

首先我们打开 Inc 目录，将目录下面的 3 个头文件 stm32f7xx_it.h，stm32f7xx_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_stm32f7xx.c，stm32f7xx_it.c, stm32f7xx_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：

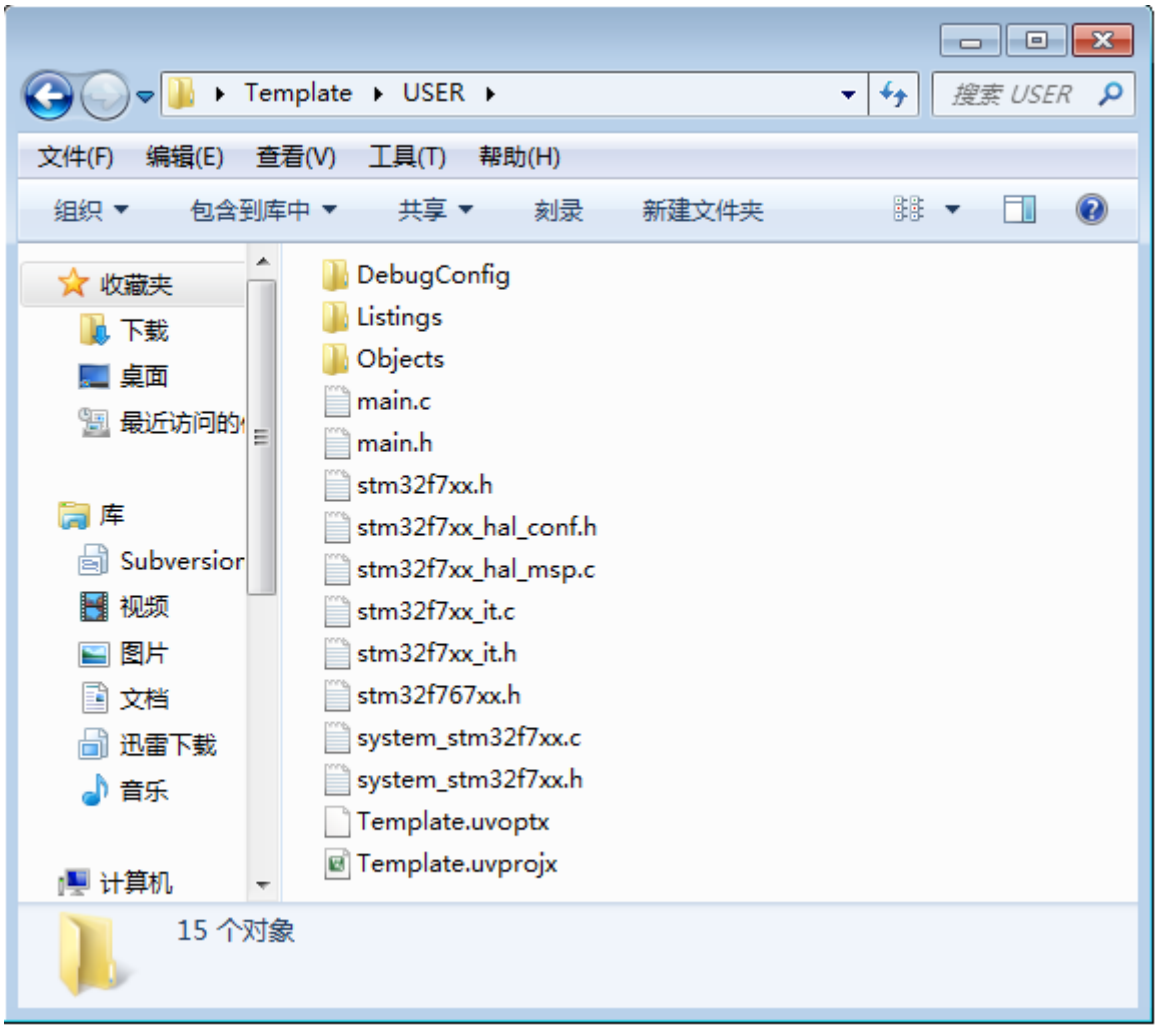


图 3.3.1.11 USER 目录文件浏览

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 STM32F767 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，这里我们新建的是库函数工程模板，所以大家从光盘程序源码目录之下的库函数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：

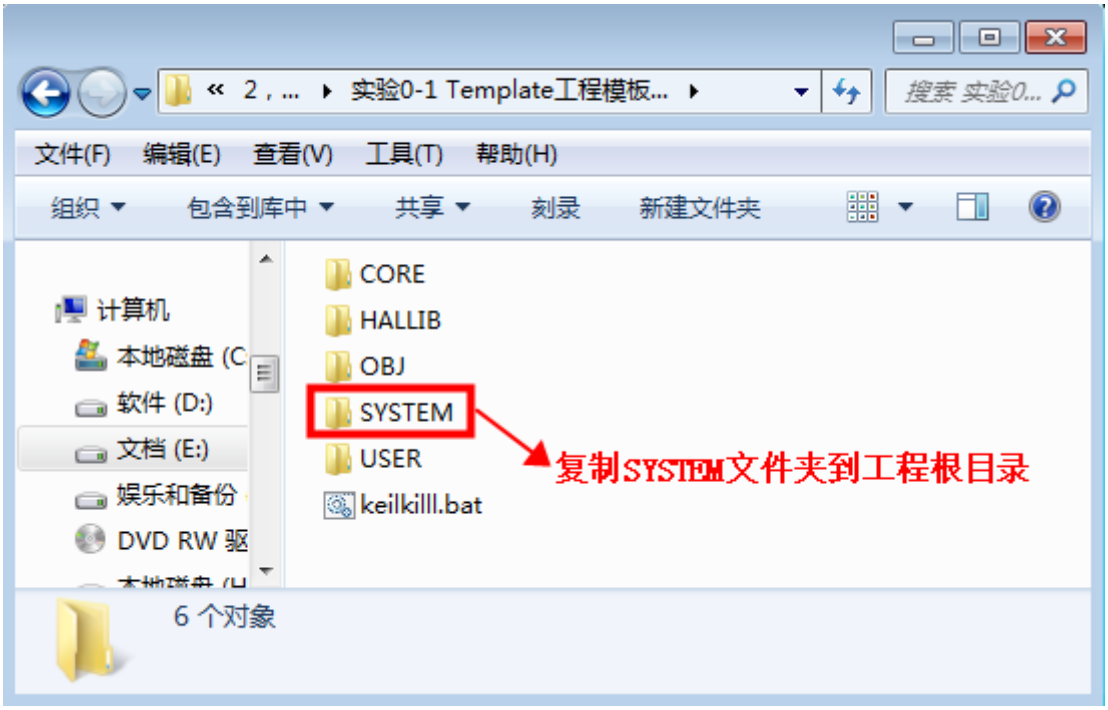


图 3.3.1.12 复制实验 0-1 的 SYSTEM 文件夹到工程根目录

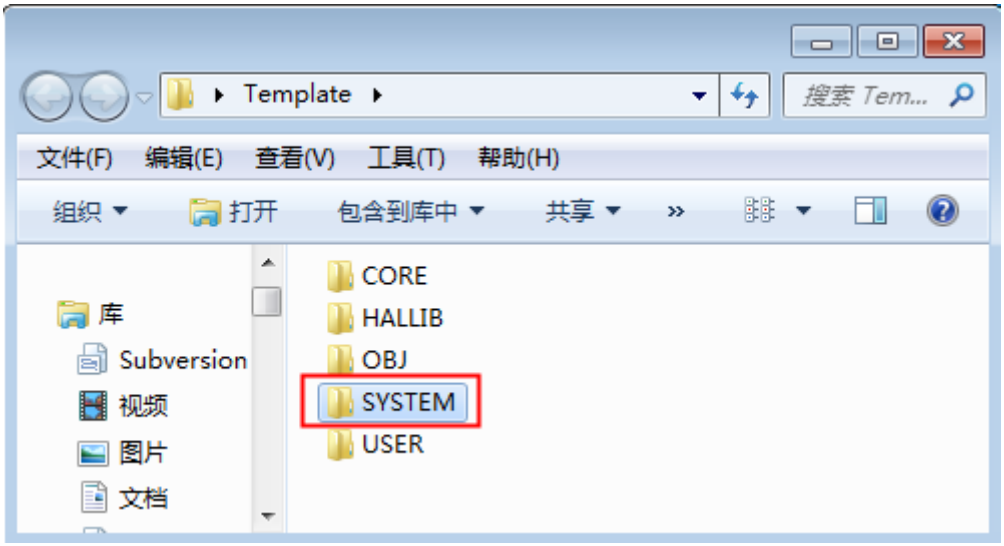


图 3.3.1.13 复制 SYSTEM 文件夹之后的 Template 根目录文件夹结构

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：

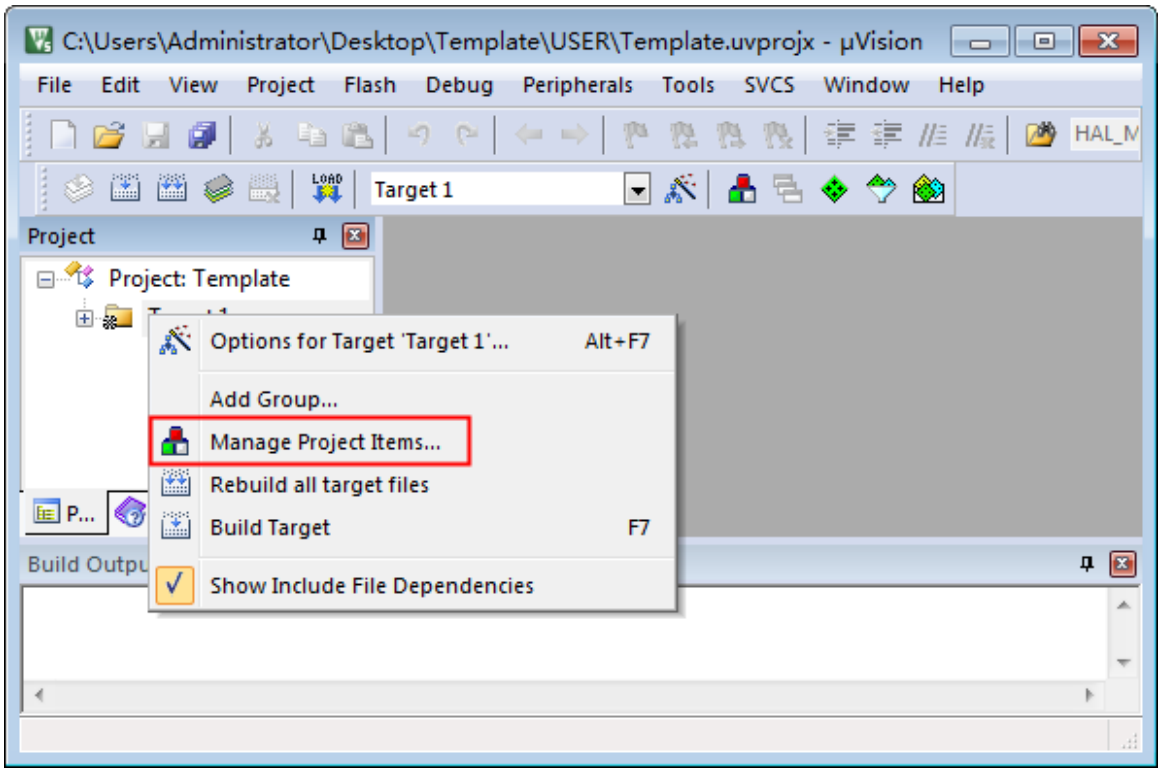


图 3.3.1.14 点击 Management Project Itmes

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:

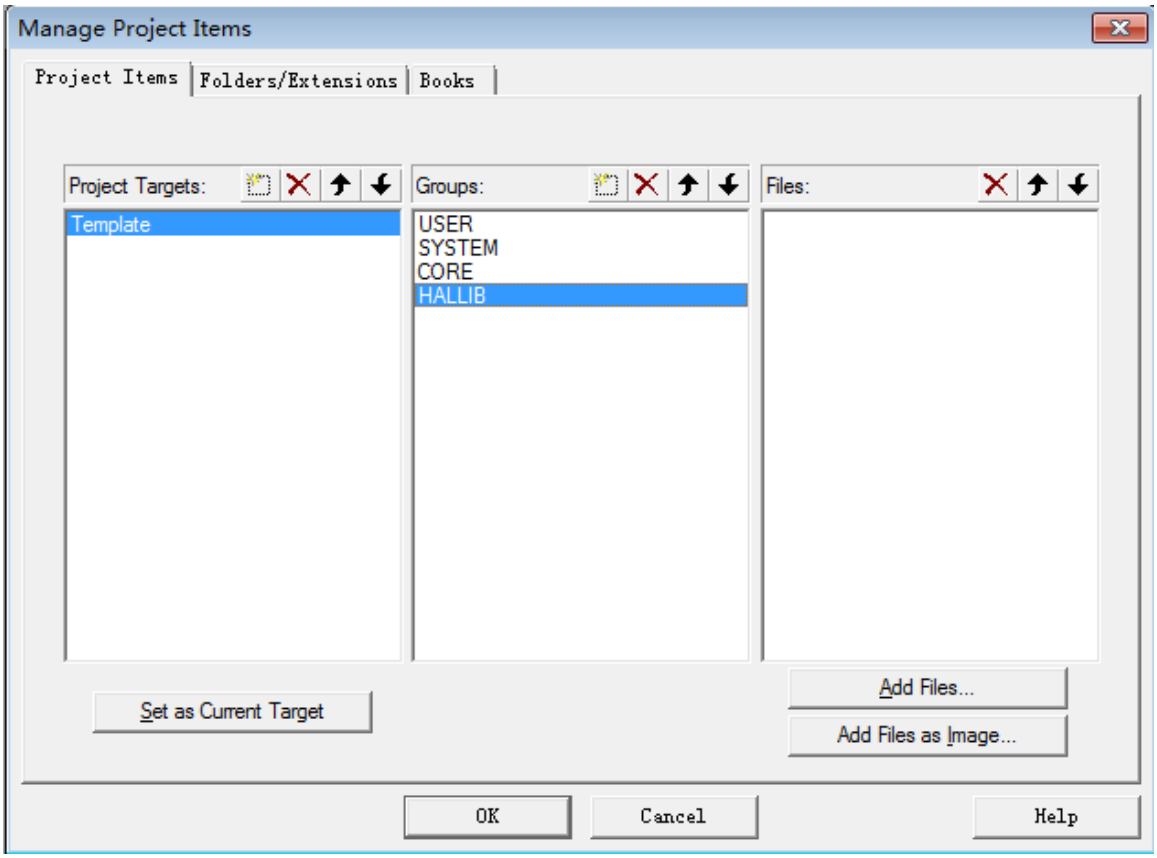


图 3.3.1.15 新建 GROUP

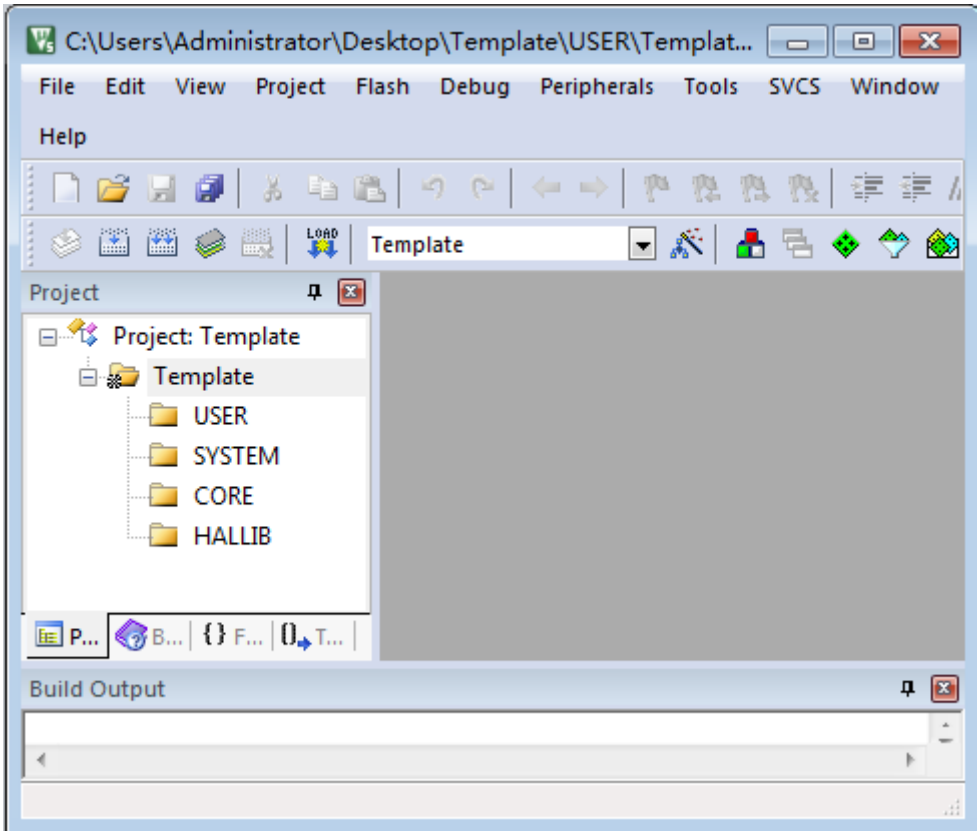


图 3.3.1.16 查看工程 Group 情况

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 stm32f7xx_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。

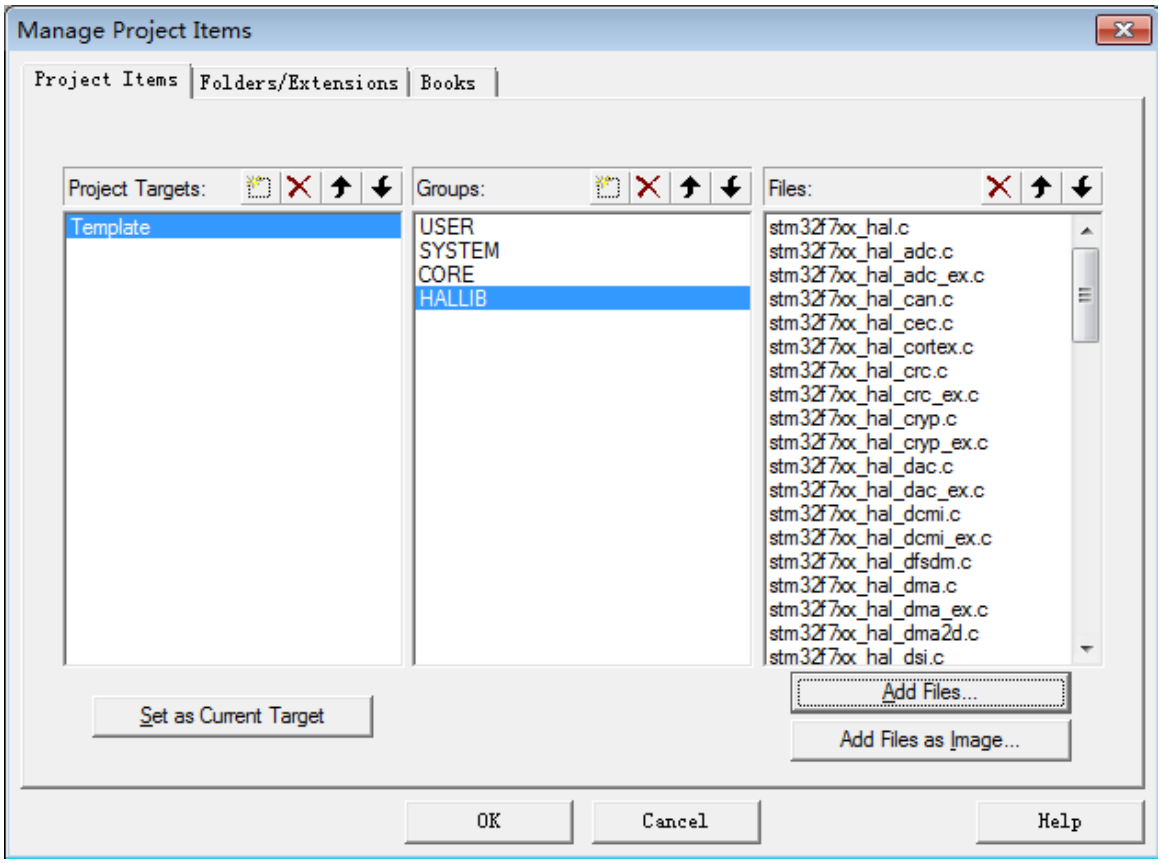


图 3.3.1.17 添加文件到 HALLIB 分组

这里有几个 template 文件不需要引入，例如 stm32f7xx_hal_msp_template.c，stm32f7xx_hal_timebase_rtc_alarm_template.c,stm32f7xx_hal_timebase_rtc_wakeup_template.c 和 stm32f7xx_hal_timebase_tim_template.c 四个文件不需要引入工程，这些文件我们在工程中可以参考，但是不需要引入工程，所以我们删除即可。删除某个引入文件方法如下图 3.3.1.18 所示：

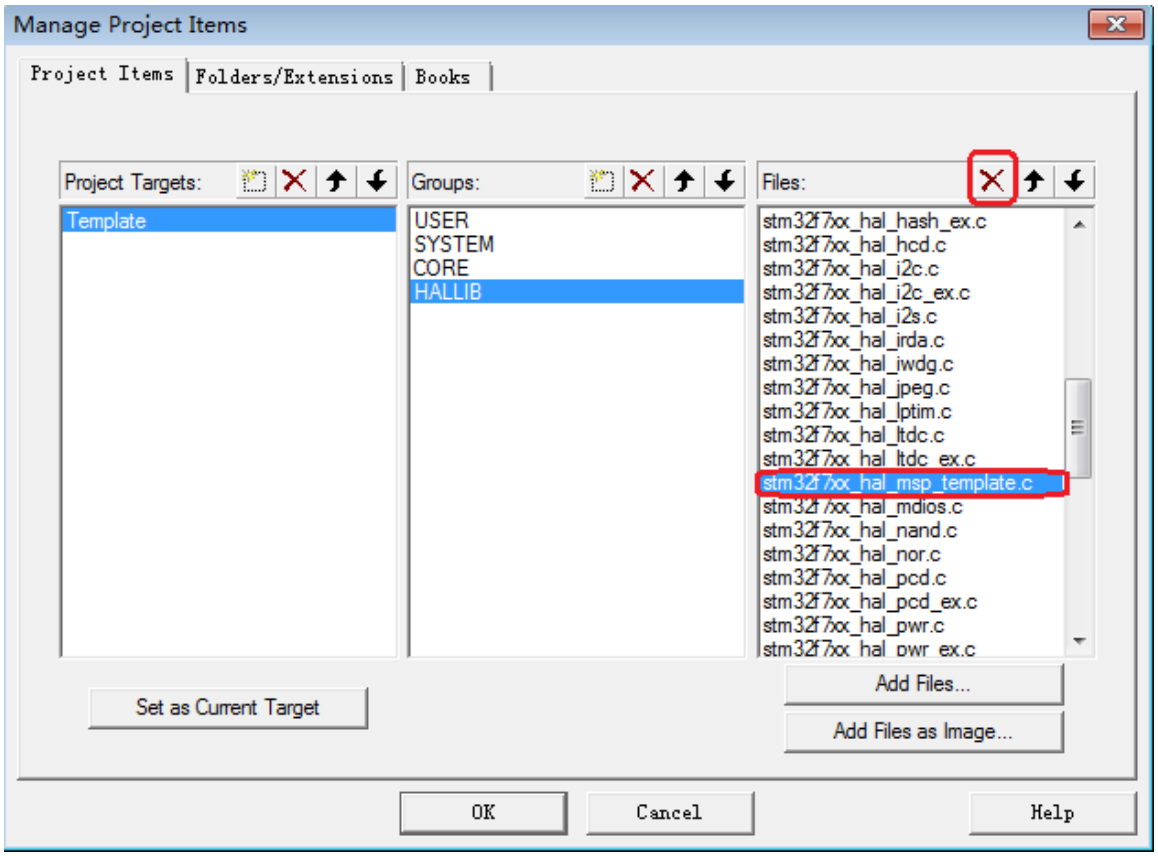


图 3.3.1.18 删掉 HALLIB 分组中不需要的源文件

使用同样的方法删除其他几个不需要添加的文件即可。

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_stm32f767xx.s(注意，默认添加的时候文件类型为.c，添加.h 头文件和 startup_stm32f767xx.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，stm32f7xx_hal_msp.c，stm32f7xx_it.c 和 system_stm32f7xx.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：

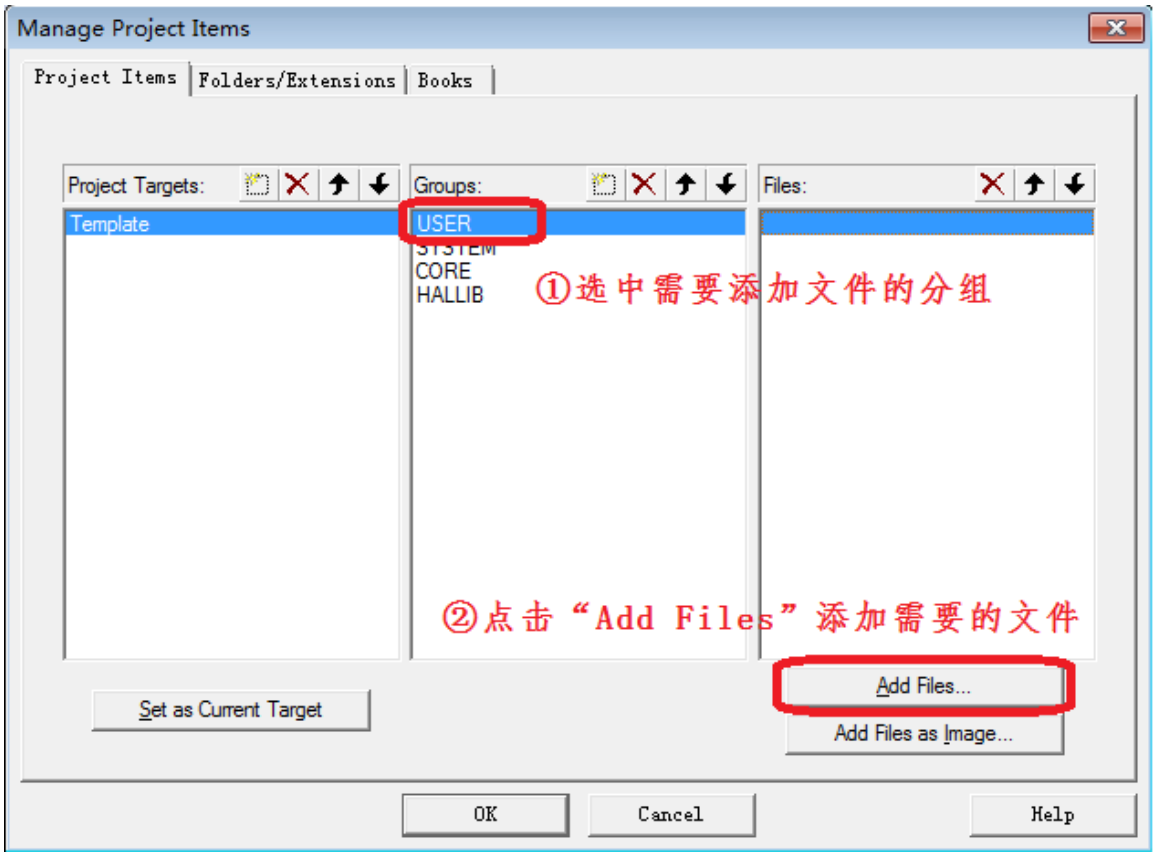


图 3.3.1.19 添加文件到 USER 分组

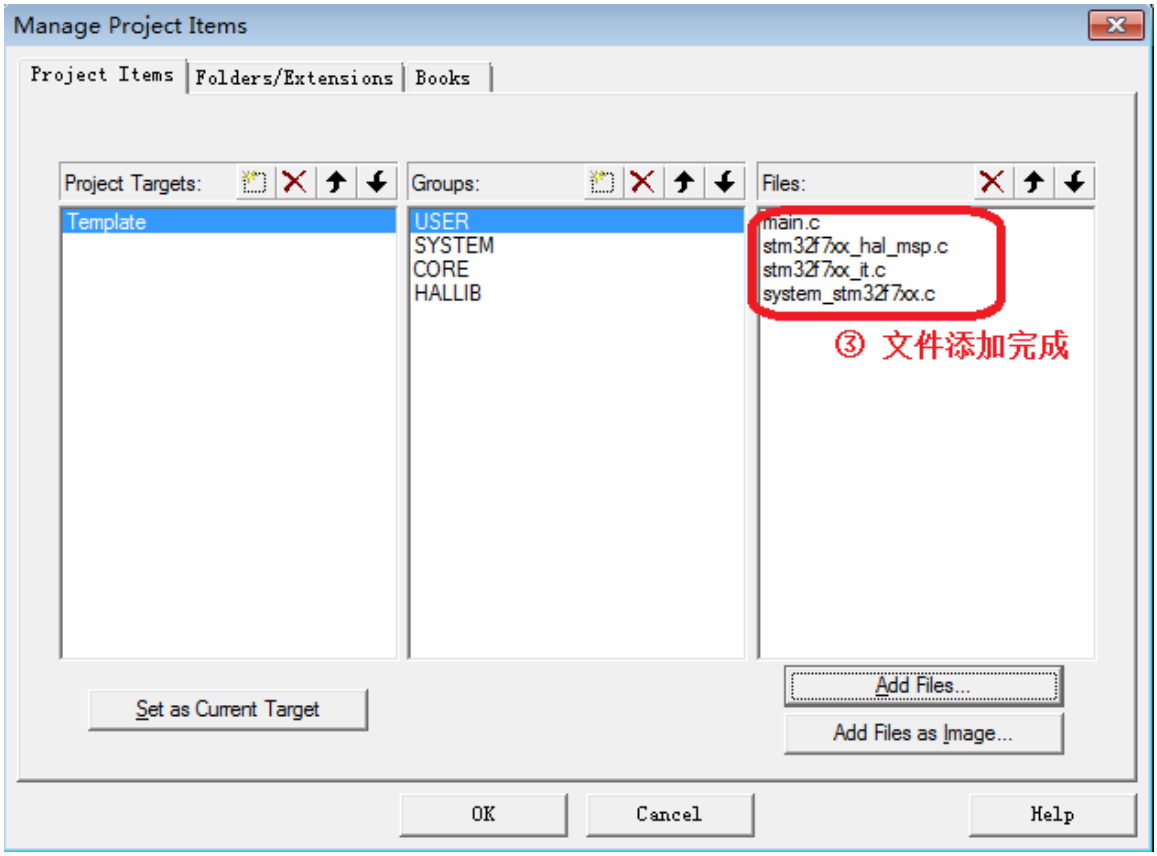


图 3.3.1.20 文件添加到 USER 分组完成

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。

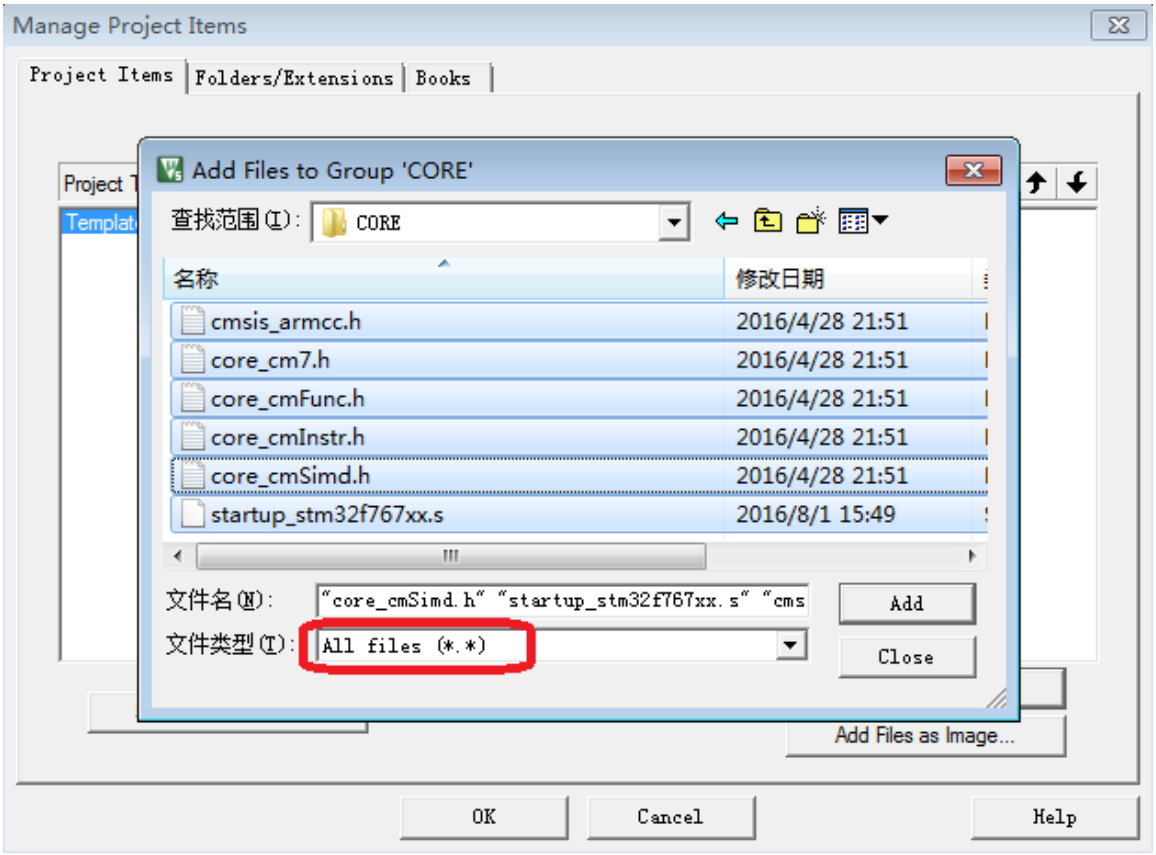


图 3.3.1.21 添加.h 头文件和启动文件到 CORE 分组

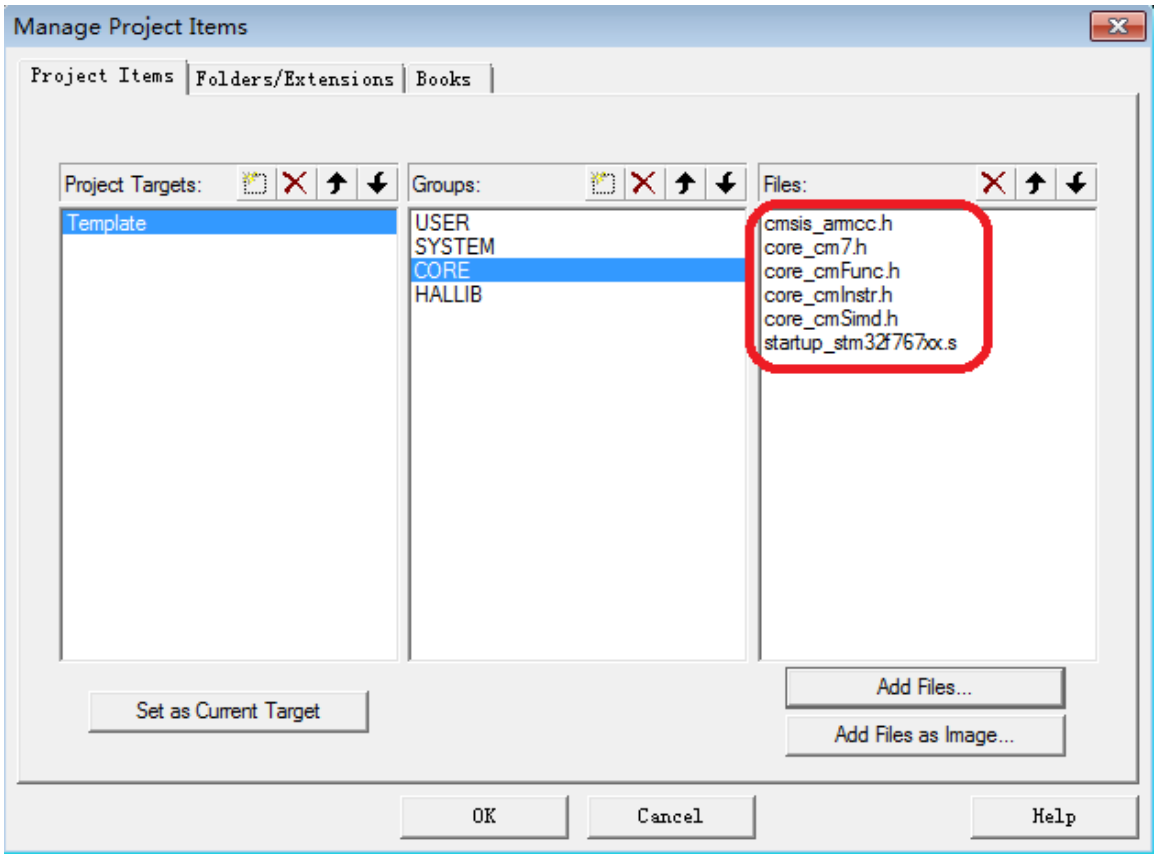


图 3.3.1.22 添加启动文件和头文件到 CORE 分组完成

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：

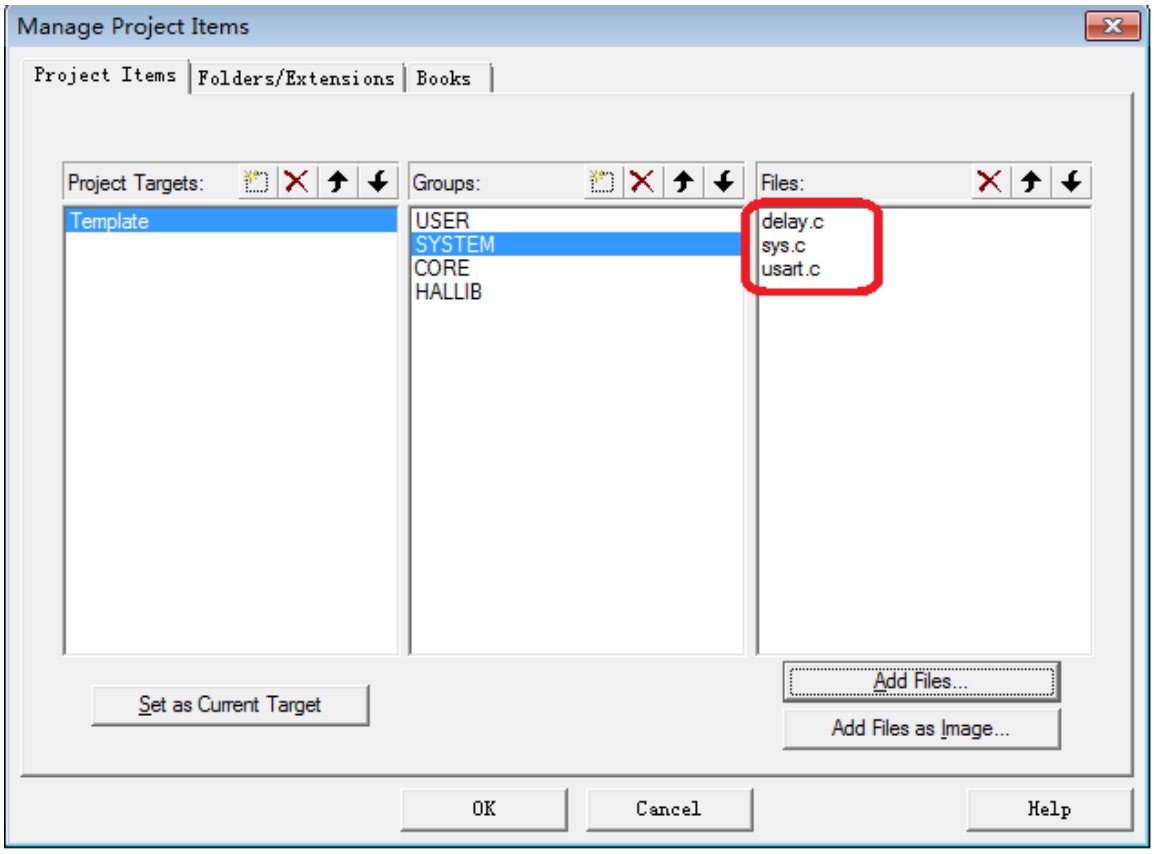


图 3.3.1.23 添加文件到 SYSTEM 分组

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：

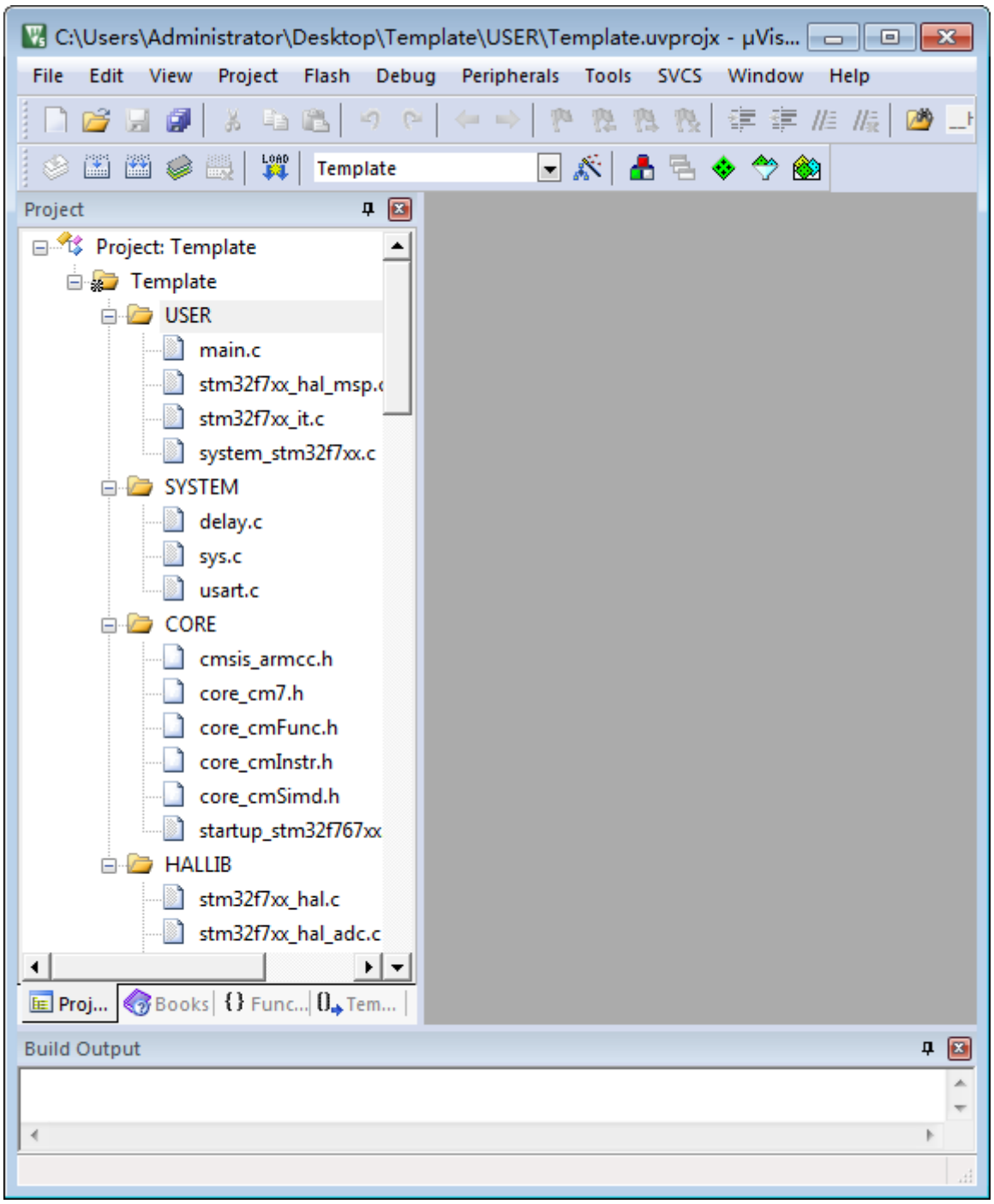


图 3.3.1.24 工程分组情况

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。

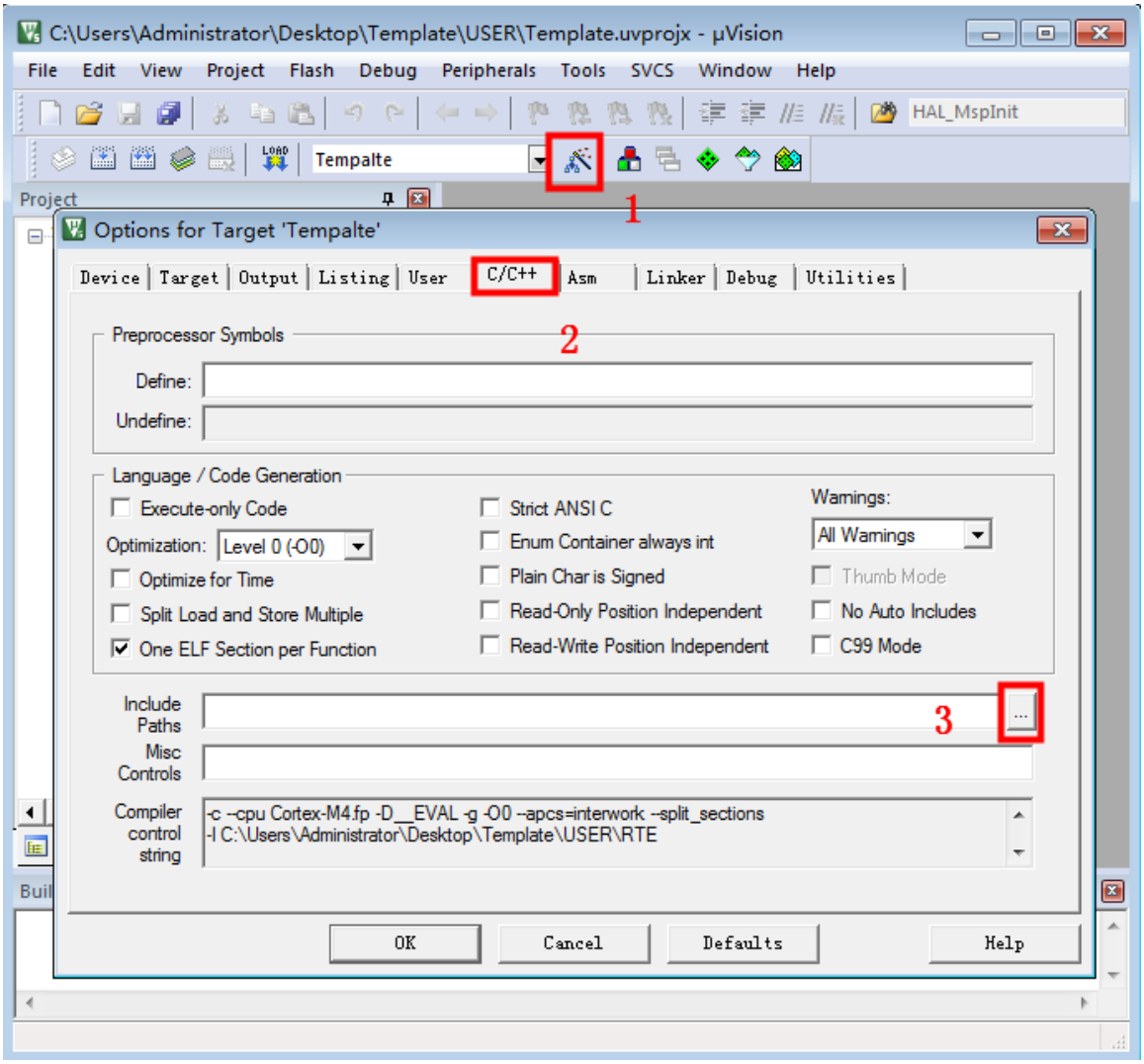


图 3.3.1.25 进入 PATH 配置界面

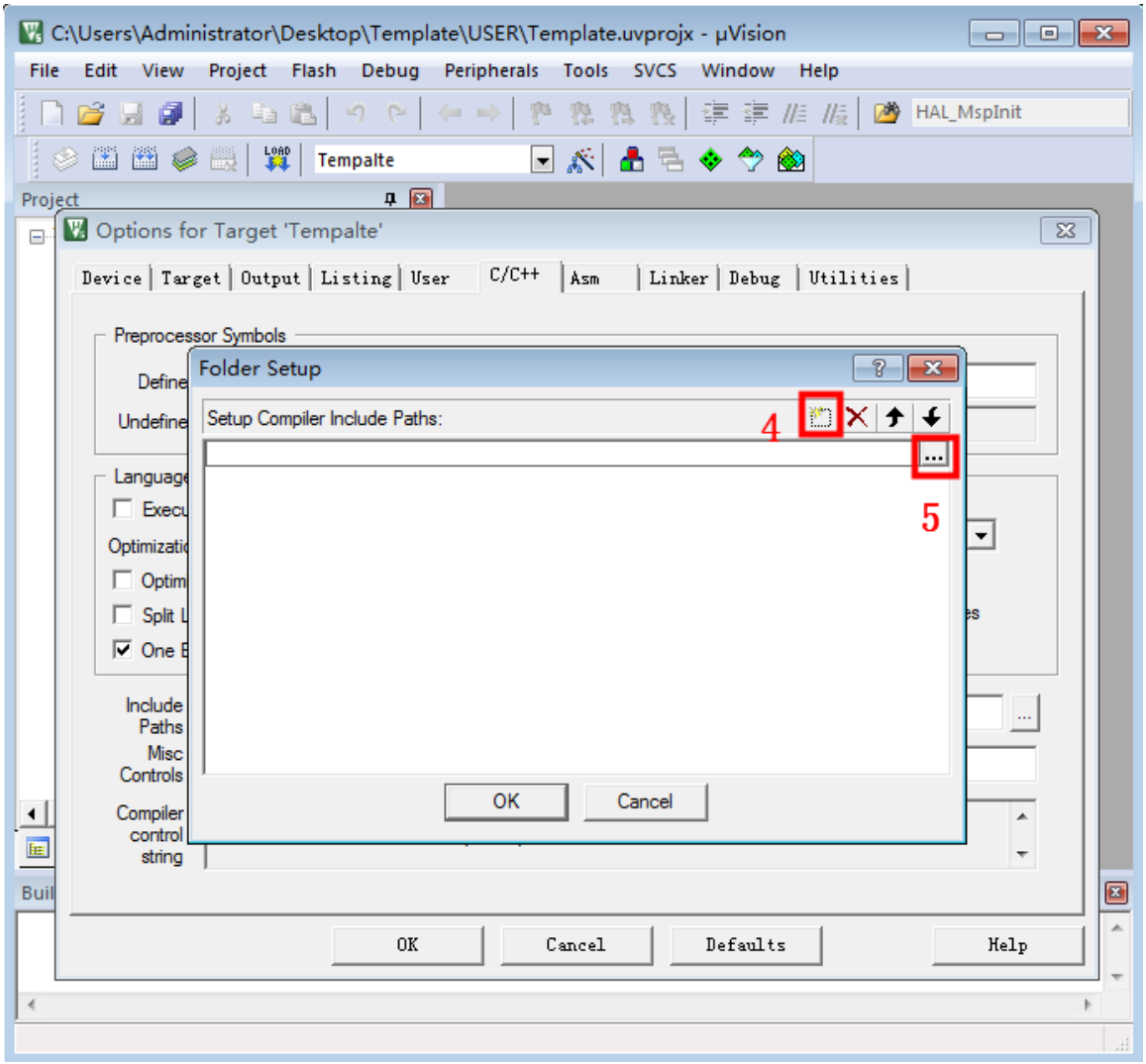


图 3. 3. 1. 26 添加头文件路径到 PATH

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。

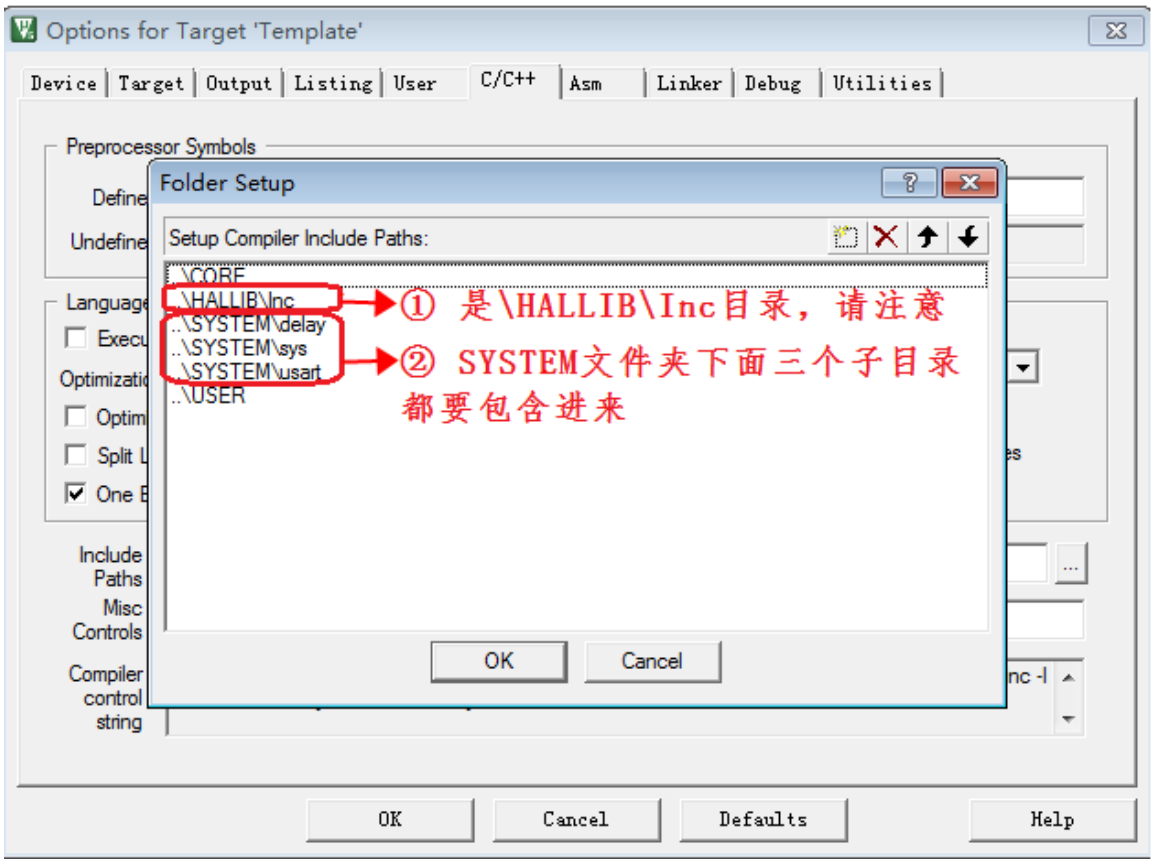


图 3.3.1.27 添加头文件路径

13） 接下来对于 STM32F7 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,STM32F767xx。注意这里是两个标识符 USE_HAL_DRIVER 和 STM32F767xx，**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：

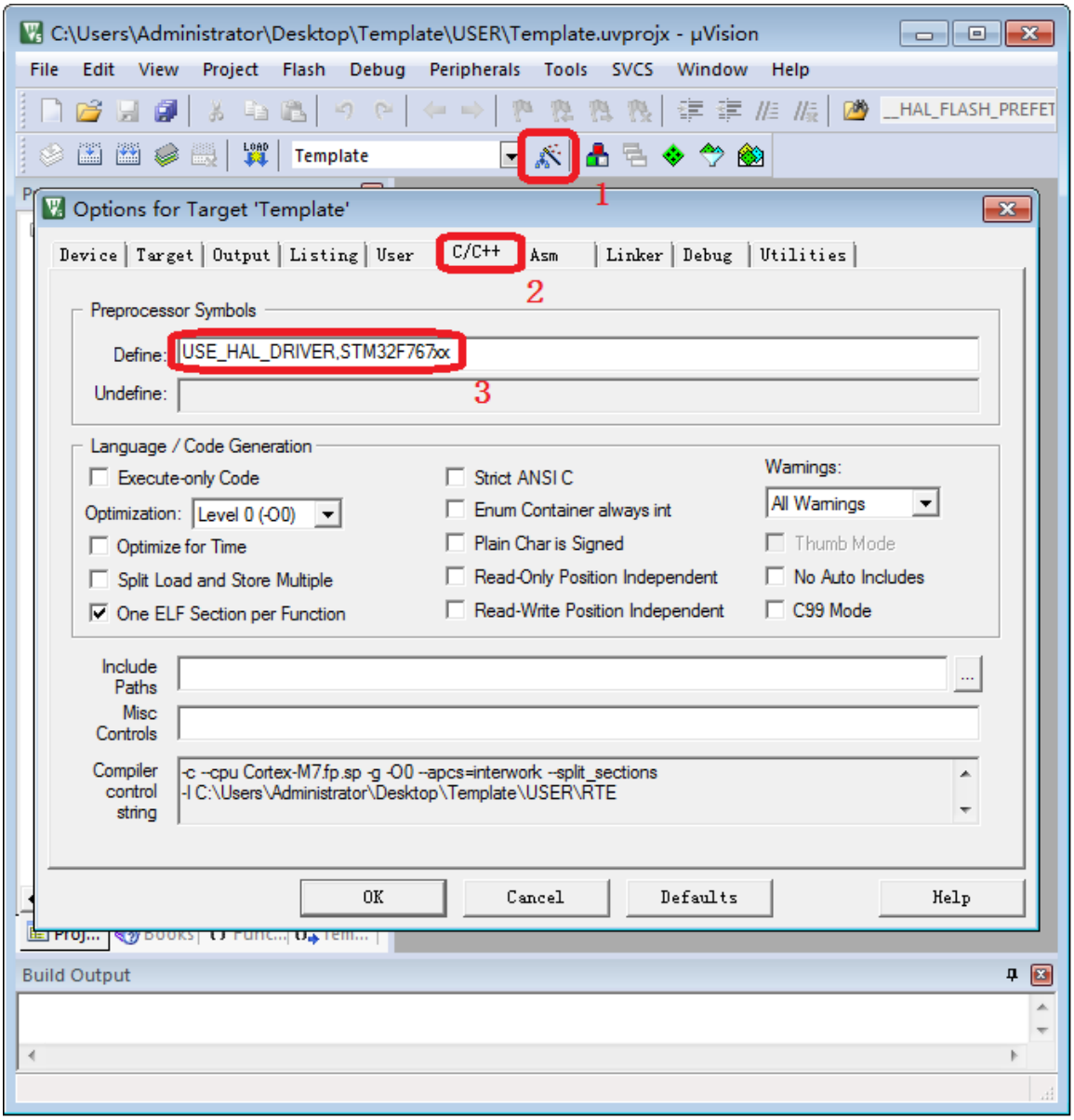


图 3.3.1.28 添加全局宏定义标识符

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：

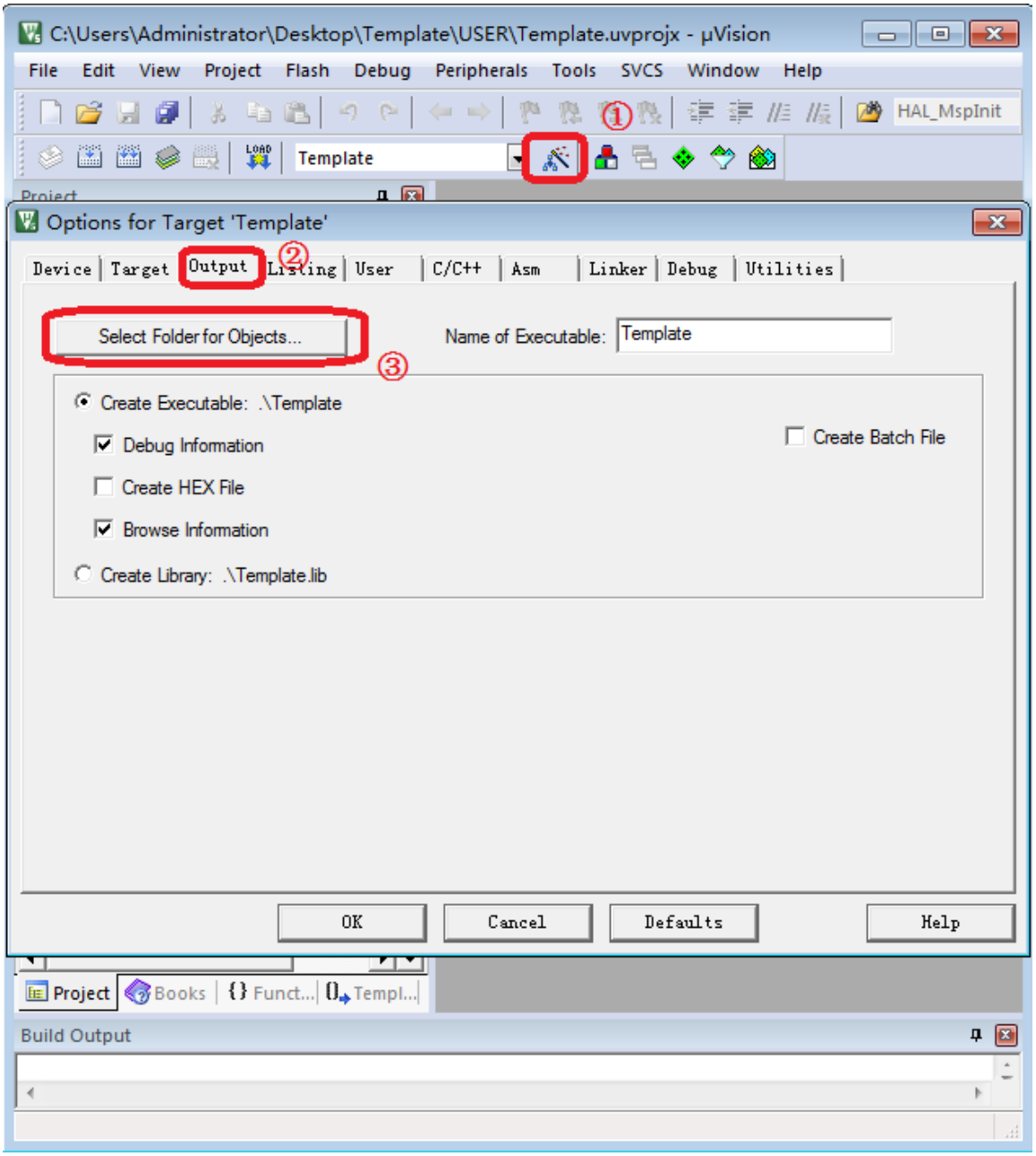


图 3.3.1.29 点击按钮“Select Folder for Objects…”

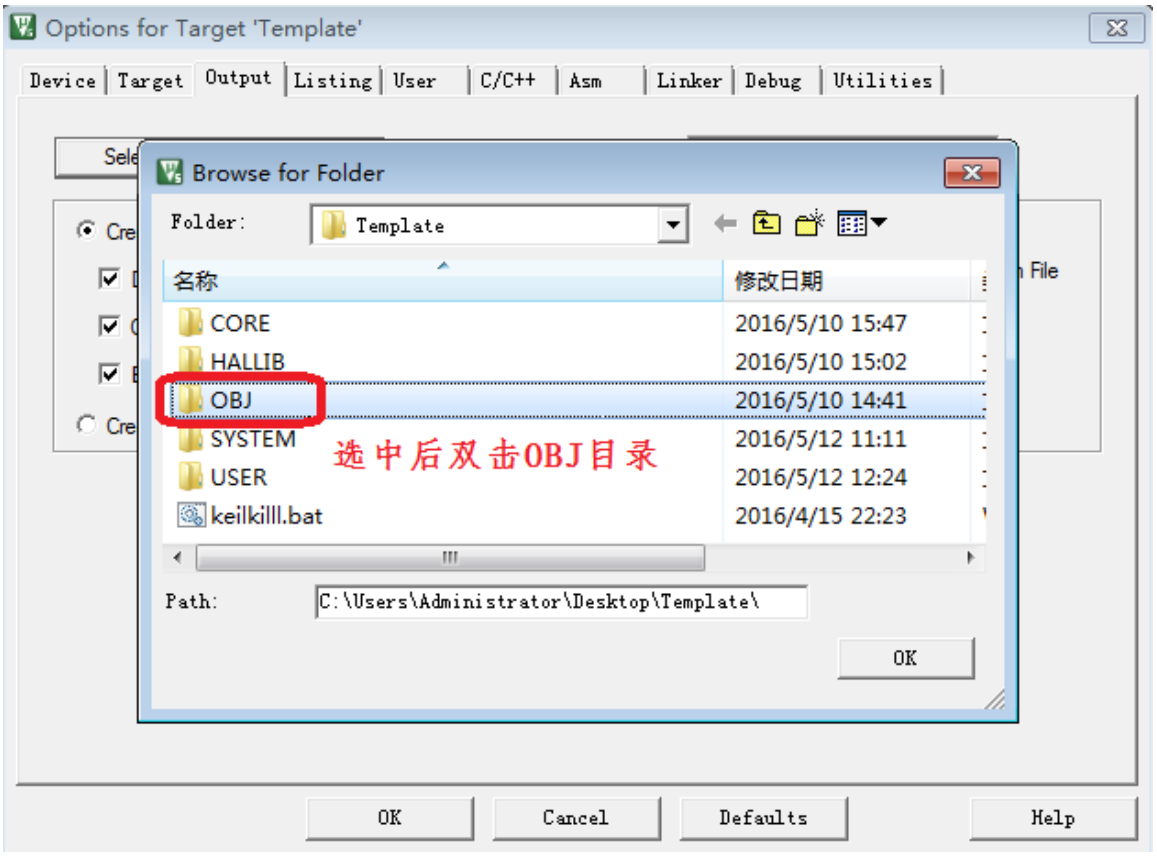


图 3.3.1.30 选择 OBJ 目录为中间文件存放目录

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：

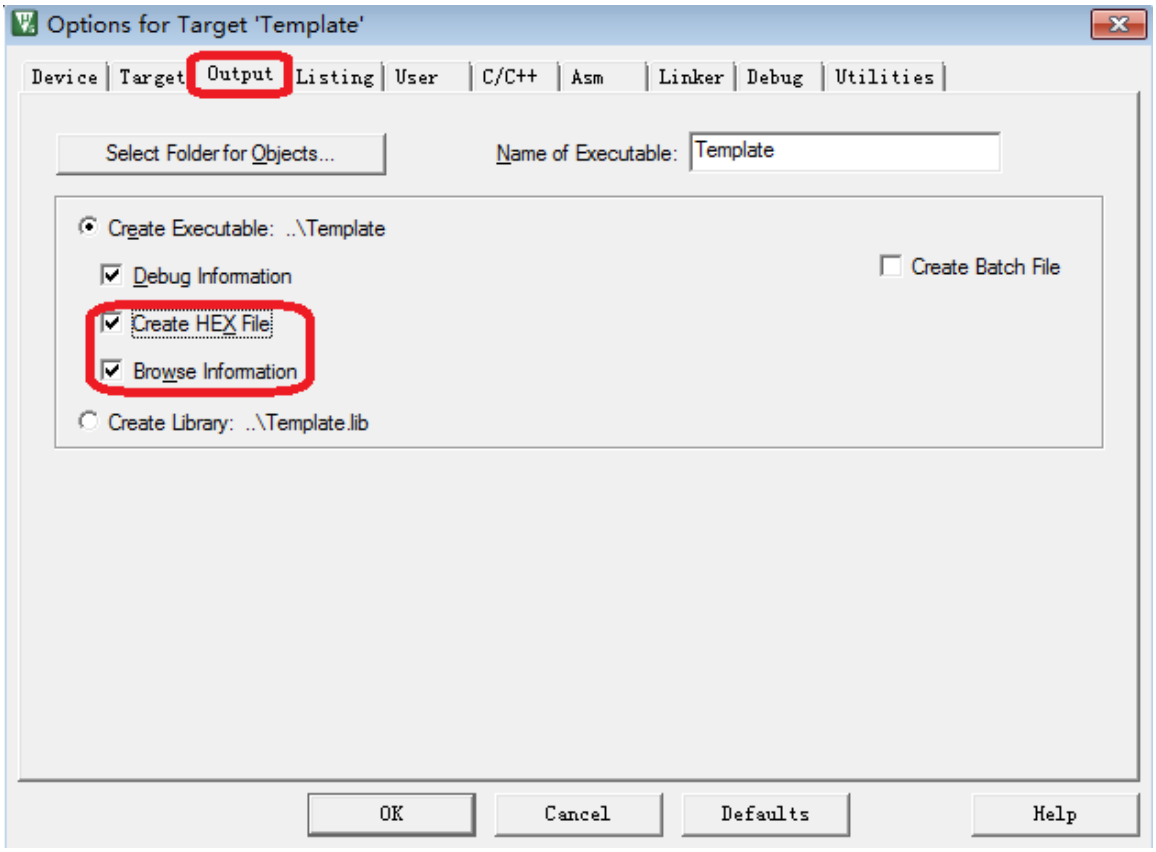


图 3.3.1.31 勾选上 Create HEX file 和 Browse Information 选项

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：

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
void Delay(__IO uint32_t nCount);
void Delay(__IO uint32_t nCount)
{
while(nCount--){}
}

int main(void)
{
    GPIO_InitTypeDef GPIO_Initure;

      Cache_Enable();                    //打开 L1-Cache
      HAL_Init();                        //初始化 HAL 库
      Stm32_Clock_Init(432,25,2,9);     //设置时钟,216Mhz
      __HAL_RCC_GPIOB_CLK_ENABLE();              //开启 GPIOB 时钟

      GPIO_Initure.Pin=GPIO_PIN_0|GPIO_PIN_1; //PB1,0
      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(GPIOB,&GPIO_Initure);

    while(1)
    {
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_SET);   //PB1 置 1
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_SET);   //PB0 置 1
        Delay(0x7FFFFF);
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);     //PB1 置 0
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_RESET);     //PB0 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） 这个时候我们对工程进行编译会发现，编译结果有错误，编译结果如下图 3.3.1.32 所示：

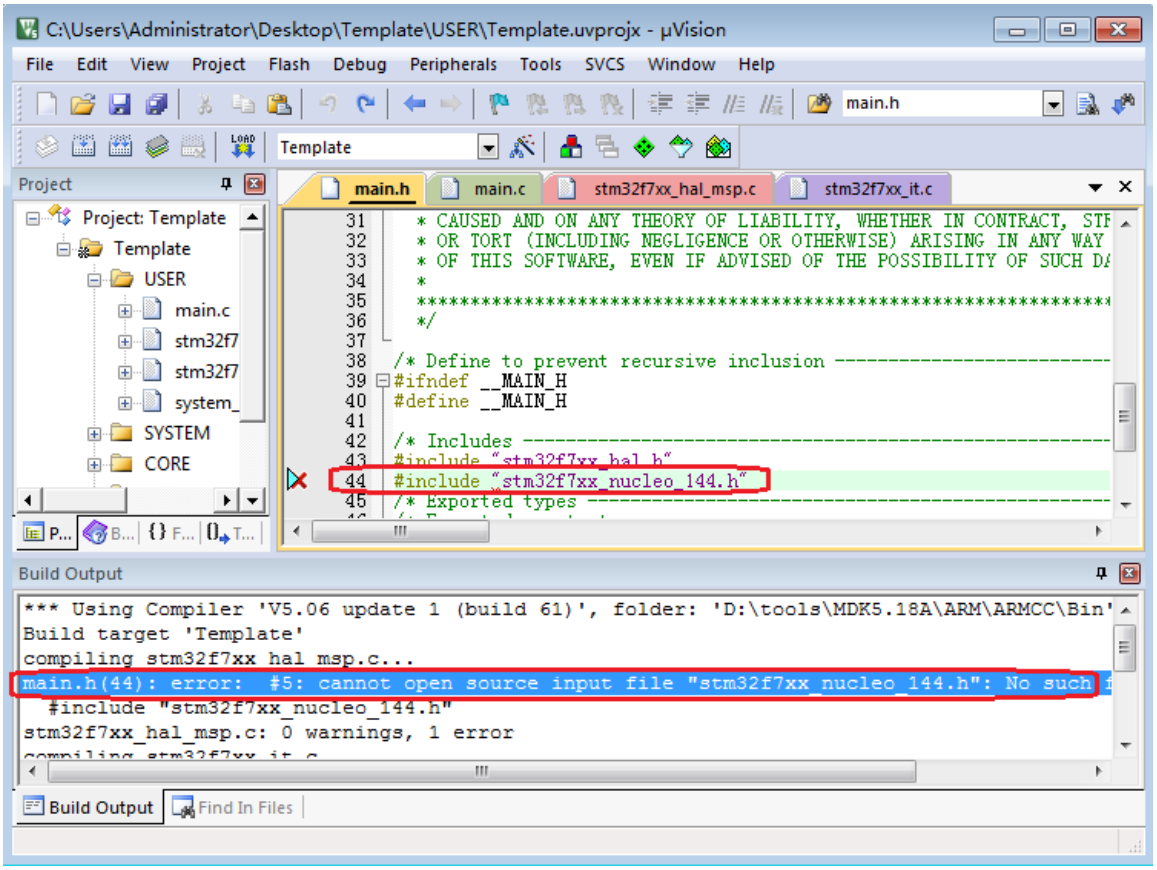
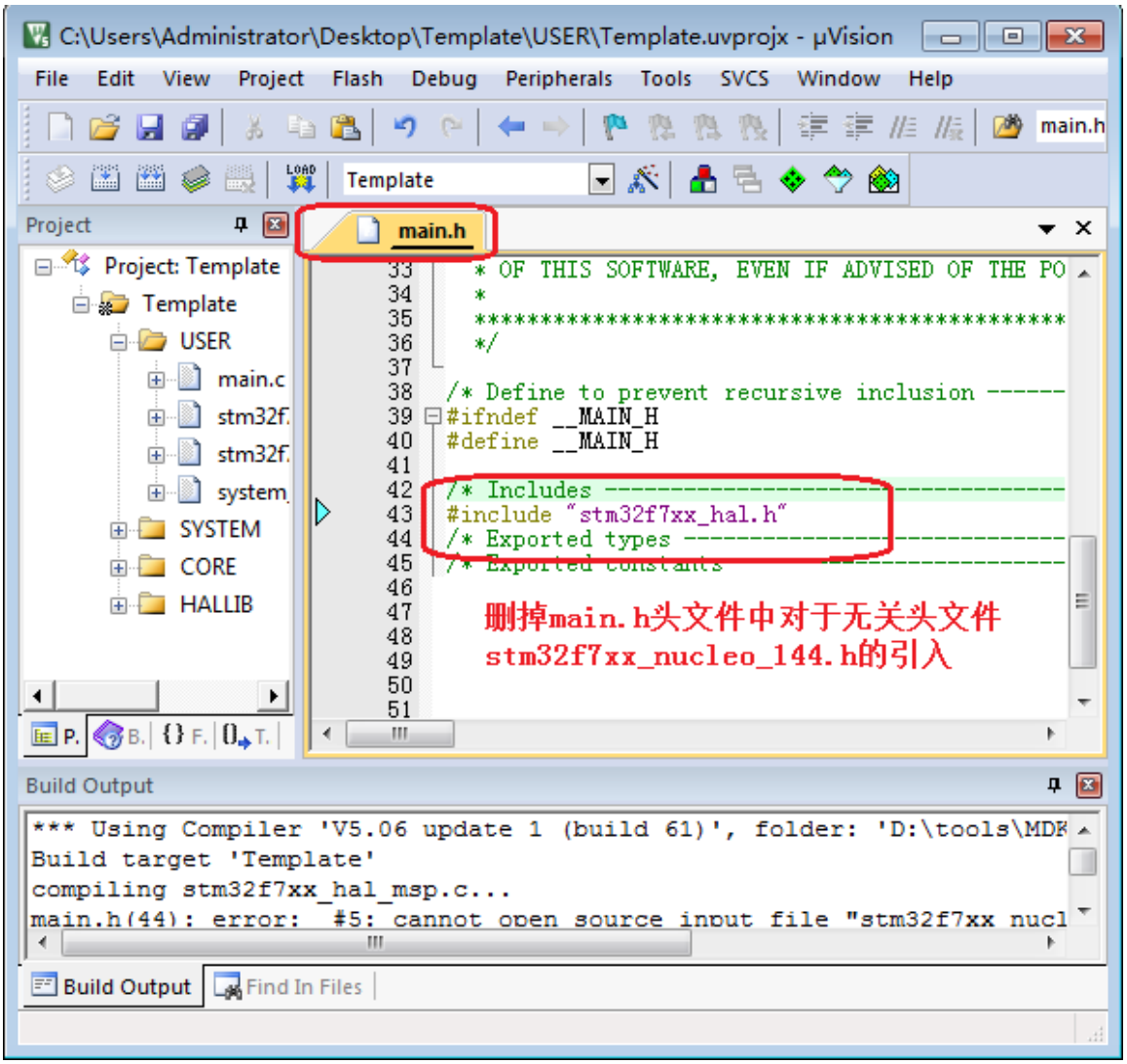


图 3.3.1.32 编译结果

从图中可以看出错误信息为“main.h(44): error: #5: cannot open source input file "stm32f7xx_nucleo_144.h": No such file or directory”，也就是说错误是在 main.h 头文件的第

44 行中引入了头文件 stm32f7xx_nucleo_144.h，而这个头文件在工程中找不到。这是因为这个头文件是从 ST 的模板中引入，ST 的模板中对于每个开发板都有一个头文件，并且在 main.h 中引入，所以这里对于我们的平台是无关头文件，我们需要从 main.h 头文件中把对该头文件的引入代码删掉。删掉后的 main.h 内容如下图所示：



17） 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。

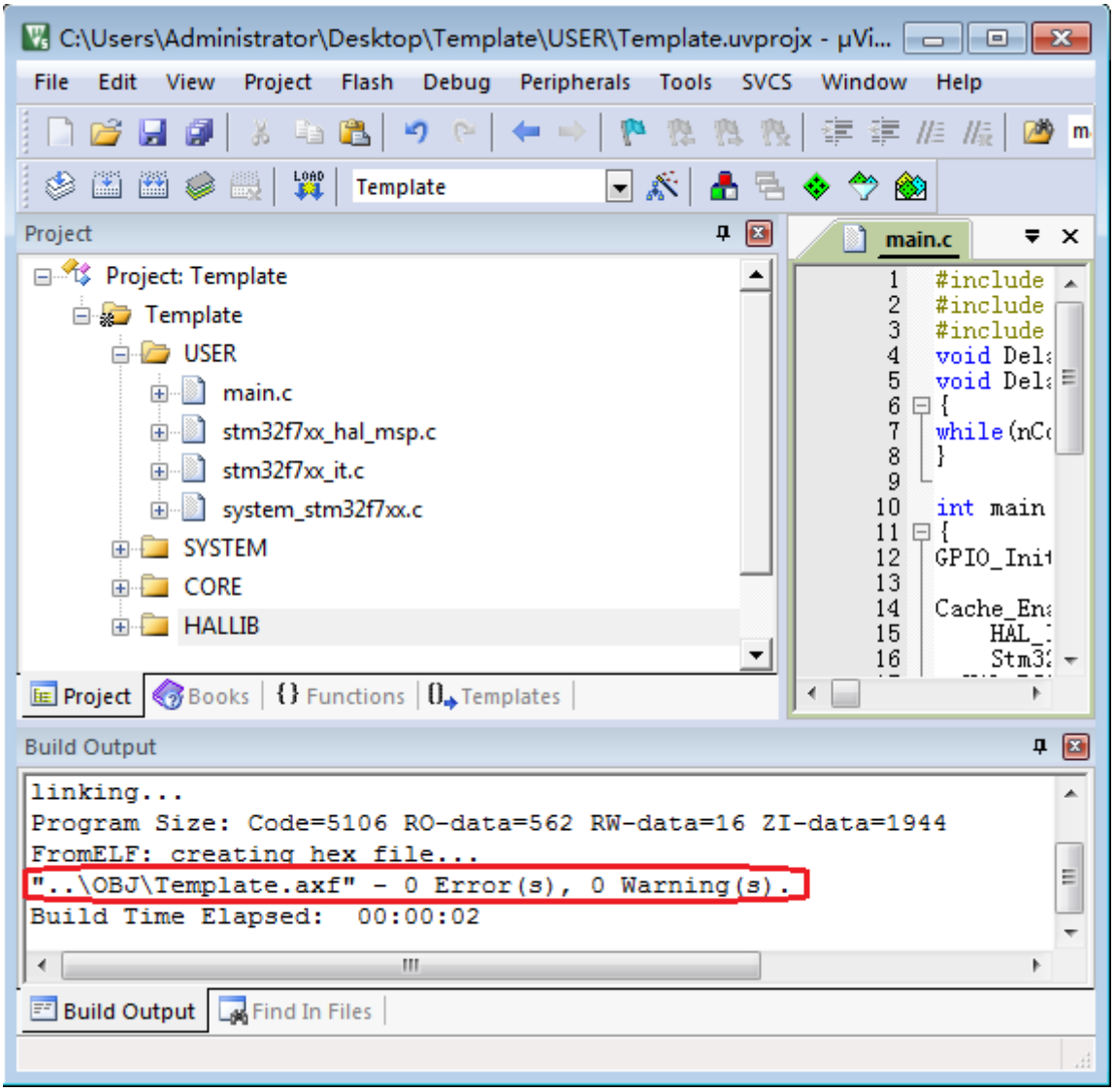


图 3.3.1.32 编译工程

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 库的工程模板就基本建立完成，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。但是这个工程到这一步实际还没有配置完成，还需要根据我们开发板的外部高速晶振值大小来修改配置文件 stm32f7xx_hal_conf.h，打开该文件定位到如下内容：

```
#if !defined  (HSE_VALUE)
    #define HSE_VALUE      ((uint32_t)8000000U)
#endif
```

因为阿波罗开发板外部晶振使用的是 25MHz，所以这里我们需要把 HSE_VALUE 值修改为 25000000U。修改后内容如下：

```
#define HSE_VALUE      ((uint32_t)25000000U)
```

18） 最后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 stm32f7xx_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 STM32F7 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 STM32F7 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
stm32f7xx_hal_ppp.c/.h	基本外设的操作 API,ppp 代表任意外设。其中 stm32f7xx_hal_cortex.c/.h 比较特殊，它是一些 Cortex 内核通用函数声明和定义，例如中断优先级 NVIC 配置，系统软复位以及 Systick 配置等。
stm32f7xx_hal_ppp_ex.c/.h	拓展外设特性的 API。
sm32f7xx_hal.c	包含 HAL 通用 API（比如 HAL_Init,HAL_DeInit,HAL_Delay 等）。
stm32f7xx_hal.h	HAL 的头文件，它应被客户代码所包含。
stm32f7xx_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。
stm32f7xx_hal_def.h	包含 HAL 的通用数据类型定义和宏定义
stm32f7xx_ll_ppp.c/.h	在一些复杂外设中实现底层功能，它们在 stm32f7xx_hal_ppp.c 中被调用

表 3.3.2.1 HAL 库文件介绍

2) stm32f7xx_it.c/stm32f7xx_it.h 文件

这两个文件非常简单，也非常好理解。stm32f7xx_it.h 中主要是一些中断服务函数的申明。stm32f7xx_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) stm32f7xx.h 头文件

头文件 stm32f7xx.h 内容看似非常少，却非常重要，它是所有 stm32f7 系列的顶层头文件。使用 STM32F7 任何型号的芯片，都需要包含这个头文件。同时，因为 stm32f7 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 STM32F767 系列，ST 定义了一个头文件 stm32f767xx.h，然后 stm32f7xx.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 stm32f7xx.h 头文件可以看到，里面有如下几行代码：

```
#if defined(STM32F756xx)
  #include "STM32F756xx.h"
  …
#elif defined(stm32f767xx)
  #include "stm32f767xx.h"
…
#else
 #error "Please select first the target STM32F7xx device used in your application"
#endif
```

这几行代码非常好理解，我们以 stm3f767 为例，如果定义了宏定义标识符 STM32F767xx，那么头文件 stm32f7xx.h 将会包含头文件 stm32f767xx.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 STM32F767xx（请参考图 3.3.1.28）。所以头文件 stm32f767xx.h 一定会被整个工程所引用。

4) stm32f767xx.h 头文件

根据前面的讲解，stm32f767xx.h 是 stm32f767 系列芯片通用的片上外设访问层头文件，只要我们进行 STM32F767 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_stm32f7xx.c/system_stm32f7xx.h 文件

头文件 system_stm32f7xx.h 和源文件 system_stm32f7xx.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_stm32f767xx.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) stm32f7xx_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。stm32f7xx_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 stm32f7xx_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 stm32f7xx_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_stm32f767xx.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 STM32F767 系列，所以我们需要使用与之对应的启动文件 startup_stm32f767xx.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_stm32f767xx.s，可以看到下面几行代码：

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

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

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。

接下来我们看看 HAL 库工程模板中各个文件之间的包含关系，如下图 3.3.2.2 所示：

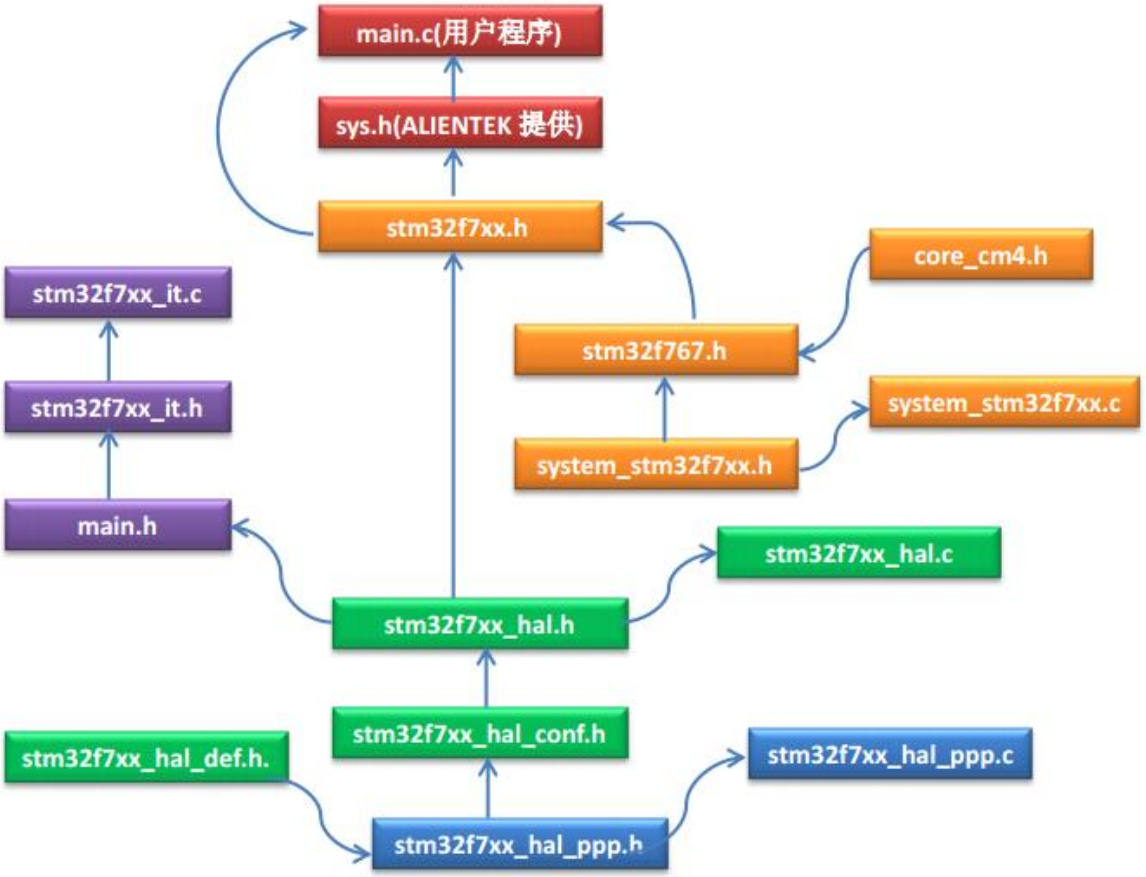


图 3.3.2.2 HAL 库工程文件包含关系

从上面工程文件包含关系图可以看出，顶层头文件 stm32f7xx.h 直接或间接包含了其他所有工程必要头文件，所以在我们的用户代码中，我们只需要包含顶层头文件 stm32f7xx.h 即可。这里我们还需要说明一下，在我们 ALIENTEK 提供的 SYSTEM 文件夹内部的 sys.h 头文件中，我们默认包含了 stm32f7xx.h 头文件，所以在我们用户代码中，只需要包含 sys.h 头文件即可，当然也可以直接包含顶层头文件 stm32f7xx.h。关于工程模板中关键文件内容我们就给大家介绍到这里。

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 stm32f7xx_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，是一个空函数，没有任何控制逻辑。同时，在 stm32f7xx_hal.c 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 stm32f7xx_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 stm32f7xx_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 80 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 stm32f7xx_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 STM32F7 来做承载，PA9 做为发送，PA10 做为接收，MSP 就是要初始化 STM32F7 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：

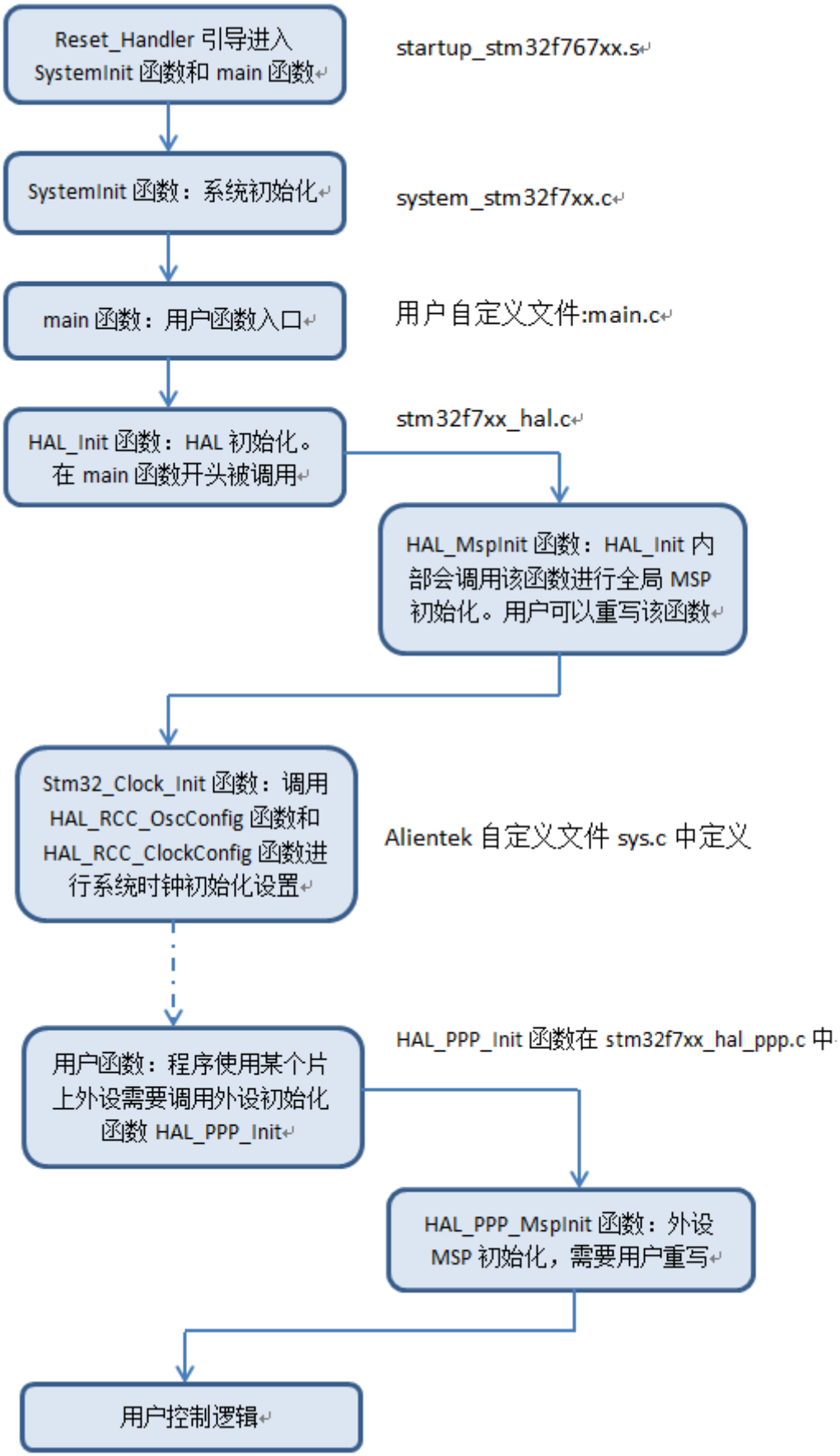

Reset_Handler 引导进入 SystemInit 函数和 main 函数
startup_stm32f767xx.s
SystemInit 函数：系统初始化
system_stm32f7xx.c
main 函数：用户函数入口
用户自定义文件:main.c
HAL_Init 函数：HAL 初始化。在 main 函数开头被调用
stm32f7xx_hal.c
HAL_MspInit 函数：HAL_Init 内部会调用该函数进行全局 MSP 初始化。用户可以重写该函数
Stm32_Clock_Init 函数：调用 HAL_RCC_OscConfig 函数和 HAL_RCC_ClockConfig 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 stm32f7xx_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_stm32f767xx.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 STM32F7 工程。本节，我们将向读者介绍 STM32F7 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、STM32F7 程序下载；2、利用 ST-LINK 对 STM32F7 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 STM32F7 程序下载

给 STM32F767 下载代码。由于 STM32F7 暂时没有比较好的串口下载软件，所以，我们一般通过仿真器下载，接下来，我们将介绍如何使用 ST LINK V2，结合 MDK，来给 STM32F7 下载代码。

ST LINK 支持 JTAG 和 SWD 两种通信接口，同时 STM32F767 也支持 JTAG 和 SWD。所以，我们有 2 种方式可以用来下载代码，JTAG 模式，占用的 IO 线比较多，而 SWD 模式则占用的 IO 线很少，只需要两根即可。所以，我们一般选择 SW 模式来给 STM32F767 下载代码。

首先，我们需要安装 ST LINK 的驱动。驱动安装，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。

在安装了 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 JTAG 接口，打开之前 3.2 节新建的工程，点击，打开 Options for Target 选项卡，在 Debug 栏选择仿真工具为 ST-Link Debugger，如图 3.4.1.1 所示：

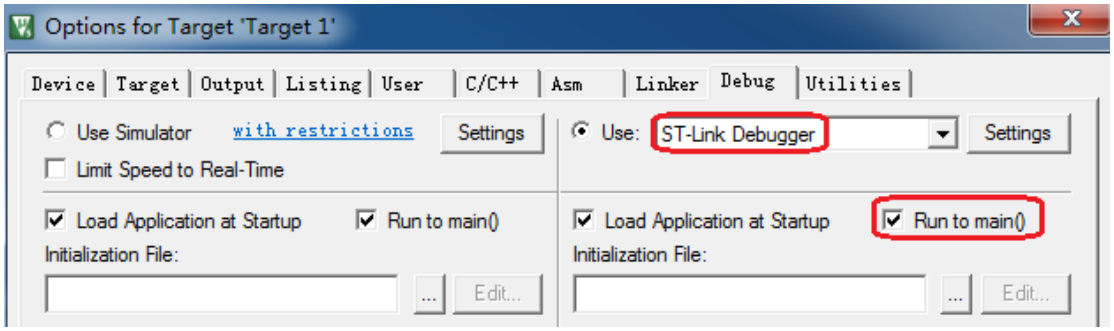


图 3.4.1.1 Debug 选项卡设置

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_stm32f767xx.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 3.4.1.2 所示：

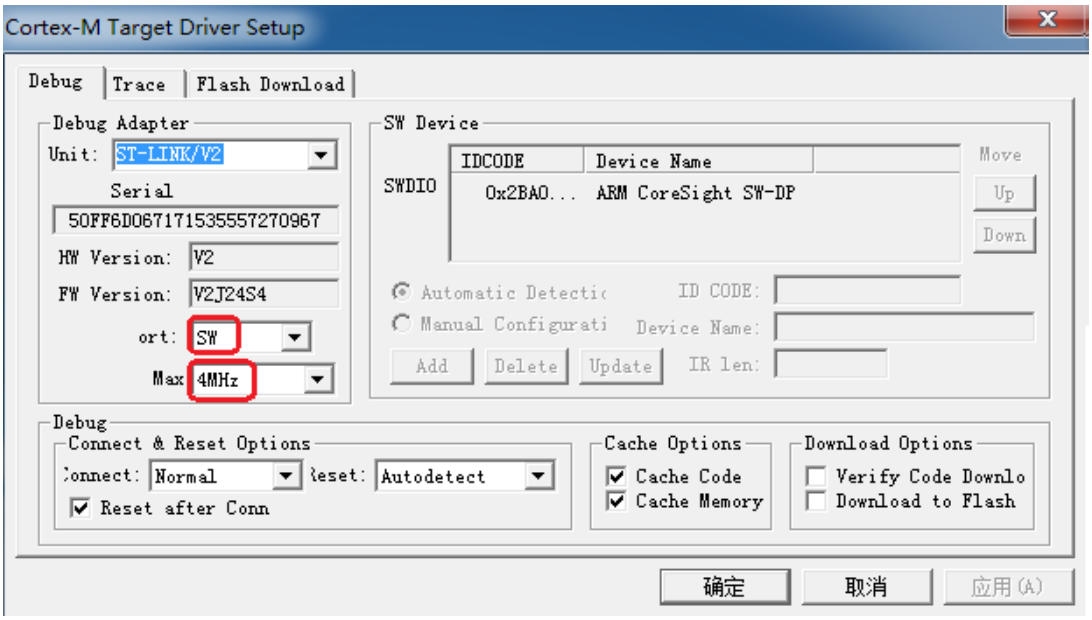


图 3.4.1.2 ST LINK 模式设置

图 3.4.1.2 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**在下载/调试代码的时候，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 3.4.1.3 所示：

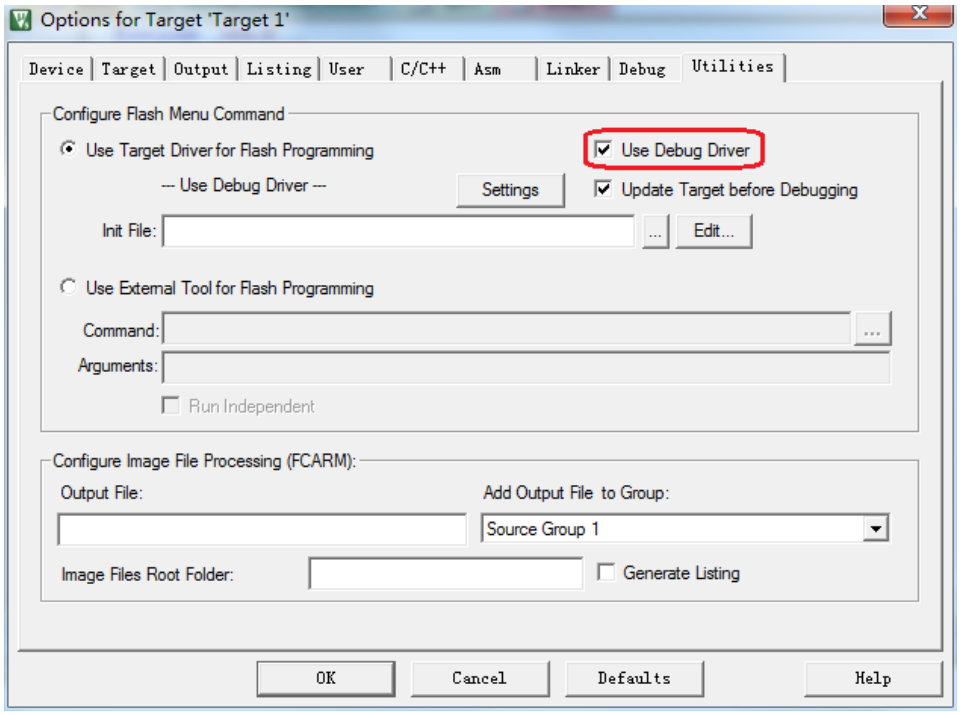


图 3.4.1.3 FLASH 编程器选择

图 3.4.1.3 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 3.4.1.4 所示：

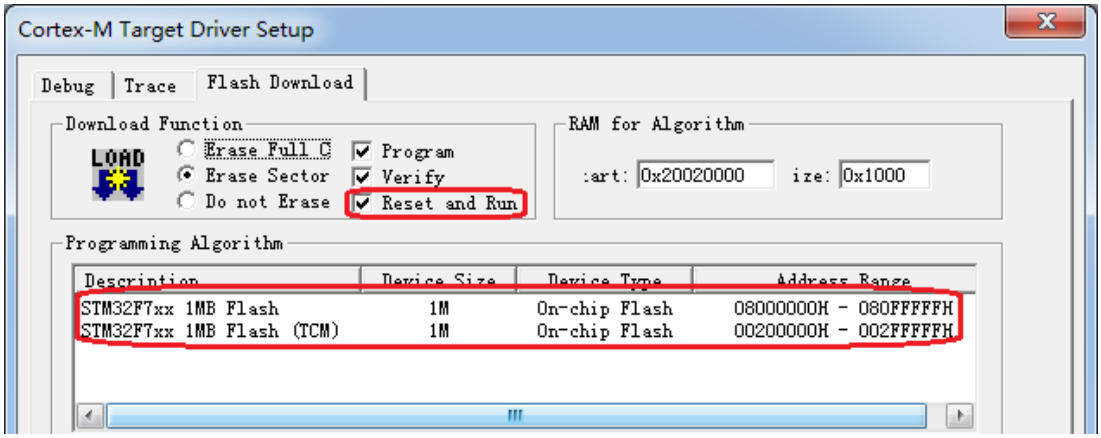


图 3.4.1.4　编程设置

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 STM32F767IGT6，FLASH 容量为 1M 字节，所以 Programming Algorithm 里面默认会有 1M 型号的 STM32F7xx FLASH 算法。MDK 默认选择的是 AXIM 总线访问的 FLASH 算法（起始地址为：0X0800 0000），为了方便大家使用，我们将 ITCM 总线访问的 FLASH 算法（起始地址为：0X0020 0000）也添加进来，由 MDK 自动选择下载算法（实际上是根据 Target 选项卡的 on-chip IROM 地址范围设置来选择的，默认为 0X0800 0000）。

特别提醒：这里的 1M flash 算法，不仅仅针对 1M 容量的 STM32F767，对于小于 1M FLASH 的型号，也是采用这个 flash 算法的。最后，选中 Reset and Run 选项，以实现在编程后自动运行，其他默认设置即可。设置完成之后，如图 3.4.1.4 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。然后点击：  （下载按钮），就可以下载代码到 STM32F767 上面了，如图 3.4.1.5 所示：

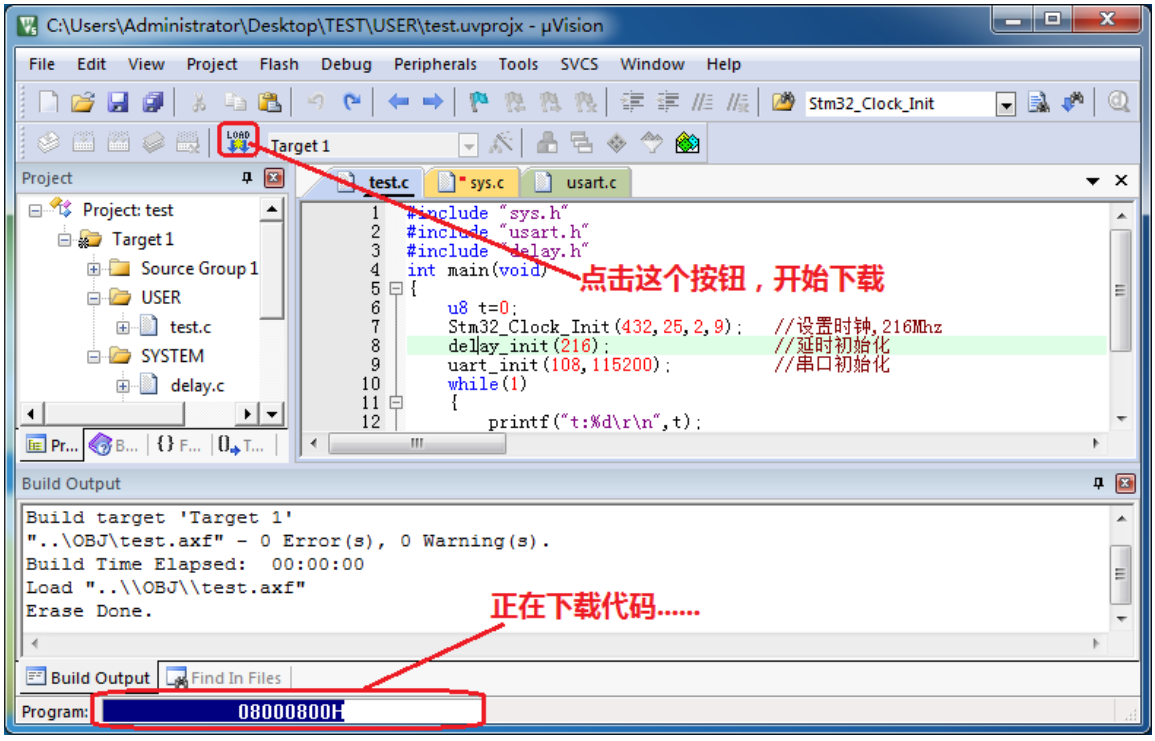


图 3.4.1.5　通过仿真器给 STM32F767 下载代码

下载完成后，在 Build Output 窗口，会提示 Programming Down，Application running…，如图 3.4.1.6 所示：

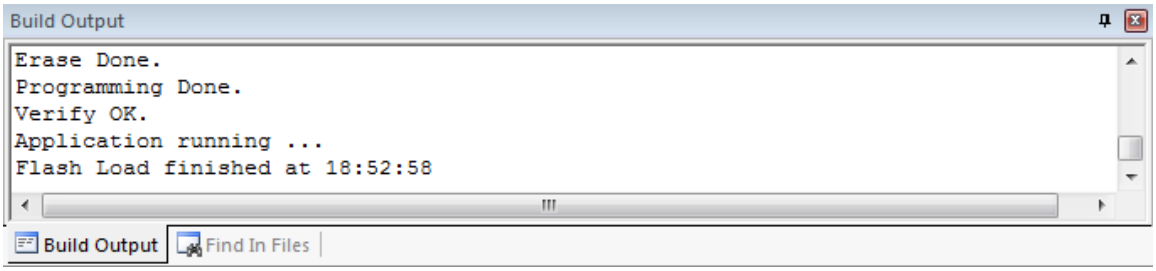


图 3.4.1.6 下载完成并运行代码

下载完后，会自动运行我们刚刚下载的代码（因为我们勾选了 Reset and run，见图 3.4.1.4），接下来我们就可以打开串口调试助手，来验证是否收到了 STM32F767 串口发送出来的数据。

我们在开发板的 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 3.4.1.7 所示：

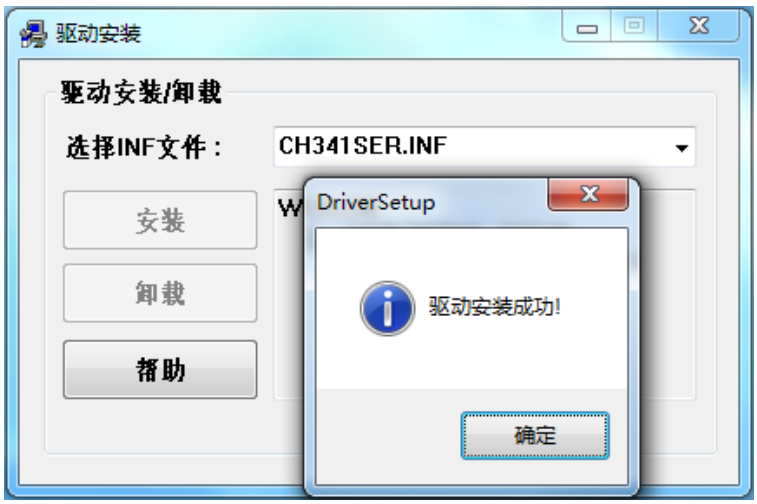


图 3.4.1.7 CH340 驱动安装

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 3.4.1.8 所示：

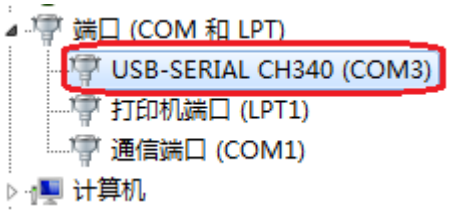


图 3.4.1.8 USB 串口

在图 3.4.1.8 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。

在安装完 USB 串口驱动之后，我们就可以开始验证了（**注意，开发板的 B0 必须接 GND，**

否则将不会运行用户下载的代码！！），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 阿波罗 STM32F767 开发板发回来的信息，如图 3.4.1.9 所示：

图 3.4.1.9 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 STM32F7 在线调试

上一节，我们介绍了如何利用 ST LINK 给 STM32 下载代码，并在 ALIENTEK 阿波罗 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，就有必要通过在线调试来解决问题了。

利用调试工具，比如 JLINK（**必须是 JLINK V9 或者以上版本！！**）、ULINK、STLINK 等，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 公司自家的仿真器：ST LINK V2 为例，说说如何在线调试 STM32F767。

通过上一节的学习，我们知道 ST LINK 支持 JTAG 和 SWD 两种通信方式，而且 SWD 方式具有占用 IO 少的优点（2 个 IO 口），所以，我们一般选择 SWD 方式进行调试。MDK 里面，对 ST LINK 的相关设置，同上一节完全一样，请参考上一节的相关介绍。

在 MDK 的 IDE 界面，编译一下工程。然后点击：  （开始/停止仿真按钮），开始仿真（如果开发板的代码没被更新过，则会先更新代码（即下载代码），再仿真。特别注意：开发板上的 B0 脚要接 GND，否则不会运行我们下载的代码！！），如图 3.4.2.1 所示：

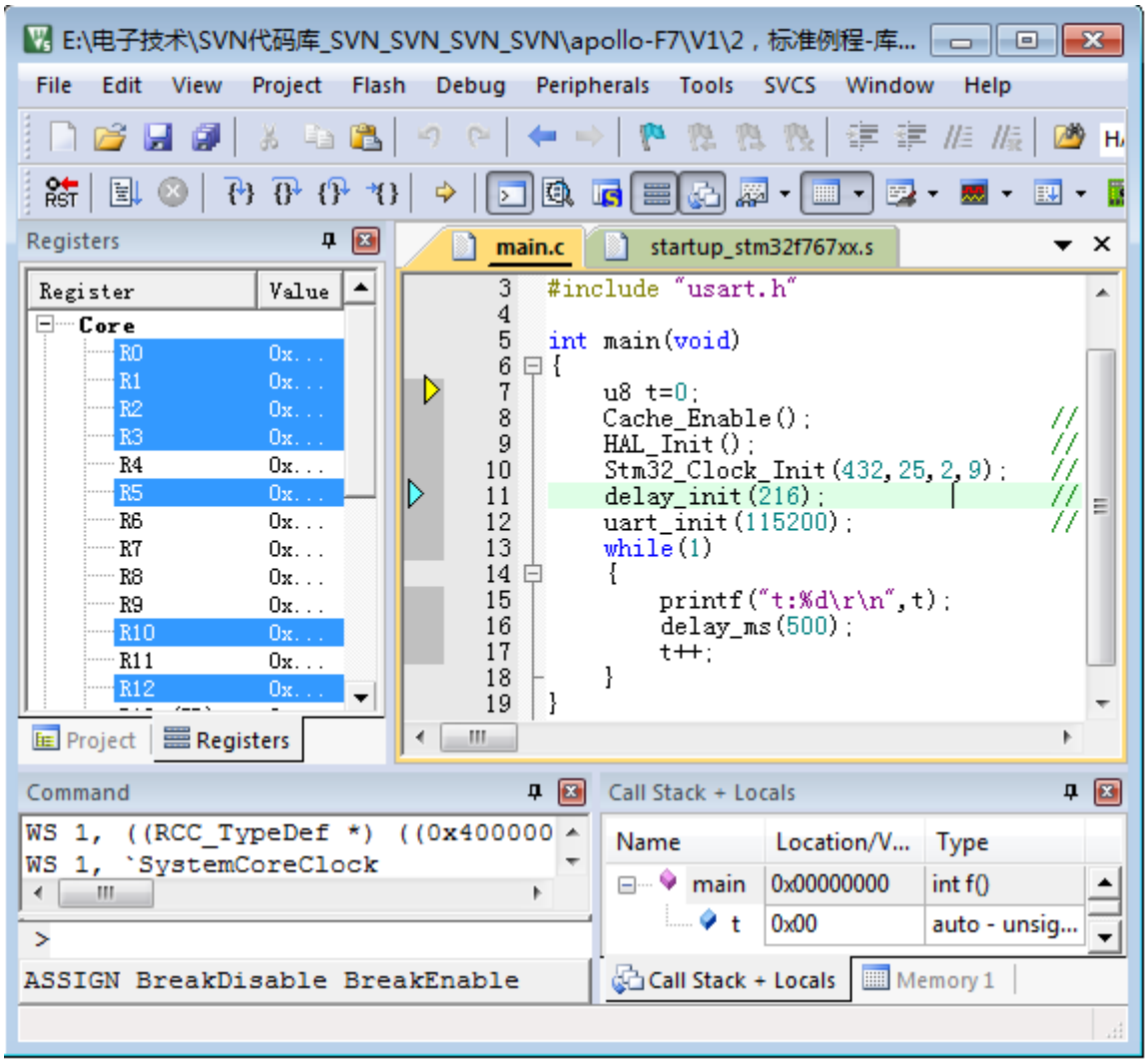


图 3.4.2.1 开始仿真

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。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 3.4.2.2 所示：

图 3.4.2.2 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。

特别注意：串口打印窗口和逻辑分析窗口仅在软件仿真的时候可用，而 MDK5 对 STM32F767 的软件仿真，基本上不支持（故本教程直接没有对软件仿真进行介绍了），所以，基本上这两个窗口用不着。但是对 STM32F1 的软件仿真，MDK5 是支持的，在 F1 开发的时候，可以用到。

这样，我们在上面的仿真界面里面调出：堆栈局部变量窗口。如图 3.4.2.3 所示：

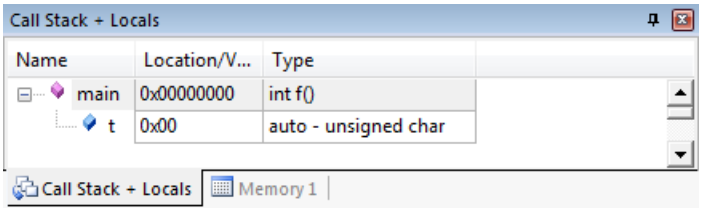


图 3.4.2.3 堆栈局部变量查看窗口

我们把光标放到 main.c 的第 12 行左侧的灰色区域，然后按下鼠标左键，即可放置一个断点（红色的实心点，也可以通过鼠标右键弹出菜单来加入），再次单击则取消。 然后点击  ，执行到该断点处，如图 3.4.2.4 所示

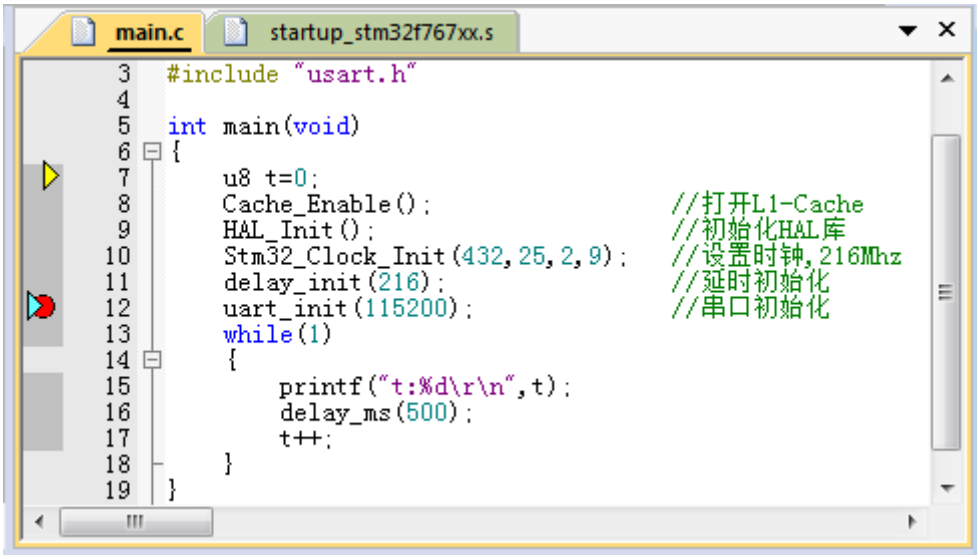


图 3.4.2.4 执行到断点处

现在先不忙着往下执行，点击菜单栏的 Peripherals→System Viewer→USART→USART1。可以看到，有很多外设可以查看，这里我们查看的是串口 1 的情况。如图 3.4.2.5 所示：

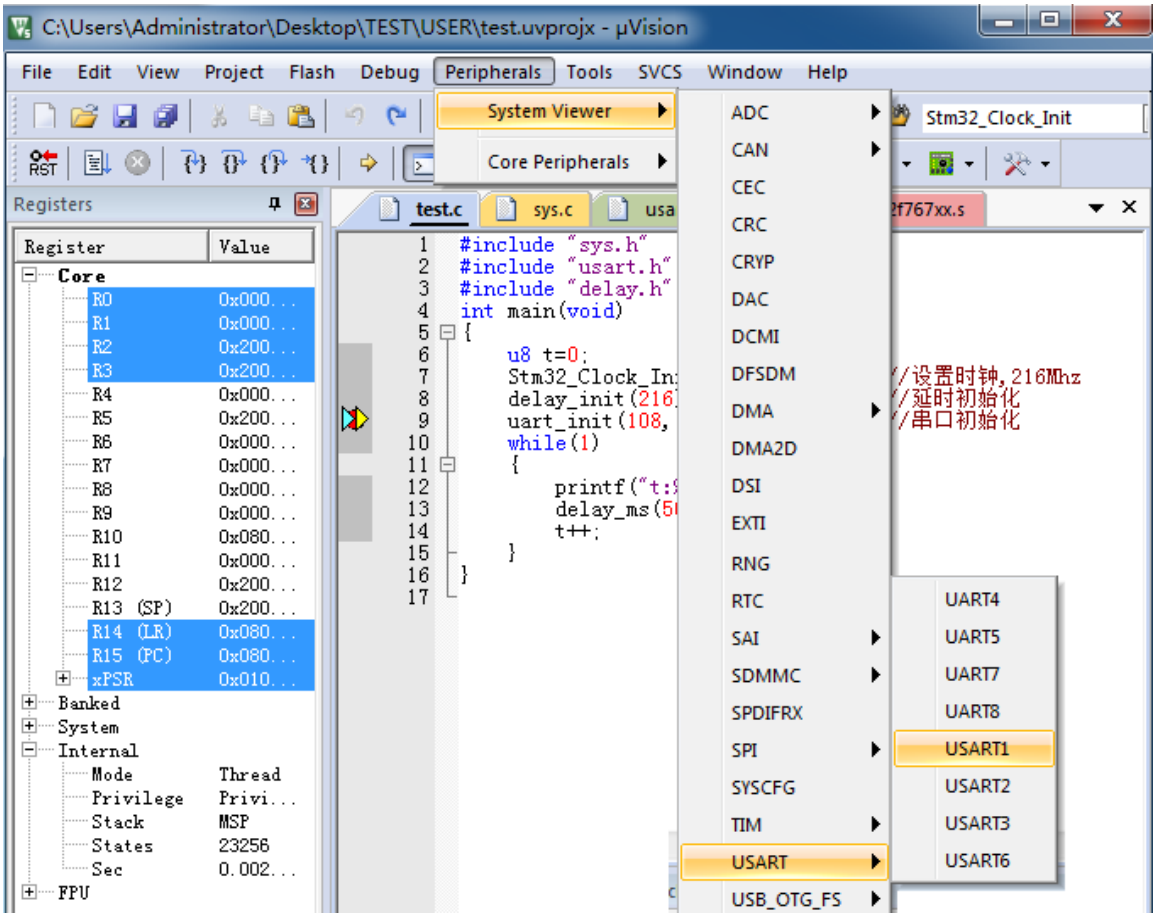
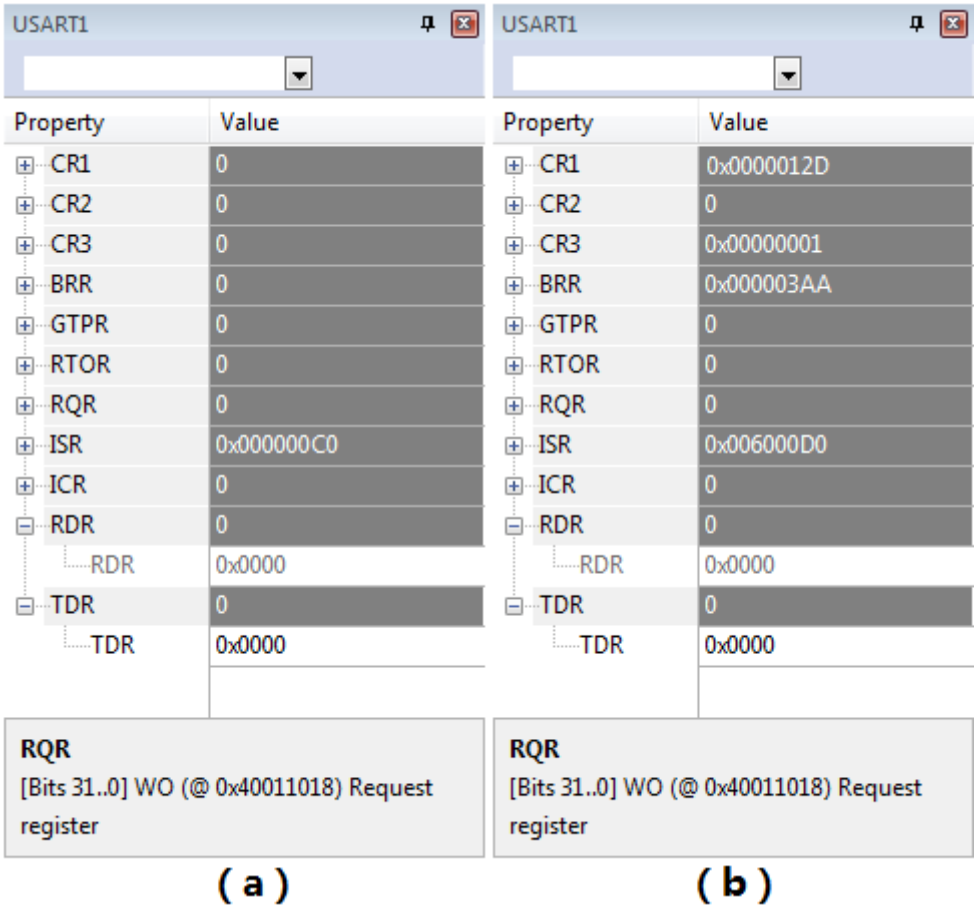


图 3.4.2.5 查看串口 1 相关寄存器

单击 USART1 后会在 IDE 右侧出现一个如图 3.4.2.6（a）所示的界面：



4.2.6 串口 1 各寄存器初始化前后对比

图 3.4.2.6（a）是 STM32 的串口 1 的默认设置状态，从中可以看到所有与串口相关的寄存器全部在这上面表示出来了。我们接着单击一下，执行完串口初始化函数，得到了如图 3.4.2.6（b）所示的串口信息。大家可以对比一下这两个图的区别，就知道在 uart_init(115200);这个函数里面大概执行了哪些操作。

通过图 3.4.2.6（b），我们可以查看串口 1 的各个寄存器设置状态，从而判断我们写的代码是否有问题，只有这里的设置正确了之后，才有可能在硬件上正确的执行。同样这样的方法也可以适用于很多其他外设，这个读者慢慢体会吧！这一方法不论是在排错还是在编写代码的时候，都是非常有用的。

此时，我们先打开串口调试助手（XCOM V2.0，在光盘→6，软件资料→软件→串口调试助手里面）设置好串口号和波特率，然后我们继续单击按钮，一步步执行，此时在堆栈局部变量窗口可以看到 t 的值变化，同时在串口调试助手中，也可看到打印出 t 的值，如图 3.4.2.7 和 4.2.8 所示：

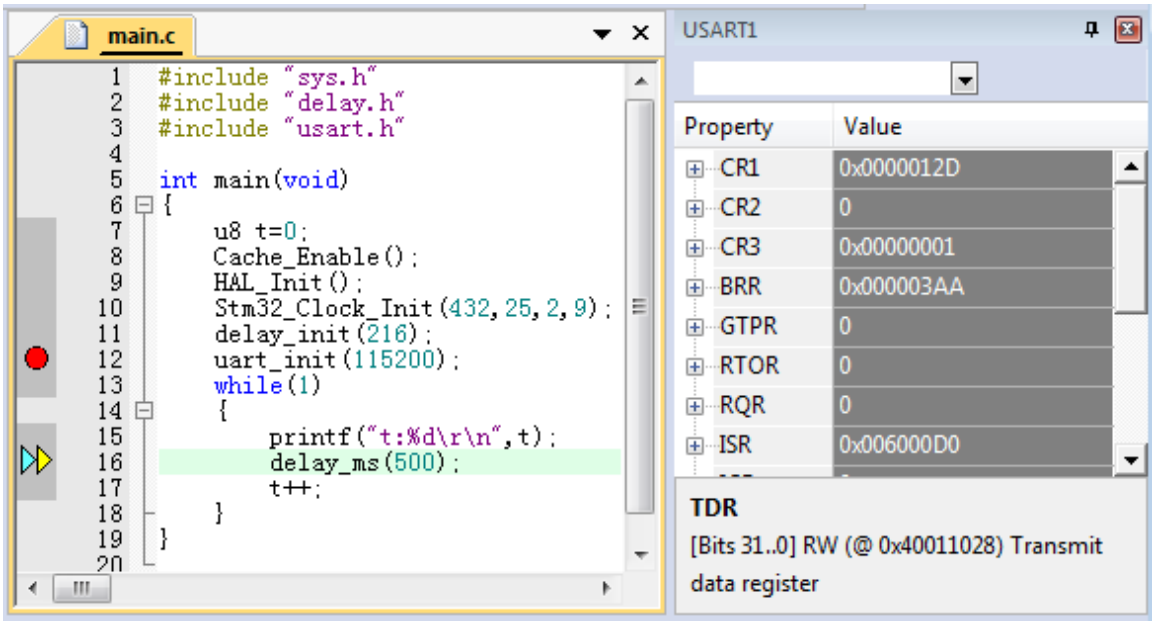


图 3.4.2.7 堆栈局部变量窗口查看 t 的值

图 3.4.2.8 串口调试助手收到的数据

关于 STM32F767 的硬件调试，我们就介绍到这里，这仅仅是一个简单的 demo 演示，在实际使用中，硬件调试更是大有用处，所以大家一定要好好掌握。

3.5 MDK5 使用技巧

通过前面的学习，我们已经了解了如何在 MDK5 里面建立属于自己的工程。下面，我们将向大家介绍 MDK5 软件的一些使用技巧，这些技巧在代码编辑和编写方面会非常有用，希望大家好好掌握，最好实际操作一下，加深印象。

3.5.1 文本美化

文本美化，主要是设置一些关键字、注释、数字等的颜色和字体。前面我们在介绍 MDK5 新建工程的时候看到界面如图 3.2.23 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：

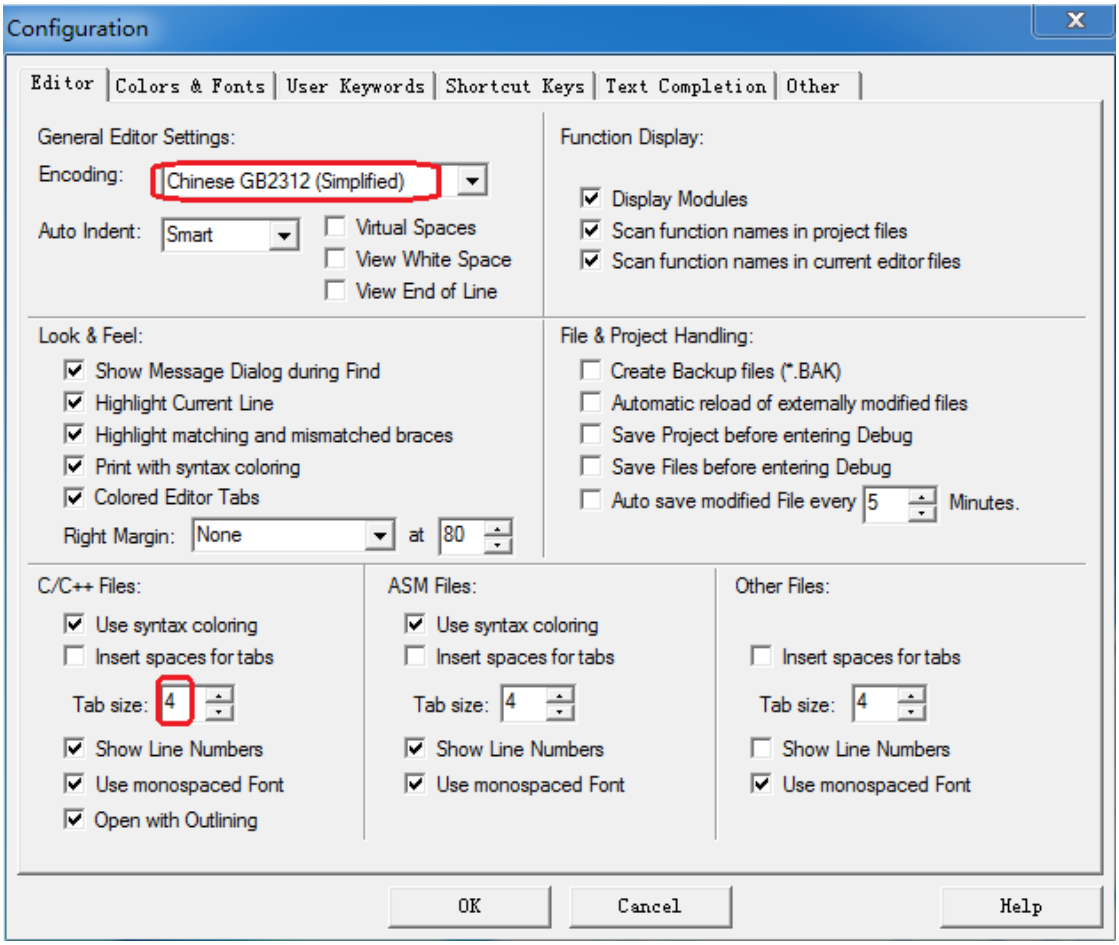


图 3.5.1.1 置对话框

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：

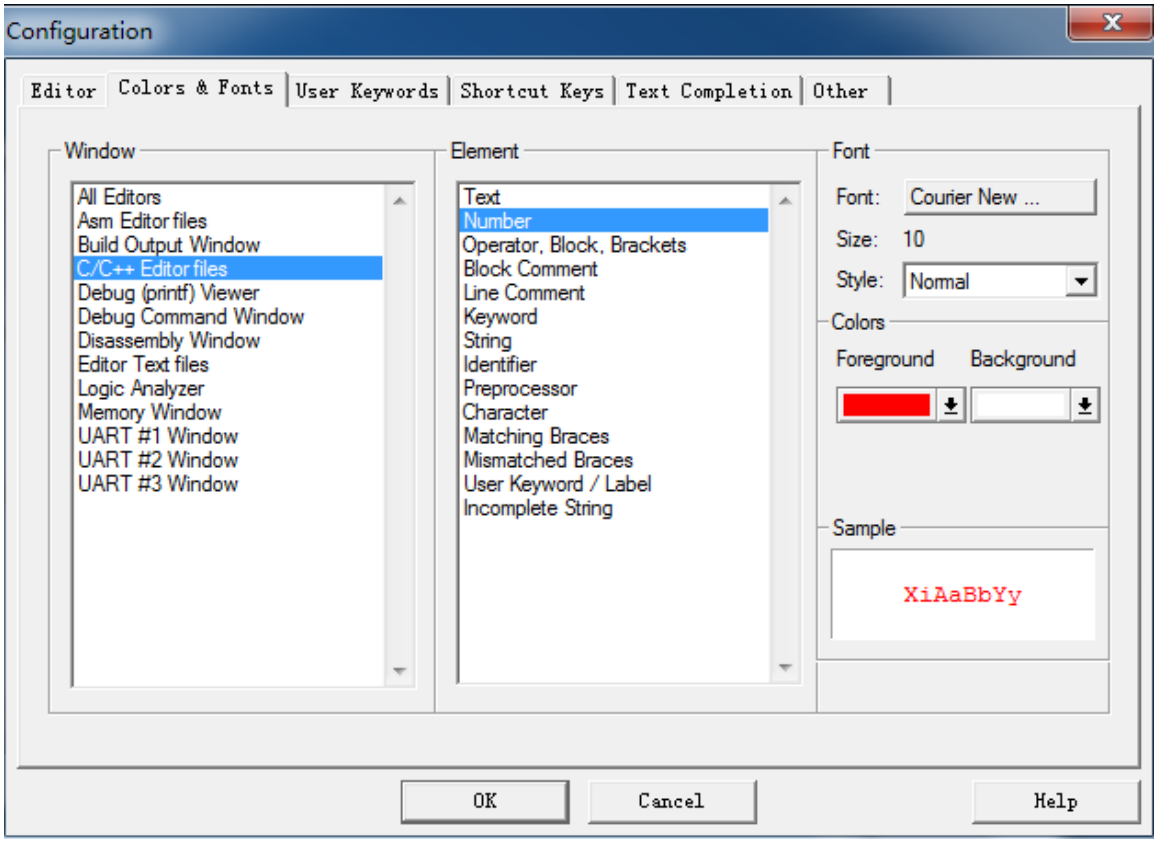


图 3.5.1.2 Colors&Fonts 选项卡

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：

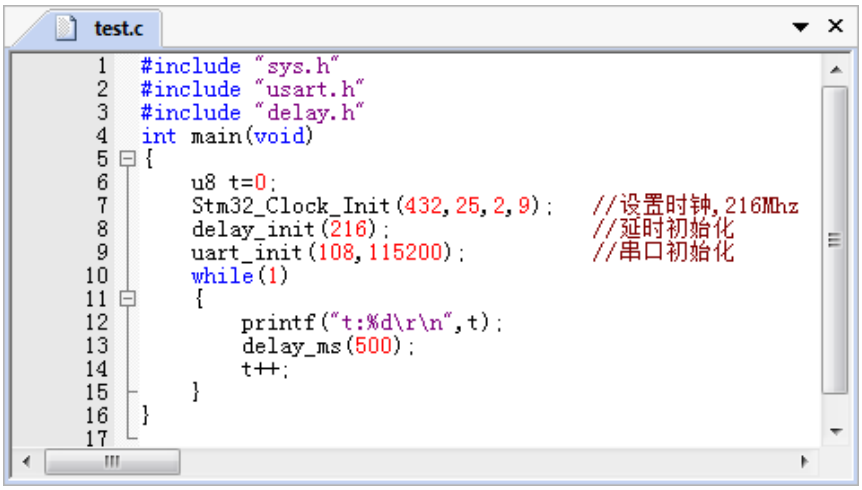


图 3.5.1.3 设置完后显示效果

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：

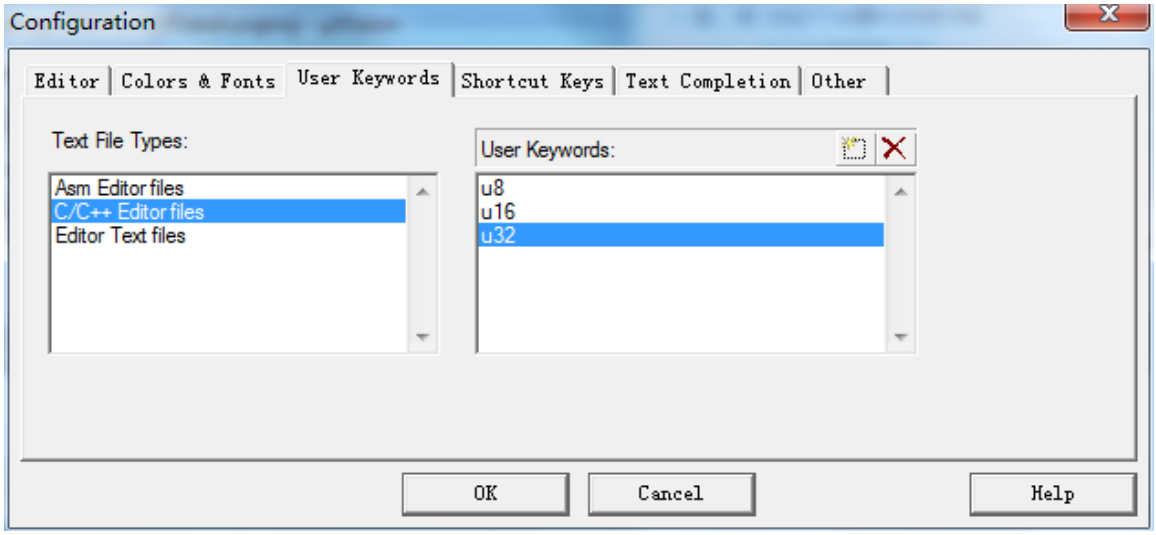


图 3.5.1.4 用户自定义关键字

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：

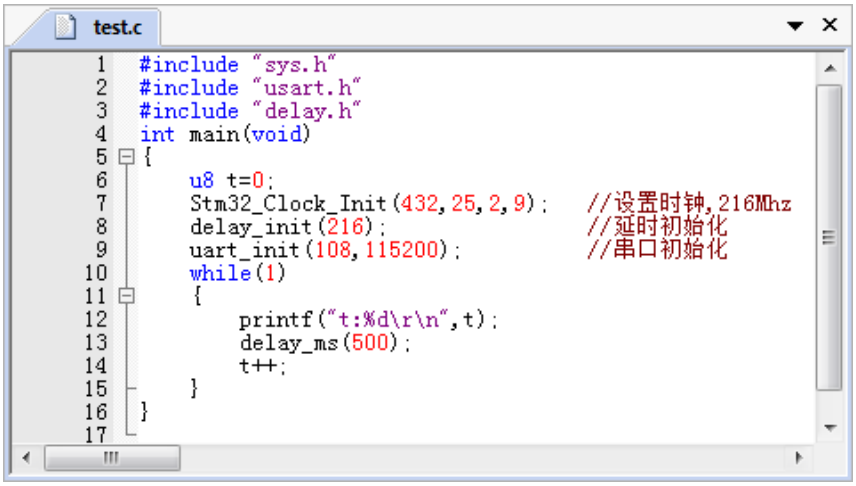


图 3.5.1.5 设置完后显示效果

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：

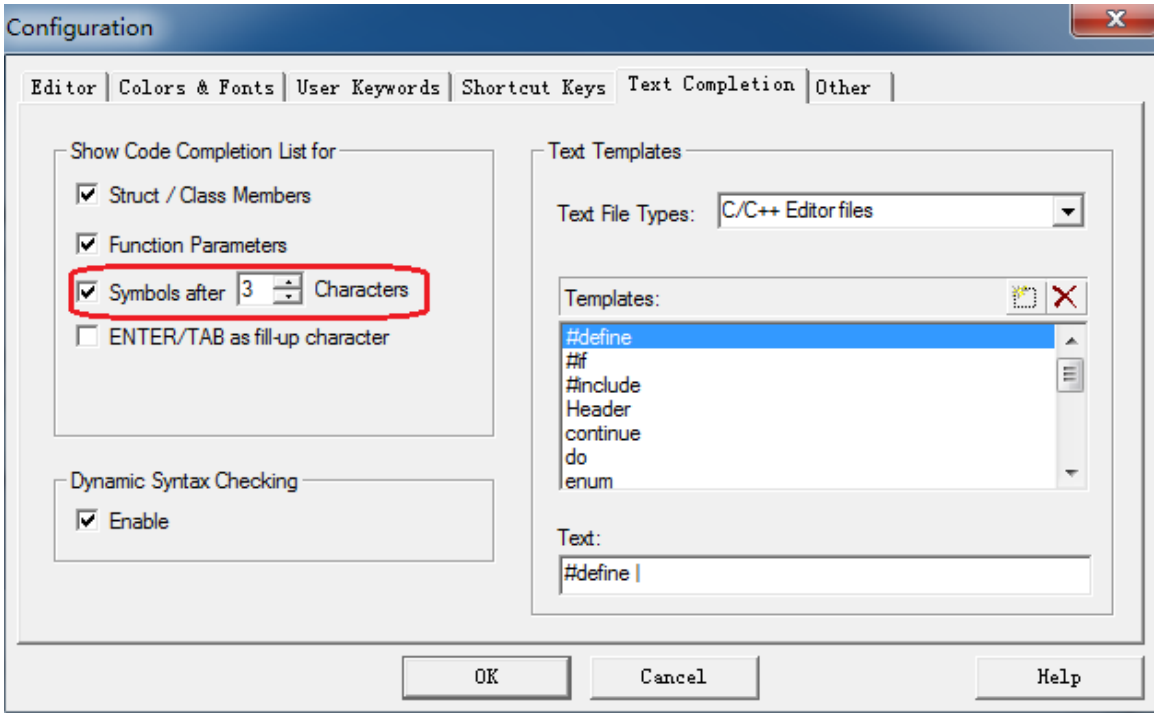


图 3.5.2.1 Text Completion 选项卡设置

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：

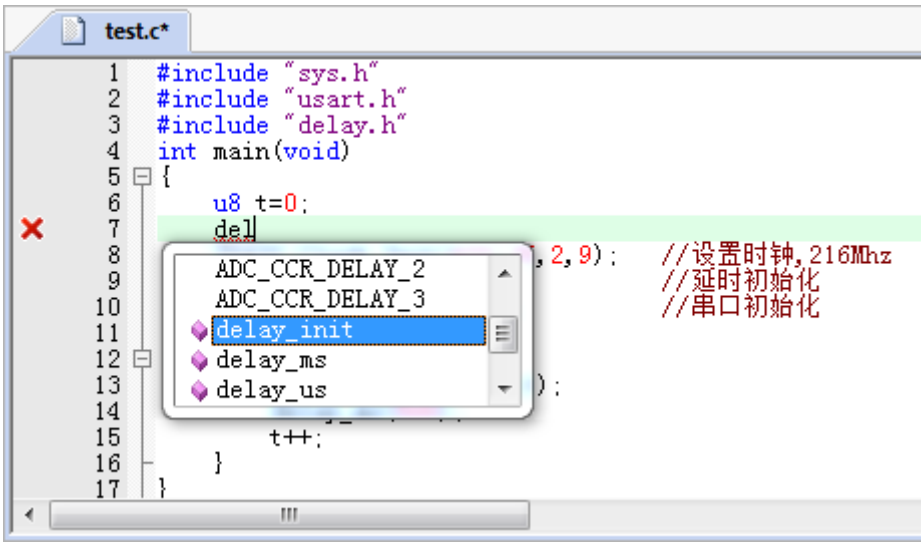


图 3.5.2.2 代码提示

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：

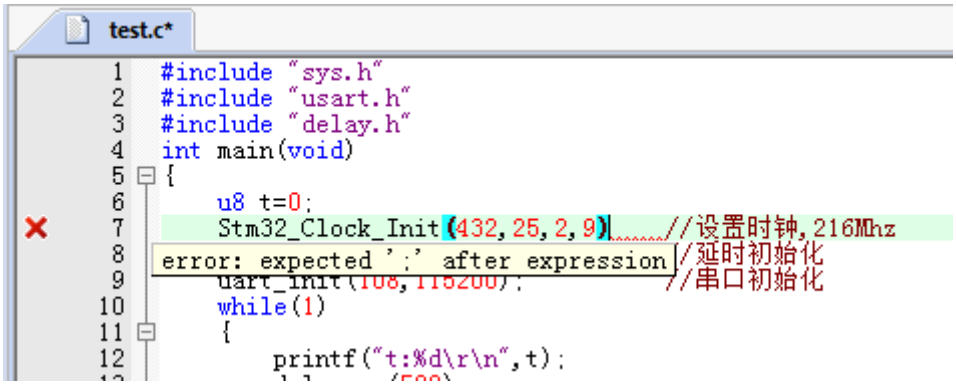


图 3.5.2.3 语法动态检测功能

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：

```
70  void USART1_IRQHandler(void)
71  {
72  u8 res;
73  #if SYSTEM_SUPPORT_OS       //如果SYSTEM_SUPPORT_OS为真，则需要支持OS.
74  OSIntEnter();
75  #endif
76  if(USART1->ISR&(1<<5))//接收到数据
77  {
78  res=USART1->RDR;
79  if((USART_RX_STA&0x8000)==0)//接收未完成
80  {
81  if(USART_RX_STA&0x4000)//接收到了0x0d
82  {
83  if(res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
84  else USART_RX_STA|=0x8000;  //接收完成了
85  }else //还没收到0X0D
86  {
87  if(res==0x0d)USART_RX_STA|=0x4000;
88  else
89  {
90  USART_RX_BUF[USART_RX_STA&0X3FFF]=res;
91  USART_RX_STA++;
92  if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
93  }
94  }
95  }
96  }
97  #if SYSTEM_SUPPORT_OS   //如果SYSTEM_SUPPORT_OS为真，则需要支持OS.
98  OSIntExit();
99  #endif
100 }
```

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;
    #if SYSTEM_SUPPORT_OS        //如果SYSTEM_SUPPORT_OS为真，则需要支持OS.
    OSIntEnter();
    #endif
    if(USART1->ISR&(1<<5))//接收到数据
    {
    res=USART1->RDR;
    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
    }
    }
    }
    }
    #if SYSTEM_SUPPORT_OS    //如果SYSTEM_SUPPORT_OS为真，则需要支持OS.
    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;
#if SYSTEM_SUPPORT_OS        //如果SYSTEM_SUPPORT_OS为真，则需要支持OS.
    OSIntEnter();
#endif
    if(USART1->ISR&(1<<5))//接收到数据
    {
        res=USART1->RDR;
        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
                }
            }
        }
    }
#if SYSTEM_SUPPORT_OS    //如果SYSTEM_SUPPORT_OS为真，则需要支持OS.
    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 ：

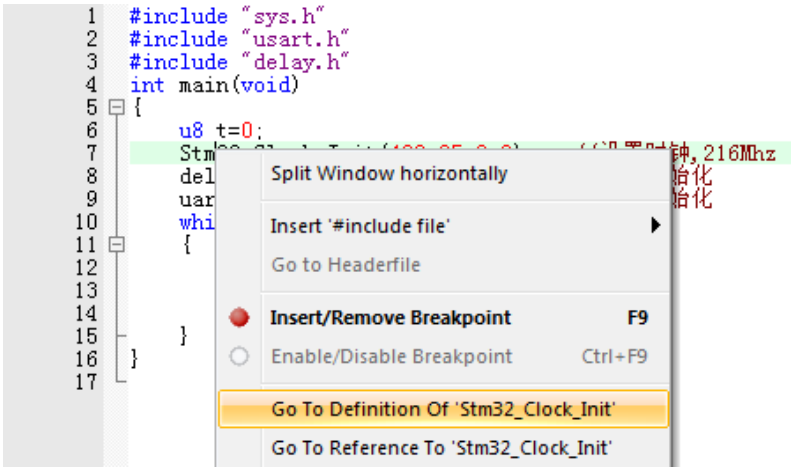


图 3.5.3.4 快速定位

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
    Cache_Enable();               //使能L1 Catch
    Sys_Clock_Set(plln,pllm,pllp,pllq);//设置时钟
    //配置向量表
#ifdef  VECT_TAB_RAM
    MY_NVIC_SetVectorTable(SRAM1_BASE,0x0);
#else
    MY_NVIC_SetVectorTable(FLASH_BASE,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：

```
241  //系统时钟初始化函数
242  //plln:主PLL倍频系数(PLL倍频),取值范围:64~432.
243  //pllm:主PLL和音频PLL分频系数(PLL之前的分频),取值范围:2~63.
244  //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!)
245  //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.
246  void Stm32_Clock_Init(u32 plln,u32 pllm,u32 pllp,u32 pllq)
247  {
248      RCC->CR|=0x00000001;          //设置HISON,开启内部高速RC振荡
249      RCC->CFGR=0x00000000;         //CFGR清零
250      RCC->CR&=0xFEF6FFFF;          //HSEON,CSSON,PLLON清零
251      RCC->PLLCFGR=0x24003010;      //PLLCFGR恢复复位值
252      RCC->CR&=~(1<<18);            //HSEBYP清零,外部晶振不旁路
253      RCC->CIR=0x00000000;          //禁止RCC时钟中断
254      Cache_Enable();               //使能L1 Catch
255      Sys_Clock_Set(plln,pllm,pllp,pllq);//设置时钟
256      //配置向量表
257  #ifdef  VECT_TAB_RAM
258      MY_NVIC_SetVectorTable(SRAM1_BASE,0x0);
259  #else
260      MY_NVIC_SetVectorTable(FLASH_BASE,0x0);
261  #endif
262  }
```

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：

```
241  //系统时钟初始化函数
242  //plln:主PLL倍频系数(PLL倍频),取值范围:64~432.
243  //pllm:主PLL和音频PLL分频系数(PLL之前的分频),取值范围:2~63.
244  //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!)
245  //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.
246  void Stm32_Clock_Init(u32 plln,u32 pllm,u32 pllp,u32 pllq)
247  {
248  //  RCC->CR|=0x00000001;          //设置HISON,开启内部高速RC振荡
249  //  RCC->CFGR=0x00000000;         //CFGR清零
250  //  RCC->CR&=0xFEF6FFFF;          //HSEON,CSSON,PLLON清零
251  //  RCC->PLLCFGR=0x24003010;      //PLLCFGR恢复复位值
252  //  RCC->CR&=~(1<<18);            //HSEBYP清零,外部晶振不旁路
253  //  RCC->CIR=0x00000000;          //禁止RCC时钟中断
254  //  Cache_Enable();               //使能L1 Catch
255  //  Sys_Clock_Set(plln,pllm,pllp,pllq);//设置时钟
256  //  //配置向量表
257  //#ifdef  VECT_TAB_RAM
258  //  MY_NVIC_SetVectorTable(SRAM1_BASE,0x0);
259  //#else
260  //  MY_NVIC_SetVectorTable(FLASH_BASE,0x0);
261  //#endif
262  }
```

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：

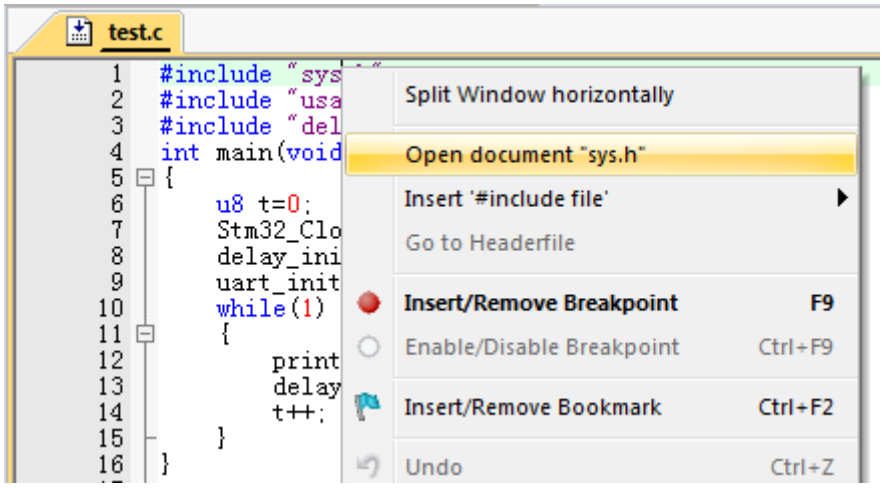


图 3.5.4.1 快速打开头文件

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：

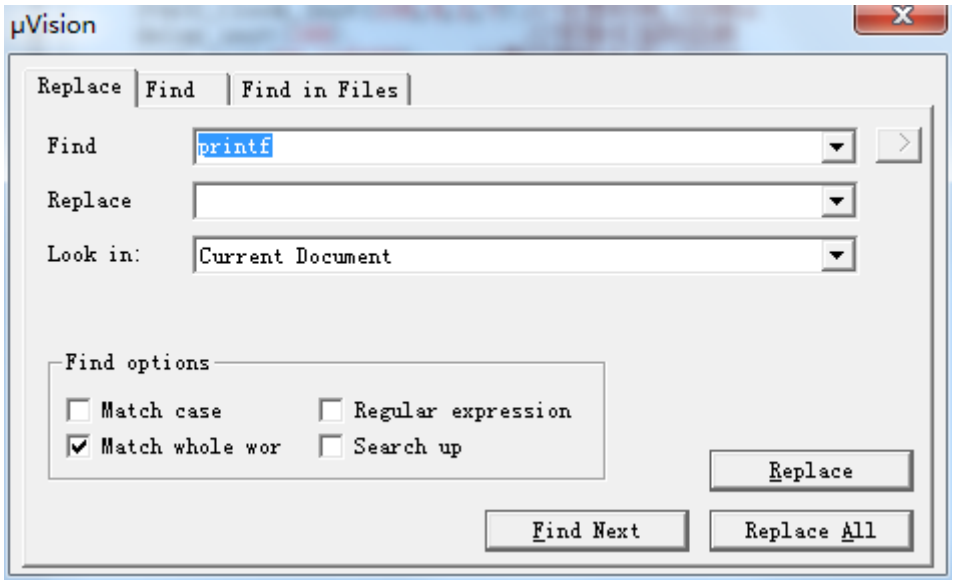


图 3.5.4.2 替换文本

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：

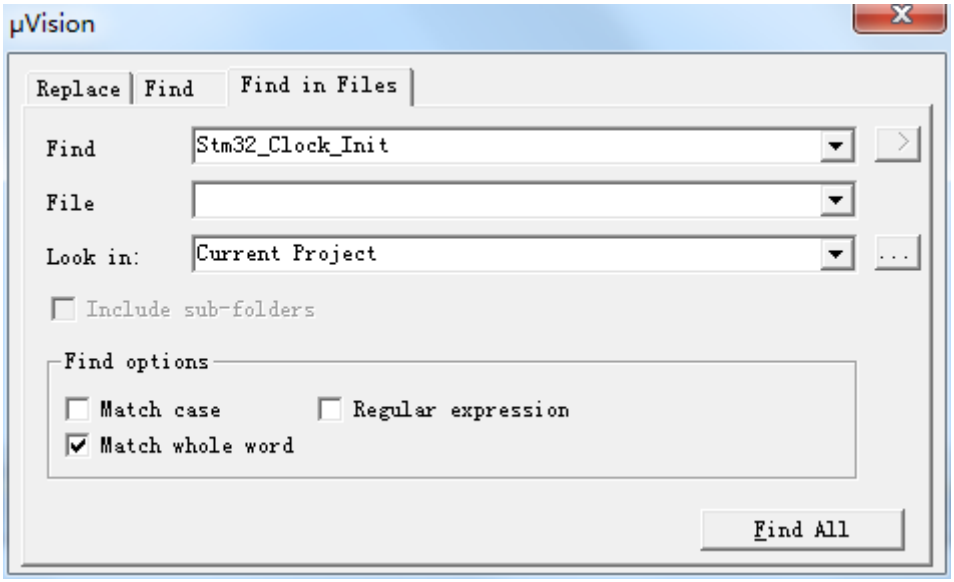


图 3.5.4.3 跨文件查找

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：

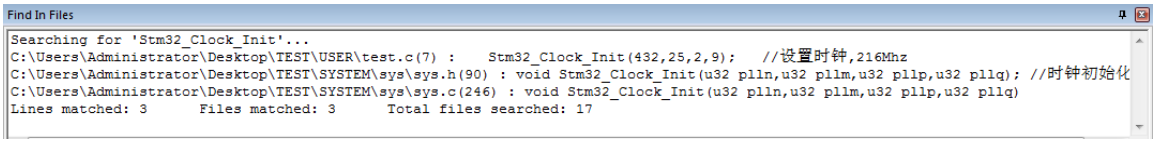


图 3.5.4.4 查找结果

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 STM32F7 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 STM32F7 系统架构
- 4.3 STM32F767 时钟系统
- 4.4 IO 引脚复用器和映射
- 4.5 STM32F7 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
- 4.8 手把手教你入门 STM32CubeMX 图形配置工具

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 STM32F7 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 stm32f7xx_hal_conf.h 这个头文件中会看到这样的语句：

```
#ifdef   HAL_GPIO_MODULE_ENABLED
  #include "stm32f7xx_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 stm32f7xx_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;
  uint32_t Alternate;
```

```
}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 STM32F7 总线架构

STM32F7 的总线架构比 51 单片机就要强大很多了。STM32F7 总线架构的知识可以在《STM32F7XX 中文参考手册》第二章有讲解，这里我们也把这一部分知识抽取出来讲解，是为了大家在学习 STM32F7 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 STM32F7 的系统架构，还需要多看看《STM32F7 中文参考手册》或者在网上搜索其他相关资料学习。

首先我们看看 STM32F7 的总线架构图如下图 4.2.1 所示：

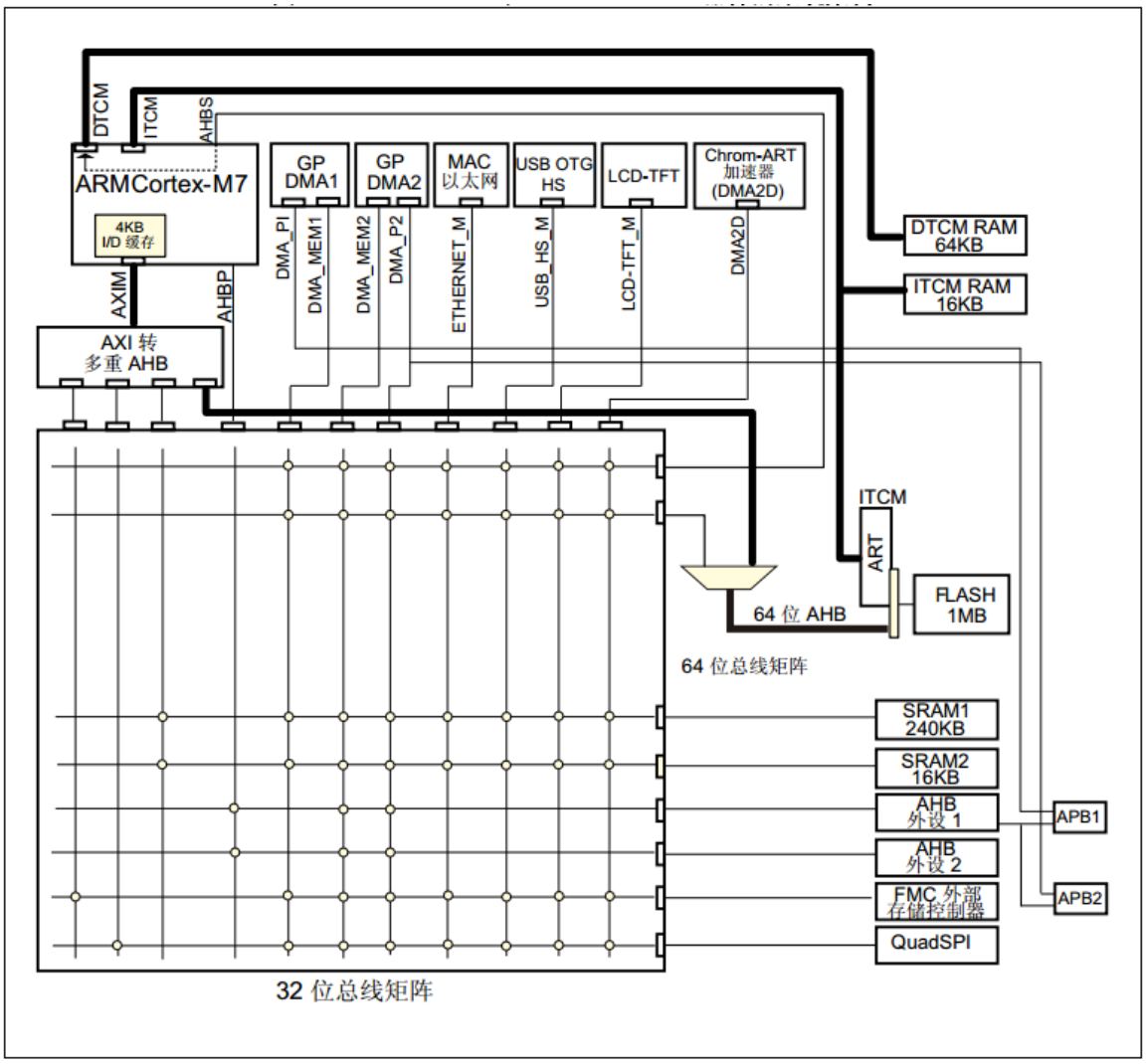


图 4.2.1 STM32F767 系统架构图

主系统架构基于 2 个子系统：

- 一个 AXI 转 multi-AHB 总线桥，用于将 AXI4 协议转换为 AHB-Lite 协议。
  ① 一个连接到内嵌 flash 的 AXI 转 64 位 AHB 总线桥
  ② 三个连接到 AHB 总线矩阵的 AXI 转 32 位 AHB 总线桥
- 一个 multi-AHB 总线矩阵

multi-AHB 总线矩阵将所有主控总线和被控总线互联，它包括：

① 32 位 multi-AHB 总线矩阵
② 64 位 multi-AHB 总线矩阵：它将来自 CPU 的 64 位 AHB 总线（通过 AXI 转 AHB 总线桥）和来自 GP DMA 与外设 DMA（增至 64 位）的 32 位 AHB 总线连接到内部 flash。

multi_AHB 总线矩阵可连接 12 个总线主控制器和 8 个总线从控制器：

- 十二个总线主控制器
  ① 3x32 位 AHB 总线以及 64 位 Cortex-M7 AXI 主控总线通过 AXI-AHB 总线桥分为 4 个总线控制器
  ② 连接到内嵌 flash 的 1x16 位 AHB 总线
  ③ Cortex-M7 AHB 外设总线
  ④ DMA1 存储器总线

⑤ DMA2 存储器总线
⑥ DMA2 外设总线
⑦ 以太网 DMA 总线
⑧ USB OTG HS DMA 总线
⑨ LCD 控制器 DMA 总线
⑩ Chrom-Art 加速器（DMA2D）存储器总线

- 八个总线从控制器
  ① AHB 总线上的内嵌 Flash（用于 Flash 读/写访问，代码执行和数据访问）
  ② Cortex-M7 AHBS 从接口（仅用于 DTCM RAM 的 DMA 数据传输）
  ③ 主 SRAM1（240KB）
  ④ 辅助 SRAM2（16KB）
  ⑤ AHB1 外设（包括 AHB-APB 总线桥和 APB 外设）
  ⑥ AHB2 外设（包括 AHB-APB 总线桥和 APB 外设）
  ⑦ FMC
  ⑧ Quad SPI

下面我们简单讲解一下几个总线的作用。

① multi-AHB 总线矩阵
multi-AHB 总线矩阵用于主控制器之间的访问仲裁管理。仲裁采用循环调度算法。借助该总线矩阵，可以实现主控总线到被控总线的访问，这样即使在多个高速外设同时运行期间，系统也可以实现并发访问和高效运行。

② AHB/APB 总线桥（APB）
借助两个 AHB/APB 总线桥 APB1 和 APB2，可在 AHB 总线与两个 APB 总线之间实现完全同步的连接，从而灵活选择外设频率。

③ CPU AXIM 总线
该总线通过 AXI-AHB 总线桥将带 FPU 的 Cortex-M7 内核的指令总线和数据总线连接到 multi-AHB 总线矩阵。

④ ITCM 总线
Cortex-M7 使用该总线对映射到 ITCM 接口上的内嵌 flash 进行取指和数据访问。但对于 ITCM RAM，该总线只能进行取指操作。

⑤ DTCM 总线
Cortex-M7 使用该总线对 DTCM RAM 进行数据访问，也可以进行取指。

⑥ CPU AHBS 总线
该总线将 Cortex-M7 的 AHB 被控总线连接到总线矩阵。该总线仅用于通用 DMA 和外设 DAM 到 DTCM RAM 上的数据传输。AHBS 上无法访问 ITCM 总线，因此 RAM 不能通过 ITCM 总线进行 DMA 数据传输。Flash 通过 ITCM 接口进行 DMA 传输时，将强制通过 AHB 总线进行所有传输。

⑦ AHB 外设总线
该总线将 Cortex-M7 的 AHB 外设总线连接到总线矩阵。内核使用该总线执行所有针对外设的数据访问。该总线的访问目标是 AHB1 总线上的外设（包括 APB 总线上的外设和 AHB2 总线上的外设）。

⑧ DMA 存储器总线
此总线用于将 DMA 存储器总线主接口连接到总线矩阵。DMA 通过此总线来执行存储

器数据的传入和传出。该总线的访问目标是数据存储器：内部 SRAM1，SRAM2 和 DTCM 以及内部 FLASH 和外部存储器。

⑨ DMA 外设总线

此总线用于将 DMA 外设主总线接口连接到总线矩阵。DMA 通过此总线访问 AHB 外设或执行存储器间的数据传输。该总线的访问目标是 AHB 和 APB 总线上的外设以及数据存储器：内部 SRAM1，SRAM2 和 DTCM 以及内部 FLASH 和外部存储器。

⑩ 以太网 DMA 总线

此总线用于将以太网 DMA 主接口连接到总线矩阵。以太网 DMA 通过此总线向存储器存取数据。该总线的访问目标是数据存储器：内部 SRAM1，SRAM2 和 DTCM，内部 Flash 和外部存储器。

⑪ USB OTG HS DMA 总线

此总线用于将 USB OTG HS DMA 主接口连接到总线矩阵。USB OTG DMA 通过此总线向存储器加载/存储数据。该总线的访问目标是数据存储器：内部 SRAM1，SRAM2 和 DTCM，内部 Flash 和外部存储器。

⑫ LCD-TFT 控制器 DMA 总线

此总线用于将 LCD 控制器 DMA 主接口连接到总线矩阵。LCD-TFT DMA 通过此总线向存储器加载/存储数据。该总线的访问目标是数据存储器：内部 SRAM1，SRAM2 和 DTCM，外部 Flash 和外部存储器。

⑬ DMA2D 总线

此总线用于将 DMA2D 主接口连接到总线矩阵。DMA2D 图形加速器通过此总线向存储器加载/存储数据。该总线的访问目标是数据存储器：内部 SRAM1，SRAM2 和 DTCM，外部 Flash 和外部存储器。

对于系统架构的知识，在刚开始学习 STM32F7 的时候只需要一个大概的了解。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 STM32F7 时钟系统

STM32F7 时钟系统的知识在《STM32F7 中文参考手册》第五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 STM32F7 时钟树概述
- 4.3.2 STM32F7 时钟初始化配置
- 4.3.3 STM32F7 时钟使能和配置

4.3.1 STM32F7 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。 STM32F7 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 STM32F7 的时钟系统图：

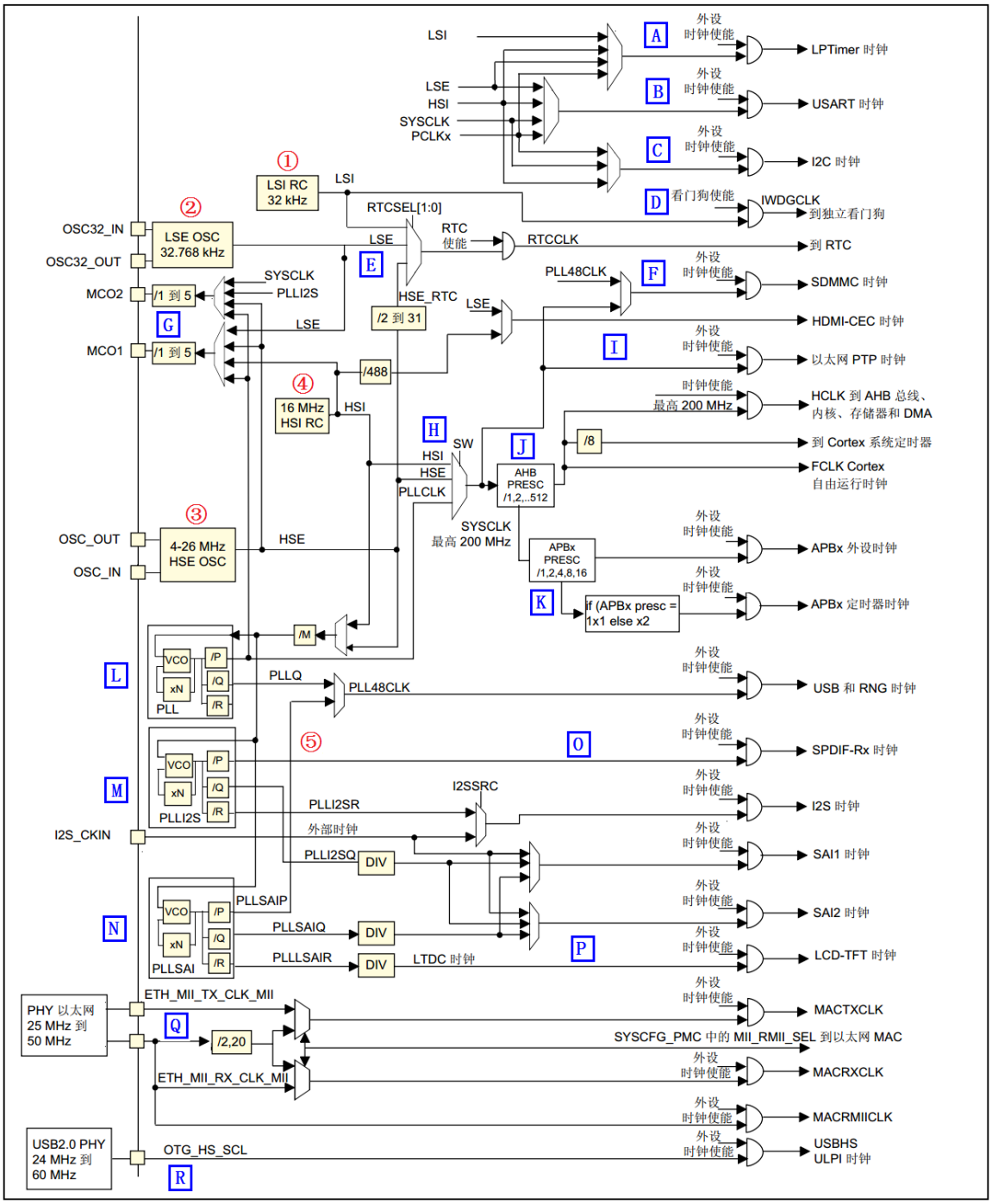


图 4.3.1.1 STM32F7 时钟系统图

在 STM32F7 中，有 5 个最重要的时钟源，为 HSI、HSE、LSI、LSE、PLL。其中 PLL 实际是分为三个时钟源，分别为主 PLL 和 I2S 部分专用 PLLI2S 和 SAI 部分专用 PLLSAI。从时钟频率来分可以分为高速时钟源和低速时钟源，在这 5 个中 HSI，HSE 以及 PLL 是高速时钟，LSI 和 LSE 是低速时钟。从来源可分为外部时钟源和内部时钟源，外部时钟源就是从外部通过接晶振的方式获取时钟源，其中 HSE 和 LSE 是外部时钟源，其他的是内部时钟源。下面我们看看 STM32F7 的这 5 个时钟源，我们讲解顺序是按图中红圈标示的顺序：

①、LSI 是低速内部时钟，RC 振荡器，频率为 32kHz 左右。LSI 主要可以作为 IWDG 独立看门狗时钟，LPTimer 低功耗定时器时钟以及 RTC 时钟。

②、LSE 是低速外部时钟，接频率为 32.768kHz 的石英晶体。这个主要是 RTC 的时钟源。

③、HSE 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz~26MHz。阿波罗 STM32F7 开发板接的是 25MHz 外部晶振。HSE 可以直接做为系统时钟或者 PLL 输入时钟，同时它经过 2~31 分频后也可以作为 RTC 时钟。

④、HSI 是高速内部时钟，RC 振荡器，频率为 16MHz。可以直接作为系统时钟或者用作 PLL 输入，同时它经过 488 分频之后也可以作为 HDMI-CEC 时钟。

⑤、PLL 为锁相环倍频输出。STM32F7 有三个 PLL:

1） 主 PLL(PLL)由 HSE 或者 HSI 提供时钟信号，并具有两个不同的输出时钟。
第一个输出 PLLP 用于生成高速的系统时钟（最高 216MHz）
第二个输出 PLLQ 为 48M 时钟，用于 USB OTG FS 时钟，随机数发生器的时钟和 SDMMC 时钟。

2） 第一个专用 PLL(PLLI2S)用于生成精确时钟，在 I2S、SAI 和 SPDIFRX 上实现高品质音频性能。其中，N 是用于 PLLI2S vco 的倍频系数，其取值范围是：50~432；R 是 I2S 时钟的分频系数，其取值范围是：2~7；Q 是 SAI 时钟分频系数，其取值范围是：2~15；P 没用到。

3） 第二个专用 PLL(PLLSAI)用于为 SAI 接口生成时钟，生成 LCD-TFT 时钟以及可供 USB OTG FS、SDMMC 和 RNG 选择的 48MHz 时钟。其中，N 是用于 PLLSAI vco 的倍频系数，其取值范围是：50~432；Q 是 SAI 时钟分频系数，其取值范围是：2~15；R 是 LTDC 时钟的分频系数，其取值范围是：2~7；P 没用到。

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

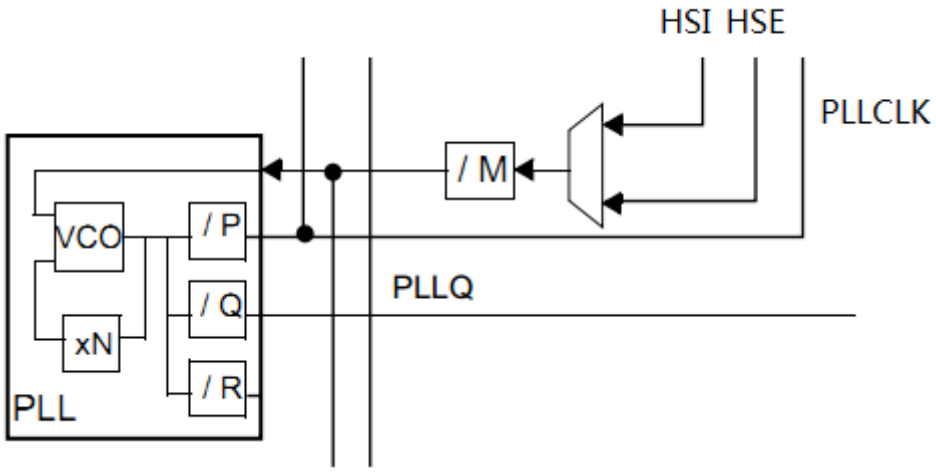


图 4.3.1.2 STM32F7 主 PLL 时钟图

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

例如我们的外部晶振选择 25MHz。同时我们设置相应的分频器 M=25，倍频器倍频系数 N=432，分频器分频系数 P=2，那么主 PLL 生成的第一个输出高速时钟 PLLP 为：

PLL=25MHz * N/ (M*P)=25MHz* 432/(25*2) = 216MHz

如果我们选择HSE为PLL时钟源，同时SYSCLK时钟源为PLL，那么SYSCLK时钟为216MHz。这对于我们后面的实验都是采用这样的配置。

上面我们简要概括了 STM32F7 的时钟源，那么这 5 个时钟源是怎么给各个外设以及系统提供时钟的呢？这里我们选择一些比较常用的时钟知识来讲解。

图 4.3.1.1 中我们用 A~R 标示我们要讲解的地方。

A. 这是低功耗定时器 LPTimer 时钟，从图中可以看出，LPTimer 有四个时钟源可以选择，分别为 LSI、HSI、LSE 和 PCLKx，默认情况下 LPTimer 选用 PCLKx 作为时钟源。

B. 这里是 USART 时钟源。从图中可以看出，USART 时钟源可选为 LSE、HSI、SYSCLK 以及 PCLKx，默认情况下 USART 选用 PCLKx 作为时钟源。

C. 这里是硬件 I2C 时钟源，从图上可以看出，I2C 可选时钟源为 HSI、SYSCLK 以及 PCLKx。默认情况下 I2C 选用 PCLKx 作为时钟源。

D. 这是 STM32F7 独立看门狗 IWDG 时钟，来源为 LSI。

E. 这里是 RTC 时钟源，可选 LSI、LSE 和 HSE 的 2~31 分频。

F. 这是SDMMC时钟源，来源为系统时钟SYSCLK或者PLL48CLK，其中PLL48CLK来源为 PLLQ 或者 PLLSAIP。

G. 这是STM32F7输出时钟MCO1和MCO2。MCO1是向芯片的PA8引脚输出时钟。它有四个时钟来源分别为：HSI,LSE,HSE 和 PLL 时钟，MCO1 时钟源经过 1~5 分频后向 PA8 引脚输出时钟。MCO2 是向芯片的 PC9 输出时钟，它同样有四个时钟来源分别为：HSE,PLL，SYSCLK 以及 PLLI2S 时钟，MCO2 时钟源同样经过 1~5 分频后向 PC9 引脚输出时钟。

H. 这是系统时钟 SYSCLK 时钟源，可选 HSI、HSE 和 PLLCLK。HSI 是内部 16MHz 时钟精度不够，HSE 是外部晶振产生时钟频率较低，大部分情况下系统都会选择 PLLCLK 作为系统时钟。

I. 这是以太网 PTP 时钟，来源为系统时钟 SYSCLK。

J. 这是 AHB 总线预分频器，分频系数为 2^N(N=0~9)。系统时钟 SYSCLK 经过 AHB 预分频器之后产生 AHB 总线时钟 HCLK。

K. 这是 APBx 预分频器（分频系数可选 1,2,4,8,16），HCLK（AHB 总线时钟）经过APBx预分频器之后，产生PCLKx。这里大家还要注意，APBx定时器时钟是PCLKx经过倍频后得来，倍频系数为 1 或者 2，如果 APBx 预分频系数等于 1，那么这里的倍频系数为 1，否则倍频系数为 2。

L~N. 这是 PLL 时钟。L 为主 PLL 时钟，M 为专用 PLL 时钟 PLLI2S，N 为专用 PLL 时钟 PLLSAI。主 PLL 主要用来产生 PLL 时钟作为系统时钟，同时 PLL48CLK 时钟也可以选择 PLLQ 或者 PLLSAIP。PLLI2S 主要用来为 I2S、SAI 和 SPDIFRX 产生精确时钟。PLLSAIP 则为 SAI 接口生成时钟，生成 LCD-TFT 时钟以及可供 USB OTG FS、SDMMC 和 RNG 选择的 48MHz 时钟 PLL48CLK。

O. 这是 SPDIFRX 时钟，由 PLLI2SP 提供。

P. 这是 LCD-TFT 时钟，由 PLLSAIP 提供。

Q. 这是 STM32F7 内部以太网 MAC 时钟的来源。对于 MII 接口来说，必须向外部 PHY 芯片提供 25Mhz 的时钟，这个时钟，可以由 PHY 芯片外接晶振，或者使用 STM32F7 的 MCO 输出来提供。然后，PHY 芯片再给 STM32F7 提供 ETH_MII_TX_CLK 和 ETH_MII_RX_CLK 时钟。对于 RMII 接口来说，外部必须提供 50Mhz 的时钟驱动 PHY 和 STM32F7 的 ETH_RMII_REF_CLK，这个 50Mhz

时钟可以来自 PHY、有源晶振或者 STM32F7 的 MCO。我们的开发板使用的是 RMII 接口，使用 PHY 芯片提供 50Mhz 时钟驱动 STM32F7 的 ETH_RMII_REF_CLK。

R. 这里是指外部 PHY 提供的 USB OTG HS（60MHZ）时钟。

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

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

4.3.2 STM32F7 时钟系统配置

上一小节我们对 STM32F7 时钟树进行了详细讲解，接下来我们来讲解通过 STM32F7 的 HAL 库进行 STM32F7 时钟系统配置步骤。实际上，STM32F7 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 STM32F7 时钟系统有更加清晰的理解。我们将在 4.8 小节讲解图形化配置工具 STM32CubeMX，大家可以对比参考学习。

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)
{
  /* FPU 设置----------------------------------------------------------*/
  #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
    SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and CP11 Full Access */
  #endif
  /*  复位 RCC 时钟配置为默认配置----------*/
  RCC->CR |= (uint32_t)0x00000001;//打开 HSION 位
  RCC->CFGR = 0x00000000;//复位 CFGR 寄存器
  RCC->CR &= (uint32_t)0xFEF6FFFF;//复位 HSEON, CSSON and PLLON 位
  RCC->PLLCFGR = 0x24003010; //复位寄存器 PLLCFGR
  RCC->CR &= (uint32_t)0xFFFBFFFF;//复位 HSEBYP 位
  RCC->CIR = 0x00000000;//关闭所有中断

  /*  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;
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;
#endif
}
```

从上面代码可以看出，SystemInit 主要做了如下三个方面工作：

1） FPU 设置

2） 复位 RCC 时钟配置为默认复位值（默认开启 HSI）

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：

```
//时钟设置函数
// VCO 频率 Fvco=Fs*(plln/pllm);
//系统时钟频率 Fsys=Fvco/pllp=Fs*(plln/(pllm*pllp));
// USB,SDIO,RNG 等的时钟频率 Fusb=Fvco/pllq=Fs*(plln/(pllm*pllq));

//Fs:PLL 输入时钟频率,可以是 HSI,HSE 等.
//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.

//外部晶振为 25M 的时候,推荐值:plln=432,pllm=25,pllp=2,pllq=9.
//得到:Fvco=25*(432/25)=432Mhz
//        Fsys=432/2=316Mhz
//        Fusb=432/9=48Mhz
//返回值:0,成功;1,失败
void Stm32_Clock_Init(u32 plln,u32 pllm,u32 pllp,u32 pllq)
{
    HAL_StatusTypeDef ret = HAL_OK;
    RCC_OscInitTypeDef RCC_OscInitStructure;
    RCC_ClkInitTypeDef RCC_ClkInitStructure;

    __HAL_RCC_PWR_CLK_ENABLE(); //使能 PWR 时钟
    __HAL_PWR_VOLTAGESCALING_CONFIG(
                PWR_REGULATOR_VOLTAGE_SCALE1);//设置调压器输出电压级别
    RCC_OscInitStructure.OscillatorType=RCC_OSCILLATORTYPE_HSE; //时钟源为 HSE
    RCC_OscInitStructure.HSEState=RCC_HSE_ON;                    //打开 HSE
    RCC_OscInitStructure.PLL.PLLState=RCC_PLL_ON;
    RCC_OscInitStructure.PLL.PLLSource=RCC_PLLSOURCE_HSE;
    RCC_OscInitStructure.PLL.PLLM=pllm;
    RCC_OscInitStructure.PLL.PLLN=plln;
    RCC_OscInitStructure.PLL.PLLP=pllp;
    RCC_OscInitStructure.PLL.PLLQ=pllq;
    ret=HAL_RCC_OscConfig(&RCC_OscInitStructure);
    if(ret!=HAL_OK) while(1);

    ret=HAL_PWREx_EnableOverDrive(); //开启 Over-Driver 功能
    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);
    RCC_ClkInitStructure.SYSCLKSource=RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStructure.AHBCLKDivider=RCC_SYSCLK_DIV1;
    RCC_ClkInitStructure.APB1CLKDivider=RCC_HCLK_DIV4;
    RCC_ClkInitStructure.APB2CLKDivider=RCC_HCLK_DIV2;

    ret=HAL_RCC_ClockConfig(&RCC_ClkInitStructure,FLASH_LATENCY_7);
    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。我们首先来看看使用 HAL 库配置 STM32F7 时钟系统的一般步骤：

1） 使能 PWR 时钟：调用函数__HAL_RCC_PWR_CLK_ENABLE()。

2） 设置调压器输出电压级别：调用函数__HAL_PWR_VOLTAGESCALING_CONFIG()。

3） 选择是否开启 Over-Driver 功能：调用函数 HAL_PWREx_EnableOverDrive()。

4） 配置时钟源相关参数：调用函数 HAL_RCC_OscConfig()。

5） 配置系统时钟源以及 AHB,APB1 和 APB2 的分频系数：调用函数 HAL_RCC_ClockConfig()。

步骤 2 和 3，具有一定的关联性，我们放在后面讲解。对于步骤 1 之所以要使能 PWR 时钟，是因为后面的步骤设置调节器输出电压级别以及开启 Over-Driver 功能都是电源控制相关配置，所以必须开启 PWR 时钟。接下来我们先着重讲解步骤 4 和步骤 5 的内容，这也是时钟系统配置的关键步骤。

对于步骤 4，使用 HAL 来配置时钟源相关参数，我们调用的函数为 HAL_RCC_OscConfig()，该函数在 HAL 库关键头文件 stm32f7xx_hal_rcc.h 中声明，在文件 stm32f7xx_hal_rcc.c 中定义。首先我们来看看该函数声明：

```
__weak HAL_StatusTypeDef    HAL_RCC_OscConfig(RCC_OscInitTypeDef *RCC_OscInitStruct);
```

该函数只有一个入口参数，就是结构体 RCC_OscInitTypeDef 类型指针。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 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 PLLM;       //PLL 分频系数 M
  uint32_t PLLN;       //PLL 倍频系数 N
  uint32_t PLLP;       //PLL 分频系数 P
  uint32_t PLLQ;       //PLL 分频系数 Q
}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.PLL.PLLState=RCC_PLL_ON;               //打开 PLL
    RCC_OscInitStructure.PLL.PLLSource=RCC_PLLSOURCE_HSE;//PLL 时钟源为 HSE
    RCC_OscInitStructure.PLL.PLLM=pllm;
    RCC_OscInitStructure.PLL.PLLN=plln;
    RCC_OscInitStructure.PLL.PLLP=pllp;
    RCC_OscInitStructure.PLL.PLLQ=pllq;
    ret=HAL_RCC_OscConfig(&RCC_OscInitStructure);
```

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 4 个入口参数直接设置作为 PLL 的参数 M,N,P 和 Q 的值，这样就达到了设置 PLL 时钟源相关参数的目的。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，也就是我们前面提到的步骤 5。

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跟步骤 2 和步骤 3 一起讲解。

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);
RCC_ClkInitStructure.SYSCLKSource=RCC_SYSCLKSOURCE_PLLCLK;//系统时钟源 PLL
```

```
RCC_ClkInitStructure.AHBCLKDivider=RCC_SYSCLK_DIV1;//AHB 分频系数为 1
RCC_ClkInitStructure.APB1CLKDivider=RCC_HCLK_DIV4; //APB1 分频系数为 4
RCC_ClkInitStructure.APB2CLKDivider=RCC_HCLK_DIV2; //APB2 分频系数为 2
ret=HAL_RCC_ClockConfig(&RCC_ClkInitStructure,FLASH_LATENCY_7);
```

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 4。
第五个参数 APB2CLKDivider 配置 APB2 分频系数为 2。

根据我们在主函数中调用 Stm32_Clock_Init(436,25,2,9)时候设置的入口参数值，我们可以计算出，PLL 时钟为 PLLCLK=HSE*N/M*P=25MHz*436/(25*2)=216MHz，同时我们选择系统时钟源为 PLL，所以系统时钟 SYSCLK=216MHz。AHB 分频系数为 1，故其频率为 HCLK=SYSCLK/1=216MHz。APB1 分频系数为 4，故其频率为 PCLK1=HCLK/4=54MHz。APB2 分频系数为 2，故其频率为 PCLK2=HCLK/2=216/2=108MHz。最后我们总结一下通过调用函数 Stm32_Clock_Init(432,25,2,9)之后的关键时钟频率值：

```
SYSCLK(系统时钟)                    =216MHz
PLL 主时钟                          =216MHz
AHB 总线时钟（HCLK=SYSCLK/1）       =216MHz
APB1 总线时钟（PCLK1=HCLK/4）       =54MHz
APB2 总线时钟（PCLK2=HCLK/2）       =108MHz
```

最后我们来看看步骤 2，步骤 3 以及步骤 5 中函数 HAL_RCC_ClockConfig 第二个入口参数 FLatency 的含义。这里我们不想讲解得太复杂，大家只需要知道调压器输出电压级别 VOS, Over-Driver 功能开启以及 FLASH 的延迟 Latency 三个参数，在我们芯片电源电压和 HCLK 固定之后，他们三个参数也是固定的。首先我们来看看调压器输出电压级别 VOS，它是由 PWR 控制寄存器 CR 的位 15:14 来确定的：

位 15:14 VOS[1:0]

00:保留（默认模式 3 选中）

01:级别 3：HCLK 最大频率 144MHz

10:级别 2: HCLK 最大频率 168MHz，通过开启 Over-drive 模式可以达到 180MHz

11:级别 1：HCLK 最大频率 180MHz，通过开启 Over-drive 模式可以达到 216MHz。

所以如果我们要配置 HCLK 时钟为 216MHz，也就是在 AHB 的分频系数为 1 的情况下需要系统时钟为 216MHz，那么我们必须配置调压器输出电压级别 VOS 为级别 1，同时开启 Over-drive 功能。所以函数 Stm32_Clock_Init 中步骤 3 和步骤 4 源码如下：

```
//步骤 3，设置调压器输出电压级别 1
__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
ret=HAL_PWREx_EnableOverDrive(); //开启 Over-Driver 功能
```

配置好调压器输出电压级别 VOS 和 Over-drive 功能之后，如果需要 HCLK 达到 216MHz，还需要配置 FLASH 延迟 Latency，。对于 STM32F7 系列，FLASH 延迟配置参数值是通过下表来确定的：

等待周期 (WS) (LATENCY)	HCLK (MHz)			
	电压范围 2.7 V - 3.6 V	电压范围 2.4 V - 2.7 V	电压范围 2.1 V - 2.4 V	电压范围 1.8 V - 2.1 V
0 WS（1 个 CPU 周期）	0 < HCLK ≤ 30	0 < HCLK ≤ 24	0 < HCLK ≤ 22	0 < HCLK ≤ 20
1 WS（2 个 CPU 周期）	30 < HCLK ≤ 60	24 < HCLK ≤ 48	22 < HCLK ≤ 44	20 < HCLK ≤ 40
2 WS（3 个 CPU 周期）	60 < HCLK ≤ 90	48 < HCLK ≤ 72	44 < HCLK ≤ 66	40 < HCLK ≤ 60
3 WS（4 个 CPU 周期）	90 < HCLK ≤ 120	72 < HCLK ≤ 96	66 < HCLK ≤ 88	60 < HCLK ≤ 80
4 WS（5 个 CPU 周期）	120 < HCLK ≤ 150	96 < HCLK ≤ 120	88 < HCLK ≤ 110	80 < HCLK ≤ 100
5 WS（6 个 CPU 周期）	150 < HCLK ≤ 180	120 < HCLK ≤ 144	110 < HCLK ≤ 132	100 < HCLK ≤ 120
6 WS（7 个 CPU 周期）	180 < HCLK ≤ 210	144 < HCLK ≤ 168	132 < HCLK ≤ 154	120 < HCLK ≤ 140
7 WS（8 个 CPU 周期）	210 < HCLK ≤ 216	168 < HCLK ≤ 192	154 < HCLK ≤ 176	140 < HCLK ≤ 160
8 WS（9 个 CPU 周期）	-	192 < HCLK ≤ 216	176 < HCLK ≤ 198	160 < HCLK ≤ 180
9 WS（10 个 CPU 周期）	-	-	198 < HCLK ≤ 216	-

表 4.3.2.1 STM32F7 系列等待周期表

从上表可以看出，在电压为 3.3V 的情况下，如果需要 HCLK 为 216MHz，那么等待周期必须为 7WS，也就是 8 个 CPU 周期。下面我们看看我们在 Stm32_Clock_Init 中调用函数 HAL_RCC_ClockConfig 的时候，第二个入口参数设置值：

```
ret=HAL_RCC_ClockConfig(&RCC_ClkInitStructure,FLASH_LATENCY_7);
```

从上可以看出，我们设置值为 FLASH_LATENCY_7，也就是 7WS，8 个 CPU 周期，与我们预期一致。时钟系统配置相关知识就给大家讲解到这里。

4.3.3 STM32F7 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《STM32F7 中文参考手册》5.3 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 STM32F7 的 HAL 库使能外设时钟的方法。

在 STM32F7 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 stm32f7xx_hal_rcc.h 定义的。大家打开 stm32f7xx_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->AHB1ENR, RCC_AHB1ENR_GPIOAEN);\
                                          tmpreg = READ_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOAEN);\
                                          UNUSED(tmpreg); \
                                         } while(0)
```

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->AHB1ENR, RCC_AHB1ENR_GPIOAEN);
```

这行代码的作用是，设置寄存器 RCC->AHB1ENR 的相关位为 1，至于是哪个位，是由宏定义

标识符 RCC_AHB1ENR_GPIOAEN 的值决定的，而它的值为：

```
#define  RCC_AHB1ENR_GPIOAEN          ((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->AHB1ENR 寄存器的最低位为 1。我们可以从 STM32F7 的中文参考手册中搜索 AHB1ENR 寄存器定义，最低位的作用是用来使用 GPIOA 时钟。AHB1ENR 寄存器的位 0 描述如下：

```
位 0      GPIOAEN: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 stm32f7xx_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->AHB1ENR &= ~(RCC_AHB1ENR_GPIOAEN))
```

同样，宏定义标识符__HAL_RCC_GPIOA_CLK_DISABLE()的作用是设置 RCC->AHB1ENR 寄存器的最低位为 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 STM32F7 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 IO 引脚复用器和映射

STM32F7 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《STM32F7 中文参考手册》第六章和芯片数据手册有详细的讲解哪些 GPIO 管脚是可以复用为哪些内置外设。

对于本小节知识，STM32F7 中文参考手册讲解比较详细，我们同样会从中抽取重要的知识点罗列出来。同时，我们会以串口使用为例给大家讲解具体的引脚复用的配置。

STM32F7 系列微控制器 IO 引脚通过一个复用器连接到内置外设或模块。该复用器一次只允许一个外设的复用功能（AF）连接到对应的 IO 口。这样可以确保共用同一个 IO 引脚的外设之间不会发生冲突。

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

1）完成复位后，所有 IO 都会连接到系统的复用功能 0（AF0）。

2）外设的复用功能映射到 AF1 到 AF13。

3）Cortex-M7 EVENTOUT 映射到 AF15。

简单的理解就是，每个引脚都可以配置为多个复用功能，那么这个引脚到底配置为哪个功能，可以通过开关（配置）来设定，就像一个模拟开关一样。复用器示意图如下图 4.4.1：

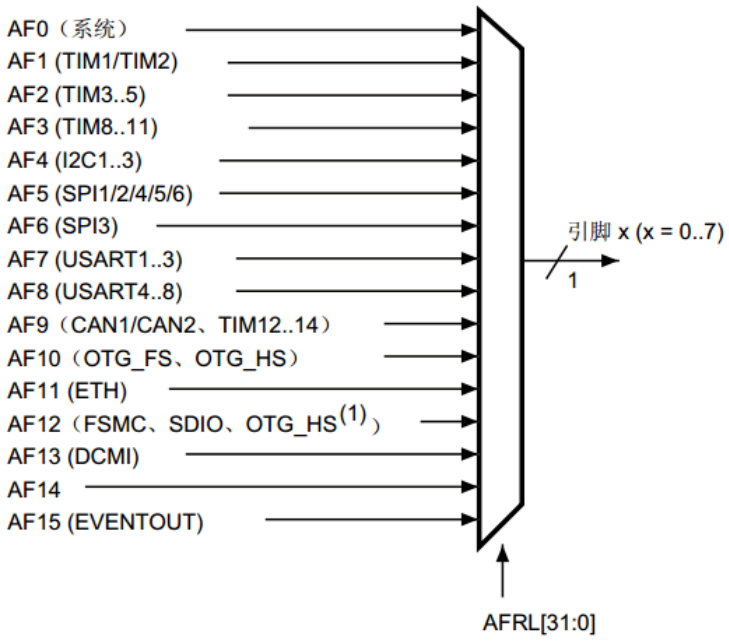


图 4.4.1STM32F7 复用器示意图

接下来，我们简单说明一下这个图要如何看，举个例子，阿波罗 STM32F7 开发板的原理图上 PC11 的原理图如图 4.4.2 所示：

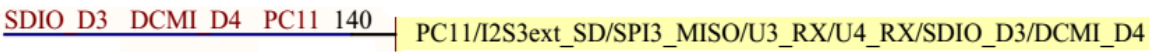


图 4.4.2 阿波罗 STM32F7 开发板 PC11 原理图

如上图所示，PC11 可以作为 SPI3_MISO/U3_RX/U4_RX/SDIO_D3/DCMI_D4/I2S3ext_SD 等复用功能输出。这么多复用功能，如果这些外设都开启了，那么对 STM32F7 来说那就可能乱套了，外设之间互相干扰。但是 STM32F7，由于有复用器功能，可以让 PC11 在某个时刻仅连接到需要使用的特定的外设，因此不存在互相干扰的情况。

上图 4.4.1 是针对引脚 0-7，对于引脚 8-15，控制寄存器为 GPIOx_AFRH。从图中可以看出。当需要使用复用功能的时候，我们配置相应的寄存器 GPIOx_AFRL 或者 GPIOx_AFRH，让对应引脚通过复用器连接到对应的复用功能外设。这里我们列出 GPIOx_AFRL 寄存器的描述，GPIOx_AFRH 的作用跟 GPIOx_AFRL 类似，只不过 GPIOx_AFRH 控制的是一组 IO 口的高八位，GPIOx_AFRL 控制的是一组 IO 口的低八位。GPIOx_AFRL 寄存器描述如下图 4.4.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	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
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

AFRLy 选择：

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

图 4.4.3 GPIOx_AFRL 寄存器位描述

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

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

1) JTAG/SWD:在器件复位之后，会将这些功能引脚指定为专用引脚。也就是说，这些引脚在复位后默认就是 JTAG/SWD 功能。如果我们要作为 GPIO 来使用，就需要对对应的 IO 口复用器进行配置。
2) RTC_REFIN:此引脚在系统复位之后要使用的话要配置为浮空输入模式。
3) MCO1 和 MCO2：这些引脚在系统复位之后要使用的话要配置为复用功能模式。

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

不是每个 IO 口都可以复用为任意复用功能外设。到底哪些 IO 可以复用为相关外设呢？这在芯片对应的数据手册(请参考光盘目录：\7，硬件资料\3，芯片资料\STM32F767IGT6.pdf)上面会有详细的表格列出来。对于 STM32F767，数据手册里面的 Table 12.Alternate function mapping 表格列出了所有的端口 AF 映射表，因为表格比较大，所以这里只列出 PORTA 的几个端口为例方便大家理解：

	PA0	PA5	PA8	PA9	PA10
AF0			MCO1		
AF1	TIM2_CH1 TIM2_CH1_ETR	TIM2_CH1 TIM2_CH1_ETR	TIM1_CH1	TIM1_CH2	TIM1_CH3
AF2	TIM5_CH1				
AF3	TIM8_ETR	TIM8_CH1N			
AF4			I2C3_SCL	I2C3_SMBA	
AF5				SPI2_SCK I2S2_CK	
AF6					
AF7	USART2_CTS	SPI1_SCK	USART1_CK	USART1_TX	USART1_RX

		I2S1_CK			
AF8	UART4_TX				
AF9					
AF10	SAI2_SD_B	OTG_HS_ULPI_CK	OTG_FS_SOF		OTG_FS_ID
AF11	ETH_MII_CRS				
AF12					
AF13				DCMI_D0	DCMI_D1
AF14		LCD_R4	LCD_R6		
AF15	EVENTOUT	EVENTOUT	EVENTOUT	EVENTOUT	EVENTOUT

表 4.4.4 PORTA 部分端口 AF 映射表

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

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
```

② 其次，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（**ADC 和 DAC 设置为模拟通道**）。

③ 再次,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。

④ 最后，我们需要配置 GPIOx_AFRL 或者 GPIOx_AFRH 寄存器，将 IO 连接到所需的 AFx。对于 PA9,PA10 复用为 USART1 的发送接收引脚，根据表 4.4.4 可知都需要连接 AF7。

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_FAST;         //高速
GPIO_Initure.Alternate=GPIO_AF7_USART1;//连接 AF7 复用为串口 1 的发送引脚
HAL_GPIO_Init(GPIOA,&GPIO_Initure);         //初始化 PA9
```

通过上面的配置，PA9 就通过映射器链接到 AF7，也就是复用为串口 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，同样也是链接 AF7，修改 Pin 成员变量值为 PIN_10 即可。从表 4.4.4 可以看出，PA9 还可以作为 TIM1_CH2 功能引脚，如果我们希望 PA9 作为 TIM1_CH2 引脚，那么我们需要修改 PA9 的映射关系，修改方法如下：

```
GPIO_Initure.Alternate= GPIO_AF1_TIM2;//连接 AF1 复用为 TIM2_CH1 引脚
```

对于 GPIO 初始化结构体成员变量 Alternate 的取值范围，在 HAL 库中有详细定义，取值范围如下：

```
#define IS_GPIO_AF(AF)  (((AF) == GPIO_AF0_RTC_50Hz)||((AF) == GPIO_AF9_TIM14)     || \
                         ((AF) == GPIO_AF0_MCO) || ((AF) == GPIO_AF0_TAMPER)    || \
                         ((AF) == GPIO_AF0_SWJ) || ((AF) == GPIO_AF0_TRACE)     || \
                         ((AF) == GPIO_AF1_TIM1)|| ((AF) == GPIO_AF1_TIM2)      || \
                            ...//此处省略部分代码
```

```
                                ((AF) == GPIO_AF8_UART7)|| ((AF) == GPIO_AF8_UART8) || \
                                ((AF) == GPIO_AF12_FMC) ||  ((AF) == GPIO_AF6_SAI1)    || \
                                ((AF) == GPIO_AF14_LTDC))
```

STM32F7 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

CM7 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 STM32F767 并没有使用 CM7 内核的全部东西，而是只用了它的一部分。STM32F767xx 总共有 118 个中断，以下仅以 STM32F767xx 为例讲解。

STM32F767xx 的 118 个中断里面，包括 10 个内核中断和 108 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 108 个可屏蔽中断。在 MDK 内，与 NVIC 相关的寄存器，MDK 为其定义了如下的结构体：

```
typedef struct
{
  __IOM uint32_t ISER[8U];               //Interrupt Set Enable Register
        uint32_t RESERVED0[24U];
  __IOM uint32_t ICER[8U];               //Interrupt Clear Enable Register
        uint32_t RSERVED1[24U];
  __IOM uint32_t ISPR[8U];               //Interrupt Set Pending Register
        uint32_t RESERVED2[24U];
  __IOM uint32_t ICPR[8U];               //Interrupt Clear Pending Register
        uint32_t RESERVED3[24U];
  __IOM uint32_t IABR[8U];               //Interrupt Active bit Register
        uint32_t RESERVED4[56U];
  __IOM uint8_t   IP[240U];              //Interrupt Priority Register (8Bit wide)
        uint32_t RESERVED5[644U];
  __OM   uint32_t STIR;                  //Software Trigger Interrupt Register
}  NVIC_Type;
```

STM32F767 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 STM32F767 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 CM7 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 STM32F767 的可屏蔽中断最多只有 108 个，所以对我们来说，有用的就是四个（ISER[0~3]），总共可以表示 128 个中断。而 STM32F767 只用了其中的 108 个。ISER[0]的 bit0~31 分别对应中断 0~31；ISER[1]的 bit0~32 对应中断 32~63；其他以此类推，这样总共 108 个中断就可以分别对应上了。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 stm32f767xx.h 里面的第 69 行处。

ICER[8]：全称是：Interrupt Clear Enable Registers，是一个中断除能寄存器组。该寄存器组与 ISER 的作用恰好相反，是用来清除某个中断的使能的。其对应位的功能，也和 ICER 一样。这里要专门设置一个 ICER 来清除中断位，而不是向 ISER 写 0 来清除，是因为 NVIC 的这些寄存器都是写 1 有效的，写 0 是无效的。

ISPR[8]：全称是：Interrupt Set Pending Registers，是一个中断挂起控制寄存器组。每个位对应的中断和 ISER 是一样的。通过置 1，可以将正在进行的中断挂起，而执行同级或更高级别的中断。写 0 是无效的。

ICPR[8]：全称是：Interrupt Clear Pending Registers，是一个中断解挂控制寄存器组。其作用与 ISPR 相反，对应位也和 ISER 是一样的。通过设置 1，可以将挂起的中断接挂。写 0 无效。

IABR[8]：全称是：Interrupt Active Bit Registers，是一个中断激活标志位寄存器组。对应位所代表的中断和 ISER 一样，如果为 1，则表示该位所对应的中断正在被执行。这是一个只读寄存器，通过它可以知道当前在执行的中断是哪一个。在中断执行完了由硬件自动清零。

IP[240]：全称是：Interrupt Priority Registers，是一个中断优先级控制的寄存器组。这个寄存器组相当重要！STM32F767 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 STM32F767 只用到了其中的 108 个。IP[109]~IP[0]分别对应中断 109~0(其中，98 和 79 没用到，所以，总共还是 108 个)。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 STM32F767 的中断分组：STM32F767 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 5.2.6.1 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 5.2.6.1 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 108 个中断，每个中断的中断优先寄存器的高四位中的最高 3 位是抢占优先级，低 1 位是响应优先级。每个中断，你可以设置抢占优先级为 0~7，响应优先级为 1 或 0。抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高。

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

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

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

通过以上介绍，我们熟悉了 STM32F7 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 stm32f7xx_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_cm4.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 << 8U)                              );
  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 stm32f7xx_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 stm32f7xx_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 stm32f7xx_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，点用一个内存单元，值域为 0～255。利用它可以访问 51 单片机内部的所有特殊功能寄存器。如用 sfr P1 = 0x90 这一句定义 P1 为 P1 端口在片内的寄存器。然后我们往地址为 0x80 的寄存器设值的方法是：P0=value;

那么在 STM32 中，是否也可以这样做呢？？答案是肯定的。肯定也可以通过同样的方式来做，但是 STM32 因为寄存器太多太多，如果一一以这样的方式列出来，那要好大的篇幅，既不方便开发，也显得太杂乱无序的感觉。所以 MDK 采用的方式是通过结构体来将寄存器组织在一起。下面我们就讲解 MDK 是怎么把结构体和地址对应起来的，为什么我们修改结构体成员变量的值就可以达到操作对应寄存器的值。这些事情都是在 stm32f7xx.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《STM32F7 中文参考手册》中的寄存器地址映射表 22(P197)。这里我们选用 GPIOA 为例来讲解。GPIOA 寄存器地址映射如下表 4.6.1：

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

表 4.6.1 GIPOA 寄存器地址偏移表

从这个表我们可以看出，因为 GIPO 寄存器都是 32 位，所以每组 GPIO 的 10 个寄存器中，每个寄存器占有 4 个地址，一共占用 40 个地址，地址偏移范围为（0x00~0x24）。这个地址偏移是相对 GPIOA 的基地址而言的。GPIOA 的基地址是怎么算出来的呢？因为 GPIO 都是挂载在 AHB1 总线之上，所以它的基地址是由 AHB1 总线的基地址加上 GPIOA 在 AHB1 总线上的偏移地址决定的。同理依次类推，我们便可以算出 GPIOA 基地址了。下面我们打开 stm32f767xx.h 定位到 GPIO_TypeDef 定义处：

```
typedef struct
{
  __IO uint32_t MODER;
  __IO uint32_t OTYPER;
  __IO uint32_t OSPEEDR;
  __IO uint32_t PUPDR;
```

```
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint32_t BSRR;
    __IO uint32_t LCKR;
    __IO uint32_t AFR[2];
} GPIO_TypeDef;
```

然后定位到：

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

可以看出，GPIOA 是将 GPIOA_BASE 强制转换为 GPIO_TypeDef 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              (AHB1PERIPH_BASE + 0x0000U)
```

依次类推，可以找到最顶层：

```
#define AHB1PERIPH_BASE         (PERIPH_BASE + 0x00020000U)
#define PERIPH_BASE             0x40000000U
```

所以我们便可以算出 GPIOA 的基地址位：

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

下面我们再跟《STM32F7 中文参考手册》比较一下看看 GPIOA 的基地址是不是 0x40020000 。截图 P53 存储器映射表我们可以看到，GPIOA 的起始地址也就是基地址确实是 0x40020000：

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

图 4.6.2 GPIO 存储器地址映射表

同样的道理，我们可以推算出其他外设的基地址。

上面我们已经知道 GPIOA 的基地址，那么那些 GPIOA 的 10 个寄存器的地址又是怎么算出来的呢？在上面我们讲过 GPIOA 的各个寄存器对于 GPIOA 基地址的偏移地址，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？这里涉及到结构体成员变量地址对齐方式方面的知识，这方面的知识大家可以在网上查看相关资料复习一下，这里我们不做详细讲解。在我们定义好地址对齐方式之后，每个成员变量对应的地址就可以根据其基地址来计算。对于结构体类型 GPIO_TypeDef，他的所有成员变量都是 32 位，成员变量地址具有连续性。所以自然而然我们就可以算出 GPIOA 指向的结构体成员变量对应地址了。

寄存器	偏移地址	实际地址=基地址+偏移地址
GPIOA_MODER	0x00	0x40020000+0x00
GPIOA_OTYPER	0x04	0x40020000+0x04
GPIOA_OSPEEDER	0x08	0x40020000+0x08
GPIOA_PUPDR	0x0C	0x40020000+0x0c
GPIOA_IDR	0x10	0x40020000+0x10
GPIOA_ODR	0x14	0x40010800+0x14
GPIOA_BSRR	0x18	0x40020000+0x18
GPIOA_LCKR	0x1c	0x40020000+0x1c
GPIOA_AFRL	0x20	0x40020000+0x20
GPIOA_AFRH	0x24	0x40020000+0x24

表 4.6.3 GPIOA 各寄存器实际地址表

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

这就是为什么固件库里面：GPIOA->BSRR=value;就是设置地址为 0x40020000 +0x18 (BSRR 偏移量)=0x40020018 的寄存器 BSRR 的值了。它和 51 里面 P0=value 是设置地址为 0x80 的 P0 寄存器的值是一样的道理。

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

4.7 MDK 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 stm32f7xx_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：

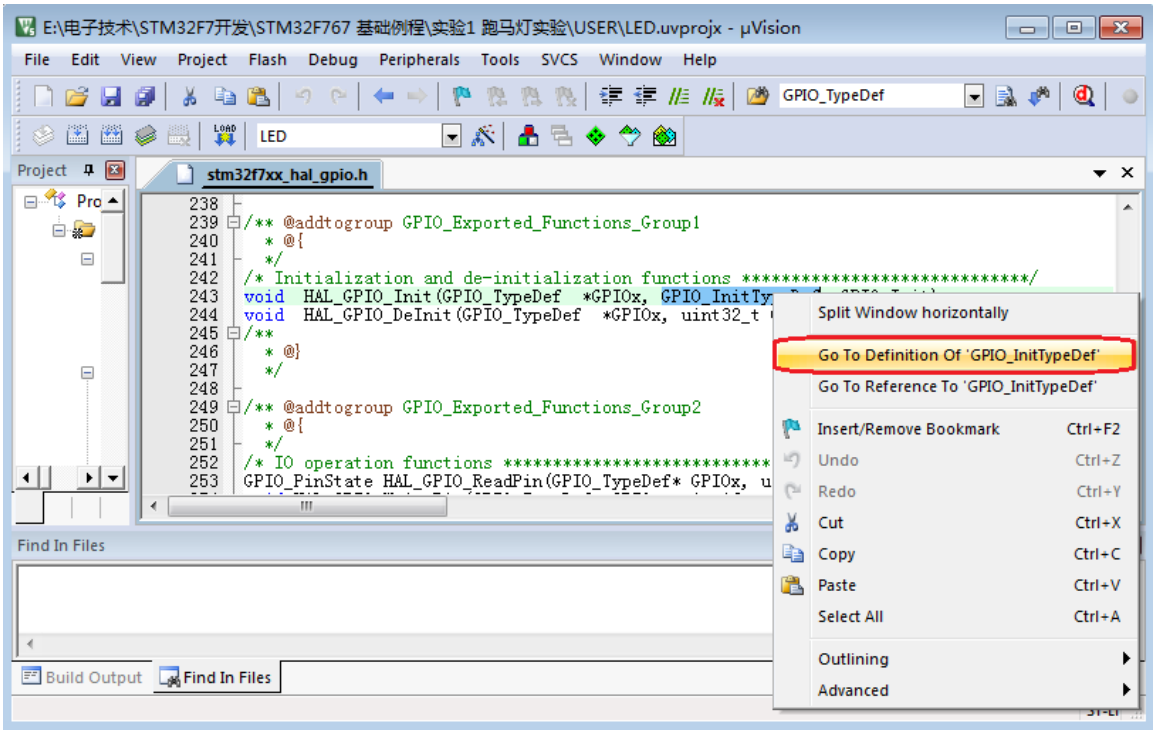


图 4.7.1 查看类型定义方法

于是定位到 stm32f7xx_hal_gpio.h 中 GPIO_InitTypeDef 的定义处：

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

可以看到这个结构体有 5 个成员变量，这也告诉我们一个信息，一个 GPIO 口的状态是由模式（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 stm32f7xx_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));
```

```
    assert_param(IS_GPIO_PULL(GPIO_Init->Pull));
    …//此处省略部分代码
    assert_param(IS_GPIO_AF(GPIO_Init->Alternate));
    …//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) || \
                                        …//此处省略部分代码
                                        ((INSTANCE) == GPIOJ) || \
                                        ((INSTANCE) == GPIOK))
```

很明显可以看出，GPIOx 的取值规定只允许是 GPIOA~GPIOK。

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

```
#define IS_GPIO_PIN(PIN)                ((PIN) & GPIO_PIN_MASK ) != (uint32_t)0x00)
```

同时，宏定义标识符 GPIO_PIN_MASK 的定义为：

```
#define GPIO_PIN_MASK                   ((uint32_t)0x0000FFFF)
```

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。

对于其他成员变量 Mode 以及 Alternate，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_FAST;         //高速
GPIO_Initure.Alternate=GPIO_AF7_USART1;   //复用为 USART1
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，Pull 和 Alternate 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_FAST;         //高速
GPIO_Initure.Alternate=GPIO_AF7_USART1;   //复用为 USART1
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 */
  uint32_t Alternate;    /*!< Peripheral to be connected to the selected pins.
                        This parameter can be a value of @ref GPIO_Alternate_function_selection */
}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。

4.8 手把手教你入门 STM32CubeMX 图形配置工具

上一章我们讲解 stm32Cube 的时候提到 stm32Cube 包含 2 个部分：一部分是上一章我们讲解的嵌入式软件包（包括 HAL 库），另一部分是图形化配置工具 STM32CubeMX。本小节我们将给大家讲解 STM32CubeMX 相关知识，带领大家入门 STM32CubeMX 图形化配置工具。之所以我们要把 STM32CubeMX 讲解放在本小节，是因为 STM32CubeMX 最基本也是最重要的用途是配置时钟系统，所以我们要先讲解 STM32F7 的时钟系统之后，才能教大家学习 STM32CubeMX。这部分内容我们分 3 个小节来讲解：

- 4.8.1 STM32CubeMX 简介
- 4.8.2 STM32CubeMX 运行环境搭建
- 4.8.3 使用 STM32CubeMX 工具配置工程模板

4.8.1 STM32CubeMX 简介

STM32CubeMX 是 ST 意法半导体近几年来大力推荐的 STM32 芯片图形化配置工具，允许用户使用图形化向导生成 C 初始化代码，可以大大减轻开发工作，时间和费用。STM32CubeMX 几乎覆盖了 STM32 全系列芯片。它具有如下特性：

① 直观的选择 MCU 型号，可指定系列、封装、外设数量等条件
② 微控制器图形化配置
③ 自动处理引脚冲突
④ 动态设置时钟树，生成系统时钟配置代码
⑤ 可以动态设置外围和中间件模式和初始化
⑥ 功耗预测
⑦ C 代码工程生成器覆盖了 STM32 微控制器初始化编译软件，如 IAR，KEIL，GCC。
⑧ 可以独立使用或者作为 Eclipse 插件使用

对于 STM32CubeMX 和 STM32Cube 的关系这里我们还需要特别说明一下，STM32Cube 包含 STM32CubeMX 图形工具和 STM32Cube 库两个部分，使用 STM32CubeMX 配置生成的代码，是基于 STM32Cube 库的。也就是说，我们使用 STM32CubeMX 配置出来的初始化代码，和 STM32Cube 库兼容，例如硬件抽象层代码就是使用的 STM32 的 HAL 库。不同的 STM32 系列芯片，会有不同的 STM32Cube 库支持，而 STM32CubeMX 图形工具只有一种。所以我们配置不同的 STM32 系列芯片，选择不同的 STM32Cube 库即可。它们之间的关系如下图 4.8.1.1：

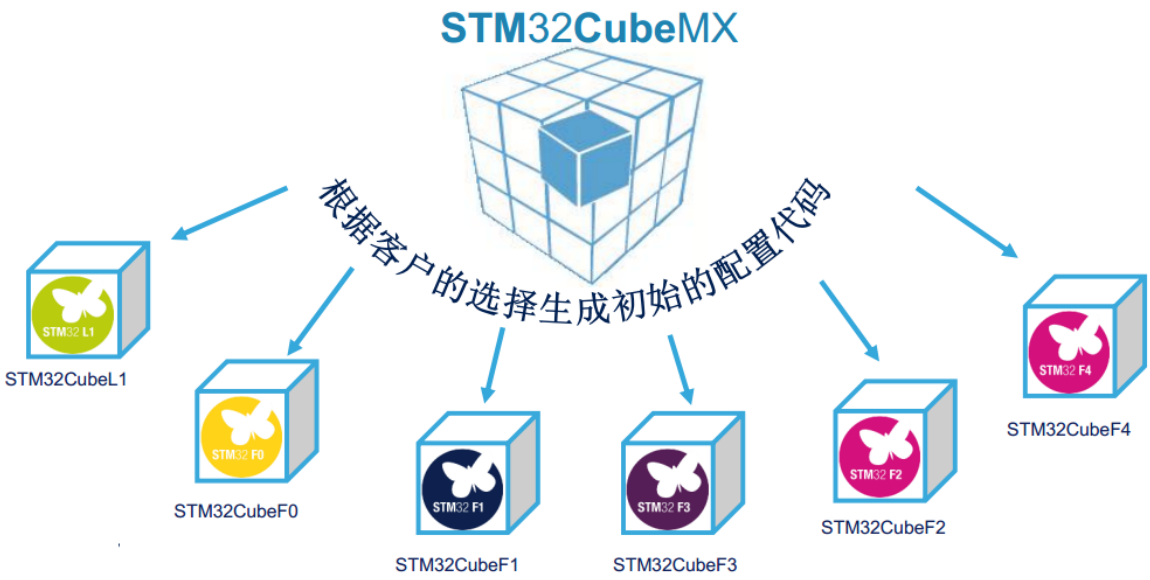


图 4.8.1.1 STM32CubeMX 和 STM32Cube 库的关系

4.8.2 STM32CubeMX 运行环境搭建

STM32CubeMX 运行环境搭建包含两个部分。首先是 Java 运行环境安装，其次是 STM32CubeMX 软件安装。对于 Java 运行环境，大家可以到 Java 官网 www.java.com 下载最新的 Java 软件，也可以直接从我们光盘复制安装包，目录为：\6，软件资料\1，软件\Java 安装包。这里大家需要注意，STM32CubeMX 的 Java 运行环境版本必须是 V1.7 及以上，如果你的电脑安装过 V1.7 以下版本，请先删掉后重新安装最新版本。

对于 Java 运行环境安装，我们这里就不做过多讲解，大家直接双击安装包，根据提示安装即可。安装完成之后提示界面如下图 4.8.2.1：

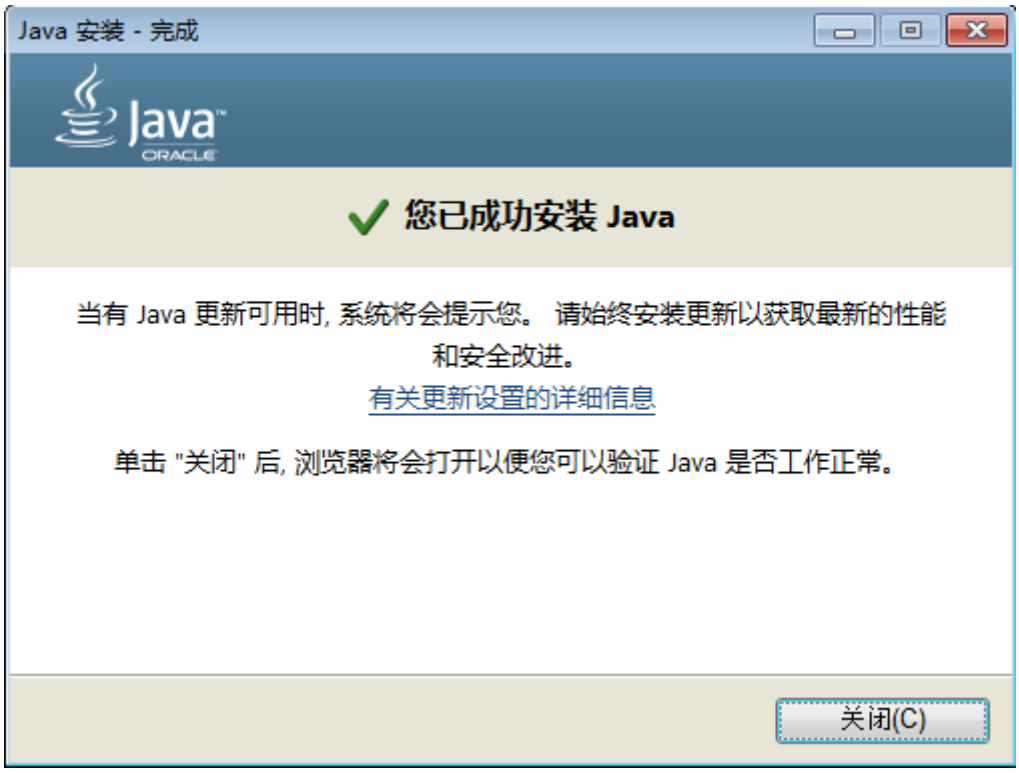


图 4.8.2.1 Java 安装成功提示界面

安装完 Java 运行环境之后，为了检测是否正常安装，我们可以打开 Windows 的命令输入框，输入：java –version 命令，如果显示 Java 版本信息，则安装成功。提示信息如下图 4.8.2.2：

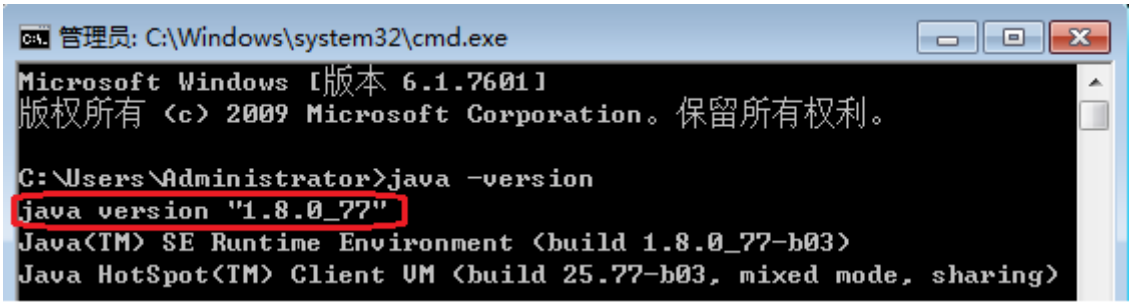


图 4.8.2.2 查看 Java 版本

在安装完 Java 运行环境之后，接下来我们安装 STM32CubeMX 图形化工具。该软件大家同样可以直接从光盘复制，目录为：\6，软件资料\1，软件\STM32CubeMX，也可以直接从 ST 官方下载，下载地址为：www.st.com/stm32cube。

接下来我们直接双击 STM32CubeMX 安装包，根据提示信息安装即可。安装完成之后提示信息如下图 4.8.2.3 所示：

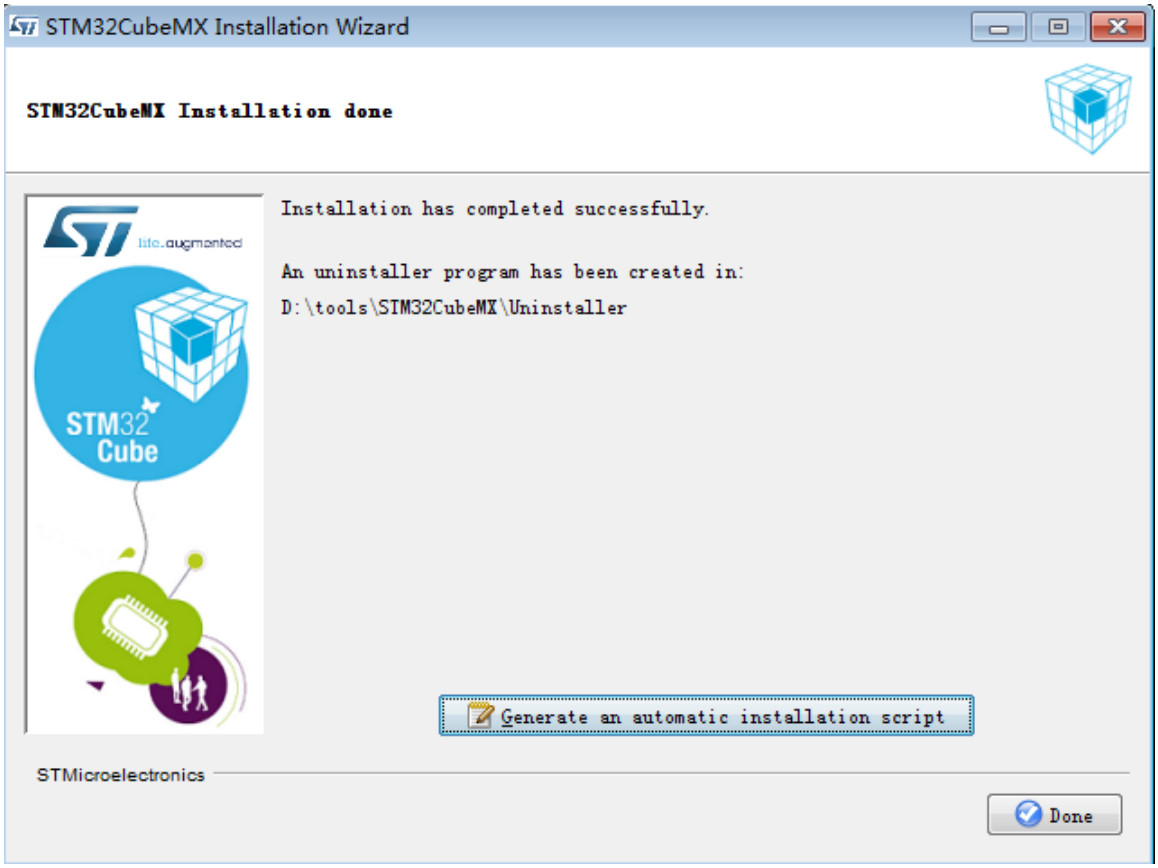


图 4.8.2.3 STM32CubeMX 安装完成界面

安装完成之后，我们打开软件，如果软件安装成功，打开软件之后的界面如下图 4.8.2.4 所示：

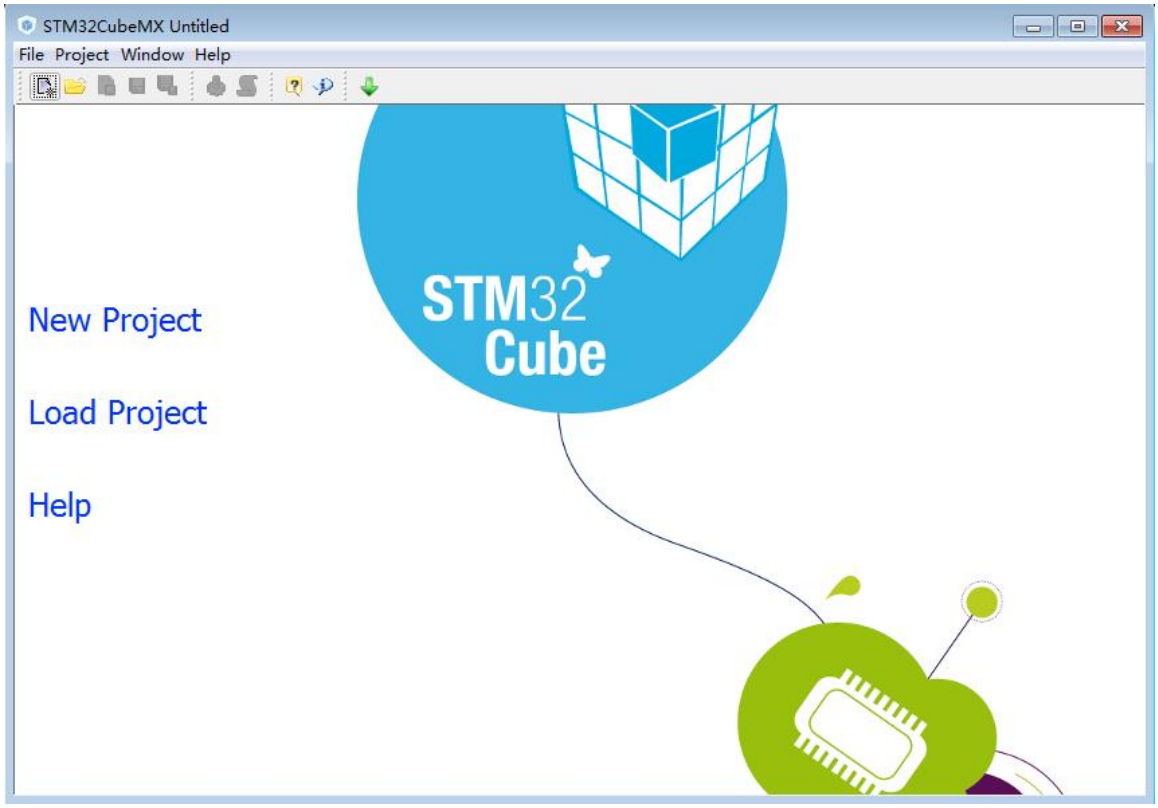


图 4.8.2.4 STM32CubeMX 打开后的显示界面

在安装好 STM32CubeMX 之后，接下来我们要在软件中指定 STM32Cube 软件包。在 STM32CubeMX 操作界面，依次点击 Help->Updater Settings，弹出界面如下图 4.8.2.5 所示：

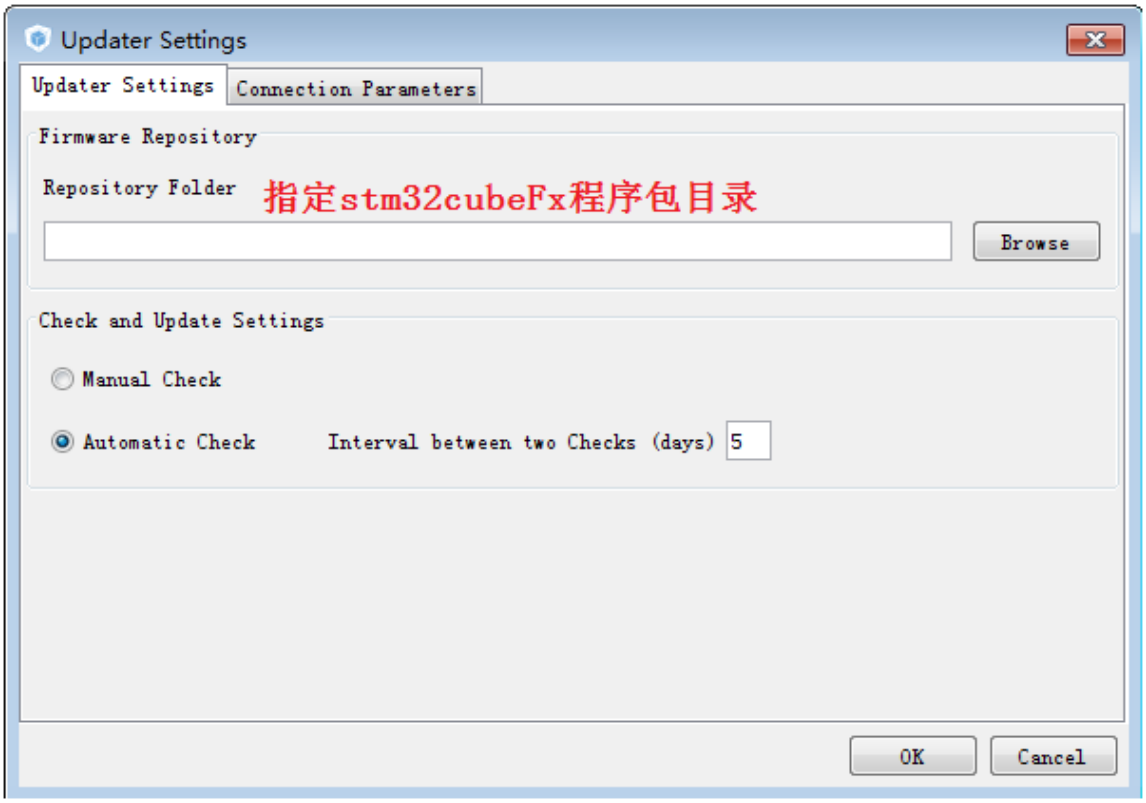


图 4.8.2.5 Updater Settings 操作界面

在上图 4.8.2.5 中，我们只需要点击 Browse 按钮，定位到我们 3.2 小节讲解的 stm32cubefx 存放目录即可。这里大家注意，stm32cubefx 文件夹名字遵循 STM32Cube_FW_Fx_Vm.n.z 格式，我们指定的“Repository Folder”下面必须存在一个或者多个 STM32Cube_FW_Fx_Vm.n.z 格式程序包，在 STM32CubeMX 生成工程的时候，会根据我们选择的芯片型号，去这个目录加载必要的库文件。一般情况下，我们会新建一个目录，然后把我们需要使用的各种 stm3cubefx 支持包解压放到该目录之下，然后把该目录指定为“Repository Folder”即可。操作方法如下图所示：

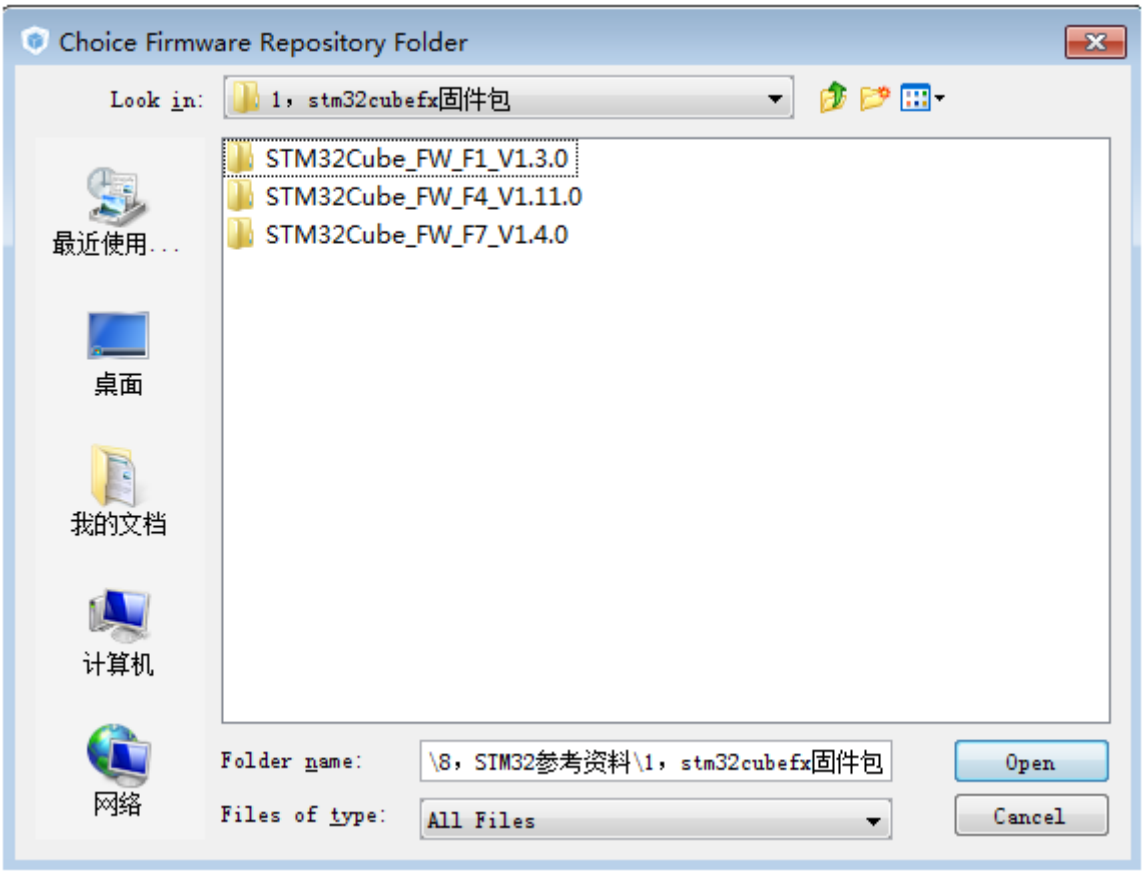


图 4.8.2.6 指定程序库目录

实际上，我们也可以直接在 STM32CubeMX 中点击 Help->Install New Libraries 下载需要的程序库，但是由于速度比较慢，而且在下载过程中很容易中断，所以我们不推荐直接在 CubeMX 中下载。

接下来我们将讲解怎么使用 STM32CubeMX 新建一个完整的 STM32F7 工程。

4.8.3 使用 STM32CubeMX 工具配置工程模板

大多数情况下，我们都只使用 STM32CubeMX 来生成工程的时钟系统初始化代码以及外设的初始化代码，因为用户控制逻辑代码是无法在 STM32CubeMX 中完成的，需要用户自己根据需求来实现。使用 STM32CubeMX 配置工程的一般步骤为：

1) 工程初步建立和保存
2) RCC 设置
3) 时钟系统（时钟树）配置
4) GPIO 功能引脚配置
5) 生成工程源码
6) 编写用户代码

接下来我们将按照上面 6 个步骤，依次教大家使用 STM32CubeMX 工具生成一个完整的工程。

4.8.3.1 工程初步建立和保存

工程建立的方法有两种方法，第一种方法是打开 STM32CubeMX 之后在主界面点击 New Project 按钮，第二种方法是在菜单栏依次点击 File->New Project。操作方法如下图 4.8.3.1.1 所

示：

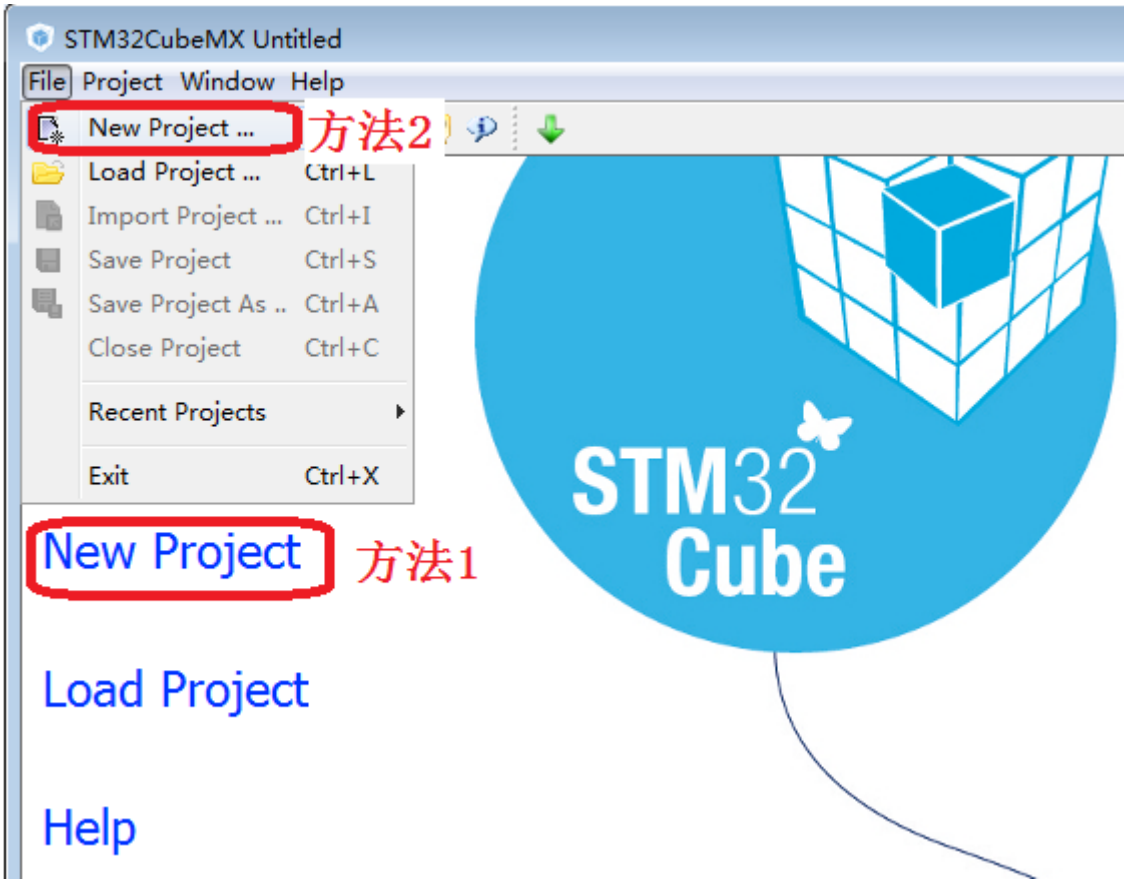


图 4.8.3.1.1 新建工程

点击新建工程按钮之后，会弹出 MCU 选择窗口。我们依次在选项卡 Series，Lines 和 Package 之下选择与我们使用的芯片 STM32F7 对应的参数，然后选择对应的芯片型号，最后点击 OK 按钮。如果是 STM32F767IGT6，请选择 Lines 为 STM32F7x7，然后选择 STM32F767IGT6，如果是 STM32F746IGT6，请选择 Lines 为 STM32F7x6，然后选择 STM32F746IGT6。这里我们以 STM32F767IGT6 为例，操作方法如下图 4.8.3.1.2 所示：

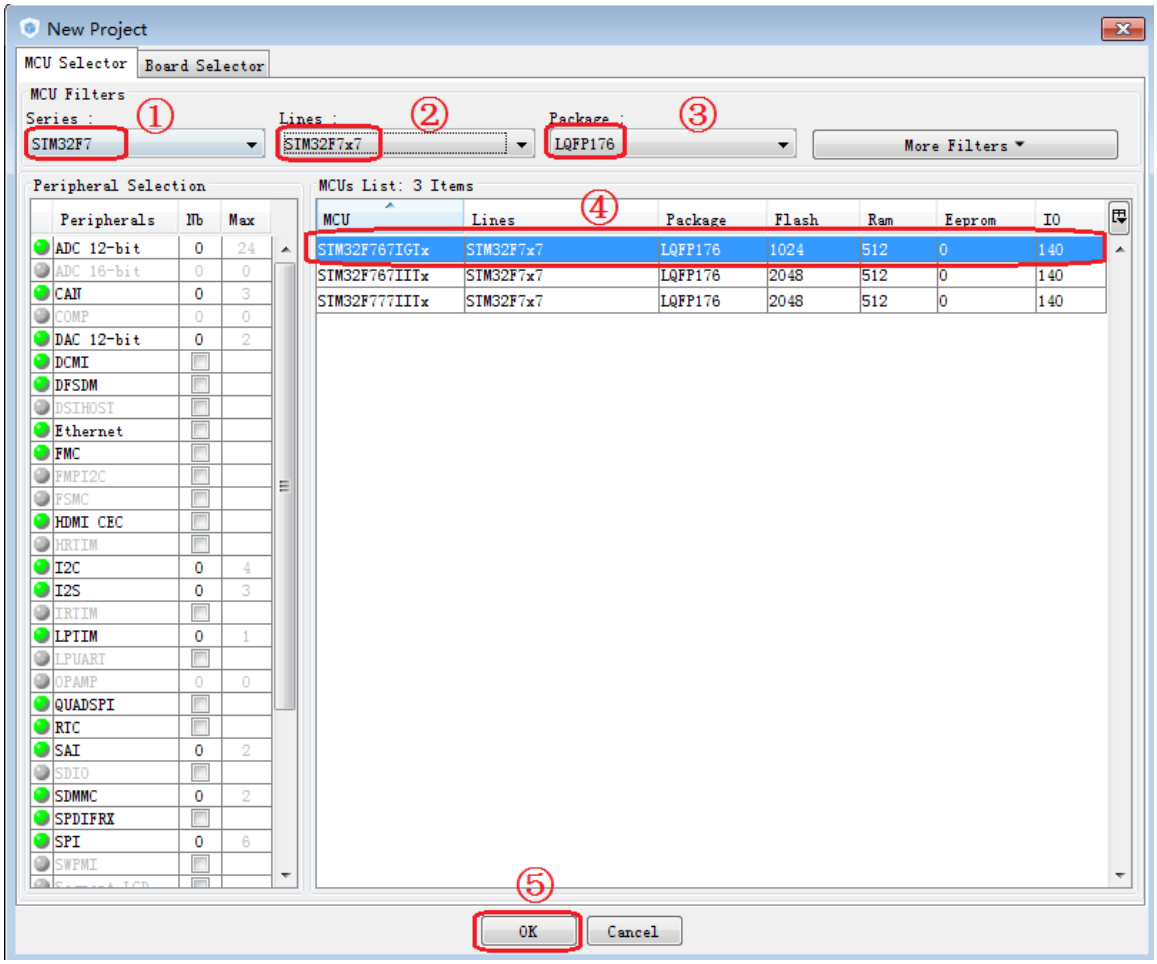


图 4.8.3.1.2 选择 MCU

为了避免在软件使用过程中出现意外导致工程没有保存，所以我们选择好芯片型号之后，先对工程进行保存。依次点击菜单栏 File->Save Project，然后保存工程到某个文件夹下面即可。操作过程如下图 4.8.3.1.3 所示：

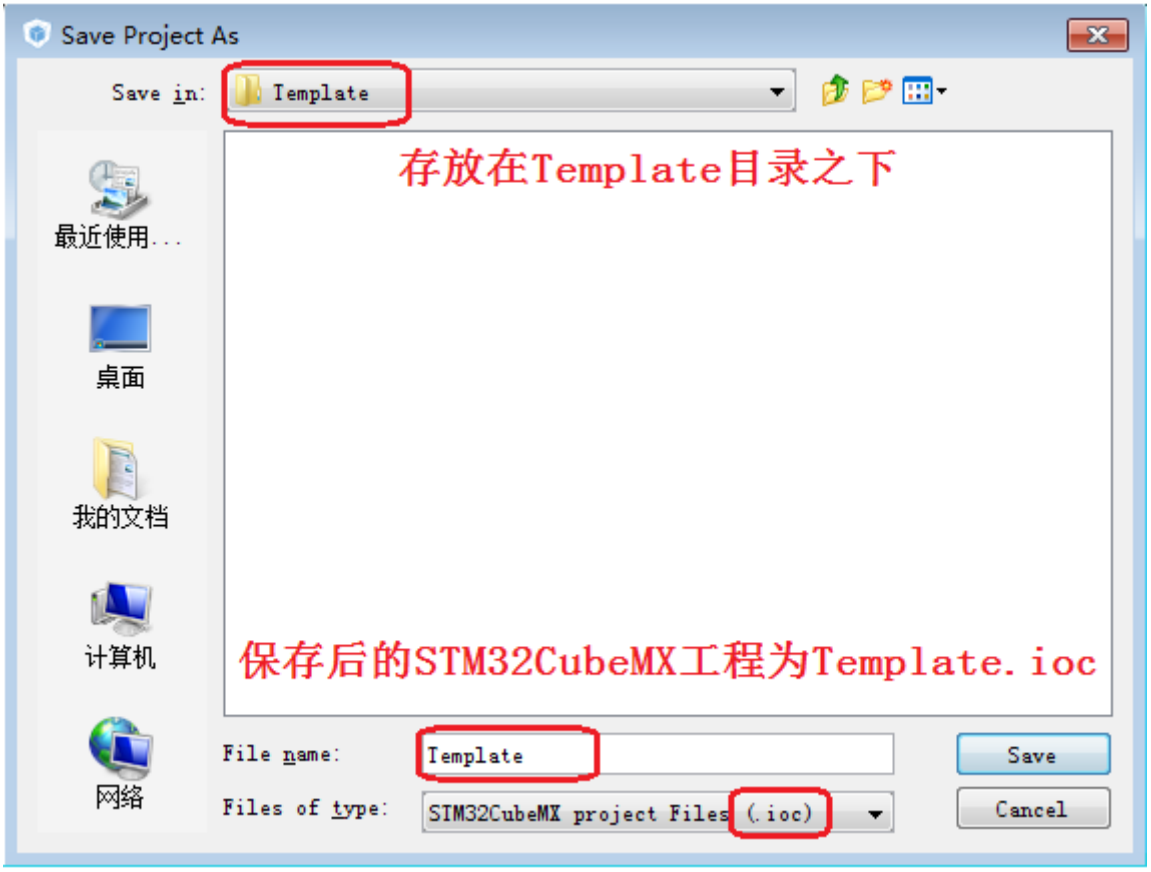


图 4.8.3.1.3 保存工程

保存完成之后，大家进入 Template 目录后发现目录中多了一个 Template.ioc 文件，下次我们点击这个文件就可以直接打开这个工程。

工程新建好之后会直接进入 Pinout 选项卡，这个时候界面会展示芯片完整引脚图，如下图 4.8.3.1.4 所示：

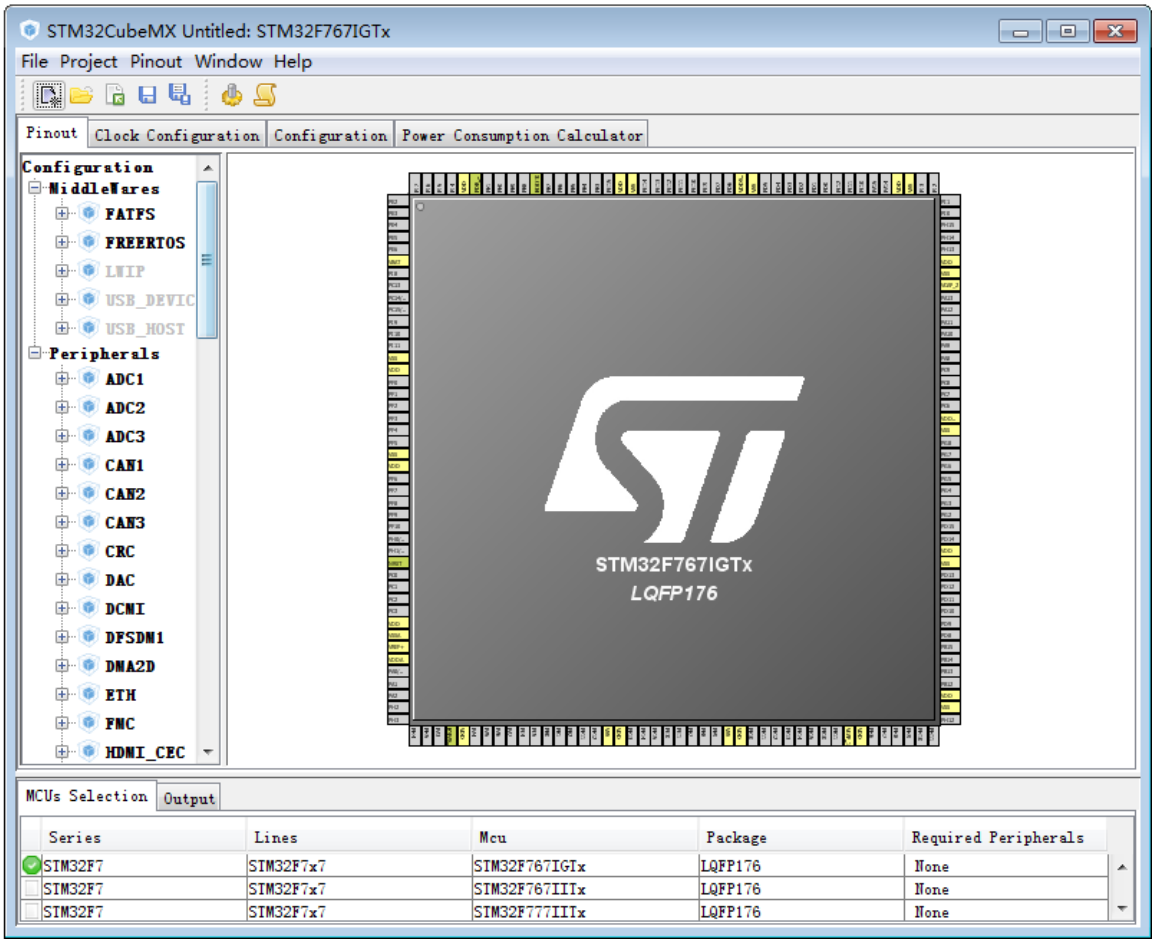


图 4.8.3.1.4 STM32CubeMX 中芯片引脚图

在引脚图中，我们可以对引脚功能进行配置。图中黄色的引脚主要是一些电源和 GND 引脚。如果某个引脚被使用，那么会显示为绿色。

4.8.3.2 RCC 设置

对 STM32 芯片而言，RCC 配置的重要性不言而喻。在 STM32CubeMX 中，RCC 相关设置却非常简单，因为它把时钟系统独立出来配置。在操作界面，依次点击选项卡 Pinout->Peripherals->RCC 便可进入 RCC 配置栏，操作步骤如下图 4.8.3.2.1 所示：

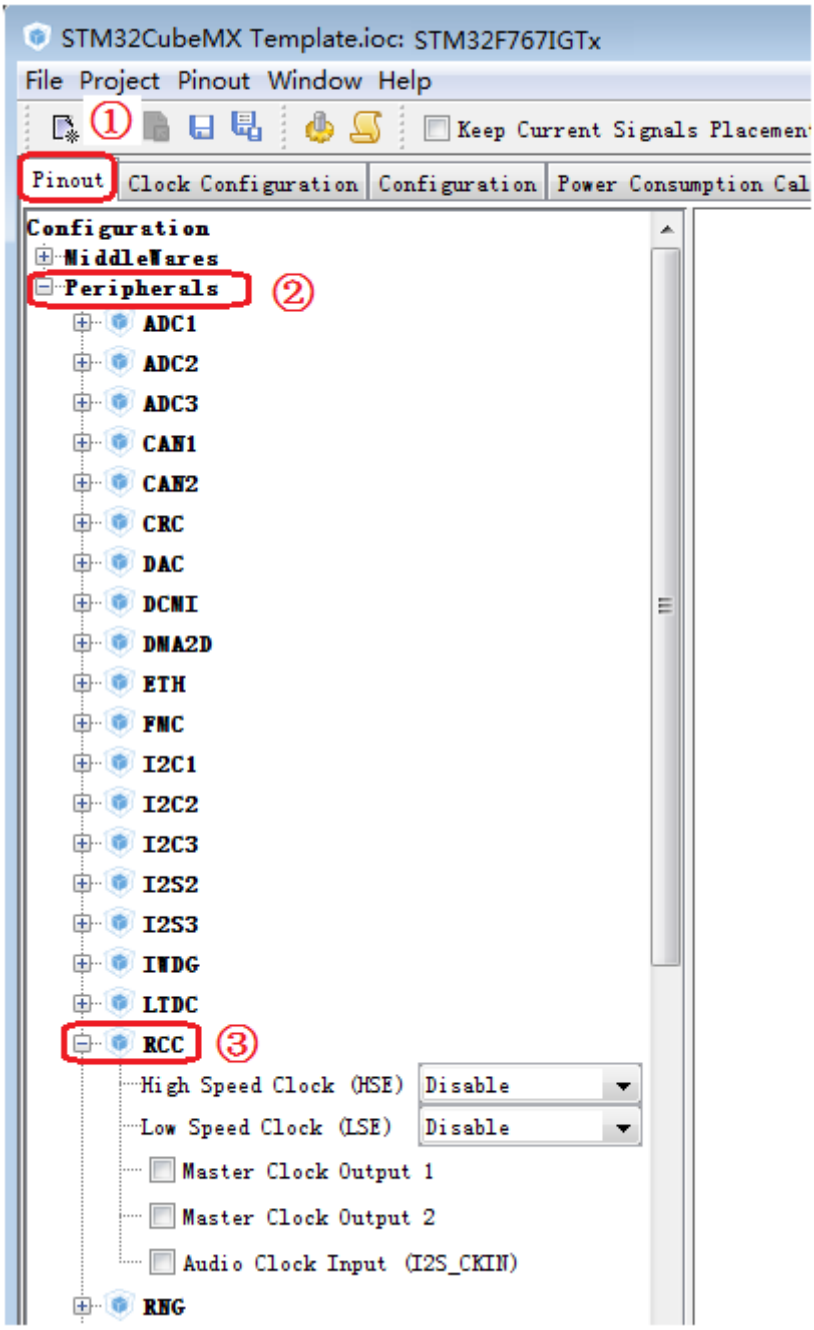


图 4.8.3.2.1 进入 RCC 配置栏

从上图可以看出，RCC 配置栏实际上只有 5 个配置项。选项 High Speed Clock（HSE）用来配置 HSE，第二个选项 Low Speed Clock（LSE）用来配置 LSE，选项 Master Clock Output 1 用来选择是否使能 MCO1 引脚时钟输出，选项 Master Clock Output 2 用来选择是否使能 MCO2 引脚时钟输出，最后一个选项 Audio Clock Input（I2S_CKIN）用来选择是否从 I2S_CKIN(PC9)输入 I2S 时钟。这里大家要注意，因为选项 Master Clock Output 2 和选项 Audio Clock Input（I2S_CKIN）都是使用的 PC9 引脚，所以如果我们使能了其中一个，那么另一个选项会自动显示为红色，也就是不允许配置，这就是 STM32CubeMX 的自动冲突检测功能。

本小节我们只使用到 HSE，所以我们设置选项 High Speed Clock（HSE）的值为 Crystal/Ceramic Resonator（使用晶振/陶瓷振荡器）即可。这里还需要说明一下，值 Bypass Clock

Source 的意思是旁路时钟源，也就是不使用使用晶振/陶瓷振荡器，直接通过外部提供一个可靠的 4-26MHz 时钟作为 HSE。配置好的 RCC 配置选项如下图 4.8.3.2.2 所示：

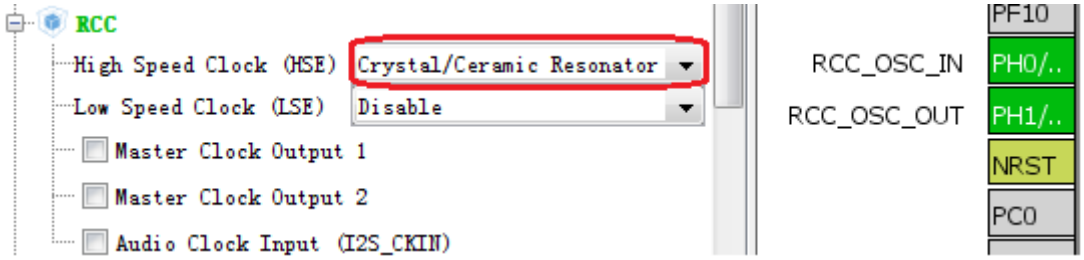


图 4.8.3.2.2 RCC 选项配置

从上图还可以看出，在我们打开了 HSE 之后，右边的引脚图中，相应的引脚会由灰色变为绿色，表示该引脚已经被使用。配置完 RCC 之后，接下来我们来看看配置时钟系统树的方法。

4.8.3.3 时钟系统（时钟树）配置

在使用 STM32CubeMX 配置时钟树之前，大家需要充分理解 STM32 时钟系统，这在我们前面 4.3 小节有非常详细的讲解，只有熟练掌握了 STM32 时钟系统，那么在软件中配置时钟树才会得心应手。

点击 Clock Configuration 选项卡即可进入时钟系统配置栏，如下图 4.8.3.3.1 所示：

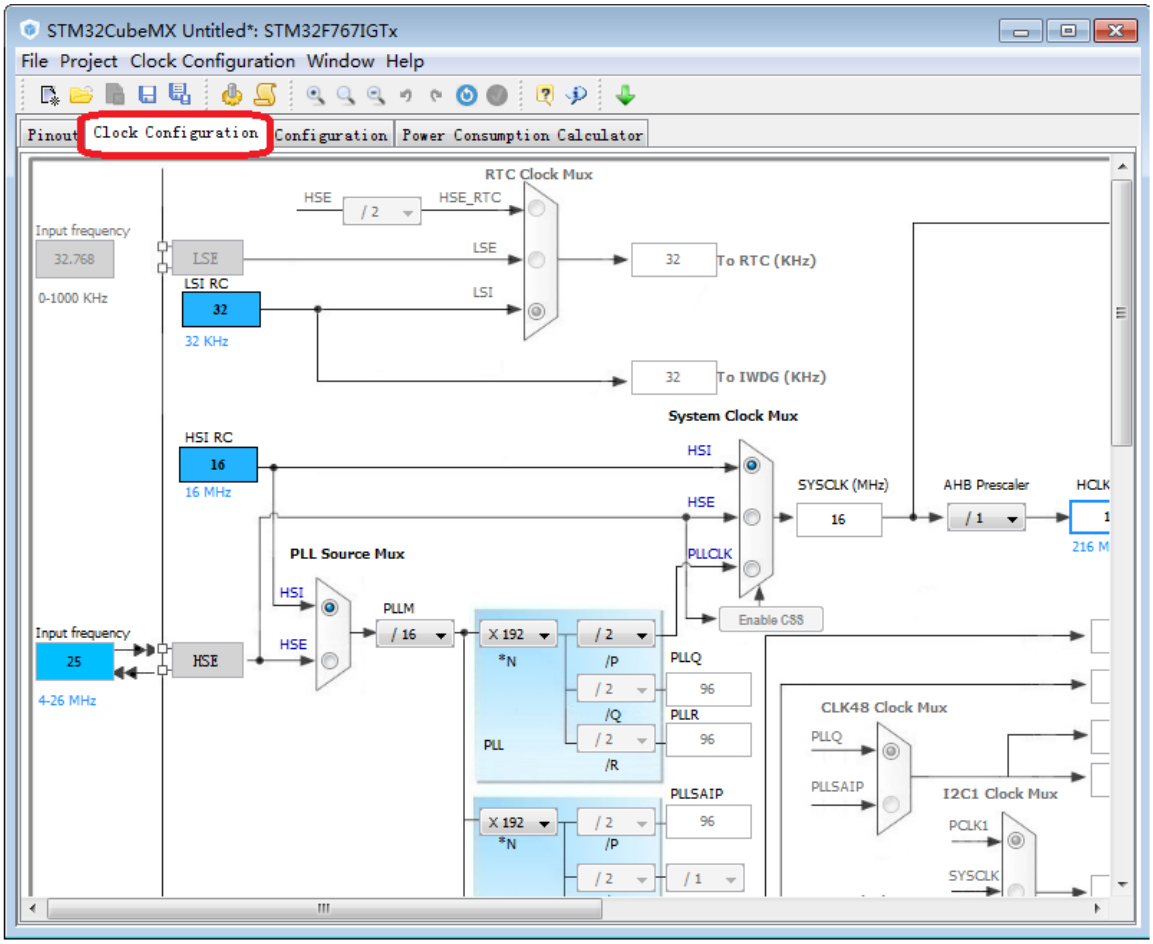


图 4.8.3.3.1 时钟系统配置栏

进入 Clock Configuration 配置栏之后可以看到，界面展现一个完整的 STM32F7 时钟系统框图。这个时钟系统框图跟我们之前时钟系统章节讲解的时钟系统框图实际是一模一样的，只不过调整了一下显示顺序。从这个时钟树配置图可以看出，配置的主要是外部晶振大小，分频系数，倍频系数以及选择器。在我们配置的工程中，时钟值会动态更新，如果某个时钟值在配置过程中超过允许值，那么相应的选项框会红色提示。

这里，我们将配置一个和我们之前讲解的 Stm32_Clock_Init 函数实现的一模一样的配置。Stm32_Clock_Init 函数主要实现的是以 HSE 为时钟源，配置主 PLL 相关参数，然后系统时钟选择 PLL 为时钟源，最终配置系统时钟为 216MHz 的过程。同时，还配置了 AHB，APB1，APB2 和 Systick 的相关分频系数。由于图片比较大，我们把主要的配置部分分两部分来讲解，第一部分是配置系统时钟，第二部分是配置 AHB，APB1 和 APB2 的分频系数。首先我们来看看第一部分配置如下图 4.8.3.3.2 所示：

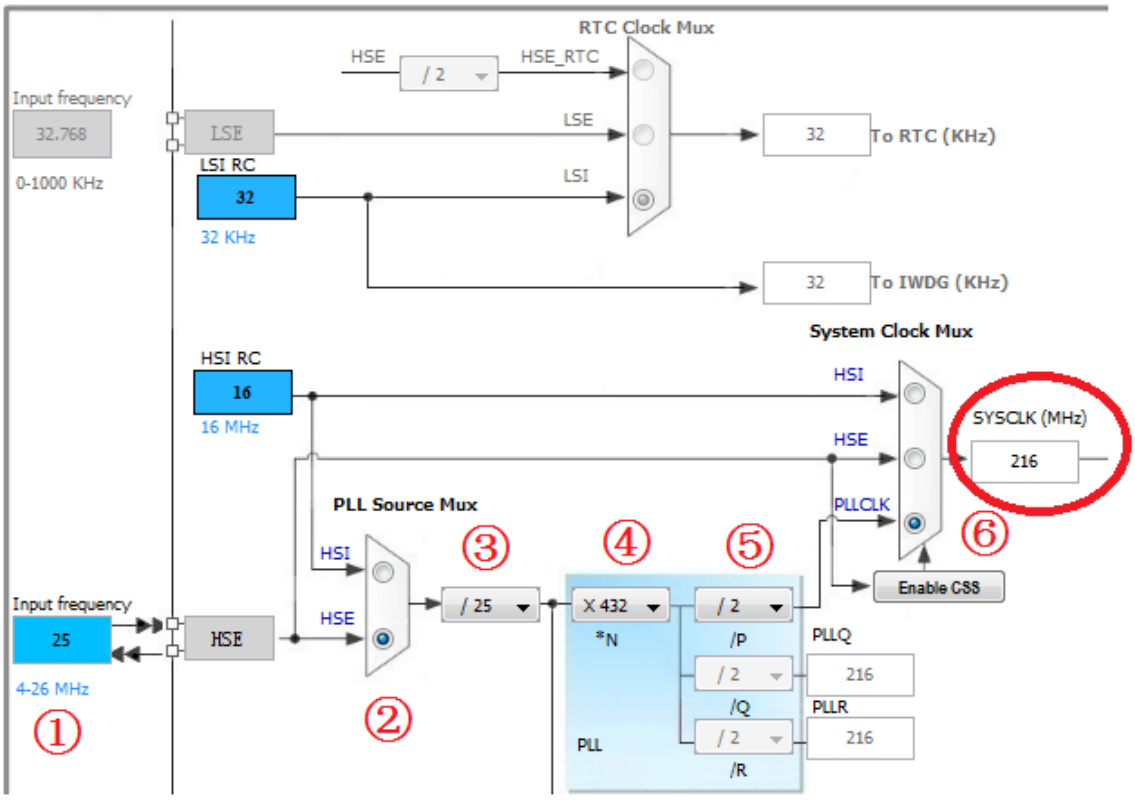


图 4.8.3.3.2 系统时钟配置图

我们把系统时钟配置分为 6 个步骤，分别用标号①~⑥表示，详细过程为：

① 时钟源参数设置：HSE 或者 HSI 配置。这里我们选择 HSE 为时钟源，所以我们之前必须在 RCC 配置中我们开启 HSE。

② 时钟源选择：HSE 还是 HSI。这里我们配置选择器选择 HSE 即可。

③ PLL 分频系数 M 配置。分频系数 M 我们设置为 25。

④ 主 PLL 倍频系数 N 配置。倍频系数 N 我们设置为 432。

⑤ 主 PLL 分频系数 P 配置。分频系数 P 我们配置为 2。

⑥ 系统时钟时钟源选择：PLL,HSI 还是 HSE。这里毫无疑问，我们选择 PLL，选择器选择 PLLCLK 即可。

经过上面的 6 个步骤，就会生成标准的 216MHz 系统时钟。接下来我们只需要配置 AHB，APB1，APB2 和 Systick 的分频系数，就可以完全实现函数 Stm32_Clock_Init 配置的时钟系统。配置如下图 4.8.3.3.3 所示：

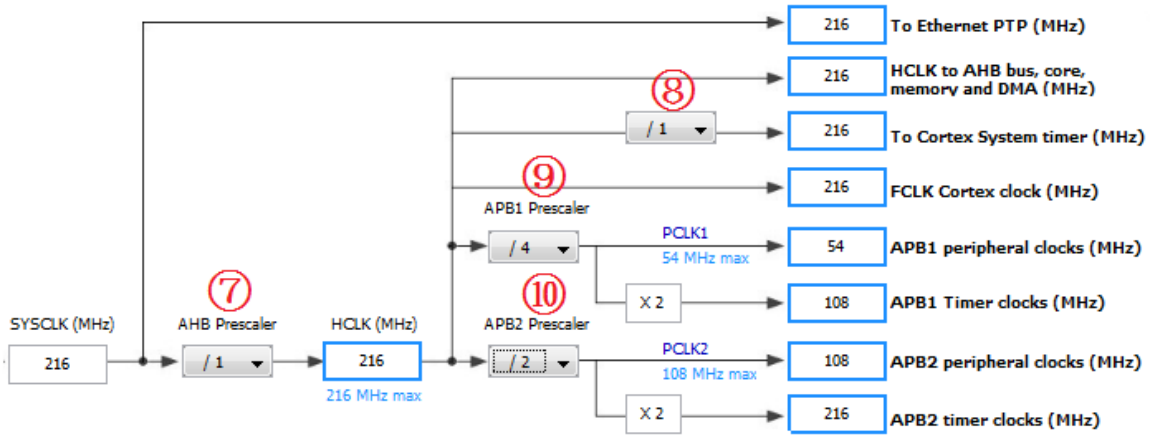


图 4.8.3.3.3 AHB,APB1 和 APB2 总线时钟配置

AHB，APB1 和 APB2 总线时钟以及 Systick 时钟的最终来源都是系统时钟 SYSCLK。其中 AHB 总线时钟 HCLK 是由 SYSCLK 经过 AHB 预分频器之后的来，如果我们要设置 HCLK 为 180MHz，那么我们只需要配置图中标号⑦的地方为 1 即可。得到 HCLK 之后，接下来我们将在图标号⑧~⑩处同样的方法依次配置 Systick， APB 以及 APB2 分频系数分别为 1，4 和 2 即可。配置完成之后，那么 HCLK=216MHZ，Systick 时钟为 216/1MHz=216MHz，PCLK1=216MHz/4=54MHz，PCLK2=216MHz/2=108MHz，这和我们使用 Stm32_Clock_Init 函数配置的时钟是一模一样的。

配置完时钟系统之后，这个时候如果我们直接使用软件生成工程，那么我们就可以从工程中提取系统时钟初始化配置相关代码。配置时钟系统实际上是 STM32CubeMX 一个很重要的功能。为了验证我们工程的正确性，下一小节我们将手把手教大家进行 IO 口配置，配置一个和我们阿波罗 STM32F7 开发板跑马灯实验初始化代码一样的效果。

4.8.3.4 GPIO 功能引脚配置

本小节，我们将讲解怎么使用 STM32CubeMX 工具配置 STM32 的 GPIO 口。在阿波罗 STM32F7 开发板的 PB0 和 PB1 引脚有连接两个 LED 灯，本小节将配置这两个 IO 口的相关参数。STM32CubeMX 可以直接在芯片引脚图上配置 IO 口参数。这里我们回到 STM32CubeMX 的 Pinout 选项，在搜索栏输入 PB0 和 PB1 即可找到 PB0 和 PB1 在引脚图中的位置如下图 4.8.3.4.1 所示：

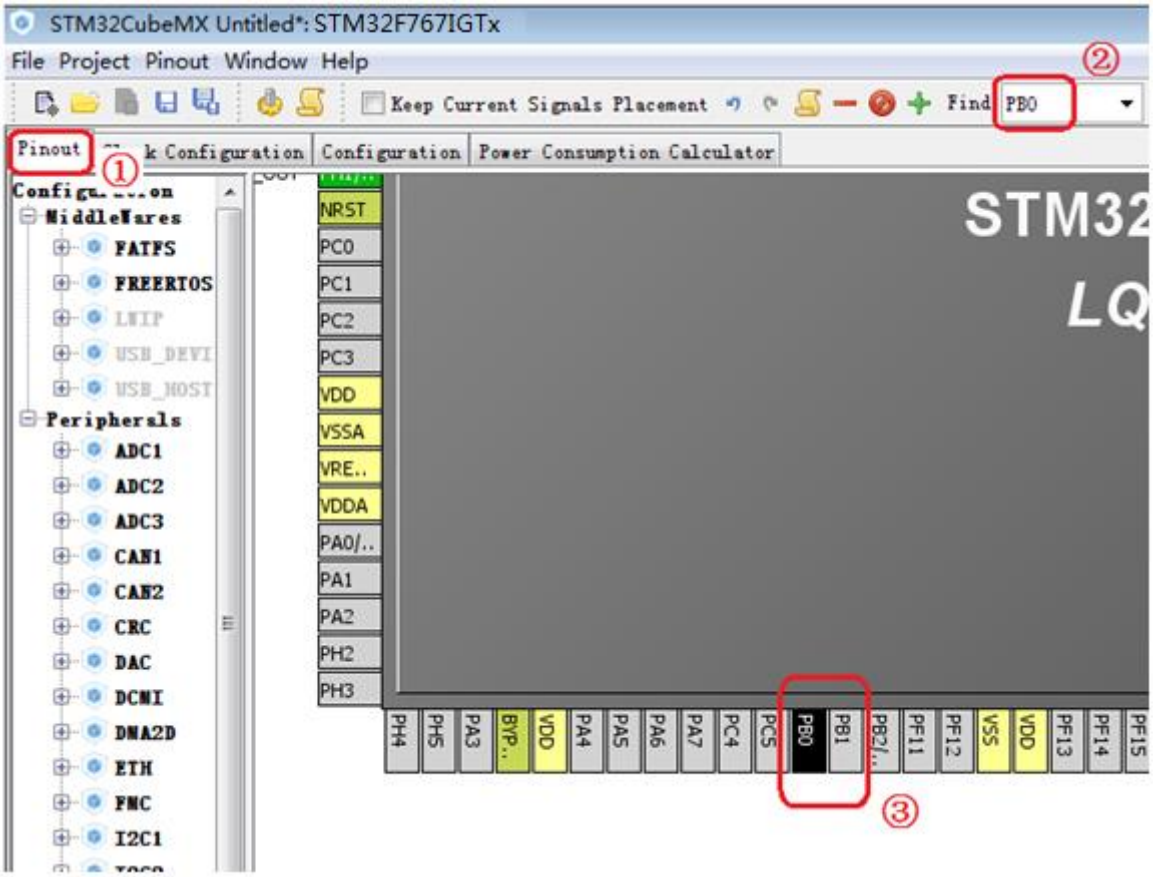


图 4.8.3.4.1 PB0/PB1 引脚位置图

接下来，我们在图 4.8.3.4.1 引脚图中点击 PB0，在弹出的下拉菜单中，选择 IO 口的功能为 GPIO_Output。操作方法如下图 4.8.3.4.2 所示：

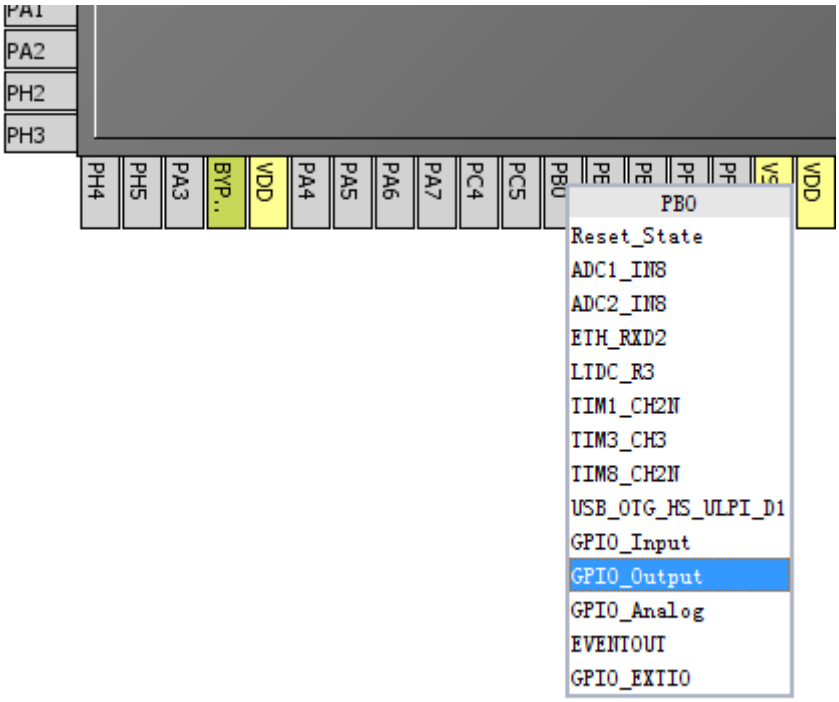


图 4.8.3.4.2 配置 GPIO 模式

同样的方法，我们配置 PB1 选择功能为 GPIO_Oput 即可。这里我们需要说明一下，如果我们要配置 IO 口为外部中断引脚或者其他复用功能，我们选择相应的选项即可。配置完 IO 口功能之后，还要配置 IO 口的速度，上下拉等参数。这些参数是在 Configuration 选项卡中配置的。配置步骤如下图 4.8.3.4.3 所示：

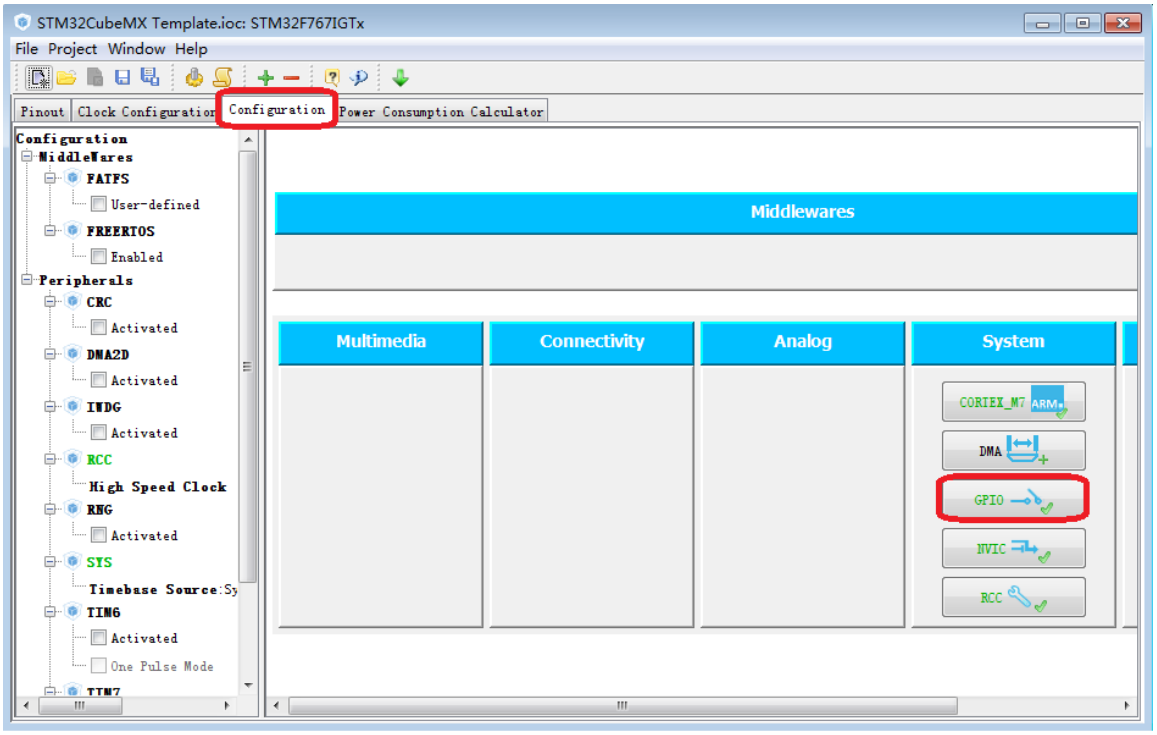


图 4.8.3.4.3 进入 GPIO 详细参数配置界面

依次点击 Configuration->GPIO 即可进入 IO 口详细配置界面， 进入 IO 口配置界面之后，界面会列出所有使用到的 IO 口的参数配置。这里，我们选中 PB0 栏，就会在显示框下方显示对应的 IO 口详细配置信息，然后我们对参数进行配置后点击 Apply 保存即可。配置方法如下图 4.8.3.4.4 所示：

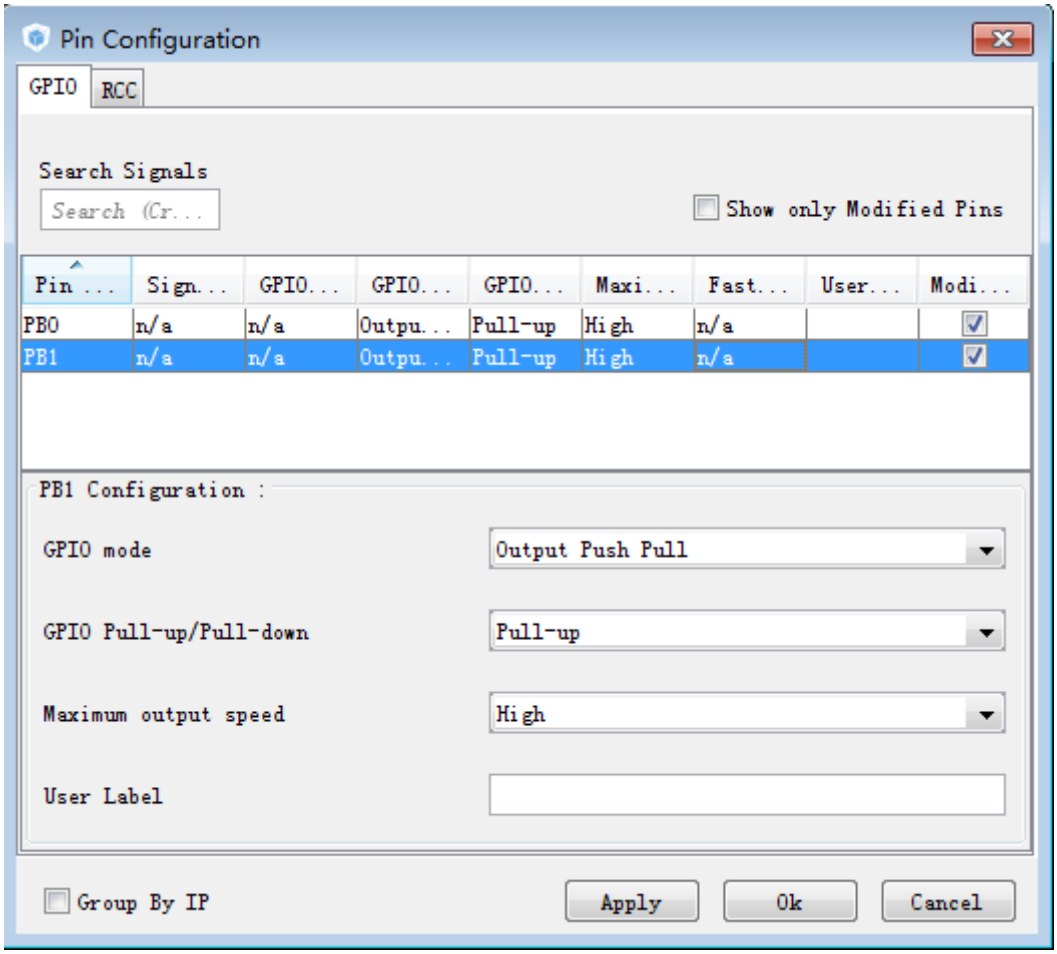


图 4.8.3.4.4 配置 GPIO 口详细参数

这个界面里 IO 参数含义这里我们就不做过多讲解，在大家学习完第一个实验跑马灯实验之后，对参数的含义理解会更加清晰。配置完成之后，我们点击 OK 后界面回到 Configuration 选项卡界面。

对于选项卡 Power Consumption Calculator，它的作用是对功耗进行计算，这里我们并没有使用到，就不详细讲解了。

4.8.3.5 Cortex-M7 内核基本配置

这里我们主要配置 Cortex-M7 内核相关的参数。我们依次点击 Configuration->Cortex_M7 ARM 进入配置界面，操作过程如下图 4.8.3.5.1 和 4.8.3.5.2 所示：

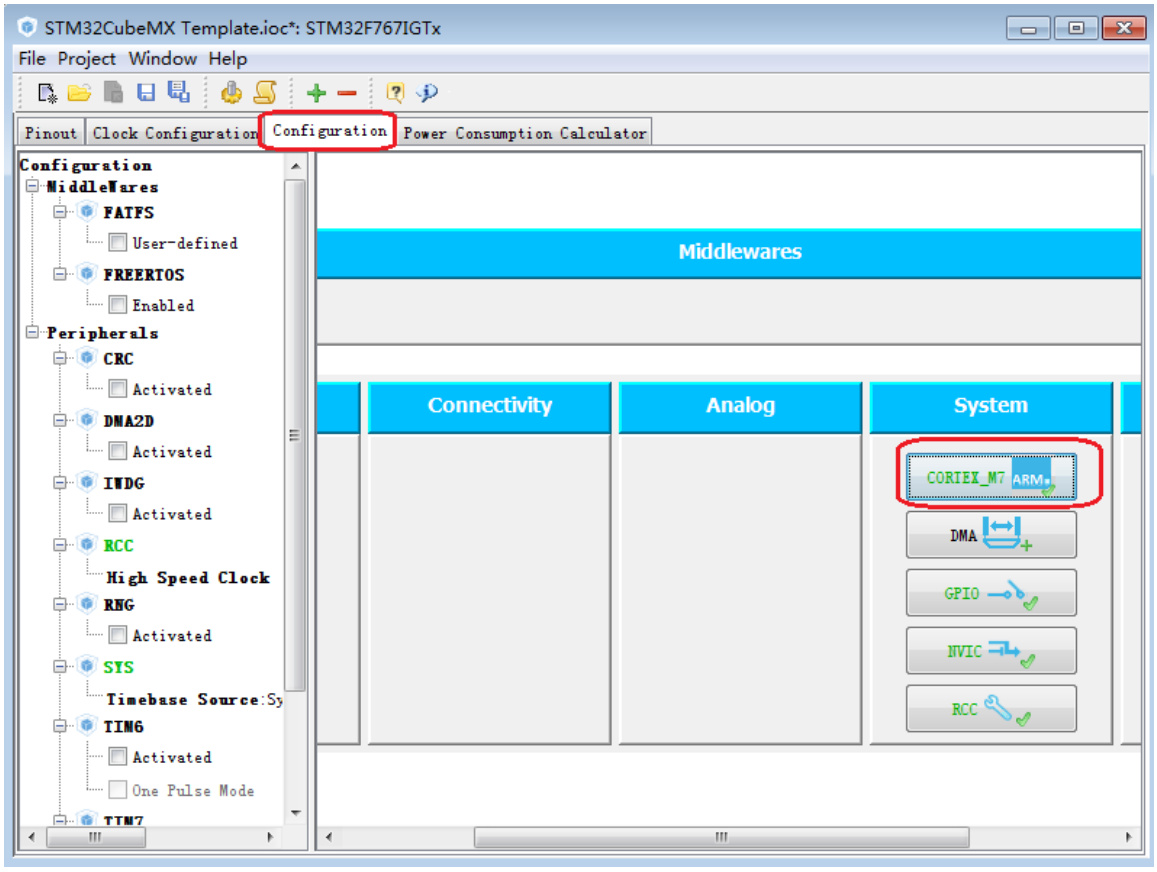


图 4.8.3.5.1 进入 Cortex_M7 ARM 配置界面

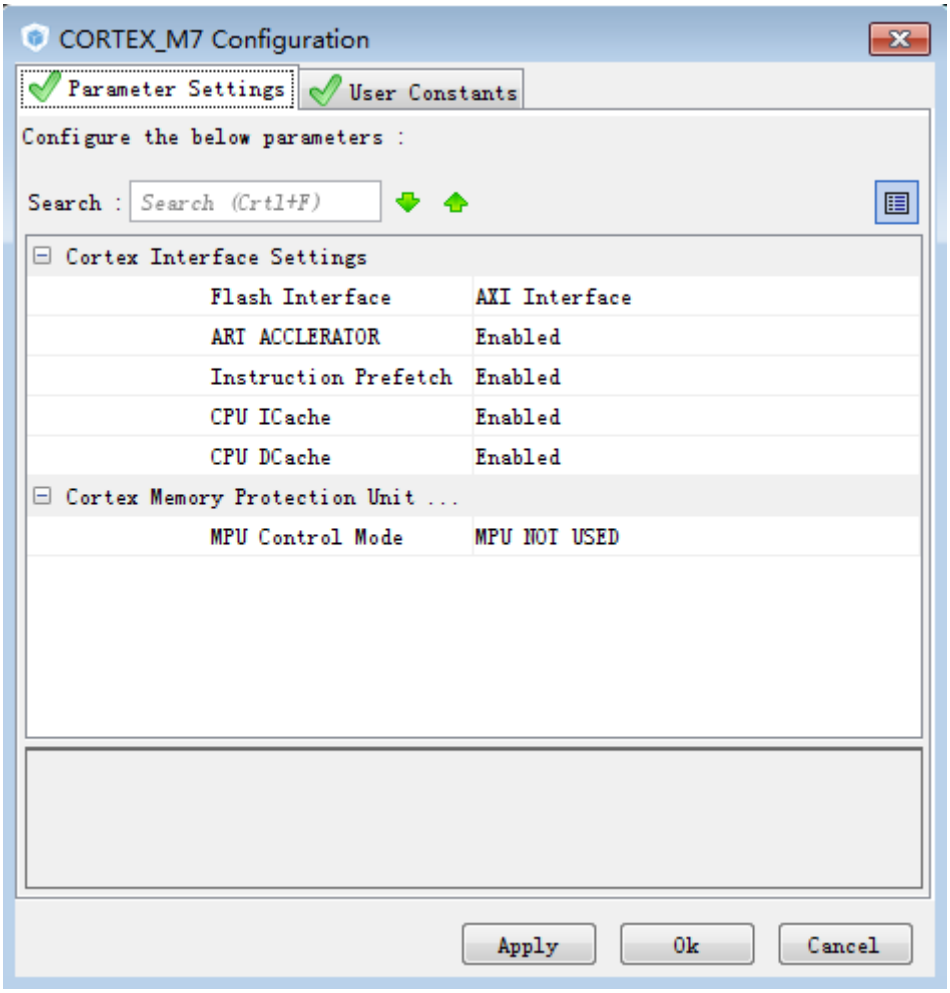


图 4.8.3.5.2 Cotex_M7 配置界面

该界面一共有两个配置栏目。第一个配置栏目 Cortex Interface Settings 下面有五个配置项：

1） Flash Interface：选择 Flash 接口，为 AXI 或者 TCM。

2） ARI ACCLERATOR：使能缓存加速。

3） Instruction Prefetch：使能指令预取。

4） CPU ICache：使能 I-Cache。

5） CPU DCache: 使能 D-Cache。

上面这 5 个参数是 CM7 内核相关配置。第二个配置栏目 Cortex Memory Protection Unit，是用来配置内存保护单元 MPU，在我们后面的实验会讲解 MPU 配置。

4.8.3.6 生成工程源码

经过上面 5 个步骤，一个完整的系统已经配置完成。接下来，我们将使用 STM32CubeMX 生成我们需要的工程源码。在 STM32CubeMX 操作界面，依次点击菜单 Project->Generate Code 即可生成源码，操作方法如下图 4.8.3.6.1 所示：

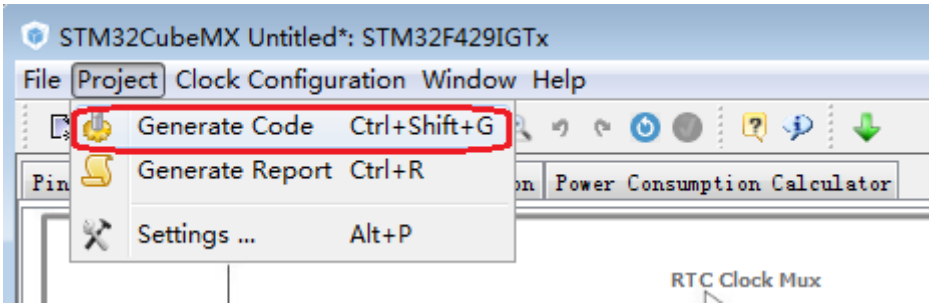


图 4.8.3.6.1 点击 Generate Code 选项

点击之后，弹出的界面会要求配置生成的工程名称，保存目录以及使用的编译软件类型。我们依次填写工程名称和保存目录即可，对于编译软件我们选择 MDK5 即可。操作过程如下图 4.8.3.6.2 所示：

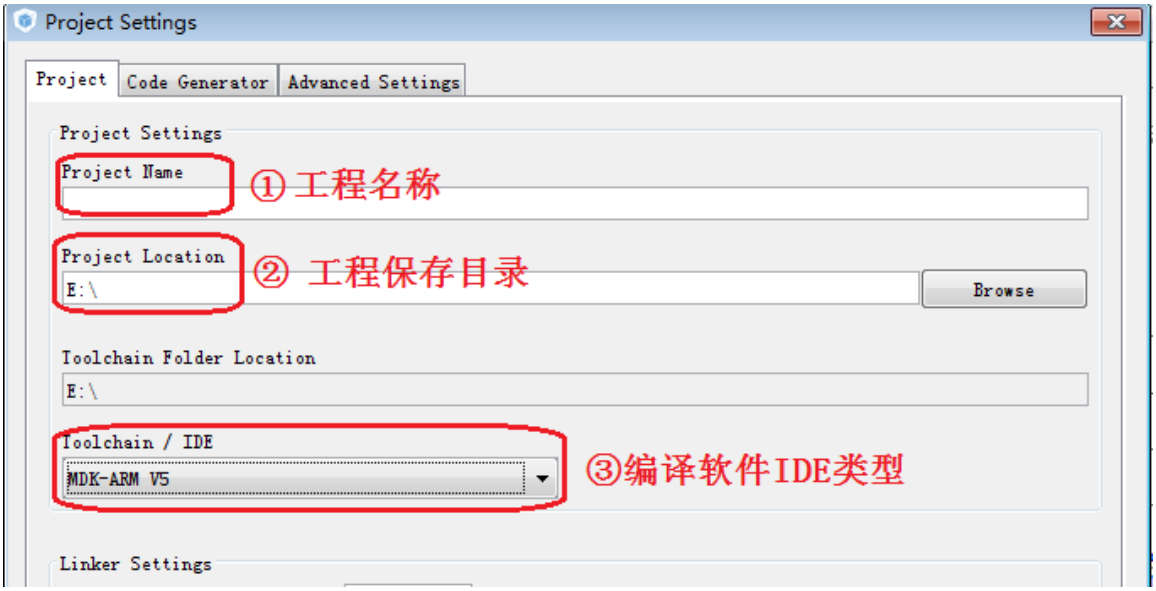


图 4.8.3.6.2 工程参数配置

配置完成后，点击 OK 开始生产源码。源码生产完成之后，就保存在我们 Project Location 选项配置的目录中，同时弹出生成成功提示界面，我们可以点击界面的“Open Folder”按钮打开工程保存目录，也可以点击界面的“Open Project”按钮直接使用 IDE 打开工程。提示界面如下图 4.8.3.6.3 所示：

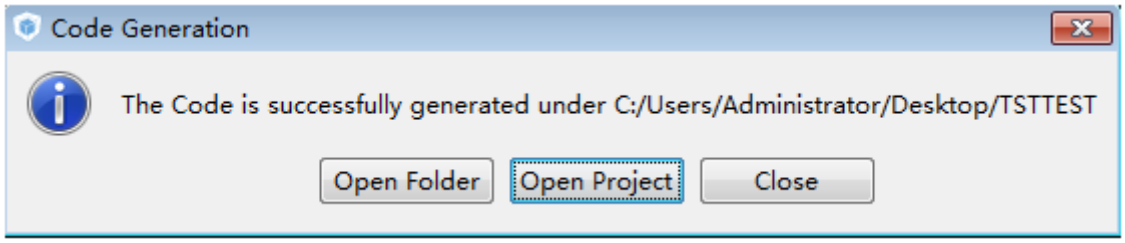


图 4.8.3.6.3 代码生成后提示界面

至此，一个完整的 STM32F7 工程就已经生成完成。生产后的工程目录结构如下图 4.8.3.6.4 所示：

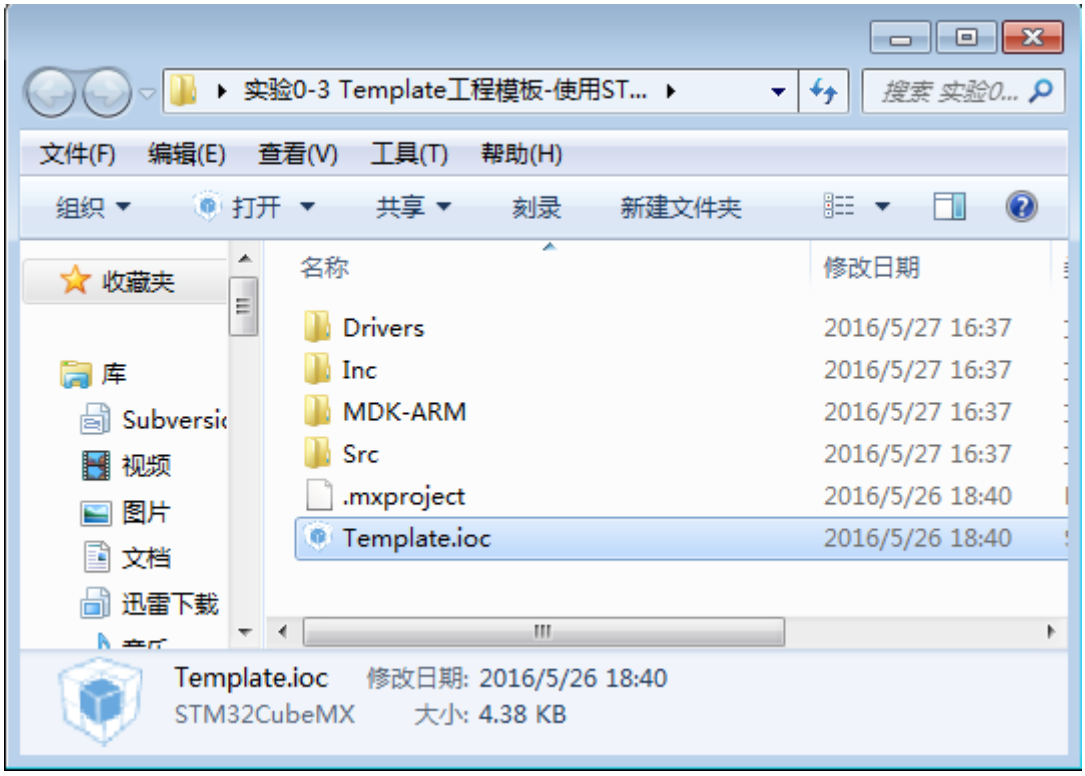


图 4.8.3.6.4 STM32CubeMX 生成的工程目录结构

Drivers 文件夹存放的是 HAL 库文件和 CMSIS 相关文件。

Inc 文件夹存放的是工程必须的部分头文件。

MDK-ARM 下面存放的是 MDK 工程文件。

Src 文件夹下面存放的是工程必须的部分源文件。

Template.ioc 是 STM32CubeMX 工程文件，双击该文件工程就会在 STM32CubeMX 中被打开。

4.8.3.7 编写用户程序

在编写用户程序之前，首先我们打开生成的工程模板进行编译，发现没有任何错误和警告。工程模板结构如下图 4.8.3.7.1 所示：

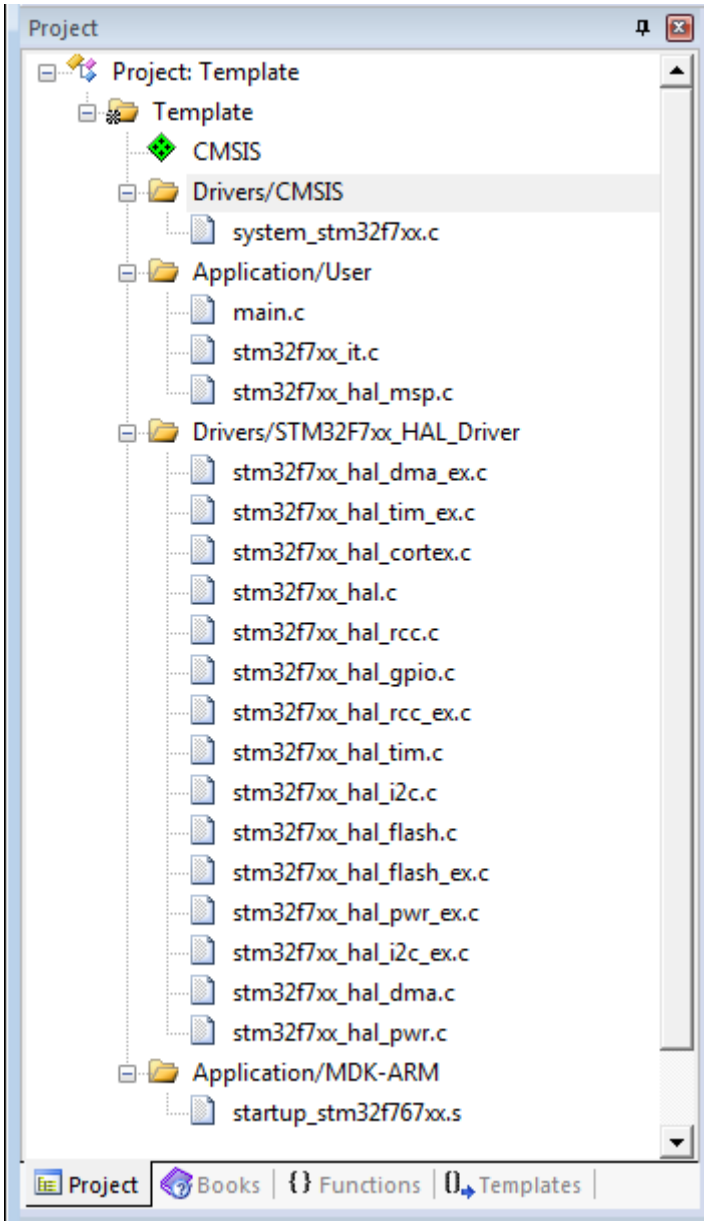


图 4.8.3.7.1 使用 STM32CubeMX 生成的工程模板

该工程模板结构跟我们前面 3.3 小节新建的工程模板实际上是类似的，只不过一些分组名称不一样，同时我们将时钟系统配置源码放在 SYSTEM 分组的 sys.c 中，而该模板直接放在 main.c 源文件中。这里我们对该模板就不做过多讲解。我们直接打开 main.c 源文件可以看到，该文件定义了两个关键函数 SystemClock_Config 和 MX_GPIO_Init，并且在 main 函数中调用了这两个函数。SystemClock_Config 函数用来配置时钟系统，和我们模板中的 Stm32_Clock_Init 函数作用一样。MX_GPIO_Init 函数用来初始化 PB0 和 PB1 相关配置，这在我们的模板中，我们直接放在 main 函数中。接下来我们看看生成的工程模板的 main 函数，这里我们删掉了源码注释，关键源码如下：

```
int main(void)
{
  SCB_EnableICache();
  SCB_EnableDCache();
```

```
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    while (1)
    {
    }
}
```

该函数 while 语句之前实现的功能和我们 3.3 小节的工程模板是一致的。这里，我们直接把我们 3.3 小节新建的工程模板中 main 函数 while 语句中的源码复制到此处的 while 语句中，然后复制 Delay 函数申明和定义到 main 函数之前。**这里大家需要注意，STM32CubeMX 生成的 main.c 文件中，有很多地方有“/* USER CODE BEGIN X */”和“/* USER CODE END X */”格式的注释，我们在这些注释的 BEGIN 和 END 之间编写代码，那么重新生成工程之后，这些代码会保留而不会被覆盖**。复制完代码之后，main 函数关键源码如下：

```
/* USER CODE BEGIN 0 */
void Delay(__IO uint32_t nCount);

void Delay(__IO uint32_t nCount)
{
  while(nCount--){}
}
/* USER CODE END 0 */
int main(void)
{
  SCB_EnableICache();
  SCB_EnableDCache();
  HAL_Init();
  MX_GPIO_Init();
/* USER CODE BEGIN WHILE */
  while (1)
  {
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_SET);      //PB1 置 1
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_SET);      //PB0 置 1
      Delay(0x7FFFFF);
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);  //PB1 置 0
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_RESET);  //PB0 置 0
      Delay(0x7FFFFF);
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

这个时候，我们对工程进行编译，发现没有任何警告和错误。同时，我们使用 3.4 小节的方法下载程序到阿波罗 STM32F7 开发板中（如果使用 ST-LINK 下载，请注意配置 MDK），运

行结果和 3.3 小节新建工程运行结果一模一样。

本小节使用 STM32CubeMX 新建的工程模板在我们光盘目录：**“4，程序源码\标准例程-库函数版本\实验 0-3 Template 工程模板-使用 STM32CubeMX 配置 ”**中有存放，大家在编写用户代码过程中可以参考该工程的 main.c 文件。

这里我们需要说明一下，**大多数情况下，我们使用 STM32CubeMX 主要用来配置时钟系统和外设初始化代码。**这里我们讲解新建一个工程模板，是为了系统全面的讲解 STM32CubeMX 生成工程的步骤。本小节就给大家讲解到这里。通过本章的学习，大家对 STM32CubeMX 的使用就有了初步的认识和理解，后面我们会在实战篇通过几个实验来巩固这方面知识。

第五章 SYSTEM 文件夹介绍

第三章，我们介绍了如何在 MDK5 下建立 STM32F7 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 STM32F7xx 系列的底层核心驱动函数，可以用在 STM32F7xx 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 STM32F7 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 3 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，我们先了解一下编程思想：CM4 内核的处理和 CM3 一样，内部都包含了一个 SysTick 定时器，SysTick 是一个 24 位的倒计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick 在《STM32F7 中文参考手册》里面基本没有介绍，其详细介绍，请参阅《STM32F7 编程手册》第 211 页，4.4 节。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*216（假设系统时钟为 216Mhz，因为我们设置 systick 的频率为系统时钟频率，那么 systick 每增加 1，就是 1/216us），然后我们就一直统计 systick 的计数变化，直到这个值变化了 50*216，一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
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 180M 下,约合 0.745s 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 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_m4.h 里面），里面包含 CTRL、LOAD、VAL、CALIB 等 4 个寄存器，

SysTick->CTRL 的各位定义如图 5.1.2.1 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。

SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);这句代码把 SysTick 的时钟选择为内核时钟，这里需要注意的是：SysTick 的时钟源自 HCLK，假设我们外部晶振为 25M，然后倍频到 216MHZ，那么 SysTick 的时钟即为 216Mhz，也就是 SysTick 的计数器 VAL 每减 1，就代表时间过了 1/216us。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，这里我们首先介绍不使用 OS 的时候，实现函数如下：

```
//延时 nus
//nus 为要延时的 us 数.
//nus:0~204522252(最大值即 2^32/fac_us@fac_us=21)
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.
        }
    };
}
```

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

该函数是用来延时指定的 ms 的，其参数 nms 为要延时的毫秒数。该函数有使用 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_osticspersec=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 stm32f7xx_hal.c 中定义
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
  /*配置 1ms 中断一次*/
  HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
  HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority ,0);
  return HAL_OK;
}

//HAL 库的 SYSTICK 配置函数：文件 stm32f7xx_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_cm4.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 stm32f7xx_it.c 中
void SysTick_Handler(void)
{
  HAL_IncTick();
}

//下面代码均在文件 stm32f7xx_hal.c 中
static __IO uint32_t uwTick; //定义计数全局变量

__weak void HAL_IncTick(void) //全局变量 uwTick 递增
{
  uwTick++;
}

__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 = 0;
  tickstart = HAL_GetTick();
  while((HAL_GetTick() - tickstart) < Delay)
  {
  }
}
```

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 里面除了函数申明外主要是定义了一些常用数据类型短关键字。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以及 Cache 相关操作函数，对于函数 Stm32_Clock_Init 的讲解请参考本手册 4.3 小节 STM32F7 时钟系统章节内容。接下来我们看看 STM32F7 的 Cache 使能函数。

5.2.1 Cache 使能函数

STM32F7 自带了指令 Cache（I Cache）和数据 Cache（D Cache），使用 I/D Cache 可以缓存指令/数据，提高 CPU 访问指令/数据的速度，从而大大提高 MCU 的性能。不过，MCU 在复位后，I/D Cache 默认都是关闭的，为了提高性能，我们需要开启 I/D Cache，在 sys.c 里面，我们提供了如下函数：

```
//使能 STM32F7 的 L1-Cache,同时开启 D cache 的强制透写
void Cache_Enable(void)
{
    SCB_EnableICache();   //使能 I-Cache,函数在 core_cm7.h 里面定义
    SCB_EnableDCache();   //使能 D-Cache,函数在 core_cm7.h 里面定义
    SCB->CACR|=1<<2;      //强制 D-Cache 透写,如不开启,实际使用中可能遇到各种问题
}
```

该函数，通过调用 SCB_EnableICache 和 SCB_EnableDCache 这两个函数来使能 I Cache 和 D Cache。不过，在使能 D Cache 之后，SRAM 里面的数据有可能会被缓存在 Cache 里面，此时如果有 DMA 之类的外设访问这个 SRAM 里面的数据，就有可能和 Cache 里面数据不同步，导致数据出错，为了防止这种问题，保证数据的一致性，我们设置了 D Cache 的强制透写功能（Write Through），这样 CPU 每次操作 Cache 里面的数据，同时也会更新到 SRAM 里面，保证 D Cache 和 SRAM 里面数据一致。关于 Cache 的详细介绍，请参考《STM32F7 Cache Oveview》和《Level 1 cache on STM32F7 Series》（见光盘：8，STM32 参考资料 文件夹）。

这里 SCB_EnableICache 和 SCB_EnableDCache 这两个函数，是在 core_cm7.h 里面定义的，我们直接调用即可，另外，core_cm7.h 里面还提供了以下五个常用函数：

1，SCB_DisableICache 函数，用于关闭 I Cache。

2，SCB_DisableDCache 函数，用于关闭 D Cache。

3，SCB_InvalidateDCache 函数，用于丢弃 D Cache 当前数据，重新从 SRAM 获取数据。

4，SCB_CleanDCache 函数，用于将 D Cache 数据回写到 SRAM 里面，同步数据。

5，SCB_CleanInvalidateDCache 函数，用于回写数据到 SRAM，并重新获取 D Cache 数据。

在 Cache_Enable 函数里面，我们直接开启了 D Cache 的透写模式，这样带来的好处就是可以保证 D Cache 和 SRAM 里面数据的一致性，坏处就是会损失一定的性能（每次都要回写数据），如果大家想自己控制 D Cache 数据的回写，以获得最佳性能，则可以关闭 D Cache 透写模式，并在适当的时候，调用 SCB_CleanDCache、SCB_InvalidateDCache 和 SCB_CleanInvalidateDCache 等函数，这对程序员的要求非常高，程序员必须清楚什么时候该回写，什么时候该更新 D Cache！如果能力不够，还是建议开启 D Cache 的透写，以免引起各种莫名其妙的问题。

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 STM32F7 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 STM32F7。

STM32F7 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 STM32F7 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 STM32F7 的各种外设，通过本篇的学习，希望大家能学会 STM32F7 绝大部分外设的使用。

本篇总共分为 59 章，每一章即一个实例，下面就让我们开始精彩的 STM32F7 之旅。

第六章 跑马灯实验

任何一个单片机，最简单的操作莫过于 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 STM32F7 之旅，通过本章的学习，你将了解到 STM32F7 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 阿波罗 STM32 开发板上的两个 LED 灯 DS0 和 DS1 交替闪烁，实现类似跑马灯的效果。 本章分为如下五个小节：

6.1 STM32F7 IO 口简介
6.2 硬件设计
6.3 软件设计
6.4 下载验证
6.5 STM32CubeMX 配置 IO 口输入

6.1 STM32F7 IO 简介

本章将要实现的是控制 ALIENTEK 阿波罗 STM32 开发板上的两个 LED 实现一个类似跑马灯的效果，该实验的关键在于如何控制 STM32F7 的 IO 口输出。了解了 STM32F7 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 STM32F7 基本 IO 口的使用，而这是迈向 STM32F7 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 STM32F7 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-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：

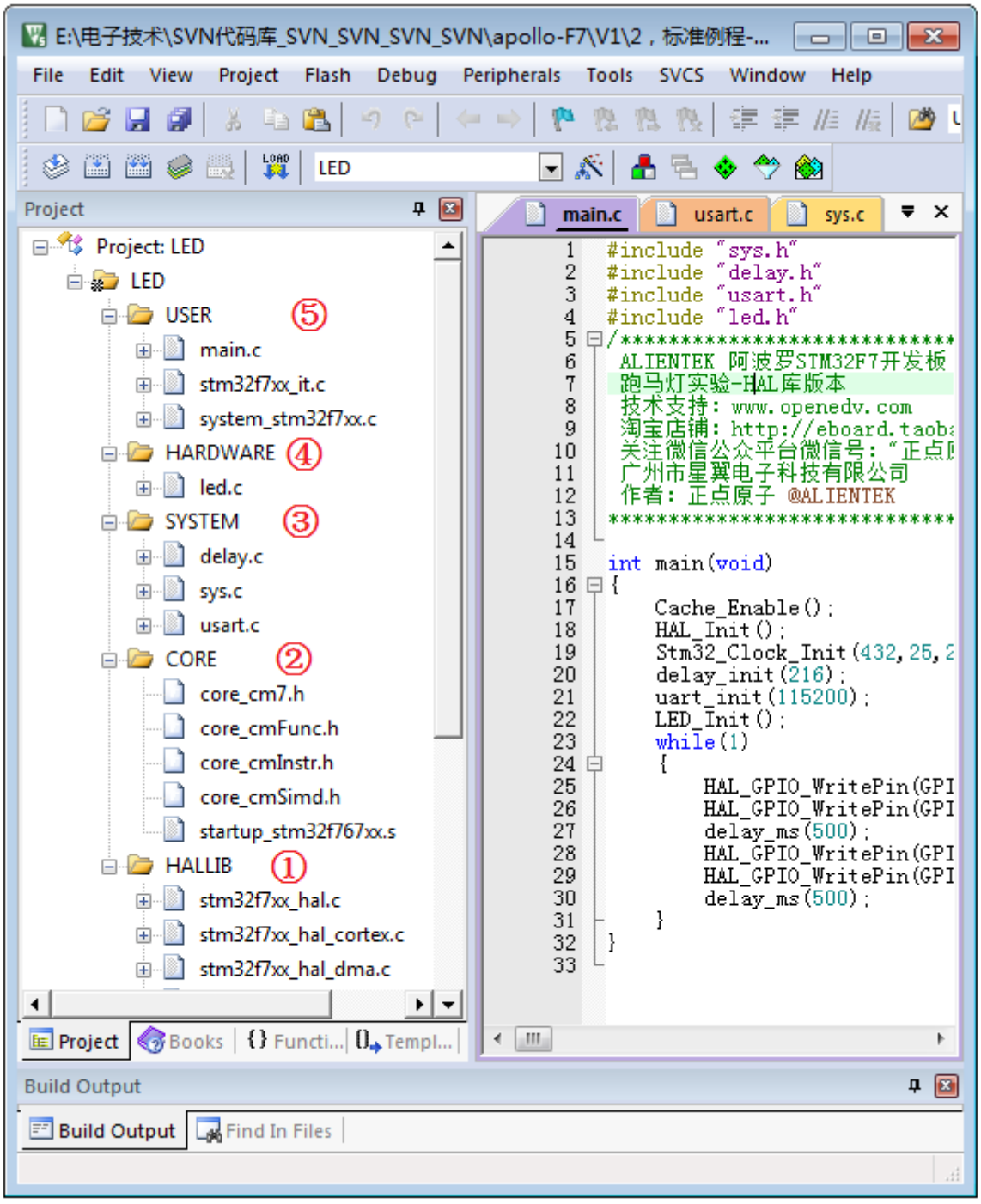


图 6.1.1 跑马灯实验目录结构

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 stm32f7xx_hal_ppp.c 都对应一个头文件 stm32f7xx_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 11 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 stm32f7xx_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_stm32f7xx.c 文件用户不需要修改，同时 stm32f7xx_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。

相对于 STM32F1 来说，STM32F7 的 GPIO 设置显得更为复杂，也更加灵活，尤其是复用功能部分，比 STM32F1 改进了很多，使用起来更加方便。

STM32F7 每组通用 I/O 端口包括 4 个 32 位配置寄存器（MODER、OTYPER、OSPEEDR 和 PUPDR）、2 个 32 位数据寄存器（IDR 和 ODR）、1 个 32 位置位/复位寄存器 (BSRR)、1 个 32 位锁定寄存器 (LCKR) 和 2 个 32 位复用功能选择寄存器（AFRH 和 AFRL）等。

这样，STM32F7 每组 IO 有 10 个 32 位寄存器控制，其中常用的有 4 个配置寄存器+2 个数据寄存器+2 个复用功能选择寄存器，共 8 个，如果在使用的时候，每次都直接操作寄存器配置 IO，代码会比较多，也不容易记住，所以我们在讲解寄存器的同时会讲解是用库函数配置 IO 的方法。

同 STM32F1 一样，STM32F7 的 IO 可以由软件配置成如下 8 种模式中的任何一种：

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

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

首先看 MODER 寄存器，该寄存器是 GPIO 端口模式控制寄存器，用于控制 GPIOx（STM32F7 最多有 9 组 IO，分别用大写字母表示，即 x=A/B/C/D/E/F/G/H/I，下同）的工作模式，该寄存器各位描述如表表 6.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	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
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

MODERy[1:0]：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 方向模式。
00：输入（复位状态）
01：通用输出模式
10：复用功能模式
11：模拟模式

表 6.1.2 GPIOx MODER 寄存器各位描述

该寄存器各位在复位后，一般都是 0（个别不是 0，比如 JTAG 占用的几个 IO 口），也就是默认条件下一般是输入状态的。每组 IO 下有 16 个 IO 口，该寄存器共 32 位，每 2 个位控制 1 个 IO，不同设置所对应的模式见表 6.1.1 描述。

然后看 OTYPER 寄存器，该寄存器用于控制 GPIOx 的输出类型，该寄存器各位描述见表表 6.1.3 所示：

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

位 31:16 保留，必须保持复位值。

位 15:0 **OTy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 端口的输出类型。
0：输出推挽（复位状态）
1：输出开漏

表 6.1.3 GPIOx OTYPER 寄存器各位描述

该寄存器仅用于输出模式，在输入模式（MODER[1:0]=00/11 时）下不起作用。该寄存器低 16 位有效，每一个位控制一个 IO 口。设置为 0 是推挽输出，设置为 1 是开漏输出。复位后，该寄存器值均为 0，也就是在输出模式下 IO 口默认为推挽输出。

然后看 OSPEEDR 寄存器，该寄存器用于控制 GPIOx 的输出速度，该寄存器各位描述见表表 6.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7 [1:0]		OSPEEDR6 [1:0]		OSPEEDR5 [1:0]		OSPEEDR4 [1:0]		OSPEEDR3 [1:0]		OSPEEDR2 [1:0]		OSPEEDR1 [1:0]		OSPEEDR0 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 2y+1:2y **OSPEEDRy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 输出速度。
00：低速
01：中速
10：快速
11：高速

表 6.1.4 GPIOx OSPEEDR 寄存器各位描述

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

然后看 PUPDR 寄存器，该寄存器用于控制 GPIOx 的上拉/下拉，该寄存器各位描述见表表 6.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

PUPDRy[1:0]：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 上拉或下拉。
00：无上拉或下拉
01：上拉
10：下拉
11：保留

表 6.1.5 GPIOx PUPDR 寄存器各位描述

该寄存器每 2 个位控制一个 IO 口，用于设置上下拉，这里提醒大家，STM32F1 是通过 ODR 寄存器控制上下拉的，而 STM32F7 则由单独的寄存器 PUPDR 控制上下拉，使用起来更加灵活。复位后，该寄存器值一般为 0。

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

GPIO 相关的函数和定义分布在 HAL 库文件 stm32f7xx_hal_gpio.c 和头文件 stm32f7xx_hal_gpio.h 文件中。

在 HAL 库中，操作四个配置寄存器初始化 GPIO 是通过 HAL_GPIO_Init 函数完成：

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

该函数有两个参数，第一个参数是用来指定需要初始化的 GPIO 对应的 GPIO 组，取值范围为 GPIOA~GPIOK。第二个参数为初始化参数结构体指针，结构体类型为 GPIO_InitTypeDef。下面我们看看这个结构体的定义。首先我们打开我们光盘的跑马灯实验，然后找到 HALLIB 组下面的 stm32f7xx_hal_gpio.c 文件，定位到 HAL_GPIO_Init 函数体处，双击入口参数类型 GPIO_InitTypeDef 后右键选择“Go to definition of …”可以查看结构体的定义如下：

```
typedef struct
{
  uint32_t Pin;        //指定 IO 口
  uint32_t Mode;     //模式设置
  uint32_t Pull;      //上下拉设置
  uint32_t Speed;     //速度设置
  uint32_t Alternate;//复用映射配置
}GPIO_InitTypeDef;
```

结构体有 5 个成员变量，关于怎么来确定这 5 个成员变量的取值范围，请参考 4.7 小节内容。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。

初始化 GPIO 的常用格式是：

```
GPIO_InitTypeDef    GPIO_Initure;
```

```
GPIO_Initure.Pin=GPIO_PIN_0;          //PB0
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(GPIOB,&GPIO_Initure);
```

上面代码的意思是设置 PB0 端口为推挽输出模式，输出速度为高速，上拉。

从上面初始化代码可以看出，结构体 GPIO_Initure 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。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，这个变量实际配置的是我们前面讲解的 GPIOx 的 MODER 寄存器。第三个成员变量 Pull 是用来设置上拉还是下拉，配置的是 GPIOx PUPDR 寄存器。第四个成员变量 Speed 用来设置输出速度，配置的是 GPIOx OSPEEDR 寄存器。第五个成员变量 Alternate，我们在 4.4 小节引脚复用器和映射已经讲解，它是用来设置引脚的复用映射的。

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，我们也不会再重复讲解定位每个参数取值范围的方法。

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

首先我们看 ODR 寄存器，该寄存器用于控制 GPIOx 的输出电平，该寄存器各位描述见表 6.1.6 所示：

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

位 31:16 保留，必须保持复位值。

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

表 6.1.6 GPIOx ODR 寄存器各位描述

该寄存器用于设置某个 IO 输出低电平(ODRy=0)还是高电平(ODRy=1)，该寄存器也仅在输出模式下有效，在输入模式（MODER[1:0]=00/11 时）下不起作用。该寄存器在 HAL 库中使用不多，操作这个寄存器的库函数主要是 HAL_GPIO_TogglePin 函数：

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

该函数是通过操作 ODR 寄存器，达到取反 IO 口输出电平的功能。

接下来我们看看另一个非常重要的寄存器 BSRR，它叫置位/复位寄存器。该寄存器和 ODR 寄存器具有类似的作用，都可以用来设置 GPIO 端口的输出位是 1 还是 0。寄存器描述如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位 31:16 **BRy**：端口 x 复位位 y (Port x reset bit y) (y = 0..15)
这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。
0：不会对相应的 ODRx 位执行任何操作
1：对相应的 ODRx 位进行复位
注意：如果同时对 BSx 和 BRx 置位，则 BSx 的优先级更高。

位 15:0 **BSy**：端口 x 置位位 y (Port x set bit y) (y= 0..15)
这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。
0：不会对相应的 ODRx 位执行任何操作
1：对相应的 ODRx 位进行置位

表 6.1.7 BSRR 寄存器各位描述

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

BSRR 寄存器使用方法如下：

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

库函数操作 BSRR 寄存器来设置 IO 电平的函数为：

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

该函数用来设置一组 IO 口中的一个或者多个 IO 口的电平状态。比如我们要设置 GPIOB.5 输出高，方法为：

```
HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_SET); //GPIOB.5 输出高
```

设置 GPIOB.5 输出低电平，方法为：

```
HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5, GPIO_PIN_RESET);  //GPIOB.5 输出低
```

接下来我们看看 IDR 寄存器，该寄存器用于读取 GPIOx 的输入数据，该寄存器各位描述见表 6.1.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留，必须保持复位值。

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

表 6.1.8 GPIOx IDR 寄存器各位描述

该寄存器用于读取某个 IO 的电平，如果对应的位为 0(IDRy=0)，则说明该 IO 输入的是低电平，如果是 1(IDRy=1)，则表示输入的是高电平。HAL 库操作该寄存器读取 IO 输入数据相关函数：

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

该函数用来读取一组 IO 下一个或者多个 IO 口电平状态。比如我们要读取 GPIOF.5 的输入电平，方法为：

```
HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_5);//读取 PF5 的输入电平
```

该函数返回值就是 IO 口电平状态。

最后我们来看看 2 个 32 位复用功能选择寄存器（AFRH 和 AFRL），这两个寄存器是用来设置 IO 口的复用功能的。实际上，在我们调用函数 HAL_GPIO_Init 的时候，如果我们设置了初始化结构体成员变量 Mode 为复用模式，同时设置了 Alternate 的值，那么会在该函数内部自动设置这两个寄存器的值，达到设置端口复用映射的目的。关于这两个寄存器的详细配置以及相关库函数的使用，在我们前面 4.4 小节 IO 引脚复用和映射有详细讲解，这里我们只是简要说明一下。

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

1） 使能 IO 口时钟，调用函数为__HAL_RCC_GPIOX_CLK_ENABLE(其中 X=A~K)。

2） 初始化 IO 参数。调用函数 HAL_GPIO_Init();

3） 操作 IO 输入输出。操作 IO 的方法就是上面我们讲解的方法。

上面我们讲解了 STM32F7 IO 口的基本知识以及 HAL 库操作 GPIO 的一些函数方法，下面我们来讲解我们的跑马灯实验的硬件和软件设计。

6.2 硬件设计

本章用到的硬件只有 LED（DS0 和 DS1）。其电路在 ALIENTEK 阿波罗 STM32 开发板上默认是已经连接好了的。DS0 接 PB1，DS1 接 PB0。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：

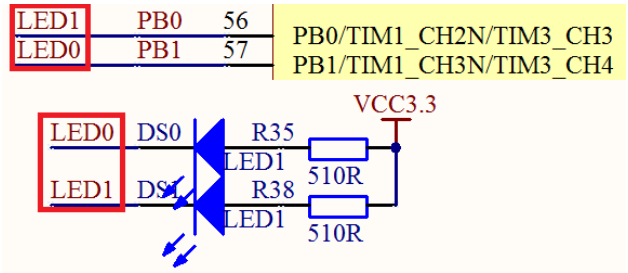


图 6.2.1 LED 与 STM32F767 连接原理图

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-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：

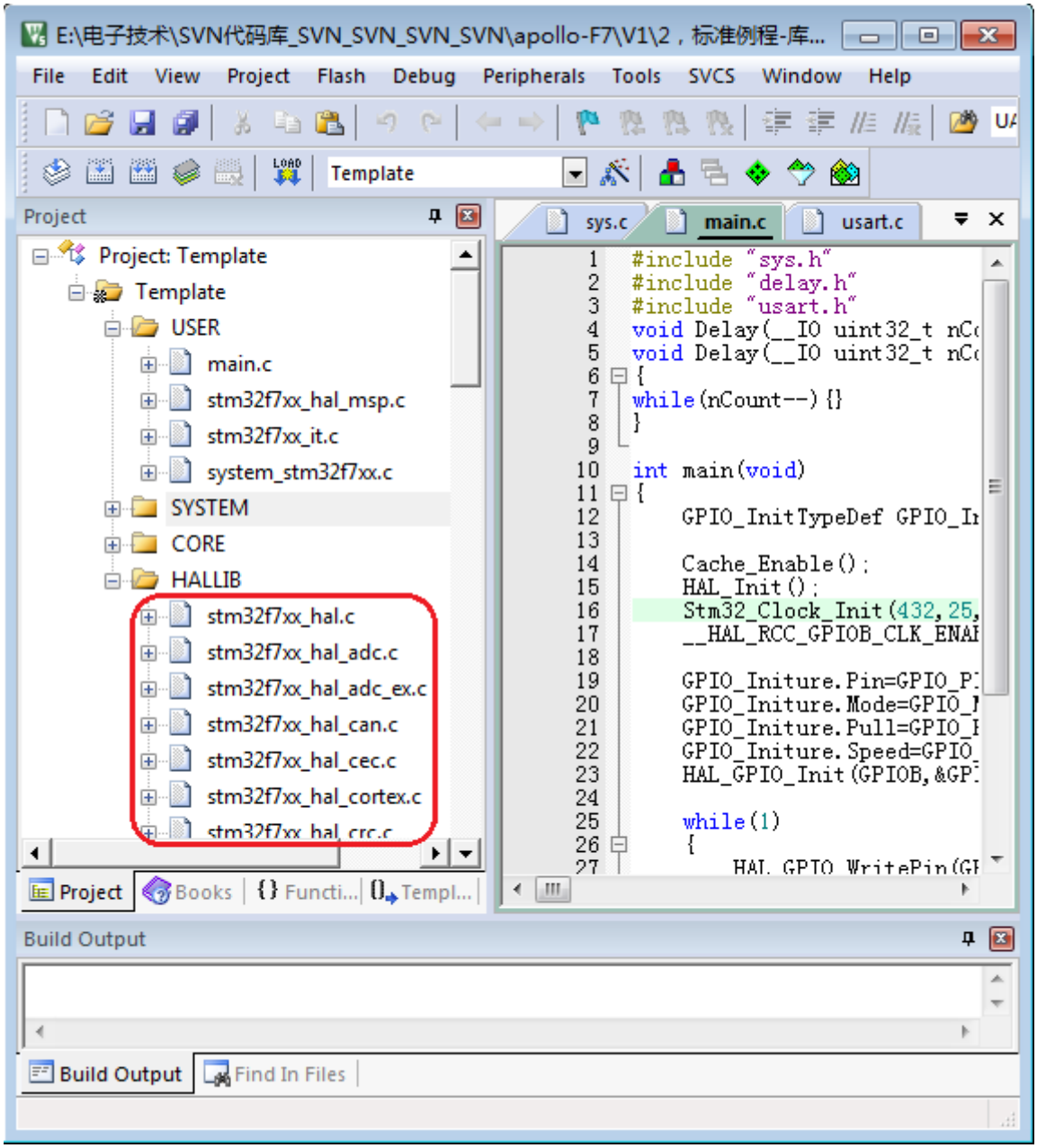


图 6.3.1　Template 模板工程结构

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 stm32f7xx_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 11 个源文件，具体哪 11 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：

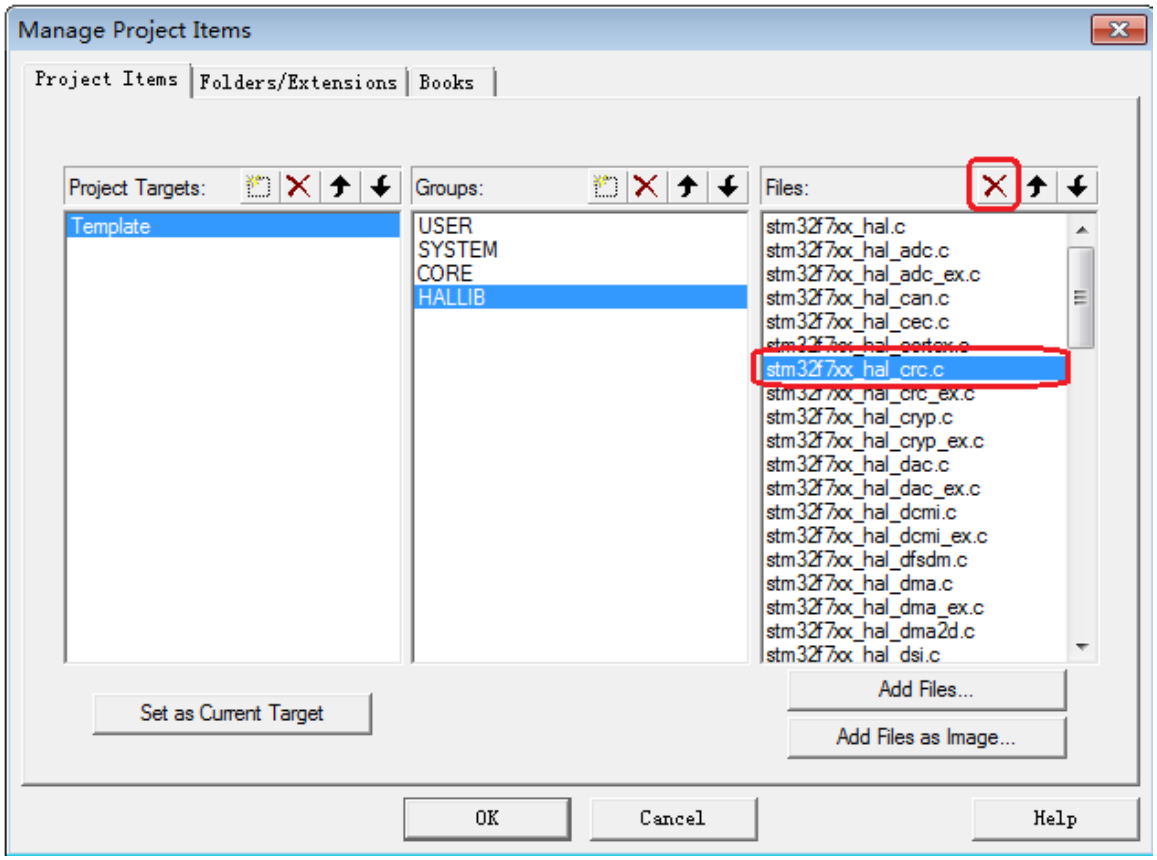


图 6.3.2 删除工程分组中的文件

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：

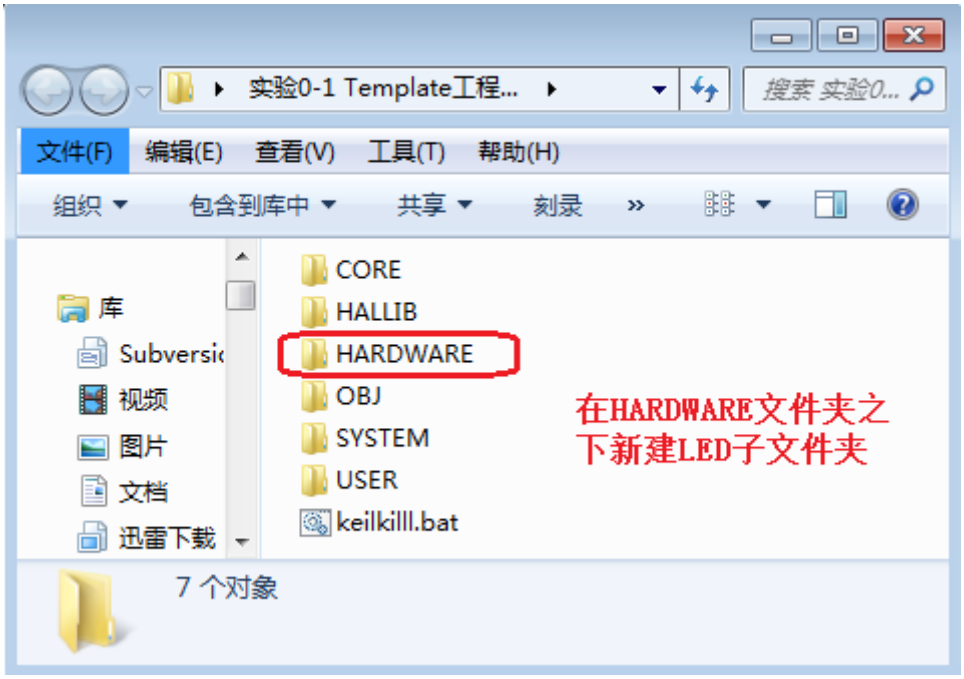


图 6.3.3 新建 HARDWARE 文件夹

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：

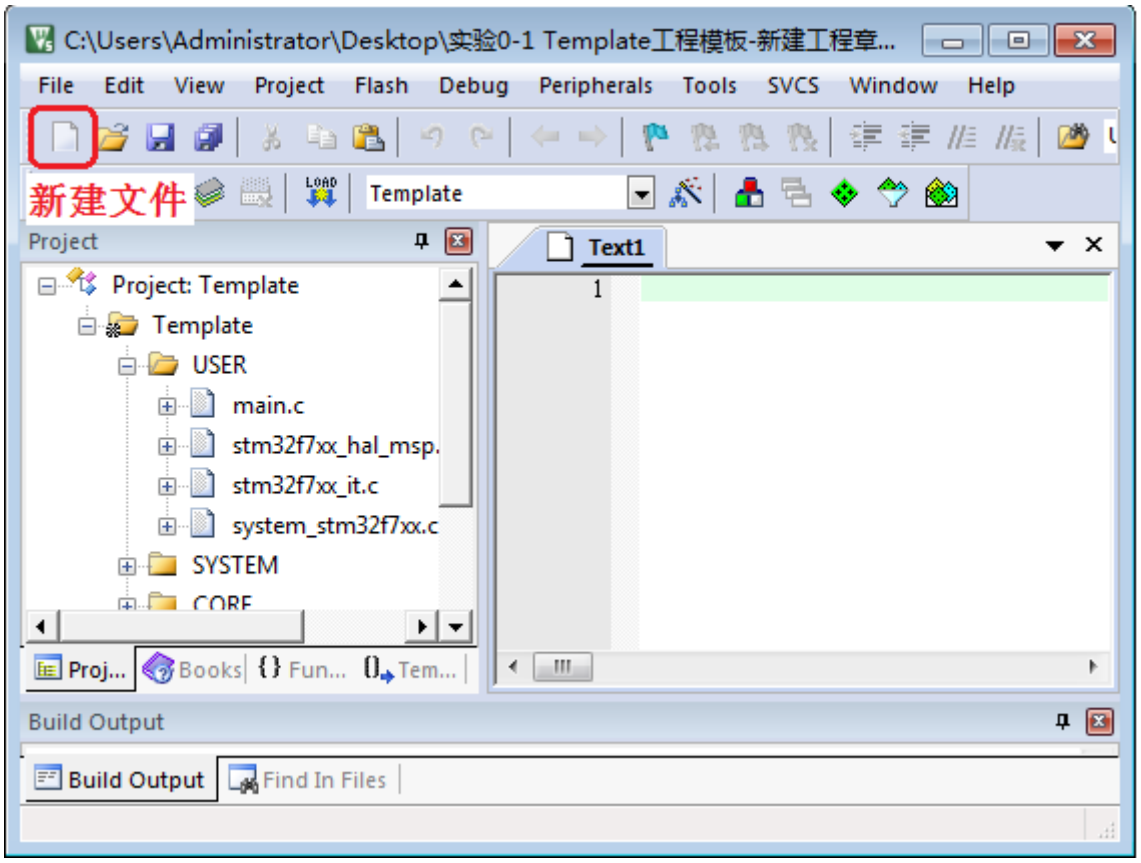


图 6.3.4 新建文件

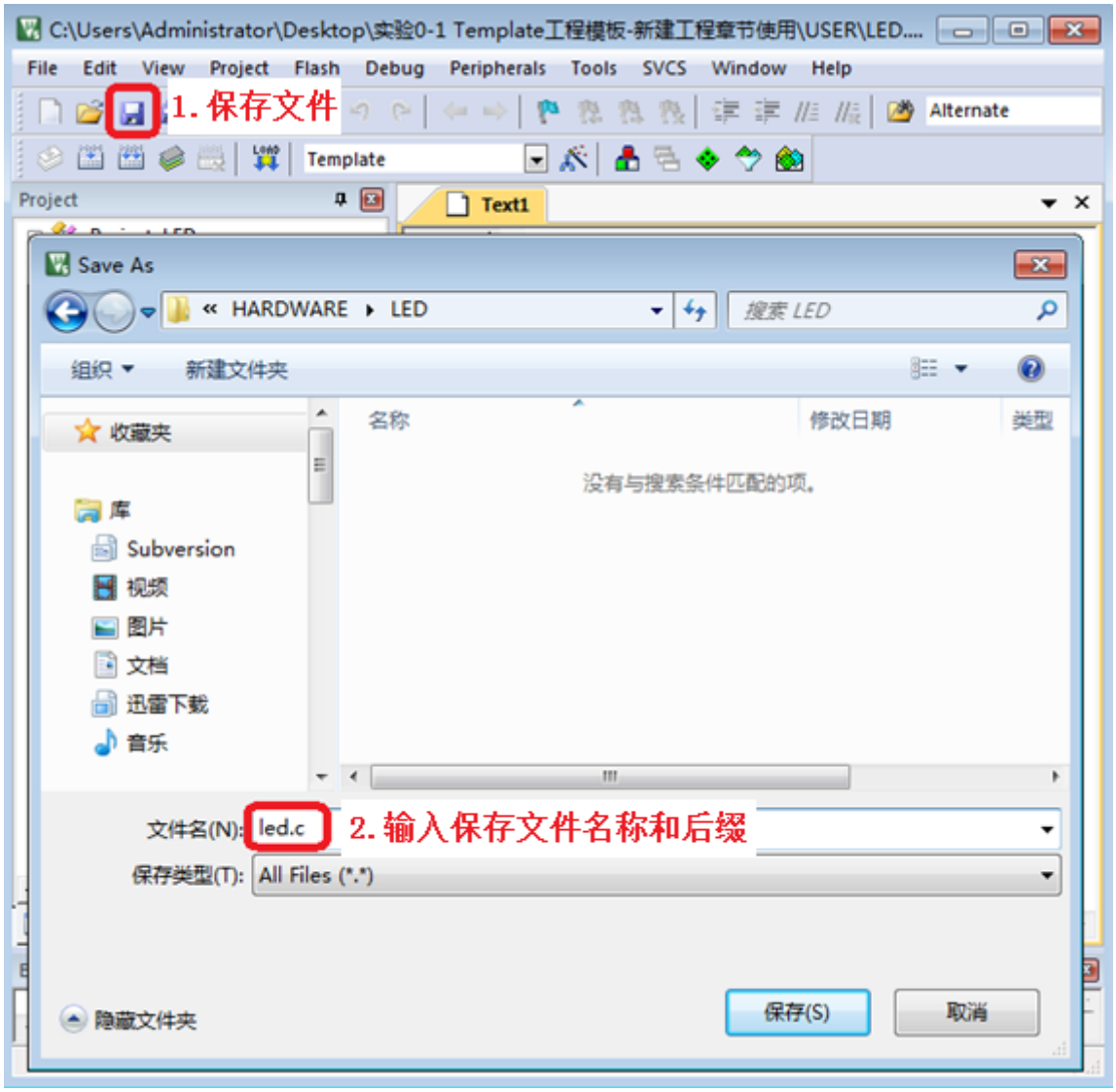


图 6.3.5 保存 led.c

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

    GPIO_Initure.Pin=GPIO_PIN_0|GPIO_PIN_1; //PB1,0
    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(GPIOB,&GPIO_Initure);

    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_SET); //PB0 置 1 ，默认灯灭
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_SET); //PB1 置 1 ，默认灯灭
}
```

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 PB0 和 PB1 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"

//LED 端口定义
#define LED0(n)         (n?HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_SET):\
                         HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_RESET))
#define LED0_Toggle (HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_1))
#define LED1(n)         (n?HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_SET):\
                         HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET))
#define LED1_Toggle (HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0))
void LED_Init(void);
#endif
```

这段代码中，对于宏定义标识符 LED0(n)，它的值是通过条件运算符来确定：当 n=0 时，标识符的值为 HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_RESET)，也就是设置 PB1 输出低电平，当 n!=0 时，标识符的值为 HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_SET，也就是设置 PB1 输出低电平。所以如果要设置 LED0 输出低电平，那么调用标识符 LED0(0)即可，当要设置 LED1 输出高电平，调用标识符 LED0(1)即可。标识符 LED1(n)和 LED0(n)作用类似。对于标识符 LED0_Toggle 和 LED1_Toggle，它们作用非常简单，就是调用 HAL 库函数 void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)来实现 IO 口输出电平取反操作。这里我们定义好上面的宏定义之后，就可以直接通过直接操作宏定义来实现 LED0 和 LED1 的状态控制，方法如下：

```
LED0(0);                    //PB1 输出低电平，LED0 亮
LED1(1);                    //PB1 输出高电平，LED0 灭
```

这里大家要注意，STM32F7 不支持位带操作，所以这里我们并没有像 F1/F4 一样通过位带操作来实现 IO 口输出输入电平控制。

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：

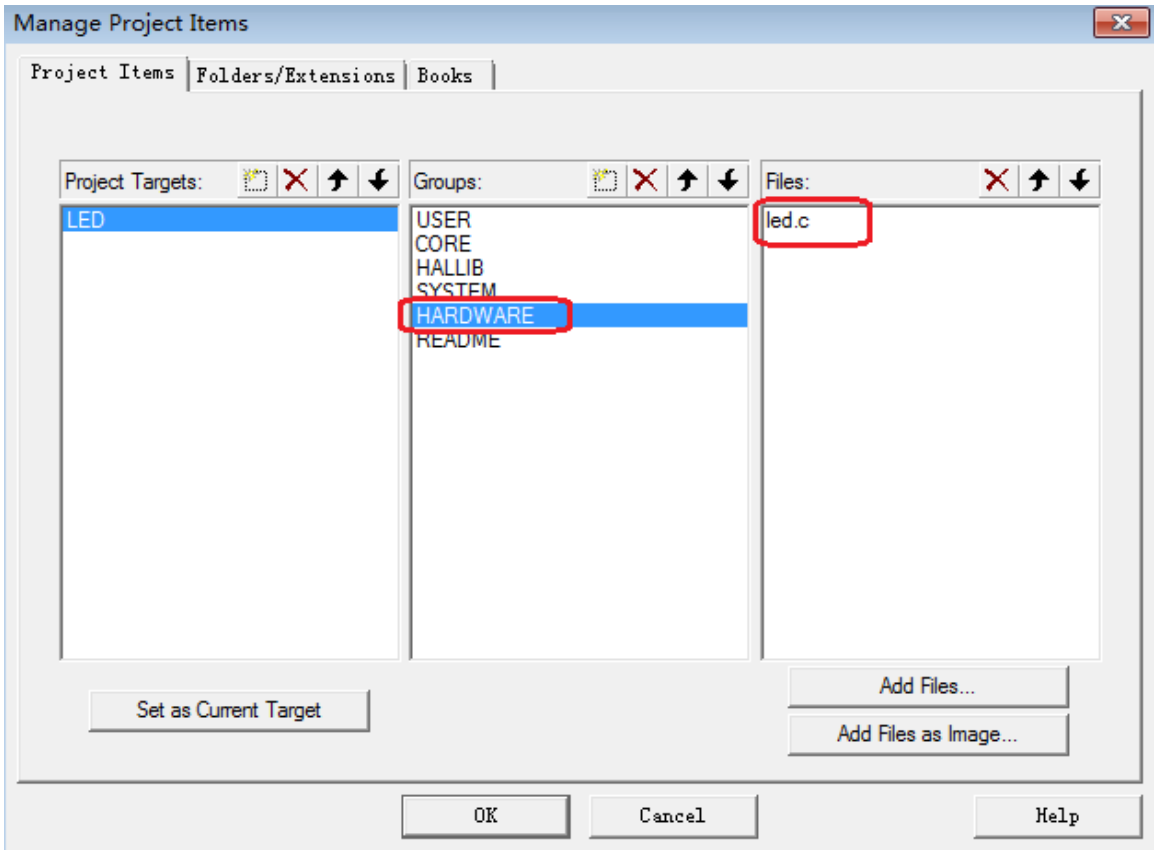


图 6.3.6 给工程新增 HARDWARE 组

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：

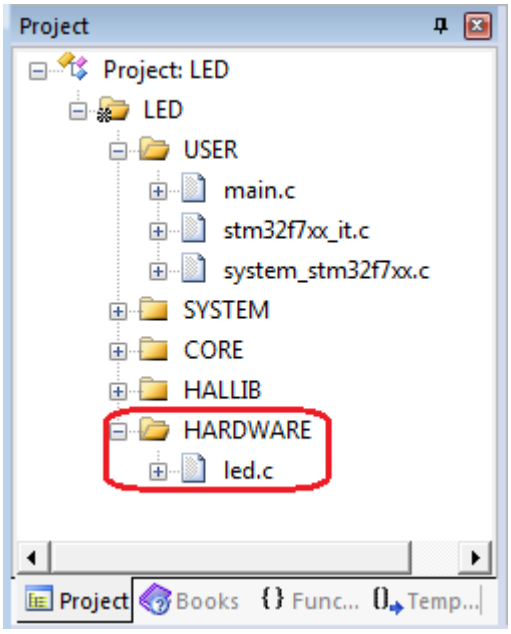


图 6.3.7 工程主界面

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：

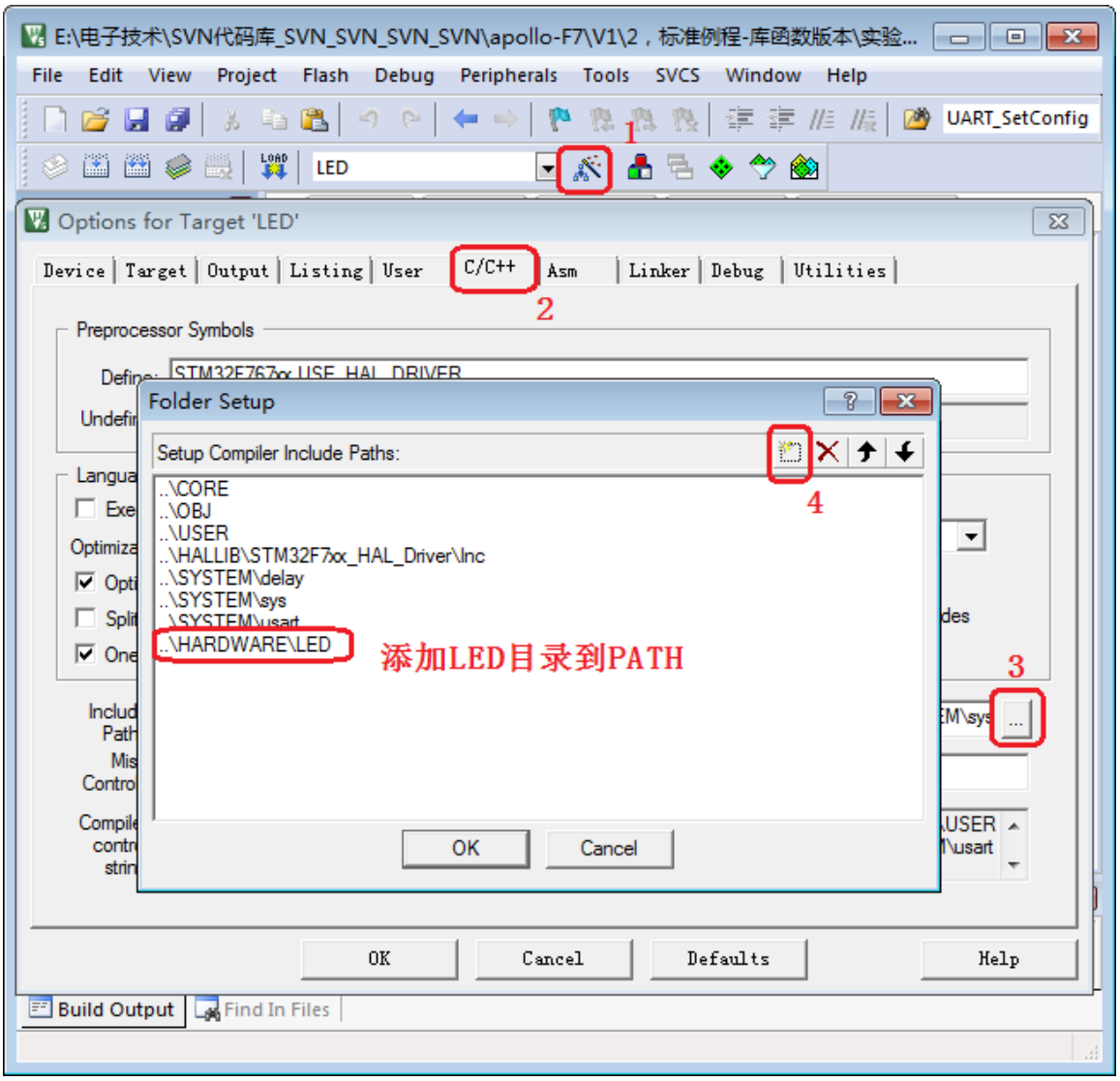


图 6.3.8 添加 LED 目录到 PATH

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)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
```

```
    LED_Init();                         //初始化 LED
    while(1)
    {
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_RESET);
                                        //LED0 对应引脚 PB1 拉低，亮，等同于 LED0(0)
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_SET);
                                        //LED1 对应引脚 PB0 拉高，灭，等同于 LED1(1)
        delay_ms(500);                  //延时 500ms
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_1,GPIO_PIN_SET);
                                        //LED0 对应引脚 PB1 拉高，灭，等同于 LED0(1)
        HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);
                                        //LED1 对应引脚 PB0 拉低，亮，等同于 LED1(0)
        delay_ms(500);                  //延时 500ms
    }
}
```

代码包含了#include "led.h"这句，使得 LED0(n)、LED1(n)、LED_Init 等能在 main()函数里被调用。main()函数非常简单，先调用 Cache_Enable()函数使能 I-Cache 和 D-Cache，然后调用 HAL_Init 函数初始化 HAL 库，调用 Stm32_Clock_Init 进行时钟系统配置，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 PB0 和 PB1 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，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)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();
    while(1)
    {
      GPIOB->BSRR=GPIO_PIN_1;           //LED0 亮
      GPIOB->BSRR=GPIO_PIN_0<<16;       //LED1 灭
      delay_ms(500);                    //延时 500ms
      GPIOB->BSRR=GPIO_PIN_1<<16;       //LED0 灭
      GPIOB->BSRR=GPIO_PIN_0;           //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，结果跟库函数操作效果一样。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：

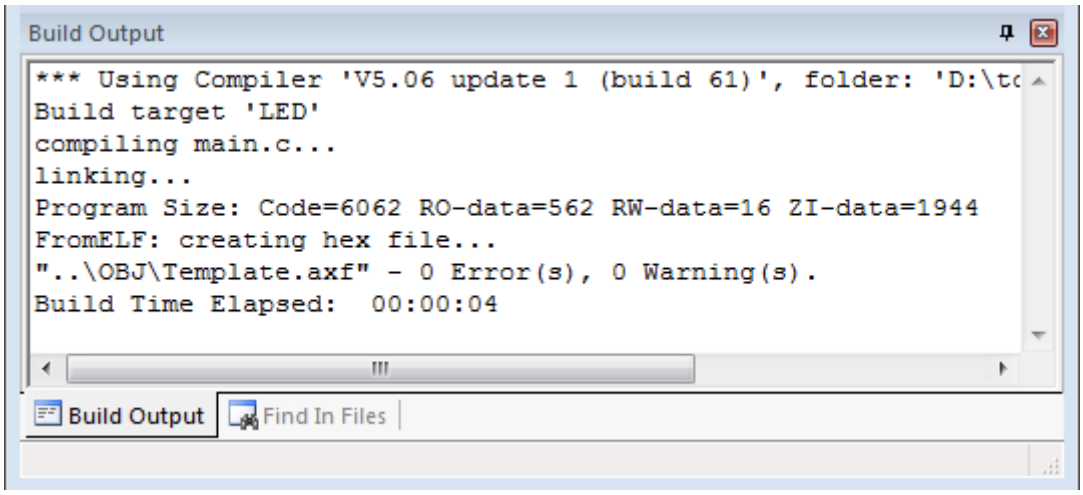


图 6.3.9 编译结果

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：6624 字节（6062+562），所用的 SRAM 大小为：1960 个字节（1944+16）。

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）

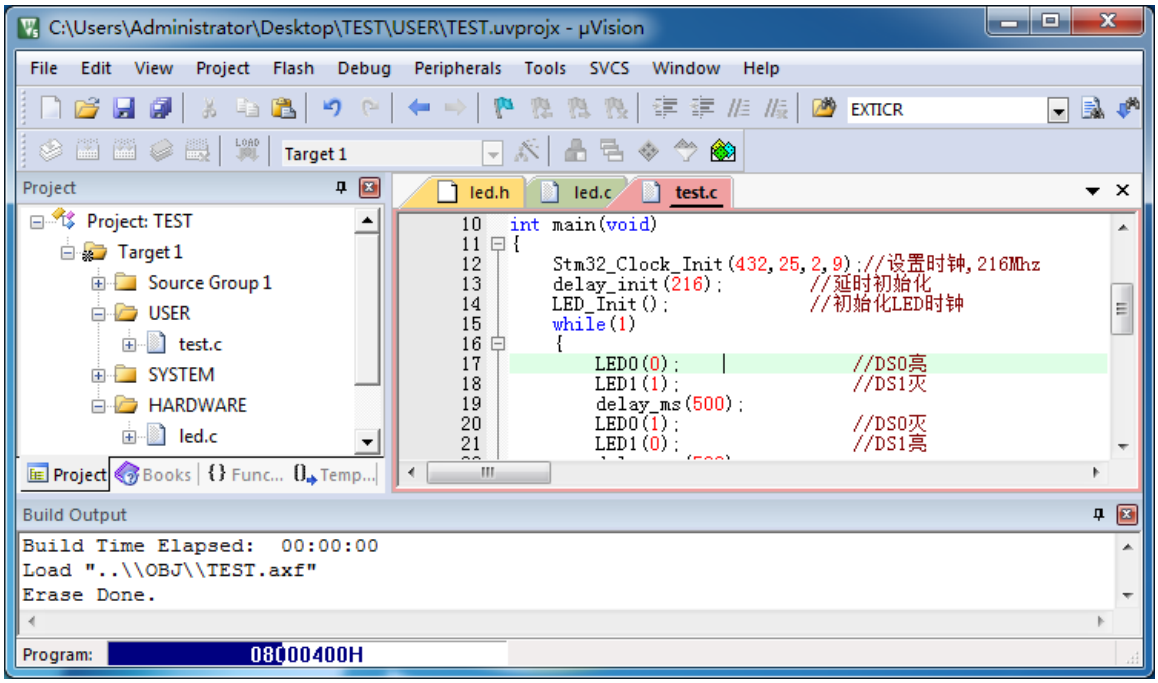
ZI-data：即 Zero Init-data，表示未被初始化的变量(SRAM)

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，STM32F7 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 ST LINK 下载（也可以通过其他仿真器下载，**如果是 JLINK，必须是 V9 或者以上版本，才可以支持 STM32F767!!** 下同），关于 ST LINK 的详细设置，请参考：4.1 节，设置完成后，在 MDK 里面点击  图标，就可以开始下载，如图 6.4.1 所示：



6.4.1 利用 ST LINK 下载代码

下载完之后，运行结果如图 6.4.2 所示：

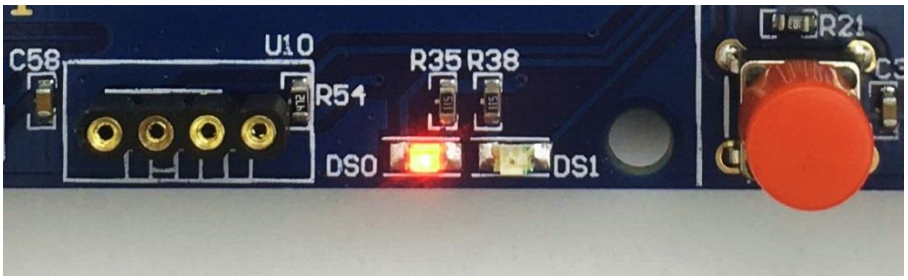


图 6.4.2 程序运行结果

至此，我们的第一章的学习就结束了，本章作为 STM32F767 的入门第一个例子，介绍了 STM32F767 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。

6.5 STM32CubeMX 配置 IO 口输入

在讲解完使用 HAL 库操作 GPIO 口之后，本小节我们教大家怎么使用 STM32CubeMX 图形化配置工具配置 GPIO 初始化过程。关于 STM32CubeMX 工具的入门使用在前面 4.8 小节我们有手把手教大家入门该工具，本小节我们就不重复讲解入门部分，我们直接讲解在 STM32CubeMX 工具中怎么来配置 GPIO 口的相关参数。

首先大家打开 STM32CubeMX 工具，参考 4.8 小节内容进行 RCC 相关配置。这里大家也可以直接打开 4.8 小节的 STM32CubeMX 工程直接在工程上面修改，该工程保存的光盘目录为：**“4，程序源码\标准例程-库函数版本\实验 0-3 Template 工程模板-使用 STM32CubeMX 配置”。**

这里大家会发现，我们在 4.8 小节实际上已经讲解了 GPIO 的配置，并且同样是以 PB0 和 PB1 为例。这里我们将详细解析在 STM32CubeMX 中配置 IO 口详细参数的过程。使用 STM32CubeMX 配置 GPIO 口的步骤如下：

第一步，打开 STM32CubeMX 工具，在引脚图中选择要配置的 IO 口。这里我们选择 PB0 为例，在弹出的下拉菜单中选择要配置的 IO 口模式，如下图 6.5.1 所示：

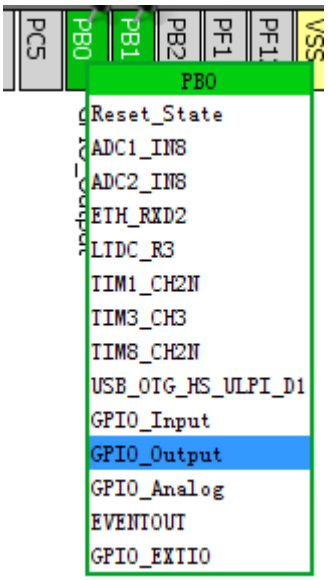


图 6.5.1 选择 IO 口模式

从上图可以看出，这里我们除了配置 IO 口为输入输出之外，还可以选择 IO 口的复用功能或者作为外部中断引脚功能，比如我们要选择 IO 口复用为 ADC1 的通道 8 引脚，那么我们只需要选选项 ADC1_IN8 即可。对于本章跑马灯实验，PB0 是作为输出，所以我们选 GPIO_Output 即可。

第二部，进入 Configuration->GPIO，在弹出的界面配置 IO 口的详细参数。如下图 6.5.2 和图 6.5.3 所示：

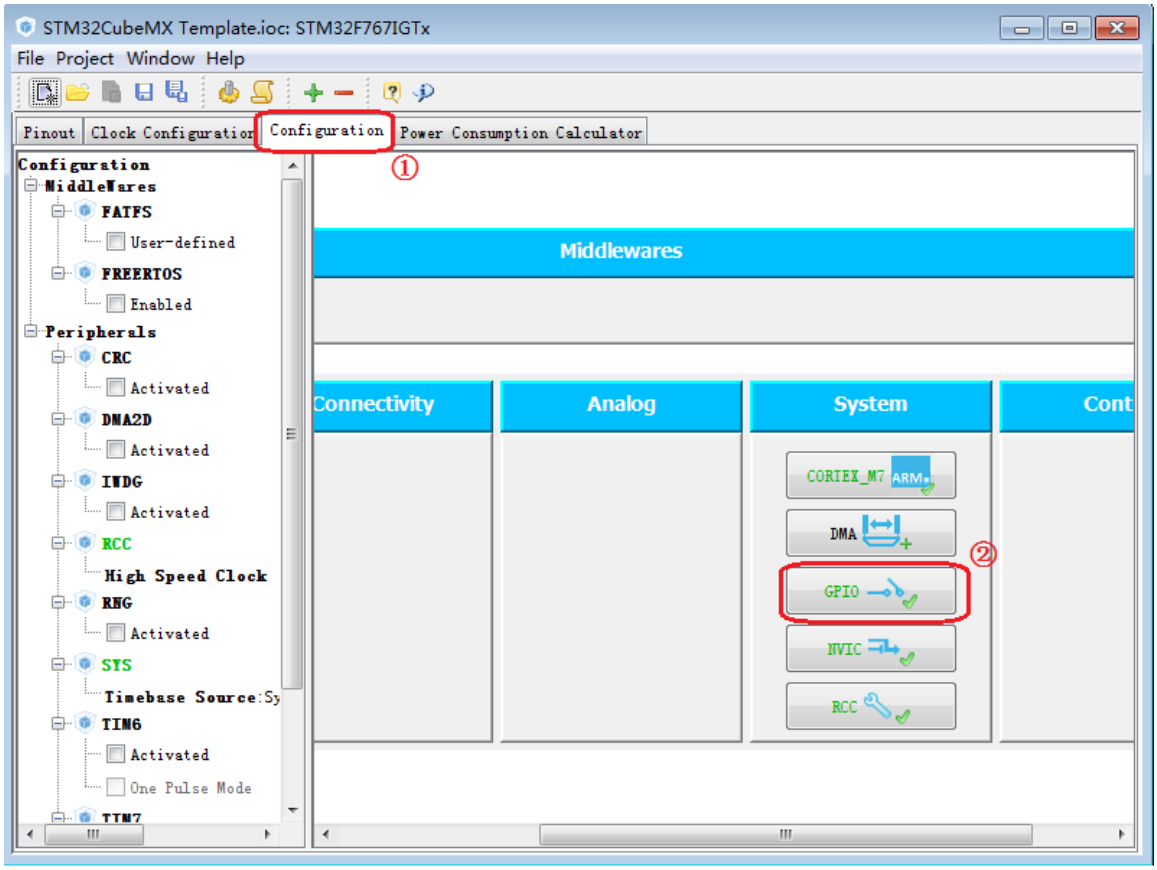


图 6.5.2 进入 GPIO 详细参数配置界面

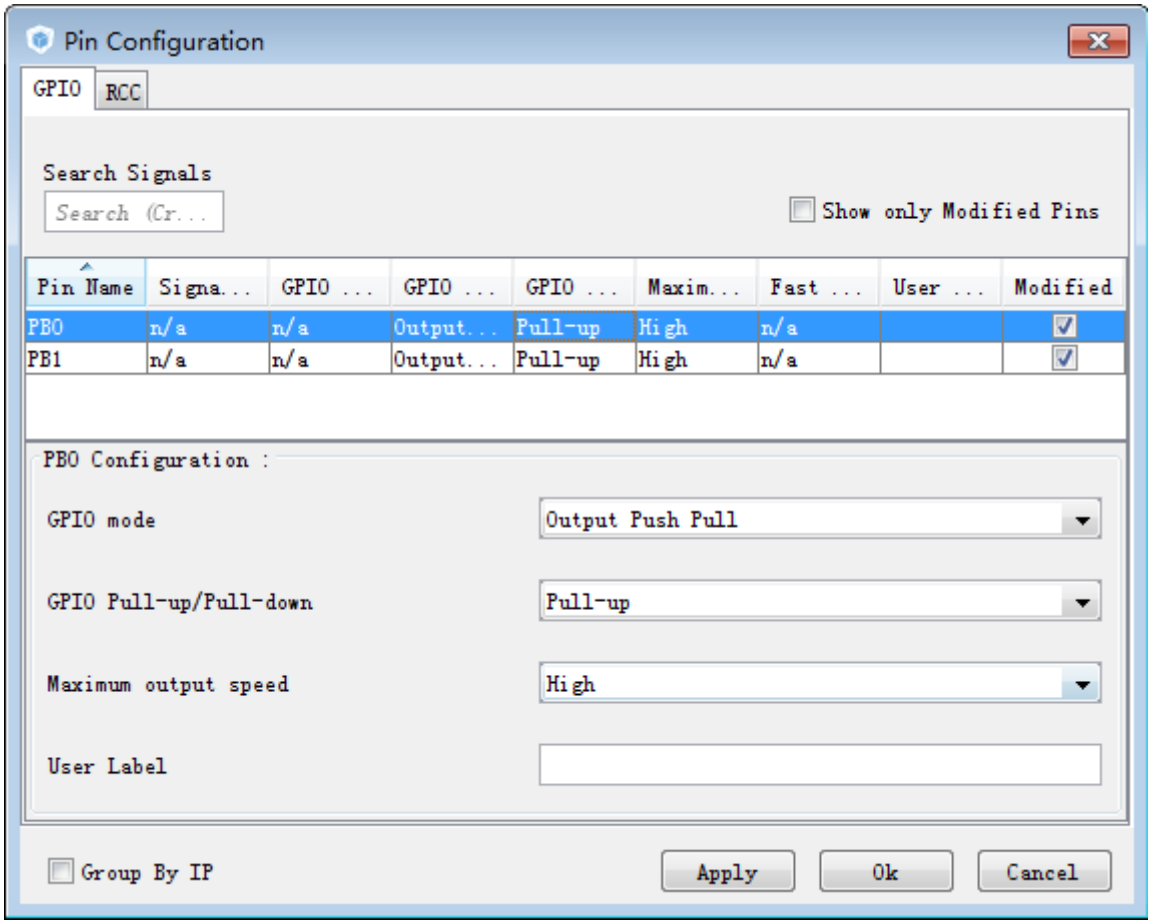


图 6.5.3 配置 IO 口详细参数

在 IO 口详细参数配置界面，点击我们要配置的 IO 口，会在窗口下方显示该 IO 口配置的详细参数表，下面我们依次来解释这些配置项的含义：

① 选项 GPIO mode 用来设置输出模式为 Output Push Pull(推挽)还是 Output Open Drain（开漏）。本实验我们设置为推挽输出 Output Push Pull。

② 选项 GPIO Pull-up/Pull-down 用来设置 IO 口是上拉/下拉/没有上下拉。本实验我们设置为上拉（Pull-up）。

③ 选项 Mzximum ouput speed 用来设置输出速度为高速(Hign)/快速(Fast)/中速(Medium)/低速(Low)。本实验我们设置为高速 High。

④ 选项 User Label 是用来设置初始化的 IO 口 Pin 值为我们自定义的宏，一般情况我们可以不用设置，有兴趣的同学可以自由设置后查看生成后的代码就很容易明白其含义。

配置完 PB0 后，PB1 配置方法和参数都一模一样，这里我们就不重复配置。

然后我们参考 4.8 下节方法，生成工程源码。接下来打开工程的 main.c 文件可以看到，该文件内部由 STM32CubeMX 生成了函数 MX_GPIO_Init，内容如下：

```
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
```

```
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_RESET);

    /*Configure GPIO pins : PB0 PB1 */
    GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

该函数的作用跟前面讲解的跑马灯实验中 LED_Init 函数作用一模一样，有兴趣的同学可以直接修改跑马灯实验工程源码，把 LED_Init 函数内容修改为 MX_GPIO_Init 内容，大家会发现实验效果一模一样。对于 IO 初始化后的默认状态，这里我们还需要根据自己需要来修改，也就是修改代码行：

```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_RESET);
```

这里我们希望默认情况下，PB1 和 PB0 输出为高电平，也就是灯灭，所以这里我们需要修改为：

```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_SET);
```

一般情况下，STM32CubeMX 的主要作用是配置时钟系统和外设初始化函数。所以我们对外设进行配置之后，生成外设初始化代码，然后把该代码应用到我们工程即可。关于本章中使用 STM32CubeMX 配置 GPIO 方法，我们就给大家讲解到这里。

第七章 按键输入实验

上一章，我们介绍了 STM32F7 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 STM32F7 的 IO 口作为输入用。在本章中，我们将利用板载的 4 个按键，来控制板载的两个 LED 的亮灭。通过本章的学习，你将了解到 STM32F7 的 IO 口作为输入口的使用方法。本章分为如下几个小节：

7.1 STM32F7 IO 口简介
7.2 硬件设计
7.3 软件设计
7.4 下载验证
7.5 STM32CubeMX 配置 IO 口输出

7.1 STM32F7 IO 口简介

STM32F7 的 IO 口在上一章已经有了比较详细的介绍，这里我们不再多说。STM32F7 的 IO 口做输入使用的时候，是通过调用函数 HAL_GPIO_ReadPin ()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 阿波罗 STM32 开发板上载有的 4 个按钮（KEY_UP、KEY0、KEY1 和 KEY2），来控制板上的 2 个 LED（DS0 和 DS1），其中 KEY_UP 控制 DS0，DS1 互斥点亮；KEY2 控制 DS0，按一次亮，再按一次灭；KEY1 控制 DS1，效果同 KEY2；KEY0 则同时控制 DS0 和 DS1，按一次，他们的状态就翻转一次。

7.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0、DS1。

2） 4 个按键：KEY0、KEY1、KEY2、和 KEY_UP。

DS0、DS1 和 STM32F767 的连接在上一章已经介绍过了，在阿波罗 STM32 开发板上的按键 KEY0 连接在 PH3 上、KEY1 连接在 PH2 上、KEY2 连接在 PC13 上、KEY_UP 连接在 PA0 上。如图 7.2.1 所示：

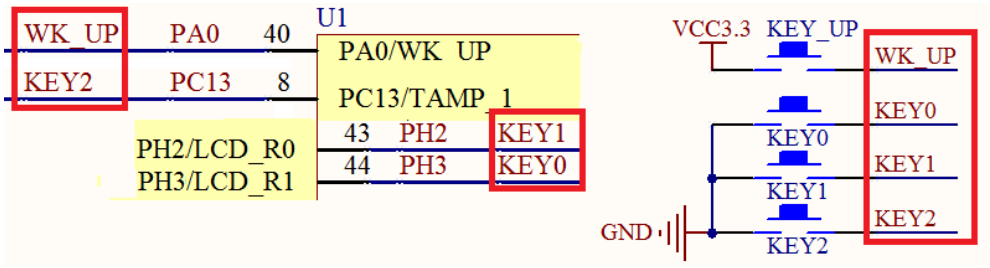


图 7.2.1 按键与 STM32F767 连接原理图

这里需要注意的是：KEY0、KEY1 和 KEY2 是低电平有效的，而 KEY_UP 是高电平有效的，并且外部都没有上下拉电阻，所以，需要在 STM32F767 内部设置上下拉。

7.3 软件设计

从这章开始，我们的软件设计主要是通过直接打开我们光盘的实验工程，而不再讲解怎么加入文件和头文件目录。工程中添加相关文件的方法在我们前面实验已经讲解非常详细。

打开按键实验工程可以看到，工程引入了 key.c 文件以及头文件 key.h。下面我们首先打开 key.c 文件，关键代码如下：

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

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

    __HAL_RCC_GPIOA_CLK_ENABLE();          //开启 GPIOA 时钟
    __HAL_RCC_GPIOC_CLK_ENABLE();          //开启 GPIOC 时钟
    __HAL_RCC_GPIOH_CLK_ENABLE();          //开启 GPIOH 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_HIGH;    //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    GPIO_Initure.Pin=GPIO_PIN_13;          //PC13
    GPIO_Initure.Mode=GPIO_MODE_INPUT;     //输入
    GPIO_Initure.Pull=GPIO_PULLUP;         //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;    //高速
    HAL_GPIO_Init(GPIOC,&GPIO_Initure);

    GPIO_Initure.Pin=GPIO_PIN_2|GPIO_PIN_3; //PH2,3
    HAL_GPIO_Init(GPIOH,&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>KEY2>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||KEY2==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(KEY2==0)    return KEY2_PRES;
        else if(WK_UP==1)   return WKUP_PRES;
    }else if(KEY0==1&&KEY1==1&&KEY2==1&&WK_UP==0)key_up=1;
    return 0;    //无按键按下
}
```

这段代码包含 2 个函数，void KEY_Init(void)和 u8 KEY_Scan(u8 mode)，KEY_Init 是用来初始化按键输入的 IO 口的。实现 PA0、PC13、PH2 和 PH3 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 KEY_UP 按键。该函数有返回值，如果有按键按下，则返回非 0 值，如果没有或者按键不正确，则返回 0。

接下来我们看看头文件 key.h 里面的代码：

```
#ifndef _KEY_H
#define _KEY_H
#include "sys.h"

/*下面的方式是通过直接操作 HAL 库函数方式读取 IO*/
#define KEY0        HAL_GPIO_ReadPin(GPIOH,GPIO_PIN_3)    //KEY0 按键 PH3
#define KEY1        HAL_GPIO_ReadPin(GPIOH,GPIO_PIN_2)    //KEY1 按键 PH2
#define KEY2        HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)   //KEY2 按键 PC13
#define WK_UP       HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_0)    //WKUP 按键 PA0

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

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(GPIOH,GPIO_PIN_3)   //KEY0 按键 PH3
```

```
#define KEY1          HAL_GPIO_ReadPin(GPIOH,GPIO_PIN_2)    //KEY1 按键 PH2
#define KEY2          HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)   //KEY2 按键 PC13
#define WK_UP         HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_0)    //WKUP 按键 PA0
```

这里使用的是调用 HAL 库函数 HAL_GPIO_ReadPin 来实现读取某个 IO 口的输入电平。函数 HAL_GPIO_ReadPin 的使用方法在上一章跑马灯实验我们已经讲解，这里我们就不累赘了。

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

最后，我们看看 main.c 里面编写的主函数代码如下：

```
int main(void)
{
    u8 key;
    u8 led0sta=1,led1sta=1;              //LED0,LED1 的当前状态
    Cache_Enable();                      //打开 L1-Cache
    HAL_Init();                          //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);        //设置时钟,216Mhz
    delay_init(216);                     //延时初始化
    uart_init(115200);                   //串口初始化
    LED_Init();                          //初始化 LED
    KEY_Init();                          //按键初始化
    while(1)
    {
        key=KEY_Scan(0);                 //得到键值
        if(key)
        {
            switch(key)
            {
                case WKUP_PRES:      //控制 LED0,LED1 互斥点亮
                    led1sta=!led1sta;
                    led0sta=!led1sta;
                    break;
                case KEY2_PRES:      //控制 LED0 翻转
                    led0sta=!led0sta;
                    break;
                case KEY1_PRES:      //控制 LED1 翻转
                    led1sta=!led1sta;
                    break;
                case KEY0_PRES:      //同时控制 LED0,LED1 翻转
                    led0sta=!led0sta;
                    led1sta=!led1sta;
                    break;
            }
            LED0(led0sta);               //控制 LED0 状态
```

```
                LED1(led1sta);                  //控制 LED1 状态
            }else delay_ms(10);
        }
    }
```

主函数代码比较简单，先进行一系列的初始化操作，然后在死循环中调用按键扫描函数 KEY_Scan()扫描按键值，最后根据按键值控制 LED 的状态。

7.4 下载验证

同样，我们还是通过 ST LINK 来下载代码，在下载完之后，我们可以按 KEY0、KEY1、KEY2 和 KEY_UP 来看看 DS0 和 DS1 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 STM32F767 的入门第二个例子，介绍了 STM32F767 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。

7.5 STM32CubeMX 配置 IO 口输出

上一章我们讲解了使用 STM32CubeMX 工具配置 GPIO 的一般方法。本章我们主要教大家配置 IO 口为输入模式，操作方法和配置 IO 口为输出模式基本一致。这里我们就直接列出 IO 口配置截图，具体方法请参考 4.8 小节和上一章跑马灯实验。

根据 7.2 小节讲解，阿波罗开发板上有 4 个按键，分别连接四个 IO 口 PA0，PC13，PH2 和 PH3。其中 WK_UP 按键按下后对应的 PA0 输入为高电平，所以默认情况下，该 IO 口（PA0）要初始化为下拉输入，其他 IO 口初始化为上拉输入即可。

使用 STM32CubeMX 打开光盘工程模板（双击工程目录的 Template.ioc），目录为**“4，程序源码\标准例程-库函数版本\实验 0-3 Template 工程模板-使用 STM32CubeMX 配置”**。我们首先在 IO 口引脚图上，依次设置四个 IO 口为输入模式 GPIO_Input。这里我们以 PA0 为例，操作方法如下图 7.5.1 所示：

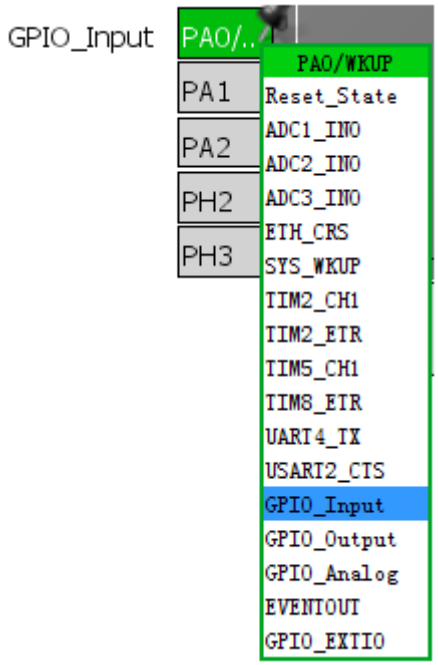


图 7.5.1 配置 PA0 为输入模式

同样的方法，我们依次配置 PC13，PH2 和 PH3 为输入模式。然后我们进入 Configuration->GPIO 配置界面，配置四个 IO 口详细参数。在配置界面点击 PA0 可以发现，当我们在前面设置 IO 口为输入 GPIO_Input 之后，其配置参数只剩下模式 GPIO Mode 和上下拉 GPIO Pull-up/Pull-down，并且模式值中只有输入模式 Input Mode 可选。这里，我们配置 PA0 为下拉输入，其他三个 IO 口配置为上拉输入即可。配置方法如下图 7.5.2 所示：

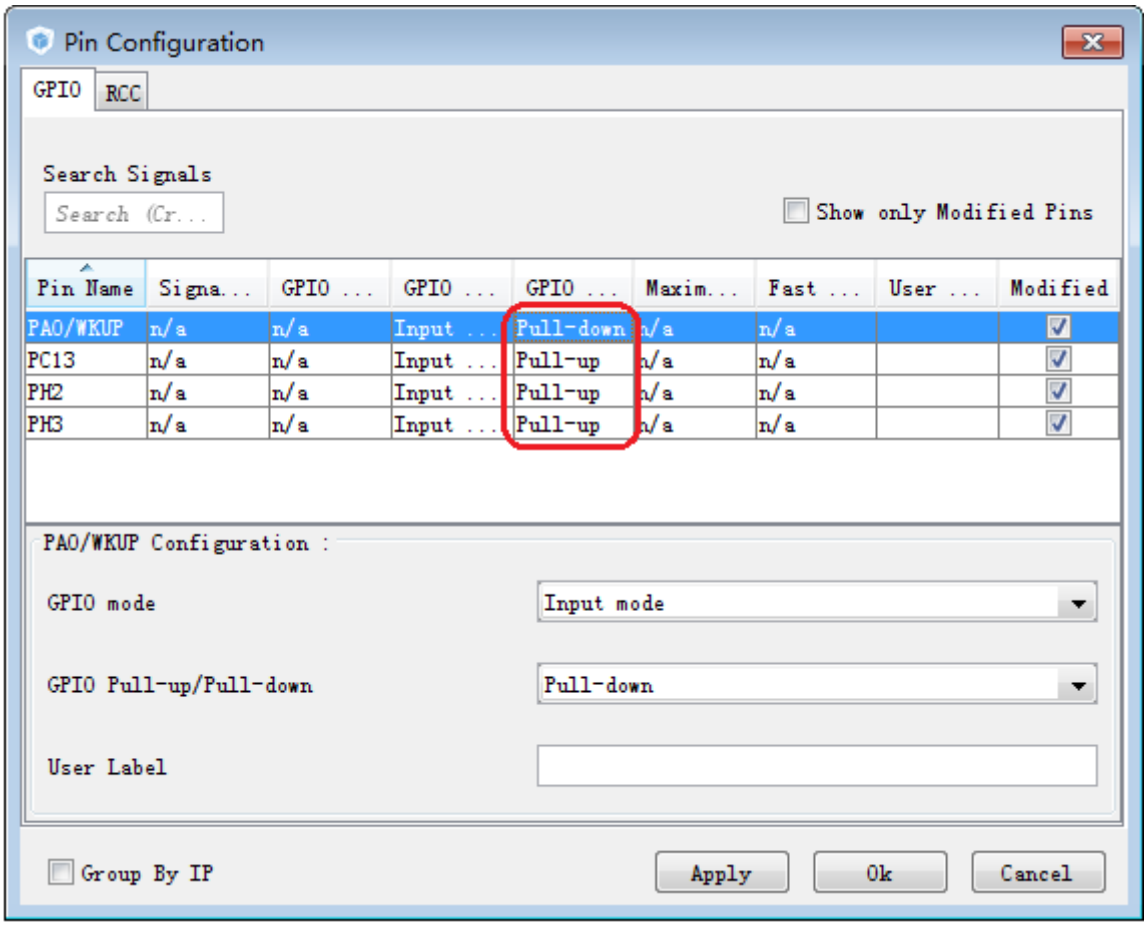


图 7.5.2 配置 IO 口详细参数

配置完成 IO 口参数之后，接下来我们同样生成工程。打开生成的工程会发现，main.c 文件中添加了函数 MX_GPIO_Init 函数，内容如下：

```
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin : PC13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
```

```
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    /*Configure GPIO pin : PA0 */
    GPIO_InitStruct.Pin = GPIO_PIN_0;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /*Configure GPIO pins : PH2 PH3 */
    GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);
}
```

该函数实现的功能和按键输入实验中 KEY_Init 函数实现的功能一模一样。有兴趣的同学可以直接复制该函数内容替换按键输入实验中的 KEY_Init 函数内容，替换后会发现实现现象完全一致。

使用 STM32CubeMX 配置 IO 口为输入模式方法就给大家介绍到这里。

第八章 串口通信实验

前面两章介绍了 STM32F767 的 IO 口操作。这一章我们将学习 STM32F767 的串口，教大家如何使用 STM32F767 的串口来发送和接收数据。本章将实现如下功能：STM32F767 通过串口和上位机的对话，STM32F767 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：

8.1 STM32F7 串口简介
8.2 硬件设计
8.3 软件设计
8.4 下载验证
8.5 STM32CubeMX 配置串口

8.1 STM32F7 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。

STM32F767 的串口资源相当丰富的，功能也相当强劲。ALIENTEK 阿波罗 STM32F767 开发板所使用的 STM32F767IGT6 最多可提供 8 路串口，支持 8/16 倍过采样、支持自动波特率检测、支持 Modbus 通信、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。阿波罗 STM32F767 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。STM32F767 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 STM32F767 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 4 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 和串口 6 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。

3，串口控制。STM32F767 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 8.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			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
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	Res.	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

图 8.1.1 USART_CR1 寄存器各位描述

该寄存器我们只介绍本节需要用到的一些位：M[1:0]位(位 28 和 12)，用于设置字长，我们一般设置为：00 表示 1 个起始位，8 个数据位，n 个停止位（n 的个数，由 USART_CR2 的[13:12]位控制）。OVER8 为过采样模式设置位，我们一般设置位 0，即 16 倍过采样已获得更好的容错性；UE 为串口使能位，通过该位置 1，以使能串口；PCE 为校验使能位，设置为 0，则禁止校验，否则使能校验；PS 为校验位选择位，设置为 0 则为偶校验，否则为奇校验；TXEIE 为发送缓冲区空中断使能位，设置该位为 1，当 USART_ISR 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_ISR 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_ISR 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《STM32F7 中文参考手册》第 945 页有详细介绍，在这里我们就不列出来了。

4，数据发送与接收。与 STM32F1 和 F4 不同，STM32F7 的串口发送和接收由两个不同的寄存器组成。发送数据是 USART_TDR 寄存器，接收数据是 USART_RDR 寄存器，USART_TDR 寄存器各位描述如图 8.1.2 所示：

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

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

图 8.1.2 USART_TDR 寄存器各位描述

可以看出，USART_TDR 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，TDR[8:0]为串口数据，具体多少位，由前面介绍的 M[1:0]决定（一般是 8 位数据）。

当我们需要发送数据的时候，往 USART_TDR 寄存器写入你想要发送的数据，就可以通过串口发送出去了。而当有串口数据接收到，需要读取出来的时候，我们则必须读取 USART_RDR 寄存器，USART_RDR 寄存器各位描述同 USART_TDR 是完全一样的，只是一个用来接收，一个用来发送。

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。

5，串口状态。串口的状态可以通过状态寄存器 USART_ISR 读取。USART_ISR 的各位描述如图 8.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TEACK	Res.	Res.	SBKF	CMF	BUSY
										r			r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ABRF	ABRE	Res.	EOBF	RTOF	CTS	CTSIF	LBDF	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
r	r		r	r	r	r	r	r	r	r	r	r	r	r	r

图 8.1.3 USART_ISR 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_RDR，通过读 USART_RDR 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_TDR 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_ISR，写 USART_TDR。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《STM32F7 中文参考手册》第 907 页至 964 页，通用同步异步收发器这一章节。

对于怎么直接使用寄存器配置串口收发，请参考我们寄存器版本教程和源码。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 stm32f7xx_hal_uart.c 和 stm32f7xx_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;
    UART_AdvFeatureInitTypeDef    AdvancedInit;
    uint8_t                       *pTxBuffPtr;
    uint16_t                      TxXferSize;
    uint16_t                      TxXferCount;
    uint8_t                       *pRxBuffPtr;
    uint16_t                      RxXferSize;
    uint16_t                      RxXferCount;
    uint16_t                      Mask;
    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，一般情况下使用串口的基本功能，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。

AdvancedInit 是用来配置串口的高级功能，有兴趣的同学可以对照中文参考手册了解一下。

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_InitTypeDef GPIO_Initure;
GPIO_Initure.Pin=GPIO_PIN_9|GPIO_PIN_10;  //PA9/PA10
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_FAST;       //高速
GPIO_Initure.Alternate=GPIO_AF7_USART1;   //复用为 USART1
HAL_GPIO_Init(GPIOA,&GPIO_Initure);       //初始化 PA9/PA10
```

3） 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 stm32f7xx_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 stm32f7xx_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
```

4） 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。

5） 串口数据接收和发送

STM32F7 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。

STM32F7 相关基础知识我们就给大家讲解到这里，接下来我们看看本实验的软硬件设计。

8.2 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 P4 的 RXD 和 TXD 用跳线帽与 PA9 和 PA10 连接起来。如图 8.2.1 所示：

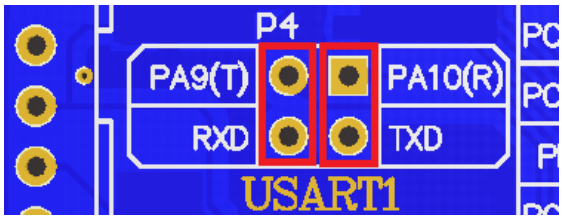


图 8.2.1 硬件连接图示意图

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。

8.3 软件设计

本小节，我们首先会讲解使用 HAL 库配置串口的一般步骤。然后我们会具体讲解我们串口实验程序实现。ALIENTEK 编写的串口相关的源码再 SYSTEM 分组之下的 usart.c 和 usart.h 中。

8.1 小节我们讲解了 HAL 库中串口操作的一般步骤以及操作函数。在使用 HAL 库配置串口的时候，HAL 库为我们封装了串口配置步骤。接下来我们以串口接收中断为例讲解 HAL 库串口程序执行流程。

和其他外设一样，HAL 库为串口的使用开放了 MSP 函数。在串口初始化函数 HAL_UART_Init 内部，会调用串口 MSP 函数 HAL_UART_MspInit 来设置与 MCU 相关的配置。根据前面的讲解，函数 HAL_UART_Init 主要用来初始化与串口相关的参数（这些参数与 MCU 无关），包括波特率，停止位等。而串口 MSP 函数 HAL_UART_MspInit 用来设置 GPIO 初始化，NVIC 配置等于 MCU 相关的配置。

这里我们定义了一个函数 uart_init 用来调用 HAL_UART_Init 初始化串口参数配置，具体函数如下：

```
UART_HandleTypeDef UART1_Handler; //UART 句柄
```

```
//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, 1);
        //该函数会开启接收中断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

        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_FAST;     //高速
        GPIO_Initure.Alternate=GPIO_AF7_USART1; //复用为 USART1
        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

#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 tmp1 = 0, tmp2 = 0;
  …//此处省略部分代码
  tmp1 = __HAL_UART_GET_FLAG(huart, UART_FLAG_RXNE);
  tmp2 = __HAL_UART_GET_IT_SOURCE(huart, UART_IT_RXNE);
  if((tmp1 != RESET) && (tmp2 != RESET))
  {
    UART_Receive_IT(huart);
  }
  …//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(--huart->RxXferCount == 0)
    {
      HAL_UART_RxCpltCallback(huart);
    }
    ...//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：

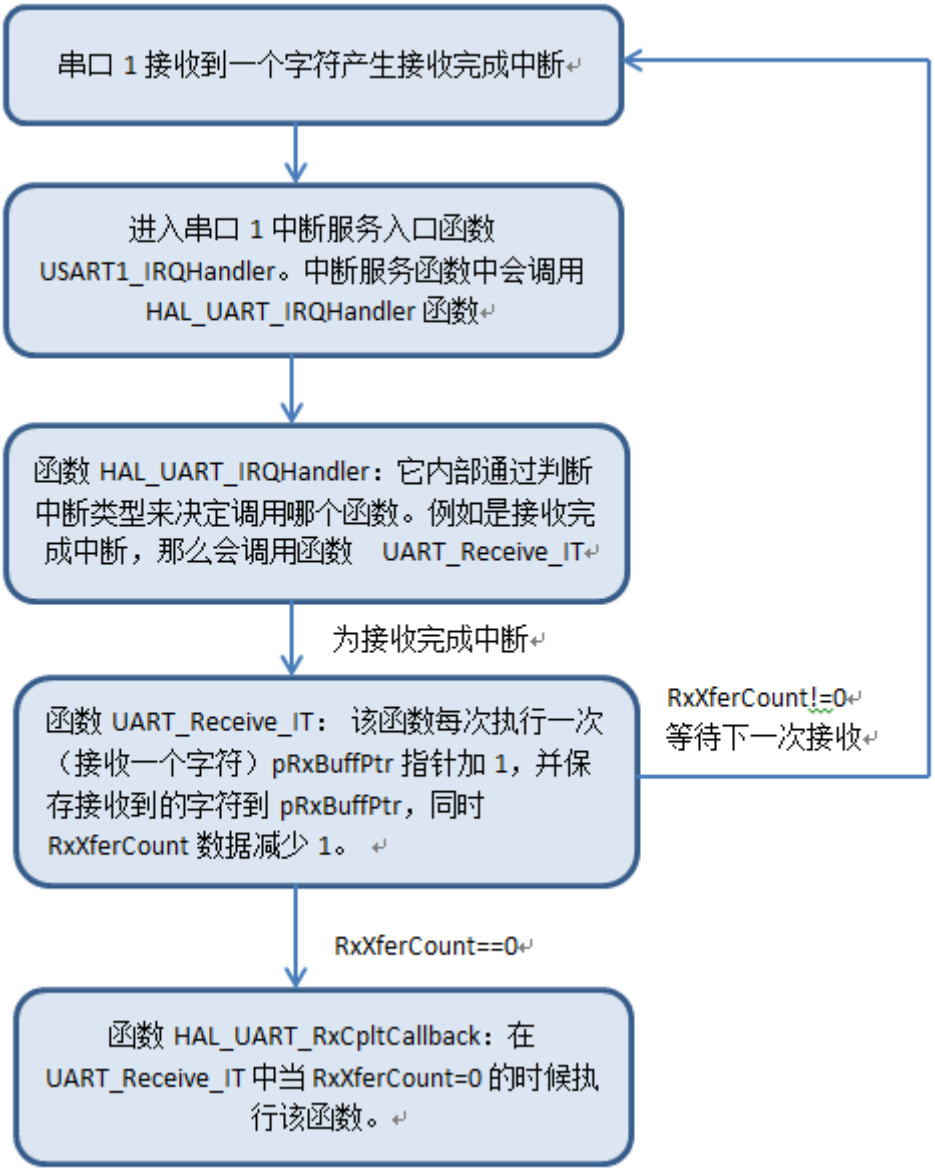


图 8.3.1 串口接收中断执行流程图

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，会直接保存到 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
```

关于这些回调函数的作用，我们在函数后面有注释，有兴趣的同学可以自行测试每个回调函数的使用方法，这里我们就不做过多讲解。大家只需要知道，每当一个事件发生，就会最终调用相应的回调函数，我们在回调函数中编写真正的控制逻辑即可。最后我们来看看主函数：

```
int main(void)
{
    u8 len;
    u16 times=0;
```

```
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    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 STM32F7 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_Toggle;//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。

8.4 下载验证

本实验需要用到的硬件资源有：

3） 指示灯 DS0

4） 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 P4 的 RXD 和 TXD 用跳线帽与 PA9 和 PA10 连接起来。如图 8.2.1 所示：

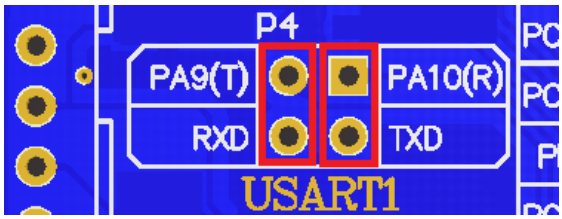


图 8.2.1 硬件连接图示意图

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。

8.5 STM32CubeMX 配置串口

前面章节我们详细讲解了使用 STM32CubeMX 配置 IO 口输入输出，本小节我们将讲解使用 STM32CubeMX 配置串口方法。同样，大家直接复制光盘的 STM32CubeMX 配置的工程模板，目录为：**“4，程序源码\标准例程-库函数版本\实验 0-3 Template 工程模板-使用 STM32CubeMX 配置”**。然后使用 STM32CubeMX 打开该工程（点击工程目录的 Template.ioc）。这里我们同样不再讲解 RCC 相关配置，我们仅仅讲解串口相关配置方法。

这里我们要配置串口 1，所以首先我们要使能串口 1 然后设置相应通信模式。打开 Pinout 选项卡界面，左侧依次进入 Configuration->Peripherals->USART1 配置栏，如下图 8.5.1 所示：

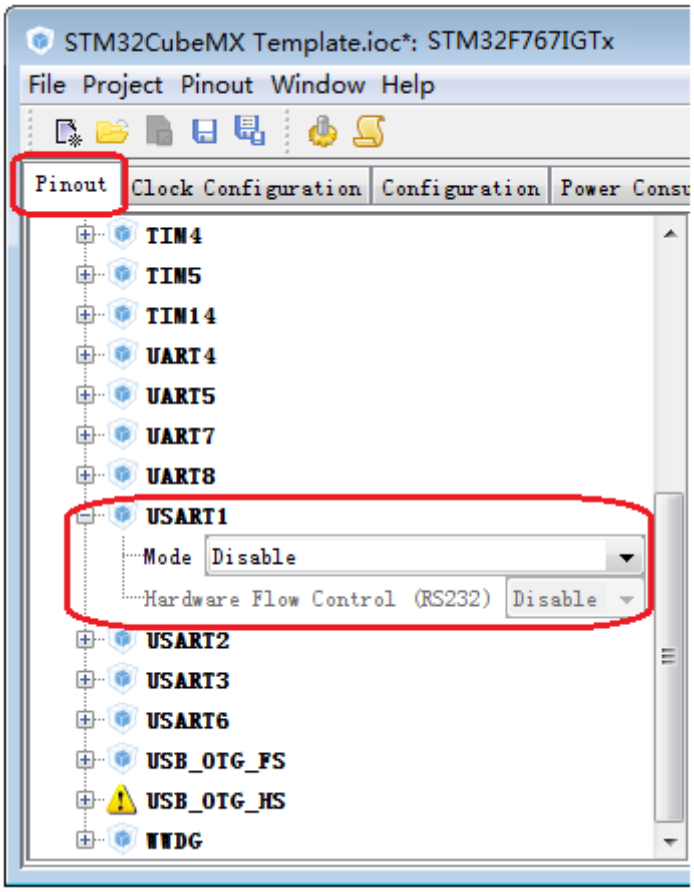


图 8.5.1 进入 Configuration->Peripherals->USART1 配置栏

USART1 配置栏有 2 个选项。第一个选项 Mode 用来设置串口 1 的模式或者关闭串口 1。第二个选项 Hardware Flow Control(RS232)用来开启/关闭串口 1 的硬件流控制，该选项只有在 Mode 选项值为 Asynchronous(异步通信)模式的前提下才有效。这里我们要开启串口 1 的异步模式，并且不使用硬件流控制，所以这里我们直接选择 Mode 值为 Asynchronous 即可。配置好的 USART1 界面如下图 8.5.2 所示：

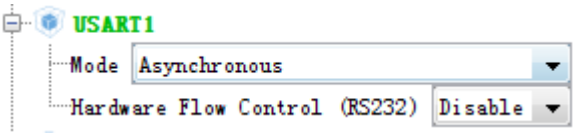


图 8.5.2 USART1 配置

配置好串口 1 为异步通信模式后，那么在硬件上会默认开启 PB14 和 PB15 作为串口 1 引脚。这时候我们进入引脚配置图可以发现，PB14 和 PB15 变为绿色，同时显示为 USART1_TX 和 USART1_RX 功能引脚，如下图 8.5.3 所示：

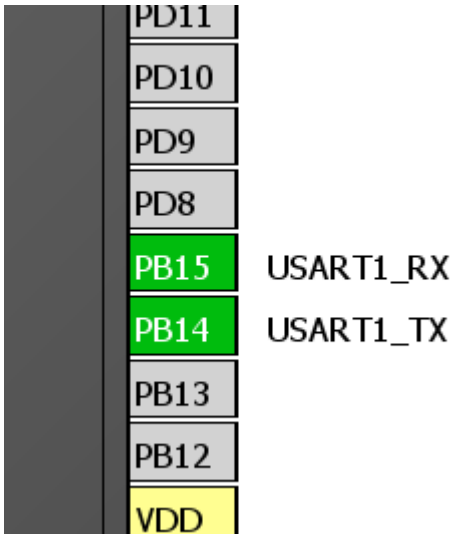


图 8.5.3 PB14/PB15 引脚模式

而这里我们需要使用 PA9 和 PA10 作为串口 1 的发送接收引脚，所以我们需要重新修改引脚模式（这里实际上涉及到 F7 的引脚复用映射方面的知识，如有不理解的地方请参考 4.4 小节）。这里我们分别选中 PA9 和 PA10，然后修改引脚模式为 USART1_TX 和 USART1_RX，那么 PB14 和 PB15 的模式就会自动设置为复位后的模式，修改后引脚图如下图 8.5.4 所示：

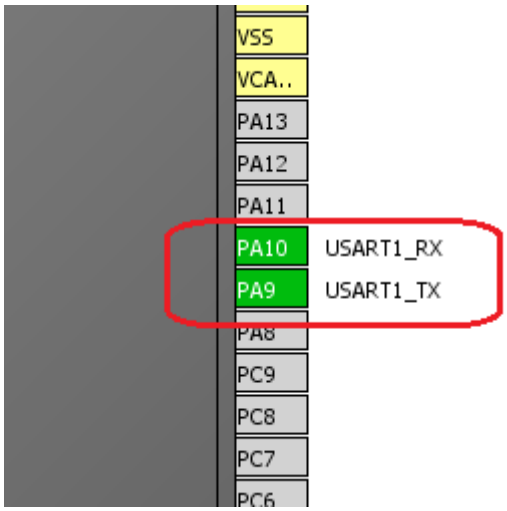


图 8.5.4PA9/PA10 引脚模式

同时，进入 GPIO 配置详细界面会发现，IO 口的模式等参数都做了相应的修改。参考 6.5 小节方法，依次进入 Configuration->GPIO 界面会发现，Pin Configuration 界面多了一个 USART1 选项卡，该选项卡界面便是用来配置和查看串口引脚 PA9 和 PA10 配置参数的。如下图 8.5.5 所示：

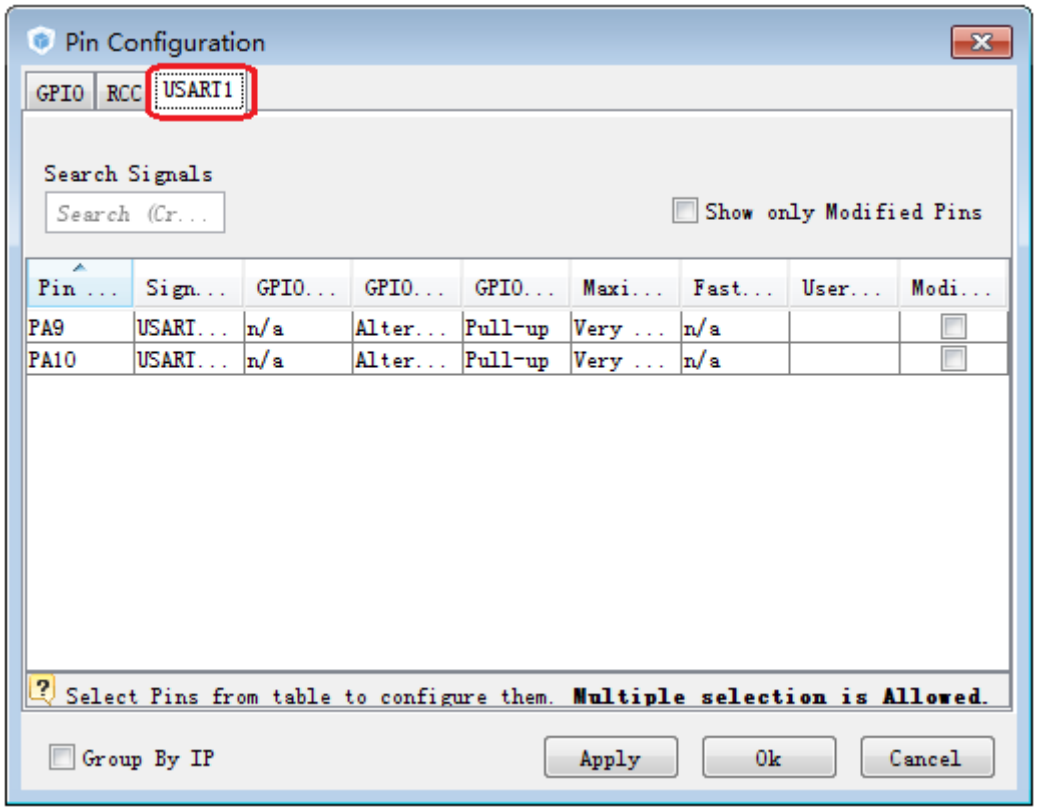


图 8.5.5 USART1 引脚详细配置界面

对于外设的功能引脚，在我们使能相应的外设（比如 USART1）之后，STM32CubeMX 会自动设置 GPIO 相关配置，一般情况下用户不再需要去修改。所以这里，对于 PA9 和 PA10 的配置我们就保留软件配置即可。

接下来我们需要配置 USART1 外设相关的参数，包括波特率，停止位等。我们直接进入 Configuration 选项卡，如果我们之前使能了 USART1,那么在 Connectivity 栏会出现 USART1 配置按钮。如下图 8.5.6 所示：

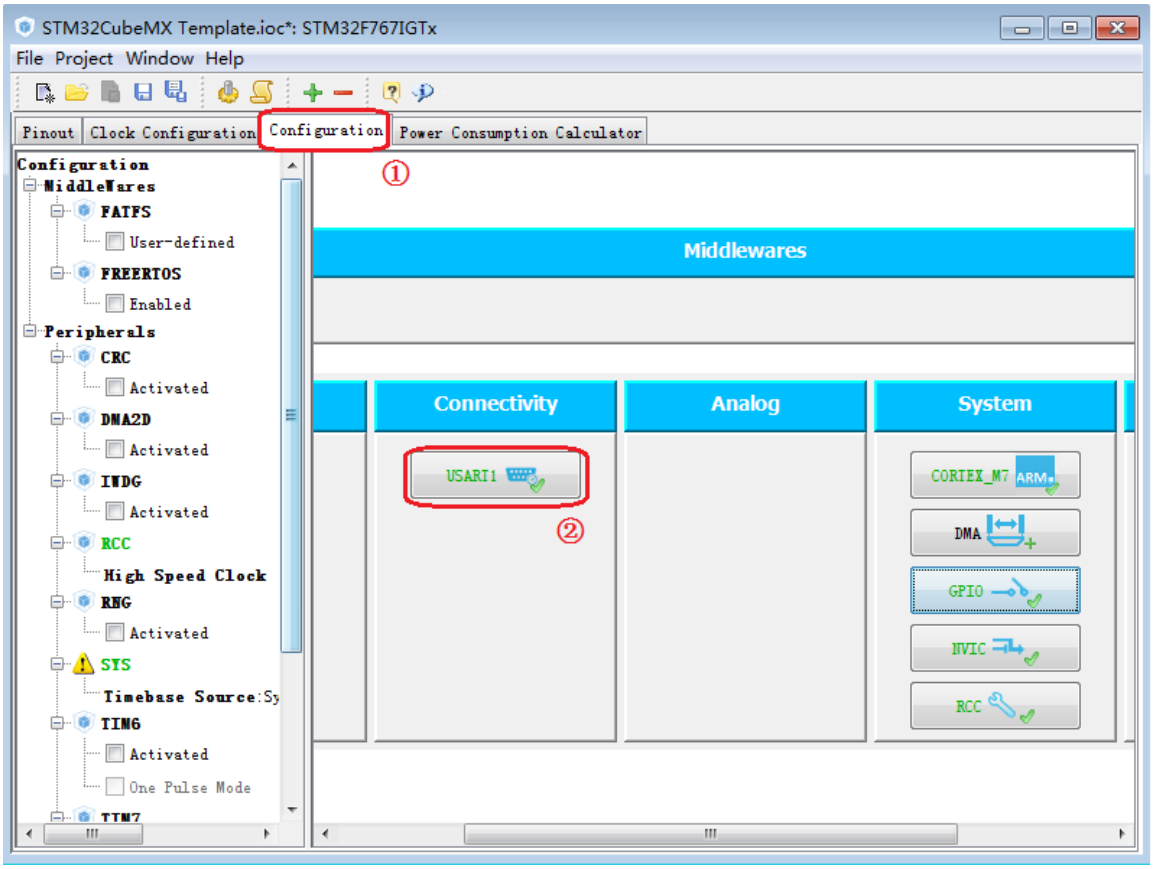


图 8.5.6Configuration 选项卡

接下来我们点击 USART1 配置按钮，进入 USART1 详细参数配置界面。在弹出的 USART1 Configuration 界面会出现 5 个配置选项卡。

Parameter Settings 选项卡用来配置 USART1 的初始化参数，包括波特率停止位等等。这里我们将 USART1 配置为：波特率 115200，8 位字长模式，无奇偶校验位，1 个停止位，发送/接收均开启。

User Constants 是用来配置用户常量。

NVIC 选项卡用来使能 USART1 中断。这里我们勾上 Enabled 选项。

DMA Setting 是在使用 USART1 DMA 的情况才需要配置，这里我们不配置。

GPIO Setting 便是查看和配置 USART1 相关的 IO 口，这和图 8.5.4 作用一致。

配置完 USART1 相关 IO 口和 USART1 参数之后，如果我们使用到串口中断，那么我们还需要设置中断优先级分组。接下来便是配置 NVIC 相关参数。同样的方法，进入 Conguration 选项卡，点击 NVIC 按钮，如下图 8.5.7 所示：

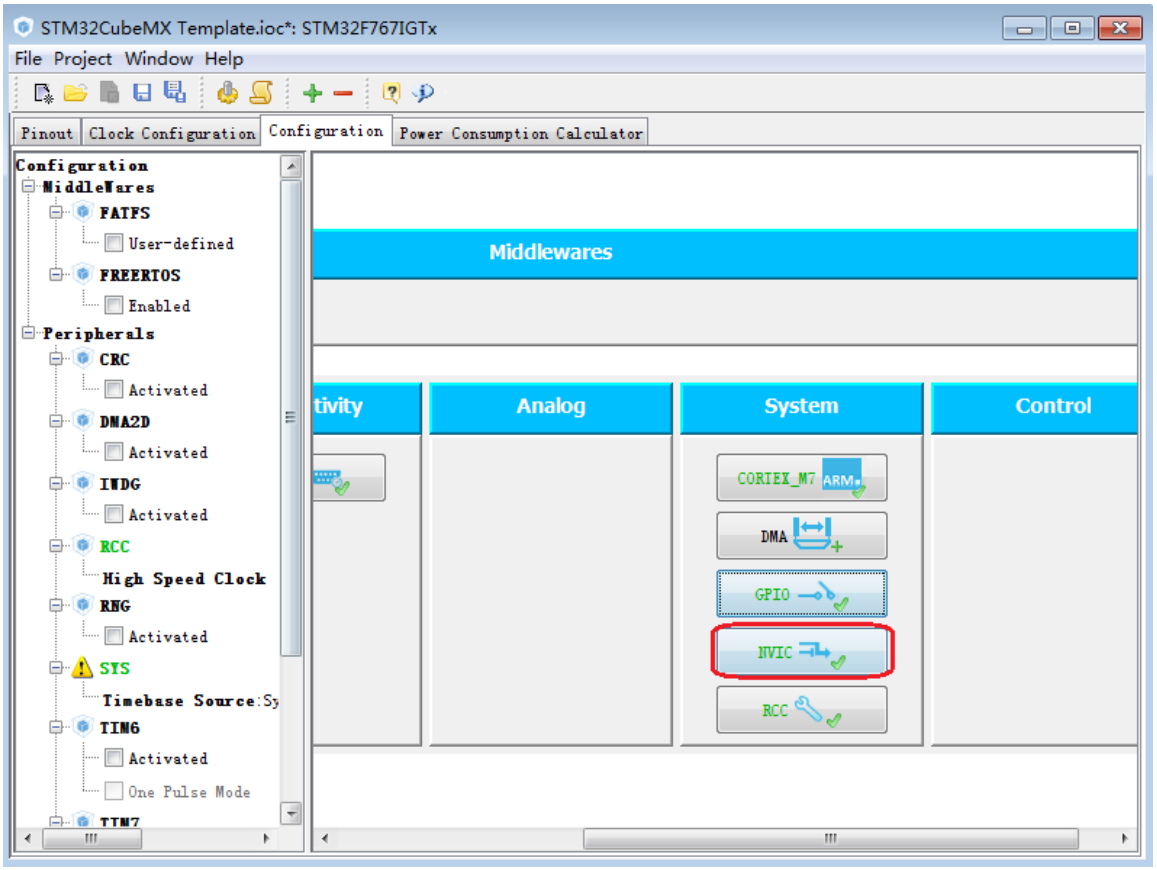


图 8.5.7Configuration->NVIC 按钮

点击 NVIC 按钮之后，弹出 NVIC 配置界面 NVIC Configuration，如下图 8.5.8 所示：

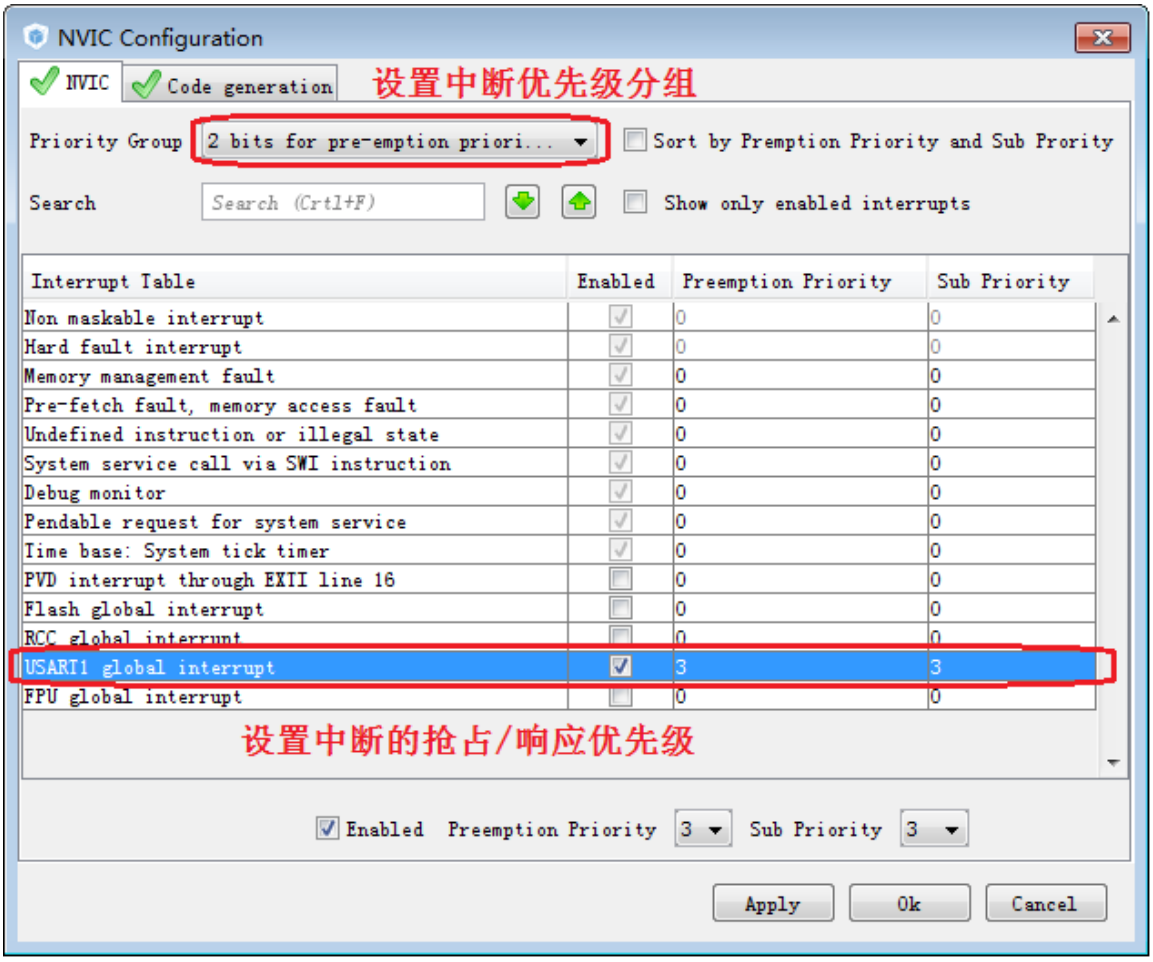


图 8.5.8 NVIC Configuration 配置界面

在弹出的 NVIC Configuration 界面，我们首先设置中断优先级分组级别，我们系统初始化设置为分组 2，那么就是 2 为抢占优先级和 2 位响应优先级。所以这里的参数我们选择“2 bits for pre-emption priority”，也就是 2 位抢占优先级。

配置完中断优先级分组之后，接下来我们要配置的是 USART1 的抢占优先级和响应优先级值，这里我们设置抢占和响应优先级均为 3 即可。

进行完上面的操作之后，接下来我们便是生成工程代码。

打开生成的工程可以看到，在 main.c 文件中生成了如下串口初始化关键代码：

```
static void MX_USART1_UART_Init(void)
{

  huart1.Instance = USART1;
  huart1.Init.BaudRate = 115200;
  huart1.Init.WordLength = UART_WORDLENGTH_7B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
```

```
    huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart1) != HAL_OK)
    {
      Error_Handler();
    }

  }
```

同时在 stm32f7xx_hal_msp.c 中，生成了串口 MSP 函数 HAL_UART_MspInit 内容如下：

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart)
{

  GPIO_InitTypeDef GPIO_InitStruct;
  if(huart->Instance==USART1)
  {
    __HAL_RCC_USART1_CLK_ENABLE();

    GPIO_InitStruct.Pin = GPIO_PIN_9|GPIO_PIN_10;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF7_USART1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    HAL_NVIC_SetPriority(USART1_IRQn, 3, 3);
    HAL_NVIC_EnableIRQ(USART1_IRQn);
  }
}
```

函数 MX_USART1_UART_Init 的内容和本章串口实验源码中函数 uart_init 中调用 HAL_UART_Init 函数作用类似，只不过波特率是通过入口参数动态设置。而生成的 MSP 函数 HAL_UART_MspInit 内容和实验中该函数的作用就几乎是一模一样了。

关于使用 STM32CubeMX 配置串口的方法就给大家介绍到这里。

第九章 外部中断实验

这一章，我们将向大家介绍如何使用 STM32F7 的外部输入中断。在前面几章的学习中，我们掌握了 STM32F7 的 IO 口最基本的操作。本章我们将介绍如何将 STM32F7 的 IO 口作为外部中断输入，在本章中，我们将以中断的方式，实现我们在第七章所实现的功能。本章分为如下几个部分：

9.1 STM32F7 外部中断简介
9.2 硬件设计
9.3 软件设计
9.4 下载验证
9.5 STM32CubeMX 配置外部中断

9.1 STM32F7 外部中断简介

STM32F7 的 IO 口在第六章有详细介绍，而中断优先级分组管理在前面也有详细的阐述。这里我们将介绍 STM32F7 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。

这里我们首先讲解 STM32F7 IO 口中断的一些基础概念。STM32F7 的每个 IO 都可以作为外部中断的中断输入口，这点也是 STM32F7 的强大之处。STM32F7 的中断控制器支持 22 个外部中断/事件请求。每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置。STM32F7 的 23 个外部中断为：

EXTI 线 0~15：对应外部 IO 口的输入中断。
EXTI 线 16：连接到 PVD 输出。
EXTI 线 17：连接到 RTC 闹钟事件。
EXTI 线 18：连接到 USB OTG FS 唤醒事件。
EXTI 线 19：连接到以太网唤醒事件。
EXTI 线 20：连接到 USB OTG HS(在 FS 中配置)唤醒事件。
EXTI 线 21：连接到 RTC 入侵和时间戳事件。
EXTI 线 22：连接到 RTC 唤醒事件。
EXTI 线 23：连接到 LPTIM1 异步事件。

从上面可以看出，中断线 0-15 对应外部 IO 口的输入中断，一共是 16 个外部中断线。STM32F7 供 IO 口使用的中断线只有 16 个，但是 STM32F7 的 IO 口却远远不止 16 个，那么 STM32F7 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 的引脚 GPIOx.0~GPIOx.15(x=A,B,C,D,E，F,G,H,I)分别对应中断线 0~15。这样每个中断线对应了最多 9 个 IO 口，以线 0 为例：它对应了 GPIOA.0、GPIOB.0、GPIOC.0、GPIOD.0、GPIOE.0、GPIOF.0、GPIOG.0,GPIOH.0,GPIOI.0。而中断线每次只能连接到 1 个 IO 口上，这样就需要通过配置来决定对应的中断线配置到哪个 GPIO 上了。下面我们看看 GPIO 跟中断线的映射关系图：

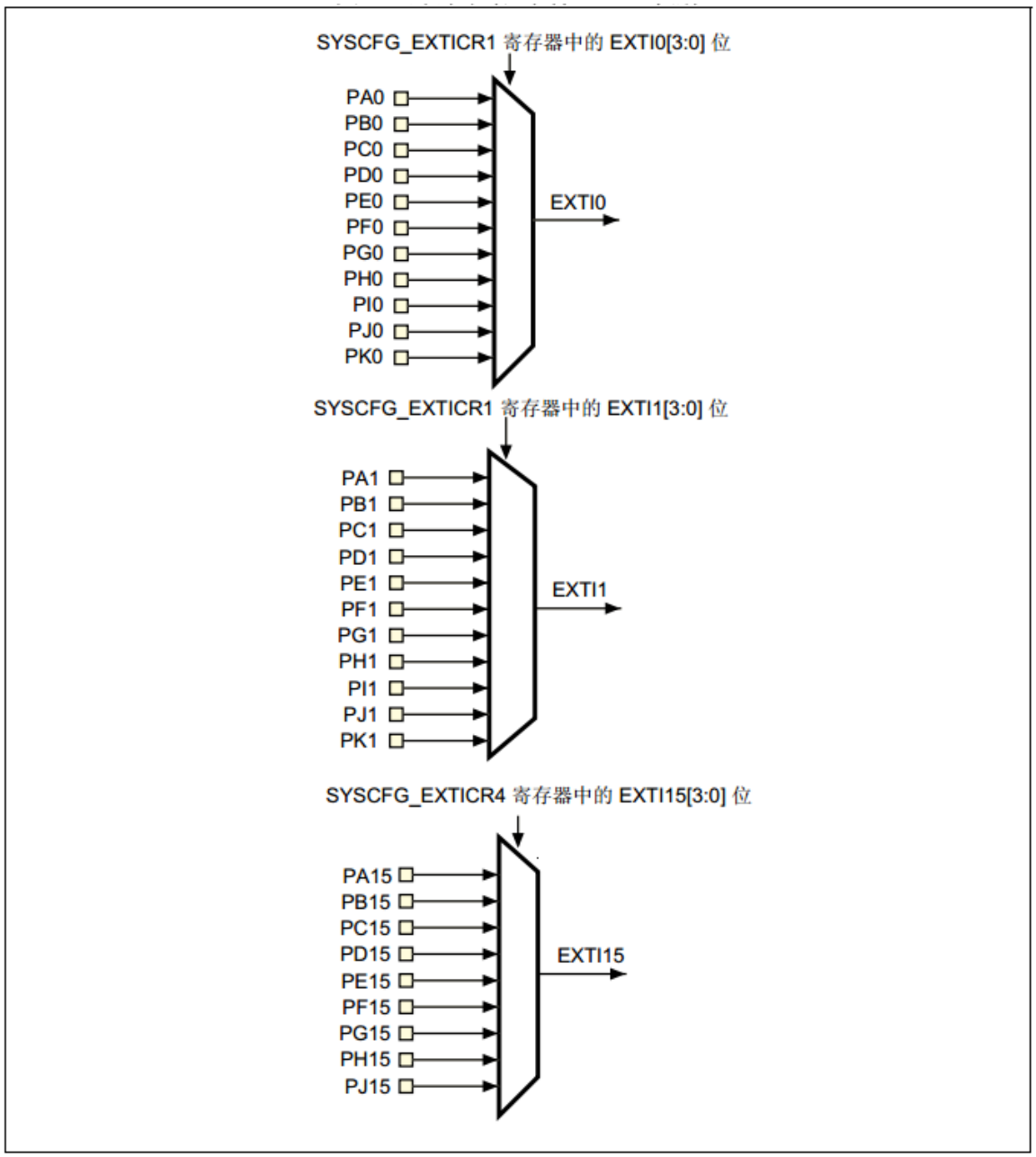


图 9.1.1 GPIO 和中断线的映射关系图

GPIO 和中断线映射关系是在寄存器 SYSCFG_EXTICR1~ SYSCFG_EXTICR4 中配置的。所以我们要配置外部中断，还需要打开 SYSCFG 时钟。

接下来我们来看看使用 HAL 库配置外部中断的一般步骤。HAL 中外部中断相关配置函数和定义在文件 stm32f7xx_hal_exti.h 和 stm32f7xx_hal_exti.c 文件中。

1) 使能 IO 口时钟。

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。

2) 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 9.1.1 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。

3) 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,2,1);       //抢占优先级为 2，子优先级为 1
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。

4) 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，STM32F7 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) != RESET)
  {
    __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，通过入口参数 GPIO_Pin 判断中断来自哪个 IO 口，然后清除相应的中断标志位，最后调用回调函数 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，在所有的外部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 实现控制逻辑。

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，还是 KEY_UP 控制 DS0，DS1 互斥点亮；KEY2 控制 DS0，按一次亮，再按一次灭；KEY1 控制 DS1，效果同 KEY2；KEY0 则同时控制 DS0 和 DS1，按一次，他们的状态就翻转一次。

9.2 硬件设计

本实验用到的硬件资源和第七章实验的一模一样，不再多做介绍了。

9.3 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_Initure;

    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟
    __HAL_RCC_GPIOC_CLK_ENABLE();           //开启 GPIOC 时钟
    __HAL_RCC_GPIOH_CLK_ENABLE();           //开启 GPIOH 时钟

    GPIO_Initure.Pin=GPIO_PIN_0;                //PA0
    GPIO_Initure.Mode=GPIO_MODE_IT_RISING;      //上升沿触发
    GPIO_Initure.Pull=GPIO_PULLDOWN;            //下拉
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    GPIO_Initure.Pin=GPIO_PIN_13;               //PC13
    GPIO_Initure.Mode=GPIO_MODE_IT_FALLING;     //下降沿触发
```

```
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    HAL_GPIO_Init(GPIOC,&GPIO_Initure);

    GPIO_Initure.Pin=GPIO_PIN_2|GPIO_PIN_3;         //PH2,3  下降沿触发，上拉
    HAL_GPIO_Init(GPIOH,&GPIO_Initure);

    //中断线 0
    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 2
    HAL_NVIC_SetPriority(EXTI2_IRQn,2,1);           //抢占优先级为 2，子优先级为 1
    HAL_NVIC_EnableIRQ(EXTI2_IRQn);                 //使能中断线 2

    //中断线 3
    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 13
    HAL_NVIC_SetPriority(EXTI15_10_IRQn,2,3);       //抢占优先级为 3，子优先级为 3
    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);             //使能中断线 13
}


//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 EXTI2_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);    //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 EXTI15_10_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);   //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)
{
    static u8 led0sta=1,led1sta=1;
    delay_ms(50);          //消抖
    switch(GPIO_Pin)
    {
        case GPIO_PIN_0:
            if(WK_UP==1)     //控制 LED0,LED1 互斥点亮
            {
                led1sta=!led1sta;
                led0sta=!led1sta;
                LED1(led1sta);
                LED0(led0sta);
            }
            break;
        case GPIO_PIN_2:
            if(KEY1==0)  //控制 LED1 翻转
            {
                led1sta=!led1sta;
                LED1(led1sta);
            };
            break;
        case GPIO_PIN_3:
            if(KEY0==0)        //同时控制 LED0,LED1 翻转
            {
                led1sta=!led1sta;
                led0sta=!led0sta;
                LED1(led1sta);
                LED0(led0sta);
            }
            break;

        case GPIO_PIN_13:
            if(KEY2==0)        //控制 LED0 翻转
            {
                led0sta=!led0sta;
                LED0(led0sta);
            }
```

```
                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。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 9.1 小节中关于 HAL_GPIO_Init 函数讲解，然后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 9.1 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。在回调函数内部，我们仅仅只是编写了简单的测试逻辑，通过不同的中断来源来控制 DS0 和 DS1 的状态。

接下来我们看看主函数，main 函数代码如下：

```
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    EXTI_Init();                    //外部中断初始化
    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，先进行各项初始化之后，在 while 死循环中不停的打印字符串到串口。当有某个外部按键按下之后，会触发中断服务函数做出相应的反应。

9.4 下载验证

在编译成功之后，我们就可以下载代码到阿波罗 STM32 开发板上，实际验证一下我们的程序是否正确。下载代码后，在串口调试助手里面可以看到如图 9.4.1 所示信息：

图 9.4.1 串口收到的数据

从图 9.4.1 可以看出，程序已经在运行了，此时可以通过按下 KEY0、KEY1、KEY2 和 KEY_UP 来观察 DS0、DS1 是否跟着按键的变化而变化。

9.5 STM32CubeMX 配置外部中断

本小节我们将教会大家用 STM32CubeMX 配置外部中断相关的初始化代码。关于 STM32CubeMX 的配置，从本小节开始，我们对于每个实验只讲解实验相关关键配置部分，如果大家不知道具体怎么进入相关界面，请仔细看看前面几个章节实验。

对于外部中断的配置，首先在 MCU 引脚配置图界面选择相应的 GPIO 设置其模式为外部中断模式。这里以 PH3 为例，如下图 9.5.1:

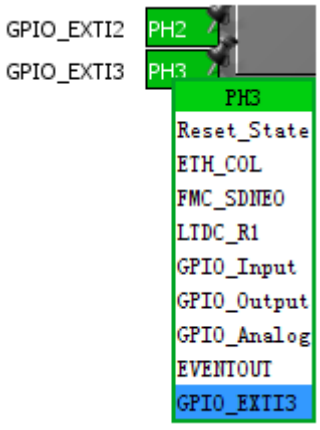


图 9.5.1 PH3 引脚配置

使用同样的方法依次配置 PC13 和 PH2/PH3，然后我们打开依次点击 Configuration->GPIO 进入 GPIO 详细配置界面 Pin Configuration，界面会列出 4 个 IO 口的配置信息。我们选中 PA0，看看此时 IO 口配置信息详情如下图 9.5.2 所示：

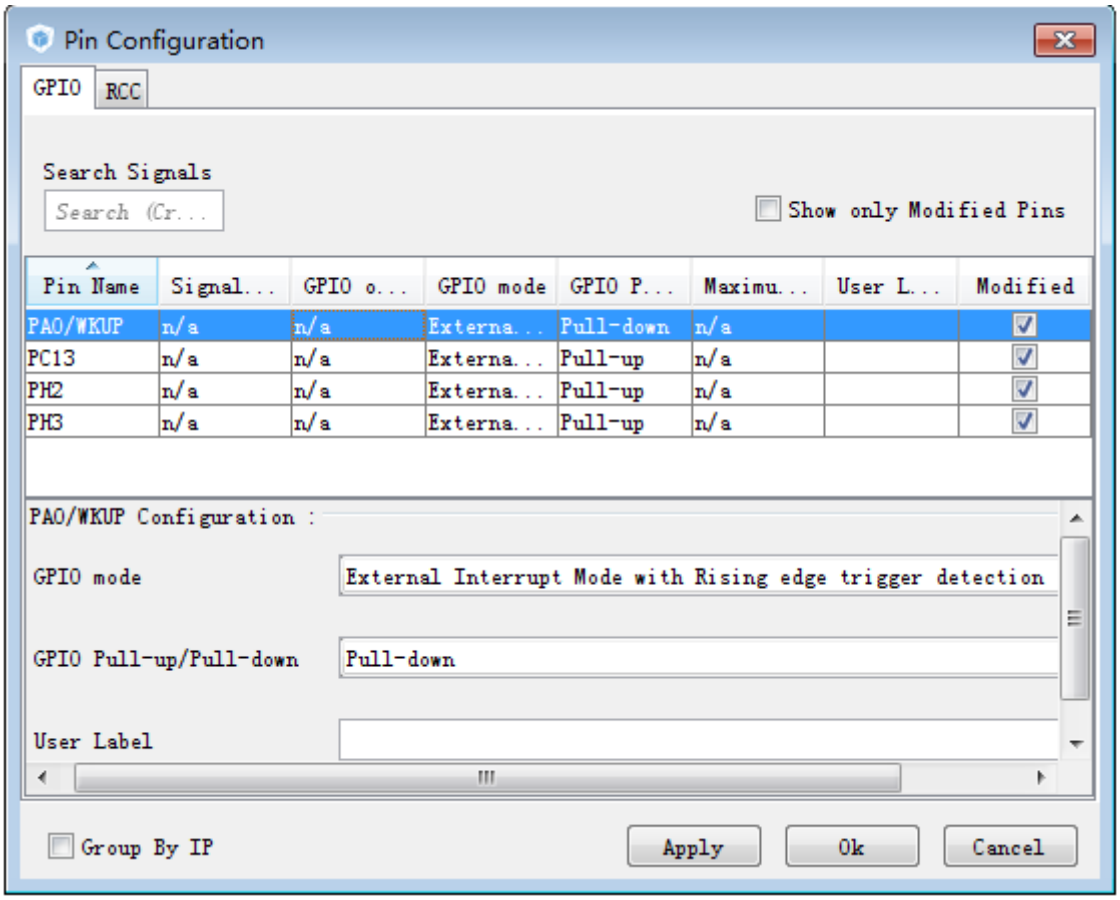


图 9.5.2 GPIO 配置详情界面

从上图界面可以看出，当我们配置 IO 口作为外部中断触发引脚之后，其详细配置界面便只有三个选项。第一个选项 GPIO mode 用来设置外部中断触发方法，上升沿触发还是下降沿触发还是双边沿触发。第二个选项 GPIO Pull-up/Pull-down 用来设置是默认上拉还是下拉。这两个参数根据我们前面讲解的外部中断知识就很好理解了。这里除了设置 PA0 为上升沿触发默认下拉外，其他 IO 口都设置为下降沿触发默认上拉。

配置好 IO 口信息之后，接下来就需要配置 NVIC 中断优先级设置。依次点击 Configuration->NVIC，进入 NVIC 配置界面。在界面可以看到有四个外部中断线可配置，这是因为我们前面开启了四个 IO 口的外部中断（对应 4 个外部中断线）。我们按照实验讲解，依次配置四个中断线的 NVIC 即可，配置完成后如下图 9.5.3 所示：

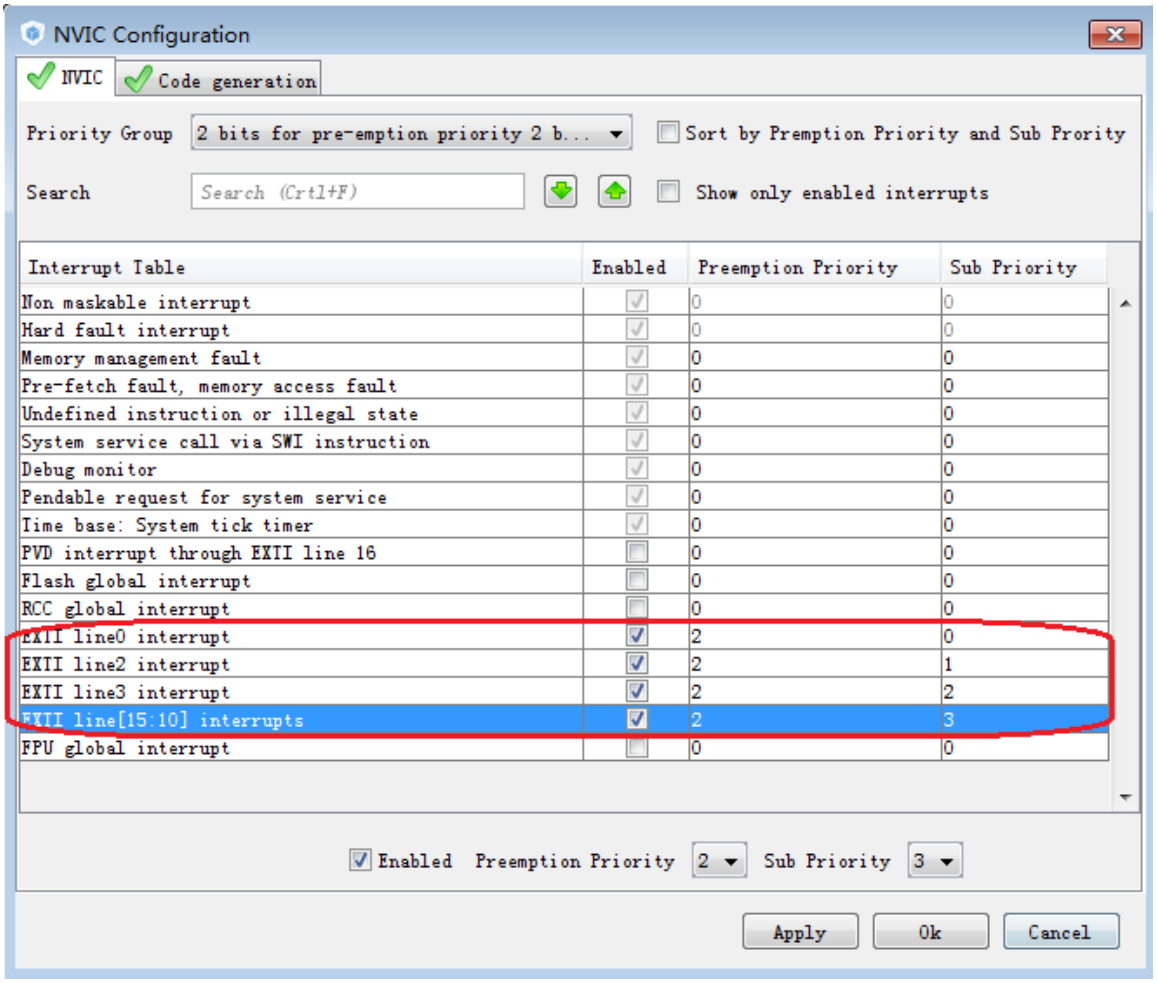


图 9.5.3 NVIC 配置界面

最后生成工程，为了篇幅考虑，这里我们就不再列出生成的关键代码。在 main.c 中生成的函数 MX_GPIO_Init 和我们实验工程中 exti.c 文件中的函数 EXTI_Init 内容一致，在 stm32f7xx_it.c 中生成了 4 个中断服务函数，和 exti.c 文件中的中断服务函数内容一致。当然，回调函数 HAL_GPIO_EXTI_Callback 的内容软件是无法自动生成的，需要我们自己编写。

第十章 独立看门狗（IWDG）实验

这一章，我们将向大家介绍如何使用 STM32F7 的独立看门狗（以下简称 IWDG）。STM32F7 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：

10.1 STM32F7 独立看门狗简介
10.2 硬件设计
10.3 软件设计
10.4 下载验证
10.5 STM32CubeMX 配置 IWDG

10.1 STM32F7 独立看门狗简介

STM32F767 的独立看门狗由内部专门的 32Khz 低速时钟（LSI）驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 32Khz，而是在 17~47Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 32Khz 的频率来计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

独立看门狗有几个寄存器与我们这节相关，我们分别介绍这几个寄存器，首先是键值寄存器 IWDG_KR，该寄存器的各位描述如图 10.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																KEY[15:0]															
																w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位 31:16 保留，必须保持复位值。
位 15:0 **KEY[15:0]**：键值 (Key value)（只写位，读为 0000h）
必须每隔一段时间便通过软件对这些位写入键值 AAAAh，否则当计数器计数到 0 时，看门狗会产生复位。
写入键值 5555h 可使能对 IWDG_PR 和 IWDG_RLR 寄存器的访问
写入键值 CCCCh 可启动看门狗（选中硬件看门狗选项的情况除外）

图 10.1.1 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 10.1.2 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
Reserved																													PR[2:0]		
																													rw	rw	rw

位 31:3 保留，必须保持复位值。

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

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

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

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

图 10.1.2 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 10.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																				RL[11:0]											
																				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:12 保留，必须保持复位值。

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

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

若要更改重载值，IWDG_SR 中的 RVU 位必须为 0。

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

图 10.1.3 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 STM32F7 的独立看门狗。

这里我们还要特别说明一下，STM32F7 的独立看门狗还可以当做窗口看门狗使用，这是通过配置窗口寄存器 IWDG_WINR 来实现的。当我们没有设置 IWDG_WINR 寄存器的时候，独立看门狗就是我们前面讲解的工作过程，窗口计数器从其复位值 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)，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。如果我们设置了 IWDG_WINR 寄存器的值（不等于 0xFFF），那么当计数器值大于窗口值（IWDG_WINR）的值的时候如果执行重装操作，则会产生复位。所以我们必须在计数器的值在 IWDG_WINR 和 0 之间的时候执行重载，也就形成了一个窗口的概念。本实验我们将不设置 IWDG_WINR 寄存器的值，也就是不开启窗口功能。

独立看门狗相关的 HAL 库操作函数在文件 stm32f7xx_hal_iwdg.c 和头文件 stm32f7xx_hal_iwdg.h 中。

接下来我们讲解一下通过 HAL 库配置独立看门狗的步骤：

1）取消寄存器写保护，设置看门狗预分频系数和重装载值

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 3 个成员变量，分别用来设置独立看门狗的预分频系数，重装载值以及窗口值，定义如下：

```
typedef struct
{
  uint32_t Prescaler;     //预分频系数
  uint32_t Reload;        //重装载值
  uint32_t Window;        //窗口值
}IWDG_InitTypeDef;
```

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
IWDG_Handler.Init.Window=IWDG_WINDOW_DISABLE;//关闭窗口功能
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，同时关闭窗口功能。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是通过宏定义标识符来实现的：

```
#define __HAL_IWDG_START(__HANDLE__)
            WRITE_REG((__HANDLE__)->Instance->KR, IWDG_KEY_ENABLE);
```

所以我们只需要调用宏定义标识符__HAL_IWDG_START 即可实现看门狗使能。实际上，当我们调用了看门狗初始化函数 HAL_IWDG_Init 之后，在内部会调用该标识符来实现看门狗启动。

通过上面 3 个步骤，我们就可以启动 STM32F7 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 STM32F7 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。

10.2 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 STM32F767 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。

10.3 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 stm32f7xx_hal_iwdg.h 和 stm32f7xx_hal_iwdg.c 文件。

iwdg.c 代码如下：

```
#include "iwdg.h"
#include "sys.h"

IWDG_HandleTypeDef IWDG_Handler; //独立看门狗句柄

//初始化独立看门狗
//prer:分频数:0~7(只有低 3 位有效!)
//rlr:自动重装载值,0~0XFFF.
//分频因子=4*2^prer.但最大值只能是 256!
//rlr:重装载寄存器值:低 11 位有效.
//时间计算(大概):Tout=((4*2^prer)*rlr)/32 (ms).
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG_Handler.Instance=IWDG;
    IWDG_Handler.Init.Prescaler=prer;        //设置 IWDG 分频系数
    IWDG_Handler.Init.Reload=rlr;            //重装载
```

```
    IWDG_Handler.Init.Window=IWDG_WINDOW_DISABLE;//关闭窗口功能
    HAL_IWDG_Init(&IWDG_Handler);
}

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

该代码就 2 个函数，void IWDG_Init(u8 prer，u16 rlr)是独立看门狗初始化函数，就是按照上面介绍的步骤 1 来初始化独立看门狗。该函数有 2 个参数，分别用来设置预分频数与重装载寄存器的值。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)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    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);                                    //先点亮红灯
    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。

10.4 下载验证

在编译成功之后，我们就可以下载代码到阿波罗 STM32 开发板上，实际验证一下，我们的程序是否正确。下载代码后，可以看到 DS0 不停的闪烁，证明程序在不停的复位，否则只会 DS0 常亮。这时我们试试不停的按 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。

10.5 STM32CubeMX 配置 IWDG

使用 STM3CubeMX 工具配置 IWDG 生成初始化代码的步骤非常简单，我们只需要使能 IWDG，同时配置 IWDG 的预分频系数和自动装载值即可。

首先我们看看使能 IWDG 的方法，在 Pinout 界面的 Peripherals 一栏选择 IWDG，然后勾选上 Activated 选项即可使能 IWDG。操作方法如下图 10.5.1：

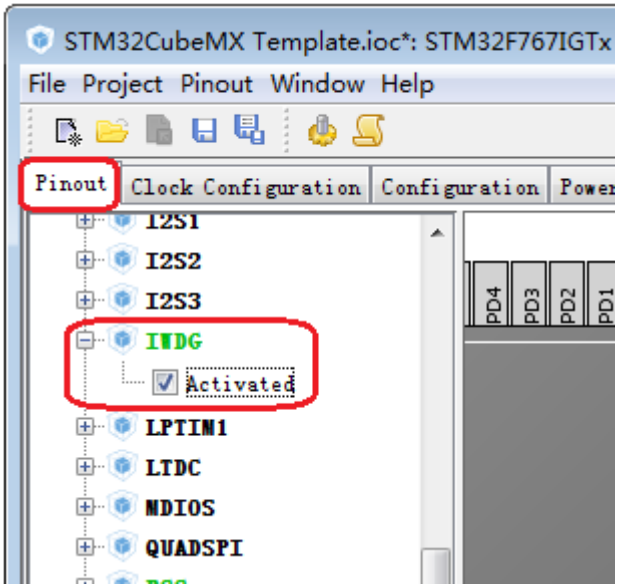


图 10.5.1 IWDG 配置选项

接下来依次点击 Configuration->IWDG，进入 IWDG 参数配置界面。进入界面后，我们依次配置 IWDG 的预分频系数，窗口值和自动装载值，这三个参数的含义我们在前面已经讲解。IWDG Configuration 配置界面如下图 10.5.2 所示：

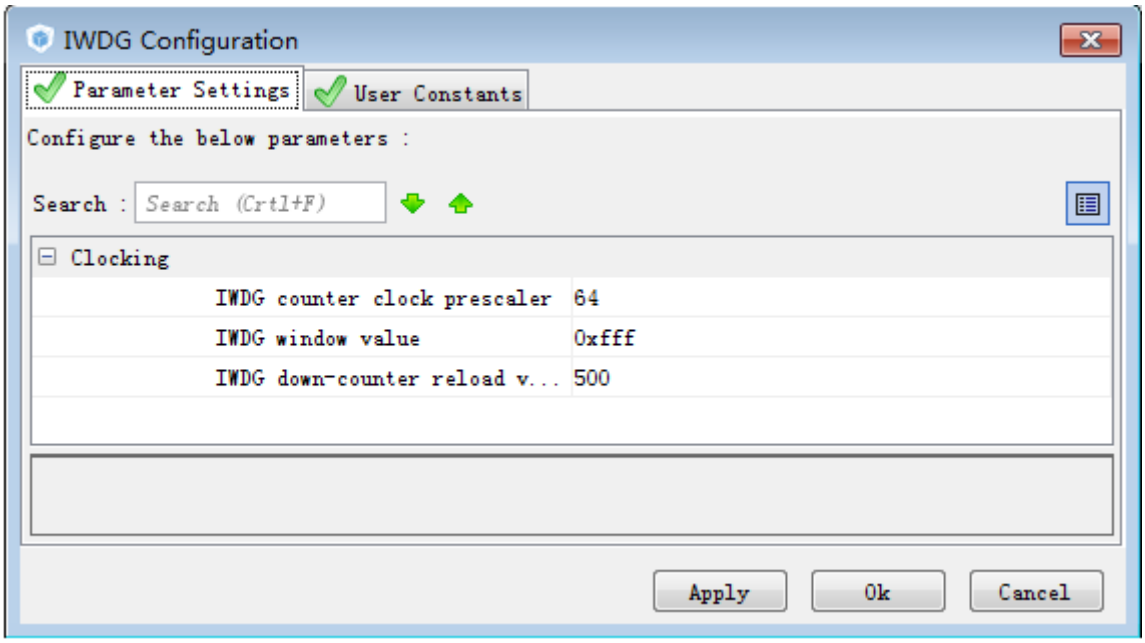


图 10.5.2 IWDG 参数配置界面

这里我们配置预分频系数为 64，同时自动装载值为 500 即可，窗口值配置为缺省的最大值 0xfff 也就是默认关闭窗口功能。配置完成后生成实验工程。在生成的工程中，打开 main.c 文件可以看到生成的函数 MX_IWDG_Init 和看门狗实验工程中的 IWDG_Init 函数内容一样，不同的是 IWDG_Init 函数的这两个参数是通过入口参数传入的。当然，对于何时启动看门狗以及何时喂狗的操作软件是无法确定的，还需要用户根据自己需求在合适的程序段中编写这两项操作。

第十一章 窗口门狗（WWDG）实验

这一章，我们将向大家介绍如何使用 STM32F7 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：

11.1 STM32F7 窗口看门狗简介

11.2 硬件设计

11.3 软件设计

11.4 下载验证

11.5 STM32CubeMX 配置 WWDG

11.1 STM32F7 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 11.1.1 来说明：

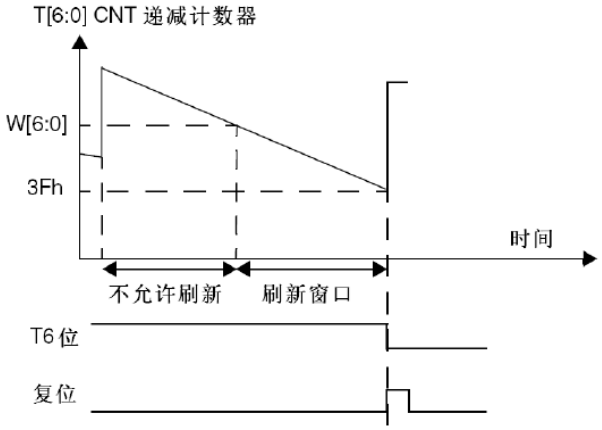


图 11.1.1 窗口看门狗工作示意图

图 11.1.1 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：

$$Twwdg=(4096\times 2^{WDGTB}\times(T[5:0]+1))/Fpclk1;$$

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=54Mhz，那么可以得到最小-最大超时时间表如表 11.1.1 所

示：

WDGTB	最小超时(us) T[5:0]=0X00	最大超时(ms) T[5:0]=0X3F
0	75.85	4.85
1	151.70	9.71
2	303.41	19.42
3	606.81	38.84

表 11.1.1 54M 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 11.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								WDGA	T[6:0]						
								rs	rw						

图 11.1.2 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 11.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						EWI	WDGTB[1:0]		W[6:0]						
						rs	rw		rw						

位 31:10 保留，必须保持复位值。

位 9 **EWI**：提前唤醒中断 (Early wakeup interrupt)
置 1 后，只要计数器值达到 0x40 就会产生中断。此中断只有在复位后才由硬件清零。

位 8:7 **WDGTB[1:0]**：定时器时基 (Timer base)
可按如下方式修改预分频器的时基：
00：CK 计数器时钟 (PCLK1 div 4096) 分频器 1
01：CK 计数器时钟 (PCLK1 div 4096) 分频器 2
10:CK 计数器时钟 (PCLK1 div 4096) 分频器 4
11:CK 计数器时钟 (PCLK1 div 4096) 分频器 8

位 6:0 **W[6:0]**：7 位窗口值 (7-bit window value)
这些位包含用于与递减计数器进行比较的窗口值。

图 11.1.3 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 54M 且 WDGTB 为 0 的条件下，

该时间为 75.85us）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 STM32F7 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 stm32f7xx_hal_wwdg.c 文件和头文件 stm32f7xx_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 32Khz 时钟所以不存在使能问题。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，Instance 成员变量设置为值 WWDG 即可。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;   //提前唤醒中断使能
}WWDG_InitTypeDef;
```

该结构体有 4 三个成员变量，分别用来设置 WWDG 的预分频系数，窗口之以，计数器值以及是否开启提前唤醒中断。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
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
```

3）开启 WWDG

HAL 库中开启 WWDG 是通过宏定义标识符实现：

```
#define __HAL_WWDG_ENABLE(__HANDLE__)
                                SET_BIT((__HANDLE__)->Instance->CR, WWDG_CR_WDGA)
```

这里需要说明一下的是，在调用函数 HAL_WWDG_Init 之后，该函数会开启窗口看门狗，所以不需要再重复开启。

4）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。

5）编写中断服务函数

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

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, uint32_t    cnt);
```

WWDG 的喂狗操作实际就是往 CR 寄存器重写计数器值，这里的第二个入口函数就是重写的计数器的值。

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

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_EarlyWakeupCallback，所以提前唤醒中断逻辑我们一般些在回调函数 HAL_WWDG_EarlyWakeupCallback 中。 回调函数声明为：

```
__weak void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg);
```

完成了以上 6 个步骤之后，我们就可以使用 STM32F7 的窗口看门狗了。这一章的实验，我们将通过 DS0 来指示 STM32F7 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。

11.2 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 STM32F767 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 STM32F767 的复位情况和窗口看门狗的喂狗情况。

11.3 软件设计

打开窗口看门狗实验可以看到，我们增加了窗口看门狗相关的库函数支持文件 stm32f7xx_hal_wwdg.c 和 stm32f7xx_hal_wwdg.h，同时新建 wwdg.c 和对应的头文件 wwdg.h 用来编写窗口看门狗相关的函数代码。

接下来我们看看 wwdg.c 文件内容如下：

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

//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=54Mhz
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);
}

//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_Toggle;
}
```

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，同时还使能看门狗提前唤醒中断。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_EarlyWakeupCallback 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。

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)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    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阿波罗 STM32F7 开发板上，看看结果是不是和我们设计的一样。

11.4 下载验证

将代码下载到阿波罗 STM32F7 后，可以看到 DS0 亮一下之后熄灭，紧接着 DS1 开始不停的闪烁。每秒钟闪烁 20 次左右，和我们预期的一致，说明我们的实验是成功的。

11.5 STM32CubeMX 配置 WWDG

上一讲我们讲解了使用 STM32CubeMX 配置 IWDG 步骤，而 WWDG 配置过程和 IWDG 配置过程基本是一模一样的，这里我们就直接列出配置图，对配置过程不做过多讲解。首先进入 Pinout 选项界面，使能 WWDG，如下图 11.5.1：

图 11.5.1 使能 WWDG

接下来配置 WWDG 的参数，进入 Configuration->WWDG 界面，如下图所示：

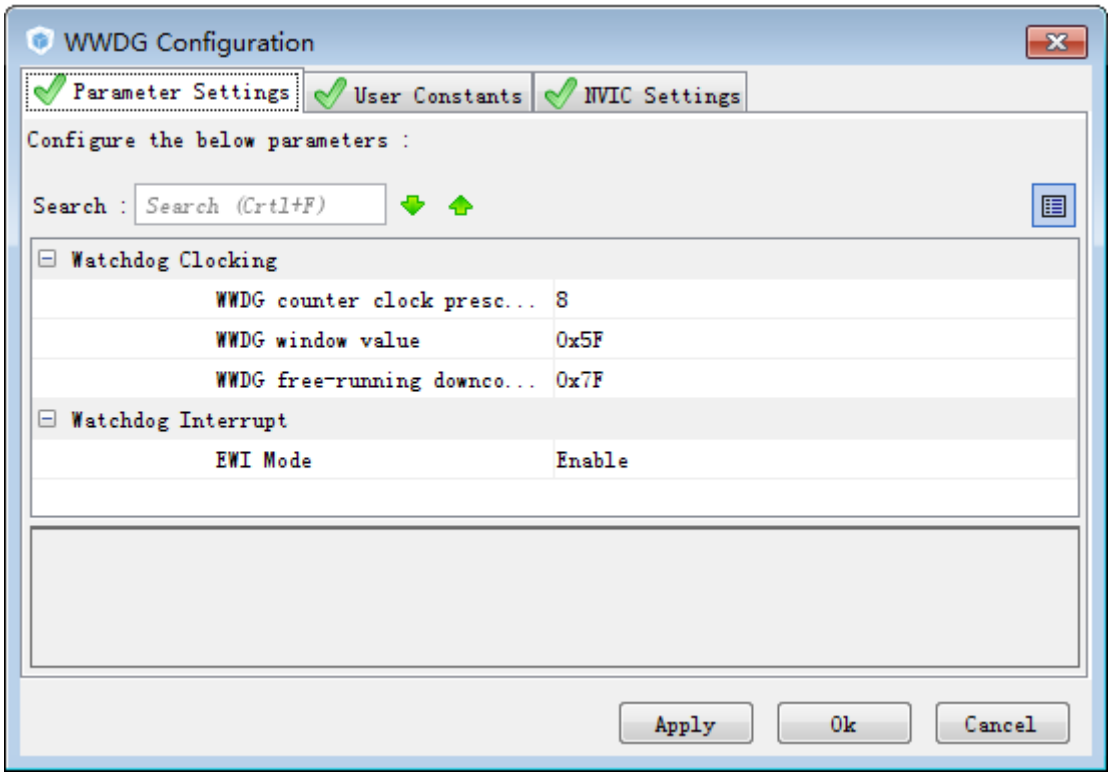


图 11.5.2 WWDG Configuration 配置界面

配置栏 Watchdog Clocking 有三个配置参数，顾名思义，第一个参数是配置分频系数，这里我们配置为 8。第二个参数是配置窗口值，第三个参数是配置计数器初始值。配置栏 Watchdog Interrupt 只有一个配置项 EWI Mode，也就是是否使能 WWDG 的提前唤醒中断，这里我们选择 Enable。配置好上述参数之后，因为我们开启了提前唤醒中断，所以这里我们要配置 NVIC 中断优先级。进入 Configuration->NVIC 界面，参考第九章配置方法配置 WWDG 中断优先级即可。配置方法如下图 11.5.3 所示：

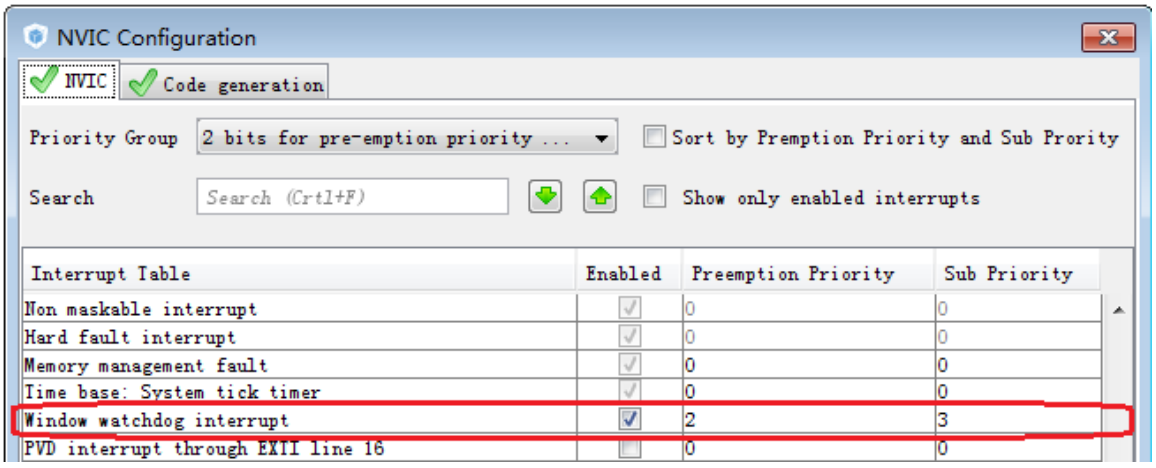


图 11.5.3 配置 WWDG 中断

配置完成之后直接生成工程源码。在 main.c 文件中生成的 MX_WWDG_Init 函数和本实验的 WWDG_Init 函数实现功能类似。在 stm32f7xx_it.c 中生成的中断服务函数和我们实验一致。在 stm32f7xx_hal_msp.c 文件中生成的 MSP 回调函数内容核我们实验内容一致。

第十二章 定时器中断实验

这一章，我们将向大家介绍如何使用 STM32F767 的通用定时器，STM32F767 的定时器功能十分强大，有 TIM1 和 TIM8 等高级定时器，有 LPTIM1 低功耗定时器，也有 TIM2~TIM5，TIM9~TIM14 等通用定时器，还有 TIM6 和 TIM7 等基本定时器，总共达 15 个定时器之多。在本章中，我们将使用 TIM3 的定时器中断来控制 DS1 的翻转，在主函数用 DS0 的翻转来提示程序正在运行。本章，我们选择难度适中的通用定时器来介绍，本章将分为如下几个部分：

12.1 STM32F7 通用定时器简介
12.2 硬件设计
12.3 软件设计
12.4 下载验证
12.5 STM32CubeMX 配置定时器更新中断功能

12.1 STM32F7 通用定时器简介

STM32F767 的通用定时器包含一个 16 位或 32 位自动重载计数器（CNT），该计数器由可编程预分频器（PSC）驱动。STM32F767 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。STM32F767 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。

STM32 的通用 TIMx (TIM2~TIM5 和 TIM9~TIM14)定时器功能包括：

1)16 位/32 位(仅 TIM2 和 TIM5)向上、向下、向上/向下自动装载计数器（TIMx_CNT），注意：TIM9~TIM14 只支持向上（递增）计数方式。

2)16 位可编程(可以实时修改)预分频器(TIMx_PSC)，计数器时钟频率的分频系数为 1～65535 之间的任意数值。

3）4 个独立通道（TIMx_CH1~4，TIM9~TIM14 最多 2 个通道），这些通道可以用来作为：

A．输入捕获
B．输出比较
C．PWM 生成(边缘或中间对齐模式) ，注意：TIM9~TIM14 不支持中间对齐模式
D．单脉冲模式输出

4）可使用外部信号（TIMx_ETR）控制定时器和定时器互连（可以用 1 个定时器控制另外一个定时器）的同步电路。

5）如下事件发生时产生中断/DMA（TIM9~TIM14 不支持 DMA）：

A．更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)
B．触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)
C．输入捕获
D．输出比较
E．支持针对定位的增量(正交)编码器和霍尔传感器电路（TIM9~TIM14 不支持）
F．触发输入作为外部时钟或者按周期的电流管理（TIM9~TIM14 不支持）

由于 STM32F767 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《STM32F7 中文参考手册》第 650 页，通用定时器一章。下面我们介绍一下与我们这章的实验密切相关的几个通用定时器的寄存器（以下均以 TIM2~TIM5 的寄存器介绍，TIM9~TIM14 的略有区别，具体请看《STM32F7 中文参考手册》对应章节）。

首先是控制寄存器 1（TIMx_CR1），该寄存器的各位描述如图 12.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	UIF RE-MAP	Res.	CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
				rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器

1：使能计数器

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

在单脉冲模式下，当发生更新事件时会自动将 CEN 位清零。

图 12.1.1 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 12.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

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

0：禁止更新中断

1：使能更新中断

图 12.1.2 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 12.1.3 所示：

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

位 15:0 **PSC[15:0]**：预分频器值 (Prescaler value)

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

PSC 包含在每次发生更新事件时要装载到实际预分频器寄存器的值。

图 12.1.3 TIMx_ PSC 寄存器各位描述

这里，定时器的时钟来源有 4 个：

1）内部时钟（CK_INT）

2）外部时钟模式 1：外部输入脚（TIx）

3）外部时钟模式 2：外部触发输入（ETR），仅适用于 TIM2、TIM3、TIM4

4）内部触发输入（ITRx）：使用 A 定时器作为 B 定时器的预分频器（A 为 B 提供时钟）。

这些时钟，具体选择哪个可以通过 TIMx_SMCR 寄存器的相关位来设置。这里的 CK_INT 时钟是从 APB1 倍频的来的，除非 APB1 的时钟分频数设置为 1（一般都不会是 1），否则通用定时器 TIMx 的时钟是 APB1 时钟的 2 倍，当 APB1 的时钟不分频的时候，通用定时器 TIMx 的时钟就等于 APB1 的时钟。这里还要注意的就是高级定时器以及 TIM9~TIM11 的时钟不是来自 APB1，而是来自 APB2 的。

这里顺带介绍一下 TIMx_CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前定时器的计数值。

接着我们介绍自动重装载寄存器（TIMx_ARR），该寄存器在物理上实际对应着 2 个寄存器。

一个是程序员可以直接操作的，另外一个是程序员看不到的，这个看不到的寄存器在《STM32F7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，才把预装载寄存器（ARR）的内容传送到影子寄存器。

自动重装载寄存器的各位描述如图 12.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ARR[31:16]（取决于定时器）															
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
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 **ARR[31:16]**：自动重载值的高 16 位（对于 TIM2 和 TIM5 ）

位 15:0 **ARR[15:0]**：自动重载值的低 16 位
ARR 为要装载到实际自动重载寄存器的值。
当自动重载值为空时，计数器不工作。

图 12.1.4 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 12.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

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

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

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

图 12.1.5 TIMx_ SR 寄存器各位描述

关于这些位的详细描述，请参考《STM32F7 中文参考手册》第 699 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 stm32f7xx_hal_tim.h 和 stm32f7xx_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=8999;                       //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。

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） 定时器 TIM3

本章将通过 TIM3 的中断来控制 DS1 的亮灭，DS1 是直接连接到 PB0 上的，这个前面已经有介绍了。而 TIM3 属于 STM32F767 的内部资源，只需要软件设置即可正常工作。

12.3 软件设计

打开我们光盘实验 7 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 文件 stm32f7xx_hal_tim.c 和头文件 stm32f7xx_hal_tim.h。timer.c 文件代码如下：

```
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!(定时器 3 挂在 APB1 上，时钟为 HCLK/2)
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);          //初始化定时器 3

    HAL_TIM_Base_Start_IT(&TIM3_Handler);    //使能定时器 3 和定时器 3 更新中断
}

//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)
{
    __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 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);
}

//定时器 3 中断服务函数调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&TIM3_Handler))
    {
        LED1_Toggle;            //LED0 反转
    }
}
```

该文件一共有 4 个函数。第一个函数 TIM3_Init 用来初始化定时器 3，使能定时器 3 更新中断以及使能定时器，实现的是 12.1 小节讲解的步骤 2 和步骤 3 配置功能。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 4 分频，所以 APB1 的时钟为 54M，而从 STM32F7 的内部时钟树图（图 4.3.1.1）得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 以及 TIM12~14 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 以及 TIM12~14 的时钟频率将为 APB1 时钟的两倍。因此，TIM3 的时钟为 108M，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）。

第二个函数 HAL_TIM_Base_MspInit 是定时器初始化回调函数，主要是使能定时器 3 时钟以及定时器 3 的 NVIC 配置，实现的是 12.1 小节讲解的步骤 1 和步骤 4 功能。第三个函数 TIM3_IRQHandler 是中断服务入口函数，该函数内部只有一行代码就是调用定时器中断共用处理函数 HAL_TIM_IRQHandler。根据前面的讲解，函数 HAL_TIM_IRQHandler 内部会判断中断来源，根据中断来源调用不同的中断处理回调函数。这里我们开启的是定时器 3 的更新中断，所以我们需要重定义更新中断回调函数 HAL_TIM_PeriodElapsedCallback。第四个函数 HAL_TIM_PeriodElapsedCallback 就是更新中断回调函数，也就是真正的中断处理函数，该函数内部通过判断中断是定时器 3 之后，然后控制 LED1 翻转。

timer.h 头文件内容比较简单，这里我们就不做讲解。

最后，我们看看主函数代码如下： ：

```
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    TIM3_Init(5000-1,10800-1);      //定时器 3 初始化，定时器时钟为 108M，分频系数
                //为 10800-1，//所以定时器 3 的频率为 108M/10800=10K，自动重装载为
```

```
                    //5000-1，那么定时器周期就是 500ms
        while(1)
        {
            LED0_Toggle;                //LED0 翻转
            delay_ms(200);              //延时 200ms
        }
    }
```

这里的代码和之前大同小异，此段代码对 TIM3 进行初始化之后，进入死循环等待 TIM3 溢出中断，当 TIM3_CNT 的值等于 TIM3_ARR 的值的时候，就会产生 TIM3 的更新中断，然后在中断里面取反 LED1，TIM3_CNT 再从 0 开始计数。

这里定时器定时时长500ms是这样计算出来的，定时器的时钟为108Mhz，分频系数为10799，所以分频后的计数频率为 108Mhz/（10799+1）=10KHz,然后计数到 4999，所以时长为（4999+1）/10000=0.5s，也就是 500ms。

12.4 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阿波罗 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停闪烁（每 400ms 闪烁一次），而 DS1 也是不停的闪烁，但是闪烁时间较 DS0 慢（1s 一次）。

12.5 STM32CubeMX 配置定时器更新中断功能

经过前面多个章节的学习，大家对 STM32CubeMX 配置已经非常熟悉。从本章开始，出于篇幅考虑，我们将不再像之前章节一样讲解那么详细，我们将只会列出配置的关键点，然后生成工程，大家自行与光盘中提供的实验代码对照学习。

定时器 3 中断配置非常简单。配置步骤如下：

① 在 Pinout->TIM3 配置项中，配置 Clock Source 为 Internal Clock，如下图 12.5.1 所示：

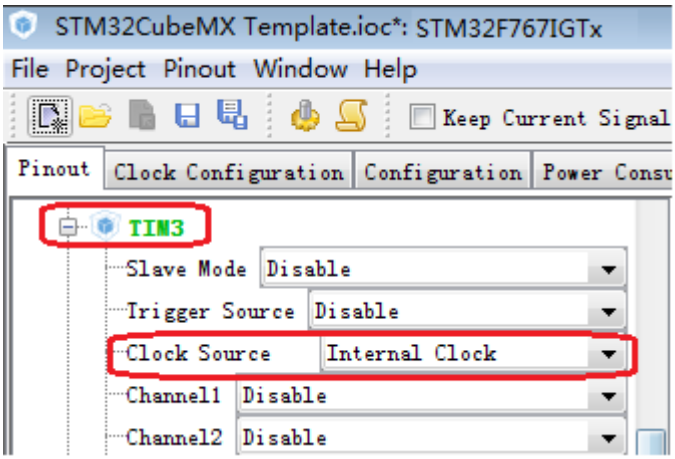


图 12.5.1 TIM3 配置

② 进入 Configuration 选项卡会发现，在 Control 栏下多出了 TIM3 按钮。点击 TIM3 按钮进入 TIM3 配置页，在弹出的界面中点击 Parameter Settings 选项卡，Counter Settings 配置栏下面的四个选项就是用来配置定时器的预分频系数，自动装载值，计数模式以及时钟分频因子。操作方法和配置值如下图 12.5.2 所示：

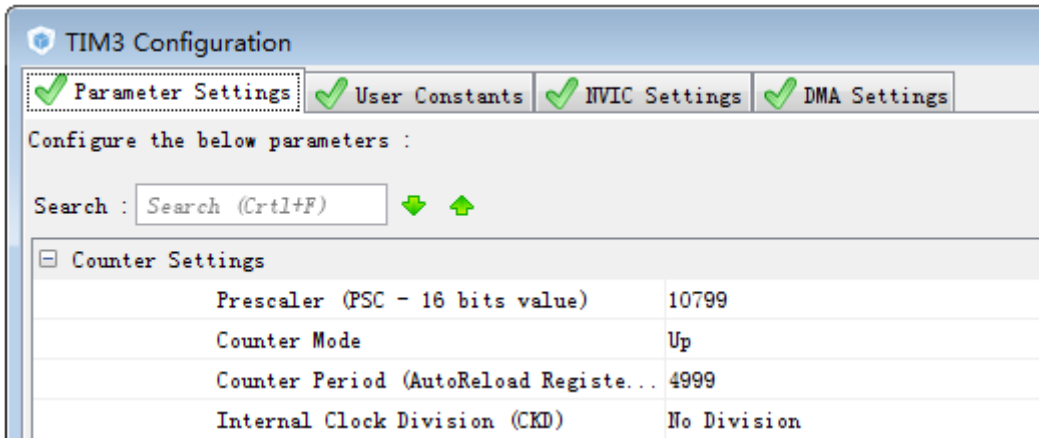


图 12.5.2 TIM3 参数设置界面

③ 进入 Configuration->NVIC 配置页，在弹出的界面中点击 NVIC 选项卡，配置 Interrupt Table 中的 TIM3 global interrupt，使能中断，配置抢占优先级为 1 和响应优先级为 3。

经过上面三个步骤，生成代码，大家对比生成的代码和实验工程的区别。这里需要说明的是，默认情况下，TIM3 的时钟来源是内部时钟 CK_INT，所以在我们实验中使用的是默认配置，没有额外在程序中体现。

第十三章 PWM 输出实验

上一章，我们介绍了 STM32F7 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 STM32F7 的 TIM3 来产生 PWM 输出。在本章中，我们将使用 TIM3 的通道 4 来产生 PWM 来控制 DS0 的亮度。本章分为如下几个部分：

13.1 PWM 简介
13.2 硬件设计
13.3 软件设计
13.4 下载验证
13.5 STM32CubeMX 配置定时器 PWM 输出功能

13.1 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 13.1.1 所示：

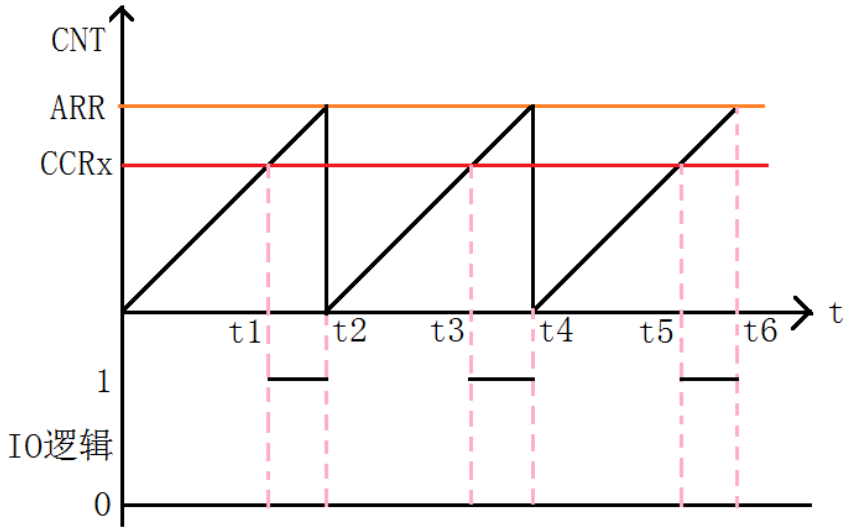


图 13.1.1 PWM 原理示意图

图 13.1.1 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。

STM32F767 的定时器除了 TIM6 和 7。其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出！这里我们仅使用 TIM3 的 CH4 产生一路 PWM 输出。

要使 STM32F767 的通用定时器 TIMx 产生 PWM 输出，除了上一章介绍的寄存器外，我们还会用到 3 个寄存器，来控制 PWM 的。这三个寄存器分别是：捕获/比较模式寄存器（TIMx_CCMR1/2）、捕获/比较使能寄存器（TIMx_CCER）、捕获/比较寄存器（TIMx_CCR1~4）。接下来我们简单介绍一下这三个寄存器。

首先是捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx _CCMR1

和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 2，而 TIMx_CCMR2 控制 CH3 和 4。以下我们将以 TIM3 为例进行介绍。TIM3_CCMR2 寄存器各位描述如图 13.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC4M [3]	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC3M [3]
							Res.								Res.
							rw								rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC4CE	OC4M[2:0]			OC4PE	OC4FE	CC4S[1:0]		OC3CE	OC3M[2:0]			OC3PE	OC3FE	CC3S[1:0]	
IC4F[3:0]				IC4PSC[1:0]				IC3F[3:0]				IC3PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 13.1.2 TIM3_CCMR2 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 13.1.2 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《STM32F7 中文参考手册》第 701 页，23.4.7 节。这里我们需要说明的是模式设置位 OC4M，此部分由 4 位组成。总共可以配置成 13 种模式，我们使用的是 PWM 模式，所以这 4 位必须设置为 0110/0111。这两种 PWM 模式的区别就是输出电平的极性相反。另外 CC4S 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 TIM3 的捕获/比较使能寄存器（TIM3_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 13.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

图 13.1.3 TIM3_ CCER 寄存器各位描述

该寄存器比较简单，我们这里只用到了 CC4E 位，该位是输入/捕获 4 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《STM32F7 中文参考手册》第 706 页，23.4.9 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 个通道 CH1~4。我们使用的是通道 4，TIM3_CCR4 寄存器的各位描述如图 13.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CCR4[31:16] (depending on timers)															
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
CCR4[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 **CCR4[31:16]**：捕获/比较 4 的高 16 位（对于 TIM2 和 TIM5）。

位 15:0 **CCR4[15:0]**：捕获/比较 4 的低 16 位 (Low Capture/Compare value)

1. 如果 CC4 通道配置为输出（CC4S 位）：
   CCR4 为要装载到实际捕获/比较 4 寄存器的值（预装载值）。
   如果没有通过 TIMx_CCMR 寄存器中的 OC4PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只有发生更新事件时，预装载值才会复制到活动捕获/比较 4 寄存器中。
   实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC4 输出上发出信号的值。
2. 如果 CC4 通道配置为输入（TIMx_CCMR4 寄存器中的 CC4S 位）：
   CCR4 为上一个输入捕获 4 事件 (IC4) 发生时的计数器值。

图 13.1.4 寄存器 TIM3_CCR4 各位描述

在输出模式下，该寄存器的值与 CNT 的值比较，根据比较结果产生相应动作。利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的输出脉宽了。

如果是通用定时器，则配置以上三个寄存器就够了，但是如果是高级定时器，则还需要配置：刹车和死区寄存器（TIMx_BDTR），该寄存器各位描述如图 13.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	BK2P	BK2E	BK2F[3:0]				BKF[3:0]			
						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
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

只要断路输入（BRK 或 BRK2）为有效状态，此位便由硬件异步清零。此位由软件置 1，也可根据 AOE 位状态自动置 1。此位仅对配置为输出的通道有效。

0：响应断路事件（2 个）。禁止 OC 和 OCN 输出

响应断路事件或向 MOE 写入 0 时：OC 和 OCN 输出被禁止或被强制为空闲状态，具体取决于 OSSI 位。

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

图 13.1.5 寄存器 TIMx_ BDTR 各位描述

该寄存器，我们只需要关注第 15 位：MOE 位，要想高级定时器的 PWM 正常输出，则必须设置 MOE 位为 1，否则不会有输出。注意：通用定时器不需要配置这个。其他位我们这里就不详细介绍了，请参考《STM32F7 中文参考手册》第 639 页，22.4.18 这一节。

本章，我们使用的是 TIM3 的通道 4，所以我们需要修改 TIM3_CCR4 以实现脉宽控制 DS0 的亮度。至此，我们把本章要用的几个相关寄存器都介绍完了，下面我们介绍通过 HAL 库来配置该功能的步骤。

首先要提到的是，PWM 实际跟上一章节一样使用的是定时器的功能，所以相关的函数设置同样在库函数文件 stm32f7xx_tim.h 和 stm32f7xx_tim.c 文件中。

1）开启 TIM3 和 GPIO 时钟，配置 PB1 选择复用功能 AF1（TIM3）输出。

要使用 TIM3，我们必须先开启 TIM14 的时钟，这点相信大家看了这么多代码，应该明白了。这里我们还要配置 PB1 为复用（AF1）输出，才可以实现 TIM13_CH4 的 PWM 经过 PB1 输出。 HAL 库使能 TIM3 时钟和 GPIO 时钟方法是：

```
__HAL_RCC_TIM3_CLK_ENABLE();            //使能定时器 3
__HAL_RCC_GPIOB_CLK_ENABLE();           //开启 GPIOB 时钟
```

接下来便是要配置 PB1 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;
GPIO_Initure.Pin=GPIO_PIN_1;            //PB1
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_HIGH;     //高速
GPIO_Initure.Alternate= GPIO_AF2_TIM3;  //PB1 复用为 TIM3_CH4
HAL_GPIO_Init(GPIOB,&GPIO_Initure);
```

在 IO 口初始化配置中，我们只需要将成员变量 Mode 配置为复用推挽输出，同时成员变量 Alternate 配置为 GPIO_AF2_TIM3，即可实现 PB1 映射为定时器 3 通道 4 的 PWM 输出引脚。

这里还需要说明一下，对于定时器通道的引脚关系，大家可以查看 STM32F7 对应的数据手册，比如我们 PWM 实验，我们使用的是定时器 3 的通道 4，对应的引脚 PB1 可以从数据手册表中查看：

PB1	I/O	FT	(4)	TIM1_CH3N, TIM3_CH4, TIM8_CH3N, LCD_R6, OTG_HS_ULPI_D2, ETH_MII_RXD3, EVENTOUT	ADC12_IN9

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_CH4 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_CH4 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 CCR4 的值小的时候，DS0 就暗，CCR4 值大的时候，DS0 就亮，所以我们要通过配置 TIM3_CCMR2 的相关位来控制 TIM3_CH4 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 1。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 4，所以取值为 TIM_CHANNEL_4 即可。

例如我们要初始化定时器 3 的通道 4 为 PWM 模式 1，输出极性为低，那么实例代码为：

```
TIM_OC_InitTypeDef TIM3_CH4Handler;         //定时器 3 通道 4 句柄
TIM3_CH4Handler.OCMode=TIM_OCMODE_PWM1; //模式选择 PWM1
TIM3_CH4Handler.Pulse=arr/2;                //设置比较值,此值用来确定占空比
TIM3_CH4Handler.OCPolarity=TIM_OCPOLARITY_LOW; //输出比较极性为低
HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH4Handler,TIM_CHANNEL_4);
```

4）使能 TIM3，使能 TIM3 的 CH4 输出。

在完成以上设置了之后，我们需要使能 TIM3 并且使能 TIM3_CH4 输出。在 HAL 库中，函数 HAL_TIM_PWM_Start 可以用来实现这两个功能，函数声明如下：

```
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,这里我们使能的是定时器的通道 4，值设置为 TIM_CHANNEL_4 即可。

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_CCR4 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_CCR4 则可以控制 CH4 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 TIM3 通道 4 的占空比
// compare:比较值
void TIM_SetTIM3Compare4(u32 compare)
{
    TIM3->CCR4=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_SetCompare4(TIM_TypeDef *TIMx,u32 compare)
{
TIM3_CH4Handler.Pulse=compare;
HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH4Handler,TIM_CHANNEL_4);
}
```

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。

13.2 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

这两个我们前面都已经介绍了，因为 TIM3_CH4 可以通过 PB1 输出 PWM，而 DS0 就是直接节在 PB1 上面的，所以电路上并没有任何变化。

13.3 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 3PWM 句柄
TIM_OC_InitTypeDef TIM3_CH4Handler;      //定时器 3 通道 4 句柄

//PWM 输出初始化
//arr：自动重装值  psc：时钟预分频数
void TIM3_PWM_Init(u16 arr,u16 psc)
{
    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_CH4Handler.OCMode=TIM_OCMODE_PWM1; //模式选择 PWM1
    TIM3_CH4Handler.Pulse=arr/2;   //设置比较值,此值用来确定占空比
    TIM3_CH4Handler.OCPolarity=TIM_OCPOLARITY_LOW; //输出比较极性为低
    HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH4Handler,
                              TIM_CHANNEL_4); //配置 TIM3 通道 4
    HAL_TIM_PWM_Start(&TIM3_Handler,TIM_CHANNEL_4);//开启 PWM 通道 4
}

//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;
    __HAL_RCC_TIM3_CLK_ENABLE();           //使能定时器 3
```

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

    GPIO_Initure.Pin=GPIO_PIN_1;           //PB1
    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_HIGH;    //高速
    GPIO_Initure.Alternate= GPIO_AF2_TIM3; //PB1 复用为 TIM3_CH4
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);
}

//设置 TIM 通道 4 的占空比
//compare:比较值
void TIM_SetTIM3Compare4(u32 compare)
{
    TIM3->CCR4=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;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    TIM3_PWM_Init(500-1,108-1);     //108M/108=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_SetTIM3Compare4(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。

13.4 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阿波罗 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 DS0 不停的由暗变到亮，然后又从亮变到暗。每个过程持续时间大概为 3 秒钟左右。

实际运行结果如下图 13.4.1 所示：

图 13.4.1 PWM 控制 DS0 亮度

13.5 STM32CubeMX 配置定时器 PWM 输出功能

使用 STM32CubeMX 配置 PWM 输出的配置步骤和配置定时器中断的配置步骤非常接近，步骤如下：

① 在 Pinout->TIM3 配置项中，配置 Channel4 的值为 PWM generation CH4，然后 Clock Source 为 Internal Clock。操作过程如下图 13.5.1 所示：

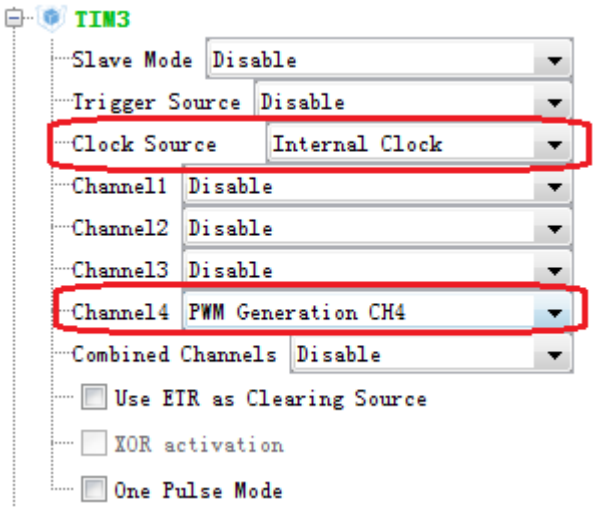


图 13.5.1 TIM3 配置

② 进入 Configuration->TIM3 配置页，在弹出的界面中点击 Parameter Settings 选项卡，Counter Settings 配置栏下面的四个选项就是用来配置定时器的预分频系数，自动装载值，计数模式以及时钟分频因子。在界面的 PWM Generation Channel4 配置栏配置 PWM 模式，比较值，极性等参数，操作方法如下图 13.5.2 所示：

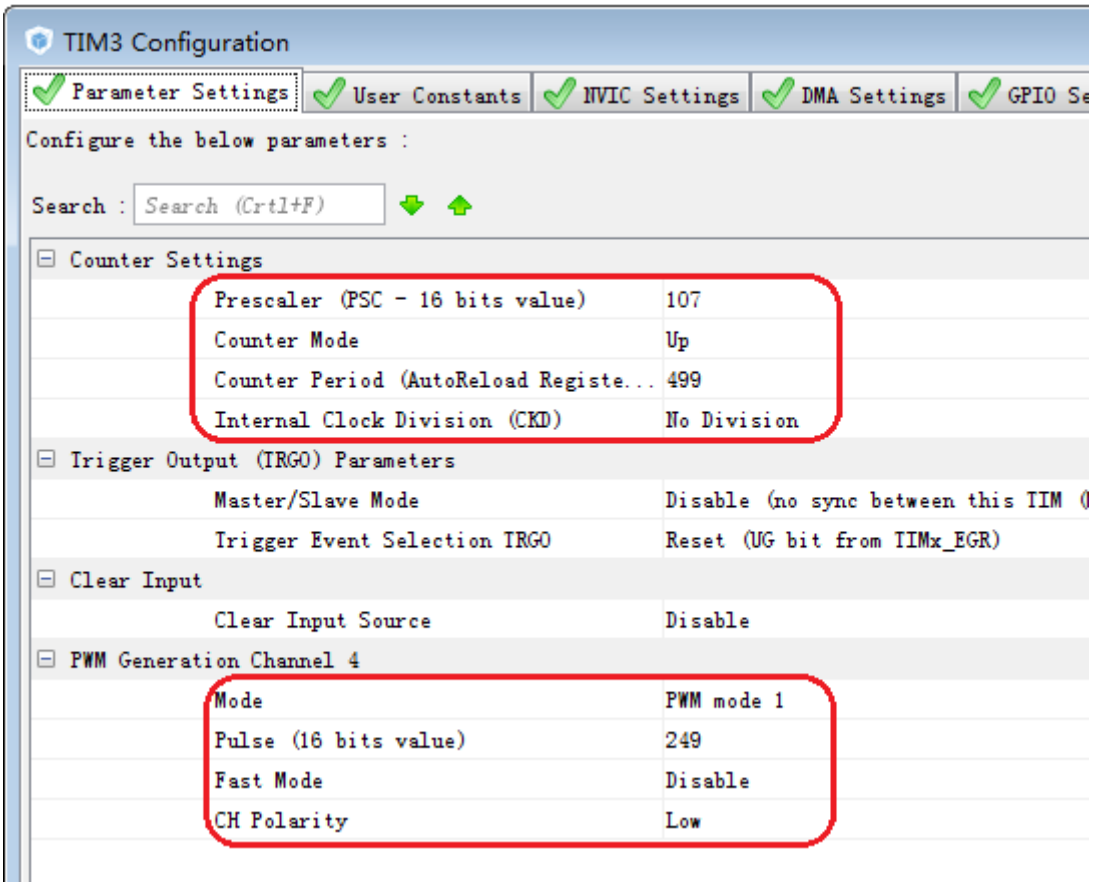


图 12.5.2 TIM3 参数设置界面

本章 PWM 输出实验，我们并没有使用到中断，所以我们不需要使能中断和配置 NVIC。

经过上面的配置就可以生成工程源码，大家可以和本实验工程对比参考学习。

第十四章 输入捕获实验

上一章，我们介绍了 STM32F7 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：

14.1 输入捕获简介
14.2 硬件设计
14.3 软件设计
14.4 下载验证
14.5 STM32CubeMX 配置定时器输入捕获功能

14.1 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 14.1.1 所示：

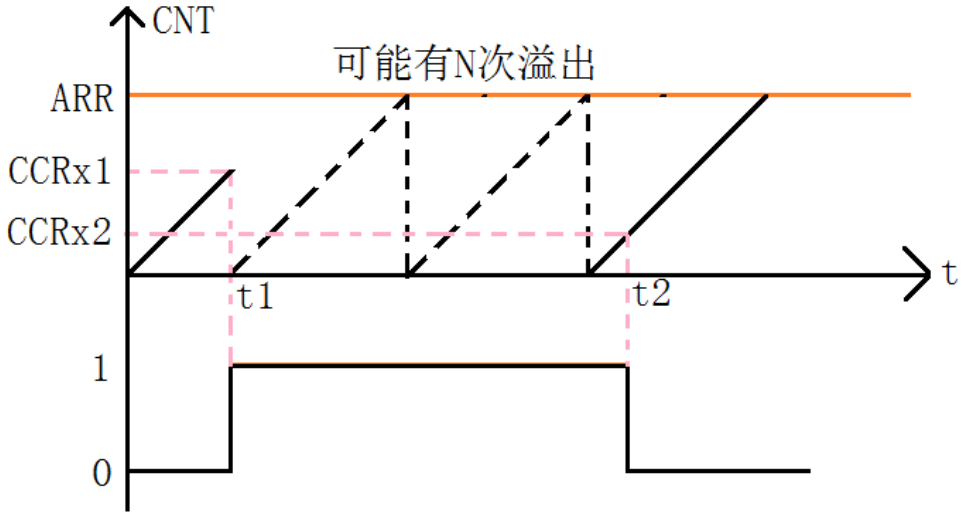


图 14.1.1 输入捕获脉宽测量原理

如图 14.1.1 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 14.1.1 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。

STM32F767 的定时器，除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。STM32F767 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 14.1.1 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 14.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC2M[3]	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC1M[3]
							Res.								Res.
							rw								rw

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

当在输入捕获模式下使用的时候，对应图 14.1.2 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《STM32F7 中文参考手册》705 页，23.4.8 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 14.1.3 所示：

位 7:4 **IC1F**：输入捕获 1 滤波器 (Input capture 1 filter)

此位域可定义 TI1 输入的采样频率和适用于 TI1 的数字滤波器带宽。数字滤波器由事件计数器组成，每 N 个事件才视为一个有效边沿：

0000：无滤波器，按 f_{DTS} 频率进行采样
0001：$f_{SAMPLING}=f_{CK\_INT}$，N=2
0010：$f_{SAMPLING}=f_{CK\_INT}$，N=4
0011：$f_{SAMPLING}=f_{CK\_INT}$，N=8
0100：$f_{SAMPLING}=f_{DTS}/2$，N=6
0101：$f_{SAMPLING}=f_{DTS}/2$，N=8
0110：$f_{SAMPLING}=f_{DTS}/4$，N=6
0111：$f_{SAMPLING}=f_{DTS}/4$，N=8
1000：$f_{SAMPLING}=f_{DTS}/8$，N=6
1001：$f_{SAMPLING}=f_{DTS}/8$，N=8
1010：$f_{SAMPLING}=f_{DTS}/16$，N=5
1011：$f_{SAMPLING}=f_{DTS}/16$，N=6
1100：$f_{SAMPLING}=f_{DTS}/16$，N=8
1101：$f_{SAMPLING}=f_{DTS}/32$，N=5
1110：$f_{SAMPLING}=f_{DTS}/32$，N=6
1111：$f_{SAMPLING}=f_{DTS}/32$，N=8

注意：在当前硅版本中，当 ICxF[3:0]= 1、2 或 3 时，将用 CK_INT 代替公式中的 f_{DTS}。

位 3:2 **IC1PSC**：输入捕获 1 预分频器 (Input capture 1 prescaler)

此位域定义 CC1 输入 (IC1) 的预分频比。
只要 CC1E=0（TIMx_CCER 寄存器），预分频器便立即复位。
00：无预分频器，捕获输入上每检测到一个边沿便执行捕获
01：每发生 2 个事件便执行一次捕获
10：每发生 4 个事件便执行一次捕获
11：每发生 8 个事件便执行一次捕获

位 1:0 **CC1S**：捕获/比较 1 选择 (Capture/Compare 1 selection)

此位域定义通道方向（输入/输出）以及所使用的输入。
00：CC1 通道配置为输出
01：CC1 通道配置为输入，IC1 映射到 TI1 上
10：CC1 通道配置为输入，IC1 映射到 TI2 上
11：CC1 通道配置为输入，IC1 映射到 TRC 上。此模式仅在通过 TS 位（TIMx_SMCR 寄存器）选择内部触发输入时有效

注意：仅当通道关闭时（TIMx_CCER 中的 CC1E = 0），才可向 CC1S 位写入数据。

图 14.1.3 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《STM32F7 中文参考手册》651 页的图 19.1 通用定时器框图），即 CC1 对应 TIMx_CH1。

输入捕获 1 预分频器 IC1PSC[1:0]，这个比较好理解。我们是 1 次边沿就触发 1 次捕获，所以选择 00 就是了。

输入捕获 1 滤波器 IC1F[3:0]，这个用来设置输入采样频率和数字滤波器长度。其中，f_{CK_INT} 是定时器的输入频率（TIMxCLK），一般为 54Mhz/108Mhz（看该定时器在那个总线上），而 f_{DTS} 则是根据 TIMx_CR1 的 CKD[1:0]的设置来确定的，如果 CKD[1:0]设置为 00，那么 $f_{DTS}=f_{CK\_INT}$。N 值就是滤波长度，举个简单的例子：假设 IC1F[3:0]=0011，并设置 IC1 映射到通道 1 上，且为上升沿触发，那么在捕获到上升沿的时候，再以 f_{CK_INT} 的频率，连续采样到 8 次通道 1 的电平，如果都是高电平，则说明却是一个有效的触发，就会触发输入捕获中断（如果开启了的话）。这样可以滤除那些高电平脉宽低于 8 个采样周期的脉冲信号，从而达到滤波的效果。这里，我们不做滤波处理，所以设置 IC1F[3:0]=0000，只要采集到上升沿，就触发捕获。

再来看看捕获/比较使能寄存器：TIMx_CCER，该寄存器的各位描述见图 14.1.3（在第 14 章）。本章我们要用到这个寄存器的最低 2 位，CC1E 和 CC1P 位。这两个位的描述如图 14.1.4 所示：

位 1 **CC1P**：*捕获/比较 1 输出极性 (Capture/Compare 1 output Polarity)*。

CC1 通道配置为输出：
0：OC1 高电平有效
1：OC1 低电平有效
CC1 通道配置为输入：
CC1NP/CC1P 位可针对触发或捕获操作选择 TI1FP1 和 TI2FP1 的极性。
00：非反相/上升沿触发
电路对 TIxFP1 上升沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式或编码器模式下执行触发操作）。
01：反相/下降沿触发
电路对 TIxFP1 下降沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 反相（在门控模式或编码器模式下执行触发操作）。
10：保留，不使用此配置。
11：非反相/上升沿和下降沿均触发
电路对 TIxFP1 上升沿和下降沿都敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式下执行触发操作）。编码器模式下不得使用此配置。

位 0 **CC1E**：*捕获/比较 1 输出使能 (Capture/Compare 1 output enable)*。

CC1 通道配置为输出：
0：关闭——OC1 未激活
1：开启——在相应输出引脚上输出 OC1 信号
CC1 通道配置为输入：
此位决定了是否可以实际将计数器值捕获到输入捕获/比较寄存器 1 (TIMx_CCR1) 中。
0：禁止捕获
1：使能捕获

图 14.1.4 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 和 GPIOA 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_PP;      //复用推挽输出
GPIO_Initure.Pull=GPIO_PULLDOWN;        //下拉
GPIO_Initure.Speed=GPIO_SPEED_HIGH      //高速
GPIO_Initure.Alternate=GPIO_AF2_TIM5;   //PA0 复用为 TIM5 通道 1
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

跟上一讲 PWM 输出类似，这里我们使用的是定时器 5 的通道 1，所以我们从 STM32F7 对应的数据手册可以查看到对应的 IO 口为 PA0:

PA0-WKUP (PA0)	I/O	FT	(5)	TIM2_CH1/TIM2_ETR, TIM5_CH1, TIM8_ETR, USART2_CTS, UART4_TX, ETH_MII_CRS, EVENTOUT	ADC123_IN0/ WKUP(4)

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=89;                      //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 STM32F7 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。

14.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） KEY_UP 按键

3） 串口

4） 定时器 TIM3

5） 定时器 TIM5

前面 4 个，在之前的章节均有介绍。本节，我们将捕获 TIM5_CH1（PA0）上的高电平脉宽，通过 KEY_UP 按键输入高电平，并从串口打印高电平脉宽。同时我们保留上节的 PWM 输出，大家也可以通过用杜邦线连接 PF9 和 PA0，来测量 PWM 输出的高电平脉宽。

14.3 软件设计

相比上一章讲解的 PWM 实验，我们直接在 timer.c 和 timer.h 中直接添加了输入捕获相关程序。对于输入捕获，我们也同样使用的定时器相关操作，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 文件中新增的内容如下：

```
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(TIM2,TIM5 是 32 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
}

//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_PP;      //复用推挽输出
    GPIO_Initure.Pull=GPIO_PULLDOWN;        //下拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;     //高速
    GPIO_Initure.Alternate=GPIO_AF2_TIM5;   //PA0 复用为 TIM5 通道 1
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);     //初始化 PA0

    HAL_NVIC_SetPriority(TIM5_IRQn,2,0);    //设置中断优先级，抢占 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(对于 32 位定时器来说,1us 计数器加 1,溢出时间:4294 秒)
u8   TIM5CH1_CAPTURE_STA=0;   //输入捕获状态
u32  TIM5CH1_CAPTURE_VAL;//输入捕获值(TIM2/TIM5 是 32 位)

//定时器 5 中断服务函数
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（溢出）中断处理回调函数， 在 HAL_TIM_IRQHandler 中会被调用
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)//更新中断发生时执行
{
    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=0XFFFFFFFF;
            }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); //清除设置
            TIM_SET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1,
```

```
                                        TIM_ICPOLARITY_RISING);//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); //清除原来设置
                    TIM_SET_CAPTUREPOLARITY(&TIM5_Handler,TIM_CHANNEL_1,
                                            TIM_ICPOLARITY_FALLING);//下降沿捕获
                    __HAL_TIM_ENABLE(&TIM5_Handler);//使能定时器 5
                }
            }

    }
```

此部分代码包含 5 个函数。函数 TIM5_CH1_Cap_Init 和回调函数 HAL_TIM_IC_MspInit 共同用来实现 14.1 小节讲解的步骤 1~6。TIM5_IRQHandler 是 TIM5 的中断服务函数，该函数内部和定时器中断实验一样，只有一行代码就是直接调用函数 HAL_TIM_IRQHandler。根据我们前面的讲解，函数 HAL_TIM_IRQHandler 内部会对中断来源进行判断（中断标志位），然后分别调用对应的中断处理回调函数，最后还会自动清除相应的中断标志位。函数 HAL_TIM_PeriodElapsedCallback 和 HAL_TIM_IC_CaptureCallback 就是我们要着重讲解的定时器更新中断（溢出）以及输入捕获中断处理回调函数。

同时，在该文件中我们还定义了两个全局变量，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 14.3.1 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 14.3.1 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（执行中断处理回调函数 HAL_TIM_IC_CaptureCallback），此时如果 TIM5CH1_CAPTURE_STA 的第 6 位为 0，则表示还没有捕获到新的上升沿，就先把 TIM5CH1_CAPTURE_STA、TIM5CH1_CAPTURE_VAL 和计数器值 TIM5->CNT 等清零，然后再设置 TIM5CH1_CAPTURE_STA 的第 6 位为 1，标记捕获到高电平，最后设置为下降沿捕获，等待下降沿到来。如果等待下降沿到来期间，定时器发生了溢出（执行溢出中断处理回调函数 HAL_TIM_PeriodElapsedCallback），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;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    TIM3_PWM_Init(500-1,108-1);  //108M/108=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(0XFFFFFFFF,108-1); //以 1MHZ 的频率计数
    while(1)
    {
        delay_ms(10);
        TIM_SetTIM3Compare4(TIM_GetTIM3Capture4()+1);
        if(TIM_GetTIM3Capture4()==300)TIM_SetTIM3Compare4(0);
        if(TIM5CH1_CAPTURE_STA&0X80)       //成功捕获到了一次高电平
        {
            temp=TIM5CH1_CAPTURE_STA&0X3F;
            temp*=0XFFFFFFFF;                   //溢出时间总和
            temp+=TIM5CH1_CAPTURE_VAL;  //得到总的高电平时间
            printf("HIGH:%lld us\r\n",temp);//打印总的高点平时间
            TIM5CH1_CAPTURE_STA=0;          //开启下一次捕获
        }
    }
}
```

该 main 函数是在 PWM 实验的基础上修改来的，我们保留了 PWM 输出，同时通过设置 TIM5_Cap_Init(0XFFFFFFFF,108-1)，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。

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阿波罗 STM32 开发板上，可以看到 DS0 的状态和上一章差不多，由暗→亮的循环。说明程序已经正常在跑了，我们再打开串口调试助手，选择对应的串口，然后按 KEY_UP 按键，可以看到串口打印的高电平持续时间，如图 14.4.1 所示：

图 14.4.1 PWM 控制 DS0 亮度

从上图可以看出，其中有 2 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 PB1，看看上一节中我们设置的 PWM 输出的高电平是如何变化的。

14.5 STM32CubeMX 配置定时器输入捕获功能

使用 STM32CubeMX 配置输入捕获功能初始化代码步骤如下：

① 在 Pinout->TIM5 配置项中，配置 Channel1 的值为 Input Capture direct mode，然后选中 Internal Clock。操作过程如下图 14.5.1 所示：

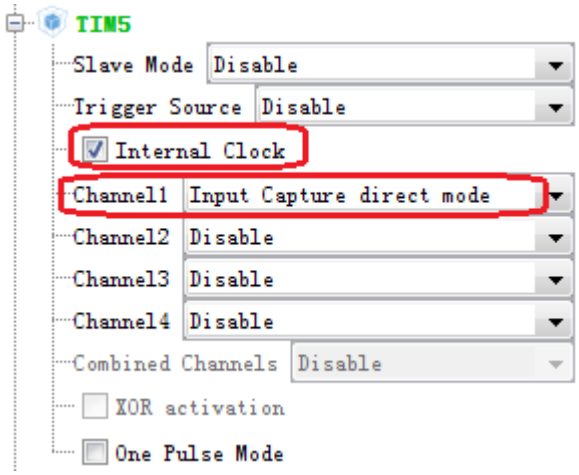
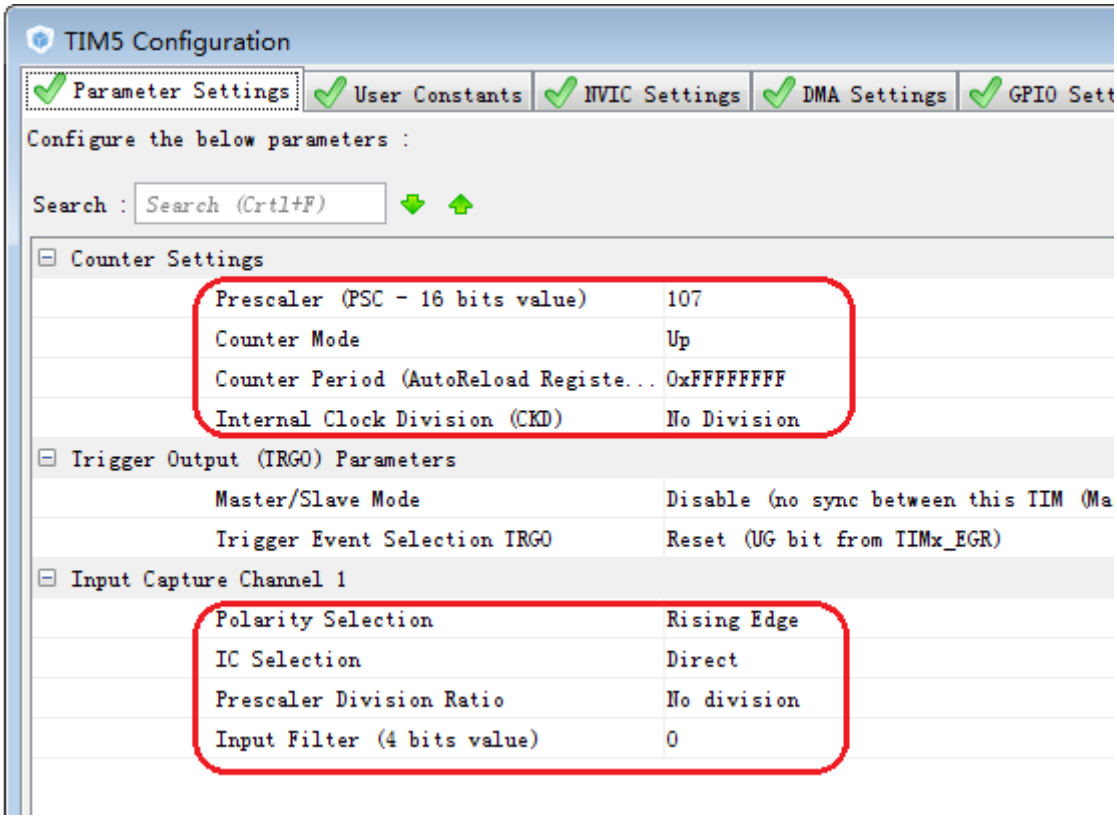


图 14.5.1 TIM3 配置

② 进入 Configuration->TIM5 配置页，在弹出的界面中点击 Parameter Settings 选项卡，

Counter Settings 配置栏下面的四个选项就是用来配置定时器的预分频系数，自动装载值，计数模式以及时钟分频因子。在界面的 Input Capture Channel 1 配置栏配置输入捕获通道 1 的捕获极性，分频系数，映射，滤波器等参数，操作方法如下图 14.5.2 所示：



③ 进入 Configuration->NVIC 配置页，在弹出的界面中点击 NVIC 选项卡，配置 Interrupt Table 中的 TIM5 global interrupt，使能中断，配置抢占优先级和响应优先级。

配置完上面步骤后，生成代码。在生成的代码中，并没有使能相应中断的代码，也没有改写中断处理回调函数，这些都是用户根据自己需要来编写的，这也说明 STM32CubeMX 不是万能的，大家还得认真学习 STM32 基础知识。

第十五章 电容触摸按键实验

上一章，我们介绍了 STM32F7 的输入捕获功能及其使用。这一章，我们将向大家介绍如何通过输入捕获功能，来做一个电容触摸按键。在本章中，我们将用 TIM2 的通道 1（PA5）来做输入捕获，并实现一个简单的电容触摸按键，通过该按键控制 DS1 的亮灭。从本章分为如下几个部分：

15.1 电容触摸按键简介
15.2 硬件设计
15.3 软件设计
15.4 下载验证

15.1 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。

我们将利用阿波罗 STM32 开发板上的触摸按键（TPAD），来实现对 DS1 的亮灭控制。这里 TPAD 其实就是阿波罗 STM32 开发板上的一小块覆铜区域，实现原理如图 15.1.1 所示：

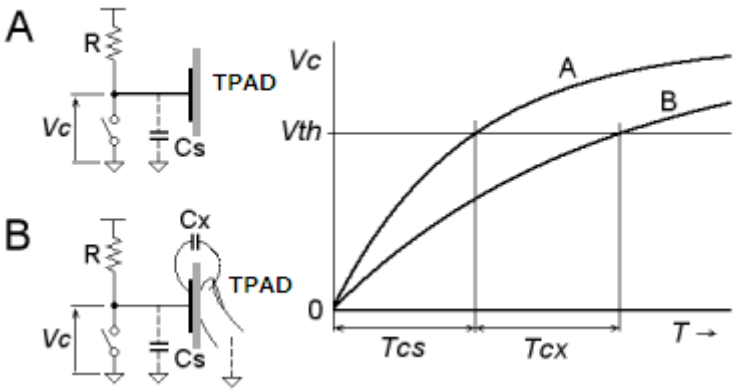


图 15.1.1 电容触摸按键原理

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 STM32F7 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，我们还可以把阿波罗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 PA5(TIM2_CH1)来检测 TPAD 是否有触摸，在每次检测之前，我们先配置 PA5 为推挽输出，将电容 Cs（或 Cs+Cx）放电，然后配置 PA5 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 TIM2_CH1 的输入捕获，检测上升沿，当检测到上升沿的时候，就认为电容充电完成了，完成一次捕获检测。

在 MCU 每次复位重启的时候，我们执行一次捕获检测（可以认为没触摸），记录此时的值，记为 tpad_default_val，作为判断的依据。在后续的捕获检测，我们就通过与 tpad_default_val 的对比，来判断是不是有触摸发生。

关于输入捕获的配置，在上一章我们已经有详细介绍了，这里我们就不再介绍。至此，电容触摸按键的原理介绍完毕。

15.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0 和 DS1

2） 定时器 TIM2

3） 触摸按键 TPAD

前面两个之前均有介绍，我们需要通过 TIM2_CH1（PA5）采集 TPAD 的信号，所以本实验需要用跳线帽短接多功能端口（P11）的 TPAD 和 ADC，以实现 TPAD 连接到 PA5。如图 15.2.1 所示：

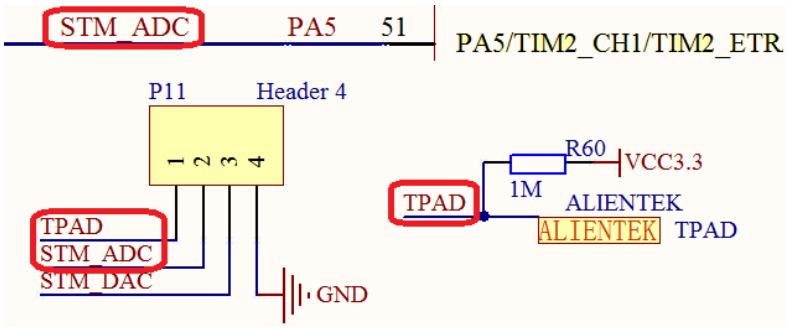


图 15.2.1 TPAD 与 STM32F7 连接原理图

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。

15.3 软件设计

打开实验工程可以看到，我们在上一章实验的基础上删掉了 timer.c 文件，同时新建了 tpad.c 和 tpad.h 文件。因为 tpad 我们也是使用的定时器输入捕获来实现，所以我们相比上个实验并没有增加任何库函数相关的文件。

接下来我们看看 tpad.c 文件代码：

```
TIM_HandleTypeDef TIM2_Handler;          //定时器 2 句柄

#define TPAD_ARR_MAX_VAL  0XFFFFFFFF      //最大的 ARR 值(TIM2 是 32 位定时器)

vu16 tpad_default_val=0;                  //空载的时候(没有手按下),计数器需要的时间
```

```
//初始化触摸按键 ：  获得空载的时候触摸按键的取值.
//psc:分频系数,越小,灵敏度越高.
//返回值:0,初始化成功;1,初始化失败
u8 TPAD_Init(u8 psc)
{
    u16 buf[10];
    u16 temp;
    u8 j,i;
    TIM2_CH1_Cap_Init(TPAD_ARR_MAX_VAL,psc-1);//设置分频系数
    for(i=0;i<10;i++)//连续读取 10 次
    {
        buf[i]=TPAD_Get_Val();
        delay_ms(10);
    }
    for(i=0;i<9;i++)//排序
    {
        for(j=i+1;j<10;j++)
        {
            if(buf[i]>buf[j])//升序排列
            {
                temp=buf[i];  buf[i]=buf[j];  buf[j]=temp;
            }
        }
    }
    temp=0;
    for(i=2;i<8;i++)temp+=buf[i];//取中间的 8 个数据进行平均
    tpad_default_val=temp/6;
    printf("tpad_default_val:%d\r\n",tpad_default_val);
    if(tpad_default_val>TPAD_ARR_MAX_VAL/2)return 1;
                        //初始化遇到超过 TPAD_ARR_MAX_VAL/2 的数值,不正常!
    return 0;
}
//复位一次
//释放电容电量，并清除定时器的计数值
void TPAD_Reset(void)
{
    GPIO_InitTypeDef GPIO_Initure;

    GPIO_Initure.Pin=GPIO_PIN_5;                //PA5
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;      //推挽输出
    GPIO_Initure.Pull=GPIO_PULLDOWN;            //下拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

```
    HAL_GPIO_WritePin(GPIOA,GPIO_PIN_5,GPIO_PIN_RESET);  //PA5 输出 0，放电
    delay_ms(5);
    __HAL_TIM_CLEAR_IT(&TIM2_Handler,TIM_IT_CC1|TIM_IT_UPDATE);
                                                    //清除中断标志位
    __HAL_TIM_SET_COUNTER(&TIM2_Handler,0); //计数器值归 0

    GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //推挽复用
    GPIO_Initure.Pull=GPIO_NOPULL;              //不带上下拉
    GPIO_Initure.Alternate=GPIO_AF1_TIM2;       //PA5 复用为 TIM2 通道 1
    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(&TIM2_Handler,TIM_FLAG_CC1)==RESET)
                                                    //等待捕获上升沿
    {
        if(__HAL_TIM_GET_COUNTER(&TIM2_Handler)>TPAD_ARR_MAX_VAL-500)
        return __HAL_TIM_GET_COUNTER(&TIM2_Handler);//超时直接返回 CNT 的值
    };
    return HAL_TIM_ReadCapturedValue(&TIM2_Handler,TIM_CHANNEL_1);
}

//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
//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 2 通道 1 输入捕获配置
//arr：自动重装值(TIM2 是 32 位的!!)          psc：时钟预分频数
void TIM2_CH1_Cap_Init(u32 arr,u16 psc)
{
    TIM_IC_InitTypeDef TIM2_CH1Config;

    TIM2_Handler.Instance=TIM2;    //通用定时器 3
    TIM2_Handler.Init.Prescaler=psc; //分频
    TIM2_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;//向上计数器
    TIM2_Handler.Init.Period=arr;     //自动装载值
    TIM2_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_IC_Init(&TIM2_Handler);

    TIM2_CH1Config.ICPolarity=TIM_ICPOLARITY_RISING;    //上升沿捕获
```

```
    TIM2_CH1Config.ICSelection=TIM_ICSELECTION_DIRECTTI;  //映射到 TI1 上
    TIM2_CH1Config.ICPrescaler=TIM_ICPSC_DIV1;        //配置输入分频，不分频
    TIM2_CH1Config.ICFilter=0;                        //配置输入滤波器，不滤波
    HAL_TIM_IC_ConfigChannel(&TIM2_Handler,&TIM2_CH1Config,
                                          TIM_CHANNEL_1);//配置 TIM2 通道 1
    HAL_TIM_IC_Start(&TIM2_Handler,TIM_CHANNEL_1);  //开始捕获 TIM2 的通道 1
}

//定时器 2 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 2 句柄
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_TIM2_CLK_ENABLE();            //使能 TIM2 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_5;            //PA5
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //推挽复用
    GPIO_Initure.Pull=GPIO_NOPULL;          //不带上下拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;     //高速
    GPIO_Initure.Alternate=GPIO_AF1_TIM2;   //PA5 复用为 TIM2 通道 1
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
}
```

函数 TIM2_CH1_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(&TIM2_Handler,0)将计数值 TIM2_CNT 设置为 0，然后死循环等待发生上升沿捕获（或计数溢出），将捕获到的值（或溢出值）作为返回值返回。

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 TIM2_CH1_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 PA5 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 PA5 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 TIM2_CH1 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。

接下来我们看看主函数代码如下：

```
int main(void)
{
    u8 t=0;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    TPAD_Init(8);                   //初始化触摸按键,以 108/8=13.5Mhz 频率计数
    while(1)
    {
        if(TPAD_Scan(0))        //成功捕获到了一次上升沿(此函数执行时间至少 15ms)
        {
            LED1_Toggle;        //LED1 取反
        }
        t++;
        if(t==15)
        {
            t=0;
            LED0_Toggle;        //LED1 翻转
        }
        delay_ms(10);
    }
}
```

该 main 函数比较简单，TPAD_Init(8)函数执行之后，就开始触摸按键的扫描，当有触摸的时候，对 DS1 取反，而 DS0 则有规律的间隔取反，提示程序正在运行。

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。

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阿波罗 STM32 开发板上，可以看到 DS0 慢速闪烁，此时，我们用手指触摸 ALIENTEK 阿波罗 STM32 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 15.4.1 所示：

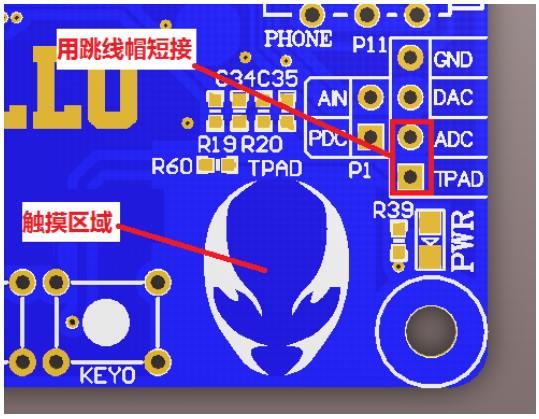


图 15.4.1 触摸区域和跳线帽短接方式示意图

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 160 左右。

第十六章 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，我们将使用阿波罗 STM32 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：

16.1 OLED 简介
16.2 硬件设计
16.3 软件设计
16.4 下载验证

16.1 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 16.1.1 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 16.1.1 OLED 模块接口方式设置表

表 16.1.1 中：“1”代表接 VCC，而“0”代表接 GND。

该模块的外观图如图 16.1.1 所示：

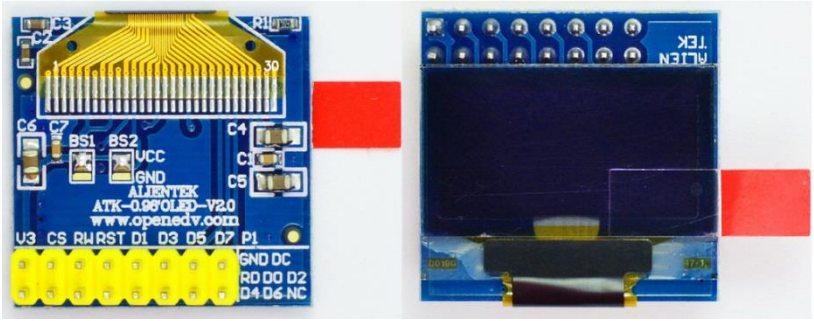


图 16.1.1 ALIENTEK OLED 模块外观图

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 16.1.2 所示：

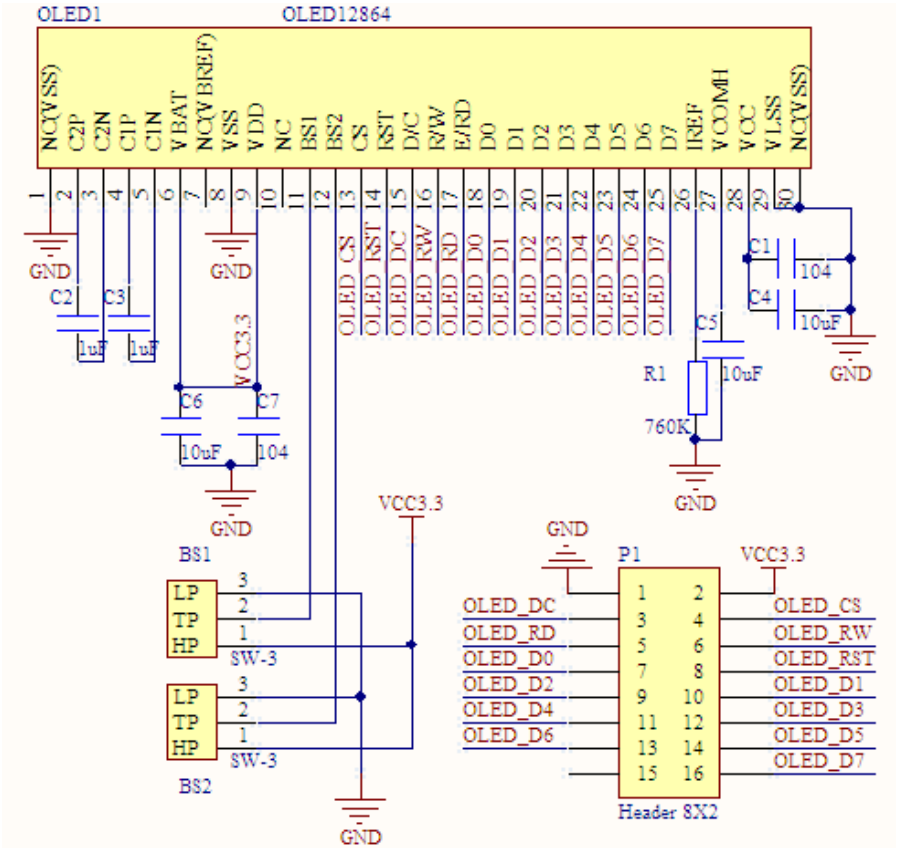


图 16.1.2 ALIENTEK OLED 模块原理图

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 STM32F767 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 16.1.3 所示：

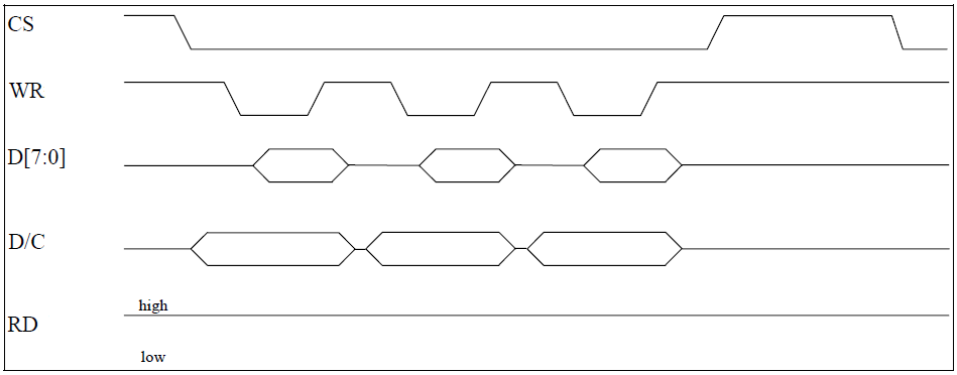


图 16.1.3 8080 并口写时序图

SSD1306 的 8080 并口读时序图如图 16.1.4 所示：

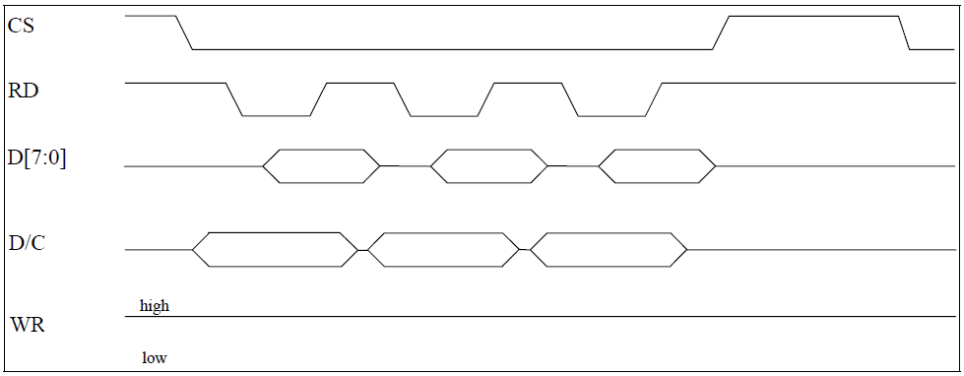


图 16.1.4 8080 并口读时序图

SSD1306 的 8080 接口方式下，控制脚的信号状态所对应的功能如表 16.1.2：

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 16.1.2 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 16.1.5 所示：

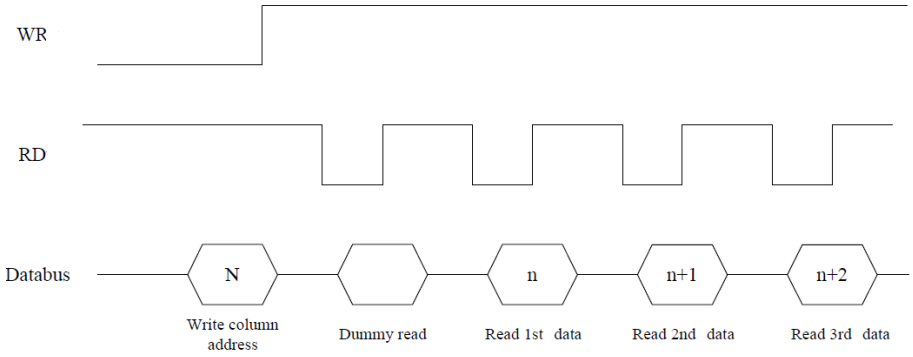


图 16.1.5 读显存时序图

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 16.1.6 所示：

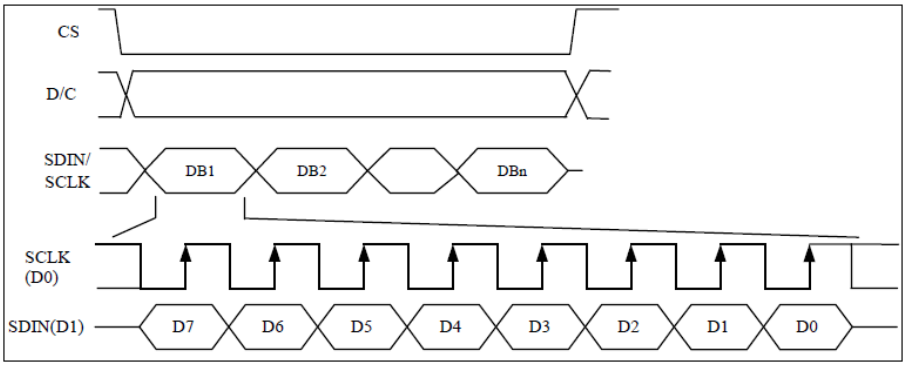


图 16.1.6 4 线 SPI 写操作时序图

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 16.1.3 所示：

	行（COL0~127）						
列（COM0~63）	SEG0	SEG1	SEG2	……	SEG125	SEG126	SEG127
	PAGE0						
	PAGE1						
	PAGE2						
	PAGE3						
	PAGE4						
	PAGE5						
	PAGE6						
	PAGE7						

表 16.1.3　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 STM32F767 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 STM32F767 上的 GRAM（实际上就是 SRAM），在修改完了之后，一次性把 STM32F767 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 16.1.4 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 16.1.4　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 16.1.7 所示：

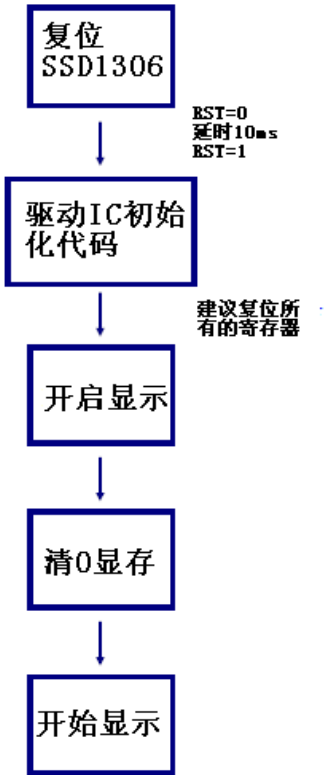


图 16.1.7　SSD1306 初始化框图

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 STM32F767 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。

16.2 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 模块与阿波罗 STM32F767 开发板的连接，开发板底板的 OLED/CAMERA 接口（P7 接口）可以和 ALIENTEK OLED 模块直接对插（靠左插！），连接如图 16.2.1 所示：

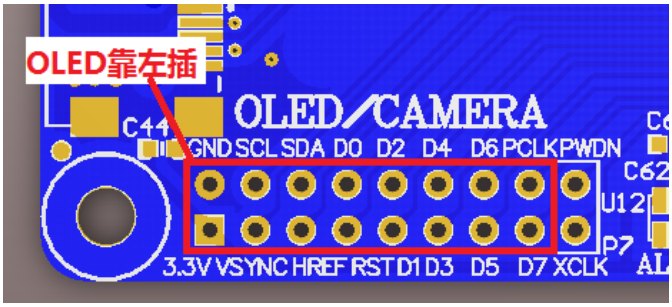


图 16.2.1 OLED 模块与开发板连接示意图

图中圈出来的部分就是连接 OLED 的接口，这里在硬件上，OLED 与阿波罗 STM32F767 开发板的 IO 口对应关系如下：

OLED_CS 对应 DCMI_VSYNC，即：PB7;

OLED_RS 对应 DCMI_SCL，即：PB4;

OLED_WR 对应 DCMI_HREF，即：PH8;

OLED_RD 对应 DCMI_SDA，即：PB3;

OLED_RST 对应 DCMI_RESET，即：PA15;

OLED_D[7:0]对应 DCMI_D[7:0]，即：PB9/PB8/PD3/PC11/PC9/PC8/PC7/PC6;

这些线的连接，开发板的内部已经连接好了，我们只需要将 OLED 模块插上去就好了，注意，这里的 OLED_D[7:0]因为不是接的连续的 IO，所以得用拼凑的方式去组合一下，后续会介绍。实物连接如图 16.2.2 所示：

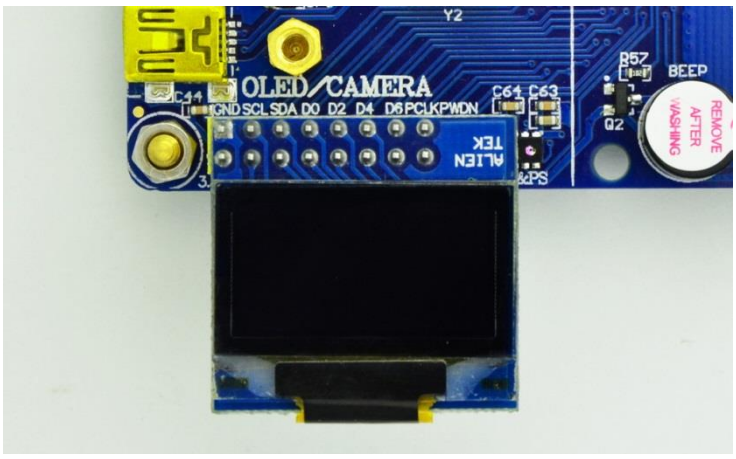


图 16.2.2 OLED 模块与开发板连接实物图

16.3 软件设计

本实验，我们新建了 oled.c 和 oled.h 文件。这两个文件用来存放 OLED 相关的驱动函数以

及申明等。

oled.c 的代码，由于比较长，这里我们就不贴出来了，仅介绍几个比较重要的函数。首先是 OLED_Init 函数，该函数的结构比较简单，开始是对 IO 口的初始化，这里我们用了宏定义 OLED_MODE 来决定要设置的 IO 口，其他就是一些初始化序列了，我们按照厂家提供的资料来做就可以。最后要说明一点的是，因为 OLED 是无背光的，在初始化之后，我们把显存都清空了，所以我们在屏幕上是看不到任何内容的，跟没通电一个样，不要以为这就是初始化失败，要写入数据模块才会显示的。OLED_Init 函数代码如下：

```
//初始化 SSD1306
void OLED_Init(void)
{
    GPIO_InitTypeDef    GPIO_Initure;

    __HAL_RCC_GPIOA_CLK_ENABLE();     //使能 GPIOA 时钟
    __HAL_RCC_GPIOB_CLK_ENABLE();     //使能 GPIOB 时钟
    __HAL_RCC_GPIOC_CLK_ENABLE();     //使能 GPIOC 时钟
    __HAL_RCC_GPIOD_CLK_ENABLE();     //使能 GPIOD 时钟
    __HAL_RCC_GPIOF_CLK_ENABLE();     //使能 GPIOF 时钟
    __HAL_RCC_GPIOH_CLK_ENABLE();     //使能 GPIOH 时钟
#if OLED_MODE==1          //使用 8080 并口模式

    //GPIO 初始化设置
    GPIO_Initure.Pin=GPIO_PIN_15;                   //PA15
    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(GPIOA,&GPIO_Initure);             //初始化

    //PB3,4,7,8,9
    GPIO_Initure.Pin=GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9;
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);//初始化

    //PC6,7,8,9,11
    GPIO_Initure.Pin=GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_11;

    HAL_GPIO_Init(GPIOC,&GPIO_Initure);//初始化

    //PD3
    GPIO_Initure.Pin=GPIO_PIN_3;
    HAL_GPIO_Init(GPIOD,&GPIO_Initure);//初始化

    //PH8
    GPIO_Initure.Pin=GPIO_PIN_8;
    HAL_GPIO_Init(GPIOH,&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_15;                   //PA15
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FAST;             //高速
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);             //初始化

    //PB4,7
    GPIO_Initure.Pin=GPIO_PIN_4|GPIO_PIN_7;
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);//初始化

    //PC6,7
    GPIO_Initure.Pin=GPIO_PIN_6|GPIO_PIN_7;
    HAL_GPIO_Init(GPIOC,&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
    …//此处省略部分代码，详情请参考实验工程

    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 STM32F7 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 STM32F7 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 STM32F7 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
//通过拼凑的方法向 OLED 输出一个 8 位数据
//data:要输出的数据
void OLED_Data_Out(u8 data)
{
    u16 dat=data&0X0F;
    GPIOC->ODR&=~(0XF<<6);          //清空 6~9
    GPIOC->ODR|=dat<<6;             //D[3:0]-->PC[9:6]

    GPIOC->ODR&=~(0X1<<11);         //清空 11
    GPIOC->ODR|=((data>>4)&0x01)<<11;

    GPIOD->ODR&=~(0X1<<3);          //清空 3
    GPIOD->ODR|=((data>>5)&0x01)<<3;

    GPIOB->ODR&=~(0X3<<8);          //清空 8,9
    GPIOB->ODR|=((data>>6)&0x01)<<8;
    GPIOB->ODR|=((data>>7)&0x01)<<9;
}
//向 SSD1306 写入一个字节。
//dat:要写入的数据/命令
//cmd:数据/命令标志 0,表示命令;1,表示数据;
void OLED_WR_Byte(u8 dat,u8 cmd)
{
    OLED_Data_Out(dat);
    OLED_RS(cmd);
    OLED_CS(0);
    OLED_WR(0);
    OLED_WR(1);
```

```
        OLED_CS(1);
        OLED_RS(1);
    }
    #else
    //向 SSD1306 写入一个字节。
    //dat:要写入的数据/命令
    //cmd:数据/命令标志 0,表示命令;1,表示数据;
    void OLED_WR_Byte(u8 dat,u8 cmd)
    {
        u8 i;
        OLED_RS(cmd); //写命令
        OLED_CS(0);
        for(i=0;i<8;i++)
        {
            OLED_SCLK(0);
            if(dat&0x80)OLED_SDIN(1);
            else OLED_SDIN(0);
            OLED_SCLK(1);
            dat<<=1;
        }
        OLED_CS(1);
        OLED_RS(1);
    }
```

#endif 首先，我们看 OLED_Data_Out 函数，这就是我们前面说的，因为 OLED 的 D0~D7 不是接的连续 IO，所以必须将数据，拆分到各个 IO，以实现一次完整的数据传输，该函数就是根据我们 OLED_D[7:0]具体连接的 IO，对数据进行拆分，然后输出给对应位的各个 IO，实现并口数据输出。这种方式会降低并口速度，但是我们 OLED 模块，是单色的，数据量不是很大，所以这种方式也不会造成视觉上的影响，大家可以放心使用，但是如果是 TFTLCD，就不推荐了。

然然后，看 OLED_WR_Byte 函数，这里有 2 个一样的函数，通过宏定义 OLED_MODE 来决定使用哪一个。如果 OLED_MODE=1，就定义为并口模式，选择第一个函数，而如果为 0，则为 4 线串口模式，选择第二个函数。这两个函数输入参数均为 2 个：dat 和 cmd，dat 为要写入的数据，cmd 则表明该数据是命令还是数据。这两个函数的时序操作就是根据上面我们对 8080 接口以及 4 线 SPI 接口的时序来编写的。

OLED_GRAM[128][8]中的 128 代表列数（x 坐标），而 8 代表的是页，每页又包含 8 行，总共 64 行（y 坐标）。从高到低对应行数从小到大。比如，我们要在 x=100，y=29 这个点写入 1，则可以用这个句子实现：

```
OLED_GRAM[100][4]|=1<<2;
```

一个通用的在点（x，y）置 1 表达式为：

```
OLED_GRAM[x][7-y/8]|=1<<(7-y%8);
```

其中 x 的范围为：0~127；y 的范围为：0~63。

因此，我们可以得出下一个将要介绍的函数：画点函数，void OLED_DrawPoint(u8 x，u8 y，u8 t)；函数代码如下：

```
void OLED_DrawPoint(u8 x,u8 y,u8 t)
{
    u8 pos,bx,temp=0;
    if(x>127||y>63)return;//超出范围了.
    pos=7-y/8;
    bx=y%8;
    temp=1<<(7-bx);
    if(t)OLED_GRAM[x][pos]|=temp;
    else OLED_GRAM[x][pos]&=~temp;
}
```

该函数有 3 个参数，前两个是坐标，第三个 t 为要写入 1 还是 0。该函数实现了我们在 OLED 模块上任意位置画点的功能。

接下来，我们介绍一下显示字符函数，OLED_ShowChar，在介绍之前，我们来介绍一下字符（ASCII 字符集）是怎么显示在 OLED 模块上去的。要显示字符，我们先要有字符的点阵数据，ASCII 常用的字符集总共有 95 个，从空格符开始，分别为： !"#$%&'()*+,-0123456789:;<=>?@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 16.3.1 所示：

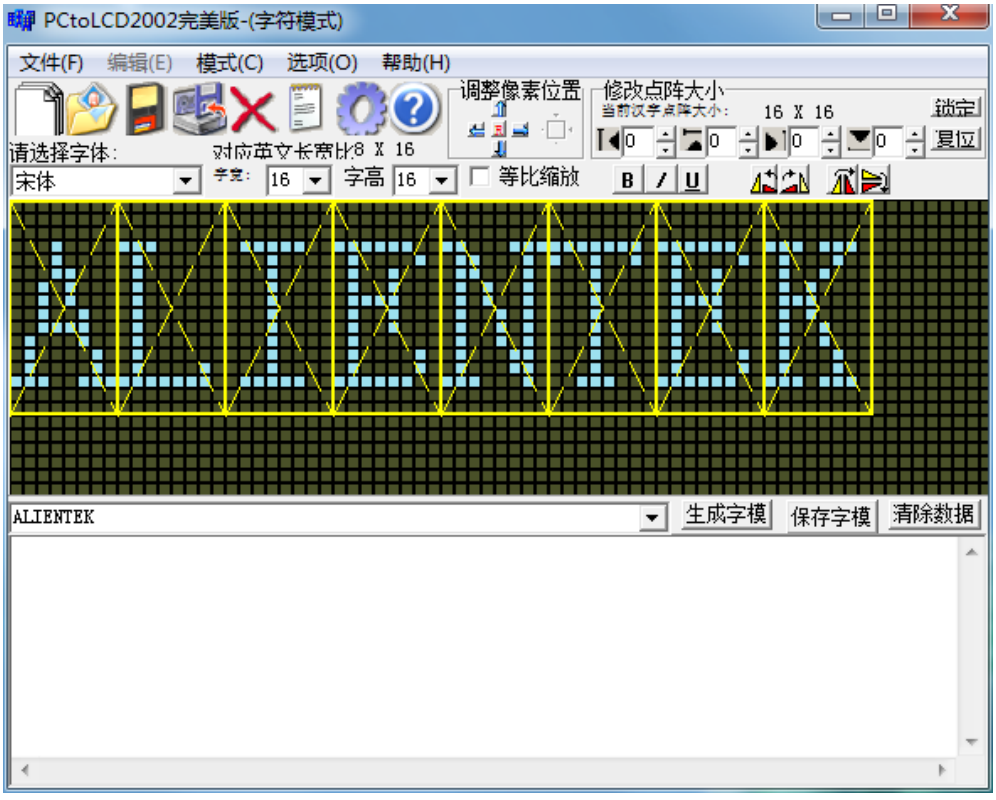


图 16.3.1 PCtoLCD2002 软件界面

然后我们选择设置，在设置里面设置取模方式如图 16.3.2 所示：

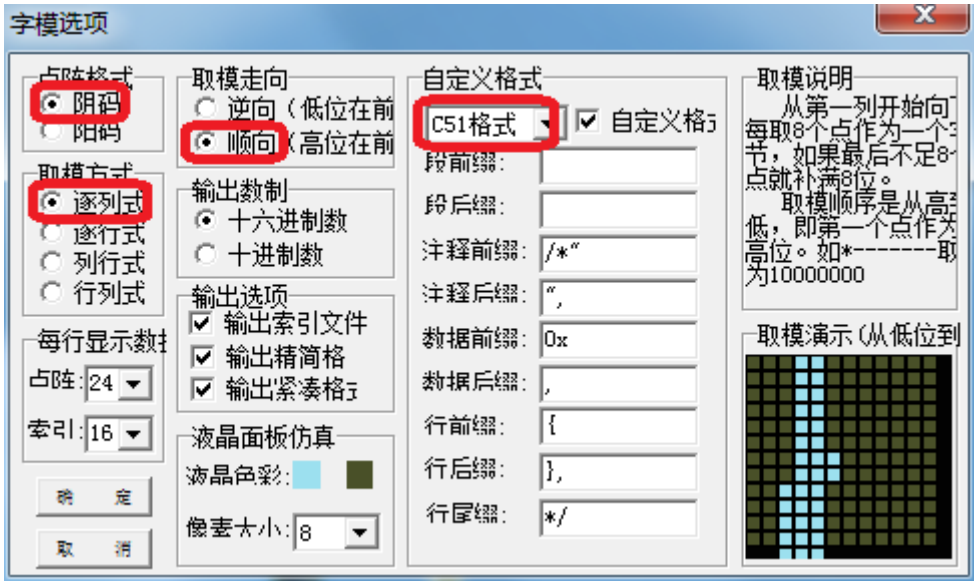


图 16.3.2 设置取模方式

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 16.3.3 所示的这种方式：

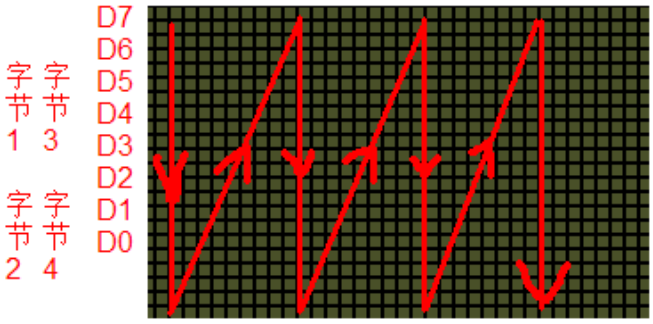


图 16.3.3 取模方式图解

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。oled.h 头文件内容比较简单，主要是一些宏定义和函数声明， 这里就不做过多讲解。

最后我们来看看主函数代码：

```
int main(void)
{
    u8 t=0;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    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 2016/7/11",12);
        OLED_ShowString(0,52,"ASCII:",12);
        OLED_ShowString(64,52,"CODE:",12);
        OLED_Refresh_Gram();//更新显示到 OLED
        t=' ';
        while(1)
        {
            OLED_ShowChar(36,52,t,12,1);   //显示 ASCII 字符
            OLED_ShowNum(94,52,t,3,12);   //显示 ASCII 字符的码值
            OLED_Refresh_Gram();           //更新显示到 OLED
            t++;
            if(t>'~')t=' ';
            delay_ms(500);
            LED0_Toggle;
        }
    }
```

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。

16.4 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 16.4.1 所示：

图 16.4.1 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。

第十七章 内存保护（MPU）实验

STM32 的 Cortex M4（STM32F3/F4 系列）和 Cortex M7（STM32F7 系列）系列的产品，都带有内存保护单元（memory protection unit），简称：MPU。使用 MPU 可以设置不同存储区域的存储器访问特性（如只支持特权访问或全访问）和存储器属性（如可缓存、可共享），从而提高嵌入式系统的健壮性，使系统更加安全。接下来，我们将以 STM32F767 为例，给大家介绍 STM32F7 内存保护单元（MPU）的使用。

本章分为如下几个小节：

17.1 MPU 简介
17.2 硬件设计
17.3 软件设计
17.4 下载验证

17.1 MPU 简介

MPU，即内存保护单元，可以设置不同存储区域的存储器访问特性（如只支持特权访问或全访问）和存储器属性（如可缓存、可缓冲、可共享），对存储器（主要是内存和外设）提供保护，从而提高系统可靠性：

1，阻止用户应用程序破坏操作系统使用的数据。
2，阻止一个任务访问其他任务的数据区，从而隔离任务。
3，可以把关键数据区域设置为只读，从根本上解决被破坏的可能。
4，检测意外的存储访问，如堆栈溢出，数组越界等。
5，将 SRAM 或者 RAM 空间定义为不可执行（用不执行，XN），防止代码注入攻击。

注意，MPU 不仅可以保护内存区域（SRAM 区），还可以保护外设区（比如 FMC）。我们可以通过 MPU 设置存储器的访问权限，当存储器访问和 MPU 定义的访问权限相冲突的时候，则访问会被阻止，并且触发一次错误异常（一般是 MemManage 异常）。然后，在异常处理的时候，就可以确定系统是否应该复位或者执行其他操作。

STM32F7 的 MPU 提供多达 8 个可编程保护区域（region），每个区域（region）都有自己的可编程起始地址、大小及设置。MPU 功能必须开启才会有效，默认条件下，MPU 是关闭的，所以，我们要向使用 MPU，必须先打开 MPU 才行。

8 个可编程保护区域(region)，一般来说是足够使用的了，如果觉得不够，每个区域(region)还可以被进一步划分为更小的子区域（sub region），另外，还允许启用一个背景区域（即没有 MPU 设置的其他所有地址空间），背景区域只允许特权访问。在启用 MPU 后，就不得再访问定义之外的地址区间，也不得访问未经授权的区域（region），否则，将以“访问违例”处理，触发 MemManage 异常。

此外，MPU 定义的区域（region）还可以相互交迭。如果某块内存落在多个区域（region）中，则访问属性和权限将由编号最大的 region 来决定。比如，若 2 号 region 与 5 号 region 交迭，则交迭的部分受 5 号 region 控制。

MPU 设置是由 CTRL、RNR、RBAR 和 RASR 等寄存器控制的，接下来，我们分别介绍一下这几个寄存器。

首先是 MPU 控制寄存器（CTRL），该寄存器只有最低三位有效，其描述如表 17.1.1 所示：

位段	名称	类型	复位值	描述
2	PRIVDEFENA	RW	0	是否为特权级打开缺省存储器映射（即背景 region）。 1=特权级下打开背景 region 0=不打开背景 region。任何访问违例以及对 region 外地址区的访问都将引起 fault
1	HFNMIENA	RW	0	1=在 NMI 和硬 fault 服务例程中不强制除能 MPU 0=在 NMI 和硬 fault 服务例程中强制除能 MPU
0	ENABLE	RW	0	使能 MPU

表 17.1.1 MPU_CTRL 寄存器各位描述

PRIVDEFENA 位用于设置是否开启背景区域（region），通过设置该位为 1，可以在没有建立任何 region 就使能 MPU 的情况下，依然允许特权级程序访问所有地址，而只有用户级程序被卡死。但是，如果设置了其它的 region（最多 8 个 region）并使能 MPU，则背景 region 与这些 region 重合的部分，就要受各 region 的限制。HFNMIENA 位用于控制是否在 NMI 和硬件 fault 中断服务例程中禁止 MPU，一般设置为 0 即可。ENABLE 位，则用于控制是否使能 MPU，我们一般在 MPU 配置完）以后，才对其进行使能，从而开启 MPU。

接下来，介绍 MPU 区域编号寄存器（RNR），该寄存器只有低 8 位有效，其描述如表 17.1.2 所示：

位段	名称	类型	复位值	描述
7:0	REGION	RW	-	选择下一个要配置的 region。因为只支持 8 个 region，所以事实上只有[2:0]有意义

表 17.1.2 MPU_RNR 寄存器各位描述

在配置任何一个区域（region）之前，必须先在 MPU 内选中这个区域，我们可以通过将区域编号写入 MPU_RNR 寄存器来完成这个操作。该寄存器只有低 8 位有效，不过由于 STM32F7 最多只支持 8 个区域，所以，实际上只有最低 3 位有效（0~7）。在配置完区域编号以后，我们就可以对区域属性进行设置了。

接下来，介绍 MPU 基地址寄存器（RBAR），该寄存器各位描述如表 17.1.3 所示：

位段	名称	类型	复位值	描述
31:N	ADDR	RW	-	Region 基址字段。N 取决于 region 容量，以使**基址在数值上能被容量整除**。在 MPU region 属性及容量寄存器中有个 SZENABLE 位段，它决定 ADDR 中有多少个位被采用。
4	VALID	RW	-	决定是否理会写入REGION字段的值 1=MPU region号寄存器被REGION覆盖 0=MPU region号寄存器的值保持不变
3:0	REGION	RW	-	MPU region 覆写位段

表 17.1.3 MPU_RBAR 寄存器各位描述

注意，表中 ADDR 字段所设置的基址必须对齐到区域（region）容量的边界。例如，我们定义某个 region 的容量是 64KB（通过 RASR 寄存器设置），那么它的基址（ADDR）就必须能被 64KB 整除，比如 0X0001 0000、0X0002 0000、0X0003 0000 等（低 16 位全为 0）。

VALID 用于控制 REGION 段（bit[3:0]）的数据是否有效，如果 VALID=1，则 REGION 段

的区域编号将覆盖 MPU_RNR 寄存器所设置的区域编号，否则将使用 MPU_RNR 所设置的区域编号。我们一般设置 VALID 为 0，这样 MPU_RBAR 寄存器的低 5 位就没有用到。

特别注意，表 17.1.3 中的 N 值最少也是 5，所以，基址必须是 32 的倍数，从而可以知道我们设置 region 的容量，必须是 32 字节的倍数。

最后，我们介绍 MPU 区域属性和容量寄存器（RASR），该寄存器各位描述如表 17.1.4 所示：

位段	长度	名称	功能
31:29	3	-	保留
28	1	XN	1=此区禁止取指 0=此区允许取指
27	1	-	保留
26:24	3	AP	访问许可
23:22	2	—	保留
21:19	3	TEX	类型扩展
18	1	S	Sharable（可否共享） 1=可共享 0=不可共享
17	1	C	Cacheable（可否缓存） 1=可缓存 0=不可缓存
16	1	B	Buffable（可否缓冲） 1=可缓冲 0=不可缓冲
15:8	8	SRD	子region除能位段。每设置SRD的一个位，就会除能与之对应的一个子region。容量大于128字节的region都被划分成8个容量相同的子region。容量小于等于128字节的region不能再分。更多信息，请参见对子Region的论述。
7:6	2	-	保留
5:1	5	REGIONSIZE	Region容量，单位是字节。容量为1<<(REGIONSIZE+1)，但是最小容量为32字节
0	1	SZENABLE	1=使能此region　　0=除能此region

表 17.1.4 MPU_RASR 寄存器各位描述

XN 位，用于控制是否允许从此区域取指，如果 XN=1，说明禁止从区域取指，如果强行取指，将产生一个 MemManage 异常。如果设置 XN=0，则允许取指。

AP 位，由 3 个位（bit[26:24]）组成，用于控制数据的访问权限（访问许可），控制关系如表 17.1.5 所示：

值	特权级下的许可	用户级下的许可	典型用法
0b000	禁止访问	禁止访问	禁止访问
0b001	RW	禁止访问	只支持特权访问
0b010	RW	RO	禁止用户程序执行写操作
0b011	RW	RW	全访问
0b100	n/a	n/a	n/a
0b101	RO	禁止访问	仅支持特权读
0b110	RO	RO	只读
0b111	RO	RO	只读

表 17.1.5 不同 AP 设置及其访问权限

TEX、S、C 和 B 等位，对应着存储系统中比较高级的概念，可以通过对这些位段的编程，来支持多样的内存管理模型，这些位组合的详细功能如表 17.1.6 所示：

TEX	C	B	描述	存储器类型	可否共享
000	0	0	强序（严格按照顺序执行）	强序	可以
000	0	1	共享的设备（可以写缓冲）	设备	可以
000	1	0	片外或片内的"写通"型内存，非写分配	普通	S位决定
000	1	1	片外或片内的"写回"型内存，非写分配	普通	S位决定
001	0	0	片外或片内的"不可缓存"型内存	普通	S位决定
001	0	1	n/a	n/a	n/a
001	1	0	由具体实现定义	n/a	n/a
001	1	1	片外或片内的"写回"型，带读和写的分配	普通	S位决定
010	1	x	共享不可的设备	设备	总是不可
010	0	1	n/a	n/a	n/a
010	1	x	n/a	n/a	n/a
1BB	A	A	带缓存的内存。BB=适用于片外内存，AA=适用于片内内存	普通	S位决定

表 17.1.6 TEX、S、C 和 B 对存储器类型的定义

有些情况下，内部和外部内存可能需要不同的缓存策略，次数需要设置 TEX 的第二位为 1，这样 TEX[1:0]的定义就会变为外部策略表（表 17.1.6 种表示为 BB），而 C 和 B 位则会变为内部策略表（表 17.1.6 种表示为 AA）。缓存策略的定义（AA 和 BB）如表 17.1.7 所示：

存储器属性编码 (AA and BB)	高速缓存策略
00	不可共享
01	写回，读写均有分配
10	写通，无写分配
11	写回，无写分配

表 17.1.7 TEX 最高位为 1 时内外缓存策略编码

S 位用于控制存储器的共享特性，设置 S=1，则二级存储器不可以缓存（Cache），如果设置 S=0，则可以缓存（Cache），一般我们设置该位为 0 即可。

C 位用于控制存储器的缓存特性，也就是是否可以 Cache，STM32F7 自带 Cache，如果我们想要某个存储器可以被 Cache，则必须设置 C=1。此位需要根据具体的需要设置。

B 位用于控制存储器的缓冲特性，设置 B=1，则二级存储器可以缓冲，即写回模式，设置 B=0，则二级存储器不可以缓冲，即写通模式。此位需根据具体的需要进行设置。

SRD[15:8]，这 8 个位用于控制子区域（sub region）使能。前面提到，STM32F7 的 MPU 最多支持 8 个 region，有时候可能不够用，通过子区域的概念，可以将每个 region 的内部进一步划分成更小的块，这就是 sub region，每个 sub region 可以独立地使能或除能（相当于可以部分地使能一个 region）。

sub region 的使用必须满足：

1，每个 region 必须 8 等分，每份是一个 sub region，其属性与主 region 完全相同。

2，可以被分为 8 个 sub region 的 region，其大小必须大于等于 256 字节。

SRD 中的 8 个位，每个位控制一个 sub region 是否被除能。如 SRD.4=0，则 4 号 sub region 被除能。如果某个 sub region 被除能，且其对应的地址范围又没有落在其它 region 中，则对该区的访问将引发 fault。

REGIONSIZE[5:1]，这 5 个位用于控制 region 的容量（大小），计算关系如下：

$$rsize=2^{(REGIONSIZE+1)}$$

rsize 即 region 的容量，必须大于等于 32 字节，即 REGIONSIZE 必须大于等于 4。region 的容量范围为：32B~4GB，根据实际需要进行设置。

SZENABLE 位，用于设置 region 的使能。该位一般最后设置，设置为 1，则启用此 region，使能 MPU 保护。

至此，关于 MPU 的简介就介绍完了，关于 MPU 更详细的说明，请参考：《STM32F7 编程手册》、《STM32 MPU 说明》和《Cortex M3 权威指南(中文)》第 14 章。接下来我们看看使用 HAL 库配置 MPU 相关函数和配置方法。MPU 相关的配置分布在头文件 stm32f7xx_hal_cortex.h 和对应的源文件 stm32f7xx_hal_cortex.c 中。

1） 禁止和使能 MPU 以及 MemManage 中断。

HAL 库中使能和禁止 MPU 以及 MemManage 中断的方法非常简单，使能函数为：

```
__STATIC_INLINE void HAL_MPU_Enable(uint32_t MPU_Control);
```

禁止函数为：

```
__STATIC_INLINE void HAL_MPU_Disable(void);
```

2） 配置某个区域的 MPU 保护参数。

上面我们讲过，在进行 MPU 配置之前我们必须先通过 MPU_RNR 区域编号寄存器用来选择下一个要配置的区域，然后通过配置 MPU_RBAR 基地址寄存器来配置基地址，最后通过区域属性和容量寄存器 RASR 来配置区域相关属性和参数。这些过程在 HAL 库中是通过函数 HAL_MPU_ConfigRegion 来实现的，函数声明如下：

```
void HAL_MPU_ConfigRegion(MPU_Region_InitTypeDef *MPU_Init);
```

该函数只有一个入口参数 MPU_Init，该参数为 MPU_Region_InitTypeDef 结构体指针类型，该结构体定义如下：

```
typedef struct
{
  uint8_t               Enable;          //区域使能/禁止
  uint8_t               Number;          //区域编号
```

```
    uint32_t        BaseAddress;        //配置区域基地址
    uint8_t         Size;               //区域容量
    uint8_t         SubRegionDisable;   //子 region 除能位段设置
    uint8_t         TypeExtField;       //类型扩展级别
    uint8_t         AccessPermission;   //设置访问权限
    uint8_t         DisableExec;        //允许/禁止取指
    uint8_t         IsShareable;        //禁止/允许共享
    uint8_t         IsCacheable;        //禁止/允许缓存
    uint8_t         IsBufferable;       //禁止/允许缓冲
}MPU_Region_InitTypeDef;
```

该结构体成员变量很多，每个成员变量的含义我们都在上面定义中有注释。这里大家注意，除了 BaseAddress 和 Number 两个成员变量是分别用来配置 MPU->RBAR 和 MPU->RNR 寄存器之外，其他成员变量都是用来配置 MPU->RASR 寄存器相关位，大家如果对这些配置项不理解的话，可以直接对照我们前面讲解的寄存器 MPU->RASR 各个位含义来理解。

MPU 配置相关 HAL 库函数我们就给大家讲解到这里。接下来我们看看硬件设计。

17.2 硬件设计

本章实验功能简介：本实验，我们将利用 STM32F7 自带的 MPU 功能，对一个特定的内存空间（数组，地址：0X20002000）进行写访问保护。开机时，串口调试助手显示：MPU closed，表示默认是没有写保护的。按 KEY0 可以往数组里面写数据，按 KEY1，可以读取数组里面的数据。按 KEY_UP 则开启 MPU 保护，此时，如果再按 KEY0 往数组写数据，就会引起 MemManage 错误，进入 MemManage_Handler 中断服务函数，此时 DS1 点亮，同时打印错误信息，最后软件复位，系统重启。DS0 用于提示程序正在运行，所有信息都是通过串口 1 输出(115200)，请用串口调试助手查看。

本实验需要用到的硬件资源有：

1）指示灯 DS0

2）串口 1

3）按键 KEY0、KEY1 和 KEY_UP(也称之为 WK_UP)

这些硬件资源，我们在之前的例程，都已经介绍过了，请参考之前的例程。

17.3 软件设计

打开本章 MPU 实验工程可以看到，我们在 HARDWARE 分组之下添加了 mpu.c 源文件，同时将对应的头文件 mpu.h 引入工程。打开 mpu.c 文件，代码如下：

```
//设置某个区域的 MPU 保护
//baseaddr:MPU 保护区域的基址(首地址)
//size:MPU 保护区域的大小(必须是 32 的倍数,单位为字节)
//可设置的值参考:CORTEX_MPU_Region_Size
//rnum:MPU 保护区编号,范围:0~7,最大支持 8 个保护区域
//可设置的值参考：CORTEX_MPU_Region_Number
//ap:访问权限,访问关系如下:
//可设置的值参考：CORTEX_MPU_Region_Permission_Attributes
//MPU_REGION_NO_ACCESS,无访问（特权&用户都不可访问）
//MPU_REGION_PRIV_RW,仅支持特权读写访问
```

```
//MPU_REGION_PRIV_RW_URO,禁止用户写访问（特权可读写访问）
//MPU_REGION_FULL_ACCESS,全访问（特权&用户都可访问）
//MPU_REGION_PRIV_RO,仅支持特权读访问
//MPU_REGION_PRIV_RO_URO,只读（特权&用户都不可以写）
//详见:STM32F7 Series Cortex-M7 processor programming manual.pdf,4.6 节,Table 89.
//返回值;0,成功.
//       其他,错误.
u8 MPU_Set_Protection(u32 baseaddr,u32 size,u32 rnum,u32 ap)
{
    MPU_Region_InitTypeDef MPU_Iniure;

    HAL_MPU_Disable();  //配置 MPU 之前先关闭 MPU,配置完成以后在使能 MPU

    MPU_Iniure.Enable=MPU_REGION_ENABLE; //使能该保护区域
    MPU_Iniure.Number=rnum;                 //设置保护区域
    MPU_Iniure.BaseAddress=baseaddr;        //设置基址
    MPU_Iniure.Size=size;                   //设置保护区域大小
    MPU_Iniure.SubRegionDisable=0X00;       //禁止子区域
    MPU_Iniure.TypeExtField=MPU_TEX_LEVEL0;    //设置类型扩展域为 level0
    MPU_Iniure.AccessPermission=(u8)ap;             //设置访问权限,
    MPU_Iniure.DisableExec=MPU_INSTRUCTION_ACCESS_ENABLE; //允许指令访问
    MPU_Iniure.IsShareable=MPU_ACCESS_NOT_SHAREABLE;        //禁止共用
    MPU_Iniure.IsCacheable=MPU_ACCESS_CACHEABLE;          //使能 cache
    MPU_Iniure.IsBufferable=MPU_ACCESS_BUFFERABLE;          //允许缓冲
    HAL_MPU_ConfigRegion(&MPU_Iniure);                      //配置 MPU
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);                 //开启 MPU
    return 0;
}

//设置需要保护的存储块
//必须对部分存储区域进行 MPU 保护,否则可能导致程序运行异常
//比如 MCU 屏不显示,摄像头采集数据出错等等问题...
void MPU_Memory_Protection(void)
{
    MPU_Set_Protection(0x60000000,MPU_REGION_SIZE_64MB,
                       MPU_REGION_NUMBER0,MPU_REGION_FULL_ACCESS);
                           //保护 MCU LCD 屏所在的 FMC 区域,,共 64M 字节
    MPU_Set_Protection(0x20000000,MPU_REGION_SIZE_512KB,
                       MPU_REGION_NUMBER1,MPU_REGION_FULL_ACCESS);
                       //保护整个内部 SRAM,包括 SRAM1,SRAM2 和 DTCM,共 512K 字节
    MPU_Set_Protection(0XC0000000,MPU_REGION_SIZE_32MB,
                       MPU_REGION_NUMBER2,MPU_REGION_FULL_ACCESS);
                               //保护 SDRAM 区域,共 32M 字节
```

```
    MPU_Set_Protection(0X80000000,MPU_REGION_SIZE_256MB,
                       MPU_REGION_NUMBER3,MPU_REGION_FULL_ACCESS);
                                    //保护整个 NAND FLASH 区域,共 256M 字节
}

//MemManage 错误处理中断
//进入此中断以后,将无法恢复程序运行!!
void MemManage_Handler(void)
{
    LED1(0);                        //点亮 DS1
    printf("Mem Access Error!!\r\n");   //输出错误信息
    delay_ms(1000);
    printf("Soft Reseting...\r\n");     //提示软件重启
    delay_ms(1000);
    NVIC_SystemReset();             //软复位
}
```

此部分总共 3 个函数：

MPU_Set_Protection 函数，用于设置某个区域（region）的详细参数，详见代码说明，通过该函数，我们可以设置某个存储区域的具体特性，从而实现内存保护。

MPU_Memory_Protection 函数，用于设置整个代码里面，我们需要保护的存储块，这里我们对 4 个存储块（使用了 4 个区域（region））进行了保护：

1，从 0x60000000 地址开始的 64MB 地址空间，禁止共用，禁止 cache，禁止缓冲，保护 MCU LCD 屏的访问地址取件，如不进行设置，可能导致 MCU LCD 白屏。

2，从 0x20000000 地址开始的 512KB 地址空间，包括 SRAM1，SRAM2 和 DTCM，禁止共用，允许 cache，允许缓冲。

3，从 0XC0000000 地址开始的 32MB 地址空间，即 SDRAM 的地址范围，禁止共用，允许 cache，允许缓冲。

4，从 0X80000000 地址开始的 256MB 地址空间，即 NAND FLASH 区域，禁止共用，禁止 cache，禁止缓冲，如不进行设置，可能导致 NAND FLASH 访问异常。

这四个地址空间的保护设置，可以提高代码的稳定性（其实就是减少使用 cache 导致的各种莫名奇妙的问题），请大家不要随意改动。此函数在本例程没有用到，不过我们在后续代码都会用到。

最后，MemManage_Handler 函数，用于处理产生 MemManage 错误的中断服务函数，在该函数里面点亮了 DS1，并输出一些串口信息，对系统进行软复位，以便观察本例程的实验结果。

头文件 mpu.h 内容非常简单，主要是函数声明，这里我们不做过多解释。

最后，打开 main.c 文件，代码如下：

```
u8 mpudata[128] __attribute__((at(0X20002000)));   //定义一个数组
int main(void)
{
    u8 i=0;
    u8 key;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
```

```
    Stm32_Clock_Init(432,25,2,9);     //设置时钟,216Mhz
    delay_init(216);                  //延时初始化
    uart_init(115200);                //串口初始化
    LED_Init();                       //初始化 LED
    KEY_Init();                       //按键初始化
    printf("\r\n\r\nMPU closed!\r\n"); //提示 MPU 关闭
    while(1)
    {
        key=KEY_Scan(0);
        if(key==WKUP_PRES)            //使能 MPU 保护数组 mpudata;
        {
            MPU_Set_Protection(0X20002000,128,0,MPU_REGION_PRIV_RO_URO,
                               0,0,1); //只读,禁止共用,禁止 catch,允许缓冲
            printf("MPU open!\r\n");  //提示 MPU 打开
        }else if(key==KEY0_PRES)      //向数组中写入数据，如果开启了 MPU 保护的
                                      //话会进入内存访问错误!
        {
            printf("Start Writing data...\r\n");
            sprintf((char*)mpudata,"MPU test array %d",i);
            printf("Data Write finshed!\r\n");
        }else if(key==KEY1_PRES)      //从数组中读取数据，不管有没有开启 MPU 保护
                                      //都不会进入内存访问错误!
        {
            printf("Array data is:%s\r\n",mpudata);
        }else delay_ms(10);
        i++;
        if((i%50)==0) LED0(led0sta^=1); //LED0 取反
    }
}
```

此部分代码，我们定义了一个 128 字节大小的数组：mpudata，其首地址为 0X20002000，默认情况下，MPU 保护关闭，可以对该数组进行读写访问。当我们按下 KEY_UP 按键的时候，通过 MPU_Set_Protection 函数，对其 0X20002000 为起始地址，大小为 128 字节的内存空间进行保护，仅支持特权读访问，此时如果再按 KEY0，对数组进行写入操作，则会引起 MemManage 访问异常，进入 MemManage_Handler 中断服务函数，执行相关操作。

其他的代码比较简单，这里就不多做说明了，在整个代码编译通过之后，我们就可以开始下载验证了。

17.4 下载验证

我们把程序下载到阿波罗 STM32F767 开发板，可以看到板子上的 DS0 开始闪烁，说明程序已经在跑了。然后，打开串口调试助手（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 17.4.1 所示信息（如果没有提示信息，请先按复位）：

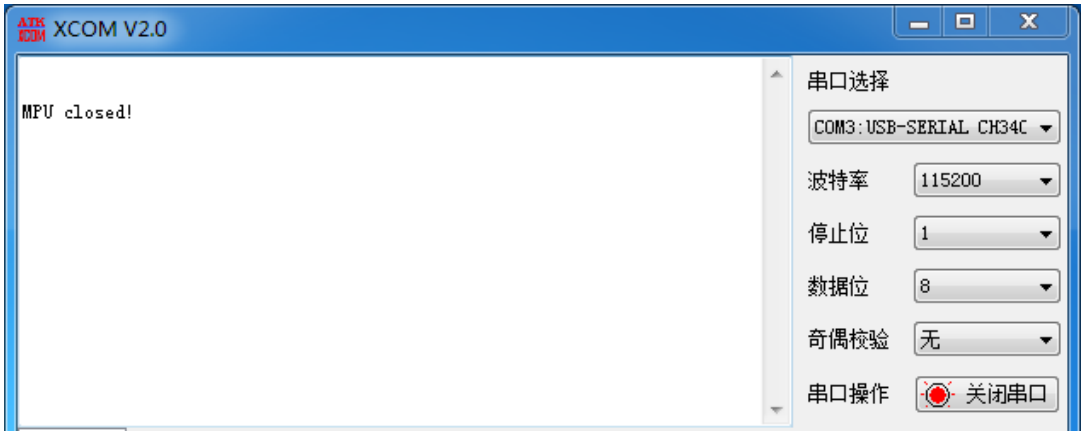


图 17.4.1 串口调试助手收到的信息

从图 17.4.1 可以看出，此时串口助手提示：MPU Closed，即 MPU 保护是关闭的，我们可以按 KEY0 往数组里面写入数据，按 KEY1，可以读取刚刚写入的数据，按 KEY_UP，则开启 MPU 保护，提示：MPU open!，此时，如果再按 KEY0，往数组里面写数据的话，则会引起 MemManage 访问异常，进入 MemManage_Handler 中断服务函数，点亮 DS0，并提示：Mem Access Error!!，并在 1 秒钟以后，重启系统（软复位），如图 17.4.2 所示：

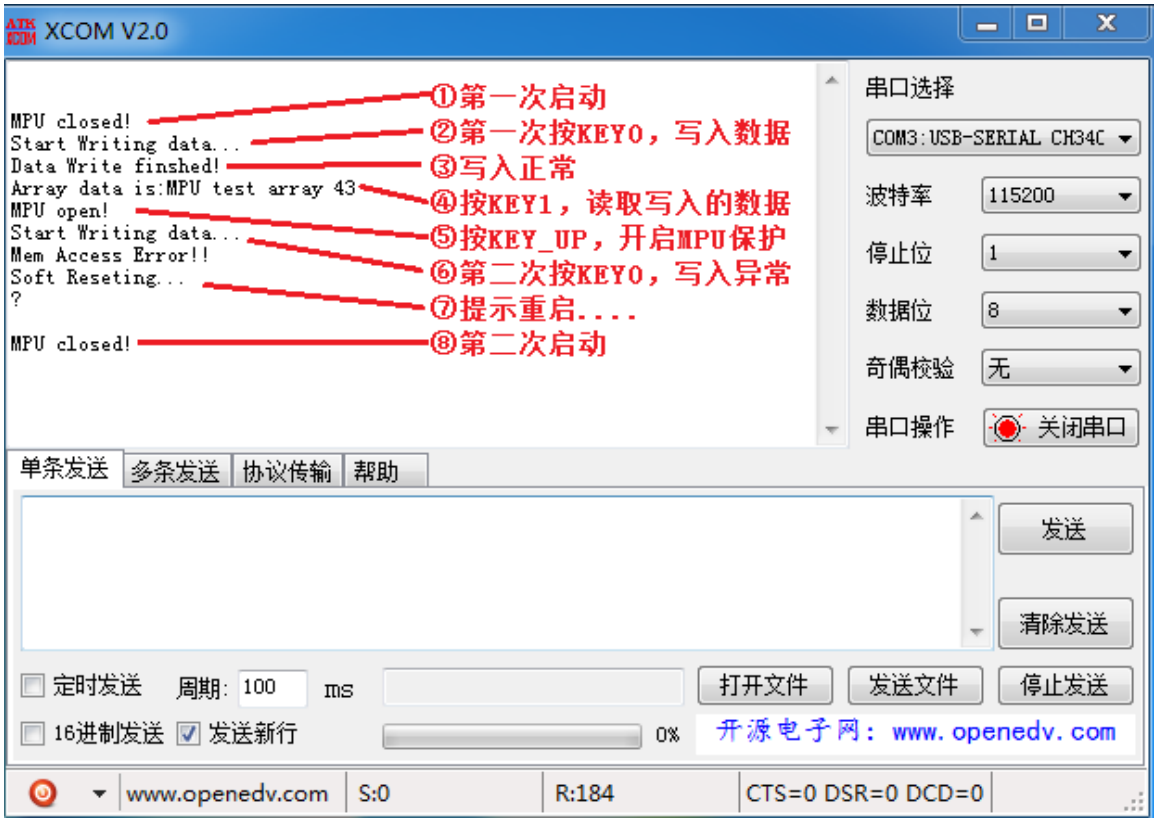


图 17.4.2 串口调试助手显示运行结果

整个过程，验证了我们代码的正确性，通过 MPU 实现了对特定内存的写保护功能。通过 MPU，我们可以提高系统的可靠性，使代码更加安全的运行。

第十八章 TFTLCD（MCU 屏）实验

在第 16 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 的 TFT LCD 模块（MCU 屏），该模块采用 TFTLCD 面板，可以显示 16 位色的真彩图片。在本章中，我们将使用阿波罗 STM32F767 开发板底板上的 TFTLCD 接口（仅支持 MCU 屏，本章仅介绍 MCU 屏的使用），来点亮 TFTLCD，并实现 ASCII 字符和彩色的显示等功能，并在串口打印 LCD 控制器 ID，同时在 LCD 上面显示。本章分为如下几个部分：

18.1 TFTLCD&FMC 简介
18.2 硬件设计
18.3 软件设计
18.4 下载验证
18.5 STM32CubeMX 配置 FMC（SRAM）

18.1 TFTLCD&FMC 简介

本章我们将通过 STM32F767 的 FMC 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 FMC。

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(MCU 接口)，该模块有如下特点：

1，2.8’/3.5’/4.3’/7’等 4 种大小的屏幕可选。
2，320×240 的分辨率（3.5’分辨率为:320*480，4.3’和 7’分辨率为：800*480）。
3，16 位真彩显示。
4，自带触摸屏，可以用来作为控制输入。

本章，我们以 2.8 寸（其他 3.5 寸/4.3 寸等 LCD 方法类似，请参考 2.8 的即可）的 ALIENTEK TFTLCD 模块为例介绍，该模块支持 65K 色显示，显示分辨率为 320×240，接口为 16 位的 80 并口，自带触摸屏。

该模块的外观图如图 18.1.1.1 所示：

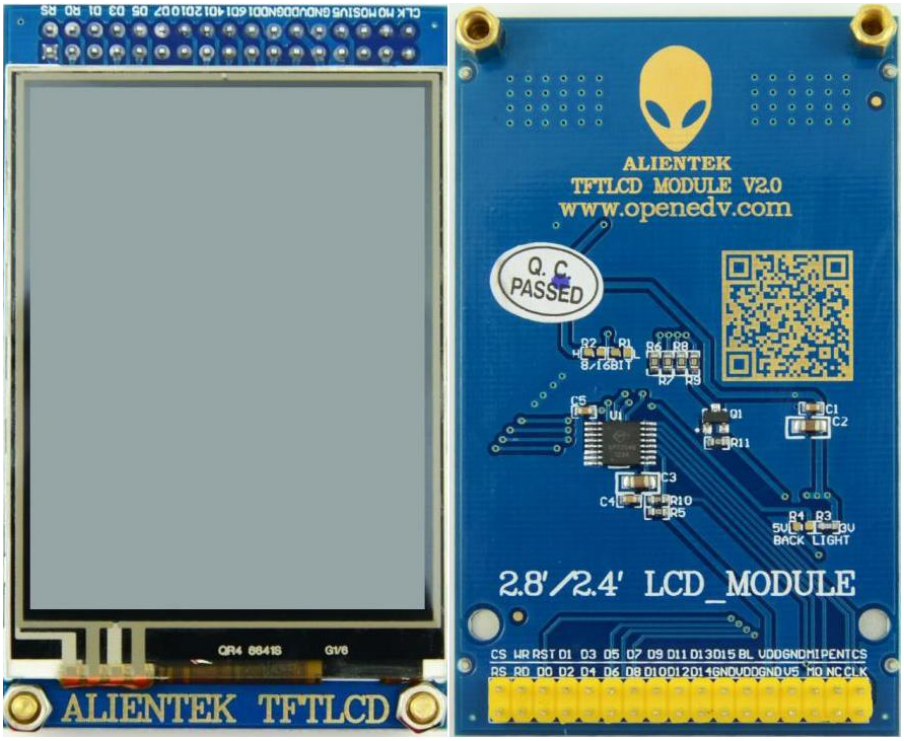


图 18.1.1.1 ALIENTEK 2.8 寸 TFTLCD 外观图

模块原理图如图 18.1.1.2 所示：

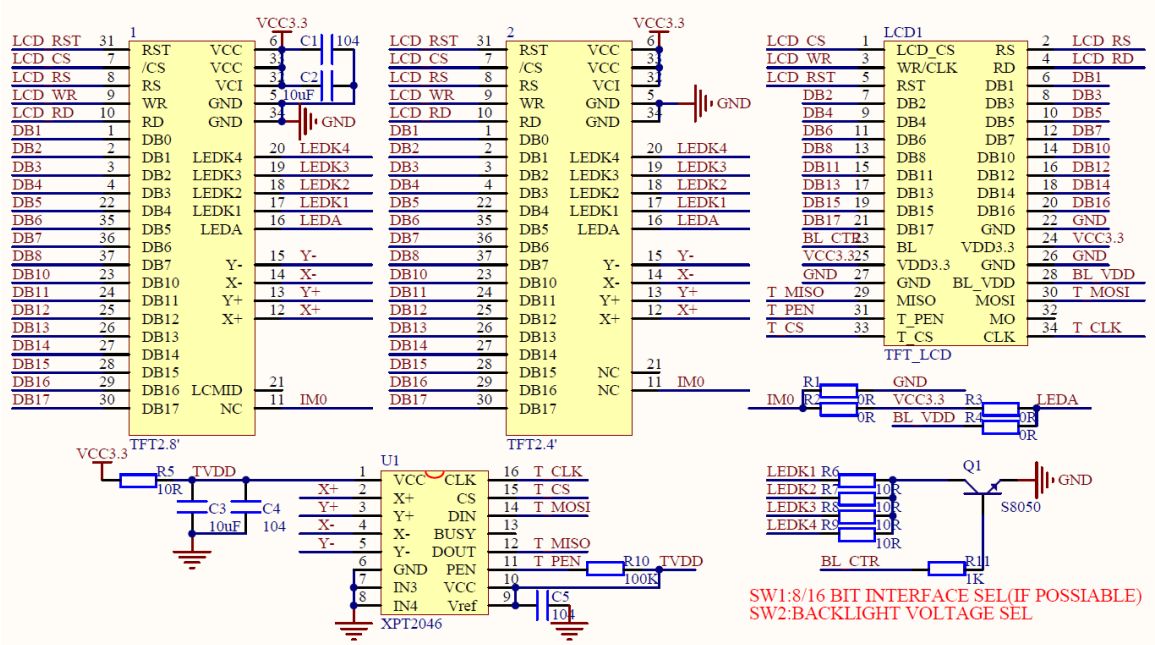


图 18.1.1.2　ALIENTEK 2.8 寸 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，接口定义如图 18.1.1.3 所示：

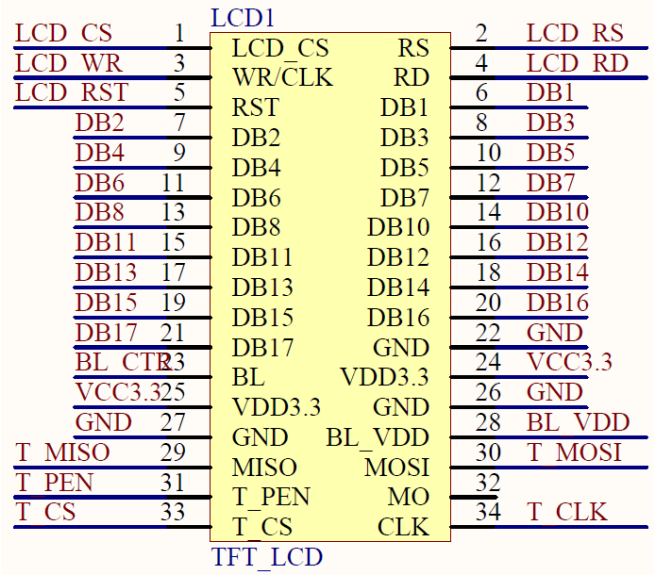


图 18.1.1.3　ALIENTEK 2.8 寸 TFTLCD 模块接口图

从图 18.1.1.3 可以看出，ALIENTEK TFTLCD 模块采用 16 位的并方式与外部连接，之所以不采用 8 位的方式，是因为彩屏的数据量比较大，尤其在显示图片的时候，如果用 8 位数据线，就会比 16 位方式慢一倍以上，我们当然希望速度越快越好，所以我们选择 16 位的接口。图 18.1.1.3 还列出了触摸屏芯片的接口，关于触摸屏本章我们不多介绍，后面的章节会有详细的介绍。该模块的 80 并口有如下一些信号线：

CS：TFTLCD 片选信号。

WR：向 TFTLCD 写入数据。

RD：从 TFTLCD 读取数据。

D[15：0]：16 位双向数据线。

RST：硬复位 TFTLCD。

RS：命令/数据标志（0，读写命令；1，读写数据）。

80 并口在上一节我们已经有详细的介绍了，这里我们就不再介绍，需要说明的是，TFTLCD 模块的 RST 信号线是直接接到 STM32F767 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 4 种不同尺寸和分辨率的 TFTLCD 模块，其驱动芯片为：ILI9341/NT35310/NT35510/SSD1963 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。

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：

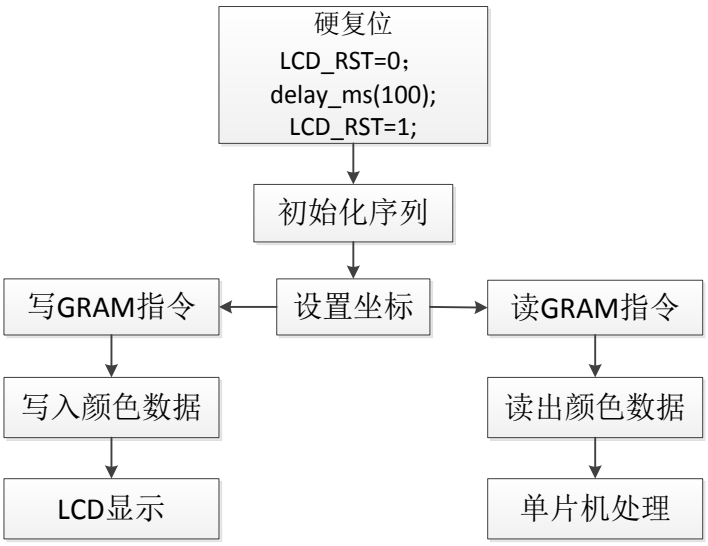


图 18.1.1.5 TFTLCD 使用流程

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 STM32F767 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 FMC，FMC 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，因为阿波罗 STM32F767 开发板的 LCD 接口，将 TFTLCD 的 RST 同 STM32F767 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。

STM32F767xx 系列芯片都带有 FMC 接口，即可变存储存储控制器，能够与同步或异步存储器、SDRAM 存储器和 NAND FLASH 等连接，STM32F767 的 FMC 接口支持包括 SRAM、SDRAM、NAND FLASH、NOR FLASH 和 PSRAM 等存储器。FMC 的框图如图 18.1.2.1 所示：

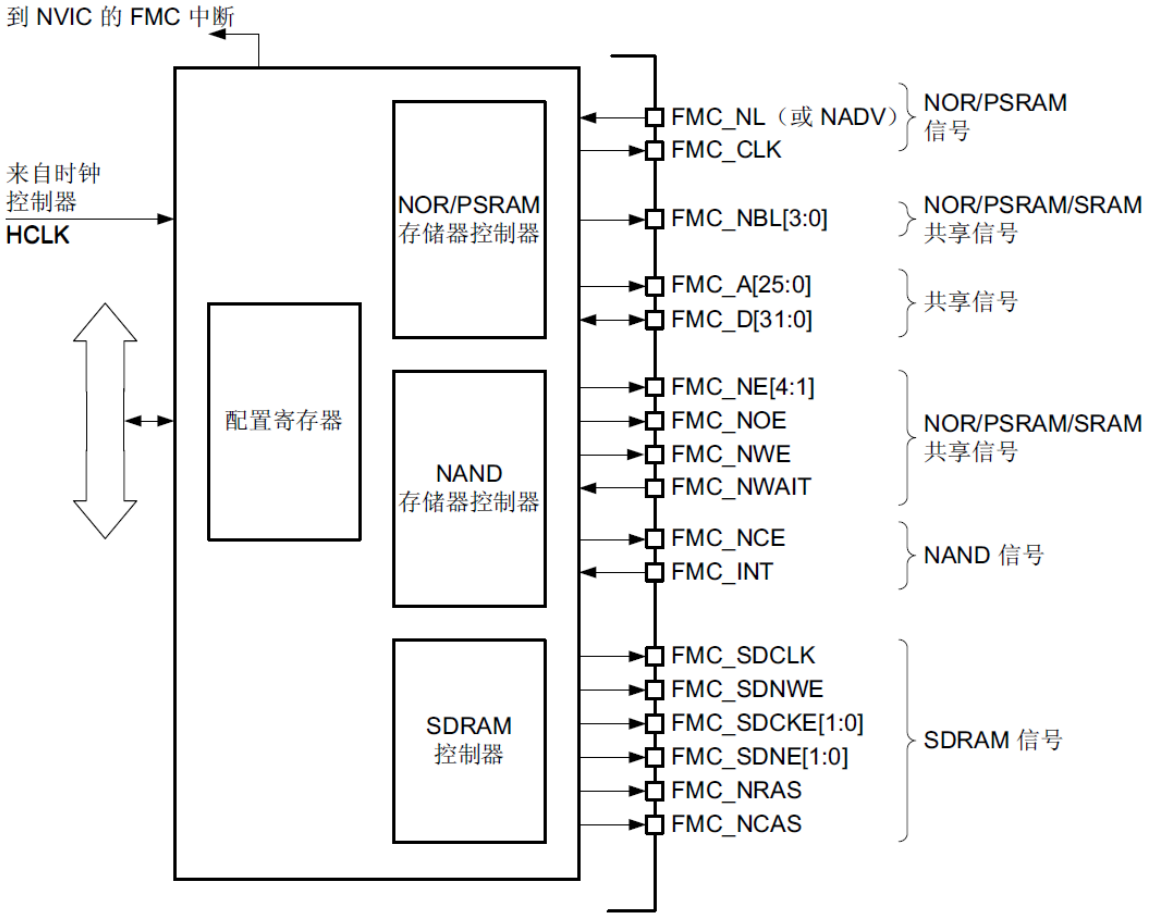


图 18.1.2.1 FMC 框图

从上图我们可以看出，STM32F767 的 FMC 将外部设备分为 3 类：NOR/PSRAM 设备、NAND 设备和 SDRAM 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 FMC_NE1 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 FMC 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 FMC 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，阿波罗 STM32F767 开发板是把 RS 连接在 A18 上面的。

STM32F767 的 FMC 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 FMC 的外部设备地址映像，STM32F767 的 FMC 将外部存储器划分为 6 个固定大小为 256M 字节的存储区域，如图 18.1.2.2 所示：

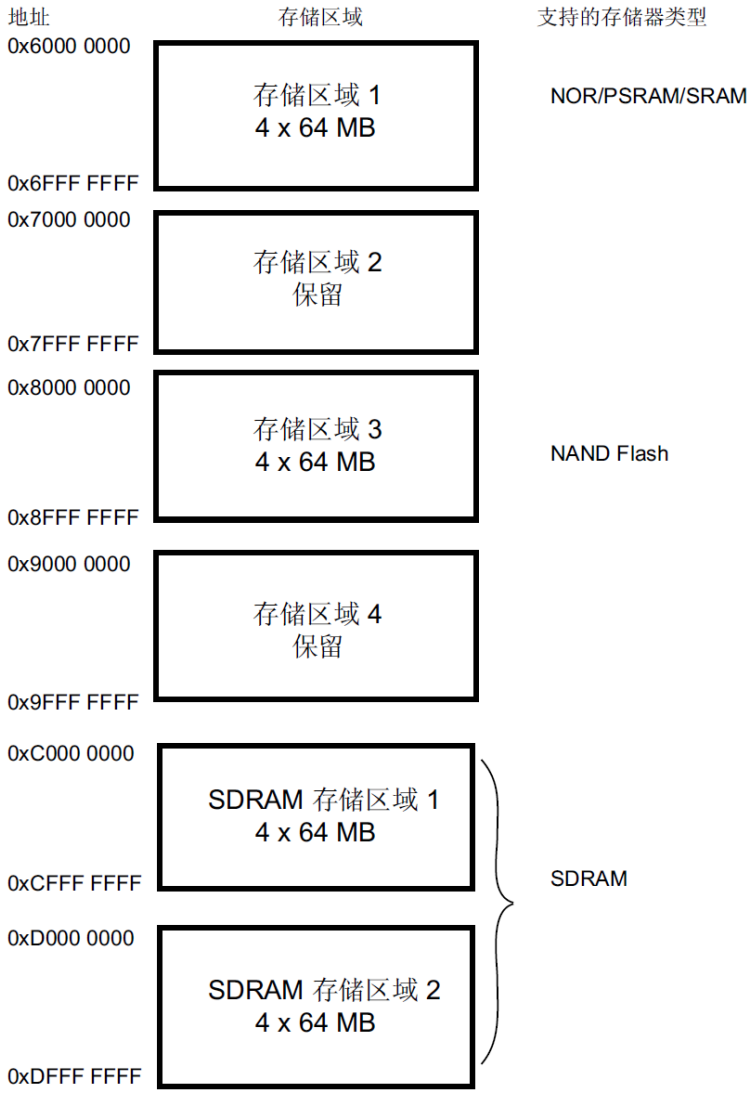


图 18.1.2.2 FMC 存储块地址映像

从上图可以看出，FMC 总共管理 1.5GB 空间，拥有 6 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《STM32F7 中文参考手册》第 13 章（286 页）的相关介绍。

STM32F767 的 FMC 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 FMC_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	FMC_NE1	0X6000,0000~63FF,FFFF	00	FMC_A[25:0]
第 2 区	FMC_NE2	0X6400,0000~67FF,FFFF	01	
第 3 区	FMC_NE3	0X6800,0000~6BFF,FFFF	10	
第 4 区	FMC_NE4	0X6C00,0000~6FFF,FFFF	11	

表 18.1.2.1 Bank1 存储区选择表

HADDR[25:0]位包含外部存储器的地址，由于 HADDR 为字节地址，而存储器按字寻址，所以，根据存储器数据宽度的不同，实际上向存储器发送的地址也有所不同，如表 18.1.2.2 所示：

存储器宽度	向存储器发出的数据地址	最大存储器容量（位）
8 位	HADDR[25:0]	64 MB x 8 = 512 Mb
16 位	HADDR[25:1] >> 1	64 MB/2 x 16 = 512 Mb
32 位	HADDR[25:2] >> 2	64 MB/4 x 32 = 512 Mb

表 18.1.2.2 NOR/PSRAM 外部存储器地址

因此，FMC 内部 HADDR 与存储器寻址地址的实际对应关系就是：

当接的是 32 位宽度存储器的时候：HADDR[25:2]→ FMC_A [23:0]。

当接的是 16 位宽度存储器的时候：HADDR[25:1]→ FMC_A [24:0]。

当接的是 8 位宽度存储器的时候：HADDR[25:0]→ FMC_A [25:0]。

不论外部接 8 位/16 位/32 位宽设备，FMC_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]→ FMC_A[24:0]，相当于右移了一位，这里请大家特别留意。另外，HADDR[27:26]的设置，是不需要我们干预的，比如：当你选择使用 Bank1 的第一个区，即使用 FMC_NE1 来连接外部设备的时候，即对应了 HADDR[27:26]=00，我们要做的就是配置对应第 1 区的寄存器组，来适应外部设备即可。STM32F767 的 FMC 各 Bank 配置寄存器如表 18.1.2.3 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	FMC_BCR1/2/3/4 FMC_BTR1/2/2/3 FMC_BWTR1/2/3/4
NAND FLASH /PC CARD 控制器	Bank2	0X7000,0000~ 0X7FFF,FFFF	NAND FLASH	FMC_PCR FMC_SR FMC_PMEM FMC_PATT FMC_ECCR
	Bank3	0X8000,0000~ 0X8FFF,FFFF		
	Bank4	0X9000,0000~ 0X9FFF,FFFF	保留	保留
SDRAM 控制器	Bank5	0XC000,0000~ 0XCFFF,FFFF	SDRAM	FMC_SDCR1/2 FMC_SDTR1/2 FMC_SDCMR FMC_SDRTR FMC_SDSR
	Bank6	0XD000,0000~ 0XDFFF,FFFF	SDRAM	

表 18.1.2.3 FMC 各 Bank 配置寄存器表

对于 NOR FLASH 控制器，主要是通过 FMC_BCRx、FMC_BTRx 和 FMC_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 FMC 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。FMC 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，FMC 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 FMC_CLK。此时需要的设置的时间参数有 2 个：

1，HCLK 与 FMC_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，FMC 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)。FMC 综合了 SRAM、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 18.1.2.4 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、ADDHLD
同步突发		根据同步时钟 FMC_CK 读取多个顺序单元的数据	CLKDIV、DATLAT

表 18.1.2.4 NOR FLASH/PSRAM 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 FMC 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：

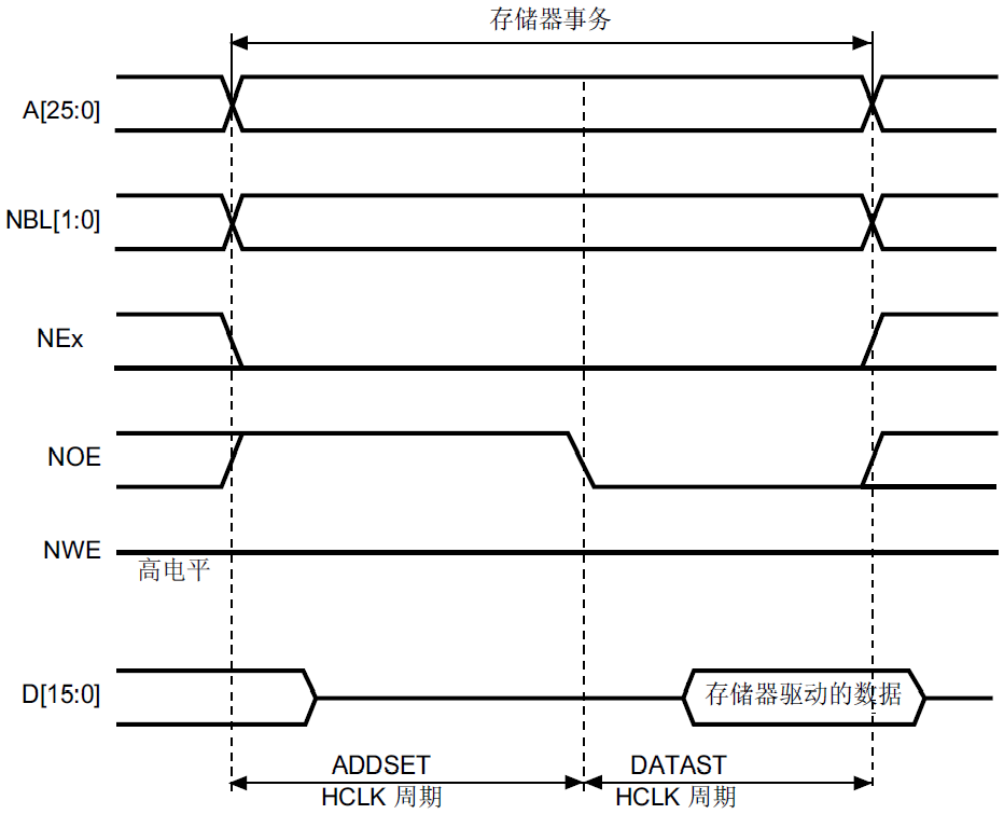


图 18.1.2.3 模式 A 读操作时序图

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 FMC 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：

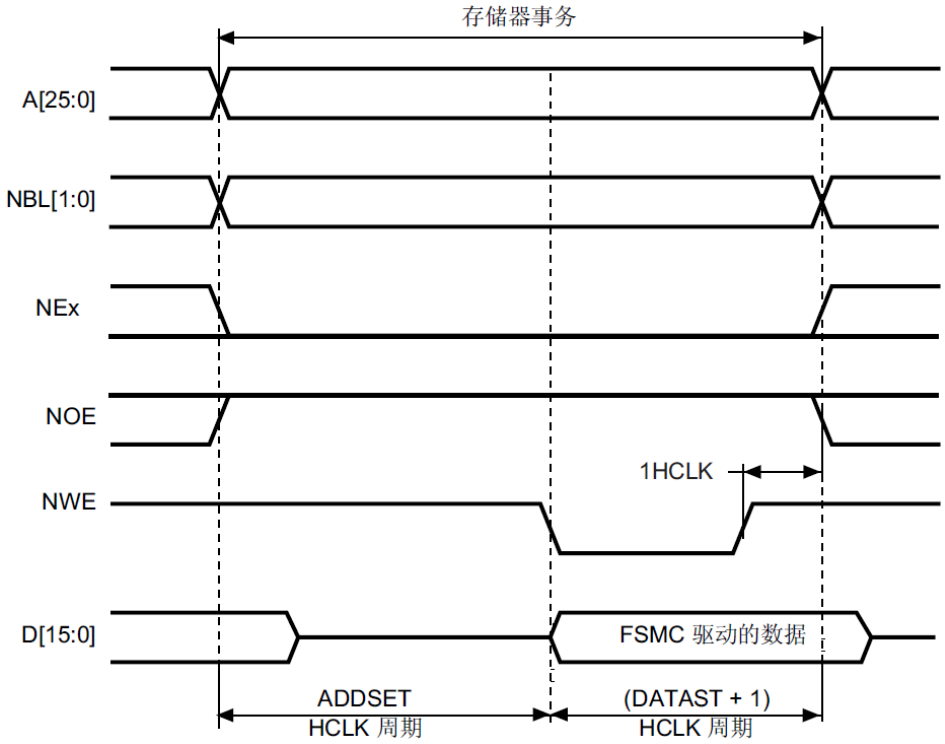


图 18.1.2.4 模式 A 写操作时序

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：FMC_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
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	WFDIS	CCLK EN	CBURST RW	CPSIZE[2:0]		
										rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ASYNC WAIT	EXT MOD	WAIT EN	WREN	WAIT CFG	Res.	WAIT POL	BURST EN	Res.	FACC EN	MWID		MTYP		MUX EN	MBK EN
rw	rw	rw	rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw

图 18.1.2.5 FMC_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 表示 32 位数据模式；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；01 表示 PSRAM；10 表示 NOR FLASH/OneNAND

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：FMC_BTRx（x=1~4），该寄存器各位描述如图 18.1.2.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	ACCMOD		DATLAT				CLKDIV				BUSTURN			
		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
DATAST								ADDHLD				ADDSET			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 18.1.2.6 FMC_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 和 NOR 闪存存储器等。如果 FMC_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(FMC_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 4.6ns 左右（1/216Mhz），为了兼容其他屏，我们这里设置 DATAST 为 80，也就是 80 个 HCLK 周期，时间大约是 368ns。

ADDSET[3:0]：地址建立时间。其建立时间为：ADDSET 个 HCLK 周期，最大为 15 个 HCLK 周期。对 ILI9341 来说，这里相当于 RD 高电平持续时间，为 90ns，我们设置 ADDSET 为最大 15，即 15*4.6=69ns（略超）。

最后，我们再来看看 SRAM/NOR 闪写时序寄存器：FMC_BWTRx（x=1~4），该寄存器各位描述如图 18.1.2.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	ACCMOD		Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	BUSTURN			
		rw	rw									rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATAST								ADDHLD				ADDSET			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 18.1.2.7 FMC_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 FMC_BTRx 一模一样，只是这里对应的是写操作的时序，ACCMOD 设置同 FMC_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 4，即 4 个 HCLK 周期，时间约为 18.4ns。然后 ADDSET 设置为 4，即 4 个 HCLK 周期，时间为 18.4ns。

至此，我们对 STM32F767 的 FMC 介绍就差不多了，关于 FMC 的详细介绍，请大家参考《STM32F7 中文参考手册》第 13 章。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 FMC_BCRx、FMC_BTRx、FMC_BWTRx 等这个单独的寄存器，而是将他们进行了一些组合。

FMC_BCRx 和 FMC_BTRx，组合成 BTCR[8]寄存器组，他们的对应关系如下：

BTCR[0]对应 FMC_BCR1，BTCR[1]对应 FMC_BTR1

BTCR[2]对应 FMC_BCR2，BTCR[3]对应 FMC_BTR2

BTCR[4]对应 FMC_BCR3，BTCR[5]对应 FMC_BTR3

BTCR[6]对应 FMC_BCR4，BTCR[7]对应 FMC_BTR4

FMC_BWTRx 则组合成 BWTR[7]，他们的对应关系如下：

BWTR[0]对应 FMC_BWTR1，BWTR[2]对应 FMC_BWTR2，

BWTR[4]对应 FMC_BWTR3，BWTR[6]对应 FMC_BWTR4，

BWTR[1]、BWTR[3]和 BWTR[5]保留，没有用到。

通过上面的讲解，通过对 FSC 相关的寄存器的描述，大家对 FMC 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 FMC 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 FMC 的配置的呢？FMC_BCRx，FMC_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。与 SRAM 和 FMC 相关的库函数定义和声明在源文件 stm32f7xx_hal_fmc.c/stm32f7xx_hal_sram.c 以及头文件 stm32f7xx_hal_fmc.h/stm32f7xx_hal_sram.h 中。

1） 使能 FMC 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 FMC 时钟使能方法：

```
__HAL_RCC_FMC_CLK_ENABLE ();                    //使能 FMC 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,
                                FMC_NORSRAM_TimingTypeDef *Timing,
                                FMC_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 FMC_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 FMC_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 FMC_Bank1E 即可，同样 MDK 定义好了宏定义标识符 FMC_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 FMC_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 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 ContinuousClock;       //启用/禁止 FMC 时钟输出到外部存储设备
  uint32_t WriteFifo;
  uint32_t PageSize;
}FMC_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 FMC_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 FMC_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 FMC_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 FMC_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 FMC_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 FMC_EXTENDED_MODE_ENABLE。ContinuousClock 用来设置启用/禁止 FMC 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 FMC_CONTINUOUS_CLOCK_SYNC_ASYNC。其他参数 WriteBurst，BurstAccessMode，WaitSignalPolarity，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 FMC_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
}FMC_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 FMC_ACCESS_MODE_A，FMC_ACCESS_MODE_B, FMC_ACCESS_MODE_C 和 FMC_ACCESS_MODE_D，这里我们用是异步模式 A，所以取值为 FMC_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，FMC 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。这里以 SDRAM 为例，HAL 提供库支持文件 stm32f7xx_hal_sdram.c 和头文件 stm32f7xx_hal_sdram.h，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：

```
__FMC_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 阿波罗 STM32F767 开发板的连接，阿波罗 STM32F767 开发板底板的 LCD 接口和 ALIENTEK TFTLCD 模块直接可以对插，连接关系如图 18.2.1 所示：

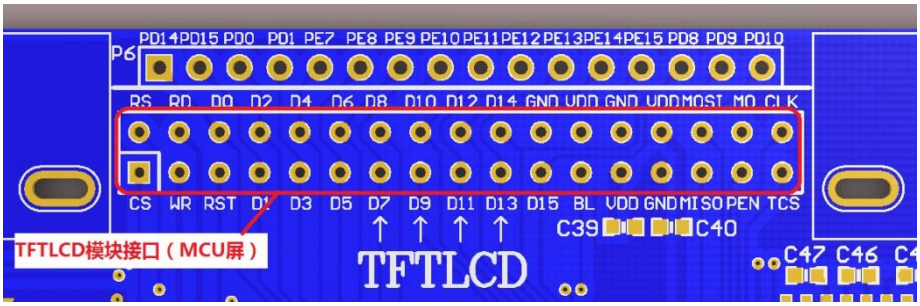


图 18.2.1 TFTLCD 与开发板连接示意图

图 18.2.1 中圈出来的部分就是连接 TFTLCD 模块的接口，液晶模块直接插上去即可。

在硬件上，TFTLCD 模块与阿波罗 STM32F767 开发板的 IO 口对应关系如下：

LCD_BL(背光控制)对应 PB5;

LCD_CS 对应 PD7 即 FMC_NE1;

LCD _RS 对应 PD13 即 FMC_A18;

LCD _WR 对应 PD5 即 FMC_NWE;

LCD _RD 对应 PD4 即 FMC_NOE;

LCD _D[15:0]则直接连接在 FMC_D15~FMC_D0;

这些线的连接，阿波罗 STM32F767 开发板的内部已经连接好了，我们只需要将 TFTLCD 模块插上去就好了。实物连接（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

打开我们光盘的实验 13 TFTLCD（MCU 屏）工程可以看到我们添加了两个文件 lcd.c 和头文件 lcd.h。同时，FMC 和 SRAM 相关的库函数和声明定义在源文件

stm32f7xx_hal_fmc.c/stm32f7xx_hal_sdram.c 和头文件 stm32f7xx_hal_fmc.h /stm32f7xx_hal_sram.h 中。

在 lcd.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。完整版的代码见光盘→4，程序源码→标准例程-寄存器版本→实验 13 TFTLCD（MCU 屏）实验 的 lcd.c 文件。

本实验，我们用到 FMC 驱动 LCD，通过前面的介绍，我们知道 TFTLCD 的 RS 接在 FMC 的 A18 上面，CS 接在 FMC_NE1 上，并且是 16 位数据总线。即我们使用的是 FMC 存储器 1 的第 1 区，我们定义如下 LCD 操作结构体（在 lcd.h 里面定义）：

```
//LCD 地址结构体
typedef struct
{
    vu16 LCD_REG;
    vu16 LCD_RAM;
} LCD_TypeDef;
//使用 NOR/SRAM 的 Bank1.sector1,地址位 HADDR[27,26]=00 A18 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)(0x60000000 | 0x0007FFFE))
#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.sector1 就是从地址 0X60000000 开始，而 0x0007FFFE，则是 A18 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 A18 为例，0x0007FFFE 转换成二进制就是：0111 1111 1111 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：A18:A0=011 1111 1111 1111 1111，此时 A18 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 0x0007FFFE +0X02），那么：A18:A0=100 0000 0000 0000 0000，时 A18 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 0X6007,FFFE，对应 A18 的状态为 0(即 RS=0)，而 LCD->LCD_RAM 的地址就是 0X6008,0000（结构体地址自增），对应 A18 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 FMC 硬件自动控制，不需要我们手动设置了。接下来，我们先介绍一下 lcd.h 里面的另一个重要结构体:

```
//LCD 重要参数集
typedef struct
{
    u16 width;          //LCD 宽度
    u16 height;         //LCD 高度
    u16 id;             //LCD ID
    u8  dir;            //横屏还是竖屏控制：0，竖屏；1，横屏。
```

```
    u16  wramcmd;           //开始写 gram 指令
    u16 setxcmd;            //设置 x 坐标指令
    u16 setycmd;            //设置 y 坐标指令
}_lcd_dev;
//LCD 参数
extern _lcd_dev lcddev; //管理 LCD 重要参数
```

该结构体用于保存一些 LCD 重要参数信息，比如 LCD 的长宽、LCD ID（驱动 IC 型号）、LCD 横竖屏状态等，这个结构体虽然占用了十几个字节的内存，但是却可以让我们的驱动函数支持不同尺寸的 LCD，同时可以实现 LCD 横竖屏切换等重要功能，所以还是利大于弊的。有了以上了解，下面我们开始介绍 lcd.c 里面的一些重要函数。

先看 7 个简单，但是很重要的函数：

```
//写寄存器函数
//regval:寄存器值
void LCD_WR_REG(vu16 regval)
{
    regval=regval;      //使用-O2 优化的时候,必须插入的延时
    LCD->LCD_REG=regval;//写入要写的寄存器序号
}
//写 LCD 数据
//data:要写入的值
void LCD_WR_DATA(vu16 data)
{
    data=data;              //使用-O2 优化的时候,必须插入的延时
    LCD->LCD_RAM=data;
}
//读 LCD 数据
//返回值:读到的值
u16 LCD_RD_DATA(void)
{
    vu16 ram;               //防止被优化
    ram=LCD->LCD_RAM;
    return ram;
}
//写寄存器
//LCD_Reg:寄存器地址
//LCD_RegValue:要写入的数据
void LCD_WriteReg(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 FMC 自动控制了 WR/RD/CS 等这些信号，所以这 7 个函数实现起来都非常简单，我们就不多说，注意，上面有几个函数，我们添加了一些对 MDK -O2 优化的支持，去掉的话，在-O2 优化的时候会出问题。这些函数实现功能见函数前面的备注，通过这几个简单函数的组合，我们就可以对 LCD 进行各种操作了。

第七个要介绍的函数是坐标设置函数，该函数代码如下：

```
//设置光标位置
//Xpos:横坐标
//Ypos:纵坐标
void LCD_SetCursor(u16 Xpos, u16 Ypos)
{
    if(lcddev.id==0X9341||lcddev.id==0X5310)
    {
        LCD_WR_REG(lcddev.setxcmd);
        LCD_WR_DATA(Xpos>>8);LCD_WR_DATA(Xpos&0XFF);
        LCD_WR_REG(lcddev.setycmd);
        LCD_WR_DATA(Ypos>>8);LCD_WR_DATA(Ypos&0XFF);
    }else if(lcddev.id==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);
    }
}
```

该函数实现将 LCD 的当前操作点设置到指定坐标(x,y)。因为 9341/5310/1963/5510 等的设置有些不太一样，所以进行了区别对待。

接下来我们介绍第八个函数：画点函数。该函数实现代码如下：

```
//画点
//x,y:坐标
//POINT_COLOR:此点的颜色
void LCD_DrawPoint(u16 x,u16 y)
{
    LCD_SetCursor(x,y);          //设置光标位置
    LCD_WriteRAM_Prepare(); //开始写入 GRAM
    LCD->LCD_RAM=POINT_COLOR;
}
```

该函数实现比较简单，就是先设置坐标，然后往坐标写颜色。其中 POINT_COLOR 是我们定义的一个全局变量，用于存放画笔颜色，顺带介绍一下另外一个全局变量：BACK_COLOR，该变量代表 LCD 的背景色。LCD_DrawPoint 函数虽然简单，但是至关重要，其他几乎所有上层函数，都是通过调用这个函数实现的。

有了画点，当然还需要有读点的函数，第九个介绍的函数就是读点函数，用于读取 LCD 的 GRAM，这里说明一下，为什么 OLED 模块没做读 GRAM 的函数，而这里做了。因为 OLED 模块是单色的，所需要全部 GRAM 也就 1K 个字节，而 TFTLCD 模块为彩色的，点数也比 OLED 模块多很多，以 16 位色计算，一款 320×240 的液晶，需要 320×240×2 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==0X5310||lcddev.id==0X1963)
        LCD_WR_REG(0X2E);//9341/3510/1963 发送读 GRAM 指令
    else if(lcddev.id==0X5510)LCD_WR_REG(0X2E00);//5510 发送读 GRAM 指令
    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
    //9341/NT35310/NT35510 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 的值,R 在前,G 在后,各占 8 位
    g<<=8;
    return (((r>>11)<<11)|((g>>10)<<5)|(b>>11));      //需要公式转换一下
}
```

在 LCD_ReadPoint 函数中，因为我们的代码不止支持一种 LCD 驱动器，所以，我们根据不同的 LCD 驱动器（(lcddev.id）型号，执行不同的操作，以实现对各个驱动器兼容，提高函数的通用性。

第十个要介绍的是字符显示函数 LCD_ShowChar，该函数同前面 OLED 模块的字符显示函数差不多，但是这里的字符显示函数多了 1 个功能，就是可以以叠加方式显示，或者以非叠加方式显示。叠加方式显示多用于在显示的图片上再显示字符。非叠加方式一般用于普通的显示。该函数实现代码如下：

```
//在指定位置显示一个字符
//x,y:起始坐标
//num:要显示的字符:" "--->"~"
//size:字体大小 12/16/24/32
//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 字库是从空格开始取模，所以-' '就是对应字符的字库
    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 if(size==32)temp=asc2_3216[num][t];  //调用 3216 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四个字符集点阵数据数组 asc2_3216、asc2_2412、asc2_1206 和 asc2_1608，这几个字符集的点阵数据的提取方式，同十六章介绍的提取方法是一模一样的。详细请参考第十六章。

最后，我们再介绍一下 TFTLCD 模块的初始化函数 LCD_Init，该函数先配置 FMC 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_Initure;
  FMC_NORSRAM_TimingTypeDef FMC_ReadWriteTim;
  FMC_NORSRAM_TimingTypeDef FMC_WriteTim;

  __HAL_RCC_GPIOB_CLK_ENABLE();           //开启 GPIOB 时钟
  GPIO_Initure.Pin=GPIO_PIN_5;            //PB5,背光控制
  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(GPIOB,&GPIO_Initure);

  LCD_MPU_Config();                       //使能 MPU 保护 LCD 区域
```

```
SRAM_Handler.Instance= FMC_NORSRAM_DEVICE;       //SRAM BANK1
SRAM_Handler.Extended= FMC_NORSRAM_EXTENDED_DEVICE;

SRAM_Handler.Init.NSBank=FMC_NORSRAM_BANK1;          //使用 NE1
SRAM_Handler.Init.DataAddressMux=FMC_DATA_ADDRESS_MUX_DISABLE;
                                                     //地址/数据线不复用
SRAM_Handler.Init.MemoryType=FMC_MEMORY_TYPE_SRAM;     //SRAM
SRAM_Handler.Init.MemoryDataWidth=FMC_NORSRAM_MEM_BUS_WIDTH_16;
                                                     //16 位数据宽度
SRAM_Handler.Init.BurstAccessMode=FMC_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
SRAM_Handler.Init.WaitSignalPolarity=FMC_WAIT_SIGNAL_POLARITY_LOW;
                                   //等待信号的极性,仅在突发模式访问下有用
SRAM_Handler.Init.WaitSignalActive=FMC_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
SRAM_Handler.Init.WriteOperation=FMC_WRITE_OPERATION_ENABLE;
                                                      //存储器写使能
SRAM_Handler.Init.WaitSignal=FMC_WAIT_SIGNAL_DISABLE;
                                                     //等待使能位,此处未用到
SRAM_Handler.Init.ExtendedMode=FMC_EXTENDED_MODE_ENABLE;
                                                   //读写使用不同的时序
SRAM_Handler.Init.AsynchronousWait=FMC_ASYNCHRONOUS_WAIT_DISABLE;
                                 //是否使能同步传输模式下的等待信号,此处未用到
SRAM_Handler.Init.WriteBurst=FMC_WRITE_BURST_DISABLE;
                                                          //禁止突发写
SRAM_Handler.Init.ContinuousClock=FMC_CONTINUOUS_CLOCK_SYNC_ASYNC;

//FMC 读时序控制寄存器
FMC_ReadWriteTim.AddressSetupTime=0x011;   //地址建立时间为 17 个 HCLK
FMC_ReadWriteTim.AddressHoldTime=0x00;
FMC_ReadWriteTim.DataSetupTime=0x55;     //数据保存时间(DATAST)为 85 个 HCLK
FMC_ReadWriteTim.AccessMode=FMC_ACCESS_MODE_A; //模式 A

//FMC 写时序控制寄存器
FMC_WriteTim.AddressSetupTime=0x15; //地址建立时间(ADDSET)为 21 个 HCLK
FMC_WriteTim.AddressHoldTime=0x00;
FMC_WriteTim.DataSetupTime=0x015;     //数据保存时间(DATAST)为 21 个 HCLK
FMC_WriteTim.AccessMode=FMC_ACCESS_MODE_A;        //模式 A
HAL_SRAM_Init(&SRAM_Handler,&FMC_ReadWriteTim,&FMC_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
        FMC_Bank1E->BWTR[0]&=~(0XF<<0);      //地址建立时间(ADDSET)清零
        FMC_Bank1E->BWTR[0]&=~(0XF<<8);      //数据保存时间清零
        FMC_Bank1E->BWTR[0]|=5<<0;   //地址建立时间(ADDSET)为 5 个 HCLK =21ns
        FMC_Bank1E->BWTR[0]|=5<<8;//数据保存时间(DATAST)  为 21ns
    }
    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 FMC 相关 IO 进行初始化，然后是 FMC 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，这个初始化函数针对多款不同的驱动 IC 执行初始化操作，这样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。

这里还要提醒大家，在 LCD_Init 函数中有如下一行代码：

```
LCD_MPU_Config();  //使能 MPU 保护 LCD 区域
```

这行代码的作用是调用函数 LCD_MPU_Config 使能 MPU 保护 LCD 区域，而函数 LCD_MPU_Config 定义的内容实际上是我们上一章给大家讲解的使能 MPU 保护 LCD 区域。这里我们之所以直接在 LCD 程序中加入 MPU 保护，是因为方便大家在移植 LCD 相关代码到自己的工程中的时候不会因为没有引入 MPU 相关配置而导致 LCD 无法正常工作。

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];
    Cache_Enable();                //打开 L1-Cache
    HAL_Init();                    //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);  //设置时钟,216Mhz
    delay_init(216);               //延时初始化
    uart_init(115200);             //串口初始化
    LED_Init();                    //初始化 LED
    LCD_Init();                    //初始化 LCD
    POINT_COLOR=RED;
    sprintf((char*)lcd_id,"LCD ID:%04X",lcddev.id);//将 LCD ID 打印到 lcd_id 数组。
    while(1)
```

```
    {
        switch(x)
        {
            case 0:LCD_Clear(WHITE);break;
            ……//此处省略部分代码
            case 11:LCD_Clear(BROWN);break;
        }
        POINT_COLOR=RED;
        LCD_ShowString(10,40,260,32,32,"Apollo STM32F4/F7");
        LCD_ShowString(10,80,240,24,24,"TFTLCD TEST");
        LCD_ShowString(10,110,240,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(10,130,240,16,16,lcd_id);        //显示 LCD ID

        LCD_ShowString(10,150,240,12,12,"2016/7/11");
        x++;
        if(x==12)x=0;
        LED0_Toggle;
        delay_ms(1000);
    }
}
```

该部分代码将显示一些固定的字符，字体大小包括 32*16、24*12、16*8 和 12*6 等四种，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，请百度学习。

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

将程序下载到阿波罗 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：

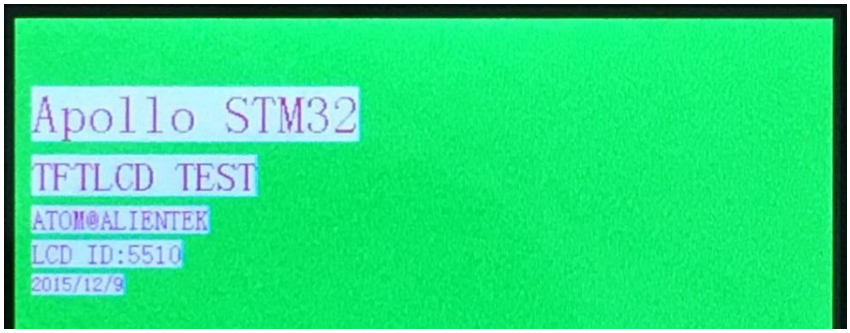


图 18.4.1 TFTLCD 显示效果图

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。

18.5 STM32CubeMX 配置 FMC（SRAM）

当大家了解了 FMC 的基本工作原理，那么使用 STM32CubeMX 配置 FMC 相关参数就会非常简单。如果大家对 FMC 没有理解，请仔细看教程学习。这里我们们不再详细讲解每个配置项的含义。使用 STM32CubeMX 配置 FMC 的一般步骤为：

① 进入 Pinout->FMC 配置栏，配置 FMC 基本参数。根据前面的讲解，这里我们使用的是 BANK1 的第一个分区 NE1，同时吧 LCD 作为 SRAM 使用，19 位地址线，16 位数据线。配置参数如下图 18.5.1 所示：

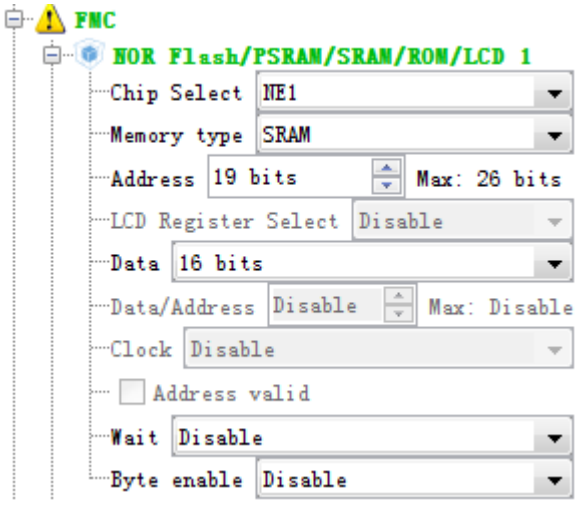


图 18.5.1 FMC 配置参数

② 点击 Configuration->FMC 进入 FMC 配置界面，在 NOR/SRAM 1 选项卡之下配置相关参数。这些参数的含义这里我们不累赘，在 18.1 小节讲解 HAL_SRAM_Init 函数的时候都有讲解。配置方法如下图 18.5.2 所示：

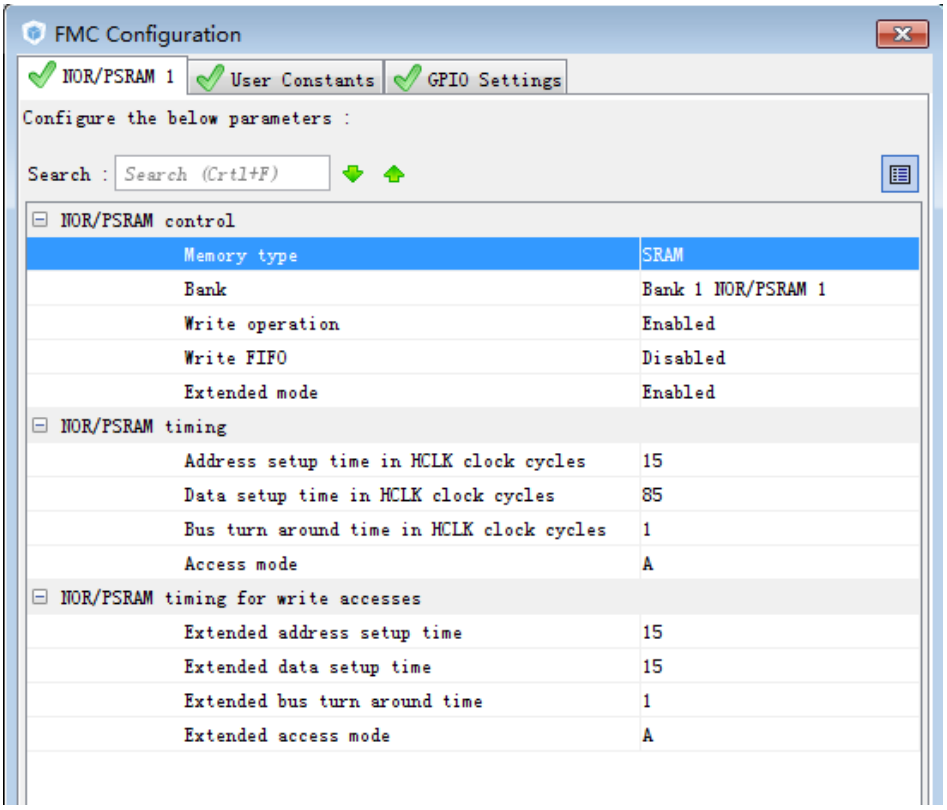


图 18.5.2 FMC Configuration 配置界面 NOR/PSRAM1 选项卡

在该配置界面，点击右边的 GPIO Settigns 选项卡，还可以配置相关 IO 口的信息。

经过上面配置步骤，我们就可以生成相应的初始化代码，大家生成后和本章实验工程对比学习。

第十九章 SDRAM 实验

STM32F767IGT6 自带了 512K 字节的 SRAM，对一般应用来说，已经足够了，不过在一些对内存要求高的场合，STM32F767 自带的这些内存就不够用了。比如使用 LTDC 驱动 RGB 屏、跑算法或者跑 GUI 等，就可能不太够用，所以阿波罗 STM32F767 开发板板载了一颗 32M 字节容量的 SDRAM 芯片：W9825G6KH，满足大内存使用的需求。

本章，我们将使用 STM32F767 来驱动 W9825G6KH，实现对 W9825G6KH 的访问控制，并测试其容量。本章分为如下几个部分：

19.1 SDRAM 简介
19.2 硬件设计
19.3 软件设计
19.4 下载验证
19.5 STM32CubeMX 配置 FMC（SDRAM）

19.1 SDRAM 简介

本章我们将通过 STM32F767 的 FMC 接口，来驱动 W9825G6KH 这颗 SDRAM 芯片，本节我们将介绍 SDRAM 相关知识点，包括：1，SDRAM 简介；2，FMC SDRAM 接口简介；

19.1.1 SDRAM 简介

SDRAM，英文名是：Synchronous Dynamic Random Access Memory，即同步动态随机存储器，相较于 SRAM（静态存储器），SDRAM 具有：容量大和价格便宜的特点。STM32F767 支持 SDRAM，因此，我们可以外挂 SDRAM，从而大大降低外扩内存的成本。

阿波罗板载的 SDRAM 型号为：W9825G6KH，其内部结构框图如图 19.1.1.1 所示：

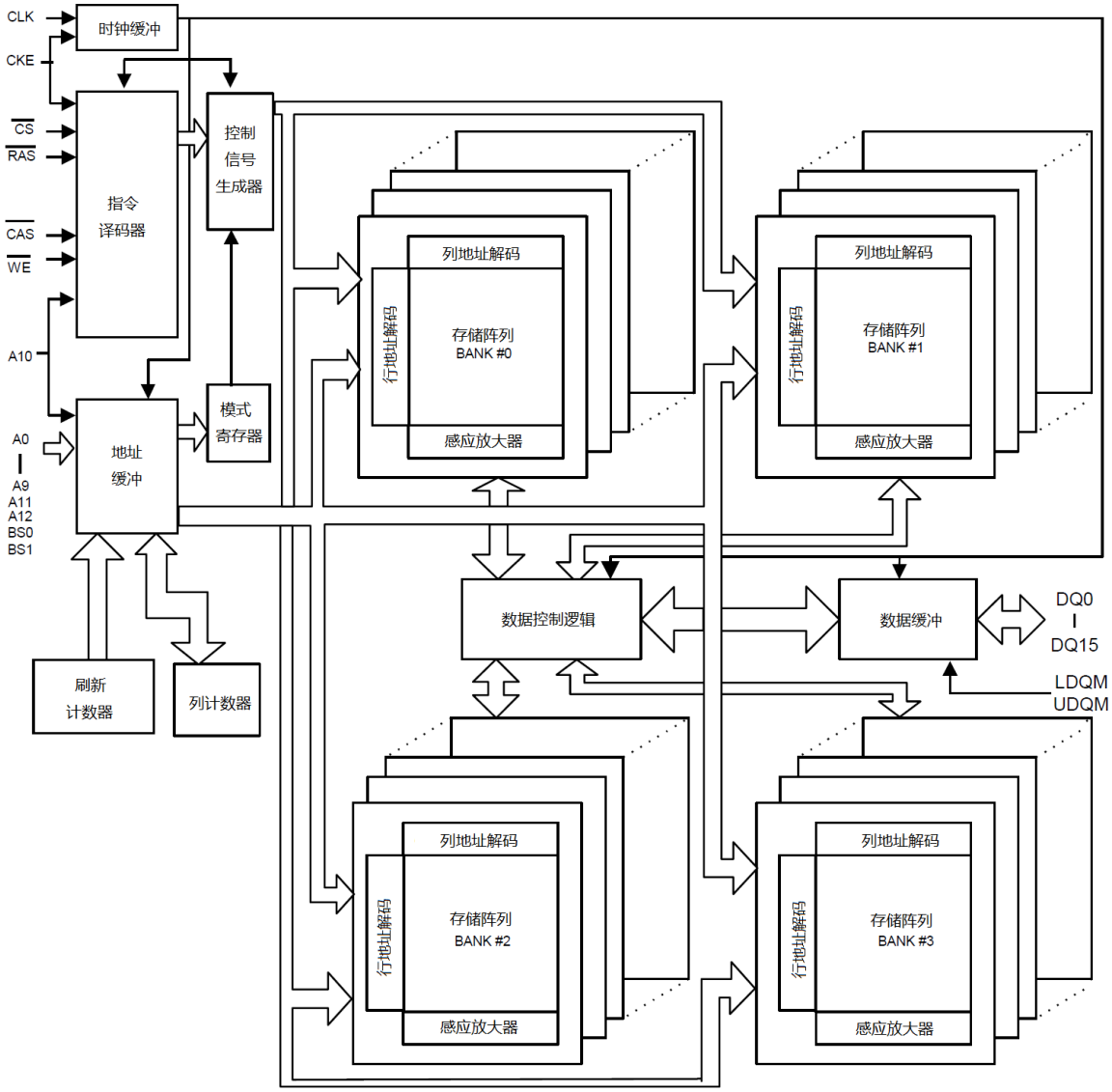


图 19.1.1.1 W9825G6KH 内部结构框图

接下来，我们结合图 19.1.1.1，对 SDRAM 的几个重要知识点进行介绍。

（1）SDRAM 信号线

SDRAM 的信号线如表 19.1.1.1 所示：

信号线	说明
CLK	时钟信号，在该时钟的上升沿采集输入信号
CKE	时钟使能，禁止时钟时，SDRAM 会进入自刷新模式
CS#	片选信号，低电平有效
RAS#	行地址选通信号，低电平时，表示行地址
CAS#	列地址选通信号，低电平时，表示列地址
WE#	写使能信号，低电平有效
A0~A12	地址线（行/列）
BS0，BS1	BANK 地址线
DQ0~15	数据线
LDQM,UDQM	数据掩码，表示 DQ 的有效部分

表 19.1.1.1 SDRAM 信号线

（2）存储单元

SDRAM 的存储单元（称之为：BANK）是以阵列的形式排列的，如图 19.1.1.1 所示。每个存储单元的结构示意图，如图 19.1.1.2 所示：

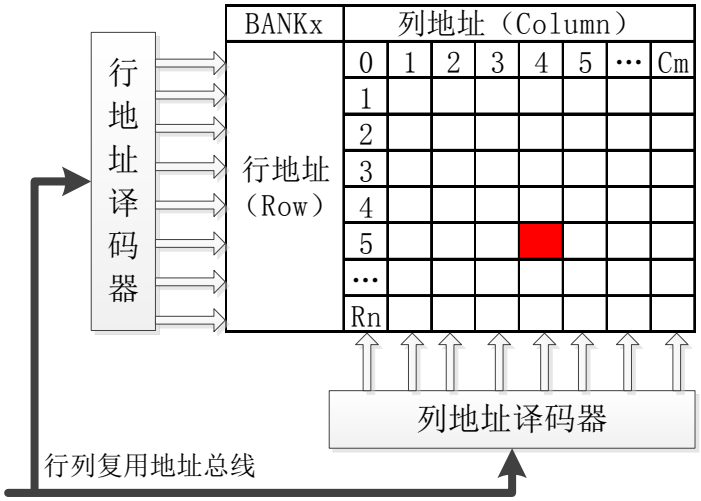


图 19.1.1.2 SDRAM BANK 结构示意图

对于这个存储阵列，我们可以将其看成是一个表格，只需要给定行地址和列地址，就可以确定其唯一位置，这就是 SDRAM 寻址的基本原理。而一个 SDRAM 芯片内部，一般又有 4 个这样的存储单元（BANK），所以，在 SDRAM 内部寻址的时候，先指定 BANK 号和行地址，然后再指定列地址，就可以查找到目标地址。

SDRAM 的存储结构示意图，如图 19.1.1.3 所示，寻址的时候，首先 RAS 信号为低电平，选通行地址，地址线 A0~A12 所表示的地址，会被传输并锁存到行地址译码器里面，最为行地址，同时 BANK 地址线上面的 BS0，BS1 所表示的 BANK 地址，也会被锁存，选中对应的 BANK，然后，CAS 信号为低电平，选通列地址，地址线 A0~A12 所表示的地址，会被传输并锁存到列地址译码器里面，作为列地址，这样，就完成了一次寻址。

W9825G6KH 的存储结构为：行地址：8192 个；列地址：512 个；BANK 数：4 个；位宽：16 位；这样，整个芯片的容量为：8192*512*4*16=32M 字节。

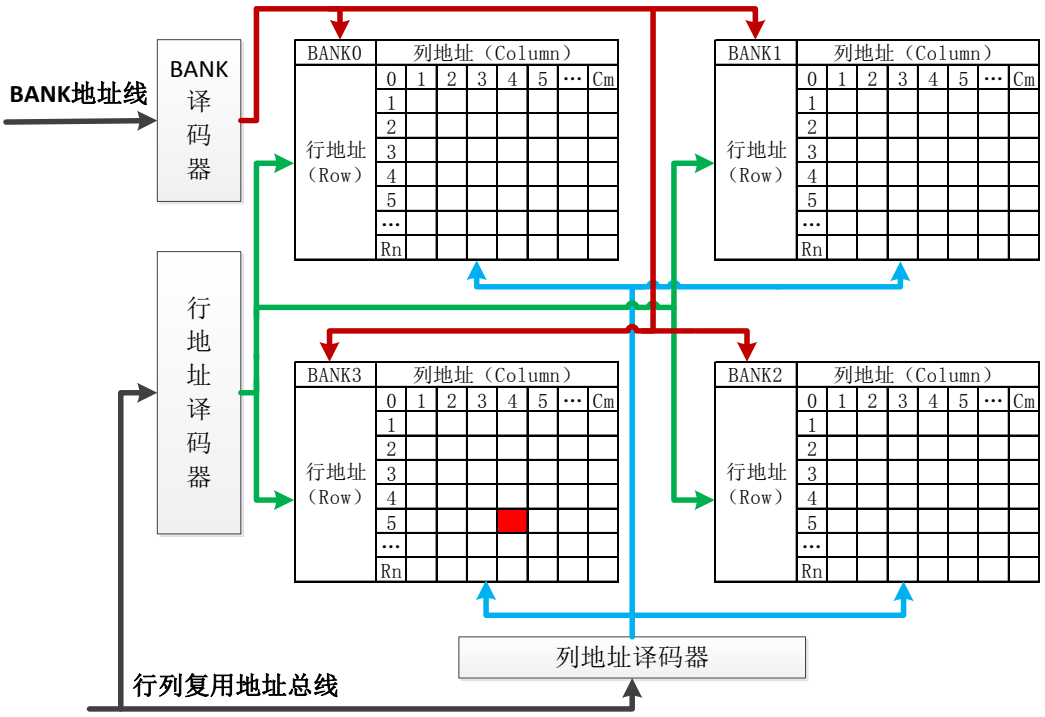


图 19.1.1.3 SDRAM 存储结构图

（3）数据传输

在完成寻址以后，数据线 DQ0~DQ15 上面的数据会通过图 19.1.1.1 中所示的数据控制逻辑写入（或读出）存储阵列。

特别注意：因为 SDRAM 的位宽，可以达到 32 位，也就是最多有 32 条数据线，在实际使用的时候，我们可能会以：8 位、16 位、24 位和 32 位等宽度来读写数据，这样的话，并不是每条数据线，都会被使用到，未被用到的数据线上面的数据，必须被忽略，这个时候就需要用到数据掩码（DQM）线来控制了，每一个数据掩码线，对应 8 个位的数据，低电平表示对应数据位有效，高电平表示对应数据位无效。

以 W9825G6KH 为例，假设以 8 位数据访问，我们只需要 DQ0~DQ7 的数据，而 DQ8~DQ15 的数据需要忽略，此时，我们只需要设置 LDQM 为低电平，UDQM 为高电平，就可以了。

（4）控制命令

SDRAM 的驱动需要用到一些命令，我们列出几个常用的命令给大家做讲解，如表 19.1.1.2 所示：

命令	CS	RAS	CAS	WE	DQM	ADDR	DQ
NO-Operation	L	H	H	H	X	X	X
Active	L	L	H	H	X	Bank/Row	X
Read	L	H	L	H	L/H	Bank/Col	DATA
Write	L	H	L	L	L/H	Bank/Col	DATA
Precharge	L	L	H	L	X	A10=H/L	X
Refresh	L	L	L	H	X	X	X
Mode Register Set	L	L	L	L	X	MODE	X
Burst Stop	L	H	H	L	X	X	DATA

表 19.1.1.2 SDRAM 控制命令

1， NO-Operation

NO-Operation，即空操作命令，用于选中 SDRAM，防止 SDRAM 接受错误的命令，为接下来的命令发送做准备。

2， Active

Active，即激活命令，该命令必须在读写操作之前被发送，用于设置所需要的 Bank 和行地址（同时设置这 2 个地址），Bank 地址由 BS0，BS1（也写作 BA0，BA1，下同）指定，行地址由 A0~A12 指定，时序图如图 19.1.1.4 所示：

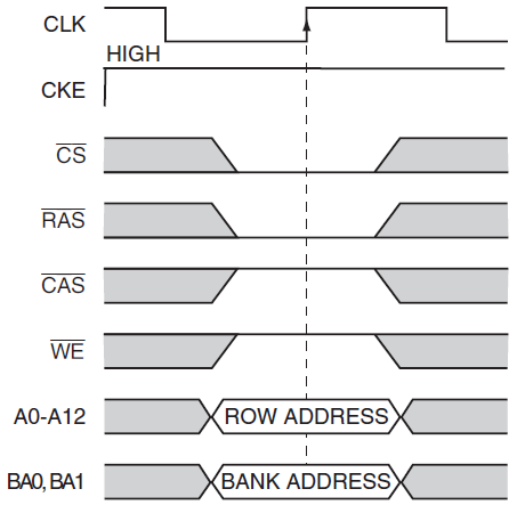


图 19.1.1.4 激活命令时序图

3， Read/Write

Read/Write，即读/写命令，在发送完激活命令后，再发送列地址就可以完成对 SDRAM 的寻址，并进行读写操作了，读/写命令和列地址的发送，是通过一次传输完成的，如图 19.1.1.5 所示：

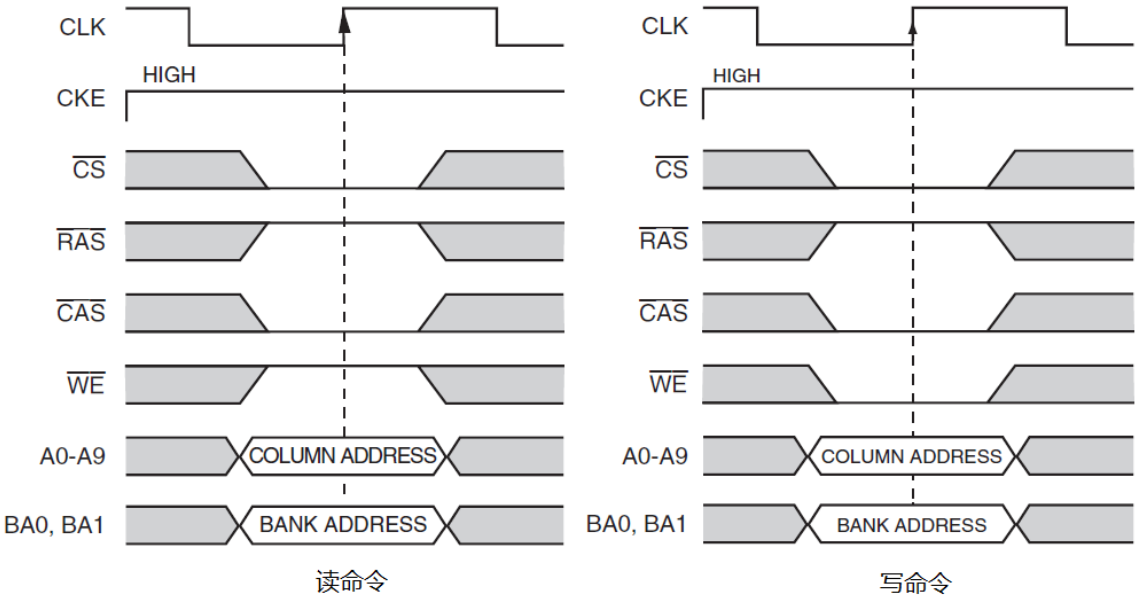


图 19.1.1.5 读/写命令时序图

列地址由 A0~A9 指定，WE 信号控制读/写命令，高电平表示读命令，低电平表示写命令，各条信号线的状态，在 CLK 的上升沿被锁存到芯片内部。

4，Precharge

Precharge，即预充电指令，用于关闭 Bank 中所打开的行地址。由于 SDRAM 的寻址具体独占性，所以在进行完读写操作后，如果要对同一 Bank 的另一行进行寻址，就要将原来有效（打开）的行关闭，重新发送行/列地址。Bank 关闭现有行，准备打开新行的操作就叫做预充电（Precharge）。

预充电命令时序，如图 19.1.1.6 所示：

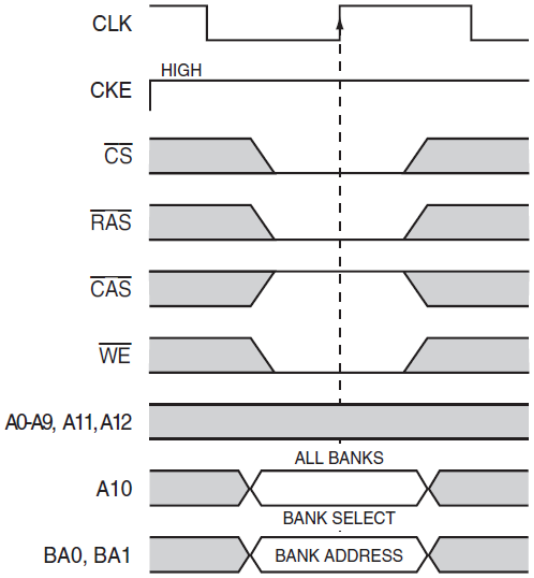


图 19.1.1.6 预充电命令时序图

预充电命令可以通过独立的命令发送，也可以在每次发送读/写命令的时候，使用地址线 A10，来设置自动预充电。在发送读/写命令的时候，当 A10=1，则使能所有 Bank 的预充电，在读/写操作完成后，自动进行预充电。这样，下次读/写操作之前，就不需要再发预充电命令了，从而提高读/写速度。

5，Refresh

Refresh，即刷新命令，用于刷新一行数据。SDRAM 里面存储的数据，需要不断的进行刷新操作才能保留住，因此刷新命令对于 SDRAM 来说，尤为重要。预充电命令和刷新命令，都可以实现对 SDRAM 数据的刷新，不过预充电仅对当前打开的行有效（仅刷新当前行），而刷新命令，则可以依次对所有的行进行刷新操作。

总共有两种刷新操作：自动刷新（Auto Refresh）和自我刷新（Self Refresh），在发送 Refresh 命令时，如果 CKE 有效（高电平），则使用自动刷新模式，否则使用自我刷新模式。不论是何种刷新方式，都不需要外部提供行地址信息，因为这是一个内部的自动操作。

自动刷新：SDRAM 内部有一个行地址生成器（也称刷新计数器）用来自动的依次生成要刷新的行地址。由于刷新是针对一行中的所有存储体进行，所以无需列寻址。刷新涉及到所有 Bank，因此在刷新过程中，所有 Bank 都停止工作，而每次刷新所占用的时间为 9 个时钟周期（PC133 标准），之后就可进入正常的工作状态，也就是说在这 9 个时钟期间内，所有工作指令只能等待而无法执行。刷新操作必须不停的执行，完成一次所有行的刷新所需要的时间，称为刷新周期，一般为 64ms。显然，刷新操作肯定会对 SDRAM 的性能造成影响，但这是没办法的事情，也是 DRAM 相对于 SRAM（静态内存，无需刷新仍能保留数据）取得成本优势的同时所付出的代价。

自我刷新：主要用于休眠模式低功耗状态下的数据保存，在发出自动刷新命令时，将 CKE 置于无效状态（低电平），就进入了自我刷新模式，此时不再依靠系统时钟工作，而是根据内部

的时钟进行刷新操作。在自我刷新期间除了 CKE 之外的所有外部信号都是无效的（无需外部提供刷新指令），只有重新使 CKE 有效（高电平）才能退出自刷新模式并进入正常操作状态。

6，Mode Register Set

Mode Register Set，即设置模式寄存器。SDRAM 芯片内部有一个逻辑控制单元，控制单元的相关参数由模式寄存器提供，我们通过设置模式寄存器命令，来完成对模式寄存器的设置，这个命令在每次对 SDRAM 进行初始化的时候，都需要用到。

发送该命令时，通过地址线来传输模式寄存器的值，W9825G6KH 的模式寄存器描述如图 19.1.1.7 所示：

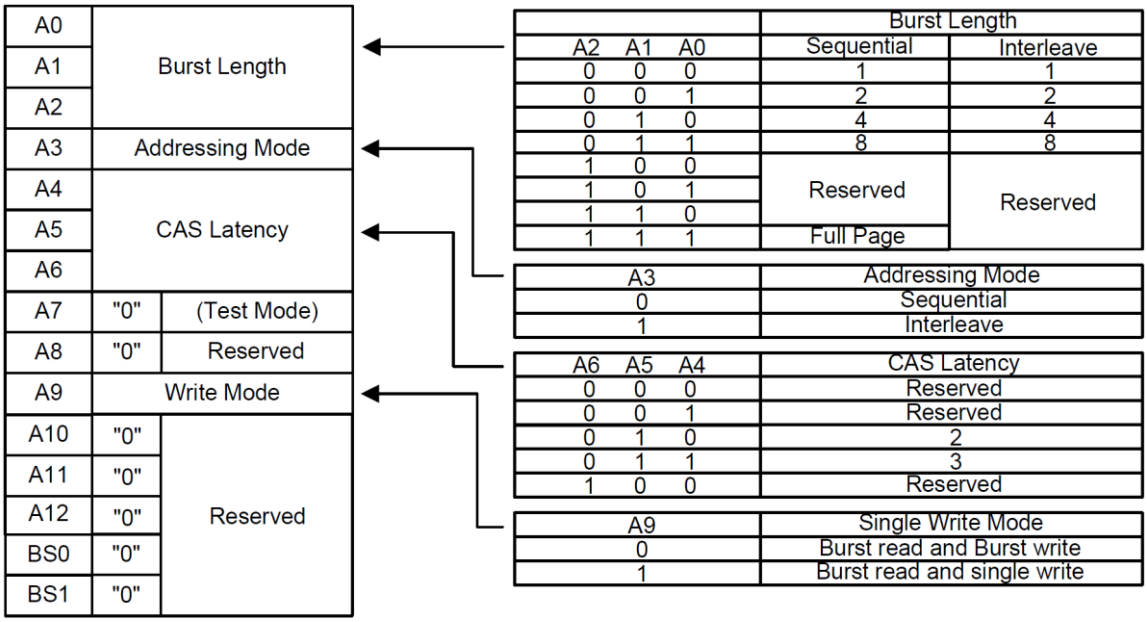


图 19.1.1.7 W9825G6KH 的模式寄存器

由图可知，模式寄存器的配置分为几个部分：

Burst Length，即突发长度（简称 BL），通过 A0~A2 设置，是指在同一行中相邻的存储单元连续进行数据传输的方式，连续传输所涉及到存储单元（列）的数量就是突发长度。

前面我们说的读/写操作，都是一次对一个存储单元进行寻址，如果要连续读/写就还要对当前存储单元的下一个单元进行寻址，也就是要不断的发送列地址与读/写命令（行地址不变，所以不用再对行寻址）。虽然由于读/写延迟相同可以让数据的传输在 I/O 端是连续的，但它占用了大量的内存控制资源，在数据进行连续传输时无法输入新的命令，效率很低。

为此，人们开发了突发传输技术，只要指定起始列地址与突发长度，内存就会依次地自动对后面相应数量的存储单元进行读/写操作而不再需要控制器连续地提供列地址。这样，除了第一个数据的传输需要若干个周期外，其后每个数据只需一个周期的即可获得。

非突发连续读取模式：不采用突发传输而是依次单独寻址，此时可等效于 BL=1。虽然可以让数据是连续的传输，但每次都要发送列地址与命令信息，控制资源占用极大。突发连续读取模式：只要指定起始列地址与突发长度，寻址与数据的读取自动进行，而只要控制好两段突发读取命令的间隔周期（与 BL 相同）即可做到连续的突发传输。 至于 BL 的数值，也是不能随便设或在数据进行传输前临时决定，而是在初始化的时候，通过模式寄存器设置命令，进行设置。目前可用的选项是 1、2、4、8、全页（Full Page），常见的设定是 4 和 8。若传输长度小于突发长度，则需要发送 Burst Stop（停止突发）命令，结束突发传输。

Addressing Mode，即突发访问的地址模式，通过 A3 设置，可以设置为：Sequential（顺序）或 Interleave（交错）。顺序方式，地址连续访问，而交错模式则地址是乱序的，一般选择连续

模式。

CAS Latency，即列地址选通延迟（简称 CL）。在读命令（同时发送列地址）发送完之后，需要等待几个时钟周期，DQ 数据线上的数据，才会有效，这个延迟时间，就叫 CL，一般设置为 2/3 个时钟周期，如图 19.1.1.8 所示：

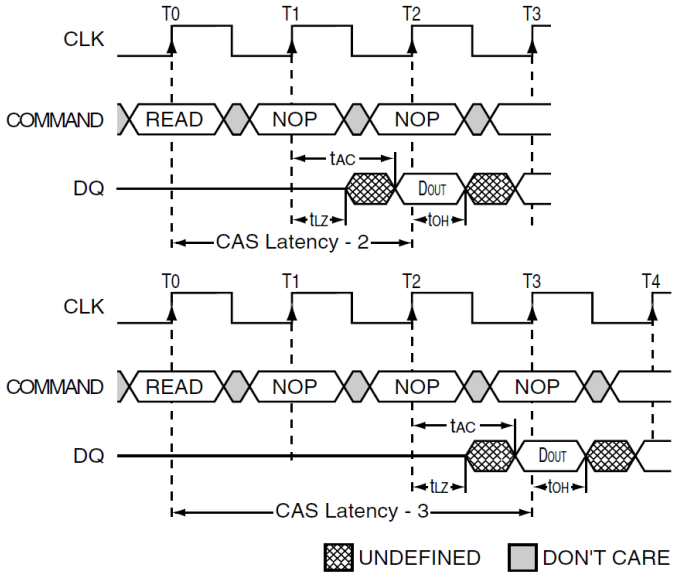


图 19.1.1.8 CAS 延迟（2/3）

特别注意：列地址选通延迟（CL），仅在读命令的时候有效果，在写命令的时候，并不需要这个延迟。

Write Mode，即写模式，用于设置单次写的模式，可以选择突发写入或者单次写入。

（五）初始化

SDRAM 上电后，必须进行初始化，才可以正常使用。SDRAM 初始化时序图如图 19.1.1.9 所示：

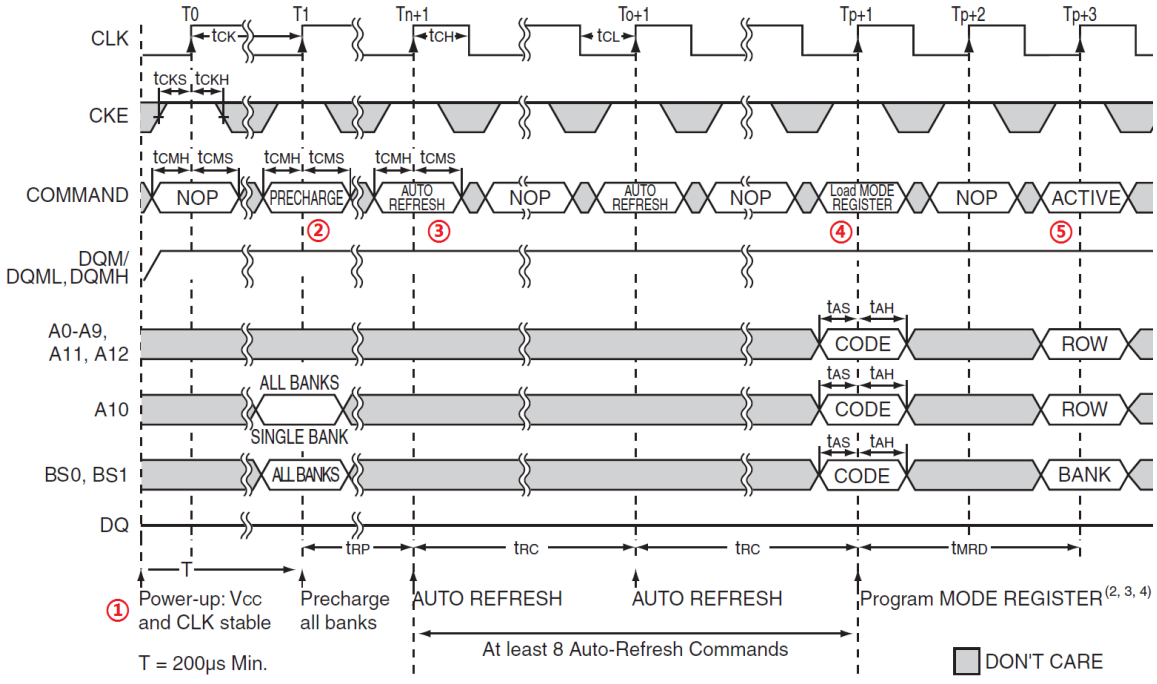


图 19.1.1.9 SDRAM 初始化时序图

初始化过程分为五步：

① 上电

此步，给 SDRAM 供电，使能 CLK 时钟，并发送 NOP（No Operation 命令），注意，上电后，要等待最少 200us，再发送其他指令。

② 发送预充电命令

第二步，就是发送预充电命令，给所有 Bank 预充电。

③ 发送自动刷新命令

这一步，至少要发送发送 8 次自刷新命令， 每一个自刷新命令之间的间隔时间为 tRC。

④ 设置模式寄存器

这一步，发送模式寄存器的值，配置 SDRAM 的工作参数。配置完成后，需要等待 tMRD（也叫 tRSC），使模式寄存器的配置生效，才能发送其他命令。

⑤ 完成

经过前面四步的操作，SDRAM 的初始化就完成了，接下来，就可以发送激活命令和读/写命令，进行数据的读/写了。

这里提到的 tRC、tMRD 和 tRSC 见 SDRAM 的芯片数据手册。

（六）写操作

在完成对 SDRAM 的初始化之后，我们就可以对 SDRAM 进行读写操作了，首先，我们来看写操作，时序图如图 19.1.1.10 所示：

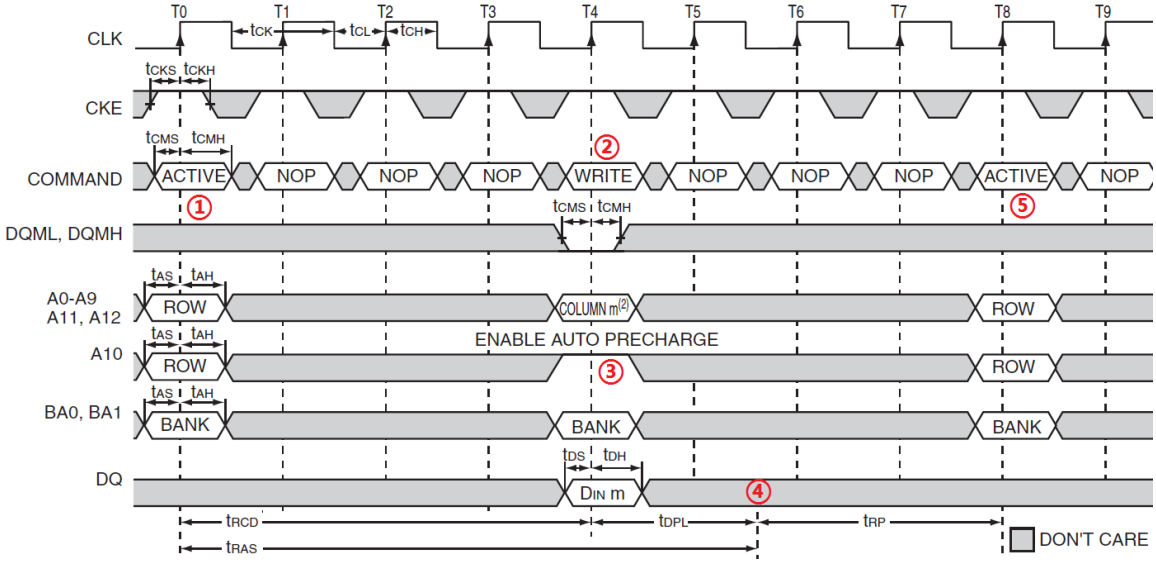


图 19.1.1.10 SDRAM 写时序图（自动预充电）

SDRAM 的写流程如下：

① 发送激活命令

此命令同时设置行地址和 BANK 地址，发送该命令后，需要等待 tRCD 时间，才可以发送写命令。

② 发送写命令

在发送完激活命令，并等待 tRCD 后，发送写命令，该命令同时设置列地址，完成对 SDRAM 的寻址。同时，将数据通过 DQ 数据线，存入 SDRAM。

③ 使能自动预充电

在发送写命令的同时，拉高 A10 地址线，使能自动预充电，以提高读写效率。

④ 执行预充电

预充电在发送激活命令的 tRAS 时间后启动，并且需要等待 tRP 时间，来完成。

⑤ 完成一次数据写入

最后，发送第二个激活命令，启动下一次数据传输。这样，就完成了一次数据的写入。

（七）读操作

前面介绍了 SDRAM 的写操作，接下来我们看读操作，读操作时序，如图 19.1.1.11 所示：

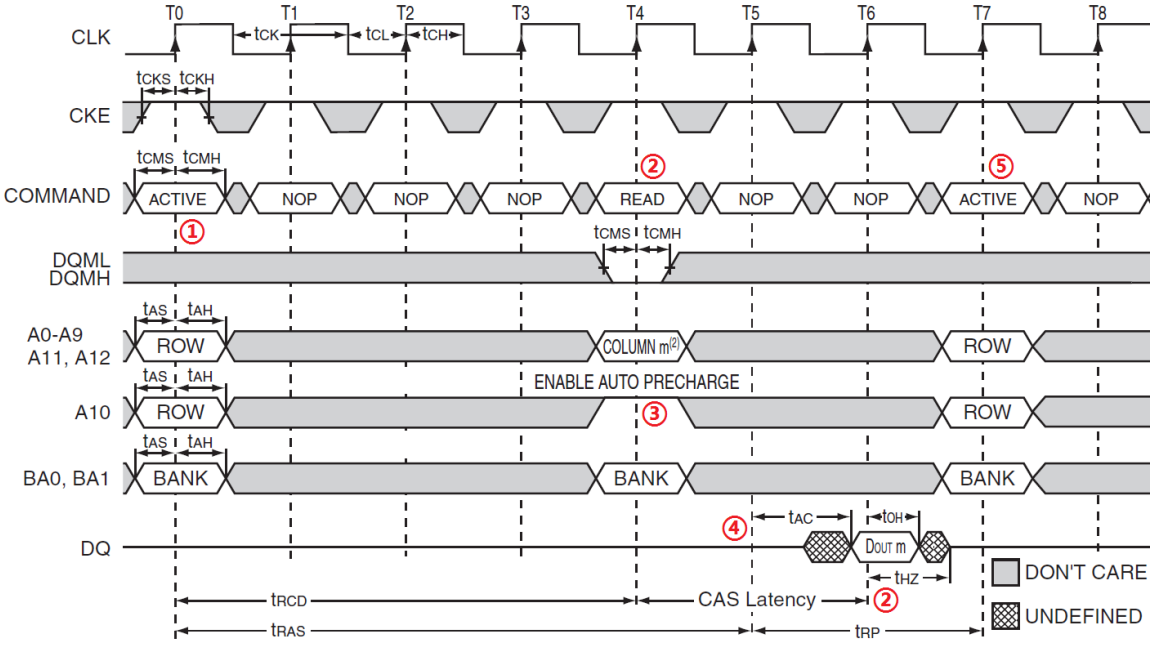


图 19.1.1.11 SDRAM 读时序图（自动预充电）

SDRAM 的读流程如下：

① 发送激活命令

此命令同时设置行地址和 BANK 地址，发送该命令后，需要等待 tRCD 时间，才可以发送读命令。

② 发送写命令

在发送完激活命令，并等待 tRCD 后，发送读命令，该命令同时设置列地址，完成对 SDRAM 的寻址。读操作还有一个 CL 延迟（CAS Latency），所以需要等待给定的 CL 延迟（2 个或 3 个 CLK）后，再从 DQ 数据线上读取数据。

③ 使能自动预充电

在发送读命令的同时，拉高 A10 地址线，使能自动预充电，以提高读写效率。

④ 执行预充电

预充电在发送激活命令的 tRAS 时间后启动，并且需要等待 tRP 时间，来完成。

⑤ 完成一次数据写入

最后，发送第二个激活命令，启动下一次数据传输。这样，就完成了一次数据的读取。

SDRAM 的简介，就给大家介绍到这里，以上，tRCD、tRAS 和 tRP 等时间参数，见 SDRAM 的数据手册，且在后续配置 FMC 的时候，需要用到。

19.1.2 FMC SDRAM 接口简介

在上一章，我们对 STM32F767 的 FMC 接口进行了简介，并利用 FMC 接口，来驱动 MCU 屏，本章，我们将介绍如何利用 FMC 接口，驱动 SDRAM。STM32F767 FMC 接口的 SDRAM 控制器，具有如下特点：

- 两个 SDRAM 存储区域，可独立配置
- 支持 8 位、16 位和 32 位数据总线宽度
- 支持 13 位行地址，11 位列地址，4 个内部存储区域：4x16Mx32bit (256MB)、4x16Mx16bit(128 MB)、4x16Mx8bit (64 MB)
- 支持字、半字和字节访问
- 自动进行行和存储区域边界管理
- 多存储区域乒乓访问
- 可编程时序参数
- 支持自动刷新操作，可编程刷新速率
- 自刷新模式
- 读 FIFO 可缓存，支持 6 行 x32 位深度（6 x14 位地址标记）

通过 19.1.1 的介绍，我们对 SDRAM 已经有了一个比较深入的了解，包括接线、命令、初始化流程和读写流程等，接下来，我们介绍一些配置 FMC SDRAM 控制器需要用到的几个寄存器。

首先，我们介绍 SDRAM 的控制寄存器：FMC_SDCRx，x=1/2，该寄存器各位描述如图 19.1.2.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	RPIPE		RBURST	SDCLK		WP	CAS		NB	MWID		NR		NC	
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 19.1.2.1 FMC_SDCRx 寄存器各位描述

该寄存器只有低 15 位有效，且都需要进行配置：

NC：这两个位定义列地址的位数（00~11，表示 8~11 位），W9825G6KH 有 9 位列地址，所以，这里应该设置为 01。

NR：这两个位定义行地址的位数（00~10，表示 11~13 位），W9825G6KH 有 13 位行地址，所以，这里设置为 10。

MWID：这两个位定义存储器数据总线宽度（00~10，表示 8~32 位），W9825G6KH 数据位宽为 16 位，所以，这里设置为 01。

NB：该位用于设置 SDRAM 内部存储区域（BANK）数量（0=2 个，1=4 个），W9825G6KH 内部有 4 个 BANK，所以，这里设置为：1。

CAS：这两个位可设置 SDRAM 的 CAS 延迟，按存储器时钟周期计（01~11，表示 1~3 个）。W9825G6KH 可以设置为 2，也可以设置为 3，我们设置为 11。

WP：该位用于写保护设置（0=写使能，1=写保护），我们需要用到写操作，所以这里设置为 1 即可。

SDCLK：这两个位用于配置 SDRAM 的时钟（10=HCLK/2，11=HCLK/3），需要在禁止 SDRAM 时钟的前提下配置。W9825G6KH 最快可以到 200M（@CL=3），为了较快的速度，我们设置为 10。

RBURST：此位用于使能突发读模式（0=禁止，1=使能）。这里我们设置为 1，使能突发读。

RPIPE：这两个位可定义在 CAS 延迟后延后多少个 HCLK 时钟周期读取数据（00~10，表示 0~2 个）。这里，我们设置为 00 即可。

接下来，我们介绍 SDRAM 的时序寄存器：FMC_SDTRx，x=1/2，该寄存器各位描述如图 19.1.2.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	TRCD				TRP				TWR			
				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
TRC				TRAS				TXSR				TMRD			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 19.1.2.2 FMC_SDTRx 寄存器各位描述

该寄存器用于控制 SDRAM 的时序，非常重要，接下来我们分别介绍各个参数：

TMRD：这四个位定义加载模式寄存器命令和激活或刷新命令之间的延迟，这个参数就是 SDRAM 数据手册里面的 tMRD 或 tRSC 参数，W9825G6KH 的 tRSC 值为 2 个时钟，所以我们设置为 1 即可（2 个时钟周期，这里的时钟周期是指 SDRAM 的时钟周期，下同）。

TXSR：这四个位定义从发出自刷新命令到发出激活命令之间的延迟。W9825G6KH 的这个时间为 72ns，我们设置 STM32F767 的时钟频率为 216Mhz，那么一个 SDRAM 的时钟频率为 108M，一个周期为 9.3ns，设置 TXSR 为 7，即 8 个时钟周期即可。

TRAS：这四个位用于设置自刷新周期。W9825G6KH 的自刷新周期为 60ns，我们设置 TRAS 为 6，即 7 个时钟周期即可。

TRC：这四个位定义刷新命令和激活命令之间的延迟，以及两个相邻刷新命令之间的延迟。W9825G6KH 的这个时间同样是 60ns，我们设置 TRC 为 6，即 7 个时钟周期即可。

TWR：这四个位定义写命令和预充电命令之间的延迟。W9825G6KH 的这个时间为 2 个时钟周期，所以，我们设置 TWR=1 即可。

TRP：这四个位定义预充电命令与其它命令之间的延迟。W9825G6KH 的这个时间为 15ns，所以，我们设置 TRP=1，即 2 个时钟周期（18.6ns）。

TRCD：这四个位定义激活命令与读/写命令之间的延迟。W9825G6KH 的这个时间为 15ns，所以，我们设置 TRP=1，即 2 个时钟周期（18.6ns）。

接下来，我们介绍 SDRAM 的命令模式寄存器：FMC_SDCMR，该寄存器各位描述如图 19.1.2.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	MRD					
										rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MRD							NRFS				CTB1	CTB2	MODE		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 19.1.2.3 FMC_SDCMR 寄存器各位描述

该寄存器用于发送控制 SDRAM 的命令，以及 SDRAM 控制器的工作模式时序，非常重要，接下来我们分别介绍各个参数：

MODE：这三个位定义发送到 SDRAM 存储器的命令。000：正常模式；**001：时钟配置使能；010：预充电所有存储区；011：自刷新命令；100：配置模式寄存器；**101：自刷新命令；110：掉电命令；111：保留。加粗部分的命令，我们配置的时候需要用到。

CTB2/CTB1：这两个位用于指定命令所发送的目标存储器，因为 SDRAM 控制器可以外挂 2 个 SDRAM，发送命令的时候，需要通过 CTB1/CTB2 指定命令发送给哪个存储器。我们使用的是第一个存储器（SDNE0），所以设置 CTB1 即可。

NRFS：这四个位定义在 MODE=011 时，所发出的连续自刷新命令的个数。0000~1110，表示 1~15 个自刷新命令。W9825G6KH 在初始化的时候，至少需要连续发送 8 个自刷新命令。

MRD：这十三个位，定义 SDRAM 模式寄存器的内容（通过地址线发送），在 MODE=100 的时候，需要配置。

接下来，我们介绍 SDRAM 的刷新定时器寄存器：FMC_SDRTR，该寄存器各位描述如图 19.1.2.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	REIE	COUNT													CRE
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	w

图 19.1.2.4 FMC_SDRTR 寄存器各位描述

该寄存器通过配置刷新定时器计数值来设置刷新循环之间的刷新速率，按 SDRAM 的时钟周期计数。计算公式为：

刷新速率=（COUNT+1）*SDRAM 频率时钟

COUNT=(SDRAM 刷新周期/行数)-20

我们以 W9825G6KH 为例讲解计算过程，W9825G6KH 的刷新周期为 64ms，行数为 8192 行，所以刷新速率为：

刷新速率=64ms/8192=7.81us

而 SDRAM 时钟频率=216Mhz/2=108Mhz（9.26ns），所以 COUNT 的值为：

COUNT=7.81us/9.26ns≈844

而如果 SDRAM 在接受读请求后，出现内部刷新请求，则必须将刷新速率增加 20 个 SDRAM 时钟周期，以获得充足的余量，所以，实际设计的 COUNT 值应该是：COUNT-20=824。所以，我们设置 FMC_SDRTR 的 COUNT=824，就可以完成对该寄存器的配置。

至此，FMC SDRAM 部分的寄存器就介绍完了，关于 FMC SDRAM 控制器的详细介绍，请大家参考《STM32F7 中文参考手册》第 13.7 节。通过以上两个小节的了解，我们可以开始写 SDRAM 的驱动代码了。不过，MDK 并没有将寄存器定义成 FMC_SDCR1/2 的形式，而是定义成：FMC_SDCR[0]/[1]，对应的就是 FMC_SDCR1/2，其他几个寄存器类似，使用的时候注意一下。

阿波罗 STM32F767 核心板板载的 W9825G6KH 芯片挂在 FMC SDRAM 的控制器 1 上面(SDNE0)，其原理图如图 19.1.2.5 所示：

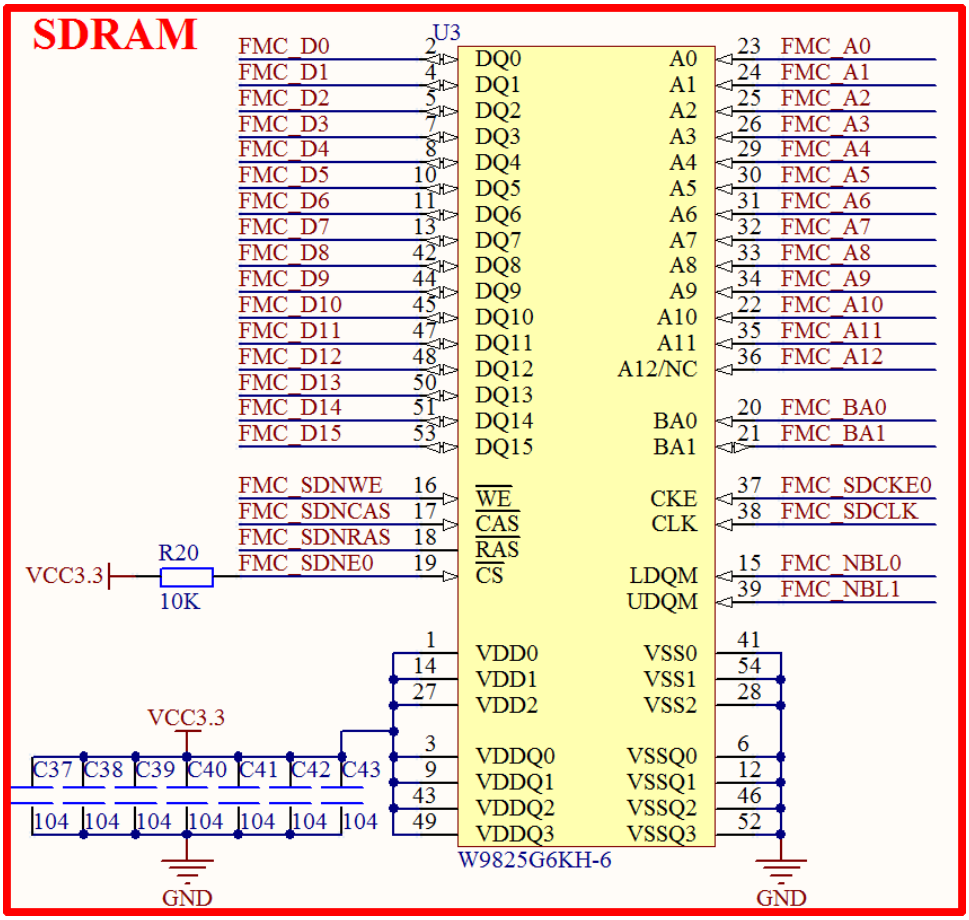


图 19.1.2.5 W9825G6KH 原理图

从原理图可以看出，W9825G6KH 同 STM32F767 的连接关系：

A[0:12]接 FMC_A[0:12]

BA[0:1]接 FMC_BA[0:1]

D[0:15]接 FMC_D[0:15]

CKE 接 FMC_SDCKE0

CLK 接 FMC_SDCLK

UDQM 接 FMC_NBL1

LDQM 接 FMC_NBL0

WE 接 FMC_SDNWE

CAS 接 FMC_SDNCAS

RAS 接 FMC_SDNRAS

CS 接 FMC_SDNE0

最后，我们来看看使用 HAL 库实现对 W9825G6KH 的驱动，需要对 FMC 进行哪些配置。对于 SDRAM 配置，我们要新引入的 HAL 库文件为 stm32f7xx_hal_sdram.c 和 stm32f7xx_hal_sdram.h。具体步骤如下：

1）使能 FMC 时钟，并配置 FMC 相关的 IO 及其时钟使能。

要使用 FMC，当然首先得开启其时钟。然后需要把 FMC_D0~15，FMCA0~12 等相关 IO 口，全部配置为复用输出，并使能各 IO 组的时钟。

使能时钟和初始化 IO 口方法前面已经多次讲解，这里就不累赘，请参考实验代码。

2)初始化 SDRAM 控制参数和时间参数，也就是设置寄存器 FMC_SDCR1 和 FMC_SDTR1。

寄存器 FMC_SDCR1 用来设置 SDRAM 的相关控制参数，比如地址线宽度、CAS 延迟、SDRAM 时钟等。设置该寄存器的 HAL 库函数为 FMC_SDRAM_Init，声明如下：

```
HAL_StatusTypeDef FMC_SDRAM_Init(FMC_SDRAM_TypeDef *Device,
                                 FMC_SDRAM_InitTypeDef *Init);
```

寄存器 FMC_SDTR1 用来设置 SDRAM 时间相关参数，比如自刷新时间、恢复延迟、预充电延迟等。设置该寄存器的 HAL 库函数为 FMC_SDRAM_Timing_Init 函数，声明如下：

```
HAL_StatusTypeDef FMC_SDRAM_Timing_Init(FMC_SDRAM_TypeDef *Device,
                                        FMC_SDRAM_TimingTypeDef *Timing, uint32_t Bank);
```

实际上，HAL 库还提供了共同设置 SDRAM 控制参数和时间参数函数 HAL_SDRAM_Init，该函数会在内部会依次调用函数 FMC_SDRAM_Init 和 FMC_SDRAM_Timing_Init 进行 SDRAM 控制参数初始化和时间参数初始化，所以这里我们着重讲解函数 FMC_SDRAM_Init，声明如下：

```
HAL_StatusTypeDef HAL_SDRAM_Init(SDRAM_HandleTypeDef *hsdram,
                                 FMC_SDRAM_TimingTypeDef *Timing);
```

在讲解该函数之前首先我们要说明一点，和其他外设初始化一样，HAL 库同样提供了 SDRAM 的 MSP 初始化回调函数，函数为 HAL_SDRAM_MspInit ，该函数声明为：

```
void HAL_SDRAM_MspInit(SDRAM_HandleTypeDef *hsdram) ;
```

SDRAM 初始化函数内部会调用该回调函数，用来进行 MCU 相关初始化。对于 MSP 函数这里我们就不做过多讲解。接下来我们继续讲解 SDRAM 初始化函数 HAL_SDRAM_Init，该函数有两个入口参数，一个入口参数是 hsdram，该参数是 SDRAM_HandleTypeDef 结构体指针类型，用来设置 SDRAM 的控制参数，另一个入口参数是 Timing，该参数是 FMC_SDRAM_TimingTypeDef 结构体指针类型，用来设置 SDRAM 的时间相关参数。我们先看看结构体 SDRAM_HandleTypeDef，定义如下。

```
typedef struct
{
  FMC_SDRAM_TypeDef                 *Instance;
  FMC_SDRAM_InitTypeDef              Init;
  __IO HAL_SDRAM_StateTypeDef        State;
  HAL_LockTypeDef                    Lock;
  DMA_HandleTypeDef                  *hdma;
}SDRAM_HandleTypeDef;
```

该结构体有 4 个成员变量，第一个成员变量用来设置 BANK 寄存器基地址，这个我们根据其入口参数有效范围即可找到，这里我们设置为 FMC_SDRAM_DEVICE 即可。第三个和第四个成员变量是 HAL 库使用的一些状态标识参数，最后一个成员变量 hdma 是与 DMA 相关，这里暂不讲解。

接下来我们重点看看第二个成员变量 Init，它是真正的初始化结构体类型变量，结构体 FMC_SDRAM_InitTypeDef 定义如下：

```
typedef struct
{
  uint32_t SDBank;
  uint32_t ColumnBitsNumber;   //列地址数量，FMC_SDCRx 寄存器的 NC 位
  uint32_t RowBitsNumber;      //行地址数量，FMC_SDCRx 寄存器的 NR 位
  uint32_t MemoryDataWidth;    //存储器数据总线宽度，FMC_SDCRx 的 MWID 位
  uint32_t InternalBankNumber; //SDRAM 内部存储区域数量，FMC_SDCRx 的 NB 位
```

```
    uint32_t CASLatency;          //SDRAM 的 CAS 延迟，FMC_SDCRx 的 CAS 位
    uint32_t WriteProtection;     //写保护，FMC_SDCRx 的 WP
    uint32_t SDClockPeriod;       //SDRAM 的时钟，FMC_SDCRx 的 SDCLK 位
    uint32_t ReadBurst;           //使能突发读模式，FMC_SDCRx 的 RBURST 位
    uint32_t ReadPipeDelay;       //读取数据延迟，也就是 FMC_SDCRx 的 RPIPE 位
}FMC_SDRAM_InitTypeDef;
```

成员变量 SDBank 用来设置是使用的 SDRAM 的第几个 BANK，前面说过 SDRAM 有两个独立的 BANK，取值为 FMC_SDRAM_BANK1 或者 FMC_SDRAM_BANK2，我们使用的是 SDRAM 的 BANK1，所以设置为 FMC_SDRAM_BANK1 即可。

其他成员变量都是用来配置 FMC_SDCRx 控制寄存器的相应位的值。

ColumnBitsNumber 用来设置列地址数量，也就是 FMC_SDCRx 寄存器的 NC 位。

RowBitsNumber 用来设置行地址数量，也就 FMC_SDCRx 寄存器的 NR 位。

MemoryDataWidth 用来设置存储器数据总线宽度，也就是 FMC_SDCRx 的 MWID 位。

InternalBankNumber 用来设置 SDRAM 内部存储区域（BANK）数量，也就是 FMC_SDCRx 的 NB 位。

CASLatency 用来设置 SDRAM 的 CAS 延迟。也就是 FMC_SDCRx 的 CAS 位。

WriteProtection 用来设置写保护，也就是 FMC_SDCRx 的 WP。

SDClockPeriod 用来设置 SDRAM 的时钟，也就是 FMC_SDCRx 的 SDCLK 位。

ReadBurst 用来设置使能突发读模式，也就是 FMC_SDCRx 的 RBURST 位。

ReadPipeDelay 用来设置在 CAS 延迟后延后多少个 HCLK 时钟周期读取数据，也就是 FMC_SDCRx 的 RPIPE 位。

讲解完 HAL_SDRAM_Init 函数的第一个入口参数 hsdram，接下来我们看看第二个入口参数 Timing，它是 FMC_SDRAM_TimingTypeDef 结构体指针类型，该结构体主要用来设置寄存器 FMC_SDTRx 的值。该结构体定义如下：

```
typedef struct
{
    uint32_t LoadToActiveDelay;    //加载模式寄存器命令和激活或刷新命令之间的延迟
    uint32_t ExitSelfRefreshDelay; //从发出自刷新命令到发出激活命令之间的延迟
    uint32_t SelfRefreshTime; //自刷新周期
    uint32_t RowCycleDelay;  //刷新和激活命令之间的延迟以及两个相邻刷新命令之间延迟
    uint32_t WriteRecoveryTime;//写命令和预充电命令之间的延迟
    uint32_t RPDelay;      //预充电命令与其它命令之间的延迟
    uint32_t RCDDelay;   //激活命令与读/写命令之间的延迟
}FMC_SDRAM_TimingTypeDef;
```

该结构体一共有 7 个成员变量，这些成员变量都是时间参数，每个而参数与寄存器的四个位对应，取值范围均为 1-16。

成员变量 LoadToActiveDelay 用来设置加载模式寄存器命令和激活或刷新命令之间的延迟，对应寄存器 FMC_SDTRx 的 TMRD 位。ExitSelfRefreshDelay 用来设置从发出自刷新命令到发出激活命令之间的延迟，对应 TXSR 位。SelfRefreshTime 用来设置自刷新周期，对应 TRAS 位。RowCycleDelay 用来设置刷新命令和激活命令之间的延迟以及两个相邻刷新命令之间的延迟，对应 TRC 位。WriteRecoveryTime 用来设置写命令和预充电命令之间的延迟，对应 TWR 位。RPDelay 用来设置预充电命令与其它命令之间的延迟，对应位 TRP。RCDDelay 用来设置激活命令与读/写命令之间的延迟，对应位 TRCD。

函数 HAL_SDRAM_Init 的使用范例如下：

```
SDRAM_HandleTypeDef SDRAM_Handler;     //SDRAM 句柄
FMC_SDRAM_TimingTypeDef SDRAM_Timing;

    SDRAM_Handler.Instance=FMC_SDRAM_DEVICE;          //SDRAM 在 BANK5,6
    SDRAM_Handler.Init.SDBank=FMC_SDRAM_BANK1;
    SDRAM_Handler.Init.ColumnBitsNumber=
                                        FMC_SDRAM_COLUMN_BITS_NUM_9;//列数量
    SDRAM_Handler.Init.RowBitsNumber=FMC_SDRAM_ROW_BITS_NUM_13; //行数量
    SDRAM_Handler.Init.MemoryDataWidth=FMC_SDRAM_MEM_BUS_WIDTH_16;
    SDRAM_Handler.Init.InternalBankNumber=FMC_SDRAM_INTERN_BANKS_NUM_4;
    SDRAM_Handler.Init.CASLatency=FMC_SDRAM_CAS_LATENCY_3;
    SDRAM_Handler.Init.WriteProtection
                                      =FMC_SDRAM_WRITE_PROTECTION_DISABLE;
    SDRAM_Handler.Init.SDClockPeriod=FMC_SDRAM_CLOCK_PERIOD_2;
    SDRAM_Handler.Init.ReadBurst=FMC_SDRAM_RBURST_ENABLE; //使能突发
    SDRAM_Handler.Init.ReadPipeDelay=FMC_SDRAM_RPIPE_DELAY_1;   //读通道延时

    SDRAM_Timing.LoadToActiveDelay=2; //加载模式到激活时间的延迟为 2 个时钟周期
    SDRAM_Timing.ExitSelfRefreshDelay=8; //退出自刷新延迟为 8 个时钟周期
    SDRAM_Timing.SelfRefreshTime=6; //自刷新时间为 6 个时钟周期
    SDRAM_Timing.RowCycleDelay=6; //行循环延迟为 6 个时钟周期
    SDRAM_Timing.WriteRecoveryTime=2;   //恢复延迟为 2 个时钟周期
    SDRAM_Timing.RPDelay=2;   //行预充电延迟为 2 个时钟周期
    SDRAM_Timing.RCDDelay=2;        //行到列延迟为 2 个时钟周期
    HAL_SDRAM_Init(&SDRAM_Handler,&SDRAM_Timing);
```

3）发送 SDRAM 初始化序列。

这里根据前面提到的 SDRAM 初始化步骤，对 SDRAM 进行初始化，首先使能时钟配置，然后等待至少 200us，对所有 BANK 进行预充电，执行自刷新命令等，最后配置模式寄存器。完成对 SDRAM 的初始化。发送初始化系列主要是向 SRAM 存储区发送命令，HAL 库提供了发送命令函数为：

```
HAL_StatusTypeDef HAL_SDRAM_SendCommand(SDRAM_HandleTypeDef *hsdram,
                                FMC_SDRAM_CommandTypeDef *Command, uint32_t Timeout) ;
```

该函数的第一个入口参数 hsdram 是 SDRAM 句柄，第三个参数是发送命令 Timeout 时间，这两个参数都比较好理解，接下来我们着重讲解第二个入口参数 Command，该参数是 FMC_SDRAM_CommandTypeDef 结构体指针类型，该结构体定义如下：

```
typedef struct
{
  uint32_t CommandMode;            //命令类型
  uint32_t CommandTarget;          //目标 SDRAM 存储区域
  uint32_t AutoRefreshNumber;      //自刷新次数
  uint32_t ModeRegisterDefinition;  // SDRAM 模式寄存器的内容
}FMC_SDRAM_CommandTypeDef;
```

成员变量 CommandMode 用来设置命令类型，我们前面讲解过，一共有 7 种命令类型，包括时钟配置使能命令 FMC_SDRAM_CMD_CLK_ENABLE，自刷新命令 FMC_SDRAM_CMD_AUTOREFRESH_MODE 等，这里我们就不一一讲解。

CommandTarget 用来设置目标 SDRAM 存储区域，因为 SDRAM 控制器可以外挂 2 个 SDRAM，发送命令的时候，需要指定命令发送给哪个存储器取值范围为：FMC_SDRAM_CMD_TARGET_BANK1，FMC_SDRAM_CMD_TARGET_BANK2 和 FMC_SDRAM_CMD_TARGET_BANK1_2。

AutoRefreshNumber 用来设置自刷新次数，ModeRegisterDefinition 用来设置 SDRAM 模式寄存器的内容。

了解了向 SRAM 存储区发送命令方法，那么发送 SDRAM 初始化序列也就是发送命令到 SRAM 存储区，这就变得非常简单了。后面我们会给大家列出发送 SDRAM 初始化序列方法。

4）设置刷新频率，也就是设置寄存器 FMC_ SDRTR 参数。

HAL 库提供的设置刷新频率函数为：

```
HAL_StatusTypeDef HAL_SDRAM_ProgramRefreshRate(SDRAM_HandleTypeDef *hsdram,
                                               uint32_t RefreshRate);
```

该函数入口参数比较简单，这里我们不做过多讲解。

通过以上几个步骤，我们就完成了 FMC 的配置，可以访问 W9825G6KH 了。

19.2 硬件设计

本章实验功能简介：开机后，显示提示信息，然后按下 KEY0 按键，即测试外部 SDRAM 容量大小并显示在 LCD 上。按下 KEY1 按键，即显示预存在外部 SDRAM 的数据。DS0 指示程序运行状态。

本实验用到的硬件资源有：

1） 指示灯 DS0

2） KEY0 和 KEY1 按键

3） 串口

4） TFTLCD 模块

5） W9825G6KH

这些我们都已经介绍过（W9825G6KH 与 STM32F767 的各 IO 对应关系，请参考光盘原理图），接下来我们开始软件设计。

19.3 软件设计

打开本章实验工程可以看到，我们新添加了 sdram.c 到 HARDWARE 分组，用来存放我们编写的 SDRAM 相关驱动函数。

打开 sdram.c 文件，代码如下：

```
SDRAM_HandleTypeDef SDRAM_Handler;     //SDRAM 句柄

//SDRAM 初始化
void SDRAM_Init(void)
{

    FMC_SDRAM_TimingTypeDef SDRAM_Timing;
```

```
    SDRAM_Handler.Instance=FMC_SDRAM_DEVICE;       //SDRAM 在 BANK5,6
    SDRAM_Handler.Init.SDBank=FMC_SDRAM_BANK1; //SDRAM 的 BANK1
    SDRAM_Handler.Init.ColumnBitsNumber=FMC_SDRAM_COLUMN_BITS_NUM_9;
    SDRAM_Handler.Init.RowBitsNumber=FMC_SDRAM_ROW_BITS_NUM_13; //行数量
    SDRAM_Handler.Init.MemoryDataWidth=FMC_SDRAM_MEM_BUS_WIDTH_16;
    SDRAM_Handler.Init.InternalBankNumber=FMC_SDRAM_INTERN_BANKS_NUM_4;
    SDRAM_Handler.Init.CASLatency=FMC_SDRAM_CAS_LATENCY_3;       //CAS 为 3
    SDRAM_Handler.Init.WriteProtection=
                        FMC_SDRAM_WRITE_PROTECTION_DISABLE;//失能写保护
    SDRAM_Handler.Init.SDClockPeriod=FMC_SDRAM_CLOCK_PERIOD_2;
    SDRAM_Handler.Init.ReadBurst=FMC_SDRAM_RBURST_ENABLE; //使能突发
    SDRAM_Handler.Init.ReadPipeDelay=FMC_SDRAM_RPIPE_DELAY_1; //读通道延时

    SDRAM_Timing.LoadToActiveDelay=2;
                //加载模式寄存器到激活时间的延迟为 2 个时钟周期
    SDRAM_Timing.ExitSelfRefreshDelay=8; //退出自刷新延迟为 8 个时钟周期
    SDRAM_Timing.SelfRefreshTime=6;   //自刷新时间为 6 个时钟周期
    SDRAM_Timing.RowCycleDelay=6; //行循环延迟为 6 个时钟周期
    SDRAM_Timing.WriteRecoveryTime=2;   //恢复延迟为 2 个时钟周期
    SDRAM_Timing.RPDelay=2;     //行预充电延迟为 2 个时钟周期
    SDRAM_Timing.RCDDelay=2;   //行到列延迟为 2 个时钟周期
    HAL_SDRAM_Init(&SDRAM_Handler,&SDRAM_Timing);

    SDRAM_Initialization_Sequence(&SDRAM_Handler);//发送 SDRAM 初始化序列

    HAL_SDRAM_ProgramRefreshRate(&SDRAM_Handler,683);//设置刷新频率
}
//发送 SDRAM 初始化序列
void SDRAM_Initialization_Sequence(SDRAM_HandleTypeDef *hsdram)
{
    u32 temp=0;
    //SDRAM 控制器初始化完成以后还需要按照如下顺序初始化 SDRAM
    SDRAM_Send_Cmd(0,FMC_SDRAM_CMD_CLK_ENABLE,1,0); //时钟配置使能
    delay_us(500);                                  //至少延时 200us
    SDRAM_Send_Cmd(0,FMC_SDRAM_CMD_PALL,1,0);           //对所有存储区预充电
    SDRAM_Send_Cmd(0,FMC_SDRAM_CMD_AUTOREFRESH_MODE,8,0);
                                                    //设置自刷新次数
      temp=(u32)SDRAM_MODEREG_BURST_LENGTH_1 |//设置突发长度:1
                SDRAM_MODEREG_BURST_TYPE_SEQUENTIAL |//设置突发类型
                SDRAM_MODEREG_CAS_LATENCY_3 |//设置 CAS 值:3(可以是 2/3)
                SDRAM_MODEREG_OPERATING_MODE_STANDARD | //标准模式
                SDRAM_MODEREG_WRITEBURST_MODE_SINGLE;   //单点访问
    SDRAM_Send_Cmd(0,FMC_SDRAM_CMD_LOAD_MODE,1,temp); //发送命令
```

```
}
//SDRAM 底层驱动，引脚配置，时钟使能
//此函数会被 HAL_SDRAM_Init()调用
//hsdram:SDRAM 句柄
void HAL_SDRAM_MspInit(SDRAM_HandleTypeDef *hsdram)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_FMC_CLK_ENABLE();             //使能 FMC 时钟
    __HAL_RCC_GPIOC_CLK_ENABLE();           //使能 GPIOC 时钟

    …//此处省略部分 IO 时钟使能，详情请参考实验工程

    GPIO_Initure.Pin=GPIO_PIN_0|GPIO_PIN_2|GPIO_PIN_3;
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //推挽复用
    GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    GPIO_Initure.Alternate=GPIO_AF12_FMC;       //复用为 FMC
    HAL_GPIO_Init(GPIOC,&GPIO_Initure);         //初始化 PC0,2,3

    …//此处省略部分 IO 口初始化，详情请参考实验工程
}

//向 SDRAM 发送命令
//bankx:0,向 BANK5 上面的 SDRAM 发送指令
//      1,向 BANK6 上面的 SDRAM 发送指令
//cmd:指令
//refresh:自刷新次数
//regval:模式寄存器的定义
//返回值:0,正常;1,失败.
u8 SDRAM_Send_Cmd(u8 bankx,u8 cmd,u8 refresh,u16 regval)
{
    u32 target_bank=0;
    FMC_SDRAM_CommandTypeDef Command;

    if(bankx==0) target_bank=FMC_SDRAM_CMD_TARGET_BANK1;
    else if(bankx==1) target_bank=FMC_SDRAM_CMD_TARGET_BANK2;
    Command.CommandMode=cmd;                //命令
    Command.CommandTarget=target_bank;      //目标 SDRAM 存储区域
    Command.AutoRefreshNumber=refresh;      //自刷新次数
    Command.ModeRegisterDefinition=regval;  //要写入模式寄存器的值
    if(HAL_SDRAM_SendCommand(&SDRAM_Handler,&Command,0X1000)==HAL_OK)
```

```
                                                    //向 SDRAM 发送命令
        {
            return 0;
        }
        else return 1;
}

//在指定地址(WriteAddr+Bank5_SDRAM_ADDR)开始,连续写入 n 个字节.
//pBuffer:字节指针
//WriteAddr:要写入的地址
//n:要写入的字节数
void FMC_SDRAM_WriteBuffer(u8 *pBuffer,u32 WriteAddr,u32 n)
{
    for(;n!=0;n--)
    {
        *(vu8*)(Bank5_SDRAM_ADDR+WriteAddr)=*pBuffer;
        WriteAddr++;
        pBuffer++;
    }
}

//在指定地址((WriteAddr+Bank5_SDRAM_ADDR))开始,连续读出 n 个字节.
//pBuffer:字节指针
//ReadAddr:要读出的起始地址
//n:要写入的字节数
void FMC_SDRAM_ReadBuffer(u8 *pBuffer,u32 ReadAddr,u32 n)
{
    for(;n!=0;n--)
    {
        *pBuffer++=*(vu8*)(Bank5_SDRAM_ADDR+ReadAddr);
        ReadAddr++;
    }
}
```

此部分代码包含 6 个函数，SDRAM_Init 函数用于初始化 FMC/SDRAM 配置，发送 SDRAM 初始化序列和设置刷新时间等，完全就是根据我们前面所说的步骤来实现的；函数 HAL_SDRAM_MspInit 是 SDRAM 的 MSP 初始化回调函数，用来初始化 IO 口和使能时钟；函数 SDRAM_Initialization_Sequence 是单独的用来发送 SRAM 初始化序列函数，在初始化函数 SDRAM_Init 内部有调用该函数。SDRAM_Send_Cmd 函数用于给 SDRAM 发送命令，在初始化的时候需要用到； FMC_SDRAM_WriteBuffer 和 FMC_SDRAM_ReadBuffer 这两个函数分别用于在外部 SDRAM 的指定地址写入和读取指定长度的数据（字节数），一般用不到。

这里需要注意的是：当位宽为 16 位的时候，HADDR 右移一位同地址对其，但是 WriteAddr /ReadAddr 我们这里却没有加 2，而是加 1，是因为我们这里用的数据位宽是 8 位，通过 FMC_NBL1 和 FMC_NBL0 来控制高低字节位，所以地址在这里是可以只加 1 的。

最后我们看看 main.c 中程序，如下：

```
u16 testsram[250000] __attribute__((at(0XC0000000)));//测试用数组

//SDRAM 内存测试
void fsmc_sdram_test(u16 x,u16 y)
{
    u32 i=0;
    u32 temp=0;
    u32 sval=0;    //在地址 0 读到的数据
    LCD_ShowString(x,y,180,y+16,16,"Ex Memory Test:       0KB ");
    //每隔 16K 字节,写入一个数据,总共写入 2048 个数据,刚好是 32M 字节
    for(i=0;i<32*1024*1024;i+=16*1024)
    {
        *(vu32*)(Bank5_SDRAM_ADDR+i)=temp;
        temp++;
    }
    //依次读出之前写入的数据,进行校验
    for(i=0;i<32*1024*1024;i+=16*1024)
    {
        temp=*(vu32*)(Bank5_SDRAM_ADDR+i);
        if(i==0)sval=temp;
        else if(temp<=sval)break;//后面读出的数据一定要比第一次读到的数据大.

        LCD_ShowxNum(x+15*8,y,(u16)(temp-sval+1)*16,5,16,0);     //显示内存容量
        printf("SDRAM Capacity:%dKB\r\n",(u16)(temp-sval+1)*16);//打印 SDRAM 容量
    }
}

int main(void)
{
    u8 key;
    u8 i=0;
    u32 ts=0;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    ……//此处省略部分液晶显示代码
```

```
    for(ts=0;ts<250000;ts++)
    {
        testsram[ts]=ts;//预存测试数据
    }
    while(1)
    {
        key=KEY_Scan(0);//不支持连按
        if(key==KEY0_PRES)fsmc_sdram_test(30,170);//测试 SRAM 容量
        else if(key==KEY1_PRES)//打印预存测试数据
        {
            for(ts=0;ts<250000;ts++)
        {
            LCD_ShowxNum(30,190,testsram[ts],6,16,0);//显示测试数据
            printf("testsram[%d]:%d\r\n",ts,testsram[ts]);
        }
        }else delay_ms(10);
        i++;
        if(i==20)//DS0 闪烁.
        {
            i=0;
            LED0_Toggle;
        }
    }
}
```

此部分代码除了 main 函数，还有一个 fmc_sdram_test 函数，该函数用于测试外部 SRAM 的容量大小，并显示其容量。main 函数则比较简单，我们就不细说了。

此段代码，我们定义了一个超大数组 testsram，我们指定该数组定义在外部 sdram 起始地址（__attribute__((at(0XC0000000)))），该数组用来测试外部 SDRAM 数据的读写。注意该数组的定义方法，是我们推荐的使用外部 SDRAM 的方法。如果想用 MDK 自动分配，那么需要用到分散加载还需要添加汇编的 FMC 初始化代码，相对来说比较麻烦。而且外部 SDRAM 访问速度远不如内部 SRAM，如果将一些需要快速访问的 SRAM 定义到了外部 SDRAM，将严重拖慢程序运行速度。而如果以我们推荐的方式来分配外部 SDRAM，那么就可以控制 SDRAM 的分配，可以针对性的选择放外部或放内部，有利于提高程序运行速度，使用起来也比较方便。

另外，fmc_sdram_test 函数和 main 函数，我们都加入了 printf 输出结果，对于没有 MCU 屏模块的朋友来说，可以打开串口调试助手，观看实验结果，软件部分就给大家介绍到这里。

19.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到如图 19.4.1 所示界面：

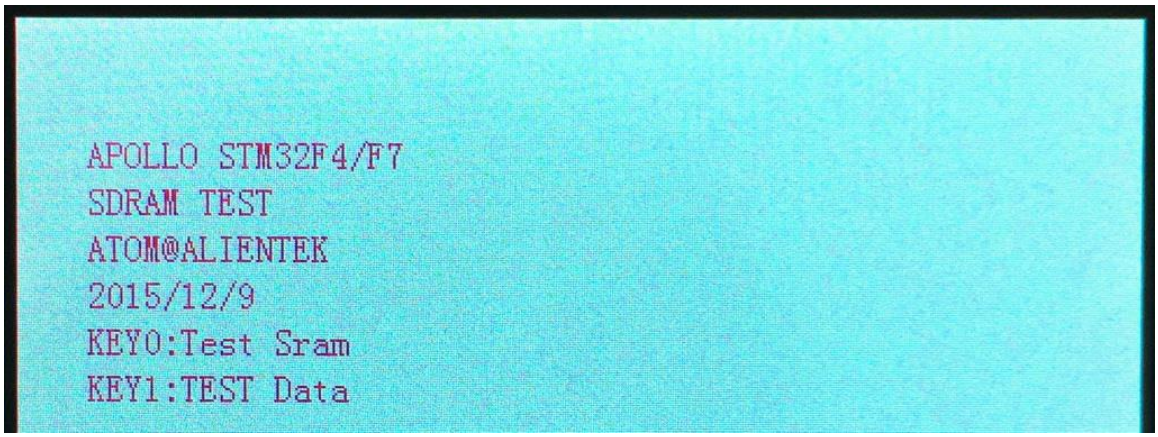


图 19.4.1 程序运行效果图

此时，我们按下 KEY0，就可以在 LCD 上看到内存测试的画面，同样，按下 KEY1，就可以看到 LCD 显示存放在数组 testsram 里面的测试数据，如图 19.4.2 所示：

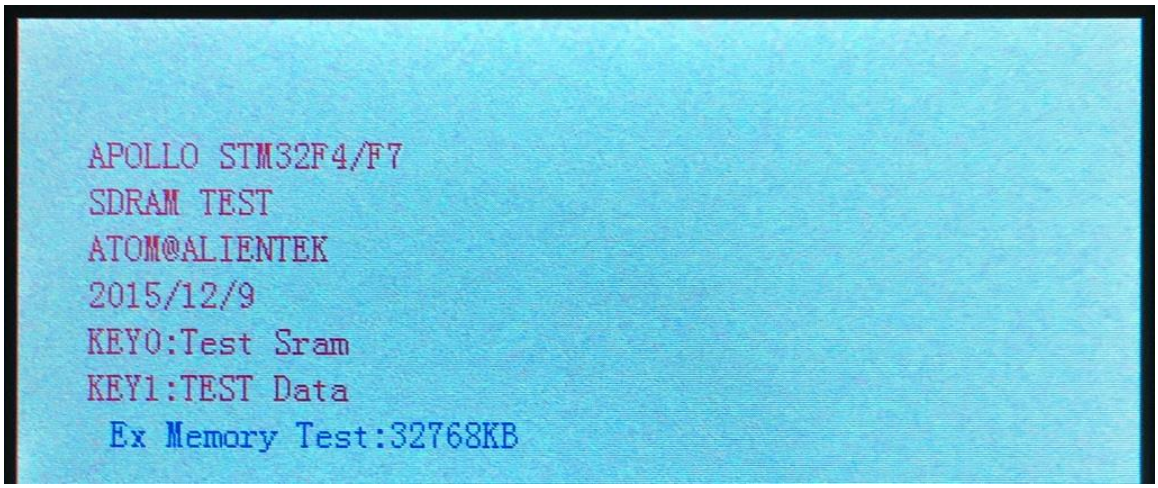


图 19.4.2 外部 SRAM 测试界面

对于没有 MCU 屏模块的朋友，我们可以用串口来检查测试结果，如图 19.4.3 所示：

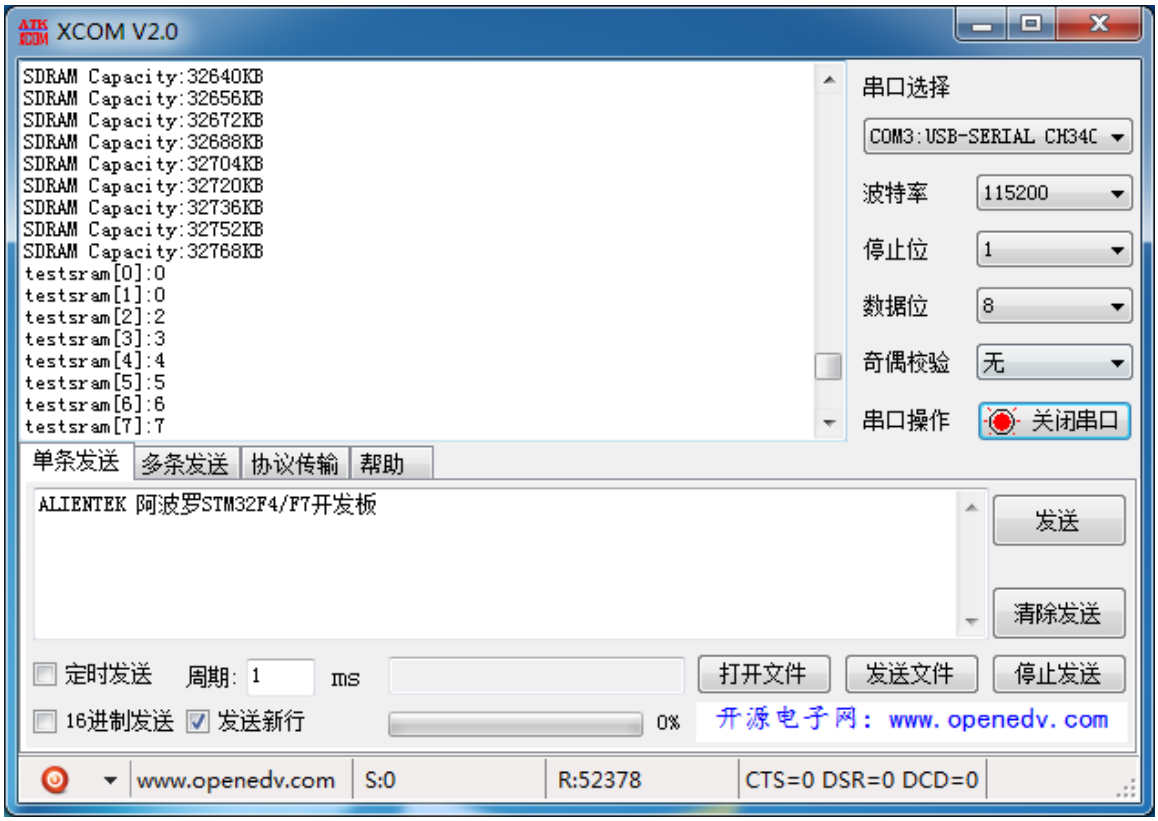


图 19.4.3 串口观看测试结果

19.5 STM32CubeMX 配置 FMC（SDRAM）

上一讲我们讲解了使用 STM32CubeMX 配置 SRAM 的方法，本小节将讲解配置 SDRAM。使用 STM32CubeMX 配置 SDRAM 的一般步骤为：

① 进入 Pinout->FMC 配置栏，配置 FMC 基本参数。这里前面讲解过，STM32F7 FMC 接口的 SDRAM 控制器一共有 2 个独立的 SDRAM 存储区域，这里我们使用的是区域 1，所以我们只需要配置 SRAM1 即可。配置如下图 19.5.1 所示：

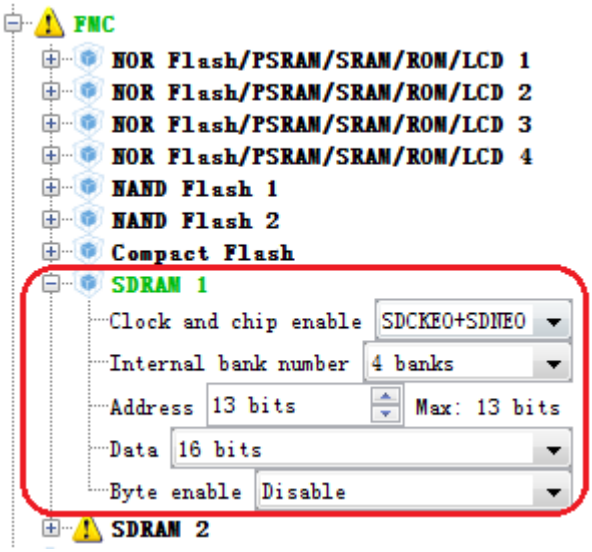


图 19.5.1 FMC 配置参数

这里我们就配置界面几个参数解释一下。Clock and chip enable 是配置时钟使能和片选引脚，很明显我们使用的 CKE 接 FMC_SDCKE0，CS 接 FMC_SDNE0，所以选择第一个即可。参数 Internal bank number 一共是 4 个，对于其他参数就很好理解了，地址 13 位，数据 16 位。

② 点击 Configuration->FMC 进入 FMC 配置界面，在 SDRAM1 选项卡之下配置相关参数。这些参数的含义这里我们不累赘，在 19.1 小节讲解 HAL_SDRAM_Init 函数的时候都有讲解。配置方法如下图 19.5.2 所示：

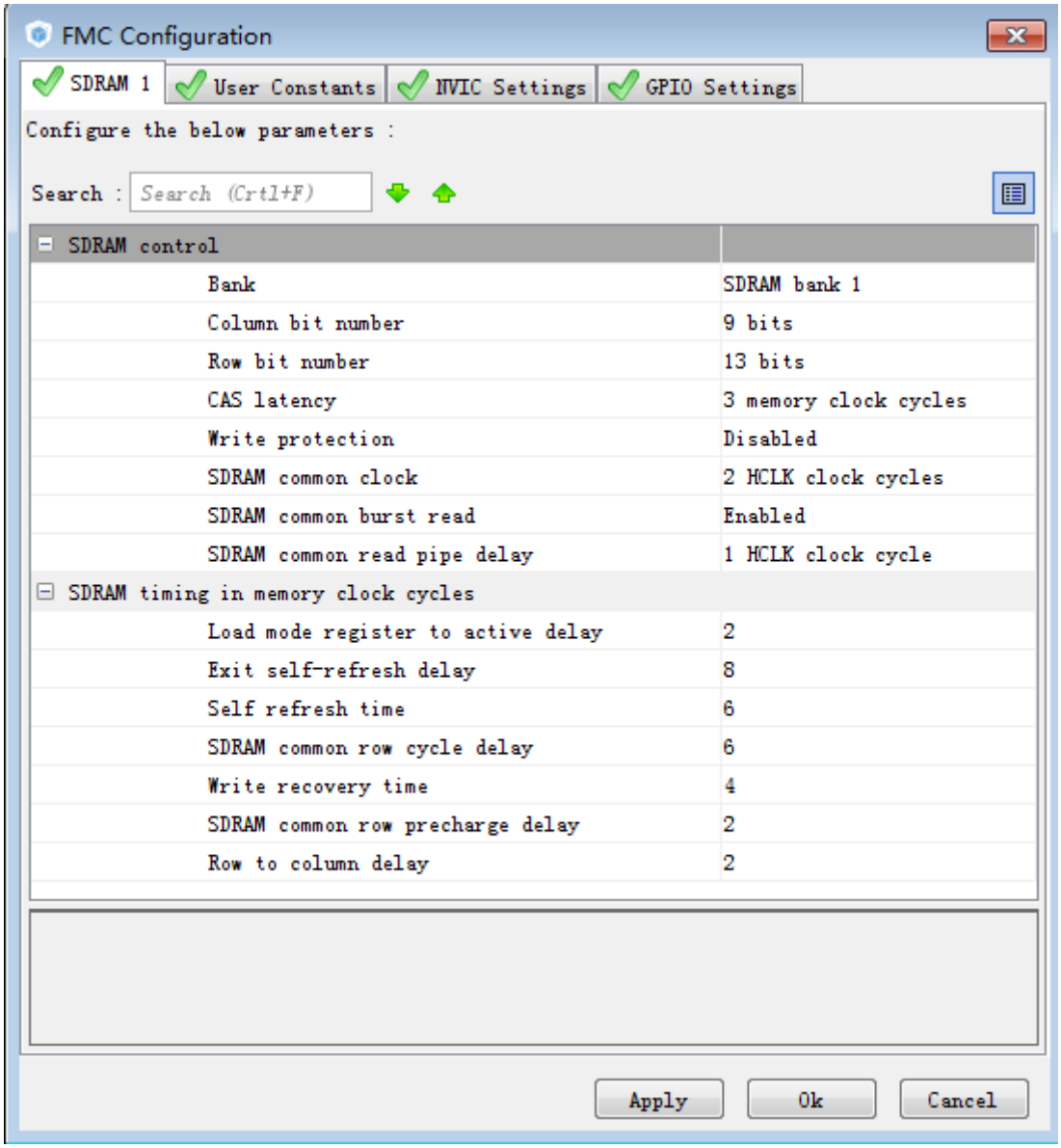


图 19.5.2 FMC Configuration 配置界面 SDRAM1 选项卡

在该配置界面，点击右边的 GPIO Settigns 选项卡，还可以配置相关 IO 口的信息。

经过上面配置步骤，我们就可以生成相应的初始化代码，大家生成后和本章实验工程对比学习。

第二十章 LTDC LCD（RGB 屏）实验

在第 18 章，我们介绍了 TFTLCD 模块（MCU 屏）的使用，但是高分辨率的屏（超过 800*480），一般都没有 MCU 屏接口，而是使用 RGB 接口的，这种接口的屏，就需要用到 STM32F767 的 LTDC 来驱动了。在本章中，我们将使用阿波罗 STM32F767 开发板核心板上的 LCD 接口（仅支持 RGB 屏，本章介绍 RGB 屏的使用），来点亮 LCD，并实现 ASCII 字符和彩色的显示等功能，并在串口打印 LCD ID，同时在 LCD 上面显示。本章分为如下几个部分：

20.1 RGBLCD&LTDC 简介
20.2 硬件设计
20.3 软件设计
20.4 下载验证

20.1 RGBLCD&LTDC 简介

本章我们将通过 STM32F767 的 LTDC 接口来驱动 RGBLCD 的显示，另外，STM32F767 的 LTDC 还有 DMA2D 图形加速，我们也顺带进行介绍。本节分为三个部分，分别介绍 RGBLCD、LTDC 和 DMA2D。

20.1.1 RGBLCD 简介

在第 18 章，我们已经介绍过 TFTLCD 液晶了，实际上 RGBLCD 也是 TFTLCD，只是接口不同而已。接下来我们简单介绍一下 RGBLCD 的驱动。

（1）RGBLCD 的信号线

RGBLCD 的信号线如表 20.1.1.1 所示：

信号线	说明
R[0:7]	红色数据线，一般为 8 位
G[0:7]	绿色数据线，一般为 8 位
B[0:7]	蓝色数据线，一般为 8 位
DE	数据使能线
VS	垂直同步信号线
HS	水平同步信号线
DCLK	像素时钟信号线

表 20.1.1.1 RGBLCD 信号线

一般的 RGB 屏都有如表 20.1.1.1 所示的信号线，有 24 根颜色数据线（RGB 各站 8 根，即 RGB888 格式），这样可以表示最多 1600W 色，DE、VS、HS 和 DCLK，用于控制数据传输。

（2）RGBLCD 的驱动模式

RGB 屏一般有 2 种驱动模式：DE 模式和 HV 模式。DE 模式使用 DE 信号来确定有效数据（DE 为高/低时，数据有效），而 HV 模式，则需要行同步和场同步，来表示扫描的行和列。

DE 模式和 HV 模式的行扫描时序图（以 800*480 的 LCD 面板为例），如图 20.1.1.1 所示：

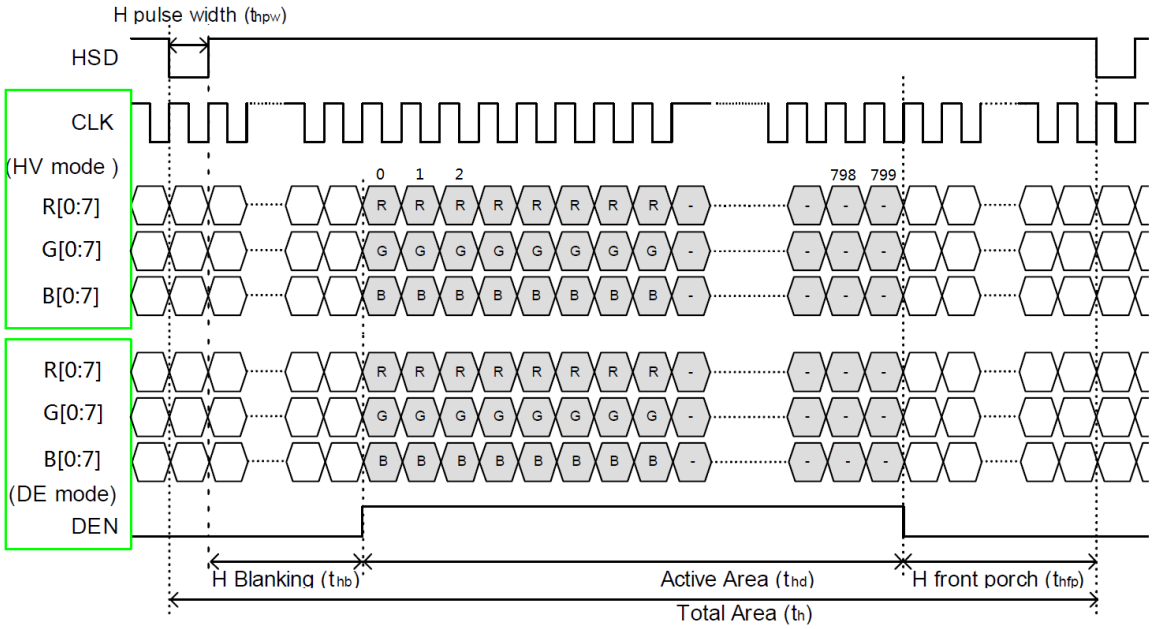


图 20.1.1.1 DE/HV 模式行扫描时序图

从图中可以看出，DE 和 HV 模式，时序基本一样，DEN 模式需要提供 DE 信号（DEN），而 HV 模式，则无需 DE 信号。图中的 HSD 即 HS 信号，用于行同步，注意：在 DE 模式下面，是可以不用 HS 信号的，即不接 HS 信号，液晶照样可以正常工作。

图中的 thpw 为水平同步有效信号脉宽，用于表示一行数据的开始；thb 为水平后廊，表示从水平有效信号开始，到有效数据输出之间的像素时钟个数；thfp 为水平前廊，表示一行数据结束后，到下一个水平同步信号开始之前的像素时钟个数；这几个时间非常重要，在配置 LTDC 的时候，需要根据 LCD 的数据手册，进行正确的设置。

图 20.1.1.1 仅是一行数据的扫描，输出 800 个像素点数据，而液晶面板总共有 480 行，这就还需要一个垂直扫描时序图，如图 20.1.1.2 所示：

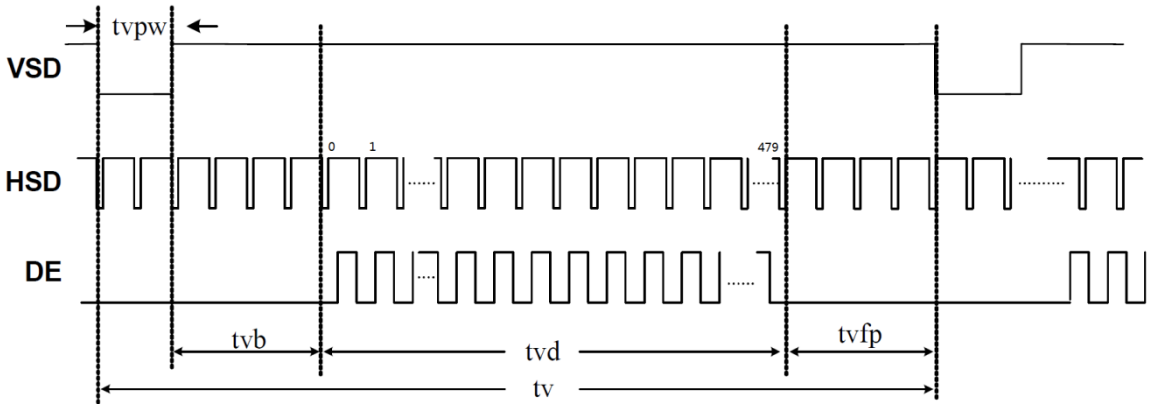


图 20.1.1.2 垂直扫描时序图

图中的 VSD 就是垂直同步信号，HSD 就是水平同步信号，DE 为数据使能信号。由图可知，一个垂直扫描，刚好就是 480 个有效的 DE 脉冲信号，每一个 DE 时钟周期，扫描一行，总共扫描 480 行，完成一帧数据的显示。这就是 800*480 的 LCD 面板扫描时序，其他分辨率的 LCD 面板，时序类似。

图中的 tvpw 为垂直同步有效信号脉宽，用于表示一帧数据的开始；tvb 为垂直后廊，表示垂直同步信号以后的无效行数，tvfp 为垂直前廊，表示一帧数据输出结束后，到下一个垂直同步信号开始之前的无效行数；这几个时间同样在配置 LTDC 的时候，需要进行设置。

（3）ALIENTEK RGBLCD 模块

ALIENTEK 目前提供大家三款 RGBLCD 模块：ATK-4342（4.3 寸，480*272）、ATK-7084（7 寸，800*480）和 ATK-7016（7 寸，1024*600），这里我们以 ATK-7084 为例，给大家介绍。该模块的接口原理图如图 20.1.1.3 所示：

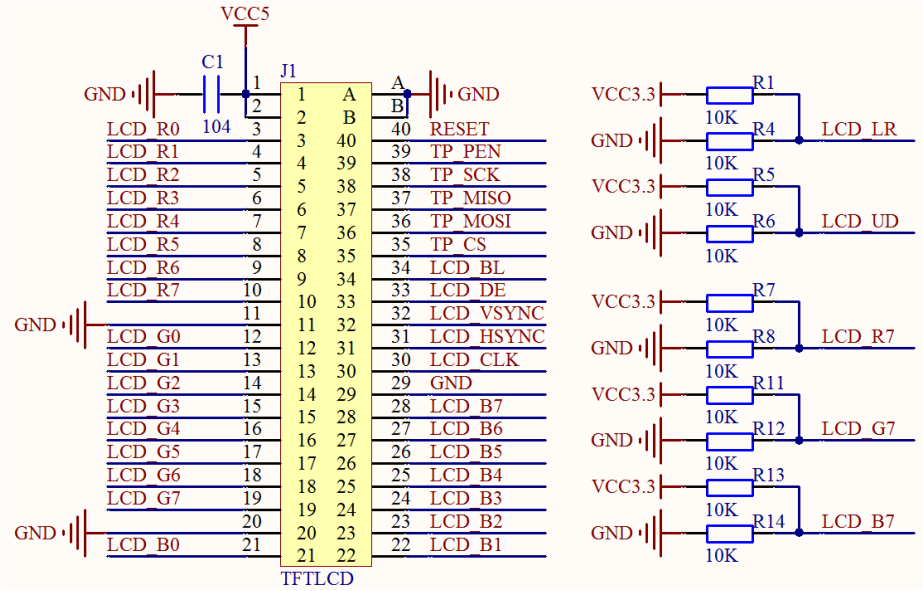


图 20.1.1.3 ATK-7084 模块对外接口原理图

图中 J1 就是对外接口，是一个 40PIN 的 FPC 座（0.5mm 间距），通过 FPC 线，可以连接到阿波罗 STM32F767 开发板的核心板上面，从而实现和 STM32F767 的连接。该接口十分完善，采用 RGB888 格式，并支持 DE&HV 模式，还支持触摸屏（电阻/电容）和背光控制。右侧的几个电阻，并不是都焊接的，而是可以用户自己选择。默认情况，R1 和 R6 焊接，设置 LCD_LR 和 LCD_UD，控制 LCD 的扫描方向，是从左到右，从上到下（横屏看）。而 LCD_R7/G7/B7 则用来设置 LCD 的 ID，由于 RGBLCD 没有读写寄存器，也就没有所谓的 ID，这里我们通过在模块上面，控制 R7/G7/B7 的上/下拉，来自定义 LCD 模块的 ID，帮助 MCU 判断当前 LCD 面板的分辨率和相关参数，以提高程序兼容性。这几个位的设置关系如表 20.1.1.2 所示：

M2 LCD_G7	M1 LCD_G7	M0 LCD_R7	LCD ID	说明
0	0	0	4342	ATK-4342 RGBLCD 模块，分辨率：480*272
0	0	1	7084	ATK-7084 RGBLCD 模块，分辨率：800*480
0	1	0	7016	ATK-7016，RGBLCD 模块，分辨率：1024*600
0	1	1	7018	ATK-7018，RGBLCD 模块，分辨率：1280*800
1	0	0	8016	ATK-8016，RGBLCD 模块，分辨率：1024*600
X	X	X	NC	暂时未用到

表 20.1.1.2 ALIENTEK RGBLCD 模块 ID 对应关系

ATK-7084 模块，就设置 M2:M0=001 即可。这样，我们在程序里面，读取 LCD_R7/G7/B7，得到 M0:M2 的值，从而判断 RGBLCD 模块的型号，并执行不同的配置，即可实现不同 LCD 模块的兼容。

RGBLCD 我们就给大家介绍到这里，接下来，我们介绍 LTDC。

20.1.2 LTDC 简介

STM32F767xx 系列芯片都带有 TFT LCD 控制器，即 LTDC，通过这个 LTDC，STM32F767 可以直接外接 RGBLCD 屏，实现液晶驱动。STM32F767 的 LTDC 具有如下特点：

- 24 位 RGB 并行像素输出；每像素 8 位数据(RGB888)
- 2 个带有专用 FIFO 的显示层（FIFO 深度 64x32 位）
- 支持查色表 (CLUT)，每层高达 256 种颜色（256x24 位）
- 可针对不同显示面板编程时序
- 可编程背景色
- 可编程 HSync、VSync 和数据使能(DE)信号的极性
- 每层有多达 8 种颜色格式可供选择：ARGB8888、RGB888、RGB565、ARGB1555、ARGB4444、L8（8 位 Luminance 或 CLUT）、AL44（4 位 alpha+4 位 luminance）和 AL88（8 位 alpha+8 位 luminance）
- 每通道的低位采用伪随机抖动输出（红色、绿色、蓝色的抖动宽度为 2 位）
- 使用 alpha 值（每像素或常数）在两层之间灵活混合
- 色键（透明颜色）
- 可编程窗口位置和大小
- 支持薄膜晶体管 (TFT) 彩色显示器
- AHB 主接口支持 16 个字的突发
- 高达 4 个可编程中断事件

LTDC 控制器主要包含：信号线、图像处理单元、AHB 接口、配置和状态寄存器以及时钟部分，其框图如图 20.1.2.1 所示：

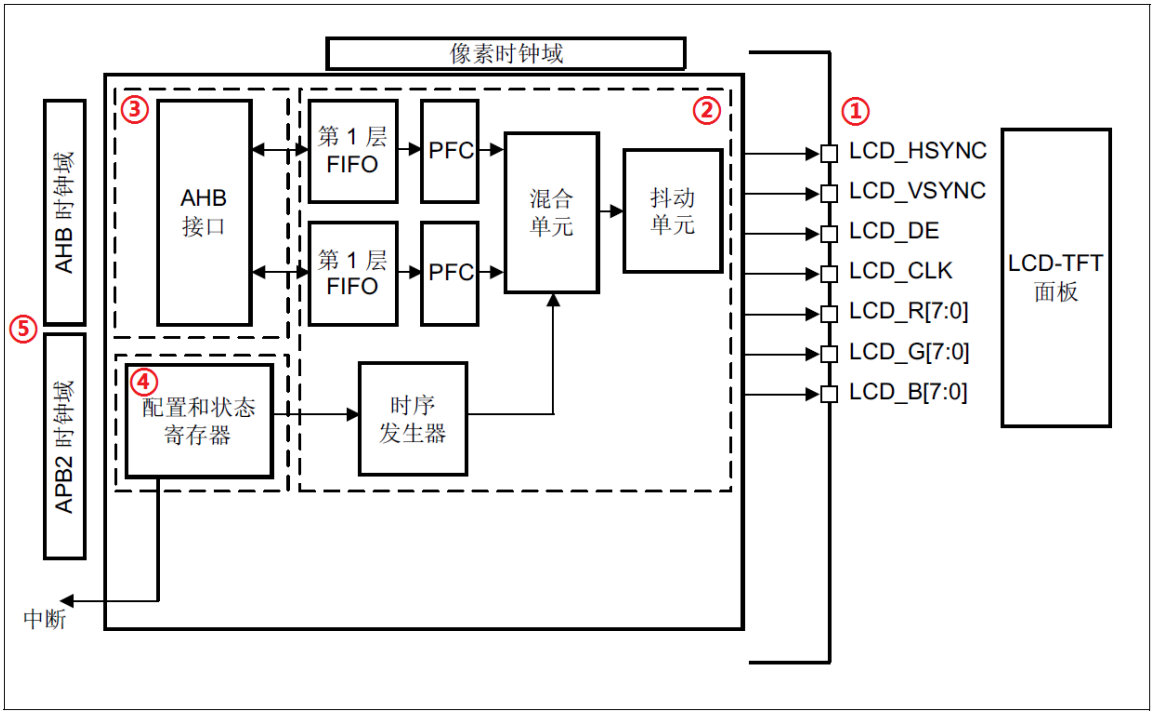


图 20.1.2.1 LTDC 控制器框图

① 信号线

这里就包含了我们前面提到的 RGBLCD 驱动所需要的所有信号线，这些信号线通过

STM32F767 核心板板载的 LCD 接口引出，其信号说明和 IO 连接关系，见表 20.1.2.1：

LTDC 信号线	对应 IO	说明
LCD_CLK	PG7	像素时钟输出
LCD_HSYNC	PI10	水平同步
LCD_VSYNC	PI9	垂直同步
LCD_DE	PF10	数据使能
LCD_R[7:3]	PG6、PH12、PH11、PH10、PH9	红色数据线，LCD_R[2:0]未用到
LCD_G[7:2]	PI2、PI1、PI0、PH15、PH14、PH13	绿色数据线，LCD_G[1:0]未用到
LCD_B[7:3]	PI7、PI6、PI5、PI4、PG11	蓝色数据线，LCD_B[2:0]未用到

表 20.1.2.1 LTDC 信号线及 IO 连接关系说明

LTDC 总共有 24 位数据线，支持 RGB888 格式，但是我们为了节省 IO，并提高图片显示速度，使用 RGB565 颜色格式，这样的话，只需要 16 个 IO 口，当使用 RGB565 格式的时候，LCD 面板的数据线，必须连接到 LTDC 数据线的 MSB，即：LTDC 的 LCD_R[7:3]接 RGBLCD 的 R[7:3]，LTDC 的 LCD_G[7:2]接 RGBLCD 的 G[7:2]，LTDC 的 LCD_B[7:3]接 RGBLCD 的 B[7:3]，这样，RGB 数据线分别是 5:6:5，即 RGB565 格式。表中对应 IO 就是我们 STM32F767 核心板上面，LCD 接口所连接的 IO。

② 图像处理单元

此部分先从 AHB 接口获取显存中的图像数据，然后经过层 FIFO（有 2 个，对应 2 个层）缓存，每个层 FIFO 具有 64*32 位存储深度，然后经过像素格式转换器（PFC），把从层的所选输入像素格式转换为 ARGB8888 格式，再通过混合单元，把两层数据合并，混合得到单层要显示的数据，最后经过抖动单元处理（可选）后，输出给 LCD 显示。

这里的 ARGB8888，即带 8 位透明通道，即最高 8 位为透明通道参数，表示透明度，值越大，则约不透明，值越小，越透明。比如 A=255 时，表示完全不透明，而 A=0 时，表示完全透明。RGB888 就表示 R、G、B 各 8 位，可表示的颜色深度为 1600W 色。

STM32F767 的 LTDC 总共有三个层：背景层、第一层和第二层，其中，背景层只可以是纯色（即单色），而第一层和第二层都可以用来显示信息，混合单元会将三个层混合起来，进行显示，显示关系如图 20.1.2.2 所示：

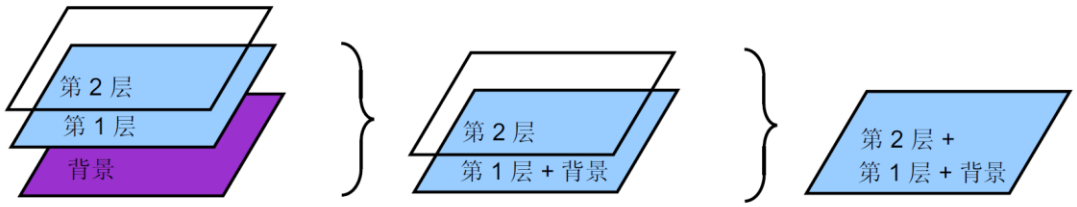


图 20.1.2.2 三个层混合关系

从图中可以看出，第二层位于最顶端，背景层位于最低端，混合单元首先将第一层与背景层进行混合，随后，第二层与第一层和第二层的混合颜色结果再次混合，完成混合后，送给 LCD 显示。

③ AHB 接口

由于 LTDC 驱动 RGBLCD 的时候，需要有很多内存来做显存，比如一个 800*480 的屏幕，按一般的 16 位 RGB565 模式，一个像素需要 2 个字节的内存，总共需要：800*480*2=768K 字节内存，STM32 内部是没有这么多内存的，所以必须借助外部 SDRAM，而 SDRAM 是挂在

AHB 总线上的，LTDC 的 AHB 接口，就是用来将显存数据，从 SDRAM 存储器传输到 FIFO 里面。

④ 配置和状态寄存器

此部分包含了 LTDC 的各种配置寄存器以及状态寄存器，用于控制整个 LTDC 的工作参数，主要有：各信号的有效电平、垂直/水平同步时间参数、像素格式、数据使能等等。LTDC 的同步时序（HV 模式）控制框图，如图 20.1.2.3 所示：

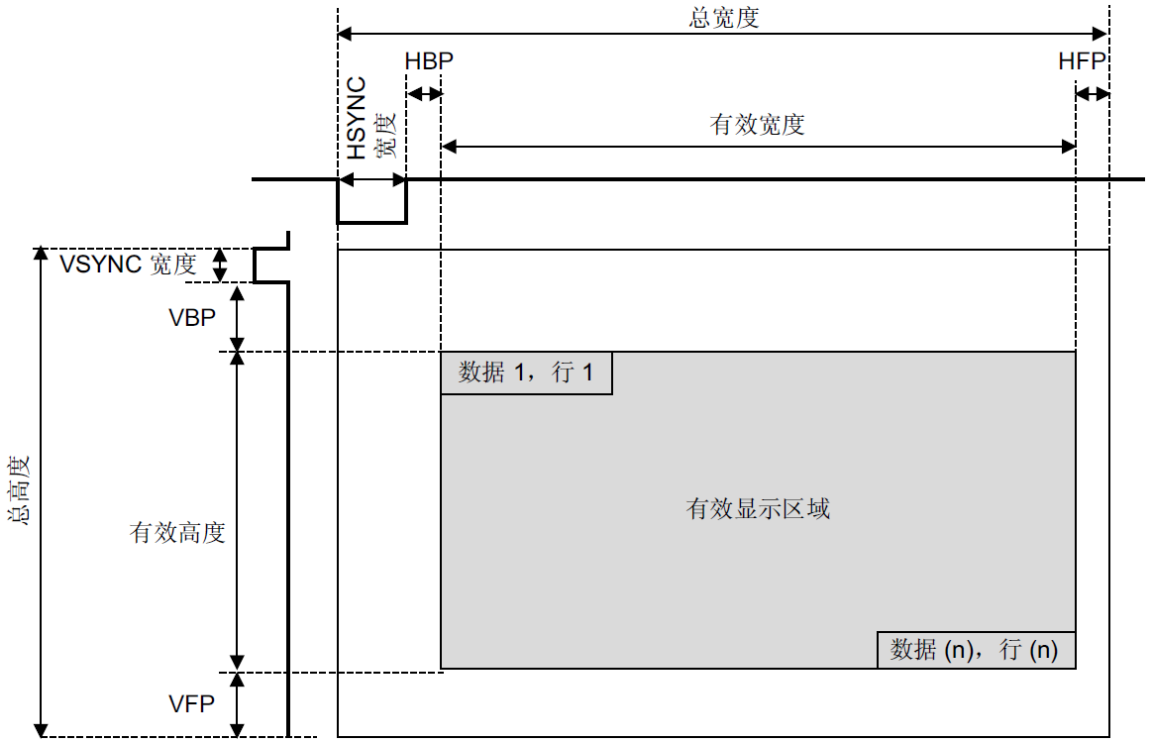


图 20.1.2.3 LTDC 同步时序框图

图中有效显示区域，就是我们 RGBLCD 面板的显示范围（即分辨率），有效宽度*有效高度，就是 LCD 的分辨率。另外，这里还有的参数包括：HSYNC 的宽度（HSW）、VSYNC 的宽度（VSW）、HBP、HFP、VBP 和 VFP 等，这些参数的说明，见表 20.1.2.2：

参数	说明
HSW(horizontal sync width)	水平同步脉宽，单位为相素时钟（CLK）个数
VSW(vertical sync width)	垂直同步脉宽，单位为行周期个数
HBP(horizontal back porch)	水平后廊，表示水平同步信号开始到行有效数据开始之间的相素时钟（CLK）个数
HFP(horizontal front porch)	水平前廊，表示行有效数据结束到下一个水平有效信号开始之前的相素时钟（CLK）个数
VBP(vertical back porch)	垂直后廊，表示垂直同步信号后，无效行的个数
VFP(vertical front porch)	垂直前廊，表示一帧数据输出结束后，到下一个垂直同步信号开始之前的无效行数

表 20.1.2.2 LTDC 驱动时序参数

如果 RGBLCD 使用的是 DE 模式，LTDC 也只需要设置表 20.1.2.2 所示的参数，然后 LTDC 会根据这些设置，自动控制 DE 信号。这些参数通过相关寄存器来配置，接下来，我们介绍一下 LTDC 的一些相关寄存器。

首先，我们来看 LTDC 全局控制寄存器：LTDC_GCR，该寄存器各位描述如图 20.1.2.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
HSPOL	VSPOL	DEPOL	PCPOL	Reserved											DEN
rw	rw	rw	rw												rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	DRW			Reserved	DGW			Reserved	DBW			Reserved			LTDCEN
	r	r	r		r	r	r		r	r	r				rw

图 20.1.2.4 LTDC_GCR 寄存器各位描述

该寄存器我们在本章用到的设置有：LTDCEN、PCPOL、DEPOL、VSPOL 和 HSPOL 这几个设置，我们将逐个介绍。

LTDCEN：TFT LCD 控制器使能位，也就是 LTDC 的开关，该位需要设置为 1。

PCPOL：像素时钟极性。控制像素时钟的极性，根据 LCD 面板的特性来设置，我们所用的 LCD 一般设置为 0 即可，表示低电平有效。

DEPOL：数据使能极性。控制 DE 信号的极性，根据 LCD 面板的特性来设置，我们所用的 LCD 一般设置为 0 即可，表示低电平有效。

VSPOL：垂直同步极性。控制 VSYNC 信号的极性，根据 LCD 面板的特性来设置，我们所用的 LCD 一般设置为 0 即可，表示低电平有效。

HSPOL：水平同步极性。控制 HSYNC 信号的极性，根据 LCD 面板的特性来设置，我们所用的 LCD 一般设置为 0 即可，表示低电平有效。

接下来，我们看看 LTDC 同步大小配置寄存器：LTDC_SSCR，该寄存器各位描述如图 20.1.2.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				HSW											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				VSH											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.5 LTDC_SSCR 寄存器各位描述

该寄存器用于设置垂直同步高度（VSH）和水平同步宽度（HSW），其中：

VSH：表示垂直同步高度(以水平扫描行为单位)，表示垂直同步脉宽减 1，即 VSW-1。

HSW：表示水平同步宽度(以像素时钟为单位)，表示水平同步脉宽减 1，即 HSW-1。

接下来，我们看看 LTDC 后沿配置寄存器：LTDC_BPCR，该寄存器各位描述如图 20.1.2.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				AHBP											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				AVBP											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.6 LTDC_BPCR 寄存器各位描述

该寄存器我们需要配置 AVBP 和 AHBP：

AVBP：累加垂直后沿(以水平扫描行为单位)，表示：VSW+VBP-1（见表 20.1.2.2）。

AHBP：累加水平后沿(以像素时钟为单位)，表示 HSW+HBP-1（见表 20.1.2.2，下同）。

接下来，我们看看 LTDC 有效宽度配置寄存器：LTDC_AWCR，该寄存器各位描述如图 20.1.2.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				AAW											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				AAH											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.7 LTDC_AWCR 寄存器各位描述

该寄存器我们需要配置 AAH 和 AAW：

AAH：累加有效高度（以水平扫描行为单位），表示：VSW+VBP+有效高度-1。

AAW：累加有效宽度（以像素时钟为单位），表示：HSW+HBP+有效宽度-1。

这里所说的有效高度和有效宽度，是指 LCD 面板的宽度和高度（构成分辨率，下同）。

接下来，我们看看 LTDC 总宽度配置寄存器：LTDC_TWCR，该寄存器各位描述如图 20.1.2.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TOTALW											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
16	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				TOTALH											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.8 LTDC_TWCR 寄存器各位描述

该寄存器我们需要配置 TOTALH 和 TOTALW：

TOTALH：总高度（以水平扫描行为单位），表示：VSW+VBP+有效高度+VFP-1。

TOTALW：总宽度（以像素时钟为单位），表示：HSW+HBP+有效宽度+HFP-1。

接下来，我们看看 LTDC 背景色配置寄存器：LTDC_BCCR，该寄存器各位描述如图 20.1.2.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								BCRED							
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BCGREEN								BCBLUE							
rw								rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.9 LTDC_BCCR 寄存器各位描述

该寄存器定义背景层的颜色（RGB888），通过低 24 位配置，我们一般设置为全 0 即可。

接下来，我们看看 LTDC 的层颜色帧缓冲区地址寄存器：LTDC_LxCFBAR（x=1/2），该寄存器各位描述如图 20.1.2.10 所示：

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

图 20.1.2.10 LTDC_LxCFBAR 寄存器各位描述

该寄存器用来定义一层显存的起始地址。STM32F767 的 LTDC 支持 2 个层，所以总共有两个寄存器，分别设置层 1 和层 2 的显存起始地址。

接下来，我们看看 LTDC 的层像素格式配置寄存器：LTDC_LxPFCR（x=1/2），该寄存器只有最低 3 位有效，用于设置层颜色的像素格式：000：ARGB8888；001：RGB888；010：RGB565；011：ARGB1555；100：ARGB4444；101：L8（8 位 Luminance）；110：AL44（4 位 Alpha，4 位 Luminance）；111：AL88（8 位 Alpha，8 位 Luminance）。我们一般使用 RGB565 格式，即该寄存器设置为：010 即可。

接下来，我们看看 LTDC 的层恒定 Alpha 配置寄存器：LTDC_LxCACR（x=1/2），该寄存器各位描述如图 20.1.2.11 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								CONSTA							
								rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.11 LTDC_LxCACR 寄存器各位描述

该寄存器低 8 位（CONSTA）有效，这些位配置混合时使用的恒定 Alpha。恒定 Alpha 由硬件实现 255 分频。关于这个恒定 Alpha 的使用，我们将在介绍 LTDC_LxBFCR 寄存器的时候进行讲解。

接下来，我们看看 LTDC 的层默认颜色配置寄存器：LTDC_LxDCCR（x=1/2），该寄存器各位描述如图 20.1.2.12 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DCALPHA								DCRED							
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
DCGREEN								DCBLUE							
rw								rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.12 LTDC_LxDCCR 寄存器各位描述

该寄存器定义采用 ARGB8888 格式的层的默认颜色。默认颜色在定义的层窗口外使用或在层禁止时使用。一般情况下，用不到，所以该寄存器一般设置为 0 即可。

接下来，我们看看 LTDC 的层混合系数配置寄存器：LTDC_LxBFCR（x=1/2），该寄存器各位描述如图 20.1.2.13 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					BF1			Reserved					BF2		
					rw	rw	rw						rw	rw	rw

图 20.1.2.13 LTDC_LxBFCR 寄存器各位描述

该寄存器用于定义混合系数：BF1 和 BF2。BF1=100 的时候，使用恒定的 Alpha 混合系数（由 LTDC_LxCACR 寄存器设置恒定 Alpha 值），BF1=110 的时候，使用像素 Alpha*恒定 Alpha。像素 Alpha 即 ARGB 格式像素的 A 值（Alpha 值），仅限 ARGB 颜色格式时使用。在 RGB565 格式下，我们设置 BF1=100 即可。BF2 同 BF1 类似，BF2=101 的时候，使用恒定的 Alpha 混合系数，BF2=111 的时候，使用像素 Alpha*恒定 Alpha。在 RGB565 格式下，我们设置 BF2=101 即可。

通用的混合公式为：

$$BC=BF1*C+BF2*Cs$$

其中：BC=混合后的颜色；BF1=混合系数 1；C=当前层颜色，即我们写入层显存的颜色值；BF2=混合系数 2；Cs=底层混合后的颜色，对于层 1 来说，Cs=背景层的颜色，对于层 2 来说，Cs=背景层和层 1 混合后的颜色。

以使用恒定的 Alpha 值，并仅使能第一层为例，给大家讲解一下混色的计算方式。恒定 Alpha 的值由 LTDC_LxCACR 寄存器设置，恒定 Alpha=LTDC_LxCACR 设置值/255。假设：LTDC_LxCACR=240；C=128；Cs（背景色）=48；那么恒定 Alpha=240/255=0.94，则：

$$BC=0.94*128+(1-0.94)*48=123$$

则混合后，颜色值变成了 123。另外，需要注意的是：BF1 和 BF2 的恒定 Alpha 值互补，他们之和为 1，且 BF1 使用的是恒定 Alpha 值，BF2 使用的是互补值。一般情况下，我们设置 LTDC_LxCACR 的值为 255，这样，在使用恒定 Alpha 值的时候，可以得到 BC=C，即混合后的颜色，就是显存里面的颜色（不进行混色）。

LTDC 的层支持窗口设置功能，通过 LTDC_LxWHPCR 和 LTDC_LxWVPCR 这两个寄存器设置，可以调整显示区域的大小，如图 20.1.2.14 所示：

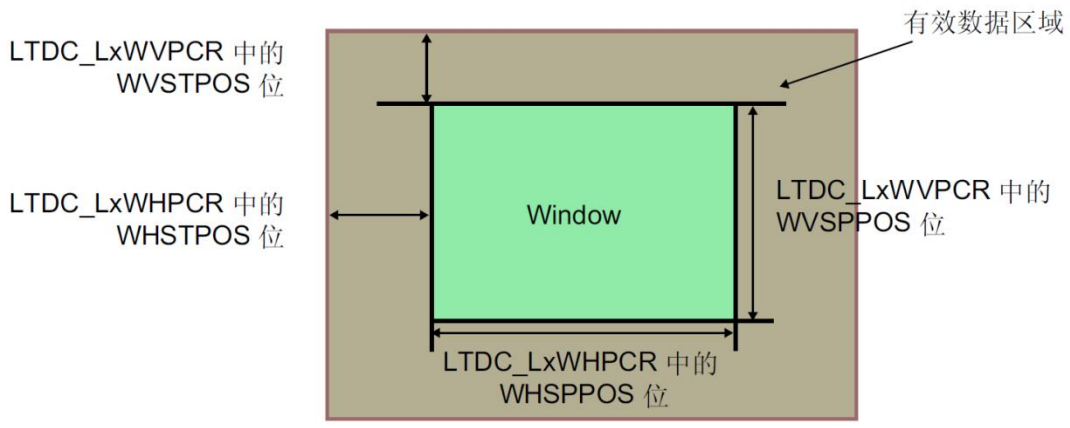


图 20.1.2.14 LTDC 层窗口设置关系图

上图中，层中的第一个和最后一个可见像素通过配置 LTDC_LxWHPCR 寄存器中的 WHSTPOS[11:0]和 WHSPPOS[11:0]进行设置。层中的第一个和最后一个可见行通过配置 LTDC_LxWVPCR 寄存器中的 WHSTPOS[11:0]和 WHSPPOS[11:0]进行设置，配置完成后，即

可确定窗口的大小。

接下来，我们来介绍这两个寄存器，首先是 LTDC 的层窗口水平位置配置寄存器：LTDC_LxWHPCR（x=1/2），该寄存器各位描述如图 20.1.2.15 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				WHSPPOS											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				WHSTPOS											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.15 LTDC_LxWHPCR 寄存器各位描述

该寄存器定义第 1 层或第 2 层窗口的水平位置（第一个和最后一个像素），其中：

WHSTPOS：窗口水平起始位置，定义层窗口的一行的第一个可见像素，见图 20.1.2.14。

WHSPPOS：窗口水平停止位置，定义层窗口的一行的最后一个可见像素，见图 20.1.2.14。

然后，我们介绍 LTDC 的层窗口垂直位置配置寄存器：LTDC_LxWVPCR（x=1/2），该寄存器各位描述如图 20.1.2.16 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					WVSPPOS										
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					WVSTPOS										
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.16 LTDC_LxWVPCR 寄存器各位描述

该寄存器定义第 1 层或第 2 层窗口的垂直位置（第一行或最后一行），其中：

WVSTPOS：窗口垂直起始位置，定义层窗口的第一个可见行，见图 20.1.2.14。

WVSPPOS：窗口垂直停止位置，定义层窗口的最后一个可见行，见图 20.1.2.14。

接下来，我们看看 LTDC 的层颜色帧缓冲区长度寄存器：LTDC_LxCFBLR（x=1/2），该寄存器各位描述如图 20.1.2.17 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved			CFBP												
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CFBLL												
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.17 LTDC_LxCFBLR 寄存器各位描述

该寄存器定义颜色帧缓冲区的行长和行间距。其中：

CFBLL：这些位定义一行像素的长度（以字节为单位）+3。行长的计算方法为：有效宽度*每像素的字节数+3。比如，LCD 面板的分辨率为 800*480，有效宽度为 800，采用 RGB565 格式，那么 CFBLL 需要设置为：800*2+3=1603。

CFBP：这些位定义从像素某行的起始处到下一行的起始处的增量（以字节为单位）。这个设置，其实同样是一行像素的长度，对于 800*480 的 LCD 面板，RGB565 格式，设置 CFBP 为：800*2=1600 即可。

最后，我们看看 LTDC 的层颜色帧缓冲区行数寄存器：LTDC_LxCFBLNR（x=1/2），该寄存器各位描述如图 20.1.2.18 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved			CFBP												
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CFBLL												
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.2.18 LTDC_LxCFBLNR 寄存器各位描述

该寄存器定义颜色帧缓冲区中的行数。CFBLNBR 用于定义帧缓冲区行数，比如，LCD 面板的分辨率为 800*480，那么帧缓冲区的行数为 480 行，则设置 CFBLNBR=480 即可。

至此，LTDC 相关的寄存器，基本就介绍完了，通过这些寄存器的配置，我们就可以完成对 LTDC 的初始化，控制 LCD 显示了。关于 LTDC 的详细介绍，和寄存器描述，请看《STM32F7 中文参考手册.pdf》第 18 章。

⑤ 时钟域

LTDC 有三个时钟域：AHB 时钟域（HCLK）、APB2 时钟域（PCLK2）和像素时钟域（LCD_CLK），AHB 时钟域用于驱动 AHB 接口，读取存储器的数据到 FIFO 里面，APB2 时钟域用于配置寄存器，像素时钟域则用于生成 LCD 接口信号，LCD_CLK 的输出应按照 LCD 面板要求进行配置。

接下来，我们重点介绍下 LCD_CLK 的配置过程。LCD_CLK 的时钟来源，如图 20.1.2.19 所示：

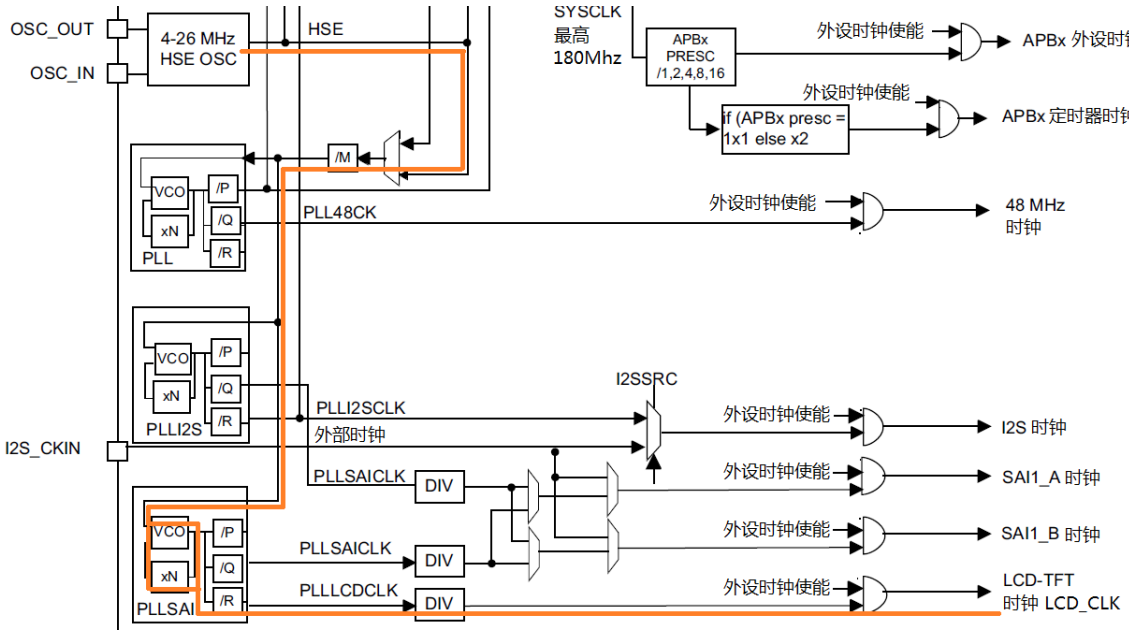


图 20.1.2.19 LCD_CLK 时钟图

由图可知，LCD_CLK 的来源，为外部晶振（假定外部晶振作为系统时钟源），经过分频器分频（/M），然后经过 PLLSAI 倍频器倍频（xN）后，经 R 分频因子输出分频后的时钟，得到 PLLLCDCLK，然后在经过 DIV 分频和时钟使能后，得到 LCD_CLK。接下来，我们简单介绍下配置 LCD_CLK 需要用到的一些寄存器。

首先是 RCC PLL SAI 配置寄存器：RCC_PLLSAICFGR，该寄存器的各位描述如图 20.1.2.20 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	PLLSAIR			PLLSAIQ				Reserved							
	rw	rw	rw	rw	rw	rw	rw								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	PLLSAIN									Reserved					
	rw	rw	rw	rw	rw	rw	rw	rw	rw						

图 20.1.2.20 RCC_PLLSAICFGR 寄存器各位描述

这个寄存器主要对 PLLSAI 倍频器的：N、Q 和 R 等参数进行配置，他们的设置关系（假定使用外部 HSE 作为时钟源）为：

f(VCO clock) = f(hse) ×(PLLSAIN / PLLM)

f(PLLSACLK) = f(VCO clock) / PLLSAIQ

f(PLLLCDCLK) = f(VCO clock) / PLLSAIR

f(hse)为我们外部晶振的频率，PLLM 就是 M 分频因子，PLLSAIN 为 PLLSAI 的倍频数，取值范围为：49~432；PLLSAIQ 为 PLLSAI 的 Q 分频系数，取值范围为：2~15；PLLSAIR 为 PLLSAI 的 R 分频系数，取值范围为：2~7；阿波罗 STM32F767 核心板所用的 HSE 晶振频率为 25Mhz，一般我们设置 PLLM 为 25，那么输入 PLLSAI 的时钟频率就是 1Mhz，然后可得：

f(PLLLCDCLK) =1Mhz* PLLSAIN/PLLSAIR

在 f(PLLLCDCLK)之后，还有一个分频器（DIV），分频后得到最终的 LCD_CLK 频率，该分频由 RCC 专用时钟配置寄存器：RCC_DCKCFGR1 配置，该寄存器各位描述如图 20.1.2.21 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TIMPRE	SAI2SEL[1:0]		SAI1SEL[1:0]		Res.	Res.	PLLSAIDIVR[1:0]	
							rw	rw	rw	rw	rw			rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	PLLSAIDIVQ[4:0]					Res.	Res.	Res.	PLLI2SDIVQ[4:0]				
			rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

图 20.1.2.21 RCC_ DCKCFGR1 寄存器各位描述

在本章，该寄存器我们只关心 PLLSAIDIVR 的配置，这两个位用于配置 f(PLLLCDCLK)之后的分频，设置范围为：0~2，表示：2^(PLLSAIDIVR+1)分频。因此，我们最终得到 LCD_CLK 的频率计算公式为(前提：HSE=25Mhz，PLLM=25)：

f(LCD_CLK)= 1Mhz* PLLSAIN/PLLSAIR/2^(PLLSAIDIVR+1)

以群创 AT070TN92 面板为例，查其数据手册，可知 DCLK 的频率典型值为：33.3Mhz，我们需要设置：PLLSAIN=396，PLLSAIR=3，PLLSAIDIVR=1，得到：

f(LCD_CLK)= 1Mhz* 396/3/2^(1+1)=33Mhz

最后，我们来看看实现 LTDC 驱动 RGBLCD，需要对 LTDC 进行哪些配置。LTDC 相关 HAL 库操作分布在函数 stm32f7xx_hal_ltdc.c 和 stm32f7xx_hal_ltdc_ex.c 以及他们对应的头文件中。操作步骤如下：

1）使能 LTDC 时钟，并配置 LTDC 相关的 IO 及其时钟使能。

要使用 LTDC，当然首先得开启其时钟。然后需要把 LCD_R/G/B 数据线、LCD_HSYNC 和 LCD_VSYNC 等相关 IO 口，全部配置为复用输出，并使能各 IO 组的时钟。 GPIO 配置这里我们就不做讲解，LTDC 时钟使能方法为：

```
__HAL_RCC_LTDC_CLK_ENABLE();                    //使能 LTDC 时钟
```

2）设置 LCD_CLK 时钟。

此步需要配置 LCD 的像素时钟，根据 LCD 的面板参数进行设置，LCD_CLK 由 PLLSAI 进行配置，前面我们已经讲解非常详细，配置使用到的 HAL 库函数为：

```
HAL_StatusTypeDef HAL_RCCEx_PeriphCLKConfig(
                                        RCC_PeriphCLKInitTypeDef  *PeriphClkInit);
```

该函数是 HAL 库提供的用来配置扩展外设时钟通用函数。LCD_CLK 前面讲解过，它来自 PLLSAI，根据前面讲解的 LCD_CLK 计算公式：

```
f(LCD_CLK)= 1Mhz* PLLSAIN/PLLSAIR/2^(PLLSAIDIVR+1);
```

可知，我们需要配置 PLLSAIN，PLLSAIR 和 PLLSAIDIVR 等参数，具体使用实例如下：

```
RCC_PeriphCLKInitTypeDef PeriphClkIniture;

PeriphClkIniture.PeriphClockSelection=RCC_PERIPHCLK_LTDC;  //LTDC 时钟
PeriphClkIniture.PLLSAI.PLLSAIN= 288;
PeriphClkIniture.PLLSAI.PLLSAIR=4;
PeriphClkIniture.PLLSAIDivR= RCC_PLLSAIDIVR_8;
HAL_RCCEx_PeriphCLKConfig(&PeriphClkIniture);
```

3）设置 RGBLCD 的相关参数，并使能 LTDC。

这一步，我们需要完成对 LCD 面板参数的配置，包括：LTDC 使能、时钟极性、HSW、VSW、HBP、HFP、VBP 和 VFP 等（见表 19.1.2.2），通过 LTDC_GCR、LTDC_SSCR、LTDC_BPCR、LTDC_AWCR 和 LTDC_TWCR 等寄存器配置。HAL 库配置 LTDC 参数并使能 LTDC 的函数为：

```
HAL_StatusTypeDef HAL_LTDC_Init(LTDC_HandleTypeDef *hltdc);
```

该函数只有一个入口参数就是 hltdc,为 LTDC_HandleTypeDef 结构体指针类型。接下来我们看看 LTDC_HandleTypeDef 结构体定义如下：

```
typedef struct
{
  LTDC_TypeDef                *Instance;
  LTDC_InitTypeDef             Init;
  LTDC_LayerCfgTypeDef         LayerCfg[MAX_LAYER];
  HAL_LockTypeDef              Lock;
  __IO HAL_LTDC_StateTypeDef   State;
  __IO uint32_t                ErrorCode;
} LTDC_HandleTypeDef;
```

该结构体有 5 个成员变量。成员变量 Lock 和 State，他们是 HAL 库用来标识一些状态过程的变量，这里就不做过多讲解。Instance 变量是 LTDC_TypeDef 结构体指针类型，和其他初始化结构体成员变量 Instance 一样都是用来设置配置寄存器的基地址，这在 HAL 库中已经通过宏定义定好好了，设置值为 LTDC 即可。成员变量 LayerCfg 是一个数组，它是用来保存 LTDC 层配置参数，下一步骤我们会讲解。最后我们重点看看成员变量 Init，它是真正用来初始化 LTDC 的结构体变量，结构体 LTDC_InitTypeDef 定义如下：

```
typedef struct
{
  uint32_t          HSPolarity;        //水平同步极性
  uint32_t          VSPolarity;        //垂直同步极性
  uint32_t          DEPolarity;        //数据使能极性
  uint32_t          PCPolarity;        //像素时钟极性
  uint32_t          HorizontalSync;    //水平同步宽度
  uint32_t          VerticalSync;      //垂直同步高度
```

```
    uint32_t        AccumulatedHBP;     //水平同步后沿宽度
    uint32_t        AccumulatedVBP;     //垂直同步后沿高度
    uint32_t        AccumulatedActiveW; //累加有效宽度
    uint32_t        AccumulatedActiveH; //累加有效高度
    uint32_t        TotalWidth;         //总宽度
    uint32_t        TotalHeigh;         //总高度
    LTDC_ColorTypeDef   Backcolor;      //屏幕背景层颜色
} LTDC_InitTypeDef;
```

这些参数含义我们都在结构体成员变量之后注释了，具体含义大家可以参考前面第四点配置和状态寄存器讲解。

```
LTDC_HandleTypeDef  LTDC_Handler;       //LTDC 句柄
LTDC_Handler.Instance=LTDC;
LTDC_Handler.Init.HSPolarity=LTDC_HSPOLARITY_AL;        //水平同步极性
LTDC_Handler.Init.VSPolarity=LTDC_VSPOLARITY_AL;        //垂直同步极性
LTDC_Handler.Init.DEPolarity=LTDC_DEPOLARITY_AL;        //数据使能极性
LTDC_Handler.Init.PCPolarity=LTDC_PCPOLARITY_IPC;       //像素时钟极性
LTDC_Handler.Init.HorizontalSync=10-1;              //水平同步宽度
LTDC_Handler.Init.VerticalSync=2-1;                 //垂直同步宽度
LTDC_Handler.Init.AccumulatedHBP=10+20-1;           //水平同步后沿宽度
LTDC_Handler.Init.AccumulatedVBP=2+2-1;             //垂直同步后沿高度
LTDC_Handler.Init.AccumulatedActiveW=10+20+480-1;   //有效宽度
LTDC_Handler.Init.AccumulatedActiveH=2+2+272-1;     //有效高度
LTDC_Handler.Init.TotalWidth=10+20+480+10-1;        //总宽度
LTDC_Handler.Init.TotalHeigh=2+2+272+4-1;           //总高度
LTDC_Handler.Init.Backcolor.Red=0;                  //屏幕背景层红色部分
LTDC_Handler.Init.Backcolor.Green=0;                //屏幕背景层绿色部分
LTDC_Handler.Init.Backcolor.Blue=0;                 //屏幕背景色蓝色部分
HAL_LTDC_Init(&LTDC_Handler); //设置 RGBLCD 的相关参数，并使能 LTDC
```

和其他外设或接口初始化一样，HAL 同样提供了 LTDC 初始化 MSP 回调函数，HAL_LTDC_MspInit，该函数一般用来使能时钟和初始化 IO 口等于 MCU 相关操作：

```
void HAL_LTDC_MspInit(LTDC_HandleTypeDef* hltdc);
```

4）设置 LTDC 层参数。

此步，我们需要设置 LTDC 某一层的相关参数，包括：帧缓存首地址、颜色格式、混合系数和层默认颜色等。通过 LTDC_LxCFBAR、LTDC_LxPFCR、LTDC_LxCACR、LTDC_LxDCCR 和 LTDC_LxBFCR 等寄存器配置。 HAL 库提供的 LTDC 层参数配置函数为：

```
HAL_StatusTypeDef HAL_LTDC_ConfigLayer(LTDC_HandleTypeDef *hltdc,
                                       LTDC_LayerCfgTypeDef *pLayerCfg, uint32_t LayerIdx);
```

基于篇幅考虑，该函数具体的入口参数定义这里我们就不做过多讲解，具体使用方法请参考 19.3 小节函数讲解以及实验工程。

5）设置 LTDC 层窗口，并使能层。

这一步，完成对 LTDC 某个层的显示窗口设置（一般设置为整层显示，不开窗），通过 LTDC_LxWHPCR、LTDC_LxWVPCR、LTDC_LxCFBLR 和 LTDC_LxCFBLNR 等寄存器配置。层使能通过配置 LTDC_LxCR 寄存器的最低位实现，使能层以后，RGBLCD 就可以正常工作了。

HAL 库提供的 LTDC 层窗口配置函数为：

```
HAL_StatusTypeDef HAL_LTDC_SetWindowSize(LTDC_HandleTypeDef *hltdc,
                                         uint32_t XSize, uint32_t YSize, uint32_t LayerIdx);//层窗口尺寸配置

HAL_StatusTypeDef HAL_LTDC_SetWindowPosition(LTDC_HandleTypeDef *hltdc,
                                         uint32_t X0, uint32_t Y0, uint32_t LayerIdx);//层窗口位置配置
```

这两个函数使用方法非常简单，这里我们就不累赘了。

通过以上几个步骤，我们就完成了 LTDC 的配置，可以控制 RGBLCD 显示了。LTDC 我们就给大家介绍到这里，接下来，我们介绍 DMA2D。

20.1.3 DMA2D 简介

为了提高 STM32F767 的图像处理能力，ST 公司设计了一个专用于图像处理的专业 DMA：Chrom-Art Accelerator™，即：DMA2D，通过 DMA2D 对图像进行填充和搬运，可以完全不用 CPU 干预，从而提高效率，减轻 CPU 负担。它可以执行下列操作：

- 用特定颜色填充目标图像的一部分或全部（可用于快速单色填充）
- 将源图像的一部分（或全部）复制到目标图像的一部分（或全部）中（可用于快速图像填充）
- 通过像素格式转换将源图像的一部分（或全部）复制到目标图像的一部分（或全部）中
- 将像素格式不同的两个源图像部分和/ 或全部混合，再将结果复制到颜色格式不同的部分或整个目标图像中。

DMA2D 有四种工作模式，通过 DMA2D_CR 寄存器的 MODE[1:0]位选择工作模式：

1， 寄存器到存储器
2， 存储器到存储器
3， 存储器到存储器并执行 PFC
4， 存储器到存储器并执行 PFC 和混合

本章，我们仅介绍前两种工作模式，后两种工作模式，请大家参考《STM32F7 中文参考手册.pdf》第 9 章。

寄存器到存储器

寄存器到存储器模式用于以预定义颜色填充用户自定义区域，也就是可以实现快速的单色填充显示，比如清屏操作。

在该模式下：颜色格式在 DMA2D_OPFCCR 中设置，DMA2D 不从任何源获取数据，它只是将 DMA2D_OCOLR 寄存器中定义的颜色写入通过 DMA2D_OMA 寻址以及 DMA2D_NLR 和 DMA2D_OOR 定义的区域。

存储器到存储器

该模式下，DMA2D 不执行任何图形数据转换。前景层输入 FIFO 充当缓冲区，数据从 DMA2D_FGMAR 中定义的源存储单元传输到 DMA2D_OMAR 寻址的目标存储单元，可用于快速图像填充。DMA2D_FGPFCCR 寄存器的 CM[3:0]位中编程的颜色模式决定输入和输出的每像素位数。对于要传输的区域大小，源区域大小由 DMA2D_NLR 和 DMA2D_FGOR 寄存器定义，目标区域大小则由 DMA2D_NLR 和 DMA2D_OOR 寄存器定义。

以上两个工作模式，LTDC 在层帧缓存里面的开窗关系都一样，如图 20.1.3.1 所示：

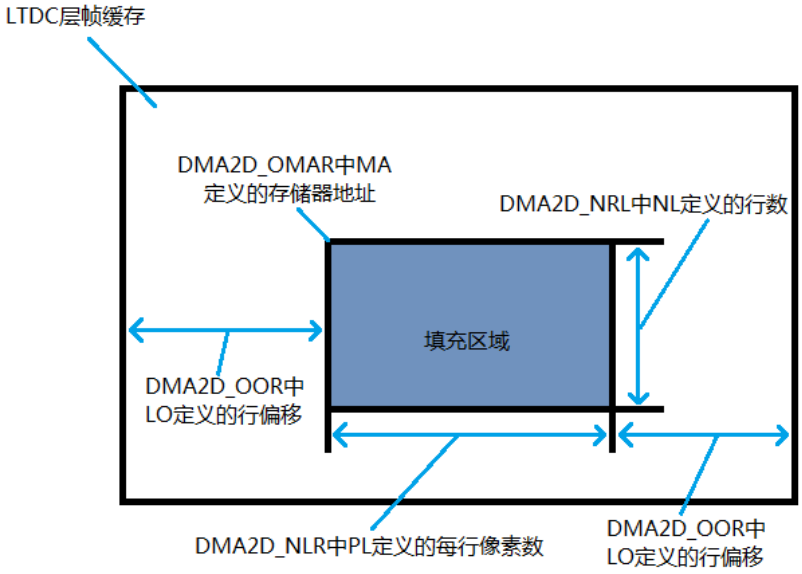


图 20.1.3.1 层帧缓冲开窗示意图

窗口显示区域的显存首地址由 DMA2D_OMAR 寄存器指定，窗口宽度和高度由 DMA2D_NRL 寄存器的 PL 和 NL 指定，行偏移（确定下一行的起始地址）由 DMA2D_OOR 寄存器指定，经过这三个寄存器的配置，就可以确定窗口的显示位置和大小。

在寄存器到存储器模式下，在开窗完成后，DMA2D 可以将 DMA2D_OCOLR 指定的颜色，自动填充到开窗区域，完成单色填充。

在存储器到存储器模式下，需要完成两个开窗：前景层和显示层，完成配置后，图像数据从前景层拷贝到显示层（仅限窗口范围内），从而显示到 LCD 上面。显示层的开窗，如图 20.1.3.1 所示，而前景层的开窗，则和图 20.1.3.1 所示相似，只是 DMA2D_OMAR 寄存器变成了 DMA2D_FGMAR，DMA2D_OOR 寄存器变成了 DMA2D_FGOR，DMA2D_NRL 则两个层共用，然后就可以完成对前景层的开窗，确定好两个窗口后，DMA2D 就将前景层窗口内的数据，拷贝到显示层窗口，完成快速图像填充。

接下来，我们介绍一下 DMA2D 的一些相关寄存器。

首先，我们来看 DMA2D 控制寄存器：DMA2D_CR，该寄存器各位描述如图 20.1.3.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved														MODE	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CEIE	CTCIE	CAEIE	TWIE	TCIE	TEIE	Reserved					ABORT	SUSP	START
		rw	rw	rw	rw	rw	rw						rs	rw	rs

图 20.1.3.2 DMA2D_CR 寄存器各位描述

该寄存器，我们主要关心 MODE 和 START 这两个设置，其中：

MODE：表示 DMA2D 的工作模式，00：存储器到存储器模式；01：存储器到存储器模式并执行 PFC；10：存储器到存储器并执行混合；11，寄存器到存储器模式；本章我们需要用到的设置为：00 或者 11。

START：该位控制 DMA2D 的启动，在配置完成后，设置该位为 1，启动 DMA2D 传输。

接下来，我们介绍 DMA2D 输出 PFC 控制寄存器：DMA2D_OPFCCR，该寄存器各位描述如图 20.1.3.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CM[2:0]		
													rw	rw	rw

图 20.1.3.3 DMA2D_OPFCCR 寄存器各位描述

该寄存器用于设置寄存器到存储器模式下的颜色格式，只有最低 3 位有效（CM[2:0]），表示的颜色格式有：000，ARGB8888；001：RGB888；010：RGB565；011：ARGB1555；100：ARGB1444。我们一般使用的是 RGB565 格式，所以设置 CM[2:0]=010 即可。

同样的，还有前景层 PFC 控制寄存器：DMA2D_FGPFCCR，该寄存器各位描述如图 20.1.3.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ALPHA[7:0]								Reserved						AM[1:0]	
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
CS[7:0]								Reserved		START	CCM	CM[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw			rc_w1	rw	rw	rw	rw	rw

图 20.1.3.4 DMA2D_FGPFCCR 寄存器各位描述

该寄存器，我们只关心最低 4 位：CM[3:0]，用于设置存储器到存储器模式下的颜色格式，这四个位表示的颜色格式为：0000：ARGB8888；0001：RGB888；0010：RGB565；0011：ARGB1555；0100：ARGB4444；0101：L8；0110：AL44；0111：AL88；1000：L4；1001：A8；1010：A4；我们一般使用 RGB565 格式，所以设置 CM[3:0]=0010 即可。

接下来，我们介绍 DMA2D 输出偏移寄存器：DMA2D_OOR，该寄存器各位描述如图 20.1.3.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	LO[13:0]														
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.3.5 DMA2D_OOR 寄存器各位描述

该寄存器仅最低 14 位有效（LO[13:0]），用于设置输出行偏移，作用于显示层，以像素为单位表示。此值用于生成地址。行偏移将添加到各行末尾，用于确定下一行的起始地址，参见图 20.1.3.1。

同样的，还有前景层偏移寄存器：DMA2D_FGOR，该寄存器同 DMA2D_OOR 一样，也是低 14 位有效，用于控制前景层的行偏移，也是用于生成地址，添加到各行末尾，从而确定下一行的起始地址。

接下来，我们介绍 DMA2D 输出存储器地址寄存器 ：DMA2D_OMAR，该寄存器各位描述如图 20.1.3.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MA[31:16]															
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
MA[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.3.6 DMA2D_OMAR 寄存器各位描述

该寄存器设置由 MA[31:0]设置输出存储器地址，也就是输出 FIFO 所存储的数据地址，该地址需要根据开窗的起始坐标来进行设置。以 800*480 的 LCD 屏为例，行长度为 800 像素，假定帧缓存数组为：ltdc_framebuf，我们设置窗口的起始地址为：sx（<800），sy（<480），颜色格式为 RGB565，每个像素 2 个字节，那么 MA 的设置值应该为：

MA[31:0]= framebuf+2*(800*sy+sx)

同样的，还有前景层偏移寄存器：DMA2D_ FGMAR，该寄存器同 DMA2D_OMAR 一样，不过是用于控制前景层的存储器地址，计算方法同 DMA2D_OMAR。

接下来，我们介绍 DMA2D 行数寄存器 ：DMA2D_NLR，该寄存器各位描述如图 20.1.3.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		PL[13:0]													
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NL[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.3.7 DMA2D_NLR 寄存器各位描述

该寄存器用于控制每行的像素和行数，该寄存器的设置对前景层和显示层均有效，通过该寄存器的配置，就可以设置开窗的大小。其中：

NL[15: 0]：设置待传输区域的行数，用于确定窗口的高度。

PL[13: 0]：设置待传输区域的每行像素数，用于确定窗口的宽度。

接下来，我们介绍 DMA2D 输出颜色寄存器 ：DMA2D_OCOLR，该寄存器各位描述如图 20.1.3.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ALPHA[7:0]								RED[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
GREEN[7:0]								BLUE[7:0]							
RED[4:0]					GREEN[5:0]					BLUE[4:0]					
A	RED[4:0]				GREEN[4:0]					BLUE[4:0]					
ALPHA[3:0]				RED[3:0]				GREEN[3:0]				BLUE[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 20.1.3.8 DMA2D_OCOLR 寄存器各位描述

该寄存器用于配置在寄存器到存储器模式下，填充时所用的颜色值，该寄存器是一个 32 位寄存器，可以支持 ARGB8888 格式，也可以支持 RGB565 格式。我们一般使用 RGB565 格式，比如要填充红色，那么直接设置 DMA2D_OCOLR=0XF800 就可以了。

接下来，我们介绍 DMA2D 中断状态寄存器 ：DMA2D_ISR，该寄存器各位描述如图 20.1.3.9

所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										CEIF	CTCIF	CAEIF	TWIF	TCIF	TEIF
										r	r	r	r	r	r

图 20.1.3.9 DMA2D_ISR 寄存器各位描述

该寄存器表示了 DMA2D 的各种状态标识，这里我们只关心 TCIF 位，表示 DMA2D 的传输完成中断标志。当 DMA2D 传输操作完成（仅限数据传输）时此位置 1，表示可以开始下一次 DMA2D 传输了。

另外，还有一个 DMA2D 中断标志清零寄存器：DMA2D_IFCR，用于清除 DMA2D_ISR 寄存器对应位的标志。通过向该寄存器的第 1 位（CTCIF）写 1，可以用于清除 DMA2D_ISR 寄存器的 TCIF 位标志。

最后，我们来看看利用 DMA2D 完成颜色填充，需要哪些步骤。**这里需要说明一下，使用官方提供的 HAL 库 DMA2D 相关库函数进行颜色填充效率极为低下，大量时间浪费在函数的入栈出栈以及过程处理，所以在项目开发中一般都不会使用 DMA2D 库函数进行颜色填充，包括 ST 官方提供的 STEMWIN 实例关于 DMA2D 部分，均采用的寄存器操作。**具体操作步骤如下：

1）使能 DMA2D 时钟，并先停止 DMA2D。

要使用 DMA2D，先得开启其时钟。然后 DMA2D 在配置其相关参数的时候，需要先停止 DMA2D 传输。 使能 DMA2D 时钟和停止 DMA2D 方法为：

```
__HAL_RCC_DMA2D_CLK_ENABLE(); //使能 DM2D 时钟
DMA2D->CR&=~DMA2D_CR_START;   //停止 DMA2D
```

2）设置 DMA2D 工作模式。

通过 DMA2D_CR 寄存器，配置 DMA2D 的工作模式。我们用了寄存器到存储器模式和存储器到存储器这两个模式。

寄存器到存储器模式设置：

```
DMA2D->CR=DMA2D_R2M;                    //寄存器到存储器模式
```

存储器到存储器模式设置：

```
DMA2D->CR= DMA2D_M2M;                   //存储器到存储器模式
```

3）设置 DMA2D 的相关参数。

这一步，我们需要设置：颜色格式、输出窗口、输出存储器地址、前景层地址（仅存储器到存储器模式需要设置）、颜色寄存器（仅寄存器到存储器模式需要设置）等，由：DMA2D_OPFCCR、DMA2D_FGPFCCR、DMA2D_OOR、DMA2D_FGOR 、DMA2D_OMAR、DMA2D_FGMAR 和 DMA2D_NLR 等寄存器进行配置。具体配置过程请参考实验源码。

4）启动 DMA2D 传输。

通过 DMA2D_CR 寄存器配置开启 DMA2D 传输，实现图像数据的拷贝填充，方法为：

```
DMA2D->CR|=DMA2D_CR_START;              //启动 DMA2D
```

5）等待 DMA2D 传输完成，清除相关标识。

最后，在传输过程中，不要再次设置 DMA2D，否则会打乱显示，所以一般在启动 DMA2D 后，需要等待 DMA2D 传输完成（判断 DMA2D_ISR），在传输完成后，清除传输完成标识（设置 DMA2D_IFCR），以便启动下一次 DMA2D 传输。方法为：

```
while((DMA2D->ISR&DMA2D_FLAG_TC)==0) ; //等待传输完成
DMA2D->IFCR|=DMA2D_FLAG_TC;              //清除传输完成标志
```

通过以上几个步骤，我们就完成了 DMA2D 填充，DMA2D 的简介，我们就介绍完了，详细的介绍请大家参考《STM32F7xx 中文参考手册-扩展章节.pdf》第十一章。

20.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） SDRAM

3） LTDC

4） RGBLCD 接口

这里的 1~3，我们在前面的介绍，都已经讲解完毕。所以，我们仅介绍 RGBLCD 接口，RGBLCD 接口在 STM32F767 核心板上，原理图如图 20.2.1 所示：

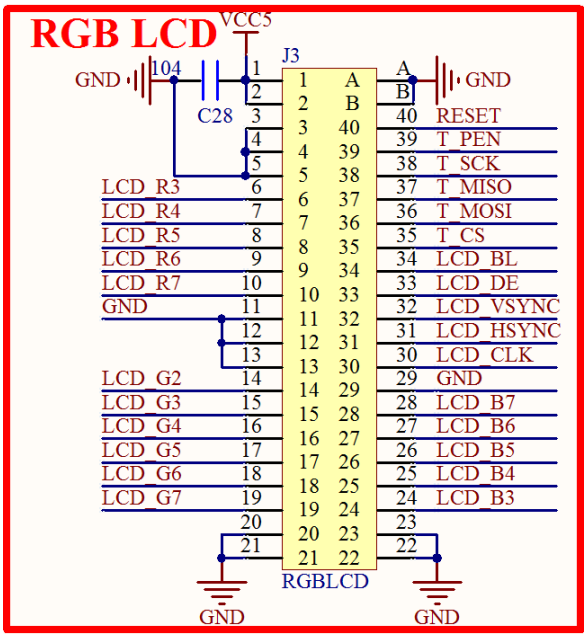


图 20.2.1 RGBLCD 接口原理图

图中 RGB LCD 接口的接线关系见表 20.1.2.1。 这些线的连接，阿波罗 STM32F767 核心板的内部已经连接好了，我们只需要将 RGBLCD 模块通过 40PIN 的 FPC 线连接这个 RGBLCD 接口即可。实物连接（7 寸 RGBLCD 模块）如图 20.2.2 所示：

图 20.2.2 RGBLCD 与开发板连接实物图

20.3 软件设计

打开本章实验工程可以看到，在 USER 分组下面添加了源文件 ltdc.c 并且包含了对应的头文件 ltdc.h，用来存放我们编写的 LTDC 相关驱动函数。

ltdc.c 代码比较多，这里就不贴一一出来了，只针对几个重要的函数进行讲解。完整版的代码见光盘→4，程序源码→标准例程-库函数→实验 15 LTDC LCD（RGB 屏）实验 的 ltdc.c 文件。

本实验，我们用到 LTDC 驱动 RGBLCD，通过前面的介绍，我们知道不同的 RGB 屏，驱动参数有一些差异，为了方便兼容不同的 RGBLCD，我们定义如下 LTDC 参数结构体（在 ltdc.h 里面定义）：

```
//LCD LTDC 重要参数集
typedef struct
{
    u32 pwidth;         //LCD 面板的宽度,固定参数,不随显示方向改变
                        //如果为 0,说明没有任何 RGB 屏接入
    u32 pheight;        //LCD 面板的高度,固定参数,不随显示方向改变
    u16 hsw;            //水平同步宽度
    u16 vsw;            //垂直同步宽度
    u16 hbp;            //水平后廊
    u16 vbp;            //垂直后廊
    u16 hfp;            //水平前廊
    u16 vfp;            //垂直前廊
    u8 activelayer;     //当前层编号:0/1
    u8 dir;             //0,竖屏;1,横屏;
    u16 width;          //LCD 宽度
    u16 height;         //LCD 高度
```

```
    u32 pixsize;            //每个像素所占字节数
}_ltdc_dev;
extern _ltdc_dev lcdltdc;
```

该结构体用于保存一些 RGBLCD 重要参数信息，比如 LCD 面板的长宽、水平后廊和垂直后廊等参数。这个结构体虽然占用了几十个字节的内存，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。

接下来，我们来看两个很重要的数组：

```
//根据不同的颜色格式,定义帧缓存数组
#if   LCD_PIXFORMAT==LCD_PIXFORMAT_ARGB8888||
      LCD_PIXFORMAT==LCD_PIXFORMAT_RGB888
      u32 ltdc_lcd_framebuf[1280][800] __attribute__((at(LCD_FRAME_BUF_ADDR)));
      //定义最大屏分辨率时,LCD 所需的帧缓存数组大小
#else
      u16 ltdc_lcd_framebuf[1280][800] __attribute__((at(LCD_FRAME_BUF_ADDR)));
      //定义最大屏分辨率时,LCD 所需的帧缓存数组大小
#endif
u32 *ltdc_framebuf[2];        //LTDC LCD 帧缓存数组指针,必须指向对应大小的内存区域
```

其中，ltdc_lcd_framebuf 的大小是 LTDC 一帧图像的显存大小，STM32F7 的 LTDC 最大可以支持 1280*800 的 RGB 屏，该数组根据我们选择的颜色格式（ARGB8888/RGB565），自动确定数组类型。另外，我们采用 __attribute__ 关键字，将数组的地址定向到 LCD_FRAME_BUF_ADDR，它在 ltdc.h 里面定义，其值为：0XC0000000，也就是 SDRAM 的首地址。这样，我们就把 ltdc_lcd_framebuf 数组定义到了 SDRAM 的首地址，大小为 800*1280*2 字节（RGB565 格式时）。

而 ltdc_framebuf 则是 LTDC 的帧缓存数组指针，LTDC 支持 2 个层，所以数组大小为 2 。该指针为 32 位类型，必须指向对应的数组，才可以正常使用。在实际使用的时候，我们编写代码：

```
ltdc_framebuf[0]=(u32*)&ltdc_lcd_framebuf;
```

就将 LTDC 第一层的帧缓存，指向了 ltdc_lcd_framebuf 数组。往 ltdc_lcd_framebuf 里面写入不同的数据，就可以修改 RGBLCD 上面显示的内容。

首先，我们来看画点函数：LTDC_Draw_Point，该函数代码如下：

```
//画点函数
//x,y:写入坐标
//color:颜色值
void LTDC_Draw_Point(u16 x,u16 y,u32 color)
{
#if   LCD_PIXFORMAT==LCD_PIXFORMAT_ARGB8888||
      LCD_PIXFORMAT==LCD_PIXFORMAT_RGB888
      if(lcdltdc.dir)         //横屏
      {
          *(u32*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
          (lcdltdc.pwidth*y+x))=color;
      }else                   //竖屏
      {
```

```
            *(u32*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
            (lcdltdc.pwidth*(lcdltdc.pheight-x-1)+y))=color;
        }
    #else
        if(lcdltdc.dir)  //横屏
        {
            *(u16*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
            (lcdltdc.pwidth*y+x))=color;
        }else                    //竖屏
        {
             *(u16*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
             (lcdltdc.pwidth*(lcdltdc.pheight-x-1)+y))=color;
        }
    #endif
    }
```

该函数实现往 RGBLCD 上面画点的功能，根据 LCD_PIXFORMAT 定义的颜色格式以及横竖屏状态，执行不同的操作。RGBLCD 的画点，实际上就是往指定坐标的显存里面写数据，以 7 寸 800*480 的屏幕，RGB565 格式，竖屏模式为例，画某个点对应到屏幕上面的关系如图 19.3.1 所示：

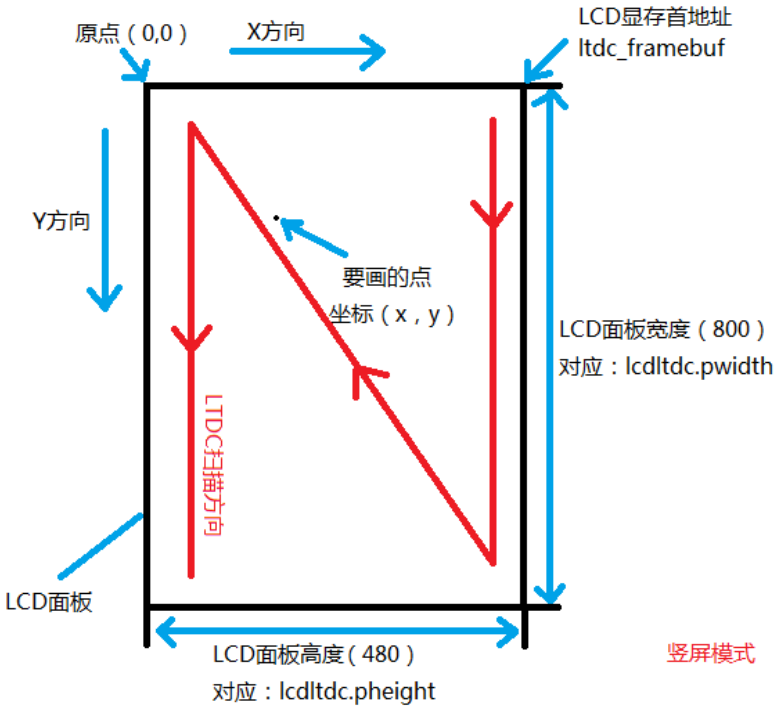


图 19.3.1 画点与 LCD 显存对应关系

注意图中的 LTDC 扫描方向（LTDC 在显存 ltdc_framebuf 里面读取 GRAM 数据的顺序也是这个方向），是从上到下，从右到左，而竖屏的时候，原点在左上角，所以有一个变换过程，经过变换后的画点函数为：

```
*(u16*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*(lcdltdc.pwidth*(lcdltdc.pheight-x-1)+y))=color;
```

其中 ltdc_framebuf，就是层帧缓冲的首地址；lcdltdc.activelayer 表示层编号：0/1 代表第 1/2 层；lcdltdc.pixsize 表示每个像素的字节数，对于 RGB565，它的值为 2；lcdltdc.pwidth 和 lcdltdc.pheight 为 LCD 面板的宽度和高度，lcdltdc.pwidth=800，lcdltdc.pheight=480；x，y 就是要写入显存的坐标（也就是显示在 LCD 上面的坐标）；color 为要写入的颜色值。

有画点函数，就有读点函数，LTDC 的读点函数代码如下：

```
//读点函数
//x,y:读取点的坐标
//返回值:颜色值
u32 LTDC_Read_Point(u16 x,u16 y)
{
#if   LCD_PIXFORMAT==LCD_PIXFORMAT_ARGB8888||
      LCD_PIXFORMAT==LCD_PIXFORMAT_RGB888
      if(lcdltdc.dir)  //横屏
      {
            return *(u32*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
                  (lcdltdc.pwidth*y+x));
      }else                  //竖屏
      {
            return *(u32*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
                  (lcdltdc.pwidth*(lcdltdc.pheight-x-1)+y));
      }
#else
      if(lcdltdc.dir)  //横屏
      {
            return *(u16*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
                  (lcdltdc.pwidth*y+x));
      }else             //竖屏
      {
            return *(u16*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*
                  (lcdltdc.pwidth*(lcdltdc.pheight-x-1)+y));
      }
#endif
}
```

画点函数和读点函数十分类似，只是过程反过来了而已，坐标的计算，也是在 ltdc_framebuf 数组内，根据坐标计算偏移量，完全和读点函数一模一样。

第三个介绍的函数是 LTDC 单色填充函数：LTDC_Fill，该函数使用了 DMA2D 操作，使得填充速度大大加快，该函数代码如下：

```
//LTDC 填充矩形,DMA2D 填充
//(sx,sy),(ex,ey):填充矩形对角坐标,区域大小为:(ex-sx+1)*(ey-sy+1)
//注意:sx,ex,不能大于 lcddev.width-1;sy,ey,不能大于 lcddev.height-1!!!
//color:要填充的颜色
void LTDC_Fill(u16 sx,u16 sy,u16 ex,u16 ey,u32 color)
{
```

```
    u32 psx,psy,pex,pey;      //以 LCD 面板为基准的坐标系,不随横竖屏变化而变化
    u32 timeout=0;
    u16 offline;
    u32 addr;
    //坐标系转换
    if(lcdltdc.dir)  //横屏
    {
        psx=sx;psy=sy;
        pex=ex;pey=ey;
    }else                       //竖屏
    {
        psx=sy;psy=lcdltdc.pheight-ex-1;
        pex=ey;pey=lcdltdc.pheight-sx-1;
    }
    offline=lcdltdc.pwidth-(pex-psx+1);
    addr=((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*(lcdltdc.pwidth*psy+psx));
    __HAL_RCC_DMA2D_CLK_ENABLE(); //使能 DM2D 时钟
    DMA2D->CR&=~(DMA2D_CR_START); //先停止 DMA2D
    DMA2D->CR=DMA2D_R2M;                //寄存器到存储器模式
    DMA2D->OPFCCR=LCD_PIXFORMAT;  //设置颜色格式
    DMA2D->OOR=offline;                 //设置行偏移
    DMA2D->OMAR=addr;                   //输出存储器地址
    DMA2D->NLR=(pey-psy+1)|((pex-psx+1)<<16);      //设定行数寄存器
    DMA2D->OCOLR=color;               //设定输出颜色寄存器
    DMA2D->CR|=DMA2D_CR_START; //启动 DMA2D
    while((DMA2D->ISR&(DMA2D_FLAG_TC))==0)  //等待传输完成
    {
        timeout++;
        if(timeout>0X1FFFFF)break;//超时退出
    }
    DMA2D->IFCR|=DMA2D_FLAG_TC;//清除传输完成标志
}
```

该函数使用 DMA2D 完成矩形色块的填充，其操作步骤，就是按 19.1.3 节最后的介绍来进行的，我们这了就不多说了，详见 19.1.3 接。另外，还有一个 LTDC 彩色填充函数，也是采用的 DMA2D 填充，函数名为 LTDC_Color_Fill，该函数代码同 LTDC_Fill 非常接近，我们这里就不介绍了，请大家参考本例程源码。

第四个介绍的函数是清屏函数：LTDC_Clear，该函数代码如下：

```
//LCD 清屏
//color:颜色值
void LTDC_Clear(u32 color)
{
    LTDC_Fill(0,0,lcdltdc.width-1,lcdltdc.height-1,color);
}
```

该函数代码非常简单，清屏操作调用了我们前面介绍的 LTDC_Fill 函数，采用 DMA2D 完成对 LCD 的清屏，提高了清屏速度。

第五个介绍的函数是 LCD_CLK 频率设置函数：LTDC_Clk_Set，该函数代码如下：

```
//LTDC 时钟(Fdclk)设置函数
//Fvco=Fin*pllsain;
//Fdclk=Fvco/pllsair/2*2^pllsaidivr=Fin*pllsain/pllsair/2*2^pllsaidivr;
//Fvco:VCO 频率
//Fin:输入时钟频率一般为 1Mhz(来自系统时钟 PLLM 分频后的时钟,见时钟树图)
//pllsain:SAI 时钟倍频系数 N,取值范围:50~432.
//pllsair:SAI 时钟的分频系数 R,取值范围:2~7
//pllsaidivr:LCD 时钟分频系数,取值范围:0~3,对应分频 2^(pllsaidivr+1)
//假设:外部晶振为 25M,pllm=25 的时候,Fin=1Mhz.
//例如:要得到 20M 的 LTDC 时钟,则可以设置:pllsain=400,pllsair=5,pllsaidivr=1
//Fdclk=1*396/3/2*2^1=396/12=33Mhz
//返回值:0,成功;1,失败。
u8 LTDC_Clk_Set(u32 pllsain,u32 pllsair,u32 pllsaidivr)
{
    RCC_PeriphCLKInitTypeDef PeriphClkIniture;

    //LTDC 输出像素时钟，需要根据自己所使用的 LCD 数据手册来配置!
    PeriphClkIniture.PeriphClockSelection=RCC_PERIPHCLK_LTDC; //LTDC 时钟
    PeriphClkIniture.PLLSAI.PLLSAIN=pllsain;
    PeriphClkIniture.PLLSAI.PLLSAIR=pllsair;
    PeriphClkIniture.PLLSAIDivR=pllsaidivr;
    if(HAL_RCCEx_PeriphCLKConfig(&PeriphClkIniture)==HAL_OK)      //配置像素时钟
    {
        return 0;     //成功
    }
    else return 1;   //失败
}
```

该函数完成对 PLLSAI 的配置，最终控制输出 LCD_CLK 的频率，LCD_CLK 的频率设置方法，我们在 19.1.2 节进行了介绍，请大家参考前面的介绍进行学习。

第六个介绍的函数是 LTDC 层参数设置函数：LTDC_Layer_Parameter_Config，该函数代码如下：

```
//LTDC,基本参数设置.
//注意:此函数,必须在 LTDC_Layer_Window_Config 之前设置.
//layerx:层值,0/1.
//bufaddr:层颜色帧缓存起始地址
//pixformat:颜色格式.0,ARGB8888;1,RGB888;2,RGB565;3,ARGB1555;
//                    4,ARGB4444;5,L8;6;AL44;7;AL88
//alpha:层颜色 Alpha 值,0,全透明;255,不透明
//alpha0:默认颜色 Alpha 值,0,全透明;255,不透明
//bfac1:混合系数 1,4(100),恒定的 Alpha;6(101),像素 Alpha*恒定 Alpha
```

```
//bfac2:混合系数 2,5(101),恒定的 Alpha;7(111),像素 Alpha*恒定 Alpha
//bkcolor:层默认颜色,32 位,低 24 位有效,RGB888 格式
//返回值:无
void LTDC_Layer_Parameter_Config(u8 layerx,u32 bufaddr,
                                 u8 pixformat,u8 alpha,u8 alpha0,u8 bfac1,u8 bfac2,u32 bkcolor)
{
    LTDC_LayerCfgTypeDef pLayerCfg;

    pLayerCfg.WindowX0=0;                       //窗口起始 X 坐标
    pLayerCfg.WindowY0=0;                       //窗口起始 Y 坐标
    pLayerCfg.WindowX1=lcdltdc.pwidth;          //窗口终止 X 坐标
    pLayerCfg.WindowY1=lcdltdc.pheight;         //窗口终止 Y 坐标
    pLayerCfg.PixelFormat=pixformat;            //像素格式
    pLayerCfg.Alpha=alpha;                      //Alpha 值设置，0~255,255 为完全不透明
    pLayerCfg.Alpha0=alpha0;                    //默认 Alpha 值
    pLayerCfg.BlendingFactor1=(u32)bfac1<<8;    //设置层混合系数
    pLayerCfg.BlendingFactor2=(u32)bfac2<<8;    //设置层混合系数
    pLayerCfg.FBStartAdress=bufaddr;            //设置层颜色帧缓存起始地址
    pLayerCfg.ImageWidth=lcdltdc.pwidth;        //设置颜色帧缓冲区的宽度
    pLayerCfg.ImageHeight=lcdltdc.pheight;      //设置颜色帧缓冲区的高度
    pLayerCfg.Backcolor.Red=(u8)(bkcolor&0X00FF0000)>>16;  //背景颜色红色部分
    pLayerCfg.Backcolor.Green=(u8)(bkcolor&0X0000FF00)>>8; //背景颜色绿色部分
    pLayerCfg.Backcolor.Blue=(u8)bkcolor&0X000000FF;       //背景颜色蓝色部分
    HAL_LTDC_ConfigLayer(&LTDC_Handler,&pLayerCfg,layerx);  //设置所选中的层
}
```

该函数中主要调用 HAL 库函数 HAL_LTDC_ConfigLayer 设置 LTDC 层的基本参数，包括：层帧缓冲区首地址、颜色格式、Alpha 值、混合系数和层默认颜色等，这些参数都需要根据大家的实际需要来进行设置。

第七个介绍的函数是 LTDC 层窗口设置函数：LTDC_Layer_Window_Config，该函数代码如下：

```
//LTDC,层窗口设置,窗口以 LCD 面板坐标系为基准
//layerx:层值,0/1.
//sx,sy:起始坐标
//width,height:宽度和高度
//LTDC,层颜窗口设置,窗口以 LCD 面板坐标系为基准
//注意:此函数必须在 LTDC_Layer_Parameter_Config 之后再设置.
//layerx:层值,0/1.
//sx,sy:起始坐标
//width,height:宽度和高度
void LTDC_Layer_Window_Config(u8 layerx,u16 sx,u16 sy,u16 width,u16 height)
{
    HAL_LTDC_SetWindowPosition(&LTDC_Handler,sx,sy,layerx);  //设置窗口的位置
    HAL_LTDC_SetWindowSize(&LTDC_Handler,width,height,layerx);//设置窗口大小
```

```
}
```

该函数依次调用 HAL 库 LTDC 窗口位置设置函数 HAL_LTDC_SetWindowPositio 和窗口大小设置函数 HAL_LTDC_SetWindowSizel 来控制 LTDC 在某一层（1/2）上面的开窗操作，这个我们在 19.1.2 节也介绍过了，请参考前面的内容进行学习。这里我们一般设置层窗口为整个 LCD 的分辨率，也就是不进行开窗操作。注意：此函数必须在 LTDC_Layer_Parameter_Config 之后再设置。另外，**当设置的窗口值不等于面板的尺寸时，对层 GRAM 的操作(读/写点函数)，也要根据层窗口的宽高来进行修改，否则显示不正常（本例程就未做修改）。**

第八个介绍的函数是 LTDC LCD ID 获取函数：LTDC_PanelID_Read，该函数代码如下：

```
//读取面板参数
//PG6=R7(M0);PI2=G7(M1);PI7=B7(M2);
//M2:M1:M0
//0 :0 :0   //4.3 寸 480*272 RGB 屏,ID=0X4342
//0 :0 :1   //7 寸 800*480 RGB 屏,ID=0X7084
//0 :1 :0   //7 寸 1024*600 RGB 屏,ID=0X7016
//0 :1 :1   //7 寸 1280*800 RGB 屏,ID=0X7018
//1 :0 :0   //8 寸 1024*600 RGB 屏,ID=0X8016
//返回值:LCD ID:0,非法;其他值,ID;
u16 LTDC_PanelID_Read(void)
{
    u8 idx=0;
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOG_CLK_ENABLE();          //使能 GPIOG 时钟
    __HAL_RCC_GPIOI_CLK_ENABLE();          //使能 GPIOI 时钟

    GPIO_Initure.Pin=GPIO_PIN_6;           //PG6
    GPIO_Initure.Mode=GPIO_MODE_INPUT;     //输入
    GPIO_Initure.Pull=GPIO_PULLUP;         //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;    //高速
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);    //初始化

    GPIO_Initure.Pin=GPIO_PIN_2|GPIO_PIN_7; //PI2,7
    HAL_GPIO_Init(GPIOI,&GPIO_Initure);        //初始化

    idx=(u8)HAL_GPIO_ReadPin(GPIOG,GPIO_PIN_6); //读取 M0
    idx|=(u8)HAL_GPIO_ReadPin(GPIOI,GPIO_PIN_2)<<1;//读取 M1
    idx|=(u8)HAL_GPIO_ReadPin(GPIOI,GPIO_PIN_7)<<2;//读取 M2
    if(idx==0)return 0X4342;           //4.3 寸屏,480*272 分辨率
    else if(idx==1)return 0X7084;      //7 寸屏,800*480 分辨率
    else if(idx==2)return 0X7016;      //7 寸屏,1024*600 分辨率
    else if(idx==3)return 0X7018;      //7 寸屏,1280*800 分辨率
    else if(idx==4)return 0X8017;      //8 寸屏,1024*768 分辨率
    else return 0;
}
```

因为 RGBLCD 屏并没有读的功能，所以，一般情况，外接 RGB 屏的时候，MCU 是无法获取屏幕的任何信息的。但是 ALIENTEK 在 RGBLCD 模块上面，利用数据线（R7/G7/B7）做了一个巧妙的设计，可以让 MCU 读取到 RGBLCD 模块的 ID，从而执行不同的初始化，实现对不同分辨率的 RGBLCD 模块的兼容。详细原理见：本章 19.1.1 节，第（3）部分： ALIENTEK RGBLCD 模块的说明。

LTDC_PanelID_Read 函数，就是用这样的方法来读取 M[2:0]的值，并将结果（转换成屏型号了）返回给上一层。

最后要介绍的函数是 LTDC 初始化函数：LTDC_Init，该函数的简化代码如下：

```
//LTDC 初始化函数
void LTDC_Init(void)
{
    u32 tempreg=0;
    u16 lcdid=0;
    lcdid=LTDC_PanelID_Read();          //读取 LCD 面板 ID
    if(lcdid==0X7084)
    {
        lcdltdc.pwidth=800;          //面板宽度,单位:像素
        lcdltdc.pheight=480;         //面板高度,单位:像素
        lcdltdc.hsw=1;               //水平同步宽度
        lcdltdc.vsw=1;               //垂直同步宽度
        lcdltdc.hbp=46;              //水平后廊
        lcdltdc.vbp=23;              //垂直后廊
        lcdltdc.hfp=210;             //水平前廊
        lcdltdc.vfp=22;              //垂直前廊
        LTDC_Clk_Set(396,3,1);       //设置像素时钟  33M(如果开双显需要
                                     //降低 DCLK 到:18.75Mhz   300/4/4,才会比较好)
    }else if(lcdid==0Xxxxx)          //其他面板
    {
        ……//省略部分代码
    }
    //LTDC 配置
    LTDC_Handler.Instance=LTDC;
    LTDC_Handler.Init.HSPolarity=LTDC_HSPOLARITY_AL;     //水平同步极性
    LTDC_Handler.Init.VSPolarity=LTDC_VSPOLARITY_AL;     //垂直同步极性
    LTDC_Handler.Init.DEPolarity=LTDC_DEPOLARITY_AL;     //数据使能极性
    LTDC_Handler.Init.PCPolarity=LTDC_PCPOLARITY_IPC;    //像素时钟极性
    LTDC_Handler.Init.HorizontalSync=lcdltdc.hsw-1;              //水平同步宽度
    LTDC_Handler.Init.VerticalSync=lcdltdc.vsw-1;                //垂直同步宽度
    LTDC_Handler.Init.AccumulatedHBP=lcdltdc.hsw+lcdltdc.hbp-1; //水平同步后沿宽度
    LTDC_Handler.Init.AccumulatedVBP=lcdltdc.vsw+lcdltdc.vbp-1; //垂直同步后沿高度
    LTDC_Handler.Init.AccumulatedActiveW=lcdltdc.hsw+lcdltdc.hbp+lcdltdc.pwidth-1;
    LTDC_Handler.Init.AccumulatedActiveH=lcdltdc.vsw+lcdltdc.vbp+lcdltdc.pheight-1;
    LTDC_Handler.Init.TotalWidth=lcdltdc.hsw+lcdltdc.hbp+lcdltdc.pwidth+lcdltdc.hfp-1;
```

```
    LTDC_Handler.Init.TotalHeigh=lcdltdc.vsw+lcdltdc.vbp+lcdltdc.pheight+lcdltdc.vfp-1;
    LTDC_Handler.Init.Backcolor.Red=0;      //屏幕背景层红色部分
    LTDC_Handler.Init.Backcolor.Green=0;    //屏幕背景层绿色部分
    LTDC_Handler.Init.Backcolor.Blue=0;     //屏幕背景色蓝色部分
    HAL_LTDC_Init(&LTDC_Handler);

    LTDC_Layer_Parameter_Config(0,(u32)ltdc_framebuf[0],LCD_PIXFORMAT,255,0,6,7,
                                                0X000000);//层参数配置
    LTDC_Layer_Window_Config(0,0,0,lcdltdc.pwidth,lcdltdc.pheight);
                    //层窗口配置,以 LCD 面板坐标系为基准,不要随便修改!
    LTDC_Display_Dir(0);            //默认竖屏
    LTDC_Select_Layer(0);           //选择第 1 层
    LCD_LED=1;                      //点亮背光
    LTDC_Clear(0XFFFFFFFF);         //清屏
```

LTDC_Init 的初始化步骤，是按照 19.1.2 节最后介绍的步骤来进行的，该函数先读取 RGBLCD 的 ID，然后根据不同的 RGBLCD 型号，执行不同的面板参数初始化，然后调用 HAL_LTDC_Init 函数来设置 RGBLCD 的相关参数并使能 LTDC，最后配置层参数和层窗口完成对 LTDC 的初始化。注意，代码里面的 lcdltdc.hsw、lcdltdc.vsw、lcdltdc.hbp 等参数的值，均是来自对应 RGBLCD 屏的数据手册，其中 lcdid=0X7084 的配置参数，来自：AT070TN92.pdf。

接下来我们看看头文件 ltdc.h 关键内容：

```
//LCD LTDC 重要参数集
typedef struct
{
    u32 pwidth;         //LCD 面板的宽度,固定参数,如果为 0,说明没有任何 RGB 屏接入
    u32 pheight;        //LCD 面板的高度,固定参数,不随显示方向改变
    u16 hsw;            //水平同步宽度
    u16 vsw;            //垂直同步宽度
    u16 hbp;            //水平后廊
    u16 vbp;            //垂直后廊
    u16 hfp;            //水平前廊
    u16 vfp;            //垂直前廊
    u8 activelayer;     //当前层编号:0/1
    u8 dir;             //0,竖屏;1,横屏;
    u16 width;          //LCD 宽度
    u16 height;         //LCD 高度
    u32 pixsize;        //每个像素所占字节数
}_ltdc_dev;
extern _ltdc_dev lcdltdc;       //管理 LCD LTDC 参数
#define LCD_PIXFORMAT_ARGB8888          0X00            //ARGB8888 格式
…//此处省略部分宏定义标识符
#define LCD_PIXFORMAT_AL88              0X07            //ARGB8888 格式
//////////////////////////////////////////////////////////////////////
```

```
//用户修改配置部分:
//定义颜色像素格式,一般用 RGB565
#define LCD_PIXFORMAT                   LCD_PIXFORMAT_RGB565
#define LTDC_BACKLAYERCOLOR 0X00000000  //定义默认背景层颜色
#define LCD_FRAME_BUF_ADDR  0XC0000000 //LCD 帧缓冲区首地址,在 SDRAM 里面.

void LTDC_Switch(u8 sw);                //LTDC 开关
…//此处省略部分函数声明
void LTDC_Init(void);                   //LTDC 初始化函数
#endif
```

这段代码主要定义了_ltdc_dev 结构体，用于保存 LCD 相关参数，另外，LCD_PIXFORMAT 定义了颜色格式，我们一般使用 RGB565 格式，LCD_FRAME_BUF_ADDR 定义了帧缓存的首地址，我们定义在 SDRAM 的首地址，其他的就不多说了。

以上，就是 ltdc 驱动部分的代码，因为阿波罗 STM32F7 开发板还有 MCU 屏接口，为了可以同时兼容 MCU 屏和 RGB 屏，我们对第 17 章介绍的 lcd.c 部分代码做了小改，添加对 RGB 屏的支持，由于篇幅所限，这里我们只挑几个重点的函数给大家介绍下。

首先读点函数，改为了：

```
//读取个某点的颜色值
//x,y:坐标
//返回值:此点的颜色
u32 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
    if(lcdltdc.pwidth!=0)                             //如果是 RGB 屏
    {
        return LTDC_Read_Point(x,y);
    }
    ……//省略部分代码
}
```

当 lcdltdc.pwidth!=0 的时候，说明接入的是 RGB 屏，所以调用 LTDC_Read_Point 函数，实现读点操作，其他情况，说明是 MCU 屏，执行 MCU 屏的读点操作（代码省略）。

然后是画点函数，改为了：

```
//画点
//x,y:坐标
//POINT_COLOR:此点的颜色
void LCD_DrawPoint(u16 x,u16 y)
{
    if(lcdltdc.pwidth!=0)//如果是 RGB 屏
    {
        LTDC_Draw_Point(x,y,POINT_COLOR);
    }else
    ……//省略部分代码
```

```
}
```

当 lcdltdc.pwidth!=0 的时候，说明接入的是 RGB 屏，所以调用 LTDC_Draw_Point 函数，实现画点操作，其他情况，说明是 MCU 屏，执行 MCU 屏的画点操作（代码省略）。同样的，lcd.c 里面的快速画点函数：LCD_Fast_DrawPoint，在使用 RGB 屏的时候，也是使用 LCD_Fast_DrawPoint 来实现画点操作的

最后，是 LCD 初始化函数，改为：

```
//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)
{
    lcddev.id=LTDC_PanelID_Read();//检查是否有 RGB 屏接入
    if(lcddev.id!=0)
    {
        LTDC_Init();                    //ID 非零,说明有 RGB 屏接入.
    }else
    ……//省略部分代码
}
```

首先通过 LTDC_PanelID_Read 函数，读取 RGBLCD 模块的 ID 值，如果合法，则说明接入了 RGB 屏，调用 LTDC_Init 函数，完成对 LTDC 的初始化。否则的话，执行 MCU 屏的初始化。

在 lcd.c 里面，其他还有一些函数进行了兼容 RGB 屏的修改，这里我们就不一一列举了，请大家参考本例程源码。在完成修改后，我们的例程就可以同时兼容 MCU 屏和 RGB 屏了，且 RGB 屏的优先级较高。

接下来，我们看看主函数内容：

```
int main(void)
{
    u8 x=0;
    u8 lcd_id[12];
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    POINT_COLOR=RED;
    sprintf((char*)lcd_id,"LCD ID:%04X",lcddev.id);//将 LCD ID 打印到 lcd_id 数组。
    while(1)
    {
        switch(x)
        {
            case 0:LCD_Clear(WHITE);break;
```

```
                case 1:LCD_Clear(BLACK);break;
                …//此处省略部分代码
                case 11:LCD_Clear(BROWN);break;
            }
            POINT_COLOR=RED;
            LCD_ShowString(10,40,260,32,32,"Apollo STM32F4/F7");
            LCD_ShowString(10,80,240,24,24,"LTDC TEST");
            LCD_ShowString(10,110,240,16,16,"ATOM@ALIENTEK");
            LCD_ShowString(10,130,240,16,16,lcd_id);        //显示 LCD ID

            LCD_ShowString(10,150,240,12,12,"2016/1/6");
            x++;
            if(x==12)x=0;
            LED0_Toggle;
            delay_ms(1000);
        }
    }
```

该部分代码与第 18 章几乎一摸一样，显示一些固定的字符，字体大小包括 32*16、24*12、16*8 和 12*6 等四种，同时显示 LCD 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。

20.4 下载验证

将程序下载到阿波罗 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 RGBLCD 模块的显示如图 20.4.1 所示：

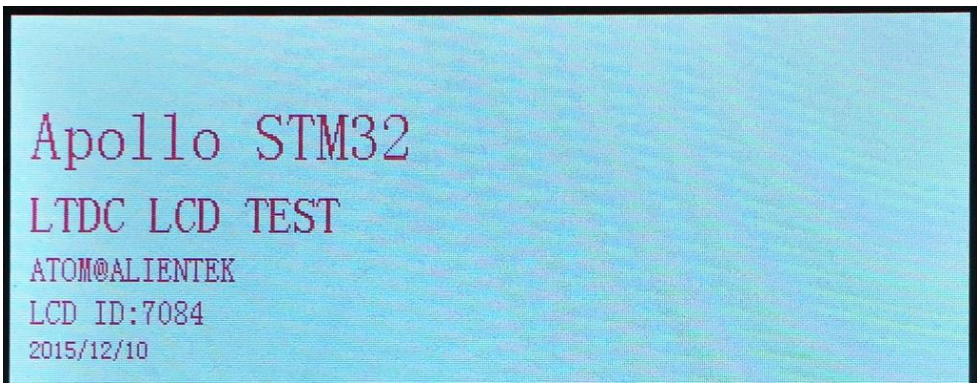


图 20.4.1 RGBLCD 显示效果图

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。最后，需要注意的是：本例程兼容 MCU 屏，所以，当插入 MCU 屏的时候（不插 RGB 屏），也可以显示同样的结果。

第二十一章 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：

21.1 USMART 调试组件简介
21.2 硬件设计
21.3 软件设计
21.4 下载验证

21.1 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 及以后的版本新特性）。
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 21.1.1 所示：

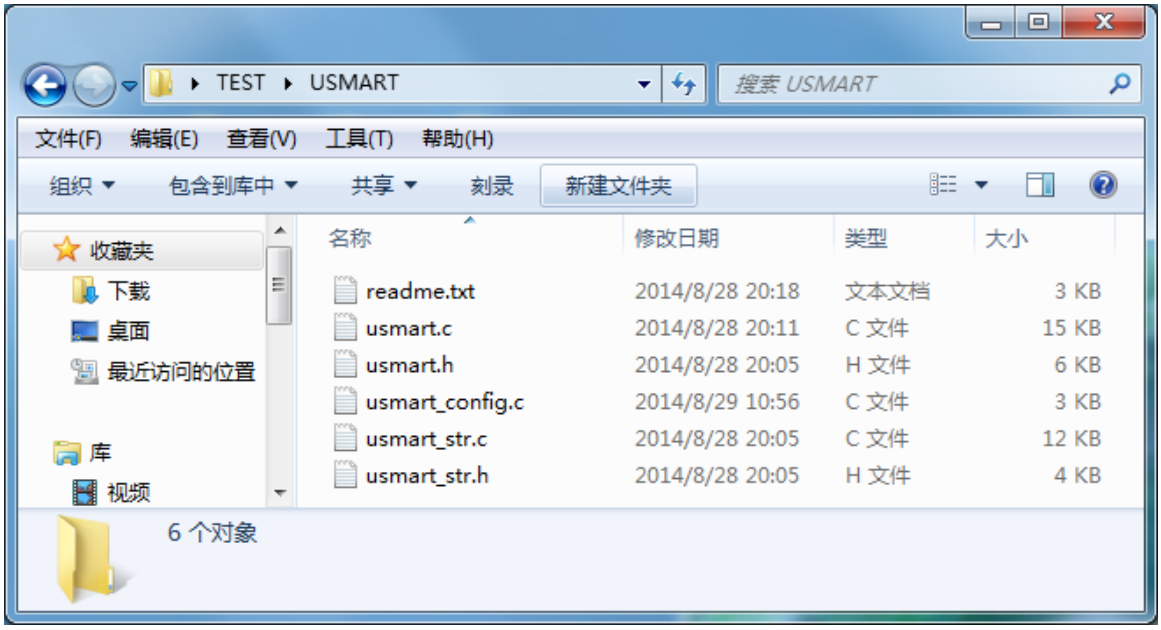


图 21.1.1 USMART 组件代码

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 STM32F7 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.
//如果非 ALIENTEK 用户,则 USART_RX_STA 和 USART_RX_BUF[]需要用户自己实现
void usmart_scan(void)
{
     u8 sta,len;
```

```
    if(USART_RX_STA&0x8000)//串口接收完成?
    {
        len=USART_RX_STA&0x3fff;    //得到此次接收到的数据长度
        USART_RX_BUF[len]='\0';  //在末尾加入结束符.
        sta=usmart_dev.cmd_rec(USART_RX_BUF);//得到函数各个信息
        if(sta==0)usmart_dev.exe();  //执行函数
        else
        {
            len=usmart_sys_cmd_exe(USART_RX_BUF);
            if(len!=USMART_FUNCERR)sta=len;
            if(sta)
            {
                switch(sta)
                {
                    case USMART_FUNCERR:
                        printf("函数错误!\r\n");
                        break;
                    case USMART_PARMERR:
                        printf("参数错误!\r\n");
                        break;
                    case USMART_PARMOVER:
                        printf("参数太多!\r\n");
                        break;
                    case USMART_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，我们将在第二十章实验的基础上，移植 USMART，并通过 USMART 调用一些 LCD 的内部函数，让大家初步了解 USMART 的使用。

21.2 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） LCD 模块（**MCU 屏/RGB 屏都可以，并包括 SDRAM 驱动代码，下同**）

这三个硬件在前面章节均有介绍，本章不再介绍。

21.3 软件设计

我们在上一章实验的基础上添加 USMART 组件相关的支持。打开上一章 LCD 显示实验工程，复制 USMART 文件夹（该文件夹可以在：光盘→标准例程-库函数版本→实验 16 USMART 调试组件实验 里面找到）到 LCD 工程文件夹根目录下面，如图 20.3.1 所示：

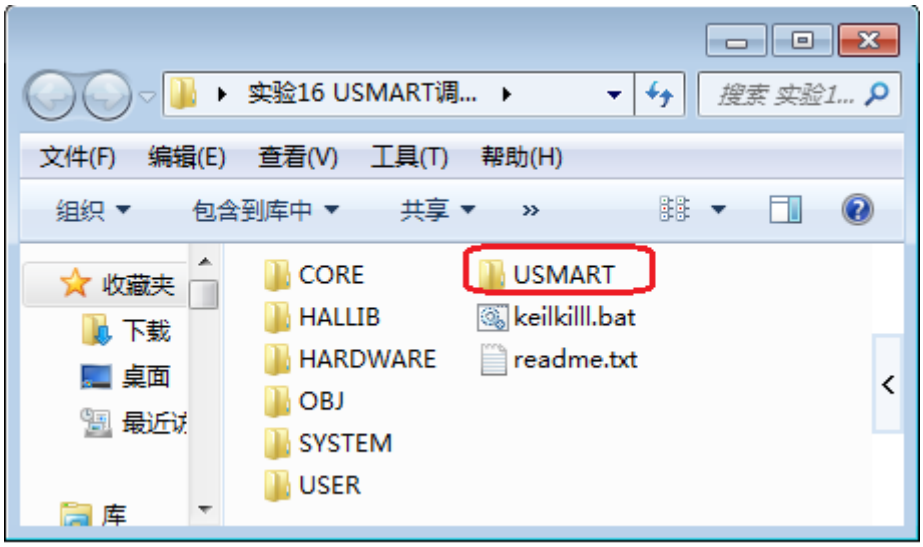


图 20.3.1 复制 USMART 文件夹到工程文件夹下

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 20.3.2 所示：

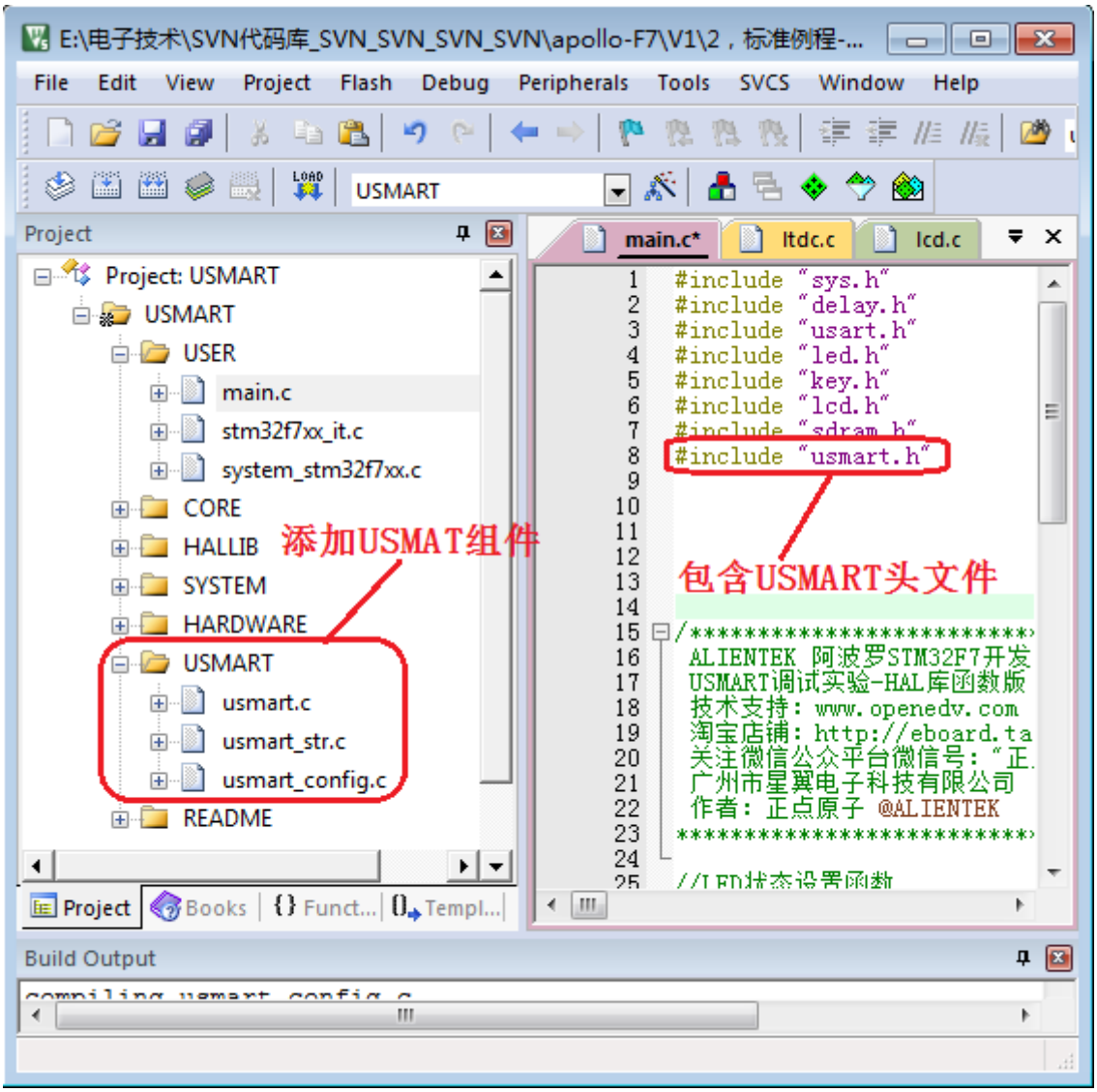


图 20.3.2 添加 USMART 组件代码

由于 USMART 默认提供了 STM32F7 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 20.3.3 所示：

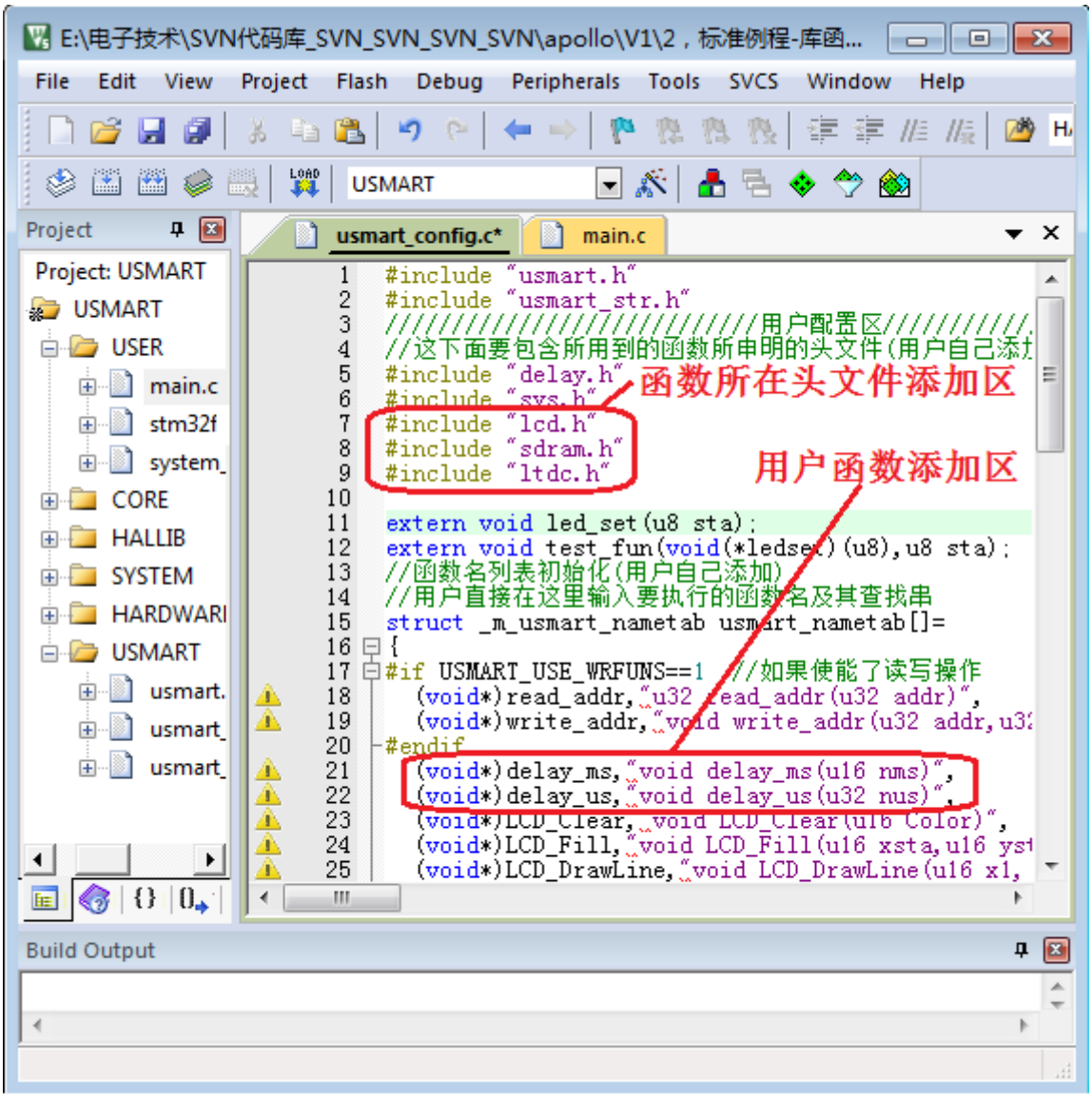


图 20.3.3 添加需要被 USMART 调用的函数

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 20.3.4 所示：

usmart_config.c* main.c

```
1  #include "usmart.h"
2  #include "usmart_str.h"
3  ////////////////////////////用户配置区///////////////////////////
4  //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)
5  #include "delay.h"
6  #include "sys.h"
7  #include "lcd.h"
8  #include "sdram.h"
9  #include "ltdc.h"
10
11 extern void led_set(u8 sta);
12 extern void test_fun(void(*ledset)(u8),u8 sta);
13 //函数名列表初始化(用户自己添加)
14 //用户直接在这里输入要执行的函数名及其查找串
15 struct _m_usmart_nametab usmart_nametab[]=
16 {
17 #if USMART_USE_WRFUNS==1  //如果使能了读写操作
18    (void*)read_addr,"u32 read_addr(u32 addr)",
19    (void*)write_addr,"void write_addr(u32 addr,u32 val)",
20 #endif
21    (void*)delay_ms,"void delay_ms(u16 nms)",
22    (void*)delay_us,"void delay_us(u32 nus)",
23    (void*)LCD_Clear,"void LCD_Clear(u16 Color)",
24    (void*)LCD_Fill,"void LCD_Fill(u16 xsta,u16 ysta,u16 xend,u1
25    (void*)LCD_DrawLine,"void LCD_DrawLine(u16 x1, u16 y1, u16 x
26    (void*)LCD_DrawRectangle,"void LCD_DrawRectangle(u16 x1, u16
27    (void*)LCD_Draw_Circle,"void Draw_Circle(u16 x0,u16 y0,u8 r)
28    (void*)LCD_ShowNum,"void LCD_ShowNum(u16 x,u16 y,u32 num,u8
29    (void*)LCD_ShowString,"void LCD_ShowString(u16 x,u16 y,u16 w
30    (void*)LCD_Fast_DrawPoint,"void LCD_Fast_DrawPoint(u16 x,u16
31    (void*)LCD_ReadPoint,"u16 LCD_ReadPoint(u16 x,u16 y)",
32    (void*)LCD_Display_Dir,"void LCD_Display_Dir(u8 dir)",
33    (void*)LCD_ShowxNum,"void LCD_ShowxNum(u16 x,u16 y,u32 num,u
34
35    (void*)LTDC_Switch,"void LTDC_Switch(u8 sw)",
36    (void*)LTDC_Layer_Switch,"void LTDC_Layer_Switch(u8 layerx,u
37    (void*)LTDC_Select_Layer,"void LTDC_Select_Layer(u8 layerx)"
38    (void*)LTDC_Display_Dir,"void LTDC_Display_Dir(u8 dir)",
39    (void*)LTDC_Draw_Point,"void LTDC_Draw_Point(u16 x,u16 y,u32
40    (void*)LTDC_Read_Point,"u32 LTDC_Read_Point(u16 x,u16 y)",
41    (void*)LTDC_Fill,"void LTDC_Fill(u16 sx,u16 sy,u16 ex,u16 ey
42    (void*)LTDC_Layer_Window_Config,"void LTDC_Layer_Window_Confi
43    (void*)LTDC_Clear,"void LTDC_Clear(u32 color)",
44
45    (void*)led_set,"void led_set(u8 sta)",
46    (void*)test_fun,"void test_fun(void(*ledset)(u8),u8 sta)",
47 };
```

图 20.3.4 添加函数后

上图中，我们添加了 lcd.h，并添加了很多 LCD 函数，最后我们还添加了 led_set 和 test_fun 两个函数，这两个函数在 main.c 里面实现，代码如下：

```
//LED 状态设置函数
void led_set(u8 sta)
{
    LED1(sta);
}
//函数参数调用测试函数
void test_fun(void(*ledset)(u8),u8 sta)
```

```
{
    ledset(sta);
}
```

led_set 函数，用于设置 LED1 的状态，而第二个函数 test_fun 则是测试 USMART 对函数参数的支持的，test_fun 的第一个参数是函数，在 USMART 里面也是可以被调用的。

在添加完函数之后，我们修改 main 函数，如下：

```
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    ……//此处省略部分液晶显示代码
    while(1)
    {
        LED0_Toggle;
        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。

这里大家注意，因为 usmart 要使用串口接收字符，为了保证接收效率和准确率，我们把中断处理过程直接编写在中断服务函数中而没有采用 HAL 库提供的回调函数，具体代码参考 usart.c 文件即可。

21.4 下载验证

将程序下载到阿波罗 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：

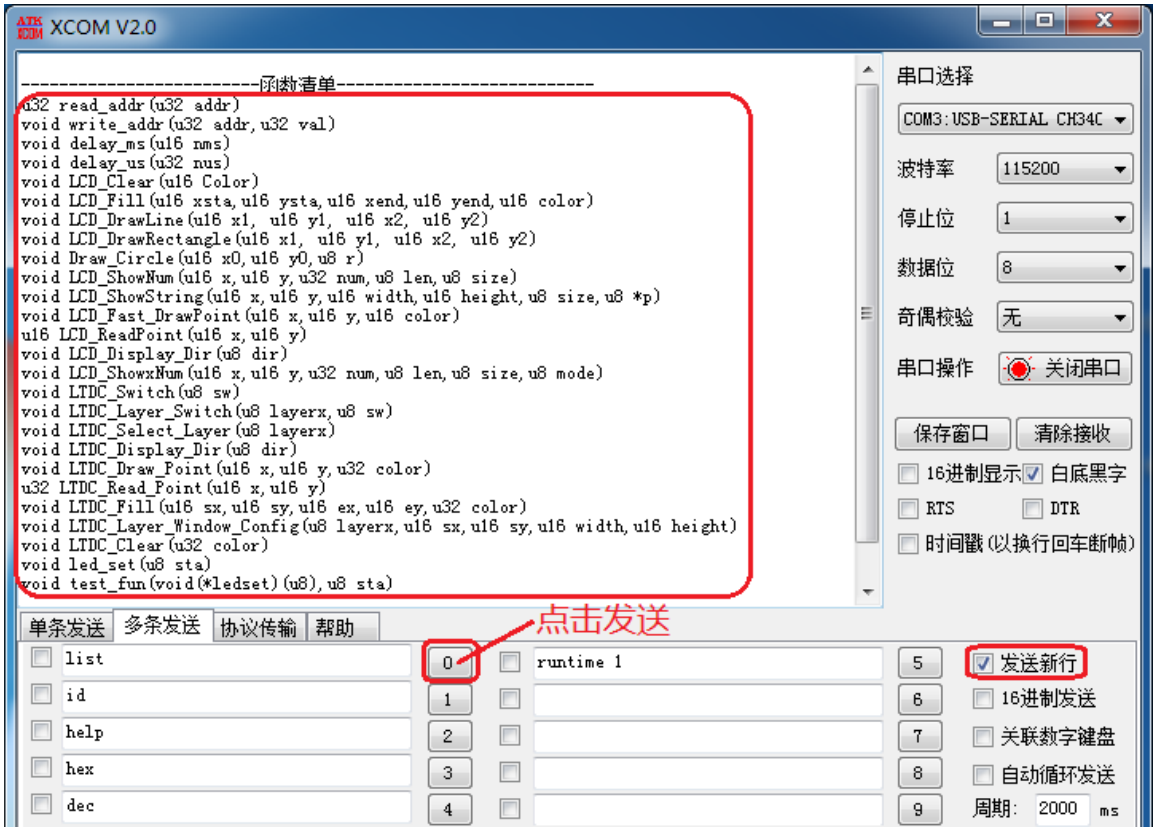


图 21.4.1 驱动串口调试助手

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 20.4.1 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 21.4.2 所示：

图 21.4.2 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 21.4.3 所示：

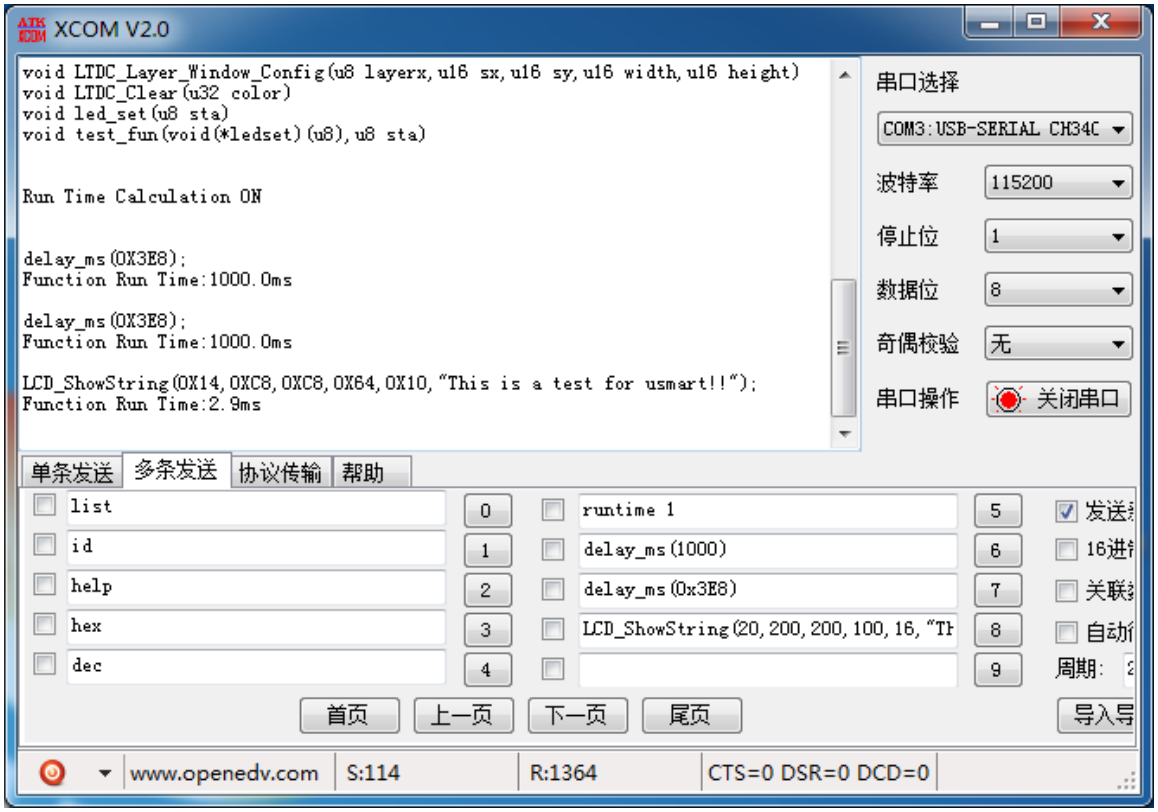


图 21.4.3 串口调用 LCD_ShowString 函数

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：0X0800931D，所以，我们在串口输入：test_fun(0X0800931D,0)，就可以控制 DS1 亮了。如图 21.4.4 所示：

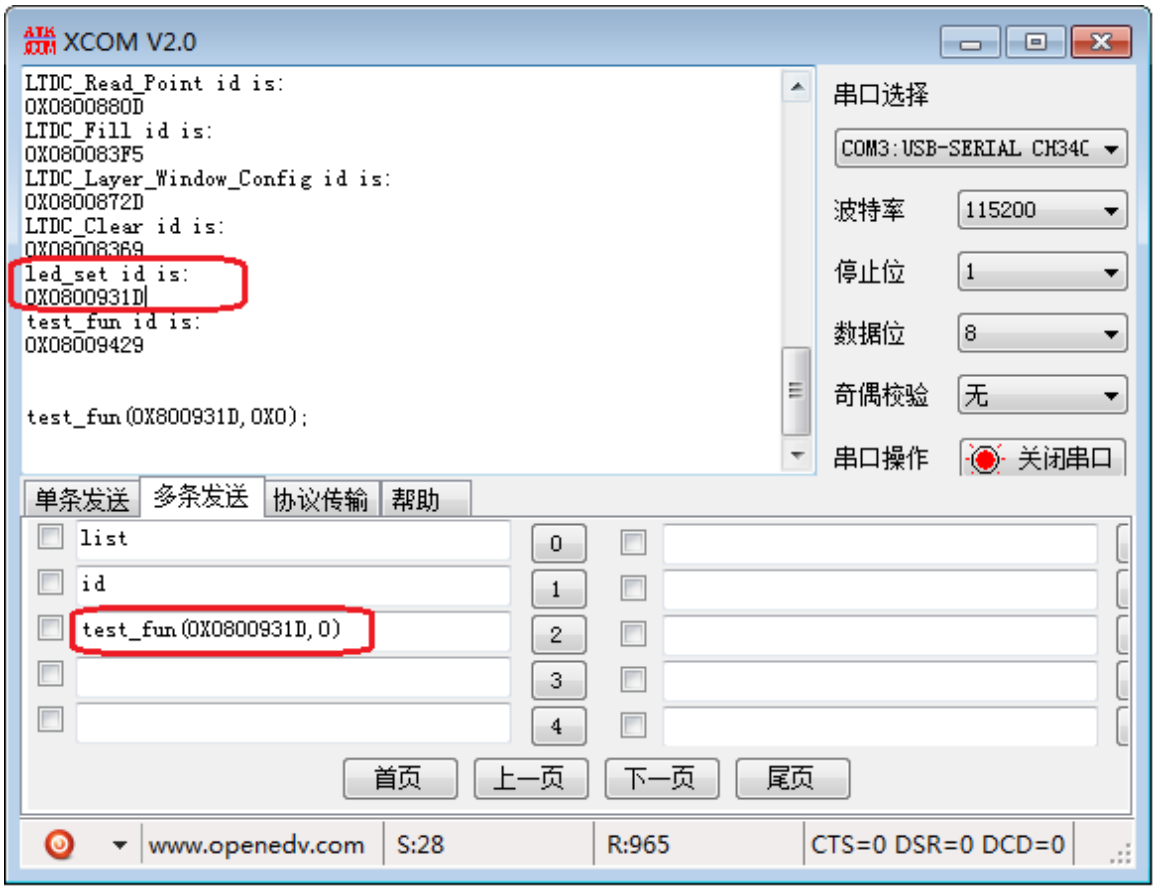


图 21.4.4 串口调用 test_fun 函数

在开发板上，我们可以看到，收到串口发送的 test_fun(0X0800931D,0)后，开发板的 DS1 亮了，然后大家可以通过发送 test_fun(0X0800931D,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。

第二十二章 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 STM32F767 的内部实时时钟（RTC）。在本章中，我们将使用 LCD 模块（MCU 屏或 RGB 屏都可以，下同）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：

22.1 STM32F767 RTC 时钟简介
22.2 硬件设计
22.3 软件设计
22.4 下载验证

22.1 STM32F767 RTC 时钟简介

STM32F767 的实时时钟（RTC）相对于 STM32F1 来说，改进了不少，带了日历功能了，STM32F767 的 RTC，是一个独立的 BCD 定时器/计数器。RTC 提供一个日历时钟（包含年月日时分秒信息）、两个可编程闹钟（ALARM A 和 ALARM B）中断，以及一个具有中断功能的周期性可编程唤醒标志。RTC 还包含用于管理低功耗模式的自动唤醒单元。

两个 32 位寄存器（TR 和 DR）包含二进码十进数格式 (BCD) 的秒、分钟、小时（12 或 24 小时制）、星期、日期、月份和年份。此外，还可提供二进制格式的亚秒值。

STM32F767 的 RTC 可以自动将月份的天数补偿为 28、29（闰年）、30 和 31 天。并且还可以进行夏令时补偿。

RTC 模块和时钟配置是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变，只要后备区域供电正常，那么 RTC 将可以一直运行。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前，先要取消备份区域（BKP）写保护。

RTC 的简化框图，如图 22.1.1 所示：

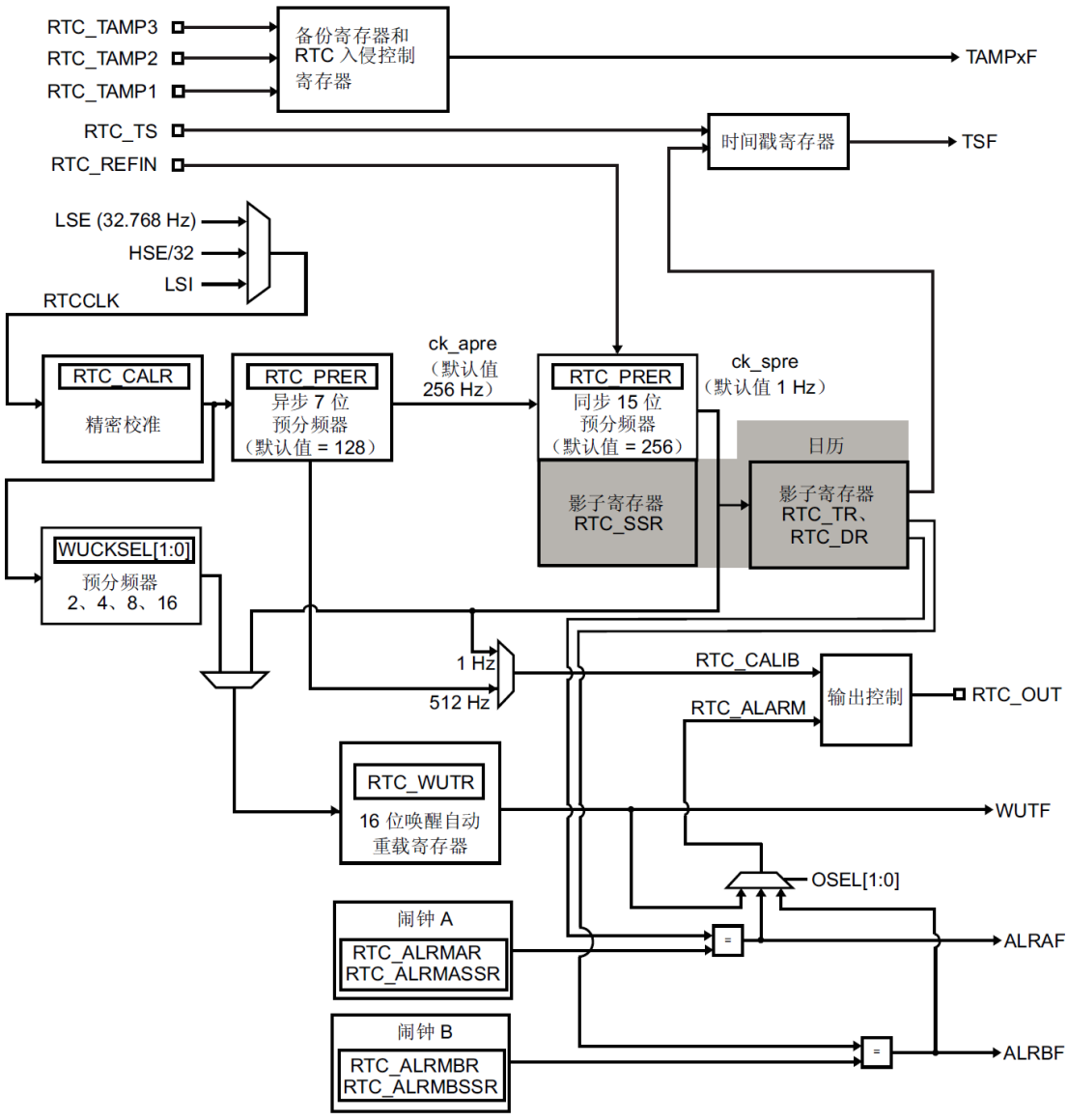


图 22.1.1 RTC 框图

本章我们用到 RTC 时钟和日历，并且用到闹钟功能。接下来简单介绍下 STM32F767 RTC 时钟的使用。

1，时钟和分频

首先，我们看 STM32F767 的 RTC 时钟分频。STM32F767 的 RTC 时钟源（RTCCLK）通过时钟控制器，可以从 LSE 时钟、LSI 时钟以及 HSE 时钟三者中选择（通过 RCC_BDCR 寄存器选择）。一般我们选择 LSE，即外部 32.768Khz 晶振作为时钟源(RTCCLK)，而 RTC 时钟核心，要求提供 1Hz 的时钟，所以，我们要设置 RTC 的可编程预分配器。STM32F767 的可编程预分配器（RTC_PRER）分为 2 个部分：

1， 一个通过 RTC_PRER 寄存器的 PREDIV_A 位配置的 7 位异步预分频器。

2， 一个通过 RTC_PRER 寄存器的 PREDIV_S 位配置的 15 位同步预分频器。

图 22.1.1 中，ck_spre 的时钟可由如下计算公式计算：

Fck_spre=Frtcclk/[(PREDIV_S+1)*(PREDIV_A+1)]

其中，Fck_spre 即可用于更新日历时间等信息。PREDIV_A 和 PREDIV_S 为 RTC 的异步和同步分频器。且推荐设置 7 位异步预分频器（PREDIV_A）的值较大，以最大程度降低功耗。

要设置为 32768 分频，我们只需要设置：PREDIV_A=0X7F，即 128 分频；PREDIV_S=0XFF，即 256 分频，即可得到 1Hz 的 Fck_spre。

另外，图 22.1.1 中，ck_apre 可作为 RTC 亚秒递减计数器（RTC_SSR）的时钟，Fck_apre 的计算公式如下：

Fck_apre=Frtcclk/(PREDIV_A+1)

当 RTC_SSR 寄存器递减到 0 的时候，会使用 PREDIV_S 的值重新装载 PREDIV_S。而 PREDIV_S 一般为 255，这样，我们得到亚秒时间的精度是：1/256 秒，即 3.9ms 左右，有了这个亚秒寄存器 RTC_SSR，就可以得到更加精确的时间数据。

2，日历时间（RTC_TR）和日期（RTC_DR）寄存器

STM32F767 的 RTC 日历时间（RTC_TR）和日期（RTC_DR）寄存器，用于存储时间和日期（也可以用于设置时间和日期），可以通过与 PCLK1（APB1 时钟）同步的影子寄存器来访问，这些时间和日期寄存器也可以直接访问，这样可避免等待同步的持续时间。

每隔 2 个 RTCCLK 周期，当前日历值便会复制到影子寄存器，并置位 RTC_ISR 寄存器的 RSF 位。我们可以读取 RTC_TR 和 RTC_DR 来得到当前时间和日期信息，不过需要注意的是：时间和日期都是以 BCD 码的格式存储的，读出来要转换一下，才可以得到十进制的数据。

3，可编程闹钟

STM32F767 提供两个可编程闹钟：闹钟 A（ALARM_A）和闹钟 B（ALARM_B）。通过 RTC_CR 寄存器的 ALRAE 和 ALRBE 位置 1 来使能闹钟。当日历的亚秒、秒、分、小时、日期分别与闹钟寄存器 RTC_ALRMASSR/RTC_ALRMAR 和 RTC_ALRMBSSR/RTC_ALRMBR 中的值匹配时，则可以产生闹钟（需要适当配置）。本章我们将利用闹钟 A 产生闹铃，即设置 RTC_ALRMASSR 和 RTC_ALRMAR 即可。

4，周期性自动唤醒

STM32F767 的 RTC 不带秒钟中断了，但是多了一个周期性自动唤醒功能。周期性唤醒功能，由一个 16 位可编程自动重载递减计数器（RTC_WUTR）生成，可用于周期性中断/唤醒。

我们可以通过 RTC_CR 寄存器中的 WUTE 位设置使能此唤醒功能。

唤醒定时器的时钟输入可以是：2、4、8 或 16 分频的 RTC 时钟(RTCCLK)，也可以是 ck_spre 时钟（一般为 1Hz）。

当选择 RTCCLK(假定 LSE 是：32.768 kHz)作为输入时钟时，可配置的唤醒中断周期介于 122us（因为 RTCCLK/2 时，RTC_WUTR 不能设置为 0）和 32 s 之间，分辨率最低为：61us。

当选择 ck_spre（1Hz）作为输入时钟时，可得到的唤醒时间为 1s 到 36h 左右，分辨率为 1 秒。并且这个 1s~36h 的可编程时间范围分为两部分：

当 WUCKSEL[2:1]=10 时为：1s 到 18h。

当 WUCKSEL[2:1]=11 时约为：18h 到 36h。

在后一种情况下，会将 2^16 添加到 16 位计数器当前值（即扩展到 17 位，相当于最高位用 WUCKSEL [1]代替）。

初始化完成后，定时器开始递减计数。在低功耗模式下使能唤醒功能时，递减计数保持有效。此外，当计数器计数到 0 时，RTC_ISR 寄存器的 WUTF 标志会置 1，并且唤醒寄存器会使用其重载值（RTC_WUTR 寄存器值）动重载，之后必须用软件清零 WUTF 标志。

通过将 RTC_CR 寄存器中的 WUTIE 位置 1 来使能周期性唤醒中断时，可以使 STM32F767 退出低功耗模式。系统复位以及低功耗模式（睡眠、停机和待机）对唤醒定时器没有任何影响，它仍然可以正常工作，故唤醒定时器，可以用于周期性唤醒 STM32F767。

接下来，我们看看本章我们要用到的 RTC 部分寄存器，首先是 RTC 时间寄存器：RTC_TR，该寄存器各位描述如图 22.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PM	HT[1:0]		HU[3:0]			
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	MNT[2:0]			MNU[3:0]				Reserved	ST[2:0]			SU[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

位 31-24 保留

位 23 保留，必须保持复位值。

位 22 **PM**：AM/PM 符号 (AM/PM notation)
0：AM 或 24 小时制
1：PM

位 21:20 **HT[1:0]**：小时的十位（BCD 格式）(Hour tens in BCD format)

位 16:16 **HU[3:0]**：小时的个位（BCD 格式）(Hour units in BCD format)

位 15 保留，必须保持复位值。

位 14:12 **MNT[2:0]**：分钟的十位（BCD 格式）(Minute tens in BCD format)

位 11:8 **MNU[3:0]**：分钟的个位（BCD 格式）(Minute units in BCD format)

位 7 保留，必须保持复位值。

位 6:4 **ST[2:0]**：秒的十位（BCD 格式）(Second tens in BCD format)

位 3:0 **SU[3:0]**：秒的个位（BCD 格式）(Second units in BCD format)

图 22.1.2 RTC_TR 寄存器各位描述

这个寄存器比较简单，注意数据保存时 BCD 格式的，读取之后需要稍加转换，才是十进制的时分秒等数据，在初始化模式下，对该寄存器进行写操作，可以设置时间。

然后看 RTC 日期寄存器：RTC_DR，该寄存器各位描述如图 22.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								YT[3:0]				YU[3:0]			
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WDU[2:0]			MT	MU[3:0]				Reserved		DT[1:0]		DU[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

位 31-24 保留

位 23:20 **YT[3:0]**：年份的十位（BCD 格式）(Year tens in BCD format)

位 19:16 **YU[3:0]**：年份的个位（BCD 格式）(Year units in BCD format)

位 15:13 **WDU[2:0]**：星期几的个位 (Week day units)
000：禁止
001：星期一
...
111：星期日

位 12 **MT**：月份的十位（BCD 格式）(Month tens in BCD format)

位 11:8 **MU**：月份的个位（BCD 格式）(Month units in BCD format)

位 7:6 保留，必须保持复位值。

位 5:4 **DT[1:0]**：日期的十位（BCD 格式）(Date tens in BCD format)

位 3:0 **DU[3:0]**：日期的个位（BCD 格式）(Date units in BCD format)

图 22.1.3 RTC_DR 寄存器各位描述

同样，该寄存器的的数据采用 BCD 码格式（如不熟悉 BCD，百度即可），其他的就比较简单了。同样，在初始化模式下，对该寄存器进行写操作，可以设置日期。

接下来，看 RTC 亚秒寄存器：RTC_SSR，该寄存器各位描述如图：22.1.4 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 15:0 **SS**：亚秒值 (Sub second value)

SS[15:0] 是同步预分频器计数器的值。此亚秒值可根据以下公式得出：

亚秒值 = (PREDIV_S - SS) / (PREDIV_S + 1)

注意：仅当执行平移操作之后，SS 才能大于 PREDIV_S。在这种情况下，正确的时间/日期比 RTC_TR/RTC_DR 所指示的时间/日期慢一秒钟。

图 22.1.4 RTC_SSR 寄存器各位描述

该寄存器可用于获取更加精确的 RTC 时间。不过，在本章没有用到，如果需要精确时间的地方，大家可以使用该寄存器。

接下来看 RTC 控制寄存器：RTC_CR，该寄存器各位描述如图 22.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	ITSE	COE	OSEL[1:0]		POL	COSEL	BKP	SUB1H	ADD1H
							rw	rw	rw	rw	rw	rw	rw	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSIE	WUTIE	ALRBE	ALRAIE	TSE	WUTE	ALRBE	ALRAE	Res.	FMT	BYPS HAD	REFCKON	TSEDGE	WUCKSEL[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

图 22.1.5 RTC_CR 寄存器各位描述

该寄存器我们不详细介绍每个位了，重点介绍几个要用到的：WUTIE，ALRAIE 是唤醒定时器中断和闹钟 A 中断使能位，本章要用到，设置为 1 即可。WUTE 和 ALRAE，则是唤醒定时器和闹钟 A 定时器使能位，同样设置为 1，开启。FMT 为小时格式选择位，我们设置为 0，选择 24 小时制。最后 WUCKSEL[2:0]，用于唤醒时钟选择，这个前面已经有介绍了，我们这里就不多说了，RTC_CR 寄存器的详细介绍，请看《STM32F7 中文参考手册》第第 29.6.3 节。

接下来看 RTC 初始化和状态寄存器：RTC_ISR，该寄存器各位描述如图 22.1.6 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TAMP 2F	TAMP 1F	TSOVF	TSF	WUTF	ALRBF	ALRAF	INIT	INITF	RSF	INITS	SHPF	WUT WF	ALRB WF	ALRA WF
	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rw	r	rc_w0	r	rc_w0	r	r	r

图 22.1.6 RTC_ISR 寄存器各位描述

该寄存器中，WUTF、ALRBF 和 ALRAF，分别是唤醒定时器闹钟 B 和闹钟 A 的中断标志位，当对应事件产生时，这些标志位被置 1，如果设置了中断，则会进入中断服务函数，这些位通过软件写 0 清除；INIT 为初始化模式控制位，要初始化 RTC 时，必须先设置 INIT=1；INITF 为初始化标志位，当设置 INIT 为 1 以后，要等待 INITF 为 1，才可以更新时间、日期和预分频寄存器等；RSF 位为寄存器同步标志，仅在该位为 1 时，表示日历影子寄存器已同步，可以正确读取 RTC_TR/RTC_TR 寄存器的值了；WUTWF、ALRBWF 和 ALRAWF 分别是唤醒定时器、闹钟 B 和闹钟 A 的写标志，只有在这些位为 1 的时候，才可以更新对应的内容，比如：要设置闹钟 A 的 ALRMAR 和 ALRMASSR，则必须先等待 ALRAWF 为 1，才可以设置。

接下来看 RTC 预分频寄存器：RTC_PRER，该寄存器各位描述如图 22.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PREDIV_A[6:0]						
									rw	rw	rw	rw	rw	rw	rw

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

位 31:24 保留

位 23 保留，必须保持复位值。

位 22:16 **PREDIV_A[6:0]**：异步预分频系数 (Asynchronous prescaler factor)
下面是异步分频系数的公式：
ck_apre 频率 = RTCCLK 频率/(PREDIV_A+1)
注意：PREDIV_A [6:0]= 000000 为禁用值。

位 15 保留，必须保持复位值。

位 14:0 **PREDIV_S[14:0]**：同步预分频系数 (Synchronous prescaler factor)
下面是同步分频系数的公式：
ck_spre 频率 = ck_apre 频率/(PREDIV_S+1)

图 22.1.7 RTC_PRER 寄存器各位描述

该寄存器用于 RTC 的分频，我们在之前也有讲过，这里就不多说了。该寄存器的配置，必须在初始化模式（INITF=1）下，才可以进行。

接下来看 RTC 唤醒定时器寄存器：RTC_WUTR，该寄存器各位描述如图 22.1.8 所示：

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

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

位 31:16 保留

位 15:0 **WUT[15:0]**：唤醒自动重载值位 (Wakeup auto-reload value bit)
当使能唤醒定时器时（WUTE 置 1），每 (WUT[15:0] + 1) 个 ck_wut 周期将 WUTF 标志置 1 一次。ck_wut 周期通过 RTC_CR 寄存器的 WUCKSEL[2:0] 位进行选择。
当 WUCKSEL[2] = 1 时，唤醒定时器变为 17 位，WUCKSEL[1] 等效为 WUT[16]，即要重载到定时器的最高有效位。
注意：WUTF 第一次置 1 发生在 WUTE 置 1 之后 (WUT+1) 个 ck_wut 周期。禁止在 WUCKSEL[2:0]=011(RTCCLK/2) 时将 WUT[15:0] 设置为 0x0000。

图 22.1.8 RTC_WUTR 寄存器各位描述

该寄存器用于设置自动唤醒重装载值，可用于设置唤醒周期。该寄存器的配置，必须等待 RTC_ISR 的 WUTWF 为 1 才可以进行。

接下来看 RTC 闹钟 A 器寄存器：RTC_ALRMAR，该寄存器各位描述如图 22.1.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MSK4	WDSEL	DT[1:0]		DU[3:0]				MSK3	PM	HT[1:0]		HU[3:0]			
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
MSK2	MNT[2:0]			MNU[3:0]				MSK1	ST[2:0]			SU[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31 **MSK4**：闹钟 A 日期掩码 (Alarm A date mask)
0：如果日期/日匹配，则闹钟 A 置 1
1：在闹钟 A 比较中，日期/日无关

位 30 **WDSEL**：星期几选择 (Week day selection)
0：DU[3:0] 代表日期的个位
1：DU[3:0] 代表星期几。DT[1:0] 为无关位。

位 29:28 **DT[1:0]**：日期的十位（BCD 格式）(Date tens in BCD format)。

位 27:24 **DU[3:0]**：日期的个位或日（BCD 格式）(Date units or day in BCD format)。

位 23 **MSK3**：闹钟 A 小时掩码 (Alarm A hours mask)
0：如果小时匹配，则闹钟 A 置 1
1：在闹钟 A 比较中，小时无关

位 22 **PM**：AM/PM 符号 (AM/PM notation)
0：AM 或 24 小时制
1：PM

位 21:20 **HT[1:0]**：小时的十位（BCD 格式）(Hour tens in BCD format)。

位 19:16 **HU[3:0]**：小时的个位（BCD 格式）(Hour units in BCD format)。

位 15 **MSK2**：闹钟 A 分钟掩码 (Alarm A minutes mask)
0：如果分钟匹配，则闹钟 A 置 1
1：在闹钟 A 比较中，分钟无关

位 14:12 **MNT[2:0]**：分钟的十位（BCD 格式）(Minute tens in BCD format)。

位 11:8 **MNU[3:0]**：分钟的个位（BCD 格式）(Minute units in BCD format)。

位 7 **MSK1**：闹钟 A 秒掩码 (Alarm A seconds mask)
0：如果秒匹配，则闹钟 A 置 1
1：在闹钟 A 比较中，秒无关

位 6:4 **ST[2:0]**：秒的十位（BCD 格式）(Second tens in BCD format)。

位 3:0 **SU[3:0]**：秒的个位（BCD 格式）(Second units in BCD format)。

图 22.1.9 RTC_ ALRMAR 寄存器各位描述

该寄存器用于设置闹铃 A，当 WDSEL 选择 1 时，使用星期制闹铃，本章我们选择星期制闹铃。该寄存器的配置，必须等待 RTC_ISR 的 ALRAWF 为 1 才可以进行。另外，还有 RTC_ALRMASSR 寄存器，该寄存器我们这里就不再介绍了，大家参考《STM32F7 中文参考手册》第 29.6.17 节。

接下来看 RTC 写保护寄存器：RTC_WPR，该寄存器比较简单，低八位有效。上电后，所有 RTC 寄存器都受到写保护（RTC_ISR[13:8]、RTC_TAFCR 和 RTC_BKPxR 除外），必须依次写入：0XCA、0X53 两关键字到 RTC_WPR 寄存器，才可以解锁。写一个错误的关键字将再次激活 RTC 的寄存器写保护。

接下来，我们介绍下 RTC 备份寄存器：RTC_BKPxR，该寄存器组总共有 32 个，每个寄存器是 32 位的，可以存储 128 个字节的用户数据，这些寄存器在备份域中实现，可在 VDD 电源关闭时通过 VBAT 保持上电状态。备份寄存器不会在系统复位或电源复位时复位，也不会在 MCU 从待机模式唤醒时复位。

复位后，对 RTC 和 RTC 备份寄存器的写访问被禁止，执行以下操作可以使能对 RTC 及

RTC 备份寄存器的写访问：

1）通过设置寄存器 RCC_APB1ENR 的 PWREN 位来打开电源接口时钟

2）电源控制寄存器(PWR_CR)的 DBP 位来使能对 RTC 及 RTC 备份寄存器的访问。

我们可以用 BKP 来存储一些重要的数据，相当于一个 EEPROM，不过这个 EEPROM 并不是真正的 EEPROM，而是需要电池来维持它的数据。

最后，我们还要介绍一下备份区域控制寄存器 RCC_BDCR。该寄存器的个位描述如图 22.1.10 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	BDRST
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTCEN	Res.	Res.	Res.	Res.	Res.	RTCSEL[1:0]		Res.	Res.	Res.	LSEDRV[1:0]		LSEBYP	LSERDY	LSEON
rw						rw	rw				rw	rw	rw	r	rw

位 16 **BDRST**：备份域软件复位 (Backup domain software reset)

此位由软件置 1 和清零。

0：复位未激活

1：复位整个备份域

注：　BKPSRAM 不受此复位影响，只能在 Flash 保护级别从级别 1 更改为级别 0 时复位 BKPSRAM。

位 15 **RTCEN**：RTC 时钟使能 (RTC clock enable)

此位由软件置 1 和清零。

0：禁止 RTC 时钟

1：使能 RTC 时钟

位 9:8 **RTCSEL[1:0]**：RTC 时钟源选择 (RTC clock source selection)

这些位由软件置 1，用于选择 RTC 的时钟源。选择 RTC 时钟源后，除非备份域复位，否则不可再将其更改。可使用 BDRST 位对其进行复位。

00：无时钟

01：LSE 振荡器时钟用作 RTC 时钟

10：LSI 振荡器时钟用作 RTC 时钟

11：由可编程预分频器分频的 HSE 振荡器时钟（通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 RTCPRE[4:0] 位选择）用作 RTC 时钟

位 4:3 **LSEDRV[1:0]**：LSE 振荡器驱动能力 (LSE oscillator drive capability)

由软件置 1，用于调整 LSE 振荡器的驱动能力。

00：低驱动能力

01：中高驱动能力

10：中低驱动能力

11：高驱动能力

位 2 **LSEBYP**：外部低速振荡器旁路 (External low-speed oscillator bypass)

由软件置 1 和清零，用于旁路振荡器。只有在禁止 LSE 时钟后才能写入该位。

0：不旁路 LSE 振荡器

1：旁路 LSE 振荡器

位 1 **LSERDY**：外部低速振荡器就绪 (External low-speed oscillator ready)

此位由硬件置 1 和清零，用于指示外部 32 kHz 振荡器已稳定。在 LSEON 位被清零后，LSERDY 将在 6 个外部低速振荡器时钟周期后转为低电平。

0：LSE 时钟未就绪

1：LSE 时钟就绪

位 0 **LSEON**：外部低速振荡器使能 (External low-speed oscillator enable)

此位由软件置 1 和清零。

0：LSE 时钟关闭

1：LSE 时钟开启

图 22.1.10 RCC_ BDCR 寄存器各位描述

RTC 的时钟源选择及使能设置都是通过这个寄存器来实现的，所以我们在 RTC 操作之前

先要通过这个寄存器选择 RTC 的时钟源，然后才能开始其他的操作。

RTC 寄存器介绍就给大家介绍到这里了，我们下面来看看要经过哪几个步骤的配置才能使 RTC 正常工作。接下来我们来看看通过 HAL 库配置 RTC 一般配置步骤。RTC 相关的 HAL 库文件为 stm32f7xx_hal_rtc.c 以及头文件 stm32f7xx_hal_rtc.h 中：

1）使能电源时钟，并使能 RTC 及 RTC 后备寄存器写访问。

前面已经介绍了，我们要访问 RTC 和 RTC 备份区域就必须先使能电源时钟，然后使能 RTC 即后备区域访问。电源时钟使能，通过 RCC_APB1ENR 寄存器来设置；RTC 及 RTC 备份寄存器的写访问，通过 PWR_CR 寄存器的 DBP 位设置。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;
  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 HourFormat;      //小时格式
  uint32_t AsynchPrediv;    //异步预分频系数
  uint32_t SynchPrediv;     //同步预分频系数
  uint32_t OutPut;          //选择连接到 RTC_ALARM 输出的标志
  uint32_t OutPutPolarity;  //设置 RTC_ALARM 的输出极性
  uint32_t OutPutType;      //设置 RTC_ALARM 的输出类型为开漏输出还是推挽输出
```

```
}RTC_InitTypeDef;
```

该结构体有 6 个成员变量。

成员变量 HourFormat 用来设置小时格式，为 12 小时制或者 24 小时制，取值为 RTC_HOURFORMAT_12 或者 RTC_HOURFORMAT_24。

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。

SynchPrediv 用来设置 RTC 的同步预分频系数，也就是设置 RTC_PRER 寄存器的 PREDIV_S 相关位，因为同步预分频系数也是 15 位，所以最大值为 0x7FFF，不能超过这个值。

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）。

OutPutPolarity 用来设置 RTC_ALARM 的输出极性，与 Output 成员变量配合使用，取值为 RTC_OUTPUT_POLARITY_HIGH（高电平）或 RTC_OUTPUT_POLARITY_LOW（低电平）。

OutPutType 用来设置 RTC_ALARM 的输出类型为开漏输出（RTC_OUTPUT_TYPE_OPENDRAIN）还是推挽输出（RTC_OUTPUT_TYPE_PUSHPULL），与成员变量 OutPut 和 OutPutPolarity 配合使用。

接下来我们看看 RTC 初始化的一般格式：

```
RTC_Handler.Instance=RTC;
RTC_Handler.Init.HourFormat=RTC_HOURFORMAT_24;//RTC 设置为 24 小时格式
RTC_Handler.Init.AsynchPrediv=0X7F;                    //RTC 异步分频系数(1~0X7F)
RTC_Handler.Init.SynchPrediv=0XFF;                     //RTC 同步分频系数(0~7FFF)
RTC_Handler.Init.OutPut=RTC_OUTPUT_DISABLE;
RTC_Handler.Init.OutPutPolarity=RTC_OUTPUT_POLARITY_HIGH;
RTC_Handler.Init.OutPutType=RTC_OUTPUT_TYPE_OPENDRAIN;
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;
  uint8_t TimeFormat;
```

```
    uint32_t SubSeconds;
    uint32_t SecondFraction;
    uint32_t DayLightSaving;
    uint32_t StoreOperation;
}RTC_TimeTypeDef;
```

前面四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，以及 AM/PM 符号，大家参考前面讲解的 RTC_TR 的位描述即可。SubSeconds 用来读取保存亚秒寄存器 RTC_SSR 的值，SecondFraction 用来读取保存同步预分频系数 的值，也就是 RTC_PRER 的位 0~14，DayLightSaving 用来设置日历时间增加 1 小时，减少 1 小时，还是不变。StoreOperation 用户可对此变量设置以记录是否已对夏令时进行更改。HAL_RTC_SetTime 函数参考实例如下：

```
    RTC_TimeTypeDef RTC_TimeStructure;
    RTC_TimeStructure.Hours=1;
    RTC_TimeStructure.Minutes=1;
    RTC_TimeStructure.Seconds=1;
    RTC_TimeStructure.TimeFormat= RTC_HOURFORMAT12_PM;
    RTC_TimeStructure.DayLightSaving=RTC_DAYLIGHTSAVING_NONE;
    RTC_TimeStructure.StoreOperation=RTC_STOREOPERATION_RESET;
    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。

22.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） 串口

3） LCD 模块（MCU 屏/RGB 屏都可以，下同）

4） RTC

前面 3 个都介绍过了，而 RTC 属于 STM32F767 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 阿波罗 STM32F767 开发板标配是有电池的）。

22.3 软件设计

打开本章实验工程可以看到，我们先在 HALLIB 下面引入了 RTC 支持的库函数文件 stm32f7xx_hal_rtc.c 和 stm32f7xx_hal_rtc_ex.c。然后我们在 HARDWARE 文件夹下新建了一个 rtc.c 的文件和 rtc.h 的头文件，同时将这两个文件引入我们的工程 HARDWARE 分组下。

由于篇幅所限，rtc.c 中的代码，我们不全部贴出了，这里针对几个重要的函数，进行简要说明，首先是 RTC_Init，其代码如下：

```
//RTC 初始化
//返回值:0,初始化成功; 2,进入初始化模式失败;
u8 RTC_Init(void)
{
    RTC_Handler.Instance=RTC;
    RTC_Handler.Init.HourFormat=RTC_HOURFORMAT_24;//RTC 设置为 24 小时格式
    RTC_Handler.Init.AsynchPrediv=0X7F;                 //RTC 异步分频系数(1~0X7F)
    RTC_Handler.Init.SynchPrediv=0XFF;                  //RTC 同步分频系数(0~7FFF)
    RTC_Handler.Init.OutPut=RTC_OUTPUT_DISABLE;
    RTC_Handler.Init.OutPutPolarity=RTC_OUTPUT_POLARITY_HIGH;
    RTC_Handler.Init.OutPutType=RTC_OUTPUT_TYPE_OPENDRAIN;
    if(HAL_RTC_Init(&RTC_Handler)!=HAL_OK) return 2;

    if(HAL_RTCEx_BKUPRead(&RTC_Handler,RTC_BKP_DR0)!=0X5050)
                                                        //是否第一次配置
    {
        RTC_Set_Time(16,13,0,RTC_HOURFORMAT12_PM); //设置时间
        RTC_Set_Date(16,1,13,3);                            //设置日期
        HAL_RTCEx_BKUPWrite(&RTC_Handler,RTC_BKP_DR0,0X5050);
                                                        //标记已经初始化过了
    }
    return 0;
```

```
}
```

该函数用来初始化 RTC 配置以及日期和时钟，但是只在第一次的时候设置时间，以后如果重新上电/复位都不会再进行时间设置了（前提是备份电池有电）。在第一次配置的时候，我们是按照上面介绍的 RTC 初始化步骤调用函数(HAL_RTC_Init 来实现的，这里就不在多说了。

这里设置时间和日期，分别是通过 RTC_Set_Time 和 RTC_Set_Date 函数来实现的，这两个函数实际就是调用库函数里面的 HAL_RTC_SetTime 函数和 HAL_RTC_SetDate 函数来实现，这里我们之所以要写两个这样的函数，目的是为了我们的 USMART 来调用，方便直接通过 USMART 来设置时间和日期。

这里默认将时间设置为 15 年 12 月 27 日星期天，23 点 59 分 56 秒。在设置好时间之后，我们调用函数 HAL_RTCEx_BKUPWrite 向 RTC 的 BKR 寄存器（地址 0）写入标志字 0X5050，用于标记时间已经被设置了。这样，再次发生复位的时候，该函数通过调用函数 HAL_RTCEx_BKUPRead 判断 RTC 对应 BKR 地址的值，来决定是不是需要重新设置时间，如果不需要设置，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。

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。

接着，我们介绍一下 RTC_Set_AlarmA 函数，该函数代码如下：

```
//设置闹钟时间(按星期闹铃,24 小时制)
//week:星期几(1~7) @ref    RTC_WeekDay_Definitions
//hour,min,sec:小时,分钟,秒钟
void RTC_Set_AlarmA(u8 week,u8 hour,u8 min,u8 sec)
{
    RTC_AlarmTypeDef RTC_AlarmSturuct;

    RTC_AlarmSturuct.AlarmTime.Hours=hour;      //小时
    RTC_AlarmSturuct.AlarmTime.Minutes=min;     //分钟
    RTC_AlarmSturuct.AlarmTime.Seconds=sec;     //秒
    RTC_AlarmSturuct.AlarmTime.SubSeconds=0;
    RTC_AlarmSturuct.AlarmTime.TimeFormat=RTC_HOURFORMAT12_AM;

    RTC_AlarmSturuct.AlarmMask=RTC_ALARMMASK_NONE;//精确匹配星期，时分秒
    RTC_AlarmSturuct.AlarmSubSecondMask=RTC_ALARMSUBSECONDMASK_NONE;
    RTC_AlarmSturuct.AlarmDateWeekDaySel=
                            RTC_ALARMDATEWEEKDAYSEL_WEEKDAY;//按星期
    RTC_AlarmSturuct.AlarmDateWeekDay=week; //星期
    RTC_AlarmSturuct.Alarm=RTC_ALARM_A;       //闹钟 A
    HAL_RTC_SetAlarm_IT(&RTC_Handler,&RTC_AlarmSturuct,RTC_FORMAT_BIN);

    HAL_NVIC_SetPriority(RTC_Alarm_IRQn,0x01,0x02); //抢占优先级 1,子优先级 2
```

```
        HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
    }
```

该函数用于设置闹钟 A，也就是设置 ALRMAR 和 ALRMASSR 寄存器的值，来设置闹钟时间，这里 HAL 库中用来设置闹钟并开启闹钟中断的函数为：

```
HAL_StatusTypeDef HAL_RTC_SetAlarm_IT(RTC_HandleTypeDef *hrtc,
                                      RTC_AlarmTypeDef *sAlarm, uint32_t Format);
```

第三个参数 RTC_Format 用来设置格式，这里前面我们讲解过，就不做过多讲解。

接下来我们着重看看第二个参数 sAlarm，该入口参数是 RTC_AlarmTypeDef 结构体指针类型，结构体定义如下：

```
 typedef struct
{
  RTC_TimeTypeDef AlarmTime;
  uint32_t AlarmMask;
  uint32_t AlarmSubSecondMask;
  uint32_t AlarmDateWeekDaySel;
  uint8_t AlarmDateWeekDay;
  uint32_t Alarm;
}RTC_AlarmTypeDef;
```

该结构体有 6 个成员变量，第一个成员变量 AlarmTime 用来设置闹钟时间，是 RTC_TimeTypeDef 结构体类型，该结构体前面我们已经讲解过各个成员变量含义，这里我们就不做过多讲解。

AlarmMask 用来设置闹钟时间掩码，也就是在我们第一个参数设置的时间中（包括后面参数 RTC_AlarmDateWeekDay 设置的星期几/哪一天），哪些是无关的。 比如我们设置闹钟时间为每天的 10 点 10 分 10 秒，那么我们可以选择值 RTC_AlarmMask_DateWeekDay，也就是我们不关心是星期几/每月哪一天。这里我们选择为 RTC_AlarmMask_None，也就是精确匹配时间，所有的时分秒以及星期几/(或者每月哪一天)都要精确匹配。

AlarmSubSecondMask 和 AlarmMask 作用类似，只不过该变量是用来设置亚秒的。

AlarmDateWeekDaySel 用来选择是闹钟是按日期还是按星期。比如我们选择 RTC_AlarmDateWeekDaySel_WeekDay 那么闹钟就是按星期。如果我们选择 RTC_AlarmDateWeekDaySel_Date 那么闹钟就是按日期。这与后面第四个参数是有关联的，我们在后面第四个参数讲解。

AlarmDateWeekDay 用来设置闹钟的日期或者星期几。比如我们第三个参数 RTC_AlarmDateWeekDaySel 设置了值为 RTC_AlarmDateWeekDaySel_WeekDay,也就是按星期，那么参数 RTC_AlarmDateWeekDay 的取值范围就为星期一~星期天，也就是 RTC_Weekday_Monday~RTC_Weekday_Sunday。如果第三个参数 RTC_AlarmDateWeekDaySel 设置值为 RTC_AlarmDateWeekDaySel_Date，那么它的取值范围就为日期值，0~31。

Alarm 用来设置是闹钟 A 还是闹钟 B，这个很好理解。

调用函数 RTC_SetAlarm 设置闹钟 A 的参数之后，最后，开启闹钟 A 中断（连接在外部中断线 17），并设置中断分组。当 RTC 的时间和闹钟 A 设置的时间完全匹配时，将产生闹钟中断。

接着，我们介绍一下 RTC_Set_WakeUp 函数，该函数代码如下：

```
void RTC_Set_WakeUp(u32 wksel,u16 cnt)
{
    __HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&RTC_Handler,
```

```
                                    RTC_FLAG_WUTF);//清除 RTC WAKE UP 的标志
    HAL_RTCEx_SetWakeUpTimer_IT(&RTC_Handler,cnt,wksel); //设置重装载值和时钟

    HAL_NVIC_SetPriority(RTC_WKUP_IRQn,0x02,0x02); //抢占优先级 1,子优先级 2
    HAL_NVIC_EnableIRQ(RTC_WKUP_IRQn);
}
```

该函数用于设置 RTC 周期性唤醒定时器，实现周期性唤醒中断，连接在外部中断线 22。该函数调用的是 HAL 库函数 HAL_RTCEx_SetWakeUpTimer_IT 实现的，该函数使用方法比较简单，这里我们就不做过多讲解。

有了中断设置函数，就必定有中断服务函数，同时因为 HAL 库会开放中断处理回调函数，接下来看这两个中断的中断服务函数和中断处理回调函数，代码如下：

```
//RTC 闹钟中断服务函数
void RTC_Alarm_IRQHandler(void)
{
    HAL_RTC_AlarmIRQHandler(&RTC_Handler);
}

//RTC WAKE UP 中断服务函数
void RTC_WKUP_IRQHandler(void)
{
    HAL_RTCEx_WakeUpTimerIRQHandler(&RTC_Handler);
}

//RTC 闹钟 A 中断处理回调函数
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("ALARM A!\r\n");
}

//RTC WAKE UP 中断处理回调函数
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef *hrtc)
{
    LED1_Toggle;
}
```

其中，RTC_Alarm_IRQHandler 函数用于闹钟中断，其中断控制逻辑写在中断回调函数 HAL_RTC_AlarmAEventCallback 中，每当闹钟 A 闹铃时，会从串口打印一个字符串“ALARM A!”。RTC_WKUP_IRQHandler 函数用于 RTC 自动唤醒定时器中断，其中断控制逻辑写在中断回调函数 HAL_RTCEx_WakeUpTimerEventCallback 中，可以通过观察 LED1 的状态来查看 RTC 自动唤醒中断的情况。

rtc.c 的其他程序，这里就不再介绍了，请大家直接看光盘的源码。rtc.h 头文件中主要是一些函数声明，我们就不多说了，有些函数在这里没有介绍，请大家参考本例程源码。

最后我们看看 main 函数源码如下：

```
int main(void)
```

```
{
    RTC_TimeTypeDef RTC_TimeStruct;
    RTC_DateTypeDef RTC_DateStruct;
    u8 tbuf[40];
    u8 t=0;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    RTC_Init();                     //初始化 RTC
    RTC_Set_WakeUp(RTC_WAKEUPCLOCK_CK_SPRE_16BITS,0);
                                    //配置 WAKE UP 中断,1 秒钟中断一次
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/7/12");
    while(1)      {
        t++;
        if((t%10)==0) //每 100ms 更新一次显示数据
        {
            HAL_RTC_GetTime(&RTC_Handler,&RTC_TimeStruct,RTC_FORMAT_BIN);
            sprintf((char*)tbuf,"Time:%02d:%02d:%02d",RTC_TimeStruct.Hours,
                    RTC_TimeStruct.Minutes,RTC_TimeStruct.Seconds);
            LCD_ShowString(30,140,210,16,16,tbuf);
            HAL_RTC_GetDate(&RTC_Handler,&RTC_DateStruct,RTC_FORMAT_BIN);
            sprintf((char*)tbuf,"Date:20%02d-%02d-%02d",RTC_DateStruct.Year,
                    RTC_DateStruct.Month,RTC_DateStruct.Date);
            LCD_ShowString(30,160,210,16,16,tbuf);
            sprintf((char*)tbuf,"Week:%d",RTC_DateStruct.WeekDay);
            LCD_ShowString(30,180,210,16,16,tbuf);
        }
        if((t%20)==0)LED0_Toggle;       //每 200ms,翻转一次 LED0
        delay_ms(10);
    }
}
```

这部分代码，也比较简单，注意，我们通过

RTC_Set_WakeUp(RTC_WAKEUPCLOCK_CK_SPRE_16BITS,0)设置 RTC 周期性自动唤醒周期为 1 秒钟，类似于 STM32F1 的秒钟中断。然后，在 main 函数不断的读取 RTC 的时间和日期（每 100ms 一次），并显示在 LCD 上面。

为了方便设置时间，我们在 usmart_config.c 里面，修改 usmart_nametab 如下：

```
struct _m_usmart_nametab usmart_nametab[]=
{
#if USMART_USE_WRFUNS==1      //如果使能了读写操作
    (void*)read_addr,"u32 read_addr(u32 addr)",
    (void*)write_addr,"void write_addr(u32 addr,u32 val)",
#endif
    (void*)RTC_Set_Time,"u8 RTC_Set_Time(u8 hour,u8 min,u8 sec,u8 ampm)",
    (void*)RTC_Set_Date,"u8 RTC_Set_Date(u8 year,u8 month,u8 date,u8 week)",
    (void*)RTC_Set_AlarmA,"void RTC_Set_AlarmA(u8 week,u8 hour,u8 min,u8 sec)",
    (void*)RTC_Set_WakeUp,"void RTC_Set_WakeUp(u8 wksel,u16 cnt)",
};
```

将 RTC 的一些相关函数加入了 usmart，这样通过串口就可以直接设置 RTC 时间、日期、闹钟 A 和周期性唤醒等操作。

至此，RTC 实时时钟的软件设计就完成了，接下来就让我们来检验一下，我们的程序是否正确了。

22.4 下载验证

将程序下载到阿波罗 STM32F7 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了，同时 DS1 每隔一秒钟亮一次，说明周期性唤醒中断工作正常。然后，可以看到 LCD 模块开始显示时间，实际显示效果如图 22.4.1 所示：

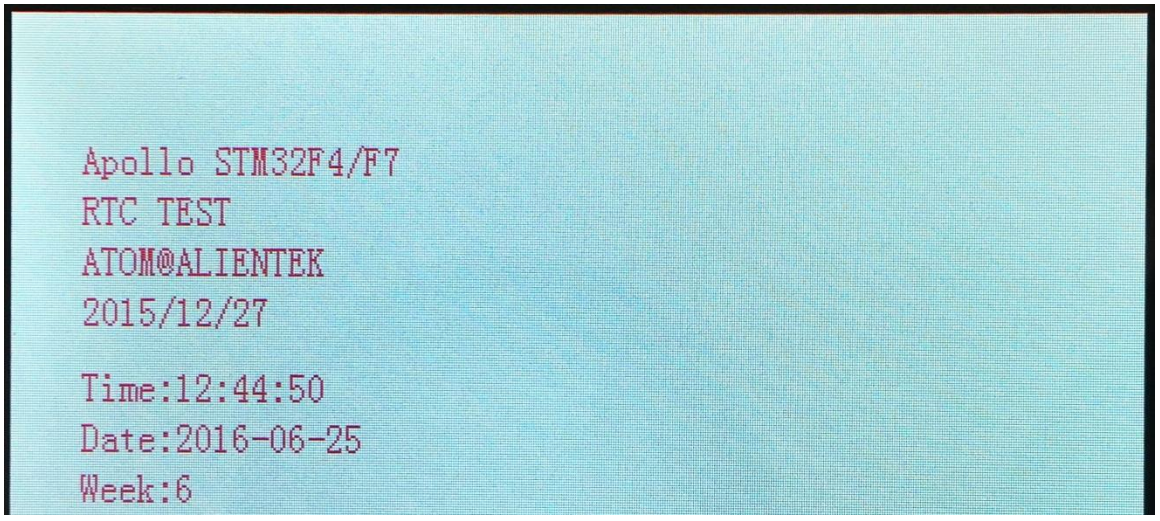


图 22.4.1 RTC 实验测试图

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 22.4.2 所示：

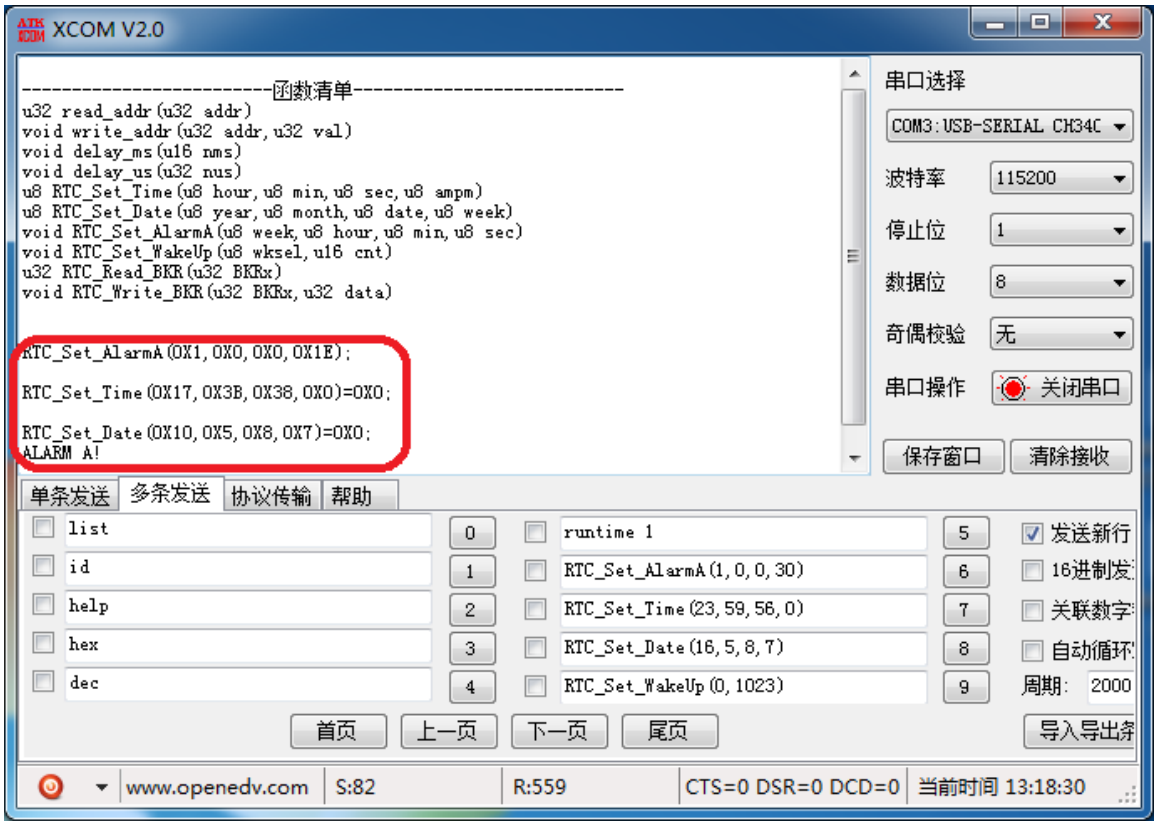


图 22.4.2 通过 USMART 设置时间和日期并测试闹钟 A

可以看到，设置闹钟 A 后，串口返回了 ALARM A!字符串，说明我们的闹钟 A 代码正常运行了！

第二十三章 硬件随机数实验

本章我们将向大家介绍 STM32F767 的硬件随机数发生器。在本章中，我们将使用 KEY0 按键来获取硬件随机数，并且将获取到的随机数值显示在 LCD 上面，同时，使用 DS0 指示程序运行状态。本章将分为如下几个部分：

23.1 STM32F767 随机数发生器简介
23.2 硬件设计
23.3 软件设计
23.4 下载验证

23.1 STM32F767 随机数发生器简介

STM32F767 自带了硬件随机数发生器（RNG），RNG 处理器是一个以连续模拟噪声为基础的随机数发生器，在主机读数时提供一个 32 位的随机数。STM32F767 的随机数发生器框图如图 23.1.1 所示：

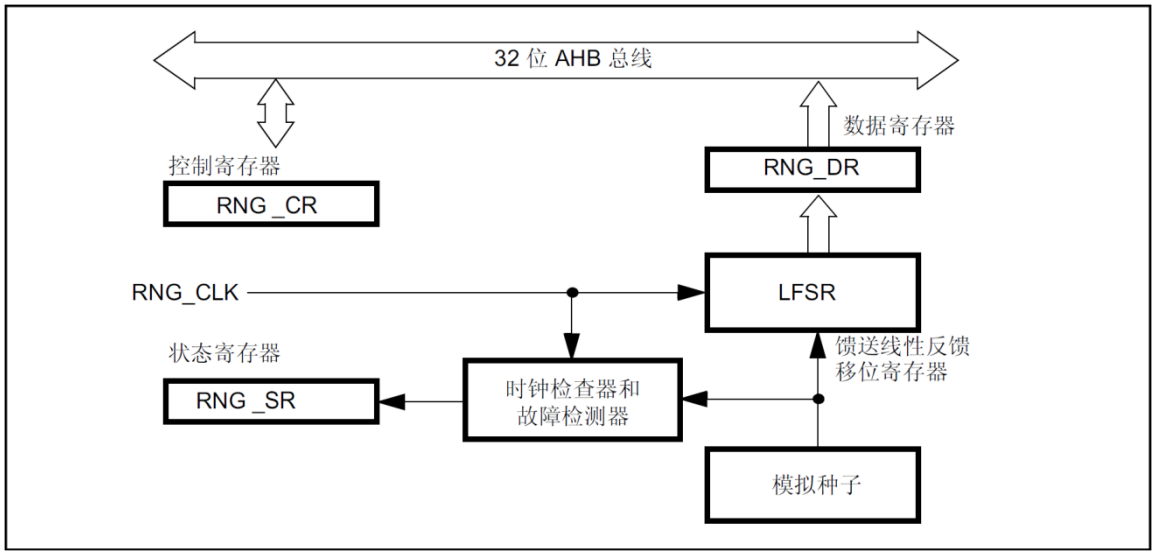


图 23.1.1 随机数发生器(RNG)框图

STM32F767 的随机数发生器（RNG）采用模拟电路实现。此电路产生馈入线性反馈移位寄存器 (RNG_LFSR) 的种子，用于生成 32 位随机数。

该模拟电路由几个环形振荡器组成，振荡器的输出进行异或运算以产生种子。RNG_LFSR 由专用时钟 (PLL48CLK，即 RNG_CLK) 按恒定频率提供时钟信息，因此随机数质量与 HCLK 频率无关。当将大量种子引入 RNG_LFSR 后，RNG_LFSR 的内容会传入数据寄存器 (RNG_DR)。

同时，系统会监视模拟种子和专用时钟 PLL48CLK，当种子上出现异常序列，或 PLL48CLK 时钟频率过低时，可以由 RNG_SR 寄存器的对应位读取到，如果设置了中断，则在检测到错误时，还可以产生中断。

接下来，我们介绍下 STM32F767 随机数发生器（RNG）的几个寄存器。

首先是 RNG 控制寄存器：RNG_CR，该寄存器各位描述如图 23.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												IE	RNGEN	Reserved	
												rw	rw		

位 31:4 保留，必须保持复位值

位 3 **IE**：中断使能 (Interrupt enable)
0：禁止 RNG 中断。
1：使能 RNG 中断。只要 RNG_SR 寄存器中 DRDY=1 或 SEIS=1 或 CEIS=1，就会挂起中断。

位 2 **RNGEN**：随机数发生器使能 (Random number generator enable)
0：禁止随机数发生器。
1：使能随机数发生器。

位 1:0 保留，必须保持复位值

图 23.1.2 RNG_CR 寄存器各位描述

该寄存器只有 bit2 和 bit3 有效，用于使能随机数发生器和中断。我们一般不用中断，所以只需要设置 bit2 为 1，使能随机数发生器即可。

然后，我们看看 RNG 状态寄存器：RNG_SR，该寄存器各位描述如图 23.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									SEIS	CEIS	Reserved		SECS	CECS	DRDY
									rc_w0	rc_w0			r	r	r

图 23.1.3 RNG_SR 寄存器各位描述

该寄存器我们仅关心最低位（DRDY 位），该位用于表示 RNG_DR 寄存器包含的随机数数据是否有效，如果该位为 1，则说明 RNG_DR 的数据是有效的，可以读取出来了。读 RNG_DR 后，该位自动清零。

最后，我们看看 RNG 数据寄存器：RNG_DR，该寄存器各位描述如图 23.4.3 所示：

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

位 31:0 **RNDATA**：随机数据 (Random data)
32 位随机数据。

图 23.1.3 RNG_SR 寄存器各位描述

在 RNG_SR 的 DRDY 位置位后，我们就可以读取该寄存器获得 32 位随机数值。此寄存器在最多 40 个 PLL48CK 时钟周期后，又可以提供新的随机数值。

至此，随机数发生器的寄存器，我们就介绍完了。接下来，我们看看要使用 HAL 库操作随机数发生器，应该如何设置。

首先，我们要说明的是，库函数中随机数发生器相关的操作在文件 stm32f7xx_hal_rng.c 和对应的头文件 stm32f7xx_hal_rng.h 中。所以我们实验工程必须引入这两个文件。

随机数发生器操作步骤如下：

1）使能随机数发生器时钟。

要使用随机数发生器，必须先使能其时钟。随机数发生器时钟来自 PLL48CK，通过 AHB2ENR 寄存器使能。 HAL 库使能随机数发生器时钟方法为：

```
__HAL_RCC_RNG_CLK_ENABLE();//使能 RNG 时钟
```

2）初始化(使能)随机数发生器。

HAL 库提供了 HAL_RNG_Init 函数，该函数非常简单，主要作用是引导调用 RNG 的 MSP 回调函数，然后使能随机数发生器。该函数声明如下：

HAL_StatusTypeDef HAL_RNG_Init(RNG_HandleTypeDef *hrng);

该函数非常简单，这里我们就不做过多讲解。使用方法如下：

```
RNG_HandleTypeDef    RNG_Handler;  //RNG 句柄
RNG_Handler.Instance=RNG;
HAL_RNG_Init(&RNG_Handler);//初始化 RNG
```

当我们使用 HAL_RNG_Init 之后，在该函数内部，会调用 RNG 的 MSP 回调函数，回调函数声明如下：

```
void HAL_RNG_MspInit(RNG_HandleTypeDef *hrng);
```

回调函数中一般编写与 MCU 相关的外设时钟初始化以及 NVIC 配置。

同时，HAL 库也提供了单独使能随机数发生器的方法为：

```
__HAL_RNG_ENABLE(hrng);   //使能 RNG
```

3）判断 DRDY 位，读取随机数值。

经过前面两个步骤，我们就可以读取随机数值了，不过每次读取之前，必须先判断 RNG_SR 寄存器的 DRDY 位，如果该位为 1，则可以读取 RNG_DR 得到随机数值，如果不为 1，则需要等待。

在 HAL 库中，判断 DRDY 位并读取随机数值的函数为：

```
uint32_t HAL_RNG_GetRandomNumber(RNG_HandleTypeDef *hrng);
```

通过以上几个步骤的设置，我们就可以使用 STM32F7 的随机数发生器（RNG）了。本章，我们将实现如下功能：通过 KEY0 获取随机数，并将获取到的随机数显示在 LCD 上面，通过 DS0 指示程序运行状态。

23.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） 串口

3） KEY0 按键

4） 随机数发生器(RNG)

5） LCD 模块

这些资源，我们都已经介绍了，硬件连接上面也不需要任何变动，插上 LCD 模块即可。

23.3 软件设计

打开本章的实验工程可以看到，我们在 HALLIB 下面添加了随机数发生器支持库函数 stm32f7xx_hal_rng.c 和对应的头文件 stm32f7xx_hal_rng.h。同时我们编写的随机数发生器相关的函数在新增的文件 rng.c 中。

接下来我们看看 rng.c 源文件内容：

```
RNG_HandleTypeDef RNG_Handler;  //RNG 句柄


//初始化 RNG
```

```
u8 RNG_Init(void)
{
    u16 retry=0;

    RNG_Handler.Instance=RNG;
    HAL_RNG_Init(&RNG_Handler);//初始化 RNG
    while(__HAL_RNG_GET_FLAG(&RNG_Handler,RNG_FLAG_DRDY)==RESET
                                        &&retry<10000)//等待 RNG 准备就绪
    {
        retry++; delay_us(10);
    }
    if(retry>=10000) return 1;//随机数产生器工作不正常
    return 0;
}
void HAL_RNG_MspInit(RNG_HandleTypeDef *hrng)
{
     __HAL_RCC_RNG_CLK_ENABLE();//使能 RNG 时钟
}
//得到随机数
//返回值:获取到的随机数
u32 RNG_Get_RandomNum(void)
{
    return HAL_RNG_GetRandomNumber(&RNG_Handler);
}

//生成[min,max]范围的随机数
int RNG_Get_RandomRange(int min,int max)
{
   return HAL_RNG_GetRandomNumber(&RNG_Handler)%(max-min+1) +min;
}
```

该部分总共 4 个函数，其中：RNG_Init 用于初始化随机数发生器；HAL_RNG_MspInit 函数是随机数发生器 MSP 回调函数，该函数下面只有一行代码就是使能 RNG 时钟。RNG_Get_RandomNum 用于读取随机数值；RNG_Get_RandomRange 用于读取一个特定范围内的随机数，实际上也是调用的函数 HAL_RNG_GetRandomNumber 来实现的。这些函数的实现方法都比较好理解。

最后我们看看 main.c 文件内容：

```
int main(void)
{
    u32 random;
    u8 t=0,key;
    Cache_Enable();                 //打开 L1-Cache ‘
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
```

```
    delay_init(216);                    //延时初始化
    uart_init(115200);                  //串口初始化
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //LCD 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,70,200,16,16,"RNG TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2016/7/12");
    while(RNG_Init())                   //初始化随机数发生器
    {
        LCD_ShowString(30,130,200,16,16,"    RNG Error! ");
        delay_ms(200);
        LCD_ShowString(30,130,200,16,16,"RNG Trying...");
    }
    ……//此处省略部分液晶显示代码
    while(1)
    {
        delay_ms(10);
        key=KEY_Scan(0);
        if(key==KEY0_PRES)
        {
            random=RNG_Get_RandomNum(); //获得随机数
            LCD_ShowNum(30+8*11,180,random,10,16); //显示随机数

        }
        if((t%20)==0)
        {
            LED0_Toggle;                        //每 200ms,翻转一次 LED0
            random=RNG_Get_RandomRange(0,9);//获取[0,9]区间的随机数
            LCD_ShowNum(30+8*16,210,random,1,16); //显示随机数
         }
        delay_ms(10);
        t++;
    }
}
```

该部分代码也比较简单，在所有外设初始化成功后，进入死循环，等待按键按下，如果 KEY0 按下，则调用 RNG_Get_RandomNum 函数，读取随机数值，并将读到的随机数显示在 LCD 上面。每隔 200ms 获取一次区间[0,9]的随机数，并实时显示在液晶上。同时 DS0，周期性闪烁，400ms 闪烁一次。这就实现了前面我们所说的功能。

至此，本实验的软件设计就完成了，接下来就让我们来检验一下，我们的程序是否正确了。

23.4 下载验证

将程序下载到阿波罗 STM32F767 后，可以看到 DS0 不停的闪烁，提示程序已经在运行了。然后我们按下 KEY0，就可以在屏幕上看到获取到的随机数。同时，就算不按 KEY0，程序也会自动的获取 0~9 区间的随机数显示在 LCD 上面。实验结果如图 23.4.1 所示：

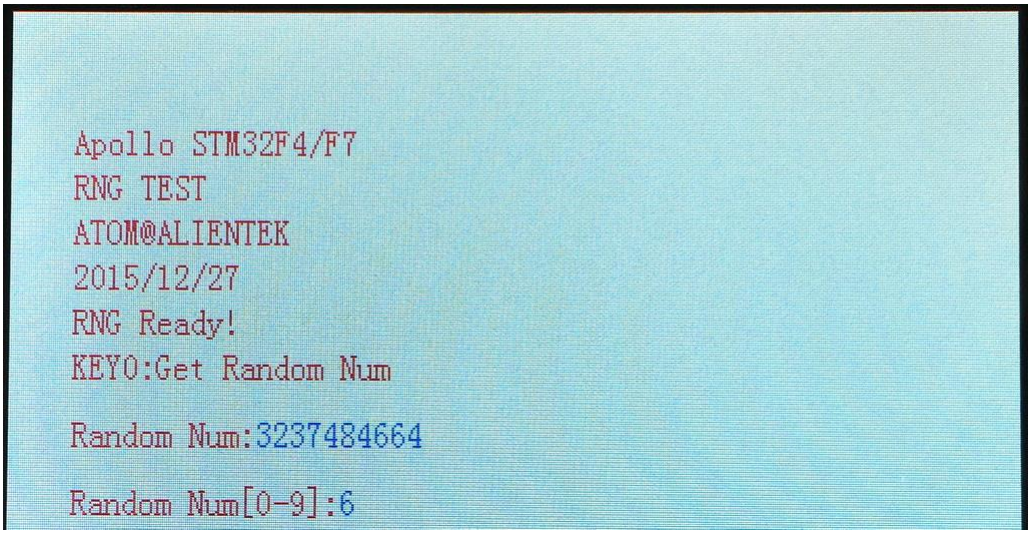


图 23.4.1 获取随机数成功

第二十四章 待机唤醒实验

本章我们将向大家介绍 STM32F767 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：

24.1 STM32F767 待机模式简介
24.2 硬件设计
24.3 软件设计
24.4 下载验证

24.1 STM32F767 待机模式简介

很多单片机都有低功耗模式，STM32F767 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。STM32F767 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。

STM32F767 提供了 3 种低功耗模式，以达到不同层次的降低功耗的目的，这三种模式如下：

1）睡眠模式（CM7 内核停止工作，外设仍在运行）；

2）停止模式（所有的时钟都停止）；

3）待机模式；

在运行模式下，我们也可以通过降低系统时钟关闭 APB 和 AHB 总线上未被使用的外设的时钟来降低功耗。三种低功耗模式一览表见表 24.1.1 所示：

模式名称	进入	唤醒	对 1.2 V 域时钟的影响	对 V_{DD} 域时钟的影响	调压器
睡眠（立即休眠或退出时休眠）	WFI	任意中断	CPU CLK 关闭对其它时钟或模拟时钟源无影响	无	开启
	WFE	唤醒事件			
停止	SLEEPDEEP 位 + WFI 或 WFE	任意 EXTI 线（在 EXTI 寄存器中配置，内部线和外部线）	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	主调压器或低功耗调压器（取决于 *PWR 电源控制寄存器 (PWR_CR1)*）
待机	PDDS 位 + SLEEPDEEP 位 + WFI 或 WFE	WKUP 引脚上升沿或下降沿、RTC 闹钟（闹钟 A 或闹钟 B）、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位、IWDG 复位			关闭

表 24.1.1 STM32F767 低功耗一览表

在这三种低功耗模式中，最低功耗的是待机模式，在此模式下，最低只需要 2.4uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 130uA 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 STM32F767 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32F767 的最低功耗。该模式是在 CM7 深睡眠模式时关闭电压调节器。整个 1.2V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失。除备份域（RTC 寄存

器、RTC 备份寄存器和备份 SRAM）和待机电路中的寄存器外，SRAM 和寄存器内容都将丢失。

那么我们如何进入待机模式呢？其实很简单，只要按图 24.1.1 所示的步骤执行就可以了：

待机模式	说明
进入模式	WFI（等待中断）或 WFE（等待事件），且： – Cortex®-M7 系统控制寄存器中的 SLEEPDEEP 置 1， – 电源控制寄存器 (PWR_CR) 中的 PDDS 位置 1， – 没有中断（针对 WFI）和事件（针对 WFE）挂起， – 电源控制寄存器 (PWR_CR) 中的 WUF 位清零， – 将与所选唤醒源（RTC 闹钟 A、RTC 闹钟 B、RTC 唤醒、RTC 入侵或 RTC 时间戳标志）对应的 RTC 标志清零
	从 ISR 恢复，条件为： – Cortex®-M7 系统控制寄存器中的 SLEEPDEEP 位置 1，及 – SLEEPONEXIT = 1，及 – 电源控制寄存器 (PWR_CR) 中的 PDDS 位置 1，及 – 没有中断挂起， – 电源控制/状态寄存器 (PWR_SR) 中的 WUF 位清零， – 将与所选唤醒源（RTC 闹钟 A、RTC 闹钟 B、RTC 唤醒、RTC 入侵或 RTC 时间戳标志）对应的 RTC 标志清零。
退出模式	WKUP 引脚上升沿或下降沿、RTC 闹钟（闹钟 A 和闹钟 B）、RTC 唤醒事件、入侵事件、时间戳事件、NRST 引脚外部复位 和 IWDG 复位。
唤醒延迟	复位阶段。

图 24.1.1 STM32F767 进入及退出待机模式的条件

图 24.1.1 还列出了退出待机模式的操作，从图 24.1.1 可知，我们有多种方式可以退出待机模式，包括：WKUP 引脚的上升沿/下降沿、RTC 闹钟、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、外部复位(NRST 引脚)、IWDG 复位等，微控制器从待机模式退出。

从待机模式唤醒后的代码执行等同于复位后的执行(采样启动模式引脚，读取复位向量等)。电源控制/状态寄存器(PWR_CSR)将会指示内核由待机状态退出。

在进入待机模式后，除了复位引脚、RTC_AF1 引脚（PC13）（如果针对入侵、时间戳、RTC 闹钟输出或 RTC 时钟校准输出进行了配置）和 WK_UP（PA0/PA2/PC1/PC13/PI8/PI11）（如果使能了）等引脚外，其他所有 IO 引脚都将处于高阻态。

图 24.1.1 已经清楚的说明了进入待机模式的通用步骤，其中涉及到多个寄存器：电源控制寄存器 1/2（PWR_CR1/PWR_CR2）和电源控制/状态寄存器 2（PWR_CSR2）。下面我们分别介绍这几个寄存器：

电源控制寄存器 1（PWR_CR1），该寄存器的各位描述如图 24.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	UDEN[1:0]		ODSWEN	ODEN
												rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VOS[1:0]		ADCDC1	Res.	MRUDS	LPUDS	FPDS	DBP	PLS[2:0]			PVDE	CSBF	Res.	PDDS	LPDS
rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rc_w1		rw	rw

图 24.1.2 PWR_CR1 寄存器各位描述

该寄存器我们只关心 LPDS 和 PDDS 这两个位，通过设置 PWR_CR1 的 PDDS 位，使 CPU

进入深度睡眠时进入待机模式，同时设置 LPDS 位，使调压器进入低功耗模式。

接下来，看电源控制寄存器 2（PWR_CR2），该寄存器的各位描述如图 24.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	WUPP6	WUPP5	WUPP4	WUPP3	WUPP2	WUPP1	Res.	Res.	CWUPF6	CWUPF5	CWUPF4	CWUPF3	CWUPF2	CWUPF1
		rw	rw	rw	rw	rw	rw			r	r	r	r	r	r

图 24.1.3 PWR_CR2 寄存器各位描述

该寄存器我们只关心 CPUPF1 和 WUPP1 这两个位，设置 CWUPF1 为 1，清除 PA0 的唤醒标志位，设置 WUPP1 为 0，设置 PA0 的唤醒极性为上升沿唤醒。

最后，看电源控制/状态寄存器 2（PWR_CSR2），该寄存器的各位描述如图 24.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	EWUP6	EWUP5	EWUP4	EWUP3	EWUP2	EWUP1	Res.	Res.	WUPF6	WUPF5	WUPF4	WUPF3	WUPF2	WUPF1
		rw	rw	rw	rw	rw	rw			r	r	r	r	r	r

图 24.1.4 PWR_CSR2 寄存器各位描述

该寄存器我们只关心 EWUP1 这个位，设置 EWUP1 为 1，选择 PA0（即 WKUP 引脚）作为唤醒引脚。关于这三个寄存器的详细描述，请看《STM32F7 中文参考手册》第 4.4 节。

对于使能了 RTC 闹钟中断或 RTC 周期性唤醒等中断的时候，进入待机模式前，必须按如下操作处理：

1， 禁止 RTC 中断（ALRAIE、ALRBIE、WUTIE、TAMPIE 和 TSIE 等）。

2， 清零对应中断标志位。

3， 清除 PWR 唤醒(WUF)标志（通过设置 PWR_CR 的 CWUF 位实现）。

4， 重新使能 RTC 对应中断。

5， 进入低功耗模式。

在有用到 RTC 相关中断的时候，必须按以上步骤执行之后，才可以进入待机模式，这个大家一定要注意，否则可能无法唤醒。详情请参考《STM32F7 中文参考手册》第 4.3.7 节。

通过以上介绍，我们了解了进入待机模式的方法，以及设置 KEY_UP 引脚用于把 STM32F7 从待机模式唤醒的方法。低功耗相关操作函数和定义在 HAL 库文件 stm32f7xx_hal_pwr.c 和头文件 stm32f7xx_hal_pwr.h 中。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《STM32F3 与 F7 系列 Cortex M4 内核编程手册》，第 214 页 4.4.6 节），接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度

睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。在库函数中，进行上面三个功能进入待机模式是在函数 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 STM32F7 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。

24.2 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） LCD 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 LCD 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。

24.3 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 stm32f7xx_hal_pwr.c 和 stm32f7xx_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_AHB1_FORCE_RESET();       //复位所有 IO 口
    while(WKUP_KD);   //等待 WK_UP 按键松开(在有 RTC 中断时,
                      //必须等 WK_UP 松开再进入待机)
    __HAL_RCC_PWR_CLK_ENABLE();             //使能 PWR 时钟
    __HAL_RCC_BACKUPRESET_FORCE();      //复位备份区域
    HAL_PWR_EnableBkUpAccess();             //后备区域访问使能

    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);
    __HAL_RTC_WRITEPROTECTION_DISABLE(&RTC_Handler);//关闭 RTC 写保护

    //关闭 RTC 相关中断，可能在 RTC 实验打开了
    __HAL_RTC_WAKEUPTIMER_DISABLE_IT(&RTC_Handler,RTC_IT_WUT);
    __HAL_RTC_TIMESTAMP_DISABLE_IT(&RTC_Handler,RTC_IT_TS);
    __HAL_RTC_ALARM_DISABLE_IT(&RTC_Handler,RTC_IT_ALRA|RTC_IT_ALRB);
```

```
    //清除 RTC 相关中断标志位
    __HAL_RTC_ALARM_CLEAR_FLAG(&RTC_Handler,RTC_FLAG_ALRAF|
                                                    RTC_FLAG_ALRBF);
    __HAL_RTC_TIMESTAMP_CLEAR_FLAG(&RTC_Handler,RTC_FLAG_TSF);
    __HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&RTC_Handler,RTC_FLAG_WUTF);

    __HAL_RCC_BACKUPRESET_RELEASE();                //备份区域复位结束
    __HAL_RTC_WRITEPROTECTION_ENABLE(&RTC_Handler); //使能 RTC 写保护
    __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;
    u8 tx=0;//记录松开的次数
    LED0(0); //亮灯 DS0
    while(1)
    {
        if(WKUP_KD)//已经按下了
        {
            t++; tx=0;
        }else
        {
            tx++;
            if(tx>3)//超过 90ms 内没有 WKUP 信号
            {
                LED0(1);    return 0;//错误的按键,按下次数不够
            }
        }
        delay_ms(30);
        if(t>=100)//按下超过 3 秒钟
        {
            LED0(0);      //点亮 DS0
            return 1;     //按下 3s 以上了
        }
    }
}
```

```
//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_FAST;     //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，我们在这里说明三点：

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)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    WKUP_Init();                    //待机唤醒初始化
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    ……//此处省略部分液晶显示代码
    LCD_ShowString(30,130,200,16,16,"WK_UP:Stanby/WK_UP");
    while(1)
    {
        LED0_Toggle;
        delay_ms(250);//延时 250ms
    }
}
```

这里我们先初始化 LED 和 WK_UP 按键（通过 WKUP_Init（）函数初始化），如果检测到有长按 WK_UP 按键 3 秒以上，则开机，并执行 LCD 初始化，在 LCD 上面显示一些内容，如果没有长按，则在 WKUP_Init 里面，调用 Sys_Enter_Standby 函数，直接进入待机模式了。

开机后，在死循环里面等待 WK_UP 中断的到来，在得到中断后，在中断函数里面判断 WK_UP 按下的时间长短，来决定是否进入待机模式，如果按下时间超过 3 秒，则进入待机，否则退出中断，继续执行 main 函数的死循环等待，同时不停的取反 LED0，让红灯闪烁。

代码部分就介绍到这里，大家记住**下载代码后，一定要长按 WK_UP 按键，来开机，否则**

将直接进入待机模式，无任何现象。

24.4 下载与测试

在代码编译成功之后，下载代码到阿波罗 STM32 开发板上，此时，看到开发板 DS0 亮了一下（Check_WKUP 函数执行了 LED0(0)的操作），就没有反应了。其实这是正常的，在程序下载完之后，开发板检测不到 WK_UP 的持续按下（3 秒以上），所以直接进入待机模式，看起来和没有下载代码一样。此时，我们长按 WK_UP 按键 3 秒钟左右，可以看到 DS0 开始闪烁，液晶也会显示一些内容。然后再长按 WK_UP，DS0 会灭掉，液晶灭掉，程序再次进入待机模式。

特别注意：如果你之前开启了 RTC 周期性唤醒中断（比如下载了 RTC 实验），那么会看到 DS0 周期性的闪烁（周期性唤醒 MCU 了）。如果想去掉这种情况，请关闭 RTC 的周期性唤醒中断。简单的办法：将 CR1220 电池去掉，然后给板子断电，等待 10 秒钟左右，让 RTC 配置全部丢失，然后再装上 CR1220 电池，之后再给开发板供电，就不会看到 DS0 周期性闪烁了。

第二十五章 ADC 实验

本章我们将向大家介绍 STM32F767 的 ADC 功能。在本章中，我们将使用 STM32F767 的 ADC1 通道 5 来采样外部电压值，并在 LCD 模块上显示出来。本章将分为如下几个部分：

25.1 STM32F767 ADC 简介
25.2 硬件设计
25.3 软件设计
25.4 下载验证

25.1 STM32F767 ADC 简介

STM32F767xx 系列有 3 个 ADC，这些 ADC 可以独立使用，也可以使用双重/三重模式（提高采样率）。STM32F767 的 ADC 是 12 位逐次逼近型的模拟数字转换器。它有 19 个通道，可测量 16 个外部源、2 个内部源和 Vbat 通道的信号。这些通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。 模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阀值。

STM32F767IGT6 包含有 3 个 ADC。STM32F767 的 ADC 最大的转换速率为 2.4Mhz，也就是转换时间为 0.41us（在 ADCCLK=36M,采样周期为 3 个 ADC 时钟下得到），不要让 ADC 的时钟超过 36M，否则将导致结果准确度下降。

STM32F767 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。

STM32F767 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《STM32F7 中文参考手册》第 394 页，第 15.3.4 节。

STM32F767 的 ADC 可以进行很多种不同的转换模式，这些模式在《STM32F7 中文参考手册》的第 15 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。

STM32F767 的 ADC 在单次转换模式下，只执行一次转换，该模式可以通过 ADC_CR2 寄存器的 ADON 位（只适用于规则通道）启动，也可以通过外部触发启动（适用于规则通道和注入通道），这时 CONT 位为 0。

以规则通道为例，一旦所选择的通道转换完成，转换结果将被存在 ADC_DR 寄存器中，EOC（转换结束）标志将被置位，如果设置了 EOCIE，则会产生中断。然后 ADC 将停止，直到下次启动。

接下来，我们介绍一下我们执行规则通道的单次转换，需要用到的 ADC 寄存器。第一个要介绍的是 ADC 控制寄存器（ADC_CR1 和 ADC_CR2）。ADC_CR1 的各位描述如图 25.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES		AWDEN	JAWDEN	Reserved					
					rw	rw	rw	rw	rw						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISC EN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 25.1.1 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《STM32F7 中文参考手册》第 15.13.2 节。

ADC_CR1 的 SCAN 位，该位用于设置扫描模式，由软件设置和清除，如果设置为 1，则使用扫描模式，如果为 0，则关闭扫描模式。在扫描模式下，由 ADC_SQRx 或 ADC_JSQRx 寄存器选中的通道被转换。如果设置了 EOCIE 或 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[25:24]用于设置 ADC 的分辨率，详细的对应关系如图 25.1.2 所示：

位 25:24 **RES[1:0]**：分辨率 (Resolution)
通过软件写入这些位可选择转换的分辨率。
00：12 位（15 ADCCLK 周期）
01：10 位（13 ADCCLK 周期）
10：8 位（11 ADCCLK 周期）
11：6 位（9 ADCCLK 周期）

图 25.1.2 ADC 分辨率选择

本章我们使用 12 位分辨率，所以设置这两个位为 0 就可以了。接着我们介绍 ADC_CR2，该寄存器的各位描述如图 25.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

图 25.1.3 ADC_CR2 寄存器各位描述

该寄存器我们也只针对性的介绍一些位：ADON 位用于开关 AD 转换器。而 CONT 位用于设置是否进行连续转换，我们使用单次转换，所以 CONT 位必须为 0。ALIGN 用于设置数据对齐，我们使用右对齐，该位设置为 0。

EXTEN[1:0]用于规则通道的外部触发使能设置，详细的设置关系如图 25.1.4 所示：

位 29:28 **EXTEN**：规则通道的外部触发使能 (External trigger enable for regular channels)
通过软件将这些位置 1 和清零可选择外部触发极性和使能规则组的触发。
00：禁止触发检测
01：上升沿上的触发检测
10：下降沿上的触发检测
11：上升沿和下降沿上的触发检测

图 25.1.4 ADC 规则通道外部触发使能设置

我们这里使用的是软件触发，即不使用外部触发，所以设置这 2 个位为 0 即可。ADC_CR2 的 SWSTART 位用于开始规则通道的转换，我们每次转换（单次转换模式下）都需要向该位写 1。

第二个要介绍的是 ADC 通用控制寄存器（ADC_CCR），该寄存器各位描述如图 25.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				ADCPRE	
								rw	rw					rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA[1:0]		DDS	Res.	DELAY[3:0]				Reserved			MULTI[4:0]				
rw	rw	rw		rw	rw	rw	rw				rw	rw	rw	rw	rw

图 25.1.5 ADC_CCR 寄存器各位描述

该寄存器我们也只针对性的介绍一些位：TSVREFE 位是内部温度传感器和 Vrefint 通道使能位，内部温度传感器我们将在下一章介绍，这里我们直接设置为 0。ADCPRE[1:0]用于设置 ADC 输入时钟分频，00~11 分别对应 2/4/6/8 分频，STM32F767 的 ADC 最大工作频率是 36Mhz，而 ADC 时钟（ADCCLK）来自 APB2，APB2 频率一般是 108Mhz，我们设置 ADCPRE=01，即 4 分频，这样得到 ADCCLK 频率为 27Mhz。MULTI[4:0]用于多重 ADC 模式选择，详细的设置关系如图 25.1.6 所示：

位 4:0 **MULTI[4:0]**：多重 ADC 模式选择 (Multi ADC mode selection)
通过软件写入这些位可选择操作模式。
所有 ADC 均独立：
00000：独立模式
00001 到 01001：双重模式，ADC1 和 ADC2 一起工作，ADC3 独立
00001：规则同时 + 注入同时组合模式
00010：规则同时 + 交替触发组合模式
00011：Reserved
00101：仅注入同时模式
00110：仅规则同时模式
仅交错模式
01001：仅交替触发模式
10001 到 11001：三重模式：ADC1、ADC2 和 ADC3 一起工作
10001：规则同时 + 注入同时组合模式
10010：规则同时 + 交替触发组合模式
10011：Reserved
10101：仅注入同时模式
10110：仅规则同时模式
仅交错模式
11001：仅交替触发模式
其它所有组合均需保留且不允许编程

图 25.1.6 多重 ADC 模式选择设置

本章我们仅用了 ADC1（独立模式），并没用到多重 ADC 模式，所以设置这 5 个位为 0 即

可。

第三个要介绍的是 ADC 采样时间寄存器（ADC_SMPR1 和 ADC_SMPR2），这两个寄存器用于设置通道 0~18 的采样时间，每个通道占用 3 个位。ADC_SMPR1 的各位描述如图 25.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					rw	rw	rw	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 31:27 保留，必须保持复位值。

位 26:0 **SMPx[2:0]**：通道 X 采样时间选择 (Channel x sampling time selection)

通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间，通道选择位必须保持不变。

注意：
000：3 个周期
001：15 个周期
010：28 个周期
011：56 个周期
100：84 个周期
101：112 个周期
110：144 个周期
111：480 个周期

图 25.1.7 ADC_SMPR1 寄存器各位描述

ADC_SMPR2 的各位描述如下图 25.1.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			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
SMP 5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:30 保留，必须保持复位值。

位 29:0 **SMPx[2:0]**：通道 X 采样时间选择 (Channel x sampling time selection)

通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间，通道选择位必须保持不变。

注意：
000：3 个周期
001：15 个周期
010：28 个周期
011：56 个周期
100：84 个周期
101：112 个周期
110：144 个周期
111：480 个周期

图 25.1.8 ADC_SMPR2 寄存器各位描述

对于每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率。ADC 的转换时间可以由以下公式计算：

Tcovn=采样时间+12 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=27Mhz 的时候，并设置 3 个周期的采样时间，则得到：Tcovn=3+12=15 个周期=0.55us。

第四个要介绍的是 ADC 规则序列寄存器（ADC_SQR1~3）,该寄存器总共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC_SQR1，该寄存器的各位描述如图 25.1.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

位 31:24 保留，必须保持复位值。

位 23:20 **L[3:0]**：规则通道序列长度 (Regular channel sequence length)
通过软件写入这些位可定义规则通道转换序列中的转换总数。
0000：1 次转换
0001：2 次转换
...
1111：16 次转换

位 19:15 **SQ16[4:0]**：规则序列中的第十六次转换 (16th conversion in regular sequence)
通过软件写入这些位，并将通道编号 (0..18) 分配为转换序列中的第十六次转换。

位 14:10 **SQ15[4:0]**：规则序列中的第十五次转换 (15th conversion in regular sequence)

位 9:5 **SQ14[4:0]**：规则序列中的第十四次转换 (14th conversion in regular sequence)

位 4:0 **SQ13[4:0]**：规则序列中的第十三次转换 (13th conversion in regular sequence)

图 25.1.9 ADC_ SQR1 寄存器各位描述

L[3：0]用于存储规则序列的长度，我们这里只用了 1 个，所以设置这几个位的值为 0。其他的 SQ13~16 则存储了规则序列中第 13~16 个通道的编号（0~18）。另外两个规则序列寄存器同 ADC_SQR1 大同小异，我们这里就不再介绍了，要说明一点的是：我们选择的是单次转换，所以只有一个通道在规则序列里面，这个序列就是 SQ1，至于 SQ1 里面哪个通道，完全由用户自己设置，通过 ADC_SQR3 的最低 5 位（也就是 SQ1）设置。

第五个要介绍的是 ADC 规则数据寄存器(ADC_DR)。规则序列中的 AD 转化结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。ADC_DR 的各位描述如图 25.1.10：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留，必须保持复位值。

位 15:0 **DATA[15:0]**：规则数据 (Regular data)
这些位为只读。它们包括来自规则通道的转换结果。数据有左对齐和右对齐两种方式。

图 25.1.10 ADC_ JDRx 寄存器各位描述

这里要提醒一点的是，该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

最后一个要介绍的 ADC 寄存器为 ADC 状态寄存器（ADC_SR），该寄存器保存了 ADC 转换时的各种状态。该寄存器的各位描述如图 25.1.11 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

图 25.1.11 ADC_ SR 寄存器各位描述

这里我们仅介绍将要用到的是 EOC 位，我们通过判断该位来决定是否此次规则通道的 AD 转换已经完成，如果该位位 1，则表示转换完成了，就可以从 ADC_DR 中读取转换结果，否则等待转换完成。

至此，本章要用到的 ADC 相关寄存器全部介绍完毕了，对于未介绍的部分，请大家参考《STM32F7xx 中文参考手册》第 15 章相关章节。通过以上介绍，我们了解了 STM32F7 的单次转换模式下的相关设置，接下来我们介绍使用库函数来设置 ADC1 的通道 5 来进行 AD 转换的步骤，这里需要说明一下，使用到的库函数分布在 stm32f7xx_hal_adc.c/stm32f7xx_hal_adc_ex.c 文件和 stm32f7xx_hal_adc.h/stm32f7xx_hal_adc_ex.h 文件中。下面讲解其详细设置步骤：

1）开启 PA 口时钟和 ADC1 时钟，设置 PA5 为模拟输入。

STM32F7IGT6 的 ADC1 通道 5 在 PA5 上，所以，我们先要使能 PORTA 的时钟，然后设置 PA5 为模拟输入。同时我们要把 PA5 复用为 ADC，所以我们要使能 ADC1 时钟。

这里特别要提醒，对于 IO 口复用为 ADC 我们要设置模式为模拟输入，而不是复用功能。

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 GPIOA5 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_5;                //PA5
GPIO_Initure.Mode=GPIO_MODE_ANALOG;         //模拟输入
GPIO_Initure.Pull=GPIO_NOPULL;              //不带上下拉
HAL_GPIO_Init(GPIOA,&GPIO_Initure);
```

这里需要说明一下，ADC 的通道与引脚的对应关系在 STM32F7 的数据手册可以查到，我们这里使用 ADC1 的通道 5，在数据手册中的表格为：

PA5	I/O	TT a	(4)	TIM2_CH1/TIM2_ETR, TIM8_CH1N, SPI1_SCK/I2S1_CK, OTG_HS_ULPI_CK, LCD_R4, EVENTOUT	ADC12_IN5, DAC_OUT2

表 23.1.12 ADC1 通道 5 对应引脚查看表

这里我们把 ADC1~ADC3 的引脚与通道对应关系列出来， 16 个外部源的对应关系如下表：

通道号	ADC1	ADC2	ADC3
通道 0	PA0	PA0	PA0
通道 1	PA1	PA1	PA1
通道 2	PA2	PA2	PA2
通道 3	PA3	PA3	PA3
通道 4	PA4	PA4	PF6
通道 5	PA5	PA5	PF7
通道 6	PA6	PA6	PF8
通道 7	PA7	PA7	PF9
通道 8	PB0	PB0	PF10
通道 9	PB1	PB1	PF3
通道 10	PC0	PC0	PC0

通道 11	PC1	PC1	PC1
通道 12	PC2	PC2	PC2
通道 13	PC3	PC3	PC3
通道 14	PC4	PC4	PF4
通道 15	PC5	PC5	PF5

表 23.1.13 ADC1~ADC3 引脚对应关系表

2）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
  __IO uint32_t               NbrOfCurrentConversionRank; //当前转换序列
  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 ClockPrescaler;//分频系数 2/4/6/8 分频 ADC_CLOCK_SYNC_PCLK_DIV4
  uint32_t Resolution;     //分辨率 12/10/8/6 位：ADC_RESOLUTION_12B
  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 EOCSelection; //EOC 标志是否设置  DISABLE
  uint32_t ContinuousConvMode;//开启连续转换模式或者单次转换模式 DISABLE
  uint32_t DMAContinuousRequests;//开启 DMA 请求连续模式或者单独模式 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
  uint32_t ExternalTrigConvEdge;//外部触发边沿
}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
  uint32_t Offset;               //备用，暂未用到
}ADC_ChannelConfTypeDef;
```

该结构体有四个成员变量，对于 STM32F7 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= ADC_CHANNEL_5;  //通道 5
ADC1_ChanConf.Rank=1;    //第 1 个序列，序列 1
ADC1_ChanConf.SamplingTime=ADC_SAMPLETIME_480CYCLES; //采样时间
ADC1_ChanConf.Offset=0;
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，阿波罗 STM32F7 开发板使用的是 STM32F7IGT6，该芯片只有 Vref+参考电压引脚，Vref+的输入范围为：1.8~VDDA。阿波罗 STM32F7 开发板通过 P5 端口，来设置 Vref+的参考电压，默认的我们是通过跳线帽将 ref+接到 3.3V，参考电压就是 3.3V。如果大家想自己设置其他参考电压，将你的参考电压接在 Vref+上就 OK 了（注意要共地）。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 STM32F7 的 ADC1 来执行 AD 转换操作了。

25.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） LCD 模块

3） ADC

4） 杜邦线

前面 2 个均已介绍过，而 ADC 属于 STM32F767 内部资源，实际上我们只需要软件设置就可以正常工作，不过我们需要在外部连接其端口到被测电压上面。本章，我们通过 ADC1 的通道 5（PA5）来读取外部电压值，阿波罗 STM32F767 开发板没有设计参考电压源在上面，但是板上有几个可以提供测试的地方：1，3.3V 电源。2，GND。3，后备电池。注意：这里不能接到板上 5V 电源上去测试，这可能会烧坏 ADC!。

因为要连接到其他地方测试电压，所以我们需要 1 跟杜邦线，或者自备的连接线也可以，一头插在多功能端口 P11 的 ADC 插针上（与 PA5 连接），另外一头就接你要测试的电压点（确保该电压不大于 3.3V 即可）。

25.3 软件设计

打开实验工程可以发现，我们在 HALLIB 分组下面新增了 stm32f7xx_hal_adc.c 和 stm32f7xx_hal_adc_ex.c 源文件，同时会引入对应的头文件。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)
{
    ADC1_Handler.Instance=ADC1;
    ADC1_Handler.Init.ClockPrescaler=ADC_CLOCK_SYNC_PCLK_DIV4;
                                        //4 分频，ADCCLK=PCLK2/4=90/4=22.5MHZ
    ADC1_Handler.Init.Resolution=ADC_RESOLUTION_12B;          //12 位模式
    ADC1_Handler.Init.DataAlign=ADC_DATAALIGN_RIGHT;          //右对齐
    ADC1_Handler.Init.ScanConvMode=DISABLE;                   //非扫描模式
    ADC1_Handler.Init.EOCSelection=DISABLE;                   //关闭 EOC 中断
    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
    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
    ADC1_Handler.Init.ExternalTrigConvEdge=
                        ADC_EXTERNALTRIGCONVEDGE_NONE;//使用软件触发
    ADC1_Handler.Init.DMAContinuousRequests=DISABLE;          //关闭 DMA 请求
    HAL_ADC_Init(&ADC1_Handler);                              //初始化
```

```
}

//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_5;            //PA5
    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_480CYCLES; //采样时间
    ADC1_ChanConf.Offset=0;
    HAL_ADC_ConfigChannel(&ADC1_Handler,&ADC1_ChanConf);    //通道配置

    HAL_ADC_Start(&ADC1_Handler);                       //开启 AD
    HAL_ADC_PollForConversion(&ADC1_Handler,10);        //轮询转换
    return (u16)HAL_ADC_GetValue(&ADC1_Handler);        //返回最近转换结果
}
//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 4 个函数，MY_Adc_Init 函数调用函数 HAL_ADC_Init 初始化 ADC1 相关参数。第二个函数 HAL_ADC_MspInit 是 MSP 初始化回调函数，用来使能时钟和初始化 IO 口。第三个函数 Get_Adc，用于读取某个通道的 ADC 值，例如我们读取通道 5 上的 ADC 值，就可以通过 Get_Adc（ADC_CHANNEL_5）得到。最后一个函数 Get_Adc_Average，用于多次获取 ADC 值，取平均，用来提高准确度。

头文件 adc.h 代码比较简单，主要是函数申明。接下来我们看看 main 函数内容：

```
int main(void)
{
    u16 adcx;
    float temp;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    MY_ADC_Init();                  //初始化 ADC1 通道 5
    ……//此处省略部分液晶显示代码
    LCD_ShowString(30,130,200,16,16,"ADC1_CH5_VAL:");
    LCD_ShowString(30,150,200,16,16,"ADC1_CH5_VOL:0.000V");  //先显示小数点

    while(1)
    {
        adcx=Get_Adc_Average(ADC_CHANNEL_5,20);//获取通道 5 的 20 次取平均值
        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); //显示整数部分，3.1111 的话显示 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*1000，例如 0.1111 就转换为 111.1，相当保留三位小数。
        LCD_ShowxNum(150,150,temp,3,16,0X80); //显示小数部分
        LED0_Toggle;
        delay_ms(250);
    }
}
```

此部分代码，我们在 TFTLCD 模块上显示一些提示信息后，将每隔 250ms 读取一次 ADC 通道 5 的值，并显示读到的 ADC 值（数字量），以及其转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。这里关于最后的 ADC 值的显示我们说明一下，首先我们在液晶固定位置显示了小数点，然后后面计算步骤中，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就在液晶上面显示转换结果的整数和小数部分。

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 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 25.4.1 所示：

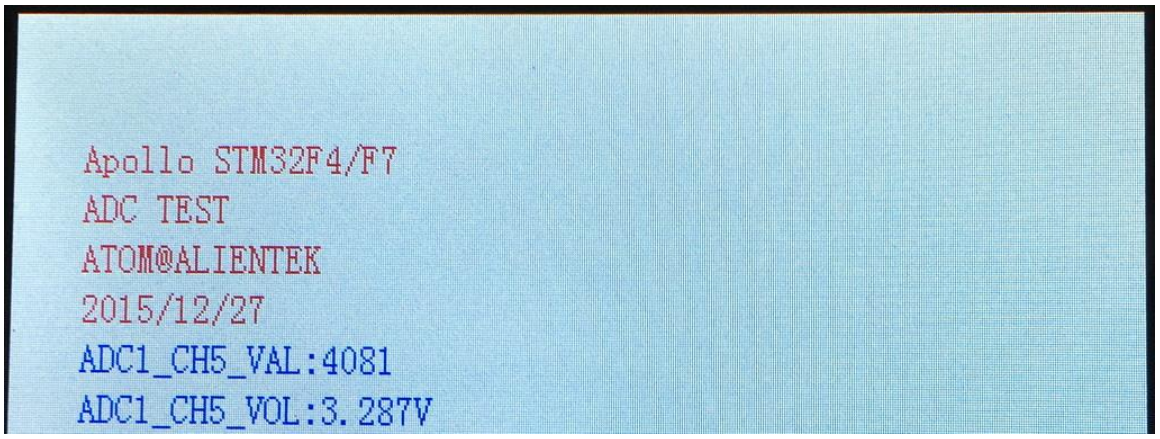


图 25.4.1 ADC 实验测试图

上图中，我们是将 ADC 和 TPAD 连接在一起（通过 P11 排针），可以看到 TPAD 信号电平为 3.3V 左右，这是因为存在上拉电阻 R60 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 STM32F767 ADC 的使用，但这仅仅是 STM32F767 强大的 ADC 功能的一小点应用。STM32F767 的 ADC 在很多地方都可以用到，其 ADC 的 DMA 功能是很不错的，建议有兴趣的大家深入研究下 STM32F767 的 ADC，相信会给你以后的开发带来方便。

第二十六章 内部温度传感器实验

本章我们将向大家介绍 STM32F767 的内部温度传感器。在本章中，我们将使用 STM32F767 的内部温度传感器来读取温度值，并在 LCD 模块上显示出来。本章分为如下几个部分：

26.1 STM32F767 内部温度传感器简介
26.2 硬件设计
26.3 软件设计
26.4 下载验证

26.1 STM32F767 内部温度传感器简介

STM32F7 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)。对于 STM32F7/439 系列来说，该温度传感器在内部和 ADC1_IN18 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。 STM32F7 的内部温度传感器支持的温度范围为：-40~125 度。精度为±1.5℃左右。

STM32F7 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部温度传感器通道就差不多了。关于 ADC 的设置，我们在上一章已经进行了详细的介绍，这里就不再多说。接下来我们介绍一下和温度传感器设置相关的 2 个地方。

第一个地方，我们要使用 STM32F7 的内部温度传感器，必须先激活 ADC 的内部通道，这里通过 ADC_CCR 的 TSVREFE 位（bit23）设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32F7IGT6 的内部温度传感器固定的连接在 ADC1 的通道 18 上，所以，我们在设置好 ADC1 之后只要读取通道 18 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：

$$T（℃）=\{（Vsense - V25）/Avg\_Slope\}+25$$

上式中：

V25=Vsense 在 25 度时的数值（典型值为：0.76）。

Avg_Slope=温度与 Vsense 曲线的平均斜率（单位为 mv/℃或 uv/℃）（典型值为 2.5mV/℃）。

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 STM32F7 内部温度传感器使用的步骤了，如下：

1）设置 ADC1，开启内部温度传感器。

关于如何设置 ADC1，上一章已经介绍了，我们采用与上一章一样的设置。在 HAL 库中开启内部温度传感器，只需要将 ADC 通道改为 ADC_CHANNEL_TEMPSENSOR 即可，调用 HAL_ADC_ConfigChannel()函数配置通道的时候，会自动检测如果是温度传感器通道会在函数中设置 TSVREFE 位。

2）读取通道 16 的 AD 值，计算结果。

在设置完之后，我们就可以读取温度传感器的电压值了，得到该值就可以用上面的公式计算温度值了。具体方法跟上一讲是一样的。

26.2 硬件设计

本实验用到的硬件资源有：

1） 指示灯 DS0

2） LCD 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 STM32F767 内部，不需要外部设置，我们只需要软件设置就 OK 了。

26.3 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,10);
                        //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=(temperate-0.76)/0.0025 + 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;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    MY_ADC_Init();                  //初始化 ADC1 通道 5
    ……//此处省略部分液晶显示代码
    LCD_ShowString(30,140,200,16,16,"TEMPERATE: 00.00C");
                                    //先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_Toggle;
            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。

26.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 26.4.1 所示：

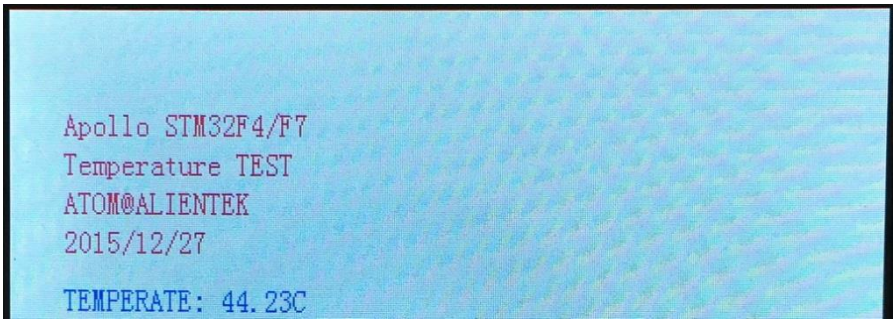


图 26.4.1 内部温度传感器实验测试图

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，所以一般会比实际温度偏高）。

第二十七章 DAC 实验

上几章，我们介绍了 STM32F767 的 ADC 使用，本章我们将向大家介绍 STM32F767 的 DAC 功能。在本章中，我们将利用按键（或 USMART）控制 STM32F767 内部 DAC1 来输出电压，通过 ADC1 的通道 5 采集 DAC 的输出电压，在 LCD 模块上面显示 ADC 获取到的电压值以及 DAC 的设定输出电压值等信息。本章将分为如下几个部分：

27.1 STM32F767 DAC 简介
27.2 硬件设计
27.3 软件设计
27.4 下载验证

27.1 STM32F767 DAC 简介

STM32F767 的 DAC 模块(数字/模拟转换模块)是 12 位数字输入，电压输出型的 DAC。DAC 可以配置为 8 位或 12 位模式，也可以与 DMA 控制器配合使用。DAC 工作在 12 位模式时，数据可以设置成左对齐或右对齐。DAC 模块有 2 个输出通道，每个通道都有单独的转换器。在双 DAC 模式下，2 个通道可以独立地进行转换，也可以同时进行转换并同步地更新 2 个通道的输出。DAC 可以通过引脚输入参考电压 Vref+（通 ADC 共用）以获得更精确的转换结果。

STM32F767 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 27.1.1 所示：

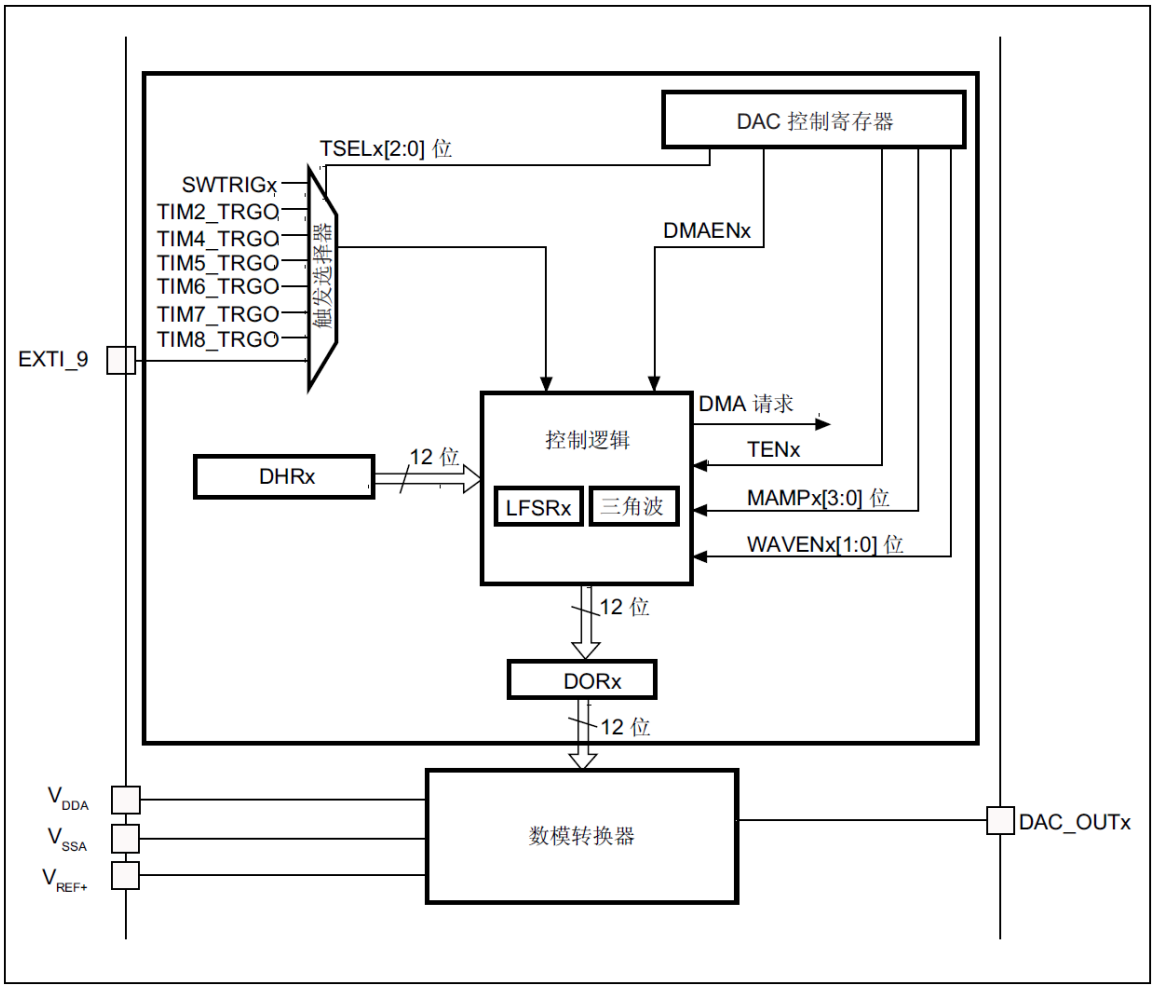


图 27.1.1 DAC 通道模块框图

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 27.1.1 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，STM32F767 的 DAC 支持 8/12 位模式，8 位模式的时候是固定的右对齐的，而 12 位模式又可以设置左对齐/右对齐。单 DAC 通道 x，总共有 3 种情况：

① 8 位数据右对齐：用户将数据写入 DAC_DHR8Rx[7:0]位（实际存入 DHRx[11:4]位）。

② 12 位数据左对齐：用户将数据写入 DAC_DHR12Lx[15:4]位（实际存入 DHRx[11:0]位）。

③ 12 位数据右对齐：用户将数据写入 DAC_DHR12Rx[11:0]位（实际存入 DHRx[11:0]位）。

我们本章使用的就是单 DAC 通道 1，采用 12 位右对齐格式，所以采用第③种情况。

如果没有选中硬件触发(寄存器 DAC_CR1 的 TENx 位置’0’)，存入寄存器 DAC_DHRx 的数据会在一个 APB1 时钟周期后自动传至寄存器 DAC_DORx。如果选中硬件触发(寄存器 DAC_CR1 的 TENx 位置’1’)，数据传输在触发发生以后 3 个 APB1 时钟周期后完成。 一旦数据从 DAC_DHRx 寄存器装入 DAC_DORx 寄存器，在经过时间 $t_{SETTLING}$ 之后，输出即有效，这段时间的长短依电源电压和模拟输出负载的不同会有所变化。我们可以从

STM32F767IGT6 的数据手册查到$t_{SETTLING}$的典型值为 3us，最大是 6us。所以 DAC 的转换速度最快是 333K 左右。

本章我们将不使用硬件触发（TEN=0），其转换的时间框图如图 27.1.2 所示：

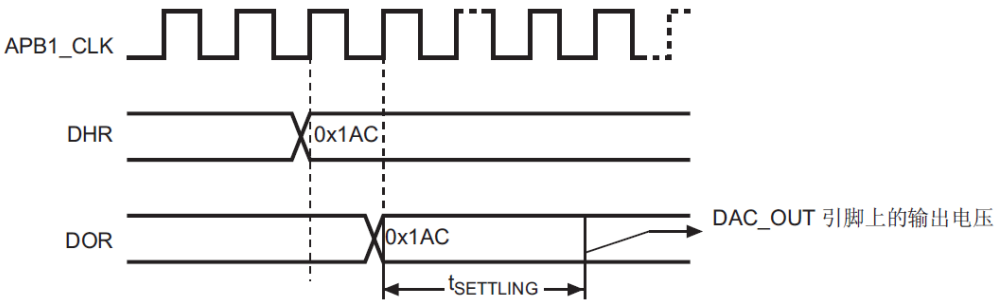


图 27.1.2 TEN=0 时 DAC 模块转换时间框图

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/4096）

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 27.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DMAUDRIE2	DMAEN2	MAMP2[3:0]				WAVE2[1:0]		TSEL2[2:0]			TEN2	BOFF2	EN2
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		DMAUDRIE1	DMAEN1	MAMP1[3:0]				WAVE1[1:0]		TSEL1[2:0]			TEN1	BOFF1	EN1
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 27.1.3 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 27.1.4 所示：

位 7:6 **WAVE1[1:0]**：DAC 1 通道噪声/三角波生成使能 (DAC channel1 noise/triangle wave generation enable)

这些位将由软件置 1 和清零。

00：禁止生成波

01：使能生成噪声波

1x：使能生成三角波

注意：只在位 TEN1 = 1（使能 DAC 1 通道触发）时使用。

位 5:3 **TSEL1[2:0]**：DAC 1 通道触发器选择 (DAC channel1 trigger selection)

这些位用于选择 DAC 1 通道的外部触发事件。

000：定时器 6 TRGO 事件　　100：定时器 2 TRGO 事件

001：定时器 8 TRGO 事件　　101：定时器 4 TRGO 事件

010：定时器 7 TRGO 事件　　110：外部中断线 9

011：定时器 5 TRGO 事件　　111：软件触发

注意：只在位 TEN1 = 1（使能 DAC 1 通道触发）时使用。

位 2 **TEN1**：DAC 1 通道触发使能 (DAC channel1 trigger enable)

此位由软件置 1 和清零，以使能/禁止 DAC 1 通道触发。

0：禁止 DAC 1 通道触发，写入 DAC_DHRx 寄存器的数据在一个 APB1 时钟周期之后转移到 DAC_DOR1 寄存器

1：使能 DAC 1 通道触发，DAC_DHRx 寄存器的数据在三个 APB1 时钟周期之后转移到 DAC_DOR1 寄存器

注意：如果选择软件触发，DAC_DHRx 寄存器的内容只需一个 APB1 时钟周期即可转移到 DAC_DOR1 寄存器。

位 1 **BOFF1**：DAC 1 通道输出缓冲器禁止 (DAC channel1 output buffer disable)

此位由软件置 1 和清零，以使能/禁止 DAC 1 通道输出缓冲器。

0：使能 DAC 1 通道输出缓冲器

1：禁止 DAC 1 通道输出缓冲器

位 0 **EN1**：DAC 1 通道使能 (DAC channel1 enable)

此位由软件置 1 和清零，以使能/禁止 DAC 1 通道。

0：禁止 DAC 1 通道

1：使能 DAC 1 通道

图 27.1.4 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 STM32F767 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 27.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				DACC1DHR[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:12 保留，必须保持复位值。

位 11:0 **DACC1DHR[11:0]**：DAC 1 通道 12 位右对齐数据 (DAC channel1 12-bit right-aligned data)
这些位由软件写入，用于为 DAC 1 通道指定 12 位数据。

图 27.1.5 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 STM32F7 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 stm32f7xx_hal_dac.c 以及头文件 stm32f7xx_hal_dac.h 中。实现上面功能的详细设置步骤如下：

1）开启 DAC 和 PA 口时钟，设置 PA4 为模拟输入。

STM32F7 的 DAC 通道 1 是接在 PA4 上的，所以，我们先要使能 GPIOA 的时钟，然后设置 PA4 为模拟输入。

这里需要特别说明一下，虽然 DAC 引脚设置为输入，但是 STM32F7 内部会连接在 DAC 模拟输出上,这在我们引脚复用映射章节有讲解。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);
```

对于 DAC 通道与引脚对应关系，这在 STM32F7 的数据手册引脚表上有列出，如下图：

PA4	I/O	TT a	(4)	SPI1_NSS/I2S1_WS, SPI3_NSS/I2S3_WS, USART2_CK, OTG_HS_SOF, DCMI_HSYNC, LCD_VSYNC, EVENTOUT	ADC12_IN4, DAC_OUT1
PA5	I/O	TT a	(4)	TIM2_CH1/TIM2_ETR, TIM8_CH1N, SPI1_SCK/I2S1_CK, OTG_HS_ULPI_CK, LCD_R4, EVENTOUT	ADC12_IN5, DAC_OUT2

图 26.1.6 DAC 通道引脚对应关系

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 STM32F7 的 DAC 通道 1 来输出不同的模拟电压了。

27.2 硬件设计

本章用到的硬件资源有：

1） 指示灯 DS0
2） KEY_UP 和 KEY1 按键
3） 串口
4） LCD 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 5 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 27.2.1 所示：

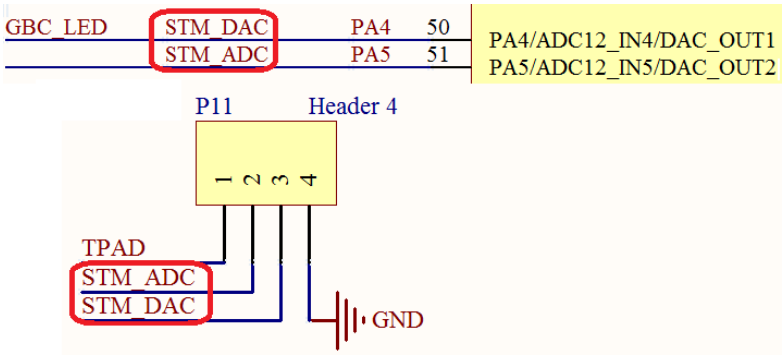


图 27.2.1 ADC、DAC 与 STM32F767 连接原理图

P11 是多功能端口，我们只需要通过跳线帽短接 P11 的 ADC 和 DAC，就可以开始做本章实验了。如图 27.2.2 所示：

图 27.2.2 硬件连接示意图

27.3 软件设计

打开本章实验工程可以发现，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 stm32f7xx_hal_dac.c 以及包含头文件 stm32f7xx_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 3 个函数，Dac1_Init 函数用于初始化 DAC 通道 1,并开启 DAC 通道。这里基本上是按我们上面的步骤 2 和步骤 3 来实现的。函数 HAL_DAC_MspInit 是 DAC 的 MSP 初始化回调函数，内部实现的是时钟使能和 IO 口配置，它和 Dac1_Init 配合使用来初始化整个 DAC 通道。经过初始化之后，我们就可以正常使用 DAC 通道 1 了。第三个函数 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;
        Cache_Enable();                 //打开 L1-Cache
        Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
        delay_init(216);                //初始化延时函数
        …//此处省略部分初始化代码
        MY_ADC_Init();                  //初始化 ADC1
        DAC1_Init();                    //初始化 DAC1
        ……//此处省略部分液晶显示代码
        HAL_DAC_SetValue(&DAC1_Handler,DAC_CHANNEL_1,DAC_ALIGN_12B_R,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_5,10); //得到 ADC 转换值
                temp=(float)adcx*(3.3/4096);            //得到 ADC 电压值
                adcx=temp;
                LCD_ShowxNum(94,190,temp,1,16,0);       //显示电压值整数部分
                temp-=adcx;
                temp*=1000;
                LCD_ShowxNum(110,190,temp,3,16,0X80);      //显示电压值的小数部分
                LED0_Toggle;
                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。

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 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 27.4.1 所示：

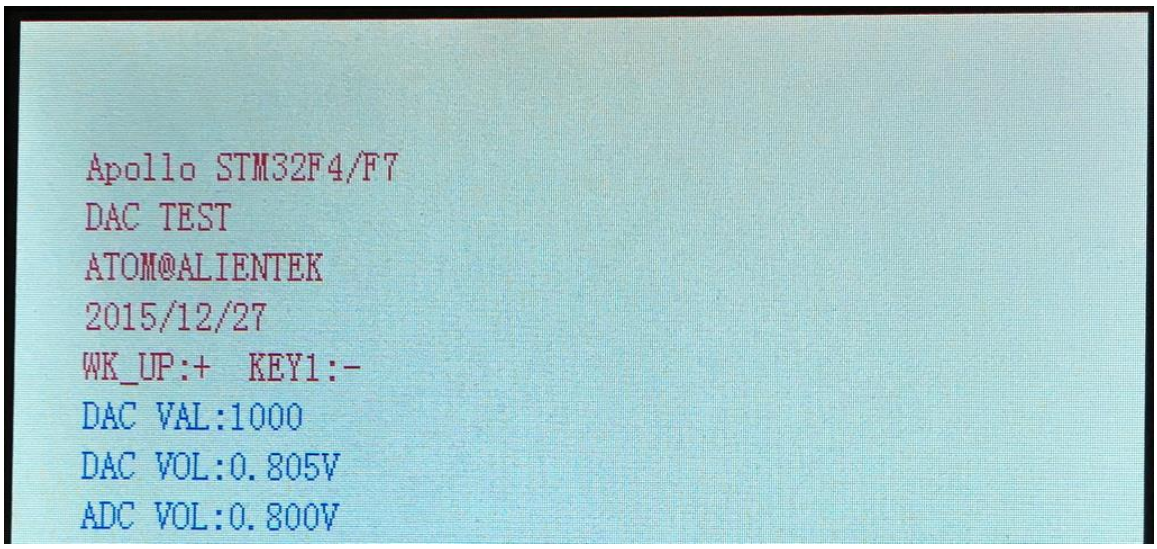


图 27.4.1 DAC 实验测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。此时，我们通过按 KEY_UP 按键，可以看到输出电压增大，按 KEY1 则变小。

第二十八章 PWM DAC 实验

上一章，我们介绍了 STM32F767 自带 DAC 模块的使用，但有时候，可能两个 DAC 不够用，此时，我们可以通过 PWM+RC 滤波来实一个 PWM DAC。本章我们将向大家介绍如何使用 STM32F767 的 PWM 来设计一个 DAC。我们将使用按键（或 USMART）控制 STM32F767 的 PWM 输出，从而控制 PWM DAC 的输出电压，通过 ADC1 的通道 5 采集 PWM DAC 的输出电压，并在 LCD 模块上面显示 ADC 获取到的电压值以及 PWM DAC 的设定输出电压值等信息。本章将分为如下几个部分：

28.1 PWM DAC 简介
28.2 硬件设计
28.3 软件设计
28.4 下载验证

28.1 PWM DAC 简介

有时候，STM32F767 自带的 2 路 DAC 可能不够用，需要多路 DAC，外扩 DAC 成本又会高不少。此时，我们可以利用 STM32F767 的 PWM+简单的 RC 滤波来实现 DAC 输出，从而节省成本。 在精度要求不是很高的时候，PWM+RC 滤波的 DAC 输出方式，是一种非常廉价的解决方案。

PWM 本质上其实就是是一种周期一定，而高低电平占空比可调的方波。实际电路的典型 PWM 波形，如图 28.1.1 所示：

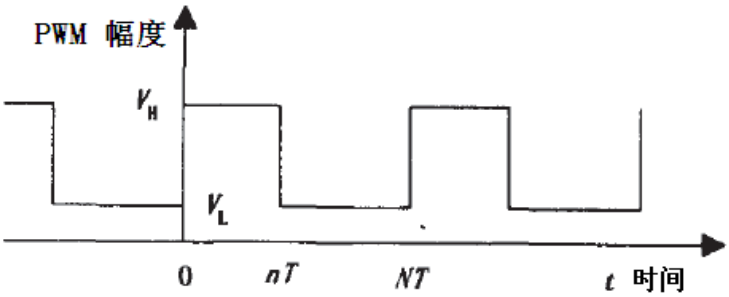


图 28.1.1 实际电路典型 PWM 波形

图 28.1.1 的 PWM 波形可以用分段函数表示为式①：

$$f(t)=\begin{cases}V_H & kNT \leqslant t \leqslant nT+kNT \\ V_L & kNT+nT \leqslant t \leqslant NT+kNT\end{cases} \quad ①$$

其中：T 是单片机中计数脉冲的基本周期，也就是 STM32F767 定时器的计数频率的倒数。N 是 PWM 波一个周期的计数脉冲个数，也就是 STM32F767 的 ARR-1 的值。n 是 PWM 波一个周期中高电平的计数脉冲个数，也就是 STM32F767 的 CCRx 的值。VH 和 VL 分别是 PWM 波的高低电平电压值，k 为谐波次数，t 为时间。我们将①式展开成傅里叶级数，得到公式②：

$$f(t)=\left[\frac{n}{N}(V_H-V_L)+V_L\right]+$$
$$2\frac{V_H-V_L}{\pi}\sin\left(\frac{n}{N}\pi\right)\cos\left(\frac{2\pi}{NT}t-\frac{n\pi}{N}k\right)+ \qquad ②$$
$$\sum_{k=2}^{\infty}2\frac{V_H-V_L}{k\pi}\left|\sin\left(\frac{n\pi}{N}k\right)\right|\cos\left(\frac{2\pi}{NT}kt-\frac{n\pi}{N}k\right)$$

从②式可以看出，式中第 1 个方括弧为直流分量，第 2 项为 1 次谐波分量，第 3 项为大于 1 次的高次谐波分量。式②中的直流分量与 n 成线性关系，并随着 n 从 0 到 N，直流分量从 VL 到 VL+VH 之间变化。这正是电压输出的 DAC 所需要的。因此，如果能把式②中除直流分量外的谐波过滤掉，则可以得到从 PWM 波到电压输出 DAC 的转换，即：PWM 波可以通过一个低通滤波器进行解调。式②中的第 2 项的幅度和相角与 n 有关，频率为 1/(NT)，其实就是 PWM 的输出频率。该频率是设计低通滤波器的依据。如果能把 1 次谐波很好过滤掉，则高次谐波就应该基本不存在了。

通过上面的了解，我们可以得到 PWM DAC 的分辨率，计算公式如下：

分辨率=log2（N）

这里假设 n 的最小变化为 1，当 N=256 的时候，分辨率就是 8 位。而 STM32F767 的定时器大部分都是 16 位的（TIM2 和 TIM5 是 32 位），可以很容易得到更高的分辨率，分辨率越高，速度就越慢。不过我们在本章要设计的 DAC 分辨率为 8 位。

在 8 位分辨条件下，我们一般要求 1 次谐波对输出电压的影响不要超过 1 个位的精度，也就是 3.3/256=0.01289V。假设 VH 为 3.3V，VL 为 0V，那么一次谐波的最大值是 2*3.3/π=2.1V，这就要求我们的 RC 滤波电路提供至少-20lg(2.1/0.01289)=-44dB 的衰减。

STM32F767 的定时器最快的计数频率是 216Mhz，某些定时器只能到 108M，所以我们以 108M 频率为例介绍，8 位分辨率的时候，PWM 频率为 108M/256=421.875Khz。如果是 1 阶 RC 滤波，则要求截止频率 2.66Khz，如果为 2 阶 RC 滤波，则要求截止频率为 33.62Khz。

阿波罗 STM32F767 开发板的 PWM DAC 输出采用二阶 RC 滤波，该部分原理图如图 28.1.2 所示：

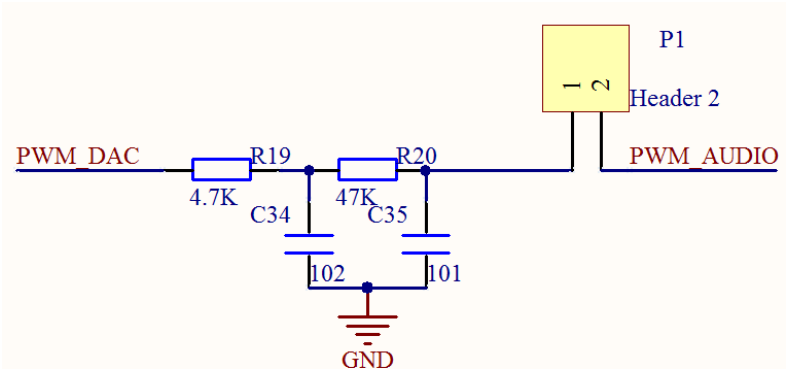


图 28.1.2 PWM DAC 二阶 RC 滤波原理图

二阶 RC 滤波截止频率计算公式为：

f=1/2πRC

以上公式要求 R28*C37=R29*C38=RC。根据这个公式，我们计算出图 28.1.2 的截止频率为：33.8Khz，和 33.62Khz 非常接近，满足设计要求。

PWM DAC 的原理部分，就为大家介绍到这里。

28.2 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） LCD 模块

5） ADC

6） PWM DAC

本章，我们使用 STM32F767 的 TIM9_CH2(PA3)输出 PWM，经过二阶 RC 滤波后，转换为直流输出，实现 PWM DAC。同上一章一样，我们通过 ADC1 的通道 5（PA5）读取 PWM DAC 的输出，并在 LCD 模块上显示相关数值，通过按键和 USMART 控制 PWM DAC 的输出值。我们需要用到 ADC 采集 DAC 的输出电压，所以需要在硬件上将 PWM DAC 和 ADC 短接起来，PWM DAC 部分原理图如图 28.2.1 所示：

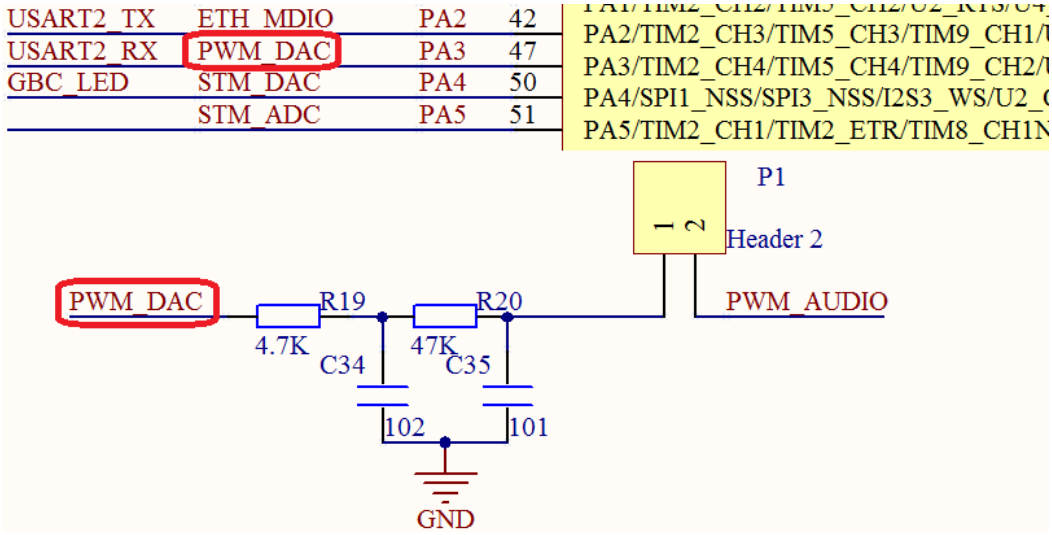


图 28.2.1 PWM DAC 原理图

从上图可知 PWM_DAC 的连接关系，但是这里有个**特别需要注意的地方**：因为 PWM_DAC 和 USART2_RX 共用了 PA3 引脚，所以在做本例程的时候，必须拔了 P8 上面 PA3(RX)的跳线帽（左侧跳线帽），否则会影响 PWM 转换结果!!!

在硬件上，我们还需要用跳线帽短接多功能端口的 PDC 和 ADC，如图 28.2.2 所示：

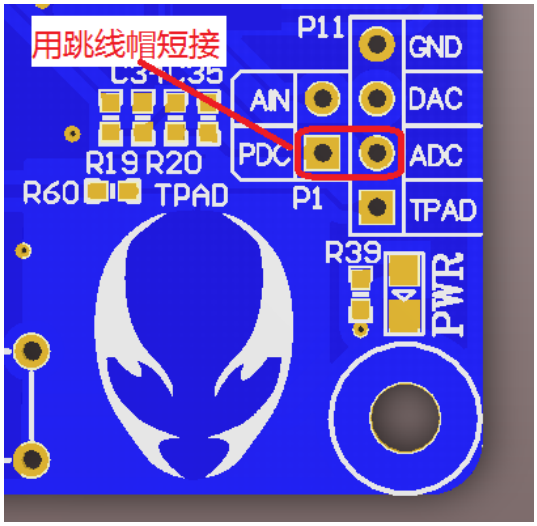


图 28.2.2 硬件连接示意图

28.3 软件设计

打开本章的实验工程可以看到，我们本章并没有增加其他新的库函数文件支持。主要是使用了 adc 和定时器相关的库函数支持。因为我们是使用定时器产生 PWM 信号作为 PWM DAC 的输入信号经过二阶 RC 滤波从而产生一定幅度模拟信号，所以我们需要添加定时器相关的库函数支持。在 HARDWARE 分组下，我们新建了 pwmdac.c 源文件和对应的头文件用来初始化定时器 9 的 PWM。接下来我们看看 pwmdac.c 源文件内容：

```
TIM_HandleTypeDef TIM9_Handler;          //定时器 9 PWM 句柄
TIM_OC_InitTypeDef TIM9_CH2Handler;      //定时器 9 通道 2 句柄

//PWM DAC 初始化(也就是 TIM9 通道 2 初始化)
//PWM 输出初始化
//arr：自动重装值
//psc：时钟预分频数
void TIM9_CH2_PWM_Init(u16 arr,u16 psc)
{
    TIM9_Handler.Instance=TIM9;                  //定时器 9
    TIM9_Handler.Init.Prescaler=psc;             //定时器分频系数
    TIM9_Handler.Init.CounterMode=TIM_COUNTERMODE_UP;//向上计数模式
    TIM9_Handler.Init.Period=arr;                //自动重装载值
    TIM9_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_PWM_Init(&TIM9_Handler);    //初始化 PWM

    TIM9_CH2Handler.OCMode=TIM_OCMODE_PWM1; //模式选择 PWM1
    TIM9_CH2Handler.Pulse=arr/2;            //设置比较值用来确定占空比为 50%
    TIM9_CH2Handler.OCPolarity=TIM_OCPOLARITY_HIGH; //输出比较极性为高
    HAL_TIM_PWM_ConfigChannel(&TIM9_Handler,&TIM9_CH2Handler,
                              TIM_CHANNEL_2);//配置 TIM9 通道 2
    HAL_TIM_PWM_Start(&TIM9_Handler,TIM_CHANNEL_2);//开启 PWM 通道 2
```

```
}

//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;
    __HAL_RCC_TIM9_CLK_ENABLE();            //使能定时器 9
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_3;            //PA3
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //复用推完输出
    GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;     //高速
    GPIO_Initure.Alternate= GPIO_AF3_TIM9;  //PA3 复用为 TIM9_CH2
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);
}

//设置 TIM 通道 2 的占空比
//TIM_TypeDef:定时器
//compare:比较值
void TIM_SetTIM9Compare2(u32 compare)
{
    TIM9->CCR2=compare;
}
```

该文件有三个函数，和 PWM 输出实验几乎是一模一样的，只不过定时器由 TIM3 换为了 TIM9，这里就不细说了。

接下来我们看看主函数内容：

```
//设置输出电压
//vol:0~330,代表 0~3.3V
void PWM_DAC_Set(u16 vol)
{
    double temp=vol;
    temp/=100;
    temp=temp*256/3.3;
  TIM_SetTIM9Compare2(temp);
}

int main(void)
{
    u16 adcx;
    float temp;
    u8 t=0;
```

```
u16 pwmval=0;
u8 key;
Cache_Enable();                 //打开 L1-Cache
HAL_Init();                     //初始化 HAL 库
Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
delay_init(216);                //延时初始化
uart_init(115200);              //串口初始化
usmart_dev.init(108);           //初始化 USMART
LED_Init();                     //初始化 LED
KEY_Init();                     //初始化按键
SDRAM_Init();                   //初始化 SDRAM
LCD_Init();                     //LCD 初始化
MY_ADC_Init();                  //初始化 ADC1

TIM9_CH2_PWM_Init(255,1); /TIM9 PWM 初始化, Fpwm=90M/256=351.562Khz.

POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
LCD_ShowString(30,70,200,16,16,"PWM DAC TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2016/1/13");
LCD_ShowString(30,130,200,16,16,"WK_UP:+    KEY1:-");
POINT_COLOR=BLUE;//设置字体为蓝色
LCD_ShowString(30,150,200,16,16,"DAC VAL:");
LCD_ShowString(30,170,200,16,16,"DAC VOL:0.000V");
LCD_ShowString(30,190,200,16,16,"ADC VOL:0.000V");
TIM_SetTIM9Compare2(pwmval);      //初始值为 0
while(1)
    {
            t++;
            key=KEY_Scan(0);
            if(key==WKUP_PRES)
            {
                    if(pwmval<250)pwmval+=10;
                    TIM_SetTIM9Compare2(pwmval);      //输出
            }else if(key==KEY1_PRES)
            {
                    if(pwmval>10)pwmval-=10;
                    else pwmval=0;
                    TIM_SetTIM9Compare2(pwmval);      //输出
            }
        if(t==10||key==KEY1_PRES||key==WKUP_PRES)
                    //WKUP/KEY1 按下了,或者定时时间到了
        {
```

```
                adcx=HAL_TIM_ReadCapturedValue(&TIM9_Handler,TIM_CHANNEL_2);
                        LCD_ShowxNum(94,150,adcx,3,16,0); //显示 DAC 寄存器值
                        temp=(float)adcx*(3.3/256);;        //得到 DAC 电压值
                        adcx=temp;
                        LCD_ShowxNum(94,170,temp,1,16,0); //显示电压值整数部分
                        temp-=adcx;
                        temp*=1000;
                        LCD_ShowxNum(110,170,temp,3,16,0x80);//显示小数部分
                        adcx=Get_Adc_Average(ADC_CHANNEL_5,20);  //得到转换值

                        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);//显示小数部分
                        t=0;
                        LED0_Toggle;
                }
                delay_ms(10);
        }
}
```

此部分代码，同上一章的基本一样，先对需要用到的模块进行初始化，然后显示一些提示信息，本章我们通过 KEY_UP 和 KEY1（也就是上下键）来实现对 PWM 脉宽的控制，经过 RC 滤波，最终实现对 DAC 输出幅值的控制。按下 KEY_UP 增加，按 KEY1 减小。同时在 LCD 上面显示 TIM4_CCR1 寄存器的值、PWM DAC 设计输出电压以及 ADC 采集到的实际输出电压。同时 DS0 闪烁，提示程序运行状况。

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 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 28.4.1 所示：

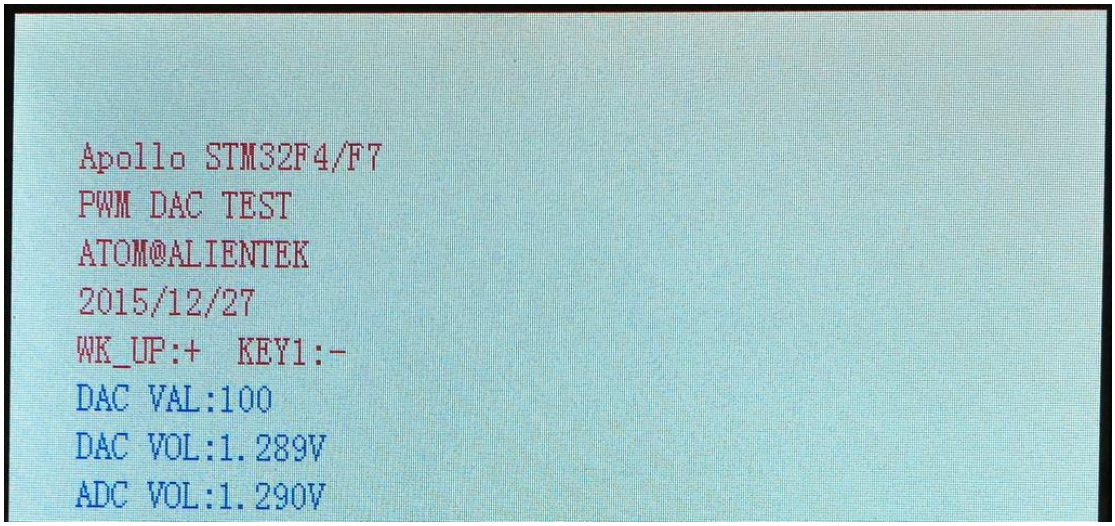


图 28.4.1 PWM DAC 实验测试图

同时伴随 DS0 的不停闪烁，提示程序在运行。此时，我们通过按 KEY_UP 按键，可以看到输出电压增大，按 KEY1 则变小。**特别提醒：此时 PA3 不能接其他任何外设，如果没有拔了 P8 排针上面 PA3 的跳线帽，那么 PWM DAC 将有很大误差！！！！**

第二十九章 DMA 实验

本章我们将向大家介绍 STM32F767 的 DMA。在本章中，我们将利用 STM32F767 的 DMA 来实现串口数据传送，并在 LCD 模块上显示当前的传送进度。本章分为如下几个部分：

29.1 STM32F767 DMA 简介
29.2 硬件设计
29.3 软件设计
29.4 下载验证

29.1 STM32F767 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。

STM32F767 最多有 2 个 DMA 控制器（DMA1 和 DMA2），共 16 个数据流（每个控制器 8 个），每一个 DMA 控制器都用于管理一个或多个外设的存储器访问请求。每个数据流总共可以有多达 8 个通道（或称请求）。每个数据流通道都有一个仲裁器，用于处理 DMA 请求间的优先级。

STM32F767 的 DMA 有以下一些特性：

● 双 AHB 主总线架构，一个用于存储器访问，另一个用于外设访问
● 仅支持 32 位访问的 AHB 从编程接口
● 每个 DMA 控制器有 8 个数据流，每个数据流有多达 8 个通道（或称请求）
● 每个数据流有单独的四级 32 位先进先出存储器缓冲区(FIFO)，可用于 FIFO 模式或直接模式。
● 通过硬件可以将每个数据流配置为：
1，支持外设到存储器、存储器到外设和存储器到存储器传输的常规通道
2，支持在存储器方双缓冲的双缓冲区通道
● 8 个数据流中的每一个都连接到专用硬件 DMA 通道（请求）
● DMA 数据流请求之间的优先级可用软件编程（4 个级别：非常高、高、中、低），在软件优先级相同的情况下可以通过硬件决定优先级（例如，请求 0 的优先级高于请求 1）
● 每个数据流也支持通过软件触发存储器到存储器的传输（仅限 DMA2 控制器）
● 可供每个数据流选择的通道请求多达 8 个。此选择可由软件配置，允许几个外设启动 DMA 请求
● 要传输的数据项的数目可以由 DMA 控制器或外设管理：
1，DMA 流控制器：要传输的数据项的数目是 1 到 65535，可用软件编程
2，外设流控制器：要传输的数据项的数目未知并由源或目标外设控制，这些外设通过硬件发出传输结束的信号
● 独立的源和目标传输宽度（字节、半字、字）：源和目标的数据宽度不相等时，DMA 自动封装/解封必要的传输数据来优化带宽。这个特性仅在 FIFO 模式下可用。
● 对源和目标的增量或非增量寻址
● 支持 4 个、8 个和 16 个节拍的增量突发传输。突发增量的大小可由软件配置，通常等于外设 FIFO 大小的一半
● 每个数据流都支持循环缓冲区管理
● 5 个事件标志（DMA 半传输、DMA 传输完成、DMA 传输错误、DMA FIFO 错误、

直接模式错误），进行逻辑或运算，从而产生每个数据流的单个中断请求

STM32F767 有两个 DMA 控制器，DMA1 和 DMA2，本章，我们仅针对 DMA2 进行介绍。STM32F767 的 DMA 控制器框图如图 29.1.1 所示：

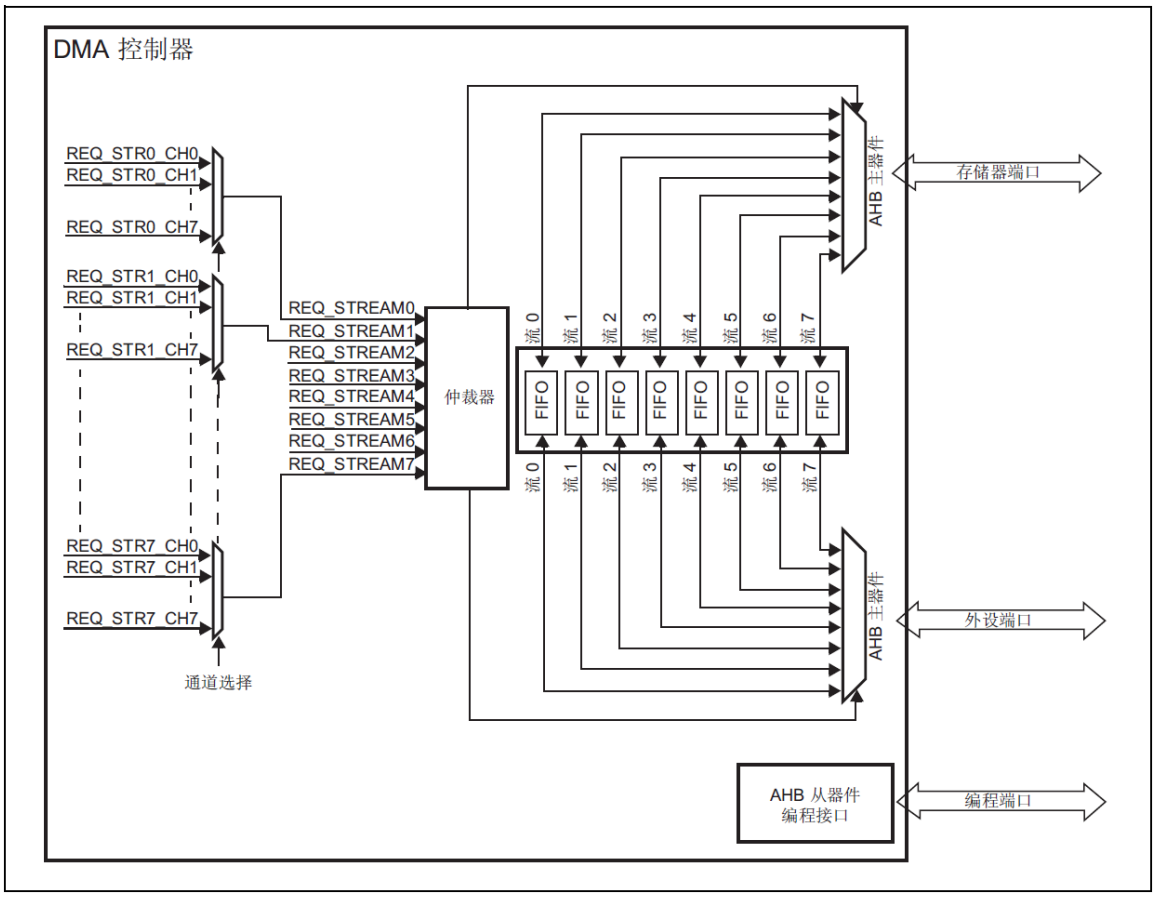


图 29.1.1 DMA 控制器框图

DMA 控制器执行直接存储器传输：因为采用 AHB 主总线，它可以控制 AHB 总线矩阵来启动 AHB 事务。它可以执行下列事务：

1，外设到存储器的传输

1， 存储器到外设的传输

3，存储器到存储器的传输

这里特别注意一下，存储器到存储器需要外设接口可以访问存储器，而仅 DMA2 的外设接口可以访问存储器，所以仅 DMA2 控制器支持存储器到存储器的传输，DMA1 不支持。

图 29.1.1 中数据流的多通道选择，是通过 DMA_SxCR 寄存器控制的，如图 29.1.2 所示：

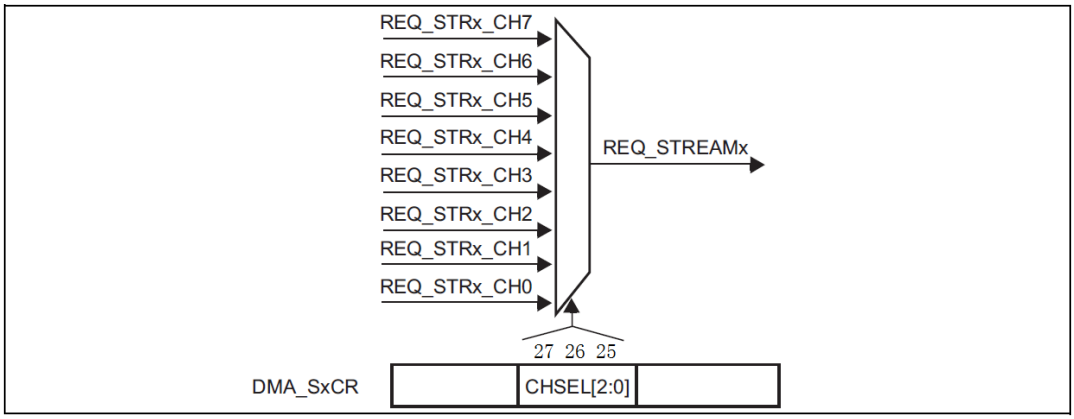


图 29.1.2 DMA 数据流通道选择

从上图可以看出，DMA_SxCR 控制数据流到底使用哪一个通道，每个数据流有 8 个通道可供选择，每次只能选择其中一个通道进行 DMA 传输。接下来，我们看看 DMA2 的各数据流通道映射表，如表 29.1.1 所示：

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1	SAI1_A	TIM8_CH1 TIM8_CH2 TIM8_CH3	SAI1_A	ADC1	SAI1_B	TIM1_CH1 TIM1_CH2 TIM1_CH3	SAI2_B
通道 1	-	DCMI	ADC2	ADC2	SAI1_B	SPI6_TX	SPI6_RX	DCMI
通道 2	ADC3	ADC3	-	SPI5_RX	SPI5_TX	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX	-	SPI1_RX	SPI1_TX	SAI2_A	SPI1_TX	SAI2_B	QUADSPI
通道 4	SPI4_RX	SPI4_TX	USART1_RX	SDMMC1	-	USART1_RX	SDMMC1	USART1_TX
通道 5	-	USART6_RX	USART6_RX	SPI4_RX	SPI4_TX	-	USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	-
通道 7	-	TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX	SPI5_TX	TIM8_CH4 TIM8_TRIG TIM8_COM

表 29.1.1 DMA2 各数据流通道映射表

上表就列出了 DMA2 所有可能的选择情况，来总共 64 种组合，比如本章我们要实现串口 1 的 DMA 发送，即 USART1_TX，就必须选择 DMA2 的数据流 7，通道 4，来进行 DMA 传输。这里注意一下，有的外设（比如 USART1_RX）可能有多个通道可以选择，大家随意选择一个就可以了。

接下来，我们介绍一下 DMA 设置相关的几个寄存器。

第一个是 DMA 中断状态寄存器，该寄存器总共有 2 个：DMA_LISR 和 DMA_HISR，每个寄存器管理 4 数据流（总共 8 个），DMA_LISR 寄存器用于管理数据流 0~3，而 DMA_HISR 用于管理数据流 4~7。这两个寄存器各位描述都完全一模一样，只是管理的数据流不一样。

这里，我们仅以 DMA_LISR 寄存器为例进行介绍，DMA_LISR 各位描述如图 29.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TCIF3	HTIF3	TEIF3	DMEIF3	Reserved	FEIF3	TCIF2	HTIF2	TEIF2	DMEIF2	Reserved	FEIF2
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
Reserved				TCIF1	HTIF1	TEIF1	DMEIF1	Reserved	FEIF1	TCIF0	HTIF0	TEIF0	DMEIF0	Reserved	FEIF0
r	r	r	r	r	r	r	r		r	r	r	r	r		r

位 31:28、15:12 保留，必须保持复位值。

位 27、21、11、5 **TCIFx**：数据流 x 传输完成中断标志 (Stream x transfer complete interrupt flag) (x = 3..0)
此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。
0：数据流 x 上无传输完成事件　　1：数据流 x 上发生传输完成事件

位 26、20、10、4 **HTIFx**：数据流 x 半传输中断标志 (Stream x half transfer interrupt flag) (x=3..0)
此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。
0：数据流 x 上无半传输事件　　1：数据流 x 上发生半传输事件

位 25、19、9、3 **TEIFx**：数据流 x 传输错误中断标志 (Stream x transfer error interrupt flag) (x=3..0)
此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。
0：数据流 x 上无传输错误　　1：数据流 x 上发生传输错误

位 24、18、8、2 **DMEIFx**：数据流 x 直接模式错误中断标志 (Stream x direct mode error interrupt flag) (x=3..0)
此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。
0：数据流 x 上无直接模式错误　　1：数据流 x 上发生直接模式错误

位 23、17、7、1 保留，必须保持复位值。

位 22、16、6、0 **FEIFx**：数据流 x FIFO 错误中断标志 (Stream x FIFO error interrupt flag) (x=3..0)
此位将由硬件置 1，由软件清零，软件只需将 1 写入 DMA_LIFCR 寄存器的相应位。
0：数据流 x 上无 FIFO 错误事件　　1：数据流 x 上发生 FIFO 错误事件

图 29.1.3 DMA_LISR 寄存器各位描述

如果开启了 DMA_LISR 中这些位对应的中断，则在达到条件后就会跳到中断服务函数里面去，即使没开启，我们也可以通过查询这些位来获得当前 DMA 传输的状态。这里我们常用的是 TCIFx 位，即数据流 x 的 DMA 传输完成与否标志。注意此寄存器为只读寄存器，所以在这些位被置位之后，只能通过其他的操作来清除。DMA_HISR 寄存器各位描述通 DMA_LISR 寄存器各位描述完全一样，只是对应数据流 4~7，这里我们就不列出来了。

第二个是 DMA 中断标志清除寄存器，该寄存器同样有 2 个：DMA_LIFCR 和 DMA_HIFCR，同样是每个寄存器控制 4 个数据流，DMA_LIFCR 寄存器用于管理数据流 0~3，而 DMA_ HIFCR 用于管理数据流 4~7。这两个寄存器各位描述都完全一模一样，只是管理的数据流不一样。

这里，我们仅以 DMA_LIFCR 寄存器为例进行介绍，DMA_LIFCR 各位描述如图 29.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CTCIF3	CHTIF3	CTEIF3	CDMEIF3	Reserved	CFEIF3	CTCIF2	CHTIF2	CTEIF2	CDMEIF2	Reserved	CFEIF2
				w	w	w	w		w	w	w	w	w		w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				CTCIF1	CHTIF1	CTEIF1	CDMEIF1	Reserved	CFEIF1	CTCIF0	CHTIF0	CTEIF0	CDMEIF0	Reserved	CFEIF0
				w	w	w	w		w	w	w	w	w		w

位 31:28、15:12 保留，必须保持复位值。

位 27、21、11、5 **CTCIFx**：数据流 x 传输完成中断标志清零 (Stream x clear transfer complete interrupt flag) (x = 3..0)
将 1 写入此位时，DMA_LISR 寄存器中相应的 TCIFx 标志将清零

位 26、20、10、4 **CHTIFx**：数据流 x 半传输中断标志清零 (Stream x clear half transfer interrupt flag) (x = 3..0)
将 1 写入此位时，DMA_LISR 寄存器中相应的 HTIFx 标志将清零

位 25、19、9、3 **CTEIFx**：数据流 x 传输错误中断标志清零 (Stream x clear transfer error interrupt flag) (x = 3..0)
将 1 写入此位时，DMA_LISR 寄存器中相应的 TEIFx 标志将清零

位 24、18、8、2 **CDMEIFx**：数据流 x 直接模式错误中断标志清零 (Stream x clear direct mode error interrupt flag) (x = 3..0)
将 1 写入此位时，DMA_LISR 寄存器中相应的 DMEIFx 标志将清零

位 23、17、7、1 保留，必须保持复位值。

位 22、16、6、0 **CFEIFx**：数据流 x FIFO 错误中断标志清零 (Stream x clear FIFO error interrupt flag) (x = 3..0)
将 1 写入此位时，DMA_LISR 寄存器中相应的 CFEIFx 标志将清零

图 29.1.4 DMA_LIFCR 寄存器各位描述

DMA_LIFCR 的各位就是用来清除 DMA_LISR 的对应位的，通过写 1 清除。在 DMA_LISR 被置位后，我们必须通过向该位寄存器对应的位写入 1 来清除。DMA_HIFCR 的使用同 DMA_LIFCR 类似，这里就不做介绍了。

第三个是 DMA 数据流 x 配置寄存器（DMA_SxCR）（x=0~7，下同）。该寄存器的我们在这里就不贴出来了，见《STM32F7 中文参考手册》第 229 页 8.5.5 一节。该寄存器控制着 DMA 的很多相关信息，包括数据宽度、外设及存储器的宽度、优先级、增量模式、传输方向、中断允许、使能等都是通过该寄存器来设置的。所以 DMA_ SxCR 是 DMA 传输的核心控制寄存器。

第四个是 DMA 数据流 x 数据项数寄存器（DMA_SxNDTR）。这个寄存器控制 DMA 数据流 x 的每次传输所要传输的数据量。其设置范围为 0~65535。并且该寄存器的值会随着传输的进行而减少，当该寄存器的值为 0 的时候就代表此次数据传输已经全部发送完成了。所以可以通过这个寄存器的值来知道当前 DMA 传输的进度。特别注意，这里是数据项数目，而不是指的字节数。比如设置数据位宽为 16 位，那么传输一次（一个项）就是 2 个字节。

第五个是 DMA 数据流 x 的外设地址寄存器（DMA_SxPAR）。该寄存器用来存储 STM32F767 外设的地址，比如我们使用串口 1，那么该寄存器必须写入 0x40011028（其实就是&USART1_TDR）。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 数据流 x 的存储器地址寄存器，由于 STM32F767 的 DMA 支持双缓存，所以存储器地址寄存器有两个：DMA_SxM0AR 和 DMA_SxM1AR，其中 DMA_SxM1AR 仅在双缓冲模式下，才有效。本章我们没用到双缓冲模式，所以存储器地址寄存器就是：DMA_SxM0AR，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[7800]数组来做存储器，那么我们在 DMA_SxM0AR 中写入&SendBuff 就可以了。

DMA 相关寄存器就为大家介绍到这里，关于这些寄存器的详细描述，请参考《STM32F7xx 中文参考手册》第 8.5 节。本章我们要用到串口 1 的发送，属于 DMA2 的数据流 7，通道 4，接下来我们就介绍下使用 HAL 库的配置步骤和方法。首先这里我们需要指出的是，DMA 相关的库函数支持在文件 stm32f7xx_hal_dma.c/stm32f7xx_hal_dma_ex.c 以及对应的头文件中，同时因

为我们是用串口的 DMA 功能，所以还要加入串口相关的文件 stm32f7xx_hal_uart.c。具体步骤如下：

1）使能 DMA2 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_DMA2_CLK_ENABLE();//DMA2 时钟使能
__HAL_RCC_DMA1_CLK_ENABLE();//DMA1 时钟使能
```

2）初始化 DMA2 数据流 7，包括配置通道，外设地址，存储器地址，传输数据量等。

DMA 的某个数据流各种配置参数初始化是通过 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                       (* XferM1HalfCpltCallback)(
                                      struct __DMA_HandleTypeDef * hdma);
   void                       (* XferErrorCallback)(
                                      struct __DMA_HandleTypeDef * hdma);
   void                       (* XferAbortCallback)(
                                      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 DMA2 的数据流 7，那么取值为 DMA2_Stream7。

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），XferM1HalfCpltCallback（Memory1 半传输完成回调函数），XferErrorCallback（传输错误回调函数）和 XferAbortCallback（传输中断回调函数）是六个函数指针，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 Channel; //通道，例如：DMA_CHANNEL_4
  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
  uint32_t FIFOMode;//FIFO 模式开启或者禁止
  uint32_t FIFOThreshold; //FIFO 阈值选择：
  uint32_t MemBurst; //存储器突发模式：单次/4 个节拍/8 个节拍/16 个节拍
  uint32_t PeriphBurst; //外设突发模式：单次/4 个节拍/8 个节拍/16 个节拍
}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_Stream7 的 DMA_CHANNEL_4，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= DMA2_Stream7;  //数据流选择
UART1TxDMA_Handler.Init.Channel=DMA_CHANNEL_4;   //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：8 位
UART1TxDMA_Handler.Init.MemDataAlignment=DMA_MDATAALIGN_BYTE; //存储器：8 位
UART1TxDMA_Handler.Init.Mode=DMA_NORMAL;                //普通模式
UART1TxDMA_Handler.Init.Priority=DMA_PRIORITY_MEDIUM;    //中等优先级
UART1TxDMA_Handler.Init.FIFOMode=DMA_FIFOMODE_DISABLE;
UART1TxDMA_Handler.Init.FIFOThreshold=DMA_FIFO_THRESHOLD_FULL;
UART1TxDMA_Handler.Init.MemBurst=DMA_MBURST_SINGLE;    //存储器突发单次传输
UART1TxDMA_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE; //外设突发单次传输
```

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(DMA2_Stream7)的 DMA 发送，启动传输

串口 1 的 DMA 发送实际是串口控制寄存器 CR3 的位 7 来控制的，在 HAL 库中操作该寄存器来使能串口 DMA 发送的函数为 HAL_UART_Transmit_DMA，该函数声明如下：

```
HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart,
                                        uint8_t *pData, uint16_t Size);
```

这里大家需要注意，调用该函数后会开启相应的 DMA 中断，对于本章实验，我们是通过查询的方法获取数据传输状态，所以并没有做中断相关处理，也没有编写中断服务函数。

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）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);
```

同样，我们也可以设置对应的 DMA 数据流传输的数据量大小,函数为：

```
__HAL_DMA_SET_COUNTER(&UART1TxDMA_Handler,1000);
```

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。

5）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 DMA2_Stream7 的中断服务函数为 DMA2_Stream7_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
```

29.2 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） LCD 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 LCD 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 P4 口的 RXD 和 TXD 是否和 PA9 和 PA10 连接上，如果没有，请先连接。

29.3 软件设计

打开本章的实验工程可以看到，我们在 HALLIB 分组下面增加了 DMA 支持文件 stm32f7xx_dma.c，同时引入了 stm32f7xx_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

//DMAx 的各通道配置
//这里的传输形式是固定的,这点要根据不同的情况来修改
//从存储器->外设模式/8 位数据宽度/存储器增量模式
//DMA_Streamx:DMA 数据流,DMA1_Stream0~7/DMA2_Stream0~7
//chx:DMA 通道选择,@ref DMA_channel DMA_CHANNEL_0~DMA_CHANNEL_7
void MYDMA_Config(DMA_Stream_TypeDef *DMA_Streamx,u32 chx)
{
    if((u32)DMA_Streamx>(u32)DMA2)//得到当前 stream 是属于 DMA2 还是 DMA1
    {
        __HAL_RCC_DMA2_CLK_ENABLE();//DMA2 时钟使能
    }else
    {
        __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=DMA_Streamx;     //数据流选择
    UART1TxDMA_Handler.Init.Channel=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 8 位
    UART1TxDMA_Handler.Init.MemDataAlignment=DMA_MDATAALIGN_BYTE; //存储器: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
    UART1TxDMA_Handler.Init.FIFOMode=DMA_FIFOMODE_DISABLE;
    UART1TxDMA_Handler.Init.FIFOThreshold=DMA_FIFO_THRESHOLD_FULL;
    UART1TxDMA_Handler.Init.MemBurst=DMA_MBURST_SINGLE; //存储器突发单次传输
    UART1TxDMA_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE; //外设突发单次传输
```

```
    HAL_DMA_DeInit(&UART1TxDMA_Handler);   //把 DMA 寄存器设置为缺省值
    HAL_DMA_Init(&UART1TxDMA_Handler);     //初始化 DMA
}
```

该部分代码仅仅 1 个函数，MYDMA_Config 函数，基本上就是按照我们上面 29.1 小节介绍的步骤 1 和步骤 2 来使能 DMA 时钟和初始化 DMA 的，该函数是一个通用的 DMA 配置函数，DMA1/DMA2 的所有通道，都可以利用该函数配置，不过有些固定参数可能要适当修改（比如位宽，传输方向等）。该函数在外部只能修改 DMA 数据流编号和通道号，更多的其他设置只能在该函数内部修改。对照前面的配置步骤的详细讲解来分析这部分代码即可。

dma.h 头文件内容比较简单，主要是函数申明，这里我们不细说。

接下来我们看看那 main 函数如下：

```
#define SEND_BUF_SIZE 7800//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 Apollo STM32F4 DMA 串口实验"};
int main(void)
{
    u16 i;
    u8 t=0;
    u8 j,mask=0;
    float pro=0;                        //进度
    Cache_Enable();                     //打开 L1-Cache
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);       //设置时钟,216Mhz
    delay_init(216);                    //延时初始化
    uart_init(115200);                  //串口初始化
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //LCD 初始化

    MYDMA_Config(DMA2_Stream7,DMA_CHANNEL_4);//初始化 DMA
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/1/24");
    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);//开启 DMA 传输
            while(1)
            {
                if(__HAL_DMA_GET_FLAG(&UART1TxDMA_Handler,DMA_FLAG_TCIF3_7))
                                        //等待 DMA2_Steam7 传输完成
                {
                  __HAL_DMA_CLEAR_FLAG(&UART1TxDMA_Handler,DMA_FLAG_TCIF3_7);
                                  //清除 DMA2_Steam7 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!");//提示完成
        }
        i++;
        delay_ms(10);
        if(i==20)
        {
            LED0_Toggle;//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(&UART1TxDMA_Handler)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。

至此，DMA 串口传输的软件设计就完成了。

29.4 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 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 29.4.1 所示：

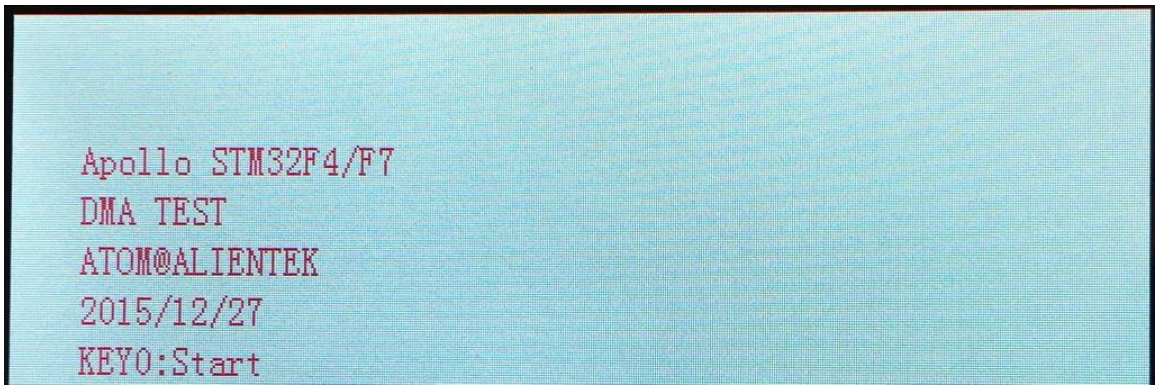


图 29.4.1 DMA 实验测试图

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 29.4.2 所示的内容：

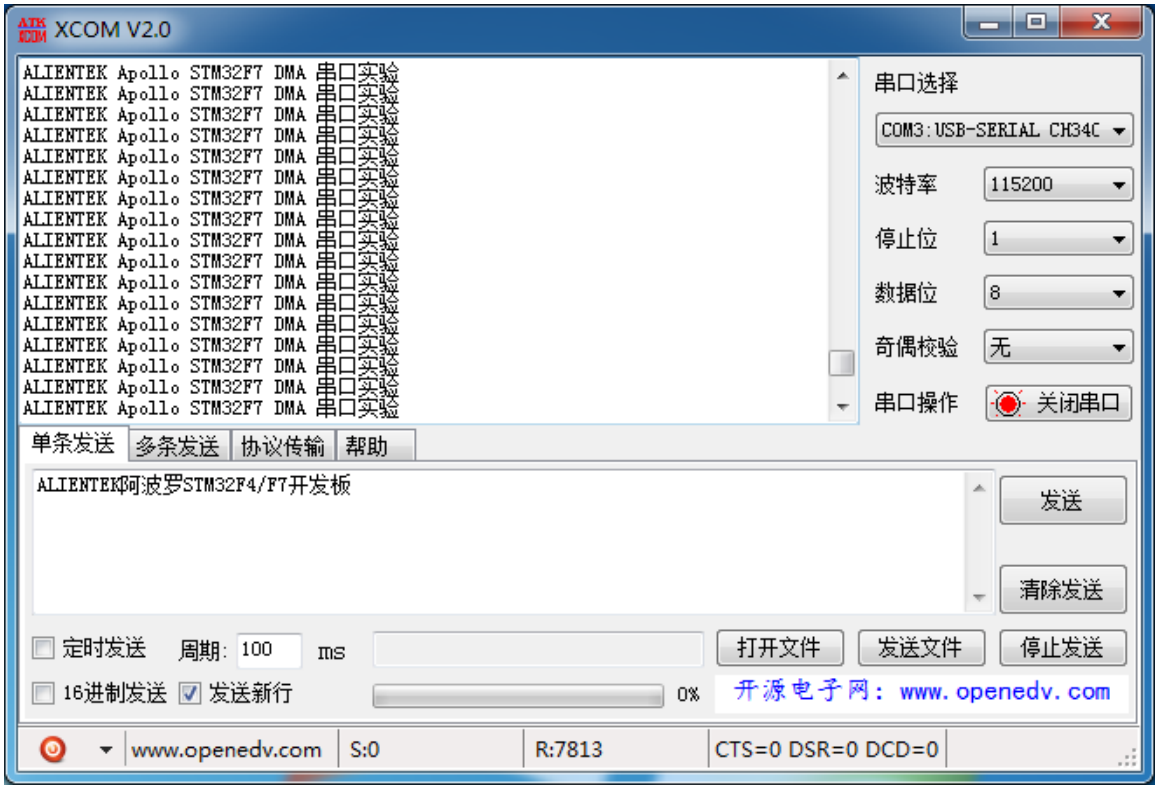
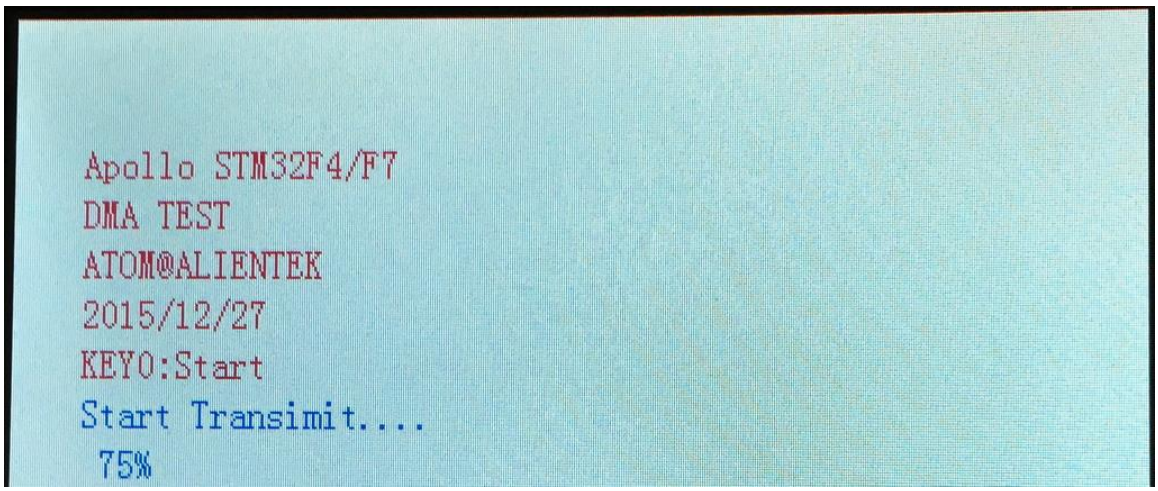


图 29.4.2 串口收到的数据内容

可以看到串口收到了阿波罗 STM32F767 开发板发送过来的数据，同时可以看到 TFTLCD 上显示了进度等信息，如图 29.4.3 所示：



29.4.3 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 STM32F767 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。

第三十章 IIC 实验

本章我们将向大家介绍如何使用 STM32F767 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 STM32F767 的普通 IO 口模拟 IIC 时序，来实现 24C02 的读写，并将结果显示在 LCD 模块上。本章分为如下几个部分：

30.1 IIC 简介
30.2 硬件设计
30.3 软件设计
30.4 下载验证

30.1 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 30.1..1 所示：

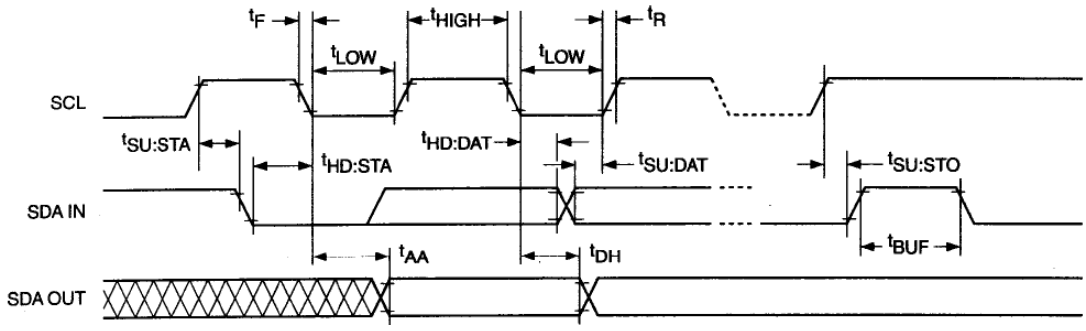


图 30.1.1 IIC 总线时序图

ALIENTEK 阿波罗 STM32F767 开发板板载的 EEPROM 芯片型号为 24C02。该芯片的总容量是 256 个字节，该芯片通过 IIC 总线与外部连接，我们本章就通过 STM32F767 来实现 24C02 的读写。

目前大部分 MCU 都带有 IIC 总线接口，STM32F767 也不例外。但是这里我们不使用 STM32F767 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 STM32F767 的硬件 IIC。

用软件模拟 IIC，最大的好处就是方便移植，同一个代码兼容所有 MCU，任何一个单片机

只要有 IO 口，就可以很快的移植过去，而且不需要特定的 IO 口。而硬件 IIC，则换一款 MCU，基本上就得重新搞一次，移植是比较麻烦的。

本章实验功能简介：开机的时候先检测 24C02 是否存在，然后在主循环里面检测两个按键，其中 1 个按键（KEY1）用来执行写入 24C02 的操作，另外一个按键（KEY0）用来执行读出操作，在 LCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

30.2 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） LCD 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 STM32F767 的连接，24C02 的 SCL 和 SDA 分别连在 STM32F767 的 PH4 和 PH5 上的，连接关系如图 30.2.1 所示：

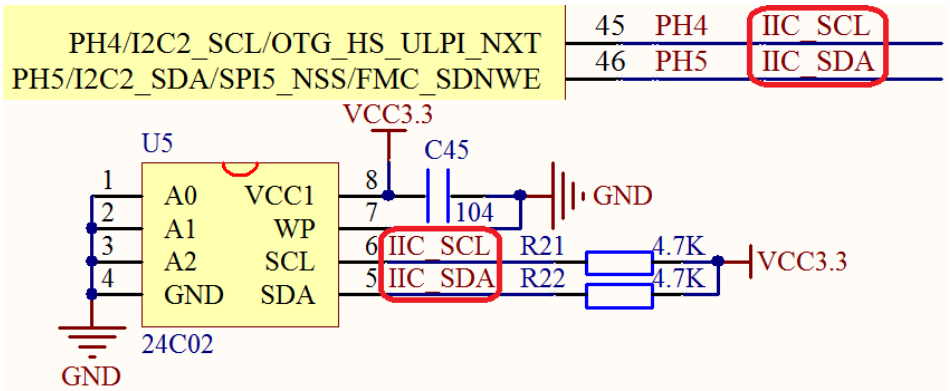


图 30.2.1 STM32F767 与 24C02 连接图

30.3 软件设计

打开本章的实验工程可以看到，我们并没有在 HALLIB 分组之下添加新的固件库文件支持，因为我们是通过 GPIO 来模拟 IIC。我们新增了 myiic.c 文件用来存放 iic 底层驱动。新增了 24cxx.c 文件用来存放 24C02 的底层驱动。

打开 myiic.c 文件，代码如下：

```
//IIC 初始化
void IIC_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_GPIOH_CLK_ENABLE();     //使能 GPIOH 时钟

    //PH4,5 初始化设置
    GPIO_Initure.Pin=GPIO_PIN_4|GPIO_PIN_5;
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
```

```
    GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FAST;         //快速
    HAL_GPIO_Init(GPIOH,&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()    {GPIOH->MODER&=~(3<<(5*2));GPIOH->MODER|=0<<5*2;}
                                                        //PH5 输入模式
    #define SDA_OUT() {GPIOH->MODER&=~(3<<(5*2));GPIOH->MODER|=1<<5*2;}
                                                        //PH5 输出模式
    //IO 操作
    #define IIC_SCL(n)    (n?HAL_GPIO_WritePin(GPIOH,GPIO_PIN_4,GPIO_PIN_SET): \
                        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_4,GPIO_PIN_RESET)) //SCL
    #define IIC_SDA(n)    (n?HAL_GPIO_WritePin(GPIOH,GPIO_PIN_5,GPIO_PIN_SET): \
                        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_5,GPIO_PIN_RESET)) //SDA
    #define READ_SDA       HAL_GPIO_ReadPin(GPIOH,GPIO_PIN_5)    //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，IIC_SCL(n), IIC_SDA(n)和 READ_SDA 分别用来设置 IIC 通信引脚的电平设置和状态读取，如果这几行代码大家不是很理解， 请仔细复习前面跑马灯实验中关于 IO 口操作函数讲解。

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 main 函数代码：

```
//要写入到 24c02 的字符串数组
const u8 TEXT_Buffer[]={"Apollo STM32F7 IIC TEST"};
#define SIZE sizeof(TEXT_Buffer)

int main(void)
{
    u8 key;
    u16 i=0;
    u8 datatemp[SIZE];
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    AT24CXX_Init();                 //初始化 24C02
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/7/12");
    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_Toggle;//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_Toggle;//提示系统正在运行
                i=0;
            }
        }
    }
```

该段代码，我们通过 KEY1 按键来控制 24C02 的写入，通过另外一个按键 KEY0 来控制 24C02 的读取。并在 LCD 模块上面显示相关信息。

至此，我们的软件设计部分就结束了。

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 阿波罗 STM32 开发板上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 30.4.1 所示：

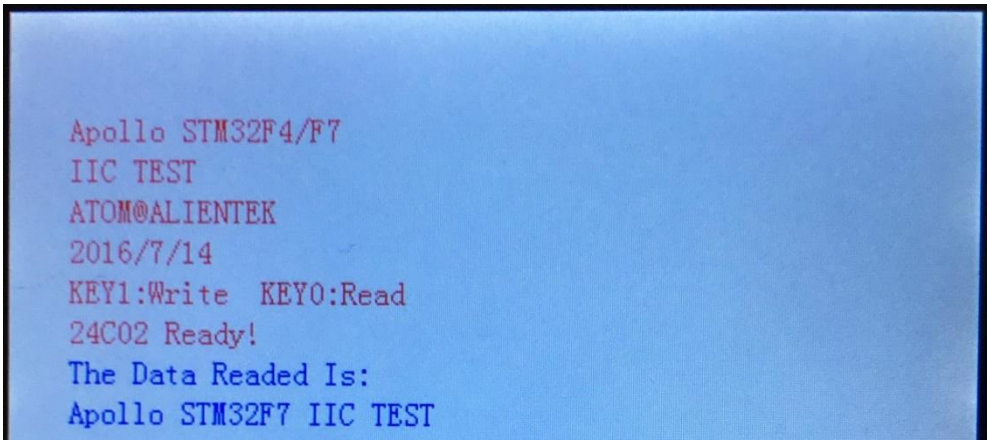


图 30.4.1 IIC 实验程序运行效果图

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 LCD 模块上显示错误信息，同时 DS0 慢闪。读者可以通过跳线帽把 PH4 和 PH5 短接就可以看到报错了。

第三十一章 IO 扩展实验

上一章，我们介绍了 IIC 驱动 24C02，本章我们将向大家介绍如何使用 IIC 来扩展 IO 口。在本章中，我们将使用 STM32F767 的普通 IO 口模拟 IIC 时序，来驱动 PCF8574/AT8574，达到扩展 IO 口的目的。本章分为如下几个部分：

31.1 PCF8574/AT8574 简介
31.2 硬件设计
31.3 软件设计
31.4 下载验证

31.1 PCF8574/AT8574 简介

PCF8574 是飞利浦公司推出的一款 IIC 接口的远程 I/O 扩展芯片，AT8574 则是芯景科技的产品，**PCF8574 和 AT8574 完全兼容，他们可以互相替换使用**，接下来的介绍和说明，我们仅以 PCF8754 为例做说明，AT8574 参考学习即可。

PCF8574 包含一个 8 位准双向口和一个 IIC 总线接口。PCF8574 电流消耗很低且口输出锁存具有大电流驱动能力可直接驱动 LED 它还带有一条中断接线（INT）可与 MCU 的中断逻辑相连，通过 INT 发送中断信号，远端 I/O 口不必经过 IIC 总线通信就可通知 MCU 是否有数据从端口输入，这意味着 PCF8574 可以作为一个单被控器。

PCF8574 有如下特性：

- 支持 2.5~6.0V 操作电压
- 低备用电流（≤10u A）
- 支持开漏中断输出
- 支持 IIC 总线扩展 8 路 I/O 口
- 口输出锁存具有大电流驱动能力可直接驱动 LED
- 通过 3 个硬件地址引脚可寻址 8 个器件

1，引脚说明

PCF8574 的引脚说明如表 31.1.1 所示：

标号	管脚 S016	描述
A0	1	地址输入 0
A1	2	地址输入 1
A2	3	地址输入 2
P0	4	准双向 I/O 口 0
P1	5	准双向 I/O 口 1
P2	6	准双向 I/O 口 2
P3	7	准双向 I/O 口 3
V_{SS}	8	地
P4	9	准双向 I/O 口 4
P5	10	准双向 I/O 口 5
P6	11	准双向 I/O 口 6
P7	12	准双向 I/O 口 7
$\overline{INT}$	13	中断输出（低电平有效）
SCL	14	串行时钟线
SDA	15	串行数据线
V_{DD}	16	电源

表 31.1.1 PCF8574 引脚说明

我们使用的 PCF8574T 采用 SO16 封装，总共 16 个脚，其中包括：8 个准双向 IO 口（P0~P7）、

3 个地址线（A0~A2）、SCL、SDA、INT、VDD 和 VSS。每个 PCF8574T 只需要最少 2 个 IO 口，就可以扩展 8 路 IO，且支持一个 IIC 总线上挂最多 8 个 PCF8574T，这样通过 2 个 IO，最多可以扩展 64 个 IO 口，在 MCU 的 IO 不够用的时候，PCF8574T 是一个非常不错的 IO 扩展方案。

2，寻址

一个 IIC 总线上，最多可以挂 8 个 PCF8574T（通过 A0~A2 寻址），PCF8574T 的从机地址格式如图 31.1.1 所示：

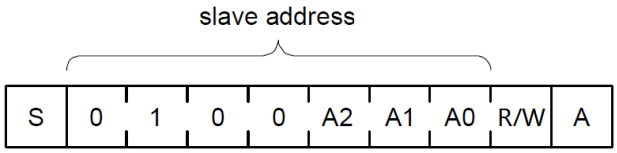


图 31.1.1 PCF8574T 从机地址格式

图中的 S 代表 IIC 的 Start 信号（启动信号）；A 代表 PCF8574T 发出的应答信号；A0~A2 为 PCF8574T 的寻址信息，我们开发板上 A0~A2 都是接 GND 的，所以，PCF8574T 的地址为：0X40（左移了一位）；R/W 为读/写控制位，R/W=0 的时候，表示写数据到 PCF8574T，输出到 P0~P7 口，R/W=1 的时候，表示读取 PCF8574T 的数据，获取 P0~P7 的 IO 口状态。

关于 IIC 协议的介绍，请参考上一章节。

3，写数据（输出）

PCF8574T 的写数据时序如图 31.1.2 所示：

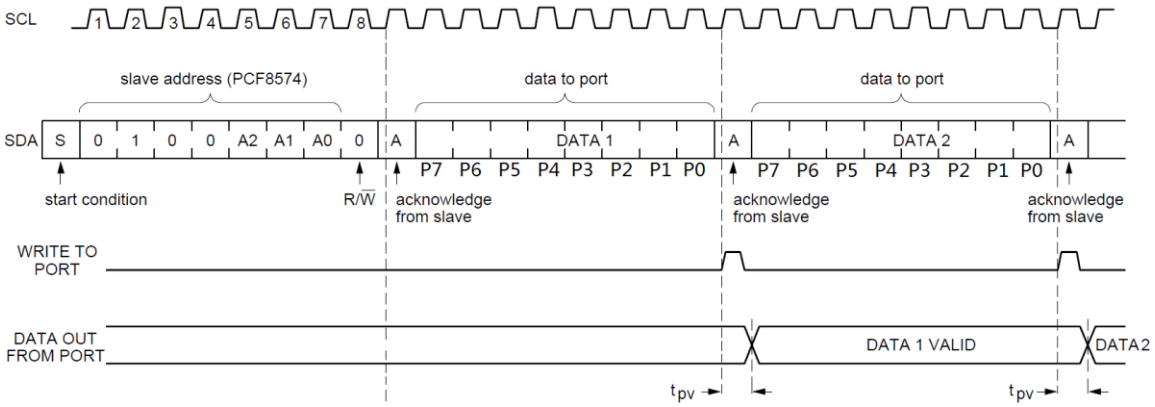


图 31.1.2 PCF8574T 写数据时序

由图可知，PCF8574T 的数据写入非常简单，首先发送 PCF8574T 的从机地址+写信号（R/W=0），然后等待 PCF8574T 的应答信号，在应答成功后，发送数据（DATA1）给 PCF8574T 就可以了，发送完数据，会受到 PCF8574T 的应答信号，在发送应答信号的同时，PCF8574T 会将接收到的数据（DATA1）输出到 P0~P7 上面（对应关系见上图）。注意：图中的 WRITE TO PORT 信号是 PCF8574T 内部自己产生的，它在每次发送应答的同时产生，用于将刚刚接收到的数据，输出到 P0~P7 上，此信号不需要 MCU 发送。

4，读数据（输入）

PCF8574T 的读数据时序如图 31.1.3 所示：

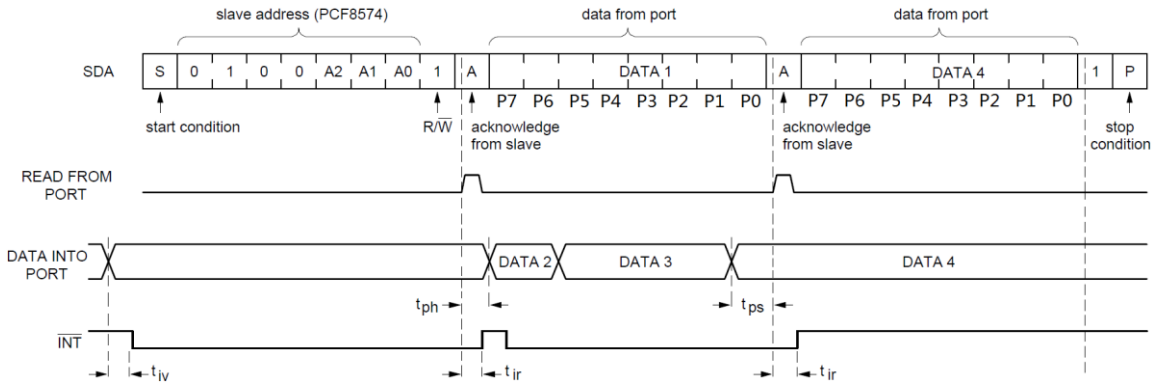


图 31.1.3 PCF8574T 读数据时序

PCF8574T 的读数据流程：首先发送 PCF8574T 的从机地址+读信号（R/W=1），然后等待 PCF8574T 应答（注意：PCF8574T 在发送应答的同时，会锁存 P0~P7 的数据），然后读取 P0~P7 的数据。数据读取支持连续读取，在最后的时候发送 STOP 信号，即可完成读数据操作。

这里需要注意的是：PCF8574T 的数据锁存（READ FROM PORT），发生在发送应答信号的时候，之后，P0~P7 发送的数据变化（比如图中的 DATA2 和 DATA3），将不会读取进来，直到下一个应答信号进行锁存。

5，中断

PCF8574T 带有中断输出脚，它可以连接到 MCU 的中断输入引脚上。在输入模式中（IO 口输出高电平，即可做输入使用），输入信的上升或下降沿都可以产生中断，在 tiv 时间之后 INT 有效。**特别注意：一旦中断有效后，必须对 PCF8574T 进行一次读取/写入操作，复位中断，才可以输出下一次中断，否则中断将一直保持（无法输出下一次输入信号变化所产生的中断）。**

关于 PCF8574 的简介，我们就介绍到这里。

本章实验功能简介：开机的时候先检测 PCF8574T 是否存在，然后在主循环里面检测 KEY0 按键和 PCF8574T 的中断信号，当 KEY0 按下时，控制 PCF8574T 的 P0 口输出，从而控制蜂鸣器（连接在 P0 口）的开关；当检测到 PCF8574T 的中断信号时，读取 EXIO（连接在 PCF8574T 的 P4 口）的状态，当 EXIO=0（即 P4=0）的时候，控制 LED1 的翻转。同时，LCD 模块上显示相关信息，并用 DS0 提示程序正在运行。另外，本例程将 PCF8574T 的相关控制函数加入 USMART 控制，我们也可以通过 USMART 控制/读取 PCF8574T。

31.2 硬件设计

本章需要用到的硬件资源有：

1） 指示灯 DS0

2）KEY0 按键

3）串口（USMART 使用）

4）LCD 模块

5）PCF8574T

6）蜂鸣器

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。这里介绍 PCF8574T 与 STM32F767 和蜂鸣器的连接，PCF8574T 同 24C02 等共用一个 IIC 接口，SCL 和 SDA 分别连在 STM32F767 的 PH4 和 PH5 上的，另外 INT 脚连接在 STM32F767 的 PB12 上面，连接关系如图 31.2.1 所示：

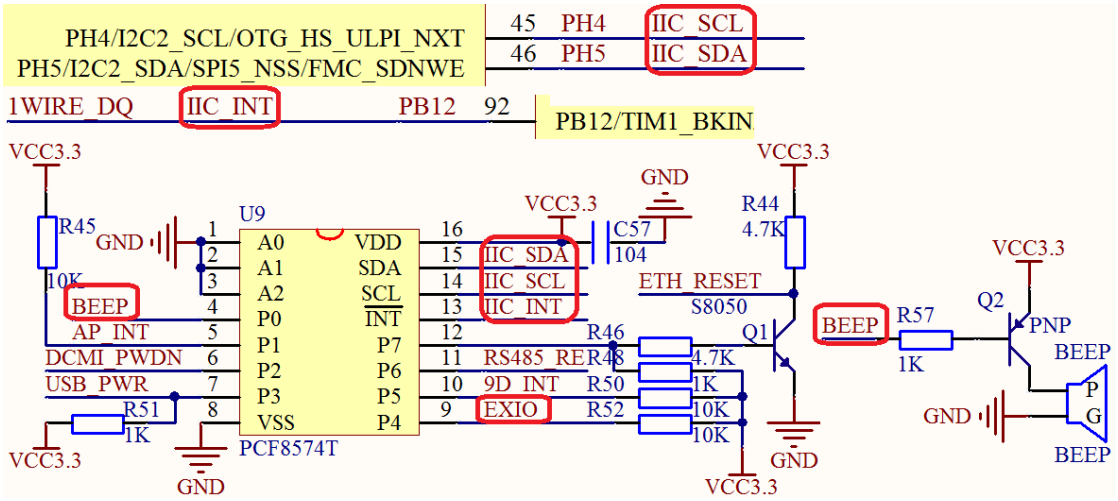


图 31.2.1 PCF8574T 与 STM32F767 和蜂鸣器的连接图

由图可知，蜂鸣器控制信号 BEEP 连接在 PCF8574T 的 P0 脚上，EXIO 连接在 P4 脚上，其他还连接了一些外设（比如网络复位脚、摄像头、USB、485 等），我们将在对应章节进行介绍，这里就不多说了。这里需要注意的是：IIC_INT 脚，同 1WIRE_DQ 共用了 PB12，在使用的时候，他们只能分时复用，不能同时使用。

31.3 软件设计

打开本章实验工程可以看到，由于 PCF8574 要使用到 IIC 接口，所以这里我们保留了上一章 icc 相关源码。同时还新建了 pcf8574.c 源文件和 pcf8574.h 头文件，PCF8574 相关的驱动代码就存放在这两个文件中。

打开 pcf8574.c 文件，代码如下：

```
//初始化 PCF8574
u8 PCF8574_Init(void)
{
    u8 temp=0;
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();          //使能 GPIOB 时钟

    GPIO_Initure.Pin=GPIO_PIN_12;          //PB12
    GPIO_Initure.Mode=GPIO_MODE_INPUT;     //输入
    GPIO_Initure.Pull=GPIO_PULLUP;         //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;    //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);    //初始化
    IIC_Init();                            //IIC 初始化
    //检查 PCF8574 是否在位
    IIC_Start();
    IIC_Send_Byte(PCF8574_ADDR);           //写地址
    temp=IIC_Wait_Ack(); //等待应答,通过判断是否有 ACK 应答,来判断 PCF8574 的状态
    IIC_Stop();                            //产生一个停止条件
    PCF8574_WriteOneByte(0XFF);            //默认情况下所有 IO 输出高电平
    return temp;
```

```
}

//读取 PCF8574 的 8 位 IO 值
//返回值:读到的数据
u8 PCF8574_ReadOneByte(void)
{
    u8 temp=0;

    IIC_Start();
    IIC_Send_Byte(PCF8574_ADDR|0X01);    //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;
}
//向 PCF8574 写入 8 位 IO 值
//DataToWrite:要写入的数据
void PCF8574_WriteOneByte(u8 DataToWrite)
{

    IIC_Start();
    IIC_Send_Byte(PCF8574_ADDR|0X00);    //发送器件地址 0X40,写数据
    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);
}

//设置 PCF8574 某个 IO 的高低电平
//bit:要设置的 IO 编号,0~7
//sta:IO 的状态;0 或 1
void PCF8574_WriteBit(u8 bit,u8 sta)
{
    u8 data;
    data=PCF8574_ReadOneByte(); //先读出原来的设置
    if(sta==0)data&=~(1<<bit);
    else data|=1<<bit;
    PCF8574_WriteOneByte(data); //写入新的数据
}

//读取 PCF8574 的某个 IO 的值
//bit：要读取的 IO 编号,0~7
```

```
//返回值:此 IO 的值,0 或 1
u8 PCF8574_ReadBit(u8 bit)
{
    u8 data;
    data=PCF8574_ReadOneByte(); //先读取这个 8 位 IO 的值
    if(data&(1<<bit))return 1;
    else return 0;
}
```

该部分为 PCF8574 的驱动代码，其中的 IIC 相关函数，直接是用的上一章 myiic.c 里面提供的相关函数，这里不做介绍。

这里总共有 5 个函数：PCF8574_Init 函数用于初始化并检测 PCF8574，这里我们初始化 PB12 为上拉输入，以检测 PCF8574T 的中断输出信号，另外，在该函数里面，我们通过检查 PCF8574 的应答信号来确认 PCF8574 是否正常（在位）；PCF8574_ReadOneByte 和 PCF8574_WriteOneByte 这两个函数用于读取/写入 PCF8574，从而读取/控制 P0~P7；最后，PCF8574_WriteBit 和 PCF8574_ReadBit 函数用于控制或者读取 PCF8574 的单个 IO。

pcf8547.h 头文件定义了 PCF8574 的中断检测脚、地址、每个 IO 连接的外设宏定义和相关操作函数声明。这里我们就不列出该头文件内容，请大家自行打开实验工程查看。

最后我们看看 main 函数，程序如下：

```
int main(void)
{
    u8 key;
    u16 i=0;
    u8 beepsta=1;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    …//此处省略部分代码
    LCD_ShowString(30,130,200,16,16,"KEY0:BEEP ON/OFF");  //显示提示信息
    LCD_ShowString(30,150,200,16,16,"EXIO:DS1 ON/OFF");   //显示提示信息
    while(PCF8574_Init())       //检测不到 PCF8574
    {
        LCD_ShowString(30,170,200,16,16,"PCF8574 Check Failed!");
        delay_ms(500);
        LCD_ShowString(30,170,200,16,16,"Please Check!      ");
        delay_ms(500);
        LED0_Toggle;//DS0 闪烁
    }
    LCD_ShowString(30,170,200,16,16,"PCF8574 Ready!");
    POINT_COLOR=BLUE;//设置字体为蓝色
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY0_PRES)//KEY0 按下,读取字符串并显示
```

```
            {
                beepsta=!beepsta;                       //蜂鸣器状态取反
                PCF8574_WriteBit(BEEP_IO,beepsta);//控制蜂鸣器
            }
            if(PCF8574_INT==0)                  //PCF8574 的中断低电平有效
            {
                key=PCF8574_ReadBit(EX_IO);
                    //读取 EXIO 状态,同时清除 PCF8574 的中断输出(INT 恢复高电平)
                if(key==0)LED1_Toggle;          //LED1 状态取反
            }
            i++;
            delay_ms(10);
            if(i==20)
            {
                LED0_Toggle;//提示系统正在运行
                i=0;
            }
        }
    }
```

该段代码，我们通过 KEY0 按键可以控制蜂鸣器的开关，另外，在 while 循环里面，会不停的检测 PCF8574 的中断引脚，是否有输出中断，如果有，就读取 EXIO 的状态（读操作会复位中断，以便检测下一个中断），当 EXIO=0 的时候控制 DS1 开关。同时，在 LCD 模块上面显示相关信息，并用 DS0 指示程序运行状态。

最后，我们将 PCF8574_ReadOneByte、PCF8574_WriteOneByte、PCF8574_ReadBit 和 PCF8574_WriteBit 这四个函数加入 USMART 控制，这样，我们就可以通过串口调试助手，控制 PCF8574，方便测试。

至此，我们的软件设计部分就结束了。

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 阿波罗 STM32 开发板上，得到如图 31.4.1 所示的界面：

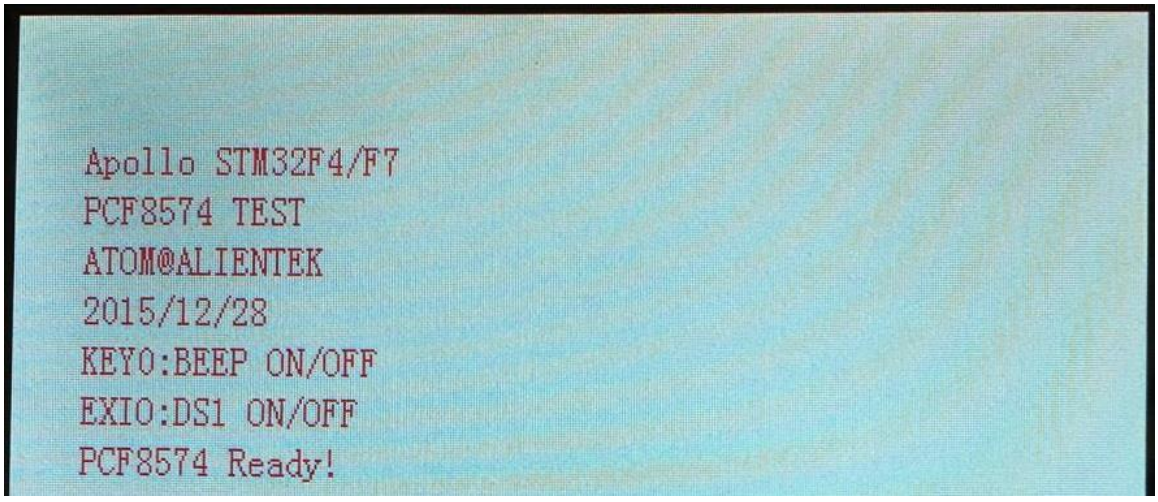


图 31.4.1 程序运行界面

屏幕提示 PCF8574 Ready!，表示 PCF8574 已经准备好，同时 DS0 会不停的闪烁，提示程序正在运行。此时，我们按按键 KEY0，就可以控制蜂鸣器的开和关。我也可以用一根杜邦线，连接 EXIO（在 P3 排针的最左下角）和 GND（短接一次 GND，改变一次 DS1 的状态），就可以控制 DS1 的开和关。

另外，本例程还可以用 USMART 调用 PCF8574 相关函数进行控制，大家可以自行测试下，我们这里就不给大家演示了。

第三十二章 光环境传感器实验

上一章，我们介绍了 IIC 驱动 PCF8574T，本章我们将向大家介绍如何使用 IIC 来驱动光环境传感器。在本章中，我们将使用 STM32F767 的普通 IO 口模拟 IIC 时序，来驱动 AP3216C，从而检测环境光强度（ALS）、接近距离(PS)和红外线强度(IR)等环境参数。本章分为如下几个部分：

32.1 AP3216C 简介
32.2 硬件设计
32.3 软件设计
32.4 下载验证

32.1 AP3216C 简介

AP3216C 是敦南科技退出的一款三合一环境传感器，它包含了：数字环境光传感器（ALS）、接近传感器（PS）和一个红外 LED（IR）。该芯片通过 IIC 接口和 MCU 连接，并支持中断（INT）输出。AP3216C 的特点如下：

- IIC 接口，支持高达 400Khz 通信速率
- 支持多种工作模式（ALS、PS+IR、ALS+PS+IR 等）
- 内置温度补偿电路
- 工作温度支持-30~80℃
- 环境光传感器具有 16 位分辨率
- 接近传感器具有 10 位分辨率
- 红外传感器具有 10 位分辨率
- 超小封装（4.1*2.4*1.35mm）

因为以上一些特性，AP3216C 被广泛应用于智能手机上面，用来检测光强度（自动背光控制），和接近开关控制（听筒靠近耳朵，手机自动灭屏功能）。 AP3216C 的框图如图 32.1.1 所示：

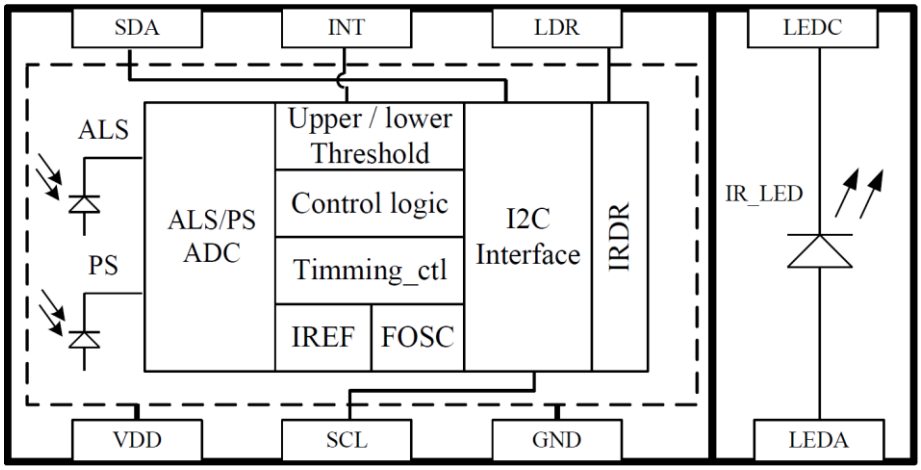


图 32.1.1 AP3216C 框图

1，引脚说明

AP3216C 的引脚说明如表 32.1.1 所示：

引脚编号	标号	说明
1	VDD	电源，接 3.3V
2	SCL	IIC 时钟信号，开漏
3	GND	地线
4	LEDA	LED 阳极，接 3.3V
5	LEDC	LED 阴极，一般连接 LDR
6	LDR	LED 驱动输出脚，一般接 LEDC
7	INT	中断输出脚
8	SDA	IIC 数据信号，开漏

表 32.1.1 AP3216C 引脚说明

AP3216C 和我们的 MCU 只需要连接 SCL、SDA 和 INT，就可以实现驱动。其 SCL 和 SDA 同 24C02 共用，连接在 PH4 和 PH5 上，INT 脚连接在 PCF8574 的 P1 上，见图 31.2.1。关于 IIC 协议的介绍，请参考 IIC 实验，这个章节。

2，写寄存器

AP3216C 的写寄存器时序如图 32.1.2 所示：

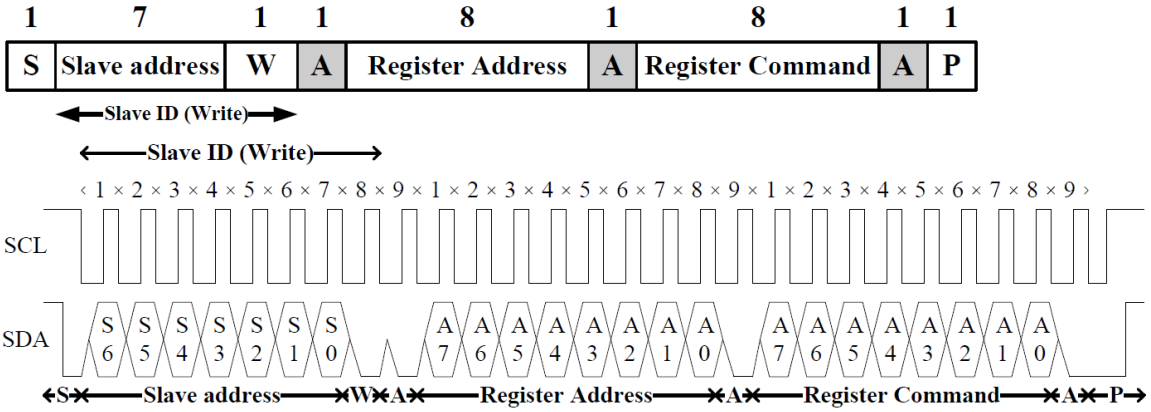


图 32.1.2 AP3216C 写寄存器时序

图中，先发送 AP3216C 的地址（7 位，0X1E，左移一位后为：0X3C），最低位 W=0 表示写数据，随后发送 8 位寄存器地址，最后发送 8 位寄存器值。其中：S，表示 IIC 起始信号；W，表示读/写标志位（W=0 表示写，W=1 表示读）；A，表示应答信号；P，表示 IIC 停止信号。

3，读寄存器

AP3216C 的读寄存器时序如图 32.1.3 所示：

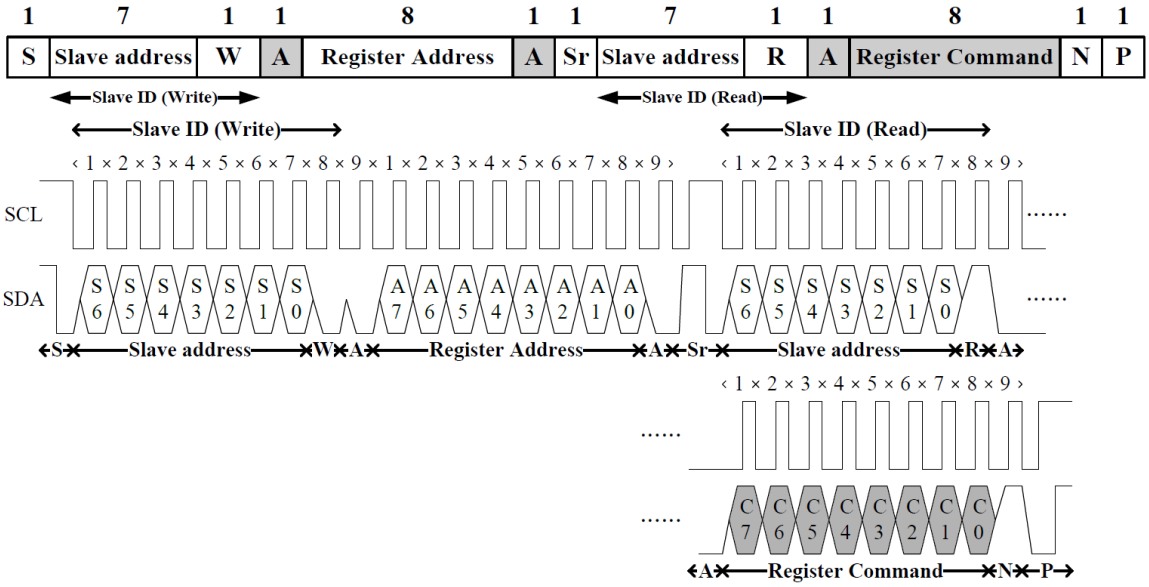


图 32.1.3 AP3216C 读寄存器时序

图中，同样是先发送 7 位地址+写操作，然后再发送寄存器地址，随后，重新发送起始信号（Sr），再次发送 7 位地址+读操作，然后读取寄存器值。其中：Sr，表示重新发送 IIC 起始信号；N，表示不对 AP3216C 进行应答；其他简写同上。

4，寄存器描述

AP3216C 有一系列寄存器，由这些寄存器来控制 AP3216C 的工作模式，以及中断配置和数据输出等。这里我们仅介绍我们在本章需要用到的一些寄存器，其他寄存器的描述和说明，请大家参考 AP3216C 的数据手册。

本章需要用到 AP3216C 的寄存器如表 32.1.2 所示：

地址	有效位	指令	说明
0X00	2:0	系统模式	000：掉电模式(默认) 001：ALS 功能激活 010：PS+IR 功能激活 011：ALS+PS+IR 功能激活 100：软复位 101：ALS 单次模式 110：PS+IR 单次模式 111：ALS+PS+IR 单次模式
0X0A	7	IR 低位数据	0:IR&PS 数据有效;1:无效
	1:0		IR 最低 2 位数据
0X0B	7:0	IR 高位数据	IR 高 8 位数据
0X0C	7:0	ALS 低位数据	ALS 低 8 位数据
0X0D	7:0	ALS 高位数据	ALS 高 8 位数据
0X0E	7	PS 低位数据	0,物体在远离;1,物体在靠近;
	6		0,IR 数据有效;1,IR 数据无效
	3:0		PS 最低 4 位数据

0X0F	7	PS 高位数据	0, 物体在远离;1, 物体在靠近;
	6		0, IR 数据有效;1, IR 数据无效
	5:0		PS 高 6 位数据

表 32.1.2 AP3216C 相关寄存器及其说明

上表中，0X00 是一个系统模式控制寄存器，主要在初始化的时候配置，初始化的时候，我们先设置其值为 100，实行一次软复位，随后设置其值为 011，开启 ALS+PS+IR 检测功能。

剩下的 6 个寄存器，为数据寄存器，输出 AP3216C 内部三个传感器所检测到的数据（ADC 值），描述如表所示，这里需要注意的是：读取间隔至少要大于 112.5ms，因为 AP3216C 内部完成一次 ALS+PS+IR 的数据转换，需要 112.5ms 的时间。

AP3216C 的简介，我们就介绍到这里，关于该芯片的详细说明，请大家参考其数据手册。

本章实验功能简介：开机的时候先检测 AP3216C 是否存在，如检测不到 AP3216C，则在 LCD 屏幕上面显示报错信息。如果检测到 AP3216C，则显示正常，并在主循环里面，循环读取 ALS+PS+IR 的传感器数据，并显示在 LCD 屏幕上面。同时，DS0 闪烁，提示程序正在运行。另外，本例程将 AP3216C 的读写操作函数加入 USMART 控制，我们也可以通过 USMART 对 AP3216C 进行控制。

32.2 硬件设计

本章需要用到的硬件资源有：

1）指示灯 DS0

2）串口（USMART 使用）

3）LCD 模块

4）AP3216C

前面 3 部分的资源，我们前面已经介绍了，请大家参考相关章节。这里介绍 AP3216C 与 STM32F767 和的连接，AP3216C 同 24C02 等共用一个 IIC 接口，SCL 和 SDA 分别连在 STM32F767 的 PH4 和 PH5 上的，另外 INT 脚连接在 PCF8574T 的 P1 口上，见图 30.2.1。连接关系如图 32.2.1 所示：

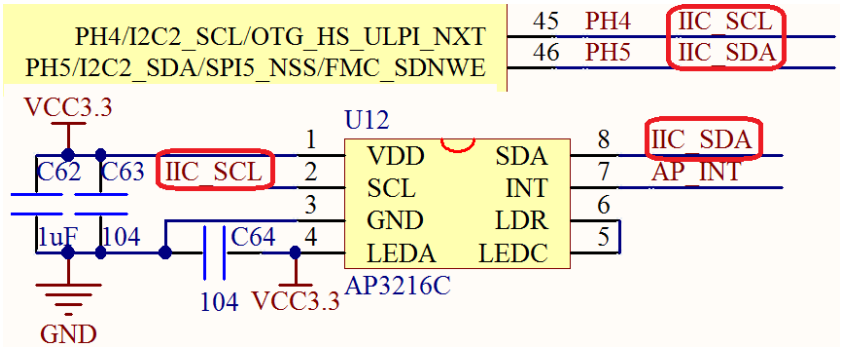


图 32.2.1 AP3216C 与 STM32F767 的连接图

这里需要说明一下：AP3216C 的 AP_INT 脚，是连接在 PCF8574T 的 P1 脚上的（见图 30.2.1），如果大家想要用 AP3216C 的中断输出功能，则必须初始化 PCF8574T，并监控 PCF8574T 的中断引脚，然后在发现有中断输入的时候，读取 PCF8574T，判断 P1 脚是否有低电平出现，从而检测 AP3216C 的中断。本章，我们并没有用到 AP3216C 的中断功能，所以，不需要配置 PCF8574T。

32.3 软件设计

打开本章实验工程可以看到，我们在 HARDWARE 分组下面添加了源文件 ap3216c.c 并且包含了其对应的头文件 ap3216c.h，AP3126C 相关驱动代码就放在这两个文件中。

打开 ap3216c.c 文件，代码如下：

```
//初始化 AP3216C
//返回值:0,初始化成功
//       1,初始化失败
u8 AP3216C_Init(void)
{
    u8 temp=0;
    IIC_Init();                                 //初始化 IIC
    AP3216C_WriteOneByte(0x00,0X04); //复位 AP3216C
    delay_ms(50);                               //AP33216C 复位至少 10ms
    AP3216C_WriteOneByte(0x00,0X03); //开启 ALS、PS+IR
    temp=AP3216C_ReadOneByte(0X00);     //读取刚刚写进去的 0X03
    if(temp==0X03)return 0;                     //AP3216C 正常
    else return 1;                              //AP3216C 失败
}

//读取 AP3216C 的数据
//读取原始数据，包括 ALS,PS 和 IR
//注意！如果同时打开 ALS,IR+PS 的话两次数据读取的时间间隔要大于 112.5ms
void AP3216C_ReadData(u16* ir,u16* ps,u16* als)
{
    u8 buf[6];
    u8 i;
    for(i=0;i<6;i++)
    {
        buf[i]=AP3216C_ReadOneByte(0X0A+i);   //循环读取所有传感器数据
    }
    if(buf[0]&0X80)*ir=0;                       //IR_OF 位为 1,则数据无效
    else *ir=((u16)buf[1]<<2)|(buf[0]&0X03);   //读取 IR 传感器的数据
    *als=((u16)buf[3]<<8)|buf[2];               //读取 ALS 传感器的数据
    if(buf[4]&0x40)*ps=0;                       //IR_OF 位为 1,则数据无效
    else *ps=((u16)(buf[5]&0X3F)<<4)|(buf[4]&0X0F); //读取 PS 传感器的数据
}

//IIC 写一个字节
//reg:寄存器地址
//data:要写入的数据
//返回值:0,正常
//       其他,错误代码
```

```
u8 AP3216C_WriteOneByte(u8 reg,u8 data)
{
    IIC_Start();
    IIC_Send_Byte(AP3216C_ADDR|0X00);//发送器件地址+写命令
    if(IIC_Wait_Ack())              //等待应答
    {
        IIC_Stop();
        return 1;
    }
    IIC_Send_Byte(reg);            //写寄存器地址
    IIC_Wait_Ack();                //等待应答
    IIC_Send_Byte(data);           //发送数据
    if(IIC_Wait_Ack())             //等待 ACK
    {
        IIC_Stop();
        return 1;
    }
    IIC_Stop();
    return 0;
}

//IIC 读一个字节
//reg:寄存器地址
//返回值:读到的数据
u8 AP3216C_ReadOneByte(u8 reg)
{
    u8 res;
    IIC_Start();
    IIC_Send_Byte(AP3216C_ADDR|0X00); //发送器件地址+写命令
    IIC_Wait_Ack();                 //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(AP3216C_ADDR|0X01); //发送器件地址+读命令
    IIC_Wait_Ack();                //等待应答
    res=IIC_Read_Byte(0);          //读数据,发送 nACK
    IIC_Stop();                    //产生一个停止条件
    return res;
}
```

该部分为 AP3216C 的驱动代码，其中的 IIC 相关函数，直接是用的第二十九章 myiic.c 里面提供的相关函数，这里不做介绍。

这里总共有 4 个函数：AP3216C_Init 函数用于初始化并检测 AP3216C，先设置 AP3216C 软复位，随后设置其工作在 ALS+PS+IR 模式，通过对系统模式寄存器的读写操作，来判断

AP3216C 是否正常（在位）；AP3216C_WriteOneByte 和 AP3216C_ReadOneByte 这两个函数实现 AP3216C 的寄存器写入和读取功能；AP3216C_ReadData 函数，则用于读取 ALS+PS+IR 传感器的数据，一般我们只需要调用该函数获取数据即可。

ap3216c.h 头文件代码非常简单，主要是函数声明以及宏定义标识符，这里大家只需要注意，宏定义标识符 AP3216C_ADDR 配置的是器件 AP3216C 的 IIC 地址。

最后我们看看主函数内容，程序如下：

```
int main(void)
{
    u16 ir,als,ps;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,70,200,16,16,"AP3216C TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2016/7/12");
    while(AP3216C_Init())       //检测不到 AP3216C
    {
        LCD_ShowString(30,130,200,16,16,"AP3216C Check Failed!");
        delay_ms(500);
        LCD_ShowString(30,130,200,16,16,"Please Check!          ");
        delay_ms(500);
        LED0_Toggle;                //DS0 闪烁
    }
    LCD_ShowString(30,130,200,16,16,"AP3216C Ready!");
    LCD_ShowString(30,160,200,16,16," IR:");
    LCD_ShowString(30,180,200,16,16," PS:");
    LCD_ShowString(30,200,200,16,16,"ALS:");
    POINT_COLOR=BLUE;               //设置字体为蓝色
    while(1)
    {
        AP3216C_ReadData(&ir,&ps,&als);  //读取数据
        LCD_ShowNum(30+32,160,ir,5,16);  //显示 IR 数据
        LCD_ShowNum(30+32,180,ps,5,16);  //显示 PS 数据
        LCD_ShowNum(30+32,200,als,5,16);//显示 ALS 数据
        LED0_Toggle;;                    //提示系统正在运行
```

```
            delay_ms(120);
        }
    }
```

该段代码，就是根据 31.1 节最后的功能简介来编写的，在初始化完成以后，我们 main 函数死循环里面，调用 AP3216C_ReadData 函数，读取 ALS+PS+IR 的数据，并显示在 LCD 上面。同时，DS0 闪烁，提示程序正在运行。这里我们延时 120ms 读取一次，确保 ALS+PS+IR 的转换全部完成，以保证数据正常。

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 阿波罗 STM32 开发板上，得到如图 32.4.1 所示的界面：

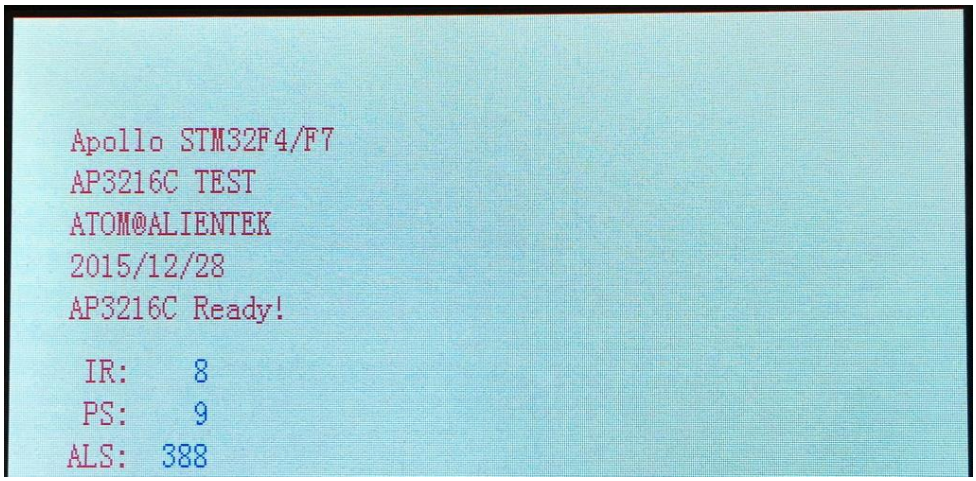


图 32.4.1 程序运行界面

此时，大家可以用手遮挡/靠近 AP3216C 传感器，可以看到三个传感器的数据变化，说明我们的代码是工作正常的。

第三十三章 QSPI 实验

本章我们将向大家介绍 STM32F767 的 QSPI 功能。在本章中，我们将使用 STM32F767 自带的 QSPI 来实现对外部 FLASH（W25Q256）的读写，并将结果显示在 LCD 模块上。本章分为如下几个部分：

33.1 QSPI 简介
33.2 硬件设计
33.3 软件设计
33.4 下载验证

33.1 QSPI 简介

本章我们将通过 STM32F767 的 QSPI 接口，来驱动 W25Q256 这颗 SPI FLASH 芯片，本节我们将介绍 QSPI 相关知识点，包括：1，QSPI 接口简介；2，W25Q256 简介；

33.1.1 QSPI 接口简介

QSPI 即 Quad SPI，是一种专用的通信接口，连接单、双或四（条数据线）SPI FLASH 存储器。STM32F7 具有 QSPI 接口，支持如下三种工作模式：

1，间接模式：使用 QSPI 寄存器执行全部操作
2，状态轮询模式：周期性读取外部 FLASH 状态寄存器，当标志位置 1 时会产生中断（如擦除或烧写完成，产生中断）
3，内存映射模式：外部 FLASH 映射到微控制器地址空间，从而系统将其视作内部存储器

STM32F7 的 QSPI 接口具有如下特点：

- 支持三种工作模式：间接模式、状态轮询模式和内存映射模式
- 支持双闪存模式，可以并行访问两个 FLASH，可同时发送/接收 8 位数据
- 支持 SDR（单倍率速率）和 DDR（双倍率速率）模式
- 针对间接模式和内存映射模式，完全可编程操作码
- 针对间接模式和内存映射模式，完全可编程帧格式
- 集成 FIFO，用于发送和接收
- 允许 8、16 和 32 位数据访问
- 具有适用于间接模式操作的 DMA 通道
- 在达到 FIFO 阈值、超时、操作完成以及发生访问错误时产生中断

STM32F7 的 QSPI 接口框图如图 33.1.1.1 所示：

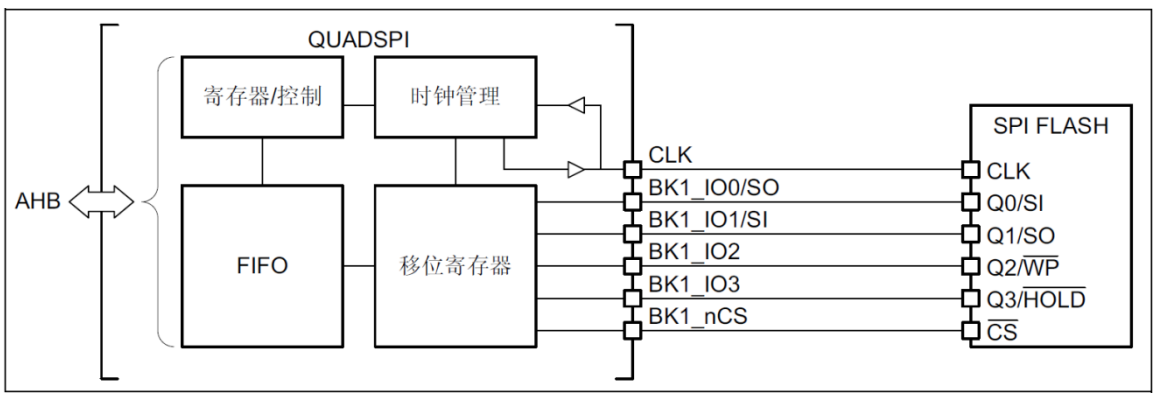


图 33.1.1.1 STM32F7 QSPI 框图

图 33.1.1.1 为 QSPI 单闪存模式的功能框图，由图可知，QSPI 接口通过 6 根线与 SPI 芯片连接，包括：4 根数据线（IO0~3）、1 根时钟线（CLK）和 1 根片选线（nCS）。我们知道普通的 SPI 通信一般只有一根数据线（MOSI），而 QSPI 则具有 4 根数据线，所以 QSPI 的速率至少是普通 SPI 的 4 倍，可以大大提高通信速率。接下来，我们给大家简单介绍一下 STM32F7 QSPI 接口的的几个重要知识点。

（1）QSPI 命令序列

QSPI 通过命令与 FLASH 通信，每条命令包括：指令、地址、交替字节、空指令和数据这五个阶段，任一阶段均可跳过，但至少要包含指令、地址、交替字节或数据阶段之一。

nCS 在每条指令开始前下降，在每条指令完成后再次上升。QSPI 四线模式下的读命令示例如图 33.1.1.2 所示：

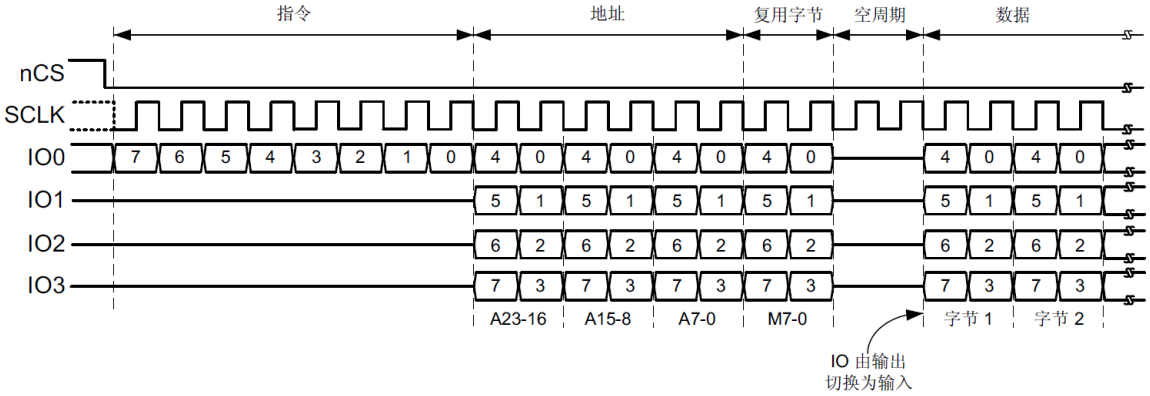


图 33.1.1.2 四线模式 QSPI 读命令示例

从上图可以看出一次 QSPI 传输的 5 个阶段，接下来我们分别介绍。

指令阶段

此阶段通过 QUADSPI_CCR[7:0]寄存器的 INSTRUCTION 字段指定一个 8 位指令发送到 FLASH。注意，指令阶段，一般是通过 IO0 单线发送，但是也可以配置为双线/四线发送指令，可以通过 QUADSPI_CCR[9:8]寄存器的 IMODE[1:0]这两个位进行配置，如 IMODE[1:0]=00，则表示无需发送指令。

地址阶段

此阶段可以发送 1~4 字节地址给 FLASH 芯片，指示要操作的地址。地址字节长度由 QUADSPI_CCR[13:12]寄存器的 ADSIZE[1:0]字段指定，0~3 表示 1~4 字节地址长度。在间接模式和轮询模式下，待发送的地址由 QUADSPI_AR 寄存器指定。地址阶段同样可以以单线/双线/四线模式发送，通过 QUADSPI_CCR[11:10]寄存器的 ADMODE[1:0]这两个位进行配置，如 ADMODE [1:0]=00，则表示无需发送地址。

交替字节（复用字节）阶段

此阶段可以发送 1~4 字节数据给 FLASH 芯片，一般用于控制操作模式。待发送的交替字节数由 QUADSPI_CCR[17:16]寄存器的 ABSIZE[1:0]位配置。待发送的数据由 QUADSPI_ABR 寄存器中指定。交替字节同样可以以单线/双线/四线模式发送，通过 QUADSPI_CCR[15:14]寄存器的 ABMODE[1:0]这两个位配置，ABMODE[1:0]=00，则跳过交替字节阶段。

空指令周期阶段

在空指令周期阶段，在给定的 1~31 个周期内不发送或接收任何数据，目的是当采用更高的时钟频率时，给 FLASH 芯片留出准备数据阶段的时间。这一阶段中给定的周期数由 QUADSPI_CCR[22:18]寄存器的 DCYC[4:0]位配置。若 DCYC 为零，则跳过空指令周期阶段，

命令序列直接进入下一个阶段。

数据阶段

此阶段可以从 FLASH 读取/写入任意字节数量的数据。在间接模式和自动轮询模式下，待发送/接收的字节数由 QUADSPI_DLR 寄存器指定。在间接写入模式下，发送到 FLASH 的数据必须写入 QUADSPI_DR 寄存器。在间接读取模式下，通过读取 QUADSPI_DR 寄存器获得从 FLASH 接收的数据。数据阶段同样可以以单线/双线/四线模式发送，通过 QUADSPI_CCR[25:24]寄存器的 DMODE [1:0]这两个位进行配置，如 DMODE [1:0]=00，则表示无数据。

以上就是 QSPI 数据传输的 5 个阶段，其中交替字节阶段我们一般用不到，可以省略（通过设置 ABMODE[1:0]=00）。另外，在本章我们是通过间接模式来访问 QSPI 的，接下来介绍间接模式。

（2）间接模式

在间接模式下，通过写入 QUADSPI 寄存器来触发命令，通过读写数据寄存器来传输数据。

当 FMODE=00 (QUADSPI_CCR[27:26])时，QUADSPI 处于间接写入模式，在数据阶段，将数据写入数据寄存器(QUADSPI_DR)，即可写入数据到 FLASH。

当 FMODE=01 时，QUADSPI 处于间接读取模式，在数据阶段，读取 QUADSPI_DR 寄存器，即可读取 FLASH 里面的数据。 。

读/写字节数由数据长度寄存器(QUADSPI_DLR)指定。当 QUADSPI_DLR=0xFFFFFFFF 时，则数据长度视为未定义，QUADSPI 将持续传输数据，直到到达 FLASH 结尾（FLASH 容量由 QUADSPI_DCR[20:16]寄存器的 FSIZE[4:0]位定义）。如果不传输任何数据，则 DMODE[1:0] (QUADSPI_CCR[25:24])应设置为 00。

当发送或接收的字节数（数据量）达到编程设定值时，如果 TCIE=1，则 TCF 置 1 并产生中断。在数据量不确定的情况下，将根据 FSIZE[4:0]定义的 FLASH 大小，在达到外部 SPI FLASH 的限制时，TCF 置 1。

在间接模式下有三种触发命令启动的方式，即：

1，当不需要发送地址（ADMODE[1:0]==00）和数据（DMODE[1:0]==00）时，对 INSTRUCTION[7:0]（QUADSPI_CCR[7:0]）执行写入操作。

2，当需要发送地址（ADMODE[1:0]!=00），但不需要发送数据（DMODE[1:0]==00），对 ADDRESS[31:0]（QUADSPI_AR）执行写入操作。

3，当需要发送地址（ADMODE[1:0]!=00）和数据（DMODE[1:0]!=00）时，对 DATA[31:0]（QUADSPI_DR）执行写入操作。

如果命令启动，BUSY 位（QUADSPI_SR 的第 5 位）将自动置 1。

（3）QSPI FLASH 配置

外部 SPI FLASH 芯片的相关参数，可以通过器件配置寄存器 (QUADSPI_DCR) 来进行设置。寄存器 QUADSPI_DCR[20:16]的 FSIZE[4:0]这 5 个位，用于指定外部存储器的大小，计算公式为：

$$Fcap=2^{[FSIZE+1]}$$

Fcap 表示 FLASH 的容量，单位为字节，在间接模式下，最高支持 4GB 容量的 FLASH 芯片。但是在内存映射模式下的可寻址空间限制为 256MB。

QSPI 连续执行两条命令时，它在两条命令之间将片选信号 (nCS) 置为高电平默认仅一个 CLK 周期。某些 FLASH 需要命令之间的时间更长，可以通过寄存器 QUADSPI_DCR[10:8]的 CSHT[2:0]（选高电平时间）这 3 个位设置高电平时长：0~7 表示 1~8 个时钟周期（最大为 8）。

时钟模式，用于指定在 nCS 为高电平时，CLK 的时钟极性。通过寄存器 QUADSPI_DCR[0]的 CKMODE 位指定：当 CKMODE=0 时，CLK 在 nCS 为高电平期间保持低电平，称之为模式

0；当 CKMODE=1 时，CLK 在 nCS 为高电平期间保持高电平，称之为模式 3；

接下来，我们介绍本章需要用到的一些寄存器。

首先是 QSPI 控制寄存器：QUADSPI_CR，该寄存器各位描述如图 33.1.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRESCALER								PMM	APMS	Res.	TOIE	SMIE	FTIE	TCIE	TEIE
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
Res.	Res.	Res.	FTHRES					FSEL	DFM	Res.	SSHIFT	TCEN	DMAEN	ABORT	EN
			rw	rw	rw	rw	rw	rw	rw		rw	rw	w1s	rw	w1s

图 33.1.1.3 QUADSPI_CR 寄存器各位描述

该寄存器我们只关心需要用到的一些位（下同），首先是 PRESCALER[7:0]，用于设置 AHB 时钟预分频器：0~255，表示 0~256 分频。我们使用的 W25Q256 最大支持 104Mhz 的时钟，这里我们设置 PRESCALER=2，即 3 分频，得到 QSPI 时钟为 72Mhz（216/3）。

FTHRES[4:0]，用于设置 FIFO 阈值，范围为 0~31，表示 FIFO 的阈值为 1~32 字节。

FSEL 位，用于选择 FLASH，我们的 W25Q256 连接在 STM32F7 的 QSPI BK1 上面，所以设置此位为 0 即可。

DFM 位，用于设置双闪存模式，我们用的是单闪存模式，所以设置此位为 0 即可。

SSHIFT 位，用于设置采样移位，默认情况下，QSPI 接口在 FLASH 驱动数据后过半个 CLK 周期开始采集数据。使用该位，可考虑外部信号延迟，推迟数据采集。我们一般设置此位为 1，移位半个周期采集，确保数据稳定。

ABORT 位，用于终止 QSPI 的当前传输，设置为 1 即可终止当前传输，在读写 FLASH 数据的时候，可能会用到。

EN 位，用于控制 QSPI 的使能，我们需要用到 QSPI 接口，所以必须设置此位为 1。

接下来，我们看 QSPI 器件配置寄存器：QUADSPI_DCR，该寄存器各位描述如图 33.1.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	FSIZE				
											rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	CSHT			Res.	Res.	Res.	Res.	Res.	Res.	Res.	CK-MODE
					rw	rw	rw								rw

图 33.1.1.4 QUADSPI_DCR 寄存器各位描述

该寄存器可以设置 FLASH 芯片的容量（FSIZE）、片选高电平时间（CSHT）和时钟模式（CKMODE）等，这些位的设置说明见前面的**（3）QSPI FLASH 配置**部分。

接下来，我们看 QSPI 通信配置寄存器：QUADSPI_CCR，该寄存器各位描述如图 33.1.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DDRM	DHHC	Res.	SIOO	FMODE[1:0]		DMODE		Res.	DCYC[4:0]					ABSIZE	
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
ABMODE		ADSIZE		ADMODE		IMODE		INSTRUCTION[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 33.1.1.5 QUADSPI_CCR 寄存器各位描述

DDRM 位，用于设置双倍率模式（DDR），我们没用到双倍率模式，所以设置此位为 0。

SIOO 位，用于设置指令是否只发送一次，我们需要每次都发送指令，所以设置此位为 0。

FMODE[1:0]，这两个位用于设置功能模式：00，间接写入模式；01，间接读取模式；10，自动轮询模式；11，内存映射模式；我们使用间接模式，所以此位根据需要设置为 00/01。

DMODE[1:0]，这两个位用于设置数据模式：00，无数据；01，单线传输数据；10，双线传输数据；11，四线传输数据；我们一般设置为 00/11。

DCYC[4:0]，这 5 个位用于设置空指令周期数，可以控制空指令阶段的持续时间，设置范围为：0~31。设置为 0，则表示没有空指令周期。

ABMODE[1:0]，这两个位用于设置交替字节模式，我们一般设置为 0，表示无交替字节。

ADMODE[1:0]，这两个位用于设置地址模式：00，无地址；01，单线传输地址；10，双线传输地址；11，四线传输地址；我们一般设置为 00/11。

IMODE[1:0]，这两个位用于设置指令模式：00，无指令；01，单线传输指令；10，双线传输指令；11，四线传输指令；我们一般设置为 00/11。

INSTRUCTION[7:0]，这 8 个位用于设置将要发送给 FLASH 的指令。

注意，以上这些位的配置，都必须在 QUADSPI_SR 寄存器的 BUSY 位为 0 时才可配置。

接下来，我们看 QSPI 数据长度寄存器：QUADSPI_DLR，该寄存器为一个 32 位寄存器，可以设置的数据长度范围为：0~0XFFFFFFFF，当 QUADSPI_DLR!=0XFFFFFFFF 时，表示传输的字节长度（+1）；当 QUADSPI_DLR==0XFFFFFFFF 时，表示不限传输长度，直到到达由 FSIZE 定义的 FLASH 结尾。

接下来，我们看 QSPI 地址寄存器：QUADSPI_AR，该寄存器为一个 32 位寄存器，用于指定发送到 FLASH 的地址。

接下来，我们看 QSPI 数据寄存器：QUADSPI_DR，该寄存器为一个 32 位寄存器，用于指定与外部 SPI FLASH 设备交换的数据。该寄存器支持字、半字和字节访问。

在间接写入模式下，写入该寄存器的数据在数据阶段发送到 FLASH，在此之前则存储于 FIFO，如果 FIFO 满了，则暂停写入，直到 FIFO 具有足够的空间接受要写入的数据才继续。

在间接模式下，读取该寄存器可获得（通过 FIFO）已从 FLASH 接收的数据。如果 FIFO 所含字节数比读取操作要求的字节数少，且 BUSY=1，则暂停读取，直到足够的数据出现或传输完成才继续。

接下来，我们看 QSPI 状态寄存器：QUADSPI_SR，该寄存器各位描述如图 33.1.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	FLEVEL[5:0]						Res.	Res.	BUSY	TOF	SMF	FTF	TCF	TEF
		r	r	r	r	r	r			r	r	r	r	r	r

图 33.1.1.6 QUADSPI_SR 寄存器各位描述

BUSY 位，指示操作是否忙。当该位为 1 时，表示 QSPI 正在执行操作。在操作完成或者 FIFO 为空的时候，该位自动清零。

FTF 位，表示 FIFO 是否到达阈值。在间接模式下，若达到 FIFO 阈值，或从 FLASH 读取完成后，FIFO 中留有数据时，该位置 1。只要阈值条件不再为“真”，该位就自动清零。

TCF 位，表示传输是否完成。在间接模式下，当传输的数据数量达到编程设定值，或在任何模式下传输中止时，该位置 1。向 QUADSPI_FCR 寄存器的 CTCF 位写 1，可以清零此位。

最后，我们看 QSPI 标志清零寄存器：QUADSPI_FCR，该寄存器各位描述如图 33.1.1.7 所

示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CTOF	CSMF	Res.	CTCF	CTEF
											w1o	w1o		w1o	w1o

图 33.1.1.7 QUADSPI_FCR 寄存器各位描述

该寄存器，我们一般只用到 CTCF 位，用于清除 QSPI 的传输完成标志。

至此，本章 QSPI 实验所需要用到的 QSPI 相关寄存器，就全部介绍完了，更详细的介绍，请参考《STM32F7 中文参考手册》 14.5 节。接下来，我们看间接模式下 QSPI 四个常见操作的简要步骤：初始化、发送命令、读数据和写数据。在 HAL 库中，QSPI 相关操作函数和定义在头文件 stm32f7xx_hal_qspi.c 和头文件 stm32f7xx_hal_qspi.h 中。

1，QSPI 初始化步骤

1）开启 QSPI 接口和相关 IO 的时钟，设置 IO 口的复用功能。

要使用 QSPI，肯定要先开启其时钟（由 AHB3ENR 控制），然后根据我们使用的 QSPI IO 口，开启对应 IO 口的时钟，并初始化相关 IO 口的复用功能（选择 QSPI 复用功能）。

QSPI 时钟使能方法为：

```
__HAL_RCC_QSPI_CLK_ENABLE();            //使能 QSPI 时钟
```

这里大家要注意，和其他外设处理方法一样，HAL 库提供了 QSPI 的初始化回调函数 HAL_QSPI_MspInit，一般用来编写与 MCU 相关的初始化操作。时钟使能和 IO 口初始化一般在回调函数中编写。

2）设置 QSPI 相关参数。

此部分需要设置两个寄存器：QUADSPI_CR 和 QUADSPI_DCR，控制 QSPI 的时钟、片选参数、FLASH 容量和时钟模式等参数，设定 SPI FLASH 的工作条件。最后，使能 QSPI，完成对 QSPI 的初始化。HAL 库中设置 QSPI 相关参数函数为 HAL_QSPI_Init，该函数声明为：

```
HAL_StatusTypeDef HAL_QSPI_Init(QSPI_HandleTypeDef *hqspi);
```

该函数只有一个 QSPI_HandleTypeDef 结构体指针类型入口参数 hqspi，该结构体定义为：

```
typedef struct
{
  QUADSPI_TypeDef              *Instance;
  QSPI_InitTypeDef             Init;
  uint8_t                      *pTxBuffPtr;
  __IO uint16_t                TxXferSize;
  __IO uint16_t                TxXferCount;
  uint8_t                      *pRxBuffPtr;
  __IO uint16_t                RxXferSize;
  __IO uint16_t                RxXferCount;
  DMA_HandleTypeDef            *hdma;
  __IO HAL_LockTypeDef         Lock;
  __IO HAL_QSPI_StateTypeDef   State;
  __IO uint32_t                ErrorCode;
  uint32_t                     Timeout;
```

```
}QSPI_HandleTypeDef;
```

该结构体成员变量较多，接下来我们一一来解释。成员变量 Instance 用来设置 QSPI 寄存器基地址，这在 HAL 库中已经定义，直接设置为标识符 QUADSPI 即可。成员变量 Init 是 QSPI_InitTypeDef 结构体类型，用来设置 QSPI 参数，这也是 QSPI 初始化最主要设置的变量，我们在讲完其他成员变量后着重讲解 Init 成员变量的设置方法。成员变量 pTxBuffPtr，TxXferSize 和 TxXferCount 分别用来设置 QSPI 发送缓冲指针，发送数据量以及发送剩余数据量。成员变量*pRxBuffPtr，RxXferSize 和 RxXferCount 分别用来设置 QSPI 接收缓冲指针，接收数据量和接收剩余数据量。Hdma 是 QSPI 的 DMA 处理相关，其他成员变量则是 HAL 库处理过程中的一些状态和错误标志。

接下来我们着重讲解 Init 成员变量，该成员变量是 QSPI_InitTypeDef 结构体类型，QSPI_InitTypeDef 结构体定义如下：

```
typedef struct
{
  uint32_t ClockPrescaler;      //时钟分频系数
  uint32_t FifoThreshold;       //设置 FIFO 阈值
  uint32_t SampleShifting;      //设置采样移位
  uint32_t FlashSize;           //设置 FLASH 大小
  uint32_t ChipSelectHighTime;  //设置片选高电平时间
  uint32_t ClockMode;           //设置时钟模式
  uint32_t FlashID;             //闪存 ID，第一片还是第二片
  uint32_t DualFlash;           //双闪存模式设置
}QSPI_InitTypeDef;
```

该结构体各个成员变量含义我们在上面已经注释，实际上这些成员变量是用来配置 QUADSPI_CR 寄存器和 QUADSPI_DCR 寄存器相应位，这在我们前面已经讲解，大家可以结合这两个寄存器的位定义和结构体定义来理解。

对于 HAL_QSPI_Init 函数使用范例请参考后面 33.3 软件设置部分程序源码。

2，QSPI 发送命令步骤

1）等待 QSPI 空闲

在 QSPI 发送命令前，必须先等待 QSPI 空闲，通过判断 QUADSPI_SR 寄存器的 BUSY 位为 0，来确定。

2）设置命令参数。

此部分主要是通过通信配置寄存器（QUADSPI_CCR）设置，将 QSPI 配置为：每次都发送指令、间接写模式，根据具体需要设置：指令、地址、空周期和数据等的传输位宽等信息。如果需要发送地址，则配置地址寄存器（QUADSPI_AR）。

在配置完成以后，即可启动发送。如果不需要传输数据，则需要等待命令发送完成（等待 QUADSPI_SR 寄存器的 TCF 位为 1）。

在 HAL 库中上述两个步骤是通过函数 HAL_QSPI_Command 来实现，该函数声明为：

```
HAL_StatusTypeDef HAL_QSPI_Command(QSPI_HandleTypeDef *hqspi,
                                   QSPI_CommandTypeDef *cmd, uint32_t Timeout);
```

该函数有三个入口参数，第一个入口参数 hqspi 和第三个入口参数 Timeout 都比较好理解，这里我们着重讲解第二个入口参数 cmd，该参数是 QSPI_CommandTypeDef 结构体指针类型，该结构体定义如下：

```
typedef struct
```

```
{
    uint32_t Instruction;              //指令
    uint32_t Address;                  //地址
    uint32_t AlternateBytes;           //交替字节
    uint32_t AddressSize;              //地址长度
    uint32_t AlternateBytesSize;       //交替字节大小
    uint32_t DummyCycles;              //控指令周期数
    uint32_t InstructionMode;          //指令模式
    uint32_t AddressMode;              //地址模式
    uint32_t AlternateByteMode;        //交替字节模式
    uint32_t DataMode;                 //数据模式
    uint32_t NbData;                   //数据长度
    uint32_t DdrMode;
    uint32_t DdrHoldHalfCycle;
    uint32_t SIOOMode;
}QSPI_CommandTypeDef;
```

该结构体关键成员变量含义在上面我们已经注释。这些成员变量主要用来配置对应的 CCR 寄存器，如果有不理解的地方请对照前面讲解的 CCR 寄存器定义来理解。

3，QSPI 读数据步骤

1）设置数据传输长度

通过设置数据长度寄存器（QUADSPI_DLR），配置需要传输的字节数。

2）设置 QSPI 工作模式并设置地址

因为要读取数据，所以，设置 QUADSPI_CCR 寄存器的 FMODE[1:0]位为 01，工作在间接读取模式。然后，通过地址寄存器（QUADSPI_AR），设置我们将要读取的数据的首地址。

3）读取数据

在发送完地址以后，就可以读取数据了，不过要等待数据准备好，通过判断 QUADSPI_SR 寄存器的 FTF 和 TCF 位，当这两个位任意一个位为 1 的时候，我们就可以读取 QUADSPI_DR 寄存器来获取从 FLASH 读到的数据。

最后，在所有数据接收完成以后，终止传输（ABORT），清除传输完成标志位（TCF）。

HAL 库中，读取数据是通过函数 HAL_QSPI_Receive 来实现的，该函数声明为：

```
HAL_StatusTypeDef HAL_QSPI_Receive(QSPI_HandleTypeDef *hqspi,
                                   uint8_t *pData, uint32_t Timeout);
```

在调用该函数读取数据之前，我们会先调用上个步骤讲解的函数 HAL_QSPI_Command 来指定读取数据的存放空间。

4，QSPI 写数据步骤

1）设置数据传输长度

通过设置数据长度寄存器（QUADSPI_DLR），配置需要传输的字节数。

2）设置 QSPI 工作模式并设置地址

因为要读取数据，所以，设置 QUADSPI_CCR 寄存器的 FMODE[1:0]位为 00，工作在间接写入模式。然后，通过地址寄存器（QUADSPI_AR），设置我们将要写入的数据的首地址。

3）写入数据

在发送完地址以后，就可以写入数据了，不过要等待 FIFO 不满，通过判断 QUADSPI_SR 寄存器的 FTF 位，当这个位为 1 的时候，表示 FIFO 可以写入数据，此时往 QUADSPI_DR 写

入需要发送的数据，就可以实现写入数据到 FLASH。

最后，在所有数据写入完成以后，终止传输（ABORT），清除传输完成标志位（TCF）。

在 HAL 库中，QSPI 发送数据是通过函数 HAL_QSPI_Transmit 来实现的，该函数声明为:

```
HAL_StatusTypeDef HAL_QSPI_Transmit(QSPI_HandleTypeDef *hqspi,
                                    uint8_t *pData, uint32_t Timeout);
```

同理，在调用该函数发送数据之前，我们会先调用上个步骤讲解的函数 HAL_QSPI_Command 来指定要写入数据的存储地址信息。

STM32F7 的 QSPI 接口简介就给大家介绍到这里，接下来，我们介绍 W25Q256。

33.1.2 W25Q256 简介

W25Q256 是华邦公司生产的一颗容量为 32M 字节的串行 FLASH 芯片，它将 32M 的容量分为 512 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。W25Q256 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 W25Q256 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。

W25Q256 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，W25Q256 支持标准的 SPI，还支持双输出/四输出 SPI 和 QPI（QPI 即 QSPI，下同），最高时钟频率可达 104Mhz(双输出时相当于 208Mhz，四输出时相当于 416M)，本章我们将利用 STM32F7 的 QSPI 接口来实现对 W25Q256 的驱动。

接下来，我们介绍一下本章驱动 W25Q256 需要用到的一些指令，如表 33.1.2.1 所示：

输入/输出数据		字节 1	字节 2	字节 3	字节 4	字节 5	字节 6	字节 7
时钟数	SPI 模式	0-7	8-15	16-23	24-31	32-39	40-47	48-55
	QPI 模式	0, 1	2, 3	4, 5	6, 7	8, 9	10, 11	12, 13
W25X_ReadStatusReg1		0X05	S7-S0					
W25X_ReadStatusReg2		0X35	S15-S8					
W25X_ReadStatusReg3		0X15	S23-S16					
W25X_WriteStatusReg1		0X01	S7-S0					
W25X_WriteStatusReg2		0X31	S15-S8					
W25X_WriteStatusReg3		0X11	S23-S16					
W25X_ManufactDeviceID		0X90	Dummy	Dummy	0X00	MF7-MF0	ID7-ID0	
W25X_EnterQPIMode		0X38						
W25X_Enable4ByteAddr		0XB7						
W25X_SetReadParam		0XC0	P7-P0					
W25X_WriteEnable		0X06						
W25X_FastReadData		0X0B	A31-A24	A23-A16	A15-A8	A7-A0	Dummy[1]	D7-D0[2]
W25X_PageProgram		0X02	A31-A24	A23-A16	A15-A8	A7-A0	D7-D0[2]	D7-D0[2]
W25X_SectorErase		0X20	A31-A24	A23-A16	A15-A8	A7-A0		
W25X_ChipErase		0XC7						

1，在 QPI 模式下 dummy 时钟的个数，由读参数控制位 P[5:4]位控制。

2，传输的数据量，只要不停的给时钟就可以持续传输，对 W25X_PageProgram 指令，则单次传输最多不超过 256 字节，否则将覆盖之前写入的数据。

表 33.1.2.1 W25Q256 指令

表 33.1.2.1 列出了本章我们驱动 W25Q256 所需要用到的所有指令和对应的参数，注意 SPI 模式和 QPI 模式下时钟数的区别，可知 QPI 模式比 SPI 模式所需要的时钟数少的多，所以速度也快得多。接下来我们简单介绍一下这些指令。

首先，前面 6 个指令，是用来读取/写入状态寄存器 1~3 的。在读取的时候，读取 S23~S0 的数据，在写入的时候，写入 S23~S0。而 S23~S0 则由三部分组成：S23~S16，S15~S8，S7~S0 即状态寄存器 3、2、1，如表 33.1.2.2 所示：

状态寄存器 3	**S23**	**S22**	**S21**	**S20**	**S19**	**S18**	**S17**	**S16**
位说明	HOLD /RST	DRV1	DRV0			WPS	ADP	ADS
状态寄存器 2	**S15**	**S14**	**S13**	**S12**	**S11**	**S10**	**S9**	**S8**
位说明	SUS	CMP	LB3	LB2	LB1		QE	SRP1
状态寄存器 1	**S7**	**S6**	**S5**	**S4**	**S3**	**S2**	**S1**	**S0**
位说明	SRP0	TB	BP3	BP2	BP1	BP0		BUSY

表 33.1.2.2 W25Q256 状态寄存器

这三个状态寄存器，我们只关心我们需要用到的一些位：ADS、QE 和 BUSY 位。其他位的说明，请看 W25Q256 的数据手册。

ADS 位，表示 W25Q256 当前的地址模式，是一个只读位，当 ADS=0 的时候，表示当前是 3 字节地址模式，当 ADS=1 的时候，表示当前是 4 字节地址模式，我们需要使用 4 字节地址模式，所以在读取到该位为 0 的时候，必须通过 W25X_Enable4ByteAddr 指令，设置为 4 字节地址模式。

QE 位，用于使能 4 线模式（Quad），此位可读可写，并且是可以保存的（掉电后可以继续保持上一次的值）。在本章，我们需要用到 4 线模式，所以在读到该位为 0 的时候，必须通过 W25X_WriteStatusReg2 指令设置此位为 1，表示使能 4 线模式。

BUSY 位，用于表示擦除/编程操作是否正在进行，当擦除/编程操作正在进行时，此位为 1，此时 W25Q256 不接受任何指令，当擦除/编程操作完成时，此位为 0。此位为只读位，我们在执行某些操作的时候，必须等待此位为 0。

W25X_ManufactDeviceID 指令，用于读取 W25Q256 的 ID，可以用于判断 W25Q256 是否正常。对于 W25Q256 来说：MF[7:0]=0XEF，ID[7:0]=0X18。

W25X_EnterQPIMode 指令，用于设置 W25Q256 进入 QPI 模式。上电时，W25Q256 默认是 SPI 模式，我们需要通过该指令设置其进入 QPI 模式。注意：在发送该指令之前，必须先设置状态寄存器 2 的 QE 位为 1！！

W25X_Enable4ByteAddr 指令，用于设置 W25Q256 进入 4 字节地址模式。当读取到 ADS 位为 0 的时候，我们必须通过此指令将 W25Q256 设置为 4 字节地址模式，否则将只能访问 16MB 的地址空间。

W25X_SetReadParam 指令，可以用于设置读参数控制位 P[5:4]，这两个位的描述如表 33.1.2.3 所示：

P5 – P4	DUMMY CLOCKS	MAXIMUM READ FREQ.	MAXIMUM READ FREQ. (A[1:0]=0,0)	MAXIMUM READ FREQ. (A[1:0]=0,0 VCC=3.0V~3.6V)
0 0	2	33MHz	33MHz	40MHz
0 1	4	55MHz	80MHz	80MHz
1 0	6	80MHz	80MHz	104MHz
1 1	8	80MHz	80MHz	104MHz

表 33.1.2.3 W25Q256 读参数控制位

为了让 W25Q256 可以工作在最大频率下，我们这里设置 P[5:4]=11，即可工作在 104Mhz 的时钟频率下。此时，读取数据时的 dummy 时钟个数为 8 个（参见 W25X_FastReadData 指令）。

W25X_WriteEnable 指令，用于设置 W25Q256 写使能。在执行擦除、编程、写状态寄存器等操作之前，都必须通过该指令，设置 W25Q256 写使能，否则无法写入。

W25X_FastReadData 指令，用于读取 FLASH 数据，在发送完该指令以后，就可以读取 W25Q256 的数据了。该指令发送完成后，我们可以持续读取 FLASH 里面的数据，只要不停的给时钟，就可以不停的读取数据。

W25X_PageProgram 指令，用于编程 FLASH（写入数据到 FLASH），该指令发送完成后，最多可以一次写入 256 字节到 W25Q256，超过 256 字节则需要多次发送该指令。

W25X_SectorErase 指令，用于擦除一个扇区（4KB）的数据。因为 FLASH 具有只可以写 0，不可以写 1 的特性，所以在写入数据的时候，一般需要先擦除（归 1），再写。W25Q256 的最小擦除单位为一个扇区（4KB）。该指令在写入数据的时候，经常要有用。

W25X_ChipErase 指令，用于全片擦除 W25Q256。

最后，我们看看 W25Q256 的初始化流程（QPI 模式）：

1）使能 QPI 模式。

因为我们是通过 QSPI 访问 W25Q256 的，所以先设置 W25Q256 工作在 QPI 模式下。通过 W25X_EnterQPIMode 指令控制。注意：在该指令发送之前，必须先使能 W25Q256 的 QE 位。

2）设置 4 字节地址模式。

W25Q256 上电后，一般默认是 3 字节地址模式，我们需要通 W25X_Enable4ByteAddr 指令，设置其为四字节地址模式，否则只能访问 16MB 的地址空间。

3）设置读参数。

这一步，我们通过 W25X_SetReadParam 指令，将 P[5:4]设置为 11，以支持最高速度访问 W25Q256（8 个 dummy，104M 时钟频率）。至此，W25Q256 的初始化流程就完成了，接下来便可以通过 QSPI 读写数据了。

更多的 W25Q256 的介绍，请参考 W25Q256 的数据手册。

33.2 硬件设计

本章实验功能简介：开机的时候先检测 W25Q256 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 W25Q256 的操作，另外一个按键（KEY0）用来执行读出操作，在 LCD 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） LCD 模块
4） QSPI
5） W25Q256

这里只介绍 W25Q256 与 STM32F767 的连接，板上的 W25Q256 是连接在 STM32F767 的 QSPI BK1 上的，连接关系如图 33.2.1 所示：

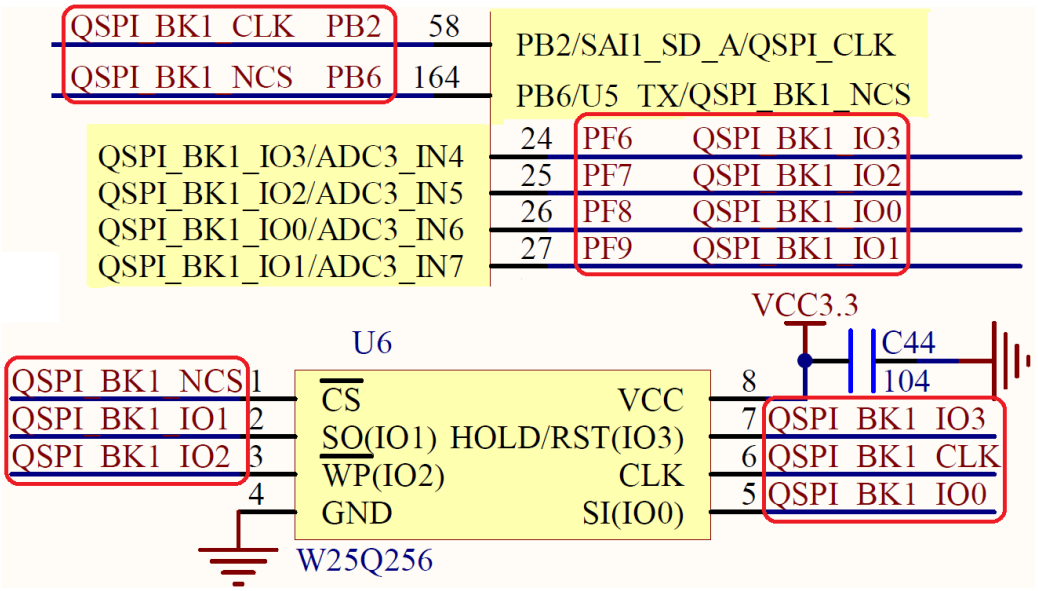


图 33.2.1 STM32F767 与 W25Q256 连接电路图

33.3 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 qspi.c,w25qxx.c 文件以及头文件 qspi.h 和 w25qxx.h，同时引入了库函数文件 stm32f7xx_hal_spi.c 文件以及头文件 stm32f7xx_hal_spi.h。

打开 qspi.c 文件，看到如下代码：

```
QSPI_HandleTypeDef QSPI_Handler;       //QSPI 句柄

//QSPI 初始化
u8 QSPI_Init(void)
{
    QSPI_Handler.Instance=QUADSPI;   //QSPI
    QSPI_Handler.Init.ClockPrescaler=2;   //QPSI 分频比，W25Q256 最大频率为 104M，
                                          //所以此处应该为 2，QSPI 频率就为 216/(2+1)=72MHZ
    QSPI_Handler.Init.FifoThreshold=4;       //FIFO 阈值为 4 个字节
    QSPI_Handler.Init.SampleShifting=QSPI_SAMPLE_SHIFTING_HALFCYCLE;
                                          //采样移位半个周期(DDR 模式下,必须设置为 0)
    QSPI_Handler.Init.FlashSize=POSITION_VAL(0X2000000)-1;
                                         //SPI FLASH 大小，W25Q256 大小为 32M 字节
    QSPI_Handler.Init.ChipSelectHighTime=QSPI_CS_HIGH_TIME_4_CYCLE;
            //片选高电平时间为 4 个时钟(13.8*4=55.2ns),即手册里面的 tSHSL 参数
    QSPI_Handler.Init.ClockMode=QSPI_CLOCK_MODE_0;        //模式 0
    QSPI_Handler.Init.FlashID=QSPI_FLASH_ID_1;                //第一片 flash
    QSPI_Handler.Init.DualFlash=QSPI_DUALFLASH_DISABLE; //禁止双闪存模式
    if(HAL_QSPI_Init(&QSPI_Handler)==HAL_OK) return 0;        //QSPI 初始化成功
    else return 1;
```

```
}

//QSPI 底层驱动,引脚配置，时钟使能
//此函数会被 HAL_QSPI_Init()调用
//hqspi:QSPI 句柄
void HAL_QSPI_MspInit(QSPI_HandleTypeDef *hqspi)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_QSPI_CLK_ENABLE();            //使能 QSPI 时钟
    __HAL_RCC_GPIOB_CLK_ENABLE();           //使能 GPIOB 时钟
    __HAL_RCC_GPIOF_CLK_ENABLE();           //使能 GPIOF 时钟

    //初始化 PB6  片选信号
    GPIO_Initure.Pin=GPIO_PIN_6;
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //复用
    GPIO_Initure.Pull=GPIO_PULLUP;
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    GPIO_Initure.Alternate=GPIO_AF10_QUADSPI;   //复用为 QSPI
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);

    //PF8,9
    GPIO_Initure.Pin=GPIO_PIN_8|GPIO_PIN_9;
    GPIO_Initure.Pull=GPIO_NOPULL;
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;          //高速
    HAL_GPIO_Init(GPIOF,&GPIO_Initure);

    //PB2
    GPIO_Initure.Pin=GPIO_PIN_2;
    GPIO_Initure.Alternate=GPIO_AF9_QUADSPI;    //复用为 QSPI
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);

    //PF6,7
    GPIO_Initure.Pin=GPIO_PIN_6|GPIO_PIN_7;
    HAL_GPIO_Init(GPIOF,&GPIO_Initure);
}

//QSPI 发送命令
//instruction:要发送的指令
//address:发送到的目的地址
//dummyCycles:空指令周期数
//instructionMode:指令模式;
//addressMode:地址模式;
```

```
//addressSize:地址长度;
//dataMode:数据模式;

void QSPI_Send_CMD(u32 instruction,u32 address,u32 dummyCycles,u32 instructionMode,u32 addressMode,u32 addressSize,u32 dataMode)
{
    QSPI_CommandTypeDef Cmdhandler;

    Cmdhandler.Instruction=instruction;                 //指令
    Cmdhandler.Address=address;                         //地址
    Cmdhandler.DummyCycles=dummyCycles;                 //设置空指令周期数
    Cmdhandler.InstructionMode=instructionMode;         //指令模式
    Cmdhandler.AddressMode=addressMode;                 //地址模式
    Cmdhandler.AddressSize=addressSize;                 //地址长度
    Cmdhandler.DataMode=dataMode;                       //数据模式
    Cmdhandler.SIOOMode=QSPI_SIOO_INST_EVERY_CMD;    //每次都发送指令
    Cmdhandler.AlternateByteMode=QSPI_ALTERNATE_BYTES_NONE; //无交替字节
    Cmdhandler.DdrMode=QSPI_DDR_MODE_DISABLE;    //关闭 DDR 模式
    Cmdhandler.DdrHoldHalfCycle=QSPI_DDR_HHC_ANALOG_DELAY;

    HAL_QSPI_Command(&QSPI_Handler,&Cmdhandler,5000);
}

//QSPI 接收指定长度的数据
//buf:接收数据缓冲区首地址
//datalen:要传输的数据长度
//返回值:0,正常
//       其他,错误代码
u8 QSPI_Receive(u8* buf,u32 datalen)
{
    QSPI_Handler.Instance->DLR=datalen-1;                                   //配置数据长度
    if(HAL_QSPI_Receive(&QSPI_Handler,buf,5000)==HAL_OK) return 0;    //接收数据
    else return 1;
}

//QSPI 发送指定长度的数据
//buf:发送数据缓冲区首地址
//datalen:要传输的数据长度
//返回值:0,正常
//       其他,错误代码
u8 QSPI_Transmit(u8* buf,u32 datalen)
{
    QSPI_Handler.Instance->DLR=datalen-1;                                   //配置数据长度
```

```
    if(HAL_QSPI_Transmit(&QSPI_Handler,buf,5000)==HAL_OK) return 0;  //发送数据
    else return 1;
}
```

此部分代码实现了 QSPI 的初始化、发送命令、读数据和写数据等四个关键函数，这几个函数是按我们在 33.1.1 节末尾所介绍的步骤来实现的，详见之前的介绍。

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_Qspi_Enable 函数，该函数代码如下：

```
//W25QXX 进入 QSPI 模式
void W25QXX_Qspi_Enable(void)
{
    u8 stareg2;
    stareg2=W25QXX_ReadSR(2);   //先读出状态寄存器 2 的原始值
    if((stareg2&0X02)==0)       //QE 位未使能
    {
        W25QXX_Write_Enable();      //写使能
        stareg2|=1<<1;              //使能 QE 位
        W25QXX_Write_SR(2,stareg2); //写状态寄存器 2
    }
    QSPI_Send_CMD(W25X_EnterQPIMode,0,0,QSPI_INSTRUCTION_1_LINE,
                  QSPI_ADDRESS_NONE,QSPI_ADDRESS_8_BITS,QSPI_DATA_NONE);
                    //写 command 指令,地址为 0,无数据_8 位地址_无地址_单线传输指令,
                    //无空周期,0 个字节数据
    W25QXX_QPI_MODE=1;                  //标记 QSPI 模式
}
```

该函数用于设置 W25Q256 进入 QPI 模式，在 W25Q256 初始化的时候，被调用。该函数末尾有：W25QXX_QPI_MODE=1，表示 W25Q256 当前为 QPI 模式。W25QXX_QPI_MODE 是我们在 w25qxx.c 里面定义的一个全局变量，用于表示 W25Q256 的当前模式（0，SPI 模式；1，QPI 模式）。

然后，我们介绍 W25QXX_Init 函数，该函数代码如下：

```
//初始化 SPI FLASH 的 IO 口
void W25QXX_Init(void)
{
    u8 temp;
    QSPI_Init();                        //初始化 QSPI
    W25QXX_Qspi_Enable();               //使能 QSPI 模式
    W25QXX_TYPE=W25QXX_ReadID();        //读取 FLASH ID.
    if(W25QXX_TYPE==W25Q256)            //SPI FLASH 为 W25Q256
    {
        temp=W25QXX_ReadSR(3);//读取状态寄存器 3，判断地址模式
        if((temp&0X01)==0)          //如果不是 4 字节地址模式,则进入 4 字节地址模式
        {
            W25QXX_Write_Enable();  //写使能
```

```
    QSPI_Send_CMD(W25X_Enable4ByteAddr,0,0,QSPI_INSTRUCTION_4_LINES,
                  QSPI_ADDRESS_NONE,QSPI_ADDRESS_8_BITS,QSPI_DATA_NONE);
                  //QPI,使能 4 字节地址指令,地址为 0,无数据_8 位地址_无地址_4 线传输指令,
                  //无空周期,0 个字节数据
            }
            W25QXX_Write_Enable();          //写使能
            QSPI_Send_CMD(W25X_SetReadParam,0,0,QSPI_INSTRUCTION_4_LINES,
              QSPI_ADDRESS_NONE,QSPI_ADDRESS_8_BITS,QSPI_DATA_4_LINES);
                  //QPI,设置读参数指令,地址为 0,4 线传数据_8 位地址_无地址_4 线传输指令,
                  //无空周期,1 个字节数据
            temp=3<<4;                      //设置 P4&P5=11,8 个 dummy clocks,104M
            QSPI_Transmit(&temp,1);         //发送 1 个字节
        }
    }
```

该函数用于初始化 W25Q256，首先调用 QSPI_Init 函数，初始化 STM32F7 的 QSPI 接口，然后依据我们在 33.1.2 节末尾介绍的初始化流程，初始化 W25Q256。在初始化完成以后，我们便可以通过 QSPI 接口读写 W25Q256 的数据了。

接下来介绍 W25QXX_Read 函数，该函数代码如下：

```
//读取 SPI FLASH,仅支持 QPI 模式
//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(最大 32bit)
//NumByteToRead:要读取的字节数(最大 65535)
void W25QXX_Read(u8* pBuffer,u32 ReadAddr,u16 NumByteToRead)
{
    QSPI_Send_CMD(W25X_FastReadData,ReadAddr,8,QSPI_INSTRUCTION_4_LINES,
                  QSPI_ADDRESS_4_LINES,QSPI_ADDRESS_32_BITS,QSPI_DATA_4_LINES);
    //QPI,快速读数据,地址为 ReadAddr,4 线传输数据_32 位地址_4 线传输地址_4 线传输
    //指令,8 空周期,NumByteToRead 个数据
    QSPI_Receive(pBuffer,NumByteToRead);
}
```

该函数用于用于从 W25Q256 的指定地址读出指定长度的数据，由于 W25Q256 支持以任意地址（但是不能超过 W25Q256 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，通过 QSPI_Send_CMD 函数，发送 W25X_FastReadData 指令，并发送读数据首地址（ReadAddr），然后通过 QSPI_Receive 函数循环读取数据，存放在 pBuffer 里面。

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_Read 的作用类似，不过是用来写数据到 W25Q256 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(最大 32bit)
//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++) W25QXX_BUF[i+secoff]=pBuffer[i];      //复制
            W25QXX_Write_NoCheck(W25QXX_BUF,secpos*4096,4096);//写入整个扇区
        }else W25QXX_Write_NoCheck(pBuffer,WriteAddr,secremain);//写擦除,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 W25Q256 的任意地址开始写入任意长度（必须不超过 W25Q256 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。最后我们看看 main.c 源文件内容：

```
//要写入到 W25Q16 的字符串数组
const u8 TEXT_Buffer[]={"Apollo STM32F7 QSPI 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(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    W25QXX_Init();                  //初始化 W25QXX
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");

    LCD_ShowString(30,70,200,16,16,"QSPI TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2016/7/12");
    LCD_ShowString(30,130,200,16,16,"KEY1:Write  KEY0:Read");  //显示提示信息

    while(W25QXX_ReadID()!=W25Q256)  //检测不到 W25Q256
    {
        LCD_ShowString(30,150,200,16,16,"QSPI Check Failed!");
        delay_ms(500);
        LCD_ShowString(30,150,200,16,16,"Please Check!      ");
        delay_ms(500);
        LED0_Toggle;      //DS0 闪烁
    }
    LCD_ShowString(30,150,200,16,16,"QSPI 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 QSPI....");
            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,"QSPI 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 QSPI.... ");
            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_Toggle;//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_ReadSR、W25QXX_Write_SR、W25QXX_ReadID 和 W25QXX_Erase_Chip 等函数加入 USMART 控制，这样，我们就可以通过串口调试助手，操作 W25Q256，方便大家测试。

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 阿波罗 STM32 开发板上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 33.4.1 所示：

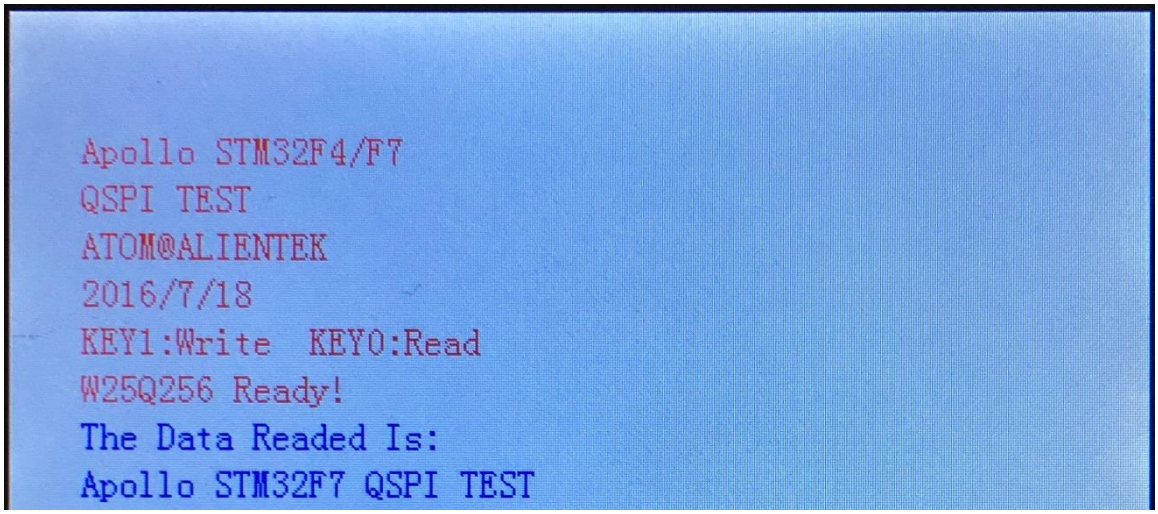


图 33.4.1 SPI 实验程序运行效果图

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 W25Q256 是否存在，如果不存在则会在 LCD 模块上显示错误信息，同时 DS0 慢闪。

第三十四章 485 实验

本章我们将向大家介绍如何使用 STM32F767 的串口实现 485 通信（半双工）。在本章中，我们将使用 STM32F767 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 LCD 模块上。本章分为如下几个部分：

34.1 485 简介
34.2 硬件设计
34.3 软件设计
34.4 下载验证

34.1 485 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。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）

④ 一般最大支持 32 个节点，如果使用特制的 485 芯片，可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

RS485 推荐使用在点对点网络中，线型，总线型，不能是星型，环型网络。理想情况下 RS485 需要 2 个终端匹配电阻，其阻值要求等于传输电缆的特性阻抗（一般为 120Ω）。没有特性阻抗的话，当所有的设备都静止或者没有能量的时候就会产生噪声，而且线移需要双端的电压差。没有终接电阻的话，会使得较快速的发送端产生多个数据信号的边缘，导致数据传输出错。485 推荐的连接方式如图 34.1.2 所示：

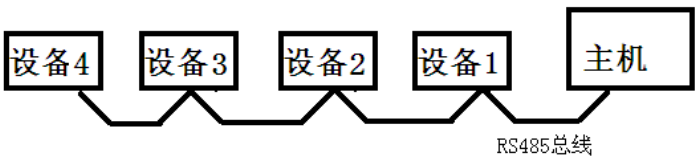


图 34.1.2 RS485 连接

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。

阿波罗 STM32F767 开发板采用 SP3485 作为收发器，该芯片支持 3.3V 供电，最大传输速度可达 10Mbps，支持多达 32 个节点，并且有输出短路保护。该芯片的框图如图 34.1.2 所示：

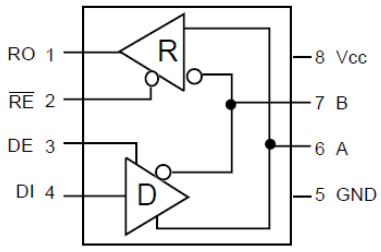


图 34.1.2 SP3485 框图

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 STM32F767 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：通过连接两个阿波罗 STM32F767 开发板的 RS485 接口，然后由 KEY0 控制发送，当按下一个开发板的 KEY0 的时候，就发送 5 个数据给另外一个开发板，并在两个开发板上分别显示发送的值和接收到的值。

本章，我们只需要配置好串口 2，就可以实现正常的 485 通信了，串口 2 的配置和串口 1 基本类似，只是串口的时钟来自 APB1，最大频率为 54Mhz。

34.2 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） LCD 模块
4） PCF8574T
5） 串口 2
6） RS485 收发芯片 SP3485

前面 4 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 34.2.1 所示：

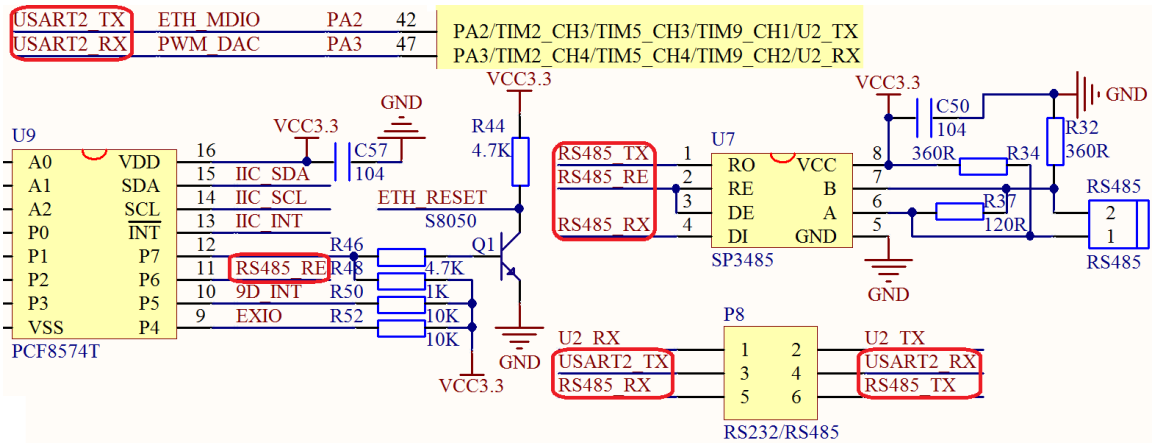


图 34.2.1 STM32F767 与 SP3485 连接电路图

从上图可以看出：STM32F767 的串口 2 通过 P8 端口设置，连接到 SP3485。注意：RS485_RE 信号，是连接在 PCF8574T 的 P6 脚上的，并没有直接连接到 MCU，需要通过 IIC 总线控制 PCF8574T，从而实现对 RS485_RE 的控制。RS485_RE 控制 SP3485 的收发，当 RS485_RE=0

的时候，为接收模式；当 RS485_RE=1 的时候，为发送模式。

另外，PA2，PA3 和 ETH_MDIO 和 PWM_DAC 有共用 IO，所以在使用的时候，注意分时复用，不能同时使用。

图中的 R34 和 R32 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 P8 排针的连接，通过跳线帽将 PA2 和 PA3 分别连接到 485_TX 和 485_RX 上面，如图 34.2.2 所示：

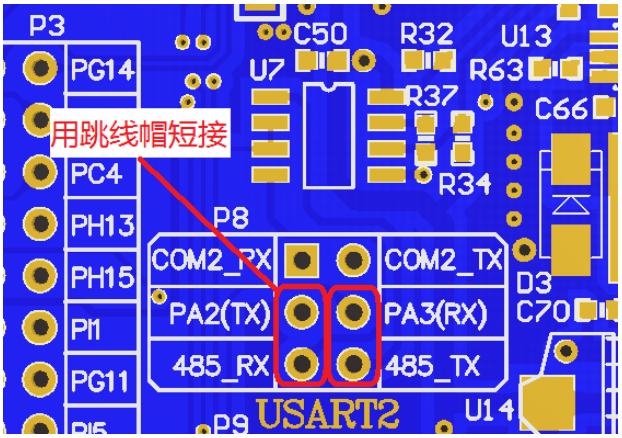


图 34.2.2 硬件连接示意图

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！！

34.3 软件设计

打开 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 stm32f7xx_hal_uart.c 文件和对应的头文件 stm32f7xx_hal_uart.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;

    PCF8574_Init();                          //初始化 PCF8574，用于控制 RE 脚

    __HAL_RCC_GPIOA_CLK_ENABLE();   /使能 GPIOA 时钟
    __HAL_RCC_USART2_CLK_ENABLE(); //使能 USART2 时钟

    GPIO_Initure.Pin=GPIO_PIN_2|GPIO_PIN_3;   //PA2,3
    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_HIGH;       //高速
    GPIO_Initure.Alternate=GPIO_AF7_USART2;   //复用为 USART2
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);       //初始化 PA2,3

    //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 8 位数据
    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_Set(0);                                    //设置为接收模式
}

//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_Set(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_Set(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
    }
}
//RS485 模式控制.
//en:0,接收;1,发送.
void RS485_TX_Set(u8 en)
{
    PCF8574_WriteBit(RS485_RE_IO,en);
}
```

此部分代码总共 5 个函数，其中 RS485_Init 函数为 485 通信初始化函数，完成对串口 2 的配置，另外，对 PCF8574 也进行了初始化，方便控制 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；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，这里重点介绍一下接收数据的流程

（超时法）：首先令 rxlen=RS485_RX_CNT，记录当前接收到的字节数，随后，等待 10ms，如果在这个 10ms 里面，没有接收到任何数据（RS485_RX_CNT 的值未增加），那么就说明接收完成了。如果有接收到其他数据（RS485_RX_CNT 变大了），那么说明还在继续接收数据，需等到下一个循环再处理；最后，RS485_TX_Set 函数，用于通过 PCF8574 控制 RS485_RE 脚。

对于串口 2 程序编写方式我们需要说明一下。在串口实验章节我们已经讲过，一般情况下，我们在串口的中断服务函数中会调用中断共用处理 HAL 库函数 HAL_UART_IRQHandler，然后在函数 HAL_UART_IRQHandler 中，会对中断进行判断，调用想用的回调处理函数。但是本章实验源码中，我们并没有在中断服务函数中调用 HAL_UART_IRQHandler，也没有修改中断接收回调函数 HAL_UART_RxCpltCallback，这是因为我们为了保持 RS485 串口部分代码的独立性，所以直接在中断服务函数中编写中断控制逻辑，方便大家阅读。大家也可以根据自己的编程习惯来选择，HAL 库是非常灵活的，当我们掌握了 HAL 的编程思路，就可以得心应手的使用。对于串口初始化 MSP 回调函数也是一样，我们并没有修改初始化回调函数 HAL_UART_MspInit 内容，而是直接在 RS485_Init 函数中一次性初始化所有步骤。

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声明，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];
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    RS485_Init(9600);               //初始化 RS485
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/7/12");
    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_Toggle;//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(54,9600)，这里用的是 54，而不是 108，是因为 APB1 的时钟是 54Mhz，故是 54，而串口 1 的时钟来自 APB2，是 108Mhz 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。

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 阿波罗 STM32 开发板上（注意要 2 个开发板都下载这个代码哦），得到如图 34.4.1 所示：

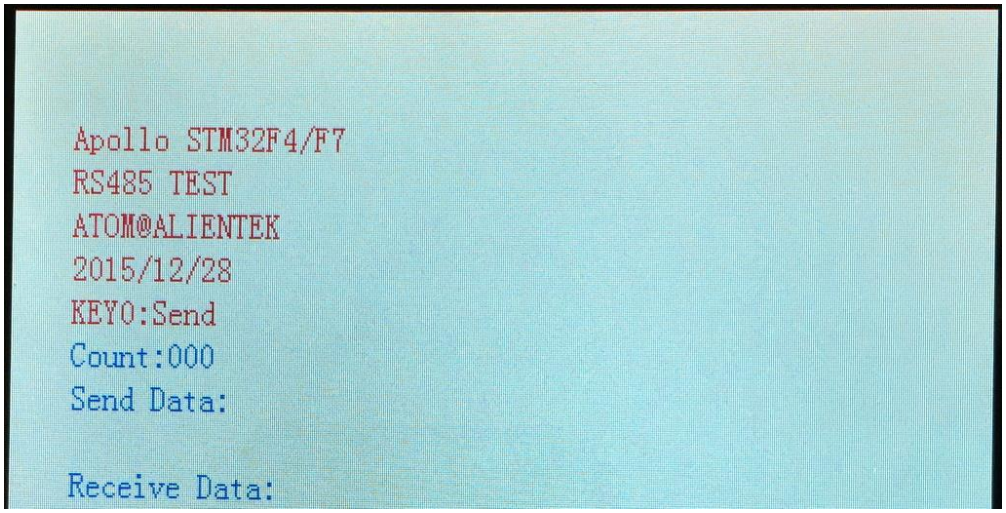


图 34.4.1 程序运行效果图

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 34.4.2 和图 41.4.3 所示：

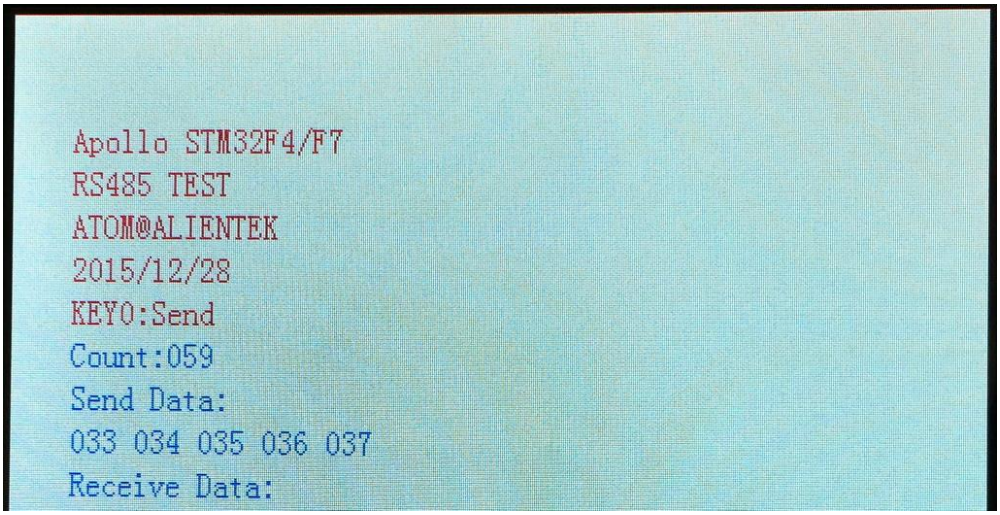


图 34.4.2 RS485 发送数据

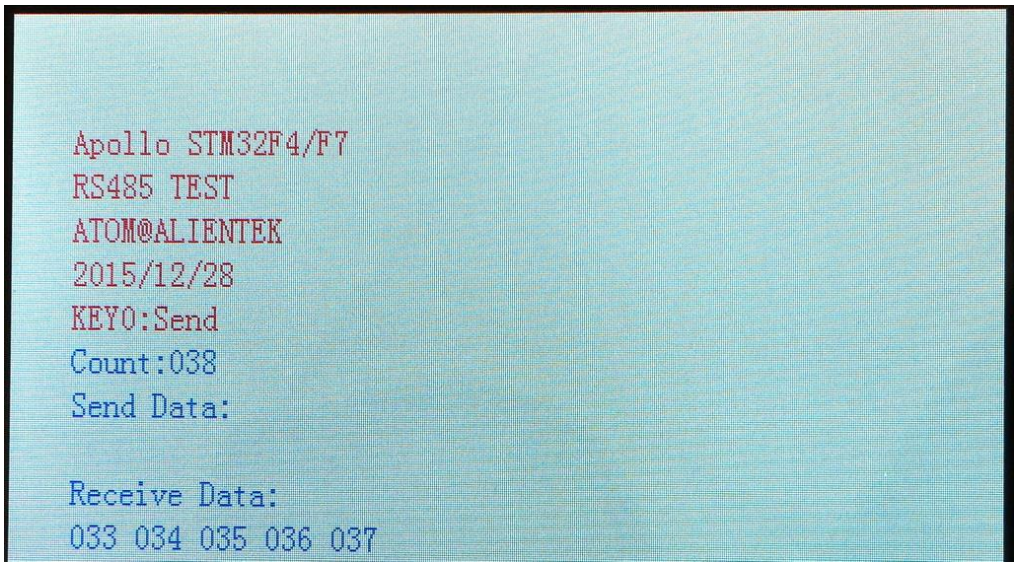


图 34.4.3 RS485 接收数据

图 34.4.2 来自开发板 A，发送了 5 个数据，图 34.4.3 来自开发板 B，接收到了来自开发板 A 的 5 个数据。

本章介绍的 485 总线时通过串口控制收发的，我们只需要将 P8 的跳线帽稍作改变，该实验就变成了一个 RS232 串口通信实验了，通过对接两个开发板的 RS232 接口，即可得到同样的实验现象，有兴趣的读者可以实验一下。

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。

第三十五章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32F767 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 LCD 模块上。本章分为如下几个部分：

35.1 CAN 简介
35.2 硬件设计
35.3 软件设计
35.4 下载验证

35.1 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 35.1.1 所示：

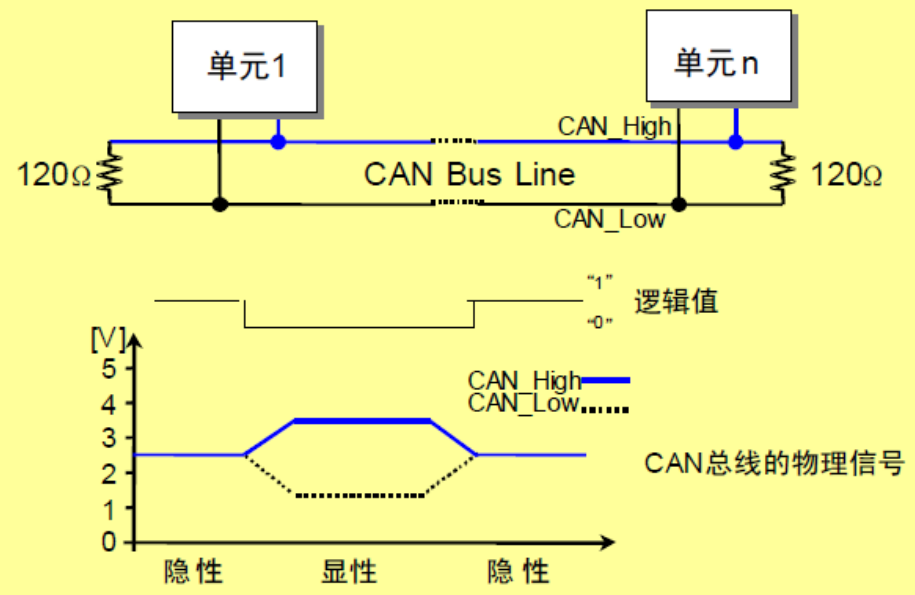


图 35.1.1 ISO11898 物理层特性

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 35.1.1 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 35.1.1 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 35.1.2 所示：

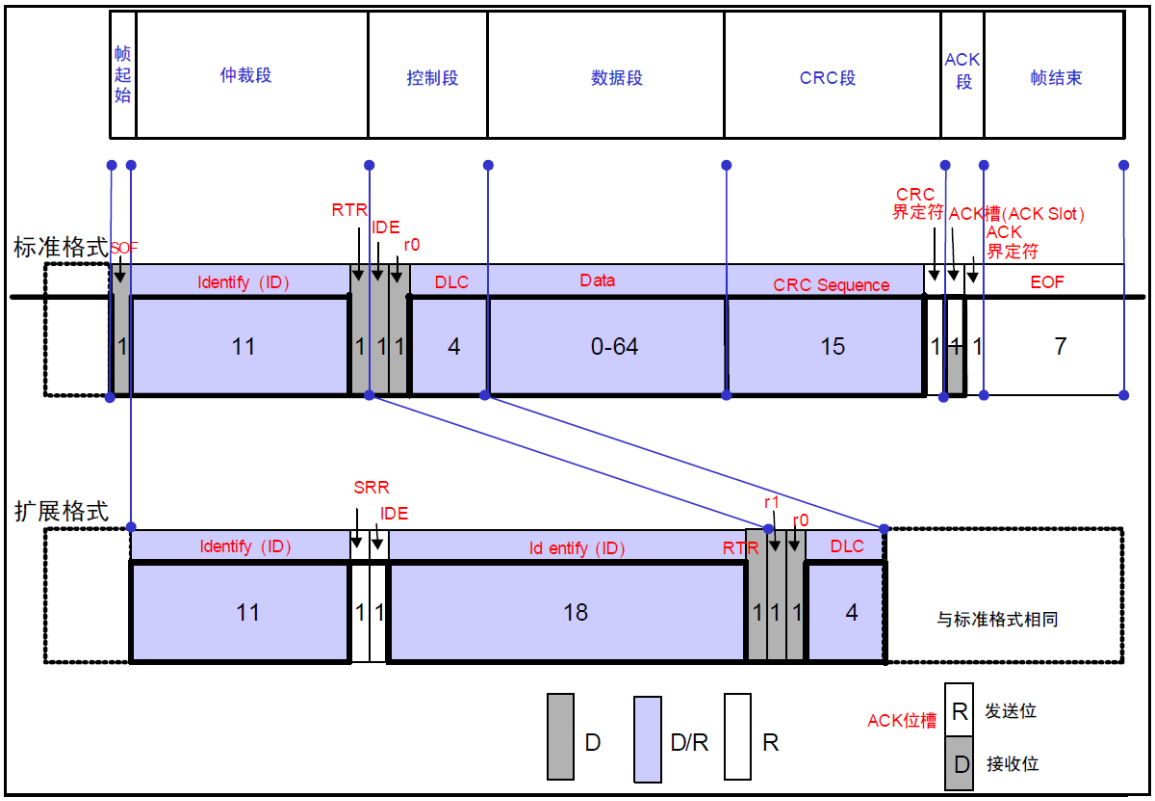


图 35.1.2 数据帧的构成

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 35.1.3 所示：

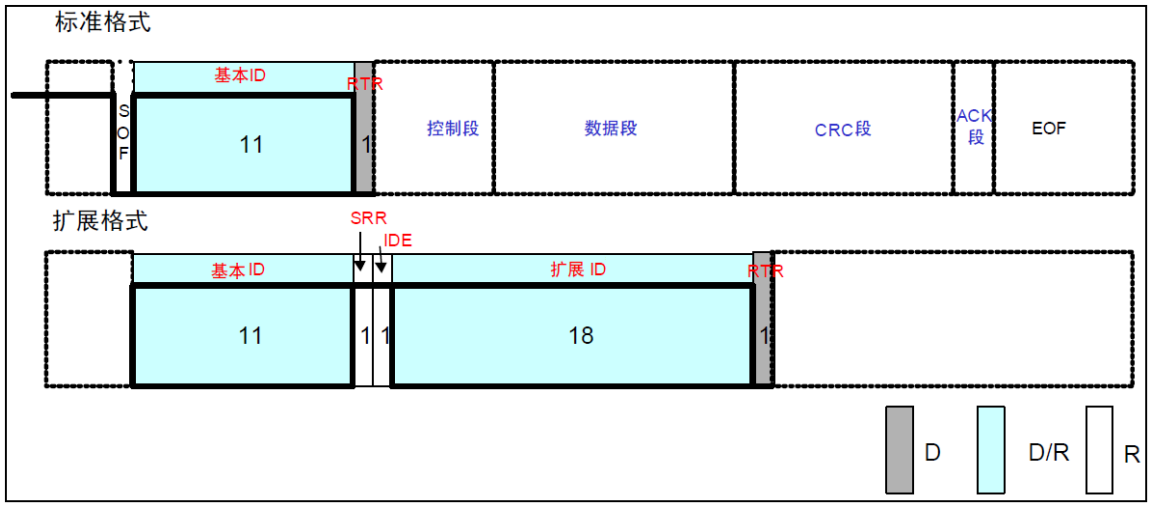


图 35.1.3 数据帧仲裁段构成

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 35.1.4 所示：

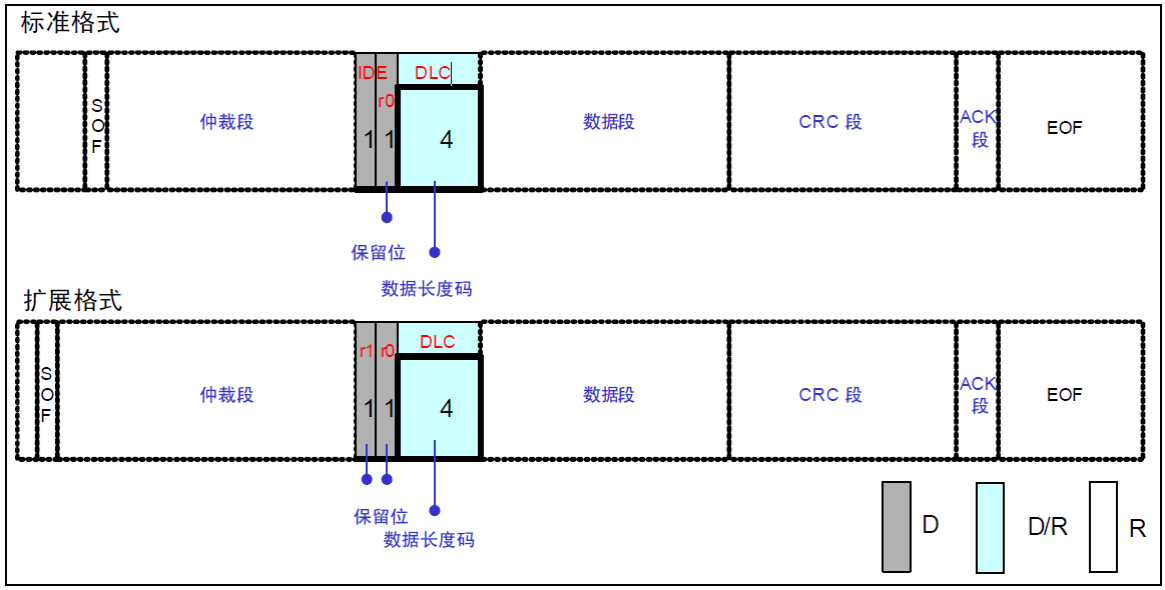


图 35.1.4 数据帧控制段构成

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 35.1.5 所示：

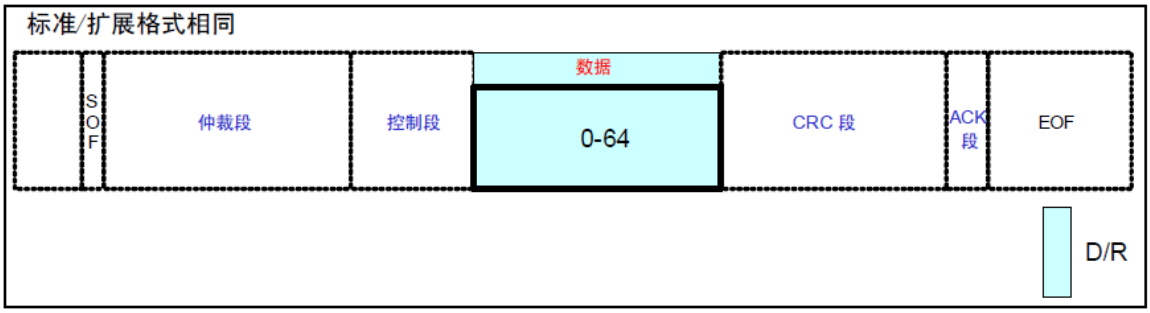


图 35.1.5 数据帧数据段构成

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 35.1.6 所示：

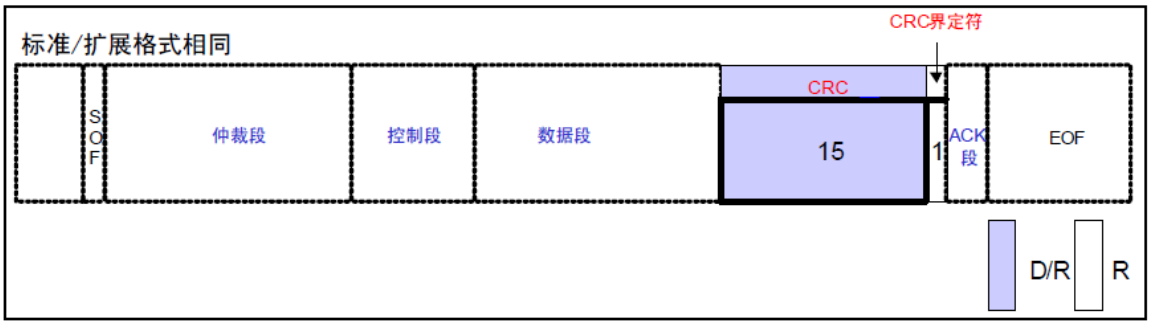


图 35.1.6 数据帧 CRC 段构成

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 35.1.7 所示：

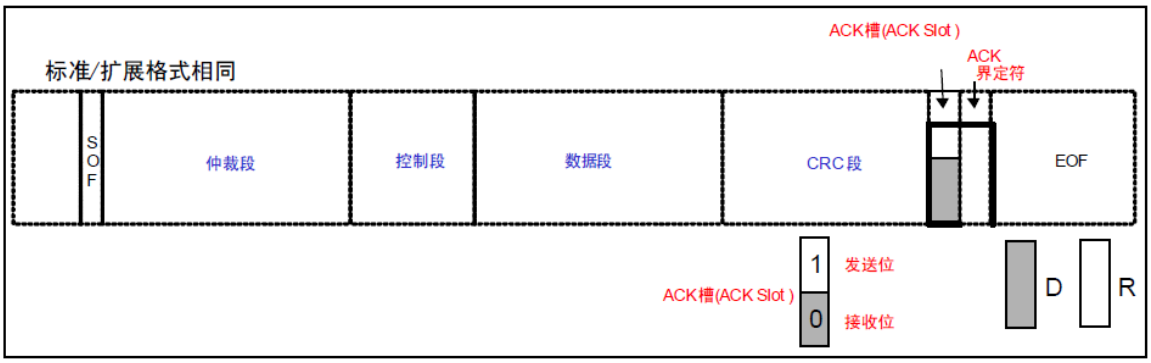


图 35.1.7 数据帧 CRC 段构成

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 35.1.2 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 35.1.2 一个位各段及其作用

1 个位的构成如图 35.1.8 所示：

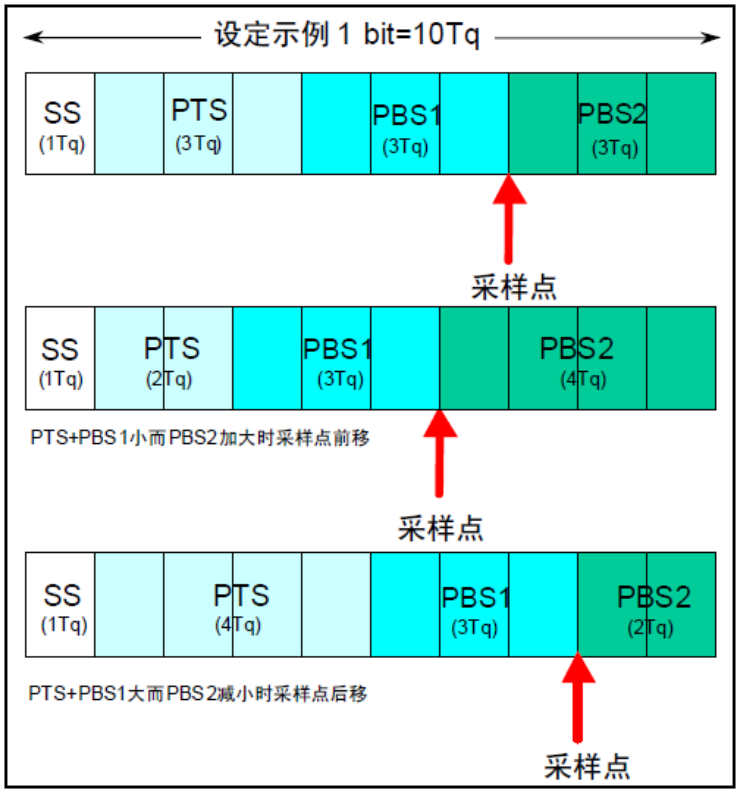


图 35.1.8 一个位的构成

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 35.1.9 所示：

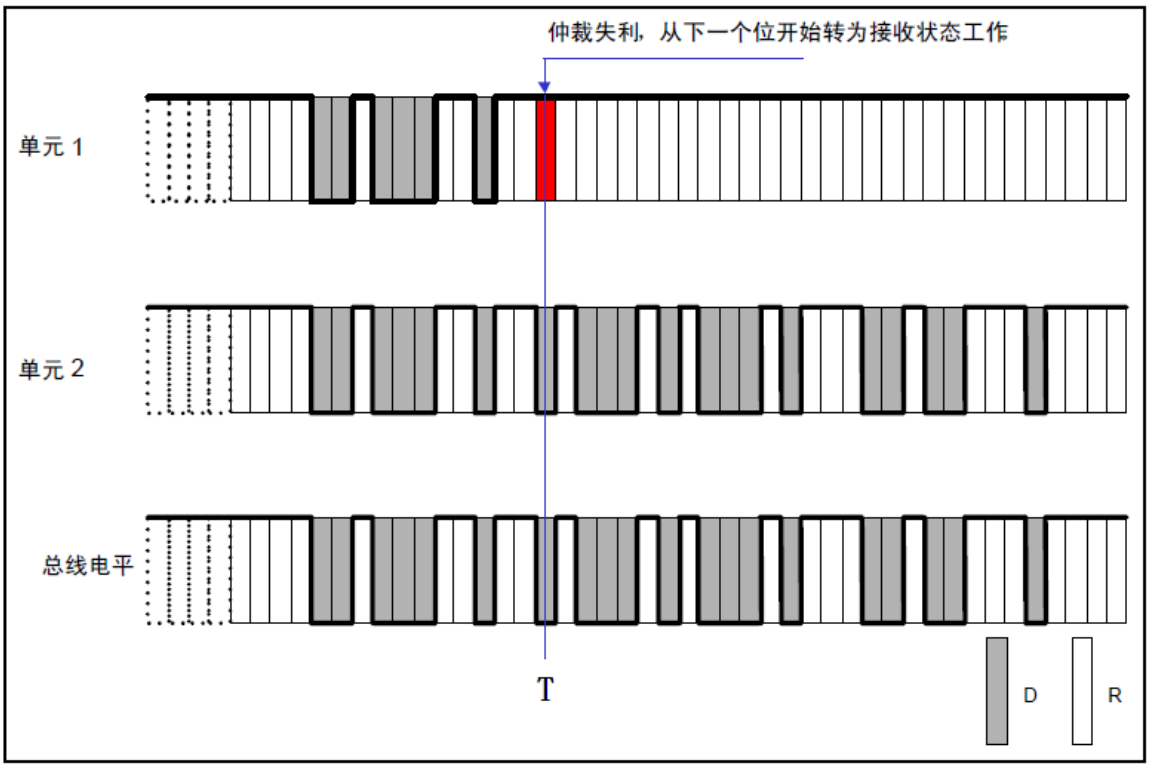


图 35.1.9 CAN 总线仲裁过程

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 STM32F767 的 CAN 控制器。

STM32F767 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。

STM32F767 的 bxCAN 的主要特点有：

- 支持 CAN 协议 2.0A 和 2.0B 主动模式
- 波特率最高达 1Mbps
- 支持时间触发通信
- 具有 3 个发送邮箱
- 具有 3 级深度的 2 个接收 FIFO
- 可变的过滤器组（28 个，CAN1 和 CAN2 共享）

在 STM32F767IGT6 中，带有 2 个 CAN 控制器，而我们本章只用了 1 个 CAN，即 CAN1。双 CAN 的框图如图 35.1.10 所示：

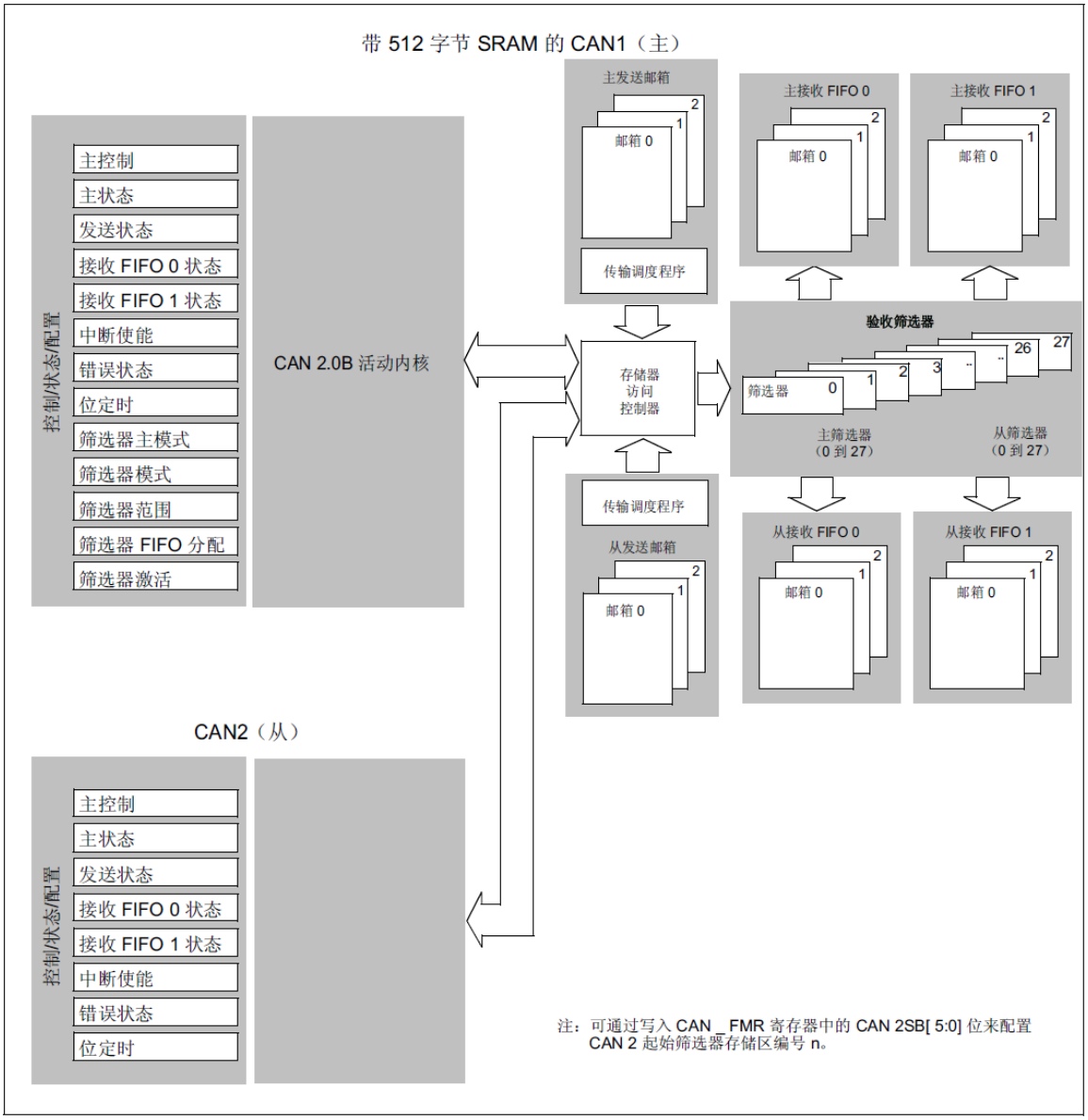


图 35.1.10 双 CAN 框图

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。

STM32F767 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。STM32F767 的过滤器（也称筛选器）组最多有 28 个，每个滤波器组 x 由 2 个 32 为寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32F767 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 35.1.11 所示：

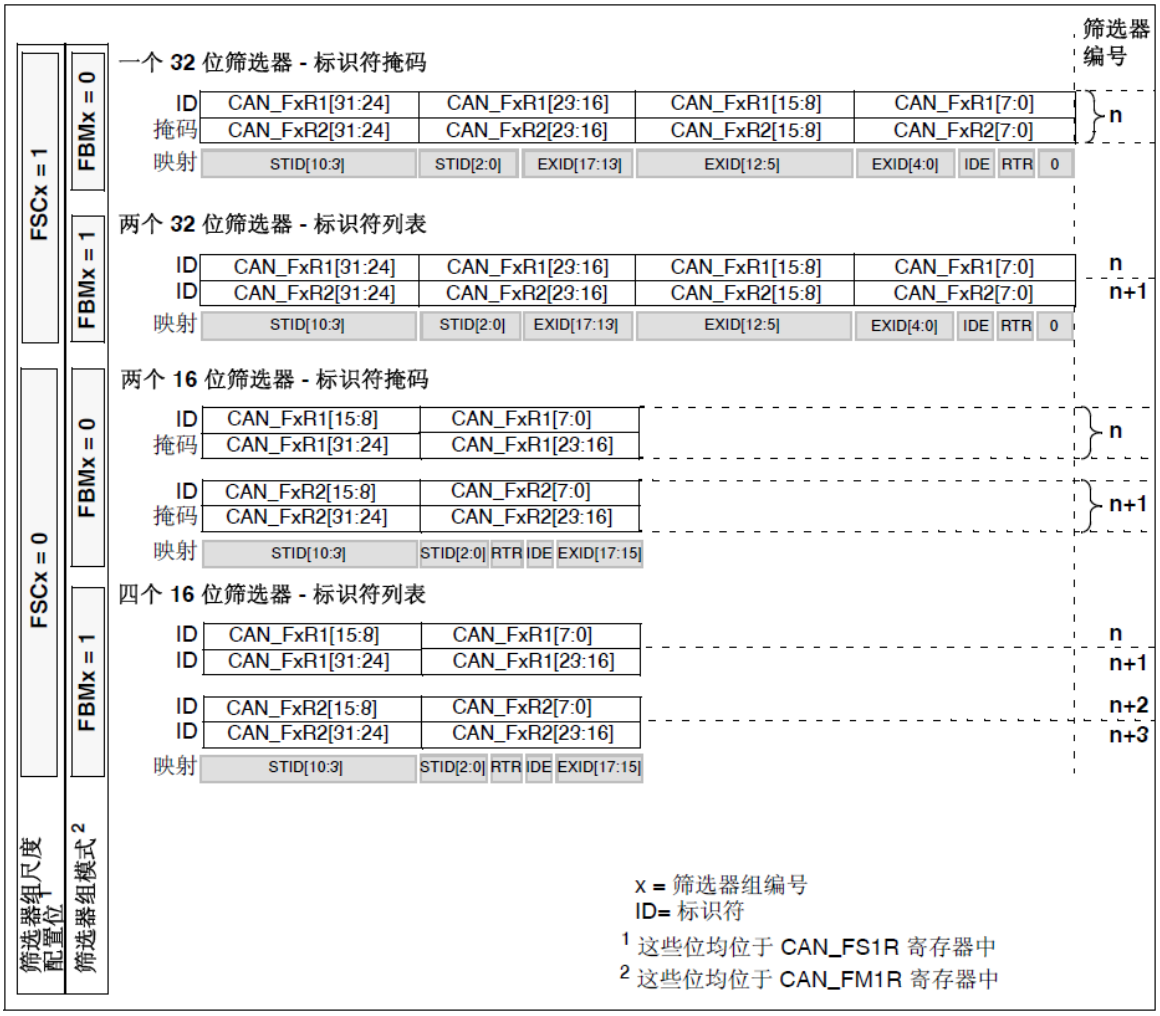


图 35.1.11 过滤器组位宽模式设置

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 35.1.11 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 位过滤器-标识符屏蔽模式，然后设置 CAN_F0R1=0XFFFF0000，CAN_F0R2=0XFF00FF00。其中存放到 CAN_F0R1 的值就是期望收到的 ID，即我们希望收到的 ID（STID+EXTID+IDE+RTR）最好是：0XFFFF0000。而 0XFF00FF00 就是设置我们需要必须关心的 ID，表示收到的 ID，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_F0R1 中对应的位一模一样，而另外的 16 个位则不关心，可以一样，也可以不一样，都认为是正确的 ID，即收到的 ID 必须是 0XFFxx00xx，才算是正确的（x 表示不关心）。

关于标识符过滤的详细介绍，请参考《STM32F7 中文参考手册》的 36.7.4 节（1152 页）。接下来，我们看看 STM32F767 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 35.1.12 所示：

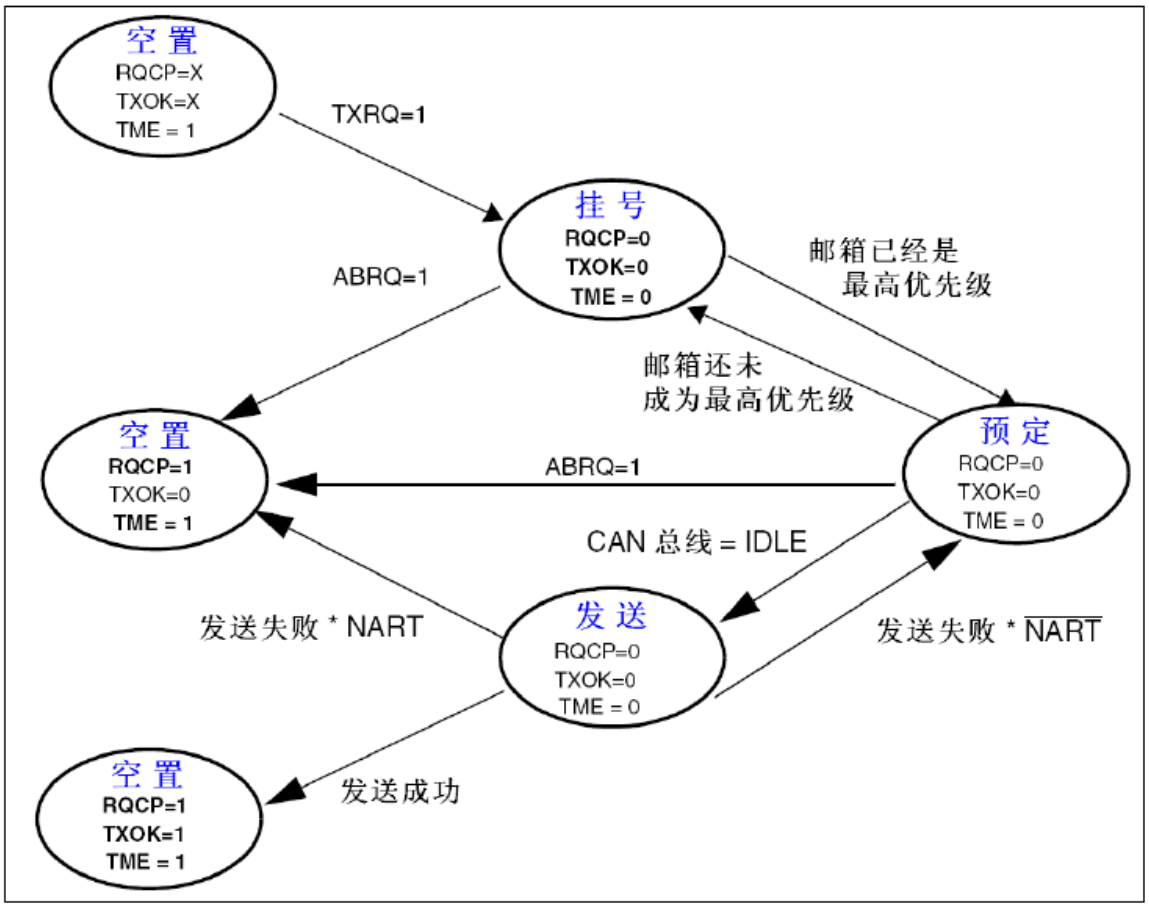


图 35.1.12 发送邮箱

上图中，还包含了很多其他处理，终止发送（ABRQ=1）和发送失败处理等。通过这个流程图，我们大致了解了 CAN 的发送流程，后面的数据发送，我们基本就是按照此流程来走。接下来再看看 CAN 的接收流程。

CAN 接收流程

CAN 接收到的有效报文，被存储在 3 级邮箱深度的 FIFO 中。FIFO 完全由硬件来管理，从而节省了 CPU 的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取 FIFO 输出邮箱，来读取 FIFO 中最先收到的报文。这里的有效报文是指那些正确被接收的（直到 EOF 都没有错误）且通过了标识符过滤的报文。前面我们知道 CAN 的接收有 2 个 FIFO，我们每个滤波器组都可以设置其关联的 FIFO，通过 CAN_FFA1R 的设置，可以将滤波器组关联到 FIFO0/FIFO1。

CAN 接收流程为：FIFO 空→收到有效报文→挂号_1（存入 FIFO 的一个邮箱，这个由硬件控制，我们不需要理会）→收到有效报文→挂号_2→收到有效报文→挂号_3→收到有效报文→溢出。

这个流程里面，我们没有考虑从 FIFO 读出报文的情况，实际情况是：我们必须在 FIFO 溢出之前，读出至少 1 个报文，否则下个报文到来，将导致 FIFO 溢出，从而出现报文丢失。每读出 1 个报文，相应的挂号就减 1，直到 FIFO 空。CAN 接收流程如图 35.1.13 所示：

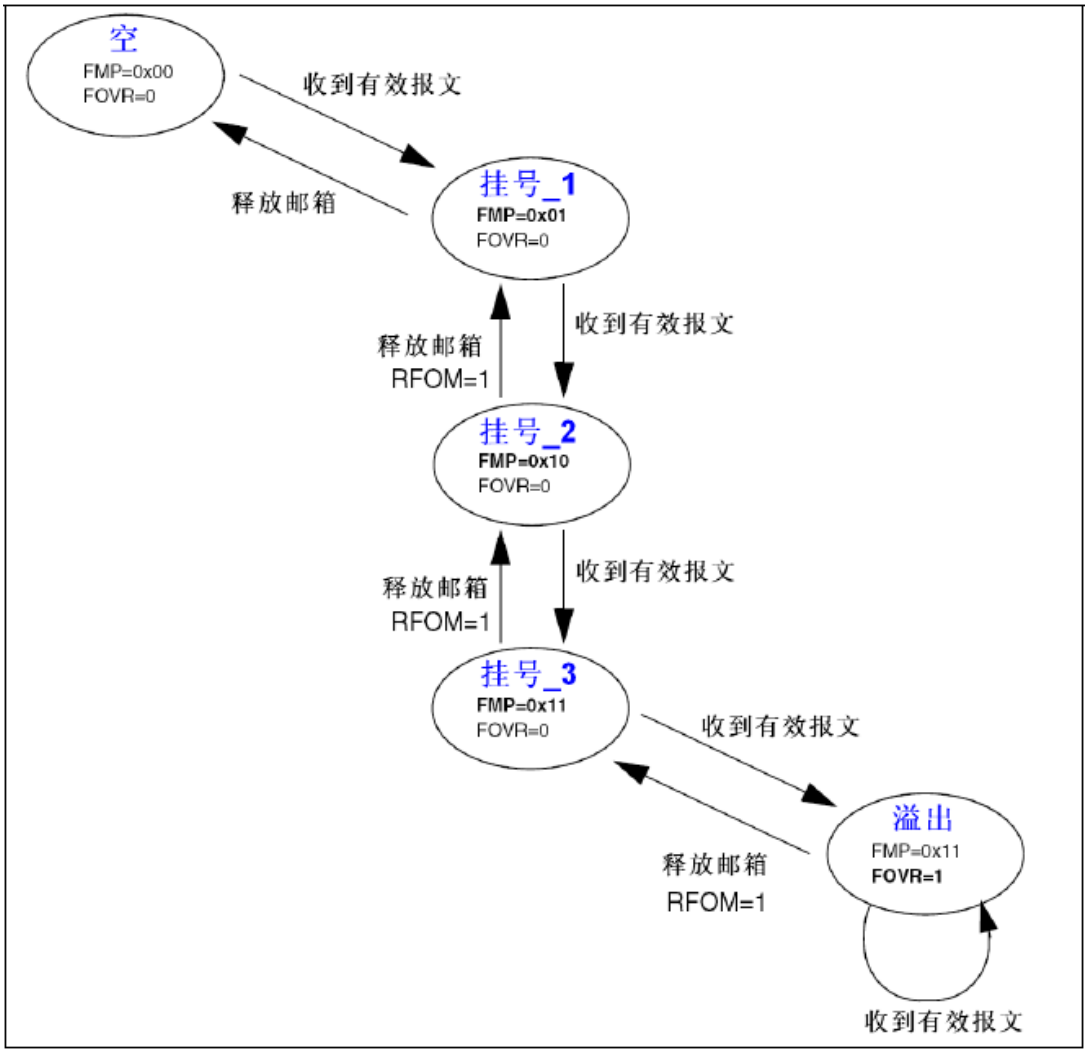
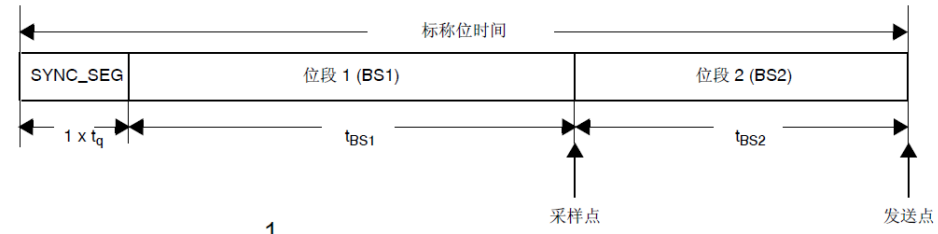


图 35.1.13 FIFO 接收报文

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 STM32F767 的 CAN 位时间特性，STM32F767 的 CAN 位时间特性和之前我们介绍的，稍有点区别。STM32F767 把传播时间段和相位缓冲段 1（STM32F767 称之为时间段 1）合并了，所以 STM32F767 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）。STM32F767 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。STM32F767 的 CAN 位时序如图 35.1.14 所示：



$$BaudRate = \frac{1}{NominalBitTime}$$

$$NominalBitTime = 1 \times t_q + t_{BS1} + t_{BS2}$$

其中：

$t_{BS1} = t_q \times (TS1[3:0] + 1)$，

$t_{BS2} = t_q \times (TS2[2:0] + 1)$，

$t_q = (BRP[9:0] + 1) \times t_{PCLK}$

其中 t_q 为时间片

t_{PCLK} = APB 时钟的时间周期，

BRP[9:0]、TS1[3:0] 和 TS2[2:0] 在 CAN_BTR 寄存器中定义。

图 35.1.14 STM32F767 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 54Mhz)，就可以方便的计算出波特率。比如设置 TS1=10、TS2=7 和 BRP=6，在 APB1 频率为 54Mhz 的条件下，即可得到 CAN 通信的波特率=54000/[(7+10+1)*6]=500Kbps。

接下来，我们介绍一下本章需要用到的一些比较重要的寄存器。首先，来看 CAN 的主控制寄存器（CAN_MCR），该寄存器各位描述如图 35.1.15：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESET	Reserved							TTCM	ABOM	AWUM	NART	RFLM	TXFP	SLEEP	INRQ
rs								rw	rw	rw	rw	rw	rw	rw	rw

图 35.1.15 寄存器 CAN_MCR 各位描述

该寄存器的详细描述，请参考《STM32F7 中文参考手册》36.9.2 节，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 35.1.16 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	Reserved				SJW[1:0]		Res.	TS2[2:0]			TS1[3:0]			
rw	rw					rw	rw		rw	rw	rw	rw	rw	rw	rw

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

位 31 **SILM**：静默模式（调试）(Silent mode (debug))
0：正常工作
1：静默模式

位 30 **LBKM**：环回模式（调试）(Loop back mode (debug))
0：禁止环回模式
1：使能环回模式

位 29:26 保留，必须保持复位值。

位 25:24 **SJW[1:0]**：再同步跳转宽度 (Resynchronization jump width)
这些位定义 CAN 硬件在执行再同步时最多可以将位加长或缩短的时间片数目。
$t_{RJW} = t_{CAN} \times (SJW[1:0] + 1)$

位 23 保留，必须保持复位值。

位 22:20 **TS2[2:0]**：时间段 2 (Time segment 2)
这些位定义时间段 2 中的时间片数目。
$t_{BS2} = t_{CAN} \times (TS2[2:0] + 1)$

位 19:16 **TS1[3:0]**：时间段 1 (Time segment 1)
这些位定义时间段 1 中的时间片数目。
$t_{BS1} = t_{CAN} \times (TS1[3:0] + 1)$

位 15:10 保留，必须保持复位值。

位 9:0 **BRP[9:0]**：波特率预分频器 (Baud rate prescaler)
这些位定义一个时间片的长度。
$t_q = (BRP[9:0]+1) \times t_{PCLK}$

图 35.1.16 寄存器 CAN_BTR 各位描述

STM32F767 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 35.1.17 所示：

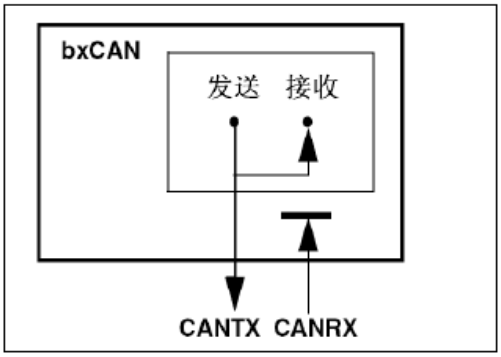


图 35.1.17 CAN 环回模式

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 35.1.18 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STID[10:0]/EXID[28:18]											EXID[17:13]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXID[12:0]													IDE	RTR	TXRQ
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:21 **STID[10:0]/EXID[28:18]**：标准标识符或扩展标识符 (Standard identifier or extended identifier)
标准标识符或扩展标识符的 MSB（取决于 IDE 位的值）。

位 20:3 **EXID[17:0]**：扩展标识符 (Extended identifier)
扩展标识符的 LSB。

位 2 **IDE**：标识符扩展 (Identifier extension)
此位用于定义邮箱中消息的标识符类型。
0：标准标识符。
1：扩展标识符。

位 1 **RTR**：远程发送请求 (Remote transmission request)
0：数据帧
1：遥控帧

位 0 **TXRQ**：发送邮箱请求 (Transmit mailbox request)
由软件置 1，用于请求发送相应邮箱的内容。
邮箱变为空后，此位由硬件清零。

图 35.1.18 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 35.1.19 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:24 **DATA3[7:0]**：数据字节 3 (Data byte 3)
消息的数据字节 3。

位 23:16 **DATA2[7:0]**：数据字节 2 (Data byte 2)
消息的数据字节 2。

位 15:8 **DATA1[7:0]**：数据字节 1 (Data byte 1)
消息的数据字节 1。

位 7:0 **DATA0[7:0]**：数据字节 0 (Data byte 0)
消息的数据字节 0。
一条消息可以包含 0 到 8 个数据字节，从字节 0 开始。

图 35.1.19 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《STM32F7 中文参考手册 》36.9 节。

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 35.1.20 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				rw	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，必须保持复位值。

位 27:0 **FBM*x***：筛选器模式 (Filter mode)
筛选器 x 的寄存器的模式
0：筛选器存储区 x 的两个 32 位寄存器处于标识符屏蔽模式。
1：筛选器存储区 x 的两个 32 位寄存器处于标识符列表模式。

图 35.1.20 寄存器 CAN_FM1R 各位描述

该寄存器用于设置各滤波器组的工作模式，对 28 个滤波器组的工作模式，都可以通过该寄存器设置，不过该寄存器必须在过滤器处于初始化模式下（CAN_FMR 的 FINIT 位=1），才可以进行设置。

第八个，我们介绍 CAN 过滤器位宽寄存器(CAN_FS1R)，该寄存器各位描述如图 35.1.21 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				rw	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，必须保持复位值。

位 27:0 **FSCx**：筛选器尺度配置 (Filter scale configuration)
这些位定义了筛选器 13-0 的尺度配置。
0：双 16 位尺度配置
1：单 32 位尺度配置

图 35.1.21 寄存器 CAN_FS1R 各位描述

该寄存器用于设置各滤波器组的位宽，对 28 个滤波器组的位宽设置，都可以通过该寄存器实现。该寄存器也只能在过滤器处于初始化模式下进行设置。

第九个，我们介绍 CAN 过滤器 FIFO 关联寄存器（CAN_FFA1R），该寄存器各位描述如图 35.1.22 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FFA15	FFA14	FFA13	FFA12	FFA11	FFA10	FFA9	FFA8	FFA7	FFA6	FFA5	FFA4	FFA3	FFA2	FFA1	FFA0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:28 保留，必须保持复位值。

位 27:0 **FFA*x***：筛选器 x 的筛选器 FIFO 分配 (Filter FIFO assignment for filter x)
通过此筛选器的消息将存储在指定的 FIFO 中。
0：筛选器分配到 FIFO 0
1：筛选器分配到 FIFO 1

图 35.1.22 寄存器 CAN_FFA1R 各位描述

该寄存器设置报文通过滤波器组之后，被存入的 FIFO，如果对应位为 0，则存放到 FIFO0；如果为 1，则存放到 FIFO1。该寄存器也只能在过滤器处于初始化模式下配置。

第十个，我们介绍 CAN 过滤器激活寄存器（CAN_FA1R），该寄存器各位对应滤波器组和前面的几个寄存器类似，这里就不列出了，对对应位置 1，即开启对应的滤波器组；置 0 则关闭该滤波器组。

最后，我们介绍 CAN 的过滤器组 i 的寄存器 x（CAN_FiRx）（i=0~27；x=1/2）。该寄存器各位描述如图 35.1.23 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

位 31:0 **FB[31:0]**：筛选器位 (Filter bits)
标识符
寄存器的每一位用于指定预期标识符相应位的级别。
0：需要显性位
1：需要隐性位
掩码
寄存器的每一位用于指定相关标识符寄存器的位是否必须与预期标识符的相应位匹配。
0：无关，不使用此位进行比较。
1：必须匹配，传入标识符的此位必须与筛选器相应标识符寄存器中指定的级别相同。

图 35.1.23 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 35.1.11。

关于 CAN 的介绍，就到此结束了。接下来，我们看看本章我们将实现的功能，及 CAN 的配置步骤。

本章，我们通过 KEY_UP 按键选择 CAN 的工作模式（正常模式/环回模式），然后通过 KEY0 控制数据发送，并通过查询的办法，将接收到的数据显示在 LCD 模块上。如果是环回模式，我们用一个开发板即可测试。如果是正常模式，我们就需要 2 个阿波罗开发板，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。

HAL 库中 CAN 相关的函数在文件 stm32f7xx_hal_can.c 和对应的头文件 stm32f7xx_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（AF9），使能 CAN 时钟。

我们要用 CAN，第一步就要使能 CAN 的时钟，CAN 的时钟通过 APB1ENR 的第 25 位来设置。其次要设置 CAN 的相关引脚为复用输出，这里我们需要设置 PA11（CAN1_RX）和 PA12（CAN1_TX）为复用功能（AF9），并使能 PA 口的时钟。具体配置过程如下：

```
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_11|GPIO_PIN_12;       //PA11,12
GPIO_Initure.Mode=GPIO_MODE_AF_PP;              //推挽复用
GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
GPIO_Initure.Speed=GPIO_SPEED_FAST;             //快速
GPIO_Initure.Alternate=GPIO_AF9_CAN1;           //复用为 CAN1
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;
  CanTxMsgTypeDef*            pTxMsg;
  CanRxMsgTypeDef*            pRxMsg;
  __IO HAL_CAN_StateTypeDef   State;
  HAL_LockTypeDef             Lock;
  __IO uint32_t               ErrorCode;
}CAN_HandleTypeDef;
```

该结构体除了 State ，Lock ，和 ErrorCode 三个 HAL 库处理状态过程变量之外，只有四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。

第三个成员变量 pTxMsg 和第四个成员变量 pRxMsg 是发送和接收消息结构体指针，在初始化 CAN 的时候要指定其指向，那么在我们后面调用发送函数 HAL_CAN_Transmit 之前，就可以初始化 pTxMsg 指定发送数据和参数，在调用接收函数 HAL_CAN_Receive 之后，就可以通过 pRxMsg 获取接收数据和参数。

接下来我们着重看看第二个成员变量 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 SJW;
  uint32_t BS1;
  uint32_t BS2;
  uint32_t TTCM;
  uint32_t ABOM;
  uint32_t AWUM;
  uint32_t NART;
  uint32_t RFLM;
  uint32_t TXFP;
}CAN_InitTypeDef;
```

这个结构体看起来成员变量比较多，实际上参数可以分为两类。前面 5 个参数是用来设置寄存器 CAN_BTR，用来设置模式以及波特率相关的参数，这在前面有讲解过，设置模式的参数是 Mode，我们实验中用到回环模式 CAN_MODE_LOOPBACKk 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，SJW，BS1 和 BS2 分别用来设置波特率分频器，重新同步跳跃宽度以及时间段 1 和时间段 2 占用的时间单元数。后面 6 个成员变量用来设置寄存器 CAN_MCR，也就是设置 CAN 通信相关的控制位。大家可以去翻翻中文参考手册中这两个寄存器的描述，非常详细，我们在前面也有讲解到。初始化实例为：

```
CAN_HandleTypeDef     CAN1_Handler;     //CAN1 句柄
CanTxMsgTypeDef       TxMessage;        //发送消息
CanRxMsgTypeDef       RxMessage;        //接收消息

CAN1_Handler.Instance=CAN1;
CAN1_Handler.pTxMsg=&TxMessage;     //发送消息
CAN1_Handler.pRxMsg=&RxMessage;     //接收消息
CAN1_Handler.Init.Prescaler=6;      //分频系数(Fdiv)为 brp+1
CAN1_Handler.Init.Mode= CAN_MODE_LOOPBACK;          //模式设置:普通模式
CAN1_Handler.Init.SJW= CAN_SJW_1TQ;      //重新同步跳跃宽度为 tsjw+1 个时间单位
CAN1_Handler.Init.BS1= CAN_BS1_8TQ;      //tbs1 范围 CAN_BS1_1TQ~CAN_BS1_16TQ
CAN1_Handler.Init.BS2= CAN_BS2_6TQ;      //tbs2 范围 CAN_BS2_1TQ~CAN_BS2_8TQ
```

```
CAN1_Handler.Init.TTCM=DISABLE;      //非时间触发通信模式
CAN1_Handler.Init.ABOM=DISABLE;      //软件自动离线管理
CAN1_Handler.Init.AWUM=DISABLE;      //睡眠模式通过软件唤醒
CAN1_Handler.Init.NART=ENABLE;       //禁止报文自动传送
CAN1_Handler.Init.RFLM=DISABLE;      //报文不锁定,新的覆盖旧的
CAN1_Handler.Init.TXFP=DISABLE;      //优先级由报文标识符决定
HAL_CAN_Init(&CAN1_Handler);
```

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_FilterConfTypeDef* 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_FilterConfTypeDef 结构体指针类型，用来设置滤波器相关参数，结构体 CAN_FilterConfTypeDef 定义为：

```
typedef struct
{
  uint32_t FilterIdHigh;
  uint32_t FilterIdLow;
  uint32_t FilterMaskIdHigh;
  uint32_t FilterMaskIdLow;
  uint32_t FilterFIFOAssignment;
  uint32_t FilterNumber;
  uint32_t FilterMode;
  uint32_t FilterScale;
  uint32_t FilterActivation;
  uint32_t BankNumber;
}CAN_FilterConfTypeDef;
```

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_FILTER_FIFO0。

第 6 个成员变量 FilterNumber 用来设置初始化的过滤器组，取值范围为 0~13。

第 7 个成员变量 FilterMode 用来设置过滤器组的模式，取值为标识符列表模式

CAN_FILTERMODE_IDLIST 和标识符屏蔽位模式 CAN_FILTERMODE_IDMASK。
第 8 个成员变量 FilterScale 用来设置过滤器的位宽为 2 个 16 位 CAN_FILTERSCALE_16BIT 还是 1 个 32 位 CAN_FILTERSCALE_32BIT。
第 9 个成员变量 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 CAN2 起始存储区。
过滤器初始化参考实例代码：

```
CAN_FilterConfTypeDef    CAN1_FilerConf;
CAN1_FilerConf.FilterIdHigh=0X0000;        //32 位 ID
CAN1_FilerConf.FilterIdLow=0X0000;
CAN1_FilerConf.FilterMaskIdHigh=0X0000; //32 位 MASK
CAN1_FilerConf.FilterMaskIdLow=0X0000;
CAN1_FilerConf.FilterFIFOAssignment=CAN_FILTER_FIFO0;//过滤器 0 关联到 FIFO0
CAN1_FilerConf.FilterNumber=0;    //过滤器 0
CAN1_FilerConf.FilterMode=CAN_FILTERMODE_IDMASK;
CAN1_FilerConf.FilterScale=CAN_FILTERSCALE_32BIT;
CAN1_FilerConf.FilterActivation=ENABLE; //激活滤波器 0
CAN1_FilerConf.BankNumber=14;
HAL_CAN_ConfigFilter(&CAN1_Handler,&CAN1_FilerConf);//初始化过滤器
```

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_Transmit(CAN_HandleTypeDef* hcan, uint32_t Timeout);
```

这个函数比较好理解，只有一个入口参数 hcan，为 CAN_HandleTypeDef 结构体指针类型。在发送消息之前，我们一般只需要再设置 hcan 的成员变量 pTxMsg 相关信息。
接受消息的函数是：

```
HAL_StatusTypeDef HAL_CAN_Receive(CAN_HandleTypeDef* hcan,
                                  uint8_t FIFONumber, uint32_t Timeout);
```

第一个入口参数为 CAN 句柄，第二个为 FIFO 号。我们接受之后，只需要读取 hcan 的成员变量 pRxMsg 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。

35.2 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） LCD 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 STM32F767 与 TJA1050 连接关系，如图 35.2.1 所示：

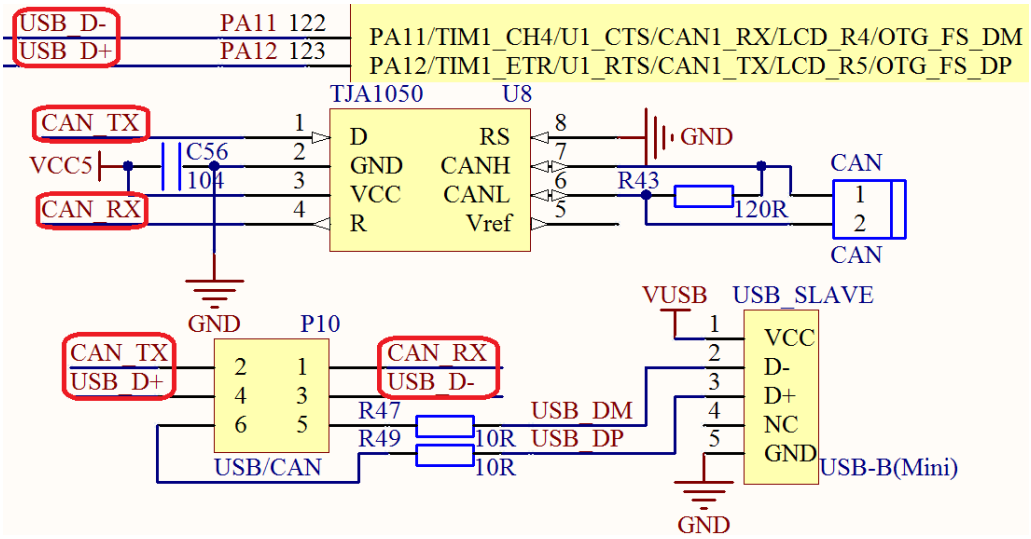


图 35.2.1 STM32F767 与 TJA1050 连接电路图

从上图可以看出：STM32F767 的 CAN 通过 P10 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，在阿波罗 STM32 开发板上面是带有 120Ω 的终端电阻的，如果我们的开发板不是作为 CAN 的终端的话，需要把这个电阻去掉，以免影响通信。另外，需要注意：CAN1 和 USB 共用了 PA11 和 PA12，所以他们不能同时使用。

这里还要注意，我们要设置好开发板上 P10 排针的连接，通过跳线帽将 PA11 和 PA12 分别连接到 CAN_RX 和 CAN_TX 上面，如图 35.2.2 所示：

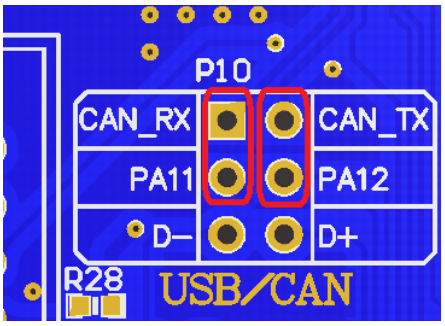


图 35.2.2 硬件连接示意图

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！！

35.3 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 stm32f7xx_hal_can.c 和头文件 stm32f7xx_hal_can.h 中。

打开 can.c 文件，代码如下：

```
CAN_HandleTypeDef     CAN1_Handler;    //CAN1 句柄
CanTxMsgTypeDef       TxMessage;       //发送消息
CanRxMsgTypeDef       RxMessage;       //接收消息

////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 54M,如果设置 CAN1_Mode_Init(
//CAN_SJW_1tq,CAN_BS2_6tq,CAN_BS1_11tq,6,CAN_MODE_LOOPBACK);
//则波特率为:54M/((6+11+1)*6)=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_FilterConfTypeDef    CAN1_FilerConf;

    CAN1_Handler.Instance=CAN1;
    CAN1_Handler.pTxMsg=&TxMessage;      //发送消息
    CAN1_Handler.pRxMsg=&RxMessage;      //接收消息
    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.SJW=tsjw;                  //重新同步跳跃宽度
    CAN1_Handler.Init.BS1=tbs1;          //tbs1 范围 CAN_BS1_1TQ~CAN_BS1_16TQ
    CAN1_Handler.Init.BS2=tbs2;          //tbs2 范围 CAN_BS2_1TQ~CAN_BS2_8TQ
    CAN1_Handler.Init.TTCM=DISABLE;      //非时间触发通信模式
    CAN1_Handler.Init.ABOM=DISABLE;      //软件自动离线管理
    CAN1_Handler.Init.AWUM=DISABLE;      //睡眠模式通过软件唤醒
    CAN1_Handler.Init.NART=ENABLE;       //禁止报文自动传送
    CAN1_Handler.Init.RFLM=DISABLE;      //报文不锁定,新的覆盖旧的
    CAN1_Handler.Init.TXFP=DISABLE;      //优先级由报文标识符决定

    if(HAL_CAN_Init(&CAN1_Handler)!=HAL_OK) return 1;     //初始化 CAN1

    CAN1_FilerConf.FilterIdHigh=0X0000;        //32 位 ID
    CAN1_FilerConf.FilterIdLow=0X0000;
    CAN1_FilerConf.FilterMaskIdHigh=0X0000; //32 位 MASK
    CAN1_FilerConf.FilterMaskIdLow=0X0000;
    CAN1_FilerConf.FilterFIFOAssignment=CAN_FILTER_FIFO0;//过滤器 0 关联到 FIFO0
    CAN1_FilerConf.FilterNumber=0;                //过滤器 0
    CAN1_FilerConf.FilterMode=CAN_FILTERMODE_IDMASK;
    CAN1_FilerConf.FilterScale=CAN_FILTERSCALE_32BIT;
    CAN1_FilerConf.FilterActivation=ENABLE; //激活滤波器 0
    CAN1_FilerConf.BankNumber=14;
```

```
    if(HAL_CAN_ConfigFilter(&CAN1_Handler,&CAN1_FilerConf)!=HAL_OK) return 2;
                                                        //滤波器初始化
    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_11|GPIO_PIN_12;       //PA11,12
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;              //推挽复用
    GPIO_Initure.Pull=GPIO_PULLUP;                  //上拉
    GPIO_Initure.Speed=GPIO_SPEED_FAST;             //快速
    GPIO_Initure.Alternate=GPIO_AF9_CAN1;           //复用为 CAN1
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);             //初始化

#if CAN1_RX0_INT_ENABLE
     __HAL_CAN_ENABLE_IT(&CAN1_Handler,CAN_IT_FMP0);//FIFO0 挂起中断允许.
    HAL_NVIC_SetPriority(CAN1_RX0_IRQn,1,2);        //抢占优先级 1，子优先级 2
    HAL_NVIC_EnableIRQ(CAN1_RX0_IRQn);              //使能中断
#endif
}

#if CAN1_RX0_INT_ENABLE         //使能 RX0 中断
//CAN 中断服务函数
void CAN1_RX0_IRQHandler(void)
{
    HAL_CAN_IRQHandler(&CAN1_Handler);//此函数会调用 CAN_Receive_IT()接收数据
}

//CAN 中断处理过程
//此函数会被 CAN_Receive_IT()调用
//hcan:CAN 句柄
void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef* hcan)
{
    int i=0;
```

```
        //CAN_Receive_IT()函数会关闭 FIFO0 消息挂号中断，因此我们需要重新打开
        __HAL_CAN_ENABLE_IT(&CAN1_Handler,CAN_IT_FMP0);
                                                    //重新开启 FIF00 消息挂号中断
        printf("id:%d\r\n",CAN1_Handler.pRxMsg->StdId);
        printf("ide:%d\r\n",CAN1_Handler.pRxMsg->IDE);
        printf("rtr:%d\r\n",CAN1_Handler.pRxMsg->RTR);
        printf("len:%d\r\n",CAN1_Handler.pRxMsg->DLC);
        for(i=0;i<8;i++)
        printf("rxbuf[%d]:%d\r\n",i,CAN1_Handler.pRxMsg->Data[i]);
}
#endif

//can 发送一组数据(固定格式:ID 为 0X12,标准帧,数据帧)
//len:数据长度(最大为 8)
//msg:数据指针,最大为 8 个字节.
//返回值:0,成功;
//          其他,失败;
u8 CAN1_Send_Msg(u8* msg,u8 len)
{
        u16 i=0;
        CAN1_Handler.pTxMsg->StdId=0X12;                //标准标识符
        CAN1_Handler.pTxMsg->ExtId=0x12;                //扩展标识符(29 位)
        CAN1_Handler.pTxMsg->IDE=CAN_ID_STD;            //使用标准帧
        CAN1_Handler.pTxMsg->RTR=CAN_RTR_DATA;          //数据帧
        CAN1_Handler.pTxMsg->DLC=len;
        for(i=0;i<len;i++)
        CAN1_Handler.pTxMsg->Data[i]=msg[i];
        if(HAL_CAN_Transmit(&CAN1_Handler,10)!=HAL_OK) return 1; //发送
        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;
        if(HAL_CAN_Receive(&CAN1_Handler,CAN_FIFO0,0)!=HAL_OK) return 0;
                                                    //接收数据，超时时间设置为 0
        for(i=0;i<CAN1_Handler.pRxMsg->DLC;i++)
        buf[i]=CAN1_Handler.pRxMsg->Data[i];
        return CAN1_Handler.pRxMsg->DLC;
```

```
}
```

此部分代码总共 5 个函数，首先是：CAN_Mode_Init 函数。该函数用于 CAN 的初始化，该函数带有 5 个参数，可以设置 CAN 通信的波特率和工作模式等，在该函数中，我们就是按 34.1 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，HAL_CAN_MspInit 函数。该函数为 CAN 的 MSP 初始化回调函数。

第三个函数，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。

第四个函数，Can_Receive_Msg 函数。用来接受数据并且将接受到的数据存放到 buf 中。

can.c 里面，还包含了中断接收的配置，通过 can.h 的 CAN1_RX0_INT_ENABLE 宏定义，来配置是否使能中断接收，本章我们不开启中断接收的。其他函数我们就不一一介绍了，都比较简单，大家自行理解即可。

can.h 头文件中，CAN1_RX0_INT_ENABLE 用于设置是否使能中断接收，本章我们不用中断接收，故设置为 0。最后我们看看主函数，代码如下：

```
int main(void)
{
    u8 key,i=0,t=0,cnt=0,canbuf[8],res;
    u8 mode=1;

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
   …//此处省略部分初始化代码
    CAN1_Mode_Init(CAN_SJW_1TQ,CAN_BS2_6TQ,CAN_BS1_8TQ,6,
                   CAN_MODE_LOOPBACK); //CAN 初始化,波特率 500Kbps
    …//此处省略部分代码
    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_6TQ,CAN_BS1_8TQ,6,
                  CAN_MODE_NORMAL);    //回环模式,波特率 500Kbps
        elseif(mode==1)
          CAN1_Mode_Init(CAN_SJW_1TQ,CAN_BS2_6TQ,CAN_BS1_8TQ,6,
                   CAN_MODE_LOOPBACK);  //回环模式,波特率 500Kbps
        POINT_COLOR=RED;//设置字体为红色
        if(mode==0)//普通模式，需要 2 个开发板
        {
            LCD_ShowString(30,130,200,16,16,"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)//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_Toggle;//提示系统正在运行
        t=0;
        cnt++;
        LCD_ShowxNum(30+48,170,cnt,3,16,0X80);     //显示数据
    }
```

```
        }
    }
```

此部分代码，我们主要关注下 CAN1_Mode_Init 初始化代码：

```
CAN1_Mode_Init(CAN_SJW_1TQ,CAN_BS2_6TQ,CAN_BS1_11TQ,6,
                    CAN_MODE_LOOPBACK);//CAN 初始化环回模式,波特率 500Kbps
```

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。计算方式请参考 CAN1_Mode_Init 函数注释。通过 KEY_UP 按键，可以随时切换 CAN 工作模式。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。

35.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到如图 35.4.1 所示：

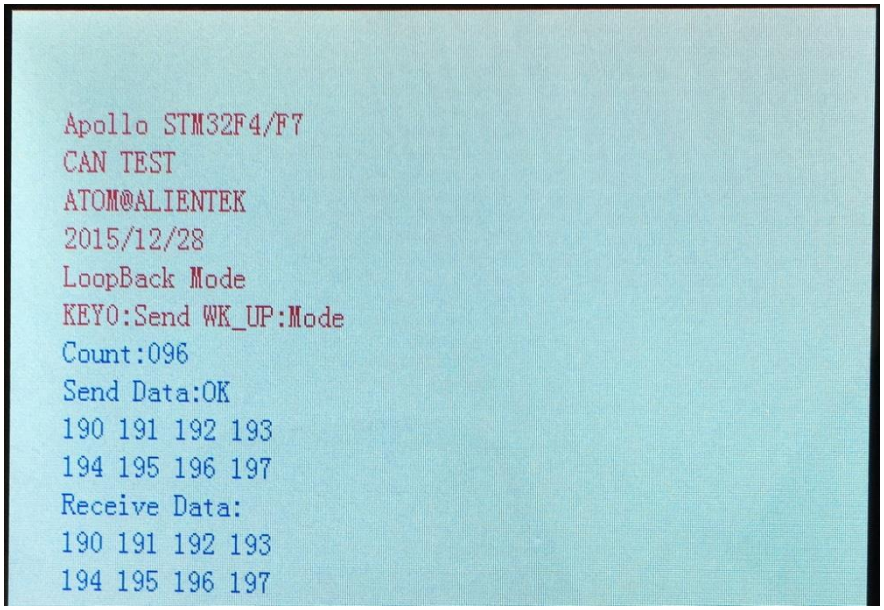


图 35.4.1 程序运行效果图

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 35.4.2 和图 35.4.3 所示：

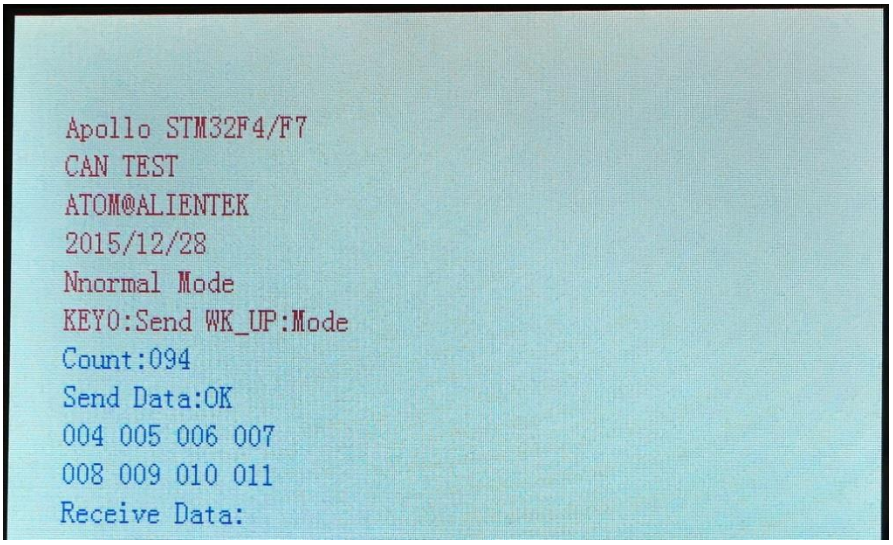


图 35.4.2 CAN 普通模式发送数据

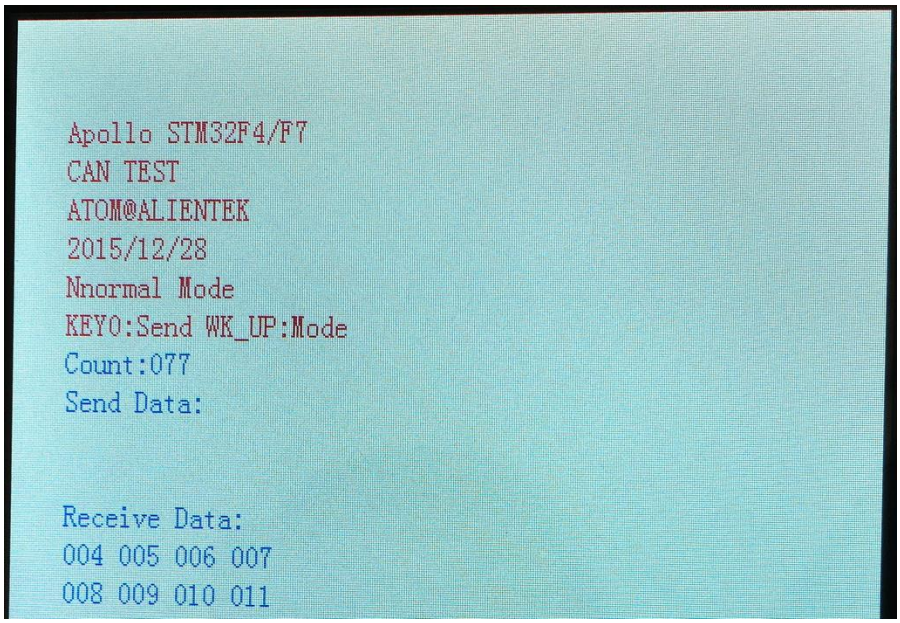


图 35.4.3 CAN 普通模式接收数据

图 35.4.2 来自开发板 A，发送了 8 个数据，图 35.4.3 来自开发板 B，收到了来自开发板 A 的 8 个数据。

第三十六章 触摸屏实验

本章，我们将介绍如何使用 STM32F767 来驱动触摸屏，ALIENTEK 阿波罗 STM32F767 开发板本身并没有触摸屏控制器，但是它支持触摸屏，可以通过外接带触摸屏的 LCD 模块（比如 ALIENTEK LCD 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 LCD 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：

36.1 电阻与电容触摸屏简介
36.2 硬件设计
36.3 软件设计
36.4 下载验证

36.1 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。

36.1.1 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 LCD 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 LCD 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 LCD 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。

36.1.2 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 LCD 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 36.1.2.1 所示：

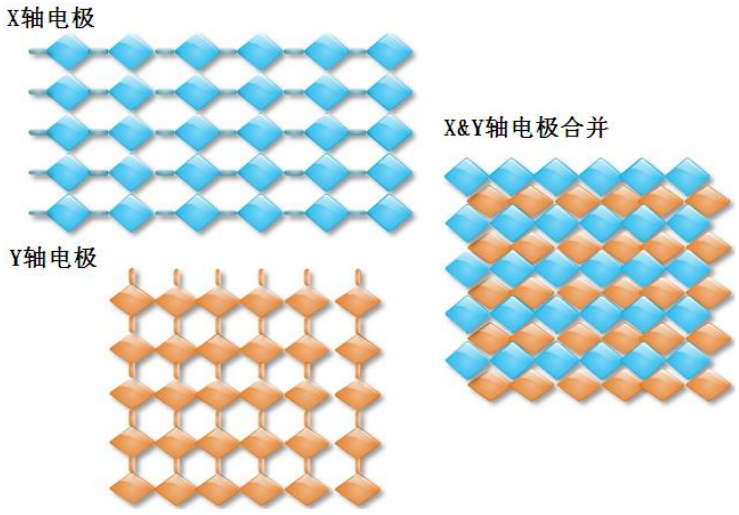


图 36.1.2.1 投射式电容屏电极矩阵示意图

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’ LCD 模块的电容触摸屏，使用 FT5206/FT5426 做为驱动 IC，采用的是 15*28 的驱动结构（15 个感应通道，28 个驱动通道）。ALIENTEK 4.3’ LCD 模块则使用 GT9147/OTT2001A 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，本例程除 CPLD 方案的 V1 版本 7 寸屏模块不支持外，其他所有 ALIENTEK 的 LCD 模块都支持，电容触摸驱动 IC，这里只介绍 GT9147 的驱动，OTT2001A、FT5206 和 FT5426 的驱动同 GT9147 类似，大家可以参考着学习即可。

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。

GT9147 与 MCU 连接是通过 4 根线：SDA、SCL、RST 和 INT。其中：SDA 和 SCL 是 IIC 通信用的，RST 是复位脚（低电平有效），INT 是中断输出信号，关于 IIC 我们就不详细介绍了，请参考第二十九章。

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 36.1.2.1 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 36.1.2.1 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。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 36.1.2.2 所示：

寄存器	bit7~0	寄存器	bit7~0
0X8150	触点 1 x 坐标低 8 位	0X8151	触点 1 x 坐标低高位
0X8152	触点 1 y 坐标低 8 位	0X8153	触点 1 y 坐标低高位
0X8154	触点 1 触摸尺寸低 8 位	0X8155	触点 1 触摸尺寸高 8 位

表 36.1.2.2 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。GT9147 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。

特别说明：FT5206 和 FT5426 的驱动代码完全一模一样，他们只是版本号读取的时候稍有

差异，读坐标数据和配置等操作完全是一模一样的。所以，这两个电容屏驱动 IC，可以共用一个.c 文件（ft5206.c）。电容式触摸屏部分，就介绍到这里。

36.2 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（不支持采用 CPLD 驱动的 7 寸 V1 屏），则执行 FT5206 的初始化代码（兼容 FT5426），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） LCD 模块（带电阻/电容式触摸屏）

4） 24C02

所有这些资源与 STM32F767 的连接图，在前面都已经介绍了，这里我们只针对 LCD 模块与 STM32F767 的连接端口再说明一下，LCD 模块的触摸屏（电阻触摸屏）总共有 5 跟线与 STM32F767 连接，连接电路图如图 36.2.1 所示：

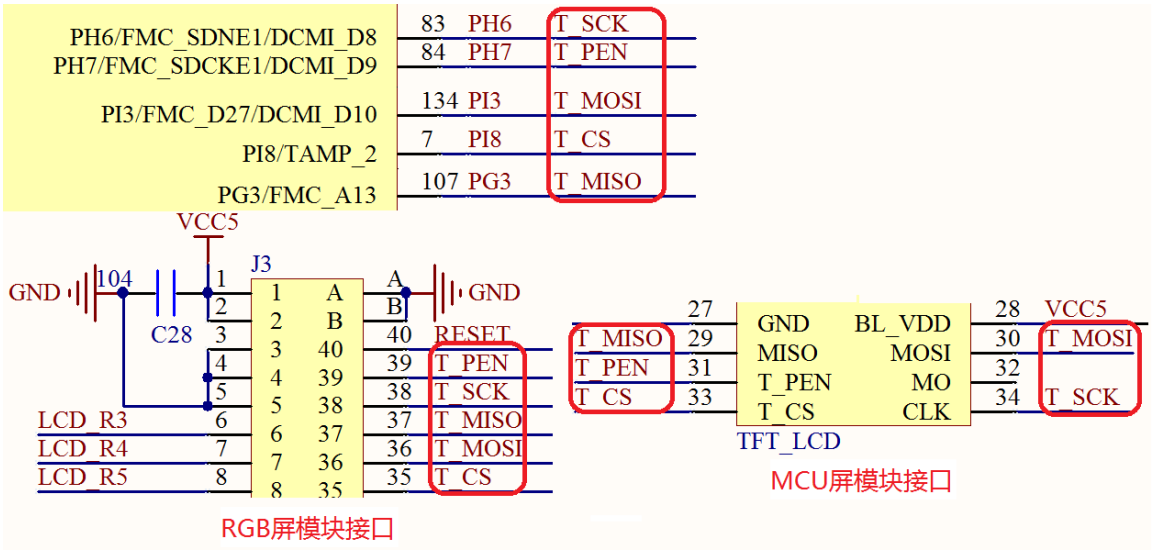


图 36.2.1 触摸屏与 STM32F767 的连接图

从图中可以看出，T_MOSI、T_MISO、T_SCK、T_CS 和 T_PEN 分别连接在 STM32F767 的：PI3、PG3、PH6、PI8 和 PH7 上。另外，阿波罗 STM32F767 开发板有 2 种屏幕接口：RGB 屏和 MCU 屏，他们共用触摸屏接口（需分时复用）。

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 和 FT5426 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数据，对于 OTT2001A/FT5206/FT5426 没有用到中断信号（CT_INT），所以同 STM32F767 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。

36.3 软件设计

打开本章实验工程目录可以看到，我们在 HARDWARE 文件夹下新建了一个 TOUCH 文件夹，然后新建了 touch.c、touch.h、ctiic.c、ctiic.h、ott2001a.c、ott2001a.h、gt9147.c、gt9147.h、ft5206.c 和 ft5206.h 等十个文件用来存放触摸屏相关的代码。同时引入这些源文件到工程 HARDWARE 分组之下，并将 TOUCH 文件夹加入头文件包含路径。其中，touch.c 和 touch.h 是电阻触摸屏部分的代码，顺带兼电容触摸屏的管理控制，其他则是电容触摸屏部分的代码。

打开 touch.c 文件，里面主要是与触摸屏相关的代码（主要是电阻触摸屏的代码），这里我们也不全部贴出来了，仅介绍几个重要的函数。

首先我们要介绍的是 TP_Read_XY2 这个函数，该函数专门用于从电阻式触摸屏控制 IC 读取坐标的值（0~4095），TP_Read_XY2 的代码如下：

```
//连续 2 次读取触摸屏 IC,且这两次的偏差不能超过
//ERR_RANGE,满足条件,则认为读数正确,否则读数错误.
//该函数能大大提高准确度
//x,y:读取到的坐标值
//返回值:0,失败;1,成功。
#define ERR_RANGE 50 //误差范围
u8 TP_Read_XY2(u16 *x,u16 *y)
{
    u16 x1,y1;
    u16 x2,y2;
    u8 flag;
    flag=TP_Read_XY(&x1,&y1);
    if(flag==0)return(0);
    flag=TP_Read_XY(&x2,&y2);
    if(flag==0)return(0);
    if(((x2<=x1&&x1<x2+ERR_RANGE)||(x1<=x2&&x2<x1+ERR_RANGE))
                                                    //前后两次采样在+-50 内
    &&((y2<=y1&&y1<y2+ERR_RANGE)||(y1<=y2&&y2<y1+ERR_RANGE)))
    {
        *x=(x1+x2)/2;
        *y=(y1+y2)/2;
        return 1;
    }else return 0;
}
```

该函数采用了一个非常好的办法来读取屏幕坐标值，就是连续读两次，两次读取的值之差不能超过一个特定的值（ERR_RANGE），通过这种方式，我们可以大大提高触摸屏的准确度。另

外该函数调用的 TP_Read_XY 函数，用于单次读取坐标值。TP_Read_XY 也采用了一些软件滤波算法，具体见光盘的源码。接下来，我们介绍另外一个函数 TP_Adjust，该函数源码如下：

```
//触摸屏校准代码
//得到四个校准参数
void TP_Adjust(void)
{
    u16 pos_temp[4][2];//坐标缓存值
    u8  cnt=0; 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)
{
    GPIO_InitTypeDef GPIO_Initure;
    if(lcddev.id==0X5510||lcddev.id==0X4342)
                                //4.3 800*40MCU 电容触摸屏或者 4.3 寸 480*272 RGB 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||lcddev.id==0X7084||lcddev.id==0X7016)
                                //SSD1963 7 寸屏或者  7 寸 800*480/1024*600 RGB 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
    {

     __HAL_RCC_GPIOH_CLK_ENABLE();  //开启 GPIOH 时钟
     __HAL_RCC_GPIOI_CLK_ENABLE();   //开启 GPIOI 时钟
     __HAL_RCC_GPIOG_CLK_ENABLE();  //开启 GPIOG 时钟

     //PH6
     GPIO_Initure.Pin=GPIO_PIN_6;                    //PH6
     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(GPIOH,&GPIO_Initure);        //初始化

        //PI3,8
        GPIO_Initure.Pin=GPIO_PIN_3|GPIO_PIN_8; //PI3,8
        HAL_GPIO_Init(GPIOI,&GPIO_Initure);        //初始化

        //PH7
        GPIO_Initure.Pin=GPIO_PIN_7;               //PH7
        GPIO_Initure.Mode=GPIO_MODE_INPUT;     //输入
        HAL_GPIO_Init(GPIOH,&GPIO_Initure);        //初始化

        //PG3
        GPIO_Initure.Pin=GPIO_PIN_3;               //PG3
        HAL_GPIO_Init(GPIOG,&GPIO_Initure);        //初始化

            TP_Read_XY(&tp_dev.x[0],&tp_dev.y[0]);//第一次读取初始化
            AT24CXX_Init();          //初始化 24CXX
            if(TP_Get_Adjdata())return 0;//已经校准
            else                     //未校准?
            {
                LCD_Clear(WHITE);//清屏
                TP_Adjust();         //屏幕校准
                TP_Save_Adjdata();
            }
            TP_Get_Adjdata();
        }
        return 1;
    }
```

该函数比较简单，重点说一下：tp_dev.scan，这个结构体函数指针，默认是指向 TP_Scan 的，如果是电阻屏则用默认的即可，如果是电容屏，则指向新的扫描函数 GT9147_Scan、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，这几个函数在后续会介绍。

其他的函数我们这里就不多介绍了，接下来打开 touch.h 文件，代码如下：

```
#define TP_PRES_DOWN 0x80   //触屏被按下
#define TP_CATH_PRES 0x40   //有按键按下了
#define CT_MAX_TOUCH   5      //电容屏支持的点数,固定为 5 点

//触摸屏控制器
typedef struct
{
    u8 (*init)(void);                 //初始化触摸屏控制器
    u8 (*scan)(u8);                   //扫描触摸屏.0,屏幕扫描;1,物理坐标;
    void (*adjust)(void);             //触摸屏校准
```

```
    u16 x[CT_MAX_TOUCH];        //当前坐标
    u16 y[CT_MAX_TOUCH];  //电容屏有最多 5 组坐标,电阻屏则用 x[0],y[0]代表
                          //此次扫描时,触屏的坐标,用 x[4],y[4]存储第一次按下时的坐标.
    u8    sta;            //笔的状态
                          //b7:按下 1/松开 0;
                          //b6:0,没有按键按下;1,有按键按下.
                          //b5:保留
                          //b4~b0:电容触摸屏按下的点数(0,表示未按下,1 表示按下)
//////////////////////触摸屏校准参数(电容屏不需要校准)//////////////////////

    float xfac;
    float yfac;
    short xoff;
    short yoff;
//新增的参数,当触摸屏的左右上下完全颠倒时需要用到.
//b0:0,竖屏(适合左右为 X 坐标,上下为 Y 坐标的 TP)
//    1,横屏(适合左右为 Y 坐标,上下为 X 坐标的 TP)
//b1~6:保留.
//b7:0,电阻屏
//    1,电容屏
    u8 touchtype;
}_m_tp_dev;

extern _m_tp_dev tp_dev;          //触屏控制器在 touch.c 里面定义

//电阻屏芯片连接引脚
//电阻屏芯片连接引脚
#define PEN             HAL_GPIO_ReadPin(GPIOH,GPIO_PIN_7) //T_PEN
#define DOUT            HAL_GPIO_ReadPin(GPIOG,GPIO_PIN_3) //T_MISO
#define TDIN(n)         (n?HAL_GPIO_WritePin(GPIOI,GPIO_PIN_3,GPIO_PIN_SET): \
        HAL_GPIO_WritePin(GPIOI,GPIO_PIN_3,GPIO_PIN_RESET))//T_MOSI
#define TCLK(n)         (n?HAL_GPIO_WritePin(GPIOH,GPIO_PIN_6,GPIO_PIN_SET): \
        HAL_GPIO_WritePin(GPIOH,GPIO_PIN_6,GPIO_PIN_RESET))//T_SCK
#define TCS(n)          (n?HAL_GPIO_WritePin(GPIOI,GPIO_PIN_8,GPIO_PIN_SET): \
        HAL_GPIO_WritePin(GPIOI,GPIO_PIN_8,GPIO_PIN_RESET))//T_CS
//电阻屏函数
void TP_Write_Byte(u8 num);                         //向控制芯片写入一个数据
……//省略部分函数定义
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，与第二十九章的 myiic.c 和 myiic.h 基本一样，这里就不单独介绍了。接下来看看文件 gt9147.c 代码如下：

```
//GT9147 配置参数表
//第一个字节为版本号(0X60),必须保证新的版本号大于等于 GT9147 内部
//flash 原有版本号,才会更新配置.
const u8 GT9147_CFG_TBL[]=
{
    0X60,0XE0,0X01,0X20,0X03,0X05,0X35,0X00,0X02,0X08,
    ……//省略部分代码
    0XFF,0XFF,0XFF,0XFF,
};
//发送 GT9147 配置参数
//mode:0,参数不保存到 flash
//     1,参数保存到 flash
u8 GT9147_Send_Cfg(u8 mode)
{
    u8 buf[2];
    u8 i=0;
    buf[0]=0;
    buf[1]=mode; //是否写入到 GT9147 FLASH?  即是否掉电保存
    for(i=0;i<sizeof(GT9147_CFG_TBL);i++)buf[0]+=GT9147_CFG_TBL[i];//计算校验和
    buf[0]=(~buf[0])+1;
    GT9147_WR_Reg(GT_CFGS_REG,(u8*)GT9147_CFG_TBL,sizeof(GT9147_CFG_TBL)
                  );//发送寄存器配置
    GT9147_WR_Reg(GT_CHECK_REG,buf,2);//写入校验和,和配置更新标记
    return 0;
}
//向 GT9147 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 GT9147_WR_Reg(u16 reg,u8 *buf,u8 len)
{
    u8 i;
    u8 ret=0;
    CT_IIC_Start();
    CT_IIC_Send_Byte(GT_CMD_WR);   //发送写命令
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg>>8);      //发送高 8 位地址
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg&0XFF);    //发送低 8 位地址
```

```
    CT_IIC_Wait_Ack();
    for(i=0;i<len;i++)
    {
        CT_IIC_Send_Byte(buf[i]);        //发数据
        ret=CT_IIC_Wait_Ack();
        if(ret)break;
    }
    CT_IIC_Stop();                       //产生一个停止条件
    return ret;
}
//从 GT9147 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 GT9147_RD_Reg(u16 reg,u8 *buf,u8 len)
{
    u8 i;
    CT_IIC_Start();
    CT_IIC_Send_Byte(GT_CMD_WR);   //发送写命令
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg>>8);            //发送高 8 位地址
    CT_IIC_Wait_Ack();
    CT_IIC_Send_Byte(reg&0XFF);          //发送低 8 位地址
    CT_IIC_Wait_Ack();
    CT_IIC_Start();
    CT_IIC_Send_Byte(GT_CMD_RD);   //发送读命令
    CT_IIC_Wait_Ack();
    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
}
//初始化 GT9147 触摸屏
//返回值:0,初始化成功;1,初始化失败
u8 GT9147_Init(void)
{
    u8 temp[5];
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_GPIOH_CLK_ENABLE();    //开启 GPIOH 时钟
    __HAL_RCC_GPIOI_CLK_ENABLE();    //开启 GPIOI 时钟
```

```
    //PH7
    GPIO_Initure.Pin=GPIO_PIN_7;                //PH7
    GPIO_Initure.Mode=GPIO_MODE_INPUT;          //输入
    GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    HAL_GPIO_Init(GPIOH,&GPIO_Initure);         //初始化

    //PI8
    GPIO_Initure.Pin=GPIO_PIN_8;                    //PI8
    GPIO_Initure.Mode=GPIO_MODE_OUTPUT_PP;  //推挽输出
    HAL_GPIO_Init(GPIOI,&GPIO_Initure);             //初始化
    CT_IIC_Init();              //初始化电容屏的 I2C 总线
    GT_RST(0);                  //复位
    delay_ms(10);
    GT_RST(1);                  //释放复位
    delay_ms(10);
    GPIO_Initure.Pin=GPIO_PIN_7;                //PH7
    GPIO_Initure.Mode=GPIO_MODE_INPUT;          //输入
    GPIO_Initure.Pull=GPIO_NOPULL;              //不带上下拉，浮空输入
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    HAL_GPIO_Init(GPIOH,&GPIO_Initure);         //初始化
    delay_ms(100);
    GT9147_RD_Reg(GT_PID_REG,temp,4);//读取产品 ID
    temp[4]=0;
    printf("CTP ID:%s\r\n",temp);       //打印 ID
    if(strcmp((char*)temp,"9147")==0)//ID==9147
    {
        temp[0]=0X02;
        GT9147_WR_Reg(GT_CTRL_REG,temp,1);//软复位 GT9147
        GT9147_RD_Reg(GT_CFGS_REG,temp,1);//读取 GT_CFGS_REG 寄存器
        if(temp[0]<0X60)//默认版本比较低,需要更新 flash 配置
        {
            printf("Default Ver:%d\r\n",temp[0]);
            if(lcddev.id==0X5510)GT9147_Send_Cfg(1);//更新并保存配置
        }
        delay_ms(10);
        temp[0]=0X00;
        GT9147_WR_Reg(GT_CTRL_REG,temp,1);//结束复位
        return 0;
    }
    return 1;
}
Const u16 GT9147_TPX_TBL[5]={GT_TP1_REG,GT_TP2_REG,GT_TP3_REG,
```

```
                                        GT_TP4_REG,GT_TP5_REG};
//扫描触摸屏(采用查询方式)
//mode:0,正常扫描.
//返回值:当前触屏状态.
//0,触屏无触摸;1,触屏有触摸
u8 GT9147_Scan(u8 mode)
{
    u8 buf[4],i=0, res=0, temp, tempsta;
    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 10 次 CTP_Scan 函数才检测 1 次,从而节省 CPU 使用率
    {
        GT9147_RD_Reg(GT_GSTID_REG,&mode,1); //读取触摸点的状态
        if(mode&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(lcddev.id==0X5510)    //4.3 寸 800*480 MCU 屏
                    {
                        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];
                        }
                    }else if(lcddev.id==0X4342) //4.3 寸 480*272 RGB 屏
                    {
```

```
                    if(tp_dev.touchtype&0X01)//横屏
                    {
                        tp_dev.x[i]=(((u16)buf[1]<<8)+buf[0]);
                        tp_dev.y[i]=(((u16)buf[3]<<8)+buf[2]);
                    }else
                    {
                        tp_dev.y[i]=((u16)buf[1]<<8)+buf[0];
                        tp_dev.x[i]=272-(((u16)buf[3]<<8)+buf[2]);
                    }                           }
                }
            }
            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) //其他点有数据,复制第二个触点的数据到第一个触点
                {
                    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;
```

```
}
```

这里总共有 5 个函数：GT9147_Send_Cfg 函数用于发送 GT9147 的配置参数，在固件更新的时候，需要用到；GT9147_WR_Reg 和 GT9147_RD_Reg 用于读写 GT9147 的寄存器；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；

最后，重点介绍：GT9147_Scan 函数，该函数用于扫描电容触摸屏是否有按键下，由我们是采用查询的方式读取数据，所以这里使了一个静态变量（static）来提高效率当无触摸时候尽减少对 CPU 的占用，当有触摸时候又保证能迅速检测到。读取数据时，先读取状态寄存器（GT_GSTID_REG）的值，从而判断触摸点的个数（最多 5 个），然后依次读取各触摸点的坐标数据，在读取到数据后，还需要根据屏幕的分辨率和横竖屏状态进行坐标变换。另外，在遇到非法数据的时候，需要对非法数据进行处理，以免干扰程序的正常运行。

对于文件 ott2001a.c、ft5206.c、ott2001a.h 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。需要注意的是：ft5206.c 同时支持 FT5206 和 FT5426 这两颗触摸 IC，他们共用一个代码。

最后我们打开 main.c，里面内容比较多，这里我们着重介绍三个主要函数：

```
//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();//清除
                else TP_Draw_Big_Point(tp_dev.x[0],tp_dev.y[0],RED);    //画图

            }
        }else delay_ms(10);      //没有按键按下的时候
        if(key==KEY0_PRES)  //KEY0 按下,则执行校准程序
        {
            LCD_Clear(WHITE);    //清屏
            TP_Adjust();              //屏幕校准
            TP_Save_Adjdata();
            Load_Drow_Dialog();
```

```
            }
            i++;
            if(i%20==0)LED0_Toggle;
        }
    }
    //电容触摸屏测试函数
    void ctp_test(void)
    {
        u8 t=0;
        u8 i=0;
        u16 lastpos[5][2];          //最后一次的数据
        while(1)
        {
            tp_dev.scan(0);
            for(t=0;t<5;t++)
            {
                if((tp_dev.sta)&(1<<t))
                {
                    //printf("X 坐标:%d,Y 坐标:%d\r\n",tp_dev.x[0],tp_dev.y[0]);
                    if(tp_dev.x[t]<lcddev.width&&tp_dev.y[t]<lcddev.height)
                    {
                        if(lastpos[t][0]==0XFFFF)
                        {
                            lastpos[t][0] = tp_dev.x[t];
                            lastpos[t][1] = tp_dev.y[t];
                        }

                        lcd_draw_bline(lastpos[t][0],lastpos[t][1],tp_dev.x[t],tp_dev.y[t],2,
                                                                    POINT_COLOR_TBL[t]);//画线
                        lastpos[t][0]=tp_dev.x[t];
                        lastpos[t][1]=tp_dev.y[t];
                        if(tp_dev.x[t]>(lcddev.width-24)&&tp_dev.y[t]<20)
                        {
                            Load_Drow_Dialog();//清除
                        }
                    }
                }else lastpos[t][0]=0XFFFF;
            }

            delay_ms(5);i++;
            if(i%20==0)LED0_Toggle;
        }
    }
```

```
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    …//此处省略部分初始化代码
    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,"Apollo STM32F4/F7");
    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,"2016/7/12");
    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。

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 阿波罗 STM32 开发板上，电阻触摸屏测试如图 36.4.1 所示界面：

图 36.4.1 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 36.4.2 所示：

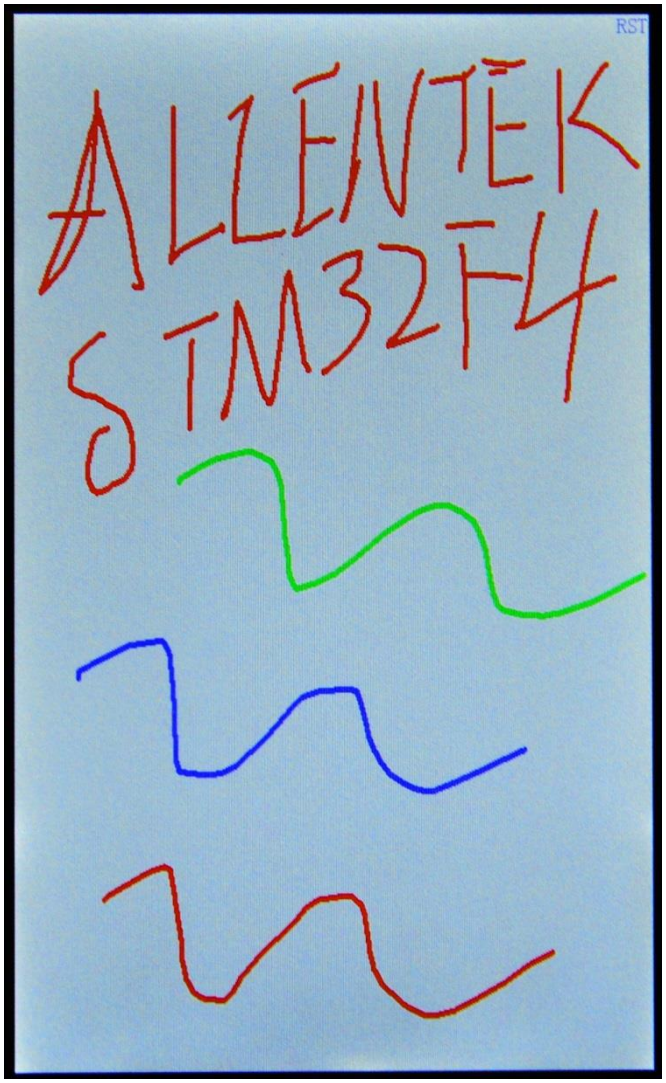


图 36.4.2 电容触摸屏测试界面

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：老款的 7 寸电容触摸屏模块（CPLD 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。

第三十七章　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 阿波罗 STM32F767 开发板标配了红外接收头和一个很小巧的红外遥控器。在本章中，我们将利用 STM32F767 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 LCD 模块上显示出来。本章分为如下几个部分：

37.1 红外遥控简介
37.2 硬件设计
37.3 软件设计
37.4 下载验证

37.1 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 阿波罗 STM32F767 开发板配套的遥控器使用的是 NEC 协议，其特征如下：

1、8 位地址和 8 位指令长度；
2、地址和命令 2 次传输（确保可靠性）
3、PWM 脉冲位置调制，以发射红外载波的占空比代表“0”和“1”；
4、载波频率为 38Khz；
5、位时间为 1.125ms 或 2.25ms；

NEC 码的位定义：一个脉冲对应 560us 的连续载波，一个逻辑 1 传输需要 2.25ms（560us 脉冲+1680us 低电平），一个逻辑 0 的传输需要 1.125ms（560us 脉冲+560us 低电平）。而遥控接收头在收到脉冲的时候为低电平，在没有脉冲的时候为高电平，这样，我们在接收头端收到的信号为：逻辑 1 应该是 560us 低+1680us 高，逻辑 0 应该是 560us 低+560us 高。

NEC 遥控指令的数据格式为：同步码头、地址码、地址反码、控制码、控制反码。同步码由一个 9ms 的低电平和一个 4.5ms 的高电平组成，地址码、地址反码、控制码、控制反码均是 8 位数据格式。按照低位在前，高位在后的顺序发送。采用反码是为了增加传输的可靠性（可用于校验）。

我们遥控器的按键“▽”按下时，从红外接收头端收到的波形如图 37.1.1 所示：

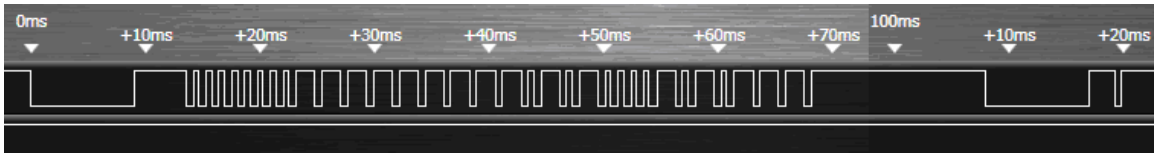


图 37.1.1 按键“▽”所对应的红外波形

从图 37.1.1 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。

第十四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，请参考第十四章的内容。

37.2 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） LCD 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 37.2.1 所示：

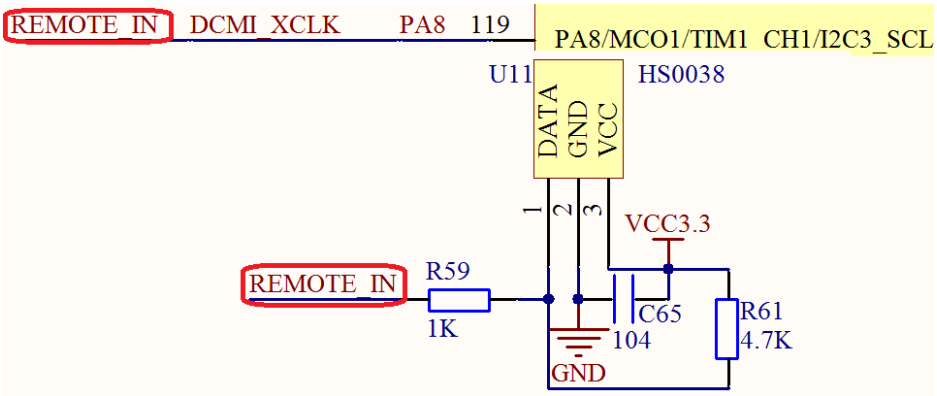


图 37.2.1 红外遥控接收头与 STM32 的连接电路图

红外遥控接收头连接在 STM32 的 PA8（TIM1_CH1）上。硬件上不需要变动，只要程序将 TIM1_CH1 设计为输入捕获，然后将收到的脉冲信号解码就可以了。不过需要注意：REMOTE_IN 和 DCMI_XCLK 共用了 PA8，所以他们不可以同时使用。

开发板配套的红外遥控器外观如图 37.2.2 所示：

图 37.2.2 红外遥控器

37.3 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 stm32f7xx_hal_tim.c 和头文件 stm32f7xx_hal_tim.h。

打开 remote.c 文件，代码如下：

```
TIM_HandleTypeDef TIM1_Handler;         //定时器 1 句柄

//红外遥控初始化
//设置 IO 以及 TIM1_CH1 的输入捕获
//TIM1 挂载 APB2 上
void Remote_Init(void)
{
    TIM_IC_InitTypeDef TIM1_CH1Config;

    TIM1_Handler.Instance=TIM1;             //通用定时器 1
    TIM1_Handler.Init.Prescaler=215;        //预分频器,1M 的计数频率,1us 加 1.
    TIM1_Handler.Init.CounterMode=TIM_COUNTERMODE_UP; //向上计数器
    TIM1_Handler.Init.Period=10000;                   //自动装载值
    TIM1_Handler.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_IC_Init(&TIM1_Handler);

    //初始化 TIM1 输入捕获参数
    TIM1_CH1Config.ICPolarity=TIM_ICPOLARITY_RISING;    //上升沿捕获
    TIM1_CH1Config.ICSelection=TIM_ICSELECTION_DIRECTTI;//映射到 TI1 上
    TIM1_CH1Config.ICPrescaler=TIM_ICPSC_DIV1;          //配置输入分频，不分频
    TIM1_CH1Config.ICFilter=0x03; //IC1F=0003 8 个定时器时钟周期滤波
    HAL_TIM_IC_ConfigChannel(&TIM1_Handler,&TIM1_CH1Config,
                             TIM_CHANNEL_1);//配置 TIM1 通道 1
    HAL_TIM_IC_Start_IT(&TIM1_Handler,TIM_CHANNEL_1); //开始捕获 TIM1 的通道 1
    __HAL_TIM_ENABLE_IT(&TIM1_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_TIM1_CLK_ENABLE();      //使能 TIM1 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();   //开启 GPIOA 时钟
```

```
    GPIO_Initure.Pin=GPIO_PIN_8;                //PA8
    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_HIGH;     //高速
    GPIO_Initure.Alternate=GPIO_AF1_TIM1;       //PA8 复用为 TIM1 通道 1
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);

    HAL_NVIC_SetPriority(TIM1_CC_IRQn,1,3);  //设置抢占优先级 1，子优先级 3
    HAL_NVIC_EnableIRQ(TIM1_CC_IRQn);       //开启 ITM1 中断

    HAL_NVIC_SetPriority(TIM1_UP_TIM10_IRQn,1,2); //设置抢占优先级 1，子优先级 2
    HAL_NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);     //开启 ITM1 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 1 更新（溢出）中断
void TIM1_UP_TIM10_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM1_Handler);//定时器共用处理函数
}

//定时器 1 输入捕获中断服务程序
void TIM1_CC_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&TIM1_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==TIM1){
        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==TIM1)
{
            if(RDATA)//上升沿捕获
            {
                TIM_RESET_CAPTUREPOLARITY(&TIM1_Handler,TIM_CHANNEL_1);
                                                        //一定要先清除原来的设置!!
                TIM_SET_CAPTUREPOLARITY(&TIM1_Handler,TIM_CHANNEL_1,
                 TIM_ICPOLARITY_FALLING);//CC1P=1  设置为下降沿捕获
                __HAL_TIM_SET_COUNTER(&TIM1_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(&TIM1_Handler,TIM_CHANNEL_1);
                                        //读取 CCR1 也可以清 CC1IF 标志位
            TIM_RESET_CAPTUREPOLARITY(&TIM1_Handler,TIM_CHANNEL_1);
                                        //一定要先清除原来的设置!!
            TIM_SET_CAPTUREPOLARITY(&TIM1_Handler,TIM_CHANNEL_1,
                                TIM_ICPOLARITY_RISING);//配置 TIM5 通道 1 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;
    }
```

该部分代码包含 7 个函数，首先是 Remote_Init 函数，该函数和 MSP 回调函数 HAL_TIM_IC_MspInit 共同完成 TIM1 的时基参数初始化，TIM1_CH1 的 1 的输入捕获配置，IO 口初始化配置，IO 口复用映射配置以及 NVIC 配置。具体内容和输入捕获实验章节基本一致，不同的是换了定时器而已。请参考输入捕获实验章节讲解。

由于 TIM1 是高级定时器，它有 2 个中断服务函数：

TIM1_UP_TIM10_IRQHandler 中断服务函数用于处理 TIM1 的溢出（更新）事件，TIM1_CC_IRQHandler 中断服务函数用于处理 TIM1 的输入捕获事件。对于我们使用 HAL 库，根据定时器中断实验讲解，会在中断服务函数中调用 HAL 库提供的公用中断处理函数 HAL_TIM_IRQHandler，该函数内部会对中断进行判断，然后调用相应的中断处理回调函数。

溢出（更新）中断回调函数为 HAL_TIM_PeriodElapsedCallback，在本例程里面，该函数用来处理 TIM1 溢出，并标用于标记键值获取完成。每一次红外按键解码，都必须通过定时器溢出事件来标记完成。输入捕获中断回调函数 HAL_TIM_IC_CaptureCallback， 在本例程里面，该函数实现对红外信号的高电平脉冲的捕获，同时根据我们之前简介的协议内容来解码。这两个中断处理回调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;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    …//此处省略部分代码
    Remote_Init();                  //初始化 红外接收

    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;
                        ..//此处省略部分代码
                        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_Toggle;
                }
        }
    }
```

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。

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 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 37.4.1 所示的内容：

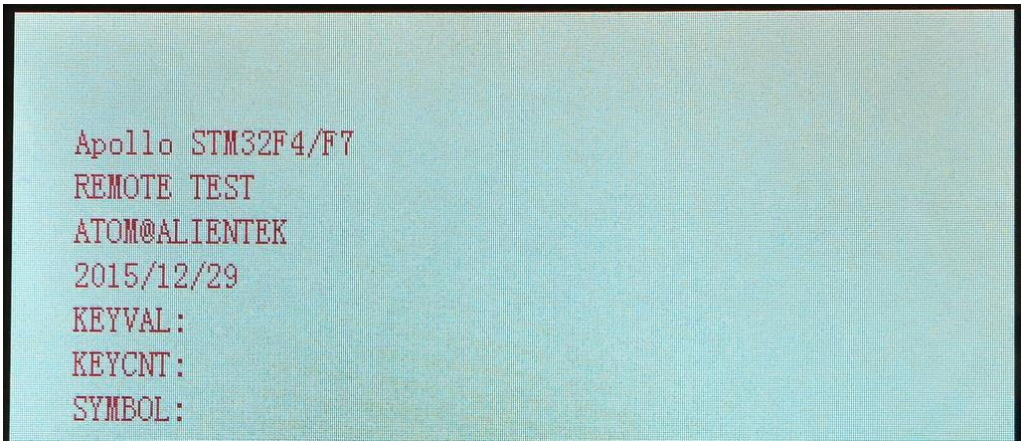


图 37.4.1 程序运行效果图

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 37.4.2 所示：

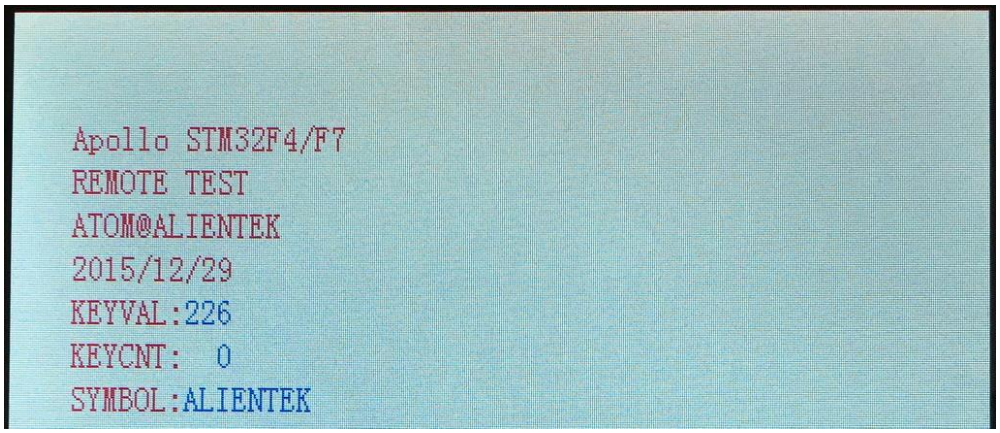


图 37.4.2 解码成功

第三十八章　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 LCD 模块上。本章分为如下几个部分：

38.1 DS18B20 简介
38.2 硬件设计
38.3 软件设计
38.4 下载验证

38.1 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 38.1.1 所示：

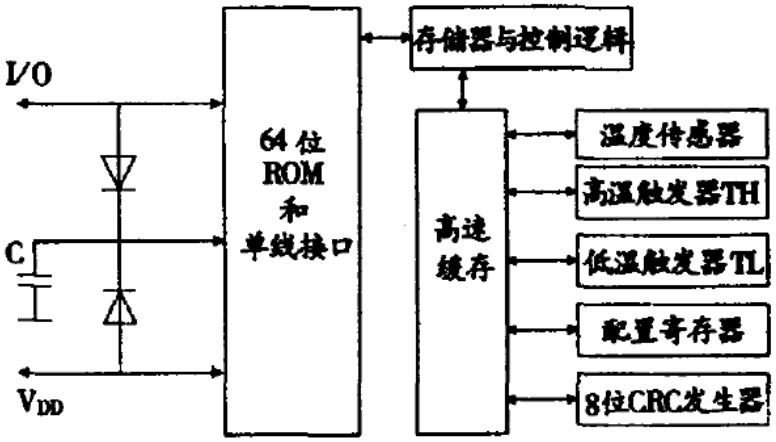


图 38.1.1 DS18B20 内部结构图

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 的数据手册。

38.2 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） LCD 模块

3） PCF8574T

4） DS18B20 温度传感器

前3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38.2.1所示：

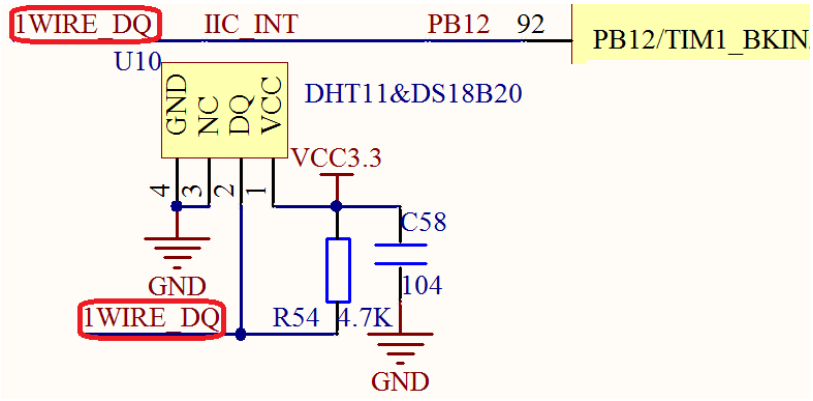


图 38.2.1 DS18B20 接口与 STM32 的连接电路图

从上图可以看出，我们使用的是 STM32 的 PB12 来连接 U10 的 DQ 引脚，图中 U10 为 DHT11（数字温湿度传感器）和 DS18B20 共用的一个接口，DHT11 我们将在下一章介绍。这里，1WIRE_DQ 和 IIC_INT（PCF8574T 用）是共用 PB12 的，所以他们不能同时使用。

注意：为了让 PCF8574T 释放 IIC_INT 脚（复位 INT），需要对 PCF8574T 进行一次读取操作，否则无法正常读取 DS18B20/DHT11!!!

DS18B20 只用到 U10 的 3 个引脚（U10 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 38.2.2 所示：

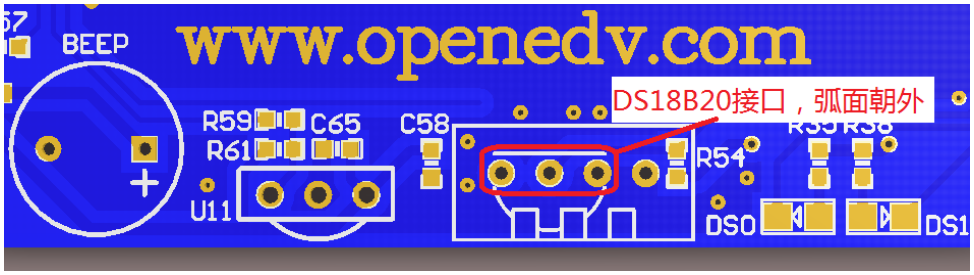


图 38.2.2 DS18B20 连接示意图

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。

38.3 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();       //设置为输出
    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();    //设置为输入
    while (DS18B20_DQ_IN&&retry<200)
    {
        retry++;
        delay_us(1);
    };
    if(retry>=200)return 1;
    else retry=0;
    while (!DS18B20_DQ_IN&&retry<240)
    {
        retry++;
        delay_us(1);
    };
    if(retry>=240)return 1;
    return 0;
}

//从 DS18B20 读取一个位
//返回值：1/0
u8 DS18B20_Read_Bit(void)
{
    u8 data;
    DS18B20_IO_OUT();   //设置为输出
    DS18B20_DQ_OUT(0);
    delay_us(2);
    DS18B20_DQ_OUT(1);
    DS18B20_IO_IN();    //设置为输入
    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();        //设置为输出
    for (j=1;j<=8;j++)
    {
        testb=dat&0x01;
        dat=dat>>1;
        if(testb)           // 写 1
        {
            DS18B20_DQ_OUT(0);
            delay_us(2);
            DS18B20_DQ_OUT(1);
            delay_us(60);
        }
        else                //写 0
        {
            DS18B20_DQ_OUT(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_GPIOB_CLK_ENABLE();              //开启 GPIOB 时钟

    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_HIGH;        //高速
    HAL_GPIO_Init(GPIOB,&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 ();                  //开始转换
    DS18B20_Rst();
    DS18B20_Check();
    DS18B20_Write_Byte(0xcc);       // skip rom
    DS18B20_Write_Byte(0xbe);       // convert
    TL=DS18B20_Read_Byte();         // LSB
    TH=DS18B20_Read_Byte();         // MSB
    if(TH>7)
    {
        TH=~TH;
        TL=~TL;
        temp=0;//温度为负
```

```
        }else temp=1;//温度为正
        tem=TH; //获得高八位
        tem<<=8;
        tem+=TL;//获得底八位
        tem=(double)tem*0.625;//转换
        if(temp)return tem; //返回温度值
        else return -tem;
    }
```

该部分代码就是根据我们前面介绍的单总线操作时序来读取 DS18B20 的温度值的，DS18B20 的温度通过 DS18B20_Get_Temp 函数读取，该函数的返回值为带符号的短整型数据，返回值的范围为-550~1250，其实就是温度值扩大了 10 倍。

接下来我们打开 ds18b20.h，可以看到跟 IIC 实验代码很类似，这里我们不做过多讲解。接下来我们看看主函数代码：

```
int main(void)
{
    u8 t=0;
    short temperature;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    …//此处省略部分初始化代码
    LCD_Init();                     //初始化 LCD
    PCF8574_Init();                 //初始化 PCF8574
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4&F7");
    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,"2016/1/16");
    PCF8574_ReadBit(BEEP_IO);       //由于DS18B20和PCF8574的中断引脚共用一个IO,
                /所以在初始化 DS18B20 之前要先读取一次 PCF8574 的任意一个 IO，使其释
                //放掉中断引脚所占用的 IO(PB12 引脚),否则初始化 DS18B20 会出问题
    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
        {
                PCF8574_ReadBit(BEEP_IO);
                    //读取一次 PCF8574 的任意一个 IO，使其释放掉 PB12 引脚，
                    //否则读取 DS18B20 可能会出问题
                 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_Toggle;
             }
        }
    }
```

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。

38.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 显示开始显示当前的温度值（假定 DS18B20 已经接上去了），如图 38.4.1 所示：

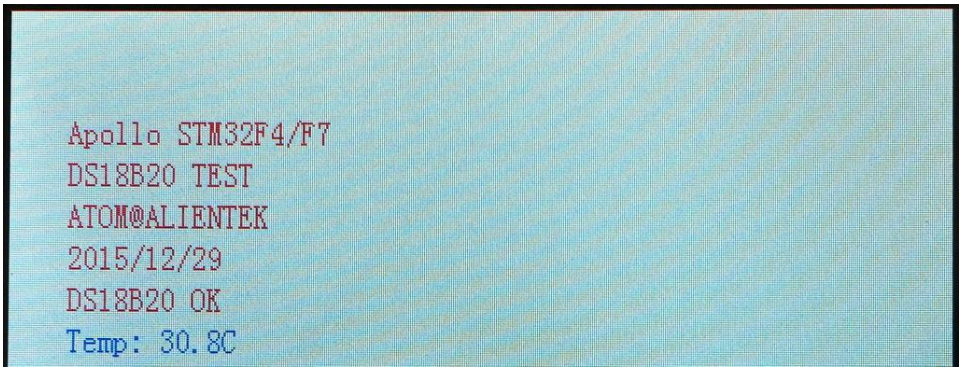


图 38.4.1 DS18B20 实验效果图

该程序还可以读取并显示负温度值的，只是由于本人在广州，是没办法看到了（除非放到冰箱），具备条件的读者可以测试一下。

第三十九章　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 STM32F767 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 LCD 模块上。本章分为如下几个部分：

39.1 DHT11 简介
39.2 硬件设计
39.3 软件设计
39.4 下载验证

39.1 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 39.1.1 所示：

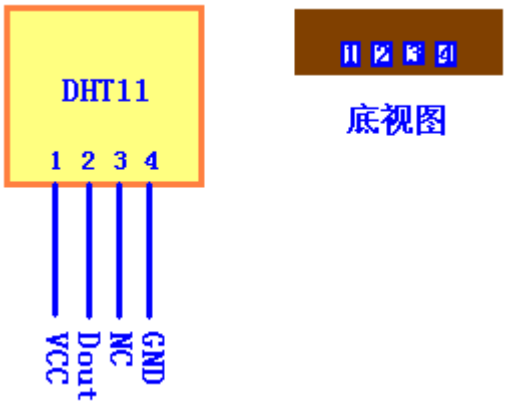


图 39.1.1 DHT11 管脚排列图

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 39.1.2 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 39.1.2 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 39.1.3 所示：

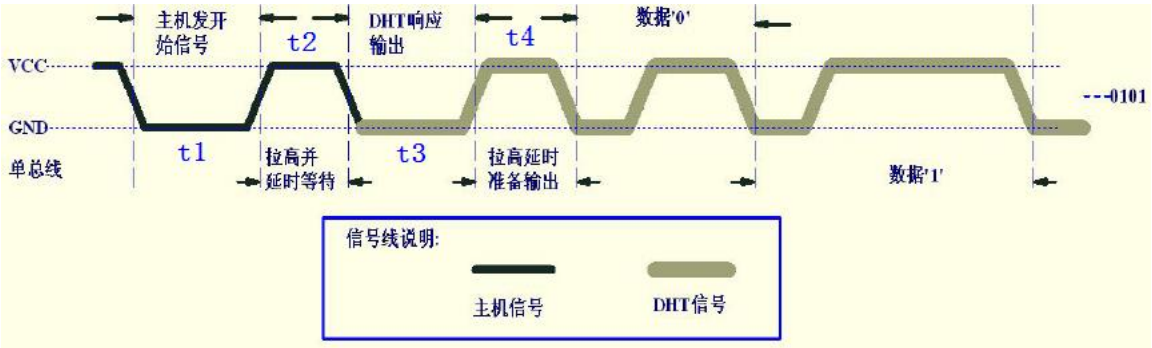


图 39.1.3 DHT11 数据发送流程

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 39.1.4 所示：

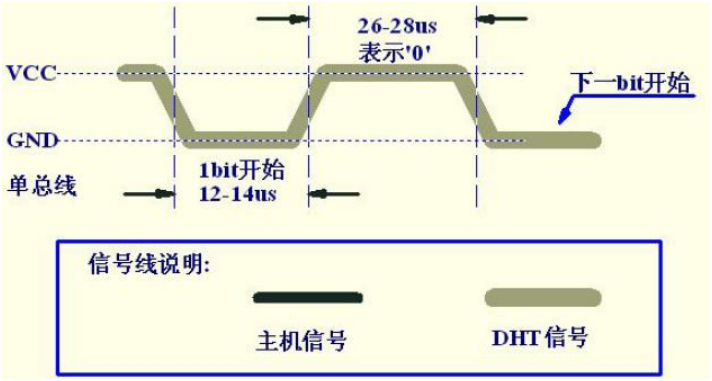


图 39.1.4 DHT11 数字‘0’时序

DHT11 输出数字‘1’的时序如图 39.1.5 所示：

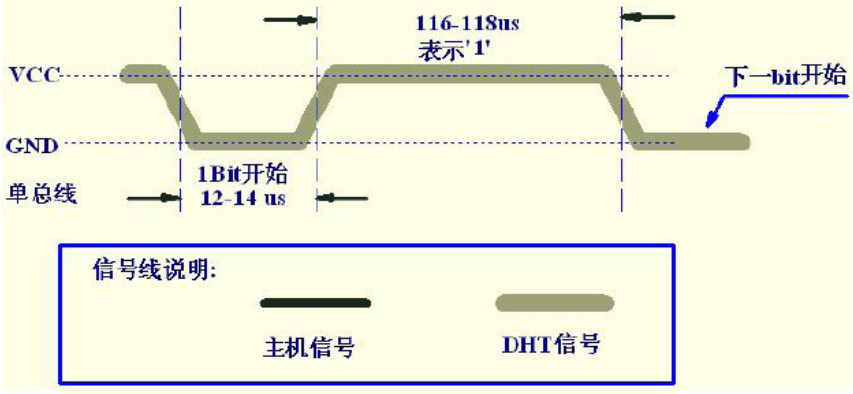


图 39.1.5 DHT11 数字‘1’时序

通过以上了解，我们就可以通过 STM32F767 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。

39.2 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） LCD 模块

3） PCF8574T

4） 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，同样，在读取 DHT11 的时候，**需要先对 PCF8574T 进行一次读取操作，以释放 IIC_INT 引脚（详见上一章节介绍）**。DHT11 有 4 引脚，需要把 U10 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 STM32F767 来读取温湿度值了。连接示意图如图 39.2.1 所示：

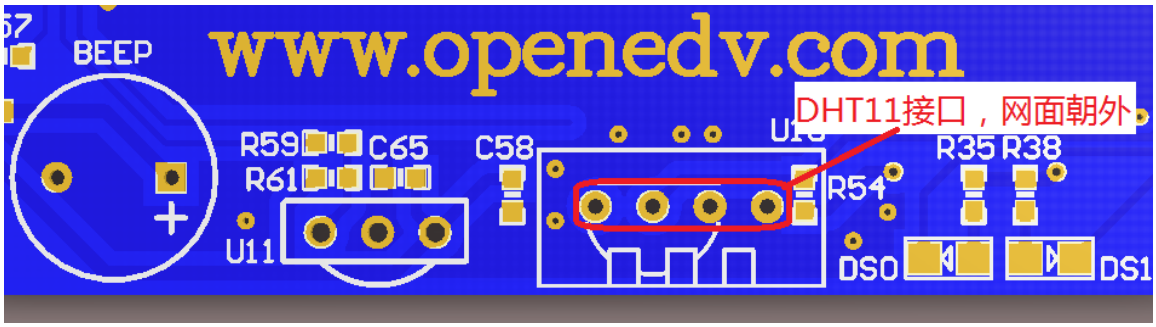


图 39.2.1 DHT11 连接示意图

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。

39.3 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();     //设置为输出
    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();      //设置为输出
    while (DHT11_DQ_IN&&retry<100)//DHT11 会拉低 40~80us
    {
        retry++;
        delay_us(1);
    };
    if(retry>=100)return 1;
    else retry=0;
    while (!DHT11_DQ_IN&&retry<100)//DHT11 拉低后会再次拉高 40~80us
    {
        retry++;
        delay_us(1);
    };
    if(retry>=100)return 1;
    return 0;
}

//从 DHT11 读取一个位
//返回值：1/0
u8 DHT11_Read_Bit(void)
{
    u8 retry=0;
```

```
    while(DHT11_DQ_IN&&retry<100)//等待变为低电平
    {
        retry++;
        delay_us(1);
    }
    retry=0;
    while(!DHT11_DQ_IN&&retry<100)//等待变高电平
    {
        retry++;
        delay_us(1);
    }
    delay_us(40);//等待 40us
    if(DHT11_DQ_IN)return 1;
    else return 0;
}

//从 DHT11 读取一个字节
//返回值：读到的数据
u8 DHT11_Read_Byte(void)
{
    u8 i,dat;
    dat=0;
    for (i=0;i<8;i++)
    {
        dat<<=1;
        dat|=DHT11_Read_Bit();
    }
    return dat;
}

//从 DHT11 读取一次数据
//temp:温度值(范围:0~50°)
//humi:湿度值(范围:20%~90%)
//返回值：0,正常;1,读取失败
u8 DHT11_Read_Data(u8 *temp,u8 *humi)
{
    u8 buf[5];
    u8 i;
    DHT11_Rst();
    if(DHT11_Check()==0)
    {
        for(i=0;i<5;i++)//读取 40 位数据
        {
```

```
            buf[i]=DHT11_Read_Byte();
        }
        if((buf[0]+buf[1]+buf[2]+buf[3])==buf[4])
        {
            *humi=buf[0];
            *temp=buf[2];
        }
    }else return 1;
    return 0;
}

//初始化 DHT11 的 IO 口 DQ 同时检测 DHT11 的存在
//返回 1:不存在
//返回 0:存在
u8 DHT11_Init(void)
{
    GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOB_CLK_ENABLE();              //开启 GPIOB 时钟

    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_HIGH;        //高速
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);        //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;
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(180);                //初始化延时函数
    uart_init(115200);              //初始化 USART
    usmart_dev.init(90);            //初始化 USMART
```

```
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD
    PCF8574_Init();                     //初始化 PCF8574
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/1/16");
    PCF8574_ReadBit(BEEP_IO);       //由于 DHT11 和 PCF8574 的中断引脚共用一个 IO，
            //所以在初始化 DHT11 之前要先读取一次 PCF8574 的任意一个 IO，
            //使其释放掉中断引脚所占用的 IO(PB12 引脚),否则初始化 DS18B20 会出问题
    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
        {
            PCF8574_ReadBit(BEEP_IO);   //读取一次 PCF8574 的任意一个 IO，
                        //使其释放掉 PB12 引脚，否则读取 DHT11 可能会出问题
            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_Toggle;
        }
    }
}
```

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。

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 阿波罗 STM32 开发板上，可以看到 LCD 显示开始显示当前的温度值（假定 DHT11 已经接上去了），如图 39.4.1 所示：

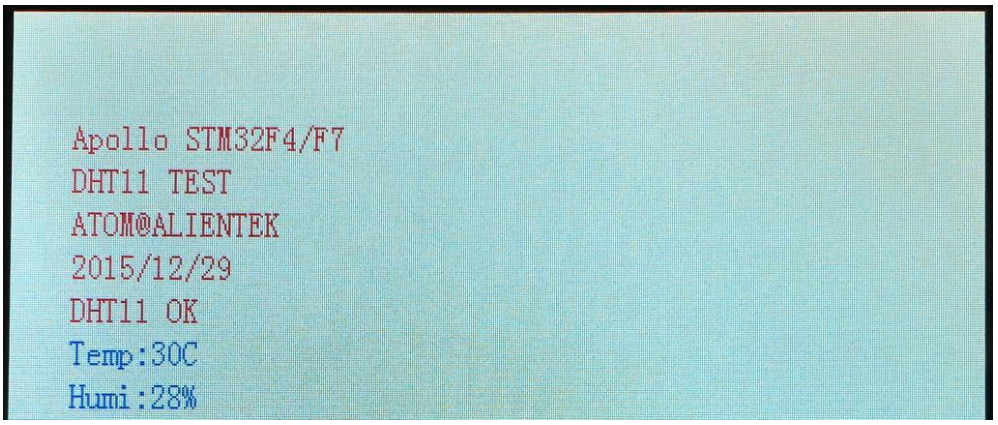


图 39.4.1 DHT11 实验效果图

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？

第四十章 MPU9250 九轴传感器实验

本章，我们介绍一款主流的九轴（三轴加速度+三轴角速度(陀螺仪)+三轴磁力计）传感器：MPU9250，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 阿波罗 STM32F767 开发板自带了 MPU9250 传感器。本章我们将使用 STM32F767 来驱动 MPU9250，读取其原始数据，并利用其自带的 DMP 结合 MPL 库实现姿态解算，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九轴传感器。本章分为如下几个部分：

40.1 MPU9250 简介
40.2 硬件设计
40.3 软件设计
40.4 下载验证

40.1 MPU9250 简介

本节，我们将分 2 个部分介绍：1，MPU9250 基础介绍。2，DMP 使用简介。另外，所有 MPU9250 的相关资料，都在光盘：**A 盘→7，硬件资料→MPU9250 资料** 文件夹里面。

40.1.1 MPU9250 基础介绍

MPU9250 是 InvenSense 公司推出的全球首款整合性 9 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，减少了体积和功耗。

MPU9250 内部集成有 3 轴陀螺仪、3 轴加速度计和 3 轴磁力计，输出都是 16 位的数字量; 可以通过集成电路总线(IIC) 接口和单片机进行数据交互，传输速率可达 400 kHz /s。陀螺仪的角速度测量范围最高达±2000（°/s），具有良好的动态响应特性。加速度计的测量范围最大为±16g(g 为重力加速度)，静态测量精度高。磁力计采用高灵度霍尔型传感器进行数据采集，磁感应强度测量范围为±4800μT，可用于对偏航角的辅助测量。

MPU9250 自带的数字运动处理器（DMP: Digital Motion Processor）硬件加速引擎，可以整合九轴传感器数据，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库（MPL：Motion Process Library），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。

MPU9250 的特点包括：

① 以数字形式输出 9 轴旋转矩阵、四元数(quaternion)、欧拉角格式(Euler Angle forma)的融合演算数据（需 DMP 支持）
② 集成 16 位分辨率，量程为：±250、±500、±1000°与±2000°/sec 的 3 轴角速度传感器(陀螺仪)
③ 集成 16 位分辨率，量程为：±2g、±4g、±8g 和±16g 的 3 轴加速度传感器
④ 集成 16 位分辨率，量程为：±4800uT 的磁场传感器（磁力计）
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 自带一个数字温度传感器
⑦ 可编程数字滤波器
⑧ 支持 SPI 接口，通信速度高达 20Mhz

⑨ 自带 512 字节 FIFO 缓冲区

⑩ 高达 400Khz 的 IIC 通信接口

⑪ 超小封装尺寸：3x3x1mm（QFN）

MPU9250 传感器的检测轴如图 40.1.1.1 所示：

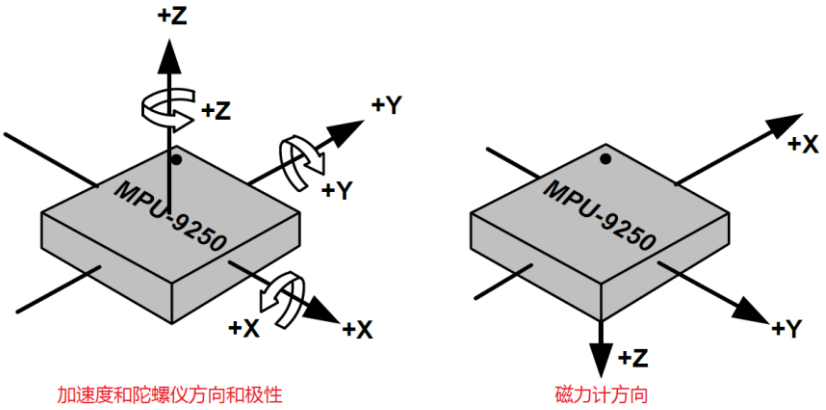


图 40.1.1.1 MPU9250 检测轴及其方向

MPU9250 的内部框图如图 40.1.1.2 所示：

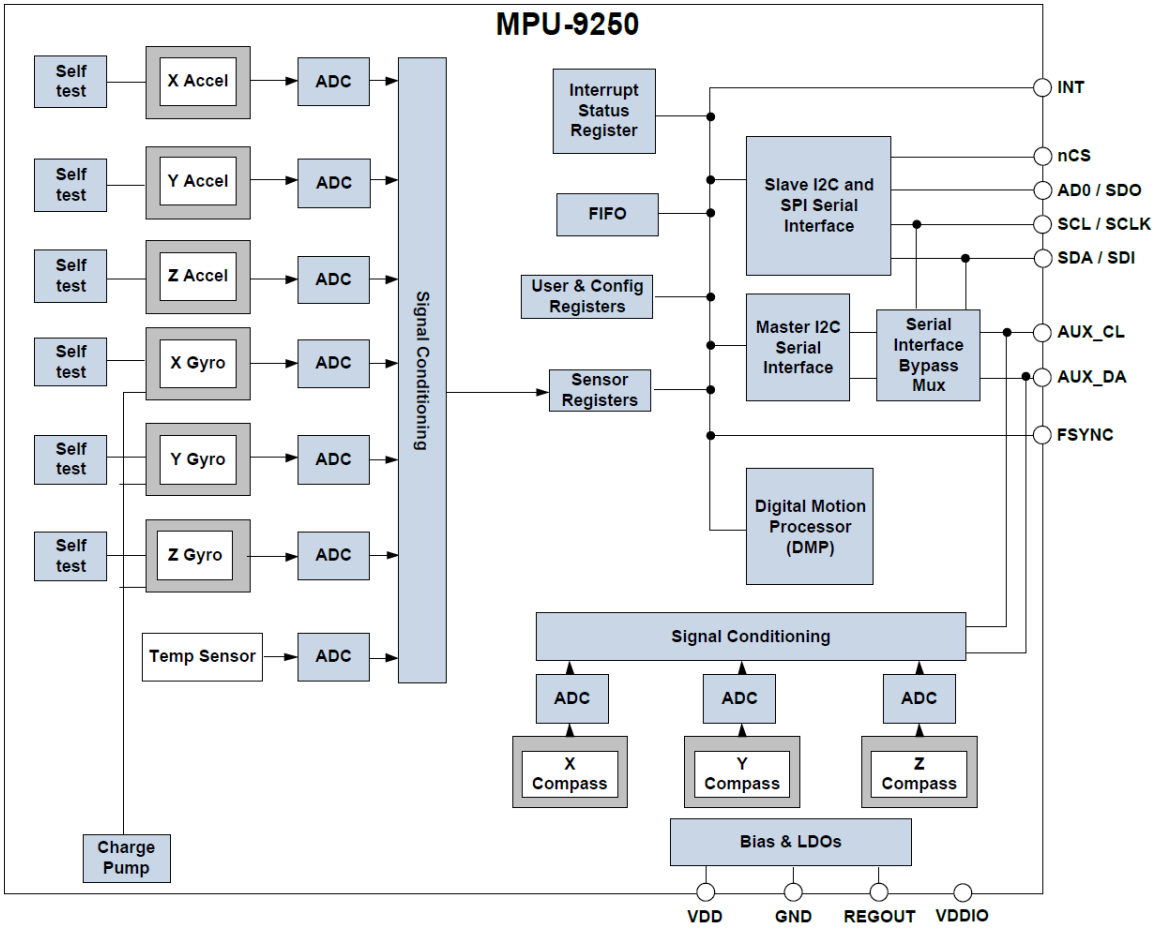


图 40.1.1.2 MPU9250 框图

其中，SCL 和 SDA 可以连接 MCU 的 IIC 接口，MCU 通过这个 IIC 接口来控制 MPU9250，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，比气压传感器。VDDIO 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 MPU9250 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）!! 注意：当使用 SPI 接口的时候，使用：SCLK、SDO、SDI 和 nCS 脚来传输数据。

这里需要和大家说明一下的是：MPU9250，实际上是内部集成了一个 MPU6500 六轴传感器和一个 AK8963 三轴磁力计，他们共用一个 IIC 接口，这样组合成一个九轴传感器。前面说了我们开发板上 MPU9250 的 IIC 地址是 0X68，实际上是指 MPU6500 的地址是 0X68，而 AK8963 磁力计的 IIC 地址，则是：0X0C（不包含最低位）。

在阿波罗 STM32 开发板上，AD0 是接 GND 的，所以 MPU9250 的 IIC 地址是 0X68（不含最低位），IIC 通信的时序我们在之前已经介绍过（第二十九章，IIC 实验），这里就不再细说了。

接下来，我们介绍一下利用 STM32F767 读取 MPU9250 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

MPU9250 采用 IIC 与 STM32F767 通信，所以我们需要先初始化与 MPU9250 连接的 SDA 和 SCL 数据线。这个在前面的 IIC 实验章节已经介绍过了，这里 MPU9250 与 24C02 共用一个 IIC，所以初始化 IIC 完全一模一样。

2）复位 MPU9250

这一步让 MPU9250 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 MPU9250，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。MPU9250 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。

6）配置 AK8963 磁场传感器（磁力计）

经过前面 5 步配置，我们完成了对 MPU6500 的配置，此步需要对 AK8963 进行配置。首先设置控制寄存器 2（0X0B）的最低位为 1，对 AK8963 进行软复位。随后设置控制寄存器 1（0X0A）为 0X11，选择 16 位输出，单次测量模式。随后就可以读取磁力计数据了。

至此，MPU9250 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，各位描述如表 40.1.1.1 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
6B	H_RESET	SLEEP	CYCLE	GYRO_STB	PD_PTAT	CLKSEL[2：0]		

图 40.1.1.1 电源管理寄存器 1 各位描述

其中，H_RESET 位用来控制复位，设置为 1，复位 MPU9250，复位结束后，MPU 硬件自动清零该位。SLEEEP 位用于控制 MPU9250 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。最后 CLKSEL[2:0]用于选择系统时钟源，选择关系如表 40.1.1.2 所示：

CLKSEL[2:0]	时钟源
000	内部 20M RC 晶振
001~101	自动选择最有效的时钟源-PLL
110	内部 20M RC 晶振
111	关闭时钟，保持时序产生电路复位状态

图 40.1.1.2 CLKSEL 选择列表

默认是使用内部 20M RC 晶振的，精度不高，我们一般设置其自动选择最有效的时钟源，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，各位描述如表 40.1.3 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1B	XGYRO_ST	YGYRO_ST	ZGYRO_ST	GYRO_FS_SEL[1:0]		NC	FCHOICE[1:0]	

表 40.1.1.3 陀螺仪配置寄存器各位描述

该寄存器我们只关心 GYRO_FS_SEL[1:0]和 FCHOICE[1:0]这四个位，GYRO_FS_SEL[1:0]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)。FCHOICE[1:0]用于控制 DLPF 旁路，我们一般设置为 3，不旁路 DLPF。

接下来，我们看加速度传感器配置寄存器，寄存器地址为：0X1C，各位描述如表 40.1.1.4 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1C	AX_ST_EN	AY_ST_EN	AZ_ST_EN	ACCEL_FS_SEL[1:0]		Reserved		

表 40.1.1.4 加速度传感器配置寄存器各位描述

该寄存器我们只关心 ACCEL_FS_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，各位描述如表 40.1.1.5 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
23	TEMP_OUT	GY_XOUT	GY_YOUT	GY_ZOUT	ACCEL	SLV_2	SLV_1	SLV_0

表 40.1.1.5 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)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，各位描述如表 40.1.1.6 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
19	SMPLRT_DIV[7:0]							

表 40.1.1.6 陀螺仪采样率分频寄存器各位描述

该寄存器用于设置 MPU9250 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、8Khz 或 32Khz，与数字低通滤波器（DLPF）的设置有关，当 FCHOICE[1:0]不为 11 的时候，频率为 32Khz，其他情况：当 DLPF_CFG=0/7 的时候，频率为 8Khz，否则是 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，各位描述如表 40.1.1.7 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1A	NC	FIFO_MODE	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		

表 40.1.1.7 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]，陀螺仪根据这三个位的配置进行过滤。DLPF_CFG 不同配置对应的过滤情况如表 40.1. 1. 8 所示：

FCHOICE[1:0]		DLPF_CFG[2:0]	角速度传感器（陀螺仪）		
			带宽(Hz)	延迟（ms）	Fs(Khz)
x	0	x	8800	0.064	32
0	1	x	3600	0.11	32
1	1	0	250	0.97	8
1	1	1	184	2.9	1
1	1	10	92	3.9	1
1	1	11	41	5.9	1
1	1	100	20	9.9	1
1	1	101	10	17.85	1
1	1	110	5	33.48	1
1	1	111	3600	0.17	8

表 40.1.1.8 DLPF_CFG 配置表

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。需要注意：FCHOICE[1:0]（通过 0X1B 寄存器配置）必须设置为 11，否则固定 32K 频率，且 DLPF_CFG 的配置无效！

接下来，我们看电源管理寄存器 2，寄存器地址为：0X6C，各位描述如表 40.1.1.9 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
6C	NC	NC	DIS_XA	DIS_YA	DIS_ZA	DIS_XG	DIS_YG	DIS_ZG

表 40.1.1.9 电源管理寄存器 2 各位描述

该寄存器低六位有效，分别控制加速度和陀螺仪的 x/y/z 轴是否开启，这里我们设置全部都开启，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 6 个，地址为：0X3B~0X40，通过读取这 6 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。

另外，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 = 21 + regval/338.87

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。

接下来，我们看 AK8963 的控制寄存器 1 ，寄存器地址为：0X0A，各位描述如表 40.1.1.10 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0A	Reserved			BIT	MODE[3:0]			

表 40.1.1.10 AK8963 控制寄存器 1

其中，BIT 位控制 AK8963 输出位数，0，表示 14 位；1，表示 16 位；我们一般设置为 1。MODE[3:0]用于控制 AK8963 的工作模式：0000，掉电模式；0001，单次测量模式；0010，连续测量模式 1；0110，连续测量模式 2；0100，外部触发测量模式；1000，自测试模式；1111，Fuse ROM 访问模式；我们一般设置 MODE[3:0]=0001，即单次测量模式。

接下来，我们看 AK8963 的控制寄存器 2，寄存器地址为：0X0B，各位描述如表 40.1.1.11 所示：

寄存器（HEX）	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0B	Reserved							SRST

表 40.1.1.11 AK8963 控制寄存器 2

该寄存器仅最低位有效，用于控制 AK8963 的软复位，我们在初始化的时候，设置 SRST=1 即可让 AK8963 进行一次软复位，复位结束后，自动设置为 0。

最后，我们看看磁力计数据输出寄存器，总共由 6 个寄存器组成，地址为：0X03~0X08，通过读取这 6 个寄存器，就可以读到磁力计 x/y/z 轴的值，比如 x 轴的数据，可以通过读取 0X03（低 8 位）和 0X04（高 8 位）寄存器得到，其他轴以此类推。

关于 MPU9250 的基础介绍，我们就介绍到这。MPU9250 的详细资料和相关寄存器介绍，请参考光盘：7，硬件资料→MPU9250 资料→ PS-MPU-9250A-01.pdf 和 RM-MPU-9250A-00.pdf 这两个文档，另外该目录还提供了部分 MPU9250 的中文资料，供大家参考学习。

40.1.2 DMP 使用简介

经过 40.1.1 节的介绍，我们可以读出 MPU9250 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 MPU9250 自带了数字运动处理器，即 DMP，并且，InvenSense 提供了一个 MPU9250 的嵌入式运动处理库（MPL），结合 MPU9250 的 DMP，可以将我们的传感器原始数据，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。

InvenSense 提供的最新 MPL 库版本为：6.12 版本，它提供了基于 STM32F4 Discovery 板的参考例程（IAR 工程），我们只需要将它移植到我们的开发板上即可。官方原版驱动在光盘：7，硬件资料→MPU9250 资料→ motion_driver_6.12.zip。解压之后，里面有 MPL 的参考例程（arm/msp430）、LIB 库（mpl libraries）和说明文档（documentation）等资料，大家可以参考 documentation 文件夹下的几个 PDF 教程来学习 MPL 的使用。

官方 MPL 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→MPU9250→MPL 文件夹内，包含 4 个文件夹，如图 40.1.2.1 所示：

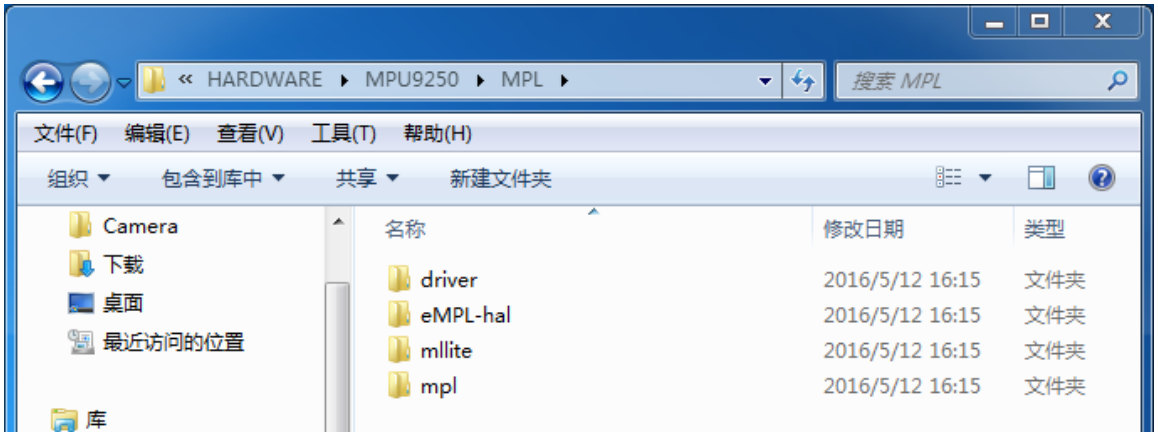


图 40.1.2.1 移植后的驱动库代码

为了方便大家使用该驱动库（MPL），我们在 inv_mpu.c 里面添加了两个函数：mpu_dmp_init 和 mpu_mpl_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 MPU9250 DMP 初始化函数，该函数代码如下：

```
//MPU9250,dmp 初始化
//返回值:0,正常
//       其他,失败
u8 mpu_dmp_init(void)
{
    u8 res=0;
    struct int_param_s int_param;
```

```
unsigned char accel_fsr;
unsigned short gyro_rate, gyro_fsr;
unsigned short compass_fsr;
IIC_Init();                         //初始化 IIC 总线
if(mpu_init(&int_param)==0)         //初始化 MPU9250
{
    res=inv_init_mpl();             //初始化 MPL
    if(res)return 1;
    inv_enable_quaternion();
    inv_enable_9x_sensor_fusion();
    inv_enable_fast_nomot();
    inv_enable_gyro_tc();
    inv_enable_vector_compass_cal();
    inv_enable_magnetic_disturbance();
    inv_enable_eMPL_outputs();
    res=inv_start_mpl();            //开启 MPL
    if(res)return 1;
    res=mpu_set_sensors(INV_XYZ_GYRO|INV_XYZ_ACCEL|
                        INV_XYZ_COMPASS);//设置所需要的传感器
    if(res)return 2;
    res=mpu_configure_fifo(INV_XYZ_GYRO | INV_XYZ_ACCEL);  //设置 FIFO
    if(res)return 3;
    res=mpu_set_sample_rate(DEFAULT_MPU_HZ);                //设置采样率
    if(res)return 4;
    res=mpu_set_compass_sample_rate(1000/COMPASS_READ_MS); //磁力计采样率
    if(res)return 5;
    mpu_get_sample_rate(&gyro_rate);
    mpu_get_gyro_fsr(&gyro_fsr);
    mpu_get_accel_fsr(&accel_fsr);
    mpu_get_compass_fsr(&compass_fsr);
    inv_set_gyro_sample_rate(1000000L/gyro_rate);
    inv_set_accel_sample_rate(1000000L/gyro_rate);
    inv_set_compass_sample_rate(COMPASS_READ_MS*1000L);
    inv_set_gyro_orientation_and_scale(
        inv_orientation_matrix_to_scalar(gyro_orientation),(long)gyro_fsr<<15);
    inv_set_accel_orientation_and_scale(
        inv_orientation_matrix_to_scalar(gyro_orientation),(long)accel_fsr<<15);
    inv_set_compass_orientation_and_scale(
        inv_orientation_matrix_to_scalar(comp_orientation),(long)compass_fsr<<15);
    res=dmp_load_motion_driver_firmware();   //加载 dmp 固件
    if(res)return 6;
    res=dmp_set_orientation(inv_orientation_matrix_to_scalar(gyro_orientation));
    //设置陀螺仪方向
```

```
            if(res)return 7;
            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 8;
            res=dmp_set_fifo_rate(DEFAULT_MPU_HZ);//设置 DMP 输出速率(不超过 200Hz)
            if(res)return 9;
            res=run_self_test();            //自检
            if(res)return 10;
            res=mpu_set_dmp_state(1);  //使能 DMP
            if(res)return 11;
        }
        return 0;
    }
```

此函数首先通过 IIC_Init(需外部提供)初始化与 MPU9250 连接的 IIC 接口，然后调用 mpu_init 函数，初始化 MPU9250，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_mpl_get_data 来读取姿态解算后的数据了。

mpu_mpl_get_data 函数代码如下：

```
//得到 mpl 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_mpl_get_data(float *pitch,float *roll,float *yaw)
{
    unsigned long sensor_timestamp,timestamp;
    short gyro[3], accel_short[3],compass_short[3],sensors;
    unsigned char more;
    long compass[3],accel[3],quat[4],temperature;
    long data[9];
    int8_t accuracy;
    if(dmp_read_fifo(gyro, accel_short, quat, &sensor_timestamp, &sensors,&more))return 1;
    if(sensors&INV_XYZ_GYRO)
    {
        inv_build_gyro(gyro,sensor_timestamp);          //把新数据发送给 MPL
        mpu_get_temperature(&temperature,&sensor_timestamp);
        inv_build_temp(temperature,sensor_timestamp); //发温度值给 MPL,仅陀螺仪需要
    }
    if(sensors&INV_XYZ_ACCEL)
    {
```

```
            accel[0] = (long)accel_short[0];
            accel[1] = (long)accel_short[1];
            accel[2] = (long)accel_short[2];
            inv_build_accel(accel,0,sensor_timestamp);        //把加速度值发给 MPL
        }
        if (!mpu_get_compass_reg(compass_short, &sensor_timestamp))
        {
            compass[0]=(long)compass_short[0];
            compass[1]=(long)compass_short[1];
            compass[2]=(long)compass_short[2];
            inv_build_compass(compass,0,sensor_timestamp); //把磁力计值发给 MPL
        }
        inv_execute_on_data();
        inv_get_sensor_type_euler(data,&accuracy,&timestamp);
        *roll    = (data[0]/q16);
        *pitch = -(data[1]/q16);
        *yaw = -data[2] / q16;
        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_stm32f767xx.s 里面设置，默认是 800)。

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。

40.2 硬件设计

本实验采用 STM32F767 的 2 个普通 IO 连接 MPU9250（IIC），本章实验功能简介：程序先初始化 MPU9250 等外设，然后利用 MPL 库，初始化 MPU9250 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、磁力计、MPL 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_TC 匿名科创地面站 v4.exe），可以实时显示 MPU9250 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） LCD 模块
4） 串口
5） MPU9250

前 4 个，在之前的实例已经介绍过了，这里我们仅介绍 MPU9250 与阿波罗 STM32F767 开发板的连接。该接口与 MCU 的连接原理图如 40.2.1 所示：

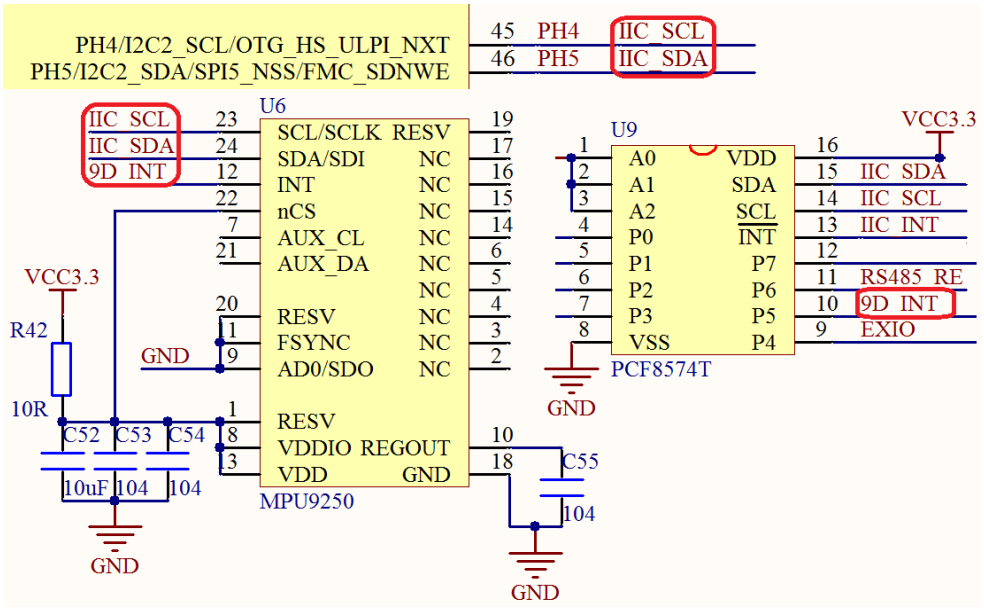


图 40.2.1 MPU9250 与 STM32F767 的连接电路图

从上图可以看出，MPU9250 的 SCL 和 SDA 与 STM32F767 开发板的 PH4 和 PH5 连接，与 24C02 等共用 IIC 总线。图中，AD0 接的 GND，所以 MPU9250 的器件地址是：0X68。

注意：9D_INT 信号，是连接在 PCF8574T 的 P5 脚上的，并没有直接连接到 MCU，所以，在需要读取 9D_INT 的时候，需要先初始化 PCF8574T。不过，本例程用不到 9D_INT，所以，可以不初始化 PCF8574T，直接通过 IIC 总线读取数据即可。

40.3 软件设计

打开本章实验工程可以看到，我们在 HARDWARE 分组之下添加了 MPU9250 驱动源文件 mpu9250.c，并且包含了其对应的头文件 mpu9250.h。同时，将 MPL 驱动库代码（见光盘例程源码：实验 35 MPU9250 九轴传感器实验\HARDWARE\MPU9250\MPL 目录）添加到新建的 MPL 分组之下，工程结构如图 39.3.1 所示：

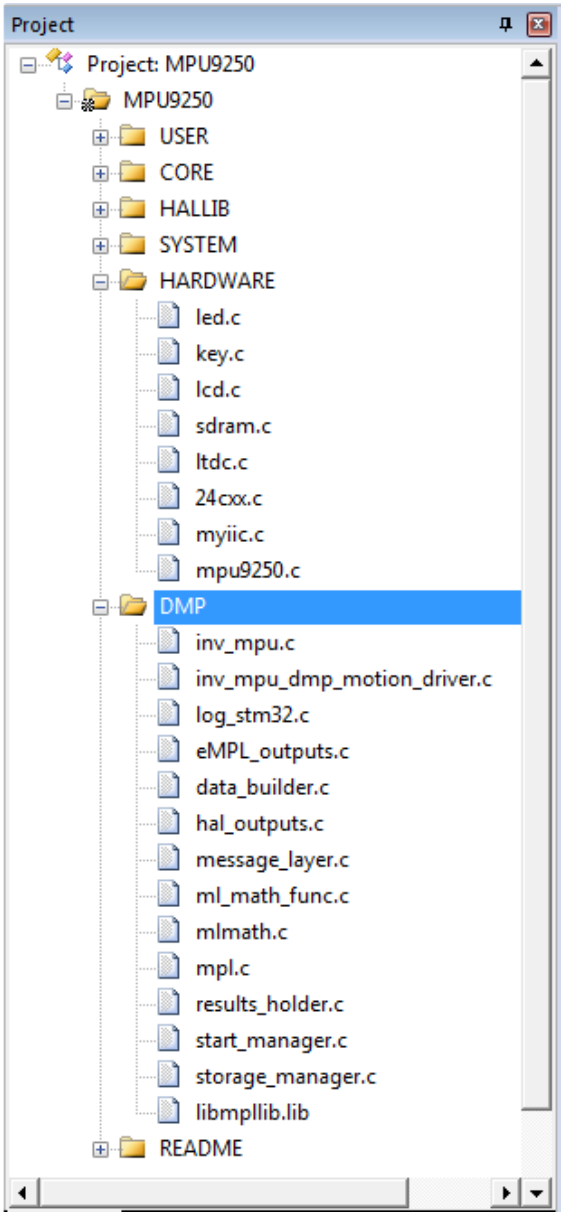


图 39.3.1 MPU9250 工程结构图

注意：MPL 代码，要求在 MDK Options for Target 的 C/C++选项卡里面，要勾选 C99 模式，否则编译出错。

由于篇幅所限，MPL 部分的代码，我们就不详细介绍了，请大家参考 motion_driver_6.12.zip 里面的相关教程进行学习。我们仅介绍 mpu9250.c 里面的部分函数，首先是：MPU_Init，该函数代码如下：

```
//初始化 MPU9250
//返回值:0,成功
//      其他,错误代码
u8 MPU9250_Init(void)
{
    u8 res=0;
    IIC_Init();        //初始化 IIC 总线
```

```
    MPU_Write_Byte(MPU9250_ADDR,MPU_PWR_MGMT1_REG,0X80);//复位 MPU9250
    delay_ms(100);  //延时 100ms
    MPU_Write_Byte(MPU9250_ADDR,MPU_PWR_MGMT1_REG,0X00);//唤醒 MPU9250
    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(MPU9250_ADDR,MPU_INT_EN_REG,0X00);  //关闭所有中断
    MPU_Write_Byte(MPU9250_ADDR,MPU_USER_CTRL_REG,0X00);//主模式关闭
    MPU_Write_Byte(MPU9250_ADDR,MPU_FIFO_EN_REG,0X00); //关闭 FIFO
    MPU_Write_Byte(MPU9250_ADDR,MPU_INTBP_CFG_REG,0X82);
                //INT 引脚低电平有效，开启 bypass 模式，可以直接读取磁力计
    res=MPU_Read_Byte(MPU9250_ADDR,MPU_DEVICE_ID_REG);
                                                //读取 MPU6500 的 ID
    if(res==MPU6500_ID) //器件 ID 正确
    {
        MPU_Write_Byte(MPU9250_ADDR,MPU_PWR_MGMT1_REG,0X01);
                                            //设置 CLKSEL,PLL X 轴为参考
        MPU_Write_Byte(MPU9250_ADDR,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;

    res=MPU_Read_Byte(AK8963_ADDR,MAG_WIA);       //读取 AK8963 ID
    if(res==AK8963_ID)
    {
MPU_Write_Byte(AK8963_ADDR,MAG_CNTL1,0X11); //设置 AK8963 为单次测量模式
    }else return 1;

    return 0;
}
```

该函数就是按我们在 39.1.1 节介绍的方法，对 MPU9250 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，我们再看 MPU_Get_Temperature、MPU_Get_Gyroscope 、MPU_Get_Accelerometer 和 MPU_Get_Magnetometer 等四个函数，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(MPU9250_ADDR,MPU_TEMP_OUTH_REG,2,buf);
    raw=((u16)buf[0]<<8)|buf[1];
```

```
    temp=21+((double)raw)/333.87;
    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(MPU9250_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(MPU9250_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;;
}
//得到磁力计值(原始值)
//mx,my,mz:磁力计 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_Magnetometer(short *mx,short *my,short *mz)
{
    u8 buf[6],res;
    res=MPU_Read_Len(AK8963_ADDR,MAG_XOUT_L,6,buf);
```

```
        if(res==0)
        {
            *mx=((u16)buf[1]<<8)|buf[0];
            *my=((u16)buf[3]<<8)|buf[2];
            *mz=((u16)buf[5]<<8)|buf[4];
        }
        MPU_Write_Byte(AK8963_ADDR,MAG_CNTL1,0X11);
        //AK8963 每次读完以后都需要重新设置为单次测量模式
        return res;;
    }
```

其中 MPU_Get_Temperature 用于获取 MPU9250 自带温度传感器的温度值，然后 MPU_Get_Gyroscope、MPU_Get_Accelerometer 和 MPU_Get_Magnetometer 分别用于读取陀螺仪、加速度传感器和磁力计的原始数据。

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;} //等待应答
    }
    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 MPL 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 MPL 部分的：i2c_read 函数。MPL 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 mpu9250.c 我们就介绍到这， mpu9250.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 内容，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){};
    USART1->TDR=c;
}
//传送数据给匿名四轴地面站(V4 版本)
//fun:功能字. 0X01~0X1C
//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]=0XAA;           //帧头
    send_buf[1]=0XAA;           //帧头
    send_buf[2]=fun;            //功能字
    send_buf[3]=len;            //数据长度
    for(i=0;i<len;i++)send_buf[4+i]=data[i];                //复制数据
    for(i=0;i<len+4;i++)send_buf[len+4]+=send_buf[i];       //计算校验和
    for(i=0;i<len+5;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+陀螺仪数据(传感器帧)
//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 mpu9250_send_data(short aacx,short aacy,short aacz,short gyrox,short gyroy,short gyroz)
{
    u8 tbuf[18];
    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[12]=0;//开启 MPL 后,无法直接读取磁力计数据,所以这里直接屏蔽掉.用 0 替代.
    tbuf[13]=0;
    tbuf[14]=0;
    tbuf[15]=0;
    tbuf[16]=0;
    tbuf[17]=0;
    usart1_niming_report(0X02,tbuf,18);//传感器帧,0X02
}
//通过串口 1 上报结算后的姿态数据给电脑(状态帧)
//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
//csb:超声波高度,单位:cm
//prs:气压计高度,单位:mm
void usart1_report_imu(short roll,short pitch,short yaw,short csb,int prs)
```

```
{
    u8 tbuf[12];
    tbuf[0]=(roll>>8)&0XFF;
    tbuf[1]=roll&0XFF;
    tbuf[2]=(pitch>>8)&0XFF;
    tbuf[3]=pitch&0XFF;
    tbuf[4]=(yaw>>8)&0XFF;
    tbuf[5]=yaw&0XFF;
    tbuf[6]=(csb>>8)&0XFF;
    tbuf[7]=csb&0XFF;
    tbuf[8]=(prs>>24)&0XFF;
    tbuf[9]=(prs>>16)&0XFF;
    tbuf[10]=(prs>>8)&0XFF;
    tbuf[11]=prs&0XFF;
    usart1_niming_report(0X01,tbuf,12);//状态帧,0X01
}
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
    Cache_Enable();                     //打开 L1-Cache
    MPU_Memory_Protection();            //保护相关存储区域
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);       //设置时钟,216Mhz
    delay_init(180);                    //初始化延时函数
    …//此处省略部分初始化代码
    while(mpu_dmp_init())
    {
        LCD_ShowString(30,130,200,16,16,"MPU9250 Error"); delay_ms(200);
        LCD_Fill(30,130,239,130+16,WHITE); delay_ms(200);
        LED0_Toggle;//DS0 闪烁
    }
    LCD_ShowString(30,130,200,16,16,"MPU9250 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_mpl_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)mpu9250_send_data(aacx,aacy,aacz,gyrox,gyroy,gyroz);
        //发送加速度+陀螺仪原始数据
        if(report)usart1_report_imu((int)(roll*100),(int)(pitch*100),(int)(yaw*100),0,0);
        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_Toggle;//DS0 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 MPU9250。上位机软件使用：ANO_TC 匿名科创地面站 v4.exe，该软件在：开发板光盘→6，软件资料→软件→匿名地面站 文件夹里面可以找到，该软件的使用方法，见该文件夹下的：飞控通信协议 v1.3-0720.pdf，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，然后上报给匿名地面站软件。MPU9250_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，通过传感器帧（02H）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 MPU9250 的姿态，传感器数据等，通过状态帧（01H）发送。

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。

至此，我们的软件设计部分就结束了。

40.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 40.4.1 所示的内容：

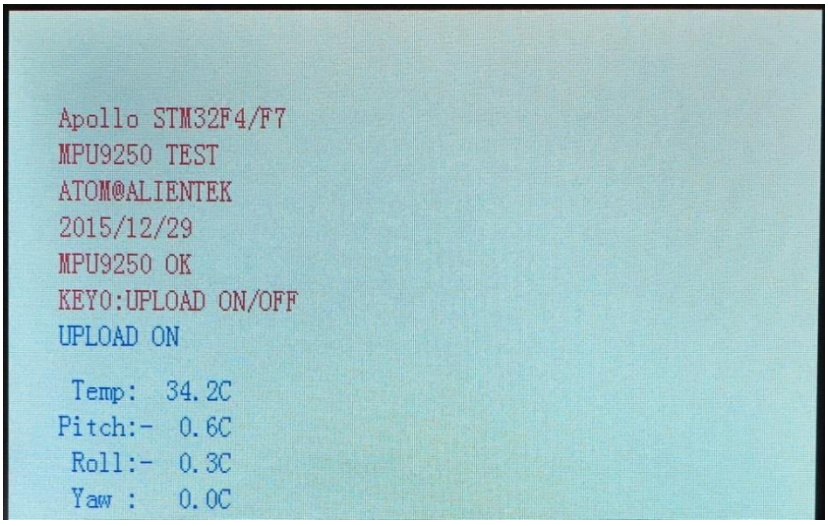


图 40.4.1 程序运行时 LCD 显示内容

屏幕显示了 MPU9250 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_TC 匿名科创地面站 v4.exe，这个软件，接收 STM32F767 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 40.4.2 和图 40.4.3 所示：

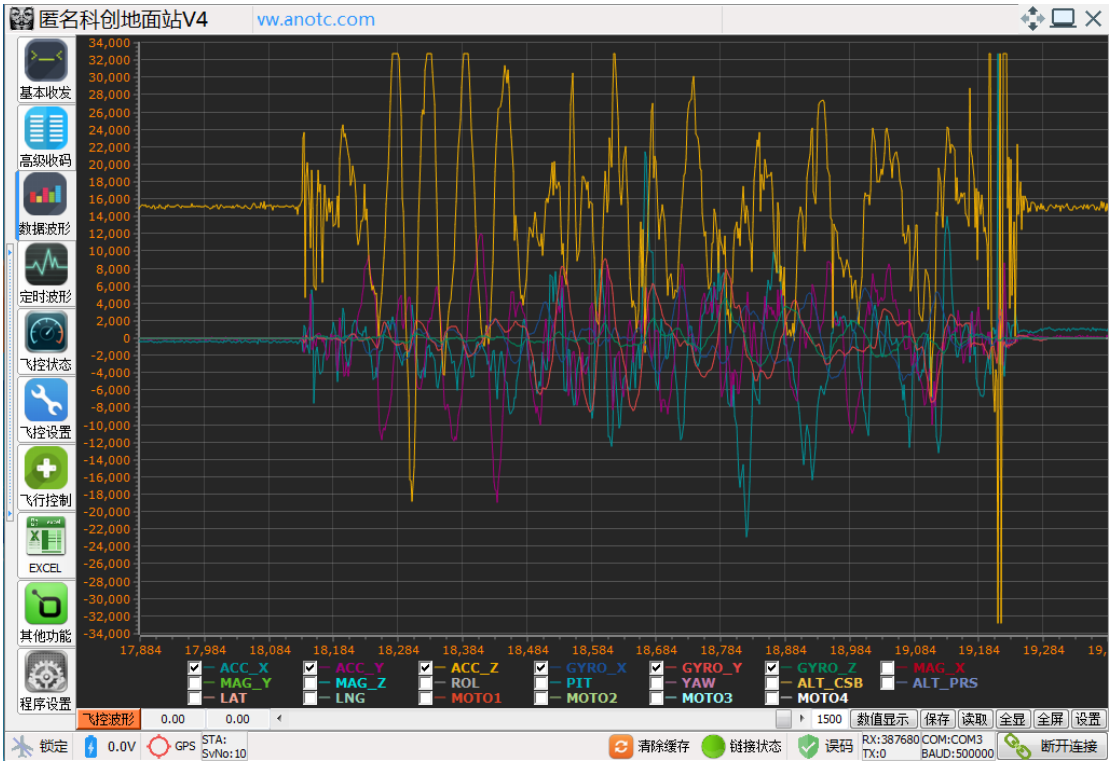


图 40.4.2 传感器数据波形显示

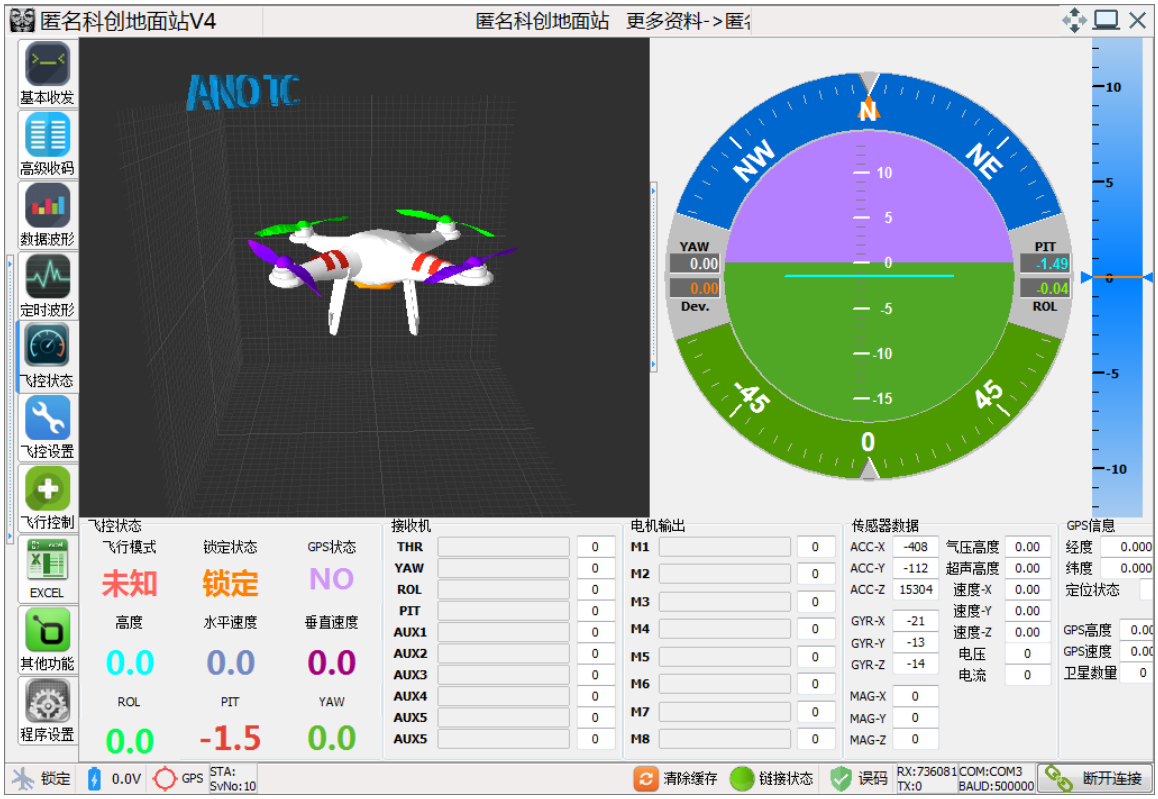


图 40.4.3 飞控状态显示

图 40.4.2 就是波形化显示我们通过 MPU9250_send_data 函数发送的数据，采用传感器帧（02）发送，总共 6 条线（ACC_X、ACC_Y、ACC_Z、GYRO_X、GYRO_Y 和 GYRO_Z）显示波形，全部来自传感器帧，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（**请注意把选项“程序设置->上位机设置->数据校验”设置为 Off，否则可能看不到数据和飞控状态变化**）。

图图 40.4.3 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，采用状态帧（01）上传，同时还显示了加速度陀螺仪等传感器的原始数据。

第四十一章 无线通信实验

ALIENTKE 阿波罗 STM32F767 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/WIFI 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 阿波罗 STM32 开发板上实现无线通信。在本章中，我们将使用两块阿波罗 STM32F767 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：

41.1 SPI&NRF24L01 无线模块简介
41.2 硬件设计
41.3 软件设计
41.4 下载验证

41.1 SPI&NRF24L01 无线模块简介

本章，我们将通过 STM32F767 的 SPI 接口来驱动 NRF24L01 无线模块，接下来我们将分别介绍 STM32F7 的 SPI 接口和 NRF24L01 无线模块。

41.1.1 SPI 接口简介

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，STM32F767 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。

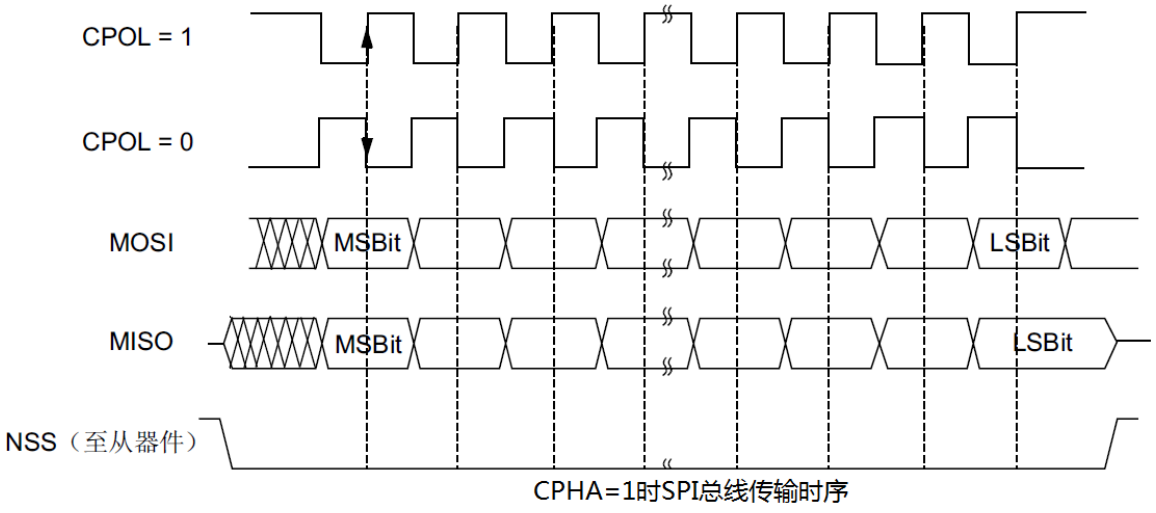
SCLK 时钟信号，由主设备产生。

CS 从设备片选信号，由主设备控制。

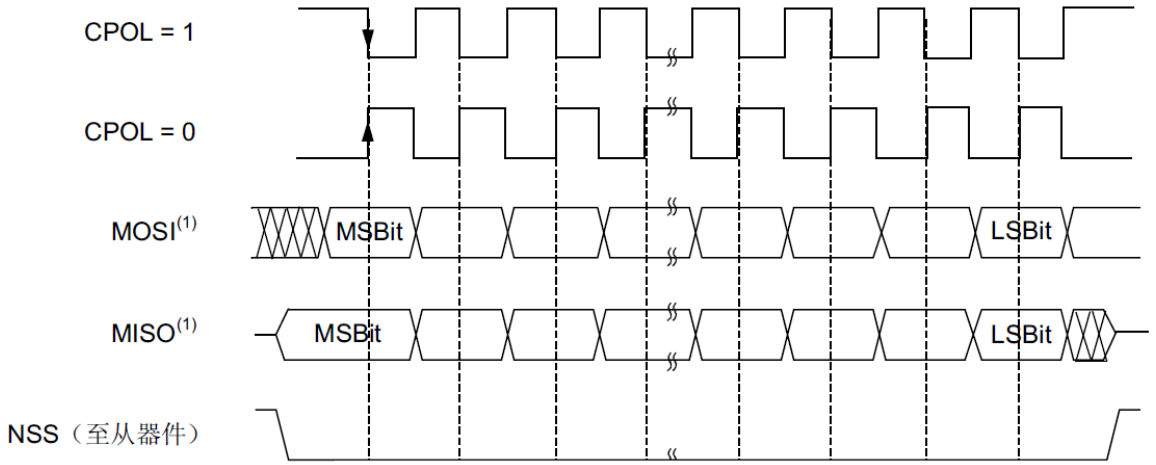
SPI 主要特点有：可以同时发出和接收串行数据；可以当作主机或从机工作；提供频率可编程时钟；发送结束中断标志；写冲突保护；总线竞争保护等。

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 41.1.1.1 所示：



CPHA=1时SPI总线传输时序



CPHA=0时SPI总线传输时序

图 41.1.1.1 不同时钟相位下的总线传输时序（CPHA=0/1）

对于 STM32F7 来说，SPI 的 MSB 和 LSB 是可以配置的，通过 SPI_CR1 的 LSBFIRST 位进行控制，当该位为 1 时，表示 LSB 在前；当该位为 0 时，表示 MSB 在前；

STM32F7 的 SPI 功能很强大，SPI 时钟最高可以到 54Mhz，支持 DMA，可以配置为 SPI 协议或者 I2S 协议（支持全双工 I2S）。

本章，我们将使用 STM32F767 的 SPI 来驱动 NRF24L01 无线模块。这里对 SPI 我们只简单介绍一下 SPI 的使用，STM32F767 的 SPI 详细介绍请参考《STM32F7 中文参考手册》第 965 页，32 节。

本章，我们使用 STM32F767 的 SPI2 来驱动 NRF24L01，HAL 库中 SPI 相关函数定义分布在源文件 stm32f7xx_hal_spi.c 和对应的头文件 stm32f7xx_hal_spi.h 中。下面就来看看 STM32F767 的 SPI2 主模式配置步骤：

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(AF5)输出，这样才会连接到 SPI2 上。这里我们使用的是 PB13、14、15 这 3 个（SCK.、MISO、MOSI，CS 使用软件管理方式），所以设置这三个为复用 IO，复用功能为 AF5。

HAL 库中，SPI2 时钟使能方法如下：

```
__HAL_RCC_SPI2_CLK_ENABLE(); //使能 SPI2 时钟
```

IO 口复用功能设置在前面我们已经多次讲解，这里就不累赘了。和串口等其他外设一样，HAL 库同样提供了 SPI 的初始化回调函数用来编写与 MCU 相关的配置。

```
void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi);
```

2）初始化 SPI2,设置 SPI2 工作模式等。

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 54Mhz），以及数据的格式（MSB 在前还是 LSB 在前）。在库函数中初始化 SPI 的函数为：

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

该函数只有一个入口参数 hspi，为 SPI_HandleTypeDef 结构体指针类型，该结构体定义如下：

```
typedef struct __SPI_HandleTypeDef
{
  SPI_TypeDef                 *Instance;          //外设寄存器基地址
  SPI_InitTypeDef             Init;               //初始化结构体
  uint8_t                     *pTxBuffPtr;        //发送缓存
  uint16_t                    TxXferSize;         /发送数据大小
  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
  uint16_t                    RxXferCount;        //还剩余多少个数据要接收
  DMA_HandleTypeDef           *hdmatx;            //DMA 发送句柄
  DMA_HandleTypeDef           *hdmarx;            //DMA 接收句柄
  void                        (*RxISR)(struct __SPI_HandleTypeDef * hspi);
  void                        (*TxISR)(struct __SPI_HandleTypeDef * hspi);
  HAL_LockTypeDef             Lock;
  __IO HAL_SPI_StateTypeDef   State;
  __IO uint32_t               ErrorCode;
}SPI_HandleTypeDef;
```

该结构体成员变量较多。成员变量 Instance 用来设置外设寄存器基地址，对于 SPI2，我们设置为宏定义标识符 SPI2 即可。成员变量 pTxBuffPtr，TxXferSize 和 TxXferCount 用来设置 SPI 发送缓存，发送数据量和发送剩余数据量。成员变量 pRxBuffPtr，RxXferSize 和 RxXferCount 用来设置接收缓存，接收数据量和接收剩余数据量。CRCSize 用来设置 CRC 校验字节数。hdmatx 和 hdmarx 是 DMA 处理句柄。RxISR 和 TxISR 是函数指针，用来指向 SPI 的接收和发送中断处理函数。这里我们着重讲解第二个成员变量 Init,该成员变量用来初始化 SPI 时序和工作模式等，Init 成员变量是 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
    uint32_t CRCLength;
    uint32_t NSSPMode;
}SPI_InitTypeDef;
```

结构体成员变量比较多，接下来我们简单讲解一下：

参数 Mode 用来设置 SPI 的主从模式，这里我们设置为主机模式 SPI_MODE_MASTER，当然有需要你也可以选择为从机模式 SPI_MODE_SLAVE。

参数 Direction 用来设置 SPI 的通信方式，可以选择为半双工，全双工，以及串行发和串行收方式，这里我们选择全双工模式 SPI_DIRECTION_2LINES。

参数 DataSize 为 8 位还是 16 位帧格式选择项，这里我们是 8 位传输，选择 SPI_DATASIZE_8BIT。

参数 CLKPolarity 用来设置时钟极性，我们设置串行同步时钟的空闲状态为高电平所以我们选择 SPI_POLARITY_HIGH。

参数 CLKPhase 用来设置时钟相位，也就是选择在串行同步时钟的第几个跳变沿（上升或下降）数据被采样，可以为第一个或者第二个条边沿采集，这里我们选择第二个跳变沿，所以选择 SPI_PHASE_2EDGE

参数 NSS 设置 NSS 信号由硬件（NSS 管脚）还是软件控制，这里我们通过软件控制 NSS 关键，而不是硬件自动控制，所以选择 SPI_NSS_SOFT。

参数 SPI_ BaudRatePrescaler 很关键，就是设置 SPI 波特率预分频值也就是决定 SPI 的时钟的参数，从 2 分频到 256 分频 8 个可选值，初始化的时候我们选择 256 分频值 SPI_BAUDRATEPRESCALER_256，传输速度为 108M/256=421.875KHz。

参数 FirstBit 设置数据传输顺序是 MSB 位在前还是 LSB 位在前，，这里我们选择 SPI_FIRSTBIT_MSB 高位在前。

参数 TIMode 用来设置 TI 模式使能还是禁止，这里我们禁止即可。

参数 CRCCalculation，CRCPolynomial 和 CRCLength 分别用来设置使能/禁止 CRC 校验，CRC 校验多项式以及 CRC 校验的长度。

参数 NSSPMode 用来设置在连续传输时，是否允许 SPI 在两个连续数据间产生 NSS 脉冲。

设置好上面参数后，我们就可以初始化 SPI 外设了。初始化的范例格式为：

```
SPI2_Handler.Instance=SPI2;                          //SP2
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;// SPI 设置为双线模式
SPI2_Handler.Init.DataSize=SPI_DATASIZE_8BIT;      // SPI 发送接收 8 位帧结构
SPI2_Handler.Init.CLKPolarity=SPI_POLARITY_HIGH; //时钟的空闲状态为高电平
SPI2_Handler.Init.CLKPhase=SPI_PHASE_2EDGE;   //同步时钟的第二个跳变沿数据采样
SPI2_Handler.Init.NSS=SPI_NSS_SOFT;              //内部 NSS 信号有 SSI 位控制
SPI2_Handler.Init.BaudRatePrescaler=SPI_BAUDRATEPRESCALER_256;//波特率 256 分频
SPI2_Handler.Init.FirstBit=SPI_FIRSTBIT_MSB;     //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 CRC 校验
SPI2_Handler.Init.CRCPolynomial=7;               //CRC 值计算的多项式
HAL_SPI_Init(&SPI2_Handler);//初始化 SPI2
```

3）使能 SPI1。

这一步通过 SPI1_CR1 的 bit6 来设置，以启动 SPI1，在启动之后，我们就可以开始 SPI 通讯了。库函数使能 SPI1 的方法为：

```
__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）SPI 中断处理

SPI1 和 SPI2 中断服务函数分别为 SPI1_IRQHandler 和 SPI2_IRQHandler，和串口中断处理过程一样，HAL 库同样提供了 SPI 中断通用处理入口函数 HAL_SPI_IRQHandler，同时提供了多个中断处理回调函数，通信过程各种中断最终都会通过相应的回调函数来处理。SPI 相关回调函数如下：

```
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi);//发送完成
void HAL_SPI_RxCpltCallback(SPI_HandleTypeDef *hspi);//接收完成
void HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi);//发送接收完成
void HAL_SPI_TxHalfCpltCallback(SPI_HandleTypeDef *hspi);//发送过半
void HAL_SPI_RxHalfCpltCallback(SPI_HandleTypeDef *hspi);//接收过半
void HAL_SPI_TxRxHalfCpltCallback(SPI_HandleTypeDef *hspi);//发送接收过半
void HAL_SPI_ErrorCallback(SPI_HandleTypeDef *hspi);//传输错误
```

SPI2 的使用就介绍到这里，接下来介绍一下 NRF24L01 无线模块。

41.1.2 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 41.1.2.1 所示：

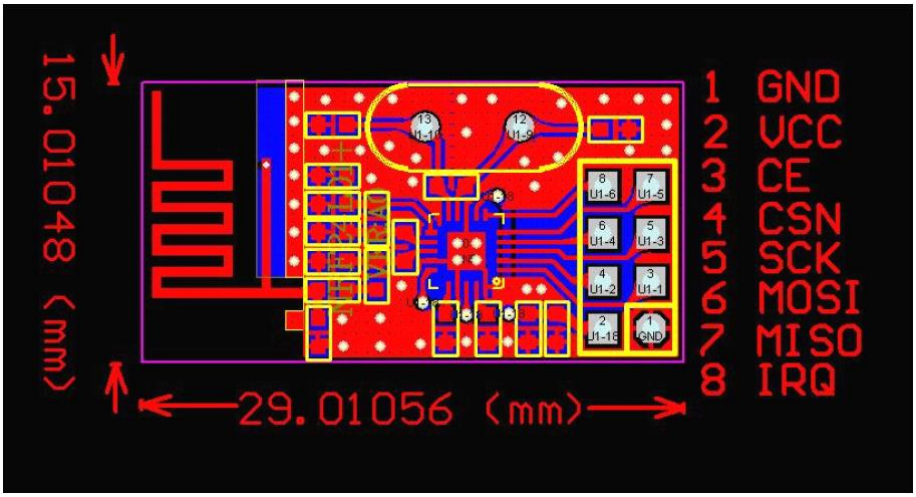


图 41.1.2.1 NRF24L01 无线模块外观引脚图

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。

41.2 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） LCD 模块

4） SPI2

5） NRF24L01 模块

NRF24L01 模块属于外部模块，这里我们仅介绍开发板上 NRF24L01 模块接口和 STM32F767 的连接情况，他们的连接关系如图 41.2.1 所示：

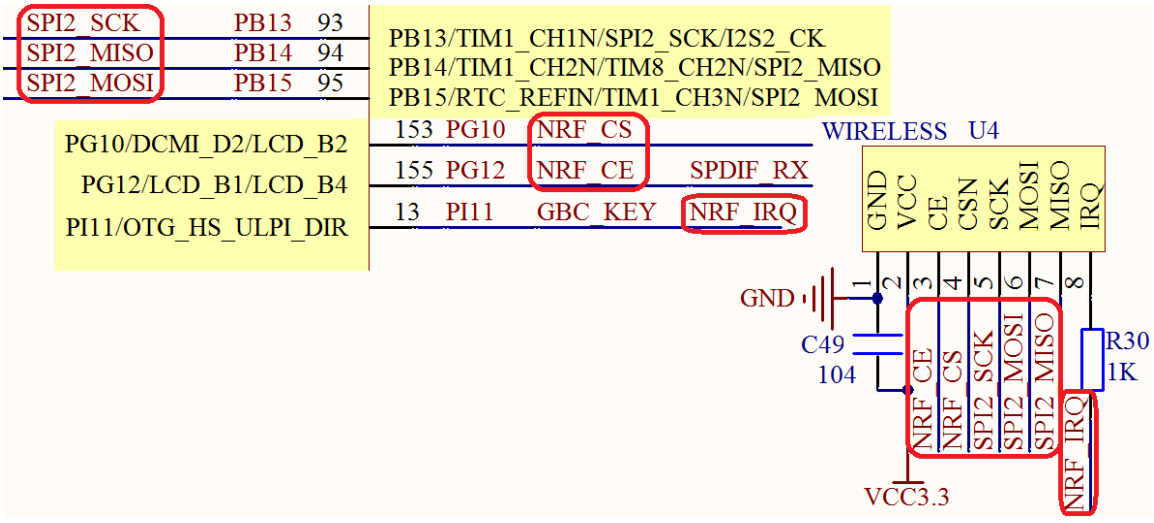


图 41.2.1 NRF24L01 模块接口与 STM32F767 连接原理图

这里NRF24L01使用的是SPI2，连接在PB13/PB14/PB15上。注意：NRF_IRQ和GBC_KEY共用了PI11，NRF_CE和SPDIF_RX共用PG12，所以，他们不能同时使用，需要分时复用。

由于无线通信实验是双向的，所以至少要有两个模块同时能工作，这里我们使用2套ALIENTEK阿波罗STM32F767开发板来向大家演示。

41.3 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 SPI2 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 spi.c 文件可以看到 spi2 相关的驱动函数，内容如下：

```
SPI_HandleTypeDef SPI2_Handler;  //SPI2 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 SPI2 的初始化
void SPI2_Init(void)
{
    SPI2_Handler.Instance=SPI2;                         //SP2
    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;//SPI 设置为双线模式
    SPI2_Handler.Init.DataSize=SPI_DATASIZE_8BIT;      // SPI 发送接收 8 位帧结构
    SPI2_Handler.Init.CLKPolarity=SPI_POLARITY_HIGH; //同步时钟空闲状态为高电平
    SPI2_Handler.Init.CLKPhase=SPI_PHASE_2EDGE; //同步时钟第二个跳变沿采样数据
    SPI2_Handler.Init.NSS=SPI_NSS_SOFT;  //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; //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);
    __HAL_SPI_ENABLE(&SPI2_Handler);                    //使能 SPI2
    SPI2_ReadWriteByte(0Xff);                           //启动传输
}

//SPI2 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 GPIOF 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_FAST;         //快速
    GPIO_Initure.Alternate=GPIO_AF5_SPI2;       //复用为 SPI2
    HAL_GPIO_Init(GPIOB,&GPIO_Initure);         //初始化
}

//SPI 速度设置函数
//SPI 速度=fAPB1/分频系数
//@ref SPI_BaudRate_Prescaler:
//SPI_BAUDRATEPRESCALER_2~SPI_BAUDRATEPRESCALER_2 256
//fAPB1 时钟一般为 54Mhz：
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

}

//SPI2 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
}
```

这里实现了 4 个函数，分别是 SPI2 初始化函数（SPI2_Init）、SPI 初始化回调函数 HAL_SPI_MspInit，SPI2 通信速度设置函数（SPI2_SetSpeed）和 SPI2 读写操作函数（SPI2_ReadWriteByte）。SPI2_Init 函数是按 41.1.1 小节的步骤 2 讲解的调用 SPI 初始化函数 HAL_SPI_Init 来实现 SPI2 初始化。SPI 初始化回调函数 HAL_SPI_MspInit 内部主要根据 41.1.1 小节讲解的步骤 1 来实现 SPI2 时钟使能和 IO 复用映射。SPI2_ReadWriteByte 函数主要是通过调用 HAL 库中 SPI 发送接收函数 HAL_SPI_TransmitReceive 来实现数据的发送和接收。这里我们着重看一下 SPI2_SetSpeed 函数，该函数用来设置 SPI2 的传输速度也就是波特率，SPI2 的传

输速度是通过 SPI2->CR1 寄存器的位 3-5 来设置，具体设置方法请参考《STM32F7 中文参考手册》32.9.1 小节中 CR1 寄存器描述。

接下来我们打开 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 SPI2 驱动
void NRF24L01_SPI_Init(void)
{
    __HAL_SPI_DISABLE(&SPI2_Handler);                //先关闭 SPI2
    SPI2_Handler.Init.CLKPolarity=SPI_POLARITY_LOW;//同步时钟的空闲状态为低电平
    SPI2_Handler.Init.CLKPhase=SPI_PHASE_1EDGE; //同步时钟第 1 个跳变沿采样数据
    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_GPIOG_CLK_ENABLE();              //开启 GPIOG 时钟
    __HAL_RCC_GPIOI_CLK_ENABLE();              //开启 GPIOI 时钟

    GPIO_Initure.Pin=GPIO_PIN_10|GPIO_PIN_12; //PG10,12
    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(GPIOG,&GPIO_Initure);           //初始化

    GPIO_Initure.Pin=GPIO_PIN_11;                 //PI11
    GPIO_Initure.Mode=GPIO_MODE_INPUT;      //输入
    HAL_GPIO_Init(GPIOI,&GPIO_Initure);          //初始化

    SPI2_Init();                    //初始化 SPI2
    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 6.75Mhz
    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
    NRF24L01_CE(0);
    NRF24L01_Write_Buf(WR_TX_PLOAD,txbuf,TX_PLOAD_WIDTH);/
    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);
    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);
    sta=NRF24L01_Read_Reg(STATUS);                  //读取状态寄存器的值
    NRF24L01_Write_Reg(NRF_WRITE_REG+STATUS,sta);
    if(sta&RX_OK)//接收到数据
    {
        NRF24L01_Read_Buf(RD_RX_PLOAD,rxbuf,RX_PLOAD_WIDTH);//读取数据
        NRF24L01_Write_Reg(FLUSH_RX,0xff);   //清除 RX FIFO 寄存器
        return 0;
    }
    return 1;//没收到任何数据
}
//该函数初始化 NRF24L01 到 RX 模式
//设置 RX 地址,写 RX 数据宽度,选择 RF 频道,波特率和 LNA HCURR
//当 CE 变高后,即进入 RX 模式,并可以接收数据了
void NRF24L01_RX_Mode(void)
{
    NRF24L01_CE(0);
    NRF24L01_Write_Buf(NRF_WRITE_REG+RX_ADDR_P0,(u8*)RX_ADDRESS,
                                        RX_ADR_WIDTH);//写 RX 节点地址
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_AA,0x01);
                                        //使能通道 0 的自动应答
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_RXADDR,0x01);
                                        //使能通道 0 的接收地址
    NRF24L01_Write_Reg(NRF_WRITE_REG+RF_CH,40);
                                        //设置 RF 通信频率
    NRF24L01_Write_Reg(NRF_WRITE_REG+RX_PW_P0,RX_PLOAD_WIDTH);
                                        //选择通道 0 的有效数据宽度
    NRF24L01_Write_Reg(NRF_WRITE_REG+RF_SETUP,0x0f);
                                //设置 TX 发射参数,0db 增益,2Mbps,低噪声增益开启
    NRF24L01_Write_Reg(NRF_WRITE_REG+CONFIG, 0x0f);
                //配置基本工作模式的参数;PWR_UP,EN_CRC,16BIT_CRC,接收模式
    NRF24L01_CE(1); //CE 为高,进入接收模式
}
//该函数初始化 NRF24L01 到 TX 模式
//设置 TX 地址,写 TX 数据宽度,设置 RX 自动应答的地址,填充 TX 发送数据,
 //选择 RF 频道,波特率和 LNA HCURR
//PWR_UP,CRC 使能
//当 CE 变高后,即进入 RX 模式,并可以接收数据了
//CE 为高大于 10us,则启动发送.
void NRF24L01_TX_Mode(void)
```

```
{
    NRF24L01_CE(0);
    NRF24L01_Write_Buf(NRF_WRITE_REG+TX_ADDR,(u8*)TX_ADDRESS,
                                        TX_ADR_WIDTH);//写 TX 节点地址
    NRF24L01_Write_Buf(NRF_WRITE_REG+RX_ADDR_P0,(u8*)RX_ADDRESS,
                RX_ADR_WIDTH); //设置 TX 节点地址,主要为了使能 ACK
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_AA,0x01);      //使能通道 0 的自动应答
    NRF24L01_Write_Reg(NRF_WRITE_REG+EN_RXADDR,0x01); //使能通道 0 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
}
```

此部分代码完成了对 NRF24L01 的初始化、模式设置（发送/接收）、数据读写等操作。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，该函数设置的是 SCK 空闲时为高，但是 NRF24L01 的 SPI 通信时序如图 40.3.1 所示：

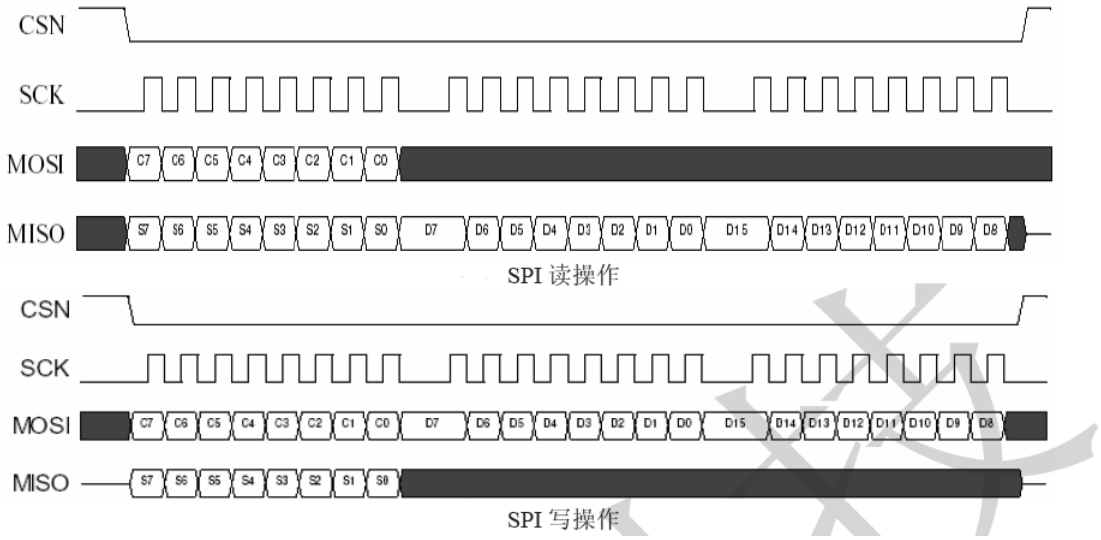


图 40.3.1 NRF24L01 读写操作时序

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；

```
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; //同步时钟的第 1 个跳变沿数据被采样
```

接下来我们看看 24101.h 头文件部分内容：

```
#ifndef __24L01_H
#define __24L01_H
```

```
#include "sys.h"
//NRF24L01 寄存器操作命令
#define READ_REG        0x00        //读配置寄存器,低 5 位为寄存器地址
......//省略部分定义
#define FIFO_STATUS     0x17        //FIFO 状态寄存器;bit0,RX FIFO 寄存器空标志;
//bit1,RX FIFO 满标志;bit2,3,保留  bit4,TX FIFO 空标志;bit5,TX FIFO 满标志;
//bit6,1, 循环发送上一数据包.0,不循环;
//24L01 操作线
#define NRF24L01_CE(n)    (n?HAL_GPIO_WritePin(GPIOG,GPIO_PIN_12,
    GPIO_PIN_SET):HAL_GPIO_WritePin(GPIOG,GPIO_PIN_12,GPIO_PIN_RESET))
                                                    //24L01 片选信号
#define NRF24L01_CSN(n)   (n?HAL_GPIO_WritePin(GPIOG,GPIO_PIN_10,\
    GPIO_PIN_SET):HAL_GPIO_WritePin(GPIOG,GPIO_PIN_10,GPIO_PIN_RESET))
                                                    //SPI 片选信号
#define NRF24L01_IRQ      HAL_GPIO_ReadPin(GPIOI,GPIO_PIN_11)//IRQ 主机数据输入
//24L01 发送接收数据宽度定义
#define TX_ADR_WIDTH      5         //5 字节的地址宽度
#define RX_ADR_WIDTH      5         //5 字节的地址宽度
#define TX_PLOAD_WIDTH    32        //32 字节的用户数据宽度
#define RX_PLOAD_WIDTH    32        //32 字节的用户数据宽度

void NRF24L01_Init(void);           //初始化
......//省略部分函数申明
u8 NRF24L01_RxPacket(u8 *rxbuf);    //接收一个包的数据
#endif
```

部分代码，主要定义了一些 24L01 的命令字（这里我们省略了一部分），以及函数声明，这里还通过 TX_PLOAD_WIDTH 和 RX_PLOAD_WIDTH 决定了发射和接收的数据宽度，也就是我们每次发射和接受的有效字节数。NRF24L01 每次最多传输 32 个字节，再多的字节传输则需要多次传送。

最后我们看看主函数：

```
int main(void)
{
    u8 key,mode;
    u16 t=0;
    u8 tmp_buf[33];
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    …//此处省略部分代码
    NRF24L01_Init();                //初始化 NRF24L01

    while(NRF24L01_Check())
```

```
{
    LCD_ShowString(30,130,200,16,16,"NRF24L01 Error");
    delay_ms(200);
    LCD_Fill(30,130,239,130+16,WHITE);
    delay_ms(200);
}
LCD_ShowString(30,130,200,16,16,"NRF24L01 OK");
while(1)
{
    key=KEY_Scan(0);
    if(key==KEY0_PRES)
    {
        mode=0;break;
    }else if(key==KEY1_PRES)
    {
        mode=1;break;
    }
    t++;
    if(t==100)LCD_ShowString(10,150,230,16,16,"KEY0:RX_Mode
                                    KEY1:TX_Mode"); //闪烁显示提示信息
    if(t==200)
    {
        LCD_Fill(10,150,230,150+16,WHITE);
        t=0;
    }
    delay_ms(5);
}
LCD_Fill(10,150,240,166,WHITE);//清空上面的显示
POINT_COLOR=BLUE;//设置字体为蓝色
if(mode==0)//RX 模式
{
    LCD_ShowString(30,150,200,16,16,"NRF24L01 RX_Mode");
    LCD_ShowString(30,170,200,16,16,"Received DATA:");
    NRF24L01_RX_Mode();
    while(1)
    {
        if(NRF24L01_RxPacket(tmp_buf)==0)//一旦接收到信息,则显示出来.
        {
            tmp_buf[32]=0;//加入字符串结束符
            LCD_ShowString(0,190,lcddev.width-1,32,16,tmp_buf);
        }else delay_us(100);
        t++;
        if(t==10000)//大约 1s 钟改变一次状态
```

```
                {
                    t=0; LED0_Toggle;
                }
            };
        }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_Toggle;
                delay_ms(1500);
            };
        }
    }
```

以上代码，我们就实现了 40.2 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。

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 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 41.4.1 所示的内容（假定 NRF24L01 模块已经接上开发板）：

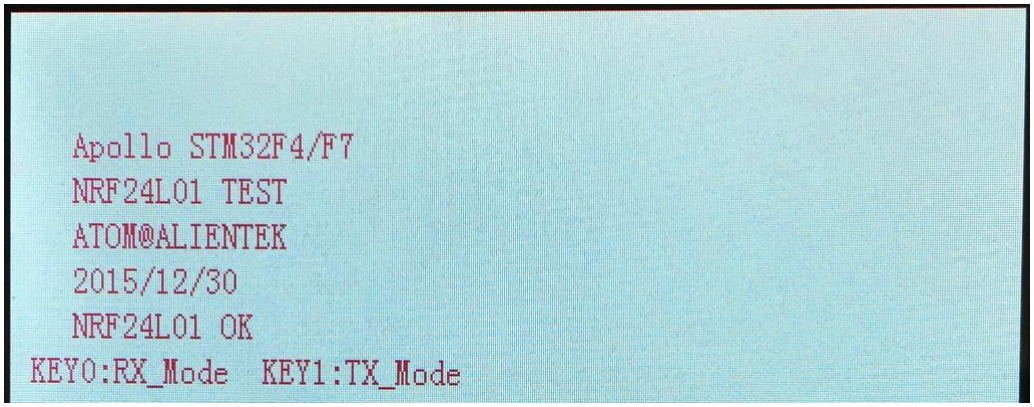


图 41.4.1 选择工作模式界面

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 41.4.2 和图 41.4.3 所示：

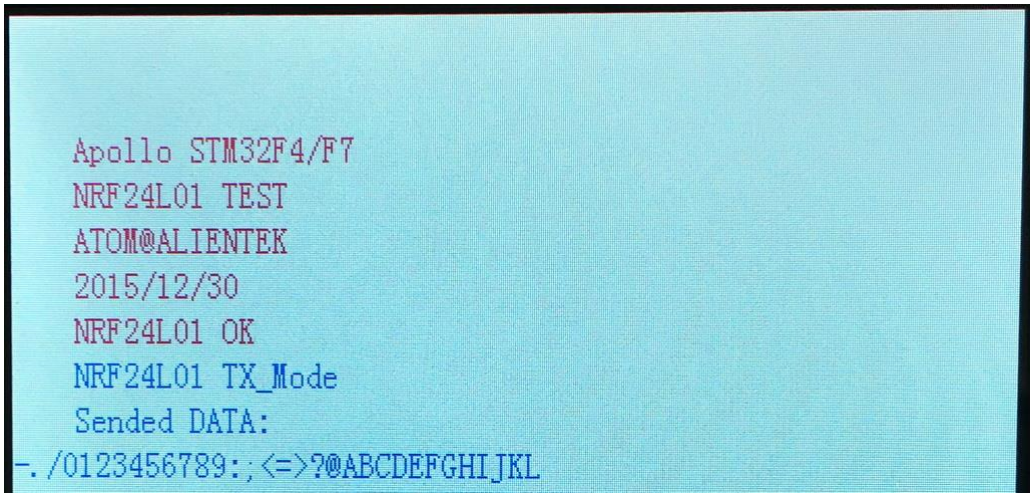


图 41.4.2 开发板 A 发送数据

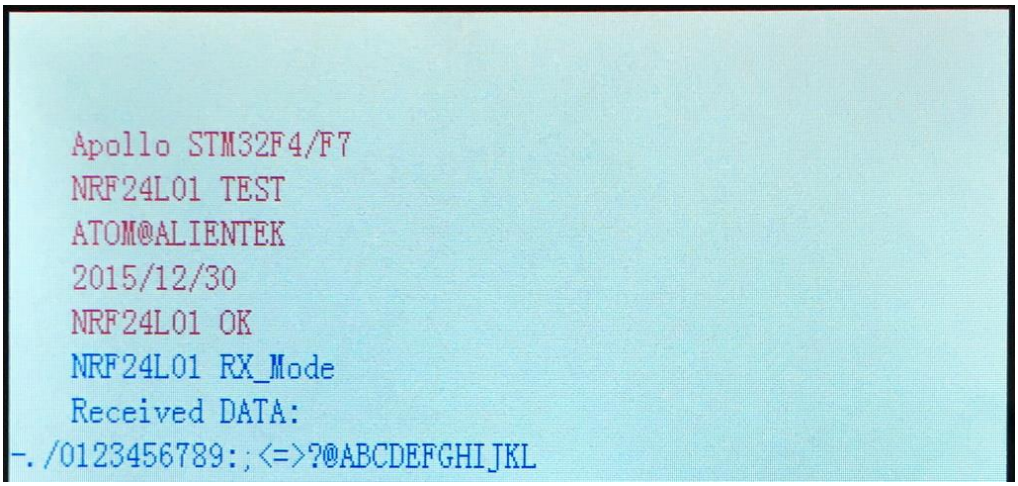


图 41.4.3 开发板 B 接收数据

图 41.4.2 来自开发板 A，工作在发送模式。图 41.4.3 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。

第四十二章 FLASH 模拟 EEPROM 实验

STM32F767 本身没有自带 EEPROM，但是 STM32F767 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 STM32F767 内部的 FLASH 来实现第三十三章实验类似的效果，不过这次我们是将数据直接存放在 STM32F767 内部，而不是存放在 W25Q256。本章分为如下几个部分：

42.1 STM32F767 FLASH 简介
42.2 硬件设计
42.3 软件设计
42.4 下载验证

42.1 STM32F767 FLASH 简介

不同型号的 STM32F7xx 芯片，其 FLASH 容量也有所不同，有 1MB/2MB 等不同容量的产品。阿波罗 STM32F767 开发板选择的 STM32F767IGT6 的 FLASH 容量为 1024K 字节（1MB），STM32F767IGT6 的闪存模块组织如表 42.1.1 所示：

块	名称	AXIM 接口上的块基址	ICTM 接口上的块基址	大小
主存储器	扇区 0	0X0800 0000-0X0800 7FFF	0X0020 0000-0X0020 7FFF	32KB
	扇区 1	0X0800 8000-0X0800 FFFF	0X0020 8000-0X0020 FFFF	32KB
	扇区 2	0X0801 0000-0X0801 7FFF	0X0021 0000-0X0021 7FFF	32KB
	扇区 3	0X0801 8000-0X0801 FFFF	0X0021 8000-0X0021 FFFF	32KB
	扇区 4	0X0802 0000-0X0803 FFFF	0X0022 0000-0X0023 FFFF	128KB
	扇区 5	0X0804 0000-0X0807 FFFF	0X0024 0000-0X0027 FFFF	256KB
	扇区 6	0X0808 0000-0X080B FFFF	0X0028 0000-0X002B FFFF	256KB
	扇区 7	0X080C 0000-0X080F FFFF	0X002C 0000-0X002F FFFF	256KB
系统存储器		0X1FF0 0000-0X1FF0 EDBF	0X0010 0000-0X0010 EDBF	60KB
OTP 区域		0X1FF0 F000-0X1FF0 F41F	0X0010 F000-0X0010 F41F	1056B
选项字节		0X1FFF 0000-0X1FFF 001F		32B

表 42.1.1 STM32F767IGT6 闪存模块组织

STM32F767IGT6 的闪存模块由：主存储器、系统存储器、OPT 区域和选项字节等 4 部分组成。

主存储器，该部分用来存放代码和数据常数（如 const 类型的数据）。它可以分为 1 个 Bank 或者 2 个 Bank，可以通过选项字节的 nDBANK 位来设置，默认是一个 Bank 的，表 42.1.1 所示即单 Bank 的闪存组织结构。关于双 Bank 的详细设置，请参考《STM32F7xx 参考手册》第 3 章的相关内容。本章我们仅以单 Bank 做介绍。

在单 Bank 模式下，STM32F767 的主存储器被分为 8 个扇区，前 4 个扇区为 32KB 大小，第五个扇区是 128KB 大小，剩下的 3 个扇区都是 256KB 大小，总共 1M 字节。

因为 STM32F7 的 FLASH 访问路径有两条：AXIM 和 ITCM，对应不同的地址映射，表中我们列出了这两条不同访问路径下的扇区地址范围。我们一般选择 AXIM 接口访问 FLASH，其主存储器的起始地址就是 0X08000000。

系统存储器，这个主要用来存放 STM32F767 的 bootloader 代码，此代码是出厂的时候就固化在 STM32F767 里面了，专门来给主存储器下载代码的。当 B0 接 V3.3，默认从系统存储器启

动。

OTP 区域，即一次性可编程区域，共 1056 字节，被划分为 16 个 64 字节的 OTP 数据块和 1 个 16 字节的 OTP 锁定块。OTP 数据块和锁定块均无法擦除。锁定块中包含 16 字节的 LOCKBi (0≤i≤15)，用于锁定相应的 OTP 数据块（块 0 到 15）。每个 OTP 数据块均可编程，除非相应的 OTP 锁定字节编程为 0x00。锁定字节的值只能是 0x0 和 0xFF，否则这些 OTP 字节无法正确使用。

闪存存储器接口寄存器，该部分用于控制闪存读写等，是整个闪存模块的控制机构。

在执行闪存写操作时，任何对闪存的读操作都会锁住总线，在写操作完成后读操作才能正确地进行；既在进行写或擦除操作时，不能进行代码或数据的读取操作。

闪存的读取

为了准确读取 Flash 数据，必须根据 CPU 时钟 (HCLK) 频率和器件电源电压在 Flash 存取控制寄存器 (FLASH_ACR) 中正确地设置等待周期数 (LATENCY)。Flash 等待周期与 CPU 时钟频率之间的对应关系，如表 42.1.2 所示：

等待周期 (WS) (LATENCY)	HCLK (MHz)			
	电压范围 2.7 V - 3.6 V	电压范围 2.4 V - 2.7 V	电压范围 2.1 V - 2.4 V	电压范围 1.8 V - 2.1 V
0 WS（1 个 CPU 周期）	0 < HCLK ≤ 30	0 < HCLK ≤ 24	0 < HCLK ≤ 22	0 < HCLK ≤ 20
1 WS（2 个 CPU 周期）	30 < HCLK ≤ 60	24 < HCLK ≤ 48	22 < HCLK ≤ 44	20 < HCLK ≤ 40
2 WS（3 个 CPU 周期）	60 < HCLK ≤ 90	48 < HCLK ≤ 72	44 < HCLK ≤ 66	40 < HCLK ≤ 60
3 WS（4 个 CPU 周期）	90 < HCLK ≤ 120	72 < HCLK ≤ 96	66 < HCLK ≤ 88	60 < HCLK ≤ 80
4 WS（5 个 CPU 周期）	120 < HCLK ≤ 150	96 < HCLK ≤ 120	88 < HCLK ≤ 110	80 < HCLK ≤ 100
5 WS（6 个 CPU 周期）	150 < HCLK ≤ 180	120 < HCLK ≤ 144	110 < HCLK ≤ 132	100 < HCLK ≤ 120
6 WS（7 个 CPU 周期）	180 < HCLK ≤ 210	144 < HCLK ≤ 168	132 < HCLK ≤ 154	120 < HCLK ≤ 140
7 WS（8 个 CPU 周期）	210 < HCLK ≤ 216	168 < HCLK ≤ 192	154 < HCLK ≤ 176	140 < HCLK ≤ 160
8 WS（9 个 CPU 周期）	-	192 < HCLK ≤ 216	176 < HCLK ≤ 198	160 < HCLK ≤ 180
9 WS（10 个 CPU 周期）	-	-	198 < HCLK ≤ 216	-

表 42.1.2 CPU 时钟（HCLK）频率对应的 FLASH 等待周期表

等待周期通过 FLASH_ACR 寄存器的 LATENCY[3:0]四三个位设置。系统复位后，CPU 时钟频率为内部 16M RC 振荡器，LATENCY 默认是 0，即 1 个等待周期。供电电压，我们一般是 3.3V，所以，在我们设置 216Mhz 频率作为 CPU 时钟之前，必须先设置 LATENCY 为 7，即 8 个等待周期，否则 FLASH 读写可能出错，导致死机。

正常工作时（216Mhz），虽然 FLASH 需要 8 个 CPU 等待周期，但是由于 STM32F767 具有自适应实时存储器加速器(ART Accelerator)，通过指令缓存存储器，预取指令，实现相当于 0 FLASH 等待的运行速度。

STM23F7 的 FLASH 读取是很简单的。例如，我们要从地址 addr，读取一个字（一个字为 32 位），可以通过如下的语句读取：

```
data=*(vu32*)addr;
```

将 addr 强制转换为 vu32 指针，然后取该指针所指向的地址的值，即得到了 addr 地址的值。类似的，将上面的 vu32 改为 vu8，即可读取指定地址的一个字节。相对 FLASH 读取来说，STM32F767 FLASH 的写就复杂一点了，下面我们介绍 STM32F767 闪存的编程和擦除。

闪存的编程和擦除

执行任何 Flash 编程操作（擦除或编程）时，CPU 时钟频率 (HCLK)不能低于 1 MHz。如

果在 Flash 操作期间发生器件复位，无法保证 Flash 中的内容。

在对 STM32F767 的 Flash 执行写入或擦除操作期间，任何读取 Flash 的尝试都会导致总线阻塞。只有在完成编程操作后，才能正确处理读操作。这意味着，写/擦除操作进行期间不能从 Flash 中执行代码或数据获取操作。

STM32F767 的闪存编程由 7 个 32 位寄存器控制，他们分别是：

- FLASH 访问控制寄存器(FLASH_ACR)
- FLASH 秘钥寄存器(FLASH_KEYR)
- FLASH 选项秘钥寄存器(FLASH_OPTKEYR)
- FLASH 状态寄存器(FLASH_SR)
- FLASH 控制寄存器(FLASH_CR)
- FLASH 选项控制寄存器(FLASH_OPTCR)
- FLASH 选项控制寄存器 1(FLASH_OPTCR1)

STM32F767 复位后，FLASH 编程操作是被保护的，不能写入 FLASH_CR 寄存器；通过写入特定的序列（0X45670123 和 0XCDEF89AB）到 FLASH_KEYR 寄存器才可解除写保护，只有在写保护被解除后，我们才能操作相关寄存器。

FLASH_CR 的解锁序列为：

1， 写 0X45670123 到 FLASH_KEYR

2， 写 0XCDEF89AB 到 FLASH_KEYR

通过这两个步骤，即可解锁 FLASH_CR，如果写入错误，那么 FLASH_CR 将被锁定，直到下次复位后才可以再次解锁。

STM32F767 闪存的编程位数可以通过 FLASH_CR 的 PSIZE 字段配置，PSIZE 的设置必须和电源电压匹配，见表：42.1.2：

	电压范围 **2.7 - 3.6 V**（使用外部 V_{PP}）	电压范围 **2.7 - 3.6 V**	电压范围 **2.4 - 2.7 V**	电压范围 **2.1 - 2.4 V**	电压范围 **1.8 V - 2.1 V**
并行位数	x64	x32	x16		x8
PSIZE(1:0)	11	10	01		00

表 42.1.2 编程/擦除并行位数与电压关系表

由于我们开发板用的电压是 3.3V，所以 PSIZE 必须设置为 10，即 32 位并行位数。擦除或者编程，都必须以 32 位为基础进行。

STM32F767 的 FLASH 在编程的时候，也必须要求其写入地址的 FLASH 是被擦除了的（也就是其值必须是 0XFFFFFFFF），否则无法写入。STM32F767 的标准编程步骤如下：

1，检查 FLASH_SR 中的 BSY 位，确保当前未执行任何 FLASH 操作。

2，将 FLASH_CR 寄存器中的 PG 位置 1，激活 FLASH 编程。

3，针对所需存储器地址（主存储器块或 OTP 区域内）执行数据写入操作：

—并行位数为 x8 时按字节写入（PSIZE=00）

—并行位数为 x16 时按半字写入（PSIZE=01）

—并行位数为 x32 时按字写入（PSIZE=02）

—并行位数为 x64 时按双字写入（PSIZE=03）

4，等待 BSY 位清零，完成一次编程。

按以上四步操作，就可以完成一次 FLASH 编程。不过有几点要注意：1，编程前，要确保要写如地址的 FLASH 已经擦除。2，要先解锁（否则不能操作 FLASH_CR）。3，编程操作对 OPT 区域也有效，方法一模一样。

我们在 STM32F767 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 STM32F767 的闪存擦除，STM32F767 的闪存擦除分为两种：扇区擦除和整片擦除。

扇区擦除步骤如下：

1，检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁

2，检查 FLASH_SR 寄存器中的 BSY 位，确保当前未执行任何 FLASH 操作

3，在 FLASH_CR 寄存器中，将 SER 位置 1，并从主存储块的 12 个扇区中选择要擦除的扇区 (SNB)

4，将 FLASH_CR 寄存器中的 STRT 位置 1，触发擦除操作

5，等待 BSY 位清零

经过以上五步，就可以擦除某个扇区。本章，我们只用到了 STM32F767 的扇区擦除功能，整片擦除功能我们在这里就不介绍了，想了解的朋友可以看《STM32F7 中文参考手册》第 3.3.6 节。

通过以上了解，我们基本上知道了 STM32F767 闪存的读写所要执行的步骤了，接下来，我们看看与读写相关的寄存器说明。

第一个介绍的是 FLASH 访问控制寄存器：FLASH_ACR。该寄存器各位描述如图 42.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	ARTRST	Res.	ARTEN	PRFTEN	Res.	Res.	Res.	Res.	LATENCY			
				rw		rw	rw					rw	rw	rw	rw

图 42.1.2 FLASH_ACR 寄存器各位描述

这里，我们重点看 LATENCY[3:0]这四个位，这四个位，必须根据我们 MCU 的工作电压和频率，来进行正确的设置，否则，可能死机，设置规则见表 42.1.2。其他 ARTEN（使能 ART 加速）和 PRFTEN（预取使能）这两个位也比较重要，为了达到最佳性能，这两个位我们一般都设置为 1 即可。

第二个介绍的是 FLASH 秘钥寄存器：FLASH_KEYR。该寄存器各位描述如图 42.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEY[31:16]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEY[15:0]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位 31:0 **FKEYR**：FPEC 密钥 (FPEC key)

要将 FLASH_CR 寄存器解锁并允许对其执行编程/擦除操作，必须顺序编程以下值：

a) KEY1 = 0x45670123

b) KEY2 = 0xCDEF89AB

图 42.1.3 FLASH_KEYR 寄存器各位描述

该寄存器主要用来解锁 FLASH_CR，必须在该寄存器写入特定的序列（KEY1 和 KEY2）解锁后，才能对 FLASH_CR 寄存器进行写操作。

第三个要介绍的是 FLASH 控制寄存器：FLASH_CR。该寄存器的各位描述如图 42.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LOCK	Res.	Res.	Res.	Res.	Res.	ERRIE	EOPIE	Res.	Res.	Res.	Res.	Res.	Res.	Res.	STRT
rs						rw	rw								rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	PSIZE[1:0]		Res.	SNB[3:0]				MER	SER	PG
						rw	rw		rw	rw	rw	rw	rw	rw	rw

图 42.1.4 FLASH_CR 寄存器各位描述

该寄存器我们本章只用到了它的 LOCK、STRT、PSIZE[1:0]、SNB[3:0]、SER 和 PG 等位。

LOCK 位，该位用于指示 FLASH_CR 寄存器是否被锁住，该位在检测到正确的解锁序列后，硬件将其清零。在一次不成功的解锁操作后，在下次系统复位之前，该位将不再改变。

STRT 位，该位用于开始一次擦除操作。在该位写入 1 ，将执行一次擦除操作。

PSIZE[1:0]位，用于设置编程宽度，3.3V 时，我们设置 PSIZE =2 即可。

SNB[3:0]位，这 4 个位用于选择要擦除的扇区编号，取值范围为 0~15。

SER 位，该位用于选择扇区擦除操作，在扇区擦除的时候，需要将该位置 1。

PG 位，该位用于选择编程操作，在往 FLASH 写数据的时候，该位需要置 1。

FLASH_CR 的其他位，我们就不在这里介绍了，请大家参考《STM32F7 中文参考手册》第 3.7.5 节。

最后要介绍的是 FLASH 状态寄存器：FLASH_SR。该寄存器各位描述如图 42.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															BSY
															r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								PGSERR	PGPERR	PGAERR	WRPERR	Reserved		OPERR	EOP
								rc_w1	rc_w1	rc_w1	rc_w1			rc_w1	rc_w1

图 42.1.5 FLASH_SR 寄存器各位描述

该寄存器我们主要用了其 BSY 位，当该位位 1 时，表示正在执行 FLASH 操作。当该位为 0 时，表示当前未执行任何 FLASH 操作。

关于 STM32F767 FLASH 的介绍，我们就介绍到这。更详细的介绍，请参考《STM32F7 中文参考手册》第三章。下面我们讲解使用 STM32F7 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 stm32f7xx_hal_flash.c/stm32f7xx_hal_flash_ex.c 以及头文件 stm32f7xx_hal_flash.h/stm32f7xx_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 stm32f7xx_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
#if defined (FLASH_OPTCR_nDBANK)
  uint32_t Banks;             //擦除的 Bank 编号
#endif
  uint32_t Sector;            //擦除的 sector 号
  uint32_t NbSectors;         //擦除的 sector 数量
  uint32_t VoltageRange;      //电压范围
} FLASH_EraseInitTypeDef;
```

成员变量 TypeErase 用来设置擦除类型，是 Sector 擦除还是 BANK 级别的批量擦除，取值为 FLASH_TYPEERASE_SECTORS 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 Sector，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 Sector 用来设置要擦除的 Sector 编号。成员变量 NbSectors 用来设置要擦除的 Sector 数量。成员变量 VoltageRange 用来设置电压范围，一共有四个值可选 FLASH_VOLTAGE_RANGE_1~ FLASH_VOLTAGE_RANGE_4，分别对应表 41.1.2 的电压范围，这里我们使用的是 3.3V，所以选择 FLASH_VOLTAGE_RANGE_3 即可。

扇区擦除的实例代码如下：

```
FlashEraseInit.TypeErase=FLASH_TYPEERASE_SECTORS; //擦除类型，扇区擦除
FlashEraseInit.Sector=3;        //擦除的扇区号
FlashEraseInit.NbSectors=1;   //一次只擦除一个扇区
FlashEraseInit.VoltageRange=FLASH_VOLTAGE_RANGE_3; //电压范围 2.7~3.6V
HAL_FLASHEx_Erase(&FlashEraseInit,&SectorError);//进行扇区擦除操作
```

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一个字的函数：

```
u32 STMFLASH_ReadWord(u32 faddr)
{
    return *(vu32*)faddr;
}
```

42.2 硬件设计

本章实验功能简介：开机的时候先显示一些提示信息，然后在主循环里面检测两个按键，其中 1 个按键（KEY1）用来执行写入 FLASH 的操作，另外一个按键（KEY0）用来执行读出操作，在 LCD 模块上显示相关信息。同时用 DS0 提示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0

2） KEY1 和 KEY0 按键

3） LCD 模块

4） STM32F767 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。

42.3 软件设计

打开 FLASH 模拟 EEPROM 实验工程可以看到，工程的 HARDWARE 分组下新添加了源文件 stm32flash.c，同时包含了对应的头文件 stm32flash.h。同时我们还引入了 HAL 库 flash 操作源文件 stm32f7xx_hal_flash.c/stm32f7xx_hal_flash_ex.c 和头文件 stm32f7xx_hal_flash.h/stm32f7xx_hal_flash_ex.h。

打开 stmflash.c 文件，代码如下：

```
//读取指定地址的字(32 位数据)
//faddr:读地址
//返回值:对应数据.
u32 STMFLASH_ReadWord(u32 faddr)
{
    return *(__IO uint32_t *)faddr;
}

//获取某个地址所在的 flash 扇区
//addr:flash 地址
//返回值:0~11,即 addr 所在的扇区
uint16_t STMFLASH_GetFlashSector(u32 addr)
{
    if(addr<ADDR_FLASH_SECTOR_1)return FLASH_SECTOR_0;
    else if(addr<ADDR_FLASH_SECTOR_2)return FLASH_SECTOR_1;
    else if(addr<ADDR_FLASH_SECTOR_3)return FLASH_SECTOR_2;
    else if(addr<ADDR_FLASH_SECTOR_4)return FLASH_SECTOR_3;
    else if(addr<ADDR_FLASH_SECTOR_5)return FLASH_SECTOR_4;
    else if(addr<ADDR_FLASH_SECTOR_6)return FLASH_SECTOR_5;
```

```
    else if(addr<ADDR_FLASH_SECTOR_7)return FLASH_SECTOR_6;
    return FLASH_SECTOR_7;
}

//从指定地址开始写入指定长度的数据
//特别注意:因为 STM32F7 的扇区实在太大,没办法本地保存扇区数据,所以本函数
//         写地址如果非 0XFF,那么会先擦除整个扇区且不保存扇区数据.所以
//         写非 0XFF 的地址,将导致整个扇区数据丢失.建议写之前确保扇区里
//         没有重要数据,最好是整个扇区先擦除了,然后慢慢往后写.
//该函数对 OTP 区域也有效!可以用来写 OTP 区!
//OTP 区域地址范围:0X1FF0F000~0X1FF0F41F
//WriteAddr:起始地址(此地址必须为 4 的倍数!!)
//pBuffer:数据指针
//NumToWrite:字(32 位)数(就是要写入的 32 位数据的个数.)
void STMFLASH_Write(u32 WriteAddr,u32 *pBuffer,u32 NumToWrite)
{
    FLASH_EraseInitTypeDef FlashEraseInit;
    HAL_StatusTypeDef FlashStatus=HAL_OK;
    u32 SectorError=0;
    u32 addrx=0;
    u32 endaddr=0;
    if(WriteAddr<STM32_FLASH_BASE||WriteAddr%4)return;    //非法地址

    HAL_FLASH_Unlock();                    //解锁
    addrx=WriteAddr;                       //写入的起始地址
    endaddr=WriteAddr+NumToWrite*4;   //写入的结束地址

    if(addrx<0X1FF00000)
    {
        while(addrx<endaddr)//扫清一切障碍.(对非 FFFFFFFF 的地方,先擦除)
        {
            if(STMFLASH_ReadWord(addrx)!=0XFFFFFFFF)
                                    //有非 0XFFFFFFFF 的地方,要擦除这个扇区
            {
                FlashEraseInit.TypeErase=FLASH_TYPEERASE_SECTORS;   //扇区擦除
                FlashEraseInit.Sector=STMFLASH_GetFlashSector(addrx);//要擦除的扇区
                FlashEraseInit.NbSectors=1; //一次只擦除一个扇区
                FlashEraseInit.VoltageRange=FLASH_VOLTAGE_RANGE_3;
                                                    //电压范围，VCC=2.7~3.6V 之间!!
                if(HAL_FLASHEx_Erase(&FlashEraseInit,&SectorError)!=HAL_OK)
                {
                    break;//发生错误了
                }
```

```
                    SCB_CleanInvalidateDCache();//清除无效的 D-Cache
                }else addrx+=4;
                FLASH_WaitForLastOperation(FLASH_WAITETIME); //等待上次操作完成
            }
        }
        FlashStatus=FLASH_WaitForLastOperation(FLASH_WAITETIME); //等待上次操作完成
        if(FlashStatus==HAL_OK)
        {
            while(WriteAddr<endaddr)//写数据
            {
                if(HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD,WriteAddr,
                                                        *pBuffer)!=HAL_OK)//写入数据
                {
                    break;    //写入异常
                }
                WriteAddr+=4;
                pBuffer++;
            }
        }
        HAL_FLASH_Lock();                //上锁
    }

    //从指定地址开始读出指定长度的数据
    //ReadAddr:起始地址
    //pBuffer:数据指针
    //NumToRead:字(32 位)数
    void STMFLASH_Read(u32 ReadAddr,u32 *pBuffer,u32 NumToRead)
    {
        u32 i;
        for(i=0;i<NumToRead;i++)
        {
            pBuffer[i]=STMFLASH_ReadWord(ReadAddr);//读取 4 个字节.
            ReadAddr+=4;//偏移 4 个字节.
        }
    }
```

该文件代码，所调用的 HAL 库函数在 41.1 小节都已经详细讲解。这里我们重点介绍一下 STMFLASH_Write 函数，该函数用于在 STM32F7 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 32 章的 W25QXX_Flash_Write 函数，不过该函数使用的时候，有几个要注意要注意：

1， 写入地址必须是用户代码区以外的地址。

2， 写入地址必须是 4 的倍数。

3， 对 OTP 区域编程也有效。

第 1 点比较好理解，如果把用户代码给擦除了，可想而知你运行的程序可能就被废了，从

而很可能出现死机的情况。不过，因为 STM32F767 的扇区都比较大（最少 32K，大的 256K），所以本函数不缓存要擦除的扇区内容，也就是如果要擦除，那么就是整个扇区擦除，所以建议大家使用该函数的时候，写入地址定位到用户代码占用扇区以外的扇区，比较保险。

第 2 点则是 3.3V 时，设置 PSIZE=2 所决定的，每次必须写入 32 位，即 4 字节，所以地址必须是 4 的倍数。第 3 点，该函数对 OTP 区域的操作同样有效，所以大家要写 OTP 字节，也可以直接通过该函数写入，不过注意 OTP 是一次写入的，无法擦除，所以，一般不要写 OTP 字节。

关于 STMFLASH_GetFlashSector 函数，这个就比较好理解了，根据地址确定其 sector 编号。其他函数我们就不做介绍了。

在 STMFLASH_Write 函数里面，我们调用了 SCB_CleanInvalidateDCache 函数，用于回写数据到 SRAM，并重新获取 D Cache 数据，如果去掉，将导致死循环。

对于头文件 stmflash.h，这里面有一点提一下，就是我们定义了从 ADDR_FLASH_SECTOR_0~ ADDR_FLASH_SECTOR_19 等一系列宏定义标识符，实际上这些标识符的值就是对应的 sector 的起始地址值，相信也比较好理解。

最后我们打开 main.c 文件，代码如下：

```
//要写入到 STM32 FLASH 的字符串数组
const u8 TEXT_Buffer[]={"STM32 FLASH TEST"};
#define TEXT_LENTH sizeof(TEXT_Buffer)                    //数组长度
#define SIZE TEXT_LENTH/4+((TEXT_LENTH%4)?1:0)

#define FLASH_SAVE_ADDR  0X08020000
  //设置 FLASH 保存地址(必须为 4 的倍数，且所在扇区,要大于本代码所占用到的扇区.
  //否则,写操作的时候,可能会导致擦除整个扇区,从而引起部分程序丢失.引起死机.
int main(void)
{
    u8 key=0;
    u16 i=0;
    u8 datatemp[SIZE];

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/7/14");
```

```
        LCD_ShowString(30,130,200,16,16,"KEY1:Write    KEY0:Read");
        while(1)
        {
            key=KEY_Scan(0);
            if(key==KEY1_PRES)  //KEY1 按下,写入 STM32 FLASH
            {
                LCD_Fill(0,170,239,319,WHITE);//清除半屏
                LCD_ShowString(30,170,200,16,16,"Start Write FLASH....");
                STMFLASH_Write(FLASH_SAVE_ADDR,(u32*)TEXT_Buffer,SIZE);
                LCD_ShowString(30,170,200,16,16,"FLASH Write Finished!");//提示传送完成
            }
            if(key==KEY0_PRES)  //KEY0 按下,读取字符串并显示
            {
                LCD_ShowString(30,170,200,16,16,"Start Read FLASH.... ");
                STMFLASH_Read(FLASH_SAVE_ADDR,(u32*)datatemp,SIZE);
                LCD_ShowString(30,170,200,16,16,"The Data Readed Is:    ");//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_Toggle;//提示系统正在运行
                i=0;
            }
        }
    }
```

主函数代码逻辑比较简单，当检测到按键 KEY1 按下后往 FLASH 指定地址开始的连续地址空间写入一段数据，当检测到按键 KEY0 按下后读取 FLASH 指定地址开始的连续空间数据。至此，我们的软件设计部分就结束了。

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 阿波罗 STM32 开发板上，通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 42.4.1 所示：

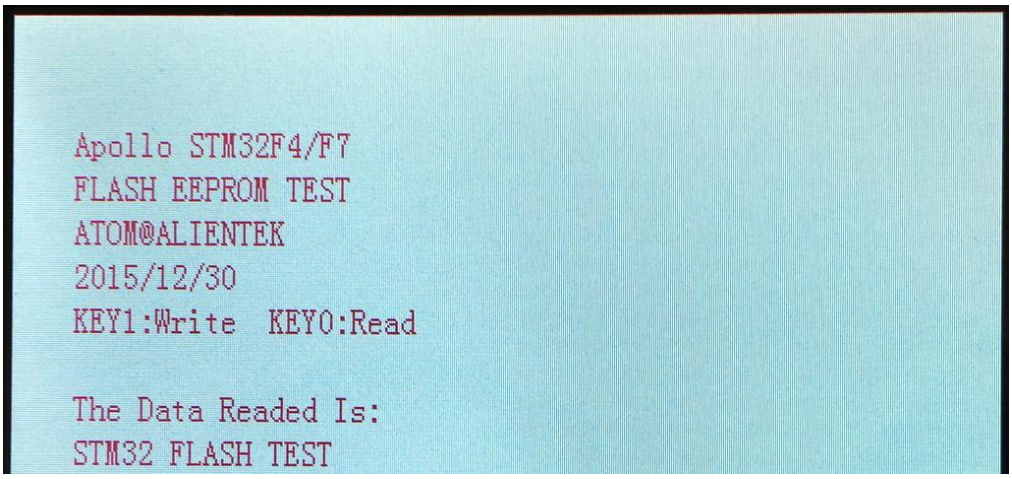


图 42.4.1 程序运行效果图

伴随 DS0 的不停闪烁，提示程序在运行。本章的测试，我们还可以借助 USMART，调用：TMFLASH_ReadWord 和 Test_Write 函数，大家可以测试下 OTP 区域的读写，注意：OTP 区域，最后 16 字节，不要乱写！！是用于锁定 OTP 数据块的的！！

另外，OTP 的一次性可编程，也并不像字面意思那样，只能写一次。而是要理解成：只能写 0，不能写 1。举个例子，你在地址：0X1FF0 F000，第一次写入 0X12345678。读出来，发现是对的，和你写入的一样。而当你在这个地址，再次写入：0X12345673 的时候，再读出来，变成了：0X12345670，不是第一次写入的值，也不是第二次写入的值，而是两次写入值相与的值，说明第二次也发生了写操作。所以，要理解成：只能写 0，不能写 1。

第四十三章 摄像头实验

ALIENTEK 阿波罗 STM32F767 开发板具有 DCMI 接口，并板载了一个摄像头接口（P7），该接口可以用来连接 ALIENTEK OV5640/OV2640 等摄像头模块。本章，我们将使用 STM32 驱动 ALIENTEK OV5640 摄像头模块，实现摄像头功能。本章分为如下几个部分：

43.1 OV5640&DCMI 简介
43.2 硬件设计
43.3 软件设计
43.4 下载验证

43.1 OV5640&DCMI 简介

本节将分为两个部分，分别介绍 OV5640 和 STM32F767 的 DCMI 接口。另外，所有 OV5640 的相关资料，都在光盘：**A 盘→7，硬件资料→OV5640 资料** 文件夹里面。

43.1.1 OV5640 简介

OV5640 是 OV（OmniVision）公司生产的一颗 1/4 寸的 CMOS QSXGA（2592*1944）图像传感器，提供了一个完整的 500W 像素摄像头解决方案，并且集成了自动对焦（AF）功能，具有非常高的性价比。

该传感器体积小、工作电压低，提供单片 QSXGA 摄像头和影像处理器的所有功能。通过 SCCB 总线控制，可以输出整帧、子采样、缩放和取窗口等方式的各种分辨率 8/10 位影像数据。该产品 QSXGA 图像最高达到 15 帧/秒（1080P 图像可达 30 帧，720P 图像可达 60 帧，QVGA 分辨率时可达 120 帧）。用户可以完全控制图像质量、数据格式和传输方式。所有图像处理功能过程包括伽玛曲线、白平衡、对比度、色度等都可以通过 SCCB 接口编程。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、拖尾、浮散等，提高图像质量，得到清晰稳定的彩色图像。

OV5640 的特点有：

- 采用 1.4μm*1.4μm 像素大小，并且使用 OmniBSI 技术以达到更高性能（高灵敏度、低串扰和低噪声）
- 自动图像控制功能：自动曝光（AEC）、自动白平衡（AWB）、自动消除灯光条纹、自动黑电平校准（ABLC）和自动带通滤波器（ABF）等。
- 支持图像质量控制：色饱和度调节、色调调节、gamma 校准、锐度和镜头校准等
- 标准的 SCCB 接口，兼容 IIC 接口
- 支持 RawRGB、RGB(RGB565/RGB555/RGB444)、CCIR656、YUV(422/420)、YCbCr（422）和压缩图像（JPEG）输出格式
- 支持 QSXGA（500W）图像尺寸输出，以及按比例缩小到其他任何尺寸
- 支持闪光灯
- 支持图像缩放、平移和窗口设置
- 支持图像压缩，即可输出 JPEG 图像数据
- 支持数字视频接口（DVP）和 MIPI 接口
- 支持自动对焦

- 自带嵌入式微处理器

OV5640 的功能框图图如图 43.1.1.1 所示：

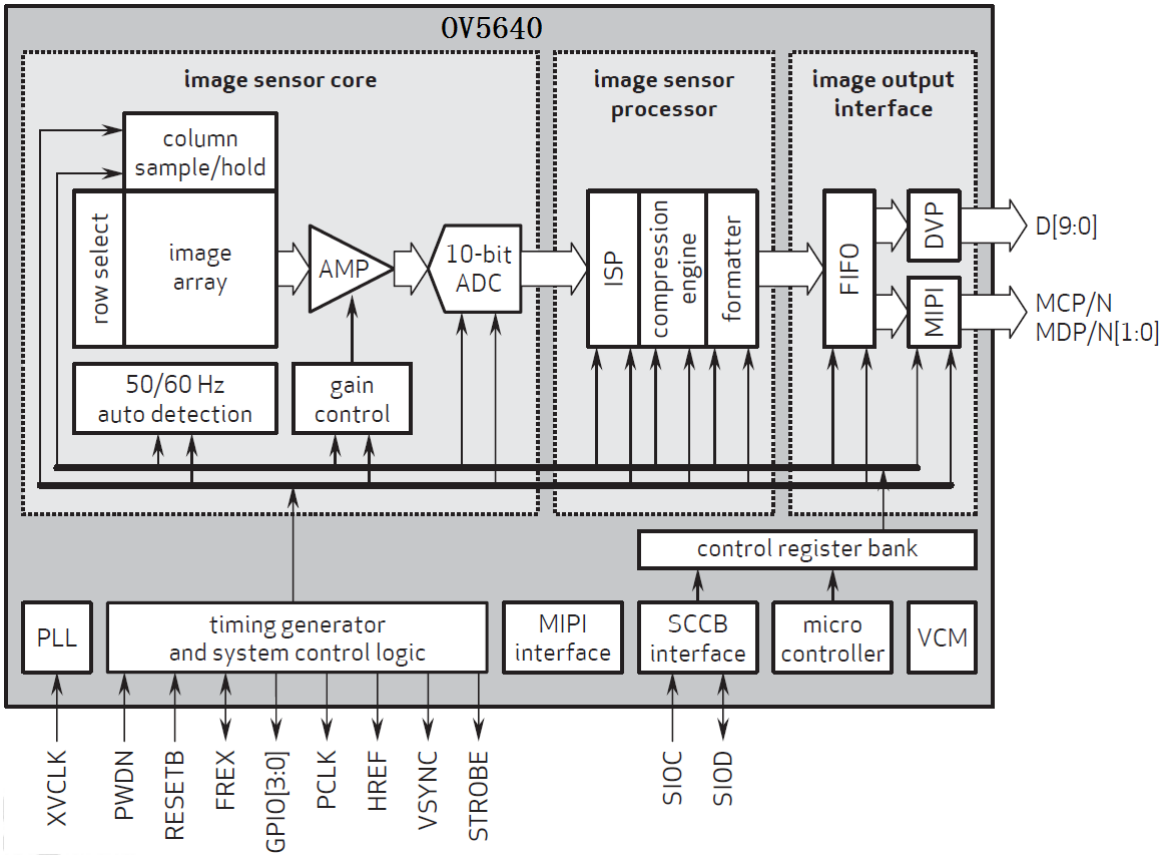


图 43.1.1.1 OV5640 功能框图

其中 image array 部分的尺寸，OV5640 的官方数据并没有给出具体的数字，其最大的有效输出尺寸为：2592*1944，即 500W 像素，我们根据官方提供的一些应用文档，发现其设置的 image array 最大为：2632*1951，所以，在接下来的介绍，我们设定其 image array 最大为 2632*1951。

1，DVP 接口说明

OV5640 支持数字视频接口（DVP）和 MIPI 接口，因为我们的 STM32F767 使用的 DCMI 接口，仅支持 DVP 接口，所以，OV5640 必须使用 DVP 输出接口，才可以连接我们的阿波罗 STM32 开发板。

OV5640 提供一个 10 位 DVP 接口(支持 8 位接法)，其 MSB 和 LSB 可以程序设置先后顺序，ALIENTEK OV5640 模块采用默认的 8 位连接方式，如图 43.1.1.2 所示：

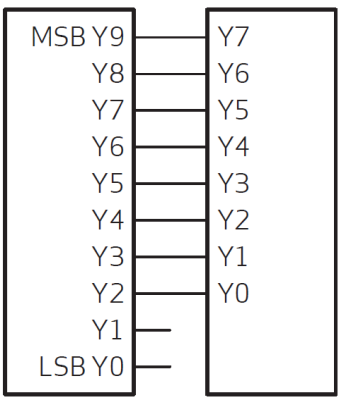


图 43.1.1.2 OV5640 默认 8 位连接方式

OV5640 的寄存器通过 SCCB 时序访问并设置，SCCB 时序和 IIC 时序十分类似，在本章我们不做介绍，请大家参考光盘《OmniVision Technologies Seril Camera Control Bus(SCCB) Specification》这个文档。

2，窗口设置说明

接下来，我们介绍一下 OV5640 的：ISP（Image Signal Processor）输入窗口设置、预缩放窗口设置和输出大小窗口设置，这几个设置与我们的正常使用密切相关，有必要了解一下。他们的设置关系，如图 43.1.1.3 所示：

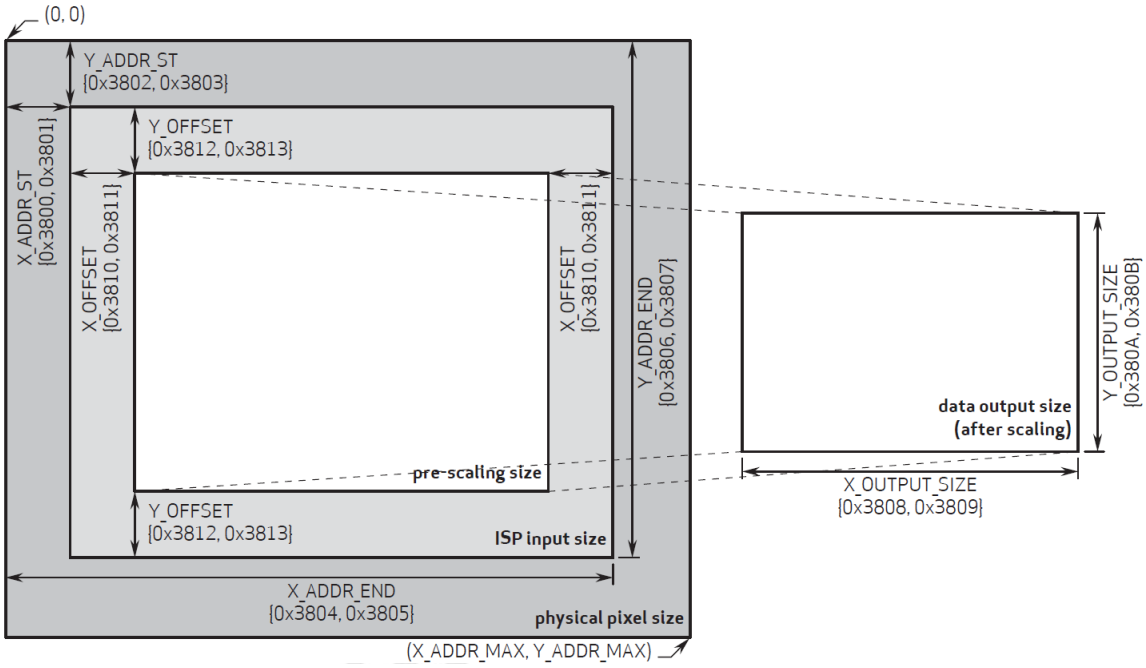


图 43.1.4 OV5640 各窗口设置关系

ISP 输入窗口设置（ISP input size）

该设置允许用户设置整个传感器区域（physical pixel size ，2632*1951）的感兴趣部分，也就是在传感器里面开窗（X_ADDR_ST、Y_ADDR_ST、X_ADDR_END 和 Y_ADDR_END），开窗范围从 0*0~2632*1951 都可以设置，该窗口所设置的范围，将输入 ISP 进行处理。

ISP 输入窗口，通过：0X3800~0X3807 等 8 个寄存器进行设置，这些寄存器的定义请看：OV5640_CSP3_DS_2.01_Ruisipusheng.pdf 这个文档（下同）。

预缩放窗口设置（pre-scaling size）

该设置允许用户在 ISP 输入窗口的基础上，再次设置将要用于缩放的窗口大小。该设置仅在 ISP 输入窗口内进行 x/y 方向的偏移（X_OFFSET/Y_OFFSET）。通过：0X3810~0X3813 等 4 个寄存器进行设置。

输出大小窗口设置（data output size）

该窗口是以预缩放窗口为原始大小，经过内部 DSP 进行缩放处理后，输出给外部的图像窗口大小。它控制最终的图像输出尺寸（X_OUTPUT_SIZE/Y_OUTPUT_SIZE）。通过：0X3808~0X380B 等 4 个寄存器进行设置。注意：当输出大小窗口与预缩放窗口比例不一致时，图像将进行缩放处理（会变形），仅当两者比例一致时，输出比例才是 1:1（正常）。

图 43.1.4 中，右侧 data output size 区域，才是 OV5640 输出给外部的图像尺寸，也就是显示在 LCD 上面的图像大小。输出大小窗口与预缩放窗口比例不一致时，会进行缩放处理，在 LCD 上面看到的图像将会变形。

3，输出时序说明

接下来，我们介绍一下 OV5640 的图像数据输出时序。首先我们简单介绍一些定义：

QSXGA，这里指：分辨率为 2592*1944 的输出格式，类似的还有：QXGA(2048*1536)、UXGA(1600*1200)、SXGA(1280*1024)、WXGA+(1440*900)、WXGA(1280*800)、XGA(1024*768)、SVGA(800*600)、VGA(640*480)、QVGA(320*240)和 QQVGA(160*120)等。

PCLK，即像素时钟，一个 PCLK 时钟，输出一个像素(或半个像素)。

VSYNC，即帧同步信号。

HREF /HSYNC，即行同步信号。

OV5640 的图像数据输出（通过 Y[9:0]）就是在 PCLK，VSYNC 和 HREF/ HSYNC 的控制下进行的。首先看看行输出时序，如图 43.1.1.5 所示：

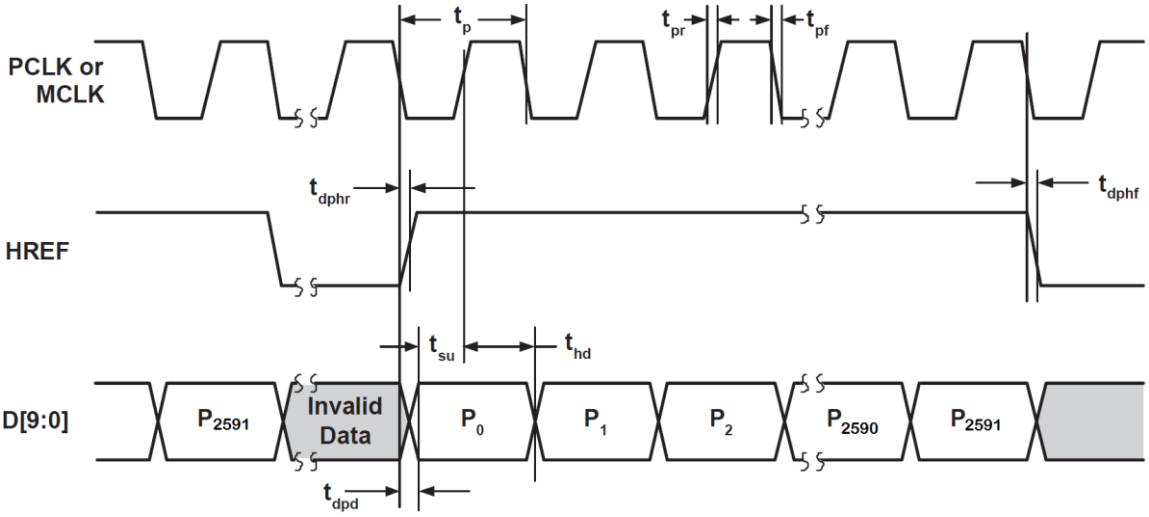


图 43.1.1.5 OV5640 行输出时序

从上图可以看出，图像数据在 HREF 为高的时候输出，当 HREF 变高后，每一个 PCLK 时钟，输出一个 8 位/10 位数据。我们采用 8 位接口，所以每个 PCLK 输出 1 个字节，且在 RGB/YUV 输出格式下，每个 tp=2 个 Tpclk，如果是 Raw 格式，则一个 tp=1 个 Tpclk。比如我们采用 QSXGA 时序，RGB565 格式输出，每 2 个字节组成一个像素的颜色（低字节在前，高字节在后），这样每行输出总共有 2592*2 个 PCLK 周期，输出 2592*2 个字节。

再来看看帧时序（QSXGA 模式），如图 43.1.1.6 所示：

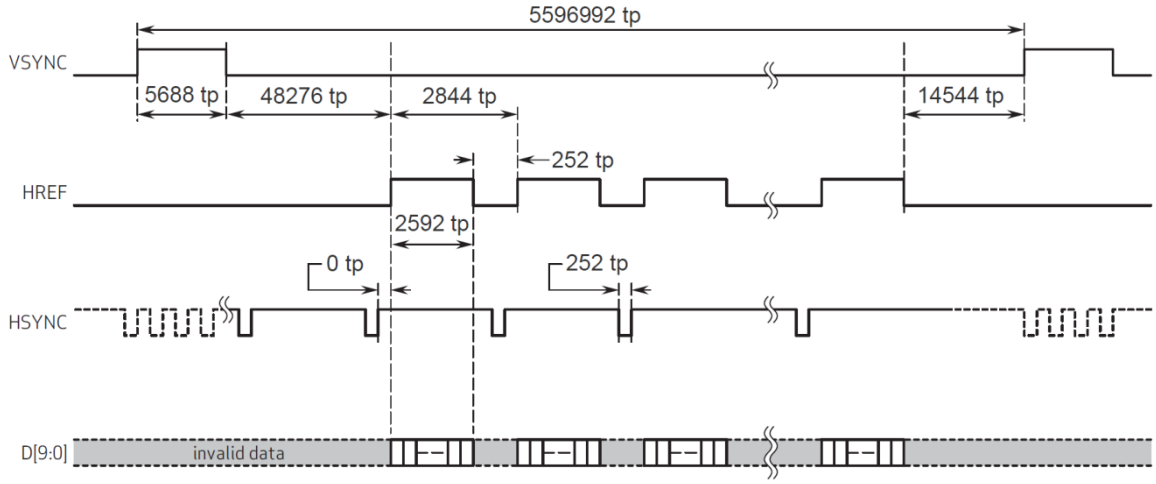


图 43.1.1.6 OV5640 帧时序

上图清楚的表示了 OV5640 在 QSXGA 模式下的数据输出。我们按照这个时序去读取 OV5640 的数据，就可以得到图像数据。

4，自动对焦（Auto Focus）说明

OV5640 由内置微型控制器完成自动对焦，并且 VCM（Voice Coil Motor，即音圈马达）驱动器也已集成在传感器内部。微型控制器的控制固件（firmware）从主机下载。当固件运行后，内置微型控制器从 OV5640 传感器读得自动对焦所需的信息，计算并驱动 VCM 马达带动镜头到达正确的对焦位置。主机可以通过 IIC 命令控制微型控制器的各种功能。

OV5640 的自动对焦命令（通过 SCCB 总线发送），如表 43.1.1.1 所示：

地址	寄存器名	描述	值
0X3022	CMD_MAIN	AF 主命令寄存器	0X03：触发单次自动对焦过程 0X04：启动持续自动对焦过程 0X06：暂停自动对焦过程 0X08：释放马达回到初始状态 0X12：设置对焦区域 0X00：命令完成
0X3023	CMD_ACK	命令确认	0X00：命令完成 0X01：命令执行中
0X3029	FW_STATUS	对焦状态	0X7F：固件下载完成，但未执行，可能是：固件有问题/微控制器关闭 0X7E：固件初始化中 0X70：释放马达，回到初始状态 0X00：正在自动对焦 0X10：自动对焦完成

表 43.1.1.1 OV5640 自动对焦命令

OV5640 内部的微控制器收到自动对焦命令后会自动将 CMD_MAIN（0X3022）寄存器数据清零，当命令完成后会将 CMD_ACK（0X3023）寄存器数据清零。

自动对焦（AF）过程

① 在第一次进入图像预览的时候（图像可以正常输出时），下载固件（firmware）

② 拍照前，自动对焦，对焦完成后，拍照

③ 拍照完毕，释放马达到初始状态

接下来，我们分别说明。

① 下载固件

OV5640 初始化完成后，就可以下载 AF 自动对焦固件了，其操作和下载初始化参数类似，AF 固件下载地址为：0X8000，初始化数组由厂家提供（本例程该数组保存在 ov5640af.h 里面），下载固件完成后，通过检查 0X3029 寄存器的值，来判断固件状态（等于 0X70，说明正常）。

② 自动对焦

OV5640 支持单次自动对焦和持续自动对焦，通过 0X3022 寄存器控制。单次自动对焦过程如下：

1，将 0X3022 寄存器写为 0X03，开始单点对焦过程。

2，读取寄存器 0X3029，如果返回值为 0X10，代表对焦已完成。

3，写寄存器 0X3022 为 0X06，暂停对焦过程，使镜头将保持在此对焦位置。

其中，前两步是必须的，第三步，可以不要，因为单次自动对焦完成以后，就不会继续自动对焦了，镜头也就不会动了。

持续自动对焦过程如下：

1， 将 0X22 寄存器写为 0X08，释放马达到初始位置（对焦无穷远）。

2， 将 0X3022 寄存器写为 0X04，启动持续自动对焦过程。

3， 读取寄存器 0X3023，等待命令完成。

4， 当 OV5640 每次检测到失焦时，就会自动进行对焦（一直检测）。

③ 释放马达，结束自动对焦

最后，在拍照完成，或者需要结束自动对焦的时候，我们对在寄存器 0X3022 写入 0X08，即可释放马达，结束自动对焦。

最后说一下 OV5640 的图像数据格式，我们一般用 2 种输出方式：RGB565 和 JPEG。当输出 RGB565 格式数据的时候，时序完全就是上面两幅图介绍的关系。以满足不同需要。而当输出数据是 JPEG 数据的时候，同样也是这种方式输出（所以数据读取方法一模一样），不过 PCLK 数目大大减少了，且不连续，输出的数据是压缩后的 JPEG 数据，输出的 JPEG 数据以：0XFF,0XD8 开头，以 0XFF,0XD9 结尾，且在 0XFF,0XD8 之前，或者 0XFF,0XD9 之后，会有不定数量的其他数据存在（一般是 0），这些数据我们直接忽略即可，将得到的 0XFF,0XD8~0XFF,0XD9 之间的数据，保存为.jpg/.jpeg 文件，就可以直接在电脑上打开看到图像了。

OV5640 自带的 JPEG 输出功能，大大减少了图像的数据量，使得其在网络摄像头、无线视频传输等方面具有很大的优势。OV5640 我们就介绍到这，关于 OV5640 更详细的介绍，请大家参考：**A 盘→7，硬件资料→OV5640 资料→ OV5640_CSP3_DS_2.01_Ruisipusheng.pdf**。

43.1.2 STM32F767 DCMI 接口简介

STM32F767 自带了一个数字摄像头（DCMI）接口，该接口是一个同步并行接口，能够接收外部 8 位、10 位、12 位或 14 位 CMOS 摄像头模块发出的高速数据流。可支持不同的数据格式：YCbCr4:2:2/RGB565 逐行视频和压缩数据 (JPEG)。

STM32F767 DCM 接口特点：

- 8 位、10 位、12 位或 14 位并行接口
- 内嵌码/外部行同步和帧同步
- 连续模式或快照模式
- 裁剪功能
- 支持以下数据格式：
  - 1，8/10/12/14 位逐行视频：单色或原始拜尔（Bayer）格式
  - 2，YCbCr 4:2:2 逐行视频
  - 3，RGB 565 逐行视频
  - 4，压缩数据：JPEG

DCMI 接口包括如下一些信号：

1， 数据输入（D[0:13]），用于接摄像头的数据输出，接 OV5640 我们只用了 8 位数据。

2， 水平同步（行同步）输入（HSYNC），用于接摄像头的 HSYNC/HREF 信号。

3， 垂直同步（场同步）输入（VSYNC），用于接摄像头的 VSYNC 信号。

4， 像素时钟输入（PIXCLK），用于接摄像头的 PCLK 信号。

DCMI 接口是一个同步并行接口，可接收高速（可达 54 MB/s）数据流。该接口包含多达 14 条数据线(D13-D0)和一条像素时钟线(PIXCLK)。像素时钟的极性可以编程，因此可以在像素时钟的上升沿或下降沿捕获数据。

DCMI 接收到的摄像头数据被放到一个 32 位数据寄存器(DCMI_DR)中，然后通过通用

DMA 进行传输。图像缓冲区由 DMA 管理，而不是由摄像头接口管理。

从摄像头接收的数据可以按行/帧来组织（原始 YUV/RGB/拜尔模式），也可以是一系列 JPEG 图像。要使能 JPEG 图像接收，必须将 JPEG 位（DCMI_CR 寄存器的位 3）置 1。

数据流可由可选的 HSYNC（水平同步）信号和 VSYNC（垂直同步）信号硬件同步，或者通 过数据流中嵌入的同步码同步。

STM32F767 DCMI 接口的框图如图 43.1.2.1 所示：

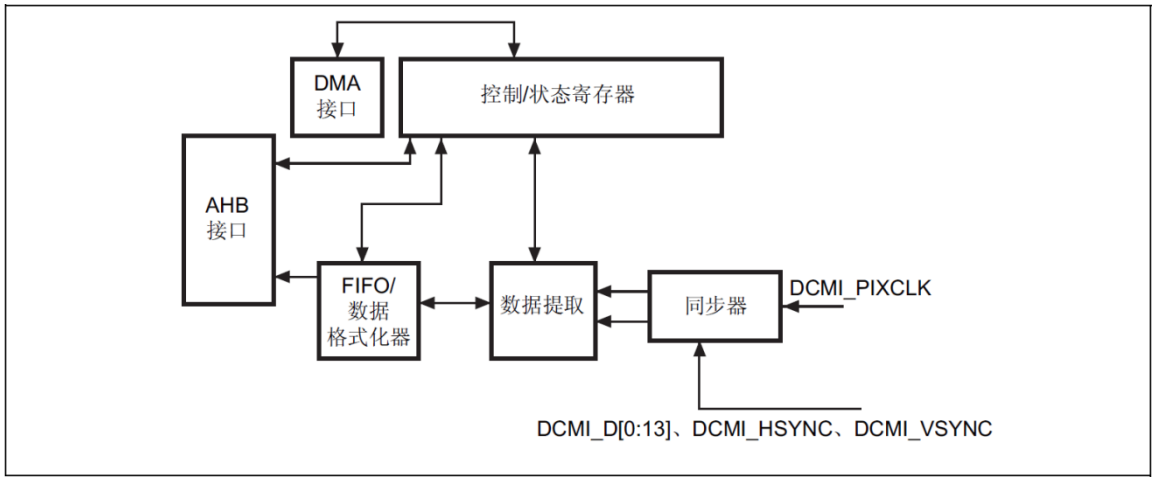


图 43.1.2.1 DCMI 接口框图

DCMI 接口的数据与 PIXCLK（即 PCLK）保持同步，并根据像素时钟的极性在像素时钟上升沿/下降沿发生变化。HSYNC（HREF）信号指示行的开始/结束，VSYNC 信号指示帧的开始/结束。DCMI 信号波形如图 43.1.2.2 所示：

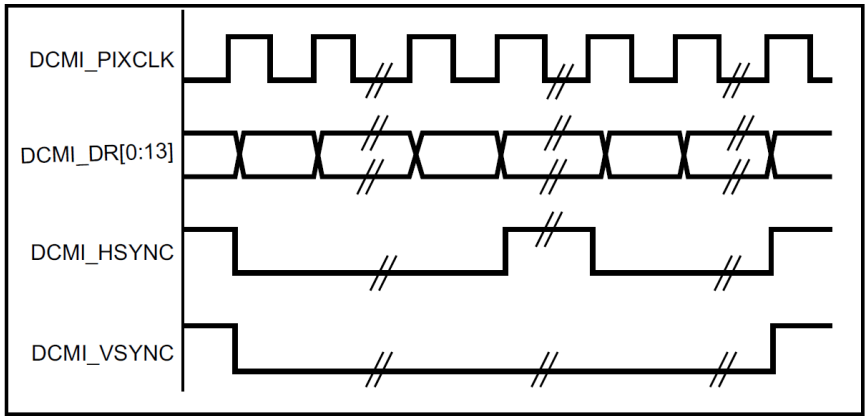


图 43.1.2.2 DCMI 信号波形

上图中，对应设置为：DCMI_PIXCLK 的捕获沿为下降沿，DCMI_HSYNC 和 DCMI_VSYNC 的有效状态为 1，注意，这里的有效状态实际上对应的是指示数据在并行接口上无效时，HSYNC/VSYNC 引脚上面的引脚电平。

本章我们用到 DCMI 的 8 位数据宽度，通过设置 DCMI_CR 中的 EDM[1:0]=00 设置。此时 DCMI_D0~D7 有效，DCMI_D8~D13 上的数据则忽略，这个时候，每次需要 4 个像素时钟来捕获一个 32 位数据。捕获的第一个数据存放在 32 位字的 LSB 位置，第四个数据存放在 32 位字的 MSB 位置 ，捕获数据字节在 32 位字中的排布如表 43.1.2.1 所示：

字节地址	31:24	23:16	15:8	7:0
0	D_{n+3}[7:0]	D_{n+2}[7:0]	D_{n+1}[7:0]	D_n[7:0]
4	D_{n+7}[7:0]	D_{n+6}[7:0]	D_{n+5}[7:0]	D_{n+4}[7:0]

表 43.1.2.1 8 位捕获数据在 32 位字中的排布

从表 43.1.2.1 可以看出，STM32F767 的 DCMI 接口，接收的数据是低字节在前，高字节在后的，所以，要求摄像头输出数据也是低字节在前，高字节在后才可以，否则就还得程序上处理字节顺序，会比较麻烦。

DCMI 接口支持 DMA 传输，当 DCMI_CR 寄存器中的 CAPTURE 位置 1 时，激活 DMA 接口。摄像头接口每次在其寄存器中收到一个完整的 32 位数据块时，都将触发一个 DMA 请求。

DCMI 接口支持两种同步方式：内嵌码同步和硬件（HSYNC 和 VSYNC）同步。我们简单介绍下硬件同步，详细介绍请参考《STM32F7 中文参考手册》第 17.5.3 节。

硬件同步模式下将使用两个同步信号 (HSYNC/VSYNC)。根据摄像头模块/模式的不同，可能在水平/垂直同步期间内发送数据。由于系统会忽略 HSYNC/VSYNC 信号有效电平期间内接收的所有数据，HSYNC/VSYNC 信号相当于消隐信号。

为了正确地将图像传输到 DMA/RAM 缓冲区，数据传输将与 VSYNC 信号同步。选择硬件同步模式并启用捕获（DCMI_CR 中的 CAPTURE 位置 1）时，数据传输将与 VSYNC 信号的无效电平同步（开始下一帧时）。之后传输便可以连续执行，由 DMA 将连续帧传输到多个连续的缓冲区或一个具有循环特性的缓冲区。为了允许 DMA 管理连续帧，每一帧结束时都将激活 VSIF（垂直同步中断标志，即帧中断），我们可以利用这个帧中断来判断是否有一帧数据采集完成，方便处理数据。

DCMI 接口的捕获模式支持：快照模式和连续采集模式。一般我们使用连续采集模式，通过 DCMI_CR 中的 CM 位设置。另外，DCMI 接口还支持实现了 4 个字深度的 FIFO，配有一个简单的 FIFO 控制器，每次摄像头接口从 AHB 读取数据时读指针递增，每次摄像头接口向 FIFO 写入数据时写指针递增。因为没有溢出保护，如果数据传输率超过 AHB 接口能够承受的速率，FIFO 中的数据就会被覆盖。如果同步信号出错，或者 FIFO 发生溢出，FIFO 将复位，DCMI 接口将等待新的数据帧开始。

关于 DCMI 接口的其他特性，我们这里就不再介绍了，请大家参考《STM32F7 中文参考手册》第 17 章相关内容。

本章，我们将使用 STM32F767IGT6 的 DCMI 接口连接 ALIENTEK OV5640 摄像头模块，该模块采用 8 位数据输出接口，自带 24M 有源晶振，无需外部提供时钟，模组支持自动对焦功能，且支持闪光灯，整个模块只需提供 3.3V 供电即可正常使用。

ALIENTEK OV5640 摄像头模块外观如图 43.1.2.3 所示：

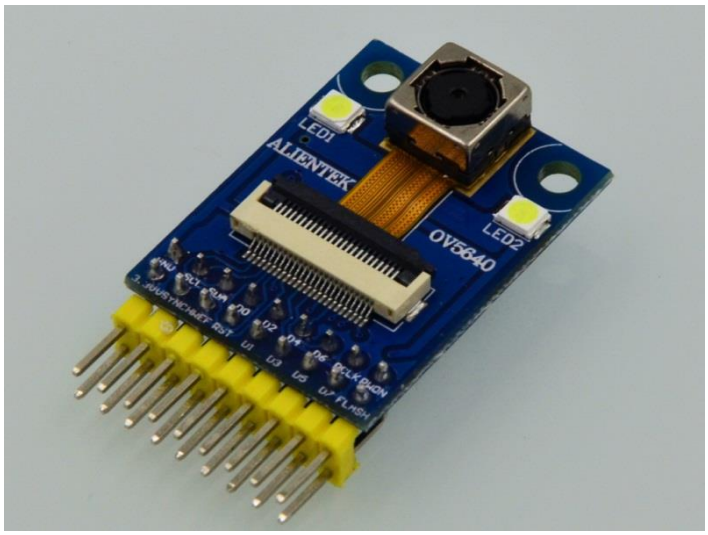


图 43.1.2.3 ALIENTEK OV5640 摄像头模块外观图

模块原理图如图 43.1.2.4 所示：

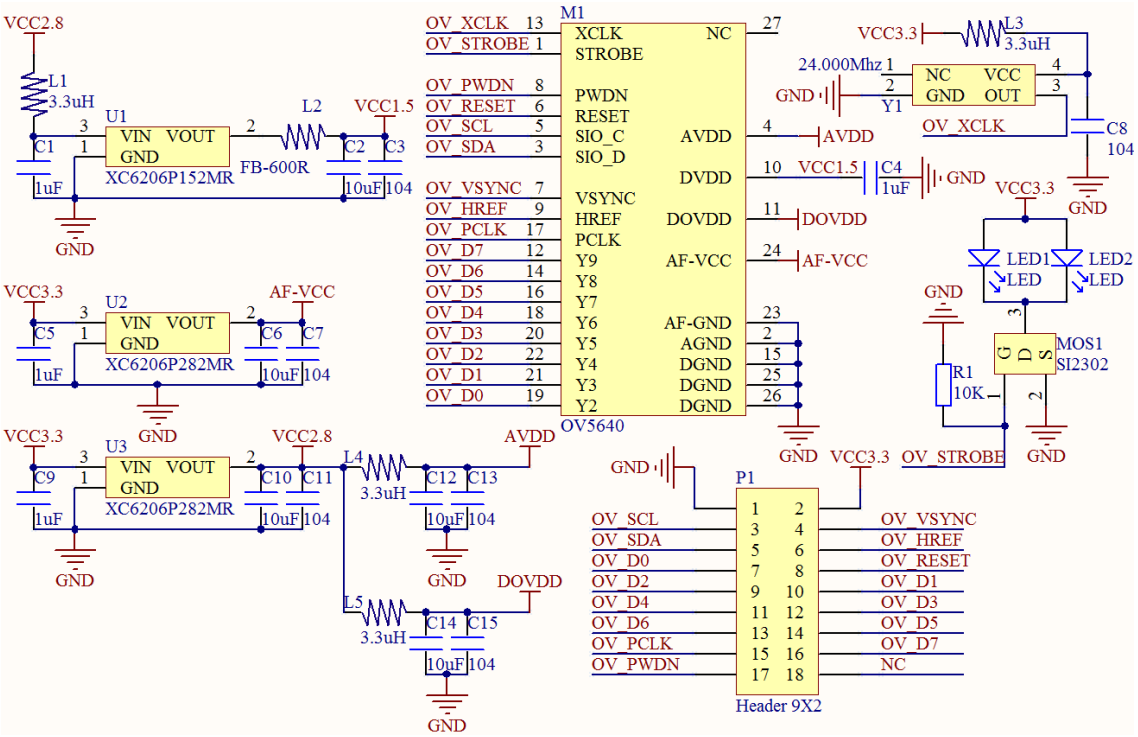


图 43.1.2.4 ALIENTEK OV5640 摄像头模块原理图

从上图可以看出，ALIENTEK OV5640 摄像头模块自带了有源晶振，用于产生 24M 时钟作为 OV5640 的 XCLK 输入，模块的闪光灯（LED1&LED2）由 OV5640 的 STROBE 脚控制（可编程控制）。同时自带了稳压芯片，用于提供 OV5640 稳定的 2.8V 和 1.5V 工作电压，模块通过一个 2*9 的双排排针（P1）与外部通信，与外部的通信信号如表 43.1.2.2 所示：

信号	作用描述	信号	作用描述
VCC3.3	模块供电脚，接 3.3V 电源	OV_PCLK	像素时钟输出
GND	模块地线	OV_PWDN	掉电使能(高有效)
OV_SCL	SCCB 通信时钟信号	OV_VSYNC	帧同步信号输出
OV_SDA	SCCB 通信数据信号	OV_HREF	行同步信号输出

OV_D[7:0]	8 位数据输出	OV_RESET	复位信号(低有效)

表 43.1.2.2 OV5640 模块信号及其作用描述

本章，我们将 OV5640 默认配置为 WXGA 输出，也就是 1280*800 的分辨率，输出信号设置为：VSYNC 高电平有效，HREF 高电平有效，输出数据在 PCLK 的下降沿输出（即上升沿的时候，MCU 才可以采集）。这样，STM32F767 的 DCMI 接口就必须设置为：VSYNC 低电平有效、HSYNC 低电平有效和 PIXCLK 上升沿有效，这些设置都是通过 DCMI_CR 寄存器控制的，该寄存器描述如图 43.1.2.5 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
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OELS	LSM	OEBS	BSM		Res.	ENABLE	Res.	Res.	EDM		FCRC		VSPOL	HSPOL	PCKPOL	ESS	JPEG	CROP	CM	CAPTURE
											rw	rw	rw	rw	rw		rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 43.1.2.5 DCMI_CR 寄存器各位描述

ENABLE，该位用于设置是否使能 DCMI，不过，在使能之前，必须将其他配置设置好。

FCRC[1:0]，这两个位用于帧率控制，我们捕获所有帧，所以设置为 00 即可。

VSPOL，该位用于设置垂直同步极性，也就是 VSYNC 引脚上面，数据无效时的电平状态，根据前面说所，我们应该设置为 0。

HSPOL，该位用于设置水平同步极性，也就是 HSYNC 引脚上面，数据无效时的电平状态，同样应该设置为 0。

PCKPOL，该位用于设置像素时钟极性，我们用上升沿捕获，所以设置为 1。

CM，该位用于设置捕获模式，我们用连续采集模式，所以设置为 0 即可。

CAPTURE，该位用于使能捕获，我们设置为 1。该位使能后，将激活 DMA，DCMI 等待第一帧开始，然后生成 DMA 请求将收到的数据传输到目标存储器中。注意：该位必须在 DCMI 的其他配置（包括 DMA）都设置好了之后，才设置!!

DCMI_CR 寄存器的其他位，我们就不介绍了，另外 DCMI 的其他寄存器这里也不再介绍，请大家参考《STM32F7 中文参考手册》第 17.8 节。

最后，我们来看下用 DCMI 驱动 OV5640 的步骤。HAL 库中 DCMI 接口相关的库函数分布在源文件 stm32f7xx_hal_dcmi.c/stm32f7xx_hal_dcmi_ex.c 以及头文件 stm32f7xx_hal_dcmi.h 中。

1）配置 OV5640 控制引脚，并配置 OV5640 工作模式。

在启动 DCMI 之前，我们先设置好 OV5640。OV5640 通过 OV_SCL 和 OV_SDA 进行寄存器配置，同时还有 OV_PWDN/OV_RESET 等信号，我们也需要配置对应 IO 状态，先设置 OV_PWDN 为 0，退出掉电模式，然后拉低 OV_RESET 复位 OV5640，之后再设置 OV_RESET 为 1，结束复位，然后就是对 OV5640 的大把寄存器进行配置了。然后，可以根据我们的需要，设置成 RGB565 输出模式，还是 JPEG 输出模式。

2）配置相关引脚的模式和复用功能（AF13），使能时钟。

OV5640 配置好之后，再设置 DCMI 接口与摄像头模块连接的 IO 口，使能 IO 和 DCMI 时钟，然后设置相关 IO 口为复用功能模式，复用功能选择 AF13(DCMI 复用)。

DCMI 时钟使能方法：

```
__HAL_RCC_DCMI_CLK_ENABLE();  //使能 DCMI 时钟
```

引脚模式配置就是通过 HAL_GPIO_Init 函数来配置，这里就不多说了。

3）配置 DCMI 相关设置，初始化 DCMI 接口。

这一步，主要通过 DCMI_CR 寄存器设置，包括 VSPOL/HSPOL/PCKPOL/数据宽度等重要参数，都在这一步设置。HAL 库提供了 DCMI 初始化函数 HAL_DCMI_Init，函数声明如下：

```
HAL_StatusTypeDef HAL_DCMI_Init(DCMI_HandleTypeDef *hdcmi);
```

结构体 DCMI_HandleTypeDef 定义为：

```
typedef struct
{
  DCMI_TypeDef                 *Instance;
  DCMI_InitTypeDef             Init;
  HAL_LockTypeDef              Lock;
  __IO HAL_DCMI_StateTypeDef   State;
  __IO uint32_t                XferCount;
  __IO uint32_t                XferSize;
  uint32_t                     XferTransferNumber;
  uint32_t                     pBuffPtr;
  DMA_HandleTypeDef            *DMA_Handle;
  __IO uint32_t                ErrorCode;
}DCMI_HandleTypeDef;
```

该结构体第一个成员变量 Instance 用来指向寄存器基地址，设置为 DCMI 即可。

成员变量 XferCount，XferSize，XferTransferNumber，pBuffPtr 和 DMA_Handle 是与 HAL 库中 DMA 处理相关中间变量，由于使用 HAL 库配置的 DCMI DMA 会非常复杂，而且灵活性不高，所以本实验我们是自由独立配置的 DMA。

成员变量 Init 是 DCMI_InitTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
  uint32_t  SynchroMode;   //同步方式为硬件同步还是内嵌码同步
  uint32_t  PCKPolarity;   //像素极性
  uint32_t  VSPolarity;    //垂直同步极性
  uint32_t  HSPolarity;     //水平同步极性
  uint32_t  CaptureRate;   //帧捕获率
  uint32_t  ExtendedDataMode; //扩展数据模式
  DCMI_CodesInitTypeDef SyncroCode;//分隔符设置
  uint32_t JPEGMode;  //JPEG 模式选择
  uint32_t ByteSelectMode; //设置字节选项模式
  uint32_t ByteSelectStart; //字节选择开始：奇数/偶数字节选择
  uint32_t LineSelectMode;//行选择模式
  uint32_t LineSelectStart;
}DCMI_InitTypeDef;
```

成员变量 SynchroMode 用来选择同步方式为硬件同步还是内嵌码同步。如果选择硬件同步值 DCMI_SYNCHRO_HARDWARE，那么数据捕获由 HSYNC/VSYNC 信号同步，如果选择内嵌码同步方式值 DCMI_SYNCHRO_EMBEDDED，那么数据捕获由数据流中嵌入的同步码同步。

成员变量 PCKPolarity 用来设置像素时钟极性为上升沿有效还是下降沿有效。我们实验使用的是上升沿有效，所以值为 DCMI_PCKPOLARITY_RISING。

成员变量 VSPolarity 用来设置垂直同步极性 VSYNC 为低电平有效还是高电平有效。也就是 VSYNC 引脚上面，数据无效时的电平状态。我们设置为 VSYNC 低电平有效。所以值为 DCMI_VSPOLARITY_LOW。

成员变量 HSPolarity 用来设置水平同步极性为高电平有效还是低电平有效，也就是 HSYNC

引脚上面，数据无效时的电平状态。我们设置为 HSYNC 低电平有效。所以值为 DCMI_HSPOLARITY_LOW。

成员变量 CaptureRate 用来设置帧捕获率。如果设置为值 DCMI_CR_ALL_FRAME，也就是全帧捕获，设置为 DCMI_CR_ALTERNATE_2_FRAME，也就 2 帧捕获一帧，设置为 DCMI_CR_ALTERNATE_4_FRAME，也就是 4 帧捕获一帧。

成员变量 ExtendedDataMode 用来设置扩展数据模式。可以设置为每个像素时钟捕获 8 位，10 位，12 位以及 14 位数据。这里我们设置为 8 位值 DCMI_EXTEND_DATA_8B。

成员变量 SyncroCode 用来设置分隔码，包括：帧结束分隔码，行结束分隔码，行开始分隔码以及帧开始分隔码。

成员变量 DCMI_CaptureMode 是用来设置捕获模式为连续捕获模式还是快照模式。我们实验采取的是连续捕获模式值 DCMI_CaptureMode_Continuous，也就是通过 DMA 连续传输数据到目标存储区。

成员变量 JPEGMode 用来设置 JPEG 格式使能。

成员变量 ByteSelectMode 用来设置字节选项模式，也就是接口对接收到的数据每隔多少个字节捕获一个字节，取值为：DCMI_BSM_ALL(捕获所有字节)，DCMI_BSM_OTHER(每隔一个字节进行捕获)，DCMI_BSM_ALTERNATE_4（每四个字节捕获一个字节）和 DCMI_BSM_ALTERNATE_2（每四个字节捕获两个字节）。

成员变量 ByteSelectStart 是奇数偶数字节选择开始。也就是接口从帧/行开始捕获第一个数据同时丢弃第二个字节（DCMI_OEBS_ODD）或者捕获第二个数据同时丢弃第一个字节（DCMI_OEBS_EVEN）。

成员变量 LineSelectMode 用来配置行选择模式，也就是选择接口捕获所有接受到的行（DCMI_LSM_ALL）还是每两行捕获一行（DCMI_LSM_ALTERNATE_2）。

成员变量 LineSelectStart 用来配置奇数偶数行选择开始。也就是接口在帧开始后捕获第一行丢弃第二行（DCMI_OELS_ODD）或者在帧开始后捕获第二行同时丢弃第一行（DCMI_OELS_EVEN）。

各个成员变量含义就给大家讲解到这里，函数 HAL_DCMI_Init 初始化实例为：

```
DCMI_HandleTypeDef    DCMI_Handler;                  //DCMI 句柄
DCMI_Handler.Instance=DCMI;
DCMI_Handler.Init.SynchroMode=DCMI_SYNCHRO_HARDWARE;//硬件同步
DCMI_Handler.Init.PCKPolarity=DCMI_PCKPOLARITY_RISING; //PCLK 上升沿有效
DCMI_Handler.Init.VSPolarity=DCMI_VSPOLARITY_LOW; //VSYNC 低电平有效
DCMI_Handler.Init.HSPolarity=DCMI_HSPOLARITY_LOW; //HSYNC 低电平有效
 DCMI_Handler.Init.CaptureRate=DCMI_CR_ALL_FRAME; //全帧捕获
 DCMI_Handler.Init.ExtendedDataMode=DCMI_EXTEND_DATA_8B; //8 位数据格式
 HAL_DCMI_Init(&DCMI_Handler); //初始化 DCMI 接口
```

同样，HAL 库也提供了 DCMI 接口的 MSP 初始化回调函数：

```
void HAL_DCMI_MspInit(DCMI_HandleTypeDef* hdcmi);
```

一般情况下，该函数内部编写时钟使能，IO 初始化以及 NVIC 相关程序。

4）配置 DMA。

本章采用连续模式采集，并将采集到的数据输出到 LCD（RGB565 模式）或内存（JPEG 模式），所以源地址都是 DCMI_DR，而目的地址可能是 LCD->RAM 或者 SRAM 的地址。DCMI 的 DMA 传输采用的是 DMA2 数据流 1 的通道 1 来实现的，关于 DMA 的介绍，请大家参考前面的 DMA 实验章节。这里我们列出本章我们的 DMA 配置源码如下：

```
__HAL_RCC_DMA2_CLK_ENABLE();     //使能 DMA2 时钟
HAL_LINKDMA(&DCMI_Handler,DMA_Handle,DMADMCI_Handler);
 //将 DMA 与 DCMI 联系起来
DMADMCI_Handler.Instance=DMA2_Stream1;   //DMA2 数据流 1
DMADMCI_Handler.Init.Channel=DMA_CHANNEL_1; //通道 1
DMADMCI_Handler.Init.Direction=DMA_PERIPH_TO_MEMORY; //外设到存储器
DMADMCI_Handler.Init.PeriphInc=DMA_PINC_DISABLE; /外设非增量模式
DMADMCI_Handler.Init.MemInc=meminc; //存储器增量模式
DMADMCI_Handler.Init.PeriphDataAlignment=
                                DMA_PDATAALIGN_WORD; //外设数据长度:32 位
DMADMCI_Handler.Init.MemDataAlignment=
                                DMA_MDATAALIGN_WORD; //存储器数据长度: 32 位
DMADMCI_Handler.Init.Mode=DMA_CIRCULAR; //使用循环模式
DMADMCI_Handler.Init.Priority=DMA_PRIORITY_HIGH; //高优先级
DMADMCI_Handler.Init.FIFOMode=DMA_FIFOMODE_ENABLE; //使能 FIFO
DMADMCI_Handler.Init.FIFOThreshold=DMA_FIFO_THRESHOLD_HALFFULL;
                                                    //使用 1/2 的 FIFO
DMADMCI_Handler.Init.MemBurst=DMA_MBURST_SINGLE; //存储器突发传输
DMADMCI_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE;   //外设突发单次传输
HAL_DMA_DeInit(&DMADMCI_Handler);              //先清除以前的设置
HAL_DMA_Init(&DMADMCI_Handler);  //初始化 DMA
```

5）设置 OV5640 的图像输出大小，使能 DCMI 捕获。

图像输出大小设置，分两种情况：在 RGB565 模式下，我们根据 LCD 的尺寸，设置输出图像大小，以实现全屏显示（图像可能因缩放而变形）；在 JPEG 模式下，我们可以自由设置输出图像大小（可不缩放）；最后，开启 DCMI 捕获，即可正常工作了。

43.2 硬件设计

本章实验功能简介：开机后，初始化摄像头模块（OV5640），如果初始化成功，则提示选择模式：RGB565 模式，或者 JPEG 模式。KEY0 用于选择 RGB565 模式，KEY1 用于选择 JPEG 模式。

当使用 RGB565 时，输出图像（固定为：WXGA）将经过缩放处理（完全由 OV5640 的 DSP 控制），显示在 LCD 上面（默认开启连续自动对焦）。我们可以通过 KEY_UP 按键选择：1:1 显示，即不缩放，图片不变形，但是显示区域小（液晶分辨率大小），或者缩放显示，即将 1280*800 的图像压缩到液晶分辨率尺寸显示，图片变形，但是显示了整个图片内容。通过 KEY0 按键，可以设置对比度；KEY1 按键，可以启动单次自动对焦；KEY2 按键，可以设置特效。

当使用 JPEG 模式时，图像可以设置任意尺寸（QSXGA~QQVGA），采集到的 JPEG 数据将先存放到 STM32F767 的 SDRAM 内存里面，每当采集到一帧数据，就会关闭 DMA 传输，然后将采集到的数据发送到串口 2（此时可以通过上位机软件（ATK-CAM.exe）接收，并显示图片），之后再重新启动 DMA 传输。我们可以通过 KEY_UP 设置输出图片的尺寸（QSXGA~QQVGA）。通过 KEY0 按键，可以设置对比度；KEY1 按键，可以启动单次自动对焦；KEY2 按键，可以设置特效。

同时时可以通过串口 1，借助 USMART 设置/读取 OV5640 的寄存器，方便大家调试。DS0 指示程序运行状态，DS1 用于指示帧中断。

本实验用到的硬件资源有：

1） 指示灯 DS0 和 DS1

2） 4 个按键

3） 串口 1 和串口 2

4） LCD 模块

5） PCF8574T

6） OV5640 摄像头模块

这些资源，基本上都介绍过了，这里我们用到串口 2 来传输 JPEG 数据给上位机，其配置同串口 1 几乎一模一样，只是串口 2 的时钟来自 APB1，频率为 54Mhz。开发板板载的摄像头模块接口与 MCU 的连接如图 43.2.1 所示：

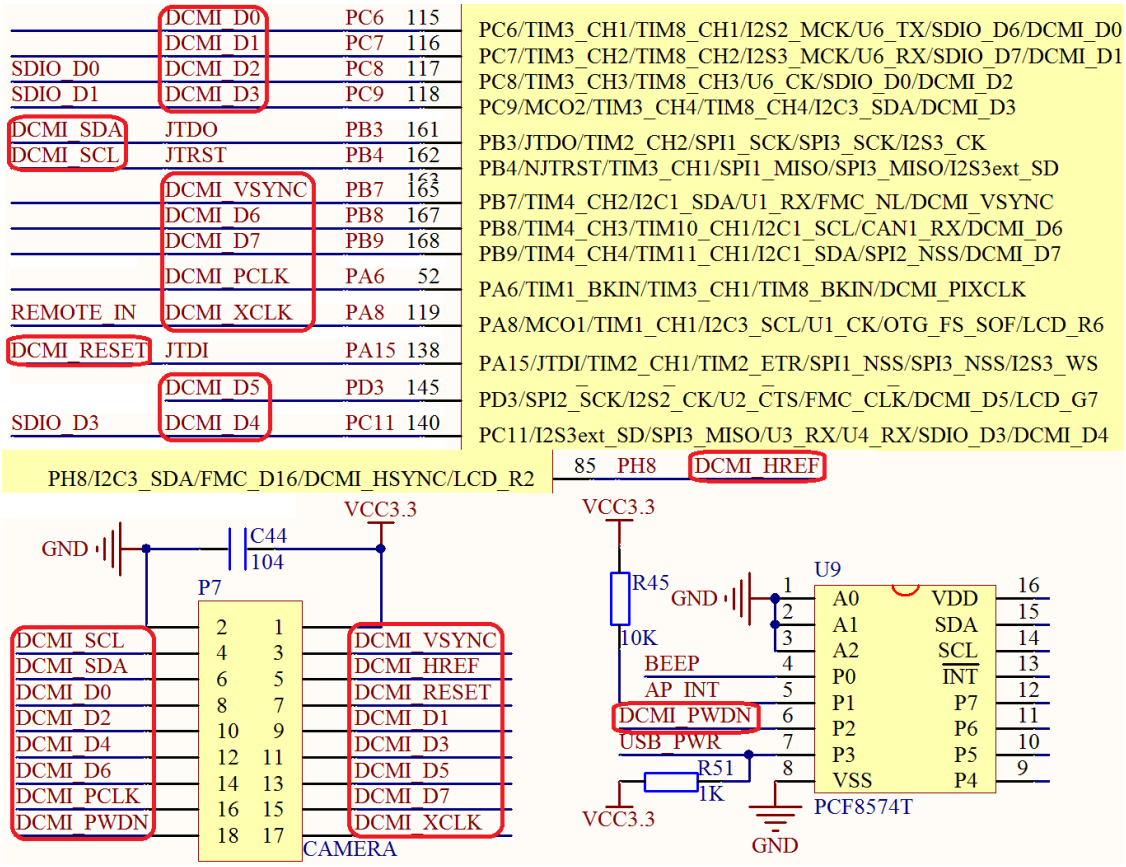


图 43.2.1 摄像头模块接口与 STM32 连接图

图中 P7 就是摄像头模块/OLED 模块共用接口，在第十六章，我们曾简单介绍过这个接口，它在开发板的左下角，是一个 2*9 的排座（P7）。本章，我们只需要将 ALIENTEK OV5640 摄像头模块插入这个接口即可。

从图 43.2.1 可以看出，OV5640 摄像头模块的各信号脚与 STM32 的连接关系为：

DCMI_VSYNC 接 PB7;

DCMI_HREF 接 PH8;

DCMI_PCLK 接 PA6;

DCMI_SCL 接 PB4;

DCMI_SDA 接 PB3;

DCMI_RESET 接 PA15;

DCMI_PWDN 接 PCF8574T 的 P2 脚;

DCMI_XCLK 接 PA8（本章未用到）；

DCMI_D[7:0]接 PB9/PB8/PD3/PC11/PC9/PC8/PC7/PC6;

这些线的连接，阿波罗 STM32F767 开发板的内部已经连接好了，我们只需要将 OV5640 摄像头模块插上去就好了。**特别注意：**DCMI 摄像头接口和 SDIO 以及红外接收头有冲突，使用的时候，必须分时复用才可以，不可同时使用。另外，DCMI_PWDN 连接在 PCF8574T 的 P2 脚上，所以本章必须使用 PCF8574T，来间接控制 DCMI_PWDN。

实物连接如图 43.2.2 所示：

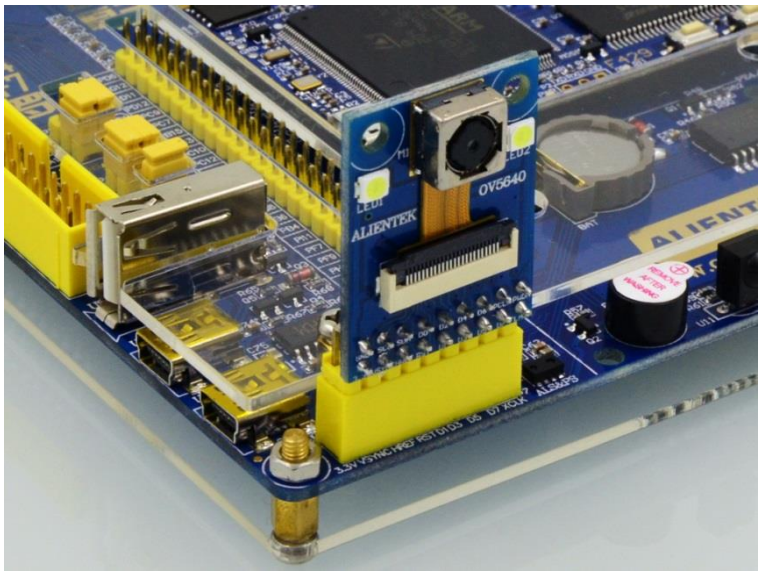


图 43.2.2 OV5640 摄像头模块与开发板连接实物图

43.3 软件设计

打开本章实验工程可以看到，因为本实验要使用定时器和串口 2，所以我们添加了 timer.c 和 usart2.c 文件。同时新建了 dcmi.c/dcmi.h，sccb.c/sccb.h 以及 ov5540.c/ov5540.h 等文件。

由于本实验代码比较多，我们就不一一列出了，仅挑几个重要的地方进行讲解。

首先，我们来看 ov5640.c 里面的 OV5640_Init 函数，该函数代码如下：

```
//初始化 OV5640
//配置完以后,默认输出是 1600*1200 尺寸的图片!!
//返回值:0,成功
//      其他,错误代码
u8 OV5640_Init(void)
{
    u16 i=0;
    u16 reg;
    //设置 IO
     GPIO_InitTypeDef GPIO_Initure;
    __HAL_RCC_GPIOA_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_Initure.Pin=GPIO_PIN_15;           //PA15
    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(GPIOA,&GPIO_Initure);     //初始化

    PCF8574_Init();             //初始化 PCF8574
    OV5640_RST(0);              //必须先拉低 OV5640 的 RST 脚,再上电
    delay_ms(20);
    OV5640_PWDN_Set(0);         //POWER ON
    delay_ms(5);
    OV5640_RST(1);              //结束复位
    delay_ms(20);
    SCCB_Init();                //初始化 SCCB  的 IO 口
    delay_ms(5);
    reg=OV5640_RD_Reg(OV5640_CHIPIDH);      //读取 ID  高八位
    reg<<=8;
    reg|=OV5640_RD_Reg(OV5640_CHIPIDL);     //读取 ID  低八位
    if(reg!=OV5640_ID)
    {
        printf("ID:%d\r\n",reg);
        return 1;
    }
    OV5640_WR_Reg(0x3103,0X11);         //system clock from pad, bit[1]
    OV5640_WR_Reg(0X3008,0X82);         //软复位
    delay_ms(10);
    //初始化  OV5640,采用 SXGA 分辨率(1600*1200)
    for(i=0;i<sizeof(ov5640_uxga_init_reg_tbl)/4;i++)
    {
        OV5640_WR_Reg(ov5640_uxga_init_reg_tbl[i][0],ov5640_uxga_init_reg_tbl[i][1]);
    }
    //检查闪光灯是否正常
    OV5640_Flash_Ctrl(1);//打开闪光灯
    delay_ms(50);
    OV5640_Flash_Ctrl(0);//关闭闪光灯
    return 0x00;    //ok
}
```

此部分代码先初始化 OV5640 相关的 IO 口（包括 PCF8574_Init 和 SCCB_Init），然后最主要的是完成 OV5640 的寄存器序列初始化。OV5640 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《OV5640_camera_module_software_application_notes_1.3_Sonix.pdf》），本章我们用到的配置序列，存放在 ov5640_ 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 ov5640cfg.h 里面。

另外，在 ov5640.c 里面，还有几个函数比较重要，这里不贴代码了，只介绍功能：

OV5640_ImageWin_Set 函数，该函数用于设置 ISP 输入窗口；

OV5640_OutSize_Set 函数，用于设置预缩放窗口和输出大小窗口；

OV5640_Focus_Init 函数，用于初始化自动对焦功能；

OV5640_Focus_Single 函数，用于实现一次自动对焦；

OV5640_Focus_Constant 函数，用于开启持续自动对焦功能；

OV5640_ImageWin_Set 和 OV5640_OutSize_Set 这就是我们在 43.1.1 节所介绍的 3 个窗口的设置，他们共同决定了图像的输出。

接下来，我们看看 ov5640cfg.h 里面 ov5640_init_reg_tbl 的内容，ov5640cfg.h 文件的代码如下：

```
//JPEG 配置.7.5 帧
//最大支持 2592*1944 的 JPEG 图像输出
const u16 OV5640_jpeg_reg_tbl[][2]=
{
    0x4300, 0x30, // YUV 422, YUYV
    ……//省略部分代码
    0x3503, 0x00, // AEC/AGC on
};
//RGB565 配置.15 帧
//最大支持 1280*800 的 RGB565 图像输出
const u16 ov5640_rgb565_reg_tbl[][2]=
{
    0x4300, 0X6F,
    ……//省略部分代码
    0x3503, 0x00, // AEC/AGC on
};
//OV5640 初始化寄存器序列表
const u16 ov5640_init_reg_tbl[][2]=
{
    // 24MHz input clock, 24MHz PCLK
    0x3008, 0x42, // software power down, bit[6]
    ……//省略部分代码
    0x4740, 0X21, //VSYNC  高有效
};
```

以上代码，我们省略了很多（全部贴出来太长了），里面总共有 3 个数组。我们大概了解下数组结构，每个数组条目的第一个字节为寄存器号（也就是寄存器地址），第二个字节为要设置的值，比如{0x4300, 0x30}，就表示在 0x4300 地址，写入 0X30 这个值。

这里面：ov5640_init_reg_tbl 数组，用于初始化 OV5640，该数组必须最先进行配置；ov5640_rgb565_reg_tbl 数组，用于设置 OV5640 的输出格式为 RGB565，分辨率为 1280*800，帧率为 15 帧，在 RGB 模式下使用；OV5640_jpeg_reg_tbl 用于设置 OV5640 的输出格式为 JPEG，分辨率为 2592*1944，帧率为 7.5 帧，在 JPEG 模式下使用。

接下来，我们看看 dcmi.c 里面的代码，如下：

```
DCMI_HandleTypeDef    DCMI_Handler;                 //DCMI 句柄
DMA_HandleTypeDef     DMADMCI_Handler;              //DMA 句柄

u8 ov_frame=0;                                      //帧率
extern void jpeg_data_process(void);    //JPEG 数据处理函数
```

```
//DCMI 初始化
void DCMI_Init(void)
{
    DCMI_Handler.Instance=DCMI;
    DCMI_Handler.Init.SynchroMode=DCMI_SYNCHRO_HARDWARE;//硬件同步
    DCMI_Handler.Init.PCKPolarity=DCMI_PCKPOLARITY_RISING; //PCLK 上升沿有效
    DCMI_Handler.Init.VSPolarity=DCMI_VSPOLARITY_LOW; //VSYNC 低电平有效
    DCMI_Handler.Init.HSPolarity=DCMI_HSPOLARITY_LOW; //HSYNC 低电平有效
    DCMI_Handler.Init.CaptureRate=DCMI_CR_ALL_FRAME; //全帧捕获
    DCMI_Handler.Init.ExtendedDataMode=DCMI_EXTEND_DATA_8B; //8 位数据格式
    HAL_DCMI_Init(&DCMI_Handler);          //初始化 DCMI，此函数会开启帧中断

    //关闭行中断、VSYNC 中断、同步错误中断和溢出中断
    __HAL_DCMI_DISABLE_IT(&DCMI_Handler,DCMI_IT_LINE| \
                                DCMI_IT_VSYNC|DCMI_IT_ERR|DCMI_IT_OVR);
    __HAL_DCMI_ENABLE_IT(&DCMI_Handler,DCMI_IT_FRAME);       //使能帧中断
    __HAL_DCMI_ENABLE(&DCMI_Handler);                        //使能 DCMI
}

//DCMI 底层驱动，引脚配置，时钟使能，中断配置
//此函数会被 HAL_DCMI_Init()调用
//hdcmi:DCMI 句柄
void HAL_DCMI_MspInit(DCMI_HandleTypeDef* hdcmi)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_DCMI_CLK_ENABLE();    //使能 DCMI 时钟

    __HAL_RCC_GPIOA_CLK_ENABLE();   //使能 GPIOA 时钟
    ……//省略部分时钟使能代码

    //初始化 PA6
    GPIO_Initure.Pin=GPIO_PIN_6;
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;      //推挽复用
    GPIO_Initure.Pull=GPIO_PULLUP;          //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;     //高速
    GPIO_Initure.Alternate=GPIO_AF13_DCMI;  //复用为 DCMI
    HAL_GPIO_Init(GPIOA,&GPIO_Initure);     //初始化

    ……//省略部分 IO 配置代码
    HAL_NVIC_SetPriority(DCMI_IRQn,2,2);    //抢占优先级 1，子优先级 2
    HAL_NVIC_EnableIRQ(DCMI_IRQn);          //使能 DCMI 中断
```

```
}

//DCMI DMA 配置
//mem0addr:存储器地址 0  将要存储摄像头数据的内存地址(也可以是外设地址)
//mem1addr:存储器地址 1  当只使用 mem0addr 的时候,该值必须为 0
//memblen:存储器位宽,可以为:DMA_MDATAALIGN_BYTE/
//DMA_MDATAALIGN_HALFWORD/DMA_MDATAALIGN_WORD
//meminc:存储器增长方式,可以为:DMA_MINC_ENABLE/DMA_MINC_DISABLE
void DCMI_DMA_Init(u32 mem0addr,u32 mem1addr,u16 memsize,u32 memblen,u32 meminc)
{
    __HAL_RCC_DMA2_CLK_ENABLE();//使能 DMA2 时钟
    __HAL_LINKDMA(&DCMI_Handler,DMA_Handle,DMADMCI_Handler);
                                        //将 DMA 与 DCMI 联系起来
    __HAL_DMA_DISABLE_IT(&DMADMCI_Handler,DMA_IT_TC);
        //先关闭 DMA 传输完成中断(否则在使用 MCU 屏的时候会出现花屏的情况)

    DMADMCI_Handler.Instance=DMA2_Stream1;        //DMA2 数据流 1
    DMADMCI_Handler.Init.Channel=DMA_CHANNEL_1; //通道 1
    DMADMCI_Handler.Init.Direction=DMA_PERIPH_TO_MEMORY; //外设到存储器
    DMADMCI_Handler.Init.PeriphInc=DMA_PINC_DISABLE; //外设非增量模式
    DMADMCI_Handler.Init.MemInc=meminc; //存储器增量模式
    DMADMCI_Handler.Init.PeriphDataAlignment=DMA_PDATAALIGN_WORD;
                                                    //外设数据长度:32 位
    DMADMCI_Handler.Init.MemDataAlignment=memblen; //存储器数据长度:8/16/32 位
    DMADMCI_Handler.Init.Mode=DMA_CIRCULAR; //使用循环模式
    DMADMCI_Handler.Init.Priority=DMA_PRIORITY_HIGH; //高优先级
    DMADMCI_Handler.Init.FIFOMode=DMA_FIFOMODE_ENABLE; //使能 FIFO
    DMADMCI_Handler.Init.FIFOThreshold=DMA_FIFO_THRESHOLD_HALFFULL;
                                                        //使用 1/2 的 FIFO
    DMADMCI_Handler.Init.MemBurst=DMA_MBURST_SINGLE; //存储器突发传输
    DMADMCI_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE; //外设突发单次传输
    HAL_DMA_DeInit(&DMADMCI_Handler); //先清除以前的设置
    HAL_DMA_Init(&DMADMCI_Handler); //初始化 DMA

    //在开启 DMA 之前先使用__HAL_UNLOCK()解锁一次 DMA
    __HAL_UNLOCK(&DMADMCI_Handler);
    if(mem1addr==0)      //开启 DMA，不使用双缓冲
    {
        HAL_DMA_Start(&DMADMCI_Handler,(u32)&DCMI->DR,mem0addr,memsize);
    }
    else //使用双缓冲
    {
        HAL_DMAEx_MultiBufferStart(&DMADMCI_Handler,(u32)&DCMI->DR, \
```

```
                                                mem0addr,mem1addr,memsize);//开启双缓冲
        __HAL_DMA_ENABLE_IT(&DMADMCI_Handler,DMA_IT_TC);//开启传输完成中断
            HAL_NVIC_SetPriority(DMA2_Stream1_IRQn,2,3);    //DMA 中断优先级
            HAL_NVIC_EnableIRQ(DMA2_Stream1_IRQn);
    }
}

//DCMI,启动传输
void DCMI_Start(void)
{
    LCD_SetCursor(0,0);
    LCD_WriteRAM_Prepare();                         //开始写入 GRAM
    __HAL_DMA_ENABLE(&DMADMCI_Handler);     //使能 DMA
    DCMI->CR|=DCMI_CR_CAPTURE;                      //DCMI 捕获使能
}

//DCMI,关闭传输
void DCMI_Stop(void)
{
    DCMI->CR&=~(DCMI_CR_CAPTURE);           //关闭捕获
    while(DCMI->CR&0X01);                           //等待传输完成
    __HAL_DMA_DISABLE(&DMADMCI_Handler);//关闭 DMA
}

//DCMI 中断服务函数
void DCMI_IRQHandler(void)
{
    HAL_DCMI_IRQHandler(&DCMI_Handler);
}

//捕获到一帧图像处理函数
//hdcmi:DCMI 句柄
void HAL_DCMI_FrameEventCallback(DCMI_HandleTypeDef *hdcmi)
{
    jpeg_data_process();//jpeg 数据处理
    LED1_Toggle;
    ov_frame++;
    //重新使能帧中断,因为 HAL_DCMI_IRQHandler()函数会关闭帧中断
    __HAL_DCMI_ENABLE_IT(&DCMI_Handler,DCMI_IT_FRAME);
}

void (*dcmi_rx_callback)(void);//DCMI DMA 接收回调函数
//DMA2 数据流 1 中断服务函数
```

```
void DMA2_Stream1_IRQHandler(void)
{
    if(__HAL_DMA_GET_FLAG(&DMADMCI_Handler,DMA_FLAG_TCIF1_5)!=RESET)
                                                    //DMA 传输完成
    {
        __HAL_DMA_CLEAR_FLAG(&DMADMCI_Handler,DMA_FLAG_TCIF1_5);
                                                    //清除 DMA 传输完成中断标志位
        dcmi_rx_callback();     //执行摄像头接收回调函数,读取数据等操作在这里面处理
    }
}
//////////////////////////////////////////////////////////////////////////////
//以下两个函数,供 usmart 调用,用于调试代码

//DCMI 设置显示窗口
//sx,sy;LCD 的起始坐标
//width,height:LCD 显示范围.
void DCMI_Set_Window(u16 sx,u16 sy,u16 width,u16 height)
{
    DCMI_Stop();
    LCD_Clear(WHITE);
    LCD_Set_Window(sx,sy,width,height);
    OV5640_OutSize_Set(0,0,width,height);
    LCD_SetCursor(0,0);
    LCD_WriteRAM_Prepare();                 //开始写入 GRAM
    __HAL_DMA_ENABLE(&DMADMCI_Handler); //开启 DMA2,Stream1
    DCMI->CR|=DCMI_CR_CAPTURE;              //DCMI 捕获使能
}

//通过 usmart 调试,辅助测试用.
//pclk/hsync/vsync:三个信号的有限电平设置
void DCMI_CR_Set(u8 pclk,u8 hsync,u8 vsync)
{
    HAL_DCMI_DeInit(&DCMI_Handler);//清除原来的设置
    DCMI_Handler.Instance=DCMI;
    DCMI_Handler.Init.SynchroMode=DCMI_SYNCHRO_HARDWARE;//硬件同步
    DCMI_Handler.Init.PCKPolarity=pclk<<5;          //PCLK 上升沿有效
    DCMI_Handler.Init.VSPolarity=vsync<<7;          //VSYNC 低电平有效
    DCMI_Handler.Init.HSPolarity=hsync<<6;          //HSYNC 低电平有效
    DCMI_Handler.Init.CaptureRate=DCMI_CR_ALL_FRAME;    //全帧捕获
    DCMI_Handler.Init.ExtendedDataMode=DCMI_EXTEND_DATA_8B;//8 位数据格式
    HAL_DCMI_Init(&DCMI_Handler);                   //初始化 DCMI
    DCMI_Handler.Instance->CR|=DCMI_MODE_CONTINUOUS;    //持续模式
}
```

其中：DCMI_IRQHandler 函数，用于处理帧中断，可以实现帧率统计（需要定时器支持）和 JPEG 数据处理等，实际上当捕获到一帧数据后，调用的是 HAL 库回调函数 HAL_DCMI_FrameEventCallback 进行处理，关于中断处理回调函数前面实验多次讲解，这里我们就不重复讲解处理过程了。DCMI_DMA_Init 函数，则用于配置 DCMI 的 DMA 传输，其外设地址固定为：DCMI->DR，而存储器地址可变（LCD 或者 SRAM）。DMA 被配置为循环模式，一旦开启，DMA 将不停的循环传输数据。DMA2_Stream1_IRQHandler 函数，用于在使用 RGB 屏的时候，双缓冲存储时，数据的搬运处理（通过 dcmi_rx_callback 函数实现）。DCMI_Init 函数用于初始化 STM32F7 的 DCMI 接口，这是根据在 43.1.2 节提到的配置步骤进行配置的。最后，DCMI_Start 和 DCMI_Stop 两个函数，用于开启或停止 DCMI 接口。

其他部分代码我们就不再细说了，请大家参考光盘本例程源码（实验 38 摄像头实验）。

最后，打开 main.c 文件，代码如下：

```
u8 ovx_mode=0;                                  //bit0:0,RGB565 模式;1,JPEG 模式
u16 curline=0;                                  //摄像头输出数据,当前行编号
u16 yoffset=0;                                  //y 方向的偏移量

#define jpeg_buf_size     30*1024*1024//定义 JPEG 数据缓存 jpeg_buf 的大小(1*4M 字节)
#define jpeg_line_size    2*1024          //定义 DMA 接收数据时,一行数据的最大值

u32 dcmi_line_buf[2][jpeg_line_size];  //RGB 屏时,摄像头采用一行一行读取,定义行缓存
u32 jpeg_data_buf[jpeg_buf_size] __attribute__((at(0XC0000000+1280*800*2)));
                                        //JPEG 数据缓存 buf,定义在 LCD 帧缓存之后

volatile u32 jpeg_data_len=0;           //buf 中的 JPEG 有效数据长度
volatile u8 jpeg_data_ok=0;             //JPEG 数据采集完成标志
                                        //0,数据没有采集完;
                                        //1,数据采集完了,但是还没处理;
                                        //2,数据已经处理完成了,可以开始下一帧接收

//JPEG 尺寸支持列表
const u16 jpeg_img_size_tbl[][2]=
{
    160,120,  //QQVGA
    ……//省略部分代码
    2592,1944,     //500W
};

const u8*EFFECTS_TBL[7]={"Normal","Cool","Warm","B&W","Yellowish ","Inverse",
                         "Greenish"}; //7 种特效
const u8*JPEG_SIZE_TBL[12]={"QQVGA","QVGA","VGA","SVGA","XGA","WXGA",
        "WXGA+","SXGA","UXGA","1080P","QXGA","500W"};//JPEG 图片  12 种尺寸
//处理 JPEG 数据
//当采集完一帧 JPEG 数据后,调用此函数,切换 JPEG BUF.开始下一帧采集.
void jpeg_data_process(void)
```

```
{
    u16 i;
    u16 rlen;                //剩余数据长度
    u32 *pbuf;
    curline=yoffset;     //行数复位
    if(ovx_mode&0X01)     //只有在 JPEG 格式下,才需要做处理.
    {
        if(jpeg_data_ok==0)      //jpeg 数据还未采集完?
        {
            __HAL_DMA_DISABLE(&DMADMCI_Handler);//关闭 DMA
            rlen=jpeg_line_size-__HAL_DMA_GET_COUNTER(&DMADMCI_Handler);
                                                    //得到剩余数据长度
            pbuf=jpeg_data_buf+jpeg_data_len;//偏移到有效数据末尾,继续添加
            if(DMADMCI_Handler.Instance->CR&(1<<19))for(i=0;i<rlen;i++)pbuf[i]=
                                    dcmi_line_buf[1][i];//读取 buf1 里面的剩余数据
            else for(i=0;i<rlen;i++)pbuf[i]=dcmi_line_buf[0][i];//读取 buf0 里面的剩余数据
            jpeg_data_len+=rlen;                //加上剩余长度
            jpeg_data_ok=1;                //标记 JPEG 数据采集完成,等待其他函数处理
        }
        if(jpeg_data_ok==2)      //上一次的 jpeg 数据已经被处理了
        {
            __HAL_DMA_SET_COUNTER(&DMADMCI_Handler,jpeg_line_size);
                                            //传输长度为 jpeg_buf_size*4 字节
            __HAL_DMA_ENABLE(&DMADMCI_Handler); //打开 DMA
            jpeg_data_ok=0;                        //标记数据未采集
            jpeg_data_len=0;                    //数据重新开始
        }
    }else
    {
        LCD_SetCursor(0,0);
        LCD_WriteRAM_Prepare();        //开始写入 GRAM
    }
}

//jpeg 数据接收回调函数
void jpeg_dcmi_rx_callback(void)
{
    u16 i;
    u32 *pbuf;
    pbuf=jpeg_data_buf+jpeg_data_len;//偏移到有效数据末尾
    if(DMADMCI_Handler.Instance->CR&(1<<19))//buf0 已满,正常处理 buf1
    {
        for(i=0;i<jpeg_line_size;i++)pbuf[i]=dcmi_line_buf[0][i];//读取 buf0 里面的数据
```

```
            jpeg_data_len+=jpeg_line_size;//偏移
        }else //buf1 已满,正常处理 buf0
        {
            for(i=0;i<jpeg_line_size;i++)pbuf[i]=dcmi_line_buf[1][i];//读取 buf1 里面的数据
            jpeg_data_len+=jpeg_line_size;//偏移
        }
        SCB_CleanInvalidateDCache();            //清除无效化 DCache
}

//JPEG 测试
//JPEG 数据,通过串口 2 发送给电脑.
void jpeg_test(void)
{
    u32 i,jpgstart,jpglen;
    u8 *p;
    u8 key,headok=0;
    u8 effect=0,contrast=2;
    u8 size=2;                  //默认是 QVGA 320*240 尺寸
    u8 msgbuf[15];              //消息缓存区
    ……//省略部分代码
    LCD_ShowString(30,180,200,16,16,msgbuf);//显示当前 JPEG 分辨率
    //自动对焦初始化
    OV5640_RGB565_Mode();  //RGB565 模式
    OV5640_Focus_Init();
    ……//省略部分代码
    OV5640_Focus_Constant();//启动持续对焦
    DCMI_Init();                //DCMI 配置
    dcmi_rx_callback=jpeg_dcmi_rx_callback;//JPEG 接收数据回调函数
    DCMI_DMA_Init((u32)&dcmi_line_buf[0],(u32)&dcmi_line_buf[1],
              jpeg_line_size,DMA_MDATAALIGN_WORD,DMA_MINC_ENABLE);
    OV5640_OutSize_Set(4,0,jpeg_img_size_tbl[size][0],jpeg_img_size_tbl[size][1]);
                                                        //设置输出尺寸
    DCMI_Start();               //启动传输
    while(1)
    {
        if(jpeg_data_ok==1)     //已经采集完一帧图像了
        {
            p=(u8*)jpeg_data_buf;
            printf("jpeg_data_len:%d\r\n",jpeg_data_len*4);//打印帧率
        LCD_ShowString(30,210,210,16,16,"Sending JPEG data..."); //提示正在传输数据
            jpglen=0;//设置 jpg 文件大小为 0
            headok=0;       //清除 jpg 头标记
            for(i=0;i<jpeg_data_len*4;i++)//查找 0XFF,0XD8 和 0XFF,0XD9,获取文件大小
```

```
                {
                    if((p[i]==0XFF)&&(p[i+1]==0XD8))//找到 FF D8
                    {
                        jpgstart=i;
                        headok=1;       //标记找到 jpg 头(FF D8)
                    }
                    if((p[i]==0XFF)&&(p[i+1]==0XD9)&&headok)//找到头以后,再找 FF D9
                    {
                        jpglen=i-jpgstart+2;
                        break;
                    }
                }
                if(jpglen) //正常的 jpeg 数据
                {
                    p+=jpgstart;                //偏移到 0XFF,0XD8 处
                    for(i=0;i<jpglen;i++)       //发送整个 jpg 文件
                    {
                        USART2->TDR=p[i];
                        while((USART2->ISR&0X40)==0);    //循环发送,直到发送完毕
                        key=KEY_Scan(0);
                        if(key)break;
                    }
                }
                if(key)    //有按键按下,需要处理
                {
                   ……//省略部分代码
                }else LCD_ShowString(30,210,210,16,16,"Send data complete!!");//提示结束
                jpeg_data_ok=2;    //标记 jpeg 数据处理完了,可以让 DMA 去采集下一帧了.
            }
        }
}

//RGB 屏数据接收回调函数
void rgblcd_dcmi_rx_callback(void)
{
    u16 *pbuf;
    if(DMA2_Stream1->CR&(1<<19))//DMA 使用 buf1,读取 buf0
    {
        pbuf=(u16*)dcmi_line_buf[0];
    }else                                //DMA 使用 buf0,读取 buf1
    {
        pbuf=(u16*)dcmi_line_buf[1];
    }
```

```
    LTDC_Color_Fill(0,curline,lcddev.width-1,curline,pbuf);//DM2D 填充
    if(curline<lcddev.height)curline++;
}

//RGB565 测试
//RGB 数据直接显示在 LCD 上面
void rgb565_test(void)
{
u8 key;
    u8 effect=0,contrast=2,fac;
    u8 scale=1;          //默认是全尺寸缩放
    u8 msgbuf[15];       //消息缓存区
    u16 outputheight=0;
    ……//省略部分代码
    LCD_ShowString(30,160,200,16,16,"KEY_UP:FullSize/Scale");       //1:1 尺寸
    //自动对焦初始化
    OV5640_RGB565_Mode();  //RGB565 模式
    OV5640_Focus_Init();
    ……//省略部分代码
    OV5640_Focus_Constant();//启动持续对焦
    DCMI_Init();              //DCMI 配置
    if(lcdltdc.pwidth!=0)     //RGB 屏
    {
        dcmi_rx_callback=rgblcd_dcmi_rx_callback;//RGB 屏接收数据回调函数
         DCMI_DMA_Init((u32)dcmi_line_buf[0],(u32)dcmi_line_buf[1],lcddev.width/2,
                       DMA_MDATAALIGN_HALFWORD,DMA_MINC_ENABLE);
    }else        //MCU 屏
    {
        DCMI_DMA_Init((u32)&LCD->LCD_RAM,0,1,
                      DMA_MDATAALIGN_HALFWORD,DMA_MINC_DISABLE);
    }
    TIM3->CR1&=~(0x01); //关闭定时器 3,关闭帧率统计（如果打开,RGB 屏会抖）
    if(lcddev.height>800)
    {
        yoffset=(lcddev.height-800)/2;
        outputheight=800;
        OV5640_WR_Reg(0x3035,0X51);//降低输出帧率，否则可能抖动
    }else
    {
        yoffset=0;
        outputheight=lcddev.height;
    }
    curline=yoffset;          //行数复位
```

```
        OV5640_OutSize_Set(4,0,lcddev.width,outputheight);      //满屏缩放显示
        DCMI_Start();                   //启动传输
        LCD_Clear(BLACK);
        while(1)
        {
            key=KEY_Scan(0);
            if(key)
            {
              ……//省略部分代码
            }
            delay_ms(10);
        }
}

int main(void)
{

        u8 key;
        u8 t;

        Cache_Enable();                 //打开 L1-Cache
        MPU_Memory_Protection();        //保护相关存储区域
        HAL_Init();                     //初始化 HAL 库
        Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
        ……//省略部分代码
        usart2_init(921600);             //初始化 USART2
        TIM3_Init(10000-1,10800-1);     //10Khz 计数,1 秒钟中断一次

        while(OV5640_Init())//初始化 OV5640
        {
          ……//省略部分代码
        }
        LCD_ShowString(30,130,200,16,16,"OV5640 OK");
        while(1)
        {
            key=KEY_Scan(0);
            if(key==KEY0_PRES){ovx_mode=0;break;}           //RGB565 模式
            else if(key==KEY1_PRES){ovx_mode=1;break;}      //JPEG 模式
            t++;
            if(t==100)LCD_ShowString(30,150,230,16,16,"KEY0:RGB565   KEY1:JPEG");
                                                            //闪烁显示提示信息
            if(t==200)
            {
```

```
                LCD_Fill(30,150,210,150+16,WHITE);
                t=0;
                LED0_Toggle;
            }
            delay_ms(5);
        }
        if(ovx_mode==1)jpeg_test();
        else rgb565_test();
    }
```

这部分代码比较长，我们省略了一些内容。详细的代码，请大家参考光盘本例程源码。注意，这里定义了一个非常大的数组 jpeg_data_buf（4MB），用来存储 JPEG 数据，因为 2592*1944 大小的 jpeg 图片，有可能大于 3MB，所以必须将这个数组尽量设置大一点。这个数组，我们定义在 SDRAM，由__attribute__关键字，指定数组地址，紧跟 LTDC GRAM 后的地址存放。

在 main.c 里面，总共有 6 个函数，我们接下来分别介绍。

1，jpeg_data_process 函数

该函数用于处理 JPEG 数据的接收，在 DCMI_IRQHandler 函数（在 dcmi.c 里面）里面被调用，它与 jpeg_dcmi_rx_callback 函数和 jpeg_test 函数共同控制 JPEG 的数据传送。JPEG 数据的接收，采用 DMA 双缓冲机制，缓冲数组为：dcmi_line_buf（u32 类型，RGB 屏接收 RGB565 数据时，也是用这个数组）；数组大小为：jpeg_line_size，我们定义的是 2*1024，即数组大小为 8K 字节（数组大小不能小于存储摄像头一行输出数据的大小）；JPEG 数据接收处理流程如图 43.3.1 所示：

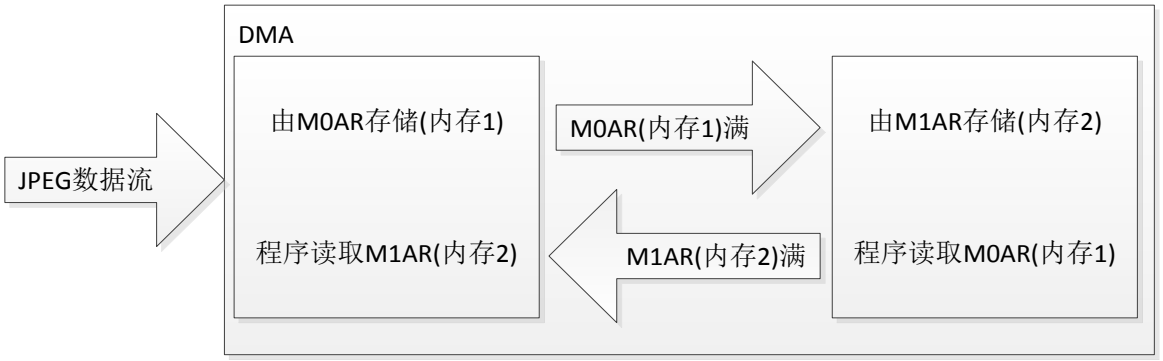


图 43.3.1 JPEG 数据流 DMA 双缓冲接收流程

JPEG 数据采集流程：当 JPEG 数据流传输给 MCU 的时候，首先由 M0AR 存储，此时如果 M1AR 有数据，则可以读取 M1AR 里面的数据，当 M0AR 数据满时，由 M1AR 存储，此时程序可以读取 M0AR 里面所存储的数据，当 M1AR 数据满时，由 M0AR 存储……。这个存储数据的操作，绝大部分是由 DMA 传输完成中断服务函数，调用 jpeg_dcmi_rx_callback 函数实现的，当一帧数据传输完成时，会进入 DCMI 帧中断服务函数，调用 jpeg_data_process 函数，对最后的剩余数据进行存储，完成一帧 JPEG 数据的采集。

2，jpeg_dcmi_rx_callback 函数

这是 jpeg 数据接收的主要函数，通过判断 DMA2_Stream1->CR 寄存器，读取不同 buf 里面的数据，存储到 SDRAM 里面（jpeg_data_buf）。该函数由 DMA 的传输完成中断服务函数：DMA2_Stream1_IRQHandler 调用。

3，jpeg_test 函数

该函数将 OV5640 设置为 JPEG 模式，并开启持续自动对焦，该函数实现 OV5640 的 JPEG

数据接收，并通过串口 2 发送给上位机软件。

4，rgblcd_dcmi_rx_callback 函数

该函数仅在使用 RGB 屏，且使用 RGB565 模式的时候用到。当使用 RGB 屏的时候，我们每接收一行数据，就使用 DMA2D 填充到 RGB 屏的 GRAM，这里同样是使用 DMA 的双缓冲机制来接收 RGB565 数据，原理参照图 43.3.1。该函数由 DMA 传输完成中断服务函数调用。

5，rgb565_test 函数

该函数将 OV5640 设置为 RGB565 模式，并将接收到的数据，传送给 LCD。当使用 MCU 屏的时候，完全由硬件 DMA 传输给 LCD，CPU 不用处理；当使用 RGB 屏的时候，数据先由 DMA 接收到双缓存里面，然后在 DMA 传输完成中断服务函数里面，调用函数：rgblcd_dcmi_rx_callback，将接收到的数据，用 DMA2D 填充到 RGB LCD，显示到屏幕上。

6，main 函数

该函数完成对各相关硬件的初始化，然后检测 OV5640，最后通过按键选择来调用 jpeg_test 还是 rgb565_test，实现 JPEG 测试和 RGB565 测试。

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 OV5640_WR_Reg 和 OV5640_RD_Reg 等相关函数，就可以轻松调试摄像头了。

43.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，在 OV5640 初始化成功后，屏幕提示选择模式，此时我们可以按 KEY0，进入 RGB565 模式测试，也可以按 KEY1，进入 JPEG 模式测试。

当按 KEY0 后，选择 RGB565 模式，LCD 满屏显示压缩放后的图像（有变形），如图 43.4.1 所示：

图 43.4.1 RGB565 模式测试图片

此时，可以按 KEY_UP 切换为 1:1 显示（不变形）。同时还可以通过 KEY0 按键，设置对比度；KEY1 按键，执行一次自动对焦；KEY2 按键，设置特效。

当按 KEY1 后，选择 JPEG 模式，此时屏幕显示 JPEG 数据传输进程，如图 43.4.2 所示：

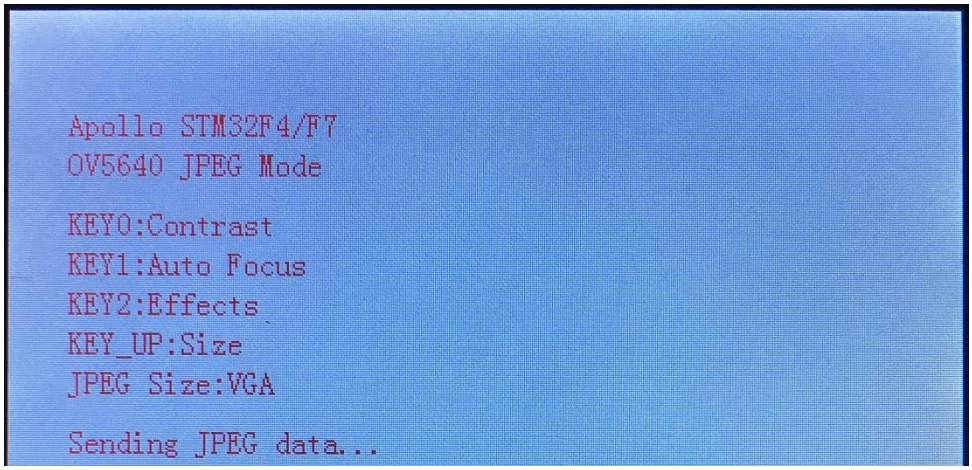


图 43.4.2 JPEG 模式测试图

默认条件下，图像分辨率是 VGA(640*480)的，硬件上：我们需要一根 RS232 串口线连接开发板的 COM2（注意要用跳线帽将 P8 的：COM2_RX 连接在 PA2(TX)）。如果没有 RS232 线，也可以借助我们开发板板载的 USB 转串口实现（有 2 个办法：1，改代码，将串口 2 输出改到串口 1；2，杜邦线连接 P8 的 PA2(TX)和 P4 的 RXD）。

我们打开上位机软件：ATK-CAM.exe（路径：光盘→6，软件资料→软件→串口&网络摄像头软件→ATK-CAM.exe），选择正确的串口，然后波特率设置为 921600，打开即可收到下位机传过来的图片了，如图 43.4.3 所示：

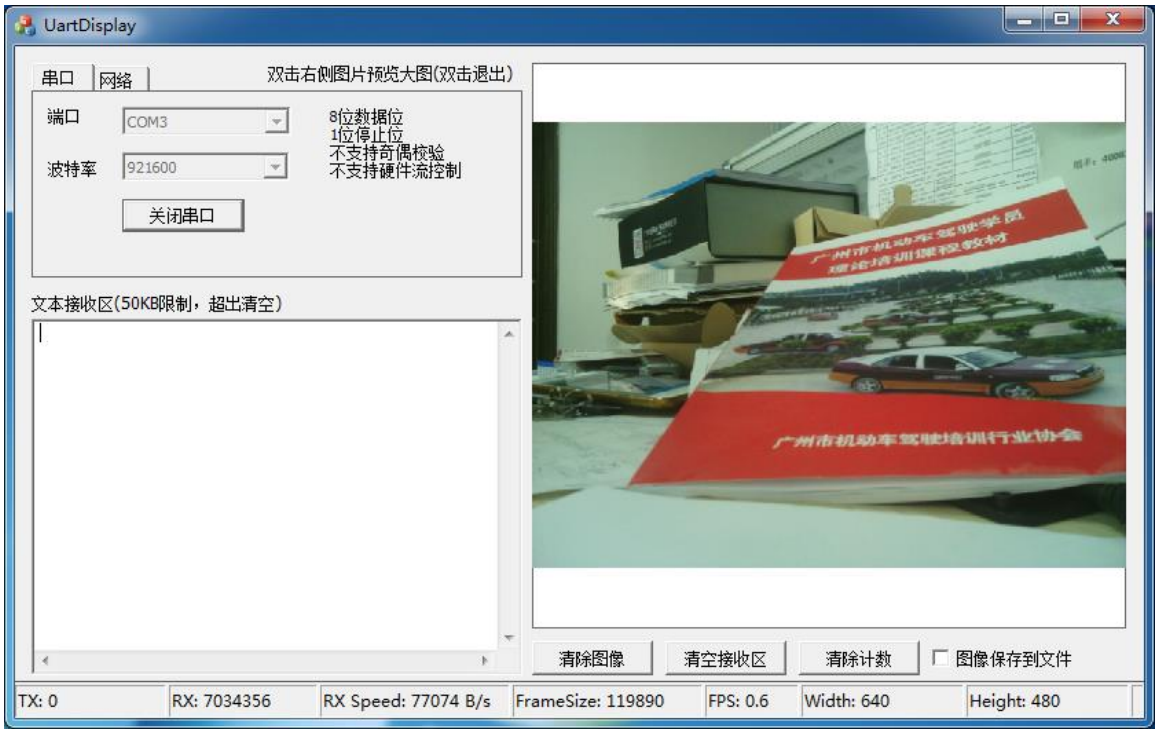


图 43.4.3 ATK-CAM 软件接收并显示 JPEG 图片

我们可以通过 KEY_UP 设置输出图像的尺寸（QQVGA~QSXGA）。通过 KEY0 按键，设置

对比度；KEY1 按键，执行一次自动对焦；KEY2 按键，设置特效。

同时，你还可以在串口（开发板的串口 1），通过 USMART 调用 SCCB_WR_Reg 等函数，来设置 OV5640 的各寄存器，达到调试测试 OV5640 的目的，如图 43.4.4 所示：

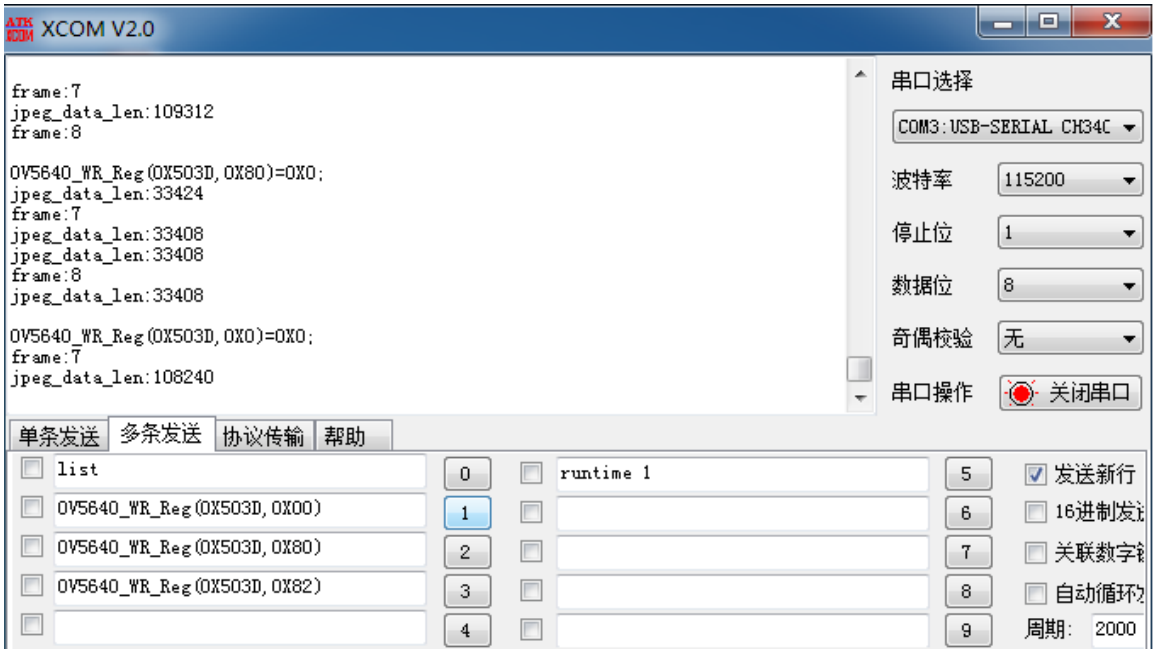


图 43.4.4 USMART 调试 OV5640

从上图还可以看出，帧率为 7/8 帧（实际上是 7.5 帧），每张 JPEG 图片的大小是 33KB 左右（分辨率为：640*480 的时候）。

第四十四章 内存管理实验

在第十九章，我们学会了使用 STM32F767 驱动外部 SDRAM，以扩展 STM32F767 的内存，加上 STM32F767 本身自带的 512K 字节内存，我们可供使用的内存还是比较多的。如果我们所用的内存都是直接定义一个数组来使用，灵活性会比较差，很多时候不能满足实际使用需求。

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：

44.1 内存管理简介
44.2 硬件设计
44.3 软件设计
44.4 下载验证

44.1 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 44.1.1 所示：

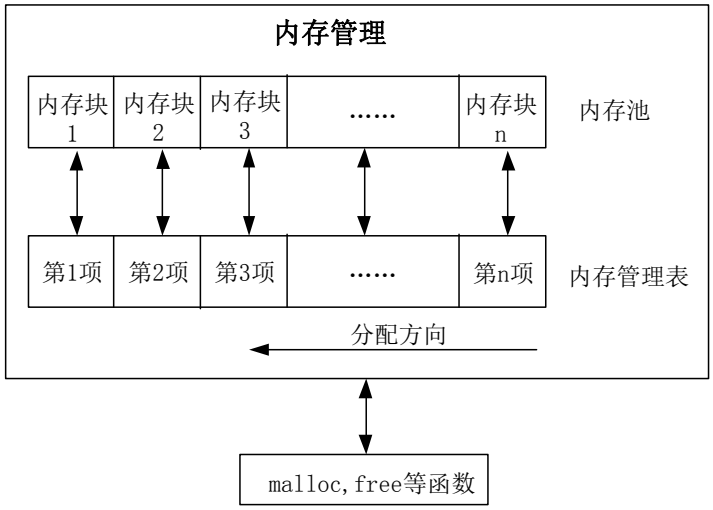


图 44.1.1 分块式内存管理原理

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。

44.2 硬件设计

本章实验功能简介：开机后，显示提示信息，等待外部输入。KEY0 用于申请内存，每次申请 2K 字节内存。KEY1 用于写数据到申请到的内存里面。KEY2 用于释放内存。KEY_UP 用于切换操作内存区（内部 SRAM 内存/外部 SDRAM 内存/内部 DTCM 内存）。DS0 用于指示程序运行状态。本章我们还可以通过 USMART 调试，测试内存管理函数。

本实验用到的硬件资源有：

1） 指示灯 DS0
2） 四个按键
3） 串口
4） LCD 模块
5） SDRAM

这些我们都已经介绍过，接下来我们开始软件设计。

44.3 软件设计

打开本章实验工程可以看到，我们新增了 MALLOC 分组，同时在分组中新建了文件 malloc.c 以及头文件 malloc.h。 内存管理相关的函数和定义主要是在这两个文件中。

打开 malloc.c 文件，代码如下：

```
//内存池(32 字节对齐)
__align(32) u8 mem1base[MEM1_MAX_SIZE];
            //内部 SRAM 内存池
__align(32) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0XC01F4000)));
      //外部 SDRAM 内存池,前面 2M 给 LTDC 用了(1280*800*2)
__align(32) u8 mem3base[MEM3_MAX_SIZE] __attribute__((at(0X20000000)));
      //内部 DTCM 内存池
//内存管理表
u32 mem1mapbase[MEM1_ALLOC_TABLE_SIZE];
                  //内部 SRAM 内存池 MAP
u32 mem2mapbase[MEM2_ALLOC_TABLE_SIZE]
    __attribute__((at(0XC01F4000+MEM2_MAX_SIZE)));      //外部 SRAM 内存池 MAP
u32 mem3mapbase[MEM3_ALLOC_TABLE_SIZE]
      __attribute__((at(0X20000000+MEM3_MAX_SIZE)));     //内部 DTCM 内存池 MAP
```

```
//内存管理参数
const u32 memtblsize[SRAMBANK]={MEM1_ALLOC_TABLE_SIZE,
        MEM2_ALLOC_TABLE_SIZE,MEM3_ALLOC_TABLE_SIZE};  //内存表大小
const u32 memblksize[SRAMBANK]={MEM1_BLOCK_SIZE,
        MEM2_BLOCK_SIZE,MEM3_BLOCK_SIZE};              //内存分块大小
const u32 memsize[SRAMBANK]={MEM1_MAX_SIZE,
        MEM2_MAX_SIZE,MEM3_MAX_SIZE};                  //内存总大小

//内存管理控制器
struct _m_mallco_dev mallco_dev=
{
      my_mem_init,                          //内存初始化
      my_mem_perused,                       //内存使用率
      mem1base,mem2base,mem3base,           //内存池
      mem1mapbase,mem2mapbase,mem3mapbase,//内存管理状态表
      0,0,0,                                //内存管理未就绪
};

//复制内存
//*des:目的地址
//*src:源地址
//n:需要复制的内存长度(字节为单位)
void mymemcpy(void *des,void *src,u32 n)
{
      u8 *xdes=des;
      u8 *xsrc=src;
      while(n--)*xdes++=*xsrc++;
}
//设置内存
//*s:内存首地址
//c :要设置的值
//count:需要设置的内存大小(字节为单位)
void mymemset(void *s,u8 c,u32 count)
{
      u8 *xs = s;
      while(count--)*xs++=c;
}
//内存管理初始化
//memx:所属内存块
void my_mem_init(u8 memx)
{
      mymemset(mallco_dev.memmap[memx],0,memtblsize[memx]*4);//内存状态表数据清零
      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(扩大了 10 倍,0~1000,代表 0.0%~100.0%)
u16 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*1000)/(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);
                                                    //拷贝旧内存内容到新内存
        myfree(memx,ptr);  //释放旧内存
        return (void*)((u32)mallco_dev.membase[memx]+offset);  //返回新内存首地址
    }
}
```

这里，我们通过内存管理控制器 mallco_dev 结构体（mallco_dev 结构体见 malloc.h），实现对三个内存池的管理控制。为甚

首先，是内部 SRAM 内存池，定义为：

```
__align(64) u8 mem1base[MEM1_MAX_SIZE];
```

然后，是外部 SDRAM 内存池，定义为：

```
__align(64) u8 mem2base[MEM2_MAX_SIZE] __attribute__((at(0XC01F4000)));
```

最后，是内部 DTCM 内存池，定义为：

```
__align(64) u8 mem3base[MEM3_MAX_SIZE] __attribute__((at(0X20000000)));
```

这里之所以要定义成 3 个，是因为这三个内存区域的地址都不一样，STM32F767 内部内存分为两大块：1，普通内存（地址从：0X2002 0000 开始，共 384KB），这部分内存任何外设都可以访问。2，DTCM 内存（地址从：0X2000 0000 开始，共 128KB），这部分内存可以被 CPU 和 DMA 等外设访问!!

而外部 SDRAM，地址是从 0XC000 0000 开始的，共 32768KB（32MB），但是，前面 2MB 用做 RGB LCD 屏的显存，不用于内存管理，所以，用于内存管理的外部 SDRAM 内存池首地址为：0XC01F4000（0XC000 0000+1280*800*2）。

这样总共有 3 部分内存，而内存池必须是连续的内存空间，才可以，这样 3 个内存区域，就有 3 个内存池，因此，分成了 3 块来管理。

其中，MEM1_MAX_SIZE、MEM2_MAX_SIZE 和 MEM3_MAX_SIZE 为在 malloc.h 里面定义的内存池大小，外部 SRAM 内存池指定地址为 0XC01F 4000，紧跟 RGB LCD 屏的显存之后，DTCM 内存池从 0X2000 0000 开始，是从 DTCM 内存的首地址开始的，而内部 SRAM 内存池的首地址则由编译器自动分配。__align(64)定义内存池为 64 字节对齐，以适应各种不同场合的需求。

此部分代码的核心函数为：my_mem_malloc 和 my_mem_free，分别用于内存申请和内存释放。思路就是我们在 43.1 接所介绍的那样分配和释放内存，不过这两个函数只是内部调用，外

部调用我们使用的是 mymalloc 和 myfree 两个函数。其他函数我们就不多介绍了。然后打开 malloc.h，关键代码如下：

```
#ifndef NULL
#define NULL 0
#endif

//定义三个内存池
#define SRAMIN     0        //内部内存池
#define SRAMEX     1        //外部内存池(SDRAM)
#define SRAMCCM    2        //CCM 内存池(此部分 SRAM 仅仅 CPU 可以访问!!!)

#define SRAMBANK    3     //定义支持的 SRAM 块数.

//mem1 内存参数设定.mem1 完全处于内部 SRAM 里面.
#define MEM1_BLOCK_SIZE                64                  //内存块大小为 64 字节
#define MEM1_MAX_SIZE              160*1024                //最大管理内存  160K
#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 SDRAM 里面
#define MEM2_BLOCK_SIZE                64                  //内存块大小为 64 字节
#define MEM2_MAX_SIZE              28912 *1024             //最大管理内存 28912K
#define MEM2_ALLOC_TABLE_SIZE      MEM2_MAX_SIZE/MEM2_BLOCK_SIZE
                                                       //内存表大小

//mem3 内存参数设定.mem3 处于 CCM,用于管理 CCM
#define MEM3_BLOCK_SIZE                64          //内存块大小为 64 字节
#define MEM3_MAX_SIZE              60 *1024        //最大管理内存 60K
#define MEM3_ALLOC_TABLE_SIZE      MEM3_MAX_SIZE/MEM3_BLOCK_SIZE
                                                   //内存表大小

//内存管理控制器
struct _m_mallco_dev
{
    void (*init)(u8);                   //初始化
    u16 (*perused)(u8);                 //内存使用率
    u8    *membase[SRAMBANK];           //内存池  管理 SRAMBANK 个区域的内存
    u32 *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)
u16 my_mem_perused(u8 memx) ;               //获得内存使用率(外/内部调用)
//////////////////////////////////////////////////////////////////////////////////
//用户调用函数
void myfree(u8 memx,void *ptr);             //内存释放(外部调用)
void *mymalloc(u8 memx,u32 size);           //内存分配(外部调用)
void *myrealloc(u8 memx,void *ptr,u32 size); //重新分配内存(外部调用)
#endif
```

这部分代码，定义了很多关键数据，比如内存块大小的定义：MEM1_BLOCK_SIZE、MEM2_BLOCK_SIZE 和 MEM3_BLOCK_SIZE，都是 64 字节。内存池总大小，内部 SRAM 内存池大小为 160K，外部 SDRAM 内存池大小为 28912K，内部 DTCM 内存池大小为 120K。

MEM1_ALLOC_TABLE_SIZE、MEM2_ALLOC_TABLE_SIZE 和 MEM3_ALLOC_TABLE_SIZE，则分别代表内存池 1、2 和 3 的内存管理表大小。

从这里可以看出，如果内存分块越小，那么内存管理表就越大，当分块为 4 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 u32 类型）。显然是不合适的，我们这里取 64 字节，比例为 1:16，内存管理表相对就比较小了。

最后来我们看看主函数代码：

```
int main(void)
{
    u8 paddr[20];                   //存放 P Addr:+p 地址的 ASCII 值
    u16 memused=0;
    u8 key,i=0,*p=0,*tp=0;
    u8 sramx=0;                     //默认为内部 sram

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    usmart_dev.init(108);           //初始化 USMART
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部内存池
    my_mem_init(SRAMDTCM);          //初始化 DTCM 内存池

    ……//此处省略部分代码
    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(sramx,2048);//申请 2K 字节
        if(p!=NULL)sprintf((char*)p,"Memory Malloc Test%03d",i);//向 p 写入一些内容
            break;
        case KEY1_PRES: //KEY1 按下
            if(p!=NULL)
            {
                sprintf((char*)p,"Memory Malloc Test%03d",i);//更新显示内容
                LCD_ShowString(30,270,200,16,16,p);              //显示 P 的内容
            }
            break;
        case KEY2_PRES: //KEY2 按下
            myfree(sramx,p);//释放内存
            p=0;              //指向空地址
            break;
        case WKUP_PRES:      //KEY UP 按下
            sramx++;
            if(sramx>2)sramx=0;
            if(sramx==0)LCD_ShowString(30,170,200,16,16,"SRAMIN ");
            else if(sramx==1)LCD_ShowString(30,170,200,16,16,"SRAMEX ");
            else LCD_ShowString(30,170,200,16,16,"SRAMDTCM");
            break;
    }
    if(tp!=p&&p!=NULL)
    {
        tp=p;
        sprintf((char*)paddr,"P Addr:0X%08X",(u32)tp);
        LCD_ShowString(30,250,200,16,16,paddr); //显示 p 的地址
        if(p)LCD_ShowString(30,270,200,16,16,p);//显示 P 的内容
        else LCD_Fill(30,270,239,266,WHITE);     //p=0,清除显示
    }
    delay_ms(10);
    i++;
    if((i%20)==0)//DS0 闪烁.
    {
        memused=my_mem_perused(SRAMIN);
        sprintf((char*)paddr,"%d.%01d%%",memused/10,memused%10);
```

```
                LCD_ShowString(30+112,190,200,16,16,paddr); //显示内部内存使用率
                memused=my_mem_perused(SRAMEX);
                sprintf((char*)paddr,"%d.%01d%%",memused/10,memused%10);
                LCD_ShowString(30+112,210,200,16,16,paddr); //显示外部内存使用率
                memused=my_mem_perused(SRAMDTCM);
                sprintf((char*)paddr,"%d.%01d%%",memused/10,memused%10);
                LCD_ShowString(30+112,230,200,16,16,paddr); //显示 CCM 内存使用率
                LED0_Toggle;
            }
        }
    }
```

该部分代码比较简单，主要是对 mymalloc 和 myfree 的应用。不过这里提醒大家，如果对一个指针进行多次内存申请，而之前的申请又没释放，那么将造成“内存泄露”，这是内存管理所不希望发生的，久而久之，可能导致无内存可用的情况！所以，在使用的时候，请大家一定记得，申请的内存在用完以后，一定要释放。

另外，本章希望利用 USMART 调试内存管理，所以在 USMART 里面添加了 mymalloc 和 myfree 两个函数，用于测试内存分配和内存释放。大家可以通过 USMART 自行测试。

44.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到如图 44.4.1 所示界面：

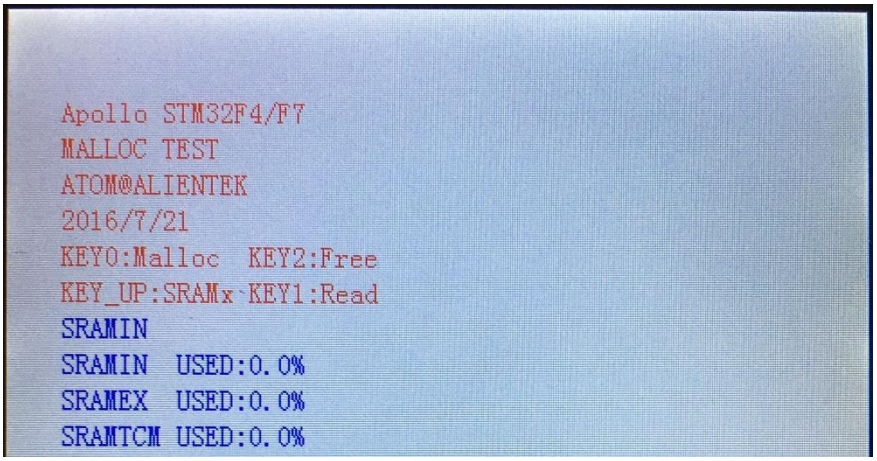


图 44.4.1 程序运行效果图

可以看到，所有内存的使用率均为 0%，说明还没有任何内存被使用，此时我们按下 KEY0，就可以看到内部 SRAM 内存被使用 1.2%了，同时看到下面提示了指针 p 所指向的地址（其实就是被分配到的内存地址）和内容。多按几次 KEY0，可以看到内存使用率持续上升（注意对比 p 的值，可以发现是递减的，说明是从顶部开始分配内存！），此时如果按下 KEY2，可以发现内存使用率降低了 1.2%，但是再按 KEY2 将不再降低，说明“内存泄露”了。这就是前面提到的对一个指针多次申请内存，而之前申请的内存又没释放，导致的“内存泄露”。

按 KEY_UP 按键，可以切换当前操作内存（内部 SRAM 内存/外部 SDRAM 内存/内部 CCM 内存），KEY1 键用于更新 p 的内容，更新后的内容将重新显示在 LCD 模块上面。**注意：当使外部 SDRAM 内存的时候，我们需要按很多次 KEY0（15 次），才可以看到内存使用率上升 0.1%，**

因为按一次只是申请 2K 字节，15 次才申请 30KB，内存总大小为 28912KB，所以，内存使用率为：30/28912=0.001（0.1%）。

第四十五章 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/SDMMC 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 阿波罗 STM32F767 开发板自带了标准的 SD 卡接口，使用 STM32F767 自带的 SDMMC 接口驱动，4 位模式，最高通信速度可达 48Mhz（分频器旁路时），最高每秒可传输数据 24M 字节，对于一般应用足够了。在本章中，我们将向大家介绍，如何在 ALIENTEK 阿波罗 STM32 开发板上实现 SD 卡的读取。本章分为如下几个部分：

45.1 SDMMC 接口简介
45.2 硬件设计
45.3 软件设计
45.4 下载验证

45.1.2 SDMMC 的时钟

从图 45.1.1.1 我们可以看到 SDMMC 总共有 3 个时钟，分别是：

卡时钟（SDMMC_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。

SDMMC 适配器时钟（SDMMCCLK）：该时钟用于驱动 SDMMC 适配器，来自 PLL48CK，一般为 48Mhz，并用于产生 SDMMC_CK 时钟（当系统时钟为 180M 的时候，PLL48CK=45Mhz）。

APB2 总线接口时钟（PCLK2）：该时钟用于驱动 SDMMC 的 APB2 总线接口，其频率为 HCLK/2，一般为 108Mhz。

前面提到，我们的 SD 卡时钟（SDMMC_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，SDMMC_CK 与 SDMMCCLK 的关系（时钟分频器不旁路时）为：

SDMMC_CK=SDMMCCLK/(2+CLKDIV)

其中，SDMMCCLK 为 PLL48CK，一般是 48Mhz，而 CLKDIV 则是分频系数，可以通过 SDMMC 的 SDMMC_CLKCR 寄存器进行设置（确保 SDMMC_CK 不超过卡的最大操作频率）。注意，以上公式，是时钟分频器不旁路时的计算公式，当时钟分频器旁路时，SDMMC_CK 直接等于 SDMMCCLK。

这里要提醒大家，在 SD 卡刚刚初始化的时候，其时钟频率（SDMMC_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）。

45.1.3 SDMMC 的命令与响应

SDMMC 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）。

SDMMC 的所有命令和响应都是通过 SDMMC_CMD 引脚传输的，任何命令的长度都是固定为 48 位，SDMMC 的命令格式如表 45.1.3.1 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 45.1.3.1 SDMMC 命令格式

所有的命令都是由 STM32F767 发出，其中开始位、传输位、CRC7 和结束位由 SDMMC 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 SDMMC_CMD 寄存器里面设置，命令参数则由寄存器 SDMMC_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。STM32F767 的 SDMMC 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 45.1.3.2 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 45.1.3.2 SDMMC 短响应格式

长响应的格式如表 45.1.3.3 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 45.1.3.3 SDMMC 长响应格式

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 SDMMC_RESPCMD 寄存器，参数则存放在 SDMMC_RESP1 寄存器里面。对于长响应，则仅留 CID/CSD 位域，存放在 SDMMC_RESP1~SDMMC_RESP4 等 4 个寄存器。

SD 存储卡总共有 5 类响应（R1、R2、R3、R6、R7），我们这里以 R1 为例简单介绍一下。R1（普通响应命令）响应输入短响应，其长度为 48 位，R1 响应的格式如表 45.1.3.4 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 45.1.3.4 R1 响应格式

在收到 R1 响应后，我们可以从 SDMMC_RESPCMD 寄存器和 SDMMC_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《STM32F7 中文参考手册》第 35 章。

最后，我们看看数据在 SDMMC 控制器与 SD 卡之间的传输。对于 SDI/SDMMC 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。

SDMMC（多）数据块读操作，如图 45.1.3.1 所示：

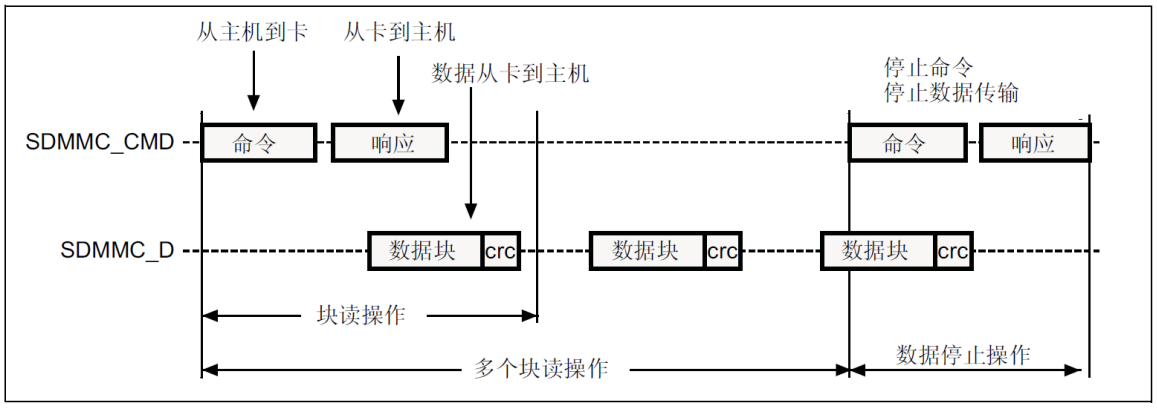


图 45.1.3.1 SDMMC（多）数据块读操作

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 SDMMC 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）。

SDMMC（多）数据块写操作，如图 45.1.3.2 所示：

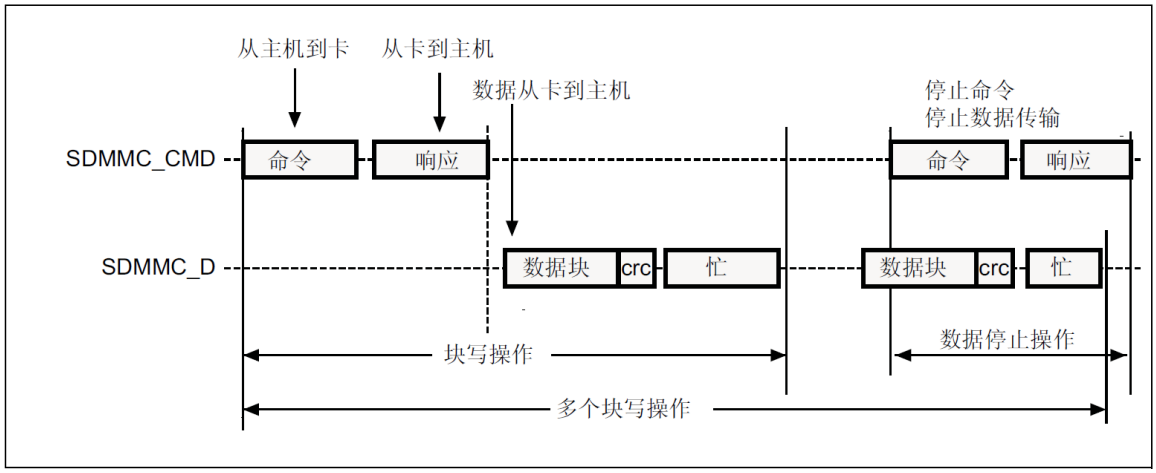


图 45.1.3.2 SDMMC（多）数据块写操作

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 SDMMC_D0，以表示繁忙，SDMMC 硬件自动控制，不需要我们软件处理。

SDMMC 的命令与响应就为大家介绍到这里。

45.1.4 SDMMC 相关寄存器介绍

第一个，我们来看 SDMMC 电源控制寄存器（SDMMC_POWER），该寄存器定义如图 45.1.4.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																														PWRCTRL	
rw	rw

位 31:2 保留，必须保持复位值

位 1:0 **PWRCTRL**：电源控制位 (Power supply control bits)。
这些位用于定义卡时钟的当前功能状态：
00：掉电：停止为卡提供时钟。 10：保留，上电
01：保留 11：通电：为卡提供时钟。

图 45.1.4.1 SDMMC_POWER 寄存器位定义

该寄存器复位值为 0，所以 SDMMC 的电源是关闭的，我们要启用 SDMMC，第一步就是要设置该寄存器最低 2 个位均为 1，让 SDMMC 上电，开启卡时钟。

第二个，我们看 SDMMC 时钟控制寄存器（SDMMC_CLKCR），该寄存器主要用于设置 SDMMC_CK 的分配系数，开关等，并可以设置 SDMMC 的数据位宽，该寄存器的定义如图 45.1.4.2 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
Reserved																	HWFC_EN	NEGEDGE	WIDBUS		BYPASS	PWRSAV	CLKEN	CLKDIV							
																	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 12:11 **WIDBUS**：宽总线模式使能位 (Wide bus mode enable bit)
00：默认总线模式：使用 SDIO_D0
01：4 位宽总线模式：使用 SDIO_D[3:0]
10：8 位宽总线模式：使用 SDIO_D[7:0]

位 10 **BYPASS**：时钟分频器旁路使能位 (Clock divider bypass enable bit)
0：禁止旁路：在驱动 SDIO_CK 输出信号前，根据 CLKDIV 值对 SDIOCLK 进行分频。
1：使能旁路：SDIOCLK 直接驱动 SDIO_CK 输出信号。

位 8 **CLKEN**：时钟使能位 (Clock enable bit)
0：禁止 SDIO_CK
1：使能 SDIO_CK

位 7:0 **CLKDIV**：时钟分频系数 (Clock divide factor)
该字段定义输入时钟 (SDIOCLK) 与输出时钟 (SDIO_CK) 之间的分频系数：
SDIO_CK 频率 = SDIOCLK / [CLKDIV + 2]。

图 45.1.4.2 SDMMC_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 SDMMC 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 SDMMC_CK，我们设置为 1。最后，CLKDIV，则用于控制 SDMMC_CK 的分频，一般设置为 0，即可得到 24Mhz 的 SDMMC_CK 频率。

第三个，我们要介绍的是 SDMMC 参数制寄存器（SDMMC_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 SDMMC 命令响应寄存器（SDMMC_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 SDMMC 响应寄存器组（SDMMC_RESP1~SDMMC_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 SDMMC_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 SDMMC_RESP1~ SDMMC_RESP4 里面，如表 45.1.4.1 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 45.1.4.1 响应类型和 SDMMC_RESPx 寄存器

第七个，我们介绍 SDMMC 命令寄存器（SDMMC_CMD），该寄存器各位定义如图 45.1.4.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																				SDIOSuspend	CPSMEN	WAITPEND	WAITINT	WAITRESP		CMDINDEX						
																				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 10 **CPSMEN**：命令路径状态机 (CPSM) 使能位 (Command path state machine (CPSM) Enable bit)
如果此位置 1，则使能 CPSM。

位 7:6 WAITRESP：等待响应位 (Wait for response bits)
这些位用于配置 CPSM 是否等待响应，如果等待，将等待哪种类型的响应。
00：无响应，但 CMDSENT 标志除外
01：短响应，但 CMDREND 或 CCRCFAIL 标志除外
10：无响应，但 CMDSENT 标志除外
11：长响应，但 CMDREND 或 CCRCFAIL 标志除外

位 5:0 **CMDINDEX**：命令索引 (Command index)
命令索引作为命令消息的一部分发送给卡。

图 45.1.4.3 SDMMC_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《STM32F7 中文参考手册》第 1095 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 SDMMC 数据定时器寄存器（SDMMC_DTIMER），该寄存器用于存储以卡总线时钟（SDMMC_CK）为周期的数据超时时间，一个计数器将从 SDMMC_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《STM32F7 中文参考手册》第 1099 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（SDMMC_DTIMER）和数据长度寄存器（SDMMC_DLEN）！

第九个，我们要介绍的是 SDMMC 数据长度寄存器（SDMMC_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 SDMMC_DCTRL 设置）的倍数。

第十个，我们要介绍的是 SDMMC 数据控制寄存器（SDMMC_DCTRL），该寄存器各位定义如图 45.1.4.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	SDIO EN	RW MOD	RW STOP	RW START	DBLOCKSIZE				DMA EN	DT MODE	DTDIR	DTEN
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 11 **SDIOEN**：SD I/O 使能功能 (SD I/O enable functions)
如果将该位置 1，则 DPSM 执行特定于 SD I/O 卡的操作。

位 10 **RWMOD**：读取等待模式 (Read wait mode)
0：通过停止 SDMMC_D2 进行读取等待控制
1：使用 SDMMC_CK 进行读取等待控制

位 9 **RWSTOP**：读取等待停止 (Read wait stop)
0：如果将 RWSTART 位置 1，则读取等待正在进行中
1：如果将 RWSTART 位置 1，则使能读取等待停止

位 8 **RWSTART**：读取等待开始 (Read wait start)
如果将该位置 1，则读取等待操作开始。

位 7:4 **DBLOCKSIZE**：数据块大小 (Data block size)
定义在选择了块数据传输模式时数据块的长度：
0000：（十进制数 0）块长度 = 2^0 = 1 字节
0001：（十进制数 1）块长度 = 2^1 = 2 字节
0010：（十进制数 2）块长度 = 2^2 = 4 字节
0011：（十进制数 3）块长度 = 2^3 = 8 字节
0100：（十进制数 4）块长度 = 2^4 = 16 字节
0101：（十进制数 5）块长度 = 2^5 = 32 字节
0110：（十进制数 6）块长度 = 2^6 = 64 字节
0111：（十进制数 7）块长度 = 2^7 = 128 字节
1000：（十进制数 8）块长度 = 2^8 = 256 字节
1001：（十进制数 9）块长度 = 2^9 = 512 字节
1010：（十进制数 10）块长度 = 2^{10} = 1024 字节
1011：（十进制数 11）块长度 = 2^{11} = 2048 字节
1100：（十进制数 12）块长度 = 2^{12} = 4096 字节
1101：（十进制数 13）块长度 = 2^{13} = 8192 字节
1110：（十进制数 14）块长度 = 2^{14} = 16384 字节
1111：（十进制数 15）保留

位 3 **DMAEN**：DMA 使能位 (DMA enable bit)
0：禁止 DMA。
1：使能 DMA。

位 2 **DTMODE**：数据传输模式选择 1 (Data transfer mode selection 1)：流或 SDIO 多字节数据传输。
0：块数据传输。
1：流或 SDIO 多字节数据传输。

位 1 **DTDIR**：数据传输方向选择 (Data transfer direction selection)
0：从控制器到卡。
1：从卡到控制器。

[0] **DTEN**：数据传输使能位 (Data transfer enabled bit)
如果 1 写入到 DTEN 位，则数据传输开始。根据方向位 DTDIR，如果在传输开始时立即将 RW 置 1开始，则 DPSM 变为 Wait_S 状态、Wait_R 状态或读取等待状态。在数据传输结束后不需要将使能位清零，但必须更新 SDMMC_DCTRL 以使能新的数据传输

图 45.1.4.4 SDMMC_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（SDMMC_STA）、清除中断寄存器（SDMMC_ICR）和中断屏蔽寄存器（SDMMC_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（SDMMC_STA）为例，该寄存器各位定义如图 45.1.4.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									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 22 **SDIOIT**：收到了 SDIO 中断 (SDIO interrupt received)

位 21 **RXDAVL**：接收 FIFO 中有数据可用 (Data available in receive FIFO)

位 20 **TXDAVL**：传输 FIFO 中有数据可用 (Data available in transmit FIFO)

位 19 **RXFIFOE**：接收 FIFO 为空 (Receive FIFO empty)

位 18 **TXFIFOE**：发送 FIFO 为空 (Transmit FIFO empty)
如果使能了硬件流控制，则 TXFIFOE 信号在 FIFO 包含 2 个字时激活。

位 17 **RXFIFOF**：接收 FIFO 已满 (Receive FIFO full)
如果使能了硬件流控制，则 RXFIFOF 信号在 FIFO 差 2 个字便变满之前激活。

位 16 **TXFIFOF**：传输 FIFO 已满 (Transmit FIFO full)

位 15 **RXFIFOHF**：接收 FIFO 半满：FIFO 中至少有 8 个字

位 14 **TXFIFOHE**：传输 FIFO 半空：至少可以写入 8 个字到 FIFO

位 13 **RXACT**：数据接收正在进行中 (Data receive in progress)

位 12 **TXACT**：数据传输正在进行中 (Data transmit in progress)

位 11 **CMDACT**：命令传输正在进行中 (Command transfer in progress)

位 10 **DBCKEND**：已发送/接收数据块（CRC 校验通过）

位 9 **STBITERR**：在宽总线模式下，并非在所有数据信号上都检测到了起始位

位 8 **DATAEND**：数据结束（数据计数器 SDIDCOUNT 为零）

位 7 **CMDSENT**：命令已发送（不需要响应）(Command sent (no response required))

位 6 **CMDREND**：已接收命令响应（CRC 校验通过）

位 5 **RXOVERR**：收到了 FIFO 上溢错误 (Received FIFO overrun error)

位 4 **TXUNDERR**：传输 FIFO 下溢错误 (Transmit FIFO underrun error)

位 3 **DTIMEOUT**：数据超时 (Data timeout)

位 2 **CTIMEOUT**：命令响应超时 (Command response timeout)
命令超时周期为固定值 64 个SDMMC_CK时钟周期。

位 1 **DCRCFAIL**：已发送/接收数据块（CRC 校验失败）

位 0 **CCRCFAIL**：已接收命令响应（CRC 校验失败）

图 45.1.4.5 SDMMC_STA 寄存器位定义

状态寄存器可以用来查询 SDMMC 控制器的当前状态，以便处理各种事务。比如 SDMMC_STA 的位 2 表示命令响应超时，说明 SDMMC 的命令响应出了问题。我们通过设置 SDMMC_ICR 的位 2 则可以清除这个超时标志，而设置 SDMMC_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 SDMMC 的数据 FIFO 寄存器（SDMMC_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 SDMMC_FIFO 寄存器，要写数据到 SD 卡，则要写 SDMMC_FIFO 寄存器。SDMMC 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 SDMMC_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，SDMMC 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《STM32F7 中文参考手册》第 35 章相关章节。

45.1.5 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 SDMMC 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 45.1.5.1 所示：

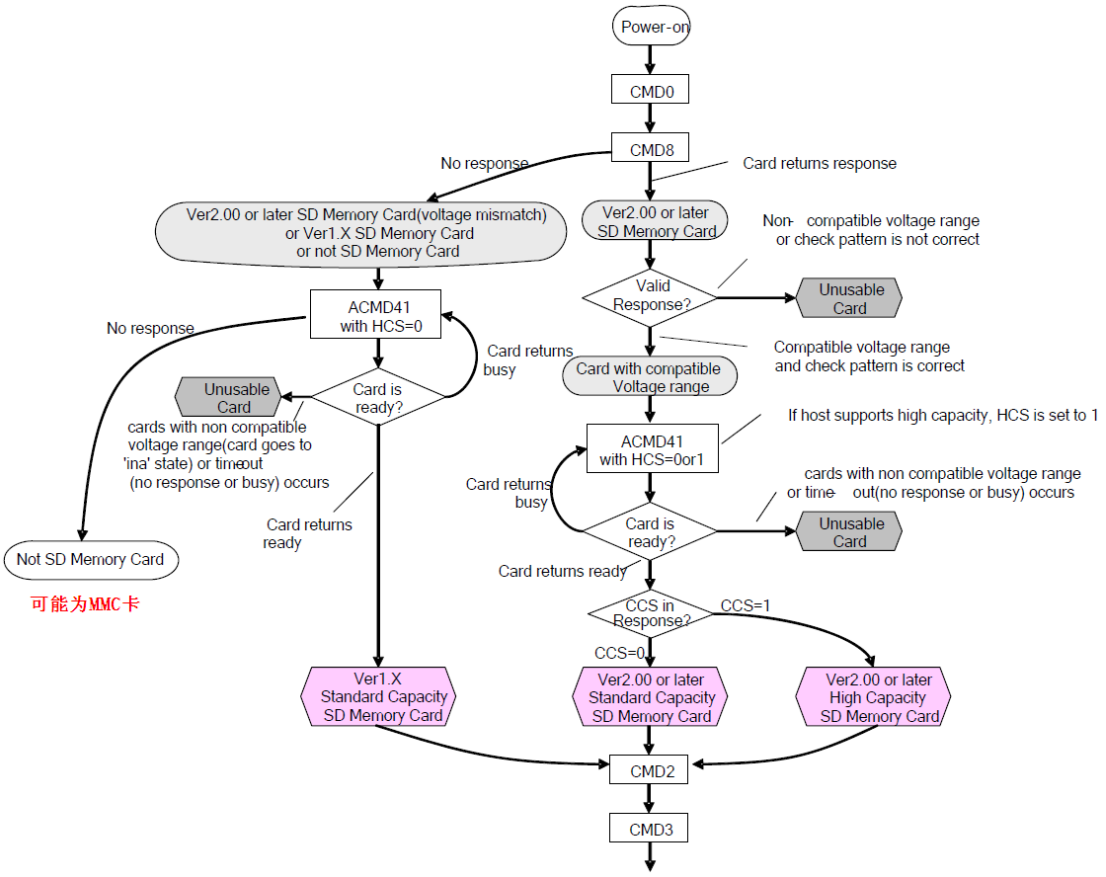


图 45.1.5.1 SD 卡初始化流程

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 SDMMC_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 45.1.5.1 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 45.1.5.1 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 45.1.5.2 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 45.1.5.2 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 45.1.5.3 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 45.1.5.3 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 45.1.5.4 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 45.1.5.4 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 45.1.5.1 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 45.1.5.5 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 45.1.5.5 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 SDMMC_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 SDMMC 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 SDMMC 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。

45.2 硬件设计

本章实验功能简介：开机的时候先初始化 SD 卡，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） LCD 模块

5） SD 卡

前面四部分，在之前的实例已经介绍过了，这里我们介绍一下阿波罗 STM32F767 开发板板载的 SD 卡接口和 STM32F767 的连接关系，如图 45.2.1 所示：

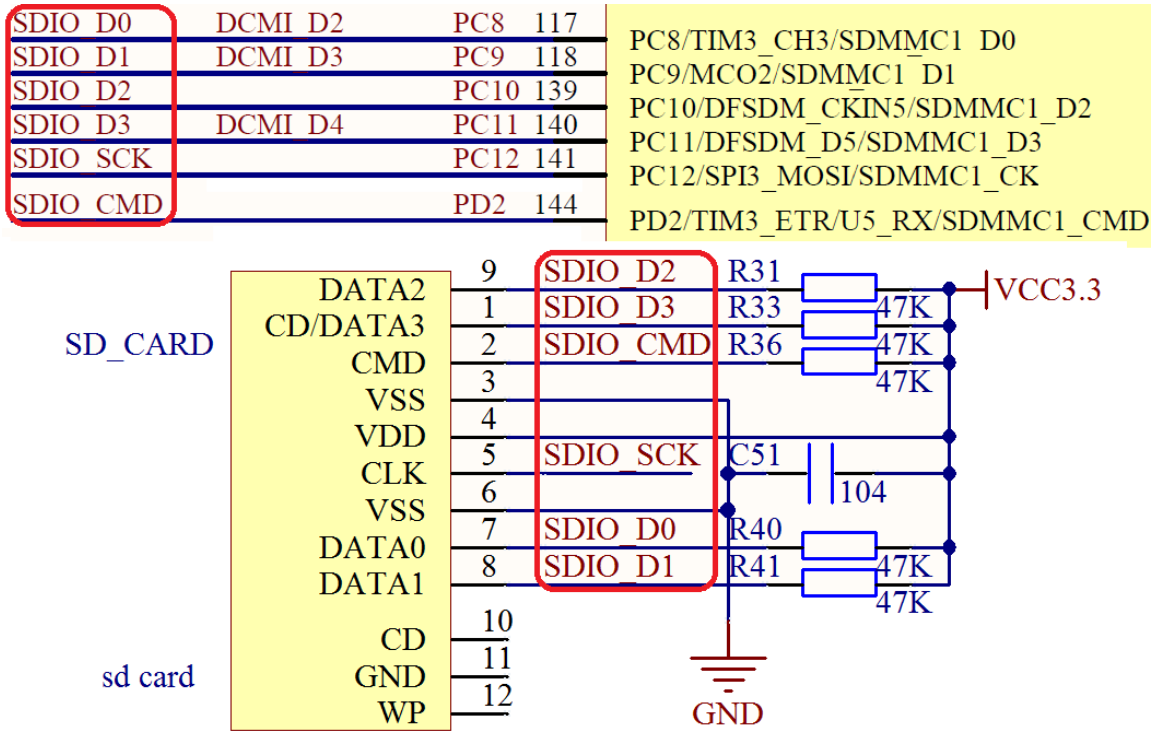


图45.2.1 SD卡接口与STM32F767连接原理图

阿波罗STM32F767开发板的SD卡座（SD_CARD），在PCB背面，SD卡座与STM32F767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。

45.3 软件设计

打开本章实验工程可以看到，我们增加了 hal 库 SD 卡支持源文件 stm32f7xx_hal_sd.c 和 stm32f7xx_II_sdmmc.h 以及对应头文件 stm32f7xx_hal_sd.h 和 stm32f7xx_II_sdmmc.h，同时我们还编写了源文件 sdmmc_sdcard.c 以及头文件 sdmmc_sdcard.h 用来存放 SD 卡相关函数。

HAL 库提供的 SD 驱动已经相当完整，我们只需要进行一些适配即可使用。接下来我们主要讲解 sdmmc_sdcard.c 中的源码。首先看看函数 SD_Init，内容如下：

```
//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=SDMMC1;
    SDCARD_Handler.Init.ClockEdge=SDMMC_CLOCK_EDGE_RISING;      //上升沿
    SDCARD_Handler.Init.ClockBypass=SDMMC_CLOCK_BYPASS_DISABLE;
```

```
                        //不使用 bypass 模式，直接用 HCLK 进行分频得到 SDMMC_CK
    SDCARD_Handler.Init.ClockPowerSave=SDMMC_CLOCK_POWER_SAVE_DISABLE;
                                                        //空闲时不关闭时钟电源
    SDCARD_Handler.Init.BusWide=SDMMC_BUS_WIDE_1B;          //1 位数据线
    SDCARD_Handler.Init.HardwareFlowControl=
                SDMMC_HARDWARE_FLOW_CONTROL_DISABLE;//关闭硬件流控
    SDCARD_Handler.Init.ClockDiv=SDMMC_TRANSFER_CLK_DIV;
                                                    //SD 传输时钟频率最大 25MHZ

    SD_Error=HAL_SD_Init(&SDCARD_Handler,&SDCardInfo);
    if(SD_Error!=SD_OK) return 1;

    SD_Error=HAL_SD_WideBusOperation_Config(&SDCARD_Handler,
                                            SDMMC_BUS_WIDE_4B);//使能宽总线模式
    if(SD_Error!=SD_OK) return 2;
    return 0;
}
```

对于该函数，主要调用函数 HAL_SD_Init 进行 SD 卡初始化流程并获取卡信息，有兴趣的同学可以看看 HAL 库中 HAL_SD_Init 函数完整内容。首先我们看看该函数声明：

```
HAL_SD_ErrorTypedef HAL_SD_Init(SD_HandleTypeDef *hsd,
                                HAL_SD_CardInfoTypedef *SDCardInfo);
```

HAL_SD_Init 函数内部先通过调用 HAL 库静态函数 SD_Initialize_Cards 来发送 CMD2 和 CMD3，获得 CID 寄存器内容和 SD 卡的相对地址（RCA），并通过 CMD9，获取 CSD 寄存器内容，完成 SD 卡的初始化流程。然后调用 HAL_SD_Get_CardInfo 函数来获取卡信息保存在入口参数 SDCardInfo 中。最后调用 SDMMC_Init 函数初始化 SDMMC 接口时钟相关参数。接下来我们分析函数 HAL_SD_Init 的入口参数。

```
typedef struct
{
  SD_TypeDef                *Instance;    //寄存器基地址
  SD_InitTypeDef            Init;         //SDMMC 初始化变量
  HAL_LockTypeDef           Lock;         //过程变量
  uint32_t                  CardType;     //卡类型
  uint32_t                  RCA;          //卡相对地址
  uint32_t                  CSD[4];       //保存 SD 卡 CSD 寄存器信息
  uint32_t                  CID[4];       //保存 SD 卡 CID 寄存器信息
  __IO uint32_t             SdTransferCplt; //非阻塞模式完成标志
  __IO uint32_t             SdTransferErr; //非阻塞模式错误标志
  __IO uint32_t             DmaTransferCplt; //dmac 传输结束标志
  __IO uint32_t             SdOperation;//sd 卡传输操作：读/写
  DMA_HandleTypeDef         *hdmarx;    //DMA 接收指针
  DMA_HandleTypeDef         *hdmatx;    //DMA 发送指针
}SD_HandleTypeDef;
```

该结构体的各个成员变量含义我们都在上面注释了。这里我们主要看看 Init 成员变量，它是 SD_InitTypeDef 结构体类型，用来设置 SDMMC 的初始化参数。该结构体实际是 SDMMC_InitTypeDef 结构体类型：

```
#define SD_InitTypeDef          SDMMC_InitTypeDef
```

接下来我们看看 SDMMC_InitTypeDef 结构体定义：

```
typedef struct
{
  uint32_t ClockEdge;// SDMMC CLK 上升沿/下降沿产生 SDMMC _CK
  uint32_t ClockBypass;          //时钟分频器旁路使能位
  uint32_t ClockPowerSave;       //节能模式配置位
  uint32_t BusWide;              //宽总线模式位数：默认/4 位/8 位
  uint32_t HardwareFlowControl;//硬件流控制使能
  uint32_t ClockDiv;//时钟分频银子
}SDMMC_InitTypeDef;
```

该结构体用在初始化 SDMMC 接口时钟相关参数，操作的是 SDMMC _CLKCR 寄存器。各位成员变量含义我们在后面已经注释了，这里我们就不累赘了，具体请参考我们前面讲解的 SDMMC _CLKCR 寄存器各位定义。

SD_Init 函数就给大家讲解到这里。接下来我们看看 HAL_SD_MspInit 函数，该函数主要由三个作用：第一是使能相应时钟，第二是初始化 SDMMC 相关 IO 口模式和映射，第三是在 DMA 模式下初始化 DMA 配置以及设置 NVIC。关于 DMA 相关知识情参考 DMA 实验讲解，这里我们就不累赘了，其他知识都比较简单。

接下来我们看看 SD_GetCardInfo 函数内容：

```
u8 SD_GetCardInfo(HAL_SD_CardInfoTypedef *cardinfo)
{
    u8 sta;
    sta=HAL_SD_Get_CardInfo(&SDCARD_Handler,cardinfo);
    return sta;
}
```

该函数非常简单，调用 HAL 库函数 HAL_SD_Get_CardInfo 获取卡信息保存在 cardinfo 中。

最后我们来看看在非 DMA 模式下 SD_ReadDisk 读卡函数和 SD_WriteDisk 写卡函数。函数内容如下：

```
//读 SD 卡
//buf:读数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 SD_ReadDisk(u8* buf,u32 sector,u8 cnt)
{
    u8 sta=SD_OK;
    long long lsector=sector;
    u8 n;
    lsector<<=9;
    INTX_DISABLE();//关闭总中断(POLLING 模式,严禁中断打断 SDIO 读写操作!!!)
```

```
    if((u32)buf%4!=0)
    {
        for(n=0;n<cnt;n++)
        {

        sta=HAL_SD_ReadBlocks(&SDCARD_Handler,(uint32_t*)SDIO_DATA_BUFFER,
                                                lsector+512*n,512,1);//单个 sector 的读操作
            memcpy(buf,SDIO_DATA_BUFFER,512);
            buf+=512;
        }
    }else
    {
        sta=HAL_SD_ReadBlocks(&SDCARD_Handler,(uint32_t*)buf,lsector,512,cnt);
                                                        //单个 sector 的读操作
    }
    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,u8 cnt)
{
    u8 sta=SD_OK;
    long long lsector=sector;
    u8 n;
    lsector<<=9;
    INTX_DISABLE();//关闭总中断(POLLING 模式,严禁中断打断 SDIO 读写操作!!!)
    if((u32)buf%4!=0)
    {
        for(n=0;n<cnt;n++)
        {
            memcpy(SDIO_DATA_BUFFER,buf,512);

        sta=HAL_SD_WriteBlocks(&SDCARD_Handler,(uint32_t*)SDIO_DATA_BUFFER,
                                                lsector+512*n,512,1);//单个 sector 的写操作
            buf+=512;
        }
    }else
    {
```

```
            sta=HAL_SD_WriteBlocks(&SDCARD_Handler,(uint32_t*)buf,lsector,512,cnt);
                                                        //多个 sector 的写操作
        }
        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，其中 SD_ReadDisk 函数用于读数据，通过调用 HAL 库 SD 卡读块函数 HAL_SD_ReadBlocks 实现。SD_WriteDisk 函数用于写数据，通过调用 HAL 库 SD 卡写块函数 HAL_SD_WriteBlocks 实现。注意，因为 FATFS 提供给 SD_ReadDisk 或者 SD_WriteDisk 的数据缓存区地址不一定是 4 字节对齐的，所以我们在这两个函数里面做了 4 字节对齐判断，如果不是 4 字节对齐的，则通过一个 4 字节对齐缓存（SDMMC_DATA_BUFFER）作为数据过度，以确保传递给底层读写函数的 buf 是 4 字节对齐的。

接下来我们看看 HAL 库提供的读块函数 HAL_SD_ReadBlocks 和写块函数 HAL_SD_WriteBlocks：声明

```
HAL_SD_ErrorTypedef HAL_SD_ReadBlocks(SD_HandleTypeDef *hsd, uint32_t
        *pReadBuffer, uint64_t ReadAddr, uint32_t BlockSize, uint32_t NumberOfBlocks);
HAL_SD_ErrorTypedef HAL_SD_WriteBlocks(SD_HandleTypeDef *hsd, uint32_t
        *pWriteBuffer, uint64_t WriteAddr, uint32_t BlockSize, uint32_t NumberOfBlocks);
```

HAL_SD_ReadBlocks 函数的作用是读取从 ReadAddr 地址开始，NumberOfBlocks 个块的数据，保存在 pReadBuffer 指针指向的连续存储空间中，同时块大小通过参数 BlockSize 设置。HAL_SD_WriteBlocks 函数的作用是把 pWriteBuffer 指向的连续存储空间中的数据写入从地址 WriteAddr 开始的一个连续区域，而连续区域大小为 NumberOfBlocks 个块，块大小通过 BlockSize 来设置。这两个函数使用起来比较简单，这里我们就不累赘了。

sdmmc_sdcard.c 内容我们就介绍到这里。sdmmc_sdcard.h 头文件中我们主要关注下面一行代码：

```
#define SD_DMA_MODE                 0     //1：DMA 模式，0：查询模式
```

宏定义标识符 SD_DMA_MODE 用来设置 SD 模式为 DMA 模式还是查询模式。默认情况下我们设置为查询模式，对于使用 DMA 模式操作代码，大家可以对照 sdmmc_sdcard.c 文件中代码学习。

最后我们看看 main.c 文件，代码如下：

```
//通过串口打印 SD 卡相关信息
void show_sdcard_info(void)
{
    switch(SDCardInfo.CardType)
    {
        case STD_CAPACITY_SD_CARD_V1_1:printf("Card Type:SDSC V1.1\r\n");break;
        case STD_CAPACITY_SD_CARD_V2_0:printf("Card Type:SDSC V2.0\r\n");break;
        case HIGH_CAPACITY_SD_CARD:printf("Card Type:SDHC V2.0\r\n");break;
        case MULTIMEDIA_CARD:printf("Card Type:MMC Card\r\n");break;
    }
    printf("Card ManufacturerID:%d\r\n",SDCardInfo.SD_cid.ManufacturerID);    //制造商 ID
    printf("Card RCA:%d\r\n",SDCardInfo.RCA);    //卡相对地址
```

```
        printf("Card Capacity:%d MB\r\n",(u32)(SDCardInfo.CardCapacity>>20));     //显示容量
        printf("Card BlockSize:%d\r\n\r\n",SDCardInfo.CardBlockSize);      //显示块大小
}
//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)
{
   …//此处省略函数定义
}
//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)
{
   …//此处省略函数定义
}

int main(void)
{
        u8 key;
        u32 sd_size;
        u8 t=0,*buf;

        Cache_Enable();                 //打开 L1-Cache
        HAL_Init();                     //初始化 HAL 库
        Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
        delay_init(216);                //延时初始化
        uart_init(115200);              //串口初始化
        usmart_dev.init(108);           //初始化 USMART
        LED_Init();                     //初始化 LED
        KEY_Init();                     //初始化按键
        SDRAM_Init();                   //初始化 SDRAM
        LCD_Init();                     //初始化 LCD
        my_mem_init(SRAMIN);            //初始化内部内存池
        my_mem_init(SRAMEX);            //初始化外部 SDRAM 内存池
        ……//此处省略部分液晶显示代码
        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_Toggle;//DS0 闪烁
        }
        show_sdcard_info();             //打印 SD 卡相关信息
        POINT_COLOR=BLUE;       //设置字体为蓝色
        //检测 SD 卡成功
        LCD_ShowString(30,150,200,16,16,"SD Card OK       ");
        LCD_ShowString(30,170,200,16,16,"SD Card Size:        MB");
        LCD_ShowNum(30+13*8,170,SDCardInfo.CardCapacity>>20,5,16);//显示 SD 卡容量
        while(1)
        {
            key=KEY_Scan(0);
            if(key==KEY0_PRES)//KEY0 按下了
            {
                buf=mymalloc(0,512);            //申请内存
                if(SD_ReadDisk(buf,0,1)==0)       //读取 0 扇区的内容
                {
                    LCD_ShowString(30,190,200,16,16,"USART1 Sending Data...");
                    printf("SECTOR 0 DATA:\r\n");
                    for(sd_size=0;sd_size<512;sd_size++)
                                        printf("%x ",buf[sd_size]);//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_Toggle;
                t=0;
            }
        }
    }
```

这里总共 4 个函数：

1，show_sdcard_info 函数

该函数用于从串口输出 SD 卡相关信息，包括：卡类型、制造商 ID、卡相对地址、容量和块大小等信息。

2，sd_test_read

该函数用于测试 SD 卡的读取，通过 USMART 调用，可以指定 SD 卡的任何地址，读取指定个数的扇区数据，将读到的数据，通过串口打印出来，从而验证 SD 卡数据的读取。

3，sd_test_write 函数

该函数用于测试 SD 卡的写入，通过 USMART 调用，可以指定 SD 卡的任何地址，写入指定个数的扇区数据，写入数据自动生成（都是 3 的倍数），写入完成后，在串口打印写入结果。我们可以通过 sd_test_read 函数，来检验写入数据是否正确。**注意：千万别乱写，否则可能把卡写成砖头/数据丢失!!写之前，先读取该地址的数据，最好全部是 0XFF 才写（全部 0X00 也行），其他情况最好别写！**

4，main 函数函数

该函数，先初化相关外设和 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。

最后，我们将 sd_test_read 和 sd_test_write 函数加入 USMART 控制，这样，我们就可以通过串口调试助手，测试 SD 卡的读写了，方便测试。

45.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 45.4.1 所示的内容（假设 SD 卡已经插上了）：

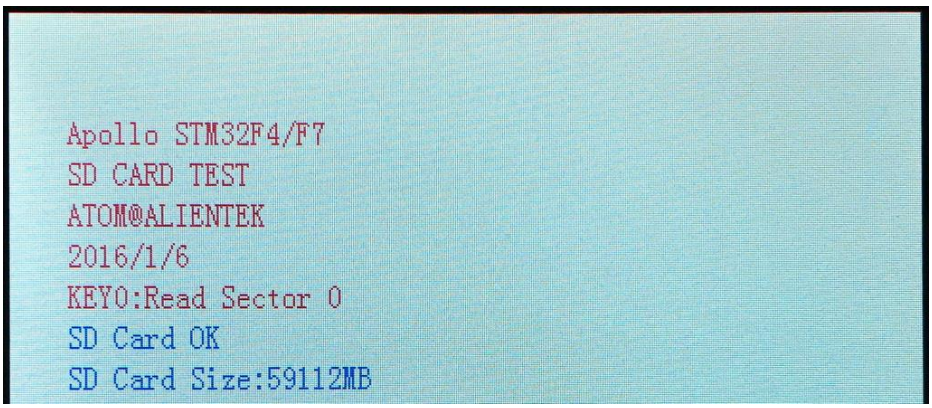


图 45.4.1 程序运行效果图

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 45.4.2 所示：

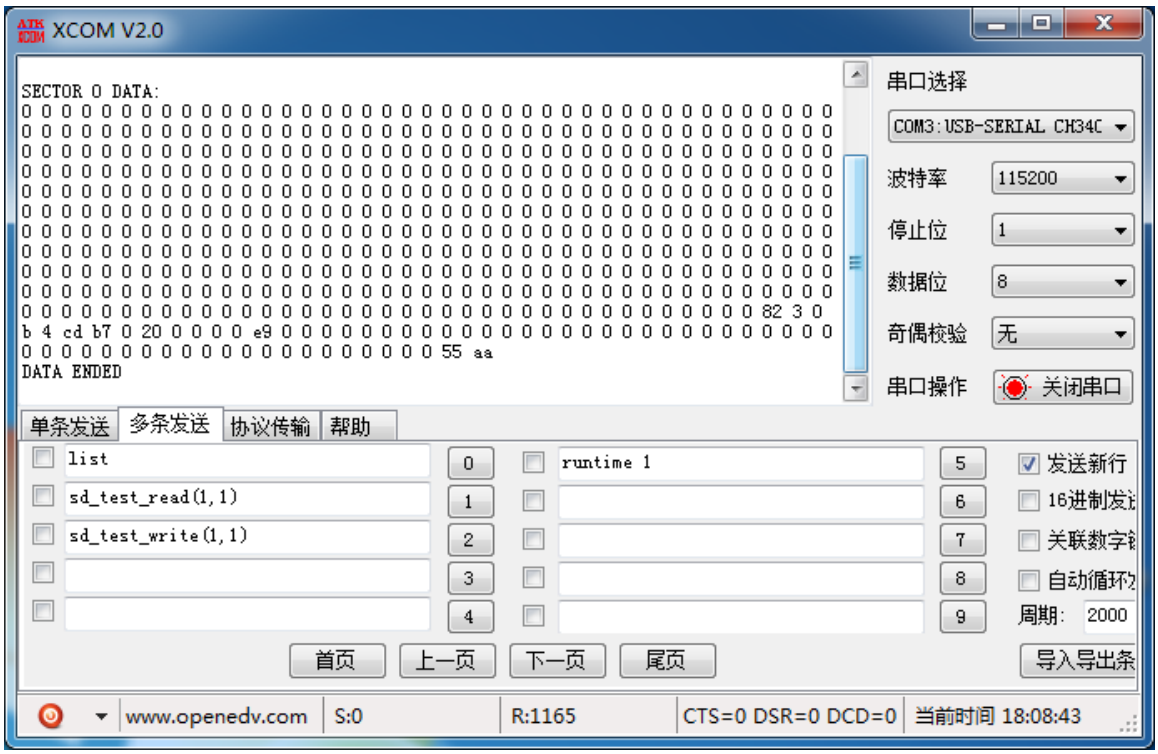


图 45.4.2 串口收到的 SD 卡扇区 0 内容

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 45.4.2 不同而感到惊讶。

最后，我们可以通过 USMART，调用 sd_test_read 和 sd_test_write 函数，来进一步检测 SD 卡的读取和写入，我们这里就不给大家演示了，大家自行测试即可。不过最后再一次**提醒大家：写 SD 卡的时候，千万别乱写，否则可能把卡写坏，或者丢失数据！！！**

第四十六章 NAND FLASH 实验

阿波罗 STM32F767 核心板上面，板载了一颗 512MB 的 NAND FLASH 芯片，型号为：MT29F4G08，我们可以用它来存储数据，相对于 SPI FLASH（W25Q256）和 SD 卡等存储设备，NAND FLASH 采用 8 位并口访问，具有访问速度快的优势。

本章，我们将使用 STM32F767 来驱动 MT29F4G08，并结合一个简单的坏块管理与磨损均衡算法，实现对 MT29F4G08 的读写控制。本章分为如下几个部分：

46.1 NAND FLASH 简介
46.2 硬件设计
46.3 软件设计
46.4 下载验证

46.1.1 NAND FLASH 简介

NAND FLASH 的概念是由东芝公司在 1989 年率先提出，它内部采用非线性宏单元模式，为固态大容量内存的实现提供了廉价有效的解决方案。NAND FLASH 存储器具有容量较大，改写速度快等优点，适用于大量数据的存储，在业界得到了广泛应用，如：SD 卡、TF 卡、U 盘等，一般都是采用 NAND FLASH 作为存储的。关于 NAND FLASH 的基础知识，请大家自行百度学习。接下来，我们介绍 NAND FLASH 的一些重要知识。

（1）NAND FLASH 信号线

NAND FLASH 的信号线如表 46.1.1.1 所示：

信号线	说明
CLE	命令锁存使能，高电平有效，表示写入的是命令
ALE	地址锁存使能，高电平有效，表示写入的是地址
CE#	芯片使能，低电平有效，用于选中 NAND 芯片
RE#	读使能，低电平有效，用于读取数据
WE#	写使能，低电平有效，用于写入数据
WP#	写保护，低电平有效
R/B	就绪/忙，注意用于判断编程/擦除操作是否完成
I/O0-7	地址/数据 输入/输出口

表 46.1.1.1 NAND FLASH 信号线

因为 NAND FLASH 地址/数据是共用数据线的，所以必须有 CLE/ALE 信号，告诉 NAND FLASH，发送的数据是命令还是地址。

（2）存储单元

NAND FLASH 存储单元介绍，我们以阿波罗 STM32F767 开发板所使用的 MT29F4G08（x8，8 位数据）为例进行介绍，MT29F4G08 的存储单元组织结构如图 46.1.1.1 所示：

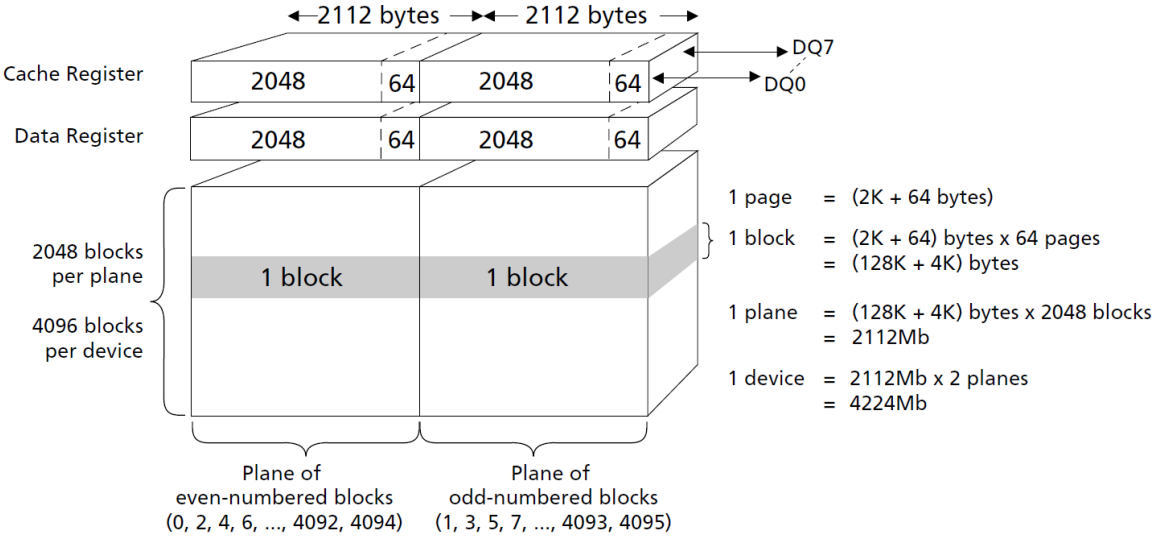


图 46.1.1.1 MT29F4G08 存储单元组织结构图

由图可知：MT29F4G08 由 2 个 plane 组成，每个 plane 有 2048 个 block，每个 block 由 64 个 page 组成，每个 page 有 2K+64 字节（2112 字节）的存储容量。所以，MT29F4G08 的总容量为：2*2048*64*（2K+64）= 553648128 字节（512MB）。其中，plane、block、page 等的个数根据 NAND FLASH 型号的不同，会有所区别，大家注意查看对应 NAND FLASH 芯片的数据手册。

NAND FLASH 的最小擦除单位是 block，对 MT29F4G08 来说，是（128+4）K 字节，NAND FLASH 的写操作具有只可以写 0，不能写 1 的特性，所以，在写数据的时候，必须先擦除 block（擦除后，block 数据全部为 1），才可以写入。

NAND FLASH 的 page 由 2 部分组成：数据存储区（data area）和备用区域（spare area），对 MT29F4G08 来说，数据存储区大小为 2K 字节，备用区域大小为 64 字节。我们存储的有效数据，一般都是存储在数据存储区（data area）。备用区域（spare area），一般用来存放 ECC（Error Checking and Correcting）校验值，在本章中，我们将利用这个区域，来实现 NAND FLASH 坏块管理和磨损均衡。

NAND FLASH 的地址分为三类：块地址（Block Address）、页地址（Page Address）和列地址（Column Address）。以 MT29F4G08 为例，这三个地址，通过 5 个周期发送，如表 46.1.1.2 所示：

周期	IO/7	IO/6	IO/5	IO/4	IO/3	IO/2	IO/1	IO/0
1	CA7	CA6	CA5	CA4	CA3	CA2	CA1	CA0
2	0	0	0	0	CA11	CA10	CA9	CA8
3	BA7	BA6	PA5	PA4	PA3	PA2	PA1	PA0
4	BA15	BA14	BA13	BA12	BA11	BA10	BA9	BA8
5	0	0	0	0	0	0	BA17	BA16

表 46.1.1.2 MT29F4G08 寻址说明

表中，CA0~CA11 为列地址（Column Address），用于在一个 Page 内部寻址，MT29F4G08 的一个 Page 大小为 2112 字节，需要 12 个地址线寻址；PA0~PA5 为页地址（Page Address），用于在一个 Block 内部寻址，MT29F4G08 一个 Block 大小为 64 个 Page，需要 6 个地址线寻址；BA6~BA17 为块地址（Block Address），用于块寻址，MT29F4G08 总共有 4096 个 Block，需要 12 根地址线寻址。

整个寻址过程，分 5 次发送（5 个周期），首先发送列地址，在发送页地址和块地址。这里

提醒一下：块地址和页地址，其实是可以写在一起的，由一个参数传递即可，所以表中的 BA 并不是由 BA0 开始的，大家可以理解为这个地址（PA+BA）为整个 NAND FLASH 的 Page 地址。在完成寻址以后，数据线 I/O0~ I/O7 来传输数据了。

（3）控制命令

NAND FLASH 的驱动需要用到一系列命令，这里我们列出常用的一些命令，给大家做一个简单介绍，方便大家了解 NAND FLASH 的操作，如表 46.1.1.3 所示：

命令（HEX）		名称	说明
1#	2#		
0X90		READID	读取 NAND 的 ID 和相关特性，可以此判断 NAND 的容量等信息
0XEF		SET FEATURE	设置 NAND 的相关参数，比如时序模式
0XFF		RESET	复位 NAND
0X70		READ STATUS	读取 NAND 的状态，比如可以判断编程/擦除操作是否完成
0X00	0X30	READ PAGE	该指令由 2 部分组成(分 2 次发)，用于读取一个 Page 里面的数据（不能跨页读）
0X80	0X10	WRITE PAGE	该指令由 2 部分组成(分 2 次发)，用于写入一个 Page 的数据（不能跨页写）
0X60	0XD0	ERASE BLOCK	该指令由 2 部分组成(分 2 次发)，用于擦除一个 Block
0X00	0X35	READ FOR INTERNAL DATA MOVE	这两个指令(分四次发)，组成 NAND 的内部数据移动操作，该操作可以实现拷贝一个 Page 到另外一个 Page（仅限同一 plane 内），且支持拷贝时写入数据，该操作可以极大的方便数据写入
0X85	0X10	PROGRAM FOR INTERNAL DATA MOVE	

表 46.1.1.3 NADN FLASH 操作常用命令

表 46.1.1.3 中，我们需要注意两点：1，有的指令一个周期完成传送，有的指令需要分两次传送（2 个周期）；2，指令名称，不同厂家的数据手册里面，标注可能不一样，但是其指令值（HEX 值）一般都是一样的。

上表中，前四条命令相对比较简单，这里我们主要介绍后面五条指令。

1，READ PAGE

该指令用于读取 NAND 的一个 Page（包括 spare 区数据，但不能跨页读），该指令时序如图 46.1.1.2 所示：

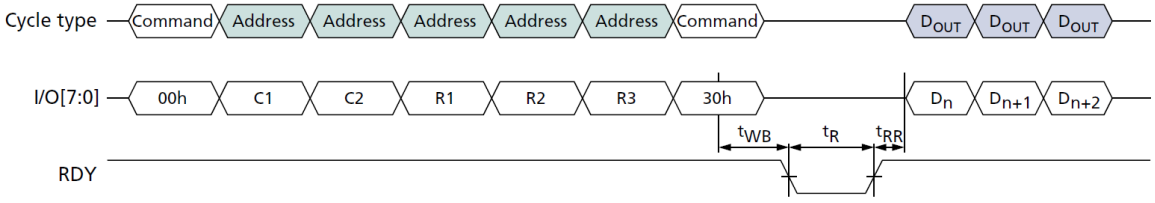


图 46.1.1.2 READ PAGE 指令时序图

由图可知，READ PAGE 的命令分两次发送，首先发送 00H 命令，然后发送 5 次地址（Block&Page&Column 地址），指定读取的地址，随后发送 30H 命令，在等待 RDY 后，即可读取 PAGE 里面的数据。注意：不能跨页读，所以最多一次读取一个 PAGE 的数据（包括 spare 区）。

2，WRITE PAGE

该指令用于写一个 Page 的数据（包括 spare 区数据，但不能跨页写），该指令时序如图 46.1.1.3 所示：

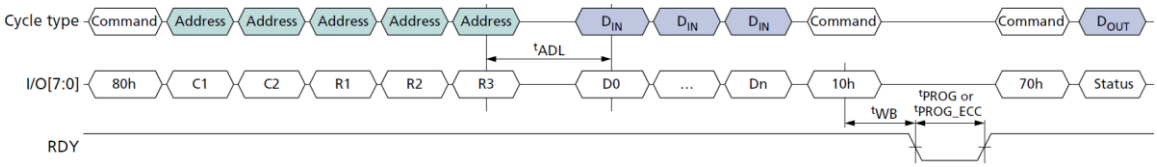


图 46.1.1.3 READ PAGE 指令时序图

由图可知，WRITE PAGE 的命令分两次发送，首先发送 80H 命令，然后发送 5 次地址（Block&Page&Column 地址），指定写入的地址，在地址写入完成后，等待 tADL 时间后，开始发送需要写入的数据，在数据发送完毕后，发送 10H 命令，最后发送 READ STATUS 命令，查询 NAND FLASH 状态，等待状态为 READY 后，完成一次 PAGE 写入操作。

3，ERASE BLOCK

该指令用于擦除 NAND 的一个 Block（NAND 的最小擦除单位），该指令时序如图 46.1.1.4 所示：

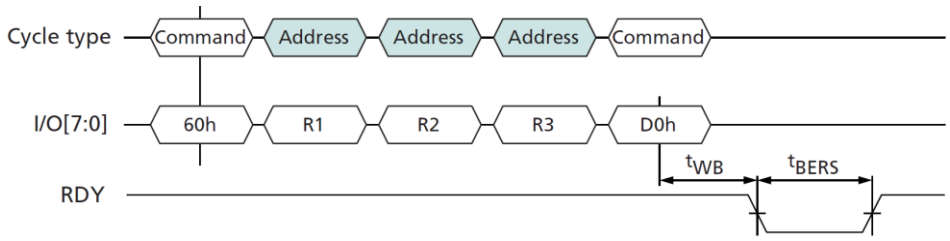


图 46.1.1.4 ERASE BLOCK 指令时序图

由图可知，ERASE BLOCK 的命令分两次发送，首先发送 60H 命令，然后发送 3 次地址（BLOCK 地址），指定要擦除的 BLOCK 地址，随后发送 D0H 命令，在等待 RDY 成功后，完成一个 BLOCK 的擦除。

4，READ FOR INTERNAL DATA MOVE

该指令用于在 NAND 内部进行数据移动时（页对页），指定需要读取的 PAGE 地址，如有必要，可以读取出 PAGE 里面的数据。该指令时序如图 46.1.1.5 所示：

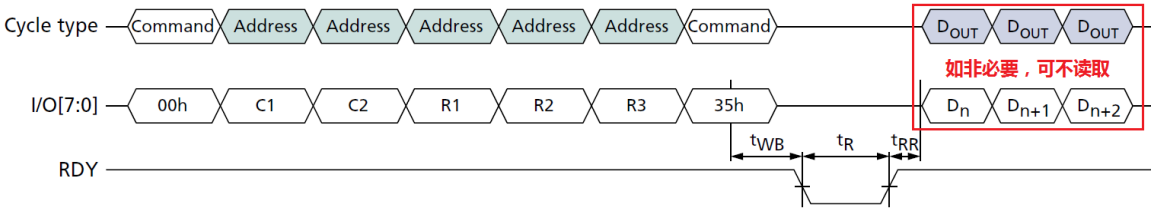


图 46.1.1.5 READ FOR INTERNAL DATA MOVE 指令时序图

由图可知，READ FOR INTERNAL DATA MOVE 的命令分两次发送，首先发送 00H 命令，然后发送 5 次地址（Block&Page&Column 地址），指定读取的地址，随后发送 35H 命令，在等待 RDY 后，可以读取对应 PAGE 里面的数据。在内部数据移动过程中，我们仅用该指令指定需要拷贝的 PAGE 地址（源地址），并不需要读取其数据，所以最后的 Dout 过程，一般都可以省略。

5，PROGRAM FOR INTERNAL DATA MOVE

该指令用于在 NAND 内部进行数据移动时（页对页），指定需要写入的 PAGE 地址（目标地址），如有必要，在拷贝过程中，可以写入新的数据，该指令时序如图 46.1.1.6 所示：

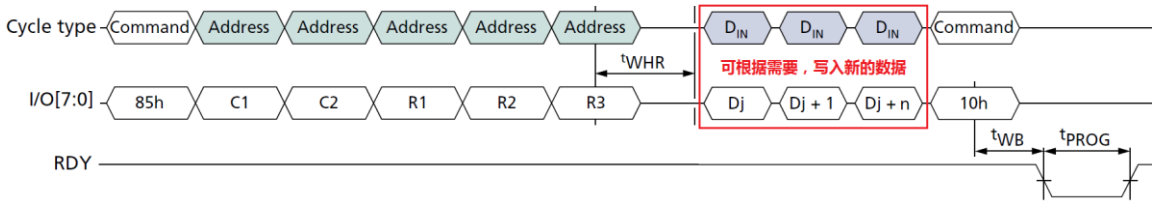


图 46.1.1.6 PROGRAM FOR INTERNAL DATA MOVE 指令时序图

如图 46.1.1.6 所示，该指令，首先发送 85H 命令，然后发送 5 次地址（Block&Page&Column 地址），指定写入的页地址（目标地址），源地址则由 READ FOR INTERNAL DATA MOVE 指令指定。接下来分两种情况：1，要写入新的数据（覆盖源 PAGE 的内容）；2，无需写入新的数据；

对于第 1 种情况，在等待 tWHR（或 tADL）之后，开始写入新的数据，数据在页内的起始地址，由 C1&C2 指定，写入完成后，发送 10H 命令，开始进行页拷贝，在等待 RDY 后，完成一次页对页的数据拷贝（带新数据写入）。

对于第 2 种情况，在发送完 5 次地址后，无需发送新数据，直接发送 10H 命令，开始进行页拷贝，在等待 RDY 后，完成一次页对页的数据拷贝（不带数据写入）。

注意：页对页拷贝，仅支持同一个 plane 里面互相拷贝（源地址和目标地址，必须在同一个 plane 里面），如果不是同一个 plane 里面的页，则不可以执行页对页拷贝。

NAND FLASH 其他命令的介绍，请大家参考 MT29F4G08 的数据手册。

（4）ECC 校验

ECC，英文全称为：Error checking and Correction，是一种对传输数据的错误检测和修正的算法。NAND FLASH 存储单元是串行组织的，当读取一个单元的时候，读出放大器所检测到信号强度会被这种串行存储结构削弱，这就降低了所读信号的准确性，导致读数出错（一般只有 1 个 bit 出错）。ECC 就可以检测这种错误，并修正错误的数据位，因此，ECC 在 NAND FLASH 驱动里面，被广泛使用。

ECC 有三种常用的算法：汉明码(Hamming Code)、RS 码(Reed Solomon Code)和 BCH 码。STM32 的 FMC 模块就支持硬件 ECC 计算，使用的就是汉明码，接下来，我们就给大家简单介绍一下汉明码的编码和使用。

1，汉明码编码

汉明码的编码计算比较简单，通过计算块上数据包得到 2 个 ECC 值（ECCo 和 ECCe）。为计算 ECC 值，数据包中的比特数据要先进行分割，如 1/2 组、1/4 组、1/8 组等，直到其精度达到单个比特为止。我们以 8 bit 即 1 字节的数据包为例进行说明，如表 46.1.1.4 所示：

数据位	7	6	5	4	3	2	1	0
1/8 偶校验		1		1		0		1
1/4 偶校验			0	1			0	1
1/2 偶校验					0	0	0	1
数据包	0	1	0	1	0	0	0	1
1/2 奇校验	0	1	0	1				
1/4 奇校验	0	1			0	0		
1/8 奇校验	0		0		0		0	

表 46.1.1.4 8bit 数据包校验的数据分割

8 位数据可以按：1/2、1/4 和 1/8 进行分割（1/8 分割时，达到单个比特精度）。简单说一下上表的看法，以 1/2 分割偶校验为例，1/2 分割时，每 4 个 bit 组成一个新 bit，新的 bit0 等于原来的 bit0~3，新的 bit1 等于原来的 bit4~7，而我们只要偶数位的数据，也就是新 bit0 的数据，

实际上就是原来的 bit0~3 的数据，这样就获取了 1/2 偶校验的数据。其他分割以此类推。

表 46.1.1.3 中，数据包上方的三行数据经计算后，得到偶校验值（ECCe），数据包下方的三行数据经计算后，得到奇校验值（ECCo）。1/2 校验值经“异或”操作构成 ECC 校验的最高有效位，1/4 校验值构成 ECC 校验的次高有效位，最低有效位由具体到比特的校验值填补。ECC 校验值（ECCo 和 ECCe）的计算过程，如图 46.1.1.7 所示：

	1/2	1/4	1/8	
ECCe=	0^0^0^1	0^1^0^1	1^1^0^1	=101
ECCo=	0^1^0^1	0^1^0^0	0^0^0^0	=010

图 46.1.1.7 计算奇偶 ECC 值

即偶校验值 ECCe 为“101”，奇校验值 ECCo 为“010”。图 1 所示为只有 1 字节数据的数据包，更大的数据包需要更多的 ECC 值。事实上，每 n bit 的 ECC 数值可满足 2^n bit 数据包的校验要求。不过，汉明码算法要求一对 ECC 数据（奇+偶），所以总共要求 2n bit 的 ECC 校验数据来处理 2^n bit 的数据包。

得到 ECC 值后，我们需要将原数据包和 ECC 数值都写入 NAND 里面。当原数据包从 NAND 中读取时，ECC 值将重新计算，如果新计算的 ECC 不同于先前编入 NAND 器件的 ECC，那么表明数据在读写过程中发生了错误。例如，原始数据 01010001 中有 1 个单一的比特出现错误，出错后的数据是 01010101。此时，重新计算 ECC 的过程如图 46.1.1.8 所示：

	1/2	1/4	1/8	
nECCe=	0^1^0^1	0^1^0^1	1^1^1^1	=000
nECCo=	0^1^0^1	0^1^0^1	0^0^0^0	=000

图 46.1.1.8 出错时计算的奇偶 ECC 值

可以看到，此时的 nECCo 和 nECCe 都为 000。此时，我们把所有 4 个 ECC 数值进行按位“异或”，就可以判断是否出现了 1 个单一比特的错误或者是多比特的错误。如果计算结果为全“0”，说明数据在读写过程中未发生变化。如果计算的结果为全“1”，表明发生了 1 bit 错误，其他情况，则说明有至少 2bit 数据出现了错误。不过，汉明码编码算法只能够保证更正单一比特的错误，如果两个或是更多的比特出错，汉明码就无能为力了（可以检测出错误，但无法修正）。不过，一般情况 SLC NAND 器件（STM32 仅支持 SLC NAND）出现 2bit 及以上的错误非常罕见，所以，我们使用的汉明码基本上够用。

对 4 个 ECC 进行异或的计算方法如下：

ECCe^ECCo^nECCe^nECCo=101^010^000^000=111

这样，经过上式计算，4 个 ECC 的“异或”结果为全 1，表示有 1 个 bit 出错了。对于这种 1 bit 错误的情况，出错的地址可通过将原有 ECCo 值和新 ECCo 值（nECCo）进行按位“异或”来得到，计算方法如下：

ECCo^nECCo=010^000=010

计算结果为 010（2），表明原数据第 2 bit 位出现了问题。然后，对出错后的数据的 bit2 进行取反（该位与 1“异或”即可），就可以得到正确的数据：01010101^00000100=01010001。

一个 8 位数据，需要 6 位 ECC 码，看起来效率不高，但是随着数据的增多，汉明码编码效率将会越来越高。比如，我们一般以 512 字节为单位来计算 ECC 值，只需要 24bit 的 ECC 码即可表示（2^12=4096bit=512 字节，12*2=24bit）。汉明码编码算法的原理，我们就给大家介绍到这里。

2，STM32 硬件 ECC

STM32 的 FMC 支持 NAND FLASH 硬件 ECC 计算，采用的就是汉明码计算方法。可以实现 1bit 错误的修正和 2bit 以上错误的检测，支持页大小按 256、512、1024、2048、4096 和 8192

字节为单位进行 ECC 计算。

当 FMC 的硬件 ECC 功能开启后，FMC 模块根据用户设置的参数（计算页大小、数据位宽等）对 NAND FLASH 数据线上传递的（读/写）数据进行 ECC 计算，数据传输结束后，ECC 计算结果自动存放在 FMC_ECCR 寄存器中。不过 STM32 的硬件 ECC 只负责计算 ECC 值，并不对数据进行修复。错误检测和数据修复，需要用户自己实现。另外，**STM32 的硬件 ECC 支持存储区域 3，其他存储区域不支持！！**

STM32 硬件 ECC 计算结果读取过程（以 512 字节页大小为例）：

1， 设置 FMC_PCR 的 ECCEN 位为 1，使能 ECC 计算

2， 写入/读取 512 字节数据

3， 等待 FMC_SR 的 FEMPT 位为 1（等待 FIFO 空）

4， 读取 FMC_ECCR，得到 ECC 值

5， 设置 FMC_PCR 的 ECCEN 位为 0，关闭 ECC，以便下一次重新计算

重复以上步骤，就可以在不同时刻进行读/写数据的 ECC 计算。在实际使用的时候，我们在写入/读取数据时，都要开启 STM32 的硬件 ECC 计算。写入的时候，将 STM32 硬件 ECC 计算出来的 ECC 值写入 NAND FLASH 数据所存 Page 的 spare 区。在读取数据的时候，STM32 硬件 ECC 会重新计算一个 ECC 值（ecccl），而从 spare 区对应位置，又可以读取之前写入的 ECC 值（eccrd），当这两个 ECC 值不相等的时候，说明读数有问题，需要进行 ECC 校验，ECC 检查和修正代码如下：

```
//获取 ECC 的奇数位/偶数位
//oe:0,偶数位
//     1,奇数位
//eccval:输入的 ecc 值
//返回值:计算后的 ecc 值(最多 16 位)
u16 NAND_ECC_Get_OE(u8 oe,u32 eccval)
{
    u8 i;
    u16 ecctemp=0;
    for(i=0;i<24;i++)
    {
        if((i%2)==oe) if((eccval>>i)&0X01)ecctemp+=1<<(i>>1);
    }
    return ecctemp;
}
//ECC 校正函数
//eccrd:读取出来,原来保存的 ECC 值
//ecccl:读取数据时,硬件计算的 ECC 只
//返回值:0,错误已修正
//       其他,ECC 错误(有大于 2 个 bit 的错误,无法恢复)
u8 NAND_ECC_Correction(u8* data_buf,u32 eccrd,u32 ecccl)
{
    u16 eccrdo,eccrde,eccclo,ecccle;
    u16 eccchk=0;
    u16 errorpos=0;
```

```
    u32 bytepos=0;
    eccrdo=NAND_ECC_Get_OE(1,eccrd);      //获取 eccrd 的奇数位
    eccrde=NAND_ECC_Get_OE(0,eccrd);      //获取 eccrd 的偶数位
    eccclo=NAND_ECC_Get_OE(1,ecccl);      //获取 ecccl 的奇数位
    ecccle=NAND_ECC_Get_OE(0,ecccl);      //获取 ecccl 的偶数位
    eccchk=eccrdo^eccrde^eccclo^ecccle;
    if(eccchk==0XFFF)                     //全 1,说明只有 1bit ECC 错误
    {
        errorpos=eccrdo^eccclo;           //计算出错 bit 位置
        bytepos=errorpos/8;               //计算字节位置
        data_buf[bytepos]^=1<<(errorpos%8); //对出错位进行取反，修正错误
    }else return 1;             //不是全 1,说明至少有 2bit ECC 错误,无法修复
    return 0;
}
```

经过以上代码处理，我们就可以利用 STM32 的硬件 ECC，修正 1bit 错误，并报告 2bit 及以上错误。

46.1.2 FTL 简介

因为 NAND FLASH 在使用过程中可能会产生坏块，且每个 BLOCK 的擦除次数是有限制的，超过规定次数后，BLOCK 将无法再擦除（即产生坏块），因此，我们需要这样一段程序，它可以实现：1，坏块管理；2，磨损均衡；从而使应用程序可以很方便的访问 NAND FLASH（无需关系坏块问题），且最大限度的延长 NAND FLASH 的寿命。

这里给大家介绍 FTL，FTL 是 Flash Translation Layer 的简写，即闪存转换层，它是一个 NAND 闪存芯片与基础文件系统之间的一个转换层，它自带了坏块管理和磨损均衡算法，使得操作系统和文件系统能够像访问硬盘一样访问 NAND 闪存设备，而无需关心坏块和磨损均衡问题。

本章，我们将给大家介绍一个比较简单的 FTL 层算法，它可以支持坏块管理和磨损均衡，提供支持文件系统（如：FATFS）的访问接口，通过这个 FTL，我们可以很容易的实现 NAND FLASH 的文件系统访问。

要做好 NAND FLASH 的坏块管理，我们有以下几点需要实现：

1， 如何识别坏块，标记坏块；

2， 转换表

3， 保留区

1，如何识别坏块，标记坏块

经过前面的介绍，我知道 NAND 在使用过程中，会产生坏块，而坏块我们是不能再用来存储数据的，必须对坏块进行识别和标记，并保存这些标记。

NAND FLASH 的坏块识别有几种方式：1，NAND 厂家出厂的时候，会在每个 Block 的第一个 page 和第二个 page 的 spare 区的第一个字节写入非 0XFF 的值来表示，我们可以通过这个判断该块是否为坏块；2，通过给每个 Block 写入数值（0XFF/0X00），然后读取出来，判断写入的数据和读取的数据是否完全一样，来识别坏块；3，通过读取数据时，校验 ECC 错误，来识别坏块。

NAND FLASH 的坏块标记：坏块标记，我们使用每个 Block 的第一个 page 和第二个 page（第二个 page 是备份用的）spare 区的第一个字节来标记，当这个字节的值为 0XFF 时，表示

该块为好块，当这个字节的值不等于 0XFF 时，表示该块为坏块。以 MT29F4G08 为例，坏块表示方法，如表 46.1.2.1 所示：

BLOCK 编号	PAGE 编号	数据区			Spare 区			
		0	……	2047	0	1	……	63
n	0		……		0XAA			
	1		……		0XAA			
	……	……						
	63		……					

表 46.1.2.1 NAND FLASH 坏块标记说明表

上图中，假设某个 Block 为坏块，那么它的第一个 page 和第二个 page 的 spare 区第一个字节，就不是 0XFF 了（我们改为 0XAA），以表示其是一个坏块。如果是好块，这两个字节，必须都是 0XFF，只要任何一个不是 0XFF，则表示该块是一个坏块。

这样，我们只需要判断每个 Block 的第一和第二个 page 的 spare 区的第一个字节，就可以判断是否为坏块。达到了标记和保存坏块标记的目的。

2，转换表

文件系统访问文件的时候，使用的是逻辑地址，是按顺序编号的，它不考虑坏块情况。而 NAND FLASH 存储地址，我们称为物理地址，是有可能存在坏块的。所以，这两个地址之间，必须有一个映射表，将逻辑地址转换为物理地址，且不能指向坏块的物理地址，这个映射表我们称之为逻辑地址-物理地址转换表，简称转换表，如图 46.1.2.1 所示：

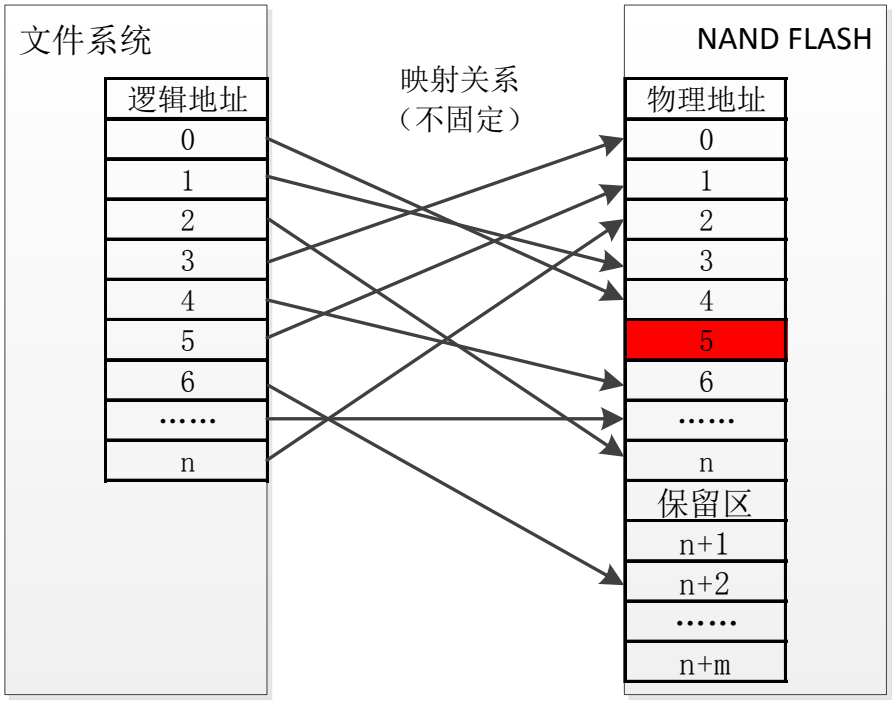


图 46.1.2.1 逻辑-物理地址转换表

图 46.1.2.1 表示某个时刻，逻辑地址与物理地址的对应关系。由图可知，逻辑地址（0~n）到物理地址的映射，映射关系不是一一对应的，而是无序的，这个映射关系是不固定的，随时可能会变化，逻辑地址到物理地址，通过映射表进行映射，所以映射表也是随时需要更新的。

图中，我们假定 NAND 的第 5 个 Block 是坏块，那么映射表一定不能将这个块地址映射给逻辑地址，所以，必须对这个块进行坏块标记，不再作为正常块使用，坏块标记请参考前面的

介绍。另外，当产生了一个坏块的时候，我们必须从保留区（下文介绍）提取一个未用过的块（Block），来替代这个坏块，以确保所有逻辑地址，都有正常的物理地址可用。

逻辑地址到物理地址的映射关系，我们采用一个数组来存储，这个数组即映射表（简称：lut 表），同时，这个映射表必须存储到 NAND FLASH 里面，以便上电后重建。这里，我们也是利用每个 Block 的第一个 page 的 spare 区来存储映射表，另外，还需要标记这个 Block 是否被使用了，所以，Block 第一个 page 的 spare 区规划，如表 46.1.2.2 所示：

BLOCK 编号	PAGE 编号	数据区		Spare 区					
		0	…… 2047	0	1	2	3	……	63
n	0		……	0XFF	0XCC	0X01	0X00		
	1		……	0XFF					
	……	……							
	63		……						

表 46.1.2.2 每个 Block 第一个 page 的 spare 区前 4 个字节规划表

如表 46.1.2.2 所示，每个 Block 第一个 page 的 spare 区第一个字节用来表示该块是否为坏块（前面已介绍）；第二个字节用来表示该块是否被占用（0XFF，表示未占用；0XCC，表示已被占用）；第三和第四个字节，用来存储该块所映射到的逻辑地址，如果为 0XFFFF，则表示该块还未被分配逻辑地址，其他值，则表示该块对应的逻辑地址，MT29F4G08 有 4096 个 Block，所以这两个字节表示的有效逻辑地址范围，就是 0~4095。因此，我们要想判断一个 Block 是否是一个未被使用的好块，只需读取这个 Block 第一个 page 的 spare 区前四个字节，如果为 0XFFFFFFFF 则说明是一个未被使用的好块。

上电的时候，重建映射表（lut 表）的过程就是读取 NAND FLASH 每个 Block 第一个 page 的 spare 区前 4 个字节，当这个块是好块（第一个字节为 0XFF），且第三和第四字节组成的 u16 类型数据（逻辑地址，记为：LBNnum），小于 NAND FLASH 的总块数，则这个 Block 地址（物理地址，记为：M）就是映射表里面第 LBNnum 个元素所对应的地址，即：lut[LBNnum]=M。

3，保留区

保留区有两个作用：1，产生坏块的时候，用来替代坏块；2，在复写数据的时候，用来替代被复写的块，以提高写入速度，并实现磨损均衡处理；

第一个作用，如图 46.1.2.1 所示，当产生坏块后（第 5 个块），使用一个保留区里面的块（n+2），来替代这个坏块，从而保证每个逻辑地址，都有对应的物理地址（好块地址）。

第二个作用，当文件系统要往某个已经被写过数据的块里面写入新数据的时候，由于 NAND 的特性，必须是要先擦除，才能写入新数据。一般的方法：先将整个块数据读出来，然后改写需要写入新数据的部分，然后擦除这个块，然后重新写入这个块，这个过程非常耗时，且需要很大的内存（MT29F4G08 一个 Block 大小为 128K 字节），所以不太实用。

比较好的办法，是利用 NAND FLASH 的页拷贝功能，它可以将 NAND FLASH 内部某个 Block 的数据，以页为单位，拷贝到另外一个空闲的 Block 里面，而且可以写入新的数据（参见 46.1.1 节的指令介绍），利用这个功能，我们无需读出整个 Block 的数据，只需要在页拷贝过程中，在正确的地址写入我们需要写入的新数据即可。这就要求 NAND FLASH 必须有空闲的块，用作页拷贝的目标地址，保留区里面的块，就可以作为空闲块，给页拷贝使用。而且，为了保证不频繁擦除一个块（提高寿命），我们在保留区里面应预留足够的空闲块，用来均分擦除次数，从而实现简单的磨损均衡处理。

这样，FTL 层的坏块管理和磨损均衡原理，就给大家介绍完了。我们根据这个原理，去设计相应的代码，就可以实现 FTL 层的功能，从而更好的使用 NAND FLASH。

前面提到，我们需要用到 ECC 校验，来确保数据的正确性，一般的软件 ECC 校验都是由 FTL 层实现，我们为了简化代码，并利用 STM32 的硬件 ECC 计算，加快 ECC 计算的速度，我们在 FTL 层并不做 ECC 计算和校验，而是放到 NAND FLASH 的底层驱动去实现。在 46.1.1 节最后，我们给大家介绍了 ECC 原理和纠错方法，每 512 个字节的数据，会生成 3 个字节的 ECC 值，而这个 ECC 值是必须存储在 NAND FLASH 里面，同样我们将 ECC 值存储在每个 page 的 spare 区，而且，为了方便读写，我们用 4 个字节来存储这 3 个字节的 ECC 值，ECC 存储关系如表 46.1.2.3 所示：

BLOCK 编号	PAGE 编号	数据区			Spare 区							
		0	…	2047	0	…	16:19	20:23	24:27	28:31	…	63
n	0		…		0XFF	…	ECC01	ECC02	ECC03	ECC04	…	
	1		…		0XFF	…	ECC11	ECC12	ECC13	ECC14	…	
	2		…			…	ECC21	ECC22	ECC23	ECC24	…	
	…	…										
	63		…			…	ECC631	ECC632	ECC633	ECC634	…	

表 46.1.2.3 每个 page 的 spare 区 ECC 存储关系表

如上表所示，每个 page 的一个数据区有 2048 个字节，而每 512 字节数据生成一个 ECC 值，用 4 个字节存储，这样，每个 page 需要 16 个字节用于存储 ECC 值。表中的 ECCx1~ECCx4,（x=0~63），就是存储的 ECC 值，从每个 page 的 spare 区第 16 个字节（0X10）开始存储 ECC 值，总共占用 16 字节，对应关系为：ECCx1→数据区 0~511 字节；ECCx2→数据区 512~1023 字节；ECCx3→数据区 1024~1535 字节；ECCx4→数据区 1536~2047 字节；

在 page 写入数据的时候，我们将 STM32 硬件计算出的 ECC 值写入 spare 区对应的地址，当读取 page 数据的时候，STM32 硬件 ECC 会计算出一个新的 ECC 值，同时，可以在该 page 的 spare 区读取之前保存的 ECC 值，比较这两个 ECC 值，就可以判断数据是否有误，以及进行数据修复（1bit）。**注意：我们对 ECC 处理是以 512 字节为单位的，如果写入/读取的数据不是 512 的整数倍，则不会进行 ECC 处理。**

46.1.3 FMC NAND FLASH 接口简介

在第十八和十九章，我们对 STM32F767 的 FMC 接口进行了简介，并利用 FMC 接口，实现了对 MCU 屏和 SDRAM 的驱动。本章，我们将介绍如何利用 FMC 接口，驱动 NAND FLASH。STM32F767 FMC 接口的 NAND FLASH/PC 卡控制器，具有如下特点：

- 两个 NAND FLASH 存储区域，可独立配置
- 支持 8 位和 16 位 NAND FLASH
- 支持 16 位 PC 卡兼容设备
- 支持硬件 ECC 计算（汉明码）
- 支持 NAND FLASH 预等待功能

通过 46.1.1 的介绍，我们对 NAND FLASH 已经有了一个比较深入的了解，包括接线、控制命令和读写流程等，接下来，我们介绍一些配置 FMC NAND FLASH 控制器需要用到的几个寄存器。

首先，我们介绍 NAND FLASH 的控制寄存器：FMC_PCR，该寄存器各位描述如图 46.1.3.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												ECCPS			TAR				TCLR				Res.	ECCEN	PWID		PTYP	PBKEN	PWAITEN	Reserved	
												rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw		

图 46.1.3.1 FMC_PCR 寄存器各位描述

该寄存器只有部分位有效，且都需要进行配置：

PWAITEN：该位用于设置等待特性：0，禁止；1，使能。这里我们设置为 0，禁止使用控制器自带的等待特性，因为如果使能的话，将导致 RGB 屏抖动（STM32 硬件 bug）。

PBKEN：该位用于使能存储区域：0，禁止；1，使能。我们要正常使用某个存储区域，必须设置该位为 1，所以，这个位要设置为 1。

PTYP：该位用于设置存储器类型：0，保留；1，NAND FLASH。我们用来驱动 NAND FLASH，所以该位设置为 1。

PWID：这两个位，用于设置数据总线宽度：00，8 位宽度；01，16 位宽度。我们使用的 MT29F4G08 为 8 位宽度，所以这里应该设置为：00。

ECCEN：该位用于使能 STM32 的硬件 ECC 计算逻辑：0，禁止/复位 ECC；1，使能 ECC 计算；每次读写数据前，应该设置该位为 1，在数据读写完毕，读取完 ECC 值之后，设置该位为 0，复位 ECC，以便下一次 ECC 计算。

TCLR：这四个位用于设置 CLE 到 RE 的延迟：0000~1111，表示 1~16 个 HCLK 周期。对应 NAND FLASH 数据手册的 tCLR 时间参数，这里设置的 t_clr=(TCLR+SET+2)*THCLK。TCLR 就是本寄存器的设置，SET 对应 MEMSET 的值（我只用到 MEMSET），THCLK 对应 HCLK 的周期。MT29F4G08 的 tCLR 时间最少为 10ns，以 216M 主频计算，一个 HCLK=4.63ns（下同），我们设置 TCLR=5，则 t_clr 至少为 6 个 HCLK 即 27.8ns。

TAR：这四个位用于设置 ALE 到 RE 的延迟：0000~1111，表示 1~16 个 HCLK 周期。对应 NAND FLASH 数据手册的 tAR 时间参数，这里设置的 t_ar=(TAR+SET+2)*THCLK。TAR 就是本寄存器的设置，SET 对应 MEMSET 的值（我只用到 MEMSET），THCLK 对应 HCLK 的周期。MT29F4G08 的 tAR 时间最少为 10ns，我们设置 TAR=5，则 t_ar 至少为 6 个 HCLK 即 27.8ns。

ECCCPS：这三个位用于设置 ECC 的页大小：000，256 字节；001，512 字节；010，1024 字节；011，2048 字节；100，4096 字节；101，8192 字节。我们需要以 512 字节为单位进行 ECC 计算，所以 ECCCPS 设置为：001 即可。

接下来，我们介绍 NAND FLASH 的空间时序寄存器：FMC_PMEM，该寄存器各位描述如图 46.1.3.2 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
MEMHIZx								MEMHOLDx								MEMWAITx								MEMSETx							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 46.1.3.2 FMC_PMEM 寄存器各位描述

该寄存器用于控制 NAND FLASH 的访问时序，非常重要。我们先来了解下 NAND FLASH 控制器的通用存储器访问波形，如图 46.1.3.3 所示：

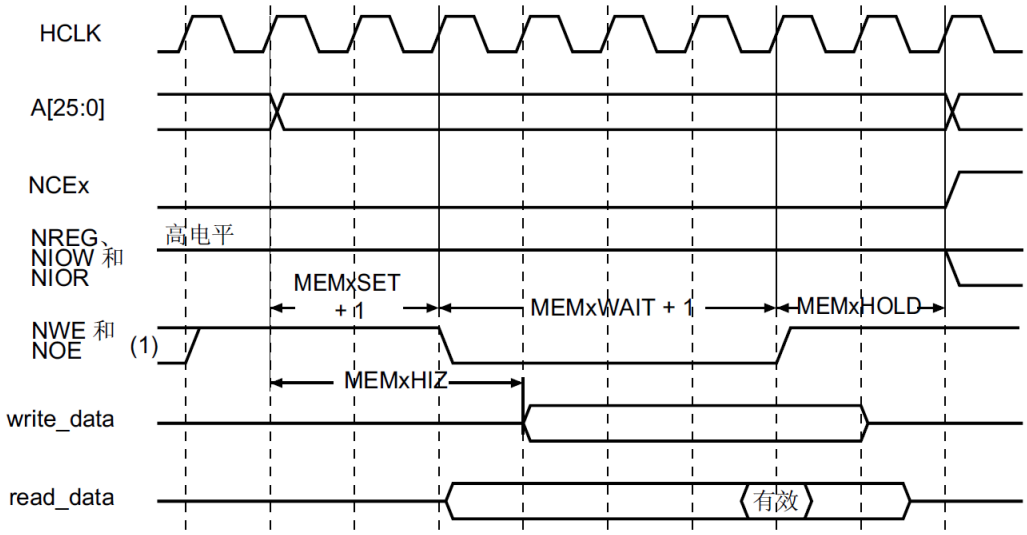


图 46.1.3.3 NAND FLASH 通用存储器访问波形

由图可知，MEMxSET+MEMxHOLD 控制 NWE/NOE 的高电平时间，MEMxWAIT 控制 NWE/NOE 的低电平时间，MEMxHIZ 控制写入时数据线高阻态时间。接下来我们分别介绍这几个参数：

MEMSETx：这八个位定义使能命令（NWE/NOE）前，地址建立所需要的 HCLK 时钟周期数，表示 NWE/NOE 的高电平时间，0000 0000~1111 1111 表示 1~256 个 HCLK 周期。MT29F4G08 的 tREH/tWH 最少为 10ns，我们设置 MEMSETx=1，即 2 个 HCLK 周期，约 9.3ns。另外，MEMHOLDx，也可以用于控制 NWE/NOE 的高电平时间，在连续访问的时候，MEMHOLDx 和 MEMSETx 共同构成 NWE/NOE 的高电平脉宽时间。

MEMWAITx：这八个位用于设置使能命令（NWE/NOE）所需的最小 HCLK 时钟周期数（使能 NWAIT 将使这个时间延长），实际上就是 NWE/NOE 的低电平时间，0000 0000~1111 1111 表示 1~256 个 HCLK 周期。MT29F4G08 的 tRP/tWP 最少为 10ns，我们设置 MEMWAITx =3，即 4 个 HCLK 周期，约 18.5ns。这里需要设置时间比较长一点，否则可能访问不正常。

MEMHOLDx：这八个位用于设置禁止使能命令（NWE/NOE）后，保持地址（和写访问数据）的 HCLK 时钟周期数，也可以用于设置一个读写周期内的 NWE/NOE 高电平时间，0000 0000~1111 1111 表示 0~255 个 HCLK 周期。我们设置 MEMHOLDx=1，表示 1 个 HCLK 周期，加上前面的 MEMSETx，所以 NEW/NOE 高电平时间为 3 个 HCLK，即 13.9ns 左右。

MEMHIZx：这八个位定义通用存储空间开始执行写访问之后，数据总线保持高阻态所持续的 HCLK 时钟周期数。该参数仅对写入事务有效，0000~1111 1111 表示 1~256 个 HCLK 周期。我们设置 MEMHIZx=1，表示 2 个 HCLK 周期，即 9.3ns 左右。

接下来，我们介绍 NAND FLASH 的 ECC 结果寄存器：FMC_ECCR，该寄存器各位描述如图 46.1.3.4 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
ECCx																															
r																															

图 46.1.3.4 FMC_ ECCR 寄存器各位描述

该寄存器包含由 ECC 计算逻辑计算所得的结果。根据 ECCPS 位（在 FMC_PCRx 寄存器）的设置，ECCx 的有效位数也有差异，如表 46.1.3.1 所示：

ECCPS[2:0]	以字节为单位的页大小	ECC 位
000	256	ECC[21:0]
001	512	ECC[23:0]
010	1024	ECC[25:0]
011	2048	ECC[27:0]
100	4096	ECC[29:0]
101	8192	ECC[31:0]

表 46.1.3.1 ECC 结果相关位

我们以 512 字节为页大小（ECCPS=001），所以 ECCx 的低 24 位有效，用于存储计算所得的 ECC 值。

接下来，我们介绍 NAND FLASH 的状态和中断寄存器：FMC_SR，该寄存器各位描述如图 46.1.3.5 所示：

31 … 7	6	5	4	3	2	1	0
Reserved	FEMPT	IFEN	ILEN	IREN	IFS	ILS	IRS
	r	rw	rw	rw	rw	rw	rw

图 46.1.3.5 FMC_SR 寄存器各位描述

该寄存器我们只关心第 6 位：FEMPT 位，该位用于表示 FIFO 的状态。当 FEMPT=0 时，表示 FIFO 非空，表示还有数据在传输；当 FEMPT=1 时，表示 FIFO 为空，表示数据传输完成。在计算 ECC 的时候，我们必须等待 FEMPT=1，再去读取 ECCR 寄存器的值，确保数据全部传输完毕。

至此，FMC NAND FLASH 部分的寄存器就介绍完了，关于 FMC NAND FLASH 控制器的详细介绍，请大家参考《STM32F7 中文参考手册》第 13.6 节。通过以上三个小节的了解，我们就可以开始编写 NAND FLASH 的驱动代码了。

阿波罗 STM32F767 核心板板载的 MT29F4G08 芯片挂在 FMC NAND FLASH 的控制器上面(NCE3)，其原理图如图 46.1.3.6 所示：

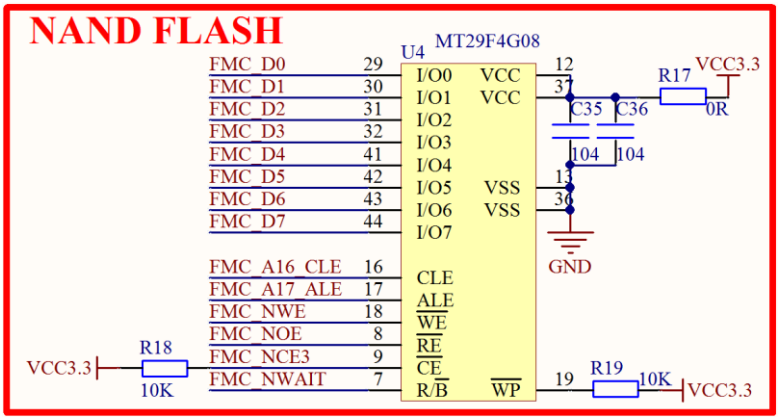


图 46.1.3.6 MT29F4G08 原理图

从原理图可以看出，MT29F4G08 同 STM32F767 的连接关系：

I/O[0:7]接 FMC_D[0:7]

CLE 接 FMC_A16_CLE

ALE 接 FMC_A17_ALE
WE 接 FMC_NWE
RE 接 FMC_NOE
CE 接 FMC_NCE3
R/B 接 FMC_NWAIT

最后，我们来看看实现对 MT29F4G08 的驱动，需要对 FMC 进行哪些配置。这里我们需要引入 stm32f7xx_II_fmc.c/stm32f7xx_hal_nand.c 源文件以及对应的头文件。具体步骤如下：

1）使能 FMC 时钟，并配置 FMC 相关的 IO 及其时钟使能。

要使用 FMC，当然首先得开启其时钟。然后需要把 FMC_D0~7、FMC_A16_CLE 和 FMC_A17_ALE 等相关 IO 口，全部配置为复用输出，并使能各 IO 组的时钟。使能 FMC 时钟和 IO 口初始化方法前面多次讲解，这里就不累赘了。

2）初始化 NAND，设置控制参数（设置 FMC_PCR3）和时间参数（设置 FMC_PMEM3）。

该步骤通过设置寄存器 FMC_PCR3（因为使用的是 FMC_NAND_BANK3 所以对应寄存器 FMC_PCR3）来设置 NAND FLASH 的相关控制参数，比如数据宽度、CLR/AR 延迟、ECC 页大小等，通过设置寄存器 FMC_PMEM3（因为使用的是 FMC_NAND_BANK3，所以对应寄存器 FMC_PMEM3）来设置 NAND 的相关时间参数，控制 FMC 访问 NAND FLASH 的时序。HAL 库提供了 NAND FLASH 初始化函数 HAL_NAND_Init：

```
HAL_StatusTypeDef    HAL_NAND_Init(NAND_HandleTypeDef *hnand,
                                   FMC_NAND_PCC_TimingTypeDef *ComSpace_Timing,
                                   FMC_NAND_PCC_TimingTypeDef *AttSpace_Timing);
```

该函数有三个入口参数，第一个入口参数 hnand 用来设置 NAND FLASH 的控制参数，第二个入口参数 ComSpace_Timing 用来设置 NAND 通用存储器空间时序，第三个入口参数 AttSpace_Timing 用来设置 NAND 特性存储器空间时序。这里我们着重讲解第一个入口参数 hnand 的定义，该参数为 NAND_HandleTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
  FMC_NAND_TypeDef                  *Instance;
  FMC_NAND_InitTypeDef               Init;
  HAL_LockTypeDef                   Lock;
  __IO HAL_NAND_StateTypeDef        State;
  NAND_InfoTypeDef                  Info;
}NAND_HandleTypeDef;
```

这里我们主要关注成员变量 Init 的含义，该成员变量是 FMC_NAND_InitTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
  uint32_t NandBank;              //BANK 编号
  uint32_t Waitfeature;           //等待 特性使能/失能
  uint32_t MemoryDataWidth;       //数据总线宽度：8 位/16 位
  uint32_t EccComputation;        //ECC 计算逻辑使能/失能
  uint32_t ECCPageSize;           //ECC 页大小：256/512/1024/2048/4096/8192 字节
  uint32_t TCLRSetupTime;         //CLE 到 RE 的延迟
  uint32_t TARSetupTime;          //ALE 到 RE 的延迟
```

```
}FMC_NAND_InitTypeDef;
```

这些成员变量设置值对应的是 FMC_ PCR 寄存器相应位，这些成员变量的含义我们都已经注释了，如有不理解的地方请参考前面 FMC_ PCR 寄存器讲解。

对于 FMC 的时序参数，这里我们就不做 过都讲解，请参考前面讲解。

HAL 库同样为 NAND 初始化提供了 MSP 回调函数 HAL_NAND_MspInit：

```
void HAL_NAND_MspInit(NAND_HandleTypeDef *hnand);
```

该函数内部一般用来使能时钟，初始化 IO 口。

3）配置 FMC_PCR3 寄存器，使能存储区域 3。

在 FMC 的配置完成后，最后，设置 FMC_PCR3 寄存器的 PBKEN 位（bit2）为 1，使能存储区域 3。 如果使用 HAL 库，那么在函数 HAL_NAND_Init 的尾部有使能存储区域的操作，我们就不需要重复进行此步骤。操作方法为：

```
__FMC_NAND_ENABLE(hnand->Instance, hnand->Init.NandBank);
```

通过以上几个步骤，我们就完成了 FMC 的配置，可以访问 MT29F4G08 了，最后，因为我们使用的是 FMC 的 BANK3，所以 MT29F4G08 的访问地址为 0X80000000，而 NAND FLASH 的命令/地址控制由 CLE/ALE 控制，也就是由 FMC_A17_CLE 和 FMC_A16_ALE 控制，因此，发送命令和地址的语句为：

```
*(vu8*)(0X80000000|(1<<17))=CMD;
*(vu8*)(0X80000000|(1<<16))=ADDR;
```

46.2 硬件设计

本章实验功能简介：开机后，先检测 NAND FLASH 并初始化 FTL，如果初始化成功，则显示提示信息，然后按下 KEY0 按键，可以通过 FTL 读取扇区 2 的数据；按 KEY1 按键，可以通过 FTL 写入扇区 2 的数据；按 KEY2 按键，则可以恢复扇区 2 的数据（防止损坏文件系统）；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 和 KEY2 按键

3） 串口

4） LCD 模块

5） MT29F4G08

这些我们都已经介绍过（MT29F4G08 与 STM32F767 的各 IO 对应关系，请参考光盘原理图），接下来我们开始软件设计。

46.3 软件设计

打开本章实验工程可以看到，我们在 HARDWARE 分组之下添加了 nand.c，ftl.c 以及 nandtester.c 三个源文件，同时包含了对应的头文件。

由于代码量比较多，我们这里就不将所有代码贴出来了，仅挑一些重点的函数，给大家介绍，详细的代码请大家打开本例程源码查看。

这里我们需要说明一下，由于 ST 官方 HAL 库提供的 NAND 相关驱动函数我们在使用过程发现了很多兼容性问题，并且没有提供坏块管理操作，使用起来并不是非常方便。所以我们 ALIENTEK 团队重写了一套 NAND 操作函数供大家参考。

在 nand.c 里面，我们只介绍：NAND_Init、HAL_NAND_MspInit、NAND_ReadPage 和 NAND_WritePage 等三四个函数。NAND_Init 函数代码如下：

```
//初始化 NAND FLASH
u8 NAND_Init(void)
{
    FMC_NAND_PCC_TimingTypeDef ComSpaceTiming,AttSpaceTiming;

    NAND_MPU_Config();
    NAND_Handler.Instance=FMC_Bank3;
    NAND_Handler.Init.NandBank=FMC_NAND_BANK3;
                                        //NAND 挂在 BANK3 上
    NAND_Handler.Init.Waitfeature=FMC_NAND_PCC_WAIT_FEATURE_DISABLE;
                                        //关闭等待特性
    NAND_Handler.Init.MemoryDataWidth=FMC_NAND_PCC_MEM_BUS_WIDTH_8;
                                        //8 位数据宽度
    NAND_Handler.Init.EccComputation=FMC_NAND_ECC_DISABLE;
                                        //禁止 ECC
    NAND_Handler.Init.ECCPageSize=FMC_NAND_ECC_PAGE_SIZE_512BYTE;
                                        //ECC 页大小为 512 字节
    NAND_Handler.Init.TCLRSetupTime=10; //设置 TCLR(tCLR=CLE 到 RE 的延时)=
                                    //(TCLR+TSET+2)*THCLK,THCLK=1/180M=4.6ns
    NAND_Handler.Init.TARSetupTime=10;
    //设置 TAR(tAR=ALE 到 RE 的延时)=(TAR+TSET+1)*THCLK,THCLK=1/180M=4.6n。

    ComSpaceTiming.SetupTime=10;        //建立时间
    ComSpaceTiming.WaitSetupTime=10;    //等待时间
    ComSpaceTiming.HoldSetupTime=10;    //保持时间
    ComSpaceTiming.HiZSetupTime=10;     //高阻态时间

    AttSpaceTiming.SetupTime=10;        //建立时间
    AttSpaceTiming.WaitSetupTime=10;    //等待时间
    AttSpaceTiming.HoldSetupTime=10;    //保持时间
    AttSpaceTiming.HiZSetupTime=10;     //高阻态时间

    HAL_NAND_Init(&NAND_Handler,&ComSpaceTiming,&AttSpaceTiming);
    NAND_Reset();                       //复位 NAND
    delay_ms(100);
    nand_dev.id=NAND_ReadID();          //读取 ID
    printf("NAND ID:%#x\r\n",nand_dev.id);
    NAND_ModeSet(4);                    //设置为 MODE4,高速模式
    if(nand_dev.id==MT29F16G08ABABA)    //NAND 为 MT29F16G08ABABA
    {
        nand_dev.page_totalsize=4320;
        nand_dev.page_mainsize=4096;
        nand_dev.page_sparesize=224;
```

```
            nand_dev.block_pagenum=128;
            nand_dev.plane_blocknum=2048;
            nand_dev.block_totalnum=4096;
        }
        else if(nand_dev.id==MT29F4G08ABADA)//NAND 为 MT29F4G08ABADA
        {
            nand_dev.page_totalsize=2112;
            nand_dev.page_mainsize=2048;
            nand_dev.page_sparesize=64;
            nand_dev.block_pagenum=64;
            nand_dev.plane_blocknum=2048;
            nand_dev.block_totalnum=4096;
        }else return 1; //错误，返回
        return 0;
    }
```

该函数用于初始化 NAND FLASH，主要是调用函数 HAL_NAND_Init 函数初始化 NAND，配置相关控制参数和 FMC 时序，另外，该函数会读取 NAND ID，从而判断 NAND FLASH 的型号，执行不同的参数初始化。nand_dev 是我们在 nand.h 里面定义的一个 NAND 属性结构体，存储 NAND FLASH 的一些特性参数，方便驱动。

函数 HAL_NAND_MspInit 内容这里我们就不列出来了，该函数是 NAND 的 MSP 初始化回调函数，用来初始化与 MCU 相关的步骤，包括时钟使能和 IO 初始化。

接下来，我们看 NAND_ReadPage 函数的代码，如下：

```
//读取 NAND Flash 的指定页指定列的数据(main 区和 spare 区都可以使用此函数)
//PageNum:要读取的页地址,范围:0~(block_pagenum*block_totalnum-1)
//ColNum:要读取的列开始地址(也就是页内地址),范围:0~(page_totalsize-1)
//*pBuffer:指向数据存储区
//NumByteToRead:读取字节数(不能跨页读)
//返回值:0,成功
//       其他,错误代码
u8 NAND_ReadPage(u32 PageNum,u16 ColNum,u8 *pBuffer,u16 NumByteToRead)
{
    vu16 i=0; u8 res=0;
    u8 eccnum=0;        //需要计算的 ECC 个数，每 NAND_ECC_SECTOR_SIZE 一个 ecc
    u8 eccstart=0;      //第一个 ECC 值所属的地址范围
    u8 errsta=0; u8 *p;
     *(vu8*)(NAND_ADDRESS|NAND_CMD)=NAND_AREA_A;     //发送命令
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)ColNum;          //发送地址
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(ColNum>>8);
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)PageNum;
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(PageNum>>8);
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(PageNum>>16);
    *(vu8*)(NAND_ADDRESS|NAND_CMD)=NAND_AREA_TRUE1;
    //下面两行代码是等待 R/B 引脚变为低电平，其实主要起延时作用的，等待 NAND
```

```
//操作 R/B 引脚。因为我们是通过将 STM32 的 NWAIT 引脚(NAND 的 R/B 引脚)配置
//为普通 IO，代码中通过读取 NWAIT 引脚的电平来判断 NAND 是否准备就绪.
res=NAND_WaitRB(0);             //先等待 RB=0
if(res)return NSTA_TIMEOUT;     //超时退出
//下面 2 行代码是真正判断 NAND 是否准备好的
res=NAND_WaitRB(1);             //等待 RB=1
if(res)return NSTA_TIMEOUT;     //超时退出
if(NumByteToRead%NAND_ECC_SECTOR_SIZE)
//不是 NAND_ECC_SECTOR_SIZE 的整数倍，不进行 ECC 校验
{
    //读取 NAND FLASH 中的数据
    for(i=0;i<NumByteToRead;i++)*(vu8*)pBuffer++ = *(vu8*)NAND_ADDRESS;
}else
{
    eccnum=NumByteToRead/NAND_ECC_SECTOR_SIZE;  //得到 ecc 计算次数
    eccstart=ColNum/NAND_ECC_SECTOR_SIZE;       //从第几个 ECC 开始
    p=pBuffer;
    for(res=0;res<eccnum;res++)
    {
        FMC_Bank2_3->PCR3|=1<<6;                //使能 ECC 校验
        for(i=0;i<NAND_ECC_SECTOR_SIZE;i++)     //读取数据
        {
            *(vu8*)pBuffer++ = *(vu8*)NAND_ADDRESS;
        }
        while(!(FMC_Bank2_3->SR3&(1<<6)));      //等待 FIFO 空
        nand_dev.ecc_hdbuf[res+eccstart]=FMC_Bank2_3->ECCR3;//读取 ECC 值
        FMC_Bank2_3->PCR3&=~(1<<6);             //复位 ECC
    }
    i=nand_dev.page_mainsize+0X10+eccstart*4;//读取 spare 区，之前存储的 ecc 值
    NAND_Delay(30);                             //等待 tADL
    *(vu8*)(NAND_ADDRESS|NAND_CMD)=0X05;        //随机读指令
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)i;      //发送地址
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(i>>8);
    *(vu8*)(NAND_ADDRESS|NAND_CMD)=0XE0;        //开始读数据
    NAND_Delay(30);                             //等待 tADL
    pBuffer=(u8*)&nand_dev.ecc_rdbuf[eccstart];
    for(i=0;i<4*eccnum;i++)                     //读取保存的 ECC 值
    {
        *(vu8*)pBuffer++= *(vu8*)NAND_ADDRESS;
    }
    for(i=0;i<eccnum;i++)                       //检验 ECC
    {
        if(nand_dev.ecc_rdbuf[i+eccstart]!=nand_dev.ecc_hdbuf[i+eccstart])//不相等
```

```
                {
                    //进行 ECC 校验，并纠正 1bit ECC 错误
                    res=NAND_ECC_Correction(p+NAND_ECC_SECTOR_SIZE*i,
                        nand_dev.ecc_rdbuf[i+eccstart],nand_dev.ecc_hdbuf[i+eccstart]);
                    if(res)errsta=NSTA_ECC2BITERR;    //标记 2BIT 及以上 ECC 错误
                    else errsta=NSTA_ECC1BITERR;      //标记 1BIT ECC 错误
                }
            }
        }
        if(NAND_WaitForReady()!=NSTA_READY)errsta=NSTA_ERROR;//失败
        return errsta;  //成功
    }
```

该函数用于读取 NAND 里面的数据，通过指定页地址（PageNum）和列地址（ColNum），就可以读取 NAND FLASH 里面任何地址的数据，不过该函数读数据时不能跨页读，所以一次最多读取一个 Page 的数据(包括 spare 区数据)。当读取数据长度为 NAND_ECC_SECTOR_SIZE（512 字节）的整数倍时，将执行 ECC 校验，ECC 校验完全是按照 46.1 节介绍的方法来实现，当出现 ECC 错误时，调用 NAND_ECC_Correction 函数（46.1.1 节已经介绍）进行 ECC 纠错，可以实现 1bit 错误纠正，并报告 2bit 及以上的错误。

接下来，我们看 NAND_WritePage 函数的代码，如下：

```
u8 NAND_WritePage(u32 PageNum,u16 ColNum,u8 *pBuffer,u16 NumByteToWrite)
{
    vu16 i=0;
    u8 res=0;
    u8 eccnum=0; //需要计算的 ECC 个数，
                 //每 NAND_ECC_SECTOR_SIZE 字节计算一个 ecc
    u8 eccstart=0;      //第一个 ECC 值所属的地址范围

    *(vu8*)(NAND_ADDRESS|NAND_CMD)=NAND_WRITE0;
    //发送地址
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)ColNum;
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(ColNum>>8);
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)PageNum;
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(PageNum>>8);
    *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(PageNum>>16);
    NAND_Delay(30);//等待 tADL
    if(NumByteToWrite%NAND_ECC_SECTOR_SIZE)
                //不是 NAND_ECC_SECTOR_SIZE 的整数倍，不进行 ECC 校验
    {
        for(i=0;i<NumByteToWrite;i++)        //写入数据
        {
            *(vu8*)NAND_ADDRESS=*(vu8*)pBuffer++;
        }
    }else
```

```
    {
        eccnum=NumByteToWrite/NAND_ECC_SECTOR_SIZE; //得到 ecc 计算次数
        eccstart=ColNum/NAND_ECC_SECTOR_SIZE;
        for(res=0;res<eccnum;res++)
        {
            FMC_Bank3->PCR|=1<<6;                          //使能 ECC 校验
            for(i=0;i<NAND_ECC_SECTOR_SIZE;i++)
                                    //写入 NAND_ECC_SECTOR_SIZE 个数据
            {
                *(vu8*)NAND_ADDRESS=*(vu8*)pBuffer++;
            }
            while(!(FMC_Bank3->SR&(1<<6)));                //等待 FIFO 空
            nand_dev.ecc_hdbuf[res+eccstart]=FMC_Bank3->ECCR;
                                                    //读取硬件计算后的 ECC 值
            FMC_Bank3->PCR&=~(1<<6);                   //禁止 ECC 校验
        }
        i=nand_dev.page_mainsize+0X10+eccstart*4;    //计算写入 ECC 的 spare 区地址
        NAND_Delay(30);//等待
        *(vu8*)(NAND_ADDRESS|NAND_CMD)=0X85;           //随机写指令
        //发送地址
        *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)i;
        *(vu8*)(NAND_ADDRESS|NAND_ADDR)=(u8)(i>>8);
        NAND_Delay(30);//等待 tADL
        pBuffer=(u8*)&nand_dev.ecc_hdbuf[eccstart];
        for(i=0;i<eccnum;i++)                          //写入 ECC
        {
            for(res=0;res<4;res++)
            {
                *(vu8*)NAND_ADDRESS=*(vu8*)pBuffer++;
            }
        }
    }
    *(vu8*)(NAND_ADDRESS|NAND_CMD)=NAND_WRITE_TURE1;
    if(NAND_WaitForReady()!=NSTA_READY)return NSTA_ERROR;//失败
    return 0;//成功
}
```

该函数用于往 NAND 里面写数据，通过指定页地址（PageNum）和列地址（ColNum），就可以往 NAND FLASH 里面任何地址写数据（包括 spare 区），同样，该函数也不支持跨页写。当读取数据长度为 NAND_ECC_SECTOR_SIZE（512 字节）的整数倍时，将执行 ECC 校验，并将 ECC 值写入 spare 区对应的地址，以便读取数据时，进行 ECC 校验。

nand.c 里面的其他代码以及 nand.h 里面的代码，请大家参考本例程源码。接下来，我看 ftl.c 里面的代码，该文件，我们只介绍：FTL_Init、FTL_Format、FTL_CreateLUT、FTL_LBNToPBN、FTL_WriteSectors 和 FTL_ReadSectors 等七个函数。

首先，FTL_Init 函数代码如下：

```
//FTL 层初始化
//返回值:0,正常     其他,失败
u8 FTL_Init(void)
{
    u8 temp;
    if(NAND_Init())return 1;                                    //初始化 NAND FLASH
    if(nand_dev.lut)myfree(SRAMIN,nand_dev.lut);
    nand_dev.lut=mymalloc(SRAMIN,(nand_dev.block_totalnum)*2);    //给 LUT 表申请内存
    memset(nand_dev.lut,0,nand_dev.block_totalnum*2);             //全部清理
    if(!nand_dev.lut)return 1;                                  //内存申请失败
    temp=FTL_CreateLUT(1);
    if(temp)
    {
        printf("format nand flash...\r\n");
        temp=FTL_Format();        //格式化 NAND
        if(temp)
        {
            printf("format failed!\r\n");
            return 2;
        }
    }else      //创建 LUT 表成功
    {
        printf("total block num:%d\r\n",nand_dev.block_totalnum);
        printf("good block num:%d\r\n",nand_dev.good_blocknum);
        printf("valid block num:%d\r\n",nand_dev.valid_blocknum);
    }
    return 0;
}
```

该函数用于初始化 FTL，包括：初始化 NAND FLASH、为 lut 表申请内存、创建 lut 表等操作，如果创建 lut 表失败，则会通过 FTL_Format 函数格式化 NAND FLASH。

FTL_Format 函数代码如下：

```
u8 FTL_Format(void)
{
    u8 temp;
    u32 i,n;
    u32 goodblock=0;
    nand_dev.good_blocknum=0;
#if FTL_USE_BAD_BLOCK_SEARCH==1            //使用擦-写-读的方式,检测坏块
    nand_dev.good_blocknum=FTL_SearchBadBlock();//搜寻坏块.耗时很久
#else                   //直接使用 NAND FLASH 的出厂坏块标志(其他块,默认是好块)
    for(i=0;i<nand_dev.block_totalnum;i++)
    {
```

```
            temp=FTL_CheckBadBlock(i);              //检查一个块是否为坏块
            if(temp==0)                             //好块
            {
                temp=NAND_EraseBlock(i);
                if(temp)                            //擦除失败,认为坏块
                {
                    printf("Bad block:%d\r\n",i);
                    FTL_BadBlockMark(i);        //标记是坏块
                }else nand_dev.good_blocknum++;     //好块数量加一
            }
        }
    #endif
        printf("good_blocknum:%d\r\n",nand_dev.good_blocknum);
        if(nand_dev.good_blocknum<100) return 1; //如果好块数量少于 100 则 NAND Flash 报废
        goodblock=(nand_dev.good_blocknum*93)/100; //%93 的好块用于存储数据
        n=0;
        for(i=0;i<nand_dev.block_totalnum;i++)          //在好块中标记上逻辑块信息
        {
            temp=FTL_CheckBadBlock(i);              //检查一个块是否为坏块
            if(temp==0)                             //好块
            {
                NAND_WriteSpare(i*nand_dev.block_pagenum,2,(u8*)&n,2);//写入逻辑块编号
                n++;                                //逻辑块编号加 1
                if(n==goodblock) break;             //全部标记完了
            }
        }
        if(FTL_CreateLUT(1))return 2;               //重建 LUT 表失败
        return 0;
    }
```

该函数用于格式化 NAND FLASH，执行的操作包括：1，检测/搜索整个 NAND 的坏块，并做标记；2，分割所有好块，93%用作物理地址（并进行逻辑编号），7%用作保留区；3，重新创建 lut 表。此函数，将我们在 46.1.2 节介绍的 FTL 层坏块管理的几个要点（识别坏块并标记、生成转换表、生成保留区），都实现了，从而完成对 NAND FLASH 的格式化（不是文件系统那种格式化，这里的格式化是指针对 FTL 层的初始化设置）。

接下来，我们看 FTL_CreateLUT 函数，该函数代码如下：

```
//重新创建 LUT 表
//mode:0,仅检查第一个坏块标记
//      1,两个坏块标记都要检查(备份区也要检查)
//返回值:0,成功
//      其他,失败
u8 FTL_CreateLUT(u8 mode)
{
    u32 i;
```

```
    u8 buf[4];
    u32 LBNnum=0;                                           //逻辑块号
    for(i=0;i<nand_dev.block_totalnum;i++)//复位 LUT 表，初始化为无效值，也就是 0XFFFF
    {
        nand_dev.lut[i]=0XFFFF;
    }
    nand_dev.good_blocknum=0;
    for(i=0;i<nand_dev.block_totalnum;i++)
    {
        NAND_ReadSpare(i*nand_dev.block_pagenum,0,buf,4);   //读取 4 个字节

if(buf[0]==0XFF&&mode)NAND_ReadSpare(i*nand_dev.block_pagenum+1,0,buf,1);
                                                            //好块,且需要检查 2 次坏块标记
        if(buf[0]==0XFF)//是好块
        {
            LBNnum=((u16)buf[3]<<8)+buf[2];             //得到逻辑块编号
            if(LBNnum<nand_dev.block_totalnum)//逻辑块号肯定小于总的块数量
            {
                nand_dev.lut[LBNnum]=i;//更新 LUT 表， 写 LBNnum 对应的物理块编号
            }
            nand_dev.good_blocknum++;
        }else printf("bad block index:%d\r\n",i);
    }
    //LUT 表建立完成以后检查有效块个数
    for(i=0;i<nand_dev.block_totalnum;i++)
    {
        if(nand_dev.lut[i]>=nand_dev.block_totalnum)
        {
            nand_dev.valid_blocknum=i;
            break;
        }
    }
    if(nand_dev.valid_blocknum<100)return 2;  //有效块数小于 100,有问题.需要重新格式化
    return 0;  //LUT 表创建完成
}
```

该函数用于重建 lut 表，读取保存在每个 Block 第一个 page 的 spare 区的逻辑编号，存储在 nand_dev.lut 表里，并初始化有效块（nand_dev.valid_blocknum）和好块（nand_dev.good_blocknum）的数量，完成转换表（lut 表）的创建。

接下来，我们看 FTL_LBNToPBN 函数，该函数代码如下：

```
//逻辑块号转换为物理块号
//LBNNum:逻辑块编号
//返回值:物理块编号
u16 FTL_LBNToPBN(u32 LBNNum)
```

```
{
    u16 PBNNo=0;
    //当逻辑块号大于有效块数的时候返回 0XFFFF
    if(LBNNum>nand_dev.valid_blocknum)return 0XFFFF;
    PBNNo=nand_dev.lut[LBNNum];
    return PBNNo;
}
```

该函数用于将逻辑块地址改为物理块地址，输入参数：LBNNum，表示逻辑块编号，返回值表示 LBNNum 对应的物理块地址。有了该函数，就可以很方便的实现逻辑块地址到物理块地址的映射。

接下来，我们看 FTL_WriteSectors 函数，该函数代码如下：

```
//写扇区(支持多扇区写)，FATFS 文件系统使用
//pBuffer:要写入的数据
//SectorNo:起始扇区号
//SectorSize:扇区大小(不能大于 NAND_ECC_SECTOR_SIZE 定义的大小,否则会出错!!)
//SectorCount:要写入的扇区数量
//返回值:0,成功
//       其他,失败
u8 FTL_WriteSectors(u8 *pBuffer,u32 SectorNo,u16 SectorSize,u32 SectorCount)
{
    u8 flag=0; u16 temp; u32 i=0;
    u16 wsecs;          //写页大小
    u32 wlen;           //写入长度
    u32 LBNNo;          //逻辑块号
    u32 PBNNo;          //物理块号
    u32 PhyPageNo;      //物理页号
    u32 PageOffset;     //页内偏移地址
    u32 BlockOffset;    //块内偏移地址
    u32 markdpbn=0XFFFFFFFF;          //标记了的物理块编号
    for(i=0;i<SectorCount;i++)
    {
        LBNNo=(SectorNo+i)/(nand_dev.block_pagenum*(nand_dev.page_mainsize
                /SectorSize));          //根据逻辑扇区号和扇区大小计算出逻辑块号
        PBNNo=FTL_LBNToPBN(LBNNo);              //将逻辑块转换为物理块
        if(PBNNo>=nand_dev.block_totalnum)return 1;  //物理块号大于总块数,则失败.
        BlockOffset=((SectorNo+i)%(nand_dev.block_pagenum*
                (nand_dev.page_mainsize/SectorSize)))*SectorSize;//计算块内偏移
        PhyPageNo=PBNNo*nand_dev.block_pagenum+BlockOffset/
                nand_dev.page_mainsize;         //计算出物理页号
        PageOffset=BlockOffset%nand_dev.page_mainsize;//计算出页内偏移地址
        temp=nand_dev.page_mainsize-PageOffset;     //page 内剩余字节数
        temp/=SectorSize;                   //可以连续写入的 sector 数
        wsecs=SectorCount-i;                //还剩多少个 sector 要写
```

```
            if(wsecs>=temp)wsecs=temp;      //大于可连续写入的 sector 数,则写入 temp 个扇区
            wlen=wsecs*SectorSize;          //每次写 wsecs 个 sector
            //读出写入大小的内容判断是否全为 0XFFFFFFFF（以 4 字节为单位读取）
            flag=NAND_ReadPageComp(PhyPageNo,PageOffset,0XFFFFFFFF,wlen/4,&temp);
            if(flag)return 2;               //读写错误，坏块
            //全为 0XFF,可以直接写数据
            if(temp==(wlen/4))flag=NAND_WritePage(PhyPageNo,PageOffset,pBuffer,wlen);
            else flag=1;                    //不全是 0XFF,则另作处理
            if(flag==0&&(markdpbn!=PBNNo))      //标记了的物理块与当前物理块不同
            {
                flag=FTL_UsedBlockMark(PBNNo);  //标记此块已经使用
                markdpbn=PBNNo;             //标记完成,标记块=当前块,防止重复标记
            }
            if(flag)//不全为 0XFF/标记失败，将数据写到另一个块
            {
                temp=((u32)nand_dev.block_pagenum*nand_dev.page_mainsize-BlockOffset)
                    /SectorSize;    //计算整个 block 还剩下多少个 SECTOR 可以写入
                wsecs=SectorCount-i;    //还剩多少个 sector 要写
                if(wsecs>=temp)wsecs=temp;//大于可连续写入的 sector 数,则写入 temp 个扇区
                wlen=wsecs*SectorSize;//每次写 wsecs 个 sector
                //拷贝到另外一个 block，并写入数据
                flag=FTL_CopyAndWriteToBlock(PhyPageNo,PageOffset,pBuffer,wlen);
                if(flag)return 3;//失败
            }
            i+=wsecs-1;
            pBuffer+=wlen;//数据缓冲区指针偏移
        }
        return 0;
    }
```

该函数非常重要，它是 FTL 层对文件系统的接口函数，用于往 NAND FLASH 里面写入数据，用户调用该函数时，无需关心坏块和磨损均衡问题，完全可以把 NAND FLASH 当成一个 SD 卡来访问。该函数输入参数：SectorNo 用于指定扇区地址，扇区大小由 SectorSize 指定，一般我们设置 SectorSize=NAND_ECC_SECTOR_SIZE，方便进行 ECC 校验处理。

该函数根据 SectorNo 和 SectorSize，首先计算出逻辑块地址（LBNNo），然后将逻辑块地址转换为物理块地址（PBNNo），然后再计算出块内的页地址（PhyPageNo）和页内的偏移地址（PageOffset），然后计算该页内还可以连续写入的扇区数（如果可以连续写，则可以提高速度），然后判断要写入的区域，数据是否全为 0XFF，如果全是 0XFF，则直接写入，写入完成对该物理块进行已被使用标记。如果不是全 0XFF，则需要利用 NAND 页拷贝功能，将本页数据拷贝到另外一个 Block，并写入要写入的数据，这个操作由 FTL_CopyAndWriteToBlock 函数来完成。

最后，我们看 FTL_ReadSectors 函数，该函数代码如下：

```
//读扇区(支持多扇区读)，FATFS 文件系统使用
//pBuffer:数据缓存区
//SectorNo:起始扇区号
```

```
//SectorSize:扇区大小
//SectorCount:要写入的扇区数量
//返回值:0,成功
//       其他,失败
u8 FTL_ReadSectors(u8 *pBuffer,u32 SectorNo,u16 SectorSize,u32 SectorCount)
{
    u8 flag=0; u32 i=0;
    u16 rsecs;                  //单次读取页数
    u32 LBNNo;                  //逻辑块号
    u32 PBNNo;                  //物理块号
    u32 PhyPageNo;              //物理页号
    u32 PageOffset;             //页内偏移地址
    u32 BlockOffset;            //块内偏移地址
    for(i=0;i<SectorCount;i++)
    {
        LBNNo=(SectorNo+i)/(nand_dev.block_pagenum*(nand_dev.page_mainsize
                /SectorSize)); //根据逻辑扇区号和扇区大小计算出逻辑块号
        PBNNo=FTL_LBNToPBN(LBNNo);              //将逻辑块转换为物理块
        if(PBNNo>=nand_dev.block_totalnum)return 1;  //物理块号大于总块数,则失败.
        BlockOffset=((SectorNo+i)%(nand_dev.block_pagenum*(nand_dev.page_mainsize/
                    SectorSize)))*SectorSize;       //计算块内偏移
        PhyPageNo=PBNNo*nand_dev.block_pagenum+BlockOffset/
                  nand_dev.page_mainsize;           //计算出物理页号
        PageOffset=BlockOffset%nand_dev.page_mainsize;//计算出页内偏移地址
        rsecs=(nand_dev.page_mainsize-PageOffset)/SectorSize; //一次最多可以读取多少页
        if(rsecs>(SectorCount-i))rsecs=SectorCount-i;    //最多不能超过 SectorCount-i
        flag=NAND_ReadPage(PhyPageNo,PageOffset,pBuffer,rsecs*SectorSize);//读取数据
        if(flag==NSTA_ECC1BITERR) //对于 1bit ecc 错误,可能为坏块，读 2 次确认
            flag=NAND_ReadPage(PhyPageNo,PageOffset,pBuffer,rsecs*SectorSize);
        if(flag==NSTA_ECC1BITERR) //重读数据,再次确认,还是有 1BIT ECC 错误
        {
            //将整个数据，搬运到另外一个 block，防止此 block 是坏块
            FTL_CopyAndWriteToBlock(PhyPageNo,PageOffset,pBuffer,rsecs*SectorSize);
            flag=FTL_BlockCompare(PhyPageNo/nand_dev.block_pagenum,0XFFFFFFFF);
            //全 1 检查,确认是否为坏块
            if(flag==0)
            {
                flag=FTL_BlockCompare(PhyPageNo/nand_dev.block_pagenum,0X00);
                //全 0 检查,确认是否为坏块
                NAND_EraseBlock(PhyPageNo/nand_dev.block_pagenum);//检测完擦除
            }
            if(flag)//全 0/全 1 检查出错,肯定是坏块了.
            {
```

```
                FTL_BadBlockMark(PhyPageNo/nand_dev.block_pagenum);//标记为坏块
                FTL_CreateLUT(1);           //重建 LUT 表
            }
            flag=0;
        }
        //2bit ecc 错误,不处理(可能是初次读取数据导致的)
        if(flag==NSTA_ECC2BITERR)flag=0;
        if(flag)return 2;                   //失败
        pBuffer+=SectorSize*rsecs;          //数据缓冲区指针偏移
        i+=rsecs-1;
    }
    return 0;
}
```

该函数也是 FTL 层对文件系统的接口函数，用于读取 NAND FLASH 里面的数据，同样，用户在调用该函数时，无需关系坏块管理和磨损均衡问题，可以像访问 SD 卡一样，调用该函数，实现读取 NAND FLASH 的数据。该函数的实现原理，同前面介绍的 FTL_WriteSectors 函数基本类似，不过该函数对读数时出现的 ECC 错误，进行了处理，对于读取数据时出现 1bit ECC 错误的 Block，进行两次读取（多次确认，以免误操作），如果两次读取都有 1bitECC 错误，那么该 Block 可能是坏块，当出现此错误后，我们先将该 Block 的数据，拷贝到另外一个 Block（备份现有数据），然后对该 Block 进行擦除和写 0，然后判断擦除/写 0 是否正常，如果正常，则说明这个块不是坏块，还可以继续使用。如果不正常，则说明该块确实是一个坏块，必须进行坏块标记，并重建 lut 表。如果读数时出现 2bit ecc 错误，这个不一定就是出错了，而有可能是读取还未写入过数据的 Block（未写入过数据，那么 ECC 值，肯定也是未写入过，如果进行 ECC 校验的话，必定出错），导致的 ECC 错误，对于此类错误，我们直接不予处理（忽略）就可以了。

ftl.c 里面的其他代码以及 ftl.h 里面的代码，这里就不做介绍了，请大家参考本例程源码。另外，nandtester.c 和 nandtester.h 这两个文件，主要用于 usmart 调试 nand.c 和 ftl.c 里面的相关函数，这里也不做介绍了，请大家参考本例程源码。

最后，打开 main.c 文件，代码如下：

```
int main(void)
{
    u8 key,t=0;
    u16 i;
    u8 *buf;
    u8 *backbuf;

    Cache_Enable();                 //打开 L1-Cache
    MPU_Memory_Protection();        //保护相关存储区域
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
```

```
LED_Init();                             //初始化 LED
KEY_Init();                             //初始化按键
SDRAM_Init();                           //初始化 SDRAM
LCD_Init();                             //初始化 LCD
my_mem_init(SRAMIN);                    //初始化内部内存池
my_mem_init(SRAMEX);                    //初始化外部 SDRAM 内存池
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
LCD_ShowString(30,70,200,16,16,"NAND TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2016/7/15");
LCD_ShowString(30,130,200,16,16,"KEY0:Read Sector 2");
LCD_ShowString(30,150,200,16,16,"KEY1:Write Sector 2");
LCD_ShowString(30,170,200,16,16,"KEY2:Recover Sector 2");
while(FTL_Init())                       //检测 NAND FLASH,并初始化 FTL
{
    LCD_ShowString(30,190,200,16,16,"NAND Error!");
    delay_ms(500);
    LCD_ShowString(30,190,200,16,16,"Please Check");
    delay_ms(500);
    LED0_Toggle;//DS0 闪烁
}
backbuf=mymalloc(SRAMIN,NAND_ECC_SECTOR_SIZE);  //申请一个扇区的缓存
buf=mymalloc(SRAMIN,NAND_ECC_SECTOR_SIZE);  //申请一个扇区的缓存
POINT_COLOR=BLUE;                       //设置字体为蓝色
sprintf((char*)buf,"NAND Size:%dMB",
                (nand_dev.block_totalnum/1024)*(nand_dev.page_mainsize/1024)* \
                nand_dev.block_pagenum);
LCD_ShowString(30,190,200,16,16,buf);    //显示 NAND 容量
FTL_ReadSectors(backbuf,2,NAND_ECC_SECTOR_SIZE,1);
                    //预先读取扇区 0 到备份区域,防止乱写导致文件系统损坏.
while(1)
{
    key=KEY_Scan(0);
    switch(key)
    {
        case KEY0_PRES://KEY0 按下,读取 sector
            key=FTL_ReadSectors(buf,2,NAND_ECC_SECTOR_SIZE,1);//读取扇区
            if(key==0)//读取成功
            {
                LCD_ShowString(30,210,200,16,16,"USART1 Sending Data...   ");
                printf("Sector 2 data is:\r\n");
                for(i=0;i<NAND_ECC_SECTOR_SIZE;i++)
```

```
                {
                    printf("%x ",buf[i]);//输出数据
                }
                printf("\r\ndata end.\r\n");
                LCD_ShowString(30,210,200,16,16,"USART1 Send Data Over!    ");
            }
            break;
        case KEY1_PRES://KEY1 按下,写入 sector
            for(i=0;i<NAND_ECC_SECTOR_SIZE;i++)buf[i]=i+t;
                                    //填充数据(随机的,根据 t 的值来确定)
            LCD_ShowString(30,210,210,16,16,"Writing data to sector..");
            key=FTL_WriteSectors(buf,2,NAND_ECC_SECTOR_SIZE,1);//写入扇区
            if(key==0)
              LCD_ShowString(30,210,200,16,16,"Write data successed ");//写入成功
            else
              LCD_ShowString(30,210,200,16,16,"Write data failed    ");//写入失败
            break;
        case KEY2_PRES://KEY2 按下,恢复 sector 的数据
            LCD_ShowString(30,210,210,16,16,"Recovering data...          ");
            key=FTL_WriteSectors(backbuf,2,
                                    NAND_ECC_SECTOR_SIZE,1);//写入扇区
            if(key==0)LCD_ShowString(30,210,200,16,16,
                                    "Recovering data OK          ");//恢复成功
            else LCD_ShowString(30,210,200,16,16,
                                    "Recovering data failed    ");//恢复失败
            break;
    }
    t++;
    delay_ms(10);
    if(t==20)
    {
        LED0_Toggle;
        t=0;
    }
  }
}
```

此部分代码比较简单，我们先初始化相关外设，然后初始化 FTL，在 FTL 初始化成功以后，先对扇区 2 的数据进行备份，随后进入死循环，检测按键，可以通过 KEY0/KEY1/KEY2 按键对扇区 2 的数据进行读取、写入和还原操作。同样，DS0 闪烁，用于提示程序正在运行。

最后，我们将 NAND_EraseChip、NAND_EraseBlock、FTL_CreateLUT、FTL_Format、test_writepage 和 test_readpage 等函数加入 USMART 控制，这样，我们就可以通过串口调试助手，测试 NAND FLASH 的各种操作了，方便大家测试。软件部分就给大家介绍到这里。

46.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到如图 46.4.1 所示界面：

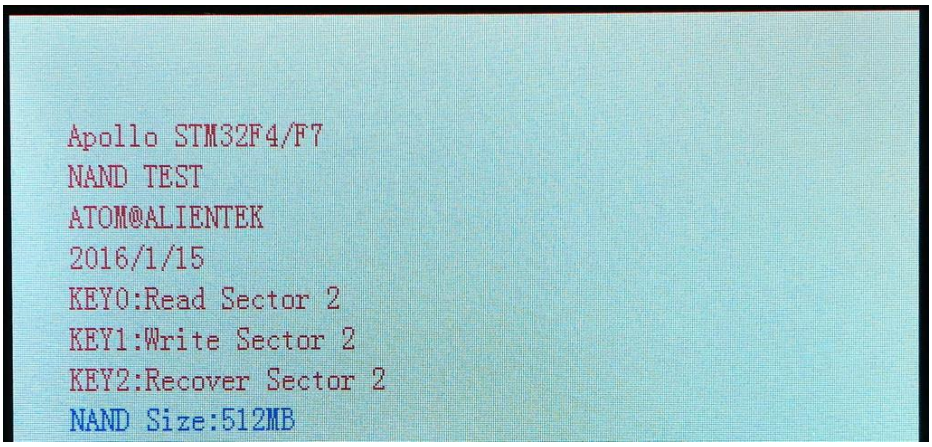


图 46.4.1 程序运行效果图

此时，我们可以按下 KEY0/KEY1/KEY2 等按键进行对应的测试。我们按 KEY0，可以读取扇区 2 里面的数据，通过串口调试助手查看，如图 46.4.2 所示：

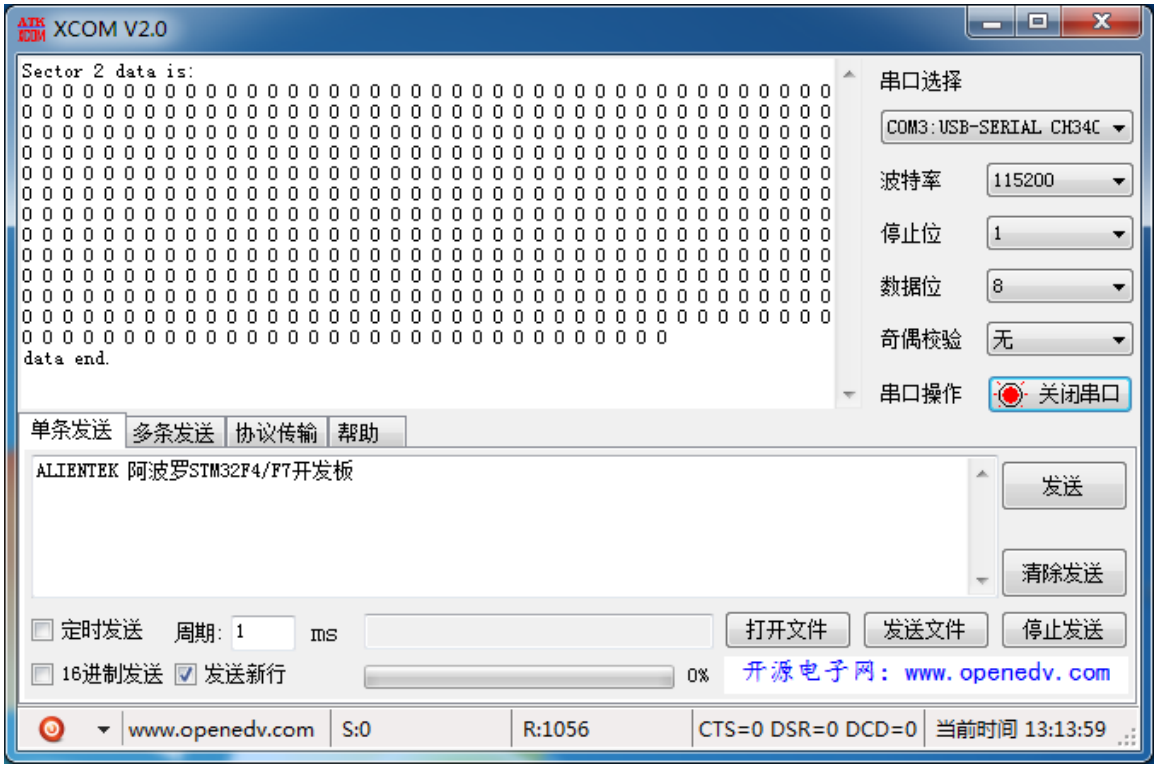


图 46.4.3 串口观看扇区 2 里面的数据

另外，我们还可以利用 usmart，调用相关函数，执行不同的操作，如图 46.4.4 所示：

图 46.4.4 USMART 调用相关函数

第四十七章　FATFS 实验

上两章，我们学习了 SD 卡和 NAND FLASH 的使用，不过仅仅是简单的实现读写扇区而已，真正要好好应用它们，必须使用文件系统管理，本章，我们将使用 FATFS 来管理 SD 卡（同时也管理 NAND FLASH 和 SPI FLASH，不过仅以 SD 卡为例讲解），实现 SD 卡文件的读写等基本功能。本章分为如下几个部分：

47.1 FATFS 简介
47.2 硬件设计
47.3 软件设计
47.4 下载验证

47.1 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 、FAT32 和 exFAT（R0.12 及以后版本），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/exFAT）
- 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 47.1.1 所示：

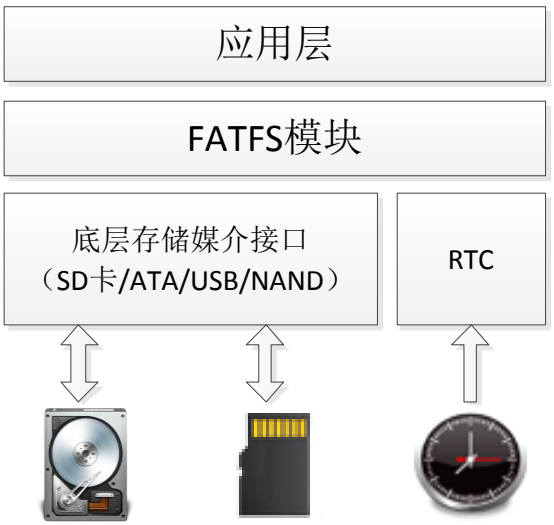


图 47.1.1 FATFS 层次结构图

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 R0.12a。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 3，即支持 3 个设备。

10）_MAX_SS。扇区缓冲的最大值，一般设置为 512。

11）_FS_EXFAT。用于定义是否支持 exFAT 文件系统，我们设置为 1，以支持 exFAT 文件

系统。

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 47.1.2 所示：

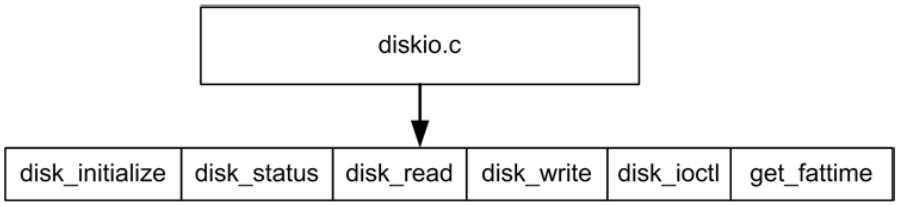


图 47.1.2 diskio 需要实现的函数

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 MDK5.21A 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 11 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 47.1.3 所示：

函数名称	disk_initialize
函数原型	DSTATUS disk_initialize(BYTE pdrv)
功能描述	初始化磁盘驱动器
函数参数	pdrv:指定要初始化的磁盘驱动器号,即盘符，取值范围：0~9
返回值	STA_NOINIT:磁盘未初始化 RES_OK:磁盘初始化成功
注意事项	1)该函数初始化一个磁盘驱动器,为读/写做准备,函数成功时，返回 0 2)应用程序不要调用该函数,否则可能损坏卷上的 FAT 结构 3)如果需要重新初始化,调用 f_mount 函数即可
使用示例	disk_initialize(0);//初始化驱动器 0(磁盘 0)

图 47.1.3 disk_initialize 函数介绍

第二个函数是 disk_status 函数，该函数介绍如图 47.1.4 所示：

函数名称	disk_status
函数原型	DSTATUS disk_status(BYTE pdrv)
功能描述	返回当前磁盘驱动器的状态
函数参数	pdrv:指定要确认的磁盘驱动器号,即盘符，取值范围：0~9

返回值	STA_NOINIT:表明磁盘未初始化 STA_NODISK:表明磁盘驱动器中没有设备 STA_PROTECT:表明磁盘被写保护 RES_OK:表示磁盘正常,可以支持接下来的操作
注意事项	无
使用示例	disk_status(0);//获取驱动器 0 的状态(磁盘 0)

图 47.1.4 disk_status 函数介绍

第三个函数是 disk_read 函数，该函数介绍如图 47.1.5 所示：

函数名称	disk_read
函数原型	DRESULT disk_read(BYTE pdrv,BYTE *buff,DWORD sector,UINT count)
功能描述	从磁盘驱动器上读取一个/多个扇区的数据
函数参数	pdrv:指定要读取的磁盘驱动器号,即盘符，取值范围：0~9 buff:数据缓冲区首地址 sector:指定要读取的起始扇区地址 count:指定要读取的扇区数(1~65535)
返回值	RES_OK:函数执行成功 RES_ERROR:读取期间产生了无法恢复的错误 RES_PARERR:非法参数 RES_NOTRDY:磁盘驱动器未准备好(未初始化)
注意事项	fatfs 指定的 buff 地址并不总是对齐的,如果硬件不支持不对齐数据传输,则需要在函数里面做相应处理,否则可能读数出错
使用示例	disk_read(0,buf,0,1);//从磁盘 0 的扇区 0 地址,读取 1 个扇区的数据

图 47.1.5 disk_read 函数介绍

第四个函数是 disk_write 函数，该函数介绍如图 47.1.6 所示：

函数名称	disk_write
函数原型	DRESULT disk_write(BYTE pdrv,BYTE *buff,DWORD sector,UINT count)
功能描述	往磁盘驱动器上写入一个/多个扇区的数据
函数参数	pdrv:指定要写入的磁盘驱动器号,即盘符，取值范围：0~9 buff:数据缓冲区首地址 sector:指定要写入的起始扇区地址 count:指定要写入的扇区数(1~65535)
返回值	RES_OK:函数执行成功 RES_ERROR:写入期间产生了无法恢复的错误 RES_PARERR:非法参数 RES_NOTRDY:磁盘驱动器未准备好(未初始化)
注意事项	fatfs 指定的 buff 地址并不总是对齐的,如果硬件不支持不对齐数据传输,则需要在函数里面做相应处理,否则可能写入出错
使用示例	disk_write(0,buf,0,1);//往磁盘 0 的扇区 0 地址,写 1 个扇区的数据

图 47.1.6 disk_write 函数介绍

第五个函数是 disk_ioctl 函数，该函数介绍如图 47.1.7 所示：

函数名称	disk_ioctl

函数原型	DRESULT disk_ioctl(BYTE pdrv,BYTE cmd,void *buff)
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	pdrv:指定要磁盘驱动器号,即盘符，取值范围：0~9 cmd:指定命令代码 buff:指向参数缓冲区首地址,取决于命令,无参数时,指向 NULL
返回值	RES_OK:函数执行成功 RES_ERROR:访问期间产生了无法恢复的错误 RES_PARERR:非法参数 RES_NOTRDY:磁盘驱动器未准备好(未初始化)
注意事项	CTRL_SYNC:确保磁盘驱动器已完成写处理,当磁盘 I/O 有一个写回缓存,立即刷新原扇区,只读配置下,该命令无效 GET_SECTOR_SIZE:获取磁盘扇区大小,_MAX_SS>=1024 时可用 GET_SECTOR_COUNT:获取磁盘扇区总数 GET_BLOCK_SIZE:获取磁盘块大小
使用示例	disk_ioctl(0,cmd,buf);//往磁盘 0 发送 cmd 命令,参数为 buf

图 47.1.7 disk_ioctl 函数介绍

最后一个函数是 get_fattime 函数，该函数介绍如图 47.1.8 所示：

函数名称	get_fattime
函数原型	DWORD get_fattime (void)
功能描述	获取当前时间
函数参数	无
返回值	当前时间以 DWORD(u32)返回,位域为: bit31:25　年(0~127),从 1980 年开始 bit24:21　月(1~12) bit20:16　日(1~31) bit15:11　时(0~23) bit10:5　分(0~59) bit4:0　秒((0~29)*2),偶数秒
注意事项	1)如果没有用到 RTC,我们可以直接返回 0,但是无法记录 2)对于秒钟,只能精确到偶数秒
使用示例	temp=get_fattime();//获取当前时间,存放到 temp

图 47.1.8 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。

47.2 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，注册三个工作区（一个给 SD 卡用、一个给 SPI FLASH 用、一个给 NAND 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） LCD 模块

4） SD 卡

5） SPI FLASH

6） NAND 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。

47.3 软件设计

打开本章实验目录可以看到，我们在工程目录下新建了一个 FATFS 的文件夹，然后将 FATFS R0.12b 程序包解压到该文件夹下。同时，我们在 FATFS 文件夹里面新建了一个 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 47.3.1 所示：

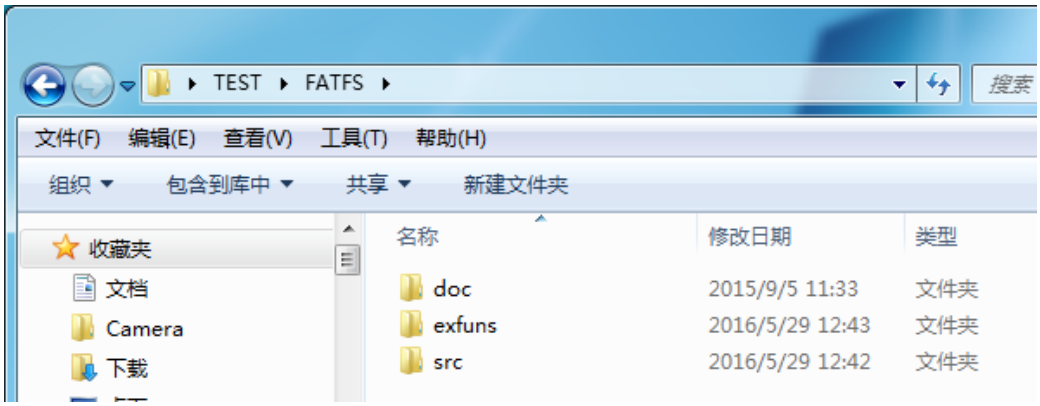


图 47.3.1 FATFS 文件夹子目录

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 spi flash,卷标为 1
#define EX_NAND          2          //外部 nand flash,卷标为 2

//对于 W25Q256
//前 25M 字节给 fatfs 用,25M 字节后存放字库,字库占用 6.01M.剩余部分,给客户自己用
#define FLASH_SECTOR_SIZE   512
#define FLASH_SECTOR_COUNT  1024*25*2     //W25Q1218,前 25M 字节给 FATFS 占用
#define FLASH_BLOCK_SIZE    8             //每个 BLOCK 有 8 个扇区

//NAND FALSH 全部归 FATFS 管理
u32     NANDFLASH_SECTOR_COUNT;
u8      NANDFLASH_BLOCK_SIZE;

//初始化磁盘
```

```
DSTATUS disk_initialize (
	BYTE pdrv				/* Physical drive nmuber (0..) */
)
{
	u8 res=0;
	switch(pdrv)
	{
		case SD_CARD:		 //SD 卡
			res=SD_Init();	//SD 卡初始化
			break;
		case EX_FLASH:		 //外部 flash
			W25QXX_Init();	//W25QXX 初始化
			break;
		case EX_NAND:		//外部 NAND
			res=FTL_Init();	//NAND 初始化
			break;
		default:
			res=1;
	}
	if(res)return	STA_NOINIT;
	else return 0;			//初始化成功
}

//获得磁盘状态
DSTATUS disk_status (
	BYTE pdrv		/* Physical drive nmuber (0..) */
)
{
	return 0;
}

//读扇区
//drv:磁盘编号 0~9
//*buff:数据接收缓冲首地址
//sector:扇区地址
//count:需要读取的扇区数
DRESULT disk_read (
	BYTE pdrv,		/* Physical drive nmuber (0..) */
	BYTE *buff,		/* Data buffer to store read data */
	DWORD sector,	/* Sector address (LBA) */
	UINT count		/* Number of sectors to read (1..128) */
)
{
```

```
    u8 res=0;
    if (!count)return RES_PARERR;//count 不能等于 0，否则返回参数错误
    switch(pdrv)
    {
        case SD_CARD://SD 卡
            res=SD_ReadDisk(buff,sector,count);
            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;
        case EX_NAND:    //外部 NAND FLASH
            res=FTL_ReadSectors(buff,sector,512,count);      //读取数据
            break;
        default:
            res=1;
    }
   //处理返回值，将 SPI_SD_driver.c 的返回值转成 ff.c 的返回值
    if(res==0x00)return RES_OK;
    else return RES_ERROR;
}

//写扇区
//drv:磁盘编号 0~9
//*buff:发送数据首地址
//sector:扇区地址
//count:需要写入的扇区数
#if _USE_WRITE
DRESULT disk_write (
    BYTE pdrv,                /* Physical drive nmuber (0..) */
    const BYTE *buff,  /* Data to be written */
```

```
    DWORD sector,           /* Sector address (LBA) */
    UINT count              /* Number of sectors to write (1..128) */
)
{
    u8 res=0;
    if (!count)return RES_PARERR;//count 不能等于 0，否则返回参数错误
    switch(pdrv)
    {
        case SD_CARD://SD 卡
            res=SD_WriteDisk((u8*)buff,sector,count);
            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;
        case EX_NAND:    //外部 NAND FLASH
            res=FTL_WriteSectors((u8*)buff,sector,512,count);//写入数据
            break;
        default:
            res=1;
    }
    //处理返回值，将 SPI_SD_driver.c 的返回值转成 ff.c 的返回值
    if(res == 0x00)return RES_OK;
    else return RES_ERROR;
}
#endif


//其他表参数的获得
 //drv:磁盘编号 0~9
 //ctrl:控制代码
```

```
 //*buff:发送/接收缓冲区指针
#if _USE_IOCTL
DRESULT disk_ioctl (
	BYTE pdrv,		/* Physical drive nmuber (0..) */
	BYTE cmd,		/* Control code */
	void *buff		/* Buffer to send/receive control data */
)
{
	DRESULT res;
	if(pdrv==SD_CARD)//SD 卡
	{
		switch(cmd)
		{
			case CTRL_SYNC:
				res = RES_OK;
				break;
			case GET_SECTOR_SIZE:
				*(DWORD*)buff = 512;
				res = RES_OK;
				break;
			case GET_BLOCK_SIZE:
				*(WORD*)buff = SDCardInfo.CardBlockSize;
				res = RES_OK;
				break;
			case GET_SECTOR_COUNT:
				*(DWORD*)buff = SDCardInfo.CardCapacity/512;
				res = RES_OK;
				break;
			default:
				res = RES_PARERR;
				break;
		}
	}else if(pdrv==EX_FLASH)  //外部 FLASH
	{
		switch(cmd)
		{
			case CTRL_SYNC:
				res = RES_OK;
				break;
			case GET_SECTOR_SIZE:
				*(WORD*)buff = FLASH_SECTOR_SIZE;
				res = RES_OK;
				break;
```

```
            case GET_BLOCK_SIZE:
                *(WORD*)buff = FLASH_BLOCK_SIZE;
                res = RES_OK;
                break;
            case GET_SECTOR_COUNT:
                *(DWORD*)buff = FLASH_SECTOR_COUNT;
                res = RES_OK;
                break;
            default:
                res = RES_PARERR;
                break;
        }
    }else if(pdrv==EX_NAND)        //外部 NAND FLASH
    {
        switch(cmd)
        {
            case CTRL_SYNC:
                res = RES_OK;
                break;
            case GET_SECTOR_SIZE:
                *(WORD*)buff = 512;   //NAND FLASH 扇区强制为 512 字节大小
                res = RES_OK;
                break;
            case GET_BLOCK_SIZE:
                *(WORD*)buff = nand_dev.page_mainsize/512;
                                        //block 大小,定义成一个 page 的大小
                res = RES_OK;
                break;
            case GET_SECTOR_COUNT:
                *(DWORD*)buff =
                        nand_dev.valid_blocknum*nand_dev.block_pagenum*
                        nand_dev.page_mainsize/512;//NAND FLASH 的总扇区大小
                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 47.1 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 3 个磁盘：SD 卡、SPI FLASH 和 NAND FLASH。SD 卡和 NAND FLASH，其扇区大小一般固定为 512 字节，而对 SPI FLASH，因为其物理扇区（擦除单位）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。

NAND FLASH 与文件系统的读写接口，就是采用上一章介绍的 FTL 层函数（FTL_ReadSectors 和 FTL_WriteSectors 函数）来实现的，有了 FTL 层，我们就可以像访问 SD 卡一样，访问 NAND FLASH，而无需担心坏块和磨损均衡问题。

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，EX_NAND 为 2，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、SPI FLASH 或 NAND 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。最后，我们打开 main.c， main 函数如下：

```
int main(void)
{
    u32 total,free;
    u8 t=0;
    u8 res=0;

    Cache_Enable();                 //打开 L1-Cache
    MPU_Memory_Protection();        //保护相关存储区域
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部内存池
    my_mem_init(SRAMDTCM);          //初始化 CCM 内存池
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/7/15");
    LCD_ShowString(30,130,200,16,16,"Use USMART for test");
    while(SD_Init())//检测不到 SD 卡
    {
        LCD_ShowString(30,150,200,16,16,"SD Card Error!");
        delay_ms(500);
        LCD_ShowString(30,150,200,16,16,"Please Check! ");
        delay_ms(500);
        LED0_Toggle;//DS0 闪烁
    }
    FTL_Init();
    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 磁盘的名字为：ALIENTEK
            LCD_ShowString(30,150,200,16,16,"Flash Disk Format Finish");//格式化完成
        }else LCD_ShowString(30,150,200,16,16,"Flash Disk Format Error ");//格式化失败
        delay_ms(1000);
    }
    res=f_mount(fs[2],"2:",1);                  //挂载 NAND FLASH.
    if(res==0X0D)//NAND FLASH 磁盘,FAT 文件系统错误,重新格式化 NAND FLASH
    {
        LCD_ShowString(30,150,200,16,16,"NAND Disk Formatting...");//格式化 NAND
        res=f_mkfs("2:",1,4096);//格式化 FLASH,2,盘符;1,不需要引导区,8 个扇区为 1 个簇
        if(res==0)
        {
            f_setlabel((const TCHAR *)"2:NANDDISK");
                                        //设置 Flash 磁盘的名字为：NANDDISK
            LCD_ShowString(30,150,200,16,16,"NAND Disk Format Finish");//格式化完成
        }else LCD_ShowString(30,150,200,16,16,"NAND 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_Toggle;//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_Toggle;
    }
```

```
}
```

在 main 函数里面，我们为 SD、SPI FLASH 和 NAND 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*)W25QXX_Erase_Chip,"void W25QXX_Erase_Chip(void)",
(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)",
(void*)NAND_EraseBlock,"u8 NAND_EraseBlock(u32 BlockNum)",
(void*)NAND_EraseChip,"void NAND_EraseChip(void)",
```

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。

47.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 显示如图 47.4.1 所示的内容（假定 SD 卡已经插上了）：

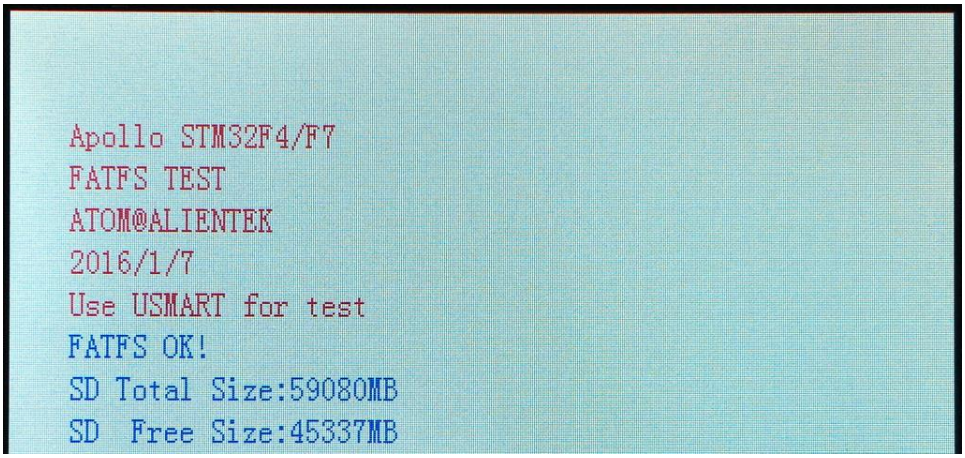


图 47.4.1 程序运行效果图

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 47.4.2 所示：

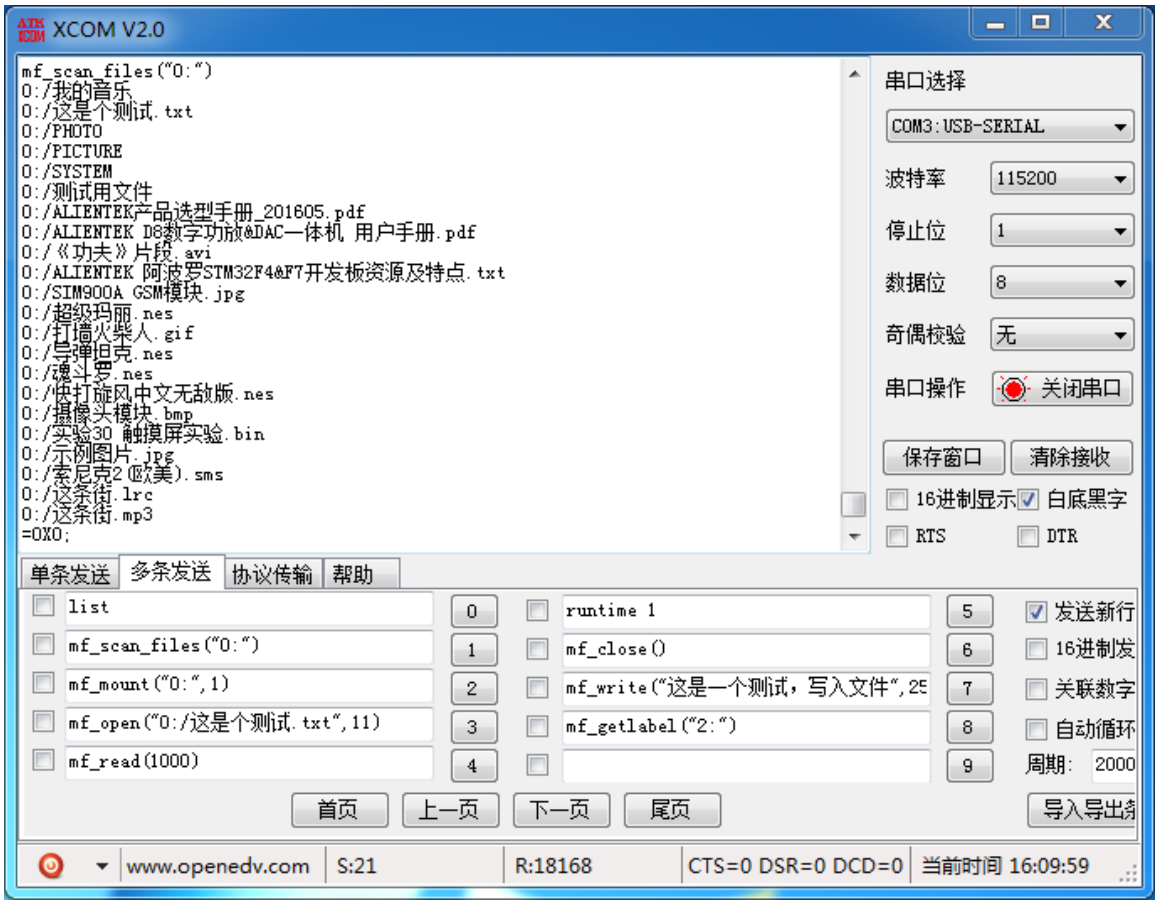


图 47.4.2 扫描 SD 卡根目录所有文件

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，2 代表 NAND 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。

第四十八章 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 STM32F767 控制 LCD 显示汉字。在本章中，我们将使用外部 SPI FLASH 来存储字库，并可以通过 SD 卡更新字库。STM32F767 读取存在 SPI 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：

48.1 汉字显示原理简介
48.2 硬件设计
48.3 软件设计
48.4 下载验证

48.1 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 W25Q256 里，这样，W25Q256 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，我们以 12*12 的汉字为例，假设其取模方向为从上到下，从左到右的方向取模，且高位在前，那么其取模原理如图 48.1.1 所示：

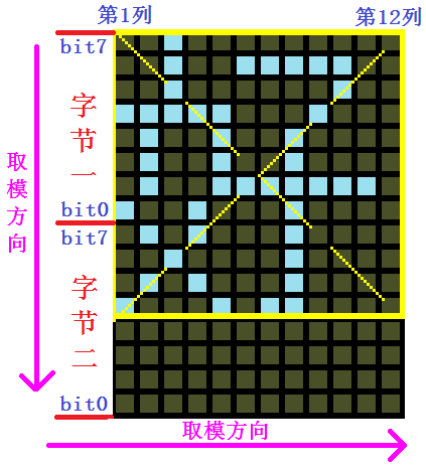


图 48.1.1 从上到下，从左到右取模原理

图中，我们取模的时候，从最左上方的点开始取（从上到下，从左到右），且高位在前（bit7 在表示第一个位），那么第一个字节就是：0X11（1，表示浅蓝色的点，即要画出来的点，0 则表示不要画出来），第二个字节是：0X10，第三个字节（到第二列了，每列 2 个字节）是：0X1E……，依次类推，一个 12*12 的汉字，总共有 12 列，每列 2 个字节，总共需要 24 个字节来表示。

在显示的时候，我们只需要读取这个汉字的点阵数据（12*12 字体，一个汉字的点阵数据为 24 个字节），然后将这些数据，按取模方式，反向解析出来（坐标要处理好），每个字节，

是 1 的位，就画出来，不是 1 的位，就忽略，这样，就可以显示出这个汉字了。

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。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 48.1.2 所示：

图 48.1.2 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 48.1.3 所示：

图 48.1.3 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（从上到下，从左到右，且字节高位在前，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，在生成完以后，我们手动修改后缀为.fon)。具体设置如图 48.1.4 所示：

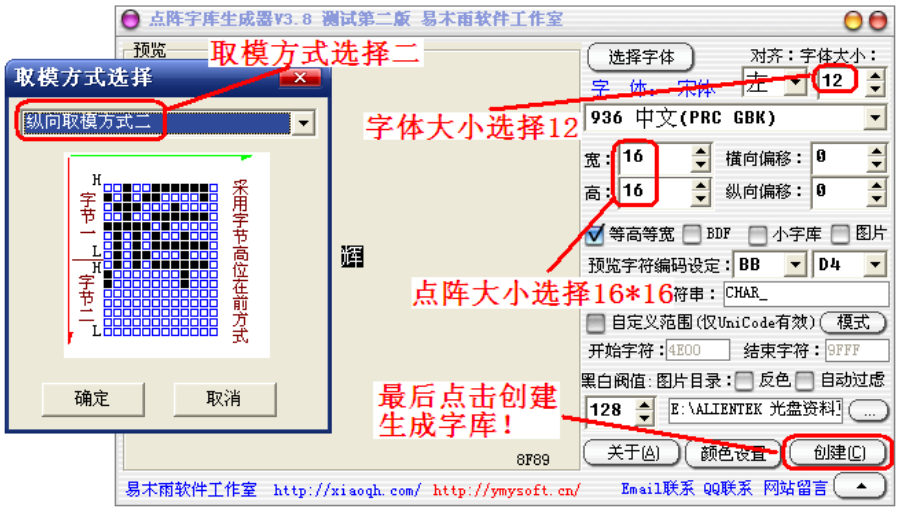


图 48.1.4 生成 GBK16*16 字库的设置方法

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、32 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）、24*24 的点阵库（GBK24.FON）和 32*32 的点阵库（GBK32.FON），总共制作 4 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。

48.2 硬件设计

本章实验功能简介：开机的时候先检测 W25Q256 中是否已经存在字库，如果存在，则按次序显示汉字(四种字体都显示)。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、GBK24.FON 和 GBK32.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） LCD 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。

48.3 软件设计

打开本章实验目录可以看到，首先在工程根目录文件夹下面新建了一个 TEXT 的文件夹。在 TEXT 文件夹下新建 fontupd.c、fontupd.h、text.c、text.h 这 4 个文件。同时，我们在实验工程中新建了 TEXT 分组，将新建的源文件加入到了分组之下，并将头文件包含路径加入到了工程的 PATH 中。

打开 fontupd.c，代码如下：

```
//字库区域占用的总扇区数大小(4 个字库+unigbk 表+字库信息=6302984 字节,约占 1539 个 W25QXX 扇区,一个扇区 4K 字节)
#define FONTSECSIZE     1539
//字库存放起始地址
#define FONTINFOADDR    1024*1024*25//开发板是从 25M 地址以后开始存放字库
                        //前面 25M 被 fatfs 占用了，25M 以后紧跟 4 个字库+UNIGBK.BIN,总大
                        //小 6.01M,被字库占用了,不能动! 31.01M 以后,用户可以自由使用.

//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 GBK_PATH[5]=
{
"/SYSTEM/FONT/UNIGBK.BIN",      //UNIGBK.BIN 的存放位置
"/SYSTEM/FONT/GBK12.FON", //GBK12 的存放位置
"/SYSTEM/FONT/GBK16.FON", //GBK16 的存放位置
"/SYSTEM/FONT/GBK24.FON", //GBK24 的存放位置
"/SYSTEM/FONT/GBK32.FON", //GBK32 的存放位置
};
```

```
//更新时的提示信息
u8*const UPDATE_REMIND_TBL[5]=
{
"Updating UNIGBK.BIN",    //提示正在更新 UNIGBK.bin
"Updating GBK12.FON",     //提示正在更新 GBK12
"Updating GBK16.FON",     //提示正在更新 GBK16
"Updating GBK24.FON",     //提示正在更新 GBK24
"Updating GBK32.FON",     //提示正在更新 GBK32
};

//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)
{
   …//此处省略部分代码
}
//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;4,gbk32;
//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);
                                                //信息头之后，紧跟 UNIGBK 转换码表
        ftinfo.ugbksize=fftemp->fsize;                  //UNIGBK 大小
        flashaddr=ftinfo.ugbkaddr;
        break;
    case 1:
        ftinfo.f12addr=ftinfo.ugbkaddr+ftinfo.ugbksize;
                                                //UNIGBK 之后，紧跟 GBK12 字库
        ftinfo.gbk12size=fftemp->fsize;     //GBK12 字库大小
        flashaddr=ftinfo.f12addr;           //GBK12 的起始地址
        break;
    case 2:
        ftinfo.f16addr=ftinfo.f12addr+ftinfo.gbk12size;
                                                //GBK12 之后，紧跟 GBK16 字库
        ftinfo.gbk16size=fftemp->fsize;     //GBK16 字库大小
        flashaddr=ftinfo.f16addr;           //GBK16 的起始地址
        break;
    case 3:
        ftinfo.f24addr=ftinfo.f16addr+ftinfo.gbk16size;
                                                //GBK16 之后，紧跟 GBK24 字库
        ftinfo.gbk24size=fftemp->fsize;                 //GBK24 字库大小
        flashaddr=ftinfo.f24addr;                       //GBK24 的起始地址
        break;
    case 4:
        ftinfo.f32addr=ftinfo.f24addr+ftinfo.gbk24size;
                                                //GBK24 之后，紧跟 GBK32 字库
        ftinfo.gbk32size=fftemp->fsize;                 //GBK32 字库大小
        flashaddr=ftinfo.f32addr;                       //GBK32 的起始地址
        break;
}

while(res==FR_OK)//死循环执行
{
    res=f_read(fftemp,tempbuf,4096,(UINT *)&bread);         //读取数据
    if(res!=FR_OK)break;                                    //执行错误
    W25QXX_Write(tempbuf,offx+flashaddr,4096);  //从 0 开始写入 4096 个数据
    offx+=bread;
    fupd_prog(x,y,size,fftemp->fsize,offx);                 //进度显示
    if(bread!=4096)break;                                   //读完了.
}
f_close(fftemp);
```

```
        }
        myfree(SRAMIN,fftemp);      //释放内存
        myfree(SRAMIN,tempbuf);     //释放内存
        return res;
}
//更新字体文件,UNIGBK,GBK12,GBK16,GBK24,GBK32 一起更新
//x,y:提示信息的显示地址
//size:字体大小
//src:字库来源磁盘."0:",SD 卡;"1:",FLASH 盘,"2:",U 盘.
//提示信息字体大小
//返回值:0,更新成功;
//              其他,错误代码.
u8 update_font(u16 x,u16 y,u8 size,u8* src)
{
        u8 *pname;
        u32 *buf;
        u8 res=0;
        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
        }
        for(i=0;i<5;i++)      //先查找文件 UNIGBK,GBK12,GBK16,GBK24,GBK32 是否正常
        {
                strcpy((char*)pname,(char*)src);                        //copy src 内容到 pname
                strcat((char*)pname,(char*)GBK_PATH[i]);        //追加具体文件路径
                res=f_open(fftemp,(const TCHAR*)pname,FA_READ);     //尝试打开
                if(res)
                {
                        rval|=1<<7;     //标记打开文件失败
                        break;          //出错了,直接退出
                }
        }
```

```
    myfree(SRAMIN,fftemp);      //释放内存
    if(rval==0)                 //字库文件都存在.
    {
        LCD_ShowString(x,y,240,320,size,"Erasing sectors... ");//提示正在擦除扇区
        for(i=0;i<FONTSECSIZE;i++)              //先擦除字库区域,提高写入速度
        {
            fupd_prog(x+20*size/2,y,size,FONTSECSIZE,i);//进度显示
            W25QXX_Read((u8*)buf,((FONTINFOADDR/4096)+i)*4096,4096);
                                                        //读出整个扇区的内容
            for(j=0;j<1024;j++)//校验数据
            {
                if(buf[j]!=0XFFFFFFFF)break;//需要擦除
            }
            if(j!=1024)W25QXX_Erase_Sector((FONTINFOADDR/4096)+i);
                                                        //需要擦除的扇区
        }
        for(i=0;i<5;i++)    //依次更新 UNIGBK,GBK12,GBK16,GBK24,GBK32
        {
            LCD_ShowString(x,y,240,320,size,UPDATE_REMIND_TBL[i]);
            strcpy((char*)pname,(char*)src);                //copy src 内容到 pname
            strcat((char*)pname,(char*)GBK_PATH[i]);        //追加具体文件路径
            res=updata_fontx(x+20*size/2,y,size,pname,i);   //更新字库
            if(res)
            {
                myfree(SRAMIN,buf);
                myfree(SRAMIN,pname);
                return 1+i;
            }
        }
        //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));
                                                    //读出 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 W25Q256 的前 25M 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 6.01M，最后 W25Q256 还剩下约 0.99M 给用户自己用。

接下来我们打开 fontupd.h 文件代码如下：

```
extern u32 FONTINFOADDR; //字体信息保存地址,占 41 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
//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 gbk24size;          //gbk24 的大小
    u32 f32addr;            //gbk32 地址
    u32 gbk32size;          //gbk32 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 41 个字节，第一个字节用来标识字库是否 OK，其他的用来记录地址和文件大小。

接下来打开 text.c 文件，代码如下：

```
//code  字符指针开始
//从字库中查找出字模
//code  字符串的开始地址,GBK 码
//mat   数据存放地址  (size/8+((size%8)?1:0))*(size) bytes 大小
//size:字体大小
void Get_HzMat(unsigned char *code,unsigned char *mat,u8 size)
{
    unsigned char qh,ql;
    unsigned char i;
    unsigned long foffset;
    u8 csize=(size/8+((size%8)?1:0))*(size);//得到字体一个字符对应点阵集所占的字节数
    qh=*code;
    ql=*(++code);
    if(qh<0x81||ql<0x40||ql==0xff||qh==0xff)//非  常用汉字
    {
        for(i=0;i<csize;i++)*mat++=0x00;//填充满格
        return; //结束访问
    }
    if(ql<0x7f)ql-=0x40;//注意!
    else ql-=0x41;
    qh-=0x81;
    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;
        case 32:
            W25QXX_Read(mat,foffset+ftinfo.f32addr,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[128];
    u8 csize=(size/8+((size%8)?1:0))*(size);//得到字体一个字符对应点阵集所占的字节数
    if(size!=12&&size!=16&&size!=24&&size!=32)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)
{
 …//此处省略部分代码
}
```

```
//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)
{
…//此处省略部分代码
}
```

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 47.1 节介绍的办法，在外部 flash 查找字库，然后返回对应的字库点阵。Show_Font 函数用于在指定地址显示一个指定大小的汉字，采用的方法和 LCD_ShowChar 所采用的方法一样，都是画点显示，这里就不细说了。

text.h 头文件是一些函数申明，我们这里不细说了。

前面提到我们对 cc936.c 文件做了修改，我们将其命名为 mycc936.c，并保存在 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，我们看看 main 函数如下：

```
int main(void)
{
    u32 fontcnt;
    u8 i,j;
    u8 fontx[2];                    //gbk 码
    u8 key,t;

    Cache_Enable();                 //打开 L1-Cache
    MPU_Memory_Protection();        //保护相关存储区域
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部 SDRAM 内存池
    my_mem_init(SRAMDTCM);          //初始化内部 DTCM 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 SPI FLASH.
    f_mount(fs[2],"2:",1);          //挂在 NAND 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,"Apollo STM32F4/F7");
        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(30,30,200,16,"阿波罗 STM32F4/F7 开发板",16,0);
    Show_Str(30,50,200,16,"GBK 字库测试程序",16,0);
    Show_Str(30,70,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,90,200,16,"2016 年 7 月 15 日",16,0);
    Show_Str(30,110,200,16,"按 KEY0,更新字库",16,0);
    POINT_COLOR=BLUE;
    Show_Str(30,130,200,16,"内码高字节:",16,0);
    Show_Str(30,150,200,16,"内码低字节:",16,0);
    Show_Str(30,170,200,16,"汉字计数器:",16,0);

    Show_Str(30,200,200,32,"对应汉字为:",32,0);
    Show_Str(30,232,200,24,"对应汉字为:",24,0);
    Show_Str(30,256,200,16,"对应汉字(16*16)为:",16,0);
    Show_Str(30,272,200,12,"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(118,150,j,3,16);  //显示内码低字节
                    LCD_ShowNum(118,170,fontcnt,5,16);//汉字计数显示
                    fontx[1]=j;
                    Show_Font(30+176,200,fontx,32,0);
                    Show_Font(30+132,232,fontx,24,0);
                    Show_Font(30+144,256,fontx,16,0);
                    Show_Font(30+108,272,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_Toggle;
                }
            }
        }
    }
```

此部分代码就实现了我们在硬件描述部分所描述的功能，至此整个软件设计就完成了。这节有太多的代码，而且工程也增加了不少，我们来看看工程的截图吧，整个工程截图如图 45.3.1 所示：

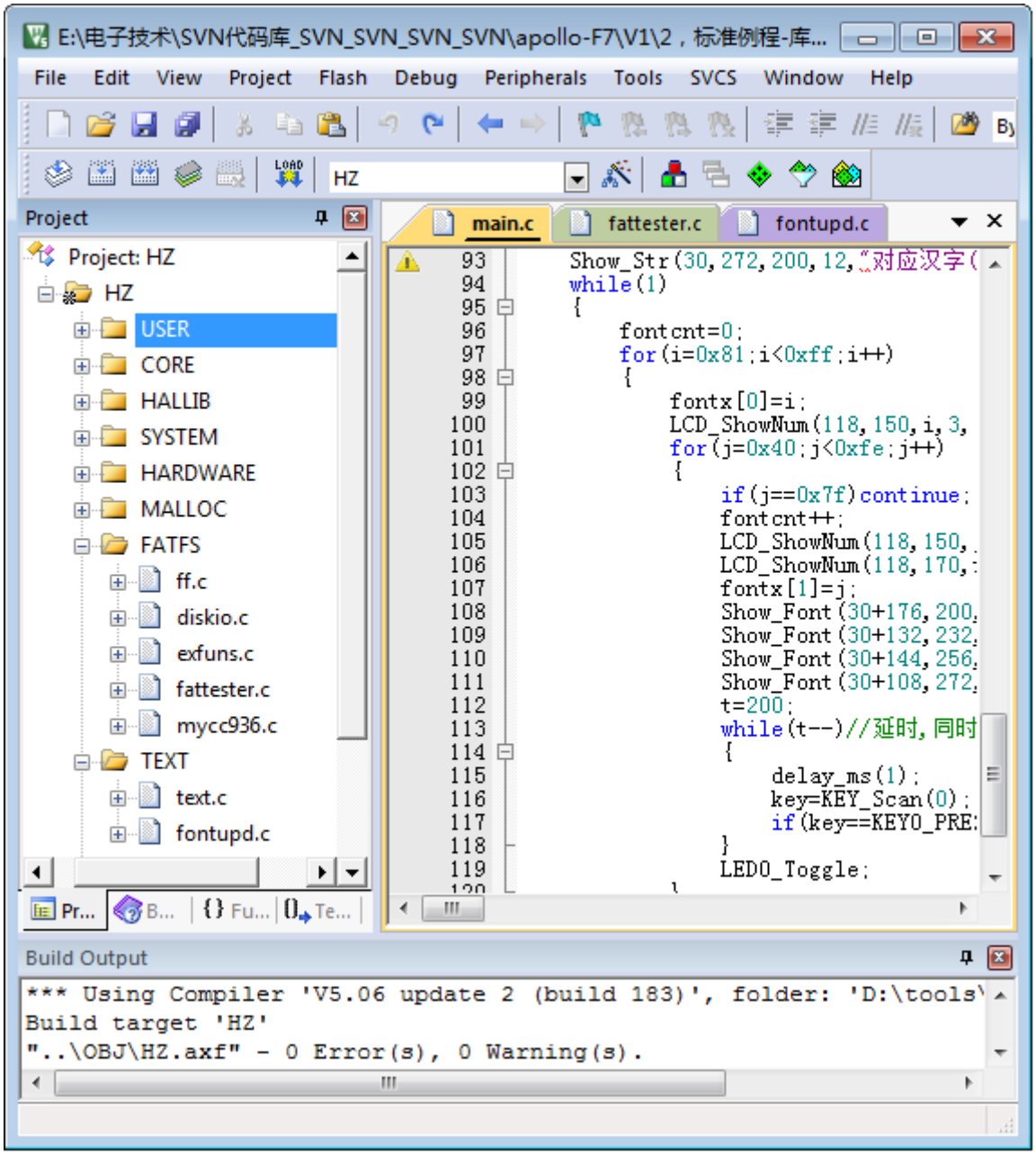


图 45.3.1 工程建成截图

48.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 开始显示汉字及汉字内码，如图 48.4.1 所示：

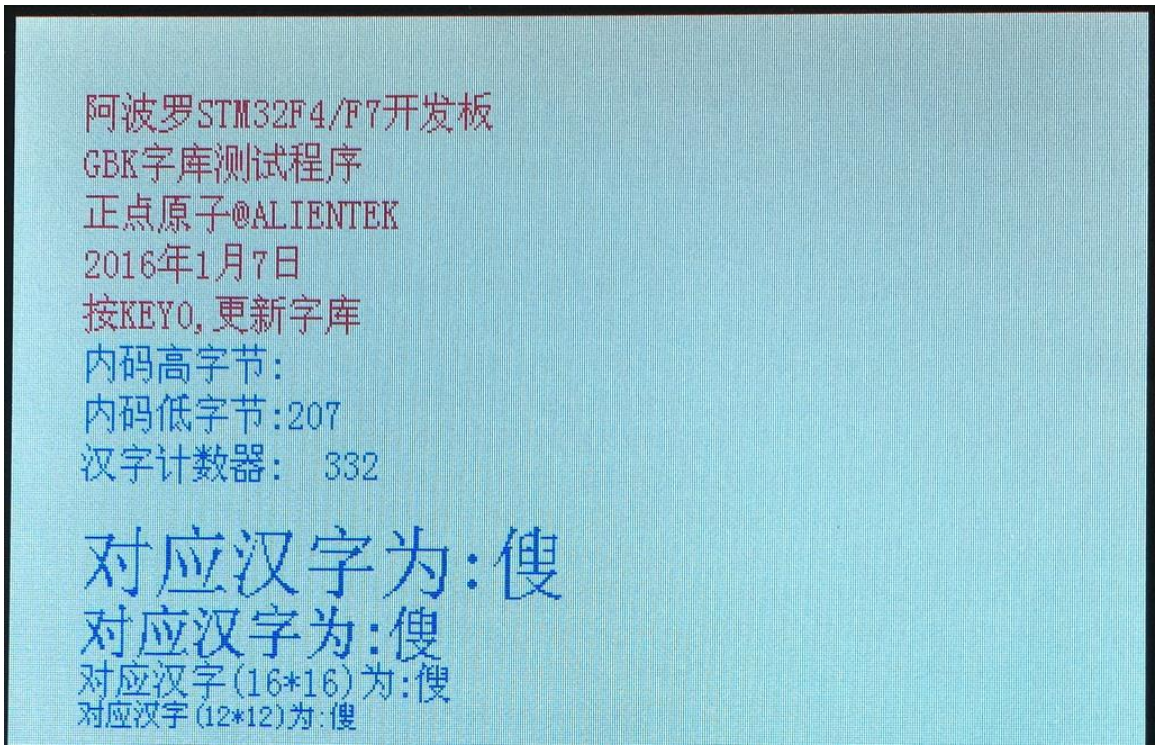


图 48.4.1 汉字显示实验显示效果

一开始就显示汉字，是因为 ALIENTEK 阿波罗 STM32F767 开发板在出厂的时候都是测试过的，里面刷了综合测试程序，已经把字库写入到了 W25Q256 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 48.4.2 所示：

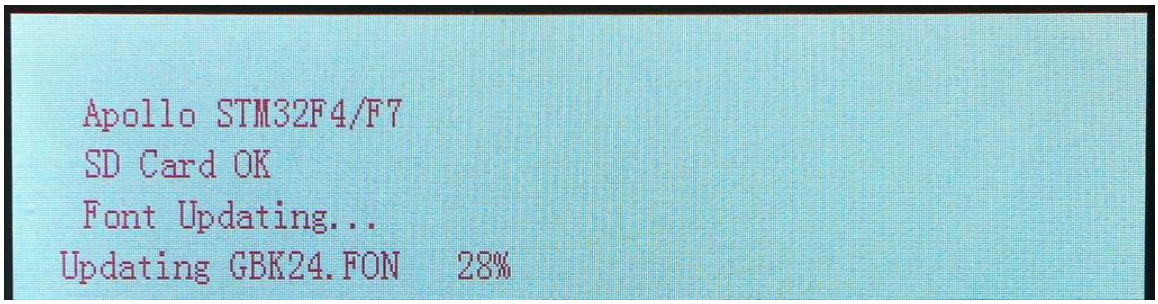


图 48.4.2 汉字字库更新界面

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。

第四十九章 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 STM32F767 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：

49.1 图片格式简介
49.2 硬件设计
49.3 软件设计
49.4 下载验证

49.1 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
1， 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；
2， 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。

49.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY2 可以快速浏览下一张和上一张，KEY_UP 按键用于暂停/继续播放，DS1 用于指示当前是否处于暂停状态。如果未找到

PICTURE 文件夹/任何图片文件，则提示错误。同样我们也是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0 和 DS1

2） KEY0、KEY2 和 KEY_UP 三个按键

3） 串口

4） LCD 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。

49.3 软件设计

打开本章实验工程目录可以看到，我们在工程根目录下面新建了一个 PICTURE 文件夹。在该文件夹里面新建了 bmp.c、bmp.h、tjpgd.c、tjpgd.h、integer.h、gif.c、gif.h、piclib.c 和 piclib.h 等 9 个文件。打开实验工程可以看到，我们在工程中新建了 PICTURE 分组，添加了相关源文件到工程，同时将 PICTURE 文件夹加入头文件包含路径。

对于这些文件，其中 bmp.c 和 bmp.h 用于实现对 bmp 文件的解码；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，代码如下：

```
extern u32 *ltdc_framebuf[2];      //LTDC LCD 帧缓存指针必须指向对应大小的内存区域

_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)
{
    u16 i,j;
    if(lcdltdc.pwidth!=0&&lcddev.dir==0)//如果是 RGB 屏,且竖屏,则填充函数不可直接用
    {
        for(i=0;i<height;i++)
        {
            for(j=0;j<width;j++)
            {
```

```
    *(u16*)((u32)ltdc_framebuf[lcdltdc.activelayer]+lcdltdc.pixsize*(lcdltdc.pwidth*(lcdltdc.pheigh
t-x-j-1)+y+i))=color[i*width+j];
                }
            }
        }else LCD_Color_Fill(x,y,x+width-1,y+height-1,color);//其他情况,直接填充
    }
    ///////////////////////////////////////////////////////////////////////
    //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);                    //解码 bmp
            break;
        case T_JPG:
        case T_JPEG:
            res=jpg_decode(filename,fast);                  //解码 JPG/JPEG
            break;
        case T_GIF:
            res=gif_decode(filename,x,y,width,height);  //解码 gif
            break;
        default:
            res=PIC_FORMAT_ERR;                             //非图片格式!!!
            break;
    }
    return res;
```

```
}
//动态分配内存
void *pic_memalloc (u32 size)
{
    return (void*)mymalloc(SRAMDTCM,size);
}
//释放内存
void pic_memfree (void* mf)
{
    myfree(SRAMDTCM,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。

接下来我们看看头文件 piclib.h 关键代码如下：

```
#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
{
    u32(*read_point)(u16,u16);   //u32 read_point(u16 x,u16 y)读点函数
    void(*draw_point)(u16,u16,u32);  //void draw_point(u16 x,u16 y,u32 color)画点函数
    void(*fill)(u16,u16,u16,u16,u32);
                //void fill(u16 sx,u16 sy,u16 ex,u16 ey,u32 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);
…//此处省略部分函数声明
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，相信大家很容易明白。最后我们看看 main.c 文件内容如下：

```
//得到 path 路径下,目标文件的总个数
```

```
//path:路径
//返回值:总有效文件数
u16 pic_get_tnum(u8 *path)
{
    u8 res;
    u16 rval=0;
    DIR tdir;              //临时目录
    FILINFO *tfileinfo;    //临时文件信息
    tfileinfo=(FILINFO*)mymalloc(SRAMIN,sizeof(FILINFO));//申请内存
    res=f_opendir(&tdir,(const TCHAR*)path);        //打开目录
    if(res==FR_OK&&tfileinfo)
    {
        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

            res=f_typetell((u8*)tfileinfo->fname);
            if((res&0XF0)==0X50)//取高四位,看看是不是图片文件
            {
                rval++;//有效文件数增加 1
            }
        }
    }
    myfree(SRAMIN,tfileinfo);//释放内存
    return rval;
}

int main(void)
{
    u8 res;
    DIR picdir;                //图片目录
    FILINFO *picfileinfo;  //文件信息
    u8 *pname;                 //带路径的文件名
    u16 totpicnum;             //图片文件总数
    u16 curindex;              //图片当前索引
    u8 key;                    //键值
    u8 pause=0;                //暂停标记
    u8 t;
    u16 temp;
    u32 *picoffsettbl;  //图片文件 offset 索引表

    Cache_Enable();                    //打开 L1-Cache
```

```
MPU_Memory_Protection();        //保护相关存储区域
HAL_Init();                     //初始化 HAL 库
Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
delay_init(216);                //延时初始化
uart_init(115200);              //串口初始化
usmart_dev.init(108);           //初始化 USMART
LED_Init();                     //初始化 LED
KEY_Init();                     //初始化按键
SDRAM_Init();                   //初始化 SDRAM
LCD_Init();                     //初始化 LCD
W25QXX_Init();                  //初始化 W25Q256
my_mem_init(SRAMIN);            //初始化内部内存池
my_mem_init(SRAMEX);            //初始化外部内存池
my_mem_init(SRAMDTCM);          //初始化 CCM 内存池
exfuns_init();                  //为 fatfs 相关变量申请内存
f_mount(fs[0],"0:",1);          //挂载 SD 卡
f_mount(fs[1],"1:",1);          //挂载 FLASH.
f_mount(fs[2],"2:",1);          //挂载 NAND 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,"阿波罗 STM32F4/F7 开发板",16,0);
Show_Str(30,70,200,16,"图片显示程序",16,0);
Show_Str(30,90,200,16,"KEY0:NEXT KEY2:PREV",16,0);
Show_Str(30,110,200,16,"KEY_UP:PAUSE",16,0);
Show_Str(30,130,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(30,150,200,16,"2016 年 7 月 15 日",16,0);
while(f_opendir(&picdir,"0:/PICTURE"))//打开图片文件夹
{
    Show_Str(30,170,240,16,"PICTURE 文件夹错误!",16,0);
    delay_ms(200);
    LCD_Fill(30,170,240,186,WHITE);//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_Toggle;//LED0 闪烁,提示程序正在运行.
                if(key==KEY2_PRES)          //上一张
                {
                    if(curindex)curindex--;
                    else curindex=totpicnum-1;
                    break;
                }else if(key==KEY0_PRES)//下一张
                {
                    curindex++;
                    if(curindex>=totpicnum)curindex=0;//到末尾的时候,自动从头开始
                    break;
                }else if(key==WKUP_PRES)
                {
                    pause=!pause;
                    LED1(!pause);       //暂停的时候 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 main 函数，还有一个 pic_get_tnum 的函数，用来得到 path 路径下，所有有效文件（图片文件）的个数。在 main 函数里面我们通过读/写偏移量（图片文件在 PICTURE 文件夹下的读/写偏移位置，可以看做是一个索引），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，来设置当前目录的偏移量（因为 f_readdir 只能沿着偏移位置一直往下找，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。

49.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 开始显示图片（假设 SD 卡及文件都准备好了，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 49.4.1 所示：

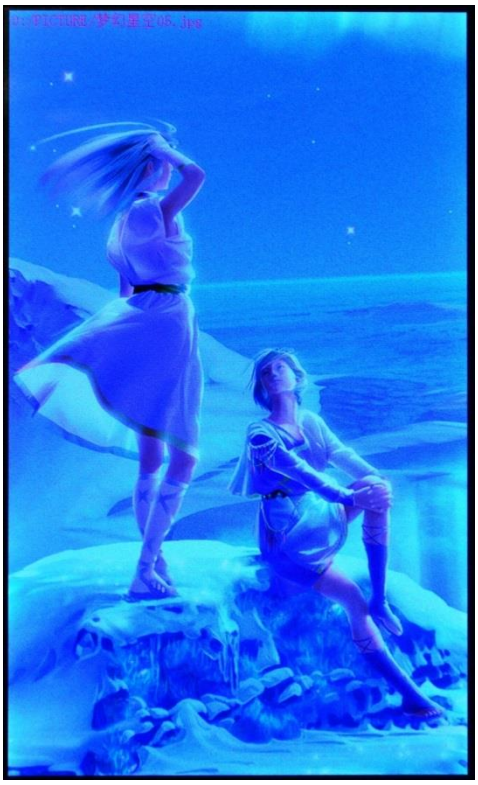


图 49.4.1 图片显示实验显示效果

按 KEY0 和 KEY2 可以快速切换到下一张或上一张，KEY_UP 按键可以暂停自动播放，同时 DS1 亮，指示处于暂停状态，再按一次 KEY_UP 则继续播放。同时，由于我们的代码支持 gif 格式的图片显示（注意尺寸不能超过 LCD 屏幕尺寸），所以可以放一些 gif 图片到 PICTURE 文件夹，来看动画了。

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。

第五十章 硬件 JPEG 解码实验

上一章，我们学习了图片解码，学会了使用软件解码显示 bmp/jpg/jpeg/gif 等格式的图片，但是软件解码速度都比较慢，本章我们将学习如何使用 STM32F767 自带的硬件 JPEG 编解码器，实现对 JPG/JPEG 图片的硬解码，从而大大提高解码速度。本章分为如下几个部分：

50.1 硬件 JPEG 编解码器简介
50.2 硬件设计
50.3 软件设计
50.4 下载验证

50.1 硬件 JPEG 编解码器简介

STM32F767 自带了硬件 JPEG 编解码器，可以实现快速 JPG/JPEG 编解码，本章我们仅使用 JPG/JPEG 解码器。STM32F7 的 JPEG 编解码器具有如下特点：

- 支持 JPEG 编码/解码
- 支持 24 位颜色深度（即 RGB888）
- 单周期解码/编码一个像素
- 支持 JPEG 头数据编解码
- 多达 4 个可编程量化表
- 完全可编程的哈弗曼表（AC 和 DC 各 2 个）
- 完全可编程的最小编码单元（MCU）
- 单周期哈弗曼编码/解码

STM32F7 的 JPEG 编解码器框图如图 50.1.1 所示：

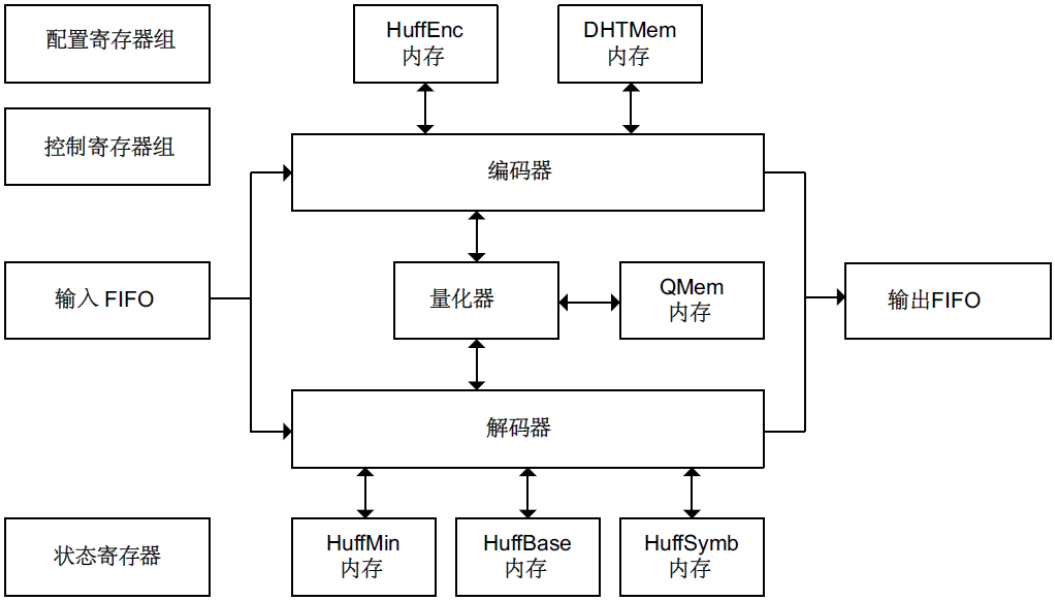


图 50.1.1 STM32F7 硬件 JPEG 编解码器框图

图 50.1.1 为 STM32F7 的硬件 JPEG 编解码器框图，我们只需要对相关寄存器进行设置，然后读写输入/输出 FIFO，即可完成 JPEG 的编解码。本章，我们只介绍如何利用 STM32F7 的硬件 JPEG 解码器实现对 JPG/JPEG 图片的解码。

硬件 JPEG 解码器，支持解码符合 ISO/IEC10918-1 协议规范的 JPEG 数据流，并且支持解码 JPEG 头（可配置），通过输入 FIFO 读取需要解码的 JPEG 数据，通过输出 FIFO 将解码完成

的 YUV 数据传输给外部。

注意：硬件 JPEG 解码器解码完成后是 YUV 格式的数据，并不是 RGB 格式的数据，所以不能直接显示到 LCD 上面，必须经过 YUV→RGB 的转换，才可以显示在 LCD 上面。

硬件 JPEG 解码时 FIFO 数据的处理（读取/写入）有两种方式：1，中断方式；2，DMA 方式。为了达到最快的解码速度，我们一般使用 DMA 来处理 FIFO 数据。接下来，我们介绍一下硬件 JPEG 解码的数据处理过程。

输入 FIFO DMA

通过设置 JPEG_CR 寄存器的 IDMAEN 位为 1，可以使能 JPEG 输入 FIFO 的 DMA，当输入 FIFO（总容量为 32 字节）至少半空的时候，将产生一个 DMA 请求，读取 16 字节数据到输入 FIFO。通过设置 IDMAEN 位为 0，可以暂停 FIFO 获取数据，这个操作在 DMA 传输完成，读取下一批 JPEG 数据的时候经常用到。

注意：在当前图片解码完成后，开启下一张图片解码之前，需要对输入 FIFO 进行一次清空（设置 JPEG_CR 寄存器的 IFF 位），否则上一张图片的数据会影响到下一张图片的解码。

输出 FIFO DMA

通过设置 JPEG_CR 寄存器的 ODMAEN 位为 1，可以使能 JPEG 输出 FIFO 的 DMA，当输出 FIFO（总容量为 32 字节）至少半满的时候，将产生一个 DMA 请求，可以从输出 FIFO 读取 16 字节数据。通过设置 ODMAEN 位为 0，可以暂停 FIFO 输出数据，这个操作在 DMA 传输完成，执行 YUV→RGB 转换的时候经常用到。

注意：当图片解码结束以后，输出 FIFO 里面可能还有数据，此时我们需要手动读取 FIFO 里面的数据，直到 JPEG_SR 寄存器的 OFNEF 位为 0。

JPEG 头解码

通过设置 JPEG_CONFR1 寄存器的 HDR 位为 1，可以使能 JPEG 头解码，通过设置 JPEG_CR 寄存器 HPDIE 位为 1，可以使能 JPEG 头解码完成中断。在完成 JPEG 头解码之后，我们可以获取当前 JPEG 图片的很多参数，包括：颜色空间、色度抽样、高度、宽度和 MCU 总数等信息。这些参数对我们后面的解码和颜色转换（YUV→RGB）非常重要。

硬件 JPEG 使用 DMA 实现 JPG/JPEG 图片解码的数据处理流程如图 50.1.2 所示：

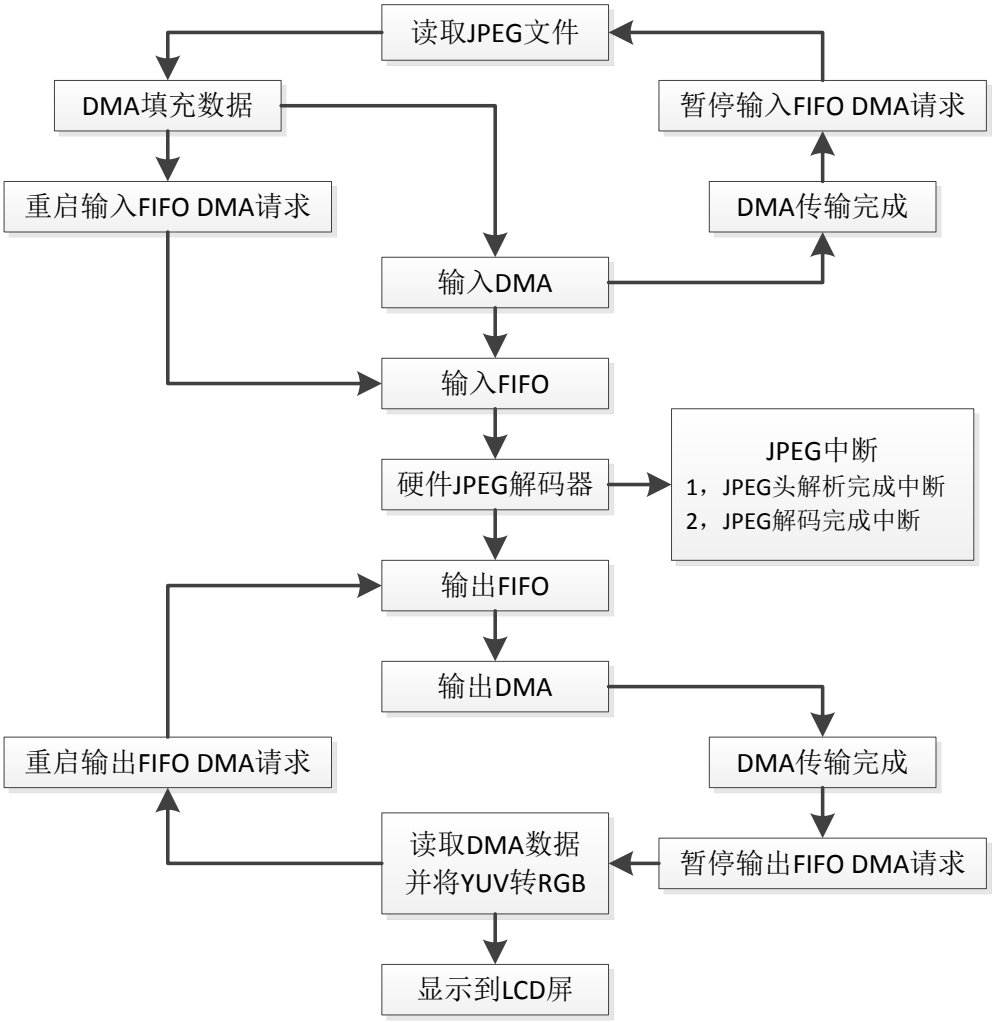


图 50.1.2 硬件 JPEG 解码数据处理流程（DMA 方式）

由图可知，数据处理主要由 2 个 DMA 完成：输入 DMA 和输出 DMA，分别处理硬件 JPEG 的输入 FIFO 和输出 FIFO 的数据。通过适当控制输入 FIFO/输出 FIFO 的暂停和重启，从而控制整个数据处理的进程，暂停 FIFO 的时间越少，解码速度就越快。

图中我们还用到了 2 个 JPEG 中断：JPEG 头解析完成中断和 JPEG 解码完成中断，他们共用一个中断服务函数。JPEG 头解析完成中断，在 JPEG 头解码完成后进入，此时我们可以获取 JPG/JPEG 图片的很多重要信息，方便后续解码。JPEG 解码完成中断，在 JPG/JPEG 图片解码完成后进入，标志着整张图片解码完成。

接下来，我们介绍本章需要用到的一些寄存器。

首先是 JPEG 内核控制寄存器：JPEG_CONFR0，该寄存器仅最低位（START 位）有效，设置该位为 1，可以启动 JPEG 解码流程。通过设置该位为 0，可以退出当前 JPEG 解码。

接下来，我们看 JPEG 配置寄存器 1：JPEG_CONFR1，该寄存器各位描述如图 50.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
YSIZE[15:0]															
rw															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	HDR	NS[1:0]		COLSPACE[1:0]		DE	Res.	NF[1:0]	

图 50.1.3 JPEG_CONFR1 寄存器各位描述

YSIZE[15:0]，定义 JPEG 图片的高度，读取该寄存器可以获得图片高度（注意：需要在 JPEG 头解析成功以后，才可以读取该寄存器获取图片高度，下同）。

HDR 位，用于设置是否使能 JPEG 头解码，我们一般设置为 1，使能 JPEG 头解码。

DE 位，用于设置硬件 JPEG 工作模式，我们设置为 1，表示使用 JPEG 解码模式。

NF[1:0]，这两个位用于定义色彩组成：00，表示灰度图片；01，未用到；10，表示 YUV/RGB；11 表示 CYMK。

接下来，我们看 JPEG 配置寄存器 3：JPEG_CONFR3，该寄存器各位描述如图 50.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
XSIZE[15:0]															
rw															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

图 50.1.4 JPEG_CONFR3 寄存器各位描述

该寄存器仅高 16 位（YSIZE[15:0]）有效，定义 JPEG 图片的宽度，读取该寄存器可以获得图片宽度。

另外，还有 JPEG 配置寄存器 4~7：JPEG_CONFR4~7，这四个寄存器 ST 官方数据手册对其解释也不是很清楚，但是我们可以参考 ST 官方提供的参考代码，知道这四个寄存器的 NB[3:0]位用来表示 YUV 的抽样方式（YUV422、YUV420、YUV444），详见本例程源码。

接下来，我们看 JPEG 控制寄存器：JPEG_CR，该寄存器各位描述如图 50.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	OFF	IFF	ODMAEN	IDMAEN	Res.	Res.	Res.	Res.	HPDIE	EOCIE	OFNEIE	OFTIE	IFNFIE	IFTIE	JCEN
	r0	r0	rw	rw					rw	rw	rw	rw	rw	rw	rw

图 50.1.5 JPEG_CR 寄存器各位描述

OFF 位，用于清空输出 FIFO，在启动新图片解码之前，需要对输出 FIFO 进行清空。

IFF 位，用于清空输入 FIFO，在启动新图片解码之前，需要对输入 FIFO 进行清空。

ODMAEN 位，用于使能输出 FIFO 的 DMA，我们设置此位为 1。

IDMAEN 位，用于使能输入 FIFO 的 DMA，我们设置此位为 1。

HPDIE 位，用于使能 JPEG 头解码完成中断，我们设置为 1，使能 JPEG 头解码完成中断，在中断服务函数里面读取 JPEG 的相关信息（长宽、颜色空间、色度抽样等），并根据色度抽样方式，获取对应的 YUV→RGB 转换函数。

EOCIE 位，用于使能 JPEG 解码完成中断，我们设置为 1，使能 JPEG 解码完成中断，在中断服务函数里面标记 JPEG 解码完成，以便结束 JPEG 解码流程。

JCEN 位，用于使能硬件 JPEG 内核，我们必须设置此位为 1，以启动硬件 JPEG 内核。

接下来，我们看 JPEG 状态寄存器：JPEG_SR，该寄存器各位描述如图 50.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	COF	HPDF	EOCF	OFNEF	OFTF	IFNFF	IFTF	Res.
								ro	ro	ro	ro	ro	ro	ro	

图 50.1.6 JPEG_SR 寄存器各位描述

HPDF 位，表示 JPEG 头解码完成的标志，当该位为 1 时，表示 JPEG 头解析成功，我们可以读取相关寄存器，获取 JPEG 图片的长宽、颜色空间和色度抽样等重要信息。向 JPEG_FCR 寄存器的 CHPDF 位写 1，可以清零此位。

EOCF 位，表示 JPEG 解码完成的标志，当该位为 1 时，表示一张 JPEG 图像解码完成。此时我们可以从输出 FIFO 读取最后的数据。向 JPEG_FCR 寄存器的 CEOCF 位写 1，可以清零此位。

接下来，我们看 JPEG 标志清零寄存器：JPEG_FCR，该寄存器各位描述如图 50.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CHPDF	CEOCF	Res.	Res.	Res.	Res.	Res.
									w1c	w1c	Res.	Res.	Res.	Res.	Res.

图 50.1.7 JPEG_FCR 寄存器各位描述

该寄存器，仅两位有效：CHPDF 位和 CEOCF 位，向这两个位写入 1，可以分别清除 JPEG_SR 寄存器的 HPDF 和 EOCF 位。

最后，还有 JPEG 数据输入寄存器（JPEG_DIR）和 JPEG 数据输出寄存器（JPEG_DOR），这两个寄存器都是 32 位有效，前者用于往输入 FIFO 写入数据。后者用于读取输出 FIFO 的数据。

至此，本实验所需要用到的相关寄存器，就全部介绍完了，更详细的介绍，请参考《STM32F7xx 参考手册》 21.5 节。

接下来，我们看看在 DMA 模式下，使用 STM32F7 的硬件 JPEG 解码 JPG/JPEG 的简要步骤，HAL 库中硬件 JPEG 解码函数分布在 stm32f7xx_hal_jpeg.c 和头文件 stm32f7xx_hal_jpeg.h 中。

1）初始化硬件 JPEG 内核。

首先，我们通过设置 AHB2ENR 的 bit1 位为 1，使能硬件 JPEG 内核时钟，然后通过 JPEG_CR 寄存器的 JCEN 位，使能硬件 JPEG。通过清零 JPEG_CONFR0 寄存器的 START 位，停止 JPEG 编解码进程。通过设置 JPEG_CONFR1 寄存器的 HDR 位，使能 JPEG 头解码。最后设置 JPEG 中断服务函数的中断优先级，完成初始化硬件 JPEG 内核过程。

在 HAL 库中，初始化 JPEG 是通过函数 HAL_JPEG_Init 来实现的，该函数声明如下：

```
HAL_StatusTypeDef HAL_JPEG_Init(JPEG_HandleTypeDef *hjpeg);
```

该函数的使用方法大家可以参考我们实验源码即可。

JPEG 时钟使能方法：

```
__HAL_RCC_JPEG_CLK_ENABLE();                    //使能 JPEG 时钟
```

和其他外设一样，HAL 库也提供了硬件 JPEG 初始化回调函数，声明如下：

```
void HAL_JPEG_MspInit(JPEG_HandleTypeDef *hjpeg);
```

一般情况下，时钟使能，中断优先级设置都放在回调函数中。

2）初始化硬件 JPEG 解码。

在初始化硬件 JPEG 内核以后，我们配置 JPEG 内核工作在 JPEG 解码模式。通过设置 JPEG_CONFR1 寄存器的 DE 位，使能 JPEG 解码模式。然后设置 JPEG_CR 寄存器的 OFF、IFF、HPDIE、EOCIE 等位，清空输出/输入 FIFO，并开启 JPEG 头解码完成和 JPEG 解码完成中断。最后，设置 JPEG_CONFR0 寄存器的 START 位，启动 JPEG 解码进程。操作过程如下：

```
JPEG->CONFR1|=JPEG_CONFR1_DE;    //使能硬件 JPEG 解码模式
__HAL_JPEG_ENABLE_IT(&JPEG_Handler,JPEG_IT_HPD);//使能 Header 解码完中断
__HAL_JPEG_ENABLE_IT(&JPEG_Handler,JPEG_IT_EOC);//使能解码完成中断
JPEG->CONFR0|=JPEG_CONFR0_START;                //使能 JPEG 编解码进程
```

注意：此时我并未开启 JPEG 的输入和输出 DMA，只要我们不往输入 FIFO 写入数据，JPEG 内核就一直处于等待数据输入状态。

3）配置硬件 JPEG 输入输出 DMA。

这一步，我们将配置 JPEG 的输入 DMA 和输出 DMA，分别负责 JPEG 输入 FIFO 和输出 FIFO 的数据传输。对于输入 DMA，目标地址为 JPEG_DIR 寄存器地址，源地址为一片内存区域，利用输入 DMA 实现 JPEG 输入 FIFO 数据的自动填充。对于输出 DMA，目标地址为一片内存区域，源地址为 JPEG_DOR 寄存器地址，利用输出 DMA 实现 JPEG 输出 FIFO 数据自动搬运到对应内存区域。对于输入 DMA 和输出 DMA，我们都需要开启传输完成中断，并设置相关中断服务函数。在传输完成中断里面，实现对输入输出数据的处理。

对于 JPEG 输入输出 DMA 配置，我们主要调用 HAL 库函数 HAL_DMA_Init 处理即可，具体的配置方法请参考 50.3 小节实验源码讲解。

```
HAL_StatusTypeDef HAL_DMA_Init(DMA_HandleTypeDef *hdma);
```

4）编写相关中断服务函数，启动 DMA。

我们总共开启了 4 个中断：JPEG 头解码完成中断、JPEG 解码完成中断、输入 DMA 传输完成中断和输出 DMA 传输完成中断。前两个中断共用一个中断服务函数，所以我们总共需要编写 3 个中断服务函数。另外，我们采用回调函数的方式，对数据进行处理，总共需要编写 4 个回调函数，分别对应 4 个中断产生时的数据处理。在配置完这些以后，启动 DMA，并通过设置 JPEG_CR 寄存器的 IDMAEN 和 ODMAEN 位，开启 JPEG 的输入和输出 FIFO DMA 请求，开始执行 JPEG 解码。

5）处理 JPEG 数据输出数据，执行 YUV→RGB 转换，并送 LCD 显示。

最后，在主循环里面，根据输入 DMA 和输出 DMA 的数据处理情况，持续从源文件读取 JPEG 数据流，并将硬件 JPEG 解码完成的 YUV 数据流转换成 RGB 格式。最后，在完成一张 JPEG 解码之后，将 RGB 数据直接一次性显示到 LCD 屏幕上，实现图片显示。

50.2 硬件设计

本章实验功能简介：本实验开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY2 可以快速浏览下一张和上一张，KEY_UP 按键用于暂停/继续播放，DS1 用于指示当前是否处于暂停状态。如果未找到 PICTURE 文件夹/任何图片文件，则提示错误。同样我们也是用 DS0 来指示程序正在运行。

本实验也可以通过 USMART 调用 ai_load_picfile 和 minibmp_decode 解码任意指定路径的图片。

注意：本例程的实验现象，同上一章(图片显示实验)完全一模一样，唯一的区别就是 JPEG 解码速度（要求图片分辨率小于等于 LCD 分辨率）变快了很多。STM32F7 的硬件 JPEG 解码性能可以在最快 40ms 内完成一张 800*480 的 JPEG 图片解码(读数据+解码+YUV→RGB 转换，但是不包括显示)。

所要用到的硬件资源如下：

1） 指示灯 DS0 和 DS1

2） KEY0、KEY2 和 KEY_UP 三个按键

3） 串口

4） LCD 模块

5） SD 卡

6） SPI FLASH

7） 硬件 JPEG 解码器

前面 6 个部分，在之前的实例中都介绍过了，我们在此就不介绍了，最后的硬件 JPEG 解码器，完全是 STM32F7 的内部资源，不需要在开发板上做任何操作，只需要软件配置即可。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。

50.3 软件设计

打开本章实验工程目录可以看到，首先在 HARDWARE 文件夹所在的文件夹下新建一个 JPEGCODEC 文件夹，并新建 jpeg_utils.c、jpeg_utils.h、jpeg_utils_tbl.h、jpegcodec.c 和 jpegcodec.h 等 5 个文件。并将 JPEGCODEC 文件夹加入头文件包含路径。其中 jpeg_utils.c、jpeg_utils.h 和 jpeg_utils_tbl.h 等三个文件，实现了 YUV→RGB 的转换，支持 YUV420、YUV422、YUV444、灰度和 CMYK 到 RGB565、RGB888 和 ARGB8888 的转换。这几个文件是我们移植 ST 官方 JPEG 解码例程相关代码而来，并作出适当修改，以获得最快的转换速度。jpegcodec.c 和 jpegcodec.h 是硬件 JPEG 解码的底层驱动代码。然后在 PICTURE 文件夹下新建 hjpgd.c 和 hjpgd.h，用于实现 JPG/JPEG 图片的硬件 JPEG 解码。

从工程界面可以看到，我们将 jpeg_utils.c 和 jpegcodec.c 加入 HARDWARE 组下，并将 hjpgd.c 加入 PICTURE 组下。由于篇幅所限，我们就不把所有代码都贴出来了，仅列出一些重要的函数给大家讲解。

首先，看 jpegcodec.c 文件里面，比较重要的函数代码如下：

```
void (*jpeg_in_callback)(void);          //JPEG DMA 输入回调函数
void (*jpeg_out_callback)(void);         //JPEG DMA 输出  回调函数
void (*jpeg_eoc_callback)(void);         //JPEG  解码完成  回调函数
void (*jpeg_hdp_callback)(void);         //JPEG Header 解码完成  回调函数
//DMA2_Stream0 中断服务函数
//处理硬件 JPEG 解码时输入的数据流
void DMA2_Stream0_IRQHandler(void)
{
    if(__HAL_DMA_GET_FLAG(&JPEGDMAIN_Handler,DMA_FLAG_TCIF0_4) \
                                                !=RESET) //DMA 传输完成
    {
        __HAL_DMA_CLEAR_FLAG(&JPEGDMAIN_Handler,DMA_FLAG_TCIF0_4);
                                                //清除 DMA 传输完成中断标志位
```

```
            JPEG->CR&=~(1<<11);             //关闭 JPEG 的 DMA IN

            __HAL_JPEG_DISABLE_IT(&JPEG_Handler,JPEG_IT_IFT|JPEG_IT_IFNF| \
                        JPEG_IT_OFT|JPEG_IT_OFNE|JPEG_IT_EOC|JPEG_IT_HPD);
                                                //关闭 JPEG 中断,防止被打断.
            if(jpeg_in_callback!=NULL)jpeg_in_callback();           //执行回调函数
            __HAL_JPEG_ENABLE_IT(&JPEG_Handler,JPEG_IT_EOC| \
                                    JPEG_IT_HPD);     //使能 EOC 和 HPD 中断.
        }
    }
    //DMA2_Stream1 中断服务函数
    //处理硬件 JPEG 解码后输出的数据流
    void DMA2_Stream1_IRQHandler(void)
    {
        if(__HAL_DMA_GET_FLAG(&JPEGDMAOUT_Handler,DMA_FLAG_TCIF1_5) \
                                                    !=RESET) //DMA 传输完成
        {
            __HAL_DMA_CLEAR_FLAG(&JPEGDMAOUT_Handler,DMA_FLAG_TCIF1_5);
                                                //清除 DMA 传输完成中断标志位
            JPEG->CR&=~(1<<12);                 //关闭 JPEG 的 DMA OUT
            __HAL_JPEG_DISABLE_IT(&JPEG_Handler,JPEG_IT_IFT|JPEG_IT_IFNF|\
              JPEG_IT_OFT|JPEG_IT_OFNE|JPEG_IT_EOC|JPEG_IT_HPD);//关闭 JPEG 中断,
            if(jpeg_out_callback!=NULL)jpeg_out_callback();//执行回调函数
            __HAL_JPEG_ENABLE_IT(&JPEG_Handler,JPEG_IT_EOC|JPEG_IT_HPD);
                                                    //使能 EOC 和 HPD 中断.
        }
    }
    //JPEG 解码中断服务函数
    void JPEG_IRQHandler(void)
    {
        if(__HAL_JPEG_GET_FLAG(&JPEG_Handler,JPEG_FLAG_HPDF)!=RESET)
                                                    //JPEG Header 解码完成
        {
            jpeg_hdp_callback();
            __HAL_JPEG_DISABLE_IT(&JPEG_Handler,JPEG_IT_HPD);
                                                //禁止 Jpeg Header 解码完成中断
            __HAL_JPEG_CLEAR_FLAG(&JPEG_Handler,JPEG_FLAG_HPDF);
                                                //清除 HPDF 位(header 解码完成位)
        }
        if(__HAL_JPEG_GET_FLAG(&JPEG_Handler,JPEG_FLAG_EOCF)!=RESET)
                                                    //JPEG 解码完成
        {
            JPEG_DMA_Stop();
```

```
            jpeg_eoc_callback();
            __HAL_JPEG_CLEAR_FLAG(&JPEG_Handler,JPEG_FLAG_EOCF);
                                                    //清除 EOC 位(解码完成位)
            __HAL_DMA_DISABLE(&JPEGDMAIN_Handler); //关闭 JPEG 数据输入 DMA
            __HAL_DMA_DISABLE(&JPEGDMAOUT_Handler);//关闭 JPEG 数据输出 DMA
        }
    }
    //初始化硬件 JPEG 内核
    //tjpeg:jpeg 编解码控制结构体
    //返回值:0,成功;
    //       其他,失败
    u8 JPEG_Core_Init(jpeg_codec_typedef *tjpeg)
    {
        u8 i;
        JPEG_Handler.Instance=JPEG;
        HAL_JPEG_Init(&JPEG_Handler);     //初始化 JPEG

        for(i=0;i<JPEG_DMA_INBUF_NB;i++)
        {
            tjpeg->inbuf[i].buf=mymalloc(SRAMIN,JPEG_DMA_INBUF_LEN);
            if(tjpeg->inbuf[i].buf==NULL)
            {
                JPEG_Core_Destroy(tjpeg);
                return 1;
            }
        }
        for(i=0;i<JPEG_DMA_OUTBUF_NB;i++)
        {
            tjpeg->outbuf[i].buf=mymalloc(SRAMIN,JPEG_DMA_OUTBUF_LEN+32);
                                                    //有可能会多需要 32 字节内存
            if(tjpeg->outbuf[i].buf==NULL)
            {
                JPEG_Core_Destroy(tjpeg);
                return 1;
            }
        }
        return 0;
    }
    //关闭硬件 JPEG 内核,并释放内存
    //tjpeg:jpeg 编解码控制结构体
    void JPEG_Core_Destroy(jpeg_codec_typedef *tjpeg)
    {
        u8 i;
```

```
    JPEG_DMA_Stop();//停止 DMA 传输
    for(i=0;i<JPEG_DMA_INBUF_NB;i++)myfree(SRAMIN,tjpeg->inbuf[i].buf);
    for(i=0;i<JPEG_DMA_OUTBUF_NB;i++)myfree(SRAMIN,tjpeg->outbuf[i].buf);
}
//初始化硬件 JPEG 解码器
//tjpeg:jpeg 编解码控制结构体
void JPEG_Decode_Init(jpeg_codec_typedef *tjpeg)
{
    u8 i;
    tjpeg->inbuf_read_ptr=0;
    tjpeg->inbuf_write_ptr=0;
    tjpeg->indma_pause=0;
    tjpeg->outbuf_read_ptr=0;
    tjpeg->outbuf_write_ptr=0;
    tjpeg->outdma_pause=0;
    tjpeg->state=JPEG_STATE_NOHEADER;  //图片解码结束标志
    tjpeg->blkindex=0;                 //当前 MCU 编号
    tjpeg->total_blks=0;               //总 MCU 数目
    for(i=0;i<JPEG_DMA_INBUF_NB;i++)
    {
        tjpeg->inbuf[i].sta=0;
        tjpeg->inbuf[i].size=0;
    }
    for(i=0;i<JPEG_DMA_OUTBUF_NB;i++)
    {
        tjpeg->outbuf[i].sta=0;
        tjpeg->outbuf[i].size=0;
    }
    JPEG->CONFR1|=JPEG_CONFR1_DE;     //使能硬件 JPEG 解码模式
    __HAL_JPEG_ENABLE_IT(&JPEG_Handler,JPEG_IT_HPD);//使能 Header 解码完中断
    __HAL_JPEG_ENABLE_IT(&JPEG_Handler,JPEG_IT_EOC);//使能解码完成中断

    JPEG->CONFR0|=JPEG_CONFR0_START;//使能 JPEG 编解码进程
}
//启动 JPEG DMA 解码过程
void JPEG_DMA_Start(void)
{
    __HAL_DMA_ENABLE(&JPEGDMAIN_Handler);  //打开 JPEG 数据输入 DMA
    __HAL_DMA_ENABLE(&JPEGDMAOUT_Handler);     //打开 JPEG 数据输出 DMA
    JPEG->CR|=3<<11;
}
//停止 JPEG DMA 解码过程
void JPEG_DMA_Stop(void)
```

```
{
    JPEG->CR&=~(3<<11);                                 //JPEG IN&OUT DMA 禁止
    JPEG->CONFR0&=~(1<<0);                              //停止 JPEG 编解码进程
    __HAL_JPEG_DISABLE_IT(&JPEG_Handler,JPEG_IT_IFT|JPEG_IT_IFNF| \
    JPEG_IT_OFT|JPEG_IT_OFNE|JPEG_IT_EOC|JPEG_IT_HPD);    //关闭所有中断
    JPEG->CFR=3<<5;                                     //清空标志
}
//暂停 DMA IN 过程
void JPEG_IN_DMA_Pause(void)
{
    JPEG->CR&=~(1<<11);             //暂停 JPEG 的 DMA IN
}
//恢复 DMA IN 过程
//memaddr:存储区首地址
//memlen:要传输数据长度(以字节为单位)
void JPEG_IN_DMA_Resume(u32 memaddr,u32 memlen)
{
    if(memlen%4)memlen+=4-memlen%4;//扩展到 4 的倍数
    memlen/=4;                                  //除以 4
    DMA2->LIFCR|=0X3D<<6*0;             //清空通道 0 上所有中断标志
    DMA2_Stream0->M0AR=memaddr;     //设置存储器地址
    DMA2_Stream0->NDTR=memlen;      //传输长度为 memlen
    DMA2_Stream0->CR|=1<<0;             //开启 DMA2,Stream0
    JPEG->CR|=1<<11;                    //恢复 JPEG DMA IN
}
//暂停 DMA OUT 过程
void JPEG_OUT_DMA_Pause(void)
{
    JPEG->CR&=~(1<<12);             //暂停 JPEG 的 DMA OUT
}
//恢复 DMA OUT 过程
//memaddr:存储区首地址
//memlen:要传输数据长度(以字节为单位)
void JPEG_OUT_DMA_Resume(u32 memaddr,u32 memlen)
{
    if(memlen%4)memlen+=4-memlen%4;//扩展到 4 的倍数
    memlen/=4;                                  //除以 4
    DMA2->LIFCR|=0X3D<<6*1;             //清空通道 1 上所有中断标志
    DMA2_Stream1->M0AR=memaddr;     //设置存储器地址
    DMA2_Stream1->NDTR=memlen;      //传输长度为 memlen
    DMA2_Stream1->CR|=1<<0;             //开启 DMA2,Stream1
    JPEG->CR|=1<<12;                    //恢复 JPEG DMA OUT
}
```

```
//获取图像信息
//tjpeg:jpeg 解码结构体
void JPEG_Get_Info(jpeg_codec_typedef *tjpeg)
{
    u32 yblockNb,cBblockNb,cRblockNb;
    switch(JPEG->CONFR1&0X03)
    {
        case 0:tjpeg->Conf.ColorSpace=JPEG_GRAYSCALE_COLORSPACE; break;
        case 2:tjpeg->Conf.ColorSpace=JPEG_YCBCR_COLORSPACE;break;
        case 3:tjpeg->Conf.ColorSpace=JPEG_CMYK_COLORSPACE;break;
    }
    tjpeg->Conf.ImageHeight=(JPEG->CONFR1&0XFFFF0000)>>16;  //获得图像高度
    tjpeg->Conf.ImageWidth=(JPEG->CONFR3&0XFFFF0000)>>16;  //获得图像宽度
    if((tjpeg->Conf.ColorSpace==JPEG_YCBCR_COLORSPACE)||
       (tjpeg->Conf.ColorSpace==JPEG_CMYK_COLORSPACE))
    {
        yblockNb   =(JPEG->CONFR4&(0XF<<4))>>4;
        cBblockNb =(JPEG->CONFR5&(0XF<<4))>>4;
        cRblockNb =(JPEG->CONFR6&(0XF<<4))>>4;
        if((yblockNb==1)&&(cBblockNb==0)&&(cRblockNb==0))
            tjpeg->Conf.ChromaSubsampling=JPEG_422_SUBSAMPLING; //16x8 block
        else if((yblockNb==0)&&(cBblockNb==0)&&(cRblockNb==0))
            tjpeg->Conf.ChromaSubsampling=JPEG_444_SUBSAMPLING;
        else if((yblockNb==3)&&(cBblockNb==0)&&(cRblockNb==0))
            tjpeg->Conf.ChromaSubsampling = JPEG_420_SUBSAMPLING;
        else tjpeg->Conf.ChromaSubsampling=JPEG_444_SUBSAMPLING;
    }else tjpeg->Conf.ChromaSubsampling=JPEG_444_SUBSAMPLING;//默认用 4:4:4
    tjpeg->Conf.ImageQuality=0;//图像质量参数在最后才可获取，先设置为 0
}
```

这里，我们总共列出了 13 个函数。接下来，我们简单介绍一下这些函数。

DMA2_Stream0_IRQHandler 中断服务函数，用于处理 JPEG 解码时输入 FIFO 的数据，当输入 DMA 传输完成时，会进入该函数，我们通过 jpeg_in_callback 回调函数（该函数在后面再做介绍），处理输入 DMA 传输完成事务。

DMA2_Stream1_IRQHandler 中断服务函数，用于处理 JPEG 解码时输出 FIFO 的数据，当输出 DMA 传输完成时，会进入该函数，我们通过 jpeg_out_callback 回调函数（该函数在后面再做介绍），处理输出 DMA 传输完成事务。

JPEG_IRQHandler 中断服务函数，根据 JPEG_SR 的状态标志位，分别处理 JPEG 头解码完成中断和 JPEG 文件解码完成中断。当 JPEG 头解码完成时，调用 jpeg_hdp_callback 回调函数处理相关事务。当 JPEG 文件解码完成时，调用 jpeg_eoc_callback 回调函数处理相关事务，同时停止 DMA 传输。

JPEG_Core_Init 函数，初始化硬件 JPEG 内核。在该函数里面，有对 tjpeg->inbuf[i].buf 和 tjpeg->outbuf[i].buf 两个数组申请内存。tjpeg 是我们在 jpegcodec.h 里面定义的一个结构体，用于控制整个 JPEG 解码，该结构体定义如下：

```
//JPEG 数据缓冲结构体
typedef struct
{
    u8 sta;             //状态:0,无数据;1,有数据.
    u8 *buf;            //JPEG 数据缓冲区
    u16 size;           //JPEG 数据长度
}jpeg_databuf_type;
//jpeg 编解码控制结构体
typedef struct
{
    JPEG_ConfTypeDef    Conf;                           //当前 JPEG 文件相关参数
    jpeg_databuf_type inbuf[JPEG_DMA_INBUF_NB];         //DMA IN buf
    jpeg_databuf_type outbuf[JPEG_DMA_OUTBUF_NB];       //DMA OUT buf
    vu8 inbuf_read_ptr;                                 //DMA IN buf 当前读取位置
    vu8 inbuf_write_ptr;                                //DMA IN buf 当前写入位置
    vu8 indma_pause;                                    //输入 DMA 暂停状态标识
    vu8 outbuf_read_ptr;                                //DMA OUT buf 当前读取位置
    vu8 outbuf_write_ptr;                               //DMA OUT buf 当前写入位置
    vu8 outdma_pause;                                   //输入 DMA 暂停状态标识
    vu8 state;                                          //解码状态：0,未识别 Header;
                                                        //1,识别到 Header; 2,解码完成;
    u32 blkindex;                                       //当前 block 编号
    u32 total_blks;                                     //jpeg 文件总 block 数
    u32 (*ycbcr2rgb)(u8 *,u8 *,u32 ,u32);               //颜色转换函数指针,原型请参考:
                                                        //JPEG_YCbCrToRGB_Convert_Function
}jpeg_codec_typedef;
```

其中 inbuf 和 outbuf，分表代表输入 DMA FIFO 和输出 DMA FIFO，使用 FIFO 来处理 DMA 数据，可以提高读写效率。注意：这里的输入 DMA FIFO 和输出 DMA FIFO 同 JPEG 的输入 FIFO 和输出 FIFO 是不一样的，要注意区分。通过 JPEG_DMA_INBUF_NB 和 JPEG_DMA_OUTBUF_NB 宏定义，我们可以修改输入 DMA FIFO 和输出 DMA FIFO 的深度。

另外，还有输入输出 DMA FIFO 的读写位置、暂停状态、解码状态、当前 MCU block 编号、总 MCU block 数和颜色转换函数指针等参数。该结构体里面的 JPEG_ConfTypeDef 结构体定义，是在 jpeg_utils.h 里面定义的，该结构体定义如下：

```
//JPEG 文件信息结构体
typedef struct
{
    u8  ColorSpace;         //图像的颜色空间: gray-scale/YCBCR/RGB/CMYK
    u8  ChromaSubsampling;  //YCBCR/CMYK 颜色空间的色度抽样情况:
                            //0：4:4:4；  1：4:2:2；  2：4:1:1；  3：4:2:0
    u32 ImageHeight;        //图像高度
    u32 ImageWidth;         //图像宽度
    u8  ImageQuality;       //图像编码质量:1~100
}JPEG_ConfTypeDef;
```

JPEG_Core_Destroy 函数，用于关闭 JPEG 处理（停止 DMA 传输），并释放内存。

JPEG_Decode_Init 函数，用于初始化硬件 JPEG 解码器，同时对输入 DMA FIFO 和输出 DMA FIFO 的相关标记进行清理处理，以便开始 JPEG 解码。

JPEG_DMA_Start 和 JPEG_DMA_Stop 函数，分别用于启动和关闭 JPEG DMA 解码。

JPEG_IN_DMA_Pause 和 JPEG_IN_DMA_Resume 函数，分别用于暂停和重启输入 DMA。

JPEG_OUT_DMA_Pause 和 JPEG_OUT_DMA_Resume 函数，分别用于暂停和重启输出 DMA。

JPEG_Get_Info 函数，用于获取 JPEG 图像信息，在 JPEG 头解码完成后，被调用。该函数可以获取 JPEG 图片的宽度、高度、颜色空间和色度抽样等重要信息。

接下来，我们看 jpeg_utils.c 文件，该文件移植自 ST 官方的硬件 JPEG 解码代码，该文件我们仅介绍 JPEG_GetDecodeColorConvertFunc 函数，该函数代码如下：

```
//获取 YCbCr 到 RGB 颜色转换函数和总的 MCU Block 数目.
//pJpegInfo:JPEG 文件信息结构体
//pFunction:JPEG_YCbCrToRGB_Convert_Function 的函数指针,根据 jpeg 图像参数,指向
//不同的颜色转换函数.
//ImageNbMCUs:总的 MCU 块数目
//返回值:0,正常;
//       1,失败;
u8 JPEG_GetDecodeColorConvertFunc(JPEG_ConfTypeDef *pJpegInfo,
        JPEG_YCbCrToRGB_Convert_Function *pFunction, u32 *ImageNbMCUs)
{
    u32 hMCU, vMCU;
    JPEG_ConvertorParams.ColorSpace=pJpegInfo->ColorSpace;          //色彩空间
    JPEG_ConvertorParams.ImageWidth=pJpegInfo->ImageWidth;          //图像宽度
    JPEG_ConvertorParams.ImageHeight=pJpegInfo->ImageHeight;        //图像高度
    JPEG_ConvertorParams.ImageSize_Bytes=pJpegInfo->ImageWidth*pJpegInfo->
        ImageHeight*JPEG_BYTES_PER_PIXEL; //转换后的图像总字节数
    JPEG_ConvertorParams.ChromaSubsampling=pJpegInfo->ChromaSubsampling;//抽样
    if(JPEG_ConvertorParams.ColorSpace==JPEG_YCBCR_COLORSPACE)//YCbCr420
    {
        if(JPEG_ConvertorParams.ChromaSubsampling==JPEG_420_SUBSAMPLING)
        {
            *pFunction=JPEG_MCU_YCbCr420_ARGB_ConvertBlocks;//YCbCr420
           JPEG_ConvertorParams.LineOffset=JPEG_ConvertorParams.ImageWidth%16;
            if(JPEG_ConvertorParams.LineOffset!=0)
            {
              JPEG_ConvertorParams.LineOffset=16-JPEG_ConvertorParams.LineOffset;
            }
            JPEG_ConvertorParams.H_factor=16;
            JPEG_ConvertorParams.V_factor=16;
        }else if(JPEG_ConvertorParams.ChromaSubsampling==
                JPEG_422_SUBSAMPLING)//YCbCr422
        {
```

```
            *pFunction=JPEG_MCU_YCbCr422_ARGB_ConvertBlocks;//YCbCr422
        JPEG_ConvertorParams.LineOffset=JPEG_ConvertorParams.ImageWidth%16;
            if(JPEG_ConvertorParams.LineOffset!=0)
            {
              JPEG_ConvertorParams.LineOffset=16-JPEG_ConvertorParams.LineOffset;
            }
            JPEG_ConvertorParams.H_factor=16;
            JPEG_ConvertorParams.V_factor=8;
        }else       // YCbCr444
        {
            *pFunction=JPEG_MCU_YCbCr444_ARGB_ConvertBlocks;//YCbCr444
        JPEG_ConvertorParams.LineOffset=JPEG_ConvertorParams.ImageWidth%8;
            if(JPEG_ConvertorParams.LineOffset!=0)
            {
               JPEG_ConvertorParams.LineOffset=8-JPEG_ConvertorParams.LineOffset;
            }
            JPEG_ConvertorParams.H_factor=8;
            JPEG_ConvertorParams.V_factor=8;
        }
    }else if(JPEG_ConvertorParams.ColorSpace==
              JPEG_GRAYSCALE_COLORSPACE)//GrayScale 颜色空间
    {
        *pFunction=JPEG_MCU_Gray_ARGB_ConvertBlocks;//使用 Y Gray 转换
        JPEG_ConvertorParams.LineOffset=JPEG_ConvertorParams.ImageWidth%8;
        if(JPEG_ConvertorParams.LineOffset!=0)
        {
            JPEG_ConvertorParams.LineOffset=8-JPEG_ConvertorParams.LineOffset;
        }
        JPEG_ConvertorParams.H_factor=8;
        JPEG_ConvertorParams.V_factor=8;
    }else if(JPEG_ConvertorParams.ColorSpace==JPEG_CMYK_COLORSPACE)
    {
        *pFunction=JPEG_MCU_YCCK_ARGB_ConvertBlocks;//使用 CMYK 颜色转换
        JPEG_ConvertorParams.LineOffset=JPEG_ConvertorParams.ImageWidth%8;
        if(JPEG_ConvertorParams.LineOffset!=0)
        {
            JPEG_ConvertorParams.LineOffset=8-JPEG_ConvertorParams.LineOffset;
        }
        JPEG_ConvertorParams.H_factor=8;
        JPEG_ConvertorParams.V_factor=8;
    }else return 0X01;  //不支持的颜色空间
    JPEG_ConvertorParams.WidthExtend=JPEG_ConvertorParams.ImageWidth+
                                     JPEG_ConvertorParams.LineOffset;
```

```
        JPEG_ConvertorParams.ScaledWidth=JPEG_BYTES_PER_PIXEL*
                                        JPEG_ConvertorParams.ImageWidth;
        hMCU=(JPEG_ConvertorParams.ImageWidth/JPEG_ConvertorParams.H_factor);
        if((JPEG_ConvertorParams.ImageWidth%JPEG_ConvertorParams.H_factor)!=0)
            hMCU++;    //+1 for horizenatl incomplete MCU
        vMCU=(JPEG_ConvertorParams.ImageHeight/JPEG_ConvertorParams.V_factor);
        if((JPEG_ConvertorParams.ImageHeight%JPEG_ConvertorParams.V_factor)!=0)
            vMCU++;    //+1 for vertical incomplete MCU
        JPEG_ConvertorParams.MCU_Total_Nb=(hMCU*vMCU);
        *ImageNbMCUs=JPEG_ConvertorParams.MCU_Total_Nb;
        return 0X00;
    }
```

该函数参数有 3 个：pJpegInfo 是一个 JPEG_ConfTypeDef 结构体指针，用于传递当前 JPEG 的相关参数；pFunction 是函数指针，类型为：JPEG_YCbCrToRGB_Convert_Function，定义如下：

```
typedef u32 (* JPEG_YCbCrToRGB_Convert_Function)(u8 *pInBuffer,u8 *pOutBuffer,u32 BlockIndex,u32 DataCount);
```

相关参数说明：

pInBuffer：指向输入的 YCbCr blocks 缓冲区

pOutBuffer：指向输出的 RGB888/ARGB8888 帧缓冲区

BlockIndex：输入 buf 里面的第一个 MCU 块编号

DataCount：输入缓冲区的大小

该函数指针可以指向在 jpeg_utils.c 里面定义的其他 5 个函数：

1，JPEG_MCU_YCbCr420_ARGB_ConvertBlocks 函数，实现 YUV420→RGB 的转换。

2，JPEG_MCU_YCbCr422_ARGB_ConvertBlocks 函数，实现 YUV422→RGB 的转换。

3，JPEG_MCU_YCbCr444_ARGB_ConvertBlocks 函数，实现 YUV444→RGB 的转换。

4，JPEG_MCU_Gray_ARGB_ConvertBlocks 函数，实现灰度图像→RGB 的转换。

5，JPEG_MCU_YCCK_ARGB_ConvertBlocks 函数，实现 CMYK→RGB 的转换。

这五个函数都是用于色彩转换，支持将 YUV、灰度和 CMYK 格式转换为 RGB565、RGB888 和 ARGB8888 等格式，由于篇幅所限，就不贴出来了，请大家参考 jpeg_utils.c 的源代码。

ImageNbMCUs，用于表示当前 JPG/JPEG 文件总 MCU 数。该函数其他代码，我们就不多说了，请大家参考源码注释理解。

接下来，我们看 hjpgd.c 里面的代码，该文件代码如下：

```
jpeg_codec_typedef hjpgd;                                   //JPEG 硬件解码结构体
//JPEG 输入数据流,回调函数,用于获取 JPEG 文件原始数据
//每当 JPEG DMA IN BUF 为空的时候,调用该函数
void jpeg_dma_in_callback(void)
{
    hjpgd.inbuf[hjpgd.inbuf_read_ptr].sta=0;                //此 buf 已经处理完了
    hjpgd.inbuf[hjpgd.inbuf_read_ptr].size=0;               //此 buf 已经处理完了
    hjpgd.inbuf_read_ptr++;                                 //指向下一个 buf
    if(hjpgd.inbuf_read_ptr>=JPEG_DMA_INBUF_NB)hjpgd.inbuf_read_ptr=0;//归零
    if(hjpgd.inbuf[hjpgd.inbuf_read_ptr].sta==0)//无有效 buf
```

```
    {
        JPEG_IN_DMA_Pause();                                    //暂停读取数据
        hjpgd.indma_pause=1;                                    //暂停读取数据
    }else                                                       //有效的 buf
    {
        JPEG_IN_DMA_Resume((u32)hjpgd.inbuf[hjpgd.inbuf_read_ptr].buf,
            hjpgd.inbuf[hjpgd.inbuf_read_ptr].size);            //继续下一次 DMA 传输
    }
}
//JPEG 输出数据流(YUV)回调函数,用于输出 YUV 数据流
void jpeg_dma_out_callback(void)
{
    u32 *pdata=0;
    hjpgd.outbuf[hjpgd.outbuf_write_ptr].sta=1;                 //此 buf 已满
    hjpgd.outbuf[hjpgd.outbuf_write_ptr].size=JPEG_DMA_OUTBUF_LEN-
                        (DMA2_Stream1->NDTR<<2);//此 buf 里面数据的长度
    if(hjpgd.state==JPEG_STATE_FINISHED)//解码完成,需读 DOR 最后的数据
    {
        pdata=(u32*)(hjpgd.outbuf[hjpgd.outbuf_write_ptr].buf+
                        hjpgd.outbuf[hjpgd.outbuf_write_ptr].size);
        while(JPEG->SR&(1<<4))
        {
            *pdata=JPEG->DOR;
            pdata++;
            hjpgd.outbuf[hjpgd.outbuf_write_ptr].size+=4;
        }
    }
    hjpgd.outbuf_write_ptr++;                                   //指向下一个 buf
    if(hjpgd.outbuf_write_ptr>=JPEG_DMA_OUTBUF_NB)hjpgd.outbuf_write_ptr=0;
    if(hjpgd.outbuf[hjpgd.outbuf_write_ptr].sta==1)//无有效 buf
    {
        JPEG_OUT_DMA_Pause();                                   //暂停输出数据
        hjpgd.outdma_pause=1;                                   //暂停输出数据
    }else                                                       //有效的 buf
    {
        JPEG_OUT_DMA_Resume((u32)hjpgd.outbuf[hjpgd.outbuf_write_ptr].buf,
            JPEG_DMA_OUTBUF_LEN);                               //继续下一次 DMA 传输
    }
}
//JPEG 整个文件解码完成回调函数
void jpeg_endofcovert_callback(void)
{
    hjpgd.state=JPEG_STATE_FINISHED;                            //标记 JPEG 解码完成
```

```
}
//JPEG header 解析成功回调函数
void jpeg_hdrover_callback(void)
{
    hjpgd.state=JPEG_STATE_HEADEROK;            //HEADER 获取成功
    JPEG_Get_Info(&hjpgd);          //获取 JPEG 相关信息,包括大小,色彩空间,抽样等
    JPEG_GetDecodeColorConvertFunc(&hjpgd.Conf,&hjpgd.ycbcr2rgb,
                &hjpgd.total_blks);//获取 JPEG 色彩转换函数,以及总 MCU 数
    picinfo.ImgWidth=hjpgd.Conf.ImageWidth;
    picinfo.ImgHeight=hjpgd.Conf.ImageHeight;
    ai_draw_init();
}
//JPEG 硬件解码图片
//注意:
//1,待解吗图片的分辨率,必须小于等于屏幕的分辨率!
//2,请保证图片的宽度是 16 的倍数,以免左侧出现花纹.
//pname:图片名字(带路径)
//返回值:0,成功
//       其他,失败
u8 hjpgd_decode(u8* pname)
{
    FIL* ftemp;
    u16* rgb565buf;
    vu32 timecnt=0; u32 mcublkindex=0;
    u8 fileover=0;u8 i=0;u8 res;
    res=JPEG_Core_Init(&hjpgd);                         //初始化 JPEG 内核
    if(res)return 1;
    ftemp=(FIL*)mymalloc(SRAMIN,sizeof(FIL));           //申请内存
    if(f_open(ftemp,(char*)pname,FA_READ)!=FR_OK) //打开图片失败
    {
        JPEG_Core_Destroy(&hjpgd);
        myfree(SRAMIN,ftemp);                           //释放内存
        return 2;
    }
    rgb565buf=mymalloc(SRAMEX,lcddev.width*lcddev.height*2);//申请整帧内存
    JPEG_Decode_Init(&hjpgd);                           //初始化硬件 JPEG 解码器
    for(i=0;i<JPEG_DMA_INBUF_NB;i++)
    {
        res=f_read(ftemp,hjpgd.inbuf[i].buf,JPEG_DMA_INBUF_LEN,&br);//填满 FIFO
        if(res==FR_OK&&br){ hjpgd.inbuf[i].size=br; hjpgd.inbuf[i].sta=1;} //标记 buf 满
        if(br==0)break;
    }
    JPEG_IN_OUT_DMA_Init((u32)hjpgd.inbuf[0].buf,(u32)hjpgd.outbuf[0].buf,
```

```
        hjpgd.inbuf[0].size,JPEG_DMA_OUTBUF_LEN);//配置 DMA
jpeg_in_callback=jpeg_dma_in_callback;          //JPEG DMA 读取数据回调函数
jpeg_out_callback=jpeg_dma_out_callback;        //JPEG DMA 输出数据回调函数
jpeg_eoc_callback=jpeg_endofcovert_callback;    //JPEG  解码结束回调函数
jpeg_hdp_callback=jpeg_hdrover_callback;        //JPEG Header 解码完成回调函数
JPEG_DMA_Start();                               //启动 DMA 传输
while(1)
{
    SCB_CleanInvalidateDCache();                //清空 D catch
    if(hjpgd.inbuf[hjpgd.inbuf_write_ptr].sta==0&&fileover==0)//有 buf 为空
    {
        res=f_read(ftemp,hjpgd.inbuf[hjpgd.inbuf_write_ptr].buf,
                   JPEG_DMA_INBUF_LEN,&br);     //填满一个缓冲区
        if(res==FR_OK&&br)
        {
            hjpgd.inbuf[hjpgd.inbuf_write_ptr].size=br; //读取
            hjpgd.inbuf[hjpgd.inbuf_write_ptr].sta=1;    //buf 满
        }else if(br==0){ timecnt=0; fileover=1;}      //清零计时器,标记文件结束
        if(hjpgd.indma_pause==1&&hjpgd.inbuf[hjpgd.inbuf_read_ptr].sta==1)
        //之前是暂停的,重新开始传输
        {
            JPEG_IN_DMA_Resume((u32)hjpgd.inbuf[hjpgd.inbuf_read_ptr].buf,
                hjpgd.inbuf[hjpgd.inbuf_read_ptr].size);//继续下一次 DMA 传输
            hjpgd.indma_pause=0;
        }
        hjpgd.inbuf_write_ptr++;
        if(hjpgd.inbuf_write_ptr>=JPEG_DMA_INBUF_NB)
            hjpgd.inbuf_write_ptr=0;
    }
    if(hjpgd.outbuf[hjpgd.outbuf_read_ptr].sta==1)   //buf 里面有数据要处理
    {
        mcublkindex+=hjpgd.ycbcr2rgb(hjpgd.outbuf[hjpgd.outbuf_read_ptr].buf,
            (u8*)rgb565buf,mcublkindex,hjpgd.outbuf[hjpgd.outbuf_read_ptr].size);
        hjpgd.outbuf[hjpgd.outbuf_read_ptr].sta=0;  //标记 buf 为空
        hjpgd.outbuf[hjpgd.outbuf_read_ptr].size=0;//数据量清空
        hjpgd.outbuf_read_ptr++;
        if(hjpgd.outbuf_read_ptr>=JPEG_DMA_OUTBUF_NB)
            hjpgd.outbuf_read_ptr=0;//限制范围
        if(mcublkindex==hjpgd.total_blks) break;
    }else if(hjpgd.outdma_pause==1&&hjpgd.outbuf[hjpgd.outbuf_write_ptr].sta==0)
    //out 暂停,且当前 writebuf 已经为空了,则恢复 out 输出
    {
        JPEG_OUT_DMA_Resume((u32)hjpgd.outbuf[hjpgd.outbuf_write_ptr].buf,
```

```
                    JPEG_DMA_OUTBUF_LEN);  //继续下一次 DMA 传输
                hjpgd.outdma_pause=0;
            }
            timecnt++;
            if(fileover)    //文件结束后,及时退出,防止死循环
            {
                if(hjpgd.state==JPEG_STATE_NOHEADER)break;  //解码失败了
                if(timecnt>0X3FFF)break;                    //超时退出
            }
        }
        if(hjpgd.state==JPEG_STATE_FINISHED)                //解码完成了
        {
            piclib_fill_color(picinfo.S_XOFF,picinfo.S_YOFF,hjpgd.Conf.ImageWidth,
                              hjpgd.Conf.ImageHeight,rgb565buf);
        }
        myfree(SRAMIN,ftemp); myfree(SRAMEX,rgb565buf);
        JPEG_Core_Destroy(&hjpgd);
        return 0;
    }
```

该文件里面，总共有 5 个函数，接下来分别介绍：

jpeg_dma_in_callback 函数，用于处理 JPEG 输入数据流，当 JPEG 输入 DMA 传输完成时，调用该函数。对已处理的 buf 标记清零，然后切换到下一个 buf。当 buf 不够时，暂停 JPEG 输入 FIFO 获取数据，并标记暂停；当 buf 足够时，切换到下一个 buf，继续传输。

jpeg_dma_out_callback 函数，用于处理 JPEG 输出数据流，当 JPEG 输入 DMA 传输完成时，调用该函数。对已满的 buf 标记满，并标记容量，然后切换到下一个 buf。当 buf 不够时，暂停获取 JPEG 输出 FIFO 的数据，并标记暂停；当 buf 足够时，切换到下一个 buf，继续传输。当解码状态结束时，需要手动读取 JPEG_DOR 寄存器的数据。

jpeg_endofcovert_callback 函数，在 JPG/JPEG 文件解码结束时调用。该函数处理非常简单，直接将当前解码状态标记为：JPEG 解码完成（JPEG_STATE_FINISHED）即可。

jpeg_hdrover_callback 函数，在 JPEG 头解码成功后调用。该函数先标记状态为 JPEG 头解码成功（JPEG_STATE_HEADEROK），然后调用 JPEG_Get_Info 函数获取 JPEG 相关信息，通过 JPEG_GetDecodeColorConvertFunc 函数取得颜色转换函数和总 MCU 数。最后初始化画图，准备解码显示。

hjpgd_decode 函数，用于解码一张 JPG/JPEG 图片。该函数采用我们在 50.1 节最后介绍的步骤来解码 JPG/JPEG 图片。请大家参考前面的介绍和源码进行理解。

另外，我们需要将 hjpgd_decode 函数加入到图片解码库里面，修改 ai_load_picfile 函数代码如下：

```
//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 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.
//      当有硬件 JPEG 解码的时候,快速解码使用硬件 jpeg 解码,以提高速度
//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;    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.
    if(width==0||height==0)return PIC_WINDOW_ERR;             //窗口设定错误
    picinfo.S_Height=height;
    picinfo.S_Width=width;
    if(picinfo.S_Height==0||picinfo.S_Width==0)               //显示区域无效
    {
        picinfo.S_Height=lcddev.height;
        picinfo.S_Width=lcddev.width;
        return FALSE;
    }
    if(pic_phy.fillcolor==NULL)fast=0;//颜色填充函数未实现,不能快速显示
    //显示的开始坐标点
    picinfo.S_YOFF=y;
    picinfo.S_XOFF=x;
    //文件名传递
    temp=f_typetell((u8*)filename);   //得到文件的类型
    switch(temp)
    {
        case T_BMP:
            res=stdbmp_decode(filename);                      //解码 bmp
            break;
        case T_JPG:
        case T_JPEG:
            if(fast)                                          //可能需要硬件解码
            {
                res=jpg_get_size(filename,&picinfo.ImgWidth,&picinfo.ImgHeight);
                if(res==0)
                {
                    if(picinfo.ImgWidth<=lcddev.width&&picinfo.ImgHeight<=
                       lcddev.height&&picinfo.ImgWidth<=picinfo.S_Width&&
                       picinfo.ImgHeight<=picinfo.S_Height)    //则可以硬件解码
                    {
                        res=hjpgd_decode((u8*)filename);//采用硬解码 JPG/JPEG
                    }else res=jpg_decode(filename,fast);    //采用软件解码 JPG/JPEG
                }
            }else res=jpg_decode(filename,fast);    //统一采用软件解码 JPG/JPEG
            break;
        case T_GIF:
            res=gif_decode(filename,x,y,width,height);  //解码 gif
```

```
                break;
            default:
                res=PIC_FORMAT_ERR;                    //非图片格式!!!
                break;
        }
        return res;
    }
```

当 JPG/JPEG 图片尺寸满足小于等于屏幕分辨率，且启用快速解码时，我们会通过调用 hjpgd_decode 函数实现硬件 JPEG 解码，从而大大提高速度。

最后，我们看看 main.c 文件，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)
{
    ……//请参考上一章例程源码/本例程源码
}
int main(void)
{
    u8 led0sta=1;u8 res;
    DIR picdir;             //图片目录
    FILINFO *picfileinfo;   //文件信息
    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; u8 t;u16 temp;
    u8 pause=0;             //暂停标记
    u32 *picoffsettbl;      //图片文件 offset 索引表
    Cache_Enable();                         //打开 L1-Cache
    MPU_Memory_Protection();                //保护相关存储区域
    HAL_Init();                             //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);           //设置时钟,216Mhz
    ……//省略部分代码
    totpicnum=pic_get_tnum("0:/PICTURE"); //得到总有效文件数
    while(totpicnum==NULL)//图片文件为 0
    {
        Show_Str(30,170,240,16,"没有图片文件!",16,0); delay_ms(200);
        LCD_Fill(30,170,240,186,WHITE); delay_ms(200);
    }
    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 内存,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); delay_ms(200);
}
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_Toggle;//LED0 闪烁,提示程序正在运行.
        if(key==KEY2_PRES)          //上一张
        {
```

```
                    if(curindex)curindex--;
                    else curindex=totpicnum-1;
                    break;
                }else if(key==KEY0_PRES)//下一张
                {
                    curindex++;
                    if(curindex>=totpicnum)curindex=0;//到末尾的时候,自动从头开始
                    break;
                }else if(key==WKUP_PRES){pause=!pause; LED1(!pause);}//暂停 LED1 亮.
                if(pause==0)t++;
                delay_ms(10);
            }
            res=0;
        }
        ……//省略部分代码
    }
```

这部分代码比较长，我们省略了一些内容。详细的代码，请大家参考光盘本例程源码。

这里除了 main 函数，还有一个 pic_get_tnum 的函数，用来得到 path 路径下，所有有效文件（图片文件）的个数。在 main 函数里面我们通过读/写偏移量（图片文件在 PICTURE 文件夹下的读/写偏移位置，可以看做是一个索引），来查找上一个/下一个图片文件（使用 dir_sdi 函数）。通过 ai_load_picfile 函数，实现对 JPG/JPEG 图片的解码。这里将 fast 参数设置为 1，当图片文件的分辨率小于等于液晶分辨率的时候，将使用硬件 JPEG 进行解码。

至此本例程代码编写完成。最后，本实验可以通过 USMART 来调用相关函数，以对比性能。将 mf_scan_files、ai_load_picfile 和 hjpgd_decode 等函数添加到 USMART 管理，即可以通过串口调用这几个函数，测试对比软件 JPEG 解码和硬件 JPEG 解码的速度差别。

50.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 开始显示图片（假设 SD 卡及文件都准备好了，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 50.4.1 所示：

图 50.4.1 硬件 JPEG 解码实验显示效果

按 KEY0 和 KEY2 可以快速切换到下一张或上一张，KEY_UP 按键可以暂停自动播放，同时 DS1 亮，指示处于暂停状态，再按一次 KEY_UP 则继续播放。对比上一章实验，我们可以发现，对于小尺寸的 JPG/JPEG 图片（小于液晶分辨率），本例程解码速度明显提升。

我们通过 USMART 调用 ai_load_picfile 函数，对比测试同一张图片，使用硬件 JPEG 解码和不使用硬件 JPEG 解码，速度差别明显，如图 50.4.2 所示：

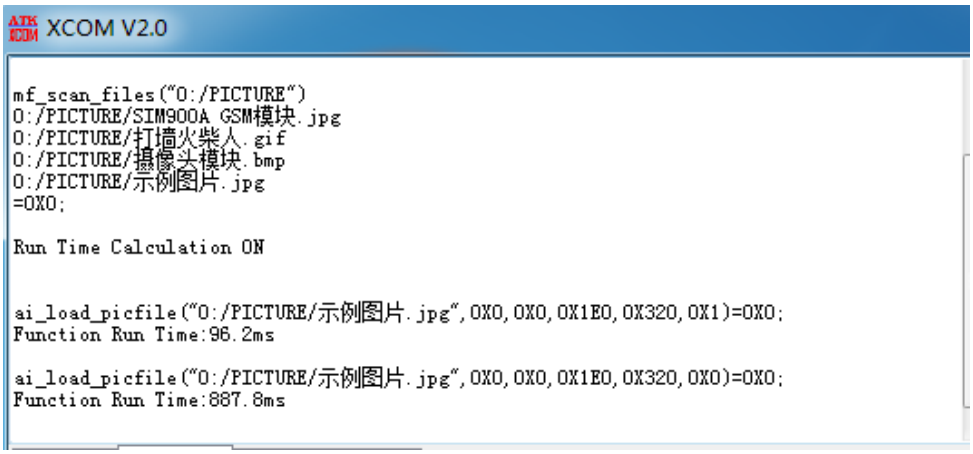


图 50.4.2 硬件 JPEG 与软件 JPEG 解码速度对比

上图，是我们使用 4.3 寸 800*480 分辨率的 MCU 屏做的测试，可以看出，对于同一张图片（图片分辨率：800*480），硬件 JPEG 解码，只需要 96.2ms，软件 JPEG 解码，则需要 887.8ms！硬件 JPEG 解码速度是软件 JPEG 解码的 9.2 倍！！可见，硬件 JPEG 解码大大提高了对 JPG/JPEG 图片的解码能力。

第五十一章 照相机实验

上一章，我们学习了图片解码，本章我们将学习 BMP&JPEG 编码，结合前面的摄像头实验，实现一个简单的照相机。本章分为如下几个部分：

51.1 BMP&JPEG 编码简介
51.2 硬件设计
51.3 软件设计
51.4 下载验证

51.1 BMP&JPEG 编码简介

本章，我们要实现的照相机，支持 BMP 图片格式的照片和 JPEG 图片格式的照片，这里简单介绍一下这两种图片格式的编码。这里我们使用 ATK-OV5640-AF 摄像头，来实现拍照。关于 OV5640 的相关知识点，请参考第四十三章。

51.1.1 BMP 编码简介

上一章，我们学习了各种图片格式的解码。本章，我们介绍最简单的图片编码方法：BMP 图片编码。通过前面的了解，我们知道 BMP 文件是由文件头、位图信息头、颜色信息和图形数据等四部分组成。我们先来了解下这几个部分。

1、BMP 文件头（14 字节）：BMP 文件头数据结构含有 BMP 文件的类型、文件大小和位图起始位置等信息。

```
//BMP 文件头
typedef __packed struct
{
    u16    bfType ;          //文件标志.只对'BM',用来识别 BMP 位图类型
    u32    bfSize ;          //文件大小,占四个字节
    u16    bfReserved1 ;     //保留
    u16    bfReserved2 ;     //保留
    u32    bfOffBits ;       //从文件开始到位图数据(bitmap data)开始之间的偏移量
}BITMAPFILEHEADER ;
```

2、位图信息头（40 字节）：BMP 位图信息头数据用于说明位图的尺寸等信息。

```
typedef __packed struct
{
    u32 biSize ;            //说明 BITMAPINFOHEADER 结构所需要的字数。
    long    biWidth ;       //说明图象的宽度，以象素为单位
    long    biHeight ;      //说明图象的高度，以象素为单位
    u16    biPlanes ;       //为目标设备说明位面数，其值将总是被设为 1
    u16    biBitCount ;     //说明比特数/象素，其值为 1、4、8、16、24、或 32
    u32 biCompression ;     //说明图象数据压缩的类型。其值可以是下述值之一：
    //BI_RGB：没有压缩；
    //BI_RLE8：每个象素 8 比特的 RLE 压缩编码，压缩格式由 2 字节组成
    //BI_RLE4：每个象素 4 比特的 RLE 压缩编码，压缩格式由 2 字节组成
```

```
    //BI_BITFIELDS：每个象素的比特由指定的掩码决定。
    u32 biSizeImage ;//说明图象的大小,以字节为单位。当用 BI_RGB 格式时,可设置为 0
    long    biXPelsPerMeter ;//说明水平分辨率，用象素/米表示
    long    biYPelsPerMeter ;//说明垂直分辨率，用象素/米表示
    u32 biClrUsed ;          //说明位图实际使用的彩色表中的颜色索引数
    u32 biClrImportant ;     //说明对图象显示有重要影响的颜色索引的数目，
                             //如果是 0，表示都重要。
}BITMAPINFOHEADER ;
```

3、颜色表：颜色表用于说明位图中的颜色，它有若干个表项，每一个表项是一个 RGBQUAD 类型的结构，定义一种颜色。

```
typedef __packed struct
{
    u8 rgbBlue ;        //指定蓝色强度
    u8 rgbGreen ;       //指定绿色强度
    u8 rgbRed ;         //指定红色强度
    u8 rgbReserved ;    //保留，设置为 0
}RGBQUAD ;
```

颜色表中 RGBQUAD 结构数据的个数由 biBitCount 来确定：当 biBitCount=1、4、8 时，分别有 2、16、256 个表项；当 biBitCount 大于 8 时，没有颜色表项。

BMP 文件头、位图信息头和颜色表组成位图信息（我们将 BMP 文件头也加进来，方便处理），BITMAPINFO 结构定义如下:

```
typedef __packed struct
{
    BITMAPFILEHEADER bmfHeader;
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1];
}BITMAPINFO;
```

4、位图数据：位图数据记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右，扫描行之间是从下到上。位图的一个像素值所占的字节数:

当 biBitCount=1 时，8 个像素占 1 个字节;
当 biBitCount=4 时，2 个像素占 1 个字节;
当 biBitCount=8 时，1 个像素占 1 个字节;
当 biBitCount=16 时，1 个像素占 2 个字节;
当 biBitCount=24 时，1 个像素占 3 个字节;
当 biBitCount=32 时，1 个像素占 4 个字节;

biBitCount=1 表示位图最多有两种颜色，缺省情况下是黑色和白色，你也可以自己定义这两种颜色。图像信息头装调色板中将有两个调色板项，称为索引 0 和索引 1。图象数据阵列中的每一位表示一个像素。如果一个位是 0，显示时就使用索引 0 的 RGB 值，如果位是 1，则使用索引 1 的 RGB 值。

biBitCount=16 表示位图最多有 65536 种颜色。每个像素用 16 位（2 个字节）表示。这种格式叫作高彩色，或叫增强型 16 位色，或 64K 色。它的情况比较复杂，当 biCompression 成员的值是 BI_RGB 时，它没有调色板。16 位中，最低的 5 位表示蓝色分量，中间的 5 位表示绿色分量，高的 5 位表示红色分量，一共占用了 15 位，最高的一位保留，设为 0。这种格式也被称

作 555 16 位位图。如果 biCompression 成员的值是 BI_BITFIELDS，那么情况就复杂了，首先是原来调色板的位置被三个 DWORD 变量占据，称为红、绿、蓝掩码。分别用于描述红、绿、蓝分量在 16 位中所占的位置。在 Windows 95（或 98）中，系统可接受两种格式的位域：555 和 565，在 555 格式下，红、绿、蓝的掩码分别是：0x7C00、0x03E0、0x001F，而在 565 格式下，它们则分别为：0xF800、0x07E0、0x001F。你在读取一个像素之后，可以分别用掩码“与”上像素值，从而提取出想要的颜色分量（当然还要再经过适当的左右移操作）。在 NT 系统中，则没有格式限制，只不过要求掩码之间不能有重叠。(注：这种格式的图像使用起来是比较麻烦的，不过因为它的显示效果接近于真彩，而图像数据又比真彩图像小的多，所以，它更多的被用于游戏软件)。

biBitCount=32 表示位图最多有 4294967296(2 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。

51.1.2 JPEG 编码简介

JPEG（Joint Photographic Experts Group）是一个由 ISO 和 IEC 两个组织机构联合组成的一个专家组，负责制定静态的数字图像数据压缩编码标准，这个专家组开发的算法称为 JPEG 算法，并且成为国际上通用的标准，因此又称为 JPEG 标准。JPEG 是一个适用范围很广的静态图像数据压缩标准，既可用于灰度图像又可用于彩色图像。

JPEG 专家组开发了两种基本的压缩算法，一种是采用以离散余弦变换（Discrete Cosine Transform，DCT）为基础的有损压缩算法，另一种是采用以预测技术为基础的无损压缩算法。使用有损压缩算法时，在压缩比为 25:1 的情况下，压缩后还原得到的图像与原始图像相比较，非图像专家难于找出它们之间的区别，因此得到了广泛的应用。

JPEG 压缩是有损压缩，它利用了人的视角系统的特性，使用量化和无损压缩编码相结合来去掉视角的冗余信息和数据本身的冗余信息。

JPEG 压缩编码分为三个步骤：

1）使用正向离散余弦变换（Forward Discrete Cosine Transform，FDCT）把空间域表示的图变换成频率域表示的图。

2）使用加权函数对 DCT 系数进行量化，这个加权函数对于人的视觉系统是最佳的。

3）使用霍夫曼可变字长编码器对量化系数进行编码。

这里我们不详细介绍 JPEG 压缩的过程了，大家可以自行查找相关资料。我们本章要实现的 JPEG 拍照，并不需要自己压缩图像，因为我们使用的 ALIENTEK OV5640 摄像头模块，直接就可以输出压缩后的 JPEG 数据，我们完全不需要理会压缩过程，所以本章我们实现 JPEG 拍照的关键，在于准确接收 OV5640 摄像头模块发送过来的编码数据，然后将这些数据保存为.jpg 文件，就可以实现 JPEG 拍照了。

在第四十二章，我们定义了一个很大的数组 jpeg_data_buf（4MB）来存储 JPEG 图像数据，但本章，我们可以使用内存管理来申请内存，无需定义这么大的数组，使用上更加灵活。另外，DCMI 接口使用 DMA 直接传输 JPEG 数据到外部 SDRAM 会出现数据丢失，所以 DMA 接收 JPEG 数据只能用内部 SRAM，然后再拷贝到外部 SDRAM。所以，我们本章将使用 DMA 的双缓冲机制来读取，DMA 双缓冲读取 JPEG 数据框图如图 51.1.2.1 所示：

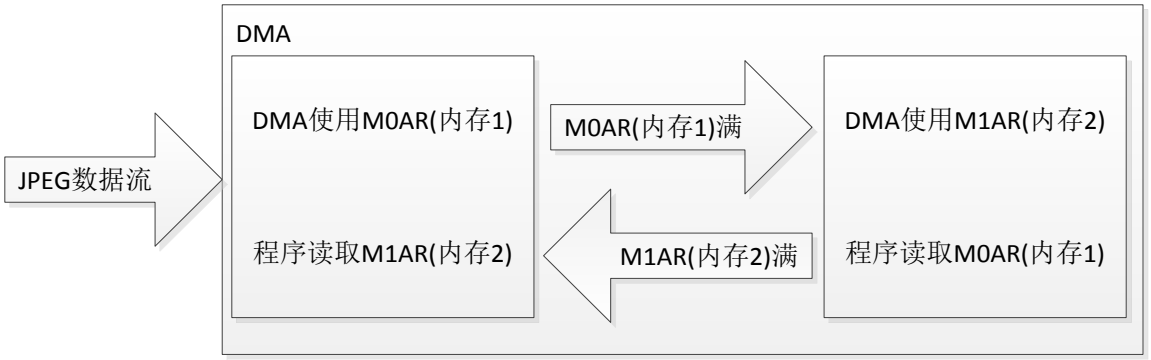


图 51.1.2.1 DMA 双缓冲读取 JPEG 数据原理框图

DMA 接收来自 OV5640 的 JPEG 数据流，首先使用 M0AR（内存 1）来存储，当 M0AR 满了以后，自动切换到 M1AR（内存 2），同时程序读取 M0AR（内存 1）的数据到外部 SDRAM；当 M1AR 满了以后，又切回 M0AR，同时程序读取 M1AR（内存 2）的数据到外部 SDRAM；依次循环（此时的数据处理，是通过 DMA 传输完成中断实现的，在中断里面处理），直到帧中断，结束一帧数据的采集，读取剩余数据到外部 SDRAM，完成一次 JPEG 数据的采集。

这里，M0AR，M1AR 所指向的内存，必须是内部内存，不过由于采用了双缓冲机制，我

们就不必定义一个很大的数组，一次性接收所有 JPEG 数据了，而是可以分批次接收，数组可以定义的比较小。

最后，将存储在外部 SDRAM 的 jpeg 数据，保存为.jpg/.jpeg 存放在 SD 卡，就完成了一次 JPEG 拍照。

51.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后检测 SD 卡根目录是否存在 PHOTO 文件夹，如果不存在则创建，如果创建失败，则报错（提示拍照功能不可用）。在找到 SD 卡的 PHOTO 文件夹后，开始初始化 OV5640，在初始化成功之后，就一直在屏幕显示 OV5640 拍到的内容。当按下 KEY_UP 按键的时候，可以选择缩放，还是 1:1 显示，默认缩放。按下 KEY0，可以拍 bmp 图片照片（分辨率为：LCD 辨率）。按下 KEY1 可以拍 JPEG 图片照片（分辨率为 QSXGA，即 2592 *1944）。拍照保存成功之后，蜂鸣器会发出“滴”的一声，提示拍照成功。DS0 还是用于指示程序运行状态，DS1 用于提示 DCMI 帧中断。

所要用到的硬件资源如下：

1） 指示灯 DS0 和 DS1
2） KEY0、KEY1 和 KEY_UP 按键
3） PCF8574T(控制蜂鸣器)
4） 串口
5） LCD 模块
6） SD 卡
7） SPI FLASH
8） OV5640 摄像头模块

这几部分，在之前的实例中都介绍过了，我们在此就不介绍了。需要注意的是：SD 卡与 DCMI 接口有部分 IO 共用，所以他们不能同时使用，必须分时复用，本章，这部分共用 IO 我们只有在拍照保存的时候，才切换为 SD 卡使用，其他时间，都是被 DCMI 占用的。

51.3 软件设计

打开本章实验工程，由于本章要用到 OV5640 和 PCF8574T 等外设，所以，先添加了 dcmi.c、sccb.c、ov5640.c、beep.c、sdram.c 和 pcf8574.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 51.1.1 节我们所介绍的方法，该函数实现了将 LCD 任意指定区域的内容，保存个为 16 位 BMP 格式，存放在指定位置（由 filename 决定）。注意，代码中的 BMP_USE_MALLOC 是在 bmp.h 定义的一个宏，用于设置是否使用 malloc，本章我们选择使用 malloc。

最后我们来看看 main.c 文件源码：

```
//处理 JPEG 数据
//当采集完一帧 JPEG 数据后,调用此函数,切换 JPEG BUF.开始下一帧采集.
void jpeg_data_process(void)
{
    u16 i;
    u16 rlen;               //剩余数据长度
    u32 *pbuf;
    curline=yoffset;     //行数复位
    if(ovx_mode&0X01)     //只有在 JPEG 格式下,才需要做处理.
    {
        if(jpeg_data_ok==0)     //jpeg 数据还未采集完?
        {
            DMA2_Stream1->CR&=~(1<<0);       //停止当前传输
            while(DMA2_Stream1->CR&0X01);   //等待 DMA2_Stream1 可配置
            rlen=jpeg_line_size-DMA2_Stream1->NDTR;//得到剩余数据长度
```

```
            pbuf=jpeg_data_buf+jpeg_data_len;//偏移到有效数据末尾,继续添加

            if(DMA2_Stream1->CR&(1<<19))for(i=0;i<rlen;i++)pbuf[i]=dcmi_line_buf[1][i];
                                        //读取 buf1 里面的剩余数据
            else for(i=0;i<rlen;i++)pbuf[i]=dcmi_line_buf[0][i];//读取 buf0 里面的剩余数据
            jpeg_data_len+=rlen;            //加上剩余长度
            jpeg_data_ok=1;             //标记 JPEG 数据采集完按成,等待其他函数处理
        }
        if(jpeg_data_ok==2)     //上一次的 jpeg 数据已经被处理了
        {
            DMA2_Stream1->NDTR=jpeg_line_size;//传输长度为 jpeg_buf_size*4 字节
            DMA2_Stream1->CR|=1<<0;             //重新传输
            jpeg_data_ok=0;                     //标记数据未采集
            jpeg_data_len=0;                    //数据重新开始
        }
    }else
    {
        if(bmp_request==1)      //有 bmp 拍照请求,关闭 DCMI
        {
            DCMI_Stop();//停止 DCMI
            bmp_request=0;      //标记请求处理完成.
        }
        LCD_SetCursor(0,0);
        LCD_WriteRAM_Prepare();  //开始写入 GRAM
    }
}
//jpeg 数据接收回调函数
void jpeg_dcmi_rx_callback(void)
{
    u16 i;
    u32 *pbuf;
    pbuf=jpeg_data_buf+jpeg_data_len;//偏移到有效数据末尾
    if(DMA2_Stream1->CR&(1<<19))//buf0 已满,正常处理 buf1
    {
        for(i=0;i<jpeg_line_size;i++)pbuf[i]=dcmi_line_buf[0][i];//读取 buf0 里面的数据
        jpeg_data_len+=jpeg_line_size;//偏移
    }else //buf1 已满,正常处理 buf0
    {
        for(i=0;i<jpeg_line_size;i++)pbuf[i]=dcmi_line_buf[1][i];//读取 buf1 里面的数据
        jpeg_data_len+=jpeg_line_size;//偏移
    }
}
```

```
//RGB 屏数据接收回调函数
void rgblcd_dcmi_rx_callback(void)
{
    u16 *pbuf;
    if(DMA2_Stream1->CR&(1<<19))//DMA 使用 buf1,读取 buf0
    {
        pbuf=(u16*)dcmi_line_buf[0];
    }else                                   //DMA 使用 buf0,读取 buf1
    {
        pbuf=(u16*)dcmi_line_buf[1];
    }
    LTDC_Color_Fill(0,curline,lcddev.width-1,curline,pbuf);//DM2D 填充
    if(curline<lcddev.height)curline++;
    if(bmp_request==1&&curline==(lcddev.height-1))//有 bmp 拍照请求,关闭 DCMI
    {
        DCMI_Stop();//停止 DCMI
        bmp_request=0;    //标记请求处理完成.
    }
}

//切换为 OV5640 模式
void sw_ov5640_mode(void)
{
    GPIO_InitTypeDef GPIO_Iniure;
    OV5640_WR_Reg(0X3017,0XFF);    //开启 OV5650 输出(可以正常显示)
    OV5640_WR_Reg(0X3018,0XFF);

    //GPIOC8/9/11 切换为 DCMI 接口
    GPIO_Iniure.Pin=GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_11;
    GPIO_Iniure.Mode=GPIO_MODE_AF_PP;          //推挽复用
    GPIO_Iniure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Iniure.Speed=GPIO_SPEED_HIGH;         //高速
    GPIO_Iniure.Alternate=GPIO_AF13_DCMI;      //复用为 DCMI
    HAL_GPIO_Init(GPIOC,&GPIO_Iniure);         //初始化
}
//切换为 SD 卡模式
void sw_sdcard_mode(void)
{
    GPIO_InitTypeDef GPIO_Iniure;
    OV5640_WR_Reg(0X3017,0X00);    //关闭 OV5640 全部输出(不影响 SD 卡通信)
    OV5640_WR_Reg(0X3018,0X00);

    //GPIOC8/9/11 切换为 SDIO 接口
```

```
    GPIO_Initure.Pin=GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_11;
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //推挽复用
    GPIO_Initure.Pull=GPIO_PULLUP;              //上拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    GPIO_Initure.Alternate=GPIO_AF12_SDMMC1;    //复用为 SDIO
    HAL_GPIO_Init(GPIOC,&GPIO_Initure);
}
//文件名自增（避免覆盖）
//mode:0,创建.bmp 文件;1,创建.jpg 文件.
//bmp 组合成:形如"0:PHOTO/PIC13141.bmp"的文件名
//jpg 组合成:形如"0:PHOTO/PIC13141.jpg"的文件名
void camera_new_pathname(u8 *pname,u8 mode)
{
……//省略部分代码
}
//OV5640 拍照 jpg 图片
//返回值:0,成功
//      其他,错误代码
u8 ov5640_jpg_photo(u8 *pname)
{
    FIL* f_jpg;
    u8 res=0,headok=0;
    u32 bwr;
    u32 i,jpgstart,jpglen;
    u8* pbuf;
    f_jpg=(FIL *)mymalloc(SRAMIN,sizeof(FIL));  //开辟 FIL 字节的内存区域
    if(f_jpg==NULL)return 0XFF;                 //内存申请失败.
    ovx_mode=1;
    jpeg_data_ok=0;
    sw_ov5640_mode();                           //切换为 OV5640 模式
    OV5640_JPEG_Mode();                         //JPEG 模式
    OV5640_OutSize_Set(16,4,2592,1944);         //设置输出尺寸(500W)
    dcmi_rx_callback=jpeg_dcmi_rx_callback;  //JPEG 接收数据回调函数
    DCMI_DMA_Init((u32)dcmi_line_buf[0],(u32)dcmi_line_buf[1],jpeg_line_size,
            DMA_MDATAALIGN_WORD,DMA_MINC_ENABLE);//DCMI DMA 配置
    DCMI_Start();               //启动传输
    while(jpeg_data_ok!=1);     //等待第一帧图片采集完
    jpeg_data_ok=2;             //忽略本帧图片,启动下一帧采集
    while(jpeg_data_ok!=1);     //等待第二帧图片采集完,第二帧,才保存到 SD 卡去.
    DCMI_Stop();                //停止 DMA 搬运
    ovx_mode=0;
    sw_sdcard_mode();       //切换为 SD 卡模式
    res=f_open(f_jpg,(const TCHAR*)pname,FA_WRITE|FA_CREATE_NEW);
```

```
                                        //模式 0,或者尝试打开失败,则创建新文件
    if(res==0)
    {
        printf("jpeg data size:%d\r\n",jpeg_data_len*4);//串口打印 JPEG 文件大小
        pbuf=(u8*)jpeg_data_buf;
        jpglen=0;//设置 jpg 文件大小为 0
        headok=0;       //清除 jpg 头标记
        for(i=0;i<jpeg_data_len*4;i++)//查找 0XFF,0XD8 和 0XFF,0XD9,获取 jpg 文件大小
        {
            if((pbuf[i]==0XFF)&&(pbuf[i+1]==0XD8))//找到 FF D8
            {
                jpgstart=i;
                headok=1;       //标记找到 jpg 头(FF D8)
            }
            if((pbuf[i]==0XFF)&&(pbuf[i+1]==0XD9)&&headok)//找到头以后,再找 FF D9
            {
                jpglen=i-jpgstart+2;
                break;
            }
        }
        if(jpglen)              //正常的 jpeg 数据
        {
            pbuf+=jpgstart;     //偏移到 0XFF,0XD8 处
            res=f_write(f_jpg,pbuf,jpglen,&bwr);
            if(bwr!=jpglen)res=0XFE;

        }else res=0XFD;
    }
    jpeg_data_len=0;
    f_close(f_jpg);
    sw_ov5640_mode();           //切换为 OV5640 模式
    OV5640_RGB565_Mode();  //RGB565 模式
    if(lcdltdc.pwidth!=0)     //RGB 屏
    {
        dcmi_rx_callback=rgblcd_dcmi_rx_callback;//RGB 屏接收数据回调函数
        DCMI_DMA_Init((u32)dcmi_line_buf[0],(u32)dcmi_line_buf[1],lcddev.width/2,
        DMA_MDATAALIGN_HALFWORD,DMA_MINC_ENABLE);//DCMI DMA 配置
    }else                           //MCU 屏
    {

       DCMI_DMA_Init((u32)&LCD->LCD_RAM,0,1,DMA_MDATAALIGN_HALFWORD,
       DMA_MINC_DISABLE); //DCMI DMA 配置,MCU 屏,竖屏
    }
```

```
    myfree(SRAMIN,f_jpg);
    return res;
}

int main(void)
{
    u8 res,fac;
    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.
    u8 scale=1;                 //默认是全尺寸缩放
    u8 msgbuf[15];              //消息缓存区
    u16 outputheight=0;

    Cache_Enable();             //打开 L1-Cache
    MPU_Memory_Protection();    //保护相关存储区域
    HAL_Init();                 //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);  //设置时钟,216Mhz
    delay_init(216);            //延时初始化
    uart_init(115200);          //串口初始化
    usmart_dev.init(108);       //初始化 USMART
    LED_Init();                 //初始化 LED
    KEY_Init();
    ……//省略部分代码
    res=f_mkdir("0:/PHOTO");        //创建 PHOTO 文件夹
    if(res!=FR_EXIST&&res!=FR_OK)   //发生了错误
    {
        res=f_mkdir("0:/PHOTO");    //创建 PHOTO 文件夹
        Show_Str(30,190,240,16,"SD 卡错误!",16,0);
        delay_ms(200);
        Show_Str(30,190,240,16,"拍照功能将不可用!",16,0);
        delay_ms(200);
        sd_ok=0;
    }
    dcmi_line_buf[0]=mymalloc(SRAMIN,jpeg_line_size*4);  //为 jpeg dma 接收申请内存
    dcmi_line_buf[1]=mymalloc(SRAMIN,jpeg_line_size*4);  //为 jpeg dma 接收申请内存
    jpeg_data_buf=mymalloc(SRAMEX,jpeg_buf_size);        //为 jpeg 申请内存(最大 4MB)
    pname=mymalloc(SRAMIN,30);//为带路径的文件名分配 30 个字节的内存
    while(pname==NULL||!dcmi_line_buf[0]||!dcmi_line_buf[1]||!jpeg_data_buf)  //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(OV5640_Init())//初始化 OV5640
    {
        Show_Str(30,190,240,16,"OV5640  错误!",16,0);
        delay_ms(200);
        LCD_Fill(30,190,239,206,WHITE);
        delay_ms(200);
    }
    Show_Str(30,210,230,16,"OV5640  正常",16,0);
    //自动对焦初始化
    OV5640_RGB565_Mode();  //RGB565 模式
    OV5640_Focus_Init();
    OV5640_Light_Mode(0);      //自动模式
    OV5640_Color_Saturation(3);//色彩饱和度 0
    OV5640_Brightness(4); //亮度 0
    OV5640_Contrast(3);          //对比度 0
    OV5640_Sharpness(33);//自动锐度
    OV5640_Focus_Constant();//启动持续对焦
    DCMI_Init();              //DCMI 配置
    if(lcdltdc.pwidth!=0)     //RGB 屏
    {
        dcmi_rx_callback=rgblcd_dcmi_rx_callback;//RGB 屏接收数据回调函数
        DCMI_DMA_Init((u32)dcmi_line_buf[0],(u32)dcmi_line_buf[1],lcddev.width/2,
        DMA_MDATAALIGN_HALFWORD,DMA_MINC_ENABLE);//DCMI DMA 配置
    }else                             //MCU  屏
    {

       DCMI_DMA_Init((u32)&LCD->LCD_RAM,0,1,DMA_MDATAALIGN_HALFWORD,
                                  DMA_MINC_DISABLE); //DCMI DMA 配置,MCU 屏,竖屏
    }
    if(lcddev.height>800)
    {
        yoffset=(lcddev.height-800)/2;
        outputheight=800;
        OV5640_WR_Reg(0x3035,0X51);//降低输出帧率，否则可能抖动
    }else
    {
        yoffset=0;
        outputheight=lcddev.height;
    }
    curline=yoffset;          //行数复位
```

```
OV5640_OutSize_Set(16,4,lcddev.width,outputheight);      //满屏缩放显示
DCMI_Start();                    //启动传输
LCD_Clear(BLACK);
while(1)
{
    key=KEY_Scan(0);//不支持连按
    if(key)
    {
        if(key!=KEY2_PRES)
        {
            if(key==KEY0_PRES)//BMP 拍照则等待 1 秒去抖以获得稳定 bmp 照片

            {
                delay_ms(300);
                bmp_request=1;          //请求关闭 DCMI
                while(bmp_request);     //等带请求处理完成
            }else DCMI_Stop();
        }
        if(key==WKUP_PRES)       //缩放处理
        {
            scale=!scale;
            if(scale==0)
            {
                fac=800/outputheight;//得到比例因子

            OV5640_OutSize_Set((1280-fac*lcddev.width)/2,
                               (800-fac*outputheight)/2,lcddev.width,outputheight);
                sprintf((char*)msgbuf,"Full Size 1:1");
            }else
            {
                OV5640_OutSize_Set(16,4,lcddev.width,outputheight);
                sprintf((char*)msgbuf,"Scale");
            }
            delay_ms(800);
        }else if(key==KEY2_PRES)OV5640_Focus_Single(); //手动单次自动对焦
        else if(sd_ok)//SD 卡正常才可以拍照
        {
            sw_sdcard_mode();//切换为 SD 卡模式
            if(key==KEY0_PRES)  //BMP 拍照
            {
                camera_new_pathname(pname,0); //得到文件名
                res=bmp_encode(pname,0,yoffset,lcddev.width,outputheight,0);
                sw_ov5640_mode();                       //切换为 OV5640 模式
```

```
            }else if(key==KEY1_PRES)//JPG 拍照
            {
                camera_new_pathname(pname,1);//得到文件名
                res=ov5640_jpg_photo(pname);
                if(scale==0)
                {
                    fac=800/lcddev.height;//得到比例因子

                    OV5640_OutSize_Set((1280-fac*lcddev.width)/2,
                        (800-fac*lcddev.height)/2,lcddev.width,lcddev.height);
                }else
                {
                    OV5640_OutSize_Set(16,4,lcddev.width,outputheight);
                }
        if(lcddev.height>800)OV5640_WR_Reg(0x3035,0X51);//降低输出帧率防抖
            }
            if(res)//拍照有误
            {
                Show_Str(30,130,240,16,"写入文件错误!",16,0);
            }else
            {
                Show_Str(30,130,240,16,"拍照成功!",16,0);
                Show_Str(30,150,240,16,"保存为:",16,0);
                Show_Str(30+56,150,240,16,pname,16,0);
                PCF8574_WriteBit(BEEP_IO,0);  //蜂鸣器短叫，提示拍照完成
                delay_ms(100);
                PCF8574_WriteBit(BEEP_IO,1);  //关闭蜂鸣器
            }
            delay_ms(1000);          //等待 1 秒钟
            DCMI_Start();//先使能 dcmi 然后关闭面再开 DCMI 防止 RGB 屏侧移
            DCMI_Stop();
        }else //提示 SD 卡错误
        {
            Show_Str(30,130,240,16,"SD 卡错误!",16,0);
            Show_Str(30,150,240,16,"拍照功能不可用!",16,0);
        }
        if(key!=KEY2_PRES)DCMI_Start();//开始显示
    }
    delay_ms(10);
    i++;
    if(i==20)//DS0 闪烁.
    {
        i=0;
```

```
                LED0_Toggle;
            }
        }
    }
```

这部分代码比较长，我们省略了一些内容。详细的代码，请大家参考光盘本例程源码。在 main.c 里面，总共有 8 个函数，我们接下来分别介绍。

1，jpeg_data_process 函数

该函数用于处理 JPEG 数据的接收，在 DCMI_IRQHandler 函数（在 dcmi.c 里面）里面被调用，它与 jpeg_dcmi_rx_callback 函数和 ov5640_jpg_photo 函数共同控制 JPEG 的数据的采集。JPEG 数据的接收，采用 DMA 双缓冲机制，缓冲数组为：dcmi_line_buf（u32 类型，RGB 屏接收 RGB565 数据时，也是用这个数组）；数组大小为：jpeg_line_size，我们定义的是 2*1024，即数组大小为 8K 字节（数组大小不能小于存储摄像头一行输出数据的大小）；JPEG 数据接收处理流程就是按图 51.1.2.1 所示流程来实现的。由 DMA 传输完成中断和 DCMI 帧中断，两个中断服务函数共同完成 jpeg 数据的采集。采集到的 JPEG 数据，全部存储在 jpeg_data_buf 数组里面，jpeg_data_buf 数组采用内存管理，从外部 SDRAM 申请 4MB 内存作为 JPEG 数据的缓存。

2，jpeg_dcmi_rx_callback 函数

这是 jpeg 数据接收的主要函数，通过判断 DMA2_Stream1->CR 寄存器，读取不同 dcmi_line_buf 里面的数据，存储到 SDRAM 里面（jpeg_data_buf）。该函数由 DMA 的传输完成中断服务函数：DMA2_Stream1_IRQHandler 调用。

3，rgblcd_dcmi_rx_callback 函数

该函数仅在使用 RGB 屏的时候用到。当使用 RGB 屏的时候，我们每接收一行数据，就使用 DMA2D 填充到 RGB 屏的 GRAM，这里同样是使用 DMA 的双缓冲机制来接收 RGB565 数据，原理参照图 51.1.2.1。该函数由 DMA 传输完成中断服务函数调用。

4，sw_ov5640_mode

因为 SD 卡和 OV5640 有几个 IO 共用，所以这几个 IO 需要分时复用。该函数用于切换 GPIO8/9/11 的复用功能为 DCMI 接口，并开启 OV5640，这样摄像头模块，可以开始正常工作。

5，sw_sdcard_mode

该数用于切换 GPIO8/9/11 的复用功能为 SDMMC 接口，并关闭 OV5640，这样，SD 卡可以开始正常工作。

6，camera_new_pathname 函数

该函数用于生成新的带路径的文件名，且不会重复，防止文件互相覆盖。该函数可以生成.bmp/.jpg 的文件名，方便拍照的时候，保存到 SD 卡里面。

7，ov5640_jpg_photo 函数

该函数实现 OV5640 的 JPEG 图像采集，并保存图像到 SD 卡，完成 JPEG 拍照。该函数首先设置 OV5640 工作在 JPEG 模式，然后，设置输出分辨率为最高的 QSXGA（2592*1944）。然后，开始采集 JPEG 数据，将第二帧 JPEG 数据，保留下来，并写入 SD 卡里面，完成一次 JPEG 拍照。这里，我们丢弃第一帧 JPEG 数据，是防止采集到的图像数据不完整，导致图片错误。

另外，在保存 jpeg 图片的时候，我们将 0XFF,0XD8 和 0XFF,0XD9 之外的数据，进行了剔除，只留下 0XFF,0XD8~0XFF,0XD9 之间的数据，保证图片文件最小，且无其他乱的数据。

注意，在保存图片的时候，必须将 PC8/9/11 切换为 SD 卡模式，并关闭 OV5640 的输出。在图片保存完成以后，切换回 OV5640 模式，并重新使能 OV5640 的输出。

8，main 函数

该函数完成对各相关硬件的初始化，然后检测 OV5640，初始化 OV5640 位 RGB565 模式，

显示采集到的图像到 LCD 上面，实现对图像进行预览。进入主循环以后，按 KEY0 按键，可以实现 BMP 拍照（实际上就是截屏，通过 bmp_encode 函数实现）；按 KEY1 按键，可实现 JPEG 拍照（2592*1944 分辨率，通过 ov5640_jpg_photo 函数实现）；按 KEY2 按键，可以实现自动对焦（单次）；按 KEY_UP 按键，可以实现图像缩放/不缩放预览。main 函数实现了我们在 51.2 节所提到的功能。

至此照相机实验代码编写完成。最后，本实验可以通过 USMART 来设置 OV5640 的相关参数，将 OV5640_Contrast、OV5640_Color_Saturation 和 OV5640_Light_Mode 等函数添加到 USMART 管理，即可通过串口设置 OV5640 的参数，方便调试。

51.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到如图 51.4.1 所示界面：

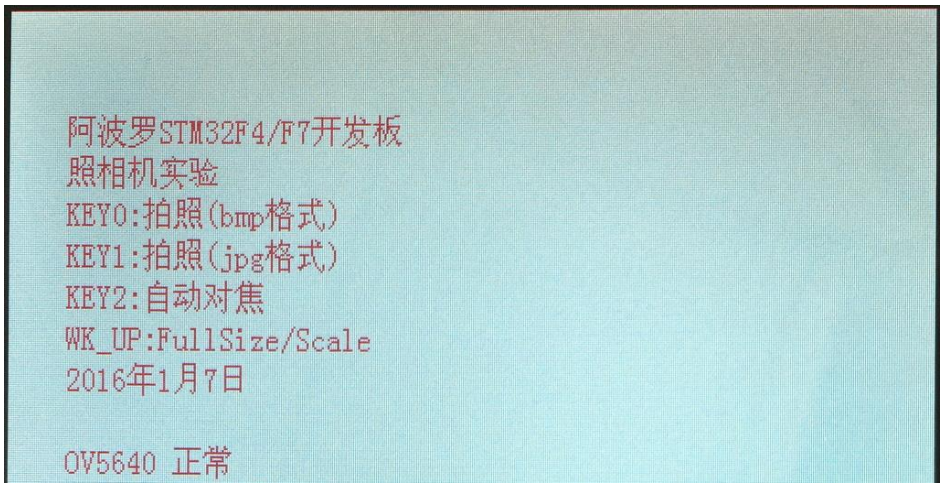


图 51.4.1 程序运行效果图

随后，进入监控界面。此时，我们可以按下 KEY0 和 KEY1，即可进行 bmp/jpg 拍照。拍照得到的照片效果如图 51.4.2 和图 51.4.3 所示：

图 51.4.2 拍照样图（bmp 拍照样图）

图 51.4.3 拍照样图（jpg 拍照样图）

如果发现对焦不清晰，可以按 KEY2 进行一次自动对焦。按 KEY_UP 可以实现缩放/不缩放显示。

第五十二章 音乐播放器实验

ALIENTEK 阿波罗 STM32F767 开发板拥有串行音频接口（SAI），支持 I2S、LSB/MSB 对其、PCM/DSP、TDM 和 AC’97 等协议，且外扩了一颗 HIFI 级 CODEC 芯片：WM8978G，支持最高 192K 24BIT 的音频播放，并且支持录音（下一章介绍）。本章，我们将利用阿波罗 STM32F767 开发板实现一个简单的音乐播放器（仅支持 WAV 播放）。本章分为如下几个部：

52.1 WAV&WM8978&SAI 简介
52.2 硬件设计
52.3 软件设计
52.4 下载验证

52.1 WAV&WM8978&SAI 简介

本章新知识点比较多，包括：WAV、WM8978 和 SAI 等三个知识点。下面我们将分别向大家介绍。

52.1.1 WAV 简介

WAV 即 WAVE 文件，WAV 是计算机领域最常用的数字化声音文件格式之一，它是微软专门为 Windows 系统定义的波形文件格式（Waveform Audio），由于其扩展名为"*.wav"。它符合 RIFF(Resource Interchange File Format)文件规范，用于保存 Windows 平台的音频信息资源，被 Windows 平台及其应用程序所广泛支持，该格式也支持 MSADPCM，CCITT A LAW 等多种压缩运算法，支持多种音频数字，取样频率和声道，标准格式化的 WAV 文件和 CD 格式一样，也是 44.1K 的取样频率，16 位量化数字，因此在声音文件质量和 CD 相差无几！

WAV 一般采用线性 PCM（脉冲编码调制）编码，本章，我们也主要讨论 PCM 的播放，因为这个最简单。

WAV 文件是由若干个 Chunk 组成的。按照在文件中的出现位置包括：RIFF WAVE Chunk、Format Chunk、 Fact Chunk(可选)和 Data Chunk。每个 Chunk 由块标识符、数据大小和数据三部分组成，如图 52.1.1.1 所示：

图 52.1.1.1 Chunk 结构示意图

其中块标识符由 4 个 ASCII 码构成，数据大小则标出紧跟其后的数据的长度（单位为字节），注意这个长度不包含块标识符和数据大小的长度，即不包含最前面的 8 个字节。所以实际 Chunk 的大小为数据大小加 8。

首先，我们来看看 RIFF 块（RIFF WAVE Chunk），该块以“RIFF”作为标示，紧跟 wav 文件大小（该大小是 wav 文件的总大小-8），然后数据段为“WAVE”，表示是 wav 文件。RIFF 块的 Chunk 结构如下：

```
//RIFF 块
typedef __packed struct
{
    u32 ChunkID;            //chunk id;这里固定为"RIFF",即 0X46464952
```

```
    u32 ChunkSize ;         //集合大小;文件总大小-8
    u32 Format;             //格式;WAVE,即 0X45564157
}ChunkRIFF ;
```

接着，我们看看 Format 块（Format Chunk），该块以“fmt ”作为标示（注意有个空格！），一般情况下，该段的大小为 16 个字节，但是有些软件生成的 wav 格式，该部分可能有 18 个字节，含有 2 个字节的附加信息。Format 块的 Chunk 结构如下：

```
//fmt 块
typedef __packed struct
{
    u32 ChunkID;            //chunk id;这里固定为"fmt ",即 0X20746D66
    u32 ChunkSize ;         //子集合大小(不包括 ID 和 Size);这里为:20.
    u16 AudioFormat;        //音频格式;0X10,表示线性 PCM;0X11 表示 IMA ADPCM
    u16 NumOfChannels;      //通道数量;1,表示单声道;2,表示双声道;
    u32 SampleRate;         //采样率;0X1F40,表示 8Khz
    u32 ByteRate;           //字节速率;
    u16 BlockAlign;         //块对齐(字节);
    u16 BitsPerSample;      //单个采样数据大小;4 位 ADPCM,设置为 4
}ChunkFMT;
```

接下来，我们再看看 Fact 块（Fact Chunk），该块为可选块，以“fact”作为标示，不是每个 WAV 文件都有，在非 PCM 格式的文件中，一般会在 Format 结构后面加入一个 Fact 块，该块 Chunk 结构如下：

```
//fact 块
typedef __packed struct
{
    u32 ChunkID;            //chunk id;这里固定为"fact",即 0X74636166;
    u32 ChunkSize ;         //子集合大小(不包括 ID 和 Size);这里为:4.
    u32 DataFactSize;       //数据转换为 PCM 格式后的大小
}ChunkFACT;
```

DataFactSize 是这个 Chunk 中最重要的数据，如果这是某种压缩格式的声音文件，那么从这里就可以知道他解压缩后的大小。对于解压时的计算会有很大的好处！不过本章我们使用的是 PCM 格式，所以不存在这个块。

最后，我们来看看数据块（Data Chunk），该块是真正保存 wav 数据的地方，以“data”'作为该 Chunk 的标示，然后是数据的大小。数据块的 Chunk 结构如下：

```
//data 块
typedef __packed struct
{
    u32 ChunkID;            //chunk id;这里固定为"data",即 0X61746164
    u32 ChunkSize ;         //子集合大小(不包括 ID 和 Size);文件大小-60.
}ChunkDATA;
```

ChunkSize 后紧接着就是 wav 数据。根据 Format Chunk 中的声道数以及采样 bit 数，wav 数据的 bit 位置可以分成如表 52.1.1.1 所示的几种形式：

单声道	取样 1	取样 2	取样 3	取样 4	取样 5	取样 6
8 位量化	声道 0	声道 0	声道 0	声道 0	声道 0	声道 0

双声道	取样 1		取样 2		取样 3	
8 位量化	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)
单声道	取样 1		取样 2		取样 3	
16 位量化	声道 0 (低字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (高字节)
双声道	取样 1				取样 2	
16 位量化	声道 0 (低字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (高字节)	声道 0 (低字节)	声道 0 (高字节)
单声道	取样 1				取样 2	
24 位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)
双声道	取样 1					
24 位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (中字节)	声道 1 (高字节)

表 52.1.1.1 WAVE 文件数据采样格式

本章，我们播放的音频支持：16 位和 24 位，立体声，所以每个取样为 4/6 个字节，低字节在前，高字节在后。在得到这些 wav 数据以后，通过 I2S 丢给 WM8978，就可以欣赏音乐了。

52.1.2 WM8978 简介

WM8978 是欧胜（Wolfson）推出的一款全功能音频处理器。它带有一个 HI-FI 级数字信号处理内核，支持增强 3D 硬件环绕音效，以及 5 频段的硬件均衡器，可以有效改善音质；并有一个可编程的陷波滤波器，用以去除屏幕开、切换等噪音。

WM8978 同样集成了对麦克风的支持，以及用于一个强悍的扬声器功放，可提供高达 900mW 的高质量音响效果扬声器功率。

一个数字回放限制器可防止扬声器声音过载。WM8978 进一步提升了耳机放大器输出功率，在推动 16 欧姆耳机的时候，每声道最大输出功率高达 40 毫瓦！可以连接市面上绝大多数适合随身听的高端 HI-FI 耳机。

WM8988 的主要特性有：

●I2S 接口，支持最高 192K,24bit 音频播放

●DAC 信噪比 98dB；ADC 信噪比 90dB

●支持无电容耳机驱动（提供 40mW@16Ω 的输出能力）

●支持扬声器输出（提供 0.9W@8Ω 的驱动能力）

●支持立体声差分输入/麦克风输入

●支持左右声道音量独立调节

●支持 3D 效果，支持 5 路 EQ 调节

WM8978 的控制通过 I2S 接口（即数字音频接口）同 MCU 进行音频数据传输（支持音频接收和发送），通过两线（MODE=0，即 IIC 接口）或三线（MODE=1）接口进行配置。WM8978 的 I2S 接口，由 4 个引脚组成：

1，ADCDAT:ADC 数据输出

2，DACDAT：DAC 数据输入

3，LRC：数据左/右对齐时钟

4，BCLK：位时钟，用于同步

WM8978 可作为 I2S 主机，输出 LRC 和 BLCK 时钟，不过我们一般使用 WM8978 作为从机，接收 LRC 和 BLCK。另外，WM8978 的 I2S 接口支持 5 中不同的音频数据模式：左（MSB）对齐标准、右（LSB）对齐标准、飞利浦（I2S）标准、DSP 模式 A 和 DSP 模式 B。本章，我们用飞利浦标准来传输 I2S 数据。

飞利浦（I2S）标准模式，数据在跟随 LRC 传输的 BCLK 的第二个上升沿时传输 MSB，其他位一直到 LSB 按顺序传输。传输依赖于字长、BCLK 频率和采样率，在每个采样的 LSB 和下一个采样的 MSB 之间都应该有未用的 BCLK 周期。飞利浦标准模式的 I2S 数据传输协议如图 52.1.2.1 所示：

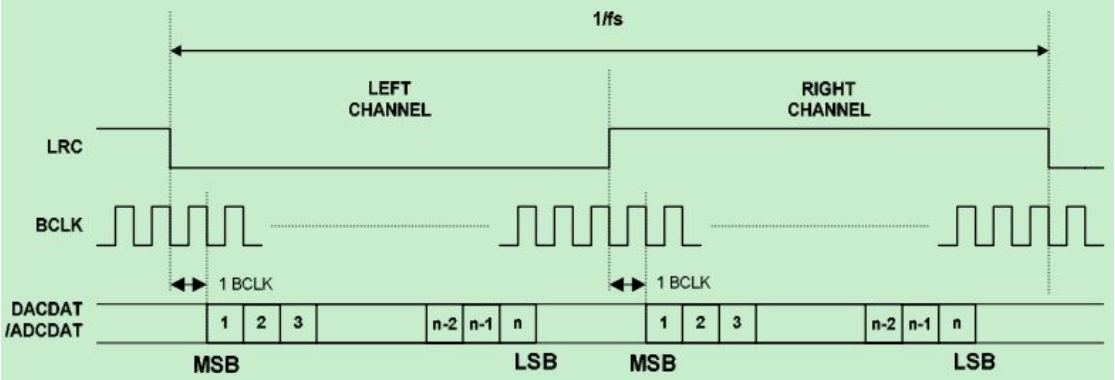


图 52.1.2.1 飞利浦标准模式 I2S 数据传输图

图中，fs 即音频信号的采样率，比如 44.1Khz，因此可以知道，LRC 的频率就是音频信号的采样率。另外，WM8978 还需要一个 MCLK，本章我们采用 STM32F767 为其提供 MCLK 时钟，MCLK 的频率必须等于 256fs，也就是音频采样率的 256 倍。

WM8978 的框图如图 52.1.2.2 所示：

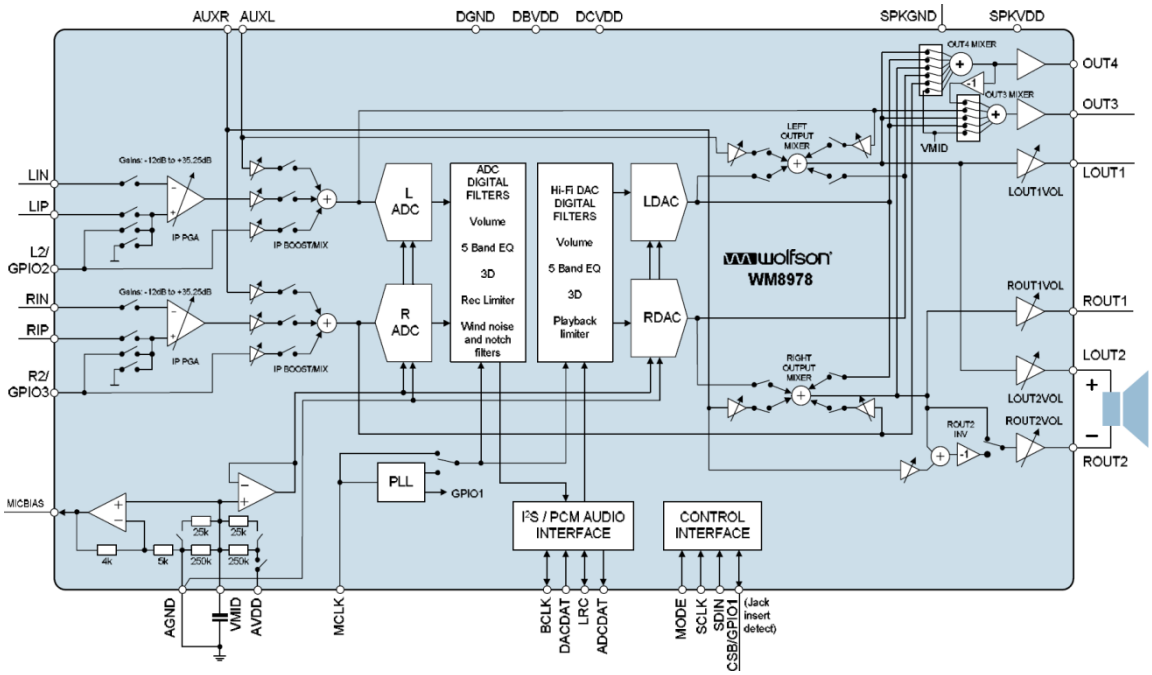


图 52.1.2.2 WM8978 框图

从上图可以看出，WM8978 内部有很多的模拟开关，用来选择通道，同时还有很多调节器，用来设置增益和音量。

本章，我们通过 IIC 接口（MODE=0）连接 WM8978，不过 WM8978 的 IIC 接口比较特殊：1，只支持写，不支持读数据；2，寄存器长度为 7 位，数据长度为 9 位。3，寄存器字节的最低位用于传输数据的最高位（也就是 9 位数据的最高位，7 位寄存器的最低位）。WM8978 的 IIC 地址固定为：0X1A。关于 WM8978 的 IIC 详细介绍，请看其数据手册第 77 页。

这里我们简单介绍一下要正常使用 WM8978 来播放音乐，应该执行哪些配置。

1，寄存器 R0（00h），该寄存器用于控制 WM8978 的软复位，写任意值到该寄存器地址，即可实现软复位 WM8978。

2，寄存器 R1（01h），该寄存器主要要设置 BIASEN（bit3），该位设置为 1，模拟部分的放大器才会工作，才可以听到声音。

3，寄存器 R2（02h），该寄存器要设置 ROUT1EN(bit8)，LOUT1EN(bit7)和 SLEEP(bit6)等三个位，ROUT1EN 和 LOUT1EN，设置为 1，使能耳机输出，SLEEP 设置为 0，进入正常工作模式。

4，寄存器 R3（03h），该寄存器要设置 LOUT2EN(bit6)，ROUT2EN(bit5)，RMIXER(bit3)，LMIXER(bit2)，DACENR(bit1)和 DACENL(bit0)等 6 个位。LOUT2EN 和 ROUT2EN，设置为 1，使能喇叭输出；LMIXER 和 RMIXER 设置为 1，使能左右声道混合器；DACENL 和 DACENR 则是使能左右声道的 DAC 了，必须设置为 1。

5，寄存器 R4（04h），该寄存器要设置 WL(bit6:5)和 FMT(bit4:3)等 4 个位。WL(bit6:5)用于设置字长(即设置音频数据有效位数)，00 表示 16 位音频，10 表示 24 位音频；FMT(bit4:3)用于设置 I2S 音频数据格式（模式），我们一般设置为 10，表示 I2S 格式，即飞利浦模式。

6，寄存器 R6（06h），该寄存器我们直接全部设置为 0 即可，设置 MCLK 和 BCLK 都来自外部，即由 STM32F767 提供。

7，寄存器 R10（0Ah），该寄存器我们要设置 SOFTMUTE(bit6)和 DACOSR128(bit3)等两个位，SOFTMUTE 设置为 0，关闭软件静音；DACOSR128 设置为 1，DAC 得到最好的 SNR。

8，寄存器 R43（2Bh），该寄存器我们只需要设置 INVROUT2 为 1 即可，反转 ROUT2 输出，更好的驱动喇叭。

9，寄存器 R49（31h），该寄存器我们要设置 SPKBOOST(bit2)和 TSDEN(bit1)这两个位。SPKBOOST 用于设置喇叭的增益，我们默认设置为 0 就好了（gain=-1），如想获得更大的声音，设置为 1（gain=+1.5）即可；TSDEN 用于设置过热保护，设置为 1（开启）即可。

10，寄存器 R50（32h）和 R51（33h），这两个寄存器设置类似，一个用于设置左声道（R50），另外一个用于设置右声道（R51）。我们只需要设置这两个寄存器的最低位为 1 即可，将左右声道的 DAC 输出接入左右声道混合器里面，才能在耳机/喇叭听到音乐。

11，寄存器 R52（34h）和 R53（35h），这两个寄存器用于设置耳机音量，同样一个用于设置左声道（R52），另外一个用于设置右声道（R53）。这两个寄存器的最高位（HPVU）用于设置是否更新左右声道的音量，最低 6 位用于设置左右声道的音量，我们可以先设置好两个寄存器的音量值，最后设置其中一个寄存器最高位为 1，即可更新音量设置。

12，寄存器 R54（36h）和 R55（37h），这两个寄存器用于设置喇叭音量，同 R52，R53 设置一模一样，这里就不细说了。

以上，就是我们用 WM8978 播放音乐时的设置，按照以上所述，对各个寄存器进行相应的配置，即可使用 WM8978 正常播放音乐了。还有其他一些 3D 设置，EQ 设置等，我们这里就不再介绍了，大家参考 WM8978 的数据手册自行研究下即可。

52.1.3 SAI 简介

STM32F767 自带了两个串行音频接口（SAI1 和 SAI2），SAI 具有灵活性高、配置多样的

特点。可以支持：I2S 标准、LSB 或 MSB 对齐、PCM/DSP、TDM 和 AC’97 等协议，适用于多声道或单声道应用。

SAI 通过两个完全独立的音频子模块来实现这种灵活性与可配置性，每个音频子模块与多达 4 个引脚（SD、SCK、FS 和 MCLK）相连。如果将两个子模块声明为同步模块，则其中一些引脚可以共用，从而可释放一些引脚用作通用 I/O。MCLK 引脚是否用作输出引脚取决于实际应用和解码的要求以及音频模块是否配置为主模块。SAI 可以配置为主模式或配置为从模式。音频子模块既可作为接收器，又可作为发送器；既可与另一模块同步，又可以不同步。

STM32F767 自带的 SAI 接口特点有：

●具有两个独立的音频子模块，子模块既可作为接收器，也可作为发送器，并自带 FIFO

●每个音频子模块集成多达 8 个字，每个字 32 位的 FIFO

●两个音频子模块间可以是同步或异步模式

●两个音频子模块的主/从配置相互独立

●当两个音频子模块都配置为主模式时，每个子模块可设置互相独立的采样率

●数据大小可配置：8 位、10 位、16 位、20 位、24 位或 32 位

●支持：I2S、LSB 或 MSB 对齐、PCM/DSP、TDM 和 AC’97 等音频协议

●高达 16 个大小可配置的 Slot，可选择音频帧中的哪些 Slot 有效

●支持 LSB 或 MSB 数据传输

●支持 DMA，有 2 个专用通道，用于处理对每个 SAI 音频子模块的专用集成 FIFO 的访问

STM32F767 的 SAI 框图如图 52.1.3.1 所示：

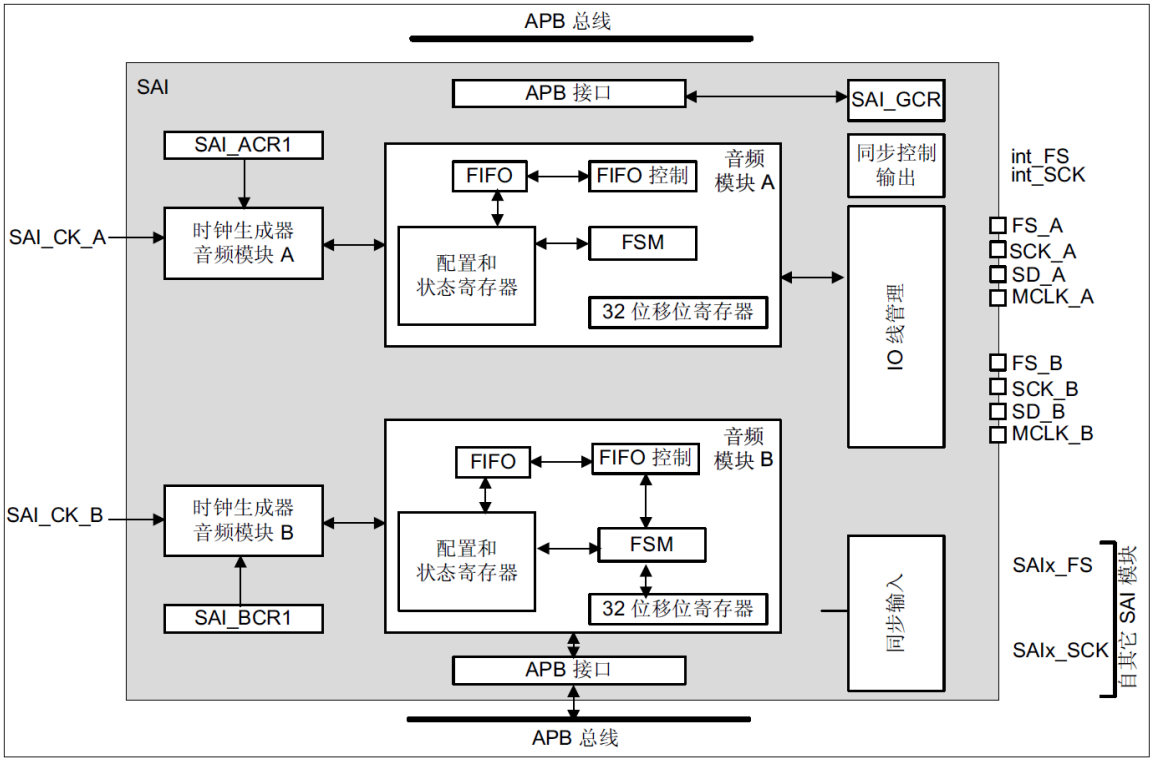


图 52.1.3.1 SAI 框图

本章，我们将用 SAI 接口来驱动 WM8978，而 WM8978 的接口是 I2S 接口的，所以，本章我们只介绍 SAI 支持 I2S 协议使用的方法，其他协议的使用介绍，请看《STM32F7 中文参考手册.pdf》第 33 章。

（1）SAI I2S 信号线

SAI 作为 I2S 使用的时候，同 I2S 接口连接的信号线如表 52.1.3.1 所示：

信号线	说明
FS_A/B	通道识别信号，连接 I2S 的左/右对齐时钟信号(LRC)
SCK_A/B	位时钟信号，连接 I2S 的位时钟信号(BLCK)
SD_A/B	数据输入/输出脚，连接 I2S 的数据输出/输入脚(DACDAT/ADCDAT)
MCLK_A/B	主时钟信号，连接 I2S 的的主时钟引脚(MCLK)

表 52.1.3.1 SAI 同 I2S 接口连接关系表

表中，A/B 表示 SAI 内部的两个独立的音频子模块，可以独立的连接 I2S，也可以共同连接同一个 I2S（主从同步模式），主从同步模式，常用于全双工 I2S 通信（读/写同时进行），主从同步模式，还可以省略一些信号线（SCK/FS/MCLK 等）。

FS_A/B：连接 I2S 的 LRC 脚，用于切换左右声道的数据，它的频率等于音频信号采样率（fs）。

SCK_A/B：连接 I2S 的 BLCK 脚，用作位时钟，是 I2S 主模式下的串行时钟输出以及从模式下的串行时钟输入。SCK_A/B 频率= FS_A/B 频率（fs）*slot 个数*单个 slot 大小（slot 后面介绍）。

SD_A/B：连接 I2S 的 DACDAT/ADCDAT 脚，是数据输入/输出脚，用于发送或接收数据（单个音频子模块，只能做半双工通信。全双工需要 2 个音频子模块同时工作，使用主从同步模式）。

MCLK_A/B：连接 I2S 的 MCLK 脚，是主时钟输出脚，固定输出频率为 256×fs，fs 即音频信号采样频率（fs）。

（2）SAI slot 简介

slot 是 SAI 音频帧中的基本元素，音频帧中 slot 的数目通过 SAI_xSLOTR 寄存器配置，每个音频帧的 slot 数，最大是 16。在 I2S 模式下，SAI 中 slot 的传输方式如图 52.1.3.2 所示：

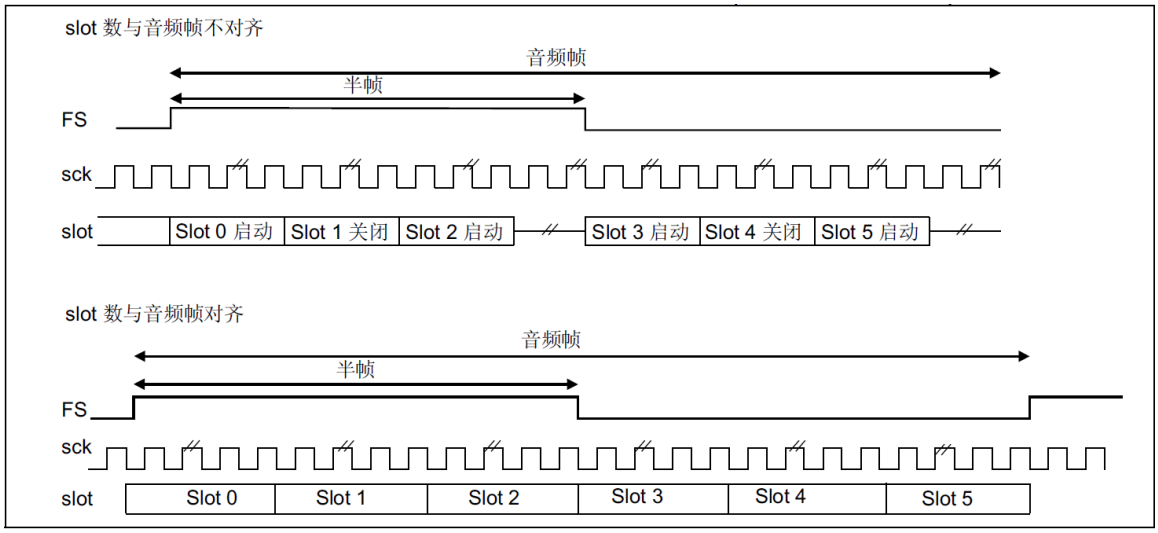


图 52.1.3.2 SAI I2S 模式下 slot 传输示意图

上图中，一个音频帧中，slot 的个数为 6 个，每个半帧有 3 个 slot，根据 slot 数与音频帧的对齐与否又分为两种情况，我们一般设计为 slot 数与音频帧对齐，也就是图 52.1.3.2 中下部分图所示的传输方式：一个半帧刚好是 3 个 slot，每个 slot 可以传输一个声道的音频数据，这样，6 个 slot 就可以传输 6 个声道的音频数据。一般音频文件都是立体声，所以只需要 2 个 slot 即可，每个半帧一个 slot。STM32 的 SAI 最多可以实现 16 声道数据传输（16 个 slot）。

每个 slot 的大小是可以配置的，如图 52.1.3.3 所示：

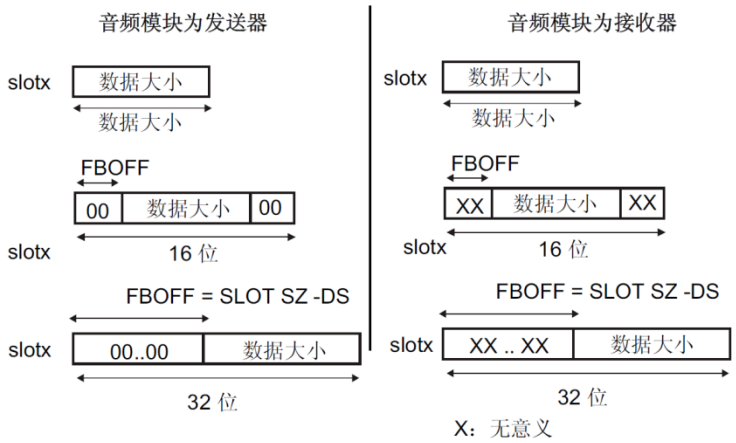


图 52.1.3.3 slot 大小配置

由图可知，数据大小（DS）可以和 slot 相等，也可以不相等（16bit/32bit），当数据大小小于 slot 大小的时候，可以通过 SAI_xSLOTR 寄存器的 FBOFF 位设置数据的偏移。各种设置的约束条件为：

$$FBOFF \leqslant (SLOTSZ-DS)$$

$$DS \leqslant SLOTSZ$$

$$NBSLOT*SLOTSZ \leqslant FRL$$

其中：FBOFF 为数据在 slot 里面的偏移量；SLOTSZ 为单个 slot 的位数；DS 为数据大小位数；NBSLOT 为一帧中 slot 的个数；FRL 为帧长度（位数）。

在 I2S 模式下，我们配置 slot 大小为 32 位，每一帧 slot 个数为 2 个，偏移量为 0，这样就可以支持 16~32 位的立体声音乐播放了。

（3）SAI 时钟发生器

SAI 每个音频子模块都有自己的时钟发生器，这样两个模块完全独立，可以同时工作，并互不干扰。当音频模块定义为主模块时，时钟发生器将产生位时钟（SCK）以及用于外部解码器的主时钟（MCLK），当音频模块定义为从模块时，时钟发生器将关闭（关 SCK 和 MCLK）。SAI 的时钟发生器架构如图 52.1.3.4 所示：

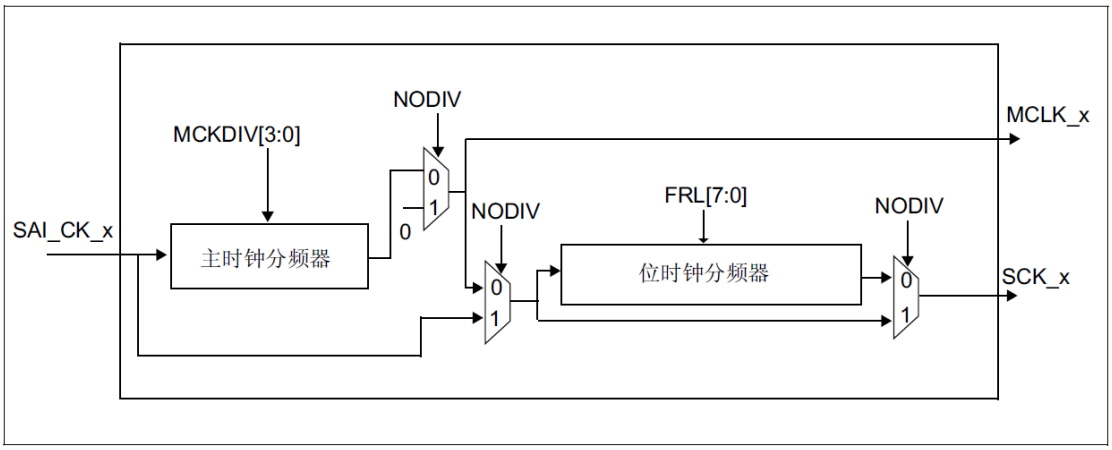


图 52.1.3.4 SAI 时钟发生器架构

图中，NODIV，可以用于控制是否使能分频器，我们一般设置为 0，使能分频器。如果设置为 1，那么分频器将关闭（主分频器和位时钟分频器都关闭），MCLK_x（x=A/B，下同）将无输出，而 SCK_x 则等于 SAI_CK_x。

SAI_CK_x 时钟来自 PLLSAI 或 PLLI2S 的 Q 分频输出，随后经过主时钟分频器（MCKDIV）

分频，作为主时钟（MCLK）提供给 WM8978，同时，经主分频（MCKDIV）分频后，还会经由位时钟分频器（FRLP[7:0]）分频，作为位时钟（SCK）提供给 WM8978。

当 MCKDIV[3:0] !=0000 的时候，MCLK_x=SAI_CK_x/(MCKDIV[3:0]*2)

当 MCKDIV[3:0] ==0000 的时候，MCLK_x=SAI_CK_x

位时钟（SCK）计算公式：SCK_x=MCLK_x*(FRL[7:0]+1)/256

其中：256 是 MCLK 和音频采样率之间的固定比率（MCLK 恒等于 fs*256）;FRL[7:0]是音频帧中的位时钟-1（在 I2S 协议下，必须是奇数，+1 后为偶数）；因此，我们可以得到音频采样率（fs）的计算公式为：

当 MCKDIV[3:0] !=0000 的时候：fs= SAI_CK_x/(MCKDIV[3:0]*512)

当 MCKDIV[3:0] ==0000 的时候：fs= SAI_CK_x/(256)

其中：SAI_CK_x 来自 PLLI2S/PLLSAI 的 Q 分频，以来自 PLLI2S 为例，计算公式为：

SAI_CK_x=(HSE/pllm)*PLLI2SN/PLLI2SQ/(PLLI2SDIVQ+1)

HSE 我们是 25Mhz，而 pllm 在系统时钟初始化就确定了，是 25，这样结合以上公式，可得 fs 的计算公式如下：

MCKDIV[3:0] !=0000 时：fs= 1000*PLLI2SN/PLLI2SQ/(PLLI2SDIVQ+1)/(MCKDIV*512)

MCKDIV[3:0] ==0000 时：fs=1000*PLLI2SN/PLLI2SQ/(PLLI2SDIVQ+1)/(256)

fs 单位是：Khz。其中：PLL2SN 取值范围：192~432；PLLI2SQ 取值范围：2~15；PLLI2SDIVQ 取值范围：0~31；MCKDIV 的值范围：0~15。根据以上约束条件，我们便可以根据 fs 来设置各个系数的值了，不过很多时候，并不能取得和 fs 一模一样的频率，只能近似等于 fs，比如 44.1Khz 采样率，我们设置 PLL2SN=429，PLL2SQ=2，PLLI2SDIVQ=18，MCKDIV=0，得到 fs=44.0995Khz，误差为：0.0011%。晶振频率决定了有时无法通过分频得到我们所要的 fs，所以，某些 fs 如果要实现 0 误差，大家必须得选用外部时钟才可以。

如果要通过程序去计算这些系数的值，是比较麻烦的，所以，我们事先计算好常用 fs 对应的系数值，建立一个表，这样，用的时候，只需要查表取值就可以了，大大简化了代码，常用 fs 对应系数表如下：

```
//表格式:采样率/10,PLLI2SN,PLLI2SQ, PLLI2SDIVQ, MCKDIV
const u16 SAI_PSC_TBL[][5]=
{
    {800 ,344,7,0,12},  //8Khz 采样率
    {1102,429,2,18,2}, //11.025Khz 采样率
    {1600,344,7, 0,6},  //16Khz 采样率
    {2205,429,2,18,1}, //22.05Khz 采样率
    {3200,344,7, 0,3},  //32Khz 采样率
    {4410,429,2,18,0}, //44.1Khz 采样率
    {4800,344,7, 0,2},  //48Khz 采样率
    {8820,271,2, 2,1},  //88.2Khz 采样率
    {9600,344,7, 0,1},  //96Khz 采样率
    {17640,271,6,0,0}, //176.4Khz 采样率
    {19200,295,6,0,0}, //192Khz 采样率
};
```

有了上面的 fs-系数对应表，我们可以很方便的完成 SAI 的时钟配置。

（4）SAI 相关寄存器

接下来，我们看看本章需要用到的一些相关寄存器。

首先，是 SAI 配置寄存器 1：SAI_xCR1(x=A/B，下同)，该寄存器各位描述如图 52.1.3.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								MCKDIV[3:0]				NODIV	Res.	DMAEN	SAIxEN
								rw	rw	rw	rw	rw		rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		OutDriv	MONO	SYNCEN[1:0]		CKSTR	LSBFIRST	DS[2:0]			Res.	PRTCFG[1:0]		MODE[1:0]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

图 52.1.3.5 寄存器 SAI_xCR1 各位描述

该寄存器的配置，需要在禁止 SAI 的状况下配置，接下来，看看本章我们需要用到的各位的描述：

MODE[1:0]位：00，主发送器；01，主接收器；10，从发送器；11，从接收器。我们用来播放音乐，设置为 00，就可以了。

PRTCFG[1:0]位：00，自由协议（I2S/LSB/MSB/TDM/PCM/DSP 等）；10，AC’97 协议。我们使用 I2S 协议，需要设置为 00。

DS[2:0]位：010~111，表示 8/10/16/20/24/32 位数据大小，我们使用的音频一般是 16/24 位，所以设置这三个位为：100（16 位）或 110（24 位）。

LSBFIRST 位：控制数据传输时是 MSB 还是 LSB，I2S 为 MSB，我们设置该位为 0。

CKSTR 位：设置时钟选通边沿，我们设置为 1，即数据在时钟的上升沿选通。

SYNCEN[1:0]位：00，音频模块异步工作；01，音频模块与另外一个音频模块同步。我们要控制 WM8978 播放音乐，需要设置音频模工作在异步模式，即设置 SYNCEN[1:0]=00。

MONO 位：用于设置单声道/立体声模式，我们设置为 0，工作在立体声模式。

OUTDIV 位：0，当 SAIEN 置 1 时，驱动音频模块输出；1，在该位设置为 1 后，立即驱动音频模块输出。这里我们设置为 1。

SAIxEN 位：0，禁止音频模块；1，使能音频模块；注意：必须在所有 SAI 配置完成以后，才设置该位为 1。

DMAEN 位：DMA 使能位，0，禁止 DMA；1，使能 DMA；我们设置为 1，使能 DMA。

NODIV 位：0，使能主时钟和位时钟分频器；1，禁止主时钟和位时钟分频器；我们一般设置为 0。

MCKDIV[3:0]：主时钟分频器，当设置为 0000 时，表示 1 分频；其他情况，则分频值为：MCKDIV[3:0]*2；我们需要根据音频采样率(fs)的不同来设置不同的值。

第二个是 SAI 帧配置寄存器：SAI_xFRCR，该寄存器各位描述如图 52.1.3.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved													FSOFF	FSPOL	FSDEF
													rw	rw	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	FSALL[6:0]							FRL[7:0]							
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 52.1.3.6 寄存器 SAI_xFRCR 各位描述

FRL[7:0]位：帧长度设置位，它等于音频帧中 SCK 的个数-1。FRL 的最小值为 8，最大值为 256，且 FRL+1 应该为偶数，并且是 2 的指数倍关系。

FSALL[6:0]位：帧同步有效电平长度，用于指定 FS 信号的有效电平长度，即高电平/低电平的宽度，应该等于帧长度的一半。计算方法为：FSALL=(FRL+1)/2-1。

FSDEF 位：帧同步定义，0，FS 信号为起始帧信号；1，FS 信号为 SOF 信号+通道识别信号。使用 I2S 协议的时候，我们设置 FS 为 1。

FSPOL 位：帧同步极性设置，0，FS 低电平有效（下降沿）；1，FS 高电平有效（上升沿）；我们设置为 FS 低电平有效，即 FSPOL 位为 0。

FSOFF：帧同步偏移，0，在 slot0 的第一位上使能 FS；1，在 slot0 的第一位的前一位上使能 FS。使用 I2S 协议时，需要设置 FSOFF 位为 1，以匹配 I2S 协议（见图 52.1.2.1）。

第三个是 SAI slot 寄存器：SAI_xSLOTR，该寄存器各位描述如图 52.1.3.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SLOTEN[15:0]															
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
Reserved				NBSLOT[3:0]				SLOTSZ[1:0]		Res	FBOFF[4:0]				
				rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw

图 52.1.3.7 寄存器 SAI_xSLOTR 各位描述

FBOFF[4:0]位：设置第一个位的偏移量，用于设置 slot 中第一个数据传输位的位置，它表示一个偏移值。由于前面我们设置了 FSOFF 位，所以，我们设置 FBOFF=0 即可。

SLOTSZ[1:0]位：设置 slot 大小，00，slot 大小等于数据大小；01，16 位；10，32 位；我们设置 SLOTSZ 为 10（32 位），以支持最高 32 位音频的播放。

NBSLOT[3:0]位：设置音频帧中 slot 的个数（设置值+1）。比如我们用立体声，使用 2 个 slot 就够了，所以设置 NBSLOT=1 即可。

SLOTEN[15:0]位：设置 slot 使能，每个位表示 1 个 slot，最多是 16 个 slot。我们使用 2 个 slot（即 slot0 和 slot1），所以，设置 SLOTEN 的最低 2 位为 1 即可。

第四个是 PLLI2S 配置寄存器：RCC_PLLI2SCFGR，该寄存器各位描述如图 52.1.3.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	PLLI2S[2:0]			PLLI2SQ[0:3]				Res.	Res.	Res.	Res.	Res.	Res.	PLLI2SP[1:0]	
	rw	rw	rw	rw	rw	rw	rw							rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	PLLI2SN[8:0]									Res.	Res.	Res.	Res.	Res.	Res.
	rw	rw	rw	rw	rw	rw	rw	rw	rw						

图 52.1.3.8 寄存器 RCC_PLLI2SCFGR 各位描述

该寄存器用于配置 PLLI2SQ 和 PLLI2SN 两个系数，PLLI2SQ 的取值范围是：2~15，PLLI2SN 的取值范围是：49~432。同样，这两个也是根据 fs 的值来设置的。

第五个是 RCC 专用时钟配置寄存器 1：RCC_DCKCFGR1，该寄存器各位描述如图 52.1.3.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TIMPRE	SAI2SEL[1:0]		SAI1SEL[1:0]		Res.	Res.	PLLSAIDIVR[1:0]	
							rw	rw	rw	rw	rw			rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	PLLSAIDIVQ[4:0]					Res.	Res.	Res.	PLLI2SDIVQ[4:0]				
			rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

图 52.1.3.9 寄存器 RCC_DCKCFGR1 各位描述

该寄存器用于配置 SAI1 和 SAI2 的时钟源（SAI1SEL[1:0]和 SAI2SEL[1:0]），以及分频系数（PLLSAIDIVQ 和 PLLI2SDIVQ）。我们使用 SAI 1 来驱动 WM8978，且时钟源为 PLLI2S，所以，需要设置 SAI1SEL[1:0]为 01，得到 SAI1_CK_x=PLLI2S_Q/PLLI2SDIVQ，其中，PLLI2S_Q 和 PLLI2SDIVQ 是根据 fs 的值来设置的。

第六个是 SAI 数据寄存器：SAI_xDR，该寄存器各位描述如图 52.1.3.10 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA[31:16]															
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
DATA[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 **DATA[31:0]**：数据 (Data)
若 FIFO 未满，写入该寄存器的效果是向 FIFO 加载数据。
若 FIFO 非空，读取该寄存器的效果是从 FIFO 取走数据。

图 52.1.3.10 寄存器 SAI_xDR 各位描述

当我们需要向 WM8978 发送音频数据的时候，通过写这个寄存器，就可以实现。不过，我们采用 DMA 来传输，所以直接设置 DMA 的外设地址为 SAI_xDR 即可。

此外，使用 FIFO 的时候，还要用到 SAI_xCR2 寄存器设置 FIFO 阈值和刷新，我们这里就不做多的介绍了，请大家参考《STM32F7 中文参考手册.pdf》第 33.3.8 节。

SAI 的相关寄存器，就给大家介绍到这里。

（5）SAI 初始化步骤

最后，我们看看要通过 STM32F7 的 SAI，驱动 WM8978 播放音乐的简要步骤。SAI 相关的库函数定义和声明分布在源文件 stm32f7xx_hal_sai.c/stm23f7xx_hal_sai_ex.c 以及头文件 stm32f7xx_hal_sai.h 中。具体步骤如下：

1）初始化 WM8978

这个过程就是在 50.1.2 节最后那十几个寄存器的配置，包括软复位、DAC 设置、输出设置和音量设置等。

2）初始化 SAI

此过程主要设置 SAI_xCR1、SAI_xFRCR 和 SAI_xSLOTR 等寄存器，设置 SAI 工作模式、协议、时钟电平特性、slot 相关参数等。HAL 库 SAI 初始化函数为：HAL_SAI_Init，声明如下：

```
HAL_StatusTypeDef HAL_SAI_Init(SAI_HandleTypeDef *hsai);
```

该函数只有一个入口参数 hsai，该参数为 SAI_HandleTypeDef 结构体指针类型。SAI_HandleTypeDef 结构体定义如下：

```
typedef struct __SAI_HandleTypeDef
{
  SAI_Block_TypeDef        *Instance;
  SAI_InitTypeDef          Init;
  SAI_FrameInitTypeDef     FrameInit;
  SAI_SlotInitTypeDef      SlotInit;
  uint8_t                  *pBuffPtr;
  uint16_t                 XferSize;
  uint16_t                 XferCount;
  DMA_HandleTypeDef        *hdmatx;
  DMA_HandleTypeDef        *hdmarx;
  SAIcallback              mutecallback;
  void (*InterruptServiceRoutine)(struct __SAI_HandleTypeDef *hsai);
  HAL_LockTypeDef          Lock;
```

```
    __IO HAL_SAI_StateTypeDef State;
    __IO uint32_t                 ErrorCode;
}SAI_HandleTypeDef;
```

该结构体成员变量比较多，大致会分为如下几种：

第一种是初始化结构体变量 Init，FrameInit，和 SlotInit，这三个成员变量都是结构体类型，分别用来初始化 SAI 的工作模式，协议，时钟电平特性和 slot 相关参数。

第二种是 HAL 库中处理 SAI 接口通信的数据指针 pBuffPtr，传输数据大小 XferSize 和剩余数据量 XferCount 三个变量，这和串口通信很相似。

第三种是 hdmatx 和 hdmarx，为 DMA_HandleTypeDef 结构体指针类型，指向 DMA 句柄。

第四种是回调函数 mutecallback 和 InterruptServiceRoutine。

第五种是 HAL 库中间过程变量。

这里我们主要讲解 Init，FrameInit，和 SlotInit 三个初始化结构体变量。

成员变量 Init 是 SAI_InitTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
  uint32_t AudioMode;       //音频模块模式 主/从 发送/接收 器
  uint32_t Synchro;         //同步使能：异步/同步
  uint32_t SynchroExt;
  uint32_t OutputDrive;//输出驱动：立即驱动音频模块输出还是当 SAIEN 置 1 后输出
  uint32_t NoDivider;       //主时钟分频器使能/失能
  uint32_t FIFOThreshold;   //FIFO 阈值
  uint32_t ClockSource;     //SAI 时钟源选择
  uint32_t AudioFrequency; //音频频率
  uint32_t Mckdiv;          //主时钟分频器系数
  uint32_t MonoStereoMode; //模式：单声道还是立体声
  uint32_t CompandingMode;//压扩模式设置
  uint32_t TriState; //数据线的三态管理
  uint32_t Protocol;//协议配置：自由协议还是 AC’97 协议
  uint32_t DataSize;//数据大小：8/10/16/20/24/32 位
  uint32_t FirstBit;// MSB 还是 LSB 在先
  uint32_t ClockStrobing; //时钟选通边沿，SCK 上升沿还是下降沿
}SAI_InitTypeDef;
```

该结构体主要用来配置 SAI_xCR1 寄存器，各个成员变量含义我们已经注释了，如有不理解的地方请参考 SAI_xCR1 寄存器定义。

成员变量 FrameInit 是 SAI_FrameInitTypeDef 结构体类型，用来进行帧设置，主要是配置 SAI_xFRCR 寄存器。结构体 SAI_FrameInitTypeDef 定义为：

```
typedef struct
{
  uint32_t FrameLength;         //帧长度
  uint32_t ActiveFrameLength;   //帧同步有效电平长度
  uint32_t FSDefinition;        //帧同步定义
  uint32_t FSPolarity;          //帧同步极性设置
  uint32_t FSOffset;            //帧同步偏移设置
```

```
}SAI_FrameInitTypeDef;
```

该结构体各个成员变量含义就比较好理解了，在讲解 SAI_xFRCR 寄存器的时候都有提到，这里我们同样也在每个成员变量后面注释了。

成员变量 SlotInit 是 SAI_SlotInitTypeDef 结构体类型，用来进行 SLOT 设置，主要是配置 SAI_xSLOTR 寄存器。结构体 SAI_SlotInitTypeDef 定义为：

```
typedef struct
{
  uint32_t FirstBitOffset;     //第一个位的偏移量
  uint32_t SlotSize;           //设置 slot 大小
  uint32_t SlotNumber;         //设置音频帧中 slot 个数
  uint32_t SlotActive;         //设置 slot 使能
}SAI_SlotInitTypeDef;
```

该结构体各个成员变量含义同样比较好理解，我们都有注释，不理解的地方请参考寄存器 SAI_xSLOTR 各位定义。

关于 HAL_SAI_Init 的使用实例这里基于篇幅考虑我们就不列出来了，详情请参考软件设计小节源码。

HAL 库同样提供了 SAI 初始化 MSP 回调函数 HAL_SAI_MspInit，定义如下：

```
void HAL_SAI_MspInit(SAI_HandleTypeDef *hsai);
```

关于回调函数使用方法，这里我们就不做过多讲解了。

3）解析 WAV 文件，获取音频信号采样率和位数并设置 SAI 时钟分频器

这里，要先解析 WAV 文件，取得音频信号的采样率（fs）和位数（16 位或 24 位），根据这两个参数，来设置 SAI 的时钟分频，这里我们用前面介绍的查表法来设置即可。我们在设置完采样率和时钟分频后，便可以使能 SAI 了。

4）设置 DMA

SAI 播放音频的时候，一般都是通过 DMA 来传输数据的，所以必须配置 DMA，本章我们用 SAI 的子模块 A，其 TX 是使用的 DMA2 数据流 3 的通道 0 来传输的。并且，STM32F7 的 DMA 具有双缓冲机制，这样可以提高效率，大大方便了我们的数据传输，本章将 DMA2 数据流 3 设置为：双缓冲循环模式，外设和存储器宽度相同（16 位/32 位），并开启 DMA 传输完成中断（方便填充数据）。DMA 配置过程请参考实验源码，并对照第二十八章 DMA 实验讲解学习。

5）编写 DMA 传输完成中断服务函数

为了方便填充音频数据，我们使用 DMA 传输完成中断，每当一个缓冲数据发送完后，硬件自动切换为下一个缓冲，同时进入中断服务函数，填充数据到发送完的这个缓冲。过程如图 50.1.3.11 所示：

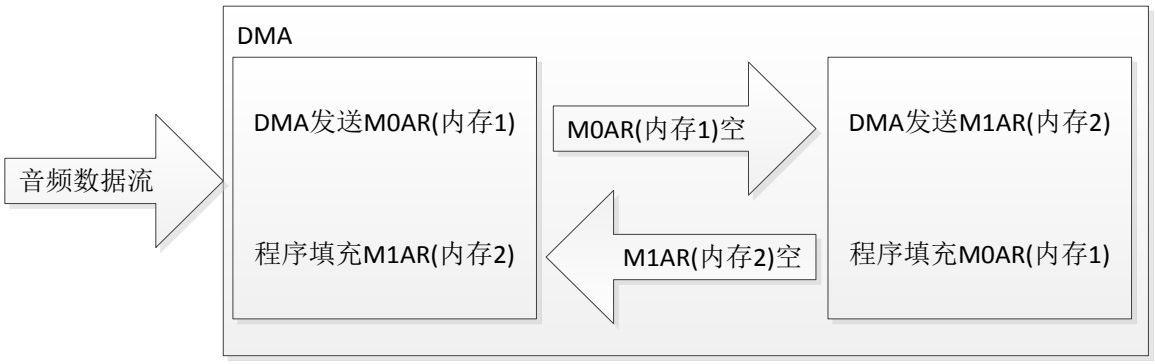


图 50.1.3.11 DMA 双缓冲发送音频数据流框图

6）开启 DMA 传输，填充数据

最后，我们就只需要开启 DMA 传输，然后及时填充 WAV 数据到 DMA 的两个缓存区即可。此时，就可以在 WM8978 的耳机和喇叭通道听到所播放音乐了。

52.2 硬件设计

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，则开始循环播放 SD 卡 MUSIC 文件夹里面的歌曲（必须在 SD 卡根目录建立一个 MUSIC 文件夹，并存放歌曲（仅支持 wav 格式）在里面），在 TFTLCD 上显示歌曲名字、播放时间、歌曲总时间、歌曲总数目、当前歌曲的编号等信息。KEY0 用于选择下一曲，KEY2 用于选择上一曲，KEY_UP 用来控制暂停/继续播放。DS0 还是用于指示程序运行状态。

本实验用到的资源如下：

1） 指示灯 DS0
2） 三个按键（KEY_UP/KEY0/KEY1）
3） 串口
4） LCD 模块
5） SD 卡
6） SPI FLASH
7） WM8978
8） SAI

这些硬件我们都已经介绍过了，不过 WM8978 和 STM32F767 的连接，还没有介绍，连接如图 52.2.1 所示：

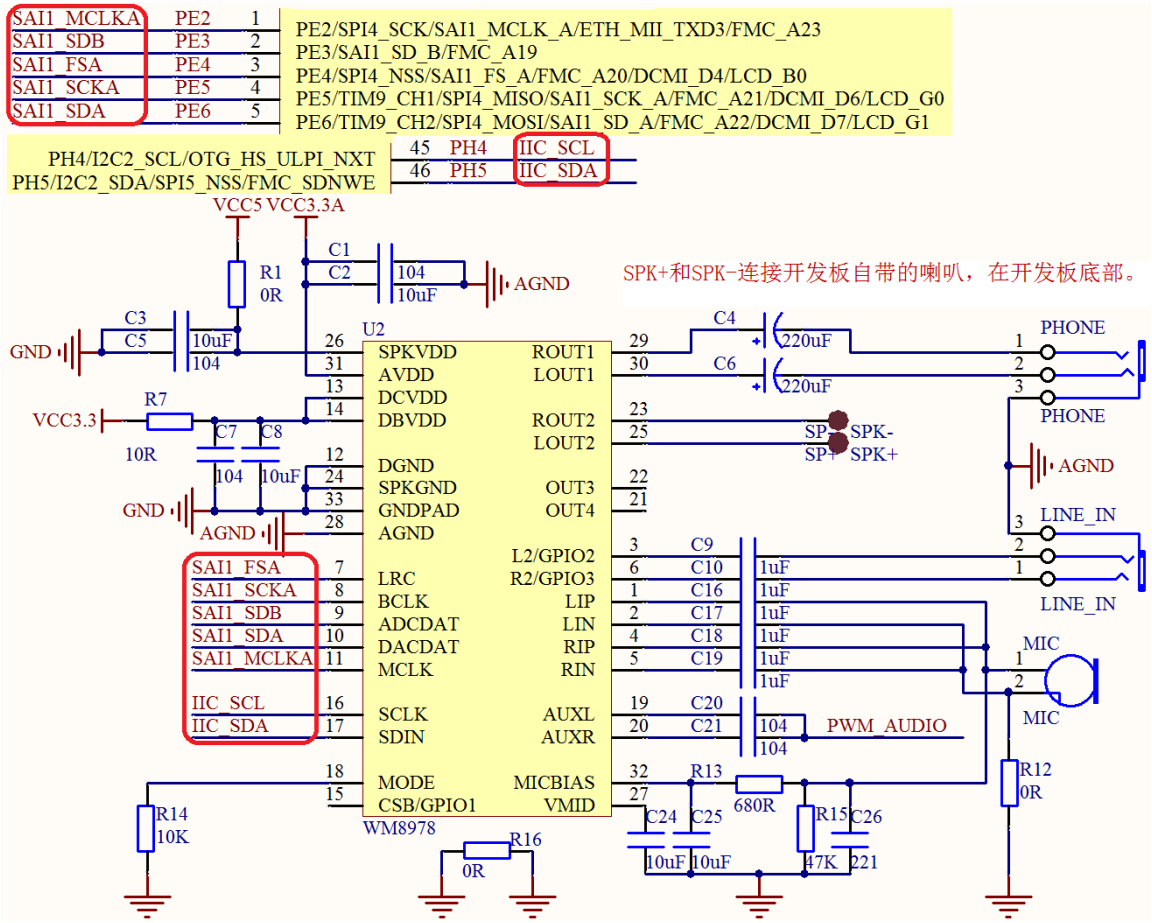


图 52.2.1 WM8978 与 STM32F767 连接原理图

图中，PHONE 接口，可以用来插耳机，SPK+和 SPK-连接了板载的喇叭（在开发板底部）。硬件上，IIC 接口和 24C02 等芯片共用。

本实验，大家需要准备 1 个 SD 卡（在里面新建一个 MUSIC 文件夹，并存放一些 wav 歌曲在 MUSIC 文件夹下），然后下载本实验就可以听歌了。

52.3 软件设计

打开本章实验工程可以看到，我们在工程中新建了 AUDIOCODEC 分组和 APP 分组，分别添加了 wavplay.c 文件和 audioplay.c 文件。同时在 HARDWARE 分组之下添加了 wm8978.c 文件和 sai.c 文件。

本章代码比较多，我们就不全部贴出来给大家介绍了，这里仅挑一些重点函数给大家介绍下。首先是 sai.c 里面，重点函数代码如下：

```
//SAI Block A 初始化,I2S,飞利浦标准
//mode:工作模式,可以设置:SAI_MODEMASTER_TX/
    //SAI_MODEMASTER_RX/SAI_MODESLAVE_TX/SAI_MODESLAVE_RX
//cpol:数据在时钟的上升/下降沿选通，可以设置
    //SAI_CLOCKSTROBING_FALLINGEDGE/SAI_CLOCKSTROBING_RISINGEDGE
//datalen:数据大小,可以设置：SAI_DATASIZE_8/10/16/20/24/32
void SAIA_Init(u32 mode,u32 cpol,u32 datalen)
{
```

```
    HAL_SAI_DeInit(&SAI1A_Handler);         //清除以前的配置
    SAI1A_Handler.Instance=SAI1_Block_A;  //SAI1 Bock A
    SAI1A_Handler.Init.AudioMode=mode;      //设置 SAI1 工作模式
    SAI1A_Handler.Init.Synchro=SAI_ASYNCHRONOUS;    //音频模块异步
    SAI1A_Handler.Init.OutputDrive=SAI_OUTPUTDRIVE_ENABLE;
                                                    //立即驱动音频模块输出
    SAI1A_Handler.Init.NoDivider=SAI_MASTERDIVIDER_ENABLE;//使能主时钟分频器
    SAI1A_Handler.Init.FIFOThreshold=SAI_FIFOTHRESHOLD_1QF;
                                                        //设置 FIFO 阈值,1/4 FIFO
    SAI1A_Handler.Init.ClockSource=SAI_CLKSOURCE_PLLI2S; //SIA 时钟源为 PLL2S
    SAI1A_Handler.Init.MonoStereoMode=SAI_STEREOMODE;     //立体声模式
    SAI1A_Handler.Init.Protocol=SAI_FREE_PROTOCOL; //设置 SAI1 协议为:自由协议
    SAI1A_Handler.Init.DataSize=datalen;                    //设置数据大小
    SAI1A_Handler.Init.FirstBit=SAI_FIRSTBIT_MSB;         //数据 MSB 位优先
    SAI1A_Handler.Init.ClockStrobing=cpol;   //数据在时钟的上升/下降沿选通

    //帧设置
    SAI1A_Handler.FrameInit.FrameLength=64;   //设置帧长度为 64,左/右通道 32 个 SCK
    SAI1A_Handler.FrameInit.ActiveFrameLength=32;//设置帧同步有效电平长度
                                                 //在 I2 模式下=1/2 帧长.
    SAI1A_Handler.FrameInit.FSDefinition=SAI_FS_CHANNEL_IDENTIFICATION;
                                               //FS 信号为 SOF 信号+通道识别信号
    SAI1A_Handler.FrameInit.FSPolarity=SAI_FS_ACTIVE_LOW; //FS 低电平有效(下降沿)
    SAI1A_Handler.FrameInit.FSOffset=SAI_FS_BEFOREFIRSTBIT; //在 slot0 的第一位的
                                                   //前一位使能 FS,以匹配飞利浦标准
    //SLOT 设置
    SAI1A_Handler.SlotInit.FirstBitOffset=0;                    //slot 偏移(FBOFF)为 0
    SAI1A_Handler.SlotInit.SlotSize=SAI_SLOTSIZE_32B;   //slot 大小为 32 位
    SAI1A_Handler.SlotInit.SlotNumber=2;                        //slot 数为 2 个
    SAI1A_Handler.SlotInit.SlotActive=SAI_SLOTACTIVE_0|SAI_SLOTACTIVE_1;
                                                          //使能 slot0 和 slot1
    HAL_SAI_Init(&SAI1A_Handler);                           //初始化 SAI
    __HAL_SAI_ENABLE(&SAI1A_Handler);                       //使能 SAI
}

void HAL_SAI_MspInit(SAI_HandleTypeDef *hsai)
{
 ……//省略 IO 口初始化代码
}
const u16 SAI_PSC_TBL[][5]=
{
    ……//省略代码，见 50.1.3 节
};
```

```
//开启 SAI 的 DMA 功能,HAL 库没有提供此函数
//因此我们需要自己操作寄存器编写一个
void SAIA_DMA_Enable(void)
{
    u32 tempreg=0;
    tempreg=SAI1_Block_A->CR1;              //先读出以前的设置
    tempreg|=1<<17;                         //使能 DMA
    SAI1_Block_A->CR1=tempreg;              //写入 CR1 寄存器中
}

//设置 SAIA 的采样率(@MCKEN)
//samplerate:采样率,单位:Hz
//返回值:0,设置成功;1,无法设置.
u8 SAIA_SampleRate_Set(u32 samplerate)
{
    u8 i=0;
    RCC_PeriphCLKInitTypeDef RCCSAI1_Sture;

    for(i=0;i<(sizeof(SAI_PSC_TBL)/10);i++)//看看改采样率是否可以支持
    {
        if((samplerate/10)==SAI_PSC_TBL[i][0])break;
    }
    if(i==(sizeof(SAI_PSC_TBL)/10))return 1;//搜遍了也找不到
    RCCSAI1_Sture.PeriphClockSelection=RCC_PERIPHCLK_SAI1;      //外设时钟源选择
    RCCSAI1_Sture.Sai1ClockSelection=RCC_SAI1CLKSOURCE_PLLSAI;
    RCCSAI1_Sture.PLLSAI.PLLSAIN=(u32)SAI_PSC_TBL[i][1]; //设置 PLLSAIN
    RCCSAI1_Sture.PLLSAI.PLLSAIQ=(u32)SAI_PSC_TBL[i][2]; //设置 PLLSAIQ
    RCCSAI1_Sture.PLLSAIDivQ=SAI_PSC_TBL[i][3];             //设置 PLLSAIDivQ
    HAL_RCCEx_PeriphCLKConfig(&RCCSAI1_Sture);              //设置时钟

    __HAL_SAI_DISABLE(&SAI1A_Handler);          //关闭 SAI
    SAI1A_Handler.Init.AudioFrequency=samplerate;   //设置播放频率
    HAL_SAI_Init(&SAI1A_Handler);                   //初始化 SAI
    SAIA_DMA_Enable();                              //开启 SAI 的 DMA 功能
    __HAL_SAI_ENABLE(&SAI1A_Handler);           //开启 SAI
    return 0;
}
//SAIA TX DMA 配置
//设置为双缓冲模式,并开启 DMA 传输完成中断
//buf0:M0AR 地址.
//buf1:M1AR 地址.
//num:每次传输数据量
//width:位宽(存储器和外设,同时设置),0,8 位;1,16 位;2,32 位;
```

```
void SAIA_TX_DMA_Init(u8* buf0,u8 *buf1,u16 num,u8 width)
{
    u32 memwidth=0,perwidth=0;          //外设和存储器位宽
    switch(width)
    {
        case 0:             //8 位
            memwidth=DMA_MDATAALIGN_BYTE;
            perwidth=DMA_PDATAALIGN_BYTE;
            break;
        case 1:             //16 位
            memwidth=DMA_MDATAALIGN_HALFWORD;
            perwidth=DMA_PDATAALIGN_HALFWORD;
            break;
        case 2:             //32 位
            memwidth=DMA_MDATAALIGN_WORD;
            perwidth=DMA_PDATAALIGN_WORD;
            break;

    }
    __HAL_RCC_DMA2_CLK_ENABLE();        //使能 DMA2 时钟
    __HAL_LINKDMA(&SAI1A_Handler,hdmatx,SAI1_TXDMA_Handler);
                                                    //将 DMA 与 SAI 联系起来
    SAI1_TXDMA_Handler.Instance=DMA2_Stream3;       //DMA2 数据流 3
    SAI1_TXDMA_Handler.Init.Channel=DMA_CHANNEL_0; //通道 0
    SAI1_TXDMA_Handler.Init.Direction=DMA_MEMORY_TO_PERIPH; //存储器到外设
    SAI1_TXDMA_Handler.Init.PeriphInc=DMA_PINC_DISABLE; //外设非增量模式
    SAI1_TXDMA_Handler.Init.MemInc=DMA_MINC_ENABLE; //存储器增量模式
    SAI1_TXDMA_Handler.Init.PeriphDataAlignment=perwidth; //外设数据长度:16/32 位
    SAI1_TXDMA_Handler.Init.MemDataAlignment=memwidth; //存储器数据长度:16/32 位
    SAI1_TXDMA_Handler.Init.Mode=DMA_CIRCULAR; //使用循环模式
    SAI1_TXDMA_Handler.Init.Priority=DMA_PRIORITY_HIGH; //高优先级
    SAI1_TXDMA_Handler.Init.FIFOMode=DMA_FIFOMODE_DISABLE; //不使用 FIFO
    SAI1_TXDMA_Handler.Init.MemBurst=DMA_MBURST_SINGLE;//单次突发传输
    SAI1_TXDMA_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE; //外设突发单次传输
    HAL_DMA_DeInit(&SAI1_TXDMA_Handler);   //先清除以前的设置
    HAL_DMA_Init(&SAI1_TXDMA_Handler);      //初始化 DMA

    HAL_DMAEx_MultiBufferStart(&SAI1_TXDMA_Handler,(u32)buf0,
                        (u32)&SAI1_Block_A->DR,(u32)buf1,num);//开启双缓冲
    __HAL_DMA_DISABLE(&SAI1_TXDMA_Handler);   //先关闭 DMA
    delay_us(10);               //10us 延时，防止-O2 优化出问题
  __HAL_DMA_ENABLE_IT(&SAI1_TXDMA_Handler,DMA_IT_TC);//开启传输完成中断
    __HAL_DMA_CLEAR_FLAG(&SAI1_TXDMA_Handler,DMA_FLAG_TCIF3_7);
```

```
                                                    //清除 DMA 传输完成中断标志位
    HAL_NVIC_SetPriority(DMA2_Stream3_IRQn,0,0);  //DMA 中断优先级
    HAL_NVIC_EnableIRQ(DMA2_Stream3_IRQn);
}
//SAI DMA 回调函数指针
void (*sai_tx_callback)(void);      //TX 回调函数
//DMA2_Stream3 中断服务函数
void DMA2_Stream3_IRQHandler(void)
{
    if(__HAL_DMA_GET_FLAG(&SAI1_TXDMA_Handler,
                                    DMA_FLAG_TCIF3_7)!=RESET) //DMA 传输完成
    {
        __HAL_DMA_CLEAR_FLAG(&SAI1_TXDMA_Handler,DMA_FLAG_TCIF3_7);
                                                    //清除 DMA 传输完成中断标志位
        sai_tx_callback();  //执行回调函数,读取数据等操作在这里面处理
    }
}
//SAI 开始播放
void SAI_Play_Start(void)
{
    __HAL_DMA_ENABLE(&SAI1_TXDMA_Handler);//开启 DMA TX 传输
}
//关闭 I2S 播放
void SAI_Play_Stop(void)
{
    __HAL_DMA_DISABLE(&SAI1_TXDMA_Handler);  //结束播放
}
```

其中，SAIA_Init 完成 SAI_A 子模块的的初始化，通过 3 个参数设置 SAI_A 的详细配置信息。另外一个函数：SAIA_SampleRate_Set，则是用前面介绍的查表法，根据音频采样率来设置 SAI_A 的时钟部分。函数 SAIA_TX_DMA_Init，用于设置 SAI_A 的 DMA 发送，使用双缓冲循环模式，发送数据给 WM8978，并开启了发送完成中断。而 DMA2_Stream3_IRQHandler 函数，则是 DMA2 数据流 3 发送完成中断的服务函数，该函数调用 sai_tx_callback 函数（函数指针，使用前需指向特定函数）实现 DMA 数据填充。函数 SAI_Play_Start 和 SAI_Play_Stop，用于开启和关闭 DMA 传输。

再来看 wm8978.c 里面的几个函数，代码如下：

```
//WM8978 初始化
//返回值:0,初始化正常
//       其他,错误代码
u8 WM8978_Init(void)
{
    u8 res;
    IIC_Init();                         //初始化 IIC 接口
    res=WM8978_Write_Reg(0,0);      //软复位 WM8978
```

```
    if(res)return 1;            //发送指令失败,WM8978 异常
    //以下为通用设置
    WM8978_Write_Reg(1,0X1B);   //R1,MICEN 设置为 1(MIC 使能),BIASEN 设置为 1
    WM8978_Write_Reg(2,0X1B0);  //R2,ROUT1,LOUT1 输出使能(耳机可以工作)
    WM8978_Write_Reg(3,0X6C);   //R3,LOUT2,ROUT2 输出使能(喇叭工作)
    WM8978_Write_Reg(6,0);      //R6,MCLK 由外部提供
    WM8978_Write_Reg(43,1<<4);  //R43,INVROUT2 反向,驱动喇叭
    WM8978_Write_Reg(47,1<<8);  //R47 设置,PGABOOSTL,左通道 MIC 获得 20 倍增益
    WM8978_Write_Reg(48,1<<8);  //R48 设置,PGABOOSTR,右通道 MIC 获得 20 倍增益
    WM8978_Write_Reg(49,1<<1);  //R49,TSDEN,开启过热保护
    WM8978_Write_Reg(49,1<<2);  //R49,SPEAKER BOOST,1.5x
    WM8978_Write_Reg(10,1<<3);  //R10,SOFTMUTE 关闭,128x 采样,最佳 SNR
    WM8978_Write_Reg(14,1<<3);  //R14,ADC 128x 采样率
    return 0;
}
//WM8978 DAC/ADC 配置
//adcen:adc 使能(1)/关闭(0)
//dacen:dac 使能(1)/关闭(0)
void WM8978_ADDA_Cfg(u8 dacen,u8 adcen)
{
    u16 regval;
    regval=WM8978_Read_Reg(3);  //读取 R3
    if(dacen)regval|=3<<0;      //R3 最低 2 个位设置为 1,开启 DACR&DACL
    else regval&=~(3<<0);       //R3 最低 2 个位清零,关闭 DACR&DACL.
    WM8978_Write_Reg(3,regval); //设置 R3
    regval=WM8978_Read_Reg(2);  //读取 R2
    if(adcen)regval|=3<<0;      //R2 最低 2 个位设置为 1,开启 ADCR&ADCL
    else regval&=~(3<<0);       //R2 最低 2 个位清零,关闭 ADCR&ADCL.
    WM8978_Write_Reg(2,regval); //设置 R2
}
//WM8978  输出配置
//dacen:DAC 输出(放音)开启(1)/关闭(0)
//bpsen:Bypass 输出(录音,包括 MIC,LINE IN,AUX 等)开启(1)/关闭(0)
void WM8978_Output_Cfg(u8 dacen,u8 bpsen)
{
    u16 regval=0;
    if(dacen)regval|=1<<0;  //DAC 输出使能
    if(bpsen)
    {
        regval|=1<<1;       //BYPASS 使能
        regval|=5<<2;       //0dB 增益
    }
    WM8978_Write_Reg(50,regval);//R50 设置
```

```
    WM8978_Write_Reg(51,regval);//R51 设置
}
//设置 I2S 工作模式
//fmt:0,LSB(右对齐);1,MSB(左对齐);2,飞利浦标准 I2S;3,PCM/DSP;
//len:0,16 位;1,20 位;2,24 位;3,32 位;
void WM8978_I2S_Cfg(u8 fmt,u8 len)
{
    fmt&=0X03;
    len&=0X03;//限定范围
    WM8978_Write_Reg(4,(fmt<<3)|(len<<5)); //R4,WM8978 工作模式设置
}
```

以上代码 WM8978_Init 用于初始化 WM8978，这里只是通用配置（ADC&DAC），初始化之后，并不能正常播放音乐，还需要通过 WM8978_ADDA_Cfg 函数，使能 DAC，然后通过 WM8978_Output_Cfg 选择 DAC 输出，通过 WM8978_I2S_Cfg 配置 I2S 工作模式，最后设置音量才可以接收 I2S 音频数据，实现音乐播放。这里设置音量、EQ、音效等函数，没有贴出了，请大家参考光盘本例程源码。

接下来，看看 wavplay.c 里面的几个函数，代码如下：

```
__wavctrl wavctrl;          //WAV 控制结构体
vu8 wavtransferend=0;   //sai 传输完成标志
vu8 wavwitchbuf=0;          //saibufx 指示标志

//WAV 解析初始化
//fname:文件路径+文件名
//wavx:wav  信息存放结构体指针
//返回值:0,成功;1,打开文件失败;2,非 WAV 文件;3,DATA 区域未找到.
u8 wav_decode_init(u8* fname,__wavctrl* wavx)
{
    FIL*ftemp;
    u8 *buf;
    u32 br=0;
    u8 res=0;

    ChunkRIFF *riff;
    ChunkFMT *fmt;
    ChunkFACT *fact;
    ChunkDATA *data;
    ftemp=(FIL*)mymalloc(SRAMIN,sizeof(FIL));
    buf=mymalloc(SRAMIN,512);
    if(ftemp&&buf)     //内存申请成功
    {
        res=f_open(ftemp,(TCHAR*)fname,FA_READ);//打开文件
        if(res==FR_OK)
        {
```

```
            f_read(ftemp,buf,512,&br);  //读取 512 字节在数据
            riff=(ChunkRIFF *)buf;      //获取 RIFF 块
            if(riff->Format==0X45564157)//是 WAV 文件
            {
                fmt=(ChunkFMT *)(buf+12);//获取 FMT 块
                fact=(ChunkFACT *)(buf+12+8+fmt->ChunkSize);//读取 FACT 块
                if(fact->ChunkID==0X74636166||fact->ChunkID==0X5453494C)
                wavx->datastart=12+8+fmt->ChunkSize+8+fact->ChunkSize;
                                    //具有 fact/LIST 块的时候(未测试)
                else wavx->datastart=12+8+fmt->ChunkSize;
                data=(ChunkDATA *)(buf+wavx->datastart);      //读取 DATA 块
                if(data->ChunkID==0X61746164)//解析成功!
                {
                    wavx->audioformat=fmt->AudioFormat;         //音频格式
                    wavx->nchannels=fmt->NumOfChannels;         //通道数
                    wavx->samplerate=fmt->SampleRate;       //采样率
                    wavx->bitrate=fmt->ByteRate*8;          //得到位速
                    wavx->blockalign=fmt->BlockAlign;       //块对齐
                    wavx->bps=fmt->BitsPerSample;           //位数,16/24/32 位

                    wavx->datasize=data->ChunkSize;             //数据块大小
                    wavx->datastart=wavx->datastart+8;      //数据流开始的地方.

                    printf("wavx->audioformat:%d\r\n",wavx->audioformat);
                    printf("wavx->nchannels:%d\r\n",wavx->nchannels);
                    printf("wavx->samplerate:%d\r\n",wavx->samplerate);
                    printf("wavx->bitrate:%d\r\n",wavx->bitrate);
                    printf("wavx->blockalign:%d\r\n",wavx->blockalign);
                    printf("wavx->bps:%d\r\n",wavx->bps);
                    printf("wavx->datasize:%d\r\n",wavx->datasize);
                    printf("wavx->datastart:%d\r\n",wavx->datastart);
                }else res=3;//data 区域未找到.
            }else res=2;//非 wav 文件

        }else res=1;//打开文件错误
    }
    f_close(ftemp);
    myfree(SRAMIN,ftemp);//释放内存
    myfree(SRAMIN,buf);
    return 0;
}

//填充 buf
```

```
//buf:数据区
//size:填充数据量
//bits:位数(16/24)
//返回值:读到的数据个数
u32 wav_buffill(u8 *buf,u16 size,u8 bits)
{
    u16 readlen=0;
    u32 bread;
    u16 i;
    u32 *p,*pbuf;
    if(bits==24)//24bit 音频,需要处理一下
    {
        readlen=(size/4)*3;        //此次要读取的字节数
        f_read(audiodev.file,audiodev.tbuf,readlen,(UINT*)&bread);//读取数据
        pbuf=(u32*)buf;
        for(i=0;i<size/4;i++)
        {
            p=(u32*)(audiodev.tbuf+i*3);
            pbuf[i]=p[0];
        }
        bread=(bread*4)/3;        //填充后的大小.
    }else
    {
        f_read(audiodev.file,buf,size,(UINT*)&bread);//16bit 音频,直接读取数据
        if(bread<size)//不够数据了,补充 0
        {
            for(i=bread;i<size-bread;i++)buf[i]=0;
        }
    }
    return bread;
}
//WAV 播放时,SAI DMA 传输回调函数
void wav_sai_dma_tx_callback(void)
{
    u16 i;
    if(DMA2_Stream3->CR&(1<<19))
    {
        wavwitchbuf=0;
        if((audiodev.status&0X01)==0)
        {
            for(i=0;i<WAV_SAI_TX_DMA_BUFSIZE;i++)//暂停
            {
                audiodev.saibuf1[i]=0;//填充 0
```

```
                }
            }
        }else
        {
            wavwitchbuf=1;
            if((audiodev.status&0X01)==0)
            {
                for(i=0;i<WAV_SAI_TX_DMA_BUFSIZE;i++)//暂停
                {
                    audiodev.saibuf2[i]=0;//填充 0
                }
            }
        }
        wavtransferend=1;
}
//得到当前播放时间
//fx:文件指针
//wavx:wav 播放控制器
void wav_get_curtime(FIL*fx,__wavctrl *wavx)
{
        long long fpos;
        wavx->totsec=wavx->datasize/(wavx->bitrate/8);//歌曲总长度(单位:秒)
        fpos=fx->fptr-wavx->datastart;                    //得到当前文件播放到的地方
        wavx->cursec=fpos*wavx->totsec/wavx->datasize;    //当前播放到第多少秒了?
}
//播放某个 WAV 文件
//fname:wav 文件路径.
//返回值:
//KEY0_PRES:下一曲
//KEY1_PRES:上一曲
//其他:错误
u8 wav_play_song(u8* fname)
{
        u8 key, t=0,res;
        u32 fillnum;
        audiodev.file=(FIL*)mymalloc(SRAMIN,sizeof(FIL));
        audiodev.saibuf1=mymalloc(SRAMIN,WAV_SAI_TX_DMA_BUFSIZE);
        audiodev.saibuf2=mymalloc(SRAMIN,WAV_SAI_TX_DMA_BUFSIZE);
        audiodev.tbuf=mymalloc(SRAMIN,WAV_SAI_TX_DMA_BUFSIZE);
        if(audiodev.file&&audiodev.saibuf1&&audiodev.saibuf2&&audiodev.tbuf)
        {
            res=wav_decode_init(fname,&wavctrl);//得到文件的信息
            if(res==0)//解析文件成功
```

```
{
    if(wavctrl.bps==16)
    {
        WM8978_I2S_Cfg(2,0);//飞利浦标准,16 位数据长度

        SAIA_Init(SAI_MODEMASTER_TX,
                  SAI_CLOCKSTROBING_RISINGEDGE,SAI_DATASIZE_16);
        SAIA_SampleRate_Set(wavctrl.samplerate);//设置采样率
        SAIA_TX_DMA_Init(audiodev.saibuf1,audiodev.saibuf2,
                  WAV_SAI_TX_DMA_BUFSIZE/2,1); //配置 TX DMA,16 位
    }else if(wavctrl.bps==24)
    {
        WM8978_I2S_Cfg(2,2);//飞利浦标准,24 位数据长度
        SAIA_Init(SAI_MODEMASTER_TX,
                 SAI_CLOCKSTROBING_RISINGEDGE,SAI_DATASIZE_24);
        SAIA_SampleRate_Set(wavctrl.samplerate);//设置采样率
        SAIA_TX_DMA_Init(audiodev.saibuf1,audiodev.saibuf2,
                  WAV_SAI_TX_DMA_BUFSIZE/4,2); //配置 TX DMA,32 位
    }
    sai_tx_callback=wav_sai_dma_tx_callback; //回调函数
    audio_stop();
    res=f_open(audiodev.file,(TCHAR*)fname,FA_READ);    //打开文件
    if(res==0)
    {
        f_lseek(audiodev.file, wavctrl.datastart);          //跳过文件头
        fillnum=wav_buffill(audiodev.saibuf1,
                    WAV_SAI_TX_DMA_BUFSIZE,wavctrl.bps);
        fillnum=wav_buffill(audiodev.saibuf2,
                    WAV_SAI_TX_DMA_BUFSIZE,wavctrl.bps);
        audio_start();
        while(res==0)
        {
            while(wavtransferend==0);//等待 wav 传输完成;
            wavtransferend=0;
            if(fillnum!=WAV_SAI_TX_DMA_BUFSIZE)//播放结束?
            {
                res=KEY0_PRES;
                break;
            }
            if(wavwitchbuf)fillnum=wav_buffill(audiodev.saibuf2,
                  WAV_SAI_TX_DMA_BUFSIZE,wavctrl.bps);//填充 buf2
            else fillnum=wav_buffill(audiodev.saibuf1,
                  WAV_SAI_TX_DMA_BUFSIZE,wavctrl.bps);//填充 buf1
```

```
                while(1)
                {
                    key=KEY_Scan(0);
                    if(key==WKUP_PRES)//暂停
                    {
                        if(audiodev.status&0X01)audiodev.status&=~(1<<0);
                        else audiodev.status|=0X01;
                    }
                    if(key==KEY2_PRES||key==KEY0_PRES)//下一曲/上一曲
                    {
                        res=key;
                        break;
                    }
                    wav_get_curtime(audiodev.file,&wavctrl);
                                //得到总时间和当前播放的时间
                    audio_msg_show(wavctrl.totsec,wavctrl.cursec,wavctrl.bitrate);
                    t++;
                    if(t==20)
                    {
                        t=0;
                        LED0_Toggle;
                    }
                    if((audiodev.status&0X01)==0)delay_ms(10);
                    else break;
                }
            }
            audio_stop();
        }else res=0XFF;
    }else res=0XFF;
    }else res=0XFF;
    myfree(SRAMIN,audiodev.tbuf);          //释放内存
    myfree(SRAMIN,audiodev.saibuf1);       //释放内存
    myfree(SRAMIN,audiodev.saibuf2);       //释放内存
    myfree(SRAMIN,audiodev.file);          //释放内存
    return res;
}
```

以上，wav_decode_init 函数，用来对 wav 文件进行解析，得到 wav 的详细信息（音频采样率，位数，数据流起始位置等）；wav_buffill 函数，用 f_read 读取数据，填充数据到 buf 里面，注意 24 位音频的时候，读出的数据需要扩展为 32 位才可填充到 buf；wav_sai_dma_tx_callback 函数，则是 DMA 发送完成的回调函数（sai_tx_callback 函数指针指向该函数），这里面，我们并没有对数据进行填充处理（暂停时进行了填 0 处理），而是采用 2 个标志量：wavtransferend 和 wavwitchbuf，来告诉 wav_play_song 函数是否传输完成，以及应该填充哪个数据 buf（saibuf1 或 saibuf2）；

最后，wav_play_song 函数，是播放 WAV 的最终执行函数，该函数解析完 WAV 文件后，设置 WM8978 和 I2S 的参数（采样率，位数等），并开启 DMA，然后不停填充数据，实现 WAV 播放，该函数还进行了按键扫描控制，实现上下取切换和暂停/播放等操作。该函数通过判断 wavtransferend 是否为 1 来处理是否应该填充数据，而到底填充到哪个 buf（saibuf1 或 saibuf2），则是通过 wavwitchbuf 标志来确定的，当 wavwitchbuf=0 时，说明 DMA 正在使用 saibuf2，程序应该填充 saibuf1；当 wavwitchbuf=1 时，说明 DMA 正在使用 saibuf1，程序应该填充 saibuf2；

接下来，看看 audioplay.c 里面的几个函数，代码如下：

```
void audio_play(void)
{
    u8 res;
    DIR wavdir;                 //目录
    FILINFO *wavfileinfo;  //文件信息
    u8 *pname;                  //带路径的文件名
    u16 totwavnum;              //音乐文件总数
    u16 curindex;               //当前索引
    u8 key;                     //键值
    u32 temp;
    u32 *wavoffsettbl;  //音乐 offset 索引表

    WM8978_ADDA_Cfg(1,0);  //开启 DAC
    WM8978_Input_Cfg(0,0,0);//关闭输入通道
    WM8978_Output_Cfg(1,0);  //开启 DAC 输出

    while(f_opendir(&wavdir,"0:/MUSIC"))//打开音乐文件夹
    {
        Show_Str(60,190,240,16,"MUSIC 文件夹错误!",16,0);
        delay_ms(200);
        LCD_Fill(60,190,240,206,WHITE);//清除显示
        delay_ms(200);
    }
    totwavnum=audio_get_tnum("0:/MUSIC"); //得到总有效文件数
    while(totwavnum==NULL)//音乐文件总数为 0
    {
        Show_Str(60,190,240,16,"没有音乐文件!",16,0);
        delay_ms(200);
        LCD_Fill(60,190,240,146,WHITE);//清除显示
        delay_ms(200);
    }
    wavfileinfo=(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
    wavoffsettbl=mymalloc(SRAMIN,4*totwavnum);
                        //申请 4*totwavnum 个字节的内存,用于存放音乐文件 off block 索引
    while(!wavfileinfo||!pname||!wavoffsettbl)//内存分配出错
```

```
{
    Show_Str(60,190,240,16,"内存分配失败!",16,0);
    delay_ms(200);
    LCD_Fill(60,190,240,146,WHITE);//清除显示
    delay_ms(200);
}
//记录索引
res=f_opendir(&wavdir,"0:/MUSIC"); //打开目录
if(res==FR_OK)
{
    curindex=0;//当前索引为 0
    while(1)//全部查询一遍
    {
        temp=wavdir.dptr;                                   //记录当前 index
        res=f_readdir(&wavdir,wavfileinfo);                 //读取目录下的一个文件
        if(res!=FR_OK||wavfileinfo->fname[0]==0)break;//错误了/到末尾了,退出

        res=f_typetell((u8*)wavfileinfo->fname);
        if((res&0XF0)==0X40)//取高四位,看看是不是音乐文件
        {
            wavoffsettbl[curindex]=temp;//记录索引
            curindex++;
        }
    }
}
curindex=0;                                                 //从 0 开始显示
res=f_opendir(&wavdir,(const TCHAR*)"0:/MUSIC");    //打开目录
while(res==FR_OK)//打开成功
{
    dir_sdi(&wavdir,wavoffsettbl[curindex]);                //改变当前目录索引
    res=f_readdir(&wavdir,wavfileinfo);                     //读取目录下的一个文件
    if(res!=FR_OK||wavfileinfo->fname[0]==0)break;      //错误了/到末尾了,退出

    strcpy((char*)pname,"0:/MUSIC/");                       //复制路径(目录)
    strcat((char*)pname,(const char*)wavfileinfo->fname);//将文件名接在后面
    LCD_Fill(60,190,lcddev.width-1,190+16,WHITE);   //清除之前的显示
    Show_Str(60,190,lcddev.width-60,16,(u8*)wavfileinfo->fname,16,0);//显示歌曲名字
    audio_index_show(curindex+1,totwavnum);
    key=audio_play_song(pname);                             //播放这个音频文件
    if(key==KEY2_PRES)        //上一曲
    {
        if(curindex)curindex--;
        else curindex=totwavnum-1;
```

```
            }else if(key==KEY0_PRES)//下一曲
            {
                curindex++;
                if(curindex>=totwavnum)curindex=0;//到末尾的时候,自动从头开始
            }else break;    //产生了错误
        }
        myfree(SRAMIN,wavfileinfo);          //释放内存
        myfree(SRAMIN,pname);                //释放内存
        myfree(SRAMIN,wavoffsettbl);         //释放内存
    }
```

这里，audio_play 函数在 main 函数里面被调用，该函数首先设置 WM8978 相关配置，然后查找 SD 卡里面的 MUSIC 文件夹，并统计该文件夹里面总共有多少音频文件（统计包括：WAV/MP3/APE/FLAC 等），然后，该函数调用 audio_play_song 函数，按顺序播放这些音频文件。

在 audio_play_song 函数里面，通过判断文件类型，调用不同的解码函数，本章，只支持 WAV 文件，通过 wav_play_song 函数实现 WAV 解码。其他格式：MP3/APE/FLAC 等，在综合实验我们会实现其解码函数，大家可以参考综合实验代码，这里就不做介绍了。

最后我们看看 main 函数源码：

```
int main(void)
{

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    W25QXX_Init();                  //初始化 W25Q256
    WM8978_Init();                  //初始化 WM8978
    WM8978_HPvol_Set(40,40);        //耳机音量设置
    WM8978_SPKvol_Set(50);          //喇叭音量设置
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部 SDRAM 内存池
    my_mem_init(SRAMDTCM);          //初始化内部 DTCM 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 SPI FLASH.
    while(font_init())              //检查字库
    {
        LCD_ShowString(30,50,200,16,16,"Font Error!");
```

```
            delay_ms(200);
            LCD_Fill(30,50,240,66,WHITE);//清除显示
            delay_ms(200);
        }
        POINT_COLOR=RED;
        Show_Str(60,50,200,16,"阿波罗 STM32F4/F7 开发板",16,0);
        Show_Str(60,70,200,16,"音乐播放器实验",16,0);
        Show_Str(60,90,200,16,"正点原子@ALIENTEK",16,0);
        Show_Str(60,110,200,16,"2016 年 7 月 18 日",16,0);
        Show_Str(60,130,200,16,"KEY0:NEXT     KEY2:PREV",16,0);
        Show_Str(60,150,200,16,"KEY_UP:PAUSE/PLAY",16,0);
        while(1)
        {
            audio_play();
        }
    }
```

该函数就相对简单了，在初始化各个外设后，通过 audio_play 函数，开始音频播放。软件部分就介绍到这里，其他未贴出代码，请参考光盘本例程源码。

52.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，程序先执行字库检测，然后当检测到 SD 卡根目录的 MUSIC 文件夹有有效音频文件（WAV 格式音频）的时候，就开始自动播放歌曲了，如图 52.4.1 所示：

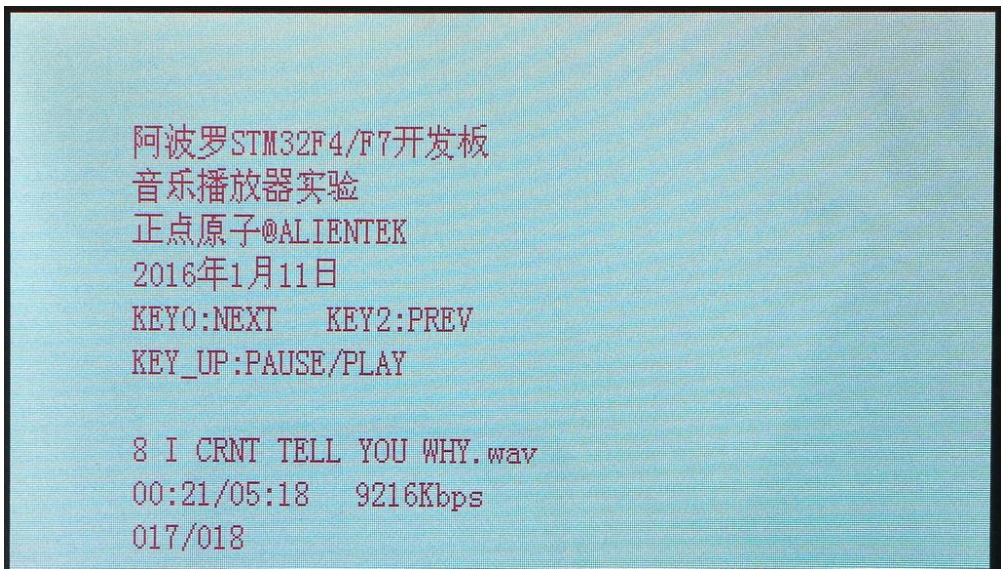


图 52.4.1 音乐播放中

从上图可以看出，当前正在播放第 17 首歌曲，总共 18 首歌曲，歌曲名、播放时间、总时长、码率、音量等信息等也都有显示。此时 DS0 会随着音乐的播放而闪烁。

图中我们播放的是 192Khz，24 位的音乐，码率=192*24*2=9216Kbps，这比最好的 MP3（320Kbps）足足高了 28 倍多!!!因而可以带来更好的音质享受，发烧友的最爱。

我们在开发板的 PHONE 端子插入耳机，就可以通过耳机欣赏音乐了。同时，我们可以通

过按 KEY0 和 KEY2 来切换下一曲和上一曲，通过 KEY_UP 控制暂停和继续播放。

本实验，我们还可以通过 USMART 来测试 WM8978 的其他功能，通过将 wm8978.c 里面的部分函数加入 USMART 管理，我们可以很方便的设置 wm8978 的各种参数（音量、3D、EQ 等都可以设置），达到验证测试的目的。有兴趣的朋友，可以实验测试一下。

至此，我们就完成了一个简单的音乐播放器了，虽然只支持 WAV 文件，但是大家可以在此基础上，增加其他音频格式解码器（可参考综合实验），便可实现其他音频格式解码了。

第五十三章 录音机实验

上一章，我们实现了一个简单的音乐播放器，本章我们将在上一章的基础上，实现一个简单的录音机，实现 WAV 录音。本章分为如下几个部：

53.1 SAI 录音简介
53.2 硬件设计
53.3 软件设计
53.4 下载验证

53.1 SAI 录音简介

本章涉及的知识点基本上在上一章都有介绍。本章要实现 WAV 录音，还是和上一章一样，要了解：WAV 文件格式、WM8978 和 SAI 接口。WAV 文件格式，我们在上一章已经做了详细介绍了，这里就不作介绍了。

ALIENTEK 阿波罗 STM32F767 开发板将板载的一个 MIC 分别接入到了 WM8978 的 2 个差分输入通道（LIP/LIN 和 RIP/RIN，原理图见：图 52.2.1）。代码上，我们采用立体声 WAV 录音，不过，左右声道的音源都是一样的，录音出来的 WAV 文件，听起来就是个单声道效果。

WM8978 上一章也做了比较详细的介绍，本章我们主要看一下要进行 MIC 录音，WM8978 的配置步骤：

1，寄存器 R0（00h），该寄存器用于控制 WM8978 的软复位，写任意值到该寄存器地址，即可实现软复位 WM8978。

2，寄存器 R1（01h），该寄存器主要要设置 MICBEN(bit4)和 BIASEN(bit3)两个位为 1，开启麦克风(MIC)偏置，以及使能模拟部分放大器。

3，寄存器 R2（02h），该寄存器要设置 SLEEP(bit6)、INPGAENR(bit3)、INPGAENL(bit2)、ADCENR(bit1)和 ADCENL(bit0)等五个位。SLEEP 设置为 0，进入正常工作模式；INPGAENR 和 INPGAENL 设置为 1，使能 IP PGA 放大器；ADCENL 和 ADCENR 设置为 1，使能左右通道 ADC。

4，寄存器 R4（04h），该寄存器要设置 WL(bit6:5)和 FMT(bit4:3)等 4 个位。WL(bit6:5)用于设置字长（即设置音频数据有效位数），00 表示 16 位音频，10 表示 24 位音频；FMT(bit4:3)用于设置 I2S 音频数据格式（模式），我们一般设置为 10，表示 I2S 格式，即飞利浦模式。

5，寄存器 R6（06h），该寄存器我们直接全部设置为 0 即可，设置 MCLK 和 BCLK 都来自外部，即由 STM32F767 提供。

6，寄存器 R14（0Eh），该寄存器要设置 ADCOSR128(bit3)为 1，ADC 得到最好的 SNR。

7，寄存器 R44（2Ch），该寄存器我们要设置 LIP2INPPGA(bit0)、LIN2INPPGA(bit1)、RIP2INPPGA(bit4)和 RIN2INPPGA(bit5)等 4 个位，将这 4 个位都设置为 1，将左右通道差分输入接入 IN PGA。

8，寄存器 R45（2Dh）和 R46（2Eh），这两个寄存器用于设置 PGA 增益（调节麦克风增益），一个用于设置左通道（R45），另外一个用于设置右通道（R46）。这两个寄存器的最高位（INPPGAUPDATE）用于设置是否更新左右通道的增益，最低 6 位用于设置左右通道的增益，我们可以先设置好两个寄存器的增益，最后设置其中一个寄存器最高位为 1，即可更新增益设置。

9，寄存器 R47（2Fh）和 R48（30h），这两个寄存器也类似，我们只关心其最高位(bit8)，都设置为 1，可以让左右通道的 MIC 各获得 20dB 的增益。

10，寄存器 R49（31h），该寄存器我们要设置 TSDEN(bit1)这个位，设置为 1，开启过热

保护。

以上，就是我们用 WM8978 录音时的设置，按照以上所述，对各个寄存器进行相应的配置，即可使用 WM8978 正常录音了。不过我们本章还要用到播放录音的功能，WM8978 的播放配置在 50.1.2 节已经介绍过了，请大家参考这个章节。

上一章我们向大家介绍了 STM32F767 的 SAI 放音，通过上一章的了解，我们知道：STM32F767 SAI 的全双工通信，需要用到 SAI 的两个子模块（SAI_A 和 SAI_B）），一个工作在主模式，产生 FS、SCK 和 MCLK，一个工作在从模式，通过 SD 引脚接收数据。

本章我们必须向 WM8978 提供 WS(FS)，CK(SCK)和 MCK(MCLK)等时钟，同时又要录音，所以只能使用全双工模式。工作在主模式的 SAI 子模块循环发送数据 0X0000，给 WM8978，以产生 CK、WS 和 MCK 等信号，工作在从模式的 SAI 子模块，则接收来自 WM8978 的 ADC 数据（ADCDAT），并保存到 SD 卡，实现录音。

本章我们将同时使用 SAI 的两个子模块，以实现录音功能，SAI 的相关寄存器，我们在上一章已经介绍的差不多了，这里就不再进行寄存器介绍，大家可以参考《STM32F7 中文参考手册.pdf》第 33.5 小节。

要实现录音功能，我们根据上一章，图 50.2.1 的连接关系可知，SAI_A 子模块必须工作在主模式，循环发送 0X0000，以提供 FS、SCK 和 MCLK 等时钟信号，SAI_B 子模块则工作在从模式，读取 ADCDAT 输出的数据流（SAI_SD_B），从而实现录音功能。

最后，我们看看要通过 STM32F767 的 SAI，驱动 WM8978 实现 WAV 录音的简要步骤，如下：

1）初始化 WM8978

这个过程就是前面所讲的 WM8978 MIC 录音配置步骤，让 WM8978 的 ADC 以及其模拟部分工作起来。

2）初始化 SAI_A 和 SAI_B

本章要用到 SAI 的全双工模式，所以，SAI_A 和 SAI_B 都需要配置，其中 SAI_A 配置为主模式，SAI 设置为从模式，且与 SAI_A 同步。他们的其他配置（协议、时钟电平特性、slot 相关参数）基本一样，只是一个是发送一个是接收，且都要使能 DMA。同时，还需要设置音频采样率，不过这个只需要设置 SAI_A 的即可，还是通过上一章介绍的查表法设置。

3）设置发送和接收 DMA

放音和录音都是采用 DMA 传输数据的，本章放音其实就是个幌子，不过也得设置 DMA（使用 DMA2 数据流 3 的通道 0），配置同上一章一模一样，不过不需要开启 DMA 传输完成中断。对于录音，则使用的是 DMA2 数据流 5 的通道 0 实现的 DMA 数据接收，我们需要配置 DMA2 的数据流 5，本章将 DMA2 数据流 5 设置为：双缓冲循环模式，外设和存储器都是 16 位宽，并开启传输完成中断（方便接收数据）。

4）编写接收通道 DMA 传输完成中断服务函数

为了方便接收音频数据，我们使用 DMA 传输完成中断，每当一个缓冲接数据满了，硬件自动切换为下一个缓冲，同时进入中断服务函数，将已满缓冲的数据写入 SD 卡的 wav 文件。过程如图 53.1.1 所示：

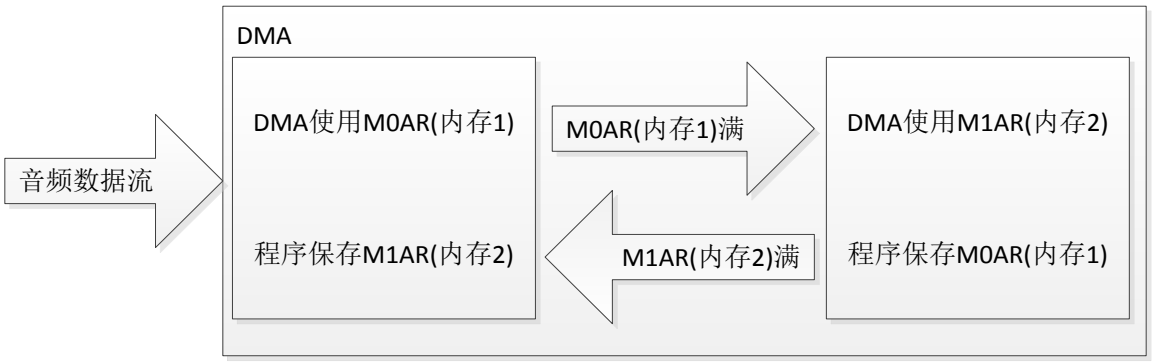


图 53.1.1 DMA 双缓冲接收音频数据流框图

5）创建 WAV 文件，并保存 wav 头

前面 4 步完成，其实就可以开始读取音频数据了，不过在录音之前，我们需要先在创建一个新的文件，并写入 wav 头，然后才能开始写入我们读取到的的 PCM 音频数据。

6）开启 DMA 传输，接收数据

然后，我们就只需要开启 DMA 传输，然后及时将 SAI_SD_B 读到的数据写入到 SD 卡之前新建的 wav 文件里面，就可以实现录音了。

7）计算整个文件大小，重新保存 wav 头并关闭文件

在结束录音的时候，我们必须知道本次录音的大小（数据大小和整个文件大小），然后更新 wav 头，重新写入文件，最后因为 FATFS，在文件创建之后，必须调用 f_close，文件才会真正体现在文件系统里面，否则是不会写入的！所以最后还需要调用 f_close，以保存文件。

53.2 硬件设计

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，再检测 SD 卡根目录是否存在 RECORDER 文件夹，如果不存在则创建，如果创建失败，则报错。在找到 SD 卡的 RECORDER 文件夹后，即进入录音模式（包括配置 WM8978 和 SAI 等），此时可以在耳机（或喇叭）听到采集到的音频。KEY0 用于开始/暂停录音，KEY2 用于保存并停止录音，KEY_UP 用于播放最近一次的录音。

当我们按下 KEY0 的时候，可以在屏幕上看到录音文件的名字、码率以及录音时间等，然后通过 KEY2 可以保存该文件，同时停止录音（文件名和时间也都将清零），在完成一段录音后，我们可以通过按 KEY_UP 按键，来试听刚刚的录音。DS0 用于提示程序正在运行，DS1 用于提示是否处于暂停录音状态。

本实验用到的资源如下：

1） 指示灯 DS0，DS1
2） 三个按键（KEY_UP/KEY0/KEY2）
3） 串口
4） LCD 模块
5） SD 卡
6） SPI FLASH
7） WM8978
8） SAI

这些前面都已介绍过。本实验，大家需要准备 1 个 SD 卡和一个耳机，分别插入 SD 卡接口和耳机接口（PHONE），然后下载本实验就可以实现一个简单的录音机了。

53.3 软件设计

打开本章实验工程可以看到我们在 APP 分组下新增了 recorder.c 文件，用来存放录音相关源码。因为 recorder.c 代码比较多，我们这里仅介绍其中几个重要的函数，代码如下：

```
u8 *sairecbuf1;                         //SAI1 DMA 接收 BUF1
u8 *sairecbuf2;                         //SAI1 DMA 接收 BUF2
//REC 录音 FIFO 管理参数.
//由于 FATFS 文件写入时间的不确定性,如果直接在接收中断里面写文件,可能导致某次写
//入时间过长从而引起数据丢失,故加入 FIFO 控制,以解决此问题.
vu8 sairecfifordpos=0;                  //FIFO 读位置
vu8 sairecfifowrpos=0;                  //FIFO 写位置
u8 *sairecfifobuf[SAI_RX_FIFO_SIZE];    //定义 10 个录音接收 FIFO
FIL* f_rec=0;                           //录音文件
u32 wavsize;                            //wav 数据大小(字节数,不包括文件头!!)
u8 rec_sta=0;                           //录音状态
                                        //[7]:0,没有开启录音;1,已经开启录音;
                                        //[6:1]:保留
                                        //[0]:0,正在录音;1,暂停录音;
//读取录音 FIFO
//buf:数据缓存区首地址
//返回值:0,没有数据可读;
//       1,读到了 1 个数据块
u8 rec_sai_fifo_read(u8 **buf)
{
    if(sairecfifordpos==sairecfifowrpos)return 0;
    sairecfifordpos++;      //读位置加 1
    if(sairecfifordpos>=SAI_RX_FIFO_SIZE)sairecfifordpos=0;//归零
    *buf=sairecfifobuf[sairecfifordpos];
    return 1;
}
//写一个录音 FIFO
//buf:数据缓存区首地址
//返回值:0,写入成功;
//       1,写入失败
u8 rec_sai_fifo_write(u8 *buf)
{
    u16 i;
    u8 temp=sairecfifowrpos;//记录当前写位置
    sairecfifowrpos++;      //写位置加 1
    if(sairecfifowrpos>=SAI_RX_FIFO_SIZE)sairecfifowrpos=0;//归零
    if(sairecfifordpos==sairecfifowrpos)
    {
        sairecfifowrpos=temp;//还原原来的写位置,此次写入失败
```

```
            return 1;
        }
        for(i=0;i<SAI_RX_DMA_BUF_SIZE;i++)sairecfifobuf[sairecfifowrpos][i]=buf[i];//拷贝
        return 0;
}
//录音  SAI_DMA 接收中断服务函数.在中断里面写入数据
void rec_sai_dma_rx_callback(void)
{
        if(rec_sta==0X80)//录音模式
        {
            if(DMA2_Stream5->CR&(1<<19))rec_sai_fifo_write(sairecbuf1);//sairecbuf1 写 FIFO
            else rec_sai_fifo_write(sairecbuf2);//sairecbuf2 写入 FIFO
        }
}
const u16 saiplaybuf[2]={0X0000,0X0000};//2 个数据,用于录音时 SAI_A 主机循环发送 0.
//进入 PCM  录音模式
void recoder_enter_rec_mode(void)
{
        WM8978_ADDA_Cfg(0,1);           //开启 ADC
        WM8978_Input_Cfg(1,1,0);        //开启输入通道(MIC&LINE IN)
        WM8978_Output_Cfg(0,1);         //开启 BYPASS 输出
        WM8978_MIC_Gain(46);            //MIC 增益设置
        WM8978_SPKvol_Set(0);           //关闭喇叭.
        WM8978_I2S_Cfg(2,0);            //飞利浦标准,16 位数据长度
        SAIA_Init(SAI_MODEMASTER_TX,SAI_CLOCKSTROBING_RISINGEDGE,
                        SAI_DATASIZE_16); //SAI1 Block A,主发送,16 位数据
        SAIB_Init(SAI_MODESLAVE_RX,SAI_CLOCKSTROBING_RISINGEDGE,
                        SAI_DATASIZE_16);//SAI1 Block B 从模式接收,16 位
        SAIA_SampleRate_Set(REC_SAMPLERATE);//设置采样率
        SAIA_TX_DMA_Init((u8*)&saiplaybuf[0],(u8*)&saiplaybuf[1],1,1);//TX DMA,16 位
        __HAL_DMA_DISABLE_IT(&SAI1_TXDMA_Handler,DMA_IT_TC);
                                                        //关闭传输完成中断(这里不用中断送数据)
        SAIA_RX_DMA_Init(sairecbuf1,sairecbuf2,SAI_RX_DMA_BUF_SIZE/2,1);
                                                      //配置 RX DMA
        sai_rx_callback=rec_sai_dma_rx_callback;//初始化回调函数指 sai_rx_callback
        SAI_Play_Start();               //开始 SAI 数据发送(主机)
        SAI_Rec_Start();                //开始 SAI 数据接收(从机)
        recoder_remindmsg_show(0); }
//初始化 WAV 头.
void recoder_wav_init(__WaveHeader* wavhead) //初始化 WAV 头
{
        wavhead->riff.ChunkID=0X46464952;       //"RIFF"
        wavhead->riff.ChunkSize=0;              //还未确定,最后需要计算
```

```
        wavhead->riff.Format=0X45564157;        //"WAVE"
        wavhead->fmt.ChunkID=0X20746D66;        //"fmt "
        wavhead->fmt.ChunkSize=16;              //大小为 16 个字节
        wavhead->fmt.AudioFormat=0X01;          //0X01,表示 PCM;0X01,表示 IMA ADPCM
        wavhead->fmt.NumOfChannels=2;           //双声道
        wavhead->fmt.SampleRate=REC_SAMPLERATE;//设置采样速率
        wavhead->fmt.ByteRate=wavhead->fmt.SampleRate*4;//采样率*通道数*(ADC 位数/8)
        wavhead->fmt.BlockAlign=4;              //块大小=通道数*(ADC 位数/8)
        wavhead->fmt.BitsPerSample=16;          //16 位 PCM
        wavhead->data.ChunkID=0X61746164;       //"data"
        wavhead->data.ChunkSize=0;              //数据大小,还需要计算
}
//WAV 录音
void wav_recorder(void)
{
        u8 res,i; u8 key; u8 rval=0;
        __WaveHeader *wavhead=0;
        DIR recdir;                             //目录
        u8 *pname=0; u8 *pdatabuf;
        u8 timecnt=0;                           //计时器
        u32 recsec=0;                           //录音时间
        while(f_opendir(&recdir,"0:/RECORDER"))//打开录音文件夹
        {
                Show_Str(30,230,240,16,"RECORDER 文件夹错误!",16,0); delay_ms(200);
                LCD_Fill(30,230,240,246,WHITE); delay_ms(200);          //清除显示
                f_mkdir("0:/RECORDER");                                 //尝试创建该目录
        }
        sairecbuf1=mymalloc(SRAMIN,SAI_RX_DMA_BUF_SIZE);  //SAI 录音内存 1 申请
        sairecbuf2=mymalloc(SRAMIN,SAI_RX_DMA_BUF_SIZE);  //SAI 录音内存 2 申请
        for(i=0;i<SAI_RX_FIFO_SIZE;i++)
        {
                sairecfifobuf[i]=mymalloc(SRAMIN,SAI_RX_DMA_BUF_SIZE);//FIFO 内存申请
                if(sairecfifobuf[i]==NULL)break;                //申请失败
        }
        f_rec=(FIL *)mymalloc(SRAMIN,sizeof(FIL));  //开辟 FIL 字节的内存区域
        wavhead=(__WaveHeader*)mymalloc(SRAMIN,sizeof(__WaveHeader));//申请内存
        pname=mymalloc(SRAMIN,30);//申请 30 字节内存,类似"0:RECORDER/REC00001.wav"
        if(!sairecbuf1||!sairecbuf2||!f_rec||!wavhead||!pname||i!=SAI_RX_FIFO_SIZE)rval=1;
        if(rval==0)
        {
                recoder_enter_rec_mode();   //进入录音模式,此时耳机可以听到咪头采集到的音频
                pname[0]=0;                 //pname 没有任何文件名
                while(rval==0)
```

```
{
    key=KEY_Scan(0);
    switch(key)
    {
        case KEY2_PRES: //STOP&SAVE
            if(rec_sta&0X80)//有录音
            {
                rec_sta=0;                                  //关闭录音
                wavhead->riff.ChunkSize=wavsize+36;         //整个文件的大小-8;
                wavhead->data.ChunkSize=wavsize;            //数据大小
                f_lseek(f_rec,0);                           //偏移到文件头.
                f_write(f_rec,(const void*)wavhead,sizeof(__WaveHeader),&bw);
                f_close(f_rec);
                wavsize=0;
                sairecfifordpos=0;  //FIFO 读写位置重新归零
                sairecfifowrpos=0;
            }
            rec_sta=0; recsec=0;
            LED1(1);                                        //关闭 DS1
            LCD_Fill(30,190,lcddev.width-1,lcddev.height-1,WHITE);//清除显示
            break;
        case KEY0_PRES: //REC/PAUSE
            if(rec_sta&0X01) rec_sta&=0XFE;//原来是暂停,取消暂停,继续录音
            else if(rec_sta&0X80) rec_sta|=0X01;//已经在录音了,则暂停
            else                //还没开始录音
            {
                recsec=0;
                recoder_new_pathname(pname);                //得到新的名字
                Show_Str(30,190,lcddev.width,16,"录制:",16,0);
                Show_Str(30+40,190,lcddev.width,16,pname+11,16,0);//显示名字
                recoder_wav_init(wavhead);                  //初始化 wav 数据
                res=f_open(f_rec,(const TCHAR*)pname,
                           FA_CREATE_ALWAYS | FA_WRITE);
                if(res)             //文件创建失败
                {
                    rec_sta=0;      //创建文件失败,不能录音
                    rval=0XFE;      //提示是否存在 SD 卡
                }else
                {
                    res=f_write(f_rec,(const void*)wavhead,
                                sizeof(__WaveHeader),&bw);//写入头数据
                    recoder_msg_show(0,0);
                    rec_sta|=0X80;      //开始录音
```

```
                    }
                }
                if(rec_sta==0X80)LED1(0);//提示正在暂停
                else LED1(1);
                break;
            case WKUP_PRES:     //播放最近一段录音
                if(rec_sta!=0X80)//没有在录音
                {
                    if(pname[0])//如果触摸按键被按下,且 pname 不为空
                    {
                        Show_Str(30,190,lcddev.width,16,"播放:",16,0);
                        Show_Str(30+40,190,lcddev.width,16,pname+11,16,0);//显示
                        LCD_Fill(30,210,lcddev.width-1,230,WHITE);
                        recoder_enter_play_mode();  //进入播放模式
                        audio_play_song(pname);          //播放 pname
                        LCD_Fill(30,190,lcddev.width-1,lcddev.height-1,WHITE);
                        recoder_enter_rec_mode();   //重新进入录音模式
                    }
                }
                break;
        }
        if(rec_sai_fifo_read(&pdatabuf))//读取一次数据,读到数据了,写入文件
        {
            res=f_write(f_rec,pdatabuf,SAI_RX_DMA_BUF_SIZE,(UINT*)&bw);//写
            if(res)printf("write error:%d\r\n",res);
            wavsize+=SAI_RX_DMA_BUF_SIZE;
        }else delay_ms(5);
        timecnt++;
        if((timecnt%20)==0)LED0_Toggle;     //DS0 闪烁
        if(recsec!=(wavsize/wavhead->fmt.ByteRate))     //录音时间显示
        {
            LED0_Toggle;//DS0 闪烁
            recsec=wavsize/wavhead->fmt.ByteRate;     //录音时间
            recoder_msg_show(recsec,wavhead->fmt.SampleRate*wavhead->
                fmt.NumOfChannels*wavhead->fmt.BitsPerSample);//显示码率
        }
    }
}
myfree(SRAMIN,sairecbuf1);      //释放内存
myfree(SRAMIN,sairecbuf2);      //释放内存
for(i=0;i<SAI_RX_FIFO_SIZE;i++)myfree(SRAMIN,sairecfifobuf[i]);//FIFO 内存释放
myfree(SRAMIN,f_rec);       //释放内存
myfree(SRAMIN,wavhead);         //释放内存
```

```
    myfree(SRAMIN,pname);           //释放内存
}
```

这里总共 6 个函数，接下来，我们分别介绍。

1，rec_sai_fifo_read 和 rec_sai_fifo_write 函数

这两个函数用于我们构建的 FIFO 里面的数据读取和写入，SAI 采集到的数据，通过 FATFS 写入 SD 卡的时候，因为 FATFS 写入时间不确定（有时候短，有时候长），可能导致数据写入不及时，出现数据丢失，从而录音会有间隔（丢失一部分）。所以，我们构建了一个 FIFO，SAI 采集的数据，通过 rec_sai_fifo_write 函数写入 FIFO 里面，在主循环里面，我们通过 rec_sai_fifo_read 函数不停的读取 FIFO 里面的数据，并将数据通过 FATFS 写入 SD 卡里面，只要 rec_sai_fifo_read 的速度，不小于 rec_sai_fifo_write 的速度，就可以保证数据不丢失，这个 FIFO 起到了一个缓冲的作用，从而保证录音文件的流畅性。

2，rec_sai_dma_rx_callback 函数

该函数用于 SAI_B 的 DMA 接收完成中断回调函数(通过 sai_rx_callback 指向该函数实现)，在该函数里面调用 rec_sai_fifo_write 函数，将采集到的音频数据，写入 FIFO。

3，recoder_enter_rec_mode 函数

该函数用于设置 WM8978 进入录音模式，并设置 SAI_A 和 SAI_B 的工作模式和位数等信息，然后配置 DMA 和回调函数的指向，最后开启录音。调用该函数后，就可以开始录音了。

4，recoder_wav_init 函数

该函数初始化 wav 头的绝大部分数据，采样率通过 REC_SAMPLERATE 宏定义修改，默认是 44.1Khz，位数为 16 位，线性 PCM 格式，另外由于录音还未真正开始，所以文件大小和数据大小都还是未知的，要等录音结束才能知道。该函数__WaveHeader 结构体就是由上一章（50.1.1 节）介绍的三个 Chunk 组成，结构为：

```
//wav 头
typedef __packed struct
{
    ChunkRIFF riff;     //riff 块
    ChunkFMT fmt;       //fmt 块
//  ChunkFACT fact;     //fact 块  线性 PCM,没有这个结构体
    ChunkDATA data;     //data 块
}__WaveHeader;
```

5，wav_recorder 函数

该函数实现了我们在硬件设计时介绍的功能（开始/暂停录音、保存录音文件、播放最近一次录音等），实现方法请大家参考源码理解。另外，该函数使用上一章实现的 audio_play_song 函数，来播放最近一次录音。

recorder.c 的其他代码和 recorder.h 的代码我们这里就不再贴出了，请大家参考光盘本实验的源码。然后，我们在 sai.c 里面也增加了几个函数，如下：

```
//SAI Block B 初始化,I2S,飞利浦标准
//mode:工作模式,可以设置:SAI_MODEMASTER_TX/
        //SAI_MODEMASTER_RX/SAI_MODESLAVE_TX/SAI_MODESLAVE_RX
//cpol:数据在时钟的上升/下降沿选通，可以设置：
    //SAI_CLOCKSTROBING_FALLINGEDGE/SAI_CLOCKSTROBING_RISINGEDGE
//datalen:数据大小,可以设置：SAI_DATASIZE_8/10/16/20/24/32
void SAIB_Init(u32 mode,u32 cpol,u32 datalen)
```

```
{
    HAL_SAI_DeInit(&SAI1B_Handler);              //清除以前的配置
    SAI1B_Handler.Instance=SAI1_Block_B;         //SAI1 Bock B
    SAI1B_Handler.Init.AudioMode=mode;           //设置 SAI1 工作模式
    SAI1B_Handler.Init.Synchro=SAI_SYNCHRONOUS;      //音频模块同步
    SAI1B_Handler.Init.OutputDrive=SAI_OUTPUTDRIVE_ENABLE; //立即驱动输出
    SAI1B_Handler.Init.NoDivider=SAI_MASTERDIVIDER_ENABLE; //使能主时钟分频器
    SAI1B_Handler.Init.FIFOThreshold=SAI_FIFOTHRESHOLD_1QF //设置 FIFO 阈值
    SAI1B_Handler.Init.ClockSource=SAI_CLKSOURCE_PLLI2S; //SIA 时钟源为 PLL2S
    SAI1B_Handler.Init.MonoStereoMode=SAI_STEREOMODE;      //立体声模式
    SAI1B_Handler.Init.Protocol=SAI_FREE_PROTOCOL;   //设置 SAI1 协议为自由协议
    SAI1B_Handler.Init.DataSize=datalen;                    //设置数据大小
    SAI1B_Handler.Init.FirstBit=SAI_FIRSTBIT_MSB;        //数据 MSB 位优先
    SAI1B_Handler.Init.ClockStrobing=cpol;   //数据在时钟的上升/下降沿选通

    //帧设置
    SAI1B_Handler.FrameInit.FrameLength=64; //设置帧长度为 64,左/右通道各 32 个 SCK,
    SAI1B_Handler.FrameInit.ActiveFrameLength=32; //设置帧同步有效电平长度
    SAI1B_Handler.FrameInit.FSDefinition=SAI_FS_CHANNEL_IDENTIFICATION;
                                                  //FS 信号为 SOF 信号+通道识别信号
    SAI1B_Handler.FrameInit.FSPolarity=SAI_FS_ACTIVE_LOW; //FS 低电平有效(下降沿)
    SAI1B_Handler.FrameInit.FSOffset=SAI_FS_BEFOREFIRSTBIT;
                                  //在 slot0 的第一位的前一位使能 FS,以匹配飞利浦标准
    //SLOT 设置
    SAI1B_Handler.SlotInit.FirstBitOffset=0;                    //slot 偏移(FBOFF)为 0
    SAI1B_Handler.SlotInit.SlotSize=SAI_SLOTSIZE_32B;        //slot 大小为 32 位
    SAI1B_Handler.SlotInit.SlotNumber=2;                      //slot 数为 2 个
    SAI1B_Handler.SlotInit.SlotActive=SAI_SLOTACTIVE_0|SAI_SLOTACTIVE_1;
                                                          //使能 slot0 和 slot1

    HAL_SAI_Init(&SAI1B_Handler);
    SAIB_DMA_Enable();          //使能 SAI 的 DMA 功能
    __HAL_SAI_ENABLE(&SAI1B_Handler);   //使能 SAI
}

//SAIA TX DMA 配置
//设置为双缓冲模式,并开启 DMA 传输完成中断
//buf0:M0AR 地址.
//buf1:M1AR 地址.
//num:每次传输数据量
//width:位宽(存储器和外设,同时设置),0,8 位;1,16 位;2,32 位;
void SAIA_RX_DMA_Init(u8* buf0,u8 *buf1,u16 num,u8 width)
{
```

```
u32 memwidth=0,perwidth=0;          //外设和存储器位宽
switch(width)
{
    case 0:             //8 位
        memwidth=DMA_MDATAALIGN_BYTE;
        perwidth=DMA_PDATAALIGN_BYTE;
        break;
    case 1:             //16 位
        memwidth=DMA_MDATAALIGN_HALFWORD;
        perwidth=DMA_PDATAALIGN_HALFWORD;
        break;
    case 2:             //32 位
        memwidth=DMA_MDATAALIGN_WORD;
        perwidth=DMA_PDATAALIGN_WORD;
        break;

}
__HAL_RCC_DMA2_CLK_ENABLE();  //使能 DMA2 时钟
__HAL_LINKDMA(&SAI1B_Handler,hdmarx,SAI1_RXDMA_Handler);
                                                    //将 DMA 与 SAI 联系起来
SAI1_RXDMA_Handler.Instance=DMA2_Stream5;  //DMA2 数据流 5
SAI1_RXDMA_Handler.Init.Channel=DMA_CHANNEL_0;    //通道 0
SAI1_RXDMA_Handler.Init.Direction=DMA_PERIPH_TO_MEMORY; //外设到存储器
SAI1_RXDMA_Handler.Init.PeriphInc=DMA_PINC_DISABLE;  //外设非增量模式
SAI1_RXDMA_Handler.Init.MemInc=DMA_MINC_ENABLE; //存储器增量模式
SAI1_RXDMA_Handler.Init.PeriphDataAlignment=perwidth; //外设数据长度:16/32 位
SAI1_RXDMA_Handler.Init.MemDataAlignment=memwidth; //存储器数据长度:16/32 位
SAI1_RXDMA_Handler.Init.Mode=DMA_CIRCULAR;    //使用循环模式
SAI1_RXDMA_Handler.Init.Priority=DMA_PRIORITY_MEDIUM; //中等优先级
SAI1_RXDMA_Handler.Init.FIFOMode=DMA_FIFOMODE_DISABLE; //不使用 FIFO
SAI1_RXDMA_Handler.Init.MemBurst=DMA_MBURST_SINGLE; //存储器单次突发
SAI1_RXDMA_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE;  //外设单次突发
HAL_DMA_DeInit(&SAI1_RXDMA_Handler);      //先清除以前的设置
HAL_DMA_Init(&SAI1_RXDMA_Handler);        //初始化 DMA

HAL_DMAEx_MultiBufferStart(&SAI1_RXDMA_Handler,
                    (u32)&SAI1_Block_B->DR,(u32)buf0,(u32)buf1,num);//开启双缓冲
__HAL_DMA_DISABLE(&SAI1_RXDMA_Handler);  //先关闭接收 DMA
delay_us(10);  //10us 延时，防止-O2 优化出问题
__HAL_DMA_CLEAR_FLAG(&SAI1_RXDMA_Handler,
                    DMA_FLAG_TCIF1_5); //清除 DMA 传输完成中断标志位
__HAL_DMA_ENABLE_IT(&SAI1_RXDMA_Handler,DMA_IT_TC);//开启传输完成中断
```

```
    HAL_NVIC_SetPriority(DMA2_Stream5_IRQn,0,1);   //DMA 中断优先级
    HAL_NVIC_EnableIRQ(DMA2_Stream5_IRQn);
}

void (*sai_rx_callback)(void);      //RX 回调函数
//DMA2_Stream5 中断服务函数
void DMA2_Stream5_IRQHandler(void)
{
    if(__HAL_DMA_GET_FLAG(&SAI1_RXDMA_Handler,
                          DMA_FLAG_TCIF1_5)!=RESET) //DMA 传输完成
    {
        __HAL_DMA_CLEAR_FLAG(&SAI1_RXDMA_Handler,DMA_FLAG_TCIF1_5);
                                                  //清除 DMA 传输完成中断标志位
        if(sai_rx_callback!=NULL)sai_rx_callback();
                                                  //执行回调函数,读取数据等操作在这里面处理
    }
}
//SAI 开始录音
void SAI_Rec_Start(void)
{
    __HAL_DMA_ENABLE(&SAI1_RXDMA_Handler);//开启 DMA RX 传输
}
//关闭 SAI 录音
void SAI_Rec_Stop(void)
{
    __HAL_DMA_DISABLE(&SAI1_RXDMA_Handler);//结束录音
}
```

这里新增了 5 个函数，SAIB_Init 函数完成 SAI_B 子模块的初始化，通过 3 个参数设置 SAI_B 的详细配置信息。SAIB_RX_DMA_Init 函数，用于设置 SAI_B 的 DMA 接收，使用双缓冲循环模式，接收来自 WM8978 的数据，并开启了传输完成中断。而 DMA2_Stream5_IRQHandler 函数，则是 DMA2 数据流 5 传输完成中断的服务函数，该函数调用 sai_rx_callback 函数（函数指针，使用前需指向特定函数）实现 DMA 数据接收保存。最后，SAI_ Rec_Start 和 SAI_ Rec_Stop，用于开启和关闭 SAI_B 的 DMA 传输。

其他代码，我们就不再介绍了，请大家参考开发板光盘本例程源码。最后我们看看 main 函数源码：

```
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
```

```
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    WM8978_Init();                  //初始化 WM8978
    WM8978_HPvol_Set(40,40);        //耳机音量设置
    WM8978_SPKvol_Set(40);          //喇叭音量设置
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部 SDRAM 内存池
    my_mem_init(SRAMDTCM);          //初始化内部 DTCM 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 SPI FLASH.
    f_mount(fs[2],"2:",1);          //挂载 NAND 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);
    }
    POINT_COLOR=RED;
    Show_Str(30,40,200,16,"阿波罗 STM32F4/F7 开发板",16,0);
    Show_Str(30,60,200,16,"录音机实验",16,0);
    Show_Str(30,80,200,16,"正点原子@ALIENTEK",16,0);
    Show_Str(30,100,200,16,"2016 年 1 月 29 日",16,0);
    while(1)
    {
        wav_recorder();
    }
}
```

该函数代码同上一章的 main 函数代码几乎一样，十分简单，我们就不再多说了。

至此，本实验的软件设计部分结束。

53.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，程序先检测字库，然后检测 SD 卡的 RECORDER 文件夹，一切顺利通过之后，进入录音模式，得到，如图 53.4.1 所示：

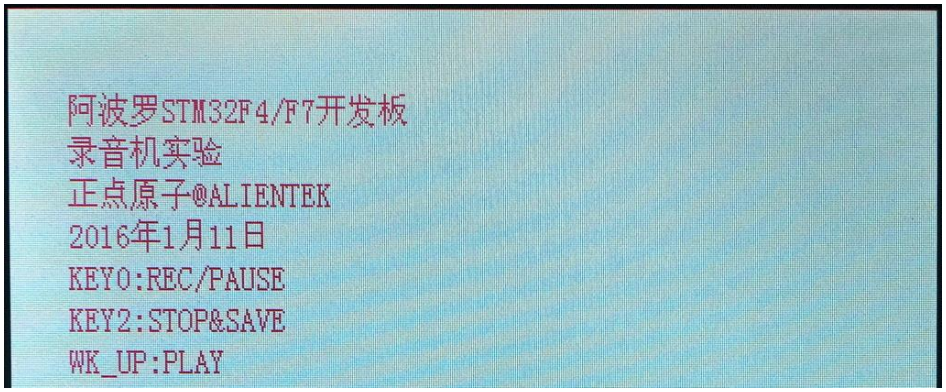


图 53.4.1 录音机界面

此时，我们按下 KEY0 就开始录音了，此时看到屏幕显示录音文件的名字、码率以及录音时长，如图 53.4.2 所示：

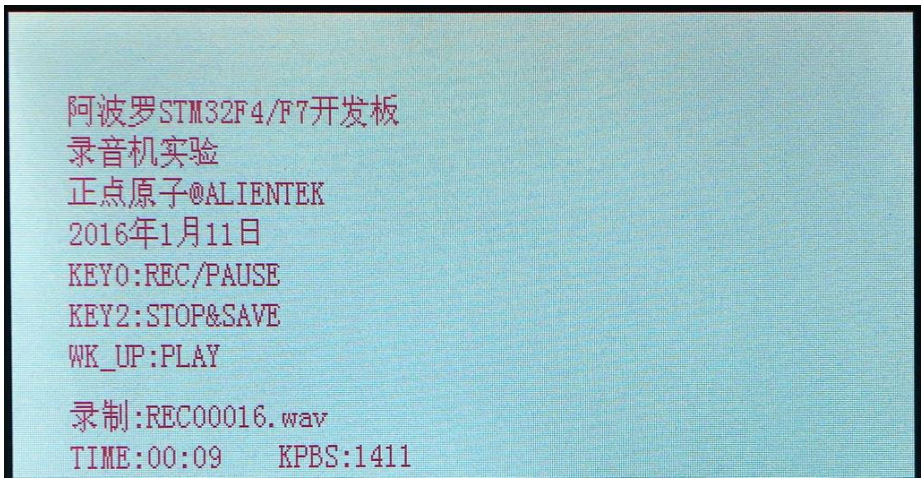


图 53.4.2 录音进行中

在录音的时候按下 KEY0 则执行暂停/继续录音的切换，通过 DS1 指示录音暂停。通过按下 KEY2，可以停止当前录音，并保存录音文件。在完成一次录音文件保存之后，我们可以通过按 KEY_UP 按键，来实现播放这个录音文件（即播放最近一次的录音文件），实现试听。

我们将开发板的录音文件放到电脑上面，可以通过属性查看录音文件的属性，如图 53.4.3 所示：

图 53.4.3 录音文件属性

这和我们预期的效果一样，通过电脑端的播放器（winamp/千千静听等）可以直接播放我们所录的音频。经实测，效果还是非常不错的。

第五十四章 SPDIF(光纤音频)实验

前面，我们介绍了 STM32F7 的 SAI 接口，实现了音乐播放器、录音机等功能，本章，我们将介绍 STM32F7 的 SPDIF 接口，结合 SAI 接口和 WM8978 解码器，实现对光纤音频信号的解码。本章分为如下几个部：

54.1 SPDIF 简介
54.2 硬件设计
54.3 软件设计
54.4 下载验证

54.1 SPDIF 简介

SPDIF 是 Sony/Philip Digital Interface Format 的缩写，是由索尼和飞利浦公司联合开发的数字音频接口简称，分为 SPDIF 输入（IN）和 SPDIF 输出（OUT）两种，STM32F7 的 SPDIF 接口，仅支持 SPDIF IN，称之为 SPDIF RX。

STM32F7 的 SPDIF 接口的主要特点有：

- 提供多达 4 路输入
- 自动符号率检测
- 最大符号率：12.288 MHz
- 支持 8 kHz 到 192 kHz 的立体声
- 支持 IEC-60958 和 IEC-61937 音频标准，消费类应用
- 奇偶校验位管理
- 使用 DMA 通信进行音频采样
- 支持控制和用户信息 DMA 传输

STM32F7 的 SPDIF RX 接口支持符合 IEC-60958 和 EC-61937 标准的 SPDIF 数据流。支持高采样率的简单立体声以及压缩的多通道环绕声（Dolby 或 DTS 音频）。SPDIF RX 接口框图如图 54.1.1 所示：

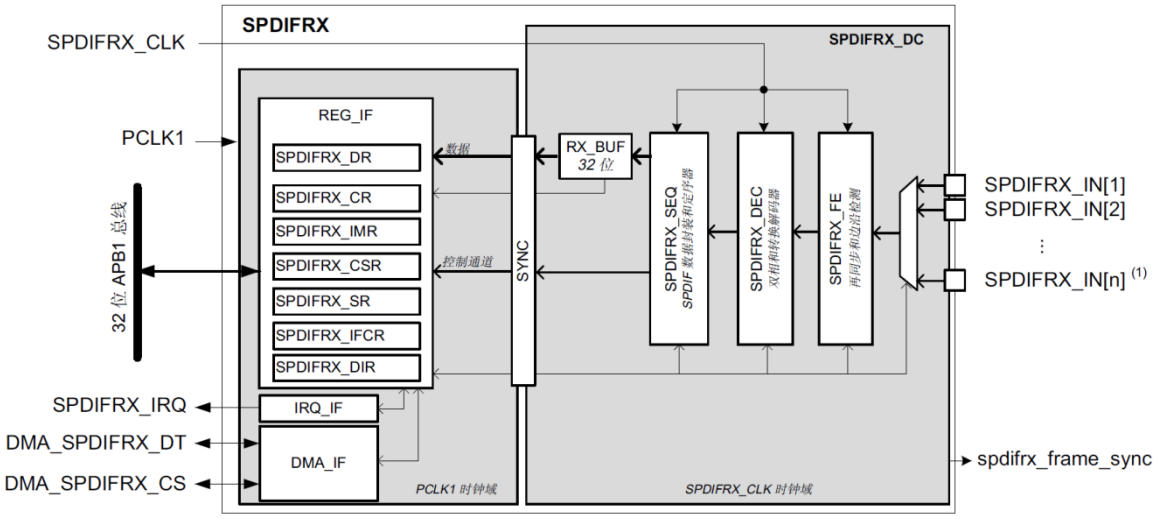


图 54.1.1 SPDIF RX 接口框图

图中 SPDIFRX_DC 模块负责解码从 SPDIFRX_IN[4:1] 输入接收的 SPDIF 数据流。该模块重新采样传入的信号、解码曼彻斯特数据流、并识别帧、子帧和块元素。它传送到 REG_IF 部分、解码数据和相关的状态标志。关于 SPDIFRX_DC 模块的详细介绍，请参考《STM32F7 中

文参考手册》34.3.2 节。

SPDIF RX 接口通过 APB1 总线完全控制并且能够处理两个 DMA 通道：

1，专用于传输音频采样的 DMA 通道

2，专用于传输 IEC60958 通道状态和用户信息的 DMA 通道

此外，还提供了中断服务，既可用作 DMA 的复用功能，也可用来指示外设的错误或关键状态。接下来，我们简单介绍一下 SPDIF 协议。

1）SPDIF 块

一个 SPDIF 块由 192 个 SPDIF 帧组成，如图 54.1.2 所示：

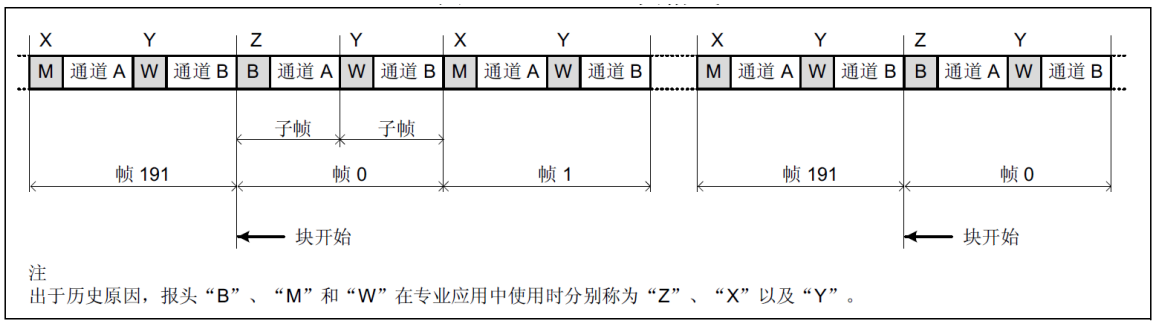


图 54.1.2 SPDIF 块格式

每个 SPDIF 帧又由 2 个子帧组成，如图 54.1.3 所示：

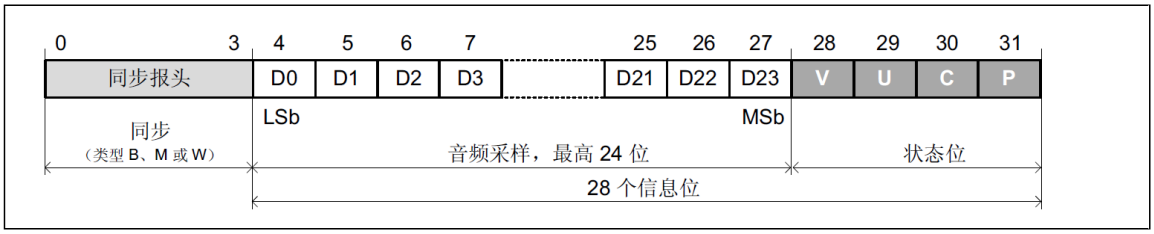


图 54.1.3 SPDIF 子帧格式

每个子帧包含 32 个位，他们的组成如下：

1，位 0 到 3 包含同步报头之一（B/M/W）；

2，位 4 到 27 包含以线性 2 的补码表示的音频采样字。最高有效位(MSB)为位 27；

3，使用 20 位编码范围时，位 8 到 27 包含音频采样字，其中位 8 为 LSB。

4，位 28（有效性位“V”）表示数据是否有效（例如转换为模拟数据）。

5，位 29（用户数据位“U”）包含用户数据信息，如光盘的音轨编号。

6，位 30（通道状态位“C”）包含通道状态信息，如采样率和复制保护。

7，位 31（奇偶检验位“P”）包含奇偶校验位，位 4 到 31 将包含偶数个 1 和 0（偶校验）。

对于线性编码音频应用，第一个子帧（立体声操作中的左声道或“A”通道以及单声道操作中的主通道）通常以报头“M”开始。但是，报头每 192 帧切换为报头“B”一次，以识别用于组织通道状态和用户信息的块结构的开始。第二个子帧（立体声操作中的右声道或“B”通道以及单声道操作中的辅助通道）始终以报头“W”开始。

2）同步报头

SPDIF 协议规定，总共有三种同步报头：B、M、W（也可以称为 Z、X、Y）。同步报头总是以和前半个位相反的电平开始的。使能第一个帧的第一个“B”报头的传输前，此前半位值为线路的电平。对于其它报头，此前半位值为之前子帧的奇偶校验位的第二个半位。

SPDIF 三种同步报头的编码方式如图 54.1.4 所示：

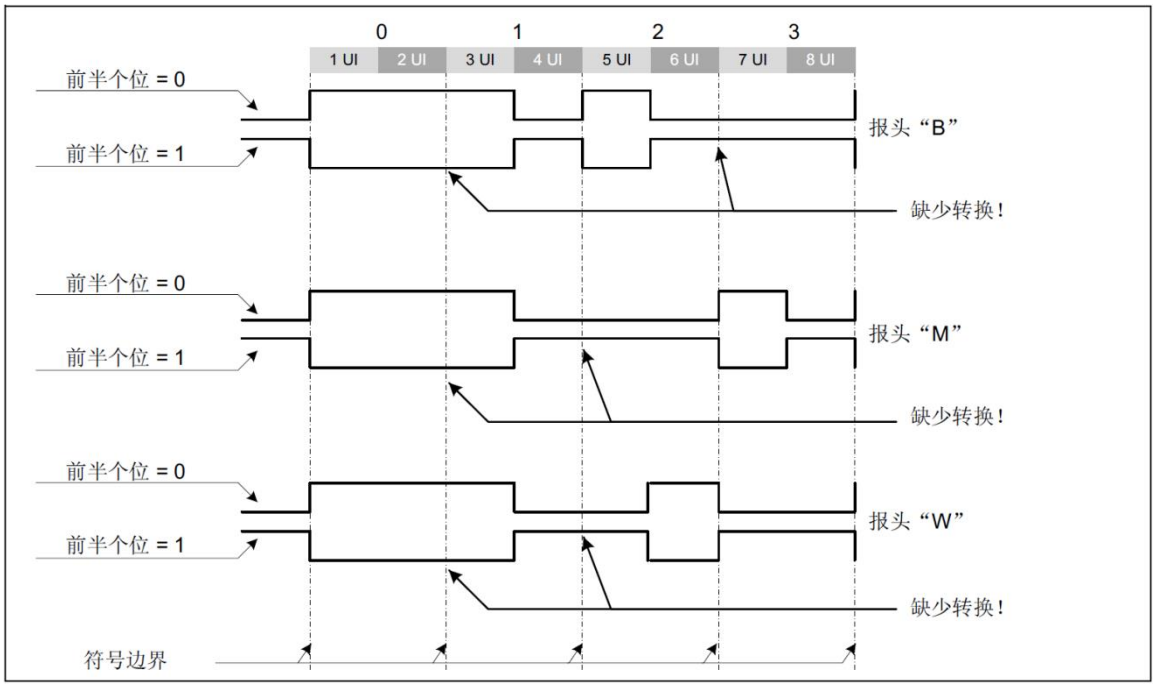


图 54.1.4 SPDIF 同步报头

3）位编码

为最大程度减小传输线路上的直流分量值，并加快时钟从数据流中恢复的速度，从第 4 位开始，到第 31 位，全部采用双相符号编码。

双向符号编码原理：要传输的各个位通过由两个连续二进制状态构成的符号表示。符号的第一个状态始终不同于前一个符号的第二个状态。如果要传输的位为逻辑 0，则符号的第二个状态与第一个状态相同。但如果该位为逻辑 1，则两个状态不同。在 IEC-60958 规范中，这些状态称为“UI”（单位间隔）。

SPDIF 的位编码原理如图 54.1.5 所示：

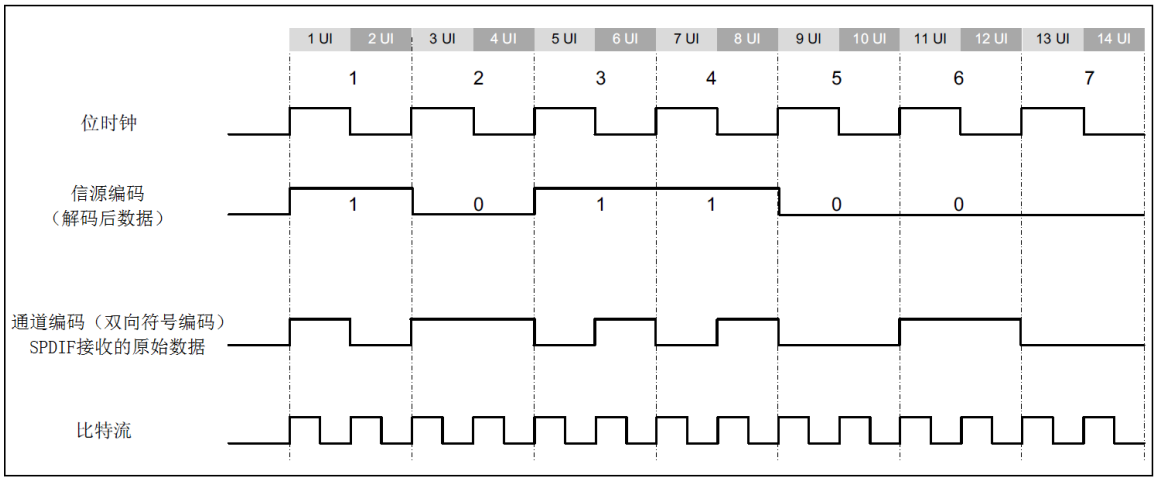


图 54.1.5 通道编码示例

图中比特流实际上就是通道解码时钟，每 2 个时钟解码一个位，在两个时钟内，通道编码状态有变化的（10/01），解码为逻辑 1；在两个解码时钟内，通道编码状态没有变化的（00/11），解码为逻辑 0；

结合通道位编码原理，以及图 54.1.4 所示的同步报头编码方式，我们可以看出，前面 4 个

位是不能用这个位编码原理来编码的（因为有 3 个 UI 的状态！！）。

SPDIF 协议我们就介绍到这里，接下来，我们看看 STM32F7 SPDIFRX 的状态流程。如图 54.1.6 所示：

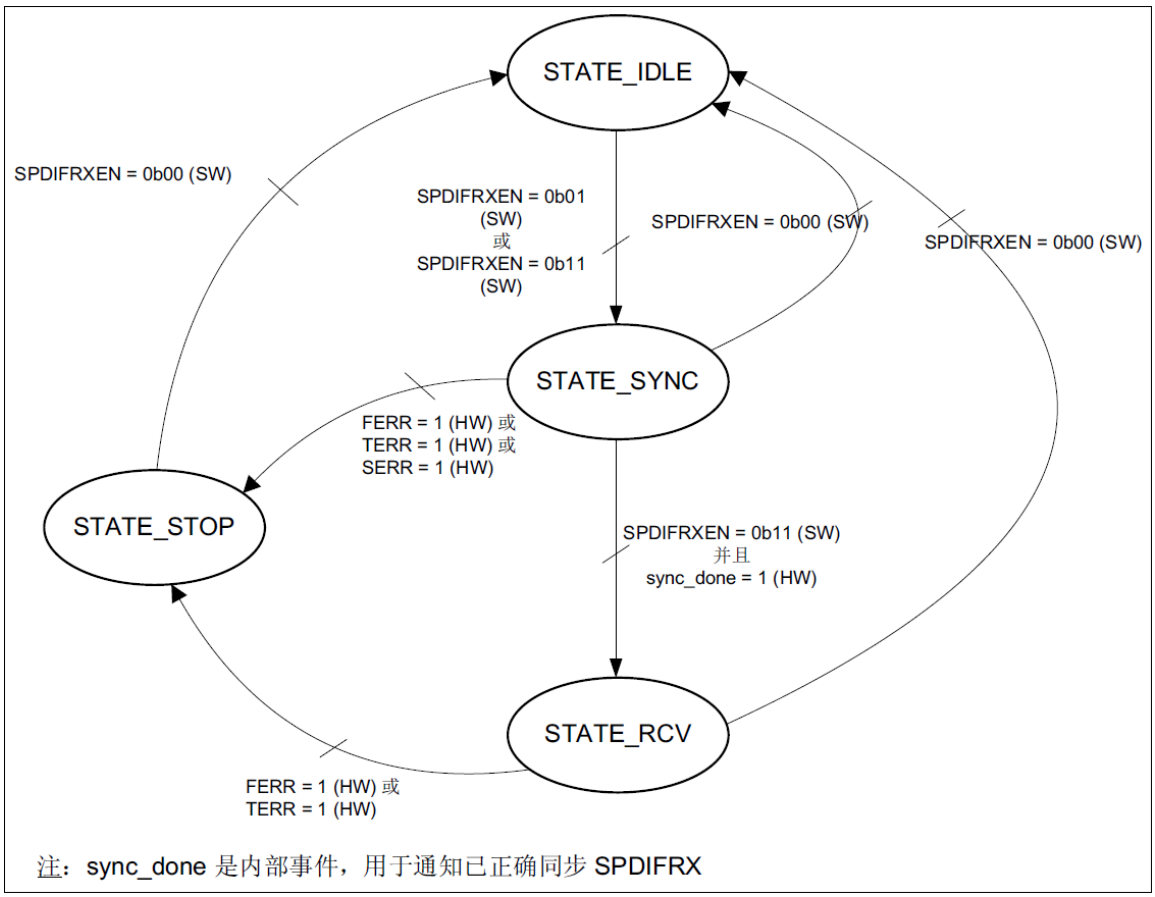


图 54.1.6 STM32F7 SPDIFRX 状态流程

由图可知，STM32F7 的 SPDIFRX 接口总共有四种状态：

1，STATE_IDLE

该状态下，禁止 SPDIF 外设，SPDIFRX_CLK 域复位，PCLK1 域功能正常。

2，STATE_SYNC

该状态下，SPDIF 将与数据流同步（同步过程请参考《STM32F7 中文参考手册》34.3.4 节），阈值定期更新，可通过中断或 DMA 读取用户和通道状态。

3，STATE_RCV：

该状态下，SPDIF 将与数据流同步，阈值定期更新，可通过中断或 DMA 通道读取用户、通道状态和音频采样。当检测到“B”报头后，开始保存音频数据。

4，STOP_STATE：

该状态下，SPDIF 将不再同步，用户、通道状态和音频数据的接收都将停止

我们可以通过 SPDIFRX_CR 寄存器的 SPDIFRXEN 字段，控制 SPDIFRX 的状态。

当 SPDIFRX 处于 STATE_IDLE 时：

通过将 SPDIFRXEN 设置为 01 或 11，将切换到 STATE_SYNC 状态。

当 SPDIFRX 处于 STATE_SYNC 时：

如果同步失败或者接收的数据未正确解码且无法在不进行再同步的情况下恢复（FERR 或 SERR 或 TERR=1），则 SPDIFRX 将进入 STATE_STOP 状态并且等待软件应答。

当同步阶段完成时，如果 SPDIFRXEN= 01，将保持该状态。

将 SPDIFRXEN 设置为 0，SPDIFRX 将立刻返回至 STATE_IDLE 状态。

当 SPDIFRXEN=11，且 SYNCD=1，则 SPDIFRX 进入 STATE_RCV 状态。

当 SPDIFRX 处于 STATE_RCV 时：

如果接收的数据未正确解码且无法在不进行再同步的情况下恢复（FERR 或 SERR 或 TERR=1），则 SPDIFRX 将进入 STATE_STOP 状态并且等待软件应答。

将 SPDIFRXEN 设置为 0，SPDIFRX 将立刻返回至 STATE_IDLE 状态。

在此状态下，如果数据接收正确，我们将其通过解码器播放出来，就能实现解码了。

当 SPDIFRX 处于 STATE_STOP 时：

SPDIFRX 停止接收和同步，并等待软件将 SPDIFRXEN 位设置为 0，以清零错误标志。

当 SPDIFRXEN 设置为 0 时，IP 被禁止，这意味着所有状态机被复位，并且 RX_BUF 被刷新。还要注意，标志 FERR、SERR 和 TERR 也会复位。

数据接收

SPDIFRX 为音频采样接收提供了两个 32 位双缓冲区：RX_BUF 和 SPDIFRX_DR。如果 SPDIFRX_DR 为空，则包含在 RX_BUF 中的有效数据将立刻传输至 SPDIFRX_DR。

SPDIFRX 可以使用 DMA 或中断将音频采样传输到存储器中，我们一般使用 DMA 来传输，这样效率比较高。

SPDIFRX 提供多种处理接收数据的方法，用户可以分别处理控制信息流和音频采样流，或将二者一起处理。对于各子帧，数据接收寄存器 SPDIFRX_DR 包含 24 个数据位和可选的 V、U、C、PE 状态位以及 PT，这些可选的位，通过 SPDIFRX_CR 寄存器的 VMSK、CUMSK、PMSK 和 PTMSK 等位控制，我们将在寄存器介绍的时候，给大家详细讲解。

PE 位表示奇偶校验错误位，该位将在解码的子帧中检测到奇偶校验错误时置 1；PT 字段包含报头类型（B、M 或 W）；V、U 和 C 则是从 SPDIF 接口接收的值的直接副本。

通过 SPDIFRX_CR 寄存器的 DRFMT 位，我们可以选择三种音频格式，如图 54.1.7 所示：

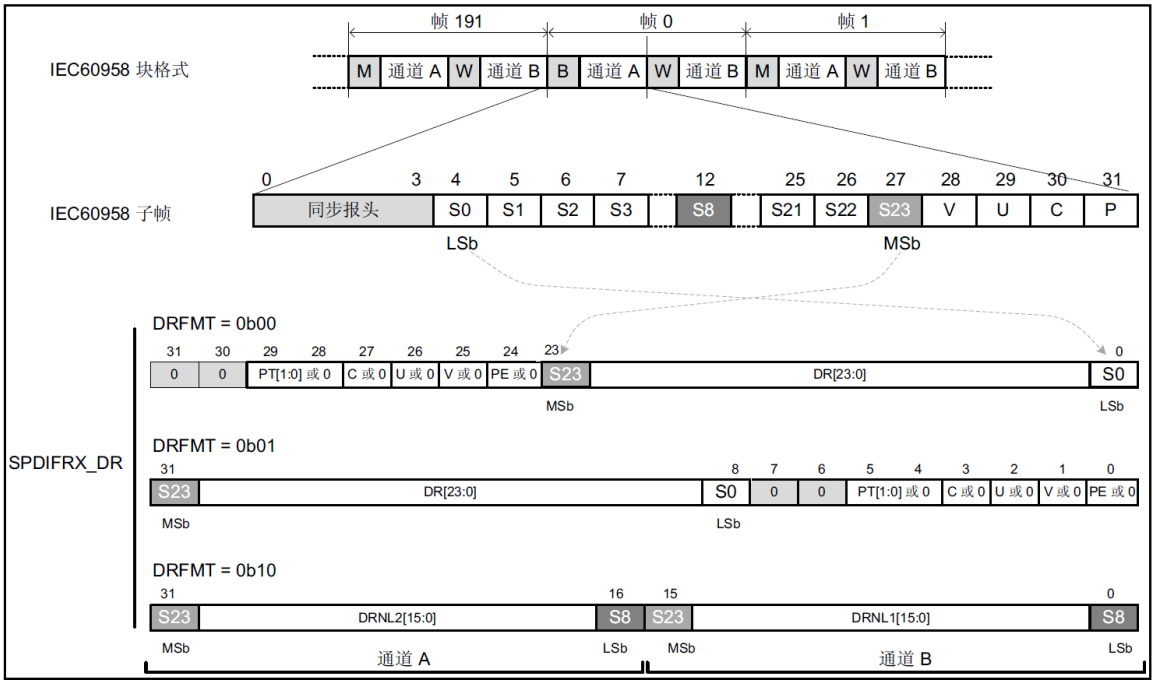


图 54.1.7 SPDIFRX_DR 寄存器格式

将 DRFMT 设置为 00 或 01，可以使数据在 SPDIFRX_DR 寄存器中右对齐或左对齐，此时

我们可以根据软件所需的处理方式启用状态信息（V/C/U/PE/PT 等）或将其强制设置为零。

将 DRFMT 设置为 10 得出的格式在非线性模式下相关，因为每个子帧仅使用 16 位。使用此格式，两个连续子帧的数据存储到 SPDIFRX_DR 中，存储器占用量将减半。

本章我们将 DRFMT 设置为 00，使用右对齐格式，且可以支持 24 位音频数据传输。

时钟策略

SPDIFRX 块需要两个不同的时钟：

1，APB1 时钟 (PCLK1)，用于寄存器接口，其频率必须至少大于符号率；

2，SPDIFRX_CLK，主要由 SPDIFRX_DC 部分使用；

为正确解码传入的 SPDIF 数据流，SPDIFRX_DC 应以至少比最大符号率（符号率=采样率*32*2）高 11 倍或者比音频采样率高 704 倍的时钟重新采样接收的数据。

例如，如果用户预期接收到最高 12.288 MHz 的符号率（对应音频采样率为 192Khz），则采样率至少为 135.2MHz。

接下来，我们看看本章需要用到的一些相关寄存器。

首先，是 SPDIFRX 控制寄存器：SPDIFRX_CR，该寄存器各位描述如图 54.1.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	INSEL[1:0](1)		
													rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	WFA(1)	NBTR[1:0](1)		CHSEL(1)	CBDMAEN(1)	PTMSK(1)	CUMSK(1)	VMSK(1)	PMSK(1)	DRFMT(1)		RXSTEO(1)	RXDMAEN(1)	SPDIFRXEN[1:0](1)	
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 54.1.8 SPDIFRX_CR 寄存器各位描述

该寄存器我们只介绍本章需要用到的一些位（下同）：

INSEL：这三个位，用于选择 SPDIFRX 的输入通道，STM32F7 总共有 4 个输入通道（0~3），我们硬件上连接在通道 1 上面，所以设置 INSEL=001 即可。

NBTR：这两个位用于设置在同步阶段允许的最大重试次数：00，表示不允许重试；01，表示允许重试 3 次；10，表示允许重试 15 次；11，表示允许重试 63 次。本章我们设置为 10，允许重试 15 次。

CHSEL：用于选择通道状态获取路径。我们设置为 0，选择从通道 A 获取通道状态。

CBDMAEN：用于设置通道状态和用户数据是否使能 DMA 接收。我们设置为 0，禁止 DMA 方式接收通道状态和用户数据。

PTMSK：用于设置报头类型屏蔽位。我们设置为 1，禁止将报头数据写入 SPDIFRX_DR。

CUMSK：用于设置通道状态和用户数据屏蔽位。我们设置为 1，禁止将通道状态和用户数据写入 SPDIFRX_DR。

VMSK：用于设置有效性位屏蔽位。我们设置为 1，禁止将有效性位写入 SPDIFRX_DR。

PMSK：用于设置奇偶校验屏蔽位。我们设置为 1，禁止将奇偶校验写入 SPDIFRX_DR。

DRFMT：用于设置 SPDIFRX_DR 的数据格式。我们设置为 00，选择右对齐格式（LSB）。

RXSTEO：用于设置是否使能立体声模式。我没设置为 1，使能立体声模式。

RXDMAEN：用于设置是否使能数据流 DMA 接收。我们设置为 1，使能 DMA 接收。

SPDIFRXEN：用于设置 SPDIFRX 的使能。实际上就是控制 SPDIFRX 的工作状态：00，

禁止 SPDIFRX；01，使能 SPDIFRX 同步；10，保留；11，使能 SPDIFRX 接收器。关于该寄存器的设置对 SPDIFRX 工作状态的影响，请参考图 54.1.6。

接下来，我们介绍 SPDIFRX 中断屏蔽寄存器：SPDIFRX_IMR，该寄存器的各位描述如图 54.1.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	IFEIE	SYNCDIE	SBLKIE	OVRIE	PERRIE	CSRNEIE	RXNEIE
									rw	rw	rw	rw	rw	rw	rw

图 54.1.9 SPDIFRX_IMR 寄存器各位描述

IFEIE：串行接口错误中断使能屏蔽位。我们设置为 1，当 SPDIFRX_SR 寄存器中的 SERR=1、TERR=1 或者 FERR=1 时，将产生 SPDIFRX 中断。

PERRIE：奇偶校验错误中断屏蔽位。我们设置为 1，当 SPDIFRX_SR 寄存器中的 PERR=1 时，将产生 SPDIFRX 中断。

接下来，我们介绍 SPDIFRX 状态寄存器：SPDIFRX_SR，该寄存器的各位描述如图 54.1.10 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	WIDTH5[14:0]														
	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
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TERR	SERR	FERR	SYNCD	SBD	OVR	PERR	CSRNE	RXNE
							r	r	r	r	r	r	r	r	r

图 54.1.10 SPDIFRX_SR 寄存器各位描述

WIDTH5：使用 SPDIFRX_CLK 计数 5 个符号的持续时间。它表示 5 个连续符号的时间内包含的 SPDIFRX_CLK 时钟周期数。该值可用于估算 SPDIFRX 的音频采样率。其精度受 SPDIFRX_CLK 的频率限制。其估算公式为：

Fs = 5 * SPDIFRX_CLK / (WIDTH5 * 64)

Fs 为估算的音频采样率。假定 SPDIFRX_CLK 为 84Mhz，WIDTH5 为 147。计算可得 Fs 为 44.6Khz，最接近的标准采样率为 44.1Khz，所以，我们基本可以确定采样率为 44.1Khz。

注意，需要在同步完成以后（SYNCD=1），才可以读取 WIDTH5 的值，判断采样率。

TERR：超时错误标志。当该位为 1 时，表示检测到序列错误。

SERR：同步错误标志。当该位为 1 时，表示检测到同步错误。

FERR：帧错误标志。当该位为 1 时，表示检测到曼彻斯特编码错误（帧错误）。

SYNCD：同步完成标志。当该位为 1 时，表示同步完成。

OVR：上溢错误标志。当该位为 1 时，表示检测到上溢错误。

PERR：奇偶校验错误标志。当该位为 1 时，表示检测到奇偶校验错误。

接下来，我们介绍 SPDIFRX 中断标志清零寄存器：SPDIFRX_IFCR，该寄存器的各位描述如图 54.1.11 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SYNCDCF	SBDCF	OVRCF	PERRCF	Res.	Res.
										w	w	w	w		

图 54.1.11 SPDIFRX_IFCR 寄存器各位描述

OVRCF：清零上溢错误标志。向该位写 1，可以清除上溢错误标志。

PERRCF：清零奇偶校验错误标志。向该位写 1，可以清除奇偶校验错误标志。

最后，我们介绍 SPDIFRX 数据寄存器：SPDIFRX_DR，该寄存器的各位描述如图 54.1.12 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	PT[1:0]		C	U	V	PE	DR[23:16]							
		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
DR[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

图 54.1.12 SPDIFRX_DR 寄存器各位描述

该寄存器，有三种数据格式可选（见图 54.1.7），此图为 DRFMT=00 时的格式，使用右对齐（LSB）格式的 SPDIFRX_DR 寄存器各位描述。该寄存器里面的 PT/C/U/V/PE 等位，我们都不用，我们只用低 24 位(DR[23:0])，用于读取音频数据。当进入接收模式以后（SPDIFRXEN=11），我们不停的读取 SPDIFRX_DR 的数据，并传送给 SAI 接口，再由 SAI 传输给 WM8978，就可以播放来自 SPDIFRX 接收到的音乐了。不过，我们采用 DMA 来传输，所以直接设置 DMA 的外设地址为 SPDIFRX_DR 即可。

SPDIFRX 的相关寄存器，就给大家介绍到这里，更详细的介绍，请大家参考《STM32F7 中文参考手册.pdf》第 34.5 节。

最后，我们看看要通过 STM32F767 的 SPDIFRX 接口接收光纤音频数据，并通过 SAI 驱动 WM8978 播放音乐的简要步骤。HAL 库中 SPDIFRX 相关的库函数定义和声明分布在源文件 stm32f7xx_hal_spdifrx.c 和头文件 stm32fxx_hal_spdifrx.h 中。具体欧配置步骤如下：

1）初始化 WM8978

最终要通过 WM8978 来输出音乐，我们需要先对其进行配置，详细的配置过程，请参考第 52 章。

2）初始化 SPDIF

此步需要初始化 SPDIFRX 对应的 IO 口、开启 SPDIFRX 时钟、设置 SPDIFRX 的模式和相关配置，主要通过对 SPDIFRX_CR 寄存器的配置来实现。

在 4.3 小节讲解时钟系统的时候讲解过，SPDIF 时钟由 PLLI2SP 提供，HAL 库中配置 PLLI2SP 时钟是通过函数 HAL_RCCEx_PeriphCLKConfig 来实现的，该函数我们在讲解时钟系统的时候已经给大家讲解，这里我们列出配置 SPDIF 时钟详细源码：

```
SPDIFCLK_Sture.PeriphClockSelection=RCC_PERIPHCLK_SPDIFRX;//SPDIF RX 时钟
SPDIFCLK_Sture.PLLI2S.PLLI2SN=316; //设置 PLLI2SN
SPDIFCLK_Sture.PLLI2S.PLLI2SP= RCC_PLLI2SP_DIV2;//设置 PLLI2SP：2 分频
HAL_RCCEx_PeriphCLKConfig(&SPDIFCLK_Sture);//设置时钟
```

SPDIF RX 时钟计算公式为：

SPDIF RX CLK 的频率=(HSE/pllm)*PLLI2SN/PLLI2SP

这里 HSE=25MHz,pllm 为系统初始化的时候设置为 25，当我们设置 PLLI2SN=316,，RLLI2SP 为 2 分频值 RCC_PLLI2SP_DIV2 之后我们可以得出，SPDIF RX 时钟频率为:25/25*316/2=158MHz。

HAL 库中初始化 SPDIF 函数为 HAL_SPDIFRX_Init,该函数声明如下：

```
HAL_StatusTypeDef HAL_SPDIFRX_Init(SPDIFRX_HandleTypeDef *hspdif);
```

该函数只有一个 SPDIFRX_HandleTypeDef 结构体类型指针类型入口参数 hspdif，接下来我们看看结构体 SPDIFRX_HandleTypeDef 定义，如下：

```
typedef struct
{
  SPDIFRX_TypeDef                  *Instance;
  SPDIFRX_InitTypeDef              Init;
  uint32_t                         *pRxBuffPtr;
  uint32_t                         *pCsBuffPtr;
  __IO uint16_t                    RxXferSize;
  __IO uint16_t                     RxXferCount;
  __IO uint16_t                    CsXferSize;
  __IO uint16_t                    CsXferCount;
  DMA_HandleTypeDef                *hdmaCsRx;
  DMA_HandleTypeDef                *hdmaDrRx;
  __IO HAL_LockTypeDef             Lock;
  __IO HAL_SPDIFRX_StateTypeDef    State;
  __IO uint32_t   ErrorCode;
}SPDIFRX_HandleTypeDef;
```

对于 SPDIFRX 初始化，这里我们主要讲解 Init 成员变量，该成员变量用来设置 SPDIFRX 的初始化参数，为 SPDIFRX_InitTypeDef 结构体类型，该结构体定义为：

```
typedef struct
{
  uint32_t InputSelection;          //选择 SPDIFRX 的输入通道
  uint32_t Retries;                 //设置在同步阶段允许的最大重试次数
  uint32_t WaitForActivity;         //执行同步前是否等待 SPDIFRX_IN 线路上的活动
  uint32_t ChannelSelection;        //选择通道状态获取路径
  uint32_t DataFormat;              //设置 SPDIFRX_DR 的数据格式
  uint32_t StereoMode;              //设置是否使能立体声模式
  uint32_t PreambleTypeMask;        //设置报头类型屏蔽位
  uint32_t ChannelStatusMask;       //设置通道状态和用户数据屏蔽位
  uint32_t ValidityBitMask;         //设置有效性位屏蔽位
  uint32_t ParityErrorMask;         //设置奇偶校验屏蔽位
}SPDIFRX_InitTypeDef;
```

该结构体各个成员变量的含义都在上面注释了，实际上配置的是 SPDIFRX_CR 寄存器各个位，有不理解的地方可以对照中文参考手册中寄存器位定义描述来理解。

和其他外设接口一样，HAL 库同样提供了 SPDIFRX 初始化回调函数：

```
void HAL_SPDIFRX_MspInit(SPDIFRX_HandleTypeDef *hspdif);
```

3）设置 SPDIFRX 的 DMA

我们通过 DMA 双缓冲模式来接收 SPDIF RX 接收到的数据，从而提高效率。每当一个缓冲区数据接收满以后，硬件自动切换为下一个缓冲区，同时可以将满的缓冲区数据通过 SAI 接口，传输给 WM8978，从而实现音乐播放。SPDIFRX 使用 DMA 双缓冲接收数据的过程如图 54.1.13 所示：

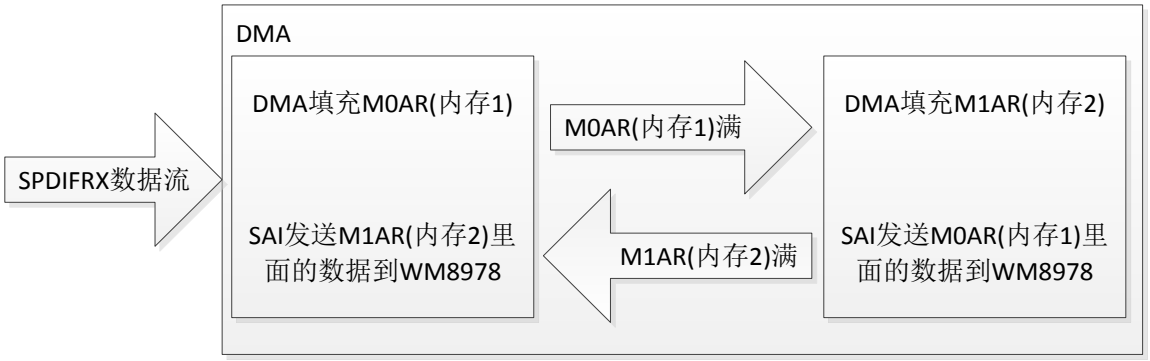


图 54.1.13 DMA 双缓冲发送音频数据流框图

4）配置 SAI

在 SPDIFRX 接口同步完成，且与 SAI 接口完成时钟同步（后续介绍），并成功获取音频数据流的采样率以后，就可以配置 SAI 接口，然后将 SPDIFRX 接收到的音频数据流，传输给 WM8978，实现音频播放。此过程主要配置 SAI 的采样率、工作模式、协议、时钟电平特性、slot 相关参数和 DMA 等。我们同样通过 DMA 双缓冲模式将数据传输给 WM8978。SAI 接口的 DMA 处理流程，请参考第五十二章。

5）编写 SPDIFRX 中断服务函数

当 SPDIFRX 传输出现错误的时候（比如突然断开光纤线或采样率发生了变化），需要在 SPDIFRX 中断服务函数里面对其进行处理，当发生不可恢复的错误时，需要重新设置 SPDIFRX 进入 IDLE 状态，以便重新同步。SPDIFRX 中断服务函数为 SPDIF_RX_IRQHandler。

6）开启 DMA 传输

最后，我们就只需要开启 SPDIFRX 和 SAI 的 DMA 传输，就可以实现 SPDIF 接收光纤音频数据，并通过 WM8978 播放出来。此时，就可以在 WM8978 的耳机和喇叭通道听到光纤传输过来的音乐了。

54.2 硬件设计

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，则初始化 WM8978 的 DAC 工作，并开启 DAC 输出，随后设置 SPDIFRX 的 DMA 和回调函数. 然后进入死循环等待，不停的检测 SPDIF 的连接状态,当 SPDIF RX 同步完成，且与 SAI 时钟同步完成后，开启 SPDIFRX 和 SAI 的 DMA 数据传输。此时，在屏幕上会显示当前的音频信号采样率，同时在喇叭/耳机,可以听到音乐。另外，我们可以通过 KEY_UP 和 KEY1 来调节音量，KEY_UP 用于增加音量，KEY1 用于减少音量。

本实验用到的硬件资源如下：

1） 指示灯 DS0
2） 两个按键（KEY_UP/KEY1）
3） 串口
4） LCD 模块

5） SPI FLASH
6） WM8978
7） 光纤接口（SPDIFRX）
8） SAI

其中，除了光纤接口，其他资源在之前的学习中都已经介绍过了。光纤接口我们使用的是 DLR1150 光纤座，该接口和 STM32F767 的连接，如图 54.2.1 所示：

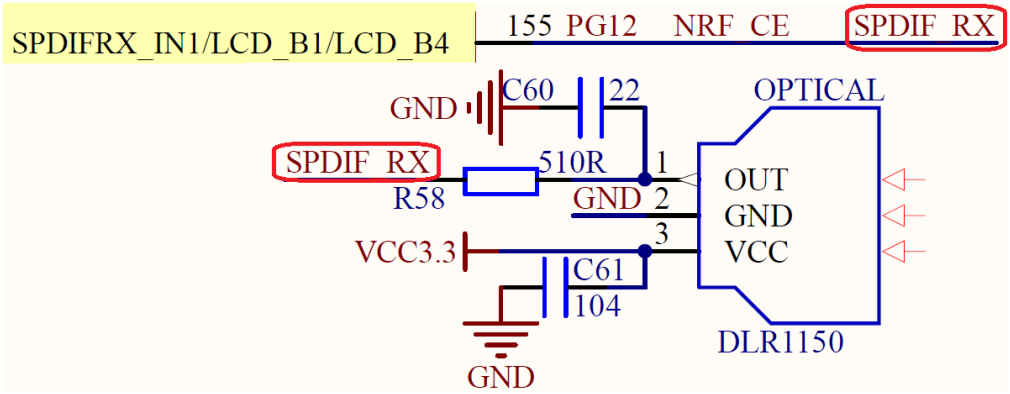


图 54.2.1 光纤接口与 STM32F767 连接原理图

图中 DLR1150 就是我们所使用的光纤座（即光纤接口），它连接在 STM32F767 的 PG12 脚上，是 SPDIFRX 的输入通道 1。

注意：SPDIF_RX 和 NRF_CE 共用 PG12，他们不可以同时使用！

另外，本实验还需要用到一个音频光纤信号发送设备和一条光纤线，这些都需要大家自备。光纤音频发送设备推荐购买 ALIENTEK 的贝斯（BASE）蓝牙音频接收器，支持光纤输出，购买地址：http://www.openedv.com/thread-86409-1-1.html

54.3 软件设计

打开本章实验工程可以看到，我们在 HARDWARE 分组下面添加了 spdif.c 文件，同时包含了头文件 spdif.h。打开 spdif.c，内容如下：

```
    #define AUDIODATA_SIZE 200
    u32 spdif_audiobuff[2][AUDIODATA_SIZE]; //音频数据双缓冲区,200*4=800 字节
    u32 spdif_controlbuff[10];                    //SPDIF 传输通道状态和用户信息

    spdif_struct spdif_dev;                       //SPDIF 控制结构体

    SPDIFRX_HandleTypeDef SPDIFIN1_Handle;   //SPDIF IN1 句柄
    DMA_HandleTypeDef SPDIF_DTDMA_Handler;   //SPDIF 音频数据 DMA

    //初始化 SPDIF
 void SPDIFRX_Init(void)
{
     spdif_dev.spdif_clk=1580000; //默认为 158M，单位为 100HZ
     SPDIFCLK_Config();          //配置 SPDIF 时钟
     SPDIFIN1_Handle.Instance=SPDIFRX;
      SPDIFIN1_Handle.Init.InputSelection=SPDIFRX_INPUT_IN1; //SPDIF 输入 1
```

```
    SPDIFIN1_Handle.Init.Retries=SPDIFRX_MAXRETRIES_15; //同步阶段允许重试次数
    SPDIFIN1_Handle.Init.WaitForActivity=SPDIFRX_WAITFORACTIVITY_ON;//等待同步
    SPDIFIN1_Handle.Init.ChannelSelection=SPDIFRX_CHANNEL_A;
                                                    //控制流从通道 A 获取通道状态
    SPDIFIN1_Handle.Init.DataFormat=SPDIFRX_DATAFORMAT_32BITS;     //右对齐
SPDIFIN1_Handle.Init.StereoMode=SPDIFRX_STEREOMODE_ENABLE;//使能立体声模式
    SPDIFIN1_Handle.Init.PreambleTypeMask=SPDIFRX_PREAMBLETYPEMASK_OFF;
                                                    //报头类型不复制到 SPDIFRX_DR 中
    SPDIFIN1_Handle.Init.ChannelStatusMask=SPDIFRX_CHANNELSTATUS_OFF;
                                                //通道状态和用户位不复制到 SPDIFRX_DR 中
    SPDIFIN1_Handle.Init.ValidityBitMask=SPDIFRX_VALIDITYMASK_ON;
                                                //有效性位不复制到 SPDIFRX_DR 中
      SPDIFIN1_Handle.Init.ParityErrorMask=SPDIFRX_PARITYERRORMASK_ON;
                                                //奇偶校验错误位不复制到 SPDIFRX_DR 中
      HAL_SPDIFRX_Init(&SPDIFIN1_Handle);
      SPDIFIN1_Handle.Instance->CR|=SPDIFRX_CR_RXDMAEN;
                                              //SPDIF 音频数据使用 DMA 来接收
      SPDIFIN1_Handle.Instance->CR|=SPDIFRX_CR_CBDMAEN;
                                              //SPDIF 传输通道状态和用户信息使用 DMA 来接收
      //使能 SPDIF 的串行接口错误中断、上溢错误和奇偶校验错误
      __HAL_SPDIFRX_ENABLE_IT(&SPDIFIN1_Handle,SPDIFRX_IT_IFEIE| \
                                                          SPDIFRX_IT_PERRIE);
      SPDIF_AUDIODATA_DMA_Init((u32*)&spdif_audiobuff[0],(u32*)& \
                                                  spdif_audiobuff[1],AUDIODATA_SIZE,2);
}

//SPDIF 时钟配置,设置为 158MHZ
void SPDIFCLK_Config(void)
{
      RCC_PeriphCLKInitTypeDef SPDIFCLK_Sture;

      SPDIFCLK_Sture.PeriphClockSelection=RCC_PERIPHCLK_SPDIFRX; //SPDIF RX 时钟
      SPDIFCLK_Sture.PLLI2S.PLLI2SN=316; //设置 PLLI2SN
      SPDIFCLK_Sture.PLLI2S.PLLI2SP=RCC_PLLI2SP_DIV2; //设置 PLLI2SP:2 分频
      HAL_RCCEx_PeriphCLKConfig(&SPDIFCLK_Sture); //设置时钟
}

//SPDIF  音频数据接收 DMA 配置
//设置为双缓冲模式,并开启 DMA 传输完成中断
//buf0:M0AR 地址.   buf1:M1AR 地址.
//num:每次传输数据量  width:位宽(存储器和外设,同时设置),0,8 位;1,16 位;2,32 位;
void SPDIF_AUDIODATA_DMA_Init(u32* buf0,u32 *buf1,u16 num,u8 width)
{
```

```
    u32 memwidth=0,perwidth=0;          //外设和存储器位宽
    switch(width)
    {
        case 0:             //8 位
            memwidth=DMA_MDATAALIGN_BYTE;
            perwidth=DMA_PDATAALIGN_BYTE;
            break;
        case 1:             //16 位
            memwidth=DMA_MDATAALIGN_HALFWORD;
            perwidth=DMA_PDATAALIGN_HALFWORD;
            break;
        case 2:             //32 位
            memwidth=DMA_MDATAALIGN_WORD;
            perwidth=DMA_PDATAALIGN_WORD;
            break;

    }
    __HAL_RCC_DMA1_CLK_ENABLE();        //使能 DMA1 时钟
    __HAL_LINKDMA(&SPDIFIN1_Handle,hdmaDrRx,SPDIF_DTDMA_Handler);
                                        //将 DMA 与 SPDIF 联系起来
    SPDIF_DTDMA_Handler.Instance=DMA1_Stream1;     //DMA1 数据流 1
    SPDIF_DTDMA_Handler.Init.Channel=DMA_CHANNEL_0;   //通道 0
    SPDIF_DTDMA_Handler.Init.Direction=DMA_PERIPH_TO_MEMORY; //外设到存储器
    SPDIF_DTDMA_Handler.Init.PeriphInc=DMA_PINC_DISABLE;      //外设非增量模式
    SPDIF_DTDMA_Handler.Init.MemInc=DMA_MINC_ENABLE;          //存储器增量模式
    SPDIF_DTDMA_Handler.Init.PeriphDataAlignment=perwidth;   //外设数据长度:16/32 位
    SPDIF_DTDMA_Handler.Init.MemDataAlignment=memwidth;//存储器数据长度 16/32 位
    SPDIF_DTDMA_Handler.Init.Mode=DMA_CIRCULAR; //使用循环模式
    SPDIF_DTDMA_Handler.Init.Priority=DMA_PRIORITY_HIGH; //最高优先级
    SPDIF_DTDMA_Handler.Init.FIFOMode=DMA_FIFOMODE_DISABLE;//不使用 FIFO
    SPDIF_DTDMA_Handler.Init.MemBurst=DMA_MBURST_SINGLE;//单次突发传输
    SPDIF_DTDMA_Handler.Init.PeriphBurst=DMA_PBURST_SINGLE;//突发单次传输
    HAL_DMA_DeInit(&SPDIF_DTDMA_Handler);     //先清除以前的设置
    HAL_DMA_Init(&SPDIF_DTDMA_Handler);       //初始化 DMA

    HAL_DMAEx_MultiBufferStart(&SPDIF_DTDMA_Handler,(u32)&SPDIFRX->DR,
                               (u32)buf0,(u32)buf1,num);//开启双缓冲
}

//进入同步状态，同步完成以后进入接收状态
//返回值:0  未同步;1  同步
u8 WaitSync_TORecv(void)
{
```

```
    u8 flag=0;
    u8 timeout=0;
    __HAL_SPDIFRX_SYNC(&SPDIFIN1_Handle);
    while(__HAL_SPDIFRX_GET_FLAG(&SPDIFIN1_Handle,
                                 SPDIFRX_FLAG_SYNCD)==0)//等待同步完成
    {
        timeout++;
        delay_ms(5);
        if (timeout>100) break;//超时，跳出
    }
    if(timeout>100) flag=0;//未同步
    else //同步完成
    {
        flag=1;
        __HAL_SPDIFRX_RCV(&SPDIFIN1_Handle);//同步完成，进入接收阶段
    }
    return flag;
}

//获取 SPDIF 收到的音频采样率
void SPDIF_GetRate(void)
{
    u16 spdif_w5;
    u32 spdif_rate;

    spdif_w5=(SPDIFIN1_Handle.Instance->SR)>>16;
    spdif_rate=(spdif_dev.spdif_clk*5)/(spdif_w5&0X7FFF);
    spdif_rate>>=6;         //除以 64
    spdif_rate*=100;        //乘以 100，得到最终的实际采样率

    if((44100-1500<=spdif_rate)&&(spdif_rate<=44100+1500))
        spdif_dev.spdifrate=44100; //44.1K 的采样率
    else if((48000-1500<=spdif_rate)&&(spdif_rate<=48000+1500))
        spdif_dev.spdifrate=48000; //48K 的采样率
    else if((88200-1500<=spdif_rate)&&(spdif_rate<=88200+1500))
        spdif_dev.spdifrate=88200; //88.2K 的采样率
    else if((96000-1500<=spdif_rate)&&(spdif_rate<=96000+1500))
        spdif_dev.spdifrate=96000; //96K 的采样率
    else if((176400-1500<=spdif_rate)&&(spdif_rate<=176400+1500))
        spdif_dev.spdifrate=176400; //176.4K 的采
    else if((192000-1500<=spdif_rate)&&(spdif_rate<=192000+1500))
        spdif_dev.spdifrate=192000; //192K 的采
    else spdif_dev.spdifrate=0;
```

```
}

//SPDIF 底层 IO 初始化和时钟使能
//此函数会被 HAL_SPDIF_Init()调用
//hltdc:SPDIF 句柄
void HAL_SPDIFRX_MspInit(SPDIFRX_HandleTypeDef *hspdif)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_SPDIFRX_CLK_ENABLE();     //使能 SPDIF RX 时钟
    __HAL_RCC_GPIOG_CLK_ENABLE();       //使能 GPIOG 时钟

    //初始化 PG12，SPDIF IN 引脚
    GPIO_Initure.Pin=GPIO_PIN_12;               //PG12，SPDIF IN 引脚
    GPIO_Initure.Mode=GPIO_MODE_AF_PP;          //复用
    GPIO_Initure.Pull=GPIO_NOPULL;              //无上下拉
    GPIO_Initure.Speed=GPIO_SPEED_HIGH;         //高速
    GPIO_Initure.Alternate=GPIO_AF7_SPDIFRX;    //复用为 SPDIF RX
    HAL_GPIO_Init(GPIOG,&GPIO_Initure);

    HAL_NVIC_SetPriority(SPDIF_RX_IRQn,1,0); //SPDIF 中断
    HAL_NVIC_EnableIRQ(SPDIF_RX_IRQn);
}

//SPDIF 接收中断服务函数
void SPDIF_RX_IRQHandler(void)
{
    //发生超时、同步和帧错误中断,这三个中断一定要处理!
    if( __HAL_SPDIFRX_GET_FLAG(&SPDIFIN1_Handle,SPDIFRX_FLAG_FERR)||\
        __HAL_SPDIFRX_GET_FLAG(&SPDIFIN1_Handle,SPDIFRX_FLAG_SERR)||\
        __HAL_SPDIFRX_GET_FLAG(&SPDIFIN1_Handle,SPDIFRX_FLAG_TERR))
    {
        SPDIF_Play_Stop();//发生错误，关闭 SPDIF 播放\
       //当发生超时、同步和帧错误的时候要将 SPDIFRXEN 写 0 来清除中断
        __HAL_SPDIFRX_IDLE(&SPDIFIN1_Handle);
       //当清除中断以后需要重新将 SPDIF 设置为接收模式
        __HAL_SPDIFRX_RCV(&SPDIFIN1_Handle);
    }else if(__HAL_SPDIFRX_GET_FLAG(&SPDIFIN1_Handle,SPDIFRX_FLAG_OVR))
    {
        __HAL_SPDIFRX_CLEAR_IT(&SPDIFIN1_Handle, SPDIFRX_FLAG_OVR);
    }else if(__HAL_SPDIFRX_GET_FLAG(&SPDIFIN1_Handle,SPDIFRX_FLAG_PERR))
    {
        __HAL_SPDIFRX_CLEAR_IT(&SPDIFIN1_Handle, SPDIFRX_FLAG_PERR);
```

```
        }
    }

    //配置音频接口
    //AudioFreq:音频采样率
    uint32_t SPDIF_AUDIO_Init(uint32_t AudioFreq)
    {
        SAIA_Init(SAI_MODEMASTER_TX,SAI_CLOCKSTROBING_RISINGEDGE,
                                    SAI_DATASIZE_16);//设置 SAI,主发送,16 位数据
        SAIA_SampleRate_Set(AudioFreq);     //设置采样率
        SAIA_TX_DMA_Init((u8*)&spdif_audiobuff[0],(u8*)&spdif_audiobuff[1],
                                    AUDIODATA_SIZE*2,1); //配置 TX DMA,16 位
        SAI_Play_Start();                               //开启 DMA
        return 0;
    }

    //SPDIF 开始播放
    void SPDIF_Play_Start(void)
    {
        spdif_dev.connsta=1;        //标记已经打开 SPDIF
        __HAL_DMA_ENABLE(&SPDIF_DTDMA_Handler); //开启 SPDIF DMA 传输
    }
    //SPDIF 关闭
    void SPDIF_Play_Stop(void)
    {
        spdif_dev.connsta=0;        //标记已经关闭 SPDIF
        __HAL_DMA_DISABLE(&SPDIF_DTDMA_Handler);//关闭 SPDIF DMA 传输
        //两个缓冲区一定要清零，否则在断开的时候会有很大杂音!!!
        memset((u8*)&spdif_audiobuff[0],0,AUDIODATA_SIZE*4);
        memset((u8*)&spdif_audiobuff[1],0,AUDIODATA_SIZE*4);
    }
```

这里总共 10 个函数，接下来，我们分别介绍。

1，SPDIFRX_Init 函数

该函数用于初始化 SPDIFRX 接口，包括设置 SPDIFRX 时钟、设置 SPDIFRX 相关参数等，最后开启了奇偶校验错误中断和接口错误中断，用于处理当接收数据出现错误时（包括光纤断开、采样率切换等），重新同步。

2，HAL_SPDIFRX_MspInit 函数

该函数是 SPDIFRX 的初始化回调函数，主要用来配置时钟使能，IO 口模式和中断分组等。

3，SPDIFCLK_Config 函数

该函数用于设置 SPDIFRX 接口的时钟频率，SPDIFRX 的时钟频率来自 PLLI2S 的 N 分频，我们默认设置 N 分频为 2，所以 SPDIFRX_CLK 的计算公式就是：

SPDIFRX_CLK=PLLI2SN/2

单位为 Mhz，另外，SPDIFRX 接口的时钟频率必须大于音频采样率的 704 倍，对于 192Khz

的音频采样率来说，SPDIFRX_CLK 的频率必须大于 135.168Mhz，才可以识别，所以我们在 SPDIF_RX_Init 函数里面，通过 SPDIFCLK_Config 函数，设置 SPDIFRX_CLK 为 158Mhz，以支持最高 192Khz 的音频采样率。

4，WaitSync_TORecv 函数

该函数用于等待 SPDIFRX 同步完成，SPDIF 必须在等待同步完成以后，才可以获取音频采样率，然后进入接收状态，接收音频数据。

5，SPDIF_AUDIODATA_DMA_Init 函数

该函数用于配置 SPDIFRX 接口的数据接收 DMA，使用双缓冲模式，将 SPDIFRX_DR 的数据，存储到内存里面，实现音频数据接收。

6，SPDIF_GetRate 函数

该函数用于获取 SPDIFRX 识别到的音频采样率。该函数首先通过 SPDIFRX_SR 寄存器的 WIDTH5 位，获取大概的音频采样率，然后标准化为最接近的音频采样率。该函数必须在 SPDIF 同步完成以后才可以调用。

7，SPDIF_RX_IRQHandler 函数

该函数用于处理 SPDIFRX 接口的错误中断，当发生超时错误、同步错误和帧错误的时候，必须停止 SPDIFRX 接口，然后重新进入 IDLE 状态，以便重新同步。

8，SPDIF_AUDIO_Init 函数

该函数用来配置音频接口采样率。

9，SPDIF_Play_Start 和 SPDIF_Play_Stop 函数

这两个函数，前者用于启动 SPDIFRX DMA 传输，在所有配置和状态都正常的情况下，用于启动 SPDIF 音频播放。后者用于关闭 SPDIFRX DMA 传输，停止 SPDIF 播放。其中 spdif_dev 结构体用于控制 SPDIF 接收状态，结构体定义（见 spdif.h）如下：

```
//SPDIF 控制结构体
typedef struct
{
    u8 connsta;      //连接状态，0 未连接;1 连接上
    u32 spdifrate;   //SPDIF 采样率
    u32 spdif_clk;   //SPDIF 时钟，默认为 158M
}spdif_struct;
```

connsta 表示 SPDIF 连接状态，当 SPDIF 开启 DMA 传输的时候，设置为 1，表示 SPDIF 正在播放。当其为 0 的时候，表示 SPDIF 停止播放。

spdifrate 表示 SPDIF 识别到的音频采样率，单位为 Hz。

spdif_clk 表示 SPDIFRX_CLK 的时钟频率，单位为 Hz。

最后，我们看看 main.c 文件代码：

```
//显示采样率
//samplerate:音频采样率(单位:Hz)
void spdif_show_samplerate(u32 samplerate)
{
  ……//此处省略部分显示代码
}

int main(void)
{
```

```
u8 i,strbuff[20];
u8 volume=45,key=0;

Cache_Enable();                 //打开 L1-Cache
HAL_Init();                     //初始化 HAL 库
Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
delay_init(216);                //延时初始化
uart_init(115200);              //串口初始化
usmart_dev.init(108);           //初始化 USMART
LED_Init();                     //初始化 LED
KEY_Init();                     //初始化按键
SDRAM_Init();                   //初始化 SDRAM
LCD_Init();                     //初始化 LCD
W25QXX_Init();                  //初始化 W25Q256
WM8978_Init();                  //初始化 WM8978
WM8978_HPvol_Set(volume,volume);//耳机音量设置
WM8978_SPKvol_Set(volume);      //喇叭音量设置
SPDIFRX_Init();                 //SPDIF 初始化

my_mem_init(SRAMIN);            //初始化内部内存池
my_mem_init(SRAMEX);            //初始化外部 SDRAM 内存池
my_mem_init(SRAMDTCM);          //初始化内部 DTCM 内存池

exfuns_init();                  //为 fatfs 相关变量申请内存
f_mount(fs[0],"0:",1);          //挂载 SD 卡
POINT_COLOR=RED;

while(font_init())              //检查字库
{
    LCD_ShowString(30,50,200,16,16,"Font Error!");
    delay_ms(200);
    LCD_Fill(30,50,240,66,WHITE);//清除显示
    delay_ms(200);
}
POINT_COLOR=RED;
Show_Str(30,40,200,16,"阿波罗 STM32F4/F7 开发板",16,0);
Show_Str(30,60,200,16,"SPDIF(光纤音频)实验",16,0);
Show_Str(30,80,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(30,100,200,16,"2016 年 8 月 2 日",16,0);
Show_Str(30,120,200,16,"KEY_UP:VOL+    KEY1:VOL-",16,0);
Show_Str(30,150,200,16,"音量:",16,0);
Show_Str(30,170,200,16,"采样率:",16,0);
POINT_COLOR=BLUE;
```

```
LCD_ShowxNum(30+40,150,volume,2,16,0X80);      //显示音量

spdif_show_samplerate(0);                      //显示采样率
WM8978_ADDA_Cfg(1,0);          //开启 DAC
WM8978_Input_Cfg(0,0,0);       //关闭输入通道
WM8978_Output_Cfg(1,0);        //开启 DAC 输出
while(1)
{
    key=KEY_Scan(1);
    if(key==WKUP_PRES||key==KEY1_PRES)      //音量控制
    {
        if(key==WKUP_PRES)
        {
            volume++;
            if(volume>63)volume=63;
        }else
        {
            if(volume>0)volume--;
        }
        WM8978_HPvol_Set(volume,volume);       //设置耳机音量设置
        WM8978_SPKvol_Set(volume);             //设置喇叭音量设置
        LCD_ShowxNum(30+40,150,volume,2,16,0X80);//显示音量
    }
    if(spdif_dev.connsta==0)//未连接
    {
        if(WaitSync_TORecv())//等待同步
        {
            SPDIF_GetRate();//获得采样率
            spdif_show_samplerate(spdif_dev.spdifrate);      //显示采样率.
            SPDIF_Play_Start(); //同步完成，打开 SPDIF
            if(spdif_dev.spdifrate)
            {
                SPDIF_AUDIO_Init(spdif_dev.spdifrate);
            }
        }
    }
    delay_ms(20);
    i++;
    if(i>10)
    {
        i=0;
        LED0_Toggle;
    }
```

```
        }
    }
```

在 main 函数里面我们初始化 WM8978 和 SPDIFRX 等外设，随后在死循环里面等待 SPDIF 同步完成，在首次同步完成以后，获取音频采样率，然后设置 SAI 接口，设置 SAI 的音频采样率，标记 SAI 与 SPDIF 时钟同步完成，然后重新同步（SPDIF 和光纤信号同步），重新同步完成以后，开启 SPDIF 和 SAI 的 DMA 传输，启动音频播放，此时就可以从喇叭或者耳机听到光纤传输过来的音频信号了。在主循环里面我们还加入了音量调节的代码，可以通过 KEY_UP 和 KEY1 调节音量的大小。

至此，本实验的软件设计部分结束。

54.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，程序先检测字库，然后进入主循环，等待 SPDIF 信号（光纤信号）的输入，屏幕提示：正在识别…。如图 54.4.1 所示：

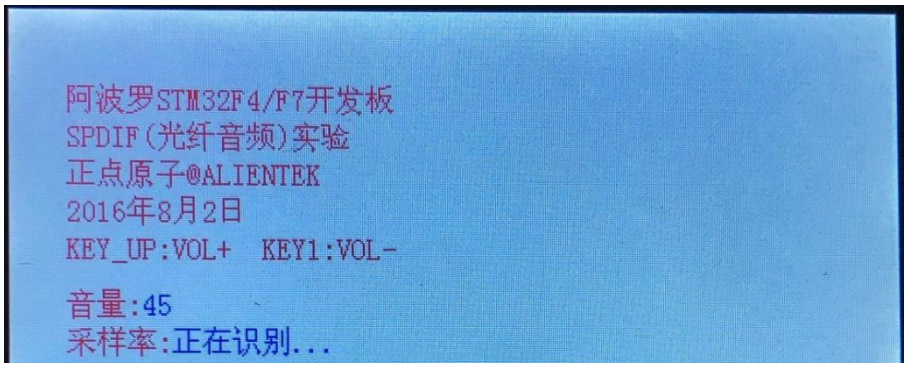


图 54.4.1 等待 SPDIF 信号输入（光纤音频信号）

此时，我们利用光纤线连接开发板的光纤座和光纤音频输出设备（比如贝斯蓝牙音频接收器），然后播放歌曲，就可以看到屏幕显示了音频信号的采样率，然后，通过耳机和板载喇叭就可以欣赏所播放的歌曲了。如图 54.4.2 所示：

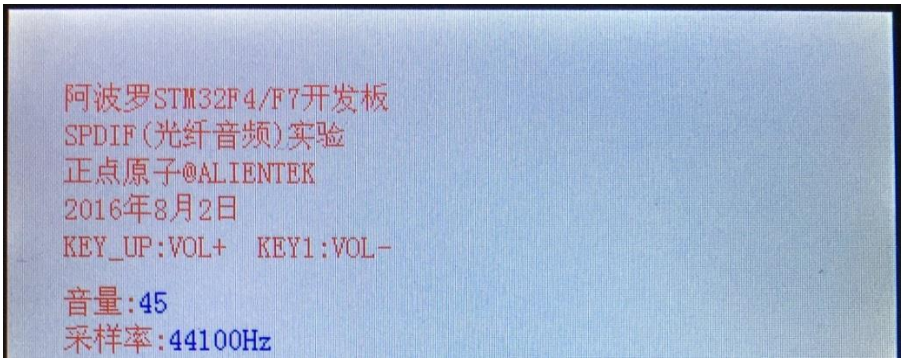


图 54.4.2 识别采样率，播放音乐

此时，我们按下 KEY_UP 按键，可以提高音频，按 KEY1 按键，可以降低音量。STM32F7 自带的 SPDIFRX 功能，可以实现对光纤、同轴等数字音频信号的解码，在 HiFi 解码方面可以有广泛的应用前景。

最后，本例程支持 44.1K、48K、88.2K 和 96Khz 等音频采样率，暂不支持 176.4K 和 192K 等高采样率的音频信号。

第五十五章 视频播放器实验

STM32F767 自带了硬件 JPEG 解码器，完全可以用来播放视频!本章，我们将使用 STM32F7 的硬件 JPEG 解码器来实现播放 AVI 视频（MJPEG 编码），本章我们将实现一个简单的视频播放器，实现 AVI 视频播放。本章分为如下几个部：

55.1 AVI 简介
55.2 硬件设计
55.3 软件设计
55.4 下载验证

555.1 AVI 简介

本章，我们使用 STM32F7 的硬件 JPEG 解码器，来实现 MJPG 编码的 AVI 格式视频播放，硬件 JPEG 解码器我们在第五十章已经介绍过了。接下来给大家简单介绍一下 AVI 格式。

AVI 是音频视频交错(Audio Video Interleaved)的英文缩写，它是微软开发的一种符合 RIFF 文件规范的数字音频与视频文件格式，原先用于 Microsoft Video for Windows (简称 VFW)环境，现在已被多数操作系统直接支持。

AVI 格式允许视频和音频交错在一起同步播放，支持 256 色和 RLE 压缩，但 AVI 文件并未限定压缩标准，AVI 仅仅是一个容器，用不同压缩算法生成的 AVI 文件，必须使用相应的解压缩算法才能播放出来。比如本章，我们使用的 AVI，其音频数据采用 16 位线性 PCM 格式（未压缩），而视频数据，则采用 MJPG 编码方式。

在介绍 AVI 文件前，我们要先来看看 RIFF 文件结构。AVI 文件采用的是 RIFF 文件结构方式，RIFF（Resource Interchange File Format，资源互换文件格式）是微软定义的一种用于管理 WINDOWS 环境中多媒体数据的文件格式，波形音频 WAVE，MIDI 和数字视频 AVI 都采用这种格式存储。构造 RIFF 文件的基本单元叫做数据块（Chunk），每个数据块包含 3 个部分，

1、4 字节的数据块标记（或者叫做数据块的 ID）
2、数据块的大小
3、数据

整个 RIFF 文件可以看成一个数据块，其数据块 ID 为 RIFF，称为 RIFF 块。一个 RIFF 文件中只允许存在一个 RIFF 块。RIFF 块中包含一系列的子块，其中有一种子块的 ID 为"LIST"，称为 LIST 块，LIST 块中可以再包含一系列的子块，但除了 LIST 块外的其他所有的子块都不能再包含子块。

RIFF 和 LIST 块分别比普通的数据块多一个被称为形式类型（Form Type）和列表类型（List Type）的数据域，其组成如下：

1、4 字节的数据块标记（Chunk ID）
2、数据块的大小
3、4 字节的形式类型或者列表类型（ID）
4、数据

下面我们看看 AVI 文件的结构。AVI 文件是目前使用的最复杂的 RIFF 文件，它能同时存储同步表现的音频视频数据。AVI 的 RIFF 块的形式类型（Form Type）是 AVI，它一般包含 3 个子块，如下所述：

1、信息块，一个 ID 为"hdrl"的 LIST 块，定义 AVI 文件的数据格式。
2、数据块，一个 ID 为 "movi"的 LIST 块，包含 AVI 的音视频序列数据。
3、索引块，ID 为"idxl"的子块，定义"movi"LIST 块的索引数据，是可选块（不一定有）。

接下来，我们详细介绍下 AVI 文件的各子块构造，AVI 文件的结构如图 55.1.1 所示：

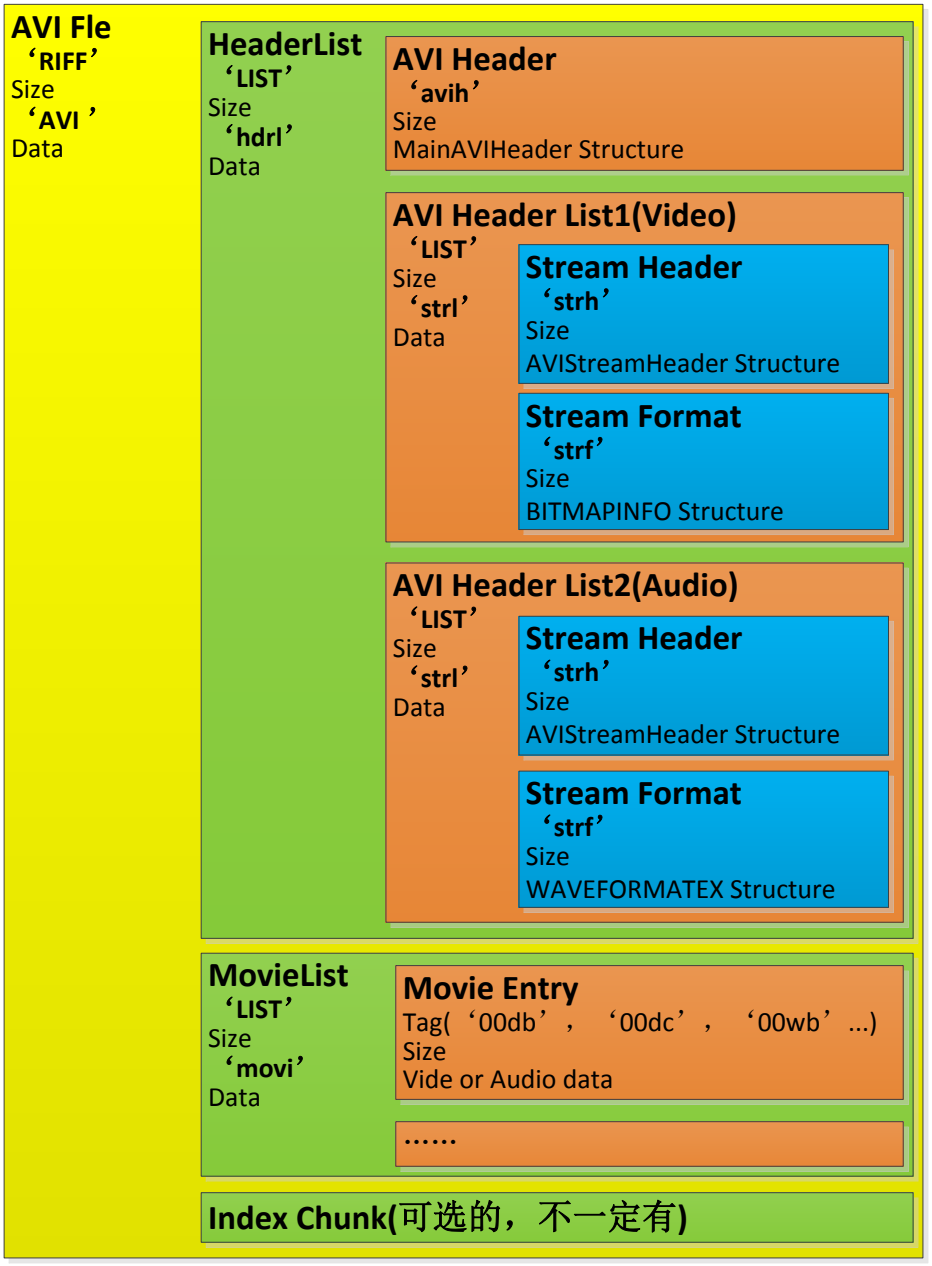


图 55.1.1 AVI 文件结构图

从上图可以看出(注意'AVI '，是带了一个空格的)，AVI 文件，由：信息块(HeaderList)、数据块(MovieList)和索引块(Index Chunk)等三部分组成，下面，我们分别介绍这几个部分。

1，信息块(HeaderList)

信息块，即 ID 为"hdrl"的 LIST 块，它包含文件的通用信息，定义数据格式，所用的压缩算法等参数等。hdrl 块还包括了一系列的字块，首先是：avih 块，用于记录 AVI 的全局信息，比如数据流的数量，视频图像的宽度和高度等信息，avih 块(结构体都有把 BlockID 和 BlockSize 包含进来，下同）的定义如下：

```
//avih 子块信息
typedef struct
```

```
{
    u32 BlockID;            //块标志:avih==0X61766968
    u32 BlockSize;          //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    u32 SecPerFrame;        //视频帧间隔时间(单位为 us)
    u32 MaxByteSec;         //最大数据传输率,字节/秒
    u32 PaddingGranularity; //数据填充的粒度
    u32 Flags;              //AVI 文件的全局标记，比如是否含有索引块等
    u32 TotalFrame;         //文件总帧数
    u32 InitFrames;         //为交互格式指定初始帧数（非交互格式应该指定为 0）
    u32 Streams;            //包含的数据流种类个数,通常为 2
    u32 RefBufSize;         //建议读取本文件的缓存大小（应能容纳最大的块）
    u32 Width;              //图像宽
    u32 Height;             //图像高
    u32 Reserved[4];        //保留
}AVIH_HEADER;
```

这里有很多我们要用到的信息，比如 SecPerFrame，通过该参数，我们可以知道每秒钟的帧率，也就知道了每秒钟需要解码多少帧图片，才能正常播放。TotalFrame 告诉我们整个视频有多少帧，结合 SecPerFrame 参数，就可以很方便计算整个视频的时间了。Streams 告诉我们数据流的种类数，一般是 2，即包含视频数据流和音频数据流。

在 avih 块之后，是一个或者多个 strl 子列表，文件中有多少种数据流（即前面的 Streams），就有多少个 strl 子列表。每个 strl 子列表，至少包括一个 strh（Stream Header）块和一个 strf（Stream Format）块，还有一个可选的 strn（Stream Name）块（未列出）。注意：strl 子列表出现的顺序与媒体流的编号（比如：00dc，前面的 00，即媒体流编号 00）是对应的，比如第一个 strl 子列表说明的是第一个流（Stream 0），假设是视频流，则表征视频数据块的四字符码为“00dc”，第二个 strl 子列表说明的是第二个流（Stream 1），假设是音频流，则表征音频数据块的四字符码为“01dw”，以此类推。

先看 strh 子块，该块用于说明这个流的头信息，定义如下：

```
//strh 流头子块信息(strh∈strl)
typedef struct
{
    u32 BlockID;            //块标志:strh==0X73747268
    u32 BlockSize;          //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    u32 StreamType;         //数据流种类，vids(0X73646976):视频;auds(0X73647561):音频
    u32 Handler;            //指定流的处理者,对于音视频来说即解码器,如 MJPG/H264 等.
    u32 Flags;              //标记：是否允许这个流输出？调色板是否变化？
    u16 Priority;           //流的优先级（当有多个同类型的流时优先级最高的为默认流）
    u16 Language;           //音频的语言代号
    u32 InitFrames;         //为交互格式指定初始帧数
    u32 Scale;              //数据量, 视频每桢的大小或者音频的采样大小
    u32 Rate;               //Scale/Rate=每秒采样数
    u32 Start;              //数据流开始播放的位置，单位为 Scale
    u32 Length;             //数据流的数据量，单位为 Scale
    u32 RefBufSize;         //建议使用的缓冲区大小
```

```
    u32 Quality;            //解压缩质量参数，值越大，质量越好
    u32 SampleSize;         //音频的样本大小
    struct                  //视频帧所占的矩形
    {
        short Left;
        short Top;
        short Right;
        short Bottom;
    }Frame;
}STRH_HEADER;
```

这里面，对我们最有用的即 StreamType 和 Handler 这两个参数了，StreamType 用于告诉我们此 strl 描述的是音频流（“auds”），还是视频流（“vids”）。而 Handler 则告诉我们所使用的解码器，比如 MJPG/H264 等（实际以 strf 块为准）。

然后是 strf 子块，不过 strf 字块，需要根据 strh 字块的类型而定。

如果 strh 子块是视频数据流（StreamType=“vids”），则 strf 子块的内容定义如下：

```
//BMP 结构体
typedef struct
{
    u32   BmpSize;          //bmp 结构体大小,包含(BmpSize 在内)
    long Width;             //图像宽
    long Height;            //图像高
    u16   Planes;           //平面数，必须为 1
    u16   BitCount;         //像素位数,0X0018 表示 24 位
    u32   Compression;      //压缩类型，比如:MJPG/H264 等
    u32   SizeImage;        //图像大小
    long XpixPerMeter;      //水平分辨率
    long YpixPerMeter;      //垂直分辨率
    u32   ClrUsed;          //实际使用了调色板中的颜色数,压缩格式中不使用
    u32   ClrImportant;     //重要的颜色
}BMP_HEADER;
//颜色表
typedef struct
{
    u8   rgbBlue;           //蓝色的亮度(值范围为 0-255)
    u8   rgbGreen;          //绿色的亮度(值范围为 0-255)
    u8   rgbRed;            //红色的亮度(值范围为 0-255)
    u8   rgbReserved;       //保留，必须为 0
}AVIRGBQUAD;
//对于 strh,如果是视频流,strf(流格式)使 STRF_BMPHEADER 块
typedef struct
{
    u32 BlockID;            //块标志,strf==0X73747266
    u32 BlockSize;          //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
```

```
    BMP_HEADER bmiHeader;       //位图信息头
    AVIRGBQUAD bmColors[1];     //颜色表
}STRF_BMPHEADER;
```

这里有 3 个结构体，strf 子块完整内容即：STRF_BMPHEADER 结构体，不过对我们有用的信息，都存放在 BMP_HEADER 结构体里面，本结构体对视频数据的解码起决定性的作用，它告诉我们视频的分辨率（Width 和 Height），以及视频所用的编码器（Compression），因此它决定了视频的解码。本章例程仅支持解码视频分辨率小于屏幕分辨率，且编解码器必须是 MJPG 的视频格式。

如果 strh 子块是音频数据流（StreamType=“auds”），则 strf 子块的内容定义如下：

```
//对于 strh,如果是音频流,strf(流格式)使 STRF_WAVHEADER 块
typedef struct
{
    u32 BlockID;            //块标志,strf==0X73747266
    u32 BlockSize;          //块大小(不包含最初的 8 字节,即 BlockID 和 BlockSize 不算)
    u16 FormatTag;          //格式标志:0X0001=PCM,0X0055=MP3...
    u16 Channels;           //声道数,一般为 2,表示立体声
    u32 SampleRate;         //音频采样率
    u32 BaudRate;           //波特率
    u16 BlockAlign;         //数据块对齐标志
    u16 Size;               //该结构大小
}STRF_WAVHEADER;
```

本结构体对音频数据解码起决定性的作用，他告诉我们音频信号的编码方式(FormatTag)、声道数（Channels）和采样率（SampleRate）等重要信息。本章例程仅支持 PCM 格式（FormatTag=0X0001）的音频数据解码。

2，数据块（MovieList）

信息块，即 ID 为“movi”的 LIST 块，它包含 AVI 的音视频序列数据，是这个 AVI 文件的主体部分。音视频数据块交错的嵌入在“movi”LIST 块里面，通过标准类型码进行区分，标准类型码有如下 4 种：

1，“##db”（非压缩视频帧）、
2，“##dc”（压缩视频帧）、
3，“##pc”（改用新的调色板）、
4，“##wb”（音频帧）。

其中##是编号，得根据我们的数据流顺序来确定，也就是前面的 strl 块。比如，如果第一个 strl 块是视频数据，那么对于压缩的视频帧，标准类型码就是：00dc。第二个 strl 块是音频数据，那么对于音频帧，标准类型码就是：01wb。

紧跟着标准类型码的是 4 个字节的数据长度（不包含类型码和长度参数本身，也就是总长度必须要加 8 才对），该长度必须是偶数，如果读到为奇数，则加 1 即可。我们读数据的时候，一般一次性要读完一个标准类型码所表征的数据，方便解码。

3，索引块（Index Chunk）

最后，紧跟在‘hdrl’列表和‘movi’列表之后的，就是 AVI 文件可选的索引块。这个索引块为 AVI 文件中每一个媒体数据块进行索引，并且记录它们在文件中的偏移（可能相对于

‘movi’列表，也可能相对于 AVI 文件开头）。本章我们用不到索引块，这里就不详细介绍了。

关于 AVI 文件，我们就介绍到这，有兴趣的朋友，可以再看看光盘：6，软件资料→AVI 学习资料 里面的相关文档。

最后，我们看看要实现 avi 视频文件的播放，主要有哪些步骤，如下：

1）初始化各外设

要解码视频，相关外设肯定要先初始化好，比如：SDMMC（驱动 SD 卡用）、I2S、DMA、WM8978、LCD 和按键等。这些具体初始化过程，在前面的例程都有介绍，大同小异，这里就不再细说了。

2）读取 AVI 文件，并解析

要解码，得先读取 avi 文件，按 55.1.1 节的介绍，读取出音视频关键信息，音频参数：编码方式、采样率、位数和音频流类型码(01wb/00wb)等；视频参数：编码方式、帧间隔、图片尺寸和视频流类型码(00dc/01dc)等；共同的：数据流起始地址。有了这些参数，我们便可以初始化音视频解码，为后续解码做好准备。

3）根据解析结果，设置相关参数

根据第 2 步解析的结果，设置 I2S 的音频采样率和位数，同时要让视频显示在 LCD 中间区域，得根据图片尺寸，设置 LCD 开窗时 x，y 方向的偏移量。

4）读取数据流，开始解码

前面三步完成，就可以正式开始播放视频了。读取视频流数据（movi 块），根据类型码，执行音频/视频解码。对于音频数据(01wb/00wb)，本例程只支持未压缩的 PCM 数据，所以，直接填充到DMA 缓冲区即可，由 DMA 循环发送给 WM8978，播放音频。对于视频数据(00dc/01dc)，本例程只支持 MJPG，通过硬件 JPEG 解码，硬件 JPEG 解码流程详见第五十章。然后，利用定时器来控制帧间隔，以正常速度播放视频，从而实现音视频解码。

5）解码完成，释放资源

最后在文件读取完后（或者出错了），需要释放申请的内存、恢复 LCD 窗口、关闭定时器、停止 I2S 播放音乐和关闭文件等一系列操作，等待下一次解码。

55.2 硬件设计

本章实验功能简介：开机后，先初始化各外设，然后检测字库是否存在，如果检测无问题，则开始播放 SD 卡 VIDEO 文件夹里面的视频（.avi 格式）。注意：1，在 SD 卡根目录必须建立一个 VIDEO 文件夹，并存放 AVI 视频（仅支持 MJPG 视频，音频必须是 PCM，且视频分辨率必须小于等于屏幕分辨率）在里面。2，我们所需要的视频，可以通过：狸窝全能视频转换器，转换后得到，具体步骤后续会讲到（55.4 节）。

视频播放时，LCD 上还会显示视频名字、当前视频编号、总视频数、声道数、音频采样率、帧率、播放时间和总时间等信息。KEY0 用于选择下一个视频，KEY2 用于选择上一个视频，KEY_UP 可以快进，KEY1 可以快退。DS0 还是用于指示程序运行状态（仅字库错误时）。

本实验用到的资源如下：

1） 指示灯 DS0
2） 4 个按键（KEY_UP/KEY0/KEY1/KEY2）
3） 串口
4） LCD 模块
5） SD 卡
6） SPI FLASH
7） WM8978

8） SAI

9） 硬件 JPEG 解码器

这些前面都已介绍过。本实验，大家需要准备 1 个 SD 卡和一个耳机，分别插入 SD 卡接口和耳机接口（PHONE），然后下载本实验就可以看视频了！

55.3 软件设计

本实验，我们音乐播放器实验（第五十二章）的基础上进行修改。本章要用到硬件 JPEG 解码和定时器，所以添加 jpegcodec.c、jpeg_utils.c 和 timer.c。

之后，在工程目录新建 MJPEG 文件夹，在该文件夹里面新建 JPEG 文件夹，新建 avi.c、avi.h、mjpeg.c 和 mjpeg.h 四个文件。然后，工程里面，新建 MJPEG 分组，将 avi.c 和 mjpeg.c 添加到该分组下面，并将 MJPEG 文件夹加入头文件包含路径。

最后，在 APP 文件夹下面新建 videoplayer.c 和 videoplayer.h 两个文件，然后将 videoplayer.c 加入到工程的 APP 组下。本例程代码比较多，这里我们只介绍一些重要的函数。详细代码，请大家参考本例程源码。

首先是 avi.c 里面的几个函数，代码如下：

```
AVI_INFO avix;                          //avi 文件相关信息
u8*const AVI_VIDS_FLAG_TBL[2]={"00dc","01dc"};//视频编码标志字符串,00dc/01dc
u8*const AVI_AUDS_FLAG_TBL[2]={"00wb","01wb"};//音频编码标志字符串,00wb/01wb

//avi 解码初始化
//buf:输入缓冲区
//size:缓冲区大小
//返回值:AVI_OK,avi 文件解析成功
//              其他,错误代码
AVISTATUS avi_init(u8 *buf,u32 size)
{
    u16 offset;
    u8 *tbuf;
    AVISTATUS res=AVI_OK;
    AVI_HEADER *aviheader;
    LIST_HEADER *listheader;
    AVIH_HEADER *avihheader;
    STRH_HEADER *strhheader;

    STRF_BMPHEADER *bmpheader;
    STRF_WAVHEADER *wavheader;

    tbuf=buf;
    aviheader=(AVI_HEADER*)buf;
    if(aviheader->RiffID!=AVI_RIFF_ID)return AVI_RIFF_ERR;      //RIFF ID 错误
    if(aviheader->AviID!=AVI_AVI_ID)return AVI_AVI_ERR;         //AVI ID 错误
    buf+=sizeof(AVI_HEADER);                                    //偏移
    listheader=(LIST_HEADER*)(buf);
```

```
if(listheader->ListID!=AVI_LIST_ID)return AVI_LIST_ERR;          //LIST ID 错误
if(listheader->ListType!=AVI_HDRL_ID)return AVI_HDRL_ERR;        //HDRL ID 错误
buf+=sizeof(LIST_HEADER);                                        //偏移
avihheader=(AVIH_HEADER*)(buf);
if(avihheader->BlockID!=AVI_AVIH_ID)return AVI_AVIH_ERR;         //AVIH ID 错误
avix.SecPerFrame=avihheader->SecPerFrame;                        //得到帧间隔时间
avix.TotalFrame=avihheader->TotalFrame;                          //得到总帧数
buf+=avihheader->BlockSize+8;                                    //偏移
listheader=(LIST_HEADER*)(buf);
if(listheader->ListID!=AVI_LIST_ID)return AVI_LIST_ERR;          //LIST ID 错误
if(listheader->ListType!=AVI_STRL_ID)return AVI_STRL_ERR;        //STRL ID 错误
strhheader=(STRH_HEADER*)(buf+12);
if(strhheader->BlockID!=AVI_STRH_ID)return AVI_STRH_ERR;         //STRH ID 错误
if(strhheader->StreamType==AVI_VIDS_STREAM)                      //视频帧在前
{
    if(strhheader->Handler!=AVI_FORMAT_MJPG)return AVI_FORMAT_ERR;
                                                                 //非 MJPG 视频流,不支持
    avix.VideoFLAG=(u8*)AVI_VIDS_FLAG_TBL[0];        //视频流标记   "00dc"
    avix.AudioFLAG=(u8*)AVI_AUDS_FLAG_TBL[1];      //音频流标记   "01wb"
    bmpheader=(STRF_BMPHEADER*)(buf+12+strhheader->BlockSize+8);//strf
    if(bmpheader->BlockID!=AVI_STRF_ID)return AVI_STRF_ERR;//STRF ID 错误
    avix.Width=bmpheader->bmiHeader.Width;
    avix.Height=bmpheader->bmiHeader.Height;
    buf+=listheader->BlockSize+8;    //偏移
    listheader=(LIST_HEADER*)(buf);
    if(listheader->ListID!=AVI_LIST_ID)//是不含有音频帧的视频文件
    {
        avix.SampleRate=0;           //音频采样率
        avix.Channels=0;             //音频通道数
        avix.AudioType=0;            //音频格式

    }else
    {
        if(listheader->ListType!=AVI_STRL_ID)return AVI_STRL_ERR;//STRL ID 错误
        strhheader=(STRH_HEADER*)(buf+12);
        if(strhheader->BlockID!=AVI_STRH_ID)
                return AVI_STRH_ERR;//STRH ID 错误
        if(strhheader->StreamType!=AVI_AUDS_STREAM)
                return AVI_FORMAT_ERR;//格式错误
        wavheader=(STRF_WAVHEADER*)(buf+12+strhheader->BlockSize+8);//strf
        if(wavheader->BlockID!=AVI_STRF_ID)
                return AVI_STRF_ERR;//STRF ID 错误
        avix.SampleRate=wavheader->SampleRate; //音频采样率
```

```
            avix.Channels=wavheader->Channels;  //音频通道数
            avix.AudioType=wavheader->FormatTag;    //音频格式
        }
    }else if(strhheader->StreamType==AVI_AUDS_STREAM)//音频帧在前
    {
        avix.VideoFLAG=(u8*)AVI_VIDS_FLAG_TBL[1];   //视频流标记    "01dc"
        avix.AudioFLAG=(u8*)AVI_AUDS_FLAG_TBL[0]; //音频流标记    "00wb"
        wavheader=(STRF_WAVHEADER*)(buf+12+strhheader->BlockSize+8);//strf
        if(wavheader->BlockID!=AVI_STRF_ID)return AVI_STRF_ERR;    //STRF ID 错误
        avix.SampleRate=wavheader->SampleRate;      //音频采样率
        avix.Channels=wavheader->Channels;          //音频通道数
        avix.AudioType=wavheader->FormatTag;        //音频格式
        buf+=listheader->BlockSize+8;               //偏移
        listheader=(LIST_HEADER*)(buf);
        if(listheader->ListID!=AVI_LIST_ID)return AVI_LIST_ERR;    //LIST ID 错误
        if(listheader->ListType!=AVI_STRL_ID)return AVI_STRL_ERR;//STRL ID 错误
        strhheader=(STRH_HEADER*)(buf+12);
        if(strhheader->BlockID!=AVI_STRH_ID)
                                    return AVI_STRH_ERR;//STRH ID 错误
        if(strhheader->StreamType!=AVI_VIDS_STREAM)
                                    return AVI_FORMAT_ERR;//格式错误
        bmpheader=(STRF_BMPHEADER*)(buf+12+strhheader->BlockSize+8);//strf
        if(bmpheader->BlockID!=AVI_STRF_ID)return AVI_STRF_ERR;    //STRF ID 错误
        if(bmpheader->bmiHeader.Compression!=AVI_FORMAT_MJPG)
                                    return AVI_FORMAT_ERR;//格式错误
        avix.Width=bmpheader->bmiHeader.Width;
        avix.Height=bmpheader->bmiHeader.Height;
    }
    offset=avi_srarch_id(tbuf,size,"movi");     //查找 movi ID
    if(offset==0)return AVI_MOVI_ERR;           //MOVI ID 错误
    if(avix.SampleRate)//有音频流,才查找
    {
        tbuf+=offset;
        offset=avi_srarch_id(tbuf,size,avix.AudioFLAG);     //查找音频流标记
        if(offset==0)return AVI_STREAM_ERR;                 //流错误
        tbuf+=offset+4;
        avix.AudioBufSize=*((u16*)tbuf);                    //得到音频流 buf 大小.
    }
    return res;
}

//查找 ID
//buf:待查缓存区
```

```
//size:缓存大小
//id:要查找的 id,必须是 4 字节长度
//返回值:0,查找失败,其他:movi ID 偏移量
u16 avi_srarch_id(u8* buf,u32 size,u8 *id)
{
    u16 i;
    size-=4;
    for(i=0;i<size;i++)
    {
        if(buf[i]==id[0])
            if(buf[i+1]==id[1])
                if(buf[i+2]==id[2])
                    if(buf[i+3]==id[3])return i;//找到"id"所在的位置
    }
    return 0;
}
//得到 stream 流信息
//buf:流开始地址(必须是 01wb/00wb/01dc/00dc 开头)
AVISTATUS avi_get_streaminfo(u8* buf)
{
    avix.StreamID=MAKEWORD(buf+2);          //得到流类型
    avix.StreamSize=MAKEDWORD(buf+4);       //得到流大小
    if(avix.StreamSize%2)avix.StreamSize++;  //奇数加 1(avix.StreamSize,必须是偶数)
    if(avix.StreamID==AVI_VIDS_FLAG||avix.StreamID==AVI_AUDS_FLAG)
                                            return AVI_OK;
    return AVI_STREAM_ERR;
}
```

这里三个函数，其中 avi_ini 用于解析 AVI 文件，获取音视频流数据的详细信息，为后续解码做准备。而 avi_srarch_id 用于查找某个 ID，可以是 4 个字节长度的 ID，比如 00dc，01wb，movi 之类的，在解析数据以及快进快退的时候，有用到。avi_get_streaminfo 函数，则是用来获取当前数据流信息，重点是取得流类型和流大小，方便解码和读取下一个数据流。

接下来，我们看 mjpeg.c 里面的几个函数，代码如下：

```
//mjpeg  解码初始化
//offx,offy:x,y 方向的偏移
//返回值:0,成功;
//       1,失败
u8 mjpeg_init(u16 offx,u16 offy,u32 width,u32 height)
{
    u8 res;
    res=JPEG_Core_Init(&mjpeg);                         //初始化 JPEG 内核
    if(res)return 1;
    rgb565buf=mymalloc(SRAMEX,width*height*2);      //申请 RGB 缓存
    if(rgb565buf==NULL)return   2;
```

```
    imgoffx=offx;
    imgoffy=offy;
    mjpeg_rgb_framebuf=(u16*)ltdc_framebuf[lcdltdc.activelayer];//指向 RGBLCD 当前显存
    return 0;
}
//mjpeg 结束,释放内存
void mjpegdec_free(void)
{
    JPEG_Core_Destroy(&mjpeg);
    myfree(SRAMEX,rgb565buf);
}
//填充颜色
//x,y:起始坐标
//width，height：宽度和高度。
//*color：颜色数组
void mjpeg_fill_color(u16 x,u16 y,u16 width,u16 height,u16 *color)
{
    u16 i,j;
    u32 param1;
    u32 param2;
    u32 param3;
    u16* pdata;
    if(lcdltdc.pwidth!=0&&lcddev.dir==0)//如果是 RGB 屏,且竖屏,则填充函数不可直接用
    {
        param1=lcdltdc.pixsize*lcdltdc.pwidth*(lcdltdc.pheight-x-1)+lcdltdc.pixsize*y;
                                                    //将运算先做完,提高速度
        param2=lcdltdc.pixsize*lcdltdc.pwidth;
        for(i=0;i<height;i++)
        {
            param3=i*lcdltdc.pixsize+param1;
            pdata=color+i*width;
            for(j=0;j<width;j++)
            {
                *(u16*)((u32)mjpeg_rgb_framebuf+param3-param2*j)=pdata[j];
            }
        }
    }else LCD_Color_Fill(x,y,x+width-1,y+height-1,color);//其他情况,直接填充
}
//解码一副 JPEG 图片
//buf:jpeg 数据流数组
//bsize:数组大小
//返回值:0,成功
//      其他,错误
```

```
u8 mjpegdec_decode(u8* buf,u32 bsize)
{
    vu32 timecnt=0;
    u8 fileover=0;
    u8 i=0;
    u32 mcublkindex=0;
    if(bsize==0)return 0;
    JPEG_Decode_Init(&mjpeg);                       //初始化硬件 JPEG 解码器
    for(i=0;i<JPEG_DMA_INBUF_NB;i++)
    {
        if(bsize>JPEG_DMA_INBUF_LEN)
        {
            mymemcpy(mjpeg.inbuf[i].buf,buf,JPEG_DMA_INBUF_LEN);
            mjpeg.inbuf[i].size=JPEG_DMA_INBUF_LEN; //读取了的数据长度
            mjpeg.inbuf[i].sta=1;                   //标记 buf 满
            buf+=JPEG_DMA_INBUF_LEN;                //源数组往后偏移
            bsize-=JPEG_DMA_INBUF_LEN;              //文件大小减少
        }else
        {
            mymemcpy(mjpeg.inbuf[i].buf,buf,bsize);
            mjpeg.inbuf[i].size=bsize;              //读取了的数据长度
            mjpeg.inbuf[i].sta=1;                   //标记 buf 满
            buf+=bsize;                             //源数组往后偏移
            bsize=0;                                //文件大小为 0 了.
            break;
        }
    }
    JPEG_IN_OUT_DMA_Init((u32)mjpeg.inbuf[0].buf,(u32)mjpeg.outbuf[0].buf,
                         mjpeg.inbuf[0].size,JPEG_DMA_OUTBUF_LEN);//配置 DMA
    jpeg_in_callback=mjpeg_dma_in_callback;         //JPEG DMA 读取数据回调函数
    jpeg_out_callback=mjpeg_dma_out_callback;       //JPEG DMA 输出数据回调函数
    jpeg_eoc_callback=mjpeg_endofcovert_callback;//JPEG  解码结束回调函数
    jpeg_hdp_callback=mjpeg_hdrover_callback;       //JPEG Header 解码完成回调函数
    JPEG_DMA_Start();                               //启动 DMA 传输
    while(1)
    {
        SCB_CleanInvalidateDCache();                //清空 D catch
        if(mjpeg.inbuf[mjpeg.inbuf_write_ptr].sta==0&&fileover==0)  //有 buf 为空
        {
            if(bsize>JPEG_DMA_INBUF_LEN)
            {
                mymemcpy(mjpeg.inbuf[mjpeg.inbuf_write_ptr].buf,buf,
                                                JPEG_DMA_INBUF_LEN);
```

```
                mjpeg.inbuf[mjpeg.inbuf_write_ptr].size=JPEG_DMA_INBUF_LEN;
                                                        //读取了的数据长度
                mjpeg.inbuf[mjpeg.inbuf_write_ptr].sta=1;  //标记 buf 满
                buf+=JPEG_DMA_INBUF_LEN;        //源数组往后偏移
                bsize-=JPEG_DMA_INBUF_LEN;      //文件大小减少
            }else
            {
                mymemcpy(mjpeg.inbuf[mjpeg.inbuf_write_ptr].buf,buf,bsize);
                mjpeg.inbuf[mjpeg.inbuf_write_ptr].size=bsize;  //读取了的数据长度
                mjpeg.inbuf[mjpeg.inbuf_write_ptr].sta=1;       //标记 buf 满
                buf+=bsize;                                     //源数组往后偏移
                bsize=0;                                        //文件大小为 0 了.
                timecnt=0;      //清零计时器
                fileover=1;     //文件结束了...
            }
            if(mjpeg.indma_pause==1&&mjpeg.inbuf[mjpeg.inbuf_read_ptr].sta==1)
                                                    //之前是暂停的了,继续传输
            {
                JPEG_IN_DMA_Resume((u32)mjpeg.inbuf[mjpeg.inbuf_read_ptr].buf,
                    mjpeg.inbuf[mjpeg.inbuf_read_ptr].size);    //继续下一次 DMA 传输
                mjpeg.indma_pause=0;
            }
            mjpeg.inbuf_write_ptr++;
            if(mjpeg.inbuf_write_ptr>=JPEG_DMA_INBUF_NB)mjpeg.inbuf_write_ptr=0;
        }
        if(mjpeg.outbuf[mjpeg.outbuf_read_ptr].sta==1)  //buf 里面有数据要处理
        {
            mcublkindex+=mjpeg.ycbcr2rgb(mjpeg.outbuf[mjpeg.outbuf_read_ptr].buf,
(u8*)rgb565buf,mcublkindex,mjpeg.outbuf[mjpeg.outbuf_read_ptr].size);//YUV --> RGB565
            mjpeg.outbuf[mjpeg.outbuf_read_ptr].sta=0;//标记 buf 为空
            mjpeg.outbuf[mjpeg.outbuf_read_ptr].size=0;     //数据量清空
            mjpeg.outbuf_read_ptr++;
            if(mjpeg.outbuf_read_ptr>=JPEG_DMA_OUTBUF_NB)
                mjpeg.outbuf_read_ptr=0;//限制范围
            if(mcublkindex==mjpeg.total_blks)
            {
                break;
            }
        }else if(mjpeg.outdma_pause==1&&mjpeg.outbuf[mjpeg.outbuf_write_ptr].sta==0)
                                //out 暂停,且当前 writebuf 已经为空了,则恢复 out 输出
        {
            JPEG_OUT_DMA_Resume((u32)mjpeg.outbuf[mjpeg.outbuf_write_ptr].buf,
                                JPEG_DMA_OUTBUF_LEN);//继续下一次 DMA 传输
```

```
                mjpeg.outdma_pause=0;
            }
            timecnt++;
            if(fileover)//文件结束后,及时退出,防止死循环
            {
                if(mjpeg.state==JPEG_STATE_NOHEADER)break;  //解码失败了
                if(timecnt>0X3FFF)break;                    //超时退出
            }
        }
        if(mjpeg.state==JPEG_STATE_FINISHED) //解码完成了
        {
            mjpeg_fill_color(imgoffx,imgoffy,mjpeg.Conf.ImageWidth,
                                        mjpeg.Conf.ImageHeight,rgb565buf);
        }
        return 0;
    }
```

其中，mjpeg_init 函数，用于初始化 jpeg 解码 ，调用 JPEG_Core_Init 函数，对硬件 JPEG 解码内核进行初始化，然后申请内存，确定视频在液晶上面的偏移（让视频显示在 LCD 中央）。

mjpeg_free 函数，用于释放内存，解码结束后调用。

mjpeg_fill_color 函数，用于解码完成后，将 RGB565 数据填充到液晶屏上，对于 RGB 屏的竖屏模式，不能用 DMA2D 填充，只能打点的方式填充，通过计算参量，提高打点速度。对于 MCU 屏和 RGB 横屏，则直接调用 LCD_Color_Fill 函数进行填充即可。

mjpeg_decode 函数，是解码 jpeg 的主要函数，解码步骤参见第五十章相关内容。解码后将 YUV 转换成 RGB565 数据，存放在 rgb565buf 里面，然后通过 mjpeg_fill_color 函数，将 RGB565 数据显示到 LCD 屏幕上 。

接下来，我们看 videoplayer.c 里面 video_play_mjpeg 函数，代码如下：

```
//播放一个 mjpeg 文件
//pname:文件名
//返回值:
//KEY0_PRES:下一曲
//KEY1_PRES:上一曲
//其他:错误
u8 video_play_mjpeg(u8 *pname)
{
    u8* framebuf; //视频解码 buf
    u8* pbuf;     //buf 指针
    FIL *favi;
    u8  res=0;
    u16 offset=0;
    u32 nr;
    u8 key;
    u8 saisavebuf;
    saibuf[0]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
```

```
saibuf[1]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
saibuf[2]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
saibuf[3]=mymalloc(SRAMIN,AVI_AUDIO_BUF_SIZE); //申请音频内存
framebuf=mymalloc(SRAMIN,AVI_VIDEO_BUF_SIZE); //申请视频 buf
favi=(FIL*)mymalloc(SRAMIN,sizeof(FIL));                    //申请 favi 内存
memset(saibuf[0],0,AVI_AUDIO_BUF_SIZE);
memset(saibuf[1],0,AVI_AUDIO_BUF_SIZE);
memset(saibuf[2],0,AVI_AUDIO_BUF_SIZE);
memset(saibuf[3],0,AVI_AUDIO_BUF_SIZE);
if(!saibuf[3]||!framebuf||!favi)
{
    printf("memory error!\r\n");
    res=0XFF;
}
while(res==0)
{
    res=f_open(favi,(char *)pname,FA_READ);
    if(res==0)
    {
        pbuf=framebuf;
        res=f_read(favi,pbuf,AVI_VIDEO_BUF_SIZE,&nr);//开始读取
        if(res)
        {
            printf("fread error:%d\r\n",res);
            break;
        }
        //开始 avi 解析
        res=avi_init(pbuf,AVI_VIDEO_BUF_SIZE);      //avi 解析
        if(res)
        {
            printf("avi err:%d\r\n",res);
            break;
        }
        video_info_show(&avix);
        TIM6_Init(avix.SecPerFrame/100-1,10800-1);//10Khz 计数频率,加 1 是 100us
        offset=avi_srarch_id(pbuf,AVI_VIDEO_BUF_SIZE,"movi");//寻找 movi ID
        avi_get_streaminfo(pbuf+offset+4);                  //获取流信息
        f_lseek(favi,offset+12);              //跳过标志 ID,读地址偏移到流数据开始处

    res=mjpeg_init((lcddev.width-avix.Width)/2,110+(lcddev.height-110-avix.Height)/2,
                   avix.Width,avix.Height);//JPG 解码初始化
        if(avix.SampleRate)                             //有音频信息,才初始化
        {
```

```
            WM8978_I2S_Cfg(2,0);//飞利浦标准,16 位数据长度

        SAIA_Init(SAI_MODEMASTER_TX,SAI_CLOCKSTROBING_RISINGEDGE,
            SAI_DATASIZE_16);//设置 SAI,主发送,16 位数据
            SAIA_SampleRate_Set(avix.SampleRate);  //设置采样率
            SAIA_TX_DMA_Init(saibuf[1],saibuf[2],avix.AudioBufSize/2,1);
                                                        //配置 DMA
            sai_tx_callback=audio_sai_dma_callback;
                                    //回调函数指向 SAI_DMA_Callback
            saiplaybuf=0;
            saisavebuf=0;
            SAI_Play_Start(); //开启 sai 播放
        }
        while(1)//播放循环
        {
            if(avix.StreamID==AVI_VIDS_FLAG)//视频流
            {
                pbuf=framebuf;
                f_read(favi,pbuf,avix.StreamSize+8,&nr);
                            //读入整帧+下一数据流 ID 信息
                res=mjpeg_decode(pbuf,avix.StreamSize);
                if(res)
                {
                    printf("decode error!\r\n");
                }
                while(frameup==0);//等待时间到达(在 TIM6 的中断里面设置为 1)
                frameup=0;              //标志清零
                frame++;
            }else       //音频流
            {
                video_time_show(favi,&avix);      //显示当前播放时间
                saisavebuf++;
                if(saisavebuf>3)saisavebuf=0;
                do
                {
                    nr=saiplaybuf;
                    if(nr)nr--;
                    else nr=3;
                }while(saisavebuf==nr);//碰撞等待.
                f_read(favi,saibuf[saisavebuf],avix.StreamSize+8,&nr);//填充 saibuf
                pbuf=saibuf[saisavebuf];
            }
            key=KEY_Scan(0);
```

```
                    if(key==KEY0_PRES||key==KEY2_PRES)
                                        //KEY0/KEY2 按下,播放下一个/上一个视频
                    {
                        res=key;
                        break;
                    }else if(key==KEY1_PRES||key==WKUP_PRES)
                    {
                        SAI_Play_Stop();//关闭音频
                        video_seek(favi,&avix,framebuf);
                        pbuf=framebuf;
                        SAI_Play_Start();//开启 DMA 播放
                    }
                    if(avi_get_streaminfo(pbuf+avix.StreamSize))//读取下一帧  流标志
                    {
                        printf("frame error \r\n");
                        res=KEY0_PRES;
                        break;
                    }
                }
                SAI_Play_Stop();  //关闭音频
                TIM6->CR1&=~(1<<0); //关闭定时器 6
                LCD_Set_Window(0,0,lcddev.width,lcddev.height);//恢复窗口
                mjpeg_free();       //释放内存
                f_close(favi);
            }
        }
        myfree(SRAMIN,saibuf[0]);
        myfree(SRAMIN,saibuf[1]);
        myfree(SRAMIN,saibuf[2]);
        myfree(SRAMIN,saibuf[3]);
        myfree(SRAMIN,framebuf);
        myfree(SRAMIN,favi);
        return res;
    }
```

该函数用来播放一个 avi 视频文件（mjpg 编码），解码过程就是根据前面我们在 55.1.2 节最后所介绍的步骤进行，不过在这里，我们的音频播放用了 4 个 buf，以提高解码的流畅度。其他代码，我们就不再介绍了，请大家参考本例程源码。

最后，看看主函数：

```
int main(void)
{

    Cache_Enable();                 //打开 L1-Cache
    MPU_Memory_Protection();        //保护相关存储区域
```

```
HAL_Init();                         //初始化 HAL 库
Stm32_Clock_Init(432,25,2,9);       //设置时钟,216Mhz
delay_init(216);                    //延时初始化
uart_init(115200);                  //串口初始化
TIM3_Init(10000-1,10800-1);         //10Khz 计数,1 秒钟中断一次
LED_Init();                         //初始化 LED
KEY_Init();                         //初始化按键
SDRAM_Init();                       //初始化 SDRAM
LCD_Init();                         //初始化 LCD
W25QXX_Init();                      //初始化 W25Q256
WM8978_Init();                      //初始化 WM8978
WM8978_ADDA_Cfg(1,0);               //开启 DAC
WM8978_Input_Cfg(0,0,0);            //关闭输入通道
WM8978_Output_Cfg(1,0);             //开启 DAC 输出
WM8978_HPvol_Set(40,40);            //耳机音量设置
WM8978_SPKvol_Set(50);              //喇叭音量设置
my_mem_init(SRAMIN);                //初始化内部内存池
my_mem_init(SRAMEX);                //初始化外部 SDRAM 内存池
my_mem_init(SRAMDTCM);              //初始化内部 CCM 内存池
exfuns_init();                      //为 fatfs 相关变量申请内存
f_mount(fs[0],"0:",1);              //挂载 SD 卡
f_mount(fs[1],"1:",1);              //挂载 SPI FLASH.
f_mount(fs[2],"2:",1);              //挂载 NAND 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);
}
POINT_COLOR=RED;
Show_Str(60,50,200,16,"阿波罗 STM32F4/F7 开发板",16,0);
Show_Str(60,70,200,16,"视频播放器实验",16,0);
Show_Str(60,90,200,16,"正点原子@ALIENTEK",16,0);
Show_Str(60,110,200,16,"2016 年 7 月 18 日",16,0);
Show_Str(60,130,200,16,"KEY0:NEXT     KEY2:PREV",16,0);
Show_Str(60,150,200,16,"KEY_UP:FF     KEY1：REW",16,0);
delay_ms(1500);
while(1)
{
    video_play();
}
```

```
}
```

该函数代码比较简单，我们就不多说了。最后，为了提高速度，我们对编译器进行设置，选择使用-O2 优化，从而优化代码，提高速度（但调试效果不好，建议调试时设置为-O0），编译器设置如图 55.3.1 所示：

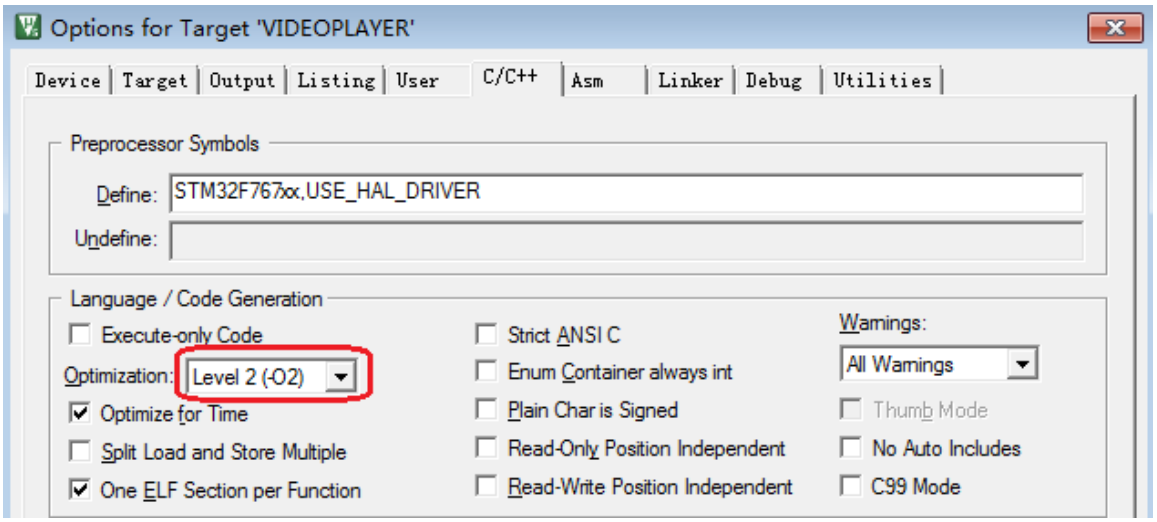


图 52.3.2 编译器优化设置

设置完后，重新编译即可。至此，本实验的软件设计部分结束。

55.4 下载验证

本章，我们例程仅支持 MJPG 编码的 avi 格式视频，且音频必须是 PCM 格式，另外视频分辨率不能大于 LCD 分辨率。要满足这些要求，现成的 avi 文件是很难找到的，所以我们需要用软件，将通用视频（任何视频都可以）转换为我们需要的格式，这里我们通过：狸窝全能视频转换器，这款软件来实现（路径：光盘：6，软件资料→软件→视频转换软件→狸窝全能视频转换器.exe）。安装完后，打开，然后进行相关设置，软件设置如图 55.4.1 和 55.4.2 所示：

图 55.4.1 软件启动界面和设置

图 55.4.2 高级设置

首先，如图 55.4.1 所示，点击 1 处，添加视频，找到你要转换的视频，添加进来。有的视频可能有独立字幕，比如我们打开的这个视频就有，所以在 2 处选择下字幕（如果没有的，可以忽略此步）。然后在 3 处，点击▼图标，选择预制方案：AVI-Audio-Video Interleaved(*.avi)，即生成.avi 文件，然后点击 4 处的高级设置按钮，进入 55.4.2 所示的界面，设置详细参数如下：

视频编码器：选择 MJPEG。本例程仅支持 MJPG 视频解码，所以选择这个编码器。

视频尺寸：480x272。这里得根据所用 LCD 分辨率来选择，假设我们用 480*800 的 4.3 寸电容屏模块，则这里最大可以设置：480x272。PS:如果是 2.8 屏，最大宽度只能是 240）。

比特率：1000。这里设置越大，视频质量越好，解码就越慢（可能会卡），我们设置为 1000，可以得到比较好的视频质量，同时也不怎么会卡。

帧率：10。即每秒钟 10 帧。对于 480*272 的视频，本例程最高能播放 30 帧左右的视频，如果要想提高帧率，有几个办法：1，降低分辨率；2，降低比特率；3，降低音频采样率。

音频编码器：PCMS16LE。本例程只支持 PCM 音频，所以选择音频编码器为这个。

采样率：这里设置为 11025，即 11.025Khz 的采样率。这里越高，声音质量越好，不过，转换后的文件就越大，而且视频可能会卡。

其他设置，采用默认的即可。设置完以后，点击确定，即可完成设置。

点击图 55.4.1 的 5 处的文件夹图标，设置转换后视频的输出路径，这里我们设置到了桌面，这样转换后的视频，会保存在桌面。最后，点击图中 6 处的按钮，即可开始转换了，如图 55.4.3 所示：

图 55.4.3 正在转换

等转换完成后，将转换后的.avi 文件，拷贝到 SD 卡→VIDEO 文件夹下，然后插入开发板的 SD 卡接口，就可以开始测试本章例程了。

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，程序先检测字库，然后检测 SD 卡的 VIDEO 文件夹，并查找 avi 视频文件，在找到有效视频文件后，便开始播放视频，如图 55.4.4 所示：

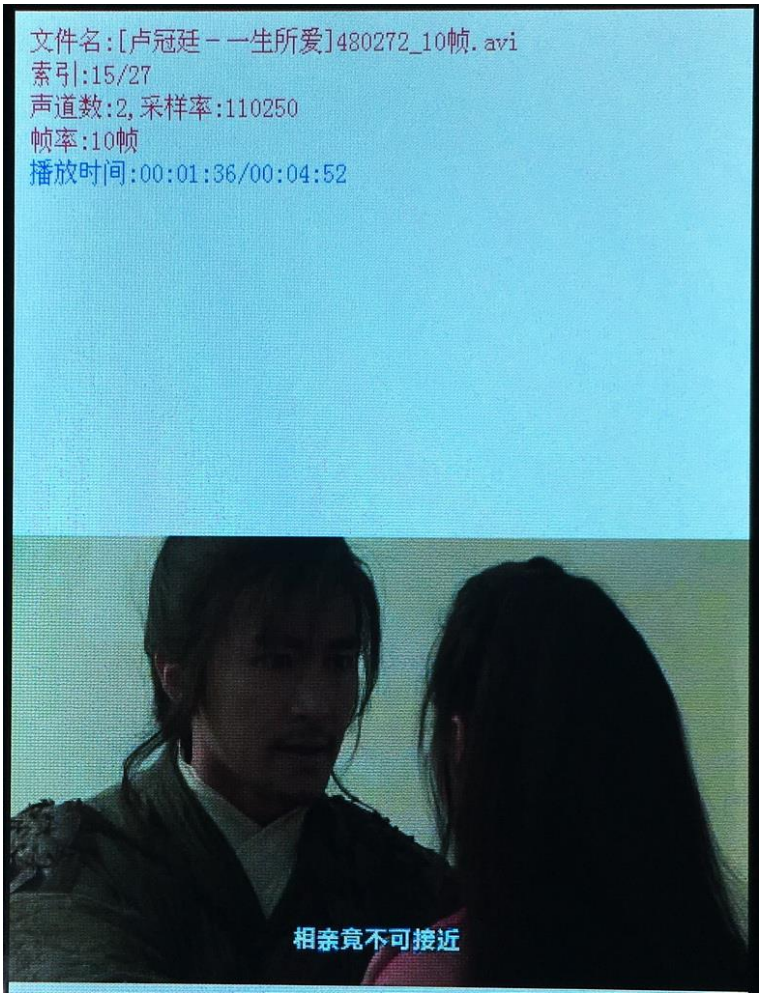


图 55.4.4 视频播放中

可以看到，屏幕显示了文件名、索引、声道数、采样率、帧率和播放时间等参数。然后，我们按 KEY0/KEY2，可以切换到下一个/上一个视频，按 KEY_UP/KEY1，可以快进/快退。

至此，本例程介绍就结束了。本实验，我们在阿波罗 STM32 开发板上实现了视频播放，体现了 STM32F767 强大的处理能力。

本例程只支持竖屏宽度的分辨率解码（比如 800*480 的屏，最大只支持 480 宽度的视频解码），如果想要支持更大分辨率的视频解码，则必须使用横屏模式，需要在本例程源码的基础上稍作修改（参见综合实验的视频播放器功能）。

附 STM32F767 硬件 JPEG 视频解码性能：

对 480*272 及以下分辨率，可达 30 帧

对 800*480 分辨率，可达 20 帧

对 1024*600 分辨率，可达 10 帧

最后提醒大家，转换的视频分辨率，一定要根据自己的 LCD 设置，不能超过 LCD 的尺寸!!否则无法播放（可能只听到声音，看不到图像）。

第五十六章 FPU 测试(Julia 分形)实验

本章，我们将向大家介绍如何开启 STM32F767 的硬件 FPU，并对比使用硬件 FPU 和不使用硬件 FPU 的速度差别，以体现硬件 FPU 的优势。本章分为如下几个部：

56.1 FPU&Julia 分形简介
56.2 硬件设计
56.3 软件设计
56.4 下载验证

56.1 FPU&Julia 分形简介

本节将分别介绍 STM32F767 的 FPU 和 Julia 分形。

56.1.1 FPU 简介

FPU 即浮点运算单元（Float Point Unit）。浮点运算，对于定点 CPU（没有 FPU 的 CPU）来说必须要按照 IEEE-754 标准的算法来完成运算，是相当耗费时间的。而对于有 FPU 的 CPU 来说，浮点运算则只是几条指令的事情，速度相当快。

STM32F767 属于 Cortex M7 架构，带有 32 位双精度硬件 FPU，支持浮点指令集，相对于 Cortex M0 和 Cortex M3 等，高出数十倍甚至上百倍的运算性能。

STM32F767 硬件上要开启 FPU 是很简单的，通过一个叫：协处理器控制寄存器（CPACR）的寄存器设置即可开启 STM32F767 的硬件 FPU，该寄存器各位描述如图 56.1.1.1 所示：

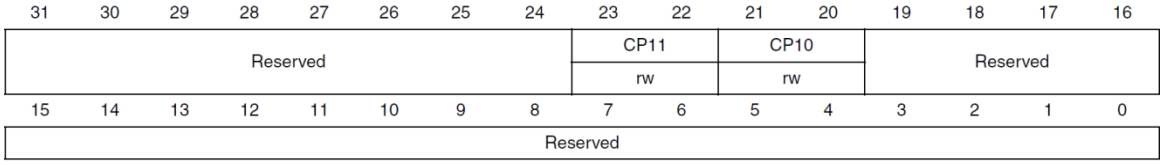


图 56.1.1.1 协处理器控制寄存器（CPACR）各位描述

这里我们就是要设置 CP11 和 CP10 这 4 个位，复位后，这 4 个位的值都为 0，此时禁止访问协处理器（禁止了硬件 FPU），我们将这 4 个位都设置为 1，即可完全访问协处理器（开启硬件 FPU），此时便可以使用 STM32F7 内置的硬件 FPU 了。CPACR 寄存器这 4 个位的设置，我们在 system_stm32f7xx_c 文件里面开启，代码如下：

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

此部分代码是系统初始化函数的部分内容，功能就是设置 CPACR 寄存器的 20~23 位为 1，以开启 STM32F7 的硬件 FPU 功能。从程序可以看出，只要我们定义了全局宏定义标识符__FPU_PRESENT 以及__FPU_USED 为 1，那么就可以开启硬件 FPU。其中宏定义标识符__FPU_PRESENT 用来确定处理器是否带 FPU 功能，标识符__FPU_USED 用来确定是否开启

FPU 功能。

实际上，因为 F7 是带 FPU 功能的，所以在我们的 stm32f767xx.h 头文件里面，我们默认是定义了__FPU_PRESENT 为 1。大家可以打开文件搜索即可找到下面一行代码：

```
#define __FPU_PRESENT             1
```

但是，仅仅只是说明处理器有 FPU 功能是不够的，我们还需要开启 FPU 功能。开启 FPU 有两种方法，第一种是直接在头文件 STM32f767xx.h 中定义宏定义标识符__FPU_USED 的值为 1。也可以直接在 MDK 编译器上面设置，我们在 MDK5 编译器里面，点击按钮，然后在 Target 选项卡里面，设置 Floating Point Hardware 为 Use Double Precision，如图 53.1.1.2 所示：

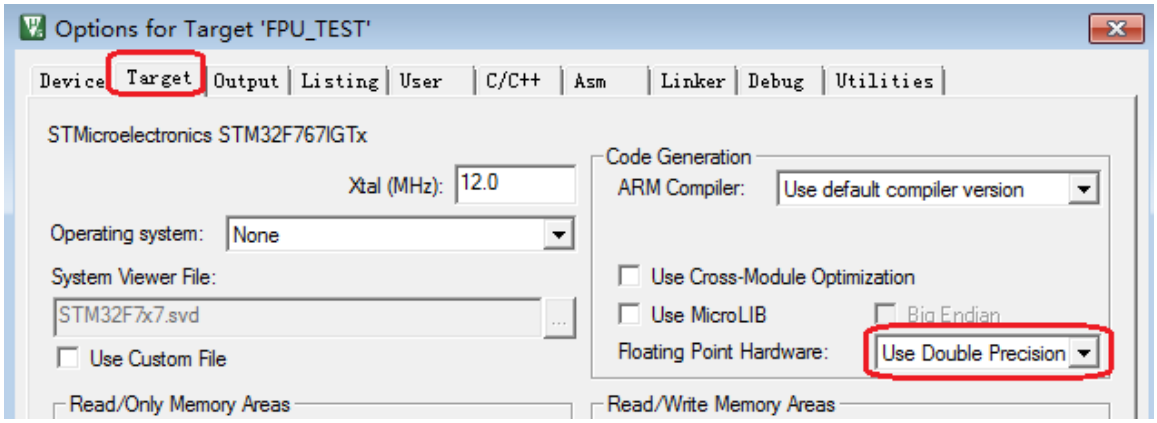


图 53.1.1.2 编译器开启硬件 FPU 选型

经过这个设置，编译器会自动加入标识符__FPU_USED 为 1。这样遇到浮点运算就会使用硬件 FPU 相关指令，执行浮点运算，从而大大减少计算时间。

最后，总结下 STM32F7 硬件 FPU 使用的要点：

1， 设置 CPACR 寄存器 bit20~23 为 1，使能硬件 FPU（参考 SystemInit 函数开头部分）。

2， MDK 编译器 Target 选项卡中 Floating Point Hardware 选项设置为：Use Double Precision。

经过这两步设置，我们的编写的浮点运算代码，即可使用 STM32F7 的硬件 FPU 了，可以大大加快浮点运算速度。

56.1.2 Julia 分形简介

Julia 分形即 Julia 集，它最早由法国数学家 Gaston Julia 发现，因此命名为 Julia（朱利亚）集。Julia 集合的生成算法非常简单：对于复平面的每个点，我们计算一个定义序列的发散速度。该序列的 Julia 集计算公式为：

$$z_n+1 = z_n^2 + c$$

针对复平面的每个 $x + i.y$ 点，我们用 $c = c_x + i.c_y$ 计算该序列：

$$x_{n+1} + i.y_{n+1} = x_n^2 - y_n^2 + 2.i.x_n.y_n + c_x + i.c_y$$

$$x_n+1 = x_n^2 - y_n^2 + c_x \text{ 且 } y_{n+1} = 2.x_n.y_n + c_y$$

一旦计算出的复值超出给定圆的范围（数值大小大于圆半径），序列便会发散，达到此限值时完成的迭代次数与该点相关。随后将该值转换为颜色，以图形方式显示复平面上各个点的分散速度。

经过给定的迭代次数后，若产生的复值保持在圆范围内，则计算过程停止，并且序列也不发散，本例程生成 Julia 分形图片的代码如下：

```
#define         ITERATION               128             //迭代次数
```

```
#define         REAL_CONSTANT     0.285f        //实部常量
#define         IMG_CONSTANT      0.01f         //虚部常量

//产生 Julia 分形图形
//size_x,size_y:屏幕 x,y 方向的尺寸
//offset_x,offset_y:屏幕 x,y 方向的偏移
//zoom:缩放因子
void GenerateJulia_fpu(u16 size_x,u16 size_y,u16 offset_x,u16 offset_y,u16 zoom)
{
    u8 i;
    u16 x,y;
    float tmp1,tmp2;
    float num_real,num_img;
    float radius;
    for(y=0;y<size_y;y++)
    {
        for(x=0;x<size_x;x++)
        {
            num_real=y-offset_y;
            num_real=num_real/zoom;
            num_img=x-offset_x;
            num_img=num_img/zoom;
            i=0;
            radius=0;
            while((i<ITERATION-1)&&(radius<4))
            {
                tmp1=num_real*num_real;
                tmp2=num_img*num_img;
                num_img=2*num_real*num_img+IMG_CONSTANT;
                num_real=tmp1-tmp2+REAL_CONSTANT;
                radius=tmp1+tmp2;
                i++;
            }
            if(lcdltdc.pwidth!=0)lcdbuf[lcddev.width-x-1]=color_map[i];
                                                    //保存颜色值到 lcdbuf
            else LCD->LCD_RAM=color_map[i];//绘制到屏幕
        }
        if(lcdltdc.pwidth!=0)LTDC_Color_Fill(0,y,lcddev.width-1,y,lcdbuf);  //DM2D 填充

    }
}
```

这种算法非常有效地展示了 FPU 的优势：无需修改代码，只需在编译阶段激活或禁止 FPU（在 MDK 的 Float Point Hardware 选项里面设置：Use Double Precision/Not Used），即可

测试使用硬件 FPU 和不使用硬件 FPU 的差距。

注意，是该函数将颜色数据填充到 LCD 的时候，根据 MCU 屏还是 RGB 屏，做了不同的处理：MCU 屏可以直接写 LCD_RAM，将颜色显示到 LCD 上面；而 RGB 屏，则需要先缓存到 lcdbuf，然后通过 DMA2D 一次性填充，以提高速度。

56.2 硬件设计

本章实验功能简介：开机后，根据迭代次数生成颜色表（RGB565），然后计算 Julia 分形，并显示到 LCD 上面。同时，程序开启了定时器 3，用于统计一帧所要的时间（ms），在一帧 Julia 分形图片显示完成后，程序会显示运行时间、当前是否使用 FPU 和缩放因子（zoom）等信息，方便观察对比。KEY0/KEY2 用于调节缩放因子，KEY_UP 用于设置自动缩放，还是手动缩放。DS0 用于提示程序运行状况。

本实验用到的资源如下：

1，指示灯 DS0

2，三个按键（KEY_UP/KEY0/KEY2）

3，串口

4，LCD 模块

这些前面都已介绍过。

56.3 软件设计

本章代码，分成两个工程：

1，实验 51_1 FPU 测试(Julia 分形)实验_开启硬件 FPU

2，实验 51_2 FPU 测试(Julia 分形)实验_关闭硬件 FPU

这两个工程的代码一模一样，只是前者使用硬件 FPU 计算 Julia 分形集（MDK 设置 Use Double Precision），后者使用 IEEE-754 标准计算 Julia 分形集（MDK 设置 Not Used）。由于两个工程代码一模一样，我们这里仅介绍其中一个：实验 51_1 FPU 测试(Julia 分形)实验_开启硬件 FPU。

本章代码，我们在 TFTLCD 显示实验的基础上修改，打开 TFTLCD 显示实验的工程，由于要统计帧时间和按键设置，所以在 HARDWARE 组下加入 timer.c 和 key.c 两个文件。

本章不需要添加其他.c 文件，所有代码均在 main.c 里面实现，整个代码如下：

```
//FPU 模式提示
#if __FPU_USED==1
#define SCORE_FPU_MODE                  "FPU On"
#else
#define SCORE_FPU_MODE                  "FPU Off"
#endif
#define     ITERATION           128         //迭代次数
#define     REAL_CONSTANT       0.285f      //实部常量
#define     IMG_CONSTANT        0.01f       //虚部常量
//颜色表
u16 color_map[ITERATION];
//缩放因子列表
const u16 zoom_ratio[] =
{
```

```
    120, 110, 100, 150, 200, 275, 350, 450,
    600, 800, 1000, 1200, 1500, 2000, 1500,
    1200, 1000, 800, 600, 450, 350, 275, 200,
    150, 100, 110,
};
//初始化颜色表
//clut:颜色表指针
void InitCLUT(u16 * clut)
{
    u32 i=0x00;
    u16   red=0,green=0,blue=0;
    for(i=0;i<ITERATION;i++)//产生颜色表
    {
        //产生 RGB 颜色值
        red=(i*8*256/ITERATION)%256;
        green=(i*6*256/ITERATION)%256;
        blue=(i*4*256 /ITERATION)%256;
        //将 RGB888,转换为 RGB565
        red=red>>3;
        red=red<<11;
        green=green>>2;
        green=green<<5;
        blue=blue>>3;
        clut[i]=red+green+blue;
    }
}
//产生 Julia 分形图形
//size_x,size_y:屏幕 x,y 方向的尺寸
//offset_x,offset_y:屏幕 x,y 方向的偏移
//zoom:缩放因子
void GenerateJulia_fpu(u16 size_x,u16 size_y,u16 offset_x,u16 offset_y,u16 zoom)
{
    ……//代码省略，详见 53.1.2 节
}

u8 timeout;
int main(void)
{
    u8 key, i=0, autorun=0;
    float time;
    u8 buf[50];

    Cache_Enable();                 //打开 L1-Cache
```

```
HAL_Init();                     //初始化 HAL 库
Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
delay_init(216);                //延时初始化
uart_init(115200);              //串口初始化
LED_Init();                     //初始化 LED
KEY_Init();                     //初始化按键
SDRAM_Init();                   //初始化 SDRAM
LCD_Init();                     //LCD 初始化
TIM3_Init(65535,10800-1);       //10Khz 计数频率,最大计时 6.5 秒超出
……//此处省略部分代码
InitCLUT(color_map);            //初始化颜色表
while(1)
{
    key=KEY_Scan(0);
    switch(key)
    {
        case KEY0_PRES:
            i++;
            if(i>sizeof(zoom_ratio)/2-1)i=0;//限制范围
            break;
        case KEY2_PRES:
            if(i)i--;
            else i=sizeof(zoom_ratio)/2-1;
            break;
        case WKUP_PRES:
            autorun=!autorun;  //自动/手动
            break;
    }
    if(autorun==1)//自动时,自动设置缩放因子
    {
        i++;
        if(i>sizeof(zoom_ratio)/2-1)i=0;//限制范围
    }
    LCD_Set_Window(0,0,lcddev.width,lcddev.height);//设置窗口
    LCD_WriteRAM_Prepare();
    __HAL_TIM_SET_COUNTER(&TIM3_Handler,0);//重设 TIM3 定时器的计数器值
    timeout=0;
    GenerateJulia_fpu(lcddev.width,lcddev.height,lcddev.width/2,
                                                    lcddev.height/2,zoom_ratio[i]);
    time=__HAL_TIM_GET_COUNTER(&TIM3_Handler)+(u32)timeout*65536;
    sprintf((char*)buf,"%s: zoom:%d    runtime:%0.1fms\r\n",
                                        SCORE_FPU_MODE,zoom_ratio[i],time/10);
    LCD_ShowString(5,lcddev.height-5-12,lcddev.width-5,12,12,buf);//显示运行情况
```

```
            printf("%s",buf);//输出到串口
            LED0_Toggle;
        }
    }
```

这里面，总共 3 个函数：InitCLUT、GenerateJulia_fpu 和 main 函数。

InitCLUT 函数，该函数用于初始化颜色表，该函数根据迭代次数（ITERATION）计算出颜色表，这些颜色值将显示在 TFTLCD 上。

GenerateJulia_fpu 函数，该函数根据给定的条件计算 Julia 分形集，当迭代次数大于等于 ITERATION 或者半径大于等于 4 时，结束迭代，并在 TFTLCD 上面显示迭代次数对应的颜色值，从而得到漂亮的 Julia 分形图。我们可以通过修改 REAL_CONSTANT 和 IMG_CONSTANT 这两个常量的值来得到不同的 Julia 分形图。

main 函数，完成我们在 56.2 节所介绍的实验功能，代码比较简单。这里我们用到一个缩放因子表：zoom_ratio，里面存储了一些不同的缩放因子，方便演示效果。

最后，为了提高速度，同上一章一样，我们在 MDK 里面选择使用-O2 优化，优化代码速度，本例程代码就介绍到这里。

再次提醒大家：本例程两个代码（实验 51_1 和 51_2）程序是完全一模一样的，他们的区别就是 MDK→Options for Target ‘Target1’→Target 选项卡→Floating Point Hardware 的设置不一样，当设置 Use Double Precision 时，使用硬件 FPU；当设置 Not Used 时，不使用硬件 FPU。分别下载这两个代码，通过屏幕显示的 runtime 时间，即可看出速度上的区别。

56.4 下载验证

代码编译成功之后，下载本例程任意一个代码（这里以 51_1 为例）到 ALIENTEK 阿波罗 STM32 开发板上，可以看到 LCD 显示 Julia 分形图，并显示相关参数，如图 56.4.1 所示：

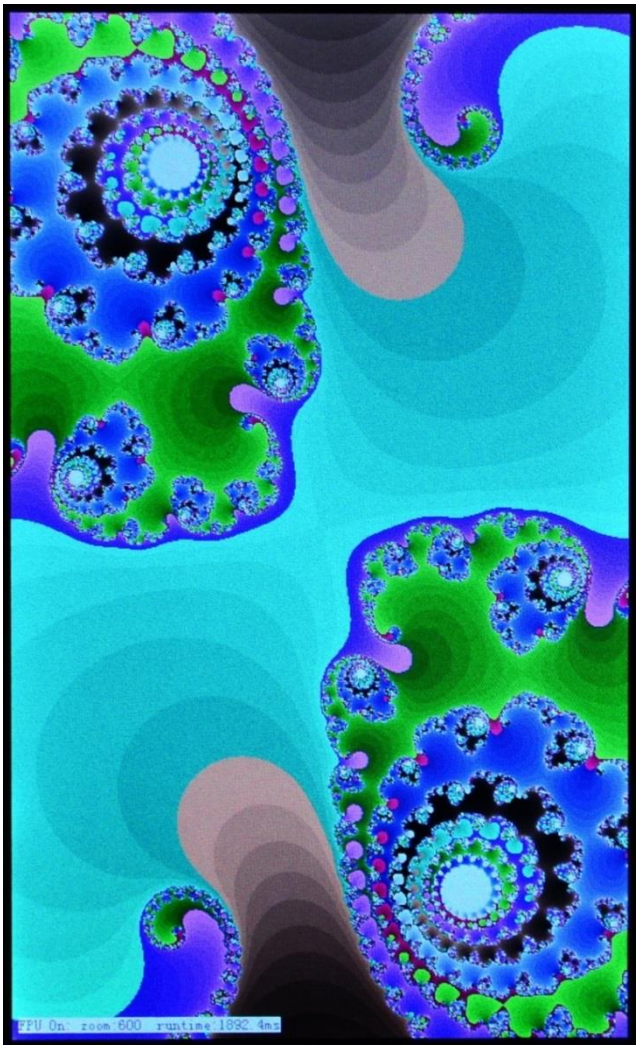


图 56.4.1 Julia 分形显示效果

实验 51_1 是开启了硬件 FPU 的，所以显示 Julia 分形图片速度比较快。如果下载实验 51_2，同样的缩放因子，会比实验 51_1 慢 11 倍左右。

因此可以看出，使用硬件 FPU 和不使用硬件 FPU 对比，同样的条件下，硬件 FPU 快了近 11 倍，充分体现了 STM32F767 硬件 FPU 的优势。

第五十七章 DSP 测试实验

上一章，我们在 ALIENTEK 阿波罗 STM32 开发板上测试了 STM32F767 的硬件 FPU。STM32F767 除了集成硬件 FPU 外，还支持多种 DSP 指令集。同时 ST 还提供了一整套 DSP 库方便我们工程中开发应用。

本章，我们将指导大家入门 STM32F767 的 DSP，手把手教大家搭建 DSP 库测试环境，同时通过对 DSP 库中的几个基本数学功能函数和 FFT 快速傅里叶变换函数的测试，让大家对 STM32F767 的 DSP 库有个基本的了解。本章分为如下几个部分：

57.1 DSP 简介与环境搭建
57.2 硬件设计
57.3 软件设计
57.4 下载验证

57.1 DSP 简介与环境搭建

本节将分两个部分：1，STM32F7 DSP 简介；2，DSP 库运行环境搭建

57.1.1 STM32F7 DSP 简介

STM32F7 采用 Cortex-M7 内核，相比 Cortex-M3 系列除了内置硬件 FPU 单元，在数字信号处理方面还增加了 DSP 指令集，支持诸如单周期乘加指令（MAC），优化的单指令多数据指令（SIMD），饱和算数等多种数字信号处理指令集。相比 Cortex-M3，Cortex-M4 在数字信号处理能力方面得到了大大的提升。Cortex-M7 执行所有的 DSP 指令集都可以在单周期内完成，而 Cortex-M3 需要多个指令和多个周期才能完成同样的功能。

接下来我们来看看 Cortex-M7 的两个 DSP 指令：MAC 指令（32 位乘法累加）和 SIMD 指令。

32 位乘法累加（MAC）单元包括新的指令集，能够在单周期内完成一个 32×32+ 64→ 64 的操作或两个 16×16 的操作，其计算能力，如表 57.1.1.1 所示：

计算	指令	周期
16 × 16 = 32	SMULBB , SMULBT , SMULTB , SMULTT	1
16 × 16 + 32 = 32	SMLABB , SMLABT , SMLATB , SMLATT	1
16 × 16 + 64 = 64	SMLALBB , SMLALBT , SMLALTB , SMLALTT	1
16 × 32 = 32	SMULWB , SMULWT	1
(16 × 32) + 32 = 32	SMLAWB , SMLAWT	1
(16 × 16) ± (16 × 16) = 32	SMUAD , SMUADX , SMUSD , SMUSDX	1
(16 × 16) ± (16 × 16) + 32 = 32	SMLAD , SMLADX , SMLSD , SMLSDX	1
(16 × 16) ± (16 × 16) + 64 = 64	SMLALD , SMLALDX , SMLSLD , SMLSLDX	1
32 × 32 = 32	MUL	1
32 ± (32 × 32) = 32	MLA , MLS	1
32 × 32 = 64	SMULL , UMULL	1
(32 × 32) + 64 = 64	SMLAL , UMLAL	1
(32 × 32) + 32 + 32 = 64	UMAAL	1
2 ± (32 × 32) = 32 (上)	SMMLA , SMMLAR , SMMLS , SMMLSR	1
(32 × 32) = 32 (上)	SMMUL , SMMULR	1

图 57.1.1.1 32 位乘法累加（MAC）单元的计算能力

Cortex-M7 支持 SIMD 指令集，这在 Cortex-M3/M0 系列是不可用的。上述表中的指令，有的属于 SIMD 指令。与硬件乘法器一起工作（MAC），使所有这些指令都能在单个周期内执行。受益于 SIMD 指令的支持，Cortex-M4 处理器能在单周期内完成高达 32×32+64→64 的运算，为其他任务释放处理器的带宽，而不是被乘法和加法消耗运算资源。

比如一个比较复杂的运算：两个 16×16 乘法加上一个 32 位加法，如图 57.1.1.2 所示：

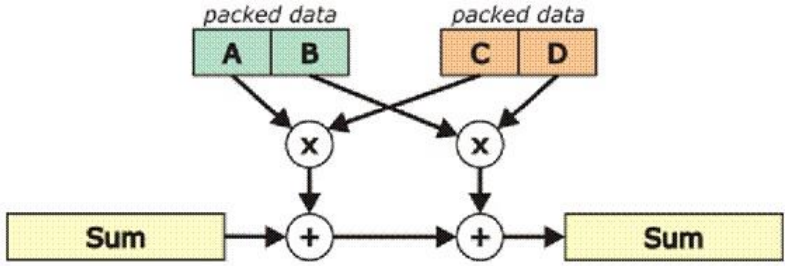


图 57.1.1.2 SUM 运算过程

以上图片所示的运算，即：SUM = SUM +（A* C）+（B *D），在 STM32F7 上面，可以被编译成由一条单周期指令完成。

上面我们简单的介绍了 Cortex-M7 的 DSP 指令，接下来我们来介绍一下 STM32F7 的 DSP 库。STM32F7 的 DSP 库源码和测试实例在 ST 提供的 HAL 库：en.stm32cubef7.zip 里面就有（该文件可以在 www.st.com 网站下载，搜索 STM32CubeF7 即可找到最新版本），该文件在：光盘→ 8，STM32 参考资料→1，STM32CubeF7 固件包文件夹里面，解压该文件，即可找到 ST 提供的 DSP 库，详细路径为：光盘→8，STM32 参考资料→ 1，STM32CubeF7 固件包→ STM32Cube_FW_F7_V1.4.0→ Drivers→CMSIS→
DSP_Lib，该文件夹下目录结构如图 57.1.1.3 所示：

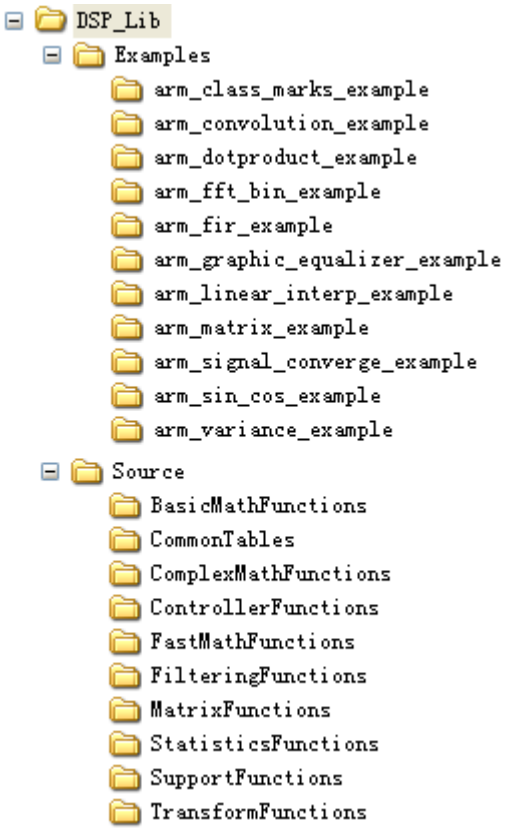


图 57.1.1.3 DSP_Lib 目录结构

DSP_Lib 源码包的 Source 文件夹是所有 DSP 库的源码，Examples 文件夹是相对应的一些测试实例。这些测试实例都是带 main 函数的，也就是拿到工程中可以直接使用。接下来我们一一讲解一下 Source 源码文件夹下面的子文件夹包含的 DSP 库的功能。

BasicMathFunctions

基本数学函数：提供浮点数的各种基本运算函数，如向量加减乘除等运算。

CommonTables

arm_common_tables.c 文件提供位翻转或相关参数表。

ComplexMathFunctions

复杂数学功能，如向量处理，求模运算的。

ControllerFunctions

控制功能函数。包括正弦余弦，PID 电机控制，矢量 Clarke 变换，矢量 Clarke 逆变换等。

FastMathFunctions

快速数学功能函数。提供了一种快速的近似正弦，余弦和平方根等相比 CMSIS 计算库要快的数学函数。

FilteringFunctions

滤波函数功能，主要为 FIR 和 LMS（最小均方根）等滤波函数。

MatrixFunctions

矩阵处理函数。包括矩阵加法、矩阵初始化、矩阵反、矩阵乘法、矩阵规模、矩阵减法、矩阵转置等函数。

StatisticsFunctions

统计功能函数。如求平均值、最大值、最小值、计算均方根 RMS、计算方差/标准差等。

SupportFunctions

支持功能函数，如数据拷贝，Q 格式和浮点格式相互转换，Q 任意格式相互转换。

TransformFunctions

变换功能。包括复数 FFT（CFFT）/复数 FFT 逆运算（CIFFT）、实数 FFT（RFFT）/实数 FFT 逆运算（RIFFT）、和 DCT（离散余弦变换）和配套的初始化函数。

所有这些 DSP 库代码合在一起是比较多的，因此，ST 为我们提了.lib 格式的文件，方便使用。这些.lib 文件就是由 Source 文件夹下的源码编译生成的，如果想看某个函数的源码，大家可以在 Source 文件夹下面查找。.lib 格式文件 HAL 库包路径：Drivers→CMSIS→Lib→ARM，总共有 6 个.lib 文件，如下：

① arm_cortexM7b_math.lib （Cortex-M7 大端模式）
② arm_cortexM7l_math.lib （Cortex-M7 小端模式）
③ arm_cortexM7bfdp_math.lib （双精度浮点 Cortex-M7 大端模式）
④ arm_cortexM7lfdp_math.lib （双精度浮点 Cortex-M7 小端模式）
⑤ arm_cortexM7bfsp_math.lib （单精度浮点 Cortex-M7 大端模式）
⑥ arm_cortexM7lfsp_math.lib （单精度浮点 Cortex-M7 小端模式）

我们得根据所用 MCU 内核类型以及端模式来选择符合要求的.lib 文件，本章我们所用的 STM32F7 属于 CortexM7F 内核，双精度浮点小端模式，应选择：arm_cortexM7lfdp_math.lib (双精度浮点 Cortex-M7 小端模式)。

对于 DSP_Lib 的子文件夹 Examples 下面存放的文件，是 ST 官方提供的一些 DSP 测试代码，提供简短的测试程序，方便上手，有兴趣的朋友可以根据需要自行测试。

57.1.2 DSP 库运行环境搭建

本节我们将讲解怎么搭建 DSP 库运行环境，只要运行环境搭建好了，使用 DSP 库里面的函数来做相关处理就非常简单了。本节，我们将以上一章例程（实验 52_1）为基础，搭建 DSP 运行环境。

在 MDK 里面搭建 STM32F7 的 DSP 运行环境(使用.lib 方式)是很简单的，分为 3 个步骤：

1， 添加文件。

首先，我们在例程工程目录下新建：DSP_LIB 文件夹，存放我们将要添加的文件：arm_cortexM7lfdp_math.lib 和相关头文件，如图 57.1.2.1 所示：

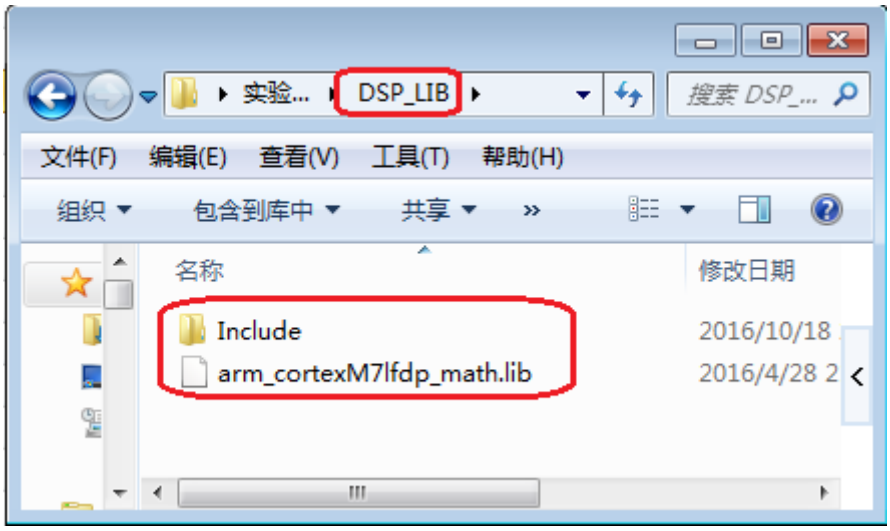


图 57.1.2.1 DSP_LIB 文件夹添加文件

其中 arm_cortexM7lfdp_math.lib 的由来，在 57.1.1 节已经介绍过了。Include 文件夹，则是直接拷贝：STM32Cube_FW_F7_V1.4.0→Drivers→CMSIS→Include 这个文件夹，里面包含了我们可能要用到的相关头文件。

然后，打开工程，新建 DSP_LIB 分组，并将 arm_cortexM7lfdp_math.lib 添加到工程，如图 57.1.2.2 所示：

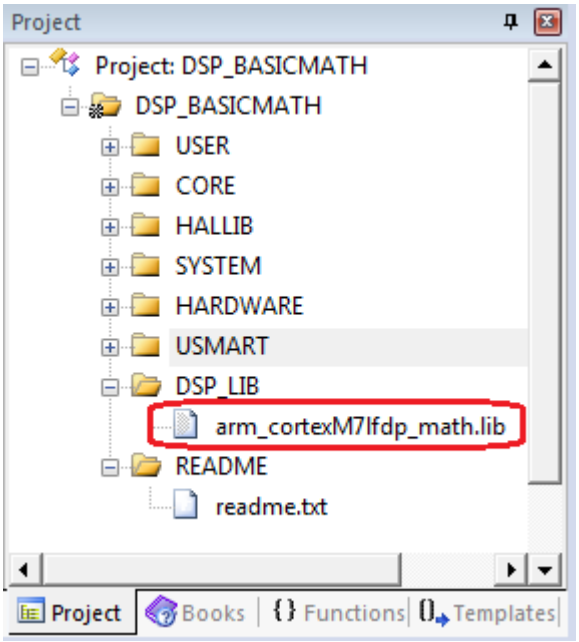


图 57.1.2.2 添加.lib 文件

这样，添加文件就结束了（就添加了一个.lib 文件）。

2， 添加头文件包含路径

添加好.lib 文件后，我们要添加头文件包含路径，将第一步拷贝的 Include 文件夹和 DSP_LIB 文件夹，加入头文件包含路径，如图 57.1.2.3 所示：

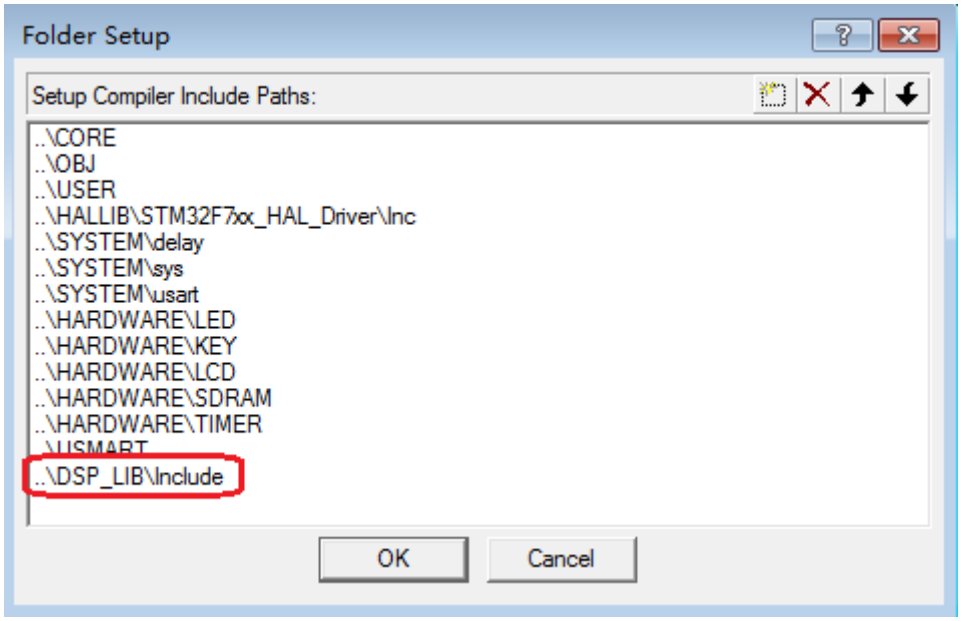


图 57.1.2.3 添加相关头文件包含路径

3， 添加全局宏定义

最后，为了使用 DSP 库的所有功能，我们还需要添加几个全局宏定义：

1，__FPU_USED

2，__FPU_PRESENT

3，ARM_MATH_CM7

4，__CC_ARM

5，ARM_MATH_MATRIX_CHECK

6，ARM_MATH_ROUNDING

添加方法：点击→C/C++选项卡，然后在 Define 里面进行设置，如图 57.1.2.4 所示：

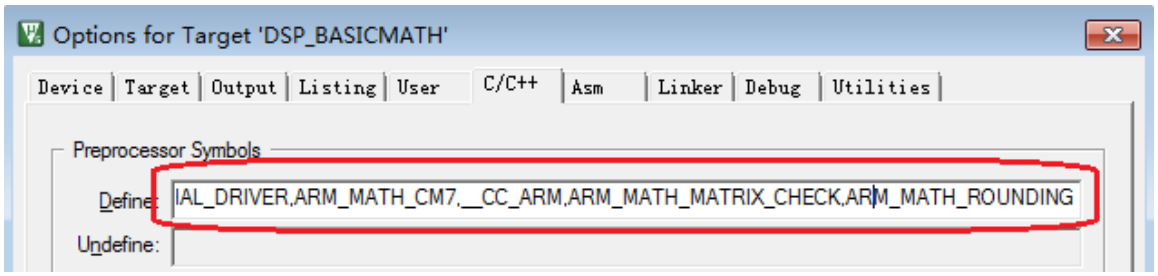


图 57.1.2.4 DSP 库支持全局宏定义设置

这里，两个宏之间用“,”隔开。并且，上面的全局宏里面，我们没有添加__FPU_USED，因为这个宏定义在 Target 选项卡设置 Floating Point Hardware 的时候（上一章有介绍），选择了：Use Double Precision（如果没有设置 Use Double Precision，则必须设置!!），故 MDK 会自动添加这个全局宏，因此不需要我们手动添加了。同时__FPU_PRESENT 全局宏我们 FPU 实验已经

讲解，这个宏定义在 stm32f7xx.h 头文件里面已经定义。这样，在 Define 处要输入的所有宏为：STM32F767xx,USE_HAL_DRIVER,ARM_MATH_CM7,__CC_ARM,ARM_MATH_MATRIX_CHECK,ARM_MATH_ROUNDING 共 6 个。

至此，STM32F7 的 DSP 库运行环境就搭建完成了。

特别注意，为了方便调试，本章例程我们将 MDK 的优化设置为-O0 优化，以得到最好的调试效果。

57.2 硬件设计

本例程包含 2 个源码：实验 52_1 DSP BasicMath 测试和实验 52_2 DSP FFT 测试，他们除了 main.c 里面内容不一样外，其他源码完全一模一样（包括 MDK 配置）。

实验 52_1 DSP BasicMath 测试 实验功能简介：测试 STM32F7 的 DSP 库基础数学函数：arm_cos_f32 和 arm_sin_f32 和标准库基础数学函数：cosf 和 sinf 的速度差别，并在 LCD 屏幕上面显示两者计算所用时间，DS0 用于提示程序正在运行。

实验 52_2 DSP FFT 测试 实验功能简介：测试 STM32F7 的 DSP 库的 FFT 函数，程序运行后，自动生成 1024 点测试序列，然后，每当 KEY0 按下后，调用 DSP 库的 FFT 算法（基 4 法）执行 FFT 运算，在 LCD 屏幕上面显示运算时间，同时将 FFT 结果输出到串口，DS0 用于提示程序正在运行。

本实验用到的资源如下：

1，指示灯 DS0

2，KEY0 按键

3，串口

4，TFTLCD 模块

这些前面都已介绍过。

57.3 软件设计

本章代码，分成两个工程：1，实验 52_1 DSP BasicMath 测试；2，实验 52_2 DSP FFT 测试，接下来我们分别介绍。

57.3.1 DSP BasicMath 测试

这是我们使用 STM32F7 的 DSP 库进行基础数学函数测试的一个例程。使用大家耳熟能详的公式进行计算：

$$\sin(x)^2+\cos(x)^2=1$$

这里我们用到的就是 sin 和 cos 函数，不过实现方式不同。MDK 的标准库（math.h）提供我们：sin、cos、sinf 和 cosf 等 4 个函数，带 f 的表示单精度浮点型运算，即 float 型，而不带 f 的表示双精度浮点型，即 double。

STM32F7 的 DSP 库，则提供我们另外两个函数：arm_sin_f32 和 arm_cos_f32（**注意：需要添加：arm_math.h 头文件才可使用!!!**），这两个函数也是单精度浮点型的，用法同 sinf 和 cosf 一模一样。

本例程就是测试：arm_sin_f32& arm_cos_f32 同 sinf&cosf 的速度差别。

因为 57.1.2 节已经搭建好 DSP 库运行环境了，所以我们这里直接只需要修改 main.c 里面的代码即可，main.c 代码如下：

```
#include "math.h"
```

```
#include "arm_math.h"

#define  DELTA  0.00005f      //误差值

//sin cos 测试
//angle:起始角度  times:运算次数  mode:0,不使用 DSP 库;1,使用 DSP 库
//返回值：0,成功;0XFF,出错
u8 sin_cos_test(float angle,u32 times,u8 mode)
{
    float sinx,cosx,result;
    u32 i=0;
    if(mode==0)
    {
        for(i=0;i<times;i++)
        {
            cosx=cosf(angle);              //不使用 DSP 优化的 sin，cos 函数
            sinx=sinf(angle);
            result=sinx*sinx+cosx*cosx;    //计算结果应该等于 1
            result=fabsf(result-1.0f);     //对比与 1 的差值
            if(result>DELTA)return 0XFF;   //判断失败
            angle+=0.001f;                 //角度自增
        }
    }else
    {
        for(i=0;i<times;i++)
        {

            cosx=arm_cos_f32(angle);       //使用 DSP 优化的 sin，cos 函数
            sinx=arm_sin_f32(angle);
            result=sinx*sinx+cosx*cosx;    //计算结果应该等于 1
            result=fabsf(result-1.0f);     //对比与 1 的差值
            if(result>DELTA)return 0XFF;   //判断失败
            angle+=0.001f;                 //角度自增
        }
    }
    return 0;//任务完成
}
u8 timeout;

int main(void)
{
    float time;
    u8 buf[50];
```

```
    u8 res;

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //LCD 初始化
    TIM3_Init(65535,10800-1);       //10Khz 计数频率,最大计时 6.5 秒超出
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,70,200,16,16,"DSP BasicMath TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2016/1/17");
    LCD_ShowString(30,150,200,16,16," No DSP runtime:");  //显示提示信息
    LCD_ShowString(30,190,200,16,16,"Use DSP runtime:");  //显示提示信息
    POINT_COLOR=BLUE;      //设置字体为蓝色
    while(1)
    {
        //不使用 DSP 优化
        __HAL_TIM_SET_COUNTER(&TIM3_Handler,0);//重设 TIM3 定时器的计数器值
        timeout=0;
        res=sin_cos_test(PI/6,200000,0);
        time=__HAL_TIM_GET_COUNTER(&TIM3_Handler)+(u32)timeout*65536;
        sprintf((char*)buf,"%0.1fms\r\n",time/10);
        if(res==0)LCD_ShowString(30+16*8,150,100,16,16,buf);  //显示运行时间
        else LCD_ShowString(30+16*8,150,100,16,16,"error！");  //显示当前运行情况
        //使用 DSP 优化
        __HAL_TIM_SET_COUNTER(&TIM3_Handler,0);//重设 TIM3 定时器的计数器值
        timeout=0;
        res=sin_cos_test(PI/6,200000,1);
        time=__HAL_TIM_GET_COUNTER(&TIM3_Handler)+(u32)timeout*65536;
        sprintf((char*)buf,"%0.1fms\r\n",time/10);
        if(res==0)LCD_ShowString(30+16*8,190,100,16,16,buf);  //显示运行时间
        else LCD_ShowString(30+16*8,190,100,16,16,"error！");  //显示错误
        LED0_Toggle;
    }
}
```

这里包括 2 个函数：sin_cos_test 和 main 函数，sin_cos_test 函数用于根据给定参数，执行 $sin(x)^2+cos(x)^2=1$

的计算。计算完后，计算结果同给定的误差值（DELTA）对比，如果不大于误差值，则认为计算成功，否则计算失败。该函数可以根据给定的模式参数(mode)来决定使用哪个基础数学函数执行运算，从而得出对比。

main 函数则比较简单，这里我们通过定时器 3 来统计 sin_cos_test 运行时间，从而得出对比数据。主循环里面，每次循环都会两次调用 sin_cos_test 函数，首先采用不使用 DSP 库方式计算，然后采用使用 DSP 库方式计算，并得出两次计算的时间，显示在 LCD 上面。

DSP 基础数学函数测试的程序设计就讲解到这里。

57.3.1 DSP FFT 测试

这是我们使用 STM32F7 的 DSP 库进行 FFT 函数测试的一个例程。

首先，我们简单介绍下 FFT：FFT 即快速傅里叶变换，可以将一个时域信号变换到频域。因为有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了，这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。简而言之，FFT 就是将一个信号从时域变换到频域方便我们分析处理。

在实际应用中，一般的处理过程是先对一个信号在时域进行采集，比如我们通过 ADC，按照一定大小采样频率 F 去采集信号，采集 N 个点，那么通过对这 N 个点进行 FFT 运算，就可以得到这个信号的频谱特性。

这里还涉及到一个采样定理的概念：在进行模拟/数字信号的转换过程中，当采样频率 F 大于信号中最高频率 f_{max} 的 2 倍时($F>2*f_{max}$)，采样之后的数字信号完整地保留了原始信号中的信息，采样定理又称奈奎斯特定理。举个简单的例子：比如我们正常人发声，频率范围一般在 8KHz 以内，那么我们要通过采样之后的数据来恢复声音，我们的采样频率必须为 8KHz 的 2 倍以上，也就是必须大于 16KHz 才行。

模拟信号经过 ADC 采样之后，就变成了数字信号，采样得到的数字信号，就可以做 FFT 变换了。N 个采样点数据，在经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。

假设采样频率为 F，对一个信号采样，采样点数为 N，那么 FFT 之后结果就是一个 N 点的复数，每一个点就对应着一个频率点（以基波频率为单位递增），这个点的模值（sqrt(实部 ^{2}+虚部 ^{2})）就是该频点频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 N/2 倍，而第一个点就是直流分量，它的模值就是直流分量的 N 倍。

这里还有个基波频率，也叫频率分辨率的概念，就是如果我们按照 F 的采样频率去采集一个信号，一共采集 N 个点，那么基波频率（频率分辨率）就是 f_k=F/N。这样，第 n 个点对应信号频率为：F*(n-1)/N；其中 n≥1，当 n=1 时为直流分量。

关于 FFT 我们就介绍到这。

如果我们要自己实现 FFT 算法，对于不懂数字信号处理的朋友来说，是比较难的，不过，ST 提供的 STM32F7 DSP 库里面就有 FFT 函数给我们调用，因此我们只需要知道如何使用这些函数，就可以迅速的完成 FFT 计算，而不需要自己学习数字信号处理，去编写代码了，大大方便了我们的开发。

STM32F7 的 DSP 库里面，提供了定点和浮点 FFT 实现方式，并且有基 4 的也有基 2 的，大家可以根据需要自由选择实现方式。注意：对于基 4 的 FFT 输入点数必须是 4^n，而基 2 的 FFT 输入点数则必须是 2^n，并且基 4 的 FFT 算法要比基 2 的快。

本章我们将采用 DSP 库里面的基 4 浮点 FFT 算法来实现 FFT 变换，并计算每个点的模值，

所用到的函数有：

```
arm_status arm_cfft_radix4_init_f32(
  arm_cfft_radix4_instance_f32 * S,
  uint16_t fftLen,uint8_t ifftFlag,uint8_t bitReverseFlag)
void arm_cfft_radix4_f32(const arm_cfft_radix4_instance_f32 * S,float32_t * pSrc)
void arm_cmplx_mag_f32(float32_t * pSrc,float32_t * pDst,uint32_t numSamples)
```

第一个函数 arm_cfft_radix4_init_f32，用于初始化 FFT 运算相关参数，其中：fftLen 用于指定 FFT 长度（16/64/256/1024/4096），本章设置为 1024；ifftFlag 用于指定是傅里叶变换(0)还是反傅里叶变换(1)，本章设置为 0；bitReverseFlag 用于设置是否按位取反，本章设置为 1；最后，所有这些参数存储在一个 arm_cfft_radix4_instance_f32 结构体指针 S 里面。

第二个函数 arm_cfft_radix4_f32 就是执行基 4 浮点 FFT 运算的，pSrc 传入采集到的输入信号数据（实部+虚部形式），同时 FFT 变换后的数据，也按顺序存放在 pSrc 里面，pSrc 必须大于等于 2 倍 fftLen 长度。另外，S 结构体指针参数是先由 arm_cfft_radix4_init_f32 函数设置好，然后传入该函数的。

第三个函数 arm_cmplx_mag_f32 用于计算复数模值，可以对 FFT 变换后的结果数据，执行取模操作。pSrc 为复数输入数组（大小为 2*numSamples）指针，指向 FFT 变换后的结果；pDst 为输出数组（大小为 numSamples）指针，存储取模后的值；numSamples 就是总共有多少个数据需要取模。

通过这三个函数，我们便可以完成 FFT 计算，并取模值。本节例程（实验 49_2 DSP FFT 测试）同样是在 57.1.2 节已经搭建好 DSP 库运行环境上面修改代码，只需要修改 main.c 里面的代码即可，本例程 main.c 代码如下：

```
#define FFT_LENGTH        1024            //FFT 长度，默认是 1024 点 FFT

float fft_inputbuf[FFT_LENGTH*2];   //FFT 输入数组
float fft_outputbuf[FFT_LENGTH];    //FFT 输出数组

u8 timeout;
int main(void)
{
        arm_cfft_radix4_instance_f32 scfft;
    u8 key,t=0;
    float time;
    u8 buf[50];
    u16 i;

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
```

```
    LCD_Init();                             //LCD 初始化
    TIM3_Init(65535,108-1);                 //1Mhz 计数频率,最大计时 6.5 秒超出
    ……//此处省略部分代码
    arm_cfft_radix4_init_f32(&scfft,FFT_LENGTH,0,1);
                                            //初始化 scfft 结构体，设定 FFT 相关参数
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY0_PRES)
        {
            for(i=0;i<FFT_LENGTH;i++)//生成信号序列
            {
                fft_inputbuf[2*i]=100+
                                    10*arm_sin_f32(2*PI*i/FFT_LENGTH)+
                                    30*arm_sin_f32(2*PI*i*4/FFT_LENGTH)+
                                    50*arm_cos_f32(2*PI*i*8/FFT_LENGTH);
                                                    //生成输入信号实部
                fft_inputbuf[2*i+1]=0;//虚部全部为 0
            }
            __HAL_TIM_SET_COUNTER(&TIM3_Handler,0);//重设 TIM3 计数器值
            timeout=0;
            arm_cfft_radix4_f32(&scfft,fft_inputbuf);    //FFT 计算（基 4）
            time=__HAL_TIM_GET_COUNTER(&TIM3_Handler)+(u32)timeout*65536;
                                                    //计算所用时间
            sprintf((char*)buf,"%0.3fms\r\n",time/1000);
            LCD_ShowString(30+12*8,160,100,16,16,buf);  //显示运行时间
            arm_cmplx_mag_f32(fft_inputbuf,fft_outputbuf,FFT_LENGTH);
                                                    //把运算结果复数求模得幅值
            printf("\r\n%d point FFT runtime:%0.3fms\r\n",FFT_LENGTH,time/1000);
            printf("FFT Result:\r\n");
            for(i=0;i<FFT_LENGTH;i++)
            {
                printf("fft_outputbuf[%d]:%f\r\n",i,fft_outputbuf[i]);
            }
        }else delay_ms(10);
        t++;
        if((t%10)==0)LED0_Toggle;
    }
}
```

以上代码只有一个 main 函数，里面通过我们前面介绍的三个函数：arm_cfft_radix4_init_f32、arm_cfft_radix4_f32 和 arm_cmplx_mag_f32 来执行 FFT 变换并取模值。每当按下 KEY0 就会重新生成一个输入信号序列，并执行一次 FFT 计算，将 arm_cfft_radix4_f32 所用时间统计出来，显示在 LCD 屏幕上面，同时将取模后的模值通过串口打印出来。

这里，我们在程序上生成了一个输入信号序列用于测试，输入信号序列表达式：

```
fft_inputbuf[2*i]=100+
                  10*arm_sin_f32(2*PI*i/FFT_LENGTH)+
                  30*arm_sin_f32(2*PI*i*4/FFT_LENGTH)+
                  50*arm_cos_f32(2*PI*i*8/FFT_LENGTH); //实部
```

通过该表达式我们可知，信号的直流分量为 100，外加 2 个正弦信号和一个余弦信号，其幅值分别为 10、30 和 50。

关于输出结果分析，请看 57.4 节，软件设计我们就介绍到这里。

57.4 下载验证

代码编译成功之后，便可以下载到我们的阿波罗 STM32 开发板上验证了。

对于实验 52_1 DSP BasicMath 测试，下载后，可以在屏幕看到两种实现方式的速度差别，如图 57.4.1 所示：

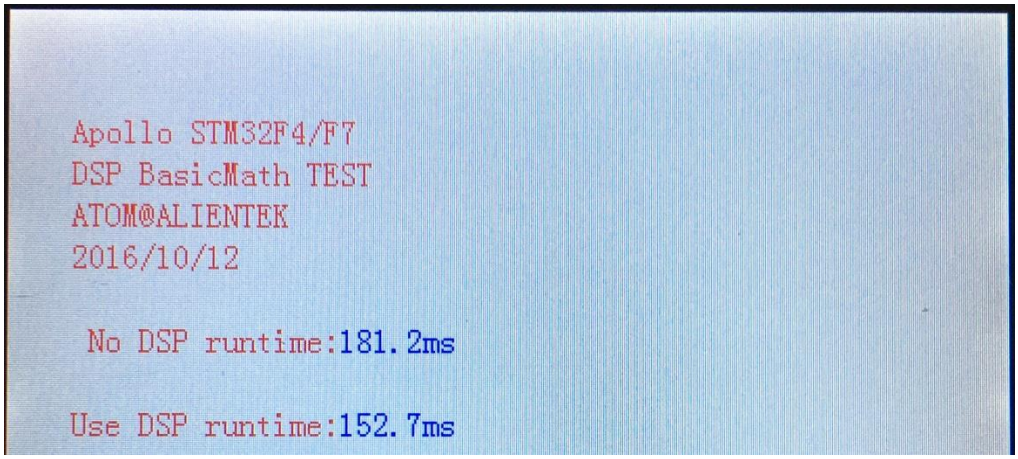


图 57.4.1 使用 DSP 库和不使用 DSP 库的基础数学函数速度对比

从图中可以看出，使用 DSP 库的基础数学函数计算所用时间比不使用 DSP 库的短，使用 STM32F7 的 DSP 库，速度上面比传统的实现方式提升了约 16%（**具体数据以实测为准**）。

对于实验 52_2 DSP FFT 测试，下载后，屏幕显示提示信息，然后我们按下 KEY0 就可以看到 FFT 运算所耗时间，如图 57.4.2 所示：

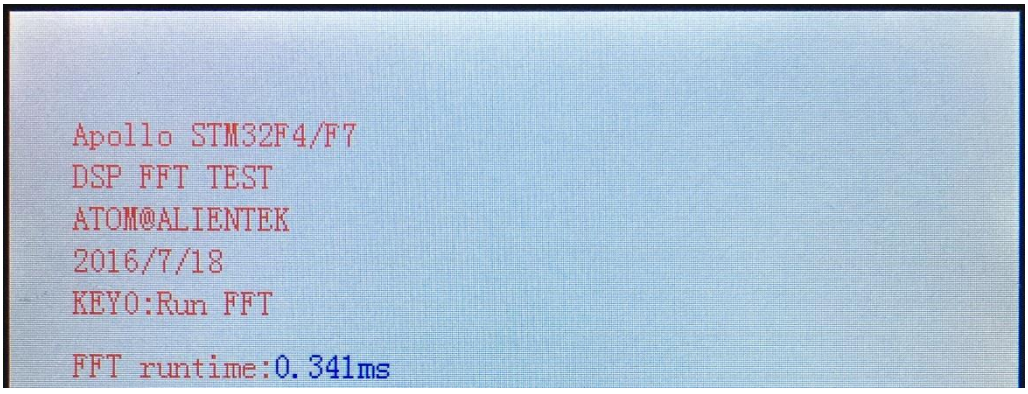


图 57.4.2 FFT 测试界面

可以看到，STM32F7 采用基 4 法计算 1024 个浮点数的 FFT，只用了 0.374ms，速度是相当快的了。同时，可以在串口看到 FFT 变换取模后的各频点模值，如图 57.4.3 所示：

图 57.4.3 FFT 变换后个频点模值

查看所有数据，会发现：第 0、1、4、8、1016、1020、1023 这 7 个点的值比较大，其他点的值都很小，接下来我们就简单分析一下这些数据。

由于 FFT 变换后的结果具有对称性，所以，实际上有用的数据，只有前半部分，后半部分和前半部分是对称关系，比如 1 和 1023，4 和 1020,8 和 1016 等，就是对称关系，因此我们只需要分析前半部分数据即可。这样，就只有第 0、1、4、8 这四个点，比较大，重点分析。

假设我们采样频率为 1024Hz，那么总共采集 1024 个点，频率分辨率就是 1Hz，对应到频谱上面，两个点之间的间隔就是 1Hz。因此，上面我们生成的三个叠加信号：10*sin(2*PI*i/1024)+30*sin(2*PI*i*4/1024)+50*cos(2*PI*i*8/1024)，频率分别是：1Hz、4Hz 和 8Hz。

对于上述 4 个值比较大的点，结合 57.3.1 节的知识，很容易分析得出：第 0 点，即直流分量，其 FFT 变换后的模值应该是原始信号幅值的 N 倍，N=1024，所以值是 100*1024=102400，与理论完全一样，然后其他点，模值应该是原始信号幅值的 N/2 倍，即 10*512、30*512、50*512，而我们计算结果是：5119.999023、15360、256000，除了第 1 个点，稍微有点点误差（说明精度上有损失），其他同理论值完全一致。

DSP 测试实验，我们就讲解到这里，DSP 库的其他测试实例，大家可以自行研究下，我们这里就不再介绍了。

第五十八章 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 阿波罗 STM32 开发板上实现一个简单的数字字母手写识别。本章分为如下几个部：

58.1 手写识别简介
58.2 硬件设计
58.3 软件设计
58.4 下载验证

58.1 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。阿波罗 STM32 开发板自带的 TFTLCD 触摸屏（2.8/3.5/4.3/7 寸等），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 58.1.1 所示：

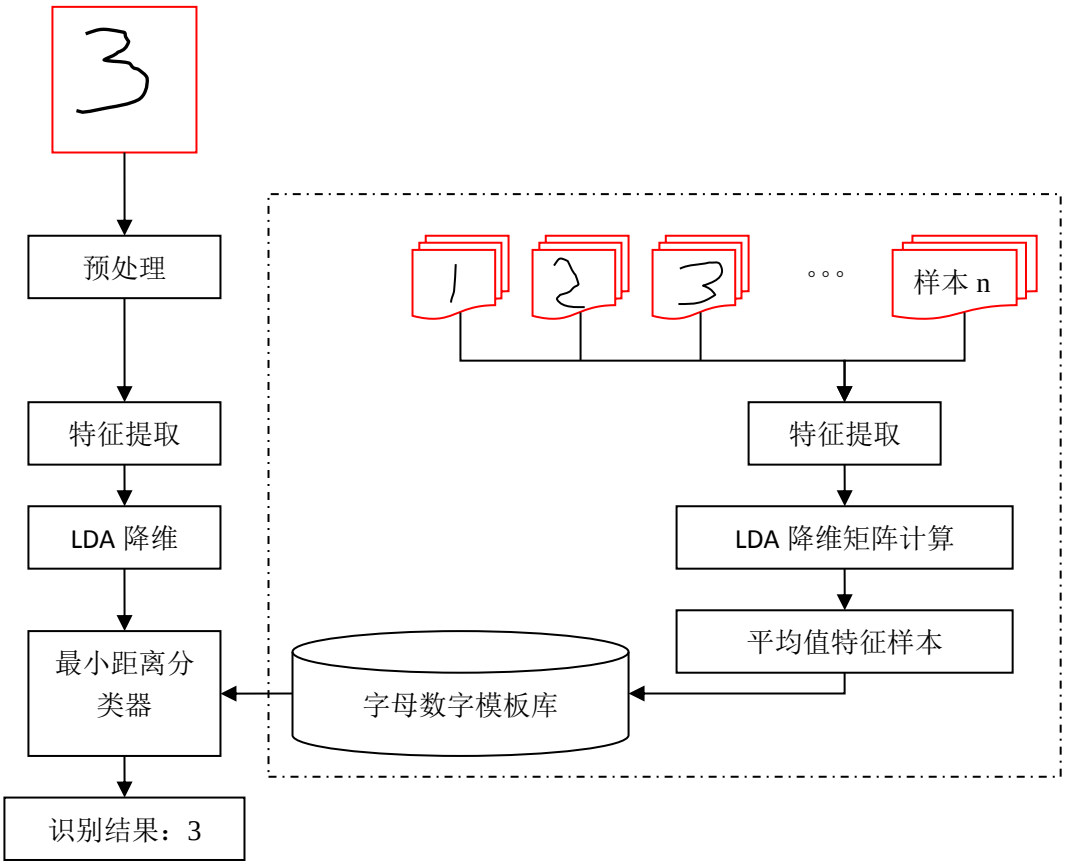


图 58.1.1 字母数字识别系统示意图。

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 中模式：

1，仅识别数字

2，进识别大写字母

3，仅识别小写字母

4，混合识别(全部识别)

最后一个参数是 result，用来输出结果，注意这个结果是 ASCII 码格式的。

alientek_ncr_memset、alientek_ncr_free 和 alientek_ncr_free 这 3 个函数在 atk_ncr.c 里面实现，这里就不多说了。

最后，我们看看通过 ALIENTEK 提供的手写数字字母识别库实现数字字母识别的步骤：

1） 调用 alientek_ncr_init 函数,初始化识别程序

该函数用来初始化识别器，在手写识别进行之前，必须调用该函数。

2） 获取输入的点阵数据

此步，我们通过触摸屏获取输入轨迹点阵坐标，然后存放到一个缓存区里面，注意至少要输入 2 个不同坐标的点阵数据，才能正常识别。注意输入点数不要太多，太多的话，需要更多的内存，我们推荐的输入点数范围：100~200 点。

3） 调用 alientek_ncr 函数,得到识别结果.

通过调用 alientek_ncr 函数，我们可以得到输入点阵的识别结果，结果将保存在 result 参数里面，采用 ASCII 码格式存储

4） 调用 alientek_ncr_stop 函数,终止识别.

如果不需要继续识别，则调用 alientek_ncr_stop 函数，终止识别器。如果还需要继续识别，重复步骤 2 和步骤 3 即可。

以上 4 个步骤，就是使用 ALIENTEK 手写识别库的方法，十分简单。

58.2 硬件设计

本章实验功能简介：开机的时候先初始化手写识别器，然后检测字库，之后进入等待输入状态。此时，我们在手写区写数字/字符，在每次写入结束后，自动进入识别状态，进行识别，然后将识别结果输出在 LCD 模块上面（同时打印到串口）。通过按 KEY0 可以进行模式切换（4 种模式都可以测试），通过按 KEY2，可以进入触摸屏校准（如果发现触摸屏不准，请执行此操作）。DS0 用于指示程序运行状态。

本实验用到的资源如下：

1） 指示灯 DS0

2） KEY0 和 KEY2 两个按键

3） 串口

4） LCD 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。

58.3 软件设计

打开本章实验工程目录可以看到，我们在工程根目录文件夹下新建一个 ATKNCR 的文件夹。将 ALIETENK 提供的手写识别库文件（ATKNCR_M_V2.0.lib、ATKNCR_N_V2.0.lib、atk_ncr.c 和 atk_ncr.h 这四个个文件，在光盘→ 4，程序源码→5，ATKNCR(数字字母手写识别库) 文件夹里面）拷贝到该文件夹下，然后在工程里面新建一个 ATKNCR 的组，将 atk_ncr.c 和 ATKNCR_M_V2.0.lib 加入到该组下面（这里我们使用内存管理版本的识别库）。最后，将 ATKNCR 文件夹加入头文件包含路径。

关于 ATKNCR_M_V2.0.lib 和 atk_ncr.c 前面已有介绍，我们这里就不再多说，我们在 main.c 里面修改代码如下：

```
//最大记录的轨迹点数
atk_ncr_point READ_BUF[200];
//画水平线
//x0,y0:坐标     len:线长度     color:颜色
void gui_draw_hline(u16 x0,u16 y0,u16 len,u16 color)
{
    if(len==0)return;
    LCD_Fill(x0,y0,x0+len-1,y0,color);
}
//画实心圆
//x0,y0:坐标     r:半径    color:颜色
void gui_fill_circle(u16 x0,u16 y0,u16 r,u16 color)
{
    ……//省略部分非关键代码，具体代码请参考实验源码
}
```

```
//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)
{
    ……//省略部分非关键代码，具体代码请参考实验源码
}
//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)
{
    ……//省略部分非关键代码，具体代码请参考实验源码
}
int main(void)
{
    u8 i=0, tcnt=0,res[10],key;
    u16 pcnt=0;
    u8 mode=4;                          //默认是混合模式
    u16 lastpos[2];                     //最后一次的数据

    Cache_Enable();                     //打开 L1-Cache
    HAL_Init();                         //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);       //设置时钟,216Mhz
    delay_init(216);                    //延时初始化
    uart_init(115200);                  //串口初始化
    usmart_dev.init(108);               //初始化 USMART
    LED_Init();                         //初始化 LED
    KEY_Init();                         //初始化按键
    SDRAM_Init();                       //初始化 SDRAM
    LCD_Init();                         //初始化 LCD
    W25QXX_Init();                      //初始化 W25Q256
    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 SDRAM 内存池
    my_mem_init(SRAMDTCM);              //初始化内部 CCM 内存池
    alientek_ncr_init();                //初始化手写识别
    while(font_init())                  //检查字库
    {
        LCD_ShowString(60,50,200,16,16,"Font Error!");
        delay_ms(200);
        LCD_Fill(60,50,240,66,WHITE);//清除显示
    }
```

```
RESTART:
    ……//此处省略部分代码
    tcnt=100;
    tcnt=100;
    while(1)
    {
        key=KEY_Scan(0);
        if(key==KEY2_PRES&&(tp_dev.touchtype&0X80)==0)
        {
            TP_Adjust();        //屏幕校准
            LCD_Clear(WHITE);
            goto RESTART;    //重新加载界面
        }
        if(key==KEY0_PRES)
        {
            LCD_Fill(20,115,219,314,WHITE);//清除当前显示
            mode++;
            if(mode>4)mode=1;
            switch(mode)
            {
                case 1:
                    Show_Str(80,207,200,16,"仅识别数字",16,0);
                    break;
                case 2:
                    Show_Str(64,207,200,16,"仅识别大写字母",16,0);
                    break;
                case 3:
                    Show_Str(64,207,200,16,"仅识别小写字母",16,0);
                    break;
                case 4:
                    Show_Str(88,207,200,16,"全部识别",16,0);
                    break;
            }
            tcnt=100;
        }
        tp_dev.scan(0);//扫描
        if(tp_dev.sta&TP_PRES_DOWN)//有按键被按下
        {
            delay_ms(1);//必要的延时,否则老认为有按键按下.
            tcnt=0;//松开时的计数器清空
            if((tp_dev.x[0]<(lcddev.width-20-2)&&tp_dev.x[0]>=(20+2))&& \
                    (tp_dev.y[0]<(lcddev.height-5-2)&&tp_dev.y[0]>=(115+2)))
            {
```

```
                if(lastpos[0]==0XFFFF)
                {
                    lastpos[0]=tp_dev.x[0];
                    lastpos[1]=tp_dev.y[0];
                }
                lcd_draw_bline(lastpos[0],lastpos[1],tp_dev.x[0],tp_dev.y[0],2,BLUE);/画线
                lastpos[0]=tp_dev.x[0];
                lastpos[1]=tp_dev.y[0];
                if(pcnt<200)//总点数少于 200
                {
                    if(pcnt)
                    {
                        if((READ_BUF[pcnt-1].y!=tp_dev.y[0])&& \
                                (READ_BUF[pcnt-1].x!=tp_dev.x[0]))//x,y 不相等
                        {
                            READ_BUF[pcnt].x=tp_dev.x[0];
                            READ_BUF[pcnt].y=tp_dev.y[0];
                            pcnt++;
                        }
                    }else
                    {
                        READ_BUF[pcnt].x=tp_dev.x[0];
                        READ_BUF[pcnt].y=tp_dev.y[0];
                        pcnt++;
                    }
                }
            }
        }else //按键松开了
        {
            lastpos[0]=0XFFFF;
            tcnt++;
            delay_ms(10);
            //延时识别
            i++;
            if(tcnt==40)
            {
                if(pcnt)//有有效的输入
                {
                    printf("总点数:%d\r\n",pcnt);
                    alientek_ncr(READ_BUF,pcnt,6,mode,(char*)res);
                    printf("识别结果:%s\r\n",res);
                    pcnt=0;
                    POINT_COLOR=BLUE;//设置画笔蓝色
```

```
                        LCD_ShowString(60+72,90,200,16,16,res);
                    }
                    LCD_Fill(20,115,lcddev.width-20-1,lcddev.height-5-1,WHITE);
                }
            }
            if(i==30)
            {
                i=0;
                LED0_Toggle;
            }
        }
    }
```

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 58.1.2 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。

58.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到，如图 58.4.1 所示：

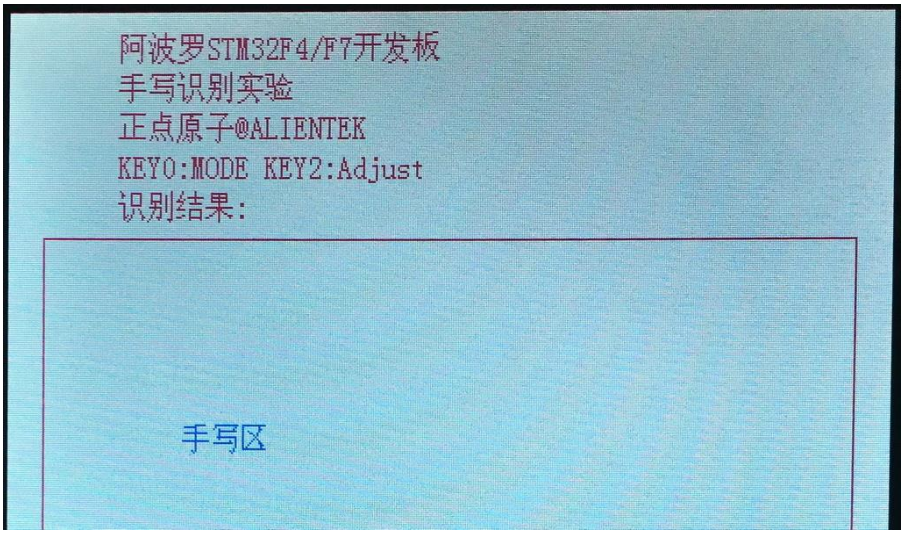


图 58.4.1 手写识别界面

此时，我们在手写区写数字/字母，即可得到识别结果，如图 58.4.2 所示：

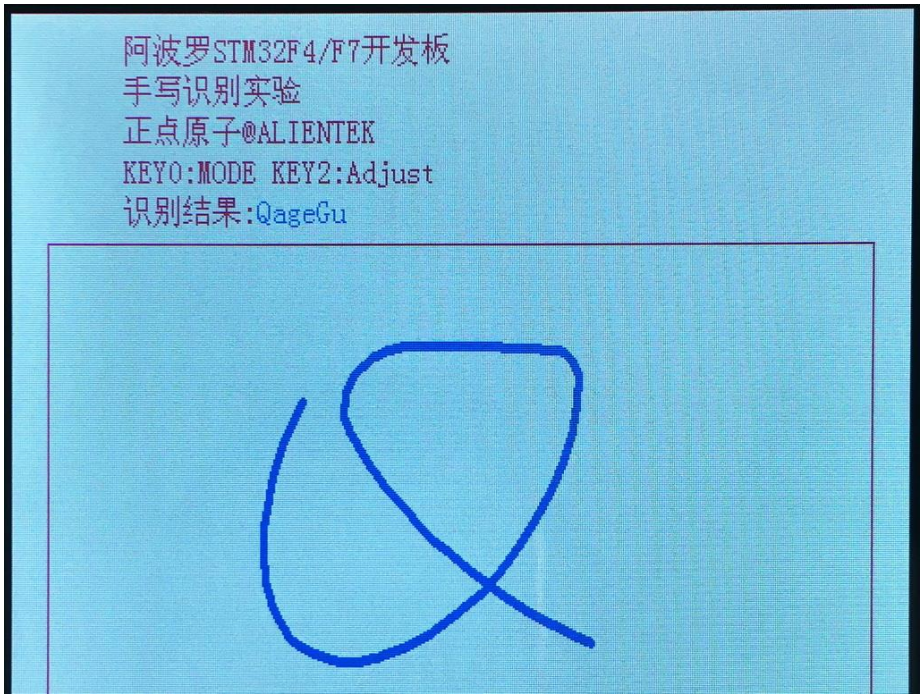


图 58.4.2 手写识别结果

按下 KEY0 可以切换识别模式，同时在识别区提示当前模式。按下 KEY2 可以进行屏幕校准（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。

第五十九章 T9 拼音输入法实验

上一章，我们在 ALIENTEK 阿波罗 STM32 开发板上实现了手写识别输入，但是该方法只能输入数字或者字母，不能输入汉字。本章，我们将给大家介绍如何在 ALIENTEK 阿波罗 STM32 开发板上实现一个简单的 T9 中文拼音输入法。本章分为如下几个部：

59.1 拼音输入法简介
59.2 硬件设计
59.3 软件设计
59.4 下载验证

59.1 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 59.1.1 所示：

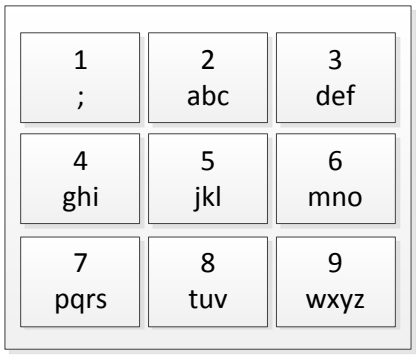


图 59.1.1 手机拼音输入键盘

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      []={"啊阿腌吖锕厄嗄錒呵腌"};
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。

59.2 硬件设计

本章实验功能简介：开机的时候先检测字库，然后显示提示信息和绘制拼音输入表，之后进入等待输入状态。此时用户可以通过屏幕上的拼音输入表输入拼音数字串（通过 DEL 可以实现退格），然后程序自动检测与之对应的拼音和汉字，并显示在屏幕上（同时输出到串口）。如果有多个匹配的拼音，则通过 KEY_UP 和 KEY1 进行选择。按键 KEY0 用于清除一次输入，按键 KEY2 用于触摸屏校准。

本实验用到的资源如下：

1） 指示灯 DS0
2） 四个按键（KEY0/KEY1/KEY2/KEY_UP）
3） 串口
4） LCD 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。

59.3 软件设计

打开本章实验工程可以看到，我们在根目录文件夹下新建了一个 T9INPUT 的文件夹。在该文件夹下面新建了 pyinput.c、pyinput.h 和 pymb.h 三个文件，然后在工程里面新建一个 T9INPUT 的组，将 pyinput.c 加入到该组下面。最后，将 T9INPUT 文件夹加入头文件包含路径。

打开 pyinput.c，代码如下：

```
//拼音输入法
pyinput t9=
{
    get_pymb,
```

```
    0,
};
//比较两个字符串的匹配情况
//返回值:0xff,表示完全匹配.
//            其他,匹配的字符数
u8 str_match(u8*str1,u8*str2)
{
    u8 i=0;
    while(1)
    {
        if(*str1!=*str2)break;                //部分匹配
        if(*str1=='\0'){i=0XFF;break;}    //完全匹配
        i++; str1++; str2++;
    }
    return i;//两个字符串相等
}
//获取匹配的拼音码表
//*strin,输入的字符串,形如:"726"
//**matchlist,输出的匹配表.
//返回值:[7],0,表示完全匹配；1，表示部分匹配（仅在没有完全匹配的时候才会出现）
//            [6:0],完全匹配的时候，表示完全匹配的拼音个数
//                    部分匹配的时候，表示有效匹配的位数
u8 get_matched_pymb(u8 *strin,py_index **matchlist)
{
    py_index *bestmatch=0;//最佳匹配
    u16 pyindex_len=0;
    u16 i=0;
    u8 temp,mcnt=0,bmcnt=0;
    bestmatch=(py_index*)&py_index3[0];//默认为 a 的匹配
    pyindex_len=sizeof(py_index3)/sizeof(py_index3[0]);//得到 py 索引表的大小.
    for(i=0;i<pyindex_len;i++)
    {
        temp=str_match(strin,(u8*)py_index3[i].py_input);
        if(temp)
        {
            if(temp==0XFF)matchlist[mcnt++]=(py_index*)&py_index3[i];
            else if(temp>bmcnt)//找最佳匹配
            {
                bmcnt=temp;
                bestmatch=(py_index*)&py_index3[i];//最好的匹配.
            }
        }
    }
```

```
    if(mcnt==0&&bmcnt)//没有完全匹配的结果,但是有部分匹配的结果
    {
        matchlist[0]=bestmatch;
        mcnt=bmcnt|0X80;        //返回部分匹配的有效位数
    }
    return mcnt;//返回匹配的个数
}
//得到拼音码表.
//str:输入字符串
//返回值:匹配个数.
u8 get_pymb(u8* str)
{
    return get_matched_pymb(str,t9.pymb);
}
//串口测试用
void test_py(u8 *inputstr)
{
    ……代码省略
}
```

这里总共就 4 个函数，其中 get_matched_pymb，是核心，该函数实现将用户输入拼音数字串同拼音索引表里面的各个项对比，找出匹配结果，并将完全匹配的项目存放在 matchlist 里面，同时记录匹配数。对于那些没有完全匹配的输入串，则查找与其最佳匹配的项目，并将匹配的长度返回。函数 test_py（代码省略）用于给 usmart 调用，实现串口测试，该函数可有可无，只是在串口测试的时候才用到，如果不使用的话，可以去掉，本章，我们将其加入 usmart 控制，大家可以通过该函数实现串口调试拼音输入法。

其他两个函数，也比较简单了，我们这里就不细说了。打开 pyinput.h，代码如下：

```
#ifndef __PYINPUT_H
#define __PYINPUT_H
#include "sys.h"
//拼音码表与拼音的对应表
typedef struct
{
  u8 *py_input;//输入的字符串
  u8 *py;          //对应的拼音
  u8 *pymb;       //码表
}py_index;
#define MAX_MATCH_PYMB    10    //最大匹配数
//拼音输入法
typedef struct
{
  u8(*getpymb)(u8 *instr);                //字符串到码表获取函数
  py_index *pymb[MAX_MATCH_PYMB];     //码表存放位置
}pyinput;
```

```
extern pyinput t9;
u8 str_match(u8*str1,u8*str2);
u8 get_matched_pymb(u8 *strin,py_index **matchlist);
u8 get_pymb(u8* str);
void test_py(u8 *inputstr);
#endif
```

pymb.h 里面完全就是我们前面介绍的拼音码表，该文件很大，里面存储了所有我们可以输入的汉字，此部分代码就不贴出来了，请大家参考光盘本例程的源码。

最后，我们看看主函数代码：

```
const u8* kbd_tbl[9]={"←","2","3","4","5","6","7","8","9",};//数字表
const u8* kbs_tbl[9]={"DEL","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz",};//字符表
u16 kbdxsize; //虚拟键盘按键宽度
u16 kbdysize; //虚拟键盘按键高度
//加载键盘界面
//x,y:界面起始坐标
void py_load_ui(u16 x,u16 y)
{
    ……//此处省略部分代码
}
//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,result_num,cur_index,key,inputstr[7];//最大输入 6 个字符+结束符
```

```
    u8 inputlen;                    //输入长度

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    usmart_dev.init(108);           //初始化 USMART
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    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 SDRAM 内存池
    my_mem_init(SRAMDTCM);          //初始化内部 DTCM 内存池

RESTART:
    POINT_COLOR=RED;
    while(font_init())              //检查字库
    {
        LCD_ShowString(60,50,200,16,16,"Font Error!");
        delay_ms(200);
        LCD_Fill(60,50,240,66,WHITE);//清除显示
    }
    ……//此处省略部分代码
    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==KEY2_PRES&&tp_dev.touchtype==0)//KEY2 按下,且是电阻屏
    {
        tp_dev.adjust();
        LCD_Clear(WHITE);
        goto RESTART;
```

```
            }
            if(result_num) //存在匹配的结果
            {
                switch(key)
                {
                    case WKUP_PRES://上翻
                        if(cur_index<result_num)cur_index++;
                        else cur_index=1;
                        py_show_result(cur_index);  //显示第 cur_index 的匹配结果
                        break;
                    case KEY1_PRES://下翻
                        if(cur_index>1)cur_index--;
                        else cur_index=result_num;
                        py_show_result(cur_index);  //显示第 cur_index 的匹配结果
                        break;
                    case KEY0_PRES://清除输入
                        LCD_Fill(30+40,145,lcddev.width-1,145+48,WHITE);//清除之前显示
                        goto RESTART;
                }
            }
            if(i==30)
            {
                i=0;
                LED0_Toggle;
            }
        }
    }
```

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 59.2 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。

59.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32 开发板上，得到，如图 59.4.1 所示：

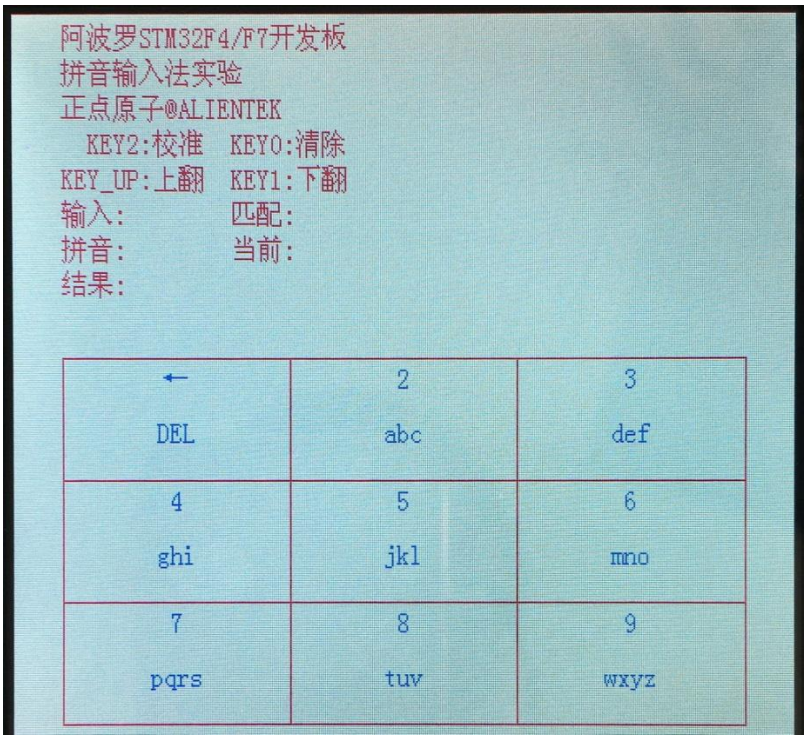


图 59.4.1 汉字输入法界面

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 59.4.2 所示：

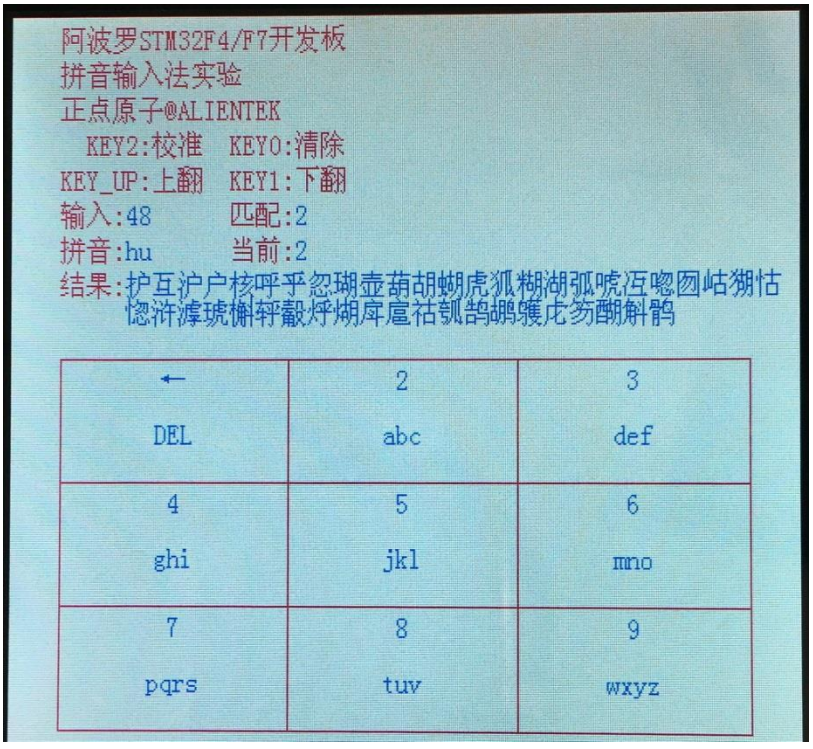


图 59.4.2 实现拼音输入

如果发现输入错了，可以通过屏幕上的 DEL 按钮，来退格。如果有多个匹配的情况（匹配值大于 1），则可以通过 KEY_UP 和 KEY1 来选择拼音。通过按下 KEY0，可以清除当前输入，通过按下 KEY2，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 59.4.3 所示：

图 59.4.3 USMART 调试 T9 拼音输入法

第六十章 串口 IAP 实验

IAP，即在应用编程。很多单片机都支持这个功能，STM32F767 也不例外。在之前的 FLASH 模拟 EEPROM 实验里面，我们学习了 STM32F767 的 FLASH 自编程，本章我们将结合 FLASH 自编程的知识，通过 STM32F767 的串口实现一个简单的 IAP 功能本章分为如下几个部：

60.1 IAP 简介
60.2 硬件设计
60.3 软件设计
60.4 下载验证

60.1 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 STM32F767 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。

STM32F7 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，一个用于内部 SRAM 运行。

STM32F7 的 FLASH 可以映射到两个地址：0X0020 0000 或 0X0800 0000，我们仅以 0X0800 0000 为例进行介绍。STM32F7 正常的程序运行流程，如图 60.1.1 所示：

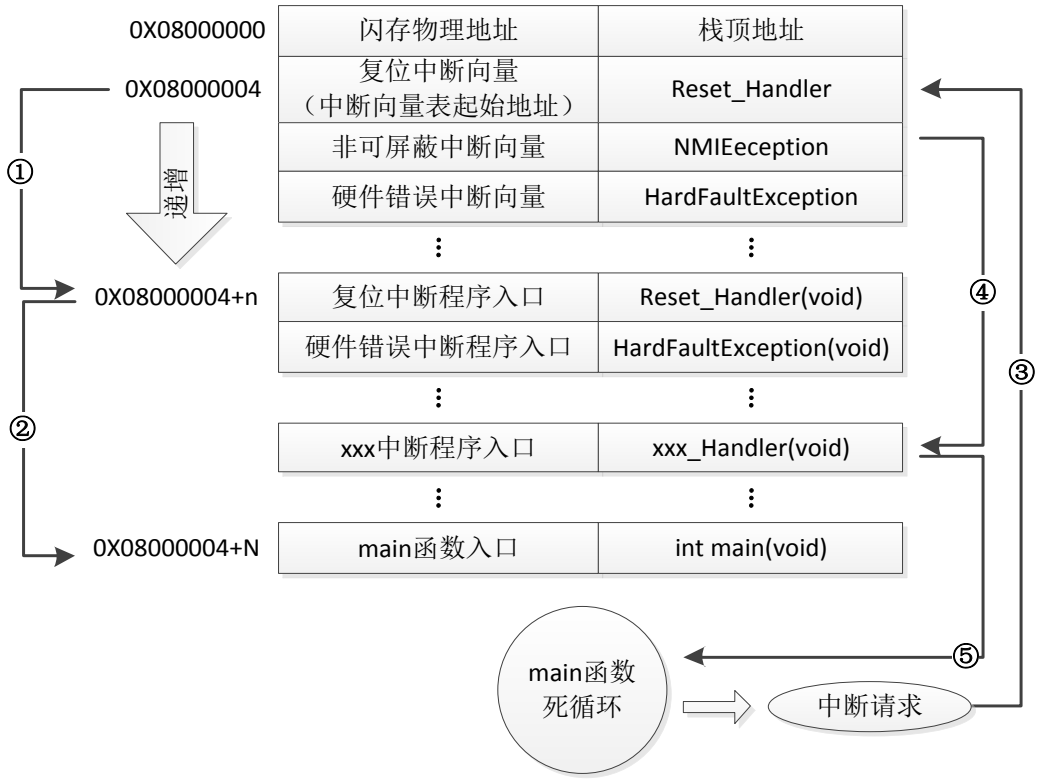


图 60.1.1 STM32F767 正常运行流程图

STM32F7 的 FLASH 可以映射到两个地址，本章以映射到 0X0800 0000 为例，一般情况下，程序文件就从此地址开始写入。此外 STM32F767 是基于 Cortex-M7 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，STM32F767 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 60.1.1 中，STM32F767 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（发生了中断），此时 STM32F767 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 60.1.2 所示：

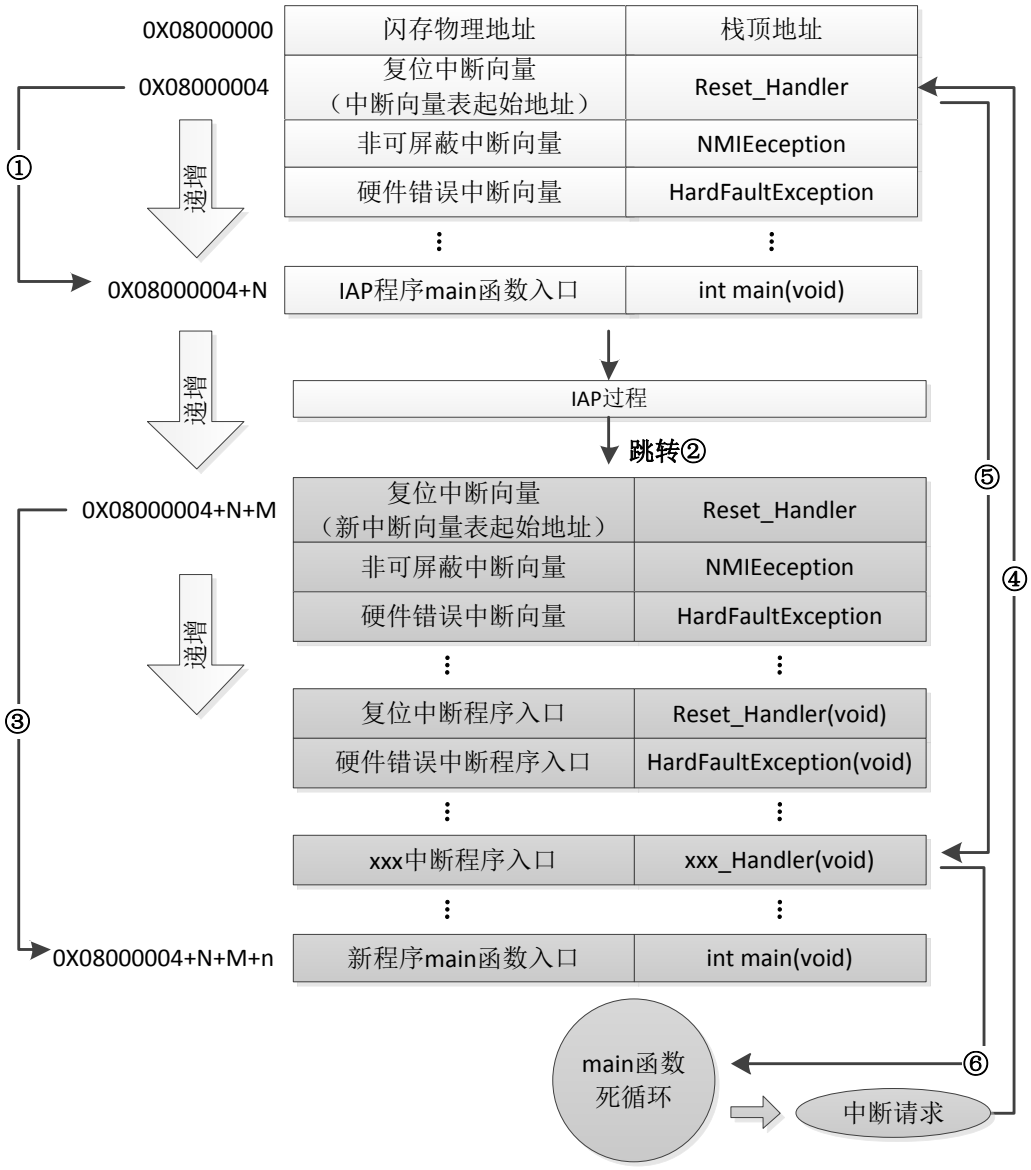


图 60.1.2 加入 IAP 之后程序运行流程图

在图 60.1.2 所示流程中，STM32F767 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 60.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32F767 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 STM32F767 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；

3） 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 60.1.2 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 60.1.3 所示：

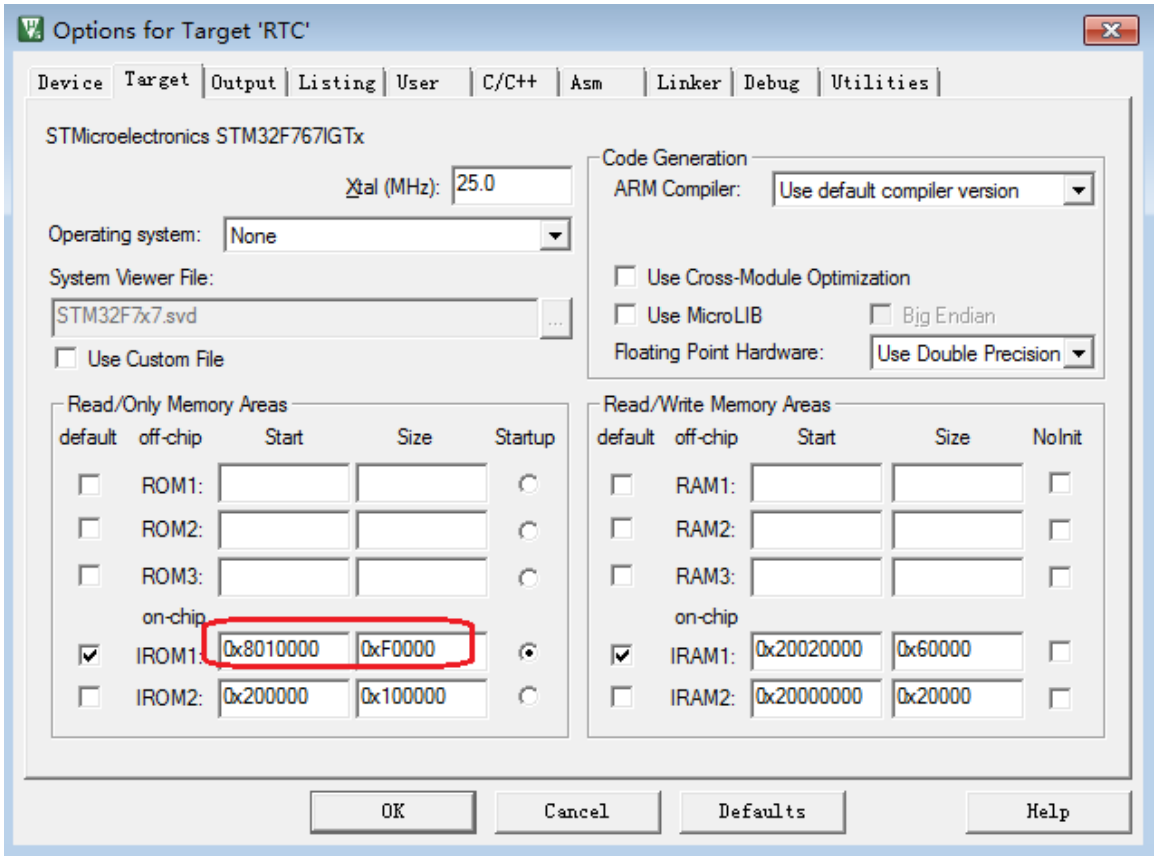


图 60.1.3 FLASH APP Target 选项卡设置

默认的条件下，图中 IROM1 的起始地址（Start）一般为 0X08000000，大小（Size）为 0X100000，即从 0X08000000 开始的 1024K 空间为我们的程序存储区。而图中，我们设置起始地址（Start）为 0X08010000，即偏移量为 0X10000（64K 字节），因而，留给 APP 用的 FLASH 空间（Size）只有 0X100000-0X10000=0XF0000（960K 字节）大小了。设置好 Start 和 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 63K 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 60.1.4 所示：

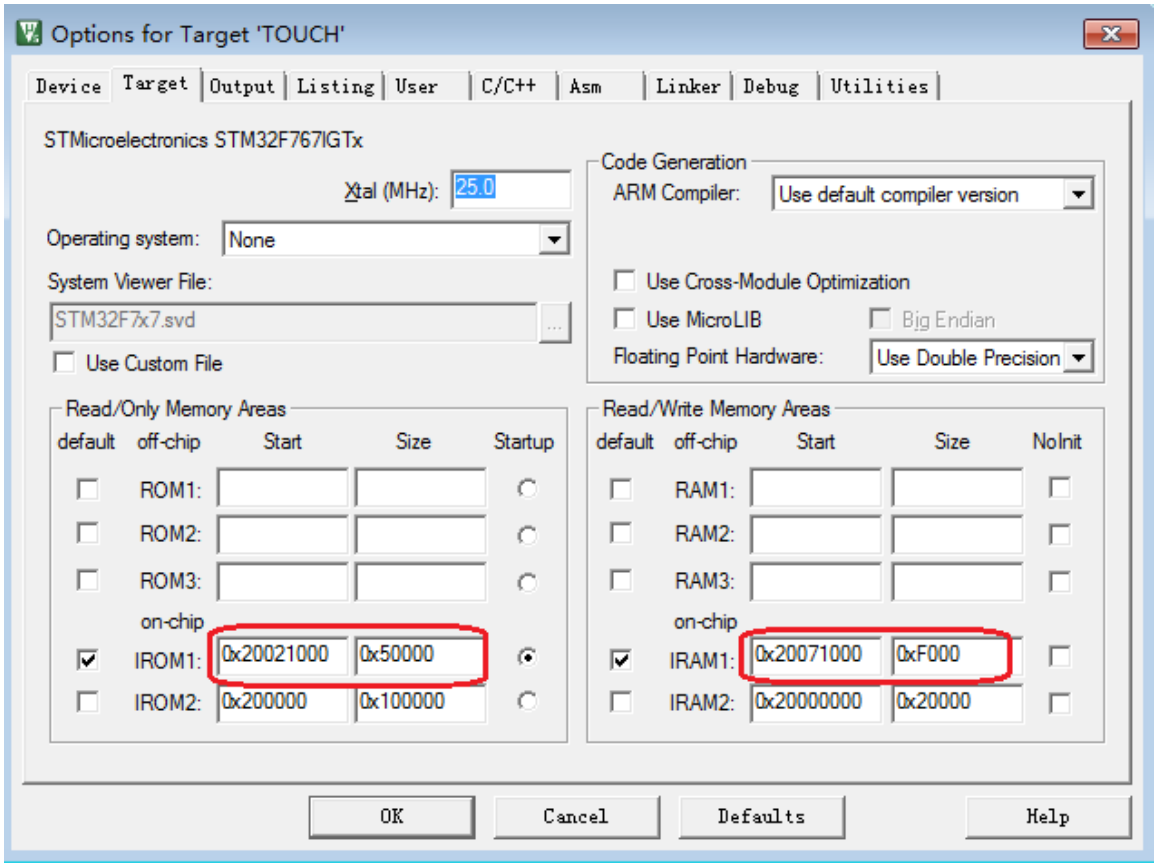


图 60.1.4 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址（Start）定义为：0X20021000，大小为 0X50000（320K 字节），即从地址 0X20020000 偏移 0X1000 开始，存放 APP 代码。因为整个 STM32F767IGT6 的 SRAM 大小（不算 DTCM）为 384K 字节，所以 IRAM1（SRAM）的起始地址变为 0X20071000，大小只有 0XF000（60K 字节）。这样，整个 STM32F767IGT6 的 SRAM（不含 DTCM）分配情况为：最开始的 4K 给 Bootloader 程序使用，随后的 320K 存放 APP 程序，最后 60K，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 = RAMDTCM_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 = SRAM1_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\ARMCC\bin 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：D:\tools\MDK5.2\ARM\ARMCC\bin\fromelf.exe --bin -o ..\OBJ\RTC.bin ..\OBJ\RTC.axf ，如图 60.1.5 所示：

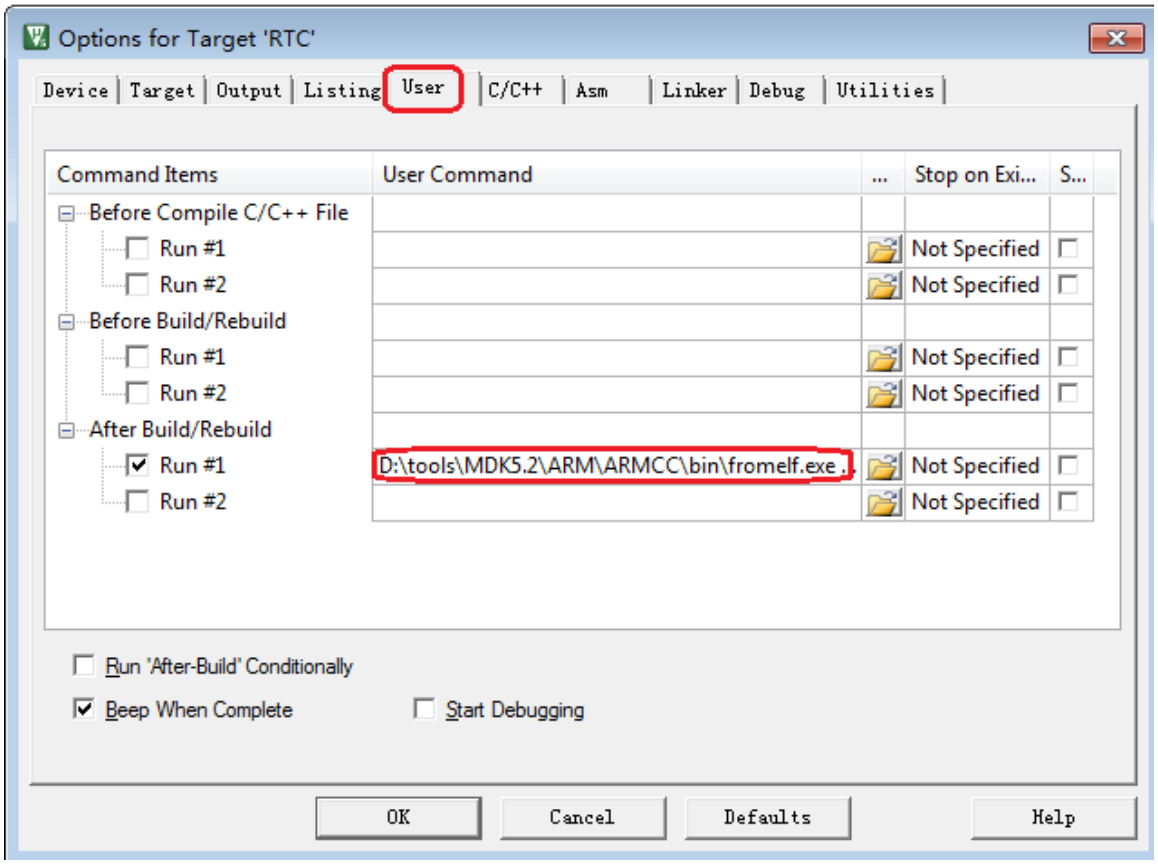


图 60.1.5 MDK 生成.bin 文件设置方法

通过这一步设置，我们就可以在 MDK 编译成功之后，调用 fromelf.exe（注意，我的 MDK 是安装在 D:\tools\MDK5.2 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。

60.2 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 STM32 的内部 FLASH，之后再按 KEY2 既可以执行这个 FLASH APP 程序。通过 KEY1 按键，可以手动清除串口接收到的 APP 程序。DS0 用于指示程序运行状态。

本实验用到的资源如下：

1） 指示灯 DS0

2） 四个按键（KEY0/KEY1/KEY2/KEY_UP）

3） 串口

4） LCD 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。

60.3 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（在第二十章二介绍）来做为 FLASH APP 程序（起始地址为 0X08010000），选择触摸屏实验（在第三十六章介绍）来做 SRAM APP 程序（起始地址为 0X20021000）。Bootloader 则是通过 TFTLCD 显示实验（在第而是章介绍）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 60.1 节的介绍，自行理解。本章软件设计仅针对 Bootloader 程序。

复制第二十章的工程（即实验 14），作为本章的工程模版（命名为：IAP Bootloader V1.0），并复制第四十二章实验（FLASH 模拟 EEPROM 实验）的 STMFLASH 文件夹到本工程的 HARDWARE 文件夹下，打开本实验工程，并将 STMFLASH 文件夹内的 stmflash.c 加入到 HARDWARE 组下，同时将 STMFLASH 加入头文件包含路径。

在 HARDWARE 文件夹所在的文件夹下新建一个 IAP 的文件夹，并在该文件夹下新建 iap.c 和 iap.h 两个文件。然后在工程里面新建一个 IAP 的组，将 iap.c 加入到该组下面。最后，将 IAP 文件夹加入头文件包含路径。

打开 iap.c，输入如下代码：

```
iapfun jump2app;
u32 iapbuf[512];    //2K 字节缓存
//appxaddr:应用程序的起始地址
//appbuf:应用程序 CODE.
//appsize:应用程序大小(字节).
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 appsize)
{
    u32 t;
    u16 i=0;
    u32 temp;
    u32 fwaddr=appxaddr;//当前写入的地址
    u8 *dfu=appbuf;
    for(t=0;t<appsize;t+=4)
    {
        temp=(u32)dfu[3]<<24;
        temp|=(u32)dfu[2]<<16;
        temp|=(u32)dfu[1]<<8;
        temp|=(u32)dfu[0];
        dfu+=4;//偏移 4 个字节
        iapbuf[i++]=temp;
        if(i==512)
        {
            i=0;
            STMFLASH_Write(fwaddr,iapbuf,512);
            fwaddr+=2048;//偏移 2048    512*4=2048
        }
    }
    if(i)STMFLASH_Write(fwaddr,iapbuf,i);//将最后的一些内容字节写进去.
}

//跳转到应用程序段
//appxaddr:用户代码起始地址.
void iap_load_app(u32 appxaddr)
{
    if(((*(vu32*)appxaddr)&0x2FF00000)==0x20000000)      //检查栈顶地址是否合法.
    {
        jump2app=(iapfun)*(vu32*)(appxaddr+4);
                                    //用户代码区第二个字为程序开始地址(复位地址)
        MSR_MSP(*(vu32*)appxaddr);  //初始化 APP 堆栈指针(用户代码区的第一个字用
                                    //于存放栈顶地址)
        jump2app();                                     //跳转到 APP.
    }
}
```

该文件总共只有 2 个函数，其中，iap_write_appbin 函数用于将存放在串口接收 buf 里面的 APP 程序写入到 FLASH。iap_load_app 函数，则用于跳转到 APP 程序运行，其参数 appxaddr 为 APP 程序的起始地址，程序先判断栈顶地址是否合法，在得到合法的栈顶地址后，通过 MSR_MSP 函数（该函数在 sys.c 文件）设置栈顶地址，最后通过一个虚拟的函数（jump2app）跳转到 APP 程序执行代码，实现 IAP→APP 的跳转。

保存 iap.c，打开 iap.h 输入如下代码：

```
#ifndef __IAP_H__
#define __IAP_H__
#include "sys.h"
typedef  void (*iapfun)(void);                    //定义一个函数类型的参数.
#define FLASH_APP1_ADDR          0x08010000
//第一个应用程序起始地址(存放在 FLASH)
//保留 0X08000000~0X0800FFFF 的空间为 Bootloader 使用(共 64KB)
void iap_load_app(u32 appxaddr); //跳转到 APP 程序执行
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 applen);//在指定地址开始,写入 bin
#endif
```

这部分代码比较简单，保存 iap.h。本章，我们是通过串口接收 APP 程序的，我们将 usart.c 和 usart.h 做了稍微修改，在 usart.h 中，我们定义 USART_REC_LEN 为 360K 字节，也就是串口最大一次可以接收 360K 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(0X20021000)));
//接收缓冲,最大 USART_REC_LEN 个字节,起始地址为 0X20021000.
//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
//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_CNT<USART_REC_LEN)
        {
```

```
                USART_RX_BUF[USART_RX_CNT]=Res;
                USART_RX_CNT++;
            }
        }
        HAL_UART_IRQHandler(&UART1_Handler);
    #if SYSTEM_SUPPORT_OS       //使用 OS
        OSIntExit();
    #endif
    }
```

这里，我们指定 USART_RX_BUF 的地址是从 0X20021000 开始，该地址也就是 SRAM APP 程序的起始地址！然后在 USART1_IRQHandler 函数里面，将串口发送过来的数据，全部接收到 USART_RX_BUF，并通过 USART_RX_CNT 计数。代码比较简单，我们就不多说了。

改完 usart.c 和 usart.h 之后，我们在 main.c 修改 main 函数如下：

```
int main(void)
{
    u8 t;
    u8 key;
    u32 oldcount=0;                 //老的串口接收数据值
    u32 applenth=0;                 //接收到的 app 代码长度
    u8 clearflag=0;

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    …//此处省略部分代码
    LCD_ShowString(30,130,200,16,16,"KEY_UP:Copy APP2FLASH");
    LCD_ShowString(30,150,200,16,16,"KEY1:Erase SRAM APP");
    LCD_ShowString(30,170,200,16,16,"KEY0:Run SRAM APP");
    LCD_ShowString(30,190,200,16,16,"KEY2:Run FLASH APP");
    POINT_COLOR=BLUE;
    //显示提示信息
    POINT_COLOR=BLUE;//设置字体为蓝色
    while(1)
    {
        if(USART_RX_CNT)
        {
            if(oldcount==USART_RX_CNT)
                        //新周期内,没有收到任何数据,认为本次数据接收完成.
            {
                applenth=USART_RX_CNT;
                oldcount=0;
                USART_RX_CNT=0;
                printf("用户程序接收完成!\r\n");
```

```
            printf("代码长度:%dBytes\r\n",applenth);
        }else oldcount=USART_RX_CNT;
    }
    t++;
    delay_ms(10);
    if(t==30)
    {
        LED0_Toggle;
        t=0;
        if(clearflag)
        {
            clearflag--;
            if(clearflag==0)LCD_Fill(30,210,240,210+16,WHITE);//清除显示
        }
    }
    key=KEY_Scan(0);
    if(key==WKUP_PRES) //WK_UP 按键按下
    {
        if(applenth)
        {
            printf("开始更新固件...\r\n");
            LCD_ShowString(30,210,200,16,16,"Copying APP2FLASH...");
            if(((*(vu32*)(0x20021000+4))&0xFF000000)==0x08000000)
                                        //判断是否为 0X08XXXXXX.
            {
                iap_write_appbin(FLASH_APP1_ADDR,USART_RX_BUF,
                                                applenth);//更新 FLASH 代码
                LCD_ShowString(30,210,200,16,16,"Copy APP Successed!!");
                printf("固件更新完成!\r\n");
            }else
            {
                LCD_ShowString(30,210,200,16,16,"Illegal FLASH APP!    ");
                printf("非 FLASH 应用程序!\r\n");
            }
        }else
        {
            printf("没有可以更新的固件!\r\n");
            LCD_ShowString(30,210,200,16,16,"No APP!");
        }
        clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示

    }
    if(key==KEY1_PRES)  //KEY1 按下
```

```
{
    if(applenth)
    {

        printf("固件清除完成!\r\n");
        LCD_ShowString(30,210,200,16,16,"APP Erase Successed!");
        applenth=0;
    }else
    {
        printf("没有可以清除的固件!\r\n");
        LCD_ShowString(30,210,200,16,16,"No APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示

}
if(key==KEY2_PRES)  //KEY2 按下
{
    printf("开始执行 FLASH 用户代码!!\r\n");
    if(((*(vu32*)(FLASH_APP1_ADDR+4))&0xFF000000)==0x08000000)
                                                //判断是否为 0X08XXXXXX.
    {
        iap_load_app(FLASH_APP1_ADDR);//执行 FLASH APP 代码
    }else
    {
        printf("非 FLASH 应用程序,无法执行!\r\n");
        LCD_ShowString(30,210,200,16,16,"Illegal FLASH APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}
if(key==KEY0_PRES)  //KEY0 按下
{
    printf("开始执行 SRAM 用户代码!!\r\n");
    if(((*(vu32*)(0x20021000+4))&0xFF000000)==0x20000000)
                                            //判断是否为 0X20XXXXXX.
    {
        iap_load_app(0x20021000);//SRAM 地址
    }else
    {
        printf("非 SRAM 应用程序,无法执行!\r\n");
        LCD_ShowString(30,210,200,16,16,"Illegal SRAM APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}
```

```
        }
    }
```

该段代码，实现了串口数据处理，以及 IAP 更新和跳转等各项操作。Bootloader 程序就设计完成了，但是一般要求 bootloader 程序越小越好（给 APP 省空间嘛），实际应用时，可以尽量精简代码来得到最小的 IAP。本章例程我们仅作演示用，所以不对代码做任何精简，最后得到工程截图如图 60.3.1 所示：

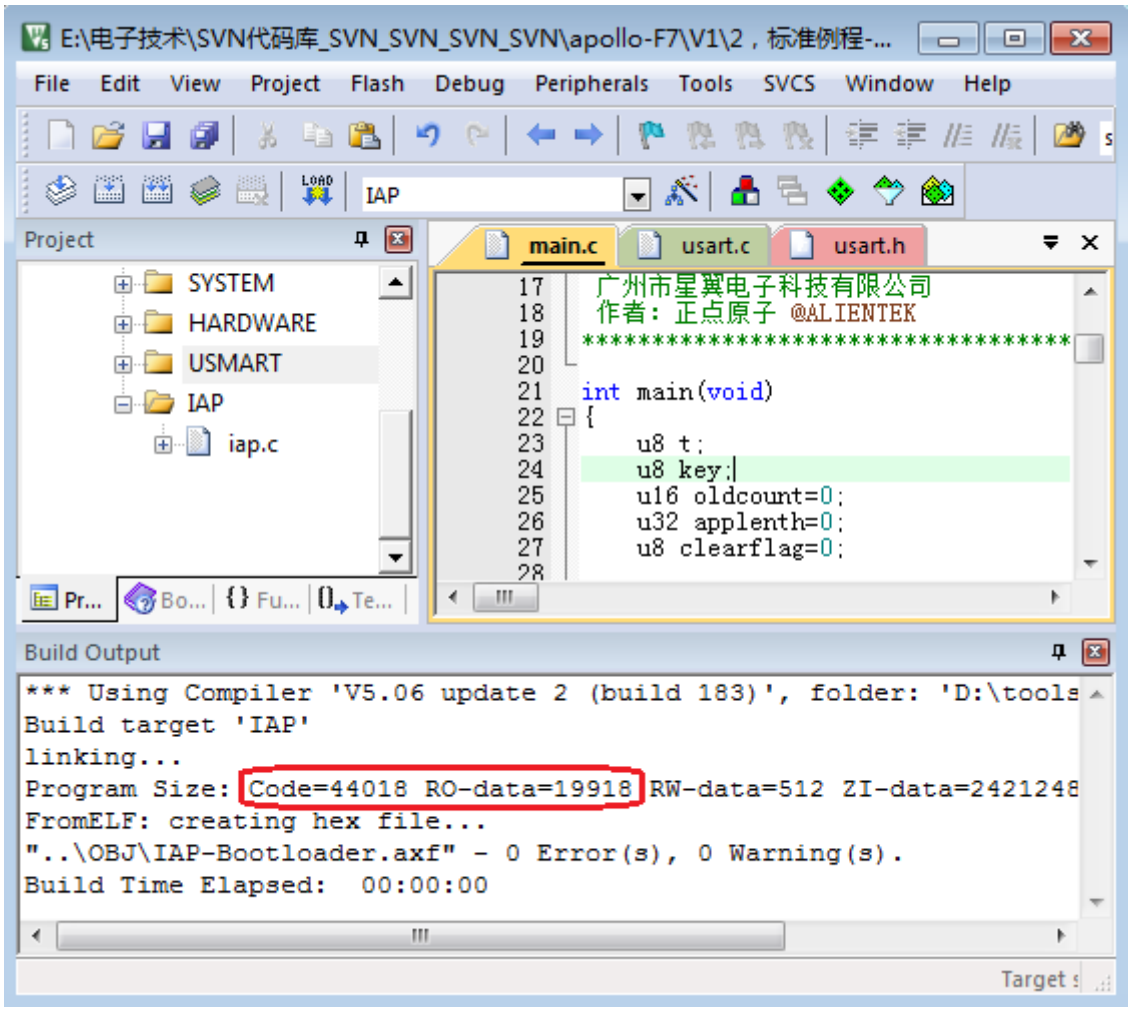


图 60.3.1 Bootloader 工程截图

从上图可以看出，Bootloader 大小为 42K 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 60.1 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 0X20021000。

60.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 阿波罗 STM32F7 开发板上，得到，如图 60.4.1 所示：

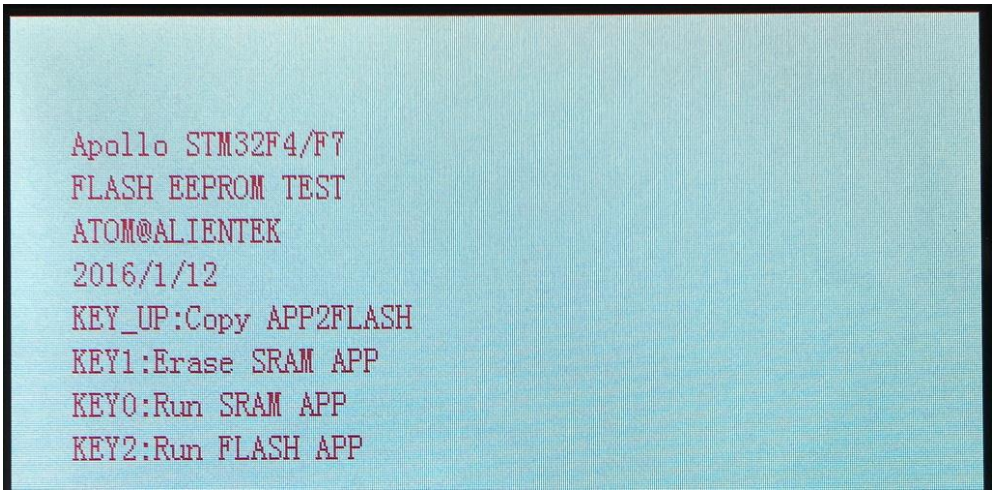


图 60.4.1 IAP 程序界面

此时，我们可以通过串口，发送 FLASH APP 或者 SRAM APP 到阿波罗 STM32F7 开发板，如图 60.4.2 所示：

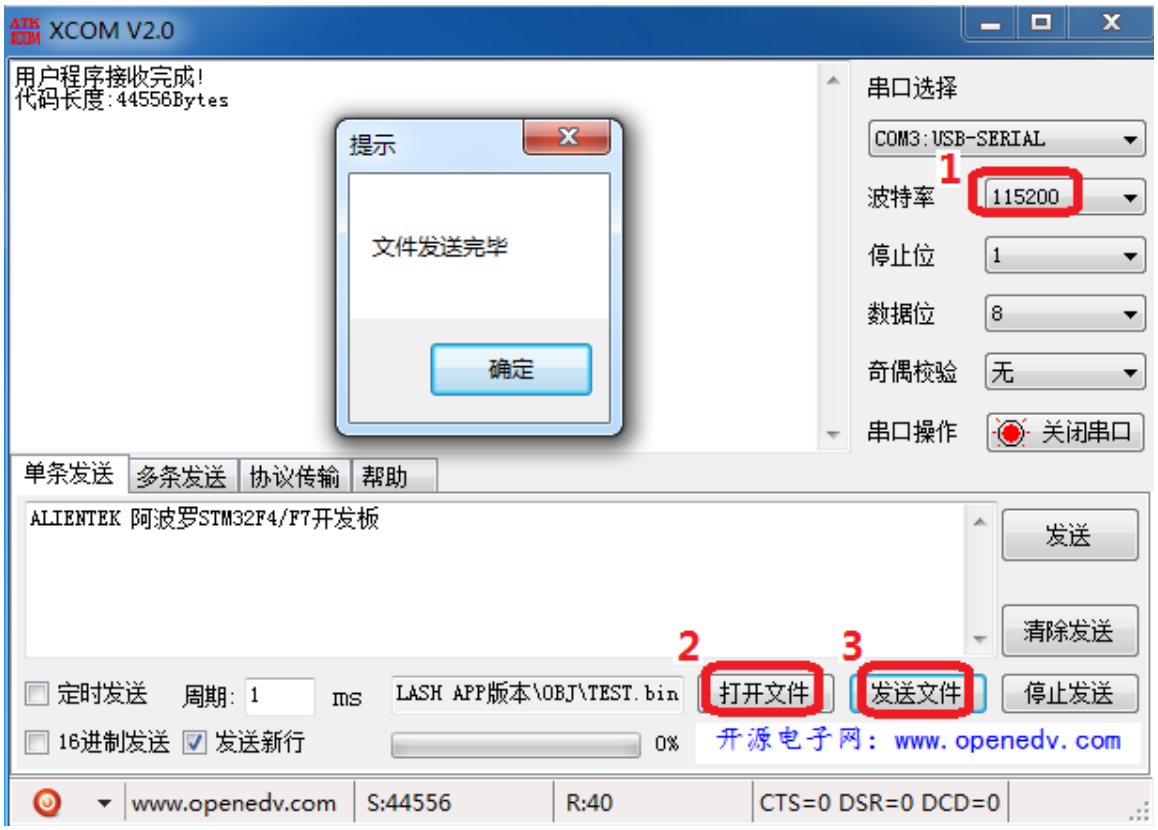


图 60.4.2 串口发送 APP 程序界面

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 文件发送给阿波罗 STM32 开发板，发送完成后，XCOM 会提示文件发送完毕。

开发板在收到 APP 程序之后，我们就可以通过 KEY0/KEY2 运行这个 APP 程序了（如果是 FLASH APP，则先需要通过 KEY_UP 将其存入对应 FLASH 区域）。

第六十一章 USB 读卡器(Slave)实验

STM32F7 系列芯片都自带了 USB OTG FS 和 USB OTG HS（HS 需要外扩高速 PHY 芯片实现，速度可达 480Mbps），支持 USB Host 和 USB Device，阿波罗 STM32F7 开发板没有外扩高速 PHY 芯片，仅支持 USB OTG FS（FS，即全速，12Mbps），所有 USB 相关例程，均使用 USB OTG FS 实现。

本章，我们将向大家介绍如何利用 USB OTG FS 在 ALIENTEK 阿波罗 STM32F7 开发板实现一个 USB 读卡器。本章分为如下几个部分：

61.1 USB 简介
61.2 硬件设计
61.3 软件设计
61.4 下载验证

61.1 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。STM32F767 自带的 USB 符合 USB2.0 规范。

标准 USB 共四根线组成,除 VCC/GND 外，另外为 D+和 D-，这两根数据线采用的是差分电压的方式进行数据传输的。在 USB 主机上，D-和 D+都是接了 15K 的电阻到地的，所以在没有设备接入的时候，D+、D-均是低电平。而在 USB 设备中，如果是高速设备，则会在 D+上接一个 1.5K 的电阻到 VCC，而如果是低速设备，则会在 D-上接一个 1.5K 的电阻到 VCC。这样当设备接入主机的时候，主机就可以判断是否有设备接入，并能判断设备是高速设备还是低速设备。接下来，我们简单介绍一下 STM32 的 USB 控制器。

STM32F767 系列芯片自带有 USB OTG FS（全速）和 USB OTG HS（高速），其中 HS 需要外扩高速 PHY 芯片实现，我们这里不做介绍。

STM32F767 的 USB OTG FS 是一款双角色设备 (DRD) 控制器，同时支持从机功能和主机功能，完全符合 USB 2.0 规范的 On-The-Go 补充标准。此外，该控制器也可配置为“仅主机”模式或“仅从机” 模式，完全符合 USB 2.0 规范。在主机模式下，OTG FS 支持全速（FS，12 Mb/s）和低速（LS，1.5 Mb/s）收发器，而从机模式下则仅支持全速（FS，12 Mb/s）收发器。OTG FS 同时支持 HNP 和 SRP。

STM32F767 的 USB OTG FS 主要特性可分为三类：通用特性、主机模式特性和从机模式特性。

1，通用特性

- 经 USB-IF 认证，符合通用串行总线规范第 2.0 版
- 集成全速 PHY，且完全支持定义在标准规范 OTG 补充第 1.3 版中的 OTG 协议
  1，支持 A-B 器件识别（ID 线）
  2，支持主机协商协议(HNP)和会话请求协议(SRP)
  3，允许主机关闭 VBUS 以在 OTG 应用中节省电池电量
  4，支持通过内部比较器对 VBUS 电平采取监控

5，支持主机到从机的角色动态切换

- 可通过软件配置为以下角色：
  1， 具有 SRP 功能的 USB FS 从机（B 器件）
  2， 具有 SRP 功能的 USB FS/LS 主机（A 器件）
  3，USB On-The-Go 全速双角色设备
- 支持 FS SOF 和 LS Keep-alive 令牌
  1，SOF 脉冲可通过 PAD 输出
  2，SOF 脉冲从内部连接到定时器 2 (TIM2)
  3，可配置的帧周期
  3， 可配置的帧结束中断
- 具有省电功能，例如在 USB 挂起期间停止系统、关闭数字模块时钟、对 PHY 和 DFIFO 电源加以管理
- 具有采用高级 FIFO 控制的 1.25 KB 专用 RAM
  1，可将 RAM 空间划分为不同 FIFO，以便灵活有效地使用 RAM
  2，每个 FIFO 可存储多个数据包
  3，动态分配存储区
  4，FIFO 大小可配置为非 2 的幂次方值，以便连续使用存储单元
- 一帧之内可以无需要应用程序干预，以达到最大 USB 带宽

2，主机（Host）模式特性

- 通过外部电荷泵生成 VBUS 电压。
- 多达 12 个（FS）/16 个（HS）主机通道（管道）：每个通道都可以动态实现重新配置，可支持任何类型的 USB 传输。
- 内置硬件调度器可：
  1，在周期性硬件队列中存储多达 12(FS)/16(HS)个中断加同步传输请求
  2，在非周期性硬件队列中存储多达 12(FS)/16(HS)个控制加批量传输请求
- 管理一个共享 RX FIFO、一个周期性 TX FIFO 和一个非周期性 TX FIFO，以有效使用 USB 数据 RAM。

3，从机（Slave/Device）模式特性

- 1 个双向控制端点 0
- 5(FS)/7(HS)个 IN 端点 (EP)，可配置为支持批量传输、中断传输或同步传输
- 5(FS)/7(HS)个 OUT 端点(EP)，可配置为支持批量传输、中断传输或同步传输
- 管理一个共享 Rx FIFO 和一个 Tx-OUT FIFO，以高效使用 USB 数据 RAM
- 管理多达 6(FS)/8(HS)个专用 Tx-IN FIFO（分别用于每个使能的 IN EP），降低应用程序负荷支持软断开功能。

STM32F767 USB OTG FS 框图如图 61.1.1 所示：

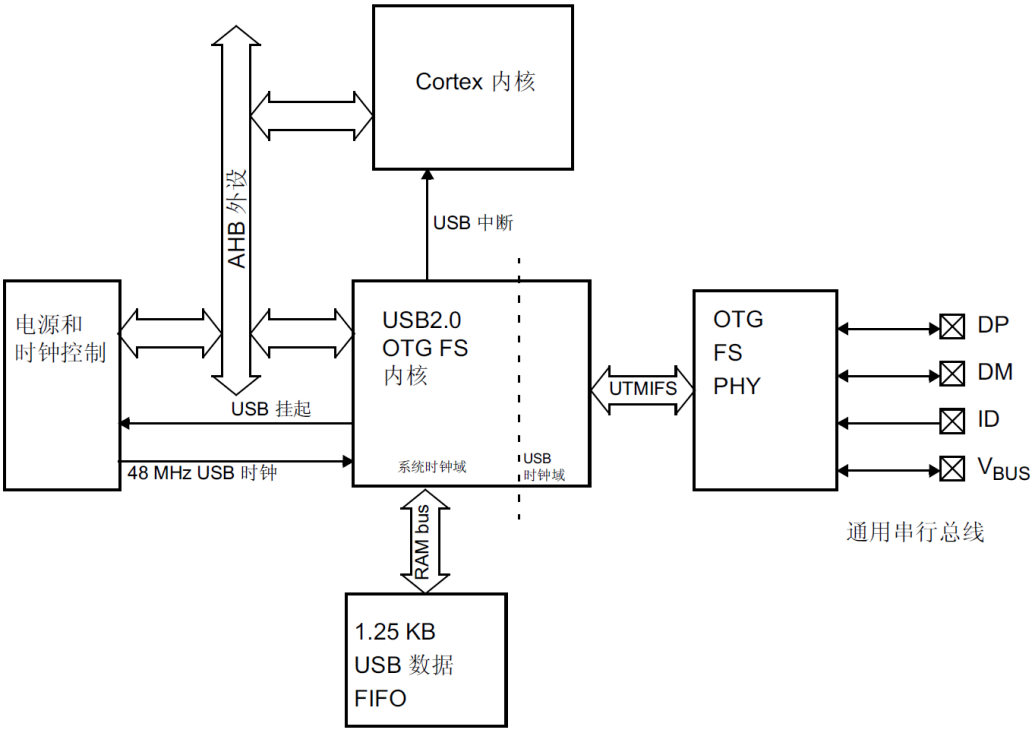


图 61.1.1 USB OTG 框图

对于 USB OTG FS 功能模块，STM32F767 通过 AHB 总线访问（AHB 频率必须大于 14.2Mhz），其中 48Mhz 的 USB 时钟，是来自时钟树图里面的 PLL48CLK（和 SDMMC、RNG 共用）。

STM32F7 的主频一般为 216Mhz，而 USB 需要 48Mhz 的时钟，由主 PLL 经过 Q 分频得到：PLL48CLK=Fvco/PLLQ，Fvco 为 432，设置 PLLQ=9 就可以得到 48Mhz 的 PLL48CLK 频率。

STM32F767 USB OTG FS 的其他介绍，请大家参考《STM32F7 中文参考手册》第 37 章内容，我们这里就不再详细介绍了。

要正常使用 STM32F767 的 USB，就得编写 USB 驱动，而整个 USB 通信的详细过程是很复杂的，本书篇幅有限，不可能在这里详细介绍，有兴趣的朋友可以去看看电脑圈圈的《圈圈教你玩 USB》这本书，该书对 USB 通信有详细讲解。如果要我们自己编写 USB 驱动，那是一件相当困难的事情，尤其对于从没了解过 USB 的人来说，基本上不花个一两年时间学习，是没法搞定的。不过，ST 提供了我们一个完整的 USB OTG 驱动库（包括主机和设备），通过这个库，我们可以很方便的实现我们所要的功能，而不需要详细了解 USB 的整个驱动，大大缩短了我们的开发时间和精力。

STM32F7 的 USB 例程全部是以 HAL 库的形式提供，使用起来比较复杂，为了更好的与 STM32F1/F2/F4 兼容，我们通过修改 STM32F1/F2/F4 的 USB OTG 库来支持 STM32F7。

ST 提供的 STM32F1/F2/F4 USB OTG 库在：http://www.stmcu.org/document/list/index/category-523 这里可以下载到（STSW-STM32046）。不过，我们已经帮大家下载到开发板光盘：8，STM32 参考资料→STM32 USB 学习资料，文件名：stm32_f105-07_f2_f4_usb-host-device_lib.zip。该库包含了 STM32F1/F2/F4 的 USB 主机（Host）和从机（Device）驱动库，并提供了 14 个例程供我们参考，如图 61.1.2 所示：

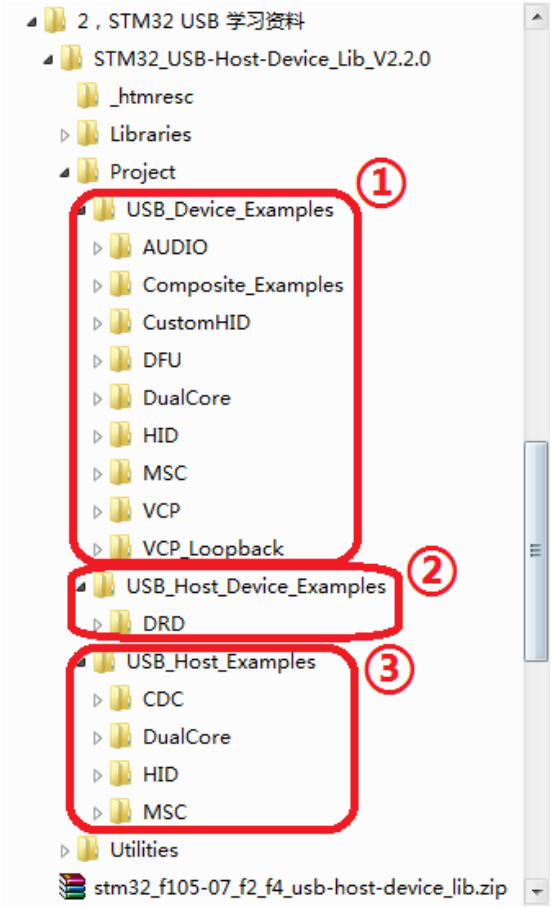


图 61.1.2 ST 提供的 USB OTG 例程

如图 61.1.2 所示，ST 提供了 3 类例程：①即设备类（Device，即 Slave）、②主从一体类（Host_Device）和③主机类（Host），总共 14 个例程。整个 USB OTG 库还有一个说明文档：CD00289278.pdf(在光盘有提供)，即 UM1021，该文档详细介绍了 USB OTG 库的各个组成部分以及所提供的例程使用方法，有兴趣学习 USB 的朋友，这个文档是必须仔细看的。

这 14 个例程，虽然都是基于官方 STM32F1/F2/F4 EVAL 板，但是很容易移植到我们的阿波罗 STM32F767 开发板上，稍作修改就可以支持 STM32F7 系列。本章我们就是移植：STM32_USB-Host-Device_Lib_V2.2.0\Project\USB_Device_Examples\MSC 这个例程，以实现 USB 读卡器功能。

61.2 硬件设计

本章实验功能简介：开机的时候先检测 SD 卡、SPI FLASH 和 NAND 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三个可移动磁盘。我们用 DS1 来指示 USB 正在读写，并在液晶上显示出来，同样，我们还是用 DS0 来指示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1
2） 串口
3） LCD 模块
4） SD 卡
5） SPI FLASH
6） NAND FLASH

7） USB SLAVE 接口

前面 6 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 阿波罗 STM32 开发板采用的是 5PIN 的 MiniUSB 接头，用来和电脑的 USB 相连接，连接电路如图 61.2.1 所示：

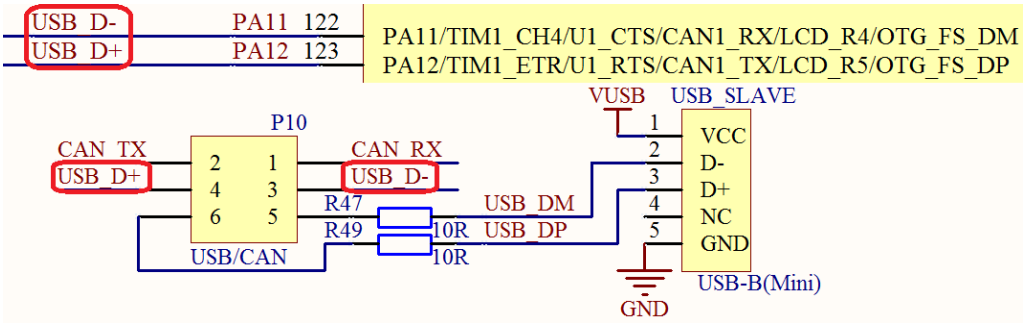


图 61.2.1 MiniUSB 接口与 STM32 的连接电路图

从上图可以看出，USB 座没有直接连接到 STM32F767 上面，而是通过 P10 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 61.2.2 所示：

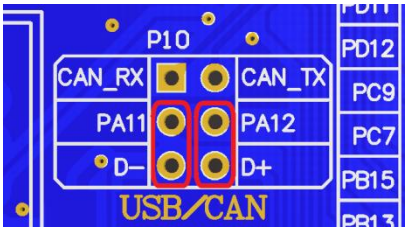


图 61.2.2 硬件连接示意图

不过这个 MiniUSB 座和 USB-A 座（USB_HOST）是共用 D+和 D-的，所以他们不能同时使用。这个在使用的时候，要特别注意！！本实验测试时，USB_HOST 不能插入任何 USB 设备！另外，如果只有 STM32F767 核心板的，可以利用核心板上面的 MicroUSB 接电脑，同样也可以实现本例程的功能。

61.3 软件设计

本章，我们在：实验 41 NAND FLASH 实验 的基础上修改，代码移植自 ST 官方例程：STM32_USB-Host-Device_Lib_V2.2.0\Project\USB_Device_Examples\MSC。由于 V2.2.0 的库，仅提供 IAR 工程，所以无法用 MDK 直接打开 ST 的这个例程，不过大家可以参考 V2.1.0 的库（光盘有提供：STM32_USB-Host-Device_Lib_V2.1.0.rar），V2.1.0 的库提供了 MDK 工程，它的工程结构和 V2.2.0 的库是一样的。

我们使用 IAR 打开该例程（V2.2.0 仅提供 IAR 工程）即可知道 USB 相关的代码有哪些，如图 61.3.1 所示：

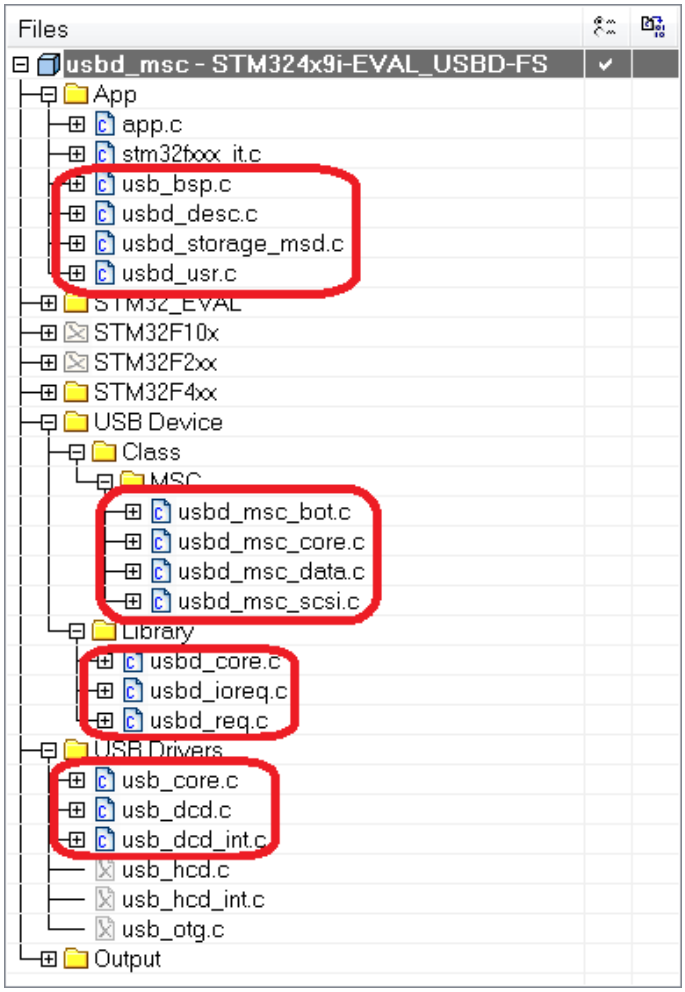


图 61.3.1 ST 官方例程 USB 相关代码

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 41 NAND FLASH 实验）的工程文件夹下面，新建一个 USB 文件夹，并拷贝官方 USB 驱动库相关代码到该文件夹下，即拷贝：光盘→ 8，STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.2.0→Libraries 文件夹下的 STM32_USB_Device_Library、STM32_USB_HOST_Library 和 STM32_USB_OTG_Driver 等三个文件夹的源码到该文件夹下面。

然后，在 USB 文件夹下，新建 USB_APP 文件夹存放 MSC 实现相关代码，即：STM32_USB-Host-Device_Lib_V2.2.0→Project→USB_Device_Examples→MSC→src 下的部分代码：usb_bsp.c、usbd_storage_msd.c、usbd_desc.c 和 usbd_usr.c 等 4 个.c 文件，同时拷贝 STM32_USB-Host-Device_Lib_V2.2.0→Project→USB_Device_Examples→MSC→inc 下面的：usb_conf.h、usbd_conf.h 和 usbd_desc.h 等三个文件到 USB_APP 文件夹下，最后 USB_APP 文件夹下的文件如图 61.3.2 所示：

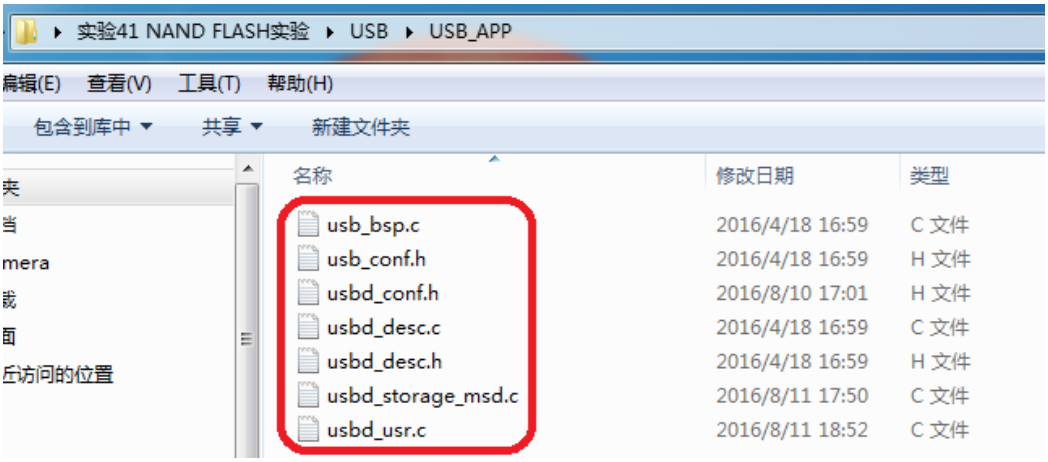


图 61.3.2 USB_APP 代码

之后，根据 ST 官方 MSC 例程，在我们本章例程的基础上新建分组添加相关代码，具体细节，这里就不详细介绍了，添加好之后，如图 61.3.3 所示：

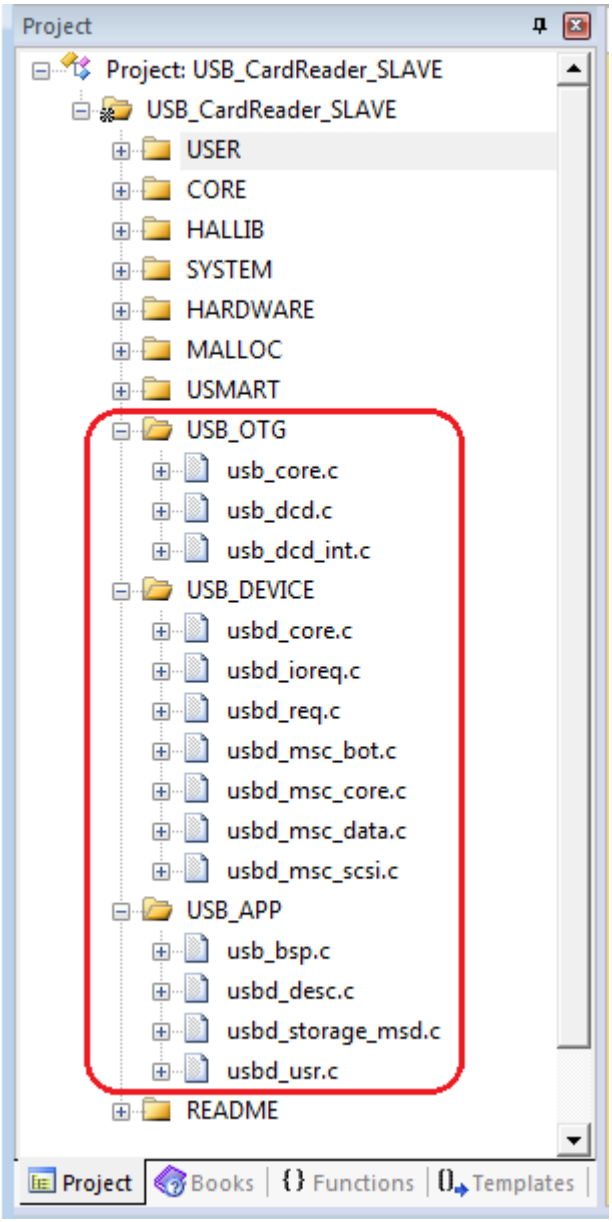


图 61.3.3 添加 USB 驱动等相关代码

因为这个 USB 库是针对 STM32F1/F2/F4 系列的，并不是针对 STM32F7 系列，需要对部分代码稍作修改。需要修改的地方有：

1，usb_core.c，USB_OTG_CoreInit 函数改为：

```
USB_OTG_STS USB_OTG_CoreInit(USB_OTG_CORE_HANDLE *pdev)
{
  USB_OTG_STS status = USB_OTG_OK;
  USB_OTG_GUSBCFG_TypeDef    usbcfg;
  USB_OTG_GCCFG_TypeDef      gccfg;
  USB_OTG_GAHBCFG_TypeDef    ahbcfg;
#if defined (STM32F446xx) || defined (STM32F469_479xx) || defined (STM32F767xx)
//增加对 STM32F767xx 的判断
  USB_OTG_DCTL_TypeDef       dctl;
```

```
    u32 tempreg;
#endif
    usbcfg.d32 = 0;
    ……//省略部分未改动的代码
    USB_OTG_EnableCommonInt(pdev);
#endif
#if defined (STM32F446xx) || defined (STM32F469_479xx) || defined (STM32F767xx)
    //增加对 STM32F767xx 的判断
    //必须注释掉这个(对 usbcfg.srpcap 的修改),否则无法正常使用!!
    //usbcfg.d32 = USB_OTG_READ_REG32(&pdev->regs.GREGS->GUSBCFG);
    //usbcfg.b.srpcap = 1;
    //必须新增对 GOTGCTL 寄存器 bit6,bit7 的设置,否则 USB 工作不正常
    tempreg=USB_OTG_READ_REG32(&pdev->regs.GREGS->GOTGCTL);  //读 GOTGCTL
    tempreg|=1<<6;          //设置 BVALOEN=1
    tempreg|=1<<7;          //设置 BVALOVAL=1
    USB_OTG_WRITE_REG32(&pdev->regs.GREGS->GOTGCTL,tempreg);  //写 GOTGCTL
    dctl.d32 = USB_OTG_READ_REG32(&pdev->regs.DREGS->DCTL);
    dctl.b.sftdiscon    = 0;
    USB_OTG_WRITE_REG32(&pdev->regs.DREGS->DCTL, dctl.d32);
    dctl.d32 = USB_OTG_READ_REG32(&pdev->regs.DREGS->DCTL);
    //USB_OTG_WRITE_REG32(&pdev->regs.GREGS->GUSBCFG, usbcfg.d32);
    USB_OTG_EnableCommonInt(pdev);
#endif
      return status;
}
```

2，usb_dcd.c ，DCD_Init 函数，将：

```
#if defined (STM32F446xx) || defined (STM32F469_479xx)
```

改为：

```
#if defined (STM32F446xx) || defined (STM32F469_479xx) || defined (STM32F767xx)
//增加对 STM32F767xx 的判断
```

3，usb_dcd_int.c ，DCD_HandleEnumDone_ISR 函数，改为：

```
static uint32_t DCD_HandleEnumDone_ISR(USB_OTG_CORE_HANDLE *pdev)
{
    uint32_t hclk = 216000000;              //hclk 时钟为 216Mhz
    USB_OTG_GINTSTS_TypeDef    gintsts;
    USB_OTG_GUSBCFG_TypeDef    gusbcfg;
    //RCC_ClocksTypeDef RCC_Clocks;       //屏蔽掉,系统时钟直接赋值
    USB_OTG_EP0Activate(pdev);
    /* Get HCLK frequency */
  // RCC_GetClocksFreq(&RCC_Clocks);    //屏蔽掉,系统时钟直接赋值
  //hclk = RCC_Clocks.HCLK_Frequency;   //屏蔽掉,系统时钟直接赋值
    ……//省略部分未改动的代码
}
```

经过这三处修改(接下来的 USB 例程，都需要做这个修改)，就可以使得该库支持 STM32F7 系列芯片了。此外，移植官方库的时候，我们重点要修改的就是 USB_APP 文件夹下面的代码。其他代码（USB_OTG 和 USB_DEVICE 文件夹下的代码）一般不用修改。

usb_bsp.c 提供了几个 USB 库需要用到的底层初始化函数，包括：IO 设置、中断设置、VBUS 配置以及延时函数等，需要我们自己实现。USB Device（Slave）和 USB Host 共用这个.c 文件。

usbd_desc.c 提供了 USB 设备类的描述符，直接决定了 USB 设备的类型、断点、接口、字符串、制造商等重要信息。这个里面的内容，我们一般不用修改，直接用官方的即可。注意，这里：usbd_desc.c 里面的：usbd 即 device 类，同样：usbh 即 host 类，所以通过文件名我们可以很容易区分该文件是用在 device 还是 host，而只有 usb 字样的那就是 device 和 host 可以共用的。

usbd_usr.c 提供用户应用层接口函数，即 USB 设备类的一些回调函数，当 USB 状态机处理完不同事务的时候，会调用这些回调函数，我们通过这些回调函数，就可以知道 USB 当前状态，比如：是否枚举成功了？是否连接上了？是否断开了？等，根据这些状态，用户应用程序可以执行不同操作，完成特定功能。

usbd_storage_msd.c 提供一些磁盘操作函数，包括支持的磁盘个数，以及每个磁盘的初始化和读写等函数。本章我们设置了 3 个磁盘：SD 卡、SPI FLASH 和 NAND FLASH。

以上 4 个.c 文件里面的函数，基本上都是以回调函数的形式，被 USB 驱动库调用的。这些代码的具体修改过程，我们这里不详细介绍，请大家参考光盘本例程源码，这里只提几个重点地方讲解下：

1，要使用 USB OTG FS，必须在 MDK 编译器的全局宏定义里面，定义：USE_USB_OTG_FS 宏，如图 58.3.4 所示：

图 61.3.4 定义全局宏 USE_USB_OTG_FS

2，因为阿波罗 STM32F767 开发板没有用到 VUSB 电压检测，所以要在 usb_conf.h 里面，将宏定义：#define VBUS_SENSING_ENABLED，屏蔽掉。

3，通过修改 usbd_conf.h 里面的 MSC_MEDIA_PACKET 定义值大小，可以一定程度提高 USB 读写速度（越大越快），本例程我们设置 32*1024，也就是 32KB 大小。

4，官方例程不支持大于 4G 的 SD 卡，得修改 usbd_msc_scsi.c 里面的 SCSI_blk_addr 类型为 uint64_t，才可以支持大于 4G 的卡，官方默认是 uint32_t，最大只能支持 4G 卡。

5，官方例程在 2 个或以上磁盘支持的时候，存在 bug，需要修改 usbd_msc_scsi.c 里面的 SCSI_blk_nbr 变量，将其改为数组形式：uint32_t SCSI_blk_nbr[3]; 这里，数组大小是 3，我们可以支持最多 3 个磁盘，修改数组的大小，即可修改支持的最大磁盘个数。修改该参数后，相应的有一些函数要做修改，请大家参考本例程源码。

6，修改 usbd_msc_core.c 里面的 USBD_MSC_MaxLun 定义方式，去掉 static 关键字。然后，在 usbd_msc_bot.c 里面修改 MSC_BOT_CBW_Decode 函数，将 MSC_BOT_cbw.bLUN > 1 改为：MSC_BOT_cbw.bLUN > USBD_MSC_MaxLun，以支持多个磁盘。

以上 6 点，就是我们移植的时候需要特别注意的，其他我们就不详细介绍了（USB 相关

源码解释，请参考：CD00289278.pdf 这个文档），最后修改 main.c 里面代码如下：

```
USB_OTG_CORE_HANDLE USB_OTG_dev;
extern vu8 USB_STATUS_REG;          //USB 状态
extern vu8 bDeviceState;            //USB 连接 情况

int main(void)
{
    u8 offline_cnt=0;
    u8 tct=0;
    u8 USB_STA;
    u8 Divece_STA;

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    PCF8574_Init();                 //初始化 PCF8574
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部内存池
    my_mem_init(SRAMDTCM);          //初始化 DTCM 内存池
    POINT_COLOR=RED;
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    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,"2016/7/20");
    if(SD_Init())LCD_ShowString(30,130,200,16,16,"SD Card Error!");  //检测 SD 卡错误
    else //SD 卡正常
    {
        LCD_ShowString(30,130,200,16,16,"SD Card Size:        MB");
        LCD_ShowNum(134,130,SDCardInfo.CardCapacity>>20,5,16);//显示 SD 卡容量
    }
    if(W25QXX_ReadID()!=W25Q256)
        LCD_ShowString(30,130,200,16,16,"W25Q128 Error!");   //检测 W25Q128 错误
    else //SPI FLASH 正常
    {
        LCD_ShowString(30,150,200,16,16,"SPI FLASH Size:25MB");
    }
```

```
if(FTL_Init())LCD_ShowString(30,170,200,16,16,"NAND Error!");  //检测 NAND 错误
else //NAND FLASH  正常
{
    LCD_ShowString(30,170,200,16,16,"NAND Flash Size:      MB");
    LCD_ShowNum(158,170,nand_dev.valid_blocknum*nand_dev.block_pagenum*\
            nand_dev.page_mainsize>>20,4,16);    //显示 SD 卡容量
}
LCD_ShowString(30,190,200,16,16,"USB Connecting..."); //提示正在建立连接
MSC_BOT_Data=mymalloc(SRAMIN,MSC_MEDIA_PACKET);                //申请内存
USBD_Init(&USB_OTG_dev,USB_OTG_FS_CORE_ID,&USR_desc,
                                        &USBD_MSC_cb,&USR_cb);
delay_ms(1800);
while(1)
{
    delay_ms(1);
    if(USB_STA!=USB_STATUS_REG)//状态改变了
    {
        LCD_Fill(30,210,240,210+16,WHITE);//清除显示
        if(USB_STATUS_REG&0x01)//正在写
        {
            LED1(0);
            LCD_ShowString(30,210,200,16,16,"USB  Writing...");//提示正在写入数据

        }
        if(USB_STATUS_REG&0x02)//正在读
        {
            LED1(0);
            LCD_ShowString(30,210,200,16,16,"USB  Reading...");//提示正读出数据

        }
        if(USB_STATUS_REG&0x04)LCD_ShowString(30,230,200,16,16,
                                        "USB Write Err ");//提示写入错误
        else LCD_Fill(30,230,240,230+16,WHITE);//清除显示
        if(USB_STATUS_REG&0x08)LCD_ShowString(30,250,200,16,16,
                                    "USB Read    Err ");//提示读出错误
        else LCD_Fill(30,250,240,250+16,WHITE);//清除显示
        USB_STA=USB_STATUS_REG;//记录最后的状态
    }
    if(Divece_STA!=bDeviceState)
    {
        if(bDeviceState==1)LCD_ShowString(30,190,200,16,16,
                    "USB Connected      ");//提示 USB 连接已经建立
        else LCD_ShowString(30,190,200,16,16,
```

```
                                        "USB DisConnected ");//提示 USB 被拔出了
                Divece_STA=bDeviceState;
            }
            tct++;
            if(tct==200)
            {
                tct=0;
                LED1(1);
                LED0_Toggle;//提示系统在运行
                if(USB_STATUS_REG&0x10)
                {
                    offline_cnt=0;//USB 连接了,则清除 offline 计数器
                    bDeviceState=1;
                }else//没有得到轮询
                {
                    offline_cnt++;
                    if(offline_cnt>10)bDeviceState=0;//2s 内没收到在线标记代表 USB 被拔出

                }
                USB_STATUS_REG=0;
            }
        }
    }
```

其中，USB_OTG_CORE_HANDLE 是一个全局结构体类型，用于存储 USB 通信中 USB 内核需要使用的的各种变量、状态和缓存等，任何 USB 通信（不论主机，还是从机），我们都必须定义这么一个结构体以实现 USB 通信，这里定义成：USB_OTG_dev。

然后，USB 初始化非常简单，只需要调用 USBD_Init 函数即可，顾名思义，该函数是 USB 设备类初始化函数，本章的 USB 读卡器属于 USB 设备类，所以使用该函数。该函数初始化了 USB 设备类处理的各种回调函数，以便 USB 驱动库调用。执行完该函数以后，USB 就启动了，所有 USB 事务，都是通过 USB 中断触发，并由 USB 驱动库自动处理。USB 中断服务函数在 usbd_usr.c 里面：

```
//USB OTG  中断服务函数，处理所有 USB 中断
void OTG_FS_IRQHandler(void)
{
    USBD_OTG_ISR_Handler(&USB_OTG_dev);
}
```

该函数调用 USBD_OTG_ISR_Handler 函数来处理各种 USB 中断请求。因此在 main 函数里面，我们的处理过程就非常简单，main 函数里面通过两个全局状态变量（USB_STATUS_REG 和 bDeviceState），来判断 USB 状态，并在 LCD 上面显示相关提示信息。

USB_STATUS_REG 在 usbd_storage_msd.c 里面定义的一个全局变量，不同的位表示不同状态，用来指示当前 USB 的读写等操作状态。

bDeviceState 是在 usbd_usr.c 里面定义的一个全局变量，0 表示 USB 还没有连接；1 表示 USB 已经连接。

注意：因为 USB 通信需要 48Mhz 的 USB 时钟，这了我们把 STM32 的主频倍频到 192Mhz（稍微超频，不影响正常使用），以得到 48Mhz 的 USB 时钟。接下来的几个 USB 例程，我们也都采用 192Mhz 的主频，以得到 48Mhz 的 USB 时钟。软件设计部分，就给大家介绍到这里。

61.4 下载验证

在代码编译成功之后，我们下载到阿波罗 STM32 开发板上，在 USB 配置成功后（假设已经插入 SD 卡，**注意：USB 数据线，要插在 USB_SLAVE 口！不是 USB_232 端口！另外，USB_HOST 接口，也不要插入任何设备，否则会干扰！！**），LCD 显示效果如图 61.4.1 所示：

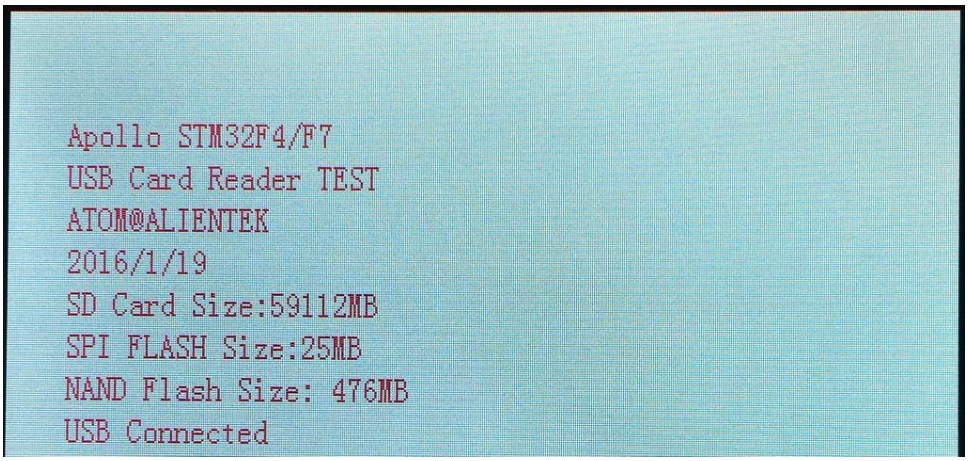


图 61.4.1 USB 连接成功

此时，电脑提示发现新硬件，并开始自动安装驱动，如图 61.4.2 所示：

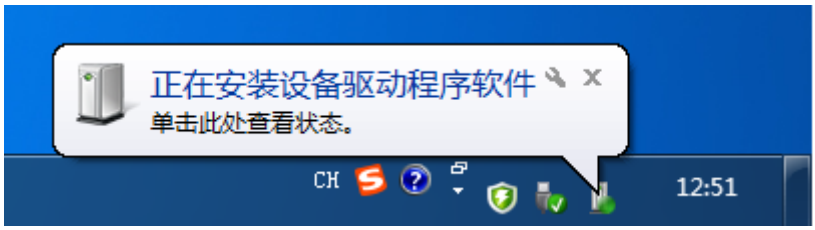


图 61.4.2 USB 读卡器被电脑找到

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 61.4.3 所示：

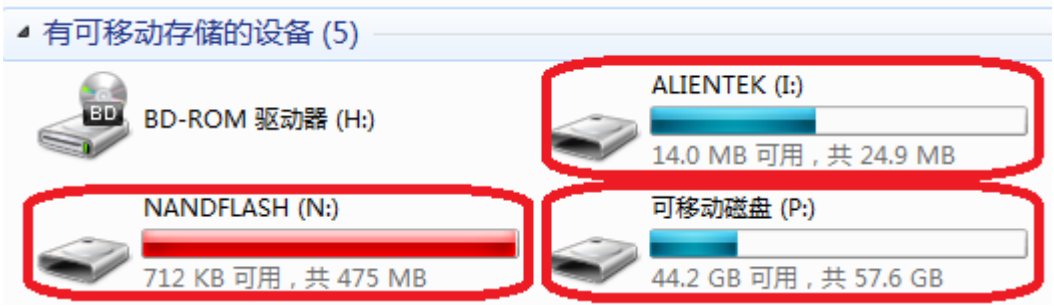


图 61.4.3 电脑找到 USB 读卡器的三个盘符

我们打开设备管理器，在通用串行总线控制器里面可以发现多出了一个 USB 大容量存储设备，同时看到磁盘驱动器里面多了 3 个磁盘，如图 61.4.4 所示：

图 61.4.4 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、SPI FLASH 和 NAND 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!!

第六十二章 USB 声卡(Slave)实验

上一章我们向大家介绍了如何利用 STM32F767 的 USB 接口来做一个 USB 读卡器，本章我们将利用 STM32F767 的 USB 来做一个声卡。本章分为如下几个部分：

62.1 USB 声卡简介
62.2 硬件设计
62.3 软件设计
62.4 下载验证

62.1 USB 声卡简介

ALIENTEK 阿波罗 STM32F767 开发板板载了一颗高性能 CODEC 芯片：WM8978，我们可以利用 STM32F767 的 SAI 接口，控制 WM8978 播放音乐，同样，如果结合 STM32F767 的 USB 功能，就可以实现一个 USB 声卡。

同上一章一样，我们直接移植官方的 USB AUDIO 例程，官方例程路径：8，STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.2.0→Project→USB_Device_Examples→AUDIO，该例程采用 USB 同步传输来传输音频数据流并且支持某些控制命令(比如静音控制)，例程仅支持 USB FS 模式（不支持 HS），同时例程不需要特殊的驱动支持，大多数操作系统直接就可以识别。

62.2 硬件设计

本节实验功能简介：开机的时候先显示一些提示信息，之后开始 USB 配置，在配置成功之后就可以在电脑上发现多出一个 USB 声卡。我们用 DS1 来指示 USB 是否连接成功，并在液晶上显示 USB 连接状况，如果成功连接，DS1 会亮，此时，我们可以将耳机插入开发板的 PHONE 端口，听到来自电脑的音频信号，同时通过两个按键可以调节音量：按 KEY0 可以增大音量，KEY2 可以减少音量。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） KEY0 和 KEY2 两个按键
3） 串口
4） LCD 模块
5） USB SLAVE 接口
6） WM8978

这几个部分，在之前的实例中都已经介绍过了，我们在此就不多说了。这里再次提醒大家，P10 的连接，要通过跳线帽连接 PA11 和 D-以及 PA12 和 D+。

62.3 软件设计

本章，我们在第五十二章 音乐播放器实验（实验 47）的基础上修改，先打开实验 47 的工程，在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹，同上一章一样，对照官方 AUDIO 例子，将相关文件拷贝到 USB 文件夹下。

然后，我们在工程里面去掉一些不必要的代码，并添加 USB 相关代码，最终得到如图 62.3.1

所示的工程：

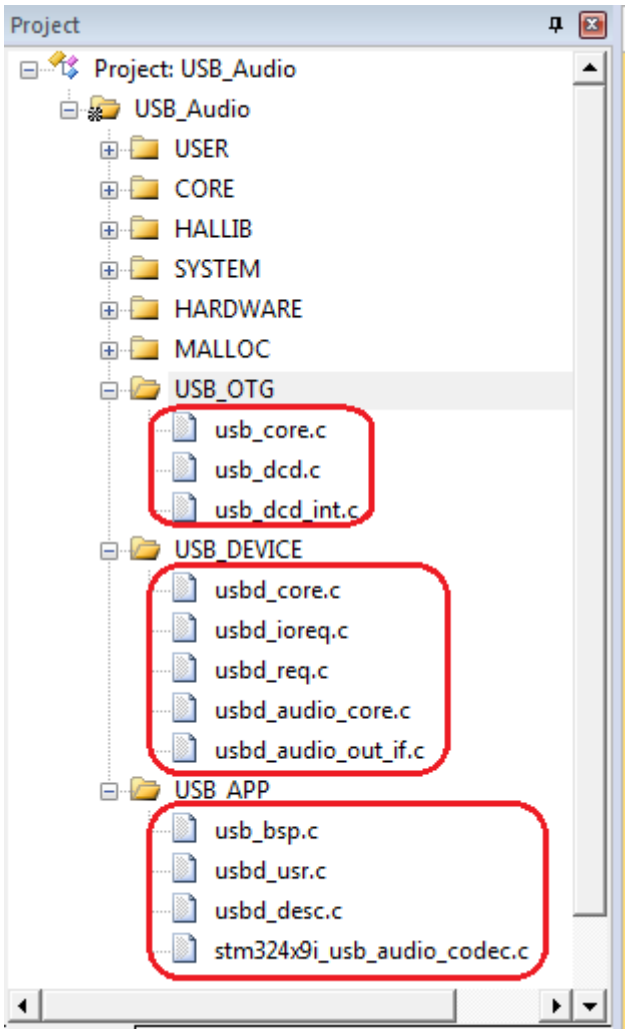


图 59.3.1 USB 声卡工程截图

可以看到，USB 部分代码，同上一章的在结构上是一模一样的，只是.c 文件稍微有些变化。同样，我们移植需要修改的代码，就是 USB_APP 里面的这四个.c 文件了。

其中 usb_bsp.c 和 usbd_usr.c 的代码，和上一章基本一样，可以用上一章的代码直接替换即可正常使用。

usb_desc.c 代码，同上一章不一样，上一章描述符是大容量存储设备，本章变成了 USB 声卡了，所以直接用 ST 官方的就行。

最后 stm324xg_usb_audio_codec.c，这里面的代码，是重点要修改的，该文件是配合 USB 声卡的 WM8994 底层驱动相关代码，官方 STM32F7xx 的板子，用的是 WM8994，而我们用的是 WM8978，需要修改代码，修改后代码如下：

```
u8 volume=0;                                    //当前音量

vu8 audiostatus=0;                              //bit0:0,暂停播放;1,继续播放
vu8 saiplaybuf=0;                               //即将播放的音频帧缓冲编号
vu8 saisavebuf=0;                               //当前保存到的音频缓冲编号
#define AUDIO_BUF_NUM       100     //由于采用的是 USB 同步传输数据播放
//而 STM32 SAI 的速度和 USB 传送过来数据的速度存在差异,比如在 48Khz 下,实
```

```
    //际 SAI 是低于 48Khz(47.991Khz)的,所以电脑送过来的数据流,会比 STM32 播放
    //速度快,缓冲区写位置追上播放位置(saisavebuf==saiplaybuf)时,就会出现
    //混叠.设置尽量大的 AUDIO_BUF_NUM 值,可以尽量减少混叠次数.

    u8 *saibuf[AUDIO_BUF_NUM];                          // 音 频 缓 冲 帧 , 占 用 内 存 数
=AUDIO_BUF_NUM*AUDIO_OUT_PACKET 字节

    //音频数据 SAI DMA 传输回调函数
    void audio_sai_dma_callback(void)
    {
        if((saiplaybuf==saisavebuf)&&audiostatus==0)
        {
            SAI_Play_Stop();
        }else
        {
            saiplaybuf++;
            if(saiplaybuf>(AUDIO_BUF_NUM-1))saiplaybuf=0;
            if(DMA2_Stream1->CR&(1<<19))
            {
                DMA2_Stream1->M0AR=(u32)saibuf[saiplaybuf];//指向下一个 buf
            }
            else
            {
                DMA2_Stream1->M1AR=(u32)saibuf[saiplaybuf];//指向下一个 buf
            }
        }
    }

    //配置音频接口
    //OutputDevice:输出设备选择,未用到.
    //Volume:音量大小,0~100
    //AudioFreq:音频采样率
    uint32_t EVAL_AUDIO_Init(uint16_t OutputDevice, uint8_t Volume, uint32_t AudioFreq)
    {
        u16 t=0;
        for(t=0;t<AUDIO_BUF_NUM;t++)            //内存申请
        {
            saibuf[t]=mymalloc(SRAMIN,AUDIO_OUT_PACKET);
        }
        if(saibuf[AUDIO_BUF_NUM-1]==NULL)  //内存申请失败
        {
            printf("Malloc Error!\r\n");
            for(t=0;t<AUDIO_BUF_NUM;t++)myfree(SRAMIN,saibuf[t]);
```

```
            return 1;
        }
        SAIA_Init(SAI_MODEMASTER_TX,SAI_CLOCKSTROBING_RISINGEDGE,
                                SAI_DATASIZE_16);//设置 SAI,主发送,16 位数据
        SAIA_SampleRate_Set(AudioFreq);     //设置采样率
        EVAL_AUDIO_VolumeCtl(Volume);   //设置音量
        SAIA_TX_DMA_Init(saibuf[0],saibuf[1],AUDIO_OUT_PACKET/2,1);
                                                    //配置 TX DMA,16 位
        sai_tx_callback=audio_sai_dma_callback;         //回调函数指向 audio_sai_dma_callback
        SAI_Play_Start();                               //开启 DMA
        printf("EVAL_AUDIO_Init:%d,%d\r\n",Volume,AudioFreq);
        return 0;
}

//开始播放音频数据
//pBuffer:音频数据流首地址指针
//Size:数据流大小(单位:字节)
uint32_t EVAL_AUDIO_Play(uint16_t* pBuffer, uint32_t Size)
{
        printf("EVAL_AUDIO_Play:%x,%d\r\n",pBuffer,Size);
        return 0;
}

//暂停/恢复音频流播放
//Cmd:0,暂停播放;1,恢复播放
//返回值:0,成功
//       其他,设置失败
uint32_t EVAL_AUDIO_PauseResume(uint32_t Cmd)
{
        if(Cmd==AUDIO_PAUSE)
        {
            audiostatus=0;
        }else
        {
            audiostatus=1;
            SAI_Play_Start();   //开启 DMA
        }
        return 0;
}

//停止播放
//Option:控制参数,1/2,详见:CODEC_PDWN_HW 定义
//返回值:0,成功
```

```
//      其他,设置失败
uint32_t EVAL_AUDIO_Stop(uint32_t Option)
{
    printf("EVAL_AUDIO_Stop:%d\r\n",Option);
    audiostatus=0;
    return 0;
}
//音量设置
//Volume:0~100
//返回值:0,成功
//      其他,设置失败
uint32_t EVAL_AUDIO_VolumeCtl(uint8_t Volume)
{
    volume=Volume;
    WM8978_HPvol_Set(volume*0.63,volume*0.63);
    WM8978_SPKvol_Set(volume*0.63);
    return 0;
}
//静音控制
//Cmd:0,正常
//      1,静音
//返回值:0,正常
//      其他,错误代码
uint32_t EVAL_AUDIO_Mute(uint32_t Cmd)
{
    if(Cmd==AUDIO_MUTE_ON)
    {
        WM8978_HPvol_Set(0,0);
        WM8978_SPKvol_Set(0);
    }else
    {
        WM8978_HPvol_Set(volume*0.63,volume*0.63);
        WM8978_SPKvol_Set(volume*0.63);
    }
    return 0;
}
//播放音频数据流
//Addr:音频数据流缓存首地址
//Size:音频数据流大小(单位:harf word,也就是 2 个字节)
void Audio_MAL_Play(uint32_t Addr, uint32_t Size)
{
    u16 i;
    u8 t=saisavebuf;
```

```
u8 *p=(u8*)Addr;
u8 curplay=saiplaybuf;  //当前正在播放的缓存帧编号
if(curplay)curplay--;
else curplay=AUDIO_BUF_NUM-1;
audiostatus=1;
t++;
if(t>(AUDIO_BUF_NUM-1))t=0;
if(t==curplay)              //写缓存碰上了当前正在播放的帧,跳到下一帧
{
    t++;
    if(t>(AUDIO_BUF_NUM-1))t=0;
    printf("bad position:%d\r\n",t);
}
saisavebuf=t;
for(i=0;i<Size*2;i++)
{
    saibuf[saisavebuf][i]=p[i];
}
SAI_Play_Start();       //开启 DMA
```

这里特别说明一下，USB AUDIO 我们使用的是 USB 同步数据传输，音频采样率固定为：48Khz（通过 USBD_AUDIO_FREQ 设置，在 usbd_conf.h 里面），这样，USB 传输过来的数据都是 48Khz 的音频数据流，STM32F767 必须以同样的频率传输数据给 SAI，以同步播放音乐。

但是，STM32F767 我们采用的是外部 25M 时钟倍频后分频作为 SAI 时钟的，在使能主时钟（MCK）输出的时候，只能以 47.991Khz 频率播放，稍微有点误差，这样，导致 USB 送过来的数据，会比传输给 SAI 的数据快一点点，如果不做处理，就很容易产生数据混叠，产生噪音。

因此，我们这里提供了一个简单的解决办法：建立一个类似 FIFO 结构的缓冲数组，USB 传输过来的数据全部存放在这些数组里面，同时通过 SAI DMA 双缓冲机制，播放这些数组里面的音频数据，当混叠发生时（USB 传过来的数据，赶上 SAI 播放的数据了），直接越过当前正在播放的数组，继续保存。这样，虽然会导致一些数据丢失（混叠时），但是避免了混叠，保证了良好的播放效果（听不到噪音），同时，数组个数越多，效果就越好（越不容易混叠）。

以上代码 AUDIO_BUF_NUM 就是我们定义的 FIFO 结构数组的大小，越大，效果越好，这里我们定义成 100，每个数组的大小由音频采样率和位数决定，计算公式为：

(USBD_AUDIO_FREQ * 2 * 2) /1000

单位为字节，其中 USBD_AUDIO_FREQ 即音频采样率：48Khz，这样，每个数组大小就是 192 字节。100 个数组，我们总共用了 19200 字节。

audio_sai_dma_callback 函数是 SAI 播放音频的回调函数，完成 SAI 数据流的发送（切换 DMA 源地址），其他函数则基本都是在 usbd_audio_out_if.c 里面被调用，这里就不再详细介绍了。

最后在 main.c 里面，我们修改 main 函数如下：

```
USB_OTG_CORE_HANDLE USB_OTG_dev;
extern vu8 bDeviceState;            //USB 连接 情况
extern u8 volume;                   //音量(可通过按键设置)
```

```
int main(void)
{
    u8 key;
    u8 t=0;
    u8 Divece_STA=0XFF;

    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    SDRAM_Init();                   //初始化 SDRAM
    LCD_Init();                     //初始化 LCD
    W25QXX_Init();                  //初始化 W25Q256
    PCF8574_Init();                 //初始化 PCF8574
    WM8978_Init();                  //初始化 WM8978
    WM8978_ADDA_Cfg(1,0);           //开启 DAC
    WM8978_Input_Cfg(0,0,0);        //关闭输入通道
    WM8978_Output_Cfg(1,0);         //开启 DAC 输出
    my_mem_init(SRAMIN);            //初始化内部内存池
    my_mem_init(SRAMEX);            //初始化外部内存池
    my_mem_init(SRAMDTCM);          //初始化 DTCM 内存池

    POINT_COLOR=RED;//设置字体为红色
    LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
    LCD_ShowString(30,70,200,16,16,"USB Sound Card TEST");
    LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,110,200,16,16,"2016/8/10");
    LCD_ShowString(30,130,200,16,16,"KEY2:Vol-    KEY0:vol+");
    POINT_COLOR=BLUE;//设置字体为蓝色
    LCD_ShowString(30,160,200,16,16,"VOLUME:");          //音量显示
    LCD_ShowxNum(30+56,160,DEFAULT_VOLUME,3,16,0X80);//显示音量
    LCD_ShowString(30,180,200,16,16,"USB Connecting...");//提示正在建立连接
    USBD_Init(&USB_OTG_dev,USB_OTG_FS_CORE_ID,&USR_desc,
                                        &AUDIO_cb,&USR_cb);
    while(1)
    {
        key=KEY_Scan(1);//支持连按
        if(key)
        {
```

```
            if(key==KEY0_PRES)          //KEY0 按下,音量增加
            {
                volume++;
                if(volume>100)volume=100;
            }else if(key==KEY2_PRES)//KEY2 按下,音量减少
            {
                if(volume)volume--;
                else volume=0;
            }
            EVAL_AUDIO_VolumeCtl(volume);
            LCD_ShowxNum(30+56,160,volume,3,16,0X80);//显示音量
            delay_ms(20);
        }
        if(Divece_STA!=bDeviceState)//状态改变了
        {
            if(bDeviceState==1)
            {
                LED1(0);
                LCD_ShowString(30,180,200,16,16,
                               "USB Connected      ");//提示 USB 连接已经建立
            }else
            {
                LED1(1);
                LCD_ShowString(30,180,200,16,16,
                               "USB DisConnected ");//提示 USB 连接失败
            }
            Divece_STA=bDeviceState;
        }
        delay_ms(20);
        t++;
        if(t>10)
        {
            t=0;
            LED0_Toggle;
        }
    }
}
```

此部分代码比较简单，同上一章一样定义了 USB_OTG_dev 结构体，然后通过 USBD_Init 初始化 USB，不过本章实现的是 USB 声卡功能。本章我们保留了原例程（实验 47）的 USMART 部分，同样可以通过串口 1 设置 WM8978 相关参数。

其他部分我们就不详细介绍了，软件设计部分就为大家介绍到这里。

62.4 下载验证

在代码编译成功之后，我们通过下载代码到阿波罗 STM32 开发板上，在 USB 配置成功后（注意：USB 数据线，要插在 USB_SLAVE 端口！不是 USB_232 端口！另外，USB_HOST 接口不要插任何外设！），LCD 显示效果如图 62.4.1 所示：

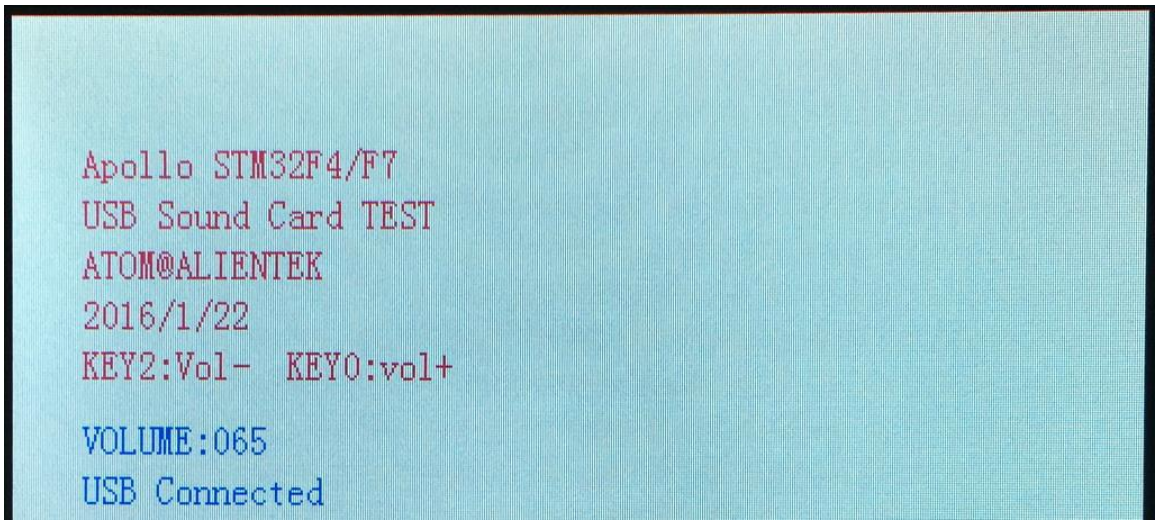


图 62.4.1 USB 连接成功

此时，电脑提示发现新硬件，并自动完成驱动安装，如图 62.4.2 所示：

图 62.4.2 电脑找到 ALIENTEK USB 声卡

等 USB 配置成功后，DS1 常亮，DS0 闪烁，并且在设备管理器→声音、视频和游戏控制器里面看到多了 ALIENTEK STM32F4/F7 USB AUDIO 设备，如图 62.4.3 所示：

图 62.4.3 设备管理器找到 ALIENTEK USB 声卡

然后，设置 ALIENTEK STM32F4/F7 USB AUDIO 为电脑的默认播放设备，则电脑的所有声音都被切换到开发板输出，将耳机插入阿波罗 STM32 开发板的 PHONE 端口，即可听到来自

电脑的声音。通过按键 KEY0/KEY2 可以增大/减少音量，默认音量设置的是 65，大家可以自己调节（范围：0~100）。

第六十三章 USB 虚拟串口(Slave)实验

上一章我们向大家介绍了如何利用 STM32F7 的 USB 接口来做一个 USB 声卡，本章我们将利用 STM32F7 的 USB 来做一个虚拟串口（VCP）。本章分为如下几个部分：

63.1 USB 虚拟串口简介
63.2 硬件设计
63.3 软件设计
63.4 下载验证

63.1 USB 虚拟串口简介

USB 虚拟串口，简称 VCP，是 Virtual COM Port 的简写，它是利用 USB 的 CDC 类来实现的一种通信接口。

我们可以利用 STM32 自带的 USB 功能，来实现一个 USB 虚拟串口，从而通过 USB，实现电脑与 STM32 的数据互传。上位机无需编写专门的 USB 程序，只需要一个串口调试助手即可调试，非常实用。

同上一章一样，我们直接移植官方的 USB VCP 例程，官方例程路径：8，STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.2.0→Project→USB_Device_Examples→VCP，该例程采用 USB CDC 类来实现，利用 STM32 的 USB 接口，实现一个 USB 转串口的功能。

63.2 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 3（串口通信实验），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，并虚拟出一个串口（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 3 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） LCD 模块
4） USB SLAVE 接口

这几个部分，在之前的实例中都已经介绍过了，我们在此就不多说了。这里再次提醒大家，P10 的连接，要通过跳线帽连接 PA11 和 D-以及 PA12 和 D+。

63.3 软件设计

本章，我们在第三十章 IO 扩展实验（实验 25）的基础上修改，先打开实验 25 的工程，在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹，同上一章一样，对照官方 VCP 例

子，将相关文件拷贝到 USB 文件夹下。

然后，我们在工程里面去掉一些不必要的代码，并添加 USB 相关代码，最终得到如图 63.3.1 所示的工程：

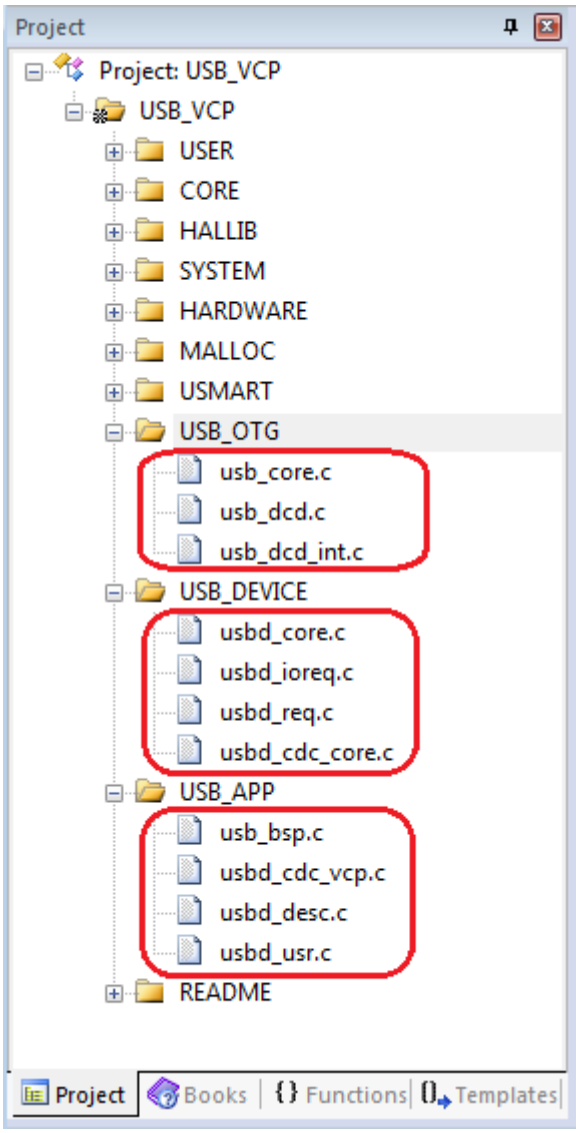


图 63.3.1 USB 虚拟串口工程截图

图 63.3.1 USB 虚拟串口工程截图

可以看到，USB 部分代码，同上一章的在结构上是一模一样的，只是.c 文件稍微有些变化。同样，我们移植需要修改的代码，就是 USB_APP 里面的这四个.c 文件了。

其中 usb_bsp.c 和 usbd_usr.c 的代码，和上一章基本一样，可以用上一章的代码直接替换即可正常使用。

usb_desc.c 代码，同上一章不一样，上一章描述符是大容量存储设备，本章变成了 USB 声卡了，所以直接用 ST 官方的就行。

最后 usbd_cdc_vcp.c，这里面的代码，是重点要修改的，修改后代码如下：

```
//USB 虚拟串口相关配置参数
LINE_CODING linecoding =
{
```

```
    115200,         //波特率
    0x00,     //停止位,默认 1 位
    0x00,     //校验位,默认无
    0x08            //数据位,默认 8 位
};

u8   USART_PRINTF_Buffer[USB_USART_REC_LEN]; //usb_printf 发送缓冲区

//用类似串口 1 接收数据的方法,来处理 USB 虚拟串口接收到的数据.
u8 USB_USART_RX_BUF[USB_USART_REC_LEN];
                                        //接收缓冲,最大 USART_REC_LEN 个字节.
//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 USB_USART_RX_STA=0;                             //接收状态标记


extern uint8_t    APP_Rx_Buffer [];          //虚拟串口发送缓冲区(发给电脑)
extern uint32_t APP_Rx_ptr_in;              //虚拟串口接收缓冲区(接收来自电脑的数据)

//虚拟串口配置函数(供 USB 内核调用)
CDC_IF_Prop_TypeDef VCP_fops =
{
    VCP_Init,
    VCP_DeInit,
    VCP_Ctrl,
    VCP_DataTx,
    VCP_DataRx
};

//初始化 VCP
//返回值:USBD_OK
uint16_t VCP_Init(void)
{
    return USBD_OK;
}
//复位 VCP
//返回值:USBD_OK
uint16_t VCP_DeInit(void)
{
    return USBD_OK;
}
```

```
//控制 VCP 的设置
//buf:命令数据缓冲区/参数保存缓冲区
//len:数据长度
//返回值:USBD_OK
uint16_t VCP_Ctrl (uint32_t Cmd, uint8_t* Buf, uint32_t Len)
{
    switch (Cmd)
    {
        case SEND_ENCAPSULATED_COMMAND:break;
        case GET_ENCAPSULATED_RESPONSE:break;
        case SET_COMM_FEATURE:break;
        case GET_COMM_FEATURE:break;
        case CLEAR_COMM_FEATURE:break;
        case SET_LINE_CODING:
            linecoding.bitrate = (uint32_t)(Buf[0] | \
              (Buf[1] << 8) | (Buf[2] << 16) | (Buf[3] << 24));
            linecoding.format = Buf[4];
            linecoding.paritytype = Buf[5];
            linecoding.datatype = Buf[6];
            //打印配置参数
            printf("linecoding.format:%d\r\n",linecoding.format);
            printf("linecoding.paritytype:%d\r\n",linecoding.paritytype);
            printf("linecoding.datatype:%d\r\n",linecoding.datatype);
            printf("linecoding.bitrate:%d\r\n",linecoding.bitrate);
            break;
        case GET_LINE_CODING:
            Buf[0] = (uint8_t)(linecoding.bitrate);
            Buf[1] = (uint8_t)(linecoding.bitrate >> 8);
            Buf[2] = (uint8_t)(linecoding.bitrate >> 16);
            Buf[3] = (uint8_t)(linecoding.bitrate >> 24);
            Buf[4] = linecoding.format;
            Buf[5] = linecoding.paritytype;
            Buf[6] = linecoding.datatype;
            break;
        case SET_CONTROL_LINE_STATE:break;
        case SEND_BREAK:break;
        default:break;
    }
    return USBD_OK;
}
//发送一个字节给虚拟串口(发给电脑)
//data:要发送的数据
//返回值:USBD_OK
```

```
uint16_t VCP_DataTx (uint8_t data)
{
    APP_Rx_Buffer[APP_Rx_ptr_in]=data;      //写入发送 buf
    APP_Rx_ptr_in++;                        //写位置加 1
    if(APP_Rx_ptr_in==APP_RX_DATA_SIZE)     //超过 buf 大小了,归零.
    {
        APP_Rx_ptr_in = 0;
    }
    return USBD_OK;
}
//处理从 USB 虚拟串口接收到的数据
//databuffer:数据缓存区
//Nb_bytes:接收到的字节数.
//返回值:USBD_OK
uint16_t VCP_DataRx (uint8_t* Buf, uint32_t Len)
{
    u8 i;
    u8 res;
    for(i=0;i<Len;i++)
    {
        res=Buf[i];
        if((USB_USART_RX_STA&0x8000)==0)        //接收未完成
        {
            if(USB_USART_RX_STA&0x4000)         //接收到了 0x0d
            {
                if(res!=0x0a)USB_USART_RX_STA=0;//接收错误,重新开始
                else USB_USART_RX_STA|=0x8000; //接收完成了
            }else //还没收到 0X0D
            {
                if(res==0x0d)USB_USART_RX_STA|=0x4000;
                else
                {
                    USB_USART_RX_BUF[USB_USART_RX_STA&0X3FFF]=res;
                    USB_USART_RX_STA++;
                    if(USB_USART_RX_STA>(USB_USART_REC_LEN-1))\
                        USB_USART_RX_STA=0;//接收数据错误,重新开始接收
                }
            }
        }
    }
    return USBD_OK;
}
//usb 虚拟串口,printf  函数
```

```
//确保一次发送数据不超 USB_USART_REC_LEN 字节
void usb_printf(char* fmt,...)
{
    u16 i,j;
    va_list ap;
    va_start(ap,fmt);
    vsprintf((char*)USART_PRINTF_Buffer,fmt,ap);
    va_end(ap);
    i=strlen((const char*)USART_PRINTF_Buffer);//此次发送数据的长度
    for(j=0;j<i;j++)//循环发送数据
    {
        VCP_DataTx(USART_PRINTF_Buffer[j]);
    }
}
```

此部分总共 6 个函数，其中前 5 个函数，用于初始化 VCP_fops 结构体，给 USB 内核调用，以实现相关功能。接下来我们分别介绍这几个函数。

VCP_Init 用于初始化 VCP，在初始化的时候由 USB 内核调用，这里我们无需任何操作，所以直接范围 USBD_OK 即可。

VCP_DeInit 用于复位 VCP，我们用不到，所以直接返回 USBD_OK 即可。

VCP_Ctrl 用于控制 VCP 的相关参数，根据 cmd 的不同，执行不同的操作，这里主要用到 SET_LINE_CODING 命令，该命令用于设置 VCP 的相关参数，比如波特率、数据类型（位数）、校验类型（奇偶校验）等，保存在 linecoding 结构体里面，在需要的时候，应用程序可以读取 linecoding 结构体里面的参数，以获得当前 VCP 的相关信息。

VCP_DataTx 用于发送一个字节的数据给 VCP，应用程序每调用一次该函数，就可以发送一个字节给 VCP，由 VCP 通过 USB 传输给电脑，实现 VCP 的数据发送。

VCP_DataRx 用于 VCP 的数据接收，当 STM32 的 USB 接收到电脑端串口发送过来的数据时，由 USB 内核程序调用该函数，实现 VCP 的数据接收。我们只需要在该函数里面，将接收到的数据，保存起来即可，接收的原理同第八章（实验 3 串口通信实验）完全一样。

usb_printf 用于实现和普通串口一样的 printf 操作，该函数将数据格式化输出到 USB VCP，功能完全同 printf，方便大家使用。

USB VCP 相关代码，就给大家介绍到这里，详细的介绍，请大家参考：CD00289278.pdf 这个文档。

最后在 main.c 里面，我们修改 main 函数如下：

```
USB_OTG_CORE_HANDLE      USB_OTG_dev;
extern vu8 bDeviceState;                //USB 连接  情况

int main(void)
{
    u16 t;
    u16 len;
    u16 times=0;
    u8 usbstatus=0;
```

```
Cache_Enable();                 //打开 L1-Cache
HAL_Init();                     //初始化 HAL 库
Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
delay_init(216);                //延时初始化
uart_init(115200);              //串口初始化
LED_Init();                     //初始化 LED
KEY_Init();                     //初始化按键
SDRAM_Init();                   //初始化 SDRAM
LCD_Init();                     //初始化 LCD
W25QXX_Init();                  //初始化 W25Q256
PCF8574_Init();                 //初始化 PCF8574
POINT_COLOR=RED;
LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
LCD_ShowString(30,70,200,16,16,"USB Virtual USART TEST");
LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,110,200,16,16,"2016/8/10");
LCD_ShowString(30,130,200,16,16,"USB Connecting...");//提示 USB 开始连接
USBD_Init(&USB_OTG_dev,USB_OTG_FS_CORE_ID,&USR_desc,
                                        &USBD_CDC_cb,&USR_cb);
while(1)
{
    if(usbstatus!=bDeviceState)//USB 连接状态发生了改变.
    {
        usbstatus=bDeviceState;//记录新的状态
        if(usbstatus==1)
        {
            POINT_COLOR=BLUE;
            LCD_ShowString(30,130,200,16,16,"USB Connected    ");//提示连接成功
            LED1(0);//DS1 亮
        }else
        {
            POINT_COLOR=RED;
            LCD_ShowString(30,130,200,16,16,"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 您发送的消息为:%d\r\n\r\n",len);
        for(t=0;t<len;t++)
        {
            VCP_DataTx(USB_USART_RX_BUF[t]);//以字节方式,发送给 USB
```

```
                }
                usb_printf("\r\n\r\n");//插入换行
                USB_USART_RX_STA=0;
            }else
            {
                times++;
                if(times%5000==0)
                {
                    usb_printf("\r\n 阿波罗 STM32F4/F7 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_Toggle;//闪烁 LED,提示系统正在运行.
                delay_ms(10);
            }
        }
    }
```

此部分代码比较简单，同上一章一样定义了 USB_OTG_dev 结构体，然后通过 USBD_Init 初始化 USB，不过本章实现的是 USB 虚拟串口的功能。然后在死循环里面轮询 USB 状态并检查是否接收到数据，如果接收到了数据，则通过 VCP_DataTx 将数据通过 VCP 原原本本的返回给电脑端串口调试助手。

其他部分我们就不详细介绍了，软件设计部分就为大家介绍到这里。

63.4 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，我们下载代码到阿波罗 STM32 开发板上，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 63.4.1 所示：

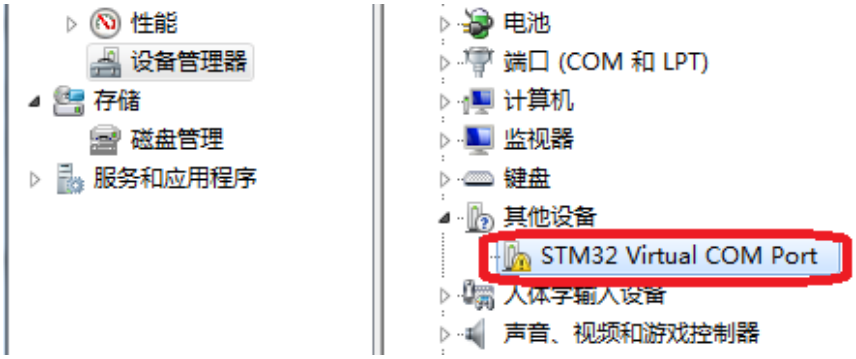


图 63.4.1 自动安装失败

此时，我们可手动选择驱动（以 WIN7 为例），进行安装，在如图 63.4.1 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 63.4.2 所示：

图 63.4.2 发现 STM32 USB 虚拟串口

如图 63.4.2，STM32 通过 USB 虚拟的串口，被电脑识别了，端口号为：COM15（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 63.4.3 所示：

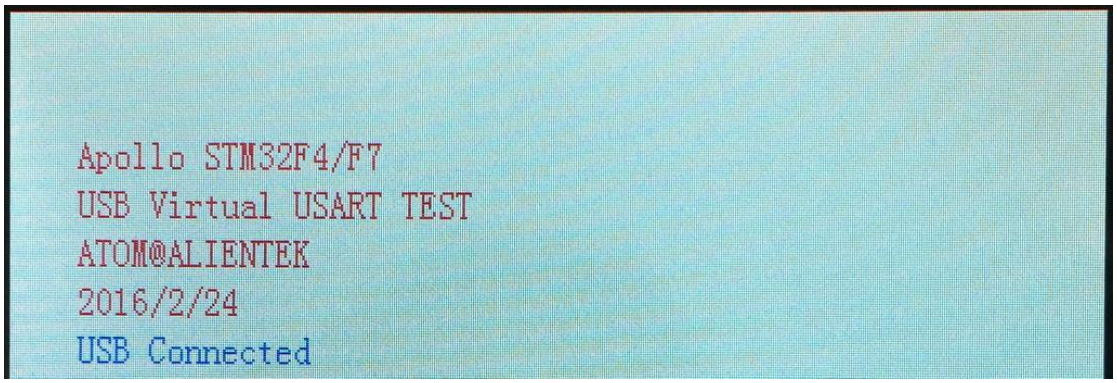


图 63.4.3 USB 虚拟串口连接成功

然后我们打开 XCOM，选择 COM15（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 63.4.4 所示：

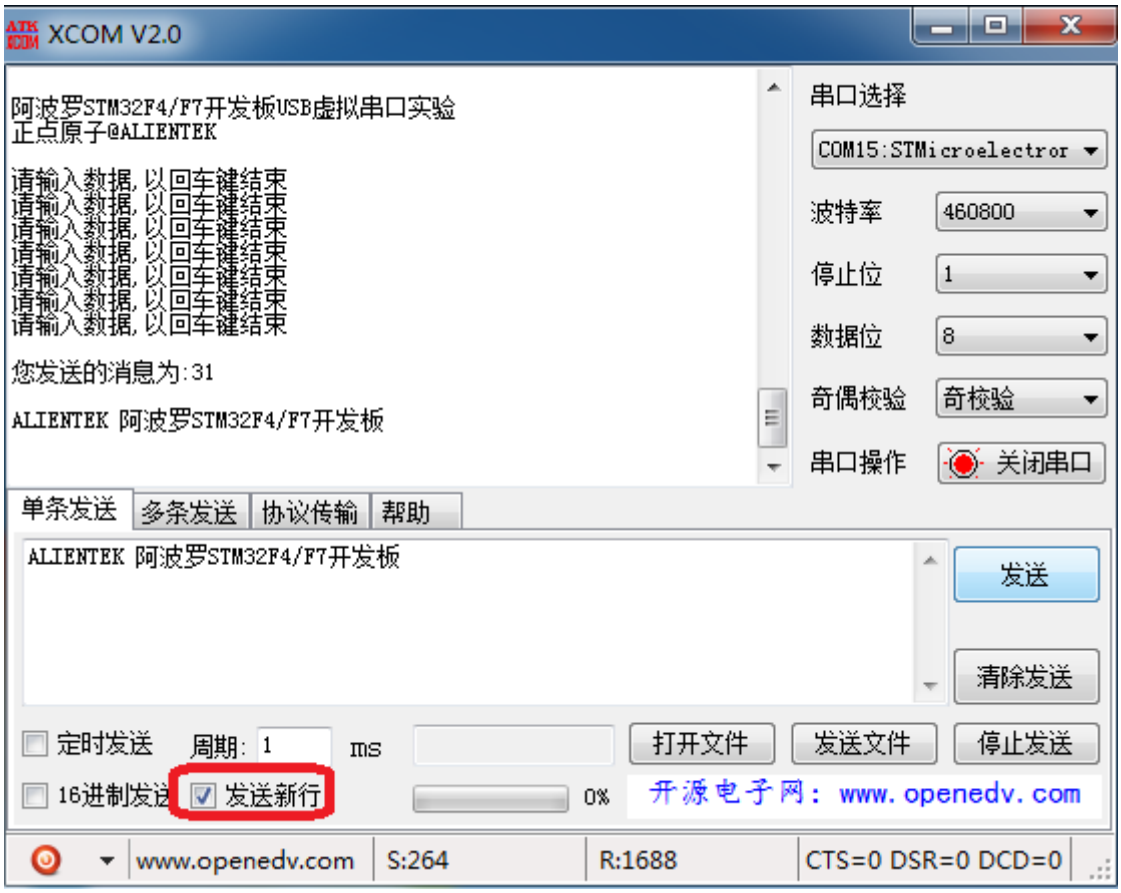


图 63.4.4 STM32 虚拟串口通信测试

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。实验现象同第八章完全一样。

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。

第六十四章 USB U 盘(Host)实验

前面两章，我们介绍了 STM32F767 的 USB SLAVE 应用，本章我们介绍 STM32F767 的 USB HOST 应用，即通过 USB HOST 功能，实现读写 U 盘/读卡器等大容量 USB 存储设备。本章分为如下几个部分：

64.1 U 盘简介
64.2 硬件设计
64.3 软件设计
64.4 下载验证

64.1 U 盘简介

U 盘，全称 USB 闪存盘，英文名“USB flash disk”。它是一种使用 USB 接口的无需物理驱动器的微型高容量移动存储产品，通过 USB 接口与主机连接，实现即插即用，是最常用的移动存储设备之一。

STM32F767 的 USB OTG FS 支持 U 盘，并且 ST 官方提供了 USB HOST 大容量存储设备（MSC）例程，ST 官方例程路径：光盘→8，STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.2.0→Project→USB_Host_Examples→MSC。本章代码，我们就要移植该例程到阿波罗 STM32 开发板上，以通过 STM32F767 的 USB HOST 接口，读写 U 盘或 SD 卡读卡器等设备。

64.2 硬件设计

本章实验功能简介：开机后，检测字库，然后初始化 USB HOST，并不断轮询。当检测并识别 U 盘后，在 LCD 上面显示 U 盘总容量和剩余容量，此时便可以通过 USMART 调用 FATFS 相关函数，来测试 U 盘数据的读写了，方法同 FATFS 实验一模一样。当 U 盘没插入的时候，DS0 闪烁，提示程序运行，当 U 盘插入后，DS1 闪烁，提示可以通过 USMART 测试了。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1。
2） 串口
3） LCD 模块
4） SPI FLASH
5） USB HOST 接口

前面 4 部分，在之前的实例中都介绍过了，我们在此就不介绍了。接下来看看我们电脑 USB 与 STM32 的 USB HOST 连接口。

ALIENTEK 阿波罗 STM32 开发板的 USB HOST 接口采用的是侧式 USB-A 座，它和 USB SLAVE 的 5PIN MiniUSB 接头是共用 USB_DM 和 USB_DP 信号的，所以 USB HOST 和 USB SLAVE 不能同时使用。USB HOST 同 STM32 的连接原理图，如图 64.2.1 所示：

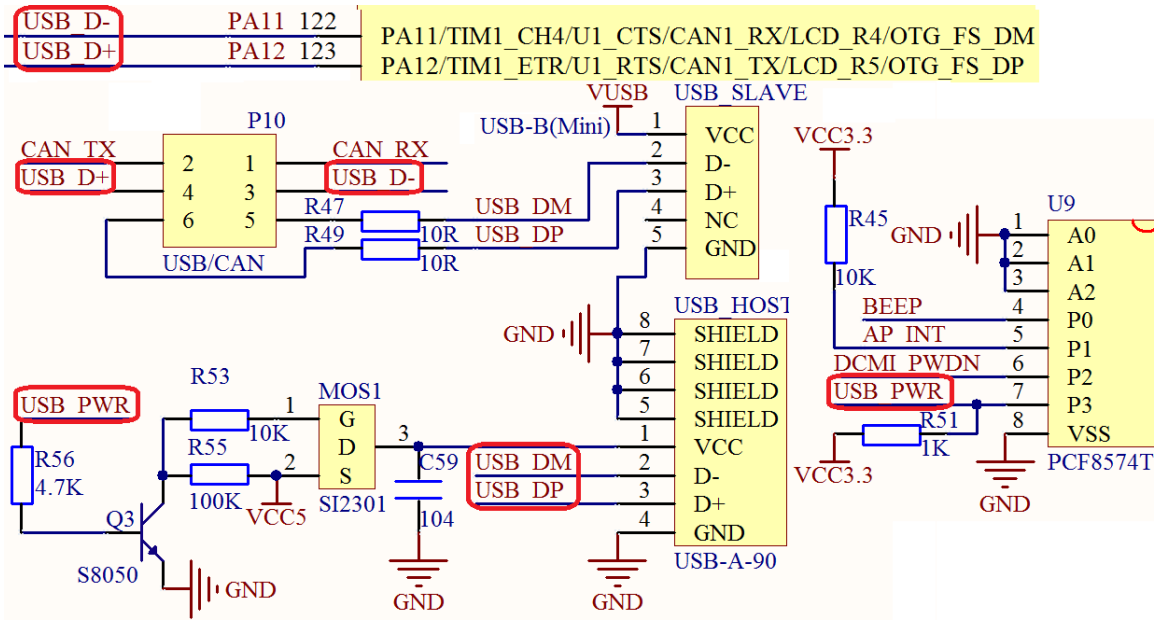


图 64.2.1 USB HOST 接口与 STM32F767 的连接原理图

从上图可以看出，USB_HOST 和 USB_SLAVE 共用 USB_DM/DP 信号，通过 P10 连接到 STM32F767。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 64.2.2 所示：

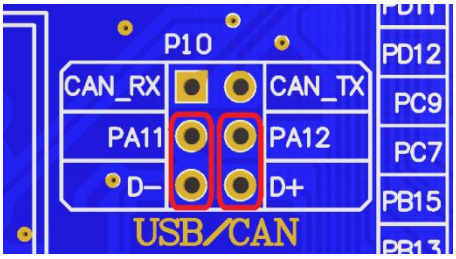


图 64.2.2 硬件连接示意图

图 64.2.1 中，我们还有一个 USB_PWR 的控制信号，用于控制给 USB 设备供电，该信号连接在 PCF8574T 的 P3 口上面，通过 PCF874T 进行间接控制。PCF8574T 的使用说明见第三十一章 IO 扩展实验。

使用 USB HOST 驱动外部 USB 设备的时候，必须要先控制 USB_PWR 输出 1，给外部设备供电，之后才可以识别到外部设备！

64.3 软件设计

本章，我们在：实验 44 图片显示实验 的基础上修改，代码移植自 ST 官方例程：STM32_USB-Host-Device_Lib_V2.2.0\Project\USB_Host_Examples\MSC，我打开该例程（用 IAR 打开）即可知道 USB 相关的代码有哪些，如图 64.3.1 所示：

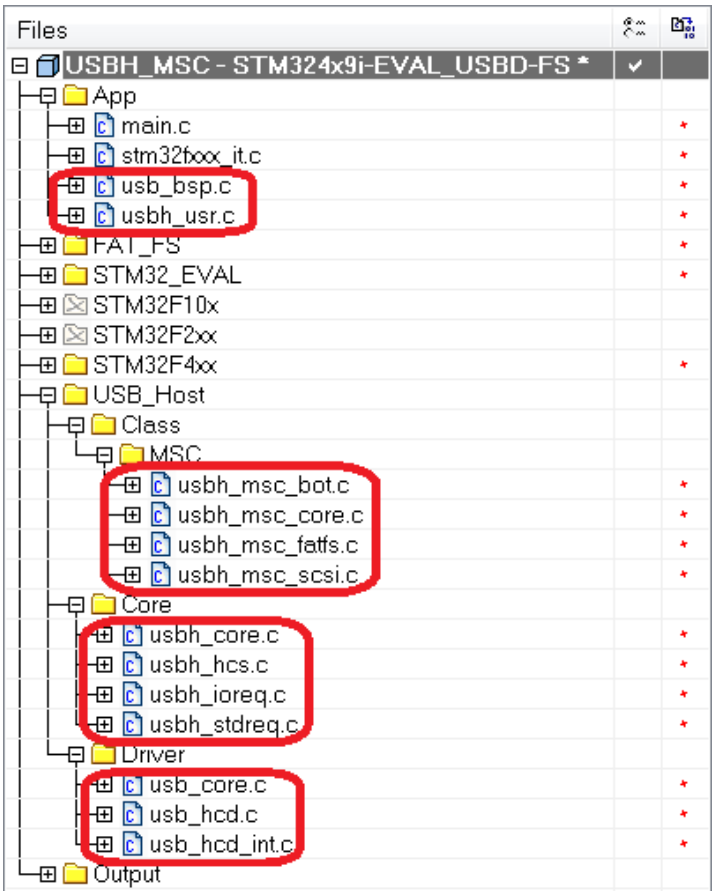


图 64.3.1 ST 官方例程 USB 相关代码

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。

这里面 usbh_msc_fatfs.c，是为了支持 fatfs 而写的一些底层接口函数，我们例程就直接放到 diskio.c 里面了，方便统一管理。

本例程的具体移植步骤，我们这里就不一一介绍了，最终移植好之后的工程截图，如图 64.3.2 所示：

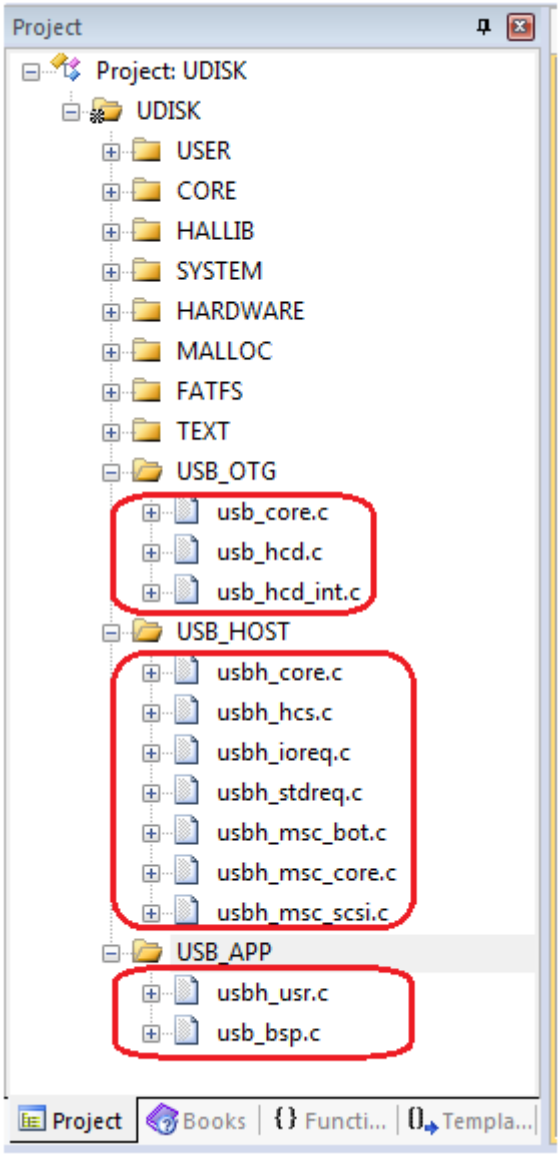


图 64.3.2 添加 USB 驱动等相关代码

注意：为了支持 STM32F7，USB OTG 库部分代码要做修改，详见 61.3 节的介绍（USB HOST 实验只需要修改 usb_core.c 这一个文件就可以支持 STM32F7 了）。

移植时，我们重点要修改的就是 USB_APP 文件夹下面的代码。其他代码（USB_OTG 和 USB_HOST 文件夹下的代码）一般不用修改。

usb_bsp.c 的代码，和上一章的一样，可以用上一章的代码直接替换即可正常使用。

usbh_usr.c 提供用户应用层接口函数，相比前两章例程，USB HOST 通信的回调函数更多一些，这里重点介绍 3 个函数，代码如下：

```
extern u8 USH_User_App(void);          //用户测试主程序
//USB HOST MSC 类用户应用程序
int USBH_USR_MSC_Application(void)
{
    u8 res=0;
    switch(AppState)
    {
```

```
            case USH_USR_FS_INIT://初始化文件系统
                printf("开始执行用户程序!!!\r\n");
                AppState=USH_USR_FS_TEST;
                break;
            case USH_USR_FS_TEST:   //执行 USB OTG  测试主程序
                res=USH_User_App(); //用户主程序
                if(res)AppState=USH_USR_FS_INIT;
                break;
            default:break;
        }
        return res;
}
//用户定义函数,实现 fatfs diskio 的接口函数
extern USBH_HOST            USB_Host;
//读 U 盘
//buf:读数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 USBH_UDISK_Read(u8* buf,u32 sector,u32 cnt)
{
        u8 res=1;
        if(HCD_IsDeviceConnected(&USB_OTG_Core)&&AppState==USH_USR_FS_TEST)
        //连接还存在,且是 APP 测试状态
        {
            do
            {
                res=USBH_MSC_Read10(&USB_OTG_Core,buf,sector,512*cnt);
                USBH_MSC_HandleBOTXfer(&USB_OTG_Core ,&USB_Host);
                if(!HCD_IsDeviceConnected(&USB_OTG_Core))
                {
                    res=1;//读写错误
                    break;
                };
            }while(res==USBH_MSC_BUSY);
        }else res=1;
        if(res==USBH_MSC_OK)res=0;
        return res;
}
//写 U 盘
//buf:写数据缓存区
//sector:扇区地址
//cnt:扇区个数
```

```
//返回值:错误状态;0,正常;其他,错误代码;
u8 USBH_UDISK_Write(u8* buf,u32 sector,u32 cnt)
{
    u8 res=1;
    if(HCD_IsDeviceConnected(&USB_OTG_Core)&&AppState==USH_USR_FS_TEST)
    //连接还存在,且是 APP 测试状态
    {
        do
        {
            res=USBH_MSC_Write10(&USB_OTG_Core,buf,sector,512*cnt);
            USBH_MSC_HandleBOTXfer(&USB_OTG_Core ,&USB_Host);
            if(!HCD_IsDeviceConnected(&USB_OTG_Core))
            {
                res=1;//读写错误
                break;
            };
        }while(res==USBH_MSC_BUSY);
    }else res=1;
    if(res==USBH_MSC_OK)res=0;
    return res;
}
```

其中，USBH_USR_MSC_Application 函数通过状态机的方式，处理相关事务，执行到这个函数，说明 U 盘已经被成功识别了，此时用户可以执行一些自己想要做的事情，比如读取 U 盘文件什么的，这里我们直接进入到 USH_User_App 函数，执行各种处理，后续会介绍该函数。

USBH_UDISK_Read 和 USBH_UDISK_Write 这两个函数，用于 U 盘读写，从指定扇区地址读写指定个数的扇区数据，这两个函数，再配合 fatfs，即可实现对 U 盘的文件读写访问。

其他代码，我们就不详细讲解了，请大家参考光盘本例程源码，最后看看 main.c 代码如下：

```
USBH_HOST    USB_Host;
USB_OTG_CORE_HANDLE    USB_OTG_Core;
//用户测试主程序
//返回值:0,正常
//          1,有问题
u8 USH_User_App(void)
{
    u8 led1sta=1; u8 res=0;  u32 total,free;
    Show_Str(30,140,200,16,"设备连接成功!.",16,0);
    f_mount(fs[3],"3:",1);    //重新挂载 U 盘
    res=exf_getfree("3:",&total,&free);
    if(res==0)
    {
        POINT_COLOR=BLUE;//设置字体为蓝色
        LCD_ShowString(30,160,200,16,16,"FATFS OK!");
        LCD_ShowString(30,180,200,16,16,"U Disk Total Size:        MB");
```

```
            LCD_ShowString(30,200,200,16,16,"U Disk    Free Size:         MB");
            LCD_ShowNum(174,180,total>>10,5,16);//显示 U 盘总容量  MB
            LCD_ShowNum(174,200,free>>10,5,16);
        }
        while(HCD_IsDeviceConnected(&USB_OTG_Core))//设备连接成功
        {
            LED1(led1sta^=1);
            delay_ms(200);
        }
        LED1(1);              //关闭 LED1
        f_mount(0,"3:",1);  //卸载 U 盘
        POINT_COLOR=RED;//设置字体为红色
        Show_Str(30,140,200,16,"设备连接中...",16,0);
        LCD_Fill(30,160,239,220,WHITE);
        return res;
}
int main(void)
{
        u8 t;

        Cache_Enable();                 //打开 L1-Cache
        HAL_Init();                     //初始化 HAL 库
        Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
        delay_init(216);                //延时初始化
        uart_init(115200);              //串口初始化
        LED_Init();                     //初始化 LED
        KEY_Init();                     //初始化按键
        SDRAM_Init();                   //初始化 SDRAM
        LCD_Init();                     //初始化 LCD
        W25QXX_Init();                  //初始化 W25Q256
        PCF8574_Init();                 //初始化 PCF8574
        my_mem_init(SRAMIN);            //初始化内部内存池
        my_mem_init(SRAMEX);            //初始化外部内存池
        my_mem_init(SRAMDTCM);          //初始化 DTCM 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 SPI FLASH.
        f_mount(fs[2],"2:",1);          //挂载 NAND FLASH.
        POINT_COLOR=RED;
        while(font_init())              //检查字库
        {
            LCD_ShowString(60,50,200,16,16,"Font Error!");
            delay_ms(200);
```

```
            LCD_Fill(60,50,240,66,WHITE);//清除显示
            delay_ms(200);
        }
        Show_Str(30,50,200,16,"阿波罗 STM32F4/F7 开发板",16,0);
        Show_Str(30,70,200,16,"USB U 盘实验",16,0);
        Show_Str(30,90,200,16,"2016 年 8 月 11 日",16,0);
        Show_Str(30,110,200,16,"正点原子@ALIENTEK",16,0);
        Show_Str(30,140,200,16,"设备连接中...",16,0);
        //初始化 USB 主机
        USBH_Init(&USB_OTG_Core,USB_OTG_FS_CORE_ID,&USB_Host,
                                                    &USBH_MSC_cb,&USR_cb);
        while(1)
        {
            USBH_Process(&USB_OTG_Core, &USB_Host);
            delay_ms(1);
            t++;
            if(t==200)
            {
                LED0_Toggle;
                t=0;
            }
        }
    }
```

相比 USB SLAVE 例程，我们这里多了一个 USB_HOST 的结构体定义：USB_Host，用于存储主机相关状态。所以，使用 USB 主机的时候，需要两个结构体：USB_OTG_CORE_HANDLE 和 USB_HOST。

然后，USB 初始化，使用的是 USBH_Init，用于 USB 主机初始化，包括对 USB 硬件和 USB 驱动库的初始化。如果是：USB SLAVE 通信，则只需要调用 USBD_Init 函数即可，不过 USB HOST 则还需要调用另外一个函数 USBH_Process，该函数用于实现 USB 主机通信的核心状态机处理，该函数必须在主函数里面，被循环调用，而且调用频率得比较快才行（越快越好），以便及时处理各种事务。注意，USBH_Process 函数仅在 U 盘识别阶段，需要频繁反复调用，但是当 U 盘被识别后，剩下的操作（U 盘读写），都可以由 USB 中断处理。

以上代码，main 函数十分简单，就不多做介绍了，这里主要看看 USH_User_App 函数，该函数前面有提到，是在 USBH_USR_MSC_Application 函数里面被调用，用于实现 U 盘插入后，用户想要实现的功能，一旦进入到该函数，即表示 U 盘已经成功识别了，所以，函数里面提示设备连接成功，挂载 U 盘（U 盘盘符为 3，0：SD 卡，1：SPI FLASH，2：NAND FLASH）并读取 U 盘总容量和剩余容量，显示在 LCD 上面，然后，进入死循环，只要 USB 连接一直存在，则一直死循环，同时控制 LED1 闪烁，提示 U 盘已经准备好了。

当 U 盘拔出来后，卸载 U 盘，然后再次提示设备连接中，会到 main 函数死循环，等待 U 盘再次连上。

最后，我们需要将 FATFS 相关测试函数(mf_open/ mf_close 等函数)，加入 USMART 管理，这里同第四十七章（FATFS 实验）一模一样，可以参考第四十七章的方法操作。

软件设计部分，就给大家介绍到这里。

64.4 下载验证

在代码编译成功之后，我们下载到阿波罗 STM32 开发板上，然后在 USB_HOST 端子插入 U 盘/读卡器（带卡），**注意：此时 USB SLAVE 口不要插 USB 线到电脑，否则会干扰！！**

等 U 盘成功识别后，便可以看到 LCD 显示 U 盘容量等信息，如图 64.4.1 所示：

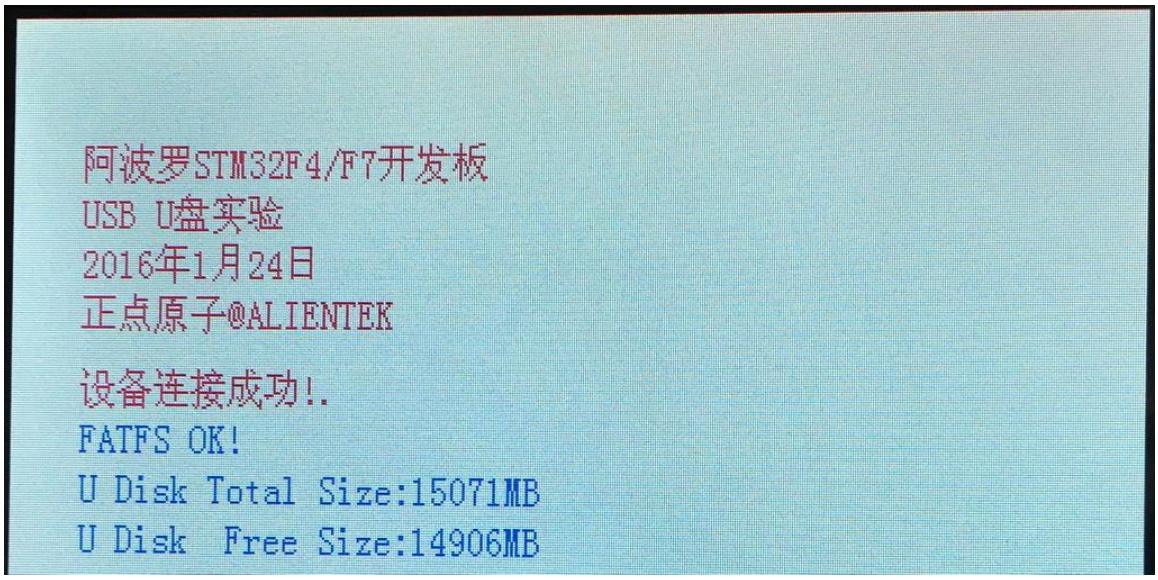


图 64.4.1 U 盘识别成功

此时，我们便可以通过 USMART 来测试 U 盘读写了，如图 64.4.2 和图 64.4.3 所示：

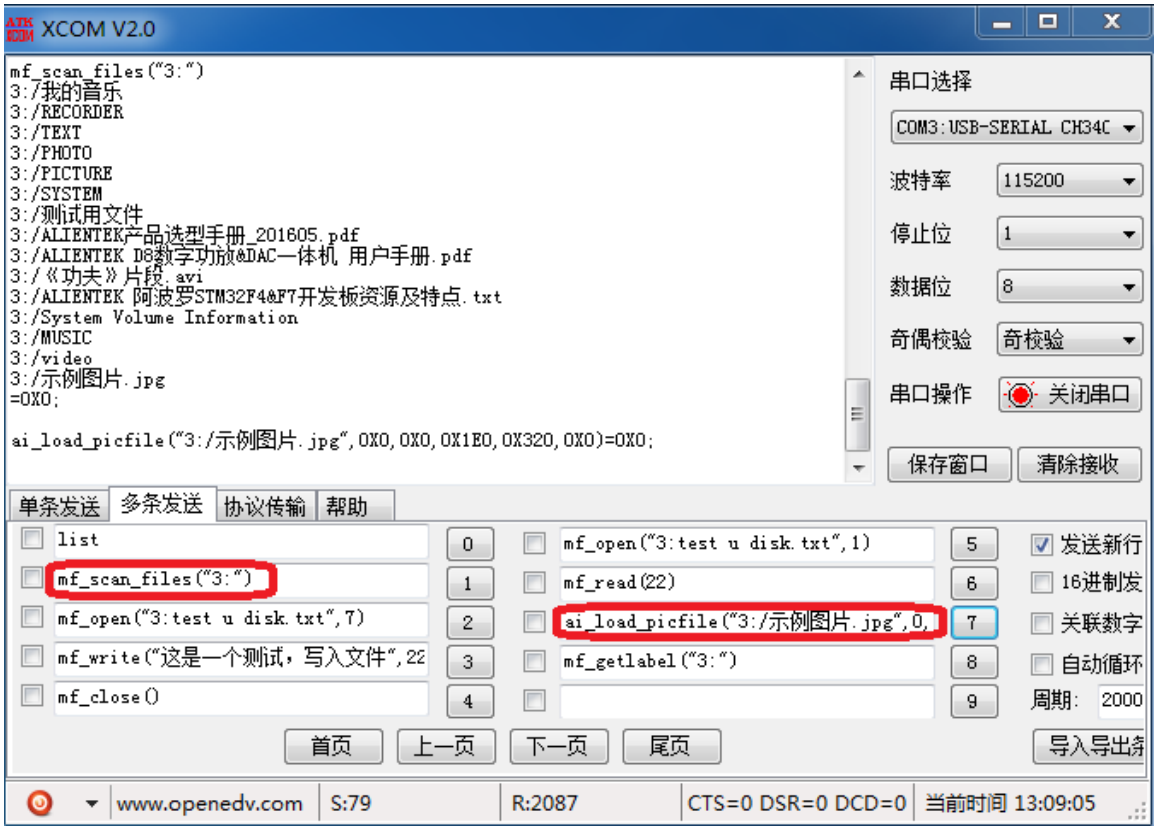


图 64.4.2 测试读取 U 盘读取

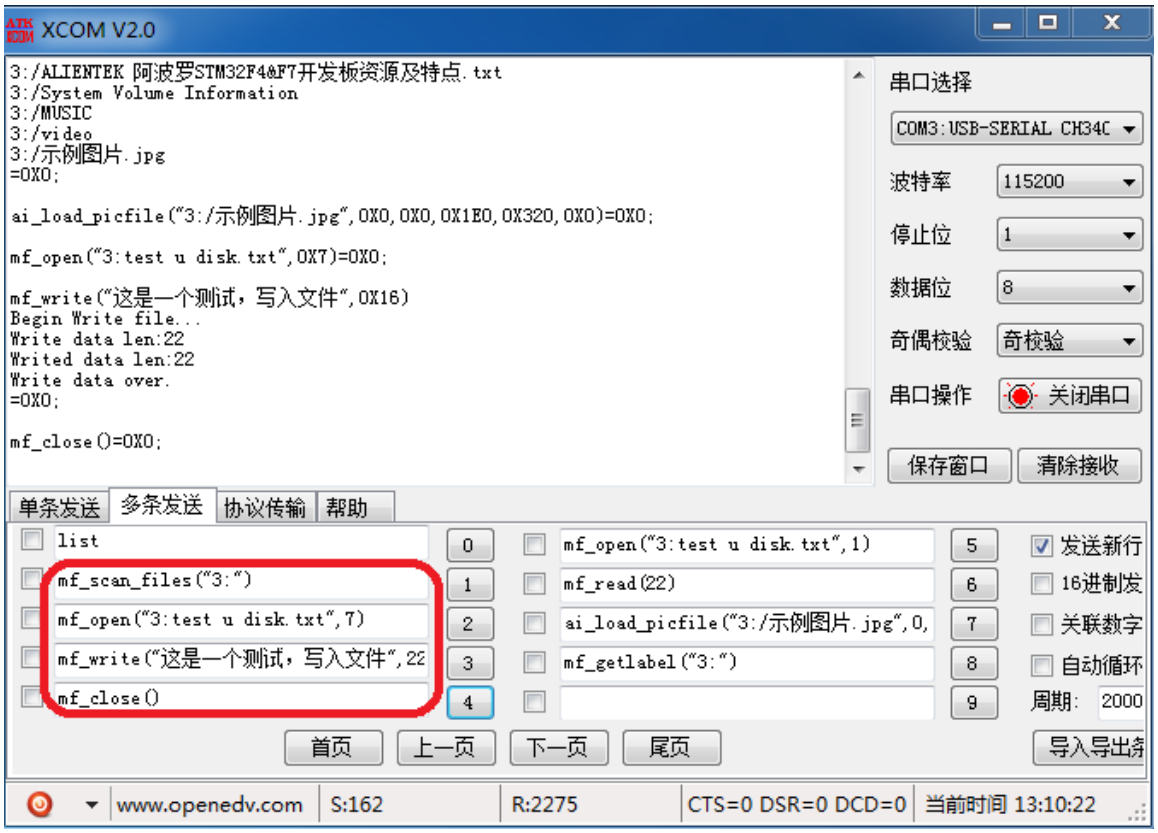


图 64.4.3 测试 U 盘写入

图 64.4.2 通过发送：mf_scan_files("3:")，扫描 U 盘根目录所有文件，然后通过 ai_load_picfile ("3:/示例图片.jpg",0,0,480,800,1)，解码图片，并显示在 LCD 上面。说明读 U 盘是没问题的。

图 64.4.3 通过发送：mf_open("3:test u disk.txt",7)，在 U 盘根目录创建 test u disk.txt 这个文件，然后发送：mf_write("这是一个测试，写入文件",22)，写入“这是一个测试，写入文件”到这个文件里面，然后发送：mf_close()，关闭文件，完成一次文件创建。最后，发送：mf_scan_files("3:")，扫描 U 盘根目录文件，发现比图 64.4.2 所示多出了一个 test u disk.txt 的文件，说明 U 盘写入成功。

这样，就完成了本实验的设计目的：实现 U 盘的读写操作。最后，大家还可以调用其他函数，实现相关功能测试，这里就不给大家一一演示了，测试方法同：FATFS 实验（第四十七章）。

第六十五章 USB 鼠标键盘(Host)实验

上一章我们向大家介绍了如何利用 STM32F767 的 USB HOST 接口来驱动 U 盘，本章，我们将利用 STM32F767 的 USB HOST 来驱动 USB 鼠标/键盘。本章分为如下几个部分：

65.1 USB 鼠标键盘简介
65.2 硬件设计
65.3 软件设计
65.4 下载验证

65.1 USB 鼠标键盘简介

传统的鼠标和键盘是采用 PS/2 接口和电脑通信的，但是现在 PS/2 接口在电脑上逐渐消失，所以现在越来越多的鼠标键盘采用的是 USB 接口，而不是 PS/2 接口的了。

USB 鼠标键盘属于 USB HID 设备。USB HID 即：Human Interface Device（人机交互设备）的缩写，键盘、鼠标与游戏杆等都属于此类设备。不过 HID 设备并不一定要有人机接口，只要符合 HID 类别规范的设备都是 HID 设备。关于 USB HID 的知识，我们这里就不详细介绍了，请大家自行百度学习。

本章，我们同上一章一样，我们直接移植官方的 USB HID 例程，官方例程路径：光盘→8，STM32 参考资料→STM32 USB 学习资料→STM32_USB-Host-Device_Lib_V2.2.0→Project→USB_Host_Examples→HID，该例程支持 USB 鼠标和键盘等 USB HID 设备，本章我们将移植这个例程到阿波罗 STM32 开发板上。

65.2 硬件设计

本节实验功能简介：开机的时候先显示一些提示信息，然后初始化 USB HOST，并不断轮询。当检测到 USB 鼠标/键盘的插入后，显示设备类型，并显示设备输入数据，

如果是 USB 鼠标：将显示鼠标移动的坐标（X，Y 坐标），滚轮滚动数值（Z 坐标）以及按键（左中右）。

如果是 USB 键盘：将显示键盘输入的数字/字母等内容（不是所有按键都支持，部分按键没有做解码支持，比如 F1~F12）。

最后，还是用 DS0 提示程序正在运行。

所要用到的硬件资源如下：

1） 指示灯 DS0
2） 串口
3） LCD 模块
4） USB HOST 接口

这几个部分，在之前的实例中都已经介绍过了，我们在此就不多说了。这里再次提醒大家，P10 的连接，要通过跳线帽连接 PA11 和 D-以及 PA12 和 D+。

65.3 软件设计

本章，我们在第二十章实验 （实验 15 LTDC LCD（RGB 屏）实验）的基础上修改，先打开实验 15 的工程，在 HARDWARE 文件夹所在文件夹下新建一个 USB 的文件夹，对照官方 HID 例子，将相关文件拷贝到 USB 文件夹下。

然后，我们在工程里面添加 USB HID 相关代码，最终得到如图 65.3.1 所示的工程：

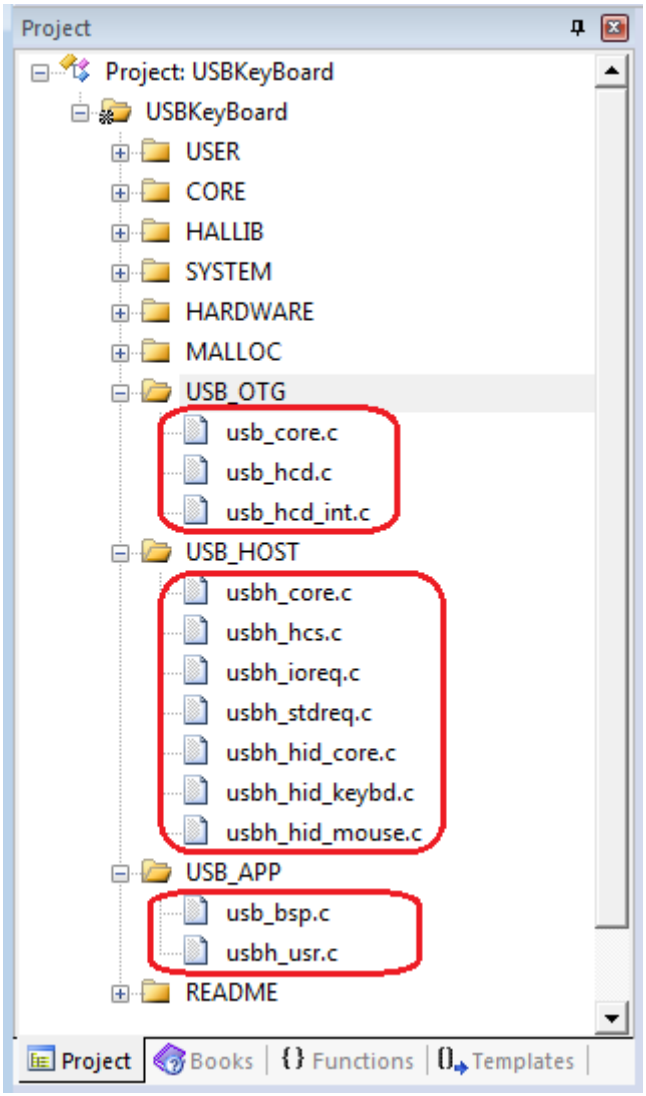


图 62.3.1 USB 鼠标键盘工程截图

注意：为了支持 STM32F7，USB OTG 库部分代码要做修改，详见 61.3 节的介绍（USB HOST 实验只需要修改 usb_core.c 这一个文件就可以支持 STM32F7 了）。

可以看到，USB 部分代码，同上一章的在结构上是一模一样的，只是.c 文件稍微有些变化。同样，我们移植需要修改的代码，就是 USB_APP 里面的这两个.c 文件了。

其中 usb_bsp.c 的代码，和之前的章节一模一样，可以用上一章的代码直接替换即可正常使用。

usbh_usr.c 里面的代码，则有所变化，重点代码如下：

```
//下面两个函数,为 ALIENTEK 添加,以防止 USB 死机
//USB 枚举状态死机检测,防止 USB 枚举失败导致的死机
//phost:USB_HOST 结构体指针
//返回值:0,没有死机
//          1,死机了,外部必须重新启动 USB 连接.
u8 USBH_Check_EnumeDead(USBH_HOST *phost)
{
```

```
    static u16 errcnt=0;
    //这个状态,如果持续存在,则说明 USB 死机了.
    if(phost->gState==HOST_CTRL_XFER&&(phost->EnumState==ENUM_IDLE||
       phost->EnumState==ENUM_GET_FULL_DEV_DESC))
    {
        errcnt++;
        if(errcnt>2000)//死机了
        {
            errcnt=0;
            RCC->AHB2RSTR|=1<<7;  //USB OTG FS  复位
            delay_ms(5);
            RCC->AHB2RSTR&=~(1<<7);   //复位结束
            return 1;
        }
    }else errcnt=0;
    return 0;
}
//USB HID 通信死机检测,防止 USB 通信死机(暂时仅针对:DTERR,即 Data toggle error)
//pcore:USB_OTG_Core_dev_HANDLE 结构体指针
//phidm:HID_Machine_TypeDef 结构体指针
//返回值:0,没有死机
//        1,死机了,外部必须重新启动 USB 连接.
u8 USBH_Check_HIDCommDead(USB_OTG_CORE_HANDLE *pcore,
                          HID_Machine_TypeDef *phidm)
{
    if(pcore->host.HC_Status[phidm->hc_num_in]==HC_DATATGLERR)//DTERR 错误
    {
        return 1;
    }
    return 0;
}
//USB 键盘鼠标数据处理
//鼠标初始化
void USR_MOUSE_Init(void)
{
    USBH_Msg_Show(2);       //USB  鼠标
    USB_FIRST_PLUGIN_FLAG=1;//标记第一次插入
}
//键盘初始化
void  USR_KEYBRD_Init(void)
{
    USBH_Msg_Show(1);       //USB  键盘
    USB_FIRST_PLUGIN_FLAG=1;//标记第一次插入
```

```
}
//临时数组,用于存放鼠标坐标/键盘输入内容(4.3 屏,最大可以输入 2016 字节)
__align(4) u8 tbuf[2017];
//USB 鼠标数据处理
//data:USB 鼠标数据结构体指针
void USR_MOUSE_ProcessData(HID_MOUSE_Data_TypeDef *data)
{
    static signed short x,y,z;
    if(USB_FIRST_PLUGIN_FLAG)//第一次插入,将数据清零
    {
        USB_FIRST_PLUGIN_FLAG=0;
        x=y=z=0;
    }
    x+=(signed char)data->x;
    if(x>9999)x=9999;
    if(x<-9999)x=-9999;
    y+=(signed char)data->y;
    if(y>9999)y=9999;
    if(y<-9999)y=-9999;
    z+=(signed char)data->z;
    if(z>9999)z=9999;
    if(z<-9999)z=-9999;
    POINT_COLOR=BLUE;
    sprintf((char*)tbuf,"BUTTON:");
    if(data->button&0X01)strcat((char*)tbuf,"LEFT");
    if((data->button&0X03)==0X02)strcat((char*)tbuf,"RIGHT");
    else if((data->button&0X03)==0X03)strcat((char*)tbuf,"+RIGHT");
    if((data->button&0X07)==0X04)strcat((char*)tbuf,"MID");
    else if((data->button&0X07)>0X04)strcat((char*)tbuf,"+MID");
    LCD_Fill(30+56,180,lcddev.width,180+16,WHITE);
    LCD_ShowString(30,180,210,16,16,tbuf);
    sprintf((char*)tbuf,"X POS:%05d",x); LCD_ShowString(30,200,200,16,16,tbuf);
    sprintf((char*)tbuf,"Y POS:%05d",y); LCD_ShowString(30,220,200,16,16,tbuf);
    sprintf((char*)tbuf,"Z POS:%05d",z); LCD_ShowString(30,240,200,16,16,tbuf);
}
//USB 键盘数据处理
//data:USB 键盘数据结构体指针
void   USR_KEYBRD_ProcessData (uint8_t data)
{
    static u16 pos;
    static u16 endx,endy;
    static u16 maxinputchar;
    u8 buf[4];
```

```
        if(USB_FIRST_PLUGIN_FLAG)//第一次插入,将数据清零
        {
            USB_FIRST_PLUGIN_FLAG=0;
            endx=((lcddev.width-30)/8)*8+30;        //得到 endx 值
            endy=((lcddev.height-220)/16)*16+220;       //得到 endy 值
            maxinputchar=((lcddev.width-30)/8);
            maxinputchar*=(lcddev.height-220)/16;//当前 LCD 最大可以显示的字符数.
            pos=0;
        }
        POINT_COLOR=BLUE;
        sprintf((char*)buf,"%02X",data);
        LCD_ShowString(30+56,180,200,16,16,buf);//显示键值
        if(data>=' '&&data<='~')
        {
            tbuf[pos++]=data;
            tbuf[pos]=0;            //添加结束符.
            if(pos>maxinputchar)pos=maxinputchar;//最大输入这么多
        }else if(data==0X0D)    //退格键
        {
            if(pos)pos--;
            tbuf[pos]=0;            //添加结束符.
        }
        if(pos<=maxinputchar)  //没有超过显示区
        {
            LCD_Fill(30,220,endx,endy,WHITE);
            LCD_ShowString(30,220,endx-30,endy-220,16,tbuf);
        }
    }
```

ST 官方的 USB HID 例程，仅仅是能用，很多地方还要改善，比如识别率低，容易死机（枚举/通信都可能死机）等问题，这里：USBH_Check_EnumeDead 和 USBH_Check_HIDCommDead 这两个函数，就是我们针对官方 HID 例程现有 bug 做出的改进处理，通过这两个函数，可以检测枚举/通信是否正常，当出现异常时，直接重启 USB 内核，重新连接设备，这样可以防止死机造成的程序无响应情况。

另外，为了提高对鼠标键盘的识别率和兼容性，对 usbh_hid_core.c 里面的两处代码进行了修改：

1，USBH_HID_ClassRequest 函数，修改代码（394 行）为：

```
classReqStatus = USBH_Set_Idle (pdev, pphost, 100, 0);//这里 duration 官方设置的是 0,修改为
                                                      //100,提高兼容性
```

2，USBH_Set_Idle 函数，修改代码（542 行）为：

```
phost->Control.setup.b.wLength.w = 100;    //官方的这里设置的是 0,导致部分鼠标无法识别,
                                           //这里修改为 100 以后,识别率明显提高.
```

以上两处地方，官方默认值都是设置的 0，我们修改为 100 后，可以明显提高 USB 鼠标/键盘的识别率，兼容性好很多。

还有，在 usbh_hid_keybd.h 里面，要修改键盘类型的定义，改为：

```
#define QWERTY_KEYBOARD        //通用键盘
//#define AZERTY_KEYBOARD      //法国版键盘
```

ST 官方例程，是使用的法国版键盘，一般我们国内用的是通用键盘，所以，需要换一个宏定义（换成：QWERTY_KEYBOARD）。

最后，在 usbh_hid_mouse.c 里面，MOUSE_Decode 函数用于鼠标数据解析，但是 ST 官方例程仅对 4 字节鼠标数据做了解析，而忽略了 5 字节/6 字节鼠标数据的处理，所以，需要修改该函数为：

```
extern HID_Machine_TypeDef HID_Machine;
static void    MOUSE_Decode(uint8_t *data)
{
    if(HID_Machine.length==5||HID_Machine.length==6||HID_Machine.length==8)
                                                    //5/6/8 字节长度的 USB 鼠标数据处理
    {
        HID_MOUSE_Data.button = data[0];
        HID_MOUSE_Data.x      = data[1];
        HID_MOUSE_Data.y      = data[3]<<4|data[2]>>4;
        HID_MOUSE_Data.z      = data[4];
    }else if(HID_Machine.length==4) //4 字节长度的 USB  鼠标数据处理
    {
        HID_MOUSE_Data.button = data[0];
        HID_MOUSE_Data.x      = data[1];
        HID_MOUSE_Data.y      = data[2];
        HID_MOUSE_Data.z      = data[3];
    }
    USR_MOUSE_ProcessData(&HID_MOUSE_Data);
}
```

再回到 usbh_usr.c，USR_MOUSE_Init 和 USR_MOUSE_ProcessData 用于处理鼠标数据，这两个函数在 usbh_hid_mouse.c 里面被调用，USR_MOUSE_Init 在鼠标初始化的时候被调用，而 USR_MOUSE_ProcessData 函数，则在鼠标初始化成功，轮询数据的时候调用，处理鼠标数据，该函数将得到的鼠标数据显示在 LCD 上面。

同样，USR_KEYBRD_Init 和 USR_KEYBRD_ProcessData 用于处理键盘数据，这两个函数在 usbh_hid_keybd.c 里面被调用，USR_KEYBRD_Init 在键盘初始化的时候被调用，而 USR_KEYBRD_ProcessData 函数，则在键盘初始化成功，轮询数据的时候调用，处理键盘数据，该函数将键盘输入的字符显示在 LCD 上面。

其他代码，我们就不再介绍了，请大家参考开发板光盘本例程源码。

最后，来看看 main.c 里面的代码，如下：

```
USBH_HOST    USB_Host;
USB_OTG_CORE_HANDLE    USB_OTG_Core_dev;
extern HID_Machine_TypeDef HID_Machine;

//USB 信息显示
//msgx:0,USB 无连接
```

```
//          1,USB 键盘
//          2,USB 鼠标
//          3,不支持的 USB 设备
void USBH_Msg_Show(u8 msgx)
{
    POINT_COLOR=RED;
    switch(msgx)
    {
        case 0:    //USB 无连接
            LCD_ShowString(30,130,200,16,16,"USB Connecting...");
            LCD_Fill(0,150,lcddev.width,lcddev.height,WHITE);
            break;
        case 1:    //USB 键盘
            LCD_ShowString(30,130,200,16,16,"USB Connected       ");
            LCD_ShowString(30,150,200,16,16,"USB KeyBoard");
            LCD_ShowString(30,180,210,16,16,"KEYVAL:");
            LCD_ShowString(30,200,210,16,16,"INPUT STRING:");
            break;
        case 2:    //USB 鼠标
            LCD_ShowString(30,130,200,16,16,"USB Connected       ");
            LCD_ShowString(30,150,200,16,16,"USB Mouse");
            LCD_ShowString(30,180,210,16,16,"BUTTON:");
            LCD_ShowString(30,200,210,16,16,"X POS:");
            LCD_ShowString(30,220,210,16,16,"Y POS:");
            LCD_ShowString(30,240,210,16,16,"Z POS:");
            break;
        case 3:    //不支持的 USB 设备
            LCD_ShowString(30,130,200,16,16,"USB Connected       ");
            LCD_ShowString(30,150,200,16,16,"Unknow Device");
            break;
    }
}
//HID 重新连接
void USBH_HID_Reconnect(void)
{
    //关闭之前的连接
    USBH_DeInit(&USB_OTG_Core_dev,&USB_Host); //复位 USB HOST
    USB_OTG_StopHost(&USB_OTG_Core_dev);        //停止 USBhost
    if(USB_Host.usr_cb->DeviceDisconnected)       //存在,才禁止
    {
        USB_Host.usr_cb->DeviceDisconnected();  //关闭 USB 连接
        USBH_DeInit(&USB_OTG_Core_dev, &USB_Host);
        USB_Host.usr_cb->DeInit();
```

```
            USB_Host.class_cb->DeInit(&USB_OTG_Core_dev,&USB_Host.device_prop);
        }
        USB_OTG_DisableGlobalInt(&USB_OTG_Core_dev);//关闭所有中断
        //重新复位 USB
        __HAL_RCC_USB_OTG_FS_FORCE_RESET();//USB OTG FS  复位
        delay_ms(5);
        __HAL_RCC_USB_OTG_FS_RELEASE_RESET();//复位结束
        memset(&USB_OTG_Core_dev,0,sizeof(USB_OTG_CORE_HANDLE));
        memset(&USB_Host,0,sizeof(USB_Host));
        //重新连接 USB HID 设备
        USBH_Init(&USB_OTG_Core_dev,USB_OTG_FS_CORE_ID,&USB_Host,
                                                &HID_cb,&USR_Callbacks);
}

int main(void)
{
        u8 t;

        Cache_Enable();                 //打开 L1-Cache
        HAL_Init();                     //初始化 HAL 库
        Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
        delay_init(216);                //延时初始化
        uart_init(115200);              //串口初始化
        LED_Init();                     //初始化 LED
        KEY_Init();                     //初始化按键
        SDRAM_Init();                   //初始化 SDRAM
        LCD_Init();                     //初始化 LCD
        W25QXX_Init();                  //初始化 W25Q256
        PCF8574_Init();                 //初始化 PCF8574
        POINT_COLOR=RED;
        LCD_ShowString(30,50,200,16,16,"Apollo STM32F4/F7");
        LCD_ShowString(30,70,200,16,16,"USB MOUSE/KEYBOARD TEST");
        LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(30,110,200,16,16,"2016/8/11");
        LCD_ShowString(30,130,200,16,16,"USB Connecting...");
        //初始化 USB 主机
        USBH_Init(&USB_OTG_Core_dev,USB_OTG_FS_CORE_ID,
                                        &USB_Host,&HID_cb,&USR_Callbacks);
        while(1)
        {
            USBH_Process(&USB_OTG_Core_dev, &USB_Host);
            if(bDeviceState==1)//连接建立了
            {
```

```
                if(USBH_Check_HIDCommDead(&USB_OTG_Core_dev,&HID_Machine))
                                                        //检测 USB HID 通信,是否还正常?
                {
                    USBH_HID_Reconnect();//重连
                }

            }else       //连接未建立的时候,检测
            {
                if(USBH_Check_EnumeDead(&USB_Host))
                                    //检测 USB HOST  枚举是否死机了?死机了,则重新初始化
                {
                    USBH_HID_Reconnect();//重连
                }
            }
            t++;
            if(t==200000)
            {
                LED0_Toggle;
                t=0;
            }
        }
    }
```

这里总共三个函数：USBH_Msg_Show 用于显示一些提示信息，在 usbh_usr.c 里面被相关函数调用。USBH_HID_Reconnect 则用于 USB HID 重新连接，当发现枚举/通信死机的时候，调用该函数实现 USB 复位重启，以重新连接；最后，main 函数就比较简单了，处理方式和上一章几乎一样，只是多了一些通信死机处理。

软件设计部分就为大家介绍到这里。

65.4 下载验证

在代码编译成功之后，我们下载到阿波罗 STM32 开发板上，然后在 USB_HOST 端子插入 USB 鼠标/键盘，**注意：此时 USB SLAVE 口不要插 USB 线到电脑，否则会干扰！！**

等 USB 鼠标/键盘成功识别后，便可以看到 LCD 显示 USB Connected，并显示设备类型：USB Mouse 或者 USB KeyBoard，同时也会显示输入的数据，如图 65.4.1 和图 65.4.2 所示：

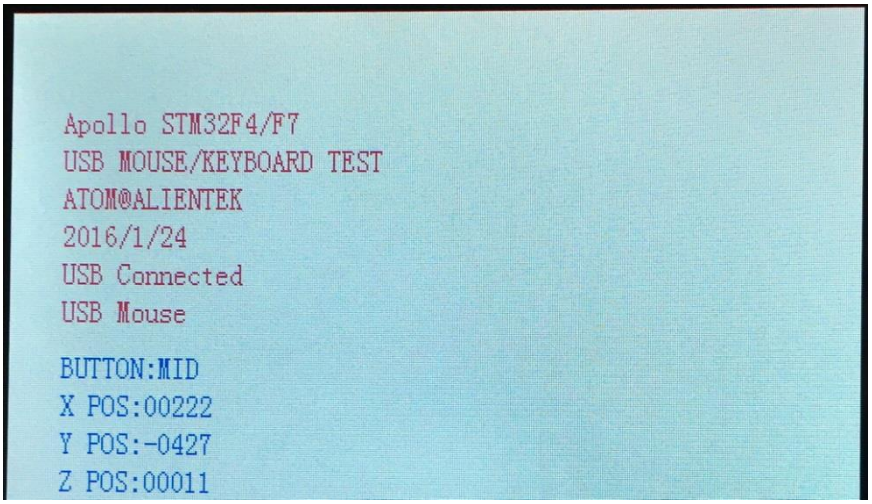


图 65.4.1 USB 鼠标测试

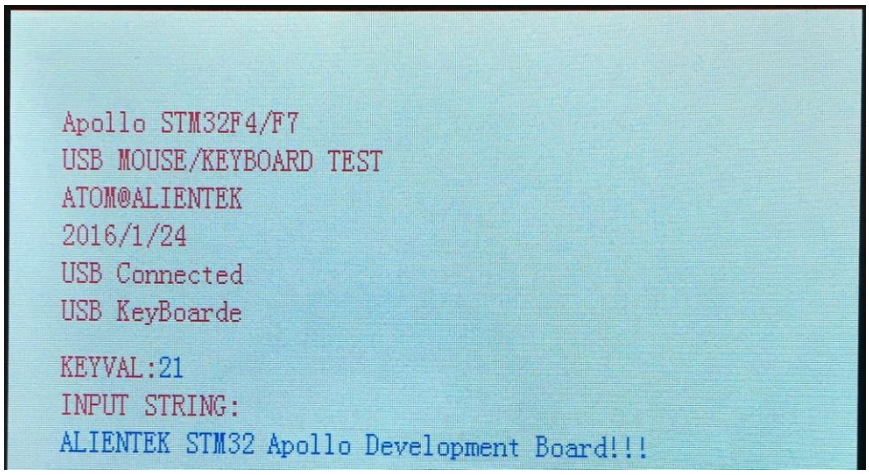


图 65.4.2 USB 键盘测试

其中，图 65.4.1 是 USB 鼠标测试界面，图 65.4.2 是 USB 键盘测试界面。

最后，特别提醒大家，由于例程的 HID 内核，只处理了第一个接口描述符，所以对于 USB 符合设备，只能识别第一个描述符所代表的设备。体现到实际使用中，就是：USB 无线鼠标，一般是无法使用（被识别为键盘），而 USB 无线键盘，可以使用，因为键盘在第一个描述符，鼠标在第二个描述符。

如果想支持 USB 无线鼠标，可以通过修改 usbh_hid_core.c 里面的 USBH_HID_InterfaceInit 函数来支持。

第六十六章 网络通信实验

本章，我们将向大家介绍阿波罗 STM32F767 开发板的网口及其使用。本章，我们将使用 ALIENTEK 阿波罗 STM32F767 开发板自带的网口和 LWIP 实现：TCP 服务器、TCP 客服端、UDP 以及 WEB 服务器等四个功能。本章分为如下几个部分：

66.1 STM32F767 以太网以及 TCP/IP LWIP 简介
66.2 硬件设计
66.3 软件设计
66.4 下载验证

66.1 STM32F767 以太网以及 TCP/IP LWIP 简介

本章，我们需要用到 STM32F767 的以太网控制器和 LWIP TCP/IP 协议栈。接下来分别介绍这两个部分。

66.1.1 STM32F767 以太网简介

STM32F767 芯片自带以太网模块，该模块包括带专用 DMA 控制器的 MAC 802.3（介质访问控制）控制器，支持介质独立接口 (MII) 和简化介质独立接口 (RMII)，并自带了一个用于外部 PHY 通信的 SMI 接口，通过一组配置寄存器，用户可以为 MAC 控制器和 DMA 控制器选择所需模式和功能。

STM32F767 自带以太网模块特点包括：

- 支持外部 PHY 接口，实现 10M/100Mbit/s 的数据传输速率
- 通过符合 IEEE802.3 的 MII/RMII 接口与外部以太网 PHY 进行通信
- 支持全双工和半双工操作
- 可编程帧长度，支持高达 16KB 巨型帧
- 可编程帧间隔（40~96 位时间，以 8 为步长）
- 支持多种灵活的地址过滤模式
- 通过 SMI（MDIO）接口配置和管理 PHY 设备
- 支持以太网时间戳（参见 IEEE1588-2008），提供 64 位时间戳
- 提供接收和发送两组 FIFO。
- 支持 DMA

STM32F767 以太网功能框图如图 66.1.1.1 所示：

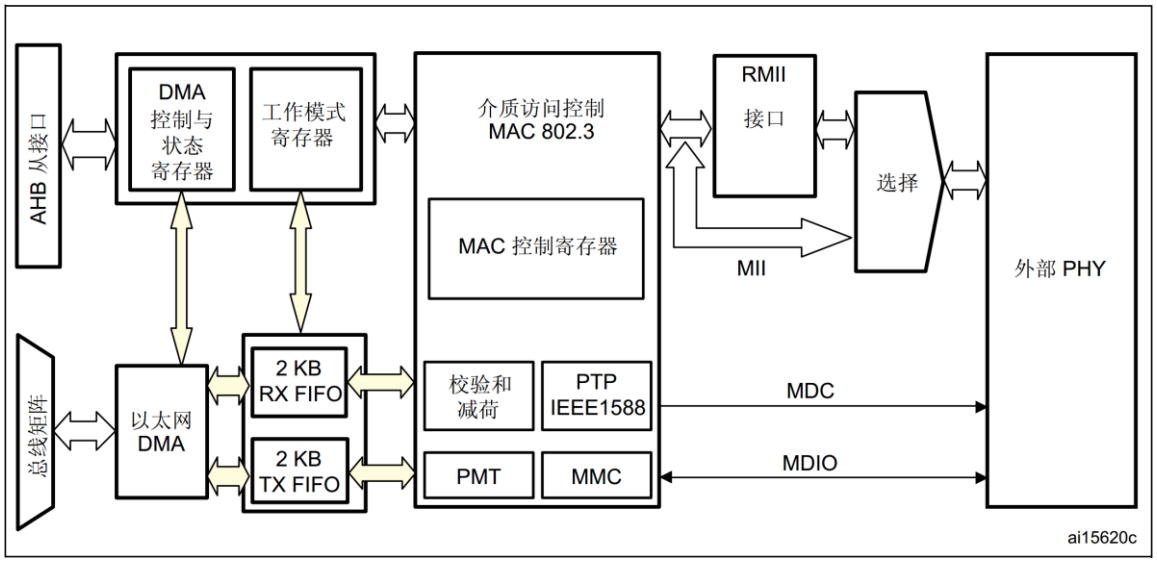


图 66.1.1.1 STM32F767 以太网框图

从上图可以看出，STM32F767 是必须外接 PHY 芯片，才可以完成以太网通信的，外部 PHY 芯片可以通过 MII/RMII 接口与 STM32F767 内部 MAC 连接，并且支持 SMI（MDIO&MDC）接口配置外部以太网 PHY 芯片。

接下来分别介绍 SMI/MII/RMII 接口和外部 PHY 芯片。

SMI 接口，即站管理接口，该接口允许应用程序通过 2 条线：时钟(MDC)和数据线(MDIO)访问任意 PHY 寄存器。该接口支持访问多达 32 个 PHY，应用程序可以从 32 个 PHY 中选择一个 PHY，然后从任意 PHY 包含的 32 个寄存器中选择一个寄存器，发送控制数据或接收状态信息。任意给定时间内只能对一个 PHY 中的一个寄存器进行寻址。

MII 接口，即介质独立接口，用于 MAC 层与 PHY 层进行数据传输。STM32F767 通过 MII 与 PHY 层芯片的连接如图 66.1.1.2 所示。

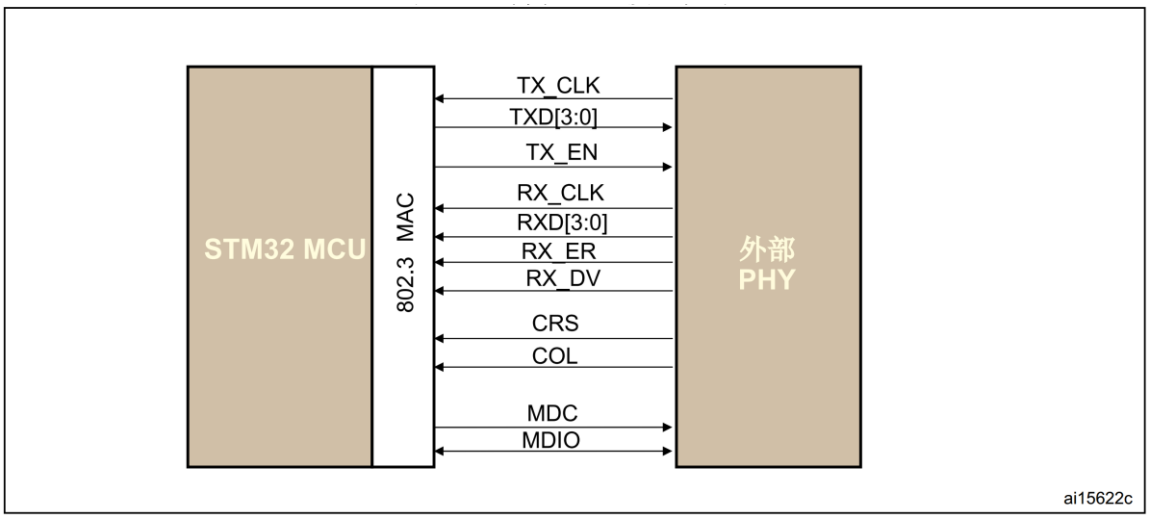


图 66.1.1.2 MII 接口信号

- MII_TX_CLK：连续时钟信号。该信号提供进行 TX 数据传输时的参考时序。标称频率为：速率为 10 Mbit/s 时为 2.5 MHz；速率为 100 Mbit/s 时为 25 MHz。
- MII_RX_CLK：连续时钟信号。该信号提供进行 RX 数据传输时的参考时序。标称频率为：速率为 10 Mbit/s 时为 2.5 MHz；速率为 100 Mbit/s 时为 25 MHz。

- MII_TX_EN：发送使能信号。
- MII_TXD[3:0]：数据发送信号。该信号是 4 个一组的数据信号，
- MII_CRS：载波侦听信号。
- MII_COL：冲突检测信号。
- MII_RXD[3:0]：数据接收信号。该信号是 4 个一组的数据信号
- MII_RX_DV：接收数据有效信号。
- MII_RX_ER：接收错误信号。该信号必须保持一个或多个周期(MII_RX_CLK)，从而向 MAC 子层指示在帧的某处检测到错误。

RMII 接口，即精简介质独立接口，该接口降低了在 10/100 Mbit/s 下微控制器以太网外设与外部 PHY 间的引脚数。根据 IEEE 802.3u 标准，MII 包括 16 个数据和控制信号的引脚。RMII 规范将引脚数减少为 7 个。

RMII 接口是 MAC 和 PHY 之间的实例化对象。这有助于将 MAC 的 MII 转换为 RMII。RMII 具有以下特性：

1，支持 10Mbit/s 和 100Mbit/s 的运行速率

2，参考时钟必须是 50 MHz

3，相同的参考时钟必须从外部提供给 MAC 和外部以太网 PHY

4，它提供了独立的 2 位宽（双位）的发送和接收数据路径

STM32F767 通过 RMII 接口与 PHY 层芯片的连接如图 66.1.1.3 所示：

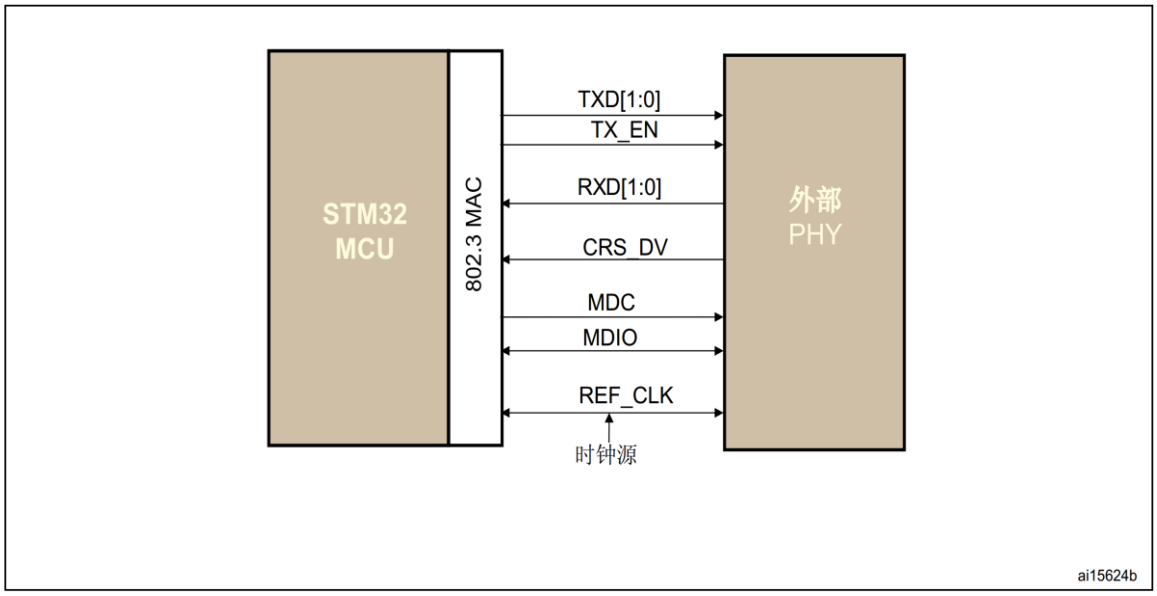


图 66.1.1.3 RMII 接口信号

从上图可以看出 RMII 相比 MII，引脚数量精简了不少。注意，图中的 REF_CLK 信号，是 RMII 和外部 PHY 共用的 50Mhz 参考时钟，必须由外部提供，比如有源晶振，或者 STM32F767 的 MCO 输出。不过有些 PHY 芯片可以自己产生 50Mhz 参考时钟，同时提供给 STM32F767，这样也是可以的。

本章我们采用 RMII 接口和外部 PHY 芯片连接，实现网络通信功能，阿波罗 STM32F767 开发板使用的是 LAN8720A 作为 PHY 芯片。接下来，我们简单介绍一下 LAN8720A 这个芯片。

LAN8720A 是低功耗的 10/100M 以太网 PHY 层芯片，I/O 引脚电压符合 IEEE802.3-2005 标准，支持通过 RMII 接口与以太网 MAC 层通信，内置 10-BASE-T/100BASE-TX 全双工传输模

块，支持 10Mbps 和 100Mbps。

LAN8720A 可以通过自协商的方式与目的主机最佳的连接方式(速度和双工模式)，支持 HP Auto-MDIX 自动翻转功能，无需更换网线即可将连接更改为直连或交叉连接。LAN8720A 的主要特点如下：

- 高性能的 10/100M 以太网传输模块
- 支持 RMII 接口以减少引脚数
- 支持全双工和半双工模式
- 两个状态 LED 输出
- 可以使用 25M 晶振以降低成本
- 支持自协商模式
- 支持 HP Auto-MDIX 自动翻转功能
- 支持 SMI 串行管理接口
- 支持 MAC 接口

LAN8720A 功能框图如图 66.1.1.4 所示。

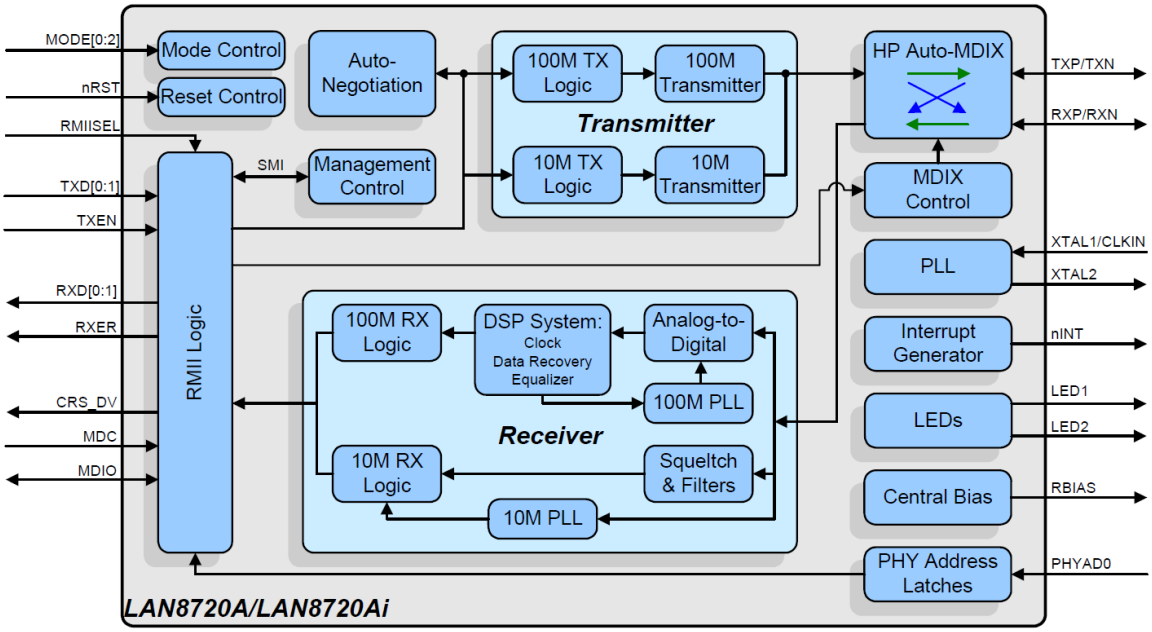


图 66.1.1.4 LAN8720A 功能框图

LAN8720A 的应脚数是比较少的，因此，很多引脚具有多个功能。这里，我们介绍几个重要的设置。

1，PHY 芯片地址设置

LAN8720A 可以通过 PHYAD0 引脚来配置，该引脚与 RXER 引脚复用，芯片内部自带下拉电阻，当硬复位结束后，LAN8720A 会读取该引脚电平，作为器件的 SMI 地址，接下拉电阻时（浮空也可以，因为芯片内部自带了下拉电阻），设置 SMI 地址为 0，当外接上拉电阻后，可以设置为 1。本章我们采用的是该引脚浮空，即设置 LAN8720 地址为 0。

2，nINT/REFCLKO 引脚功能配置

nINT/REFCLKO 引脚可以用作中断输出，或者参考时钟输出。通过 LED2（nINTSEL）引脚设置，LED2 引脚的值在芯片复位后，被 LAN8720A 读取，当该引脚接上拉电阻（或浮空，内置上拉电阻），那么正常工作后，nINT/REFCLKO 引脚将作为中断输出引脚（选中 REF_CLK IN 模式）。当该引脚接下拉电阻时，正常工作后，nINT/REFCLKO 引脚将作为参考时钟输出（选

中 REF_CLK OUT 模式）。

在 REF_CLK IN 模式，外部必须提供 50Mhz 参考时钟给 LAN8720A 的 XTAL1（CLKIN）引脚。

在 REF_CLK OUT 模式，LAN8720A 可以外接 25Mhz 石英晶振，通过内部倍频到 50Mhz，然后通过 REFCLKO 引脚，输出 50Mhz 参考时钟给 MAC 控制器。这种方式，可以降低 BOM 成本。

本章，我们设置 nINT/REFCLKO 引脚为参考时钟输出（REF_CLK OUT 模式），用于给 STM32F767 的 RMII 提供 50Mhz 参考时钟。

3，1.2V 内部稳压器配置

LAN8720A 需要 1.2V 电压给 VDDCR 供电，不过芯片内部集成了 1.2V 稳压器，可以通过 LED1(REGOFF)来配置是否使用内部稳压器，当不使用内部稳压器的时候，必须外部提供 1.2V 电压给 VDDCR 引脚。这里我们使用内部稳压器，所以我们在 LED1 接下拉电阻（浮空也行，内置了下拉电阻），以控制开启内部 1.2V 稳压器。

最后，我们来看下 LAN8720A 同我们阿波罗 STM32F767 开发板的连接关系，如图 66.1.1.5 所示：

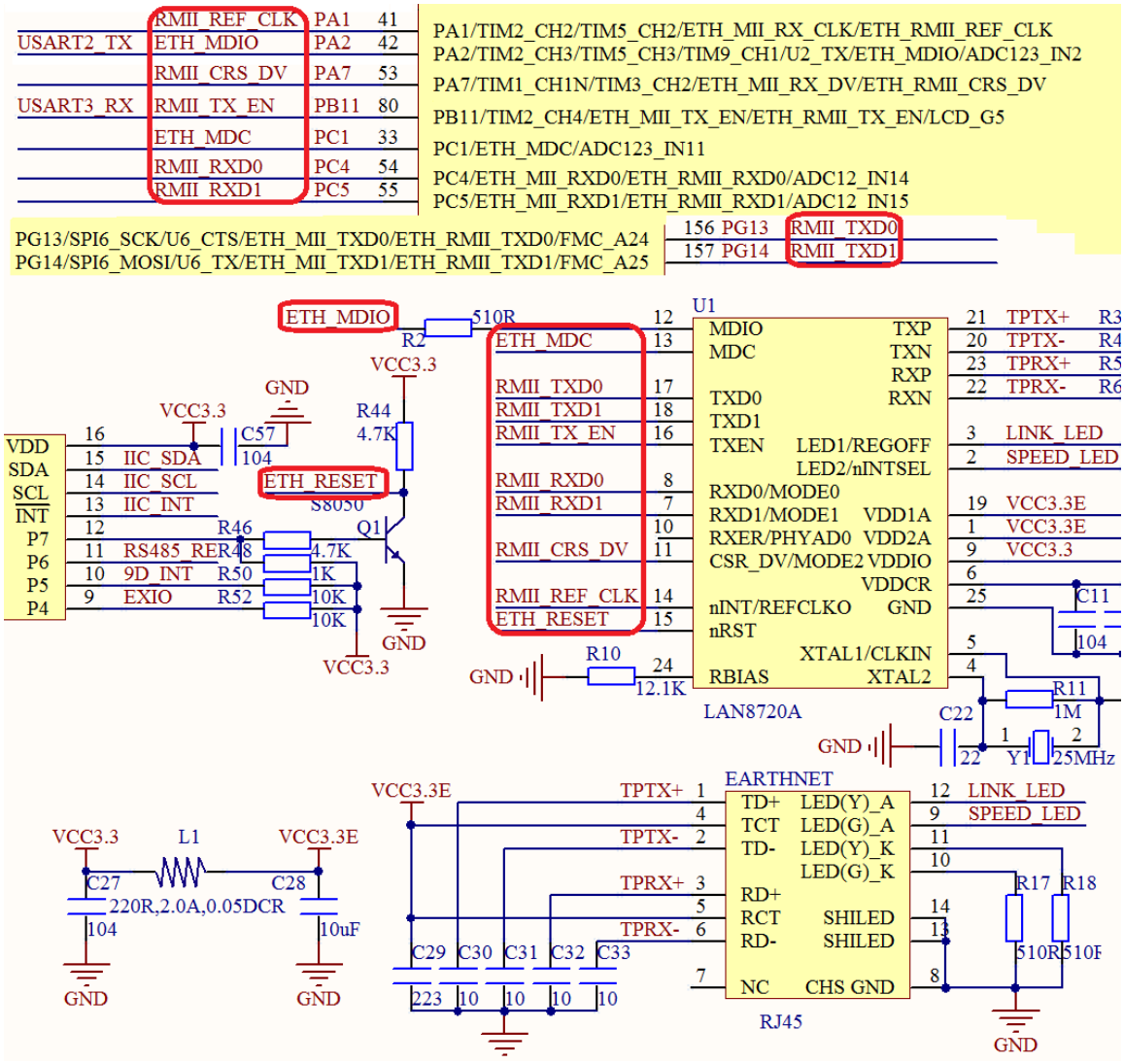


图 66.1.1.5 LAN8720A 与 STM32F767IGT6 连接原理图

从上图可以看出，LAN8720A 总共通过 10 跟线同 STM32F767 连接，注意：ETH_MDIO 和 USART2_TX 共用、RMII_TX_EN 和 USART3_RX 共用，所以他们不能同时使用，使用时需要注意这个问题。另外 LAN8720A 的 ETH_RESET 脚是连接在 PCF8574 的 P7 上面的（经过 Q1 取反），所以，使用网络功能的时候，必须配置 PCF8574 对 ETH_RESET 进行控制，才可以正常运行。

66.1.2 TCP/IP LWIP 简介

1，TCP/IP 简介

TCP/IP 中文名为传输控制协议/因特网互联协议，又名网络通讯协议，是 Internet 最基本的协议、Internet 国际互联网络的基础，由网络层的 IP 协议和传输层的 TCP 协议组成。TCP/IP 定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了 4 层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。通俗而言：TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而 IP 是给因特网的每一台联网设备规定一个地址。

TCP/IP 协议不是 TCP 和 IP 这两个协议的合称，而是指因特网整个 TCP/IP 协议族。从协议分层模型方面来讲，TCP/IP 由四个层次组成：网络接口层、网络层、传输层、应用层。OSI 是传统的开放式系统互连参考模型,该模型将 TCP/IP 分为七层: 物理层、数据链路层（网络接口层）、网络层（网络层）、传输层（传输层）、会话层、表示层和应用层（应用层）。TCP/IP 模型与 OSI 模型对比如表 66.1.2.1 所示。

编号	OSI 模型	TCP/IP 模型
1	应用层	应用层
2	表示层	
3	会话层	
4	传输层	传输层
5	网络层	互联层
6	数据链路层	链路层
7	物理层	

表 66.1.2.1 TCP/IP 模型与 OSI 模型对比

具体一点理解，本例程中的：PHY 层芯片 LAN8720A 相当于物理层，STM32F767 自带的 MAC 层相当于数据链路层，而 LWIP 提供的就是网络层、传输层的功能，应用层是需要用户自己根据自己想要的功能去实现的。

2，LWIP 简介

LWIP 是瑞典计算机科学院(SICS)的 Adam Dunkels 等开发的一个小型开源的 TCP/IP 协议栈，是 TCP/IP 的一种实现方式。LWIP 是轻量级 IP 协议，有无操作系统的支持都可以运行，LWIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LWIP 协议栈适合在低端的嵌入式系统中使用。 目前 LWIP 的最新版本是 1.4.1。本教程采用的就是 1.4.1 版本的 LWIP。

关于 LWIP 的详细信息大家可以去 http://savannah.nongnu.org/projects/lwip 这个网站去查阅，LWIP 的主要特性如下：

- ARP 协议，以太网地址解析协议；
- IP 协议，包括 IPv4 和 IPv6，支持 IP 分片与重装，支持多网络接口下数据转发；
- ICMP 协议，用于网络调试与维护；

- IGMP 协议，用于网络组管理，可以实现多播数据的接收；
- UDP 协议，用户数据报协议；
- TCP 协议，支持 TCP 拥塞控制，RTT 估计，快速恢复与重传等；
- 提供三种用户编程接口方式：raw/callback API、sequential API、BSD-style socket API；
- DNS，域名解析；
- SNMP，简单网络管理协议；
- DHCP，动态主机配置协议；
- AUTOIP，IP 地址自动配置；
- PPP，点对点协议，支持 PPPoE

我们从 LWIP 官网上下载 LWIP 1.4.1 版本，打开后如图 66.1.2.1 所示。

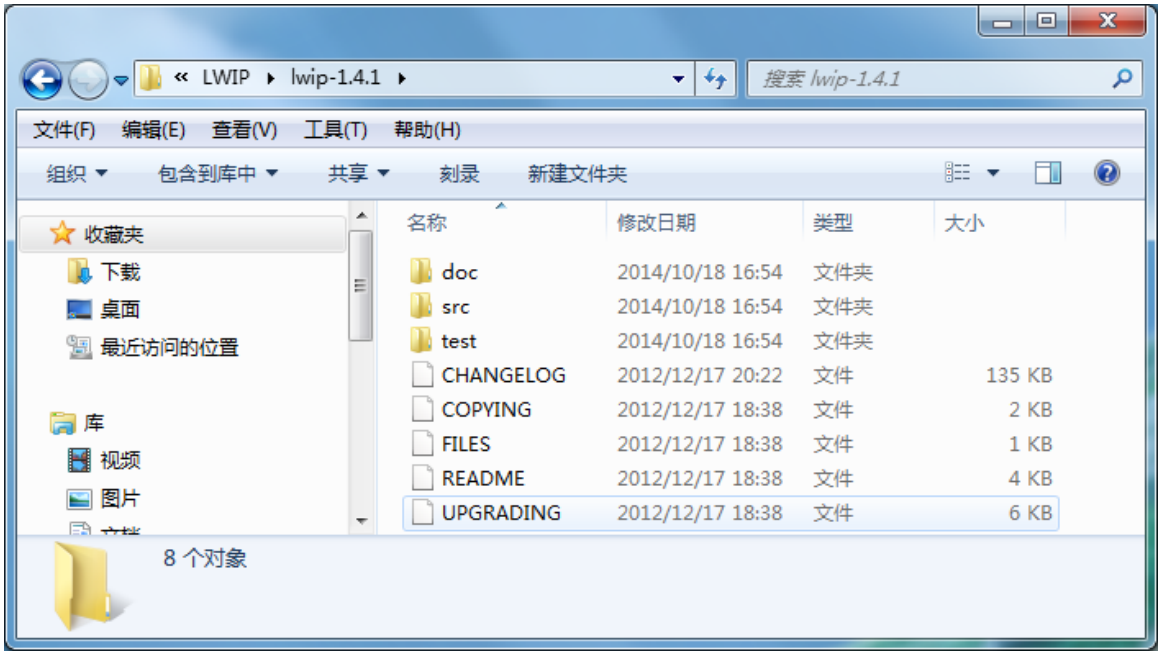


图 66.1.2.1　LWIP 1.4.1 源码内容

打开从官网上下载下来的 LWIP1.4.1 其中包括 doc，src 和 test 三个文件夹和 5 个其他文件。doc 文件夹下包含了几个与协议栈使用相关的文本文档，doc 文件夹里面有两个比较重要的文档:rawapi.txt 和 sys_arch.txt。

rawapi.txt 告诉读者怎么使用 raw/callback API 进行编程，sys_arch.txt 包含了移植说明，在移植的时候会用到。src 文件夹是我们的重点，里面包含了 LWIP 的源码。test 是 LWIP 提供的一些测试程序，方便大家使用 LWIP。打开 src 源码文件夹，如图 66.1.2.2 所示：

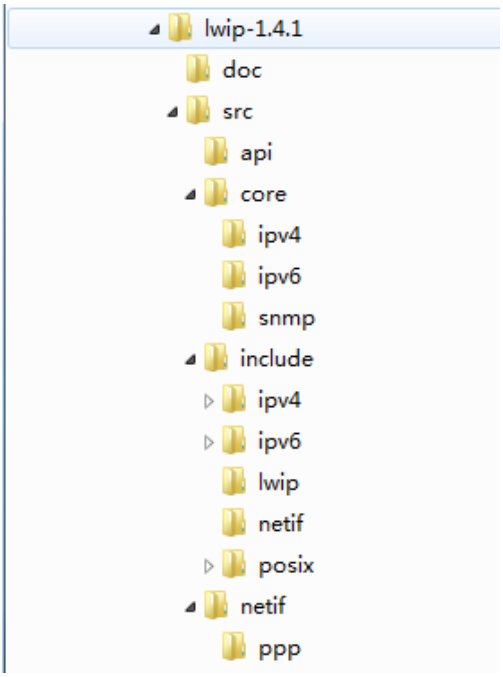


图 66.1.2.2　LWIP src 文件夹内容

src 文件夹由 4 个文件夹组成：api、core、include、netif 四个文件夹。api 文件夹里面是 LWIP 的 sequential API(Netconn)和 socket API 两种接口函数的源码，要使用这两种 API 需要操作系统支持。core 文件夹是 LWIP 内核源码，实现了各种协议支持，include 文件夹里面是 LWIP 使用到的头文件，netif 文件夹里面是与网络底层接口有关的文件。

关于 LWIP 的移植，请参考：《STM32F767 LWIP 开发手册.pdf》（文档路径：光盘 A 盘根目录）第一章，该文档详细介绍了 LWIP 在 STM32F767 上面的移植。这里我们就不详细介绍了。

66.2 硬件设计

本节实验功能简介：开机后，程序初始化 LWIP，包括：初始化 LAN8720A、申请内存、开启 DHCP 服务、添加并打开网卡，然后等待 DHCP 获取 IP 成功，当 DHCP 获取成功后，将在 LCD 屏幕上显示 DHCP 得到的 IP 地址，如果 DHCP 获取失败，那么将使用静态 IP（固定为：192.168.1.30），然后开启 Web Server 服务，并进入主循环，等待按键输入选择需要测试的功能：

KEY0 按键，用于选择 TCP Server 测试功能。

KEY1 按键，用于选择 TCP Client 测试功能

KEY2 按键，用于选择 UDP 测试功能

TCP Server 测试的时候，直接使用 DHCP 获取到的 IP（DHCP 失败，则使用静态 IP）作为服务器地址，端口号固定为：8088。在电脑端，可以使用网络调试助手（TCP Client 模式）连接开发板，连接成功后，屏幕显示连接上的 Client 的 IP 地址，此时便可以互相发送数据了。按 KEY0 发送数据给电脑，电脑端发送过来的数据将会显示在 LCD 屏幕上。按 KEY_UP 可以退出 TCP Server 测试。

TCP Client 测试的时候，先通过 KEY0/KEY2 来设置远端 IP 地址（Server 的 IP），端口号固定为：8087。设置好之后，通过 KEY_UP 确认，随后，开发板会不断尝试连接到所设置的远端 IP 地址（端口：8087），此时我们需要在电脑端使用网络调试助手(TCP Server 模式)，

设置端口为：8087，开启 TCP Server 服务，等待开发板连接。当连接成功后，测试方法同 TCP Server 测试的方法一样。

UDP 测试的时候，同 TCP Client 测试几乎一模一样，先通过 KEY0/KEY2 设置远端 IP 地址（电脑端的 IP），端口号固定为：8089，然后按 KEY_UP 确认。电脑端使用网络调试助手（UDP 模式），设置端口为：8089，开启 UDP 服务。不过对于 UDP 通信，我们得先按开发板 KEY0，发送一次数据给电脑，随后才可以电脑发送数据给开发板，实现数据互发。按 KEY_UP 可以退出 UDP 测试。

Web Server 的测试相对简单，只需要在浏览器端输入开发板的 IP 地址（DHCP 获取到的 IP 地址或者 DHCP 失败时使用的静态 IP 地址），即可登录一个 Web 界面，在 Web 界面，可以实现对 DS1(LED1)的控制、蜂鸣器的控制、查看 ADC1 通道 5 的值、内部温度传感器温度值以及查看 RTC 时间和日期等。

DS0 用于提示程序正在运行。

本例程所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1

2） 四个按键（KEY0/KEY1/KEY2/KEY_UP）

3） 串口

4） LCD 模块

5） ETH（STM32F767 自带以太网功能）

6） LAN8720A

7） PCF8574

这几个部分我们都已经详细介绍过了。本实验测试，需自备网线一根，路由器一个。

66.3 软件设计

本章，我们综合了 ALIENTEK《STM32F767 LWIP 开发手册.pdf》这个文档里面的 4 个 LWIP 基础例程：UDP 实验、TCP 客户端（TCP Client）实验、TCP 服务器（TCP Server）实验和 Web Server 实验。这些实验测试代码在工程 LWIP→lwip_app 文件夹下，如图 66.3.1 所示：

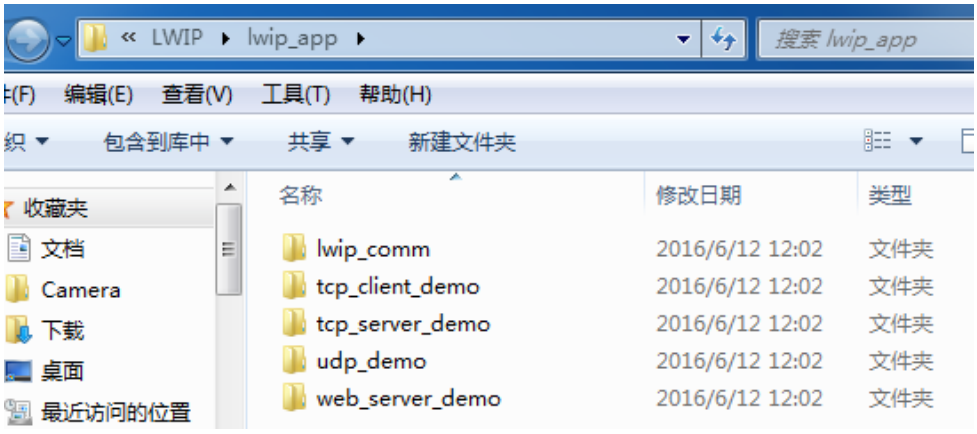


图 66.3.1 LWIP 文件夹内容

这里面总共 4 个文件夹：lwip_comm 文件夹，存放了 ALIENTEK 提供的 LWIP 扩展支持代码，方便使用和配置 LWIP，其他四个文件夹，则分别存放了 TCP Client、TCP Server、UDP 和 Web Server 测试 demo 程序。这里我们就不详细介绍这些内容了，详细的介绍，请参考：ALIENTEK《STM32F767 LWIP 开发手册.pdf》这个文档。本例程工程结构如图 66.3.2 所示：

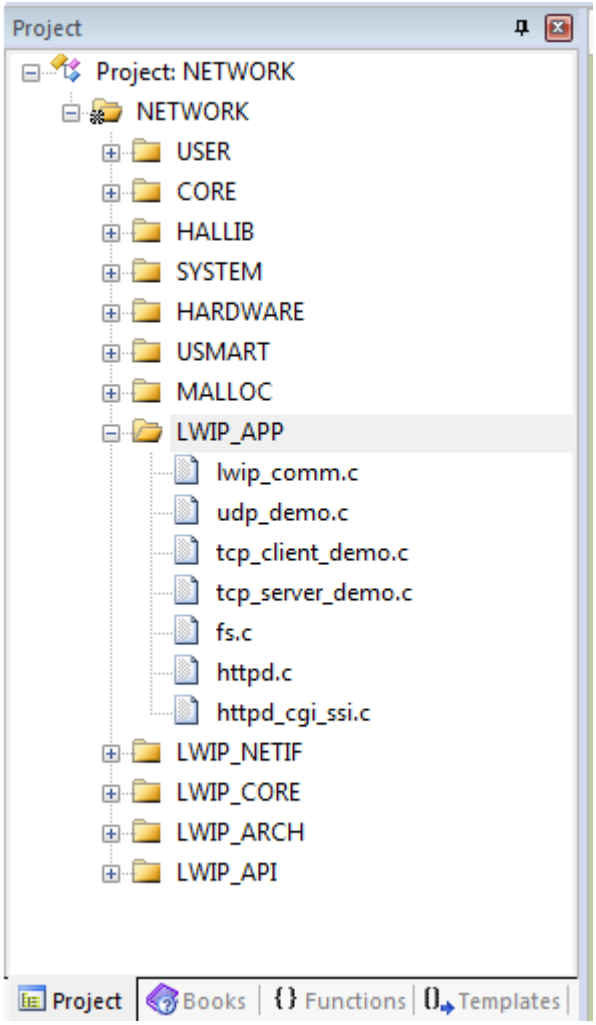


图 66.3.2 例程工程结构体

本章例程所实现的功能，全部由 LWIP_APP 组下的几个.c 文件实现，这些文件的具体介绍在：ALIENTEK 《STM32F767 LWIP 开发手册.pdf》里面，请大家参考该文档学习。

其他部分代码我们就不详细介绍了，最后，我们来看看 main.c 里面的代码，如下：

```
//加载 UI
//mode:
//bit0:0,不加载;1,加载前半部分 UI
//bit1:0,不加载;1,加载后半部分 UI
void lwip_test_ui(u8 mode)
{
    u8 speed;
    u8 buf[30];
    POINT_COLOR=RED;
    if(mode&1<<0)
    {
        LCD_Fill(30,30,lcddev.width,110,WHITE); //清除显示
        LCD_ShowString(30,30,200,16,16,"Apollo STM32F4/F7");
        LCD_ShowString(30,50,200,16,16,"Ethernet lwIP Test");
```

```
            LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
            LCD_ShowString(30,90,200,16,16,"2016/7/19");
        }
        if(mode&1<<1)
        {
            LCD_Fill(30,110,lcddev.width,lcddev.height,WHITE);//清除显示
            LCD_ShowString(30,110,200,16,16,"lwIP Init Successed");
            if(lwipdev.dhcpstatus==2)sprintf((char*)buf,"DHCP IP:%d.%d.%d.%d",lwipdev.ip[0],
                                    lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);//打印动态 IP 地址
            else sprintf((char*)buf,"Static IP:%d.%d.%d.%d",lwipdev.ip[0],lwipdev.ip[1],
                                                lwipdev.ip[2],lwipdev.ip[3]);//打印静态 IP 地址
            LCD_ShowString(30,130,210,16,16,buf);
            speed=LAN8720_Get_Speed();//得到网速
            if(speed&1<<1)LCD_ShowString(30,150,200,16,16,"Ethernet Speed:100M");
            else LCD_ShowString(30,150,200,16,16,"Ethernet Speed:10M");
            LCD_ShowString(30,170,200,16,16,"KEY0:TCP Server Test");
            LCD_ShowString(30,190,200,16,16,"KEY1:TCP Client Test");
            LCD_ShowString(30,210,200,16,16,"KEY2:UDP Test");
        }
    }

    int main(void)
    {
        u8 t;
        u8 key;

        Cache_Enable();                 //打开 L1-Cache
        MPU_Memory_Protection();        //保护相关存储区域
        HAL_Init();                     //初始化 HAL 库
        Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
        delay_init(216);                //延时初始化
        uart_init(115200);              //串口初始化
        usmart_dev.init(108);           //初始化 USMART
        LED_Init();                     //初始化 LED
        KEY_Init();                     //初始化按键
        SDRAM_Init();                   //初始化 SDRAM
        LCD_Init();                     //初始化 LCD
        PCF8574_Init();                 //初始化 PCF8574
        MY_ADC_Init();                  //初始化 ADC
        RTC_Init();                     //初始化 RTC
        TIM3_Init(1000-1,1080-1); //定时器 3 初始化，定时器时钟为 108M，分频系数为 10800-1，
                        //所以定时器 3 的频率为 108M/1080=100K，自动重装载为 1000-1，
                        //那么定时器周期就是 10ms
```

```
    my_mem_init(SRAMIN);                //初始化内部内存池
    my_mem_init(SRAMEX);                //初始化外部内存池
    my_mem_init(SRAMDTCM);              //初始化 DTCM 内存池
    POINT_COLOR=RED;
    LED0(1);
    lwip_test_ui(1);                    //加载前半部分 UI
    LCD_ShowString(30,110,200,16,16,"lwIP Initing...");
    while(lwip_comm_init())             //lwip 初始化
    {
        LCD_ShowString(30,110,200,20,16,"LWIP Init Falied! ");
        delay_ms(500);
        LCD_ShowString(30,110,200,16,16,"Retrying...          ");
        delay_ms(500);
    }
    LCD_ShowString(30,110,200,20,16,"LWIP Init Success!");
    LCD_ShowString(30,130,200,16,16,"DHCP IP configing...");  //等待 DHCP 获取
#if LWIP_DHCP     //使用 DHCP
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0XFF))/
                                                  /等待 DHCP 获取成功/超时溢出
    {
        lwip_periodic_handle(); //LWIP 内核需要定时处理的函数
    }
#endif
    lwip_test_ui(2);            //加载后半部分 UI
    httpd_init();               //HTTP 初始化(默认开启 websever)
    while(1)
    {
        key=KEY_Scan(0);
        switch(key)
        {
            case KEY0_PRES://TCP Server 模式
                tcp_server_test();
                lwip_test_ui(3);//重新加载 UI
                break;
            case KEY1_PRES://TCP Client 模式
                tcp_client_test();
                lwip_test_ui(3);//重新加载 UI
                break;
            case KEY2_PRES://UDP 模式
                udp_demo_test();
                lwip_test_ui(3);//重新加载 UI
                break;
        }
```

```
            lwip_periodic_handle();
            delay_ms(2);
            t++;
            if(t==100)LCD_ShowString(30,230,200,16,16,"Please choose a mode!");
            if(t==200)
            {
                t=0;
                LCD_Fill(30,230,230,230+16,WHITE);//清除显示
                LED0_Toggle;
            }
        }
    }
```

这里，我们开启了定时器 3 来给 LWIP 提供时钟，然后通过 lwip_comm_init 函数，初始化 LWIP，该函数处理包括：初始化 STM32F767 的以太网外设、初始化 LAN8720A、分配内存、使能 DHCP、添加并打开网卡等操作。

这里特别注意：因为我们配置 STM32F767 的网卡使用自动协商功能（双工模式和连接速度），如果协商过程中遇到问题，则会进行多次重试，需要等待很久，而且如果协商失败，那么直接返回错误，导致 LWIP 初始化失败，因此一定要插上网线，然后 LWIP 才能初始化成功，否则肯定会初始化失败，而这个失败，不是硬件问题，是因为你没插网线的缘故!!!

在 LWIP 初始化成功后，进入 DHCP 获取 IP 状态，当 DHCP 获取成功后，显示开发板获取到的 IP 地址，然后开启 HTTP 服务。此时可以在浏览器输入开发板 IP 地址，登录 Web 控制界面，进行 Web Server 测试。

在主循环里面，我们可以通过按键选择：TCP Server 测试、TCP Client 测试和 UDP 测试等测试项目，主循环还调用了 lwip_periodic_handle 函数，周期性处理 LWIP 事务。

软件设计部分就为大家介绍到这里。

66.4 下载验证

在开始测试之前，我们先用网线（需自备）将开发板和电脑连接起来。

对于有路由器的用户，直接用网线连接路由器，同时电脑也连接路由器，即可完成电脑与开发板的连接设置。

对于没有路由器的用户，则直接用网线连接电脑的网口，然后设置电脑的本地连接属性，如图 66.4.1 所示：

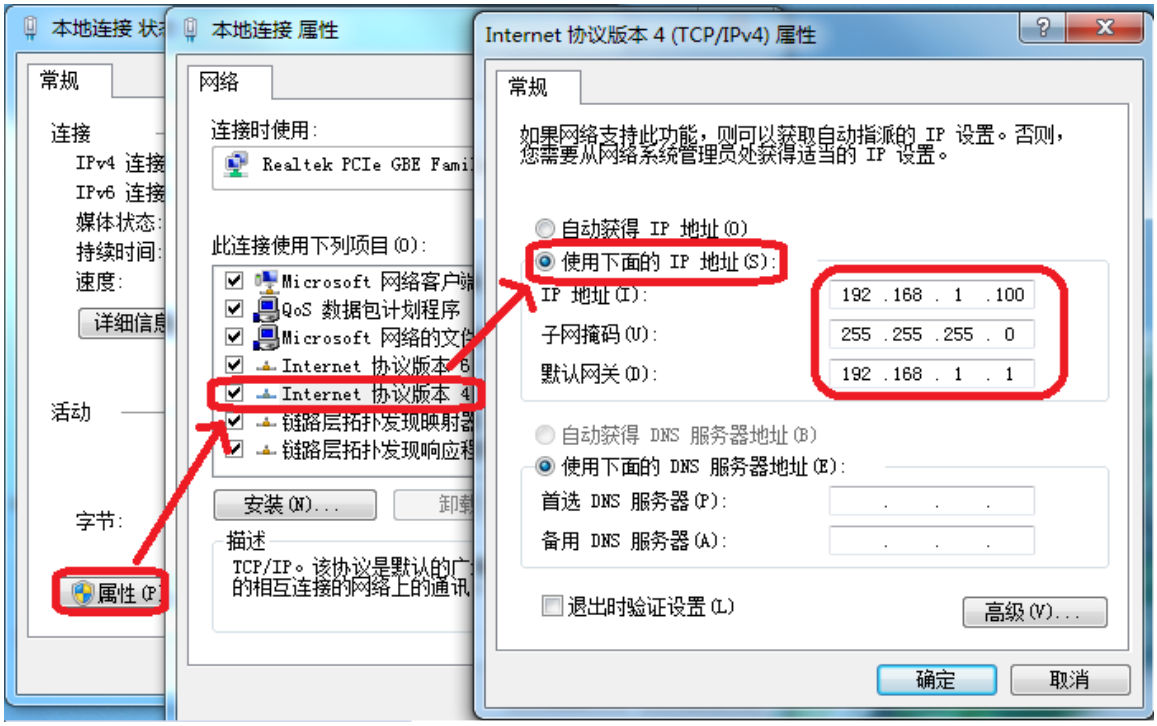


图 66.4.1 开发板与电脑直连时电脑本地连接属性设置

这里，我们设置 IPV4 的属性，设置 IP 地址为：192.168.1.100（100 是可以随意设置的，但是不能是 30 和 1）；子网掩码：255.255.255.0；网关：192.168.1.1；DNS 部分可以不用设置。

设置完后，点击确定，即可完成电脑端设置，这样开发板和电脑就可以通过互相通信了。

然后，在代码编译成功之后，我们通过下载代码到阿波罗 STM32 开发板上（这里我们以路由器连接方式介绍，下同，且假设 DHCP 获取 IP 成功），LCD 显示如图 66.4.2 所示界面：

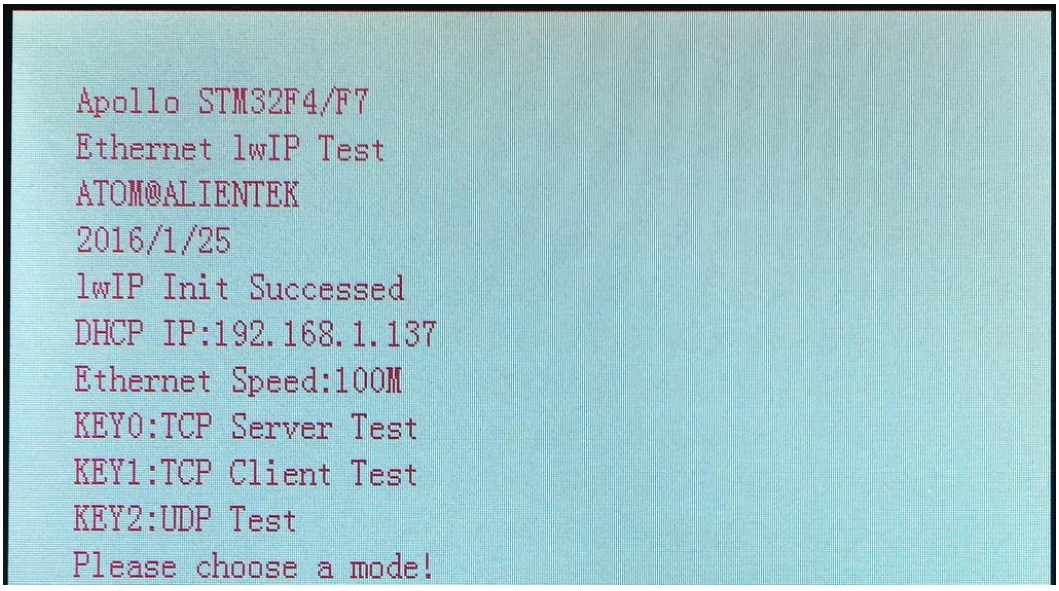


图 66.4.2 DHCP 获取 IP 成功

此时屏幕提示选择测试模式，可以选择 TCP Server、TCP Client 和 UDP 三项测试。不过，我们先来看看网络连接是否正常。从 66.4.2 可以看到，我们开发板通过 DHCP 获取到的 IP 地址为：192.168.1.137，因此，我们在电脑上先来 ping 一下这个 IP，看看能否 ping 通，以检查连接是否正常（Start→运行→CMD），如图 66.4.3 所示：

图 66.4.3 ping 开发板 IP 地址

可以看到开发板所显示的 IP 地址，是可以 ping 通的，说明我们的开发板和电脑连接正常，可以开始后续测试了。

66.4.1 Web Server 测试

这个测试不需要任何操作来开启，开发板在获取 IP 成功（也可以使用静态 IP）后，即开启了 Web Server 功能。我们在浏览器输入：192.168.1.137（开发板显示的 IP 地址），即可进入一个 Web 界面，如图 66.4.1.1 所示：

图 66.4.1.1 Web Server 测试网页

该界面总共有 5 个子页面：主页、LED/BEEP 控制、ADC/内部温度传感器、RTC 实时时钟和联系我们等。登录 Web 时默认打开的是主页面，介绍了我们阿波罗 STM32F767 开发板的一些资源和特点和 LWIP 的一些简介。

点击：LED/BEEP 控制，进入该子页面，即可对开发板板载的 DS0（LED1）和蜂鸣器进行控制，如图 66.4.1.2 所示：

图 66.4.1.2 LED/BEEP 控制页面

此时，选择 ON，然后点击 SEND 按钮，即可点亮 LED1 或者打开蜂鸣器。同样，发送 OFF 即可关闭 LED1 或蜂鸣器。

点击：ADC/内部温度传感器，进入该子页面，会显示 ADC1 通道 5 的值和 STM32 内部温度传感器所测得的温度，如图 66.4.1.3 所示：

图 66.4.1.3 ADC/内部温度传感器测试页面

ADC1_CH5 是我们开发板多功能接口 ADC 的输入通道，默认连接在 TPAD 上，TPAD 带

有上拉电阻，所以这里显示 3.3V 左右，大家可以将 ADC 接其他地方来测量电压。同时，该界面还显示了内部温度传感器采集到的温度值。该界面每个一秒钟刷新一次。

点击：RTC 实时时钟，进入该子页面，会显示 STM32 内部 RTC 的时间和日期，如图 66.4.1.4 所示：

图 66.4.1.4 RTC 实时时钟测试页面

此界面显示了阿波罗 STM32F767 自带的 RTC 实时时钟的当前时间和日期等参数，每隔 1 秒钟刷新一次。

最后，点击联系我们，即可进入到 ALIENTEK 官方店铺，这里就不再介绍了。

66.4.2 TCP Server 测试

在提示界面，按 KEY0 即可进入 TCP Server 测试，此时，开发板作为 TCP Server。此时，LCD 屏幕上显示 Server IP 地址(就是开发板的 IP 地址)，Server 端口固定为：8088。如图 66.4.2.1 所示：

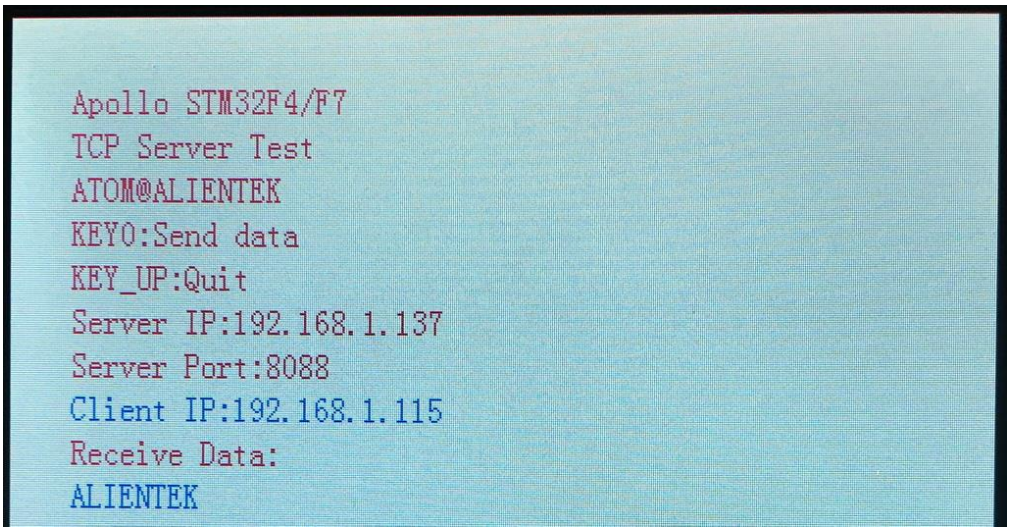


图 66.4.2.1 TCP Sever 测试界面

图中显示了 Server IP 地址是 192.168.1.137，Server 端口号是：8088。上位机配合我们测试，需要用到一个网络调试助手的软件，该软件在光盘→ 6，软件资料→软件→网络调试助手→网络调试助手 V3.8.exe。

我们在电脑端打开网络调试助手，设置协议类型为：TCP Client，服务器 IP 地址为：192.168.1.137，服务器端口号为：8088，然后点击连接，即可连上开发板的 TCP Sever，此时，开发板的液晶显示：Client IP:192.168.1.115（电脑的 IP 地址），如图 66.4.2.1 所示，而网络调试助手端则显示连接成功，如图 6.4.2.2 所示：

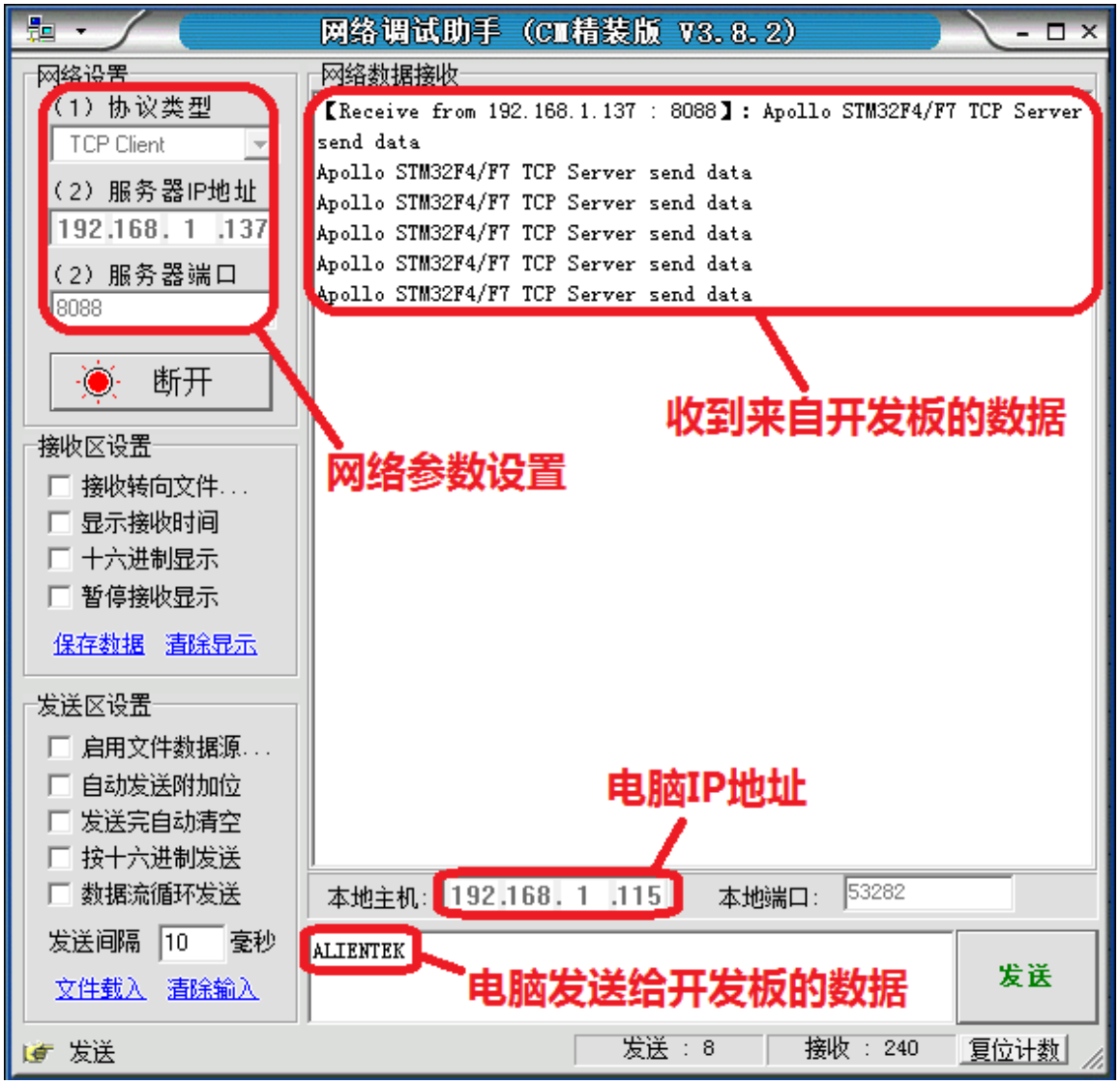


图 66.4.2.2 电脑端网络调试助手 TCP Client 测试界面

按开发板的 KEY0 按键，即可发送数据给电脑。同样，电脑端输入数据，也可以通过网络调试助手发送给开发板。如图 63.4.2.1 和图 66.4.2.2 所示。按 KEY_UP 按键，可以退出 TCP Sever 测试，返回选择界面。

66.4.3 TCP Client 测试

在提示界面，按 KEY1 即可进入 TCP Client 测试，此时，先进入一个远端 IP 设置界面，也就是 Client 要去连接的 Server 端的 IP 地址。通过 KEY0/KEY2 可以设置 IP 地址，通过 66.4.3.2 节的测试，我们知道电脑的 IP 是 192.168.1.115，所以我们这里设置 Client 要连接的远端 IP 为

192.168.1.115，如图 66.4.3.1 所示：

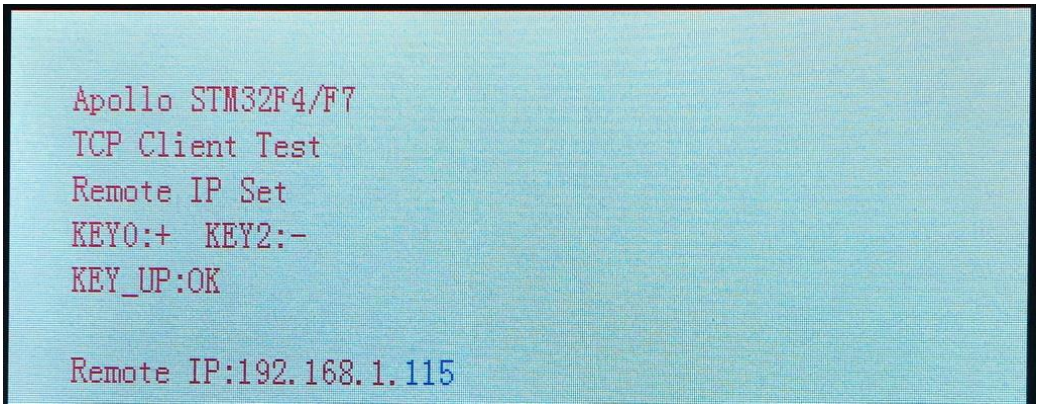


图 66.4.3.1 远端 IP 地址设置

设置好之后，按 KEY_UP，确认，进入 TCP Client 测试界面。开始的时候，屏幕显示 Disconnected。然后我们在电脑端打开网络调试助手，设置协议类型为：TCP Server，本地 IP 地址为：192.168.1.115（电脑 IP），本地端口号为：8087，然后点击连接，开启电脑端的 TCP Server 服务，如图 66.4.3.2 所示：

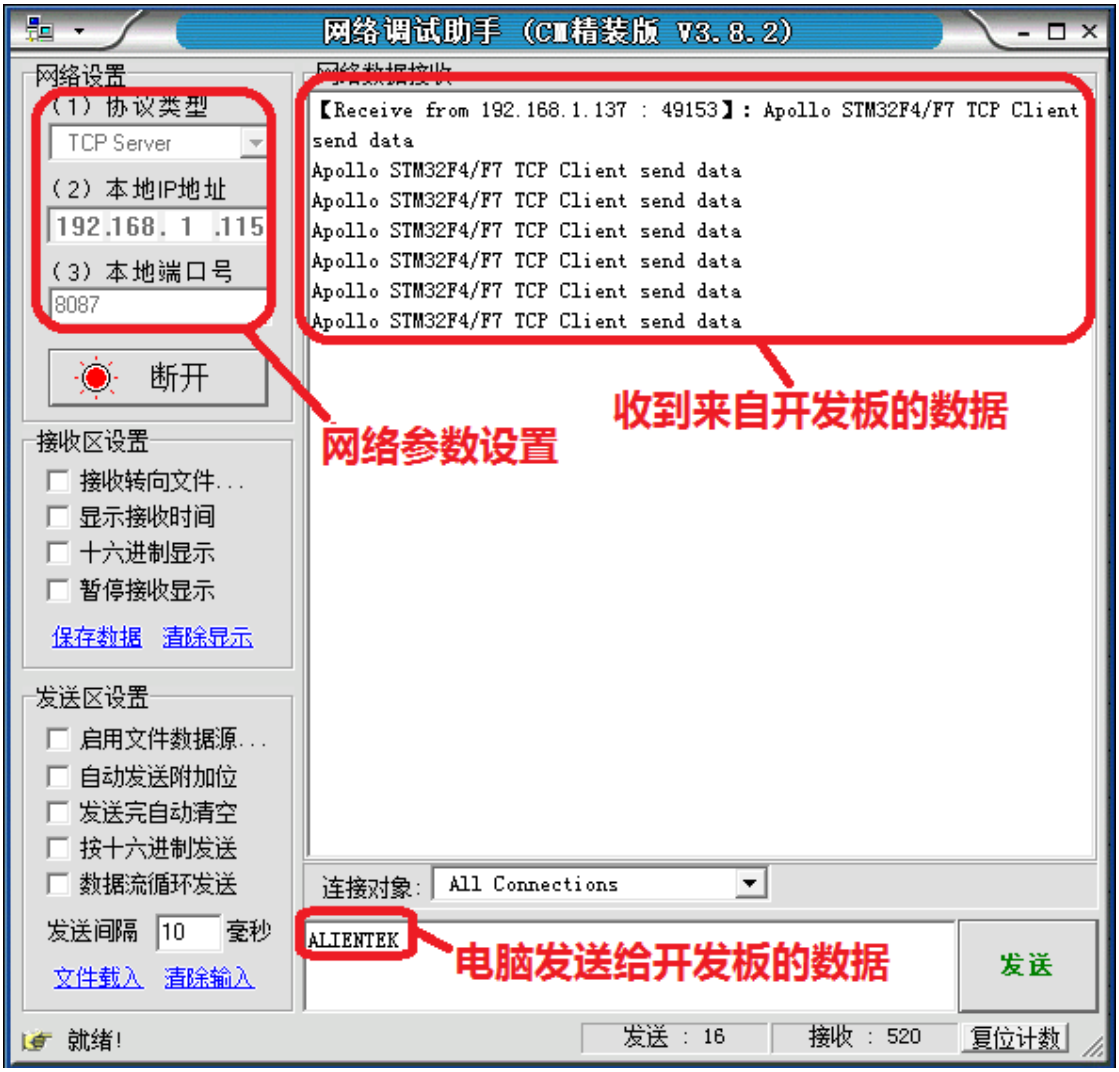


图 66.4.3.2 电脑端网络调试助手 TCP Server 测试界面

在电脑端开启 Server 后，稍等片刻，开发板的 LCD 即显示 Connected，如图 66.4.3.3 所示：

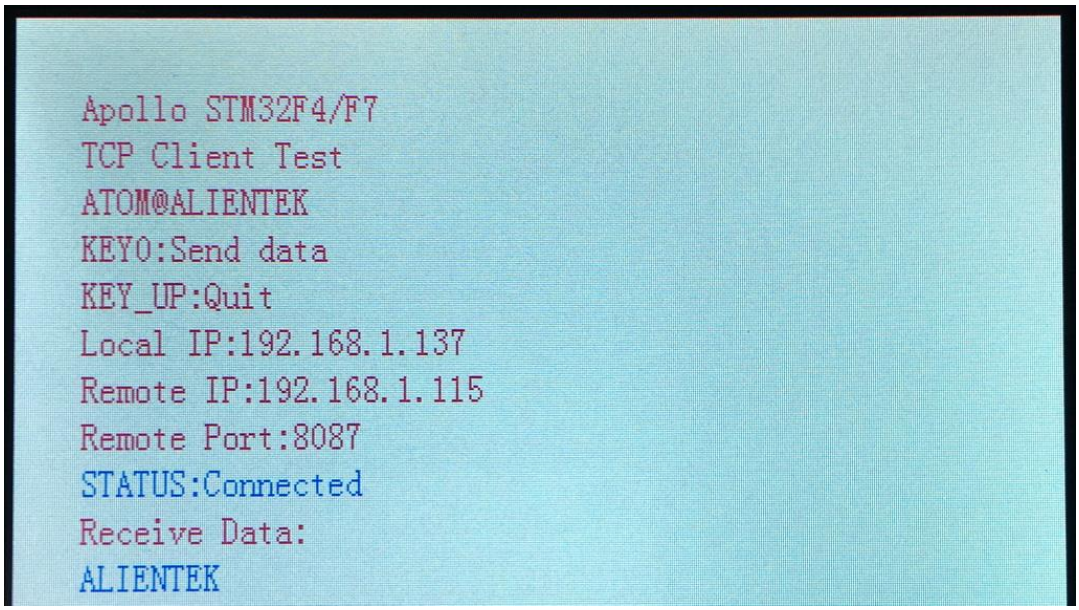


图 66.4.3.3 TCP Client 测试界面

在连接成功后，电脑和开发板即可互发数据，同样开发板还是按 KEY0 发送数据给电脑，测试结果如图 66.4.3.2 和图 66.4.3.3 所示。按 KEY_UP 按键，可以退出 TCP Client 测试，返回选择界面。

66.4.4 UDP 测试

在提示界面，按 KEY2 即可进入 UDP 测试，UDP 测试同 TCP Client 测试一样，要先设置远端 IP 地址，设置好之后，进入 UDP 测试界面，如图 66.4.4.1 所示：

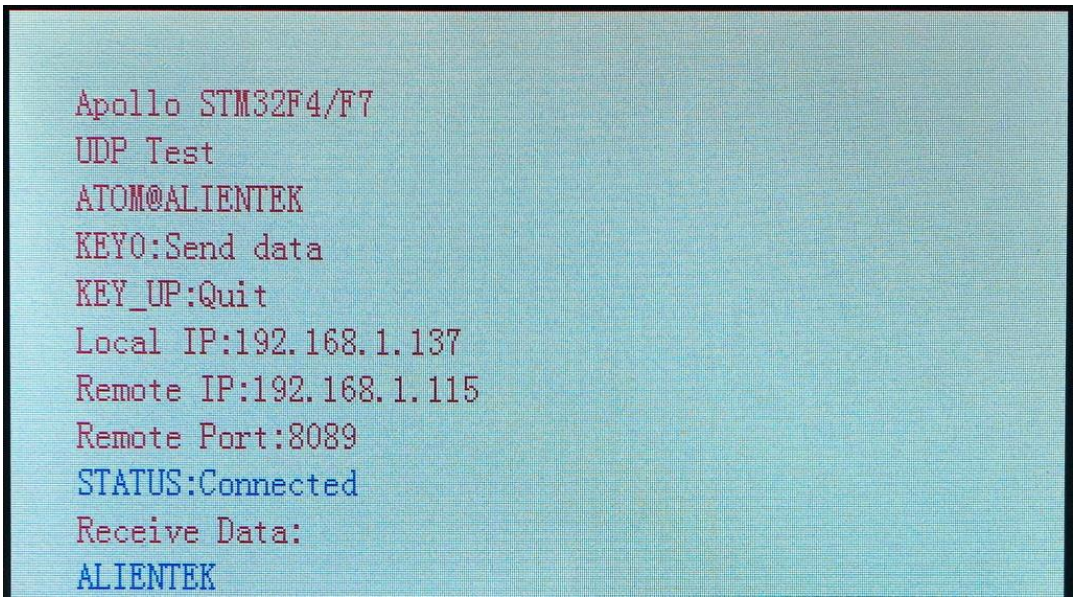


图 66.4.4.1 UDP 测试界面

可以看到，UDP 测试时我们要连接的端口号为：8089，所以网络调试助手需要设置端口号为：8089。另外，UDP 不是基于连接的传输协议，所以，这里直接就显示 Connected 了。在电脑端打开网络调试助手，设置协议类型为：UDP，本地 IP 地址为：192.168.1.115（电脑 IP），

本地端口号为：8089，然后点击连接，开启电脑端的 UDP 服务，如图 66.4.4.2 所示：

图 66.4.3.2 电脑端网络调试助手 UDP 测试界面

然后，我们先按开发板的 KEY0，发送一次数据给电脑端网络调试助手，这样电脑端网络调试助手便会识别出开发板的 IP 地址，然后就可以互相发送数据了。按 KEY_UP 按键，可以退出 UDP 测试，返回选择界面。

第六十七章 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：

67.1 UCOSII 简介
67.2 硬件设计
67.3 软件设计
67.4 下载验证

67.1 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（V2.92 版本）体系结构如图 67.1.1 所示：

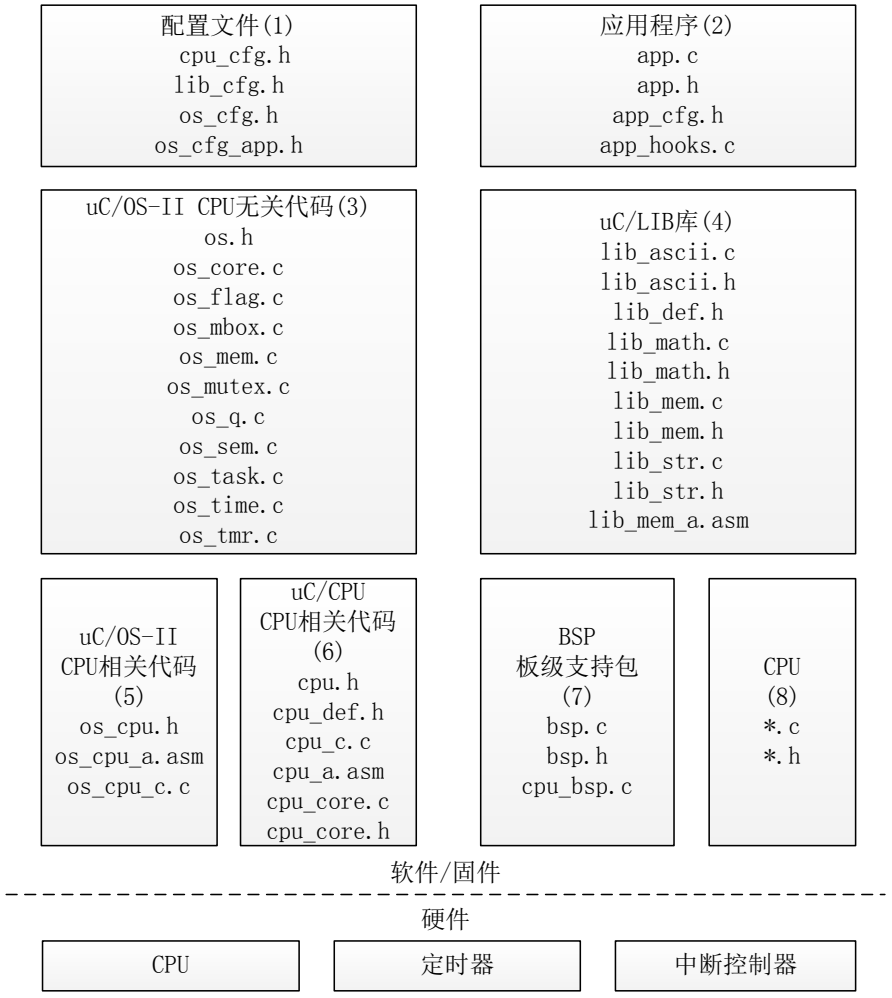


图 67.1.1 UCOSII 体系结构图

(1)、这部分是系统配置文件，用来配置所需的系统功能，比如需要用到的 UCOSII 的模块、时钟频率等等。

(2)、这部分为用户的应用程序，即使用 UCOSII 完成的应用层代码，文件不一定命名为 app.c，可以命名为其他的。注意，app_hooks.c 里面是钩子函数的应用层代码，app_cfg.h 是与 APP 配置有关的，这个是 Micrium 公司提供的模板，不使用的话就可以直接删掉。

(3)、这部分是 UCOSII 的核心源码，它们是与处理器无关的代码，都是由高度可移植的 ANSI C 编写的。

(4)、Micrium 重写了 stdlib 库中的一些函数，如内存复制，字符串相关函数等。这样做的目的是为了保证在不同应用程序和编译器之间的可移植性。

(5)、这部分的文件需要根据不同的 CPU 架构去做修改，也就是移植的过程。从这里可看出移植的真正核心就是这三个文件的修改。

(6)、此部分是 Micrium 官方封装起来的 CPU 相关功能代码，比如打开和关闭中断等。

(7)、板级支持包(BSP)，说白了就是外设驱动代码，根据需求用户自行编写，不一定要用 bsp.c 和 bsp.h 这样的文件命名。cpu_bsp.c 是与 cpu 有关的驱动。

(8)、CPU 厂商提供的针对本公司 CPU 所制作的库函数，比如 ST 针对 STM32 提供的 STD 和 HAL 这种库函数。

图 67.1.1 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 STM32F7 的 SYSTICK 定时器来提供 UCOSII 时钟节拍。

关于 UCOSII 在 STM32F7 的详细移植过程，请参考光盘资料：《STM32F7 UCOS 开发手册.pdf》，教程在光盘根目录，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（V2.92）。

所谓的任务，其实就是一个死循环函数，该函数实现一定的功能，一个工程可以有很多这样的任务（最多 255 个），UCOSII 对这些任务进行调度管理，让这些任务可以并发工作（注意不是同时工作！！，并发只是各任务轮流占用 CPU，而不是同时占用，任何时候还是只有 1 个任务能够占用 CPU），这就是 UCOSII 最基本的功能。Ucos 任务的一般格式为：

```
void MyTask (void *pdata)
{
 任务准备工作…
 While(1)//死循环
{
    任务 MyTask 实体代码;
    OSTimeDlyHMSM(x,x,x,x);//调用任务延时函数，释放 cpu 控制权，
}
}
```

假如我们新建了 2 个任务为 MyTask 和 YourTask,这里我们先忽略任务优先级的概念，两个任务死循环中延时时间为 1s。如果某个时刻，任务 MyTask 在执行中，当它执行到延时函数 OSTimeDlyHMSM 的时候，它释放 cpu 控制权，这个时候，任务 YourTask 获得 cpu 控制权开始执行，任务 YourTask 执行过程中，也会调用延时函数延时 1s 释放 CPU 控制权，这个过程中任务 A 延时 1s 到达，重新获得 CPU 控制权，重新开始执行死循环中的任务实体代码。如此循环，现象就是两个任务交替运行，就好像 CPU 在同时做两件事情一样。

疑问来了，如果有很多任务都在等待，那么先执行那个任务呢？如果任务在执行过程中，想停止之后去执行其他任务是否可行呢？这里就涉及到任务优先级以及任务状态任务控制的一些知识，我们在后面会有所提到。如果要详细的学习，建议看任哲老师的《ucosII 实时操作系统》一书。

前面我们学习的所有实验，都是一个大任务（死循环），这样，有些事情就比较不好处理，比如：音乐播放器实验，在音乐播放的时候，我们还希望显示歌词，如果是 1 个死循环（一个任务），那么很可能在显示歌词的时候，音频可能出现停顿（尤其是采样率高的时候），这主要是歌词显示占用太长时间，导致 IIS 数据无法及时填充而停顿。而如果用 UCOSII 来处理，那么我们可以分 2 个任务，音乐播放一个任务（优先级高），歌词显示一个任务（优先级低）。这样，由于音乐播放任务的优先级高于歌词显示任务，音乐播放任务可以打断歌词显示任务，从而及时给 IIS 填充数据，保证音频不断，而显示歌词又能顺利进行。这就是 UCOSII 带来的好处。

这里有几个 UCOSII 相关的概念需要大家了解一下：任务优先级，任务堆栈，任务控制块，任务就绪表和任务调度器。

任务优先级，这个概念比较好理解，ucos 中，每个任务都有唯一的一个优先级。优先级是

任务的唯一标识。在 UCOSII 中，使用 CPU 的时候，优先级高（数值小）的任务比优先级低的任务具有优先使用权，即任务就绪表中总是优先级最高的任务获得 CPU 使用权，只有高优先级的任务让出 CPU 使用权（比如延时）时，低优先级的任务才能获得 CPU 使用权。UCOSII 不支持多个任务优先级相同，也就是每个任务的优先级必须不一样。

任务堆栈，就是存储器中的连续存储空间。为了满足任务切换和响应中断时保存 CPU 寄存器中的内容以及任务调用其他函数时的需要，每个任务都有自己的堆栈。在创建任务的时候，任务堆栈是任务创建的一个重要入口参数。

任务控制块 OS_TCB，用来记录任务堆栈指针，任务当前状态以及任务优先级等任务属性。UCOSII 的任何任务都是通过任务控制块（TCB）的东西来控制的，一旦任务创建了，任务控制块 OS_TCB 就会被赋值。每个任务管理块有 3 个最重要的参数：1，任务函数指针；2，任务堆栈指针；3，任务优先级；任务控制块就是任务在系统里面的身份证（UCOSII 通过优先级识别任务），任务控制块我们就不再详细介绍了，详细介绍请参考任哲老师的《嵌入式实时操作系统 UCOSII 原理及应用》一书第二章。

任务就绪表，简而言之就是用来记录系统中所有处于就绪状态的任务。它是一个位图，系统中每个任务都在这个位图中占据一个进制位，该位置的状态（1 或者 0）就表示任务是否处于就绪状态。

任务调度的作用一是在任务就绪表中查找优先级最高的就绪任务，二是实现任务的切换。比如说，当一个任务释放 cpu 控制权后，进行一次任务调度，这个时候任务调度器首先要去任务就绪表查询优先级最高的就绪任务，查到之后，进行一次任务切换，转而去执行下一个任务。关于任务调度的详细介绍，请参考《嵌入式实时操作系统 UCOSII 原理及应用》一书第三章相关内容。

UCOSII 的每个任务都是一个死循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是：睡眠状态、 就绪状态、 运行状态、 等待状态(等待某一事件发生)和中断服务状态。

睡眠状态，任务在没有被配备任务控制块或被剥夺了任务控制块时的状态。

就绪状态，系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，任务已经准备好了，但由于该任务的优先级比正在运行的任务的优先级低， 还暂时不能运行，这时任务的状态叫做就绪状态。

运行状态，该任务获得 CPU 使用权，并正在运行中，此时的任务状态叫做运行状态。

等待状态，正在运行的任务，需要等待一段时间或需要等待一个事件发生再运行时，该任务就会把 CPU 的使用权让给别的任务而使任务进入等待状态。

中断服务状态，一个正在运行的任务一旦响应中断申请就会中止运行而去执行中断服务程序，这时任务的状态叫做中断服务状态。

UCOSII 任务的 5 个状态转换关系如图 67.1.2 所示：

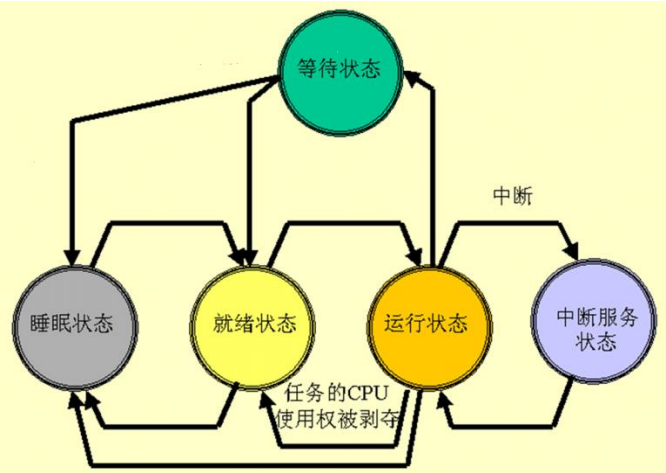


图 67.1.2 UCOSII 任务状态转换关系

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 给我们提供了 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。但是，如果某个任务中要使用到 FPU，那么就只能用函数 OSTaskCreateExt 来创建，因为 OSTaskCreate 函数并没有提供针对 FPU 的处理选项，而函数 OSTaskCreateExt 有。OSTaskCreate 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 函数原型为：

```
INT8U    OSTaskCreateExt (void (*task)(void *p_arg),     void      *p_arg,
                          OS_STK   *ptos,                INT8U     prio,
                          INT16U    id,                  OS_STK    *pbos,
                          INT32U    stk_size,            void      *pext,
                          INT16U    opt);
```

该函数的参数就比较多了：task 是指向任务函数的函数指针；p_arg 指向传递给任务函数的参数；ptos 是分配给任务的堆栈的栈顶指针；prio 是任务优先级；id 是任务 ID 号，范围是 0~65535；pbos 是任务堆栈的栈底指针；stk_size 是任务堆栈大小；pext 是用户补充的存储区，作为对 TCB 的补充，不使用的时候设置为 0；opt 是任务选项，有三个可选选项：OS_TASK_OPT_STK_CHK、OS_TASK_OPT_STK_CLR 和 OS_TASK_OPT_SAVE_FP。它们分别为：检查任务堆栈、任务堆栈清零和保存浮点(FPU)寄存器。

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 STM32F7 上面运行 UCOSII 的步骤：

1） 移植 UCOSII

要想 UCOSII 在 STM32F7 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：STM32F7 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，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 STM32F7 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。

67.2 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。

67.3 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 源码分为五个文件夹：uC-CPU、uC-LIB、UCOS_BSP、uCOS-CONFIG 和 uCOS-II）。

打开工程，新建 UCOSII_BSP、UCOSII_CPU、UCOSII_LIB、UCOSII_CORE、UCOSII_PORT 和 UCOSII_CONFIG 六个分组，分别添加 UCOSII 五个文件夹下的源码，并且添加相应的头文件路径，最后得到工程如图 67.3.1 所示：

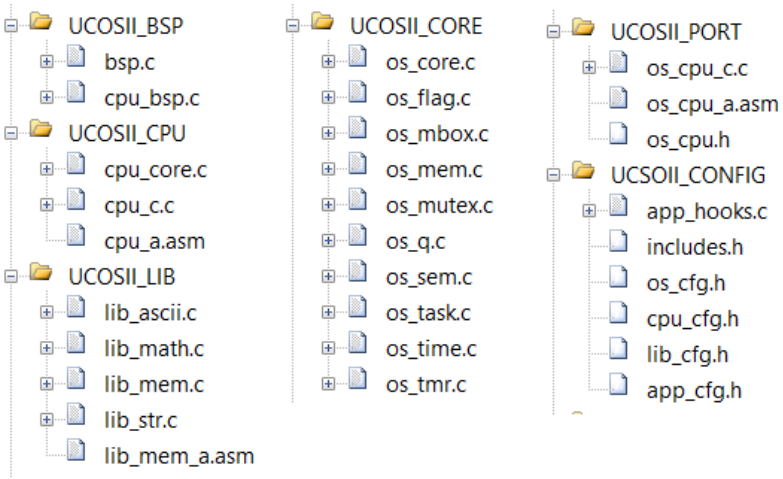


图 67.3.1 向分组中添加源码

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 20，也就是最多 20 个任务（包括空闲任务和统计任务在内），其他配置我们就不详细介绍了，请参考本实验源码。

前面提到，我们需要在 sys.h 里面设置 SYSTEM_SUPPORT_UCOS 为 1，以支持 UCOSII，通过这个设置，我们不仅可以实现利用 delay_init 来初始化 SYSTICK，产生 UCOSII 的系统时钟节拍，还可以让 delay_us 和 delay_ms 函数在 UCOSII 下能够正常使用（实现原理请参考 5.1 节），这使得我们之前的代码，可以十分方便的移植到 UCOSII 下。虽然 UCOSII 也提供了延时函数：OSTimeDly 和 OSTimeDLyHMSM，但是这两个函数的最少延时单位只能是 1 个 UCOSII 时钟节拍，在本章，即 5ms，显然不能实现 us 级的延时，而 us 级的延时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_UCOS 为 1 之后，UCOSII 的时钟节拍由 SYSTICK 的中断服务函数提供，该部分代码如下：

```
//systick 中断服务函数,使用 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();                        //退出中断,会触发进行中断级任务切换
    }
}
```

以上代码，其中 OSIntEnter 是进入中断服务函数，用来记录中断嵌套层数（OSIntNesting 增加 1）；OSTimeTick 是系统时钟节拍服务函数，在每个时钟节拍了解每个任务的延时状态，使已经到达延时时限的非挂起任务进入就绪状态；OSIntExit 是退出中断服务函数，该函数可能触发一次任务切换（当 OSIntNesting==0&&调度器未上锁&&就绪表最高优先级任务！=被中断的任务优先级时），否则继续返回原来的任务执行代码（如果 OSIntNesting 不为 0，则减 1）。

事实上，任何中断服务函数，我们都应该加上 OSIntEnter 和 OSIntExit 函数，这是因为 UCOSII 是一个可剥夺型的内核，中断服务子程序运行之后，系统会根据情况进行一次任务调度去运行优先级别最高的就绪任务，而并不一定接着运行被中断的任务！

最后，我们打开 main.c，输入如下代码：

```
//////////////////////////UCOSII 任务设置//////////////////////////////////
//START 任务
#define START_TASK_PRIO                 10          //设置任务优先级
#define START_STK_SIZE                  128         //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                   128         //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                   128         //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
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
```

```
    delay_init(216);                    //延时初始化
    uart_init(115200);                  //串口初始化
    LED_Init();                         //初始化 LED

    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)
    //LED0 任务
    OSTaskCreateExt((void(*)(void*) )led0_task,
                    (void*              )0,
                    (OS_STK*            )&LED0_TASK_STK[LED0_STK_SIZE-1],
                    (INT8U              )LED0_TASK_PRIO,
                    (INT16U             )LED0_TASK_PRIO,
                    (OS_STK*            )&LED0_TASK_STK[0],
                    (INT32U             )LED0_STK_SIZE,
                    (void*              )0,

            (INT16U             )OS_TASK_OPT_STK_CHK|OS_TASK_OPT_STK_CLR|\
                        OS_TASK_OPT_SAVE_FP);
    //LED1 任务
    OSTaskCreateExt((void(*)(void*) )led1_task,
                    (void*              )0,
```

```
                        (OS_STK*        )&LED1_TASK_STK[LED1_STK_SIZE-1],
                        (INT8U          )LED1_TASK_PRIO,
                        (INT16U         )LED1_TASK_PRIO,
                        (OS_STK*        )&LED1_TASK_STK[0],
                        (INT32U         )LED1_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
}

//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 128（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或者其他可以引发任务切换的函数，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。

67.4 下载验证

在代码编译成功之后，我们通过下载代码到阿波罗 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。

67.5 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 62 UCOSII 实验 1-2-任务创建删除挂起恢复”中，主函数文件 main.c 源码如下：

前面我们简单的建立了两个任务，主要是让大家了解 UCOSII 怎么运行以及怎样创建任务。下面我们在这一节补充一个实验测试任务的删除，挂起和恢复。为了和寄存器版本手册章节保持一致，我们这里不另起一章。实验代码在我们光盘的“实验 62 UCOSII 入门实验 1-2-任务创建删除挂起恢复”中，主函数文件 main.c 源码如下：

```
//START 任务
#define START_TASK_PRIO                 10          //开始任务的优先级为最低
#define START_STK_SIZE                  128         //设置任务堆栈大小
OS_STK START_TASK_STK[START_STK_SIZE];  //任务任务堆栈
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                    128         //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           128         //设置任务堆栈大小
OS_STK BEEP_TASK_STK[BEEP_STK_SIZE];    //创建任务堆栈空间
void beep_task(void *pdata); //任务函数接口

//按键扫描任务
#define KEY_TASK_PRIO           3       //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];  //创建任务堆栈空间
void key_task(void *pdata); //任务函数接口

int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574
    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],
                    (INT8U          )START_TASK_PRIO,       //任务优先级
                    (INT16U         )START_TASK_PRIO,       //任务 ID
                    (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);
    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_Toggle;
        LED1_Toggle;
        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
        }
        PCF8574_WriteBit(BEEP_IO,0);                //打开蜂鸣器
        delay_ms(60);
        PCF8574_WriteBit(BEEP_IO,1);                //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。

第六十八章 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：

68.1 UCOSII 信号量和邮箱简介
68.2 硬件设计
68.3 软件设计
68.4 下载验证

68.1 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 68.1.1 所示：

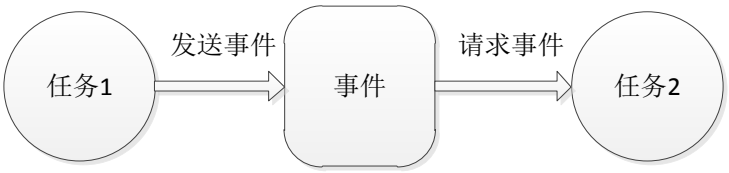


图 68.1.1 两个任务使用事件进行通信的示意图

在图 68.1.1 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。

68.2 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 6 个任务：开始任务、LED 任务、触摸屏任务、蜂鸣器任务、按键扫描任务和主任务，开始任务用于创建信号量、创建邮箱、初始化统计任务以及其他任务的创建，之后挂起；LED 任务用于 DS0 控制，提示程序运行状况；蜂鸣器任务用于测试信号量，是请求信号量函数，每得到一个信号量，蜂鸣器就叫一次；触摸屏任务用于在屏幕上画图，可以用于测试 CPU 使用率；按键扫描任务用于按键扫描，优先级最

高，将得到的键值通过消息邮箱发送出去；主任务则通过查询消息邮箱获得键值，并根据键值执行 DS1 控制、信号量发送（蜂鸣器控制）、触摸区域清屏和触摸屏校准等控制。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1

2） 4 个按键（KEY0/KEY1/KEY2/KEY_UP）

3） PCF8574（控制蜂鸣器）

4） LCD 模块

这些，我们在前面的学习中都已经介绍过了。

68.3 软件设计

本章，我们在第三十六章实验 （实验 31 ）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 main.c 函数了，打开 main.c，输入如下代码：

```
//////////////////////////UCOSII 任务设置//////////////////////////////////
//START 任务
#define START_TASK_PRIO                 10          //设置任务优先级
#define START_STK_SIZE                  128         //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                    128         //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                   128         //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                    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)
{
    …//此处省略函数定义
}
int main(void)
{
    Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                     //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);                //延时初始化
    uart_init(115200);              //串口初始化
    LED_Init();                     //初始化 LED
    KEY_Init();                     //初始化按键
    PCF8574_Init();                 //初始化 PCF8574

    SDRAM_Init();                   //初始化 SDRAM
    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();                       //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
    OSStart(); //开始任务
 }
//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)
```

```
{
    …//此处省略函数定义}
//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)
{
    …//此处省略函数定义
}
//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)
{
    …//此处省略函数定义
}
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();                      //开启统计任务
    OS_ENTER_CRITICAL();               //进入临界区(关闭中断)
    //触摸任务
    OSTaskCreateExt((void(*)(void*) )touch_task,
                    (void*          )0,
                    (OS_STK*        )&TOUCH_TASK_STK[TOUCH_STK_SIZE-1],
                    (INT8U          )TOUCH_TASK_PRIO,
                    (INT16U         )TOUCH_TASK_PRIO,
                    (OS_STK*        )&TOUCH_TASK_STK[0],
                    (INT32U         )TOUCH_STK_SIZE,
                    (void*          )0,
                    (INT16U         )OS_TASK_OPT_STK_CHK|\
                                     OS_TASK_OPT_STK_CLR|\
                                     OS_TASK_OPT_SAVE_FP);
    OSTaskCreateExt(……//省略部分代码);          //LED 任务
```

```
        OSTaskCreateExt(……//省略部分代码);          //蜂鸣器任务
        OSTaskCreateExt(……//省略部分代码);          //主任务
        OSTaskCreateExt(……//省略部分代码);          //按键任务
        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)
{
        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);       //请求信号量
                PCF8574_WriteBit(BEEP_IO,0);      //打开蜂鸣器
                delay_ms(60);
                PCF8574_WriteBit(BEEP_IO,1);      //关闭蜂鸣器
                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];
                    }
                    //进入临界段,防止其他任务,打断 LCD 操作,导致液晶乱序.
                    OS_ENTER_CRITICAL();
                    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;
                delay_ms(10); //没有按键按下的时候
            }
        }
}
//主任务
void main_task(void *pdata)
{
    u32 key=0;
    u8 err, semmask=0, tcnt=0;
    while(1)
    {
        key=(u32)OSMboxPend(msg_key,10,&err);
        switch(key)
        {
            case 1://控制 DS1
                LED1_Toggle; break;
            case 2://发送信号量
                semmask=1;
                OSSemPost(sem_beep);
                break;
            case 3://清除
                LCD_Fill(0,121,lcddev.width-1,lcddev.height-1,WHITE);
                break;
            case 4://校准，仅电阻屏有效，电容屏无需校准。
```

```
                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;
                //显示信号量的值
                LCD_ShowxNum(212,50,sem_beep->OSEventCnt,3,16,0X80);
                if(sem_beep->OSEventCnt==0)semmask=0;//停止更新
            }
            if(tcnt==10)//0.6 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，堆栈大小都是 128。

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。

68.4 下载验证

在代码编译成功之后，我们通过下载代码到阿波罗 STM32 开发板上，可以看到 LCD 显示界面如图 68.4.1 所示：

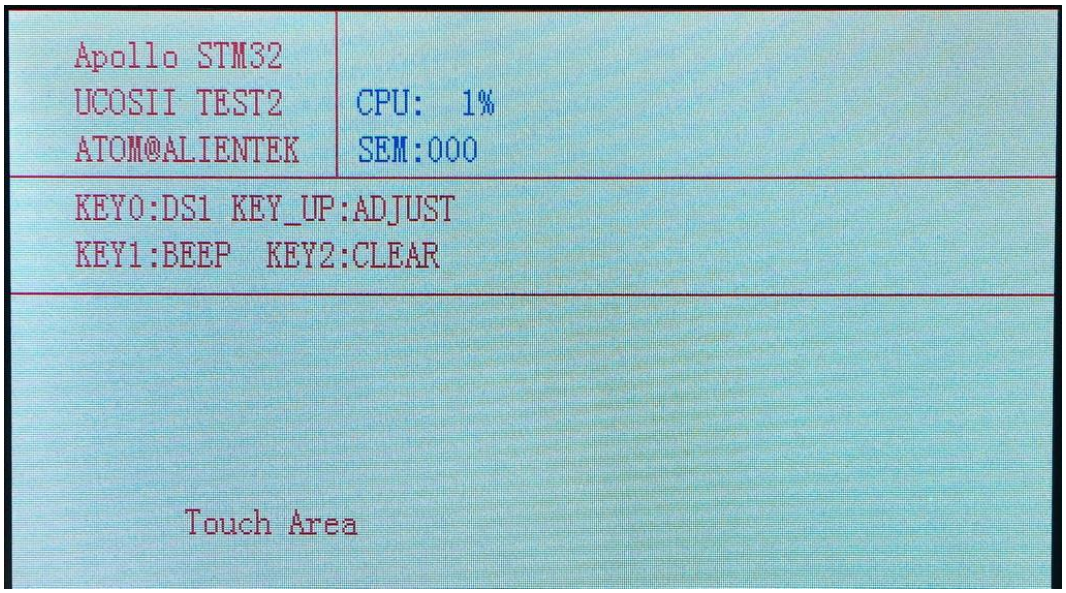


图 68.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率仅为 1%左右。通过按 KEY0，可以控制 DS1 的亮灭；通过按 KEY1 则可以控制蜂鸣器的发声（连续按下多次后，可以看到蜂鸣每隔 1 秒叫一次），同时，可以在 LCD 上面看到信号量的当前值；通过按 KEY2，可以清除触摸屏的输入；通过按 KEY_UP 可以进入校准程序，进行触摸屏校准（注意，电容触摸屏不需要校准，所以如果是电容屏，按 KEY_UP，就相当于清屏一次的效果，不会进行校准）。

第六十九章 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：

69.1 UCOSII 消息队列、信号量集和软件定时器简介

69.2 硬件设计

69.3 软件设计

69.4 下载验证

69.1 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 69.1.1 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。

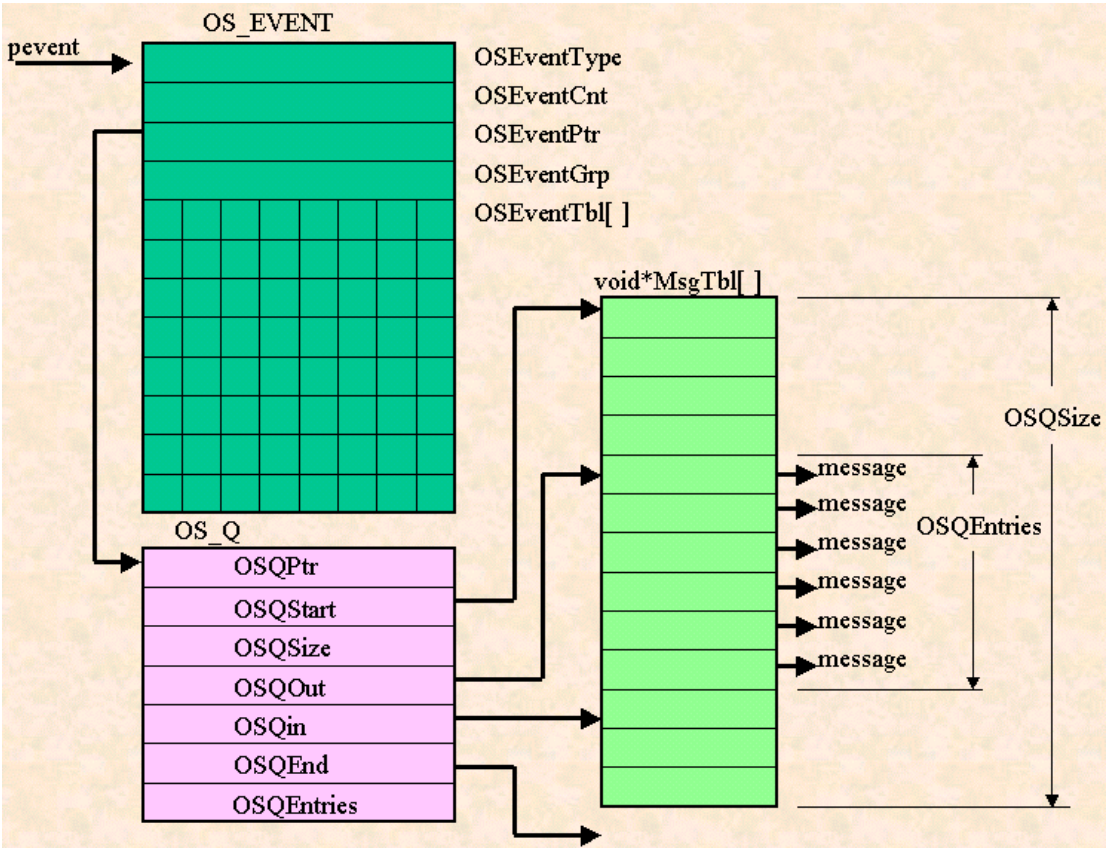


图 69.1.1 消息队列的数据结构

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 69.1.1 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 69.1.1 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 69.1.2 所示的循环的队列。

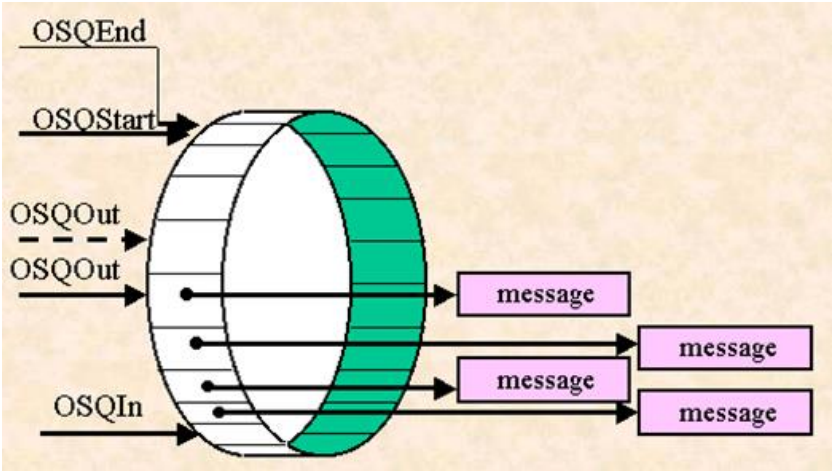


图 69.1.2 消息指针数组构成的环形数据缓冲区

在 UCOSII 初始化时，系统将按文件 os_cfg.h 中的配置常数 OS_MAX_QS 定义 OS_MAX_QS

个队列控制块，并用队列控制块中的指针 OSQPtr 将所有队列控制块链接为链表。由于这时还没有使用它们，故这个链表叫做空队列控制块链表。

接下来我们看看在 UCOSII 中，与消息队列相关的几个函数（未全部列出，下同）。

1） 创建消息队列函数

创建一个消息队列首先需要定义一指针数组，然后把各个消息数据缓冲区的首地址存入这个数组中，然后再调用函数 OSQCreate 来创建消息队列。创建消息队列函数 OSQCreate 的原型为：

```
OS_EVENT *OSQCreate(void**start,INT16U size);
```

其中，start 为存放消息缓冲区指针数组的地址，size 为该数组大小。该函数的返回值为消息队列指针。

2） 请求消息队列函数

请求消息队列的目的是为了从消息队列中获取消息。任务请求消息队列需要调用函数 OSQPend，该函数原型为：

```
void*OSQPend(OS_EVENT*pevent,INT16U timeout,INT8U *err);
```

其中，pevent 为所请求的消息队列的指针，timeout 为任务等待时限，err 为错误信息。

3） 向消息队列发送消息函数

任务可以通过调用函数 OSQPost 或 OSQPostFront 两个函数来向消息队列发送消息。函数 OSQPost 以 FIFO（先进先出）的方式组织消息队列，函数 OSQPostFront 以 LIFO（后进先出）的方式组织消息队列。这两个函数的原型分别为：

```
INT8U OSQPost(OS_EVENT *pevent,void *msg);
INT8U OSQPostFront (OS_EVENT*pevent,void*msg);
```

其中，pevent 为消息队列的指针，msg 为待发消息的指针。

消息队列还有其他一些函数，这里我们就不介绍了，感兴趣的朋友可以参考《嵌入式实时操作系统 UCOSII 原理及应用》第五章，关于队列更详细的介绍，也请参考该书。

信号量集

在实际应用中，任务常常需要与多个事件同步，即要根据多个信号量组合作用的结果来决定任务的运行方式。UCOSII 为了实现多个信号量组合的功能定义了一种特殊的数据结构——信号量集。

信号量集所能管理的信号量都是一些二值信号，所有信号量集实质上是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如图 69.1.3 所示

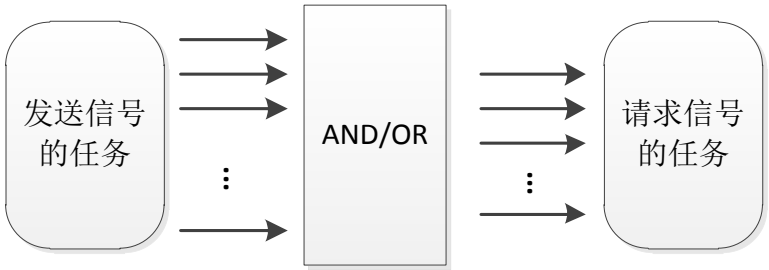


图 69.1.3 信号量集示意图

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 69.1.2 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 69.1.2 OSFlagNodeWaitType 可选值及其意义

OSFlagFlags、OSFlagNodeFlags、OSFlagNodeWaitType 三者的关系如图 69.1.4 所示：

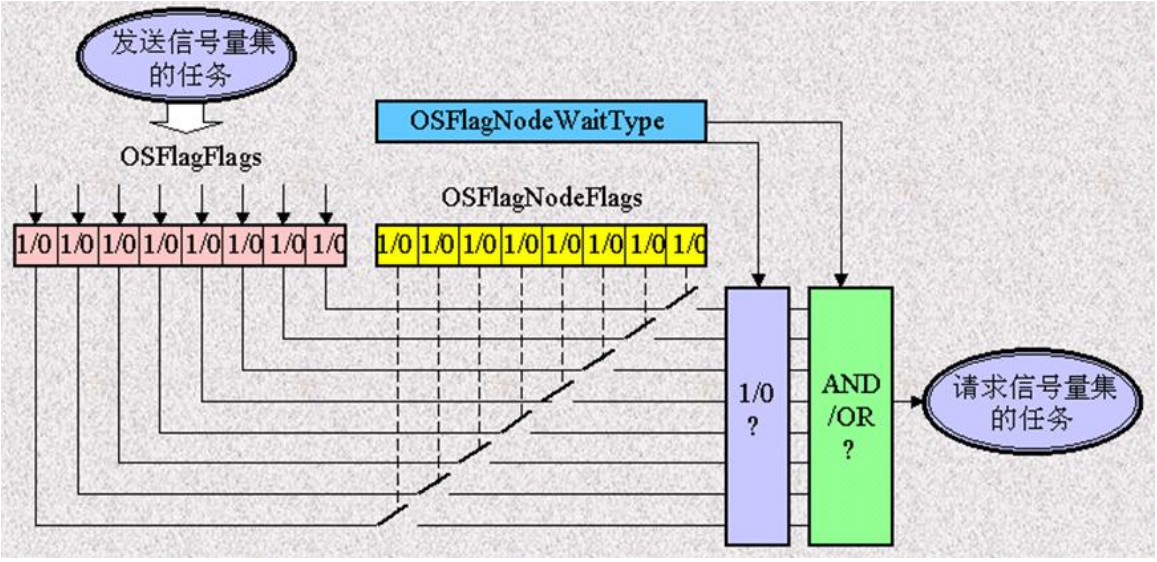


图 69.1.4 标志组与等待任务共同完成信号量集的逻辑运算及控制

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 69.1.5 所示：

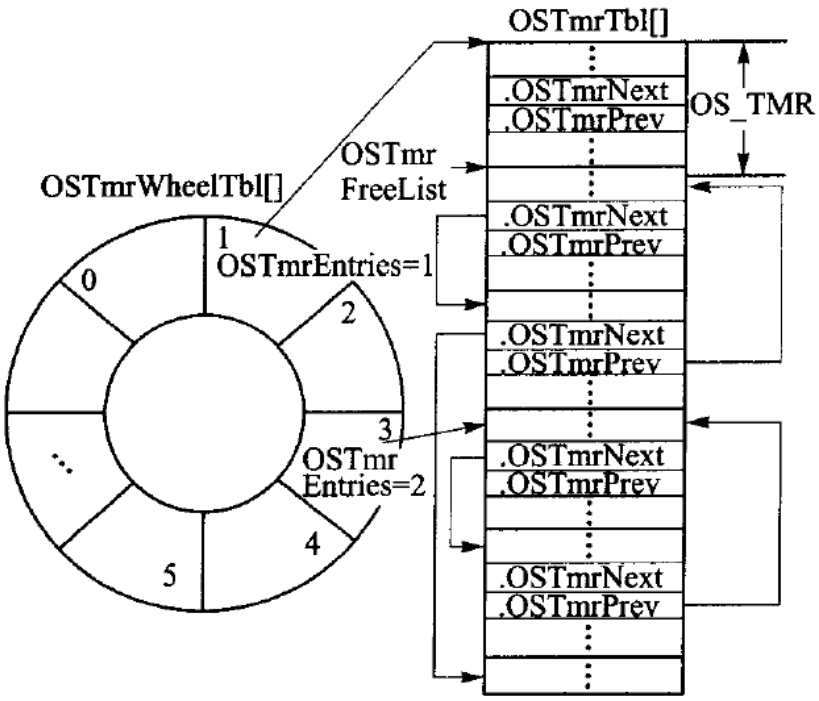


图 69.1.5 软件定时器管理所需的数据结构示意图

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 69.1.6 所示。

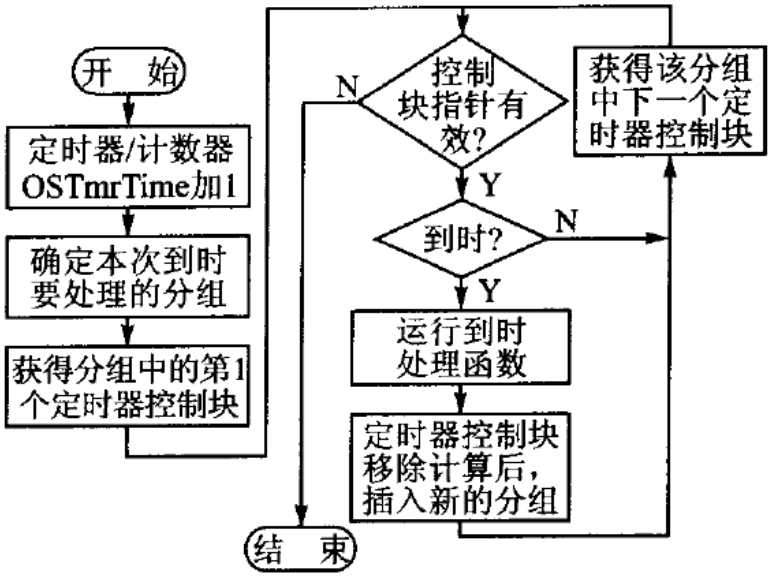


图 69.1.6 软件定时器管理任务流程

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。

69.2 硬件设计

本节实验功能简介：本章我们在 UCOSII 里面创建 7 个任务：开始任务、LED 任务、触摸屏任务、队列消息显示任务、信号量集任务、按键扫描任务和主任务，开始任务用于创建邮箱、消息队列、信号量集以及其他任务，之后挂起；触摸屏任务用于在屏幕上画图，测试 CPU 使用率；队列消息显示任务请求消息队列，在得到消息后显示收到的消息数据；信号量集任务用于测试信号量集，采用 OS_FLAG_WAIT_SET_ANY 的方法，任何按键按下（包括 TPAD），该任务都会控制蜂鸣器发出“滴”的一声；按键扫描任务用于按键扫描，优先级最高，将得到的键值通过消息邮箱发送出去；主任务创建 3 个软件定时器（定时器 1，100ms 溢出一次，显示 CPU 和内存使用率；定时 2，200ms 溢出一次，在固定区域不停的显示不同颜色；定时 3，,100ms 溢出一次，用于自动发送消息到消息队列），并通过查询消息邮箱获得键值，根据键值执行 DS1 控制、控制软件定时器 3 的开关、触摸区域清屏、触摸屏校和软件定时器 2 的开关控制等。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1

2） 4 个机械按键（KEY0/KEY1/KEY2/KEY_UP）

3） TPAD 触摸按键

5） PCF8574（控制蜂鸣器）

4） LCD 模块

这些，我们在前面的学习中都已经介绍过了。

69.3 软件设计

本章，我们在第四十四章实验 （实验 39）的基础上修改，首先，是 UCOSII 代码的添加，具体方法同第 67 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。

另外，我们还需要在 os_cfg.h 里面修改软件定时器管理部分的宏定义，修改如下：

```
#define OS_TMR_EN                       1u      //使能软件定时器功能
#define OS_TMR_CFG_MAX                  16u     //最大软件定时器个数
#define OS_TMR_CFG_NAME_EN              1u      //使能软件定时器命名
#define OS_TMR_CFG_WHEEL_SIZE           8u      //软件定时器轮大小
```

```
#define OS_TMR_CFG_TICKS_PER_SEC    100u    //软件定时器的时钟节拍（10ms）
#define OS_TASK_TMR_PRIO            0u      //软件定时器的优先级,设置为最高
```

这样我们就使能 UCOSII 的软件定时器功能了，并且设置最大软件定时器个数为 16，定时器轮大小为 8，软件定时器时钟节拍为 10ms（即定时器的最少溢出时间为 10ms）。

最后，我们只需要修改 main.c 函数了，打开 main.c，输入如下代码：

```
//////////////////////////UCOSII 任务设置//////////////////////////////////
//START 任务
#define START_TASK_PRIO                 10          //设置任务优先级
#define START_STK_SIZE                  128         //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                    128         //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)
{
    …//此处省略函数定义
}
int main(void)
{
Cache_Enable();                 //打开 L1-Cache
    HAL_Init();                 //初始化 HAL 库
    Stm32_Clock_Init(432,25,2,9);   //设置时钟,216Mhz
    delay_init(216);            //延时初始化
    uart_init(115200);          //串口初始化
    LED_Init();                 //初始化 LED
    KEY_Init();                 //初始化按键
    PCF8574_Init();             //初始化 PCF8574
    SDRAM_Init();               //初始化 SDRAM
    LCD_Init();                 //初始化 LCD
    TPAD_Init(8);               //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();                   //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
    OSStart(); //开始任务
}
////////////////////////////////////////////////////////////////////////////////////////////////
//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)
{
   …//此处省略函数定义
}
//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)
{
    …//此处省略函数定义
}
//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)
{
    …//此处省略函数定义
}
////////////////////////////////////////////////////////////////////////////////////////////////
//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();  //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);
    OSTaskCreateExt(……//省略部分代码);    //触摸任务
    OSTaskCreateExt(……//省略部分代码);    //消息队列显示任务
    OSTaskCreateExt(……//省略部分代码);    //主任务
    OSTaskCreateExt(……//省略部分代码);    //信号量集任务
    OSTaskCreateExt(……//省略部分代码);    //按键任务
    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)
{
    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];
                }
                //进入临界段,防止其他任务,打断 LCD 操作,导致液晶乱序.
                OS_ENTER_CRITICAL();
                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;
            delay_ms(10); //没有按键按下的时候
        }
    }
}
//队列消息显示任务
void qmsgshow_task(void *pdata)
{
    u8 *p,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,\    //100ms 执行一次
				(OS_TMR_CALLBACK)tmr1_callback,0,"tmr1",&err);
	tmr2=OSTmrCreate(10,20,OS_TMR_OPT_PERIODIC,\    //200ms 执行一次
				(OS_TMR_CALLBACK)tmr2_callback,0,"tmr2",&err);
	tmr3=OSTmrCreate(10,10,OS_TMR_OPT_PERIODIC,\    //100ms 执行一次
				(OS_TMR_CALLBACK)tmr3_callback,0,"tmr3",&err);
	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
		   //设置对应的信号量为 1
			OSFlagPost(flags_key,1<<(key-1),OS_FLAG_SET,&err);
		}
		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_Toggle;break;
			case 2://控制软件定时器 3
				tmr3sta=!tmr3sta;
				if(tmr3sta)OSTmrStart(tmr3,&err);
				else OSTmrStop(tmr3,OS_TMR_OPT_NONE,0,&err);//关闭软件定时器 3
				break;
			case 3://清除
				LCD_Fill(0,221,129,lcddev.height-1,WHITE);
				break;
			case 4://校准
```

```
                OSTaskSuspend(TOUCH_TASK_PRIO);//挂起触摸屏任务
                OSTaskSuspend(QMSGSHOW_TASK_PRIO);//挂起队列信息显示任务

                OSTmrStop(tmr1,OS_TMR_OPT_NONE,0,&err);//关闭软件定时器 1
                //关闭软件定时器 2
                if(tmr2sta)OSTmrStop(tmr2,OS_TMR_OPT_NONE,0,&err);
                if((tp_dev.touchtype&0X80)==0)TP_Adjust();
                OSTmrStart(tmr1,&err);                  //重新开启软件定时器 1
                if(tmr2sta)OSTmrStart(tmr2,&err);       //重新开启软件定时器 2
                OSTaskResume(TOUCH_TASK_PRIO);          //解挂
                OSTaskResume(QMSGSHOW_TASK_PRIO);       //解挂
                ucos_load_main_ui();                    //重新加载主界面
                break;
            case 5://软件定时器 2 开关
                tmr2sta=!tmr2sta;
                if(tmr2sta)OSTmrStart(tmr2,&err);       //开启软件定时器 2
                else
                {
                    //关闭软件定时器 2
                    OSTmrStop(tmr2,OS_TMR_OPT_NONE,0,&err);
                    //提示定时器 2 关闭了
                    LCD_ShowString(148,262,240,16,16,"TMR2 STOP");
                }
                break;
        }
        delay_ms(10);
    }
}
//信号量集处理任务
void flags_task(void *pdata)
{
    u16 flags;
    u8 err;
    while(1)
    {
        //等待信号量
        flags=OSFlagPend(flags_key,0X001F,OS_FLAG_WAIT_SET_ANY,0,&err);
        if(flags&0X0001)LCD_ShowString(140,162,240,16,16,"KEY0 DOWN    ");
        if(flags&0X0002)LCD_ShowString(140,162,240,16,16,"KEY1 DOWN    ");
        if(flags&0X0004)LCD_ShowString(140,162,240,16,16,"KEY2 DOWN    ");
        if(flags&0X0008)LCD_ShowString(140,162,240,16,16,"KEY_UP DOWN");
        if(flags&0X0010)LCD_ShowString(140,162,240,16,16,"TPAD DOWN    ");
        PCF8574_WriteBit(BEEP_IO,0);
```

```
            delay_ms(50);
            PCF8574_WriteBit(BEEP_IO,1);
            OSFlagPost(flags_key,0X001F,OS_FLAG_CLR,&err);//全部信号量清零
        }
    }
    //按键扫描任务
    void key_task(void *pdata)
    {
        u8 key;
        while(1)
        {
            key=KEY_Scan(0);
            if(key==0)
            {
                if(TPAD_Scan(0))key=5;
            }
            if(key)OSMboxPost(msg_key,(void*)key);//发送消息
            delay_ms(10);
        }
    }
```

本章 main.c 的代码有点多，因为我们创建了 7 个任务，3 个软件定时器及其回调函数，所以，整个代码有点多，我们创建的 7 个任务为：start_task、led_task、touch_task、qmsgshow_task 、flags_task 、main_task 和 key_task，优先级分别是 10 和 7~2，堆栈大小都是 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 69.2 节介绍的功能：KEY0 控制 LED1 亮灭；KEY1 控制软件定时器 tmr3 的开关，间接控制队列信息的发送；KEY2 清除触摸屏输入；KEY_UP 用于触摸屏校准，在校准的时候，要先挂起触摸屏任务、队列消息显示任务，并停止软件定时器 tmr1 和 tmr2，否则可能对校准时的 LCD 显示造成干扰；TPAD 按键用于控制软件定时器 tmr2 的开关，间接控制屏幕显示。

软件设计部分就为大家介绍到这里。

69.4 下载验证

在代码编译成功之后，我们通过下载代码到阿波罗 STM32 开发板上，可以看到 LCD 显示界面如图 69.4.1 所示：

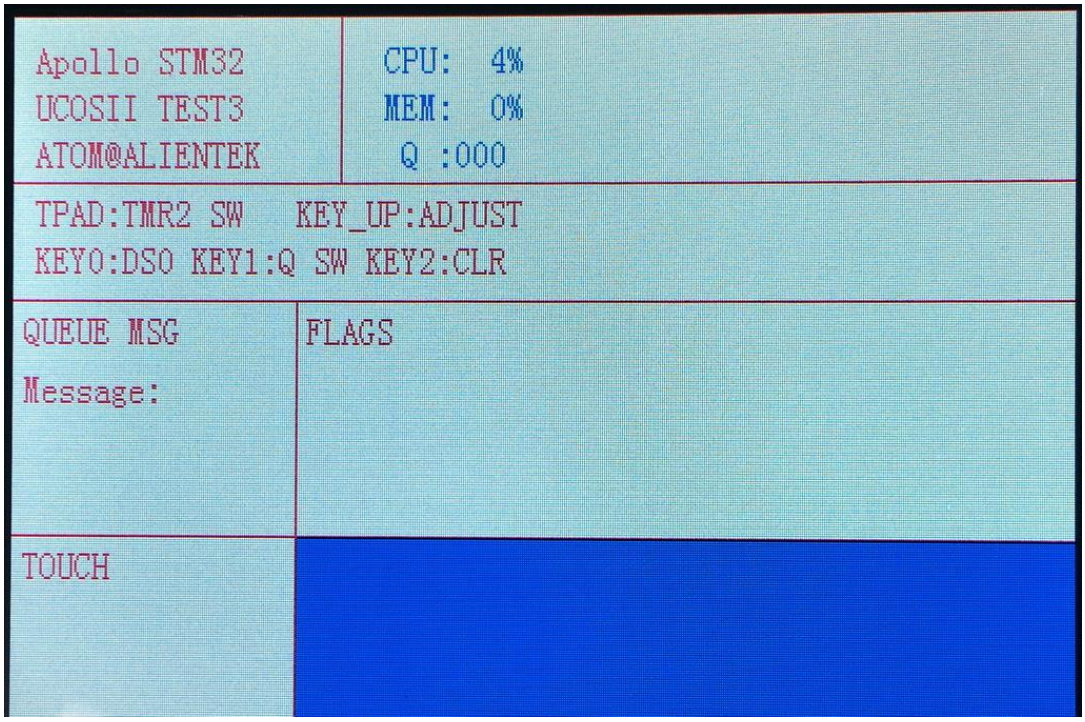


图 69.4.1 初始界面

从图中可以看出，默认状态下，CPU 使用率为 7%左右。比上一章多出一些，这主要是 key_task 里面增加不停的刷屏（tmr2）操作导致的。

通过按 KEY0，可以控制 DS1 的亮灭；

通过按 KEY1 则可以启动 tmr3 控制消息队列发送，可以在 LCD 上面看到 Q 和 MEM 的值慢慢变大（说明队列消息在增多，占用内存也随着消息增多而增大），在 QUEUE MSG 区，开始显示队列消息，再按一次 KEY1 停止 tmr3，此时可以看到 Q 和 MEM 逐渐减小。当 Q 值变为 0 的时候，QUEUE MSG 也停止显示（队列为空）。

通过 KEY2 按键，清除 TOUCH 区域的输入。

通过 KEY_UP 按键，可以进行触摸屏校准(仅电阻屏，电容屏无需校准)。

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。

正点原子
2016-10-28
于广州

广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛 (开源电子网) ：www.openedv.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com **QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友 情 提 示

如果您想及时免费获取“正点原子”最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。

关注方法：

(1)微信“扫一扫”，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入“正点原子”→关注

(3)微信→添加朋友→输入“alientek_stm32”→关注