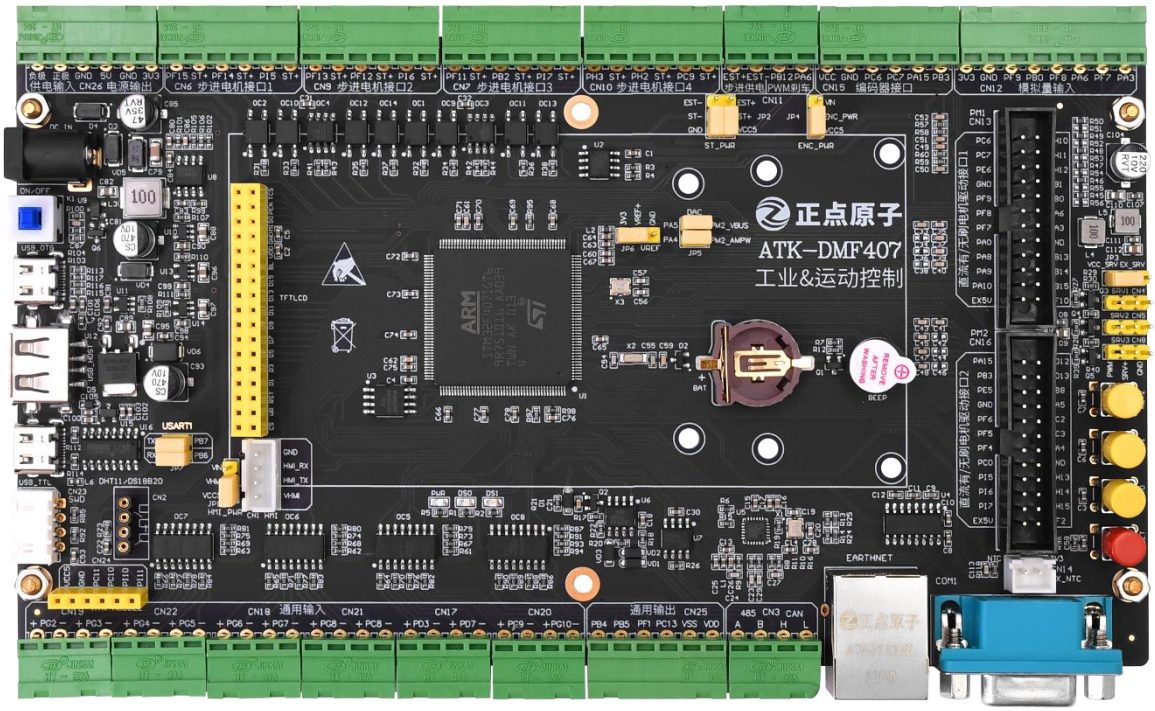
DMF407 开发指南

V1.1

-正点原子 DMF407 开发板教程



修订历史：

版本	日期	修改内容
V1.0	2022/06/06	第一次发布
V1.1	2023/06/05	更新 LCD 驱动代码，修正一些描述错误

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

电话：020-38271790 传真：020-36773971

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

扫码下载“原子哥”APP

扫码关注正点原子公众号

内容简介 1
第一篇 基础篇 2
第一章 本书学习方法 3
1.1 本书学习顺序 3
1.2 本书参考资料 3
1.3 本书编写规范 3
1.4 本书代码规范 4
1.5 例程资源说明 4
1.6 学习资料查找 5
1.7 给初学者的建议 7
第二章 STM32 简介 8
2.1 初识 STM32 8
2.2 STM32F407 资源简介 9
2.3 STM32F407 设计选型 9
2.3.1 STM32 系列 9
2.3.2 STM32 命名 10
2.3.3 STM32 选型 11
2.3.4 STM32 设计 11
第三章 开发环境搭建 16
3.1 常用开发工具简介 16
3.2 MDK 安装 16
3.3 仿真器驱动安装 17
3.4 CH340 USB 虚拟串口驱动安装 17
第四章 STM32 初体验 19
4.1 使用 MDK5 编译例程 19
4.2 使用 DAP 下载与调试程序 22
4.2.1 使用 DAP 下载程序 23
4.2.2 使用 DAP 仿真调试程序 25
4.2.3 仿真调试注意事项 30
4.3 MDK5 使用技巧 31
4.3.1 文本美化 31
4.3.2 语法检测&代码提示 33
4.3.3 代码编辑技巧 35
4.3.4 其他小技巧 39
第五章 STM32 基础知识入门 41
5.1 C 语言基础知识复习 41
5.1.1 位操作 41
5.1.2 define 宏定义 42

5.1.3 ifdef 条件编译 42
5.1.4 extern 外部申明 42
5.1.5 typedef 类型别名 43
5.1.6 结构体 43
5.1.7 指针 44
5.2 寄存器基础知识 45
5.3 STM32F407 系统架构 46
5.3.1 Cortex M4 内核 & 芯片 46
5.3.2 STM32 系统架构 46
5.3.3 存储器映射 48
5.3.4 寄存器映射 51
第六章 新建寄存器版本 MDK 工程 55
6.1 新建寄存器版本 MDK 工程 55
6.1.1 新建工程文件夹 55
6.1.2 新建一个工程框架 57
6.1.3 添加文件 59
6.1.4 魔术棒设置 64
6.1.5 添加 main.c，并编写代码 69
6.2 下载验证 71
第七章 认识 HAL 库 72
7.1 初识 STM32 HAL 库 72
7.1.1 CMSIS 标准 72
7.1.2 HAL 库简介 73
7.1.3 HAL 库能做什么 75
7.2 HAL 库驱动包 75
7.2.1 如何获取 HAL 库固件包 76
7.2.2 STM32Cube 固件包分析 77
7.2.3 CMSIS 文件夹关键文件 79
7.2.4 stdint.h 简介 81
7.3 HAL 库框架结构 81
7.3.1 HAL 库文件夹结构 81
7.3.2 HAL 库文件介绍 82
7.4 如何使用 HAL 库 85
7.4.2 HAL 库的用户配置文件 86
7.4.3 stm32f4xx_hal.c 文件 88
7.4.4 HAL 库中断处理 92
7.4.5 正点原子对 HAL 库用法的个性化修改 92
7.5 HAL 库使用注意事项 92
第八章 新建 HAL 版本 MDK 工程 94
8.1 新建 HAL 库版本 MDK 工程 94
8.1.1 新建工程文件夹 94

8.1.2 新建一个工程框架 96
8.1.3 添加文件 99
8.1.4 魔术棒设置 105
8.1.5 添加 main.c，并编写代码 109
8.2 下载验证 112
第九章 STM32 启动过程分析 113
9.1 启动模式 113
9.2 启动文件分析 114
9.2.1 启动文件中的一些指令 114
9.2.2 启动文件代码讲解 116
9.2.3 系统启动流程 120
9.3 map 文件分析 122
9.3.1 MDK 编译生成文件简介 122
9.3.2 map 文件分析 123
第十章 STM32CubeMX 简介 127
10.1 STM32CubeMX 的作用 127
10.2 安装 STM32CubeMX 128
10.2.1 安装 JAVA 环境 128
10.2.2 安装 STM32CubeMX 128
10.3 使用 STM32CubeMX 新建工程 131
10.3.1 打开 STM32CubeMX 131
10.3.2 下载和关联的 STM32Cube 固件包 132
10.3.3 新建工程 133
10.4 STM32CubeMX 新建工程使用建议 142
第十一章 STM32 时钟系统 144
11.1 认识时钟树 144
11.1.1 时钟源 145
11.1.2 锁相环 PLL 146
11.1.3 系统时钟 SYSCLK 147
11.2 如何修改主频 149
11.2.1 STM32F4 时钟系统配置 149
11.2.2 STM32F4 时钟使能和配置 154
第十二章 SYSTEM 文件夹介绍 156
12.1 deley 文件夹代码介绍 156
12.1.1 操作系统支持宏定义及相关函数 157
12.1.2 delay_init 函数 159
12.1.3 delay_us 函数 160
12.1.4 delay_ms 函数 161
12.1.5 HAL 库延时函数 HAL_Delay 162
12.2 sys 文件夹代码介绍 163

12.3 usart 文件夹代码介绍 164
第二篇 入门篇 166
第十三章 跑马灯实验 167
13.1 STM32F407 GPIO 简介 167
13.1.1 GPIO 功能模式 167
13.1.2 GPIO 基本结构分析 168
13.1.3 GPIO 寄存器介绍 173
13.2 硬件设计 176
1. 例程功能 176
2. 硬件资源 176
3. 原理图 176
13.3 程序设计 177
13.3.1 GPIO 的 HAL 库驱动分析 177
13.3.2 程序流程图 179
13.3.3 程序解析 179
13.4 下载验证 181
第十四章 蜂鸣器实验 182
14.1 蜂鸣器简介 182
14.2 硬件设计 182
1. 例程功能 182
2. 硬件资源 182
3. 原理图 183
14.3 程序设计 183
14.3.1 程序流程图 183
14.3.2 程序解析 184
14.4 下载验证 185
第十五章 按键输入实验 186
15.1 按键与输入数据寄存器简介 186
15.1.1 独立按键简介 186
15.1.2 GPIO 端口输入数据寄存器（IDR） 186
15.2 硬件设计 187
1. 例程功能 187
2. 硬件资源 187
3. 原理图 187
15.3 程序设计 188
15.3.1 HAL_GPIO_ReadPin 函数 188
15.3.2 程序流程图 188
15.3.3 程序解析 189
15.4 下载验证 191
第十六章 外部中断实验 192

16.1 NVIC 和 EXTI 简介 192
16.1.1 NVIC 简介 192
16.1.2 EXTI 简介 195
16.2 硬件设计 197
1. 例程功能 197
2. 硬件资源 197
3. 原理图 198
16.3 程序设计 198
16.3.1 EXTI 的 HAL 库驱动 198
16.3.2 程序流程图 199
16.3.3 程序解析 200
16.4 下载验证 203
第十七章 串口通信实验 204
17.1 串口简介 204
17.1.1 数据通信的基础概念 204
17.1.2 串口通信协议简介 206
17.1.3 STM32F407 的串口简介 207
17.1.4 GPIO 引脚复用功能 211
17.2 硬件设计 213
1. 例程功能 213
2. 硬件资源 213
3. 原理图 213
17.3 程序设计 213
17.3.1 USART 的 HAL 库驱动 213
17.3.2 程序流程图 217
17.3.3 程序解析 217
17.4 下载验证 222
第十八章 独立看门狗（IWDG）实验 224
18.1 IWDG 简介 224
18.1.1 IWDG 框图 224
18.1.2 IWDG 寄存器 225
18.2 硬件设计 226
1. 例程功能 226
2. 硬件资源 226
3. 原理图 226
18.3 程序设计 226
18.3.1 IWDG 的 HAL 库驱动 226
18.3.2 程序流程图 228
18.3.3 程序解析 228
18.4 下载验证 229

第十九章 窗口门狗（WWDG）实验 230
19.1 WWDG 简介 230
19.1.1 WWDG 框图 230
19.1.2 WWDG 寄存器 231
19.2 硬件设计 233
1. 例程功能 233
2. 硬件资源 233
3. 原理图 233
19.3 程序设计 233
19.3.1 WWDG 的 HAL 库驱动 233
19.3.2 程序流程图 234
19.3.3 程序解析 235
19.4 下载验证 236
第二十章 TFT-LCD（MCU 屏）实验 238
20.1 TFT-LCD 简介 238
20.1.1 TFT-LCD 简介 238
20.1.2 液晶显示控制器 240
20.1.3 FSMC 简介 243
20.1.4 FSMC 关联寄存器简介 248
20.2 硬件设计 251
1. 例程功能 251
2. 硬件资源 251
3. 原理图 251
20.3 程序设计 252
20.3.1 FSMC 和 SRAM 的 HAL 库驱动 252
20.3.2 程序流程图 255
20.3.2 程序解析 255
20.4 下载验证 267
第二十一章 USMART 调试组件实验 268
21.1 USMART 调试组件简介 268
21.2 硬件设计 269
1. 例程功能 269
2. 硬件资源 269
21.3 程序设计 269
21.3.1 程序流程图 269
21.3.2 程序解析 270
21.4 下载验证 273
第二十二章 RTC 实时时钟实验 277
22.1 RTC 时钟简介 277
22.1.1 RTC 框图 277

22.1.2 RTC 寄存器 279
22.2 硬件设计 284
1. 例程功能 284
2. 硬件资源 284
3. 原理图 285
22.3 程序设计 285
22.3.1 RTC 的 HAL 库驱动 285
22.3.2 程序流程图 289
22.3.3 程序解析 289
22.4 下载验证 294
第二十三章 硬件随机数实验 296
23.1 随机数发生器简介 296
23.1.1 RNG 框图 296
23.1.2 RNG 寄存器 297
23.2 硬件设计 298
1. 例程功能 298
2. 硬件资源 298
3. 原理图 298
23.3 程序设计 298
23.3.1 RNG 的 HAL 库驱动 298
23.3.2 程序流程图 300
23.3.3 程序解析 300
23.4 下载验证 303
第二十四章 低功耗实验 304
24.1 电源控制（PWR）简介 304
24.1.1 电源系统 304
24.1.2 电源监控 306
24.1.3 电源管理 308
24.2 PVD 电压监控实验 311
24.2.1 PWR 寄存器 311
24.2.2 硬件设计 313
24.2.3 程序设计 313
24.2.4 下载验证 316
24.3 睡眠模式实验 317
24.3.1 EXTI 寄存器 317
24.3.2 硬件设计 318
24.3.3 程序设计 318
24.3.4 下载验证 322
24.4 停止模式实验 323
24.4.1 PWR 寄存器 323
24.4.2 硬件设计 324

24.4.3 程序设计 324
24.4.4 下载验证 327
24.5 待机模式实验 328
24.5.1 PWR 寄存器 328
24.5.2 硬件设计 328
24.5.3 程序设计 329
24.5.4 下载验证 331
第二十五章 DMA 实验 332
25.1 DMA 简介 332
25.1.1 DMA 框图 333
25.1.2 DMA 寄存器 335
25.2 硬件设计 337
1. 例程功能 337
2. 硬件资源 337
3. 原理图 337
25.3 程序设计 337
25.3.1 DMA 的 HAL 库驱动 337
25.3.2 程序流程图 340
25.3.3 程序解析 340
25.4 下载验证 343
第二十六章 IIC 实验 345
26.1 IIC 及 24C02 介绍 345
26.1.1 IIC 简介 345
26.1.2 24C02 简介 347
26.2 硬件设计 349
1. 例程功能 349
2. 硬件资源 349
3. 原理图 349
26.3 程序设计 349
使用 IIC 传输数据的配置步骤 349
26.3.1 程序流程图 350
26.3.2 程序解析 350
26.4 下载验证 360
第二十七章 SPI 实验 362
27.1 SPI 及 NOR Flash 介绍 362
27.1.1 SPI 介绍 362
27.1.1.1 SPI 框图 362
27.1.2 NOR Flash 简介 366
27.2 硬件设计 369
1. 例程功能 369
2. 硬件资源 369

3. 原理图 369
27.3 程序设计 370
27.3.1 SPI 的 HAL 库驱动 370
27.3.2 程序流程图 372
27.3.3 程序解析 372
27.4 下载验证 381
第二十八章 485 实验 382
28.1 485 简介 382
28.2 硬件设计 383
1. 例程功能 383
2. 硬件资源 383
3. 原理图 383
28.3 程序设计 384
28.3.1 RS485 的 HAL 库驱动 384
28.3.2 程序流程图 385
28.3.3 程序解析 385
28.4 下载验证 389
第二十九章 232 实验 391
29.1 232 简介 391
29.2 硬件设计 391
1. 例程功能 391
2. 硬件资源 391
3. 原理图 392
29.3 程序设计 392
29.3.1 RS232 的 HAL 库驱动 392
29.3.2 程序流程图 393
29.3.3 程序解析 393
29.4 下载验证 396
第三十章 CAN 通讯实验 399
30.1 CAN 总线简介 399
30.1.1 CAN 简介 399
30.1.2 CAN 协议 401
30.1.3 CAN 寄存器 410
30.2 硬件设计 415
1. 例程功能 415
2. 硬件资源 415
3. 原理图 415
30.3 程序设计 416
30.3.1 CAN 的 HAL 库驱动 416
CAN 的初始化配置步骤 418

30.3.2 程序流程图 419
30.3.3 程序解析 419
30.4 下载验证 424
第三十一章 触摸屏实验 426
31.1 触摸屏简介 426
31.1.1 电阻式触摸屏 426
31.1.2 电容式触摸屏 427
31.1.3 触摸控制原理 429
31.2 硬件设计 430
1. 例程功能 430
2. 硬件资源 430
3. 原理图 431
31.3 程序设计 431
31.3.1 HAL 库驱动 431
31.3.2 程序流程图 432
31.3.3 程序解析 432
31.4 下载验证 448
第三十二章 DS18B20 数字温度传感器实验 450
32.1 DS18B20 及其时序简介 450
32.1.1 DS18B20 简介 450
32.1.2 DS18B20 时序简介 450
32.2 硬件设计 452
1. 例程功能 452
2. 硬件资源 452
3. 原理图 452
32.3 程序设计 453
DS18B20 配置步骤 453
32.3.1 程序流程图 454
32.3.2 程序解析 454
32.4 下载验证 459
第三十三章 DHT11 数字温湿度传感器 460
33.1 DHT11 及时序简介 460
33.1.1 DHT11 简介 460
33.1.2 DHT11 时序简介 460
33.2 硬件设计 462
1. 例程功能 462
2. 硬件资源 462
3. 原理图 462
33.3 程序设计 463
DHT11 配置步骤 463

33.3.1 程序流程图 463
33.3.2 程序解析 464
33.4 下载验证 468
第三十四章 FLASH 模拟 EEPROM 实验 469
34.1 STM32F407 FLASH 简介 469
34.1.1 闪存的读取 470
34.1.2 闪存的编程和擦除 470
34.1.3 FLASH 寄存器 472
34.2 硬件设计 474
1. 例程功能 474
2. 硬件资源 474
34.3 程序设计 475
34.3.1 FLASH 的 HAL 库驱动 475
34.3.2 程序流程图 476
34.3.3 程序解析 476
34.4 下载验证 482
第三篇 提高篇 484
第三十五章 内存管理实验 485
35.1 内存管理简介 485
35.2 硬件设计 486
1. 例程功能 486
2. 硬件资源 486
35.3 程序设计 487
35.3.1 程序流程图 487
35.3.2 程序解析 487
35.4 下载验证 494
第三十六章 FPU 测试(Julia 分形)实验 497
36.1 FPU&Julia 分形简介 497
36.1.1 FPU 简介 497
36.1.2 Julia 分形简介 498
36.2 硬件设计 499
1. 例程功能 499
2. 硬件资源 499
36.3 程序设计 500
36.3.1 程序流程图 500
36.3.2 程序解析 500
36.4 下载验证 504
第三十七章 DSP 测试实验 506
37.1 DSP 简介与环境搭建 506
37.1.1 STM32F407 DSP 简介 506

37.1.2 DSP 库运行环境搭建 509
37.2 硬件设计 511
1. 例程功能 511
2. 硬件资源 511
37 程序设计 512
37.1 DSP BasicMath 测试 512
37.3.2 DSP FFT 测试 514
37.4 下载验证 517
第三十八章 串口 IAP 实验 519
38.1 IAP 简介 519
38.2 硬件设计 525
1. 例程功能 525
2. 硬件资源 525
38.3 程序设计 526
38.3.1 程序流程图 526
38.3.2 程序解析 526
38.4 下载验证 530
第三十九章 USB 读卡器(Slave)实验 532
39.1 USB 简介 532
39.1.1 USB 简介 532
39.1.2 STM32F4 的 USB 特性 534
39.2 硬件设计 538
1. 例程功能 538
2. 硬件资源 538
3. 原理图 538
39.3 程序设计 540
39.3.1 程序流程图 540
39.3.2 程序解析 540
39.4 下载验证 549
第四十章 USB 虚拟串口(Slave)实验 551
40.1 USB 虚拟串口简介 551
40.2 硬件设计 551
1. 例程功能 551
2. 硬件资源 551
40.3 程序设计 552
40.3.1 程序流程图 552
40.3.2 程序解析 552
40.4 下载验证 559
第四十一章 USB 鼠标键盘(Host)实验 561
41.1 USB 鼠标键盘简介 561

41.2 硬件设计 561
1. 例程功能 561
2. 硬件资源 561
3. 原理图 561
41.3 程序设计 563
41.3.1 程序流程图 563
41.3.2 程序解析 563
41.4 下载验证 573
第四十二章 USB U 盘(Host)实验 575
42.1 FATFS 简介 575
42.2 U 盘简介 579
42.3 硬件设计 579
1. 例程功能 579
2. 硬件资源 579
3. 原理图 580
42.4 程序设计 581
42.4.1 程序流程图 581
42.4.2 程序解析 581
42.5 下载验证 595
第四十三章 汉字显示实验 598
43.1 汉字显示介绍 598
43.1.1 字符编码介绍 598
43.1.2 字库的生成 602
43.1.3 汉字显示原理 604
43.2 硬件设计 604
1. 例程功能 604
2. 硬件资源 604
43.3 程序设计 605
43.3.1 程序流程图 605
43.3.2 程序解析 606
43.4 下载验证 614

内容简介

本书将由浅入深，带领大家学习 STM32F407 的各个功能，为您开启 STM32F407 的学习之旅。

本书总共分为三篇：

1， 基础篇，主要介绍 STM32F407 的基础入门知识，包括 STM32F407 介绍、软件安装、新建工程、时钟系统介绍、SYSTEM 文件夹介绍等。必须好好学习并掌握；

2， 入门篇，主要介绍 STM32F407 常用外设的使用，包括 GPIO、定时器、DMA、触摸液晶屏等。必须好好学习并掌握；

3， 提高篇，主要介绍 STM32F407 较难一点外设的使用及一些高级例程，包括：USB、内存管理、文件系统等。这些例程难度相对高一点，大家一定要沉下心来，戒骄戒躁，循序渐进的学习。对于一些较难的知识点，我们只需要知道怎么用就可以了，并不需要详细研究具体实现过程，比如文件系统、USB 协议栈等；

本书为正点原子 DMF407 电机开发板的配套教程，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。

本书不仅非常适合广大学生和电子爱好者学习 STM32F407，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。

第一篇 基础篇

万事开头难，如果打好了基础，后面学习就事半功倍了！本篇将详细介绍 STM32F407 学习的基础知识，包括：环境搭建、STM32 入门知识、新建工程、HAL 库介绍、启动过程分析、时钟系统、SYSTEM 文件夹介绍等部分。学好了这些基础知识，在后面的例程学习部分，将会有非常大的帮助，能极大的提高大家的学习效率。

如果您是初学者，建议好好学习本并理解这些知识，手脑并用，不要漏过任何内容，一遍学不会的可以多学几遍，总之这些知识点都要掌握。

如果您已经学过 STM32 了，本篇内容则可以挑选着学习。

本篇将分为如下章节：

1，本书学习方法
2，STM32 简介
3，开发环境搭建
4，STM32 初体验
5，STM32 基础知识入门
6，新建寄存器版本 MDK 工程
7，认识 HAL 库
8，新建 HAL 库版本 MDK 工程
9，STM32 启动过程分析
10，STM32CubeMX 简介
11，STM32 时钟系统
12，SYSTEM 文件夹简介

第一章 本书学习方法

为了让大家更好的学习和使用本书，本章将给大家介绍一下本书的学习方法，包括：本书的学习顺序、编写规范、代码规范、资料查找、学习建议等内容。

本章将分为如下几个小节：

1.1 本书学习顺序
1.2 本书参考资料
1.3 本书编写规范
1.4 本书代码规范
1.5 例程资源说明
1.6 学习资料查找
1.7 给初学者的建议

1.1 本书学习顺序

为了让大家更好的学习和使用本书，我们做了以下几点考虑：

1，坚持循序渐进的思路编写，从基础到入门，从简单到复杂。

2，将知识进行分类介绍，简化学习过程，包括：基础篇、入门篇、提高篇。

3，将硬件介绍独立成一个文档（《ATK-DMF407 硬件参考手册.pdf》），本书着重介绍软件知识。

因此，大家在学习本书的时候，我们建议：先通读一遍《ATK-DMF407 硬件参考手册.pdf》，对开发板的硬件资源有个大概了解，然后从基础篇开始，再到入门篇，最后学习提高篇，循序渐进，逐一攻克。

对初学者来说，尤其要按照以上顺序学习，不要跳跃式学习，因为我们书本的知识都是一环扣一环的，如果前面的知识没学好，后面的知识学起来就会很困难。

对于已经学过 STM32 的朋友来说，就可以跳跃式学习了，当然如有不懂，也得翻阅前面的知识点进行巩固。

1.2 本书参考资料

本书的主要参考资料有以下四份文档：

《STM32F4xx 参考手册_V4（中文版）.pdf》
《STM32F4xx 参考手册_V6（英文版）.pdf》
《STM32F407IGT6.pdf》
《Cortex-M3 与 M4 权威指南.pdf》

前两份是 ST 官方针对 STM32 的一份通用参考资料，重点介绍 STM32 内部资源及使用，寄存器描述等，内容翔实，但是没有实例，也没有对 Cortex-M4 构架进行多少介绍，读者只能根据自己对书本的理解来编写相关代码。

《STM32F407IGT6.pdf》是对 STM32F407IGT6 芯片的英文介绍。

《Cortex-M3 与 M4 权威指南.pdf》是专门介绍 Cortex-M4 构架的书，有简短的实例，但没有专门针对 STM32 的介绍。所以，在学习 STM32 的时候，必须结合这份资料来看。

1.3 本书编写规范

本书通过数十个例程，给大家详细介绍 STM32 的所有功能和外设，按难易程度以及知识结构，我们将本书分为三个篇章：基础篇、入门篇和提高篇。

基础篇，共 12 章，主要是一些基础知识介绍，包括开发环境搭建、新建工程、HAL 库介绍、时钟树介绍、SYSTEM 文件夹介绍等，这些章节在结构上没有共性，但是互相有关联，有一个集成的关系在里面，即：必须先学了前面的知识，才好学习后面的知识点。

入门篇和提高篇，共三十一章，详细介绍了 STM32F407 每一个外设的使用方法及驱动代码，并且还介绍了一些非常实用的程序代码（纯软件例程），如：内存管理、文件系统读写、IAP 等。这部分内容占了本书的绝大部分篇幅，而且这些章节在结构上都比较有共性，一般分为 4 个部分，如下：

1，外设功能介绍
2，硬件设计
3，程序设计
4，下载验证

外设功能介绍，简单介绍具体章节所要用到的外设功能、框图和寄存器等，让大家对所用外设的功能有一个基本了解，方便后面的程序设计。

硬件设计，包括具体章节的实验具体功能说明、所用到的硬件资源及原理图连接方式，从而知道要做什么？ 需要用到哪些 IO 口？ 是怎么接线的？ 方便程序设计的时候编写驱动代码。

程序设计，一般包括：驱动介绍、配置步骤、程序流程图、关键代码分析、main 函数讲解等三部分。一点点介绍程序代码是怎么来的，注意事项等，从而学会整个代码。

下载验证，属于实践环节，在完成程序设计后，教大家如何下载并验证我们的例程是否正确？ 完成一个闭环过程。

1.4 本书代码规范

为了方便大家编写高质量代码，我们对本书的代码风格进行了统一，详细的代码规范说明文档，见光盘：A 盘→1，入门资料→《嵌入式单片机 C 代码规范与风格.pdf》，初学者务必好好学习一下这个文档。

总结几个规范的关键点：

1，所有函数/变量名字非特殊情况，一般使用小写字母；
2，注释风格使用 doxgen 风格，除屏蔽外，一律使用 /* */ 方式进行注释；
3，TAB 键统一使用 4 个空格对齐，不使用默认的方式进行对齐；
4，每两个函数之间，一般有且只有一个空行；
5，相对独立的程序块之间，使用一个空行隔开；
6，全局变量命名一般用 g_开头，全局指针命名一般用 p_开头；
7，if、for、while、do、case、switch、default 等语句单独占一行，一般无论有多少行执行语句，都要用加括号：{}。

1.5 例程资源说明

DMF407 提供的标准例程多达 31 个，提供：HAL 库版本的代码（本手册以 HAL 库版本例程做为介绍，**我们不再提供寄存器版本文档教程，也没有寄存器版本例程**）。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。

DMF407 的例程列表如表 1.5.1 所示：

编号	实验名字	编号	实验名字
1	跑马灯实验	16	485 实验
2	蜂鸣器实验	17	232 实验
3	按键输入实验	18	CAN 实验
4	外部中断实验	19	触摸屏实验
5	串口通信实验	20	DS18B20 数字温度传感器实验
6	独立看门狗实验	21	DHT11 数字温湿度传感器实验
7	窗口看门狗实验	22	FLASH 模拟 EEPROM 实验
8	TFTLCD（MCU 屏）实验	23	内存管理实验
9	USMART 调试实验	24_1	FPU 测试(Julia 分形)实验_开启

10	RTC 实验	24_2	FPU 测试(Julia 分形)实验_关闭
11	随机数发生器实验	25_1	DSP BasicMath 测试实验
12-1	PVD 电压监控实验	25_2	DSP FFT 测试实验
12-2	睡眠模式实验	26	串口 IAP 实验
12-3	停止模式实验	27	USB 读卡器(Slave)实验
12-4	待机模式实验	28	USB 虚拟串口(Slave)实验
12	DMA 实验	29	USB U 盘(Host)实验
13	IIC 实验	30	USB 鼠标键盘实验(Host)
14	SPI 实验	31	汉字显示实验

表 1.5.1 DMF407 例程表

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，正点原子 DMF407 是非常适合初学者的。当然，对于想深入了解 STM32F407 内部资源的朋友，正点原子 DMF407 也绝对是一个不错的选择。

1.6 学习资料查找

学习资料包括三个方面：

1，ST 官方的学习资料

ST 官方资料有两个网址：www.stmcu.org 和 www.st.com 。

www.stmcu.org 是 ST 中文社区，里面的资料全部由 ST 中国区的人负责更新和整理，包含了所有 ST 公司的 MCU 资料，比如：STM32F4 最新的芯片文档（参考手册、数据手册、勘误手册、编程手册等）、软件资源（固件库、配置工具、PC 软件等）、硬件资源（各种官方评估板）等，如图 1.6.1 所示：



图 1.6.1 STM32F4 相关资料（stmcu.org）

www.st.com 是 ST 官网，ST 最新最全的资料，一般都是放在该网站，ST 中文社区的资料，一般都是从 ST 官网搬过来的，所以如果你想找最新的 STM32 官方资料，应该在 ST 官网找。对于初学者，一般从 ST 中文社区获取 ST 官方资料就可以了。ST 官网的 STM32F4 资料页面如

图 1.6.2 所示（注意：默认是英文语言，需要在网页右上角设置成中文）：



图 1.6.2 STM32F4 官网资料（st.com）

① STM32F407 的硬件相关资源，在：产品→微控制器→STM32 ARM Cortex 32 位微控制器→STM32 主流 MCU→STM32F4 系列→STM32F4→文件路径下面可以找到。

② STM32F407 的软件相关资源，在：工具与软件→嵌入式软件→微控制器软件→STM32 微控制器软件 路径下面可以找到。

2，正点原子的学习资料

正点原子给大家提供的学习资料，都放在正点原子文档中心，大家可以在文档中心下载所有正点原子最新最全的学习资料，文档中心地址：www.openedv.com/docs/index.html ，如图 1.6.3 所示：



图 1.6.3 正点原子文档中心

在文档中心下面，我们可以找到正点原子所有开发板、模块、产品等的详细资料下载地址。

3，正点原子论坛

正点原子论坛，即开源电子网：www.openedv.com/forum.php ，该论坛从 2010 年成立至今，已有 10 年时间，数十万注册用户，STM32 相关帖子数量有 20 多万，每天数百人互动，是一个非常好的 STM32 学习交流平台。

在学习过程中，我们难免会遇到一些问题，有任何问题，大家都可以先去开源电子网搜索一下，基本上你能遇到的问题，我们论坛都有人问过了，所以可以很方便的找到一些参考解决

方法。如果实在找不到，你也可以在论坛提问，每天原子哥都会在上面给大家做解答。

不过，大家在论坛发帖的时候，建议先阅读一下提问的智慧，缕清思路，不要乱问一通，提高提问质量，提问的智慧阅读地址：www.openedv.com/thread-7245-1-1.html 。

1.7 给初学者的建议

对于学习 STM32，这里我给大家提以下三点建议：

1，准备一款合适的开发板（强烈建议配仿真器）

任何实验，我们都需要验证，最好的验证方式就是在开发板上面实际跑起来，然后通过仿真器仿真调试，查看具体的执行过程。仿真调试可以加深印象，还可以方便的查找 bug，所以学习 STM32 必备：一个开发板 + 一个仿真器。

另外，开发板在精不在多，学好一款，基本上就够用了。

2，两本参考资料，即《STM32F4xx 参考手册_V4（中文版）.pdf》和《Cortex-M3 与 M4 权威指南》。

这两个手册在本章 1.2 节有过介绍，对于我们学习 STM32 和了解 Cortex M4 内核非常有帮助，是学习 STM32 的必备资料，因此初学者尤其要多看这两个手册。

这里建议大家，要多了解一些底层的东西（可结合这两个手册，看看寄存器版本的例程），不要只会使用库，否则，一旦遇到问题，或者换个芯片，就不知道怎么办了。

3，戒骄戒躁，勤思敏行。

学习 STM32 千万不能浮躁，更加不能骄傲，初学者学习 STM32 会遇到很多问题和难点，这个时候千万不能浮躁，不要带情绪，一定要静下心来，缕清思路，逐一攻克。

我就曾经遇到一个问题半个月都没解决的情况，但是这半个月我尝试并掌握了很多解决问题的方法，最终解决问题的时候，其实带来的收获远远大于问题本身。所以不要遇到问题就认怂，就想问别人，问老师，先尝试自己解决一下，比如花个十天半个月去解决一个问题，我相信你也会有很多收获。

学习我们教程的时候，要多思考，多想想为什么要这么写？有没有其他更好的办法？然后，自己去验证，去实践。这里非常重要的一点是要多实践，一定要自己动手写代码，然后再下载到开发板验证，不要只是看看视频，看看例程就算完了，要能做到举一反三，如果自己不实践，不动手写代码，是很难真正学会的。

最后，C 语言是学习 STM32 的必备知识，所以如果 C 语言不过关的朋友，得先好好学习一下 C 语言基础，否则学起来会比较吃力。

第二章 STM32 简介

本章，我们将向大家介绍 STM32 是一个什么东西？有哪些资源？能够做什么？如何选型？等基础知识，让大家对 STM32 有一个大概了解。

本章将分为如下几个小节：

2.1 初识 STM32

2.2 STM32F407 资源简介

2.3 STM32F407 设计选型

2.1 初识 STM32

2007 年 6 月，ST 在北京发布了全球第一款基于 ARM Cortex M3 内核的 32 位通用微控制器芯片：STM32F103，以优异的性能，丰富的资源，超高的性价比，迅速占领市场，从此一鸣惊人，一发不可收拾，截止到 2020 年 6 月，STM32 累计出货量超过 45 亿颗。

战舰开发板使用的 STM32F103ZET6 芯片如图 2.1.1 所示：

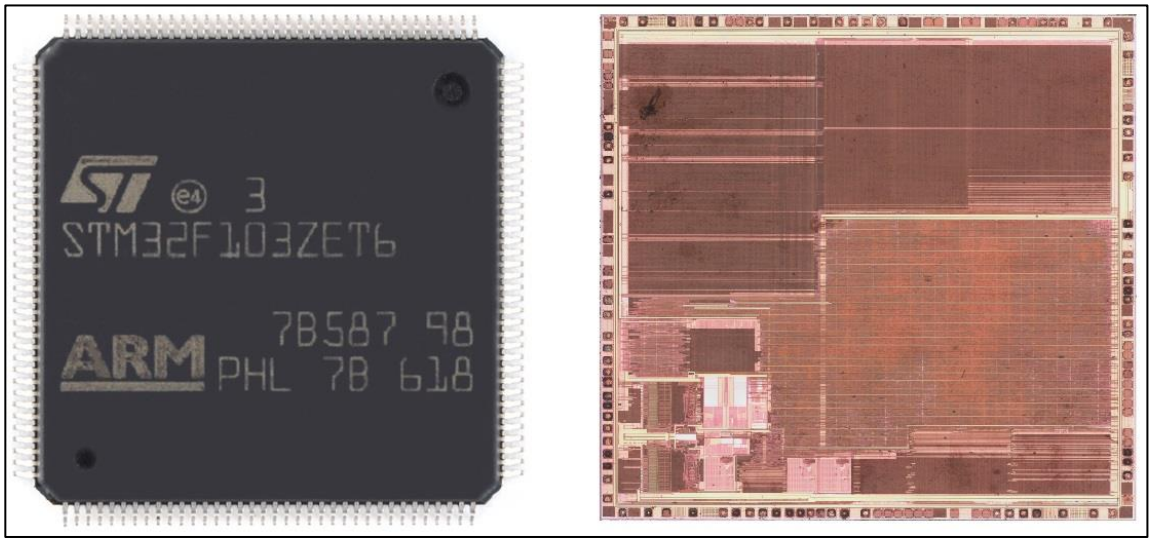


图 2.1.1 STM32F103ZET6 芯片（LQFP144 脚）

上图中，左侧是 STM32F103ZET6 芯片，右侧则是芯片开盖后的图片，即芯片内部视图，可以看到外观看上去平平无奇，但是内部是有很多东西的，需要我们花很多时间和精力去学习掌握。

STM32 的优异性体现在如下几个方面：

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

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

3，丰富的型号。STM32 仅 M3 内核就拥有 F100、F101、F102、F103、F105、F107、F207、F217 等 8 个系列上百种型号，具有 QFN、LQFP、BGA 等封装可供选择。同时 STM32 还推出了 STM32L 和 STM32W 等超低功耗和无线应用型的 M3 芯片，另外，ST 还推出了 STM32F4/F7/H7 等更高性能的芯片。

4，优异的实时性能。150 个中断，16 级可编程优先级，并且所有引脚都可以作中断输入。

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

6，极低的开发成本。通过串口即可下载程序，而且相应的仿真器也很便宜，支持 JTAG&SWD 调试接口，最少仅 2 个 IO 口即可实现仿真调试，极大的降低了开发成本。

再来看一个 STM32 与 51 的性能对比如表 2.1.1 所示：

	内核	主频(Mhz)	DMIPS	硬件 FPU
STM32H7	Cortex M7	480	1027	双精度
STM32F7	Cortex M7	216	462	双精度
STM32F4	Cortex M4	168	210	单精度
STM32F1	Cortex M3	72	90	无
STC15	8051	35	35	无

表 2.1.1 STM32 VS 51

这里我们选的 51 是性能比较好的 STC15 系列做为对比，如果换成传统 51，速度会比 STC15 慢 12 倍左右。最强 H7 的 DMIPS 性能约为 STC15 的 30 倍，即便是 STM32F103 也大概有 STC15 性能的 3 倍，由此可见 STM32 的强大，而且最便宜的 STM32F103，价格大概在 5 块多人民币，和 STC15 系列的价格差不多。

简单来说是，价格差不多的情况下，51 能做的，STM32 都能做，51 不能做的，STM32 也能做，因此，越来越多的企业选择使用 STM32 替代 51，所以，大家如果能学会 STM32，找工作的时候，也会有一定的优势。

2.2 STM32F407 资源简介

下面来看看 STM32F407IGT6 具体的内部资源如表 2.2.1 所示：

STM32F407IGT6 资源					
内核	Cortex M4	基本定时器	2	USART/UART	6
主频	168Mhz	通用定时器	10	CAN	2
FLASH	1024KB	高级定时器	2	SDIO	1
SRAM	192KB	12 位 ADC	3	FSMC	1
封装	LQFP176	ADC 通道数	18	DMA	2
IO 数量	140	12 位 DAC	2	RTC	1
工作电压	3.3V	SPI	3	IIC	3
USB Device	1	USB OTG	1	Ethernet	10/100 Ethernet

表 2.2.1 STM32F407IGT6 内部资源表

由表可知，STM32 内部资源还是非常丰富的，本书将针对这些资源进行详细的使用介绍，并提供丰富的例程，供大家参考学习，相信经过本书的学习，您会对 STM32F407 有一个全面的了解和掌握。

关于 STM32F407 内部资源的详细介绍，请大家参考光盘→A 盘→7，硬件资料→2，芯片资料→ STM32F407IGT6.pdf，该文档即 STM32F407 的数据手册，里面有 STM32F407 详细的资源说明和相关性能参数。

2.3 STM32F407 设计选型

STM32 从 2007 年推出至今，已经有 18 个系列，超过 1000 个型号，为了方便大家选择合适的型号设计产品，本节给大家讲讲 STM32 的设计选型。

2.3.1 STM32 系列

STM32 目前总共有 5 大类，18 个系列，如表 2.3.1.1 所示：

大类	系列	内核	特性
主流级 MCU	G0	Cortex M0+	全新入门级 MCU
	G4	Cortex M4	模数混合型 MCU
	F0	Cortex M0	入门级 MCU
	F1	**Cortex M3**	**基础型 MCU**
	F3	Cortex M4	混合信号 MCU
高性能 MCU	F2	Cortex M4	高性能 MCU
	F4	**Cortex M4**	**高性能 MCU**

	F7	**Cortex M7**	**高性能 MCU**
	H7	**Cortex M7**	**超高性能 MCU**，部分型号有双核（M7+M4）
超低功耗 MCU	L0	Cortex M0+	超低功耗 MCU
	L1	Cortex M3	超低功耗 MCU
	L4	**Cortex M4**	**超低功耗 MCU**
	L4+	Cortex M4	超低功耗高性能 MCU
	L5	Cortex M33	超低功耗高性能安全 MCU
	U5	Cortex M33	超低功耗高性能大容量安全 MCU
无线 MCU	WB	Cortex M4 Cortex M0+	双核无线 MCU
	WL	Cortex M4	远程无线 MCU
微处理器 MPU	**MP1**	**Cortex A7** **Cortex M4**	**双核超高性能 MPU**

表 2.3.1.1 STM32 系列分类及说明

可以看到，STM32 主要分两大块，MCU 和 MPU，MCU 就是我们常见的 STM32 微控制器，不能跑 Linux，而 MPU 则是 ST 在 19 年才推出的微处理器，可以跑 Linux。本书重点介绍 MCU 产品，电机开发板使用的 STM32F407IGT6 属于主流 MCU 分类里面的高性能 F4 系列。

STM32 MCU 提供了包括：基础入门、混合信号、高性能、超低功耗和无线等 5 方面应用的产品型号，我们可以根据自己的实际需要选择合适的 STM32 来设计。比如，我们的产品对性能要求比较高，则可以选择 ST 的高性能 MCU，包括：F2、F4、F7、H7 等 4 个系列的产品；又比如想做超低功耗，则可以选择 ST 的超低功耗 MCU，L 系列的产品。

表 2.3.1.1 中加粗的系列，正点原子都有相应的开发板，包括：主流级 F1、高性能 F4/F7/H7、超低功耗 L4 和微处理器 MP1，大家可以根据自己的需要选择合适的正点原子开发板进行学习。

由于 STM32 系列有很好的兼容性，我们只要能够熟练掌握其中一任何一款 MCU，就可以很方便的学会并使用其他系列的 MCU。

2.3.2 STM32 命名

STM32 的命名规则如图 2.3.2.1 所示：



图 2.3.2.1 STM32 MCU 命名规则（摘自 STM32 产品选型手册）

如图所示，STM32 的产品名字里面包含了：家族、类别、特定功能、引脚数、闪存容量、封装、温度范围等重要信息，这些信息可以帮助我们识别和区分 STM32 不同芯片。

电机开发板使用的 STM32F407IGT6，从命名就可以知道如表 2.3.2.1 所示信息：

描述	型号	说明
家族	STM32	ST 公司 32 位 MCU
产品类别	F	基础型
特定功能	407	STM32 高性能型
引脚数	I	176 脚
闪存容量	G	1024KB FLASH
封装	T	QFP 封装
温度范围	6	-40～+85℃ 工作温度范围（工业级）

表 2.3.2.1 STM32F407IGT6 型号说明

任何 STM32 型号，都可以按图 2.3.2.1 所示命名规则进行区分解读。

2.3.3 STM32 选型

了解了 STM32 的系列和命名以后，我们再进行 STM32 选型就会比较容易了，这里我们只要遵循：**由高到低，由大到小** 的原则，就可以很方便的完成设计选型了，具体如下：

由高到低原则：在不能评估项目所需性能的时候，可以考虑先选择高性能的 STM32 型号进行开发，比如选择 F4/F7/H7 等，在高性能 STM32 上面完成关键性能（即最需要性能的代码）开发验证，如果满足使用要求，则可以降档，如从 H7→F7→F4→F1，如不满足要求，则可以升档，如从 F4→F7→H7，通过此方法找到最佳性价比的 STM32 系列。

由大到小原则：在不能评估项目所需 FLASH 大小、SRAM 大小、GPIO 数量、定时器等资源需求的时候，可以考虑先选择容量较大的型号进行开发，比如选择 512K/甚至 1M FLASH 的型号进行开发，等到开发完成大部分功能之后，就对项目所需资源有了定论，从而可以根据实际情况进行降档选择（当然极少数情况可能还需要升档），通过此方法，找到最合适的 STM32 型号。

整个选型工作大家可以在正点原子开发板上进行验证，一般我们开发板都是选择容量比较大/资源比较多的型号进行设计的，这样可以免去大家自己设计焊接验证板的麻烦，加快项目开发进度。一些资深工程师，对项目要求认识比较深入的话，甚至都不需要验证了，直接就可以选出最合适的型号，这个效率更高。当然这个需要长期积累和多实践，相信只要大家多学习，多实践，总有一天也能达到这个级别。

2.3.4 STM32 设计

这里我们简单给大家介绍一下 STM32 的原理图设计，上一节我们通过选型原则可以确定项目所需的 STM32 具体型号，但是在选择型号以后，需要先设计原理图，然后再画 PCB、打样、焊接、调试等步骤。这里我们重点介绍如何设计 STM32F407IGT6 的原理图。

任何 MCU 部分的原理图设计，其实都遵循：最小系统 + IO 分配 的设计原则。在开始设计原理图之前，我们要通读一遍对 STM32F407IGT6 原理图设计非常有用的手册：STM32F407IGT6 的数据手册，可以说不看这个数据手册，我们就无法设计 STM32F407IGT6 原理图。

1. 数据手册

在设计 STM32F407 原理图的时候，我们需要用到一个非常重要的文档：STM32F407 数据手册，里面对 STM32F407 的 IO 定义和说明有非常详细的描述，是我们设计原理图的基础。电机开发板所使用的 STM32F407IGT6 芯片数据手册，存放在：A 盘→ 7，硬件资料→2，芯片资料→ STM32F407IGT6.pdf，接下来我们简单介绍一下如何使用该文档。

STM32F407IGT6.pdf 是最新的英文版 STM32 数据手册

大家可以根据自己的喜欢来选择合适的版本进行阅读，内容上基本大同小异，从准确性全面性的角度来说，看英文版是最好的。

STM32F407IGT6.pdf 数据手册是针对 STM32F407 系列的数据手册，主要包括 8 个章节，如表 2.3.4.1 所示：

章节	概要说明
介绍	简单说明数据手册作用：介绍大容量增强型 F407xD/E/G 产品的订购信息和机械特性
规格说明	简单介绍 STM32F407 内部所有资源及外设特点
引脚定义	介绍不同封装的引脚分布、引脚定义等，含引脚特性、复用功能、脚位等
存储器映像	介绍 STM32F407 整个 4GB 存储空间和外设的地址映射关系
电气特性	介绍 STM32F407 的详细电气特性，包括工作电压、电流、温度、各外设资源的电气性能等
封装特性	介绍了 STM32F407 不同封装的封装机械数据（脚距、长短等）、热特性等
订货代码	和 2.3.2 节内容类似，介绍 STM32 具体型号所代表的意义，方便选型订货
版本历史	介绍数据手册不同版本之间的差异和修订内容

表 2.3.4.1 STM32F407 数据手册各章节内容概要

整个 STM32F407 数据手册，对我们开发学习 STM32 来说都比较重要，因此建议大家可以简单的通读一遍这个文档，以加深印象。对于原理图设计，最重要的莫过于引脚定义这一章节了，只有知道了 STM32 的引脚定义，才能开始设计原理图。

STM32F407IGT6 引脚分布如图 2.3.4.1 所示：

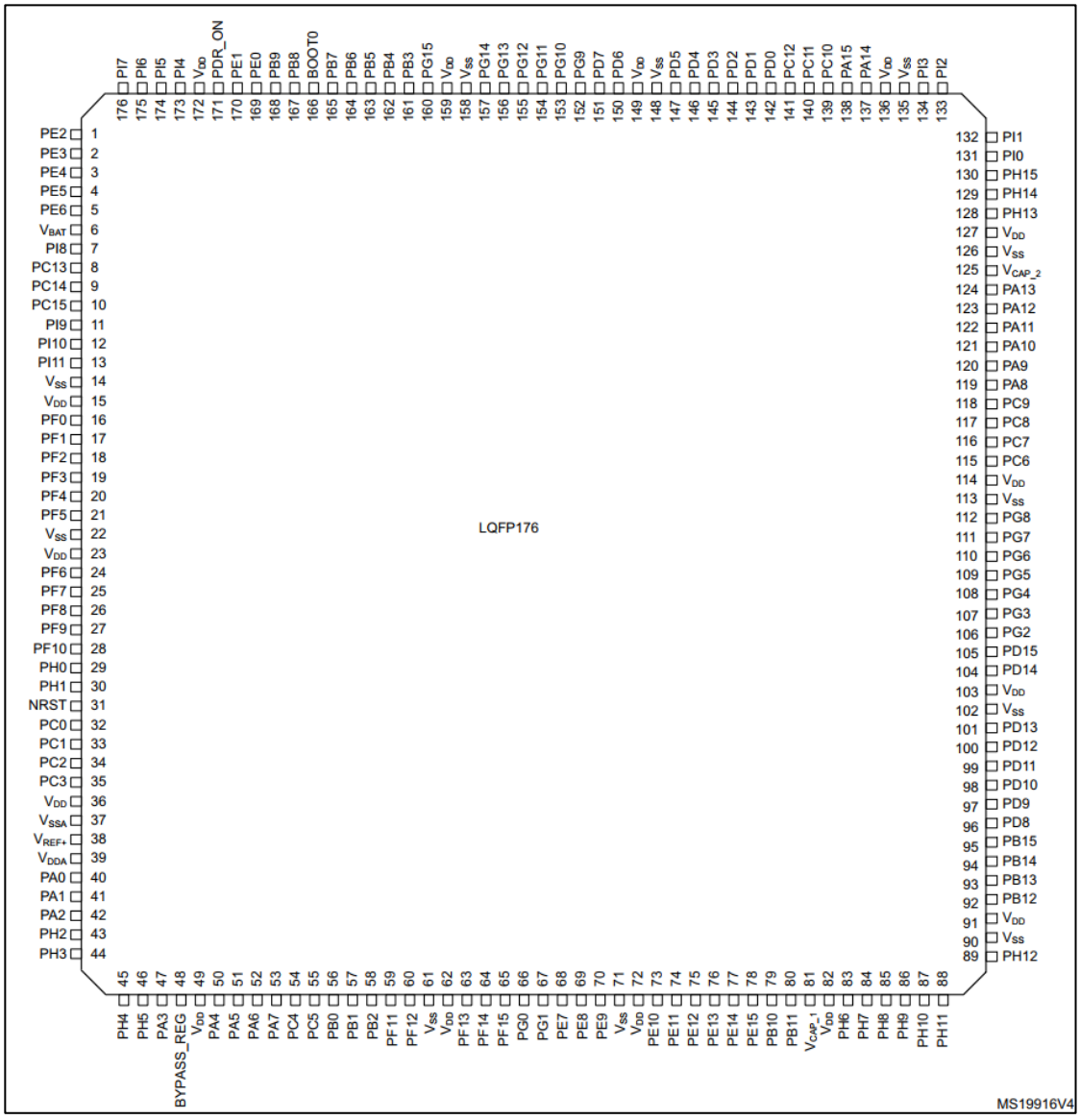


图 2.3.4.1 STM32F407IGT6 芯片引脚分布（摘自 STM32F407 数据手册）

STM32F407IGT6 引脚定义如表 2.3.4.2 所示：

Pin number ①						② Pin name (function after reset)(2)	③ Pin type	④ I / O structure	Notes	⑤ Alternate functions	⑥ Additional functions
LQFP64	WLCSP90	LQFP100	LQFP144	UFBGA176	LQFP176						
-	-	1	1	A2	1	PE2	I/O	FT	-	TRACECLK/ FSMC_A23 / ETH_MII_TXD3 / EVENTOUT	-
-	-	2	2	A1	2	PE3	I/O	FT	-	TRACED0/FSMC_A19 / EVENTOUT	-
-	-	3	3	B1	3	PE4	I/O	FT	-	TRACED1/FSMC_A20 / DCMI_D4/ EVENTOUT	-
-	-	4	4	B2	4	PE5	I/O	FT	-	TRACED2 / FSMC_A21 / TIM9_CH1 / DCMI_D6 / EVENTOUT	-

表 2.3.4.2 STM32F407IGT6 芯片引脚定义（部分）（摘自 STM32F407 数据手册）

引脚定义表的具体说明如表 2.3.4.3 所示：

序号	名称	说明
①	脚位	对应芯片的引脚，LQFP 使用纯数字表示，BGA 使用字母+数字表示 这里列出了 5 种封装的脚位描述，根据实际型号选择合适的封装查阅
②	管脚名称	即对应引脚的名字，PE2~3 表示 GPIO 引脚
③	类型	I/O ：表示输入/输出引脚 I ：表示输入引脚 S ：表示电源引脚
④	IO 结构	FT ：表示 5V 兼容的引脚（可以接 5V/3.3V） TT ：表示 5V 不兼容引脚（仅可以接 3.3V） B ：独有的 BOOT0 引脚 RST：内置弱电阻的双向复位引脚
⑤	复用功能	该引脚的复用功能
⑥	可选的复用功能	重定义功能：是指可以通过重映射的复用功能，需设置重映射寄存器

表 2.3.4.3 引脚定义说明

了解了引脚分布和引脚定义以后，我们就可以开始设计 STM32F407 的原理图了。

2. 最小系统

最小系统就是保证 MCU 正常运行的最低要求，一般是指 MCU 的供电、复位、晶振、BOOT 等部分。STM32F407 的最小系统需求如表 2.3.4.4 所示：

类型	引脚名称	说明
电源	VDD/VSS	电源正（VDD）/负（VSS）引脚，给 STM32 供电
	VDDA/VSSA	模拟部分电源正/负引脚，给 STM32 内部模拟部分供电
	VREF+/VREF-	参考电压正/负引脚，给 STM32 内部 ADC/DAC 提供参考电压 100 脚及以上的 STM32F407 型号才有这两个脚
	VBAT	RTC&后备区域供电引脚，给 RTC 和后备区域供电。一般 VBAT 接电池，用于断电维持 RTC 工作，如不需要，直接将 VBAT 接 VDD 即可

复位	NRST	复位引脚，用于复位 STM32，低电平复位
启动	BOOT0/BOOT1	启动选择引脚，一般这两个脚各接一个下拉电阻即可 其他启动配置说明详见后续分析
晶振	OSC_IN / OSC_OUT	外部 HSE 晶振引脚，用于给 STM32 提供高精度系统时钟 如果使用内部 HSI 能满足使用需求，这两个脚可以不接晶振
	OSC32_IN / OSC32_OUT	外部 LSE 晶振引脚，用于给 STM32 内部 RTC 提供时钟 如果使用内部 LSI 能满足使用需求，这两个脚可以不接晶振
调试	SWCLK/SWDIO	SWD 调试引脚，用于调试 STM32 程序，同时 STM32 还支持 JTAG 调试，不过我们不推荐使用！因为 SWD 省 IO！

表 2.3.4.4 STM32F407 最小系统需求

完成以上引脚的设计以后，STM32F407 的最小系统就完成了，关于这些引脚的实际原理图，大家可以参考我们电机开发板的原理图。接下来就可以开始进行 IO 分配了。

3. IO 分配

IO 分配就是在完成最小系统设计以后，根据项目需要对 MCU 的 IO 口进行分配，连接不同的器件，从而实现整体功能。比如：GPIO、IIC、SPI、SDIO、FSMC、USB、中断等。遵循：先分配特定外设 IO，再分配通用 IO，最后微调的原则，见表 2.3.4.5 所示：

分配	外设	说明
特定外设	IIC	IIC 一般用到 2 根线：IIC_SCL 和 IIC_SDA（ST 叫 I2C） 数据手册有 I2C_SCL、I2C_SDA 复用功能的 GPIO 都可选用
	SPI	SPI 用到 4 根线：SPI_CS/MOSI/MISO/SCK 一般 SPI_CS 我们使用通用 GPIO 即可，方便挂多个 SPI 器件 数据手册有 SPI_MOSI/MISO/SCK 复用功能的 GPIO 都可选用
	TIM	根据需要可选：TIM_CH1/2/3/4/ETR/1N/2N/3N/BKIN 等 数据手册有 TIM_CH1/2/3/4/ETR/1N/2N/3N/BKIN 复用功能的 GPIO 都可选用
	USART UART	USART 有 USART_TX/RX/CTS/RTS/CK 信号 UART 仅有 UART_TX/RX 两个信号 一般用到 2 根线：U(S)ART_TX 和 U(S)ART_RX 数据手册有 U(S)ART_TX/RX 复用功能的 GPIO 都可选用
	USB	USB 用到 2 根线：USB_DP 和 USB_DM 数据手册有 USB_DP、USB_DM 复用功能的 GPIO 都可选用
	CAN	CAN 用到 2 根线：CAN_RX 和 CAN_TX 数据手册有 USB_DP、USB_DM 复用功能的 GPIO 都可选用
	ADC	ADC 根据需要可选：ADC_IN0～ADC_IN15 数据手册有 ADC_IN0～ADC_IN15 复用功能的 GPIO 都可选用
	DAC	DAC 根据需要可选：DAC_OUT1 / DAC_OUT2 DAC 固定为：DAC_OUT1 使用 PA4、DAC_OUT2 使用 PA5
	SDIO	SDIO 一般用到 6 根线：SDIO_D0/1/2/3/SCK/CMD 数据手册有 SDIO_D0/1/2/3/SCK/CMD 复用功能的 GPIO 都可选用
	FSMC	根据需要可选：FSMC_D0~15/A0~25/ NBL0~1/NE1~4/NCE2~3/ NOE/NWE/NWAIT/CLK 等 数据手册有 FSMC_D0~15/A0~25/ NBL0~1/NE1~4/NCE2~3/NOE/ NWE/NWAIT/CLK 复用功能的 GPIO 都可选用
通用	GPIO	在完成特定外设的 IO 分配以后，就可以进行 GPIO 分配了 比如将按键、LED、蜂鸣器等仅需要高低电平读取/输出的外设连接到空闲的普通 GPIO 即可
微调	IO	微调主要包括两部分： 1，当 IO 不够用的时候，通用 GPIO 和特定外设可能要共用 IO 口

		2，为了方便布线，可能要调整某些 IO 口的位置 这两点，得根据实际情况进行调整设置，做到：尽可能多的可以同时使用所有功能，尽可能方便布线。

表 2.3.4.5 IO 分配

经过以上几个步骤，我们就可以完成 STM32F407 的原理图设计了。

第三章 开发环境搭建

本章，我们将向大家介绍 STM32 的开发环境搭建，通过本章的学习，我们将了解到有哪些常用的 STM32 开发工具，包括 IDE、调试器、串口工具等。

本章将分为如下几个小节：

3.1 常用开发工具简介
3.2 MDK5 安装
3.3 仿真器驱动安装
3.4 CH340 USB 虚拟串口驱动安装

3.1 常用开发工具简介

我们开发 STM32 需要用到一些开发工具，如：IDE、仿真器、串口调试助手等。常见的工具如表 3.1.1 所示：

工具	名称	说明
集成开发环境	**MDK**	全名：RealView MDK，是 Keil 公司（已被 ARM 收购）的一款集成开发环境，界面美观，简单易用，是 STM32 最常用的集成开发环境
	EWARM	IAR 公司的一款集成开发环境，支持 STM32 开发，对比 MDK，IAR 的使用人数少一些，用惯 IAR 的朋友可以选择这款软件开发 STM32
仿真器	**DAP**	ARM 公司的开源仿真器，可支持 STM32 仿真调试，且带虚拟串口功能。有高速和低速两个版本，具有免驱、速度快、价格便宜等特点
	STLINK	ST 公司自家的仿真器，支持 STM32 和 STM8 仿真调试，目前最常用的是 ST LINK V2，支持全面、稳定、廉价，是其特点
	JLINK	Segger 公司的仿真器，可支持 STM32 仿真调试，具有稳定、高速的特点，就是价格有点贵。
串口调试助手	**XCOM**	正点原子开发的串口调试助手，具有稳定、功能多、使用简单等特点
	SSCOM	丁丁的串口调试助手，具有稳定、小巧、使用简单等特点。

表 3.1.1 常用开发工具

大家可以根据自己的需要和喜好，选择合适的开发工具。表中加粗部分是我们推荐使用的 STM32 开发工具，即 IDE 推荐使用 MDK、仿真器推荐使用 DAP、串口调试助手推荐使用 XCOM，接下来我们介绍这几个软件的安装。

3.2 MDK 安装

注意：MDK 是一款付费集成开发环境，如果大家要商用，请联系 Keil 公司购买，我们这里仅用于教学使用。

MDK5 的安装分为两步：1，安装 MDK5；2，安装器件支持包。

MDK 软件下载地址：https://www.keil.com/download/product，目前最新版本是 MDK5.34。

器件支持包下载地址：https://www.keil.com/dd2/pack，STM32F4 支持包最新版本是 2.8.0。

MDK5.34 和 2.8.0 的 STM32F4 器件支持包我们都已经放在光盘 A 盘了，具体路径为：A 盘→6，软件资料→1，软件→1，MDK，如图 3.2.1 所示：

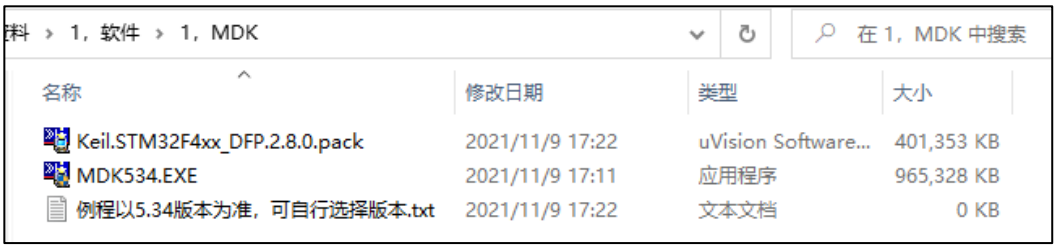


图 3.2.1 MDK5 软件及 STM32F4 器件支持包

MDK5 的安装比较简单，具体安装步骤请参考图 3.2.1 的 **安装过程.txt** 进行安装即可，需要提醒一下大家，在选择安装路径的时候，强烈建议大家将 Pack 的路径和 Core 的路径放在一个位置，比如我们安装在 D 盘（都安装在：D:\MDK5.34 路径下），如图 3.2.2 所示：

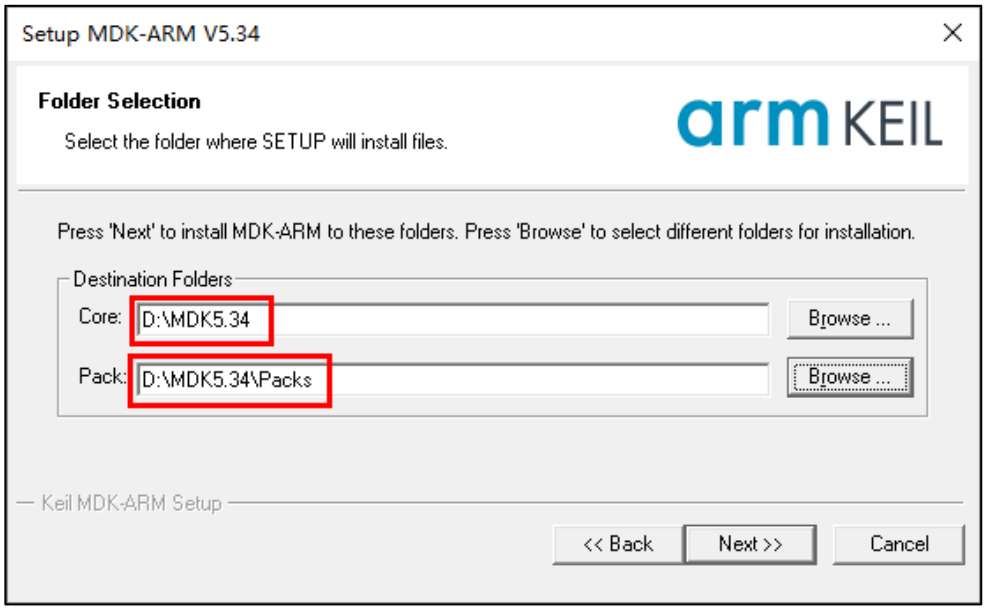


图 3.2.2 设置 Core 和 Pack 安装路径

安装完成后，在我们电脑桌面会显示 MDK5 图标，如图 3.2.3 所示；



图 3.2.3 桌面显示 MDK5 图标

3.3 仿真器驱动安装

STM32 可以通过 DAP、ST LINK、JLINK 等仿真调试器进行程序下载和仿真，我们默认推荐使用：DAP 仿真器（CMSIS-DAP Debugger），DAP 仿真器在 MDK 下是免驱动的（无需安装驱动），即插即用，非常方便。

正点原子提供两种规格的 DAP 仿真器：普速版本 DAP（ATK-DAP）和高速版本（ATK-HSDAP），这两个版本 DAP 使用完全一样，只是高速版本速度更快，大家根据需要选择即可。

如果你用的是 STLINK 仿真器，大家可以参考 A 盘→ 6，软件资料→1，软件→5，其他小工具→ST LINK 驱动及教程→ST LINK 调试补充教程.pdf，进行驱动安装。

3.4 CH340 USB 虚拟串口驱动安装

安装 CH340 USB 虚拟串口驱动，以便我们使用电脑通过 USB 和 STM32 进行串口通信。开发板使用的 USB 虚拟串口芯片是 CH340C，其驱动我们已经放在开发板 A 盘→6，软件资料→1，软件→5，其他小工具→ CH340 驱动(USB 串口驱动) 文件夹里面，如图 3.4.1 所示：

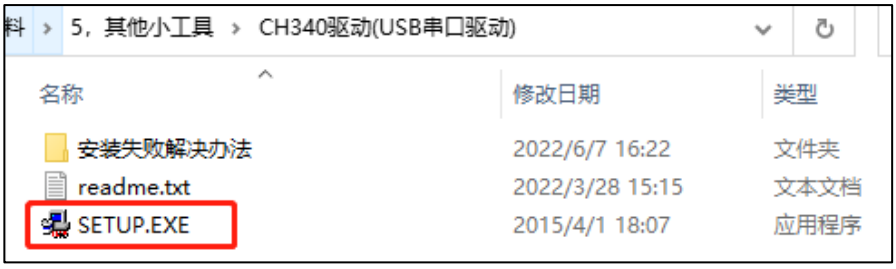


图 3.4.1 CH340 驱动

双击 SETUP.EXE 进行安装，安装完成后，如图 3.4.2 所示：

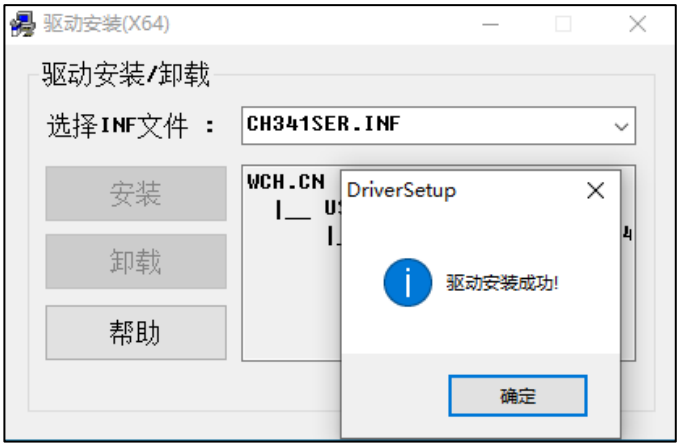


图 3.4.2 CH340 驱动安装成功

在驱动安装成功之后，将开发板的 USB_UART 接口通过 USB 连接到电脑，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 3.4.3 所示：

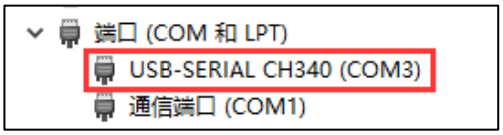


图 3.4.3 设备管理器显示 CH340 USB 虚拟串口

在图 3.4.3 中可以看到，我们的 USB 虚拟串口被识别为 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 CH340 USB 虚拟串口以后，我们就可以使用串口调试助手，比如 XCOM，和我们的开发板，通过串口进行通信了，这个我们在后续内容再给大家介绍。至此，STM32 的开发环境就搭建完成了。

第四章 STM32 初体验

本章，我们不介绍如何编写代码，而是向大家介绍如何编译、串口下载、仿真器下载、仿真调试开发板例程，体验一下 STM32 的开发流程，并介绍 MDK5 的一些使用技巧，通过本章的学习，将对 STM32 的开发流程和 MDK5 使用有个大概了解，为后续深入学习打好基础。

本章将分为如下几个小节：

4.1 使用 MDK5 编译例程

4.2 使用 DAP 下载与调试程序

4.3 MDK5 使用技巧

4.1 使用 MDK5 编译例程

我们在编写完代码以后，需要对代码进行编译，编译成功以后，才能下载到开发板进行验证、调试等操作。电机开发板标准例程源码路径：A 盘→1，电机基础代码，如图 4.1.1 所示：

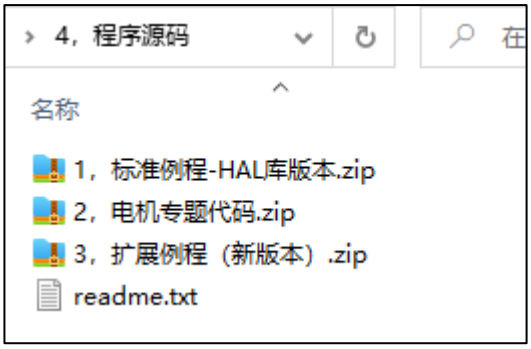


图 4.1.1 电机开发板例程源码

这 3 个压缩包说明如表 4.1.1：

压缩包	说明
标准例程-HAL 库版本	HAL 库版本标准例程，搭配 DMF407 开发指南.pdf 学习
电机专题代码	各类电机使用 DMF407 开发板的驱动代码，搭配 DMF407 电机控制专题教程开发指南.pdf 学习
扩展例程	包含 ucOS、FreeRTOS、emWIN、LittlevGL、lwIP 例程搭配对应的开发指南学习

表 4.1.1 源码压缩包说明

本书主要针对“标准例程-HAL 库版本”源码进行讲解，所以，这里我们先解压该压缩包，得到 HAL 库版本的标准例程源码如图 4.1.2 所示：

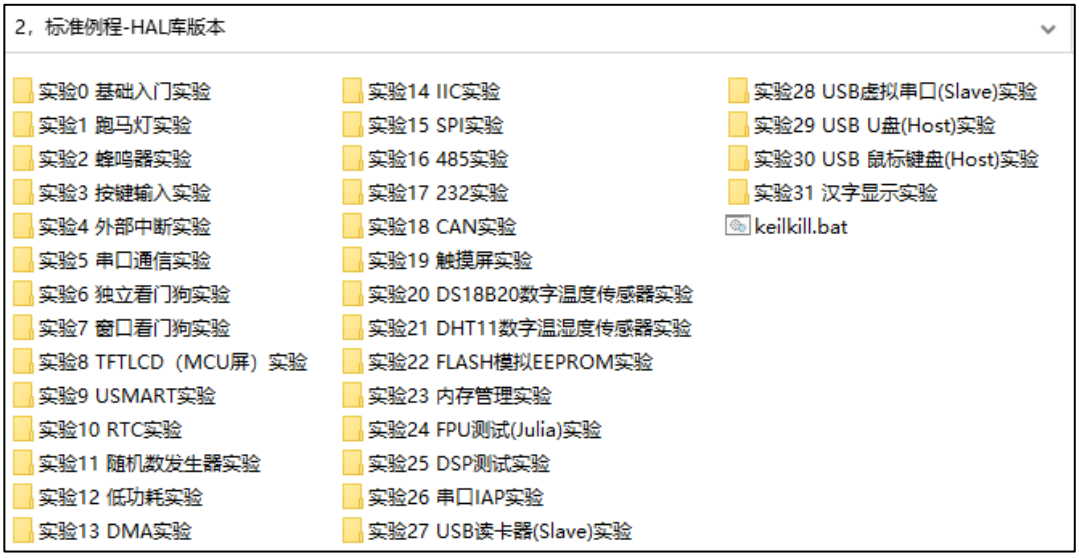


图 4.1.2 标准例程-HAL 库版本

电机基础例程总共听过 31 个实验的标准例程，部分实验有多个例程，比如低功耗、FPU、DSP 等实验。这里我们没有把实验 0 统计进来，因为实验 0 主要是新建工程等 MDK 使用基础教学例程。

秉着简单易懂的原则，我们选择：实验 1，跑马灯实验 作为 STM32 入门体验，例程目录如图 4.1.3 所示：



图 4.1.3 跑马灯例程工程目录结构

工程目录下有 5 个文件夹，我们将在后续“新建 HAL 库版本 MDK 工程”这一章给大家详细介绍其作用（其实看文件夹名字的字面意思基本就知道是做啥用的了），这里我们不多说。例程 MDK 工程文件的路径为：Projects→MDK-ARM→atk_f407.uvprojx，如图 4.1.4 所示：

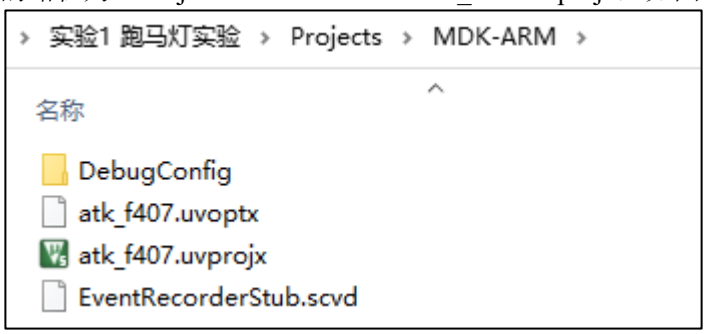


图 4.1.4 跑马灯实验例程 MDK 工程

注意：一定要先安装 MDK5（详见第三章），否则无法打开该工程文件！

双击 atk_f407.uvprojx 打开该工程文件，进入 MDK IDE 界面，如图 4.1.5 所示：

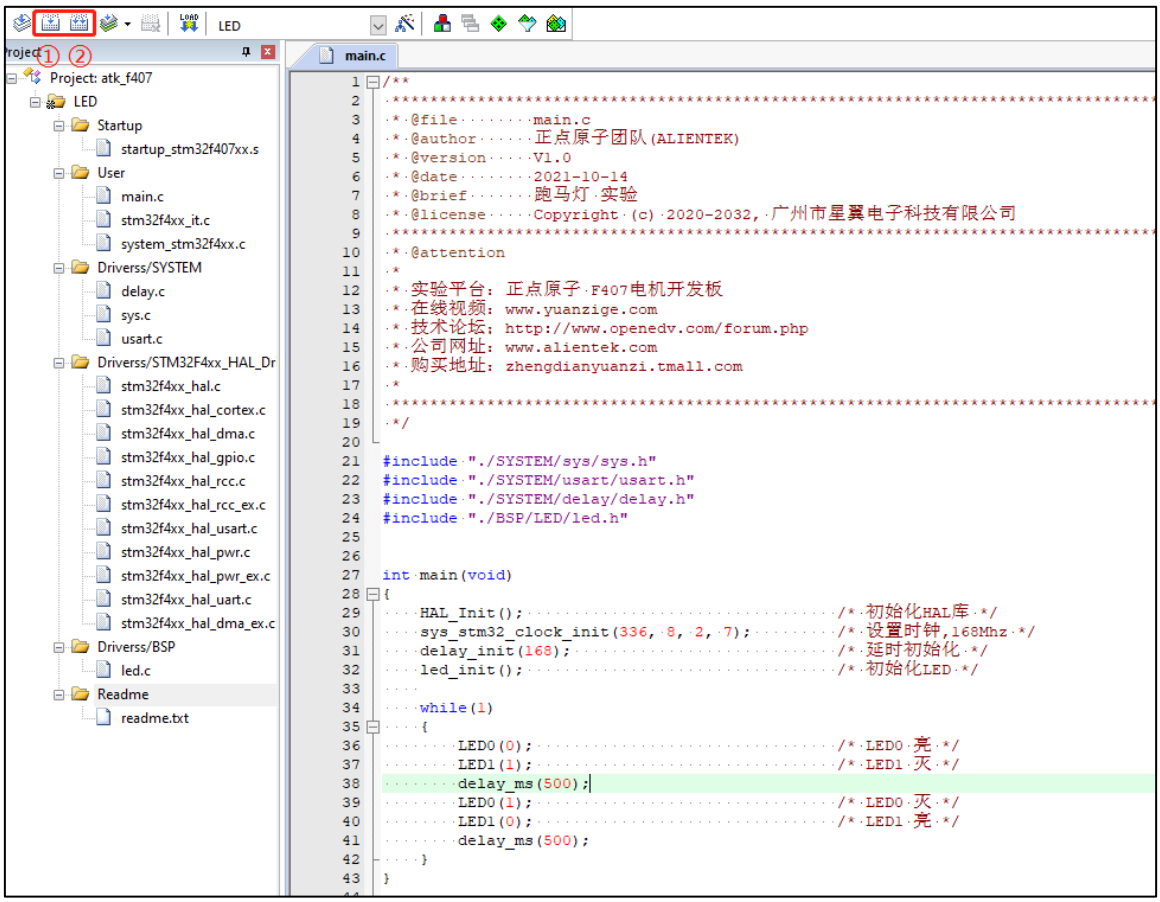


图 4.1.5 MDK 打开跑马灯实验例程

① 是编译按钮，表示编译当前工程项目文件，如果之前已经编译过了，则只会编译有改动的文件。所以一般第一次会比较耗时间，后续因为只编译改动文件，从而大大缩短了编译时间。该按钮可以通过 F7 快捷键进行操作。

② 是重新编译当前工程所有文件按钮，工程代码较多时全部重新编译会耗费比较多的时间，建议少用。

按①处的按钮，编译当前项目，在编译完成后，可以看到如图 4.1.6 所示的编译提示信息：

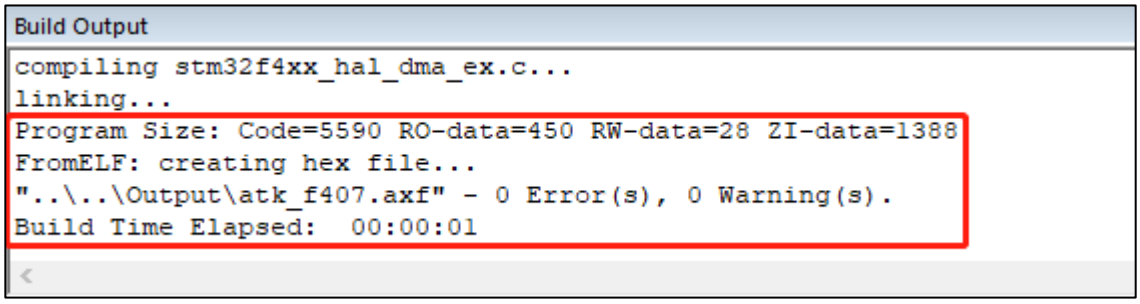


图 4.1.6 编译提示信息

图中：

Code：表示代码大小，占用 5442 字节。

RO-Data：表示只读数据所占的空间大小，一般是指 const 修饰的数据大小。

RW-Data：表示有初值（且非 0）的可读写数据所占的空间大小，它同时占用 FLASH（存放其初始值）和 RAM 空间。

ZI-Data：表示初始化为 0 的可读写数据所占空间大小，它只占用 RAM 空间。

因此图 4.1.6 的提示信息表示：代码总大小（Program Size）为：FLASH 占用 6068 字节（Code + RO + RW），SRAM 占用 1416 字节（RW + ZI）；成功创建了 Hex 文件（可执行文件，放在 Output 目录下）；编译 0 错误，0 警告；编译耗时 1 秒钟。

编译完成以后，会生成 Hex 可执行文件，默认输出在 Output 文件夹下，如图 4.1.7 所示：

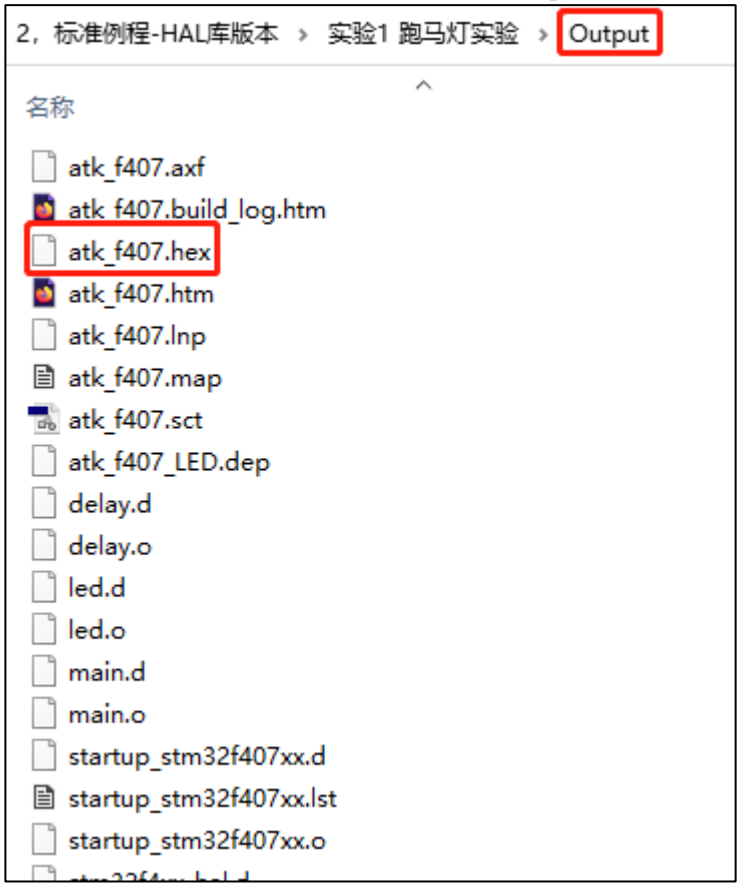


图 4.1.7 Hex 可执行文件

注意：必须编译成功，才会生成 Hex 可执行文件，否则是不会有这个文件的 !!

Output 文件夹下除了.hex 文件还有很多其他文件（.axf、.htm、.dep、.lnp、.o、.d、.lst 等），这些文件是编译过程所产生的中间文件，我们将在后续的.map 文件分析给大家详细介绍这些文件的作用。至此，例程编译完成。其他实验例程，大家可以用同样的方法进行编译。

4.2 使用 DAP 下载与调试程序

STM32 常见的下载方式有三种：

下载方式	使用接口	下载条件	优缺点
串口下载	串口 1 PA9/PA10	1，BOOT0 接 3.3V 2，BOOT1 接 GND 3，按复位	优点：仅需一个串口即可通过上位机软件下载程序，经济实惠 缺点：速度慢
SWD 下载	SWD 口 PA13/PA14	直接下载 特殊情况需 BOOT0 接 3.3V，按复位	优点：2 个 IO 口就可以下载、仿真、调试代码，高速、高效、实用 缺点：需要一个仿真器
JTAG 下载	JTAG 口 PB3/4/PA13/14/15	直接下载 特殊情况需 BOOT0 接 3.3V，按复位	优点：5 个 IO 口实现下载、仿真、调试代码，实用方便 缺点：需要一个仿真器，占用 IO 多

表 4.2.1 STM32 常见下载方式对比

经过以上对比，**SWD 下载**方式有明显优势，强烈推荐大家购买一个仿真器（如 ST LINK、CMSIS DAP 等），可以极大的方便学习和开发。

表中 BOOT0 和 BOOT1 是 STM32 芯片上面的两个引脚，用于控制 STM32 的启动方式，具体如表 4.2.2 所示：

BOOT0	BOOT1	启动模式	说明

0	X	用户闪存存储器启动 地址：0X0800 0000	用户闪存储器存启动，也就是 FLASH 启动 正常运行我们自己的程序，就应该用此设置
1	0	系统存储器启动 地址：0X1FFF F000	系统存储器启动，用于串口下载程序
1	1	SRAM 启动（不常用） 地址：0X2000 0000	SRAM 启动，一般没用

表 4.2.2 STM32 启动模式

由于从系统存储器启动，是不运行用户程序的，所以不管用户程序怎么写，此模式下都可以通过仿真器下载代码。所以，该模式可常用于异常关闭 JTAG/SWD 导致仿真器无法下载程序时的补救措施（在此模式下，下载一个不关闭 JTAG/SWD 接口的程序即可救活）。

电机开发板由于没有引出 B0 和 B1，所以不支持串口下载只能使用仿真器下载。

这里我们以 DAP 仿真器为例给大家进行讲解，如果你用的是其他仿真器，基本上也是一样的用法，只是选择仿真器的时候，选择对应的型号即可。

DAP 与开发板连接如图 4.2.1 所示：

a) DAP 通过 USB 线连接电脑，且仿真器的蓝灯常亮。
b) DAP 通过 4P 排线连接开发板①处。
c) 确保开发板已经正常供电(可经由图中的②③给开发板供电，但我们建议使用③处的电源插座+开发板适配的 DC 直流电源，以避免部分需要用到大电流的例程运行不正常)，供电后确保④处的开关是按下的状态，这里蓝色电源灯亮起表示开发板已经正常供电。
d) B0，B1 未引出，均接的 GND。

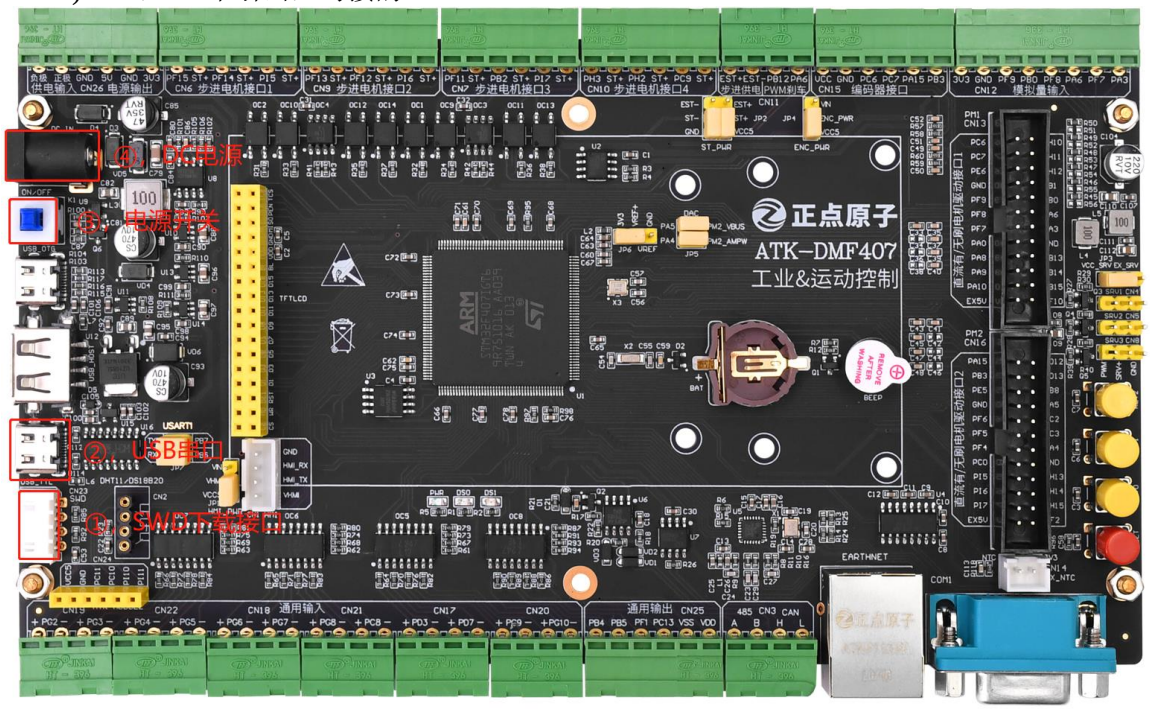


图 4.2.1 DAP 与开发板连接

4.2.1 使用 DAP 下载程序

在 4.1 节的跑马灯例程 MDK IDE 界面下，点击按钮，打开 Options for Target 选项卡，在 Debug 栏选择仿真工具为 use：CMSIS-DAP Debugger，如图 4.2.1.1 所示：

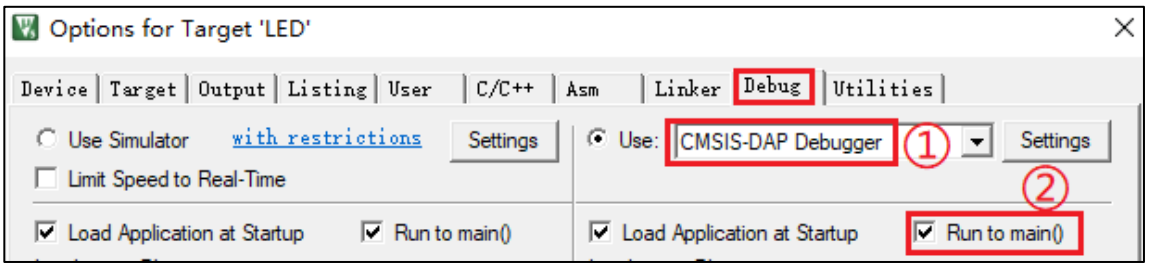


图 4.2.1.1 Debug 选项卡设置

① 选择使用 CMSIS-DAP Debugger 仿真器仿真调试代码。如果你使用的是其他仿真器，比如 STLINK、JLINK 等，请在这里选择对应的仿真器型号。

② 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_stm32f407xx.s 文件的 Reset_Handler，再跳到 main 函数。

然后我们点击 Settings，设置 DAP 的一些参数，如图 4.3.1.2 所示：

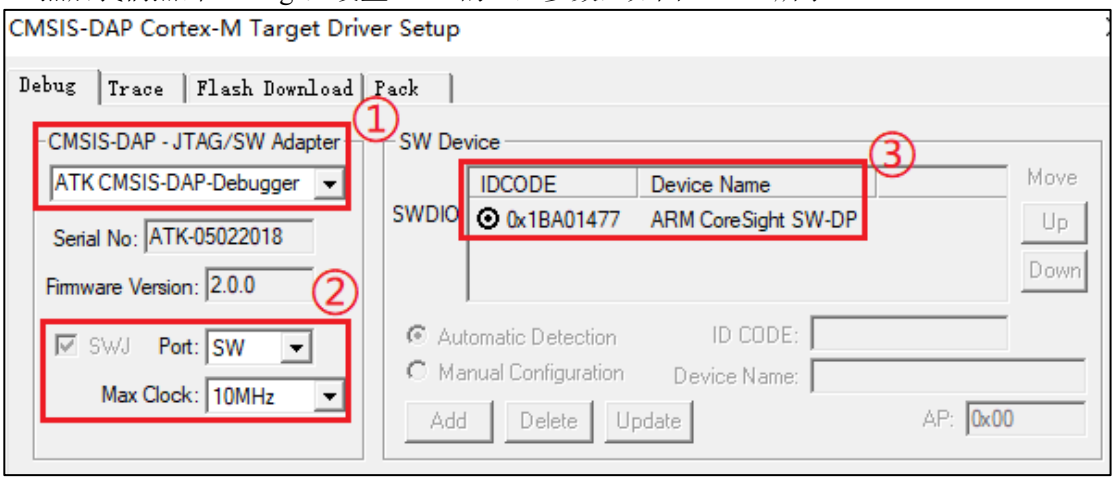


图 4.2.1.2 DAP 仿真器参数设置

① 表示 MDK 找到了 ATK CMSIS-DAP 仿真器，如果这里显示为空，则表示没有仿真器被找到，请检查你的电脑是否接了仿真器？并安装了对应的驱动？

② 设置接口方式，这里选择 SW（比 JTAG 省 IO），通信速度设置为 10Mhz（实际上大概只有 4M 的速度，MDK 会自动匹配）。

③ 表示 MDK 通过仿真器的 SW 接口找到了目标芯片，ID 为：0x1BA01477。如果这里显示：No target connected，则表示没找到任何器件，请检查仿真器和开发板连接是否正常？开发板是否供电了？

其他部分使用默认设置，设置完成以后单击“确定”按钮，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 4.2.1.3 所示：

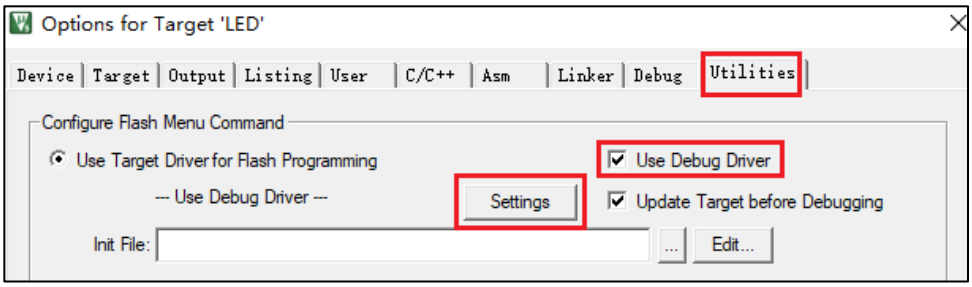


图 4.2.1.3 FLASH 编程器选择

上图中，我们直接勾选 Use Debug Driver，即和调试一样，选择 DAP 来给目标器件的 FLASH 编程，然后点击 Settings，进入 FLASH 算法设置，设置如图 4.2.1.4 所示：

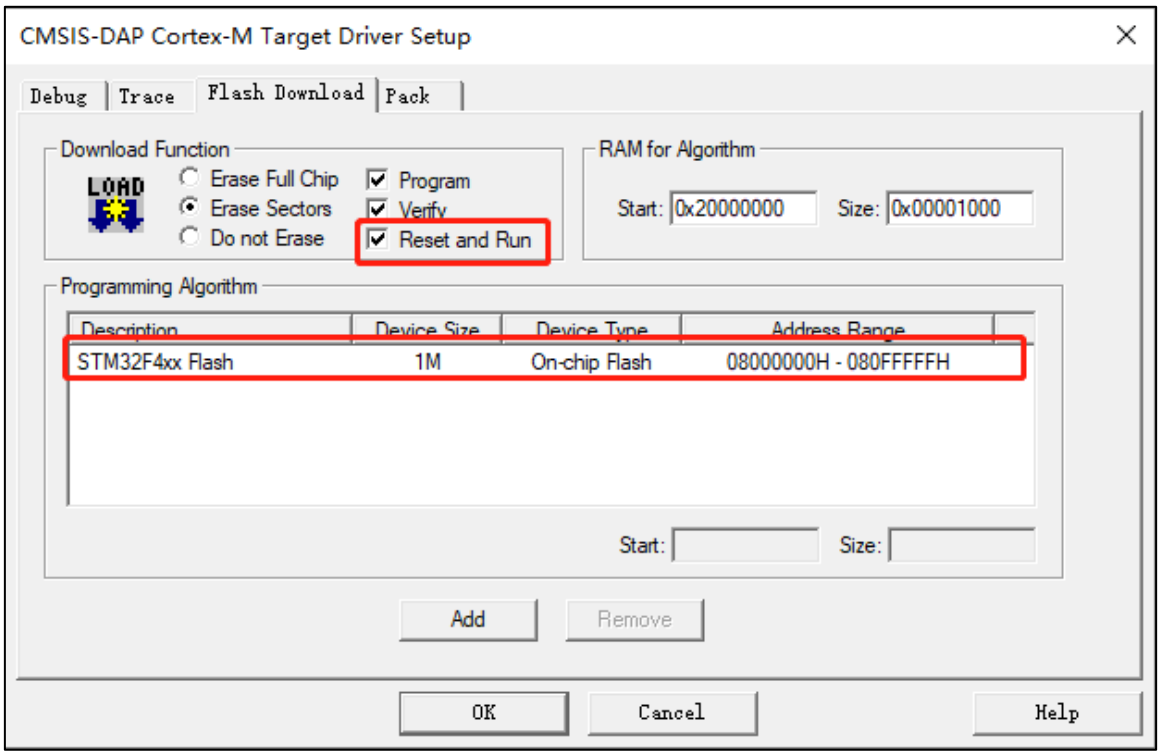


图 4.2.1.4 FLASH 算法设置

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 STM32F407IGT6，FLASH 容量为 1M 字节，所以 Programming Algorithm 里面默认会有 1M 型号的 STM32F4xx Flash 算法。另外，如果这里没有 flash 算法，大家可以点击 Add 按钮，自行添加即可。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。

在设置完之后，点击“确定”，然后再点击“OK”，回到 IDE 界面，编译（可按 F7 快捷键）一下工程，编译完成以后（0 错误，0 警告），我们按 （快捷键：F8）这个按钮，就可以将代码通过仿真器下载到开发板上，在 IDE 下方的 Build Output 窗口会提示相关信息，如下图所示：

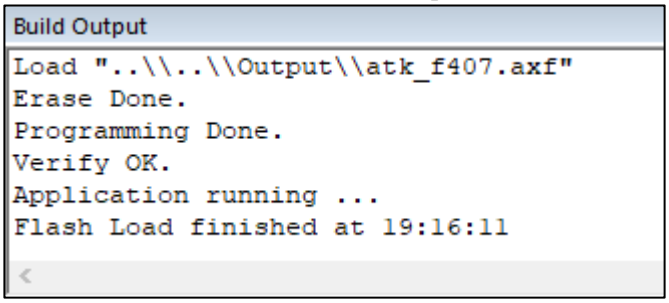


图 4.2.1.5 仿真器下载代码

下载完后，就可以看到 DS0 和 DS1 交替闪烁了，说明代码下载成功。

4.2.2 使用 DAP 仿真调试程序

在正常编译完例程以后（0 错误，0 警告），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），如图 4.2.2.1 所示：

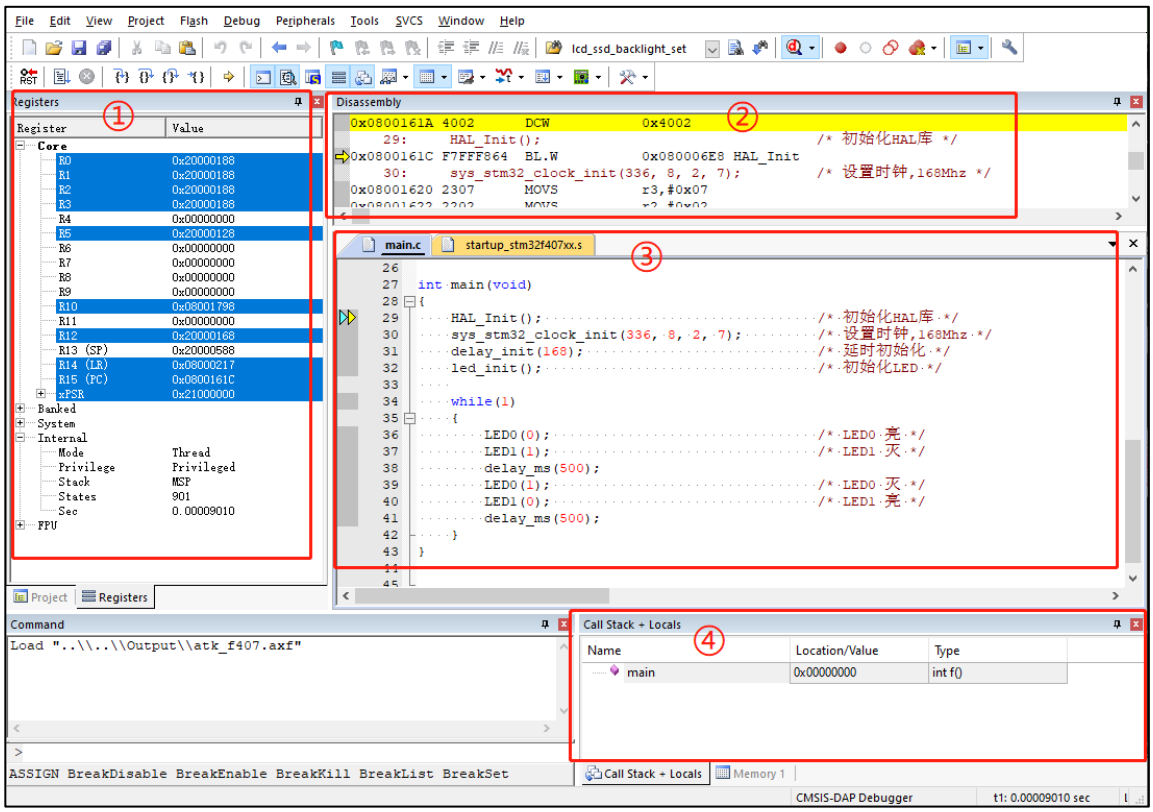


图 4.2.2.1 开始仿真

① Register：寄存器窗口，显示了 Cortex M4 内核寄存器 R0~R15 的值，还显示了内部的线程模式（处理者模式、线程模式）及特权级别（用户级、特权级），并且还显示了当前程序的运行时间（Sec），该选项卡一般用于查看程序运行时间，或者比较高级的 bug 查找（涉及到分析 R0~R14 数据是否异常了）。

② Disassembly：反汇编窗口，将 C 语言代码和汇编对比显示（指令存放地址，指令代码，指令，具体操作），方便从汇编级别查看程序运行状态，同样也属于比较高级别的 bug 查找。

③ 代码窗口，在左侧有黄绿色三角形，黄色的三角形表示将要执行的代码，绿色的三角形表示当前光标所在代码（C 代码 或 当前汇编行代码对应的 C 代码）。一般情况下，这两个三角形是同步的，只有在点击光标查看代码的时候，才可能不同步。

④ Call Stack + Locals：调用关系&局部变量窗口，通过该窗口可以查看函数调用关系，以及函数的局部变量，在仿真调试的时候，是非常有用的。

开始仿真的默认窗口，我们就给大家介绍这几个，实际上还有一些其他的窗口，比如 Watch、Memory、外设寄存器等也是很常用的，可以根据实际使用选择调用合适的窗口来查看对应的数据。

图 4.2.2.1 中，还有一个很重要的工具条：Debug 工具条，其内容和作用如图 4.2.2.2 所示：

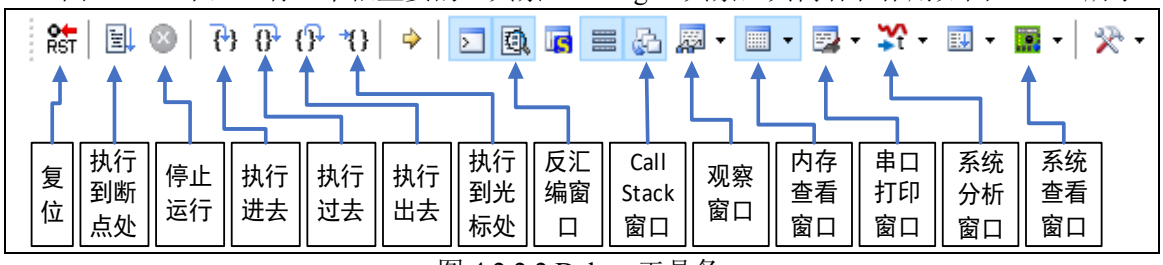


图 4.2.2.2 Debug 工具条

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，有时候可能不必再执行该函数的剩余部分了，通过该按钮就可以一步执行完该函数的余部分，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。

反汇编窗口：通过该按钮，就可以查看汇编代码，这对分析程序很有用。

Call STACK 窗口：通过该按钮，显示调用关系&局部变量窗口，显示当前函数的调用关系和局部变量，方便查看，对分析程序非常有用。

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。

系统分析窗口：该图标下面有 6 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。

我们在图 4.2.2.1 的基础上：关闭反汇编窗口（Disassembly）、添加观察窗口 1（Watch1）。然后调节一下窗口位置，然后将全局变量：g_fac_us （在 delay.c 里面定义）加入 Watch1 窗口（方法：双击 Enter expression 添加/直接拖动变量 t 到 Watch1 窗口即可），如图 4.2.2.3 所示：

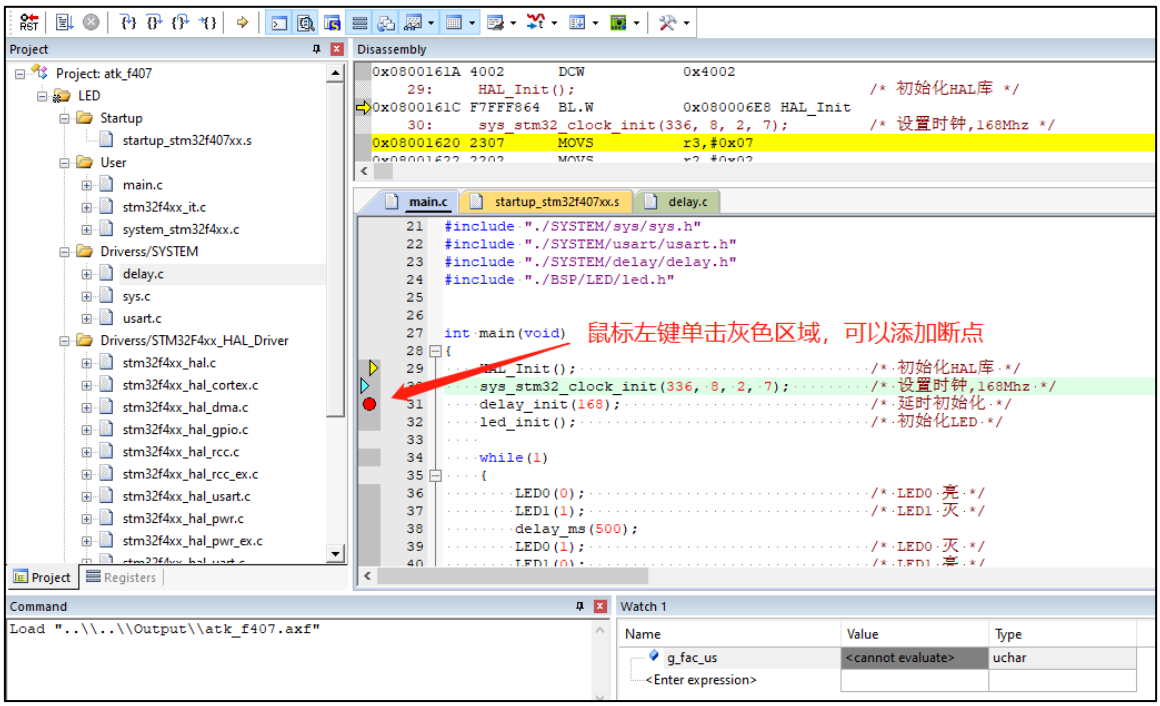


图 4.2.2.3 开始仿真

此时可以看到 Watch1 窗口的 g_fac_us 的值提示：无法计算，我们把鼠标光标放在第 32 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），这样就在 delay_init 函数处放置一个断点，然后点击：，执行到该断点处，然后再点击：，执行进入 delay_init 函数，如图 4.2.2.4 所示：

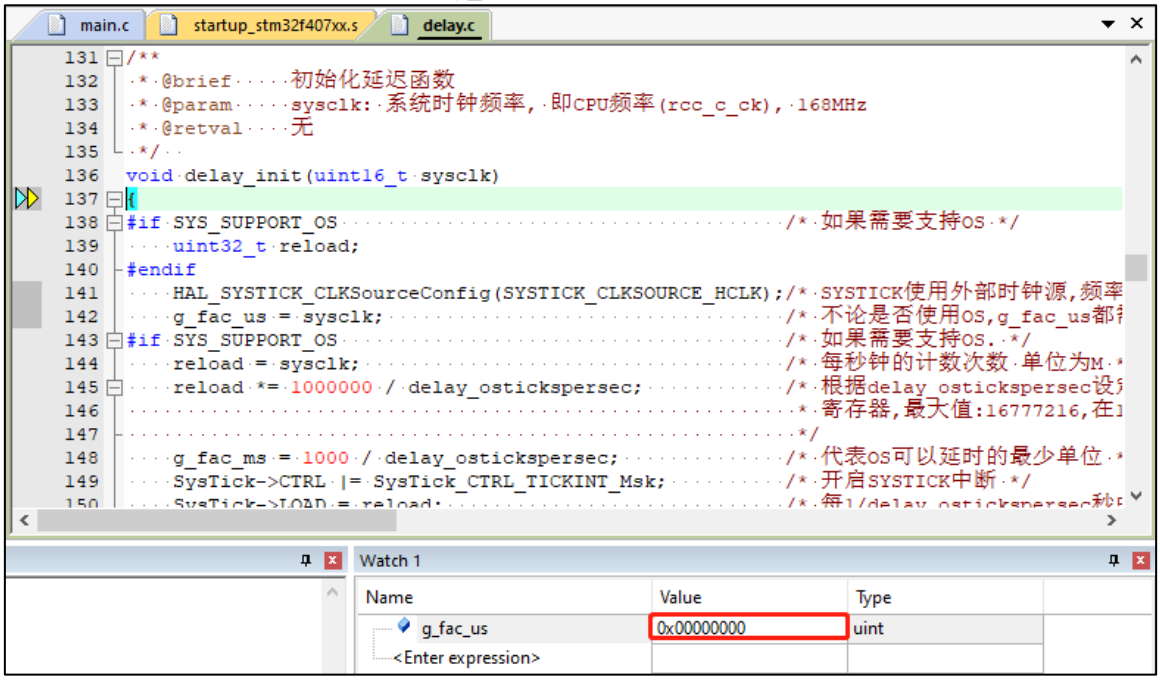


图 4.2.2.4 执行进入 delay_init 函数

此时，可以看到 g_fac_us 的值已经显示出来了，默认是 0（全局变量如果没有赋初值，一般默认都是 0），然后继续单击：，单步运行代码到 g_fac_us = sysclk; 这一行，再把鼠标光标放在 sysclk 上停留一会，就可以看到 MDK 自动提示 sysclk（delay_init 的输入参数）的值为：0X00A8，即 168，如图 4.2.2.5 所示：

```
136  void delay_init(uint16_t sysclk)
137  {
138  #if SYS_SUPPORT_OS
139      uint32_t reload;
140  #endif
141      HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
142      g_fac_us = sysclk;
143  #if SYS_SUPPORT_O  sysclk = 0x00A8
144      reload = sysclk;
145      reload *= 1000000 / delay_osticspersec;
146
```

图 4.2.2.5 单步运行到 g_fac_us 赋值

然后再单步执行过去这一行，就可以在 Watch1 窗口中看到 g_fac_us 的值变成 0X000000A8 了，如图 4.2.2.6 所示：

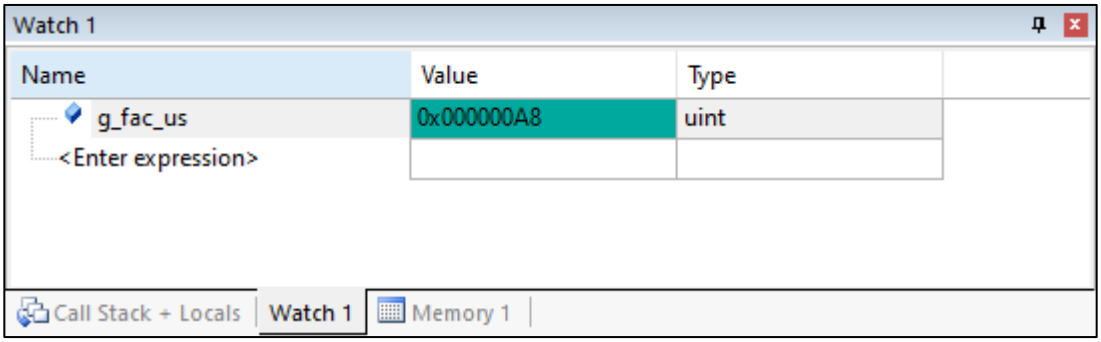


图 4.2.2.6 Watch1 窗口观看 g_fac_us 的值

由此可知，某些全局变量，我们在程序还没运行到其所在文件的时候，MDK 仿真时可能不会显示其值（如提示：cannot evaluate），当我们运行到其所在文件，并实际使用到的时候，此时就会显示其值出来了！

然后，我们再回到 main.c，在第一个 LED0(0)处放置一个断点，运行到断点处，如图 4.2.2.7 所示：

```
27  int main(void)
28  {
29      HAL_Init();                     /* 初始化HAL库 */
30      sys_stm32_clock_init(336, 8, 2, 7);  /* 设置时钟,168Mhz */
31      delay_init(168);                /* 延时初始化 */
32      led_init();                     /* 初始化LED */
33
34      while(1)
35      {
36          LED0(0);                    /* LED0 亮 */
37          LED1(1);                    /* LED1 灭 */
38          delay_ms(500);
39          LED0(1);                    /* LED0 灭 */
40          LED1(0);                    /* LED1 亮 */
41          delay_ms(500);
42      }
43  }
```

图 4.2.2.7 运行到 LED0(0)代码处

此时，我们点击：{}，执行过这一行代码，就可以看到开发板上的 DS0（红灯）亮起来了，以此继续点击{}，依次可以看到：DS0 灭→DS1 亮→DS0 亮→DS1 灭→DS0 灭→DS1 亮……，一直循环。这样如果全速运行，就可以看到 DS0，DS1 交替亮灭，不过全速运行的时候，一般是看不出 DS0 和 DS1 全亮的情况的，而通过仿真，我们就可以很容易知道有 DS0 和 DS1 同时亮的情况！

最后，我们在 delay.c 文件的 delay_us 函数，第二行处设置一个断点，然后运行到断点处，如图 4.2.2.8 所示：

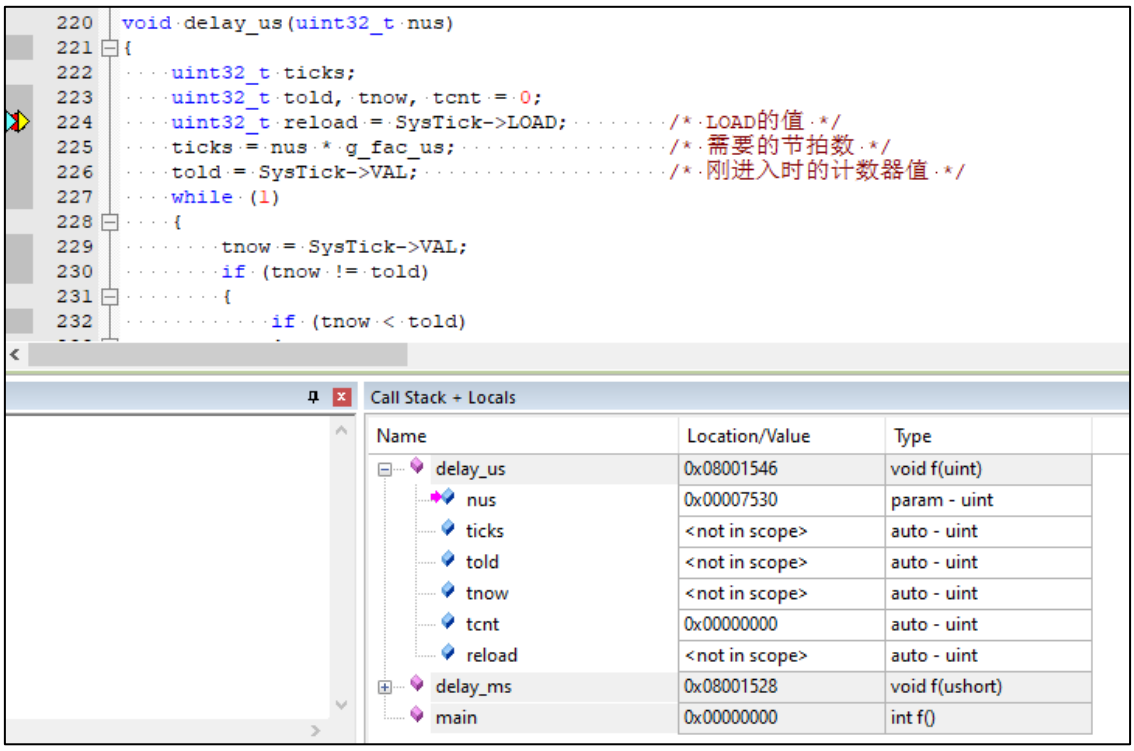


图 4.2.2.8 查看函数调用关系及局部变量

此时，我们可以从 Call Stack + Locals 窗口看到函数的调用关系，其原则是：**从下往上看**，即下一个函数调用了上一个函数，因此，其关系为：main 函数调用了 delay_ms 函数，然后 delay_ms 函数调用了 delay_us 函数。这样在一些复杂的代码里面（尤其是第三方代码），可以很容易捋出函数调用关系并查看其局部变量的值，有助于我们分析代码解决问题。

关于 DAP 的仿真调试，我们暂时就讲这么多。

4.2.3 仿真调试注意事项

1、由于 MDK5.23 以后对中文支持不是很好，具体现象是：在仿真的时候，当有断点未清除时点击结束仿真，会出现如图 4.2.3.1 所示的报错：

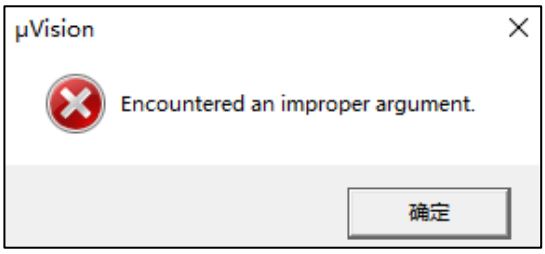


图 4.2.3.1 仿真结束时报错！

此时我们点击确定，是无法关闭 MDK 的，只能到电脑的任务管理器里面强制结束 MDK，才可以将其关闭，比较麻烦。

该错误就是由于 MDK5.23 以后的版本对中文支持不太好导致的，这里提供 2 个解决办法：**a，仿真结束前将所有设置的断点都清除掉**，可以使用 File 工具栏的：按钮，快速清除当前工程的所有断点，然后再结束仿真，就不会报错；**b，将工程路径改浅，并改成全英文路径**（比如，将源码拷贝到：E 盘→Source Code 文件夹下。注意：例程名字一般可以不用改英文，因为只要整个路径不超过 10 个汉字，一般就不会报错了，如果还报错就再减少汉字数量）。通过这两个方法，可以避免仿真结束报错的问题。我们推荐大家使用第二种方法，因为这样就不用每次都全部清除所有断点，下回仿真又得重设的麻烦。

2、关于 STM32 软件仿真，老版本的教程，我们给大家介绍过如何使用 MDK 进行 STM32 软件仿真，由于其限制较多（只支持部分 F1 型号），而且仿真器越来越便宜，硬件仿真更符合实际调试需求，调试效果更好。所以后续我们只介绍硬件仿真，不再推荐大家使用软件仿真了！

3、仿真调试找 bug 是一个软件工程师必备的基本技能。MDK 提供了很多工具和窗口来辅助我们找问题，只要多使用，多练习，肯定就可以把仿真调试学好。这对我们后续的独立开发项目，非常有帮助。因此极力推荐大家多练习使用仿真器查找代码 bug，学会这个基本技能。

4、调试代码不要浅尝辄止，要想尽办法找问题，具体的思路：先根据代码运行的实际现象分析问题，确定最可能出问题的地方，然后在相应的位置放置断点，查看变量，查看寄存器，分析运行状态和预期结果是否一致？从而找到问题原因，解决 bug。特别提醒：一定不要浅尝辄止，很多朋友只跟踪到最上一级函数，就说死机了，不会跟踪进去找问题！所以一定要一层层进入各种函数，越是底层（甚至汇编级别），越好找到问题原因。

4.3 MDK5 使用技巧

本节，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。

4.3.1 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。如果你刚装 MDK，没进行字体颜色配置，以跑马灯例程为例，你的界面效果如图 4.3.1.1 所示：

main.c

```
21  #include "./SYSTEM/sys/sys.h"
22  #include "./SYSTEM/delay/delay.h"
23  #include "./SYSTEM/usart/usart.h"
24  #include "./BSP/LED/led.h"
25
26
27  int main(void)
28  {
29      HAL_Init();                                 /* 初始化HAL库 */
30      sys_stm32_clock_init(RCC_PLL_MUL9);         /* 设置时钟,72M */
31      delay_init(72);                             /* 初始化延时函数 */
32      led_init();                                 /* 配置STM32操作LED相关的寄存器 */
33
34      while(1)
35      {
36          LED0(0);                                /* LED0 亮 */
37          LED1(1);                                /* LED1 灭 */
38          delay_ms(500);
39          LED0(1);                                /* LED0 灭 */
40          LED1(0);                                /* LED1 亮 */
41          delay_ms(500);
42      }
43  }
```

图 4.3.1.1 MDK 默认配色效果

上图是 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 4.3.1.2 所示界面：

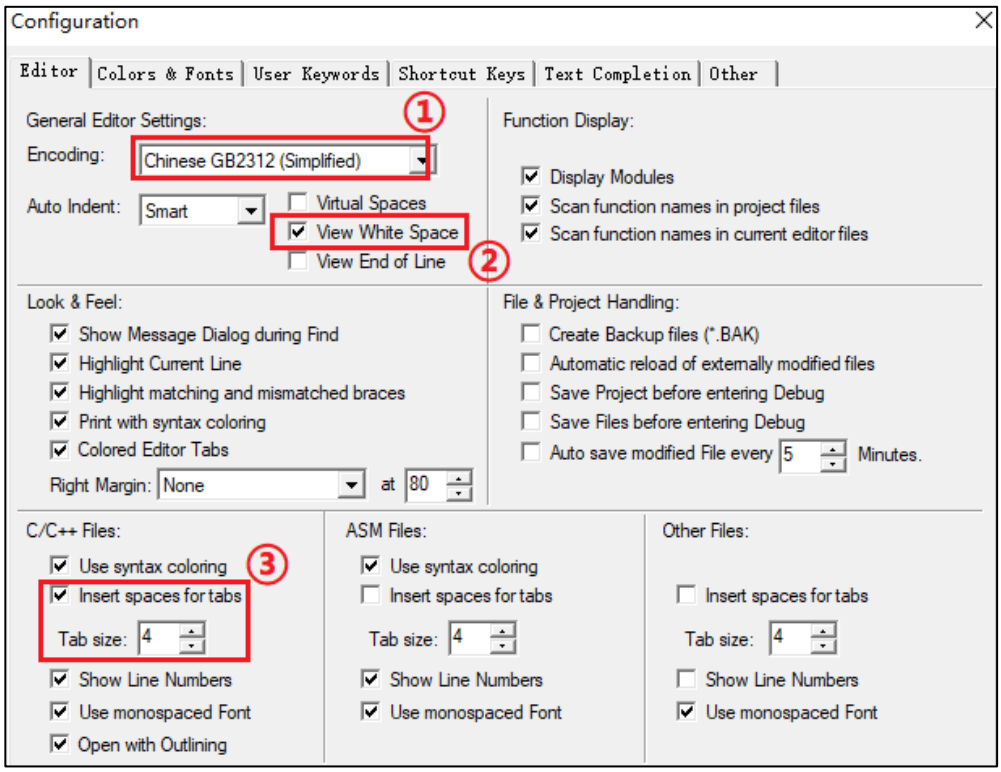


图 4.3.1.2 配置对话框

① 设置代码编辑器字体使用：Chinese GB2312(Simplified)，以更好的支持中文。

② 设置编辑器的空格可见：View White Space，所有空格使用“.”替代，TAB 使用“→”替代，这样可以方便我们对代码进行对齐操作。同时，我们推荐**所有的对齐都用空格**来替代，这样在不同软件之间查看源代码，就不会引起由于 TAB 键大小不一样导致代码不对齐的问题，方便使用不同软件查看和编辑代码。

③ 设置 C/C++文件，TAB 键的大小为 4 个字符，且字符使用空格替代（Insert spaces for tabs）。这样我们在使用 TAB 键进行代码对齐操作的时候，都会用空格替代，保证不同软件使用代码都可以对齐。

然后，选择：Colors & Fonts 选项卡，在该选项卡内，我们就可以设置自己的代码的字体和颜色了。由于我们使用的是 C 语言，故在 Window 下面选择：C/C++ Editor Files 在右边就可以看到相应的元素了。如图 4.3.1.3 示：

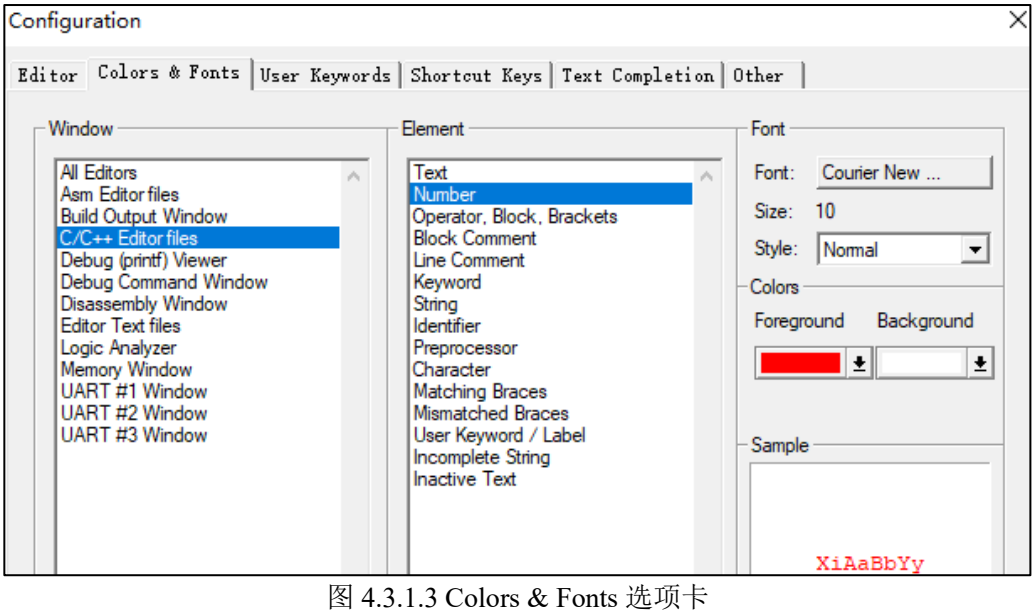


图 4.3.1.3 Colors & Fonts 选项卡

然后点击各个元素（Element）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。

然后，点击 User Keywords 选项卡，设置用户定义关键字，以便用户自定义关键字也显示对应的颜色（对应图 4.3.1.3 中的 User Keyword/Lable 颜色）。在 User Keywords 选项卡对话框下面输入你自己定义的关键字，如图 4.3.1.4 示：

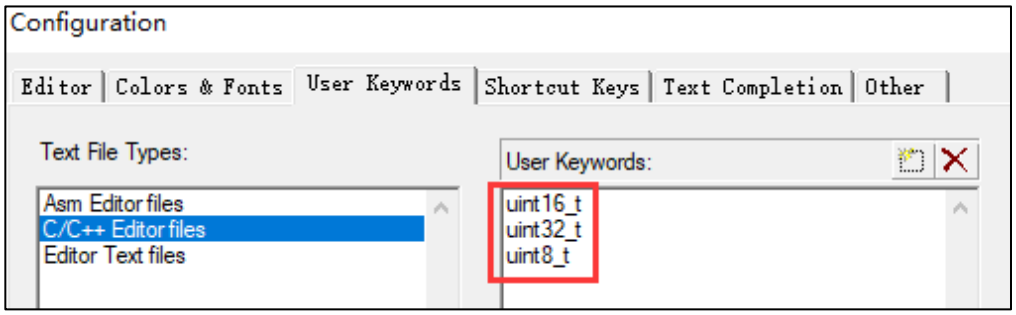


图 4.3.1.4 用户自定义关键字

这里我们设置了 uint8_t、uint16_t 和 uint32_t 等三个用户自定义关键字，相当于 unsigned char、unsigned short 和 unsigned int。如果你还有其他自定义关键字，在这里添加即可。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 4.3.1.5 示：

```
21  #include "./SYSTEM/sys/sys.h"
22  #include "./SYSTEM/delay/delay.h"
23  #include "./SYSTEM/usart/usart.h"
24  #include "./BSP/LED/led.h"
25
26
27  int main(void)
28  {
29      HAL_Init();                             /* 初始化HAL库 */
30      sys_stm32_clock_init(RCC_PLL_MUL9);     /* 设置时钟,72M */
31      delay_init(72);                         /* 初始化延时函数 */
32      led_init();                             /* 配置STM32操作LED相关的寄存器 */
33
34      while(1)
35      {
36          LED0(0);                            /* LED0 亮 */
37          LED1(1);                            /* LED1 灭 */
38          delay_ms(500);
39          LED0(1);                            /* LED0 灭 */
40          LED1(0);                            /* LED1 亮 */
41          delay_ms(500);
42      }
43  }
```

图 4.3.1.5 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。

同时，上图中可以看到空白处有很淡的一些……显示，这就是我们勾选了 View White Space 选项后体现出来的效果，可以方便我们对代码进行规范对齐整理。一开始看的时候可能有点不习惯，看多了就习惯了，大家慢慢适应就好了。

其实在这个编辑配置对话框里，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 4.3.2 节介绍。

4.3.2 语法检测&代码提示

MDK4.70 以上的版本，新增了代码提示与动态语法检测功能，使得 MDK 的编辑器越来越好用了，这里我们简单说一下如何设置，同样，点击🔧，打开配置对话框，选择 Text Completion 选项卡，如图 4.3.2.1 所示：

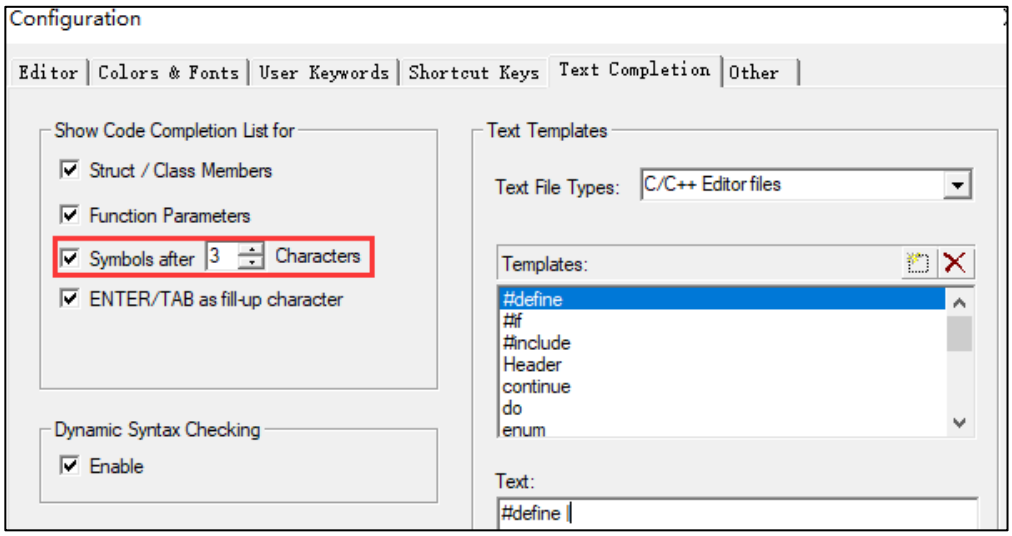


图 4.3.2.1 Text Completion 选项卡设置

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 4.3.2.2 所示：

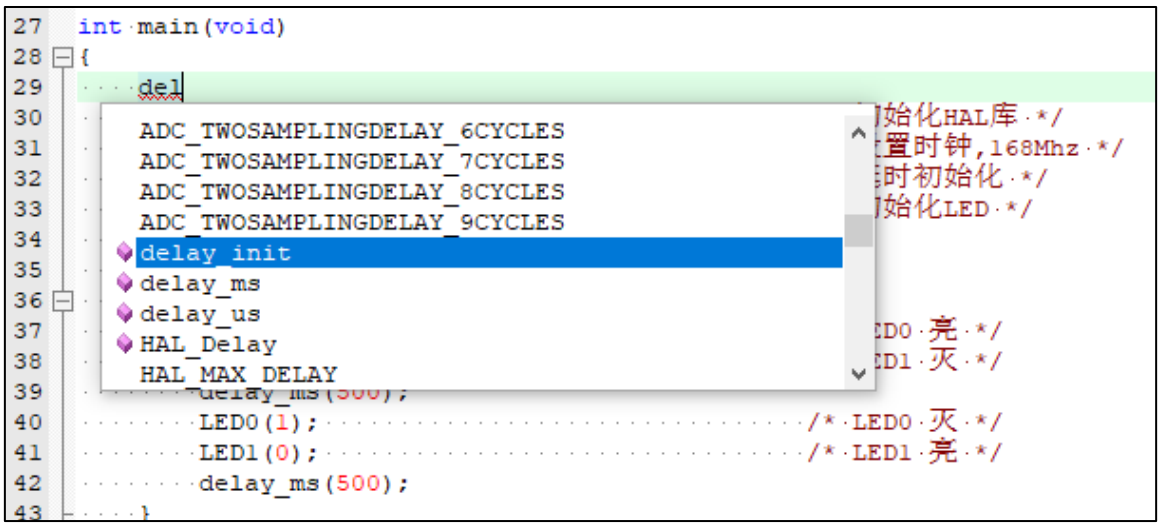


图 4.3.2.2 代码提示

ENTER/TAB as fill-up character，使用回车和 TAB 键填充字符。

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 4.3.2.3 所示：

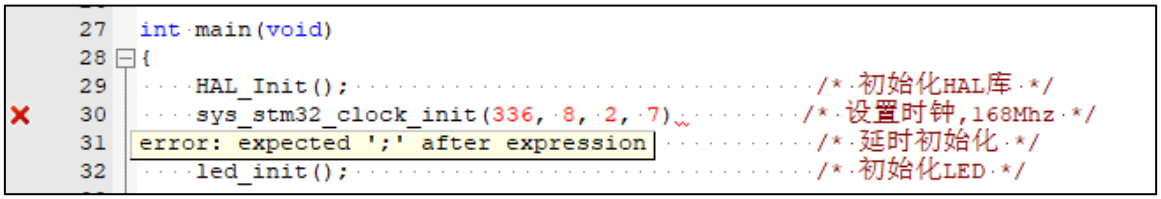


图 4.3.2.3 语法动态检测功能

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。**

4.3.3 代码编辑技巧

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

1. TAB 键的妙用

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

假设我们前面的串口 1 中断回调函数如图 4.3.3.1 所示：

```
149  void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
150  {
151    if (huart->Instance == USART_UX)                    /*如果是串口1*/
152    {
153      if ((g_usart_rx_sta & 0x8000) == 0)               /*接收未完成*/
154      {
155        if (g_usart_rx_sta & 0x4000)                    /*接收到了0x0d*/
156        {
157          if (aRxBuffer[0] != 0x0a)
158          {
159            g_usart_rx_sta = 0;                         /*接收错误,重新开始*/
160          }
161          else
162          {
163            g_usart_rx_sta |= 0x8000;                   /*接收完成了*/
164          }
165        }
166        else                                            /*还没收到0X0D*/
167        {
168          if (aRxBuffer[0] == 0x0d)
169            g_usart_rx_sta |= 0x4000;
170          else
171          {
172            g_usart_rx_buf[g_usart_rx_sta & 0X3FFF] = aRxBuffer[0];
173            g_usart_rx_sta++;
174
175            if (g_usart_rx_sta > (USART_REC_LEN - 1))
176            {
177              g_usart_rx_sta = 0;                       /*接收数据错误,重新开始接收      */
178            }
179          }
180        }
181      }
182    }
183  }
```

图 4.3.3.1 头大的代码

上图的代码很不规范，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。这时我们就可以通过 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 4.3.3.2 所示：

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART_UX)                    /*如果是串口1*/
    {
        if ((g_usart_rx_sta & 0x8000) == 0)             /*接收未完成*/
        {
            if (g_usart_rx_sta & 0x4000)                /*接收到了0x0d*/
            {
                if (aRxBuffer[0] != 0x0a)
                {
                    g_usart_rx_sta = 0;                 /*接收错误,重新开始*/
                }
                else
                {
                    g_usart_rx_sta |= 0x8000;           /*接收完成了*/
                }
            }
            else                                        /*还没收到0X0D*/
            {
                if (aRxBuffer[0] == 0x0d)
                    g_usart_rx_sta |= 0x4000;
                else
                {
                    g_usart_rx_buf[g_usart_rx_sta & 0X3FFF] = aRxBuffer[0];
                    g_usart_rx_sta++;

                    if (g_usart_rx_sta > (USART_REC_LEN - 1))
                    {
                        g_usart_rx_sta = 0;             /*接收数据错误,重新开始接收      */
                    }
                }
            }
        }
    }
}
```

图 4.3.3.2 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 4.3.3.3 所示：

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART_UX)                    /*如果是串口1*/
    {
        if ((g_usart_rx_sta & 0x8000) == 0)             /*接收未完成*/
        {
            if (g_usart_rx_sta & 0x4000)                /*接收到了0x0d*/
            {
                if (aRxBuffer[0] != 0x0a)
                {
                    g_usart_rx_sta = 0;                 /*接收错误,重新开始*/
                }
                else
                {
                    g_usart_rx_sta |= 0x8000;           /*接收完成了*/
                }
            }
            else                                        /*还没收到0X0D*/
            {
                if (aRxBuffer[0] == 0x0d)
                    g_usart_rx_sta |= 0x4000;
                else
                {
                    g_usart_rx_buf[g_usart_rx_sta & 0X3FFF] = aRxBuffer[0];
                    g_usart_rx_sta++;

                    if (g_usart_rx_sta > (USART_REC_LEN - 1))
                    {
                        g_usart_rx_sta = 0;             /*接收数据错误,重新开始接收      */
                    }
                }
            }
        }
    }
}
```

图 4.3.3.3 修改后的代码

图 4.3.3.3 中的代码相对于图 4.3.3.1 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。

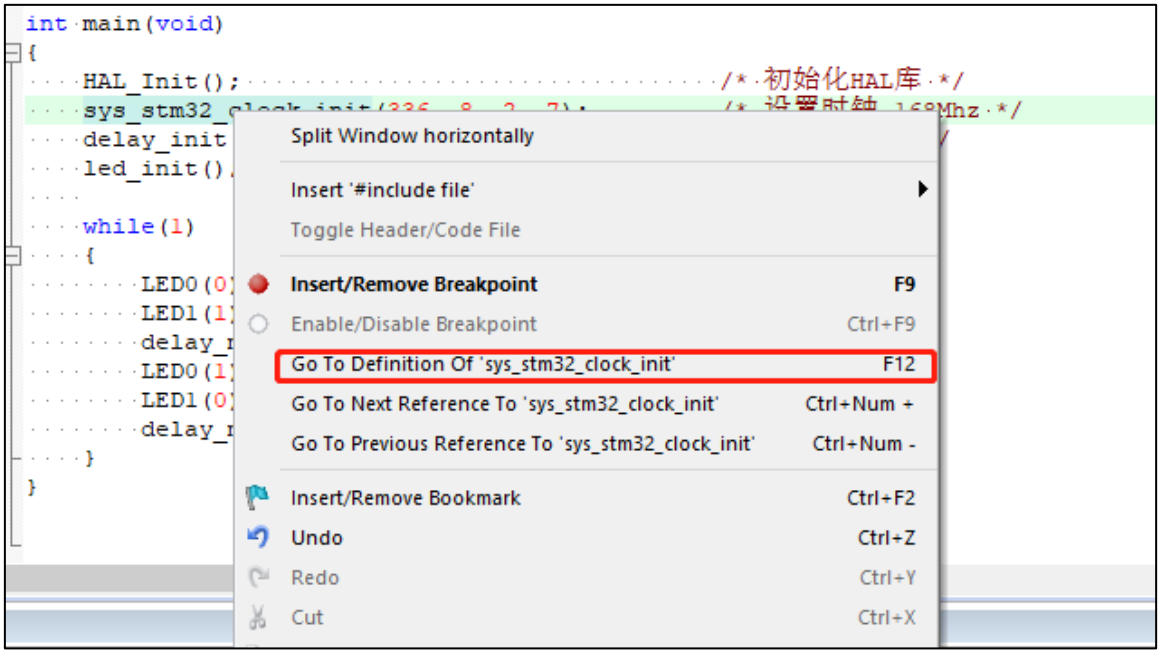


图 4.3.3.4 快速定位

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，可以看到如图 4.3.3.4 所示的菜单栏。

在图 4.3.3.4 中，我们找到 Go to Definition Of‘sys_stm32_clock_init’ 这个地方，然后单击左键就可以快速跳到 sys_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 4.3.3.5 所示：

```
124  uint8_t sys_stm32_clock_init(uint32_t plln, uint32_t pllm, uint32_t pllp, uint32_t pllq)
125 {
126     HAL_StatusTypeDef ret = HAL_OK;
127     RCC_ClkInitTypeDef rcc_clk_init_handle;
128     RCC_OscInitTypeDef rcc_osc_init_handle;
129
130     __HAL_RCC_PWR_CLK_ENABLE();                                    /* 使能PWR时钟 */
131
132     /* 下面这个设置用来设置调压器输出电压级别，以便在器件未以最大频率工作时使性能与功耗实现
133
134     __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);  /* VOS = 1, Scale1,
135
136     /* 使能HSE，并选择HSE作为PLL时钟源，配置PLL1，开启USB时钟 */
137     rcc_osc_init_handle.OscillatorType = RCC_OSCILLATORTYPE_HSE;   /* 时钟源为HSE */
138     rcc_osc_init_handle.HSEState = RCC_HSE_ON;                     /* 打开HSE */
139     rcc_osc_init_handle.PLL.PLLState = RCC_PLL_ON;                 /* 打开PLL */
140     rcc_osc_init_handle.PLL.PLLSource = RCC_PLLSOURCE_HSE;         /* PLL时钟源选择HSE
141     rcc_osc_init_handle.PLL.PLLN = plln;
142     rcc_osc_init_handle.PLL.PLLM = pllm;
143     rcc_osc_init_handle.PLL.PLLP = pllp;
144     rcc_osc_init_handle.PLL.PLLQ = pllq;
145
146     ret = HAL_RCC_OscConfig(&rcc_osc_init_handle);                 /* 初始化RCC */
147     if(ret != HAL_OK)
```

图 4.3.3.5 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 led_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 4.3.3.6 所示：

```
28  /**
29   * @brief       初始化LED相关IO口, 并使能时钟
30   * @param       无
31   * @retval      无
32   */
33  void led_init(void)
34  {
35      GPIO_InitTypeDef gpio_init_struct;
36      LED0_GPIO_CLK_ENABLE();                                 /* LED0时钟使能 */
37      LED1_GPIO_CLK_ENABLE();                                 /* LED1时钟使能 */
38
39      gpio_init_struct.Pin = LED0_GPIO_PIN;                   /* LED0引脚 */
40      gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
41      gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
42
43      gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
44      HAL_GPIO_Init(LED0_GPIO_PORT, &gpio_init_struct);       /* 初始化LED0引脚 */
45
46      gpio_init_struct.Pin = LED1_GPIO_PIN;                   /* LED1引脚 */
47      HAL_GPIO_Init(LED1_GPIO_PORT, &gpio_init_struct);       /* 初始化LED1引脚 */
48
49
50      LED0(1);                                                /* 关闭 LED0 */
51      LED1(1);                                                /* 关闭 LED1 */
52  }
```

图 4.3.3.6 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 4.3.3.7 所示：

```
/**
 * @brief       初始化LED相关IO口，并使能时钟
 * @param       无
 * @retval      无
 */
void led_init(void)
{
//    GPIO_InitTypeDef gpio_init_struct;
//    LED0_GPIO_CLK_ENABLE();                                 /* LED0时钟使能 */
//    LED1_GPIO_CLK_ENABLE();                                 /* LED1时钟使能 */

//    gpio_init_struct.Pin = LED0_GPIO_PIN;                   /* LED0引脚 */
//    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
//    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
//
//    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
//    HAL_GPIO_Init(LED0_GPIO_PORT, &gpio_init_struct);       /* 初始化LED0引脚 */

//    gpio_init_struct.Pin = LED1_GPIO_PIN;                   /* LED1引脚 */
//    HAL_GPIO_Init(LED1_GPIO_PORT, &gpio_init_struct);       /* 初始化LED1引脚 */
//

//    LED0(1);                                                /* 关闭 LED0 */
//    LED1(1);                                                /* 关闭 LED1 */
}
```

图 4.3.3.7 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。

4.3.4 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 4.3.4.1 所示：

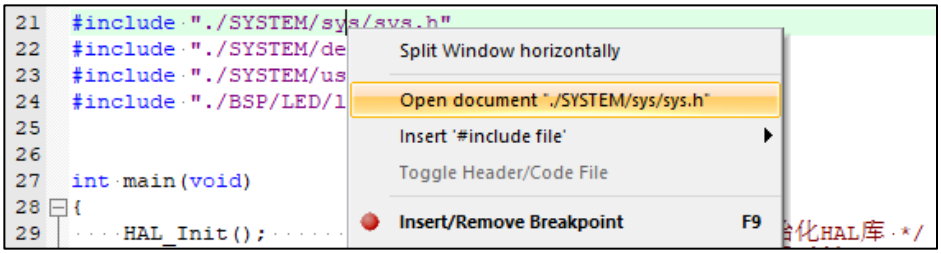


图 4.3.4.1 快速打开头文件

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 4.3.4.2 所示界面：

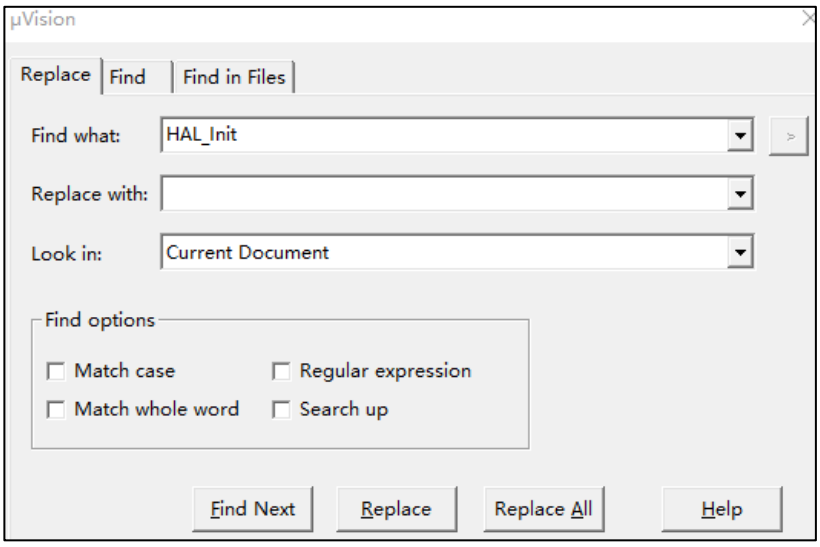


图 4.3.4.2 替换文本

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（这里以系统时钟初始化函数：sys_stm32_clock_init 为例），然后再点击 IDE 上面的，弹出如图 4.3.4.3 所示对话框：

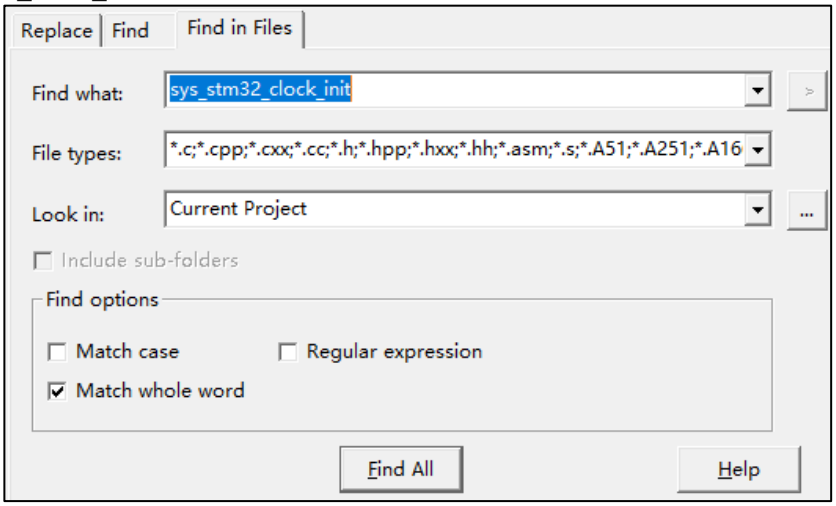


图 4.3.4.3 跨文件查找

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 4.3.4.4 所示：

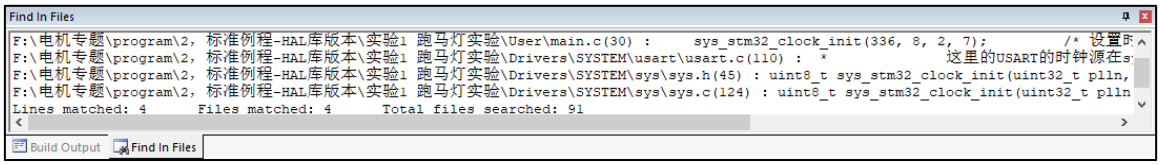


图 4.3.4.4 查找结果

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。

第五章 STM32 基础知识入门

本章，我们着重介绍 STM32 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 的学习做铺垫，方便后面的学习。本章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。

本章将分为如下几个小节:

5.1 C 语言基础知识复习
5.2 寄存器基础知识
5.3 STM32F407 系统架构
5.4 存储器映射
5.5 寄存器映射

5.1 C 语言基础知识复习

本节我们给大家介绍一下 C 语言基础知识，对于 C 语言比较熟练的读者，可以跳过此节，对于基础比较薄弱的读者，建议好好学习一下本节内容。

由于 C 语言博大精深，不可能我们一小节就全讲明白了，所以本节我们只是复习 STM32 开发时常用的几个 C 语言知识点，以便大家的更好的学习并编写 STM32 代码。

5.1.1 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	~	按位取反
|	按位或	<<	左移
^	按位异或	>>	右移

表 5.1.1.1 六种位操作

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，在不改变其他位的值的状况下，对某几个位进行设值。

这个场景在单片机开发中经常使用，方法就是先对需要设置的位用&操作符进行清零操作，然后用|操作符设值。比如我要改变 GPIOA 的 CRL 寄存器 bit6（第 6 位）的值为 1,可以先对寄存器的值进行&清零操作：

```
GPIOA->CRL &= 0XFFFFFFBF;    /* 将第 bit6 清 0 */
```

然后再与需要设置的值进行|或运算：

```
GPIOA->CRL |= 0X00000040;    /* 设置 bit6 的值为 1，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非常重要，下面是 delay_init 函数的一行代码：

```
SysTick->CTRL |= 1 << 1;
```

这个操作就是将 CTRL 寄存器的第 1 位（从 0 开始算起）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 1 位设置为 1。如果写成：

```
SysTick->CTRL |= 0X0002;
```

这个虽然也能实现同样的效果，但是可读性稍差，而且修改也比较麻烦。

3，~按位取反操作使用技巧

按位取反在设置寄存器的时候经常被使用，常用于清除某一个/某几个位。下面是 delay_us

函数的一行代码：

```
SysTick->CTRL &= ~(1 << 0) ;    /* 关闭 SYSTICK */
```

该代码可以解读为仅设置 CTRL 寄存器的第 0 位（最低位）为 0，其他位的值保持不变。同样我们也不使用按位取反，将代码写成：

```
SysTick->CTRL &= 0XFFFFFFFE;       /* 关闭 SYSTICK */
```

可见前者的可读性，及可维护性都要比后者好很多。

4，^按位异或操作使用技巧

该功能非常适合用于控制某个位翻转，常见的应用场景就是控制 LED 闪烁，如：

```
GPIOB->ODR ^= 1 << 5;
```

执行一次该代码，就会使 PB5 的输出状态翻转一次，如果我们的 LED 接在 PB5 上，就可以看到 LED 闪烁了。

5.1.2 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 HSE_VALUE    8000000U
```

定义标识符 HSE_VALUE 的值为 8000000，数字后的 U 表示 unsigned 的意思。

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。

5.1.3 ifdef 条件编译

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

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

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

```
#ifdef
     程序段 1
#endif
```

条件编译在 MDK 里面是用得很多，在 stm32f4xx.h 这个头文件中经常会看到这样的语句：

```
#if !defined (STM32F4)
#define STM32F4
#endif
```

如果没有定义 STM32F4 这个宏，则定义 STM32F4 宏。条件编译也是 c 语言的基础知识，这里也就点到为止吧。

5.1.4 extern 外部申明

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这样的语句：

```
extern uint16_t g_usart_rx_sta;
```

这个语句是申明 g_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：

```
uint16_t g_usart_rx_sta;
```

extern 的使用比较简单，但是也会经常用到，需要掌握。

5.1.5 typedef 类型别名

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

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

定义了一个结构体 GPIO，这样我们定义结构体变量的方式为：

```
   struct  _GPIO  gpiox;          /* 定义结构体变量 gpiox */
```

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

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

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

```
GPIO_TypeDef gpiox;
```

这里的 GPIO_TypeDef 就跟 struct _GPIO 是等同的作用了，但是 GPIO_TypeDef 使用起来方便很多。

5.1.6 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面的“寄存器映射”中讲到一些。

声明结构体类型：

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

例如：

```
struct U_TYPE
{
   int BaudRate
   int WordLength;
}usart1, usart2;
```

在结构体申明的时候可以定义变量，也可以申明之后定义，方法是：

```
struct  结构体名字   结构体变量列表 ;
```

例如：

```
struct U_TYPE usart1,usart2;
```

结构体成员变量的引用方法是：

```
结构体变量名字.成员名
```

比如要引用 usart1 的成员 BaudRate，方法是：usart1.BaudRate；结构体指针变量定义也是一样的，跟其他变量没有啥区别。

例如：

```
struct U_TYPE *usart3;     /* 定义结构体指针变量 usart3 */
```

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

```
usart3->BaudRate;
```

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

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

```
void usart_init(uint8_t usartx, uiut32_t BaudRate, uint32_t Parity,
                    uint32_t Mode);
```

这种方式是有效的同时在一定场合是可取的。但是试想，如果有一天，我们希望往这个函数里面再传入一个/几个参数，那么势必我们需要修改这个函数的定义，重新加入新的入口参数，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？

我们使用结构体参数，就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，usartx,BaudRate,Parity, Mode 等这些参数，他们对于串口而言，是一个有机整体，都是来设置串口参数的，所以我们可以将他们通过定义一个结构体来组合在一个有机整体。MDK 中是这样定义的：

```
typedef struct
{
  uint32_t BaudRate;
  uint32_t WordLength;
  uint32_t StopBits;
  uint32_t Parity;
  uint32_t Mode;
  uint32_t HwFlowCtl;
  uint32_t OverSampling;
} UART_InitTypeDef;
```

这样，我们在初始化串口的时候入口参数就可以是 USART_InitTypeDef 类型的变量或者指针变量了，于是我们可以改为：

```
void usart_init(UART_InitTypeDef *huart);
```

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。

5.1.7 指针

指针是一个值指向地址的变量（或常量），其本质是指向一个地址，从而可以访问一片内存区域。在编写 STM32 代码的时候，或多或少都要用到指针，它可以使不同代码共享同一片内存数据，也可以用作复杂的链接性的数据结构的构建，比如链表，链式二叉树等，而且，有些地方必须使用指针才能实现，比如内存管理等。

申明指针我们一般以 p 开头，如：

```
char * p_str = "This is a test!";
```

这样，我们就申明了一个 p_str 的指针，它指向 This is a test!这个字符串的首地址。我们编写如下代码：

```
int main(void)
{
   uint8_t temp = 0X88;          /* 定义变量 temp */
   uint8_t *p_num = &temp;       /* 定义指针 p_num，指向 temp 的地址 */
```

```
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 初始化串口 */
    printf("temp:0X%X\r\n", temp);                      /* 打印 temp 的值 */
    printf("*p_num: 0X %X\r\n", *p_num);                /* 打印*p_num 的值 */
    printf("p_num: 0X %X\r\n", (uint32_t)p_num);        /* 打印 p_num 的值 */
    printf("&p_num: 0X %X\r\n", (uint32_t)&p_num); /* 打印&p_num 的值 */
    while (1);
}
```

此代码的输出结果为：

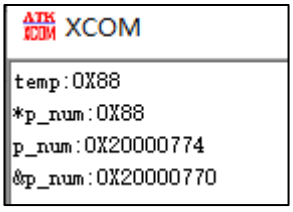


图 5.1.7.1 输出结果

p_num：是 uint8_t 类型指针，指向 temp 变量的地址，其值等于 temp 变量的地址。

*p_num：取 p_num 指向的地址所存储的值，即 temp 的值。

&p_num：取 p_num 指针的地址，即指针自身的地址。

以上，就是指针的简单使用和基本概念说明，指针的详细知识和使用范例大家可以百度学习，网上有非常多的资料可供参考。指针是 C 语言的精髓，在后续的代码中我们将会大量用到各种指针，大家务必好好学习和了解指针的使用。

5.2 寄存器基础知识

寄存器（Register）是单片机内部一种特殊的内存，它可以实现对单片机各个功能的控制，简单的来说可以把寄存器当成一些控制开关，控制包括内核及外设的各种状态。所以无论是 51 单片机还是 STM32，都需要用寄存器来实现各种控制，以完成不同的功能。

由于寄存器资源非常宝贵，一般都是一个位或者几个位控制一个功能，对于 STM32 来说，其寄存器是 32 位的，一个 32 位的寄存器，可能会有 32 个控制功能，相当于 32 个开关，由于 STM32 的复杂性，它内部有几百个寄存器，所以整体来说 STM32 的寄存器还是比较复杂的。不过，我们不要被其吓到了，实际上 STM32 是由于内部有很多外设，所以导致寄存器很多，实际上我们把它分好类，每个外设也就那么几个或者几十个寄存器，学起来就不难了。

从大方向来区分，STM32 寄存器分为两类，如表 5.2.1 所示：

大类	小类	说明
内核寄存器	内核相关寄存器	包含 R0~R15、xPSR、特殊功能寄存器等
	中断控制寄存器	包含 NVIC 和 SCB 相关寄存器，NVIC 有：ISER、ICER、ISPR、IP 等；SCB 有：VTOR、AIRCR、SCR 等
	SysTick 寄存器	包含 CTRL、LOAD、VAL 和 CALIB 四个寄存器
	内存保护寄存器	可选功能，STM32F407 没有
	调试系统寄存器	ETM、ITM、DWT、IPIU 等相关寄存器
外设寄存器		包含 GPIO、UART、IIC、SPI、TIM、DMA、ADC、DAC、RTC、I/WWDG、PWR、CAN、USB 等各种外设寄存器

表 5.2.1 STM32 寄存器分类

其中，内核寄存器，我们一般只需要关心中断控制寄存器和 SysTick 寄存器即可，其他三大类，我们一般很少直接接触。而外设寄存器，则是学到哪个外设，就了解哪个外设相关寄存器即可，所以整体来说，我们需要关心的寄存器并不是很多，而且很多都是有共性的，比如 STM32F407IGT6 有 14 个定时器，我们只需要学习了其中一个的相关寄存器，其他 12 个基本都是一样。

说了这么多，给大家举个简单的例子，我们知道寄存器的本质是一个特殊的内存，对于STM32 来说，以 GPIOB 的 ODR 寄存器为例，其寄存器地址为：0X40010C0C，所以我们对其赋值可以写成：

```
(*(unsigned int *))(0X40010C0C) = 0XFFFF;
```

这样我们就完成了对 GPIOB->ODR 寄存器的赋值，全部 0XFFFF，表示 GPIOB 所有 IO 口（16 个 IO 口）都输出高电平。对于我们来说，0X40010C0C 就是一个寄存器的特殊地址，至于它是怎么来的，我们后续再给大家介绍。

虽然上面的代码实现了我们需要的功能，但是从实用的角度来说，这么写肯定是不好的，可读性极差，可维护性也很差，所以一般我们使用结构体来访问，比如改写成这样：

```
GPIOB->ODR = 0XFFFF;
```

这样可读性就比之前的代码好多了，可维护性也相对好一点。至于 GPIOB 结构体怎么来的，我们也会在后续给大家介绍。

5.3 STM32F407 系统架构

STM32F407 是 ST 公司基于 ARM 授权 Cortex M4 内核而设计的一款芯片，而 Cortex M 内核使用的是 ARM v7-M 架构，是为了替代老旧的单片机而量身定做的一个内核，具有低成本、低功耗、实时性好、中断响应快、处理效率高等特点。

5.3.1 Cortex M4 内核 & 芯片

ARM公司提供内核（如Cortex M4，简称CM4，下同）授权，完整的MCU还需要很多其他组件。芯片公司（ST、NXP、TI、GD、华大等）在得到CM4内核授权后，就可以把CM4内核用在自己的硅片设计中，添加：存储器，外设，I/O以及其它功能块。不同厂家设计出的单片机会有不同的配置，包括存储器容量、类型、外设等都各具特色，因此才会有市面上各种不同应用的ARM芯片。Cortex M4内核和芯片的关系如图5.3.1.1所示：

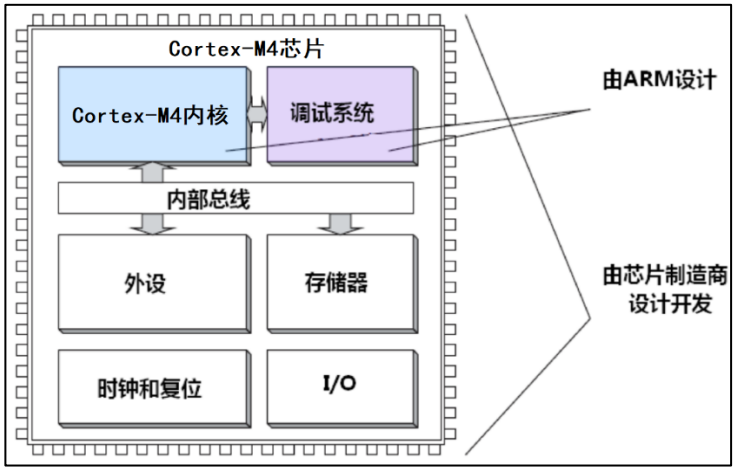


图 5.3.1.1 Cortex M4 内核 & 芯片关系

可以看到，ARM公司提供CM4内核和调试系统，其他的东西（外设（IIC、SPI、UART、TIM等）、存储器（SRAM、FLASH等）、I/O等）由芯片制造商设计开发。这里ST公司就是STM32F407芯片的制造商。

5.3.2 STM32 系统架构

STM32F407IGT6 系统架构如图 5.3.2.1：

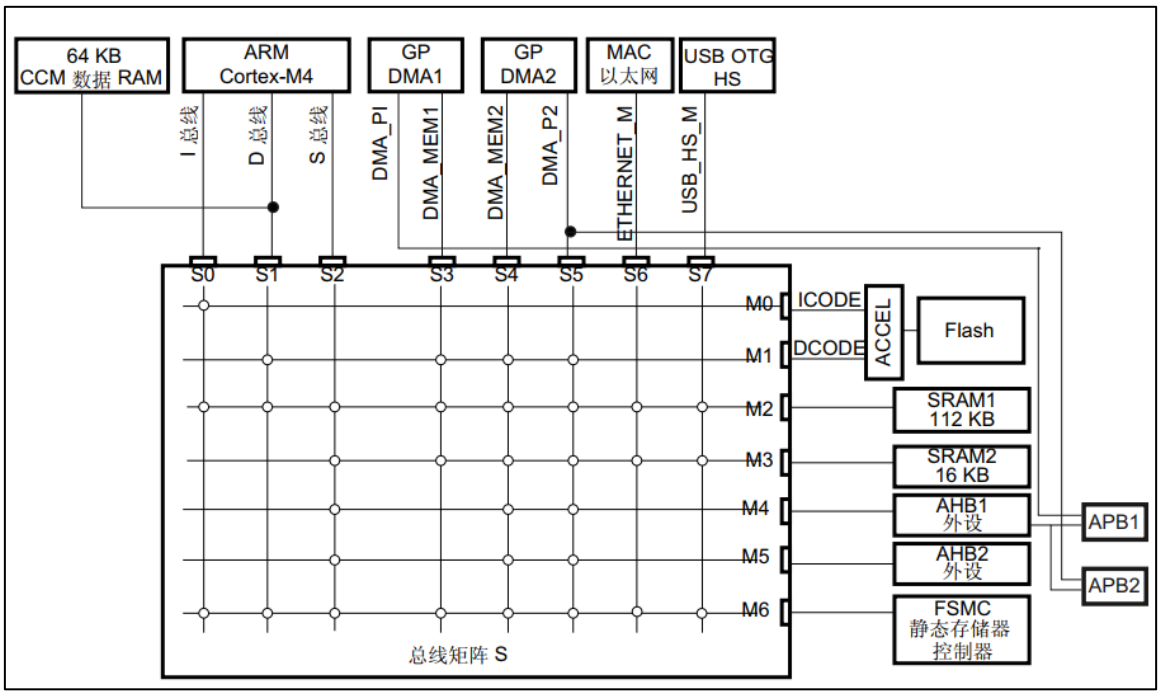


图5.3.2.1 STM32F407系统架构

主系统由32位多层AHB总线矩阵构成。总线矩阵用于主控总线之间的访问仲裁管理。这里的驱动/被动单元都是指连接了总线矩阵的部分，未连接总线矩阵的部分，则不算作驱动/被驱动单元。接下来我们介绍一下这些单元。

1. I Code 总线（I - Bus）

这是Cortex M4内核的指令总线，连接闪存指令接口（如：FLASH），用于获取指令。由于该总线功能单一，并没有直接连接到总线矩阵，因此被排除在驱动单元之外。

2. D Code 总线（D - Bus）

这是Cortex M4内核的数据总线，连接闪存存储器数据接口（如：SRAM、FLASH等），用于各种数据访问，如常量、变量等。

3. 系统总线（S - Bus）

这是Cortex M4内核的系统总线，连接所有外设（如：GPIO、SPI、IIC、TIM等），用于控制各种外设工作，如配置各种外设相关寄存器等。

4. DMA 总线

DMA是直接存储访问控制器，可以实现数据的自动搬运，整个过程不需要CPU处理。如可以实现DMA传输内存数据到DAC，输出任意波形，传输过程不需要CPU参与，可以大大节省CPU支，从而更高效的处理事务。STM32F407IGT6内部有2个DMA控制器，可以实现内存到外设、外设到内存、内存到内存的数据传输。

5. 内部 FLASH

内部FLASH即单片机的硬盘，用于代码/数据存储，CPU通过ICode总线经FLASH接口访问内部FLASH，FLASH最高访问速度是24Mhz，因此以168M速度访问时，需要插入8个时钟周期延迟。

6. 内部 SRAM

内部SRAM即单片机的内存，用于数据存储，直接挂载在总线矩阵上面，CPU通过DCode总线实现0等待延时访问SRAM，最快总线频率可达168Mhz，从而保证高效高速的访问内存。

7. FSMC

FSMC即灵活的静态存储控制器，实际上就是一个外部总线接口，可以用来访问外部SRAM、NAND/NOR FLASH、LCD等。它也是直接挂在总线矩阵上面的，以方便CPU快速访问外挂器件。

8. AHB/APB 桥

AHB总线连接总线矩阵，同时通过2个APB桥连接APB1和APB2，AHB总线速度最大为84Mhz，APB2总线速度最大也是84Mhz，但是APB1总线速度最大只能是42Mhz。这三个总线上面挂载了STM32内部绝大部分外设。

9. 总线矩阵

总线矩阵协调内核系统总线和DMA主控总线之间的访问仲裁，仲裁利用轮换算法，保证各个总线之间的有序访问，从而确保工作正常。

5.3.3 存储器映射

STM32是一个32位单片机，他可以很方便的访问4GB以内的存储空间（2^32＝4GB），因此Cortex M4内核将图5.3.2.1中的所有结构，包括：FLASH、SRAM、外设及相关寄存器等全部组织在同一个4GB的线性地址空间内，我们可以通过C语言来访问这些地址空间，从而操作相关外设（读/写）。数据字节以小端格式（小端模式）存放在存储器中，数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中。

存储器本身是没有地址信息的，我们对存储器分配地址的过程就叫存储器映射。这个分配一般由芯片厂商做好了，ST将所有的存储器及外设资源都映射在一个4GB的地址空间上（8个块），从而可以通过访问对应的地址，访问具体的外设。其映射关系如图5.3.3.1所示：

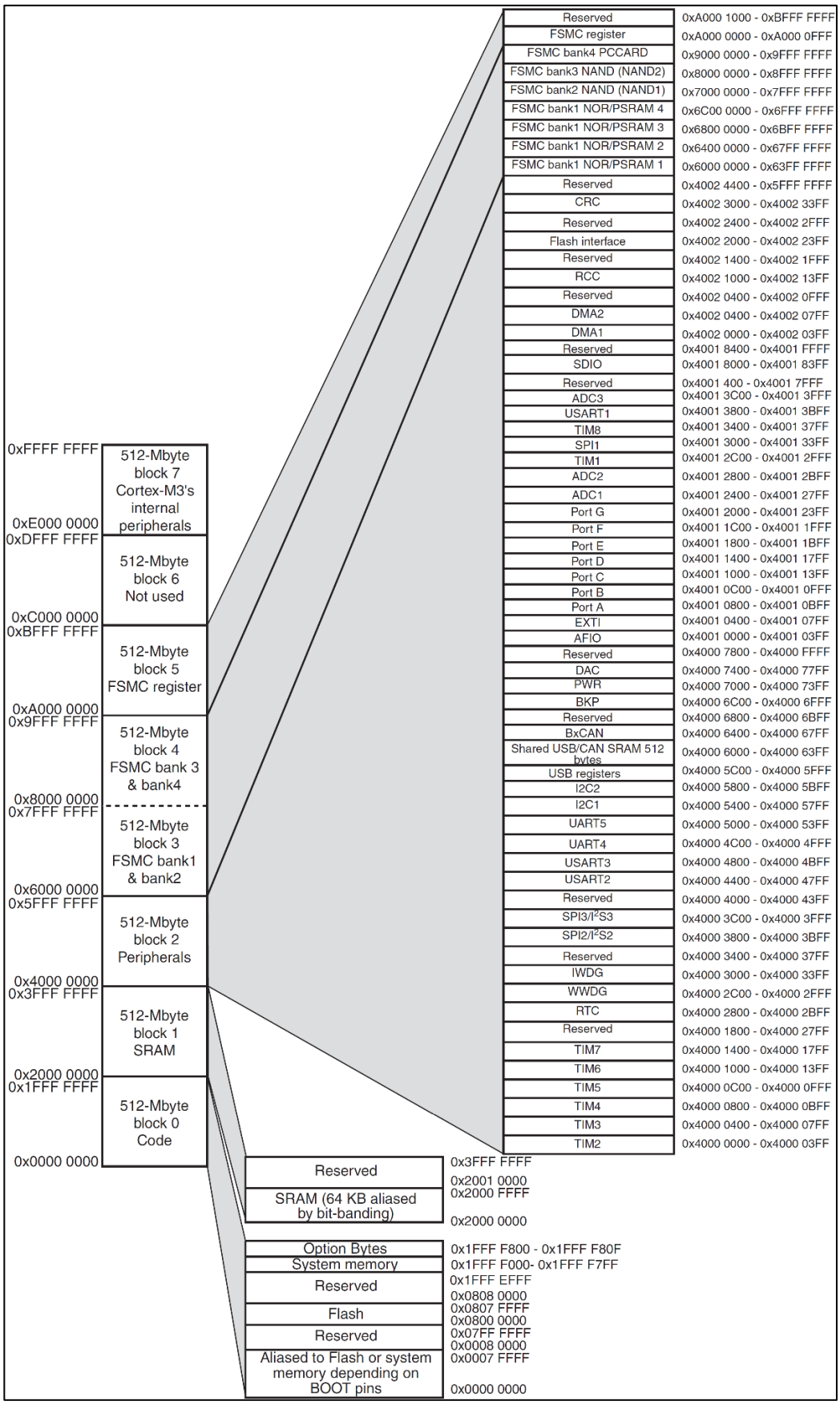


图5.3.3.1 STM32F407存储器映射

存储块功能介绍

ST将4GB空间分成8个块，每个块512MB，如上图所示，从图中我们可以看出有很多保留区域（Reserved），这是因为一般的芯片制造厂家是不可能把4GB空间用完的，同时，为了方便后续型号升级，会将一些空间预留（Reserved）。8个存储块的功能如表5.3.3.1所示：

存储块	功能	地址范围
Block 0	Code	0X0000 0000 ~ 0x1FFF FFFF（512MB）
Block 1	SRAM	0X2000 0000 ~ 0x3FFF FFFF（512MB）
Block 2	外设	0X4000 0000 ~ 0x5FFF FFFF（512MB）
Block 3	FSMC Bank1&2	0X6000 0000 ~ 0x7FFF FFFF（512MB）
Block 4	FSMC Bank3&4	0X8000 0000 ~ 0x9FFF FFFF（512MB）
Block 5	FSMC 寄存器	0XA000 0000 ~ 0xBFFF FFFF（512MB）
Block 6	没用到	0XC000 0000 ~ 0xDFFF FFFF（512MB）
Block 7	Cortex M4 内部外设	0XE000 0000 ~ 0xFFFF FFFF（512MB）

表5.3.3.1 STM32 存储块功能及地址范围

这里我们重点挑前面3个存储块给大家介绍。第一个块是Block 0，用于存储代码，即FLASH空间，其功能划分如表5.3.3.2所示：

存储块	功能	地址范围
Block 0	FLASH 或系统存储器别名区，取决于 BOOT 脚的设置	0X0000 0000 ~ 0x0007 FFFF（512KB）
	保留	0X0008 0000 ~ 0x07FF FFFF
	用户 FLASH，用于存储用户代码	0X0800 0000 ~ 0x0807 FFFF（512KB）
	保留	0X0808 0000 ~ 0x1FFF EFFF
	系统存储器，用于存储 STM32 出厂固化的 Bootloader 程序，比如用于串口下载代码	0X1FFF F000 ~ 0X1FFF F7FF（2KB）
	选项字节，用于配置读保护、设置看门狗等	0X1FFF F800 ~ 0X1FFF F80F（16B）
	保留	0X1FFF F810 ~ 0X1FFF FFFF

表5.3.3.2 STM32 存储块0的功能划分

可以看到，我们用户FLASH大小是1024KB，这是对于我们使用的STM32F407IGT6来说，如果是其他型号，可能FLASH会更小，当然，如果ST喜欢，也是可以随时推出更大容量的STM32单片机的，因为这里保留了一大块地址空间。还有，STM32的出厂固化BootLoader非常精简，整个BootLoder只占了2KB FLASH空间。

第二个块是Block 1，用于存储数据，即SRAM空间，其功能划分如表5.3.3.3所示：

存储块	功能	地址范围
Block 1	SRAM	0X2000 0000 ~ 0x2000 FFFF（64KB）
	保留	0X2001 0000 ~ 0x3FFF FFFF

表5.3.3.3 STM32 存储块1的功能划分

这个块仅使用了192KB大小（仅大容量STM32F407型号才有这么多SRAM，比如STM32F407IGT6等），用于SRAM访问，同时也有大量保留地址用于扩展。

第三个块是Block 2，用于外设访问，STM32内部大部分的外设都是放在这个块里面的，该存储块里面包括了AHB、APB1和APB2三个总线相关的外设。其功能划分如表5.3.3.4所示：

存储块	功能	地址范围
Block2	APB1 总线外设	0X4000 0000 ~ 0x4000 77FF
	保留	0X4000 7800 ~ 0x4000 FFFF
	APB2 总线外设	0X4001 0000 ~ 0x4000 3FFF
	保留	0X4001 4000 ~ 0x4001 7FFF
	AHB 总线外设	0X4001 8000 ~ 0x4002 33FF
	保留	0X4002 3400 ~ 0x5FFF FFFF

表5.3.3.4 STM32 存储块2的功能划分

同样可以看到，各个总线之间，都有预留地址空间，方便后续扩展。关于STM32各个外设具体挂在哪个总线上面,大家可以参考前面的 STM32F407系统结构图 和 STM32F407存储器映射图进行查找对应。

5.3.4 寄存器映射

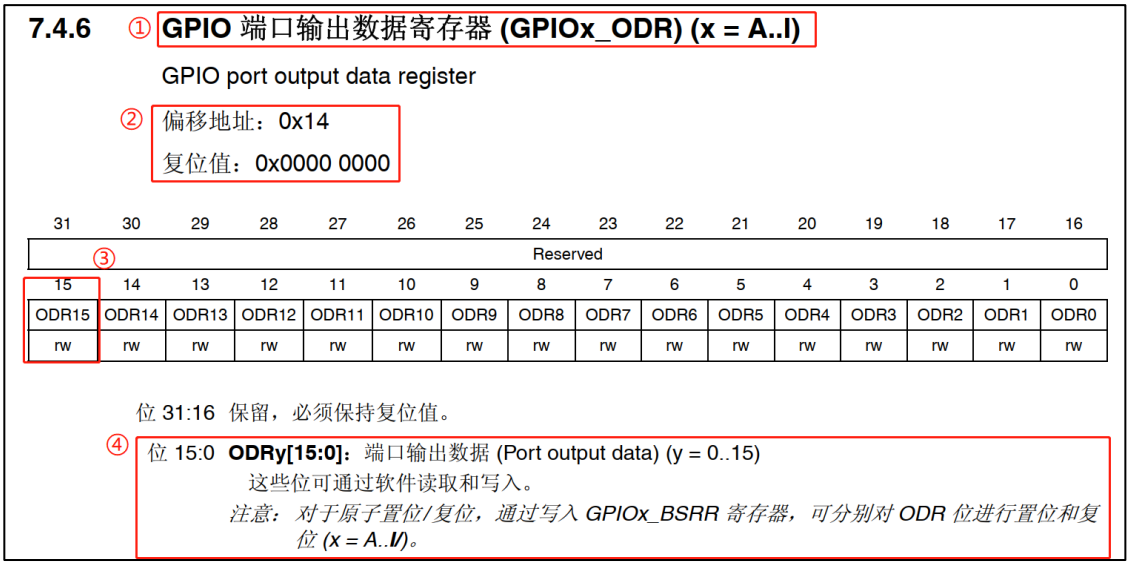
给存储器分配地址的过程叫存储器映射，寄存器是一类特殊的存储器，它的每个位都有特定的功能，可以实现对外设/功能的控制，给寄存器的地址命名的过程就叫寄存器映射。

举个简单的例子，大家家里面的纸张就好比通用存储器，用来记录数据是没问题的，但是不会有具体的动作，只能做记录，而你家里面的电灯开关，就好比寄存器了，假设你家有8个灯，就有8个开关（相当于一个8位寄存器），这些开关也可以记录状态，同时还能让电灯点亮/关闭，是会产生具体动作的。为了方便区分和使用，我们会给每个开关命名，比如厨房开关、大厅开关、卧室开关等，给开关命名的过程，就是寄存器映射。

当然STM32内部的寄存器有非常多，远远不止8个开关这么简单，但是原理是差不多的，每个寄存器的每一个位，一般都有特定的作用，涉及到寄存器描述，大家可以参考《STM32F4xx参考手册_V4（中文版）.pdf》相关章节的寄存器描述部分，有详细的描述。

1. 寄存器描述解读

我们以GPIO的ODR寄存器为例，其参考手册的描述如图5.3.4.1所示：



7.4.6 ① GPIO 端口输出数据寄存器 (GPIOx_ODR) (x = A..I)

GPIO port output data register

② 偏移地址：0x14

复位值：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
③ 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。

④ 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。

注意：对于原子置位/复位，通过写入 GPIOx_BSRR 寄存器，可分别对 ODR 位进行置位和复位 (x = A..I)。

图5.3.4.1 端口输出数据寄存器描述

①寄存器名字

每个寄存器都有一个对应的名字，以简单表达其作用，并方便记忆，这里GPIOx_ODR表示寄存器英文名，x可以从A~E，说明有5个这样的寄存器（每个端口有一个，事实上最新的STM32F407型号，可能还有F,G等端口，IO数量更多）。

②寄存器偏移量及复位值

地址偏移量表示相对该外设基地址的偏移，比如GPIOB，我们由图5.3.3.1可知其外设基地址是：0x4001 0C00。那么GPIOB_ODR寄存器的地址就是：0x4001 0C0C。知道了外设基地址和地址偏移量，我们就可以知道任何一个寄存器的实际地址。

复位值表示该寄存器在系统复位后的默认值，可以用于分析外设的默认状态。这里全部是0。

③寄存器位表

描述寄存器每一个位的作用（共32bit），这里表示ODR寄存器的第15位（bit），位名字为ODR15，rw表示该寄存器可读写（r，可读取；w，可写入）。

④位功能描述

描述寄存器每个位的功能，这里表示位0~15，对应ODR0~ODR15，每个位控制一个IO口的

输出状态。

其他寄存器描述，参照以上方法解读接口。

2. 寄存器映射举例

从前面的学习我们知道 GPIOB_ODR 寄存器的地址为：0x4001 0C0C，假设我们要控制 GPIOB 的 16 个 IO 口都输出 1，则可以写成：

```
*(unsigned int *)(0x40010C0C) = 0XFFFF;
```

这里我们先要将 0x4001 0C0C 强制转换成 unsigned int 类型指针，然后用*对这个指针的值进行设置，从而完成对 GPIOB_ODR 寄存器的写入。

这样写代码功能是没问题，但是可读性和可维护性都很差，使用起来极其不便，因此我们将代码改为：

```
#define GPIOB_ODR       *(unsigned int *)(0x40010C0C)
GPIOB_ODR = 0XFFFF;
```

这样，我们就定义了一个 GPIOB_ODR 的宏，来替代数值操作，很明显，GPIOB_ODR 的可读性和可维护性，比直接使用数值操作来的直观和方便。这个宏定义过程就可以称之为寄存器的映射。

当然，为了简单，我们只举了一个简单实例，实际上大量寄存器的映射，使用结构体是最方便的方式，stm32f407xx.h 里面使用结构体方式对 STM32F407 的寄存器做了详细映射，等下我们再介绍。

3. 寄存器地址计算

STM32F407大部分外设寄存器地址都是在存储块2上面的，见图5.3.3.1。具体某个寄存器地址，由三个参数决定：1、总线基地址（BUS_BASE_ADDR）；2，外设基于总线基地址的偏移量（PERIPH_OFFSET）；3，寄存器相对外设基地址的偏移量（REG_OFFSET）。可以表示为：

寄存器地址 = BUS_BASE_ADDR + PERIPH_OFFSET + REG_OFFSET

总线基地址（BUS_BASE_ADDR），STM32F407内部有三个总线（APB1、APB2和AHB），对应的总线基地址如表5.3.4.1所示：

总线	基地址	偏移量
APB1	0X4000 0000	0
APB2	0X4001 0000	0X1 0000
AHB	0X4001 8000	0X1 8000

表5.3.4.1 总线基地址

上表中APB1的基地址，也叫外设基地址，表中的偏移量就是相对于外设基地址的偏移量。

注意：AHB的总线基地址是0X4001 8000，从该基地址到0X4002 0000，只挂了SDIO一个外设，后续的AHB外设基地址都大于等于0X4002 0000。为了方便计算，我们可以将AHB的总线基地址改成：0X4002 0000，而SDIO则单独定义一个基地址给他即可。

外设基于总线基地址的偏移量（PERIPH_OFFSET），这个不同外设偏移量不一样，我们可以在STM32F407存储器映射图（图5.3.3.1）里面找到具体的偏移量，以GPIO为例，其偏移量如表5.3.4.2所示：

所属总线	外设	基地址	偏移量
APB2 0X4001 0000	GPIOA	0X4001 0800	0X800
	GPIOB	0X4001 0C00	0XC00
	GPIOC	0X4001 1000	0X1000
	GPIOD	0X4001 1400	0X1400
	GPIOE	0X4001 1800	0X1800
	GPIOF	0X4001 1C00	0X1C00
	GPIOG	0X4001 2000	0X2000

表5.3.4.2 GPIO外设基地址及相对总线偏移量

上表的偏移量，就是外设基于APB2总线基地址的偏移量（PERIPH_OFFSET）。

知道了外设基地址，再在参考手册里面找到具体某个寄存器相对外设基地址的偏移量就可

以知道该寄存器的实际地址了，以GPIOB的相关寄存器为例，如表5.3.4.3所示：

所属总线	所属外设	寄存器	地址	偏移量
APB2 0X4001 0000	GPIOB 0X4001 0C00	GPIOB_CRL	0X4001 0C00	0X00
		GPIOB_CRH	0X4001 0C04	0X04
		GPIOB_IDR	0X4001 0C08	0X08
		GPIOB_ODR	0X4001 0C0C	0X0C
		GPIOB_BSRR	0X4001 0C10	0X10
		GPIOB_BRR	0X4001 0C14	0X14
		GPIOB_LCKR	0X4001 0C18	0X18

表 5.3.4.3 GPIOB 寄存器相对外设基地址的偏移量

上表的偏移量，就是寄存器基于外设基地址的偏移量（REG_OFFSET）。

因此，我们根据前面的公式，很容易可以计算出GPIOB_ODR的地址：

GPIOB_ODR地址 = APB2总线基地址 + GPIOB外设偏移量 + 寄存器偏移量

所以得到：GPIOB_ODR地址 = 0X4001 0000 + 0XC00 + 0X0C = 0X4001 0C0C

关于寄存器地址计算我们就讲到这里，通过本节的学习，其他寄存器的地址大家都应该可以熟练掌握并计算出来。

4. stm32f407xx.h 寄存器映射说明

STM32F407所有寄存器映射都在stm32f407xx.h里面完成，包括各种基地址定义、结构体定义、外设寄存器映射、寄存器位定义（占了绝大部分）等，整个文件有1W多行，非常庞大。我们没有必要对该文件进行全面分析，因为很多内容都是相似的，我们只需要知道寄存器是如何被映射的，就可以了，至于寄存器位定义这些内容，知道是怎么回事就可以了。

我们还是以GPIO为例进行说明，看看stm32f407xx.h是如何对GPIO的寄存器进行映射的，通过对GPIO寄存器映射，了解stm32f407xx.h的映射规则。

stm32f407xx.h文件主要包含五个部分内容，如表5.3.4.4所示：

文件	主要组成部分	说明
stm32f407xx. h	中断编号定义	定义 IRQn_Type 枚举类型，包含 STM32F407 内部所有中断编号（中断号），方便后续编写代码
	外设寄存器结构体类型定义	以外设为基本单位，使用结构体类型定义对每个外设的所有寄存器进行封装，方便后面的寄存器映射
	寄存器映射	1，定义总线地址和外设基地址 2，使用外设结构体类型定义将外设基地址强制转换成结构体指针，完成寄存器映射
	寄存器位定义	定义外设寄存器每个功能位的位置及掩码
	外设判定	判断某个外设是否合法（即是否存在该外设）

表5.3.4.4 stm32f407xx.h文件主要组成部分

寄存器映射主要涉及到表5.3.4.4中加粗的两个组成部分：外设寄存器结构体类型定义和寄存器映射，总结起来，包括3个步骤：

1， 外设寄存器结构体类型定义

2， 外设基地址定义

3， 寄存器映射（通过将外设基地址强制转换为外设结构体类型指针即可）

以GPIO为例，其寄存器结构体类型定义如下：

```
typedef struct
{
  __IO uint32_t CRL;    /* GPIO_CRL  寄存器，相对外设基地址偏移量：0X00 */
  __IO uint32_t CRH;    /* GPIO_CRH  寄存器，相对外设基地址偏移量：0X04 */
  __IO uint32_t IDR;    /* GPIO_IDR  寄存器，相对外设基地址偏移量：0X08 */
  __IO uint32_t ODR;    /* GPIO_ODR  寄存器，相对外设基地址偏移量：0X0C */
  __IO uint32_t BSRR;   /* GPIO_BSRR 寄存器，相对外设基地址偏移量：0X10 */
  __IO uint32_t BRR;    /* GPIO_BRR  寄存器，相对外设基地址偏移量：0X14 */
  __IO uint32_t LCKR;   /* GPIO_LCKR 寄存器，相对外设基地址偏移量：0X18 */
```

```
} GPIO_TypeDef;
```

GPIO外设基地址定义如下：

```
#define PERIPH_BASE           0x40000000UL                    /* 外设基地址 */

#define APB1PERIPH_BASE       PERIPH_BASE                     /* APB1 总线基地址 */
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x00010000UL)    /* APB2 总线基地址 */
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x00020000UL)    /* AHB 总线基地址 */

#define GPIOA_BASE            (APB2PERIPH_BASE + 0x00000800UL)   /* GPIOA 基地址 */
#define GPIOB_BASE            (APB2PERIPH_BASE + 0x00000C00UL)   /* GPIOB 基地址 */
#define GPIOC_BASE            (APB2PERIPH_BASE + 0x00001000UL)   /* GPIOC 基地址 */
#define GPIOD_BASE            (APB2PERIPH_BASE + 0x00001400UL)   /* GPIOD 基地址 */
#define GPIOE_BASE            (APB2PERIPH_BASE + 0x00001800UL)   /* GPIOE 基地址 */
#define GPIOF_BASE            (APB2PERIPH_BASE + 0x00001C00UL)   /* GPIOF 基地址 */
#define GPIOG_BASE            (APB2PERIPH_BASE + 0x00002000UL)   /* GPIOG 基地址 */
```

GPIO外设寄存器映射定义如下：

```
#define GPIOA               ((GPIO_TypeDef *)GPIOA_BASE) /* GPIOA 寄存器地址映射 */
#define GPIOB               ((GPIO_TypeDef *)GPIOB_BASE) /* GPIOB 寄存器地址映射 */
#define GPIOC               ((GPIO_TypeDef *)GPIOC_BASE) /* GPIOC 寄存器地址映射 */
#define GPIOD               ((GPIO_TypeDef *)GPIOD_BASE) /* GPIOD 寄存器地址映射 */
#define GPIOE               ((GPIO_TypeDef *)GPIOE_BASE) /* GPIOE 寄存器地址映射 */
#define GPIOF               ((GPIO_TypeDef *)GPIOF_BASE) /* GPIOF 寄存器地址映射 */
#define GPIOG               ((GPIO_TypeDef *)GPIOG_BASE) /* GPIOG 寄存器地址映射 */
```

以上三部分代码，就完成了STM32F407内部GPIOA~GPIOG的寄存器映射，其原理其实是比较简单的，包括两个核心知识点：1，结构体地址自增；2，地址强制转换；

结构体地址自增，我们第一步就定义了GPIO_TypeDef结构体类型，其成员包括：CRL、CRH、IDR、ODR、BSRR、BRR和LCKR，每个成员是uint32_t类型，也就是4个字节，这样假设：CRL地址是0的话，CRH就是0X04，IDR就是0X08，ODR就是0X0C，以此类推。

地址强制转换，以GPIOB为例，GPIOB外设的基地址为：GPIOB_BASE（0X4001 0C00），我们使用GPIO_TypeDef将该地址强制转换为GPIO结构体类型指针：GPIOB，这样GPIOB->CRL的地址就是：GPIOB_BASE（0X4001 0C00），GPIOB->CRH的地址就是：GPIOB_BASE + 0X04（0X4001 0C04），GPIOB->IDR的地址就是：GPIOB_BASE + 0X08（0X4001 0C08），以此类推。

这样我们就使用结构体方式完成了对GPIO寄存器的映射，其他外设的寄存器映射也都是这个方法，这里就不一一介绍了。关于stm32f407xx.h的寄存器映射，我们就介绍到这里。

第六章 新建寄存器版本 MDK 工程

通过前面几章的学习，我们对 STM32 有了个比较清晰的了解，虽然没有寄存器的例程，但是本章我们将讲解新建寄存器库版本 MDK 工程的详细步骤。我们把本章新建好的工程放在光盘里，路径：**4，程序源码\1，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-2，新建最工程实验-寄存器版本**，大家在学习新建工程过程中间遇到一些问题，可以直接打开这个工程，然后对比学习。

本章将分为如下两个小节：

6.1 新建寄存器版本 MDK 工程

6.2 下载验证

6.1 新建寄存器版本 MDK 工程

本节我们将教大家如何新建一个 STM32F407 的 MDK5 工程。为了方便大家参考，我们将本节最终新建好的工程模板存放在 A 盘：4、程序源码\1，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-2，新建工程实验-寄存器版本，如遇新建工程问题，请打开该实验对比。

整个新建过程比较复杂，我们将其拆分为 5 个步骤进行讲解，请准备大概 2 个小时时间，耐心细致的做完！对你后续的学习非常有帮助！。

在新建工程之前，首先我们要做如下准备：

1， STM32Cube 官方固件包：我们使用的固件包版本是 STM32Cube_FW_F4_V1.26.0，固件包路径：A 盘→8，STM32 参考资料→1，STM32CubeF4 固件包。

2， 开发环境搭建：参考本书第三章相关内容。

6.1.1 新建工程文件夹

新建工程文件夹分为 2 个步骤：1，新建工程文件夹；2，拷贝工程相关文件。

1. 新建工程文件夹

首先我们在桌面新建一个工程根目录文件夹，后续的工程文件都将在这个文件夹里建立，我们把这个文件夹重命名为：实验 0-2，新建工程实验-寄存器版本。如图 6.1.1.1 所示：

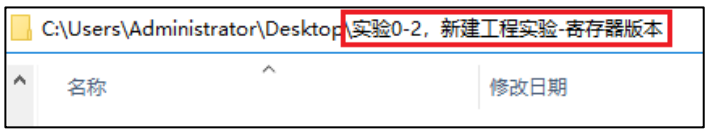


图 6.1.1.1 新建工程根目录文件夹

为了让工程的文件目录结构更加清晰易懂，我们会在工程根目录文件夹下建立以下几个文件夹，每个文件夹名称及其作用如表 6.1.1.1 所示：

文件夹名称	作用
Drivers	存放与硬件相关的驱动层文件
Middlewares	存放正点原子提供的中间层组件文件和第三方中间层文件
Output	存放工程编译输出文件
Projects	存放 MDK 工程文件
User	存放用户编写的代码，如 main.c

表 6.1.1.1 工程根目录新建文件夹及其作用

新建完成以后，最后得到我们的工程根目录文件夹如图 6.1.1.2 所示。

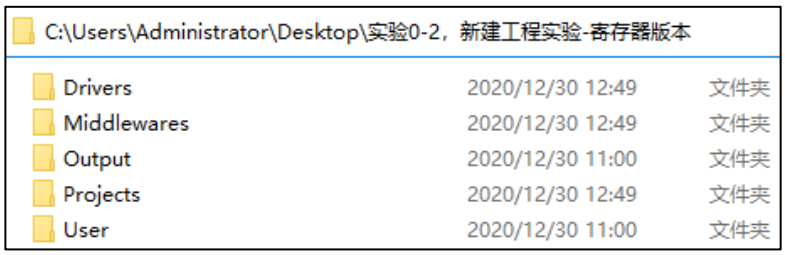


图 6.1.1.2 工程根目录文件夹

工程根目录及其相关文件夹新建好以后，我们需要拷贝一些工程相关文件过来（主要是在 Drivers 文件夹里面），以便等下的新建工程需要。

2. 拷贝工程相关文件

接下来，我们按图 6.1.1.2 的根目录文件夹顺序介绍每个文件夹及其需要拷贝的文件。

Drivers 文件夹

该文件夹用于存放与硬件相关的驱动层文件，一般包括如表 6.1.1.2 所示的三个文件夹：

文件夹名称	作用
BSP	存放开发板板级支持包驱动代码，如各种外设驱动
CMSIS	存放 CMSIS 底层代码，如启动文件（.s 文件）、stm32f4xx.h 等
SYSTEM	存放正点原子系统级核心驱动代码，如 sys.c、delay.c 和 usart.c 等

表 6.1.1.2 Drivers 包含文件夹

BSP 文件夹，用于存放正点原子提供的板级支持包驱动代码，如：LED、蜂鸣器、按键等。本章我们暂时用不到该文件夹，不过可以先建好备用。

CMSIS 文件夹，用于存放 CMSIS 底层代码（ARM 和 ST 提供），如：启动文件（.s 文件）、stm32f4xx.h 等各种头文件。该文件夹我们可以直接从 STM32CubeF4 固件包（路径：A 盘→8，STM32 参考资料→1，STM32CubeF4 固件包）里面拷贝，不过由于固件包里面的 CMSIS 兼容了太多芯片，导致非常大（100 多 MB），因此我们根据实际情况，对其进行了大幅精简，精简后的 CMSIS 文件夹大小为 1MB 左右。精简后的 CMSIS 文件夹大家可以在：A 盘→4，程序源码→1，标准例程-寄存器版本 文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

SYSTEM 文件夹，用于存放正点原子提供的系统级核心驱动代码，如：sys.c、delay.c 和 usart.c 等，方便大家快速搭建自己的工程。该文件同样可以从：A 盘→4，程序源码→1，标准例程-寄存器版本 文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

执行完以上操作后，Drivers 文件夹最终结构如图 6.1.1.3 所示：

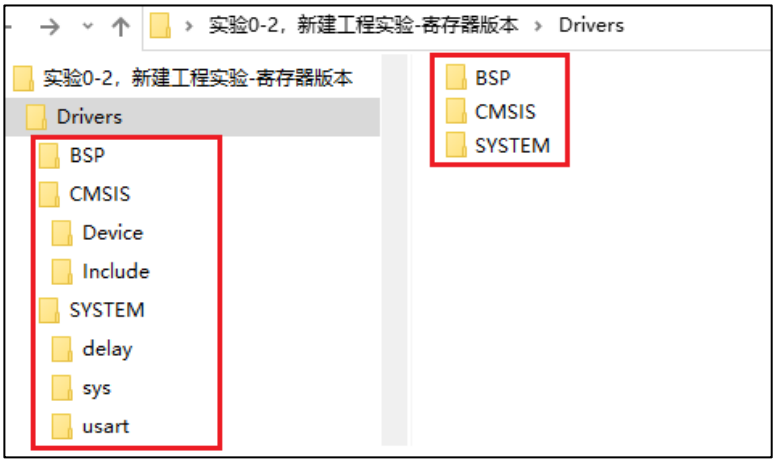


图 6.1.1.3 工程根目录文件夹

Middlewares 文件夹

该文件夹用于存放正点原子和其他第三方提供的中间层代码（组件/Lib 等），如：USMART、MALLOC、TEXT、FATFS、USB、LWIP、各种 OS、各种 GUI 等等。本章我们暂时用不到该文件夹，不过可以先建好备用，后面的实验将会陆续添加各种文件。

Output 文件夹

该文件夹用于存放编译器编译工程输出的中间文件，比如：.hex、.bin、.o 文件等等。这里不需要操作，后面只需要在 MDK 里面设置该文件夹为编译过程中间文件的存放文件夹就行。

Projects 文件夹

该文件夹用于存放编译器（MDK、IAR 等）工程文件，我们主要用 MDK，为了方便区分，我们在该文件夹下新建：MDK-ARM 文件夹，用于存放 MDK 的工程文件，如图 6.1.1.4 所示：



图 6.1.1.4 在 Projects 文件夹下新建 MDK-ARM 文件夹

User 文件夹

该文件夹用于存放用户编写的代码，如：main.c 等。目前还没有任何代码，所以暂时为空即可。

6.1.2 新建一个工程框架

首先，打开 MDK 软件。然后点击 Project→New uVision Project 如图 6.1.2.1 所示：

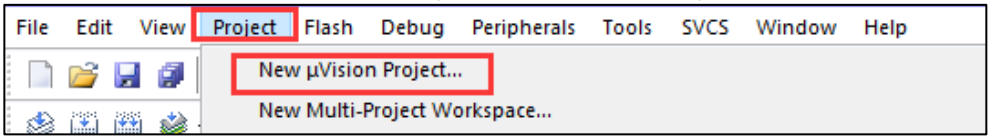


图 6.1.2.1 新建 MDK 工程

然后弹出工程命名和保存的操作窗口，我们将工程文件保存路径设置在上一节新建的工程文件夹内，具体路径为：桌面→实验 0-2，新建工程实验-寄存器版本→Projects→MDK-ARM，工程名字我们取：atk_f407，最后点击保存即可。具体操作窗口如图 6.1.2.2 所示：

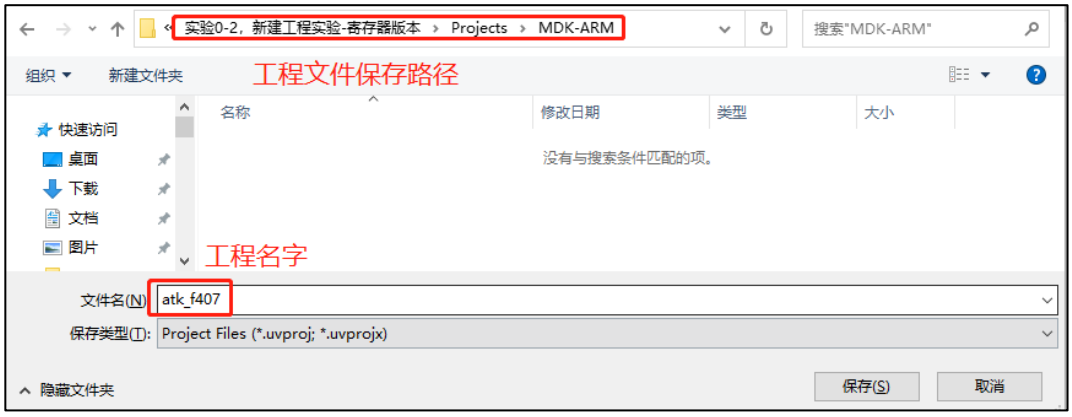


图 6.1.2.2 保存工程界面

之后，弹出器件选择对话框，如图 6.1.2.3 所示。因为电机开发板所使用的 STM32 型号为 STM32F407IGT6，所以我们选择：STMicroelectronics → STM32F4 Series → STM32F407 → STM32F407IGTx（如果使用的是其他系列的芯片，选择相应的型号就可以了，特别注意：**一定要安装对应的器件 pack** 才会显示这些内容哦！！如果没得选择，请关闭 MDK，然后安装 A 盘：6，软件资料\1，软件\1，MDK5\ Keil.STM32F4xx_DFP.2.8.0.pack 这个安装包后重试）。

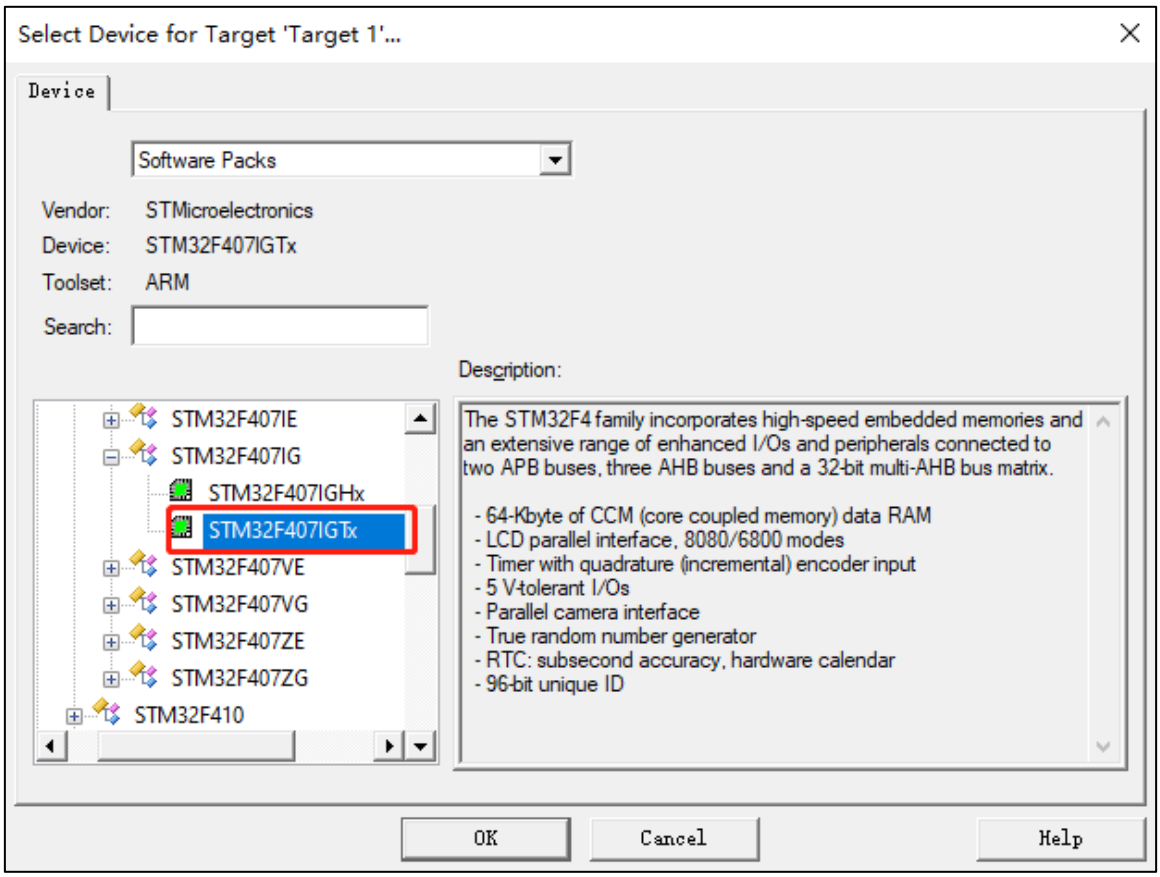


图 6.1.2.3 器件选择界面

点击 OK，MDK 会弹出 Manage Run-Time Environment 对话框，如图 6.1.2.4 所示：

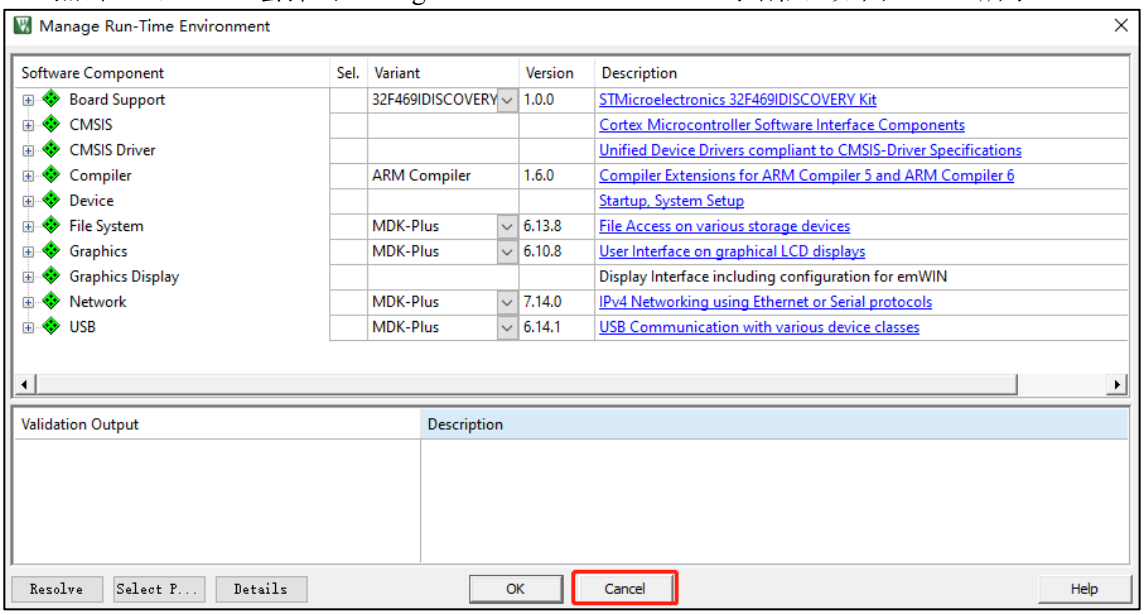


图 6.1.2.4 Manage Run-Time Environment 界面

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 6.1.2.4 所示界面，我们直接点击 Cancel，即可，得到如图 6.1.2.5 所示界面：

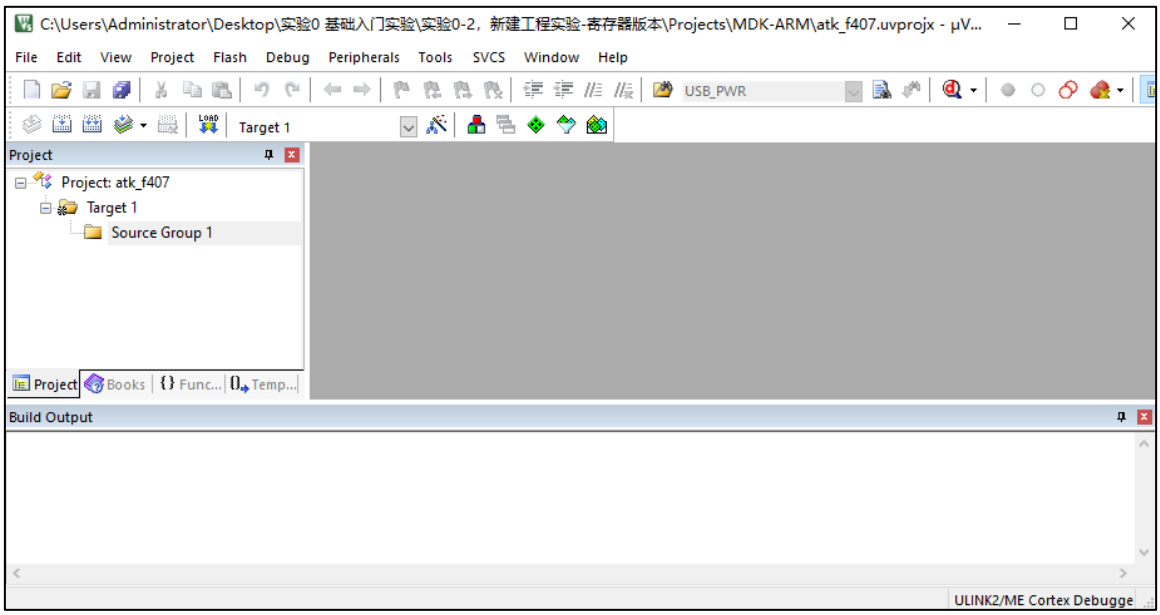


图 6.1.2.5 工程初步建立

此时，我们打开 MDK-ARM 文件夹，会看到 MDK 在该文件夹下自动创建了 3 个文夹（DebugConfig、Listings 和 Objects），如图 6.1.2.6 所示：

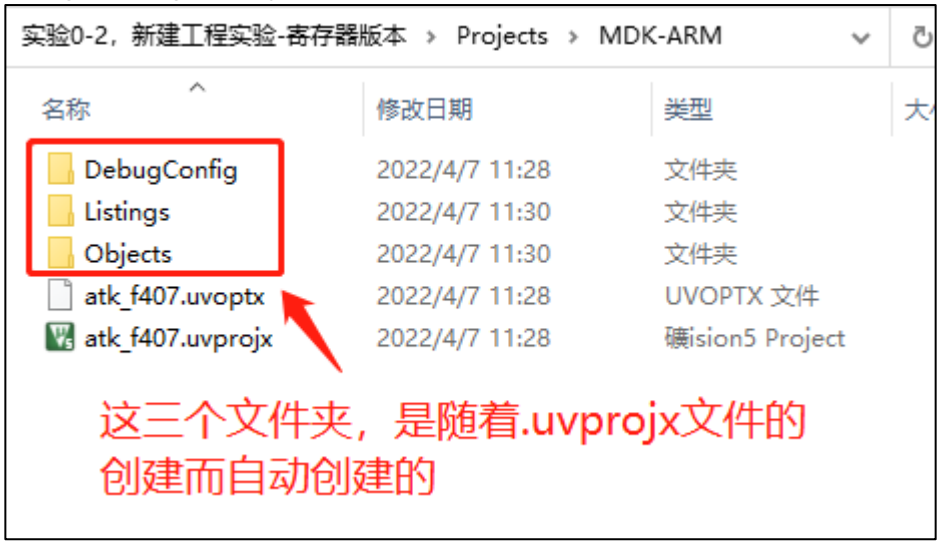


图 6.1.2.6 MDK 新建工程时自动创建的文件夹

这三个文件夹的作用如表 6.1.2.1 所示：

文件夹	作用
DebugConfig	用于存放调试设置信息文件（.dbgconf），不可删除！
Listings	用于存放编译过程产生的链接列表等文件
Objects	用于存放编译过程产生的调试信息、.hex、预览、.lib 文件等

表 6.1.2.1 三个文件夹及其作用

编译过程产生的链接列表、调试信息、预览、lib 等文件，统称为中间文件。为了统一管理，方便使用，我们会把输出在 Listings 和 Objects 文件夹的内容，统一改为输出到 Output 文件夹（通过魔术棒设置），我们先**把 MDK 自动生成的这两个文件夹（Listings 和 Objects）删除**。

至此，我们还只是建了一个框架，还有好几个步骤要做，比如添加文件、魔术棒设置、编写 main.c 等。

6.1.3 添加文件

本节将分 3 个步骤：1，设置工程名和分组名；2，添加启动文件；3，添加 SYSTEM 源码。

1. 设置工程名和分组名

在 Project→Target 上右键，选择 Manage Project Items…（方法一）或在菜单栏点击品字形红绿白图标（方法二）进入工程管理界面，如图 6.1.3.1 所示：

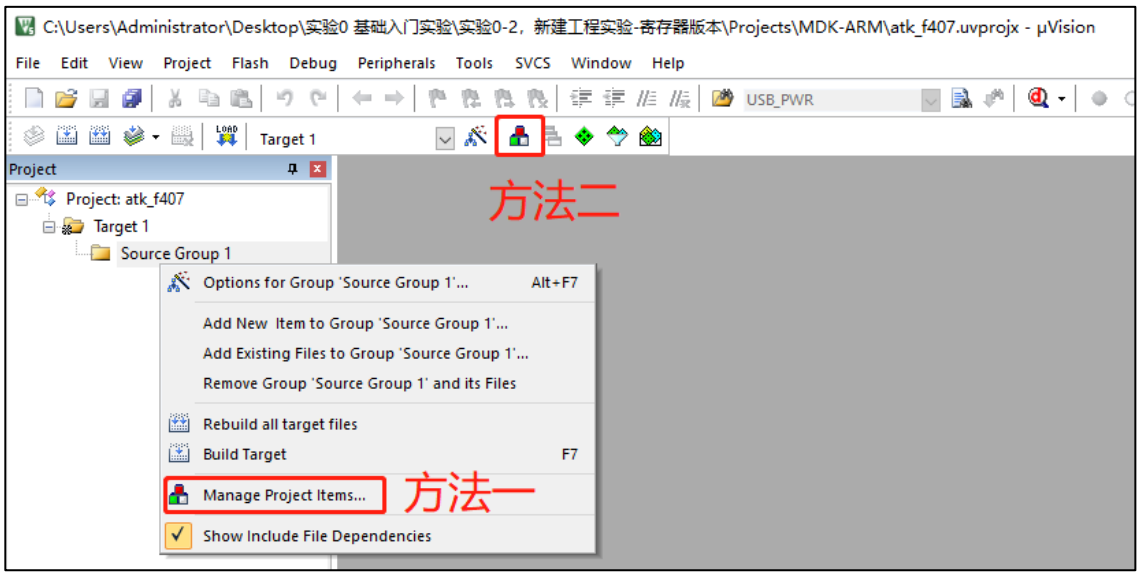


图 6.1.3.1 进入工程管理界面

在工程管理界面，我们可以执行设置工程名字（Project Targets）、分组名字（Groups）以及添加每个分组的文件（Files）等操作。我们设置工程名字为：Template，并设置四个分组：Startup（存放启动文件）、User（存放 main.c 等用户代码）、Drivers/SYSTEM（存放系统级驱动代码）、Readme（存放工程说明文件），如图 6.1.3.2 所示：

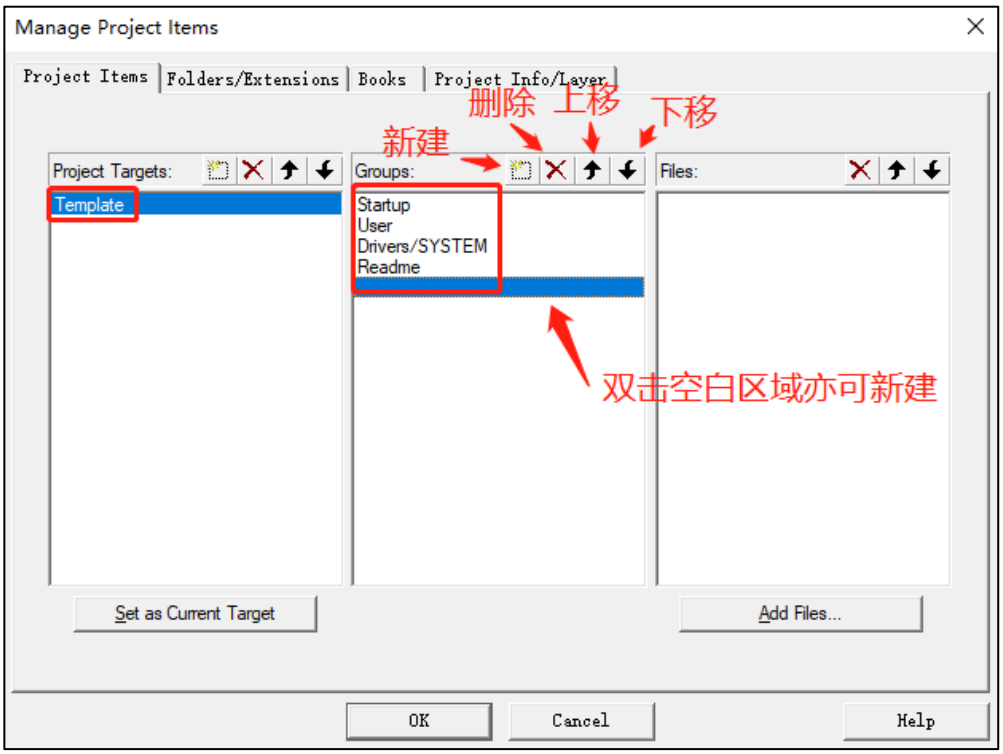


图 6.1.3.2 设置工程名和分组名

设置好之后，我们点击 OK，回到 MDK 主界面，可以看到我们设置的工程名和分组名如图 6.1.3.3 所示：

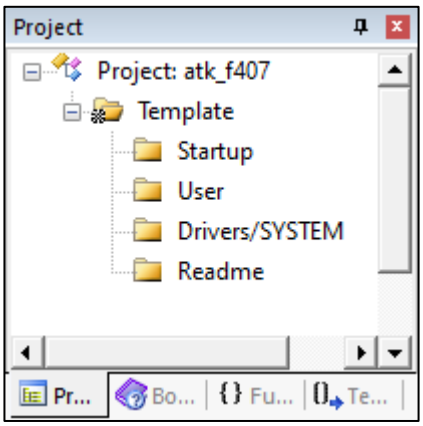


图 6.1.3.3 设置成功

这里我们只是新建了一个简单的工程，并没有添加 BSP、Middlewares 等分组，后面随着工程复杂程度的增加，我们需要一步步添加对应的分组。

注意：为了让工程结构清晰，我们会尽量让 MDK 的工程分组和我们前面新建的工程文件夹对应起来，由于 MDK 分组不支持多级目录，因此我们将路径也带入分组命名里面，以便区分。如：User 分组对应 User 文件夹里面的源码，Drivers/SYSTEM 分组，对应 Drivers/SYSTEM 文件夹里面的源码，Drivers/BSP 分组对应 Drivers/BSP 文件夹里面的源码等。

2. 添加启动文件

启动文件（.s 文件）包含 STM32 的启动代码，其主要作用包括：1、堆栈（SP）的初始化；2、初始化程序计数器（PC）；3、设置向量表异常事件的入口地址；4、调用 main 函数等，是每个工程必不可少的一个文件，我们在本书第九章会有详细介绍。

该文件由 ST 官方提供，对于 STM32F407 来说只有一个启动文件：startup_stm32f407xx.s。

启动文件存放在 STM32CubeF4 软件包的：Drivers→CMSIS→Device→ST→STM32F4xx→Source→Templates→arm 文件夹下。为了节省空间，在精简版 CMSIS 文件夹里面我们把其他启动文件都删了。而且，为了更好的匹配寄存器版本代码，我们对 startup_stm32f407xx.s 做了 2 处修改：

1，我们用不到编译器的内存管理函数，为节省内存，将 Heap_Size 改成 0，源码如下：

```
;未用到编译器自带的内存管理(malloc,free 等)，设置 Heap_Szie 为 0
Heap_Size       EQU     0x00000000
```

2，寄存器代码不需要调用 SystemInit 函数，因此修改 Reset_Handler 函数，去掉 SystemInit 调用，源码如下：

```
Reset_Handler    PROC
                 EXPORT  Reset_Handler             [WEAK]
                 IMPORT  __main

                 LDR     R0, =0xE000ED88    ;使能浮点运算 CP10,CP11
                 LDR     R1,[R0]
                 ORR     R1,R1,#(0xF << 20)
                 STR     R1,[R0]

                 ;寄存器版本代码，因为没有用到 SystemInit 函数，所以注释掉以下代码为防止报错！
                 ;HAL 库版本代码，建议加上这里（外部必须实现 SystemInit 函数），以初始化
                 ;stm32 时钟等.
                 ;IMPORT  SystemInit
                 ;LDR     R0, =SystemInit
                 ;BLX     R0

                 LDR     R0, =__main
                 BX      R0
                 ENDP
```

这段代码，我们主要加入了开启 STM32F407 硬件 FPU 的代码，以使能 STM32F407 的浮点运算单元，并关闭硬件自动压栈。其中，0xE000ED88 就是协处理器控制寄存器(CPACR)的地址，该寄存器的第 20~23 位用来控制是否支持浮点运算，这里我们全设置为 1，以支持硬件浮点运算。关于 CPACR 寄存器的详细描述，见《STM32F3 与 F4 内核编程手册.pdf》第 4.6.1 节。另外，寄存器版本，我们还屏蔽了 SystemInit 函数的调用，如果是 HAL 库版本，可以取消这个函数的注释，并在外部实现 SystemInit 函数。

特别注意：我们在汇编代码里面使能了 FPU，所以在 MDK 里面，我们也要设置使用 FPU，否则可能代码会无法运行，设置方法如下：选择 Options for Target ‘Target1’，打开 Target 选项卡，在 Code Generation 里面，选择 Single Precision，如图 6.1.3.4 所示：

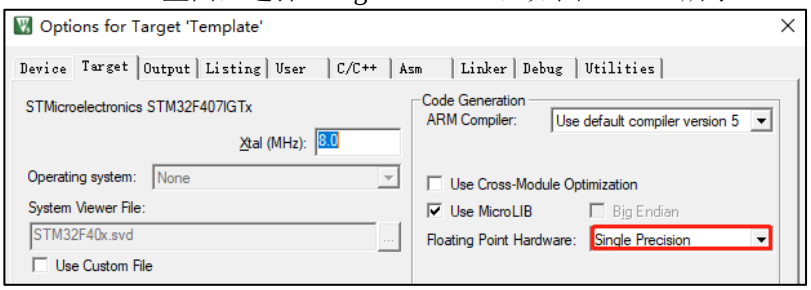


图 6.1.3.4 MDK 开启浮点运算

关于启动文件的说明，我们就介绍这么多，接下来我们看如何添加启动文件到工程里面。我们有两种方法给 MDK 的分组添加文件：1，双击 Project 下的分组名添加。2，进入工程管理界面添加。

这里我们使用方法 1 添加（路径：实验 0-2，新建工程实验-寄存器版本\Drivers\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm），如图 6.1.3.5 所示：



图 6.1.3.5 双击分组添加启动文件（startup_stm32f407xx.s）

上图中，我们也可以点击 Add 按钮进行文件添加。添加完后，点击 Close，完成启动文件添加，得到工程分组如图 6.1.3.6 所示：

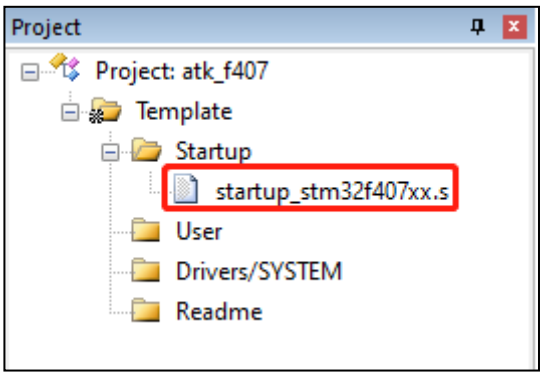


图 6.1.3.6 启动文件添加成功

3. 添加 SYSTEM 源码

这里我们在工程管理界面（方法 2）进行 SYSTEM 源码添加。点击：按钮，进入工程管理界面，选中 Drivers/SYSTEM 分组，然后点击：Add Files，进入文件添加对话框，依次添加 delay.c、sys.c 和 usart.c 到该分组下，如图 6.1.3.6 所示：

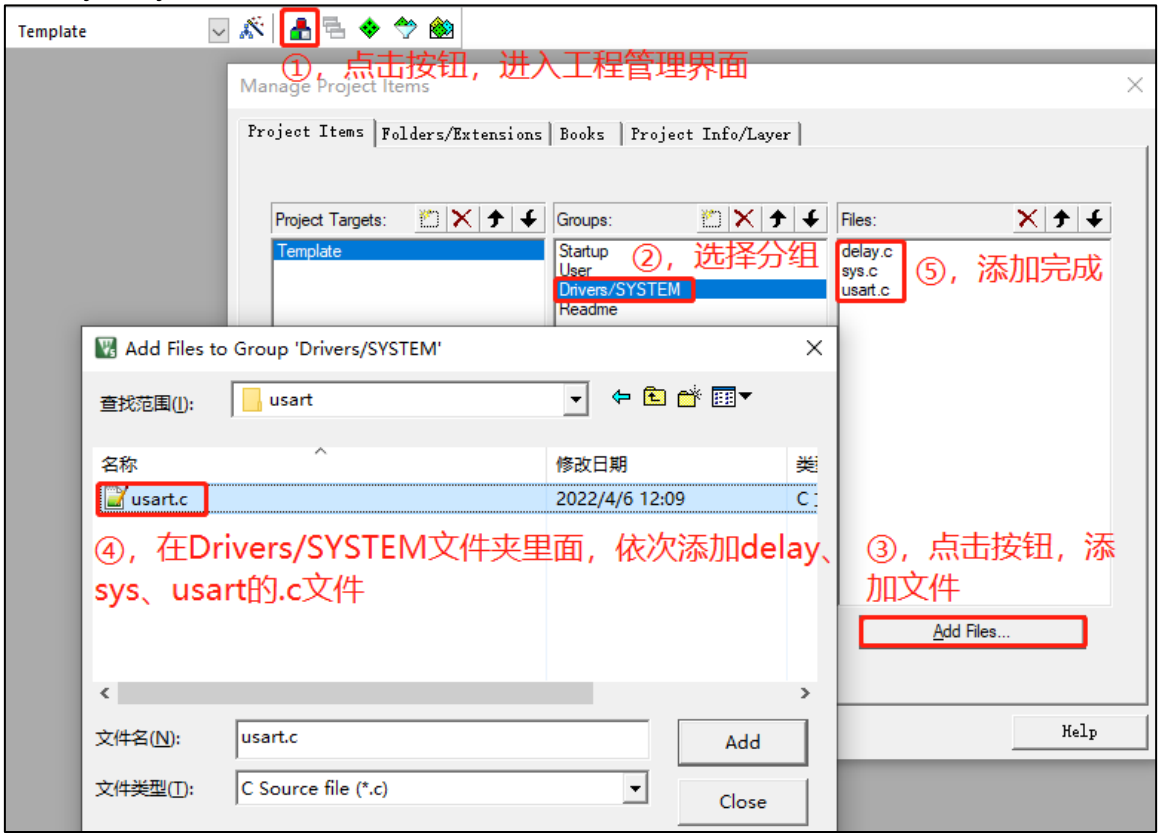


图 6.1.3.6 添加 SYSTEM 源码

注意：这些源码都是在第 6.1.1 小节的第二步拷贝过来的，如果之前没拷贝，是找不到这些源码的。添加完成后，如图 6.1.3.7 所示：

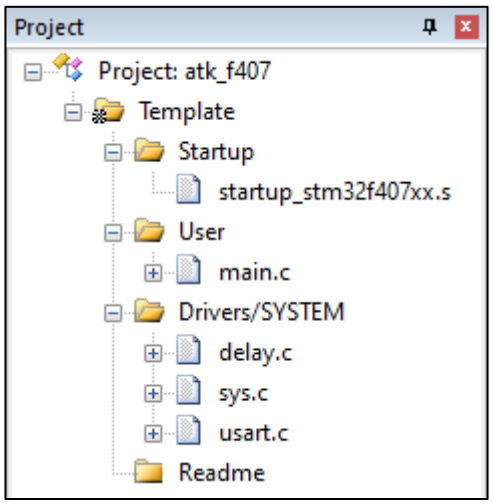


图 6.1.3.7 SYSTEM 源码添加完成

6.1.4 魔术棒设置

为避免编写代码和编译报错，我们需要通过魔术棒对 MDK 工程进行相关设置。在 MDK 主界面，点击：（魔术棒图标，即 Options for Target 按钮），进入工程设置对话框，我们将进行如下几个选项卡的设置。

1. 设置 Target 选项卡

在魔术棒→Target 选项卡里面，我们进行如图 6.1.4.1 所示设置：

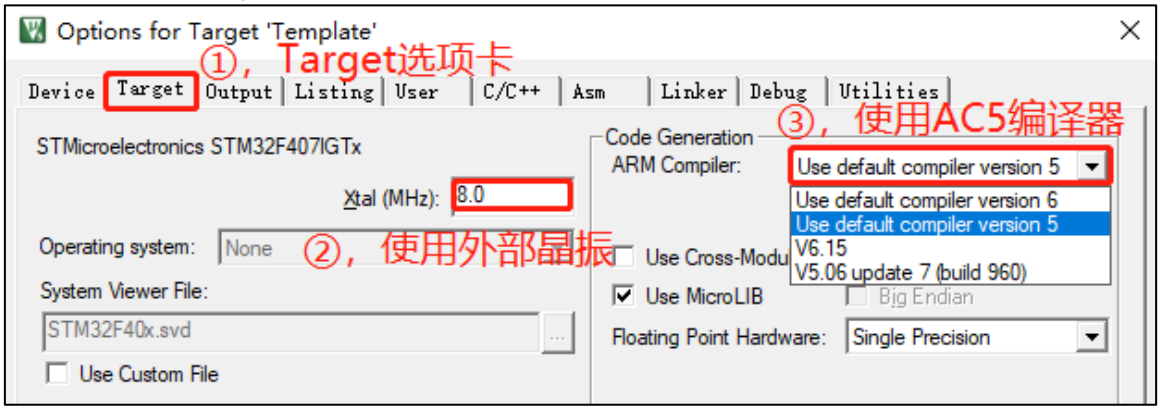


图 6.1.4.1 Target 选项卡设置

上图中，我们设置芯片所使用的外部晶振频率为 8Mhz，选择 ARM Compiler 版本为：Use default compiler version 5（即 AC5 编译器）。

这里我们说明一下 AC5 和 AC6 编译的差异，如表 6.1.4.1 所示：

对比项	AC5	AC6	说明
中文支持	较好	较差	AC6 对中文支持极差，goto definition 无法使用，误报等
代码兼容性	较好	较差	AC6 对某些代码优化可能导致运行异常，需慢慢调试
编译速度	较慢	较快	AC6 编译速度比 AC5 快
语法检查	一般	严格	AC6 语法检查非常严格，代码严谨性较好

表 6.1.4.1 AC5&AC6 简单对比

由于 AC5 对中文支持比较好，且兼容性相对好一点，**为了避免不必要的麻烦，我们推荐大家使用 AC5 编译器**。为了让大家自由选择，我们正点原子的源码，绝大部分也是支持 AC6 编译器的，不过在选项卡设置上稍有差异，具体差异如表 6.1.4.2 所示：

选项卡	AC5	AC6	说明
Target	选择 AC5 编译器	选择 AC6 编译器	选择对应的编译器

C/C++	Misc Controls 无需设置	Misc Controls 设置： -Wno-invalid-source-encoding	AC6 需设置编译选项以关闭对汉字的错误警告，AC5 则不要

表 6.1.4.2 AC5&AC6 设置差异

2. 设置 Output 选项卡

在魔术棒→Output 选项卡里面，进行如图 6.1.4.2 所示设置：

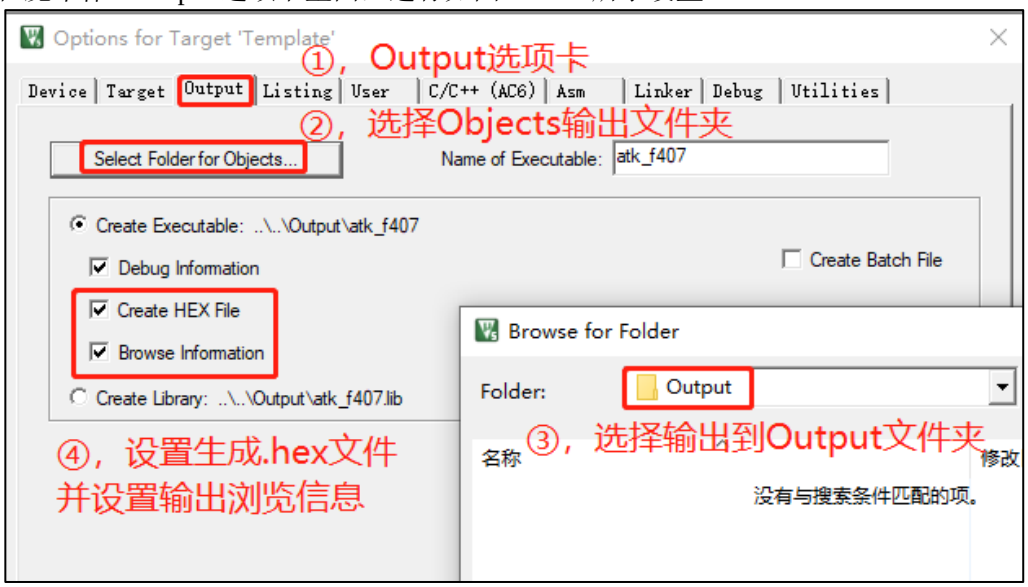


图 6.1.4.2 设置 Output 选项卡

注意，我们勾选：Browse Information，用于输出浏览信息，这样就可以使用 go to definition 查看函数/变量的定义，对我们后续调试代码比较有帮助，如果不需要调试代码，则可以去掉这个勾选，以提高编译速度。

3. 设置 Listing 选项卡

在魔术棒→Listing 选项卡里面，进行如图 6.1.4.3 所示设置：

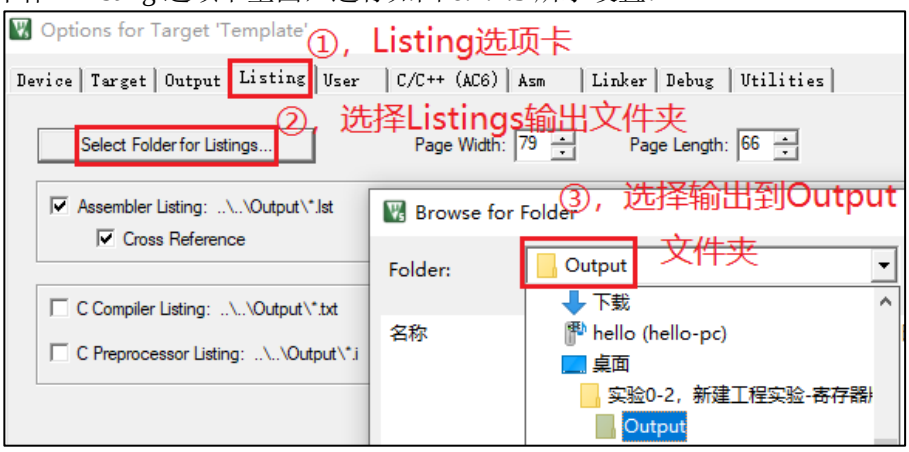


图 6.1.4.3 设置 Listing 选项卡

经过 Output 和 Listing 这两步设置，原来存储在 Objects 和 Listings 文件夹的内容（中间文件）就都改为输出到 Output 文件夹了。

4. 设置 C/C++选项卡

在魔术棒→C/C++选项卡里面，进行如图 6.1.4.4 所示设置：

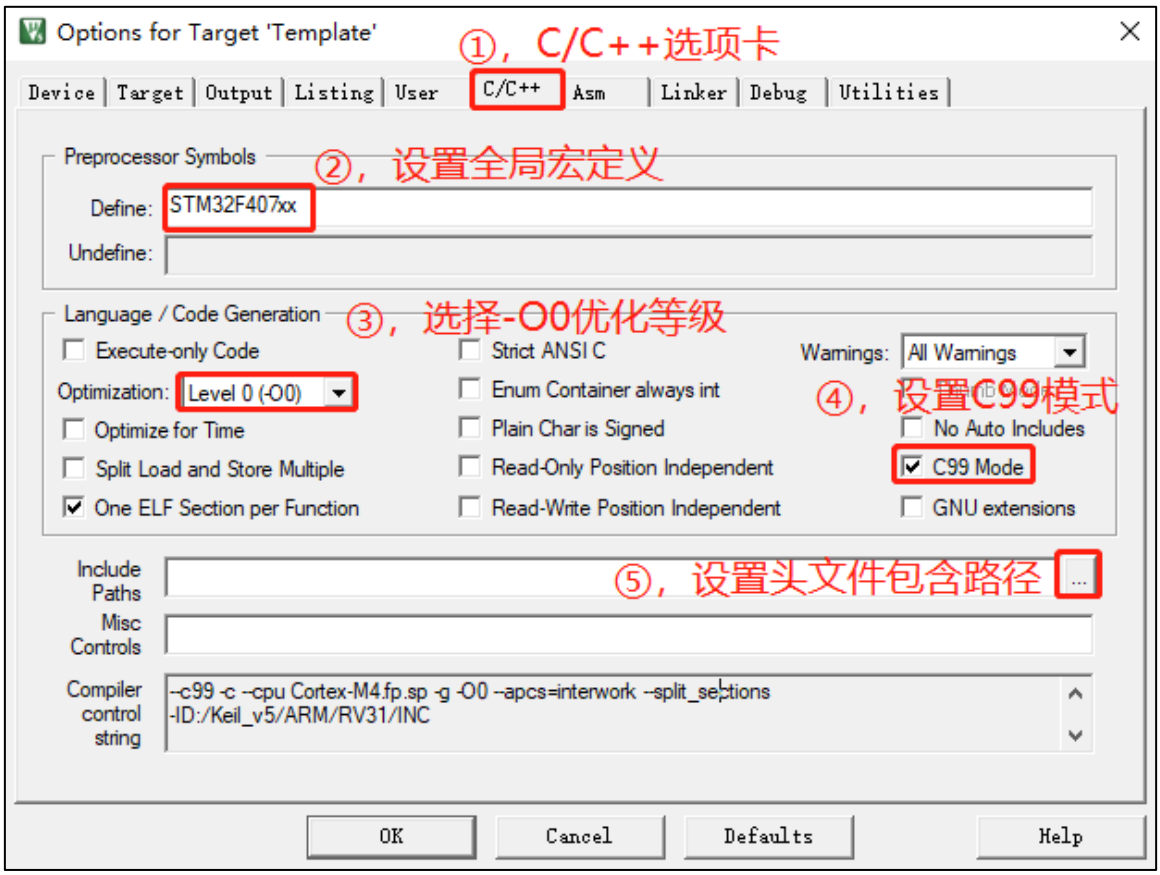


图 6.1.4.4 设置 C/C++选项卡

在②处设置了全局宏定义：STM32F407xx，用于定义所用 STM32 型号，在 stm32f4xx.h 里面会用到该宏定义。

在③处设置了优化等级为-O0，可以得到最好的调试效果，当然为了提高优化效果提升性能并降低代码量，可以设置-O1~-O3，数字越大效果越明显，不过也越容易出问题。注意：当使用 AC6 编译器的时候，这里推荐默认使用-O1 优化。

在④处勾选 C99 模式，即使用 C99 C 语言标准。

在⑤处，我们可以进行头文件包含路径设置，点击此按钮，进行如图 6.1.4.5 所示设置：

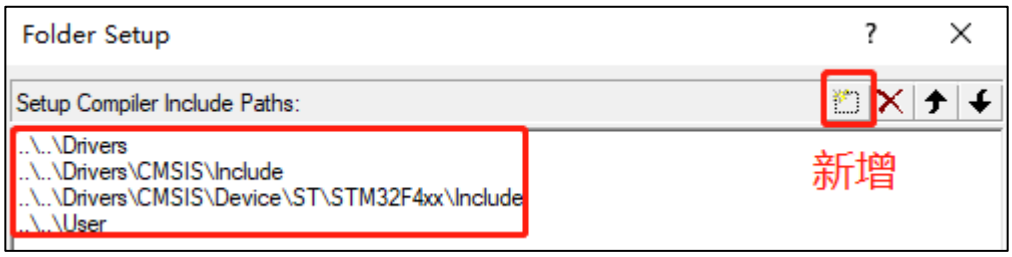


图 6.1.4.5 设置头文件包含路径

上图中我们设置了 4 个头文件包含路径，其中 3 个在 Drivers 文件夹下，一个在 User 文件夹下。为避免频繁设置头文件包含路径，正点原子最新源码的 include 全部使用相对路径，也就是我们只需要在头文件包含路径里面指定一个文件夹，那么该文件夹下的其他文件夹里面的源码，如果全部是使用相对路径，则无需再设置头文件包含路径了，直接在 include 里面就指明了头文件所在。

关于相对路径，这里大家记住 3 点：

1，默认路径就是指 MDK 工程所在的路径，即.uvprojx 文件所在路径（文件夹）

2，“./”表示当前目录（相对当前路径，也可以写做“.\”）

3，“../”表示当前目录的上一层目录（也可以写做“..\”）

举例来说，上图中：..\..\Drivers\CMSIS\Device\ST\STM32F4xx\Include，前面两个“..\”，

表示 Drivers 文件夹在当前 MDK 工程所在文件夹（MDK-ARM）的上 2 级目录下，具体解释如图 6.1.4.6 所示：

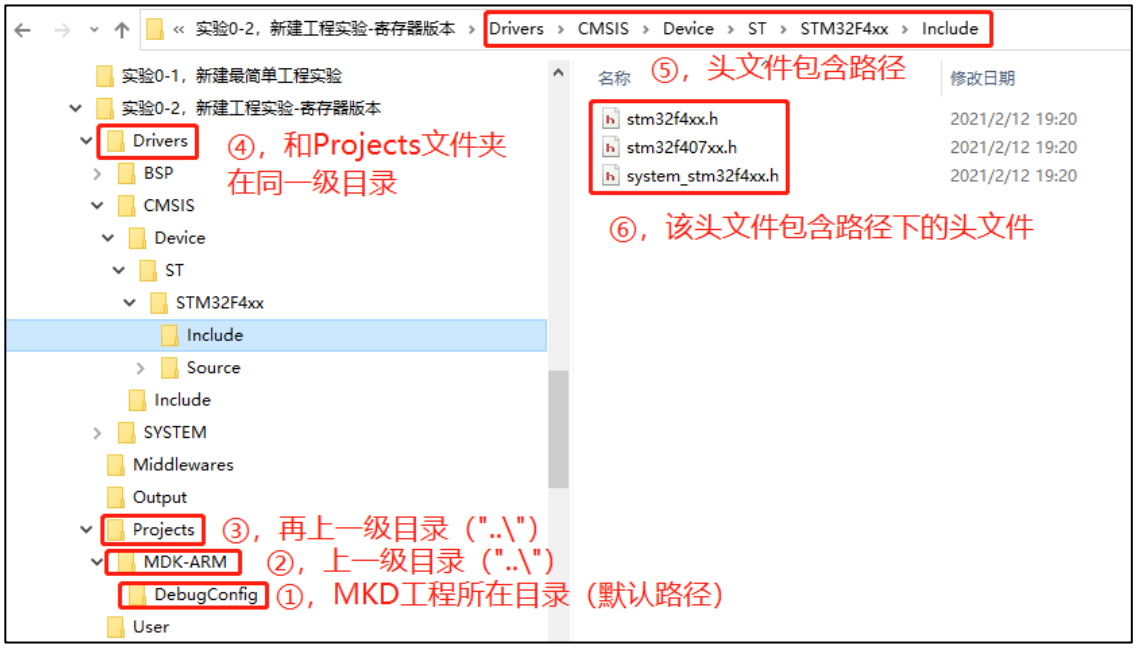


图 6.1.4.6 ..\..\Drivers\CMSIS\Device\ST\STM32F4xx\Include 的解释

上图表示根据头文件包含路径：..\..\Drivers\CMSIS\Device\ST\STM32F4xx\Include，编译器可以找到⑥处所包含的这些头文件，即代码里面可以直接 include 这些头文件使用。

再举个例子，在完成如图 6.1.4.5 所示的头文件包含路径设置以后，我们在代码里面编写：

```
#include "./SYSTEM/sys/sys.h"
```

即表示当前头文件包含路径所指示的 4 个文件夹里面，肯定有某一个文件夹包含了：SYSTEM/sys/sys.h 的路径，实际上就是在 Drivers 文件夹下面，两者结合起来就相当于：

```
#include "../../Drivers/SYSTEM/sys/sys.h"
```

这就是相对路径。它既可以减少头文件包含路径设置（即减少 MDK 配置步骤，免去频繁设置头文件包含路径的麻烦），同时又可以很方便的知道头文件具体在那个文件夹，因此我们推荐在编写代码的时候使用相对路径。

关于相对路径，我们就介绍这么多，大家搞不明白的可以在网上搜索相关资料学习，也可以在后面的学习，分析我们其他源码，慢慢体会，总之不难，但是好用。

最后，我们如果使用 AC6 编译器，则在图 6.1.4.4 的 Misc Controls 处需要设置：-Wno-invalid-source-encoding，避免中文编码报错，如果使用 AC5 编译器，则不需要该设置！！

5. 设置 Debug 选项卡

在魔术棒→Debug 选项卡里面，进行如图 6.1.4.7 所示设置：

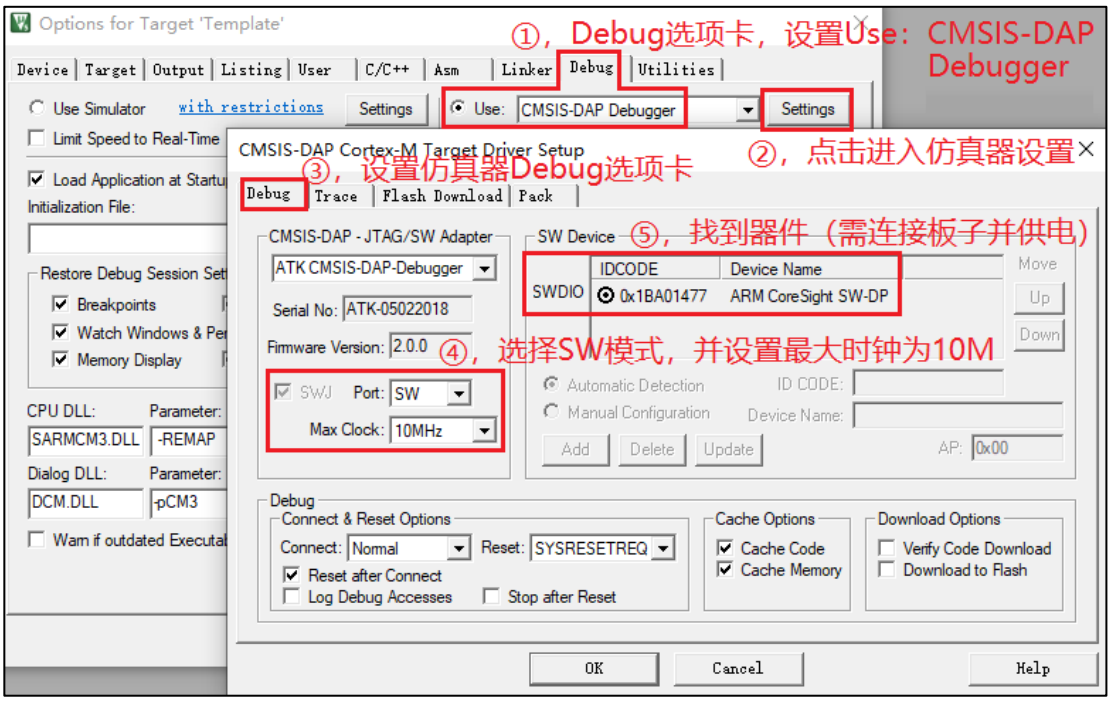


图 6.1.4.7 Debug 选项卡设置

图中，我们选择使用：CMSIS-DAP 仿真器，使用 SW 模式，并设置最大时钟频率为 10Mhz，以得到最高下载速度。当我们将仿真器和开发板连接好，并给开发板供电以后，仿真器就会找到开发板芯片，并在 SW Device 窗口显示芯片的 IDCODE、Device Name 等信息（图中⑤处），当无法找到时，请检查供电和仿真器连接状况。

6. 设置 Utilities 选项卡

在魔术棒→Debug 选项卡里面，进行如图 6.1.4.8 所示设置：

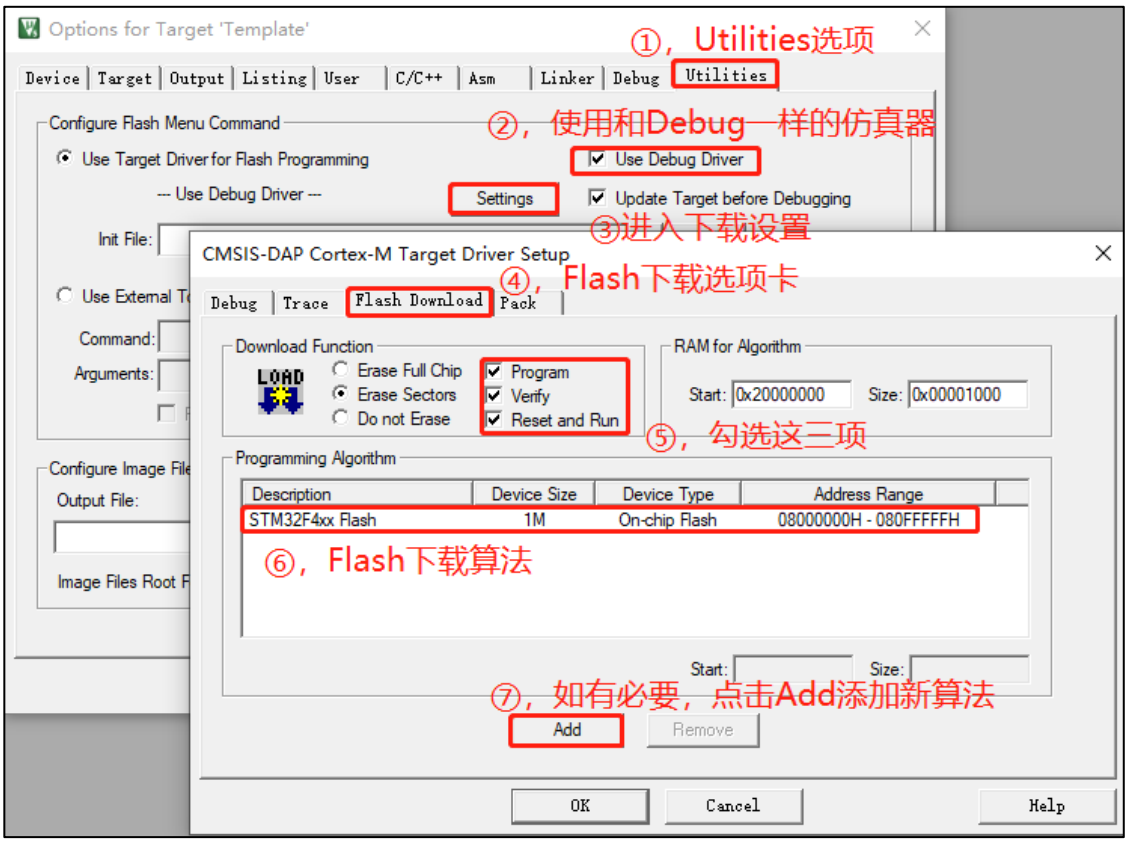


图 6.1.4.8 Utilities 选项卡设置

图中⑥处下载算法，是 MDK 默认添加的，针对 STM32F4xx 大容量系列产品（FLASH 容量为 1M）。一般我们用这个即可。如果⑥处没有下载算法，则点击⑦处按钮，执行添加一下下载算法即可（名字和⑥处的算法名字一样）。

6.1.5 添加 main.c，并编写代码

在 MDK 主界面，点击：，新建一个 main.c 文件，并保存在 User 文件夹下。然后双击 User 分组，弹出添加文件的对话框，将 User 文件夹下的 main.c 文件添加到 User 分组下。得到如图 6.1.5.1 所示的界面：

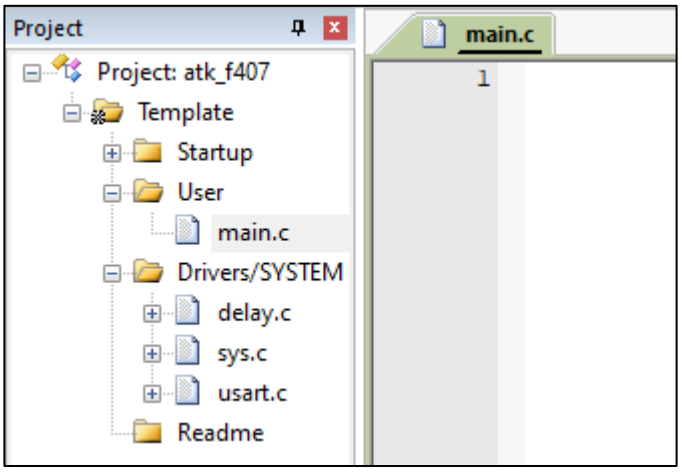


图 6.1.5.1 在 User 分组下加入 main.c 文件

至此，我们就可以开始编写我们自己的代码了。我们在 main.c 文件里面输入如下代码：

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"
```

```
int main(void)
{
    uint8_t t = 0;

    sys_stm32_clock_init(336, 8, 2, 7);      /* 设置时钟,168Mhz */
    delay_init(168);                         /* 延时初始化 */
    usart_init(84, 115200);                  /* 串口初始化为 115200 */

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

此部分代码，在 A 盘→4，程序源码→1，标准例程-寄存器版本→ 实验 0 基础入门实验→实验 0-2，新建最工程实验-寄存器版本→User→main.c 里面有，大家可以自己输入，也可以直接拷贝。强烈建议自己输入，以加深对程序的理解和印象!!

注意，这里的 include 就是使用的相对路径，关于相对路径，请参考前面 C/C++选项卡设置章节进行学习。

输入完代码，如图 6.1.5.2 所示

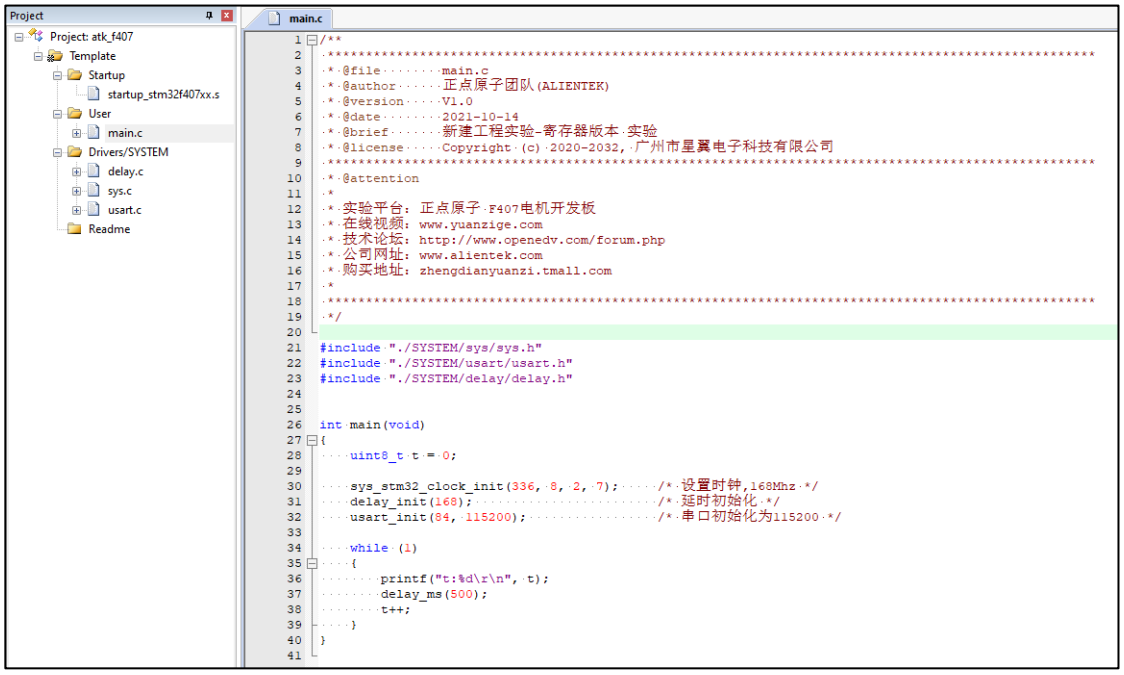


图 6.1.5.2 main.c 代码编写完成

编写完 main.c 以后，我们点击：（Rebuild）按钮，编译整个工程，编译结果如图 6.1.5.3 所示：

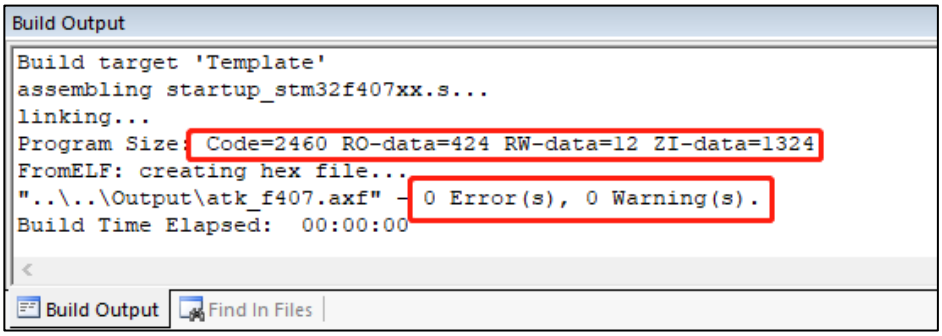


图 6.1.5.3 编译结果

编译结果提示：代码总大小（Porgram Size）为：FLASH 占用 2896 字节（Code＋RO＋RW），SRAM 占用 1326 字节（RW＋ZI）；并成功创建了 Hex 文件（可执行文件，放在 Output 目录下）；编译 0 错误，0 警告。

注意：如果编译提示有错误/警告，请根据提示，从第一个错误/警告开始解决，直到 0 错误 0 警告。如果出错，很有可能是之前的操作存在问题，请对照教程找问题。

另外，我们在 Readme 分组下还没有添加任何文件，由于只是添加一个说明性质的文件（.txt），并不是工程必备文件，因此这里我们就不添加了，开发板光盘的源码我们是有添加的，大家可以去参考一下。

至此，新建寄存器版本 MDK 工程完成。

6.2 下载验证

有两种方法可以给 STM32F407 芯片下载代码：1，使用串口下载；2，使用仿真器下载。这两种下载方法，我们在本书的 4.2 节给大家有做过详细介绍。这里我们以仿真器下载为例，在 MDK 主界面，点击：  （下载按钮，也可以按键盘快捷键：F8），就可以将代码下载到开发板，如图 6.2.1 所示：

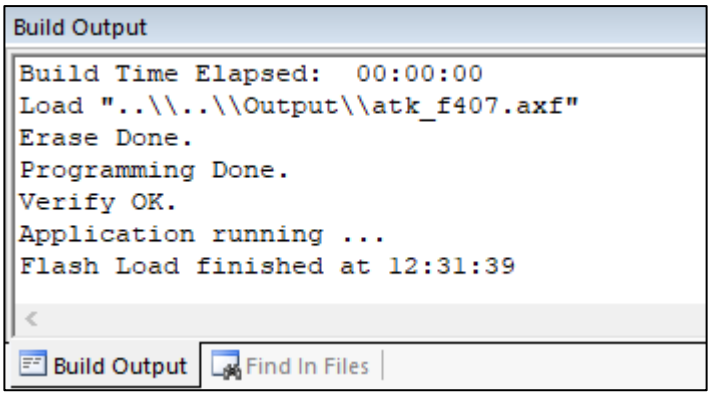


图 6.2.1 下载成功

上图提示：Application running…，则表示代码下载成功，且开始运行。此时，我们打开串口调试助手，并设置好端口号（COM 号）和波特率（115200），就可以看到打印出来的 t 值，如图 6.2.2 所示：

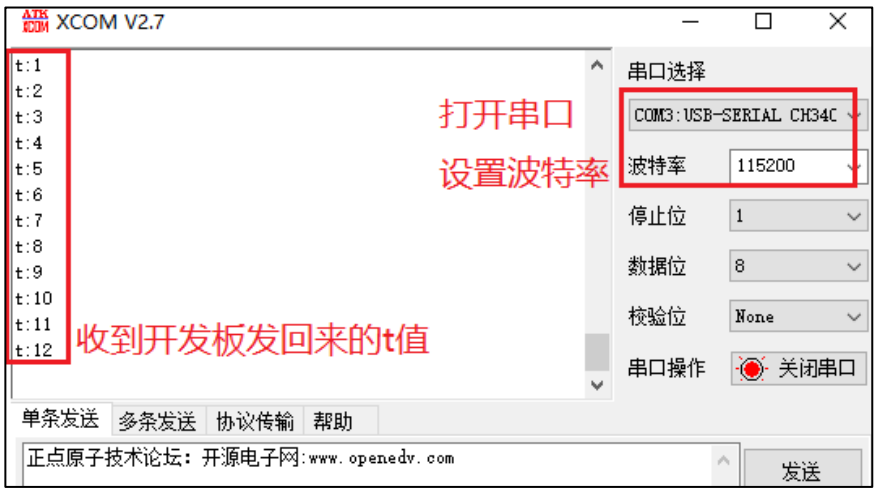


图 6.2.2 收到开发板发回来的数据

说明我们的程序运行正常，下载验证无误。

第七章 认识 HAL 库

HAL，英文全称 Hardware Abstraction Layer，即硬件抽象层。HAL 库是 ST 公司提供的外设驱动代码的驱动库，用户只需要调用库的 API 函数，便可间接配置寄存器。我们要写程序控制 STM32 芯片，其实最终就是控制它的寄存器，HAL 库就为了更方便我们去控制寄存器，从而节约开发时间。

本章将分为如下几个小节：

7.1 初识 STM32 HAL 库
7.2 HAL 库驱动包
7.3 HAL 库框架结构
7.4 如何使用 HAL 库
7.5 HAL 库使用注意事项

7.1 初识 STM32 HAL 库

STM32 开发中常说的 HAL 库开发，指的是利用 HAL 库固件包里封装好的 c 语言编写的驱动文件，来实现对 STM32 内部和外围电器元件的控制的过程。但只有 HAL 库还不能直接驱动一个 STM32 的芯片，其它的组件已经由 ARM 与众多芯片硬件、软件厂商制定的通用的软件开发标准 CMSIS 实现了，本文只简单介绍这个标准，等大家熟悉开发后再研究这个框架。

简单地了解 HAL 库的发展和作用，可以方便学习者确定 HAL 库是否适合作为学习者自己长期开发 STM32 的工具，以降低开发、学习的成本。

7.1.1 CMSIS 标准

根据一些调查研究表明，软件开发已经被嵌入式行业公认为最主要的开发成本,为了降低这个成本，ARM 与 Atmel、IAR、KEIL、SEGGER 和 ST 等诸多芯片和软件工具厂商合作，制定了一个将所有 Cortex 芯片厂商的产品的软件接口标准化的标准 CMSIS（Cortex Microcon troller Software Interface Standard）。下面来看 ARM 官方提供的 CMSIS 规范架构图，如图 7.1.1.1 所示：

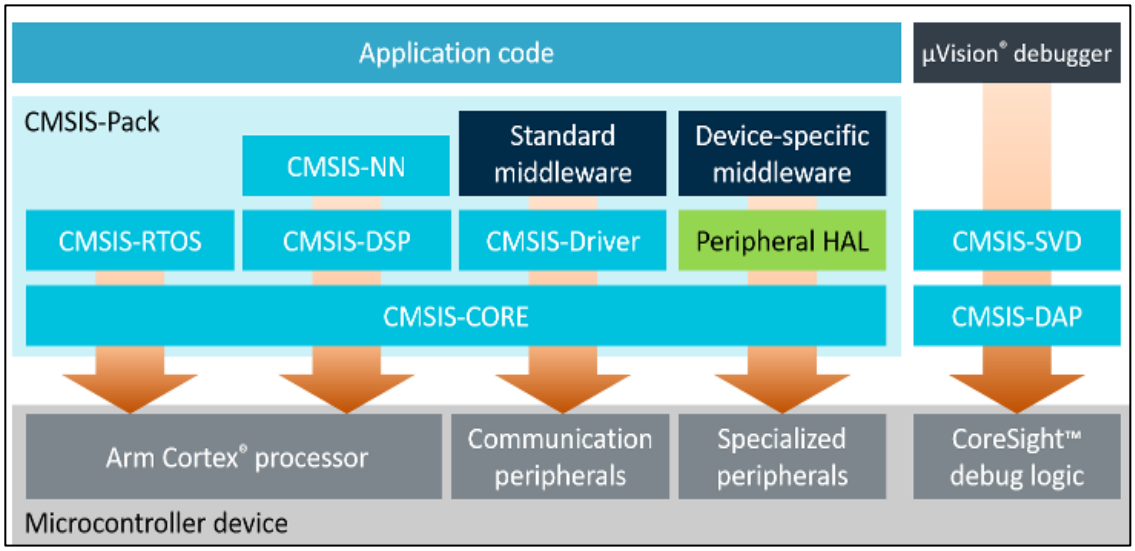


图 7.1.1.1 CorteX 芯片的 CMSIS 分级实现

从图中可以看出这个标准分级明显，从用户程序到内核底层实现做了分层。按照这个分级，HAL 库属于 CMSIS-Pack 中的“Peripheral HAL”层。CMSIS 规定的最主要的 3 个部分为：核内外设访问层（由 ARM 负责实现）、片上外设访问层和外设访问函数（后面两个由芯片厂商负责实现）。ARM 整合并提供了大量的模版，各厂商根据自己的芯片差异修改模版，这其中包括

汇编文件 startup_device.s、system_.h 和 system_.c 这些与初始化和系统相关的函数。

结合 STM32F407 的芯片来说，其 CMSIS 应用程序的简单结构框图，不包括实时操作系统和中间设备等组件，其结构如图 7.1.1.2 所示。

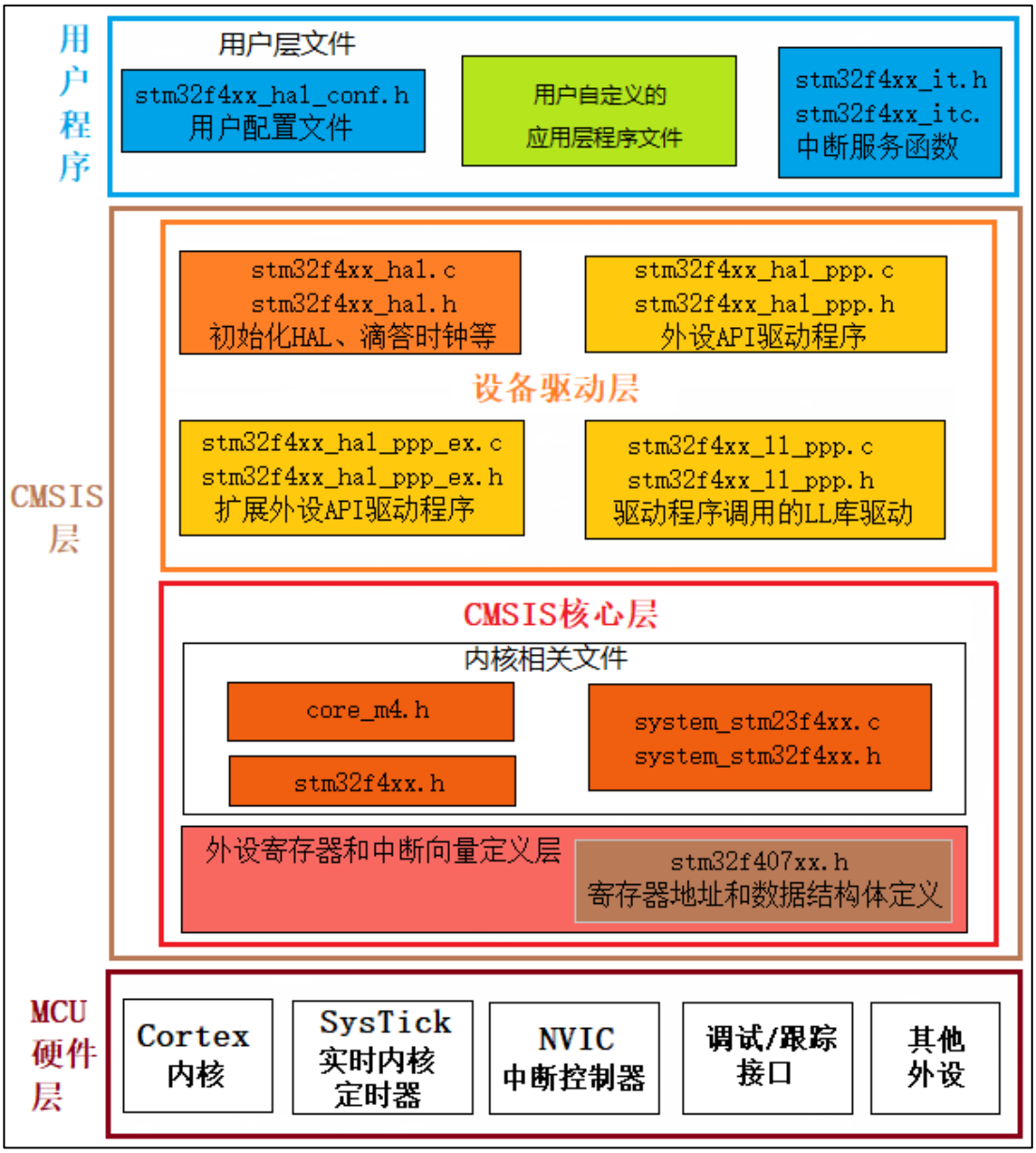


图 7.1.1.2 CMSIS 分级下的 STM32F4 的文件分布

上面的框架是根据我们现在已经学习到的知识回过头来作的一个总结，这里只是作简单介绍，告诉大家它们之间存在一定联系，关于组成这些部分的文件、文件的作用及各文件如何组合、各分层的作用和意义，我们会在今后的学习过程中慢慢学习。

7.1.2 HAL 库简介

库函数的引入，大大降低 STM 系列主控芯片开发的难度。ST 公司为了方便用户开发 STM32 芯片开发提供了三种库函数，从时间产生顺序是：标准库、HAL 库和 LL 库。目前 ST 已经逐渐暂停对部分标准库的支持，ST 的库函数维护重点对角已经转移到 HAL 库和 LL 库上，下面我们分别为这三种库作一下简单的介绍。

1. 标准外设库（Standard Peripheral Libraries）

标准外设库（Standard Peripherals Library）是对 STM32 芯片的一个完整的封装，包括所有标准器件外设的器件驱动器，是 ST 最早推出的针对 STM 系列主控的库函数。标准库的设计的

初衷是减少用户的程序编写时间，进而降低开发成本。几乎全部使用 C 语言实现并严格按照“Strict ANSI-C”、MISRA-C 2004 等多个 C 语言标准编写。但标准外设库仍然接近于寄存器操作，主要就是将一些基本的寄存器操作封装成了 C 函数。开发者仍需要关注所使用的外设是在哪个总线之上，具体寄存器的配置等底层信息。

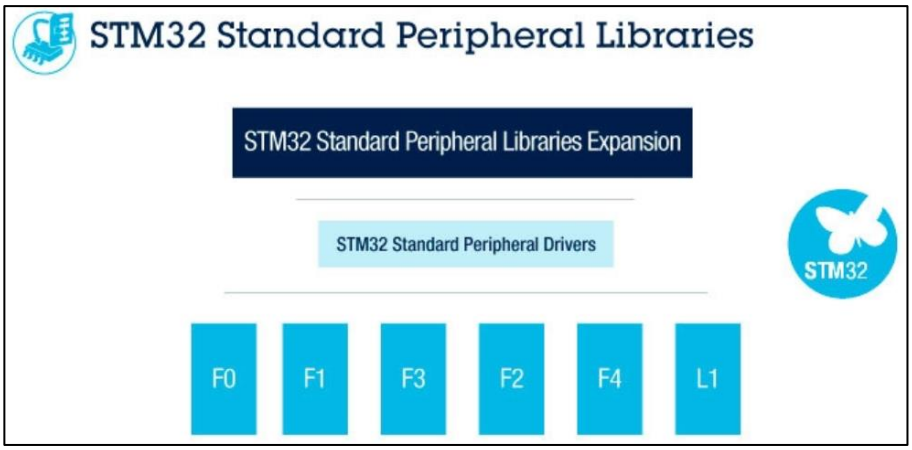


图 7.1.2.1 ST 的标准库函数家族

ST 为各系列提供的标准外设库稍微有些区别。例如，STM32F1x 的库和 STM32F3x 的库在文件结构上就有些不同，此外，在内部的实现上也稍微有些区别，这个在具体使用（移植）时，需要注意一下！但是，不同系列之间的差别并不是很大，而且在设计上是相同的。

STM32 的标准外设库涵盖以下 3 个抽象级别：

- 包含位域和寄存器在内的完整的寄存器地址映射
- 涵盖所有外围功能（具有公共 API 的驱动器）的例程和数据结构的集合。
- 一组包含所有可用外设的示例，其中包含最常用的开发工具的模板项目。

关于更详细的信息，可以参考 ST 的官方文档《STM32 固件库使用手册中文翻译版》，文档中对于标准外设库函数命名、文件结构等都有详细的说明，这里我们就不多介绍了。

值得一提的是由于 STM32 的产品性能及标准库代码的规范和易读性以及例程的全覆盖性，使 STM32 的开发难度大大下降。但 ST 从 L1 以后的芯片 L0、L4 和 F7 等系列就没有再推出相应的标准库支持包了。

2.HAL 库

HAL 是 Hardware Abstraction Layer 的缩写，即硬件抽象层。是 ST 为可以更好的确保跨 STM32 产品的最大可移植性而推出的 MCU 操作库。这种程序设计由于抽离应用程序和硬件底层的操作，更加符合跨平台和多人协作开发的需要。

HAL 库是基于一个非限制性的 BSD 许可协议（Berkeley Software Distribution）而发布的开源代码。ST 制作的中间件堆栈（USB 主机和设备库，STemWin）带有允许轻松重用的许可模式，只要是在 ST 公司的 MCU 芯片上使用，库中的中间件(USB 主机/设备库，STemWin)协议栈即被允许修改，并可以反复使用。至于基于其它著名的开源解决方案商的中间件（FreeRTOS，FatFs，LwIP 和 PolarSSL）也都具有友好的用户许可条款。

HAL 库是从 ST 公司从自身芯片的整个生产生态出发，为了方便维护而作的一次整合，以改变标准外设库带来各系列芯片操作函数结构差异大、分化大、不利于跨系列移植的情况。相比标准外设库，STM32Cube HAL 库表现出更高的抽象整合水平，HAL 库的 API 集中关注各外设的公共函数功能，这样便于定义一套通用的用户友好的 API 函数接口，从而可以轻松实现从一个 STM32 产品移植到另一个不同的 STM32 系列产品。但由于封闭函数为了适应最大的兼容性，HAL 库的一些代码实际上的执行效率要远低于寄存器操作。但即便如此，HAL 库仍是 ST 未来主推的库。

3.LL 库

LL 库（Low Layer）目前与 HAL 库捆绑发布，它的设计为比 HAL 库更接近于硬件底层的操作，代码更轻量级，代码执行效率更高的库函数组件，可以完全独立于 HAL 库来使用，但 LL

库不匹配复杂的外设，如 USB 等。所以 LL 库并不是每个外设都有对应的完整驱动配置程序。使用 LL 库需要对芯片的功能有一定的认知和了解，它可以：

- 独立使用，该库完全独立实现，可以完全抛开 HAL 库，只用 LL 库编程完成。
- 混合使用，和 HAL 库结合使用。

对于 HAL 库和 LL 库的关系，如图 7.1.2.2 Cube 的软件框架所示，可以看出它们设计为彼此独立的分支，但又同属于 HAL 库体系。

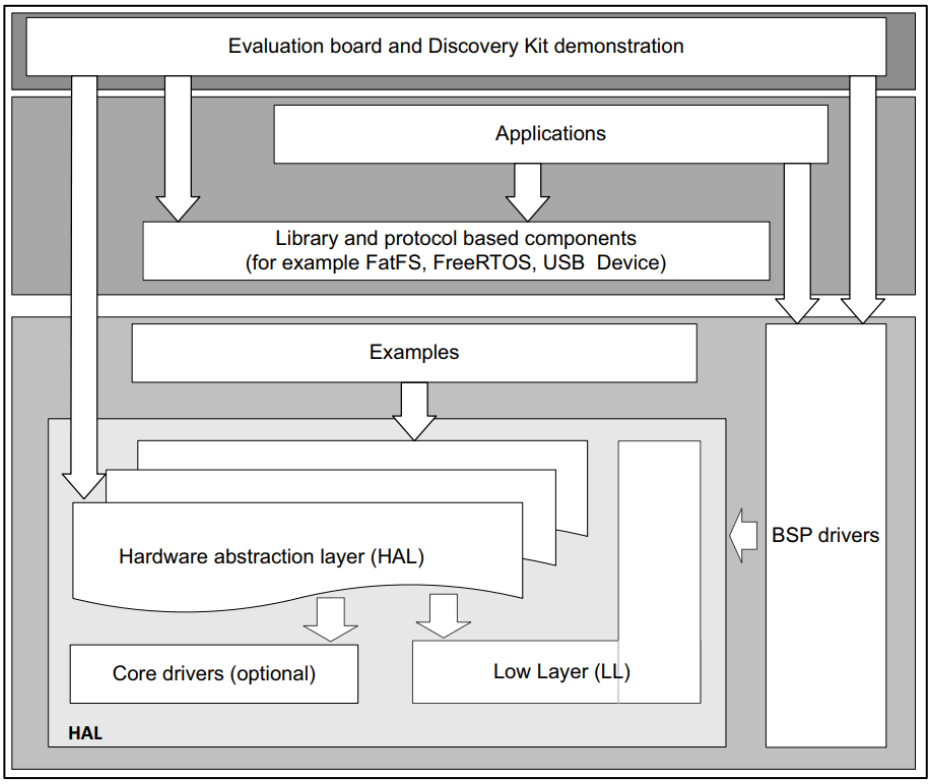


图 7.1.2.2 Cube 的软件框架

通过以上简介我们对目前主流的 STM32 开发库有了一个初步的印象。标准库和 HAL 库、LL 库完全相互独立，HAL 库更倾向于外设通用化，扩展组件中解决芯片差异操作部分；LL 倾向于最简单的寄存器操作，ST 在未来还将重点维护和建设 HAL 库，标准库已经部分停止更新。HAL 库和 LL 库的应用将是未来的一个趋势。

7.1.3 HAL 库能做什么

用过标准库的朋友应该知道，使用标准库可以忽略很多芯片寄存器的细节，根据提供的接口函数快速配置和使用一个 STM32 芯片，使用 HAL 库也是如此。不论何种库，本质都是配置指定寄存器使芯片工作在我们需要的工作模式下。HAL 库在设计的时候会更注重软硬件分离。HAL 库的 API 集中关注各个外设的公共函数功能，便于定义通用性更好、更友好的 API 函数接口，从而具有更好的可移植性。HAL 库写的代码在不同的 STM32 产品上移植，非常方便。

我们需要学会调用 HAL 库的 API 函数，配置对应外设按照我们的要求工作，这就是 HAL 库能做的事。但是无论库封装得多高级，最终还是要通过配置寄存器来实现。所以我们学习 HAL 库的同时，也建议同时学习外设的工作原理和寄存器的配置。只有掌握了原理，才能更好的使用 HAL 库，一旦发生问题也能更快速了定位和解决问题。

HAL 库还可以和 STM32CubeMX（图形化软件配置工具）配套一起使用，开发者可以使用该工具进行可视化配置，并且自动生成配置好的初始化代码，大大的节省开发时间。

7.2 HAL 库驱动包

HAL 库是一系列封装好的驱动函数，本节将从下载渠道、固件包的内容分析及在实际开发中用到的几个文件的详细介绍。

7.2.1 如何获取 HAL 库固件包

HAL 库是 ST 推出的 STM32Cube 软件生态下的一个分支。STM32Cube 是 ST 公司提供的一套免费开发工具和 STM32Cube 固件包，旨在通过减少开发工作、时间和成本来简化开发人员的工作，并且覆盖整个 STM32 产品。它包含两个关键部分：

1、允许用户通过图形化向导来生成 C 语言工程的图形配置工具 STM32CubeMX。可以通过 CubeMX 实现方便地下载各种软件或开发固件包。

2、包括由 STM32Cube 硬件抽象层（HAL），还有一组一致的中间件组件（RTOS、USB、FAT 文件系统、图形、TCP/IP 和以太网），以及一系列完整的例程组成的 STM32Cube 固件包。

ST 提供了多种获取固件包的方法。本节只介绍从 ST 官方网站上直接获取固件库的方法。网页登陆：www.st.com，在打开的页面中依次选择：“Tools & Software”-> “Ecosystem”-> “STM32Cube”->新页面->选择“Prodcut selector”，具体如下图所示：

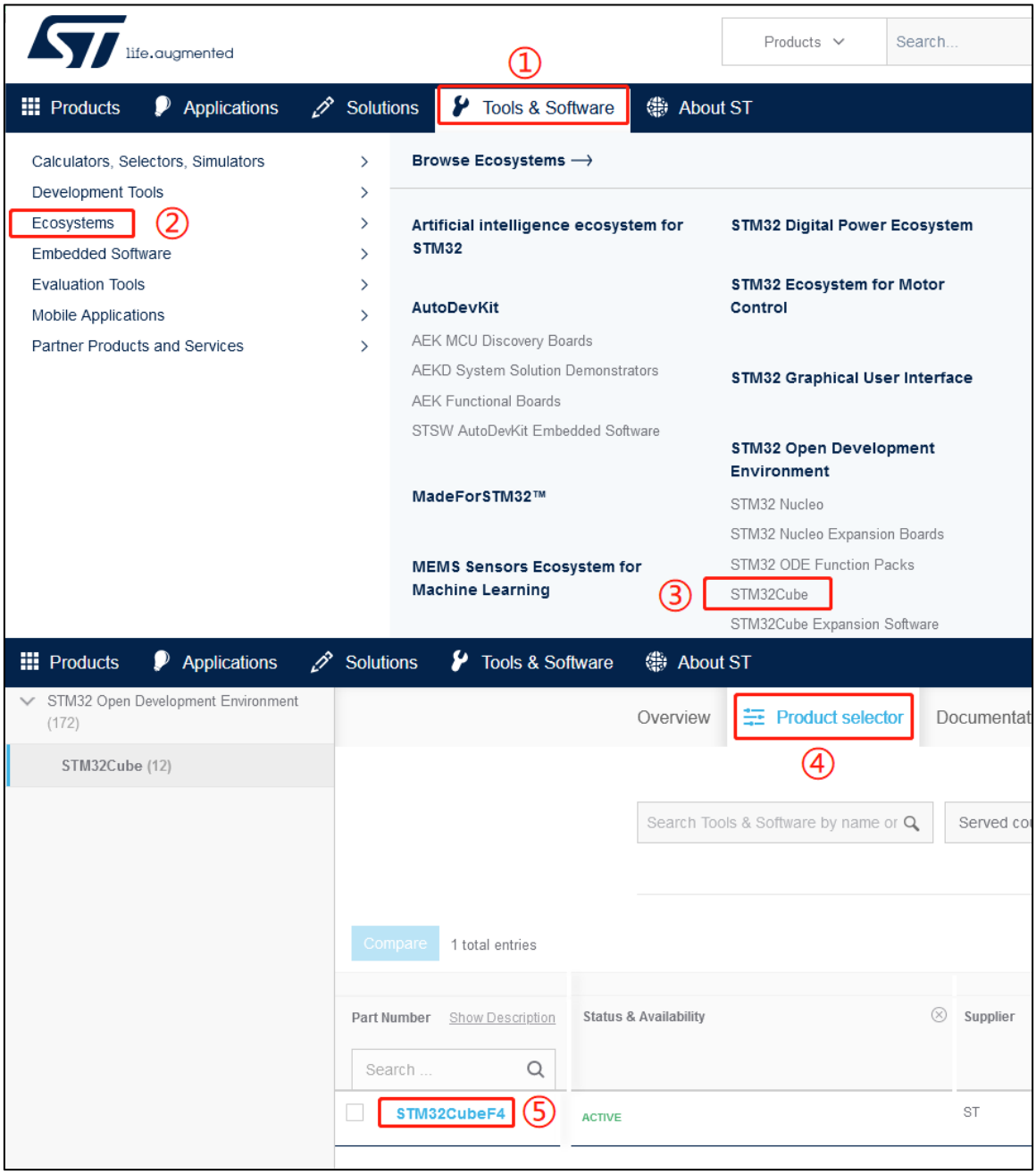


图 7.2.1.1 找到 STM32CubeF4 固件包的下载位置

选中“STM32CubeF4”，鼠标左键单击打开“Open software page”，按下图操作，在新的窗

口中拉到底部，选择适合自己的下载方式，注册帐号即可获取相应的驱动包。

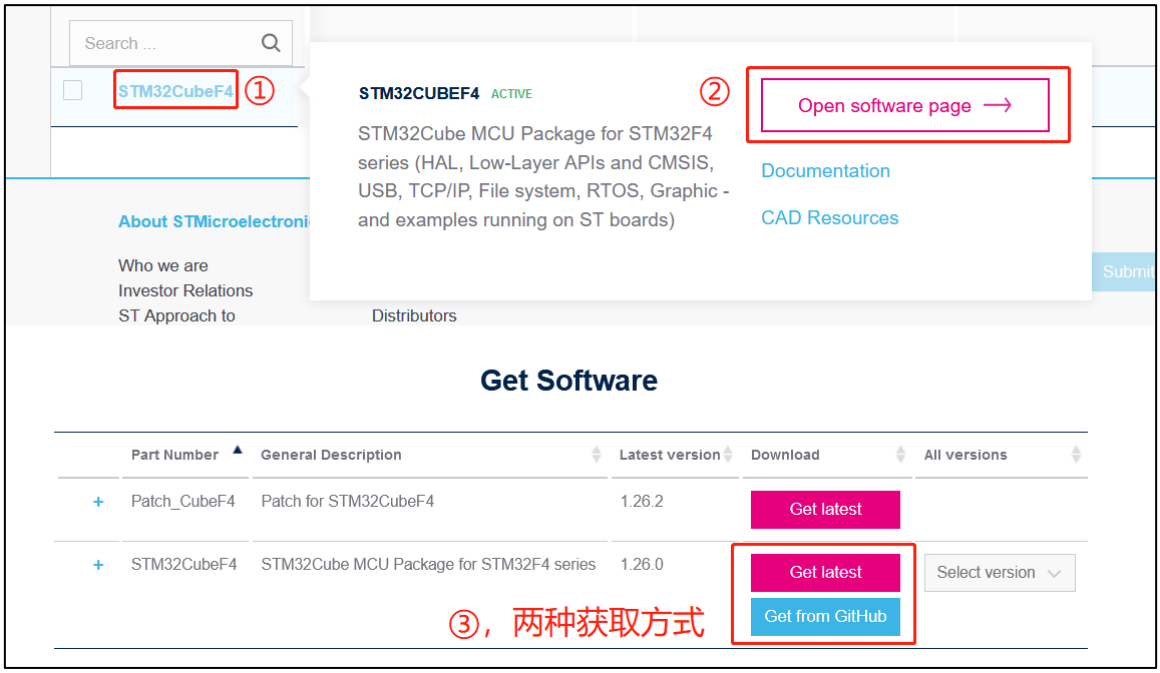


图 7.2.1.2 下载 STM32CubeF4 固件包

STM32Cube 固件包，我们已经给大家下载好并且放到 A 盘→8，STM32 参考资料→1，STM32CubeF4 固件包，当前固件包版本是：STM32Cube_FW_F4_V1.26.0。因为现在是 STM32F407 的学习，所以我们准备好的固件包是 F4 的。大家要根据自己学习的芯片，下载对应的固件包。如果需要最新的固件包，大家按照上述的方法到官网下载即可。

7.2.2 STM32Cube 固件包分析

STM32Cube 固件包完全兼容 STM32CubeMX。对于图形配置工具 STM32CubeMX 入门使用，需要 STM32F4 基础才能入门使用，所以我们安排在后面第十章给大家讲解。本小节，我们主要讲解 STM32Cube 固件包的结构。

解压 STM32CubeF4 固件包后，我们看看其目录结构，如图 7.2.2.1 所示。

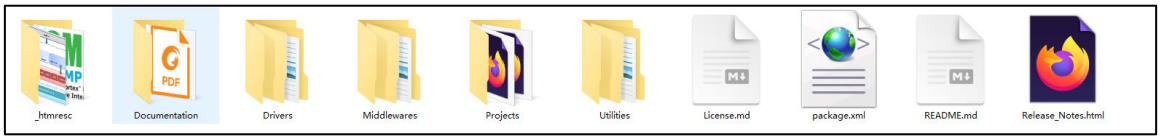


图 7.2.2.1 STM32CubeF4 固件包的目录结构

下面对 STM32CubeF4 固件包进行简要介绍。对于 Documentation 文件夹，里面是一个 STM32CubeF4 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 STM32CubeF4 中几个关键的文件夹。

（1）Drivers 文件夹

Drivers 文件夹包含 BSP，CMSIS 和 STM32F4xx_HAL_Driver 三个子文件夹。三个子文件夹具体说明请参考下表 7.2.2.1：

Drivers 文件夹	BSP 文件夹	也叫板级支持包，用于适配 ST 官方对应的开发板的硬件驱动程序，每一种开发板对应一个文件夹。例如触摸屏，LCD，NOR FLASH 以及 EEPROM 等板载硬件资源等驱动。这些文件针只匹配特定的开发板使用，不同开发板可能不能直接使用。
	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)。在新建工程的时候，会使用到这个文件夹内部很多文件。
	STM32F4xx_HAL_Driver 文件夹	这个文件夹非常重要，它包含了所有的 STM32F4xx 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 stm32f4xx_hal_ppp.c 定义格式，头文件名称基本遵循 stm32f4xx_hal_ppp.h 定义格式。比如 gpio 相关的 API 的声明和定义在文件 stm32f4xx_hal_gpio.h 和 stm32f4xx_hal_gpio.c 中。该文件夹的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 7.2.2.1 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 7.2.2.2：

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

表 7.2.2.2 Middlewares 文件夹介绍

（3）Projects 文件夹

该文件夹存放的是 ST 官方的开发板的适配例程，每个文件夹对应一个 ST 官方的 Demo 板，根据型号的不同提供 MDK 和 IAR 等类型的例程。里面有很多实例，读者可以根据自己的需要来作为参考。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，我们只需要双击它就可在 MDK 中打开工程。

（4）Utilities 文件夹

该文件夹是一些公用组件，也是主要为 ST 官方的 Demo 板提供的，在我们的例程中使用的不多。有兴趣的同学可以深入研究一下，这里我们不做过多介绍。

（5）其它几个文件

文件夹中还有几个单独的文件，用于声明软件版本或者版权信息，我们使用 ST 的芯片已经默认得到这个软件的版权使用授权，可以简单了解一下各文件的内容，实际项目中我们一般

不添加。

License.md：用于声明软件版权信息的文件。

package.xml：描述固件包版本信息的文件。

Release_Notes.html：超文本文件，用浏览器打开可知它是对固件包的补充描述和固件版本更新的记录说明。

7.2.3 CMSIS 文件夹关键文件

上一节中我们对 STM32cube 固件包的主要目录结构做了分析。这一小节在上一小节的基础上，我们来分析一下 CMSIS 文件夹：由命名可知该文件夹和 7.1.1 小节中提到的 CMSIS 标准是一致的，CMSIS 为软件包的内容制定了标准，包括文件目录的命名和内容构成，CMSIS 版本 5.7.0 的规定软件包目录如表 7.2.3.1 所示：

文件/目录		描述
LICENSE.txt		Apache 2.0 授权的许可文件
ARM.CMSIS.pdsc		描述该 CMSIS 包的文件
Device		基于 Arm Cortex-M 处理器设备的 CMSIS 参考实现
CMSIS 组件	Documentation	这个数据包的描述文档
	Core	CMSIS-Core（Cortex-M）相关文件的用户代码模板,在 ARM.CMSIS.pdsc 中引用
	Core_A	CMSIS-Core（Cortex-A）相关文件的用户代码模板，在 ARM.CMSIS.pdsc 中引用
	DAP	CMSIS-DAP 调试访问端口源代码和参考实现
	Driver	CMSIS 驱动程序外设接口 API 的头文件
	DSP_Lib	CMSIS-DSP 软件库源代码
	NN	CMSIS-NN 软件库源代码
	Include	CMSIS-Core(Cortex-M)和 CMSIS-DSP 需要包括的头文件等。
	Lib	包括 CMSIS 核心（Cortex-M）和 CMSIS-DSP 的文件
	Pack	CMSIS-Pack 示例，包含包含设备支持、板支持和软件组件的软件包示例。
	RTOS	CMSIS-RTOS 版本 1 以及 RTX4 参考实现
	RTOS2	CMSIS-RTOS 版本 2 以及 RTX5 参考实现
	SVD	CMSIS-SVD 样例，规定开发者、制造商、工具制造商的分工和职能
	Utilities	PACK.xsd(CMSIS-Pack 架构文件),PackChk.exe(检查软件包的工具)，CMSIS-SVD.xsd(MSIS-SVD 架构文件),SVDConv.exe(SVD 文件的转换工具)

表 7.2.3.1 CMSIS v5.7.0 的文件夹规范

知道了 CMSIS 规定的组件及其文件目录的大概内容后，我们再来看看 ST 提供的 CMSIS 文件夹，如上节提到的，它的位置是“STM32Cube_FW_F4_V1.26.0\Drivers\CMSIS”。打开文件夹内容如图 7.2.3.1 所示，可以发现它的目录结构完全按照 CMSIS 标准执行，仅仅是作了部分删减。

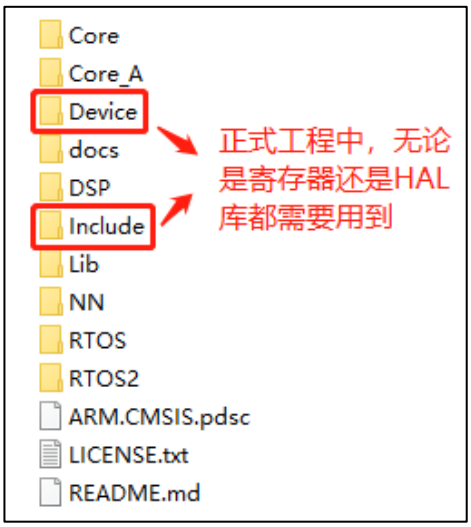


图 7.2.3.1 STM32CubeF4 固件包的 CMSIS 文件夹

CMSIS 文件夹中的 Device 和 Include 这两个文件夹中的文件是我们工程中最常用到的。下面对这两个文件夹作简单的介绍：

1）Device 文件夹

Device 文件夹关键文件介绍如下表 7.2.3.1 所示：

文件	描述
stm32f4xx.h	该文件内容看起来不多，却非常重要，是所有 STM32F4 系列的顶层头文件。使用 STM32F4 任何型号的芯片，都需要包含这个头文件。该文件包含了很多条件定义和常用的枚举变量类型，与宏定义配合，选择性包含某一特定的 STM32F4 系列芯片的头文件。这个文件使我们在使用 STM32F4 系列的不同型号芯片时，不需要每次都修改工程头文件，只需要修改宏定义并增加特定型号芯片的头文件即可快速选择使用不同类型的 F4 芯片。
stm32f407xx.h	STM32F407 系列芯片通用的片上外设访问层头文件。只要我们使用 STM32F407 系列芯片，都需要包括这个头文件。这个文件的主要作用是定义声明寄存器以及封装内存操作，以结构体和宏定义标识符的形式。
system_stm32f4xx.c system_stm32f4xx.h	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_stm32f4xx.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。
startup_stm32f407xx.s	STM32F407 系列芯片的启动文件，每个系列都有与之对应的启动文件。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。后面会细讲。

表 7.2.3.1 Device 文件夹关键文件介绍

表 7.1.2.1 列出的文件都是正式工程中必须的文件。固件包的 CMSIS 文件包括了所有 STM32F4 芯片型号的文件，而我们只用到 STM32F407 系列，所以只是挑我们用到的系列文件来讲。

（2）Include 文件夹

Include 文件夹存放了符合 CMSIS 标准的 Cortex-M 内核头文件。想要深入学习内核的朋友可以配合内核相关的手册去学习。对于 STM32F4 的工程，我们只要把我们需要的添加到工程即可，需要的头文件有：cmsis_armcc.h、cmsis_armclang.h、cmsis_compiler.h、cmsis_version.h、core_cm4.h 和 mpu_armv7.h。这几个头文件，对比起来，我们会比较多接触的是 core_cm4.h。

core_cm4.h 是内核底层的文件，由 ARM 公司提供，包含一些 AMR 内核指令，如软件复位，开关中断等功能。今后在需要的例程再去讲解其程序，现在之所以提到，是因为它包含了一个重要的头文件 stdint.h。

7.2.4 stdint.h 简介

stdint.h 是从 c99 中引进的一个标准 C 库的文件。在 2000 年 3 月，ANSI 采纳了 C99 标准。ANSI C 被几乎所有广泛使用的编译器（如：MDK、IAR）支持。多数 C 代码是在 ANSI C 基础上写的。任何仅仅使用标准 C 并且没有和任何硬件有依赖的代码实际上能保证在任何平台上用遵循 C 标准的编译器编译成功。就是说这套标准不依赖硬件，独立于任何硬件，可以跨平台。

stdint.h 可以在 MDK 安装目录下找到，如 MDK5 安装在 C 盘时，可以在路径：C:\Keil_v5\ARM\ARMCC\include 找到。stdint.h 的作用就是提供了类型定义，其部分类型定义代码如下：

```
/* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short    int int16_t;
typedef   signed          int int32_t;
typedef   signed           __INT64 int64_t;

/* exact-width unsigned integer types */
typedef unsigned           char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned           __INT64 uint64_t;
```

在今后的程序，我们都将会使用这些类型，比如：uint32_t（无符号整型）、int16_t 等。

7.3 HAL 库框架结构

这一节我们将简要分析一下 HAL 驱动文件夹下的驱动文件，帮助大家快速认识 HAL 库驱动的构成，旨在帮大家快速认识 HAL 库的函数的一些常用形式，帮助大家到遇到 HAL 库时能根据名字大致推断该函数的用法，本部分不要求大家在学习完本节后完全记住。

7.3.1 HAL 库文件夹结构

HAL 库头文件和源文件在 STM32Cube 固件包的 STM32F4xx_HAL_Driver 文件夹中，打开该文件夹，如图 7.3.1.1 所示。

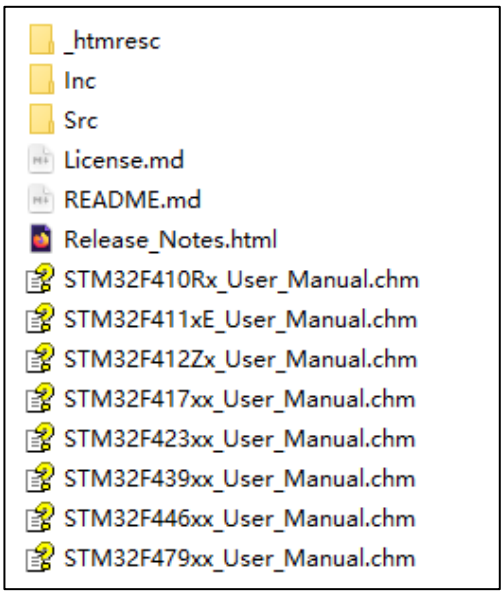


图 7.3.1.1 STM32H7xx_HAL_Driver 文件夹目录结构

STM32F407xx_HAL_Driver 文件夹下的 Src（Source 的简写）文件夹存放是所有外设的驱动程序源码，Inc（Include 的简写）文件夹存放的是对应源码的头文件。Release_Notes.html 是 HAL 库的版本更新信息。最后三个是库的用户手册，这个需要可以去熟悉一下，查阅起来很方便。

打开 Src 和 Inc 文件夹，大家会发现基本都是 stm32f4xx_hal_和 stm32f4xx_ll_开头的.c 和.h 文件。刚学 HAL 库的朋友可能会说，stm32f4xx_hal_开头的是 HAL 库，我能理解。那么 stm32f4xx_ll_开头的文件又是什么？stm32f4xx_ll_开头的文件是前面介绍过的 LL 库的文件。

7.3.2 HAL 库文件介绍

HAL 库关键文件介绍如下表 7.3.2.1 所示，表中 ppp 代表任意外设。

文件	描述
sm32f4xx_hal.c stm32f4xx_hal.h	初始化 HAL 库，（比如 HAL_Init,HAL_DeInit, HAL_Delay 等），主要实现 HAL 库的初始化、系统滴答，HAL 库延时函数、IO 重映射和 DBGMCU 功能等。
stm32f4xx_hal_conf.h HAL 库中本身没有这个文件，可以自行定义，也可以直接使用《Inc》文件夹下 stm32f4xx_hal_conf_template.h 的内容作为参考模版	HAL 的用户配置文件，stm32f4xx_hal.h 引用了这个文件，用来对 HAL 库进行裁剪。由于 Hal 库的很多配置都是通过预编译的条件宏来决定是否使用这一 HAL 库的功能，这也是当前的主流库如 LWIP /FreeRTOS 等的做法，无需修改库函数的源码，通过使能/不使能一些宏来实现库函数的裁剪。
stm32f4xx_hal_def.h	通用 HAL 库资源定义，包含 HAL 的通用数据类型定义，声明、枚举，结构体和宏定义。如 HAL 函数操作结果返回值类型 HAL_StatusTypeDef 就是在这个文件中定义的。
stm32f4xx_hal_cortex.h stm32f4xx_hal_cortex.c	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，与前面 core_cm4.h 的功能类似。
stm32f4xx_hal_ppp.c stm32f4xx_hal_ppp.h	外设驱动函数。对于所有的 STM32 该驱动名称都相同，ppp 代表一类外设，包含该外设的操作 API 函数。例如：当 ppp 为 adc 时，那么这个文件就是 stm32f4xx_hal_adc.c/h，可以分别在 Src/Inc 目录下找到。包含该外设的操作 API 函数。其中 stm32f4xx_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，MPU，系

	统软复位以及 Systick 配置等。
stm324fxx_hal_ppp_ex.c stm32f4xx_hal_ppp_ex.h	外设特殊功能的 API 文件，作为标准外设驱动的功能补充和扩展。针对部分型号才有的特殊外设作功能扩展，或外设的实现功能与标准方式完全不同的情况下作重新初始化的备用接口。ppp 的含义同标准外设驱动。
stm32f4xx_II_ppp.c stm32f4xx_II_ppp..h	LL 库文件，在一些复杂外设中实现底层功能，在部分 stm32f4xx_hal_ppp.c 中被调用。

表 7.3.2.1 HAL 库关键文件介绍

以上是 HAL 库最常见的文件的列表，在 Src/Inc 下面还有 Legacy 文件夹，用于特殊外设的补充说明。我们的教程中用到的比较少，这里不展开描述。

不止文件命名有一定规则，stm32f4xx_hal_ppp (c/h)中的函数和变量命名也严格按照命名规则，如表 7.3.2.2 所示的命名规则在大部分情况下都是正确的：

文件名	stm32f4xx_hal_ppp (c/h)	stm32f4xx_hal_ppp_ex (c/h)
函数名	HAL_PPP_Function HAL_PPP_FeatureFunction_MODE	HAL_PPPEx_Function HAL_PPPEx_FeatureFunction_MODE
外设句柄	PPP_HandleTypedef	无
初始化参数结构体	PPP_InitTypeDef	PPP_InitTypeDef
枚举类型	HAL_PPP_StructnameTypeDef	无

表 7.3.2.2 HAL 库函数、变量命名规则

对于 HAL 的 API 函数，常见的有以下几种：

- 初始化/反初始化函数:HAL_PPP_Init(), HAL_PPP_DeInit()
- 外设读写函数:HAL_PPP_Read(),HAL_PPP_Write(),HAL_PPP_Transmit(), HAL_PPP_Receive()
- 控制函数:HAL_PPP_Set (), HAL_PPP_Get ().
- 状态和错误:HAL_PPP_GetState (), HAL_PPP_GetError ().

HAL 库封装的很多函数都是通过定义好的结构体将参数一次性传给所需函数，参数也有一定的规律，主要有以下三种：

➢ 配置和初始化用的结构体:

一般为 PPP_InitTypeDef 或 PPP_ConfTypeDef 的结构体类型，根据外设的寄存器设计成易于理解和记忆的结构体成员。

➢ 特殊处理的结构体

专为不同外设而设置的，带有“Process”的字样，实现一些特异化的中间处理操作等。

➢ 外设句柄结构体

HAL 驱动的重要参数，可以同时定义多个句柄结构以支持多外设多模式。HAL 驱动的操作结果也可以通过这个句柄获得。有些 HAL 驱动的头文件中还定义了一些跟这个句柄相关的一些外设操作。如用外设结构体句柄与 HAL 定义的一些宏操作配合，即可实现一些常用的寄存器位操作。

宏定义结构	用途
__HAL_PPP_ENABLE_IT(__HANDLE__, __INTERRUPT__)	使能外设中断
__HAL_PPP_DISABLE_IT(__HANDLE__, __INTERRUPT__)	禁用外设中断
__HAL_PPP_GET_IT (__HANDLE__, __INTERRUPT__)	获取外设的某一中断源
__HAL_PPP_CLEAR_IT (__HANDLE__, __INTERRUPT__)	清除外设中断
__HAL_PPP_GET_FLAG (__HANDLE__, __FLAG__)	获取外设的状态标记
__HAL_PPP_CLEAR_FLAG (__HANDLE__, __FLAG__)	清除外设的状态标记
__HAL_PPP_ENABLE(__HANDLE__)	使能某一外设
__HAL_PPP_DISABLE(__HANDLE__)	禁用某一外设
__HAL_PPP_XXXX (__HANDLE__, __PARAM__)	针对外设的特殊操作
__HAL_PPP_GET_IT_SOURCE (__HANDLE__, __INTERRUPT__)	检查外设的中断源

表 7.3.2.3 HAL 库驱动部分与外设句柄相关的宏

但对于 SYSTICK/NVIC/RCC/FLASH/GPIO 这些内核外设或共享资源，不使用 PPP_HandleTypedef 这类外设句柄进行控制，如：HAL_GPIO_Init()只需要初始化的 GPIO 编号和具体的初始化参数。

```
HAL_StatusTypeDef HAL_GPIO_Init (GPIO_TypeDef* GPIOx, GPIO_InitTypeDef *Init)
{
/* GPIO 初始化程序…… */
}
```

最后要分享的是 HAL 库的回调函数，这部分允许用户重定义，并在其中实现用户自定义的功能，也是我们使用 HAL 库的最常用的接口之一：

回调函数	举例
HAL_PPP_MspInit() / _DeInit()	举例: HAL_USART_MspInit() 由 HAL_PPP_Init()这个 API 调用，主要在这个函数中实现外设对应的 GPIO、时钟、DMA 和中断开启的配置和操作。
HAL_PPP_ProcessCpltCallback	举例: HAL_USART_TxCpltCallback 由外设中断或 DMA 中断调用 ，调用时 API 内部已经实现中断标记的清除的操作，用户只需要专注于自己的软件功能实现即可。
HAL_PPP_ErrorCallback	举例: HAL_USART_ErrorCallback 外设或 DMA 中断中发生的错误，用于作错误处理。

表 7.3.2.4 HAL 库驱动中常用的回调函数接口

至此，我们大概对 HAL 库驱动文件的一些通用格式和命名规则有了初步印象，记住这些规则可以帮助我们快速对 HAL 库的驱动进行归类和判定这些驱动函数的用法。

ST 官方给我们提供了快速查找 API 函数的帮助文档。在路径“STM32Cube_FW_F4_V1.26.0\Drivers\STM32F4xx_HAL_Driver”下有几个 chm 格式的文档，根据我们开发板主控芯片 STM23F407IGT6 我们没有找到直接可用的，但可以查看型号接近的：STM32F410Rx_User_Manual.chm。双击打开后，可以看到左边目录下有四个主题，我们来查看 Modules。以外设 GPIO 为例，讲一下怎么使用这个文档。点击 GPIO 外设的主题下的 IO operation functions /functions 看看里面的 API 函数接口描述，如图 7.3.2.1 所示。

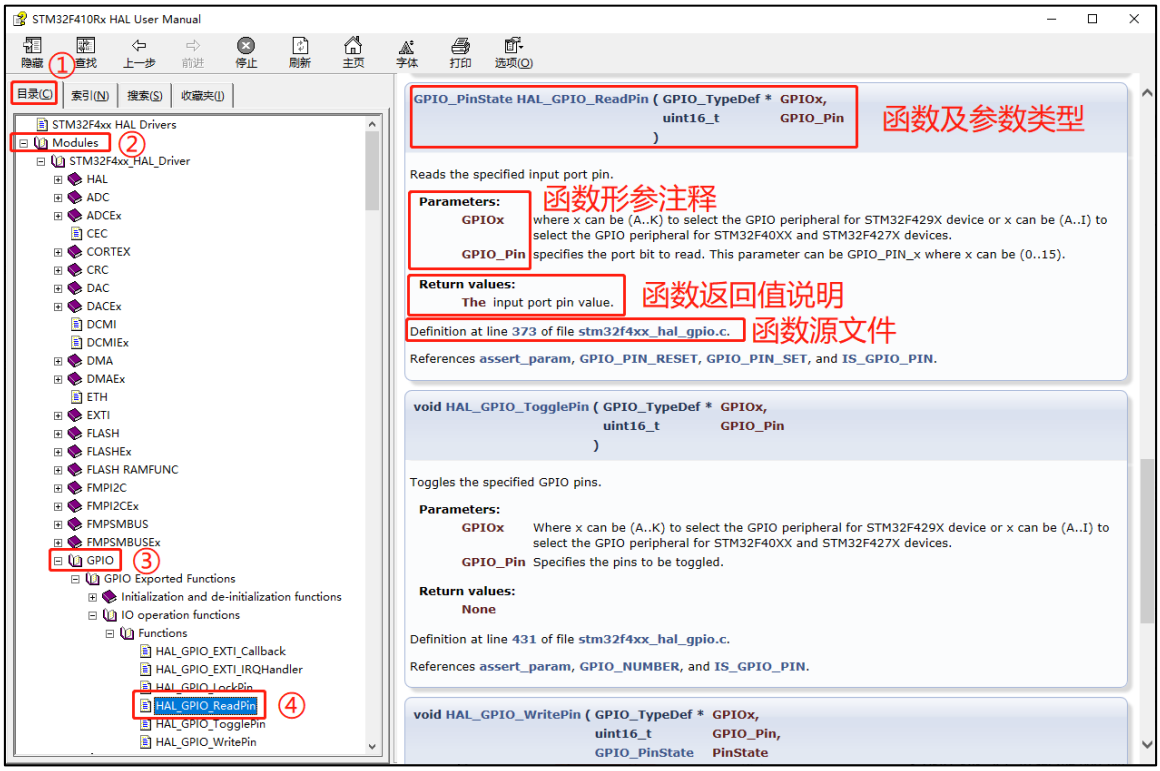


图 7.3.2.1 文档的 API 函数描述

这个文档提供的信息很全，不看源码都可以直接使用它来编写代码，还给我们指示源码位置，非常方便。大家多翻一下其他主题了解一下文档的信息结构，很容易使用。

下面举个例子，比如我们要让 PB4 输出高电平。先看函数功能，HAL_GPIO_WritePin 函数就是我们的 GPIO 口输出设置函数。

函数有三个形参：

第一个形参是 GPIO_TypeDef *GPIOx，形参描述说：x 可以是 A 到 K 之间任何一个，而我们是 PB4 引脚，所以第一个形参确认是 GPIOB。

第二个形参是 uint16_t GPIO_Pin，看形参描述：该参数可以是 GPIO_PIN_x，x 可以 1 到 15，那么我们第二个形参就是 GPIO_PIN_4。

第三个形参是 GPIO_PinState_PinState，看形参描述：该参数可以是枚举里的两个数，一个是 GPIO_PIN_RESET：表示该位清零，另一个是 GPIO_PIN_SET：表示设置该位，即置 1，我们要输出 1，所以要置 1 该位，那么我们第三个形参就是 GPIO_PIN_SET。

最后看函数返回值：None，没有返回值。

所以最后得出我们要调用的函数是：

```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
```

文档的使用就讲到这。

7.4 如何使用 HAL 库

我们要先知道 STM32 芯片的某个外设的性能和工作模式，才能借助 HAL 库来帮助我们编程，甚至修改 HAL 库来适配我们的开发项目。HAL 库的 API 虽多，但是查找和使用有规律可循，只要学会其中一个，其他的外设就是类似的，只是添加自己的特性的 API 而已。

需要按照芯片使用手册建议的步骤去配置芯片。HAL 库驱动提供了芯片的驱动接口，但我们需要强调一个概念是使用 HAL 库的开发是对芯片功能的开发，而不是开发这个库，也不是由这个库能就直接开发。如果我们对芯片的功能不作了解的话，仍然不知道按照怎样的步骤和寻找哪些可用的接口去实现想要实现的功能。

ST 提供芯片使用手册《STM32F4xx 参考手册_V6（英文版）.pdf》告诉我们使用某一外设功能时如何具体地去操作每一个用到的寄存器的细节，后面我们的例程讲解过程也会结合这个手册来分析配置过程。

嵌入式的软件开发流程总遵循以下步骤：组织工具链、编写代码、生成可执行文件、烧录到芯片、芯片根据内部指令执行我们编程生成的可执行代码。

在 HAL 库学习前期，建议以模仿和操作体验为基础，通过例程来学习如何配置和驱动外设。下面根据我们后续要学习的工程梳理出来的基于 CMSIS 的一个 HAL 库应用程序文件结构，帮助读者学习和体会这些文件的组成意义，如下表 7.4.1.1 所示。

类别	文件名	描述	是否必须
用户程序文件	main.c	存放 main 函数，不一定要在这个文件	否
	main.h	包含头文件、声明等作用，已删除	否
	stm32f4xx_it.c	用户中断服务函数存放文件，不一定放到这个文件，可删除	否
	stm32f4xx_it.h		否
	stm32f4xx_hal_conf.h	用户配置文件	是
	stm32f4xx_hal_msp.c	回调函数存放文件，已删除	否
设备驱动层	sm32f4xx_hal.c	HAL 库的初始化、系统滴答，HAL 库延时函数等功能	是
	stm32f4xx_hal.h		是
	stm32f4xx_hal_def.h	通用 HAL 库资源定义	是
	stm32f4xx_hal_ppp.c	外设的操作 API 函数文件	是
	stm32f4xx_hal_ppp.h		是
	stm32f4xx_hal_ppp_ex.c	拓展外设特性的 API 函数文件	是
	stm32f4xx_hal_ppp_ex.h		是

	stm32f4xx_II_ppp.c	LL 库文件，在一些复杂外设中实现底层功能	是
	stm32f4xx_II_ppp..h		是
CMSIS 核心层	stm32f4xx.h	STM32F4 系列的顶层头文件	是
	stm32f407xx.h	STM32F407 系列片上外设头文件	是
	system_stm32f4xx.c	主要存放系统初始化函数 SystemInit	是
	system_stm32f4xx.h		是
	startup_stm32hf407xx.s	启动文件，运行到 main 函数前的准备	是
	core_cm4.h	内核寄存器定义，如 Systick、SCB 等	是
	cmsis_armcc.h	内核头文件，一般都不需要去了解	是
	cmsis_armclang.h		
	cmsis_compiler.h		
	cmsis_version.h		
	mpu_armv7.h		

表 7.4.1.1 基于 CMSIS 应用程序文件描述

把这些文件组织起来的方法，我们会在后续章节新建工程中介绍，这只提前告诉一下大家组成我们需要的编译工具链大概需要哪些文件。

7.4.2 HAL 库的用户配置文件

stm32f4xx_hal_conf.h 用于裁剪 HAL 库和定义一些变量，官方没有直接提供这个文件，但在 STM32Cube_FW_F4_V1.26.0\Drivers\STM32F4xx_HAL_Driver\Inc 这个路径下提供了一个模版文件《stm32f4xx_hal_conf_template.h》，我们可以直接复制这个文件重命名为 stm32f4xx_hal_conf.h，做一些简单的修改即可，也可以从在官方的例程中直接复制过来。我们一般都直接从官方的模板例程中直接复制过来即可。因为我们的芯片是 STM32F407 的 G 系列，所以选择的路径是：STM32Cube_FW_F4_V1.26.0\Projects\STM324xG_EVAL\Templates\Inc。

（1）stm32f4xx_hal_conf.h 文件里面的内容不多，对我们来说最重要的是 HSE_VALUE 这个参数，这个参数表示我们的外部高速晶振的频率。这个参数请务必根据我们板子外部焊接的晶振频率来修改，官方默认是 25M。正点原子 DMF407 外部高速晶振的频率是 8MHZ。**注意事项：**我们要修改这个参数，源码在 84 行，具体修改如下：

```
#if !defined  (HSE_VALUE)
#define HSE_VALUE    ((uint32_t)8000000) /* 外部高速振荡器的值，单位 HZ */
#endif /* HSE_VALUE */
```

（2）还有一个参数就是外部低速晶振频率，这个官方默认是 32.768KHZ，我们开发板的低速晶振也是这个频率，所以不用修改，源码在 112 行。

```
#if !defined  (LSE_VALUE)
  #define LSE_VALUE    ((uint32_t)32768) /* 外部低速振荡器的值，单位 HZ */
#endif /* LSE_VALUE */
```

其他源码都可以不作修改，按照默认的配置即可。下面我们再来了解一下其他程序。

（3）用户配置文件可以用来选择使能何种外设，源码配置在 35 行到 74 行，代码如下。

```
/* ########################## Module Selection ############################## */
/**
  * @brief This is the list of modules to be used in the HAL driver
  */
#define HAL_MODULE_ENABLED
#define HAL_ADC_MODULE_ENABLED
#define HAL_CAN_MODULE_ENABLED
/* #define HAL_CAN_LEGACY_MODULE_ENABLED */
#define HAL_CRC_MODULE_ENABLED
...中间省略...
#define HAL_WWDG_MODULE_ENABLED
#define HAL_CORTEX_MODULE_ENABLED
#define HAL_PCD_MODULE_ENABLED
#define HAL_HCD_MODULE_ENABLED
```

我们只要屏蔽某个外设的宏，则这个外设的驱动代码机会被屏蔽，从而不可用。比如我们屏蔽 GPIO 外设的宏，源码在 54 行，屏蔽就是把这个宏注释掉，具体如下。

```
/* #define HAL_GPIO_MODULE_ENABLED */
```

结合同样在《stm32f4xx_hal_conf.h》中的 277 行的代码：

```
#ifdef HAL_GPIO_MODULE_ENABLED
#include "stm32f4xx_hal_gpio.h"
#endif /* HAL_GPIO_MODULE_ENABLED */
```

这是一个条件编译符，与#endif 配合使用。这里的要表达的意思是，只要工程中定义了 HAL_GPIO_MODULE_ENABLED 这个宏，#ifdef 到#endif 之间的程序（119 行到 523 行）就会参与编译，否则不编译。所以只要我们屏蔽了 stm32f4xx_hal_conf.h 文件 54 行的宏，GPIO 的驱动代码就不被编译。也就起到选择使能何种外设的功能，其他外设同理。

官方的示范例程，就是通过屏蔽外设的宏的方法来选择使能何种外设。好处就是编译时间会变短，因为屏蔽了没有用的程序，编译时间自然就短了。正点原子的例程选择另外一中方法，就是工程中只保留需要的 stm32f4xx_hal_ppp.c，不需要的不添加到工程里，这样编译时间就不会太长。

（4）大家看到 stm32f4xx_hal_conf.h 文件的 136 行。

```
#define  TICK_INT_PRIORITY    ((uint32_t)0x0F) /*!< tick interrupt priority */
```

宏定义 TICK_INT_PRIORITY 是滴答定时器的优先级。这个优先级很重要，因为如果其它的外设驱动程序的延时是通过滴答定时器提供的时间基准，来实现延时的话，又由于实现方式是滴答定时器对寄存器进行计数，所以当我们在其它中断服务程序里调用基于此时间基准的延迟函数 HAL_Delay，那么假如该中断的优先级高于滴答定时器的优先级，就会导致滴答定时器中断服务函数一直得不到运行，从而程序卡死在这里。所以滴答定时器的中断优先级一定要比这些中断高。

请注意这个时间基准可以是滴答定时器提供，也可以是其他的定时器，默认是用滴答定时器。

（5）下面说一下关于断言这个功能，这个功能用来判断函数的形参是否有效，在 HAL 库的 API 里面有用到。这个功能的使能开关代码是一个宏，在源码的 187 行，默认是关闭的，代码如下。

```
/* #define USE_FULL_ASSERT    1 */
```

通过宏 USE_FULL_ASSERT 来选择功能，在源码 426 行到 440，代码如下。

```
/* Exported macro ------------------------------------------------------------*/
#ifdef  USE_FULL_ASSERT
/**
  * @brief  The assert_param macro is used for function's parameters check.
  * @param  expr: If expr is false, it calls assert_failed function
  *         which reports the name of the source file and the source
  *         line number of the call that failed.
  *         If expr is true, it returns no value.
  * @retval None
  */
  #define assert_param(expr) ((expr) ? (void)0U : assert_failed((uint8_t
*)__FILE__, __LINE__))
/* Exported functions ------------------------------------------------------- */
  void assert_failed(uint8_t* file, uint32_t line);
#else
  #define assert_param(expr) ((void)0U)
#endif
#endif /* USE_FULL_ASSERT */
```

也是通过条件编译符来选择对应的功能。当用户自己需要使用断言功能，怎么做呢？首先需要定义宏 USE_FULL_ASSERT 来使能断言功能，即把源码的 187 行的注释去掉即可。然后看到源码 437 行的 assert_failed（）这个函数。其实这个函数是需要我们自己实现的，我们把这个函数定义在正点原子提供的 sys.c 文件里面。后面再跟大家讲 sys.c 文件，现在我们把 assert_failed（）这个函数拿出来给大家先讲，assert_failed（）函数的定义在 sys.c 的 178 行到 192 行，具体如下：

```
#ifdef  USE_FULL_ASSERT

/**
 * @brief       当编译提示出错的时候此函数用来报告错误的文件和所在行
 * @param       file：指向源文件
```

```
 *             line: 指向在文件中的行数
 * @retval      无
 */
void assert_failed(uint8_t* file, uint32_t line)
{
    while (1)
    {
    }
}
#endif
```

可以看到这个函数里面没有实现如何功能，就是一个什么不做的死循环，具体功能请根据自己的需求去实现。file 是指向源文件的指针，line 是指向源文件的行数。__FILE__是表示源文件名，__LINE__是表示在源文件中的行数。比如我们可以实现打印出这个错误的两个信息等等。

总的来说断言功能就是，在 HAL 库中，如果定义了 USE_FULL_ASSERT 这个宏，那么所有的 HAL 库函数将会检查函数的形参是否正确。如果错误将会调用 assert_failed（）这个函数，这个函数我们默认是个什么事不做的死循环，用户请根据自己的需求设计功能。使用断言功能将会增加了代码量，减慢运行速度等，所以一般只是在调试的时候用，正式发布的软件是不推荐的。

7.4.3 stm32f4xx_hal.c 文件

这个文件内容比较多，包括 HAL 库的初始化、系统滴答、基准电压配置、IO 补偿、低功耗、EXTI 配置等都集合在这个文件里面。下面我们对该文件进行讲解。

1. HAL_Init()函数

源码在 157 行到 183 行，简化函数如下（下面的代码只针对 F4 的 HAL 固件 1.26.0 版本，其它版本可能有差异）：

```
HAL_StatusTypeDef HAL_Init(void)
{
  /* 设置中断优先级分组 */
  HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

  /* 使用滴答定时器作为时钟基准，配置 1ms 滴答（重置后默认的时钟源为 HSI） */
  if(HAL_InitTick(TICK_INT_PRIORITY) != HAL_OK)
  {
    return HAL_ERROR;
  }

  /* 初始化硬件 */
  HAL_MspInit();

  /* 返回函数状态 */
  return HAL_OK;
}
```

该函数是 HAL 库的初始化函数，在程序中必须优先调用，其主要实现如下功能：

1）设置 NVIC 优先级分组为 4。

2）配置滴答定时器每 1ms 产生一个中断。

3）在这个阶段，系统时钟还没有配置好，因此系统还是默认使用内部高速时钟源 HSI 在跑程序。对于 F4 来说，HSI 的时钟频率是 16MHZ。所以如果用户不配置系统时钟的话，那么系统将会使用 HSI 作为系统时钟源。

4）调用 HAL_MspInit 函数初始化底层硬件，HAL_MspInit 函数在 stm32f4xx_hal.c 文件里面做了弱定义。关于弱定义这个概念，后面会有讲解，现在不理解没关系。正点原子的 HAL 库例程是没有使用到这个函数去初始化底层硬件，而是单独调用需要用到的硬件初始化函数。用户可以根据自己的需求选择是否重新定义该函数来初始化自己的硬件。

注意事项：

为了方便和兼容性，正点原子的 HAL 库例程中的中断优先级分组设置为分组 2，即把源码的 173 行改为如下代码：

```
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_2);
```

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。

2. HAL_DeInit ()函数

源码在 190 行到 213 行，函数如下：

```
HAL_StatusTypeDef HAL_DeInit(void)
{
  /* 复位所有外设 */
  __HAL_RCC_APB1_FORCE_RESET();
  __HAL_RCC_APB1_RELEASE_RESET();

  __HAL_RCC_APB2_FORCE_RESET();
  __HAL_RCC_APB2_RELEASE_RESET();

  __HAL_RCC_AHB1_FORCE_RESET();
  __HAL_RCC_AHB1_RELEASE_RESET();

  __HAL_RCC_AHB2_FORCE_RESET();
  __HAL_RCC_AHB2_RELEASE_RESET();

  __HAL_RCC_AHB3_FORCE_RESET();
  __HAL_RCC_AHB3_RELEASE_RESET();

  /* 对底层硬件初始化进行复位 */
  HAL_MspDeInit();

  /* 返回函数状态 */
  return HAL_OK;
}
```

该函数取消初始化 HAL 库的公共部分，并且停止 systick，是一个可选的函数。该函数做了一下的事：

1）复位了 AHB1、AHB2、AHB3、APB1H、APB2 的时钟。

2）调用 HAL_MspDeInit 函数，对底层硬件初始化进行复位。HAL_MspDeInit 也在 stm32f4xx_hal.c 文件里面做了弱定义，并且与 HAL_MspInit 函数是一对存在。HAL_MspInit 函数负责对底层硬件初始化，HAL_MspDeInit 函数则是对底层硬件初始化进行复位。这两个函数都是需要用户根据自己的需求去实现功能，也可以不使用。

3. HAL_InitTick ()函数

源码在 253 行到 274 行，简化函数如下：

```
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /* 配置滴答定时器 1ms 产生一次中断 */
  if (HAL_SYSTICK_Config(SystemCoreClock /(1000UL / (uint32_t)uwTickFreq))> 0U)
  {
    return HAL_ERROR;
  }

  /* 配置滴答定时器中断优先级 */
  if (TickPriority < (1UL << __NVIC_PRIO_BITS))
  {
    HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority, 0U);
    uwTickPrio = TickPriority;
  }
  else
  {
    return HAL_ERROR;
```

```
  }

  /* 返回函数状态 */
  return HAL_OK;
}
```

该函数用于初始化滴答定时器的时钟基准，主要功能如下：

1）配置滴答定时器 1ms 产生一次中断。

2）配置滴答定时器的中断优先级。

3）该函数是__weak 定义的“弱函数”，用户可以重新定义该函数。

该函数可以通过 HAL_Init()或者 HAL_RCC_ClockConfig()重置时钟。在默认情况下，滴答定时器是时间基准的来源。如果其他中断服务函数调用了 HAL_Delay()，必须小心，滴答定时器中断必须具有比调用了 HAL_Delay()函数的其他中断服务函数的优先级高(数值较低)，否则会导致滴答定时器中断服务函数一直得不到执行，从而卡死在这里。

4. 滴答定时器相关的函数

源码在 303 行到 435 行，相关函数如下：

```
/* 该函数在滴答定时器时钟中断服务函数中被调用，一般滴答定时器 1ms 中断一次，
   所以函数每 1ms 让全局变量 uwTick 计数值加 1 */
__weak void HAL_IncTick(void)
{
  uwTick += (uint32_t)uwTickFreq;
}

/* 获取全局变量 uwTick 当前计算值 */
__weak uint32_t HAL_GetTick(void)
{
  return uwTick;
}

/* 获取滴答时钟优先级 */
uint32_t HAL_GetTickPrio(void)
{
  return uwTickPrio;
}

/* 设置滴答定时器中断频率 */
HAL_StatusTypeDef HAL_SetTickFreq(HAL_TickFreqTypeDef Freq)
{
  HAL_StatusTypeDef status  = HAL_OK;
  HAL_TickFreqTypeDef prevTickFreq;
  assert_param(IS_TICKFREQ(Freq));
  if (uwTickFreq != Freq)
  {
    /* 备份滴答定时器中断频率 */
    prevTickFreq = uwTickFreq;

    /* 更新被 HAL_InitTick()调用的全局变量 uwTickFreq */
    uwTickFreq = Freq;

    /* 应用新的滴答定时器中断频率  */
    status = HAL_InitTick(uwTickPrio);
    if (status != HAL_OK)
    {
      /* 恢复之前的滴答定时器中断频率 */
      uwTickFreq = prevTickFreq;
    }
  }
  return status;
}
```

```
/* 获取滴答定时器中断频率 */
HAL_TickFreqTypeDef HAL_GetTickFreq(void)
{
  return uwTickFreq;
}

/*HAL 库的延时函数，默认延时单位 ms  */
__weak void HAL_Delay(uint32_t Delay)
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;

  /* Add a freq to guarantee minimum wait */
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(uwTickFreq);
  }
  while ((HAL_GetTick() - tickstart) < wait)
  {
  }
}

/* 挂起滴答定时器中断，全局变量 uwTick 计数停止 */
__weak void HAL_SuspendTick(void)
{
  /* 禁止滴答定时器中断 */
  SysTick->CTRL &= ~SysTick_CTRL_TICKINT_Msk;
}

/* 恢复滴答定时器中断，恢复全局变量 uwTick 计数 */
__weak void HAL_ResumeTick(void)
{
  /* 使能滴答定时器中断 */
  SysTick->CTRL  |= SysTick_CTRL_TICKINT_Msk;
}
```

这些函数不是很难，请参照注释理解。注意：如果函数被前缀__weak 定义，则用户可以重新定义该函数。更多的内容可以参考 8.1.5 小节。

5. HAL 库版本相关的函数

源码在 241 行到 243 行和 252 行到 254 行，相关函数声明如下：

```
uint32_t HAL_GetHalVersion(void);   /* 获取 HAL 库驱动程序版本 */
uint32_t HAL_GetREVID(void);        /* 获取设备修订标识符 */
uint32_t HAL_GetDEVID(void);        /* 获取设备标识符 */
uint32_t HAL_GetUIDw0(void);        /* 获取唯一设备标识符的第一个字 */
uint32_t HAL_GetUIDw1(void);        /* 获取唯一设备标识符的第二个字 */
uint32_t HAL_GetUIDw2(void);        /* 获取唯一设备标识符的第三个字 */
```

这些函数了解一下就好了，用得不多。

6. 调试功能相关函数

源码在 244 行到 249 行，函数声明如下：

```
void HAL_DBGMCU_EnableDBGSleepMode(void);
void HAL_DBGMCU_DisableDBGSleepMode(void);
void HAL_DBGMCU_EnableDBGStopMode(void);
void HAL_DBGMCU_DisableDBGStopMode(void);
void HAL_DBGMCU_EnableDBGStandbyMode(void);
void HAL_DBGMCU_DisableDBGStandbyMode(void);
```

这六个函数用于调试功能，默认调试器在睡眠模式下无法调试代码，开发过程中配合这些函数，可以在不同模式下(睡眠模式、停止模式和待机模式)，使能或者失能调试器，当我们使用到时再作详细。

7.4.4 HAL 库中断处理

中断是 STM32 开发的一个很重要的概念，这里我们可以简单地理解为：STM32 暂停了当前手中的事并优先去处理更重要的事务。而这些“更重要的事务”是由软件开发人员在软件中定义的。关于 STM32 中断的概念，我们会在中断例程的讲解再跟大家详细介绍。

由于 HAL 库中断处理的逻辑比较统一，我们将这个处理过程抽象为图 7.4.4.1 所表示的业务逻辑：

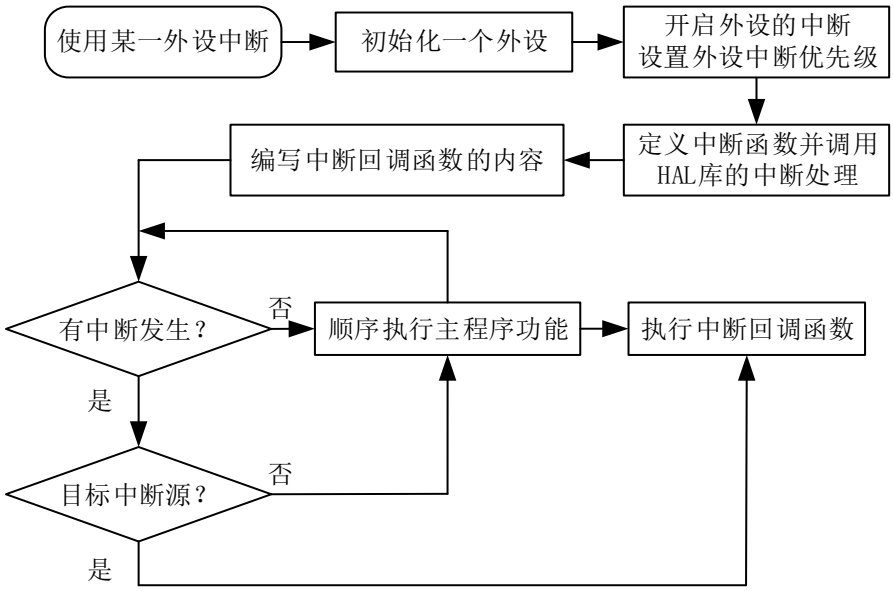


图 7.4.4.1 HAL 驱动中断处理流程

结合以前的 HAL 库文件介绍章节，以上的流程大概就是：设置外设的控制句柄结构体 PPP_HandleType 和初始化 PPP_InitType 结构体的参数，然后调用 HAL 库对应这个驱动的初始化 HAL_PPP_Init()，由于这个 API 中有针对外设初始化细节的接口 Hal_PPP_Mspinit()，我们需要重新实现这个函数并完成外设时钟、IO 等细节差异的设置， 完成各细节处理后，使用 HAL_NVIC_SetPriority()、HAL_NVIC_EnableIRQ()来使能我们的外设中断；定义中断处理函数 PPP_IRQHandler，并在中断函数中调用 HAL_ppp_function_IRQHandler()来判断和处理中断标记；HAL 库中断处理完成后，根据对应中的调用我们需要自定义的中断回调接口 HAL_PPP_ProcessCpltCallback();如串口接收函数 HAL_UART_RxCpltCallback()，我们在这个函数中实现我们对串口接收数据想做的处理；中断响应处理完成后，stm32 芯片继续顺序执行我们定义的主程序功能，按照以上处理的标准流程完成了一次中断响应。

7.4.5 正点原子对 HAL 库用法的个性化修改

前面按 ST 官方建议的 HAL 库的使用方法给介绍了一个 HAL 库。

1、将中断处理函数独立到每个外设中，便于独立驱动；同类型的外设驱动处理函数不使用 HAL 回调函数接口处理操作而直接在中断函数中处理判断对应中断。

2、我们把原来的中断分组进行了修改，由抢占式无子优先级改为中断分组 2；便于管理同类外设的优先级响应。

3、在很多芯片的初始化过程中，我们使用到了 delay_ms()、delay_us()等函数进行初始化，使用的是 Systick 作的精准延时，而 HAL 库默认也使用 Systick 作延时处理，为解决这种冲突和兼容我们大部分的驱动代码，我们在例程中使用 delay.c 中的延时函数取代 HAL_Delay();取消原来 HAL 库的 Systick 延时设置。

7.5 HAL 库使用注意事项

本小节根据经验跟大家讲述一些关于 HAL 库使用的注意事项，供读者参考。

1、即使我们已经在使用库函数作为开发工具了，我们可以忽略很多芯片的硬件外设使用上的细节，但当发生问题时，我们仍需要回归到芯片使用手册查看当前操作是否违规或缺漏。

2、使用 HAL 库和其它第三方的库开发类似，把我们需要编写的软件和第三方的库分开成相互独立的文件，开发过程中我们尽量不去修改第三方的软件源码，需要修改的部分尽量在自己的代码中实现；这样一旦我们需要更新第三方库时，我们原来编写的功能也能很快地匹配新的库去执行功能。

3、即使 HAL 库目前较以前已经相对更完善了，但它仍无法覆盖我们要想实现的所有细节功能，甚至可能存在错误，我们要有怀疑精神，辩证地去使用好这个工具；如我们在 PWM 一节编码时发现 HAL 库中有个宏定义 TIM_RESET_CAPTUREPOLARITY 括号不匹配导致编译报错，这时我们不得不修改一下 HAL 库的源码了。

4、注意 HAL 库的执行效率。由于 HAL 库的驱动对相同外设大多是可重入的，在执行 HAL 驱动的 API 函数的效率没有直接寄存器操作来得高，如果在对时序要求比较严苛的代码，建议使用简洁的寄存器操作代替。

5、我们在例程中使用 delay.c 中的延时函数取代 HAL_Delay();取消原来 HAL 库的 Systick 延时设置；但这会有一个问题：原来 HAL 库的超时处理机制不再适用，所以对于设置了超时的函数，可能会导致停留在这个函数的处理中，无法按正常的超时退出。

第八章 新建 HAL 版本 MDK 工程

在前面的章节我们介绍了 STM32F4xx 官方固件包的一些知识，本章我们将重点讲解新建 HAL 库版本的 MDK 工程的详细步骤。我们把本章新建好的工程放在光盘里，路径：**4，程序源码\1，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-3，新建工程实验-HAL 库版本**，大家在学习新建工程过程中间遇到一些问题，可以直接打开这个工程，然后对比学习。

本章将分为如下两个小节：

8.1 新建 HAL 库版本 MDK 工程

8.2 下载验证

8.1 新建 HAL 库版本 MDK 工程

本节我们将教大家如何新建一个 STM32F407 的 HAL 库版本 MDK5 工程。为了方便大家参考，我们将本节最终新建好的工程模板存放在“A 盘：4、程序源码\1，标准例程-HAL 库版本\实验 0 基础入门实验\实验 0-3，新建工程实验-HAL 库版本”，如遇新建工程问题，请打开该实验对比。

整个新建过程比较复杂，我们将其拆分为 5 个步骤进行讲解，请准备大概 2 个小时时间，耐心细致的做完！对你后续的学习非常有帮助！

在新建工程之前，首先我们要做如下准备：

1、 STM32Cube 官方固件包：我们使用的固件包版本是 STM32Cube_FW_F4_V1.26.0，固件包路径：A 盘→8，STM32 参考资料→1，STM32CubeF4 固件包。

2、开发环境搭建：参考本书第三章相关内容。

8.1.1 新建工程文件夹

新建工程文件夹分为 2 个步骤：1，新建工程文件夹；2，拷贝工程相关文件。

1. 新建工程文件夹

首先我们在桌面新建一个工程根目录文件夹，后续的工程文件都将在这个文件夹里建立，我们把这个文件夹重命名为：实验 0-3，新建工程实验-HAL 库版本。如图 8.1.1.1 所示：



图 8.1.1.1 新建工程根目录文件夹

为了让工程的文件目录结构更加清晰易懂，我们会在工程根目录文件夹下建立以下几个文件夹，每个文件夹名称及其作用如表 8.1.1.1 所示：

名称	作用
Drivers	存放与硬件相关的驱动层文件
Middlewares	存放正点原子提供的中间层组件文件和第三方中间层文件
Output	存放工程编译输出文件
Projects	存放 MDK 工程文件
User	存放 HAL 库用户配置文件、main.c、中断处理文件，以及分散加载文件

表 8.1.1.1 工程根目录新建文件夹及其作用

新建完成以后，最后得到我们的工程根目录文件夹如图 8.1.1.2 所示。

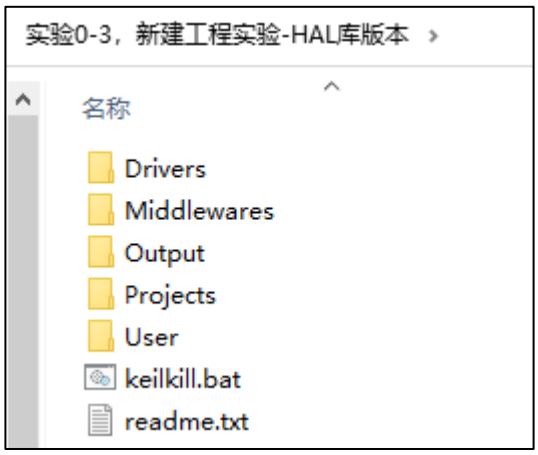


图 8.1.1.2 工程根目录文件夹

另外我们的工程根文件目录下还有一个名为 keilkill.bat 的可执行文件，双击便可执行。其作用是删除编译器编译后的无关文件，减少工程占用的内存，方便打包。还有一个名为 readme 的记事本文件，其作用是介绍本实验的各种信息。

工程根目录及其相关文件夹新建好以后，我们需要拷贝一些工程相关文件过来（主要是在 Drivers 文件夹里面），以便等下的新建工程需要。

2. 拷贝工程相关文件

接下来，我们按图 8.1.1.2 的根目录文件夹顺序介绍每个文件夹及其需要拷贝的文件。

Drivers 文件夹

该文件夹用于存放与硬件相关的驱动层文件，一般包括如表 8.1.1.2 所示的三个文件夹：

文件夹名称	作用
BSP	存放开发板板级支持包驱动代码，如各种外设驱动
CMSIS	存放 CMSIS 底层代码，如启动文件（.s 文件）、stm32f4xx.h 等
STM32F4xx_HAL_Driver	存放 HAL 库驱动代码源文件
SYSTEM	存放正点原子系统级核心驱动代码，如 sys.c、delay.c 和 usart.c

表 8.1.1.2 Drivers 包含文件夹

BSP 文件夹，用于存放正点原子提供的板级支持包驱动代码，如：LED、蜂鸣器、按键等。本章我们暂时用不到该文件夹，不过可以先建好备用。

CMSIS 文件夹，用于存放 CMSIS 底层代码（ARM 和 ST 提供），如：启动文件（.s 文件）、stm32f4xx.h 等各种头文件。该文件夹我们可以直接从 STM32CubeF4 固件包（路径：A 盘→8，STM32 参考资料→1，STM32CubeF4 固件包）里面拷贝，不过由于固件包里面的 CMISIS 兼容了太多芯片，导致非常大（170 MB），因此我们根据实际情况，对其进行了大幅精简，精简后的 CMSIS 文件夹大小为 1.6 MB 左右。精简后的 CMSIS 文件夹大家可以在：A 盘→4，程序源码→1，标准例程-HAL 库版本 文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

SYSTEM 文件夹，用于存放正点原子提供的系统级核心驱动代码，如：sys.c、delay.c 和 usart.c 等，方便大家快速搭建自己的工程。该文件同样可以从：A 盘→4，程序源码→1，标准例程-HAL 库版本 文件夹里面的任何一个实验的 Drivers 文件夹里面拷贝过来。

STM32F4xx_HAL_Driver 文件夹，用于存放 ST 提供的 F4xx HAL 库驱动代码。该文件夹我们可以直接从 STM32CubeF4 固件包里面拷贝。直接拷贝“STM32CubeF4 固件包→Drivers”路径下的 “STM32F4xx_HAL_Driver”文件夹到我们工程的 Drivers 下，只保留 Inc 和 Src 文件夹即可。

执行完以上操作后，Drivers 文件夹最终结构如图 8.1.1.3 所示：

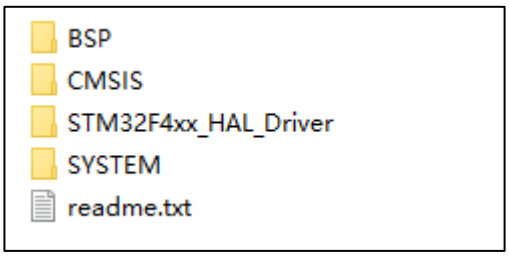


图 8.1.1.3 工程根目录下的 Drivers 文件夹

关于工程根目录下的 Drivers 文件操作到这里就完成了，可以说步骤是有点多。在此过程遇到问题的话，请大家多参考我们提供的实验 0-3，新建工程实验-HAL 库版本工程，一步步操作。

Middlewares 文件夹

该文件夹用于存放正点原子和其他第三方提供的中间层代码（组件/Lib 等），如：USMART、MALLOC、TEXT、FATFS、USB、LWIP、各种 OS、各种 GUI 等等。本章我们暂时用不到该文件夹，不过可以先建好备用，后面的实验将会陆续添加各种文件。

Output 文件夹

该文件夹用于存放编译器编译工程输出的中间文件，比如：.hex、.bin、.o 文件等等。这里不需要操作，后面只需要在 MDK 里面设置该文件夹为编译过程中间文件的存放文件夹就行。

Projects 文件夹

该文件夹用于存放编译器（MDK、IAR 等）工程文件，我们主要用 MDK，为了方便区分，我们在该文件夹下新建：MDK-ARM 文件夹，用于存放 MDK 的工程文件，如图 8.1.1.4 所示：



图 8.1.1.4 在 Projects 文件夹下新建 MDK-ARM 文件夹

User 文件夹

User 文件夹用于存放 HAL 库用户配置文件、main.c、中断处理文件，以及分散加载文件。

我们首先从官方固件包里面直接拷贝官方的模板工程下的 HAL 库用户配置文件和中断处理文件到我们的 User 文件夹里。官方的模板工程路径：STM32Cube_FW_F4_V1.26.0\Projects\STM324xG_EVAL\Templates，打开 Template_Project 文件夹，如图 8.1.1.8 所示。

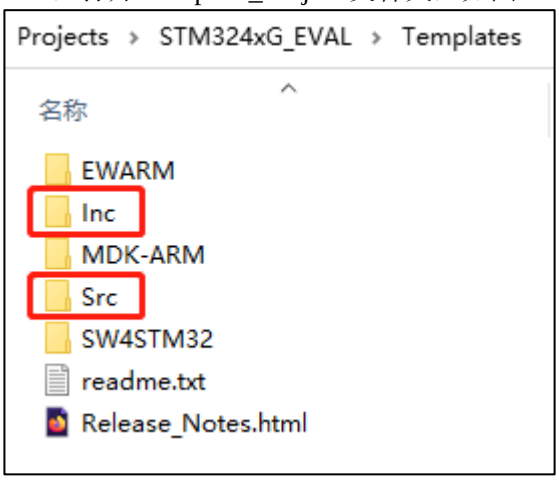


图 8.1.1.8 官方模板工程根目录

我们需要的文件就在 Inc 和 Src 文件夹里面，在这两个文件夹里面找到：stm32f4xx_it.c、stm32f4xx_it.h、stm32f4xx_hal_conf.h 这三个文件，并且拷贝到我们的 User 文件夹下。

main.c 文件我们也是放在 User 文件夹里面的，后面在 MDK 里面教大家新建.c 文件并保存。

8.1.2 新建一个工程框架

首先，打开 MDK 软件。然后点击 Project→New uVision Project 如图 8.1.2.1 所示：

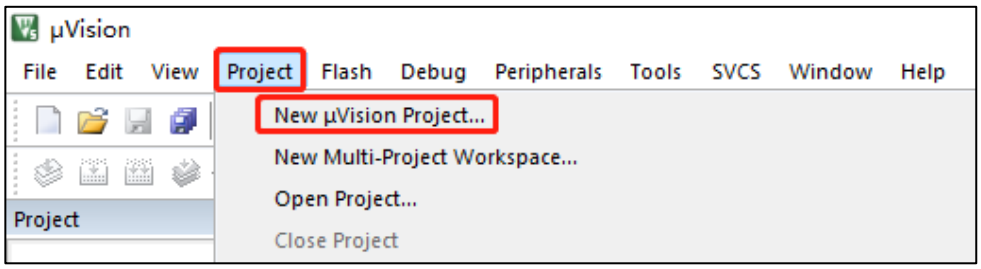
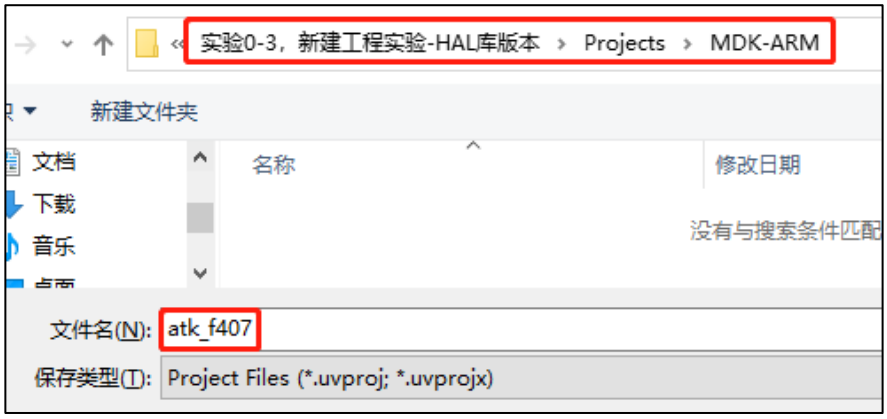


图 8.1.2.1 新建 MDK 工程

然后弹出工程命名和保存的操作窗口，我们将工程文件保存路径设置在上一节新建的工程文件夹内，具体路径为：F 盘→实验 0-3，新建工程实验-HAL 库版本→Projects→MDK-ARM，工程名字我们取：atk_f407，最后点击保存即可。具体操作窗口如图 8.1.2.2 所示：



之后，弹出器件选择对话框，如图 8.1.2.3 所示。因为 DMF407 所使用的 STM32 型号为 STM32F407IGT6，所以我们选择：STMicroelectronics→STM32F4 Series→STM32F407→STM32F407IGTx（如果使用的是其他系列的芯片，选择相应的型号就可以了，特别注意：**一定要安装对应的器件 pack** 才会显示这些内容!!如果没得选择，请关闭 MDK，然后安装 A 盘：6，软件资料\1，软件\1，MDK\ Keil.STM32F4xx_DFP.2.8.0.pack 这个安装包后重试）。

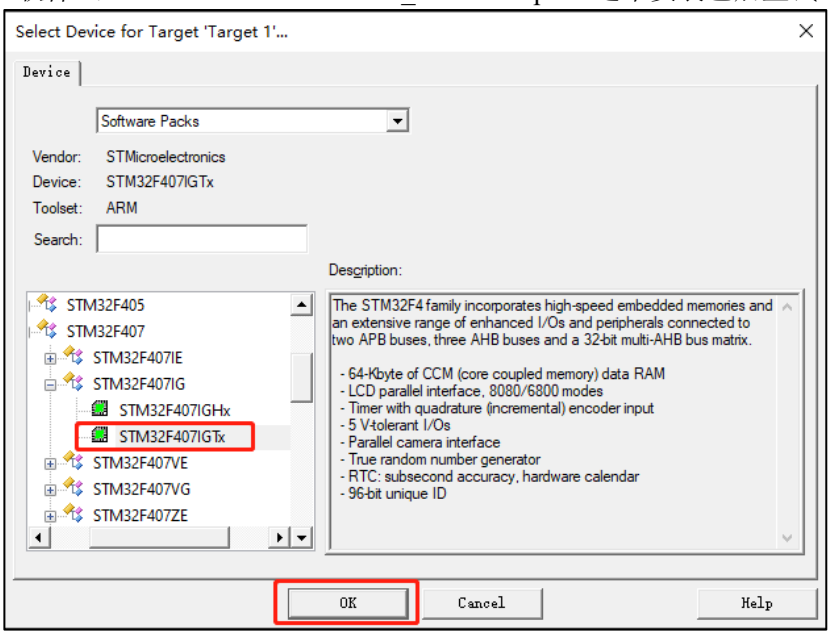


图 8.1.2.3 器件选择界面

点击 OK，MDK 会弹出 Manage Run-Time Environment 对话框，如图 8.1.2.4 所示：

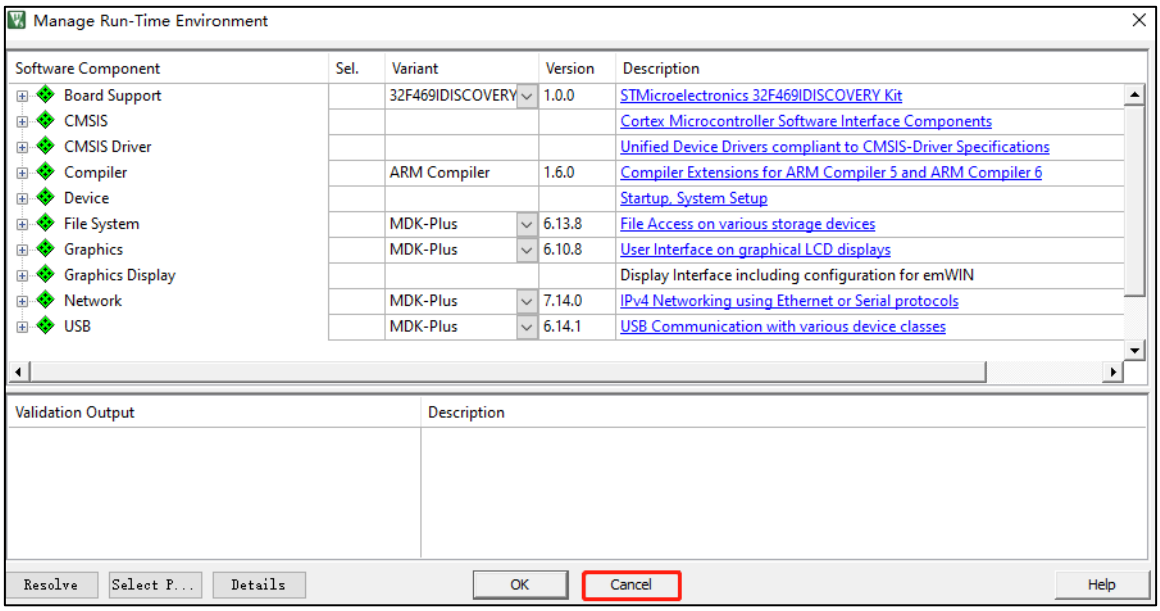


图 8.2.1.4 Manage Run-Time Environment 界面

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 8.1.2.4 所示界面，我们直接点击 Cancel，即可，得到如图 8.1.2.5 所示界面：

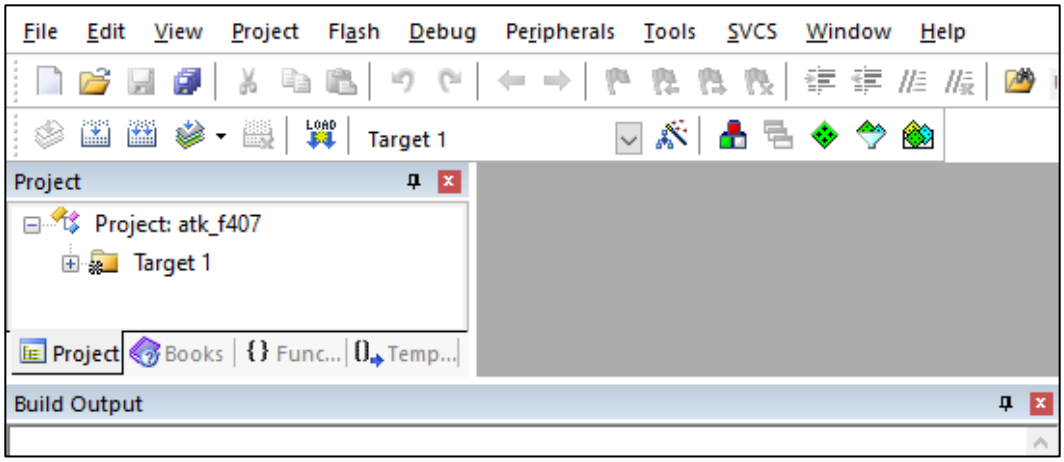


图 8.1.2.5 工程初步建立

此时，我们打开 MDK-ARM 文件夹，会看到 MDK 在该文件夹下自动创建了 3 个文件夹（DebugConfig、Listings 和 Objects），如图 8.1.2.6 所示：



图 8.1.2.6 MDK 新建工程时自动创建的文件夹

这三个文件夹的作用如表 8.1.2.1 所示：

文件夹	作用
DebugConfig	用于存放调试设置信息文件（.dbgconf），不可删除！
Listings	用于存放编译过程产生的链接列表等文件

Objects	用于存放编译过程产生的调试信息、.hex、预览、.lib 文件等

表 8.1.2.1 三个文件夹及其作用

编译过程产生的链接列表、调试信息、预览、lib 等文件，统称为中间文件。为了统一管理，方便使用，我们会把输出在 Listings 和 Objects 文件夹的内容，统一改为输出到 Output 文件夹（通过魔术棒设置），我们先**把 MDK 自动生成的这两个文件夹（Listings 和 Objects）删除**。

至此，我们还只是建了一个框架，还有好几个步骤要做，比如添加文件、魔术棒设置、编写 main.c 等。

8.1.3 添加文件

本节将分 5 个步骤：1，设置工程名和分组名；2，添加启动文件；3，添加 SYSTEM 源码 4，添加 User 源码；5，添加 STM32F4xx_HAL_Driver 源码。

1. 设置工程名和分组名

在 Project→Target 上右键，选择 Manage Project Items…（方法一）或在菜单栏点击品字形红绿白图标（方法二）进入工程管理界面，如图 8.1.3.1 所示：

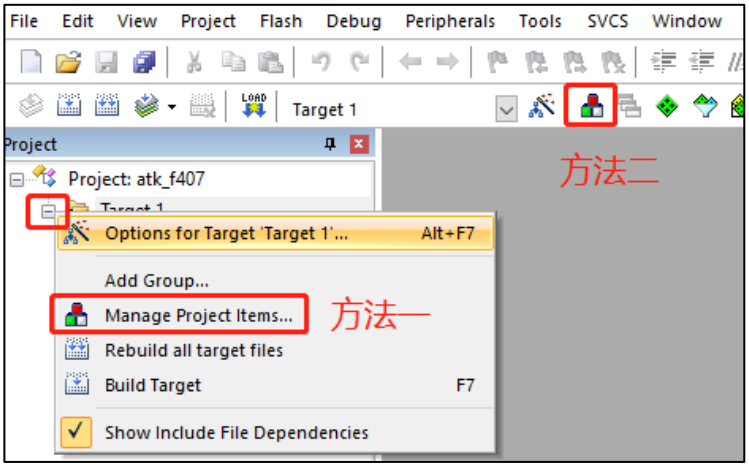


图 8.1.3.1 进入工程管理界面

在工程管理界面，我们可以执行设置工程名字（Project Targets）、分组名字（Groups）以及添加每个分组的文件（Files）等操作。我们设置工程名字为：Template，并设置四个分组：Startup（存放启动文件）、User（存放 main.c 等用户代码）、Drivers/SYSTEM（存放系统级驱动代码）Driver/STM32F4xx_HAL_Driver（存放 HAL 库代码）、Readme（存放工程说明文件），如图 8.1.3.2 所示：

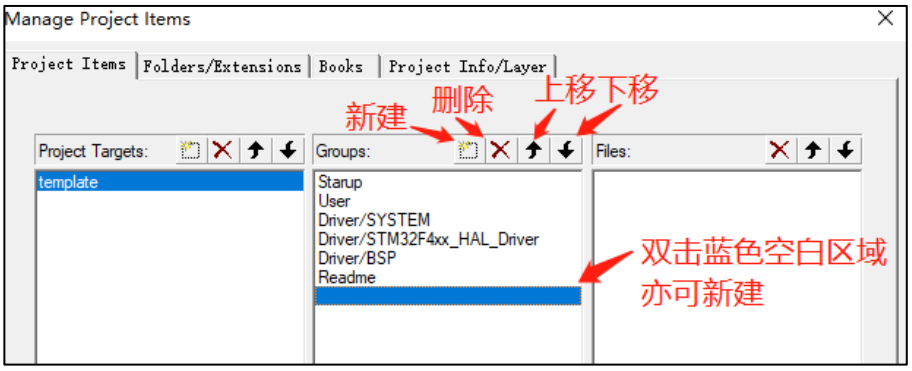


图 8.1.3.2 设置工程名和分组名

设置好之后，我们点击 OK，回到 MDK 主界面，可以看到我们设置的工程名和分组名如图 8.1.3.3 所示：

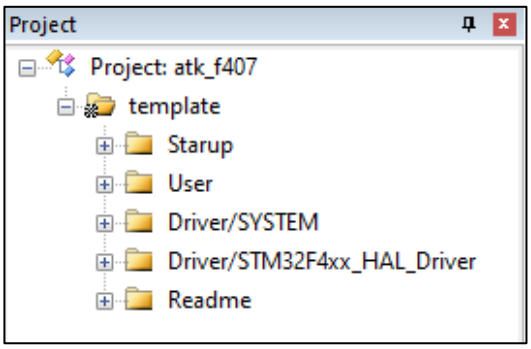


图 8.1.3.3 设置成功

这里我们只是新建了一个简单的工程，并没有添加 BSP、Middlewares 等分组，后面随着工程复杂程度的增加，我们需要一步步添加对应的分组。

注意：为了让工程结构清晰，我们会尽量让 MDK 的工程分组和我们前面新建的工程文件夹对应起来，由于 MDK 分组不支持多级目录，因此我们将路径也带入分组命名里面，以便区分。如：User 分组对应 User 文件夹里面的源码，Drivers/SYSTEM 分组，对应 Drivers/SYSTEM 文件夹里面的源码，Drivers/BSP 分组对应 Drivers/BSP 文件夹里面的源码等。

2. 添加启动文件

启动文件（.s 文件）包含 STM32 的启动代码，其主要作用包括：1、堆栈（SP）的初始化；2、初始化程序计数器（PC）；3、设置向量表异常事件的入口地址；4、调用 main 函数等，是每个工程必不可少的一个文件，我们在本书第九章会有详细介绍。

启动文件由 ST 官方提供，存放在 STM32CubeF4 软件包的：Drivers→CMSIS→Device→ST→STM32F4xx→Source→Templates→arm 文件夹下。因为我们开发板使用的是 STM32F407IGT6，对应的启动文件为：startup_stm32f407xx.s，为了节省空间，在精简版 CMSIS 文件夹里面我们把其他启动文件都删了。

关于启动文件的说明，我们就介绍这么多，接下来我们看如何添加启动文件到工程里面。我们有两种方法给 MDK 的分组添加文件：1，双击 Project 下的分组名添加。2，进入工程管理界面添加。

这里我们使用方法 1 添加（路径：实验 0-3，新建工程实验-HAL 库版本\Drivers\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm），如图 8.1.3.4 所示：

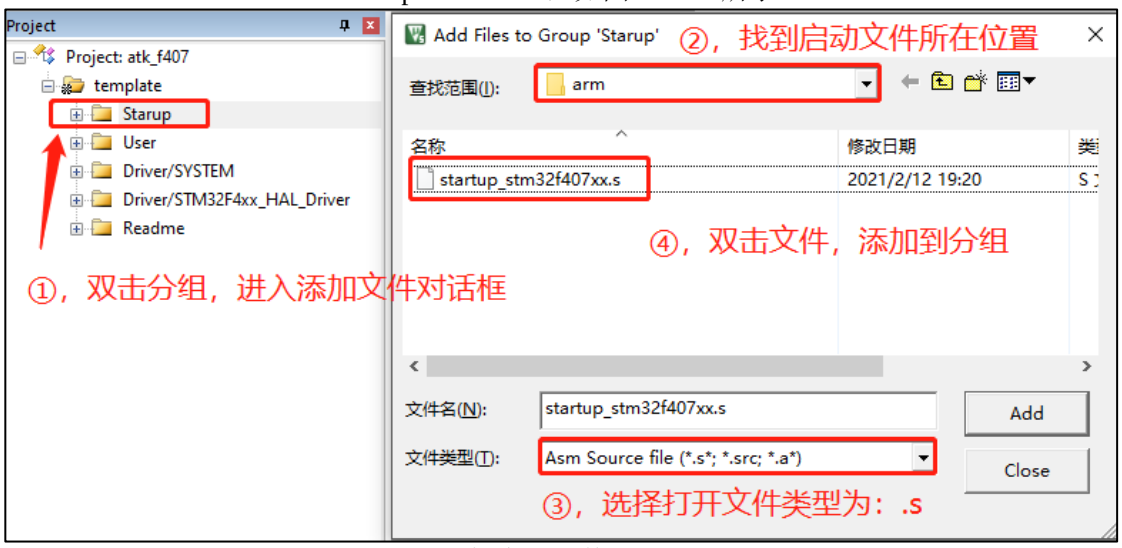


图 8.1.3.4 双击分组添加启动文件（startup_stm32f407xx.s）

上图中，我们也可以点击 Add 按钮进行文件添加。添加完后，点击 Close，完成启动文件添加，得到工程分组如图 8.1.3.5 所示：

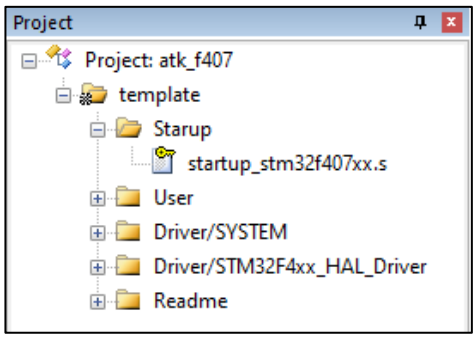


图 8.1.3.5 启动文件添加成功

3. 添加 SYSTEM 源码

这里我们在工程管理界面（方法 2）进行 SYSTEM 源码添加。点击：按钮，进入工程管理界面，选中 Drivers/SYSTEM 分组，然后点击：Add Files，进入文件添加对话框，依次添加 delay.c、sys.c 和 usart.c 到该分组下，如图 8.1.3.6 所示：

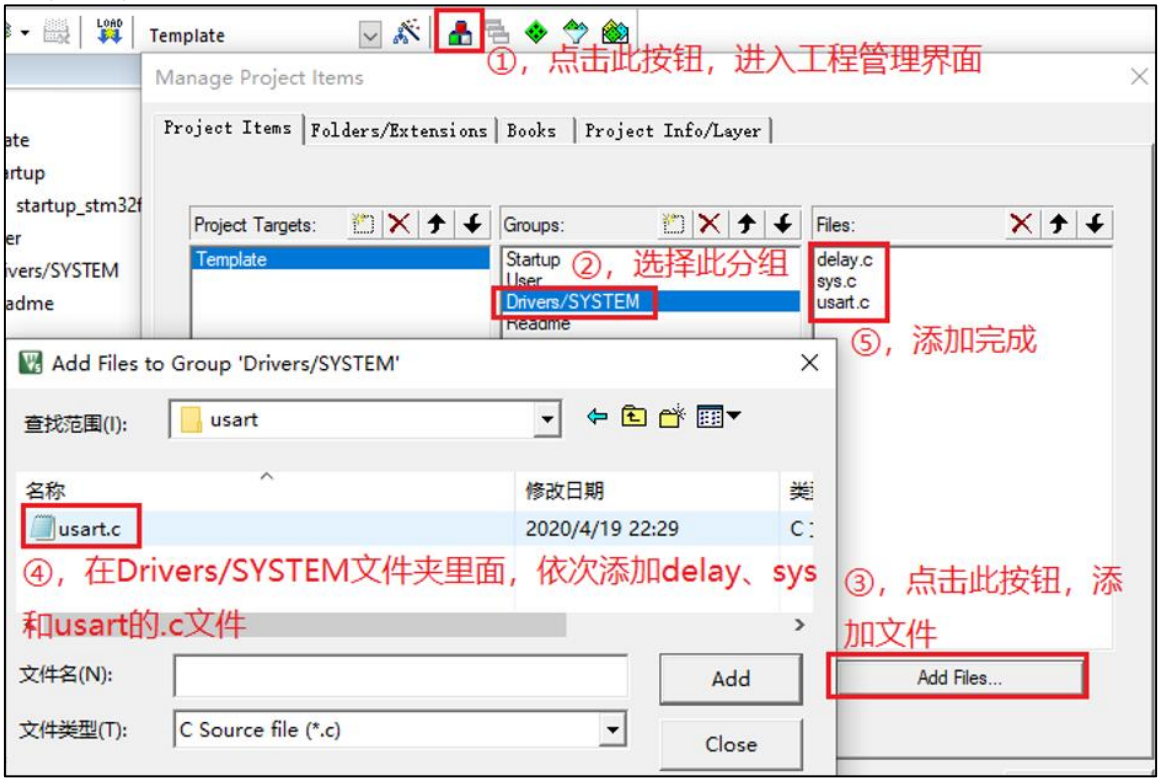


图 8.1.3.6 添加 SYSTEM 源码

注意：这些源码都是在第 8.1.1 小节的第二步拷贝过来的，如果之前没拷贝，是找不到这些源码的。添加完成后，如图 8.1.3.7 所示：

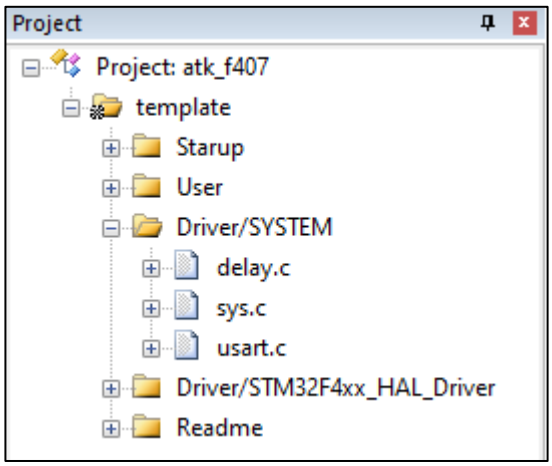


图 8.1.3.7 SYSTEM 源码添加完成

4. 添加 User 源码

这里我们在工程管理界面（方法 2）进行 User 源码添加。点击：按钮，进入工程管理界面，选中 User 分组，然后点击：Add Files，进入文件添加对话框，依次添加 stm32f4xx_it.c 和 system_stm32f4xx.c 到该分组下，如图 8.1.3.8 所示：

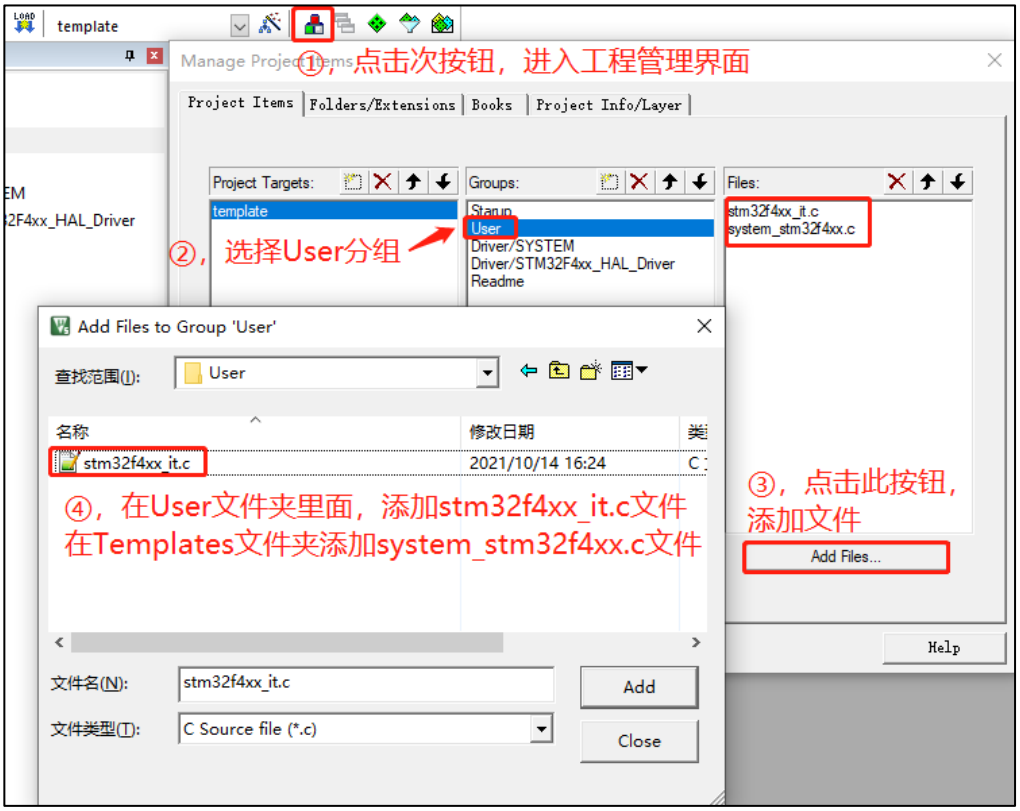


图 8.1.3.8 添加 User 源码

注意：这些源码都是在第 8.1.1 小节的第二步拷贝过来的，如果之前没拷贝，是找不到这些源码的。添加完成后，如图 8.1.3.9 所示：

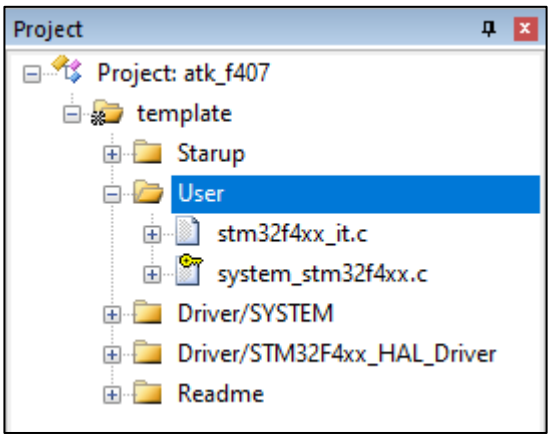


图 8.1.3.9 User 源码添加完成

5. 添加 STM32F4xx_HAL_Driver 源码

接下来我们往 Drivers/STM32F4xx_HAL_Driver 分组里添加文件。点击：按钮，进入工程管理界面，选中 Drivers/STM32F4xx_HAL_Driver 分组，然后点击：Add Files，进入文件添加对话框，依次添加 stm32f4xx_hal.c、stm32f4xx_hal_cortex.c、stm32f4xx_hal_dma.c、stm32f4xx_hal_dma_ex.c、stm32f4xx_hal_gpio.c、stm32f4xx_hal_pwr.c、stm32f4xx_hal_pwr_ex.c、stm32f4xx_hal_rcc.c、stm32f4xx_hal_rcc_ex.c、stm32f4xx_hal_uart.c、stm32f4xx_hal_usart.c 到该分组下，如图 8.1.3.10 所示：

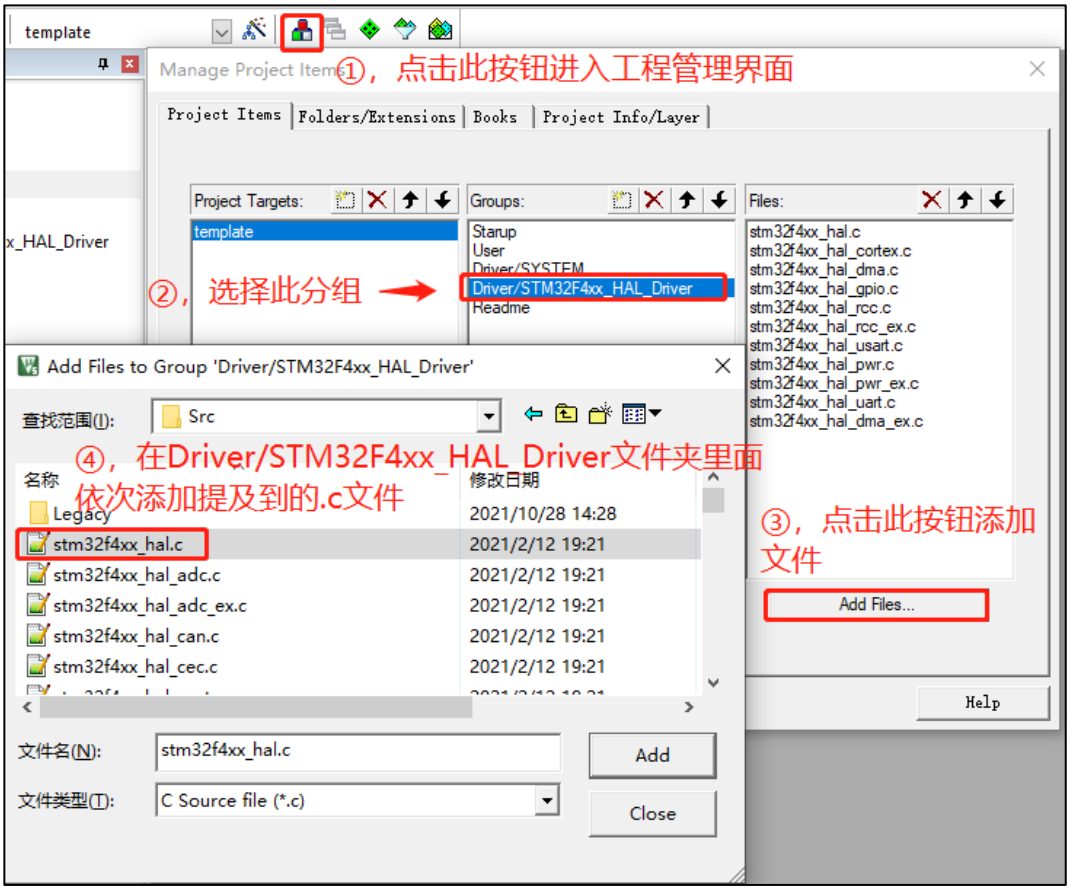


图 8.1.3.10 添加 STM32F4xx_HAL_Driver 源码

添加完成后，如图 8.1.3.11 所示：

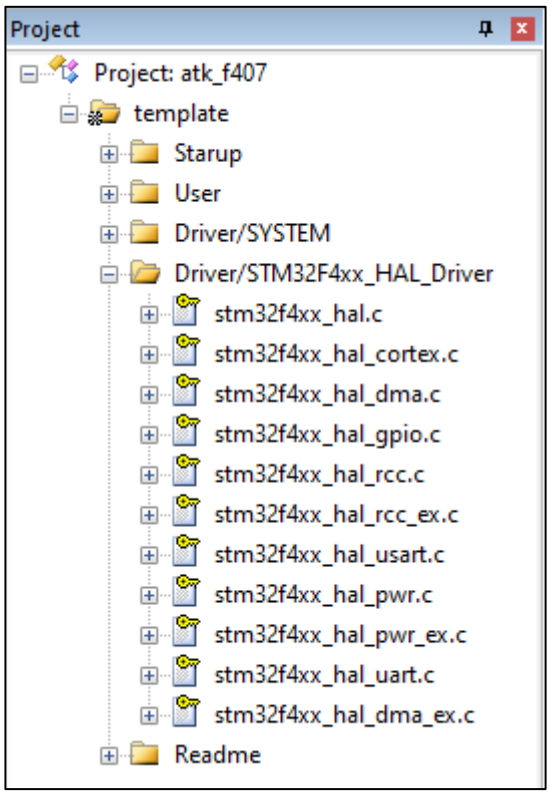


图 8.1.3.11 STM32F4xx_HAL_Driver 源码添加完成

可以看到分组中有些.c 文件有个小钥匙的符号，这是因为官方的固件包的文件设置了只读权限，我们取消只读权限就好了，方法如图 8.1.2.12 所示。

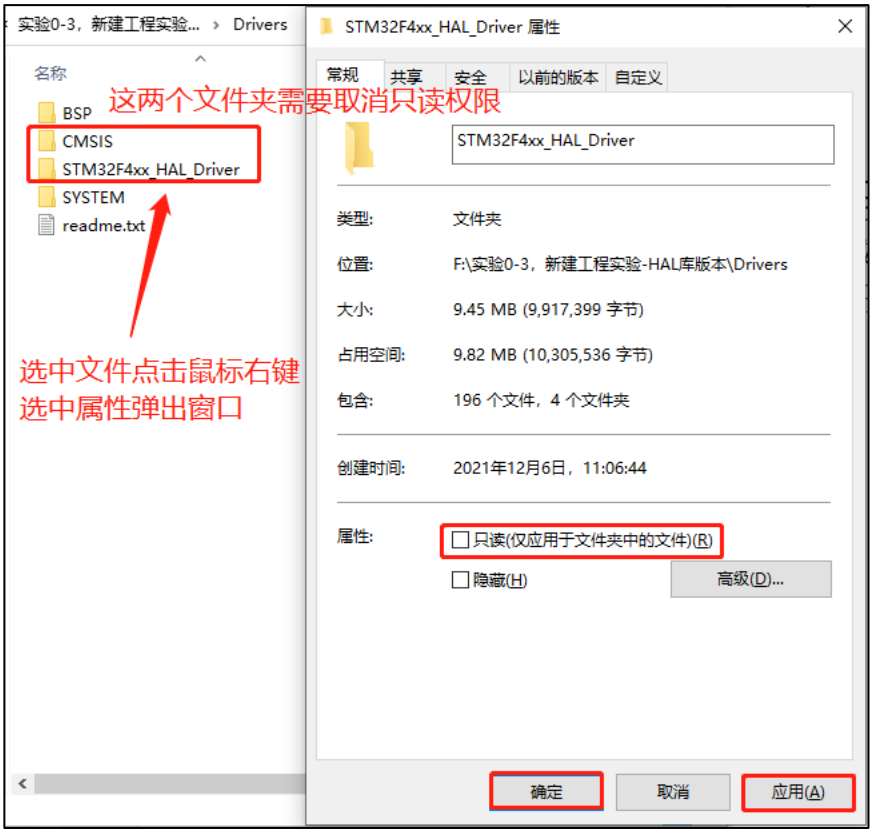


图 8.1.2.12 取消工程文件夹的只读权限

8.1.4 魔术棒设置

为避免编写代码和编译报错，我们需要通过魔术棒对 MDK 工程进行相关设置。在 MDK 主界面，点击：（魔术棒图标，即 Options for Target 按钮），进入工程设置对话框，我们将进行如下几个选项卡的设置。

1. 设置 Target 选项卡

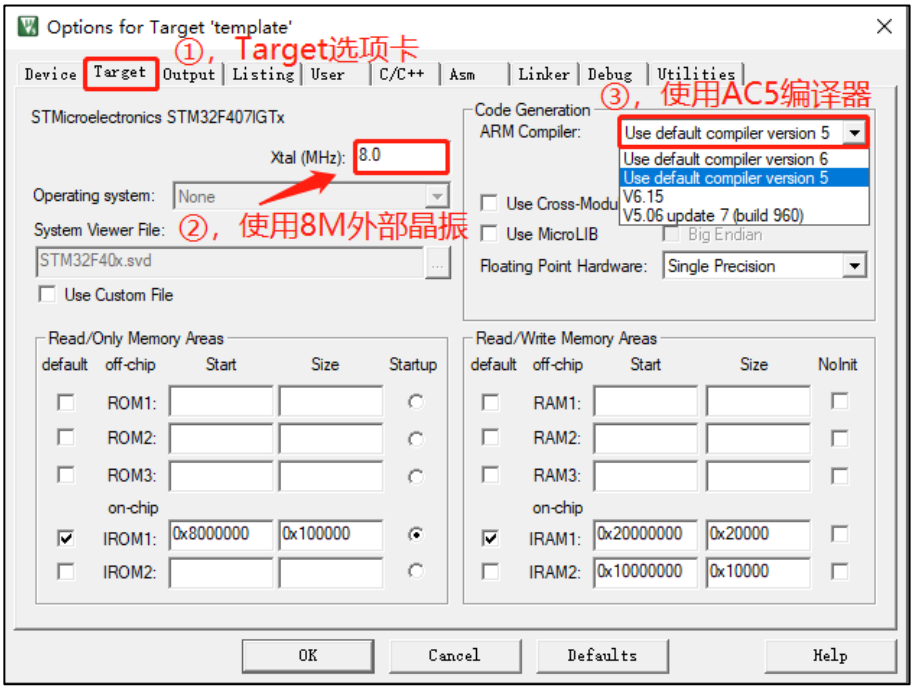


图 8.1.4.1 Target 选项卡设置

上图中，我们设置芯片所使用的外部晶振频率为 8Mhz，选择 ARM Compiler 版本为：Use default compiler version 5（即 AC5 编译器）。

这里我们说明一下 AC5 和 AC6 编译的差异，如表 8.1.4.2 所示：

对比项	AC5	AC6	说明
中文支持	较好	较差	AC6 对中文支持极差，goto definition 无法使用，误报等
代码兼容性	较好	较差	AC6 对某些代码优化可能导致运行异常，需慢慢调试
编译速度	较慢	较快	AC6 编译速度比 AC5 快
语法检查	一般	严格	AC6 语法检查非常严格，代码严谨性较好

表 8.1.4.1 AC5&AC6 简单对比

由于 AC5 对中文支持比较好，且兼容性相对好一点，**为了避免不必要的麻烦，我们推荐大家使用 AC5 编译器**。为了让大家自由选择，我们正点原子的源码，也是支持 AC6 编译器的，不过在选项卡设置上稍有差异，具体差异如表 8.1.4.2 所示：

选项卡	AC5	AC6	说明
Target	选择 AC5 编译器	选择 AC6 编译器	选择对应的编译器
C/C++	Misc Controls 无需设置	Misc Controls 设置： -Wno-invalid-source-encoding	AC6 需设置编译选项以关闭对汉字的错误警告，AC5 则不要

表 8.1.4.2 AC5&AC6 设置差异

2. 设置 Output 选项卡

在魔术棒→Output 选项卡里面，进行如图 8.1.4.2 所示设置：

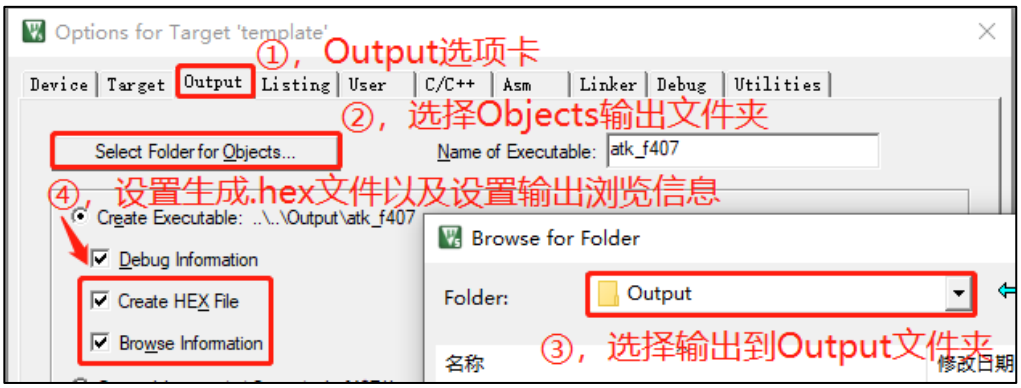


图 8.1.4.2 设置 Output 选项卡

注意，我们勾选：Browse Information，用于输出浏览信息，这样就可以使用 go to definition 查看函数/变量的定义，对我们后续调试代码比较有帮助，如果不需要调试代码，则可以去掉这个勾选，以提高编译速度。

3. 设置 Listing 选项卡

在魔术棒→Listing 选项卡里面，进行如图 8.1.4.3 所示设置：

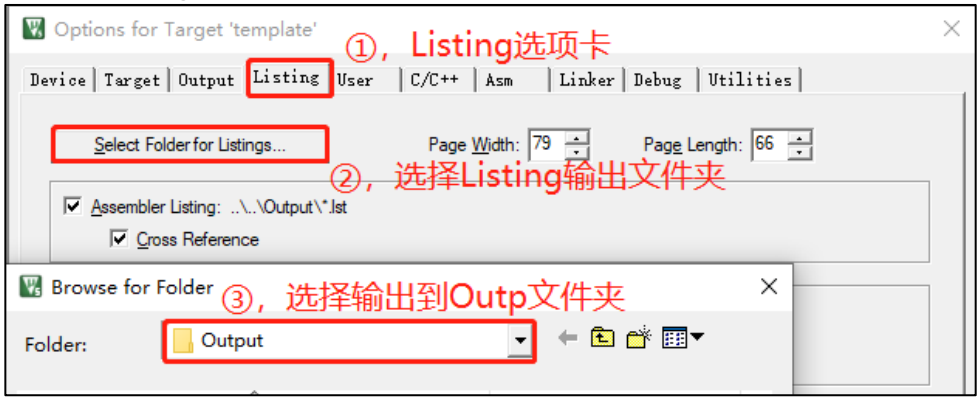


图 8.1.4.3 设置 Listing 选项卡

经过 Output 和 Listing 这两步设置，原来存储在 Objects 和 Listings 文件夹的内容（中间文件）就都改为输出到 Output 文件夹了。

4. 设置 C/C++选项卡

在魔术棒→C/C++选项卡里面，进行如图 8.1.4.4 所示设置：

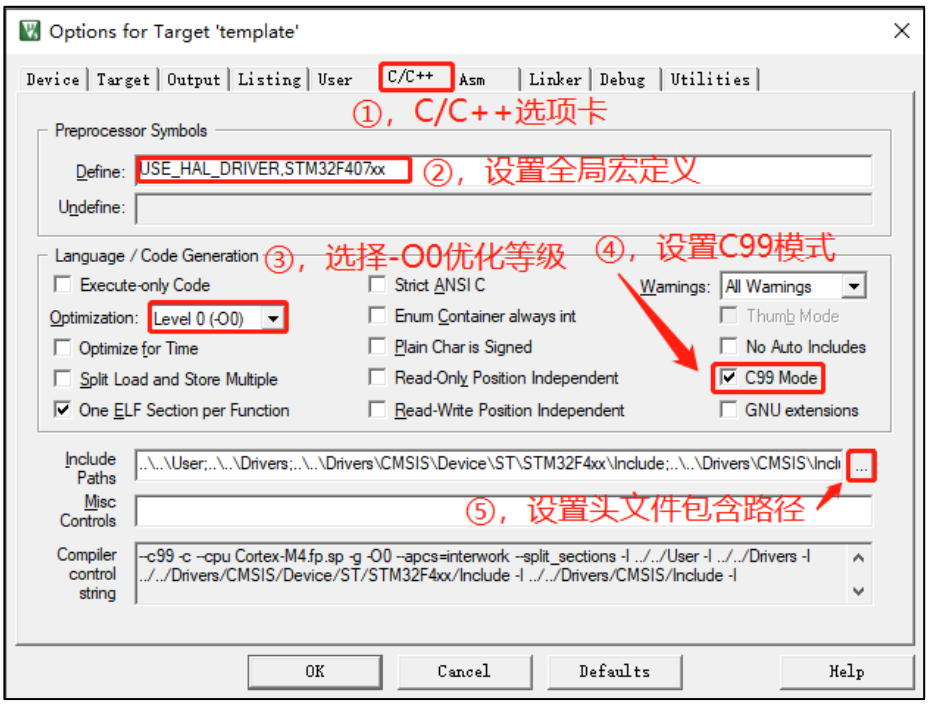


图 8.1.4.4 设置 C/C++选项卡

在②处设置了全局宏定义：STM32F407xx，用于定义所用 STM32 型号，在 stm32f4xx.h 里面会用到该宏定义。

在③处设置了优化等级为-O0，可以得到最好的调试效果，当然为了提高优化效果提升性能并降低代码量，可以设置-O1~-O3，数字越大效果越明显，不过也越容易出问题。注意：当使用 AC6 编译器的时候，这里推荐默认使用-O1 优化。

在④处勾选 C99 模式，即使用 C99 C 语言标准。

在⑤处，我们可以进行头文件包含路径设置，点击此按钮，进行如图 8.1.4.5 所示设置：

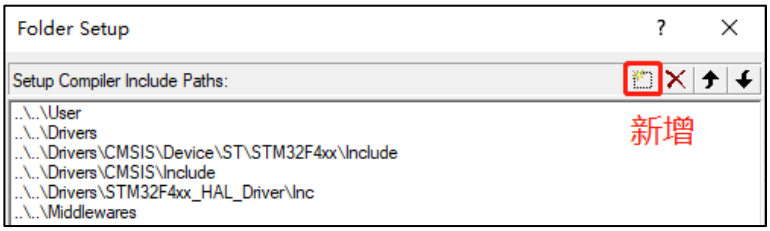


图 8.1.4.5 设置头文件包含路径

上图中我们设置了 6 个头文件包含路径，其中 3 个在 Drivers 文件夹下，一个在 User 文件夹下，一个在 Middlewares 文件夹下。为避免频繁设置头文件包含路径，正点原子最新源码的 include 全部使用相对路径，也就是我们只需要在头文件包含路径里面指定一个文件夹，那么该文件夹下的其他文件夹里面的源码，如果全部是使用相对路径，则无需再设置头文件包含路径了，直接在 include 里面就指明了头文件所在。

关于相对路径，这里大家记住 3 点：

1，默认路径就是指 MDK 工程所在的路径，即.uvprojx 文件所在路径（文件夹）

2，“./”表示当前目录（相对当前路径，也可以写做“.\”）

3，“../”表示当前目录的上一层目录（也可以写做“..\”）

举例来说，上图中：..\..\Drivers\CMSIS\Device\ST\STM32F4xx\Include，前面两个“..\”，表示 Drivers 文件夹在当前 MDK 工程所在文件夹（MDK-ARM）的上 2 级目录下，具体解释如图 8.1.4.6 所示：

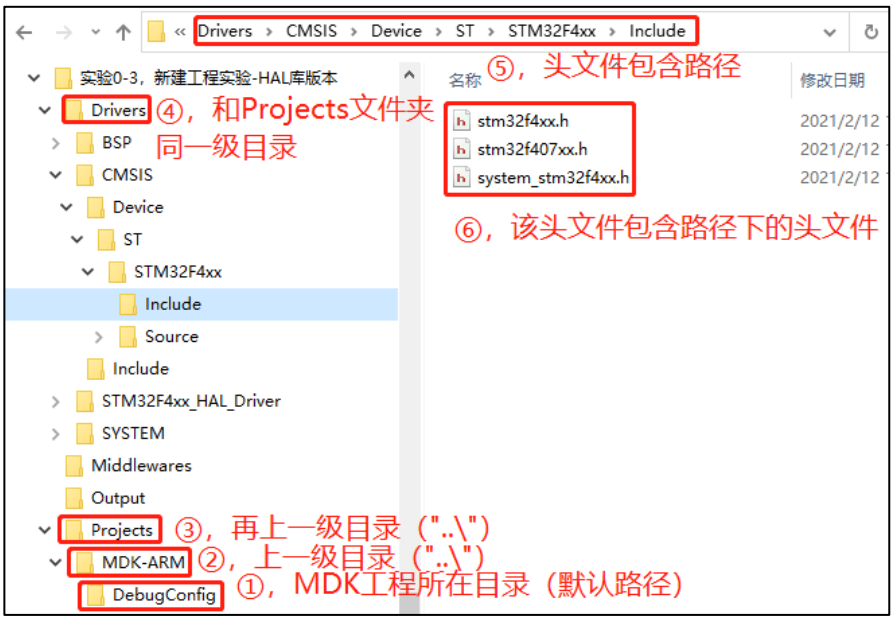


图 8.1.4.6 ..\..\Drivers\CMSIS\Device\ST\STM3F4xx\Include 的解释

上图表示根据头文件包含路径：..\..\Drivers\CMSIS\Device\ST\STM32F4xx\Include，编译器可以找到⑥处所包含的这些头文件，即代码里面可以直接 include 这些头文件使用。

再举个例子，在完成如图 6.1.4.5 所示的头文件包含路径设置以后，我们在代码里面编写：

```
#include "./SYSTEM/sys/sys.h"
```

即表示当前头文件包含路径所指示的 4 个文件夹里面，肯定有某一个文件夹包含了：SYSTEM/sys/sys.h 的路径，实际上就是在 Drivers 文件夹下面，两者结合起来就相当于：

```
#include "../../Drivers/SYSTEM/sys/sys.h"
```

这就是相对路径。它既可以减少头文件包含路径设置（即减少 MDK 配置步骤，免去频繁设置头文件包含路径的麻烦），同时又可以很方便的知道头文件具体在那个文件夹，因此我们推荐在编写代码的时候使用相对路径。

关于相对路径，我们就介绍这么多，大家搞不明白的可以在网上搜索相关资料学习，也可以在后面的学习，分析我们其他源码，慢慢体会，总之不难，但是好用。

最后，我们如果使用 AC6 编译器，则在图 6.1.4.4 的 Misc Controls 处需要设置：-Wno-invalid-source-encoding，避免中文编码报错，如果使用 AC5 编译器，则不需要该设置！！

5. 设置 Debug 选项卡

在魔术棒→Debug 选项卡里面，进行如图 8.1.4.7 所示设置：

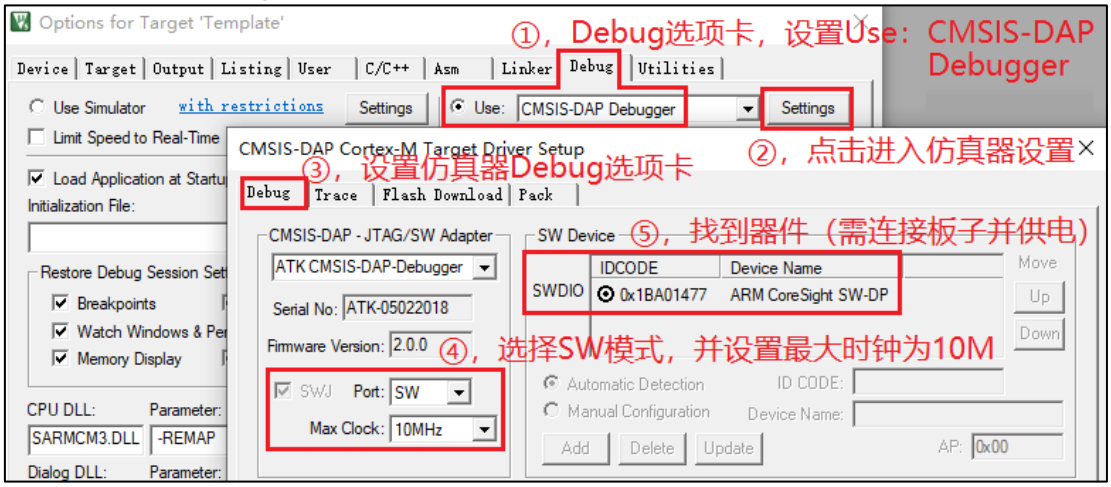


图 8.1.4.7 Debug 选项卡设置

图中，我们选择使用：CMSIS-DAP 仿真器，使用 SW 模式，并设置最大时钟频率为 10Mhz，以得到最高下载速度。当我们将仿真器和开发板连接好，并给开发板供电以后，仿真器就会找到开发板芯片，并在 SW Device 窗口显示芯片的 IDCODE、Device Name 等信息（图中⑤处），当无法找到时，请检查供电和仿真器连接状况。

6. 设置 Utilities 选项卡

在魔术棒→Debug 选项卡里面，进行如图 8.1.4.8 所示设置：

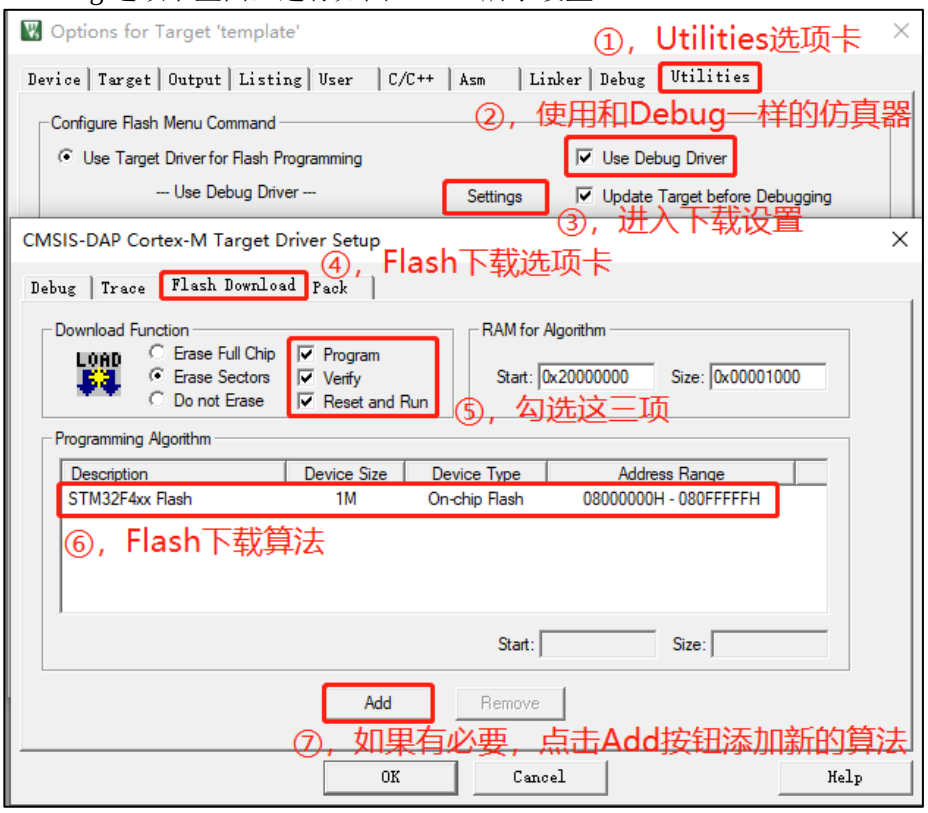


图 8.1.4.8 Utilities 选项卡设置

图中⑥处下载算法是 MDK 默认添加的，针对 STM32F4xx 系列产品。一般我们用这个即可，如果⑥处没有下载算法，则点击 Add 按钮，添加下载算法即可（算法名字和⑥处的算法名字一样）。

8.1.5 添加 main.c，并编写代码

在 MDK 主界面，点击：，新建一个 main.c 文件，并保存在 User 文件夹下。然后双击 User 分组，弹出添加文件的对话框，将 User 文件夹下的 main.c 文件添加到 User 分组下。得到如图 8.1.5.1 所示的界面：

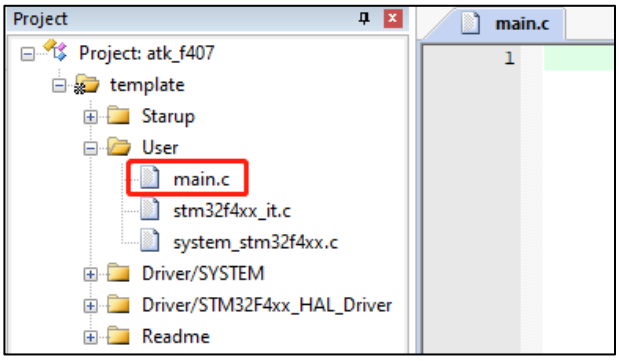


图 8.1.5.1 在 User 分组下加入 main.c 文件

至此，我们就可以开始编写我们自己的代码了。我们在 main.c 文件里面输入如下代码：

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"

void led_init(void);                    /* LED 初始化函数声明 */

int main(void)
{
    HAL_Init();                         /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                    /* 延时初始化 */
    led_init();                         /* 初始化 LED */

    while(1)
    {
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_0, GPIO_PIN_RESET);   /* LED0 亮 */
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_SET);     /* LED1 灭 */
        delay_ms(500);

        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_0, GPIO_PIN_SET);     /* LED0 灭 */
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_RESET);   /* LED1 亮 */
        delay_ms(500);
    }
}

/**
 * @brief       初始化 LED 相关 IO 口，并使能时钟
 * @param       无
 * @retval      无
 */
void led_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    __HAL_RCC_GPIOE_CLK_ENABLE();                           /* LED0，LED1 时钟使能 */

    gpio_init_struct.Pin = GPIO_PIN_0;                      /* LED0 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
    HAL_GPIO_Init(GPIOE, &gpio_init_struct);                /* 初始化 LED0 引脚 */

    gpio_init_struct.Pin = GPIO_PIN_1;                      /* LED1 引脚 */
    HAL_GPIO_Init(GPIOE, &gpio_init_struct);                /* 初始化 LED1 引脚 */

    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_0, GPIO_PIN_SET);/* LED0 灭 */
    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_1, GPIO_PIN_SET);/* LED1 灭 */
}
```

此部分代码，在 A 盘→4，程序源码→1，标准例程-HAL 库版本→ 实验 0 基础入门实验→实验 0-3，新建最工程实验-HAL 库版本→User→main.c 里面有，大家可以自己输入，也可以直接拷贝。强烈建议自己输入，以加深对程序的理解和印象!!

注意，这里的 include 就是使用的相对路径，关于相对路径，请参考前面 C/C++选项卡设置章节进行学习。

```
Build Output
assembling startup_stm32f407xx.s...
compiling stm32f4xx_it.c...
..\..\User\stm32f4xx_it.c(25): error:  #5: cannot open source input file "main.h": No such file or directory
  #include "main.h"
..\..\User\stm32f4xx_it.c: 0 warnings, 1 error
compiling stm32f4xx_hal_dma.c...
compiling stm32f4xx_hal_gpio.c...
compiling stm32f4xx_hal_pwr.c...
compiling system_stm32f4xx.c...
compiling stm32f4xx_hal.c...
compiling stm32f4xx_hal_cortex.c...
compiling stm32f4xx_hal_pwr_ex.c...
compiling sys.c...
compiling stm32f4xx_hal_rcc_ex.c...
compiling main.c...
compiling usart.c...
compiling delay.c...
compiling stm32f4xx_hal_usart.c...
compiling stm32f4xx_hal_rcc.c...
compiling stm32f4xx_hal_uart.c...
compiling stm32f4xx_hal_dma_ex.c...
"..\..\Output\atk_f407.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:06
```

双击

图 8.1.5.2 编译结果

编译结果可以看到 1 个错误，0 个警告。这个错误说找不到 main.h，因为我们也不需要用到 main.h，双击这个错误会弹出下面的 stm32f4xx_it.c 文件对应包含 main.h 的语句。我们只需要把它删除，然后重新编译，如图 8.1.5.3 所示。



图 8.1.5.3 删除包含 main.h 的语句

编译后发现又有一个警告，警告 HAL_IncTick 函数没有声明，如图 8.1.5.4 所示。

```
Build Output
Build started: Project: atk_f407
*** Using Compiler 'V5.06 update 7 (build 960)', folder: 'D:\Keil_v5\ARM\ARMCC\Bin'
Build target 'template'
compiling stm32f4xx_it.c...
..\..\User\stm32f4xx_it.c(139): warning:  #223-D: function "HAL_IncTick" declared implicitly
    HAL_IncTick();
..\..\User\stm32f4xx_it.c: 1 warning, 0 errors
linking...
Program Size: Code=5514 RO-data=450 RW-data=28 ZI-data=1900
FromELF: creating hex file...
"..\..\Output\atk_f407.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:00:00
```

图 8.1.5.4 编译报警告

因为这个函数是在 stm32f4xx_hal.c 定义了，并且在 stm32f4xx_hal.h 声明了。我们把 stm32f4xx_hal.h 包含进来即可。这里还有一个原因是整个工程没有包含 stm32f4xx_hal.h 的语句，我们需要用到它，所以在这里把它包含进来。官方的 main.h 是有包含这个头文件的。我们不用 main.h 文件，我们在 stm32f4xx_it.c 文件刚才删除包含 main.h 的语句的位置，编写包含 stm32f4xx_hal.h 语句，如图 8.1.5.5 所示。

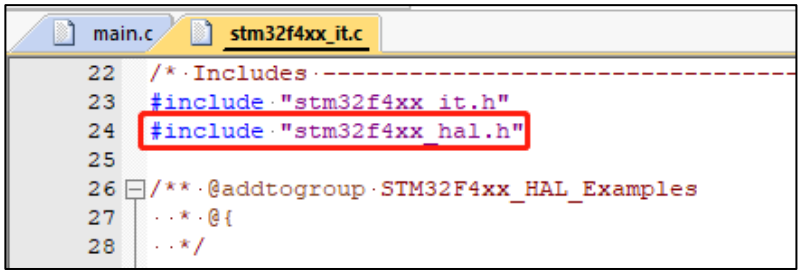


图 8.1.5.5 包含 stm32f4xx_hal.h 头文件到工程

再进行编译就会发现 0 错误 0 警告，结果如图 8.1.5.6 所示。

```
Build Output
Build started: Project: atk_f407
*** Using Compiler 'V5.06 update 7 (build 960)', folder: 'D:\Keil_v5\ARM\ARMCC\Bin'
Build target 'template'
compiling stm32f4xx_it.c...
linking...
Program Size: Code=5514 RO-data=450 RW-data=28 ZI-data=1900
FromELF: creating hex file...
"..\..\Output\atk_f407.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

图 8.1.5.6 编译结果

编译结果提示：代码总大小（Porgram Size）为：FLASH 占用 5992 字节（Code＋RO＋RW），SRAM 占用 1928 字节（RW＋ZI）；并成功创建了 Hex 文件（可执行文件，放在 Output 目录下）。

总结：如果编译提示有错误/警告，请根据提示，从第一个错误/警告开始解决，直到 0 错误 0 警告。如果出错，很有可能是之前的操作存在问题，请对照教程找问题。

另外，我们在 Readme 分组下还没有添加任何文件，由于只是添加一个说明性质的文件（.txt），并不是工程必备文件，因此这里我们就不添加了，开发板光盘的源码我们是有添加的，大家可以去参考一下。

至此，新建 HAL 库版本 MDK 工程完成。

8.2 下载验证

这里我们继续使用 DAP 仿真器下载，在 MDK 主界面，点击：  （下载按钮，也可以按键盘快捷键：F8），就可以将代码下载到开发板，如图 8.2.1 所示：

```
Build Output
Load "..\\..\\Output\\atk_f407.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 17:29:22
```

图 8.2.1 下载成功

上图提示：Application running…，则表示代码下载成功，且开始运行。此时，LED0 和 LED1 交叉闪烁。如果有朋友没能下载成功，请看第四章寻找问题，或者直接对照我们提供的实验 0-3，新建工程实验-HAL 库版本工程设置。

第九章 STM32 启动过程分析

本章给大家分析 STM32F4 的启动过程，这里的启动过程是指从 STM32 芯片上电复位执行的第一条指令开始，到执行用户编写的 main 函数这之间的过程。我们编写程序，基本都是用 C 语言编写，并且以 main 函数作为程序的入口。但是事实上，main 函数并非最先执行的，在此之前需要做一些准备工作，准备工作通过启动文件的程序来完成。理解 STM32 启动过程，对今后的学习和分析 STM32 程序有很大的帮助。

注意：学习本章内容之前，请大家最好先阅读由正点原子团队编写的《STM32 启动文件浅析》和《MAP 文件浅析》这两份文档（路径：A 盘→1，入门资料）。

本章将分为如下几个小节：

9.1 启动模式

9.2 启动文件分析

9.3 map 文件分析

9.1 启动模式

我们知道的复位方式有三种：上电复位，硬件复位和软件复位。当产生复位，并且离开复位状态后，CM4 内核做的第一件事就是读取下列两个 32 位整数的值：

（1）从地址 0x0000 0000 处取出堆栈指针 MSP 的初始值，该值就是栈顶地址。

（2）从地址 0x0000 0004 处取出程序计数器指针 PC 的初始值，该值指向复位后执行的第一条指令。下面用示意图表示，如图 9.1.1 所示。



图 9.1.1 复位序列

上述过程中，内核是从 0x0000 0000 和 0x0000 0004 两个的地址获取堆栈指针 SP 和程序计数器指针 PC。事实上，0x0000 0000 和 0x0000 0004 两个的地址可以被重映射到其他的地址空间。例如：我们将 0x0800 0000 映射到 0x0000 0000，即从内部 FLASH 启动，那么内核会从地址 0x0800 0000 处取出堆栈指针 MSP 的初始值，从地址 0x0800 0004 处取出程序计数器指针 PC 的初始值。CPU 会从 PC 寄存器指向的地址空间取出的第 1 条指令开始执行程序，就是开始执行复位中断服务程序 Reset_Handler。将 0x0000 0000 和 0x0000 0004 两个的地址重映射到其他地址空间，就是启动模式选择。

对于 STM32F4 的启动模式（也称自举模式），我们看表 9.1.1 进行分析。

启动模式选择引脚电平		启动模式	0x00000000 映射地址	0x00000004 映射地址
BOOT0	BOOT1			
0	x	内部 FLASH	0x08000000	0x08000004
1	1	内部 SRAM	0x20000000	0x20000004
1	0	系统存储器	0x1FFFF000	0x1FFFF004

表 9.1.1 启动模式选择表

注：启动引脚的电平：0：低电平；1：高电平；x:任意电平，即高低电平均可

由表 9.1.1 可以看到，STM32F4 根据 BOOT 引脚的电平选择启动模式，这两个 BOOT 引脚根据外部施加的电平来决定芯片的启动地址。（0 和 1 的准确电平范围可以查看 F4 系列数据手册 I/O 特性表，但我们最好是设置成 GND 和 VDD 的电平值）

（1）内部 FLASH 启动方式

当芯片上电后采样到 BOOT0 引脚为低电平时，0x00000000 和 0x00000004 地址被映射到内部 FLASH 的首地址 0x08000000 和 0x08000004。因此，内核离开复位状态后，读取内部 FLASH

的 0x08000000 地址空间存储的内容，赋值给栈指针 MSP，作为栈顶地址，再读取内部 FLASH 的 0x08000004 地址空间存储的内容，赋值给程序指针 PC，作为将要执行的第一条指令所在的地址。完成这两个操作后，内核就可以开始从 PC 指向的地址中读取指令执行了。

（2）内部 SRAM 启动方式

类似于内部 Flash，当芯片上电后采样到 BOOT0 和 BOOT1 引脚均为高电平时，地址 0x00000000 和 0x00000004 被映射到内部 SRAM 的首地址 0x20000000 和 0x20000004，内核从 SRAM 空间获取内容进行自举。在实际应用中，由启动文件 starttup_stm32f407xx.s 决定了 0x00000000 和 0x00000004 地址存储什么内容，链接时，由分散加载文件(sct)决定这些内容的绝对地址，即分配到内部 FLASH 还是内部 SRAM。

（3）系统存储器启动方式

当芯片上电后采样到 BOOT0=1，BOOT1=0 的组合时，内核将从系统存储器的 0x1FFFF000 及 0x1FFFF004 获取 MSP 及 PC 值进行自举。系统存储器是一段特殊的空间，用户不能访问，ST 公司在芯片出厂前就在系统存储器中固化了一段代码。因而使用系统存储器启动方式时，内核会执行该代码，该代码运行时，会为 ISP(In System Program)提供支持，在 STM32F4 上最常见的是检测 USART1 传输过来的信息，并根据这些信息更新自己内部 FLASH 的内容，达到升级产品应用程序的目的，因此这种启动方式也称为 ISP 启动方式。

9.2 启动文件分析

STM32 启动文件由 ST 官方提供，在官方的 STM32Cube 固件包里，对于 STM32F407 系列芯片的启动文件，我们选用的是 startup_stm32f407xx.s 这个文件。启动文件用汇编编写，是系统上电复位后第一个执行的程序。

启动文件主要做了以下工作：

1、初始化堆栈指针 SP = _initial_sp
2、初始化程序计数器指针 PC = Reset_Handler
3、设置堆和栈的大小
4、初始化中断向量表
5、配置外部 SRAM 作为数据存储器（可选）
6、配置系统时钟，通过调用 SystemInit 函数（可选）
7、调用 C 库中的 _main 函数初始化用户堆栈，最终调用 main 函数

9.2.1 启动文件中的一些指令

指令名称	作用
EQU	给数字常量取一个符号名，相当于 C 语言中的 define
AREA	汇编一个新的代码段或者数据段
ALIGN	编译器对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节对齐。要注意的是，这个不是 ARM 的指令，是编译器的，这里放到一起为了方便。
SPACE	分配内存空间
PRESERVE8	当前文件堆栈需要按照 8 字节对齐
THUMB	表示后面指令兼容 THUMB 指令。在 ARM 以前的指令集中有 16 位的 THUMBM 指令，现在 Cortex-M 系列使用的都是 THUMB-2 指令集，THUMB-2 是 32 位的，兼容 16 位和 32 位的指令，是 THUMB 的超级版。
EXPORT	声明一个标号具有全局属性，可被外部的文件使用
DCD	以字节为单位分配内存，要求 4 字节对齐，并要求初始化这些内存
PROC	定义子程序，与 ENDP 成对使用，表示子程序结束
WEAK	弱定义，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不会出错。要注意的是，这个不是 ARM 的指令，是编译器的，这里放到一起为了方便。

IMPORT	声明标号来自外部文件，跟 C 语言中的 extern 关键字类似
LDR	从存储器中加载字到一个存储器中
BLX	跳转到由寄存器给出的地址，并根据寄存器的 LSE 确定处理器的状态，还要把跳转前的下条指令地址保存到 LR
BX	跳转到由寄存器/标号给出的地址，不用返回
B	跳转到一个标号
IF,ELSE,ENDIF	汇编条件分支语句，跟 C 语言的类似
END	到达文件的末尾，文件结束

表 9.2.1.1 启动文件的汇编指令

上表，列举了 STM32 启动文件的一些汇编和编译器指令，关于其他更多的 ARM 汇编指令，我们可以通过 MDK 的索引搜索工具中搜索找到。打开索引搜索工具的方法：MDK->Help->uVision Help，如图 9.2.1.1 所示。

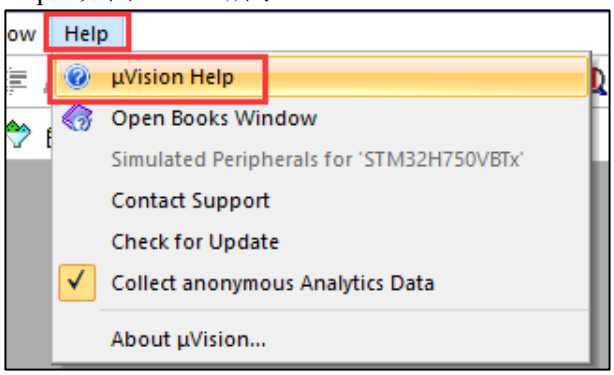


图 9.2.1.1 打开索引搜索工具的方法

打开之后，我们以 EQU 为例，演示一下怎么使用，如图 9.2.1.2 所示。

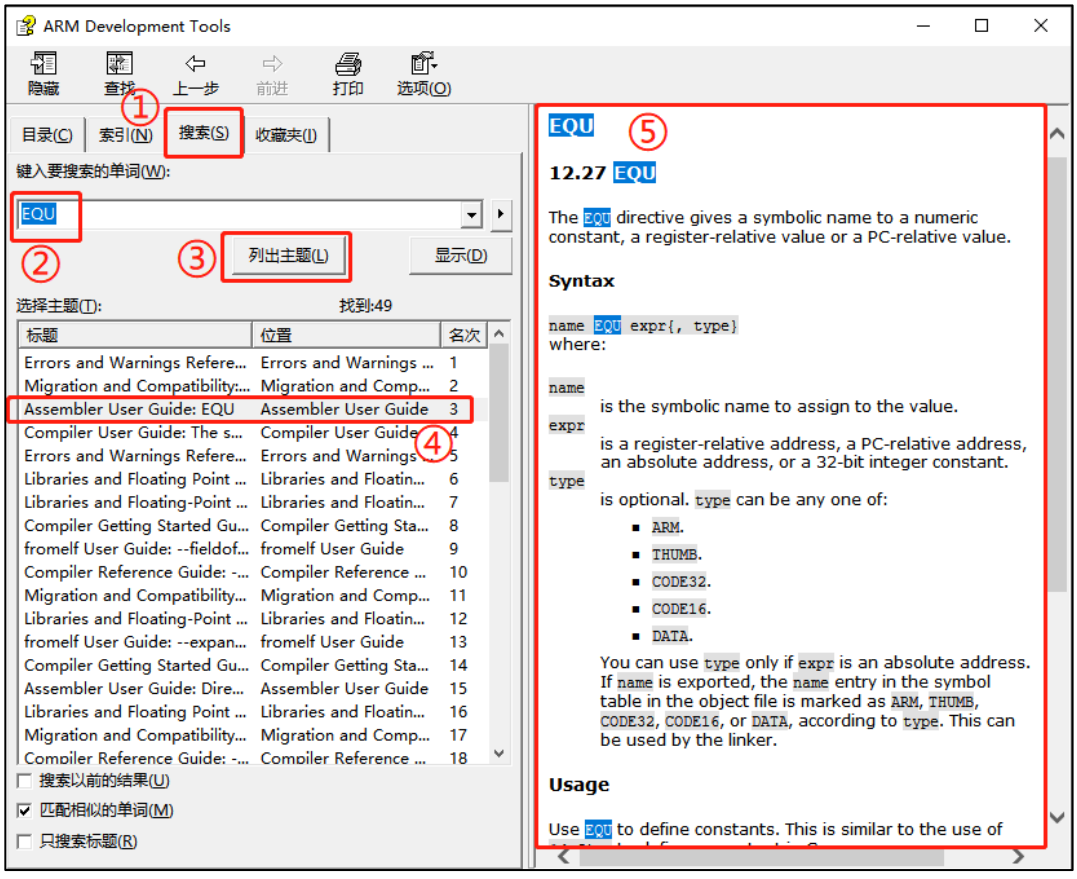


图 9.2.1.2 搜索 EQU 汇编指令

搜索到的结果有很多，我们只需要看位置为 Assembler User Guide 这部分即可。

9.2.2 启动文件代码讲解

（1）栈空间的开辟

栈空间的开辟，源码如图 9.2.2.1 所示：

```
33  Stack_Size      EQU     0x00000400
34
35                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
36  Stack_Mem       SPACE   Stack_Size
37  __initial_sp
```

图 9.2.2.1 栈空间的开辟

源码含义：开辟一段大小为 0x0000 0400（1KB）的栈空间，段名为 STACK，NOINIT 表示不初始化； READWRITE 表示可读可写；ALIGN=3，表示按照 2^3 对齐，即 8 字节对齐。

AREA 汇编一个新的代码段或者数据段。

SPACE 分配内存指令，分配大小为 Stack_Size 字节连续的存储单元给栈空间。

__initial_sp 紧挨着 SPACE 放置，表示栈的结束地址，栈是从高往低生长，所以结束地址就是栈顶地址。

栈主要用于存放局部变量，函数形参等，属于编译器自动分配和释放的内存，栈的大小不能超过内部 SRAM 的大小。如果工程的程序量比较大，定义的局部变量比较多，那么就需要在启动代码中修改栈的大小，即修改 Stack_Size 的值。如果程序出现了莫名其妙的错误，并进入了 HardFault 的时候，你就要考虑下是不是栈空间不够大，溢出了的问题。

（2）堆空间的开辟

堆空间的开辟，源码如图 9.2.2.2 所示：

```
44  Heap_Size       EQU     0x00000200
45
46                  AREA    HEAP, NOINIT, READWRITE, ALIGN=3
47  __heap_base
48  Heap_Mem        SPACE   Heap_Size
49  __heap_limit
```

图 9.2.2.2 堆空间的开辟

源码含义：开辟一段大小为 0x0000 0200（512 字节）的堆空间，段名为 HEAP，不初始化，可读可写，8 字节对齐。

__heap_base 表示堆的起始地址，__heap_limit 表示堆的结束地址。堆和栈的生长方向相反的，堆是由低向高生长，而栈是从高往低生长。

堆主要用于动态内存的分配，像 malloc()、calloc()和 realloc()等函数申请的内存就在堆上面。堆中的内存一般由程序员分配和释放，若程序员不释放，程序结束时可能由操作系统回收。

接下来是 PRESERVE8 和 THUMB 指令两行代码。如图 9.2.2.3 所示。

```
51                  PRESERVE8
52                  THUMB
```

图 9.2.2.3 PRESERVE8 和 THUMB 指令

PRESERVE8：指示编译器按照 8 字节对齐。

THUMB：指示编译器之后的指令为 THUMB 指令。

注意：由于正点原子提供了独立的内存管理实现方式（mymalloc，myfree 等），并不需要使用 C 库的 malloc 和 free 等函数，也就用不到堆空间，因此我们可以设置 Heap_Size 的大小为 0，以节省内存空间。

（3）中断向量表定义（简称：向量表）

为中断向量表定义一个数据段，如图 9.2.2.4 所示：

```
56                      AREA    RESET, DATA, READONLY
57                      EXPORT  __Vectors
58                      EXPORT  __Vectors_End
59                      EXPORT  __Vectors_Size
```

图 9.2.2.4 为中断向量表定义一个数据段

源码含义：定义一个数据段，名字为 RESET, READONLY 表示只读。EXPORT 表示声明一个标号具有全局属性，可被外部的文件使用。这里是声明了__Vectors、__Vectors_End 和__Vectors_Size 三个标号具有全局性，可被外部的文件使用。

STM32F407 的中断向量表定义代码，如图 9.2.2.5 所示。

```
61  __Vectors       DCD     __initial_sp                ; Top of Stack
62                  DCD     Reset_Handler               ; Reset Handler
63                  DCD     NMI_Handler                 ; NMI Handler
64                  DCD     HardFault_Handler           ; Hard Fault Handler
65                  DCD     MemManage_Handler           ; MPU Fault Handler
66                  DCD     BusFault_Handler            ; Bus Fault Handler
67                  DCD     UsageFault_Handler          ; Usage Fault Handler
68                  DCD     0                           ; Reserved
69                  DCD     0                           ; Reserved
70                  DCD     0                           ; Reserved
71                  DCD     0                           ; Reserved
72                  DCD     SVC_Handler                 ; SVCall Handler
73                  DCD     DebugMon_Handler            ; Debug Monitor Handler
74                  DCD     0                           ; Reserved
75                  DCD     PendSV_Handler              ; PendSV Handler
76                  DCD     SysTick_Handler             ; SysTick Handler
77
78                  ; External Interrupts（外部中断）
79                  DCD     WWDG_IRQHandler             ; Window WatchDog interrupt ( wwdg1_it)
80                  DCD     PVD_AVD_IRQHandler          ; PVD/AVD through EXTI Line detection
81                  ;中间篇幅太长，省略掉了，代码向量表与STM32H750的向量表对应
82                  DCD     WAKEUP_PIN_IRQHandler       ; Interrupt for all 6 wake-up pins
83
84
85  __Vectors_End
86
87  __Vectors_Size  EQU     __Vectors_End - __Vectors
```

图 9.2.2.5 中断向量表定义代码

__Vectors 为向量表起始地址，__Vectors_End 为向量表结束地址，__Vectors_Size 为向量表大小，__Vectors_Size = __Vectors_End - __Vectors。

DCD：分配一个或者多个以字为单位的内存，以四字节对齐，并要求初始化这些内存。

中断向量表被放置在代码段的最前面。例如：当我们的程序在 FLASH 运行时，那么向量表的起始地址是：0x0800 0000。结合图 9.2.2.5 可以知道，地址 0x0800 0000 存放的是栈顶地址。DCD：以四字节对齐分配内存，也就是下个地址是 0x0800 0004，存放的是 Reset_Handler 中断函数入口地址。

从代码上看，向量表中存放的都是中断服务函数的函数名，所以 C 语言中的函数名对芯片来说实际上就是一个地址。

STM32F4 的中断向量表可以在《STM32F4xx 参考手册_V4（中文版）.pdf》的第 10 章的 10.2 小节找到，与中断向量表定义代码是对应的。

（4）复位程序

接下来是定义只读代码段，如图 9.2.2.6 所示：

```
167  ···············AREA····|.text|,·CODE,·READONLY
```

图 9.2.2.6 定义只读代码段

定义一个段命为.text，只读的代码段，在 CODE 区。

复位子程序代码，如图 9.2.2.7 所示：

```
169  ; Reset handler
170  Reset_Handler    PROC
171                  EXPORT  Reset_Handler             [WEAK]
172          IMPORT  SystemInit
173          IMPORT  __main
174
175                  LDR     R0, =SystemInit
176                  BLX     R0
177                  LDR     R0, =__main
178                  BX      R0
179                  ENDP
```

图 9.2.2.7 复位子程序代码

利用 PROC、ENDP 这一对伪指令把程序段分为若干个过程，使程序的结构加清晰。

复位子程序是复位后第一个被执行的程序，主要是调用 SystemInit 函数配置系统时钟、还有就是初始化 FSMC 总线上外挂的 SRAM(可选)。然后在调用 C 库函数__main，最终调用 main 函数去到 C 的世界。

EXPORT 声明复位中断向量 Reset_Handler 为全局属性，这样外部文件就可以调用此复位中断服务。

WEAK：表示弱定义，如果外部文件优先定义了该标号则首先引用外部定义的标号，如果外部文件没有声明也不会出错。这里表示复位子程序可以由用户在其他文件重新实现，这里并不是唯一的。

IMPORT 表示该标号来自外部文件。这里表示 SystemInit 和__main 这两个函数均来自外部的文件。

LDR、BLX、BX 是内核指令，可在《Cortex-M3 权威指南》第四章-指令集里面查询到。

LDR 表示从存储器中加载字到一个存储器中。

BLX 表示跳转到由寄存器给出的地址，并根据寄存器的 LSE 确定处理器的状态，还要把跳转前的下条指令地址保存到 LR。

BX 表示跳转到由寄存器/标号给出的地址，不用返回。这里表示切换到__main 地址，最终调用 main 函数，不返回，进入 C 的世界。

（5）中断服务程序

```
183  ;系统异常中断部分
184  NMI_Handler     PROC
185                  EXPORT  NMI_Handler               [WEAK]
186                  B       .
187                  ENDP
188  HardFault_Handler\
189                  PROC
190                  EXPORT  HardFault_Handler         [WEAK]
191                  B       .
192                  ENDP
193  ;中间代码太长，已经省略
194  SysTick_Handler PROC
195                  EXPORT  SysTick_Handler           [WEAK]
196                  B       .
197                  ENDP
198  ;外部中断部分
199  Default_Handler PROC
200
201                  EXPORT  WWDG_IRQHandler           [WEAK]
202                  EXPORT  PVD_IRQHandler            [WEAK]
203  ;中间代码太长，已经省略
204  FPU_IRQHandler
205
206                  B       .
207
208                  ENDP
```

图 9.2.2.8 中断服务程序

接下来就是中断服务程序了，如图 9.2.2.8 所示。

可以看到这些中断服务函数都被[WEAK]声明为弱定义函数，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不会出错。

这些中断函数分为系统异常中断和外部中断，外部中断根据不同芯片有所变化。B 指令是

跳转到一个标号，这里跳转到一个‘.’，表示无限循环。

在启动文件代码中，已经把我们所有中断的中断服务函数写好了，但都是声明为弱定义，所以真正的中断服务函数需要我们在外部实现。

如果我们开启了某个中断，但是忘记写对应的中断服务程序函数又或者把中断服务函数名写错，那么中断发生时，程序就会跳转到启动文件预先写好的弱定义的中断服务程序中，并且在 B 指令作用下跳转到一个‘.’中，无限循环。

这里的系统异常中断部分是内核的，外部中断部分是外设的。

（6）用户堆栈初始化

ALIGN 指令，如图 9.2.2.9 所示：

```
365  ·····ALIGN
```

图 9.2.2.9 ALIGN 指令

ALIGN 表示对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节对齐。要注意的是，这个不是 ARM 的指令，是编译器的。

接下就是启动文件最后一部分代码，用户堆栈初始化代码，如图 9.2.2.10 所示：

```
370  ·····IF·····:DEF:__MICROLIB
371
372  ·····EXPORT··__initial_sp
373  ·····EXPORT··__heap_base
374  ·····EXPORT··__heap_limit
375
376  ·····ELSE
377
378  ·····IMPORT··__use_two_region_memory
379  ·····EXPORT··__user_initial_stackheap
380
381  __user_initial_stackheap
382
383  ·····LDR·····R0,·=··Heap_Mem
384  ·····LDR·····R1,·=(Stack_Mem·+·Stack_Size)
385  ·····LDR·····R2,·=·(Heap_Mem·+··Heap_Size)
386  ·····LDR·····R3,·=·Stack_Mem
387  ·····BX·····LR
388
389  ·····ALIGN
390
391  ·····ENDIF
392
393  ·····END
```

图 9.2.2.10 用户堆栈初始化代码

IF, ELSE, ENDIF 是汇编的条件分支语句。

370 行判断是否定义了__MICROLIB。关于__MICROLIB 这个宏定义，我们是在 KEIL 里面配置，具体方法如图 9.2.2.11 所示。

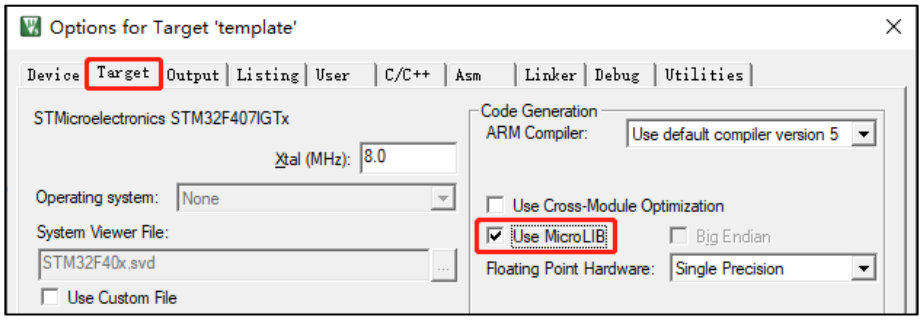


图 9.2.2.11 __MICROLIB 定义方法

勾选了 Use MicroLIB 就代表定义了__MICROLIB 这个宏。

如果定义__MICROLIB，声明__initial_sp、__heap_base 和__heap_limit 这三个标号具有全局属性，可被外部的文件使用。__initial_sp 表示栈顶地址，__heap_base 表示堆起始地址，__heap_limit 表示堆结束地址。

如果没有定义__MICROLIB，实际的情况就是我们没有定义__MICROLIB，所以使用默认的 C 库运行。那么堆栈的初始化由 C 库函数__main 来完成。

IMPORT 声明__use_two_region_memory 标号来自外部文件。

EXPORT 声明__user_initial_stackheap 具有全局属性，可被外部的文件使用。

383 行保存堆起始地址。

384 行保存栈大小。

385 行保存堆大小。

386 行保存栈顶指针。

387 行跳转到 LR 标号给出的地址，不用返回。

393 行 END 表示到达文件的末尾，文件结束。

Use MicroLIB

MicroLIB 是 MDK 自带的微库，是缺省 C 库的备选库，MicroLIB 进行了高度优化使得其代码变得很小，功能比缺省 C 库少。MicroLIB 是没有源码的，只有库。

关于 MicroLIB 更多知识可以看官方介绍 http://www.keil.com/arm/microlib.asp 。

9.2.3 系统启动流程

我们知道启动模式不同，启动的起始地址是不一样的，下面我们以代码下载到内部 FLASH 的情况举例，即代码从地址 0x0800 0000 开始被执行。

当产生复位，并且离开复位状态后，CM4 内核做的第一件事就是读取下列两个 32 位整数的值：

（1）从地址 0x0800 0000 处取出堆栈指针 MSP 的初始值，该值就是栈顶地址。

（2）从地址 0x0800 0004 处取出程序计数器指针 PC 的初始值，该值指向中断服务程序 Reset_Handler。下面用示意图表示，如图 9.2.3.1 所示。



图 9.2.3.1 复位序列

我们看看 DMF407HAL 库例程的实验 1 跑马灯实验中，取出的 MPS 和 PC 的值是多少，方法如图 9.2.3.2 所示。

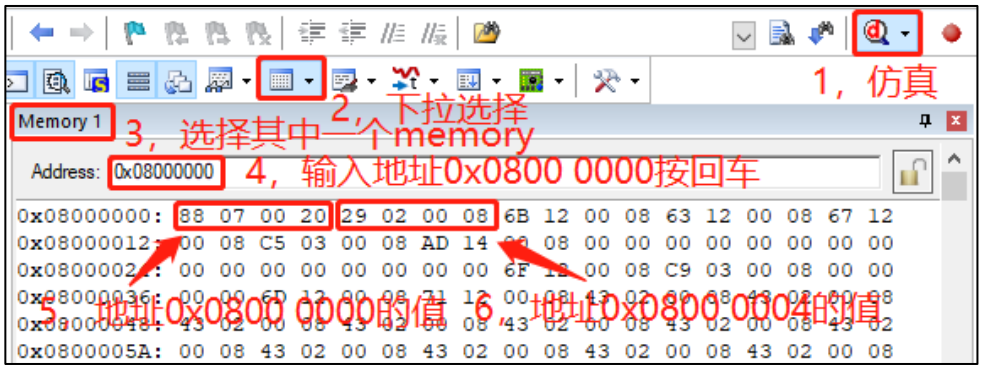


图 9.2.3.2 取出的 MPS 和 PC 的值

由图 9.2.3.2 可以知道地址 0x0800 0000 的值是 0x2000 00788，地址 0x0800 0004 的值是 0x0800 0229，即堆栈指针 SP =0x2000 00788，程序计数器指针 PC = 0x0800 0229（即复位中断服务程序 Reset_Handler 的入口地址）。因为 CM4 内核是小端模式，所以倒着读。

请注意，这与传统的 ARM 架构不同——其实也和绝大多数的其它单片机不同。传统的 ARM 架构总是从 0 地址开始执行第一条指令。它们的 0 地址处总是一条跳转指令。而在 CM4 内核

中，0 地址处提供 MSP 的初始值，然后就是向量表（向量表在以后还可以被移至其它位置）。向量表中的数值是 32 位的地址，而不是跳转指令。向量表的第一个条目指向复位后应执行的第一条指令，就是 Reset_Handler 这个函数。下面继续以正点原子探索者 STM32F407 开发板 HAL 库例程的实验 1 跑马灯实验为例，代码从地址 0x0800 0000 开始被执行，讲解一下系统启动，初始化堆栈、MSP 和 PC 后的内存情况。

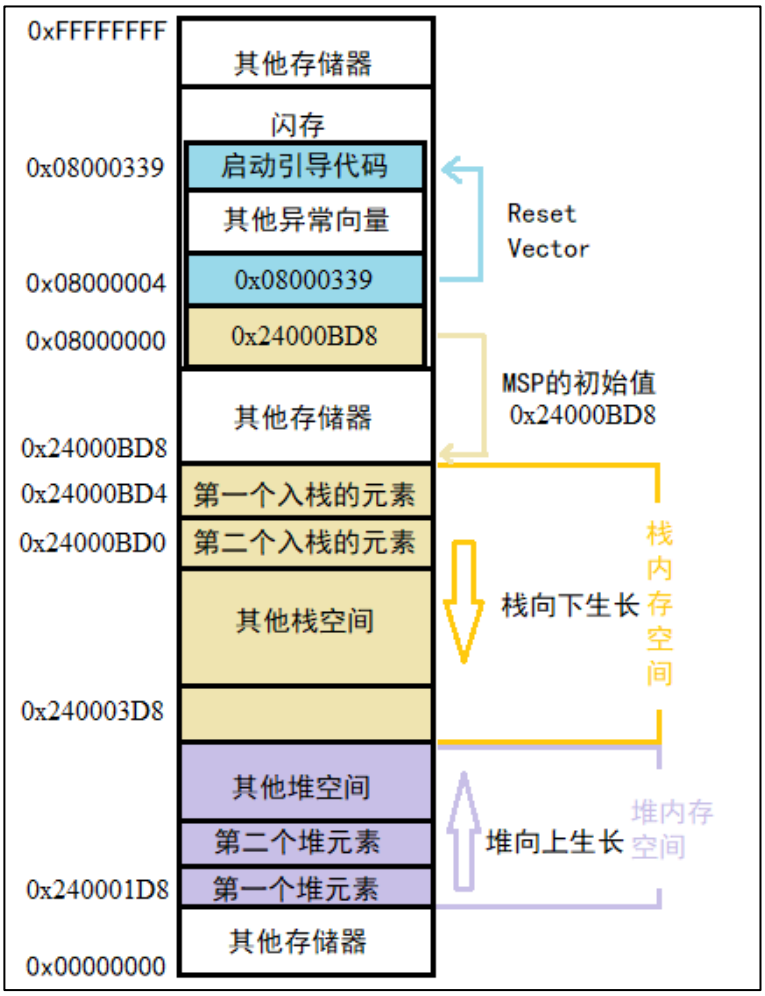


图 9.2.3.3 初始化堆栈、MSP 和 PC 后的内存情况

因为 CM4 使用的是向下生长的满栈，所以 MSP 的初始值必须是堆栈内存的末地址加 1。举例来说，如果你的栈区域在 0x2400 03D8 - 0x2400 0BD4 （2KB 大小）之间，那么 MSP 的初始值就必须是 0x2400 0BD8。

向量表跟随在 MSP 的初始值之后——也就是第 2 个表目。

R15 是程序计数器，在汇编代码中，可以使用名字“PC”来访问它。ARM 规定：PC 最低两位并不表示真实地址，最低位 LSB 用于表示是 ARM 指令（0）还是 Thumb 指令（1），因为 CM4 主要执行 Thumb 指令，所以这些指令的最低位都是 1（都是奇数）。因为 CM4 内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4。比如说：

```
0x1000:  MOV R0,  PC ;  R0 = 0x1004
```

如果向 PC 写数据，就会引起一次程序的分支（但是不更新 LR 寄存器）。CM4 中的指令至少是半字对齐的，所以 PC 的 LSB 总是读回 0。然而，在分支时，无论是直接写 PC 的值还是使用分支指令，都必须保证加载到 PC 的数值是奇数（即 LSB=1），表明是在 Thumb 状态下执行。倘若写了 0，则视为转入 ARM 模式，CM4 将产生一个 fault 异常。

正因为上述原因，图 9.2.3.3 中使用 0x0800 0339 来表达地址 0x0800 0338。当 0x0800 0339 处的指令得到执行后，就正式开始了程序的执行（即去到 C 的世界）。所以在此之前初始化 MSP 是必需的，因为可能第 1 条指令还没执行就会被 NMI 或是其它 fault 打断。MSP 初始化好后就已经为它们的服务例程准备好了堆栈。

STM32 启动文件分析就给大家介绍到这里，更多内容请看**《STM32 启动文件浅析_V1.1》**。

9.3 map 文件分析

9.3.1 MDK 编译生成文件简介

MDK 编译工程，会生成一些中间文件（如.o、.axf、.map 等），最终生成 hex 文件，以便下载到 MCU 上面执行，以 DMF407HAL 库例程的实验 1 跑马灯实验为例（其他开发板类似），编译过程产生的所有文件，都存放在 OBJ 文件夹下，如图 9.3.1.1 所示：



图 9.3.1.1 MDK 编译过程生成的文件

这里总共生成了 66 个文件，共 11 个类型，分别是：.axf、.crf、.d、.dep、.hex、.lnp、.lst、.o、.htm、bulild_log.htm 和.map。66 个文件看着不是很多，但是随着工程的增大，这些文件也会越来越多，大项目编译一次，可以生成几百甚至上千个这种文件，不过文件类型基本就是上面这些。

对于 MDK 工程来说，基本上任何工程在编译过程中都会有这 11 类文件，常见的 MDK 编译过程生产文件类型如表 9.3.1.1 所示：

文件类型	说明
.o	可重定向 1 对象文件，每个源文件（.c/.s 等）编译都会生成一个.o 文件
.axf	由 ARMCC 编译生产的可执行对象文件，不可重定向 2 （绝对地址） 多个.o 文件链接生成.axf 文件，我们在仿真的时候，需要用到该文件
.hex	Intel Hex 格式文件，可用于下载到 MCU，.hex 文件由.axf 文件转换而来
.crf	交叉引用文件，包含浏览信息（定义、标识符、引用）
.d	由 ARMCC/GCC 编译生产的依赖文件（.o 文件所对应的依赖文件） 每个.o 文件，都有一个对应的.d 文件
.dep	整个工程的依赖文件
.lnp	MDK 生成的链接输入文件，用于命令输入
.lst	C 语言或汇编编译器生成的列表文件

.htm	链接生成的列表文件
.build_log.htm	最近一次编译工程时的日志记录文件
.map	连接器生成的列表文件/MAP 文件， **该文件对我们非常有用**

表 9.3.1.1 常见的中间文件类型说明

注 1，可重定向是指该文件包含数据/代码，但是并没有指定地址，它的地址可由后续链接的时候进行指定。

注 2，不可重定向是指该文件所包含的数据/代码都已经指定地址了，不能再改变。

9.3.2 map 文件分析

.map 文件是编译器链接时生成的一个文件，它主要包含了交叉链接信息。通过.map 文件，我们可以知道整个工程的函数调用关系、FLASH 和 RAM 占用情况及其详细汇总信息，能具体到单个源文件（.c/.s）的占用情况，根据这些信息，我们可以对代码进行优化。.map 文件可以分为以下 5 个组成部分：

1， 程序段交叉引用关系（Section Cross References）
2， 删除映像未使用的程序段（Removing Unused input sections from the image）
3， 映像符号表（Image Symbol Table）
4， 映像内存分布图（Memory Map of the image）
5， 映像组件大小（Image component sizes）

9.3.2.1 map 文件的 MDK 设置

要生成 map 文件，我们需要在 MDK 的魔术棒→Listing 选项卡里面，进行相关设置，如图 9.3.2.1.1 所示：

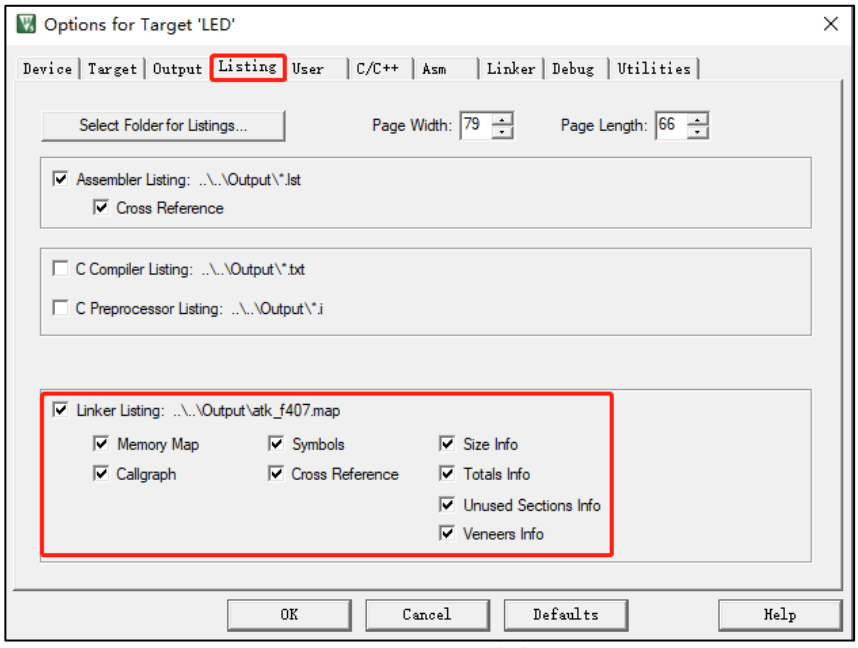


图 9.3.2.1.1 .map 文件生成设置

图 9.3.2.1.1 中红框框出的部分就是我们需要设置的，默认情况下，MDK 这部分设置就是全勾选的，如果我们想取消掉一些信息的输出，则取消相关勾选即可（一般不建议）。

如图 9.3.2.1.1 设置好 MDK 以后，全编译当前工程，当编译完成后（无错误），就会生成.map 文件。在 MDK 里面打开.map 文件的方法如图 9.3.2.1.2 所示：

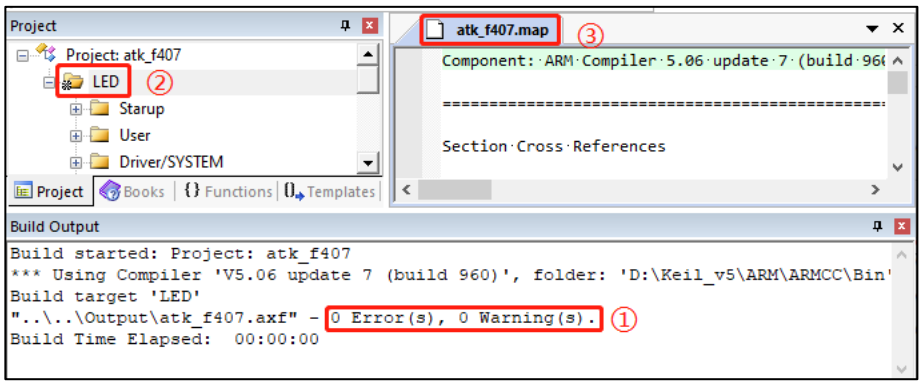


图 9.3.2.1.2 打开.map 文件

①，先确保工程编译成功（无错误）。
②，双击 LED，打开.map 文件。
③，map 文件打开成功。

9.3.2.2 map 文件的基础概念

为了更好的分析 map 文件，我们先对需要用到的一些基础概念进行一个简单介绍，相关概念如下：

- Section：描述映像文件的代码或数据块，我们简称程序段
- RO：Read Only 的缩写，包括只读数据（RO data）和代码（RO code）两部分内容，占用 FLASH 空间
- RW：Read Write 的缩写，包含可读写数据（RW data，有初值，且不为 0），占用 FLASH（存储初值）和 RAM（读写操作）
- ZI：Zero initialized 的缩写，包含初始化为 0 的数据（ZI data），占用 RAM 空间
- .text：相当于 RO code
- .constdata：相当于 RO data
- .bss：相当于 ZI data
- .data：相当于 RW data

9.3.2.3 map 文件的组成部分说明

我们前面说 map 文件分为 5 个部分组成，下面以 DMF407HAL 库例程的实验 1 跑马灯实验为例，简要讲解一下。

1. 程序段交叉引用关系（S S ection Cross References s ）

这部分内容描述了各个文件（.c/.s 等）之间函数（程序段）的调用关系，举个例子如图 9.3.2.3.1 所示：

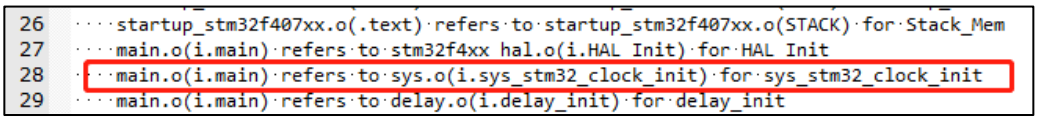


图 9.3.2.3.1 程序段交叉引用关系图

上图中，框出部分：main.o(i.main) refers to sys.o(i.sys_stm32_clock_init) for sys_stm32_clock_init 表示：main.c 文件中的 main 函数，调用了 sys.c 中的 sys_stm32_clock_init 函数。其中：i.main 表示 main 函数的入口地址，同理 i. sys_stm32_clock_init 表示 sys_stm32_clock_init 函数的入口地址。

2. 删除映像未使用的程序段（Removing Unused input sections from the image）

这部分内容描述了工程中由于未被调用而被删除的冗余程序段（函数/数据），如图 9.3.2.3.2 所示：

```
369  ================================================================================
370
371  Removing Unused input sections from the image.
372
373  //省略部分内容
374      Removing usart.o(i._ttywrch), (4 bytes).
375      Removing usart.o(i.fputc), (28 bytes).
376      Removing usart.o(i.usart_init), (56 bytes).
377  //省略部分内容
378
379  260 unused section(s) (total 20846 bytes) removed from the image.
380
381  ================================================================================
```

图 9.3.2.3.2 删除未用到的程序段

上图中，列出了所有被移除的程序段，比如 usart.c 里面的 usart_init 函数就被移除了，因为该例程没用到 usart_init 函数。

另外，在最后还有一个统计信息：260 unused section(s) (total 20846 bytes) removed from the image. 表示总共移除了 260 个程序段（函数/数据），大小为 20846 字节。即给我们的 MCU 节省了 20846 字节的程序空间。

为了更好的节省空间，我们一般在 MDK→魔术棒→C/C++选项卡里面勾选：One ELFSection per Function，如图 9.3.2.3.3 所示：

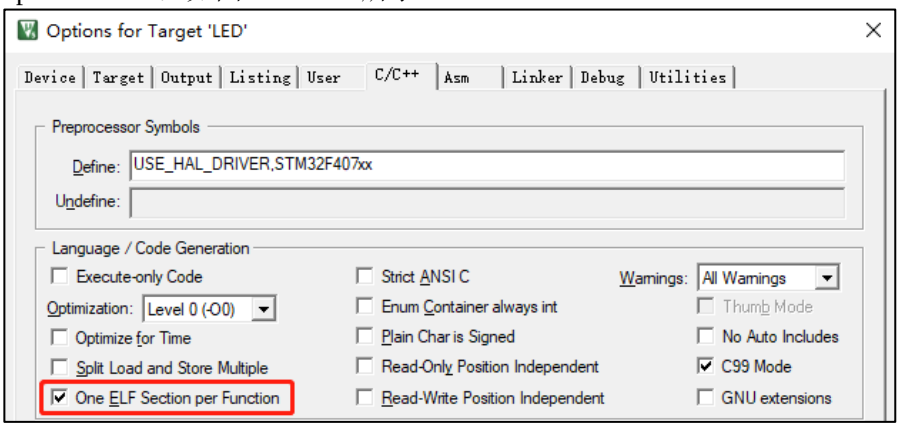


图 9.3.2.3.3 MDK 勾选 One ELF Section per Function

3. 映像符号表（Image Symbol Table）

映像符号表（Image Symbol Table）描述了被引用的各个符号（程序段/数据）在存储器中的存储地址、类型、大小等信息。映像符号表分为两类：本地符号（Local Symbols）和全局符号（Global Symbols）。

本地符号（Local Symbols）记录了用 static 声明的全局变量地址和大小，c 文件中函数的地址和用 static 声明的函数代码大小，汇编文件中的标号地址（作用域：限本文件）。

全局符号（Global Symbols）记录了全局变量的地址和大小，C 文件中函数的地址及其代码大小，汇编文件中的标号地址（作用域：全工程）。

4. 映像内存分布图（Memory Map of the image）

映像文件分为加载域（Load Region）和运行域（Execution Region），一个加载域必须有至少一个运行域（可以有多个运行域），而一个程序又可以有多个加载域。加载域为映像程序的实际存储区域，而运行域则是 MCU 上电后的运行状态。加载域和运行域的简化关系（这里仅表示一个加载域的情况）图，如图 9.3.2.3.4 所示：

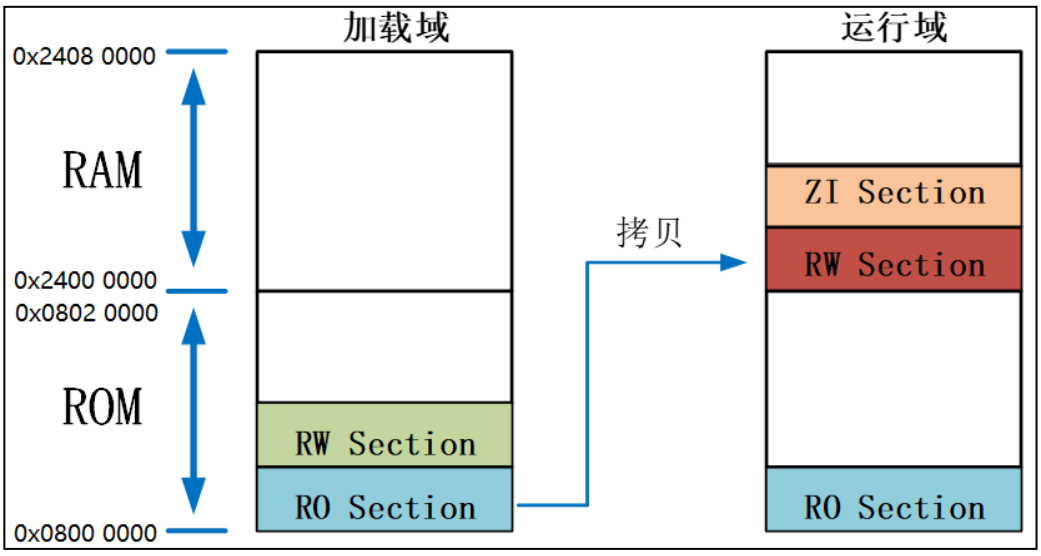


图 9.3.2.3.4 加载域运行域关系

由图可知，RW 区也是存放在 ROM（FLASH）里面的，在执行 main 函数之前，RW（有初值且不为 0 的变量）数据会被拷贝到 RAM 区，同时还会在 RAM 里面创建 ZI 区（初始化为 0 的变量）。

5. 映像组件大小（Image component sizes）

映像组件大小（Image component sizes）给出了整个映像所有代码（.o）占用空间的汇总信息。

由于篇幅较长，更多内容请大家查阅 **《MAP 文件浅析》**文档的内容。

第十章 STM32CubeMX 简介

STM32CubeMX 是由 ST 公司开发的图形化代码自动生成工具，能够快速生成初始化代码，如配置 GPIO，时钟树，中间件等，使用户专注于业务代码的开发。现在 ST 主推 HAL 库代码，经典的标准外设库已经停止维护了，新产品也只提供 HAL 库的代码，因此，我们学习 HAL 库是更加有优势的，由于 HAL 库具有低耦合、通用、抽象了硬件层，使得开发者无需太过关注硬件驱动的实现，使得开发更加的简单快速，更容易维护，因此被越来越多的产品所使用。

本章将分为如下几个小节：

10.1 STM32CubeMX 的作用；
10.2 安装 STM32CubeMX
10.3 使用 STM32CubeMX 新建工程
10.4 STM32CubeMX 新建工程使用建议

10.1 STM32CubeMX 的作用

STM32CubeMX 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；
⑨ 可作为 ST 的固件包、芯片手册等的下载引擎；

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。

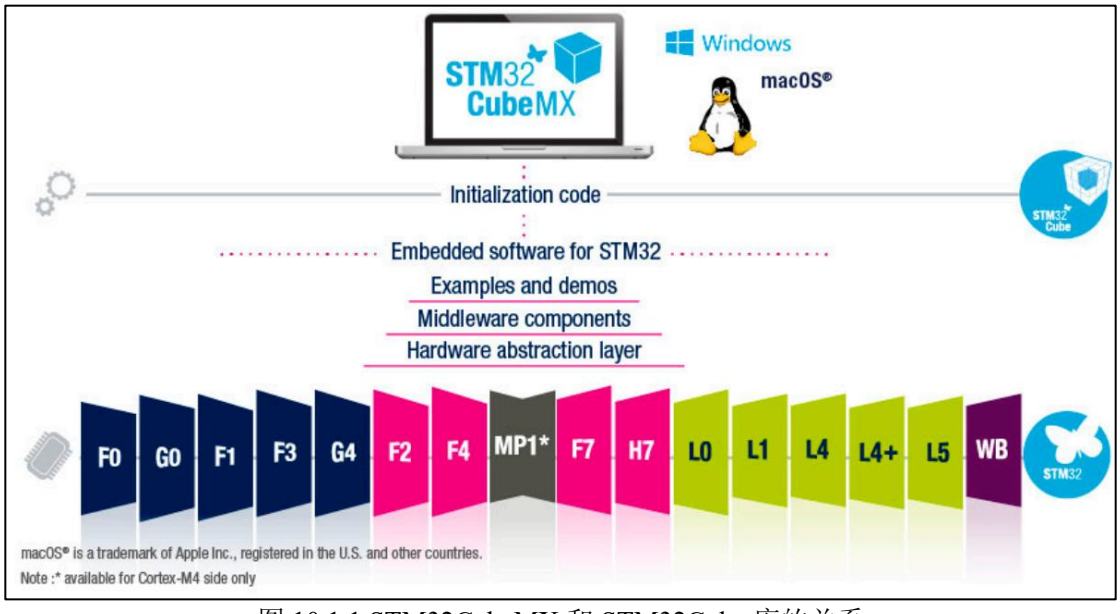


图 10.1.1 STM32CubeMX 和 STM32Cube 库的关系

当然，自动生成的驱动代码我们不去仔细专研其原理的话，对学习的提升很有限，而且在出现 BUG 的时候难以快速定位解决，因此我们也要了解其背后的原理。

10.2 安装 STM32CubeMX

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。

10.2.1 安装 JAVA 环境

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：A 盘→6，软件资料→1，软件→4，JAVA，有对应的 64 位和 32 位的离线版本。

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 10.2.1.1 所示。



图 10.2.1.1 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 10.2.1.2：



图 10.2.1.2 查看 Java 版本

10.2.2 安装 STM32CubeMX

在安装了 Java 运行环境之后，接下来我们安装 STM32CubeMX 图形化工具。该软件可以直接从光盘复制，目录为：A 盘→6，软件资料→1，软件→2，CubeMX，我们提供的是 windows 下的 V6.2 版本，如需最新版本可以直接从 ST 官方下载，下载地址为：www.st.com/stm32cube。

接下来我们直接双击 SetupSTM32CubeMX-6.2.0-Win.exe，安装步骤如下。



图 10.2.2.1 启动安装



图 10.2.2.2 接受本许可协议

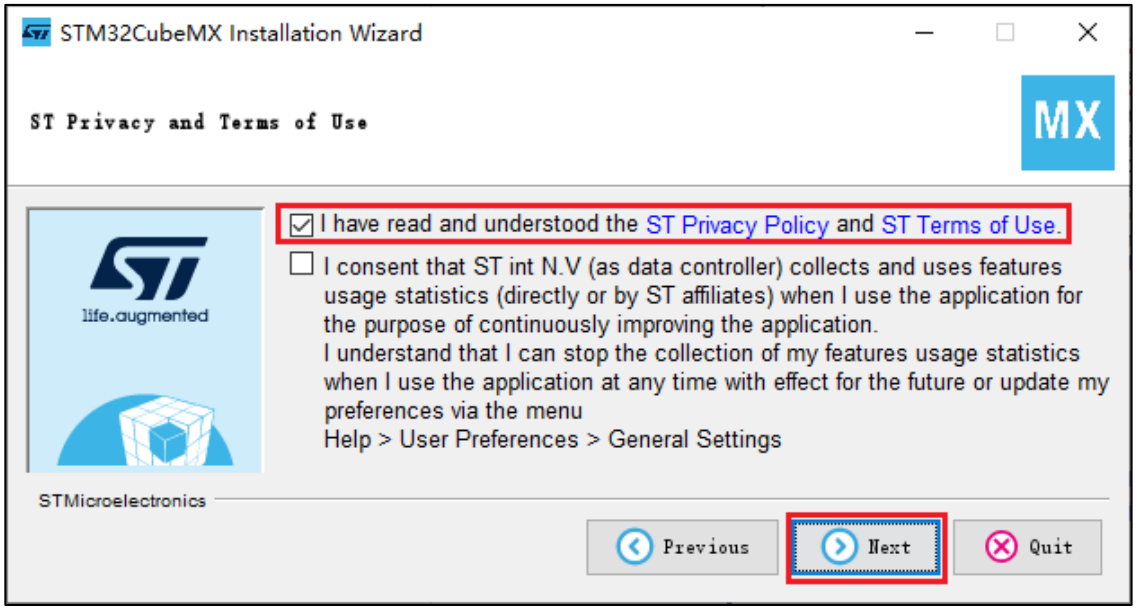


图 10.2.2.3 勾选第一项即可

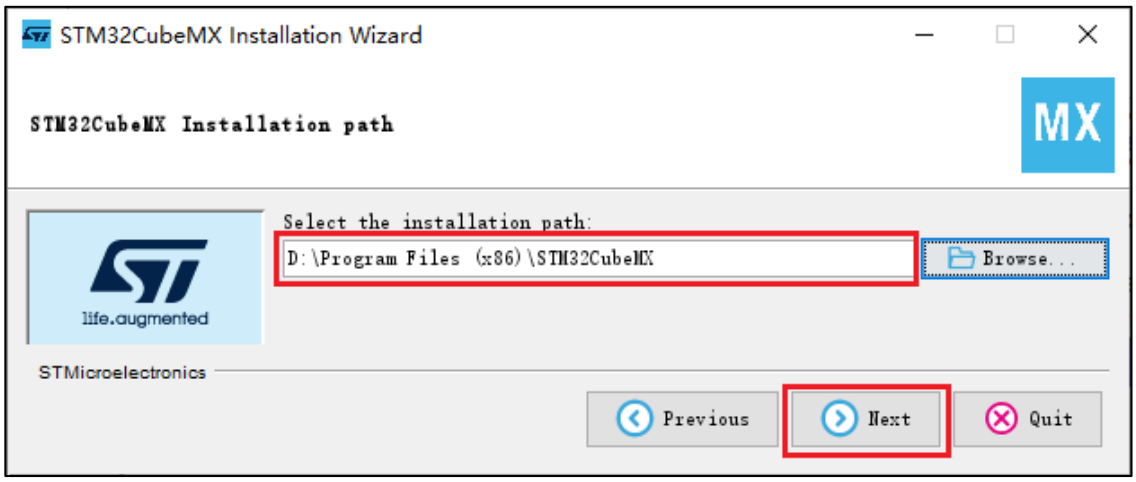


图 10.2.2.4 指定安装路径

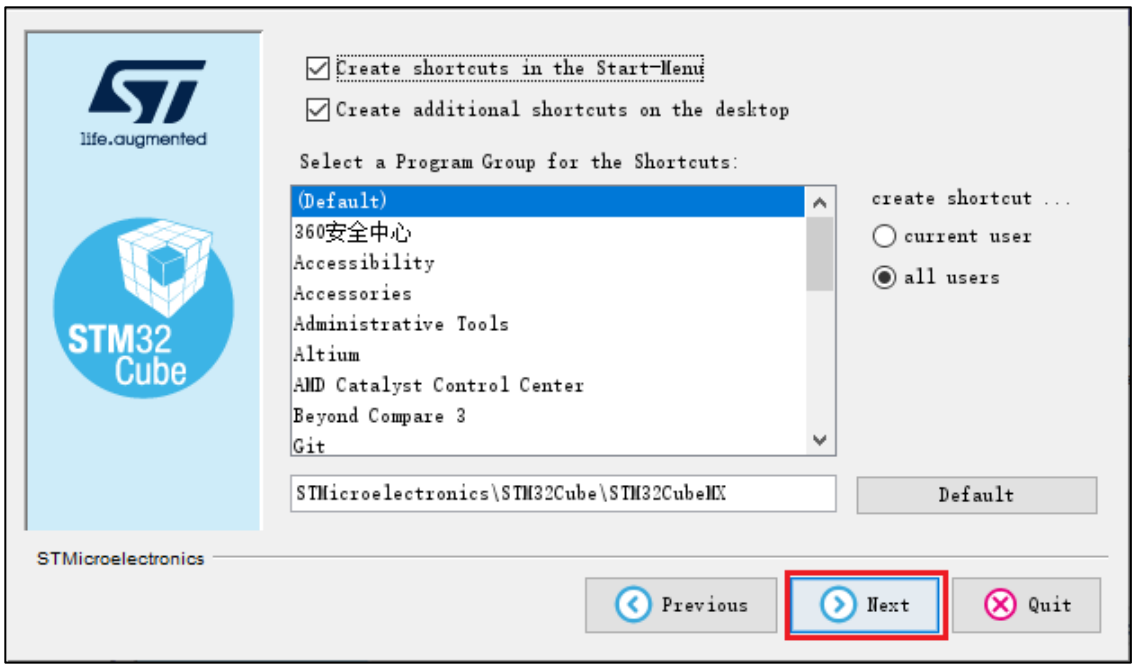


图 10.2.2.5 创建快捷方式

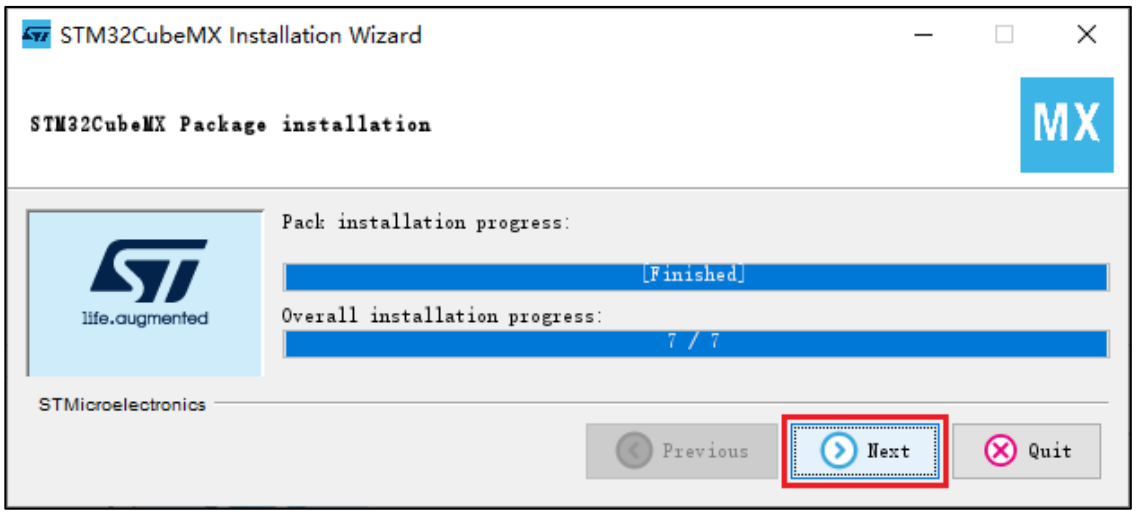


图 10.2.2.5 安装进度提示

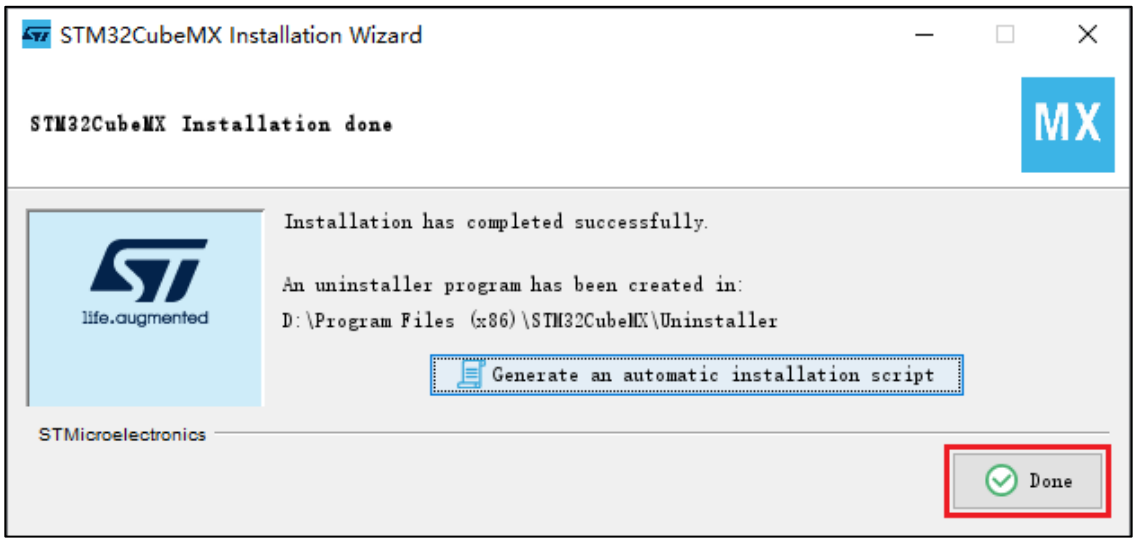


图 10.2.2.6 完成安装

10.3 使用 STM32CubeMX 新建工程

CubeMX 建立的工程结构和本书介绍的代码风格有所差异，限于篇幅，本部分只介绍如何使用 CubeMX 生成 MDK 工程的方法。

10.3.1 打开 STM32CubeMX

双击如图 10.3.1.1 所示的 STM32CubeMX 桌面快捷方式图标，打开后 CubeMX 主界面如图 10.3.1.2 所示。



图 10.3.1.1 CubeMX 快捷方式

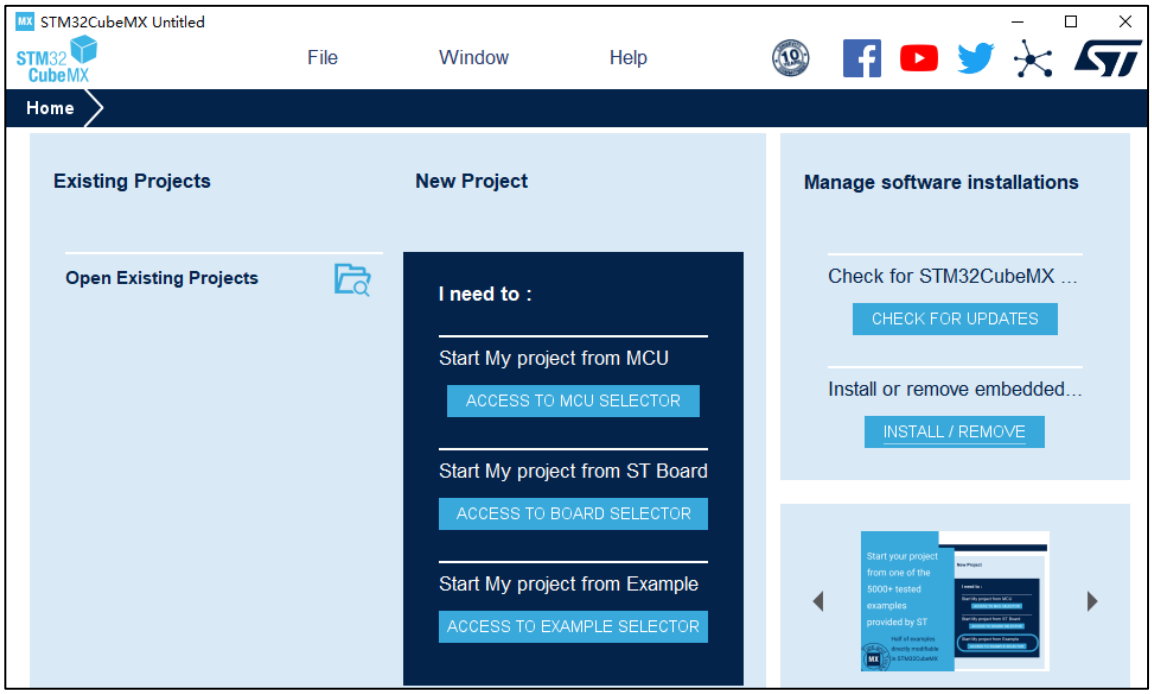


图 10.3.1.2 CubeMX 主界面

10.3.2 下载和关联的 STM32Cube 固件包

我们知道 STM32CubeMX 图形工具只有一种， STM32Cube 固件包却有多种，需要选择我们工程对应的固件包。

为了方便，新建工程前，我们先来下载和关联 STM32Cube 固件包，点击 Help->Manage embedded software packages，如图 10.3.2.1 所示。

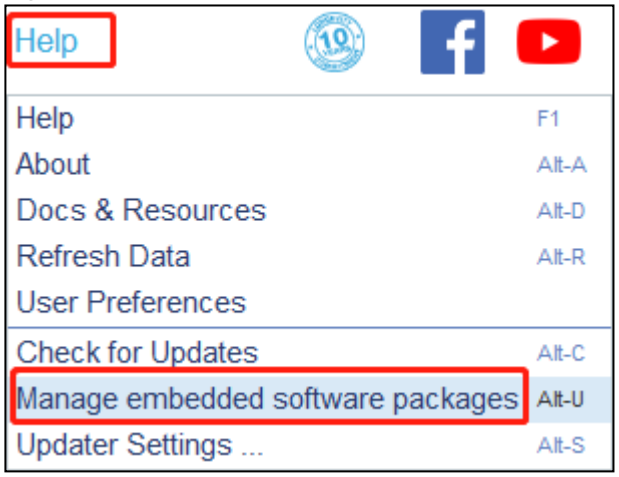


图 10.3.2.1 管理固件包

然后弹出管理界面，在该窗口找到 STM32F4 列表选项，勾选 1.26.0 版本。这里选择 1.26.0 版本是因为我们的光盘的固件包是这个版本的。关联 STM32Cube 固件包有两个方法，如图 10.3.2.2 所示.

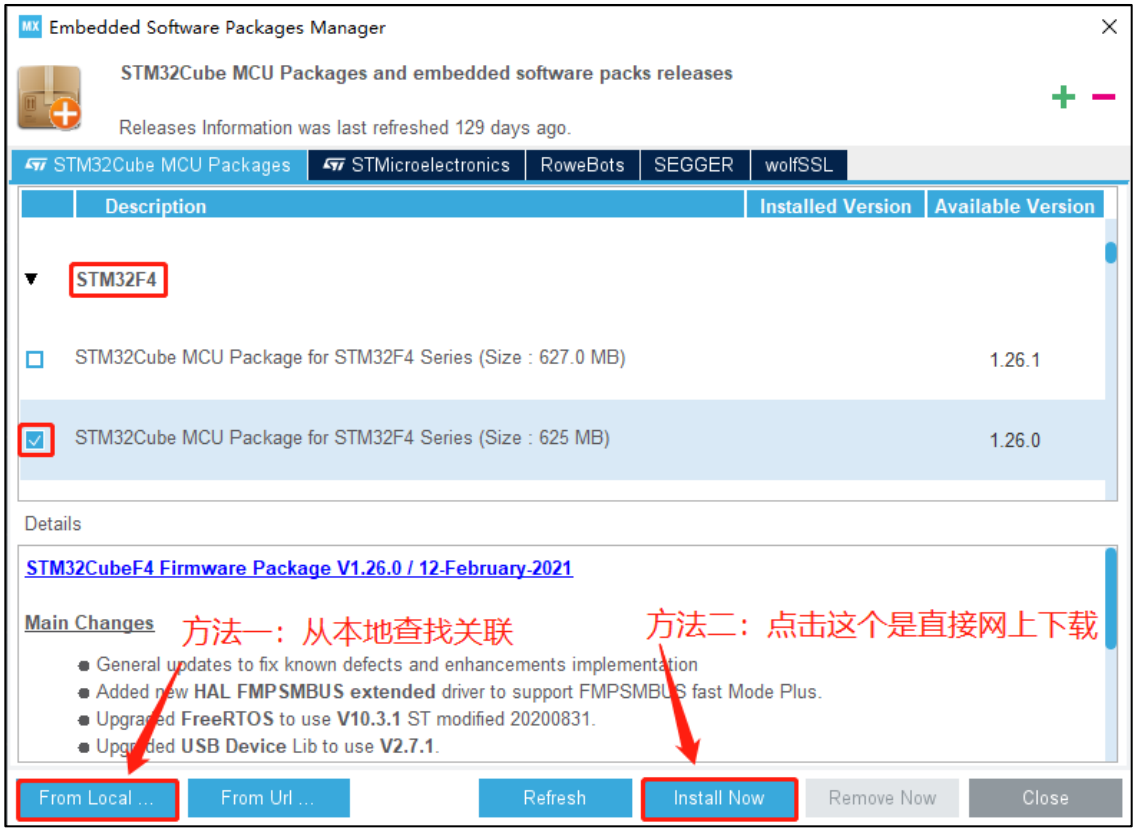


图 10.3.2.2 下载和关联 STM32Cube 固件包

方法二：下载好之后，会自动关联，所以不需要多讲什么。

方法一：点击后，弹出下面的窗口，然后选择光盘中的对应的固件包，注意这里是压缩包的形式，如图 10.3.2.3 所示。

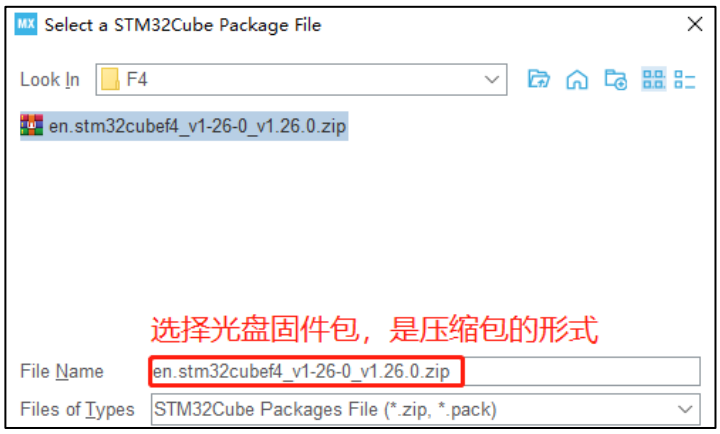


图 10.3.2.3 关联本地 STM32Cube 固件包

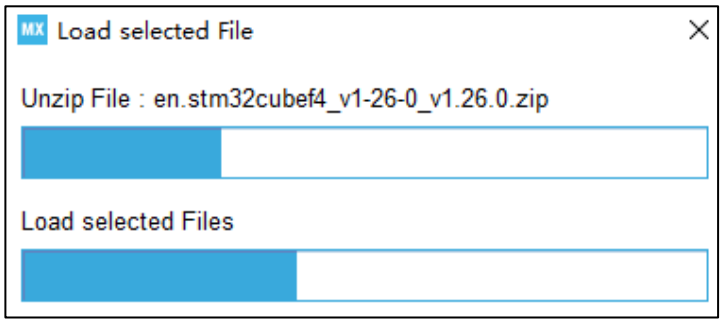


图 10.3.2.4 等待关联



图 10.3.2.5 关联成功

关联好固件包我们就可以开始新建工程了。

10.3.3 新建工程

使用 STM32CubeMX 配置工程的一般步骤为：

1. 工程初步建立
2. HSE 和 LSE 时钟源设置
3. 时钟系统（时钟树）配置
4. GPIO 功能引脚配置
5. Cortex-M4 内核基本配置（限定项）
6. 生成工程源码
7. 用户程序

接下来我们将按照上面 7 个步骤，依次教大家使用 STM32CubeMX 工具生成一个完整的工程。

1 工程初步建立

方法一：依次点击“File”，“New Project”即可建新工程。如果之前打开过的话，左侧最近打开的过程一列会有打开的工程列表，直接点击这些工程也可以打开。

方法二：直接点击 ACCESS TO MCU SELECTOR。

具体操作如图 10.3.3.1 所示。

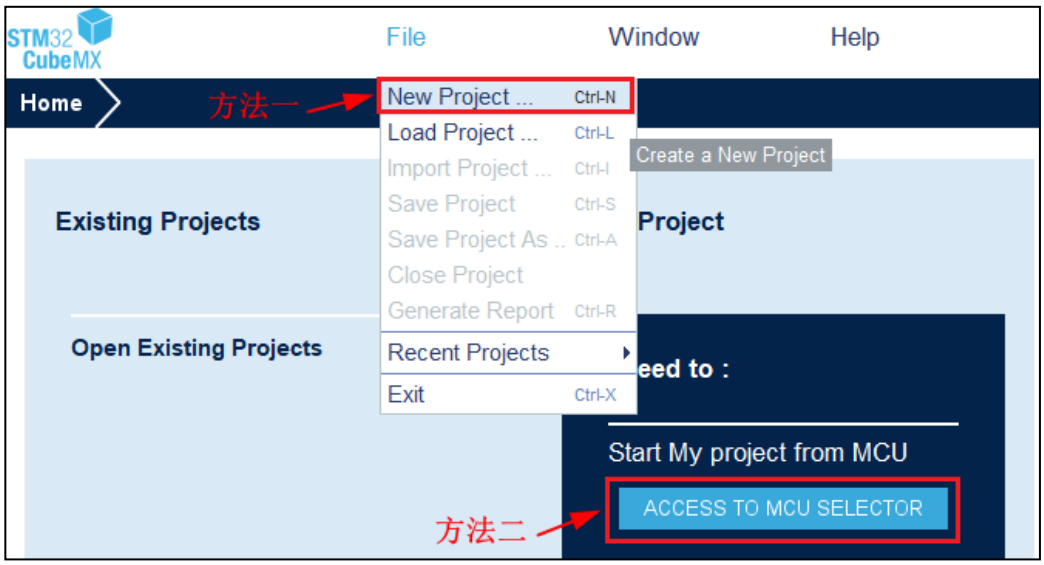


图 10.3.3.1 新建工程

点击新建工程后，第一次可能会联网下载一些的文件，可能等待时间比较长，可以直接选择取消即可。

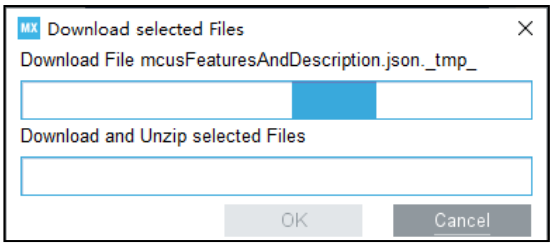


图 10.3.3.2 启动时联网更新检测

之后都可以进入芯片选型界面，如图 10.3.3.3 所示。

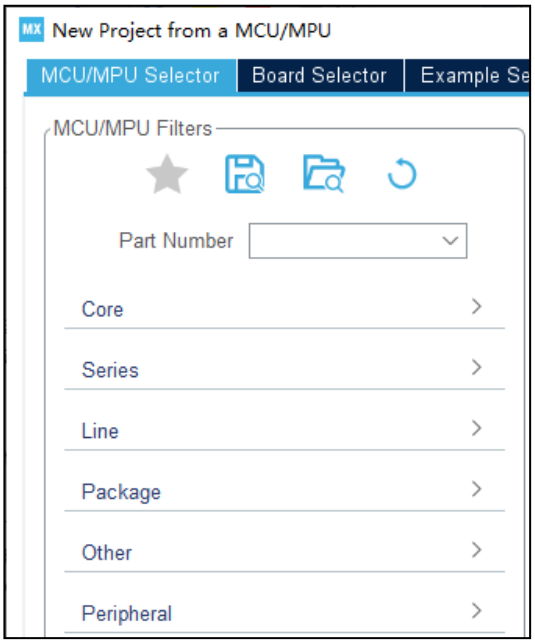


图 10.3.3.3 芯片选型界面

选择具体的芯片型号，如图 10.3.3.4 所示。

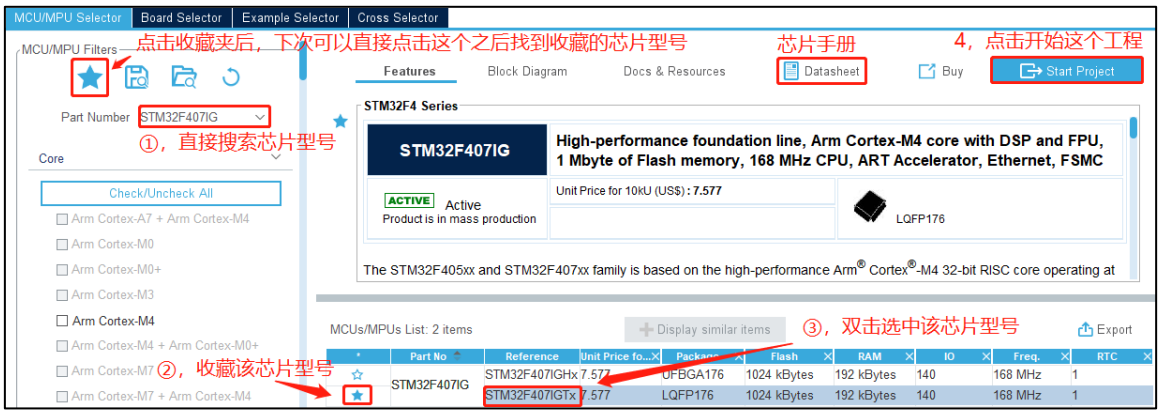


图 10.3.3.4 选择具体的芯片型号

选择了芯片型号后，弹出主设计界面，如图 10.3.3.5 所示。

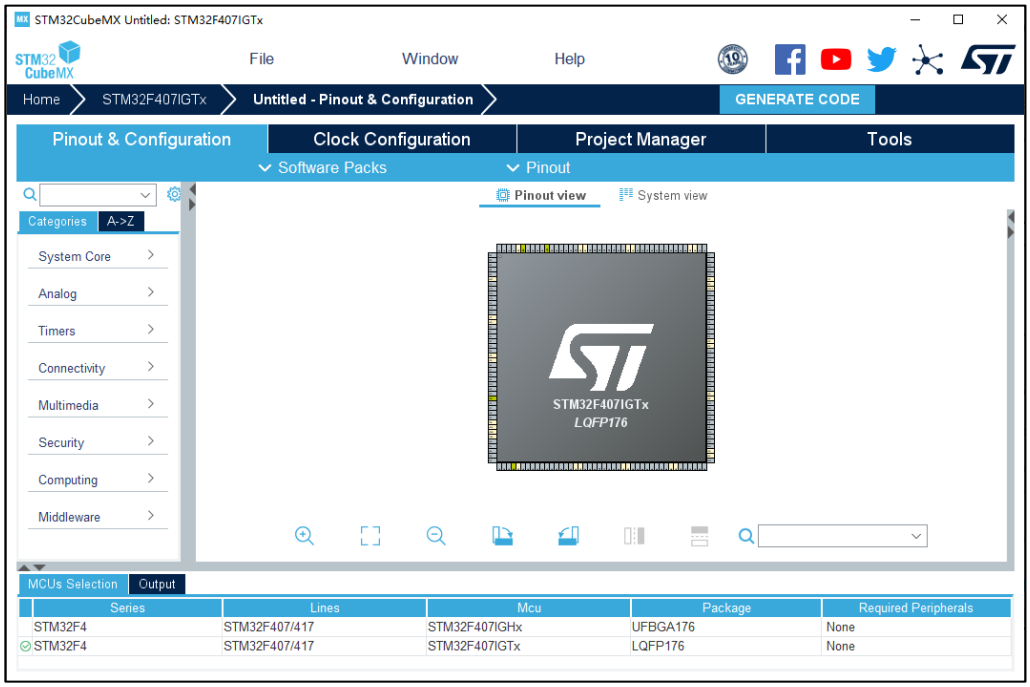


图 10.3.3.5 主设计界面

2 HSE 和 LSE 时钟源设置

进入工程主设计界面后，首先设置时钟源 HSE 和 LSE。如图 10.3.3.6 所示。

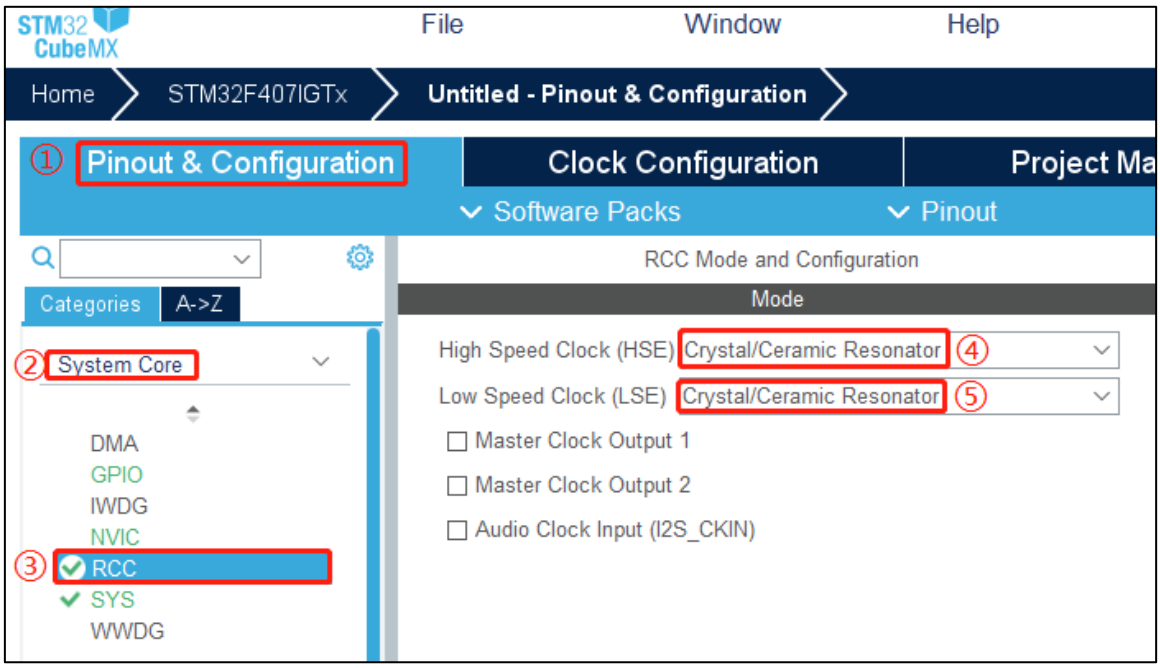


图 10.3.3.6 设置时钟源 HSE 和 LSE

图 10.3.3.6 中的标号 3 和 4，我们都选择了 Crystal/Ceramic Resonator，表示外部晶振作为它们的时钟源。我们开发板的外部高速晶振和外部低速晶振分别是：8MHz 和 32.768KHz，所以 HSE 时钟频率就是 8MHz，LSE 时钟频率就是 32.768KHz。

选项 Master Clock Output 1 用来选择是否使能 MCO1 引脚时钟输出。

3 时钟系统（时钟树）配置

点击 Clock Configuration 选项卡即可进入时钟系统配置栏，如下图 10.3.3.7 所示：

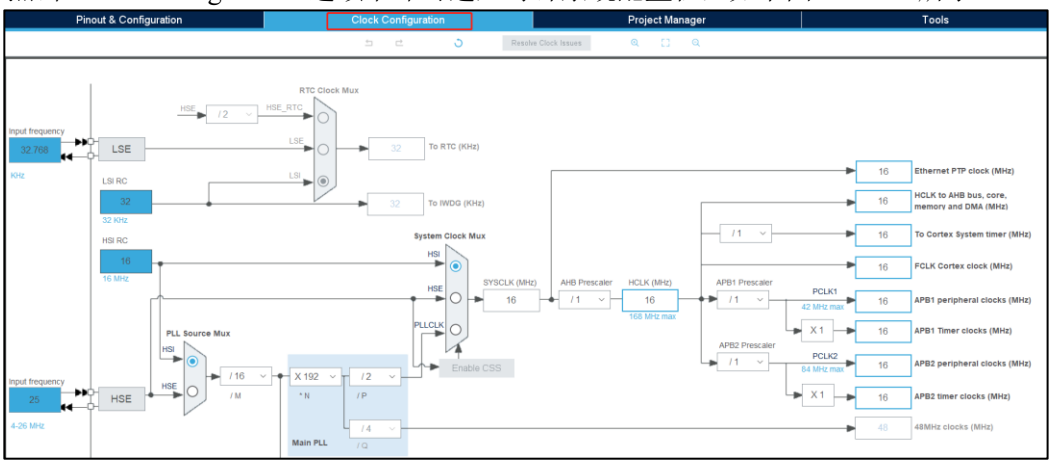


图 10.3.3.7 时钟系统配置栏

进入 Clock Configuration 配置栏之后可以看到，界面展现一个完整的 STM32F4 时钟系统框图。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，我们将配置一个以 HSE 为时钟源，配置 PLL 相关参数，然后系统时钟选择 PLLCLK 为时钟源，最终配置系统时钟为 168MHz 的过程。同时，还配置了 AHB，APB1，APB 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 SYSTICK、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 10.3.3.8 所示：

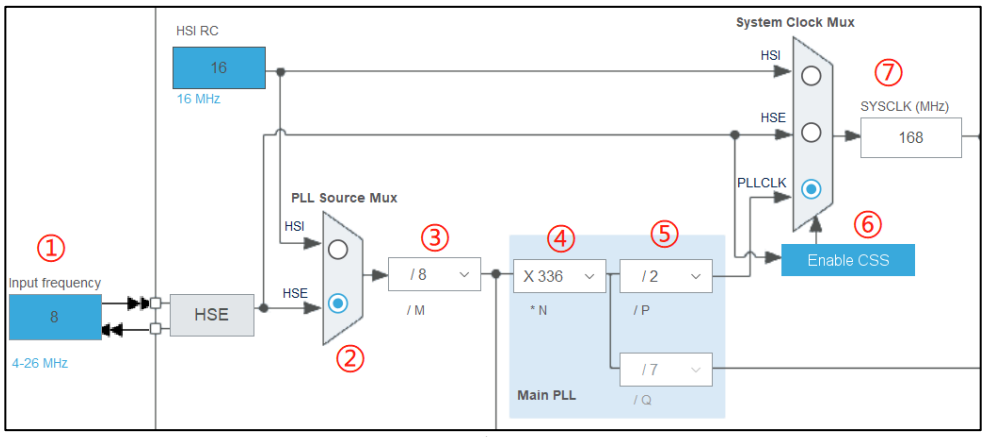


图 10.3.3.8 系统时钟配置图

我们把系统时钟配置分为七个步骤，分别用标号①~⑦表示，详细过程为：

① 时钟源参数设置：我们选择 HSE 为时钟源，所以我们要根据硬件实际的高速晶振频率（这里我们是 8MHz）填写。

② 时钟源选择：我们配置选择器选择 HSE 即可。

③ PLL1 分频系数 M 配置。分频系数 M 我们设置为 4。

④ PLL1 倍频系数 N 配置。倍频系数 N 我们设置为 168。

⑤ PLL1 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。我们选择 PLL，选择器选择 PLLCLK 即可。

⑦ 经过上面配置以后此时 SYSCLK=168MHz。

经过上面的 7 个步骤，就配置好 STM32F4 的系统时钟为 168MHz。接下来我们还需要配置 AHB、APB1、APB2 和 Systick 的分频系数，为 STM32 的片上外设或 M4 内核设置对应的工作时钟，为后续使用这些硬件功能做好准备。配置如下图 10.3.3.9 所示：

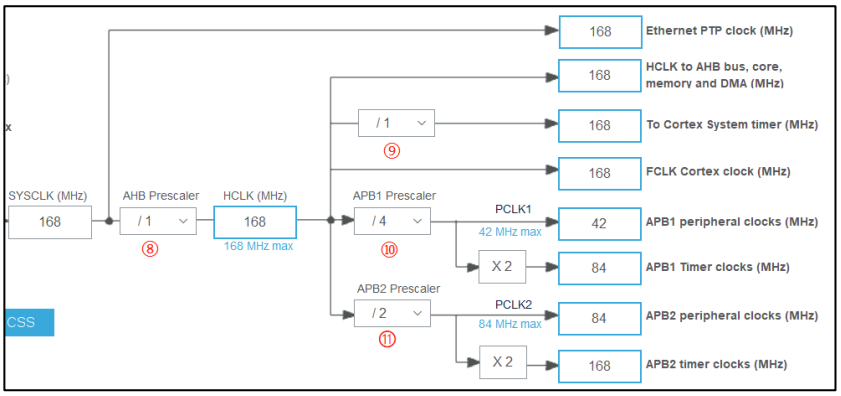


图 10.3.3.9 AHB、APB1、APB2 总线时钟配置

AHB、APB1 和 APB2 总线时钟以及 Systick 时钟的来源于系统时钟 SYSCLK。其中 AHB 总线时钟 HCLK 由 SYSCLK 经过 AHB 预分频器之后得到，如果我们要设置 HCLK 为 168MHz (最大为 168MHz)，那么我们只需要配置图中标号的地方为 1 即可。得到 HCLK 之后，接下来我们将在图标号⑨~⑪处同样的方法依次配置 Systick、APB1 和 APB2 分频系数分别为 1、4 和 2。注意！systick 固定为 168MHz，配置完成之后，那么 HCLK=168MHz，Systick=168MHz，PCLK1=42MHz，PCLK2=84MHz，这和之前例程配置的时钟频率是一样的。

以上方法是手动计算的方法，是为了帮助我们更好地去认识 STM32 时钟的配置方法，当然 CubeMX 也提供了更简单的方法：在图 10.3.3.9 的“HCLK(MHz)”位置，实际上是可以编辑的。我们直接输入我们要的时钟频率，这里是 168MHz，按回车键，CubeMX 会帮我们提供一种设置时钟频率和其它时钟的建议，选择是后会由软件自动配置好，当然只有启用外部的晶振后才能配置到 168MHz 的时钟，这里大家自己尝试一下就清楚了，我们不展开讲述了。

4 GPIO 功能引脚配置

本小节，我们讲解怎么使用 STM32CubeMX 工具配置 STM32F4 的 GPIO 口。STM32F407 电机开发板的 PE0 和 PE1 引脚各连接一个 LED 灯，我们来学习配置这两个 IO 口的相关参数。这里我们回到 STM32CubeMX 的 Pinout&Configuration 选项，在搜索栏输入 PE0 后回车，可以在引脚图中显示位置，如下图 10.3.3.11 所示：

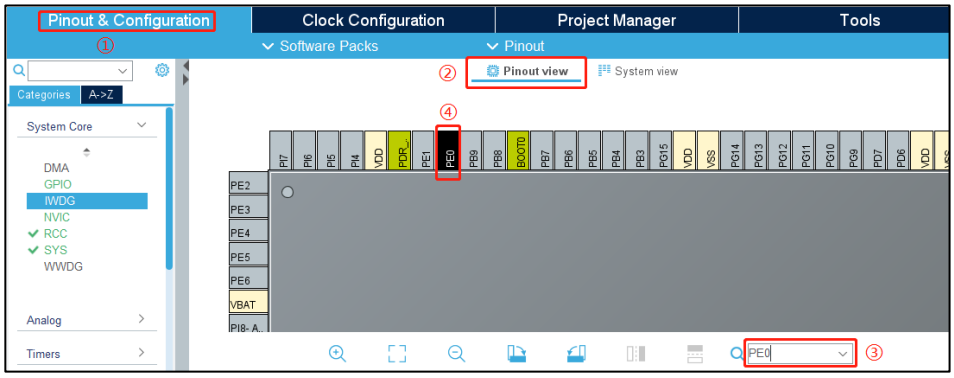


图 10.3.3.10 搜索引脚位置

接下来，我们在图 10.3.3.11 引脚图中点击 PE0，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 10.3.3.11 所示：

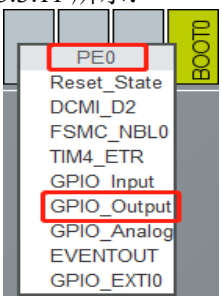
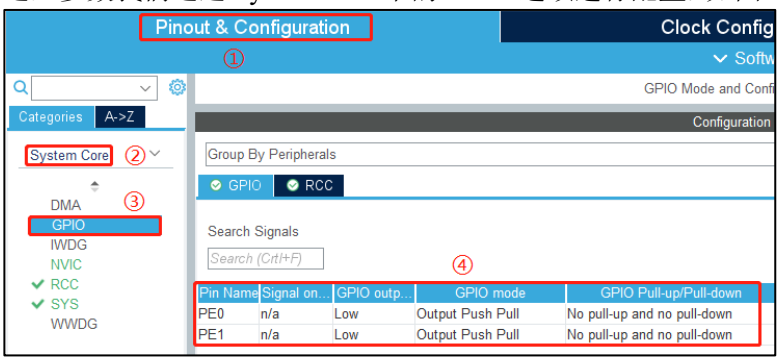


图 10.3.3.11 配置 GPIO 模式

同样的方法，我们配置 PE1 选择功能为 GPIO_Oput 即可。设置好即可看到引脚从灰色变成绿色，标识该管脚已经启用。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。这些参数我们通过 System Core 下的 GPIO 选项进行配置，如图 10.3.3.12 所示。



我们先配置 PE0。点击图 10.3.3.12 的④号框里面的 PE0，配置如图 10.3.3.13 所示。

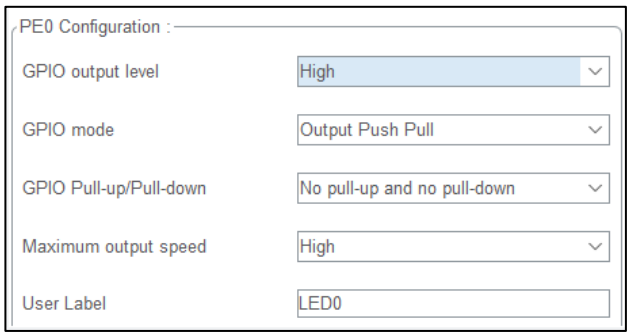


图 10.3.3.13 配置 GPIO 口详细参数

GPIO output level 是 IO 的初始值，由于 LED 一端接 VCC，另一端接 GPIO，故要点亮 LED 灯时，使 GPIO 输出低电平即可。为了一开始让 LED 灯熄灭，我们设置初始值输出高电平。

GPIO mode 我们已经在视图中配置为推挽输出了，这里不需要修改。

GPIO Pull-up/Pull-down 默认是无上下拉，我们这里用默认配置。

Maximum output speed 输出速度配置，默认是低速，我们设置为高速。

User Label 用户符号，我们可以给 PE0 起一个别名 LED0。

PE1 也是按照这样的方法配置,给 PE1 起一个别名 LED1。

5 配置 Debug 选项

由于 CubeMX 默认把 Debug 选项关闭了，这样会给我们带来麻烦：用 CubeMX 生成的工程编译下载一次后，后续再次下载就会提示错误，因此我们要把 Debug 选项打开。这里有多种选择，我们设置成图 10.3.3.14 所示的情况即可。

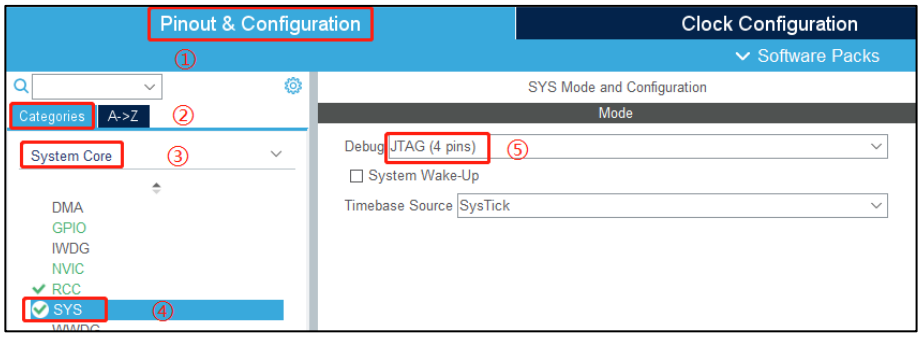


图 10.3.3.14 打开 Debug 选项

如果已经不小心关闭了 Debug 选项，那么下次下载的时候按住复位键，等到工程提示的时候松开复位键即可，因为 STM32 的芯片默认复位上电时的 Debug 引脚功能是开启的。

接下来我们学习怎么设置生成一个工程，如图 10.3.3.15 所示。选择 Project Manager-> Project 选项用来配置工程的选项，我们了解一下里面的信息。

Project Name：工程名称，填入工程名称（半角，不能有中文字符）

Project Location：工程保存路径，点击 Browse 选择保存的位置（半角，不能有中文字符）

Toolchain Folder Location:工具链文件夹位置，默认即可。

Application Structure：应用的结构，选择 Basic（基础），不勾选 Do not generate the main(), 因为我们要其生成 main 函数。

Toolchain/IDE:工具链/集成开发环境，我们使用 Keil，因此选择 MDK-ARM，Min Version 选择 V5.27，这里根据 CubeMX 的版本可能会有差异，我们默认使用 V5 以上的版本即可。

Linker Settings 链接器设置：

Minimum Heap Size 最小堆大小，默认（大工程需按需调整）。

Minimum Stack Size 最小栈大小，默认（大工程需按需调整）。

MCU and Firmware Package 是 MCU 及固件包设置：

MCU Reference：目标 MCU 系列名称。

Firmware Package Name and Version：固件包名称及版本。

勾选 Use Default Firmware Location，文本框里面的路径就是固件包的存储地址，我们使用默认地址即可。这样工程生成的设置就设置好了。

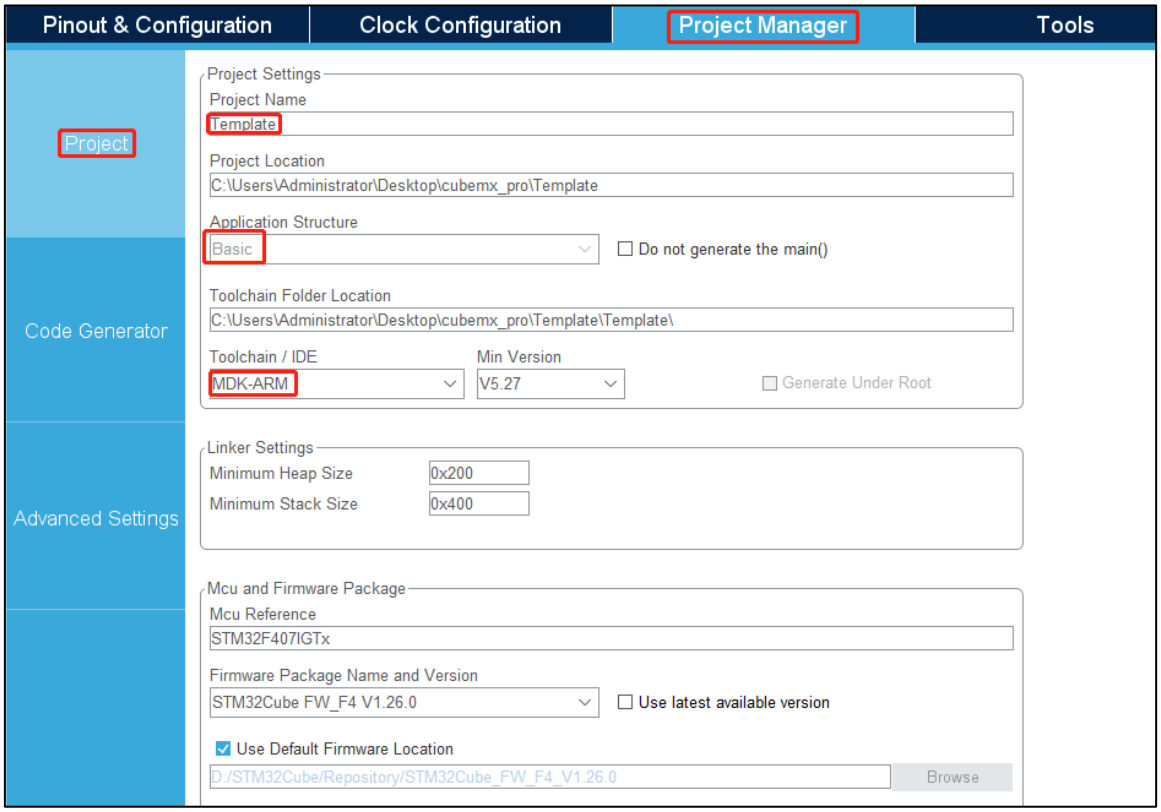


图 10.3.3.15 工程配置

接下来我们配置生成代码的配置选项。打开 Project Manager-> Code Generator 选项，Generated files 生成文件选项，勾选 Generate peripheral initialization as a pir of '.c/.h'files per peripheral，勾选这个选项的话将会将每个外设单独分开成一组.c、.h 文件，使得代码结构更加的清晰，如图 10.3.3.16 所示。

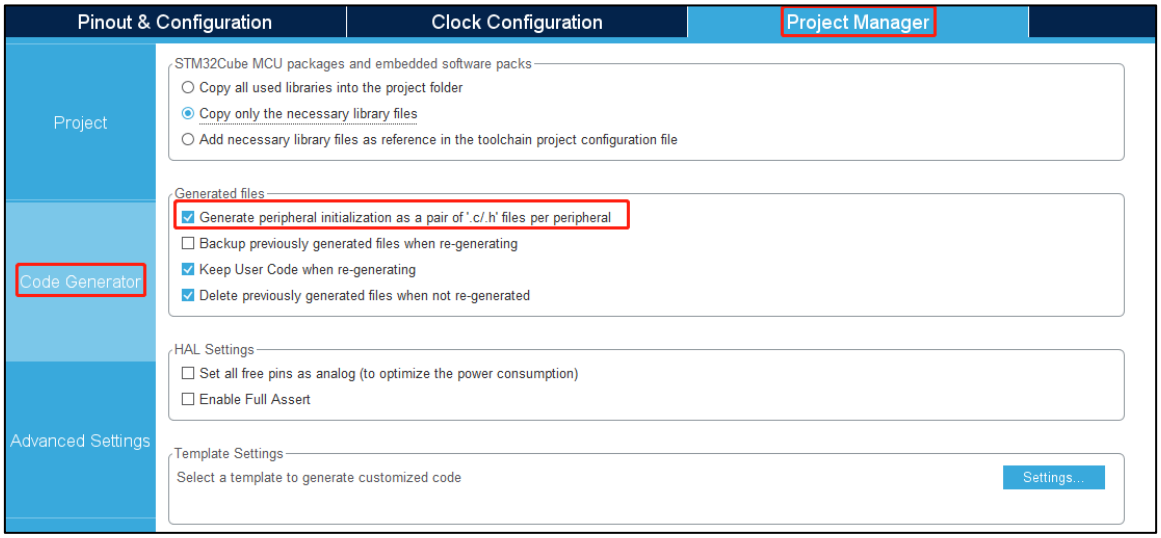


图 10.3.3.16 代码生成器设置

至此工程最基础配置就已经完成，点击蓝色按钮(SENERATE CODE)就可以生成工程。

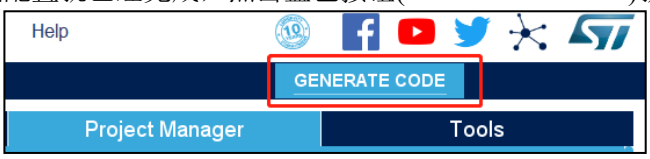


图 10.3.3.17 生成工程

在弹出来的窗口中点击 Open Project 就打开 MDK 工程。

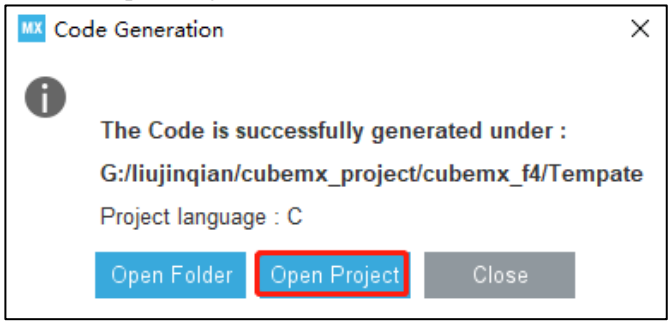


图 10.3.3.18 打开工程

完整的 STM32F4 工程就已经生成完成。生成后的工程目录结构如下图 10.3.3.19 所示：

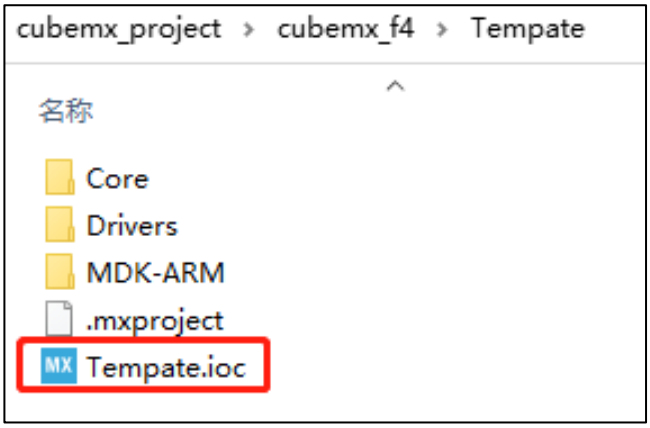


图 10.3.3.19 STM32CubeMX 生成的工程目录结构

Drivers 文件夹存放的是 HAL 库文件和 CMSIS 相关文件。

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，双击该文件就会在 STM32CubeMX 中打开。

7 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，因为我们在之前步骤生成的 CubeMX 工程为 LED_TEST.ioc，故生成的 MDK 工程位置是.\MDK-ARM\Template.uvprojx，如果大家配置的 CubeMX 的工程名和路径名不含中文或中文字符，按上述步骤生成的工程就可以直接编译通过了。

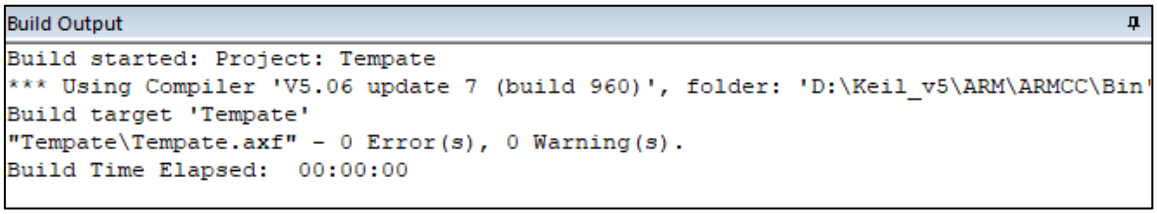


图 10.3.3.20STM32CubeMX 生成的 MDK 工程编译通过

接下来我们中生成的工程模板的 main.c 文件中找到 main 函数，这里我们删掉了源码注释，关键源码如下：

```
int main(void)
{
  HAL_Init();
  SystemClock_config();
  MX_GPIO_Init();
  /* USER CODE BEGIN WHILE */
  while (1)
```

```
  {
    /* USER CODE END WHILE */
  }
}
```

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。

我们编写一个跑马灯的用户程序，程序具体如下：

```
/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);
    HAL_Delay(500);
    HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);
    HAL_Delay(500);
    /* USER CODE END WHILE */
  }
}
```

编写好程序后，编译没有任何警告和错误。可以直接下载程序到开发板中，使用 DAP 下载，请注意设置 MDK 的下载选项，如果不清楚设置的读者可以回看本书第四章的相关知识点。下载后，可以看到 LED0 和 LED1 同时按 500ms 的频率亮灭，效果与其它版本的新建工程相同。

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\1，标准例程-库函数版本\实验 0 基础入门实验\实验 0-4，新建工程实验-CubeMX 版本”**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。

10.4 STM32CubeMX 新建工程使用建议

① 使用 CubeMX 的环境搭建工程，工程文件夹路径、文件名不要带任何中文及中文字符，否则会遇到各种报错；

② 本书以新建工程-HAL 库版本为基准来展开，不对 CubeMX 的使用过多讲解。使用 CubeMX 可以帮助我们快速搭建工程，使用户专注于应用开发，但 STM32 的开发与硬件密切相关，对 STM32 开发来说，抛开底层只专注做应用并不实际，毕竟无法使用一套通用设计来满足不同用户的需求；

③ 关于新建 CubeMX 的工程路径中有中文的情况的解决：

如果我们配置的 CubeMX 工程路径里面有中文可能会报以下的错误：

```
Build Output
linking...
LED_TEST\LED_TEST.sct(7): error: L6236E: No section matches selector - no section to be FIRST/LAST.
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
"LED_TEST\LED_TEST.axf" - 1 Error(s), 0 Warning(s).
Target not created.
```

图 10.4.1 直接编译报错

造成错误的原因是 CubeMX 对中文的支持不友好，且生成的 MDK 工程默认通过工程中的 CMSIS 那个绿色的控件选择启动文件而不是直接添加启动文件(startup_xxx.s)到我们的工程中，而有中文路径时就会找不到，有两个解决办法：

1) 用 CubeMX 生成的工程不要放置在包含中文路径的文件夹下；

2) 添加启动文件到我们的工程中，我们新建一个 Application/MDK-ARM 分组，把 startup_stm32f407xx.s 添加到这个分组，如图 10.4.2 所示：

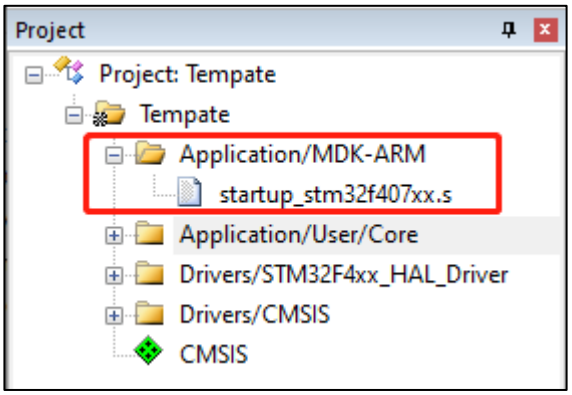


图 10.4.2 STM32CubeMX 生成的工程目录结构

3) 关于配置的文件 CubeMX 工程(.ioc 后缀)名字有中文的情况，我们建议重新新建工程或者把生成的工程文件重命名为英文。因为带中文的 CubeMX 工程生成的 MDK 的 Output 目录有中文，MDK 也会报错，尽管可以重新设置 MDK 工程的 Output 目录和添加③所描述步骤的启动文件，使本次编译通过，但下次重新用 CubeMX 生成工程时，仍旧需要重复修改配置。

第十一章 STM32 时钟系统

STM32F4 时钟系统的知识在《STM32F4xx 参考手册_V6（英文版）.pdf》第七章复位和时钟控制章节有较详细的讲解。这里我们对 STM32F4 的整体架构作一个简单的介绍，帮助大家更全面、系统地认识 STM32F4 系统的主控结构。了解时钟系统在整个 STM32 系统的贯穿和驱动作用，学会设置 STM32 的系统时钟。

本章将分为如下几个小节：

11.1 认识时钟树

11.2 如何修改主频

11.1 认识时钟树

数字电路的知识告诉我们：任意复杂的电路控制系统都可以经由门电路组成的组合电路实现。回顾《第五章 STM32 基础知识入门》的知识点，我们知道 STM32 内部也是由多种多样的电路模块组合在一起实现的。当一个电路越复杂，在达到正确的输出结果前，它可能因为延时会有一些短暂的中间状态，而这些中间状态有时会导致输出结果会有一个短暂的错误，这叫做电路中的“毛刺现象”，如果电路需要运行得足够快，那么这些错误状态会被其它电路作为输入采样，最终形成一系列的系统错误。为了解决这个问题，在单片机系统中，设计时以时序电路控制替代纯粹的组合电路，在每一级输出结果前对各个信号进行采样，从而使得电路中某些信号即使出现延时也可以保证各个信号的同步，可以避免电路中发生的“毛刺现象”，达到精确控制输出的效果。

由于时序电路的重要性，因此在 MCU 设计时就设计了专门用于控制时序的电路，在芯片设计中称为时钟树设计。由此设计出来的时钟，可以精确控制我们的单片机系统，这也是我们这节要展开分析的时钟分析。为什么是时钟树而不是时钟呢？一个 MCU 越复杂，时钟系统也会相应地变得复杂，如 STM32F4 的时钟系统比较复杂，不像简单的 51 单片机一个系统时钟就可以解决一切。对于 STM32F4 系列的芯片，正常工作的主频可以达到 168Mhz，但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 Khz 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。

STM32 本身非常复杂，外设非常的多，为了保持低功耗工作，STM32 的主控默认不开启这些外设功能。用户可以根据自己的需要决定 STM32 芯片要使用的功能，这个功能开关在 STM32 主控中也就是各个外设的时钟。

下面来看一下 STM32F4 时钟系统图。STM32F4 时钟系统图看起来有点多且复杂，但是分开几部分各个理解其实没有那么难了。

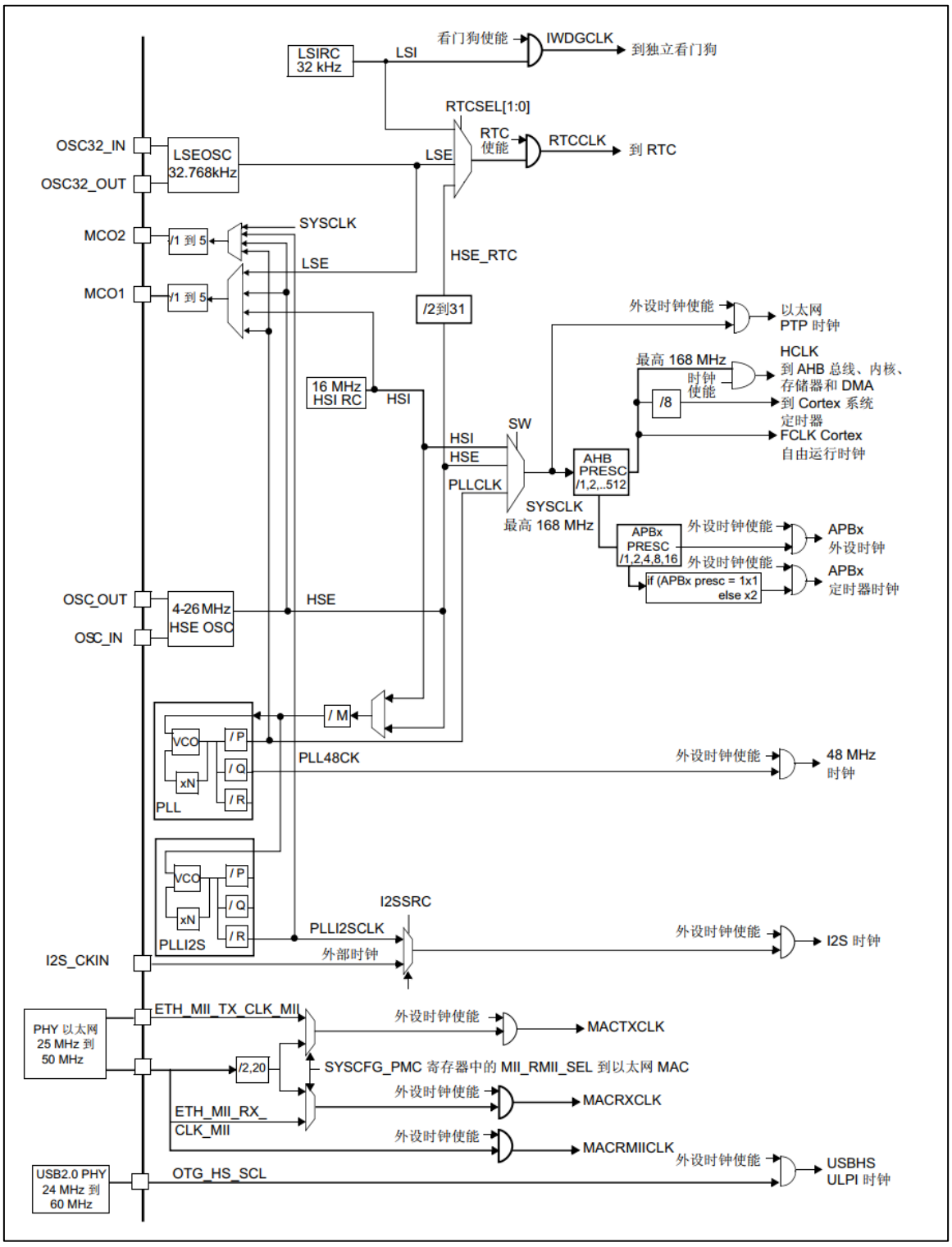


图 11.1.1 STM32F4 时钟系统图

我们将时钟系统分为：时钟源，锁相环 PLL，系统时钟 SYSCLK，时钟信号输出 MCO 四个部分展开介绍。

11.1.1 时钟源

对于 STM32F4，输入时钟源（Input Clock）主要包括 HSI，HSE，LSI，LSE。其中，从时钟频率来分可以分为高速时钟源和低速时钟源，其中 HSI、HSE 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；其他是内部时钟源，芯片上电即可产生，不需要借助外部电路。下面我们看看 STM32 的时钟源。

（1）2 个外部时钟源：

- **高速外部振荡器 HSE (High Speed External Clock signal)**
  外接石英/陶瓷谐振器，频率为 4MHz~26MHz。本开发板使用的是 8MHz。
- **低速外部振荡器 LSE (Low Speed External Clock signal)**
  外接 32.768kHz 石英晶体，主要作用于 RTC 的时钟源。

（2）2 个内部时钟源：

- **高速内部振荡器 HSI(High Speed Internal Clock signal)**
  由内部 RC 振荡器产生，频率为 16MHz。
- **低速内部振荡器 LSI(Low Speed Internal Clock signal)**
  由内部 RC 振荡器产生，频率为 32kHz，可作为独立看门狗的时钟源。

芯片上电时默认由内部的 HSI 时钟启动，如果用户进行了硬件和软件的配置，芯片才会根据用户配置调试尝试切换到对应的外部时钟源，所以同时了解这几个时钟源信号还是很有必要的。如何设置时钟的方法我们会在后文提到。

11.1.2 锁相环 PLL

锁相环是自动控制系统中常用的一个反馈电路，在 STM32 主控中，锁相环的作用主要有两个部分：输入时钟净化和倍频。前者是利用锁相环电路的反馈机制实现，后者我们用于使芯片在更高且频率稳定的时钟下工作。

在 STM32 中，锁相环的输出也可以作为芯片系统的时钟源。根据图 11.1.1 的时钟结构，使用锁相环时只需要进行三个部分的配置。为了方便查看，截取了使用 PLL 作为系统时钟源的配置部分，如图 11.1.2.1 所示。

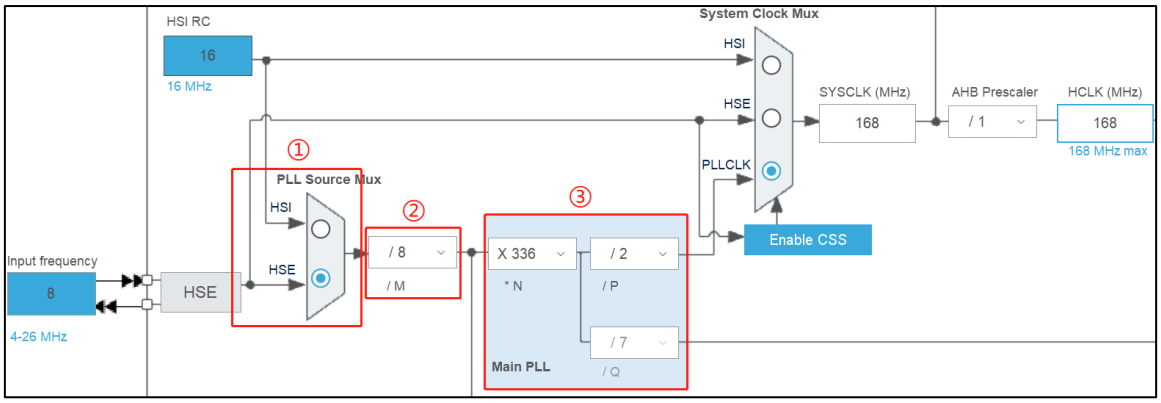


图 11.1.2.1 PLL 时钟配置图

◆ PLL Source Mux：PLL 时钟源选择器

图中标号①表示的是 PLL 时钟源的选择器，同样的，参考 F407 参考手册：

位 22 **PLLSRC**：主 PLL(PLL) 和音频 PLL (PLLI2S) 输入时钟源 (Main PLL(PLL) and audio PLL (PLLI2S) entry clock source)

由软件置 1 和清零，用于选择 PLL 和 PLLI2S 时钟源。此位只有在 PLL 和 PLLI2S 已禁止时才可写入。

0：选择 HSI 时钟作为 PLL 和 PLLI2S 时钟输入

1：选择 HSE 振荡器时钟作为 PLL 和 PLLI2S 时钟输入

图 11.1.2.3 PLLSRC 锁相环时钟源选择

它有两种可选择的输入源：一个是内部时钟 HSI 信号，另一个是外部时钟 HSE 信号。

◆ PLLM：HSE 分频器作为 PLL 输入 (HSE divider for PLL entry)

即图 11.1.2.1 在标号②的地方，主 PLL 输入时钟的分频系数，并把它的控制功能放在 RCC_PLLCFGR 寄存器中，我们引用如图 11.1.2.2。

位 5:0 **PLLM**：主 PLL (PLL) 和音频 PLL (PLLI2S) 输入时钟的分频系数 (Division factor for the main PLL (PLL) and audio PLL (PLLI2S) input clock) 由软件置 1 和清零，用于在 VCO 之前对 PLL 和 PLLI2S 输入时钟进行分频。这些位只有在 PLL 和 PLLI2S 已禁止时才可写入。

图 11.1.2.2 PLLXTPRE 设置选项值

从 F407 参考手册可知它的值有 64 个：0~63。

◆ PLLMUL：PLL 倍频系数 (PLL multiplication factor)

图中③所表示的配置锁相环倍频系数，同样地可以查到在 STM32F4 系列中，ST 设置它的有效倍频范围为 192~432 倍。

结合图 11.1.2.1，要实现 168MHz 的主频率，我们通过选择 HSE 分频作为 PLL 输入的时钟信号，输入 8Mhz，8 分频，即 1MHz，通过标号③选择倍频因子，我们选择 336 倍频，这样可以得到时钟信号为 1*336=336MHz，然后经过 2 分频，得到 168MHz。

11.1.3 系统时钟 SYSCLK

STM32 的系统时钟 SYSCLK 为整个芯片提供了时序信号。我们已经大致知道 STM32 主控是时序电路链接起来的。对于相同的稳定运行的电路，时钟频率越高，指令的执行速度越快，单位时间能处理的功能越多。STM32 的系统时钟是可配置的，在 STM32F4 系列中，它可以为 HSI、PLLCLK、HSE 中的一个，通过 CFGR 的位 SW[1:0]设置。

讲解 PLL 作为系统时钟时，根据我们开发板的资源，可以把主频通过 PLL 设置为 168MHz。从上面的图 11.2.1 时钟树图可知，AHB、APB1、APB2、内核时钟等时钟通过系统时钟分频得到。根据得到的这个系统时钟，下面我们结合外设来看一看各个外设时钟源。

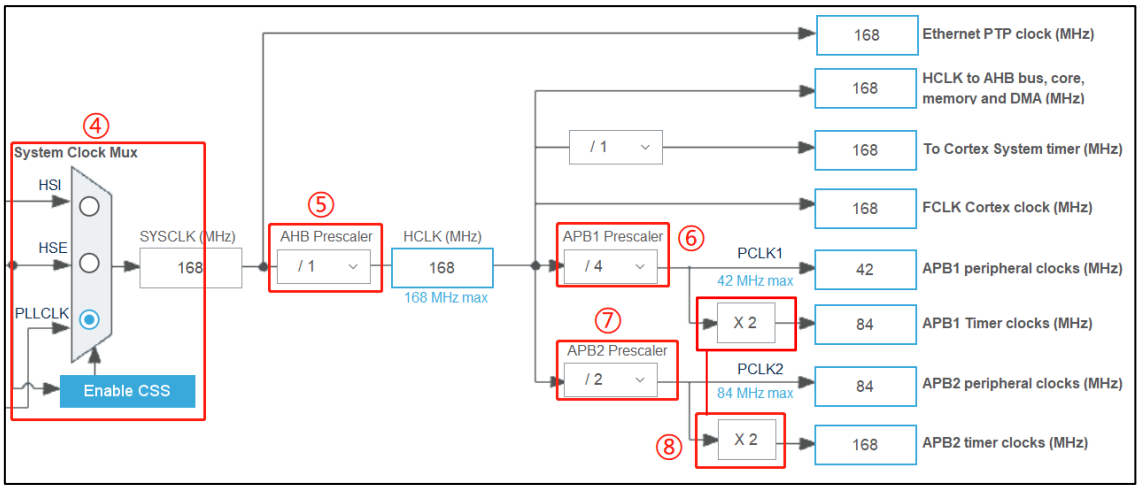


图 11.2.3.1 STM32F407 系统时钟生成图

大家看图 11.2.3.1 STM32F407 系统时钟，标号④为系统时钟输入选择，可选时钟信号有外部高速时钟 HSE(8M)、内部高速时钟 HSI(16M)和经过倍频的 PLL CLK(168M)。这里我们选择 PLL CLK 作为系统时钟，此时系统时钟的频率为 168MHz。系统时钟来到标号⑤的 AHB 预分频器，其中可选择的分频系数为 1，2，4，8，16，32，64，128，256，512，我们选择不分频，所以 AHB 总线时钟达到最大的 168MHz。

下面介绍一下由 AHB 总线时钟得到的时钟：

APB1 总线时钟，由 HCLK 经过标号⑥的低速 APB1 预分频器得到，分频因子可以选择 1，2，4，8，16，这里我们选择的是 4 分频，所以 APB1 总线时钟为 42M。由于 APB1 是低速总线时钟，APB1 总线最高频率为 42MHz，片上低速的外设就挂载在该总线上，例如有看门狗定时器、定时器 2/3/4/5/6/7、RTC 时钟、USART2/3/4/5、SPI2(I2S2)与 SPI3(I2S3)、I2C1~3、CAN 和 2 个 DAC。

APB2 总线时钟，由 HCLK 经过标号⑦的高速 APB2 预分频器得到，分频因子可以选择 1，2，4，8，16，这里我们选择的是 2 分频，所以 APB2 总线时钟频率为 84M。与 APB2 高速总线连接的外设有定时器 1、8、9、10、11，SPI1、USART1 和 USART6、3 个 ADC 和 SDIO 接口。

其中标号⑧决定了定时器时钟频率，该位由硬件自动设置，分为两种情况：

1、 如果 APB 预分频器为 1，定时器时钟频率等于 APB 域的频率；

2、 否则，等于 APB 域的频率的两倍（×2）。

此外，AHB 总线时钟直接作为 GPIO(A\B\C\D\E\F\G\H\I\)、以太网、DCMI、FSMC、AHB 总线、Cortex 内核、存储器和 DMA 的 HCLK 时钟，并作为 Cortex 内核自由运行时钟 FCLK。

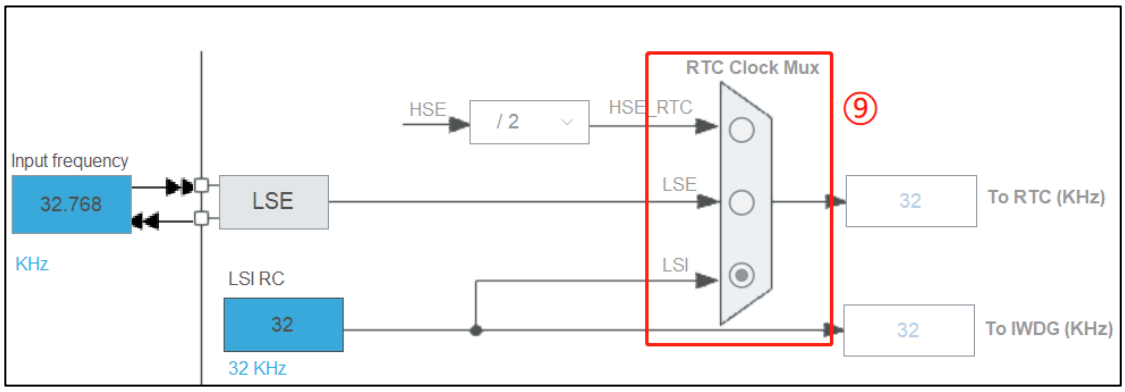


图 11.2.3.2 RTC 相关时钟

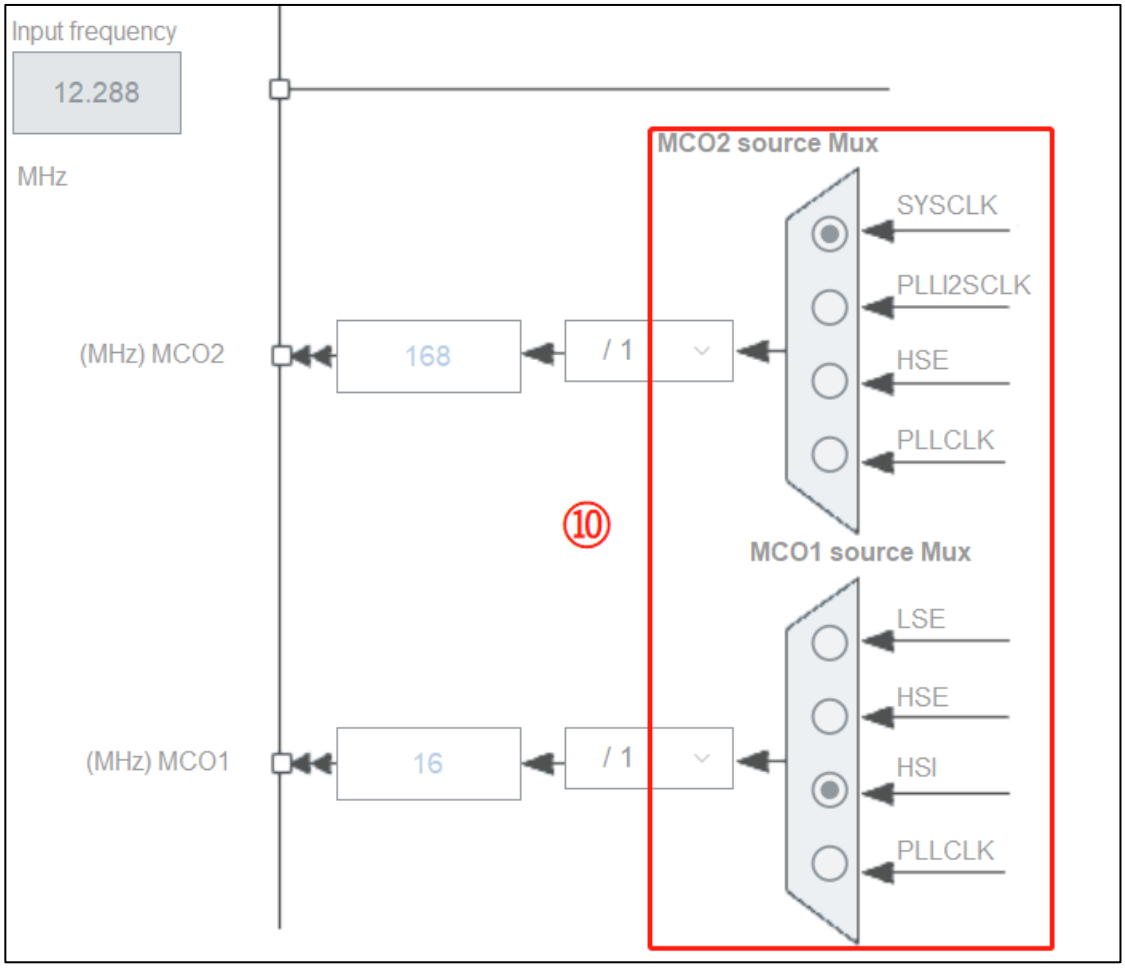


图 11.2.3.3 MACO 相关时钟

图 11.2.3.2 标号⑨是 RTC 定时器，其时钟源为 HSE/x（x = 2~31）、LSE 或 LSI。

图 11.2.3.3 标号⑩是 MCO1/MCO2 的钟源选择器，MCOx（x = 1 或 2）是 STM32 的一个时钟输出 IO（分别由 PA8，PC9 的复用功能实现）。STM32 允许通过设置，让 MCO 引脚输出一个稳定的时钟信号。

MCO1（外部器件的输出时钟 1）时钟源有四个：HSI、LSE、HSE、PLL;

MCO2（外部器件的输出时钟 2）时钟源有四个：SYSCLK、HSI、HSE、PLLI2S.

时钟源的选择由时钟配置寄存器(RCC_CFGR)中的 MCO1[1:0]和 MCO2[1:0]位控制。MCO1/MCO2 的预分频器取值范围均为：1~5，由时钟配置寄存器(RCC_CFGR)中的 MCO1PRE[2:0]和 MCO2PRE[2:0]位控制。

我们可以通过 MCO 引脚来输出时钟信号，测试输出时钟的频率，或作为其它外部电路的时钟源。

11.2 如何修改主频

STM32F407 默认的情况下（比如：串口 IAP 时或者是未初始化时钟时），使用的是内部 16M 的 HSI 作为时钟源，所以不需要外部晶振也可以下载和运行代码的。

下面我们来讲解如何让 STM32F407 芯片在 168MHz 的频率下工作，168MHz 是官方推荐使用的最高且稳定的时钟频率。而正点原子的 DMF407 的外部高速晶振的频率就是 8MHz，我们就是在这个晶振频率的基础上，通过各种倍频和分频得到 168MHz 的系统工作频率。

11.2.1 STM32F4 时钟系统配置

下面我们将分几步给大家讲解 STM32F4 时钟系统配置过程，这部分内容很重要，请大家认真阅读。

第 1 步：配置 HSE_VALUE

讲解 STM32F4xx_hal_conf.h 文件的时候，我们知道需要宏定义 HSE_VALUE 匹配我们实际硬件的高速晶振频率(这里是 8MHz)，代码中通过使用宏定义的方式来选择 HSE_VALUE 的值是 25M 或者 8M 选择定义 HSE_VALUE 的值为 8M。代码如下：

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

第 2 步：调用 SystemInit 函数

我们介绍启动文件的时候就知道，在系统启动之后，程序会先执行 SystemInit 函数，进行系统一些初始化配置。启动代码调用 SystemInit 函数如下：

```
Reset_Handler    PROC
                 EXPORT  Reset_Handler                 [WEAK]
                 IMPORT  SystemInit
                 IMPORT  __main

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

下面我们来看看 system_stm32f4xx.c 文件下定义的 SystemInit 程序，源码在 168 行到 183 行，函数如下。

```
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

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

  /* Configure the Vector Table location -------------------------------------*/
#if defined(USER_VECT_TAB_ADDRESS)
```

```
  SCB->VTOR = VECT_TAB_BASE_ADDRESS | VECT_TAB_OFFSET; /* Vector Table
Relocation in Internal SRAM */
#endif /* USER_VECT_TAB_ADDRESS */
}
```

从上面代码可以看出，SystemInit 主要做了如下两个方面工作：

1) 外部存储器配置

2) 中断向量表地址配置

然而我们的代码中实际并没有定义 DATA_IN_ExtSRAM 和 USER_VECT_TAB_ADDRESS 这两个宏，实际上 SystemInit 对于正点原子的例程并没有起作用，但我们保留了这个接口。从而避免了去修改启动文件。另外，是可以把一些重要的初始化放到 SystemInit 这里，在 main 函数运行前就把重要的一些初始化配置好（如 ST 这里是在运行 main 函数前先把外部的 SRAM 初始化），这个我们一般用不到，直接到 main 函数中处理即可，但也有厂商（如 RT-Thread）就采取了这样的做法，使得 main 函数更加简单，但对于初学者，我们暂时不建议这种用法。

HAL 库的 SystemInit 函数并没有任何时钟相关配置，所以后续的初始化步骤，我们还必须编写自己的时钟配置函数。

第 3 步：在 main 函数里调用用户编写的时钟设置函数

我们打开 HAL 库例程实验 1 跑马灯实验，看看我们在工程目录 Drivers\SYSTEM 分组下面定义的 sys.c 文件中的时钟设置函数 sys_stm32_clock_init 的内容：

```
/**
 * @brief       时钟设置函数
 * @param       plln: PLL1 倍频系数(PLL 倍频), 取值范围: 64~432.
 * @param       pllm: PLL1 预分频系数(进 PLL 之前的分频), 取值范围: 2~63.
 * @param       pllp: PLL1 的 p 分频系数(PLL 之后的分频), 分频后作为系统时钟, 取值范围:
 * 2,4,6,8.(仅限这 4 个值!)
 * @param       pllq: PLL1 的 q 分频系数(PLL 之后的分频), 取值范围: 2~15.
 * @note
 *
 *              Fvco: VCO 频率
 *              Fsys: 系统时钟频率, 也是 PLL1 的 p 分频输出时钟频率
 *              Fq:   PLL1 的 q 分频输出时钟频率
 *              Fs:   PLL 输入时钟频率, 可以是 HSI, CSI, HSE 等.
 *              Fvco = Fs * (plln / pllm);
 *              Fsys = Fvco / pllp = Fs * (plln / (pllm * pllp));
 *              Fq   = Fvco / pllq = Fs * (plln / (pllm * pllq));
 *
 *              外部晶振为 8M 的时候, 推荐值: plln = 336, pllm = 8, pllp = 2, pllq = 7.
 *              得到:Fvco = 8 * (336 / 8) = 336Mhz
 *                   Fsys = pll1_p_ck = 336 / 2 = 168Mhz
 *                   Fq   = pll1_q_ck = 336 / 7 = 48
 *
 * @retval      错误代码: 0, 成功; 1, 错误;
 */
uint8_t sys_stm32_clock_init(uint32_t plln, uint32_t pllm, uint32_t pllp,
uint32_t pllq)
{
    HAL_StatusTypeDef ret = HAL_OK;
    RCC_ClkInitTypeDef rcc_clk_init_handle;
    RCC_OscInitTypeDef rcc_osc_init_handle;

    __HAL_RCC_PWR_CLK_ENABLE();    /* 使能 PWR 时钟 */

/* 下面这个设置用来设置调压器输出电压级别，以便在器件以最大频率工作时使性能与功耗实现平衡 */

    /* VOS = 1, Scale1, 1.2V 内核电压,FLASH 访问可以得到最高性能 */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /* 使能 HSE，并选择 HSE 作为 PLL 时钟源，配置 PLL1，开启 USB 时钟 */
```

```
    rcc_osc_init_handle.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    rcc_osc_init_handle.HSEState = RCC_HSE_ON;                    /* 打开 HSE */
    rcc_osc_init_handle.PLL.PLLState = RCC_PLL_ON;                /* 打开 PLL */
    rcc_osc_init_handle.PLL.PLLSource = RCC_PLLSOURCE_HSE;        /* PLL 时钟源选择 HSE */
    rcc_osc_init_handle.PLL.PLLN = plln;
    rcc_osc_init_handle.PLL.PLLM = pllm;
    rcc_osc_init_handle.PLL.PLLP = pllp;
    rcc_osc_init_handle.PLL.PLLQ = pllq;

    ret = HAL_RCC_OscConfig(&rcc_osc_init_handle);                /* 初始化 RCC */
    if(ret != HAL_OK)
    {
        return 1;    /* 时钟初始化失败，可以在这里加入自己的处理 */
    }

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

    /* 设置系统时钟时钟源为 PLL */
    rcc_clk_init_handle.SYSCLKSource   = RCC_SYSCLKSOURCE_PLLCLK;
    rcc_clk_init_handle.AHBCLKDivider  = RCC_SYSCLK_DIV1;   /* AHB 分频系数为 1 */
    rcc_clk_init_handle.APB1CLKDivider = RCC_HCLK_DIV4;     /* APB1 分频系数为 4 */
    rcc_clk_init_handle.APB2CLKDivider = RCC_HCLK_DIV2;     /* APB2 分频系数为 2 */

    /* 同时设置 FLASH 延时周期为 5WS，也就是 6 个 CPU 周期 */
    ret = HAL_RCC_ClockConfig(&rcc_clk_init_handle, FLASH_LATENCY_5);
    if(ret != HAL_OK)
    {
        return 1;    /* 时钟初始化失败 */
    }

    /* STM32F405x/407x/415x/417x Z 版本的器件支持预取功能 */
    if (HAL_GetREVID() == 0x1001)
    {
        __HAL_FLASH_PREFETCH_BUFFER_ENABLE();     /* 使能 flash 预取 */
    }
    return 0;
}
```

函数 sys_stm32_clock_init 就是用户的时钟系统配置函数，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 STM32F4 时钟系统的一般步骤：

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

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

下面我们详细讲解这个 2 个步骤。

步骤 1：配置时钟源相关参数，使能并选择 HSE 作为 PLL 时钟源，配置 PLL1，我们调用的函数为 HAL_RCC_OscConfig()，该函数在 HAL 库头文件 STM32F4xx_hal_rcc.h 中声明，在文件 STM32F4xx_hal_rcc.c 中定义。首先我们来看看该函数声明：

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

该函数只有一个形参，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 HSEPredivValue;        /* HSE 预分频值 */
```

```
  uint32_t LSEState;              /* LSE 状态 */
  uint32_t HSIState;              /* HSI 状态 */
  uint32_t HSICalibrationValue;   /* HSI 校准值 */
  uint32_t LSIState;              /* LSI 状态 */
  RCC_PLLInitTypeDef PLL;         /* PLL 配置 */
}RCC_OscInitTypeDef;
```

该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、LSE，配置方法类似。

RCC_OscInitTypeDef 这个结构体还有一个很重要的成员变量是 PLL，它是结构体 RCC_PLLInitTypeDef 类型。它的作用是配置 PLL 相关参数，我们来看看它的定义：

```
typedef struct
{
  uint32_t PLLState;     /* PLL 状态 */
  uint32_t PLLSource;    /* PLL 时钟源 */
  uint32_t PLLMUL;       /* PLL 倍频系数 M */
}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。这个结构体的定义的相关内容请结合时钟树中红色框的内容一起理解。

接下来我们看看我们的时钟初始化函数 sys_stm32_clock_init 中的配置内容：

```
    /* 使能 HSE，并选择 HSE 作为 PLL 时钟源，配置 PLL1，开启 USB 时钟 */
    rcc_osc_init_handle.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    rcc_osc_init_handle.HSEState = RCC_HSE_ON;                    /* 打开 HSE */
    rcc_osc_init_handle.PLL.PLLState = RCC_PLL_ON;                /* 打开 PLL */
    rcc_osc_init_handle.PLL.PLLSource = RCC_PLLSOURCE_HSE; /* PLL 时钟源选择 HSE */
    rcc_osc_init_handle.PLL.PLLN = plln;
    rcc_osc_init_handle.PLL.PLLM = pllm;
    rcc_osc_init_handle.PLL.PLLP = pllp;
    rcc_osc_init_handle.PLL.PLLQ = pllq;
```

通过函数的该段程序，我们开启了 HSE 时钟源，同时选择 PLL 时钟源为 HSE，然后把 sys_stm32_clock_init 的形参直接设置作为 PLL 的参数 M 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，然后到我们的步骤 2。

步骤 2：配置系统时钟源，以及 SYSCLK、AHB、APB1 和 APB2 相关参数，用函数 HAL_RCC_ClockConfig()，声明如下：

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

该函数有两个形参，第一个形参 RCC_ClkInitStruct 是结构体 RCC_ClkInitTypeDef 类型指针变量，用于设置 SYSCLK 时钟源以及 SYSCLK、AHB、APB1 和 APB2 的分频系数。第二个形参 FLatency 用于设置 FLASH 延迟。

RCC_ClkInitTypeDef 结构体类型定义比较简单，我们来看看其定义：

```
typedef struct
{
  uint32_t ClockType;             /* 要配置的时钟 */
  uint32_t SYSCLKSource;          /* 系统时钟源 */
  uint32_t AHBCLKDivider;         /* AHB 分频系数 */
  uint32_t APB1CLKDivider;        /* APB1 分频系数 */
  uint32_t APB2CLKDivider;        /* APB2 分频系数 */
}RCC_ClkInitTypeDef;
```

我们在 sys_stm32_clock_init 函数中的实际应用配置内容如下：

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

    /* 设置系统时钟时钟源为 PLL */
```

```
    rcc_clk_init_handle.SYSCLKSource   = RCC_SYSCLKSOURCE_PLLCLK;
    rcc_clk_init_handle.AHBCLKDivider  = RCC_SYSCLK_DIV1;   /* AHB 分频系数为 1 */
    rcc_clk_init_handle.APB1CLKDivider = RCC_HCLK_DIV4;     /* APB1 分频系数为 4 */
    rcc_clk_init_handle.APB2CLKDivider = RCC_HCLK_DIV2;     /* APB2 分频系数为 2 */
```

sys_stm32_clock_init 函数中的 RCC_ClkInitTypeDef 结构体配置内容：

第一个参数 ClockType 配置表示我们要配置的是 SYSCLK、HCLK、PCLK1 和 PCLK 四个时钟。

第二个参数 SYSCLKSource 配置选择系统时钟源为 PLLCLK。

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 mian 函数中调用 sys_stm32_clock_init(336, 8, 2, 7)时设置的形参数值，我们可以计算出，PLL 时钟为 PLLCLK = HSE * 336 / 8 / 2 = 168MHz。

同时我们选择系统时钟源为 PLL，所以系统时钟 SYSCLK=168MHz。AHB 分频系数为 1，所以频率为 HCLK = SYSCLK/1 = 168MHz。APB1 分频系数为 4，故其频率为 PCLK1 = HCLK/4 = 42MHz。APB2 分频系数为 2，故其频率为 PCLK2 = HCLK/2 = 84MHz。我们总结一下通过调用函数 sys_stm32_clock_init(336, 8, 2, 7)之后的关键时钟频率值：

```
SYSCLK(系统时钟)                    =168MHz
PLL 主时钟                          =168MHz
AHB 总线时钟（HCLK=SYSCLK/1）       =168MHz
APB1 总线时钟（PCLK1=HCLK/4）       =42MHz
APB2 总线时钟（PCLK2=HCLK/2）       =84MHz
```

最后我们来看看函数 HAL_RCC_ClockConfig 第二个入口参数 FLatency 的含义，为了使 FLASH 读写正确（因为 168Mhz 的时钟比 Flash 的操作速度 24Mhz 要快得多，操作速度不匹配容易导致 Flash 操作失败），所以需要设置延时时间。对于 STM32F4 系列，FLASH 延迟配置参数值是通过下表 11.2.1.1 来确定的，具体可以参考《STM32F4xx 参考手册_V4（中文版）.pdf》的第三小节：3 嵌入式 FLASH 接口/3.8 FLASH 接口寄存器/FLASH_ACR 寄存器。

表 7. CPU 时钟 (HCLK) 频率对应的等待周期数

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

表 11.2.1.1 CPU 时钟频率对应的等待周期

由于前面我们已经配置好系统时钟为 168M，即 CPU 时钟为 168M，所以根据表 11.2.1.1 需设置等待周期为 5WS，也就是 6 个 CPU 周期，即第二个形参设置值：

```
    ret = HAL_RCC_ClockConfig(&rcc_clk_init_handle, FLASH_LATENCY_5);
```

这样 FLASH 的读写稳定就得到了保障。注：表 11.2.1.1 摘抄于《STM32F4xx 参考手册_V4（中文版）.pdf》第 61 页。

时钟系统配置相关知识就给大家讲解到这里。

11.2.2 STM32F4 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《STM32F4xx 参考手册_V4（中文版）.pdf》6.3 小节查看所有 RCC 相关寄存器的配置。接下来我们来讲解通过 STM32F4 的 HAL 库使能外设时钟的方法。

在 STM32F4 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 STM32F4xx_hal_rcc.h 定义的。大家打开 STM32F4xx_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 = 0x00U; \
                                    SET_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOAEN);\
                          tmpreg = READ_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOAEN);\
                                                          UNUSED(tmpreg); \
                                               } while(0U)
```

这段代码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_Pos          (0U)
#define RCC_AHB1ENR_GPIOAEN_Msk          (0x1UL << RCC_AHB1ENR_GPIOAEN_Pos)
#define RCC_AHB1ENR_GPIOAEN              RCC_AHB1ENR_GPIOAEN_Msk
```

上面三行代码很容易计算出来 RCC_AHB1ENR_GPIOAEN= (0x00000001<<0)，因此上面代码的作用是设置寄存器 RCC->AHB1ENR 寄存器的位 0 为 1。我们可以从 STM32F4 的参考手册中搜索 AHB1ENR 寄存器定义，位 0 的作用是用来设置 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 (IO port A clock enable)
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

图 11.2.2.1 AHB1ENR 寄存器的位 0 描述

那么我们只需要在我们的用户程序中调用宏定义标识符就可以实现 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 STM32F4xx_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 寄存器的位 2 为 0，也就是禁止 GPIOA 时钟。具体使用方法我们这里就不做过多讲解，我们这里同样列出几个常用的禁止外设时钟的宏定义标识符使用方法：

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

关于 STM32F4 的外设时钟使能和禁止方法我们就给大家讲解到这里。

第十二章 SYSTEM 文件夹介绍

SYSTEM 文件夹里面的代码由正点原子提供，是 STM32F4xx 系列的底层核心驱动函数，可以用在 STM32F4xx 系列的各个型号上面，方便大家快速构建自己的工程。本章，我们将向大家介绍这些代码的由来及其功能，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。

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。这 3 个 c 文件提供了系统时钟设置、延时和串口 1 调试功能，任何一款 STM32F4 都具备这几个基本外设，所以可以快速地将这些设置应用到任意一款 STM32F4 产品上，通过这些驱动文件实现快速移植和辅助开发的效果。

本章将分为如下几个小节：

12.1 deley 文件夹代码介绍

12.2 sys 文件夹代码介绍

12.3 usart 文件夹代码介绍

12.1 deley 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(uint32_t ticks);
void SysTick_Handler(void);
void delay_init(uint16_t sysclk);
void delay_us(uint32_t nus);
void delay_ms(uint16_t nms);
```

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 delay 延时的编程思想：CM4 内核处理器，内部包含了一个 SysTick 定时器，SysTick 是一个 24 位的向下递减的计数定时器，当计数值减到 0 时，将从 RELOAD 寄存器中自动重装载定时初值，开始新一轮计数。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。SysTick 在《STM32F4xx 参考手册_V4（中文版）.pdf》里面介绍的很简单，其详细介绍，请参阅《Cortex-M3 权威指南》第 133 页。我们就是利用 STM32 的内部 SysTick 来实现延时的，这样既不占用中断，也不占用系统定时器。

这里我们将介绍的是正点原子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*168（假设系统时钟为 168Mhz，因为 systick 的频率等于系统时钟频率，那么 systick 每增加 1，就是 1/168us），然后我们就一直统计 systick 的计数变化，直到这个值变化了 50*168，一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。

12.1.1 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：

```
/*
 *  当 delay_us/delay_ms 需要支持 OS 的时候需要三个与 OS 相关的宏定义和函数来支持
 *  首先是 3 个宏定义:
 *      delay_osrunning     :用于表示 OS 当前是否正在运行,以决定是否可以使用相关函数
 *      delay_ostickspersec:用于表示 OS 设定的时钟节拍,delay_init
 *                          将根据这个参数来初始化 systick
 *      delay_osintnesting :用于表示 OS 中断嵌套级别,因为中断里面不可以调度,
 *                          delay_ms 使用该参数来决定如何运行
 *  然后是 3 个函数:
 *      delay_osschedlock  :用于锁定 OS 任务调度,禁止调度
 *      delay_osschedunlock:用于解锁 OS 任务调度,重新开启调度
 *      delay_ostimedly     :用于 OS 延时,可以引起任务调度.
 *
 *  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

/**
 * @brief     us 级延时时,关闭任务调度(防止打断 us 级延迟)
 * @param     无
 * @retval    无
 */
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
}

/**
 * @brief     us 级延时时,恢复任务调度
 * @param     无
 * @retval    无
 */
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
}

/**
 * @brief     us 级延时时,恢复任务调度
 * @param     ticks: 延时的节拍数
 * @retval    无
 */
void delay_ostimedly(uint32_t ticks)
{
#ifdef CPU_CFG_CRITICAL_METHOD
    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
}

/**
 * @brief     systick 中断服务函数,使用 OS 时用到
 * @param     ticks: 延时的节拍数
 * @retval    无
 */
void SysTick_Handler(void)
{
    HAL_IncTick();
    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 */
    }
}
#endif
```

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。

12.1.2 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 SYS_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：

```
/**
 * @brief     初始化延迟函数
 * @param     sysclk: 系统时钟频率, 即 CPU 频率(rcc_c_ck), 168MHz
 * @retval    无
 */
void delay_init(uint16_t sysclk)
{
#if SYS_SUPPORT_OS   /* 如果需要支持 OS */
    uint32_t reload;
#endif
    /* SYSTICK 使用外部时钟源,频率为 HCLK */
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
    g_fac_us = sysclk;                    /* 不论是否使用 OS,g_fac_us 都需要使用 */
#if SYS_SUPPORT_OS                        /* 如果需要支持 OS. */
    reload = sysclk;                      /* 每秒钟的计数次数 单位为 M */
   /* 根据 delay_ostickspersec 设定溢出时间,reload 为 24 位
      寄存器,最大值:16777216,在 168M 下,约合 0. 09986s 左右 */
    reload *= 1000000 / delay_ostickspersec;
    g_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/delay_ostickspersec 秒中断一次 */
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;   /* 开启 SYSTICK */
#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 SYS_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 12.1.2.1 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 12.1.2.1 SysTick->CTRL 寄存器各位定义

SysTick-> LOAD 的定义如图 12.1.2.2 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 12.1.2.2 SysTick->LOAD 寄存器各位定义

SysTick->VAL 的定义如图 12.1.2.3 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 12.1.2.3 SysTick->VAL 寄存器各位定义

SysTick->CALIB 不常用，在这里我们也用不到，故不介绍了。

HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);这句代码把 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 8MHz，然后倍频到 168MHz，那么 SysTick 的时钟即为 168MHz，也就是 SysTick 的计数器 VAL 每减 1，就代表时间过了 1/168us。

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）。

12.1.3 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：

```
/**
 * @brief       延时 nus
 * @param       nus: 要延时的 us 数.
 * @note        nus 取值范围 : 0 ~ 190887435us(最大值即 2^32/g_fac_us @g_fac_us = 21)
 * @retval      无
 */
void delay_us(uint32_t nus)
{
    uint32_t ticks;
    uint32_t told, tnow, tcnt = 0;
    uint32_t reload = SysTick->LOAD;    /* LOAD 的值 */
    ticks = nus * g_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; /* 这里注意一下 SYSTICK 是一个递减的计数器就可以了 */
            }
            else
            {
                tcnt += reload - tnow + told;
            }
            told = tnow;
            if (tcnt >= ticks)
            {
                break;              /* 时间超过/等于要延迟的时间,则退出 */
            }
        }
    }
}
```

这里就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。

再来看看使用 OS 的时候，delay_us 的实现函数如下：

```
/**
 * @brief     延时 nus
 * @param     nus: 要延时的 us 数
 * @note      nus 取值范围: 0~190887435us (最大值即 2^32 / g_fac_us @g_fac_us = 21)
 * @retval    无
 */
void delay_us(uint32_t nus)
{
    uint32_t ticks;
    uint32_t told, tnow, tcnt = 0;
    uint32_t reload = SysTick->LOAD;    /* LOAD 的值 */
    ticks = nus * g_fac_us;             /* 需要的节拍数 */
    delay_osschedlock();                /* 阻止 OS 调度，防止打断 us 延时 */
    told = SysTick->VAL;                /* 刚进入时的计数器值 */
    while (1)
    {
        tnow = SysTick->VAL;
        if (tnow != told)
        {
            if (tnow < told)
            {
                tcnt += told - tnow; /* 这里注意一下 SYSTICK 是一个递减的计数器就可以了 */
            }
            else
            {
                tcnt += reload - tnow + told;
            }
            told = tnow;
            if (tcnt >= ticks)
            {
                break;              /* 时间超过/等于要延迟的时间,则退出 */
            }
        }
    }
    delay_osschedunlock();      /* 恢复 OS 调度 */
}
```

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

上面的 delay_osschedlock 和 delay_osschedunlock 是 OS 提供的两个函数，用于调度上锁和解锁，这里为了防止 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！

12.1.4 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：

```
/**
 * @brief       延时 nms
 * @param       nms: 要延时的 ms 数 (0< nms <= 65535)
 * @retval      无
```

```
 */
void delay_ms(uint16_t nms)
{
    uint32_t repeat = nms / 30;   /*  这里用 30,是考虑到可能有超频应用*/
    uint32_t remain = nms % 30;

    while (repeat)
    {
        delay_us(30 * 1000);      /* 利用 delay_us 实现 1000ms 延时 */
        repeat--;
    }

    if (remain)
    {
        delay_us(remain * 1000);  /* 利用 delay_us, 把尾数延时(remain ms)给做了 */
    }
}
```

该函数其实就是多次调用 delay_us 函数，来实现毫秒级延时的。我们做了一些处理，使得调用 delay_us 函数的次数减少，这样时间会更加精准。再来看看使用 OS 的时候，delay_ms 的实现函数如下：

```
/**
 * @brief       延时 nms
 * @param       nms: 要延时的 ms 数 (0< nms <= 65535)
 * @retval      无
 */
void delay_ms(uint16_t nms)
{
/* 如果 OS 已经在跑了,并且不是在中断里面(中断里面不能任务调度) */
    if (delay_osrunning && delay_osintnesting == 0)
    {
        if (nms >= g_fac_ms)                    /* 延时的时间大于 OS 的最少时间周期 */
        {
            delay_ostimedly(nms / g_fac_ms);    /* OS 延时 */
        }
        nms %= g_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((uint32_t)(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_ostickspersec=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 53 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（g_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）。

12.1.5 HAL 库延时函数 HAL_Delay

前面我们在 7.4.2 章节介绍 stm32f4xx_hal.c 文件时，已经讲解过 Systick 实现延时相关函数。实际上，HAL 库提供的延时函数，只能实现简单的毫秒级别延时，没有实现 us 级别延时。我看看 HAL 库的 HAL_Delay 函数原定义：

```
/* HAL 库的延时函数，默认延时单位 ms */
__weak void HAL_Delay(uint32_t Delay)
```

```
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;

  /* Add a freq to guarantee minimum wait */
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(uwTickFreq);
  }

  while ((HAL_GetTick() - tickstart) < wait)
  {
  }
}
```

HAL 库实现延时功能非常简单，首先定义了一个 32 位全局变量 uwTick，在 Systick 中断服务函数 SysTick_Handler 中通过调用 HAL_IncTick 实现 uwTick 值不断增加，也就是每隔 1ms 增加 uwTickFreq，而 uwTickFreq 默认是 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(虽然一般禁止在中断中使用延时函数)，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。

HAL 库的 ms 级别的延时函数__weak void HAL_Delay(uint32_t Delay)；它是弱定义函数，所以用户可以自己重新定义该函数。例如：我们在 deley.c 文件可以这样重新定义该函数：

```
/**
  * @brief  HAL 库延时函数重定义
  * @param Delay 要延时的毫秒数
  * @retval None
  */
void HAL_Delay(uint32_t Delay)
{
    delay_ms(Delay);
}
```

12.2 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，主要实现下面的几个函数，以及一些汇编函数。

```
void sys_nvic_set_vector_table(uint32_t baseaddr, uint32_t offset);
                                                        /* 设置中断偏移量 */
void sys_standby(void);             /* 进入待机模式 */
void sys_soft_reset(void);          /* 系统软复位 */
uint8_t sys_stm32_clock_init(uint32_t plln, uint32_t pllm, uint32_t pllp,
uint32_t pllq);                                         /* 配置系统时钟 */

/* 以下为汇编函数 */
void sys_wfi_set(void);             /* 执行 WFI 指令 */
void sys_intx_disable(void);        /* 关闭所有中断 */
void sys_intx_enable(void);         /* 开启所有中断 */
void sys_msr_msp(uint32_t addr);    /* 设置栈顶地址 */
```

下面讲一下函数的功能，sys_nvic_set_vector_table 函数主要是设置中断向量表偏移地址，sys_standby 函数用于进入待机模式，sys_soft_reset 函数用于系统软复位，sys_stm32_clock_init 函数是系统时钟初始化函数，在 11.2.1 小节 STM32F4 时钟系统配置章节已经有说明了，大家可以复习这部分知识点。

在 sys.h 文件中只是对于 sys.c 的函数进行声明。

12.3 usart 文件夹代码介绍

该文件夹下面有 usart.c 和 usart.h 两个文件。在我们的工程使用串口 1 和串口调试助手来实现调试功能，可以把单片机的信息通过串口助手显示到电脑屏幕。串口相关知识，**我们将在第十七章讲解串口实验的时候给大家详细讲解。**本节只给大家讲解 printf 函数支持相关的知识。

标准库下的 printf 为调试属性的函数，如果直接使用，会使单片机进入半主机模式（semihosting），这是一种调试模式，直接下载代码后出现程序无法运行，但是在连接调试器进行 Debug 时程序反而能正常工作的情况。半主机是 ARM 目标的一种机制，用于将输入/输出请求从应用程序代码通信到运行调试器的主机。例如，此机制可用于允许 C 库中的函数（如 printf()和 scanf()）使用主机的屏幕和键盘，而不是在目标系统上设置屏幕和键盘。这很有用，因为开发硬件通常不具有最终系统的所有输入和输出设备，如屏幕、键盘等。半主机是通过一组定义好的软件指令（如 SVC）SVC 指令（以前称为 SWI 指令）来实现的，这些指令通过程序控制生成异常。应用程序调用相应的半主机调用，然后调试代理处理该异常。调试代理（这里的调试代理是仿真器）提供与主机之间的必需通信。也就是说使用半主机模式必须使用仿真器调试。

如果想在独立环境下运行调试功能的函数，我们这里是 printf，printf 对字符 ch 处理后写入文件 f，最后使用 fputc 将文件 f 输出到显示设备。对于 PC 端的设备，fputc 通过复杂的源码，最终把字符显示到屏幕上。那我们需要做的，就是把 printf 调用的 fputc 函数重新实现，重定向 fputc 的输出，同时避免进入半主模式。

要避免半主机模式，现在主要有两种方式：一是使用 MicroLib，即微库；另一种方法是确保 ARM 应用程序中没有链接 MicroLib 的半主机相关函数，我们要取消 ARM 的半主机工作模式，这可以通过代码实现。

先说微库，ARM 的 C 微库 MicroLib 是为嵌入式设备开发的一套类似于标准 C 接口函数的精简代码库，用于替代默认 C 库，是专门针对专业嵌入式应用开发而设计的，特别适合那些对存储空间有特别要求的嵌入式应用程序，这些程序一般不在操作系统下运行。使用微库编写程序要注意其与默认 C 库之间存在的一些差异，如 main()函数不能声明带参数，也无须返回；不支持 stdio，除了无缓冲的 stdin、stdout 和 syderr；微库不支持操作系统函数；微库不支持可选的单或两区存储模式；微库只提供分离的堆和栈两区存储模式等等，它裁减了很多函数，而且还有很多东西不支持。如果原来用标准库可以跑，选择 MicroLib 后却突然不行了，是很常见的。与标准的 C 库不一样，微库重新实现了 printf，使用微库的情况下就不会进入半主机模式了。Keil 下使用微库的方法很简单，在“Target”下勾选“Use MicroLib”即可。

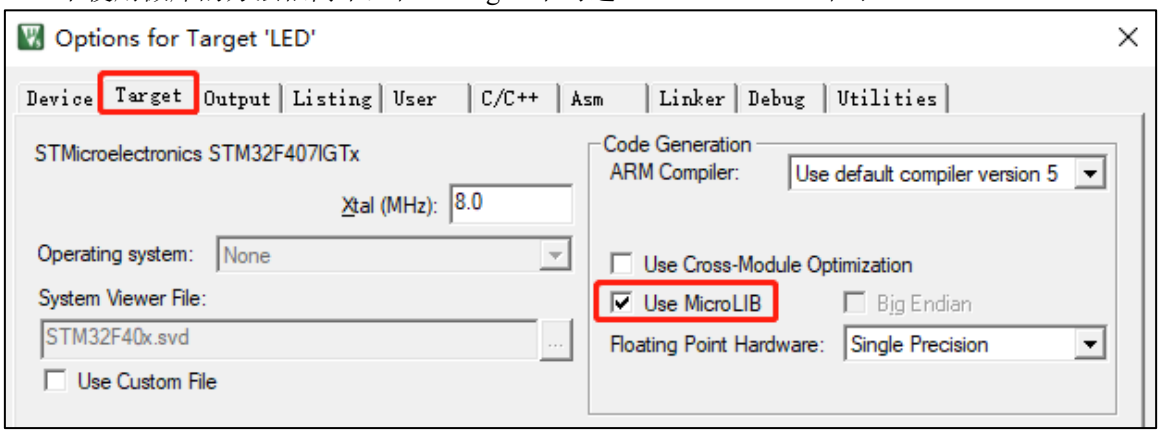


图 12.3.1.1 MDK 工程下使用微库的方法

在 keil5 中，不管是否使用半主机模式，使用 printf、scanf、fopen、fread 等都需要自己填充底层函数，以 printf 为例，需要补充定义 fputc，启用微库后，在我们初始化和使能串口 1 之后，我们只需要重新实现 fputc 的功能即可将每个传给 fputc 函数的字符 ch 重定向到串口 1，如果这时接上串口调试助手的话，可以看到串口的数据。实现的代码如下：

```
/* 重定义 fputc 函数, printf 函数最终会通过调用 fputc 输出字符串到串口 */
int fputc(int ch, FILE *f)
{
    while ((USART_UX->SR & 0X40) == 0);     /* 等待上一个字符发送完成 */
```

```
    USART_UX->DR = (uint8_t)ch;              /* 将要发送的字符 ch 写入到 DR 寄存器 */
    return ch;
}
```

上面说到了微库的一些限制，使用时注意某些函数与标准库的区别就不会影响到我们代码的正常功能。如果不想使用微库，那就要用到我们提到的第二种方法：取消 ARM 的半主机工作模式；只需在代码中添加不使用半主机的声明即可，对于 AC5 和 AC6 编译器版本，声明半主机的语法不同，为了同时兼容这两种语法，我们在利用编译器自带的宏__ARMCC_VERSION 判定编译器版本，并根据版本不同选择不同的语法声明不使用半主机模式，具体代码如下：

```
#if (__ARMCC_VERSION >= 6010050)            /* 使用 AC6 编译器时 */
__asm(".global __use_no_semihosting\n\t"); /* 声明不使用半主机模式 */
__asm(".global __ARM_use_no_argv \n\t");   /* AC6 下需要声明 main 函数为无参数格式，否
                                              则部分例程可能出现半主机模式 */

#else
/* 使用 AC5 编译器时，要在这里定义__FILE 和 不使用半主机模式 */
#pragma import(__use_no_semihosting)

/* 解决 HAL 库使用时，某些情况可能报错的 bug */
struct __FILE
{
    int handle;
    /* Whatever you require here. If the only file you are using is */
    /* standard output using printf() for debugging, no file handling */
    /* is required. */
};

#endif
```

使用的上面的代码，Keil 的编译器就不会把标准库的这部分函数链接到我们的代码里。如果用到原来半主机模式下的调试函数，需要重新实现它的一些依赖函数接口，对于 printf 函数需要实现的接口，我们的代码中将它们实现如下：

```
/* 不使用半主机模式，至少需要重定义_ttywrch\_sys_exit\_sys_command_string 函数,以同时兼
容 AC6 和 AC5 模式 */
int _ttywrch(int ch)
{
    ch = ch;
    return ch;
}
/* 定义_sys_exit()以避免使用半主机模式 */
void _sys_exit(int x)
{
    x = x;
}
char *_sys_command_string(char *cmd, int len)
{
    return NULL;
}
/* FILE 在 stdio.h 里面定义. */
FILE __stdout;
```

fputc 的重定向和之前一样，重定向到串口 1 即可，如果硬件资源允许，读者有特殊需求，也可以重定向到 LCD 或者其它串口。

```
/* 重定义 fputc 函数, printf 函数最终会通过调用 fputc 输出字符串到串口 */
int fputc(int ch, FILE *f)
{
    while ((USART1->SR & 0X40) == 0);    /* 等待上一个字符发送完成 */
    USART_UX->DR = (uint8_t)ch;          /* 将要发送的字符 ch 写入到 DR 寄存器 */
    return ch;
}
```

第二篇 入门篇

功夫不负有心人，相信学习至此你已经掌握了基础篇介绍的知识。我们希望通过前面的章节你已经掌握了 STM32 开发的工具和方法。本篇我们将和大家一起来学习 STM32 的一些基础外设，这些外设实际项目中经常会用到，希望大家认真学习和掌握，以便将来更好、更快的完成实际项目开发。

本篇将采取一章一实例的方式，介绍 STM32 常用外设的使用，通过本篇的学习，我们将带领大家进入 STM32 的精彩世界。

第十三章 跑马灯实验

本章将通过一个经典的跑马灯程序，带大家开启 STM32F407 之旅。通过本章的学习，我们将了解到 STM32F407 的 IO 口作为输出使用的方法。我们将通过代码控制开发板上的 LED 灯：LED0、LED1 交替闪烁，实现类似跑马灯的效果。

本章分为如下 4 个小节：

13.1 STM32F4 GPIO 简介
13.2 硬件设计
13.3 程序设计
13.4 下载验证

13.1 STM32F407 GPIO 简介

GPIO 是控制或者采集外部器件的信息的外设，即负责输入输出。它是按组分配存在，每组最多有 16 个 IO 引脚，组数视芯片而定。比如：STM32F407IGT6 是 176 脚的芯片，共有 140 个 IO 引脚可供我们编程使用，分为 10 个组，分别是：GPIOA、GPIOB、GPIOC、GPIOD、GPIOE、GPIOF、GPIOG、GPIOH 和 GPIOI，其中 GPIOI 组只有 12 个 IO（从 PI0 到 PI11），其它组都是 16 个 IO 引脚。这里重点说一下 STM32F407 的 IO 电平兼容性问题，STM32F407 的绝大部分 IO 口，都兼容 5V，至于到底哪些是兼容 5V 的，请看 STM32F407IGT6 的数据手册（注意是数据手册，不是中文参考手册），见表 5 大容量 STM32F40xxx 引脚定义，凡是有 FT 标志的，都是兼容 5V 电平的 IO 口，可以直接接 5V 的外设（注意：如果引脚设置的是模拟输入模式，则不能接 5V！），凡是不带 FT 标志的，就建议大家不要接 5V 了，可能烧坏 MCU。

Name	Abbreviation	Definition
Pin name	Unless otherwise specified in brackets below the pin name, the pin function during and after reset is the same as the actual pin name	
Pin type	S	Supply pin
	I	Input only pin
	I/O	Input / output pin
I/O structure	FT	5 V tolerant I/O
	TTa	3.3 V tolerant I/O directly connected to ADC
	B	Dedicated BOOT0 pin
	RST	Bidirectional reset pin with embedded weak pull-up resistor
Notes	Unless otherwise specified by a note, all I/Os are set as floating inputs during and after reset	
Alternate functions	Functions selected through GPIOx_AFR registers	
Additional functions	Functions directly selected/enabled through peripheral registers	

表 13.1.1 IO 口属性缩写符号表

13.1.1 GPIO 功能模式

GPIO 有八种工作模式，分别是：

1、输入浮空
2、输入上拉
3、输入下拉
4、模拟

5、开漏输出
6、推挽输出
7、开漏式复用功能
8、推挽式复用功能

13.1.2 GPIO 基本结构分析

我们知道了 GPIO 有八种工作模式，具体这些模式是怎么实现的？下面我们通过 GPIO 的基本结构图来分别进行详细分析，先看看总的框图，如图 13.1.2.1 所示。

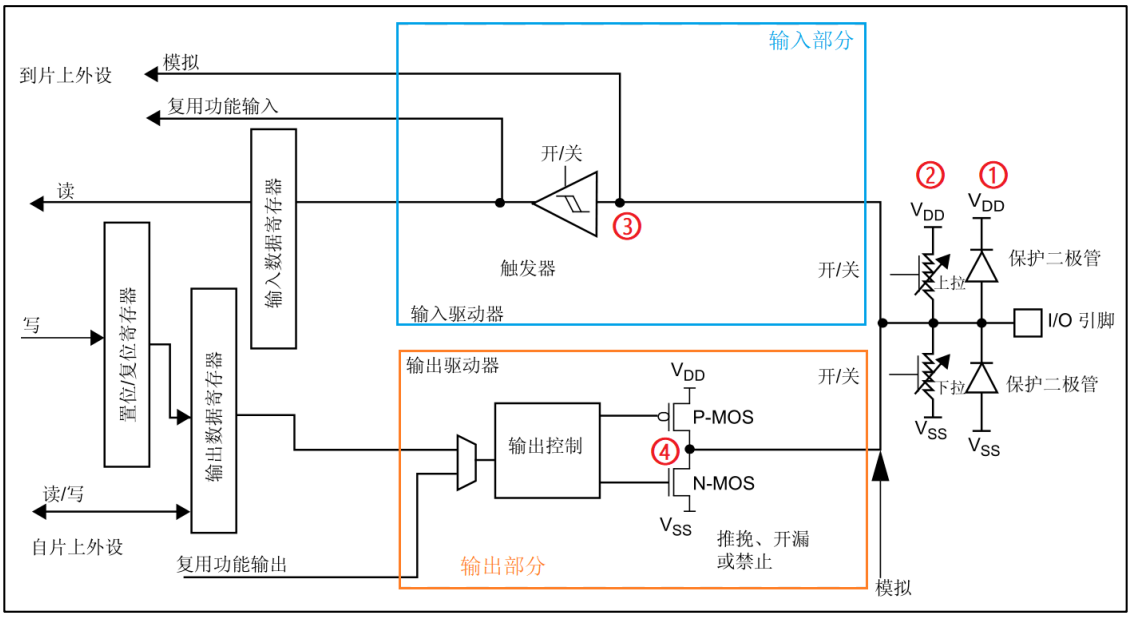


图 13.1.2.1 GPIO 的基本结构图

如上图所示，可以看到右边只有 I/O 引脚，这个 I/O 引脚就是我们可以看到的芯片实物的引脚，其他部分都是 GPIO 的内部结构。

① 保护二极管

保护二极管共有两个，用于保护引脚外部过高或过低的电压输入。当引脚输入电压高于 VDD 时，上面的二极管导通，当引脚输入电压低于 VSS 时，下面的二极管导通，从而使输入芯片内部的电压处于比较稳定的值。虽然有二极管的保护，但这样的保护却很有限，大电压大电流的接入很容易烧坏芯片。所以在实际的设计中我们要考虑设计引脚的保护电路。

③ 上拉、下拉电阻

它们阻值大概在 30~50K 欧姆之间，可以通过上、下两个对应的开关控制，这两个开关由寄存器控制。当引脚外部的器件没有干扰引脚的电压时，即没有外部的上、下拉电压，引脚的电平由引脚内部上、下拉决定，开启内部上拉电阻工作，引脚电平为高，开启内部下拉电阻工作，则引脚电平为低。同样，如果内部上、下拉电阻都不开启，这种情况就是我们所说的浮空模式。浮空模式下，引脚的电平是不可确定的。引脚的电平可以由外部的上、下拉电平决定。需要注意的是，STM32 的内部上拉是一种“弱上拉”，这样的上拉电流很弱，如果有要求大电流还是得外部上拉。

③ 施密特触发器

对于标准施密特触发器，当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；当输入在正负向阈值电压之间，输出不改变，也就是说输出由高电准位翻转为低电准位，或是由低电准位翻转为高电准位对应的阈值电压是不同的。只有当输入电压发生足够的变化时，输出才会变化，因此将这种元件命名为触发器。这种双阈值动作被称为迟滞现象，表明施密特触发器有记忆性。从本质上来说，施密特触发器是一种双稳态多谐振荡器。

施密特触发器可作为波形整形电路，能将模拟信号波形整形为数字电路能够处理的方波波形，而且由于施密特触发器具有滞回特性，所以可用于抗干扰，其应用包括在开回路配置中用

于抗扰，以及在闭回路正回授/负回授配置中用于实现多谐振荡器。

下面看看比较器跟施密特触发器的作用的比较，就清楚的知道施密特触发器对外部输入信号具有一定抗干扰能力，如图 13.1.2.2 所示。

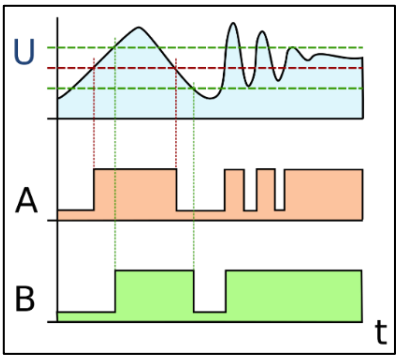


图 13.1.2.2 比较器的（A）和施密特触发器（B）作用比较

④ P-MOS 管和 N-MOS 管

这个结构控制 GPIO 的开漏输出和推挽输出两种模式。开漏输出：输出端相当于三极管的集电极，要得到高电平状态需要上拉电阻才行。推挽输出：这两只对称的 MOS 管每次只有一只导通，所以导通损耗小、效率高。输出既可以向负载灌电流，也可以从负载拉电流。推拉式输出既能提高电路的负载能力，又能提高开关速度。

上面我们对 GPIO 的基本结构图中的关键器件做了介绍，下面分别介绍 GPIO 八种工作模式对应结构图的工作情况。

- **1、输入浮空**

输入浮空模式：上拉/下拉电阻为断开状态，施密特触发器打开，输出被禁止。输入浮空模式下，IO 口的电平完全是由外部电路决定。如果 IO 引脚没有连接其他的设备，那么检测其输入电平是不确定的。该模式可以用于按键检测等场景。

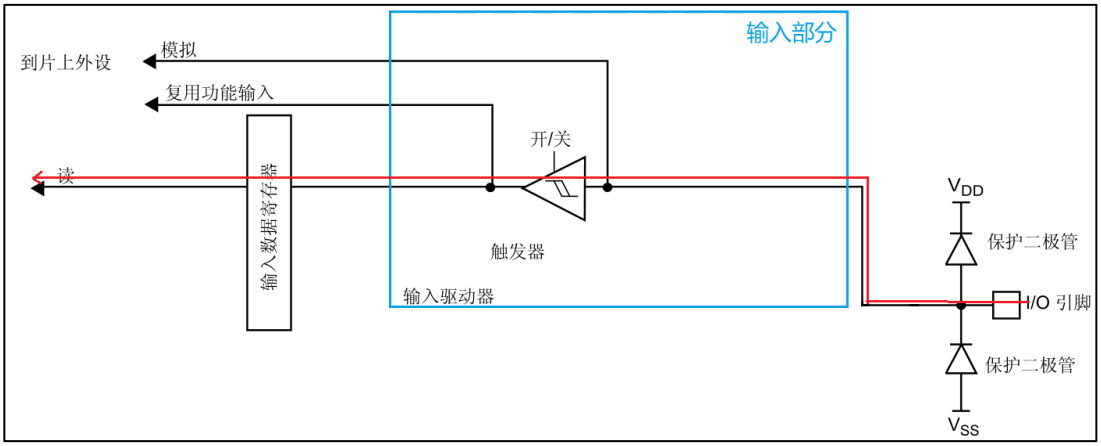


图 13.1.2.3 输入浮空模式

- **2、输入上拉**

输入上拉模式：上拉电阻导通，施密特触发器打开，输出被禁止。在需要外部上拉电阻的时候，可以使用内部上拉电阻，这样可以节省一个外部电阻，但是内部上拉电阻的阻值较大，所以只是“弱上拉”，不适合做电流型驱动。

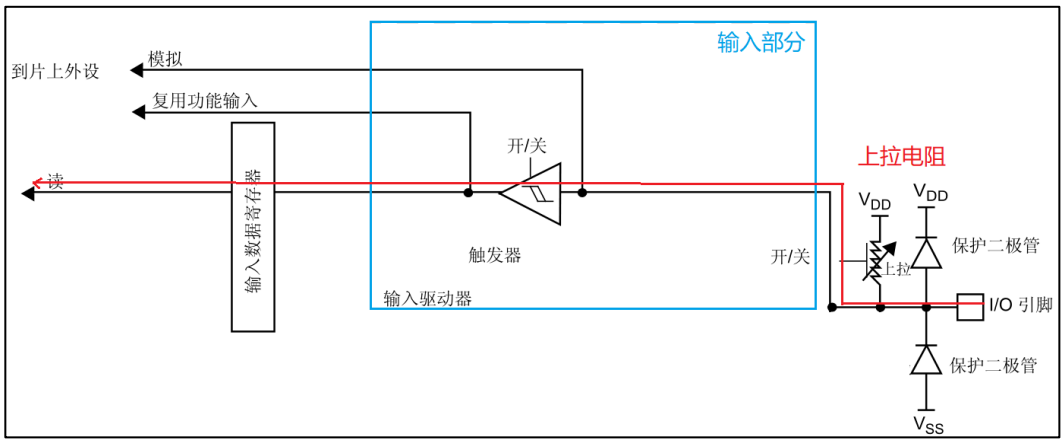


图 13.1.2.4 输入上拉模式

- **3、输入下拉**

输入下拉模式：下拉电阻导通，施密特触发器打开，输出被禁止。在需要外部下拉电阻的时候，可以使用内部下拉电阻，这样可以节省一个外部电阻，但是内部下拉电阻的阻值较大，所以不适合做电流型驱动。

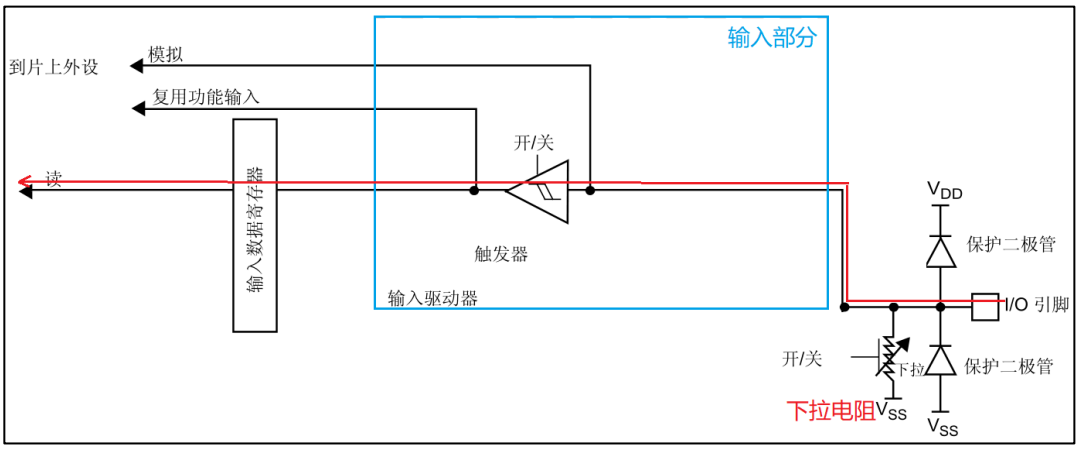


图 13.1.2.5 输入下拉模式

- **4、模拟功能**

模拟功能：上下拉电阻断开，施密特触发器关闭，双 MOS 管也关闭。该模式用于 ADC 采集或者 DAC 输出，或者低功耗下省电。

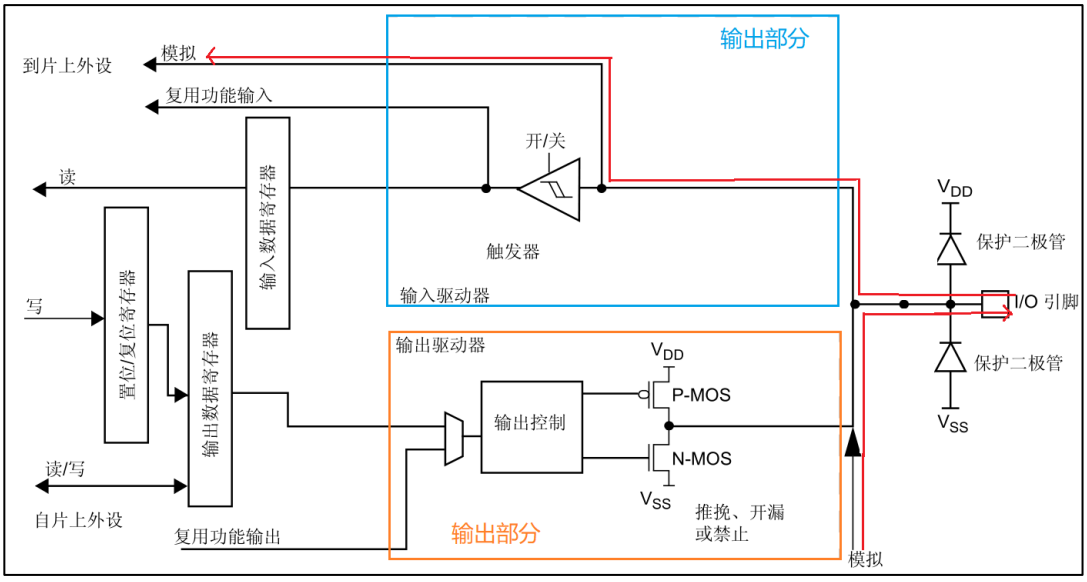


图 13.1.2.6 模拟功能

5、开漏输出

开漏输出模式：STM32 的开漏输出模式是数字电路输出的一种，从结果上看它只能输出低电平 Vss 或者高阻态。根据《STM32F4xx 参考手册_V4（中文版）.pdf》第 183 页关于“GPIO 输出配置”的描述，我们可以推知开漏模式下输出电路大致是这样工作的：

- P-MOS 被“输出控制”控制在截止状态，因此 IO 的状态取决于 N-MOS 的导通状况。
- 只有 N-MOS 还受控制于输出寄存器，“输出控制器”对输入信号进行了逻辑非的操作。
- IO 到输入电路的采样电路仍被打开，且可以选择是否使用上下拉电阻。

根据参考手册的描述，我们替换了“输出控制”部分，作出了如图 13.1.2.7 的开漏模式下的简化等效图，图 13.1.2.7 中①的输入对应的②的输出是我们最关心的开漏输出的结果。简化后的图 13.1.2.7 能更好地表示开漏输出模式的输出关系。

开漏输出的具体的理解描述如下：

开漏模式下，P-MOS 管是一直截止的，所以 P-MOS 管的栅极一直接 V_{SS}。如果输出数据寄存器设置为 0 时，经过“输出控制”的逻辑非操作后，输出逻辑 1 到 N-MOS 管的栅极，这时 N-MOS 管就会导通，使得 I/O 引脚接到 V_{SS}，即输出低电平。

如果输出数据寄存器设置为 1 时，经过“输出控制器”的逻辑非操作后，输出逻辑 0 到 N-MOS 管的栅极，这时 N-MOS 管就会截止。因为 P-MOS 管是一直截止的，使得 I/O 引脚呈现高阻态，即不输出低电平，也不输出高电平。因此要 I/O 引脚输出高电平就必须接上拉电阻。这时可以接内部上拉电阻，或者接一个外部上拉电阻。由于内部上拉电阻的阻值较大，所以只是“弱上拉”。需要大电流驱动，请接外部的上拉电阻。此外，上拉电阻具有线与特性，即如果有很多开漏模式的引脚连在一起的时候，只有当所有引脚都输出高阻态，电平才为 1，只要有其中一个为低电平时，就等于接地，使得整条线路都为低电平 0。我们的 IIC 通信（IIC_SDA）就用到这个原理。

另外在开漏输出模式下，施密特触发器是打开的，所以 IO 口引脚的电平状态会被采集到输入数据寄存器中，如果对输入数据寄存器进行读访问可以得到 IO 口的状态。也就是说开漏输出模式下，我们可以对 IO 口进行读数据。

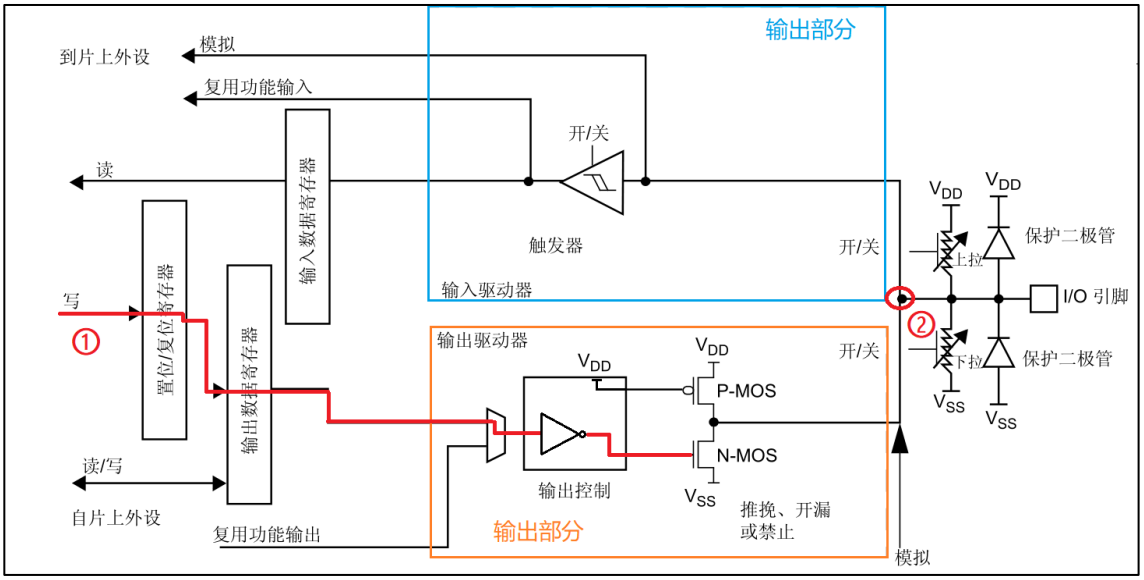


图 13.1.2.7 开漏输出模式

6、推挽输出

推挽输出模式：STM32 的推挽输出模式，从结果上看它会输出低电平 VSS 或者高电平 VDD。推挽输出跟开漏输出不同的是，推挽输出模式 P-MOS 管和 N-MOS 管都用上。同样地，我们根据参考手册推挽模式下的输出描述，列出等效原理如图 13.1.2.8 所示，根据手册描述可以把“输出控制”简单地等效为一个非门。

推挽输出的具体的理解描述如下：

如果输出数据寄存器设置为 0 时，经过“输出控制”的逻辑非操作后，输出逻辑 1 到 P-MOS 管的栅极，这时 P-MOS 管就会截止，同时也会输出逻辑 1 到 N-MOS 管的栅极，这时 N-MOS

管就会导通，使得 I/O 引脚接到 V_{SS}，即输出低电平。

如果输出数据寄存器设置为 1 时，经过“输出控制”的逻辑非操作后，输出逻辑 0 到 N-MOS 管的栅极，这时 N-MOS 管就会截止，同时也会输出逻辑 0 到 P-MOS 管的栅极，这时 P-MOS 管就会导通，使得 I/O 引脚接到 V_{DD}，即输出高电平。

上面的描述可以知道，推挽输出模式下，P-MOS 管和 N-MOS 管同一时间只能有一个 MOS 管是导通的。当引脚高低电平切换时，两个管子轮流导通，一个负责灌电流，一个负责拉电流，使其负载能力和开关速度都有很大的提高。

另外在推挽输出模式下，施密特触发器也是打开的，我们可以读取 IO 口的电平状态。

由于推挽输出模式输出高电平时，是直接连接 VDD ，所以驱动能力较强，可以做电流型驱动，驱动电流最大可达 25mA。该模式也是最常用的输出模式。

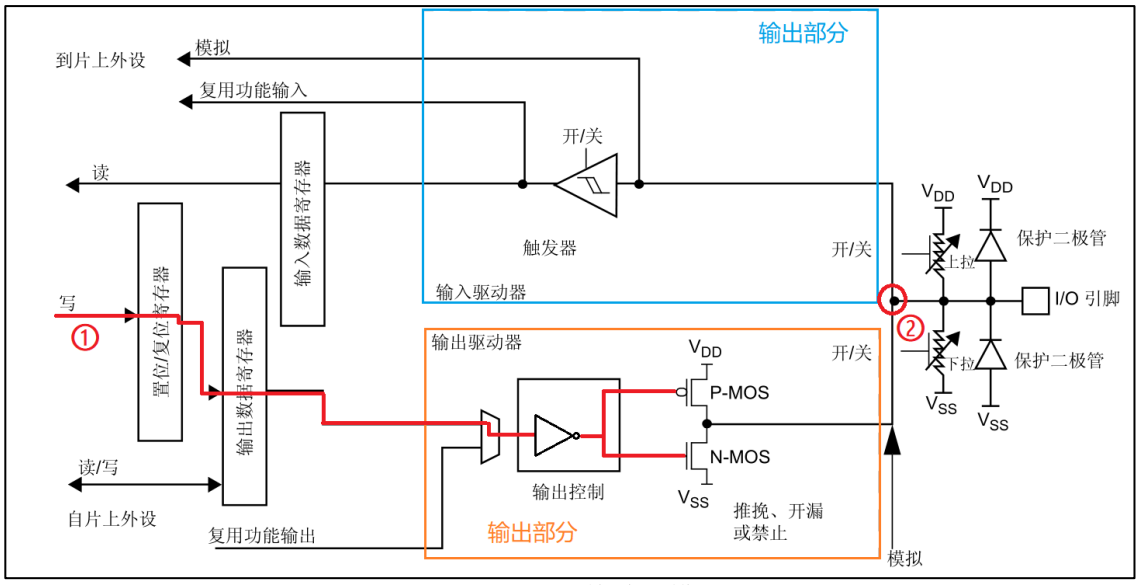


图 13.1.2.8 推挽输出模式

● 7、开漏式复用功能

开漏式复用功能：一个 IO 口可以是通用的 IO 口功能，还可以是其他外设的特殊功能引脚，这就是 IO 口的复用功能。一个 IO 口可以是多个外设的功能引脚，我们需要选择作为其中一个外设的功能引脚。当选择复用功能时，引脚的状态是由对应的外设控制，而不是输出数据寄存器。除了复用功能外，其他的结构分析请参考开漏输出模式。

另外在开漏式复用功能模式下，施密特触发器也是打开的，我们可以读取 IO 口的电平状态，同时外设可以读取 IO 口的信息。

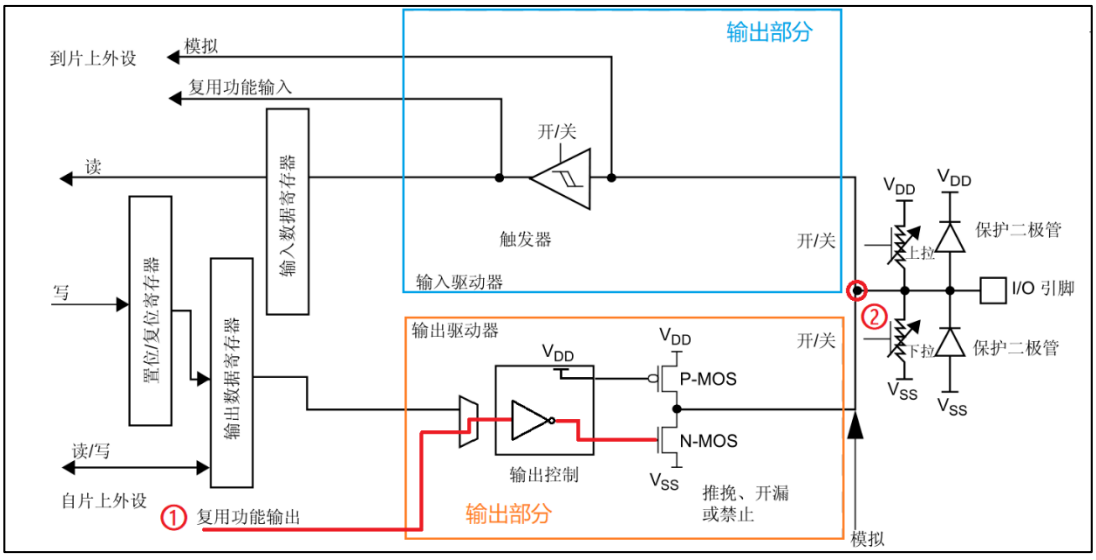


图 13.1.2.9 开漏式复用功能

- **8、推挽式复用功能**

推挽式复用功能：复用功能介绍请查看开漏式复用功能，结构分析请参考推挽输出模式，这里不再赘述。

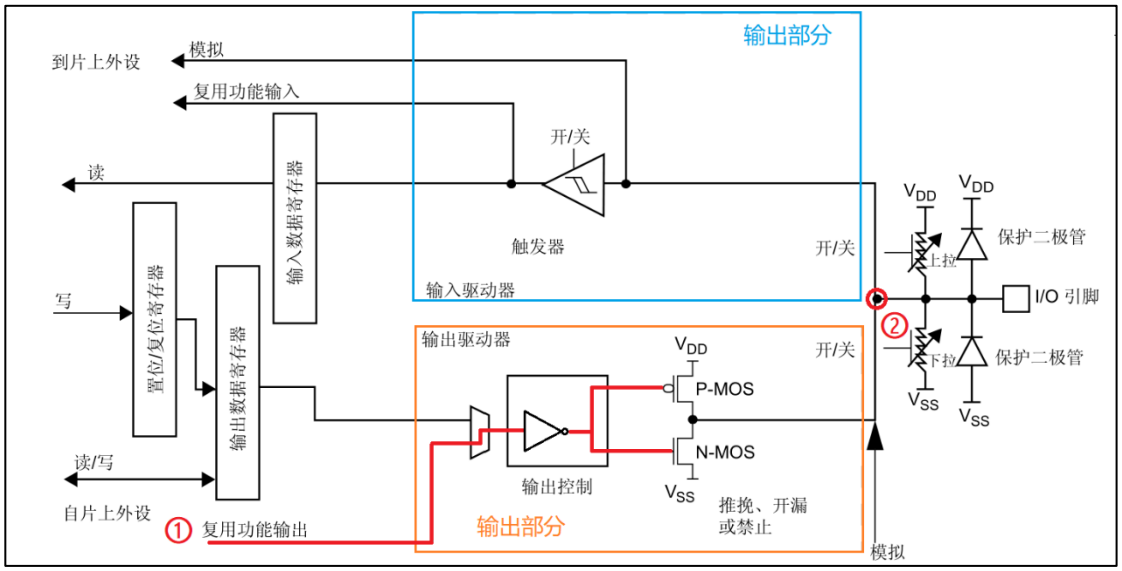


图 13.1.2.10 推挽式复用功能

13.1.3 GPIO 寄存器介绍

STM32F4 每组（这里是 A~I）通用 GPIO 口有 7 个 32 位寄存器控制，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)

2 个 32 位复用功能选择寄存器（AFRH 和 AFRL）

下面我们将带大家理解本章用到的寄存器，没有介绍到的寄存器后面用到会继续介绍。这里主要是带大家学会怎么理解这些寄存器的方法，其他寄存器理解方法是一样的。因为寄存器太多不可能一个个列出来讲，以后基本就是只会把重要的寄存器拿出来讲述，希望大家尽快培养自己学会看手册的能力。下面先看 GPIO 的 4 个 32 位配置寄存器：

- **GPIO 端口模式寄存器 (GPIOx_MODER) (x =A..I)**

该寄存器是 GPIO 口模式控制寄存器，用于控制 GPIOx（STM32F4 最多有 9 组 IO，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，寄存器描述如图 13.1.3.1 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	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

位 2y:2y+1 **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

图 13.1.3.1 MODER 寄存器描述

每组 GPIO 下有 16 个 IO 口，该寄存器共 32 位，每 2 个位控制 1 个 IO。我们看看这个寄

存器的复位值，然后用复位值举例说明一下这样的配置值代表什么意思。比如 GPIOA 的复位值是 0xABFF FFFF，低 16 位都是 1，也就是 PA0~PA7 默认都是模拟模式。高 16 位的值是 0xABFF，也就是 PA8~PA12 默认是模拟模式，PA13\PA14\PA15 则默认是复用功能模式。而 GPIOB 的复位值是 0xFFFF FEBF，只有 PB3 默认是复用功能模式，其他默认都是模拟模式。这四个默认是复用功能模式的 IO 口都是 JTAG 功能对应的 IO 口。

- **GPIO 端口输出类型寄存器 (GPIOx_OTYPER) (x = A..I)**

该寄存器用于控制 GPIOx 的输出类型，寄存器描述如图 13.1.3.2 所示。

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

位 31:16 保留，必须保持复位值。

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

图 13.1.3.2 GPIOx OTYPER 寄存器描述

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

- **GPIO 端口输出速度寄存器 (GPIOx_OSPEEDR) (x = A..I)**

该寄存器用于控制 GPIOx 的输出速度，寄存器描述如图 13.1.3.3 所示。

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

位 2y:2y+1 **OSPEEDRy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 输出速度。
00：2 MHz（低速）
01：25 MHz（中速）
10：50 MHz（快速）
11：30 pF 时为 100 MHz（高速）（15 pF 时为 80 MHz 输出（最大速度））

图 13.1.3.3 GPIOx OSPEEDR 寄存器描述

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，每两个位控制一个 IO 口。

- **GPIO 端口上拉/下拉寄存器 (GPIOx_PUPDR) (x = A..I)**

该寄存器用于控制 GPIOx 的上拉/下拉，寄存器描述如图 13.1.3.4 所示。

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

位 2y+1:2y **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　　10：下拉
01：上拉　　11：保留

图 13.1.3.4 GPIOx PUPDR 寄存器描述

该寄存器每两个位控制一个 IO 口，用于设置上下拉，复位后，该寄存器值一般为 0，即无上拉或下拉。

上面这 4 个配置寄存器就是用来配置 GPIO 的相关模式和状态，它们通过不同的配置组合方法，就决定我们所说的 8 种工作模式。下面，我们来列表阐述，如表 13.1.3.1 所示。

GPIO 工作模式	模式寄存器 MODER[0:1]	输出类型寄存器 OTYPER	输出速度寄存器 OSPEEDR[0:1]	上拉/下拉寄存器 PUPDR[0:1]
输入浮空	00-输入模式	无效	无效	00-无上拉或下拉
输入上拉				01-上拉
输入下拉				10-下拉
模拟功能	11-模拟模式			00-无上拉或下拉
开漏输出	01-通用输出	1-开漏输出	00-低速 01-中速 10-高速 11-超高速	00-无上拉或下拉 01-上拉 10-下拉 11-保留
推挽输出		0-推挽输出		
开漏式复用功能	10-复用功能	1-开漏输出		
推挽式复用功能		0-推挽输出		

表 13.1.3.1 4 个配置寄存器组合下的 8 种工作模式

因为本章需要 GPIO 作为输出口使用，所以我们再来看看端口输出数据寄存器。

- **端口输入数据寄存器（IDR）**

该寄存器用于获取 GPIOx 的输入高低电平，寄存器描述如图 13.1.3.5 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
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。

图 13.1.3.5 GPIOxIDR 寄存器描述

该寄存器为只读寄存器，并且只能以 16 位的形式读出。读出的值为对应 IO 口的状态。

- **端口输出数据寄存器（ODR）**

该寄存器用于控制 GPIOx 的输出高电平或者低电平，寄存器描述如图 13.1.3.6 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。
注意：对于原子置位/复位，通过写入 GPIOx_BSRR 寄存器，可分别对 ODR 位进行置位和复位 (x = A..I)。

图 13.1.3.6 GPIOx ODR 寄存器描述

该寄存器低 16 位有效，分别对应每一组 GPIO 的 16 个引脚。当 CPU 写访问该寄存器，如果对应的某位写 0(ODRy=0)，则表示设置该 IO 口输出的是低电平，如果写 1(ODRy=1)，则表示设置该 IO 口输出的是高电平，y=0~15。

除了 ODR 寄存器，还有一个寄存器也是用于控制 GPIO 输出的，它就是 BSRR 寄存器。

- **端口置位/复位寄存器（BSRR）**

该寄存器也用于控制 GPIOx 的输出高电平或者低电平，寄存器描述如图 13.1.3.7 所示。

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。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：复位相应的 ODRx 位
注： 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。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：置位相应的 ODRx 位

图 13.1.3.7 GPIOx BSRR 寄存器描述

为什么有了 ODR 寄存器，还要这个 BDRR 寄存器呢？我们先看看 BSRR 的寄存器描述，首先 BSRR 是只写权限，而 ODR 是可读可写权限。BSRR 寄存器 32 位有效，对于低 16 位（0-15），我们往相应的位写 1(BSy=1)，那么对应的 IO 口会输出高电平，往相应的位写 0(BSy=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(BRy=1)会输出低电平，写 0(BRy=0)没有任何影响，y=0~15。

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 寄存器还有一个好处，就是 BSRR 寄存器改变引脚状态的时候，不会被中断打断，而 ODR 寄存器有被中断打断的风险。

13.2 硬件设计

1. 例程功能

LED 灯：LED0 和 LED1 每过 500ms 一次交替闪烁，实现类似跑马灯的效果。

2. 硬件资源

1）LED 灯
LED0 - PE0
LED1 - PE1

3. 原理图

本章用到的硬件用到板载的 LED0 和 LED1。电路在开发板上已经连接好了，所以在硬件上不需要动任何东西，直接下载代码就可以测试使用。其连接原理图如图 13.2.1 所示：

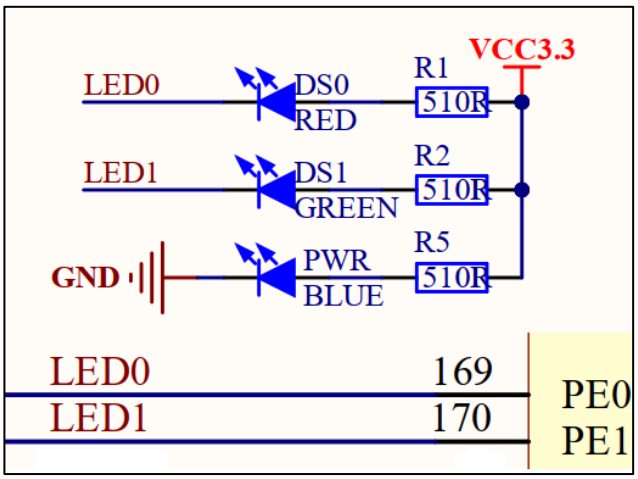


图 13.2.1 LED 与 STM32F4 连接原理图

13.3 程序设计

了解了 GPIO 的结构原理和寄存器，还有我们的实验功能，下面开始设计程序。

13.3.1 GPIO 的 HAL 库驱动分析

HAL 库中关于 GPIO 的驱动程序在 stm32f4xx_hal_gpio.c 文件以及其对应的头文件。

1. HAL_GPIO_Init 函数

要使用一个外设我们首先要对它进行初始化，所以我们先看外设 GPIO 的初始化函数。其声明如下：

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

- **函数描述：**

  用于配置 GPIO 功能模式，还可以设置 EXTI 功能。

- **函数形参：**

  形参 1 是端口号，可以有以下的选择：

```
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC                ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD                ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE                ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF                ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG                ((GPIO_TypeDef *) GPIOG_BASE)
#define GPIOH                ((GPIO_TypeDef *) GPIOH_BASE)
#define GPIOI                ((GPIO_TypeDef *) GPIOI_BASE)
#define GPIOJ                ((GPIO_TypeDef *) GPIOJ_BASE)
#define GPIOK                ((GPIO_TypeDef *) GPIOK_BASE)
```

这是库里面的选择项，实际上我们的芯片只能从 GPIOA~GPIOI，因为我们只有 9 组 IO 口。

形参 2 是 GPIO_InitTypeDef 类型的结构体变量，其定义如下：

```
typedef struct
{
  uint32_t Pin;          /* 引脚号 */
  uint32_t Mode;         /* 模式设置 */
  uint32_t Pull;         /* 上拉下拉设置 */
  uint32_t Speed;        /* 速度设置 */
} GPIO_InitTypeDef;
```

该结构体很重要，下面对每个成员介绍一下。

成员 Pin 表示引脚号，范围：GPIO_PIN_0 到 GPIO_PIN_15，另外还有 GPIO_PIN_All 和 GPIO_PIN_MASK 可选。

成员 Mode 是 GPIO 的模式选择，有以下选择项：

```
#define  GPIO_MODE_INPUT                (0x00000000U)   /* 输入模式 */
#define  GPIO_MODE_OUTPUT_PP            (0x00000001U)   /* 推挽输出 */
```

```
#define  GPIO_MODE_OUTPUT_OD            (0x00000011U)   /* 开漏输出 */
#define  GPIO_MODE_AF_PP                (0x00000002U)   /* 推挽式复用 */
#define  GPIO_MODE_AF_OD                (0x00000012U)   /* 开漏式复用 */
#define  GPIO_MODE_AF_INPUT             GPIO_MODE_INPUT

#define  GPIO_MODE_ANALOG               (0x00000003U)   /* 模拟模式 */

#define  GPIO_MODE_IT_RISING            (0x11110000U)   /* 外部中断,上升沿触发检测 */
#define  GPIO_MODE_IT_FALLING           (0x11210000U)   /* 外部中断,下降沿触发检测 */
/* 外部中断，上升和下降双沿触发检测 */
#define  GPIO_MODE_IT_RISING_FALLING    (0x11310000U)

#define  GPIO_MODE_EVT_RISING           (0x11120000U)   /* 外部事件,上升沿触发检测 */
#define  GPIO_MODE_EVT_FALLING          (0x11220000U)   /* 外部事件,下降沿触发检测 */
/* 外部事件，上升和下降双沿触发检测 */
#define  GPIO_MODE_EVT_RISING_FALLING   (0x11320000U)
```

成员 Pull 用于配置上下拉电阻，有以下选择项：

```
#define  GPIO_NOPULL                (0x00000000U)   /* 无上下拉 */
#define  GPIO_PULLUP                (0x00000001U)   /* 上拉 */
#define  GPIO_PULLDOWN              (0x00000002U)   /* 下拉 */
```

成员 Speed 用于配置 GPIO 的速度，有以下选择项：

```
#define  GPIO_SPEED_FREQ_LOW        (0x00000002U)   /* 低速 */
#define  GPIO_SPEED_FREQ_MEDIUM     (0x00000001U)   /* 中速 */
#define  GPIO_SPEED_FREQ_HIGH       (0x00000003U)   /* 高速 */
```

- **函数返回值：**

  无

- **注意事项：**

HAL 库的 EXTI 外部中断的设置功能整合到此函数里面，而不是单独一个文件。这个我们到外部中断实验再细讲。

2. HAL_GPIO_WritePin 函数

HAL_GPIO_WritePin 函数是 GPIO 口的写引脚函数。其声明如下：

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

- **函数描述：**

  用于设置引脚输出高电平或者低电平，通过 BSRR 寄存器复位或者置位操作。

- **函数形参：**

  形参 1 是端口号，可以选择范围：GPIOA~GPIOG。

  形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。

  形参 3 是要设置输出的状态，是枚举型有两个选择：GPIO_PIN_SET 表示高电平，GPIO_PIN_RESET 表示低电平。

- **函数返回值：**

  无

3. HAL_GPIO_TogglePin 函数

HAL_GPIO_TogglePin 函数是 GPIO 口的电平翻转函数。其声明如下：

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

- **函数描述：**

  用于设置引脚的电平翻转，也是通过 BSRR 寄存器复位或者置位操作。

- **函数形参：**

  形参 1 是端口号，可以选择范围：GPIOA~GPIOG。

  形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。

- **函数返回值：**

  无

本实验我们用到上面三个函数，其他的 API 函数后面用到再进行讲解。

GPIO 输出配置步骤

1）使能对应 GPIO 时钟

STM32 在使用任何外设之前，我们都要先使能其时钟（下同）。本实验用到 PE0 和 PE1 两个 IO 口，因此需要先使能 GPIOE 的时钟，代码如下:

```
__HAL_RCC_GPIOE_CLK_ENABLE();
```

2）设置对应 GPIO 工作模式（推挽输出）

本实验 GPIO 使用推挽输出模式，控制 LED 亮灭，通过函数 HAL_GPIO_Init 设置实现。

3）控制 GPIO 引脚输出高低电平

在配置好 GPIO 工作模式后，我们就可以通过 HAL_GPIO_WritePin 函数控制 GPIO 引脚输出高低电平，从而控制 LED 的亮灭了。

13.3.2 程序流程图

程序流程图能帮助我们更好的理解一个工程的功能和实现的过程，对学习和设计工程有很好的主导作用。本实验的程序流程图如下：

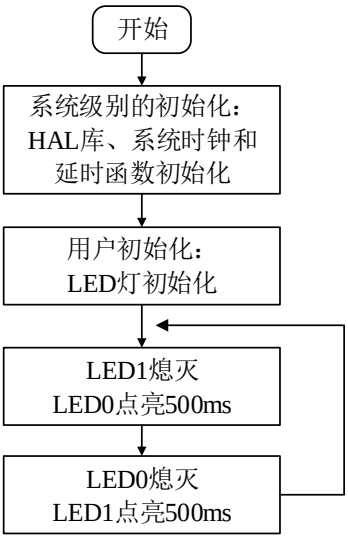


图 13.3.2.1 跑马灯实验程序流程图

13.3.3 程序解析

1. led 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。LED 驱动源码包括两个文件：led.c 和 led.h（正点原子编写的外设驱动基本都是包含一个.c 文件和一个.h 文件，下同）。

下面我们先解析 led.h 的程序，我们把它分两部分功能进行讲解。

- **LED 灯引脚宏定义**

由硬件设计小节，我们知道 LED 灯在硬件上分别连接到 PE0 和 PE1，再结合 HAL 库，我们做了下面的引脚定义。

```
/* LED0 引脚定义 */
#define LED0_GPIO_PORT              GPIOE
#define LED0_GPIO_PIN               GPIO_PIN_0
#define LED0_GPIO_CLK_ENABLE()      do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
/* LED1 引脚定义 */
#define LED1_GPIO_PORT              GPIOE
#define LED1_GPIO_PIN               GPIO_PIN_1
#define LED1_GPIO_CLK_ENABLE()      do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
```

这样的好处是进一步隔离底层函数操作，移植更加方便，函数命名更亲近实际的开发板。比如：当我们看到 LED0_GPIO_PORT 这个宏定义，我们就知道这是灯 LED0 的端口号；看到

LED0_GPIO_PIN 这个宏定义，就知道这是灯 LED0 的引脚号；看到 LED0_GPIO_CLK_ENABLE 这个宏定义，就知道这是灯 LED0 的时钟使能函数。大家后面学习时间长了就会慢慢熟悉这样的命名方式。

特别注意：这里的时钟使能函数宏定义，使用了 do{ }while(0)结构，是为了避免在某些使用场景出错的问题（下同），详见《嵌入式单片机 C 代码规范与风格》第六章第 2 点。

__HAL_RCC_GPIOx_CLK_ENABLE 函数是 HAL 库的 IO 口时钟使能函数，x=A 到 I。

- **LED 灯操作函数宏定义**

为了后续对 LED 灯进行便捷的操作，我们为 LED 灯操作函数做了下面的定义。

```
/* LED 端口操作定义 */
#define LED0(x)   do{ x ? \
              HAL_GPIO_WritePin(LED0_GPIO_PORT,LED0_GPIO_PIN, GPIO_PIN_SET) : \
              HAL_GPIO_WritePin(LED0_GPIO_PORT,LED0_GPIO_PIN, GPIO_PIN_RESET);\
                  }while(0)      /* LED0 翻转 */
#define LED1(x)   do{ x ? \
             HAL_GPIO_WritePin(LED1_GPIO_PORT, LED1_GPIO_PIN, GPIO_PIN_SET) : \
             HAL_GPIO_WritePin(LED1_GPIO_PORT, LED1_GPIO_PIN, GPIO_PIN_RESET);\
                  }while(0)      /* LED1 翻转 */
/* LED 电平翻转定义 */
#define LED0_TOGGLE()    do{ HAL_GPIO_TogglePin(LED0_GPIO_PORT,
                             LED0_GPIO_PIN); }while(0)   /* LED0 = !LED0 */
#define LED1_TOGGLE()    do{ HAL_GPIO_TogglePin(LED1_GPIO_PORT,
                             LED1_GPIO_PIN); }while(0)   /* LED1 = !LED1 */
```

LED0 和 LED1 这两个宏定义，分别是控制 LED0 和 LED1 的亮灭。例如：对于宏定义标识符 LED0(x)，它的值是通过条件运算符来确定：

当 x=0 时，宏定义的值为 HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_RESET)，也就是设置 LED0_GPIO_PORT(PE0)输出低电平；

当 n!=0 时，宏定义的值为 HAL_GPIO_WritePin (LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_SET)，也就是设置 LED0_GPIO_PORT(PE0)输出高电平。

根据前述定义，如果要设置 LED0 输出低电平，那么调用宏定义 LED0(0)即可，如果要设置 LED0 输出高电平，调用宏定义 LED0(1)即可。宏定义 LED1(x)同理。

LED0_TOGGLE 和 LED1_TOGGLE 这三个宏定义，分别是控制 LED0 和 LED1 的翻转。这里利用 HAL_GPIO_TogglePin 函数实现 IO 口输出电平翻转操作。

下面我们再解析 led.c 的程序，这里只有一个函数 led_init，这是 LED 灯的初始化函数，其定义如下：

```
/**
 * @brief       初始化 LED 相关 IO 口, 并使能时钟
 * @param       无
 * @retval      无
 */
void led_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    LED0_GPIO_CLK_ENABLE();                                 /* LED0 时钟使能 */
    LED1_GPIO_CLK_ENABLE();                                 /* LED1 时钟使能 */
    gpio_init_struct.Pin = LED0_GPIO_PIN;                   /* LED0 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
    HAL_GPIO_Init(LED0_GPIO_PORT, &gpio_init_struct);       /* 初始化 LED0 引脚 */
    gpio_init_struct.Pin = LED1_GPIO_PIN;                   /* LED1 引脚 */
    HAL_GPIO_Init(LED1_GPIO_PORT, &gpio_init_struct);       /* 初始化 LED1 引脚 */

    LED0(1);                                                /* 关闭 LED0 */
    LED1(1);                                                /* 关闭 LED1 */
}
```

对 LED 灯的两个引脚都设置为高速上拉的推挽输出。最后关闭 LED 灯的输出，防止没有

操作就亮了。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    led_init();                                 /* 初始化 LED */

    while(1)
    {
        LED0(0);                                /* LED0 亮 */
        LED1(1);                                /* LED1 灭 */
        delay_ms(500);
        LED0(1);                                /* LED0 灭 */
        LED1(0);                                /* LED1 亮 */
        delay_ms(500);
    }
}
```

首先是调用系统级别的初始化：初始化 HAL 库、系统时钟和延时函数。接下来，调用 led_init 来初始化 LED 灯。最后在无限循环里面实现 LED0 和 LED1 间隔 500ms 交替闪烁一次。

13.4 下载验证

我们先来看看编译结果，如图 13.4.1 所示。

```
Build Output
compiling stm32f4xx_hal_uart.c...
compiling led.c...
compiling stm32f4xx_hal_dma_ex.c...
linking...
Program Size: Code=5598 RO-data=450 RW-data=28 ZI-data=1388
"..\..\Output\atk_f407.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:07
```

图 13.4.1 编译结果

可以看到没有 0 错误，0 警告。从编译信息可以看出，我们的代码占用 FLASH 大小为：6076 字节（5598+450+28），所用的 SRAM 大小为：1416 个字节（28+1388）。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（FLASH + RAM）

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

有了这个就可以知道你当前使用的 flash 和 ram 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。这里我们使用 DAP 仿真器（也可以使用其他调试器）下载。

下载完之后，可以看到 LED0 和 LED1 交替亮。

至此，我们的跑马灯实验的学习就结束了，本章介绍了 STM32F407 的 IO 口的使用及注意事项，是后面学习的基础，希望大家好好理解。

第十四章 蜂鸣器实验

上一章，我们介绍了 STM32F407 的 IO 口作为输出的使用。本章，我们将通过另外一个例子继续讲述 STM32F407 的 IO 口作为输出的使用，不同的是本章讲的不是用 IO 口直接驱动器件，而是通过三极管间接驱动。我们将利用一个 IO 口来控制板载的有源蜂鸣器。

本章分为如下几个小节：

14.1 蜂鸣器简介
14.2 硬件设计
14.3 程序设计
14.4 下载验证

14.1 蜂鸣器简介

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型。

DMF407 板载的蜂鸣器是电磁式的有源蜂鸣器，如图 14.1.1 所示：



图 14.1.1 有源蜂鸣器

这里的有源不是指电源的“源”，而是指有没有自带震荡电路，有源蜂鸣器自带了震荡电路，一通电就会发声；无源蜂鸣器则没有自带震荡电路，必须外部提供 2~5Khz 左右的方波驱动，才能发声。

上一章，我们利用 STM32 的 IO 口直接驱动 LED 灯，本章的蜂鸣器，我们能否直接用 STM32 的 IO 口驱动呢？让我们来分析一下：STM32F407 的单个 IO 最大可以提供 25mA 电流（来自数据手册），而蜂鸣器的驱动电流是 30mA 左右，两者十分相近，但是全盘考虑，STM32F407 整个芯片的电流，最大也就 150mA，如果用 IO 口直接驱动蜂鸣器，其他地方用电就得省着点了，所以我们不用 STM32F407 的 IO 直接驱动蜂鸣器，而是通过三极管扩流后再驱动蜂鸣器，这样 STM32F407 的 IO 只需要提供不到 1mA 的电流就足够了。

IO 口使用虽然简单，但是和外部电路的匹配设计，还是要十分讲究的，考虑越多，设计就越可靠，可能出现的问题也就越少。

14.2 硬件设计

1. 例程功能

蜂鸣器每隔 300ms 响或者停一次。LED0 每隔 300ms 亮或者灭一次。LED0 亮的时候蜂鸣器不叫，而 LED0 熄灭的时候，蜂鸣器叫。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）蜂鸣器
BEEP - PF0

3. 原理图

蜂鸣器在硬件上是直接连接好了的，不需要经过任何设置，直接编写代码就可以了。蜂鸣器的驱动信号连接在 STM32F407 的 PF0 上。如图 14.2.1 所示：

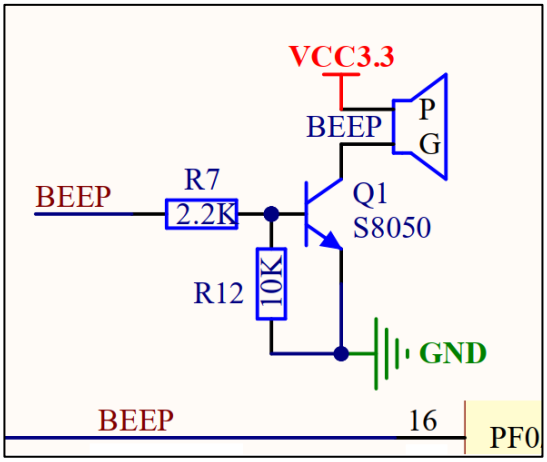


图 14.2.1 蜂鸣器与 STM32F4 连接原理图

我们用一个 NPN 三极管（S8050）来驱动蜂鸣器，驱动信号通过 R7 和 R12 间的电压获得，芯片上电时默认电平为低电平，故上电时蜂鸣器不会直接响起。当 PF0 输出高电平的时候，蜂鸣器将发声，当 PF0 输出低电平的时候，蜂鸣器停止发声。

14.3 程序设计

本实验我们只是用到 GPIO 外设输出功能，关于 HAL 库的 GPIO 的 API 函数请看跑马灯实验的介绍。下面我们直接分析本实验的程序流程图。

14.3.1 程序流程图

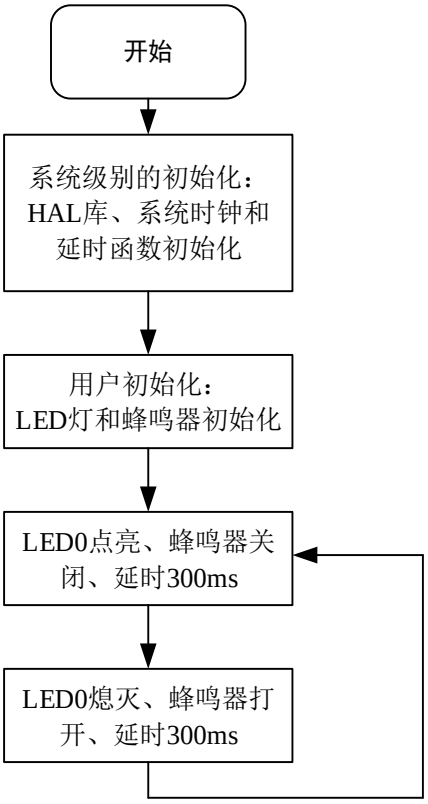


图 14.3.1.1 蜂鸣器实验程序流程图

14.3.2 程序解析

1. 蜂鸣器驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。蜂鸣器（BEEP）驱动源码包括两个文件：beep.c 和 beep.h。

下面我们先解析 beep.h 的程序，我们把它分两部分功能进行讲解。

- **蜂鸣器引脚定义**

由硬件设计小节，我们知道驱动蜂鸣器的三极管在硬件上连接到 PF0，类似跑马灯实验，我们做了下面的引脚定义。

```
/* 引脚 定义 */
#define BEEP_GPIO_PORT                  GPIOF
#define BEEP_GPIO_PIN                   GPIO_PIN_0
/* PB 口时钟使能 */
#define BEEP_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)
```

- **蜂鸣器操作函数定义**

为了后续对蜂鸣器进行便捷的操作，我们为蜂鸣器操作函数做了下面的定义。

```
/* 蜂鸣器控制 */
#define BEEP(x)         do{ x ? \
                HAL_GPIO_WritePin(BEEP_GPIO_PORT,BEEP_GPIO_PIN,GPIO_PIN_SET):\
                HAL_GPIO_WritePin(BEEP_GPIO_PORT, BEEP_GPIO_PIN,GPIO_PIN_RESET);\
                        }while(0)
/* BEEP 状态翻转 */
#define BEEP_TOGGLE()   do{HAL_GPIO_TogglePin(BEEP_GPIO_PORT,BEEP_GPIO_PIN);\
                         }while(0)
```

BEEP(x)这个宏定义就是控制蜂鸣器的打开和关闭的。例如：如果要打开蜂鸣器，那么调用宏定义 BEEP(1)即可，如果要关闭蜂鸣器，调用宏定义 BEEP(0)即可。

BEEP_TOGGLE()是控制蜂鸣器进行翻转的。这里也利用 HAL_GPIO_TogglePin 函数实现 IO 口输出电平取反操作。

下面我们再解析 beep.c 的程序，这里只有一个函数 beep_init，这是蜂鸣器的初始化函数，其定义如下：

```
/**
 * @brief       初始化 BEEP 相关 IO 口, 并使能时钟
 * @param       无
 * @retval      无
 */
void beep_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    BEEP_GPIO_CLK_ENABLE();                                 /* BEEP 时钟使能 */

    gpio_init_struct.Pin = BEEP_GPIO_PIN;                   /* 蜂鸣器引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
    HAL_GPIO_Init(BEEP_GPIO_PORT, &gpio_init_struct);       /* 初始化蜂鸣器引脚 */

    BEEP(0);                                                /* 关闭蜂鸣器 */
}
```

对蜂鸣器的控制引脚模式设置为高速上拉的推挽输出。最后关闭关闭蜂鸣器，防止没有操作就响了。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
```

```
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    led_init();                                 /* 初始化 LED */
    beep_init();                                /* 初始化蜂鸣器 */

    while(1)
    {
        LED0(0);
        BEEP(0);
        delay_ms(300);
        LED0(1);
        BEEP(1);
        delay_ms(300);
    }
}
```

首先初始化 HAL 库、系统时钟和延时函数。接下来，调用 led_init 来初始化 LED 灯，调用 beep_init 函数初始化蜂鸣器。最后在无限循环里面实现 LED0 和蜂鸣器间隔 300ms 交替闪烁和打开关闭一次。

14.4 下载验证

下载完之后，可以看到 LED0 亮的时候蜂鸣器不叫，而 LED0 熄灭的时候，蜂鸣器叫（因为他们的有效信号相反）。间隔为 0.3 秒左右，符合预期设计。

至此，本章的学习就结束了。通过本章，我们进一步学习 IO 作为输出的使用方法，同时巩固了前面知识的学习。希望大家在开发板上实际验证一下，从而加深印象。

第十五章 按键输入实验

上一章，我们介绍了 STM32F407 的 IO 口作为输出的使用。本章，我们将向大家介绍如何使用 STM32F407 的 IO 口作为输入。我们将利用板载的 3 个按键，来控制板载的两个 LED 灯亮灭。通过本章的学习，我们将了解到 STM32F407 的 IO 口作为输入的使用方法。

本章分为如下几个小节

15.1 按键与输入数据寄存器

15.2 硬件设计

15.3 程序设计

15.4 下载验证

15.1 按键与输入数据寄存器简介

15.1.1 独立按键简介

几乎每个开发板都会板载有独立按键，因为按键用处很多。常态下，独立按键是断开的，按下的时候才闭合。每个独立按键会单独占用一个 IO 口，通过 IO 口的高低电平判断按键的状态。但是按键在闭合和断开的时候，都存在抖动现象，即按键在闭合时不会马上就稳定的连接，断开时也不会马上断开。这是机械触点，无法避免。独立按键抖动波形图如下：

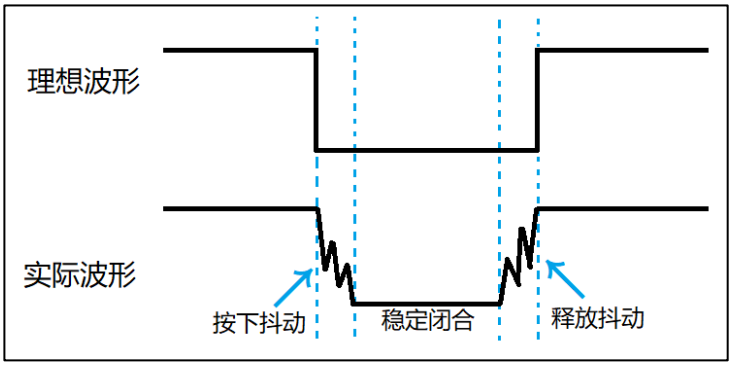


图 15.1.1.1 独立按键抖动波形图

图中的按下抖动和释放抖动的时间一般为 5~10ms，如果在抖动阶段采样，其不稳定状态可能出现一次按键动作被认为是多次按下的情况。为了避免抖动可能带来的误操作，我们要做的措施就是给按键消抖（即采样稳定闭合阶段）。消抖方法分为硬件消抖和软件消抖，我们常用软件的方法消抖。

软件消抖：方法很多，我们例程中使用最简单的延时消抖。检测到按键按下后，一般进行 10ms 延时，用于跳过抖动的时间段，如果消抖效果不好可以调整这个 10ms 延时，因为不同类型的按键抖动时间可能有偏差。待延时过后再检测按键状态，如果没有按下，那我们就判断这是抖动或者干扰造成的；如果还是按下，那么我们就认为这是按键真的按下了。对按键释放的判断同理。

硬件消抖：利用 RC 电路的电容充放电特性来对抖动产生的电压毛刺进行平滑出来，从而实现消抖，但是成本会更高一点，本着能省则省的原则，我们推荐使用软件消抖即可。

15.1.2 GPIO 端口输入数据寄存器（IDR）

本实验我们将会用到 GPIO 端口输入数据寄存器，下面来介绍一下。

该寄存器用于存储 GPIOx 的输入状态，它连接到施密特触发器上，IO 口外部的电平信号经过触发器后，模拟信号就被转化成 0 和 1 这样的数字信号，并存储到该寄存器中。寄存器描述如图 15.1.2.1 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。

图 15.1.2.1 GPIOx IDR 寄存器描述

该寄存器低 16 位有效，分别对应每一组 GPIO 的 16 个引脚。当 CPU 访问该寄存器，如果对应的某位为 0(IDRy=0)，则说明该 IO 口输入的是低电平，如果是 1(IDRy=1)，则表示输入的是高电平，y=0~15。

15.2 硬件设计

1. 例程功能

通过开发板上的三个独立按键控制 LED 灯：KEY0 控制 LED0 翻转，KEY1 控制 LED1 翻转，KEY2 控制 LED0、LED1 同时翻转。

2. 硬件资源

1）LED 灯
LED0 – PE0
LED1 – PE1
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4

3. 原理图

独立按键硬件部分的原理图，如图 15.2.1 所示：

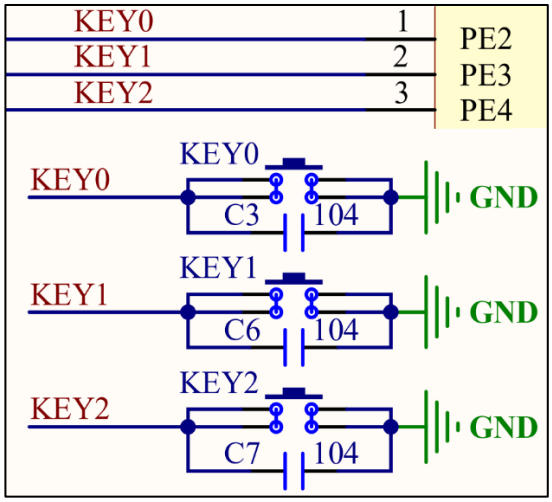


图 15.2.1 独立按键原理图

这里需要注意的是：KEY0、KEY1 和 KEY2 设计为采样到按键另一端的低电平为有效电平，由于按键外部没有上下拉电阻，所以需要在 STM32F407 内部设置上下拉。

15.3 程序设计

15.3.1 HAL_GPIO_ReadPin 函数

HAL_GPIO_ReadPin 函数是 GPIO 口的读引脚函数。其声明如下：

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

- **函数描述：**
  用于读取 GPIO 引脚状态，通过 IDR 寄存器读取。
- **函数形参：**
  形参 1 是端口号，可以选择范围：GPIOA~GPIOI。
  形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。
- **函数返回值：**
  引脚状态值 0 或者 1

GPIO 输入配置步骤

1）使能对应 GPIO 时钟

本实验按键用到 PE2/3/4 三个 IO 口，因此需要先使能 GPIOE 的时钟，代码如下:

```
__HAL_RCC_GPIOE_CLK_ENABLE();
```

2）设置对应 GPIO 工作模式（上拉/下拉输入）

本实验 GPIO 使用输入模式（带上拉/下拉），从而可以读取 IO 口的状态，实现按键检测，GPIO 模式通过函数 HAL_GPIO_Init 设置实现。

3）读取 GPIO 引脚高低电平

在配置好 GPIO 工作模式后，我们就可以通过 HAL_GPIO_ReadPin 函数读取 GPIO 引脚的高低电平，从而实现按键检测了。

15.3.2 程序流程图

我们为各个按键定义键值，下面看看本实验的程序流程图：

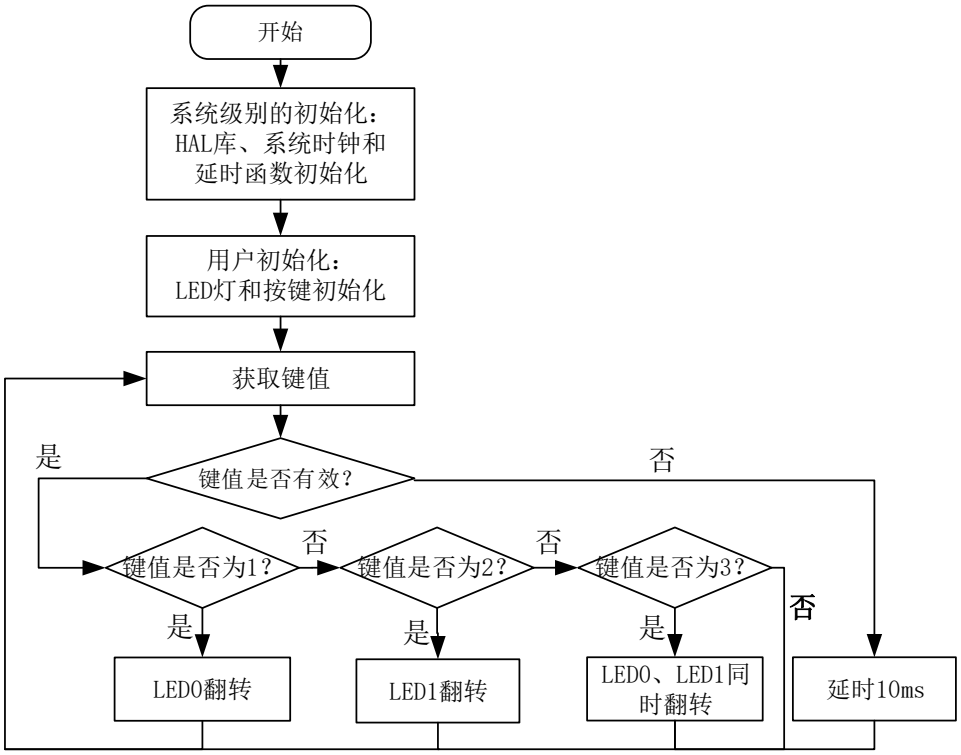


图 15.3.2.1 按键输入实验程序流程图

15.3.3 程序解析

1. 按键驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。按键（KEY）驱动源码包括两个文件：key.c 和 key.h。

下面我们先解析 key.h 的程序，我们把它分两部分功能进行讲解。

- **按键引脚定义**

由硬件设计小节，我们知道 KEY0、KEY1、KEY2 分别来连接到 PE2、PE3、PE4 上，我们做了下面的引脚定义。

```
/* 引脚 定义 */
#define KEY0_GPIO_PORT                  GPIOE
#define KEY0_GPIO_PIN                   GPIO_PIN_2
/* PE 口时钟使能 */
#define KEY0_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define KEY1_GPIO_PORT                  GPIOE
#define KEY1_GPIO_PIN                   GPIO_PIN_3
/* PE 口时钟使能 */
#define KEY1_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)

#define KEY2_GPIO_PORT                  GPIOE
#define KEY2_GPIO_PIN                   GPIO_PIN_4
/* PE 口时钟使能 */
#define KEY2_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
```

- **按键操作函数定义**

为了后续对按键进行便捷的操作，我们为按键操作函数做了下面的定义。

```
#define KEY0  HAL_GPIO_ReadPin(KEY0_GPIO_PORT, KEY0_GPIO_PIN)  /* 读取 KEY0 引脚 */
#define KEY1  HAL_GPIO_ReadPin(KEY1_GPIO_PORT, KEY1_GPIO_PIN)  /* 读取 KEY1 引脚 */
#define KEY2  HAL_GPIO_ReadPin(KEY2_GPIO_PORT, KEY2_GPIO_PIN)  /* 读取 KEY2 引脚 */

#define KEY0_PRES    1              /* KEY0 按下 */
#define KEY1_PRES    2              /* KEY1 按下 */
#define KEY2_PRES    3              /* KEY2 按下 */
```

KEY0、KEY1、KEY2 分别是读取对应按键状态的宏定义。用 HAL_GPIO_ReadPin 函数实现，该函数的返回值就是 IO 口的状态，返回值是枚举类型，取值 0 或者 1。

KEY0_PRES、KEY1_PRES、KEY2_PRES 则是按键对应的三个键值宏定义标识符。

下面我们再解析 key.c 的程序，这里有两个函数，先看按键初始化函数，其定义如下：

```
/**
 * @brief       按键初始化函数
 * @param       无
 * @retval      无
 */
void key_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;                  /* GPIO 配置参数存储变量 */
    KEY0_GPIO_CLK_ENABLE();                             /* KEY0 时钟使能 */
    KEY1_GPIO_CLK_ENABLE();                             /* KEY1 时钟使能 */
    KEY2_GPIO_CLK_ENABLE();                             /* KEY2 时钟使能 */

    gpio_init_struct.Pin = KEY0_GPIO_PIN;               /* KEY0 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_INPUT;            /* 输入 */
    gpio_init_struct.Pull = GPIO_PULLUP;                /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /* 高速 */
    HAL_GPIO_Init(KEY0_GPIO_PORT, &gpio_init_struct);   /* KEY0 引脚模式设置 */

    gpio_init_struct.Pin = KEY1_GPIO_PIN;               /* KEY1 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_INPUT;            /* 输入 */
```

```
    gpio_init_struct.Pull = GPIO_PULLUP;                /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /* 高速 */
    HAL_GPIO_Init(KEY1_GPIO_PORT, &gpio_init_struct);   /* KEY1 引脚模式设置 */

    gpio_init_struct.Pin = KEY2_GPIO_PIN;               /* KEY2 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_INPUT;            /* 输入 */
    gpio_init_struct.Pull = GPIO_PULLUP;                /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /* 高速 */
    HAL_GPIO_Init(KEY2_GPIO_PORT, &gpio_init_struct);   /* KEY2 引脚模式设置 */
}
```

这里需要注意的是：KEY0、KEY1 和 KEY2 是低电平有效的（即一端接地），所以我们要设置为内部上拉。

另一个函数是按键扫描函数，其定义如下：

```
/**
 * @brief       按键扫描函数
 * @note        该函数有响应优先级(同时按下多个按键): KEY2 > KEY1 > KEY0!!
 * @param       mode:0 / 1, 具体含义如下:
 *   @arg       0, 不支持连续按(当按键按下不放时, 只有第一次调用会返回键值,
 *                 必须松开以后, 再次按下才会返回其他键值)
 *   @arg       1, 支持连续按(当按键按下不放时, 每次调用该函数都会返回键值)
 * @retval      键值, 定义如下:
 *              KEY0_PRES, 1, KEY0 按下
 *              KEY1_PRES, 2, KEY1 按下
 *              KEY2_PRES, 3, KEY2 按下
 */
uint8_t key_scan(uint8_t mode)
{
    static uint8_t key_up = 1;  /* 按键按松开标志 */
    uint8_t keyval = 0;
    if (mode) key_up = 1;       /* 支持连按 */

    if (key_up && (KEY0 == 0 || KEY1 == 0 || KEY2 == 0))
    {/* 按键松开标志为 1, 且有任意一个按键按下了 */
        delay_ms(10);           /* 去抖动 */
        key_up = 0;
        if (KEY0 == 0)  keyval = KEY0_PRES;
        if (KEY1 == 0)  keyval = KEY1_PRES;
        if (KEY2 == 0)  keyval = KEY2_PRES;
    }
    else if (KEY0 == 1 && KEY1 == 1 && KEY2 == 1)
    {/* 没有任何按键按下, 标记按键松开 */
        key_up = 1;
    }

    return keyval;              /* 返回键值 */
}
```

key_scan 函数用于扫描这 3 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。可以看到该函数的消抖延时是 10ms。同时还有一点要注意的是，该函数的按键扫描是有优先级的，最优先的是 KEY2，第二优先的是 KEY1，最后是按键 KEY0。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。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */

    while(1)
    {
        key = key_scan(0);                      /* 得到键值 */

        if (key)
        {
            switch (key)
            {
                case KEY0_PRES:                 /* 控制 LED0(RED)翻转 */
                    LED0_TOGGLE();              /* LED0 状态取反 */
                    break;

                case KEY1_PRES:                 /* 控制 LED1(GREEN)翻转 */
                    LED1_TOGGLE();              /* LED1 状态取反 */
                    break;

                case KEY2_PRES:                 /* 同时控制 LED0, LED1 翻转 */
                    LED0_TOGGLE();              /* LED0 状态取反 */
                    LED1_TOGGLE();              /* LED1 状态取反 */
                    break;
                default : break;
            }
        }
        else
        {
            delay_ms(10);
        }
    }
}
```

首先是调用系统级别的初始化：初始化 HAL 库、系统时钟和延时函数。接下来，调用 led_init 来初始化 LED 灯，调用 key_init 函数初始化按键。最后在无限循环里面扫描获取键值，接着用键值判断哪个按键按下，如果有按键按下则翻转相应的灯，如果没有按键按下则延时 10ms。

15.4 下载验证

在下载好程序后，我们可以按 KEY0、KEY1、KEY2 来看看 LED 灯的变化，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。本章学习了 STM32F407 的 IO 作为输入的使用方法，在前面的 GPIO 输出的基础上又学习了一种 GPIO 使用模式，大家可以回顾前面跑马灯实验介绍的 GPIO 的八种模式类型巩固 GPIO 的知识。

第十六章 外部中断实验

在前面几章的学习中，我们掌握了 STM32F407 的 IO 口最基本的操作。本章我们将介绍如何把 STM32F407 的 IO 口作为外部中断输入来使用，在本章中，我们将以中断的方式，实现我们在第十五章所实现的功能。

本章分为如下几个小节：

16.1 STM32F4 NVIC 和外部中断简介
16.2 硬件设计
16.3 程序设计
16.4 下载验证

16.1 NVIC 和 EXTI 简介

16.1.1 NVIC 简介

什么是 NVIC？NVIC 即嵌套向量中断控制器，全称 Nested vectored interrupt controller。它是内核的器件，所以它的更多描述可以看内核有关的资料。M3/M4/M7 内核都是支持 256 个中断，其中包含了 16 个系统中断和 240 个外部中断，并且具有 256 级的可编程中断设置。然而芯片厂商一般不会把内核的这些资源全部用完，如 STM32F407 的系统中断有 10 个，外部中断有 82 个。这个在讲解启动文件的时候有提到过，可以回顾一下。下面我们看看系统中断部分：

位置	优先级	优先级类型	名称	说明	地址
	-	-	-	保留	0x0000_0000
	-3	固定	Reset	复位	0x0000_0004
	-2	固定	NMI	不可屏蔽中断 RCC时钟安全系统(CSS)联接到NMI向量	0x0000_0008
	-1	固定	硬件失效(HardFault)	所有类型的失效	0x0000_000C
	0	可设置	存储管理(MemManage)	存储器管理	0x0000_0010
	1	可设置	总线错误(BusFault)	预取指失败，存储器访问失败	0x0000_0014
	2	可设置	错误应用(UsageFault)	未定义的指令或非法状态	0x0000_0018
	-	-	-	保留	0x0000_001C ~0x0000_002B
	3	可设置	SVCall	通过SWI指令的系统服务调用	0x0000_002C
	4	可设置	调试监控(DebugMonitor)	调试监控器	0x0000_0030
	-	-	-	保留	0x0000_0034
	5	可设置	PendSV	可挂起的系统服务	0x0000_0038
	6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C

表 16.1.1.1 中断向量表-系统中断部分

关于 82 个外部中断部分在《STM32F4xx 参考手册_V4（中文版）.pdf》的 10.2 小节有详细的列表，这里就不列出来了。STM32F407 的中断向量表在 stm32f407xx.h 文件中被定义。

16.1.1.1 NVIC 寄存器

NVIC 相关的寄存器定义了可以在 core_cm4.h 文件中找到。我们直接通过程序的定义来分析 NVIC 相关的寄存器，其定义如下：

```
typedef struct
{
```

```
    __IOM   uint32_t ISER[8U];              /* 中断使能寄存器 */
            uint32_t RESERVED0[24U];
    __IOM uint32_t ICER[8U];                /* 中断清除使能寄存器 */
          uint32_t RSERVED1[24U];
    __IOM uint32_t ISPR[8U];                /* 中断使能挂起寄存器 */
          uint32_t RESERVED2[24U];
    __IOM uint32_t ICPR[8U];                /* 中断解挂寄存器 */
          uint32_t RESERVED3[24U];
    __IOM uint32_t IABR[8U];                /* 中断有效位寄存器 */
          uint32_t RESERVED4[56U];
    __IOM uint8_t  IP[240U];                /* 中断优先级寄存器（8Bit 位宽） */
          uint32_t RESERVED5[644U];
    __OM  uint32_t STIR;                    /* 中断触发中断寄存器 */
} NVIC_Type;
```

STM32F407 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 STM32F407 的中断。下面重点介绍这几个寄存器：

ISER[8]：ISER 全称是：Interrupt Set Enable Registers，这是一个中断使能寄存器组。上面说了 CM4 内核支持 256 个中断，这里用 8 个 32 位寄存器来控制，每个位控制一个中断。但是 STM32F407 的可屏蔽中断最多只有 82 个，所以对我们来说，有用的就是两个(ISER[0~3])，总共可以表示 128 个中断。而 STM32F407 只用了其中的 82 个。ISER[0]的 bit0~31 分别对应中断 0~31；ISER[1]的 bit0~31 对应中断 32~63; ISER[2]的 bit0~16 对应中断 64~81，这样总共 82 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 stm32f407xx.h 里面的第 68 行。

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，是一个中断使能挂起控制寄存器组。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，可以将挂起的中断解挂。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。在中断执行完成后，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！STM32F407 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 STM32F407 只用到了其中的 82 个。IP[81]~IP[0]分别对应中断 81~0。而每个可屏蔽中断占用的 8bit 并没有全部使用，而是只用了高 4 位。这 4 位，又分为抢占优先级和子优先级。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。关于中断优先级控制的寄存器组我们下面详细讲解。

16.1.1.2 中断优先级

STM32 中的中断优先级可以分为：抢占式优先级和响应优先级，响应优先级也称子优先级，每个中断源都需要被指定这两种优先级。抢占式优先级和响应优先级的区别：

抢占优先级：抢占优先级高的中断可以打断正在执行的抢占优先级低的中断。

响应优先级：抢占优先级相同，响应优先级高的中断不能打断响应优先级低的中断。

还有一种情况就是当两个或者多个中断的抢占式优先级和响应优先级相同时，那么就遵循自然优先级，看中断向量表的中断排序，数值越小，优先级越高。

在 NVIC 中由寄存器 NVIC_IPR0-NVIC_IPR59 共 60 个寄存器控制中断优先级，每个寄存

器的每 8 位又分为一组，可以分 4 组，所以就有了 240 组宽度为 8bit 的中断优先级控制寄存器，原则上每个外部中断可配置的优先级为 0~255，数值越小，优先级越高。但是实际上 M3 /M4 /M7 芯片为了精简设计，只使用了高四位[7:4]，低四位取零，这样以至于最多只有 16 级中断嵌套，即 2^4=16。

对于 NVCI 的中断优先级分组：STM32F407 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 16.1.1.2.1 所示：

优先级分组	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 16.1.1.2.1 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，例如优先级分组设置为 3，那么此时所有的 82 个中断，每个中断的中断优先寄存器的高四位中的最高 3 位是抢占优先级，低 1 位是响应优先级。每个中断，你可以设置抢占优先级为 0~7，响应优先级为 1 或 0。抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高。

结合实例说明一下：假定设置中断优先级分组为 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！

16.1.1.3 NVIC 相关函数

ST 公司把 core_cm4.h 文件的 NVIC 相关函数封装到 stm32f4xx_hal_cortex.c 文件中，下面列出我们较为常用的函数进行，想了解更多其他的函数请自行查阅。

1. HAL_NVIC_SetPriorityGrouping 函数

HAL_NVIC_SetPriorityGrouping 是设置中断优先级分组函数。其声明如下：

```
void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
```

- **函数描述：**

  用于设置中断优先级分组。

- **函数形参：**

  形参 1 是中断优先级分组号，可以选择范围：NVIC_PRIORITYGROUP_0 到 NVIC_PRIORITYGROUP_4（共 5 组）。

- **函数返回值：**无

- **注意事项：**

  这个函数在一个工程里基本只调用一次，而且是在程序 HAL 库初始化函数里面已经被调用，后续就不会再调用了。因为当后续调用设置成不同的中断优先级分组时，有可能造成前面设置好的抢占优先级和响应优先级不匹配。

2. HAL_NVIC_SetPriority 函数

HAL_NVIC_SetPriority 是设置中断优先级函数。其声明如下：

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

- **函数描述：**

  用于设置中断的抢占优先级和响应优先级（子优先级）。

- **函数形参：**

  形参 1 是中断号，可以选择范围：IRQn_Type 定义的枚举类型，定义在 stm32f407xx.h。

  形参 2 是抢占优先级，可以选择范围：0 到 15。

  形参 3 是响应优先级，可以选择范围：0 到 15。

- **函数返回值：**无

3. HAL_NVIC_EnableIRQ 函数

HAL_NVIC_EnableIRQ 是中断使能函数。其声明如下：

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

- **函数描述：**

  用于使能中断。

- **函数形参：**

  形参 1 是中断号，可以选择范围：IRQn_Type 定义的枚举类型，定义在 stm32f407xx.h。

- **函数返回值：**无

4. HAL_NVIC_DisableIRQ 函数

HAL_NVIC_DisableIRQ 是中断失能函数。其声明如下：

```
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

- **函数描述：**

  用于中断除能。

- **函数形参：**无形参
- **函数返回值：**无

5. HAL_NVIC_SystemReset 函数

HAL_NVIC_SystemReset 是系统复位函数。其声明如下：

```
void HAL_NVIC_SystemReset(void);
```

- **函数描述：**

  用于软件复位系统。

- **函数形参：**无形参
- **函数返回值：**无

其他的 NVIC 函数用得较少，我们就不一一列出来了。NVIC 的介绍就到这，下面介绍外部中断。

16.1.2 EXTI 简介

EXTI 即是外部中断和事件控制器，它是由 20 个产生事件/中断请求的边沿检测器组成。每一条输入线都可以独立地配置输入类型（脉冲或挂起）和对应的触发事件（上升沿或下降沿或者双边沿都触发）。每个输入线都可以独立地被屏蔽。挂起寄存器保持着状态线的中断请求。

EXTI 的功能框图是最直接把有关 EXTI 的知识点连接起来的图，掌握了该图的来龙去脉，就会对 EXTI 有了一个整体熟悉，编程时候可以得心应手。EXTI 的功能框图如图 16.1.2.1。

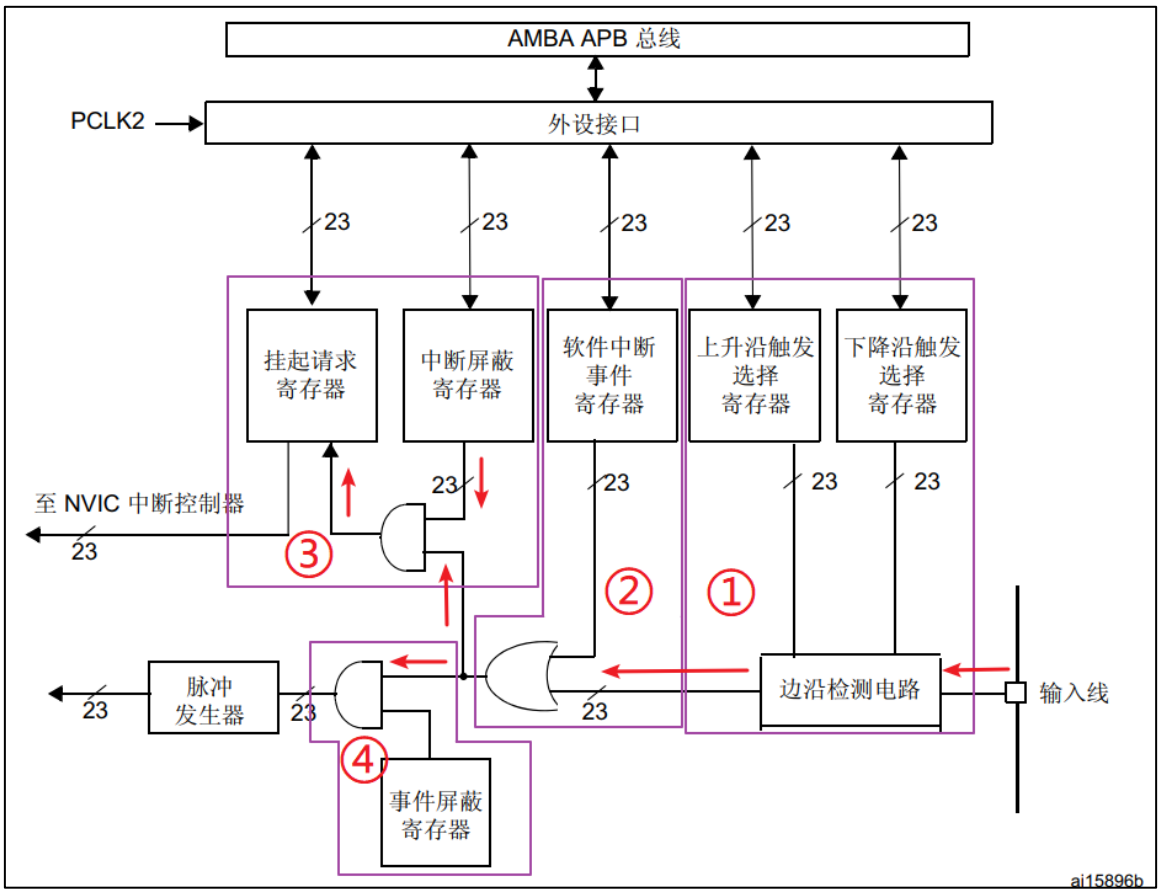


图 16.1.2.1 EXTI 功能框图

从 EXTI 功能框图可以看到有两条主线，一条是由输入线到 NVIC 中断控制器，一条是由输入线到脉冲发生器。这就恰恰是 EXTI 的两大部分功能，产生中断与产生事件，两者从硬件上就存在不同。

下面让我们看一下 EXTI 功能框图的产生中断的线路，最终信号是流入 NVIC 控制器中。输入线是线路的信息输入端，它可以通过配置寄存器设置为任何一个 GPIO 口，或者是一些外设的事件。输入线一般都是存在电平变化的信号。

标号①是一个边沿检测电路，包括边沿检测电路，上升沿触发选择寄存器(EXTI_RTSR)和下降沿触发选择寄存器(EXTI_FTSR)。边沿检测电路以输入线作为信号输入端，如果检测到有边沿跳变就输出有效信号‘1’，就输出有效信号‘1’到标号②部分电路，否则输入无效信号‘0’。边沿跳变的标准在于开始的时候对于上升沿触发选择寄存器或下降沿触发选择寄存器对应位的设置，对应位的设置可以参照一下表 16.1.2.1。

标号②是一个或门电路，它的两个信号输入端分别是软件中断事件寄存器(EXTI_SWIER)和边沿检测电路的输入信号。或门电路只要输入端有信号‘1’，就会输出‘1’，所以就会输出‘1’到标号③电路和标号④电路。通过对软件中断事件寄存器的读写操作就可以启动中断/事件线，即相当于输出有效信号‘1’到或门电路输入端。

标号③是一个与门电路，它的两个信号输入端分别是中断屏蔽寄存器(EXTI_IMR)和标号②电路输出信号。与门电路要求输入都为‘1’才输出‘1’，这样子的情况下，如果中断屏蔽寄存器(EXTI_IMR)设置为 0 时，不管从标号②电路输出的信号特性如何，最终标号③电路输出的信号都是 0；假如中断屏蔽寄存器(EXTI_IMR)设置为 1 时，最终标号③电路输出的信号才由标号②电路输出信号决定，这样子就可以简单控制 EXTI_IMR 来实现中断的目的。标号④电路输出‘1’就会把请求挂起寄存器(EXTI_PR)对应位置 1。

最后，请求挂起寄存器(EXTI_PR)的内容就输出到 NVIC 内，实现系统中断事件的控制。

接下来我们看看 EXTI 功能框图的产生事件的线路。

产生事件线路是从标号 2 之后与中断线路有所不同，之前的线路都是共用的。标号④是一

个与门，输入端来自标号 2 电路以及来自于事件屏蔽寄存器(EXTI_EMR)。如果 EXTI_EMR 寄存器设置为 0，那不管标号 2 电路输出的信号是‘0’还是‘1’，最终标号 4 输出的是‘0’；如果 EXTI_EMR 寄存器设置为 1，最终标号④电路输出信号就由标号③电路输出的信号决定，这样子就可以简单的控制 EXTI_EMR 来实现是否产生事件的目的。

标号④电路输出有效信号 1 就会使脉冲发生器电路产生一个脉冲，而无效信号就不会使其产生脉冲信号。脉冲信号产生可以给其他外设电路使用，例如定时器，模拟数字转换器等，这样的脉冲信号一般用来触发 TIM 或者 ADC 开始转换。

产生中断线路目的使把输入信号输入到 NVIC，进一步运行中断服务函数，实现功能。而产生事件线路目的是传输一个脉冲信号给其他外设使用，属于硬件级功能。

EXTI 支持 19 个外部中断/事件请求，这些都是信息输入端，也就是上面提及到了输入线，具体如下：

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

从上面可以看出，STM32F407 供给 IO 口使用的中断线只有 16 个，但是 STM32F407 的 IO 口却远远不止 16 个，所以 STM32 把 GPIO 管脚 GPIOx.0~GPIOx.15(x=A,B,C,D,E,F,G,H,I)分别对应中断线 0~15。这样子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 和 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决定对应的中断线配置到哪个 GPIO 上了。

16.2 硬件设计

1. 例程功能

通过外部中断的方式让开发板上的三个独立按键控制 LED 灯：KEY0 控制 LED0 翻转，KEY1 控制 LED1 翻转，KEY2 控制 LED1 和 LED2 同时翻转。

2. 硬件资源

1）LED 灯
LED0 - PE0
LED1 - PE1
2）独立按键
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4

3. 原理图

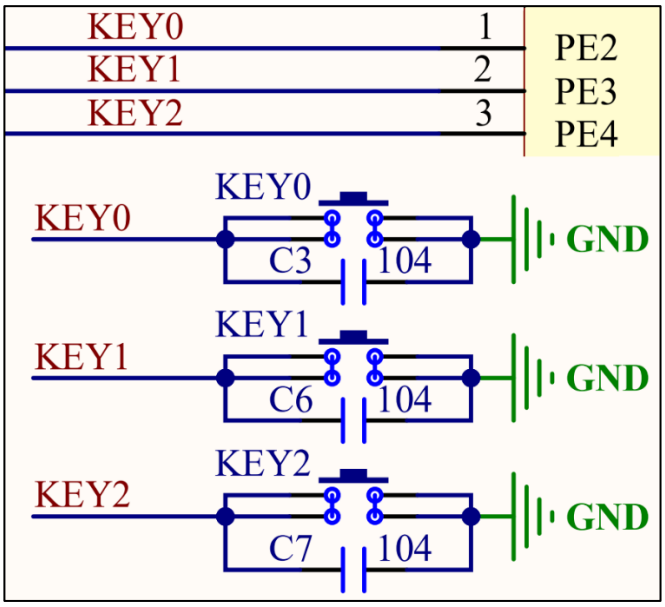


图 16.2.1 独立按键与 STM32F4 连接原理图

这里需要注意的是：KEY0、KEY1 和 KEY2 设计为采样到按键另一端的低电平为有效，并且外部都没有上下拉电阻，所以需要在 STM32F407 内部设置上下拉以设置空闲电平。

16.3 程序设计

16.3.1 EXTI 的 HAL 库驱动

前面讲解 HAL_GPIO_Init 函数的时候有提到过：HAL 库的 EXTI 外部中断的设置功能整合到 HAL_GPIO_Init 函数里面，而不是单独独立一个文件。所以我们的外部中断的初始化函数也是用 HAL_GPIO_Init 函数。

既然是要用到外部中断，所以我们的 GPIO 的模式要从下面的三个模式中选中一个：

```
#define  GPIO_MODE_IT_RISING           (0x10110000U)  /* 外部中断，上升沿触发检测 */
#define  GPIO_MODE_IT_FALLING          (0x10210000U)  /* 外部中断，下降沿触发检测 */
/* 外部中断，上升和下降双沿触发检测 */
#define  GPIO_MODE_IT_RISING_FALLING    (0x10310000U)
```

程序设计按键为按下触发中断，KEY0、KEY1 和 KEY2 是低电平有效的，所以我们要选择下降沿触发检测。

EXTI 外部中断配置步骤

1） 使能 IO 口时钟。

本实验用到的 GPIO 和按键输入实验是一样的，因此 GPIO 时钟使能也是一样的，请参考上一章代码。

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。

这些步骤 HAL 库全部封装在 HAL_GPIO_Init 函数里面，我们只需要设置好对应的参数，再调用 HAL_GPIO_Init 函数即可完成配置。

3）配置中断优先级（NVIC），并使能中断。

配置好 GPIO 模式以后，我们需要设置中断优先级和使能中断，中断优先级我们使用 HAL_NVIC_SetPriority 函数设置，中断使能我们使用 HAL_NVIC_EnableIRQ 函数设置。

4） 编写中断服务函数。

每开启一个中断，就必须编写其对应的中断服务函数，否则将导致死机（CPU 将找不到中断服务函数）。中断服务函数接口厂家已经在 startup_stm32f407xx.s 中写好了。STM32F4 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

HAL 库为了用户使用方便，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。

我们先来看一下 HAL_GPIO_EXTI_IRQHandler 函数定义：

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00U)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);      /* 清中断标志位 */
    HAL_GPIO_EXTI_Callback(GPIO_Pin);        /* 外部中断回调函数 */
  }
}
```

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。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。

因此我们可以在 HAL_GPIO_EXTI_Callback 里面实现控制逻辑编写，详见本实验源码。

16.3.2 程序流程图

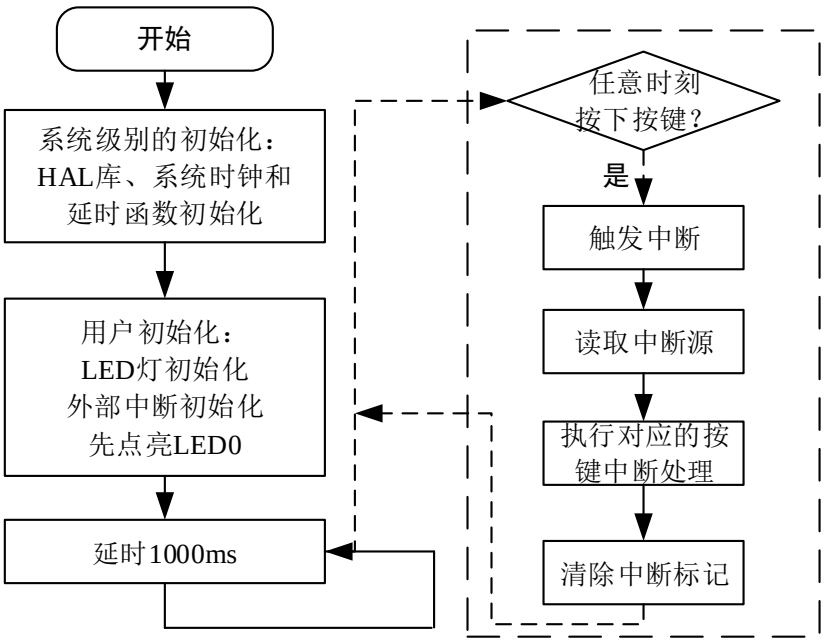


图 16.3.2.1 外部中断实验程序流程图

主程序初始外设，在按键初始化时初始化按键的采样边缘。

16.3.3 程序解析

1. 外部中断驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。外部中断（EXTI）驱动源码包括两个文件：exti.c 和 exti.h。

下面我们先解析 exti.h 的程序。

- **外部中断引脚定义**

由硬件设计小节，我们知道 KEY0、KEY1 和 KEY2 分别来连接到 PE2、PE3 和 PE4，我们做了下面的引脚定义。

```
/* 引脚 和 中断编号 & 中断服务函数 定义 */
#define KEY0_INT_GPIO_PORT          GPIOE
#define KEY0_INT_GPIO_PIN           GPIO_PIN_2
/* PE 口时钟使能 */
#define KEY0_INT_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
#define KEY0_INT_IRQn               EXTI2_IRQn
#define KEY0_INT_IRQHandler         EXTI2_IRQHandler

#define KEY1_INT_GPIO_PORT          GPIOE
#define KEY1_INT_GPIO_PIN           GPIO_PIN_3
/* PE 口时钟使能 */
#define KEY1_INT_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
#define KEY1_INT_IRQn               EXTI3_IRQn
#define KEY1_INT_IRQHandler         EXTI3_IRQHandler

#define KEY2_INT_GPIO_PORT          GPIOE
#define KEY2_INT_GPIO_PIN           GPIO_PIN_4
/* PE 口时钟使能 */
#define KEY2_INT_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
#define KEY2_INT_IRQn               EXTI4_IRQn
#define KEY2_INT_IRQHandler         EXTI4_IRQHandler
```

KEY0、KEY1 和 KEY2 分别连接 PE2、PE3 和 PE4，即对应了 EXTI2、EXTI3 和 EXTI4 这三条外部中断线。这里需要注意的是 EXTI0 到 EXTI4 都是有单独的中断向量，EXTI5 到 EXTI9 是公用 EXTI9_5_IRQn，EXTI10 到 EXTI15 是公用 EXTI15_10_IRQn。

下面我们再解析 exti.c 的程序，先看外部中断初始化函数，其定义如下：

```
/**
 * @brief       外部中断初始化程序
 * @param       无
 * @retval      无
 */
void extix_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    key_init();

    gpio_init_struct.Pin = KEY0_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;          /* 下降沿触发 */
    gpio_init_struct.Pull = GPIO_PULLUP;
    /* KEY0 配置为下降沿触发中断 */
    HAL_GPIO_Init(KEY0_INT_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = KEY1_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;          /* 下降沿触发 */
    gpio_init_struct.Pull = GPIO_PULLUP;
    /* KEY1 配置为下降沿触发中断 */
    HAL_GPIO_Init(KEY1_INT_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = KEY2_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_IT_FALLING;          /* 下降沿触发 */
    gpio_init_struct.Pull = GPIO_PULLUP;
```

```
    /* KEY2 配置为下降沿触发中断 */
    HAL_GPIO_Init(KEY2_INT_GPIO_PORT, &gpio_init_struct);

    HAL_NVIC_SetPriority(KEY0_INT_IRQn, 0, 2);              /* 抢占 0，子优先级 2 */
    HAL_NVIC_EnableIRQ(KEY0_INT_IRQn);                      /* 使能中断线 2 */

    HAL_NVIC_SetPriority(KEY1_INT_IRQn, 1, 2);              /* 抢占 1，子优先级 2 */
    HAL_NVIC_EnableIRQ(KEY1_INT_IRQn);                      /* 使能中断线 3 */

    HAL_NVIC_SetPriority(KEY2_INT_IRQn, 2, 2);              /* 抢占 2，子优先级 2 */
    HAL_NVIC_EnableIRQ(KEY2_INT_IRQn);                      /* 使能中断线 4 */
}
```

外部中断初始化函数主要做了两件事情，先是调用 IO 口初始化函数 HAL_GPIO_Init 来初始化 IO 口，然后设置中断优先级并使能中断线。

3 个外部中断服务函数，其定义如下：

```
/**
 * @brief       KEY0 外部中断服务程序
 * @param       无
 * @retval      无
 */
void KEY0_INT_IRQHandler(void)
{
    /* 调用中断处理公用函数 清除 KEY0 所在中断线 的中断标志位 */
    HAL_GPIO_EXTI_IRQHandler(KEY0_INT_GPIO_PIN);
    /* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
    __HAL_GPIO_EXTI_CLEAR_IT(KEY0_INT_GPIO_PIN);
}

/**
 * @brief       KEY1 外部中断服务程序
 * @param       无
 * @retval      无
 */
void KEY1_INT_IRQHandler(void)
{
    /* 调用中断处理公用函数 清除 KEY1 所在中断线 的中断标志位 */
    HAL_GPIO_EXTI_IRQHandler(KEY1_INT_GPIO_PIN);
    /* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
    __HAL_GPIO_EXTI_CLEAR_IT(KEY1_INT_GPIO_PIN);
}

/**
 * @brief       KEY2 外部中断服务程序
 * @param       无
 * @retval      无
 */
void KEY2_INT_IRQHandler(void)
{
    /* 调用中断处理公用函数 清除 KEY2 所在中断线 的中断标志位 */
    HAL_GPIO_EXTI_IRQHandler(KEY2_INT_GPIO_PIN);
    /* HAL 库默认先清中断再处理回调，退出时再清一次中断，避免按键抖动误触发 */
    __HAL_GPIO_EXTI_CLEAR_IT(KEY2_INT_GPIO_PIN);
}
```

所有的外部中断服务函数里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。但是它们的形参不同，我们的回调函数也是根据形参去判断是哪个 IO 口的外部中断线被触发。

外部中断回调函数，其定义如下：

```
/**
 * @brief       外部中断回调函数
```

```
 * @param       GPIO_Pin: 中断引脚号
 * @note        在 HAL 库中所有的外部中断服务函数都会调用此函数
 * @retval      无
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    /* 消抖，此处为了方便使用了延时函数，实际代码中禁止在中断服务函数中调用
    任何 delay 之类的延时函数！！！ */
    delay_ms(20);
    switch (GPIO_Pin)
    {
        case KEY0_INT_GPIO_PIN:
            if (KEY0 == 0)          /* KEY0U 中断 */
            {
                LED0_TOGGLE();      /* LED0 状态取反 */
            }
            break;

        case KEY1_INT_GPIO_PIN:
            if (KEY1 == 0)          /* KEY1 中断 */
            {
                LED1_TOGGLE();      /* LED1 状态取反 */
            }
            break;

        case KEY2_INT_GPIO_PIN:
            if (KEY2 == 0)          /* KEY2 中断 */
            {
                LED0_TOGGLE();      /* LED0 状态取反 */
                LED1_TOGGLE();      /* LED1 状态取反 */
            }
            break;

        default : break;
    }
}
```

外部中断回调函数 HAL_GPIO_EXTI_Callback 是用来编写真正的外部中断控制逻辑。该函数有一个形参就是 IO 引脚号。所以我们在该函数内部，一般通过判断 IO 引脚号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_2，那么一定是来自 PE2，因为中断线一次只能连接一个 IO 口，而三个 IO 口中引脚号为 2 的 IO 口只有 PE2，所以中断线 2 一定是连接 PE2，也就是外部中断由 PE2 触发。其他的引脚号的逻辑类似。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    led_init();                             /* 初始化 LED */
    extix_init();                           /* 初始化外部中断输入 */
    LED0(0);                                /* 点亮 LED0 */
    while(1)
    {
        delay_ms(1000);
    }
}
```

首先是调用系统级别的初始化：初始化 HAL 库、系统时钟和延时函数。接下来，调用 led_init 来初始化 LED 灯，调用 extix_init 函数初始化外部中断，点亮红灯。最后在无限循环里面执行

延时 1000ms 的重复动作。逻辑控制代码都在中断回调函数中完成。

16.4 下载验证

在下载好程序后，我们可以按 KEY0、KEY1、KEY2 来看看 LED 灯的变化，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。本章学习了 STM32F407 外部中断的使用方法。

第十七章 串口通信实验

本章我们将学习 STM32F407 的串口，教大家如何使用 STM32F407 的串口来发送和接收数据。本章将实现如下功能：STM32F407 通过串口和上位机的对话，STM32F407 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：

17.1 串口简介
17.2 硬件设计
17.3 程序设计
17.4 下载验证

17.1 串口简介

学习串口前，我们先来了解一下数据通信的一些基础概念。

17.1.1 数据通信的基础概念

在单片机的应用中，数据通信是必不可少的一部分，比如：单片机和上位机、单片机和外围器件之间，它们都有数据通信的需求。由于设备之间的电气特性、传输速率、可靠性要求各不相同，于是就有了各种通信类型、通信协议，我们最常见的有：USART、IIC、SPI、CAN、USB 等。下面，我们先来学习数据通信的一些基础概念。

1. 数据通信方式

按数据通信方式分类，可分为串行通信和并行通信两种。串行和并行的对比如下图所示：

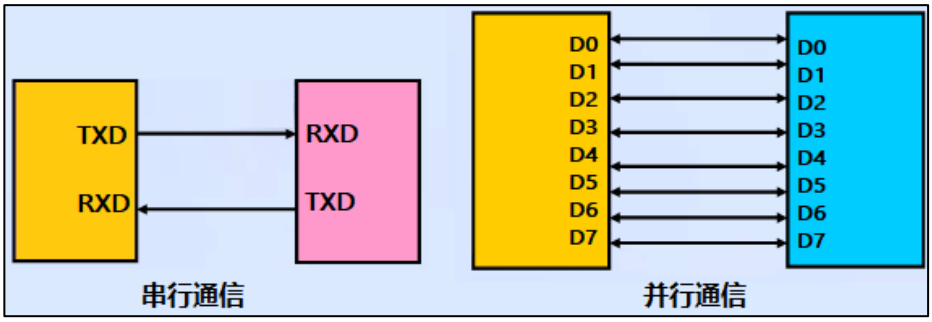


图 17.1.1.1 数据传输方式

串行通信的基本特征是数据逐位顺序依次传输，优点是传输线少、布线成本低、灵活度高等优点，一般用于近距离人机交互，特殊处理后也可以用于远距离，缺点就是传输速率低。

而并行通信是数据各位可以通过多条线同时传输，优点是传输速率高，缺点就是布线成本高，抗干扰能力差，适用于短距离、高速率的通信。

2. 数据传输方向

根据数据传输方向，通信又可分为全双工、半双工和单工通信。全双工、半双工和单工通信的比较如下图所示：

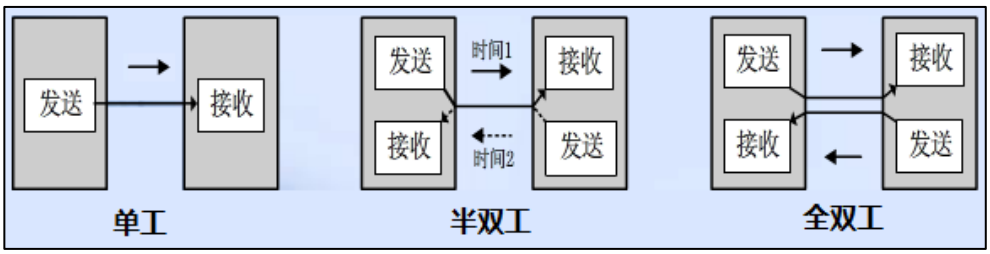


图 17.1.1.2 数据传输方式

单工是指数据传输仅能沿一个方向，不能实现反方向传输，如校园广播。

半双工是指数据传输可以沿着两个方向，但是需要分时进行，如对讲机。

全双工是指数据可以同时进行双向传输，日常的打电话属于这种情形。

这里注意全双工和半双工通信的区别：半双工通信是共用一条线路实现双向通信，而全双工是利用两条线路，一条用于发送数据，另一条用于接收数据。

3. 数据同步方式

根据数据同步方式，通信又可分为同步通信和异步通信。同步通信和异步通信比较如下图所示：

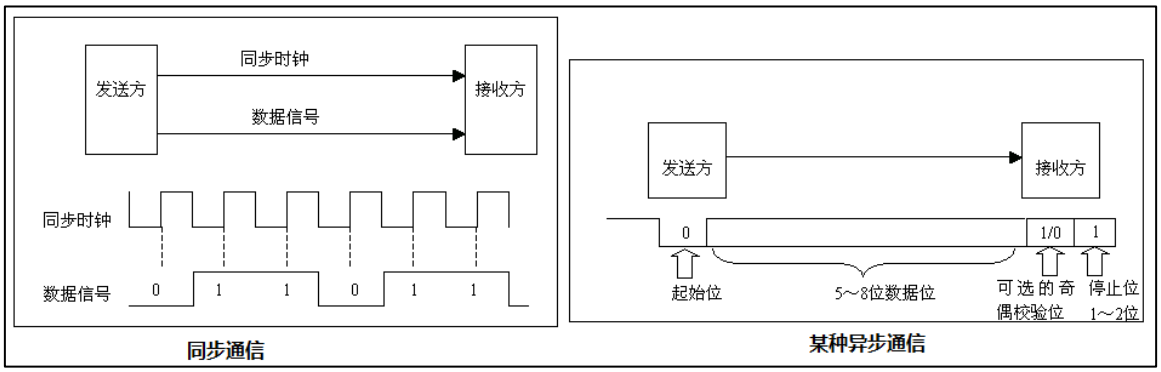


图 17.1.1.3 数据同步方式

同步通信要求通信双方共用同一时钟信号，在总线上保持统一的时序和周期完成信息传输。优点：可以实现高速率、大容量的数据传输，以及点对多点传输。缺点：要求发送时钟和接收时钟保持严格同步，收发双方时钟允许的误差较小，同时硬件复杂。

异步通信不需要时钟信号，而是在数据信号中加入开始位和停止位等一些同步信号，以便使接收端能够正确地将每一个字符接收下来，某些通信中还需要双方约定传输速率。优点：没有时钟信号硬件简单，双方时钟可允许一定误差。缺点：通信速率较低，只适用点对点传输。

4. 通信速率

在数字通信系统中，通信速率（传输速率）指数据在信道中传输的速度，它分为两种：传信率和传码率。

传信率：每秒钟传输的信息量，即每秒钟传输的二进制位数，单位为 bit/s（即比特每秒），因而又称为比特率。

传码率：每秒钟传输的码元个数，单位为 Baud（即波特每秒），因而又称为波特率。

比特率和波特率这两个概念又常常被人们混淆。比特率很好理解，我们来看看波特率，波特率被传输的是码元，码元是信号被调制后的概念，每个码元都可以表示一定 bit 的数据信息量。举个例子，在 TTL 电平标准的通信中，用 0V 表示逻辑 0，5V 表示逻辑 1，这时候这个码元就可以表示两种状态。如果电平信号 0V、2V、4V 和 6V 分别表示二进制数 00、01、10、11，这时候每一个码元就可以表示四种状态。

由上述可以看出，码元携带一定的比特信息，所以比特率和波特率也是有一定的关系的。

比特率和波特率的关系可以用以下式子表示：

$$比特率 = 波特率 * \log_2 M$$

其中 M 表示码元承载的信息量。我们也可以理解 M 为码元的进制数。

举个例子：波特率为 100 Baud，即每秒传输 100 个码元，如果码元采用十六进制编码（即 M=16，代入上述式子），那么这时候的比特率就是 400 bit/s。如果码元采用二进制编码（即 M=2，代入上述式子），那么这时候的比特率就是 100 bit/s。

可以看出采用二进制的时候，波特率和比特率数值上相等。但是这里要注意，它们的相等只是数值相等，其意义上不同，看波特率和波特率单位就知道。由于我们的所用的数字系统都是二进制的，所以有部分人久而久之就直接把波特率和比特率混淆了。

17.1.2 串口通信协议简介

串口通信是一种设备间常用的串行通信方式，串口按位（bit）发送和接收字节。尽管比特字节（byte）的串行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。**串口通信协议**是指规定了数据包的内容，内容包含了起始位、主体数据、校验位及停止位，双方需要约定一致的数据包格式才能正常收发数据的有关规范。在串口通信中，常用的协议包括 RS-232、RS-422 和 RS-485 等。

随着科技的发展，RS-232 在工业上还有广泛的使用，但是在商业技术上，已经慢慢的使用 USB 转串口取代了 RS-232 串口。我们只需要在电路中添加一个 USB 转串口芯片，就可以实现 USB 通信协议和标准 UART 串行通信协议的转换，而我们开发板上的 USB 转串口芯片是 CH340C 这个芯片。关于 USB 转串口芯片的原理图请看 17.2 小节。

下面我们来学习串口通信协议，这里主要学习串口通信的协议层。

串口通信的数据包由发送设备的 TXD 接口传输到接收设备的 RXD 接口。在串口通信的协议层中，规定了数据包的内容，它由起始位、主体数据、校验位以及停止位组成，通讯双方的数据包格式要约定一致才能正常收发数据，其组成如图 17.1.2.1 所示。

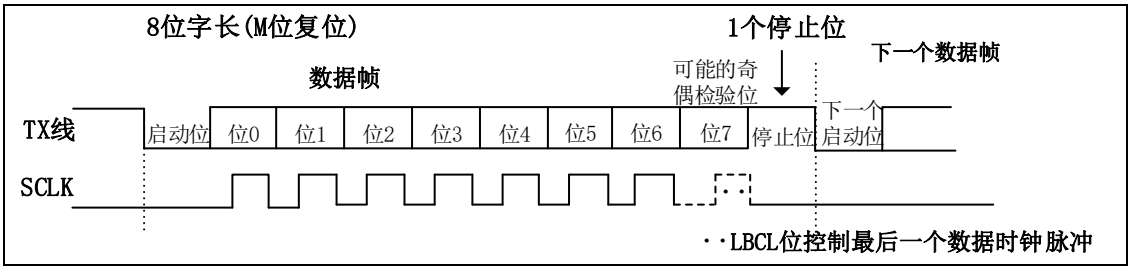


图 17.1.2.1 串口通信协议数据帧格式

串口通信协议数据包组成可以分为波特率和数据帧格式两部分。

1. 波特率

本章主要讲解的是串口异步通信，异步通信是不需要时钟信号的，但是这里需要我们约定好两个设备的波特率。波特率表示每秒钟传送的码元符号的个数，所以它决定了数据帧里面每一个位的时间长度。两个要通信的设备的波特率一定要设置相同，我们常见的波特率是 4800、9600、115200 等。

2. 数据帧格式

数据帧格式需要我们提前约定好，串口通信的数据帧包括起始位、停止位、有效数据位以及校验位。

- **起始位和停止位**

串口通信的一个数据帧是从起始位开始，直到停止位。数据帧中的起始位是由一个逻辑 0 的数据位表示，而数据帧的停止位可以是 0.5、1、1.5 或 2 个逻辑 1 的数据位表示，只要双方约定一致即可。

- **有效数据位**

数据帧的起始位之后，就接着是数据位，也称有效数据位，这就是我们真正需要的数据，有效数据位通常会被约定为 5、6、7 或者 8 个位长。有效数据位是低位（LSB）在前，高位（MSB）在后。

- **校验位**

校验位可以认为是一个特殊的数据位。校验位一般用来判断接收的数据位有无错误，检验方法有：奇检验、偶检验、0 检验、1 检验以及无检验。下面分别介绍一下：

奇校验是指有效数据为和校验位中“1”的个数为奇数，比如一个 8 位长的有效数据为：10101001，总共有 4 个“1”，为达到奇校验效果，校验位设置为“1”，最后传输的数据是 8 位的有效数据加上 1 位的校验位总共 9 位。

偶校验与奇校验要求刚好相反，要求帧数据和校验位中“1”的个数为偶数，比如数据帧：

11001010，此时数据帧“1”的个数为 4 个，所以偶校验位为“0”。

0 校验是指不管有效数据中的内容是什么，校验位总为“0”，**1 校验**是校验位总为“1”。

无校验是指数据帧中不包含校验位。

我们一般是使用无校验的情况。

17.1.3 STM32F407 的串口简介

STM32 的串口资源相当丰富，功能也相当强劲。STM32407IGT6 最多可提供 6 路串口，有分数波特率发生器、支持同步单线通信和半双工单线通讯、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范、具有 DMA 等。

STM32F407 的串口分为两种：USART（即通用同步异步收发器）和 UART（即通用异步收发器）。UART 是在 USART 基础上裁剪掉了同步通信功能，只剩下异步通信功能。简单区分同步和异步就是看通信时需不需要对外提供时钟输出，我们平时用串口通信基本都是异步通信。

STM32F407 有 4 个 USART 和 2 个 UART，其中 USART1 和 USART6 的时钟源来于 APB2 时钟，其最大频率为 84MHz，其他 4 个串口的时钟源可以来于 APB1 时钟，其最大频率为 42MHz。

STM32 的串口输出的是 TTL 电平信号，如果需要 RS-232 标准的信号可使用 MAX3232 芯片进行转换，而本实验我们是通过 USB 转串口芯片 CH340C 来与电脑的上位机进行通信。

17.1.3.1 USART 框图

下面先来学习如图 17.1.3.1.1 所示的 USART 框图，通过 USART 框图引出 USART 的相关知识，从而有了一个很好的整体掌握，对之后的编程也会有一个清晰的思路。

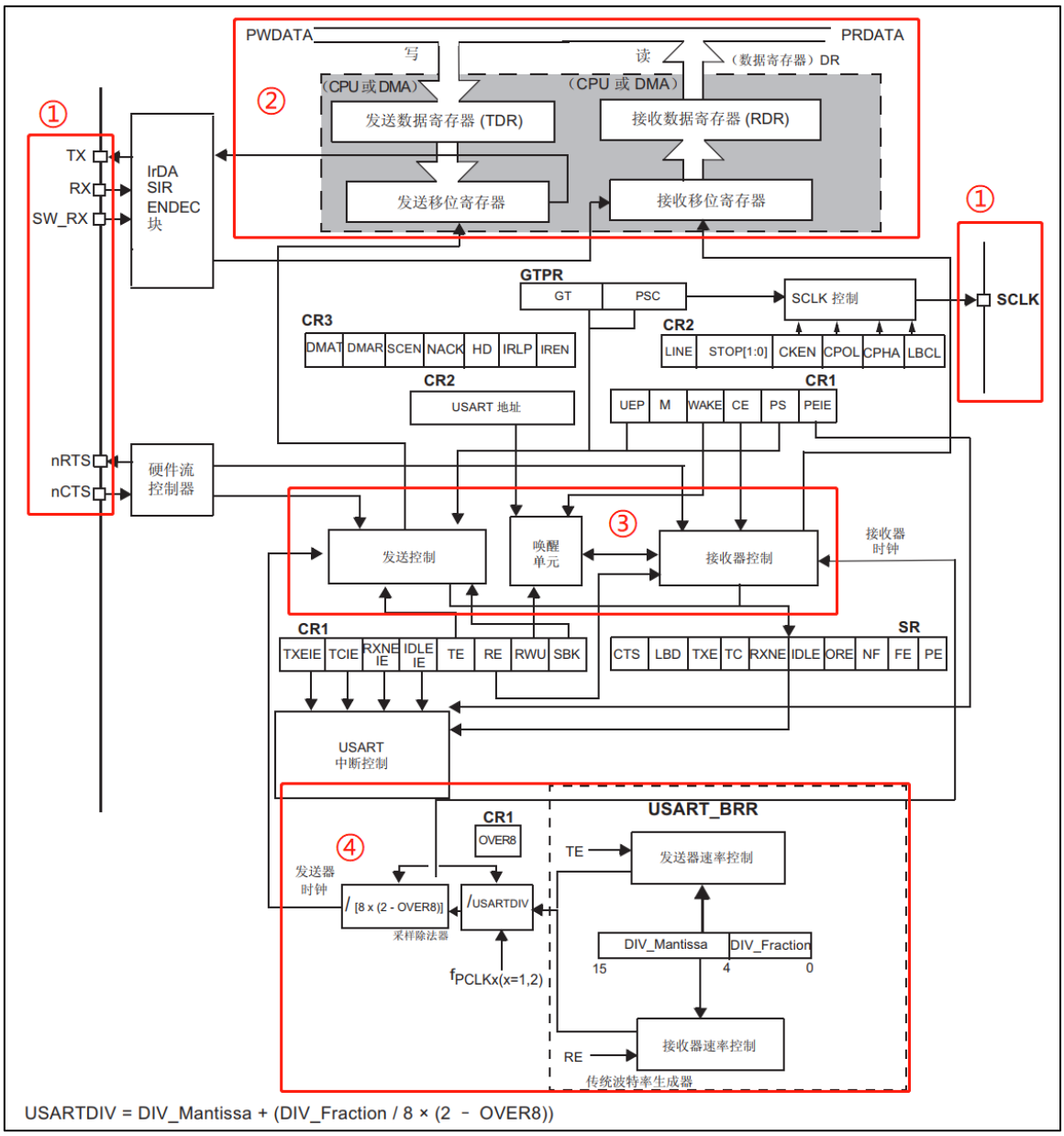


图 17.1.3.1.1 USART 框图

为了方便大家理解，我们把整个框图分成几个部分来介绍。

① USART 信号引脚

TX：发送数据输出引脚

RX：接收数据输入引脚

SCLK：发送器时钟输出，适用于同步传输

SW_RX：数据接收引脚，属于内部引脚，用于智能卡模式

IrDA_RDI：IrDA 模式下的数据输入

IrDA_TDO：IrDA 模式下的数据输出

nRTS：发送请求，若是低电平，表示 USART 准备好接收数据

nCTS：清除发送，若是高电平，在当前数据传输结束时阻断下一次的数据发送

② 数据寄存器

USART_DR 包含了已发送或接收到的数据。由于它本身就是两个寄存器组成的，一个专门给发送用的（TDR），一个专门给接收用的（RDR）,该寄存器具备读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。当进行数据发送操作时，往 USART_DR 中写入数据会自动存储在 TDR

内；当进行读取操作时，向 USART_DR 读取数据会自动提取 RDR 数据。

USART 数据寄存器（USART_DR）低 9 位数据有效，其他数据位保留。USART_DR 的第 9 位数据是否有效跟 USART_CR1 的 M 位设置有关，当 M 位为 0 表示 8 位数据字长；当 M 位为 1 时表示 9 位数据字长，一般使用 8 位数据字长。

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。

③ 控制器

USART 有专门控制发送的发送器，控制接收的接收器，还有唤醒单元、中断控制等等，具体在后面讲解 USART 寄存器的时候细讲。

④ 时钟与波特率

这部分的主要功能就是为 USART 提供时钟以及配置波特率。

波特率，即每秒钟传输的码元个数，在二进制系统中（串口的数据帧就是二进制的形式），波特率与波特率的数值相等，所以我们今后在把串口波特率理解为每秒钟传输的二进制位数。

波特率通过以下公式得出：

$$baud = \frac{fck}{16 * \text{USARTDIV}}$$

fck 是给串口的时钟（USART2\3 和 UART4\5 的时钟源为 PCLK1，USART1\6 的时钟源为 PCLK2），USARTDIV 是一个无符号的定点数，存放在波特率寄存器（USART_BRR）的低 16 位，DIV_Mantissa[11:0]存放的是 USARTDIV 的整数部分，DIV_Fractionp[3:0]存放的是 USARTDIV 的小数部分。

下面举个例子说明：

当串口 1 设置需要得到 115200 的波特率，fck = 84MHz，那么可得：

$$115200 = \frac{84000000}{16 * \text{USARTDIV}}$$

得到 USARTDIV ≈ 45.5729，分离 USARTDIV 的整数部分与小数部分，整数部分为 45，即 0x27，那么 DIV_Mantissa = 0x2D；小数部分为 0.5729，转化为十六进制即 0.5729*16 ≈ 9，所以 DIV_Fractionp = 0x9，USART_BRR 寄存器应该赋值为 0x2D9，成功设置波特率为 115200。

值得注意 USARTDIV 是允许有余数的，我们用四舍五入进行取整，这样会导致波特率会有所偏差，而这样的小误差是可以被允许的。

17.1.3.2 USART 寄存器

使用 STM32F407 的 USART 的配置步骤在《STM32F4xx 参考手册_V4（中文版）.pdf》中有列出，这里我们引用手册中的配置步骤：

1. 通过在 USART_CR1 寄存器上置位 UE 位来激活 USART。
2. 编程 USART_CR1 的 M 位来定义字长。
3. 在 USART_CR2 中编程停止位的位数。
4. 如果采用多缓冲器通信，配置 USART_CR3 中的 DMA 使能位(DMAT)。按多缓冲器通信中的描述配置 DMA 寄存器。
5. 利用 USART_BRR 寄存器选择要求的波特率。
6. 设置 USART_CR1 中的 TE 位，发送一个空闲帧作为第一次数据发送。
7. 把要发送的数据写进 USART_DR 寄存器(此动作清除 TXE 位)。在只有一个缓冲器的情况下，对每个待发送的数据重复步骤 7。
8. 在 USART_DR 寄存器中写入最后一个数据字后，要等待 TC=1，它表示最后一个数据帧的传输结束。当需要关闭 USART 或需要进入停机模式之前，需要确认传输结束，避免破坏最后一次传输。

我们按照上面的步骤配置就可以使用 STM32F407 的串口了：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。总结地来说，我们要学会配置 USART 对应的寄存器就可以使用串口功能了。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 STM32F407 的一个外设，其时钟由外设时钟使能寄存器控制，这里我们使用的串口 1 是在 APB2ENR 寄存器的第 14 位。

注意：除了串口 1 的时钟使能在 APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR 寄存器，而 APB2（84MHz）的频率一般是 APB1（42MHz）的一倍。

（2）串口复位

一般系统刚开始配置外设的时候，都会先执行复位该外设的操作，可以使外设的对应寄存器恢复到默认值，方便我们进行配置。串口 1 的复位就是通过配置 APB2RSTR 寄存器的第 4 位来实现的。APB2RSTR 寄存器的描述如图 17.1.3.2.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved													TIM11 RST	TIM10 RST	TIM9 RST
													rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reser-ved	SYSCFG RST	Reser-ved	SPI1 RST	SDIO RST	Reserved		ADC RST	Reserved		USART6 RST	USART1 RST	Reserved		TIM8 RST	TIM1 RST
	rw		rw	rw			rw			rw	rw			rw	rw

图 17.1.3.2.1 RCC_APB2RSTR 寄存器

（3）串口波特率设置

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。在前面的 USART 框图部分描述过，为了让大家更好了解波特率寄存器，下面截取 USART_BRR 寄存器图，如图 17.1.3.2.2 所示。

（3）串口波特率设置

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。在前面的 USART 框图部分描述过，为了让大家更好了解波特率寄存器，下面截取 USART_BRR 寄存器图，如图 17.1.3.2.2 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
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
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 15:4 **DIV_Mantissa[11:0]**：USARTDIV 的尾数
这 12 个位用于定义 USART 除数 (USARTDIV) 的尾数

位 3:0 **DIV_Fraction[3:0]**：USARTDIV 的小数
这 4 个位用于定义 USART 除数 (USARTDIV) 的小数。当 OVER8 = 1 时，不考虑 DIV_Fraction3 位，且必须将该位保持清零。

图 17.1.3.2.2 USART_BRR 寄存器

（4）串口控制

STM32F407 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。USART_CR1 寄存器的描述如图 17.1.3.2.3 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
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 17.1.3.2.3 USART_CR1 寄存器

该寄存器的高 16 位没有用到，低 16 位用于串口的功能设置。我们在这里只介绍需要用到的一些位，其他位可以参考《STM32F4xx 参考手册_V4（中文版）.pdf》。UE 为串口使能位，通过该位置 1，使能串口。M 为字长，当该位为 0 的时候设置串口为 8 个字长外加 n 个停止位，停止位的个数（n）是根据 USART_CR2 的[13:12]位设置来决定的，默认为 0。PCE 为校验使能位，设置为 0，即禁止校验，否则使能校验。PS 为校验位选择，设置为 0 为偶校验，否则奇校

验。TXIE 为发送缓冲区空中断使能位，设置该位为 1，当 USART_SR 中的 TXE 位为 1 时，将产生串口中断。TCIE 为发送完成中断使能位，设置该位为 1，当 USART_SR 中的 TC 位为 1 时，将产生串口中断。RXNEIE 为接收缓冲区非空中断使能，设置该位为 1，当 USART_SR 中的 ORE 或者 RXNE 位为 1 时，将产生串口中断。TE 为发送使能位，设置为 1，将开启串口的发送功能。RE 为接收使能位，用法同 TE。

（5）数据发送与接收

STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和 RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到数据的时候，也是存在该寄存器内。在前面的 USART 框图已经对 USART_DR 有详细的介绍，大家可以自行查阅。下面看一下寄存器的各位描述如图 17.1.3.2.4。

偏移地址：0x04

复位值：0xXXXX XXXX

位 31:9 保留，必须保持复位值

位 8:0 **DR[8:0]**：数据值

包含接收到数据字符或已发送的数据字符，具体取决于所执行的操作是“读取”操作还是“写入”操作。

因为数据寄存器包含两个寄存器，一个用于发送 (TDR)，一个用于接收 (RDR)，因此它具有双重功能（读和写）。

TDR 寄存器在内部总线和输出移位寄存器之间提供了并行接口（参见图 1）。

RDR 寄存器在输入移位寄存器和内部总线之间提供了并行接口。

在使能奇偶校验位的情况下（USART_CR1 寄存器中的 PCE 位被置 1）进行发送时，由于 MSB 的写入值（位 7 或位 8，具体取决于数据长度）会被奇偶校验位所取代，因此该值不起任何作用。

在使能奇偶校验位的情况下进行接收时，从 MSB 位中读取的值为接收到的奇偶校验位。

图 17.1.3.2.4 USART_DR 寄存器

（6）串口状态

串口状态通过状态寄存器 USART_SR 读取。USART_SR 的各位描述如图 17.1.3.2.5 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						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

图 17.1.3.2.5 USART_SR 寄存器

这里我们关注一下两个位，第 5、6 位 RXNE 和 TC。

RXNE（读数据寄存器非空），当该位被置 1 的时候，就是提示已经有数据被接收到了，并且可以读出来了。这时候我们要做的就是尽快去读取 USART_DR，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。

TC（发送完成），当该位被置位的时候，表示 USART_DR 内的数据已经被发送完成了。如果设置了这个位的中断，则会产生中断。该位也有两种清零方式：

1）读 USART_SR，写 USART_DR

2）直接向该位写 0

通过以上一些寄存器的操作再加上 IO 口的配置，我们就可以达到串口最基本的配置了，关于串口更详细的介绍，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》关于通用同步异步收发器这一章的相关知识。

17.1.4 GPIO 引脚复用功能

我们知道芯片有许多外设，而引脚的资源是很有限的，为了解决这个问题，方法就是引脚复用，这样使得引脚除了作为普通的 IO 口之外，还会与一些外设关联起来，作为第二功能使用，而且一个引脚不单单只有一种复用功能，而是拥有多个第二功能，但是一次只允许一个外设的复用功能，以确保共用同一个 IO 引脚的外设之间不会产生冲突。

下面我们把之前没讲解的复用功能寄存器 AFIO 讲解一下。

- **GPIO 复用功能寄存器（AFRH 和 AFRL）**

复用功能寄存器有 2 个，都是 32 位有效的寄存器，分高位（AFRH）和低位（AFRL）。复用器采用 16 路复用功能输入 AF0~AF15，通过 GPIOx_AFRL（引脚 0~7）、GPIOx_AFRH（引脚 8~15）寄存器对复用功能输入进行配置，每四位控制 1 路复用。

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

图 17.1.4.1 AFRL 寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[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
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 17.1.4.2 AFRH 寄存器

AFRL 寄存器配置引脚 0~7 复用功能，AFRH 寄存器配置引脚 8~15 复用功能。

IO 口并不是想复用什么功能都可以，是有规定的，每个 IO 引脚的复用可以通过查阅数据手册《STM32F407IGT6.pdf》，我们看看 Table 9. Alternate function mapping，其他的端口请自行查阅。

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11	I2C1/2/3	SPI1/SPI2/I2S2/I2S2ext	SPI3/I2Sext/I2S3	USART1/2/3/I2S3ext	UART4/5/USART6	CAN1/2 TIM12/13/14	OTG_FS/OTG_HS	ETH	FSMC/SDIO/OTG_FS	DCMI		
Port B	PB0	-	TIM1_CH2N	TIM3_CH3	TIM8_CH2N	-	-	-	-	-	-	OTG_HS_ULPI_D1	ETH_MII_RXD2	-	-	-	EVENTOUT
	PB1	-	TIM1_CH3N	TIM3_CH4	TIM8_CH3N		-	-	-	-	-	OTG_HS_ULPI_D2	ETH_MII_RXD3	-	-	-	EVENTOUT
	PB2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PB3	JTDO/TRACESWO	TIM2_CH2	-	-	-	SPI1_SCK	SPI3_SCK I2S3_CK	-	-	-	-	-	-	-	-	EVENTOUT
	PB4	NJTRST	-	TIM3_CH1		-	SPI1_MISO	SPI3_MISO	I2S3ext_SD	-	-	-	-	-	-	-	EVENTOUT
	PB5	-	-	TIM3_CH2		I2C1_SMBA	SPI1_MOSI	SPI3_MOSI I2S3_SD		-	CAN2_RX	OTG_HS_ULPI_D7	ETH_PPS_OUT	-	DCMI_D10	-	EVENTOUT
	PB6	-	-	TIM4_CH1		I2C1_SCL	-	-	USART1_TX	-	CAN2_TX	-	-	-	DCMI_D5	-	EVENTOUT
	PB7	-	-	TIM4_CH2		I2C1_SDA	-	-	USART1_RX	-	-	-	-	FSMC_NL	DCMI_VSYNC	-	EVENTOUT
	PB8	-	-	TIM4_CH3	TIM10_CH1	I2C1_SCL	-	-	-	-	CAN1_RX	-	ETH_MII_TXD3	SDIO_D4	DCMI_D6	-	EVENTOUT
	PB9	-	-	TIM4_CH4	TIM11_CH1	I2C1_SDA	SPI2_NSS I2S2_WS	-	-	-	CAN1_TX	-	-	SDIO_D5	DCMI_D7	-	EVENTOUT
	PB10	-	TIM2_CH3	-	-	I2C2_SCL	SPI2_SCK I2S2_CK	-	USART3_TX	-	-	OTG_HS_ULPI_D3	ETH_MII_RX_ER	-	-	-	EVENTOUT
	PB11	-	TIM2_CH4	-	-	I2C2_SDA	-	-	USART3_RX	-	-	OTG_HS_ULPI_D4	ETH_MII_TX_EN ETH_RMII_TX_EN	-	-	-	EVENTOUT
	PB12	-	TIM1_BKIN	-	-	I2C2_SMBA	SPI2_NSS I2S2_WS	-	USART3_CK	-	CAN2_RX	OTG_HS_ULPI_D5	ETH_MII_TXD0 ETH_RMII_TXD0	OTG_HS_ID	-	-	EVENTOUT
	PB13	-	TIM1_CH1N	-	-	-	SPI2_SCK I2S2_CK	-	USART3_CTS	-	CAN2_TX	OTG_HS_ULPI_D6	ETH_MII_TXD1 ETH_RMII_TXD1	-	-	-	EVENTOUT
	PB14	-	TIM1_CH2N	-	TIM8_CH2N	-	SPI2_MISO	I2S2ext_SD	USART3_RTS	-	TIM12_CH1	-	-	OTG_HS_DM	-	-	EVENTOUT
	PB15	RTC_REFIN	TIM1_CH3N	-	TIM8_CH3N	-	SPI2_MOSI I2S2_SD	-	-	-	TIM12_CH2	-	-	OTG_HS_DP	-	-	EVENTOUT

表 17.1.4.1 Port B 引脚复用

我们圈出来 PB6 和 PB7 对应 AF7 这列，因为我们的串口 1 就是用到这两个 IO 口，配置复用功能，使得 PB6 用做串口 1 的发送引脚 TX，PB7 则用做串口 1 的接收引脚 RX。

我们还需要学会在 HAL 库中寻找这些复用的宏定义，在 **stm32f4xx_hal_gpio_ex.h** 文件中可以找到。从表 17.1.4.1 我们知道 PB6 和 PB7 都是在复用器的 AF7 这一路中，所以我们在 HAL 库中就找 AF7 的宏定义，其定义如下：

```
/**
  * @brief   AF 7 selection
  */
#define GPIO_AF7_USART1  ((uint8_t)0x07) /* USART1 Alternate Function mapping */
#define GPIO_AF7_USART2  ((uint8_t)0x07) /* USART2 Alternate Function mapping */
#define GPIO_AF7_USART3  ((uint8_t)0x07) /* USART3 Alternate Function mapping */
#define GPIO_AF7_I2S3ext ((uint8_t)0x07)
```

```
/* I2S3ext_SD Alternate Function mapping */
```

细心的朋友可以看出，这些宏定义的值都是一样的，即都是(uint8_t)0x07，而宏名只是为了区分是哪个外设而已。因为我们的外设是串口 1，所以就很容易选择到我们复用的功能要用到的宏定义，就是 GPIO_AF7_USART1。具体的场景应用请看我们的串口 1 的初始化源码。

17.2 硬件设计

1. 例程功能

LED0 闪烁，提示程序在运行。STM32 通过串口 1 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，会返回给上位机。同时每隔一定时间，通过串口 1 输出一段信息到电脑。

2. 硬件资源

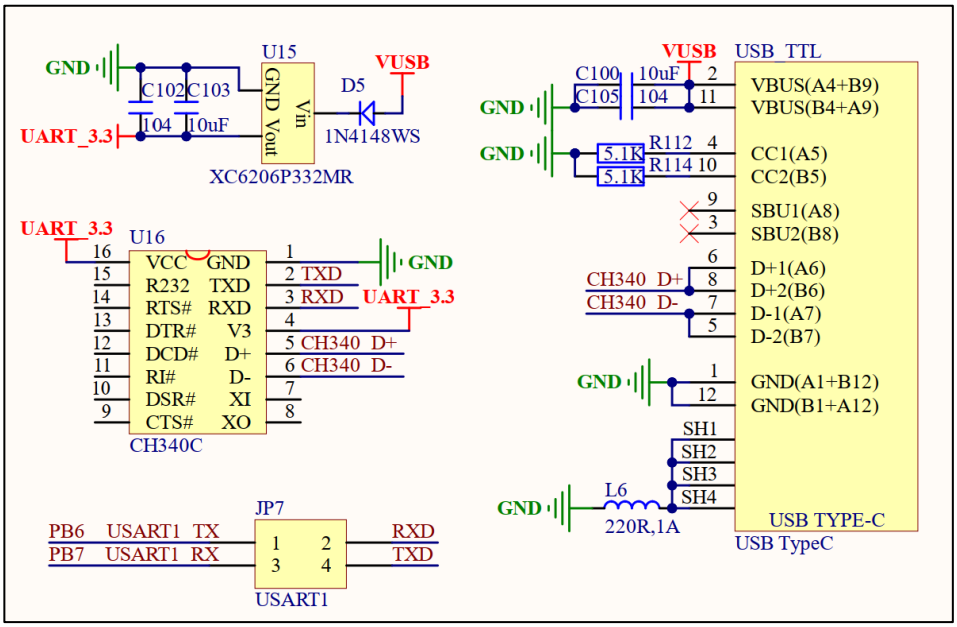
1）LED 灯

LED0 - PF0

2）串口 1 (PB6/PB7 连接在板载 USB 转串口芯片 CH340C 上面) ，需要跳线帽连接。

3. 原理图

USB 转串口硬件部分的原理图，如图 17.2.1 所示：



17.2.1 USB 转串口原理图

这里需要注意的是：这里我们要把 JP7 的 RXD 和 PB6 用跳线帽连接，以及 TXD 和 PB7 也用跳线帽连接，否则串口的信号线跟 USB 接口是没有连接的。

17.3 程序设计

17.3.1 USART 的 HAL 库驱动

HAL 库中关于串口的驱动程序比较多，我们主要先来学习本章需要用到的，其余的后续用到再讲解。因为我们现在只是用到异步收发器功能，所以我们现在只需要 stm32f4xx_hal_uart.c 文件（及其头文件）的驱动代码，stm32f4xx_hal_usart.c 是通用同步异步收发器，暂时没有用到，可以暂时不看。用到一个外设第一个函数是其初始化函数。

1. HAL_UART_Init 函数

要使用一个外设首先要对它进行初始化，所以先看串口的初始化函数，其声明如下：

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

- **函数描述：**
  用于初始化异步模式的收发器。
- **函数形参：**
  形参 1 是串口的句柄，UART_HandleTypeDef 结构体类型，其定义如下：

```
typedef struct
{
  USART_TypeDef                 *Instance;        /* UART 寄存器基地址 */
  UART_InitTypeDef              Init;             /* UART 通信参数 */
  UART_AdvFeatureInitTypeDef    AdvancedInit;     /* UART 高级功能配置结构体 */
  uint8_t                       *pTxBuffPtr;      /* 指向 UART 发送缓冲区 */
  uint16_t                      TxXferSize;       /* UART 发送数据的大小 */
  __IO uint16_t                 TxXferCount;      /* UART 发送数据的个数 */
  uint8_t                       *pRxBuffPtr;      /* 指向 UART 接收缓冲区 */
  uint16_t                      RxXferSize;       /* UART 接收数据大小 */
  __IO uint16_t                 RxXferCount;      /* UART 接收数据的个数 */
  uint16_t                      Mask;             /*  UART 数据接收寄存器掩码 */
  DMA_HandleTypeDef             *hdmatx;          /* UART 发送参数设置（DMA） */
  DMA_HandleTypeDef             *hdmarx;          /* UART 接收参数设置（DMA） */
  HAL_LockTypeDef               Lock;             /* 锁定对象 */
  __IO HAL_UART_StateTypeDef    gState;           /* UART 发送状态结构体 */
  __IO HAL_UART_StateTypeDef    RxState;          /* UART 接收状态结构体 */
  __IO uint32_t                 ErrorCode;        /* UART 操作错误信息 */
}UART_HandleTypeDef;
```

1）Instance：指向 UART 寄存器基地址。实际上这个基地址 HAL 库已经定义好了，可以选择范围：USART1~USART3、USART6、UART4、UART5、UART7、UART8。

2）Init：UART 初始化结构体，用于配置通讯参数，如波特率、数据位数、停止位等等。下面我们再详细讲解这个结构体。

3）AdvancedInit：用于配置高级功能，如自动波特率，MSB 先行等。

4）pTxBuffPtr，TxXferSize，TxXferCount：分别是指向发送数据缓冲区的指针，发送数据的大小，发送数据的个数。

5）pRxBuffPtr，RxXferSize，RxXferCount：分别是指向接收数据缓冲区的指针，接受数据的大小，接收数据的个数。

6）Mask：UART 数据接收寄存器的掩码，用于存放数据的校验位。

7）hdmatx，hdmarx：配置串口发送接收数据的 DMA 具体参数。

8）Lock：对资源操作增加操作锁保护，可选 HAL_UNLOCKED 或者 HAL_LOCKED 两个参数。如果 gState 的值等于 HAL_UART_STATE_RESET，则认为串口未被初始化，此时，分配锁资源，并且调用 HAL_UART_MspInit 函数来对串口的 GPIO 和时钟进行初始化。

9）gState，RxState：分别是 UART 的发送状态、工作状态的结构体和 UART 接受状态的结构体。HAL_UART_StateTypeDef 是一个枚举类型，列出串口在工作过程中的状态值，有些值只适用于 gState，如 HAL_UART_STATE_BUSY。

10）ErrorCode：串口错误操作信息。主要用于存放串口操作的错误信息。

下面，我们来了解 UART_InitTypeDef 这个结构体类型，该结构体用于配置 UART 的各个通信参数，包括波特率，停止位等，具体说明如下：

```
typedef struct
{
  uint32_t BaudRate;                  /* 波特率 */
  uint32_t WordLength;                /* 字长 */
  uint32_t StopBits;                  /* 停止位 */
  uint32_t Parity;                    /* 校验位 */
  uint32_t Mode;                      /* UART 模式 */
  uint32_t HwFlowCtl;                 /* 硬件流设置 */
  uint32_t OverSampling;              /* 过采样设置 */
}UART_InitTypeDef
```

1）BaudRate：波特率设置。一般设置为 2400、9600、19200、115200。

2）WordLength：数据帧字长，可选 8 位或 9 位。这里我们设置为 8 位字长数据格式。
3）StopBits：停止位设置，可选 0.5 个、1 个、1.5 个和 2 个停止位，一般我们选择 1 个停止位。
4）Parity：奇偶校验控制选择，我们设定为无奇偶校验位。
5）Mode：UART 模式选择，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。
6）HwFlowCtl：硬件流控制选择，我们设置为无硬件流控制。
7）OverSampling：过采样选择，选择 8 倍过采样或者 16 过采样，一般选择 16 过采样。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值，有 4 个，分别是 HAL_OK 表示成功，HAL_ERROR 表示错误，HAL_BUSY 表示忙碌，HAL_TIMEOUT 超时。后续遇到该结构体也是一样的。

2. HAL_UART_Receive_IT 函数

HAL_UART_Receive_IT 函数是开启串口接收中断函数。其声明如下：

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                      uint8_t *pData, uint16_t Size);
```

- **函数描述：**

用于开启以中断的方式接收指定字节。数据接收在中断处理函数里面实现。

- **函数形参：**

形参 1 是 UART_HandleTypeDef 结构体指针类型的串口句柄。
形参 2 是要接收的数据地址。
形参 3 是要接收的数据大小，以字节为单位。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

3. HAL_UART_IRQHandler 函数

HAL_UART_IRQHandler 函数是 HAL 库中断处理公共函数。其声明如下：

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

- **函数描述：**

该函数是 HAL 库中断处理公共函数，在串口中断服务函数中被调用。

- **函数形参：**

形参 1 是 UART_HandleTypeDef 结构体指针类型的串口句柄。

- **函数返回值：**

无

- **注意事项：**

该函数是 HAL 库已经定义好，用户一般不能随意修改。如果用户要在中断中实现自己的逻辑代码，可以直接在函数 HAL_UART_IRQHandler 的前面或者后面添加新代码，也可以直接在 HAL_UART_IRQHandler 调用的各种回调函数里面执行，这些回调都是弱定义的，方便用户直接在其它文件里面重定义。串口回调函数主要有下面几个：

```
__weak void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_AbortCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_AbortTransmitCpltCallback(UART_HandleTypeDef *huart)
__weak void HAL_UART_AbortReceiveCpltCallback(UART_HandleTypeDef *huart)
```

本实验我们用到的是接收回调函数 HAL_UART_RxCpltCallback，就是在接收回调函数里面编写我们的接收逻辑代码，具体请参考实验源码。

串口通信配置步骤

1) 串口参数初始化（波特率、字长、奇偶校验等），并使能串口。

HAL 库通过调用串口初始化函数 HAL_UART_Init 完成对串口参数初始化，详见例程源码。

注意：该函数会调用：HAL_UART_MspInit 函数来完成对串口底层的初始化，包括：串口及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2）使能串口和 GPIO 口时钟

本实验用到 USART1 串口，使用 PB6 和 PB7 作为串口的 TX 和 RX 脚，因此需要先使能 USART1 和 GPIOB 时钟。参考代码如下：

```
__HAL_RCC_USART1_CLK_ENABLE();            /* 使能 USART1 时钟 */
__HAL_RCC_GPIOB_CLK_ENABLE();             /* 使能 GPIOB 时钟 */
```

3）GPIO 模式设置（速度、上下拉、复用功能等）

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

4）开启串口相关中断，配置串口中断优先级

本实验我们使用串口中断来接收数据。我们使用 HAL_UART_Receive_IT 函数开启串口中断接收，并设置接收 buffer 及其长度。通过 HAL_NVIC_EnableIRQ 函数使能串口中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5）编写中断服务函数

串口 1 中断服务函数为：USART1_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库为了使用方便，提供了一个串口中断通用处理函数 HAL_UART_IRQHandler，该函数在串口接收完数据后，又会调用回调函数 HAL_UART_RxCpltCallback，用于给用户处理串口接收到的数据。

因此我们需要在 HAL_UART_RxCpltCallback 函数实现数据接收处理，详见本例程源码。

6）串口数据接收和发送

最后我们可以通过读写 USART_DR 寄存器，完成串口数据的接收和发送，HAL 库也给我们提供了：HAL_UART_Receive 和 HAL_UART_Transmit 两个函数用于串口数据的接收和发送。

大家可以根据实际情况选择使用那种方式来收发串口数据。

17.3.2 程序流程图

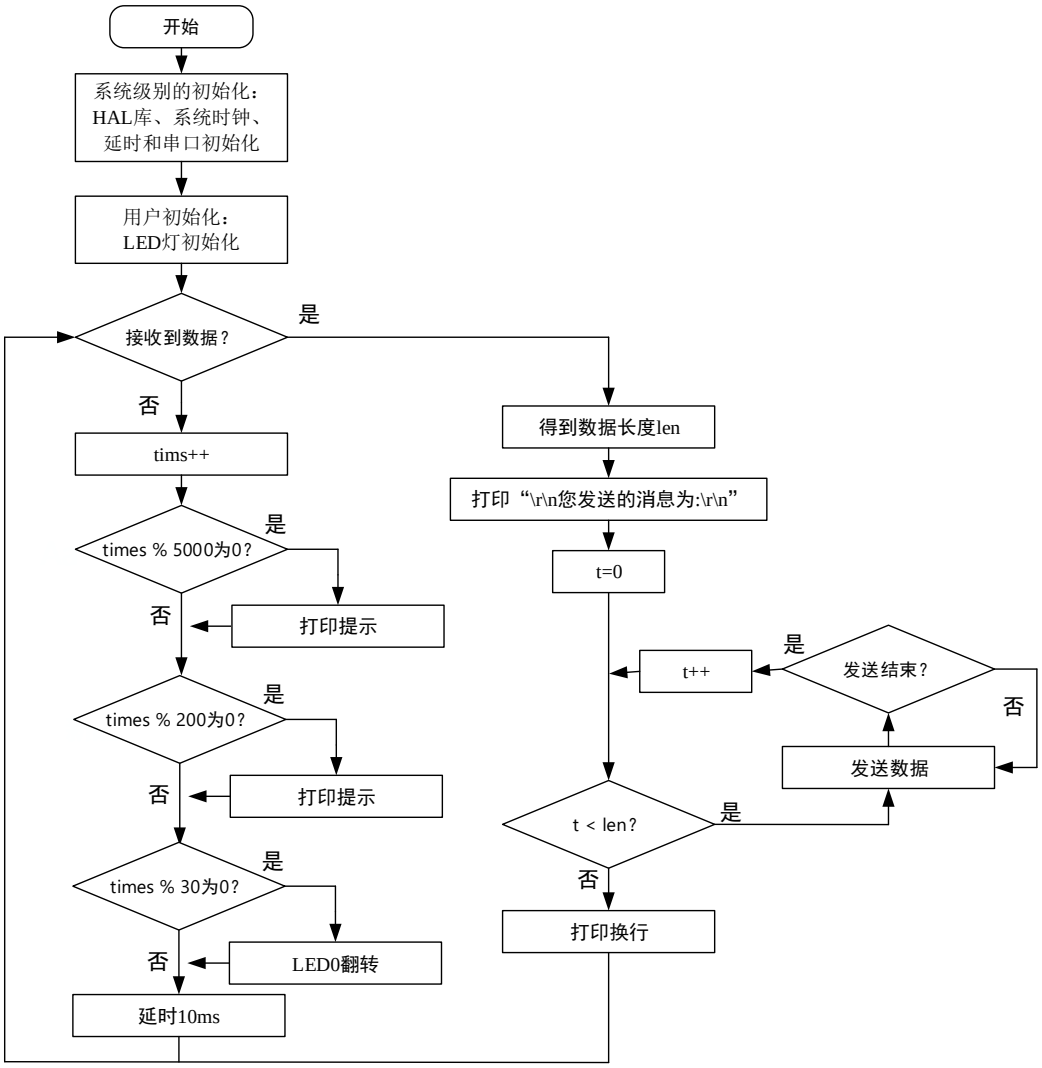


图 17.3.2.1 串口通信实验程序流程图

17.3.3 程序解析

1. 串口 1 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。串口 1（USART1）驱动源码包括两个文件：usart.c 和 usart.h。

下面我们先解析 usart.h 的程序。

- **串口 1 引脚定义**

由硬件设计小节，我们知道 PB6 和 PB7 分别被复用为串口 1 的发送和接收引脚，我们做了下面的引脚定义。

```
/* 串口 1 的 GPIO */
#define USART_TX_GPIO_PORT                GPIOB
#define USART_TX_GPIO_PIN                 GPIO_PIN_6
#define USART_TX_GPIO_AF                  GPIO_AF7_USART1
/* 发送引脚时钟使能 */
#define USART_TX_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define USART_RX_GPIO_PORT                GPIOB
#define USART_RX_GPIO_PIN                 GPIO_PIN_7
#define USART_RX_GPIO_AF                  GPIO_AF7_USART1
```

```
/* 接收引脚时钟使能 */
#define USART_RX_GPIO_CLK_ENABLE()   do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define USART_UX                     USART1
#define USART_UX_IRQn                USART1_IRQn
#define USART_UX_IRQHandler          USART1_IRQHandler
/* USART1 时钟使能 */
#define USART_UX_CLK_ENABLE()      do{ __HAL_RCC_USART1_CLK_ENABLE(); }while(0)
```

USART1_IRQn 也就是我们中断向量表的 37 号中断，USART1_IRQHandler 是串口 1 的中断服务函数。每个串口都有自己的中断函数，但是我们最终都是通过回调函数去实现逻辑代码，当然我们亦可在中断函数里实现逻辑代码。

另外我们还定义了三个宏，具体如下：

```
#define USART_REC_LEN   200         /* 定义最大接收字节数 200 */
#define USART_EN_RX     1           /* 使能（1）/禁止（0）串口 1 接收 */
#define RXBUFFERSIZE    1           /* 缓存大小 */
```

可以看到 USART_REC_LEN 表示最大接收字节数，这里定义的是 200 个字节，后续如果有需求要发送更大的数据包，可以改大这个值，这里不改太大，是避免浪费太多内存。USART_EN_RX 则是用于使能串口 1 的接收数据。RXBUFFERSIZE 是缓冲大小。

下面我们再解析 usart.c 的程序，先看串口 1 的初始化函数，其定义如下：

```
/**
 * @brief       串口 X 初始化函数
 * @param       baudrate: 波特率，根据自己需要设置波特率值
 * @note        注意：必须设置正确的时钟源，否则串口波特率就会设置异常.
 *              这里的 USART 的时钟源在 sys_stm32_clock_init()函数中已经设置过了.
 * @retval      无
 */
void usart_init(uint32_t baudrate)
{
    uartx_handle.Instance = USART_UX;                    /* USART1 */
    uartx_handle.Init.BaudRate = baudrate;               /* 波特率 */
    uartx_handle.Init.WordLength = UART_WORDLENGTH_8B;   /* 字长为 8 位数据格式 */
    uartx_handle.Init.StopBits = UART_STOPBITS_1;        /* 一个停止位 */
    uartx_handle.Init.Parity = UART_PARITY_NONE;         /* 无奇偶校验位 */
    uartx_handle.Init.HwFlowCtl = UART_HWCONTROL_NONE;   /* 无硬件流控 */
    uartx_handle.Init.Mode = UART_MODE_TX_RX;            /* 收发模式 */
    HAL_UART_Init(&uartx_handle);             /* HAL_UART_Init()会使能 UART1 */

    /*该函数会开启接收中断：标志位 UART_IT_RXNE，并且设置接收缓冲以及接收缓冲接收最大数据量*/
    HAL_UART_Receive_IT(&uartx_handle, (uint8_t *)aRxbuffer, RXBUFFERSIZE);
}
```

uartx_handle 是结构体 UART_HandleTypeDef 类型的全局变量，UART_HandleTypeDef 结构体成员的含义请回到前面回顾。波特率我们直接赋值给 uartx_handle.Init.BaudRate 这个成员，可以看出很方便。需要注意的是，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。

上面的初始化函数只是串口初始化的其中一部分，我们还有一部分初始化需要 HAL_UART_MspInit 函数去完成。HAL_UART_MspInit 是 HAL 库定义的弱定义函数，这里我们做重定义以实现我们的初始化需求。HAL_UART_MspInit 函数在 HAL_UART_Init 函数中会被调用，其定义如下：

```
/**
 * @brief       UART 底层初始化函数
 * @param       huart: UART 句柄类型指针
 * @note        此函数会被 HAL_UART_Init()调用
 *              完成时钟使能，引脚配置，中断配置
 * @retval      无
 */
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
```

```
{
    GPIO_InitTypeDef gpio_init_struct;
    if(huart->Instance == USART_UX)                    /* 如果是串口 1，进行串口 1 MSP 初始化 */
    {
        USART_UX_CLK_ENABLE();                         /* USART1 时钟使能 */
        USART_TX_GPIO_CLK_ENABLE();                    /* 发送引脚时钟使能 */
        USART_RX_GPIO_CLK_ENABLE();                    /* 接收引脚时钟使能 */

        gpio_init_struct.Pin = USART_TX_GPIO_PIN;                /* TX 引脚 */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                 /* 复用推挽输出 */
        gpio_init_struct.Pull = GPIO_PULLUP;                     /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;           /* 高速 */
        gpio_init_struct.Alternate = USART_TX_GPIO_AF;           /* 复用为 USART1 */
        HAL_GPIO_Init(USART_TX_GPIO_PORT, &gpio_init_struct);    /* 初始化发送引脚 */

        gpio_init_struct.Pin = USART_RX_GPIO_PIN;                /* RX 引脚 */
        gpio_init_struct.Alternate = USART_RX_GPIO_AF;           /* 复用为 USART1 */
        HAL_GPIO_Init(USART_RX_GPIO_PORT, &gpio_init_struct);    /* 初始化接收引脚 */

#if USART_EN_RX
        HAL_NVIC_EnableIRQ(USART_UX_IRQn);               /* 使能 USART1 中断通道 */
        HAL_NVIC_SetPriority(USART_UX_IRQn, 3, 3);       /* 抢占优先级 3，子优先级 3 */
#endif
    }
}
```

怎么来理解这个函数呢？该函数主要实现底层的初始化，事实上这个函数的代码还可以直接放到 usart_init 函数里面，但是 HAL 库为了代码的功能分层初始化，定义这个函数方便用户使用。所以我们也按照 HAL 库的这个结构来初始化外设。这个函数首先是调用 if(huart->Instance == USART_UX)判断是要初始化那个串口，因为每个串口初始化都会调用 HAL_UART_MspInit 这个函数，所以需要判断是哪个串口要初始化才做相应的处理。只能说 HAL 库这样的结构机制有好处，自然也有坏处。

首先就是使能串口以及 PB6 和 PB7 的时钟，PB6 和 PB7 需要用做复用功能，复用功能模式有两个选择：GPIO_MODE_AF_PP 推挽式复用和 GPIO_MODE_AF_OD 开漏式复用，我们选择的是推挽式复用，因为 PB6 是一个发送管脚，所以复用模式设置为 USART_TX_GPIO_AF，而 PB7 是一个接收管脚，所以它的复用模式设置为 USART_RX_GPIO_AF。然后就是调用 HAL_GPIO_Init 函数进行 IO 口的初始化。

最后因为我们用到串口中断，所以还需要中断相关的配置。HAL_NVIC_EnableIRQ 函数使能串口 1 复用通道。HAL_NVIC_SetPriority 函数配置串口中断的抢占优先级以及响应优先级。

串口初始化由上述两个函数完成，下面就该讲到串口中断服务函数了，其定义如下：

```
/**
 * @brief       串口 X 中断服务函数
 *              注意,读取 USARTx->SR 能避免莫名其妙的错误
 * @param       无
 * @retval      无
 */
void USART_UX_IRQHandler(void)
{
#if SYSTEM_SUPPORT_OS                                    /*使用 OS*/
    OSIntEnter();
#endif
    HAL_UART_IRQHandler(&uartx_handler);                 /*调用 HAL 库中断处理公用函数*/
    while (HAL_UART_Receive_IT(&uartx_handler, (uint8_t *)aRxBuffer,
      RXBUFFERSIZE) != HAL_OK) /*一次处理完成之后，重新开启中断并设置 RxXferCount 为 1*/
    {
        /* 如果出错会卡死在这里 */
    }
#if SYSTEM_SUPPORT_OS                                    /*使用 OS*/
    OSIntExit();
```

```
#endif
}
```

从代码逻辑可以看出，在中断服务函数内部通过调用 HAL_UART_GetState 函数获取串口状态，计数处理时间是否超时，然后完成一次传输后，调用 UART_Receive_IT 函数重新开启中断。UART_Receive_IT 函数的作用就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，同时调用接收回调函数 HAL_UART_RxCpltCallback 进行处理。HAL_MAX_DELAY 最大延时时间在 stm32f4xx_hal_def.h 中定义。

下面列出串口接收中断的一般流程，如图 17.3.3.1 所示：

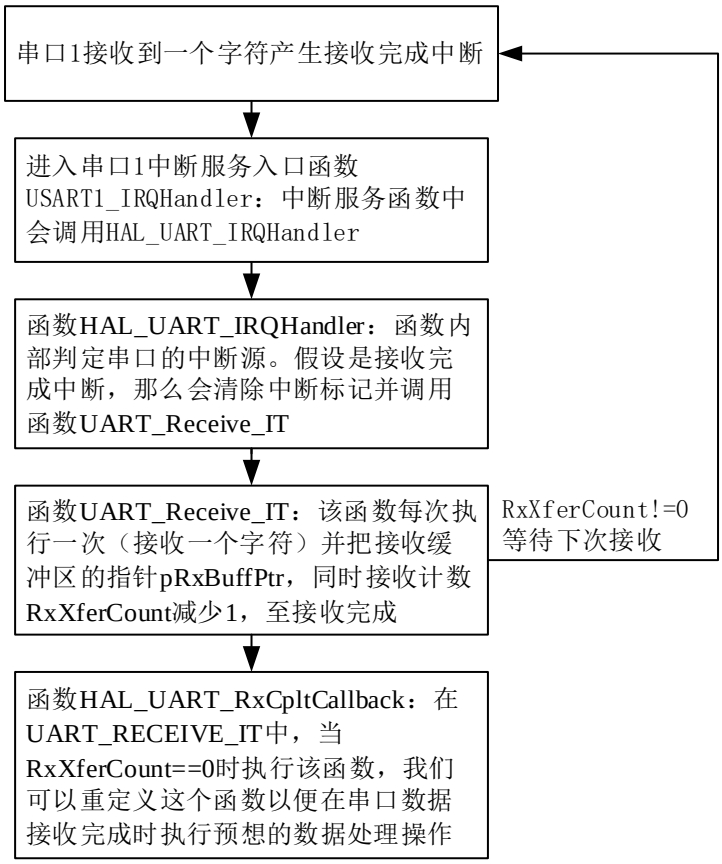


图 17.3.3.1 串口接收中断执行流程图

串口接收回调函数定义如下：

```
/**
 * @brief       UART 数据接收回调接口
                数据处理在这里进行
 * @param       huart:串口句柄
 * @retval      无
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance == USART_UX)                     /* 如果是串口 1 */
    {
        if((g_usart_rx_sta & 0x8000) == 0)              /* 接收未完成 */
        {
            if(g_usart_rx_sta & 0x4000)                 /* 接收到了 0x0d（即回车键）*/
            {
                if(aRxBuffer[0] != 0x0A)                /* 接收到的不是 0x0A（即不是换行键）*/
                {
                    g_usart_rx_sta = 0;                 /*接收错误,重新开始 */
                }
                else                                    /* 接收到的是 0x0a（即换行键） */
```

```
            {
                g_usart_rx_sta |= 0x8000;       /* 接收完成了 */
            }
        }
        else                                    /* 还没收到 0x0D（即回车键） */
        {
            if (aRxBuffer[0] == 0x0D)
                g_usart_rx_sta |= 0x4000;
            else
            {
                g_usart_rx_buf[g_usart_rx_sta & 0x3FFF] = aRxBuffer[0];
                g_usart_rx_sta++;
                if (g_usart_rx_sta > (USART_REC_LEN - 1))
                {
                    g_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，那么每当串口 1 接收到一个字符后触发接收完成中断，便会在中断服务函数中引导执行该回调函数。当串口接受到一个字符后，它会保存在缓存 g_rx_buffer 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 g_usart_rx_buf，一个接收状态寄存器 g_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。数组 g_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。g_usart_rx_sta 是一个接收状态寄存器其各的定义如表 17.3.3.1 所示：

g_usart_rx_sta		
bit15	bit14	bit13~0
接收完成标志	接收到 0x0D 标志	接收到的有效字节个数

表 15.3.3.1 接收状态寄存器位定义表

设计思路如下：

当接收到从电脑发过来的数据，把接收到的数据保存在数组 g_usart_rx_buf 中，同时在接收状态寄存器（g_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 g_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。

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。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t len;
    uint16_t times = 0;
```

```
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */

    while(1)
    {
       if (g_usart_rx_sta & 0x8000)         /* 接收到了数据? */
        {
            len = g_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(&g_uart1_handle, (uint8_t*)g_usart_rx_buf, len,
                              1000);        /* 发送接收到的数据 */

            /* 等待发送结束 */
            while(__HAL_UART_GET_FLAG(&g_uart1_handle,UART_FLAG_TC) != SET);

            printf("\r\n\r\n");             /* 插入换行 */
            g_usart_rx_sta = 0;
        }
        else
        {
            times++;

            if (times % 5000 == 0)
            {
                printf("\r\n 正点原子 STM32 开发板 串口实验\r\n");
                printf("正点原子@ALIENTEK\r\n\r\n\r\n");
            }

            if (times % 200 == 0) printf("请输入数据,以回车键结束\r\n");

            if (times % 30  == 0) LED0_TOGGLE(); /* 闪烁 LED,提示系统正在运行. */

            delay_ms(10);
        }
    }
}
```

我们主要看无限循环里面的逻辑：首先判断全局变量 g_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，在上位机显示。这里比较重点的两条语句是：第一条是调用 HAL 串口发送函数 HAL_UART_Transmit 来发送一段字符到串口。第二条是我们发送一个字节之后，要检测这个数据是否已经被发送完成了。如果全局变量 g_usart_rx_sta 的最高位为 0，则执行一段时间往上位机发送提示字符，以及让 LED0 每隔一段时间翻转，提示系统正在运行。

17.4 下载验证

在下载好程序后，可以看到板子上的 LED0 开始闪烁，说明程序已经在跑了。串口调试助手，我们用 XCOM，该软件在光盘有提供，且无需安装，直接可以运行，但是需要你的电脑安装有.NET Framework 4.0(WIN 自带了)或以上版本的环境才可以，该软件的详细介绍请看：http://www.openedv.com/posts/list/22994.htm 这个帖子。

接着我们打开 XCOM（正点原子的串口调试助手，位于光资料盘(A 盘)→6，软件资料→1，软件→5，其他小工具→串口调试助手→XCOM），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 COM15，另外，请注意：波特率是 115200）。因为我们在程序上面设置了必须输入回车，串口才认可接收到的数据，所以必须在发送数据后再发送一个回车符，这里 XCOM 提供的发送方法是通过勾选发送新行实现，只要勾选了这个选项，每次发送数据后，XCOM 都会自动多发一个回车(0x0D+0x0A)。设置好了发送新行，我们

再在发送区输入你想要发送的文字，然后单击发送，可以看到如图 15.4.1 所示信息：

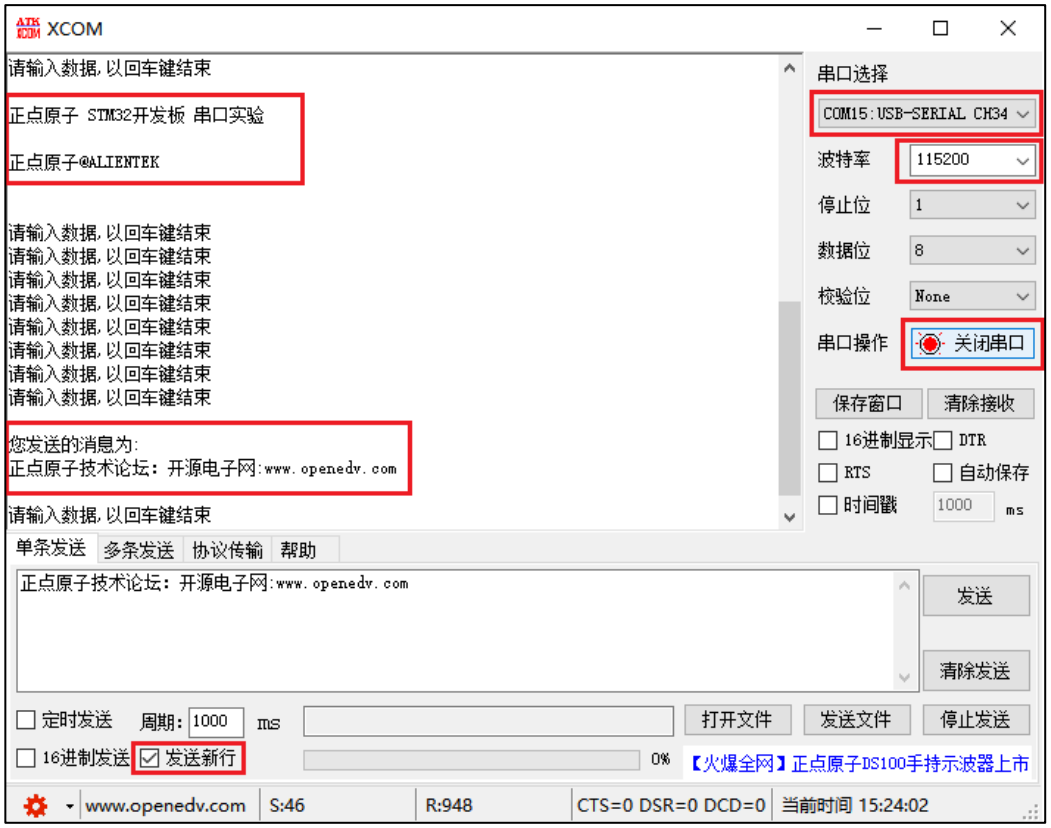


图 15.4.1 串口助手

可以看到，我们发送的消息被发送回来了。大家可以试试，如果不发送回车（取消发送新行），在输入内容之后，直接按发送是什么结果，大家测试一下吧。

第十八章 独立看门狗（IWDG）实验

本章我们学习如何使用 STM32F407 的独立看门狗（以下简称 IWDG）。STM32F407 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 KEY0 来喂狗，然后通过 LED0 提示复位状态。

本章分为如下几个小节：

18.1 IWDG 简介
18.2 硬件设计
18.3 程序设计
18.4 下载验证

18.1 IWDG 简介

独立看门狗本质上是一个定时器，这个定时器有一个输出端，可以输出复位信号。该定时器是一个 12 位的递减计数器，当计数器的值减到 0 的时候，就会产生一个复位信号。如果在计数没减到 0 之前，重置计数器的值的话，那么就不会产生复位信号，这个动作我们称为喂狗。看门狗功能由 VDD 电压域供电，在停止模式和待机模式下仍然可以工作。

18.1.1 IWDG 框图

下面先来学习 IWDG 框图，通过学习 IWDG 框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

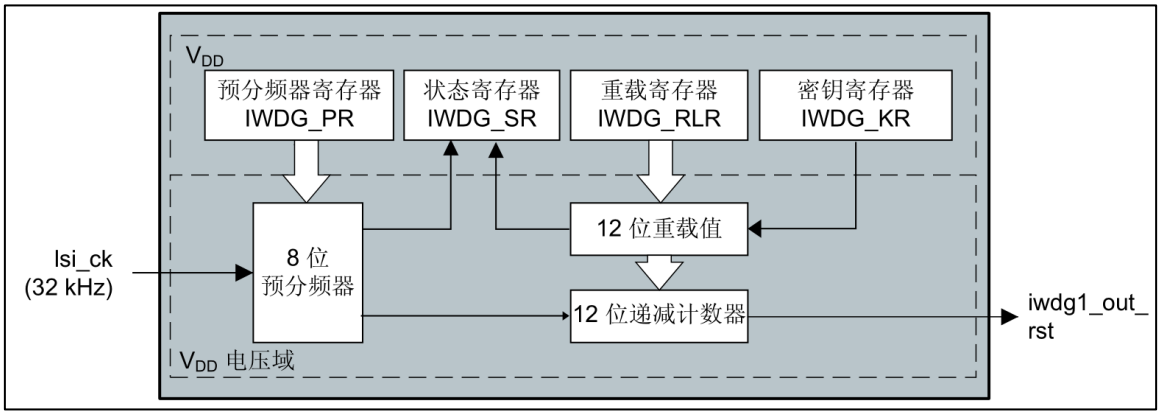


图 18.1.1.1 IWDG 框图

从 IWDG 框图整体认知就是，IWDG 有一个输入（时钟 LSI），经过一个 8 位的可编程预分频器提供时钟给一个 12 位递减计数器，满足条件就会输出一个复位信号。IWDG 内部输入/输出信号如下表：

信号名称	信号类型	说明
lsi_ck	数字信号	LSI 时钟
iwdg1_out_rst	数字信号	IWDG1 复位信号输出

表 18.1.1.1 IWDG 内部输入/输出信号

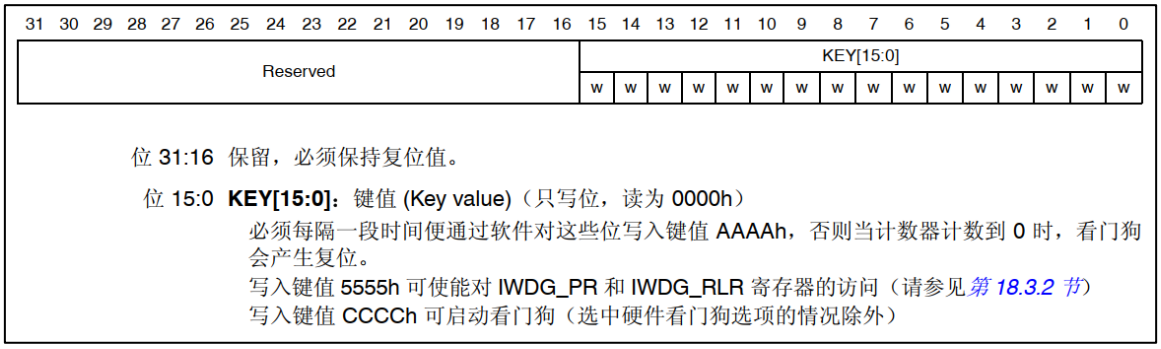
STM32F407 的独立看门狗由内部专门的 32Khz 低速时钟（lsi_ck）驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 32Khz，而是在 17~47Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 32Khz 的频率来计算，看门狗对时间的要求不是很精确，所以，时钟有些偏差，都是可以接受的。

18.1.2 IWDG 寄存器

IWDG 的框图很简单，用到的寄存器也不多。我们主要用到其中 3 个寄存器：

- **键寄存器（IWDG_KR）**

键寄存器可以看作是独立看门狗的控制寄存器，其描述如图 18.1.2.1 所示：



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（请参见*第 18.3.2 节*）

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）

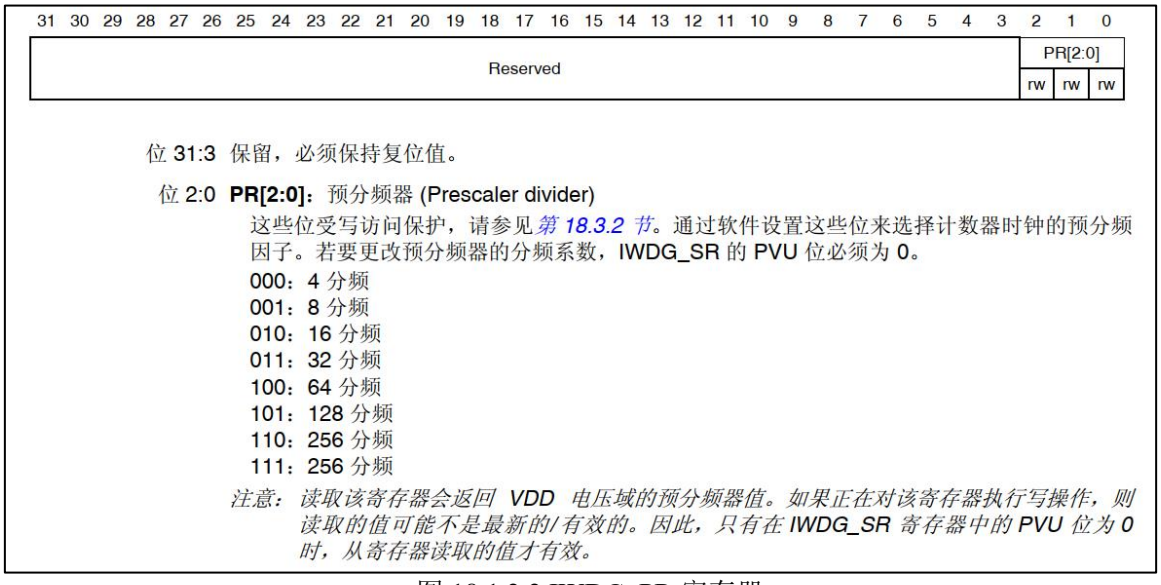
图 18.1.2.1 WDG_KR 寄存器

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 0x0000 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。

- **预分频寄存器（IWDG_PR）**

预分频寄存器描述如图 18.1.2.2 所示：



位 31:3 保留，必须保持复位值。

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

这些位受写访问保护，请参见*第 18.3.2 节*。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。

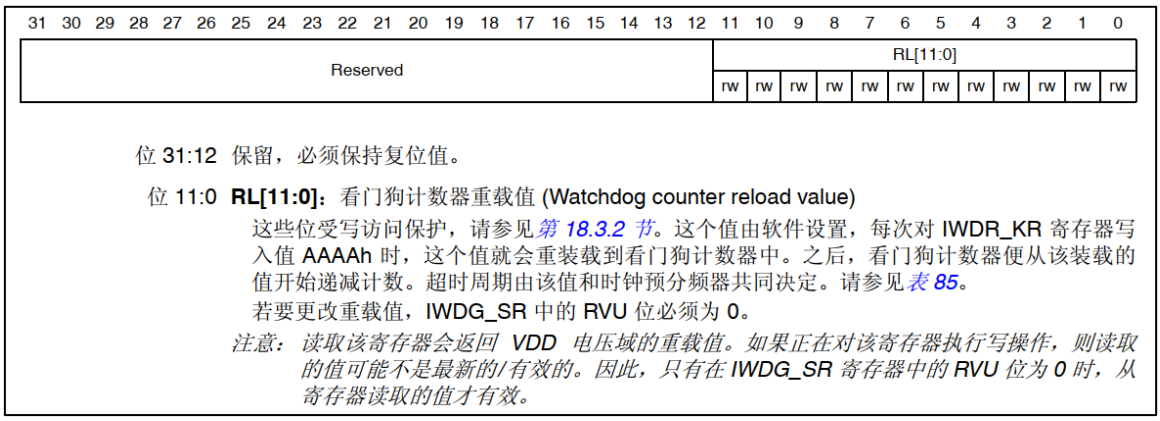
000：4 分频
001：8 分频
010：16 分频
011：32 分频
100：64 分频
101：128 分频
110：256 分频
111：256 分频

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 18.1.2.2 IWDG_PR 寄存器

该寄存器用来设置看门狗时钟（LSI）的分频系数，最低为 4，最高为 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。

- **重载寄存器（IWDG_RLR）**



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，请参见*第 18.3.2 节*。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。请参见*表 85*。

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 18.1.2.3 IWDG_RLR 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，只有低 12 位是有效的。

18.2 硬件设计

1. 例程功能

在配置看门狗后，LED0 将常亮，如果 KEY0 按键按下，就喂狗，只要 KEY0 不停的按，看门狗就一直不会产生复位，保持 LED0 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 LED0 熄灭一次。

2. 硬件资源

1）LED 灯
LED0 - PF0
2）独立按键
KEY0 - PE2
3）独立看门狗

3. 原理图

独立看门狗实验的核心是在 STM32F407 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，一个用来触发喂狗信号，另外一个用来指示程序是否重启。喂狗我们采用板上的 KEY0 键来操作，而程序重启，则是通过 LED0 来指示的。

18.3 程序设计

18.3.1 IWDG 的 HAL 库驱动

IWDG 在 HAL 库中的驱动代码在 stm32f4xx_hal_iwdg.c 文件（及其头文件）中。

1. HAL_IWDG_Init 函数

IWDG 的初始化函数，其声明如下：

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

- **函数描述：**
  用于初始化 IWDG。
- **函数形参：**
  形参 1 是 IWDG 句柄，IWDG_HandleTypeDef 结构体类型，其定义如下：

```
typedef struct
{
  IWDG_TypeDef                *Instance;  /* IWDG 寄存器基地址 */
  IWDG_InitTypeDef             Init;      /* IWDG 初始化参数 */
```

```
}IWDG_HandleTypeDef;
```

1）Instance：指向 IWDG 寄存器基地址。

2）Init：IWDG 初始化结构体，用于配置计数器的相关参数。

IWDG_InitTypeDef 这个结构体类型定义如下：

```
typedef struct
{
  uint32_t Prescaler;    /* 预分频系数 */
  uint32_t Reload;       /* 重装载值 */
} IWDG_InitTypeDef;
```

1）Prescaler：预分频系数，IWDG_PRESCALER_4 到 IWDG_PRESCALER_256。

2）Reload：重装载值，范围：0 到 0x0FFF。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

2. HAL_IWDG_Refresh 函数

HAL_IWDG_Refresh 函数是独立看门狗的喂狗函数。其声明如下：

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

- **函数描述：**

  用于把重装载寄存器的值重载到计数器中，喂狗，防止 IWDG 复位。

- **函数形参：**

  形参 1 是 IWDG_HandleTypeDef 结构体指针类型的 IWDG 句柄。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

独立看门狗配置步骤

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。

通过该函数设置看门狗的分频系数，和重装载的值。看门狗的喂狗时间（也就是看门狗溢出时间）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);
```

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，在内部已经调用了该宏启动看门狗。

18.3.2 程序流程图

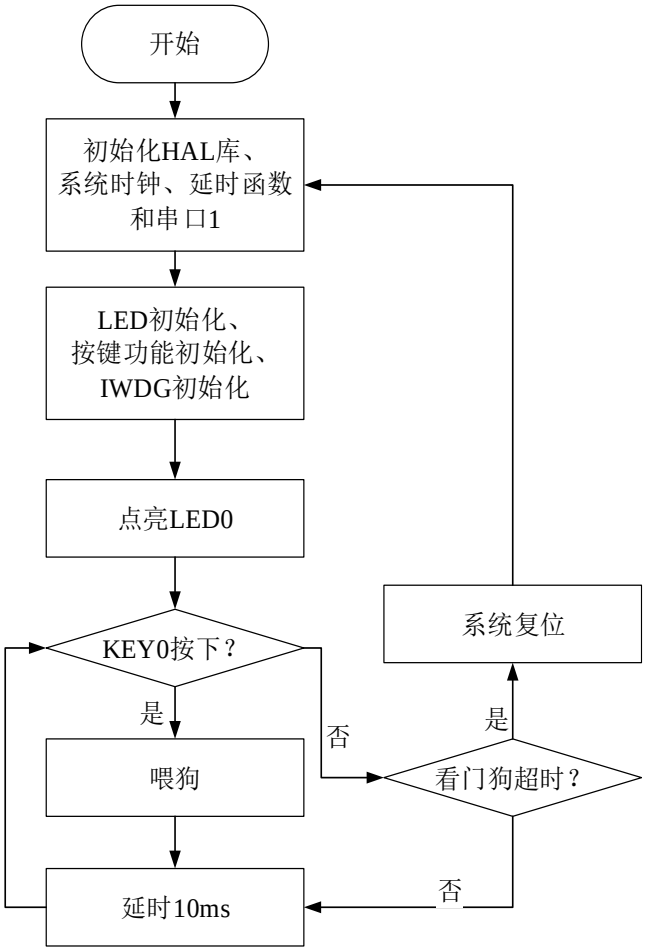


图 18.3.2.1 独立看门狗实验程序流程图

18.3.3 程序解析

1. IWDG 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。独立看门狗（IWDG）驱动源码包括两个文件：wdg.c 和 wdg.h。

wdg.h 头文件只有函数的声明，就不解释了。下面我们直接解析 wdg.c 的程序，先看 IWDG 的初始化函数，其定义如下：

```
/**
 * @brief       初始化独立看门狗
 * @param       prer: IWDG_PRESCALER_4~IWDG_PRESCALER_256,对应 4~256 分频
 *   @arg       分频因子 = 4 * 2^prer. 但最大值只能是 256!
 * @param       rlr: 自动重装载值,0~0XFFF.
 * @note        时间计算(大概):Tout=((4 * 2^prer) * rlr) / 40 (ms).
 * @retval      无
 */
void iwdg_init(uint8_t prer, uint16_t rlr)
{
    g_iwdg_handle.Instance = IWDG;
    g_iwdg_handle.Init.Prescaler = prer;     /* 设置 IWDG 分频系数 */
    g_iwdg_handle.Init.Reload = rlr;         /* 重装载值 */
    HAL_IWDG_Init(&g_iwdg_handle);
}
```

IWDG_Init 是独立看门狗初始化函数，主要设置预分频数和重装载寄存器的值。通过这两个寄存器，就可以大概知道看门狗复位的时间周期为多少了。

```
/**
 * @brief       喂独立看门狗
 * @param       无
 * @retval      无
 */
void iwdg_feed(void)
{
    HAL_IWDG_Refresh(&g_iwdg_handle); /* 重装载计数器 */
}
```

iwdg_feed 函数用来喂狗，在该函数内部只需调用 HAL 库函数 HAL_IWDG_Refresh。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    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);/* 预分频数为 64,重载值为 500,溢出时间约为 1s */
    LED0(0);                                /* 点亮 LED0(红灯) */
    while(1)
    {
       if (key_scan(1) == KEY0_PRES)     /* 如果 KEY0 按下,则喂狗,支持连按 */
        {
            iwdg_feed();                    /* 喂狗 */
        }
        delay_ms(10);
    }
}
```

在 main 函数里，先初始化系统和用户的外设代码，然后先点亮 LED0，在无限循环里开始获取按键的键值，并判断是不是按键 KEY0 按下，是的话就喂狗，不是则延时 10ms，继续上述操作。当 1 秒钟后都没测到按键 KEY0 按下，IWDG 就会产生一次复位信号，系统复位，可以看到 LED0 因系统复位熄灭一次，再亮。反之，当按下按键 KEY0 后，1 秒内再按下按键 KEY0，就会及时喂狗，结果就是系统不会复位，LED0 也就不会闪烁。

iwdg_init(IWDG_PRESCALER_64, 500);这个语句有必要跟大家说明一下，这里的第一个形参直接使用 HAL 库自定义的 IWDG_PRESCALER_64，即预分频系数为 64，重装载值是 500，所以可由公式得到 Tout=64×500/32=1000ms，即溢出时间就是 1s。只要你在一秒钟之内，有一次写 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。

18.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，证明系统在不停的复位，否则 LED0 常亮。这时我们试试不停的按 KEY0 按键，可以看到 LED0 就常亮了，不会再闪烁。说明我们的实验是成功的。

第十九章 窗口门狗（WWDG）实验

本章我们学习如何使用 STM32F407 的另外一个看门狗，窗口看门狗（以下简称 WWDG）。我们将使用窗口看门狗的中断功能来喂狗，通过 LED0 和 LED1 提示程序的运行状态。

本章分为如下几个小节：

19.1 WWDG 简介
19.2 硬件设计
19.3 程序设计
19.4 下载验证

19.1 WWDG 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。窗口看门狗跟独立看门狗一样，也是一个递减计数器，不同的是它们的复位条件不一样。窗口看门狗产生复位信号有两个条件：

1） 当递减计数器的数值从 0x40 减到 0x3F 时（T6 位跳变到 0）。

2） 当喂狗的时候如果计数器的值大于 W[6:0]时，此数值在 WWDG_CFR 寄存器定义。

上述的两个条件详细解释是，当计数器的值减到 0x40 时还不喂狗的话，到下一个计数就会产生复位，这个值称为窗口的下限值，是固定的值，不能改变。这个跟独立看门狗类似，不同的是窗口看门狗的计数器的值在减到某一个数之前喂狗的话也会产生复位，这个值叫窗口的上限，上限值 W[6:0]由用户设置。窗口看门狗计数器的上限值和下限值就是窗口的含义，喂狗也必须在窗口之内，否则就会复位。

19.1.1 WWDG 框图

下面先来学习 WWDG 框图，通过学习 WWDG 框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

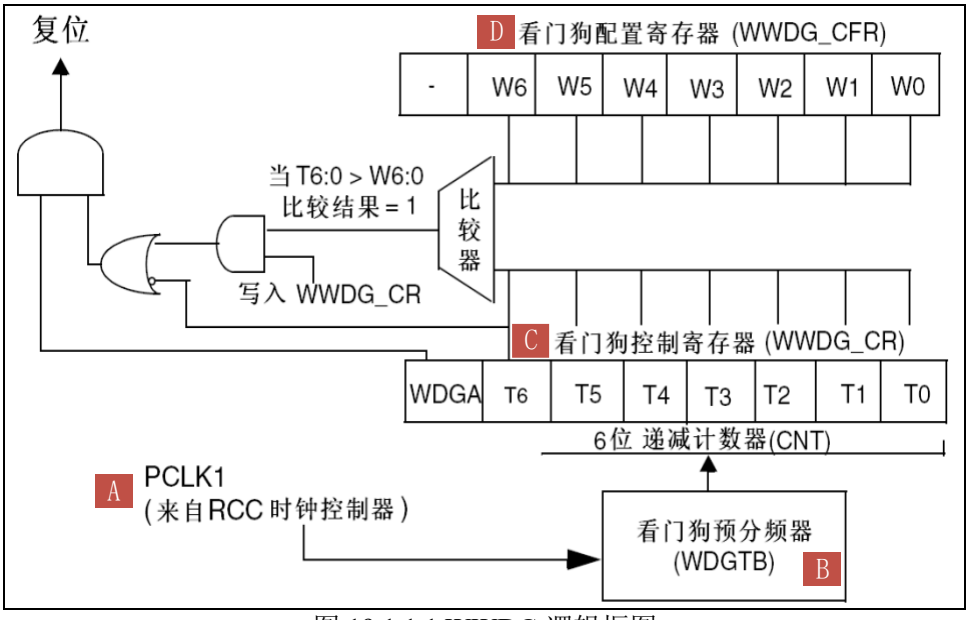


图 19.1.1.1 WWDG 逻辑框图

WWDG 有一个来自 RCC 的 PCLK1 输入时钟，经过一个 4096 的分频器(4096 分频在设计时已经设定死了，图中并没有给出来，但我们可以通过查看寄存器 WWDG_CFR 的 WDGTB 位的描述知道)，再经过一个分频系数可选（1、2、4、8）的可编程预分频器提供时钟给一个 7 位递减计数器，这里有两个输出信号。

结合寄存器分析窗口看门狗的上限值和下限值。W[6:0] 是 WWDG_CFR 寄存器的低 7 位，用于与递减计数器 T[6:0]比较的窗口值，也就是我们说的上限值，由用户设置。0x40 就是下限值，递减计数器达到这个值就会产生复位。T6 位就是 WWDG_CR 寄存器的位 6，即递减计数器 T[6:0]的最高位。他们的关系可以用图 19.1.1.2 来说明：

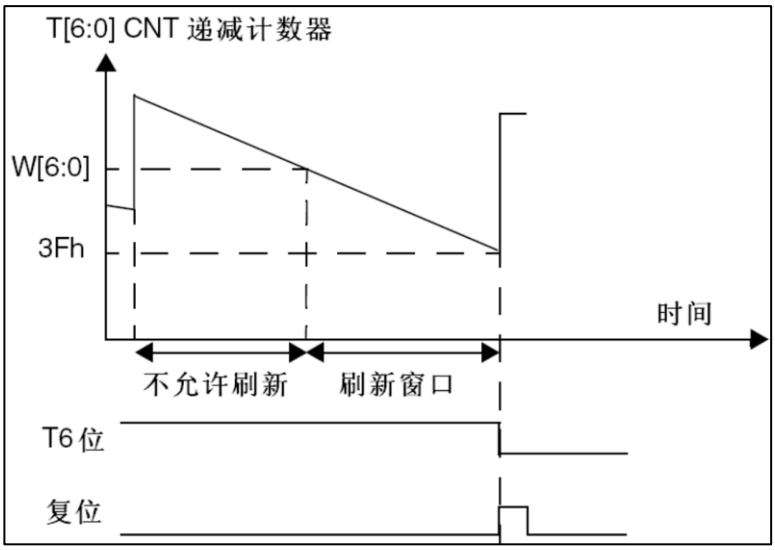


图 19.1.1.2 窗口看门狗工作示意图

图 19.1.1.2 可以看出，递减计数器的值递减过程中，当 T[6:0]>W[6:0]是不允许刷新 T[6:0]的值，即不允许喂狗，否则会产生复位。只有在 W[6:0]<T[6:0]< 0x3F 这个时间可以喂狗，这就是喂狗的窗口时间。当 T[6:0]=0x3F，即 T6 位为 0 这一刻，也会产生复位。

上限值 W[6:0]是由用户自己设置，但是一定要确保大于 0x40，否则就不存在上图的窗口了，下限值 0x40 是固定的，不可修改。

知道了窗口看门狗的工作原理，下面学习如何计算窗口看门狗的超时公式：

$$T_{WWDG} = T_{PCLK1} \times 4096 \times 2^{\wedge WDGTB} \times (T[5:0] + 1)$$

其中：

T_{WWDG}： WWDG 超时时间(单位为 ms）

T_{PCLK1}： APB1 以 ms 为单位的时钟间隔（即单位为 1 /KHz）

T[5:0]：窗口看门狗的计数器低 6 位(T6 位固定 1，范围 0x7F~0x40，对应值 0x3F~0x00)

根据以上公式，我们来计算一下对应当重装值为 0x40 时，分频系数 WDGTB=3 时对应，假设 PCLK1 = 42MHz，下一个计数将发生复位，到达下个复位的时间是：1/(42MHz)*4096*2^3*(0+1) =780.19us ，类似地，可以得到在 42MHz 时钟下不同分频值那么可以得到最小-最大超时时间表如表 19.1.1.1 所示：

WDGTB	最小超时值	最大超时值
0	97.52μs	6.24ms
1	195.05μs	12.48ms
2	390.10μs	24.97ms
3	780.19μs	49.96ms

表 19.1.1.1 42M 时钟下窗口看门狗的最小最大超时表

19.1.2 WWDG 寄存器

WWDG 只有 3 个寄存器，具体如下：

- **控制寄存器（WWDG_CR）**

窗口看门狗的控制寄存器描述如图 19.1.2.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								WDGA	T[6:0]						
								rs	rw						

位 31:8 保留，必须保持复位值。

位 7 **WDGA**：激活位 (Activation bit)
此位由软件置 1，只有复位后才由硬件清零。当 WDGA = 1 时，看门狗可产生复位。
0:禁止看门狗
1:使能看门狗

位 6:0 **T[6:0]**：7 位计数器 (MSB 到 LSB)
这些位用来存储看门狗计数器的值。它每隔 (4096 x 2^{WDGTB}) PCLK1 个周期递减一次。当它从 0x40 滚动到 0x3F（T6 清零）时会产生复位。

图 19.1.2.1 WWDG_CR 寄存器

该寄存器只有低八位有效，其中 T[6:0]用来存储看门狗的计数器的值，随时更新的，每隔（4096×2^WDGTB[2:0]）PCLK 个周期减 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，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。

- **配置寄存器（WWDG_CFR）**

配置寄存器描述如图 19.1.2.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
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 19.1.2.2 WWDG_CFR 寄存器

该寄存器中的 EWI 位是提前唤醒中断，如果该位置 1，当递减计数器等于 0x40 时产生提前唤醒中断，我们就可以及时喂狗以避免 WWDG 复位。因此，我们一般都会用该位来设置中断，当窗口看门狗的计数器值减到 0x40 的时候，如果该位设置，并开启了中断，则会产生中断，我们可以在中断里面向 WWDG_CR 重新写入计数器的值，来达到喂狗的目的。注意这里在进入中断后，必须在不大于 1 个窗口看门狗计数周期的时间（在 pclk1 频率为 42M 且 WDGTB 为 0 的条件下，该时间为 97.52us）内重新写 WWDG_CR，否则，看门狗将产生复位！

- **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 0x40 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。状态寄存器描述如图 19.1.2.3 所示。

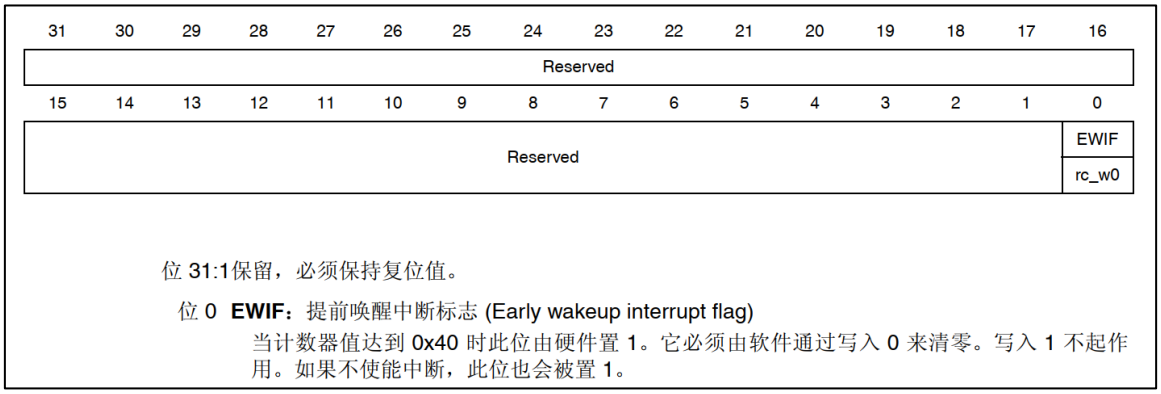


图 19.1.2.3 WWDG_SR 寄存器

19.2 硬件设计

1. 例程功能

先点亮 LED0 延时 300ms 后，初始化窗口看门狗，进入死循环，关闭 LED0。然后等待窗口看门狗中断的到来，在中断里面，喂狗，并执行 LED1 的翻转操作。我们将通过 LED0 来指示 STM32 是否被复位了，如果被复位了就会点亮 300ms。LED1 用来指示中断喂狗，每次中断喂狗翻转一次。

2. 硬件资源

1）LED 灯
LED0 - PE0
LED1 - PE1
2）窗口看门狗

3. 原理图

窗口看门狗属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 LED0 和 LED1 来指示 STM32F407 的复位情况和窗口看门狗的喂狗情况。

19.3 程序设计

19.3.1 WWDG 的 HAL 库驱动

WWDG 在 HAL 库中的驱动代码在 stm32f4xx_hal_wwdg.c 文件（及其头文件）中。

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

- **函数描述：**
  用于初始化 WWDG。
- **函数形参：**
  形参 1 是 WWDG 句柄，WWDG_HandleTypeDef 结构体类型，其定义如下：

```
typedef struct
{
  WWDG_TypeDef             *Instance;    /* WWDG 寄存器基地址 */
  WWDG_InitTypeDef          Init;        /* WWDG 初始化参数 */
}WWDG_HandleTypeDef;
```

1）Instance：指向 WWDG 寄存器基地址。
2）Init：WWDG 初始化结构体，用于配置计数器的相关参数。
WWDG_InitTypeDef 这个结构体类型定义如下：

```
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;
```

1）Prescaler：预分频系数，WWDG_PRESCALER_1 到 WWDG_PRESCALER_128。
2）Window：窗口值，即上限值。
3）Counter：计数器值，用于保存要设置计数器的值。
4）EWIMode：提前唤醒中断使能。

- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

窗口看门狗配置步骤

1） 使能 WWDG 时钟

WWDG 不同于 IWDG，IWDG 有自己独立的 32KHz 时钟。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();
```

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，详见本例程源码。

3） 开启 WWDG

通过设置 WWDG_CR 寄存器的 WDGA(bit7)位为 1 来实现开启窗口看门狗，同样是在 HAL_WWDG_Init 函数里面实现。

4） 使能中断通道并配置优先级（如果开启了 WWDG 中断）

WWDG 的中断也是通过：HAL_NVIC_EnableIRQ 函数使能，通过 HAL_NVIC_SetPriority 函数设置优先级。

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，我们均放在该回调函数中。

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：WWDG_IRQHandler，喂狗函数为：HAL_WWDG_Refresh。

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

HAL 库定义了一个 WWDG 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。同时该函数会调用回调函数 HAL_WWDG_EarlyWakeupCallback，提前唤醒中断逻辑（喂狗、闪灯）我们写在回调函数 HAL_WWDG_EarlyWakeupCallback 中。

19.3.2 程序流程图

本实验利用窗口看门狗的特性，配置一个合适的窗口时间，并开启了提前唤醒中断，如果程序未在合适的时间喂狗，则会触发窗口看门狗中断。我们在进入中断后需要第一时间喂狗，否则系统将会复位，并用 LED1 的翻转来表示一次未及时喂狗的事件。

下面的流程图表示了 main 函数必要的编程步骤。

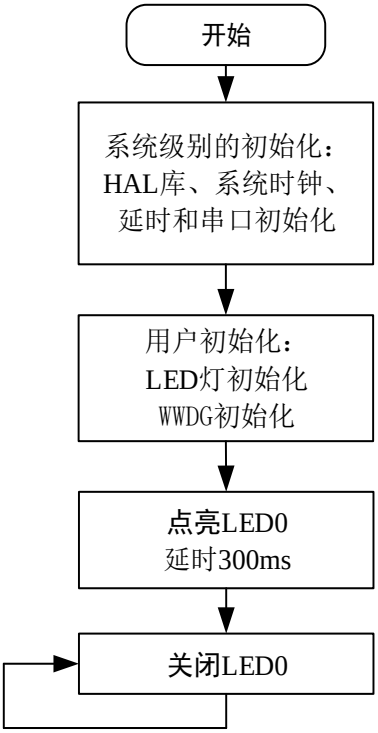


图 19.3.2.1 窗口看门狗实验程序流程图

19.3.3 程序解析

1. WWDG 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。窗口看门狗（WWDG）驱动源码包括两个文件：wdg.c 和 wdg.h。

wdg.h 头文件只有函数的声明，就不解释了。下面我们直接解析 wdg.c 的程序，先看 WWDG 的初始化函数，其定义如下：

```
/**
 * @brief       初始化窗口看门狗
 * @param       tr: T[6:0],计数器值
 * @param       tw: W[6:0],窗口值
 * @param       fprer: 分频系数（WDGTB）,范围:WWDG_PRESCALER_1~WWDG_PRESCALER_8,
 *                  表示 2^WDGTB 分频, Fwwdg=PCLK1/(4096*2^fprer). 一般 PCLK1=42Mhz
 * @retval      无
 */
void wwdg_init(uint8_t tr, uint8_t wr, uint32_t fprer)
{
    wwdg_handler.Instance = WWDG;
    wwdg_handler.Init.Prescaler = fprer;            /* 设置分频系数 */
    wwdg_handler.Init.Window = wr;                  /* 设置窗口值 */
    wwdg_handler.Init.Counter = tr;                 /* 设置计数器值 */
    wwdg_handler.Init.EWIMode = WWDG_EWI_ENABLE;    /* 使能窗口看门狗提前唤醒中断 */
    HAL_WWDG_Init(&wwdg_handler);                   /* 初始化 WWDG */
}
```

WWDG_Init 是独立看门狗初始化函数，主要设置预分频数、窗口值和计数器的值，以及选择是否使能窗口看门狗提前唤醒中断。

因为用到中断，我们用 HAL_WWDG_MspInit 函数来编写窗口看门狗中断的初始化代码。当然大家也可以 HAL_WWDG_MspInit 函数的代码放到 wwdg_init 函数里面。这个初始化框架就是 HAL 库的特点。

```
void HAL_WWDG_MspInit(WWDG_HandleTypeDef *hwwdg)
{
```

```
    __HAL_RCC_WWDG1_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 */
}
```

HAL_WWDG_MspInit 函数会被 HAL_WWDG_Init 函数调用。该函数使能窗口看门狗的时钟，并设置窗口看门狗中断的抢占优先级为 2，响应优先级为 3。

HAL_WWDG_MspInit 函数会被 HAL_WWDG_Init 函数调用。该函数使能窗口看门狗的时钟，并设置窗口看门狗中断的抢占优先级为 2，响应优先级为 3。

```
/**
 * @brief       窗口看门狗中断服务程序
 * @param       无
 * @retval      无
 */
void WWDG_IRQHandler(void)
{
    HAL_WWDG_IRQHandler(&g_wwdg_handle);
}
```

WWDG_IRQHandler 函数是窗口看门狗中断服务函数，而这个函数实际上就是调用 HAL 库的中断处理函数 HAL_WWDG_IRQHandler。逻辑程序在下面的这个回调函数中：

```
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg)
{
    HAL_WWDG_Refresh(&g_wwdg_handle);/* 更新窗口看门狗值 */
    LED1_TOGGLE();                   /* LED1 闪烁 */
}
```

在回调函数内部调用 HAL_WWDG_Refresh 函数喂狗，并翻转 LED1。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */

    LED0(0);                                /* 点亮 LED0(红灯) */
    delay_ms(300);    /* 延时 100ms 再初始化看门狗,LED0 的变化"可见" */
    /* 计数器值为 7f,窗口寄存器为 5f,分频数为 8 */
    wwdg_init(0X7F, 0X5F, WWDG_PRESCALER_8);
    while(1)
    {
        LED0(1);                           /* 关闭 LED0(红灯) */
    }
}
```

在 main 函数里，先初始化系统和用户的外设代码，然后先点亮 LED0，延时 300ms 后，初始化窗口看门狗，进入死循环，关闭 LED0。

调用 wwdg_init(0X7F, 0X5F, WWDG_PRESCALER_8)这个语句，就设置计数器值为 7f，窗口寄存器为 5f，分频数为 8，然后可由前面的公式得到窗口上限时间 Twwdg=4096×8×（0x7F-0x5F）/42MHz=24.98ms，窗口下限时间 Twwdg=4096×8×（0x7F-0x3F）/42MHz=49.97ms，即喂狗的窗口区间为 24.98~49.97ms。我们在程序的其它地方没有喂狗，所以程序会在 49.97ms 左右进入死前中断，我们在中断中喂狗一次，并翻转 LED1。

19.4 下载验证

下载代码后，可以看到 LED0 亮了一下就熄灭，紧接着 LED1 开始不停的闪烁。可以接入

示波器测试得每秒钟闪烁 20 次左右，说明程序在中断不停的喂狗，和我们预期的一致。

第二十章 TFT-LCD（MCU 屏）实验

本章我们将介绍正点原子的 TFT-LCD 模块（MCU 屏），该模块采用 TFT-LCD 面板，可以显示 16 位色的真彩图片。在本章中，我们将使用开发板上的 TFTLCD 接口（仅支持 MCU 屏，本章仅介绍 MCU 屏的使用），来点亮 TFT-LCD，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。

本章分为如下几个小节：

20.1 TFTLCD 和 FSMC 简介
20.2 硬件设计
20.3 程序设计
20.4 下载验证

20.1 TFT-LCD 简介

本章我们将通过 STM32F407 的 FSMC 外设来控制 TFT-LCD 的显示，这样我们就可以用 STM32 输出一些信息到显示屏上了。

20.1.1 TFT-LCD 简介

液晶显示器，即 Liquid Crystal Display，利用了液晶导电后透光性可变的特性，配合显示器光源、彩色滤光片和电压控制等工艺，最终可以在液晶阵列上显示彩色的图像。目前液晶显示技术以 TN、STN、TFT 三种技术为主，TFT-LCD 即采用了 TFT（Thin Film Transistor）技术的液晶显示器，也叫薄膜晶体管液晶显示器。

TFT-LCD 与无源 TN-LCD、STN-LCD 的简单矩阵不同的是，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 式显示器具有很多优点：高响应度，高亮度，高对比度等等。TFT 式屏幕的显示效果非常出色，广泛应用于手机屏幕、笔记本电脑和台式机显示器上。

由于液晶本身不会发光，加上液晶本身的特性等原因，使得液晶屏的成像角受限，我们从屏幕的的一侧可能无法看清液晶的显示内容。液晶显示器的成像角的大小也是评估一个液晶显示器优劣的指标，目前，规格较好的液晶显示器成像角一般在 120°~160°之间。

正点原子 TFT-LCD 模块(MCU 屏)配套开发板上液晶插座，它有如下特点：

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，我们以正点原子 2.8 寸（此处的寸是代表英寸，下同）的 TFT-LCD 模块为例介绍，（其他尺寸的 LCD 可参考具体的 LCD 型号的资料，也比较类似），该模块支持 65K 色显示，显示分辨率为 320×240，接口为 16 位的 8080 并口，自带触摸功能。

该模块的外观如图 20.1.1.1 所示：



图 20.1.1.1 正点原子 2.8 寸 TFTLCD 外观图

模块原理图如图 20.1.1.2 所示：

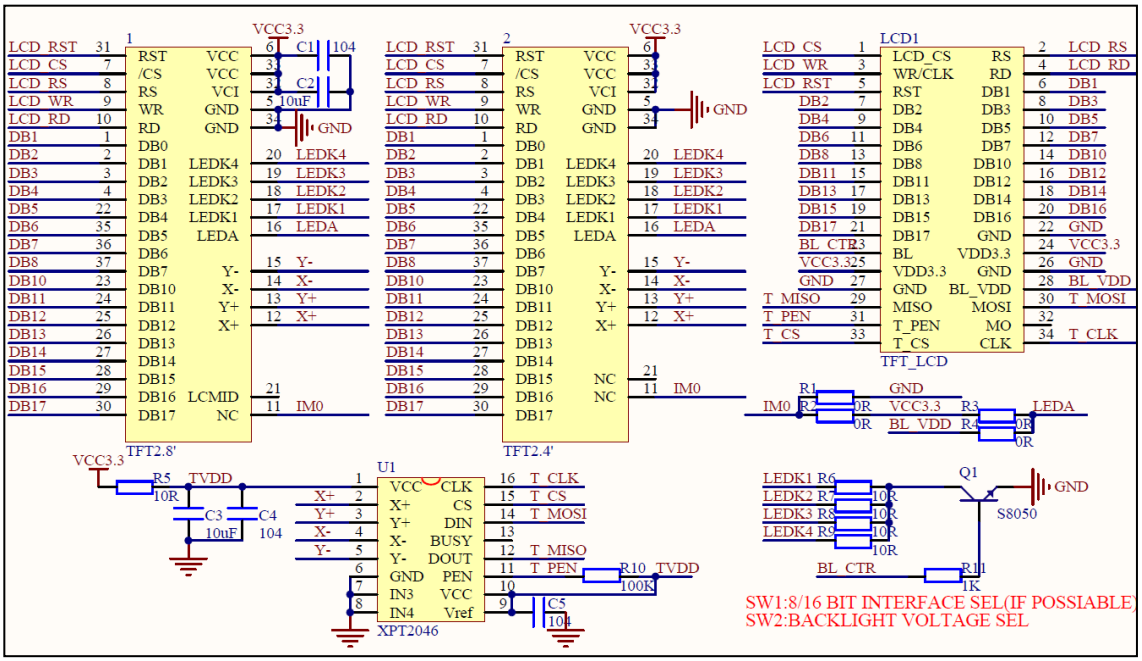


图 20.1.1.2 正点原子 TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，即图中 TFT_LCD 部分。从图 20.1.1.2 可以看出，正点原子 TFTLCD 模块采用 16 位的并口方式与外部连接。图 20.1.1.2 还列出了触摸控制的接口，但触摸控制是在显示的基础上叠加的一个控制功能，不配置也不会对显示造成影响，我们放到以后的章节再介绍触摸的用法。该模块与显示功能相关的信号线如表 20.1.1.1 所示：

名称	功能
CS	TFT-LCD 片选信号
WR	向 TFT-LCD 写入数据
RD	从 TFT-LCD 读取数据
D[15:0]	16 位双向数据线
RST	硬复位 TFT-LCD
RS	命令/数据标志（0，读写命令；1，读写数据）
LCD_BL	背光控制

表 20.1.1.1 TFT-LCD 接口信号线

上述的接口线实际是对应到液晶显示控制器上的，这个芯片位于液晶屏的下方，所以我们从外观图上看不到。控制 LCD 显示的过程，就是按其显示驱动芯片的时序，把色彩和位置信息正确地写入对应的寄存器。

20.1.2 液晶显示控制器

正点原子提供 2.8/3.5/4.3/7 寸等不同尺寸和分辨率的 TFTLCD 模块，驱动显示的芯片为：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，可配置支持 8/9/16/18 位的总线中的一种，可以通过 3/4 线串行协议或 8080 并口驱动。正点原子的 TFT-LCD 模块上的电路配置为 8080 并口方式，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 20.1.2.1 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 20.1.2.1 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，在我们的 LCD 模块里面，ILI9341 的 D0 和 D12 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。

知道了屏幕的显色信息后，我们如何驱动它呢？我们通过《ILI9341_DS.pdf》来了解 8080 并口方式下如何操作这个芯片。

以写周期为例，8080 方式下的操作时序如图 20.1.2.2 所示。

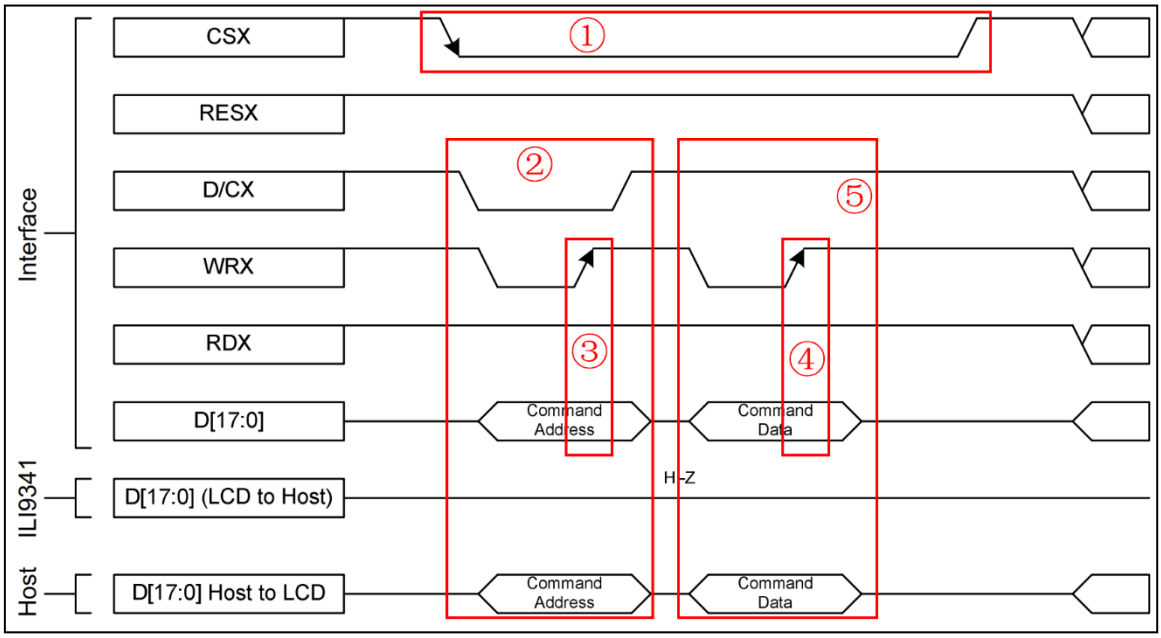


图 20.1.2.2 8080 方式下对液晶控制器的写操作

上图中的各个控制线与我们在表 20.1.1.1 提到的命名有些差异，因为我们在原理图时往往为了方便自己记忆会对命名进行微调，为了方便读者对照，我们把图 20.1.2.2 中列出的引脚引脚与我们的 TFTLCD 模块的的对应关系再列出，如表 20.1.2.1 所示。

TFTLCD 名称	ILI9341	功能
CS	CSX	TFT-LCD 片选信号
WR	WRX	向 TFT-LCD 写入数据
RD	RDX	从 TFT-LCD 读取数据
D[15:0]	D[17:13] D[11:1]	16 位双向数据线
RS	D/CX	命令/数据标志（0，读写命令；1，读写数据）

表 20.1.2.1 TFT-LCD 引脚与液晶控制器的对应关系

这下我们再来分析一下图 20.1.2.2 所示的写操作的时序，控制液晶的主机，在整个写周期内需要控制片选 CSX 拉低（标注为①），之后对其它的控制线的电平才有效。在标号②表示的这个写命令周期中，D/CX 被位低（参考 ILI9341 的引脚定义），同时把命令码通过数据线 D[17:0]（我们实际只用了 16 个引脚）按位编码。注意到③处，需要数据线在入电平拉高后再保持一段时间以便数据被正确采样。

图 20.1.2.2 中⑤表示写数据操作，与前面描述的写命令操作只有 D/CX 的操作不同，读者们可以尝试自己分析一下。更多的关于 ILI9341 的读写操作时序则参考《ILI9341_DS.pdf》。

通过前述的时序分析，我们知道了对于 ILI9341 来说，控制命令有命令码、数据码之分，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。

指令 0xD3，是读 ID4 指令，用于读取 LCD 控制器的 ID，该指令如表 20.1.2.2 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	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 20.1.2.2 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 20.1.2.3 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 20.1.2.3 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 20.1.2.4 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 20.1.2.4 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 20.1.2.5 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	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 20.1.2.5 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 20.1.2.6 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	0	1	1	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 20.1.2.6 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 20.1.2.7 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 20.1.2.7 0X2C 指令描述

由表 20.1.2.6 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，期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 20.1.2.8 所示：

顺序	控制			各位描述 D[15:0]												HEX
	RS	RD	WR	15~11	10	9	8	7	6	5	4	3	2	1	0	
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 20.1.2.8 0X2E 指令描述

该指令用于读取 GRAM，如表 20.1.2.7 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。

20.1.3 FSMC 简介

ILI9341 的 8080 通讯接口时序可以由 STM32 使用 GPIO 接口进行模拟，但这样效率太低，STM32 提供了一种更高效的控制方法——使用 FSMC 接口实现 8080 时序，但 FSMC 是 STM32 片上外设的一种，并非所有的 STM32 都拥有这种硬件接口，使用何种方式驱动需要在芯片选型时就确定好。我们的开发板支持 FSMC 接口，下面我们来了解一下这个接口的功能。

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡连接，FSMC 接口可以通过地址信号，快速地找到存储器对应存储块上的数据。STM32F407 系列的芯片都带有 FSMC 接口，FSMC 接口支持包括 SRAM、NAND FLASH、NOR FLASH 和 PSRAM 等存储器。FSMC 接口的结构如图 20.1.3.1 所示：

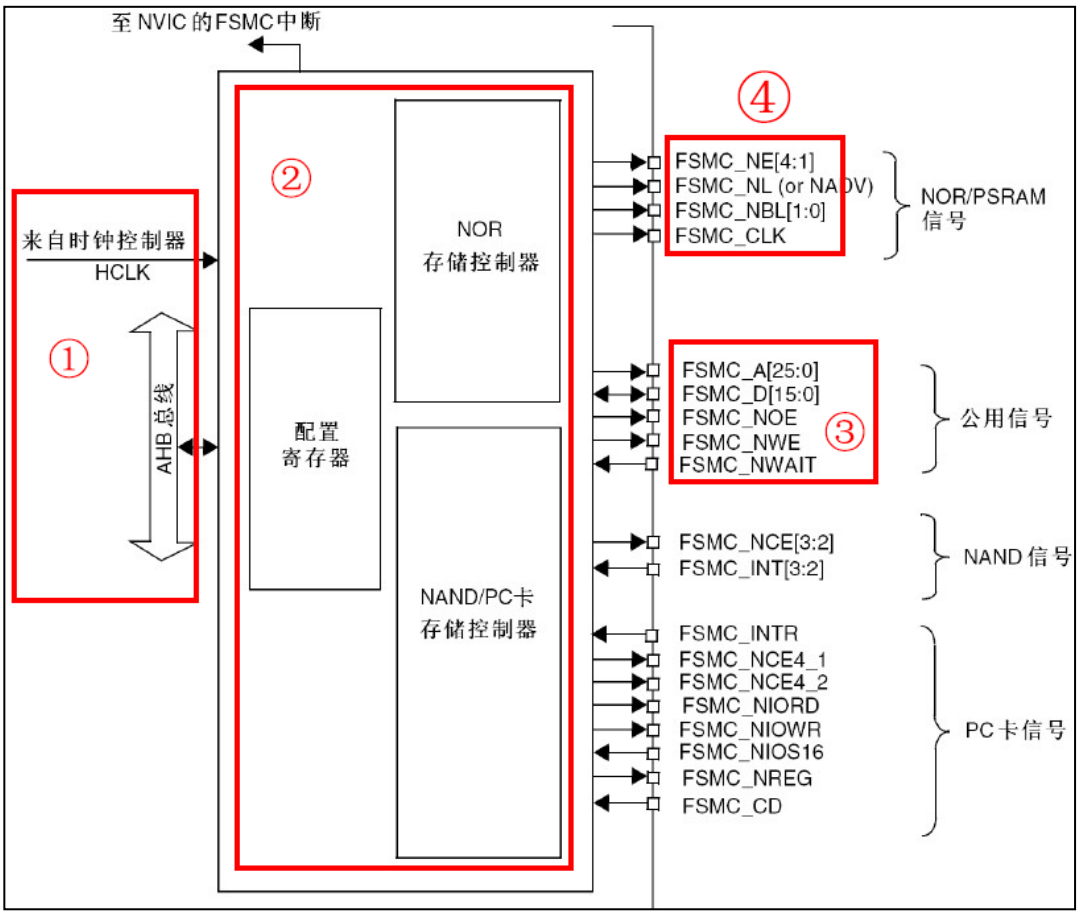


图 20.1.3.1 FSMC 框图

从图 20.1.3.1 我们可以看出，STM32 的 FSMC 可以驱动 NOR/PSRAM、NAND、PC 卡这 3 类设备，他们具有不同的 CS 以区分不同的设备。本部分我们要用到的是 NOR/PSRAM 的功能。①为 FSMC 的总线和时钟源，②为 STM32 内部的 FSMC 控制单元，③是连接硬件的引脚，这里的“公共信号”表示不论我们驱动前面提到的 3 种设备中的哪种，这些 IO 是共享的，所以如果需要用到多种功能的情况，程序上还要考虑分时复用。④是 NOR/PSRAM 会使用到的信号控制线，③和④这些信号比较重要，它们的功能如表 20.1.3.1：

FSMC 信号名称	信号方向	功能
FSMC_NE[x]	输出	STM32F4 有四个片选引脚，x = 1...4，每个对应不同的内存块
FSMC_CLK	输出	时钟(同步突发模式使用)
FSMC_A[25:0]	输出	地址总线
FSMC_D[15:0]	输入/输出	双向数据总线
FSMC_NOE	输出	输出使能
FSMC_NWE	输出	写使能
FSMC_NWAIT	输入	NOR 闪存要求 FSMC 等待的信号
FSMC_NADV	输出	地址、数据线复用时作锁存信号

表 20.1.3.1 FSMC 信号线的的功能

在数电的课程中有介绍过存储器的知识，它是可以存储数据的器件。复杂的存储器为了存储更多的数据，常常通过地址线来管理数据存储的位置，这样只要先找到需要读写数据的位置，然后对进行数据读写的操作。由于存储器的这种数据和地址对应关系，采用 FSMC 这种专门硬件接口就能加快对存储器的数据访问。

STM32F407 的 FSMC 将外部存储器划分为固定大小为 256M 字节的四个存储块，FSMC 的外部设备地址映像如图 20.1.3.2 所示：

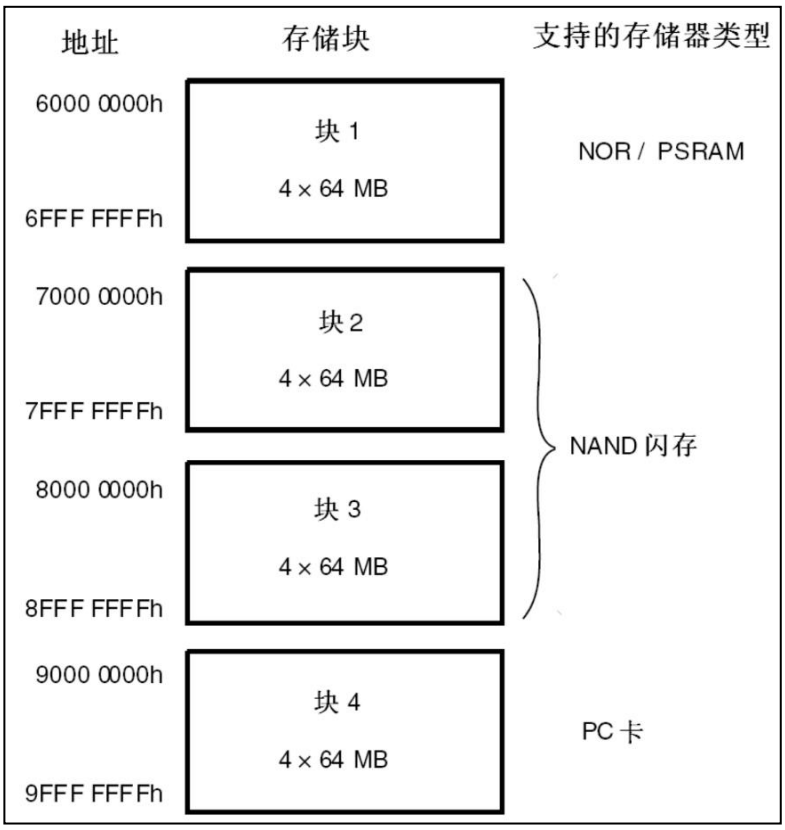


图20.1.3.2 FSMC存储块地址映像

从上图可以看出，FSMC 总共管理 1GB 空间，拥有 4 个存储块（Bank），FSMC 各 Bank 配置寄存器如表 20.1.3.2：

内部控制器	存储块	管理的地址范围	支持的设备类型	配置寄存器
NOR FLASH 控制器	Bank1	0x6000,0000~ 0x6FFF,FFFF	SRAM/ROM NOR FLASH PSRAM	FSMC_BCR1/2/3/4 FSMC_BTR1/2/2/3 FSMC_BWTR1/2/3/4
NAND FLASH PC CARD 控制器	Bank2	0x7000,0000~ 0x7FFF,FFFF	NAND FLASH	FSMC_PCR2/3/4 FSMC_SR2/3/4 FSMC_PMEM2/3/4 FSMC_PATT2/3/4 FSMC_PIO4
	Bank3	0x8000,0000~ 0x8FFF,FFFF		
	Bank4	0x9000,0000~ 0x9FFF,FFFF	PC Card	

表20.1.3.2 FSMC各Bank配置寄存器表

本章，我们用到的是块 1，所以在本章我们仅讨论块 1 的相关配置，其他块的配置，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 32 章（1191 页）的相关介绍。

STM32F4 的 FSMC 存储块 1（Bank1）被分为 4 个区，每个区管理 64M 字节空间，每个区都有独立的寄存器对所连接的存储器进行配置。Bank1 的 256M 字节空间可以通过 28 根地址线（HADDR[27:0]）寻址后访问。这里 HADDR 是内部 AHB 地址总线，其中 HADDR[25:0]来自外部存储器地址 FSMC_A[25:0]，而 HADDR[26:27]对 4 个区进行寻址。如表 20.1.3.3 所示：

Bank1 所选区	片选信号	地址范围	HADDR	
			[27:26]	[25:0]
第 1 区	FSMC_NE1	0x6000,0000~0x63FF,FFFF	00	FSMC_A[25:0]
第 2 区	FSMC_NE2	0x6400,0000~0x67FF,FFFF	01	
第 3 区	FSMC_NE3	0x6800,0000~0x6BFF,FFFF	10	
第 4 区	FSMC_NE4	0x6C00,0000~0x6FFF,FFFF	11	

表 20.1.3.3 Bank1 存储区选择表

表 20.1.3.3 中，我们要特别注意 HADDR[25:0]的对应关系：

当 Bank1 接的是 16 位宽度存储器的时候：HADDR[25:1]→FSMC_A[24:0]。

当 Bank1 接的是 8 位宽度存储器的时候：HADDR[25:0] →FSMC_A[25:0]。

不论外部接 8 位/16 位宽设备，FSMC_A[0]永远接在外部设备地址 A[0]。这里，TFTLCD 使用的是 16 位数据宽度，所以 HADDR[0]并没有用到，只有 HADDR[25:1]是有效的，对应关系变为：HADDR[25:1]→FSMC_A[24:0]，相当于右移了一位，具体来说，比如地址：0x7E，对应二进制是：01111110，此时 FSMC_A6 是 0 而不是 1，因为要右移一位，这里请特别注意。

另外，HADDR[27:26]的设置，是不需要我们干预的，比如：当你选择使用 Bank1 的第三个区，即使用 FSMC_NE3 来连接外部设备的时候，即对应了 HADDR[27:26]=10，我们要做的就是配置对应第 3 区的寄存器组，来适应外部设备即可。对于 NOR FLASH 控制器，主要是通过 FSMC_BCRx、FSMC_BTRx 和 FSMC_BWTRx 寄存器设置（其中 x=1~4，对应 4 个区）。通过这 3 个寄存器，可以设置 FSMC 访问外部存储器的时序参数，拓宽了可选用的外部存储器的速度范围。FSMC 的 NORFLASH 控制器支持同步和异步突发两种访问方式。选用同步突发访问方式时，FSMC 将 HCLK(系统时钟)分频后，发送给外部存储器作为同步时钟信号 FSMC_CLK。此时需要设置的时间参数有 2 个：

1，HCLK与FSMC_CLK的分频系数(CLKDIV)，可以为2～16分频；

2，同步突发访问中获得第1个数据所需要的等待延迟(DATLAT)。

对于异步突发访问方式，FSMC 主要设置 3 个时间参数：地址建立时间(ADDSET)、数据建立时间(DATAST)和地址保持时间(ADDHLD)。FSMC 综合了 SRAM / ROM、PSRAM 和 NOR Flash 产品的信号特点，定义了 4 种不同的异步时序模型。选用不同的时序模型时，需要设置不同的时序参数，如表 20.1.3.4 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、ADDHLK
同步突发		根据同步时钟FSMC_CK读取多个顺序单元的数据	CLKDIV、DATLAT

表20.1.3.4 NOR FLASH控制器支持的时序模型

在实际扩展时，根据选用存储器的特征确定时序模型，从而确定各时间参数与存储器读/写周期参数指标之间的计算关系；利用该计算关系和存储芯片数据手册中给定的参数指标，可计算出 FSMC 所需要的各时间参数，从而对时间参数寄存器进行合理的配置。

模式A支持独立的读写时序控制。这个对我们驱动TFTLCD来说非常有用，因为TFTLCD在读的时候，一般比较慢，而在写的时候可以比较快，如果读写用一样的时序，那么只能以读的时序为基准，从而导致写的速度变慢，或者在读数据的时候，重新配置FSMC的延时，在读操作完成的时候，再配置写的时序，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的写操作及读操作时序分别如图20.1.3.3和图20.1.3.4所示：

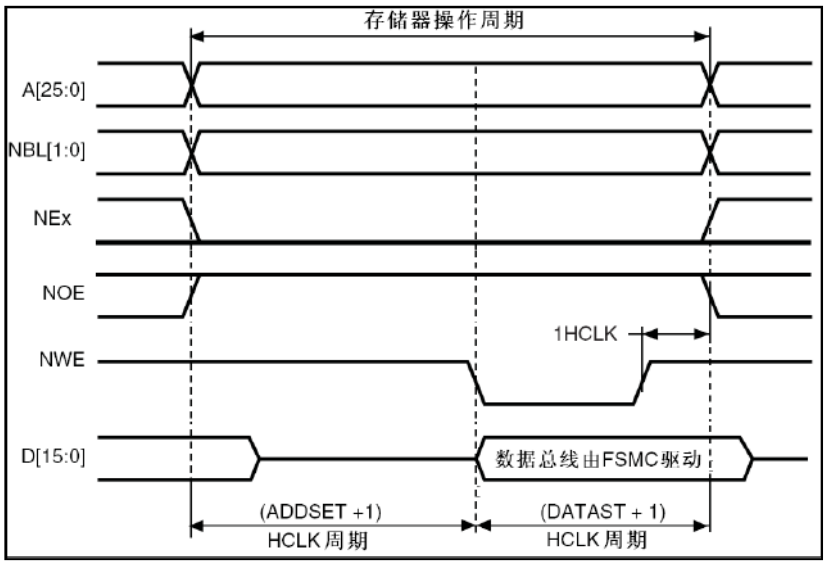


图20.1.3.3 模式A写操作时序

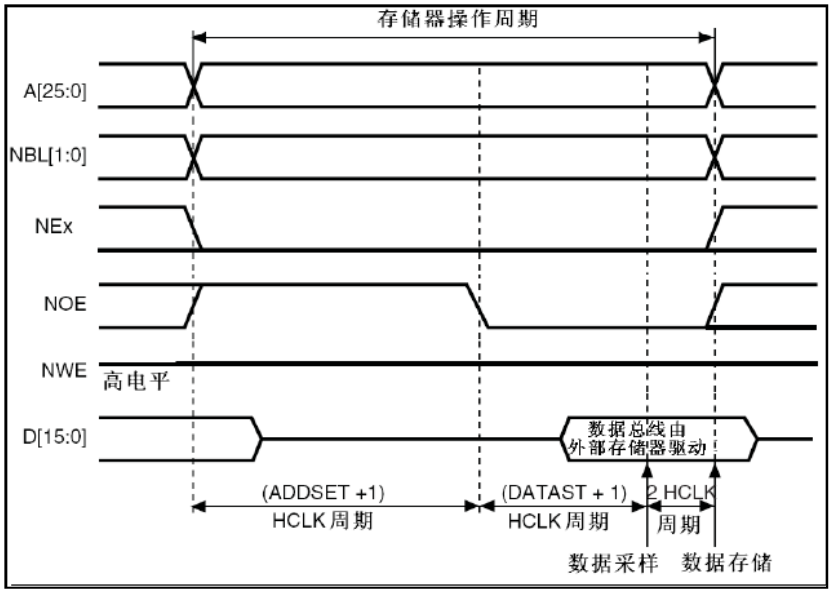


图20.1.3.4 模式A读操作时序图

图 20.1.3.3 和图 20.1.3.4 中的 ADDSET 与 DATAST，是通过不同的寄存器设置的。

以图 20.1.3.3 所示的写操作时序为例，该图表示一个存储器操作周期由地址建立周期(ADDSET)、数据建立周期(DATAST)组成。在地址建立周期中，数据建立周期期间 NWE 信号拉低发出写信号，接着 FSMC 把数据通过数据线传输到存储器中。注意 NWE 拉高后的那 1HCLK 是必要的，以保证数据线上的信号被准确采样。

读操作模式时序类似，区别是它的一个存储器操作周期由地址建立周期(ADDSET)和数据建立周期(DATAST)以及 2 个 HCLK 周期组成，且在数据建立周期期间地址线发出要访问的地址，数据掩码信号线指示出要读取地址的高、低字节部分，片选信号使能存储器芯片；地址建立周期结束后读使能信号线发出读使能信号，接着存储器通过数据信号线把目标数据传输给 FSMC，FSMC 把它交给内核。

当 FSMC 外设被配置成正常工作，并且外部接了 PSRAM，若向 0x60000000 地址写入数据如 0xABCD，FSMC 会自动在各信号线上产生相应的电平信号，写入数据。FSMC 会控制片选信号 NE1 输出低电平，相应的 PSRAM 芯片被片选激活，然后使用地址线 A[25:0]输出 0x60000000，在 NWE 写使能信号线上发出低电平的写使能信号，而要写入的数据信号 0xABCD 则从数据线 D[15:0]输出，然后数据就被保存到 PSRAM 中了。

到这里大家发现没有，之前讲的液晶控制器的 8080 并口模式与 FSMC 接口很像，区别是

FSMC 通过地址访问设备数据，并且可以自动控制相应电平，而 8080 方式则是直接控制，且没有地址线。对比图 20.1.2.2 和图 20.1.3.3，不难发现它们的相似点，我们概括如表 20.1.3.5：

FSMC(NOR/PSRAM)	功能	8080 信号线	功能
FSMC_NEx	片选信号	CSX	片选信号
FSMC_NWR	写使能	WRX	写使能
FSMC_NOE	读使能	RDX	读使能
FSMC_D[15:0]	数据信号	D[17:0]	数据信号
FSMC_A[25:0]	地址信号	D/CX	数据/命令选择

表20.1.3.5 FSMC（NOR/PSRAM）方式和8080并口对比

如果能用某种方式把 FSMC 的地址线和 8080 方式下的等效起来，那不就可以直接用 FSMC 等效 8080 方式操作 LCD 屏的显存了？FSMC 利用地址线访问数据，并自动设置地址线和相关控制信号线的电平，如果我们对命令操作和数据操作采用不同的地址来访问，同时使得操作数据时，地址线上的一个引脚的电平为高，操作命令时，同一个引脚的电平为低的话，就可以完美解决这个问题了！

F407 电机开发板把 TFT-LCD 就是用的 FSMC_NE4 做片选，把 RS 连接在 A10 上面的。我们来分析一下要让实现上面的通过地址自动切换命令和数据的实现方式。

首先 NOR/PSRAM 储块地址范围：0x6000 0000 ~ 0x6FFF FFFF，基地址是 0x6000 0000，每个存储块是 64Mb，那么这时候我们访问 LCD 的地址应该是第 4 个存储块，编号从 1 开始，访问 LCD 的起始地址就是 0x6000 0000 + (0x400 0000 * (x - 1)) = 0x6C00 0000，即从 0x6C00 0000 起的 64Mb 内存地址都可以去访问 LCD。

FSMC_A10 对应地址值：2^10 * 2 = 0x800（16 位模式时，参考表 20.1.3.3 及之后对 HADDR 和 FSMC 地址线对应关系的描述: HADDR[25:1]→FSMC_A[24:0]，所以这里计算时还需要乘 2）；则写命令时的地址为：0x6C00 0000 + 2^10 * 2 = 0x6C00 0800。写数据的地址就是使 FSMC_A10 为 0 的其它任意地址。（大家不要被地址访问的思路带进去了，以为接下来就是用 FSMC 的地址偏移来操作显存了，实际显存的操作还是归 MCU 屏管理。我们使能了 FSMC 功能后，就可以直接在我们设置的地址读写数据。实际上我们只用到了两个固定的地址：一个地址把 FSMC_A10 位置 1，另一个把该位置 0，但要保证这两个地址在各个 BANK 的管理范围内）。

STM32F407 的 FSMC 支持 8/16/32 位数据宽度，我们这里用到的 LCD 是 16 位宽度的，所以在设置的时候，选择 16 位宽就 OK 了。向这两个地址写的 16 进制数据会被直接送到数据线上，根据地址自动解析为命令或者数据，通过这样一个过程，我们就完成了用 FSMC 模拟 8080 并口的操作，最终完成对液晶控制器的控制。

20.1.4 FSMC 关联寄存器简介

接下来我们讲解一下 Bank1 的几个控制寄存器。

首先，我们介绍 SRAM/NOR 闪存片选控制寄存器：FSMC_BCRx（x=1~4），寄存器描述如图 20.1.4.1 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												CBURSTRW	Reserved			ASCYCWAIT	EXTMOD	WAITEN	WREN	WAITCFG	WRAPMOD	WAITPOL	BURSTEN	Reserved	FACCEN	MWID		MTYP		MUXEN	MBKEN
												rw				rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

位 14 **EXTMOD**：扩展模式使能 (Extended mode enable)。

FSMC 可对 FSMC_BWTR 寄存器中的写入时间进行配置，此配置由 EXTMOD 位使能，进而使读取和写入操作采用不同时序。

0：不考虑 FSMC_BWTR 寄存器中的值（复位后的默认值）

位 12 **WREN**：写入使能位 (Write enable bit)。

该位指示 FSMC 是否在存储区域内使能/禁止了写入操作：

0：FSMC 在存储区域内禁止了写入操作，如果进行写操作将报告 AHB 错误，

1：FSMC 在存储区域内使能了写入操作（复位后的默认值）。

位 5:4 **MWID**：存储器数据总线宽度 (Memory databus width)。

定义外部存储器器件宽度，对所有类型的存储器均有效。

00：8 位，

01：16 位（复位后的默认值），

10：保留，不使用，

11：保留，不使用。

位 3:2 **MTYP**：存储器类型 (Memory type)。

定义与相应存储区域相连的外部存储器类型：

00：SRAM、ROM（对于存储区域 2...4，复位后的默认值）

01：PSRAM（Cellular RAM：CRAM）

10：NOR Flash/OneNAND Flash （对于存储区域 1，复位后的默认值）

11：保留

位 0 **MBKEN**：存储区域使能位 (Memory bank enable bit)。

使能存储区域。复位后使能存储区域 1，其它存储区域均禁止。访问禁止的存储区域会引起 AHB 总线上的错误。

0：禁止相应的存储区域

1：使能相应的存储区域

图20.1.4.1 FSMC_BCRx寄存器描述

该寄存器我们在本章用到的设置有：EXTMOD、WREN、MWID、MTYP 和 MBKEN 这几个设置，我们将逐个介绍。

EXTMOD：扩展模式使能位，也就是是否允许读写不同的时序，很明显，我们本章需要读写不同的时序，故该位需要设置为 1。

WREN：写使能位。我们需要向 TFTLCD 写数据，故该位必须设置为 1。

MWID[1:0]：存储器数据总线宽度。00，表示 8 位数据模式；01 表示 16 位数据模式；10 和 11 保留。我们的 TFTLCD 是 16 位数据线，所以设置 WMID[1:0]=01。

MTYP[1:0]：存储器类型。00 表示 SRAM、ROM；01 表示 PSRAM；10 表示 NOR FLASH;11 保留。前面提到，我们把 TFTLCD 当成 SRAM 用，所以需要设置 MTYP[1:0]=00。

MBKEN：存储块使能位。这个容易理解，我们需要用到该存储块控制 TFTLCD，当然要使能这个存储块了。

接下来，我们看看 SRAM/NOR 闪存片选时序寄存器：FSMC_BTRx（x=1~4），该寄存器各位描述如图 20.1.4.2 所示：

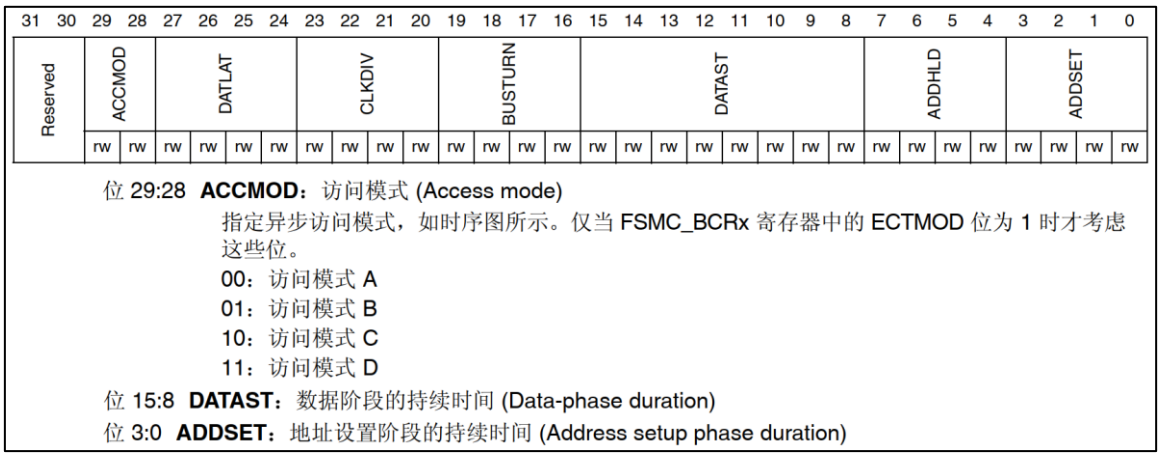


图20.1.4.2 FSMC_BTRx寄存器各位描述

这个寄存器包含了每个存储器块的控制信息，可以用于SRAM、ROM和NOR闪存存储器。如果FSMC_BCRx寄存器中设置了EXTMOD位，则有两个时序寄存器分别对应读(本寄存器)和写操作(FSMC_BWTRx寄存器)。因为我们要求读写分开时序控制，所以EXTMOD是使能了的，也就是本寄存器是读操作时序寄存器，控制读操作的相关时序。本章我们要用到的设置有：ACCMOD、DATAST和ADDSET这三个设置。

ACCMOD[1:0]：访问模式。00表示访问模式A；01表示访问模式B；10表示访问模式C；11表示访问模式D，本章我们用到模式A，故设置为00。

DATAST[7:0]：数据保持时间。0为保留设置，其他设置则代表保持时间为:DATAST个HCLK时钟周期，最大为255个HCLK周期。对ILI9341来说，其实就是RD低电平持续时间，一般为355ns。而一个HCLK时钟周期为6ns左右（1/168Mhz），为了兼容其他屏，我们这里设置DATAST为60，也就是60个HCLK周期，时间大约是360ns（未计算数据存储的2个HCLK时间，对9341来说超频了，但是实际上是可以正常使用的）。

ADDSET[3:0]：地址建立时间。其建立时间为：ADDSET个HCLK周期，最大为15个HCLK周期。对ILI9341来说，这里相当于RD高电平持续时间为90ns，我们设置ADDSET为15，即15*6ns=90ns。

最后，我们再来看看SRAM/NOR闪存写时序寄存器：FSMC_BWTRx（x=1~4），该寄存器各位描述如图20.1.4.3所示：

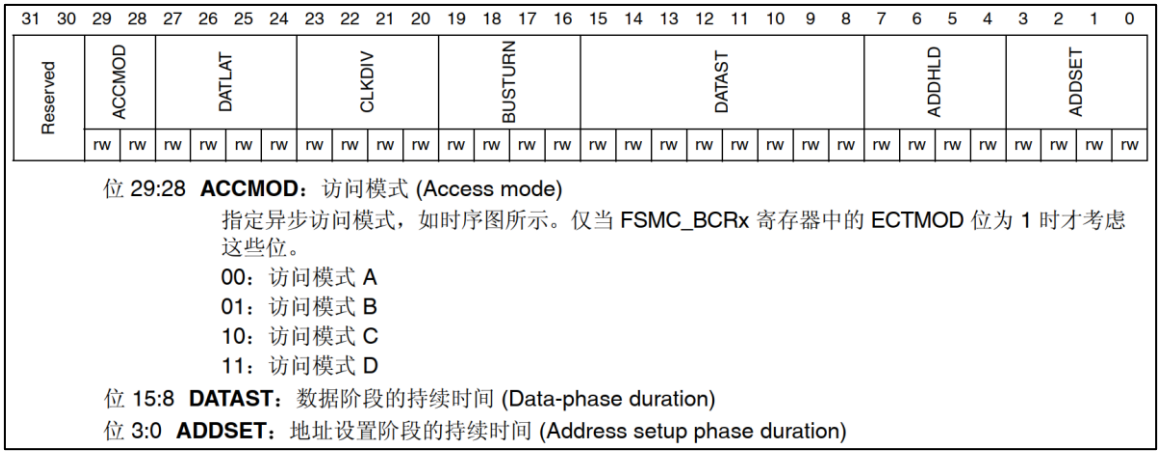


图20.1.4.3 FSMC_BWTRx寄存器各位描述

该寄存器在本章用作写操作时序控制寄存器，需要用到的设置同样是：ACCMOD、DATAST和 ADDSET 这三个设置。这三个设置的方法同 FSMC_BTRx 一模一样，只是这里对应的是写操作的时序，ACCMOD 设置同 FSMC_BTRx 一模一样，同样是选择模式 A，另外 DATAST 和 ADDSET 则对应低电平和高电平持续时间，对 ILI9341 来说，这两个时间只需要 15ns 就够了，比读操作快得多。所以我们这里设置 DATAST 为 2，即 3 个 HCLK 周期，时间约为 18ns。然后 ADDSET 设置为 3，即 3 个 HCLK 周期，时间约为 18ns。

至此，我们对 STM32F407 的 FSMC 介绍就差不多了，通过以上两个小节的了解，我们可以开始写 LCD 的驱动代码了。不过，这里还要给大家做下科普，在 MDK 的寄存器定义里面，并没有定义 FSMC_BCRx、FSMC_BTRx、FSMC_BWTRx 等单独的寄存器，而是将他们进行了一些组合。

FSMC_BCRx 和 FSMC_BTRx，组合成 BTCR[8]寄存器组，他们的对应关系如下：

BTCR[0]对应 FSMC_BCR1，
BTCR[1]对应 FSMC_BTR1，
BTCR[2]对应 FSMC_BCR2，
BTCR[3]对应 FSMC_BTR2，
BTCR[4]对应 FSMC_BCR3，
BTCR[5]对应 FSMC_BTR3，
BTCR[6]对应 FSMC_BCR4，
BTCR[7]对应 FSMC_BTR4，

FSMC_BWTRx 则组合成 BWTR[7]，他们的对应关系如下：

BWTR[0]对应 FSMC_BWTR1，
BWTR[2]对应 FSMC_BWTR2，
BWTR[4]对应 FSMC_BWTR3，
BWTR[6]对应 FSMC_BWTR4，

BWTR[1]、BWTR[3]和 BWTR[5]保留，没有用到。

20.2 硬件设计

1. 例程功能

使用开发板的 MCU 屏接口连接正点原子 TFTLCD 模块(仅限 MCU 屏模块)，实现 TFTLCD 模块的显示。通过把 LCD 模块插入底板上的 TFTLCD 模块接口，按下复位之后，就可以看到 LCD 模块不停的显示一些信息并不断切换底色。同时该实验会显示 LCD 驱动器的 ID，并且会在串口打印（按复位一次，打印一次）。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

TFTLCD 模块的电路见图 20.1.1.2，而开发板的 LCD 接口和正点原子 TFTLCD 模块直接可以对插，开发板上的 LCD 接口如图 20.2.1 所示：



图 25.2.1 TFTLCD 模块与开发板对接的 LCD 接口示意图

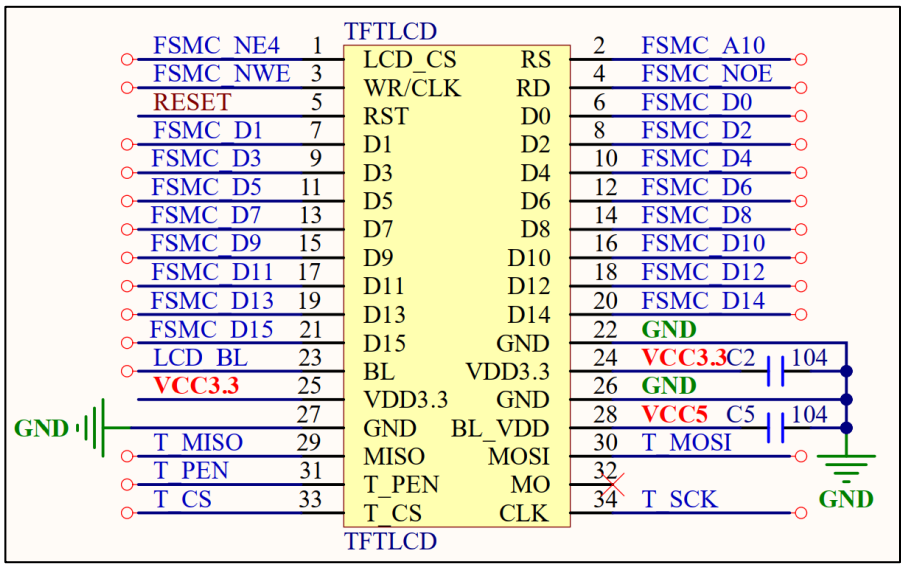


图 20.2.2 TFTLCD 模块与开发板的连接原理图

在硬件上，TFTLCD 模块与开发板的 IO 口对应关系如下：

LCD_BL(背光控制)对应 PH9；

LCD_CS 对应 PG12 即 FSMC_NE4；

LCD_RS 对应 PG0 即 FSMC_A6；

LCD_WR 对应 PD5 即 FSMC_NWE；

LCD_RD 对应 PD4 即 FSMC_NOE；

LCD_D[15:0]则直接连接在 FSMC_D15~FSMC_D0;

这些线的连接，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。

需要说明的是，开发板上设计的 TFT-LCD 模块插座，已经把模块模块的 RST 信号线直接接到我们开发板的复位脚上，所以不需要软件控制，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 LCD 的背光灯，因为 LCD 不会自发光，没有背光灯的情况下我们是看不到 LCD 上显示的内容的。所以，我们总共需要的 IO 口数目为 22 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等同于 D0~D15，这样理解起来就比较简单一点。

20.3 程序设计

20.3.1 FSMC 和 SRAM 的 HAL 库驱动

SRAM 和 FSMC 在 HAL 库中的驱动代码在 stm32f4xx_ll_fsmc.c/stm32f4xx_hal_sram.c 以及 stm32f4xx_ll_fsmc.h/stm32f4xx_hal_sram.h 中。

1. HAL_SRAM_Init 函数

SRAM 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_SRAM_Init(SRAM_HandleTypeDef *hsram,
    FSMC_NORSRAM_TimingTypeDef *Timing, FSMC_NORSRAM_TimingTypeDef *ExtTiming);
```

- **函数描述：**

  用于初始化 SRAM，注意这个函数不限制一定是 SRAM，只要时序类似，均可使用。前面说过，这里我们把 LCD 当作 SRAM 使用，因为他们时序类似。

- **函数形参：**

  形参 1 是 SRAM_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  FSMC_NORSRAM_TypeDef           *Instance;     /* 寄存器基地址 */
  FSMC_NORSRAM_EXTENDED_TypeDef  *Extended;     /* 扩展模式寄存器基地址 */
  FSMC_NORSRAM_InitTypeDef       Init;          /* SRAM 设备控制配置结构体 */
```

```
  HAL_LockTypeDef                  Lock;          /* SRAM 锁定对象 */
  __IO HAL_SRAM_StateTypeDef       State;         /* SRAM 设备访问状态 */
  DMA_HandleTypeDef                *hdma;         /* 指针 DMA 处理配置 */
} SRAM_HandleTypeDef;
```

1）Instance：指向 FSMC 寄存器基地址。我们直接写 FSMC_NORSRAM_DEVICE 即可，因为 HAL 库定义好了宏定义 FSMC_NORSRAM_DEVICE，也就是如果是 SRAM 设备，直接填写这个宏定义标识符即可。

2）Extended：指向 FSMC 扩展模式寄存器基地址，因为我们要配置的读写时序是不一样的，前面讲的 FSMC_BCRx 寄存器的 EXTMOD 位我们会配置为 1 允许读写不同的时序，所以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，这里设置为 FSMC_Bank1E_R 即可，同样 HAL 库定义了 FSMC_NORSRAM_EXTENDED_DEVICE，直接填写这个宏定义标识符即可。

3）Init：配置 FSMC 外接 SRAM 或者相同时序设备时的基本参数。

4）Lock：用于配置锁状态。

5）State：SRAM 设备访问状态。

6）hdma：用于配置关联 MAMA 句柄。

成员变量 Init 是 FSMC_NORSRAM_InitTypeDef 结构体指针类型，该变量才是真正用来设置 SRAM 控制接口参数的。下面详细了解这个结构体定义：

```
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;        /* 突发模式配置 */
  uint32_t WaitSignalPolarity;     /* 设置等待信号的极性 */
  uint32_t WrapMode;               /* 突发下存储器传输使能*/
  uint32_t WaitSignalActive;       /* 等待状态之前或等待状态期间 */
  uint32_t WriteOperation;         /* 存储器写使能 */
  uint32_t WaitSignal;             /* 使能或者禁止通过等待信号来插入等待状态 */
  uint32_t ExtendedMode;           /* 使能或者禁止使能扩展模式 */
  uint32_t AsynchronousWait;       /* 用于异步传输期间，使能或者禁止等待信号 */
  uint32_t WriteBurst;             /* 用于使能或者禁止异步的写突发操作 */
  uint32_t PageSize;               /* 设置页大小 */
}FSMC_NORSRAM_InitTypeDef;
```

NSBank 用来指定使用到的存储块区号，我们硬件设计时使用的存储块区号 4，所以选择值为 FSMC_NORSRAM_BANK4。

DataAddressMux 用来设置是否使能地址/数据复用,该变量仅对 NOR/PSRAM 有效，所以这里我们选择不使能地址/数据复用值 FSMC_DATA_ADDRESS_MUX_DISABLE 即可。

MemoryType 用来设置存储器类型，这里我们把 LCD 当 SRAM 使用，所以设置为 FSMC_MEMORY_TYPE_SRAM 即可。

MemoryDataWidth 用来设置存储器数据总线宽度，可选 8 位还是 16 位，这里我们选择 16 位数据宽度 FSMC_NORSRAM_MEM_BUS_WIDTH_16。

WriteOperation 用来设置存储器写使能，也就是是否允许写入。毫无疑问我们会进行存储器写操作，所以这里设置为 FSMC_WRITE_OPERATION_ENABLE。

ExtendedMode 用来设置是否使能扩展模式，也就是是否允许读写使用不同时序，前面讲解过本实验读写采用不同时序，所以设置值为使能值 FSMC_EXTENDED_MODE_ENABLE。

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。

形参 2 Timing 和**形参 3** ExtTiming 都是 FSMC_NORSRAM_TimingTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  uint32_t AddressSetupTime;             /* 地址建立时间 */
```

```
  uint32_t AddressHoldTime;              /* 地址保持时间 */
  uint32_t DataSetupTime;                /* 数据建立时间 */
  uint32_t BusTurnAroundDuration;        /* 总线周转阶段的持续时间 */
  uint32_t CLKDivision;                  /* CLK 时钟输出信号的周期 */
  uint32_t DataLatency;                  /* 同步突发 NOR FLASH 的数据延迟 */
  uint32_t AccessMode;                   /* 异步模式配置 */
}FSMC_NORSRAM_TimingTypeDef;
```

对于本实验，读速度比写速度慢得多，因此读写时序不一样，所以对于 Timing 和 ExtTiming 要设置了不同的值，其中 Timing 设置写时序参数，ExtTiming 设置读时序参数。

下面解析一下结构体的成员变量：

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，HAL 库允许其取值范围为 FSMC_ACCESS_MODE_A、FSMC_ACCESS_MODE_B、FSMC_ACCESS_MODE_C 和 FSMC_ACCESS_MODE_D，这里我们用的是异步模式 A，所以取值为 FSMC_ACCESS_MODE_A。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

  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);
```

2. FSMC_NORSRAM_Extended_Timing_Init 函数

FSMC_NORSRAM_Extended_Timing_Init 函数是初始化扩展时序模式函数。其声明如下：

```
HAL_StatusTypeDef  FSMC_NORSRAM_Extended_Timing_Init(
    FSMC_NORSRAM_EXTENDED_TypeDef *Device, FSMC_NORSRAM_TimingTypeDef *Timing,
     uint32_t Bank, uint32_t ExtendedMode);
```

- **函数描述：**

  该函数用于初始化扩展时序模式。

- **函数形参：**

  形参 1 是 FSMC_NORSRAM_EXTENDED_TypeDef 结构体类型指针变量，扩展模式寄存器基地址选择。

  形参 2 是 FSMC_NORSRAM_TimingTypeDef 结构体类型指针变量，可以是读或者写时序结构体。

  形参 3 是储存区块号。

  形参 4 是使能或者禁止扩展模式。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

  该函数我们用于重新配置写或者读时序。

FSMC 驱动 LCD 显示配置步骤

1）使能 FSMC 和相关 GPIO 时钟，并设置好 GPIO 工作模式

我们通过 FSMC 控制 LCD，所以先需要使能 FSMC 以及相关 GPIO 口的时钟，并设置好 GPIO 的工作模式。

2）设置 FSMC 参数

这里我们需要设置 FSMC 的相关访问参数（数据位宽、访问时序、工作模式等），以匹配液晶驱动 IC，这里我们通过 HAL_SRAM_Init 函数完成 FSMC 参数配置，详见本例程源码。

3）初始化 LCD

由于我们例程兼容了很多种液晶驱动 IC，所以先要读取对应 IC 的驱动型号，然后根据不同的 IC 型号来调用不同的初始化函数，完成对 LCD 的初始化。

注意：这些初始化函数里面的代码，都是由 LCD 厂家提供，一般不需要改动，也不需要深究，我们直接照抄即可。

4）实现 LCD 画点&读点函数

在初始化 LCD 完成以后，我们就可以控制 LCD 显示了，而最核心的一个函数，就是画点和读点函数，只要实现这两个函数，后续的各种 LCD 操作函数，都可以基于这两个函数实现。

5）实现其他 LCD 操作函数

在完成画点和读点两个最基础的 LCD 操作函数以后，我们就可以基于这两个函数实现各种 LCD 操作函数了，比如画线、画矩形、显示字符、显示字符串、显示数字等，如果不够用还可以根据自己需要来添加。详见本例程源码。

20.3.2 程序流程图

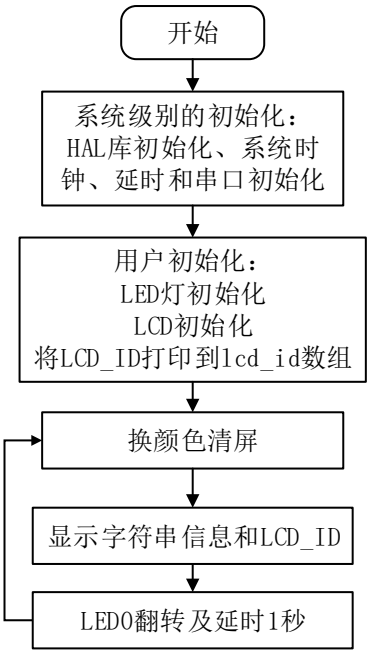


图 20.3.2.1 TFTLCD（MCU 屏）实验程序流程图

20.3.2 程序解析

1. LCD 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。液晶（LCD）驱动源码包括四个文件：lcd.c、lcd.h、lcd_ex.c 和 lcdfont.h。

lcd.c 和 lcd.h 文件是驱动函数和引脚接口宏定义以及函数声明等。lcd_ex.c 存放各个 LCD 驱动 IC 的寄存器初始化部分代码，是 lcd.c 文件的补充文件，起到简化 lcd.c 文件的作用。lcdfont.h 头文件存放了 4 种字体大小不一样的 ASCII 字符集（12*12、16*16、24*24 和 32*32）。

下面我们还是先介绍 lcd.h 文件，首先是 LCD 的引脚定义：

```
/* LCD RST/WR/RD/BL/CS/RS 引脚 定义
 * LCD_D0~D15,由于引脚太多,就不在这里定义了,直接在 lcd_init 里面修改.所以在移植的时候,除了
 * 改这 6 个 IO 口, 还得改 LCD_Init 里面的 D0~D15 所在的 IO 口.
 */

/* RESET 和系统复位脚共用 所以这里不用定义 RESET 引脚 */

#define LCD_WR_GPIO_PORT            GPIOD
#define LCD_WR_GPIO_PIN             GPIO_PIN_5
```

```
/* 所在 IO 口时钟使能 */
#define LCD_WR_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOD_CLK_ENABLE();}while(0)

#define LCD_RD_GPIO_PORT            GPIOD
#define LCD_RD_GPIO_PIN             GPIO_PIN_4
/* 所在 IO 口时钟使能 */
#define LCD_RD_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOD_CLK_ENABLE();}while(0)
/* 所在 IO 口时钟使能 */
#define LCD_BL_GPIO_PORT            GPIOH
#define LCD_BL_GPIO_PIN             GPIO_PIN_9
/* 背光所在 IO 口时钟使能 */
#define LCD_BL_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOB_CLK_ENABLE();}while(0)

/* LCD_CS(需要根据 LCD_FSMC_NEX 设置正确的 IO 口) 和 LCD_RS(需要根据 LCD_FSMC_AX 设置正确
的 IO 口) 引脚 定义 */
#define LCD_CS_GPIO_PORT            GPIOG
#define LCD_CS_GPIO_PIN             GPIO_PIN_12
/* 所在 IO 口时钟使能 */
#define LCD_CS_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0)

#define LCD_RS_GPIO_PORT            GPIOG
#define LCD_RS_GPIO_PIN             GPIO_PIN_0
/* 所在 IO 口时钟使能 */
#define LCD_RS_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOG_CLK_ENABLE();}while(0)
```

第一部分的宏定义是 LCD WR/RD/BL/CS/RS/DATA 引脚定义，需要注意的是：LCD 的 RST 引脚和系统复位脚连接在一起，所以不用单独使用一个 IO 口（节省一个 IO 口）。而 DATA 引脚直接用的是 FSMC_D[x]引脚，具体可以查看前面的描述。

下面介绍我们在 lcd.h 里面定义的一个重要的结构体：

```
/* LCD 重要参数集 */
typedef struct
{
    uint16_t width;                 /* LCD 宽度 */
    uint16_t height;                /* LCD 高度 */
    uint16_t id;                    /* LCD ID */
    uint8_t dir;                    /* 横屏还是竖屏控制：0，竖屏；1，横屏。 */
    uint16_t wramcmd;               /* 开始写 gram 指令 */
    uint16_t setxcmd;               /* 设置 x 坐标指令 */
    uint16_t setycmd;               /* 设置 y 坐标指令 */
} _lcd_dev;
extern _lcd_dev lcddev; /* 管理 LCD 重要参数 */
/* LCD 的画笔颜色和背景色 */
extern uint32_t  g_point_color;     /* 默认红色 */
extern uint32_t  g_back_color;      /* 背景颜色.默认为白色 */
```

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。最后声明_lcd_dev 结构体类型变量 lcddev，lcddev 在 lcd.c 中定义。

紧接着就是 g_point_color 和 g_back_color 变量的声明，它们也是在 lcd.c 中被定义。g_point_color 变量用于保存 LCD 的画笔颜色，g_back_color 则是保存 LCD 的背景色。

下面是 LCD 背光控制 IO 口的宏定义：

```
/* LCD 背光控制 */
#define LCD_BL(x)   do{ x ? \
        HAL_GPIO_WritePin(LCD_BL_GPIO_PORT, LCD_BL_GPIO_PIN, GPIO_PIN_SET) : \
        HAL_GPIO_WritePin(LCD_BL_GPIO_PORT, LCD_BL_GPIO_PIN, GPIO_PIN_RESET); \
                }while(0)
```

本实验，我们用到 FSMC 驱动 LCD，通过前面的介绍，我们知道 TFTLCD 的 RS 接在 FSMC 的 A10 上面，CS 接在 FSMC_NE4 上，并且是 16 位数据总线。即我们使用的是 FSMC 存储器 1 的第 4 区，我们定义如下 LCD 操作结构体（在 lcd.h 里面定义）：

```
/* LCD 地址结构体 */
```

```
typedef struct
{
    volatile uint16_t LCD_REG;
    volatile uint16_t LCD_RAM;
} LCD_TypeDef;

/* LCD_BASE 的详细解算方法:
 * 我们一般使用 FSMC 的块 1(BANK1)来驱动 TFTLCD 液晶屏(MCU 屏), 块 1 地址范围总大小为 256MB,
 * 均分成 4 块:
 * 存储块 1(FSMC_NE1)地址范围: 0x6000 0000 ~ 0x63FF FFFF
 * 存储块 2(FSMC_NE2)地址范围: 0x6400 0000 ~ 0x67FF FFFF
 * 存储块 3(FSMC_NE3)地址范围: 0x6800 0000 ~ 0x6BFF FFFF
 * 存储块 4(FSMC_NE4)地址范围: 0x6C00 0000 ~ 0x6FFF FFFF
 *
 * 我们需要根据硬件连接方式选择合适的片选(连接 LCD_CS)和地址线(连接 LCD_RS)
 * F407 电机开发板使用 FSMC_NE4 连接 LCD_CS,FSMC_A10 连接 LCD_RS,16 位数据线,计算方法如下:
 * 首先 FSMC_NE4 的基地址为: 0x6C00 0000;    NEx 的基址为(x=1/2/3/4):
 *                                          0x6000 0000 + (0x400 0000 * (x - 1))
 * FSMC_A10 对应地址值: 2^10 * 2 = 0x800;  FSMC_Ay 对应的地址为(y = 0 ~ 25): 2^y * 2
 *
 * LCD->LCD_REG,对应 LCD_RS = 0(LCD 寄存器); LCD->LCD_RAM,对应 LCD_RS = 1(LCD 数据)
 * 则 LCD->LCD_RAM 的地址为: 0x6C00 0000 + 2^10 * 2 = 0x6C00 0800
 *    LCD->LCD_REG 的地址可以为 LCD->LCD_RAM 之外的任意地址.
 * 由于我们使用结构体管理 LCD_REG 和 LCD_RAM(REG 在前,RAM 在后,均为 16 位数据宽度)
 * 因此 结构体的基地址(LCD_BASE) = LCD_RAM - 2 = 0x6C00 0800 - 2
 *
 * 更加通用的计算公式为((片选脚 FSMC_NEx)x=1/2/3/4, (RS 接地址线 FSMC_Ay)y=0~25):
 *         LCD_BASE = (0x6000 0000 + (0x400 0000 * (x - 1))) | (2^y * 2 - 2)
 *         等效于(使用移位操作)
 *         LCD_BASE = (0x6000 0000 + (0x400 0000 * (x - 1))) | ((1 << y) * 2 -2)
 */
#define LCD_BASE   (uint32_t)((0x60000000 + (0x4000000 * (LCD_FSMC_NEX - 1))) |
                              (((1 << LCD_FSMC_AX) * 2) - 2))
#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 的存储块 4 的寻址范围为 0x6C000000~0x6FFFFFFF，我们需要在这个地址范围内找到两个地址，实现对 RS 位（FSMC_A10 位）的 0 和 1 的控制。这两个地址的取值方法，我们在前面的 20.1.3 的末尾已经详细说明了。为了方便控制和节省内存，我们使这两个地址变成相邻的两个 16 进制指针，这样就可以用前面定义的 LCD_TypeDef 来管理这两个地址了。

根据我们的算法和定义，我们将这个地址强制转换为 LCD_TypeDef 结构体地址，那么可以得到 LCD->LCD_REG 的地址就是 0x6C00 07FE，对应 A10 的状态为 0(即 RS=0)，而 LCD->LCD_RAM 的地址就是 0x6C00 0800(结构体地址自增),对应 A10 的状态为 1(即 RS=1)。

所以，有了这个定义，当我们要往 LCD 写命令/数据的时候，可以这样写：

```
LCD->LCD_REG=CMD;   /* 写命令 */
LCD->LCD_RAM=DATA;  /* 写数据 */
```

而读的时候反过来操作就可以了，如下所示：

```
CMD= LCD->LCD_REG;   /* 读 LCD 寄存器 */
DATA = LCD->LCD_RAM; /* 读 LCD 数据 */
```

这其中，CS、WR、RD 和 IO 口方向都是由 FSMC 硬件自动控制，不需要我们手动设置了。

最后是一些其他的宏定义，包括 LCD 扫描方向和颜色，以及 SSD1963 相关配置参数。

下面开始对 lcd.c 文件介绍，先看 LCD 初始化函数，其定义如下：

```
/**
 * @brief       初始化 LCD
 *   @note      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)
 *
 * @param       无
 * @retval      无
 */
```

```
void lcd_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;
    FSMC_NORSRAM_TimingTypeDef fsmc_read_handle;
    FSMC_NORSRAM_TimingTypeDef fsmc_write_handle;

    LCD_CS_GPIO_CLK_ENABLE();   /* LCD_CS 脚时钟使能 */
    LCD_WR_GPIO_CLK_ENABLE();   /* LCD_WR 脚时钟使能 */
    LCD_RD_GPIO_CLK_ENABLE();   /* LCD_RD 脚时钟使能 */
    LCD_RS_GPIO_CLK_ENABLE();   /* LCD_RS 脚时钟使能 */
    LCD_BL_GPIO_CLK_ENABLE();   /* LCD_BL 脚时钟使能 */

    gpio_init_struct.Pin = LCD_CS_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;              /* 推挽复用 */
    gpio_init_struct.Pull = GPIO_PULLUP;                  /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;        /* 高速 */
    gpio_init_struct.Alternate = GPIO_AF12_FSMC;          /* 复用为 FSMC */
    HAL_GPIO_Init(LCD_CS_GPIO_PORT, &gpio_init_struct);   /* 初始化 LCD_CS 引脚 */

    gpio_init_struct.Pin = LCD_WR_GPIO_PIN;
    HAL_GPIO_Init(LCD_WR_GPIO_PORT, &gpio_init_struct);   /* 初始化 LCD_WR 引脚 */

    gpio_init_struct.Pin = LCD_RD_GPIO_PIN;
    HAL_GPIO_Init(LCD_RD_GPIO_PORT, &gpio_init_struct);   /* 初始化 LCD_RD 引脚 */

    gpio_init_struct.Pin = LCD_RS_GPIO_PIN;
    HAL_GPIO_Init(LCD_RS_GPIO_PORT, &gpio_init_struct);   /* 初始化 LCD_RS 引脚 */

    gpio_init_struct.Pin = LCD_BL_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;          /* 推挽输出 */
    HAL_GPIO_Init(LCD_BL_GPIO_PORT, &gpio_init_struct);

    g_sram_handle.Instance = FSMC_NORSRAM_DEVICE;
    g_sram_handle.Extended = FSMC_NORSRAM_EXTENDED_DEVICE;

    g_sram_handle.Init.NSBank = FSMC_NORSRAM_BANK4;       /* 使用 NE4 */
    /* 地址/数据线不复用 */
    g_sram_handle.Init.DataAddressMux = FSMC_DATA_ADDRESS_MUX_DISABLE;
    /* 16 位数据宽度 */
    g_sram_handle.Init.MemoryDataWidth = FSMC_NORSRAM_MEM_BUS_WIDTH_16;
    /* 是否使能突发访问,仅对同步突发存储器有效,此处未用到 */
    g_sram_handle.Init.BurstAccessMode = FSMC_BURST_ACCESS_MODE_DISABLE;
    /* 等待信号的极性,仅在突发模式访问下有用 */
    g_sram_handle.Init.WaitSignalPolarity = FSMC_WAIT_SIGNAL_POLARITY_LOW;
    /* 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 */
    g_sram_handle.Init.WaitSignalActive = FSMC_WAIT_TIMING_BEFORE_WS;
    /* 存储器写使能 */
    g_sram_handle.Init.WriteOperation = FSMC_WRITE_OPERATION_ENABLE;
    /* 等待使能位,此处未用到 */
    g_sram_handle.Init.WaitSignal = FSMC_WAIT_SIGNAL_DISABLE;
    /* 读写使用不同的时序 */
    g_sram_handle.Init.ExtendedMode = FSMC_EXTENDED_MODE_ENABLE;
    /* 是否使能同步传输模式下的等待信号,此处未用到 */
    g_sram_handle.Init.AsynchronousWait = FSMC_ASYNCHRONOUS_WAIT_DISABLE;
    /* 禁止突发写 */
    g_sram_handle.Init.WriteBurst = FSMC_WRITE_BURST_DISABLE;

    /* FSMC 读时序控制寄存器 */
    /* 地址建立时间(ADDSET)为 15 个 fsmc_ker_ck(1/168=6)即 6*15=90ns */
    fsmc_read_handle.AddressSetupTime = 0x0F;
    /* 地址保持时间(ADDHLD) 模式 A 是没有用到 */
    fsmc_read_handle.AddressHoldTime = 0x00;
```

```
    /* 数据保存时间(DATAST)为 60 个 fsmc_ker_ck=6*60=360ns */
    fsmc_read_handle.DataSetupTime = 60;
    /* 因为液晶驱动 IC 的读数据的时候,速度不能太快,尤其是个别奇葩芯片 */
    fsmc_read_handle.AccessMode = FSMC_ACCESS_MODE_A;   /* 模式 A */

    /* FSMC 写时序控制寄存器 */
    /* 地址建立时间(ADDSET)为 9 个 fsmc_ker_ck=6*9=54ns */
    fsmc_write_handle.AddressSetupTime = 9;
    /* 地址保持时间(ADDHLD) 模式 A 是没有用到 */
    fsmc_write_handle.AddressHoldTime = 0x00;
    /* 数据保存时间(DATAST)为 9 个 fsmc_ker_ck=6*9=54ns */
    fsmc_write_handle.DataSetupTime = 9;
    /* 注意：某些液晶驱动 IC 的写信号脉宽，最少也得 50ns */
    fsmc_write_handle.AccessMode = FSMC_ACCESS_MODE_A;  /* 模式 A */

    HAL_SRAM_Init(&g_sram_handle, &fsmc_read_handle, &fsmc_write_handle);
    delay_ms(50);

    /* 尝试 9341 ID 的读取 */
    lcd_wr_regno(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)    /* 不是 9341 , 尝试看看是不是 ST7789 */
    {
        lcd_wr_regno(0x04);
        lcddev.id = lcd_rd_data();      /* dummy read */
        lcddev.id = lcd_rd_data();      /* 读到 0x85 */
        lcddev.id = lcd_rd_data();      /* 读取 0x85 */
        lcddev.id <<= 8;
        lcddev.id |= lcd_rd_data();     /* 读取 0x52 */

        if (lcddev.id == 0x8552)        /* 将 8552 的 ID 转换成 7789 */
        {
            lcddev.id = 0x7789;
        }

        if (lcddev.id != 0x7789)        /* 也不是 ST7789, 尝试是不是 NT35310 */
        {
            lcd_wr_regno(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 ST7796 */
            {
                lcd_wr_regno(0XD3);
                lcddev.id = lcd_rd_data();  /* dummy read */
                lcddev.id = lcd_rd_data();  /* 读到 0X00 */
                lcddev.id = lcd_rd_data();  /* 读取 0X77 */
                lcddev.id <<= 8;
                lcddev.id |= lcd_rd_data(); /* 读取 0X96 */

                if (lcddev.id != 0x7796)    /* 也不是 ST7796,尝试看看是不是 NT35510 */
                {
                    /* 发送密钥（厂家提供） */
                    lcd_write_reg(0xF000, 0x0055);
```

```
                lcd_write_reg(0xF001, 0x00AA);
                lcd_write_reg(0xF002, 0x0052);
                lcd_write_reg(0xF003, 0x0008);
                lcd_write_reg(0xF004, 0x0001);

                lcd_wr_regno(0xC500);       /* 读取 ID 低八位 */
                lcddev.id = lcd_rd_data();  /* 读回 0x80 */
                lcddev.id <<= 8;

                lcd_wr_regno(0xC501);       /* 读取 ID 高八位 */
                lcddev.id |= lcd_rd_data(); /* 读回 0x00 */

/* 等待 5ms, 因为 0XC501 指令对 1963 来说就是软件复位指令, 等待 5ms 让 1963 复位完成再操作 */
                delay_ms(5);

                /* 也不是 NT5510,尝试看看是不是 ILI9806 */
                if (lcddev.id != 0x5510)
                {
                    lcd_wr_regno(0XD3);
                    lcddev.id = lcd_rd_data();  /* dummy read */
                    lcddev.id = lcd_rd_data();  /* 读回 0X00 */
                    lcddev.id = lcd_rd_data();  /* 读回 0X98 */
                    lcddev.id <<= 8;
                    lcddev.id |= lcd_rd_data(); /* 读回 0X06 */

                    /* 也不是 ILI9806,尝试看看是不是 SSD1963 */
                    if (lcddev.id != 0x9806)
                    {
                        lcd_wr_regno(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 */
                        /* SSD1963 读回的 ID 是 5761H,为方便区分,我们强制设置为 1963 */
                        if (lcddev.id == 0x5761) lcddev.id = 0x1963;
                    }
                }
            }
        }
    }

    /* 特别注意, 如果在 main 函数里面屏蔽串口 1 初始化, 则会卡死在 printf
     * 里面(卡死在 f_putc 函数), 所以, 必须初始化串口 1, 或者屏蔽掉下面
     * 这行 printf 语句 !!!!!!!
     */
    printf("LCD ID:%x\r\n", lcddev.id); /* 打印 LCD ID */

    if (lcddev.id == 0x7789)
    {
        lcd_ex_st7789_reginit();    /* 执行 ST7789 初始化 */
    }
    else if (lcddev.id == 0x9341)
    {
        lcd_ex_ili9341_reginit();   /* 执行 ILI9341 初始化 */
    }
    else if (lcddev.id == 0x5310)
    {
        lcd_ex_nt35310_reginit();   /* 执行 NT35310 初始化 */
    }
    else if (lcddev.id == 0x7796)
    {
        lcd_ex_st7796_reginit();    /* 执行 ST7796 初始化 */
```

```
    }
    else if (lcddev.id == 0x5510)
    {
        lcd_ex_nt35510_reginit();   /* 执行 NT35510 初始化 */
    }
    else if (lcddev.id == 0x9806)
    {
        lcd_ex_ili9806_reginit();   /* 执行 ILI9806 初始化 */
    }
    else if (lcddev.id == 0x1963)
    {
        lcd_ex_ssd1963_reginit();   /* 执行 SSD1963 初始化 */
        lcd_ssd_backlight_set(100); /* 背光设置为最亮 */
    }

    /* 初始化完成以后,提速 */

    /* 7789/9341/1963 提速 */
    if (lcddev.id == 0x7789 || lcddev.id == 0x9341 || lcddev.id == 0x1963)
    {
        /* 重新配置写时序控制寄存器的时序 */
        /* 地址建立时间(ADDSET)为 3 个 fsmc_ker_ck=6*3=18ns */
        fsmc_write_handle.AddressSetupTime = 3;
        /* 数据保持时间(DATAST)为 3 个 fsmc_ker_ck=6*3=18ns */
        fsmc_write_handle.DataSetupTime = 3;
        FSMC_NORSRAM_Extended_Timing_Init(g_sram_handle.Extended,
                                &fsmc_write_handle, g_sram_handle.Init.NSBank,
                                g_sram_handle.Init.ExtendedMode);
    }
    /* 如果是这几个 IC, 则设置 WR 时序为最快 */
    else if (lcddev.id == 0x5310 || lcddev.id == 0x7796 ||
             lcddev.id == 0x5510 || lcddev.id == 0x9806)
    {
        /* 重新配置写时序控制寄存器的时序 */
        /* 地址建立时间(ADDSET)为 2 个 fsmc_ker_ck=6*2=12ns */
        fsmc_write_handle.AddressSetupTime = 2;
        /* 数据保持时间(DATAST)为 2 个 fsmc_ker_ck=6*2=12ns */
        fsmc_write_handle.DataSetupTime = 2;
        FSMC_NORSRAM_Extended_Timing_Init(g_sram_handle.Extended,
                                &fsmc_write_handle, g_sram_handle.Init.NSBank,
                                g_sram_handle.Init.ExtendedMode);
    }

    lcd_display_dir(0); /* 默认为竖屏 */
    LCD_BL(1);          /* 点亮背光 */
    lcd_clear(WHITE);
}
```

该函数先对 FSMC 相关 IO 进行初始化，然后使用 HAL_SRAM_Init 函数初始化 FSMC 控制器，同时我们使用 HAL_SRAM_MspInit 回调函数来初始化相应的 IO 口，最后读取 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，这样提高了整个程序的通用性。为了简化 lcd.c 的初始化程序，不同控制 IC 的芯片对应的初始化程序（如：lcd_ex_st7789_reginit()、lcd_ex_ili9341_reginit()等）我们放在 lcd_ex.c 文件中，这些初始化代码完成对 LCD 寄存器的初始化，由 LCD 厂家提供，一般是不需要做任何修改的，我们直接调用就可以了。

下面是 6 个简单，但是很重要的函数：

```
/**
 * @brief       LCD 写数据
 * @param       data: 要写入的数据
 * @retval      无
 */
void lcd_wr_data(volatile uint16_t data)
{
```

```
    data = data;            /* 使用-O2 优化的时候,必须插入的延时 */
    LCD->LCD_RAM = data;
}

/**
 * @brief       LCD 写寄存器编号/地址函数
 * @param       regno: 寄存器编号/地址
 * @retval      无
 */
void lcd_wr_regno(volatile uint16_t regno)
{
    regno = regno;          /* 使用-O2 优化的时候,必须插入的延时 */
    LCD->LCD_REG = regno;   /* 写入要写的寄存器序号 */
}

/**
 * @brief       LCD 写寄存器
 * @param       regno:寄存器编号/地址
 * @param       data:要写入的数据
 * @retval      无
 */
void lcd_write_reg(uint16_t regno, uint16_t data)
{
    LCD->LCD_REG = regno;   /* 写入要写的寄存器序号 */
    LCD->LCD_RAM = data;    /* 写入数据 */
}

/**
 * @brief       LCD 延时函数,仅用于部分在 mdk -O1 时间优化时需要设置的地方
 * @param       t:延时的数值
 * @retval      无
 */
static void lcd_opt_delay(uint32_t i)
{
    while (i--); /* 使用 AC6 时空循环可能被优化,可使用 while(1) __asm volatile(""); */
}

/**
 * @brief       LCD 读数据
 * @param       无
 * @retval      读取到的数据
 */
static uint16_t lcd_rd_data(void)
{
    volatile uint16_t ram;  /* 防止被优化 */
    lcd_opt_delay(2);
    ram = LCD->LCD_RAM;
    return ram;
}

/**
 * @brief       准备写 GRAM
 * @param       无
 * @retval      无
 */
void lcd_write_ram_prepare(void)
{
    LCD->LCD_REG = lcddev.wramcmd;
}
```

因为 FSMC 自动控制了 WR/RD/CS 等这些信号，所以这 6 个函数实现起来都非常简单，我们就不多说，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。

下面要介绍的函数是坐标设置函数，该函数代码如下：

```
/**
 * @brief       设置光标位置(对 RGB 屏无效)
 * @param       x,y: 坐标
 * @retval      无
 */
void lcd_set_cursor(uint16_t x, uint16_t y)
{
    if (lcddev.id == 0x1963)
    {
        if (lcddev.dir == 0)    /* 竖屏模式, x 坐标需要变换 */
        {
            x = lcddev.width - 1 - x;
            lcd_wr_regno(lcddev.setxcmd);
            lcd_wr_data(0);
            lcd_wr_data(0);
            lcd_wr_data(x >> 8);
            lcd_wr_data(x & 0xFF);
        }
        else                    /* 横屏模式 */
        {
            lcd_wr_regno(lcddev.setxcmd);
            lcd_wr_data(x >> 8);
            lcd_wr_data(x & 0xFF);
            lcd_wr_data((lcddev.width - 1) >> 8);
            lcd_wr_data((lcddev.width - 1) & 0xFF);
        }

        lcd_wr_regno(lcddev.setycmd);
        lcd_wr_data(y >> 8);
        lcd_wr_data(y & 0xFF);
        lcd_wr_data((lcddev.height - 1) >> 8);
        lcd_wr_data((lcddev.height - 1) & 0xFF);

    }
    else if (lcddev.id == 0x5510)
    {
        lcd_wr_regno(lcddev.setxcmd);
        lcd_wr_data(x >> 8);
        lcd_wr_regno(lcddev.setxcmd + 1);
        lcd_wr_data(x & 0xFF);
        lcd_wr_regno(lcddev.setycmd);
        lcd_wr_data(y >> 8);
        lcd_wr_regno(lcddev.setycmd + 1);
        lcd_wr_data(y & 0xFF);
    }
    else    /* 9341/5310/7789/7796/9806 等 设置坐标 */
    {
        lcd_wr_regno(lcddev.setxcmd);
        lcd_wr_data(x >> 8);
        lcd_wr_data(x & 0xFF);
        lcd_wr_regno(lcddev.setycmd);
        lcd_wr_data(y >> 8);
        lcd_wr_data(y & 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。

接下来介绍画点函数，其定义如下：

```
/**
 * @brief       画点
 * @param       x,y: 坐标
```

```
 * @param       color: 点的颜色(32 位颜色,方便兼容 LTDC)
 * @retval      无
 */
void lcd_draw_point(uint16_t x, uint16_t y, uint32_t color)
{
    lcd_set_cursor(x, y);        /* 设置光标位置 */
    lcd_write_ram_prepare();     /* 开始写入 GRAM */
    LCD->LCD_RAM = 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。lcd_draw_point 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。

下面介绍读点函数,用于读取 LCD 的 GRAM,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_read_point,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_set_cursor 函数来实现。lcd_read_point 的代码如下:

```
/**
 * @brief       读取个某点的颜色值
 * @param       x,y:坐标
 * @retval      此点的颜色(32 位颜色,方便兼容 LTDC)
 */
uint32_t lcd_read_point(uint16_t x, uint16_t y)
{
    uint16_t 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_set_cursor(x, y);       /* 设置坐标 */

    if (lcddev.id == 0x5510)
    {
        lcd_wr_regno(0x2E00);   /* 5510 发送读 GRAM 指令 */
    }
    else
    {
        /* 9341/5310/1963/7789/7796/9806 等发送读 GRAM 指令 */
        lcd_wr_regno(0x2E);
    }

    r = lcd_rd_data();          /* 假读(dummy read) */

    if (lcddev.id == 0x1963)
    {
        return r;               /* 1963 直接读就可以 */
    }

    r = lcd_rd_data();          /* 实际坐标颜色 */

    if (lcddev.id == 0x7796)    /* 7796 一次读取一个像素值 */
    {
        return r;
    }

    /* 9341/5310/5510/7789/9806 要分 2 次读出 */
    b = lcd_rd_data();
    /* 对于 9341/5310/5510/7789/9806,第一次读取的是 RG 的值,R 在前,G 在后,各占 8 位 */
    g = r & 0xFF;
    g <<= 8;
```

```
    /* ILI9341/NT35310/NT35510/ST7789/ILI9806 需要公式转换一下 */
    return (((r >> 11) << 11) | ((g >> 10) << 5) | (b >> 11));
}
```

在 lcd_read_point 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_show_char，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：

```
/**
 * @brief       在指定位置显示一个字符
 * @param       x,y  : 坐标
 * @param       chr  : 要显示的字符:" "--->"~"
 * @param       size : 字体大小 12/16/24
 * @param       mode : 叠加方式(1); 非叠加方式(0);
 * @retval      无
 */
void lcd_show_char(uint16_t x, uint16_t y, char chr, uint8_t size,
                   uint8_t mode, uint16_t color)
{
    uint8_t temp, t1, t;
    uint16_t y0 = y;
    uint8_t csize = 0;
    uint8_t *pfont = 0;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size / 2);
    /* 得到偏移后的值（ASCII 字库是从空格开始取模，所以-' '就是对应字符的字库） */
    chr = chr - ' ';

    switch (size)
    {
        case 12:
            pfont = (uint8_t *)asc2_1206[chr];  /* 调用 1206 字体 */
            break;
        case 16:
            pfont = (uint8_t *)asc2_1608[chr];  /* 调用 1608 字体 */
            break;
        case 24:
            pfont = (uint8_t *)asc2_2412[chr];  /* 调用 2412 字体 */
            break;
        case 32:
            pfont = (uint8_t *)asc2_3216[chr];  /* 调用 3216 字体 */
            break;
        default:
            return ;
    }

    for (t = 0; t < csize; t++)
    {
        temp = pfont[t];                        /* 获取字符的点阵数据 */
        for (t1 = 0; t1 < 8; t1++)              /* 一个字节 8 个点 */
        {
            if (temp & 0x80)                    /* 有效点,需要显示 */
            {
                lcd_draw_point(x, y, color);    /* 画点出来,要显示这个点 */
            }
            else if (mode == 0)                 /* 无效点,不显示 */
            {
                /* 画背景色,相当于这个点不显示(注意背景色由全局变量控制) */
                lcd_draw_point(x, y, g_back_color);
            }
```

```
            temp <<= 1;                                 /* 移位，以便获取下一个位的状态 */
            y++;
            if (y >= lcddev.height)return;              /* 超区域了 */
            if ((y - y0) == size)                       /* 显示完一列了？ */
            {
                y = y0;                                 /* y 坐标复位 */
                x++;                                    /* x 坐标递增 */
                if (x >= lcddev.width)return;           /* x 坐标超区域了 */
                break;
            }
        }
    }
}
```

在 lcd_show_char 函数里面，我们用到了四个字符集点阵数据数组 asc2_1206、asc2_1608、asc2_2412 和 asc2_3216。

lcd.c 的函数比较多，其他的函数请大家自行查看源码，都有详细的注释。

2. main.c 代码

在 main.c 文件代码如下：

```
int main(void)
{
  uint8_t x = 0;
  uint8_t lcd_id[12];

  HAL_Init();                                   /* 初始化 HAL 库 */
  sys_stm32_clock_init(336, 8, 2, 7);           /* 设置时钟, 168Mhz */
  delay_init(168);                              /* 延时初始化 */
  usart_init(115200);                           /* 串口初始化为 115200 */
  led_init();                                   /* 初始化 LED */
  lcd_init();                                   /* 初始化 LCD */
  g_point_color = RED;
  sprintf((char *)lcd_id, "LCD ID:%04X", lcddev.id); /* 将 id 打印到 lcd_id 数组 */

  while (1)
  {
    switch (x)
    {
     case 0: lcd_clear(WHITE); break;
     case 1: lcd_clear(BLACK);break;
     case 2: lcd_clear(BLUE);break;
     case 3: lcd_clear(RED);break;
     case 4: lcd_clear(MAGENTA);break;
     case 5: lcd_clear(GREEN);break;
     case 6: lcd_clear(CYAN); break;
     case 7: lcd_clear(YELLOW); break;
     case 8: lcd_clear(BRRED); break;
     case 9: lcd_clear(GRAY); break;
     case 10: lcd_clear(LGRAY); break;
     case 11: lcd_clear(BROWN); break;
    }
     lcd_show_string(10, 40, 240, 32, 32, "STM32", RED);
     lcd_show_string(10, 80, 240, 24, 24, "TFTLCD TEST", RED);
     lcd_show_string(10, 110, 240, 16, 16, "ATOM@ALIENTEK", RED);
     lcd_show_string(10, 130, 240, 16, 16,(char *)lcd_id, RED); /*显示 LCD_ID*/
     x++;
     if (x == 12)
     {
         x = 0;
     }
     LED0_TOGGLE();   /* LED0 闪烁 */
     delay_ms(1000);
  }
```

```
}
```

main 函数功能主要是显示一些固定的字符，字体大小包括 32*16、24*12、16*8 和 12*6 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，最终在死循环中通过 lcd_show_strinig 函数进行屏幕显示，sprintf 的详细用法，请百度学习。

特别注意：

usart_init 函数，不能去掉，因为在 lcd_init 函数里面调用了 printf，所以一旦去掉这个初始化，就会死机！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，停在 usart.c 里面的 fputc 函数出不来。

20.4 下载验证

下载代码后，LED0 不停的闪烁，提示程序已经在运行了。同时可以看到 TFTLCD 模块的显示背景色不停切换，如图 20.4.1 所示：

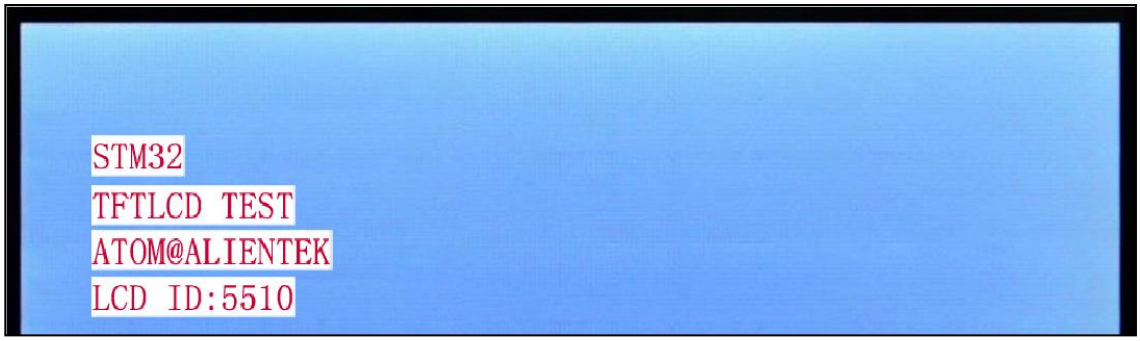


图 20.4.1 TFTLCD 显示效果图

此外，为了让大家能直观的了解 LCD 屏的扫描方式，我们额外编写了两个 main.c 文件（main1.c 和 main2.c，放到 User 文件夹中），方便大家编译下载，观察现象。

使用方法：关闭工程后，先把原实验中的 main.c 改成其他名字，然后把 main1.c 重命名为 main.c，双击 keilkill.bat 清理编译的中间文件，最后打开工程重新编译下载，就可以观察实验现象。观察了 main1.c，可以再观察 main2.c，main2.c 文件的操作方法类似。这两个 main.c 文件的程序非常简单，这里就不讲解，具体请看源码。

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。该组件由正点原子开发提供，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。

本章分为如下几个小节：

21.1 USMART 调试组件简介
21.2 硬件设计
21.3 程序设计
21.4 下载验证

21.1 USMART 调试组件简介

USMART 是由正点原子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 V3.5。

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。

接下来我们看下 USMART 的组成，USMART 组件总共包含 8 个文件，如图 21.1.1 所示：

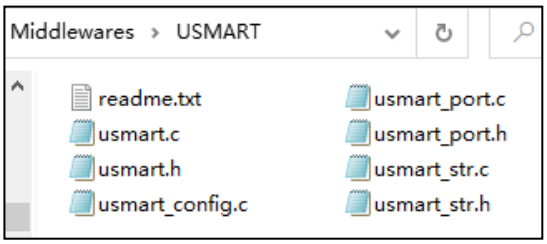


图 21.1.1 USMART 组件代码

USMART 每个文件的作用如表 21.1.1 所示：

USMART 文件	说明
usmart.c	USMART 核心文件，用于处理命令以及对外互交
usmart.h	USMART 核心文件头文件，定义结构体，宏定义、函数申明等
usmart_str.c	USMART 字符串处理文件，用于字符串转换、参数获取等
usmart_str.h	USMART 字符串处理头文件，用于函数申明
usmart_port.c	USMART 移植文件，用于 USMART 移植
usmart_port.h	USMART 移植头文件，定义用户配置参数、宏定义、函数申明等
usmart_config.c	USMART 函数管理文件，用于添加用户需要 USMART 管理的函数
readme.txt	USMART 介绍文件，用于说明 USMART 版本和功能

表 21.1.1 USMART 文件介绍

经过以上简单介绍，我们对 USMART 有了大概了解，接下来我们将在下一小节给大家介绍 USMART 组件的移植和使用。

21.2 硬件设计

1. 例程功能

本实验通过 usmart 调用单片机里面的函数，实现对 LCD 显示和 LED 以及延时的控制。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
LED1 - PE1
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）定时器 4
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

21.3 程序设计

21.3.1 程序流程图

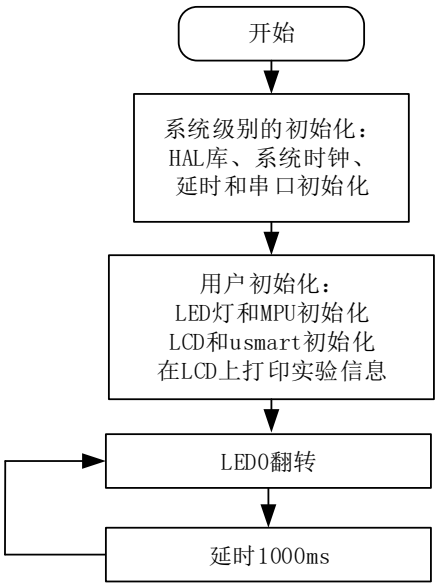


图 21.3.1.1 USMART 调试组件实验程序流程图

21.3.2 程序解析

1. USMART 驱动代码

要使用 USMART，我们先得进行代码移植，USMART 的移植非常简单，我们只需要修改 usmart_port.c 里面的 5 个函数即可完成移植。

第一个函数，USMART 输入数据流获取函数，该函数的实现代码如下：

```
/**
 * @brief       获取输入数据流(字符串)
 *   @note      USMART 通过解析该函数返回的字符串以获取函数名及参数等信息
 * @param       无
 * @retval
 *   @arg       0,  没有接收到数据
 *   @arg       其他,数据流首地址(不能是 0)
 */
char *usmart_get_input_string(void)
{
    uint8_t len;
    char *pbuf = 0;

    if (g_usart_rx_sta & 0x8000)        /* 串口接收完成? */
    {
        len = g_usart_rx_sta & 0x3fff;  /* 得到此次接收到的数据长度 */
        g_usart_rx_buf[len] = '\0';     /* 在末尾加入结束符. */
        pbuf = (char*)g_usart_rx_buf;
        g_usart_rx_sta = 0;             /* 开启下一次接收 */
    }

    return pbuf;
}
```

该函数通过 SYSTEM 文件夹默认的串口接收来实现输入数据流获取。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（详细接收原理请参考：串口通信实验）。

第二个是 usmart_timx_init 函数，该函数的实现代码如下：

```
/**
 * @brief       定时器初始化函数
 * @param       arr:自动重装载值
 *              psc:定时器分频系数
 * @retval      无
 */
void usmart_timx_init(uint16_t arr, uint16_t psc)
{
    USMART_TIMX_CLK_ENABLE();

    g_timx_usmart_handle.Instance = USMART_TIMX;          /* 通用定时器 4 */
    g_timx_usmart_handle.Init.Prescaler = psc;            /* 分频系数 */
    g_timx_usmart_handle.Init.CounterMode = TIM_COUNTERMODE_UP;  /* 向上计数器 */
    g_timx_usmart_handle.Init.Period = arr;               /* 自动装载值 */
    HAL_TIM_Base_Init(&g_timx_usmart_handle);
    HAL_TIM_Base_Start_IT(&g_timx_usmart_handle);         /* 使能定时器和定时器中断 */
    HAL_NVIC_SetPriority(USMART_TIMX_IRQn, 3, 3);         /* 抢占优先级 3，子优先级 3 */
    HAL_NVIC_EnableIRQ(USMART_TIMX_IRQn);                 /* 开启 TIM 中断 */
}
```

该函数只有在：USMART_ENTIMX_SCAN 值为 1 时，才需要实现，用于定时器初始化，利用定时器完成对 usmart_scan 函数的周期性调用，并实现函数运行时间计时（runtime）功能。该函数里面的：USMART_TIMX_CLK_ENABLE()、USMART_TIMX 和 USMART_TIMX_IRQn 等定时器相关宏定义我们在 usmart_port.h 里面定义，方便大家修改。本实验我们用的 TIM4。

注意：usmart_timx_init 函数是在 usmart.c 里面，被 usmart_init 调用，usmart_init 源码如下：

```
/**
 * @brief       初始化 USMART
 * @param       tclk: 定时器的工作频率(单位:Mhz)
 * @retval      无
 */
void usmart_init(uint16_t tclk)
{
#if USMART_ENTIMX_SCAN == 1
    usmart_timx_init(1000, tclk * 100 - 1);
#endif
    usmart_dev.sptype = 1;  /* 十六进制显示参数 */
}
```

该函数有一个参数 tclk，就是用于定时器初始化。注意：这里的 tclk 是指所选 TIM 的时钟频率，而非系统主频！这里我们用的 TIM4，对于 STM32F407 来说，TIM4 的时钟源来自 2 倍的 APB1，频率为：84MHz。

另外，USMART_ENTIMX_SCAN 是在 usmart_port.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_timx_reset_time 和 usmart_timx_get_time，这两个函数代码如下：

```
/**
 * @brief       复位 runtime
 *   @note      需要根据所移植到的 MCU 的定时器参数进行修改
 * @param       无
 * @retval      无
 */
void usmart_timx_reset_time(void)
{
    __HAL_TIM_CLEAR_FLAG(&g_timx_usmart_handle, TIM_FLAG_UPDATE);/* 清中断标志 */
    __HAL_TIM_SET_AUTORELOAD(&g_timx_usmart_handle, 0xFFFF);    /* 重载值设置最大 */
    __HAL_TIM_SET_COUNTER(&g_timx_usmart_handle, 0);            /* 清定时器 CNT */
    usmart_dev.runtime = 0;
}

/**
 * @brief       获得 runtime 时间
 *   @note      需要根据所移植到的 MCU 的定时器参数进行修改
 * @param       无
 * @retval      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
 */
uint32_t usmart_timx_get_time(void)
{
    /* 在运行期间,产生了定时器溢出 */
    if (__HAL_TIM_GET_FLAG(&g_timx_usmart_handle, TIM_FLAG_UPDATE) == SET)
    {
        usmart_dev.runtime += 0xFFFF;
    }
    usmart_dev.runtime += __HAL_TIM_GET_COUNTER(&g_timx_usmart_handle);
    return usmart_dev.runtime;      /* 返回计数值 */
}
```

usmart_timx_reset_time 函数在每次 USMART 调用函数之前执行，清除定时器的计数器，然后在函数执行完之后，调用 usmart_timx_get_time 获取计数器值，从而得到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 STM32F407 的 TIM4 来说，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_TIMX_IRQHandler 函数，该函数的实现代码如下：

```
/**
 * @brief       USMART 定时器中断服务函数
 * @param       无
 * @retval      无
 */
void USMART_TIMX_IRQHandler(void)
{
    /* 溢出中断 */
    if(__HAL_TIM_GET_IT_SOURCE(&g_timx_usmart_handle,TIM_IT_UPDATE)==SET)
    {
        usmart_dev.scan();                                          /* usmart 扫描 */
        __HAL_TIM_SET_COUNTER(&g_timx_usmart_handle, 0);;           /* 清定时器 CNT */
        __HAL_TIM_SET_AUTORELOAD(&g_timx_usmart_handle, 100);       /* 恢复原来的设置 */
    }

    __HAL_TIM_CLEAR_IT(&g_timx_usmart_handle, TIM_IT_UPDATE);       /* 清除中断标志位 */
}
```

该函数是定时器 TIMX 的中断服务函数，也是一个宏定义函数，同样是在 usmart_port.h 里面定义，方便大家修改。该函数主要用于周期性调用 usmart 扫描函数（实际函数：usmart_scan），完成对输入数据流的处理。同时，清除定时器的 CNT 值，并设置自动重装载值。

完成这几个函数的移植，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。

外我们还需要在 usmart_config.c 文件里面添加想要被 USMART 调用的函数。打开 usmart_config.c 文件，如图 21.3.2.1 所示：

```
/*****************************************************************************************************/
/* 用户配置区
 * 这下面要包含所用到的函数所申明的头文件(用户自己添加)
 */

#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/delay/delay.h"
#include "./BSP/LCD/lcd.h"

extern void led_set(uint8_t sta);
extern void test_fun(void(*ledset)(uint8_t), uint8_t sta);

/* 函数名列表初始化(用户自己添加)
 * 用户直接在这里输入要执行的函数名及其查找串
 */
struct _m_usmart_nametab usmart_nametab[] =
{
#if USMART_USE_WRFUNS == 1      /* 如果使能了读写操作 */
    (void *)read_addr, "uint32_t read_addr(uint32_t addr)",
    (void *)write_addr, "void write_addr(uint32_t addr,uint32_t val)",
#endif
    (void *)delay_ms, "void delay_ms(uint16_t nms)",
    (void *)delay_us, "void delay_us(uint32_t nus)",
```

函数所在头文件添加区

用户函数添加区

图 21.3.2.1 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 宏定义，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，usmart_config.c 文件中我们已经添加了 LCD 的相关函数，大家可以查看本实验的 usmart_config.c 文件，并可在串口助手中调用。

具体的调用方法 21.4 小节有具体讲解。

2. main.c 代码

在 main 函数之前，我们添加了 led_set 和 test_fun 两个函数，代码如下：

```
/* LED 状态设置函数 */
void led_set(uint8_t sta)
{
    LED1(sta);
}

/* 函数参数调用测试函数 */
void test_fun(void(*ledset)(uint8_t), uint8_t 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。

main 函数代码如下：

```
int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    usmart_dev.init(84);                    /* USMART 初始化 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USMART TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while(1)
    {
        LED0_TOGGLE();  /* LED0(RED) 闪烁 */
        delay_ms(500);
    }
}
```

此部分代码功能如下：经过一系列初始化，显示使用信息后，就是在无限循环中 LED0 翻转延时，并等待串口数据。

21.4 下载验证

将程序下载到开发板后，可以看到 LED0 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：

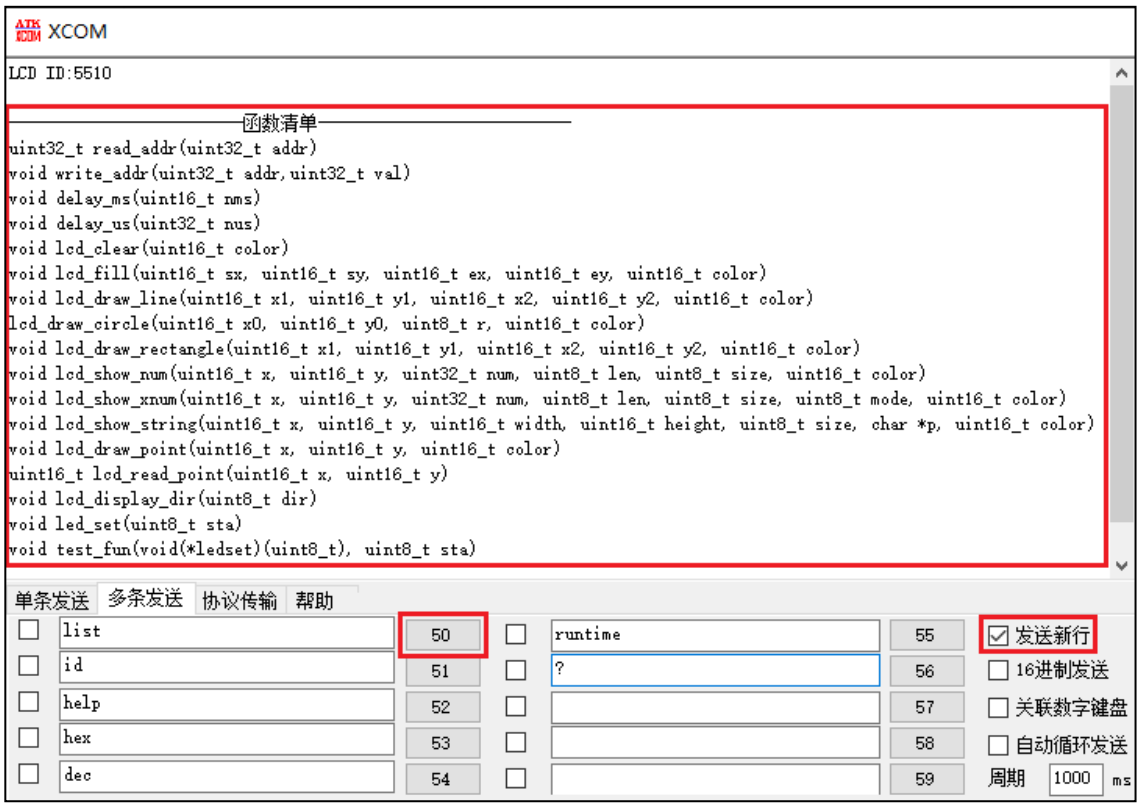


图 21.4.1 驱动串口调试助手

上图中 list、id、help、hex、dec、？和 runtime 都属于 usmart 自带的系统命令，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 21.4.1 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：

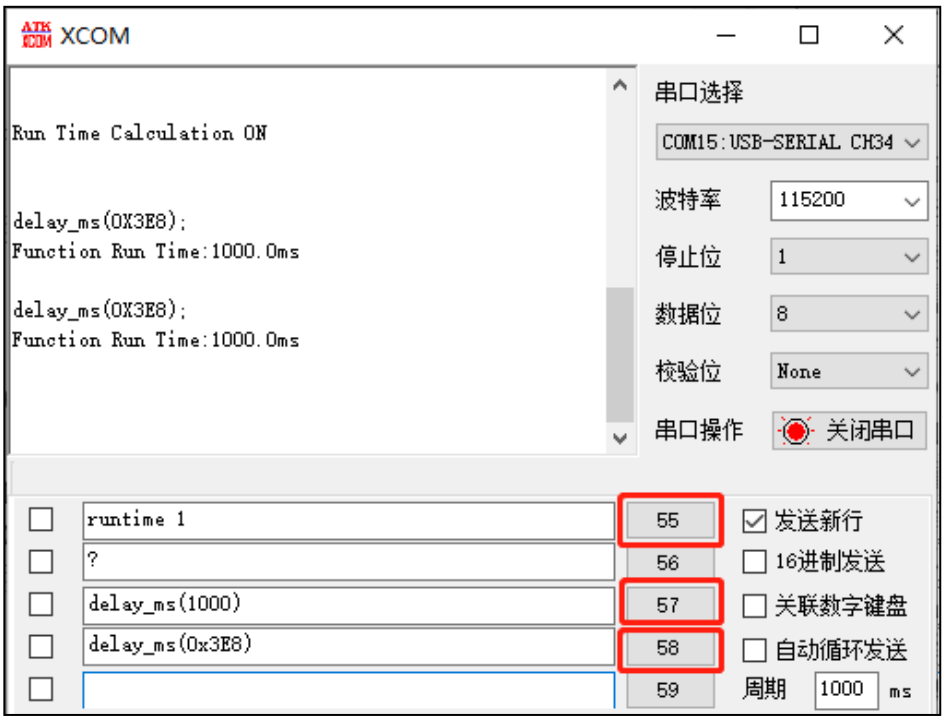


图 21.4.2 串口调用 delay_ms 函数

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_show_string 函数，该函数用于显示字符串，我们通过串口输入：lcd_show_string(30,200,200,16,16," This is a test for usmart!!", 0xF800)，如图 26.4.3 所示：

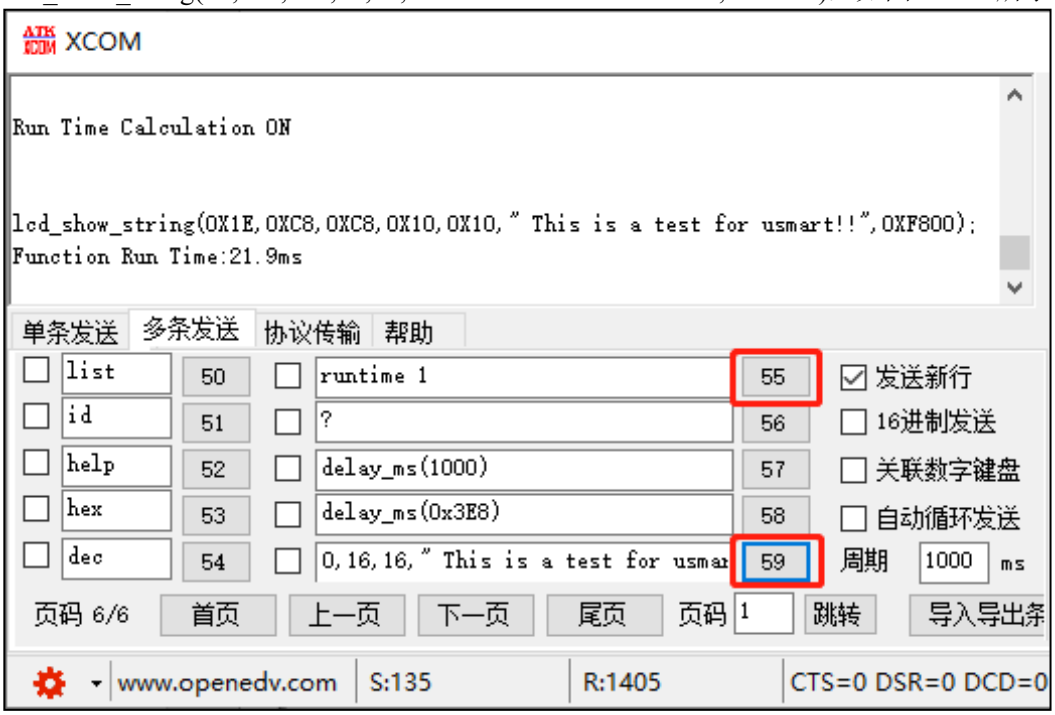


图 21.4.3 串口调用 lcd_show_string 函数

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 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：0x08005599，所以，我们在串口输入：test_fun(0x08005599,0)，就可以控制 LED1 亮了。如图 21.4.4 所示：

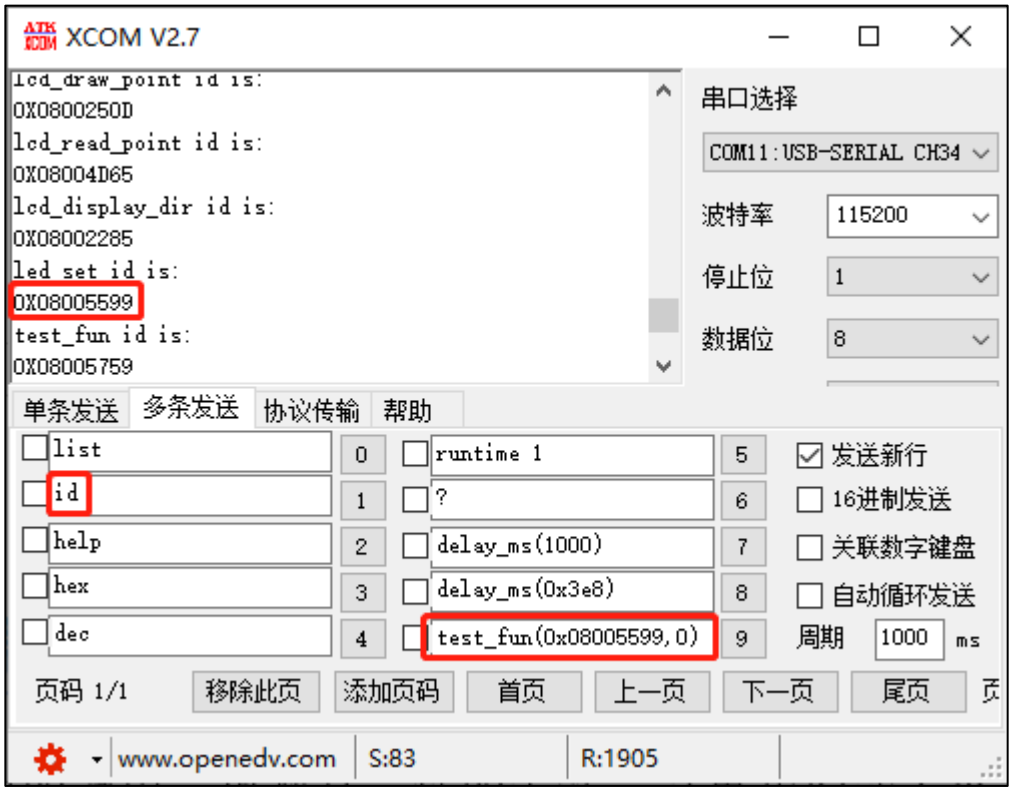


图 21.4.4 串口调用 test_fun 函数

在开发板上，我们可以看到，收到串口发送的 test_fun(0x08005599,0)后，开发板的 LED1 亮了，然后大家可以通过发送 test_fun(0x08005599,1)，来关闭 LED1。说明我们成功的通过 test_fun 函数调用 led_set，实现了对 LED1 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

本章，我们将介绍 STM32F407 的内部实时时钟（RTC）。我们将使用 LCD 模块来显示日期和时间，实现一个简单的实时时钟，并可以设置闹铃，另外还将介绍 BKP 的使用。

本章分为如下几个小节：

22.1 RTC 时钟简介
22.2 硬件设计
22.3 程序设计
22.4 下载验证

22.1 RTC 时钟简介

STM32F407 的实时时钟（RTC）是一个独立的定时器。STM32 的 RTC 模块拥有一组连续计数的计数器，在相对应的软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统的当前时间和日期。

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。

22.1.1 RTC 框图

下面先来学习 RTC 框图，通过学习 RTC 框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。RTC 的框图，如图 22.1.1 所示：

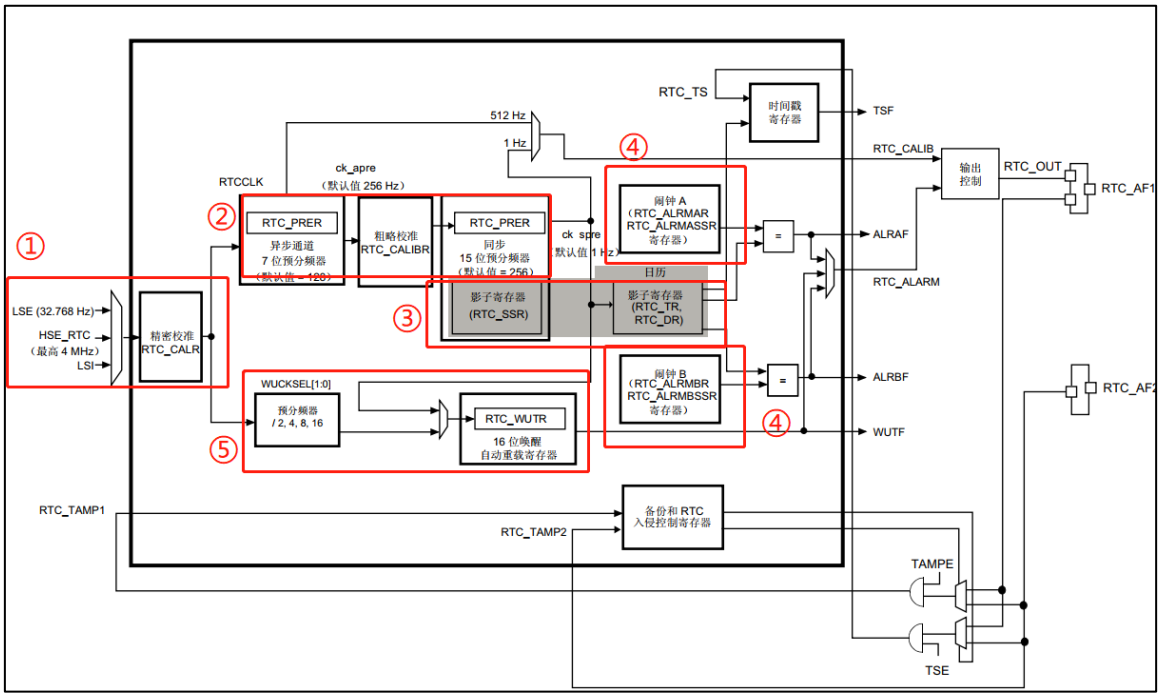


图 22.1.1 RTC 框图

我们把 RTC 框图分成以下几个部分讲解：

① 时钟源

STM32 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 时钟三者中选择其一（通过设置 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)。外部晶振具有精度高的优点。LSI 是 STM32 芯片内部

的低速 RC 振荡器，频率约 32 KHz，缺点是精度较低，所以一般不建议使用。比如当没有外部低速晶振（32.768KHz）的时候，分频后的 HSE 可以作为备选使用的时钟源。

② 预分频器

预分配器（RTC_PRER）分为 2 个部分：一个通过 RTC_PRER 寄存器的 PREDIV_A 位配置的 7 位异步预分频器。另一个通过 RTC_PRER 寄存器的 PREDIV_S 位配置的 15 位同步预分频器。

经过 7 位异步预分频器出来的时钟 ck_apre 可作为 RTC_SSR 亚秒递减计数器（RTC_SSR）的时钟，ck_apre 时钟频率的计算公式如下：

$$Fck\_apre = \frac{Frtcclk}{(PREDIV\_A + 1)}$$

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。

经过 15 位同步预分频器出来的时钟 ck_spre 可以用于更新日历，也可以用作 16 位唤醒自动重载定时器的时基，ck_apre 时钟频率的计算公式如下：

$$Fck\_spre = \frac{Frtcclk}{(PREDIV\_S + 1) * (PREDIV\_A + 1)}$$

PREDIV_A 和 PREDIV_S 分别为 RTC 的异步和同步分频器，使用两个预分频器时，我们推荐设置 7 位异步预分频器（PREDIV_A）的值较大，以最大程度降低功耗。例如：本实验我们的外部低速晶振的频率 32.768KHz 经过 7 位异步预分频器后，再经过 15 位同步预分频器，要得到 1Hz 频率的时钟用于更新日历。通过计算知道，32.768KHz 的时钟要经过 32768 分频，才能得到 1Hz 的 ck_spre。于是我们只需要设置：PREDIV_A=0X7F，即 128 分频；PREDIV_S=0XFF，即 256 分频，即可得到 1Hz 的 Fck_spre，PREDIV_A 的值我们也是往尽量大的原则，以最大程度降低功耗。

③ 时间和日期相关寄存器

该部分包括三个影子寄存器，RTC_SSR（亚秒）、RTC_TR（时间）、RTC_DR（日期）。实时时钟一般表示为：时/分/秒/亚秒。RTC_TR 寄存器用于存储时/分/秒时间数据，可读可写（即可设置或者获取时间）。RTC_DR 寄存器用于存储日期数据，包括年/月/日/星期，可读可写（即可设置或者获取日期）。RTC_SSR 寄存器用于存储亚秒级的时间，这样我们可以获取更加精确的时间数据。

这三个影子寄存器可以通过与 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。

④ 可编程闹钟

STM32F407 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。

⑤ 周期性自动唤醒

STM32F407 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）自动重载，之后必须用软件清零 WUTF 标志。

通过将 RTC_CR 寄存器中的 WUTIE 位置 1 来使能周期性唤醒中断时，可以使 STM32 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 STM32。

22.1.2 RTC 寄存器

接下来，我们介绍本实验我们要用到的 RTC 寄存器。

- **RTC 时间寄存器（RTC_TR）**

RTC 时间寄存器描述如图 22.1.2.1 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 22.1.2.1 RTC_TR 寄存器

该寄存器是 RTC 的时间寄存器，可读可写，对该寄存器写，可以设置时间，对该寄存器读，可以获取当前的时间，此外该寄存器受到寄存器写保护，通过 RTC 写保护寄存器(RTC_WPR)设置，后面会讲解到 RTC_WPR 寄存器。需要注意的是：本寄存器存储的数据都是 BCD 格式的，读取之后需要进行转换，方可得到十进制的时分秒等数据。

- **RTC 日期寄存器（RTC_DR）**

RTC 日期寄存器描述如图 22.1.2.2 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 22.1.2.2 RTC_DR 寄存器

该寄存器是 RTC 的日期寄存器，可读可写，对该寄存器写，可以设置日期，对该寄存器读，可以获取当前的日期，同样该寄存器也受到寄存器写保护，存储的数据也都是 BCD 格式的。

- **RTC 控制寄存器（RTC_CR）**

RTC 控制寄存器描述如图 22.1.2.3 所示：

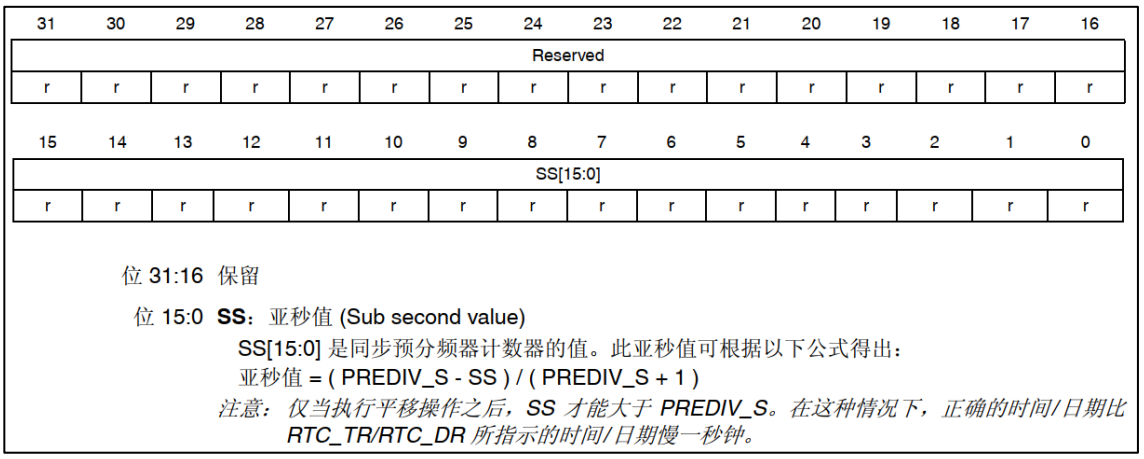
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								COE	OSEL[1:0]		POL	COSEL	BKP	SUB1H	ADD1H
								rw	rw	rw	rw	rw	rw	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSIE	WUTIE	ALRBE	ALRAIE	TSE	WUTE	ALRBE	ALRAE	DCE	FMT	BYPS HAD	REFCKON	TSEDGE	WUCKSEL[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 22.1.2.3 RTC_CR 寄存器

该寄存器重点介绍几个要用到的位：WUTIE 是唤醒定时器中断使能位，ALRAIE 是闹钟 A 中断使能位，本章用到这两个使能位，都设置为 1 即可。WUTE 和 ALRAE 分别是唤醒定时器和闹钟 A 使能位，同样都设置为 1，开启。FMT 为小时格式选择位，我们设置为 0，选择 24 小时制。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 亚秒寄存器（RTC_SSR）**

RTC 亚秒寄存器描述如图 22.1.2.4 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 31:16 保留

位 15:0 **SS**：亚秒值 (Sub second value)

SS[15:0] 是同步预分频器计数器的值。此亚秒值可根据以下公式得出：

亚秒值 = (PREDIV_S - SS) / (PREDIV_S + 1)

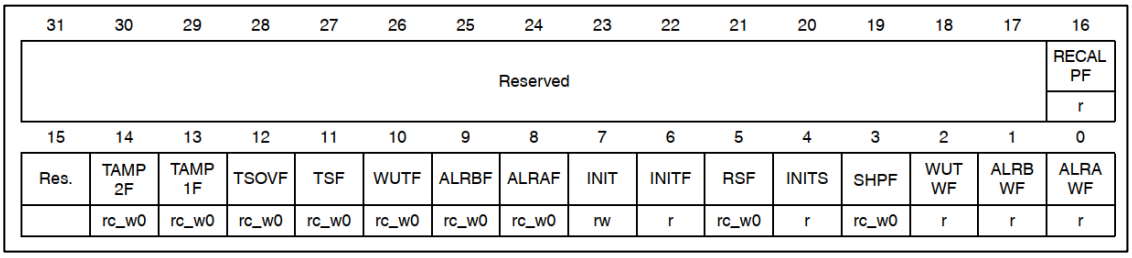
注意：仅当执行平移操作之后，SS 才能大于 PREDIV_S。在这种情况下，正确的时间/日期比 RTC_TR/RTC_DR 所指示的时间/日期慢一秒钟。

图 22.1.2.4 RTC_SSR 寄存器

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。

- **RTC 初始化和状态寄存器（RTC_ISR）**

RTC 初始化和状态寄存器描述如图 22.1.2.5 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.	TAMP2F	TAMP1F	TSOVF	TSF	WUTF	ALRBF	ALRAF	INIT	INITF	RSF	INITS	SHPF	WUTWF	ALRBWF	ALRAWF
	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 22.1.2.5 RTC_ISR 寄存器

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。

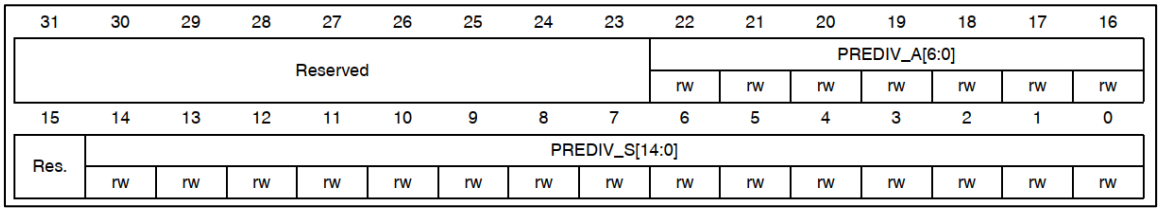
INITF 为初始化标志位，当设置 INIT 为 1 以后，要等待 INITF 为 1，才可以更新时间、日期和预分频寄存器等。

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。

- **RTC 预分频寄存器（RTC_PRER）**

RTC 预分频寄存器描述如图 22.1.2.6 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

图 22.1.2.6 RTC_PRER 寄存器

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。

- **RTC 唤醒寄存器（RTC_WUTR）**

RTC 唤醒寄存器描述如图 22.1.2.7 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 22.1.2.7 RTC_WUTR 寄存器

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。

- **RTC 闹钟 A 寄存器（RTC_ALRMAR）**

RTC 闹钟 A 寄存器描述如图 22.1.2.8 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 22.1.2.8 RTC_ALRMAR 寄存器

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，大家参考手册。

- **RTC 写保护寄存器（RTC_WPR）**

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。

- **RTC 备份寄存器（RTC_BKPxR）**

RTC 备份寄存器描述如图 22.1.2.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BKP[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
BKP[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	w	rw	rw

位 31:0 **BKP[31:0]**

应用可向/从这些寄存器写入/读取数据。

当 V_{DD} 关闭时，这些寄存器由 V_{BAT} 供电，因而系统复位时，这些寄存器不会复位，并且当器件在低功耗模式下工作时，寄存器的内容仍然有效。

发生入侵检测事件时该寄存器会被复位，并且只要 TAMPxF=1，该寄存器就一直保持复位。

图 22.1.2.9 RTC_BKPxR 寄存器

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 RTC 及 RTC 备份寄存器的写访问：

1）电源控制寄存器(PWR_CR)的 DBP 位来使能 RTC 及 RTC 备份寄存器的访问。

2）往 RTC_WPR 写入 0xCA、0x53 解锁序列（先写 0xCA，再写 0x53）。

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。

- **备份区域控制（RCC_BDCR）**

备份区域控制寄存器描述如图 22.1.2.10 所示：

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTCEN	Reserved					RTCSEL[1:0]		Reserved					LSEBYP	LSERDY	LSEON
rw						rw	rw						rw	r	rw

位 31:17 保留，必须保持复位值。

位 16 **BDRST**：备份域软件复位 (Backup domain software reset)
由软件置 1 和清零。
0：复位未激活
1：复位整个备份域
注意：BKPSRAM 不受此复位影响，只能在 Flash 保护级别从级别 1 更改为级别 0 时复位 BKPSRAM。

位 15 **RTCEN**：RTC 时钟使能 (RTC clock enable)
由软件置 1 和清零。
0：RTC 时钟禁止
1：RTC 时钟使能

位 14:10 保留，必须保持复位值。

位 9:8 **RTCSEL[1:0]**：RTC 时钟源选择 (RTC clock source selection)
由软件置 1，用于选择 RTC 的时钟源。选择 RTC 时钟源后，除非备份域复位，否则其不可再更改。可使用 BDRST 位对其进行复位。
00：无时钟
01：LSE 振荡器时钟用作 RTC 时钟
10：LSI 振荡器时钟用作 RTC 时钟
11：由可编程预分频器分频的 HSE 振荡器时钟（通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 RTCPRE[4:0] 位选择）用作 RTC 时钟

位 7:3 保留，必须保持复位值。

位 2 **LSEBYP**：外部低速振荡器旁路 (External low-speed oscillator bypass)
由软件置 1 和清零，用于旁路调试模式下的振荡器。只有在禁止 LSE 时钟后才能写入该位。
0：不旁路 LSE 振荡器
1：旁路 LSE 振荡器

位 1 **LSERDY**：外部低速振荡器就绪 (External low-speed oscillator ready)
由硬件置 1 和清零，用于指示外部 32 kHz 振荡器已稳定。在 LSEON 位被清零后，LSERDY 将在 6 个外部低速振荡器时钟周期后转为低电平。
0：LSE 时钟未就绪
1：LSE 时钟就绪

位 0 **LSEON**：外部低速振荡器使能 (External low-speed oscillator enable)
由软件置 1 和清零。
0：LSE 时钟关闭
1：LSE 时钟开启

图 22.1.2.10 RCC_BDCR 寄存器

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。

22.2 硬件设计

1. 例程功能

本实验通过 LCD 显示 RTC 时间，并可以通过 usmart 设置 RTC 时间，从而调节时间，或设置 RTC 闹钟，还可以写入或者读取 RTC 后备区域 SRAM。LED1 每两秒闪烁一次，表示进入 WAKE UP 中断。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯

LED0 - PF0
LED1 - PF1

2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）RTC(实时时钟)
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

RTC 属于 STM32F407 内部资源，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。

22.3 程序设计

22.3.1 RTC 的 HAL 库驱动

RTC 在 HAL 库中的驱动代码在 stm3f4xx_hal_rtc.c 文件（及其头文件）中。下面介绍几个重要的 RTC 函数，其他没有介绍的请看源码。

1. HAL_RTC_Init 函数

RTC 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc);
```

- **函数描述：**
  用于初始化 RTC。
- **函数形参：**
  形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  RTC_TypeDef                 *Instance;    /* 寄存器基地址 */
  RTC_InitTypeDef             Init;         /* RTC 配置结构体 */
  HAL_LockTypeDef             Lock;         /* RTC 锁定对象 */
  __IO HAL_RTCStateTypeDef    State;        /* RTC 设备访问状态 */
}RTC_HandleTypeDef;
```

1）Instance：指向 RTC 寄存器基地址。
2）Init：是真正的 RTC 初始化结构体，其结构体类型 RTC_InitTypeDef 定义如下：

```
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;
```

HourFormat 用来设置小时格式，可以是 12 小时制或者 24 小时制，这两个选项的宏定义分别为 RTC_HOURFORMAT_12 和 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。

3）Lock：用于配置锁状态。

4）State：RTC 设备访问状态。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

2. HAL_RTC_SetTime 函数

HAL_RTC_SetTime 是设置 RTC 的时间函数。其声明如下：

```
HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc,
                                  RTC_TimeTypeDef *sTime, uint32_t Format);
```

- **函数描述：**

  该函数用于设置 RTC 的时间，即设置时间寄存器 RTC_TR 的相关位的值。

- **函数形参：**

  形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，即 RTC 的句柄。

  形参 2 是 RTC_TimeTypeDef 结构体类型指针变量，定义如下：

```
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。

形参 3 是 uint32_t 类型变量，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 或者 RTC_FORMAT_BCD。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

3. HAL_RTC_SetDate 函数

HAL_RTC_SetDate 是设置 RTC 的日期函数。其声明如下：

```
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc,
                                  RTC_DateTypeDef *sDate, uint32_t Format);
```

- **函数描述：**

  该函数用于设置 RTC 的日期，即设置日期寄存器 RTC_DR 的相关位的值。

- **函数形参：**

  形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，即 RTC 的句柄。

  形参 2 是 RTC_DateTypeDef 结构体类型指针变量，定义如下：

```
typedef struct
{
  uint8_t WeekDay;  /* 星期 */
  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，这四个成员变量分别对应星期、月份、日期和年份，对应的是 RTC_DR 寄存器。

形参 3 是 uint32_t 类型变量，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 或者 RTC_FORMAT_BCD。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

4. HAL_RTC_GetTime 函数

HAL_RTC_GetTime 是获取当前 RTC 时间函数。其声明如下：

```
HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc,
                                  RTC_TimeTypeDef *sTime, uint32_t Format);
```

- **函数描述：**

  该函数用于获取当前 RTC 时间，即读时间寄存器 RTC_TR 的相关位的值。

- **函数形参：**

  形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，即 RTC 的句柄。

  形参 2 是 RTC_TimeTypeDef 结构体类型指针变量，对应的是 RTC_TR 寄存器。

  形参 3 是 uint32_t 类型变量，用来设置获取的时间格式为 BIN 格式还是 BCD 格式，可选值为 RTC_FORMAT_BIN 或者 RTC_FORMAT_BCD。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

5. HAL_RTC_SetDate 函数

HAL_RTC_SetDate 是获取当前 RTC 日期函数。其声明如下：

```
HAL_StatusTypeDef HAL_RTC_SetDate (RTC_HandleTypeDef *hrtc,
                                   RTC_DateTypeDef *sDate, uint32_t Format);
```

- **函数描述：**

  该函数用于获取当前 RTC 日期，即读时间寄存器 RTC_DR 的相关位的值。

- **函数形参：**

  形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，即 RTC 的句柄。

  形参 2 是 RTC_DateTypeDef 结构体类型指针变量，对应的是 RTC_DR 寄存器。

  形参 3 是 uint32_t 类型变量，用来设置获取的时间格式为 BIN 格式还是 BCD 格式，可选值为 RTC_FORMAT_BIN 或者 RTC_FORMAT_BCD。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

6. HAL_RTC_SetAlarm_IT 函数

HAL_RTC_SetAlarm_IT 是设置闹钟并开启闹钟中断的函数。其声明如下：

```
HAL_StatusTypeDef HAL_RTC_SetAlarm_IT(RTC_HandleTypeDef *hrtc,
                                      RTC_AlarmTypeDef *sAlarm, uint32_t Format);
```

- **函数描述：**

  该函数用于设置闹钟并开启闹钟中断。

- **函数形参：**

  形参 1 是 RTC_HandleTypeDef 结构体类型指针变量，即 RTC 的句柄。

  形参 2 是 RTC_AlarmTypeDef 结构体类型指针变量。

  形参 3 是 uint32_t 类型变量，用来设置获取的时间格式为 BIN 格式还是 BCD 格式，可选值为 RTC_FORMAT_BIN 或者 RTC_FORMAT_BCD。

  重点介绍 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;
```

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。

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到星期天，也就是 RTC_Weekday_Monday 到 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。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

RTC 配置步骤

1）使能电源时钟，并使能 RTC 及 RTC 后备寄存器写访问

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_RCC_BKP_CLK_ENABLE();  /* 使能备份时钟 */
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

调用 HAL_RCC_OscConfig 函数配置开启 LSE。

调用 HAL_RCCEx_PeriphCLKConfig 函数选择 RTC 时钟源。

使能 RTC 函数为：__HAL_RCC_RTC_ENABLE。

3）初始化 RTC，设置 RTC 的分频，以及配置 RTC 参数

在 HAL 中，通过 HAL_RTC_Init 函数配置 RTC 分频系数，以及 RTC 的工作参数。

注意：该函数会调用：HAL_RTC_MspInit 函数来完成对 RTC 的底层初始化，包括：RTC 时钟使能、时钟源选择等。

4）设置 RTC 的时间

调用 HAL_RTC_SetTime 函数设置 RTC 时间，该函数实际设置时间寄存器 RTC_TR 的相关位的值。

5）设置 RTC 的日期

调用 HAL_RTC_SetDate 函数设置 RTC 的日期，该函数实际设置日期寄存器 RTC_DR 的相关位的值。

6）获取 RTC 当前日期和时间

调用 HAL_RTC_GetTime 函数获取当前 RTC 时间，该函数实际读取 RTC_TR 寄存器，然后将值存放到相应的结构体中。

调用 HAL_RTC_GetDate 函数获取当前 RTC 日期，该函数实际读取 RTC_DR 寄存器，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，我们还可以唤醒中断、闹钟等，这些将在后面介绍。

22.3.2 程序流程图

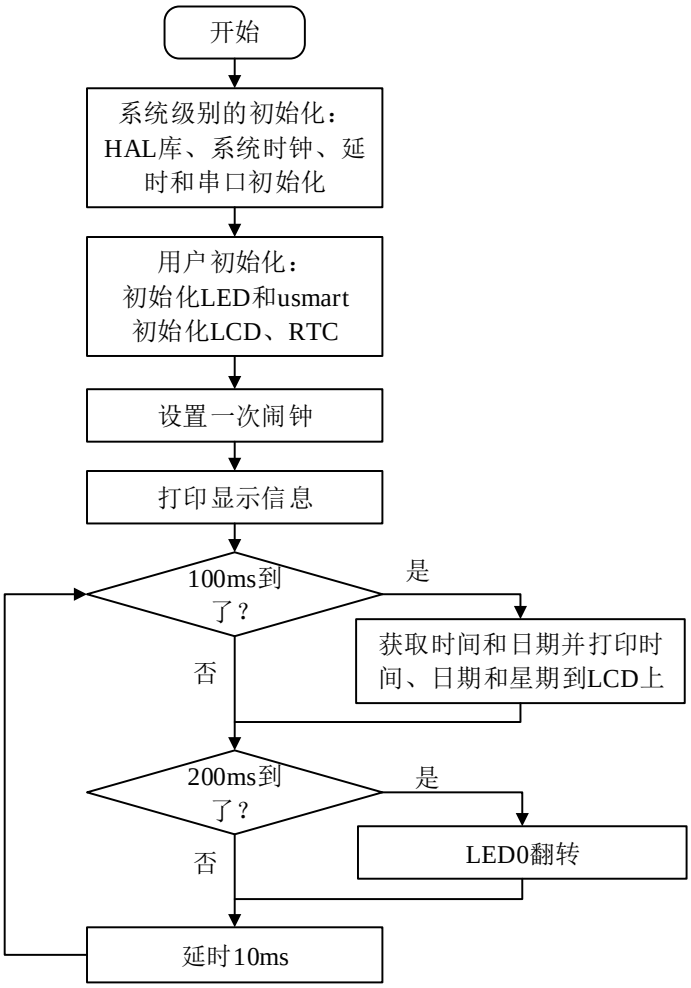


图 22.3.2.1 RTC 实时时钟实验程序流程图

22.3.3 程序解析

1. RTC 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。RTC 驱动源码包括两个文件：rtc.c 和 rtc.h。

由于篇幅所限，rtc.c 中的代码，我们不全部贴出了，只针对几个重要的函数进行介绍。

rtc.h 头文件只有函数的声明，下面我们直接介绍 rtc.c 的程序，先看 RTC 的初始化函数，其定义如下：

```
/**
 * @brief       RTC 初始化
 * @note        默认尝试使用 LSE,当 LSE 启动失败后,切换为 LSI.
 *              通过 BKP 寄存器 0 的值,可以判断 RTC 使用的是 LSE/LSI:
 *              当 BKP0==0x5050 时,使用的是 LSE
 *              当 BKP0==0x5051 时,使用的是 LSI
 *              注意:切换 LSI/LSE 将导致时间/日期丢失,切换后需重新设置.
 *
 * @param       无
 * @retval      0,成功
 *              1,进入初始化模式失败
 */
uint8_t rtc_init(void)
```

```
{
    uint16_t bkpflag = 0;

    g_rtc_handle.Instance = RTC;
    g_rtc_handle.Init.HourFormat = RTC_HOURFORMAT_24;  /* RTC 设置为 24 小时格式 */
    g_rtc_handle.Init.AsynchPrediv = 0x7F;  /* RTC 异步分频系数(1~0x7F) */
    g_rtc_handle.Init.SynchPrediv = 0xFF;   /* RTC 同步分频系数(0~0x7FFF) */
    g_rtc_handle.Init.OutPut = RTC_OUTPUT_DISABLE;
    g_rtc_handle.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
    g_rtc_handle.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;

    /* 检查是不是第一次配置时钟 */
    bkpflag = rtc_read_bkr(RTC_BKP_DR0);    /* 读取 BKP0 的值 */

    if (HAL_RTC_Init(&g_rtc_handle) != HAL_OK)
    {
        return 1;
    }

    if ((bkpflag != 0x5050) && (bkpflag != 0x5051)) /* 之前未初始化过，重新配置 */
    {
        rtc_set_time(6, 59, 56, RTC_HOURFORMAT12_AM);/* 设置时间，根据实际时间修改 */
        rtc_set_date(20, 4, 22, 3);    /* 设置日期 */
    }
    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）。

我们通过读取 BKP 寄存器 0 的值来判断是否需要进行时间的设置，对 BKP 寄存器 0 的写操作是在 HAL_RTC_MspInit 回调函数中实现，下面会讲。第一次未对 RTC 进行初始化 BKP 寄存器 0 的值非 0x5050 非 0x5051，当进行 RTC 初始化时，BKP 寄存器 0 的值就是 0x5050 或 0x5051，所以以上代码操作确保时间只会设置一次，复位时不会重新设置时间。电池正常供电时，我们设置的时间不会因复位或者断电而丢失。

读取后备寄存器的函数其实还是调用 HAL 库提供的函数接口，写后备寄存器函数同样也是。这两个函数如下：

```
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。这里我们只是略微介绍一下。

我们用 HAL_RTC_MspInit 函数来编写 RTC 时钟配等代码，其定义如下：

```
/**
 * @brief       RTC 底层驱动，时钟配置
 * @param       hrtc : RTC 句柄
 * @note        此函数会被 HAL_RTC_Init()调用
 * @retval      无
 */
void HAL_RTC_MspInit(RTC_HandleTypeDef* hrtc)
{
    uint16_t retry = 200;

    RCC_OscInitTypeDef rcc_osc_init_handle;
    RCC_PeriphCLKInitTypeDef rcc_periphclk_init_handle;

    __HAL_RCC_RTC_ENABLE();      /* RTC 使能 */
    HAL_PWR_EnableBkUpAccess(); /* 取消备份区域写保护 */
    __HAL_RCC_PWR_CLK_ENABLE(); /* 使能电源时钟 PWR */
    __HAL_RCC_RTC_ENABLE();      /* RTC 使能 */
```

```
    /* 使用寄存器的方式去检测 LSE 是否可以正常工作 */
    RCC->BDCR |= 1 << 0;          /* 尝试开启 LSE */

    while (retry && ((RCC->BDCR & 0x02) == 0))  /* 等待 LSE 准备好 */
    {
        retry--;
        delay_ms(5);
    }

    if (retry == 0)     /* LSE 起振失败 使用 LSI */
    {
        rcc_osc_init_handle.OscillatorType = RCC_OSCILLATORTYPE_LSI;
        /* 选择要配置的振荡器 */
        rcc_osc_init_handle.PLL.PLLState = RCC_PLL_NONE;    /* PLL 不配置 */
        rcc_osc_init_handle.LSIState = RCC_LSI_ON;          /* LSI 状态:开启 */
        HAL_RCC_OscConfig(&rcc_osc_init_handle);/* 配置设置的 rcc_oscinitstruct */

        rcc_periphclk_init_handle.PeriphClockSelection = RCC_PERIPHCLK_RTC;
        /* 选择要配置外设 RTC */
        rcc_periphclk_init_handle.RTCClockSelection = RCC_RTCCLKSOURCE_LSI;
        /* RTC 时钟源选择 LSI */
        HAL_RCCEx_PeriphCLKConfig(&rcc_periphclk_init_handle);
        /* 配置设置的 rcc_periphclkinitstruct */
        rtc_write_bkr(0, 0x5051);
    }
    else
    {
        rcc_osc_init_handle.OscillatorType = RCC_OSCILLATORTYPE_LSE;
        /* 选择要配置的振荡器 */
        rcc_osc_init_handle.PLL.PLLState = RCC_PLL_NONE;    /* PLL 不配置 */
        rcc_osc_init_handle.LSEState = RCC_LSE_ON;          /* LSE 状态:开启 */
        HAL_RCC_OscConfig(&rcc_osc_init_handle); /* 配置设置的 rcc_oscinitstruct */

        rcc_periphclk_init_handle.PeriphClockSelection = RCC_PERIPHCLK_RTC;
        /* 选择要配置外设 RTC */
        rcc_periphclk_init_handle.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;
        /* RTC 时钟源选择 LSE */
        HAL_RCCEx_PeriphCLKConfig(&rcc_periphclk_init_handle);
        /* 配置设置的 rcc_periphclkinitstruct */
        rtc_write_bkr(0, 0x5050);
    }
}
```

接着，我们介绍一下 rtc_set_alarma 函数，其定义如下：

```
/**
 * @breif       设置闹钟时间(按星期闹铃,24 小时制)
 * @param       week         : 星期几(1~7)
 * @param       hour,min,sec : 小时,分钟,秒钟
 * @retval      无
 */
void rtc_set_alarma(uint8_t week, uint8_t hour, uint8_t min, uint8_t sec)
{
    RTC_AlarmTypeDef rtc_alarm_handle;

    rtc_alarm_handle.AlarmTime.Hours = hour;    /* 小时 */
    rtc_alarm_handle.AlarmTime.Minutes = min;   /* 分钟 */
    rtc_alarm_handle.AlarmTime.Seconds = sec;   /* 秒 */
    rtc_alarm_handle.AlarmTime.SubSeconds = 0;
    rtc_alarm_handle.AlarmTime.TimeFormat = RTC_HOURFORMAT12_AM;

    rtc_alarm_handle.AlarmMask = RTC_ALARMMASK_NONE;    /* 精确匹配星期，时分秒 */
    rtc_alarm_handle.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_NONE;
    /* 按星期 */
```

```
    rtc_alarm_handle.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_WEEKDAY;
    rtc_alarm_handle.AlarmDateWeekDay = week;   /* 星期 */
    rtc_alarm_handle.Alarm = RTC_ALARM_A;       /* 闹钟 A */
    HAL_RTC_SetAlarm_IT(&g_rtc_handle, &rtc_alarm_handle, RTC_FORMAT_BIN);

    HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 1, 2); /* 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，这里使用 HAL 库的 HAL_RTC_SetAlarm_IT 函数来初始化闹钟相关寄存器，前面已经介绍过这个函数，请回顾。

调用函数 rtc_set_wakeup 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：

```
/**
 * @breif       周期性唤醒定时器设置
 * @param       wksel
 *   @arg       RTC_WAKEUPCLOCK_RTCCLK_DIV16        ((uint32_t)0x00000000)
 *   @arg       RTC_WAKEUPCLOCK_RTCCLK_DIV8         ((uint32_t)0x00000001)
 *   @arg       RTC_WAKEUPCLOCK_RTCCLK_DIV4         ((uint32_t)0x00000002)
 *   @arg       RTC_WAKEUPCLOCK_RTCCLK_DIV2         ((uint32_t)0x00000003)
 *   @arg       RTC_WAKEUPCLOCK_CK_SPRE_16BITS      ((uint32_t)0x00000004)
 *   @arg       RTC_WAKEUPCLOCK_CK_SPRE_17BITS      ((uint32_t)0x00000006)
 * @note        000,RTC/16;001,RTC/8;010,RTC/4;011,RTC/2;
 * @note        注意:RTC 就是 RTC 的时钟频率,即 RTCCLK!
 * @param       cnt: 自动重装载值.减到 0,产生中断.
 * @retval      无
 */
void rtc_set_wakeup(uint8_t wksel, uint16_t cnt)
{
    /* 清除 RTC WAKE UP 的标志 */
    __HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&g_rtc_handle, RTC_FLAG_WUTF);
    HAL_RTCEx_SetWakeUpTimer_IT(&g_rtc_handle, cnt, wksel);/* 设置重装载值和时钟 */
    HAL_NVIC_SetPriority(RTC_WKUP_IRQn, 2, 2);  /* 抢占优先级 2,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。

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：

```
/**
 * @breif       RTC 闹钟中断服务函数
 * @param       无
 * @retval      无
 */
void RTC_Alarm_IRQHandler(void)
{
    HAL_RTC_AlarmIRQHandler(&g_rtc_handle);
}

/**
 * @breif       RTC 闹钟 A 中断处理回调函数
 * @param       hrtc : RTC 句柄
 * @retval      无
 */
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("ALARM A!\r\n");
}
```

```
/**
 * @breif       RTC WAKE UP 中断服务函数
 * @param       无
 * @retval      无
 */
void RTC_WKUP_IRQHandler(void)
{
    HAL_RTCEx_WakeUpTimerIRQHandler(&g_rtc_handle);
}

/**
 * @breif       RTC WAKE UP 中断处理处理回调函数
 * @param       hrtc : RTC 句柄
 * @retval      无
 */
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，请大家直接看源码。

2. main.c 代码

main 函数代码如下：

```
int main(void)
{
    uint8_t hour, min, sec, ampm;
    uint8_t year, month, date, week;
    uint8_t tbuf[40];
    uint8_t t=0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    usmart_dev.init(84);                        /* 初始化 USMART */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    rtc_init();                                 /* 初始化 RTC */
    /* 配置 WAKE UP 中断,1 秒钟中断一次 */
    rtc_set_wakeup(RTC_WAKEUPCLOCK_CK_SPRE_16BITS, 0);
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "RTC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    while (1)
    {
        t++;
        if ((t % 10) == 0)  /* 每 100ms 更新一次显示数据 */
        {
            rtc_get_time(&hour, &min, &sec, &ampm);
            sprintf((char *)tbuf, "Time:%02d:%02d:%02d", hour, min, sec);
            lcd_show_string(30, 140, 210, 16, 16, (char*)tbuf, RED);
            rtc_get_date(&year, &month, &date, &week);
            sprintf((char *)tbuf, "Date:20%02d-%02d-%02d", year, month, date);
            lcd_show_string(30, 160, 210, 16, 16, (char*)tbuf, RED);
            sprintf((char *)tbuf, "Week:%d", week);
```

```
            lcd_show_string(30, 180, 210, 16, 16, (char*)tbuf, RED);
        }
        if ((t % 20) == 0)
        {
            LED0_TOGGLE();  /* 每 200ms,翻转一次 LED0 */
        }
        delay_ms(10);
    }
}
```

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。然后，在无限循环中每 100ms 读取 RTC 的时间和日期（一次），并显示在 LCD 上面。每 200ms，翻转一次 LED0。

为方便 RTC 相关函数的调用验证，在 usmart_config.c 里面，修改了 usmart_nametab 如下：

```
/* 函数名列表初始化(用户自己添加)
 * 用户直接在这里输入要执行的函数名及其查找串
 */
struct _m_usmart_nametab usmart_nametab[] =
{
#if USMART_USE_WRFUNS == 1      /* 如果使能了读写操作 */
    (void *)read_addr, "uint32_t read_addr(uint32_t addr)",
    (void *)write_addr, "void write_addr(uint32_t addr,uint32_t val)",
#endif
    (void *)delay_ms, "void delay_ms(uint16_t nms)",
    (void *)delay_us, "void delay_us(uint32_t nus)",

    (void *)rtc_read_bkr, "uint32_t rtc_read_bkr(uint32_t bkrx)",
    (void *)rtc_write_bkr, "void rtc_write_bkr(uint32_t bkrx, uint32_t data)",
    (void *)rtc_set_time, "uint8_t rtc_set_time(uint8_t hour, uint8_t min,
                                                uint8_t sec, uint8_t ampm)",
    (void *)rtc_set_date, "uint8_t rtc_set_date(uint8_t year, uint8_t month,
                                                uint8_t date, uint8_t week)",

    (void *)rtc_set_wakeup, "void rtc_set_wakeup(uint8_t wksel, uint16_t cnt)",
    (void *)rtc_get_week, "uint8_t rtc_get_week(uint16_t year, uint8_t month,
                                                uint8_t day)",
    (void *)rtc_set_alarma, "void rtc_set_alarma(uint8_t week, uint8_t hour,
                                                 uint8_t min, uint8_t sec)",
};
```

将 RTC 的一些相关函数加入了 usmart，这样通过串口就可以直接设置 RTC 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。

22.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了，同时 LED1 每两秒闪烁一次，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：

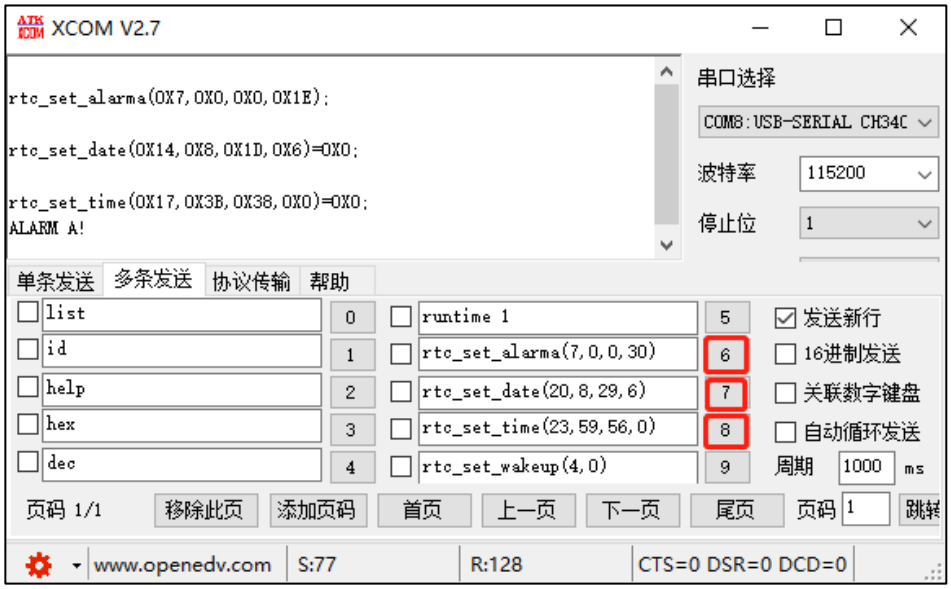


图 22.4.2 通过 USMART 设置时间和日期并测试闹钟 A

按照图中编号 6、7、8 顺序，设置闹钟 A、设置日期、设置时间。然后等待我们设置的时间到来后，串口打印 ALARM A!这个字符串，证明我们的闹钟 A 程序正常运行了！

第二十三章 硬件随机数实验

本章，我们将介绍 STM32F407 的硬件随机数发生器。我们使用 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 LED0 指示程序运行状态。

本章分为如下几个小节：

23.1 随机数发生器简介
23.2 硬件设计
23.3 程序设计
23.4 下载验证

23.1 随机数发生器简介

STM32F407 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。

23.1.1 RNG 框图

下面先来学习 RNG 框图，通过学习 RNG 框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。STM32F407 的随机数发生器框图如图 23.1.1.1 所示：

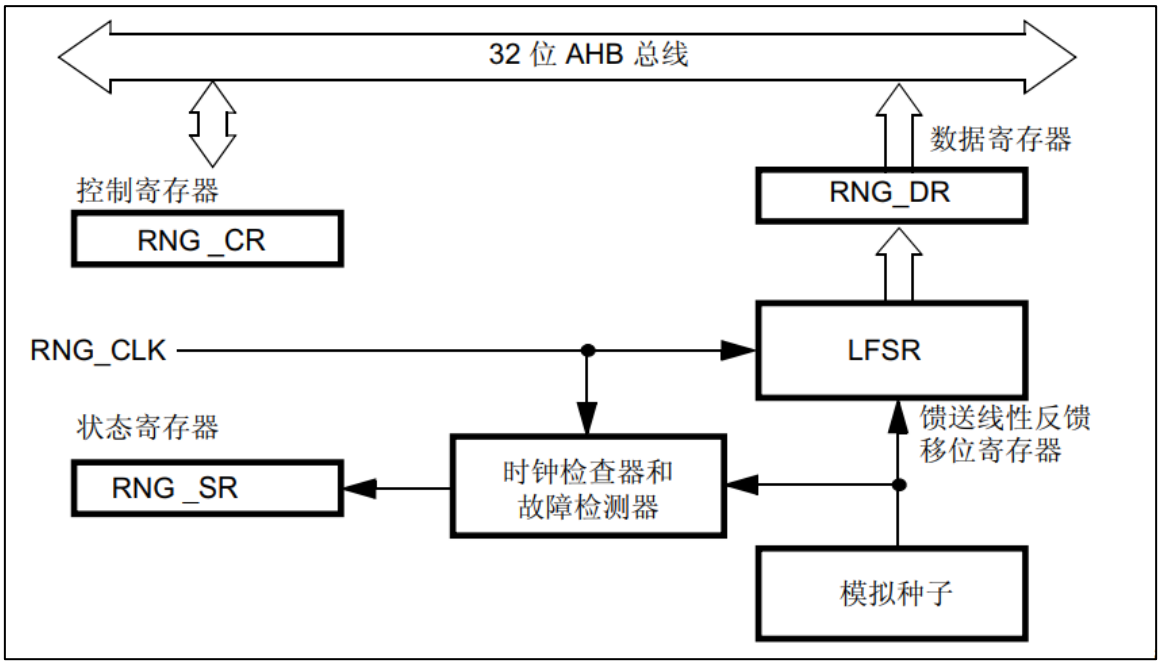


图 23.1.1.1 随机数发生器(RNG)框图

STM32F407 的随机数发生器（RNG）采用模拟电路实现。此电路产生馈入线性反馈移位寄存器（RNG_LFSR）的种子，用于生成 32 位随机数。

该模拟电路由几个环形振荡器组成，振荡器的输出进行异或运算以产生种子。RNG_LFSR 由专用时钟(PLL48CLK)按恒定频率提供时钟信息，因此随机数质量与 HCLK 频率无关。当将大量种子引入 RNG_LFSR 后，RNG_LFSR 的内容会传入数据寄存器(RNG_DR)。

同时，系统会监视模拟种子和专用时钟 rng_clk，当种子上出现异常序列，或 rng_clk 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。

23.1.2 RNG 寄存器

- **RNG 控制寄存器（RNG_CR）**

RNG 控制寄存器描述如图 23.1.2.1 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 23.1.2.1 RNG_CR 寄存器

该寄存器我们只关心 RNGEN 位，该位用于使能随机数发生器，所以必须设置为 1。

- **RNG 状态寄存器（RNG_SR）**

RNG 状态寄存器描述如图 23.1.2.2 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									SEIS	CEIS	Reserved		SECS	CECS	DRDY
									rc_w0	rc_w0			r	r	r

位 0 **DRDY**：数据就绪 (Data ready)
0：RNG_DR 寄存器尚未有效，无可用随机数据
1：RNG_DR 寄存器包含有效随机数据
注意：如果 RNG_CR 寄存器中 IE = 1，则会挂起中断。
读取 RNG_DR 寄存器后，此位恢复到 0，直到计算出新的有效值。

图 23.1.2.2 RNG_SR 寄存器

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。

- **RNG 数据寄存器（RNG_DR）**

RNG 数据寄存器描述如图 23.1.2.3 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 23.1.2.3 RNG_DR 寄存器

RNG_DR 寄存器是只读寄存器，我们可以读取该寄存器获得 32 位随机数值。此寄存器在最多 216 个 AHB 时钟周期后，又可以提供新的随机数值。

23.2 硬件设计

1. 例程功能

本实验使用 STM32F407 自带的硬件随机数生成器（RNG），获取随机数，并显示在 LCD 屏幕上。按 KEY0 可以获取一次随机数。同时程序自动获取 0~9 范围内的随机数，显示在屏幕上。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
　　LED0 – PE0
2）独立按键
　　KEY0 – PE2
3）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
5）RNG（硬件随机数生成器）

3. 原理图

RNG 属于 STM32F407 内部资源，通过软件设置好就可以了。本实验通过配合按键获取随机数和通过 LCD 显示。

23.3 程序设计

23.3.1 RNG 的 HAL 库驱动

RNG 在 HAL 库中的驱动代码在 stm32f4xx_hal_rng.c 文件（及其头文件）中。

1. HAL_RNG_Init 函数

RNG 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_RNG_Init(RNG_HandleTypeDef *hrng);
```

- **函数描述：**
  用于初始化 RNG。
- **函数形参：**
  形参 1 是 RNG_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  RNG_TypeDef                 *Instance;     /* RNG 基地址 */
  HAL_LockTypeDef              Lock;         /* RNG 锁设置 */
  __IO HAL_RNG_StateTypeDef    State;        /* RNG 设备访问状态 */
  __IO  uint32_t               ErrorCode;    /* RNG 错误代码 */
  uint32_t                     RandomNumber; /* RNG 最后生成的随机数 */
} RNG_HandleTypeDef;
```

1）Instance：指向 RNG 寄存器基地址。
2）Lock：用于配置锁状态。
3）State：RNG 设备访问状态。
4）ErrorCode ：RNG 错误代码
5）RandomNumber ：该变量存储 RNG 最后生成的随机数

- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

2. HAL_RNG_GenerateRandomNumber 函数

HAL_RNG_GenerateRandomNumber 是 RNG 生成随机数函数。其声明如下：

```
HAL_StatusTypeDef HAL_RNG_GenerateRandomNumber(RNG_HandleTypeDef *hrng,
                                               uint32_t *random32bit);
```

- **函数描述：**
  该函数用于 RNG 生成随机数。
- **函数形参：**
  形参 1 是 RNG_HandleTypeDef 结构体类型指针变量，即 RNG 的句柄。
  形参 2 是 uint32_t 类型指针变量，随机 32 位指针，生成随机变量。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

RNG 配置步骤

1）使能随机数发生器时钟

调用__HAL_RCC_RNG_CLK_ENABLE 函数使能随机数发生器时钟，实际是通过设置 AHB2ENR 寄存器的相关位。

2）初始化(使能)随机数发生器

通过调用 HAL_RNG_Init 函数初始化随机数发生器，然后自行调用 RNG 的 MSP 回调函数，并使能随机数发生器。

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。回调函数中一般编写与 MCU 相关的外设时钟初始化以及 NVIC 配置。

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，通过调用 HAL_RNG_GenerateRandomNumber 函数判断 DRDY 位并读取随机数值。

通过以上 3 个步骤的设置，我们就可以使用 STM32F407 的随机数发生器（RNG）了。

23.3.2 程序流程图

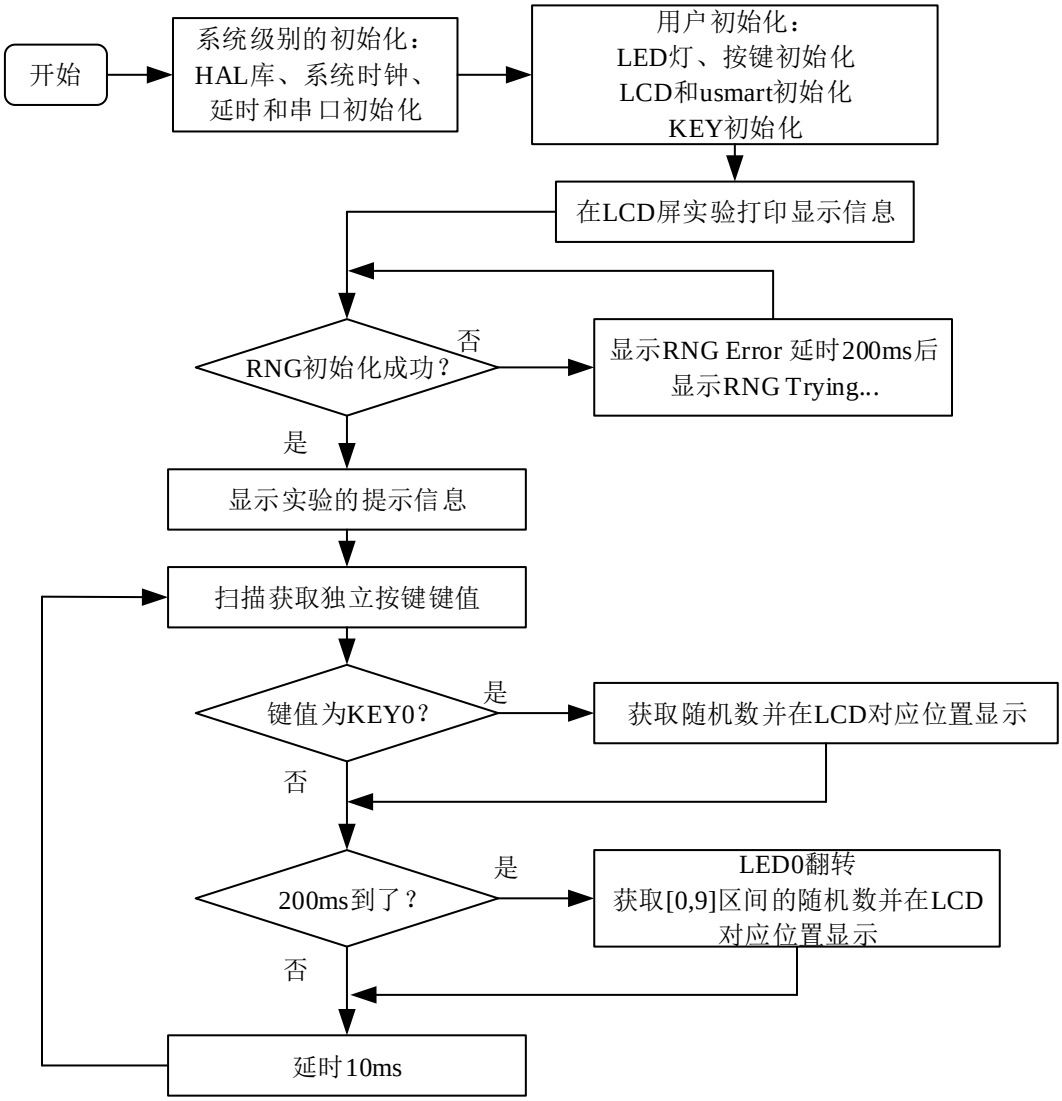


图 23.3.2.1 硬件随机数实验程序流程图

23.3.3 程序解析

1. RNG 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。RNG 驱动源码包括两个文件：rng.c 和 rng.h。

rng.h 头文件只有函数的声明，下面我们直接介绍 rng.c 的程序，先看 RNG 的初始化函数，其定义如下：

```
/**
 * @brief       初始化 RNG
 * @param       无
 * @retval      0,成功;1,失败
 */
uint8_t rng_init(void)
{
    uint16_t retry = 0;

    rng_handle.Instance = RNG;
    HAL_RNG_DeInit(&rng_handle);
```

```
    HAL_RNG_Init(&rng_handle);    /* 初始化 RNG */
    while (__HAL_RNG_GET_FLAG(&rng_handle, 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;
}
```

rng_init 函数中调用 HAL_RNG_Init 函数初始化 RNG，然后等待 RNG 初始化完成，同时判断初始化过程是否超时。

我们用 HAL_RTC_MspInit 回调函数来编写 RNG 时钟配置等代码，其定义如下：

```
/**
 * @brief       RNG 底层驱动，时钟源设置和使能
 * @note        此函数会被 HAL_RNG_Init()调用
 * @param       hrng:RNG 句柄
 * @retval      无
 */
void HAL_RNG_MspInit(RNG_HandleTypeDef *hrng)
{
    __HAL_RCC_RNG_CLK_ENABLE();/* 使能 RNG 时钟 */
}
```

最后两个是我们获取随机数函数，它们的定义如下：

```
/**
 * @brief       得到随机数
 * @param       无
 * @retval      获取到的随机数(32bit)
 */
uint32_t rng_get_random_num(void)
{
    uint32_t randomnum;
    HAL_RNG_GenerateRandomNumber(&rng_handle, &randomnum);
    return randomnum;
}

/**
 * @brief       得到某个范围内的随机数
 * @param       min,max: 最小,最大值.
 * @retval      得到的随机数(rval),满足:min<=rval<=max
 */
int rng_get_random_range(int min, int max)
{
    uint32_t randomnum;
    HAL_RNG_GenerateRandomNumber(&rng_handle, &randomnum);
    return randomnum%(max-min+1) + min;
}
```

其中 rng_get_random_num 用于读取随机数值，而 rng_get_random_range 用于读取一个特定范围内的随机数，实际上它们都是通过调用的函数 HAL_RNG_GenerateRandomNumber 来实现的。

2. main.c 代码

main 函数代码如下：

```
int main(void)
{
    uint32_t random;
    uint8_t t = 0, key;
```

```
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    usmart_dev.init(84);                        /* 初始化 USMART */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "RNG TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (rng_init())  /* 初始化随机数发生器 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "  RNG Error! ", RED);
        delay_ms(200);
        lcd_show_string(30, 130, 200, 16, 16, "RNG Trying...", RED);
    }

    lcd_show_string(30, 110, 200, 16, 16, "RNG Ready!   ", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY0:Get Random Num", RED);
    lcd_show_string(30, 150, 200, 16, 16, "Random Num:", RED);
    lcd_show_string(30, 180, 200, 16, 16, "Random Num[0-9]:", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES)
        {
            random = rng_get_random_num();
            lcd_show_num(30 + 8 * 11, 180, random, 10, 16, BLUE);
        }
        if ((t % 20) == 0)
        {
            LED0_TOGGLE();                          /* 每 200ms,翻转一次 LED0 */
            random = rng_get_random_range(0, 9); /* 取[0,9]区间的随机数 */
            lcd_show_num(30 + 8 * 16, 210, random, 1, 16, BLUE);    /* 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_random_num 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 LED0，周期性闪烁，400ms 闪烁一次。

最后，为了方便测试，我们将 rng_get_random_num 和 rng_get_random_range 加入 USMART，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, "uint32_t read_addr(uint32_t addr)",
    (void *)write_addr, "void write_addr(uint32_t addr,uint32_t val)",
#endif
    (void *)delay_ms, "void delay_ms(uint16_t nms)",
    (void *)delay_us, "void delay_us(uint32_t nus)",

    (void *)rng_get_random_num, "uint32_t rng_get_random_num(void)",
    (void *)rng_get_random_range, "int rng_get_random_range(int min, int max)",
};
```

这样，我们便可以在串口输入我们想要调用的函数（rng_get_random_range 或 rng_get_random_num）进行测试。

23.4 下载验证

将程序下载到开发板后，可以看到 LED0 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：

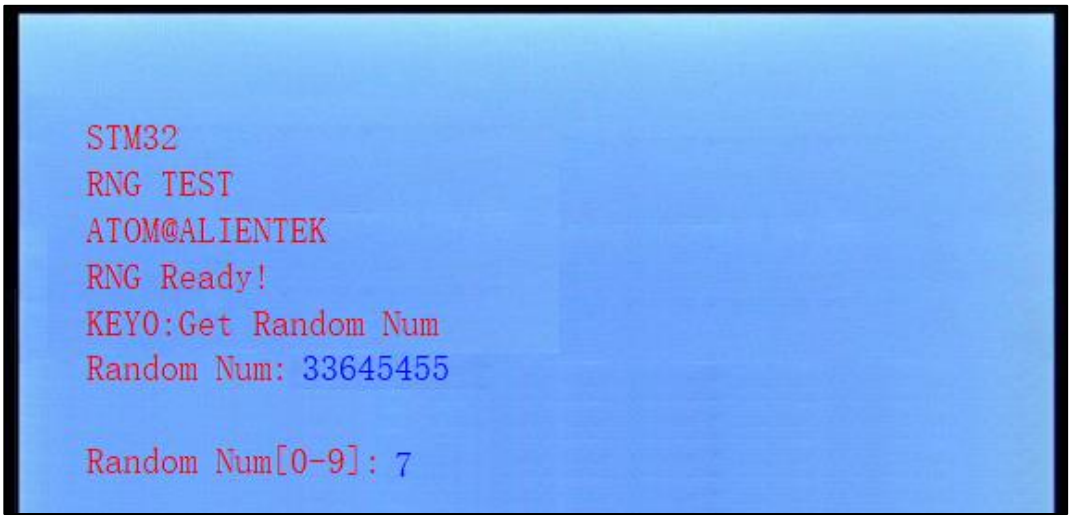


图 23.4.1 获取随机数成功

第二十四章 低功耗实验

本章，我们将介绍 STM32F407 的电源控制（PWR），并实现低功耗模式相关功能。我们将通过四个实验来学习并实现低功耗相关功能，分别是 PVD 电压监控实验、睡眠模式实验、停止模式实验和待机模式实验。

本章分为如下几个小节：

24.1 电源控制（PWR）简介
24.2 PVD 电压监控实验
24.3 睡眠模式实验
24.4 停止模式实验
24.5 待机模式实验

24.1 电源控制（PWR）简介

电源控制部分（PWR）概述了不同电源域的电源架构以及电源配置控制器。PWR 的内容比较多，我们把它们的主要特性概括为以下 3 点：

电源系统：USB 稳压器、内核域(V_{CORE})、VDD 域、备份域、模拟域（V_{DDA}）。

电源监控：POR/PDR 监控器、BOR 监控器、PVD 监控器、AVD 监控器、VBAT 阈值、温度阈值。

电源管理：VBAT 电池充电、工作模式、电压调节控制、低功耗模式。

下面将分别对这 3 个特性进行简单介绍。

24.1.1 电源系统

为了方便对电源系统进行管理，设计者把 STM32 的内核和外设等器件根据功能划分了不同的电源区域，具体如图 24.1.1.1 所示。

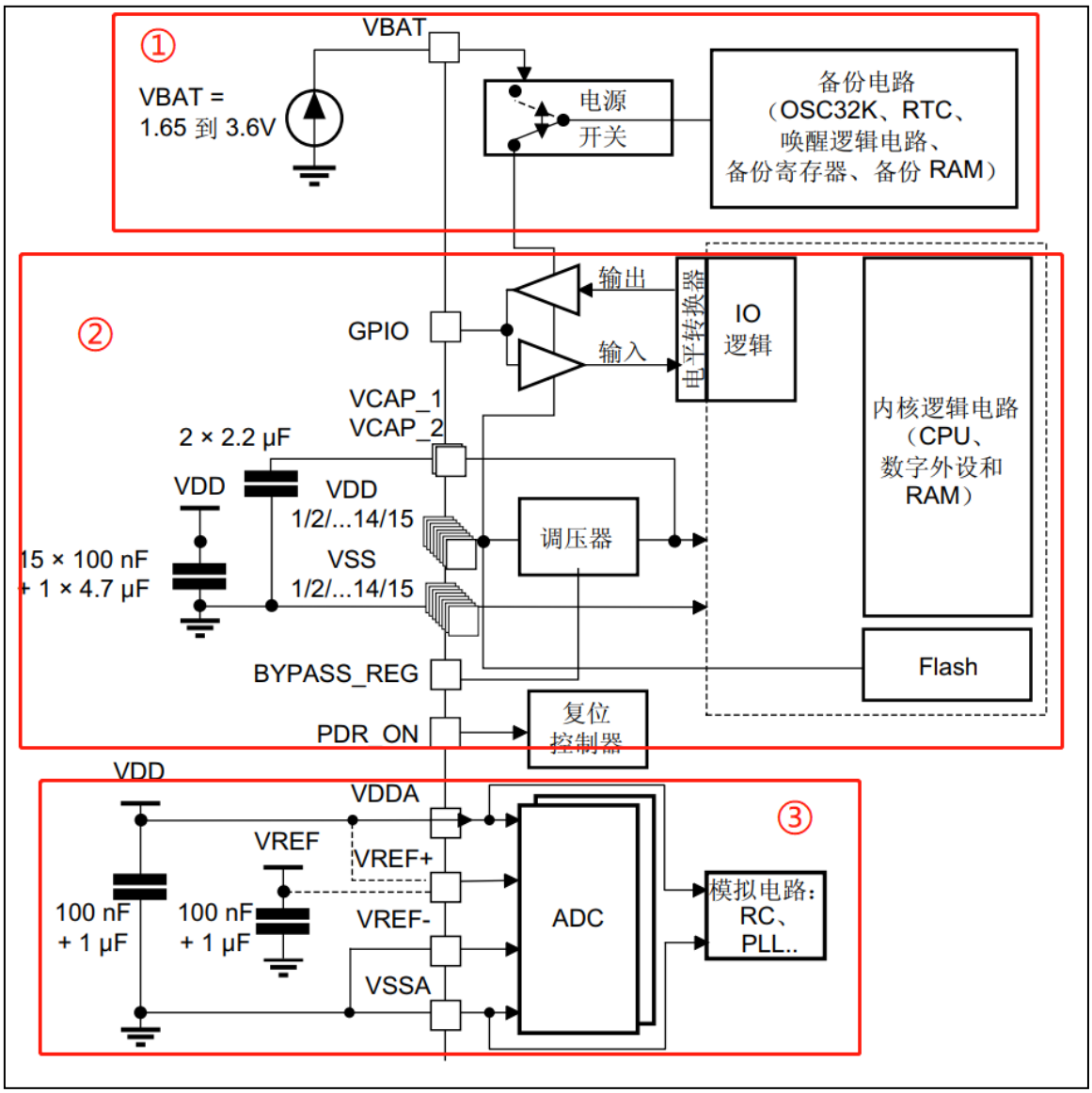


图 24.1.1.1 电源概述框图

在电源概述框图中我们划分了 3 个区域① ② ③，分别是独立的 A/D 转换器供电和参考电压、电压调节器、电池备份区域。下面分别进行简单介绍：

① 电池备份区域（后备供电区域）

电池备份区域也就是后备供电区域，使用电池或者其他电源连接到 V_{BAT} 脚上，当 V_{DD} 断电时，可以保存备份寄存器的内容和维持 RTC 的功能。同时 VBAT 引脚也为 RTC、SRAM 和 LSE 振荡器供电，这保证了当主要电源被切断时，RTC 能够继续工作。切换到 V_{BAT} 供电由复位模块中的掉电复位功能控制。

② 电压调节器（VDD /1.8V 供电区域）

嵌入式线性调压器为备份域和待机电路以外的所有数字电路供电。调压器输出电压约为 1.2 V。此调压器需要将两个外部电容连接到专用引脚 VCAP_1 和 VCAP_2，所有封装都配有这两个引脚。为激活或停用调压器，必须将特定引脚连接到 VSS 或 VDD。具体引脚与封装有关。通过软件激活时，调压器在复位后始终处于使能状态。

根据应用模式的不同，可采用三种不同的模式工作。在运转模式下，调节器以正常工号模式为内核、内存和外设提供 1.2V；在停止模式下，调节器以低功耗模式提供 1.2V 电源，以保存寄存器和 SRAM 的内容。在待机模式下，调节器停止供电，除了备用电路和备份域外，寄存器和 SRAM 的内容全部丢失。

③ 独立的 A/D 转换器供电和参考电压（VDDA 供电区域）

为了提高转换精度，ADC 配有独立电源，可以单独滤波并屏蔽 PCB 上的噪声。ADC 电源电压从单独的 VDDA 引脚接入，VSSA 提供了独立的电源接地连接。为了确保测量低电压时具有更高的精度，用户可以再 VREF 上连接到单独的 ADC 外部参考电压输入，VREF 电压介于 1.8V 到 VDDA 之间。

24.1.2 电源监控

电源监控的部分我们主要关注 PVD 监控器，此外还需要知道上电复位(POR)/掉电复位(PDR)。其他部分的内容请大家查看《STM32F4xx 参考手册_V4（中文版）》（第 5.2 节 90 页）。

- **上电复位(POR)/掉电复位(PDR)**

上电时，当 V_{DD} 低于指定 V_{POR} 阈值时，系统无需外部复位电路便会保持复位模式。一旦 V_{DD} 电源电压高于 V_{POR} 阈值，系统便会退出复位状态，芯片正常工作。掉电时，当 V_{DD} 低于指定 V_{PDR} 阈值时，系统就会保持复位模式。如图 24.1.2.1 所示。

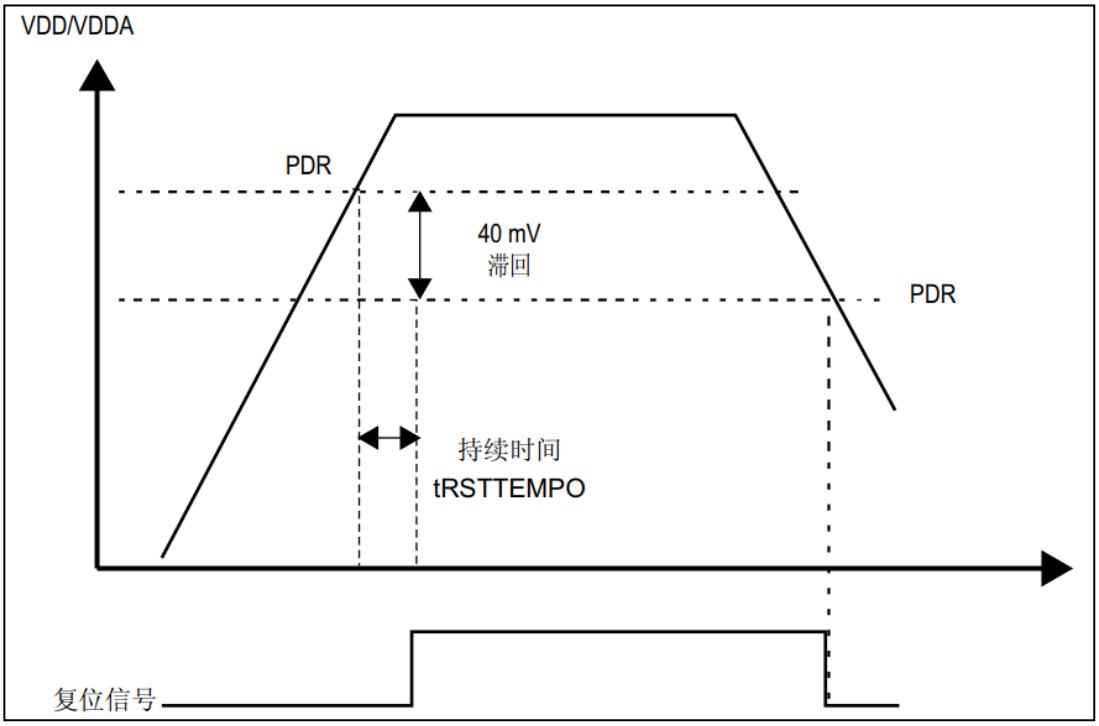


图 24.1.2.1 上电复位/掉电复位波形

- **欠压复位(BOR)**

上电期间，欠压复位(BOR)将使系统保持复位状态，直到 V_{DD} 电源电压达到指定的 V_{BOR} 阈值。V_{BOR} 阈值通过系统选项字节（某些寄存器的 BOR_LEV 位）进行配置。默认情况下，BOR 关闭。可选择以下可编程 V_{BOR} 阈值：

欠压复位（BOR）	电压范围
BOR 关闭 (VBOR0)	1.80 V 到 2.10 V 电压范围的复位阈值级别
BOR 级别 1 (VBOR1)	2.10 V 到 2.40 V 电压范围的复位阈值级别
BOR 级别 2 (VBOR2)	2.40 V 到 2.70 V 电压范围的复位阈值级别
BOR 级别 3 (VBOR3)	2.40 V 到 2.70 V 电压范围的复位阈值级别

欠压复位的描述波形图如图 24.1.2.2 所示。

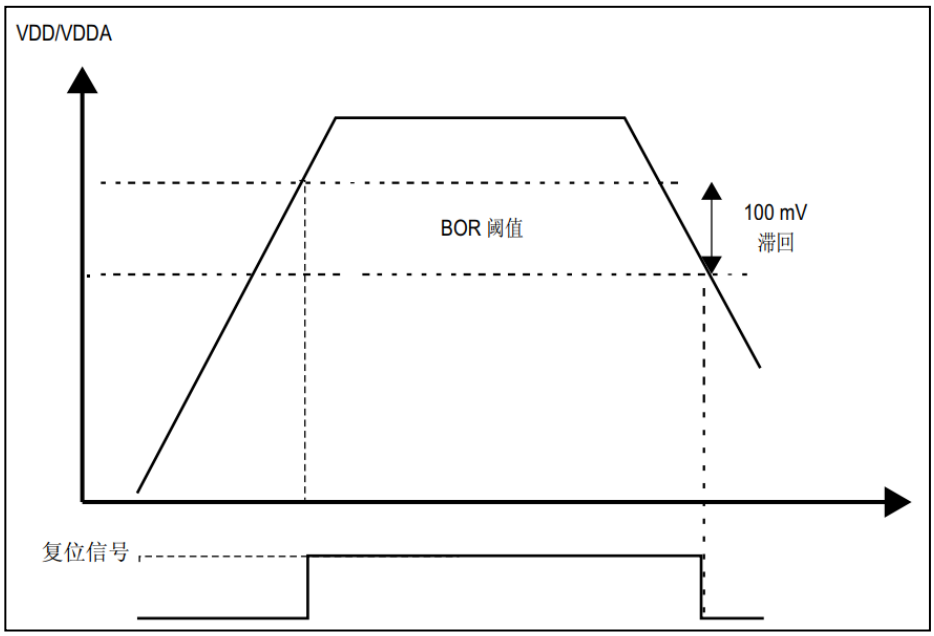


图 24.1.2.2 欠压复位波形

- **可编程电压检测器(PVD)**

上面介绍的 POR、PDR 以及 BOR 功能都是设置电压阈值与外部供电电压 V_{DD} 比较，当 V_{DD} 低于设置的电压阈值时，就会直接进入复位状态，防止电压不足导致的误操作。

下面介绍可编程电压检测器（PVD），它可以实时监视 V_{DD} 的电压，方法是将 V_{DD} 与 PWR 控制寄存器 1（PWR_CR1）中的 PLS[2:0]位所选的 V_{PVD} 阈值进行比较。当检测到电压低于 V_{PVD} 阈值时，如果使能 EXTI16 线中断(即使能 PVD & AVD 中断)，可以产生 PVD 中断，具体取决于 EXTI16 线配置为检测上升还是下降沿，然后在复位前，在中断服务程序中执行紧急关闭系统等任务。PVD 阀值检测波形，如图 24.1.2.1 所示。

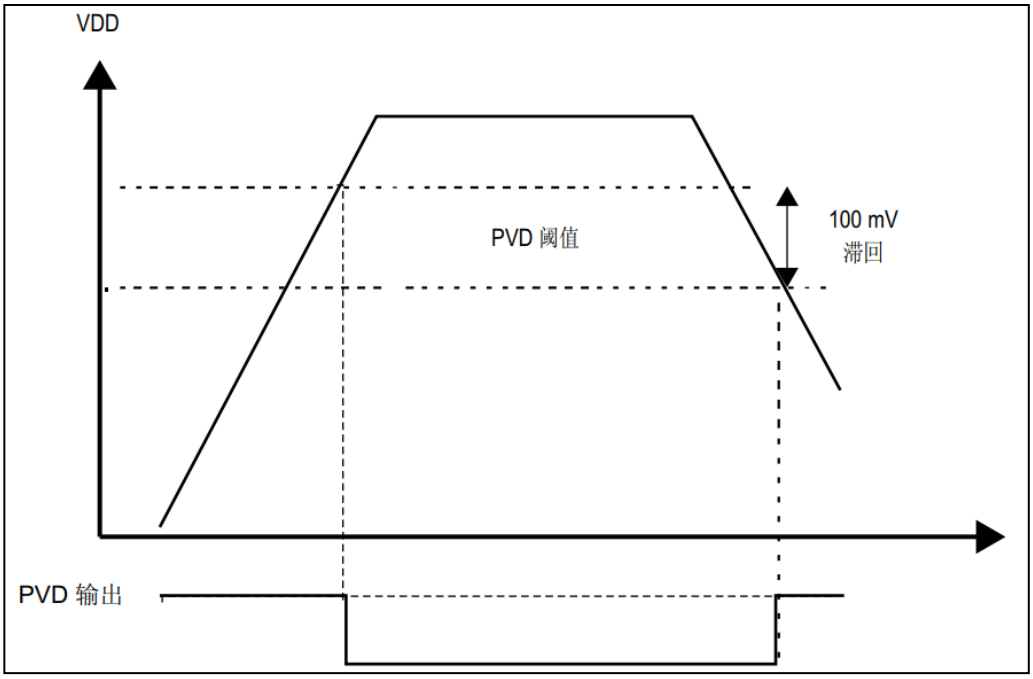


图 24.1.2.3 PVD 检测波形

PVD 阀值有 8 个等级，有上升沿和下降沿的区别，分别就是图 24.1.2.3 中 PVDrise 电压为上升沿阀值，PVDfall 为下降沿阀值，具体如表 24.1.2.2 所示。

Symbol	Parameter	Conditions	Min	Typ	Max
V_{PVD0}	Programmable Voltage Detector threshold 0	Rising edge	1.90	1.96	2.01
		Falling edge	1.81	1.86	1.91
V_{PVD1}	Programmable Voltage Detector threshold 1	Rising edge	2.05	2.10	2.16
		Falling edge	1.96	2.01	2.06
V_{PVD2}	Programmable Voltage Detector threshold 2	Rising edge	2.19	2.26	2.32
		Falling edge	2.10	2.15	2.21
V_{PVD3}	Programmable Voltage Detector threshold 3	Rising edge	2.35	2.41	2.47
		Falling edge	2.25	2.31	2.37
V_{PVD4}	Programmable Voltage Detector threshold 4	Rising edge	2.49	2.56	2.62
		Falling edge	2.39	2.45	2.51
V_{PVD5}	Programmable Voltage Detector threshold 5	Rising edge	2.64	2.71	2.78
		Falling edge	2.55	2.61	2.68
V_{PVD6}	Programmable Voltage Detector threshold 6	Rising edge	2.78	2.86	2.94
		Falling edge in Run mode	2.69	2.76	2.83

表 24.1.2.2 PVD 阀值等级

24.1.3 电源管理

电源管理的部分我们要关注低功耗模式，在 STM32 的正常工作中，具有四种工作模式，运行、睡眠、停止以及待机。在上电复位后，STM32 处于运行状态时，当内核不需要继续运行，就可以选择进入后面的三种模式降低功耗。这三种低功耗模式电源消耗不同、唤醒时间不同和唤醒源不同，我们要根据自身的需要选择合适的低功耗模式。

下面是低功耗模式汇总介绍，如下表所示。

模式名称	进入	唤醒	对 1.2 V 域时钟的影响	对 V_{DD} 域时钟的影响	调压器
睡眠（立即休眠或退出时休眠）	WFI	任意中断	CPU CLK 关闭对其它时钟或模拟时钟源无影响	无	开启
	WFE	唤醒事件			
停止	PDDS 和 LPDS 位 + SLEEPDEEP 位 + WFI 或 WFE	任意 EXTI 线（在 EXTI 寄存器中配置，内部线和外部线）	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	开启或处于低功耗模式（取决于 *用于 STM32F405xx/07xx 和 STM32F415xx/17xx 的 PWR 电源控制寄存器 (PWR_CR)* 和 *用于STM32F42xxx 和 STM32F43xxx 的 PWR 电源控制寄存器 (PWR_CR)*）
待机	PDDS 位 + SLEEPDEEP 位 + WFI 或 WFE	WKUP 引脚上升沿、RTC 闹钟（闹钟 A 或闹钟 B）、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位、IWDG 复位	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	关闭

表 24.1.3.1 低功耗模式汇总

下面对睡眠模式、停止模式和待机模式的进入及退出方法进行介绍。

1、睡眠模式

进入睡眠模式，CPU 时钟关闭，但是其他所有的外设仍可以运行，所以任何中断或事件都

可以唤醒睡眠模式。有两种方式进入睡眠模式，这两种方式进入的睡眠模式唤醒的方法不同，分别是 WFI(wait for interrupt)和 WFE(wait for event)，即由等待“中断”唤醒和由“事件”唤醒。下面我们看看睡眠模式进入及退出方法：

睡眠模式	说明
进入模式	立即睡眠：在执行 WFI 或 WFE 指令时立刻进入睡眠模式 触发条件：内核寄存器 SLEEPDEEP 以及 SLEEPONEXIT 位置 0，调用 WFI 或 WFE 指令立刻进入睡眠模式
	退出时睡眠：在退出优先级最低的中断服务程序后才进入睡眠模式 触发条件 ：内核寄存器 SLEEPDEEP 以及 SLEEPONEXIT 位置 1，从中断服务程序返回后立刻睡眠，这种情况下，处理器的所有工作就只是响应中断了，其他时间在睡眠
退出模式	如果执行 WFI 进入睡眠模式，可使用任意中断唤醒 如果执行 WFE 进入睡眠模式，可使用事件唤醒
唤醒延迟	无延时

表 24.1.3.2 睡眠模式进入及退出方法

2、停止模式

进入停止模式，所有的时钟都关闭，所有的外设也就停止了工作。但是 V_{DD} 电源是没有关闭的，所以内核的寄存器和内存信息都保留下来，等待重新开启时钟就可以从上次停止的地方继续执行程序。

值得注意的是：当电压调节器处于低功耗模式下，当系统从停止模式退出时，将会有一段额外的启动延时。如果在停止模式期间保持内部调节器开启，则退出启动时间会缩短，但相应的功耗会增加。

停止模式	说明
进入模式	内核寄存器的 SLEEPDEEP 位置 1，PWR_CR 寄存器的 PDDS 置 0，然后调用 WFI 或 WFE 指令即可进入停止模式 注意：要进入停止模式，所有 EXIT 线挂起位、RTC 闹钟 A、RTC 闹钟 B、RTC 唤醒、RTC 入侵和 RTC 时间戳标志位必须复位，否则将不会进入停止模式，继续执行程序。
退出模式	如果执行 WFI 进入停止模式： 设置任一外部中断线为中断模式进行中断唤醒 如果执行 WFE 进入停止模式： 设置任一外部中断线为事件模式进行中断唤醒
唤醒延迟	HSI RC 唤醒时间 + 电压调节器从低功耗唤醒的时间。

表 24.1.3.2 睡眠模式进入及退出方法

3、待机模式

待机模式可实现最低功耗。该模式是在 CM4 深睡眠模式时关闭电压调节器，整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。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 和其他寄存器内容都将丢失。不过如果我们使能了备份区域（备份 SRAM、RTC、LSE），那么待机模式下的功耗，将达到 6uA 左右。

那么我们如何进入待机模式呢？其实很简单，只要按表 24.1.3.3 所示的步骤执行就可以了：

待机模式	说明
进入模式	进入待机模式步骤如下： ① 设置 CM4 系统控制寄存器中的 SLEEPDEEP 位置 1 ② 设置电源控制寄存器（PWR_CR）中 PDDS 位置 1 ③ 设置电源控制寄存器（PWR_CR）中的 WUF 位置 0 ④ 调用 WFI 或 WFE 指令即可进入待机模式
退出模式	退出条件分别如下： ① WKUP 引脚的上升沿 ② RTC 闹钟事件的上升沿

	③ NRST 引脚上外部复位 ④ ，IWDG 复位
唤醒延迟	复位阶段时电压调节器的启动

表 24.1.3.3 待机模式进入及退出方法

24.2 PVD 电压监控实验

本小节我们来学习 PVD 电压监控实验，该部分的知识点内容请回顾 24.1.2 电源监控。我们直接从寄存器介绍开始。

24.2.1 PWR 寄存器

本实验用到 PWR 的部分寄存器，在《STM32F4xx 参考手册（中文版）》手册的第 5.4 小节（100 页）可以找到 PWR 的寄存器描述。这里我们只介绍 PVD 电压监控实验我们用到的 PWR 的控制寄存器（PWR_CR），还有就是我们要用到 EXTI16 线中断，所以还要配置 EXTI 相关的寄存器，具体如下：

- **PWR 控制寄存器（PWR_CR）**

PWR 控制寄存器 1 描述如图 24.2.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	VOS	Reserved				FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
	rw					rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

位 7:5 **PLS[2:0]**：PVD 级别选择 (PVD level selection)
这些位由软件写入，用于选择电压检测器检测的电压阈值
000：2.0 V
001：2.1 V
010：2.3 V
011：2.5 V
100：2.6 V
101：2.7 V
110：2.8 V
111：2.9 V
注意：有关详细信息，请参见数据手册的电气特性。

位 4 **PVDE**：使能电源电压检测器 (Power voltage detector enable)
此位由软件置 1 和清零。
0：禁止 PVD
1：使能 PVD

图 24.2.1.1 PWR_CR 寄存器（部分）

位[7:5]PLS 用于设置 PVD 检测的电压阀值，就是前面我们介绍 PVD 的 8 个等级阀值选择。PVDE 位，用于使能或者禁止 PVD 检测，显然我们要使能 PVD 检测，该位置 1。

这个寄存器还有其它的位我们没有列出来，也是跟电源相关的，如待机，掉电等，我们后面的实验再讲解这些功能，这里先跳过。

- **EXTI 中断屏蔽寄存器（EXTI_IMR）**

EXTI 中断屏蔽寄存器描述如图 24.2.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:23 保留，必须保持复位值。

位 22:0 **MRx**：x 线上的中断屏蔽 (Interrupt mask on line x)
0：屏蔽来自 x 线的中断请求
1：开放来自 x 线的中断请求

图 24.2.1.2 EXTI_IMR 寄存器

我们要使用到 EXTI16 线中断，所以 MR16 位要置 1，即开放来自 EXTI16 线的中断请求。

- **EXTI 上升沿触发选择寄存器（EXTI_RTSR）**

EXTI 上升沿触发选择寄存器描述如图 24.2.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:23 保留，必须保持复位值。

位 22:0 **TRx**：线 x 的上升沿触发事件配置位 (Rising trigger event configuration bit of line x)
0：禁止输入线上升沿触发（事件和中断）
1：允许输入线上升沿触发（事件和中断）

注意：外部唤醒线配置为边沿触发时，在这些线上不能出现毛刺信号。
如果在向 EXTI_RTSR 寄存器写入值的同时外部中断线上产生上升沿，挂起位将被置位。
在同一中断线上，可以同时设置上升沿和下降沿触发。即任一边沿都可触发中断。

图 24.2.1.4 EXTI_RTSR 寄存器

我们要使用到 EXTI16 线中断，所以 TR16 位要置 1，即允许 EXTI16 线上的下降沿触发。

- **EXTI 下降沿触发选择寄存器（EXTI_FTSR）**

EXTI 下降沿触发选择寄存器描述如图 24.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:23 保留，必须保持复位值。

位 22:0 **TRx**：线 x 的下降沿触发事件配置位 (Falling trigger event configuration bit of line x)
0：禁止输入线下降沿触发（事件和中断）
1：允许输入线下降沿触发（事件和中断）

注意：外部唤醒线配置为边沿触发时，在这些线上不能出现毛刺信号。
如果在向 EXTI_FTSR 寄存器写入值的同时外部中断线上产生下降沿，挂起位不会被置位。
在同一中断线上，可以同时设置上升沿和下降沿触发。即任一边沿都可触发中断。

图 24.2.1.4 EXTI_FTSR 寄存器

我们要使用到 EXTI16 线中断，所以 TR16 位要置 1，即允许 EXTI16 线上的下降沿触发。

- **EXTI 挂起寄存器（EXTI_PR）**

EXTI 挂起寄存器描述如图 24.2.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

位 31:23 保留，必须保持复位值。

位 22:0 **PRx**：挂起位 (Pending bit)

0：没有发生触发请求

1：发生了选择的触发请求

当在外部中断线上发生了选择的边沿事件，该位被置“1”。在此位中写入“1”可以清除它，也可以通过改变边沿检测的极性清除。

图 24.2.1.5 EXTI_PR 寄存器

EXTI 挂起寄存器 EXTI_PR 管理的是 EXTI0 线到 EXTI19 线的中断标志位。在 PVD 中断服务函数里面，我们记得要对 PR16 位写 1，来清除 EXTI16 线的中断标志。

24.2.2 硬件设计

1. 例程功能

开发板供电正常的话，LCD 屏会显示"PVD Voltage OK!"。当供电电压过低，则会通过 PVD 中断服务函数将 LED1 点亮；当供电电压正常，会在 PVD 中断服务函数将 LED1 熄灭。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯

LED0 - PE0

LED1 - PE1

2）PVD(可编程电压监测器)

3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PVD 属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 LED0 和 LCD 来指示进入 PVD 中断的情况。

24.2.3 程序设计

24.2.3.1 PWR 的 HAL 库驱动

PWR 在 HAL 库中的驱动代码在 stm32f4xx_hal_pwr.c 文件（及其头文件）中。

1. HAL_PWR_ConfigPVD 函数

PVD 的初始化函数，其声明如下：

```
void HAL_PWR_ConfigPVD(PWR_PVDTypeDef *sConfigPVD);
```

- **函数描述：**

  用于初始化 PWR。

- **函数形参：**

  形参 1 是 PWR_PVDTypeDef 结构体类型变量，其定义如下：

```
typedef struct
{
  uint32_t PVDLevel; /* 指定 PVD 检测级别 */
  uint32_t Mode;     /* 指定 PVD 的 EXTI 检测模式 */
```

```
}PWR_PVDTypeDef;
```

1）PVDLevel：指向 PVD 检测级别，对应 PWR_CR 寄存器的 PLS 位的设置，取值范围 PWR_PVDLEVEL_0 到 PWR_PVDLEVEL_7，共八个级别。

2）Mode：指定 PVD 的 EXTI 边沿触发模式。

- **函数返回值：**

无

PVD 电压监控配置步骤

1）配置 PVD，使能 PVD 时钟。

调用 HAL_PWR_ConfigPVD 函数配置 PVD，包括检测电压级别、使用中断线触发方式等。

2）使能 PVD 检测，配置 PVD/AVD 中断优先级，开启 PVD 中断。

通过 HAL_PWR_EnablePVD 函数使能 PVD 检测。

通过 HAL_NVIC_EnableIRQ 函数使能 PVD 中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

3）编写中断服务函数。

PVD 中断服务函数为 PVD_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库有专门的 PVD 中断处理函数，我们只需要在 PVD 中断服务函数里面调用 HAL_PWR_PVD_IRQHandler() 函数，然后逻辑代码在 PVD 中断服务回调函数 HAL_PWR_PVDCallback 中编写，详见本例程源码。

24.2.3.2 程序流程图

下面看看本实验的程序流程图：

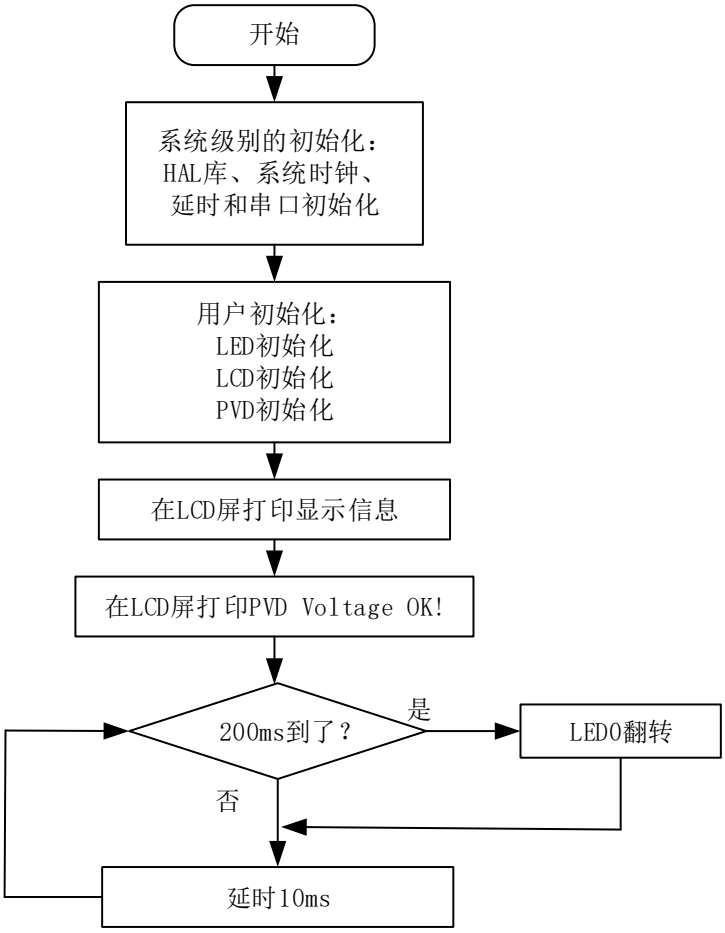


图 24.2.3.2.1 PVD 电压监控实验程序流程图

24.2.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。本章节有四个实验，每一个实验的代码都是在上一个实验后面追加。

pwr.h 头文件只有函数声明，下面直接开始介绍 pwr.c 的程序，首先是 PVD 初始化函数。

```
/**
 * @brief       初始化 PVD 电压监视器
 * @param       pls: 电压等级(PWR_PVD_detection_level)
 *   @arg       PWR_PVDLEVEL_0,2.2V;
 *   @arg       PWR_PVDLEVEL_1,2.3V;
 *   @arg       PWR_PVDLEVEL_2,2.4V;
 *   @arg       PWR_PVDLEVEL_3,2.5V;
 *   @arg       PWR_PVDLEVEL_4,2.6V;
 *   @arg       PWR_PVDLEVEL_5,2.7V;
 *   @arg       PWR_PVDLEVEL_6,2.8V;
 *   @arg       PWR_PVDLEVEL_7,2.9V;
 * @retval      无
 */
void pwr_pvd_init(uint32_t pls)
{
    PWR_PVDTypeDef pvd_handle = {0};

    __HAL_RCC_PWR_CLK_ENABLE();             /*使能 PWR 时钟*/

    pvd_handle.PVDLevel = pls;
   /* 使用中断线的上升沿和下降沿双边缘触发 */
    pvd_handle.Mode = PWR_PVD_MODE_IT_RISING_FALLING;
    HAL_PWR_ConfigPVD(&pvd_handle);         /* 开启 PWR 对应的 lin16 上的中断 */

    HAL_NVIC_SetPriority(PVD_IRQn, 3 ,3);   /* 组 3，最低优先级 */
    HAL_NVIC_EnableIRQ(PVD_IRQn);
    HAL_PWR_EnablePVD();
}
```

这里需要注意的就是 PVD 中断线选择的是上升沿和下降沿双边缘触发，其他的内容前面已经讲过。

下面介绍的是 PVD 中断服务函数及其回调函数，函数定义如下：

```
/**
 * @brief       PVD 中断服务函数
 * @param       无
 * @retval      无
 */
void PVD_IRQHandler(void)
{
    HAL_PWR_PVD_IRQHandler();
}

/**
 * @brief       PVD 中断服务回调函数
 * @param       无
 * @retval      无
 */
void HAL_PWR_PVDCallback(void)
{
   if (__HAL_PWR_GET_FLAG(PWR_FLAG_PVDO))   /* 电压比 PLS 所选电压还低 */
   {
       /* LCD 显示电压低 */
        lcd_show_string(30, 130, 200, 16, 16, "PVD Low Voltage!", RED);
        LED1(0);    /* 点亮 LED1，表明电压低了 */
    }
    else
   {
```

```
        /* LCD 显示电压正常 */
        lcd_show_string(30, 130, 200, 16, 16, "PVD Voltage OK! ", BLUE);
        LED1(1);    /* 灭掉绿灯 */
    }
}
```

HAL_PWR_PVDCallback 回调函数中首先是判断 V_{DD} 电压是否比 PLS 所选电压还低，是的话，就在 LCD 显示 PVD Low Voltage!并且点亮 LED1，否则，在 LCD 显示 PVD Voltage OK!并且关闭 LED1。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t t=0;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    pwr_pvd_init(PWR_PVDLEVEL_7);       /* PVD 2.9V 检测 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "PVD TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    /* 默认 LCD 显示电压正常 */
    lcd_show_string(30, 130, 200, 16, 16, "PVD Voltage OK! ", BLUE);

    while (1)
    {
        if ((t % 20) == 0)
        {
            LED0_TOGGLE();                  /* 每 200ms,翻转一次 LED0 */
        }

        delay_ms(10);
        t++;
    }
}
```

这里我们选择 PVD 的检测电压阀值为 2.9V，其他的代码很好理解，最后下载验证一下。

24.2.4 下载验证

下载代码后，默认 LCD 屏会显示"PVD Voltage OK!"，当供电电压过低，则 LED1 会点亮，并且 LCD 屏会显示 PVD Low Voltage!。当开发板供电正常，LED1 会熄灭，LCD 屏会继续显示"PVD Voltage OK!"。

24.3 睡眠模式实验

本小节我们来学习睡眠模式实验，该部分的知识点内容请回顾 24.1.2.3 电源管理。我们直接从寄存器介绍开始。

24.3.1 EXTI 寄存器

本实验我们用到外部中断来唤醒睡眠模式。我们用到 WFI 指令（参考 24.1.3 电源管理对这个指令的讲解）进入睡眠模式，这个后面会讲，进入睡眠模式后，使用外部中断唤醒。进入外部中断后，EXTI_IMR 寄存器的值会自动清零，我们需要对对应的外部中断线位置 1，取消屏蔽，相当于其他中断的中断标志位进入中断后硬件自动置 1，需要手动清零。

- **EXTI 中断屏蔽寄存器（EXTI_IMR）**

EXTI 中断屏蔽寄存器描述如图 24.3.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:23 保留，必须保持复位值。

位 22:0 **MRx**：x 线上的中断屏蔽 (Interrupt mask on line x)
0：屏蔽来自 x 线的中断请求
1：开放来自 x 线的中断请求

图 24.3.1.1 EXTI_IMR 寄存器

本实验使用 KEY1（PE3）唤醒，即 EXTI3 线中断，所以在外部中断服务函数要把 MR3 位置 1。

- **EXTI 上升沿触发选择寄存器（EXTI_RTSR）**

EXTI 上升沿触发选择寄存器描述如图 24.3.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:23 保留，必须保持复位值。

位 22:0 **TRx**：线 x 的上升沿触发事件配置位 (Rising trigger event configuration bit of line x)
0：禁止输入线上升沿触发（事件和中断）
1：允许输入线上升沿触发（事件和中断）

注意： *外部唤醒线配置为边沿触发时，在这些线上不能出现毛刺信号。*
如果在向 EXTI_RTSR 寄存器写入值的同时外部中断线上产生上升沿，挂起位将被置位。
在同一中断线上，可以同时设置上升沿和下降沿触发。即任一边沿都可触发中断。

图 24.3.1.2 EXTI_RTSR 寄存器

我们要使用到 EXTI3 线中断，所以 TR3 位要置 1，即 EXTI3 使用的是上升沿进行触发。

- **EXTI 挂起寄存器（EXTI_PR）**

EXTI 挂起寄存器描述如图 24.3.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

位 31:23 保留，必须保持复位值。

位 22:0 **PRx**：挂起位 (Pending bit)

0：没有发生触发请求

1：发生了选择的触发请求

当在外部中断线上发生了选择的边沿事件，该位被置“1”。在此位中写入“1”可以清除它，也可以通过改变边沿检测的极性清除。

图 24.3.1.3 EXTI_PR 寄存器

在 EXTI3 中断服务函数里面，需要清除 EXTI3 中断标记，即对 PR3 位写 1。

24.3.2 硬件设计

1. 例程功能

LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入睡眠模式，此时 LED0 不再闪烁，说明已经进入睡眠模式。按下按键 KEY1 后，LED1 熄灭，提示退出睡眠模式，此时 LED0 继续闪烁，说明已经退出睡眠模式。

2. 硬件资源

1）LED 灯

LED0 - PE0

LED1 - PE1

2）独立按键

KEY0 - PE2

KEY1 - PE3

3）电池管理(低功耗模式 - 睡眠模式)

4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PWR 属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 KEY0 让 CPU 进入睡眠模式，再通过 KEY1 触发 EXTI 中断来唤醒 CPU。LED0 指示程序是否执行，LED1 指示 CPU 是否进入睡眠模式。

24.3.3 程序设计

24.3.3.1 PWR 的 HAL 库驱动

1. HAL_PWR_EnterSLEEPMode 函数

进入睡眠模式函数，其声明如下：

```
void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry);
```

- **函数描述：**

  用于设置 CPU 进入睡眠模式。

- **函数形参：**

  形参 1 指定稳压器的状态。有两个选择，PWR_MAINREGULATOR_ON 表示稳压器处于正常模式，PWR_LOWPOWERREGULATOR_ON 表示稳压器处于低功耗模式。对应的

是 PWR_CR 寄存器的 LPDS 位的设置（该形参在该函数中没有实质用处）。

形参 2 指定进入睡眠模式的方式。有两个选择，PWR_SLEEPENTRY _WFI 表示使用 WFI 指令，PWR_SLEEPENTRY_WFE 表示使用 WFE 指令。我们选择前者，这两个指令的区别，请参考本教程 24.1.3 电源管理的描述。

- **函数返回值：**无

睡眠模式配置步骤

1）配置唤醒睡眠模式的方式

这里我们用外部中断的方式唤醒睡眠模式，所以这里需要配置一个外部中断功能，我们用 KEY1 按键作为中断触发源，接下来就是配置 PE3（连接按键 KEY1）。

通过__HAL_RCC_GPIOE_CLK_ENABLE 函数使能 GPIOE 的时钟。

通过 HAL_GPIO_Init 函数配置 PE3 为具有上升边缘触发检测的外部中断模式，开启下拉电阻等。

通过 HAL_NVIC_EnableIRQ 函数使能 EXTI3 中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

编写 EXTI3_IRQHandle 中断函数，在中断服务函数中调用 HAL_GPIO_EXTI_IRQHandler 函数。

最后编写 HAL_GPIO_EXTI_Callback 回调函数。由于前面已经介绍过外部中断的配置步骤，这里就介绍到这里，详见本例程源码。

2）进入 CPU 睡眠模式

通过 HAL_PWR_EnterSLEEPMode 函数进入睡眠模式。

3）通过按下按键触发外部中断唤醒睡眠模式

在本实验中，通过按下 KEY0 按键进入睡眠模式，然后通过按下 KEY1 按键触发外部中断唤醒睡眠模式。

24.3.3.2 程序流程图

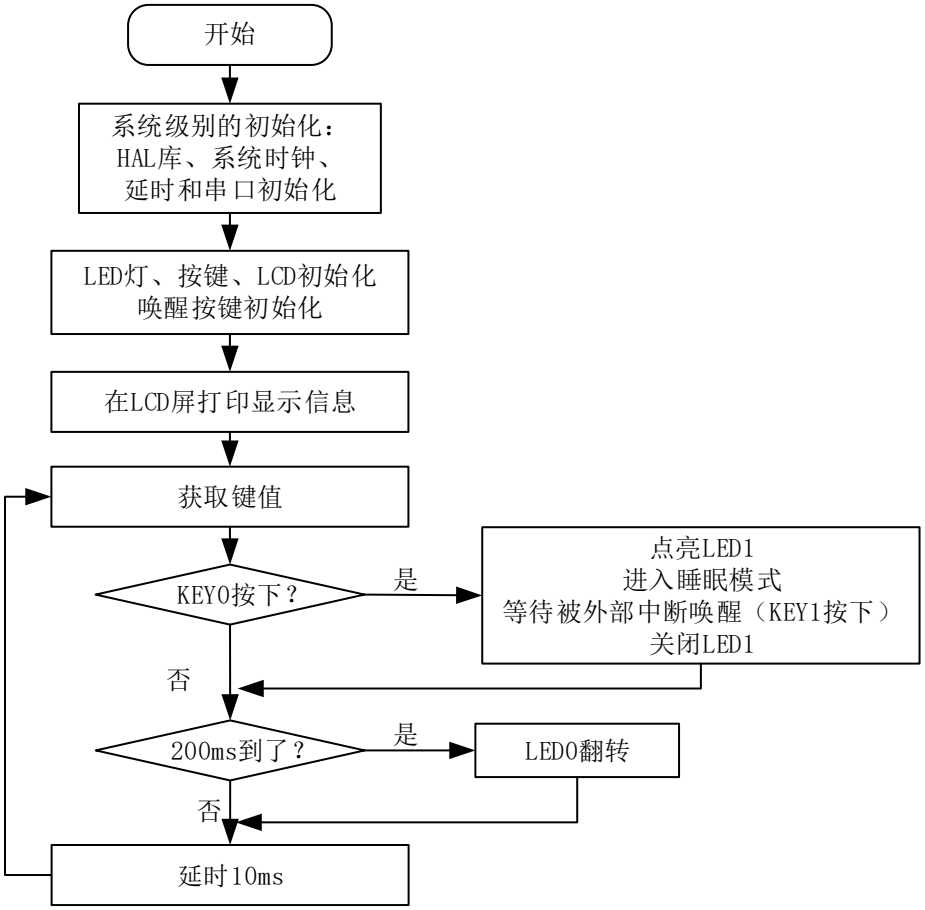


图 24.3.3.2.1 睡眠模式实验程序流程图

24.3.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。睡眠模式实验代码在电压监控实验源码后追加。

首先看本实验在 pwr.h 头文件定义的几个宏定义：

```
/* PWR KEY 按键 引脚和中断 定义
 * 我们通过 KEY1 按键唤醒 MCU， 因此必须定义这个按键及其对应的中断服务函数
 */
#define PWR_KEY1_GPIO_PORT          GPIOE
#define PWR_KEY1_GPIO_PIN           GPIO_PIN_3
/* PE 口时钟使能 */
#define PWR_KEY1_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOE_CLK_ENABLE();}while(0)

#define PWR_KEY1_INT_IRQn           EXTI3_IRQn
#define PWR_KEY1_INT_IRQHandler     EXTI3_IRQHandler
```

这些定义是 KEY1 按键的相关宏定义，以及其对应的外部中断线 3 的相关定义。

pwr.h 头文件就介绍这部分的程序，下面是 pwr.c 文件，先看低功耗模式下的按键初始化函数，其定义如下：

```
/**
 * @brief       低功耗模式下的按键初始化(用于唤醒睡眠模式/停止模式/待机模式)
 * @param       无
 * @retval      无
 */
void pwr_key_init(void)
{
```

```
    GPIO_InitTypeDef gpio_init_struct;
    PWR_KEY1_GPIO_CLK_ENABLE();                                 /* KEY1 时钟使能 */

    gpio_init_struct.Pin = PWR_KEY1_GPIO_PIN;                   /* KEY1 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_IT_RISING;                /* 中断,上升沿 */
    gpio_init_struct.Pull = GPIO_PULLDOWN;                      /* 下拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;              /* 高速 */
    HAL_GPIO_Init(PWR_KEY1_GPIO_PORT, &gpio_init_struct);       /* KEY1 引脚初始化 */

    HAL_NVIC_SetPriority(PWR_KEY1_INT_IRQn, 2, 2); /* 抢占优先级 2，子优先级 2 */
    HAL_NVIC_EnableIRQ(PWR_KEY1_INT_IRQn);
}
```

该函数初始化 KEY1 按键（PE3），并设置上升沿触发的外部中断线 2，最后设置中断优先级并使能外部中断线 2。

下面介绍的是进入 CPU 睡眠模式函数，其定义如下：

```
/**
 * @brief       进入 CPU 睡眠模式
 * @param       无
 * @retval      无
 */
void pwr_enter_sleep(void)
{
    /* 进入睡眠模式 */
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
}
```

函数内直接调用 HAL_PWR_EnterSLEEPMode 函数使用 WFI 指令进入睡眠模式。

下面介绍的是 KEY1 按键外部中断服务函数及其回调函数，函数定义如下：

```
/**
 * @brief       KEY1 按键 外部中断服务程序
 * @param       无
 * @retval      无
 */
void PWR_KEY1_INT_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(PWR_WKUP_GPIO_PIN);
}

/**
 * @brief       外部中断回调函数
 * @param       GPIO_Pin:中断线引脚
 * @note        此函数会被 PWR_KEY1_INT_IRQHandler()调用
 * @retval      无
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == PWR_KEY1_GPIO_PIN)
    {
        /* HAL_GPIO_EXTI_IRQHandler()函数已经为我们清除了中断标志位，
           所以我们进了回调函数可以只关注中断时的控制逻辑，不用再操作寄存器 */
    }
}
```

在 KEY1 按键外部中断服务函数中我们调用 HAL 库的 HAL_GPIO_EXTI_IRQHandler 函数来处理外部中断。该函数会调用__HAL_GPIO_EXTI_CLEAR_IT 函数取消屏蔽对应的外部中断线位，这里是 EXTI_IMR 寄存器相应位，还有其他寄存器控制其他外部中断线。我们只是唤醒睡眠模式而已，不需要其他的逻辑程序，所以 HAL_GPIO_EXTI_Callback 回调函数可以什么都不用做，甚至也可以不重新定义这个回调函数（屏蔽该回调函数也可以）。

最后在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t=0;
```

```
    uint8_t key = 0;

    HAL_Init();                             /*初始化 HAL 库*/
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */
    pwr_key_init();                         /* 唤醒按键初始化 */

    lcd_show_string(30, 50,  200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70,  200, 16, 16, "SLEEP TEST", RED);
    lcd_show_string(30, 90,  200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Enter SLEEP MODE", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY1:Exit SLEEP MODE", RED);

    while (1)
    {
        key = key_scan(0);
        if (key == KEY0_PRES)
        {
            LED1(0);                    /* 点亮绿灯,提示进入睡眠模式 */
            pwr_enter_sleep();          /* 进入睡眠模式 */
            LED1(1);                    /* 关闭绿灯,提示退出睡眠模式 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGLE();              /* 每 200ms,翻转一次 LED0 */
        }
        delay_ms(10);
        t++;
    }
}
```

该部分程序，功能就是按下 KEY0 后，点亮 LED1，进入睡眠模式。然后一直等待外部中断唤醒，当按下按键 KEY1，就触发外部中断，睡眠模式就被唤醒，然后继续执行后面的程序，关闭 LED1 等。

24.3.4 下载验证

下载代码后，LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入睡眠模式，此时 LED0 不再闪烁，说明已经进入睡眠模式。按下按键 KEY1 后，LED1 熄灭，提示退出睡眠模式，此时 LED0 继续闪烁，说明已经退出睡眠模式。

24.4 停止模式实验

本小节我们来学习停止模式实验，该部分的知识点内容请回顾 24.1.2.3 电源管理。我们直接从寄存器介绍开始。

24.4.1 PWR 寄存器

本实验我们用到外部中断来唤醒停止模式。我们用到 WFI 指令进入停止模式，这个后面会讲，进入停止模式后，使用外部中断唤醒。外部中断部分内容参照睡眠模式即可，都是共用同样的配置。

下面主要介绍 PWR_CR 寄存器相关位。

- **PWR 控制寄存器（PWR_CR）**

PWR 的控制寄存器描述如图 24.4.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	VOS	Reserved				FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
	rw					rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

位 1 **PDDS**：深度睡眠掉电 (Power-down deepsleep)
此位由软件置 1 和清零。与 LPDS 位结合使用。
0：器件在 CPU 进入深度睡眠时进入停止模式。调压器状态取决于 LPDS 位。
1：器件在 CPU 进入深度睡眠时进入待机模式。

位 0 **LPDS**：深度睡眠低功耗 (Low-power deepsleep)
此位由软件置 1 和清零。与 PDDS 位结合使用。
0：停止模式下调压器开启
1：停止模式下调压器进入低功耗模式

图 24.4.1.1 PWR_CR 寄存器

通过 PDDS 位选择进入停止模式还是待机模式，停止模式即对 PDDS 位置 0 即可。在停止模式下，电压调节器有两种模式：开启或者低功耗，选择低功耗模式，即 LPDS 置 1。

- **系统控制寄存器（SCB_SCR）**

系统控制寄存器描述如图 24.4.1.2 所示：

位段	名称	类型	复位值	描述
4	SEVONPEND	RW	-	发生异常悬起时请发送事件，用于在一个新的中断悬起时从 WFE 指令处唤醒。不管这个中断的优先级是否比当前的高，都唤醒。如果没有 WFE 导致睡眠，则下次使用 WFE 时将立即唤醒
3	保留	-	-	-
2	SLEEPDEEP	R/W	0	当进入睡眠模式时，使能外部的 SLEEPDEEP 信号，以允许停止系统时钟
1	SLEEPONEXIT	R/W	-	激活“SleepOnExit”功能
0	保留	-	-	-

图 24.4.1.2 系统控制寄存器

该寄存器存在于 ARM 内核中，详细描述可查阅《Cortex-M4 权威指南》，在本实验中，我们需要把 SLEEPDEEP 位置 1，这样子后面调用 WFI 命令时，进入的就是停止模式了。在唤醒后，需要清除 SLEEPDEEP 位，进行置 0。

24.4.2 硬件设计

1. 例程功能

LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入停止模式，此时 LED0 不再闪烁，说明已经进入停止模式。按下按键 KEY1 后，LED1 熄灭，提示退出停止模式，此时 LED0 继续闪烁，说明已经退出停止模式。

2. 硬件资源

1）LED 灯
LED0 - PE0
LED1 - PE1
2）独立按键
KEY0 - PE2
KEY1 - PE3
3）电池管理(低功耗模式 - 停止模式)
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PWR 属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 KEY0 让 CPU 进入停止模式，再通过 KEY1 触发 EXTI 中断来唤醒 CPU。LED0 指示程序是否执行，LED1 指示 CPU 是否进入停止模式。

24.4.3 程序设计

24.4.3.1 PWR 的 HAL 库驱动

1. HAL_PWR_EnterSTOPMode 函数

进入睡眠模式函数，其声明如下：

```
void HAL_PWR_EnterSTOPMode (uint32_t Regulator, uint8_t STOPEntry);
```

- **函数描述：**
  用于设置 CPU 进入停止模式。
- **函数形参：**
  形参 1 指定稳压器在睡眠模式下的状态。有两个选择，PWR_MAINREGULATOR_ON 表示稳压器处于正常模式，PWR_LOWPOWERREGULATOR_ON 表示稳压器处于低功耗模式。对应的是 PWR_CR1 寄存器的 LPDS 位的设置。
  形参 2 指定用 WFI 还是 WFE 指令进入停止模式。有两个选择，PWR_STOPENTRY_WFI 表示使用 WFI 指令，PWR_STOPENTRY_WFE 表示使用 WFE 指令。我们选择前者，不了解这两种指令的区别，可以回看 24.1.3 小节的知识。
- **函数返回值：**
  无

停止模式配置步骤

1）配置唤醒停止模式的方式

这里我们用外部中断的方式唤醒停止模式，所以这里需要配置一个外部中断功能，我们用 WK_UP 按键作为中断触发源，接下来就是配置 PE3（连接按键 KEY1）。

通过__HAL_RCC_GPIOE_CLK_ENABLE 函数使能 GPIOE 的时钟。

通过 HAL_GPIO_Init 函数配置 PE3 为具有上升边缘触发检测的外部中断模式，开启下拉电阻等。

通过 HAL_NVIC_EnableIRQ 函数使能 EXTI3 中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

编写 EXTI3_IRQHandle 中断函数，在中断服务函数中调用 HAL_GPIO_EXTI_IRQHandler 函数。

最后编写 HAL_GPIO_EXTI_Callback 回调函数。由于前面已经介绍过外部中断的配置步骤，这里就介绍到这里，详见本例程源码。

2）进入 CPU 停止模式

通过 HAL_PWR_EnterSTOPMode 函数进入停止模式。

3）通过按下按键触发外部中断唤醒停止模式

在本实验中，通过按下 KEY0 按键进入停止模式，然后通过按下 KEY1 按键触发外部中断唤醒停止模式。

24.4.3.2 程序流程图

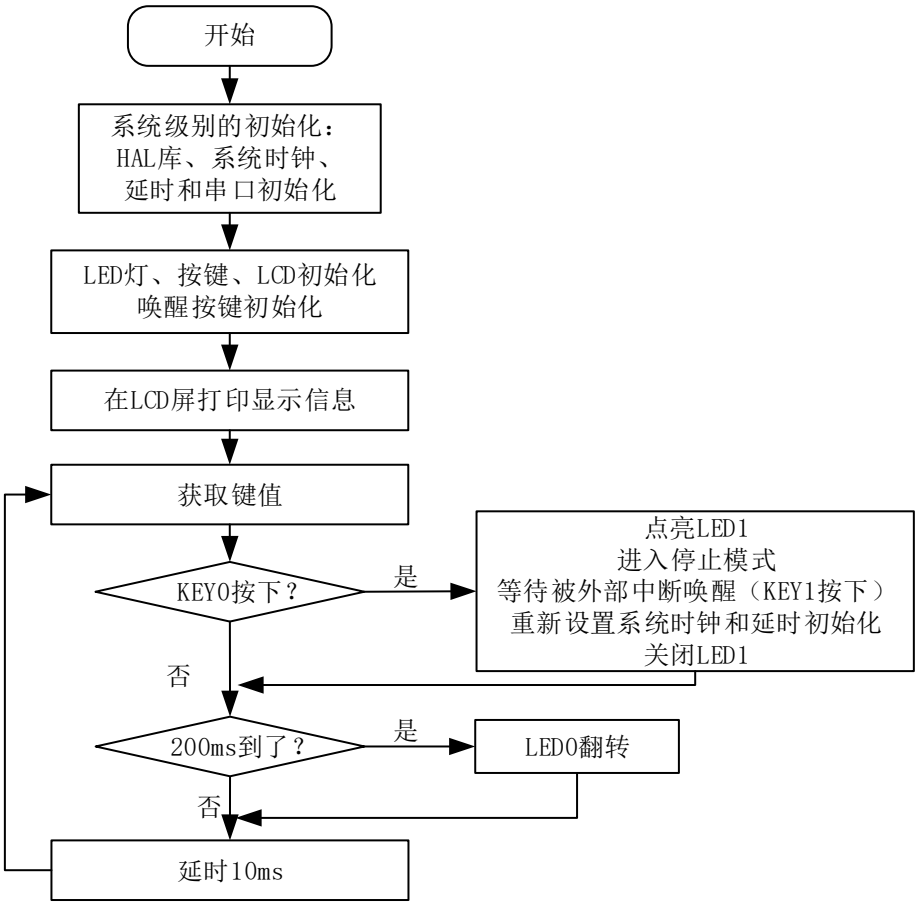


图 24.4.3.2.1 停止模式实验程序流程图

24.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。停止模式实验代码在睡眠模式实验源码后追加。

首先看 pwr.h 头文件，因为我们还是用到 KEY1 对应的外部中断线来唤醒停止模式的 CPU，pwr.h 头文件的 KEY1 按键对应的宏定义我们也是用到的，上个实验已经讲过，这里不再赘述。下面是 pwr.c 文件，KEY1 按键的相关函数我们还是用上个实验的，我们主要介绍进入停止模式函数，其定义如下：

```
/**
 * @brief       进入停止模式
 * @param       无
 * @retval      无
```

```
 */
void pwr_enter_stop(void)
{
    __HAL_RCC_PWR_CLK_ENABLE();
    HAL_SuspendTick();  /* 暂停滴答时钟，防止通过滴答时钟中断唤醒 */
    /* 进入停止模式，掉电并进入低功耗模式并待机 */
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
}
```

该函数先调用__HAL_RCC_PWR_CLK_ENABLE 函数使能 PWR 时钟，然后调用 HAL_SuspendTick 函数暂停滴答时钟，最后调用 HAL_PWR_EnterSTOPMode 函数进入停止模式，形参 1 选择 PWR_LOWPOWERREGULATOR_ON 表示设置稳压器为低功耗模式，形参 2 则是选择 WFI 指令。

最后在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t=0;
    uint8_t key = 0;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */
    pwr_key_init();                         /* 唤醒按键初始化 */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "STOP TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Enter STOP MODE", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY1:Exit STOP MODE", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES)
        {
            LED1(0);            /* 点亮绿灯,提示进入停止模式 */
            pwr_enter_stop();/* 进入停止模式 */

            /* 从停止模式唤醒，需要重新设置系统时钟，168Mhz */
            sys_stm32_clock_init(336, 8, 2, 7);
            delay_init(168);  /* 延时初始化 */
            LED1(1);            /* 关闭绿灯,提示退出停止模式 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGLE();    /* 每 200ms,翻转一次 LED0 */
        }

        delay_ms(10);
        t++;
    }
}
```

该部分程序，功能就是按下 KEY0 后，点亮 LED1、暂停滴答时钟并进入停止模式。然后一直等待外部中断唤醒，当按下按键 KEY1，就触发外部中断，停止模式就被唤醒，然后继续执行后面的程序，重新设置系统时钟 168Mhz 和延时初始化，关闭 LED1 等。

24.4.4 下载验证

下载代码后，LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，LED1 点亮，提示进入停止模式，此时 LED0 不再闪烁，说明已经进入停止模式。按下按键 KEY1 后，LED1 熄灭，提示退出停止模式，此时 LED0 继续闪烁，说明已经退出停止模式。

24.5 待机模式实验

本小节我们来学习待机模式实验，该部分的知识点内容请回顾 24.1.2.3 电源管理。我们直接从寄存器介绍开始。

24.5.1 PWR 寄存器

本实验是先对相关的电源控制寄存器配置待机模式的参数，然后通过 WFI 指令进入待机模式，使用 PA0 引脚的上升沿来唤醒。

下面主要介绍 PWR_CR 寄存器相关位。

- **PWR 控制寄存器（PWR_CR）**

PWR 的控制寄存器描述如图 24.5.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	VOS	Reserved				FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
	rw					rw	rw	rw	rw	rw	rw	rc_w1	rc_w1	rw	rw

位 2 **CWUF**：将唤醒标志清零 (Clear wakeup flag)
此位始终读为 0。
0：无操作
1：写 1 操作 2 个系统时钟周期后将 WUF 唤醒标志清零

位 1 **PDDS**：深度睡眠掉电 (Power-down deepsleep)
此位由软件置 1 和清零。与 LPDS 位结合使用。
0：器件在 CPU 进入深度睡眠时进入停止模式。调压器状态取决于 LPDS 位。
1：器件在 CPU 进入深度睡眠时进入待机模式。

图 24.5.1.1 PWR_CR 寄存器（部分）

这里我们通过设置 PDDS 位，使 CPU 进入深度睡眠时进入待机模式，同时，我们可以通过 CWUF 位清除之前的唤醒位。

- **电源控制/状态寄存器（PWR_CSR）**

电源控制/状态寄存器描述如图 24.5.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
Res.															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	VOS RDY	Reserved				BRE	EWUP	Reserved Res.				BRR	PVDO	SBF	WUF
	r					rw	rw					r	r	r	r

位 8 **EWUP**：使能 WKUP 引脚 (Enable WKUP pin)
此位由软件置 1 和清零。
0：WKUP 引脚用作通用 I/O。WKUP 引脚上的事件不会把器件从待机模式唤醒。
1：WKUP 用于从待机模式唤醒器件并被强制配置成输入下拉（WKUP 引脚出现上升沿时从待机模式唤醒系统）。
注意：此位通过系统复位进行复位。

图 24.5.1.1 PWR_CSR 寄存器

该寄存器我们只关心 EWUP 位，设置 EWUP 为 1，即 PA0 引脚作为待机模式的唤醒源。

24.5.2 硬件设计

1. 例程功能

LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，进入待机模式，待机模式下大部分引脚处于高阻态，所以说这时候 LED0 会熄灭，TFTLCD 也会熄灭。按下 WK_UP 后，退出待机模式（相当于复位操作），程序重新执行，LED0 继续闪烁，TFTLCD 屏点亮。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
WK_UP - PA0(开发板 PA0 没有接按键，需要自己接地测试)
3）电池管理(低功耗模式 - 待机模式)
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

PWR 属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 KEY0 让 CPU 进入待机模式，再通过 WK_UP 上升沿来唤醒 CPU。LED0 指示程序是否执行。

24.5.3 程序设计

24.5.3.1 PWR 的 HAL 库驱动

1. HAL_PWR_EnableWakeUpPin 函数

使能唤醒引脚函数，其声明如下：

```
void HAL_PWR_EnableWakeUpPin(uint32_t WakeUpPinPolarity);
```

- **函数描述：**
  用于使能唤醒引脚。
- **函数形参：.**
  形参 1 取值范围：PWR_WAKEUP_PIN1。
- **函数返回值：**
  无
- **注意事项：**
  禁止某个唤醒引脚使用的函数如下：

```
void HAL_PWR_DisableWakeUpPin(uint32_t WakeUpPinPolarity);
```

2. HAL_PWR_EnterSTANDBYMode 函数

进入待机模式函数，其声明如下：

```
void HAL_PWR_EnterSTANDBYMode(void);
```

- **函数描述：**
  用于使 CPU 进入待机模式，进入待机模式，首先要设置 SLEEPDEEP 位，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 上升沿的到来。
- **函数形参：**
  无
- **函数返回值：**
  无

待机模式配置步骤

1）进入 CPU 停止模式

在进入待机模式之前我们需要做一些准备：
涉及到操作 PWR 寄存器的内容，所以首先先进行 PWR 时钟的初始化，用__HAL_RCC_PWR_CLK_ENABLE 函数实现。
通过 HAL_PWR_EnableWakeUpPin 函数使能 WKUP 的唤醒功能。
通过__HAL_PWR_CLEAR_FLAG 函数清除唤醒标记，详看源码。
通过 HAL_PWR_EnterSTANDBYMode 函数进入待机模式。

2）通过 WKUP 引脚上升沿触发唤醒睡眠模式

在本实验中，通过按下 KEY0 按键进入待机模式，然后通过按下 WK_UP 按键，使用待机模式中的 WKUP 引脚上升沿的唤醒信号，而不是普通的外部中断，唤醒待机模式。

24.5.3.2 程序流程图

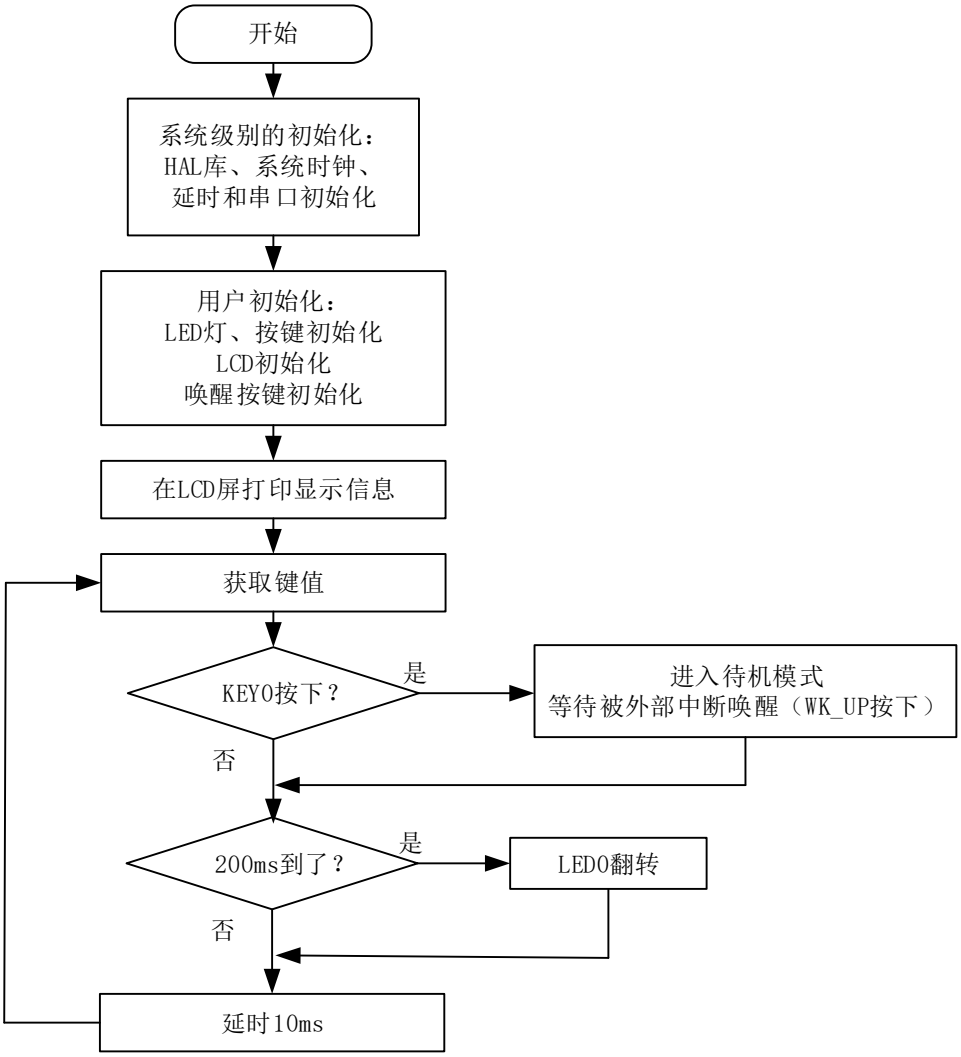


图 24.4.3.2.1 待机模式实验程序流程图

24.5.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。PWR 源码包括两个文件：pwr.c 和 pwr.h。待机模式实验代码在停止模式实验源码后追加。

首先看 pwr.h 头文件，因为我们用到 WKUP 引脚上升沿来唤醒待机模式的 CPU，pwr.h 头文件的 WK_UP 按键对应的宏定义我们也是用到的，WKUP 定义如下。

```
#define PWR_WKUP_GPIO_PORT              GPIOA
#define PWR_WKUP_GPIO_PIN               GPIO_PIN_0
/* PA 口时钟使能 */
#define PWR_WKUP_GPIO_CLK_ENABLE()    do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)
```

下面是 pwr.c 文件，我们主要介绍进入待机模式函数，其定义如下：

```
/**
 * @brief       进入待机模式
 * @param       无
 * @retval      无
 */
void pwr_enter_standby(void)
```

```
{
    __HAL_RCC_PWR_CLK_ENABLE();                 /* 使能电源时钟 */

    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);   /* 使能 KEY_UP 引脚的唤醒功能 */
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);          /* 需要清此标记，否则将保持唤醒状态 */
    HAL_PWR_EnterSTANDBYMode();                 /* 进入待机模式 */
}
```

该函数首先是调用__HAL_RCC_PWR_CLK_ENABLE 来使能 PWR 时钟，然后调用函数 HAL_PWR_EnableWakeUpPin 来设置 WK_UP 脚作为唤醒源。最后调用函数 HAL_PWR_EnterSTANDBYMode 进入待机模式。

最后在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t=0;
    uint8_t key = 0;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */
    pwr_wkup_key_init();                    /* 唤醒按键初始化 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "STANDBY TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Enter STANDBY MODE", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY_UP:Exit STANDBY MODE", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY0_PRES)
        {
            pwr_enter_standby();        /* 进入待机模式 */
            /* 从待机模式唤醒相当于系统重启(复位), 因此不会执行到这里 */
        }

        if ((t % 20) == 0)
        {
            LED0_TOGGLE();              /* 每 200ms,翻转一次 LED0 */
        }

        delay_ms(10);
        t++;
    }
}
```

该部分程序，经过一系列初始化后，判断到 KEY0 按下就调用 pwr_enter_standby 函数进入待机模式，然后等待按下 WK_UP 按键产生 WKUP 上升沿唤醒 CPU。注意待机模式唤醒后，系统会进行复位。

24.5.4 下载验证

下载代码后，LED0 闪烁，表明代码正在运行。按下按键 KEY0 后，TFTLCD 屏熄灭，提示进入停止模式，此时 LED0 不再闪烁，说明已经进入待机模式。按下按键 WK_UP 后，TFTLCD 屏点亮，提示退出待机模式，此时 LED0 继续闪烁，说明已经退出待机模式。

第二十五章 DMA 实验

本章，我们将介绍 STM32F407 的 DMA。我们将利用 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。

本章分为如下几个小节：

25.1 DMA 简介
25.2 硬件设计
25.3 程序设计
25.4 下载验证

25.1 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。

STM32F407 最多有 2 个 DMA 控制器(DMA1 和 DMA2)，两个 DMA 控制器总共有 16 个数据流。每个通道专门用来管理来自于一个或多个外设对存储器访问的请求。还有一个仲裁器来协调各个 DMA 请求的优先权。

STM32F407 的 DMA 有以下一些特性：

① 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。

② 仅支持 32 位访问的 AHB 从编程接口。

③ 每个 DMA 控制器有 8 个数据流，每个数据流有多达 8 个通道（请求）。

④ 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。

⑤ 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。

⑥ 8 个数据流中的每一个都连接到专用硬件 DMA 通道（请求）。

⑦ 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）。

⑧ 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）。

⑨ 可供每个数据流选择的通道请求数多达 8 个。此选择可由软件配置，允许多个外设启动 DMA 请求。

⑩ 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。

⑪ 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。

⑫ 对源和目标的增量或非增量寻址。

⑬ 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。

⑭ 每个数据流都支持循环缓冲区管理。

⑮ 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。

25.1.1 DMA 框图

STM32F407 有两个 DMA 控制器，DMA1 和 DMA2，本章，我们仅针对 DMA1 进行介绍。

下面先来学习 DMA 控制器框图，通过学习 DMA 控制器框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。STM32F407 的 DMA 控制器框图如图 25.1.1.1 所示：

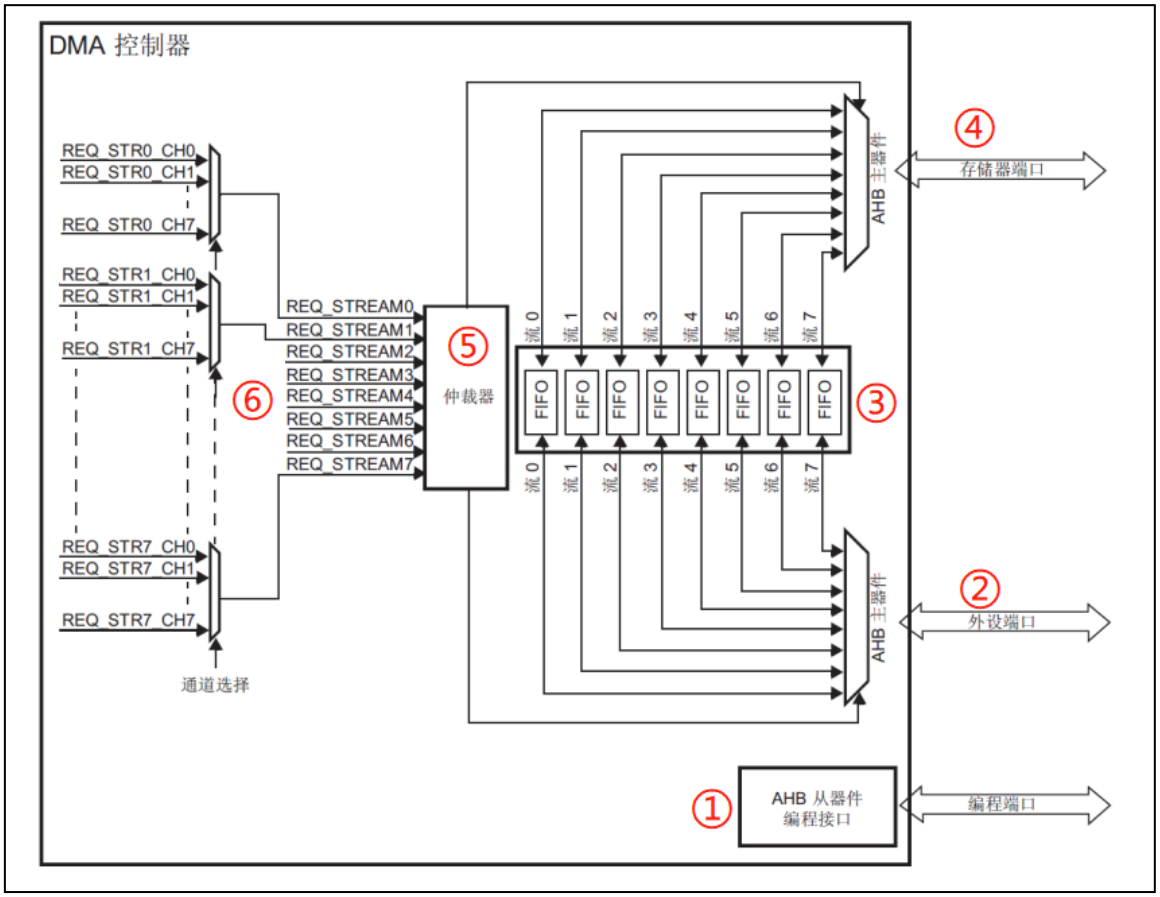


图 25.1.1.1 DMA 控制器框图

图中，我们标记了 6 处位置，起作用分别是：

① DMA 控制器的从机编程接口，通过该接口可以对 DMA 的相关控制寄存器进行设置，从而配置 DMA，实现不同的功能。

② DMA 控制器的外设接口，用于访问相关外设，特别的，当外设接口设置的访问地址是内存地址的时候，DMA 就可以工作在内存到内存模式了。

③ DMA 控制器的 FIFO 区，每个数据流（总共 8 个数据流）都有一个独立的 FIFO，可以实现存储器接口到外设接口之间的数据长度非对齐传输。

④ DMA 控制器的存储器接口，用于访问外部存储器，特别的当存储器地址是外设地址的时候，可以实现类似外设到外设的传输效果。

⑤ DMA 控制器的仲裁器，用于仲裁数据流 0~7 的请求优先级，保证数据有序传输。

⑥ DMA 控制器数据流多通道选择，通过 DMA_SxCR 寄存器控制，每个数据流有多达 8 个通道请求可以选择，如图 25.1.1.2 所示。

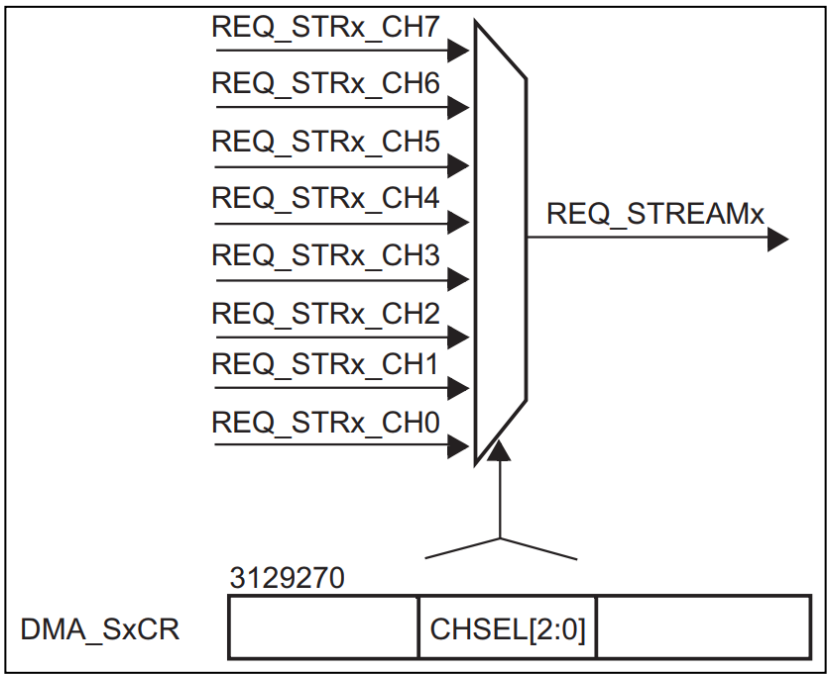


图 25.1.1.2 DMA 数据流通道选择

外设的 8 个请求独立连接到每个通道，由 DMA_SxCR 控制数据流选择哪一个通道，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。DMA1 各数据流通道映射具体见表 25.1.1.1，DMA2 各数据流通道映射具体见表 25.1.1.2。

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	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
通道 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
通道 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
通道 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
通道 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
通道 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3
通道 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
通道 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

1. 这些请求仅在 STM32F42xxx 和 STM32F43xxx 上可用。

表 25.1.1.1 DMA1 各数据流通道映射表

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道 1		DCMI	ADC2	ADC2		SPI6_TX(1)	SPI6_RX(1)	DCMI
通道 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
通道 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

1. 这些请求在 STM32F42xxx 和 STM32F43xxx 上可用。

表 25.1.1.2 DMA2 各数据流通道映射表

25.1.2 DMA 寄存器

- **DMA 中断状态寄存器（DMA_LISR 和 DMA_HISR）**

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 25.1.2.1 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 25.1.2.1 DMA_LISR 寄存器

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 寄存器各位描述和 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。

- **DMA 中断标志清除寄存器（DMA_LIFCR 和 DMA_HIFCR）**

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 25.1.2.2 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 25.1.2.2 DMA_LIFCR 寄存器

DMA_LIFCR 的各位就是用来清除 DMA_LISR 的对应位的，通过写 1 清除。在 DMA_LISR 被置位后，我们必须通过向该位寄存器对应的位写入 1 来清除。DMA_HIFCR 的使用同 DMA_LIFCR 类似，这里就不做介绍了。

第三个是 DMA 数据流 x 配置寄存器（DMA_SxCR）（x=0~7，下同）。该寄存器的我们在这里就不贴出来了，见《STM32F4xx 参考手册_V4（中文版）.pdf》第 9.5.5 小节（223 页）。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 STM32F407 外设的地址，比如我们使用串口 1，那么该寄存器必须写入 0x40011004（USART1_TDR 的地址）。如果使用其他外设，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 STM32F407 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。

25.2 硬件设计

1. 例程功能

每按下按键 KEY0，串口 1 就会以 DMA 方式发送数据，同时在 LCD 上面显示传送进度。打开串口调试助手，可以收到 DMA 发送的内容。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
3）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

3. 原理图

DMA 属于 STM32F407 内部资源，通过软件设置好就可以了。

25.3 程序设计

25.3.1 DMA 的 HAL 库驱动

DMA 在 HAL 库中的驱动代码在 stm32f4xx_hal_dma.c 文件（及其头文件）中。

1. HAL_DMA_Init 函数

DMA 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_DMA_Init(DMA_HandleTypeDef *hdma);
```

- **函数描述：**
  用于初始化 DMA1，DMA2。
- **函数形参：**
  形参 1 是 DMA_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct __DMA_HandleTypeDef
{
  void                          *Instance;  /* 寄存器基地址 */
  DMA_InitTypeDef               Init;       /* DAM 通信参数 */
  HAL_LockTypeDef               Lock;       /* DMA 锁对象 */
  __IO HAL_DMA_StateTypeDef     State;      /* DMA 传输状态 */
  void                          *Parent;    /* 父对象状态，HAL 库处理的中间变量 */
  void (*XferCpltCallback)( struct __DMA_HandleTypeDef *hdma);/*DMA 传输完成回调*/
  /* DMA 一半传输完成回调 */
  void (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  /* DMA 传输完整的 Memory1 回调 */
  void (* XferM1CpltCallback)( struct __DMA_HandleTypeDef * hdma);
  /* DMA 传输半完全内存回调 */
  void (* XferM1HalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  /*DMA 传输错误回调*/
  void (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
  /* DMA 传输中止回调 */
  void (* XferAbortCallback)( struct __DMA_HandleTypeDef * hdma);
  __IO uint32_t                 ErrorCode;              /* DMA 存取错误代码 */
```

```
  DMA_TypeDef                 *DmaBaseAddress;        /* DMA 通道基地址 */
  uint32_t                    ChannelIndex;           /* DMA 通道索引 */
}DMA_HandleTypeDef;
```

这个结构体内容比较多，上面已注释中文翻译，下面列出几个成员说明一下。

Instance：是用来设置寄存器基地址，例如要设置的对象是串口 1 的发送，那么就要参考表 25.1.1.1，串口 1 的 DMA 传输需要用到的是 DMA2 的数据流 7，即 DMA2_Stream7。

Parent：是 HAL 库处理中间变量，用来指向 DMA 通道外设句柄。

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。

其他成员变量是 HAL 库处理过程状态标识变量，这里就不做过多讲解。

接下来我们重点介绍 Init，它是 DMA_InitTypeDef 结构体类型变量，该结构体定义如下：

```
typedef struct
{
  uint32_t Channel;             /* 传输通道，例如：DMA_CHANEL_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;
```

我们通过该结构体成配置 DMA_SxCR 寄存器和 DMA_SxFCR 寄存器的相应位。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

以 DMA 的方式传输串口数据的配置步骤

1）使能 DMA 时钟。

DMA 的时钟使能是通过 AHB1ENR 寄存器来控制的，这里我们要先使能时钟，才可以配置 DMA 相关寄存器。HAL 库方法为：

```
__HAL_RCC_DMA1_CLK_ENABLE(); /* DMA1 时钟使能 */
__HAL_RCC_DMA2_CLK_ENABLE(); /* DMA2 时钟使能 */
```

2）初始化 DMA。

调用 HAL_DMA_Init 函数初始化 DMA 的相关参数，包括配置通道，外设地址，存储器地址，传输数据量等。

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(&g_uart1_handle, hdmatx, g_dma_handle);
```

其中 g_uart1_handle 是串口初始化句柄，我们在 usart.c 中定义过了。g_dma_handle 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 g_uart1_handle 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 g_uart1_handle 句柄的成员变量 hdmatx 和 DMA 句柄 g_dma_handle 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。

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 */
```

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，使用的方法是通过检测 DMA 寄存器的相关位实现：

```
__HAL_DMA_GET_FLAG(&g_dma_handle, DMA_FLAG_TCIF3_7);
```

获取当前传输剩余数据量：

```
__HAL_DMA_GET_COUNTER(&g_dma_handle);
```

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 (&g_dma_handle, 1000);
```

DMA 相关的库函数我们就讲解到这里，大家可以查看 HAL 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。HAL 库提供了通用 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 */
```

25.3.2 程序流程图

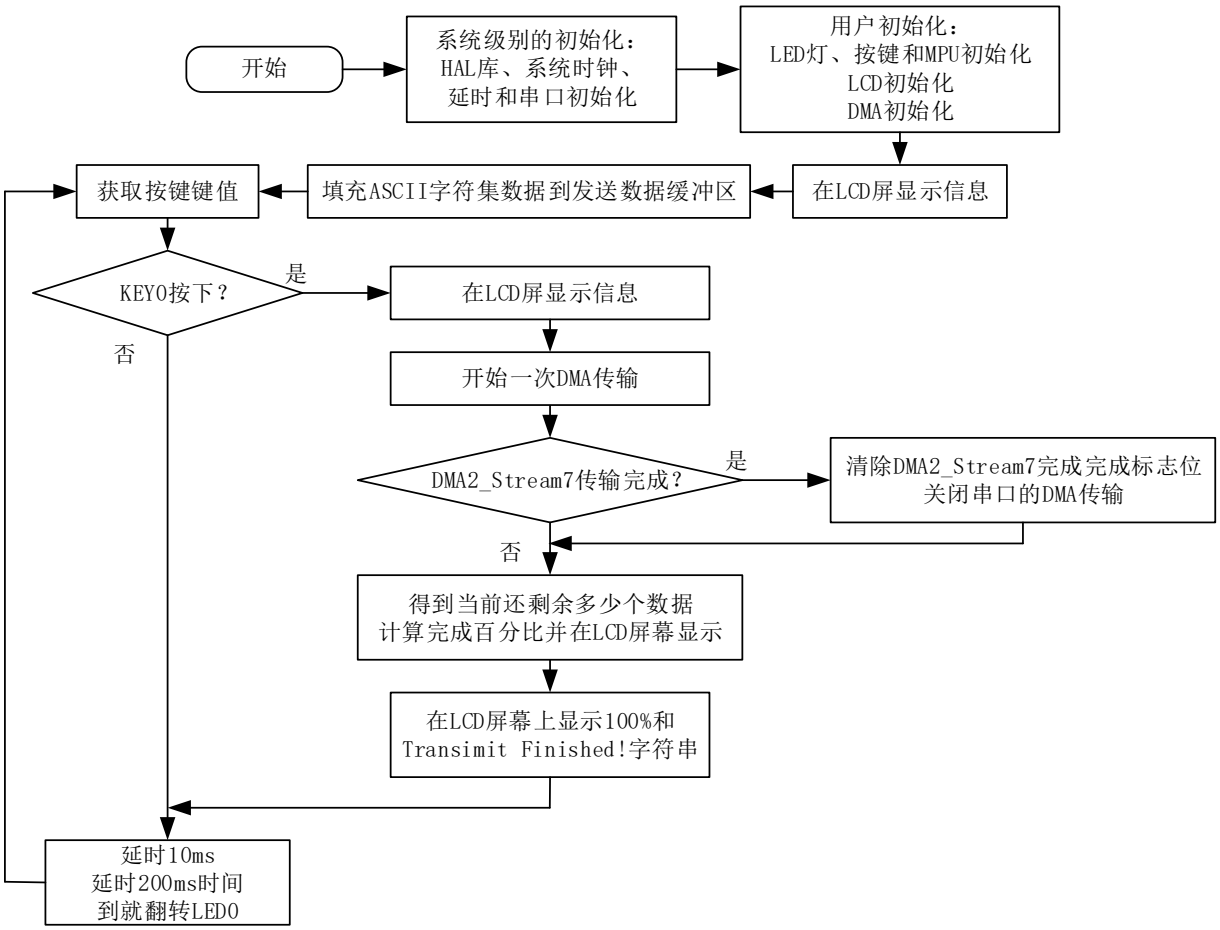


图 25.3.2.1 DMA 实验程序流程图

25.3.3 程序解析

1. DMA 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DMA 驱动源码包括两个文件：dma.c 和 dma.h。

dma.h 头文件只有函数得声明，就不解释了，我们直接介绍 dma.c 的程序。下面是与 DMA 初始化相关的函数，其定义如下：

```
/**
 * @brief       串口 TX DMA 初始化函数
 *   @note      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
 *
 * @param       dma_stream_handle : DMA 数据流,DMA1_Stream0~7/DMA2_Stream0~7
 * @retval      无
 */
void dma_init(DMA_Stream_TypeDef *dma_stream_handle, uint32_t ch)
{
    /* 得到当前 stream 是属于 DMA2 还是 DMA1 */
    if ((uint32_t)dma_stream_handle > (uint32_t)DMA2)
    {
        __HAL_RCC_DMA2_CLK_ENABLE();    /* DMA2 时钟使能 */
    }
    else
    {
```

```
        __HAL_RCC_DMA1_CLK_ENABLE();    /* DMA1 时钟使能 */
    }
    /* 将 DMA 与 USART1 联系起来(发送 DMA) */
    __HAL_LINKDMA(&g_uart1_handle,hdmatx, g_dma_handle);

    /* Tx DMA 配置 */
    g_dma_handle.Instance = dma_stream_handle;                  /* 数据流选择 */
    g_dma_handle.Init.Channel = ch;                             /* DMA 通道选择 */
    g_dma_handle.Init.Direction = DMA_MEMORY_TO_PERIPH;         /* 存储器到外设 */
    g_dma_handle.Init.PeriphInc = DMA_PINC_DISABLE;             /* 外设非增量模式 */
    g_dma_handle.Init.MemInc = DMA_MINC_ENABLE;                 /* 存储器增量模式 */
    g_dma_handle.Init.PeriphDataAlignment=DMA_PDATAALIGN_BYTE;/*外设数据长度:8 位*/
    g_dma_handle.Init.MemDataAlignment=DMA_MDATAALIGN_BYTE;  /*存储器数据长度:8 位*/
    g_dma_handle.Init.Mode = DMA_NORMAL;                        /* 外设流控模式 */
    g_dma_handle.Init.Priority = DMA_PRIORITY_MEDIUM;           /* 中等优先级 */
    g_dma_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;          /* 关闭 FIFO 模式 */
    g_dma_handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL; /* FIFO 阈值配置 */
    g_dma_handle.Init.MemBurst = DMA_MBURST_SINGLE;         /* 存储器突发单次传输 */
    g_dma_handle.Init.PeriphBurst = DMA_PBURST_SINGLE;      /* 外设突发单次传输 */

    HAL_DMA_DeInit(&g_dma_handle);
    HAL_DMA_Init(&g_dma_handle);
}
```

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。

2. main.c 代码

main.c 代码如下：

```
const uint8_t TEXT_TO_SEND[] ={"正点原子 STM32 DMA 串口实验"};/* 要循环发送的字符串 */
#define SEND_BUF_SIZE      (sizeof(TEXT_TO_SEND) + 2) * 200     /* 发送数据长度*/
uint8_t g_sendbuf[SEND_BUF_SIZE];   /* 发送数据缓冲区 */

int main(void)
{
    uint8_t  key = 0;
    uint16_t i, k;
    uint16_t len;
    uint8_t  mask = 0;
    float pro = 0;    /* 进度 */

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */

    dma_init(DMA2_Stream7, DMA_CHANNEL_4);  /* 初始化串口 1 TX DMA */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DMA TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Start", RED);

    len = sizeof(TEXT_TO_SEND);
    k = 0;
```

```
for (i = 0; i < SEND_BUF_SIZE; i++)    /* 填充 ASCII 字符集数据 */
{
    if (k >= len)    /* 入换行符 */
    {
        if (mask
        {
            g_sendbuf[i] = 0x0a;
            k = 0;
        }
        else
        {
            g_sendbuf[i] = 0x0d;
            mask++;
        }
    }
    else    /* 复制 TEXT_TO_SEND 语句 */
    {
        mask = 0;
        g_sendbuf[i] = TEXT_TO_SEND[k];
        k++;
    }
}

i = 0;

while (1)
{
    key = key_scan(0);
    if (key == KEY0_PRES)   /* KEY0 按下 */
    {
        printf("\r\nDMA DATA:\r\n");
        lcd_show_string(30, 130, 200, 16, 16, "Start Transimit....", BLUE);
        lcd_show_string(30, 150, 200, 16, 16, "   %", BLUE);    /* 显示百分号 */
        /* 开始一次 DMA 传输！ */
        HAL_UART_Transmit_DMA(&g_uart1_handle, g_sendbuf, SEND_BUF_SIZE);

        /* 等待 DMA 传输完成，此时我们来做另外一些事情，比如点灯
         * 实际应用中，传输数据期间，可以执行另外的任务 */
        while (1)
        {
            /* 等待 DMA2_Steam7 传输完成 */
            if (__HAL_DMA_GET_FLAG(&g_dma_handle, DMA_FLAG_TCIF3_7))
            {
                /* 清除 DMA2_Steam7 传输完成标志 */
                __HAL_DMA_CLEAR_FLAG(&g_dma_handle, DMA_FLAG_TCIF3_7);
                HAL_UART_DMAStop(&g_uart1_handle);  /* 传输完成以后关闭串口 DMA */
                break;
            }
            Pro =__HAL_DMA_GET_COUNTER(&g_dma_handle);/* 获取当前还剩余多少数据 */
            len = SEND_BUF_SIZE;      /* 总长度 */
            pro = 1 - (pro / len);    /* 得到百分比 */
            pro *= 100;               /* 扩大 100 倍 */
            lcd_show_num(30, 150, pro, 3, 16, BLUE);
        }
        lcd_show_num(30, 150, 100, 3, 16, BLUE);    /* 显示 100% */
        /* 提示传送完成 */
        lcd_show_string(30, 130, 200, 16, 16, "Transimit Finished!", BLUE);
    }

    i++;
    delay_ms(10);

    if (i == 20)
```

```
        {
            LED0_TOGGLE();  /* LED0 闪烁,提示系统正在运行 */
            i = 0;
        }
    }
}
```

main 函数的流程大致是：先初始化发送数据缓冲区 g_sendbuf 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(&g_dma_handle)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。

25.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 25.4.1 所示：



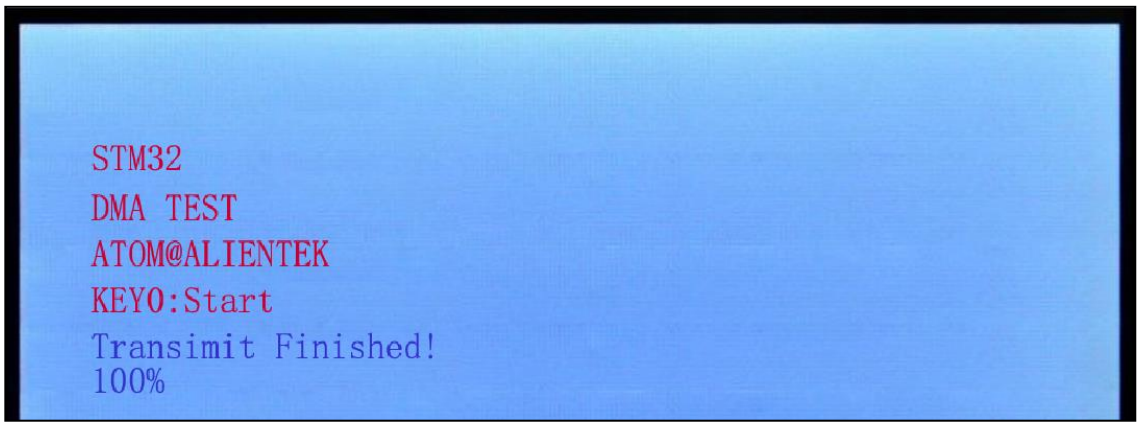
图 25.4.1 DMA 实验测试图

我们打开串口调试助手，然后按 KEY0，可以看到串口显示如图 25.4.2 所示的内容：



图 25.4.2 串口收到的数据内容

可以看到串口收到了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 25.4.3 所示：



25.4.3 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 STM32F407 的 DMA 使用。DMA 是个非常好的功能，它不但能减轻 CPU 负担，还能提高数据传输速度，合理的应用 DMA，往往能让你的程序设计变得简单。

第二十六章 IIC 实验

本章，我们将介绍如何使用 STM32F407 的普通 IO 口模拟 IIC 时序，并实现和 24C02 之间的双向通信，并把结果显示在 TFTLCD 模块上。本章分为如下几个小节：

26.1 IIC 及 24C02 介绍
26.2 硬件设计
26.3 程序设计
26.4 下载验证

26.1 IIC 及 24C02 介绍

26.1.1 IIC 简介

IIC(Inter-Integrated Circuit)总线是由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器以及其外围设备。它是由数据线 SDA 和时钟线 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。

IIC 总线有如下特点：

①总线由数据线 SDA 和时钟线 SCL 构成的串行总线，数据线用来传输数据，时钟线用来同步数据收发。

②总线上每一个器件都有一个唯一的地址识别，所以我们只需要知道器件的地址，根据时序就可以实现微控制器与器件之间的通信。

③数据线 SDA 和时钟线 SCL 都是双向线路，都通过一个电流源或上拉电阻连接到正的电压，所以当总线空闲的时候，这两条线路都是高电平。

④总线上数据的传输速率在标准模式下可达 100kbit/s 在快速模式下可达 400kbit/s 在高速模式下可达 3.4Mbit/s。

⑤总线支持设备连接。在使用 IIC 通信总线时，可以有多个具备 IIC 通信能力的设备挂载在上面，同时支持多个主机和多个从机，连接到总线的接口数量只由总线电容 400pF 的限制决定。IIC 总线挂载多个器件的示意图，如下图所示：

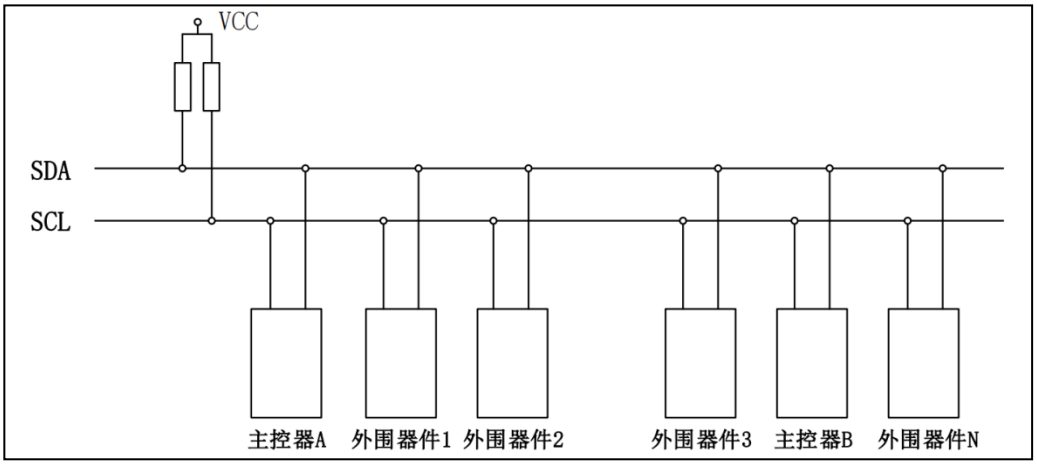


图 26.1.1.1 IIC 总线挂载多个器件

下面来学习 IIC 总线协议，IIC 总线时序图如下所示：

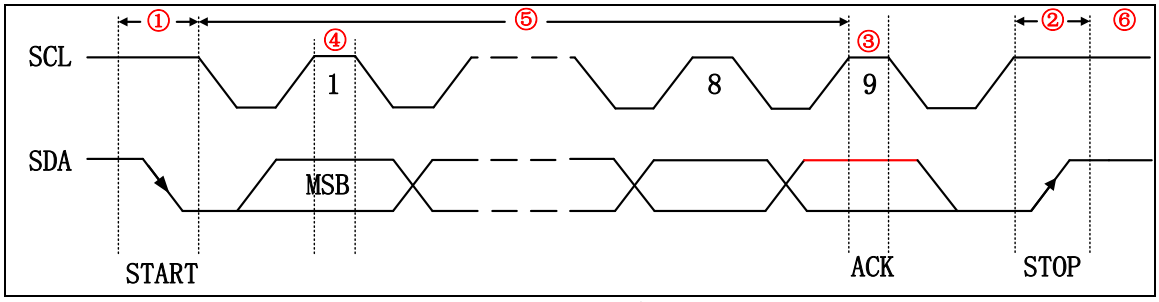


图 26.1.1.2 IIC 总线时序图

为了便于大家更好的了解 IIC 协议，我们从起始信号、停止信号、应答信号、数据有效性、数据传输以及空闲状态等 6 个方面讲解，大家需要对应图 26.1.1.2 的标号来理解。

① 起始信号

当 SCL 为高电平期间，SDA 由高到低的跳变。起始信号是一种电平跳变时序信号，而不是一个电平信号。该信号由主机发出，在起始信号产生后，总线就处于被占用状态，准备数据传输。

② 停止信号

当 SCL 为高电平期间，SDA 由低到高的跳变。停止信号也是一种电平跳变时序信号，而不是一个电平信号。该信号由主机发出，在停止信号发出后，总线就处于空闲状态。

③ 应答信号

发送器每发送一个字节，就在时钟脉冲 9 期间释放数据线，由接收器反馈一个应答信号。应答信号为低电平时，规定为有效应答位（ACK 简称应答位），表示接收器已经成功地接收了该字节；应答信号为高电平时，规定为非应答位（NACK），一般表示接收器接收该字节没有成功。

观察上图标号③就可以发现，有效应答的要求是从机在第 9 个时钟脉冲之前的低电平期间将 SDA 线拉低，并且确保在该时钟的高电平期间为稳定的低电平。如果接收器是主机，则在它收到最后一个字节后，发送一个 NACK 信号，以通知被控发送器结束数据发送，并释放 SDA 线，以便主机接收器发送一个停止信号。

④ 数据有效性

IIC 总线进行数据传送时，时钟信号为高电平期间，数据线上的数据必须保持稳定，只有在时钟线上的信号为低电平期间，数据线上的高电平或低电平状态才允许变化。数据在 SCL 的上升沿到来之前就需准备好。并在下降沿到来之前必须稳定。

⑤ 数据传输

在 I2C 总线上传送的每一位数据都有一个时钟脉冲相对应（或同步控制），即在 SCL 串行时钟的配合下，在 SDA 上逐位地串行传送每一位数据。数据位的传输是边沿触发。

⑥ 空闲状态

IIC 总线的 SDA 和 SCL 两条信号线同时处于高电平时，规定为总线的空闲状态。此时各个器件的输出级场效应管均处在截止状态，即释放总线，由两条信号线各自的上拉电阻把电平拉高。

了解前面的知识后，下面介绍一下 IIC 基本的读写通讯过程，包括主机写数据到从机即写操作，主机到从机读取数据即读操作。下面先看一下写操作通讯过程图，见图 26.1.1.3 所示：

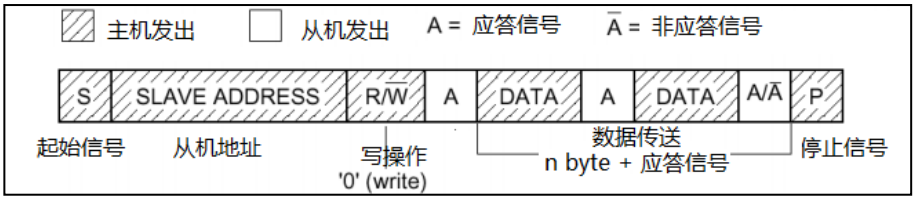


图 26.1.1.3 写操作通讯过程图

主机首先在 IIC 总线上发送起始信号，那么这时总线上的从机都会等待接收由主机发出的数据。主机接着发送从机地址+0(写操作)组成的 8bit 数据，所有从机接收到该 8bit 数据后，自行检验是否是自己的设备的地址，假如是自己的设备地址，那么从机就会发出应答信号。主机

在总线上接收到有应答信号后，才能继续向从机发送数据。注意：IIC 总线上传送的数据信号是广义的，既包括地址信号，又包括真正的数据信号。

接着讲解一下 IIC 总线的读操作过程，先看一下读操作通讯过程图，见图 26.1.1.4 所示。

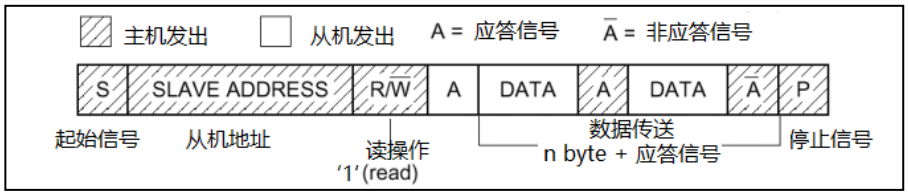


图 26.1.1.4 读操作通讯过程图

主机向从机读取数据的操作，一开始的操作与写操作有点相似，观察两个图也可以发现，都是由主机发出起始信号，接着发送从机地址+1(读操作)组成的 8bit 数据，从机接收到数据验证是否是自身的地址。那么在验证是自己的设备地址后，从机就会发出应答信号，并向主机返回 8bit 数据，发送完之后从机就会等待主机的应答信号。假如主机一直返回应答信号，那么从机可以一直发送数据，也就是图中的（n byte + 应答信号）情况，直到主机发出非应答信号，从机才会停止发送数据。

24C02 的数据传输时序是基于 IIC 总线传输时序，下面讲解一下 24C02 的数据传输时序。

26.1.2 24C02 简介

24C02 是一个 2K bit 的串行 EEPROM 存储器，内部含有 256 个字节。在 24C02 里面还有一个 8 字节的页写缓冲器。该设备的通信方式 IIC，通过其 SCL 和 SDA 与其他设备通信，芯片的引脚图如图 26.1.2.1 所示。

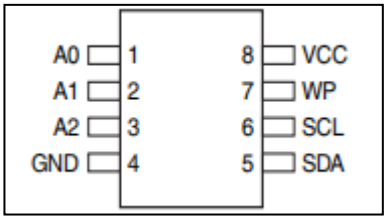


图 26.1.2.1 24C02 引脚图

上图中有一个 WP，这个是写保护引脚，接高电平只读，接地允许读和写，我们的板子设计是把该引脚接地。每一个设备都有自己的设备地址，24C02 也不例外，但是 24C02 的设备地址是包括不可编程部分和可编程部分，可编程部分是根据上图的硬件引脚 A0、A1 和 A2 所决定。设备地址最后一位用于设置数据的传输方向，即读操作/写操作，0 是写操作，1 是读操作，具体格式如下图 26.1.2.2 所示：

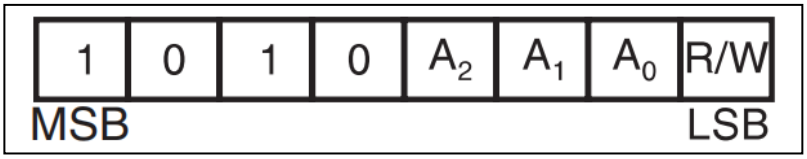


图 26.1.2.2 24C02 设备地址格式图

根据我们的板子设计，A0、A1 和 A2 均接地处理，所以 24C02 设备的读操作地址为：0xA1；写操作地址为：0xA0。

在前面已经说过 IIC 总线的基本读写操作，那么我们就可以基于 IIC 总线的时序的上，理解 24C02 的数据传输时序。

下面把实验中到的数据传输时序讲解一下，分别是对 24C02 的写时序和读时序。24C02 写时序图如图 26.1.2.3 所示。

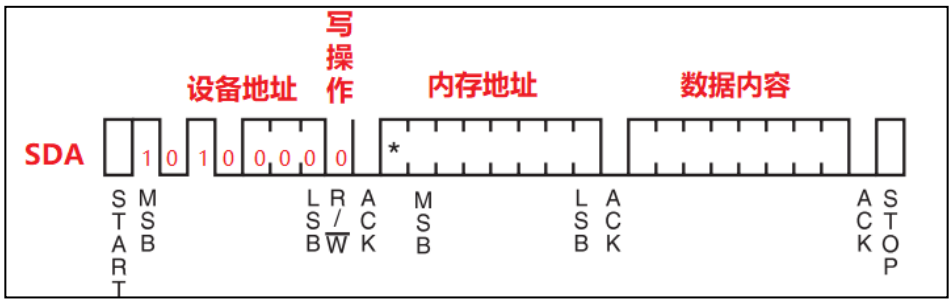


图 26.1.2.3 24C02 写时序图

上图展示的主机向 24C02 写操作时序图，主机在 IIC 总线发送第 1 个字节的数据为 24C02 的设备地址 0xA0，用于寻找总线上找到 24C02，在获得 24C02 的应答信号之后，继续发送第 2 个字节数据，该字节数据是 24C02 的内存地址，再等到 24C02 的应答信号，主机继续发送第 3 字节数据，这里的数据即是写入在第 2 字节内存地址的数据。主机完成写操作后，可以发出停止信号，终止数据传输。

上面的写操作只能单字节写入到 24C02，效率比较低，所以 24C02 有页写入时序，大大提高了写入效率，下面看一下 24C02 页写时序图，图 26.1.2.4 所示。

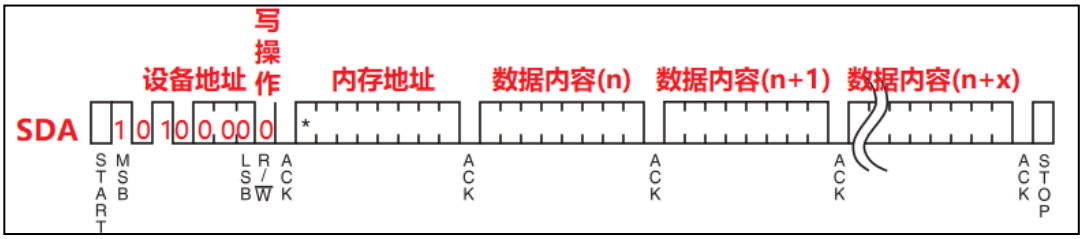


图 26.1.2.4 24C02 页写时序

在单字节写时序时，每次写入数据时都需要先写入设备的内存地址才能实现，在页写时序中，只需要告诉 24C02 第一个内存地址 1，后面数据会按照顺序写入到内存地址 2，内存地址 3 等，大大节省了通信时间，提高了时效性。因为 24C02 每次只能写 8bit 数据，所以它的页大小也就是 1 字节。页写时序的操作方式跟上面的单字节写时序差不多，所以不作过多解释了。参考以上说明去理解页写时序。

说完两种写入方式之后，下面看一下图 26.1.2.5 关于 24C02 的读时序。

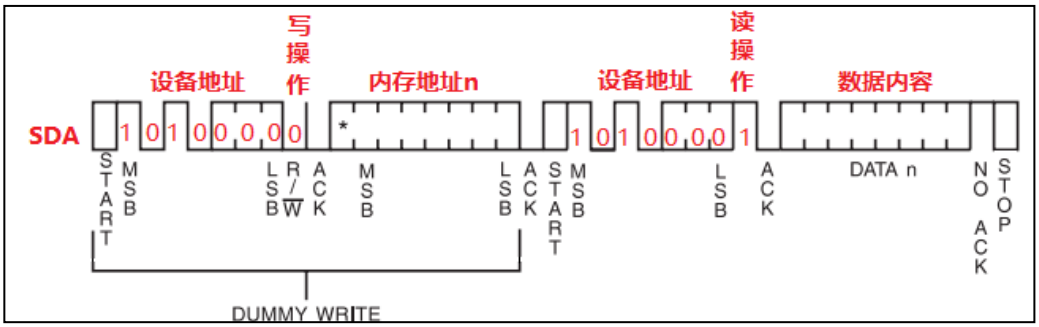


图 26.1.2.5 24C02 读时序图

24C02 读取数据的过程是一个复合的时序，其中包含写时序和读时序。先看第一个通信过程，这里是写时序，起始信号产生后，主机发送 24C02 设备地址 0xA0，获取从机应答信号后，接着发送需要读取的内存地址；在读时序中，起始信号产生后，主机发送 24C02 设备地址 0xA1，获取从机应答信号后，接着从机返回刚刚在写时序中内存地址的数据，以字节为单位传输在总线上，假如主机获取数据后返回的是应答信号，那么从机会一直传输数据，当主机发出的是非应答信号并以停止信号发出为结束，从机就会结束传输。

目前大部分 MCU 都带有 IIC 总线接口，STM32F407 也不例外。但是这里我们不使用 STM32F407 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。有兴趣的读者可以自行研究 STM32F407 的硬件 IIC 的使用。

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，基本上就得重新移植，这也是我们推荐使用软件模拟 IIC 的另外一个原因。

26.2 硬件设计

1. 例程功能

每按下 KEY1，MCU 通过 IIC 总线向 24C02 写入数据，通过按下 KEY0 来控制 24C02 读取数据。同时在 LCD 上面显示相关信息。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
KEY1 - PE3
3）EEPROM AT24C02
4）正点原子 2.8/3.5/4.3/7 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
5）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)(USMART 使用)

3. 原理图

我们主要来看看 24C02 和开发板的连接，如下图所示：

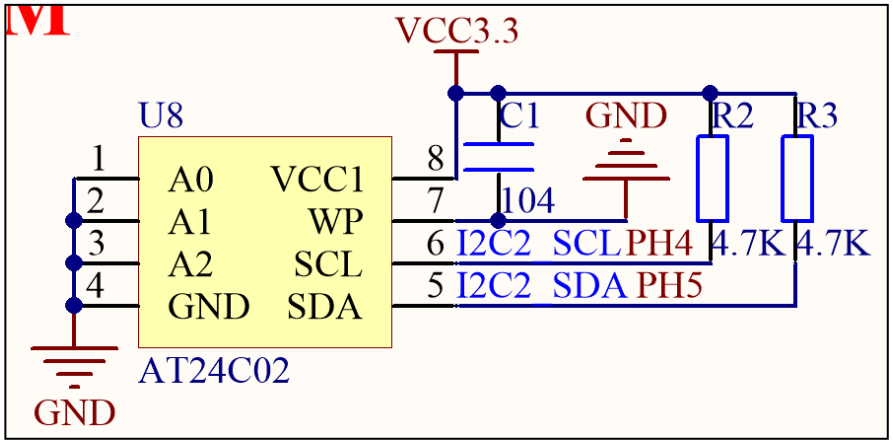


图 26.2.3.1 24C02 连接原理

24C02 的 SCL 和 SDA 分别连接在 STM32 的 PH4 和 PH5 上。本实验通过软件模拟 IIC 信号建立起与 24C02 的通信，进行数据发送与接收，使用按键 KEY0 和 KEY1 去触发，LCD 屏幕进行显示。

26.3 程序设计

IIC 实验中使用的是软件模拟 IIC，所以用到的是 HAL 中 GPIO 相关函数，前面也有介绍到，这里就不做展开了。下面介绍一下使用 IIC 传输数据的配置步骤：

使用 IIC 传输数据的配置步骤

1）使能 IIC 的 SCL 和 SDA 对应的 GPIO 时钟。

本实验中 IIC 使用的 SCL 和 SDA 分别是 PH4 和 PH5，因此需要先使能 GPIOH 的时钟，代码如下：

```
__HAL_RCC_GPIOH_CLK_ENABLE();    /* 使能 GPIOH 时钟 */
```

2）设置对应 GPIO 工作模式（SCL 推挽输出 SDA 开漏输出）

SDA 线的 GPIO 模式使用开漏输出模式（硬件已接外部上拉电阻，对于 F4 以上板子也可以用内部的上拉电阻），而 SCL 线的 GPIO 模式使用推挽输出模式，通过函数 HAL_GPIO_Init 设置实现。

3）参考 IIC 总线协议，编写信号函数（起始信号，停止信号，应答信号）

起始信号：SCL 为高电平时，SDA 由高电平向低电平跳变。

停止信号：SCL 为高电平时，SDA 由低电平向高电平跳变。

应答信号：接收到 IC 数据后，向 IC 发出特定的低电平脉冲表示已接收到数据。

4）编写 IIC 的读写函数

通过参考时序图，在一个时钟周期内发送 1bit 数据或者读取 1bit 数据。读写函数均以一字节数据进行操作。

有了读和写函数，我们就可以对外设进行驱动了。

26.3.1 程序流程图

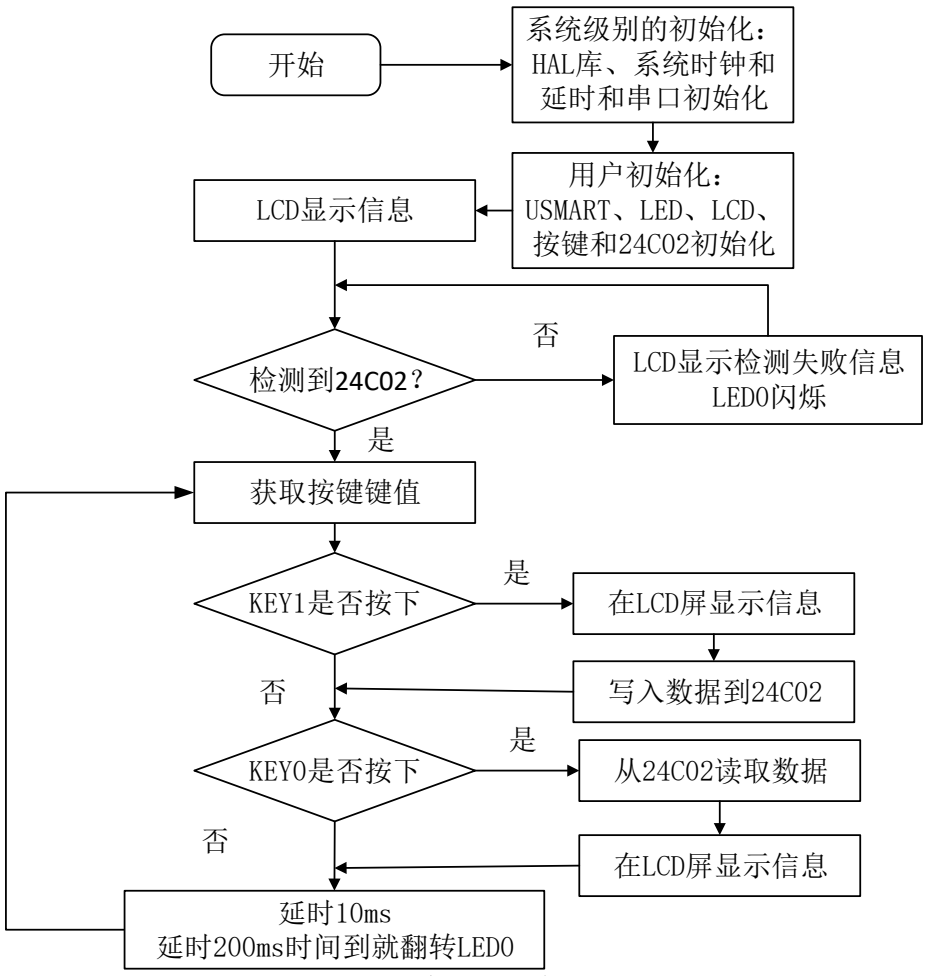


图 26.3.1.1 IIC 实验程序流程图

26.3.2 程序解析

本实验中，我们通过 GPIO 来模拟 IIC，所以不需要添加 IIC 的 HAL 库文件支持。实验工程中，我们新增了 myiic.c 存放 iic 底层驱动代码，24cxx.c 文件存放 24C02 驱动。

1. IIC 底层驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。IIC 驱动源码包括

两个文件：myiic.c 和 myiic.h。

下面我们直接介绍 IIC 相关的程序，首先先介绍 myiic.h 文件，其定义如下：

```
/* 引脚 定义 */
#define IIC_SCL_GPIO_PORT           GPIOH
#define IIC_SCL_GPIO_PIN            GPIO_PIN_4
/* PB 口时钟使能 */
#define IIC_SCL_GPIO_CLK_ENABLE()   do{ __HAL_RCC_GPIOH_CLK_ENABLE();}while(0)

#define IIC_SDA_GPIO_PORT           GPIOH
#define IIC_SDA_GPIO_PIN            GPIO_PIN_5
/* PB 口时钟使能 */
#define IIC_SDA_GPIO_CLK_ENABLE()   do{ __HAL_RCC_GPIOH_CLK_ENABLE();}while(0)
/* IO 操作 */
#define IIC_SCL(x)        do{ x ? \
        HAL_GPIO_WritePin(IIC_SCL_GPIO_PORT, IIC_SCL_GPIO_PIN, GPIO_PIN_SET):\
        HAL_GPIO_WritePin(IIC_SCL_GPIO_PORT, IIC_SCL_GPIO_PIN, GPIO_PIN_RESET);\
                        }while(0)       /* SCL */

#define IIC_SDA(x)        do{ x ? \
        HAL_GPIO_WritePin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN, GPIO_PIN_SET):\
        HAL_GPIO_WritePin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN, GPIO_PIN_RESET);\
                       }while(0)       /* SDA */
/* 读取 SDA */
#define IIC_READ_SDA HAL_GPIO_ReadPin(IIC_SDA_GPIO_PORT, IIC_SDA_GPIO_PIN)
```

我们通过宏定义标识符的方式去定义 SCL 和 SDA 两个引脚，同时通过宏定义的方式定义了 IIC_SCL() 和 IIC_SDA()设置这两个管脚可以输出 0 或者 1，主要还是通过 HAL 库的 GPIO 操作函数实现的。另外方便在 iic 操作函数中调用读取 SDA 管脚的数据，这里直接宏定义 IIC_READ_SDA 实现，在后面 iic 模拟信号实现中会频繁调用。

接下来我们看一下 myiic.c 代码中的初始化函数，代码如下：

```
/**
 * @brief       初始化 IIC
 * @param       无
 * @retval      无
 */
void iic_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    IIC_SCL_GPIO_CLK_ENABLE();  /* SCL 引脚时钟使能 */
    IIC_SDA_GPIO_CLK_ENABLE();  /* SDA 引脚时钟使能 */

    gpio_init_struct.Pin = IIC_SCL_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
    HAL_GPIO_Init(IIC_SCL_GPIO_PORT, &gpio_init_struct);    /* SCL */

    /* SDA 引脚模式设置,开漏输出,上拉, 这样就不用再设置 IO 方向了, 开漏输出的时候(=1),
       也可以读取外部信号的高低电平 */
    gpio_init_struct.Pin = IIC_SDA_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD;            /* 开漏输出 */
    HAL_GPIO_Init(IIC_SDA_GPIO_PORT, &gpio_init_struct);    /* SDA */
    iic_stop();     /* 停止总线上所有设备 */
}
```

在 iic_init 函数中主要工作就是对于 GPIO 的初始化，用于 iic 通信，不过这里需要注意的一点是 SDA 线的 GPIO 模式要使用开漏模式，SDA 和 SCL 必须要外接上拉电阻！

接下来介绍在上面已经在文字上说明过的 IIC 模拟信号：起始信号、停止信号、应答信号，下面以代码方法实现，大家可以对着图去看代码，有利于理解。

```
/**
 * @brief       IIC 延时函数,用于控制 IIC 读写速度
```

```
 * @param       无
 * @retval      无
 */
static void iic_delay(void)
{
    delay_us(2);    /* 2us 的延时，读写速度在 250Khz 以内 */
}

/**
 * @brief       产生 IIC 起始信号
 * @param       无
 * @retval      无
 */
void iic_start(void)
{
    IIC_SDA(1);
    IIC_SCL(1);
    iic_delay();
    IIC_SDA(0);     /* START 信号：当 SCL 为高时，SDA 从高变成低，表示起始信号 */
    iic_delay();
    IIC_SCL(0);     /* 钳住 I2C 总线，准备发送或接收数据 */
    iic_delay();
}

/**
 * @brief       产生 IIC 停止信号
 * @param       无
 * @retval      无
 */
void iic_stop(void)
{
    IIC_SDA(0);     /* STOP 信号：当 SCL 为高时，SDA 从低变成高，表示停止信号 */
    iic_delay();
    IIC_SCL(1);
    iic_delay();
    IIC_SDA(1);     /* 发送 I2C 总线结束信号 */
    iic_delay();
}
```

在这里首先定义一个 iic_delay 函数，目的就是控制 IIC 的读写速度，通过示波器检测读写速度在 250KHz 内，所以一秒钟传送 500Kb 数据，即一个 bit 位需要 2us，在这个延时时间内可以让器件获得一个稳定性的数据采集。

为了大家更加清晰了解代码实现的过程，下面单独把起始信号和停止信号从 iic 总线时序图中抽取出来，见图 26.2.1.1。

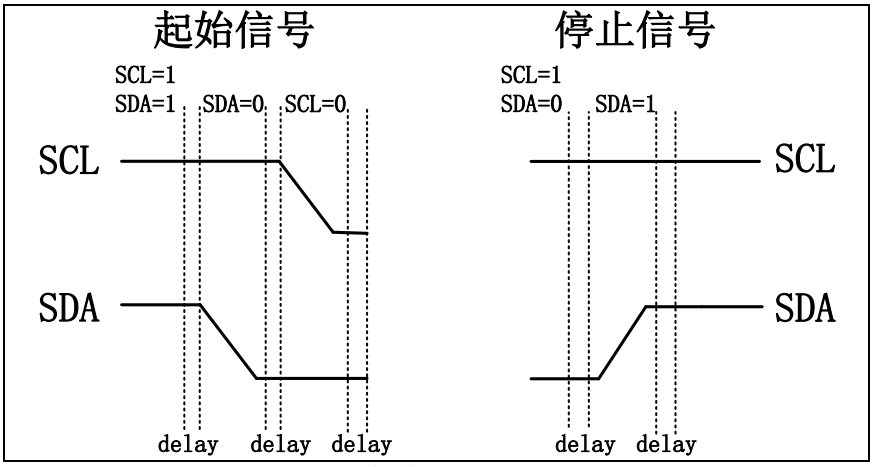


图 26.2.1.1 起始信号与停止信号图

iic_start 函数中，通过调用 myiic.h 中通过宏定义好的可以输出高低电平的 SCL 和 SDA 来

模拟 iic 总线中起始信号的发送，在 SCL 时钟线为高电平的时候，SDA 数据线从高电平状态转化到低电平状态，最后拉低时钟线，准备发送或者接收数据。

iic_stop 函数中，也是按着模拟 iic 总线中停止信号的逻辑，在 SCL 时钟线为高电平的时候，SDA 数据线从低电平状态转化到高电平状态。

接下来讲解一下 iic 的发送函数，其定义如下：

```
/**
 * @brief       IIC 发送一个字节
 * @param       data: 要发送的数据
 * @retval      无
 */
void iic_send_byte(uint8_t data)
{
    uint8_t t;

    for (t = 0; t < 8; t++)
    {
        IIC_SDA((data & 0x80) >> 7);    /* 高位先发送 */
        iic_delay();
        IIC_SCL(1);
        iic_delay();
        IIC_SCL(0);
        data <<= 1;     /* 左移 1 位,用于下一次发送 */
    }
    IIC_SDA(1);         /* 发送完成, 主机释放 SDA 线 */
}
```

在 iic 的发送函数 iic_send_byte 中，我们把需要发送的数据作为形参，形参大小为 1 个字节。在 iic 总线传输中，一个时钟信号就发送一个 bit，所以该函数需要循环八次，模拟八个时钟信号，才能把形参的 8 个位数据都发送出去。这里使用的是形参 data 和 0x80 与运算的方式，判断其最高位的逻辑值，假如为 1 即需要控制 SDA 输出高电平，否则为 0 控制 SDA 输出低电平。为了更好说明，数据发送的过程，单独拿出数据传输时序图，见图 26.2.1.2。

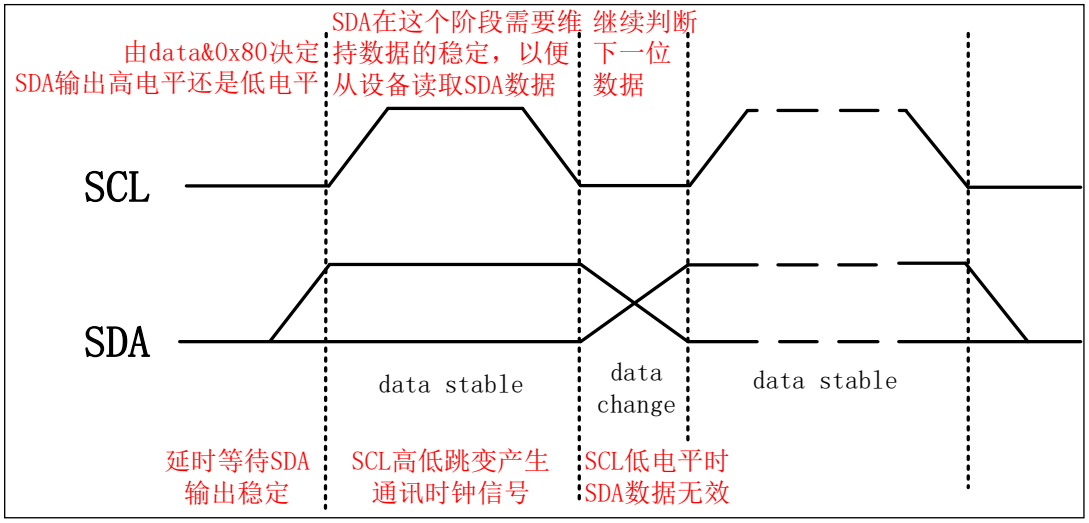


图 26.2.1.2 数据传输时序图

通过上图就可以很清楚了解数据传输时的细节，经过第一步的 SDA 高低电平的确定后，接着需要延时，确保 SDA 输出的电平稳定，在 SCL 保持高电平期间，SDA 线上的数据是有效的，此过程也是需要延时，使得从设备能够采集到有效的电平。然后准备下一位的数据，所以这里需要的是把 data 左移一位，等待下一个时钟的到来，从设备进行读取。把上述的操作重复 8 次就可以把 data 的 8 个位数据发送完毕，循环结束后，把 SDA 线拉高，等待接收从设备发送过来的应答信号。

接着讲解一下 iic 的读取函数 iic_read_byte，它的定义如下：

```
/**
 * @brief       IIC 读取一个字节
```

```
 * @param       ack:  ack=1 时，发送 ack；ack=0 时，发送 nack
 * @retval      接收到的数据
 */
uint8_t iic_read_byte(uint8_t ack)
{
    uint8_t i, receive = 0;

    for (i = 0; i < 8; i++ )    /* 接收 1 个字节数据 */
    {
        receive <<= 1;  /* 高位先输出,所以先收到的数据位要左移 */
        IIC_SCL(1);
        iic_delay();

        if (IIC_READ_SDA)
        {
            receive++;
        }

        IIC_SCL(0);
        iic_delay();
    }

    if (!ack)
    {
        iic_nack();     /* 发送 nACK */
    }
    else
    {
        iic_ack();      /* 发送 ACK */
    }

    return receive;
}
```

iic_read_byte 函数具体实现的方式跟 iic_send_byte 函数有所不同。首先可以明确的是时钟信号是通过主机发出的，而且接收到的数据大小为 1 字节，但是 IIC 传输的单位是 bit，所以就需要执行 8 次循环，才能把一字节数据接收完整。

具体实现过程：首先需要一个变量 receive 存放接收到的数据，在每一次循环开始前都需要对 receive 进行左移 1 位操作，那么 receive 的 bit0 位每一次赋值前都是空的，用来存放最新接收到的数据位，然后在 SCL 线进行高低电平切换时输出 IIC 时钟，在 SCL 高电平期间加入延时，确保有足够的时间能让数据发送并进行处理，使用宏定义 IIC_READ_SDA 就可以判断读取到的高低电平，假如 SDA 为高电平，那么 receive++即在 bit0 置 1，否则不做处理即保持原来的 0 状态。当 SCL 线拉低后，需要加入延时，便于从机切换 SDA 线输出数据。在 8 次循环结束后，我们就获得了 8bit 数据，把它作为返回值返回，然而按照时序图，作为主机就需要发送应答或者非应答信号，去回复从机。

上面提及到应答信号和非应答信号是在读时序中发生的，此外在写时序中也存在有一个信号响应，当发送完 8bit 数据后，这里是一个等待从机应答信号的操作，这里我们也定义了，下面看一下它们的定义：

```
/**
 * @brief       等待应答信号到来
 * @param       无
 * @retval      1，接收应答失败
 *              0，接收应答成功
 */
uint8_t iic_wait_ack(void)
{
    uint8_t waittime = 0;
    uint8_t rack = 0;

    IIC_SDA(1);     /* 主机释放 SDA 线(此时外部器件可以拉低 SDA 线) */
    iic_delay();
```

```
    IIC_SCL(1);     /* SCL=1, 此时从机可以返回 ACK */
    iic_delay();

    while (IIC_READ_SDA)    /* 等待应答 */
    {
        waittime++;

        if (waittime > 250)
        {
            iic_stop();
            rack = 1;
            break;
        }
    }

    IIC_SCL(0);     /* SCL=0, 结束 ACK 检查 */
    iic_delay();
    return rack;
}

/**
 * @brief       产生 ACK 应答
 * @param       无
 * @retval      无
 */
void iic_ack(void)
{
    IIC_SDA(0);     /* SCL 0 -> 1  时 SDA = 0,表示应答 */
    iic_delay();
    IIC_SCL(1);     /* 产生一个时钟 */
    iic_delay();
    IIC_SCL(0);
    iic_delay();
    IIC_SDA(1);     /* 主机释放 SDA 线 */
    iic_delay();
}

/**
 * @brief       不产生 ACK 应答
 * @param       无
 * @retval      无
 */
void iic_nack(void)
{
    IIC_SDA(1);     /* SCL 0 -> 1  时 SDA = 1,表示不应答 */
    iic_delay();
    IIC_SCL(1);     /* 产生一个时钟 */
    iic_delay();
    IIC_SCL(0);
    iic_delay();
}
```

首先先讲解一下 iic_wait_ack 函数，该函数主要用在写时序中，当启动起始信号，发送完 8bit 数据到从机时，我们就需要等待以及处理接收从机发送过来的响应信号或者非响应信号，一般就是在 iic_send_byte 函数后面调用。

具体实现：首先先释放 SDA，把电平拉高，延时等待从机操作 SDA 线，然后主机拉高时钟线并延时，确保有充足的时间让主机接收到从机发出的 SDA 信号，这里使用的是 IIC_READ_SDA 宏定义去读取，根据 IIC 协议，主机读取 SDA 的值为低电平，就表示“应答信号”；读到 SDA 的值为高电平，就表示“非应答信号”。在这个等待读取的过程中加入了超时判断，假如超过这个时间没有接收到数据，那么主机直接发出停止信号，跳出循环，返回等于 1 的变量。在正常等待到应答信号后，主机会把 SCL 时钟线拉低并延时，返回是否接收到应答信号。

当主机作为接收端时，调用 iic_read_byte 函数之后，按照 iic 通信协议，需要给从机返回应答或者是非应答信号，这里就是用到了 iic_ack 和 iic_nack 函数。

具体实现：从上面的说明已经知道了 SDA 为低电平即应答信号，高电平即非应答信号，那么还是老规矩，首先先根据返回“应答”或者“非应答”两种情况拉低或者拉高 SDA，并延时等待 SDA 电平稳定，然后主机拉高 SCL 线并延时，确保从机能有足够时间去接收 SDA 线上的电平信号。然后主机拉低时钟线并延时，完成这一位数据的传送。最后把 SDA 拉高，呈高阻态，方便后续通信用到。

2. 24C02 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。24CXX 驱动源码包括两个文件：24cxx.c 和 24cxx.h。

在上一小节已经对 IIC 协议中的需要用到的信号都用函数封装好了，那么现在就要定义符合 24C02 时序的函数。为了使代码功能更加健全，所以在 24cxx.h 中宏定义了不同容量大小的 24C 系列型号，具体定义如下：

```
#define AT24C01     127
#define AT24C02     255
#define AT24C04     511
#define AT24C08     1023
#define AT24C16     2047
#define AT24C32     4095
#define AT24C64     8191
#define AT24C128    16383
#define AT24C256    32767

/* 开发板使用的是 24c02，所以定义 EE_TYPE 为 AT24C02 */
#define EE_TYPE     AT24C02
```

在 24cxx.c 文件中，读/写操作函数对于不同容量大小的 24Cxx 芯片都有相对应的代码块解决处理。下面先看一下 at24cxx_write_one_byte 函数，实现在 AT24Cxx 芯片指定地址写入一个数据，代码如下：

```
/**
 * @brief       在 AT24CXX 指定地址写入一个数据
 * @param       addr: 写入数据的目的地址
 * @param       data: 要写入的数据
 * @retval      无
 */
void at24cxx_write_one_byte(uint16_t addr, uint8_t data)
{
    /* 原理说明见:at24cxx_read_one_byte 函数, 本函数完全类似 */
    iic_start();                        /* 发送起始信号 */


    if (EE_TYPE > AT24C16)              /* 24C16 以上的型号, 分 2 个字节发送地址 */
    {
        iic_send_byte(0xA0);            /* 发送写命令, IIC 规定最低位是 0, 表示写入 */
        iic_wait_ack();                 /* 每次发送完一个字节,都要等待 ACK */
        iic_send_byte(addr >> 8);       /* 发送高字节地址 */
    }
    else
    {   /* 发送器件 0XA0 + 高位 a8/a9/a10 地址,写数据 */
        iic_send_byte(0xA0 + ((addr >> 8) << 1));
    }

    iic_wait_ack();                 /* 每次发送完一个字节,都要等待 ACK */
    iic_send_byte(addr % 256);      /* 发送低位地址 */
    iic_wait_ack();                 /* 等待 ACK, 此时地址发送完成了 */

    /* 因为写数据的时候,不需要进入接收模式了,所以这里不用重新发送起始信号了 */
    iic_send_byte(data);            /* 发送 1 字节 */
```

```
    iic_wait_ack();                 /* 等待 ACK */
    iic_stop();                     /* 产生一个停止条件 */
    delay_ms(10);               /* 注意: EEPROM 写入比较慢,必须等到 10ms 后再写下一个字节 */
}
```

该函数的操作流程跟前面已经分析过的 24C02 单字节写时序一样，首先调用 iic_start 函数产生起始信号，然后调用 iic_send_byte 函数发送第 1 个字节数据设备地址，等待 24Cxx 设备返回应答信号；收到应答信号后，继续发送第 2 个 1 字节数据内存地址 addr；等待接收应答后，最后发送第 3 个字节数据写入内存地址的数据 data，24Cxx 设备接收完数据，返回应答信号，主机调用 iic_stop 函数产生停止信号终止数据传输，最终需要延时 10ms，等待 eeprom 写入完毕。

我们的函数兼容 24Cxx 系列多种容量，就在发送设备地址处做了处理，这里说一下为什么需要这样子设计。大家请看一下 24Cxx 芯片内存组织表，见表 26.2.2.1 所示。

芯片	页数	每页字节数	总的字节数	字节寻址地址线数量
AT24C01A	16	8	128	7
AT24C02	32	8	256	8
AT24C04	32	16	512	9
AT24C08A	64	16	1024	10
AT24C16A	128	16	2048	11
AT24C32	128	32	4096	12
AT24C64A	256	32	8192	13

表 26.2.2.1 24Cxx 芯片内存组织表

主机发送的设备地址和内存地址共同确定了要写入的地方，这里分析一下 24C16 的使用的是 iic_send_byte(0xA0+((addr>>8)<<1))和 iic_send_byte(addr % 256)确定写入位置，由于它内存大小一共 2048 字节，所以只需要定义 11 个寻址地址线，2048 = 2^11。主机下发读写命令的时候带了 3 位，后面再跟 1 个字节（8 位）的地址，正好 11 位，就不需要再发后续的地址字节了。

而容量大于 24C16 的芯片，需要单独发送 2 个字节（甚至更多）的地址，如 24C32，它的大小为 4096，需要 12 个寻址地址线支持，4096 = 2^12。24C16 是 2 个字节刚刚好，而它需要三个字节才能确定写入的位置。24C32 芯片规定设备写地址 0xA0/读地址 0xA1，后面接着发送 8 位高地址，最后才发送 8 位低地址。与函数里面的操作是一致。

接下来看一下 at24cxx_read_one_byte 函数，其定义如下：

```
/**
 * @brief       在 AT24CXX 指定地址读出一个数据
 * @param       readaddr: 开始读数的地址
 * @retval      读到的数据
 */
uint8_t at24cxx_read_one_byte(uint16_t addr)
{
    uint8_t temp = 0;
    iic_start();                    /* 发送起始信号 */

    /* 根据不同的 24CXX 型号, 发送高位地址
     * 1, 24C16 以上的型号, 分 2 个字节发送地址
     * 2, 24C16 及以下的型号, 分 1 个低字节地址 + 占用器件地址的 bit1~bit3 位 用于表示高位地
          址, 最多 11 位地址
     *    对于 24C01/02, 其器件地址格式(8bit)为: 1  0  1  0  A2  A1  A0  R/W
     *    对于 24C04,    其器件地址格式(8bit)为: 1  0  1  0  A2  A1  a8  R/W
     *    对于 24C08,    其器件地址格式(8bit)为: 1  0  1  0  A2  a9  a8  R/W
     *    对于 24C16,    其器件地址格式(8bit)为: 1  0  1  0  a10 a9  a8  R/W
     *    R/W      : 读/写控制位 0,表示写; 1,表示读;
     *    A0/A1/A2 : 对应器件的 1,2,3 引脚(只有 24C01/02/04/8 有这些脚)
     *    a8/a9/a10: 对应存储整列的高位地址, 11bit 地址最多可以表示 2048 个位置,可以寻址
          24C16 及以内的型号
     */
    if (EE_TYPE > AT24C16)          /* 24C16 以上的型号, 分 2 个字节发送地址 */
    {
        iic_send_byte(0xA0);        /* 发送写命令, IIC 规定最低位是 0, 表示写入 */
```

```
        iic_wait_ack();                 /* 每次发送完一个字节,都要等待 ACK */
        iic_send_byte(addr >> 8);       /* 发送高字节地址 */
    }
    else
    {  /* 发送器件 0XA0 + 高位 a8/a9/a10 地址,写数据 */
        iic_send_byte(0xA0 + ((addr >> 8) << 1));
    }

    iic_wait_ack();                     /* 每次发送完一个字节,都要等待 ACK */
    iic_send_byte(addr % 256);          /* 发送低位地址 */
    iic_wait_ack();                     /* 等待 ACK, 此时地址发送完成了 */

    iic_start();                        /* 重新发送起始信号 */
    iic_send_byte(0xA1);                /* 进入接收模式, IIC 规定最低位是 1, 表示读取 */
    iic_wait_ack();                     /* 每次发送完一个字节,都要等待 ACK */
    temp = iic_read_byte(0);            /* 接收一个字节数据 */
    iic_stop();                         /* 产生一个停止条件 */
    return temp;
}
```

这里的函数的实现跟前面第 26.1.4 小节 24C02 数据传输中的读时序一致，主机首先调用 iic_start 函数产生起始信号，然后调用 iic_send_byte 函数发送第 1 个字节数据设备写地址，使用 iic_wait_ack 函数等待 24Cxx 设备返回应答信号；收到应答信号后，继续发送第 2 个 1 字节数据内存地址 addr；等待接收应答后，重新调用 iic_start 函数产生起始信号，这一次的设备方向改变了，调用 iic_send_byte 函数发送设备读地址，然后使用 iic_wait_ack 函数去等待设备返回应答信号，同时使用 iic_read_byte 去读取从从机发出来的数据。由于 iic_read_byte 函数的形参是 0，所以在获取完 1 个字节的数据后，主机发送非应答信号，停止数据传输，最终调用 iic_stop 函数产生停止信号，返回从从机 addr 中读取到的数据。

为了方便检测 24Cxx 芯片是否正常工作，在这里也定义了一个检测函数，代码如下：

```
/**
 * @brief       检查 AT24CXX 是否正常
 *   @note      检测原理：在器件的末地址写如 0X55, 然后再读取, 如果读取值为 0X55
 *              则表示检测正常. 否则,则表示检测失败.
 * @param       无
 * @retval      检测结果
 *              0: 检测成功
 *              1: 检测失败
 */
uint8_t at24cxx_check(void)
{
    uint8_t temp;
    uint16_t addr = EE_TYPE;
    temp = at24cxx_read_one_byte(addr);     /* 避免每次开机都写 AT24CXX */

    if (temp == 0x55)                       /* 读取数据正常 */
    {
        return 0;
    }
    else                                    /* 排除第一次初始化的情况 */
    {
        at24cxx_write_one_byte(addr, 0x55); /* 先写入数据 */
        temp = at24cxx_read_one_byte(255);  /* 再读取数据 */
        if (temp == 0X55)return 0;
    }
    return 1;
}
```

学到这个地方相信大家，对于这个操作并不陌生了，在前面的 RTC 实验也有相似的操作，可以翻回去看看。这里利用的是 EEPROM 芯片掉电不丢失的特性，在第一次写入了某个值之后，再去读一下是否写入成功，这种方式去检测芯片是否正常工作。

此外方便多字节写入和读取，还定义了在指定地址读取指定个数的函数以及在指令地址写

入指定个数的函数，代码如下：

```
/**
 * @brief       在 AT24CXX 里面的指定地址开始读出指定个数的数据
 * @param       addr    : 开始读出的地址 对 24c02 为 0~255
 * @param       pbuf    : 数据数组首地址
 * @param       datalen : 要读出数据的个数
 * @retval      无
 */
void at24cxx_read(uint16_t addr, uint8_t *pbuf, uint16_t datalen)
{
    while (datalen--)
    {
        *pbuf++ = at24cxx_read_one_byte(addr++);
    }
}

/**
 * @brief       在 AT24CXX 里面的指定地址开始写入指定个数的数据
 * @param       addr    : 开始写入的地址 对 24c02 为 0~255
 * @param       pbuf    : 数据数组首地址
 * @param       datalen : 要写入数据的个数
 * @retval      无
 */
void at24cxx_write(uint16_t addr, uint8_t *pbuf, uint16_t datalen)
{
    while (datalen--)
    {
        at24cxx_write_one_byte(addr, *pbuf);
        addr++;
        pbuf++;
    }
}
```

对于这两个函数都是调用前面的单字节操作函数去实现的，利用 for 循环，连续调用单字节操作函数去实现，这里就不多讲。

3. main.c 代码

在 main.c 里面编写如下代码：

```
const uint8_t g_text_buf[] = {"STM32 IIC TEST"};   /* 要写入到 24c02 的字符串数组 */
#define TEXT_SIZE       sizeof(g_text_buf)          /* TEXT 字符串长度 */

int main(void)
{
    uint8_t key;
    uint16_t i = 0;
    uint8_t datatemp[TEXT_SIZE];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    usmart_dev.init(168);                       /* 初始化 USMART */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    at24cxx_init();                             /* 初始化 24CXX */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "IIC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY1:Write  KEY0:Read", RED);
```

```
    while (at24cxx_check()) /* 检测不到 24c02 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "24C02 Check Failed!", RED);
        delay_ms(500);
        lcd_show_string(30, 130, 200, 16, 16, "Please Check!      ", RED);
        delay_ms(500);
        LED0_TOGGLE();       /* 红灯闪烁 */
    }

    lcd_show_string(30, 130, 200, 16, 16, "24C02 Ready!", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY1_PRES)   /* KEY1 按下,写入 24C02 */
        {
            lcd_fill(0, 150, 239, 319, WHITE);  /* 清除半屏 */
            lcd_show_string(30, 150, 200, 16, 16, "Start Write 24C02....", BLUE);
            at24cxx_write(0, (uint8_t *)g_text_buf, TEXT_SIZE);
            /* 提示传送完成 */
            lcd_show_string(30, 150, 200, 16, 16, "24C02 Write Finished!", BLUE);
        }

        if (key == KEY0_PRES)   /* KEY0 按下,读取字符串并显示 */
        {
            lcd_show_string(30, 150, 200, 16, 16, "Start Read 24C02.... ", BLUE);
            at24cxx_read(0, datatemp, TEXT_SIZE);
            /* 提示传送完成 */
            lcd_show_string(30, 150, 200, 16, 16, "The Data Readed Is:  ", BLUE);
            /* 显示读到的字符串 */
            lcd_show_string(30, 170, 200, 16, 16, (char *)datatemp, BLUE);
        }

        i++;

        if (i == 20)
        {
            LED0_TOGGLE();  /* 红灯闪烁 */
            i = 0;
        }

        delay_ms(10);
    }
}
```

main 函数的流程大致是：在 main 函数外部定义要写入 24C02 的字符串数组 g_text_buf。在完成系统级和用户级初始化工作后，检测 24c02 是否存在，然后通过 KEY0 去读取 0 地址存放得数据并把数据显示在 LCD 上；另外还可以通过 KEY1 去 0 地址处写入 g_text_buf 数据并在 LCD 界面中显示传输中，完成后并显示“24C02 Write Finished!”。

26.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了，先按下 KEY1 写入数据，然后再按 KEY0 读取数据，最终 LCD 显示的内容如图 26.4.1 所示：

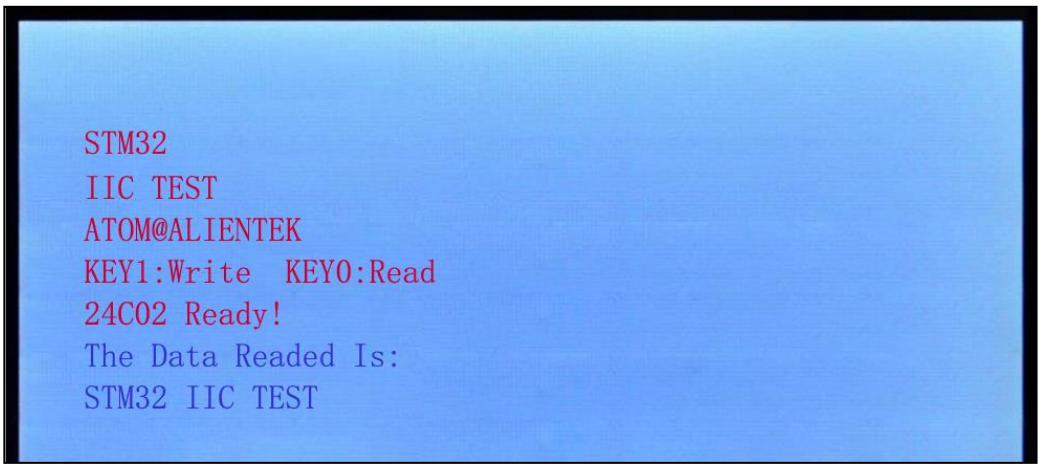


图 26.4.1 IIC 实验程序运行效果图

假如大家需要验证 24C02 的自检函数，可以用跟杜邦线把 PH4 和 PH5 短接，重新上电看看是否能看到报错。

该实验还支持 USMART，在这里我们可以方便测试 24C02 的读写功能，可以操作 24C02 的任意地址，不过在 0~255 这个范围，读写测试图如图 26.4.2 所示。

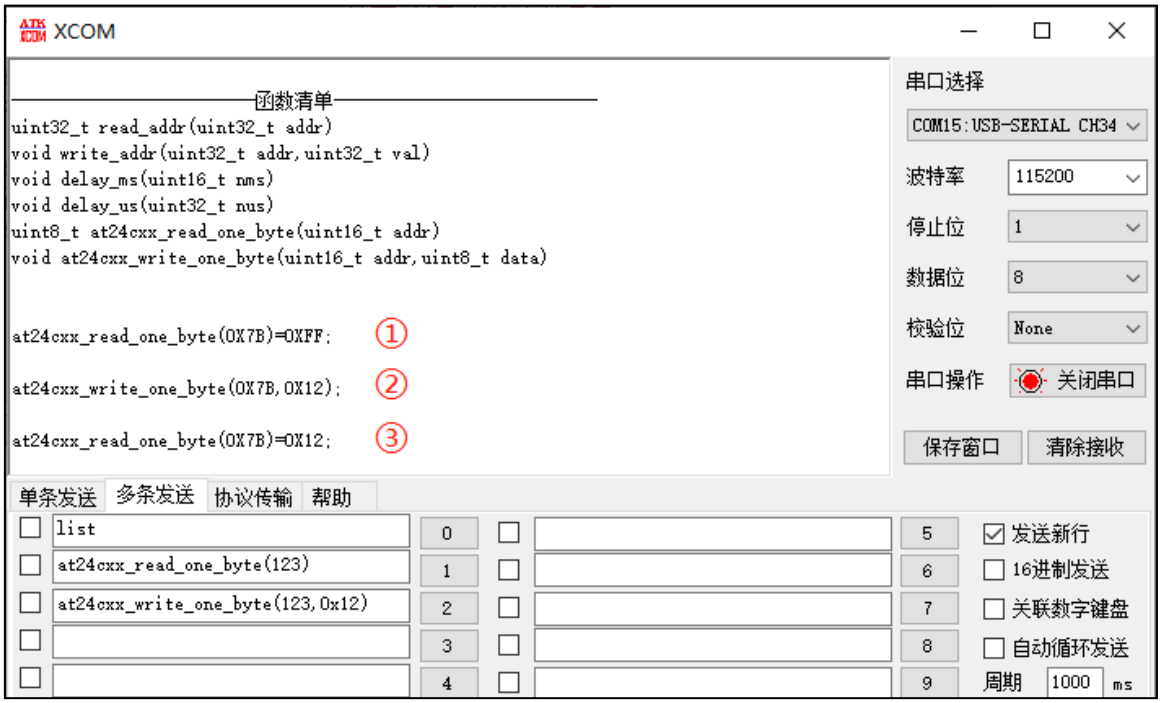


图 26.4.2 24C02 读写测试图

图中，我们首先调用 at24cxx_read_one_byte 函数在 123 地址处读取，获取的数据为 0xFF。通过调用 at24cxx_write_one_byte 函数在 123 地址处写入 0x12 的值，然后继续调用 at24cxx_read_one_byte 函数在 123 地址处读取，获得 0x12 的值，表明实验成功。

至此，我们整个 IIC 实验就结束了，本章内容比较多，需要大家花多点时间去理解，一定要自己去用一下 IIC 通信协议。市面上很多器件都是具有 IIC 通信接口的，可以尝试去驱动它们，这样才能学以致用。

第二十七章 SPI 实验

本章，我们将介绍如何使用 STM32F407 的 SPI 功能，并实现对外部 FLASH 的读写并把结果显示在 TFTLCD 模块上。本章分为如下几个小节：

27.1 SPI 及 NOR Flash 芯片介绍
27.2 硬件设计
27.3 程序设计
27.4 下载验证

27.1 SPI 及 NOR Flash 介绍

27.1.1 SPI 介绍

我们将从结构、时序和寄存器三个部分来介绍 SPI。

27.1.1.1 SPI 框图

SPI 是英语 Serial Peripheral interface 缩写，顾名思义就是串行外围设备接口。SPI 通信协议是 Motorola 公司首先在其 MC68HCXX 系列处理器上定义的。SPI 接口是一种高速的全双工同步的通信总线，已经广泛应用在众多 MCU、存储芯片、AD 转换器和 LCD 之间。大部分 STM32 是有 3 个 SPI 接口，本实验使用的是 SPI2。

我们先看 SPI 的结构框图，了解它的大致功能，如图 27.1.1.1.1 所示。

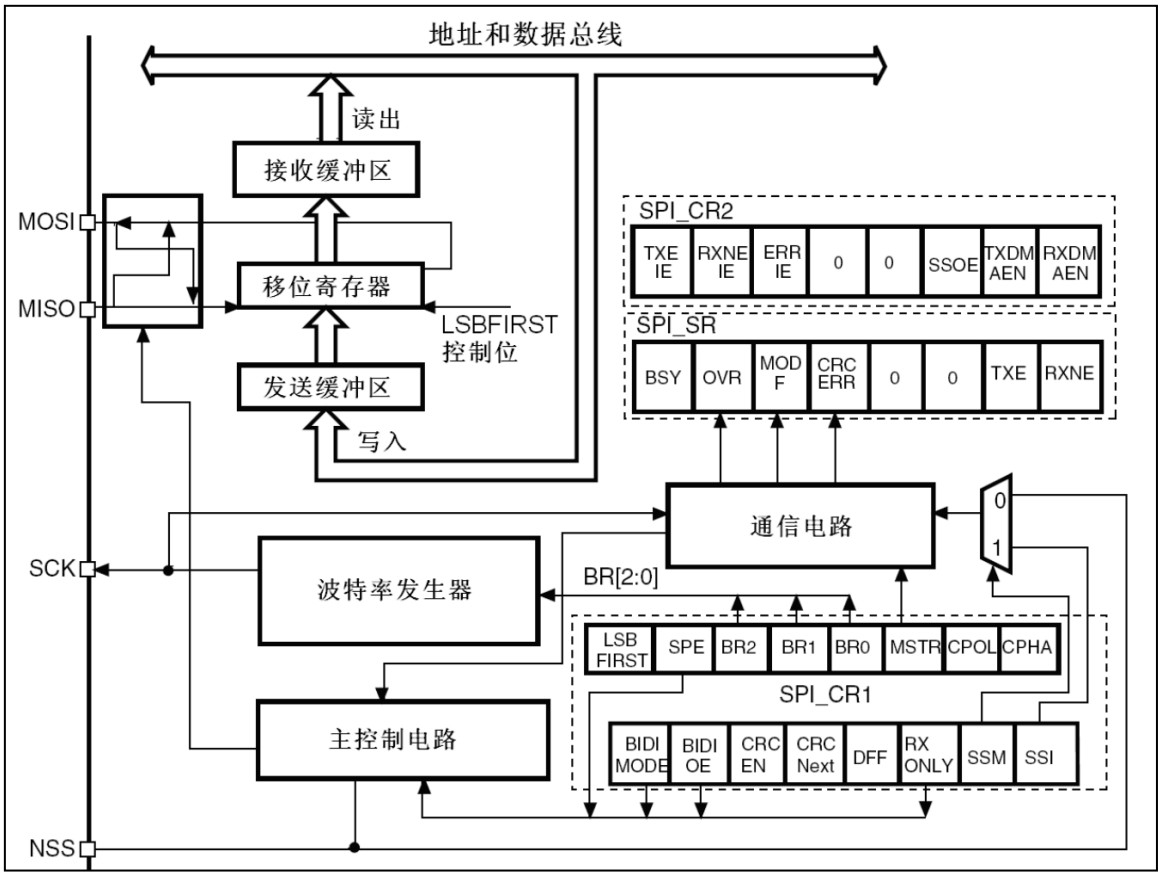


图 27.1.1.1.1 SPI 框图

围绕框图，我们展开介绍一下 SPI 的引脚信息、工作原理以及传输方式，把 SPI 的 4 种工作方式放在后面讲解。

SPI 的引脚信息：

MISO（Master In / Slave Out）主设备数据输入，从设备数据输出。

MOSI（Master Out / Slave In）主设备数据输出，从设备数据输入。

SCLK（Serial Clock）时钟信号，由主设备产生。

CS（Chip Select）从设备片选信号，由主设备产生。

SPI 的工作原理：在主机和从机都有一个串行移位寄存器，主机通过向它的 SPI 串行寄存器写入一个字节来发起一次传输。串行移位寄存器通过 MOSI 信号线将字节传送给从机，从机也将自己的串行移位寄存器中的内容通过 MISO 信号线返回给主机。这样，两个移位寄存器中的内容就被交换。外设的写操作和读操作是同步完成的。如果只是进行写操作，主机只需忽略接收到的字节。反之，若主机要读取从机的一个字节，就必须发送一个空字节引发从机传输。

SPI 的传输方式：SPI 总线具有三种传输方式：全双工、单工以及半双工传输方式，见表 27.1.1.1.1。

全双工通信	在任何时刻，主机与从机之间都可以同时进行数据的发送和接收
单工通信	在同一时刻，只有一个传输的方向，发送或者是接收
半双工通信	在同一时刻，只能为一个方向传输数据

表 27.1.1.1.1 SPI 传输方式

27.1.1.2 SPI 工作模式

STM32 要与具有 SPI 接口的器件进行通信，就必须遵循 SPI 的通信协议。每一种通信协议都有各自的读写数据时序，当然 SPI 也不例外。SPI 通信协议就具备 4 种工作模式，在讲这 4 种工作模式前，首先先知道两个单词 CPOL 和 CPHA。

CPOL，详称 Clock Polarity，就是时钟极性，当主从机没有数据传输的时候（空闲状态）SCL 线的电平状态。假如空闲状态是高电平，CPOL=1；若空闲状态时低电平，那么 CPOL＝0。

CPHA，详称 Clock Phase，就是时钟相位。在这里先科普一下数据传输的常识：同步通信时，数据的变化和采样都是在时钟边沿上进行的，每一个时钟周期都会有上升沿和下降沿两个边沿，那么数据的变化和采样就分别安排在两个不同的边沿，由于数据在产生和到它稳定是需要一定的时间，那么假如我们在第 1 个边沿信号把数据输出了，从机只能从第 2 个边沿信号去采样这个数据。

CPHA 实质指的是数据的采样时刻，CPHA＝0 的情况就表示数据的采样是从第 1 个边沿信号上即奇数边沿，具体是上升沿还是下降沿的问题，是由 CPOL 决定的。这里就存在一个问题：当开始传输第一个 bit 的时候，第 1 个时钟边沿就采集该数据了，那数据是什么时候输出来的呢？那么就有两种情况：一是 CS 使能的边沿，二是上一帧数据的最后一个时钟沿。

CPHA = 1 的情况就是表示数据采样是从第 2 个边沿即偶数边沿，它的边沿极性要注意一点，不是和上面 CPHA=0 一样的边沿情况。前面的是奇数边沿采样数据，从 SCL 空闲状态的直接跳变，空闲状态是高电平，那么它就是下降沿，反之就是上升沿。由于 CPHA=1 是偶数边沿采样，所以需要根据偶数边沿判断，假如第一个边沿即奇数边沿是下降沿，那么偶数边沿的边沿极性就是上升沿。不理解的，可以看一下下面 4 种 SPI 工作模式。

由于 CPOL 和 CPHA 都有两种不同状态，所以 SPI 分成了 4 种模式。我们在开发的时候，使用比较多的是模式 0 和模式 3。下面请看表 27.1.1.2.1 SPI 工作模式表。

SPI 工作模式	CPOL	CPHA	SCL 空闲状态	采样边沿	采样时刻
0	0	0	低电平	上升沿	奇数边沿
1	0	1	低电平	下降沿	偶数边沿
2	1	0	高电平	下降沿	奇数边沿
3	1	1	高电平	上升沿	偶数边沿

表 27.1.1.2.1 SPI 工作模式表

下面分别对 SPI 的 4 种工作模式进行分析：

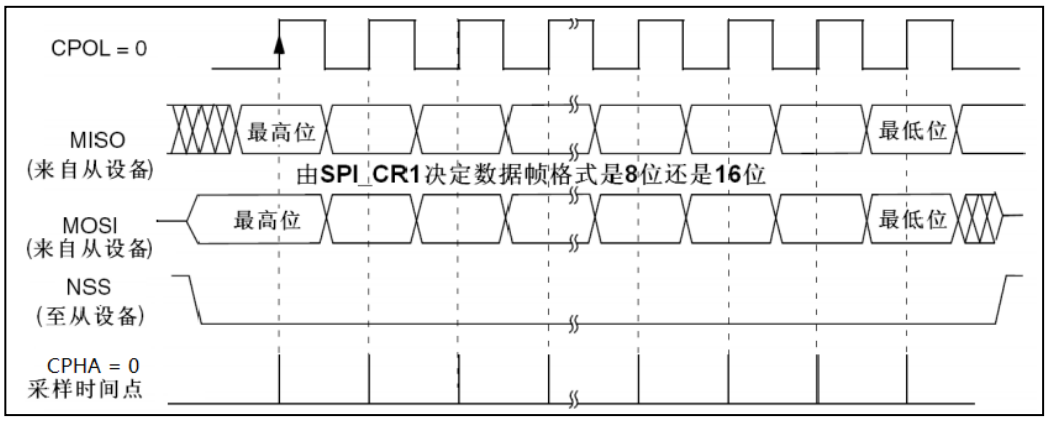


图 27.1.1.2.1 串行时钟的奇数边沿上升沿采样时序图

我们分析一下 CPOL=0&&CPHA=0 的时序，图 27.1.1.2.1 就是串行时钟的奇数边沿上升沿采样的情况，首先由于配置了 CPOL=0，可以看到当数据未发送或者发送完毕，SCL 的状态是低电平，再者 CPHA=0 即是奇数边沿采集。所以传输的数据会在奇数边沿上升沿被采集，MOSI 和 MISO 数据的有效信号需要在 SCK 奇数边沿保持稳定且被采样，在非采样时刻，MOSI 和 MISO 的有效信号才发生变化。

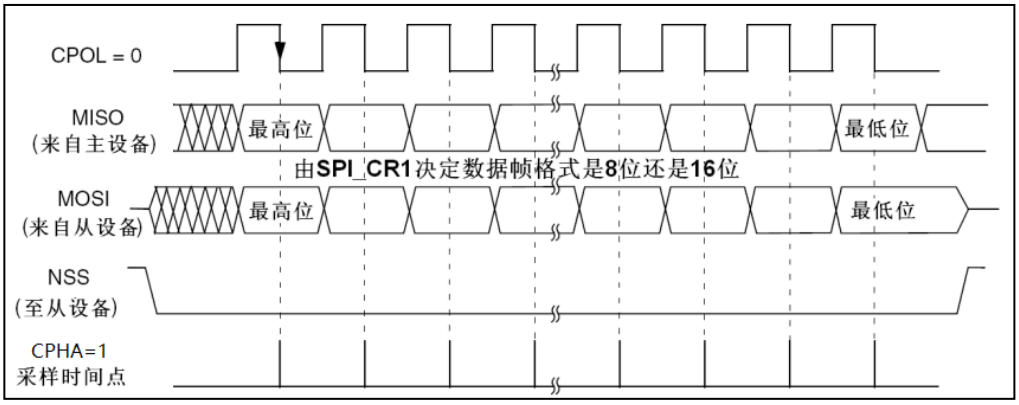


图 27.1.1.2.2 串行时钟的偶数边沿下降沿采样图

现在分析一下 CPOL=0&CPHA=1 的时序，图 36.1.1.2.2 是串行时钟的偶数边沿下降沿采样的情况。由于 CPOL=0，所以 SCL 的空闲状态依然是低电平，CPHA=1 数据就从偶数边沿采样，至于是上升沿还是下降沿，从上图就可以知道，是下降沿。这里有一个误区，空闲状态是低电平的情况下，不是应该上升沿吗，为什么这里是下降沿？首先我们先明确这里是偶数边沿采样，那么看图就很清晰，SCL 低电平空闲状态下，上升沿是在奇数边沿上，下降沿是在偶数边沿上。

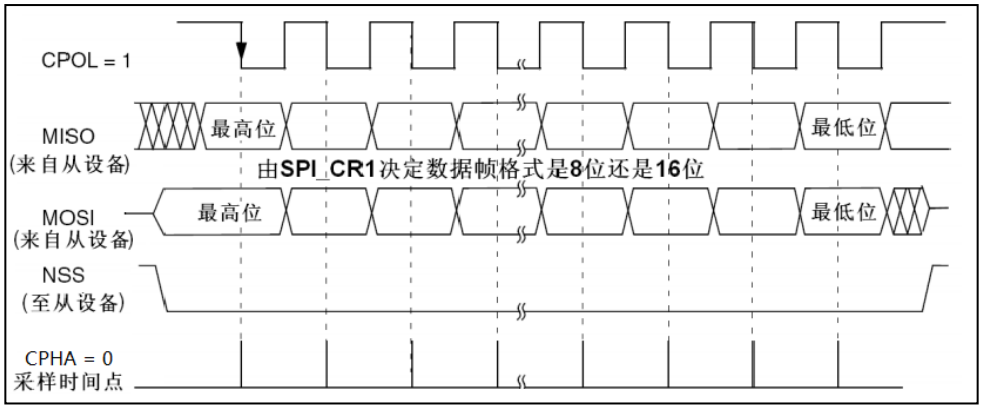


图 27.1.1.2.3 串行时钟的奇数边沿下降沿采样图

图 27.1.1.2.3 这种情况和第一种情况相似，只是这里是 CPOL=1，即 SCL 空闲状态为高电平，在 CPHA=0，奇数边沿采样的情况下，数据在奇数边沿下降沿要保持稳定并等待采样。

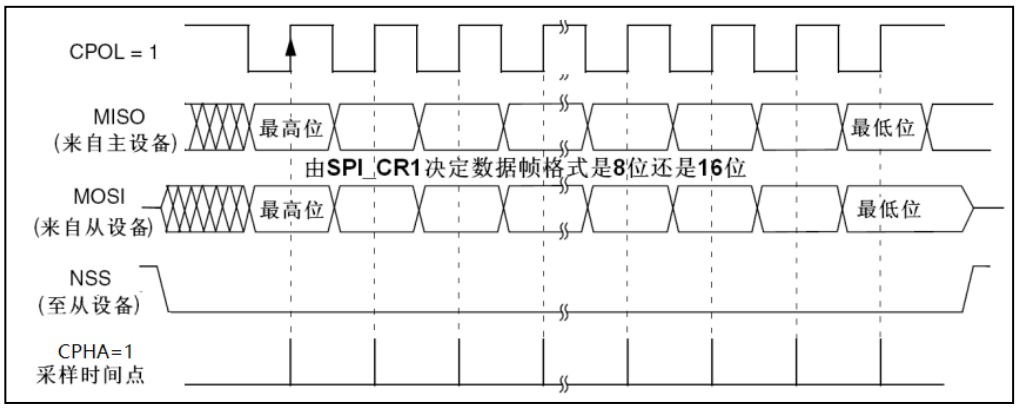


图 27.1.1.2.4 串行时钟的偶数边沿上升沿采样图

图 27.1.1.2.4 是 CPOL=1&&CPHA=1 的情形，可以看到未发送数据和发送数据完毕，SCL 的状态是高电平，奇数边沿的边沿极性是上升沿，偶数边沿的边沿极性是下降沿。因为 CPHA=1，所以数据在偶数边沿上升沿被采样。在奇数边沿的时候 MOSI 和 MISO 会发生变化，在偶数边沿时候是稳定的。

27.1.1.3 SPI 寄存器

在这里我们简单介绍一下本实验用到的寄存器。

- **SPI 控制寄存器 1（SPI_CR1）**

SPI 控制寄存器 1 描述如图 27.1.1.3.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRCEN	CRC NEXT	DFF	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR[2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位11	**DFF**：数据帧格式 (Data frame format) 0：使用8位数据帧格式进行发送/接收；　1：使用16位数据帧格式进行发送/接收。 注：只有当SPI禁止(SPE=0)时，才能写该位，否则出错。　注：I^2S模式下不使用。
位10	**RXONLY**：只接收 (Receive only) 该位和BIDIMODE位一起决定在“双线双向”模式下的传输方向。在多个从设备的配置中，在未被访问的从设备上该位被置1，使得只有被访问的从设备有输出，从而不会造成数据线上数据冲突。注：I^2S模式下不使用。 0：全双工(发送和接收)；　1：禁止输出(只接收模式)。
位9	**SSM**：软件从设备管理 (Software slave management) 当SSM被置位时，NSS引脚上的电平由SSI位的值决定。注：I^2S模式下不使用。 0：禁止软件从设备管理；　1：启用软件从设备管理。
位7	**LSBFIRST**：帧格式 (Frame format) 0：先发送MSB；　1：先发送LSB。 注：当通信在进行时不能改变该位的值。　注：I^2S模式下不使用。
位6	**SPE**：SPI使能 (SPI enable) 0：禁止SPI设备；　1：开启SPI设备。 注：I^2S模式下不使用。　注：当关闭SPI设备时，请按照第23.3.8节的过程操作。
位5:3	**BR[2:0]**：波特率控制 (Baud rate control) 000：$f_{PCLK}/2$　001：$f_{PCLK}/4$　010：$f_{PCLK}/8$　011：$f_{PCLK}/16$ 100：$f_{PCLK}/32$　101：$f_{PCLK}/64$　110：$f_{PCLK}/128$　111：$f_{PCLK}/256$ 当通信正在进行的时候，不能修改这些位。　注意：I^2S模式下不使用。
位2	**MSTR**：主设备选择 (Master selection) 0：配置为从设备；　1：配置为主设备。 注：当通信正在进行的时候，不能修改该位。　注：I^2S模式下不使用。
位1	**CPOL**：时钟极性 (Clock polarity) 0：空闲状态时，SCK保持低电平；　1：空闲状态时，SCK保持高电平。 注：当通信正在进行的时候，不能修改该位。　注：I^2S模式下不使用。
位0	**CPHA**：时钟相位 (Clock phase) 0：数据采样从第一个时钟边沿开始；　1：数据采样从第二个时钟边沿开始。 注：当通信正在进行的时候，不能修改该位。　注：I^2S模式下不使用。

图 27.1.1.3.1 SPI_CR1 寄存器（部分）

该寄存器控制着 SPI 很多相关信息，包括主设备模式选择，传输方向，数据格式，时钟极性、时钟相位和使能等。下面讲解一下本实验配置的位，在位 CPHA 置 1，数据采样从第二个时钟边沿开始；在位 CPOL 置 0，在空闲状态时，SCK 保持低电平；在位 MSTR 置 1，配置为主设备；在位 BR[2:0]置 7，使用 256 分频，速度最低；在位 SPE 置 1，开启 SPI 设备；在位 LSBFIRST 置 0，MSB 先传输；在位 SSI 置 1，禁止软件从设备，即做主机；在位 SSM 置 1，软件片选 NSS 控制；在位 RXONLY 置 0，传输方式采用的是全双工模式；在位 DFF 置 0，使用 8 位数据帧格式。

- **SPI 状态寄存器（SPI_SR）**

SPI 状态寄存器描述如图 27.1.1.3.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							FRE	BSY	OVR	MODF	CRC ERR	UDR	CHSIDE	TXE	RXNE
							r	r	r	r	rc_w0	r	r	r	r

位 1 **TXE**：发送缓冲区为空 (Transmit buffer empty)
0：发送缓冲区非空
1：发送缓冲区为空

位 0 **RXNE**：接收缓冲区非空 (Receive buffer not empty)
0：接收缓冲区为空
1：接收缓冲区非空

图 27.1.1.3.2 SPI_SR 寄存器（部分）

该寄存器是查询当前 SPI 的状态，我们在实验中用到的是 TXE 位和 RXNE 位，即发送完成和接收完成是否的标记。

- **SPI 数据寄存器（SPI_DR）**

SPI 数据寄存器描述如图 27.1.4.3 所示：

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

位 15:0 **DR[15:0]**：数据寄存器 (Data register)
已接收或者要发送的数据。
数据寄存器分为 2 个缓冲区，一个用于写入（发送缓冲区），一个用于读取（接收缓冲区）。对数据寄存器执行写操作时，数据将写入发送缓冲区，从数据寄存器执行读取时，将返回接收缓冲区中的值。
针对 **SPI** 模式的说明：
发送或接收的数据为 8 位或 16 位，具体取决于数据帧格式选择位（SPI_CR1 寄存器中的 DFF）。必须在使能 SPI 前进行此项选择，以确保操作正确。
对于 8 位数据帧，缓冲区为 8 位，只有寄存器的 LSB (SPI_DR[7:0]) 用于发送/接收。在接收模式下，寄存器的 MSB (SPI_DR[15:8]) 强制为 0。
对于 16 位数据帧，缓冲区为 16 位，整个寄存器 SPI_DR[15:0] 均用于发送/接收。

图 27.1.1.3.2 SPI_DR 寄存器

该寄存器是 SPI 数据寄存器，是一个双寄存器，包括了发送缓存和接收缓存。当向该寄存器写数据的时候，SPI 就会自动发送，当收到数据的时候，也是存在该寄存器内。

27.1.2 NOR Flash 简介

27.1.2.1 Flash 简介

Flash 是常见的用于存储数据的半导体器件，它具有容量大、可重复擦写、按“扇区/块”擦除、掉电后数据可继续保存的特性。常见的 Flash 主要有 NOR Flash 和 Nand Flash 两种类型，它们的特性如表 27.1.2.1.1 所示。NOR 和 NAND 是两种数字门电路，可以简单地认为 Flash 内部存储单元使用哪种门作存储单元就是哪类型的 Flash。U 盘，SSD，eMMC 等为 NAND 型，而 NOR Flash 则根据设计需要灵活应用于各类 PCB 上，如 BIOS，手机等。

特性	NOR FLASH	NAND FLASH

容量	较小	很大
同容量存储器成本	较贵	较便宜
擦除单元	以“扇区/块”擦除	以“扇区/块”擦除
读写单元	可以基于字节读写	必须以“块”为单位读写
读取速度	较高	较低
写入速度	较低	较高
集成度	较低	较高
介质类型	随机存储	连续存储
地址线和数据线	独立分开	共用
坏块	较少	较多
是否支持XIP	支持	不支持

表 27.1.2.1.1 NOR Flash 和 NAND Flash 特性对比

NOR 与 NAND 在数据**写入前都需要有擦除操作**，但实际上 NOR Flash 的一个 bit 可以从 1 变成 0，而要从 0 变 1 就要擦除后再写入，NAND Flash 这两种情况都需要擦除。擦除操作的最小单位为“扇区/块”，这意味着有时候即使只写一字节的数据，则这个“扇区/块”上之前的数据都可能会被擦除。

NOR 的地址线和数据线分开，它可以按“字节”读写数据，符合 CPU 的指令译码执行要求，所以假如 NOR 上存储了代码指令，CPU 给 NOR 一个地址，NOR 就能向 CPU 返回一个数据让 CPU 执行，中间不需要额外的处理操作，这体现于表 27.1.2.1.1 中的支持 XIP 特性(eXecute In Place)。因此可以用 NOR Flash 直接作为嵌入式 MCU 的程序存储空间。

NAND 的数据和地址线共用，只能按“块”来读写数据，假如 NAND 上存储了代码指令，CPU 给 NAND 地址后，它无法直接返回该地址的数据，所以不符合指令译码要求。

若代码存储在 NAND 上，可以把它先加载到 RAM 存储器上，再由 CPU 执行。所以在功能上可以认为 NOR 是一种断电后数据不丢失的 RAM，但它的擦除单位与 RAM 有区别，且读写速度比 RAM 要慢得多。

Flash 也有对应的缺点，我们在使用过程中需要尽量去规避这些问题：一是 Flash 的使用寿命，另一个是可能的位反转。

使用寿命体现在：读写上是 FLASH 的擦除次数都是有限的(NOR Flash 普遍是 10 万次左右)，当它的使用接近寿命的时候，可能会出现写操作失败。由于 NAND 通常是整块擦写，块内有一位失效整个块就会失效，这被称为坏块。使用 NAND Flash 最好通过算法扫描介质找出坏块并标记为不可用，因为坏块上的数据是不准确的。

位反转是数据位写入时为 1，但经过一定时间的环境变化后可能实际变为 0 的情况，反之亦然。位反转的原因很多，可能是器件特性也可能与环境、干扰有关，由于位反转的问题可能存在，所以 FLASH 存储器需要“探测/错误更正(EDC/ECC)”算法来确保数据的正确性。

FLASH 芯片有很多种芯片型号，在我们的 norflash.h 头文件中有定义芯片 ID 的宏定义，对应的就是不同型号的 NOR FLASH 芯片，比如有：W25Q128、BY25Q128、NM25Q128，它们是来自不同的厂商的同种规格的 NOR FLASH 芯片，内存空间都是 128M 字，即 16M 字节。它们的很多参数、操作都是一样的，所以我们的实验都是兼容它们的。

下面我们以 NM25Q128 为例，认识一下具体的 NOR Flash 的特性。

NM25Q128 是一款大容量 SPI FLASH 产品，其容量为 16M。它将 16M 字节的容量分为 256 个块（Block），每个块大小为 64K 字节，每个块又分为 16 个扇区（Sector），每一个扇区 16 页，每页 256 个字节，即每个扇区 4K 个字节。NM25Q128 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 NM25Q128 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。

NM25Q128 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，NM25Q128 支持标准的 SPI，还支持双输出/四输出的 SPI，最大 SPI 时钟可以到 80Mhz（双输出时相当于 160Mhz，四输出时相当于 320Mhz）。

下面我们看一下 NM25Q128 芯片的管脚图，如图 27.1.2.1.2 所示。

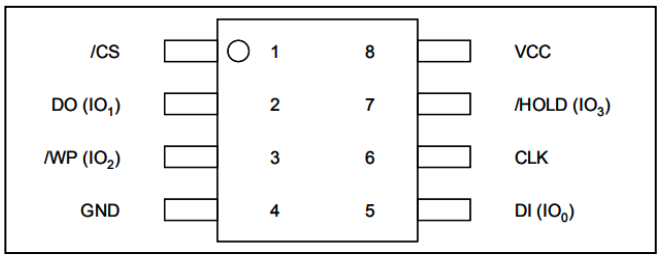


图 27.1.2.1.2 NM25Q128 芯片引脚图

芯片引脚连接如下：CS 即片选信号输入，低电平有效；DO 是 MISO 引脚，在 CLK 管脚的下降沿输出数据；WP 是写保护管脚，高电平可读可写，低电平仅仅可读；DI 是 MOSI 引脚，主机发送的数据、地址和命令从 SI 引脚输入到芯片内部，在 CLK 管脚的上升沿捕获数据；CLK 是串行时钟引脚，为输入输出提供时钟脉冲；HOLD 是保持管脚，低电平有效。

STM32F407 通过 SPI 总线连接到 NM25Q128 对应的引脚即可启动数据传输。

27.1.2.2 NOR FLASH 工作时序

前面对于 NM25Q128 的介绍中也提及其存储的体系，NM25Q128 有写入、读取还有擦除的功能，下面就对这三种操作的时序进行分析，在后面通过代码的形式驱动它。

下面先让我们看一下读操作时序，如图 27.1.2.1 所示：

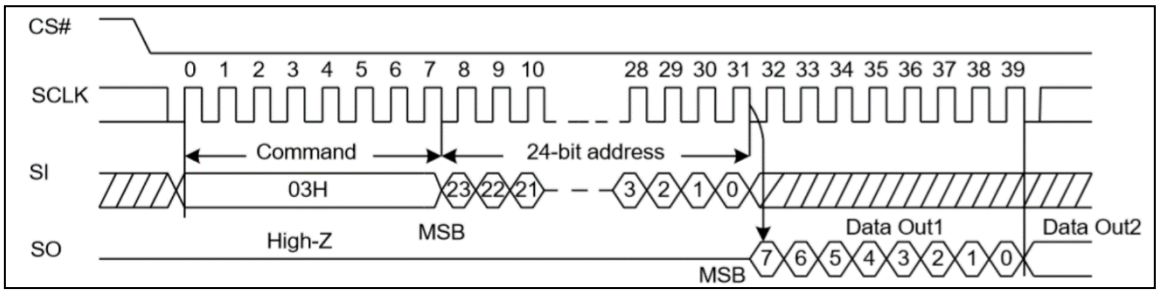


图 27.1.2.2.1 NM25Q128 读操作时序图

从上图可知读数据指令是 03H，可以读出一个字节或者多个字节。发起读操作时，先把 CS 片选管脚拉低，然后通过 MOSI 引脚把 03H 发送芯片，之后再发送要读取的 24 位地址，这些数据在 CLK 上升沿时采样。芯片接收完 24 位地址之后，就会把相对应地址的数据在 CLK 引脚下降沿从 MISO 引脚发送出去。从图中可以看出只要 CLK 一直在工作，那么通过一条读指令就可以把整个芯片存储区的数据读出来。当主机把 CS 引脚拉高，数据传输停止。

接着我们看一下写时序，这里我们先看页写时序，如图 27.1.2.2.2 所示：

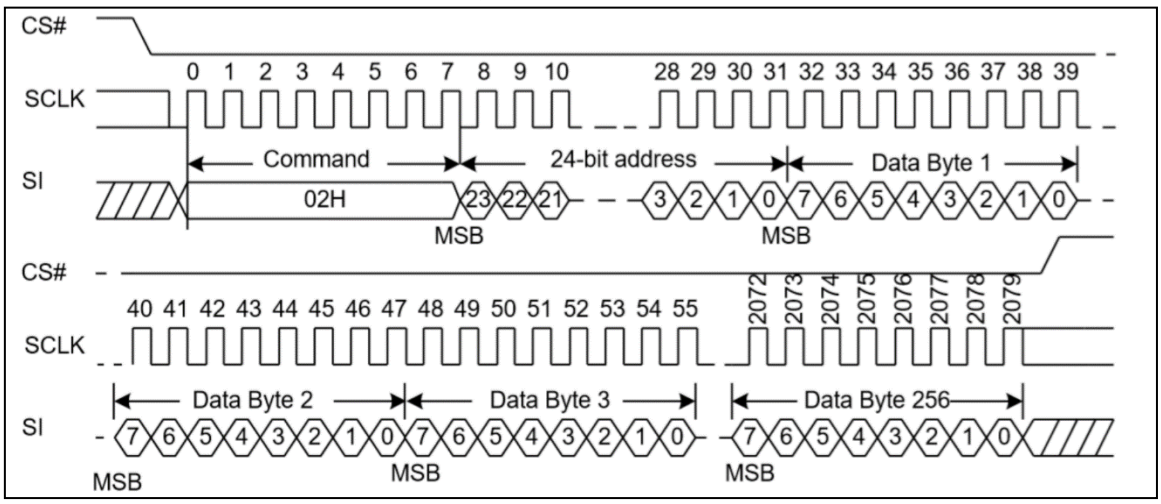


图 27.1.2.2.2 NM25Q128 页写时序

在发送页写指令之前，需要先发送“写使能”指令。然后主机拉低 CS 引脚，然后通过 MOSI 引脚把 02H 发送到芯片，接着发送 24 位地址，最后你就可以发送你需要写的字节数据到芯片。

完成数据写入之后，需要拉高 CS 引脚，停止数据传输。

下面介绍一下扇区擦除时序，如图 27.1.2.2.3 所示：

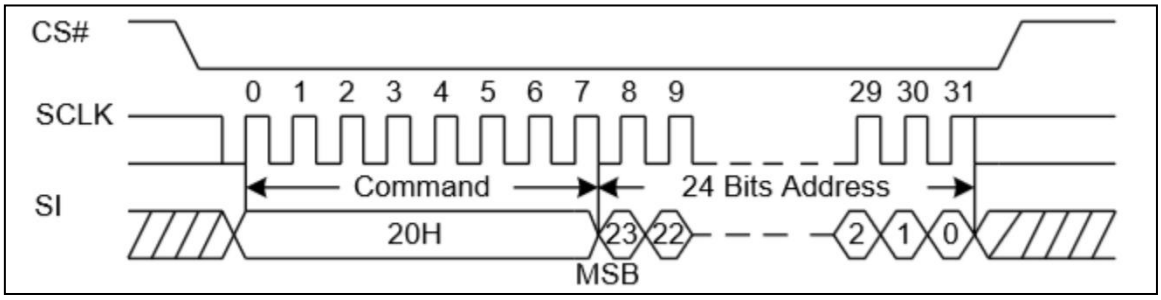


图 27.1.2.2.3 扇区擦除时序图

扇区擦除指的是将一个扇区擦除，通过前面的介绍也知道，NM25Q128 的扇区大小是 4K 字节。擦除扇区后，扇区的位全置 1，即扇区字节为 FFh。同样的，在执行扇区擦除之前，需要先执行写使能指令。这里需要注意的是当前 SPI 总线的状态，假如总线状态是 BUSY，那么这个扇区擦除是无效的，所以在拉低 CS 引脚准备发送数据前，需要先确定 SPI 总线的状态，这就需要执行读状态寄存器指令，读取状态寄存器的 BUSY 位，需要等待 BUSY 位为 0，才可以执行擦除工作。

接着按时序图分析，主机先拉低 CS 引脚，然后通过 MOSI 引脚发送指令代码 20h 到芯片，然后接着把 24 位扇区地址发送到芯片，然后需要拉高 CS 引脚，通过读取寄存器状态等待扇区擦除操作完成。

此外还有对整个芯片进行擦除的操作，时序比扇区擦除更加简单，不用发送 24bit 地址，只需要发送指令代码 C7h 到芯片即可实现芯片的擦除。

在 NM25Q128 手册中还有许多种方式的读/写/擦除操作，我们这里只分析本实验用到的，其他大家可以参考 NM25Q128 手册。

27.2 硬件设计

1. 例程功能

通过 KEY1 按键来控制 norflash 的写入，通过按键 KEY0 来控制 norflash 的读取。并在 LCD 模块上显示相关信息。我们还可以通过 USMART 控制读取 norflash 的 ID、擦除某个扇区或整片擦除。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
3）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)(USMART 使用)
4）正点原子 2.8/3.5/4.3/7 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
5）SPI2(PI0/PI1/PI2/PI3)
6）norflash(本例程使用的是 W25Q128，连接在 SPI2)

3. 原理图

我们主要来看看 norflash 和开发板的连接，如图 27.3.1 所示：

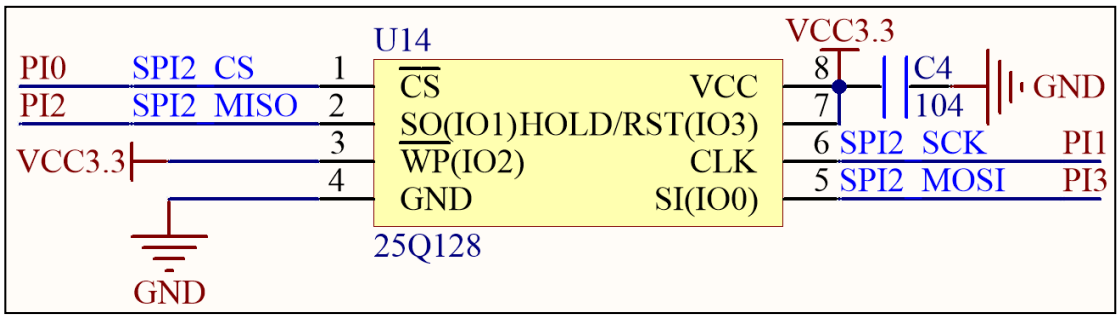


图 27.3.1 NOR FLASH 与开发板的连接原理

通过上图可知，NOR FLASH 的 CS、SCK、MISO 和 MOSI 分别连接在 PI0、PI1、PI2 和 PI3 上。本实验还支持多种型号的 SPI FLASH 芯片，比如：BY25Q128/NM25Q128/W25Q128 等等，具体请看 norflash.h 文件的宏定义，在程序上只需要稍微修改一下，后面讲解程序的时候会提到。

27.3 程序设计

27.3.1 SPI 的 HAL 库驱动

SPI 在 HAL 库中的驱动代码在 stm32f4xx_hal_spi.c 文件（及其头文件）中。

1. HAL_SPI_Init 函数

SPI 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

- **函数描述：**

  用于初始化 SPI。

- **函数形参：**

  形参 1 是 SPI_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct __SPI_HandleTypeDef
{
  SPI_TypeDef                *Instance;      /* SPI 寄存器基地址 */
  SPI_InitTypeDef            Init;          /* SPI 通信参数 */
  uint8_t                    *pTxBuffPtr;   /* SPI 的发送缓存 */
  uint16_t                   TxXferSize;    /* SPI 的发送数据大小 */
  __IO uint16_t              TxXferCount;   /* SPI 发送端计数器 */
  uint8_t                    *pRxBuffPtr;   /* SPI 的接收缓存 */
  uint16_t                   RxXferSize;    /* SPI 的接收数据大小 */
  __IO uint16_t              RxXferCount;   /* SPI 接收端计数器 */
  void (*RxISR)(struct __SPI_HandleTypeDef *hspi);   /* SPI 的接收端中断服务函数 */
  void (*TxISR)(struct __SPI_HandleTypeDef *hspi);   /* SPI 的发送端中断服务函数 */
  DMA_HandleTypeDef          *hdmatx;       /* SPI 发送参数设置(DMA) */
  DMA_HandleTypeDef          *hdmarx;       /* SPI 接收参数设置(DMA) */
  HAL_LockTypeDef            Lock;          /* SPI 锁对象 */
  __IO HAL_SPI_StateTypeDef  State;         /* SPI 传输状态 */
  __IO uint32_t              ErrorCode;     /* SPI 操作错误代码 */
} SPI_HandleTypeDef;
```

我们这里主要讲解第二个成员变量 Init，它是 SPI_InitTypeDef 结构体类型，该结构体定义如下：

```
typedef struct
{
  uint32_t Mode;                /* 模式：主:SPI_MODE_MASTER  从:SPI_MODE_SLAVE */
  uint32_t Direction;           /* 方向：只接收模式 单线双向通信数据模式 全双工 */
  uint32_t DataSize;            /* 数据帧格式：8 位/16 位 */
  uint32_t CLKPolarity;         /* 时钟极性 CPOL 高/低电平 */
  uint32_t CLKPhase;            /* 时钟相位 奇/偶数边沿采集 */
  uint32_t NSS;                 /* SS 信号由硬件（NSS）管脚控制还是软件控制 */
```

```
  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;       /* 设置 CRC 多项式*/
} SPI_InitTypeDef;
```

- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

使用 SPI 传输数据的配置步骤

1）SPI 参数初始化(工作模式、数据时钟极性、时钟相位等)。

HAL 库通过调用 SPI 初始化函数 HAL_SPI_Init 完成对 SPI 参数初始化，详见例程源码。

注意：该函数会调用：HAL_SPI_MspInit 函数来完成对 SPI 底层的初始化，包括：SPI 及 GPIO 时钟使能、GPIO 模式设置等。

2）使能 SPI 时钟和配置相关引脚的复用功能。

本实验用到 SPI2，使用 PI1、PI2 和 PI3 作为 SPI_SCK、SPI_MISO 和 SPI_MOSI，因此需要先使能 SPI2 和 GPIOI 时钟。参考代码如下：

```
__HAL_RCC_SPI2_CLK_ENABLE();
__HAL_RCC_GPIOI_CLK_ENABLE();
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

3）使能 SPI

通过__HAL_SPI_ENABLE 函数使能 SPI，便可进行数据传输。

4）SPI 传输数据

通过 HAL_SPI_Transmit 函数进行发送数据。

通过 HAL_SPI_Receive 函数进行接收数据。

也可以通过 HAL_SPI_TransmitReceive 函数进行发送与接收操作。

5）设置 SPI 传输速度

SPI 初始化结构体 SPI_InitTypeDef 有一个成员变量是 BaudRatePrescaler，该成员变量用来设置 SPI 的预分频系数，从而决定了 SPI 的传输速度。但是 HAL 库并没有提供单独的 SPI 分频系数修改函数，如果我们需要在程序中偶尔修改速度，那么我们就要通过设置 SPI_CR1 寄存器来修改，具体实现方法请参考后面软件设计小节相关函数。

27.3.2 程序流程图

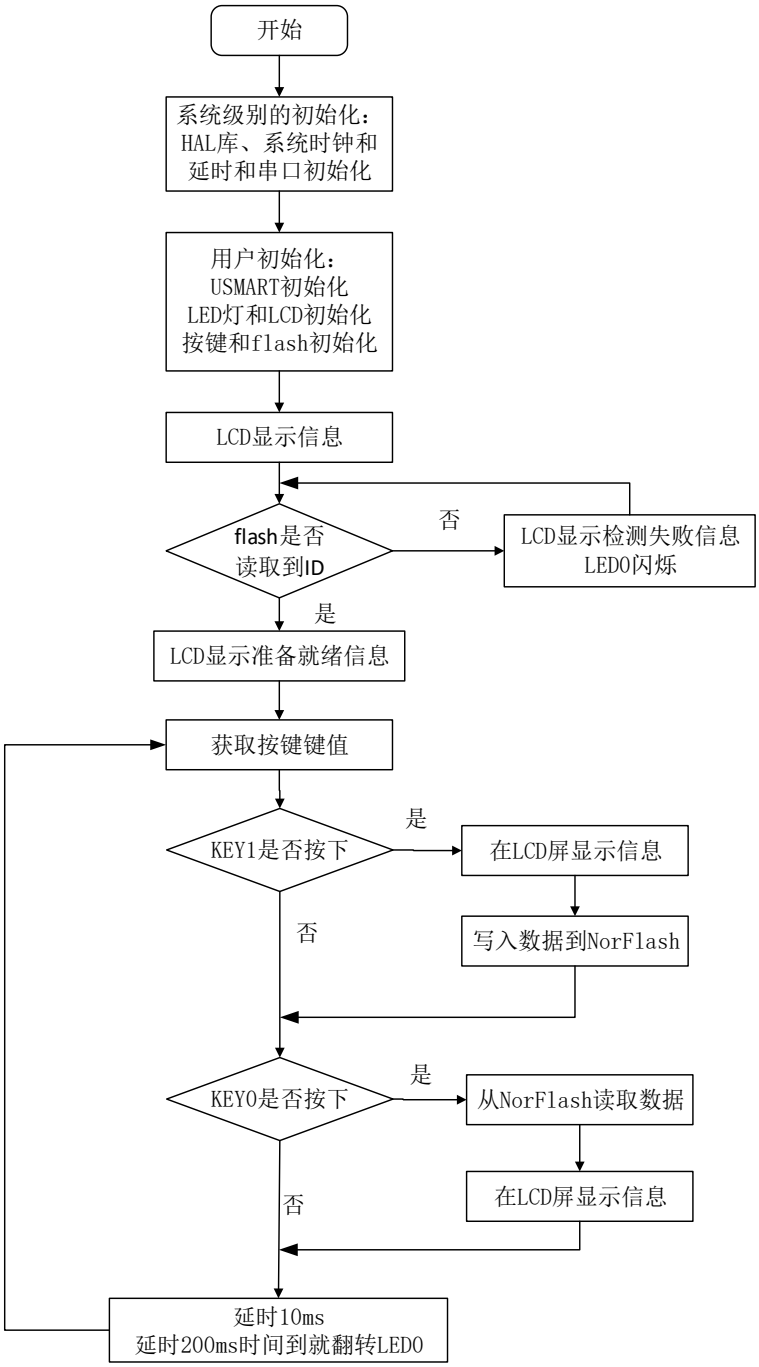


图 27.3.2.1 SPI 实验程序流程图

27.3.3 程序解析

本实验中，我们通过调用 HAL 库的函数去驱动 SPI 进行通信，所以需要在工程中的 Driver/STM32F4xx_HAL_Driver 分组下添加 stm32f4xx_hal_spi.c 文件。实验工程中，我们新增了 spi.c 存放 spi 底层驱动代码，norflash.c 文件存放 BY25Q128/W25Q128/NM25Q128 驱动。

1. SPI 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。SPI 驱动源码包括两个文件：spi.c 和 spi.h。

下面我们直接介绍 SPI 相关的程序，首先先介绍 spi.h 文件，其定义如下：

```
/* SPI2 引脚 定义 */
#define SPI2_SCK_GPIO_PORT              GPIOI
#define SPI2_SCK_GPIO_PIN               GPIO_PIN_1
#define SPI2_SCK_GPIO_CLK_ENABLE()      do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)

#define SPI2_MISO_GPIO_PORT             GPIOI
#define SPI2_MISO_GPIO_PIN              GPIO_PIN_2
#define SPI2_MISO_GPIO_CLK_ENABLE()     do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)

#define SPI2_MOSI_GPIO_PORT             GPIOI
#define SPI2_MOSI_GPIO_PIN              GPIO_PIN_3
#define SPI2_MOSI_GPIO_CLK_ENABLE()     do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)

/* SPI2 相关定义 */
#define SPI2_SPI                        SPI2
#define SPI2_SPI_CLK_ENABLE()           do{ __HAL_RCC_SPI2_CLK_ENABLE(); }while(0)
```

我们通过宏定义标识符的方式去定义 SPI 通信用到的三个管脚 SCK、MISO 和 MOSI，同时还宏定义 SPI2 的相关信息。

接下来我们看一下 spi.c 代码中的初始化函数，代码如下：

```
/**
 * @brief       SPI 初始化代码
 *   @note      主机模式,8 位数据,禁止硬件片选
 * @param       无
 * @retval      无
 */
SPI_HandleTypeDef g_spi2_handler;               /* SPI2 句柄 */

void spi2_init(void)
{
    SPI2_SPI_CLK_ENABLE();                      /* SPI2 时钟使能 */

    g_spi2_handler.Instance = SPI2_SPI;         /* SPI2 */
    g_spi2_handler.Init.Mode = SPI_MODE_MASTER; /* 设置 SPI 工作模式，设置为主模式 */
    /* 设置 SPI 单向或者双向的数据模式:SPI 设置为双线模式 */
    g_spi2_handler.Init.Direction = SPI_DIRECTION_2LINES;
    /* 设置 SPI 的数据大小:SPI 发送接收 8 位帧结构 */
    g_spi2_handler.Init.DataSize = SPI_DATASIZE_8BIT;
    /* 串行同步时钟的空闲状态为高电平 */
    g_spi2_handler.Init.CLKPolarity = SPI_POLARITY_HIGH;
    /* 串行同步时钟的第二个跳变沿（上升或下降）数据被采样 */
    g_spi2_handler.Init.CLKPhase = SPI_PHASE_2EDGE;
    /* NSS 信号由硬件（NSS 管脚）还是软件（使用 SSI 位）管理:内部 NSS 信号有 SSI 位控制 */
    g_spi2_handler.Init.NSS = SPI_NSS_SOFT;
    /* 定义波特率预分频的值:波特率预分频值为 256 */
    g_spi2_handler.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
    /* 指定数据传输从 MSB 位还是 LSB 位开始:数据传输从 MSB 位开始 */
    g_spi2_handler.Init.FirstBit = SPI_FIRSTBIT_MSB;
    g_spi2_handler.Init.TIMode = SPI_TIMODE_DISABLE;    /* 关闭 TI 模式 */
    /* 关闭硬件 CRC 校验 */
    g_spi2_handler.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    g_spi2_handler.Init.CRCPolynomial = 7;              /* CRC 值计算的多项式 */
    HAL_SPI_Init(&g_spi2_handler);                      /* 初始化 */

    __HAL_SPI_ENABLE(&g_spi2_handler);                  /* 使能 SPI2 */
    /* 启动传输，实际上就是产生 8 个时钟脉冲，达到清空 DR 的作用，非必需 */
    spi2_read_write_byte(0xFF);
}
```

在 spi_init 函数中主要工作就是对于 SPI 参数的配置，这里包括工作模式、数据模式、数据大小、时钟极性、时钟相位、波特率预分频值等。关于 SPI 的管脚配置就放在了 HAL_SPI_MspInit 函数里，其代码如下：

```
/**
 * @brief       SPI 底层驱动，时钟使能，引脚配置
 *   @note      此函数会被 HAL_SPI_Init()调用
 * @param       hspi:SPI 句柄
 * @retval      无
 */
void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi)
{
    GPIO_InitTypeDef GPIO_Initure;
    if (hspi->Instance == SPI2_SPI)
    {
        SPI2_SCK_GPIO_CLK_ENABLE();  /* SPI2_SCK 脚时钟使能 */
        SPI2_MISO_GPIO_CLK_ENABLE(); /* SPI2_MISO 脚时钟使能 */
        SPI2_MOSI_GPIO_CLK_ENABLE(); /* SPI2_MOSI 脚时钟使能 */

        GPIO_Initure.Pin = SPI2_SCK_GPIO_PIN;
        GPIO_Initure.Mode = GPIO_MODE_AF_PP;     /* SCK 引脚模式设置(复用输出) */
        GPIO_Initure.Pull = GPIO_PULLUP;
        GPIO_Initure.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(SPI2_SCK_GPIO_PORT, &GPIO_Initure);

        GPIO_Initure.Pin = SPI2_MISO_GPIO_PIN;  /* MISO 引脚模式设置(复用输出) */
        HAL_GPIO_Init(SPI2_MISO_GPIO_PORT, &GPIO_Initure);

        GPIO_Initure.Pin = SPI2_MOSI_GPIO_PIN;  /* MOSI 引脚模式设置(复用输出) */
        HAL_GPIO_Init(SPI2_MOSI_GPIO_PORT, &GPIO_Initure);
    }
}
```

通过以上两个函数的作用就可以完成 SPI 初始。接下来介绍 SPI 的发送和接收函数，其定义如下：

```
/**
 * @brief       SPI2 读写一个字节数据
 * @param       txdata  : 要发送的数据(1 字节)
 * @retval      接收到的数据(1 字节)
 */
uint8_t spi2_read_write_byte(uint8_t txdata)
{
    uint8_t rxdata;
    HAL_SPI_TransmitReceive(&g_spi2_handler, &txdata, &rxdata, 1, 1000);
    return rxdata; /* 返回收到的数据 */
}
```

这里的 spi_read_write_byte 函数直接调用了 HAL 库内置的函数进行接收发送操作。前面已经有介绍了，这里就不展开对 HAL_SPI_TransmitReceive 函数的解析。

由于不同的外设需要的通信速度不一样，所以这里我们定义了一个速度设置函数，通过操作寄存器的方式去实现，其代码如下：

```
/**
 * @brief       SPI2 速度设置函数
 *   @note      SPI2 时钟选择来自 APB1，即 PCLK1，为 42Mhz
 *              SPI 速度 = PCLK1 / 2^(speed + 1)
 * @param       speed  : SPI2 时钟分频系数
 *              取值为 SPI_BAUDRATEPRESCALER_2~SPI_BAUDRATEPRESCALER_2 256
 * @retval      无
 */
void spi2_set_speed(uint8_t speed)
{
    assert_param(IS_SPI_BAUDRATE_PRESCALER(speed)); /* 判断有效性 */
    __HAL_SPI_DISABLE(&g_spi2_handler);             /* 关闭 SPI */
    g_spi2_handler.Instance->CR1 &= 0xFFC7;         /* 位 3-5 清零，用来设置波特率 */
    g_spi2_handler.Instance->CR1 |= speed << 3;     /* 设置 SPI 速度 */
    __HAL_SPI_ENABLE(&g_spi2_handler);              /* 使能 SPI */
}
```

2. norflash 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。NOR FLASH 驱动源码包括两个文件：norflash.c 和 norflash.h。

在上一小节已经对 SPI 协议需要用到的东西都封装好了。那么现在就要在 SPI 通信的基础上，通过前面分析的 NM25Q128 的工作时序拟定通信代码。

由于这部分的代码量比较多，这里就不一一贴出来介绍。介绍几个重点，其余的请自行查看源码。首先是 norflash.h 头文件，我们做了一个 FLASH 芯片列表（宏定义），这些宏定义是一些支持的 FLASH 芯片的 ID。接下来是 FLASH 芯片指令表的宏定义，这个请参考 FLASH 芯片手册比对得到，这里就不将代码列出来了。

下面介绍 norflash.c 文件几个重要的函数，首先是 NOR FLASH 初始化函数，其定义如下：

```
/**
 * @brief       初始化 SPI NOR FLASH
 * @param       无
 * @retval      无
 */
void norflash_init(void)
{
    uint8_t temp;

    NORFLASH_CS_GPIO_CLK_ENABLE();    /* NORFLASH CS 脚 时钟使能 */

    /* CS 引脚模式设置(复用输出) */
    GPIO_InitTypeDef gpio_init_struct;
    gpio_init_struct.Pin = NORFLASH_CS_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(NORFLASH_CS_GPIO_PORT, &gpio_init_struct);

    NORFLASH_CS(1); /* 取消片选 */

    spi2_init();     /* 初始化 SPI2 */
    spi2_set_speed(SPI_SPEED_4);            /* SPI2 切换到高速状态 21Mhz */

    g_norflash_type = norflash_read_id();   /* 读取 FLASH ID. */

    if (g_norflash_type == W25Q256)/* SPI FLASH 为 W25Q256, 必须使能 4 字节地址模式 */
    {
        temp = norflash_read_sr(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 */
        {
            norflash_write_enable(); /* 写使能 */
            temp |= 1 << 1;          /* ADP=1, 上电 4 位地址模式 */
            norflash_write_sr(3, temp);    /* 写 SR3 */

            NORFLASH_CS(0);
            spi1_read_write_byte(FLASH_Enable4ByteAddr);    /* 使能 4 字节地址指令 */
            NORFLASH_CS(1);
        }
    }

    //printf("ID:%x\r\n", g_norflash_type);
}
```

在初始化函数中，将 SPI 通信协议用到的 CS 引脚配置好，同时根据 FLASH 的通信要求，通过调用 spi2_set_speed 函数把 SPI2 切换到高速状态。然后尝试读取 flash 的 ID，由于 W25Q256 的容量比较大，通信的时候需要 4 个字节，为了函数的兼容性，我们这里做了判断处理。当然，我们使用的 W25Q128 是 3 字节地址模式的。如果能读到 ID 则说明我们的 SPI 时序能正常操作 Flash，便可以通过 SPI 接口读写 NOR FLASH 的数据了。

进行其它数据操作时，由于每一次读写操作的时候都需要发送地址，所以这里我们把这个板块封装成函数，函数名是 norflash_send_address，实质上就是通过 SPI 的发送接收函数 spi2_read_write_byte 实现的，这里就不列出来了，大家可以查看光盘源码。

下面介绍一下 FLASH 读取函数，这里可以根据前面的时序图对照理解，其定义如下：

```
/**
 * @brief       读取 SPI FLASH
 *   @note      在指定地址开始读取指定长度的数据
 * @param       pbuf    : 数据存储区
 * @param       addr    : 开始读取的地址(最大 32bit)
 * @param       datalen : 要读取的字节数(最大 65535)
 * @retval      无
 */
void norflash_read(uint8_t *pbuf, uint32_t addr, uint16_t datalen)
{
    uint16_t i;

    NORFLASH_CS(0);
    spi2_read_write_byte(FLASH_ReadData);       /* 发送读取命令 */
    norflash_send_address(addr);                /* 发送地址 */

    for(i=0;i<datalen;i++)
    {
        pbuf[i] = spi2_read_write_byte(0xFF);   /* 循环读取 */
    }

    NORFLASH_CS(1);
}
```

该函数用于从 NOR FLASH 的指定位置读出指定长度的数据，由于 NOR FLASH 支持以任意地址（但是不能超过 NOR FLASH 的地址范围）开始读取数据，所以，这个代码相对来说比较简单。首先拉低片选信号，发送读取命令，接着发送 24 位地址之后，程序就可以开始循环读数据，其地址就会自动增加，读取完数据后，需要拉高片选信号，结束通信。

有读函数，那肯定就有写函数，接下来我们介绍一下 NOR FLASH 写函数，其定义如下：

```
/**
 * @brief       写 SPI FLASH
 *   @note      在指定地址开始写入指定长度的数据 , 该函数带擦除操作!
 *              SPI FLASH 一般是: 256 个字节为一个 Page,4Kbytes 为一个 Sector, 16 个扇区
 *              为 1 个 Block,擦除的最小单位为 Sector.
 *
 * @param       pbuf    : 数据存储区
 * @param       addr    : 开始写入的地址(最大 32bit)
 * @param       datalen : 要写入的字节数(最大 65535)
 * @retval      无
 */
uint8_t g_norflash_buf[4096];   /* 扇区缓存 */
void norflash_write(uint8_t *pbuf, uint32_t addr, uint16_t datalen)
{
    uint32_t secpos;
    uint16_t secoff;
    uint16_t secremain;
    uint16_t i;
    uint8_t *norflash_buf;

    norflash_buf = g_norflash_buf;
    secpos = addr / 4096;       /* 扇区地址 */
    secoff = addr % 4096;       /* 在扇区内的偏移 */
    secremain = 4096 - secoff;  /* 扇区剩余空间大小 */

    if (datalen <= secremain)
    {
        secremain = datalen;    /* 不大于 4096 个字节 */
```

```
    }

    while (1)
    {
        norflash_read(norflash_buf, secpos * 4096, 4096);   /* 读出整个扇区的内容 */
        for (i = 0; i < secremain; i++)   /* 校验数据 */
        {
            if (norflash_buf[secoff + i] != 0xFF)
            {
                break;          /* 需要擦除, 直接退出 for 循环 */
            }
        }

        if (i < secremain)   /* 需要擦除 */
        {
            norflash_erase_sector(secpos);   /* 擦除这个扇区 *
            for (i = 0; i < secremain; i++)  /* 复制 */
            {
                norflash_buf[i + secoff] = pbuf[i];
            }
            /* 写入整个扇区 */
            norflash_write_nocheck(norflash_buf, secpos * 4096, 4096);
        }
        else    /* 写已经擦除了的,直接写入扇区剩余区间. */
        {
            norflash_write_nocheck(pbuf, addr, secremain);    /* 直接写扇区 */
        }

        if (datalen == 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 */
            pbuf += secremain;       /* 指针偏移 */
            addr += secremain;       /* 写地址偏移 */
            datalen -= secremain;    /* 字节数递减 */
            if (datalen > 4096)
            {
                secremain = 4096;    /* 下一个扇区还是写不完 */
            }
            else
            {
                secremain = datalen; /* 下一个扇区可以写完了 */
            }
        }
    }
}
```

该函数可以在 NOR FLASH 的任意地址开始写入任意长度（必须不超过 NOR FLASH 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 g_norflash_buf 的全局变量，用于擦除时缓存扇区内的数据。

简单介绍一下写函数的实质调用，它用到的是通过无检验写 SPI_FLASH 函数实现的，而最终是用到页写函数 norflash_write_page，在前面也对页写时序进行了分析，现在看一下代码：

```
/**
 * @brief       SPI 在一页(0~65535)内写入少于 256 个字节的数据
 *   @note      在指定地址开始写入最大 256 字节的数据
 * @param       pbuf    : 数据存储区
 * @param       addr    : 开始写入的地址(最大 32bit)
 * @param       datalen : 要写入的字节数(最大 256),该数不应该超过该页的剩余字节数!!!
 * @retval      无
 */
static void norflash_write_page(uint8_t *pbuf, uint32_t addr, uint16_t datalen)
{
    uint16_t i;

    norflash_write_enable();    /* 写使能 */

    NORFLASH_CS(0);
    spi2_read_write_byte(FLASH_PageProgram);    /* 发送写页命令 */
    norflash_send_address(addr);                /* 发送地址 */

    for (I = 0; i < datalen; i++)
    {
        spi2_read_write_byte(pbuf[i]);          /* 循环写入 */
    }
    NORFLASH_CS(1);
    norflash_wait_busy();    /* 等待写入结束 */
}
```

在页写功能的代码中，先发送写使能命令，才发送页写命令，然后发送写入的地址，再把写入的内容通过一个 for 循环写入，发送完后拉高片选 CS 引脚结束通信，等待 flash 内部写入结束。检测 flash 内部的状态可以通过查看 NM25Qxx 状态寄存器 1 的位 0。在这里科普一下 NM25Qxx 的状态寄存器，可以通过寄存器相关位判断 NM25Qxx 的状态,下面是 NM25Qxx 状态寄存器表：

状态寄存器	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
状态寄存器 1	SPR	RV	TB	BP2	BP1	BP0	WEL	BUSY
状态寄存器 2	SUS	CMP	LB3	LB2	LB1	(R)	QE	SRP1
状态寄存器 3	HOLD/RST	DRV1	DRV0	(R)	(R)	WPS	ADP	ADS

表 27.3.3.1 NM25Qxx 状态寄存器表

我们也定义了一个函数 norflash_read_sr，去读取 NM25Qxx 状态寄存器的值，这里就不列出来了，主要实现的方式也是老套路：根据传参判断需要获取的是哪个状态寄存器，然后拉低片选线，调用 spi2_read_write_byte 函数发送该寄存器的命令，然后通过发送一字节空数据获取读取到的数据，最后拉高片选线，函数返回读取到的值。

在 norflash_write_page 函数的基础上，增加了 norflash_write_nocheck 函数进行封装解决写入字节可能大于该页剩下的字节数问题，方便解决写入错误问题，其代码如下：

```
/**
 * @brief       无检验写 SPI FLASH
 *   @note      必须确保所写的地址范围内的数据全部为 0XFF,否则在非 0XFF 处写入的数据将失败!
 *              具有自动换页功能
 *              在指定地址开始写入指定长度的数据,但是要确保地址不越界!
 *
 * @param       pbuf    : 数据存储区
 * @param       addr    : 开始写入的地址(最大 32bit)
 * @param       datalen : 要写入的字节数(最大 65535)
 * @retval      无
 */
static void norflash_write_nocheck(uint8_t *pbuf, uint32_t addr,
                                   uint16_t datalen)
{
    uint16_t pageremain;
    pageremain = 256 - addr % 256;  /* 单页剩余的字节数 */
```

```
    if (datalen <= pageremain)      /* 不大于 256 个字节 */
    {
        pageremain = datalen;
    }
    while (1)
    {
    /* 当写入字节比页内剩余地址还少的时候，一次性写完
     * 当写入直接比页内剩余地址还多的时候，先写完整个页内剩余地址，然后根据剩余长度进行不同处理
     */
        norflash_write_page(pbuf, addr, pageremain);

        if (datalen == pageremain)  /* 写入结束了 */
        {
            break;
        }
        else    /* datalen > pageremain */
        {
            pbuf += pageremain;     /* pbuf 指针地址偏移,前面已经写了 pageremain 字节 */
            addr += pageremain;     /* 写地址偏移,前面已经写了 pageremain 字节 */
            datalen -= pageremain; /* 写入总长度减去已经写入了的字节数 */

            if (datalen > 256)      /* 剩余数据还大于一页,可以一次写一页 */
            {
                pageremain = 256;   /* 一次可以写入 256 个字节 */
            }
            else    /* 剩余数据小于一页,可以一次写完 */
            {
                pageremain = datalen;    /* 不够 256 个字节了 */
            }
        }
    }
}
```

上面函数的实现主要是逻辑处理，通过判断传参中的写入字节的长度与单页剩余的字节数，来决定是否是需要在新页写入剩下的字节。这里需要大家自行理解一下。通过调用该函数实现了 norflash_write 的功能。

下面简单介绍一下擦除函数 norflash_erase_sector，前面工作时序中也有对此描述，现在就来看一下代码：

```
/**
 * @brief       擦除一个扇区
 *   @note      注意,这里是扇区地址,不是字节地址!!
 *              擦除一个扇区的最少时间:150ms
 *
 * @param       saddr : 扇区地址 根据实际容量设置
 * @retval      无
 */
void norflash_erase_sector(uint32_t saddr)
{
    saddr *= 4096;
    norflash_write_enable();     /* 写使能 */
    norflash_wait_busy();        /* 等待空闲 */
    NORFLASH_CS(0);
    spi2_read_write_byte(FLASH_SectorErase);    /* 发送写页命令 */
    norflash_send_address(saddr);   /* 发送地址 */
    NORFLASH_CS(1);
    norflash_wait_busy();           /* 等待扇区擦除完成 */
}
```

该代码也是老套路，通过发送擦除指令实现擦除功能，要注意的是使用扇区擦除指令前，需要先发送写使能指令，拉低片选线，发送扇区擦除指令之后，发送擦除的扇区地址，实现擦除，最后拉高片选线结束通信。在函数最后通过读取寄存器状态的函数，等待扇区擦除完成。

3. main.c 代码

在 main.c 里面编写如下代码：

```
const uint8_t g_text_buf[] = {"STM32 SPI TEST"};   /* 要写到 FLASH 的字符串数组 */
#define TEXT_SIZE sizeof(g_text_buf)               /* TEXT 字符串长度 */

int main(void)
{
    uint8_t key;
    uint16_t i = 0;
    uint8_t datatemp[TEXT_SIZE];
    uint32_t flashsize;
    uint16_t id = 0;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usmart_dev.init(84);                    /* 初始化 USMART */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */
    norflash_init();                        /* 初始化 NORFLASH */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "SPI TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY1:Write  KEY0:Read", RED);

    id = norflash_read_id();                /* 读取 FLASH ID */
    while ((id == 0) || (id == 0xFFFF)) /* 检测不到 FLASH 芯片 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "FLASH Check Failed!", RED);
        delay_ms(500);
        lcd_show_string(30, 130, 200, 16, 16, "Please Check!      ", RED);
        delay_ms(500);
        LED0_TOGGLE();                      /* LED0 闪烁 */
    }
    lcd_show_string(30, 130, 200, 16, 16, "SPI FLASH Ready!", BLUE);
    flashsize = 16 * 1024 * 1024;           /* FLASH 大小为 16M 字节 */

    while (1)
    {
        key = key_scan(0);

        if (key == KEY1_PRES) /* KEY1 按下,写入 */
        {   /* 从倒数第 100 个地址处开始,写入 SIZE 长度的数据 */
            lcd_fill(0, 150, 239, 319, WHITE);   /* 清除半屏 */
            lcd_show_string(30, 150, 200, 16, 16, "Start Write FLASH....", BLUE);
            sprintf((char *)datatemp, "%s%d", (char *)g_text_buf, i);
            norflash_write((uint8_t *)datatemp, flashsize - 100, TEXT_SIZE);
            lcd_show_string(30, 150, 200, 16, 16, "FLASH Write Finished!", BLUE);
        }

        if (key == KEY0_PRES) /* KEY0 按下,读取字符串并显示 */
        {   /* 从倒数第 100 个地址处开始,读出 SIZE 个字节 */
            lcd_show_string(30, 150, 200, 16, 16, "Start Read FLASH...", BLUE);
            norflash_read(datatemp, flashsize - 100, TEXT_SIZE);
            lcd_show_string(30, 150, 200, 16, 16, "The Data Readed Is:   ", BLUE);
            lcd_show_string(30, 170, 200, 16, 16, (char *)datatemp, BLUE);
        }

        i++;
        if (i == 20)
```

```
        {
            LED0_TOGGLE(); /* LED0 闪烁 */
            i = 0;
        }
        delay_ms(10);
    }
}
```

在 main 函数前面，我们定义了 g_text_buf 数组，用于存放要写入到 FLASH 的字符串。main 函数代码和 IIC 实验那部分代码大同小异，具体流程大致是：在完成系统级和用户级初始化工作后，读取 FLASH 的 ID，然后通过 KEY0 去读取倒数第 100 个地址处开始的数据并把数据显示在 LCD 上；另外还可以通过 KEY1 去倒数第 100 个地址处写入 g_text_buf 数据并在 LCD 界面中显示传输中，完成后并显示“FLASH Write Finished!”。

27.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 27.4.1 所示：



图 27.4.1 SPI 实验程序运行效果图

通过先按下 KEY1 写入数据，然后再按 KEY0 读取数据，得到如图 27.4.2 所示：

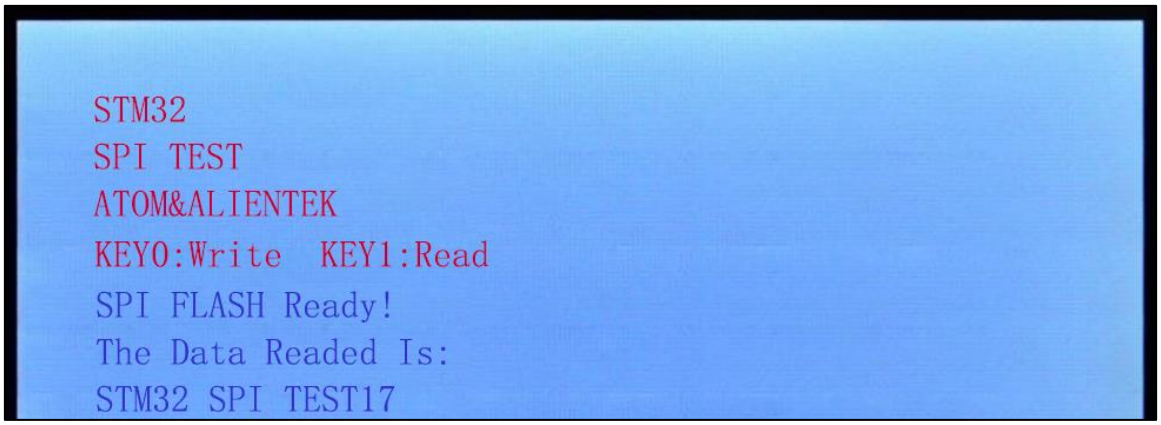


图 27.4.2 操作后的显示效果图

程序在开机的时候会检测 Norflash 是否存在，如果不存在则会在 TFTLCD 模块上显示错误信息，同时 LED0 慢闪。大家可以通过跳线帽把 PI2 和 PI3 短接就可以看到报错了。

该实验还支持 USMART，在这里我们加入了 norflash_read_id 和 norflash_erase_chip 以及 norflash_erase_sector 函数。可以通过 USMART 调用 norflash_read_id 函数去读取 SPI_FLASH 的 ID，也可以调用另外两个擦除函数。需要注意的是假如调用了 norflash_erase_chip 函数将会对整个 SPI_FLASH 进行擦除，一般情况不建议对整个 SPI_FLASH 进行擦除，因为会导致字库和综合例程所需要的系统文件全部丢失。

第二十八章 485 实验

本章我们将向大家介绍如何使用 STM32F407 的串口实现 485 通信（半双工）。在本章中，我们将使用 STM32F407 的串口 3 来实现两块开发板之间的 485 通信，并将结果显示在 TFTLCD 模块上。

本章分为如下几个部分：

28.1 485 简介
28.2 硬件设计
28.3 程序设计
28.4 下载验证

28.1 485 简介

485（一般称作 RS485/EIA-485）隶属于 OSI 模型物理层，是串行通讯的一种。电气特性规定为 2 线，半双工，多点通信的类型。它的电气特性和 RS-232 大不一样。用缆线两端的电压差值来表示传递信号。RS485 仅仅规定了接受端和发送端的电气特性。它没有规定或推荐任何数据协议。

RS485 的特点包括：

1. 接口电平低，不易损坏芯片。RS485 的电气特性：逻辑“1”以两线间的电压差为+(2~6)V 表示；逻辑“0”以两线间的电压差为-(2~6)V 表示。接口信号电平比 RS232 降低了，不易损坏接口电路的芯片，且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。
2. 传输速率高。10 米时，RS485 的数据最高传输速率可达 35Mbps，在 1200m 时，传输速度可达 100Kbps。
3. 抗干扰能力强。RS485 接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。
4. 传输距离远，支持节点多。RS485 总线最长可以传输 1200m 左右，更远的距离则需要中继传输设备支持（速率≤100Kbps 才能稳定传输），一般最大支持 32 个节点，如果使用特制的 485 芯片，可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

RS485 推荐使用在点对点网络中，比如：线型，总线型网络等，而不能是星型，环型网络。理想情况下 RS485 需要 2 个终端匹配电阻，其阻值要求等于传输电缆的特性阻抗（一般为 120Ω）。没有特性阻抗的话，当所有的设备都静止或者没有能量的时候就会产生噪声，而且线移需要双端的电压差。没有终端匹配电阻的话，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 推荐的一主多从连接方式如图 28.1.1 所示：

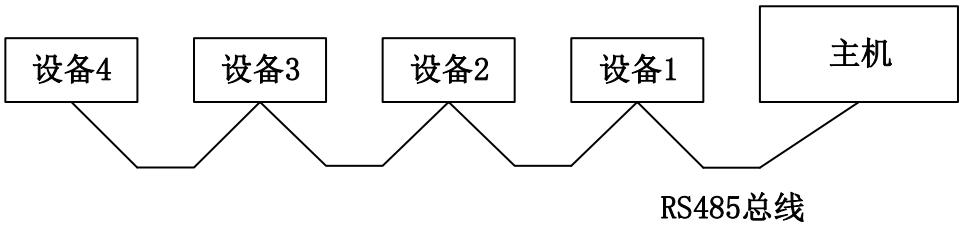


图 28.1.1 RS485 连接

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。实际多设备时收发器有范围为-7V 到+12V 的共模电压，为了稳定传输，也有使用 3 线的布线方式，即在原有的 A、B 两线上多增加一条地线。（4 线制只能实现点对点的全双工通讯方式，这种也叫 RS422，由于布线的难度和通讯局限，相对使用得比较少）。

TP8485E/SP3485 可作为 RS485 的收发器，该芯片支持 3.3V~5.5V 供电，最大传输速度可达 250Kbps，支持多达 256 个节点(单位负载为 1/8 的条件下)，并且支持输出短路保护。该芯片

的框图如图 28.1.2 所示：

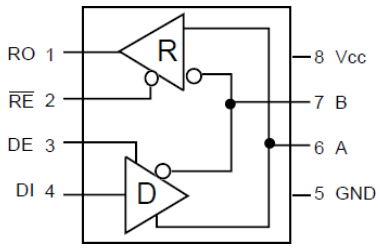


图 28.1.2 TP8485E/SP3485 框图

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）。

28.2 硬件设计

1. 例程功能

经过前面的学习我们知道实际的 RS485 仍是串行通讯的一种电平传输方式，那么我们实际通讯时可以使用串口进行实际数据的收发处理，使用 485 转换芯片将串口信号转换为 485 的电平信号进行传输，本章，我们只需要配置好串口 3，就可以实现正常的 485 通信了，串口 3 的配置和串口 1 基本类似，只是串口 3 的时钟来自 APB1，最大频率为 42Mhz。

本章将实现这样的功能：通过连接两个电机开发板的 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。

2. 硬件资源

1）LED 灯

LED0 – PE0

2）USART3，用于 485 信号串行通讯。

3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4）RS485 收发芯片 TP8485/SP3485

5）开发板两块（485 半双式模式无法自收发，我们需要用两个开发板或者 USB 转 485 调试器+串口助手来帮助我们完成测试，大家根据自己的实际条件选择）

3. 原理图

根据我们需要实现的程序功能，我们设计电路原理如下：

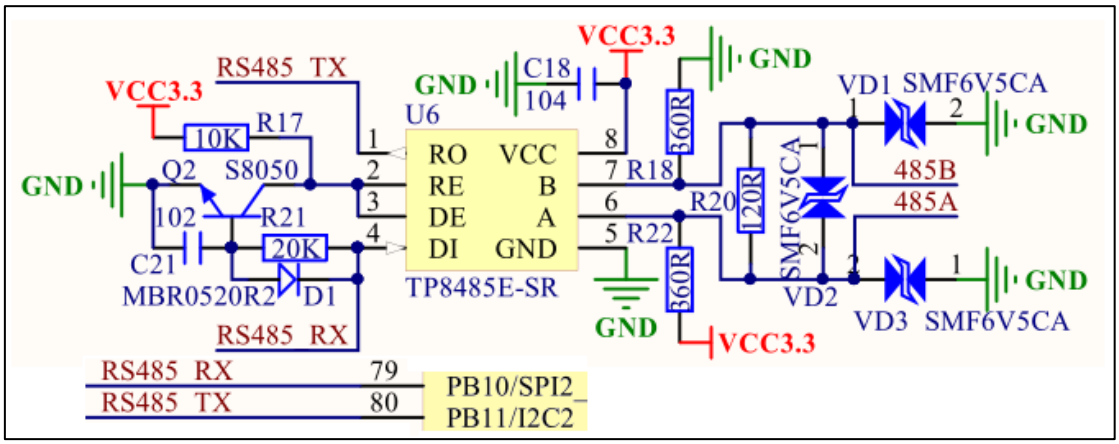


图 28.2.2 RS485 连接原理设计

从上图可以看出：开发板的串口 3（PB10、PB11）连接在 TP8485 上，485A 和 485B 是接线端子，图中的 R18 和 R22 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因 A、B 压差不稳定，可能出现的乱码。我们只需要用 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!!

28.3 程序设计

28.3.1 RS485 的 HAL 库驱动

由于 485 实际上是串口通讯，我们参照串口实验一节使用类似的 HAL 库驱动即可，在这里分析一下 RS485 配置步骤。

RS485 配置步骤

1）使能串口和 GPIO 口时钟

本实验用到 USART3 口，使用 PB10 和 PB11 作为串口的 TX 和 RX 脚，因此需要先使能 USART3 和 GPIOB 时钟。参考代码如下：

```
__HAL_RCC_USART3_CLK_ENABLE();          /* 使能 USART3 时钟 */
__HAL_RCC_GPIOB_CLK_ENABLE();           /* 使能 GPIOB 时钟 */
```

2) 串口参数初始化（波特率、字长、奇偶校验等）

HAL 库通过调用串口初始化函数 HAL_UART_Init 完成对串口参数初始化，详见例程源码。

该函数通常会调用：HAL_UART_MspInit 函数来完成对串口底层的初始化，包括：串口及 GPIO 时钟使能、GPIO 模式设置、中断设置等。但是本实验避免与 USART1 冲突，所以把串口底层初始化没有放在 HAL_UART_MspInit 函数里。

3）GPIO 模式设置（速度，上下拉，复用功能等）

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

4）开启串口相关中断，配置串口中断优先级

本实验我们使用串口中断来接收数据。我们使用 HAL_UART_Receive_IT 函数开启串口中断接收，并设置接收 buffer 及其长度。通过 HAL_NVIC_EnableIRQ 函数使能串口中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5）编写中断服务函数

串口 3 中断服务函数为：USART3_IRQHandler，当发生中断的时候，程序就会执行中断服务函数，在这里就可以对接收到的数据进行处理，详见本例程源码。

6）串口数据接收和发送

最后我们可以通过读写 USART_DR 寄存器，完成串口数据的接收和发送，HAL 库也给我们提供了：HAL_UART_Receive 和 HAL_UART_Transmit 两个函数用于串口数据的接收和发送。

大家可以根据实际情况选择使用哪种方式来收发串口数据。

28.3.2 程序流程图

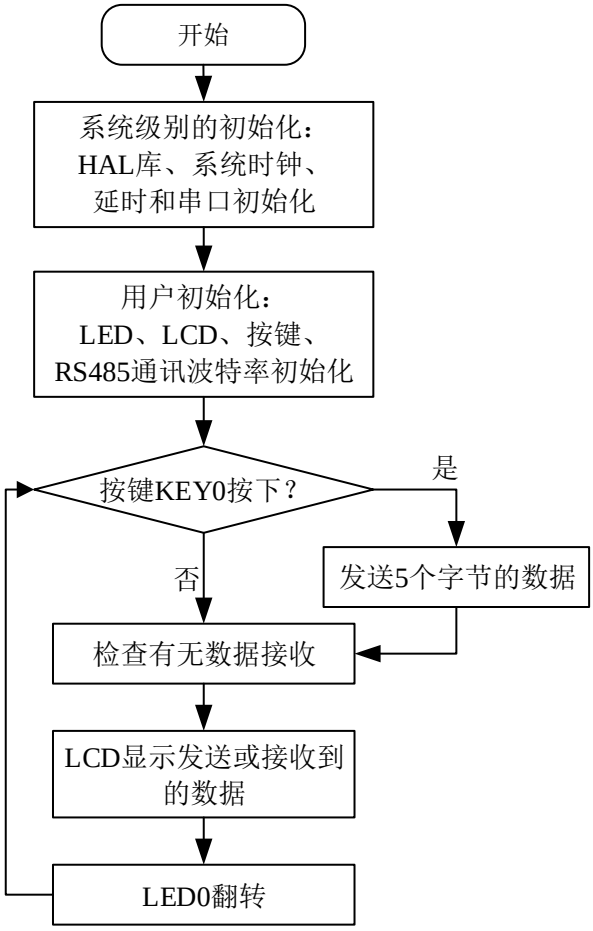


图 28.3.2.1 RS485 实验程序流程图

28.3.3 程序解析

1. RS485 驱动

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。RS485 驱动相关源码包括两个文件：rs485.c 和 rs485.h。

为方便修改，我们在 rs485.h 中使用宏定义 485 相关的控制引脚和串口编号，如果需要使用其它的引脚或者串口，修改宏和串口的定义即可，它们在 rs485.h 中定义，它们列出如下：

```
/* RS485 引脚 和 串口 定义 */
#define RS485_TX_GPIO_PORT          GPIOB
#define RS485_TX_GPIO_PIN           GPIO_PIN_10
#define RS485_TX_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define RS485_RX_GPIO_PORT          GPIOB
#define RS485_RX_GPIO_PIN           GPIO_PIN_11
#define RS485_RX_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define RS485_UX                    USART3
#define RS485_UX_IRQn               USART3_IRQn
#define RS485_UX_IRQHandler         USART3_IRQHandler
#define RS485_UX_CLK_ENABLE()       do{ __HAL_RCC_USART3_CLK_ENABLE(); }while(0)
```

1）rs485_init 函数

rs485_init 的配置与串口类似，也需要设置波特率等参数，另外还需要配置收发模式的驱动引脚，我们的程序设计如下：

```
/**
 * @brief       RS485 初始化函数
 *   @note      该函数主要是初始化串口
 * @param       baudrate: 波特率, 根据自己需要设置波特率值
 * @retval      无
 */
void rs485_init(uint32_t baudrate)
{
    /* IO 及 时钟配置 */
    RS485_TX_GPIO_CLK_ENABLE(); /* 使能 串口 TX 脚 时钟 */
    RS485_RX_GPIO_CLK_ENABLE(); /* 使能 串口 RX 脚 时钟 */
    RS485_UX_CLK_ENABLE();      /* 使能 串口 时钟 */

    GPIO_InitTypeDef gpio_init_struct;

    gpio_init_struct.Pin = RS485_TX_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(RS485_TX_GPIO_PORT, &gpio_init_struct);  /* 串口 TX 脚 模式设置 */

    gpio_init_struct.Pin = RS485_RX_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_INPUT;
    /* 串口 RX 脚设置成输入模式 */
    HAL_GPIO_Init(RS485_RX_GPIO_PORT, &gpio_init_struct);

    /* USART 初始化设置 */
    g_rs458_handler.Instance = RS485_UX;                    /* 选择 485 对应的串口 */
    g_rs458_handler.Init.BaudRate = baudrate;               /* 波特率 */
    g_rs458_handler.Init.WordLength = UART_WORDLENGTH_8B;   /* 字长为 8 位数据格式 */
    g_rs458_handler.Init.StopBits = UART_STOPBITS_1;        /* 一个停止位 */
    g_rs458_handler.Init.Parity = UART_PARITY_NONE;         /* 无奇偶校验位 */
    g_rs458_handler.Init.HwFlowCtl = UART_HWCONTROL_NONE;   /* 无硬件流控 */
    g_rs458_handler.Init.Mode = UART_MODE_TX_RX;            /* 收发模式 */
    HAL_UART_Init(&g_rs458_handler);                        /* 使能对应的串口 */
    __HAL_UART_DISABLE_IT(&g_rs458_handler, UART_IT_TC);

    __HAL_UART_ENABLE_IT(&g_rs458_handler, UART_IT_RXNE);   /* 开启接收中断 */
    HAL_NVIC_EnableIRQ(RS485_UX_IRQn);                      /* 使能 USART3 中断 */
    HAL_NVIC_SetPriority(RS485_UX_IRQn, 3, 3);          /* 抢占优先级 3，子优先级 3 */

    RS485_RE(0);    /* 默认为接收模式 */
}
```

可以看到代码基本跟串口的配置一样，只是多了收发控制引脚的配置。

2）发送函数

发送函数用于输出 485 信号到 485 总线上，默认的 485 方式一般空闲时为接收状态，只有发送数据时我们才控制 485 芯片进入发送状态，发送完成后马上回到空闲接收状态，这样可以保证操作过程中 485 的数据丢失最小。我们实现的发送函数如下：

```
/**
 * @brief       RS485 发送 len 个字节
 * @param       buf : 发送区首地址
 * @param       len : 发送字节数(为了和本代码接收匹配,这里不要超过 RS485_REC_LEN 个字节)
 * @retval      无
 */
void rs485_send_data(uint8_t *buf, uint8_t len)
{
    HAL_UART_Transmit(&g_rs458_handler, buf, len, 1000);    /* 串口 3 发送数据 */
    g_RS485_rx_cnt = 0;
}
```

3）485 接收中断函数

RS485 的接收与串口中断一样，不过要注意空闲时要切换回接收状态，否则会收不到数据。我们定义了一个全局的缓冲区 g_RS485_rx_buf 进行接收测试，通过串口中断接收数据，编写的接收代码如下：

```
uint8_t g_RS485_rx_buf[RS485_REC_LEN]; /* 接收缓冲, 最大 RS485_REC_LEN 个字节. */
uint8_t g_RS485_rx_cnt = 0;            /* 接收到的数据长度 */

void RS485_UX_IRQHandler(void)
{
    uint8_t res;

    if ((__HAL_UART_GET_FLAG(&g_rs458_handler, UART_FLAG_RXNE) != RESET))
    {   /* 接收到数据 */
        HAL_UART_Receive(&g_rs458_handler, &res, 1, 1000);

        if (g_RS485_rx_cnt < RS485_REC_LEN)          /* 缓冲区未满 */
        {
            g_RS485_rx_buf[g_RS485_rx_cnt] = res;    /* 记录接收到的值 */
            g_RS485_rx_cnt++;                        /* 接收数据增加 1 */
        }
    }
}
```

4）485 查询接收数据函数

该函数用于查询 485 总线上接收到的数据，主要实现的逻辑是：一开始进入函数时，先记录下当前接收计数器的值，再来一个延时去判断接收是否结束（即该期间有无接收到数据），假如说接收计数器的值没有改变，就证明接收结束，我们就可以把当前接收缓冲区传递出去。函数实现如下：

```
void rs485_receive_data(uint8_t *buf, uint8_t *len)
{
    uint8_t rxlen = g_RS485_rx_cnt;
    uint8_t 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 == g_RS485_rx_cnt && rxlen) /* 接收到了数据,且接收完成了 */
    {
        for (i = 0; i < rxlen; i++)
        {
            buf[i] = g_RS485_rx_buf[i];
        }

        *len = g_RS485_rx_cnt;   /* 记录本次数据长度 */
        g_RS485_rx_cnt = 0;      /* 清零 */
    }
}
```

RS485 的代码就讲到这里，基本是串口的知识，大家不明白的配置可以翻看之前串口章节的知识。

2. main.c 代码

在 main.c 中编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint8_t i = 0, t = 0;
    uint8_t cnt = 0;
    uint8_t rs485buf[5];

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
```

```
    usart_init(115200);                     /* 串口初始化为 115200 */
    usmart_dev.init(84);                    /* 初始化 USMART */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */
    rs485_init(9600);                       /* 初始化 RS485 */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "RS485 TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Send", RED);    /* 显示提示信息 */

    lcd_show_string(30, 130, 200, 16, 16, "Count:", RED);       /* 显示当前计数值 */
    lcd_show_string(30, 150, 200, 16, 16, "Send Data:", RED);   /* 提示发送的数据 */
    lcd_show_string(30, 190, 200, 16, 16, "Receive Data:", RED);/*提示收到的数据*/

    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 */
                /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 170, rs485buf[i], 3, 16, 0x80, BLUE);
            }
            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++)
            {   /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 210, rs485buf[i], 3, 16, 0x80, BLUE);
            }
        }

        t++;
        delay_ms(10);

        if (t == 20)
        {
            LED0_TOGGLE();  /* LED0 闪烁, 提示系统正在运行 */
            t = 0;
            cnt++;
            lcd_show_xnum(30 + 48, 130, cnt, 3, 16, 0x80, BLUE);    /* 显示数据 */
        }
    }
}
```

我们是通过按键控制数据的发送。在此部分代码中，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 总线收到数据得时候，就将收到的数据直接显示在 LCD 屏幕上。

28.4 下载验证

在代码编译成功之后，将代码下载到两块开发板，程序运行效果如图 28.4.1 所示：

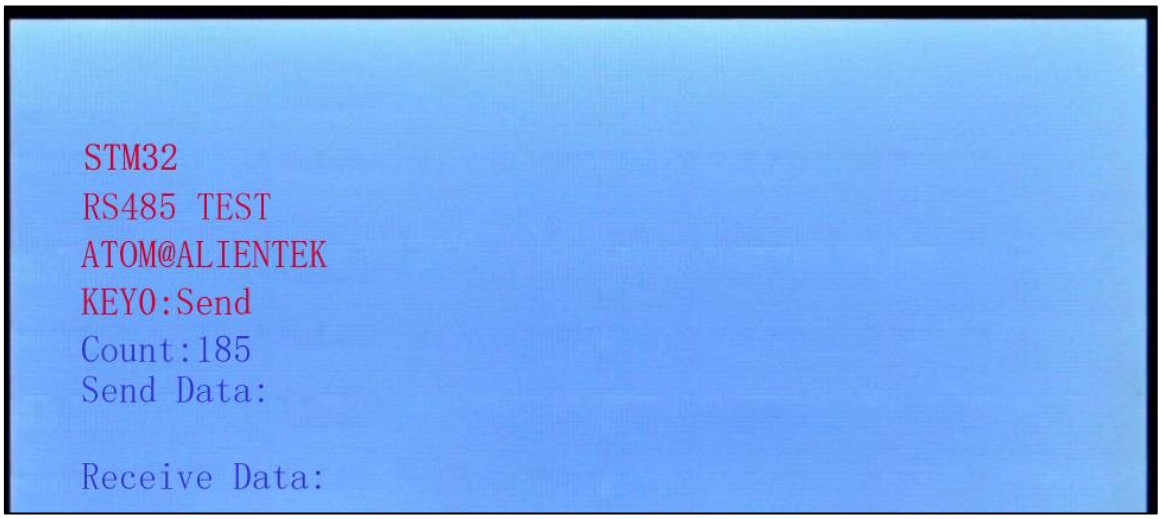


图 28.4.1 程序运行效果图

LED0 不停闪烁，提示程序在运行。此时，按下 KEY0，在另外一个开发板上面收到这个开发板发送的数据。如图 28.4.2 和图 28.4.3 所示：

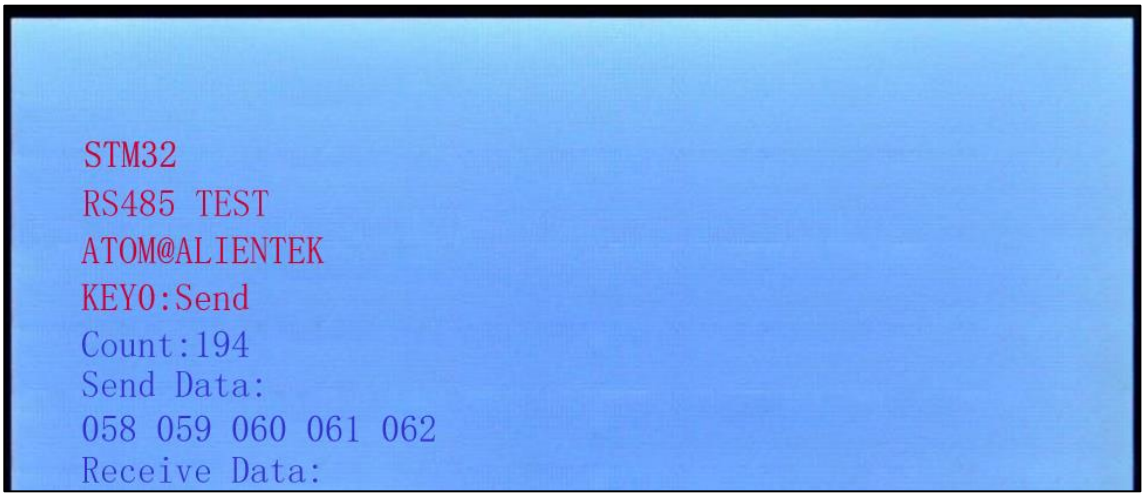


图 28.4.2 发送 RS485 数据的开发板界面

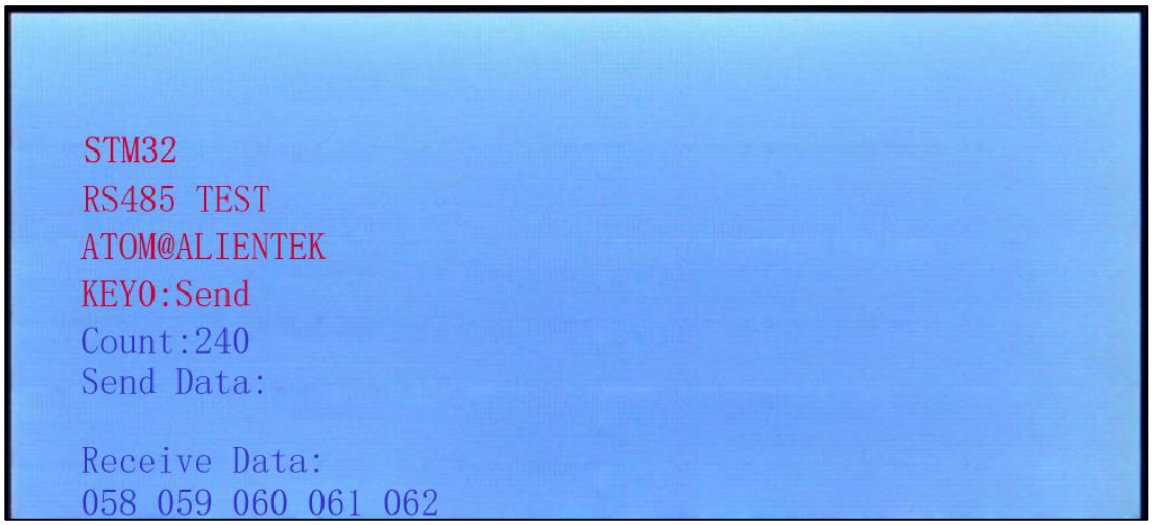


图 28.4.3 接收 RS485 数据的开发板

图 28.4.2 来自开发板 A，发送了 5 个数据，图 28.4.3 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，只需要连接两块开发板 RS485 的 A 和 B 就可以进行数据传输了。

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。

第二十九章 232 实验

在本章，我们将通过 USB 转 232 连接电机开发板实现与 PC 端的通信，并将发送与接收到的数据显示在 TFTLCD 模块上。

本章分为如下几个部分：

29.1 232 简介
29.2 硬件设计
29.3 程序设计
29.4 下载验证

29.1 232 简介

RS-232 接口符合美国电子工业联盟（EIA）制定的串行数据通信的接口标准，原始编号全称是 EIA-RS-232（简称 232，RS232）。它被广泛用于计算机串行接口外设连接。连接电缆和机械、电气特性、信号功能及传送过程。RS232 的特点包括：

① 接口的信号电平值较高，易损坏接口电路的芯片。RS232 接口任何一条信号线的电压均为负逻辑关系，即：逻辑“1”为-3—-15V；逻辑“0”：+3—+15V ，噪声容限为 2V。即要求接收器能识别高于+3V 的信号作为逻辑“0”，低于-3V 的信号作为逻辑“1”，TTL 电平为 5V 为逻辑正，0 为逻辑负 。与 TTL 电平不兼容故需使用电平转换电路方能与 TTL 电路连接。

② RS-232 规定的标准传送速率有 50b/s、75b/s、110b/s、150b/s、300b/s、600b/s、1200b/s、2400b/s、4800b/s、9600b/s、19200b/s，可以灵活地适应不同速率的设备。

③ RS-232 总线规定了 25 条线，包含了两个信号通道，即第一通道（称为主通道）和第二通道（称为副通道）。利用 RS- 232 总线可以实现全双工通信，通常使用的是主通道，而副通道使用较少。在一般应用中，使用 3 条～9 条信号线就可以实现全双工通信，采用三条信号线（接收线、发送线和信号线）能实现简单的全双工通信过程。

④ RS -232 采用串行传送方式，将 TTL 电平转换为 RS-232C 电平，其传送距离一般可达 30 m。若采用光电隔离 20 mA 的电流环进行传送，其传送距离可以达到 1000 m。另外，如果在 RS-232 总线接口再加上 Modem，通过有线、无线或光纤进行传送，其传输距离可以更远。

29.2 硬件设计

1. 例程功能

经过前面的学习我们知道实际的 RS232 仍是串行通讯的一种电平传输方式，那么我们实际通讯时可以使用串口进行实际数据的收发处理，使用 232 转换芯片将串口信号转换为 232 的电平信号进行传输，本章，我们只需要配置好串口 5，就可以实现正常的 232 通信了，串口 5 的配置和串口 1 基本类似，只是串口 5 的时钟来自 APB1，最大频率为 42Mhz。

本章将实现这样的功能：通过 USB 转 232 连接电机开发板的 DB9 接口，按下 KEY0，发送 5 个数据给串口调试助手，并在开发板上显示发送和接收到的值，也可以通过 USMART 调用 rs232_send_data 函数，实现发送指定的数据。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）UART5，用于 232 信号串行通讯。
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）RS232 收发芯片 SP3232
5）USB 转 232 串口线

3. 原理图

根据我们需要实现的程序功能，我们设计电路原理如下：

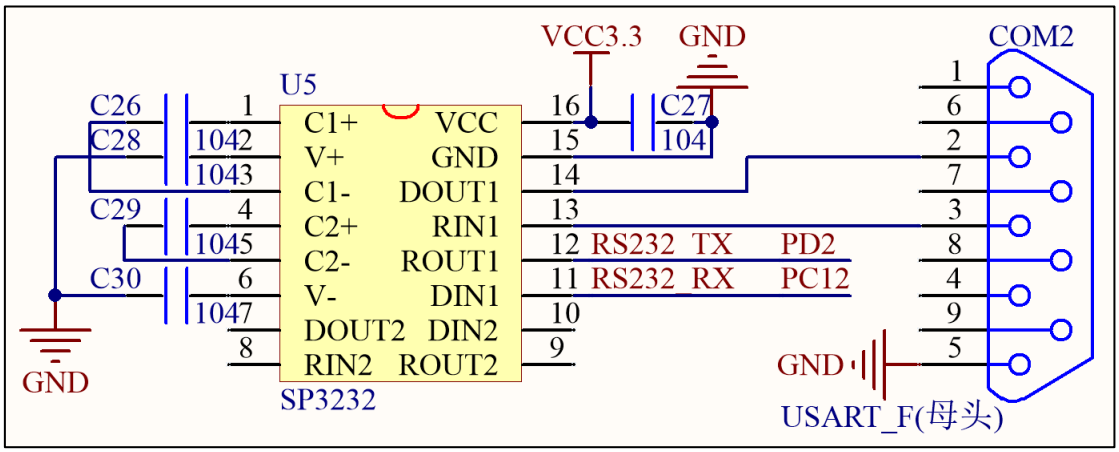


图 29.2.2 RS232 连接原理设计

从上图可以看出：电机开发板的串口 5（PC12、PD2）连接在 SP3232 上，COM2 是 DB9 母头，我们只需要将 USB 转 232 接在 COM2 就可以实现 PC 与电机开发板的通信。

29.3 程序设计

29.3.1 RS232 的 HAL 库驱动

由于 232 实际上是串口通讯，我们参照串口实验一节使用类似的 HAL 库驱动即可，在这里分析一下 RS232 配置步骤。

RS232 配置步骤

1）使能串口和 GPIO 口时钟

本实验用到 UART5 口，使用 PC12 和 PD2 作为串口的 TX 和 RX 脚，因此需要先使能 UART5 和 GPIOC、GPIOD 时钟。参考代码如下：

```
__HAL_RCC_UART5_CLK_ENABLE();           /* 使能 UART5 时钟 */
__HAL_RCC_GPIOC_CLK_ENABLE();           /* 使能 GPIOC 时钟 */
__HAL_RCC_GPIOD_CLK_ENABLE();           /* 使能 GPIOD 时钟 */
```

2）串口参数初始化（波特率、字长、奇偶校验等）

HAL 库通过调用串口初始化函数 HAL_UART_Init 完成对串口参数初始化，详见例程源码。

该函数通常会调用：HAL_UART_MspInit 函数来完成对串口底层的初始化，包括：串口及 GPIO 时钟使能、GPIO 模式设置、中断设置等。但是本实验避免与 USART1 冲突，所以把串口底层初始化没有放在 HAL_UART_MspInit 函数里。

3）GPIO 模式设置（速度，上下拉，复用功能等）

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

4）开启串口相关中断，配置串口中断优先级

本实验我们使用串口中断来接收数据。我们使用 HAL_UART_Receive_IT 函数开启串口中断接收，并设置接收 buffer 及其长度。通过 HAL_NVIC_EnableIRQ 函数使能串口中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5）编写中断服务函数

串口 5 中断服务函数为：UART5_IRQHandler，当发生中断的时候，程序就会执行中断服务函数，在这里就可以对接收到的数据进行处理，详见本例程源码。

6）串口数据接收和发送

最后我们可以通过读写 UART_DR 寄存器，完成串口数据的接收和发送，HAL 库也给我们提供了：HAL_UART_Receive 和 HAL_UART_Transmit 两个函数用于串口数据的接收和发送。

大家可以根据实际情况选择使用哪种方式来收发串口数据。

29.3.2 程序流程图

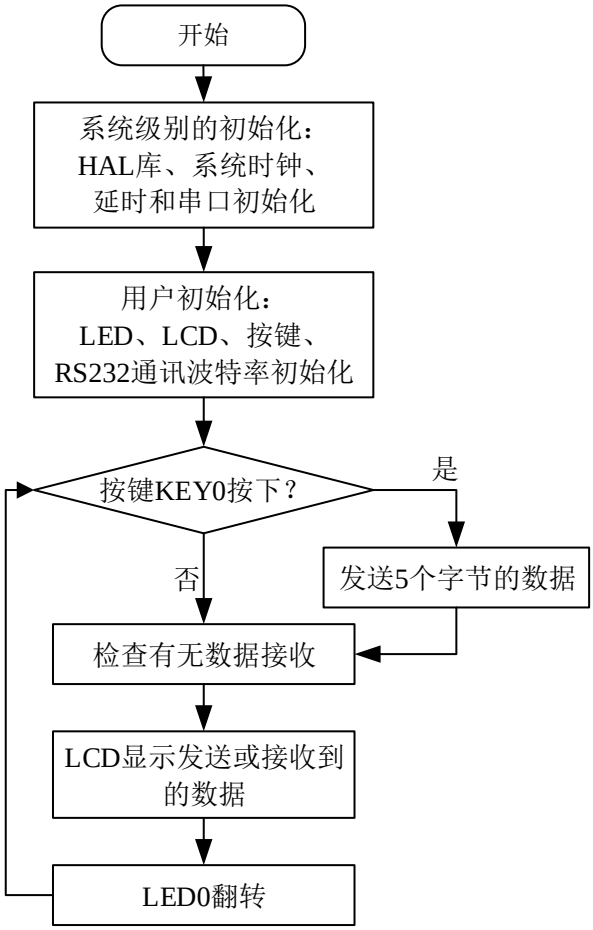


图 29.3.2.1 RS232 实验程序流程图

29.3.3 程序解析

1. RS232 驱动

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。RS232 驱动相关源码包括两个文件：rs232.c 和 rs232.h。

为方便修改，我们在 rs232.h 中使用宏定义 232 相关的控制引脚和串口编号，如果需要使用其它的引脚或者串口，修改宏和串口的定义即可，它们在 rs232.h 中定义，它们列出如下：

```
/* RS232 引脚 和 串口 定义 */
#define RS232_TX_GPIO_PORT          GPIOC
#define RS232_TX_GPIO_PIN           GPIO_PIN_12
#define RS232_TX_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)

#define RS232_RX_GPIO_PORT          GPIOD
#define RS232_RX_GPIO_PIN           GPIO_PIN_2
#define RS232_RX_GPIO_CLK_ENABLE()  do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)

#define RS232_UX                    UART5
#define RS232_UX_IRQn               UART5_IRQn
#define RS232_UX_IRQHandler         UART5_IRQHandler
#define RS232_UX_CLK_ENABLE()       do{ __HAL_RCC_UART5_CLK_ENABLE(); }while(0)
```

1）rs232_init 函数

rs232_init 的配置与串口类似，也需要设置波特率等参数，另外还需要配置收发模式的驱

动引脚，我们的程序设计如下：

```
/**
 * @brief       RS232 初始化函数
 *   @note      该函数主要是初始化串口
 * @param       baudrate: 波特率, 根据自己需要设置波特率值
 * @retval      无
 */
void rs232_init(uint32_t baudrate)
{
    /* IO 及 时钟配置 */
    RS232_TX_GPIO_CLK_ENABLE(); /* 使能 串口 TX 脚 时钟 */
    RS232_RX_GPIO_CLK_ENABLE(); /* 使能 串口 RX 脚 时钟 */
    RS232_UX_CLK_ENABLE();      /* 使能 串口 时钟 */

    GPIO_InitTypeDef gpio_init_struct;
    gpio_init_struct.Pin = RS232_TX_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;
    gpio_init_struct.Pull = GPIO_PULLUP;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(RS232_TX_GPIO_PORT, &gpio_init_struct);  /* 串口 TX 脚 模式设置 */

    gpio_init_struct.Pin = RS232_RX_GPIO_PIN;
    HAL_GPIO_Init(RS232_RX_GPIO_PORT, &gpio_init_struct);  /* 串口 RX 脚 模式设置 */

    /* USART 初始化设置 */
    g_rs323_handler.Instance = RS232_UX;                    /* 选择 232 对应的串口 */
    g_rs323_handler.Init.BaudRate = baudrate;               /* 波特率 */
    g_rs323_handler.Init.WordLength = UART_WORDLENGTH_8B;   /* 字长为 8 位数据格式 */
    g_rs323_handler.Init.StopBits = UART_STOPBITS_1;        /* 一个停止位 */
    g_rs323_handler.Init.Parity = UART_PARITY_NONE;         /* 无奇偶校验位 */
    g_rs323_handler.Init.HwFlowCtl = UART_HWCONTROL_NONE;   /* 无硬件流控 */
    g_rs323_handler.Init.Mode = UART_MODE_TX_RX;            /* 收发模式 */
    HAL_UART_Init(&g_rs232_handler);                        /* 使能对应的串口 */
    __HAL_UART_DISABLE_IT(&g_rs232_handler, UART_IT_TC);

#if RS232_EN_RX /* 如果使能了接收 */
    __HAL_UART_ENABLE_IT(&g_rs232_handler, UART_IT_RXNE);   /* 开启接收中断 */
    HAL_NVIC_EnableIRQ(RS232_UX_IRQn);            /* 使能 RS232 中断 */
    HAL_NVIC_SetPriority(RS232_UX_IRQn, 3, 3);    /* 抢占优先级 3, 子优先级 3 */
#endif
}
```

可以看到代码基本跟串口的配置一样，只是多了收发控制引脚的配置。

2）发送函数

发送函数用于输出 232 信号到 232 总线上，默认的 232 方式一般空闲时为接收状态，只有发送数据时我们才控制 232 芯片进入发送状态，发送完成后马上回到空闲接收状态，这样可以保证操作过程中 232 的数据丢失最小。我们实现的发送函数如下：

```
/**
 * @brief   RS232 发送 len 个字节
 * @param   buf : 发送区首地址
 * @param   len : 发送字节数(为了和本代码接收匹配,这里不要超过 RS232_REC_LEN 个字节)
 * @retval  无
 */
void rs232_send_data(uint8_t *buf, uint8_t len)
{
    HAL_UART_Transmit(&g_rs232_handler, buf, len, 1000);    /* 串口 5 发送数据 */
    g_RS232_rx_cnt = 0;
}
```

3）232 接收中断函数

RS232 的接收与串口中断一样，不过要注意空闲时要切换回接收状态，否则会收不到数据。

我们定义了一个全局的缓冲区 g_RS232_rx_buf 进行接收测试，通过串口中断接收数据，编写的接收代码如下：

```
uint8_t g_RS232_rx_buf[RS232_REC_LEN]; /* 接收缓冲, 最大 RS232_REC_LEN 个字节. */
uint8_t g_RS232_rx_cnt = 0;            /* 接收到的数据长度 */

void RS232_UX_IRQHandler(void)
{
    uint8_t res;

    if ((__HAL_UART_GET_FLAG(&g_RS232_handler, UART_FLAG_RXNE) != RESET))
    {   /* 接收到数据 */
        HAL_UART_Receive(&g_RS232_handler, &res, 1, 1000);

        if (g_RS232_rx_cnt <RS232_REC_LEN)          /* 缓冲区未满 */
        {
            g_RS232_rx_buf[g_RS232_rx_cnt] = res;   /* 记录接收到的值 */
            g_RS232_rx_cnt++;                       /* 接收数据增加 1 */
        }
    }
}
```

4）232 查询接收数据函数

该函数用于查询 232 总线上接收到的数据，主要实现的逻辑是：一开始进入函数时，先记录下当前接收计数器的值，再来一个延时去判断接收是否结束（即该期间有无接收到数据），假如说接收计数器的值没有改变，就证明接收结束，我们就可以把当前接收缓冲区传递出去。函数实现如下：

```
void rs232_receive_data(uint8_t *buf, uint8_t *len)
{
    uint8_t rxlen = g_RS232_rx_cnt;
    uint8_t 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 == g_RS232_rx_cnt && rxlen) /* 接收到了数据,且接收完成了 */
    {
        for (i = 0; i < rxlen; i++)
        {
            buf[i] = g_RS232_rx_buf[i];
        }

        *len = g_RS232_rx_cnt;   /* 记录本次数据长度 */
        g_RS232_rx_cnt = 0;      /* 清零 */
    }
}
```

RS232 的代码就讲到这里，基本是串口的知识，大家不明白的配置可以翻看之前串口章节的知识。

2. main.c 代码

在 main.c 中编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint8_t i = 0, t = 0;
    uint8_t cnt = 0;
    uint8_t rs232buf[5];

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    usmart_dev.init(84);                    /* 初始化 USMART */
```

```
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    rs232_init(9600);                           /* 初始化 RS232 */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "RS232 TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Send", RED);    /* 显示提示信息 */

    lcd_show_string(30, 130, 200, 16, 16, "Count:", RED);       /* 显示当前计数值 */
    lcd_show_string(30, 150, 200, 16, 16, "Send Data:", RED);   /* 提示发送的数据 */
    lcd_show_string(30, 190, 200, 16, 16, "Receive Data:", RED);/*提示收到的数据*/

    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++)
            {
                rs232buf[i] = cnt + i;          /* 填充发送缓冲区 */
                /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 170, rs232buf[i], 3, 16, 0x80, BLUE);
            }
            rs232_send_data(rs232buf, 5);       /* 发送 5 个字节 */
        }
        rs232_receive_data(rs232buf, &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++)
            {   /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 210, rs232buf[i], 3, 16, 0x80, BLUE);
            }
        }

        t++;
        delay_ms(10);

        if (t == 20)
        {
            LED0_TOGGLE();  /* LED0 闪烁, 提示系统正在运行 */
            t = 0;
            cnt++;
            lcd_show_xnum(30 + 48, 130, cnt, 3, 16, 0x80, BLUE);    /* 显示数据 */
        }
    }
}
```

我们是通过按键控制数据的发送。在此部分代码中，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 232 总线收到数据的时候，就将收到的数据直接显示在 LCD 屏幕上。

29.4 下载验证

在代码编译成功之后，将代码下载到电机开发板，程序运行效果如图 29.4.1 所示：

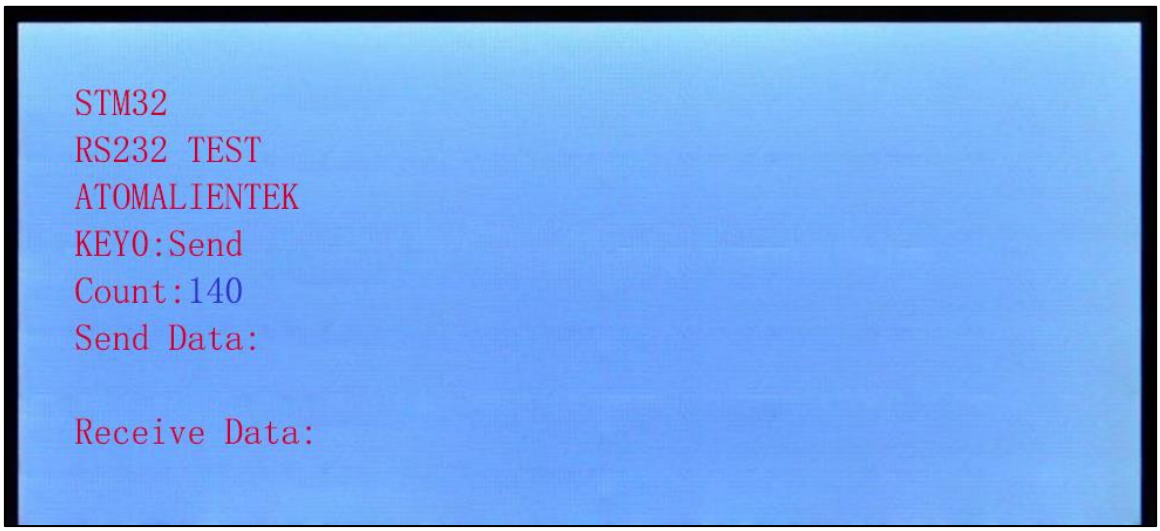


图 29.4.1 程序运行效果图

LED0 不停闪烁，提示程序在运行。此时，按下 KEY0，在 XCOM 上位机上面收到这个开发板发送的数据。如图 29.4.2 和图 29.4.3 所示：

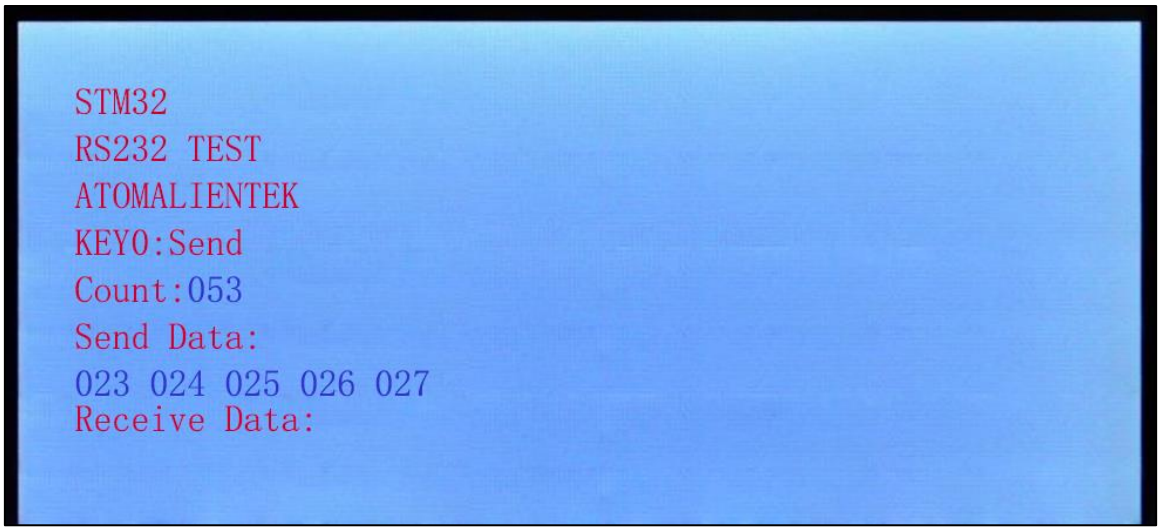


图 29.4.2 发送 RS232 数据的开发板界面



图 29.4.3 接收 RS232 数据的开发板

图 29.4.2，开发板向串口调试助手发送了 5 个数据（十进制），图 29.4.3，XCOM 上位机接收到了来自串口调试助手的 5 个数据（十六进制），XCOM 上位机也可以发送 5 个数据（十六进制），开发板显示接收到的 5 个数据（十进制），注意：如果使用 RS232 接收数据功能，必须使能 RS232 接收中断。

本章介绍的 232 总线时通过串口控制收发的，只需要通过 USB 转 232 连接开发板和电脑就可以进行数据传输了。

另外，利用 USMART 测试的部分，我们这里就不做介绍了，大家可自行验证下。

第三十章 CAN 通讯实验

本章我们将向大家介绍如何使用 STM32 自带的 CAN 控制器来实现 CAN 的收发功能，并将结果显示在 TFTLCD 模块上。本章分为如下几个部分：

30.1 CAN 简介：
30.2 硬件设计
30.3 程序设计
30.4 下载验证

30.1 CAN 总线简介

30.1.1 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 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（高速 CAN）和 ISO11519-2 标准（低速 CAN）。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 ISO11898 标准，也就是高速 CAN，其拓扑图如图 30.1.1.1 所示。

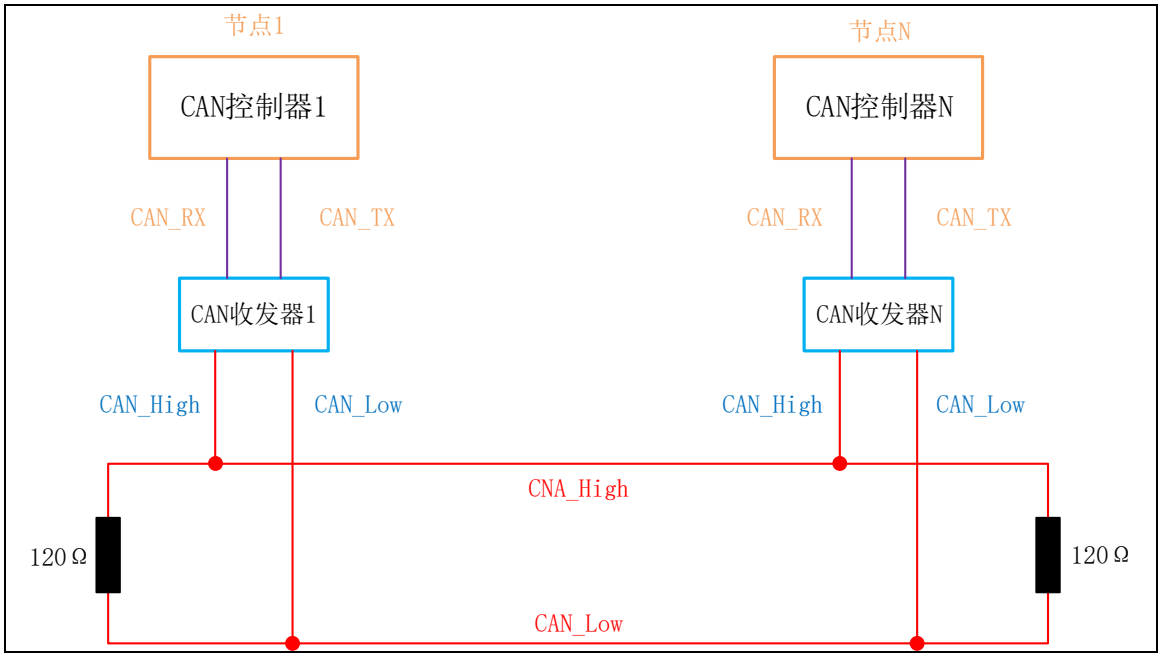


图 30.1.1 高速 CAN 拓扑结构图

从上图可知，高速 CAN 总线呈现的是一个闭环结构，总线是由两根线 CAN_High 和 CAN_Low 组成，且在总线两端各串联了 120Ω 的电阻（用于阻抗匹配，减少回波反射），同时总线上可以挂载多个节点。每个节点都有 CAN 收发器以及 CAN 控制器，CAN 控制器通常是 MCU 的外设，集成在芯片内部；CAN 收发器则是需要外加芯片转换电路。

CAN 类似 RS485 也是通过差分信号传输数据。根据 CAN 总线上两根线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。这是属于物理层特征，ISO11898 物理层特性如图 30.1.1.2 所示：

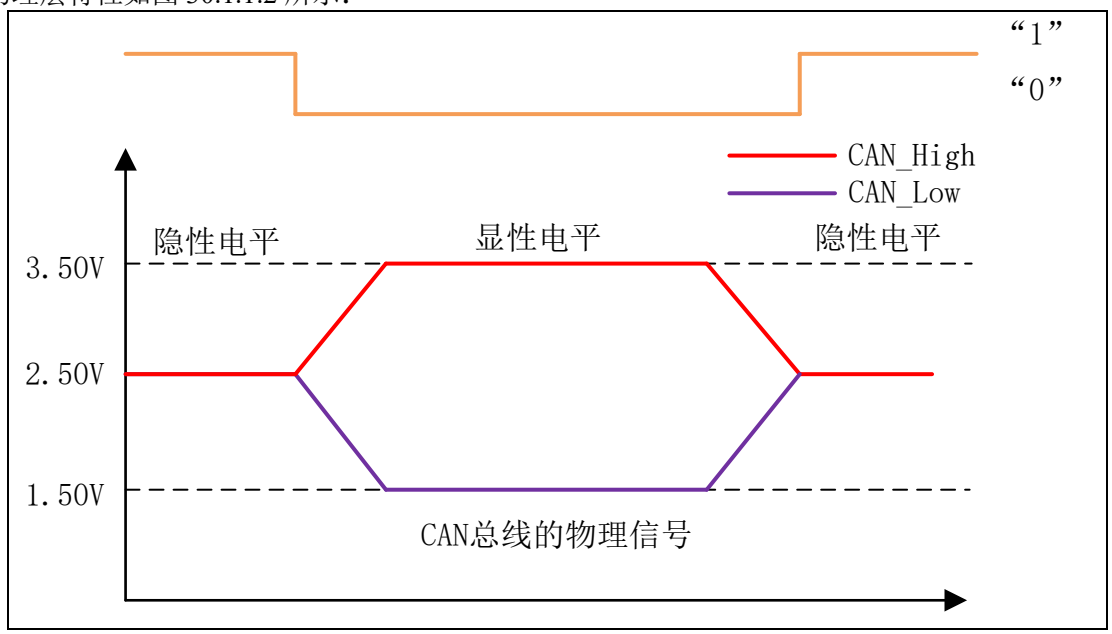


图 30.1.1.2 ISO11898 物理层特性

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 V 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）。

30.1.2 CAN 协议

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 29.1.2.1 所示：

帧类型	帧用途
数据帧	用于发送单元向接收单元传送数据的帧
遥控帧	用于接收单元向具有相同ID的发送单元请求数据的帧
错误帧	用于当检测出错误时向其它单元通知错误的帧
过载帧	用于接收单元通知其尚未做好接收准备的帧
间隔帧	用于将数据帧及遥控帧与前面的帧分离开来的帧

表 30.1.2.1 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：

帧起始。表示数据帧开始的段。
仲裁段。表示该帧优先级的段。
控制段。表示数据的字节数及保留位的段。
数据段。数据的内容，一帧可发送 0~8 个字节的数据。
CRC 段。检查帧的传输错误的段。
ACK 段。表示确认正常接收的段。
帧结束。表示数据帧结束的段。

数据帧的构成如图 30.1.2.1 所示：

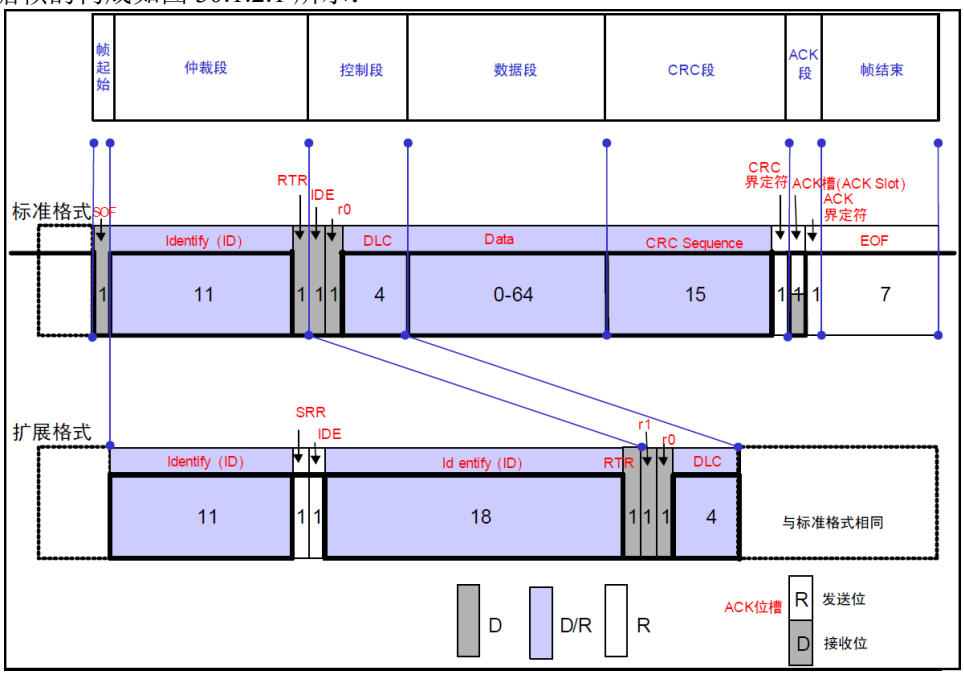


图 30.1.2.1 数据帧的构成图中 D 表示显性电平，R 表示隐形电平（下同）。

帧起始，这个比较简单，标准帧和扩展帧都是由 1 个位的显性电平表示帧起始。

仲裁段，表示数据优先级的段，标准帧和扩展帧格式在本段有所区别，如图 30.1.2.2 所示：

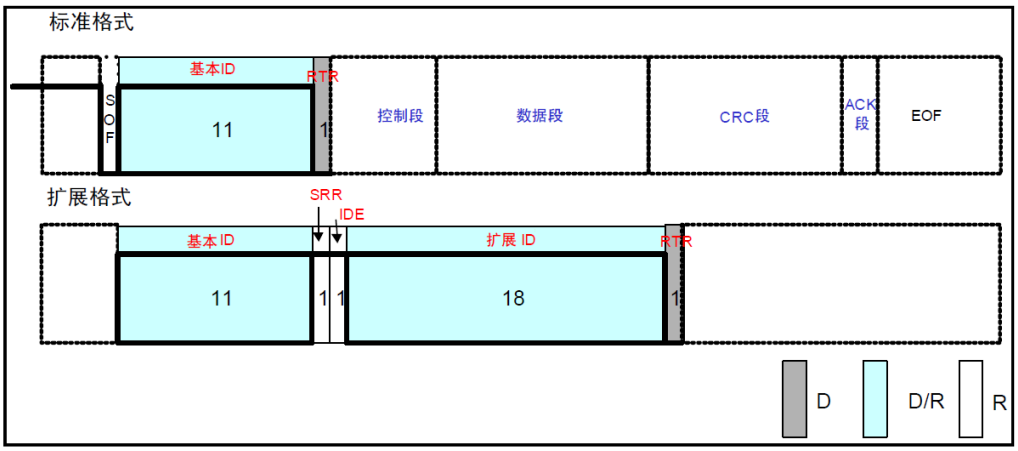


图 30.1.2.2 数据帧仲裁段构成

标准格式的 ID 有 11 个位。禁止高 7 位都为隐性（禁止设定：ID=1111111XXXX）。扩展格式的 ID 有 29 个位。基本 ID 从 ID28 到 ID18，扩展 ID 由 ID17 到 ID0 表示。基本 ID 和标准格式的 ID 相同。禁止高 7 位都为隐性（禁止设定：基本 ID=1111111XXXX）。

其中 RTR 位用于标识是否是远程帧（0，数据帧；1，远程帧），IDE 位为标识符选择位（0:使用标准标识符；1:使用扩展标识符），SRR 位为代替远程请求位，为隐性位，它代替了标准帧中的 RTR 位。

控制段，由 6 个位构成，表示数据段的字节数。标准帧和扩展帧的控制段稍有不同，如图 30.1.2.3 所示：

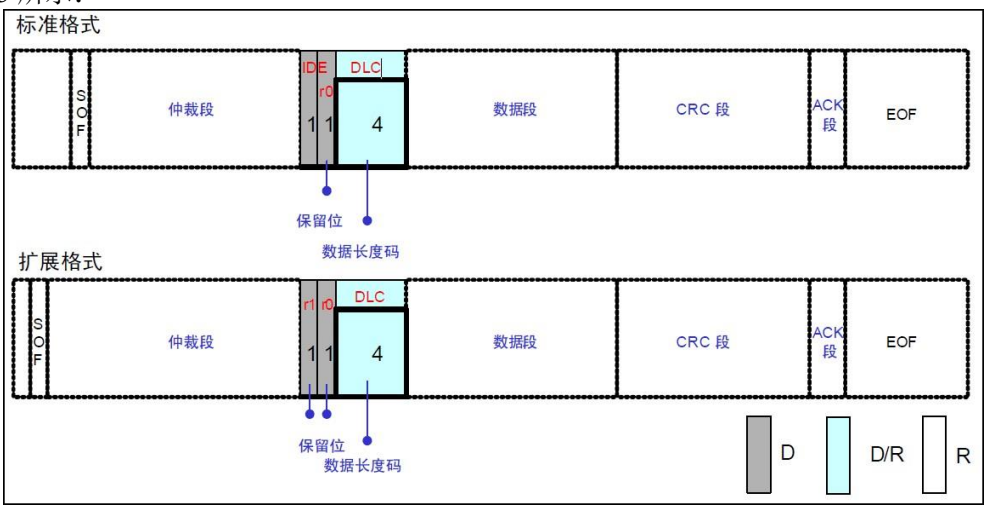


图 30.1.2.3 数据帧控制段构成

上图中，r0 和 r1 为保留位，必须全部以显性电平发送，但是接收端可以接收显性、隐性及任意组合的电平。DLC 段为数据长度表示段，高位在前，DLC 段有效值为 0~8，但是接收方接收到 9~15 的时候并不认为是错误。

数据段，该段可包含 0~8 个字节的数据。从最高位（MSB）开始输出，标准帧和扩展帧在这个段的定义都是一样的。如图 30.1.2.4 所示：

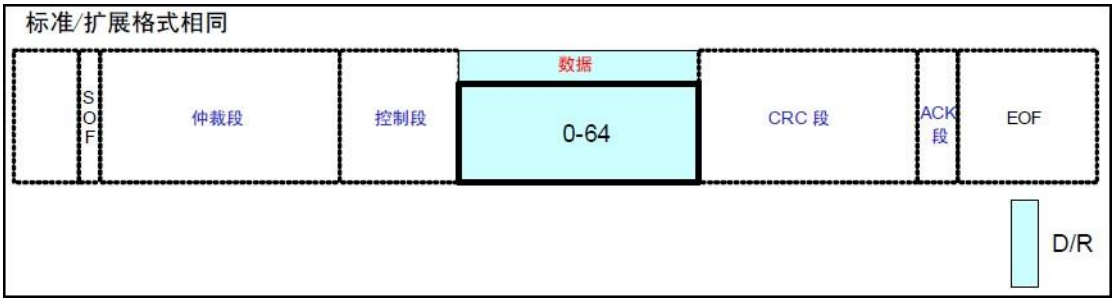


图 30.1.2.4 数据帧数据段构成

CRC 段，该段用于检查帧传输错误。由 15 个位的 CRC 顺序和 1 个位的 CRC 界定符（用于分隔的位）组成，标准帧和扩展帧在这个段的格式也是相同的。如图 30.1.2.5 所示：

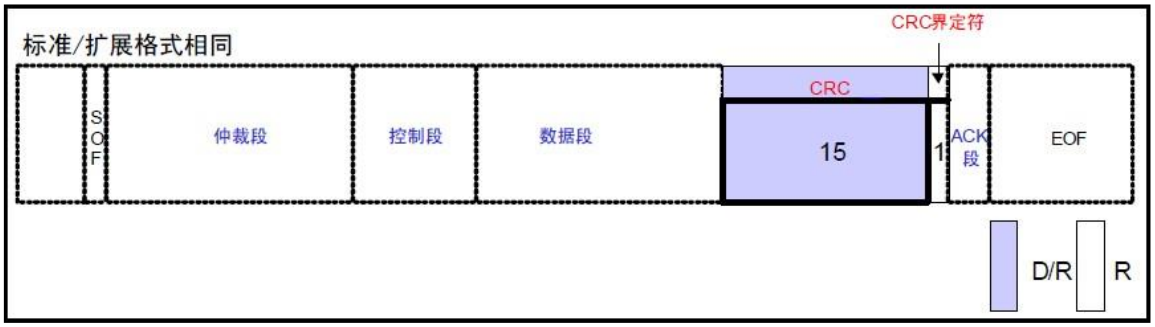


图 30.1.2.5 数据帧 CRC 段构成

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(ACKSlot)和 ACK 界定符 2 个位组成。标准帧和扩展帧在这个段的格式也是相同的。如图 30.1.2.6 所示：

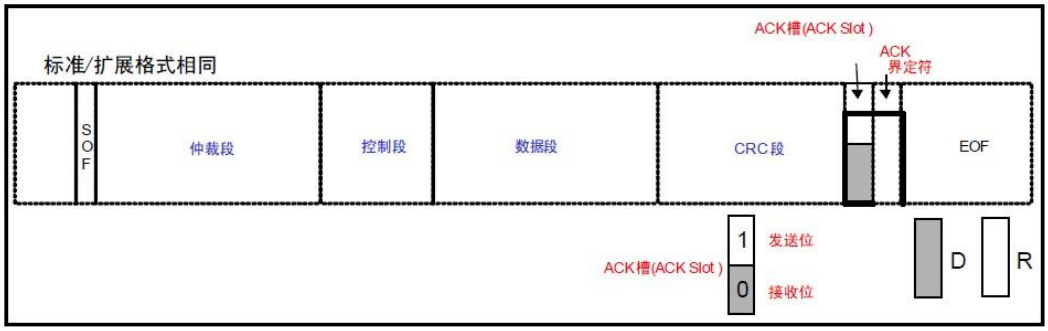


图 30.1.2.6 数据帧 CRC 段构成

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（ACKSlot）发送显性位，通知发送单元正常接收结束，这个过程叫发送 ACK/返回 ACK。发送 ACK 的是在既不处于总线关闭态也不处于休眠态的所有接收单元中，接收到正常消息的单元（发送单元不发送 ACK）。所谓正常消息是指不含填充错误、格式错误、CRC 错误的消息。

帧结束，这个段也比较简单，标准帧和扩展帧在这个段格式一样，由 7 个位的隐性位组成。至此，数据帧的 7 个段就介绍完了，其他帧的介绍，请大家参考光盘的《CAN 入门书.pdf》相关章节。接下来，我们再来看看 CAN 的位时序。

由发送单元在非同步的情况下发送的每秒钟的位数称为位速率。一个位可分为 4 段。

- 同步段（SS）
- 传播时间段（PTS）
- 相位缓冲段 1（PBS1）
- 相位缓冲段 2（PBS2）

这些段又由可称为 Time Quantum（以下称为 Tq）的最小时间单位构成。

1 位分为 4 个段，每个段又由若干个 Tq 构成，这称为位时序。

1 位由多少个 Tq 构成、每个段又由多少个 Tq 构成等，可以任意设定位时序。通过设定位时序，多个单元可同时采样，也可任意设定采样点。各段的作用和 Tq 数如表 30.1.2.2 所示：

段名称	段的作用	Tq 数	
同步段 （SS: Synchronization Segment）	多个连接在总线上的单元通过此段实现时序调整，同步进行接收和发送的工作。由隐性电平到显性电平的边沿或由显性电平到隐性电平边沿最好出现在此段中。	1Tq	8～25Tq
传播时间段 （PTS: Propagation Time Segment）	用于吸收网络上的物理延迟的段。 所谓的网络的物理延迟指发送单元的输出延迟、总线上信号的传播延迟、接收单元的输入延迟。 这个段的时间为以上各延迟时间的和的两倍。	1～8Tq	
相位缓冲段 1 (PBS1: Phase Buffer Segment 1)	当信号边沿不能被包含于 SS 段中时，可在此段进行补偿。 由于各单元以各自独立的时钟工作，细微的时钟误差会累积起来，PBS 段可用于吸收此误差。 通过对相位缓冲段加减 SJW 吸收误差。 SJW 加大后允许误差加大，但通信速度下降。	1～8Tq	
相位缓冲段 2 (PBS2: Phase Buffer Segment 2)		2～8Tq	
再同步补偿宽度 (SJW: reSynchronization Jump Width)	因时钟频率偏差、传送延迟等，各单元有同步误差。SJW 为补偿此误差的最大值。	1～4Tq	

表 30.1.2.2 一个位各段及其作用

1 个位的构成如图 30.1.2.7 所示：

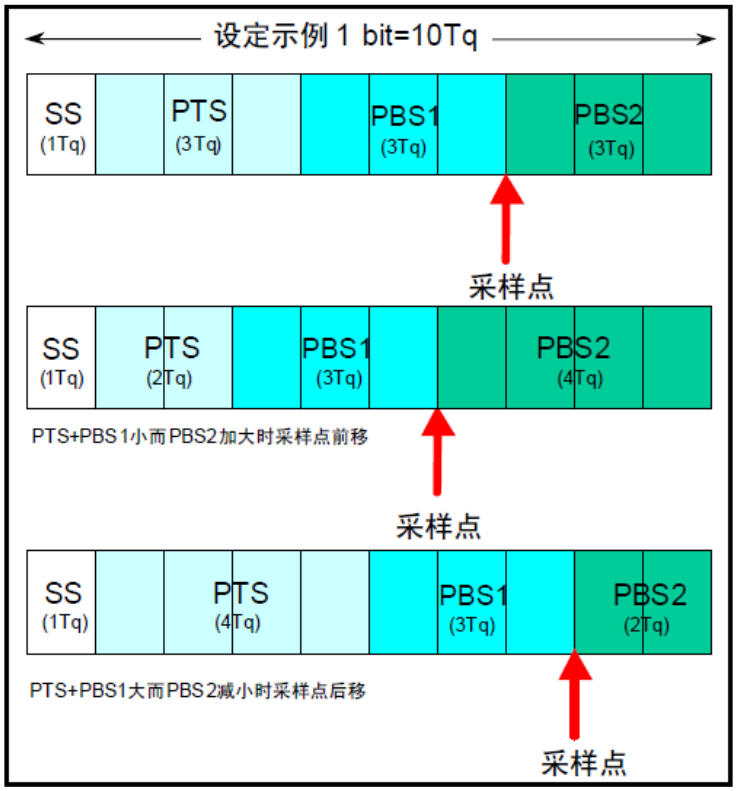


图 30.1.2.7 一个位的构成

上图的采样点，是指读取总线电平，并将读到的电平作为位值的点。位置在 PBS1 结束处。根据这个位时序，我们就可以计算 CAN 通信的波特率了。具体计算方法，我们等下再介绍，前面提到的 CAN 协议具有仲裁功能，下面我们来看看是如何实现的。

在总线空闲态，最先开始发送消息的单元获得发送权。当多个单元同时开始发送时，各发送单元从仲裁段的第一位开始进行仲裁。连续输出显性电平最多的单元可继续发送。实现过程，如图 30.1.2.8 所示：

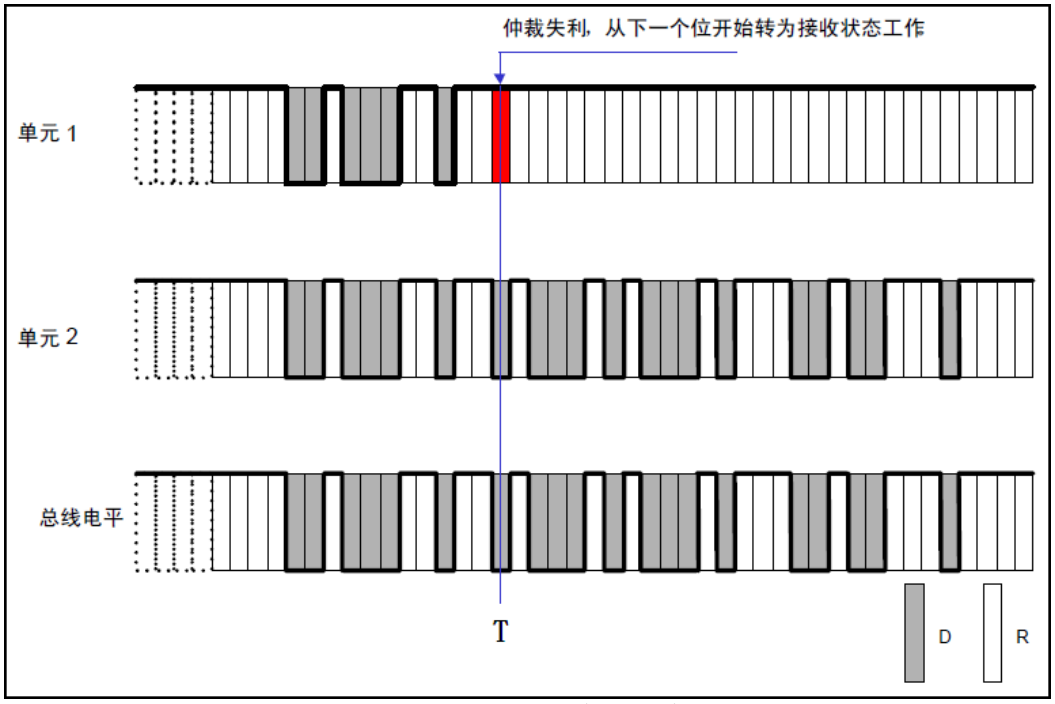


图 30.1.2.8 CAN 总线仲裁过程

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 STM32F407 的 CAN 控制器。

STM32F407 自带的是 bxCAN，即基本扩展 CAN。它支持 CAN 协议 2.0A 和 2.0B。CAN2.0A 只能处理标准数据帧，扩展帧的内容会识别错误；CAN2.0B Active 可以处理标准数据帧和扩展数据帧；而 CAN2.0B passive 只能处理标准数据帧，扩展帧的内容会忽略。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。

STM32F407 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 级深度的接收 FIFO
- 可变的过滤器组（最多 28 个）

在 STM32F407IGT6 中，带有 2 个 CAN 控制器，而我们本章只用了一个 CAN，即 CAN1。双 CAN 的框图如图 30.1.2.9 所示：

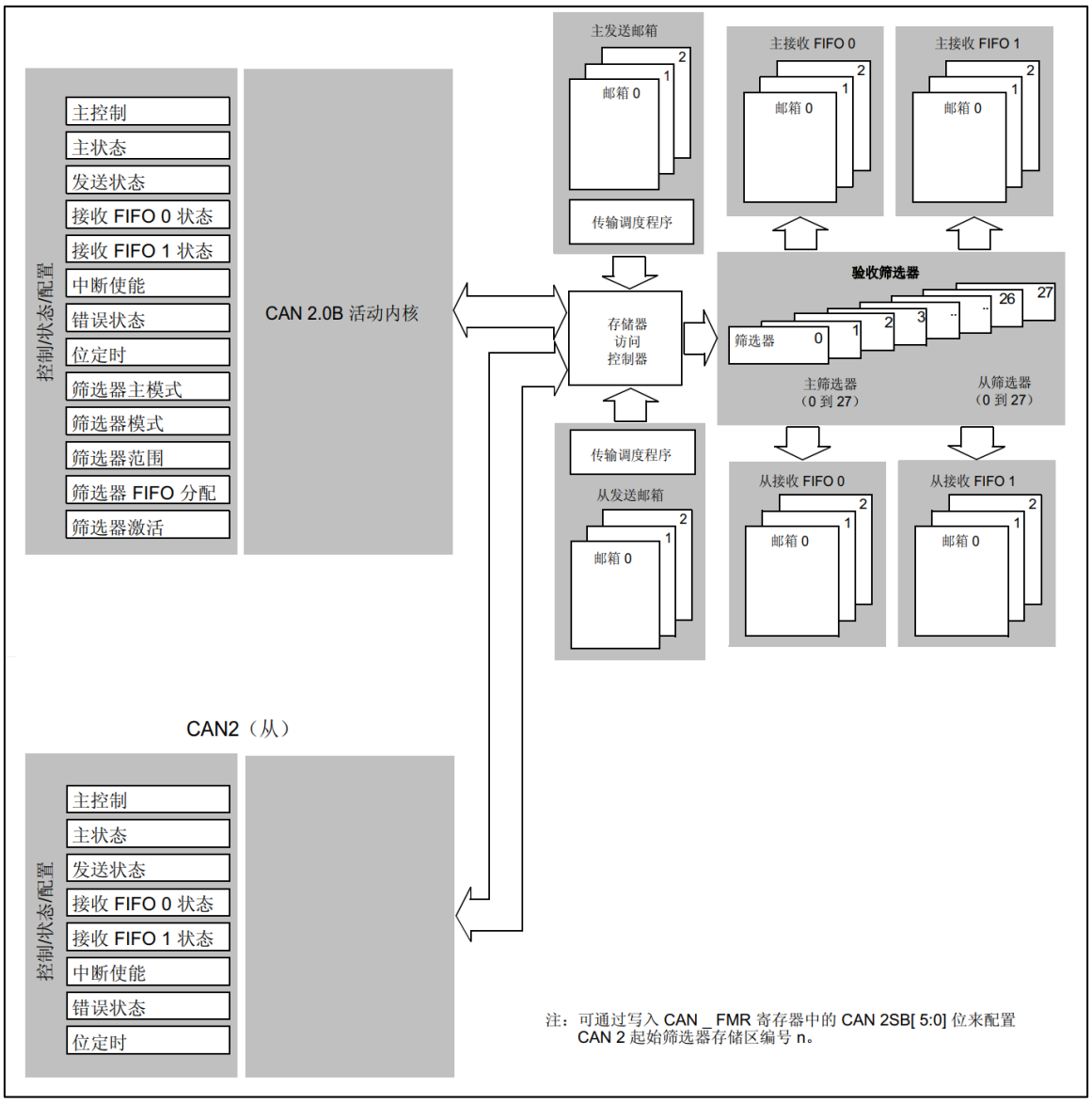


图 30.1.2.9 双 CAN 框图

从图中可以看出两个 CAN 都分别拥有自己的发送邮箱和接收 FIFO，但是他们共用 28 个滤波器。通过 CAN_FMR 寄存器的设置，可以设置滤波器的分配方式。

STM32 的标识符过滤比较复杂，它的存在减少了 CPU 处理 CAN 通信的开销。STM32F407 的过滤器（也称筛选器）组最多有 28 个（互联型），每个滤波器组 x 由 2 个 32 为寄存器，CAN_FxR1 和 CAN_FxR2 组成。

STM32 每个过滤器组的位宽都可以独立配置，以满足应用程序的不同需求。根据位宽的不同，每个过滤器组可提供：

- 1 个 32 位过滤器，包括：STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位
- 2 个 16 位过滤器，包括：STDID[10:0]、IDE、RTR 和 EXTID[17:15]位

此外过滤器可配置为，屏蔽位模式和标识符列表模式。

在屏蔽位模式下，标识符寄存器和屏蔽寄存器一起，指定报文标识符的任何一位，应该按照“必须匹配”或“不用关心”处理。

而在标识符列表模式下，屏蔽寄存器也被当作标识符寄存器用。因此，不是采用一个标识符加一个屏蔽位的方式，而是使用 2 个标识符寄存器。接收报文标识符的每一位都必须跟过滤器标识符相同。

通过 CAN_FMR 寄存器，可以配置过滤器组的位宽和工作模式，如图 30.1.2.10 所示：

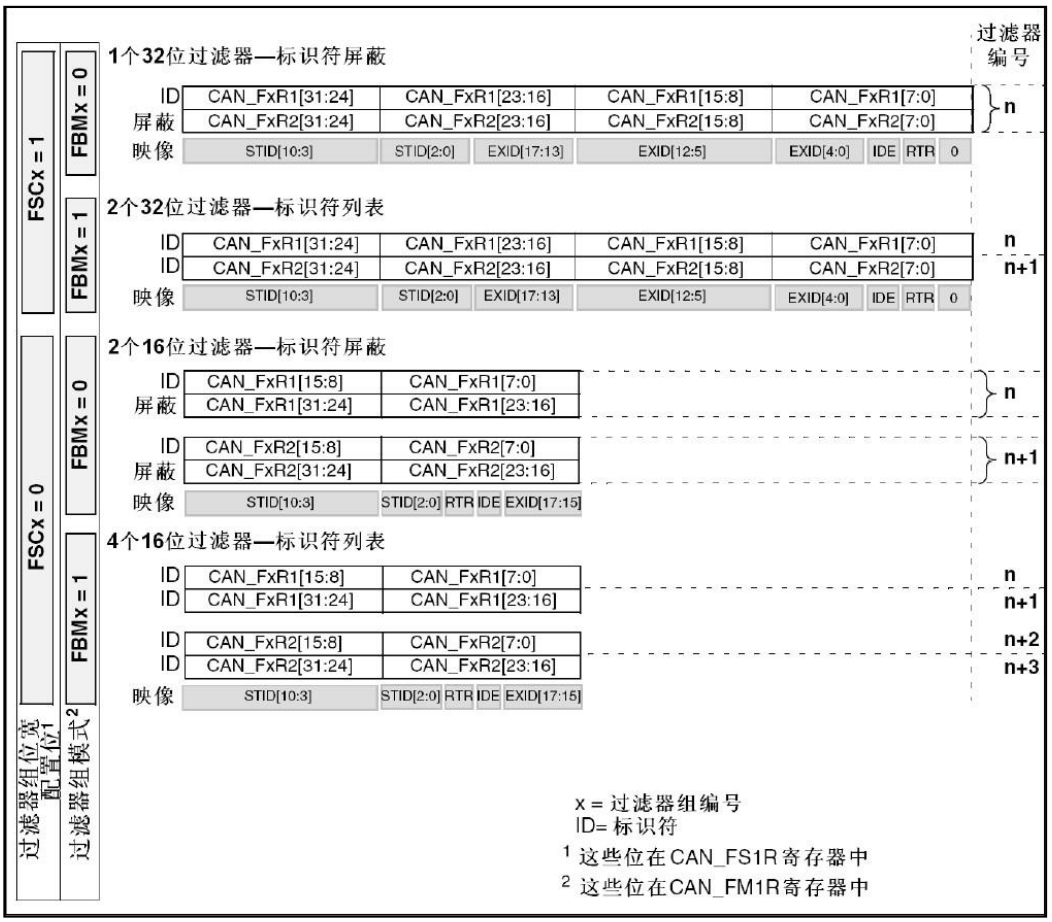


图 30.1.2.10 过滤器组位宽模式设置

为了过滤出一组标识符，应该设置过滤器组工作在屏蔽位模式。

为了过滤出一个标识符，应该设置过滤器组工作在标识符列表模式。应用程序不用的过滤器组，应该保持在禁用状态。

过滤器组中的每个过滤器，都被编号为(叫做过滤器号，图 30.1.2.10 中的 n)从 0 开始，到某个最大数值－取决于过滤器组的模式和位宽的设置。

举个简单的例子，我们设置过滤器组 0 工作在：1 个 32 为位过滤器-标识符屏蔽模式，然后设置 CAN_F0R1=0xFFFF0000，CAN_F0R2=0xFF00FF00。其中存放到 CAN_F0R1 的值就是期望收到的 ID，即我们希望收到的映像（STID+EXTID+IDE+RTR）最好是：0xFFFF0000。而 0xFF00FF00 就是设置我们需要必须关心的 ID，表示收到的映像，其位[31:24]和位[15:8]这 16 个位的必须和 CAN_F0R1 中对应的位一模一样，而另外的 16 个位则不关心，可以一样，也可以不一样，都认为是正确的 ID，即收到的映像必须是 0xFFxx00xx，才算是正确的（x 表示不关心）。这里需要注意的是标识符选择位 IDE 和帧类型 RTR 需要一致。其情况如下图 30.1.2.11 所示：

32位过滤器-标识符屏蔽模式（过滤出一组标识符）																														
bit	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
ID CAN_F0R1 (0xFFFF0000)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
屏蔽 CAN_F0R2 (0xFF00FF00)	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
映像	STID[10:3]								STID[2:0]			EXID[17:13]					EXID[12:5]								EXID[4:0]					IDE	RTR	0
过滤出ID	1	1	1	1	1	1	1	1	X	X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	X	X	X	X	X	0	0	0

图 30.1.2.11 过滤器举例图

关于标识符过滤的详细介绍，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》的 24.7.4 节（616 页）。接下来，我们看看 STM32 的 CAN 发送和接收的流程。

CAN 发送流程

CAN 发送流程为：程序选择 1 个空置的邮箱（TME=1） 设置标识符（ID），数据长度和发送数据 设置 CAN_TIxR 的 TXRQ 位为 1，请求发送 邮箱挂号（等待成为最高优先级） 预定发送（等待总线空闲） 发送 邮箱空置。整个流程如图 30.1.2.12 所示：

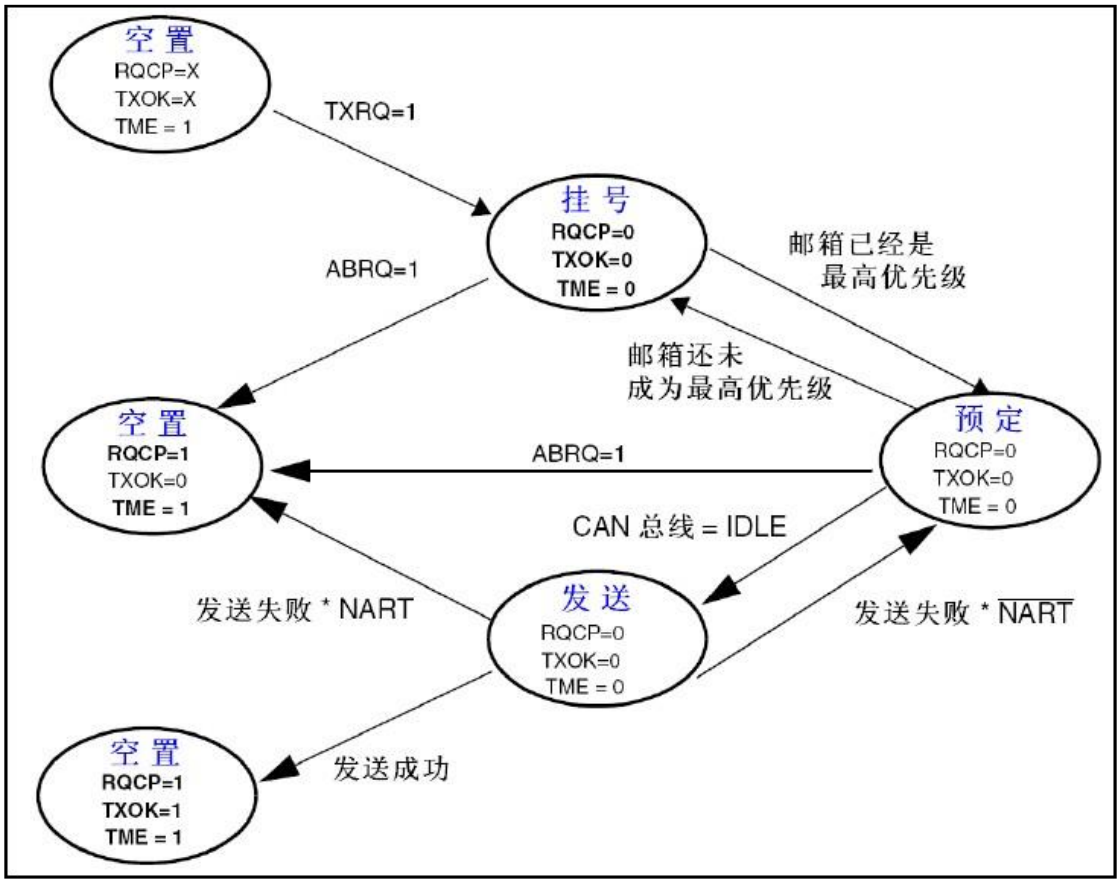


图 30.1.2.12 发送邮箱

上图中，还包含了很多其他处理，不强制退出发送（ABRQ=1）和发送失败处理等。通过这个流程图，我们大致了解了 CAN 的发送流程，后面的数据发送，我们基本就是按照此流程来走。接下来再看看 CAN 的接收流程。

CAN 接收流程

CAN 接收到的有效报文，被存储在 3 级邮箱深度的 FIFO 中。FIFO 完全由硬件来管理，从而节省了 CPU 的处理负荷，简化了软件并保证了数据的一致性。应用程序只能通过读取 FIFO 输出邮箱，来读取 FIFO 中最先收到的报文。这里的有效报文是指那些正确被接收的（直到 EOF 都没有错误）且通过了标识符过滤的报文。前面我们知道 CAN 的接收有 2 个 FIFO，我们每个滤波器组都可以设置其关联的 FIFO，通过 CAN_FFA1R 的设置，可以将滤波器组关联到 FIFO0/FIFO1。

CAN 接收流程为：FIFO 空→收到有效报文→挂号_1（存入 FIFO 的一个邮箱，这个由硬件控制，我们不需要理会）→收到有效报文→挂号_2→收到有效报文→挂号_3→收到有效报文溢出。

这个流程里面，我们没有考虑从 FIFO 读出报文的情况，实际情况是：我们必须在 FIFO 溢出之前，读出至少 1 个报文，否则下个报文到来，将导致 FIFO 溢出，从而出现报文丢失。每读出 1 个报文，相应的挂号就减 1，直到 FIFO 空。CAN 接收流程如图 30.1.2.13 所示：

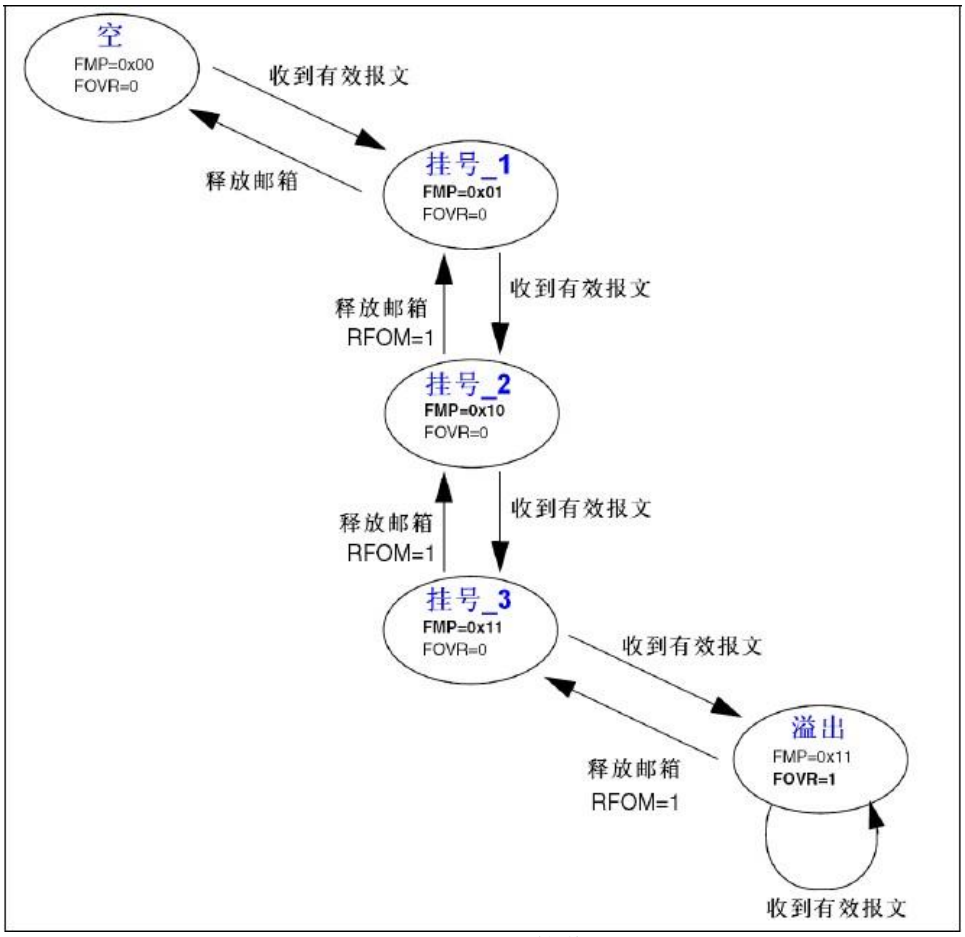


图 30.1.2.13 FIFO 接收报文

FIFO 接收到的报文数，我们可以通过查询 CAN_RFxR 的 FMP 寄存器来得到，只要 FMP 不为 0，我们就可以从 FIFO 读出收到的报文。

接下来，我们简单看看 STM32 的 CAN 位时间特性，STM32 的 CAN 位时间特性和之前我们介绍的 CAN 协议中，稍有点区别。STM32 把传播时间段和相位缓冲段 1（STM32 称之为时间段 1）合并了，所以 STM32 的 CAN 一个位只有 3 段：同步段（SYNC_SEG）、时间段 1（BS1）和时间段 2（BS2）。STM32 的 BS1 段可以设置为 1~16 个时间单元，刚好等于我们上面介绍的传播时间段和相位缓冲段 1 之和。STM32 的 CAN 位时序如图 30.1.2.14 所示：

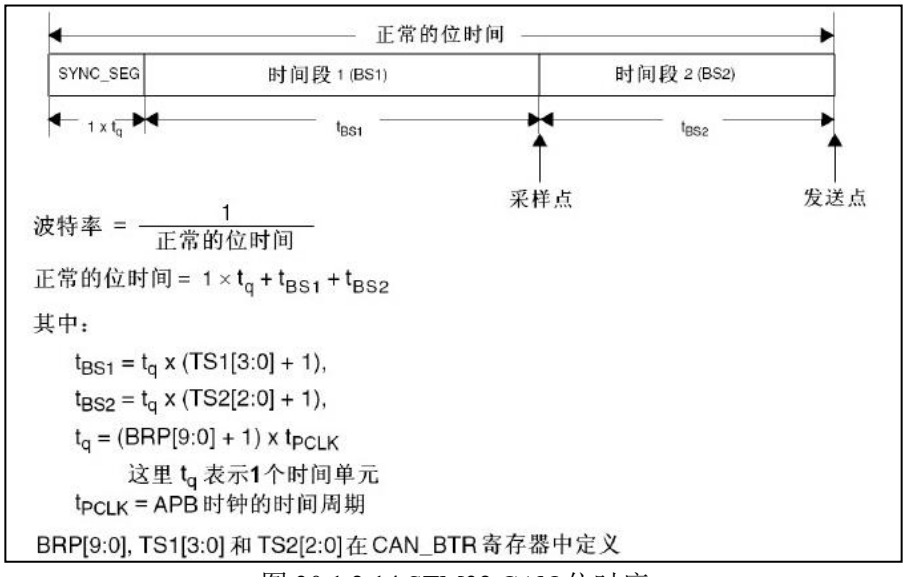


图 30.1.2.14 STM32 CAN 位时序

图中还给出了 CAN 波特率的计算公式，我们只需要知道 BS1 和 BS2 的设置，以及 APB1 的时钟频率（STM32F4 的时钟频率一般为 42Mhz），就可以方便的计算出波特率。比如设置 TS1=6、TS2=5 和 BRP=5，在 APB1 频率为 42Mhz 的条件下，即可得到 CAN 通信的波特率 =42000/[(7+6+1)*6]=500Kbps。

30.1.3 CAN 寄存器

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。

- **CAN 的主控制寄存器（CAN_MCR）**

CAN 的主控制寄存器（CAN_MCR）各位描述如图 30.1.3.1 所示：

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

位 0 **INRQ**：初始化请求 (Initialization request)

软件通过将此位清零，来将硬件切换到正常模式。一旦在 Rx 信号上监测到连续 11 个隐性位，CAN 硬件即完成同步并准备进行发送和接收。硬件通过将 CAN_MSR 寄存器的 INAK 位清零来指示此事件。

软件通过将此位置 1 来请求 CAN 硬件进入初始化模式。一旦软件将 INRQ 位置 1，CAN 硬件将等待当前 CAN 活动（发送或接收）结束，然后进入初始化模式。硬件通过将 CAN_MSR 寄存器的 INAK 位置 1 来指示此事件。

图 30.1.3.1 寄存器 CAN_MCR 各位描述（部分）

该寄存器负责管理 CAN 的工作模式。这里我们仅介绍 INRQ 位，该位用来控制初始化请求。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。

该寄存器的详细描述，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》24.9.2 节（625 页）。

- **CAN 位时序寄存器（CAN_BTR）**

CAN 位时序寄存器各位描述如图 30.1.3.2 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 30.1.3.2 寄存器 CAN_BTR 各位描述（部分）

CAN 位时序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 的测试模式，STM32 提供了三种测试模式，回环模式、静默模式和回环静默模式。这里我们简单介绍下回环模式。在回环模式下，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器组)在接收 FIFO 的输出邮箱里。也就是回环模式是一个自发自收的模式，如图 30.1.3.3 所示：

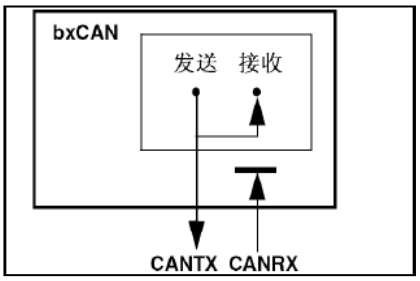


图 30.1.3.3 CAN 回环模式

回环模式可用于自测试。为了避免外部的影响，在回环模式下 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)。在回环模式下，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。

- **CAN 发送邮箱标识符寄存器（CAN_TIxR）**

CAN 发送邮箱标识符寄存器（CAN_TIxR）（x=0~3），该寄存器各位描述如图 30.1.3.4 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 30.1.3.4 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（位 0）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。

- **CAN 发送邮箱数据长度和时间戳寄存器(CAN_TDTxR)**

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。低 4 位的描述如图 30.1.3.5 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TIME[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							TGT	Reserved				DLC[3:0]			
							rw					rw	rw	rw	rw

位 3:0 **DLC[3:0]**：数据长度代码 (Data length code)
该字段定义数据帧或遥控帧请求中的数据字节数。
一条消息可以包含 0 到 8 个数据字节，具体取决于 DLC 字段的值。

图 30.1.3.5 寄存器 CAN_TDTxR 低四位描述

- **CAN 发送邮箱低字节数据寄存器(CAN_TDLxR)**

CAN 发送邮箱低字节数据寄存器(CAN_TDLxR)(x=0~2)，该寄存器各位描述如图 30.1.3.6 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 30.1.3.6 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。

- **CAN 接收 FIFO 邮箱标识符寄存器(CAN_RIxR)**

CAN 接收 FIFO 邮箱标识符寄存器(CAN_RIxR)(x=0/1)，该寄存器各位描述同 CAN_TIxR 寄存器几乎一模一样，只是最低位为保留位，该寄存器用于保存接收到的报文标识符等信息，我们可以通过读该寄存器获取相关信息。

同样的，CAN 接收 FIFO 邮箱数据长度和时间戳寄存器(CAN_RDTxR)、CAN 接收 FIFO 邮箱低字节数据寄存器(CAN_RDLxR)和 CAN 接收 FIFO 邮箱高字节数据寄存器(CAN_RDHxR)分别和发送邮箱的：CAN_TDTxR、CAN_TDLxR 以及 CAN_TDHxR 类似，这里我们就不单独一一介绍了。详细介绍，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》24.9.3 节（635 页）。

- **CAN 过滤器模式寄存器（CAN_FM1R）**

CAN 过滤器模式寄存器（CAN_FM1R），该寄存器各位描述如图 30.1.3.7 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

注意：　请参见第 617 页的图 231：筛选器组尺度配置 - 寄存器构成。

位 31:28　保留，必须保持复位值。

位 27:0 **FBMx**：筛选器模式 (Filter mode)
筛选器 x 的寄存器的模式
0：筛选器存储区 x 的两个 32 位寄存器处于标识符屏蔽模式。
1：筛选器存储区 x 的两个 32 位寄存器处于标识符列表模式。

图 30.1.3.7 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。

- **CAN 过滤器位宽寄存器(CAN_FS1R)**

CAN 过滤器位宽寄存器(CAN_FS1R)，该寄存器各位描述如图 30.1.3.8 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 30.1.3.8 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。

- **CAN 过滤器 FIFO 关联寄存器（CAN_FFA1R）**

CAN 过滤器 FIFO 关联寄存器（CAN_FFA1R），该寄存器各位描述如图 30.1.3.9 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 30.1.3.9 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。

- **CAN 过滤器激活寄存器（CAN_FA1R）**

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，把对应位置 1，即开启对应的滤波器组；置 0 则关闭该滤波器组。

- **CAN 的过滤器组 i 的寄存器 x（CAN_FiRx）**

CAN 的过滤器组 i 的寄存器 x（CAN_FiRx）（互联产品中 i=0~27，其它产品中 i=0~13；x=1/2）。该寄存器各位描述如图 30.1.3.10 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FB31	FB30	FB29	FB28	FB27	FB26	FB25	FB24	FB23	FB22	FB21	FB20	FB19	FB18	FB17	FB16
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FB15	FB14	FB13	FB12	FB11	FB10	FB9	FB8	FB7	FB6	FB5	FB4	FB3	FB2	FB1	FB0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

在所有配置中：

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 30.1.3.10 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，请参见前面的图 30.1.2.10（过滤器组位宽模式设置）的说明。

关于 CAN 的介绍，就到此结束了。

30.2 硬件设计

1. 例程功能

通过 KEY1 按键选择 CAN 的工作模式（正常模式/回环模式），然后通过 KEY0 控制数据发送，接着查询是否有数据接收到，假如接收到数据，就将接收到的数据显示在 LCD 模块上。如果是回环模式，我们不需要 2 个开发板。如果是正常模式，我们就需要 2 个开发板，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。

2. 硬件资源

1） LED 灯
LED0 - PE0
2） KEY0 和 KEY2 按键
KEY0 - PE2
KEY2 - PE4
3） TFTLCD 模块
4） STM32 自带 CAN 控制器
5） CAN 收发芯片 JTA1050/SIT1050T

3. 原理图

STM32 有 CAN 的控制器，但要实现 CAN 通讯的差分电平，我们还需要借助外围电路来实现，根据我们需要实现的程序功能，我们设计电路原理如下：

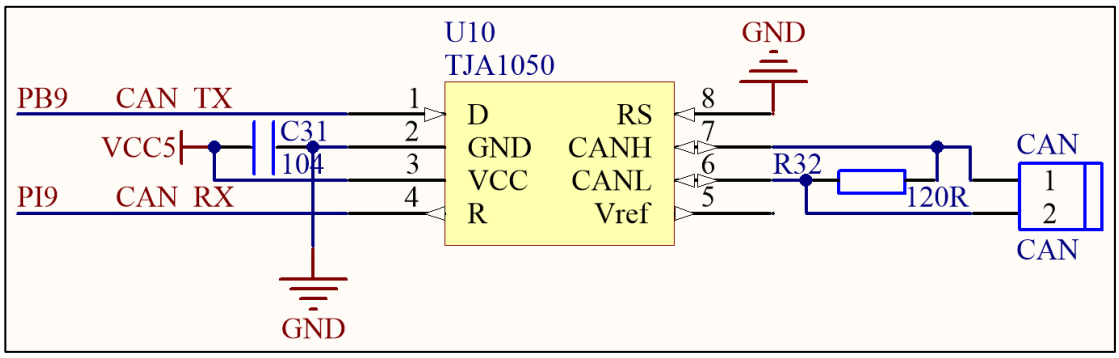


图 30.2.2 CAN 连接原理设计

从上图可以看出：STM32F407 的 CAN 连接到 TJA1050/SIT1050T 收发芯片，然后通过接线端子（CAN）同外部的 CAN 总线连接，在 F407 电机开发板上带有 120Ω 终端电阻，如果开发板不作为 CAN 的终端的话，需要把这个电阻去掉，以免影响通信。

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 是错误接法），接反了会导致通讯异常!!

30.3 程序设计

30.3.1 CAN 的 HAL 库驱动

CAN 在 HAL 库中的驱动代码在 stm32f4xx_hal_can.c 文件（及其头文件）中。

1. HAL_CAN_Init 函数

要使用一个外设首先要对它进行初始化，所以先看 CAN 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_CAN_Init(CAN_HandleTypeDef *hcan);
```

- **函数描述：**

  用于 CAN 控制器的初始化。

- **函数形参：**

  形参 1 是 CAN 的控制句柄，结构体类型是 CAN_HandleTypeDef，其定义如下：

```
typedef struct __CAN_HandleTypeDef
{
  CAN_TypeDef                 *Instance;      /* CAN 控制寄存器基地址 */
  CAN_InitTypeDef             Init;          /* 初始化参数结构体 */
  __IO HAL_CAN_StateTypeDef   State;         /* CAN 通讯状态 */
  __IO uint32_t               ErrorCode;     /* CAN 通讯结果编码 */
} CAN_HandleTypeDef;
```

1）Instance：指向 CAN 寄存器基地址。可以根据 STM32F407 的寄存器的偏移量定义找到各个配置 CAN 的寄存器并对其进行操作。

2）Init：can 初始化结构体，用于配置 CAN 的工作模式、波特率等等。这个结构体定义在 stm32f4xx_hal_can.h 中有列出。

```
typedef struct
{
  uint32_t Prescaler;         /* 分频值，可以配置为 1~1024 间的任意整数 */
  uint32_t Mode;              /* can 操作模式，有效值参考 CAN_operating_mode 的描述 */
  uint32_t SyncJumpWidth;     /* CAN 硬件的最大超时时间 */
  uint32_t TimeSeg1;          /* CAN_time_quantum_in_bit_segment_1 */
  uint32_t TimeSeg2;          /* CAN_time_quantum_in_bit_segment_2 */
  FunctionalState TimeTriggeredMode;        /* 启用或禁用时间触发模式 */
  FunctionalState AutoBusOff;               /* 禁止/使能软件自动断开总线的功能 */
  FunctionalState AutoWakeUp;               /* 禁止/使能 CAN 的自动唤醒功能 */
  FunctionalState AutoRetransmission;       /* 禁止/使能 CAN 的自动传输模式 */
  FunctionalState ReceiveFifoLocked;        /* 禁止/使能 CAN 的接收 FIFO */
  FunctionalState TransmitFifoPriority;     /* 禁止/使能 CAN 的发送 FIFO */
} CAN_InitTypeDef;
```

调用 CAN 的初始化函数时,我们主要也是对这个结构体赋值，配置 CAN 的工作模式。

3）State：CAN 操作状态，主要用于 HAL 库中的函数。

4）ErrorCode：CAN 错误操作信息。CAN 定义了多个错误返回值，可以方便查找通讯异常的可能原因，可以看到 CAN 总线的容错能力要大于串口。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值，有 4 个，分别是 HAL_OK 表示成功，HAL_ERROR 表示错误，HAL_BUSY 表示忙碌，HAL_TIMEOUT 为超时。

调用初始化函数之后，同样我们需要重定义 HAL_CAN_MspInit 来初始化跟底层硬件相关的配置，我们后面编写初始化函数时用到。

2. HAL_CAN_ConfigFilter 函数

CAN 的接收过滤器是属于硬件，可以根据软件的设置，在接收报文的时候，可以过滤出符合过滤器配置条件的报文 ID，大大节省了 CPU 的开销，过滤器配置函数定义如下：

```
HAL_StatusTypeDef HAL_CAN_ConfigFilter(CAN_HandleTypeDef *hcan,
                                       CAN_FilterTypeDef *sFilterConfig)
```

- **函数描述：**

用于配置 CAN 的接收过滤器。

- **函数形参：**

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

形参 2 是过滤器的结构体，这个是根据 STM32 的 CAN 过滤器模式设置的一些配置参数，它的结构如下：

```
typedef struct
{
  uint32_t FilterIdHigh;          /* 过滤器标识符高位 */
  uint32_t FilterIdLow;           /* 过滤器标识符低位 */
  uint32_t FilterMaskIdHigh;      /* 过滤器掩码号高位（列表模式下，也是属于标识符） */
  uint32_t FilterMaskIdLow;       /* 过滤器掩码号低位（列表模式下，也是属于标识符） */
  uint32_t FilterFIFOAssignment;  /* 与过滤器组管理的 FIFO */
  uint32_t FilterBank;            /* 指定过滤器组，单 CAN 为 0~13，双 CAN 可为 0~27 */
  uint32_t FilterMode;            /* 过滤器的模式 标识符屏蔽位模式/标识符列表模式 */
  uint32_t FilterScale;           /* 过滤器的位宽 32 位/16 位 */
  uint32_t FilterActivation;      /* 禁用或者使能过滤器 */
  uint32_t SlaveStartFilterBank;  /* 双 CAN 模式下，规定 CAN 的主从模式的过滤器分配 */
} CAN_FilterTypeDef;
```

我们通过配置过滤器，通过过滤器组的报文，即可从关联的 FIFO 的输出邮箱中获取。

- **函数返回值：**

我们只关注 HAL_OK 的情况。

3. HAL_CAN_Start 函数

使能 CAN 控制器以接入总线进行数据收发处理。

```
HAL_StatusTypeDef HAL_CAN_Start(CAN_HandleTypeDef *hcan)
```

- **函数描述：**

按需要配置完 CAN 总线后，使能 CAN 控制器以接入总线进行数据收发处理。

- **函数形参：**

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

- **函数返回值：**

我们只关注 HAL_OK 的情况。

4. HAL_CAN_ActivateNotification 函数

使能 CAN 的各种中断。

```
HAL_StatusTypeDef HAL_CAN_ActivateNotification(CAN_HandleTypeDef *hcan,
uint32_t ActiveITs)
```

- **函数描述：**

CAN 定义了多种传输中断以满足多种需求，我们只需要在 ActiveITs 中填入相关中断即可，中断源可以在 CAN_IER 寄存器中找到。

- **函数形参：**

形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。

- **函数返回值：**
  我们只关注 HAL_OK 的情况。

5. HAL_CAN_AddTxMessage 函数

发送报文函数。

```
HAL_StatusTypeDef HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan,
           CAN_TxHeaderTypeDef *pHeader, uint8_t aData[], uint32_t *pTxMailbox)
```

- **函数描述：**
  该函数用于向发送邮箱添加发送报文，并激活发送请求
- **函数形参：**
  形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。
  形参 2 是 CAN 发送的结构体，它的结构如下：

```
typedef struct
{
  uint32_t StdId;    /* 标准标识符 11 位 范围:0~0x7FF */
  uint32_t ExtId;    /* 扩展标识符 29 位 范围:0~0x1FFFFFFF */
  uint32_t IDE;      /* 标识符类型 CAN_ID_STD / CAN_ID_EXT */
  uint32_t RTR;      /* 帧类型 CAN_RTR_DATA / CAN_RTR_REMOTE */
  uint32_t DLC;      /* 帧长度 范围:0~8byte */
  FunctionalState TransmitGlobalTime;   /* 时间戳是否在开始时捕获 */

} CAN_TxHeaderTypeDef;
```

这里需要注意的是：当标识符选择位 IDE 为 CAN_ID_STD 时，表示本报文是标准帧，使用 StdId 成员存储报文 ID；当它的值为 CAN_ID_EXT 时，表示本报文是扩展帧，使用 ExtId 成员存储报文 ID。其他成员可以对照发送邮箱寄存器相关位进行理解。

形参 3 是报文的内容。

形参 4 是发送邮箱编号，可选三个发送邮箱之一。

6. HAL_CAN_GetRxMessage 函数

接收消息函数。

```
HAL_StatusTypeDef HAL_CAN_GetRxMessage(CAN_HandleTypeDef *hcan, uint32_t
                          RxFifo, CAN_RxHeaderTypeDef *pHeader, uint8_t aData[])
```

- **函数描述：**
  该函数可从接收 FIFO 里面的输出邮箱获取到消息报文。
- **函数形参：**
  形参 1 是 CAN 的控制句柄指针，初始化函数已经介绍过它的结构了，这里不重复了。
  形参 2 是接收 FIFO，具体是 FIFO0/1 ，得看过滤器组关联的 FIFO。
  形参 3 是 CAN 接收的结构体，它的结构如下：

```
typedef struct
{
  uint32_t StdId;        /* 标准标识符 11 位 范围:0~0x7FF */
  uint32_t ExtId;        /* 扩展标识符 29 位 范围:0~0x1FFFFFFF */
  uint32_t IDE;          /* 标识符类型 CAN_ID_STD / CAN_ID_EXT */
  uint32_t RTR;          /* 帧类型 CAN_RTR_DATA / CAN_RTR_REMOTE */
  uint32_t DLC;          /* 帧长度 范围:0~8byte */
  uint32_t Timestamp;    /* 在帧接收开始时开始捕获的时间戳 */
  uint32_t FilterMatchIndex; /* 过滤器匹配序号 */
} CAN_RxHeaderTypeDef;
```

在发送结构体中，同样的，也是通过 IDE 位确认该消息报文的标识符类型，该结构体不同于发送结构体还有一个过滤器匹配序号成员，可以查看到此报文是通过哪里过滤器到达接收 FIFO。其他成员可以对照发送邮箱寄存器相关位进行理解。

形参 4 是接收报文的内容。

CAN 的初始化配置步骤

1) **CAN 参数初始化（工作模式、波特率等）**
   HAL 库通过调用 CAN 初始化函数 HAL_CAN_Init 完成对 CAN 参数初始化，详见例程源

码。

注意：该函数会调用：HAL_CAN_MspInit 函数来完成对 CAN 底层的初始化，包括：CAN 以及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2) 开启 CAN 和对应管脚时钟，配置 CAN_TX 和 CAN_RX 的复用功能输出

首先开启 CAN 的时钟，然后配置 CAN 相关引脚为复用功能（对应的引脚可查看中文参考手册 P191）。本实验中 CAN_TX 对应的是 PB9，CAN_RX 对应的是 PI9。他们的时钟开启方法如下：

```
__HAL_RCC_CAN1_CLK_ENABLE();            /* 使能 CAN1 */
__HAL_RCC_GPIOB_CLK_ENABLE();           /* 开启 GPIOB 时钟 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

3) 设置滤波器

HAL 库通过调用 HAL_CAN_ConfigFilter 完成 CAN 的滤波器相关参数初始化。

4) CAN 数据接收和发送

通过调用 HAL_CAN_AddTxMessage 函数进行发送消息。

通过调用 HAL_CAN_GetRxMessage 函数进行接收数据。

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。本实验也提供 CAN 接收中断，详看例程源码，这里就不作介绍了。

30.3.2 程序流程图

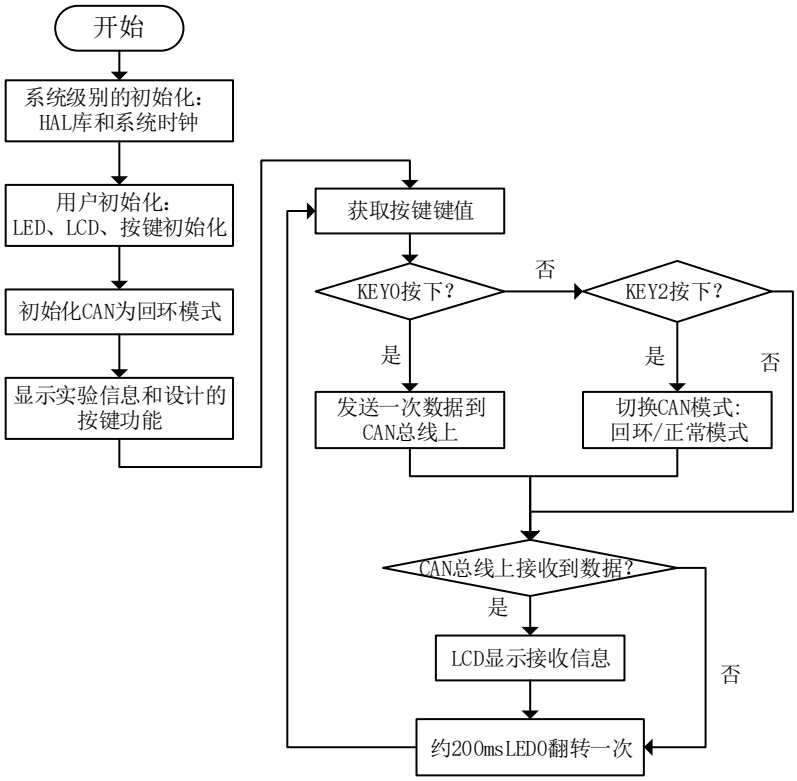


图 30.3.2.1 CAN 通讯实验程序流程图

30.3.3 程序解析

我们要使用 LED、LCD、按键这些功能，直接复制上一个 232 实验的代码，把 rs232 的代码从工程中移除，并在 Drivers/BSP 目录下新建一个《CAN》文件夹，与之前一样，新建 can.c/can.h 文件并把它们加入到工程中。

1. can.c 函数

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。CAN 驱动相关源码包括两个文件：can.c 和 can.h。

我们根据《STM32F4xx 参考手册_V4（中文版）.pdf》第 24 章的内容，我们利用前面介绍的 HAL 库函数来配置 CAN 的接收时钟及模式等参数，配置过滤器以使能硬件自动过滤功能，最后使能 CAN 以开始 CAN 控制器的工作，编写 CAN 初始化函数。

```
/**
 * @brief       CAN 初始化
 * @param       tsjw    : 重新同步跳跃时间单元.范围: 1~3;
 * @param       tbs2    : 时间段 2 的时间单元.范围: 1~8;
 * @param       tbs1    : 时间段 1 的时间单元.范围: 1~16;
 * @param       brp     : 波特率分频器.范围: 1~1024;
 *   @note      以上 4 个参数, 在函数内部会减 1, 所以, 任何一个参数都不能等于 0
 *              CAN 挂在 APB1 上面, 其输入时钟频率为 Fpclk1 = PCLK1 = 42Mhz
 *              tq     = brp * tpclk1;
 *              波特率 = Fpclk1 / ((tbs1 + tbs2 + 1) * brp);
 *              我们设置 can_init(1, 6, 7, 6, 1), 则 CAN 波特率为:
 *              42M / ((6 + 7 + 1) * 6) = 500Kbps
 * @param       mode    : CAN_MODE_NORMAL,  正常模式;
 *                        CAN_MODE_LOOPBACK,回环模式;
 * @retval      0,  初始化成功; 其他, 初始化失败;
 */
uint8_t can_init(uint32_t tsjw, uint32_t tbs2, uint32_t tbs1, uint16_t brp,
                 uint32_t mode)
{
    g_canx_handler.Instance = CAN1;
    g_canx_handler.Init.Prescaler = brp;        /* 分频系数(Fdiv)为 brp+1 */
    g_canx_handler.Init.Mode = mode;            /* 模式设置 */
    /* 重新同步跳跃宽度(Tsjw)为 tsjw+1 个时间单位 CAN_SJW_1TQ~CAN_SJW_4TQ */
    g_canx_handler.Init.SyncJumpWidth = tsjw;
    g_canx_handler.Init.TimeSeg1 = tbs1;  /* tbs1 范围 CAN_BS1_1TQ~CAN_BS1_16TQ */
    g_canx_handler.Init.TimeSeg2 = tbs2;  /* tbs2 范围 CAN_BS2_1TQ~CAN_BS2_8TQ */
    g_canx_handler.Init.TimeTriggeredMode = DISABLE;    /* 非时间触发通信模式 */
    g_canx_handler.Init.AutoBusOff = DISABLE;           /* 软件自动离线管理 */
    g_canx_handler.Init.AutoWakeUp = DISABLE;           /* 通过软件唤醒睡眠模式 */
    g_canx_handler.Init.AutoRetransmission = ENABLE;    /* 禁止报文自动传送 */
    g_canx_handler.Init.ReceiveFifoLocked = DISABLE;    /* 报文不锁定,新的覆盖旧的 */
    g_canx_handler.Init.TransmitFifoPriority = DISABLE;/* 优先级由报文标识符决定 */

    if (HAL_CAN_Init(&g_canx_handler) != HAL_OK)
    {
        return 1;
    }

#if CAN_RX0_INT_ENABLE
    /* 使用中断接收,FIFO0 消息挂号中断允许 */
    __HAL_CAN_ENABLE_IT(&g_canx_handler, CAN_IT_RX_FIFO0_MSG_PENDING);
    HAL_NVIC_EnableIRQ(USB_LP_CAN1_RX0_IRQn);           /* 使能 CAN 中断 */
    HAL_NVIC_SetPriority(USB_LP_CAN1_RX0_IRQn, 1, 0);   /* 抢占优先级 1，子优先级 0 */
#endif

    CAN_FilterTypeDef sFilterConfig;                    /*配置 CAN 过滤器*/
    sFilterConfig.FilterBank = 0;                       /* 过滤器 0 */
    sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
    sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
    sFilterConfig.FilterIdHigh = 0x0000;                /* 32 位 ID */
    sFilterConfig.FilterIdLow = 0x0000;
    sFilterConfig.FilterMaskIdHigh = 0x0000;            /* 32 位 MASK */
    sFilterConfig.FilterMaskIdLow = 0x0000;
```

```
    sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0; /* 过滤器 0 关联到 FIFO0 */
    sFilterConfig.FilterActivation = ENABLE;           /* 激活滤波器 0 */
    sFilterConfig.SlaveStartFilterBank = 14;

    if (HAL_CAN_ConfigFilter(&g_canx_handler, &sFilterConfig) != HAL_OK)
    { /* 过滤器配置 */
        return 2;
    }

    if (HAL_CAN_Start(&g_canx_handler) != HAL_OK)
    { /* 启动 CAN 外围设备 */
        return 3;
    }

    return 0;
}
```

调用 HAL_CAN_Init 后会调用 HAL_CAN_MspInit，我们重定义这个函数，在函数中初始化我们用于控制 CAN 的收发引脚：

```
void HAL_CAN_MspInit(CAN_HandleTypeDef *hcan)
{
    if (CAN1 == hcan->Instance)
    {
        CAN_RX_GPIO_CLK_ENABLE();        /* CAN_RX 脚时钟使能 */
        CAN_TX_GPIO_CLK_ENABLE();        /* CAN_TX 脚时钟使能 */
        __HAL_RCC_CAN1_CLK_ENABLE();     /* 使能 CAN1 时钟 */

        GPIO_InitTypeDef gpio_init_struct;

        gpio_init_struct.Pin = CAN_TX_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;
        gpio_init_struct.Pull = GPIO_PULLUP;
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
        gpio_init_struct.Alternate = GPIO_AF9_CAN1;
        HAL_GPIO_Init(CAN_TX_GPIO_PORT, &gpio_init_struct);/* CAN_TX 脚 模式设置 */

        gpio_init_struct.Pin = CAN_RX_GPIO_PIN;
        /* CAN_RX 脚 必须设置成输入模式 */
        HAL_GPIO_Init(CAN_RX_GPIO_PORT, &gpio_init_struct);
    }
}
```

初始化函数就编写完成了，我们要设置它的工作波特率为 500Kbps，设置工作模式为回环模式，参照各个参数的意义，我们最后用以下的配置来完成初始化设置：

```
    can_init(CAN_SJW_1TQ, CAN_BS2_6TQ, CAN_BS1_7TQ, 6, CAN_MODE_LOOPBACK);
```

要与其它的 CAN 节点设备通讯，我们需要编写 CAN 相关的收发函数。首先是发送函数，发送报文是有 ID，所以我们在发送时需要设定 ID，故需要设计一个形参为 ID 号，我们利用 HAL 库的发送函数封装一个更方便我们使用的函数，代码如下：

```
/**
 * @brief       CAN 发送一组数据
 *   @note      发送格式固定为: 标准 ID, 数据帧
 * @param       id      : 标准 ID(11 位)
 * @retval      发送状态 0, 成功; 1, 失败;
 */
uint8_t can_send_msg(uint32_t id, uint8_t *msg, uint8_t len)
{
    uint32_t TxMailbox = CAN_TX_MAILBOX0;
    g_canx_txheader.StdId = id;          /* 标准标识符 */
    g_canx_txheader.ExtId = id;          /* 扩展标识符(29 位) */
    g_canx_txheader.IDE = CAN_ID_STD;    /* 使用标准帧 */
    g_canx_txheader.RTR = CAN_RTR_DATA; /* 数据帧 */
    g_canx_txheader.DLC = len;

    if (HAL_CAN_AddTxMessage(&g_canx_handler, &g_canx_txheader,
```

```
                                msg, &TxMailbox) != HAL_OK) /* 发送消息 */
    {
        return 1;
}

    /* 等待发送完成,所有邮箱为空(3 个邮箱) */
    while (HAL_CAN_GetTxMailboxesFreeLevel(&g_canx_handler) != 3);

    return 0;
}
```

在 CAN 初始化时，我们对于过滤器的配置是不过滤任何报文 ID，也就是说可以接收全部报文。但是我们可以编写接收函数时，使用软件的方式过滤报文 ID，通过形参来跟接收到的报文 ID 进行匹配。接收函数代码具体如下：

```
/**
 * @brief       CAN 接收数据查询
 *   @note      接收数据格式固定为: 标准 ID, 数据帧
 * @param       id      : 要查询的 标准 ID(11 位)
 * @param       buf     : 数据缓存区
 * @retval      接收结果
 *   @arg       0   , 无数据被接收到;
 *   @arg       其他, 接收的数据长度
 */
uint8_t can_receive_msg(uint32_t id, uint8_t *buf)
{
    if (HAL_CAN_GetRxFifoFillLevel(&g_canx_handler, CAN_RX_FIFO0) != 1)
    {
        return 0;
    }

    if (HAL_CAN_GetRxMessage(&g_canx_handler, CAN_RX_FIFO0, &g_canx_rxheader,
                             buf) != HAL_OK)
    {
        return 0;
    }

    /* 接收到的 ID 不对 / 不是标准帧 / 不是数据帧 */
    if (g_canx_rxheader.StdId!= id || g_canx_rxheader.IDE != CAN_ID_STD ||
         g_canx_rxheader.RTR != CAN_RTR_DATA)
    {
        return 0;
    }
    return g_canx_rxheader.DLC;
}
```

最后，我们可以把 can_send_msg 函数加到 USMART 接口中，就可以方便地用串口来调试 CAN 接口了。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint8_t i = 0, t = 0;
    uint8_t cnt = 0;
    uint8_t canbuf[8];
    uint8_t rxlen = 0;
    uint8_t res;
    uint8_t mode = 1; /* CAN 工作模式: 0,正常模式; 1,环回模式 */

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
```

```
    usart_init(115200);                         /* 串口初始化为 115200 */
    usmart_dev.init(84);                        /* 初始化 USMART */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    /* CAN 初始化, 环回模式, 波特率 500Kbps */
    can_init(CAN_SJW_1TQ, CAN_BS2_6TQ, CAN_BS1_7TQ, 6, CAN_MODE_LOOPBACK);

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "CAN TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "LoopBack Mode", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY0:Send KEY2:Mode", RED);
    lcd_show_string(30, 150, 200, 16, 16, "Count:", RED);      /* 显示当前计数值 */
    lcd_show_string(30, 170, 200, 16, 16, "Send Data:", RED);  /* 提示发送的数据 */
    lcd_show_string(30, 230, 200, 16, 16, "Receive Data:", RED);/*提示接收的数据*/

    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)
                {   /* 显示数据 */
                    lcd_show_xnum(30 + i * 32, 190, canbuf[i], 3, 16, 0x80, BLUE);
                }
                else
                {   /* 显示数据 */
                    lcd_show_xnum(30 + (i-4)*32,210, canbuf[i], 3, 16, 0x80, BLUE);
                }
            }

            res = can_send_msg(0x12, canbuf, 8); /* ID = 0x12, 发送 8 个字节 */
            if (res)
            {   /* 提示发送失败 */
                lcd_show_string(30 + 80, 170, 200, 16, 16, "Failed", BLUE);
            }
            else
            {   /* 提示发送成功 */
                lcd_show_string(30 + 80, 170, 200, 16, 16, "OK    ", BLUE);
            }
        }
        else if (key == KEY2_PRES) /* KEY2 按下, 改变 CAN 的工作模式 */
        {
            mode = !mode;
            /* CAN 初始化, 普通(0)/回环(1)模式, 波特率 500Kbps */
            can_init(CAN_SJW_1TQ, CAN_BS2_6TQ, CAN_BS1_7TQ, 6,
                     mode ? CAN_MODE_LOOPBACK : CAN_MODE_NORMAL);

            if (mode == 0) /* 正常模式, 需要 2 个开发板 */
            {
                lcd_show_string(30, 110, 200, 16, 16, "Normal Mode ", RED);
            }
            else /* 回环模式,一个开发板就可以测试了. */
            {
                lcd_show_string(30, 110, 200, 16, 16, "LoopBack Mode", RED);
            }
        }

        rxlen = can_receive_msg(0x12, canbuf); /* CAN ID = 0x12, 接收数据查询 */
```

```
        if (rxlen) /* 接收到有数据 */
        {
            lcd_fill(30, 270, 130, 310, WHITE); /* 清除之前的显示 */
            for (i = 0; i < rxlen; i++)
            {
                if (i < 4)
                {    /* 显示数据 */
                lcd_show_xnum(30 + i * 32, 250, canbuf[i], 3, 16, 0x80, BLUE);
                }
                else
                {    /* 显示数据 */
                 lcd_show_xnum(30 + (i-4) * 32, 270, canbuf[i], 3, 16,0x80, BLUE);
                }
            }
        }

        t++;
        delay_ms(10);
        if (t == 20)
        {
            LED0_TOGGLE(); /* 提示系统正在运行 */
            t = 0;
            cnt++;
            lcd_show_xnum(30 + 48, 150, cnt, 3, 16, 0x80, BLUE); /* 显示数据 */
        }
    }
}
```

main 函数的执行过程与程序流程图是一致的，这里需要注意：在选择正常模式的情况下，要使两个开发板通信成功，必须保持一致的波特率。其它细节，参考光盘资料中的源码即可。

30.4 下载验证

在代码编译成功之后，我们通过下载代码到开发板上，默认设置的回环模式，按下 KEY0 就可以在 LCD 模块上面看到自发自收的数据，如图 30.4.1 所示：

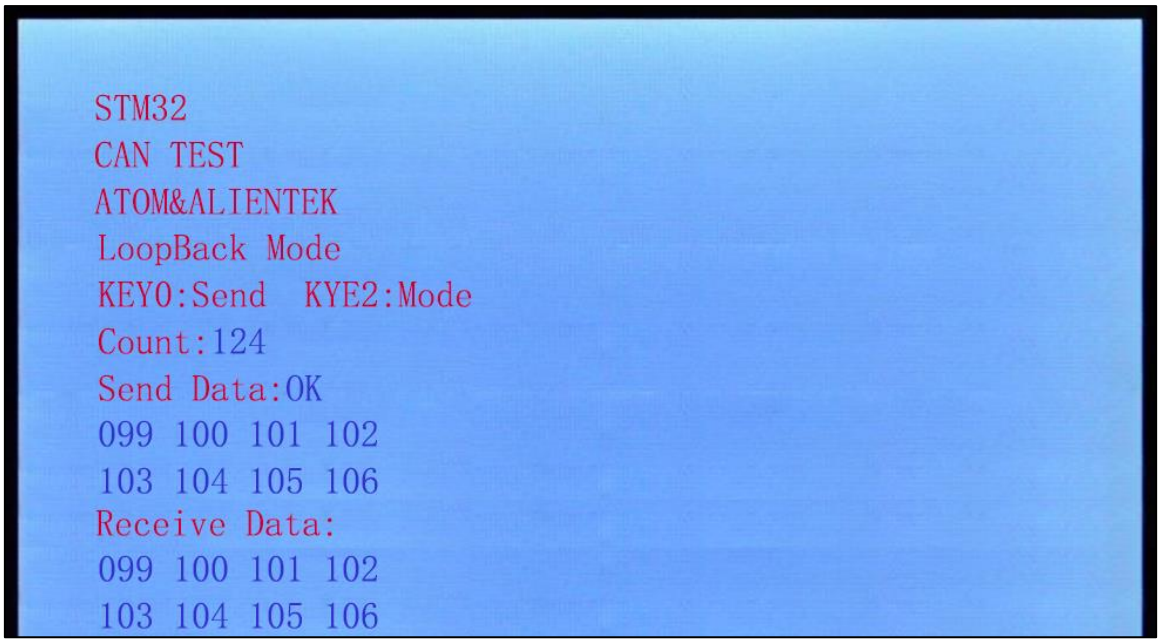


图 30.4.1 程序运行效果图

伴随 LED0 的不停闪烁，提示程序在运行。如果我们选择正常模式（KEY2 按键切换），就必须连接两个开发板的 CAN 接口，然后就可以互发数据了。如图 30.4.2 和图 30.4.3 所示：

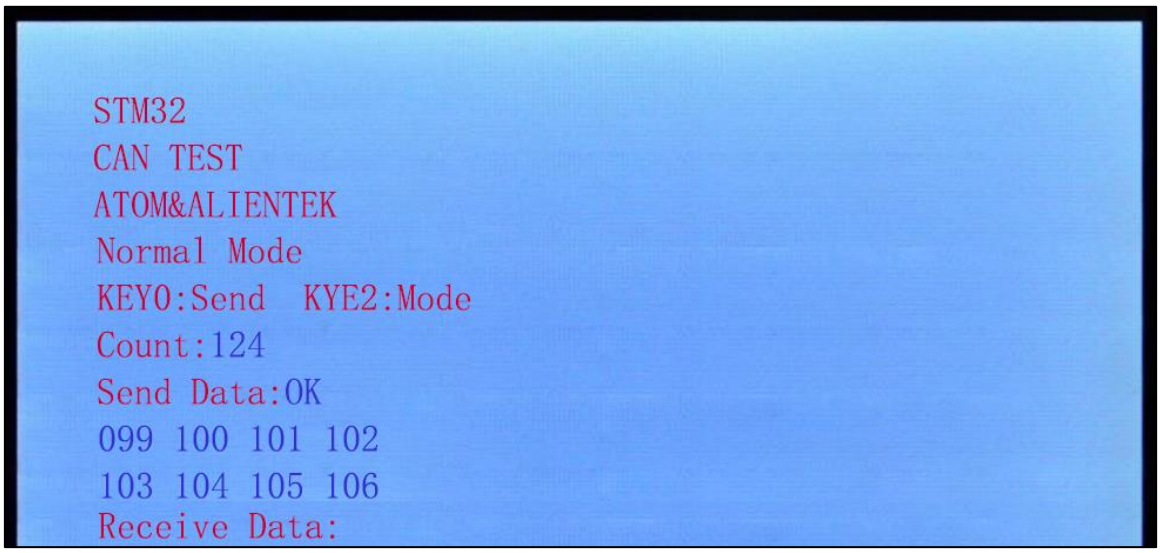


图 30.4.2 CAN 正常模式发送数据

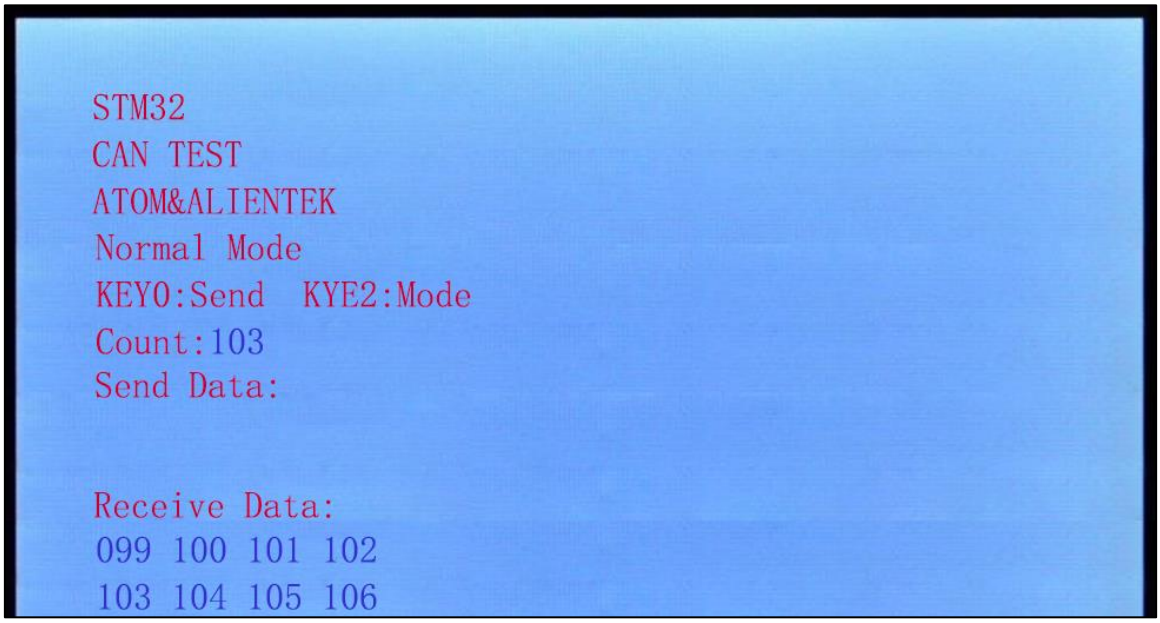


图 30.4.3 CAN 正常模式接收数据

图 30.4.2 来自开发板 A，发送了 8 个数据，图 30.4.3 来自开发板 B，收到了来自开发板 A 的个数据。另外，利用 USMART 测试的部分，我们这里就不做介绍了，大家可自行验证下。

第三十一章 触摸屏实验

本章，我们将介绍如何使用 STM32F407 来驱动触摸屏，STM32F407 本身并没有触摸屏控制器，但是它可以通过外接带触摸屏的 LCD 模块（比如正点原子 TFTLCD 模块），来实现触摸屏控制。在本章中，我们将向大家介绍 STM32 控制正点原子 TFTLCD 模块（包括电阻触摸与电容触摸），实现触摸屏驱动，最终实现一个手写板的功能。

本章分为如下几个小节：

31.1 触摸屏简介
31.2 硬件设计
31.3 程序设计
31.4 下载验证

31.1 触摸屏简介

触摸屏是在显示屏的基础上，在屏幕或屏幕上方分布一层与屏幕大小相近的传感器形成的组合器件。触摸和显示功能由软件控制，可以独立也可以组合实现，用户可以通过侦测传感器的触点再配合相应的软件实现触摸效果。目前最常用的触摸屏有两种：电阻式触摸屏与电容式触摸屏。下面，我们来分别介绍。

31.1.1 电阻式触摸屏

正点原子 2.4/2.8/3.5 寸 TFTLCD 模块自带的触摸屏都属于电阻式触摸屏，下面简单介绍下电阻式触摸屏的原理。

电阻式触摸屏是一种附在显示器上略大于显示区域的传感器，它将矩形区域中触摸点(X,Y)的物理位置转换为代表 X 坐标和 Y 坐标的电压。很多 LCD 模块都采用了电阻式触摸屏，这种屏幕可以用四线、五线、七线或八线来产生屏幕偏置电压，同时读回触摸点的电压。电阻式触摸屏基本上是薄膜加上玻璃的结构，薄膜和玻璃相邻的一面上均涂有 ITO(纳米铟锡金属氧化物)涂层，ITO 具有很好的导电性和透明性。当触摸操作时，薄膜下层的 ITO 会接触到玻璃上层的 ITO，经由感应器传出相应的电信号，经过转换电路送到处理器，通过运算转化为屏幕上的 X、Y 值，而完成点选的动作，并呈现在屏幕上。

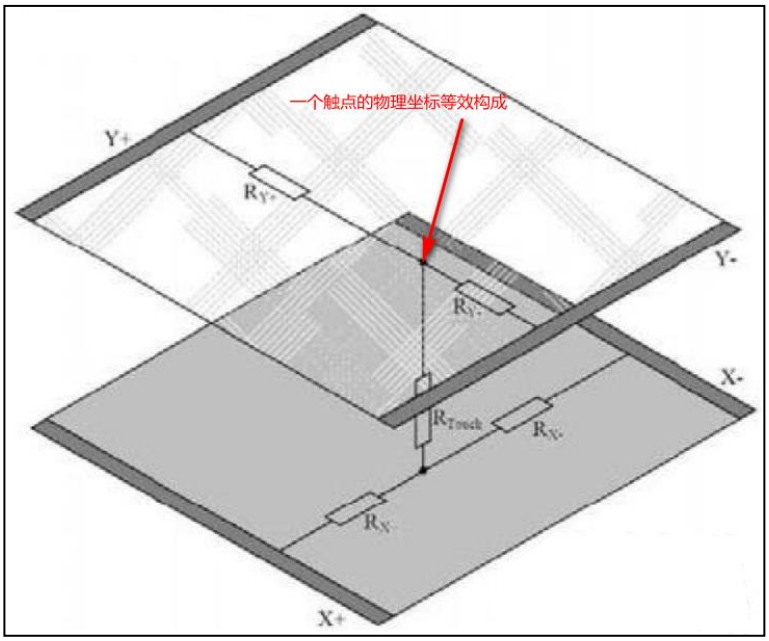


图 31.1.1.1 电阻式触摸屏的触点坐标结构

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。正点原子 TFTLCD 模块选择的是四线电阻式触摸屏，这种触摸屏的控制芯片有很多，包括：ADS7843、ADS7846、TSC2046、XPT2046 和 AK4182 等。这几款芯片的驱动基本上是一样的，也就是你只要写出了 ADS7843 的驱动，这个驱动对其他几个芯片也是有效的。而且封装也有一样的，完全 PINTOPIN 兼容。所以在替换起来，很方便。

正点原子 TFTLCD 模块自带的触摸屏控制芯片为 XPT2046。XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置，除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。XPT2046 采用微小的封装形式：TSSOP-16，QFN-16(0.75mm 厚度)和 VFBGA－48。工作温度范围为-40℃～+85℃。该芯片完全是兼容 ADS7843 和 ADS7846 的，关于这个芯片的详细使用，可以参考这两个芯片的 datasheet。

电阻式触摸屏就介绍到这里。

31.1.2 电容式触摸屏

现在几乎所有智能手机，包括平板电脑都是采用电容屏作为触摸屏，电容屏是利用人体感应进行触点检测控制，不需要直接接触或只需要轻微接触，通过检测感应电流来定位触摸坐标。正点原子 4.3/7 寸 TFTLCD 模块自带的触摸屏采用的是电容式触摸屏，下面简单介绍下电容式触摸屏的原理。

电容式触摸屏主要分为两种：

1、表面电容式电容触摸屏。

表面电容式触摸屏技术是利用 ITO(铟锡氧化物，是一种透明的导电材料)导电膜，通过电场感应方式感测屏幕表面的触摸行为进行。但是表面电容式触摸屏有一些局限性，它只能识别一个手指或者一次触摸。

2、投射式电容触摸屏。

投射电容式触摸屏是传感器利用触摸屏电极发射出静电场线。一般用于投射电容传感技术的电容类型有两种：自我电容和交互电容。

自我电容又称绝对电容，是最广为采用的一种方法，自我电容通常是指扫描电极与地构成的电容。在玻璃表面有用 ITO 制成的横向与纵向的扫描电极，这些电极和地之间就构成一个电容的两极。当用手或触摸笔触摸的时候就会并联一个电容到电路中去，从而使在该条扫描线上的总体的电容量有所改变。在扫描的时候，控制 IC 依次扫描纵向和横向电极，并根据扫描前后的电容变化来确定触摸点坐标位置。笔记本电脑触摸输入板就是采用的这种方式，笔记本电脑的输入板采用 X*Y 的传感电极阵列形成一个传感格子，当手指靠近触摸输入板时，在手指和传感电极之间产生一个小量电荷。采用特定的运算法则处理来自行、列传感器的信号来确定手指的位置。

交互电容又叫做跨越电容，它是在玻璃表面的横向和纵向的 ITO 电极的交叉处形成电容。交互电容的扫描方式就是扫描每个交叉处的电容变化，来判定触摸点的位置。当触摸的时候就会影响到相邻电极的耦合，从而改变交叉处的电容量，交互电容的扫面方法可以侦测到每个交叉点的电容值和触摸后电容变化，因而它需要的扫描时间与自我电容的扫描方式相比要长一些，需要扫描检测 X*Y 根电极。目前智能手机/平板电脑等的触摸屏，都是采用交互电容技术。

正点原子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。

投射式电容触摸屏采用纵横两列电极组成感应矩阵，来感应触摸。以两个交叉的电极矩阵，即：X 轴电极和 Y 轴电极，来检测每一格感应单元的电容变化，如图 31.1.2.1 所示：

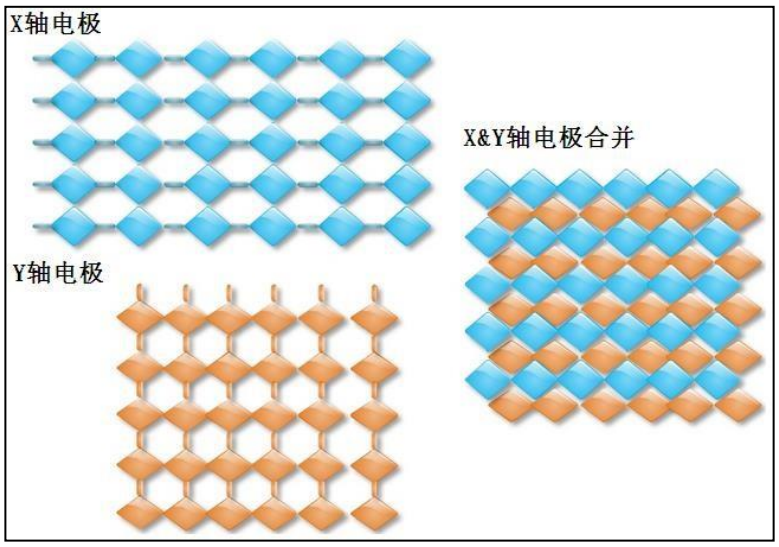


图31.1.2.1 投射式电容屏电极矩阵示意图

示意图中的电极，实际是透明的，这里是为了方便大家理解。图中，X、Y 轴的透明电极电容屏的精度、分辨率与 X、Y 轴的通道数有关，通道数越多，精度越高。以上就是电容触摸屏的基本原理，接下来看看电容触摸屏的优缺点：

电容触摸屏的优点：手感好、无需校准、支持多点触摸、透光性好。电容触摸屏的缺点：成本高、精度不高、抗干扰能力差。

这里特别提醒大家电容触摸屏对工作环境的要求是比较高的，在潮湿、多尘、高低温环境下面，都是不适合使用电容屏的。

电容触摸屏一般都需要一个驱动 IC 来检测电容触摸，正点原子的电容触摸屏使用的是 IIC 接口输出触摸数据的触摸芯片。

正点原子 7 寸 TFTLCD 模块的电容触摸屏，采用的是 15*10 的驱动结构（10 个感应通道，15 个驱动通道），采用的是 GT811/FT5206 做为驱动 IC。正点原子 4.3 寸 TFTLCD 模块有两种成触摸屏： 1，使用 GT9147 作为驱动 IC，采用 17*10 的驱动结构（10 个感应通道，17 个驱动通道）；2，使用 FT5206 作为驱动 IC，采用 13*8 的驱动结构（8 个感应通道，13 个驱动通道）。

这两个模块都只支持最多 5 点触摸，本例程支持正点原子的 4.3 寸屏模块和新版的 7 寸屏模块（采用 SSD1963+FT5206 方案），电容触摸驱动 IC，这里只介绍 GT9147，GT811/FT5206 的驱动方法同这两款 IC 是类似的，大家可以参考着学习即可。

GT9147 与 MCU 通过 4 根线连接：SDA、SCL、RST 和 INT。不过，GT9147 的 IIC 地址，可以是 0x14 或者 0x5D，当复位结束后的 5ms 内，如果 INT 是高电平，则使用 0x14 作为地址，否则使用 0x5D 作为地址，具体的设置过程，请看：GT9147 数据手册.pdf 这个文档。本章我们使用 0x14 作为器件地址（不含最低位，换算成读写命令则是读： 0x29，写：0x28），接下来，介绍一下 GT9147 的几个重要的寄存器。

1，控制命令寄存器（0x8040）

该寄存器可以写入不同值，实现不同的控制，我们一般使用 0 和 2 这两个值，写入 2，即可软复位 GT9147，在硬复位之后，一般要往该寄存器写 2，实行软复位。然后，写入 0，即可正常读取坐标数据（并且会结束软复位）。

2，配置寄存器组（0x8047~0x8100）

这里共 186 个寄存器，用于配置 GT9147 的各个参数，这些配置一般由厂家提供给我们（一个数组），所以我们只需要将厂家给我们的配置，写入到这些寄存器里面，即可完成 GT9147 的配置。由于 GT9147 可以保存配置信息（可写入内部 FLASH，从而不需要每次上电都更新配置），我们有几点注意的地方提醒大家：1，0x8047 寄存器用于指示配置文件版本号，程序写入的版本号，必须大于等于 GT9147 本地保存的版本号，才可以更新配置。2，0x80FF 寄存器用于存储校验和，使得 0x8047~0x80FF 之间所有数据之和为 0。3，0x8100 用于控制是否将配置保存在本地，写 0，则不保存配置，写 1 则保存配置。

3，产品 ID 寄存器（0x8140~0x8143）

这里总共由 4 个寄存器组成，用于保存产品 ID，对于 GT9147，这 4 个寄存器读出来就是： 9，1，4，7 四个字符（ASCII 码格式）。因此，我们可以通过这 4 个寄存器的值，来判断驱动 IC 的型号，从而判断是 OTT2001A 还是 GT9147，以便执行不同的初始化。

4，状态寄存器（0x814E）

该寄存器各位描述如表表 31.1.2.1 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 31.1.2.1 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。这和前面 OTT2001A 的表示方法稍微有点区别，OTT2001A 是每个位表示一个触点，这里是有多少有效触点值就是多少。最后，该寄存器在每次读取后，如果 bit7 有效，则必须写 0，清除这个位，否则不会输出下一次数据!! 这个要特别注意!!!

5，坐标数据寄存器（共 30 个）

这里共分成 5 组（5 个点），每组 6 个寄存器存储数据，以触点 1 的坐标数据寄存器组为例，如表 31.1.2.2 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 坐标高 8 位
0x8152	触点 1 y 坐标低 8 位	0x8153	触点 1 y 坐标高 8 位
0x8154	触点 1 触摸尺寸低 8 位	0x8155	触点 1 触摸尺寸高 8 位

表 31.1.2.2 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。同样 GT9147 也支持寄存器地址自增，我们只需要发送寄存器组的首地址，然后连续读取即可，GT9147 会自动地址自增，从而提高读取速度。

GT9147 相关寄存器的介绍就介绍到这里，更详细的资料，请参考：GT9147 编程指南.pdf 这个文档。

GT9147 只需要经过简单的初始化就可以正常使用了，初始化流程：硬复位→延时 10ms→结束硬复位→设置 IIC 地址→延时 100ms→软复位→更新配置（需要时）→结束软复位。此时 GT9147 即可正常使用了。然后，我们不停的查询 0x814E 寄存器，判断是否有有效触点，如果有，则读取坐标数据寄存器，得到触点坐标，特别注意，如果 0x814E 读到的值最高位为 1，就必须对该位写 0，否则无法读到下一次坐标数据。

电容式触摸屏部分，就介绍到这里。

31.1.3 触摸控制原理

前面已经简单地介绍了电阻屏和电容屏的原理，并且知道了不同类型的触摸屏其实是屏幕+触摸传感器组成。那么这里就会有两组相互独立的参数：屏幕坐标和触摸坐标。要实现触摸功能，就是要把触摸点和屏幕坐标对应起来。

我们以 LCD 显示屏为例，我们知道屏幕的扫描方向是可以编程设定的，而触摸点，在触摸传感器安装好后，AD 值的变化向方向则是固定的，我们以最常见的屏幕坐标方向：先从左到右，再从上到下扫描为例，此时，屏幕坐标和触点 AD 的坐标有类似的规律：从坐标原点出发，水平方向屏幕坐标增加时，AD 值的 X 方向也增加；屏幕坐标的 Y 方向坐标增加，AD 值的 Y 方向也增加；坐标减少时对应的关系也类似，可以用图 31.1.3.1 的示意图来表示这种关系：

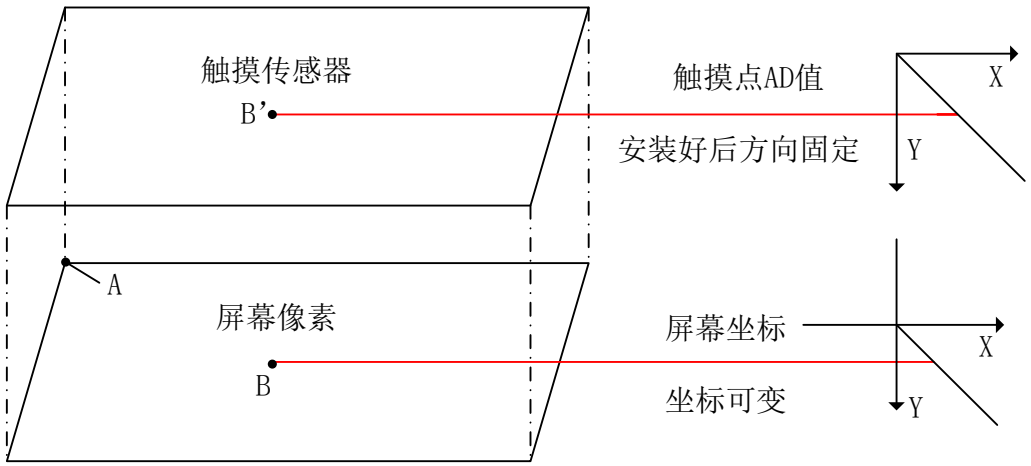


图 31.1.3.1 屏幕坐标和触摸坐标的一种对应关系

这里再来引入两个概念，物理坐标和逻辑坐标。物理坐标指触摸屏上点的实际位置，通常以液晶上点的个数来度量。逻辑坐标指这点被触摸时 A/D 转换后的坐标值。仍以图 30.1.3.1 为例，我们假定液晶最左上角为坐标轴原点 A，在液晶上任取一点 B（实际人手比像素点大得多，一次按下会有多个触点，此处取十字线交叉中心），B 在 X 方向与 A 相距 100 个点，在 Y 方向与 A 距离 200 个点，则这点的物理坐标 B 为（100，200）。如果我们触摸这一点时得到的 X 向 A/D 转换值为 200，Y 向 A/D 转换值为 400，则这点的逻辑坐标 B’为（200，400）。

需要特别说明的是，正点原子的电容屏的参数已经在出厂时由厂家调好，所以无需进行校准，而且可以直接读到转换后的触点坐标；对于电阻屏，请大家理解并熟记物理坐标和逻辑坐标逻辑上的对应关系，我们后面编程需要用到。

31.2 硬件设计

1. 例程功能

正点原子的触摸屏种类很多，并且设计了规格相对统一的接口。根据屏幕的种类不同，设置了相应的硬件 ID，可以通过软件进行判断触摸屏的种类。

本章实验功能简介：开机的时候先初始化 LCD，读取 LCD ID，随后，根据 LCD ID 判断是电阻触摸屏还是电容触摸屏，如果是电阻触摸屏，则先读取 24C02 的数据判断触摸屏是否已经校准过，如果没有校准，则执行校准程序，校准过后再进入电阻触摸屏测试程序，如果已经校准了，就直接进入电阻触摸屏测试程序。

如果是 4.3 寸电容触摸屏，则先读取芯片 ID，判断是不是 GT9147，如果是则执行 GT9147 的初始化代码，如果不是，则执行 OTT2001A 的初始化代码；如果是 7 寸电容触摸屏（仅支持新款 7 寸屏，使用 SSD1963+FT5206 方案），则执行 FT5206 的初始化代码，在初始化电容触摸屏完成后，进入电容触摸屏测试程序（电容触摸屏无需校准!!）。

电阻触摸屏测试程序和电容触摸屏测试程序基本一样，只是电容触摸屏支持最多 5 点同时触摸，电阻触摸屏只支持一点触摸，其他一模一样。测试界面的右上角会有一个清空的操作区域（RST），点击这个地方就会将输入全部清除，恢复白板状态。使用电阻触摸屏的时候，可以通过按 KEY0 来实现强制触摸屏校准。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/ TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键 ：KEY0 – PE2
5）触摸屏(电阻式/电容式)

6）AT24C02

3. 原理图

LCD 模块的触摸屏（电阻触摸屏）总共有 5 根线与 STM32F407 连接，连接电路图如下图所示：

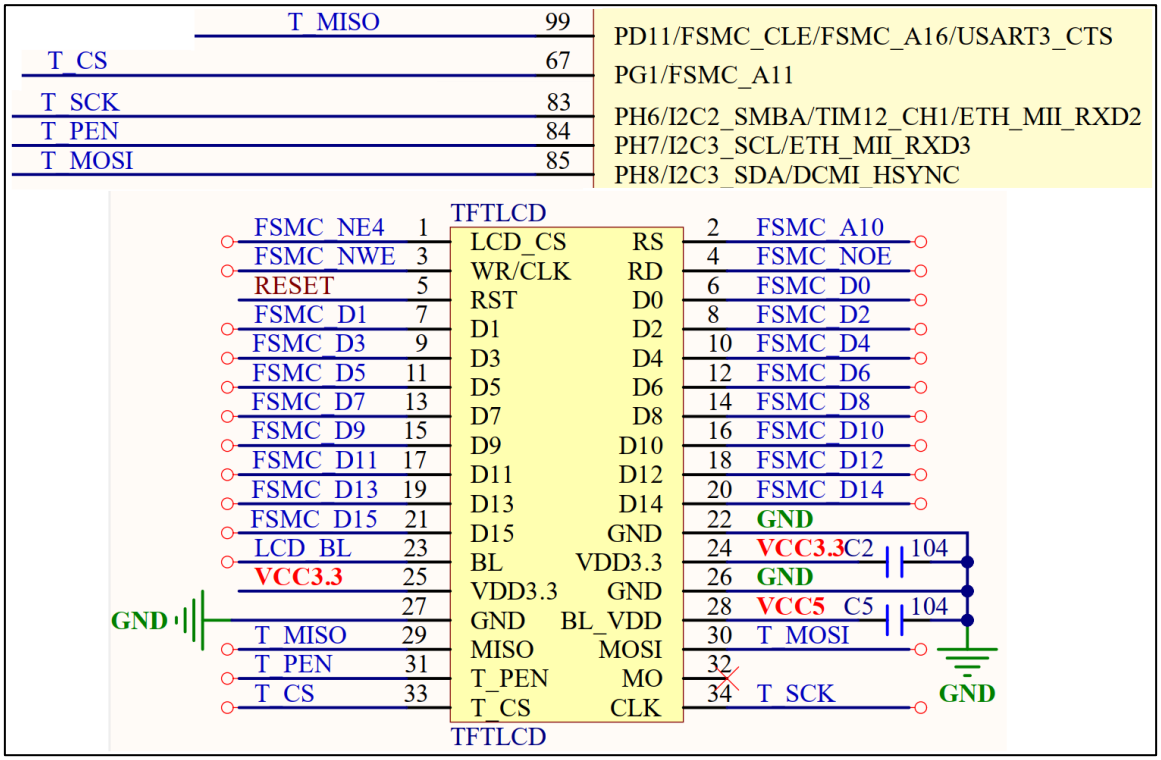


图 31.2.1 触摸屏与 STM32F407 的连接图

从图中可以看出， T_MISO、T_CS、T_SCK、T_PEN 和 T_MOSI 分别连接在 STM32F407 的：PD11、PG1、PH6、PH7 和 PH8 上。

如果是电容式触摸屏，我们的接口和电阻式触摸屏一样，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 GT9147/FT5206 的：中断输出信号、复位信号，IIC 的 SCL 和 SDA 信号。这里，我们用查询的方式读取 GT9147/FT5206 的数据，对于 FT5206 没有用到中断信号（CT_INT），所以同 STM32F407 的连接，最少只需要 3 根线即可，不过 GT9147 还需要用到 CT_INT 做 IIC 地址设定，所以需要 4 根线连接。

31.3 程序设计

31.3.1 HAL 库驱动

触摸芯片我们使用到的是 IIC 和 SPI 的驱动，这部分的内容可以参考之前 IIC/SPI 的章节，我们直接使用的是软件模拟的方式，所以只需要使用 HAL 库的驱动的 GPIO 操作部分。

触摸 IC 初始化步骤

1）使能连接在触摸 IC 的 GPIO 时钟，以及配置 GPIO 工作模式（速度，上下拉，模式等）

使能 GPIOx 时钟方法如下：

```
__HAL_RCC_GPIOx_CLK_ENABLE();           /* 使能 GPIOx 时钟，x=A……K */
```

GPIO 模式设置通过调用 HAL_GPIO_Init 函数实现，详见本例程源码。

2）编写读写函数

通过参考时序图，在一个时钟周期内发送 1bit 数据或者读取 1bit 数据。读写函数均以一字

节数据进行操作。

3）参考触摸 IC 时序图，编写触摸 IC 驱动函数（读和写函数）

在软件模拟 IIC 或者 SPI 的基础上，编写触摸 IC 的读写函数，详见本例程源码。

4）编写测试程序实现触摸效果（电阻触摸和电容触摸）

对触摸 IC 的寄存器读取坐标并调用 LCD 画点函数进行显示。

31.3.2 程序流程图

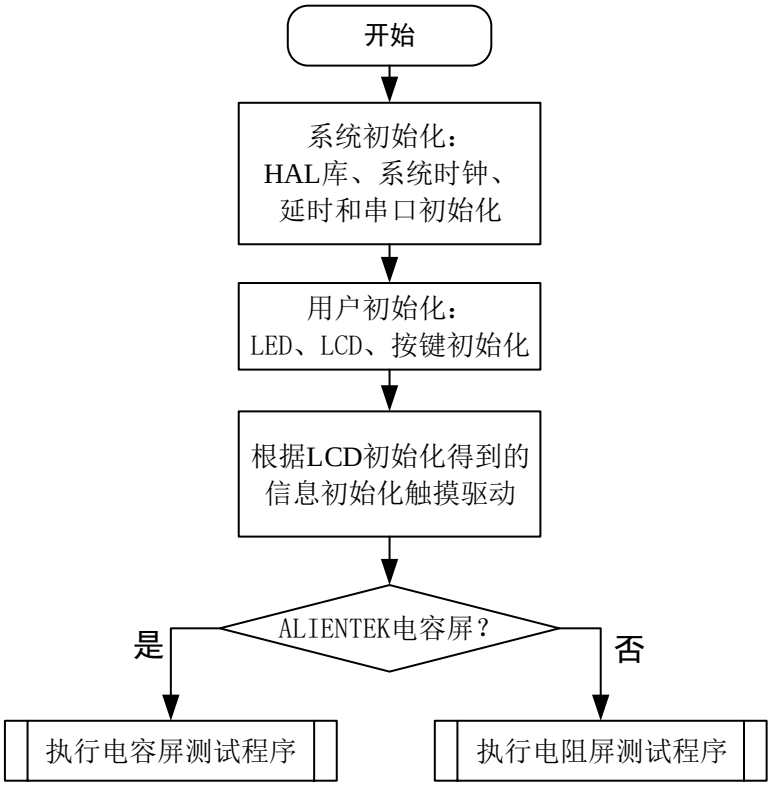


图 31.3.2.1 触摸测试实验程序流程图

31.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。TOUCH 驱动源码包括八个文件：ctiic.c、ctiic.h 、ft5206.c、ft5206.h、gt9xxx.c、gt9xxx.h、touch.c 和 touch.h。

由于正点原子的 TFTLCD 的型号很多，触摸控制这部分驱动代码根据不同屏幕搭载的触摸芯片驱动而有不同，在我们的屏幕上使用的是 LCD ID 来帮助软件上作驱分。为了解决多种驱动芯片的问题，我们设计了 touch.c/touch.h 这两个文件统一管理各类型的驱动。不同的驱动芯片类型可以在 touch.c 中集中添加，并通过 touch.c 中的接口统一调用，不同的触摸芯片各自编写独立的.c/.h 文件，需要时被 touch.c 调用。

1. 触摸管理 touch.c/touch.h

因为需要支持的触摸驱动比较多，为了方便管理和添加新的驱动，我们用 touch.c 文件来统一管理这些触摸驱动，然后针对各类触摸芯片编写独立的驱动。为了方便管理触摸，我们定义一个用于管理触摸信息的结构体类型。在 touch.h 中定义下面的结构体：

```
/* 触摸屏控制器 */
typedef struct
{
    uint8_t (*init)(void);      /* 初始化触摸屏控制器 */
    uint8_t (*scan)(uint8_t); /* 扫描触摸屏.0,屏幕扫描;1,物理坐标; */
    void (*adjust)(void);       /* 触摸屏校准 */
```

```
    uint16_t x[CT_MAX_TOUCH]; /* 当前坐标 */
    uint16_t y[CT_MAX_TOUCH]; /* 电容屏有最多 10 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[9],y[9]存储第一次按下时的坐标 */

    uint16_t sta;           /* 笔的状态
                             * b15:按下 1/松开 0;
                             * b14:0,没有按键按下;1,有按键按下.
                             * b13~b10:保留
                             * b9~b0:电容触摸屏按下的点数(0,表示未按下,1 表示按下)
                             */

    /* 5 点校准触摸屏校准参数(电容屏不需要校准) */
    float xfac;               /* 5 点校准法 x 方向比例因子 */
    float yfac;               /* 5 点校准法 y 方向比例因子 */
    short xc;                 /* 中心 X 坐标物理值(AD 值) */
    short yc;                 /* 中心 Y 坐标物理值(AD 值) */

    /* 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
     */
    uint8_t touchtype;
} _m_tp_dev;

extern _m_tp_dev tp_dev;        /* 触屏控制器在 touch.c 里面定义 */
```

我们希望能方便地调用不同触摸芯片的坐标扫描函数，这里我们定义了一个函数指针 uint8_t (*scan) (uint8_t)，我们只要把相应的芯片的初始化函数指针赋值给它，就可以使用这个通用接口方便地调用不同芯片的描述函数得到相应的触点坐标参数。

接下来是触摸初始化的核心程序，我们根据前面介绍的知识点，可以知道触摸的参数与屏幕大小和使用的触摸芯片有关，为了兼容 ALIENTK 的所有 LCD 屏幕的触摸驱动，我们会根据我们屏幕设计时搭载的触摸芯片（单独下载我们的 LCD 液晶屏对应型号的说明资料），调用对应的触摸初始化程序接口，根据前面定义的结构体我们希望坐标扫描函数为 xxx_scan()，对应不同的屏幕：

1、我们使用的是4.3寸电容屏时，我们使用的是汇顶科技的GT9271 作为电容触摸屏的驱动IC，这是一个IIC接口的驱动芯片，我们要编写这个芯片的gt9xxx系列芯片的初始化程序，并编写一个坐标扫描程序，这里我们先预留这两个接口分别为gt9xxx_init()和gt9xxx_scan()，在gt9xxx.c文件中再专门实现这两个驱动，标记使用的为电容屏。

2、类似地，当我们使用的是SSD1963 7寸屏、7寸800*480/1024*600 RGB屏时，我们的屏幕搭载的触摸驱动芯片是ft5206，预留这两个接口分别为ft5206_init()和ft5206_scan()，在ft5206.c文件中再专门实现这两个驱动，标记使用的为电容屏。

3、当为其它ID时，默认为电阻屏，我们默认使用的是SPI接口驱动的XPT2046芯片，因为屏幕尺寸不同，所以需要根据屏幕来校准电阻屏的AD值与实际的屏幕坐标，因此我们会需要一组校准参数，为了避免每次都要进行校准的麻烦，我们使用AT24C02来存储我们的校准信息。我们同样希望编写一个用于扫描电阻屏坐标的函数，我们的程序定义为tp_scan()。

根据以上的描述，SPI 的驱动接口前面我们介绍过，这里只介绍 touch.c 的触摸初始化函数，这里为了使代码更通用，我们使用软件 IIC 和软件 SPI 实现相应的 IO 口控制，触摸初始化函数 tp_init 编写如下：

```
/**
 * @brief       触摸屏初始化
 * @param       无
 * @retval      0,没有进行校准
 *              1,进行过校准
```

```
 */
uint8_t tp_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    tp_dev.touchtype = 0;                    /* 默认设置(电阻屏 & 竖屏) */
    tp_dev.touchtype |= lcddev.dir & 0X01;  /* 根据 LCD 判定是横屏还是竖屏 */

    /* 3.5 寸屏有两种，一种屏幕 ID 为 0x5510 带电阻触摸屏，
       一种屏幕 ID 为 0x7796 带 GT 型号的电容触摸屏 */
    if (lcddev.id == 0x7796)
    {
        if (gt9xxx_init() == 0) /* 初始化 GT 系列触摸屏成功,即当前 3.5 寸屏为电容触摸屏 */
        {
            tp_dev.scan = gt9xxx_scan;       /* 扫描函数指向 GT9147 触摸屏扫描 */
            tp_dev.touchtype |= 0X80;        /* 电容屏 */
            return 0;
        }
    }

    if (lcddev.id == 0X5510 || lcddev.id == 0X9806 || lcddev.id == 0X4342 ||
lcddev.id == 0X4384 || lcddev.id == 0X1018)  /* 电容触摸屏,4.3 寸/10.1 寸屏 */
    {
        gt9xxx_init();
        tp_dev.scan = gt9xxx_scan;  /* 扫描函数指向 GT9147 触摸屏扫描 */
        tp_dev.touchtype |= 0X80;   /* 电容屏 */
        return 0;
    }
    /* SSD1963 7 寸屏或者 7 寸 800*480/1024*600 RGB 屏 */
    else if (lcddev.id == 0X1963 || lcddev.id == 0X7084 || lcddev.id == 0X7016)
    {
        /* 触摸 IC 是 FT 系列的就执行 ft5206_init 函数以及使用 ft5206_scan 扫描函数 */
        if (!ft5206_init())
        {
            tp_dev.scan = ft5206_scan;  /* 扫描函数指向 FT5206 触摸屏扫描 */
        }
        else /* 触摸 IC 是 GT 系列的就执行 gt9xxx_init 函数以及使用 gt9xxx_scan 扫描函数 */
        {
            gt9xxx_init();
            tp_dev.scan = gt9xxx_scan;  /* 扫描函数指向 GT9147 触摸屏扫描 */
        }
        tp_dev.touchtype |= 0X80;       /* 电容屏 */
        return 0;
    }
    else
    {
        T_PEN_GPIO_CLK_ENABLE();    /* T_PEN 脚时钟使能 */
        T_CS_GPIO_CLK_ENABLE();     /* T_CS 脚时钟使能 */
        T_MISO_GPIO_CLK_ENABLE();   /* T_MISO 脚时钟使能 */
        T_MOSI_GPIO_CLK_ENABLE();   /* T_MOSI 脚时钟使能 */
        T_CLK_GPIO_CLK_ENABLE();    /* T_CLK 脚时钟使能 */

        gpio_init_struct.Pin = T_PEN_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_INPUT;              /* 输入 */
        gpio_init_struct.Pull = GPIO_PULLUP;                  /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;   /* 高速 */
        HAL_GPIO_Init(T_PEN_GPIO_PORT, &gpio_init_struct);    /* 初始化 T_PEN 引脚 */

        gpio_init_struct.Pin = T_MISO_GPIO_PIN;
        HAL_GPIO_Init(T_MISO_GPIO_PORT, &gpio_init_struct); /* 初始化 T_MISO 引脚 */

        gpio_init_struct.Pin = T_MOSI_GPIO_PIN;
```

```
        gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;        /* 推挽输出 */
        gpio_init_struct.Pull = GPIO_PULLUP;                /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH; /* 高速 */
        HAL_GPIO_Init(T_MOSI_GPIO_PORT, &gpio_init_struct); /* 初始化 T_MOSI 引脚 */

        gpio_init_struct.Pin = T_CLK_GPIO_PIN;
        HAL_GPIO_Init(T_CLK_GPIO_PORT, &gpio_init_struct);  /* 初始化 T_CLK 引脚 */

        gpio_init_struct.Pin = T_CS_GPIO_PIN;
        HAL_GPIO_Init(T_CS_GPIO_PORT, &gpio_init_struct);   /* 初始化 T_CS 引脚 */

        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_adjust_data())
        {
            return 0;           /* 已经校准 */
        }
        else                    /* 未校准? */
        {
            lcd_clear(WHITE);   /* 清屏 */
            tp_adjust();        /* 屏幕校准 */
            tp_save_adjust_data();
        }

        tp_get_adjust_data();
    }

    return 1;
}
```

正点原子的电容屏在出厂时已经由厂家较对好参数了，而电阻屏由于工艺和每个屏的线性线有所差异，我们需要先对其进行“校准”，我们在下一点补充说明它的实现。

通过上面的触摸初始化后，我们就可以读取相关的触点信息用于显示编程了，注意到上面还有很多个函数还没实现，比如读取坐标和校准，我们在接下来的代码中将它补充完整。

2. 电阻屏触摸函数

前面我们介绍过了电阻式触摸屏的原理，由于电阻屏的驱动代码都比较类似，我们决定把电阻屏的驱动函数直接添加在 touch.c/touch.h 中实现。

通过前面的电阻式触摸屏的分析，我们大概知道了电阻屏的驱动主要通过计算 X、Y 轴的 AD 值，然后换算成对应的屏幕坐标，在 touch.c 的初始化函数中我们已经对使用到的 SPI 接口 IO 进行了初始化。接下来我们介绍一下获取触摸点在屏幕上坐标的算法：先获取逻辑坐标（AD 值），再转换成屏幕坐标。

在不研究触点压力的情况下，我们知道一次触摸会同时产生 XY 坐标的 AD 值，根据 XTP2046 手册的原理，我们可以通过 SPI 时序一次把这两个 AD 值读出来。由于 XPT2046 采用 8 位控制命令来标记需要转换的地址，为了方便，我们用软件 SPI 编写一个 tp_write_byte()函数单独用于发送这 8 位命令（这部分时序参考我们软件 SPI 的算法和源码即可）；设置命令后需要 13 个时钟完成 AD 转换，第一位为 BUZY，最后一位在第 13 个时钟输出，且只有高 12 位的数据有效，于是可以得到转换函数 tp_read_ad()如下：

```
/**
 * @brief       SPI 读数据
 *   @note      从触摸屏 IC 读取 adc 值
 * @param       cmd: 指令
 * @retval      读取到的数据,ADC 值(12bit)
 */
static uint16_t tp_read_ad(uint8_t cmd)
{
    uint8_t count = 0;
    uint16_t num = 0;
```

```
    T_CLK(0);             /* 先拉低时钟 */
    T_MOSI(0);            /* 拉低数据线 */
    T_CS(0);              /* 选中触摸屏 IC */
    tp_write_byte(cmd);   /* 发送命令字 */
    delay_us(6);          /* ADS7846 的转换时间最长为 6us */
    T_CLK(0);
    delay_us(1);
    T_CLK(1);             /* 给 1 个时钟，清除 BUSY */
    delay_us(1);
    T_CLK(0);

    for (count = 0; count < 16; count++)    /* 读出 16 位数据,只有高 12 位有效 */
    {
        num <<= 1;
        T_CLK(0);         /* 下降沿有效 */
        delay_us(1);
        T_CLK(1);
        if (T_MISO)num++;
    }
    num >>= 4;            /* 只有高 12 位有效. */
    T_CS(1);              /* 释放片选 */
    return num;
}
```

一次读取的误差会很大，我们采用平均值滤波的方法，多次读取数据并丢弃波动最大的最大和最小值，余下的部分平均值。

```
/* 电阻触摸驱动芯片 数据采集 滤波用参数 */
#define TP_READ_TIMES   5       /* 读取次数 */
#define TP_LOST_VAL     1       /* 丢弃值 */

/**
 * @brief       读取一个坐标值(x 或者 y)
 *   @note      连续读取 TP_READ_TIMES 次数据,对这些数据升序排列,
 *              然后去掉最低和最高 TP_LOST_VAL 个数, 取平均值
 *              设置时需满足: TP_READ_TIMES > 2*TP_LOST_VAL 的条件
 *
 * @param       cmd : 指令
 *   @arg       0xD0: 读取 X 轴坐标(@竖屏状态,横屏状态和 Y 对调.)
 *   @arg       0x90: 读取 Y 轴坐标(@竖屏状态,横屏状态和 X 对调.)
 *
 * @retval      读取到的数据(滤波后的), ADC 值(12bit)
 */
static uint16_t tp_read_xoy(uint8_t cmd)
{
    uint16_t i, j;
    uint16_t buf[TP_READ_TIMES];
    uint16_t sum = 0;
    uint16_t temp;

    for (i = 0; i < TP_READ_TIMES; i++)      /* 先读取 TP_READ_TIMES 次数据 */
    {
        buf[i] = tp_read_ad(cmd);
    }
    for (i = 0; i < TP_READ_TIMES - 1; i++)  /* 对数据进行排序 */
    {
        for (j = i + 1; j < TP_READ_TIMES; j++)
        {
            if (buf[i] > buf[j])    /* 升序排列 */
            {
                temp = buf[i];
                buf[i] = buf[j];
                buf[j] = temp;
            }
```

```
        }
    }
    sum = 0;
    for (i = TP_LOST_VAL; i < TP_READ_TIMES - TP_LOST_VAL; i++)/*去掉两端的丢弃值*/
    {
        sum += buf[i];    /* 累加去掉丢弃值以后的数据. */
    }
    temp = sum / (TP_READ_TIMES - 2 * TP_LOST_VAL);    /* 取平均值 */
    return temp;
}
```

有了前述代码，我们就可以通过 tp_read_xoy(uint8_t cmd)接口调取需要的 x 或者 y 坐标的 AD 值了。这里我们加上横屏或者竖屏的处理代码，编写一个可以通过指针一次得到 x 和 y 的两个 AD 值的接口，代码如下：

```
/**
 * @brief       读取 x, y 坐标
 * @param       x,y: 读取到的坐标值
 * @retval      无
 */
static void tp_read_xy(uint16_t *x, uint16_t *y)
{
    uint16_t xval, yval;

    if (tp_dev.touchtype & 0x01)    /* X,Y 方向与屏幕相反 */
    {
        xval = tp_read_xoy(0x90);   /* 读取 X 轴坐标 AD 值, 并进行方向变换 */
        yval = tp_read_xoy(0xD0);   /* 读取 Y 轴坐标 AD 值 */
    }
    else    /* X,Y 方向与屏幕相同 */
    {
        xval = tp_read_xoy(0xD0);   /* 读取 X 轴坐标 AD 值 */
        yval = tp_read_xoy(0x90);   /* 读取 Y 轴坐标 AD 值 */
    }

    *x = xval;
    *y = yval;
}
```

为了进一步保证参数的精度，我们连续读两次触摸数据并取平均值作为最后的触摸参数，并对这两次滤波值平均后再传给目标存储区，由于 AD 的精度为 12 位，故该函数读取坐标的值 0~4095，tp_read_xy2 的代码如下：

```
/* 连续两次读取 X,Y 坐标的数据误差最大允许值 */
#define TP_ERR_RANGE    50      /* 误差范围 */

/**
 * @brief       连续读取 2 次触摸 IC 数据, 并滤波
 *   @note      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.
 *
 * @param       x,y : 读取到的坐标值
 * @retval      0, 失败; 1, 成功;
 */
static uint8_t tp_read_xy2(uint16_t *x, uint16_t *y)
{
    uint16_t x1, y1;
    uint16_t x2, y2;

    tp_read_xy(&x1, &y1);   /* 读取第一次数据 */
    tp_read_xy(&x2, &y2);   /* 读取第二次数据 */

    /* 前后两次采样在+-TP_ERR_RANGE 内 */
    if (((x2 <= x1 && x1 < x2 + TP_ERR_RANGE) ||
         (x1 <= x2 && x2 < x1 + TP_ERR_RANGE))&&
```

```
        ((y2 <= y1 && y1 < y2 + TP_ERR_RANGE) ||
        (y1 <= y2 && y2 < y1 + TP_ERR_RANGE)))
    {
        *x = (x1 + x2) / 2;
        *y = (y1 + y2) / 2;
        return 1;
    }

    return 0;
}
```

根据以上的流程，可以得到电阻屏触摸点的 AD 信息。每次触摸屏幕时会对应一组 X、Y 的 AD 值，由于坐标的 AD 值是在 X、Y 方向都是线性的，很容易想到要把触摸信息的 AD 值和屏幕坐标联系起来，这里需要编写一个坐标转换函数，前面在编写初始化接口时讲到的校准函数这时候就派上用场了。

从前面的知识我们就知道触摸屏的 AD 的 X_{AD}、Y_{AD} 可以构成一个逻辑平面，LCD 屏的屏幕坐标 X、Y 也是一个逻辑平面，由于存在误差，这两个平面并不重合，校准的作用就是要将逻辑平面映射到物理平面上，即得到触点在液晶屏上的位置坐标。校准算法的中心思想也就是要建立这样一个映射函数现有的校准算法大多是基于线性校准，即首先假定物理平面和逻辑平面之间的误差是线性误差，由旋转和偏移形成。

常用的电阻式触摸屏校正方法有两点校准法和三点校准法。本文这里介绍的是结合了不同的电阻式触摸屏校正法的优化算法：五点校正法。其中主要的原理是使用 4 点校正法的比例运算以及三点校正法的基准点运算。五点校正法优势在于可以更加精确的计算出 X 和 Y 方向的比例缩放系数，同时提供了中心基准点，对于一些线性电阻系数比较差电阻式触摸屏有很好的校正功能。校正相关的变量主要有：

- x［5］，y［5］五点定位的物理坐标（LCD 坐标）
- xl［5］，yl［5］五点定位的逻辑坐标（触摸 AD 值）
- KX，KY 横纵方向伸缩系数
- XLC，YLC 中心基点逻辑坐标
- XC，YC 中心基点物理坐标（数值采用 LCD 显示屏的物理长宽分辨率的一半）

x［5］，y［5］五点定位的物理坐标是已知的，其中 4 点分别设置在 LCD 的角落，一点设置在 LCD 正中心，作为基准校正点，校正关键点和距离布局如图 31.3.1。

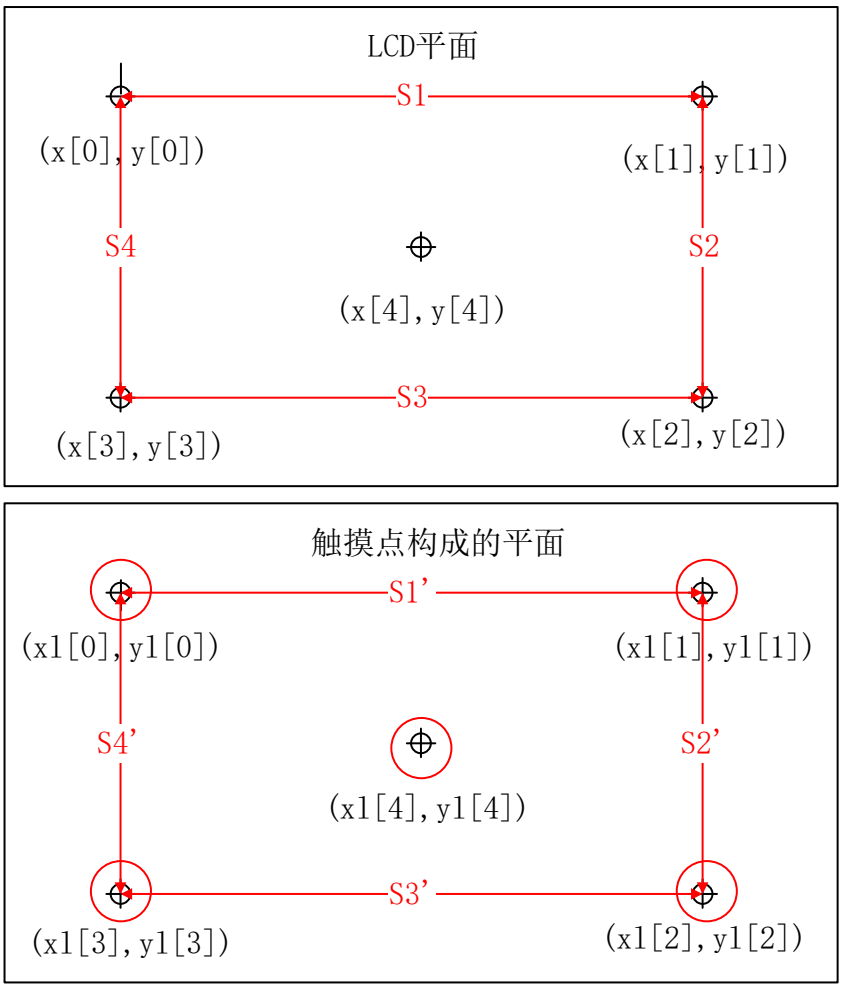


图 31.3.1 电阻屏五点校准法的参考点设定

校正步骤如下：

1. 通过先后点击 LCD 的 4 个角落的校正点，获取 4 个角落的逻辑坐标值。
2. 计算屏幕坐标和四点间距：

```
S1=x[1]-x[0]
S3=x[2]-x[3]
S2=y[2]-y[1]
S4=y[3]-y[0]
```

一般取点可以人为的设定 S1=S3 和 S2=S4，以方便运算。

计算逻辑坐标的四点“间距”，由于实际触点肯定会存在误差，所以触摸点会落在实际设定点的更大范围内，在图 31.3.1 中，设定点为五个点⊕，但实际采样时触点有时会落在稍大的外圈范围，图中用红色的圆圈标注了，所以有必要设定一个误差范围：

```
S1' = xl[1]-xl[0]
S3' = xl[2]-xl[3]
S2' = yl[2]-yl[1]
S4' = yl[3]-yl[0]
```

由于触点的误差，对于逻辑点 S1’和 S3’则大概率不会相等，同样的，S2’和 S4’也很验取到相等的点，那么为了简化计算，我们强制以（S1’+S3’）/2 的线长作一个矩形一边，以（S2’+S4’）/2 为矩形另一边，这样构建的矩形在误差范围是可以接受的，也方便计算，于是得到 X 和 Y 方向的近似缩放系数：

```
KX=(S1'+S3')/2/S1
KY=(S2'+S4')/2/S2
```

3. 点击 LCD 正中心，获取中心点的逻辑坐标，作为校正的基准点。这里也同样的需要限制误差，之后可以得到一个中心点的 AD 值坐标(xl[4],yl[4])，这个点的 AD 值我们就作为我们对比

的基准点，即 xl[4]=XLC，yl[4]=YLC；

4. 完成以上步骤则校正完成。下次点击触摸屏的时候获取的逻辑值 XL 和 YL，便可以按下以公式转换为物理坐标：

```
X=（XL-XLC）/KX+XC
Y=（YL-YLC）/KY+YC
```

最后一步的转换公式可能不好理解，大家换个角度，如果我们求到的缩放比例是正确的，在取新的触摸的时候，这个触摸点的逻辑坐标和物理坐标的转换，必然与中心点在两方向上的缩放比例相等，用中学数学直线斜率相等的情况，变换便可得到上述公式。

通过上述得到校准参数后，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，得到的就是触摸点的屏幕坐标。为了省去每次都需要校准的麻烦，我们保存这些参数到 AT24Cxx 的指定扇区地址，这样只要校准一次就可以重复使用这些参数了。

根据上面的原理，我们设计的校准函数 tp_adjust 如下：

```
/**
 * @brief       触摸屏校准代码
 *   @note      使用五点校准法(具体原理请百度)
 *              本函数得到 x 轴/y 轴比例因子 xfac/yfac 及物理中心坐标值(xc,yc)等 4 个参数
 *              我们规定：物理坐标即 AD 采集到的坐标值,范围是 0~4095.
 *                      逻辑坐标即 LCD 屏幕的坐标，范围为 LCD 屏幕的分辨率.
 *
 * @param       无
 * @retval      无
 */
void tp_adjust(void)
{
    uint16_t pxy[5][2];      /* 物理坐标缓存值 */
    uint8_t  cnt = 0;
    short s1, s2, s3, s4;    /* 4 个点的坐标差值 */
    double px, py;           /* X,Y 轴物理坐标比例,用于判定是否校准成功 */
    uint16_t outtime = 0;
    cnt = 0;

    lcd_clear(WHITE);        /* 清屏 */
    lcd_show_string(40, 40, 160, 100, 16, TP_REMIND_MSG_TBL, RED);/*显示提示信息*/
    tp_draw_touch_point(20, 20, RED);   /* 画点 1 */
    tp_dev.sta = 0;      /* 消除触发信号 */

    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 */
        /* 按键按下了一次(此时按键松开了.) */
        if ((tp_dev.sta & 0xc000) == TP_CATH_PRES)
        {
            outtime = 0;
            tp_dev.sta &= ~TP_CATH_PRES; /* 标记按键已经被处理过了. */
            pxy[cnt][0] = tp_dev.x[0];   /* 保存 X 物理坐标 */
            pxy[cnt][1] = tp_dev.y[0];   /* 保存 Y 物理坐标 */
            cnt++;
            switch (cnt)
            {
                case 1:
                    tp_draw_touch_point(20, 20, WHITE);                /* 清除点 1 */
                    tp_draw_touch_point(lcddev.width - 20, 20, RED); /* 画点 2 */
                    break;

                case 2:
                    tp_draw_touch_point(lcddev.width - 20, 20, WHITE);/* 清除点 2 */
                    tp_draw_touch_point(20, lcddev.height - 20, RED); /* 画点 3 */
                    break;
```

```
            case 3:
                tp_draw_touch_point(20, lcddev.height - 20, WHITE);/* 清除点 3*/
                tp_draw_touch_point(lcddev.width - 20, lcddev.height - 20,
                                    RED);    /* 画点 4 */
                break;

            case 4:
                lcd_clear(WHITE);   /* 画第五个点了，直接清屏 */
                /* 画点 5 */
                tp_draw_touch_point(lcddev.width / 2, lcddev.height / 2, RED);
                break;

            case 5:     /* 全部 5 个点已经得到 */
                /* 第 2 个点和第 1 个点的 X 轴物理坐标差值(AD 值) */
                s1 = pxy[1][0] - pxy[0][0];
                /* 第 4 个点和第 3 个点的 X 轴物理坐标差值(AD 值) */
                s3 = pxy[3][0] - pxy[2][0];
                /* 第 4 个点和第 2 个点的 Y 轴物理坐标差值(AD 值) */
                s2 = pxy[3][1] - pxy[1][1];
                /* 第 3 个点和第 1 个点的 Y 轴物理坐标差值(AD 值) */
                s4 = pxy[2][1] - pxy[0][1];
                px = (double)s1 / s3;       /* X 轴比例因子 */
                py = (double)s2 / s4;       /* Y 轴比例因子 */
                if (px < 0)px = -px;        /* 负数改正数 */
                if (py < 0)py = -py;        /* 负数改正数 */
                /* 比例不合格 */
                if (px < 0.95 || px > 1.05 || py < 0.95 || py > 1.05
                    || abs(s1) > 4095 || abs(s2) > 4095 || abs(s3) > 4095
                    || abs(s4) > 4095 ||  /* 差值不合格，大于坐标范围 */
                    abs(s1) == 0 || abs(s2) == 0 || abs(s3) == 0
                    || abs(s4) == 0 )     /* 差值不合格，等于 0 */
                {
                    cnt = 0;
                    tp_draw_touch_point(lcddev.width / 2, lcddev.height / 2,
                                        WHITE);       /* 清除点 5 */
                    tp_draw_touch_point(20, 20, RED); /* 重新画点 1 */
                    tp_adjust_info_show(pxy, px, py); /* 显示当前信息,方便找问题 */
                    continue;
                }
                tp_dev.xfac = (float)(s1 + s3) / (2 * (lcddev.width - 40));
                tp_dev.yfac = (float)(s2 + s4) / (2 * (lcddev.height - 40));
                tp_dev.xc = pxy[4][0];      /* X 轴,物理中心坐标 */
                tp_dev.yc = pxy[4][1];      /* Y 轴,物理中心坐标 */

                lcd_clear(WHITE);   /* 清屏 */
                lcd_show_string(35, 110, lcddev.width, lcddev.height, 16,
                                "Touch Screen Adjust OK!", BLUE);   /* 校正完成 */
                delay_ms(1000);
                tp_save_adjust_data();

                lcd_clear(WHITE);    /* 清屏 */
                return;    /* 校正完成 */
            }
        }
        delay_ms(10);
        outtime++;
        if (outtime > 1000)
        {
            tp_get_adjust_data();
            break;
        }
    }
}
```

注意该函数里面多次使用了 lcddev.width 和 lcddev.height，用于坐标设置，故在程序调用前需要预先初始化 LCD 得到 LCD 的一些屏幕信息，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）。

有了校准参数后，由于我们需要频繁地进行屏幕坐标和物理坐标的转换，我们为电阻屏增加一个 tp_scan(uint8_t mode)用于转换，为了实际使用上更灵活，我们使这个参数支持物理坐标和屏幕坐标，设计的函数如下：

```
/**
 * @brief       触摸按键扫描
 * @param       mode: 坐标模式
 *   @arg       0, 屏幕坐标;
 *   @arg       1, 物理坐标(校准等特殊场合用)
 *
 * @retval      0, 触屏无触摸; 1, 触屏有触摸;
 */
uint8_t tp_scan(uint8_t mode)
{
    if (T_PEN == 0)       /* 有按键按下 */
    {
        if (mode)         /* 读取物理坐标, 无需转换 */
        {
            tp_read_xy2(&tp_dev.x[0], &tp_dev.y[0]);
        }
        else if (tp_read_xy2(&tp_dev.x[0], &tp_dev.y[0]))/* 读取屏幕坐标,需要转换 */
        {
            /* 将 X 轴 物理坐标转换成逻辑坐标(即对应 LCD 屏幕上面的 X 坐标值) */
            tp_dev.x[0] = (signed short)(tp_dev.x[0] - tp_dev.xc)
                          / tp_dev.xfac + lcddev.width / 2;
            /* 将 Y 轴 物理坐标转换成逻辑坐标(即对应 LCD 屏幕上面的 Y 坐标值) */
            tp_dev.y[0] = (signed short)(tp_dev.y[0] - tp_dev.yc)
                                        / tp_dev.yfac + lcddev.height / 2;
        }

        if ((tp_dev.sta & TP_PRES_DOWN) == 0)    /* 之前没有被按下 */
        {
            tp_dev.sta = TP_PRES_DOWN | TP_CATH_PRES;    /* 按键按下 */
            tp_dev.x[CT_MAX_TOUCH - 1] = tp_dev.x[0];    /* 记录第一次按下时的坐标 */
            tp_dev.y[CT_MAX_TOUCH - 1] = tp_dev.y[0];
        }
    }
    else
    {
        if (tp_dev.sta & TP_PRES_DOWN)      /* 之前是被按下的 */
        {
            tp_dev.sta &= ~TP_PRES_DOWN;    /* 标记按键松开 */
        }
        else    /* 之前就没有被按下 */
        {
            tp_dev.x[CT_MAX_TOUCH - 1] = 0;
            tp_dev.y[CT_MAX_TOUCH - 1] = 0;
            tp_dev.x[0] = 0xffff;
            tp_dev.y[0] = 0xffff;
        }
    }

    return tp_dev.sta & TP_PRES_DOWN;    /* 返回当前的触屏状态 */
}
```

之后的使用，只要调取 tp_scan()函数，就能灵活地得到我们需要的触摸坐标用于显示交互的编程了，电阻屏的触摸就讲到这里。

3. 电容屏驱动 gt9xxxx.c/gt9xxxx.h

电容触摸芯片我们使用的是 IIC 接口的触摸 IC。IIC 接口部分代码，我们可以参考 myiic.c 和 myiic.h 的代码，为了使代码独立，我们在 Touch 文件夹下采用软件模拟 IIC 的方式实现 ctiic.c 和 ctiic.h，这样 IO 的使用更灵活，部分参考 IIC 章节，这里不再重复介绍了。

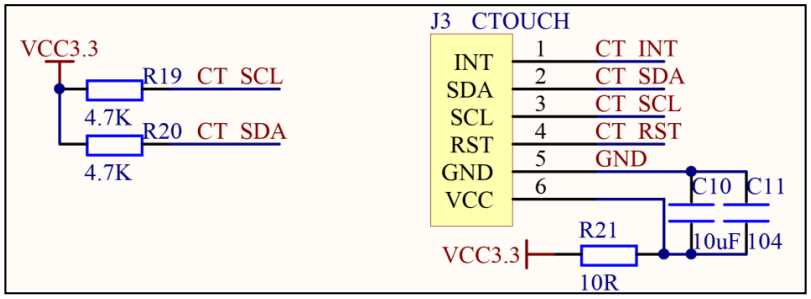


图 31.3.3.1 正点原子 LCD 上的 GT9XX 触摸芯片通讯接口

gt9xxx_init 的实现也比较简单，实现 CT_INT\CT_RST 引脚初始化，注意我们使 ctiic.c\ ctiic.h 的 IO 对应于 CT_SDA\CT_SCL，调用其 IIC 初始化接口即可。

同样地，我们需要通过 IIC 来读取触摸点的物理坐标，由于电容屏在设计时是根据屏幕进行参数设计的，参数已经保存在 gt9xxx 芯片的内部了，我们只需要按手册推荐的 IIC 时序把对应的 XY 坐标读出来，转换成 LCD 的像素坐标即可。与电阻屏不同的是，我们是通过 IIC 读取当前是否在触摸状态而非引脚电平。gt9xx 系列可以通过中断或轮询方式读写，我们使用的是轮询方式：

1、按第二节时序，先读取寄存器 0x814E，若当前 buffer（buffer status 为 1）数据准备好，则依据手指个数读、按键状态取相应个数的坐标、按键信息。

2、若在1中发现 buffer 数据（buffer status 为 0）未准备好，则等待 1ms 再进行读取。

gt9xxx_scan()函数的实现如下：

```
/* GT9XXX 10 个触摸点(最多) 对应的寄存器表 */
const uint16_t GT9XXX_TPX_TBL[10] =
{
    GT9XXX_TP1_REG, GT9XXX_TP2_REG, GT9XXX_TP3_REG, GT9XXX_TP4_REG,
    GT9XXX_TP5_REG, GT9XXX_TP6_REG, GT9XXX_TP7_REG, GT9XXX_TP8_REG,
    GT9XXX_TP9_REG, GT9XXX_TP10_REG,
};
/**
 * @brief       扫描触摸屏(采用查询方式)
 * @param       mode : 电容屏未用到次参数, 为了兼容电阻屏
 * @retval      当前触屏状态
 *   @arg       0, 触屏无触摸;
 *   @arg       1, 触屏有触摸;
 */
uint8_t gt9xxx_scan(uint8_t mode)
{
    uint8_t buf[4];
    uint8_t i = 0;
    uint8_t res = 0;
    uint16_t temp;
    uint16_t tempsta;
    static uint8_t t = 0;   /* 控制查询间隔,从而降低 CPU 占用率 */
    t++;
    /* 空闲时,每进入 10 次 CTP_Scan 函数才检测 1 次,从而节省 CPU 使用率 */
    if ((t % 10) == 0 || t < 10)
    {
        gt9xxx_rd_reg(GT9XXX_GSTID_REG, &mode, 1);  /* 读取触摸点的状态 */
        if ((mode & 0x80) && ((mode & 0xF) <= g_gt_tnum))
        {
            i = 0;
            gt9xxx_wr_reg(GT9XXX_GSTID_REG, &i, 1); /* 清标志 */
```

```
    }
    if ((mode & 0xF) && ((mode & 0xF) <= g_gt_tnum))
    {
        /* 将点的个数转换为 1 的位数,匹配 tp_dev.sta 定义 */
        temp = 0xFFFF << (mode & 0xF);
        tempsta = tp_dev.sta;           /* 保存当前的 tp_dev.sta 值 */
        tp_dev.sta = (~temp) | TP_PRES_DOWN | TP_CATH_PRES;
        /* 保存触点 0 的数据,保存在最后一个上 */
        tp_dev.x[g_gt_tnum - 1] = tp_dev.x[0];
        tp_dev.y[g_gt_tnum - 1] = tp_dev.y[0];
        for (i = 0; i < g_gt_tnum; i++)
        {
            if (tp_dev.sta & (1 << i))  /* 触摸有效? */
            {
                gt9xxx_rd_reg(GT9XXX_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] = ((uint16_t)buf[1] << 8) + buf[0];
                        tp_dev.x[i] = 800 - (((uint16_t)buf[3] << 8) + buf[2]);
                    }
                    else
                    {
                        tp_dev.x[i] = ((uint16_t)buf[1] << 8) + buf[0];
                        tp_dev.y[i] = ((uint16_t)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] = (((uint16_t)buf[1] << 8) + buf[0]);
                        tp_dev.y[i] = (((uint16_t)buf[3] << 8) + buf[2]);
                    }
                    else
                    {
                        tp_dev.y[i] = ((uint16_t)buf[1] << 8) + buf[0];
                        tp_dev.x[i] = 272 - (((uint16_t)buf[3] << 8) + buf[2]);
                    }
                }
                else if (lcddev.id == 0x1018)       /* 10.1 寸 1280*800 RGB 屏 */
                {
                    if (tp_dev.touchtype & 0x01)    /* 横屏 */
                    {
                        tp_dev.y[i] = ((uint16_t)buf[3] << 8) + buf[2];
                        tp_dev.x[i] = ((uint16_t)buf[1] << 8) + buf[0];
                    }
                    else
                    {
                        tp_dev.x[i] = 800 - (((uint16_t)buf[3] << 8) + buf[2]);
                        tp_dev.y[i] = ((uint16_t)buf[1] << 8) + buf[0];
                    }
                }
            }
        }
        res = 1;
        /* 非法数据(坐标超出了) */
        if (tp_dev.x[0] > lcddev.width || tp_dev.y[0] > lcddev.height)
        {
            if ((mode & 0xF) > 1)/*有其他点有数据,则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[g_gt_tnum - 1];
                tp_dev.y[0] = tp_dev.y[g_gt_tnum - 1];
                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 &= ~TP_PRES_DOWN;    /* 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 &= 0XE000;           /* 清除点有效标记 */
    }
}
if (t > 240)t = 10;                     /* 重新从 10 开始计数 */
return res;
}
```

大家可以打开 gt9xxx 芯片对应的编程手册，对照时序，即可理解上述的实现过程，只是程序中为了匹配多种屏幕和横屏显示，添加了一些代码。

电容屏驱动 ft5206.c/ft5206.h 的驱动实现与 gt9xxx 的实现类似，大家参考光盘中的源码对照着实现即可。

电容屏的触摸实验代码讲解到这里。

4. main 函数和测试代码

最后我们打开 main.c，修改部分代码，这里就不全部贴出来了，仅介绍三个重要的函数：

```
/**
 * @brief       电阻触摸屏测试函数
 * @param       无
 * @retval      无
 */
void rtp_test(void)
{
    uint8_t key;
    uint8_t 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_draw_dialog();    /* 清除 */
            }
            else
            {
                tp_draw_big_point(tp_dev.x[0], tp_dev.y[0], RED);    /* 画点 */
            }
        }
    }
    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_adjust_data();
        load_draw_dialog();
    }
    i++;
    if (i % 20 == 0)
    {
        LED0_TOGGLE();
    }
  }
}
/* 10 个触控点的颜色(电容触摸屏用) */
const uint16_t POINT_COLOR_TBL[10] = {RED, GREEN, BLUE, BROWN, YELLOW, MAGENTA,
CYAN, LIGHTBLUE, BRRED, GRAY};

/**
 * @brief       电容触摸屏测试函数
 * @param       无
 * @retval      无
 */
void ctp_test(void)
{
    uint8_t t = 0;
    uint8_t i = 0;
    uint16_t lastpos[10][2];        /* 最后一次的数据 */
    uint8_t maxp = 5;

    if (lcddev.id == 0x1018)maxp = 10;
    while (1)
    {
        tp_dev.scan(0);
        for (t = 0; t < maxp; t++)
        {
            if ((tp_dev.sta) & (1 << t))
            {
                /* 坐标在屏幕范围内 */
                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_draw_dialog();/* 清除 */
                }
            }
        }
        else
        {
            lastpos[t][0] = 0xFFFF;
        }
    }
    delay_ms(10);
    i++;
    if (i % 20 == 0)LED0_TOGGLE();
  }
}

int main(void)
{
  HAL_Init();                             /* 初始化 HAL 库 */
  sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
  delay_init(168);                        /* 延时初始化 */
  usart_init(115200);                     /* 串口初始化为 115200 */
  led_init();                             /* 初始化 LED */
  lcd_init();                             /* 初始化 LCD */
  key_init();                             /* 初始化按键 */
  tp_dev.init();                          /* 触摸屏初始化 */

  lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
  lcd_show_string(30, 70, 200, 16, 16, "TOUCH TEST", RED);
  lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
  if (tp_dev.touchtype != 0xFF)
  {
    /* 电阻屏才显示 */
    lcd_show_string(30, 110, 200, 16, 16, "Press KEY0 to Adjust", RED);
  }
  delay_ms(1500);
  load_draw_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。

31.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。电阻触摸屏测试如图 31.4.1 所示界面：



图 31.4.1 电阻触摸屏测试程序运行效果

图中我们在电阻屏上画了一些内容，右上角的 RST 可以用来清屏，点击该区域，即可清屏重画。另外，按 KEY0 可以进入校准模式，如果发现触摸屏不准，则可以按 KEY0，进入校准， 重新校准一下，即可正常使用。

如果是电容触摸屏，测试界面如图 31.4.2 所示：

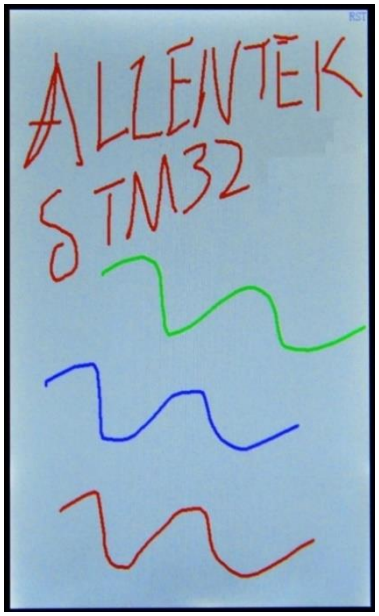


图 31.4.2 电容触摸屏测试界面

图中，同样输入了一些内容。电容屏支持多点触摸，每个点的颜色都不一样，图中的波浪线就是三点触摸画出来的，最多可以 5 点触摸。注意：电容触摸屏支持：正点原子 4.3 寸电容触摸屏模块或者正点原子新款 7 寸电容触摸屏模块（SSD1963+FT5206 方案），老款的 7 寸电容触摸屏模块（CPLD+GT811 方案）本例程不支持!!

同样，按右上角的 RST 标志，可以清屏。电容屏无需校准，所以按 KEY0 无效。KEY0 校准仅对电阻屏有效。

第三十二章 DS18B20 数字温度传感器实验

本章，我们将介绍 STM32 如何读取外部温度传感器的温度，来得到较为准确的环境温度。我们将学习单总线技术，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 上。

本章分为如下几个小节：

32.1 DS18B20 及其时序简介
32.2 硬件设计
32.3 程序设计
32.4 下载验证

32.1 DS18B20 及其时序简介

32.1.1 DS18B20 简介

DS18B20 是由 DALLAS 半导体公司推出的一种“单总线”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。单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的概念，测试温度范围为-55~+125℃，精度为±0.5℃。现场温度直接以单总线的数字方式传输，大大提高了系统的抗干扰性。它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~12 位的数字值读数方式。它工作在 3~5.5V 的电压范围，采用多种封装形式，从而使系统设置灵活、方便，设定分辨率以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 32.1.1 所示：

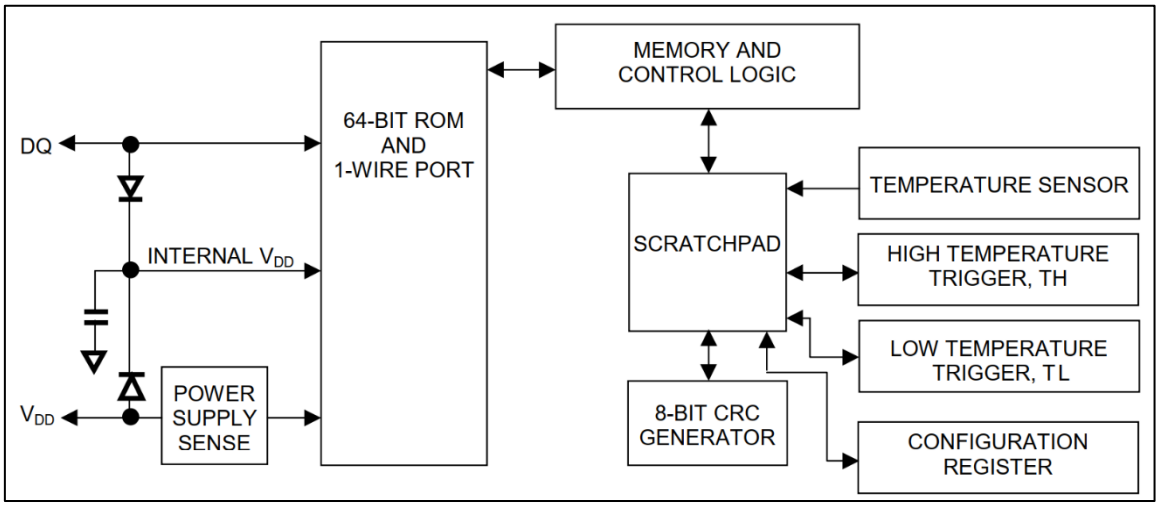


图 32.1.1.1 DS18B20 内部结构图

ROM 中的 64 位序列号是出厂前被标记好的，它可以看作使该 DS18B20 的地址序列码，每个 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，这样设计可以允许一根总线上挂载多个 DS18B20 模块同时工作且不会引起冲突。

32.1.2 DS18B20 时序简介

所有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，都是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

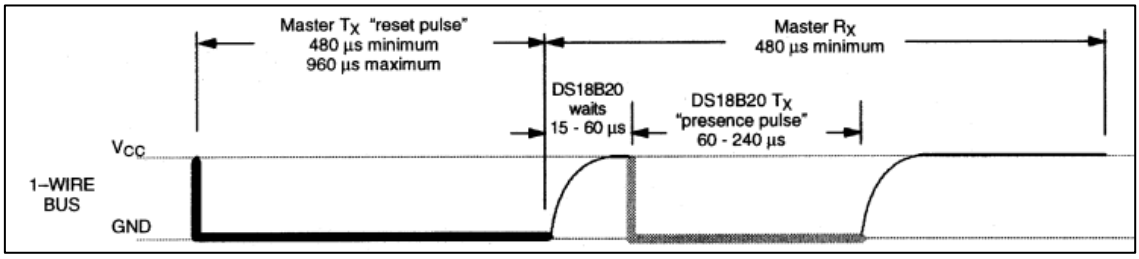


图 32.1.2.1 复位脉冲和应答脉冲时序图

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要在480us，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，延时时间要在15~60us，并进入接收模式（Rx）。接着 DS18B20 拉低总线 60~240us，以产生低电平应答脉冲。

2） 写时序

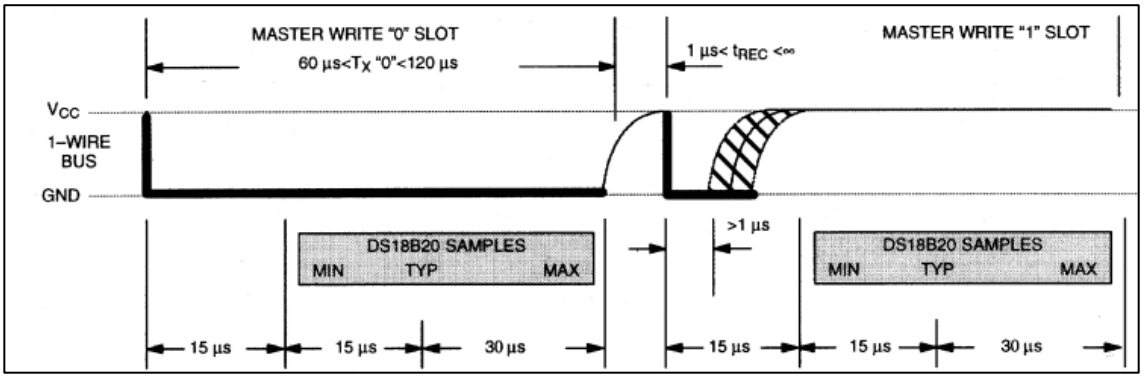


图 32.1.2.2 写时序图

写时序包括写 0 时序和写 1 时序。所有写时序至少需要 60us，且在两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

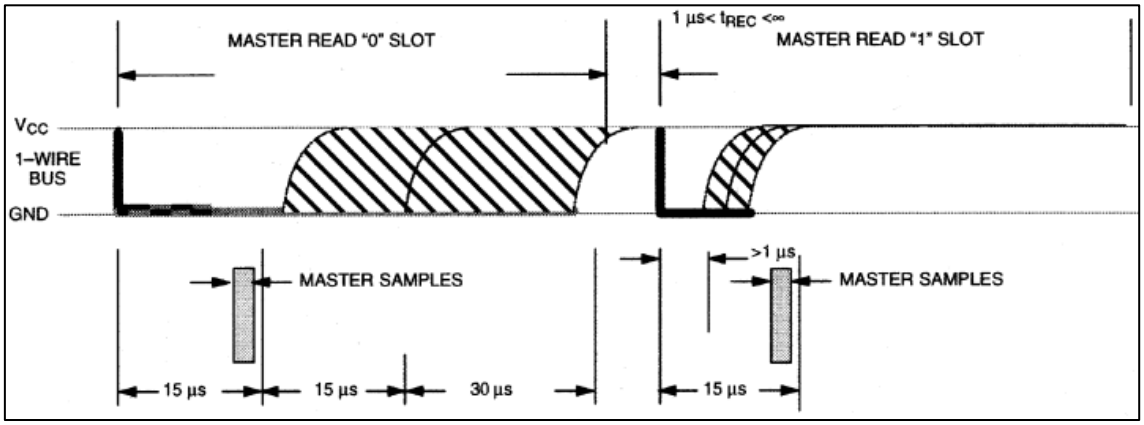


图 32.1.2.3 读时序图

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。

在了解单总线时序之后，我们来看一下 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（0xCC）→发开始转换命令（0x44）→延时→复位→发送 SKIP ROM 命令（0xCC）→发送存储器命令（0xBE）→连续读取两个字节数据（即温度）→结束。

32.2 硬件设计

1. 例程功能

本实验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。 LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）DS18B20 温度传感器(接在 PG15 上)

3. 原理图

DS18B20 接口与 STM32 的连接关系，如下图所示：

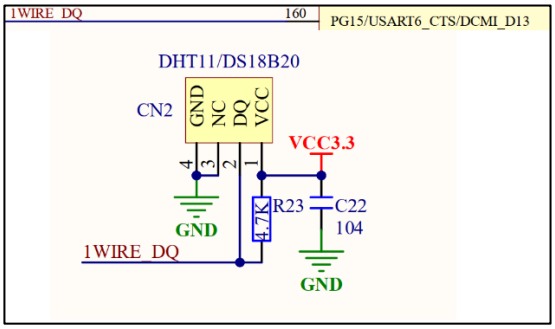


图 32.2.1 DS18B20 接口与 STM32 的连接电路图

从上图可以看出，我们使用的是 STM32 的 PG15 来连接 CN2 的 DQ 引脚，图中 CN2 为 DHT11（数字温湿度传感器）和 DS18B20 共用的一个接口，DHT11 我们将在下一章介绍。

DS18B20 只用到 CN2 的 3 个引脚（CN2 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 32.2.2 所示：

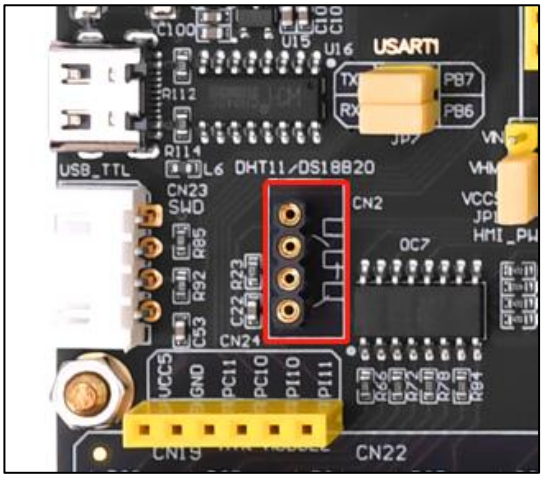


图 32.2.2 DS18B20 连接示意图

从上图可以看出，DS18B20 的平面部分（有字的那面）应该朝左，而曲面部分朝右。然后插入如图所示的三个孔内。

32.3 程序设计

DS18B20 实验中使用的是单总线协议，用到的是 HAL 中 GPIO 相关函数，前面也有介绍到，这里就不做展开了。下面介绍一下如何驱动 DS18B20。

DS18B20 配置步骤

1）使能 DS18B20 数据线对应的 GPIO 时钟。

本实验中 DS18B20 的数据线引脚是 PG15，因此需要先使能 GPIOG 的时钟，代码如下：

```
__HAL_RCC_GPIOG_CLK_ENABLE();       /* PG 口时钟使能 */
```

2）设置对应 GPIO 工作模式（开漏输出）

本实验 GPIO 使用开漏输出模式，通过函数 HAL_GPIO_Init 设置实现。

3）参考单总线协议，编写信号函数（复位脉冲、应答脉冲、写 0/1、读 0/1）

复位脉冲：主机发出低电平，保持低电平时间至少 480us。

应答脉冲：DS18B20 拉低总线 60~240us，以产生低电平应答信号。

写 1 信号：主机输出低电平，延时 2us，然后释放总线，延时 60us。

写 0 信号：主机输出低电平，延时 60us，然后释放总线，延时 2us。

读 0/1 信号：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。

4）编写 DS18B20 的读和写函数

基于写 1bit 数据和读 1bit 数据的基础上，编写 DS18B20 写 1 字节和读 1 字节函数。

5）编写 DS18B20 获取温度函数

参考 DS18B20 典型温度读取过程，编写获取温度函数。

32.3.1 程序流程图

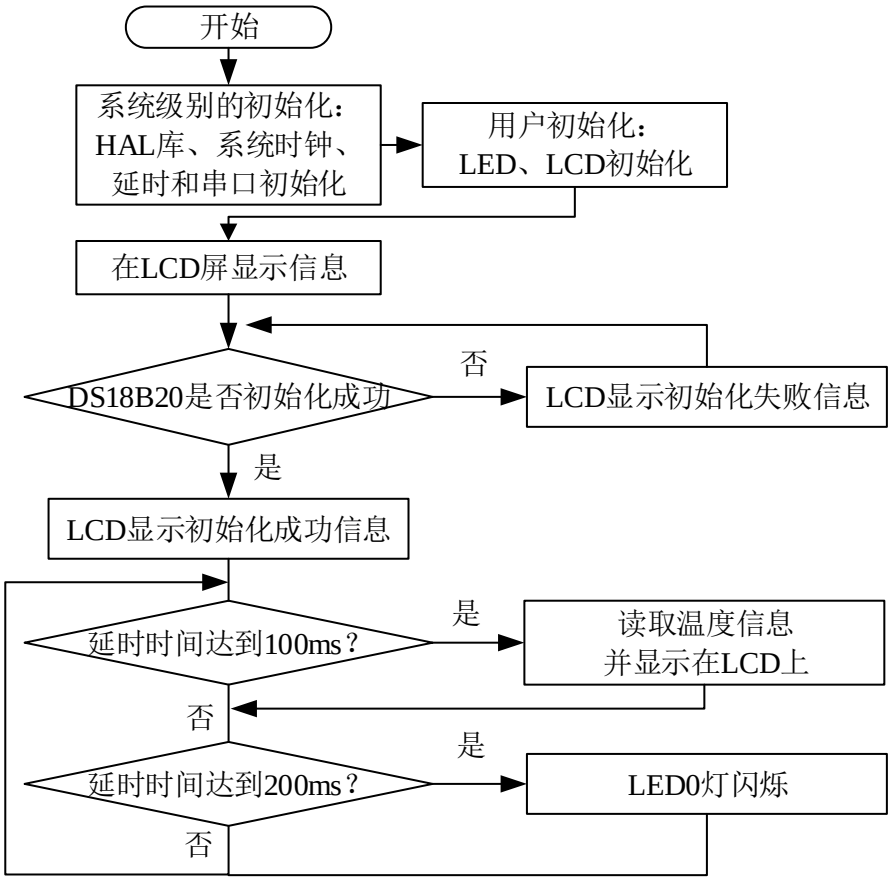


图 32.3.1.1 DS18B20 实验程序流程图

32.3.2 程序解析

1.DS18B20 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DS18B20 驱动源码包括两个文件：ds18b20.c 和 ds18b20.h。

首先我们先看一下 ds18b20.h 头文件里面的内容，其定义如下：

```
/* DS18B20 引脚 定义 */
#define DS18B20_DQ_GPIO_PORT                GPIOG
#define DS18B20_DQ_GPIO_PIN                 GPIO_PIN_15
#define DS18B20_DQ_GPIO_CLK_ENABLE()
            do{ __HAL_RCC_GPIOG_CLK_ENABLE(); }while(0)       /* PG 口时钟使能 */

/* IO 操作函数 */
#define DS18B20_DQ_OUT(x) do{ x ? \
                          HAL_GPIO_WritePin(DS18B20_DQ_GPIO_PORT,
                                            DS18B20_DQ_GPIO_PIN, GPIO_PIN_SET) : \
                          HAL_GPIO_WritePin(DS18B20_DQ_GPIO_PORT,
                                            DS18B20_DQ_GPIO_PIN, GPIO_PIN_RESET); \
                         }while(0)                                 /* 数据端口输出 */
#define DS18B20_DQ_IN     HAL_GPIO_ReadPin(DS18B20_DQ_GPIO_PORT,
                                           DS18B20_DQ_GPIO_PIN)     /* 数据端口输入 */
```

在 ds18b20.h 的操作跟 IIC 实验代码很类似，主要对用到 GPIO 口进行宏定义，以及宏定义 IO 操作函数，方便时序函数调用。

下面我们直接介绍 ds18b20.c 的程序，首先介绍的是 DS18B20 传感器的初始化函数，其定

义如下：

```
/**
 * @brief       初始化 DS18B20 的 IO 口 DQ 同时检测 DS18B20 的存在
 * @param       无
 * @retval      0, 正常
 *              1, 不存在/不正常
 */
uint8_t ds18b20_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    DS18B20_DQ_GPIO_CLK_ENABLE();   /* 开启 DQ 引脚时钟 */

    gpio_init_struct.Pin = DS18B20_DQ_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD;            /* 开漏输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;     /* 高速 */
    /* 初始化 DS18B20_DQ 引脚 */
    HAL_GPIO_Init(DS18B20_DQ_GPIO_PORT, &gpio_init_struct);
    /* DS18B20_DQ 引脚模式设置, 开漏输出,上拉, 这样就不用再设置 IO 方向了,
       开漏输出的时候(=1), 也可以读取外部信号的高低电平 */
    ds18b20_reset();
    return ds18b20_check();
}
```

在 ds18b20 的初始化函数中，主要对用到的 GPIO 口进行初始化，同时在函数最后调用复位函数和自检函数，这两个函数在后面会解释到。

下面介绍的是复位 DS18B20 函数和等待 DS18B20 的回应函数，它们的定义如下：

```
/**
 * @brief       复位 DS18B20
 * @param       data: 要写入的数据
 * @retval      无
 */
static void ds18b20_reset(void)
{
    DS18B20_DQ_OUT(0);  /* 拉低 DQ,复位 */
    delay_us(750);      /* 拉低 750us */
    DS18B20_DQ_OUT(1);  /* DQ=1, 释放复位 */
    delay_us(15);       /* 延迟 15US */
}

/**
 * @brief       等待 DS18B20 的回应
 * @param       无
 * @retval      0, DS18B20 正常
 *              1, DS18B20 异常/不存在
 */
uint8_t ds18b20_check(void)
{
    uint8_t retry = 0;
    uint8_t rval = 0;

    while (DS18B20_DQ_IN && retry < 200)        /* 等待 DQ 变低, 等待 200us */
    {
        retry++;
        delay_us(1);
    }

    if (retry >= 200)
    {
        rval = 1;
    }
    else
```

```
    {
        retry = 0;
        while (!DS18B20_DQ_IN && retry < 240)    /* 等待 DQ 变高，等待 240us */
        {
            retry++;
            delay_us(1);
        }
        if (retry >= 240) rval = 1;
    }
    return rval;
}
```

以上两个函数分别代表着前面所说的复位脉冲与应答信号，大家可以对比前面的时序图进行理解。由于复位脉冲比较简单，所以这里不做展开。现在看一下应答信号函数，函数主要是对于 DS18B20 传感器的回应信号进行检测，对此判断其是否存在。函数的实现也是依据时序图进行逻辑判断，例如当主机发送了复位信号之后，按照时序，DS18B20 会拉低数据线 60~240us，同时主机接收最小时间为 480us，我们就依据这两个硬性条件进行判断，首先需要设置一个时限等待 DS18B20 响应，后面也设置一个时限等待 DS18B20 释放数据线拉高，满足这两个条件即 DS18B20 成功响应。

下面介绍的是写函数，其定义如下：

```
/**
 * @brief       写一个字节到 DS18B20
 * @param       data: 要写入的字节
 * @retval      无
 */
static void ds18b20_write_byte(uint8_t data)
{
    uint8_t j;
    for (j = 1; j <= 8; j++)
    {
        if (data & 0x01)
        {
            DS18B20_DQ_OUT(0);  /*  Write 1 */
            delay_us(2);
            DS18B20_DQ_OUT(1);
            delay_us(60);
        }
        else
        {
            DS18B20_DQ_OUT(0);  /*  Write 0 */
            delay_us(60);
            DS18B20_DQ_OUT(1);
            delay_us(2);
        }
        data >>= 1;             /* 右移,获取高一位数据 */
    }
}
```

通过形参决定是写 1 还是写 0，按照前面对写时序的分析，我们可以很清晰知道写函数的逻辑处理。

有写函数肯定就有读函数，下面介绍的是读函数，其定义如下：

```
/**
 * @brief       从 DS18B20 读取一个位
 * @param       无
 * @retval      读取到的位值: 0 / 1
 */
static uint8_t ds18b20_read_bit(void)
{
    uint8_t data = 0;
    DS18B20_DQ_OUT(0);
    delay_us(2);
    DS18B20_DQ_OUT(1);
    delay_us(12);
```

```
    if (DS18B20_DQ_IN)
    {
        data = 1;
    }
    delay_us(50);
    return data;
}

/**
 * @brief       从 DS18B20 读取一个字节
 * @param       无
 * @retval      读到的数据
 */
static uint8_t ds18b20_read_byte(void)
{
    uint8_t i, b, data = 0;
    for (i = 0; i < 8; i++)
    {
        b = ds18b20_read_bit();  /* DS18B20 先输出低位数据 ,高位数据后输出 */
        data |= b << i;          /* 填充 data 的每一位 */
    }
    return data;
}
```

在这里 ds18b20_read_bit 函数从 DS18B20 处读取 1 位数据，在前面已经对读时序也进行了详细的分析，所以这里也不展开解释了。

下面介绍读取温度函数，其定义如下：

```
/**
 * @brief       开始温度转换
 * @param       无
 * @retval      无
 */
static void ds18b20_start(void)
{
    ds18b20_reset();
    ds18b20_check();
    ds18b20_write_byte(0xcc);   /*  skip rom */
    ds18b20_write_byte(0x44);   /*  convert */
}
/**
 * @brief       从 ds18b20 得到温度值(精度：0.1C)
 * @param       无
 * @retval      温度值 （-550~1250）
 *   @note      返回的温度值放大了 10 倍.
 *              实际使用的时候,要除以 10 才是实际温度.
 */
short ds18b20_get_temperature(void)
{
    uint8_t flag = 1;           /* 默认温度为正数 */
    uint8_t TL, TH;
    short temp;

    ds18b20_start();            /* ds1820 start convert */
    ds18b20_reset();
    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)
    {    /* 温度为负，查看 DS18B20 的温度表示法与计算机存储正负数据的原理一致：
            正数补码为寄存器存储的数据自身，负数补码为寄存器存储值按位取反后+1
            所以我们直接取它实际的负数部分，但负数的补码为取反后加一，但考虑到低位可能+1 后
```

```
            有进位和代码冗余，
            我们这里先暂时没有作+1 的处理，这里需要留意  */
        TH = ~TH;
        TL = ~TL;
        flag = 0;     /* 温度为负 */
    }

    temp = TH;        /* 获得高八位 */
    temp <<= 8;
    temp += TL;       /* 获得底八位 */

    /* 转换成实际温度 */
    if (flag == 0)
    {   /* 将温度转换成负温度，这里的+1 参考前面的说明 */
        temp = (double)(temp+1) * 0.625;
        temp = -temp;
    }
    else
    {
        temp = (double)temp * 0.625;
     }
     return temp;
}
```

在这里简单介绍一下上面用到的 RAM 指令：

跳过 ROM(0xCC)，该指令只适合总线只有一个节点，它通过允许总线上的主机不提供 64 位 ROM 序列号而直接访问 RAM，节省了操作时间。

温度转换(0x44)，启动 DS18B20 进行温度转换，结果存入内部 RAM。

读暂存器(0xBE)，读暂存器 9 个字节内容，该指令从 RAM 的第一个字节（字节 0）开始读取，直到九个字节（字节 8，CRC 值）被读出为止。如果不需要读出所有字节的内容，那么主机可以在任何时候发出复位信号以中止读操作。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t = 0;
    short temperature;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DS18B20 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (ds18b20_init())              /* DS18B20 初始化 */
    {
        lcd_show_string(30, 110, 200, 16, 16, "DS18B20 Error", RED);
        delay_ms(200);
        lcd_fill(30, 110, 239, 130 + 16, WHITE);
        delay_ms(200);
    }

    lcd_show_string(30, 110, 200, 16, 16, "DS18B20 OK", RED);
    lcd_show_string(30, 130, 200, 16, 16, "Temp:   . C", BLUE);

    while (1)
```

```
    {
        if (t % 10 == 0)    /* 每 100ms 读取一次 */
        {
            temperature = ds18b20_get_temperature();
            if (temperature < 0)
            {
                lcd_show_char(30 + 40, 130, '-', 16, 0, BLUE);  /* 显示负号 */
                temperature = -temperature; /* 转为正数 */
            }
            else
            {
                lcd_show_char(30 + 40, 130, ' ', 16, 0, BLUE);  /* 去掉负号 */
            }
            /* 显示正数部分 */
            lcd_show_num(30 + 40 + 8, 130, temperature / 10, 2, 16, BLUE);
            /* 显示小数部分 */
            lcd_show_num(30 + 40 + 32, 130, temperature % 10, 1, 16, BLUE);
        }

        delay_ms(10);
        t++;

        if (t == 20)
        {
            t = 0;
            LED0_TOGGLE();  /* LED0 闪烁 */
        }
    }
}
```

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_temperature 函数获取温度值，然后显示在 LCD 上。

32.4 下载验证

假定 DS18B20 传感器已经接上去正确的位置，将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示当前的温度值的内容如图 32.4.1 所示：

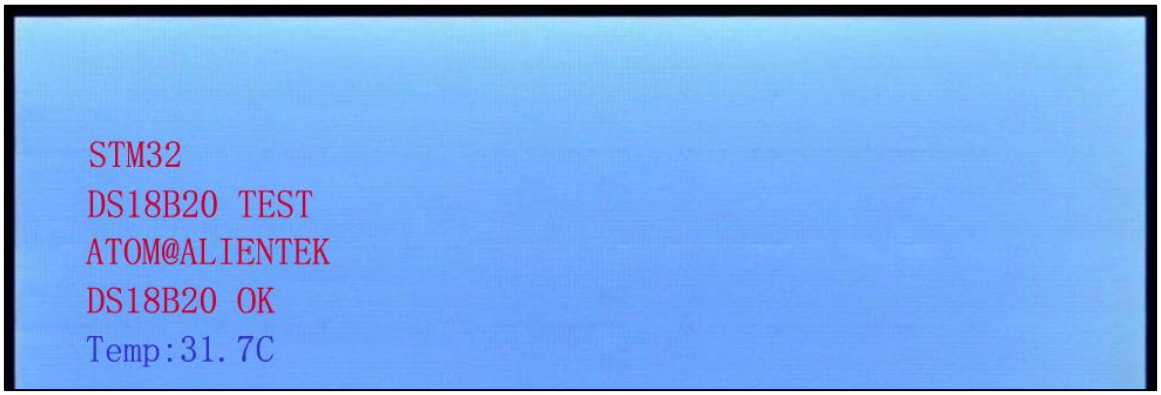


图 32.4.1 程序运行效果图

该程序还可以读取并显示负温度值，具备零下温度条件可以测试一下。

第三十三章 DHT11 数字温湿度传感器

本章，我们将介绍数字温湿度传感器 DHT11 的使用，与前一章的温度传感器相比，该传感器不但能测温度，还能测湿度。我们将学习如何获取 DHT11 传感器的温湿度数据，并把数据显示在 LCD 上。

本章分为如下几个小节：

33.1 DHT11 及其时序简介
33.2 硬件设计
33.3 程序设计
33.4 下载验证

33.1 DHT11 及时序简介

33.1.1 DHT11 简介

DHT11 是一款温湿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 5 ~ 95%RH，温度-20 ~ 60℃
- 精度：湿度±5%，温度±2℃
- 分辨率：湿度 1%，温度 0.1℃

DHT11 的管脚排列如图 33.1.1 所示：

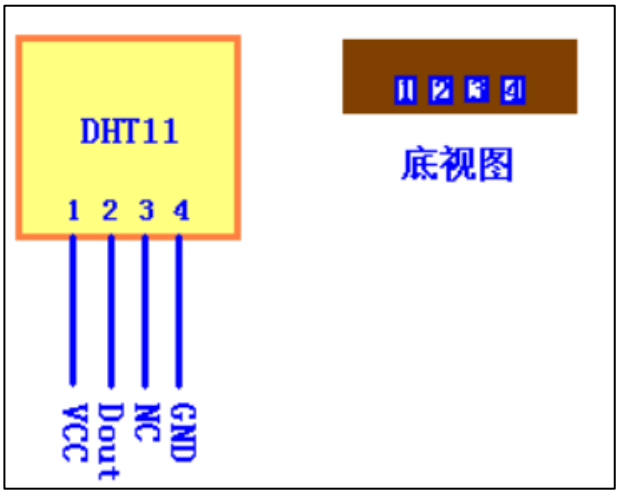


图 33.1.1.1 DHT11 管脚排列图

33.1.2 DHT11 时序简介

虽然 DHT11 与 DS18B20 类似，都是单总线访问，但是 DHT11 的访问，相对 DS18B20 来说简单很多。下面我们先来看看 DHT11 的数据结构。

DHT11 数字温湿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，高位先处。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 温度小数部分+8bit 校验和。其中校验和数据为前面四个字节相加。

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 33.1.2.1 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 33.1.2.1 某次读取到 DHT11 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 的数据格式十分简单的，DHT11 和 MCU 的一次通信最大为 34ms 左右，建议主机连续读取时间间隔不要小于 2s。

下面，我们介绍一下 DHT11 的传输时序。DHT11 的数据发送流程如图 33.1.2.2 所示：

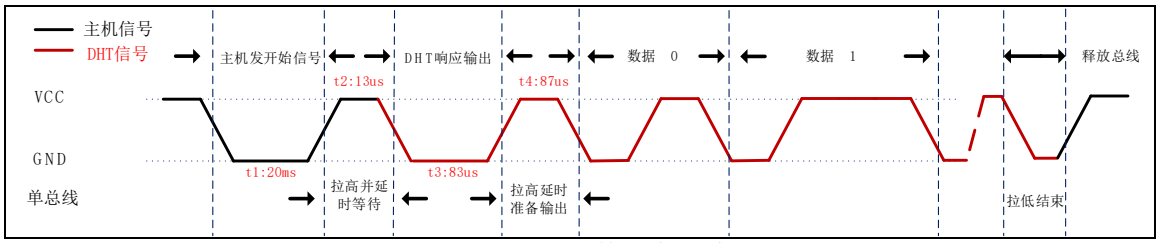


图 33.1.2.2 DHT11 数据发送流程图

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（10~35us）时间，然后读取 DHT11 的响应，正常的话，DHT11 会拉低数据线，保持 t3（78~88us）时间，作为响应信号，然后 DHT11 拉高数据线，保持 t4（80~92us）时间后，开始输出数据。

DHT11 输出数字‘0’时序如图 33.1.2.3 所示：

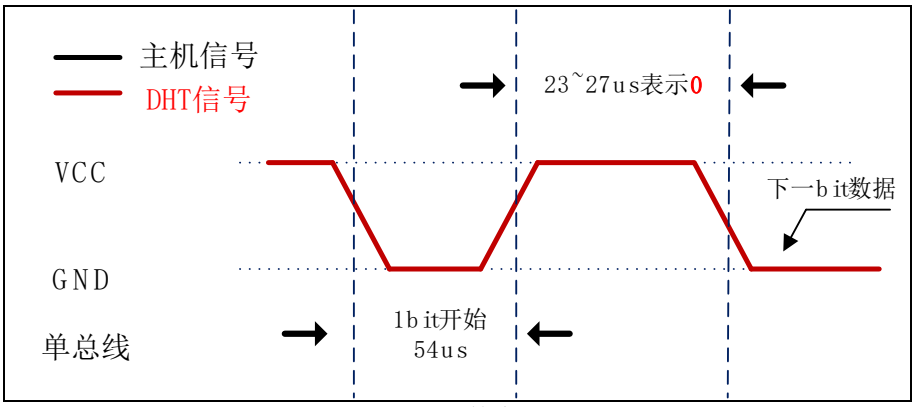


图 33.1.2.3 DHT11 数字‘0’时序图

DHT11 输出数字‘1’的时序如图 33.1.2.4 所示：

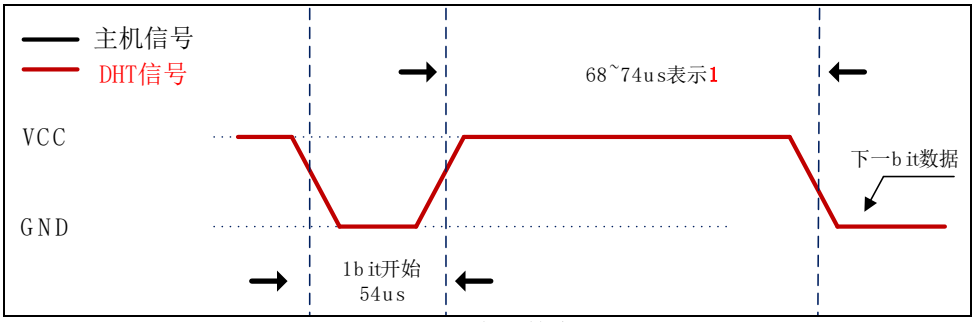


图 33.1.2.4 DHT11 输出数字‘1’时序图

DHT11 输出数字‘0’和‘1’时序，一开始都是 DHT11 拉低数据线 54us，后面拉高数据线保持的时间就不一样，数字‘0’就是 23~27us，而数字‘1’就是 68~74us。

通过以上了解，我们就可以通过 STM32 实现对 DHT11 的数据读取了。DHT11 的介绍就到这里，更详细的介绍，请参考 DHT11 的数据手册。

33.2 硬件设计

1. 例程功能

本实验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。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）DHT11 传感器(接在 PG15 上)

3. 原理图

DHT11 接口与 STM32 的连接关系跟上一章节中 DS18B20 和 STM32 的关系是一样的，使用到的 GPIO 口是 PG15。这里原理图就不列出来了，可以翻看上一章节原理图。

DHT11 和 DS18B20 的接口是共用一个的，不过 DHT11 有 4 个引脚，需要把 CN2 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 STM32F407 来读取温湿度值了。连接示意图如图 33.2.1 所示：

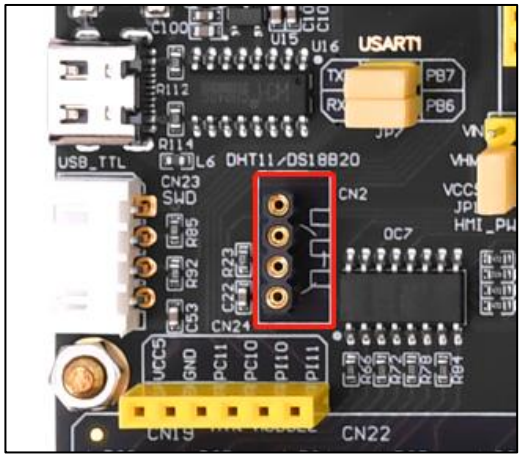


图 33.2.2 DHT11 连接示意图

这里要注意，将 DHT11 贴有字的一面朝左，而有很多孔的一面(网面)朝右，然后插入如图

所示的四个孔内就可以了。

33.3 程序设计

DHT11 实验中使用的是单总线协议，用到的是 HAL 中 GPIO 相关函数，前面也有介绍到，这里就不做展开了。下面介绍一下如何驱动 DHT11。

DHT11 配置步骤

1）使能 DHT11 数据线对应的 GPIO 时钟。

本实验中 DHT11 的数据线引脚是 PG15，因此需要先使能 GPIOG 的时钟，代码如下：

```
__HAL_RCC_GPIOG_CLK_ENABLE();       /* PG 口时钟使能 */
```

2）设置对应 GPIO 工作模式（开漏输出）

本实验 GPIO 使用开漏输出模式，通过函数 HAL_GPIO_Init 设置实现。

3）参考单总线协议，编写信号代码（复位脉冲、应答脉冲、读 0/1）

复位脉冲：拉低数据线，保持至少 18ms 时间，然后拉高数据线 10~35us 时间。

应答脉冲：DHT11 拉低数据线，保持 78~88us 时间。

读 0/1 信号：DHT11 拉低数据线延时 54us，然后拉高数据线延时一定时间，主机通过判断高电平时间得到 0 或者 1。

4）编写 DHT11 的读函数

基于读 1bit 数据的基础上，编写 DHT11 读 1 字节函数。

5）编写 DHT11 获取温度函数

参考 DHT11 典型温湿度读取过程，编写获取温湿度函数。

33.3.1 程序流程图

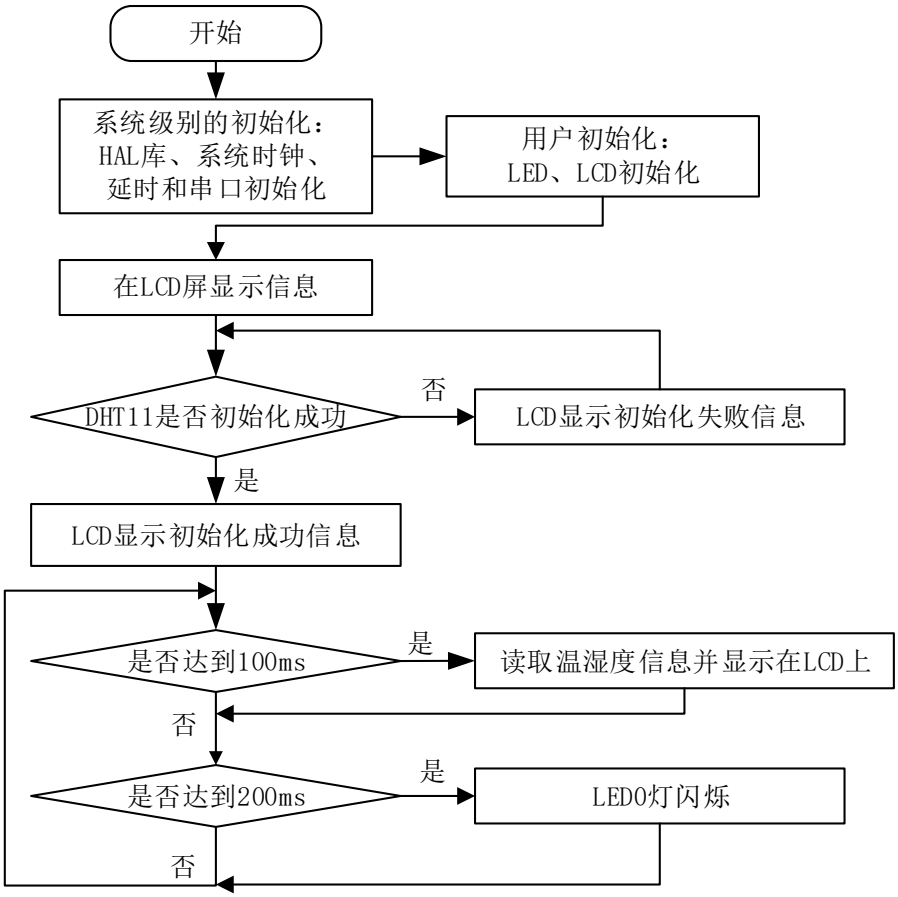


图 33.3.1.1 DHT11 实验程序流程图

33.3.2 程序解析

1.DHT11 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DHT11 驱动源码包括两个文件：dht11.c 和 dht11.h。

首先我们先看一下 dht11.h 头文件里面的内容，其定义如下：

```
/* DHT11 引脚 定义 */
#define DHT11_DQ_GPIO_PORT                  GPIOG
#define DHT11_DQ_GPIO_PIN                   GPIO_PIN_15
#define DHT11_DQ_GPIO_CLK_ENABLE()
                  do{ __HAL_RCC_GPIOG_CLK_ENABLE(); }while(0)   /* PG 口时钟使能 */
/* IO 操作函数 */
#define DHT11_DQ_OUT(x)   do{ x ? \
 HAL_GPIO_WritePin(DHT11_DQ_GPIO_PORT, DHT11_DQ_GPIO_PIN, GPIO_PIN_SET) : \
 HAL_GPIO_WritePin(DHT11_DQ_GPIO_PORT, DHT11_DQ_GPIO_PIN, GPIO_PIN_RESET); \
                          }while(0)          /* 数据端口输出 */
/* 数据端口输入 */
#define DHT11_DQ_IN    HAL_GPIO_ReadPin(DHT11_DQ_GPIO_PORT, DHT11_DQ_GPIO_PIN)
```

对 DHT11 的相关引脚以及 IO 操作进行宏定义，方便程序中调用。

下面我们直接介绍 dht11.c 的程序，首先先介绍一下 DHT11 传感器的初始化函数，其定义如下：

```
/**
 * @brief       初始化 DHT11 的 IO 口 DQ 同时检测 DHT11 的存在
 * @param       无
 * @retval      0, 正常
 *              1, 不存在/不正常
 */
uint8_t dht11_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    DHT11_DQ_GPIO_CLK_ENABLE();     /* 开启 DQ 引脚时钟 */

    gpio_init_struct.Pin = DHT11_DQ_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_OD;            /* 开漏输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;     /* 高速 */
    HAL_GPIO_Init(DHT11_DQ_GPIO_PORT, &gpio_init_struct);   /* 初始化 DQ 引脚 */
    /* DHT11_DQ 引脚模式设置,开漏输出,上拉, 这样就不用再设置 IO 方向了, 开漏输出的时候(=1),
       也可以读取外部信号的高低电平 */

    dht11_reset();
    return dht11_check();
}
```

在 DHT11 的初始化函数中，主要对用到的 GPIO 口进行初始化，同时在函数最后调用复位函数和自检函数，这两个函数在后面会解释到。

下面介绍的是复位 DHT11 函数和等待 DHT11 的回应函数，它们的定义如下：

```
/**
 * @brief       复位 DHT11
 * @param       data: 要写入的数据
 * @retval      无
 */
static void dht11_reset(void)
{
    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 10~35us */
```

```
}

/**
 * @brief       等待 DHT11 的回应
 * @param       无
 * @retval      0, DHT11 正常
 *              1, DHT11 异常/不存在
 */
uint8_t dht11_check(void)
{
    uint8_t retry = 0;
    uint8_t rval = 0;

    while (DHT11_DQ_IN && retry < 100)      /* DHT11 会拉低 83us */
    {
        retry++;
        delay_us(1);
    }
    if (retry >= 100)
    {
        rval = 1;
    }
    else
    {
        retry = 0;
        while (!DHT11_DQ_IN && retry < 100) /* DHT11 拉低后会再次拉高 87us */
        {
            retry++;
            delay_us(1);
        }
        if (retry >= 100) rval = 1;
    }

    return rval;
}
```

以上两个函数分别代表着前面所说的复位脉冲与应答信号，大家可以对比前面的时序图进行理解。那么在上一章 DS18B20 的实验中，也对复位脉冲以及应答信号进行了详细的解释，大家也可以对比理解。

DHT11 与 DS18B20 有所不同，DHT11 是不需要写函数，只需要读函数即可，下面介绍从 DHT11 读取一个位函数和从 DHT11 读取一个字节函数，它们的定义如下：

```
/**
 * @brief       从 DHT11 读取一个位
 * @param       无
 * @retval      读取到的位值：0 / 1
 */
uint8_t dht11_read_bit(void)
{
    uint8_t 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)        /* 根据引脚状态返回 bit */
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

/**
 * @brief       从 DHT11 读取一个字节
 * @param       无
 * @retval      读到的数据
 */
static uint8_t dht11_read_byte(void)
{
    uint8_t i, data = 0;

    for (i = 0; i < 8; i++)             /* 循环读取 8 位数据 */
    {
        data <<= 1;                     /* 高位数据先输出, 先左移一位 */
        data |= dht11_read_bit();       /* 读取 1bit 数据 */
    }

    return data;
}
```

在这里 dht11_read_bit 函数从 DHT11 处读取 1 位数据，大家可以对照前面的读时序图进行分析，读数字 0 和 1 的不同，在于高电平的持续时间，所以这个作为判断的依据。dht11_read_byte 函数就是调用一字节读取函数进行实现。

下面介绍读取温湿度函数，其定义如下：

```
/**
 * @brief       从 DHT11 读取一次数据
 * @param       temp: 温度值(范围:-20~60°)
 * @param       humi: 湿度值(范围:5%~95%)
 * @retval      0, 正常.
 *              1, 失败
 */
uint8_t dht11_read_data(uint8_t *temp, uint8_t *humi)
{
    uint8_t buf[5];
    uint8_t i;

    dht11_reset();

    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;
}
```

读取温湿度函数也是根据时序图进行实现的，在发送复位信号以及应答信号产生后，即可以读取 5Byte 数据进行处理，校验成功即读取数据有效成功。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t = 0;
    uint8_t temperature;
    uint8_t humidity;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DHT11 TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (dht11_init())                /* DHT11 初始化 */
    {
        lcd_show_string(30, 130, 200, 16, 16, "DHT11 Error", RED);
        delay_ms(200);
        lcd_fill(30, 130, 239, 130 + 16, WHITE);
        delay_ms(200);
    }

    lcd_show_string(30, 130, 200, 16, 16, "DHT11 OK", RED);
    lcd_show_string(30, 150, 200, 16, 16, "Temp:  C", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "Humi:  %", BLUE);

    while (1)
    {
        if (t % 10 == 0)    /* 每 100ms 读取一次 */
        {
            dht11_read_data(&temperature, &humidity);           /* 读取温湿度值 */
            lcd_show_num(30 + 40, 150, temperature, 2, 16, BLUE);/* 显示温度 */
            lcd_show_num(30 + 40, 170, humidity, 2, 16, BLUE);  /* 显示湿度 */
        }

        delay_ms(10);

        t++;

        if (t == 20)
        {
            t = 0;
            LED0_TOGGLE();  /* LED0 闪烁 */
        }
    }
}
```

主函数代码比较简单，一系列硬件初始化后，如果 DHT11 初始化成功，那么在循环中调用 dht11_get_temperature 函数获取温湿度值，每隔 100ms 读取数据并显示在 LCD 上。

33.4 下载验证

假定 DHT11 传感器已经接上去正确的位置，将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示当前的温度值的内容如图 33.4.1 所示：

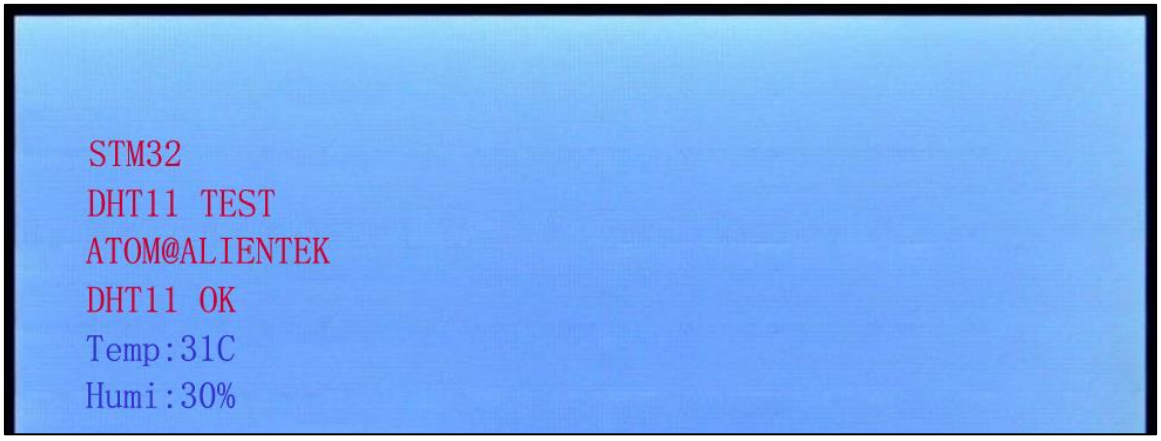


图 33.4.1 程序运行效果图

至此，本章实验结束。大家可以将本章通过 DHT11 读取到的温度值，和前一章的通过 DS18B20 读取到的温度值对比一下，看看哪个更准确？

第三十四章 FLASH 模拟 EEPROM 实验

STM32F407 本身没有自带 EEPROM，但是 STM32F407 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 STM32F407 内部的 FLASH 来实现第二十七章实验类似的效果，不过这次我们是将数据直接存放在 STM32F407 内部，而不是存放在 NOR FLASH。

本章分为如下几个小节：

34.1 STM32F407 FLASH 简介
34.2 硬件设计
34.3 程序设计
34.4 下载验证

34.1 STM32F407 FLASH 简介

不同型号的 STM32，其 FLASH 容量也有所不同，最小的只有 128K 字节，最大的则达到了 1024K 字节。电机开发板选择的是 STM32F407IGT6，FLASH 容量为 1024K 字节，属于大容量产品，STM32F40xx/41xx 闪存模块组织如图 34.1.1 所示：

块	名称	FLASH 起始地址	大小
主存储器	扇区 0	0x0800 0000 – 0x0800 03FF	16KB
	扇区 1	0x0800 4000 – 0x0800 7FFF	16KB
	扇区 2	0x0800 8000 – 0x0800 BFFF	16KB
	扇区 3	0x0800 C000 – 0x0800 FFFF	16KB
	扇区 4	0x0801 0000 – 0x0801 FFFF	64KB
	扇区 5	0x0802 0000 – 0x0803 FFFF	128KB
	扇区 6	0x0804 0000 – 0x0805 FFFF	128KB
	……	……	……
	扇区 11	0x080E 0000 – 0x080F FFFF	128KB
系统存储器		0x1FFF 0000 – 0x1FFF 77FF	30KB
OTP 区域		0x1FFF 7800 – 0x1FFF 7A0F	528 字节
选项字节		0x1FFF C000 – 0x1FFF C00F	16 字节

表 34.1 STM32F40xx/41xx 闪存模块组织表

STM32F407 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）。分为 12 个扇区，前 4 个扇区为 16KB 大小，扇区 4 为 64KB 大小，扇区 5~11 为 128KB 大小，不同容量的 STM32，拥有的扇区数不一样，比如我们的 STM32F407IGT6，拥有 12 个扇区。从表 34.1 可以看出，主存储器的起始地址为 0x08000000，B0、B1 都接 GND 的时候，就是从 0x08000000 开始运行代码。

系统存储器，主要用来存放 STM32F407 的 bootloader 代码，此代码在出厂的时候就固化在 STM32F407 里面了，专门用来给主存储器下载代码的。当 B0 接 V3.3，B1 接 GND 的时候，从该存储器启动（即进入串口下载模式）。

OTP 区域，即一次性可编程区域，总共 528 字节大小，被分成两个部分，前面 512 字节（32 字节为 1 块，分成 16 块），可以用来存储一些用户数据（一次性的，写完一次，永远不可以擦除!!），后面 16 字节，用于锁定对应块。

选项字节，用于配置读保护、BOR 级别、软件/硬件看门狗以及器件处于待机或停止模式下的复位。

闪存存储器接口寄存器，该部分用于控制闪存读写等，是整个闪存模块的控制结构。

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。

34.1.1 闪存的读取

STM32F407 可以通过内部的 I-Code 指令总线或 D-Code 数据总线访问内置闪存模块，本章主要讲解的数据读写，即通过 D-Code 数据总线来访问内部闪存模块。未来准确读取 Flash 数据，必须根据 CPU 时钟（HCLK）频率和器件电源电压在 Flash 存取控制寄存器（FLASH_ACR）中正确地设置等待周期数（LATENCY）。当电源电压低于 2.1V 时，必须关闭预取缓冲器。Flash 等待周期与 CPU 时钟频率之间的对应关系，如表 34.1.1.2 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 预取关闭
0 WS（1 个 CPU 周期）	0 < HCLK ≤ 30	0 < HCLK ≤ 24	0 < HCLK ≤ 22	0 < HCLK ≤ 20
1 WS（2 个 CPU 周期）	30 < HCLK ≤ 60	24 < HCLK ≤ 48	22 < HCLK ≤ 44	20 < HCLK ≤ 40
2 WS（3 个 CPU 周期）	60 < HCLK ≤ 90	48 < HCLK ≤ 72	44 < HCLK ≤ 66	40 < HCLK ≤ 60
3 WS（4 个 CPU 周期）	90 < HCLK ≤ 120	72 < HCLK ≤ 96	66 < HCLK ≤ 88	60 < HCLK ≤ 80
4 WS（5 个 CPU 周期）	120 < HCLK ≤ 150	96 < HCLK ≤ 120	88 < HCLK ≤ 110	80 < HCLK ≤ 100
5 WS（6 个 CPU 周期）	150 < HCLK ≤ 168	120 < HCLK ≤ 144	110 < HCLK ≤ 132	100 < HCLK ≤ 120
6 WS（7 个 CPU 周期）		144 < HCLK ≤ 168	132 < HCLK ≤ 154	120 < HCLK ≤ 140
7 WS（8 个 CPU 周期）			154 < HCLK ≤ 168	140 < HCLK ≤ 160

表 34.1.1.2 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[2:0]三个位设置。系统复位后，CPU 时钟频率为内部 16 M RC 振荡器（HSI），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 168 Mhz 频率作为 CPU 时钟之前，必须先设置 LATENCY 为 5，否则 FLASH 读写可能出错，导致死机。

正常工作时（168 Mhz），虽然 FLASH 需要 6 个 CPU 等待周期，但是由于 STM32F407 具有自适应实时存储器加速器(ART Accelerator)，通过指令缓存存储器，预取指令，实现相当于 0 FLASH 等待的运行速度。关于自适应实时存储器加速器的详细介绍，请大家参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 3.4.2 节。STM23F4 的 FLASH 读取是很简单的。例如，我们要从地址 addr，读取一个字（字节为 8 位，半字为 16 位，字为 32 位），可以通过如下的语句读取：

```
data=*(volatile uint32_t *)addr;
```

将 addr 强制转换为 volatile uint32_t 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 volatile uint32_t 改为 volatile uint16_t，即可读取指定地址的一个半字。相对 FLASH 读取来说，STM32F407 FLASH 的写就复杂一点了，下面我们介绍 STM32F407 闪存的编程和擦除。

34.1.2 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 STM32F407 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。

STM32F407 用户闪存的编程一般由 6 个 32 位寄存器控制，他们分别是：

- FLASH 访问控制寄存器(FLASH_ACR)
- FLASH 秘钥寄存器(FLASH_KEYR)
- FLASH 选项秘钥寄存器(FLASH_OPTKEYR)
- FLASH 状态寄存器(FLASH_SR)
- FLASH 控制寄存器(FLASH_CR)
- FLASH 选项控制寄存器(FLASH_OPTCR)

STM32F407 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 寄存器才可以解除写保护，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。

STM32F407 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 34.1.2.1：

	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 34.1.2.1 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。

FLASH 配置步骤

STM32F407 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），无法写入。STM32F407 的标准编程步骤如图 34.1.2.1 所示：

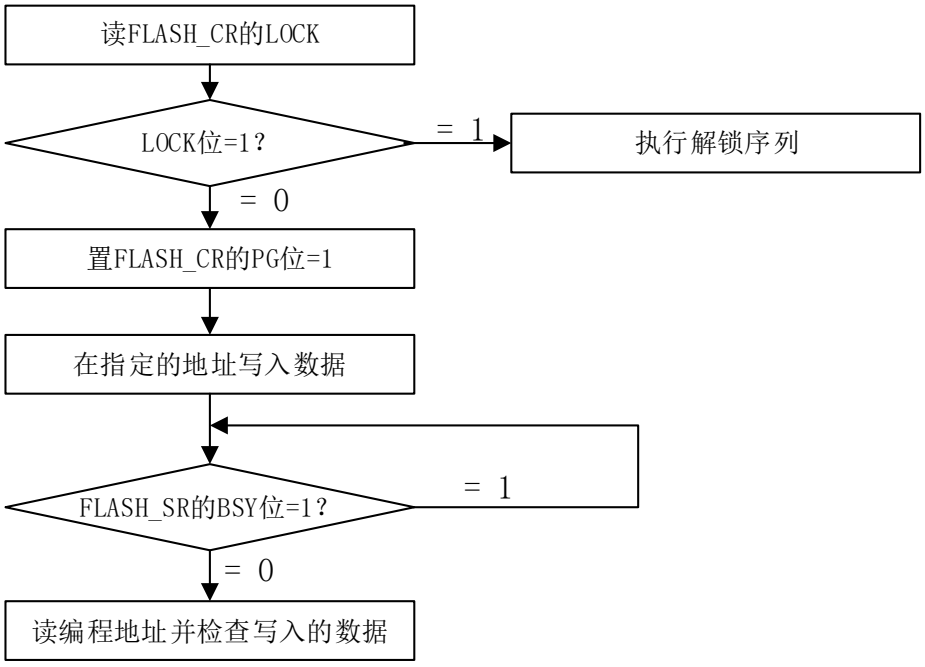


图 34.1.2.1 STM32 闪存编程过程

从上图可以得到闪存的编程顺序如下：

1）检查 FLASH_CR 的 LOCK 是否解锁，如果没有则先解锁

2）检查 FLASH_SR 寄存器的 BSY 位，以确认没有其他正在进行的编程操作

3）设置 FLASH_CR 寄存器的 PG 位为‘1’

4）在指定的地址写入数据（一次写入 32 字节，不能超过 32 字节）

5）等待 BSY 位变为‘0’

6）读出写入地址并验证数据

前面提到，我们在 STM32 的 FLASH 编程的时候，要先判断缩写地址是否被擦出了，所以，我们有必要再介绍一下 STM32 的闪存擦除，STM32 的闪存擦除分为两种：页擦除和整片擦除。页擦除过程如图 34.1.2.2 所示：

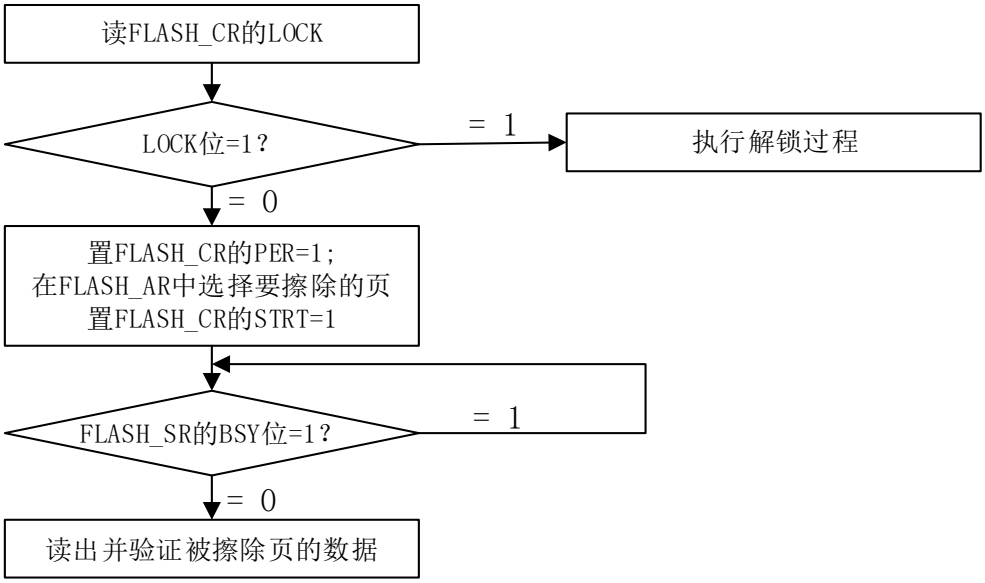


图 34.1.2.2 STM32 闪存页擦除过程

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，并设置 SNB=0（只有 1 个扇区，扇区 0）。

4，将 FLASH_CR 寄存器中的 START 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 STM32F407 的扇区擦除功能。整片擦除功能我们在这里就不介绍了，想了解的朋友请看《STM32F4xx 参考手册_V4（中文版）.pdf》第 3.5.3 节。

34.1.3 FLASH 寄存器

通过上面的讲解，我们基本对 STM32 闪存的读写执行步骤有所了解。接下来，我们介绍本实验需要用到的一些 FLASH 寄存器。

- **Flash 访问控制寄存器（FLASH_ACR）**

Flash 访问控制寄存器描述如图 34.1.3.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			DCRST	ICRST	DCEN	ICEN	PRFTEN	Reserved					LATENCY		
			rw	w	rw	rw	rw						rw	rw	rw

位 31:11 保留，必须保持清零。

位 12 **DCRST**：数据缓存复位 (Data cache reset)
0：数据缓存不复位
1：数据缓存复位
只有在关闭数据缓存时才能在该位中写入值。

位 11 **ICRST**：指令缓存复位 (Instruction cache reset)
0：指令缓存不复位
1：指令缓存复位
只有在关闭指令缓存时才能在该位中写入值。

位 10 **DCEN**：数据缓存使能 (Data cache enable)
0：关闭数据缓存
1：使能数据缓存

位 9 **ICEN**：指令缓存使能 (Instruction cache enable)
0：关闭指令缓存
1：使能指令缓存

位 8 **PRFTEN**：预取使能 (Prefetch enable)
0：关闭预取
1：使能预取

位 7:3 保留，必须保持清零。

位 2:0 **LATENCY**：延迟 (Latency)
这些位表示 CPU 时钟周期与 Flash 访问时间之比。
000：零等待周期
001：一个等待周期
010：两个等待周期
011：三个等待周期
100：四个等待周期
101：五个等待周期
110：六个等待周期
111：七个等待周期

图 34.1.3.1 FLASH_ACR 寄存器

我们重点介绍 LATENCY[2:0]这三个位，这三个位，必须根据我们 MCU 的工作电压和频率来进行正确的设置，否则，可能会死机。用于控制 FLASH 读延迟，必须根据我们 MCU 内核的工作电压和频率，来进行正确的设置，否则，可能死机，设置规则见表 34.1.1.2。其他 DCEN、ICEN 和 PRFTEN 这三个位也比较重要，为了达到最佳性能，这三个位我们一般都设置为 1 即可。

- **FLASH 密钥寄存器（FLASH_KEYR）**

FLASH 密钥寄存器描述如图 34.1.3.2 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 34.1.3.2 FLASH_KEYR 寄存器

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。

- **FLASH 控制寄存器（FLASH_CR）**

FLASH 控制寄存器描述如图 34.1.3.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LOCK	Reserved					ERRIE	EOPIE	Reserved							STRT
rs						rw	rw								rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						PSIZE[1:0]		Res.	SNB[3:0]				MER	SER	PG
						rw	rw		rw	rw	rw	rw	rw	rw	rw

图 34.1.3.3 FLASH_CR 寄存器

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。在该为写入 1，将执行一次擦除操作。

PSIZE[1:0]位，用于设置编程宽度，我们一般设置 PSIZE =2 即可（32 位）。

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 0~1。

SER 位，该位用于选择扇区擦除操作，在扇区擦除的时候，需要将该位置 1。

PG 位，该位用于选择编程操作，在往 FLASH 写数据的时候，该位需要置 1。

FLASH_CR 的其他位，我们就不在这里介绍了，请大家参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 3.8.5 节。

- **FLASH 状态寄存器（FLASH_SR）**

FLASH 状态寄存器描述如图 34.1.3.4 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 34.1.3.4 FLASH_SR 寄存器

该寄存器我们主要用了 BSY 位：表示 BANK 当前正在执行编程操作，当该位为 1 时，表示正在执行 FLASH 操作，当该位为 0 时，表示当前未执行 FLASH 操作。

关于 STM32F407 FLASH 的介绍我们就介绍到这里，更详细的介绍，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》第三章。

34.2 硬件设计

1. 例程功能

按键 KEY1 控制写入 FLASH 的操作，按键 KEY0 控制读出操作，并在 TFTLCD 模块上显示相关信息，还可以借助 USMART 进行读取或者写入操作。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
　　LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键
　　KEY0 - PE2
　　KEY1 - PE3

34.3 程序设计

34.3.1 FLASH 的 HAL 库驱动

FLASH 在 HAL 库中的驱动代码在 stm32f4xx_hal_flash.c 和 stm32f4xx_hal_flash_ex.c 文件（及其头文件）中。

1. HAL_FLASH_Unlock 函数

解锁闪存控制寄存器访问的函数，其声明如下：

```
HAL_StatusTypeDef HAL_FLASH_Unlock(void);
```

- **函数描述：**
  用于解锁闪存控制寄存器的访问，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）。
- **函数形参：**无
- **函数返回值：**HAL_StatusTypeDef 枚举类型的值。

2. HAL_FLASH_Lock 函数

锁定闪存控制寄存器访问的函数，其声明如下：

```
HAL_StatusTypeDef HAL_FLASH_Lock(void);
```

- **函数描述：**
  用于锁定闪存控制寄存器的访问。
- **函数形参：**无
- **函数返回值：**HAL_StatusTypeDef 枚举类型的值。

3. HAL_FLASH_Program 函数

闪存写操作函数，其声明如下：

```
HAL_StatusTypeDef HAL_FLASHEx_Program(uint32_t TypeProgram, uint32_t Address,
                                      uint64_t Data);
```

- **函数描述：**
  该函数用于 FLASH 的写入。
- **函数形参：**
  形参 1 是 TypeProgram 用来区分要写入的数据类型。
  形参 2 是 Address 用来设置要写入数据的 FLASH 地址
  形参 3 是 Data 是要写入的数据类型。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

4. HAL_FLASHEx_Erase 函数

闪存擦除函数，其声明如下：

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
                                    uint32_t *SectorError);
```

- **函数描述：**
  该函数用于大量擦除或擦除指定的闪存扇区。
- **函数形参：**
  形参 1 是 FLASH_EraseInitTypeDef 结构体类型指针变量。

```
typedef struct
{
  uint32_t TypeErase;       /* 擦除类型 */
  uint32_t Banks;           /* 擦除的 Bank 编号 */
  uint32_t PageAddress;     /* 擦除页面地址 */
  uint32_t NbPages;         /* 擦除的页面数 */
  uint32_t VoltageRange;    /* 擦除的器件电压范围*/
} FLASH_EraseInitTypeDef;
```

  形参 2 是 uint32_t 类型指针变量，存放错误码，0xFFFFFFFF 值表示扇区已被正确擦除，其它值表示擦除过程中的错误扇区。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

5. FLASH_WaitForLastOperation 函数

等待 FLASH 操作完成函数，其声明如下：

```
HAL_StatusTypeDef FLASH_WaitForLastOperation(uint32_t Timeout);
```

- **函数描述：**
  该函数用于等待 FLASH 操作完成。
- **函数形参：**
  形参 1 是 FLASH 操作超时时间。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

34.3.2 程序流程图

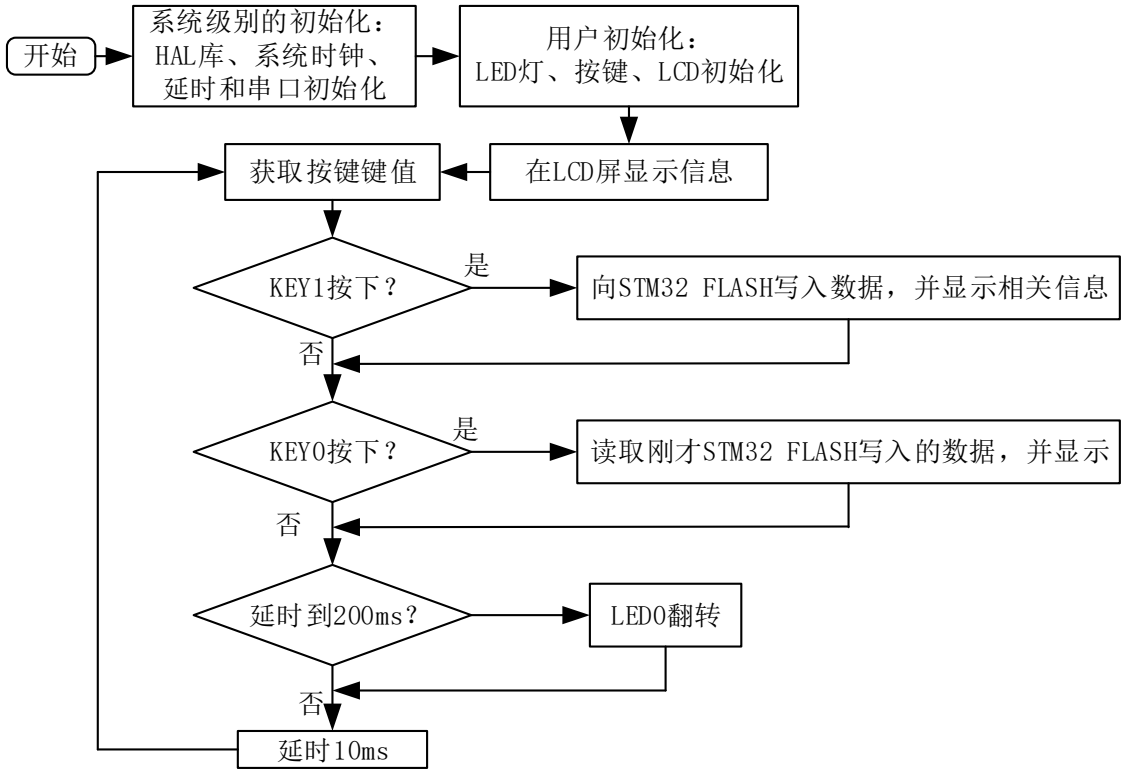


图 34.3.2.1 FLASH 模拟 EEPROM 实验程序流程图

34.3.3 程序解析

1. STMFLASH 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。STMFLASH 驱动源码包括两个文件：stmflash.c 和 stmflash.h。

stmflash.h 头文件做了一些比较重要的宏定义，定义如下：

```
/*********************************************************************************/
/* FLASH 起始地址 */
#define STM32_FLASH_BASE        0x08000000   /* STM32 FLASH 的起始地址 */
#define STM32_FLASH_SIZE        0x100000     /* STM32 FLASH 总大小 */
#define FLASH_WAITETIME         50000        /* FLASH 等待超时时间 */

/* FLASH 扇区的起始地址*/
/* 扇区 0 起始地址, 16 Kbytes */
#define ADDR_FLASH_SECTOR_0     ((uint32_t )0x08000000)
/* 扇区 1 起始地址, 16 Kbytes */
#define ADDR_FLASH_SECTOR_1     ((uint32_t )0x08004000)
```

```
/* 扇区 2 起始地址, 16 Kbytes */
#define ADDR_FLASH_SECTOR_2     ((uint32_t )0x08008000)
/* 扇区 3 起始地址, 16 Kbytes */
#define ADDR_FLASH_SECTOR_3     ((uint32_t )0x0800C000)
/* 扇区 4 起始地址, 64 Kbytes */
#define ADDR_FLASH_SECTOR_4     ((uint32_t )0x08010000)
/* 扇区 5 起始地址, 128 Kbytes */
#define ADDR_FLASH_SECTOR_5     ((uint32_t )0x08020000)
/* 扇区 6 起始地址, 128 Kbytes */
#define ADDR_FLASH_SECTOR_6     ((uint32_t )0x08040000)
/* 扇区 7 起始地址, 128 Kbytes */
#define ADDR_FLASH_SECTOR_7     ((uint32_t )0x08060000)
/* 扇区 8 起始地址, 128 Kbytes */
#define ADDR_FLASH_SECTOR_8     ((uint32_t )0x08080000)
/* 扇区 9 起始地址, 128 Kbytes */
#define ADDR_FLASH_SECTOR_9     ((uint32_t )0x080A0000)
/* 扇区 10 起始地址,128 Kbytes */
#define ADDR_FLASH_SECTOR_10    ((uint32_t )0x080C0000)
/* 扇区 11 起始地址,128 Kbytes */
#define ADDR_FLASH_SECTOR_11    ((uint32_t )0x080E0000)
```

STM32_FLASH_BASE 和 STM32_FLASH_SIZE 分别是 FLASH 的起始地址和 FLASH 总大小，这两个宏定义随着芯片是固定的，电机开发板的 STM32F407IGT6 的 FLASH 是 1024K 字节，所以 STM32_FLASH_SIZE 宏定义值为 0x100000。

下面我们开始介绍 stmflash.c 的程序，具体程序源码如下：

```
/**
 * @brief       得到 FLASH 的错误状态
 * @param       无
 * @retval      执行结果
 *   @arg       0    : 已完成
 *   @arg       其他 : 错误编号
 */
static uint8_t stmflash_get_error_status(void)
{
    uint32_t res = 0;
    res = FLASH->SR;

    if (res & (1 << 16)) return 1;  /* BSY=1, 繁忙 */
    if (res & (1 << 7))  return 2;  /* PGSERR=1,编程序列错误 */
    if (res & (1 << 6))  return 3;  /* PGPERR=1,编程并行位数错误 */
    if (res & (1 << 5))  return 4;  /* PGAERR=1,编程对齐错误 */
    if (res & (1 << 4))  return 5;  /* WRPERR=1,写保护错误 */

    return 0;   /* 没有任何状态/操作完成. */
}

/**
 * @brief       等待操作完成
 * @param       time : 要延时的长短
 * @retval      执行结果
 *   @arg       0    : 已完成
 *   @arg       0XFF: 超时
 *   @arg       其他 : 错误编号
 */
static uint8_t stmflash_wait_done(uint32_t time)
{
    uint8_t res;

    do
    {
        res = stmflash_get_error_status();
```

```
        if (res != 1)
        {
            break;      /* 非忙，无需等待了，直接退出 */
        }

        time--;
    } while (time);

    if (time == 0)res = 0XFF;   /* 超时 */

    return res;
}

/**
 * @brief       擦除扇区
 * @param       saddr   : 扇区地址 0 ~ 11
 *                        0~3, 16K 扇区; 4,64K 扇区; 5~11, 128K 扇区.
 * @retval      执行结果
 *   @arg       0   : 已完成
 *   @arg       0XFF: 超时
 *   @arg       其他 : 错误编号
 */
static uint8_t stmflash_erase_sector(uint32_t saddr)
{
    uint8_t res = 0;

    res = stmflash_wait_done(0XFFFFFFFF);   /* 等待上次操作结束 */

    if (res == 0)
    {
        FLASH->CR &= ~(3 << 8);             /* 清除 PSIZE 原来的设置 */
        FLASH->CR |= 2 << 8;                /* 设置为 32bit 宽,确保 VCC=2.7~3.6V 之间!! */
        FLASH->CR &= ~(0X1F << 3);          /* 清除原来的设置 */
        FLASH->CR |= saddr << 3;            /* 设置要擦除的扇区 */
        FLASH->CR |= 1 << 1;                /* 扇区擦除 */
        FLASH->CR |= 1 << 16;               /* 开始擦除 */
        res = stmflash_wait_done(0XFFFFFFFF);    /* 等待操作结束 */

        if (res != 1)                       /* 非忙 */
        {
            FLASH->CR &= ~(1 << 1);         /* 清除扇区擦除标志 */
        }
    }

    return res;
}

/**
 * @brief       在 FLASH 指定地址写一个字 (32 位数据)
 *   @note      这了写入一个字，是指 4 个字节
 * @param       faddr   : 写入地址 (此地址必须为 4 的倍数!!)
 * @param       data    : 要写入的数据(32 位)
 * @retval      执行结果
 *   @arg       0   : 已完成
 *   @arg       0XFF: 超时
 *   @arg       其他 : 错误编号
 */
static uint8_t stmflash_write_word(uint32_t faddr, uint32_t data)
{
    uint8_t res;

    res = stmflash_wait_done(0XFFFFF);
```

```
    if (res == 0)   /* OK */
    {
        FLASH->CR &= ~(3 << 8);                 /* 清除 PSIZE 原来的设置 */
        FLASH->CR |= 2 << 8;                /* 设置为 32bit 宽,确保 VCC=2.7~3.6V 之间!! */
        FLASH->CR |= 1 << 0;                    /* 编程使能 */
        *(volatile uint32_t *)faddr = data; /* 写入数据 */
        res = stmflash_wait_done(0XFFFFF);  /* 等待操作完成,一个字编程 */

        if (res != 1)                           /* 操作成功 */
        {
            FLASH->CR &= ~(1 << 0);             /* 清除 PG 位 */
        }
    }

    return res;
}

/**
 * @brief       从指定地址读取一个字 (32 位数据)
 * @param       faddr   : 读取地址 (此地址必须为 4 倍数!!)
 * @retval      读取到的数据 (32 位)
 */
uint32_t stmflash_read_word(uint32_t faddr)
{
    return *(volatile uint32_t *)faddr;
}

/**
 * @brief       获取某个地址所在的 flash 扇区
 * @param       addr : flash 地址
 * @retval      0~11,即 addr 所在的扇区
 */
uint8_t  stmflash_get_flash_sector(uint32_t  addr)
{
    if (addr < ADDR_FLASH_SECTOR_1) return FLASH_SECTOR_0;
    else if (addr < ADDR_FLASH_SECTOR_2) return FLASH_SECTOR_1;
    else if (addr < ADDR_FLASH_SECTOR_3) return FLASH_SECTOR_2;
    else if (addr < ADDR_FLASH_SECTOR_4) return FLASH_SECTOR_3;
    else if (addr < ADDR_FLASH_SECTOR_5) return FLASH_SECTOR_4;
    else if (addr < ADDR_FLASH_SECTOR_6) return FLASH_SECTOR_5;
    else if (addr < ADDR_FLASH_SECTOR_7) return FLASH_SECTOR_6;
    else if (addr < ADDR_FLASH_SECTOR_8) return FLASH_SECTOR_7;
    else if (addr < ADDR_FLASH_SECTOR_9) return FLASH_SECTOR_8;
    else if (addr < ADDR_FLASH_SECTOR_10) return FLASH_SECTOR_9;
    else if (addr < ADDR_FLASH_SECTOR_11) return FLASH_SECTOR_10;
    return FLASH_SECTOR_11;
}

/**
 * @brief       在 FLASH 指定位置, 写入指定长度的数据(自动擦除)
 *   @note      因为 STM32F4 的扇区实在太大,没办法本地保存扇区数据,所以本函数写地址如果非
 *              0XFF,那么会先擦除整个扇区且不保存扇区数据.所以写非 0XFF 的地址,将导致整个
 *              扇区数据丢失.
 *              建议写之前确保扇区里没有重要数据,最好是整个扇区先擦除了,然后慢慢往后写.
 *              该函数对 OTP 区域也有效!可以用来写 OTP 区!
 *              OTP 区域地址范围:0X1FFF7800~0X1FFF7A0F(注意: 最后 16 字节, 用于 OTP 数据
 *              块锁定, 别乱写! ! )
 * @param       waddr   : 起始地址 (此地址必须为 4 的倍数!!,否则写入出错!)
 * @param       pbuf    : 数据指针
 * @param       length  : 要写入的 字(32 位)数(就是要写入的 32 位数据的个数)
 * @retval      无
 */
```

```
void stmflash_write(uint32_t waddr, uint32_t *pbuf, uint32_t length)
{
    uint8_t status = 0;
    uint32_t addrx = 0;
    uint32_t endaddr = 0;

        /* 写入地址小于 STM32_FLASH_BASE, 或不是 4 的整数倍, 非法. */
    if (waddr < STM32_FLASH_BASE || waddr % 4 ||
        waddr > (STM32_FLASH_BASE + STM32_FLASH_SIZE))
       /* 写入地址大于 STM32_FLASH_BASE + STM32_FLASH_SIZE, 非法. */
    {
        return;
    }

    HAL_FLASH_Unlock();             /* 解锁 */
    FLASH->ACR &= ~(1 << 10);       /* FLASH 擦除期间,必须禁止数据缓存!!! */

    addrx = waddr;                  /* 写入的起始地址 */
    endaddr = waddr + length * 4;   /* 写入的结束地址 */

    if (addrx < 0X1FFF0000)         /* 只有主存储区,才需要执行擦除操作!! */
    {
        while (addrx < endaddr)     /* 扫清一切障碍.(对非 FFFFFFFF 的地方,先擦除) */
        {     /* 有非 0XFFFFFFFF 的地方,要擦除这个扇区 */
            if (stmflash_read_word(addrx) != 0XFFFFFFFF)
            {
                status = stmflash_erase_sector(stmflash_get_flash_sector(addrx));

                if (status)break;     /* 发生错误了 */
            }
            else
            {
                addrx += 4;
            }
        }
    }

    if (status == 0)
    {
        while (waddr < endaddr)                     /* 写数据 */
        {
            if (stmflash_write_word(waddr, *pbuf))  /* 写入数据 */
            {
                break;                              /* 写入异常 */
            }

            waddr += 4;
            pbuf++;
        }
    }

    FLASH->ACR |= 1 << 10;  /* FLASH 擦除结束,开启数据 fetch */

    HAL_FLASH_Lock();       /* 上锁 */
}

/**
 * @brief       从指定地址开始读出指定长度的数据
 * @param       raddr : 起始地址
 * @param       pbuf  : 数据指针
 * @param       length: 要读取的字(32)数,即 4 个字节的整数倍
 * @retval      无
 */
```

```
void stmflash_read(uint32_t raddr, uint32_t *pbuf, uint32_t length)
{
    uint32_t i;

    for (i = 0; i < length; i++)
    {
        pbuf[i] = stmflash_read_word(raddr);    /* 读取 4 个字节. */
        raddr += 4; /* 偏移 4 个字节. */
    }
}

/*****************************************************************************/
/* 测试用代码 */

/**
 * @brief       测试写数据(写 1 个字)
 * @param       waddr : 起始地址
 * @param       wdata : 要写入的数据
 * @retval      读取到的数据
 */
void test_write(uint32_t WriteAddr, uint32_t WriteData)
{
    stmflash_write(WriteAddr, &WriteData, 1);/* 写入一个字 */
}
```

该部分代码，我们重点介绍一下 stmflash_write 函数，该函数用于在 STM32F4 的指定地址写入指定长度的数据，类似第有几个要注意的点：

1，写入地址必须是用户代码区以外的地址

2，写入地址必须是 4 的倍数。

第 1 点比较好理解，如果把用户代码给擦了，可想而知你运行的程序可能就被废了，从而很可能出现死机的情况。不过，因为 STM32F4 的扇区都比较大（最少 16K，大的 128K），所以本函数不缓存要擦除的扇区内容，也就是如果要擦除，那么就是整个扇区擦除，所以建议大家使用该函数的时候，写入地址定位到用户代码占用扇区以外的扇区，比较保险。

第 2 点则是每次必须写入 32 位，即 4 字节，所以地址必须是 4 的倍数。

2. main.c 代码

在 main.c 里面编写如下代码：

```
/* 要写入到 STM32 FLASH 的字符串数组 */
const uint8_t g_text_buf[] = {"STM32 FLASH TEST"};

#define TEXT_LENTH sizeof(g_text_buf)          /* 数组长度 */

/*SIZE 表示字长(4 字节), 大小必须是 4 的整数倍, 如果不是的话, 强制对齐到 4 的整数倍 */
#define SIZE  TEXT_LENTH / 4 + ((TEXT_LENTH % 4) ? 1 : 0)

/* 设置 FLASH 保存地址(必须为 4 的整数倍，且大于本代码所占用 FLASH 的大小+0x08000000) */
#define FLASH_SAVE_ADDR     0x080100000

int main(void)
{
    uint8_t key = 0;
    uint16_t i = 0;
    uint8_t datatemp[SIZE];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    usmart_dev.init(84);                        /* 初始化 USMART */
    led_init();                                 /* 初始化 LED */
```

```
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "FLASH EEPROM TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY1:Write  KEY0:Read", RED);

    while (1)
    {
        key = key_scan(0);

        if (key == KEY1_PRES)   /* KEY1 按下,写入 STM32 FLASH */
        {
            lcd_fill(0, 150, 239, 319, WHITE);  /* 清除半屏 */
            lcd_show_string(30, 150, 200, 16, 16, "Start Write FLASH....", RED);
            stmflash_write(FLASH_SAVE_ADDR, (uint32_t *)g_text_buf, SIZE);
            /* 提示传送完成 */
            lcd_show_string(30, 150, 200, 16, 16, "FLASH Write Finished!", RED);
        }

        if (key == KEY0_PRES)   /* KEY0 按下,读取字符串并显示 */
        {
            lcd_show_string(30, 150, 200, 16, 16, "Start Read FLASH.... ", RED);
            stmflash_read(FLASH_SAVE_ADDR, (uint32_t *)datatemp, SIZE);
            /* 提示传送完成 */
            lcd_show_string(30, 150, 200, 16, 16, "The Data Readed Is:  ", RED);
            /* 显示读到的字符串 */
            lcd_show_string(30, 170, 200, 16, 16, (char*)datatemp, BLUE);
        }

        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。最后，我们将 stmflash_read_word 和 test_write 函数加入 USMART 控制，这样，我们就可以通过串口调试助手，调用 STM32F407 的 FLASH 读写函数，方便测试。

34.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 34.4.1 所示：

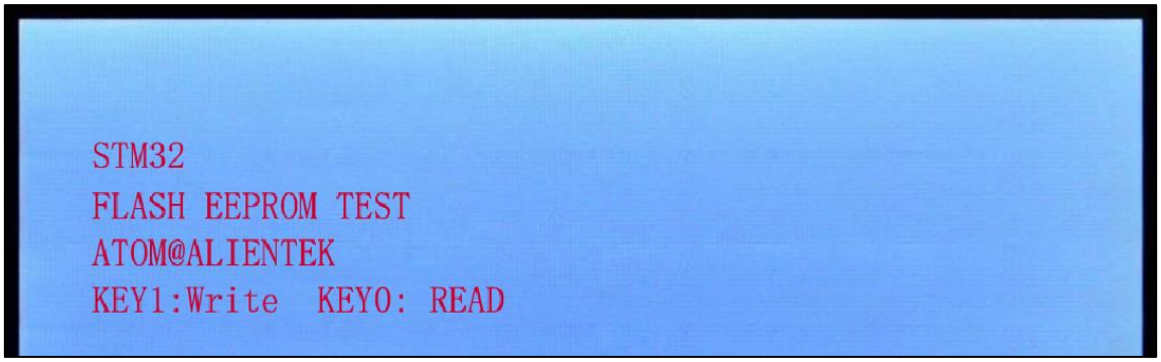


图 34.4.1 程序运行效果图

通过先按 KEY1 按键写入数据，然后按 KEY0 读取数据，得到如图 34.4.2 所示：

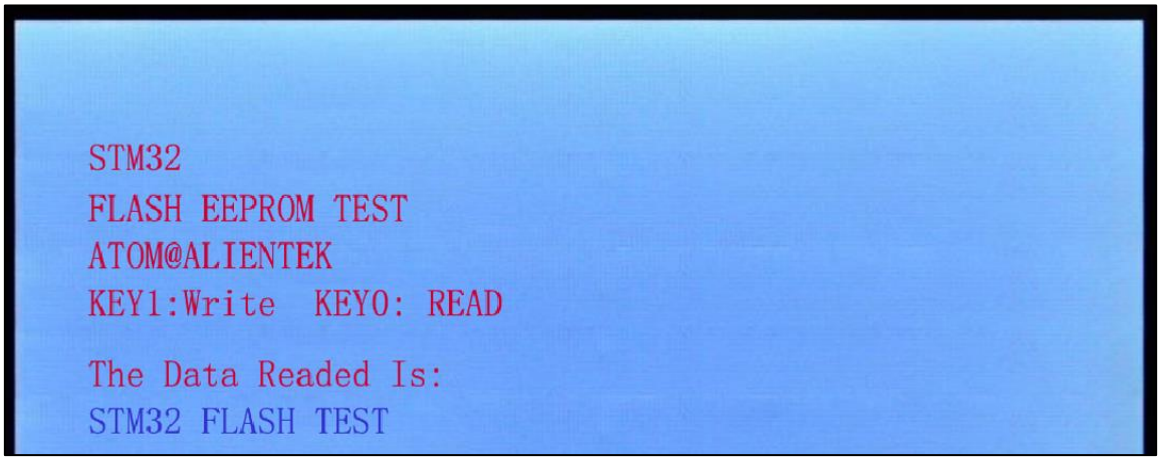


图 34.4.2 操作后的显示效果图

本实验的测试，我们还可以借助 USMART，调用：stmflash_read_word 和 test_write 函数进行测试！

第三篇 提高篇

现在我们已经学会如何使用 STM32 的一些基础外设了，STM32 我们算是入门了。接下来我们将学习提高篇的内容，本篇大部分内容都和 STM32 硬件底层关系不是很大，主要以各种第三方组件代码为主，比如：内存管理、文件系统、图片解码库、字库、手写识别库、T9 输入法、USB 库、网络协议栈、OS 等。这些内容偏向较复杂应用，掌握起来难度较大，需要大家认真学习，多多练习，做到举一反三。

本篇同样使用一章一实例的方式给大家介绍 STM32 的各种高级应用，通过本篇的学习，我们将可以完成更复杂的 STM32 项目开发。

第三十五章 内存管理实验

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。为了解决这些问题，我们来学习内存管理，实现对内存的动态管理。

本章分为如下几个小节：

35.1 内存管理简介
35.2 硬件设计
35.3 程序设计
35.4 下载验证

35.1 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，其实最终都是要实现两个函数：malloc 和 free。malloc 函数用来内存申请，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 35.1.1 所示：

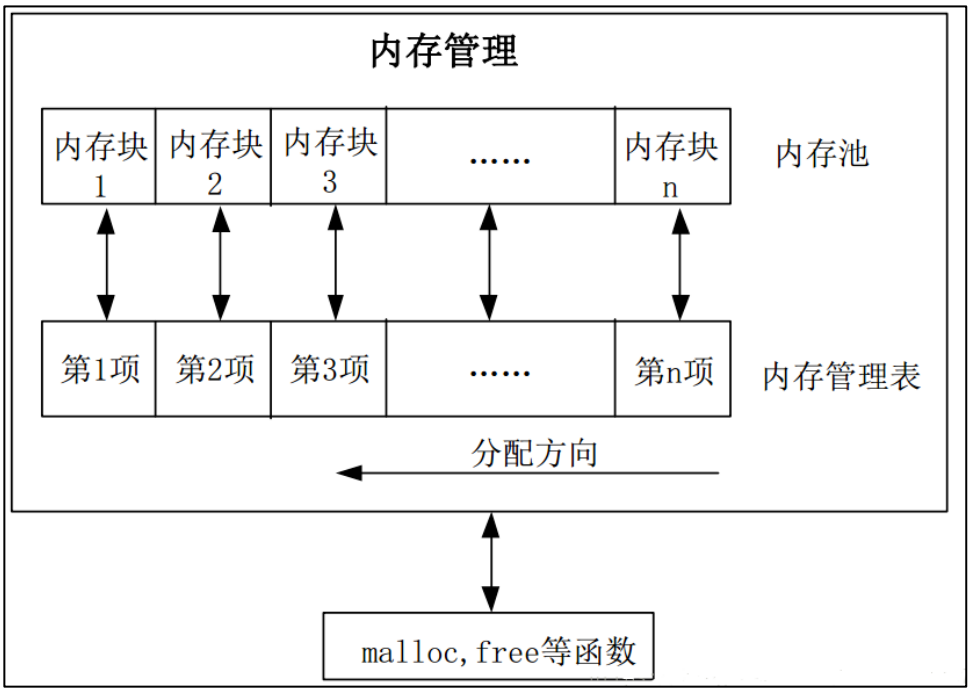


图 35.1.1 分块式内存管理原理

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。内存池被等分为了 n 块，对应的内存管理表，大小也为 n，内存管理表的每一个项对应内存池的一块内存。

内存管理表的项值代表的意义为：当该项值为 0 的时候，代表对应的内存块未被占用，当该项值非零的时候，代表该项对应的内存块已经被占用，其数值则代表被连续占用的内存块数。比如某项值为 10，那么说明包括本项对应的内存块在内，总共分配了 10 个内存块给外部的某个指针。

内存分配方向如上图所示，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 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（找到最后也没有找到连续 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。

35.2 硬件设计

1. 例程功能

开机后，显示提示信息，等待外部输入。KEY0 用于申请内存，每次申请 2K 字节内存，KEY1 用于释放内存，KEY2 用于切换操作内存区（SRAMIN/SRAMCCM，总共管理 2 个内存块）。还可以通过 USMART 调试，测试内存管理函数。

LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）串口 1 (PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

35.3 程序设计

35.3.1 程序流程图

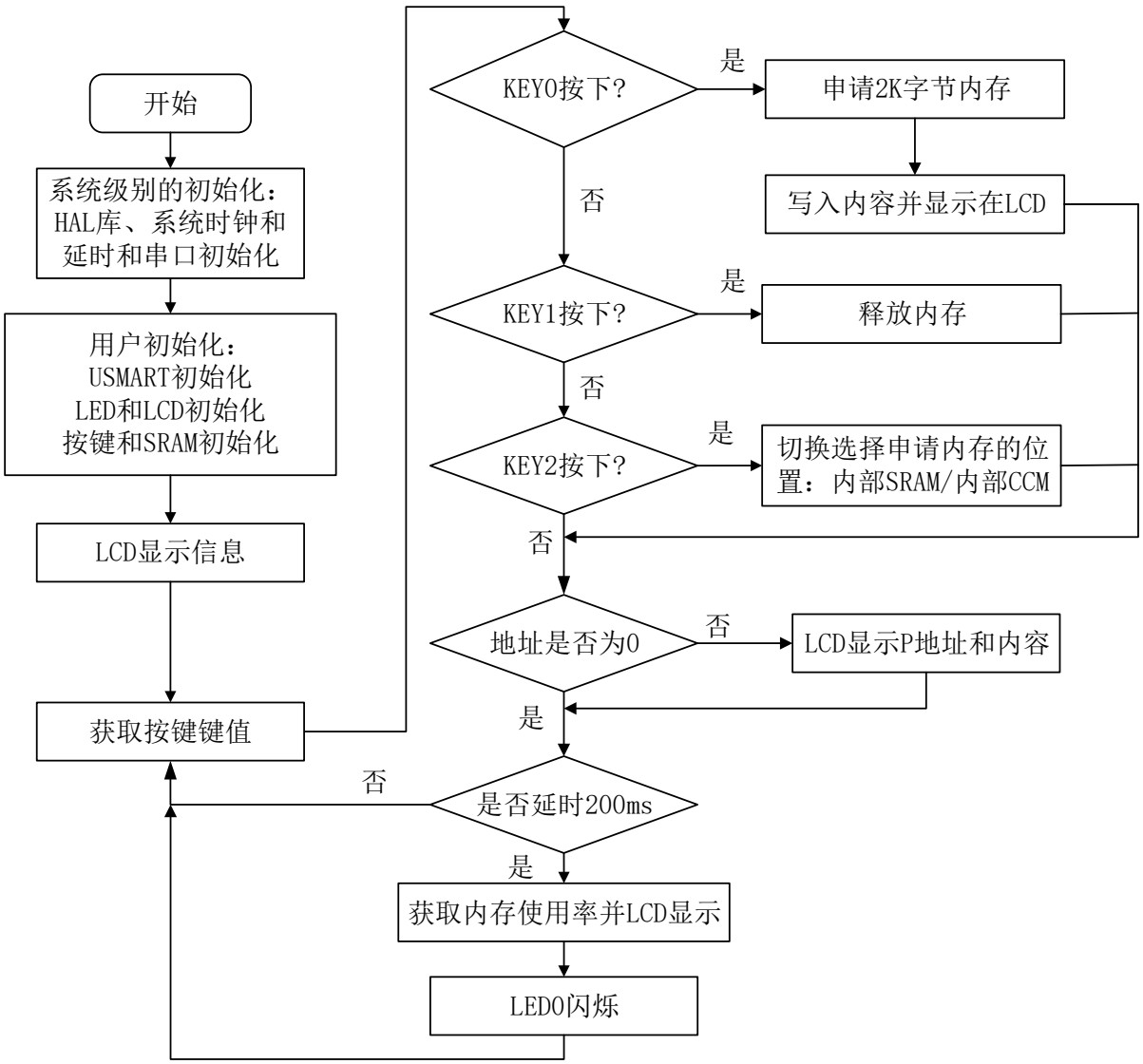


图 35.3.1.1 内存管理实验程序流程图

35.3.2 程序解析

1. MALLOC 代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。MALLOC 驱动源码包括两个文件：malloc.c 和 malloc.h。

下面我们介绍 malloc.h 中比较重要的一些结构体和内存参数宏定义，其定义如下：

```
/* 定义三个内存池 */
#define     SRAMIN                  0     /* 内部内存池 */
#define     SRAMCCM                 1     /* CCM 内存池(此部分 SRAM 仅仅 CPU 可以访问!!!) */
#define     SRAMBANK                2     /* 定义支持的 SRAM 块数 */

/* 定义内存管理表类型,当外扩 SDRAM 的时候，必须使用 uint32_t 类型,否则定义成 uint16_t,以节省
内存占用 */
```

```
#define MT_TYPE     uint16_t

/* mem1 内存参数设定.mem1 完全处于内部 SRAM 里面 */
#define     MEM1_BLOCK_SIZE         32                  /* 内存块大小为 32 字节 */
#define     MEM1_MAX_SIZE           100 * 1024          /* 最大管理内存 100K */
#define     MEM1_ALLOC_TABLE_SIZE   MEM1_MAX_SIZE/MEM1_BLOCK_SIZE/* 内存表大小 */

/* mem2 内存参数设定.mem2 处于 CCM,用于管理 CCM(特别注意,这部分 SRAM,仅 CPU 可以访问!!) */
#define     MEM2_BLOCK_SIZE         32                  /* 内存块大小为 32 字节 */
#define     MEM2_MAX_SIZE           60 * 1024           /* 最大管理内存 60K */
#define     MEM2_ALLOC_TABLE_SIZE   MEM2_MAX_SIZE/MEM2_BLOCK_SIZE /* 内存表大小 */

/* 内存管理控制器 */
struct _m_mallco_dev
{
    void (*init)(uint8_t);              /* 初始化 */
    uint16_t (*perused)(uint8_t);       /* 内存使用率 */
    uint8_t *membase[SRAMBANK];         /* 内存池 管理 SRAMBANK 个区域的内存 */
    MT_TYPE *memmap[SRAMBANK];          /* 内存管理状态表 */
    uint8_t  memrdy[SRAMBANK];          /* 内存管理是否就绪 */
};
```

我们可以定义几个不同的内存管理表，再分配相应的指针给到管理控制器即可。程序中我们用宏定义 MEM1_BLOCK_SIZE 来定义 malloc 可以管理的内部内存池总大小，实际上我们定义为一个大小为 MEM1_BLOCK_SIZE 的数组，这样编译后就能获得一块实际的连续内存区域，这里是 100K，MEM1_ALLOC_TABLE_SIZE 代表内存池的内存管理表大小。我们可以定义多个内存管理表，这样就可以同时管理多块内存。

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 uint16_t 类型）。显然是不合适的，我们这里取 32 字节，比例为 1:32，内存管理表相对就比较小了。

通过这个内存管理控制器_m_malloc_dev 结构体，我们把分块式内存管理的相关信息，其初始化函数、获取使用率、内存池、内存管理表以及内存管理的状态保存下来，实现对内存池的管理控制。

下面介绍 malloc.c 文件，内存池、内存管理表、内存管理参数和内存管理控制器的定义如下：

```
#if !(__ARMCC_VERSION >= 6010050)       /* 不是 AC6 编译器，即使用 AC5 编译器时 */
/* 内存池(64 字节对齐) */
static __align(64) uint8_t mem1base[MEM1_MAX_SIZE];     /* 内部 SRAM 内存池 */
static __align(64) uint8_t mem2base[MEM2_MAX_SIZE]
__attribute__((at(0x10000000)));        /* 内部 CCM 内存池 */
static __align(64) uint8_t mem3base[MEM3_MAX_SIZE]
__attribute__((at(0x68000000)));        /* 外部 SRAM 内存池 */

/* 内存管理表 */
static MT_TYPE mem1mapbase[MEM1_ALLOC_TABLE_SIZE];      /* 内部 SRAM 内存池 MAP */
static MT_TYPE mem2mapbase[MEM2_ALLOC_TABLE_SIZE] __attribute__((at(0x10000000
+ MEM2_MAX_SIZE)));  /* 内部 CCM 内存池 MAP */
#else       /* 使用 AC6 编译器时 */
/* 内存池(64 字节对齐) */
static __ALIGNED(64) uint8_t mem1base[MEM1_MAX_SIZE]; /* 内部 SRAM 内存池 */
static __ALIGNED(64) uint8_t mem2base[MEM2_MAX_SIZE]
__attribute__((section(".bss.ARM.__at_0x10000000"))); /* 内部 CCM 内存池 */
static __ALIGNED(64) uint8_t mem3base[MEM3_MAX_SIZE]
__attribute__((section(".bss.ARM.__at_0x68000000"))); /* 外部 SRAM 内存池 */

/* 内存管理表 */
static MT_TYPE mem1mapbase[MEM1_ALLOC_TABLE_SIZE];      /* 内部 SRAM 内存池 MAP */
```

```
static MT_TYPE mem2mapbase[MEM2_ALLOC_TABLE_SIZE]
__attribute__((section(".bss.ARM.__at_0x1000F000"))); /* 内部 CCM 内存池 MAP */
static MT_TYPE mem3mapbase[MEM3_ALLOC_TABLE_SIZE]
__attribute__((section(".bss.ARM.__at_0x680F0C00"))); /* 外部 SRAM 内存池 MAP */
#endif

/* 内存管理参数 */
const uint32_t memtblsize[SRAMBANK] = {MEM1_ALLOC_TABLE_SIZE,
MEM2_ALLOC_TABLE_SIZE, MEM3_ALLOC_TABLE_SIZE};          /* 内存表大小 */
const uint32_t memblksize[SRAMBANK] = {MEM1_BLOCK_SIZE, MEM2_BLOCK_SIZE,
MEM3_BLOCK_SIZE};                                       /* 内存分块大小 */
const uint32_t 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 */
};
```

MDK 支持用**__attirbute__((at(地址)))**的方法把变量定义到指定的区域，而且这个变量支持是算式，大家可以去 MKD 的帮助文件中查找__attribute__这个关键字查找相关信息，有比较详细的介绍。

我们通过判断编译器的版本，来执行不同方式的定义，对于 AC5 来说，使用的是**__attirbute__((at(地址)))**，但是如果你想换成 AC6 编译器，指定变量位置的函数变成**__attribute__((section(".bss.ARM.__at_地址")))**的方式，其中的.bss 表示初始化值为 0，而且这个方式不支持算式，所以还用上面的方法直接用宏计算出 SRAM 的地址的方法不可行了，所以我们需要直接手动算出 SRAM 对应的内存地址，同样地__align(64)在 AC6 下的写法也变成了__ALIGNED(64)，还有其它差异的部分，大家参考 MDK 官方提供的 AC5 到 AC6 的迁移方法的文档，我们主要介绍 AC5。

我们通过内存管理控制器 mallco_dev 结构体，实现对两个内存池的管理控制。

第一个是内部 SRAM 内存池，定义为：

```
static __align(64) uint8_t  mem1base[MEM1_MAX_SIZE];        /* 内部 SRAM 内存池 */
```

第二个是内部 CCM 内存池，定义为：

```
static __align(64) uint8_t mem2base[MEM2_MAX_SIZE]
                           __attribute__((at(0x10000000))); /* 内部 CCM 内存池 */
```

这里之所以要定义成 2 个，是因为这两个内存区域的地址不一样，STM32F4 内部内存分为两大块：1，普通内存（又分为主要内存和辅助内存，地址从：0x2000 0000 开始，共 128KB），这部分内存任何外设都可以访问。2，CCM 内存（地址从：0x1000 0000 开始，共 64KB），这部分内存仅 CPU 可以访问，DMA 之类的不可以直接访问，使用时得特别注意!!

下面介绍其他的 malloc 代码，具体如下：

```
/**
 * @brief       复制内存
 * @param       *des : 目的地址
 * @param       *src : 源地址
 * @param       n    : 需要复制的内存长度(字节为单位)
 * @retval      无
 */
void my_mem_copy(void *des, void *src, uint32_t n)
{
    uint8_t *xdes = des;
    uint8_t *xsrc = src;
    while (n--)*xdes++ = *xsrc++;
}
```

```
/**
 * @brief       设置内存值
 * @param       *s     : 内存首地址
 * @param       c      : 要设置的值
 * @param       count  : 需要设置的内存大小(字节为单位)
 * @retval      无
 */
void my_mem_set(void *s, uint8_t c, uint32_t count)
{
    uint8_t *xs = s;
    while (count--)*xs++ = c;
}

/**
 * @brief       内存管理初始化
 * @param       memx : 所属内存块
 * @retval      无
 */
void my_mem_init(uint8_t memx)
{
    /* 内存状态表数据清零 */
    mymemset(mallco_dev.memmap[memx], 0, memtblsize[memx] * 2);
    mymemset(mallco_dev.membase[memx], 0, memsize[memx]);  /* 内存池所有数据清零 */
    mallco_dev.memrdy[memx] = 1;    /* 内存管理初始化 OK */
}

/**
 * @brief       获取内存使用率
 * @param       memx : 所属内存块
 * @retval      使用率(扩大了 10 倍,0~1000,代表 0.0%~100.0%)
 */
uint16_t my_mem_perused(uint8_t memx)
{
    uint32_t used = 0;
    uint32_t i;

    for (i = 0; i < memtblsize[memx]; i++)
    {
        if (mallco_dev.memmap[memx][i]) used++;
    }

    return (used * 1000) / (memtblsize[memx]);
}

/**
 * @brief       内存分配(内部调用)
 * @param       memx : 所属内存块
 * @param       size : 要分配的内存大小(字节)
 * @retval      内存偏移地址
 *   @arg       0 ~ 0xFFFFFFFE : 有效的内存偏移地址
 *   @arg       0xFFFFFFFF     : 无效的内存偏移地址
 */
static uint32_t my_mem_malloc(uint8_t memx, uint32_t size)
{
    signed long offset = 0;
    uint32_t nmemb;      /* 需要的内存块数 */
    uint32_t cmemb = 0; /* 连续空内存块数 */
    uint32_t 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++;
    /* 搜索整个内存控制区 */
    for (offset = memtblsize[memx] - 1; offset >= 0; offset--)
    {
        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 */
}

/**
 * @brief       释放内存(内部调用)
 * @param       memx   : 所属内存块
 * @param       offset : 内存地址偏移
 * @retval      释放结果
 *   @arg       0, 释放成功;
 *   @arg       1, 释放失败;
 *   @arg       2, 超区域了(失败);
 */
static uint8_t my_mem_free(uint8_t memx, uint32_t 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. */
    }
```

```
}

/**
 * @brief       释放内存(外部调用)
 * @param       memx : 所属内存块
 * @param       ptr  : 内存首地址
 * @retval      无
 */
void myfree(uint8_t memx, void *ptr)
{
    uint32_t offset;

    if (ptr == NULL)return;     /* 地址为0. */

    offset = (uint32_t)ptr - (uint32_t)mallco_dev.membase[memx];
    my_mem_free(memx, offset);  /* 释放内存 */
}

/**
 * @brief       分配内存(外部调用)
 * @param       memx : 所属内存块
 * @param       size : 要分配的内存大小(字节)
 * @retval      分配到的内存首地址.
 */
void *mymalloc(uint8_t memx, uint32_t size)
{
    uint32_t offset;
    offset = my_mem_malloc(memx, size);

    if (offset == 0xFFFFFFFF)   /* 申请出错 */
    {
        return NULL;            /* 返回空(0) */
    }
    else    /* 申请没问题, 返回首地址 */
    {
        return (void *)((uint32_t)mallco_dev.membase[memx] + offset);
    }
}

/**
 * @brief       重新分配内存(外部调用)
 * @param       memx : 所属内存块
 * @param       *ptr : 旧内存首地址
 * @param       size : 要分配的内存大小(字节)
 * @retval      新分配到的内存首地址.
 */
void *myrealloc(uint8_t memx, void *ptr, uint32_t size)
{
    uint32_t offset;
    offset = my_mem_malloc(memx, size);

    if (offset == 0xFFFFFFFF)   /* 申请出错 */
    {
        return NULL;            /* 返回空(0) */
    }
    else    /* 申请没问题, 返回首地址 */
    {
        my_mem_copy((void *)((uint32_t)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 *)((uint32_t)mallco_dev.membase[memx]
                        + offset);      /* 返回新内存首地址 */
    }
```

```
}
```

整个 malloc 代码的核心函数：my_mem_malloc 和 my_mem_free，分别用于内存申请和内存释放。思路就是前面 35.1 所介绍的分配内存和释放内存，不过在这里，这两个函数知识内部调用，外部调用我们另外定义了 mymalloc 和 myfree 两个函数，其他函数我们就不多介绍了。

2. main.c 代码

main.c 代码如下：

```
const char *SRAM_NAME_BUF[SRAMBANK] = {" SRAMIN ", " SRAMCCM "};

int main(void)
{
    uint8_t paddr[20];  /* 存放 P Addr:+p 地址的 ASCII 值 */
    uint16_t memused = 0;
    uint8_t key;
    uint8_t i = 0;
    uint8_t *p = 0;
    uint8_t *tp = 0;
    uint8_t sramx = 0;  /* 默认为内部 sram */

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    usmart_dev.init(84);                    /* 初始化 USMART */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */

    my_mem_init(SRAMIN);                    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM);                   /* 初始化内部 SRAMCCM 内存池 */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "MALLOC TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Malloc & WR & Show", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY2:SRAMx KEY1:Free", RED);
    lcd_show_string(60, 160, 200, 16, 16, " SRAMIN ", BLUE);
    lcd_show_string(30, 176, 200, 16, 16, "SRAMIN   USED:", BLUE);
    lcd_show_string(30, 192, 200, 16, 16, "SRAMCCM   USED:", BLUE);
    while (1)
    {
        key = key_scan(0);   /* 不支持连按 */

        switch (key)
        {
            case KEY0_PRES:   /* KEY0 按下 */
              /* 申请 2K 字节,并写入内容,显示在 lcd 屏幕上面 */
               p = mymalloc(sramx, 2048);

               if (p != NULL)
               {
                   /* 向 p 写入一些内容 */
                   sprintf((char *)p, "Memory Malloc Test%03d", i);
                   /* 显示 P 的内容 */
                   lcd_show_string(30, 260, 209, 16, 16, (char *)p, BLUE);
               }

               break;

            case KEY1_PRES:       /* KEY1 按下 */
               myfree(sramx, p); /* 释放内存 */
```

```
            p = 0;              /* 指向空地址 */
            break;

        case KEY2_PRES:         /* KEY2 按下 */
            sramx++;

            if (sramx >= SRAMBANK) sramx = 0;

            lcd_show_string(60, 160, 200, 16, 16,(char *)SRAM_NAME_BUF[sramx],
                            BLUE);
            break;
    }

    if (tp != p)
    {
        tp = p;
        sprintf((char *)paddr, "P Addr:0X%08X", (uint32_t)tp);
         /* 显示 p 的地址 */
        lcd_show_string(30, 240, 209, 16, 16, (char *)paddr, BLUE);

        if (p)
        {
            /* 显示 p 的内容 */
            lcd_show_string(30, 260, 280, 16, 16, (char *)p, BLUE);
        }
        else
        {
            lcd_fill(30, 260, 209, 296, 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);
        /* 显示内部内存使用率 */
        lcd_show_string(30 + 112, 176, 200, 16, 16, (char *)paddr, BLUE);

        memused = my_mem_perused(SRAMCCM);
        sprintf((char *)paddr, "%d.%01d%%", memused / 10, memused % 10);
        /* 显示 TCM 内存使用率 */
        lcd_show_string(30 + 112, 192, 200, 16, 16, (char *)paddr, BLUE);

        LED0_TOGGLE();    /* LED0 闪烁 */
    }
  }
}
```

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。

35.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 内容如下图所示：

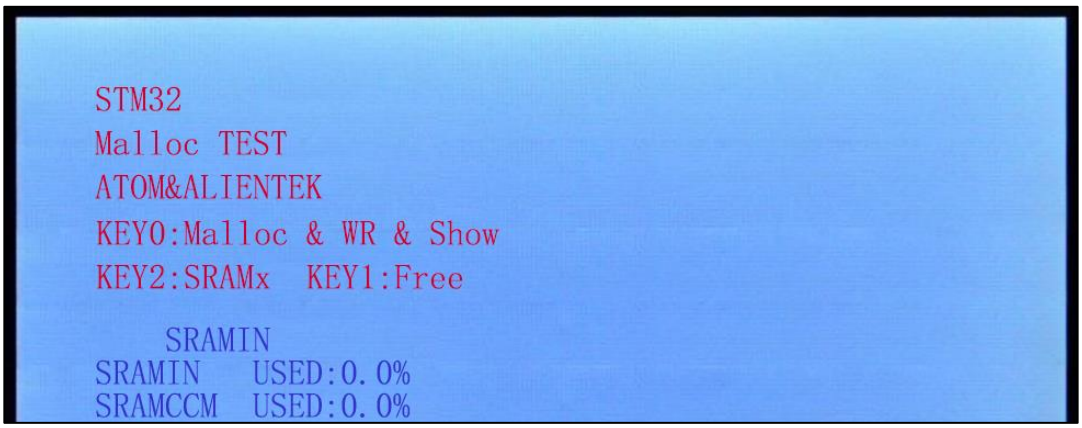


图 35.4.1 内存管理实验测试图

可以看到，内存的使用率均为 0%，说明还没有任何内存被使用。我们可以通过 KEY2 选择申请内存的位置：SRAIN 为内部，SRAMCCM 为内部，SRAMEX 为外部。此时我们选择从内部申请内存，按下 KEY0，就可以看到申请了 2%的一个内存块，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。效果如图 35.4.2 所示。

KEY0 键用来更新 p 的内容，更新后的内容将重新显示在 LCD 模块上。多按几次 KEY0，可以看到内存使用率持续上升（注意比对 p 的值，可以发现是递减的，说明是从顶部开始分配内存！）。每次申请一个内存块后，可以通过按下 KEY0 释放本次申请的内存，如果我们每次申请完内存不再使用却不及时释放掉，再按 KEY1 将无法释放之前的内存了，当这样的情况重复了多次，就会造成“内存泄漏”。我们程序就是模拟这样一个情况，请大家在实际使用的时候去注意到这种做法的危险性，必须在编程时严格避免内存泄漏的情况发生。

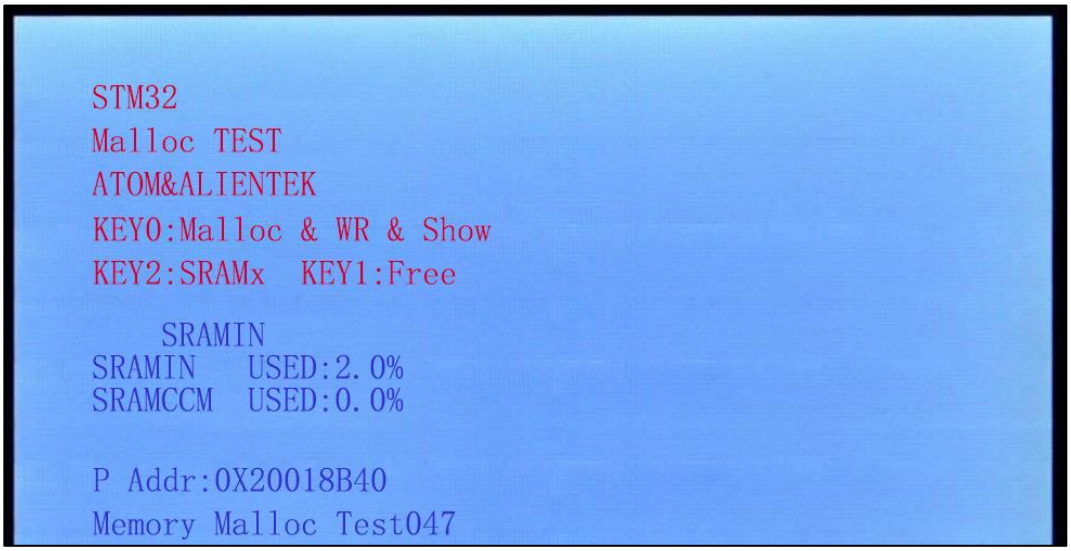


图 35.4.2 按下 KEY0 申请了部分内存

本章，我们还可以借助 USMART，测试内存的分配和释放，有兴趣的朋友可以动手试试。如图 35.4.3 所示：



图 35.4.3 USMART 测试内存管理图

图中，我们先申请了 4660 字节的内存，然后得到申请到的内存首地址为：0x24075940，说明我们申请内存成功（如果不成功，则会收到 0），然后释放内存的时候，参数是指针的地址，即执行：myfree(0x24075940)，就可以释放我们申请到的内存。其他情况，大家可以自行测试并分析。

第三十六章 FPU 测试(Julia 分形)实验

本章我们将学习如何开启 STM32F407 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。

本章分为如下几个小节：

36.1 FPU&Julia 分形简介
36.2 硬件设计
36.3 程序设计
36.4 下载验证

36.1 FPU&Julia 分形简介

本节将分别介绍 STM32F407 的 FPU 和 Julia 分形。

36.1.1 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。

STM32F407 属于 Cortex M4 架构，带有 32 位单精度硬件 FPU，支持浮点指令集，相对于 Cortex M0 和 Cortex M3 等，高出数十倍甚至上百倍的运算性能。

STM32F407 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 STM32F407 的硬件 FPU，该寄存器各位描述如图 36.1.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								CP11		CP10		Reserved			
								rw		rw					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

图 36.1.1.1 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 STM32F407 内置的硬件 FPU 了。CPACR 寄存器这 4 个位的设置，我们在 system_stm32f4xx.c 文件里面开启，代码如下：

```
void SystemInit (void)
{
  /* 省略部分代码 */
  /* 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 STM32F407 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 STM32F407 是带 FPU 功能的，所以在我们的 stm32f407xx.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             1U       /* FPU present */
```

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 stm32f407xx.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 Single Precision，如图 36.1.1.2 所示：

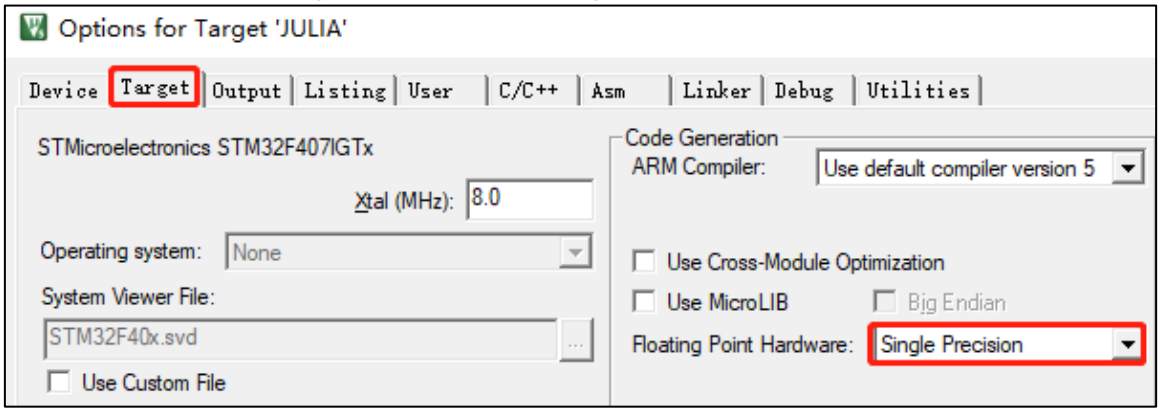


图 36.1.1.2 编译器开启硬件 FPU 选型

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 STM32F407 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 Not Used 选项设置为：Single Precision。

经过这两步设置，我们的编写的浮点运算代码，即可使用 STM32F407 的硬件 FPU 了，可以大大加快浮点运算速度。

36.1.2 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 */

/**
 * @brief       产生 Julia 分形图形
 * @param       size_x   : 屏幕 x 方向的尺寸
 * @param       size_y   : 屏幕 y 方向的尺寸
 * @param       offset_x : 屏幕 x 方向的偏移
 * @param       offset_y : 屏幕 y 方向的偏移
 * @param       zoom     : 缩放因子
 * @retval      无
 */
void julia_generate_fpu(uint16_t size_x, uint16_t size_y, uint16_t offset_x,
                        uint16_t offset_y, uint16_t zoom)
{
```

```
    uint8_t i;
    uint16_t x, y;
    float tmp1, tmp2;
    float num_real, num_img;
    float radius;

    for (y = 0; y < size_y; y++)
    {
        for (x = 0; x < size_x; x++)
        {
            num_real = y - offset_y;
            num_real = num_real / zoom;
            num_img = x - offset_x;
            num_img = num_img / zoom;
            i = 0;
            radius = 0;

            while ((i < ITERATION - 1) && (radius < 4))
            {
                tmp1 = num_real * num_real;
                tmp2 = num_img * num_img;
                num_img = 2 * num_real * num_img + IMG_CONSTANT;
                num_real = tmp1 - tmp2 + REAL_CONSTANT;
                radius = tmp1 + tmp2;
                i++;
            }
            LCD->LCD_RAM = g_color_map[i]; /* 绘制到屏幕 */
        }
    }
}
```

这种算法非常有效地展示了 FPU 的优势：无需修改代码，只需在编译阶段激活或禁止 FPU（在 MDK Code Generation 的 Float Point Hardware 选项里面设置：Single 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。注意：本实验不支持 RGB 屏，所以 RGB 屏的情况忽略即可。

36.2 硬件设计

1. 例程功能

1、实验开机后，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 6，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/KEY1 用于调节缩放因子，KEY2 用于设置自动缩放，还是手动缩放。

2、LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键 ：
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4

5）FPU(浮点计算单元)
6）定时器 6

36.3 程序设计

36.3.1 程序流程图

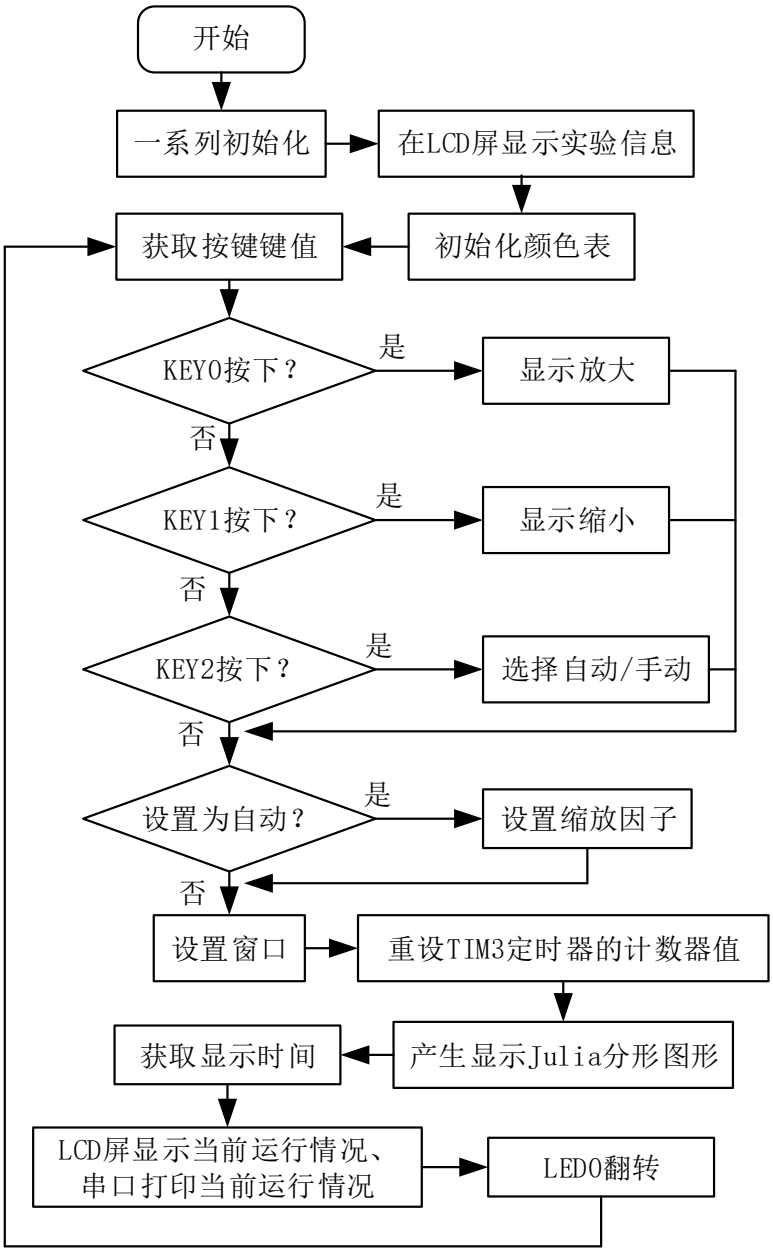


图 36.3.1.1 FPU 测试(Julia 分形)实验程序流程图

36.3.2 程序解析

本实验例程，分成两个工程：
1，实验 24_1 FPU 测试(Julia 分形)实验_开启硬件 FPU
2，实验 24_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 Single 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 24_1 FPU 测试(Julia 分形)实验_开启硬件 FPU。

1. TIMER 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。由于要统计帧时间，所以我们需要用到 TIMER 驱动代码的 btim.c 和 btim.h 文件。

btim.h 文件不需要做任何修改，只要在 btim.c 文件中，修改定时器更新中断回调函数即可，具体修改后的代码如下：

```
extern uint8_t g_timeout ;      /* 在 main.c 里面定义 */

/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim == (&g_timx_handle))
    {
        g_timeout++;
        //LED1_TOGGLE();
    }
}
```

在定时器更新中断回调函数中，我们只是需要让 g_timeout 自加就行，最后把 g_timeout 变量的值 * 定时器更新中断的周期就可以知道帧时间了。当然在程序上，我们还要声明 g_timeout 变量。

2. main.c 代码

下面是 main.c 的程序，具体如下：

```
/* 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 */
uint16_t g_color_map[ITERATION];

/* 缩放因子列表 */
const uint16_t 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,
};

/**
 * @brief       初始化颜色表
 * @param       clut    : 颜色表指针
 * @retval      无
 */
void julia_clut_init(uint16_t *clut)
{
    uint32_t i = 0x00;
    uint16_t  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;
    }
}

/* RGB LCD 缓存*/
uint16_t g_lcdbuf[800];

/**
 * @brief       产生 Julia 分形图形
 * @param       size_x      : 屏幕 x 方向的尺寸
 * @param       size_y      : 屏幕 y 方向的尺寸
 * @param       offset_x    : 屏幕 x 方向的偏移
 * @param       offset_y    : 屏幕 y 方向的偏移
 * @param       zoom        : 缩放因子
 * @retval      无
 */
void julia_generate_fpu(uint16_t size_x, uint16_t size_y, uint16_t offset_x,
                        uint16_t offset_y, uint16_t zoom)
{
    uint8_t i;
    uint16_t x, y;
    float tmp1, tmp2;
    float num_real, num_img;
    float radius;

    for (y = 0; y < size_y; y++)
    {
        for (x = 0; x < size_x; x++)
        {
            num_real = y - offset_y;
            num_real = num_real / zoom;
            num_img = x - offset_x;
            num_img = num_img / zoom;
            i = 0;
            radius = 0;
            while ((i < ITERATION - 1) && (radius < 4))
            {
                tmp1 = num_real * num_real;
                tmp2 = num_img * num_img;
                num_img = 2 * num_real * num_img + IMG_CONSTANT;
                num_real = tmp1 - tmp2 + REAL_CONSTANT;
                radius = tmp1 + tmp2;
                i++;
            }
            LCD->LCD_RAM = g_color_map[i];    /* 绘制到屏幕 */
        }
    }
}

uint8_t g_timeout;

int main(void)
{
```

```
    uint8_t key;
    uint8_t i = 0;
    uint8_t autorun = 0;
    float time;
    char buf[50];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    btim_timx_int_init(65535, 8400 - 1);        /* 10Khz 计数频率,最大计时 6.5 秒超出 */
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "FPU TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:+    KEY1:-", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY2:AUTO/MANUL", RED);/*显示提示信息*/
    delay_ms(1200);
    julia_clut_init(g_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 KEY1_PRES:
                if (i) i--;
                else i = sizeof(zoom_ratio) / 2 - 1;
                break;
            case KEY2_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_write_ram_prepare();
        BTIM_TIMX_INT->CNT = 0;      /* 重设 TIM6 定时器的计数器值 */
        g_timeout = 0;
        julia_generate_fpu(lcddev.width, lcddev.height, lcddev.width / 2,
                           lcddev.height / 2, zoom_ratio[i]);
        time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
        sprintf(buf, "%s: zoom:%d  runtime:%0.1fms\r\n", SCORE_FPU_MODE,
                zoom_ratio[i], time / 10);
        lcd_show_string(5, lcddev.height - 5 - 12, lcddev.width - 5,
                        12, 12, buf, RED);   /* 显示当前运行情况 */
        printf("%s", buf);           /* 输出到串口 */
        LED0_TOGGLE();
    }
}
```

这部分程序，总共 3 个函数：julia_clut_init、julia_generate_fpu 和 main 函数。

julia_clut_init 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 36.2 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。

为了提高速度，我们还可以在 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 24_1 和实验 24_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 Single 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。

36.4 下载验证

将实验 24_1 的程序下载到开发板后，可以看到 LCD 首先显示一些实验相关的信息，然后开始显示 Julia 分形图，并显示相关参数，如图 36.4.1 所示：

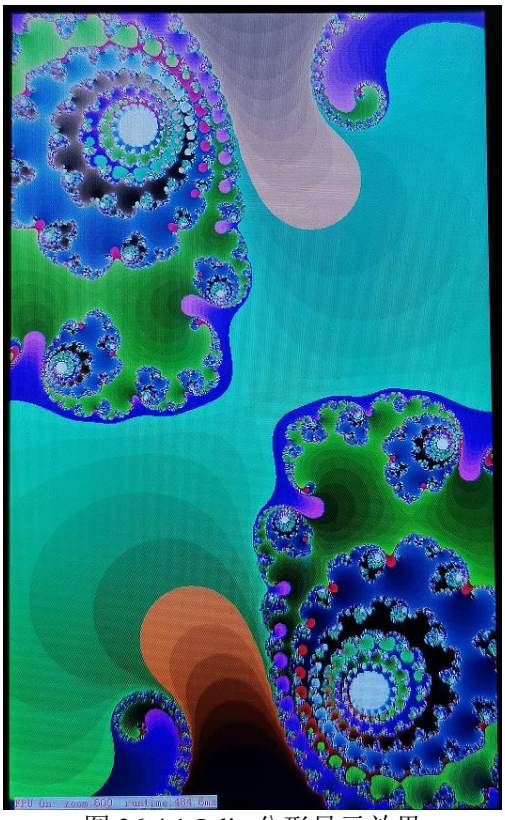


图 36.4.1 Julia 分形显示效果

实验 24_1 是开启了硬件 FPU 的，所以显示 Julia 分形图片速度比较快。除了 LCD 屏幕，还可以通过串口调试助手观察相关信息，如图 36.4.2 所示：



图 36.4.2 Julia 分形显示效果（开启硬件 FPU）

下面是关闭硬件 FPU 的运行情况，如图 36.4.3 所示：

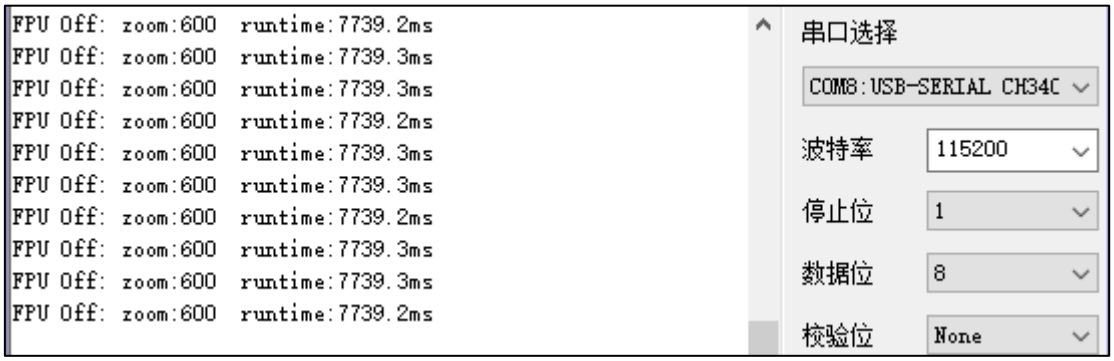


图 36.4.3 Julia 分形显示效果（关闭硬件 FPU）

对比图 36.4.2 和图 36.4.3 知道，关闭硬件 FPU 会比开启硬件 FPU，同等情况下慢 16 倍左右，这充分体现了 STM32F407 硬件 FPU 的优势。

第三十七章 DSP 测试实验

上一章，我们测试了 STM32F407 的硬件 FPU。STM32F407 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 STM32F407 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 STM32F407 的 DSP 库有个基本的了解。

本章分为如下几个小节：

37.1 DSP 简介与环境搭建
37.2 硬件设计
37.3 程序设计
37.4 下载验证

37.1 DSP 简介与环境搭建

本节将分两个部分：1，STM32F407 DSP 简介；2，DSP 库运行环境搭建。

37.1.1 STM32F407 DSP 简介

STM32F407 采用 Cortex-M4 内核，相比 Cortex-M3 系列除了内置硬件 FPU 单元，在数字信号处理方面还增加了 DSP 指令集，支持诸如单周期乘加指令（MAC），优化的单指令多数据指令（SIMD），饱和算数等多种数字信号处理指令集。相比 Cortex-M3，Cortex-M4 在数字信号处理能力方面得到了大大的提升。Cortex-M4 执行所有的 DSP 指令集都可以在单周期内完成，而 Cortex-M3 需要多个指令和多个周期才能完成同样的功能。

接下来来介绍 Cortex-M4 的两个 DSP 指令：MAC 指令（32 位乘法累加）和 SIMD 指令。

32 位乘法累加（MAC）单元包括新的指令集，能够在单周期内完成一个 32×32+64 → 64 的操作或两个 16×16 的操作，其计算能力，如表 37.1.1.1 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 37.1.1.1 32 位乘法累加（MAC）单元的计算能力

Cortex-M4 支持 SIMD 指令集，这在 Cortex-M3/M0 系列是不可用的。上述表中的指令，有的属于 SIMD 指令。与硬件乘法器一起工作（MAC），使所有这些指令都能在单个周期内执行。受益于 SIMD 指令的支持，Cortex-M4 处理器能在单周期内完成高达 32×32+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 37.1.1.2 所示：

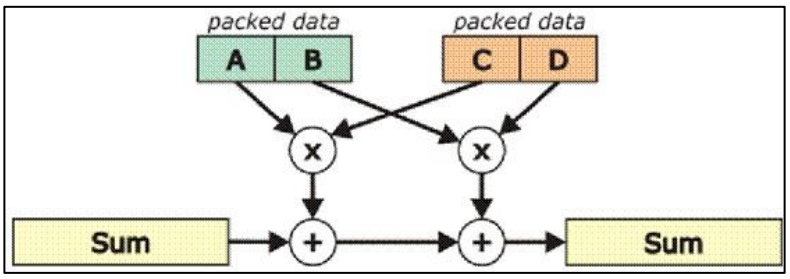


图 37.1.1.2 SUM 运算过程

以上图片所示的运算，即：SUM = SUM +（A* C）+（B *D），在 STM32F407 上面，可以被编译成由一条单周期指令完成。

上面我们简单的介绍了 Cortex-M4 的 DSP 指令，接下来我们来介绍一下 STM32F407 的 DSP 库。STM32F407 的 DSP 库源码和测试实例在 ST 提供的 HAL 库：en.stm32cubef4_v1-26-0_v1.26.0.zip 里面就有（该文件可以在 www.st.com 网站下载，搜索 STM32CubeF4 即可找到最新版本），该文件在：光盘→ 8，STM32 参考资料→1，STM32CubeF4 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，STM32CubeF4 固件包→ STM32Cube_FW_F4_V1.26.0→ Drivers→CMSIS→DSP，该文件夹下目录结构如图 37.1.1.3 所示：

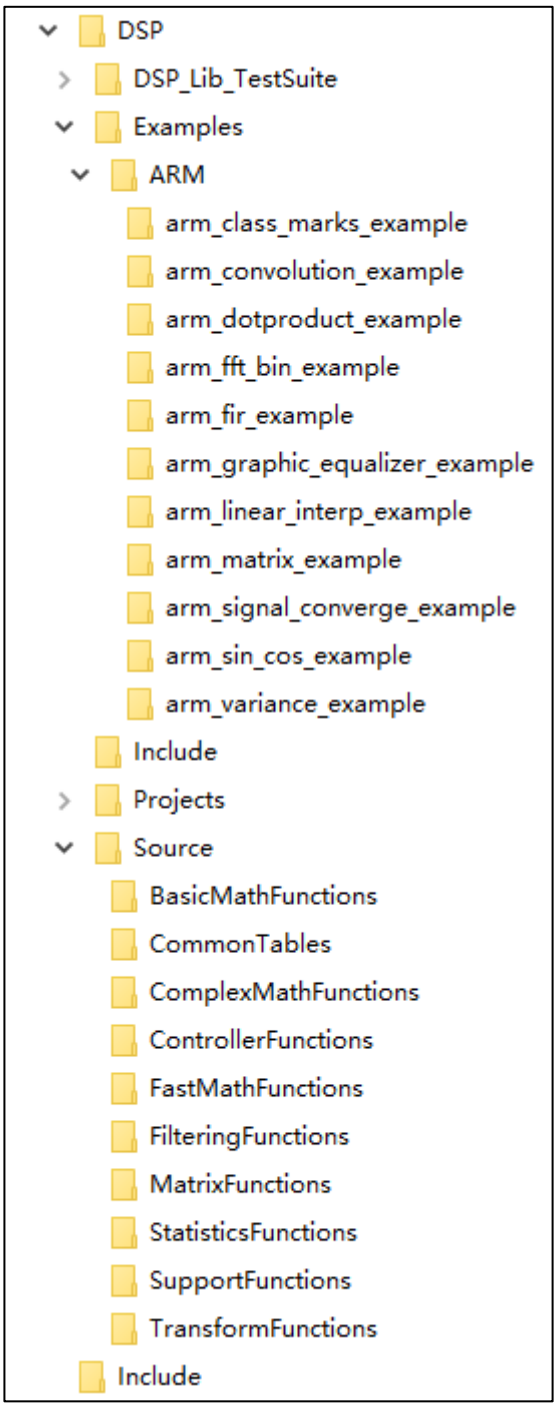


图 37.1.1.3 DSP_Lib 目录结构

DSP 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。arm_const_structs.c 文件提供一些常用的常量结构体，方便用户使用。
- **ComplexMathFunctions**
  复数学功能，如向量处理，求模运算。
- **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。

- **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 4 个.lib 文件，如下：

① arm_cortexM4b_math.lib (Cortex-M4 大端模式)
② arm_cortexM4l_math.lib (Cortex-M4 小端模式)
③ arm_cortexM4bf_math.lib (浮点 Cortex-M4 大端模式)
④ arm_cortexM4lf_math.lib (浮点 Cortex-M4 小端模式)

我们得根据所用 MCU 内核类型以及端模式来选择符合要求的.lib 文件，本章我们所用的 STM32F407 属于 Cortex-M4 内核，双精度浮点小端模式，应选择：

arm_cortexM4lf_math.lib (单精度浮点 Cortex-M4 小端模式)。

对于 DSP 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。

37.1.2 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 24_1）为基础，搭建 DSP 运行环境。

在 MDK 里面搭建 STM32F407 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，我们在实验 25_1 DSP BasicMath 测试实验\Drivers\CMSIS 目录下新建：DSP 和 Lib 文件夹，然后把官方的相应文件拉到我们的工程的相应位置：arm_cortexM4lf_math.lib 和相关头文件，再把没用到的文件删除，如图 37.1.2.1、图 37.2.2 所示：

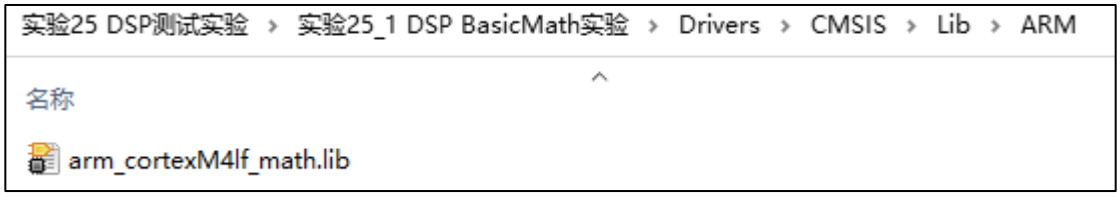


图 37.1.2.1 DSP_LIB 文件夹添加文件

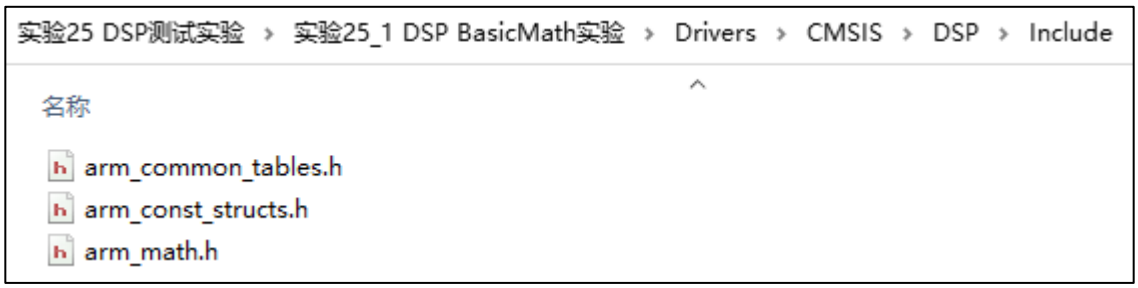


图 37.1.2.2 DSP_LIB 文件夹添加文件

这个步骤具体请参考工程源码文件，arm_cortexM4lf_math.lib 文件在 37.1.1 节已经介绍过了。Include 文件夹里面包含了我们可能要用到的相关头文件，所以要添加到工程中。

然后，打开工程，新建 Drivers/CMSIS 分组，并将 arm_cortexM4lf_math.lib 添加到工程里面，如图 37.1.2.3 所示：

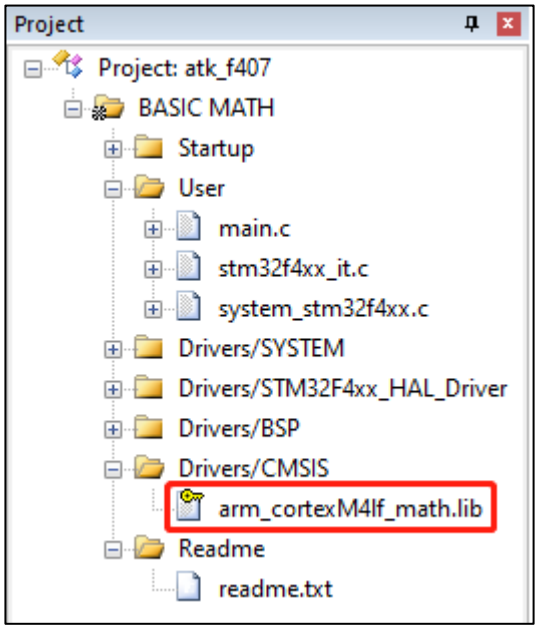


图 37.1.2.3 添加.lib 文件

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，这个我们工程都统一添加好了，所以不用额外再操作，这里是提醒一下大家要添加头文件包含路径，如图 37.1.2.4 所示：

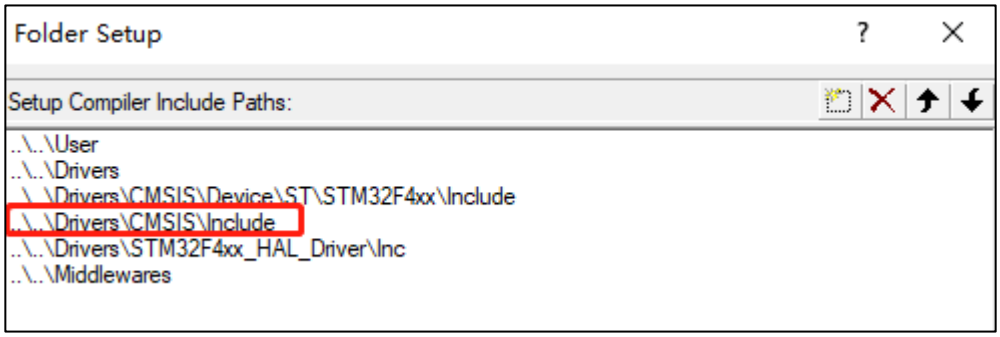


图 37.1.2.4 添加相关头文件包含路径

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，ARM_MATH_DSP,

2，ARM_MATH_MATRIX_CHECK,

3，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 37.1.2.5 所示：

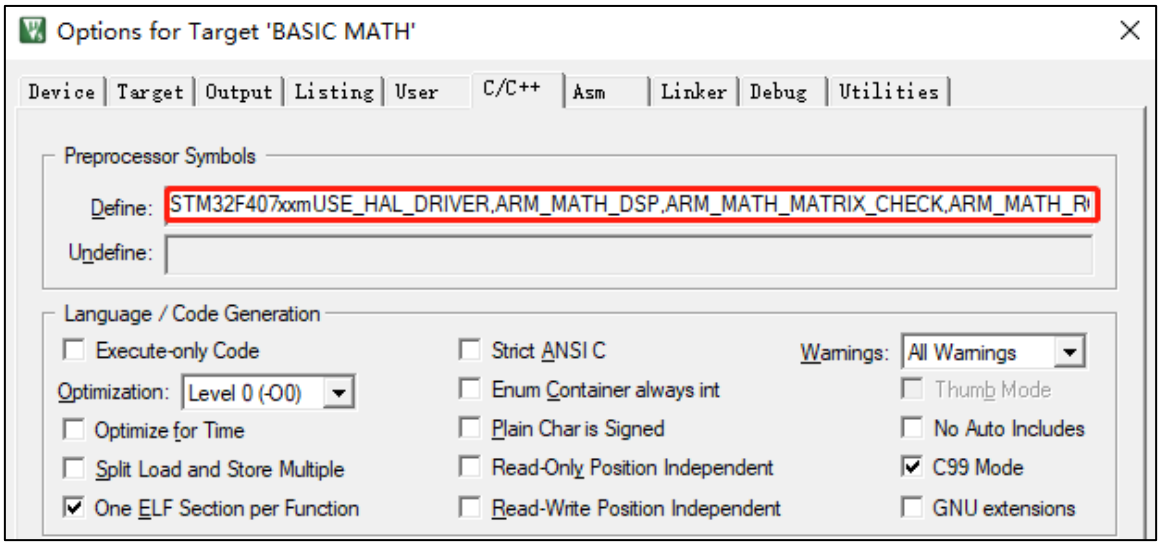


图 37.1.2.5 DSP 库支持全局宏定义设置

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 和__FPU_PRESENT，因为在 Target 选项卡设置 Floating Point Hardware 的时候（上一章有介绍），只要我们选择了：Double Precision（如果没有设置 Single Precision，则必须设置），MDK 就会自动添加这两个全局宏，因此不需要我们手动添加了。这样，在 Define 处要输入的所有宏为：STM32F4xx,USE_HAL_DRIVER,ARM_MATH_DSP,ARM_MATH_MATRIX_CHECK,ARM_MATH_ROUNDING 等共 5 个。

至此，STM32F407 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。

37.2 硬件设计

1. 例程功能

本例程包含 2 个源码：**实验 25_1 DSP BasicMath 测试实验**和**实验 25_2 DSP FFT 测试实验**，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 25_1 DSP BasicMath 测试实验功能简介：测试 STM32F407 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。LED0 闪烁，提示程序运行。

实验 25_2 DSP FFT 测试实验功能简介：测试 STM32F407 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。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键：
KEY0 - PE2
5）FPU(浮点计算单元)
6）定时器 6

37 程序设计

本章代码，分成两个工程：1，实验 25 DSP BasicMath 测试；2，实验 25 DSP FFT 测试，接下来我们分别介绍。

37.1 DSP BasicMath 测试

这是我们使用 STM32F407 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。

STM32F407 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 37.1.2 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：

```
#define DELTA   0.0001f         /* 误差值 */

/**
 * @brief       sin cos 测试
 * @param       angle : 起始角度
 * @param       times : 运算次数
 * @param       mode  : 是否使用 DSP 库
 *   @arg       0 , 不使用 DSP 库;
 *   @arg       1 , 使用 DSP 库;
 *
 * @retval      无
 */
uint8_t sin_cos_test(float angle, uint32_t times, uint8_t mode)
{
    float sinx, cosx;
    float result;
    uint32_t 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 */
}

uint8_t g_timeout;

int main(void)
{
    float time;
    char buf[50];
    uint8_t res;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    btim_timx_int_init(65535, 8400 - 1);        /* 10Khz 计数频率,最大计时 6.5 秒超出 */

    /*显示提示信息*/
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DSP BasicMath TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 120, 200, 16, 16, "No DSP runtime:", RED);
    lcd_show_string(30, 150, 200, 16, 16, "Use DSP runtime:", RED);
    while (1)
    {
        /* 不使用 DSP 优化 */
        BTIM_TIMX_INT->CNT = 0;    /* 重设 TIM3 定时器的计数器值 */
        g_timeout = 0;
        res = sin_cos_test(PI / 6, 200000, 0);
        time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
        sprintf(buf, "%0.1fms\r\n", time / 10);

        if (res == 0)
        {
            /* 显示运行时间 */
            lcd_show_string(30 + 16 * 8, 120, 100, 16, 16, buf, BLUE);
        }
        else
        {
            /* 显示当前运行情况 */
            lcd_show_string(30 + 16 * 8, 120, 100, 16, 16, "error! ", BLUE);
        }
        /* 使用 DSP 优化 */
        BTIM_TIMX_INT->CNT = 0;    /* 重设 TIM6 定时器的计数器值 */
        g_timeout = 0;
        res = sin_cos_test(PI / 6, 200000, 1);
        time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
        sprintf(buf, "%0.1fms\r\n", time / 10);
        if (res == 0)
        {
            /* 显示运行时间 */
            lcd_show_string(30 + 16 * 8, 150, 100, 16, 16, buf, BLUE);
        }
        else
        {
            /* 显示错误 */
            lcd_show_string(30 + 16 * 8, 150, 100, 16, 16, "error! ", BLUE);
        }
        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 6 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。

37.3.2 DSP FFT 测试

这是我们使用 STM32F407 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 STM32F407 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。

STM32F407 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 4n，而基 2 的 FFT 输入点数则必须是 2n，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：

```
/* Deprecated */
arm_status arm_cfft_radix4_init_f32(arm_cfft_radix4_instance_f32 * S,
                    uint16_t fftLen, uint8_t ifftFlag, uint8_t bitReverseFlag);

/* Deprecated */
void arm_cfft_radix4_f32(const arm_cfft_radix4_instance_f32 * S,
```

```
                        float32_t * pSrc);

/**
 * @brief       Floating-point complex magnitude
 * @param[in]   pSrc        points to the complex input vector
 * @param[out]  pDst        points to the real output vector
 * @param[in]   numSamples  number of complex samples in the input vector
 */
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 25_2 DSP FFT 测试实验）同样是在 37.1.2 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：

```
/* FFT 长度，默认是 1024 点 FFT
 * 可选范围为: 16, 64, 256, 1024.
 */
#define FFT_LENGTH      1024

float fft_inputbuf[FFT_LENGTH * 2];     /* FFT 输入数组 */
float fft_outputbuf[FFT_LENGTH];        /* FFT 输出数组 */

uint8_t g_timeout;

int main(void)
{
    float time;
    char buf[50];
    arm_cfft_radix4_instance_f32 scfft;
    uint8_t key, t = 0;
    uint16_t i;

    sys_cache_enable();                     /* 打开 L1-Cache */
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(240, 2, 2, 4);     /* 设置时钟, 480Mhz */
    delay_init(480);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    mpu_memory_protection();                /* 保护相关存储区域 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    key_init();                             /* 初始化按键 */
    btim_timx_int_init(65535, 8400 - 1);    /* 10Khz 计数频率,最大计时 6.5 秒超出 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DSP FFT TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY0:Run FFT", RED); /* 显示提示信息 */
    lcd_show_string(30, 160, 200, 16, 16, "FFT runtime:", RED); /* 显示提示信息 */
    /* 初始化 scfft 结构体，设定 FFT 相关参数 */
```

```
    arm_cfft_radix4_init_f32(&scfft, FFT_LENGTH, 0, 1);

    while (1)
    {
        key = key_scan(0);
        if (key == KEY0_PRES)
        {
            for (i = 0; i < FFT_LENGTH; i++)   /* 生成信号序列 */
            {
                /* 生成输入信号实部 */
                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);
                fft_inputbuf[2 * i + 1] = 0; /* 虚部全部为 0 */
            }

            BTIM_TIMX_INT->CNT = 0;; /* 重设 BTIM_TIMX_INT 定时器的计数器值 */
            g_timeout = 0;

            arm_cfft_radix4_f32(&scfft, fft_inputbuf);      /* FFT 计算（基 4） */
            /* 计算所用时间 */
            time = BTIM_TIMX_INT->CNT + (uint32_t)g_timeout * 65536;
            sprintf((char *)buf, "%0.3fms\r\n", time / 10);
            /* 显示运行时间 */
            lcd_show_string(30 + 12 * 8, 160, 100, 16, 16, buf, BLUE);
            /* 把运算结果复数求模得幅值 */
            arm_cmplx_mag_f32(fft_inputbuf, fft_outputbuf, FFT_LENGTH);

            printf("\r\n%d point FFT runtime:%0.3fms\r\n", FFT_LENGTH,time / 10);
            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 % 20) == 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：

```
/* 生成输入信号实部 */
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);
```

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。

37.4 下载验证

将实验 25_1 DSP BasicMath 测试实验的程序下载到开发板后，可以在屏幕看到两种实现方式的速度差别，如图 37.4.1 所示：

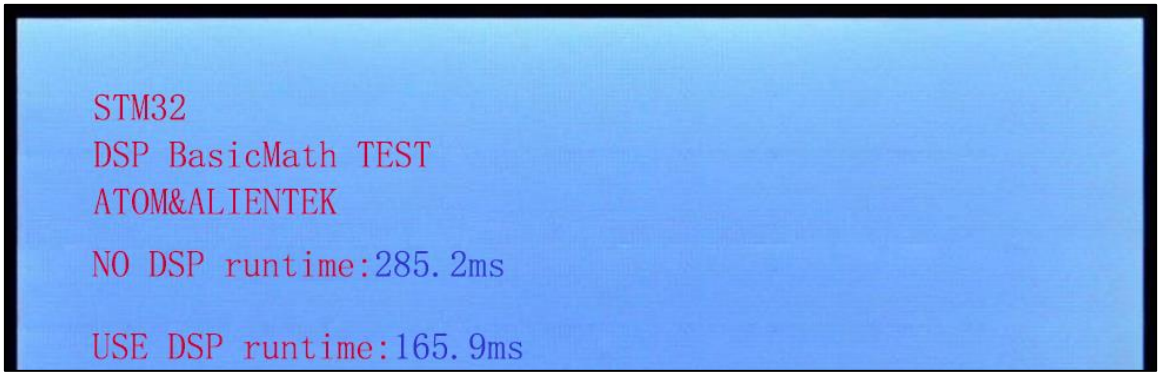


图 37.4.1 使用 DSP 库和不使用 DSP 库的基础数学函数速度对比

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 STM32F407 的 DSP 库，速度上面比传统的实现方式提升了约 41%。

对于实验 25_2 DSP FFT 测试实验，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 37.4.2 所示：

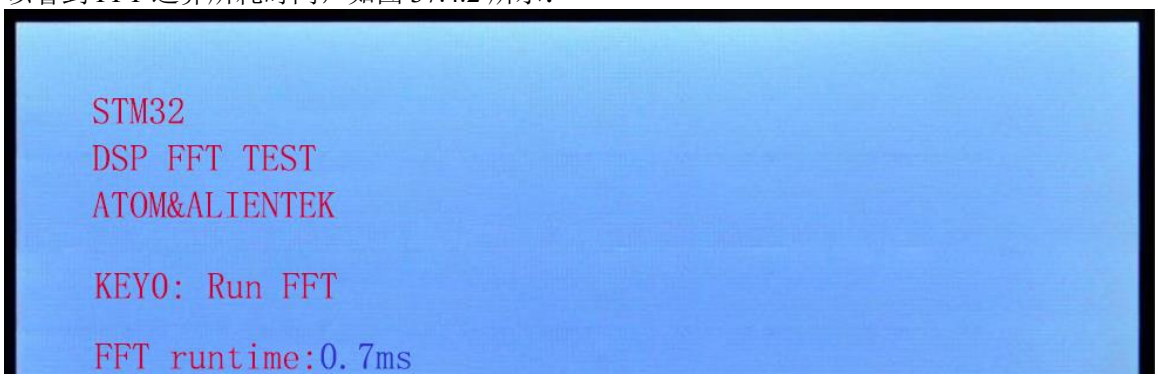


图 37.4.2 FFT 测试界面

可以看到，STM32F407 采用基 4 法计算 1024 个浮点数的 FFT，仅用了 0.7ms，速度是非常的快。同时，可以在串口看到 FFT 变换取模后的各频点模值，如图 37.4.3 所示：



图 37.4.3 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 37.3.1 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：5119.950684、15359.999023、256000，同理论值非常接近。

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。

第三十八章 串口 IAP 实验

IAP，即在应用编程，通俗地说法就是“程序升级”。产品阶段设计完成后，在脱离实验室的调试环境下，如果想对产品做功能升级或 BUG 修复会十分麻烦，如果硬件支持，在出厂时预留一套升级固件的流程，就可以很好解决这个问题，IAP 技术就是为此而生的。在之前的 FLASH 模拟 EEPROM 实验里面，我们学习了 STM32F407 的 FLASH 自编程，本章我们将结合 FLASH 自编程的知识，通过 STM32F407 的串口实现一个简单的 IAP 功能。

本章分为如下几个小节：

38.1 IAP 简介
38.2 硬件设计
38.3 程序设计
38.4 下载验证

38.1 IAP 简介

IAP（In Application Programming）即在应用编程。在讲解 STM32 的启动模式时我们已经知道 STM32 可以通过设置 MSP 的方式从不同的地址启动：包括 Flash 地址、RAM 地址等，在默认方式下，我们的嵌入式程序是以连续二进制的方式烧录到 STM32 的可寻址 Flash 区域上的。如果我们用的 Flash 容量大到可以存储两个或多个的完整程序，在保证每个程序完整的情况下，上电后的程序通过修改 MSP 的方式，就可以保证一个单片机上有多个有功能差异的嵌入式软件，这就是我们要讲解的 IAP 的设计思路。

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，它做如下操作：

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 STM32F407 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。

STM32F407 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。

我们先来看看 STM32F407 正常的程序运行流程（为了方便说明 IAP 过程，我们仅考虑代码全部存放在内部 FLASH 的情况），如图 38.1.1 所示：

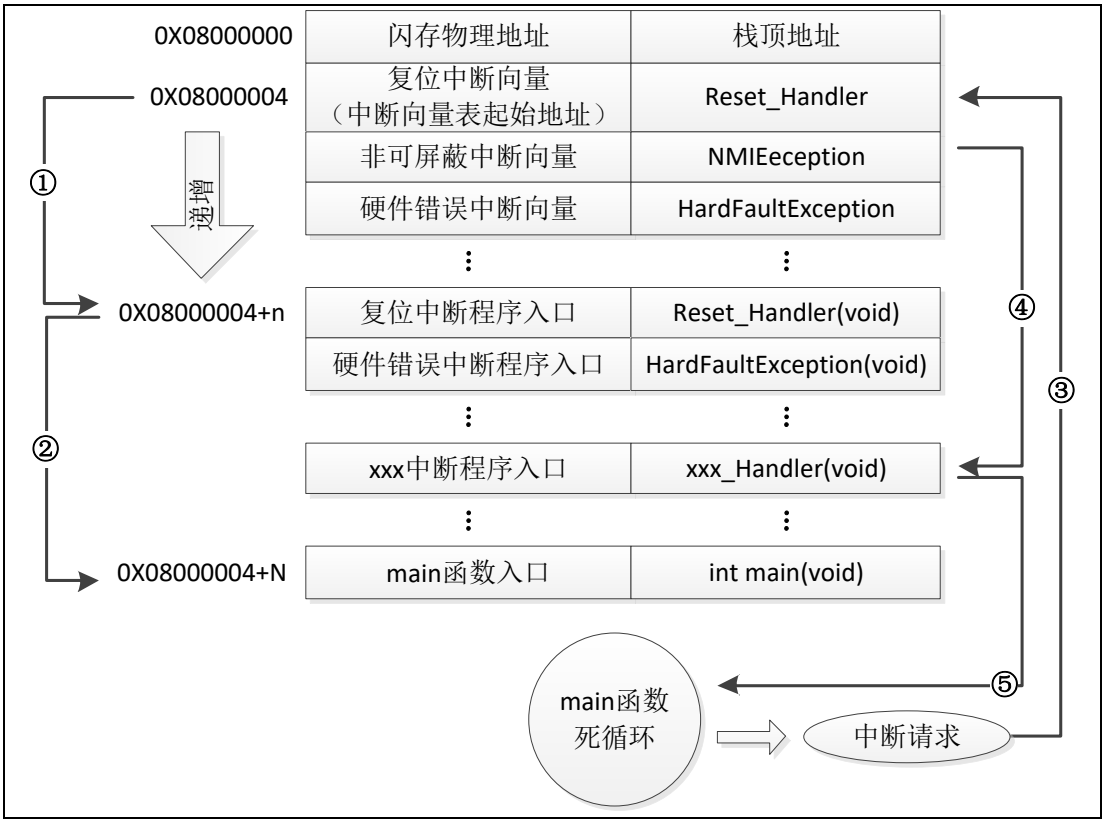


图 38.1.1 STM32F407 正常运行流程图

STM32F407 的内部闪存（FLASH）地址起始于 0x0800 0000，一般情况下，程序文件就从此地址开始写入。此外 STM32F407 是基于 Cortex-M4 内核的微控制器，其内部通过一张“中断向量表”来响应中断，程序启动后，将首先从“中断向量表”取出复位中断向量执行复位中断程序完成启动，而这张“中断向量表”的起始地址是 0x08000004，当中断来临，STM32F407 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 38.1.1 中，STM32F407 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 STM32F407 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 38.1.2 所示：

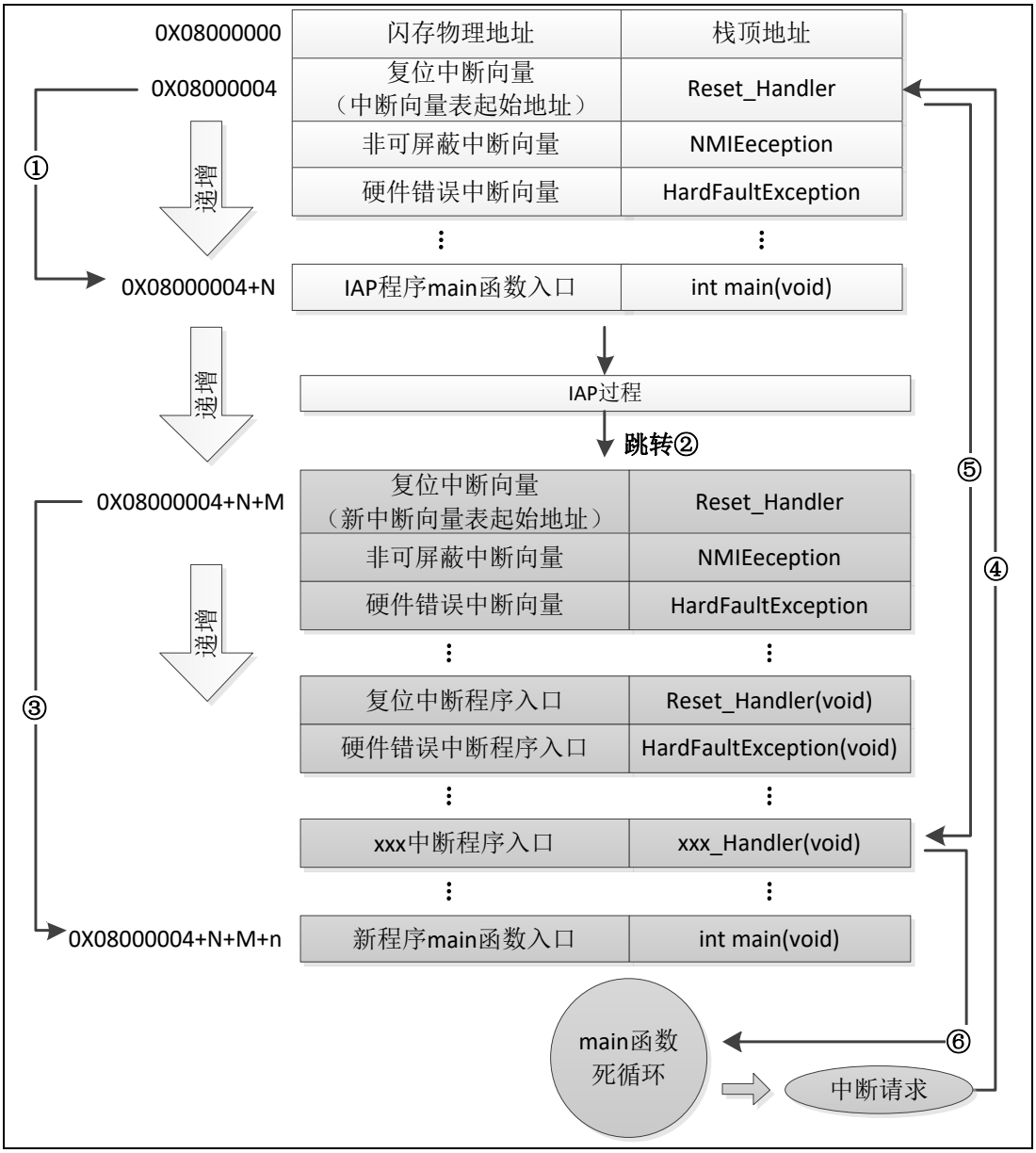


图 38.1.2 加入 IAP 之后程序运行流程图

在图 38.1.2 所示流程中，STM32F407 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 38.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32F407 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 STM32F407 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 指针仍然会强制跳转到地址 0x08000004 中断向量表处，而不是新程序的中断向量表，如图标号④所示；程序再根据我们设置的中断向量表偏移量，跳转到对应中断源新的中断服务程序中，如图标号⑤所示；在执行完中断服务程序后，程序返回 main 函数继续运行，如图标号⑥所示。

通过以上两个过程的分析，我们知道 IAP 程序必须满足两个要求：

1） 新程序必须在 IAP 程序之后的某个偏移量为 x 的地址开始；

2） 必须将新程序的中断向量表相应的移动，移动的偏移量为 x；

本章，我们有 2 个 APP 程序：

1，FLASH APP 程序，即只运行在内部 FLASH 的 APP 程序。

2，SRAM APP 程序，即只运行在内部 SRAM 的 APP 程序，其运行过程和图 38.1.2 相似，不过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

APP 程序使用以前例程即可，不需要对程序进行修改。

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。

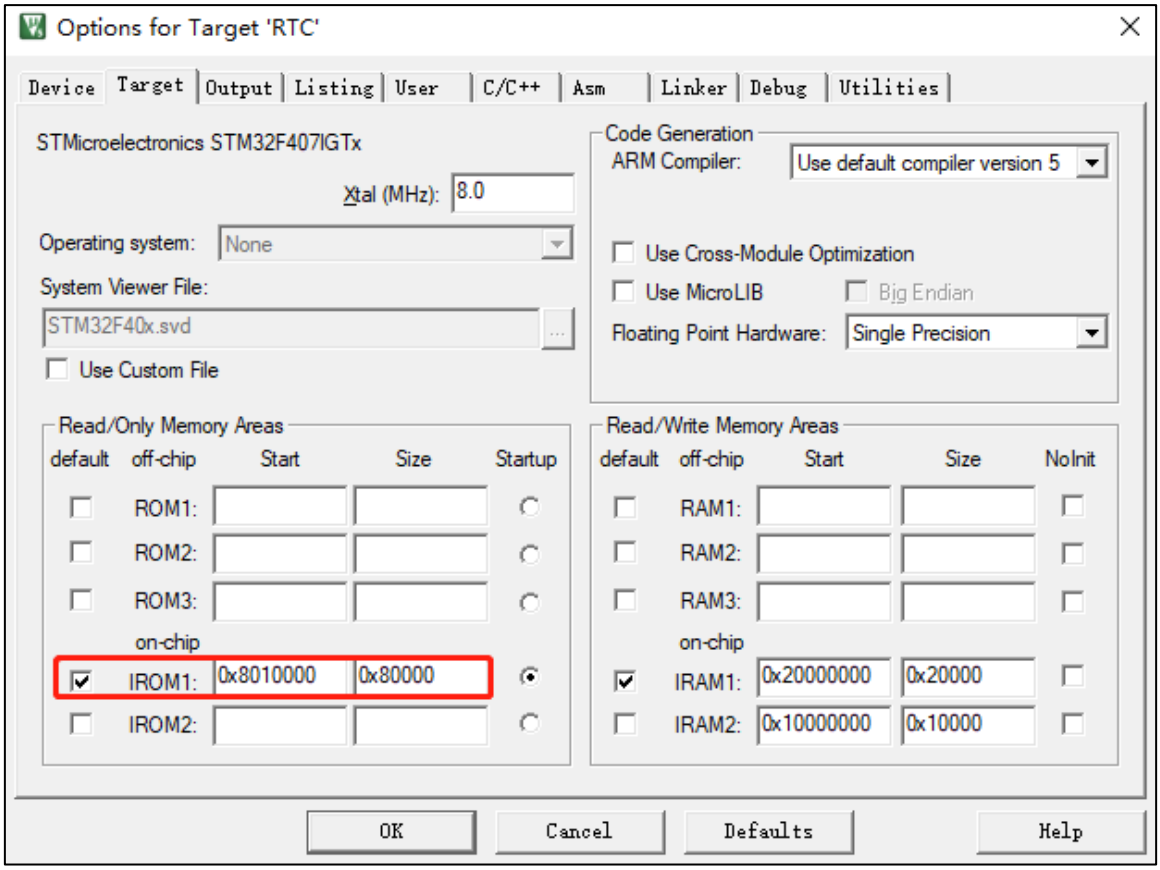


图 38.1.3 FLASH APP Target 选项卡设置

图 38.1.3 中，我们设置起始地址（Start）为 0x08010000，即偏移量为 0x80000（64K 字节，即留给 BootLoader 的空间），因而，留给 APP 用的 FLASH 空间（Size）为 0x80000-0x10000=0x70000（448KB）。设置好 Start 和 Size，就完成 APP 程序的起始地址设置。IRAM 是内存的地址，APP 可以独占这些内存，我们不需要修改。

注意：需要确保 APP 起始地址在 Bootloader 程序结束位置之后，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 38.1.4 所示：

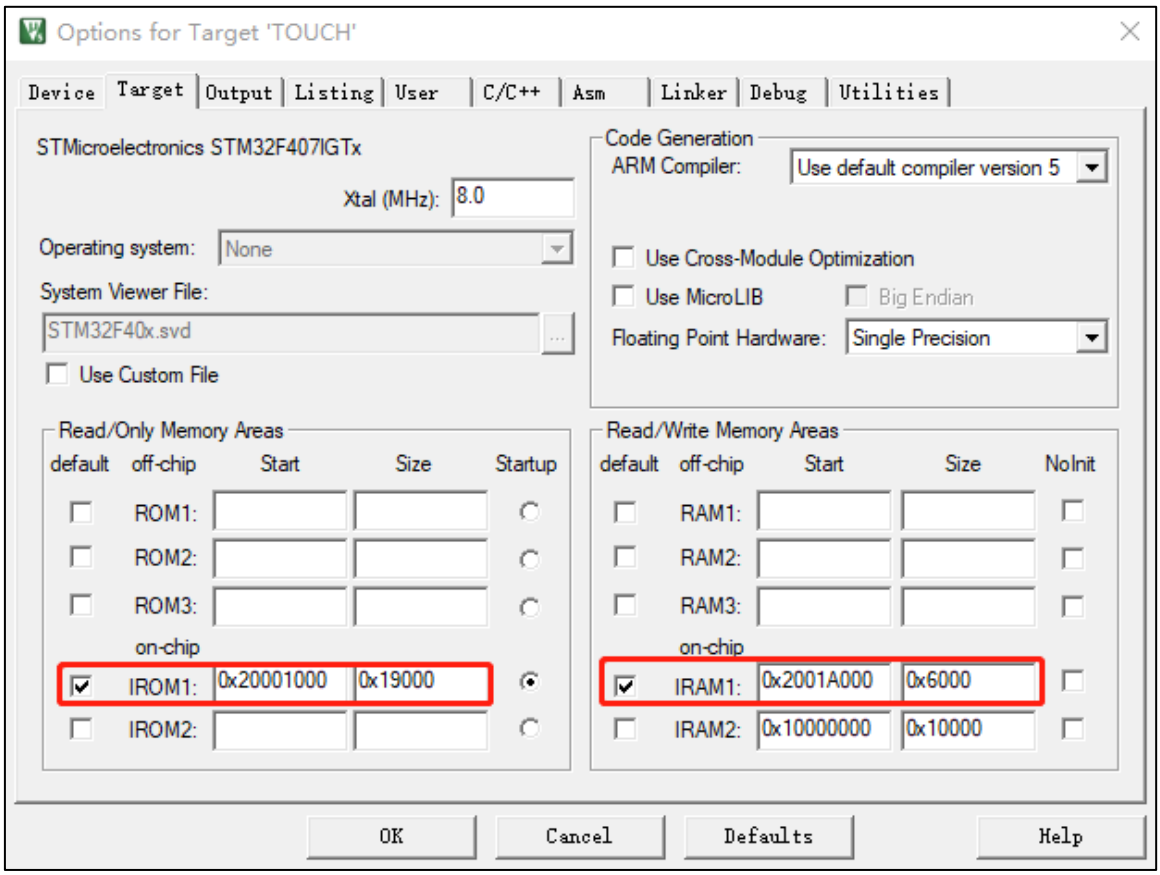


图 38.1.6 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址（Start）定义为：0x20001000，大小为 0x19000（100K 字节），即从地址 0x20000000 偏移 0x1000 开始，存放 SRAM APP 代码。这个分配关系大家可以根据自己的实际情况修改，由于 STM32F407IGT6 只有一个 128K(不算 CCM)的片内 SRAM，存放程序的位置与变量的加载位置不能重复，所以我们需要设置 IRAM1 中的地址（SRAM）的起始地址变为 0x2001A000,分配大小只有 0x6000(24K 字节)。整个 STM32F407IGT6 的 SRAM（不含 CCM）分配情况为：最开始的 4K 给 Bootloader 程序使用，随后的 100K 存放 APP 程序，最后 24K，用作 APP 程序的内存。

关于 APP 起始地址的设置方法，我们就介绍到这里，大家可以根据自己项目的实际需求进行修改。

2. 中断向量表的偏移量设置方法

VTOR 寄存器存放的是中断向量表的起始地址。默认的情况它由 BOOT 的启动模式决定，对于 STM32F407 来说就是指向 0x0800 0000 这个位置，也就是从默认的启动位置加载中断向量等信息,不过 ST 允许重定向这个位置，这样就可以从 Flash 区域的任意位置启动我们的代码了。我们可以通过调用 sys.c 里面的 sys_nvic_set_vector_table 函数实现，该函数定义如下：

```
/**
 * @brief       设置中断向量表偏移地址
 * @param       baseaddr : 基址
 * @param       offset   : 偏移量
 * @retval      无
 */
void sys_nvic_set_vector_table(uint32_t baseaddr, uint32_t offset)
{
    /* 设置 NVIC 的向量表偏移寄存器,VTOR 低 9 位保留,即[8:0]保留 */
    SCB->VTOR = baseaddr | (offset & (uint32_t)0xFFFFFE00);
}
```

该函数用于设置中断向量偏移，baseaddr 为基地址（即 APP 程序首地址），Offset 为偏移量，需要根据自己的实际情况进行设置。比如 FLASH APP 设置中断向量表偏移量为 0x10000，调用

情况如下：

```
/* 设置中断向量表偏移量为 0x10000 */
sys_nvic_set_vector_table(FLASH_BASE, 0x10000);
```

这是设置 FLASH APP 的情况，SRAM APP 的情况可以参考触摸屏实验_SRAM APP 版本，其具体的调用情况请看到 main 函数。

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，如果安装在 C 盘的默认路径，它的位置是 D:\Keil_v5\ARM\ARMCC\bin\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》。

本实验，我们可以通过在 MDK 点击 Options for Target→User 选项卡，在 After Build/Rebuild 一栏中，勾选 Run #1，我们推荐使用相对地址，在勾选的同一行后的输入框并写入命令行：fromelf --bin -o ..\..\Output\@L.bin ..\..\Output\%L，如图 38.1.7 所示：

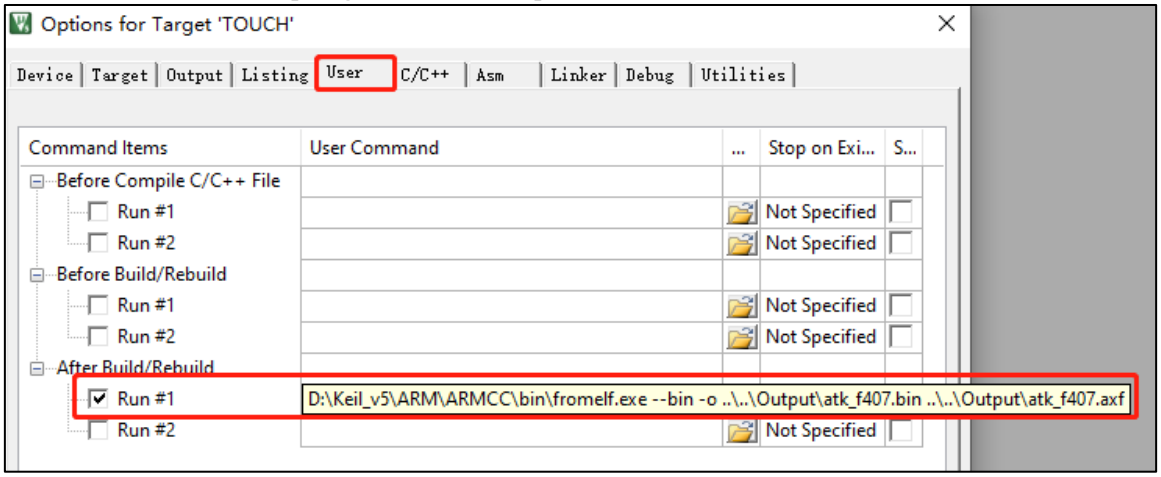


图 38.1.7 设置生成编译结果文件名

通过这一步设置，我们就可以在 MDK 编译成功之后，调用 fromelf.exe，..\..\Output\%L 表示当前编译的链接文件（..\是相对路径，表示上级目录，编译器默认从工程文件*.uvprojx 开始查找，根据我的工程文件 Output 的位置就能明白路径的含义），指令--bin –o ..\..\Output\@L.bin 表示在 Output 目录下生成一个.bin 文件，@L 在 Keil 的下表示 Output 选项卡下的 Name of Executable 后面的字符串，即在 Output 文件夹下生成一个 atk_f407.bin 文件。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

此步，通过调用 sys_nvic_set_vector_table 函数，实现对中断向量表偏移量的设置。

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，就可以得到一个.bin 的 APP 程序，通过 Bootlader 程序即可实现更新。

38.2 硬件设计

1. 例程功能

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 KEY2 按键，将串口接收到的 APP 程序存放到 STM32F407 的 FLASH，之后再按 KEY1 即可以执行这个 FLASH APP 程序。LED0 用于指示程序运行状态。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/B7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键：
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4

38.3 程序设计

38.3.1 程序流程图

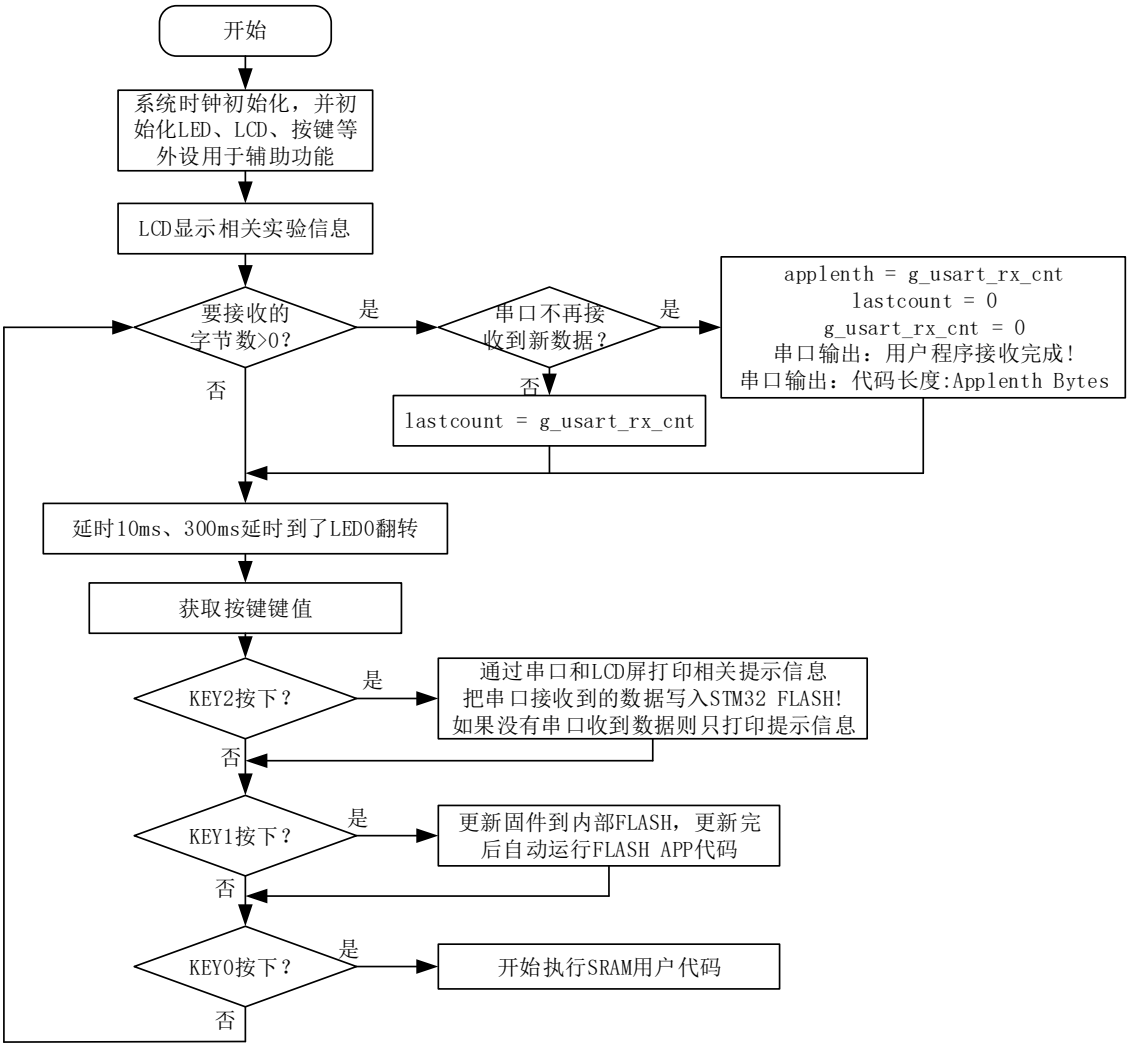


图 38.3.1.1 串口 IAP 实验程序流程图

IAP 我们设置为有按键才跳转的方式，可以用串口接收不同的 APP，再根据按键选择跳转到具体的 APP（Flash APP 或者 SRAM APP），方便我们进行验证和记忆。

38.3.2 程序解析

本实验，我们总共需要 3 个程序（1 个 IAP，2 个 APP）：

1、FLASH IAP Bootloader，起始地址为 0x08000000，设置为我们用于升级的跳转的程序，我们将用串口 1 来作数据接收程序，通过按键功能手动跳转到指定 APP。

2、FLASH APP，仅使用 STM32 内部 FLASH，大小为 120KB。本程序使用：实验 10 RTC 实验，作为 FLASH APP 程序（起始地址为 0x08010000）

3、SRAM APP，使用 STM32 内部 SRAM，生成的 bin 大小为 47KB。本程序使用：实验 15 触摸屏实验，作为 SRAM APP 程序（起始地址为 0x20001000）。

本章关于 APP 程序的生成和修改比较简单，我们就不细说，请大家结合光盘源码以及 38.1 节的介绍，自行理解。本章程序解析小节仅针对 Bootloader 程序。

1. IAP 程序

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码，IAP 的驱动主要包括两个文件：iap.c 和 iap.h。

由于 STM32 芯片的 Flash 的容量一般要比 SRAM 要大，所以我们只编写对 Flash 的写功能和对 MSP 的设置功能以实现程序的跳转。写 STM32 内部 Flash 的功能我们用到 STM32 的 Flash 操作，通过封装之前《实验 22 FLASH 模拟 EEPROM 实验》的驱动，我们实现 IAP 的写 Flash 操作如下：

```
/**
 * @brief       IAP 写入 APP BIN
 * @param       appxaddr : 应用程序的起始地址
 * @param       appbuf   : 应用程序 CODE
 * @param       appsize  : 应用程序大小(字节)
 * @retval      无
 */
void iap_write_appbin(uint32_t appxaddr, uint8_t *appbuf, uint32_t appsize)
{
    uint16_t t;
    uint16_t i = 0;
    uint16_t temp;
    uint32_t fwaddr = appxaddr; /* 当前写入的地址 */
    uint8_t *dfu = appbuf;
    for (t = 0; t < appsize; t += 4)
    {
        temp  = (uint16_t)dfu[3] << 24;
        temp |= (uint16_t)dfu[2] << 16;
        temp |= (uint16_t)dfu[1] << 8;
        temp |= (uint16_t)dfu[0];
        dfu += 4;                   /* 偏移 2 个字节 */
        g_iapbuf[i++] = temp;

        if (i == 512)
        {
            i = 0;
            stmflash_write(fwaddr, g_iapbuf, 512);
            fwaddr += 2048;         /* 偏移 2048  16 = 2 * 8  所以要乘以 2 */
        }
    }
    if (i)
    {
        stmflash_write(fwaddr, g_iapbuf, i);    /* 将最后的一些内容字节写进去 */
    }
}
```

在保存了一个完整的 APP 到了对应的位置后，我们需要对栈顶进行检查操作，初步检查程序设置正确再进行跳转。我们以 Flash APP 为例，用 bin 文件查看工具（A 盘→6，软件资料→1，软件→5，其他小工具→winhex），可以看到 bin 的内容默认为小端结构，如图 38.3.2.1 所示。

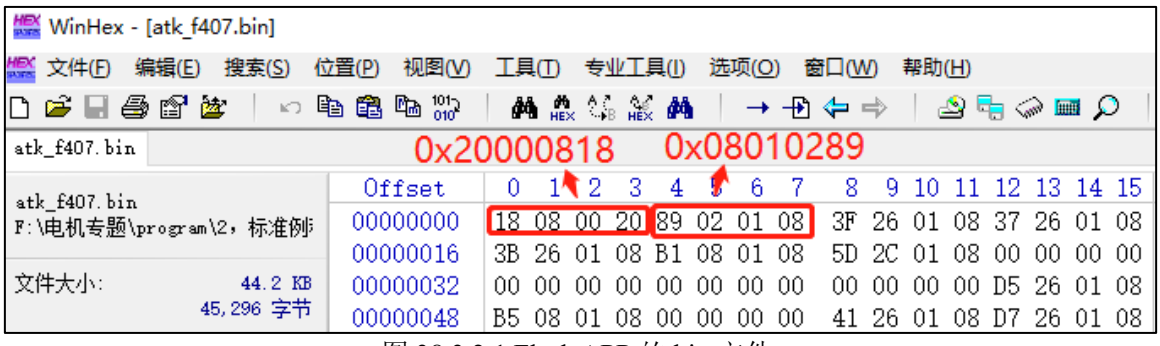


图 38.3.2.1 Flash APP 的 bin 文件

我们利用 stm32 的 bin 文件的特性，按 32 位取的数据，开始的第一个字为 SP 的地址，第

二个为 Reset_Handler 的地址，我们利用这个特性在跳转前做一个初步的判定，然后设置主堆栈，这部分我们用到 sys.c 下的嵌入汇编函数 sys_msr_msp()，我们实现代码如下：

```
/**
 * @brief       跳转到应用程序段(执行 APP)
 * @param       appxaddr : 应用程序的起始地址

 * @retval      无
 */
void iap_load_app(uint32_t appxaddr)
{
    if (((*(volatile uint32_t *)appxaddr) & 0x2FFE0000) == 0x20000000)
    {/* 检查栈顶地址是否合法.可以放在内部 SRAM 共 64KB(0x20000000) */
        /* 用户代码区第二个字为程序开始地址(复位地址) */
        jump2app = (iapfun) * (volatile uint32_t *)(appxaddr + 4);
        /* 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) */
        sys_msr_msp(*(volatile uint32_t *)appxaddr);
        /* 跳转到 APP */
        jump2app();
    }
}
```

2. IAP Bootloader 程序

根据我们流程图的设想，所以我们需要用到 LCD、串口、按键和 STM32 内部 Flash 的操作，所以我们通过复制以前的《FLASH 模拟 EEPROM 实验》来修改，重命名为《串口 IAP 实验》，工程内的组重命名为 IAP。

我们需要修改串口接收部分的程序，为了便于测试，我们定义一个大的接收数组 g_usart_rx_buf[USART_REC_LEN]，并保证这个数组能接收并缓存一个完整的 bin 文件,程序中我们定了了这个大小为 120KB，因为我们有 SRAM 程序（47KB），所以把这部分的数组放用 __attribute__ ((at(0x20001000)))直接放到 SRAM 程序的位置，这样接收完成的 SRAM 程序后我们直接跳转就可以了。

```
uint8_t g_usart_rx_buf[USART_REC_LEN] __attribute__ ((at(0x20001000)));
```

接收的数据处理方法与我们之前的串口处理方式类似，大家查看串口接收处理的源码即可。我们把接收标记的处理放在 main.c 中处理，具体如下：

```
int main(void)
{
    uint8_t t;
    uint8_t key;
    uint32_t lastcount = 0;      /* 上一次串口接收数据值 */
    uint32_t applenth = 0;       /* 接收到的 app 代码长度 */
    uint8_t clearflag = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟, 168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "IAP TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY2: Copy APP2FLASH!", RED);
    lcd_show_string(30, 130, 200, 16, 16, "KEY1: Run FLASH APP", RED);
    lcd_show_string(30, 150, 200, 16, 16, "KEY0: Run SRAM APP", RED);

    while (1)
    {
        if (g_usart_rx_cnt)
```

```
    {
        if (lastcount == g_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 = g_usart_rx_cnt;
            lastcount = 0;
            g_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 lastcount = g_usart_rx_cnt;
    }

    t++;
    delay_ms(100);
    if (t == 3)
    {
        LED0_TOGGLE();
        t = 0;
        if (clearflag)
        {
            clearflag--;
            if (clearflag == 0)
            {
                lcd_fill(30, 190, 240, 210 + 16, WHITE);    /* 清除显示 */
            }
        }
    }
    key = key_scan(0);
    if (key == KEY2_PRES)   /* KEY2 按下,更新固件到 FLASH */
    {
     if (applenth)
     {
            printf("开始更新固件...\r\n");
            lcd_show_string(30, 190, 200, 16, 16, "Copying APP2FLASH...",
                            BLUE);
            if (((*(volatile uint32_t *)(0x20001000 + 4)) & 0xFF000000) ==
                  0x08000000)  /* 判断是否为 0x08XXXXXX */
            {/* 更新 FLASH 代码 */
                iap_write_appbin(FLASH_APP1_ADDR, g_usart_rx_buf, applenth);
                lcd_show_string(30,190,200,16,16,"Copy APP Successed!!", BLUE);
                printf("固件更新完成!\r\n");
            }
            else
            {
                lcd_show_string(30,190,200,16,16,"Illegal FLASH APP!  ", BLUE);
                printf("非 FLASH 应用程序!\r\n");
            }
        }
        else
        {
            printf("没有可以更新的固件!\r\n");
            lcd_show_string(30, 190, 200, 16, 16, "No APP!", BLUE);
        }
        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 按键按下, 运行 FLASH APP 代码 */
    {
      if (((*(volatile uint32_t *)(FLASH_APP1_ADDR + 4)) & 0xFF000000) ==
            0x08000000)       /* 判断 FLASH 里面是否有 APP,有的话执行 */
        {
            printf("开始执行 FLASH 用户代码!!\r\n\r\n");
            delay_ms(10);
            iap_load_app(FLASH_APP1_ADDR);/* 执行 FLASH APP 代码 */
        }
```

```
            else
            {
                printf("没有可以运行的固件!\r\n");
                lcd_show_string(30, 190, 200, 16, 16, "No APP!", BLUE);
            }
            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\r\n");
            delay_ms(10);
            if (((*(volatile uint32_t *)(0x20001000 + 4)) & 0xFF000000) ==
                    0x20000000)                 /* 判断是否为 0x20XXXXXX */
            {
                iap_load_app(0x20001000);       /* SRAM 地址 */
            }
            else
            {
              printf("非 SRAM 应用程序,无法执行!\r\n");
              lcd_show_string(30, 190, 200, 16, 16, "Illegal SRAM APP!", BLUE);
            }
            clearflag = 7;      /* 标志更新了显示,并且设置 7*300ms 后清除显示 */
        }
    }
}
```

APP 代码我们在这里就不做介绍了，大家可以参考本例程提供的源代码，注意在 mian 函数起始处重新设置中断向量表（寄存器 SCB->VTOR）的偏移量，否则 APP 无法正常运行，仍以 Flash APP 为例，我们编译通过后执行了 fromelf.exe 生成 bin 文件，如图 38.3.2.6 所示：



图 38.3.2.6 多存储段 APP 程序生成.bin 文件

38.4 下载验证

将程序下载到开发板后，可以看到 LCD 首先显示一些实验相关的信息，如图 38.4.1 所示：

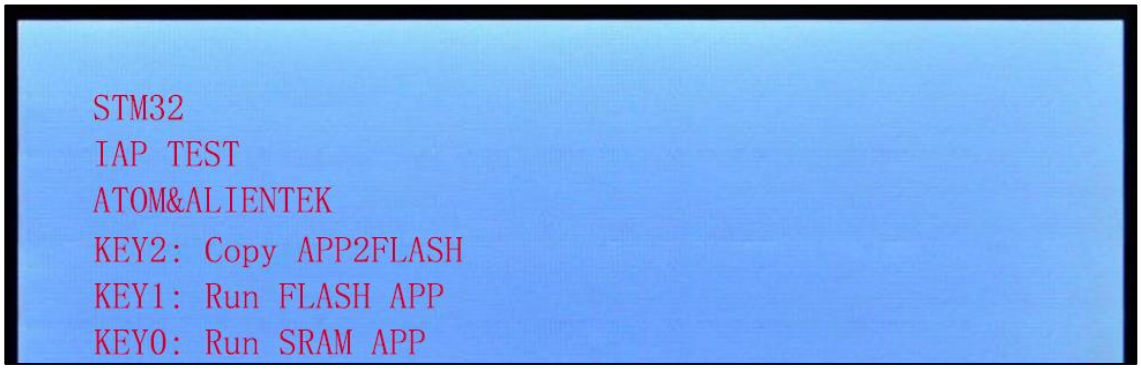


图 38.4.1 IAP 程序界面

此时，我们可以通过 XCOM，发送 FLASH APP、SRAM APP 到开发板，我们以 FLASH APP 为例进行演示，如图 38.4.2 所示：

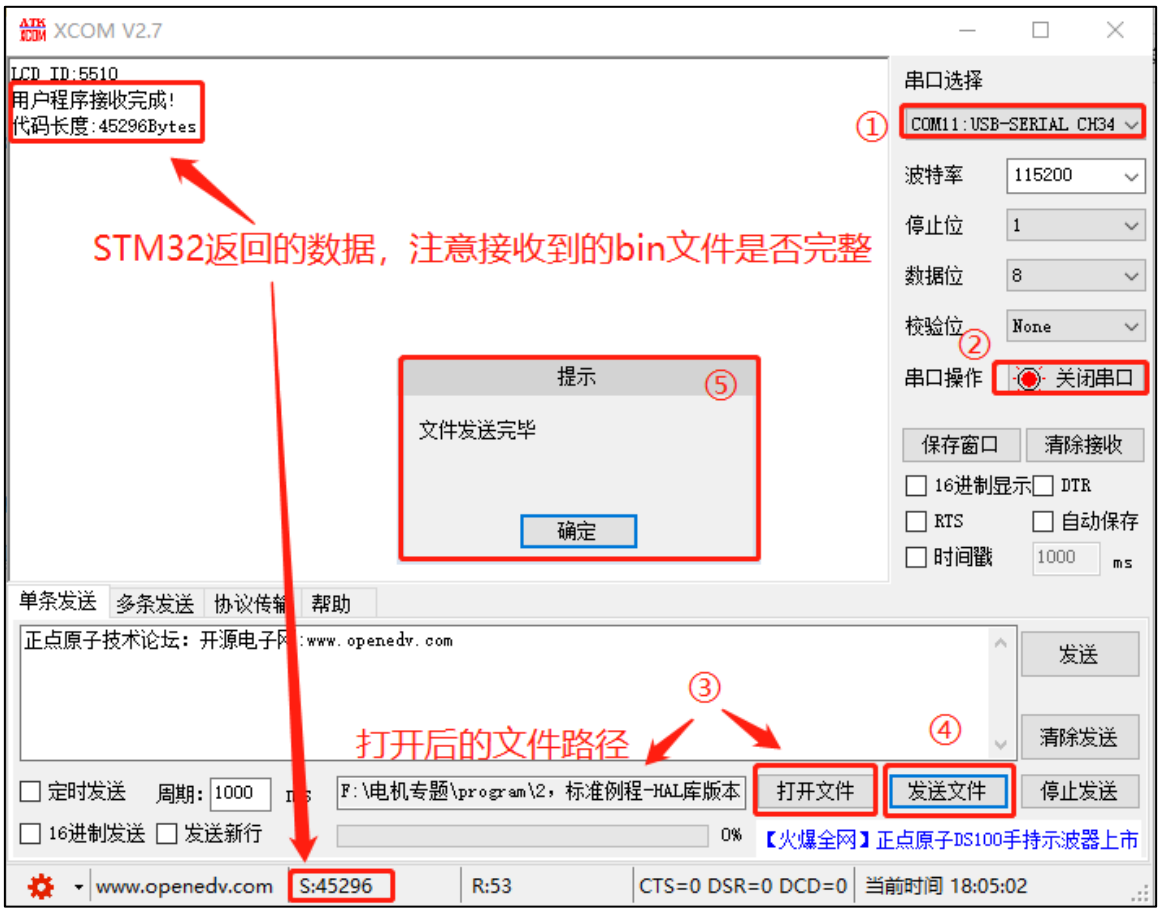


图 38.4.2 串口发送 APP 程序界面

首先找到开发板 USB 转串口的串口号，打开串口（我电脑是 COM11），然后设置波特率为 115200 并打开串口，然后，点击打开文件按钮（图中标号 3 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 4 所示），将.bin 文件发送给 STM32 开发板，发送完成后，XCOM 会提示文件发送完毕（图中标号 5 所示）。

开发板收到 APP 程序之后会打印提示信息，我们可以根据发送的数据与开发板的提示信息确认开发板接收到的 bin 文件是否完整，我们就可以通过 KEY1/KEY0 运行这个 APP 程序了（如果是 FLASH APP，需要通过 KEY2 将其存入对应 FLASH 区域），此时我们根据程序设计，按下 KEY1 即可执行 FLASH APP 程序，更新 SRAM APP 的过程类似，大家自行测试即可。

第三十九章 USB 读卡器(Slave)实验

STM32F407 系列芯片都自带了 USB OTG FS 和 USB OTG HS（HS 需要外扩高速 PHY 芯片实现，速度可达 480Mbps），支持 USB Host 和 USB Device，DMF407 没有外扩高速 PHY 芯片，所以仅支持 USB OTG FS（FS，即全速，12Mbps），所有 USB 相关例程，均使用 USB OTG FS 实现。

下面，我们将介绍如何使用 USB OTG FS 在 DMF407 上实现一个 USB 读卡器。

本章分为如下几个小节：

39.1 USB 简介
39.2 硬件设计
39.3 程序设计
39.4 下载验证

39.1 USB 简介

USB，即通用串行总线(Universal Serial Bus)，包括 USB 协议和 USB 硬件两个方面，支持热插拔功能。现在日常生活的很多方面都离不开 USB 的应用，如充电和数据传输等方面的应用。

USB 经过多次修改，1996 年确定了初始规范版本 USB1.0，目前由非盈利组织 USB-IF(https://www.usb.org)管理。STM32 自带的 USB 符合 USB2.0 规范,故 2.0 版本仍是本文的重点介绍对象。

39.1.1 USB 简介

USB 本身的知识体系非常复杂，本节只能作一点知识点的引入。本书篇幅有限，不可能在这里详细介绍，想更系统地学习 USB 的知识可以参考《圈圈教你玩 USB》、塞普拉斯提供的《USB 101：通用串行总线 2.0 简介》等文献，下面我们一起来看 USB 的简单特性：

- **USB 的硬件接口**

USB 协议有漫长的发展历程，为的不同的场合和硬件功能而发展出不同的接口：Type-A、Type-B、Type-C，Type-C 规范碰巧是跟着 USB3.1 的规范一起发布的。常见的接口类型列出如图 39.1.1 所示。

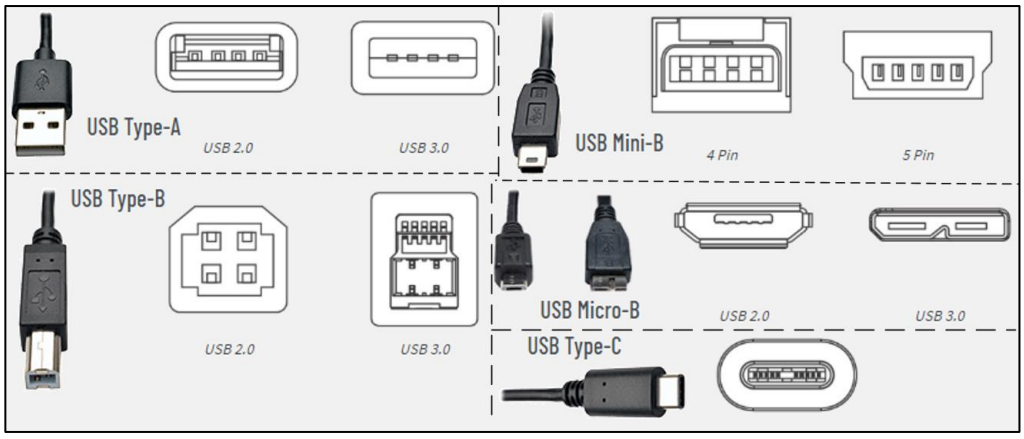


图 39.1.1 常见的 USB 连接器的形状

USB 发展到现在已经有 USB1.0/1.1/2.0/3.x/4 等多个版本。目前用的最多的就是版本 USB1.1 和 USB2.0，USB3.x/USB4 目前也在加速推广。从图中可以发现不同的版本的 USB 接口内的引脚数量是有差异的。USB3.0 以后为了提高速度，采用了更多数量的通讯线，比如同样的是 Type A 接口，USB2.0 版本内部只有四根线，采用半双工式广播式通讯，USB3.0 版本则将通讯线提高到了 9 根，并可以支持全双工非广播式的总线，允许两个单向数据管道分别处理一个

单向通信。

USB2.0 常使用四根线：VCC（5V）、GND、D+(3.3V)和 D-(3.3V) (注：五线模式多了一个 DI 脚用于支持 OTG 模式，OTG 为 USB 主机+USB 设备双重角色)，其中数据线采用差分电压的方式进行数据传输。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 3.3V，而如果是低速设备，则会在 D-上接一个 1.5K 的电阻到 3.3V。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。

关于 USB 硬件还有更多具体的细节规定，硬件设计时需要严格按照 USB 的器件的使用描述和 USB 标准所规定的参数来设计。

- **USB 速度**

USB 规范已经为 USB 系统定义了以下四种速度模式：低速（Low-Speed）、全速（Full-Speed）、高速（Hi-Speed）和超高速（SuperSpeedUSB）。接口的速度上限与设备支持的 USB 协议标准和导线长度、阻抗有关，不同协议版本对硬件的传输线数量、阻抗等要求各不相同，各个版本的能达到的理论速度上限对应如图 39.1.2。

Standard	Also Known As	Logo	Year Introduced	Connector Types	Max. Data Transfer Speed	Cable Length**
USB 1.1	Basic Speed USB		1998	USB-A USB-B	12 Mbps	3 m
USB 2.0	Hi-Speed USB		2000	USB-A USB-B USB Micro A USB Micro B USB Mini A USB Mini B	480 Mbps	5 m
USB 3.2 Gen 1	USB 3.0 USB 3.1 Gen 1 SuperSpeed USB		2008 (USB 3.0) 2013 (USB 3.1)	USB-A USB-B USB Micro B USB-C*	5 Gbps	3 m
USB 3.2 Gen 2	USB 3.1 USB 3.1 Gen 2 SuperSpeed+ SuperSpeed USB 10Gbps		2013 (USB 3.1)	USB-A USB-B USB Micro B USB-C*	10 Gbps	3 m
USB 3.2 Gen 2x2	USB 3.2 SuperSpeed USB 20Gbps		2017 (USB 3.2)	USB-C*	20 Gbps	3 m
Thunderbolt™ 2			2013	Mini DisplayPort	20 Gbps	3 m
Thunderbolt™ 3			2015	USB-C*	20 Gbps (Passive Cable)	2 m
					40 Gbps (Passive Cable)	0.5 m
					40 Gbps (Active Cable)	2 m
USB 4			2019	USB-C*	Up to 40 Gbps	0.8 m
Thunderbolt™ 4			2020	USB-C*	40 Gbps	2 m

图 39.1.2 USB 协议发展与版本对应的速度

USB 端口和连接器有时会标上颜色，以指示 USB 规格及其支持的功能。这些颜色不是 USB 规范所要求的，并且在设备制造商之间不一致。例如，常见的支持 USB3.0 的 U 盘和电脑等设备使用蓝色指示，英特尔使用橙色指示充电端口等。

- **USB 系统**

USB 系统主要包括三个部分：控制器(Host Controller)、集线器(Hub)和 USB 设备。

控制器(Host Controller)，主机一般可以有一个或多个控制器，主要负责执行由控制器驱动程序发出的命令。控制器驱动程序(Host Controller Driver)在控制器与 USB 设备之间建立通信信道。

集线器(Hub)连接到 USB 主机的根集线器，可用于拓展主机可访问的 USB 设备的数量。

USB 设备(USB Device)则是我们常用的如 U 盘，USB 鼠标这类受主机控制的设备。

- **USB 通讯**

USB 针对主机、集线器和设备制定了严格的协议。概括来讲，通过检测、令牌、传输控制、数据传输等多种方式，定义了主机和从机在系统中的不同职能。USB 系统通过“管道”进行通讯，有“控制管道”和“数据管道”两种，“控制管道”是双向的，而每个“数据管道”则是单向的，这种关系如图 39.1.3 所示。

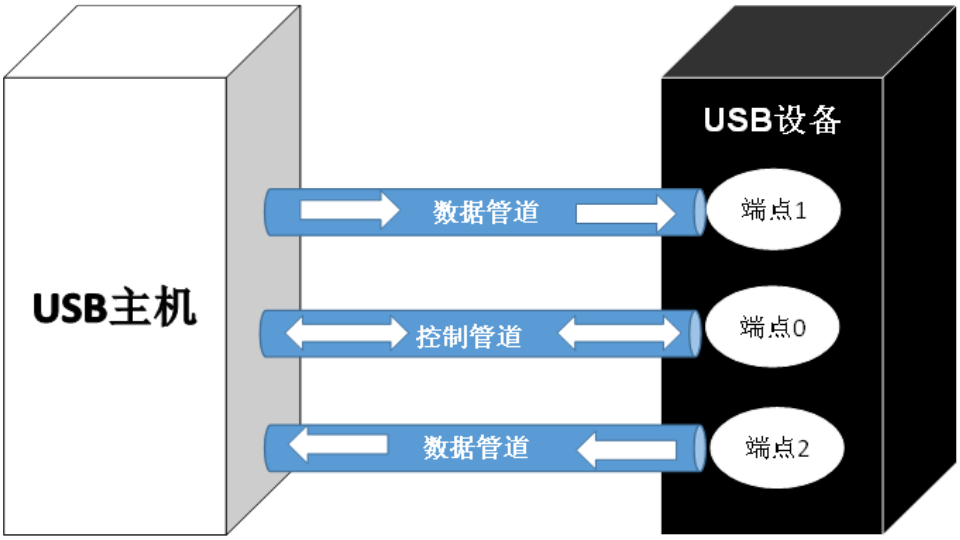


图 39.1.3 USB 管道模型

USB 通讯中的检测和断开总是由主机发起。USB 主机与设备首次进行连接时会交换信息，这一过程叫“USB 枚举”。枚举是设备和主机间进行的信息交换过程，包含用于识别设备的信息。此外，枚举过程主机需要分配设备地址、读取描述符（作为提供有关设备信息的数据结构），并分配和加载设备驱动程序，而从机需要提供相应的描述符使主机知悉如何操作此设备。整个过程需要数秒时间。完成该过程后设备才可以向主机传输数据。数据传输也有规定的三种类型，分别是：IN/读取/上行数据传输、OUT/写入/下行数据传输、控制数据传输。

USB 通过设备端点寻址，在主机和设备间实现信息交流。枚举发生前有一套专用的端点用于与设备进行通信。这些专用的端点统称为控制端点或端点 0，有端点 0 IN 和端点 0 OUT 两个不同的端点，但对开发者来说，它们的构建和运行方式是一样的。每一个 USB 设备都需要支持端点 0。因此，端点 0 不需要使用独立的描述符。除了端点 0 外，特定设备所支持的端点数量将由各自的设计要求决定。简单的设计（如鼠标）可能仅要一个 IN 端点。复杂的设计可能需要多个数据端点。

USB 规定的数据 4 种数据传输方式也是通过管道进行，分别是控制传输(Control Transfer)、中断传输(Interrupt Transfer)、批量传输或叫块传输(Bulk Transfer)、实时传输或叫同步传输(Isochronous Transfer)，每种模式规定了各自通讯时使用的管道类型。

关于 USB 还有很多更详细的时序和要求，像 USB 描述符、VID/PID 的规定、USB 类设备和调试等，因为 USB2.0 和之后的版本有差异，这里就不再为大家列举了，ST 对 USB2.0 也有专门的培训资料，这部分我们也放到“光盘资料 A 盘→8，STM32 参考资料→2，STM32 USB 学习资料”中了，感兴趣的朋友自行去查阅更的 USB 的相关扩展知识，我们对 USB 的简介就到这里。

39.1.2 STM32F4 的 USB 特性

USB 发展到现在已经有 USB1.0/1.1/2.0/3.0 等多个版本。目前用的最多的就是 USB1.1 和 USB2.0，USB3.0 目前已经开始普及。STM32F407 自带的 USB 符合 USB2.0 规范。

标准 USB 共四根线组成，除 VCC/GND 外，另外为 D+和 D-，这两根数据线采用的是差分电压的方式进行数据传输的。在 USB 主机上，D-和 D+都是接了 15K 的电阻到地的，所以在没有设备接入的时候，D+、D-均是低电平。而在 USB 设备中，如果是高速设备，则会在 D+上接一个 1.5K 的电阻到 VCC，而如果是低速设备，则会在 D-上接一个 1.5K 的电阻到 VCC。这样

当设备接入主机的时候，主机就可以判断是否有设备接入，并能判断设备是高速设备还是低速设备。接下来，我们简单介绍一下 STM32 的 USB 控制器。

STM32F407 系列芯片自带有 2 个 USB OTG，高速 USB（USB OTG HS）；全速 USB（USB OTG FS），高速 USB（HS）需要外扩高速 PHY 芯片实现，我们这里不做介绍。

STM32F407 的 USB OTG FS 是一款双角色设备 (DRD) 控制器，同时支持从机功能和主机功能，完全符合 USB 2.0 规范的 On-The-Go 补充标准。此外，该控制器也可配置为“仅主机”模式或“仅从机” 模式，完全符合 USB 2.0 规范。在主机模式下，OTG FS 支持全速（FS，12 Mb/s）和低速（LS，1.5 Mb/s）收发器，而从机模式下则仅支持全速（FS，12 Mb/s）收发器。OTG FS 同时支持 HNP 和 SRP。

STM32F407 的 USB OTG FS 主要特性可分为三类：通用特性、主机模式特性和从机模式特性。

1、通用特性

- 经 USB-IF 认证，符合通用串行总线规范第 2.0 版
- 集成全速 PHY，且完全支持定义在标准规范 OTG 补充第 1.3 版中的 OTG 协议
  1，支持 A-B 器件识别（ID 线）
  2，支持主机协商协议(HNP)和会话请求协议(SRP)
  3，允许主机关闭 VBUS 以在 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
  4，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 1.25KB 专用 RAM
  1，可将 RAM 空间划分为不同 FIFO，以便灵活有效地使用 RAM
  2，每个 FIFO 可存储多个数据包
  3，动态分配存储区
  4，FIFO 大小可配置为非 2 的幂次方值，以便连续使用存储单元
- 一帧之内可以无需要应用程序干预，以达到最大 USB 带宽

2、主机（Host）模式特性

- 通过外部电荷泵生成 VBUS 电压。
- 多达 8 个主机通道（管道）：每个通道都可以动态实现重新配置，可支持任何类型的 USB 传输。
- 内置硬件调度器：
  1，在周期性硬件队列中存储多达 16 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 16 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
- 8 个 IN 端点 (EP)，可配置为支持批量传输、中断传输或同步传输
- 8 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 9 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。

STM32F407 USB OTG FS 框图如图 39.1.1 所示：

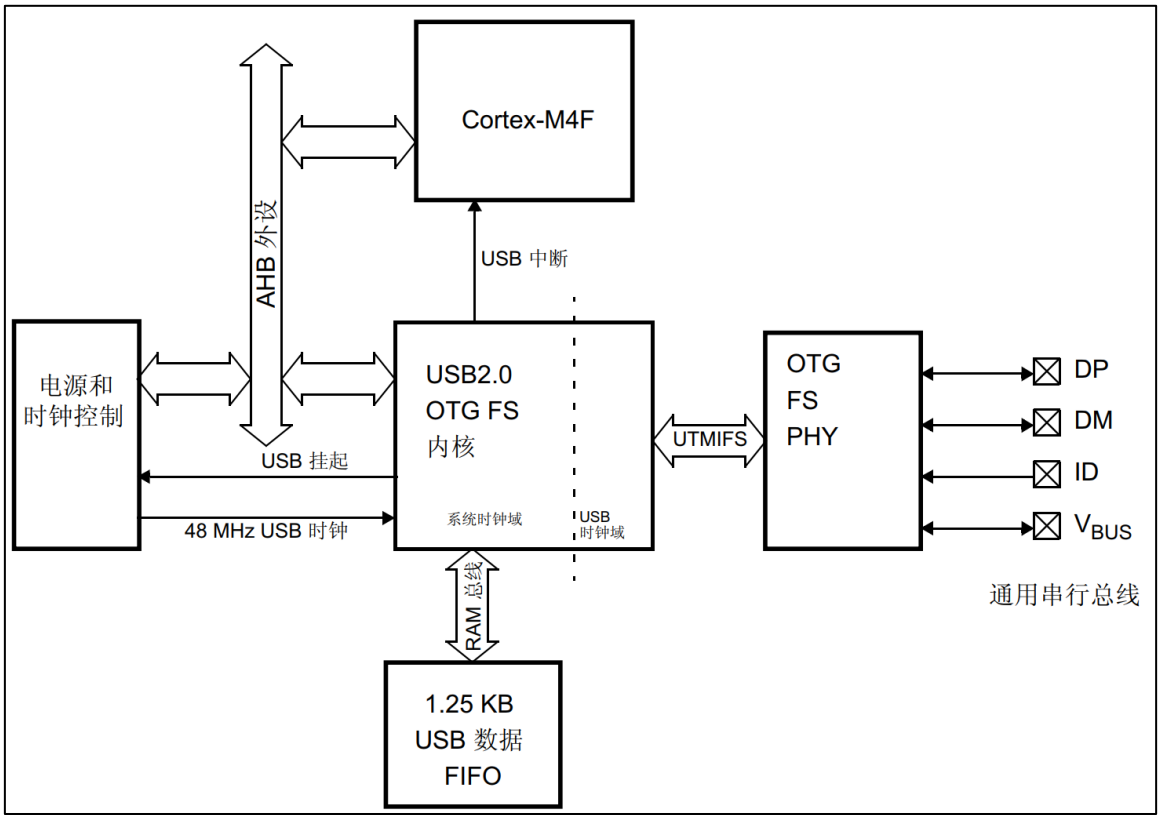


图 39.1.1 USB OTG FS 框图

对于 USB OTG FS 功能模块，STM32F407 通过 AHB 总线访问（AHB 频率必须大于 14.2Mhz），另外，USB OTG 的内核时钟必须是 48Mhz，是来自时钟树里面的 PLL48CK（和 SDIO 共用）。

要正常使用 STM32F407 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。

STM32F4 系列的 USB 例程全部是以 HAL 库的形式提供，为了简化开发设计，我们直接使用 ST 提供的 HAL 库版本 USB 驱动库来设计相关例程。

ST 提供的 USB OTG 库和相关参考例程在 en.stm32cubef4_v1-26-0_v1.26.0.zip 里面可以找到，该文件可以在 www.st.com 网站搜索：cubef4 找到。不过，我们已经帮大家下载到开发板光盘：8，STM32 参考资料→ 1，STM32CubeF4 固件包→ en.stm32cubef4_v1-26-0_v1.26.0.zip。解压可以得到 STM32F4 的 Cube 固件支持包：STM32Cube_FW_F4_V1.26.0，该文件夹里面包含了 STM32F4 的 USB 主机（Host）和从机（Device）驱动库，如图 39.1.2 所示，并提供了 17 个例程供我们参考，如图 39.1.3 所示：

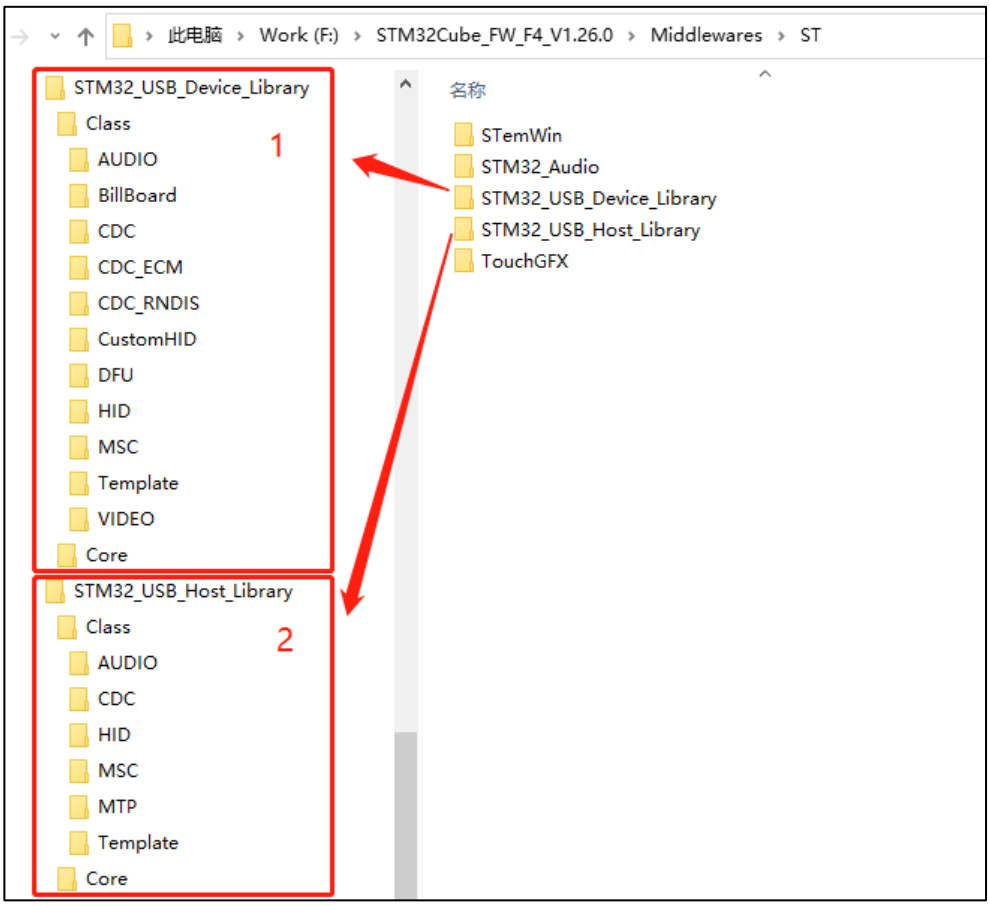


图 39.1.2 ST 提供的 USB OTG 库

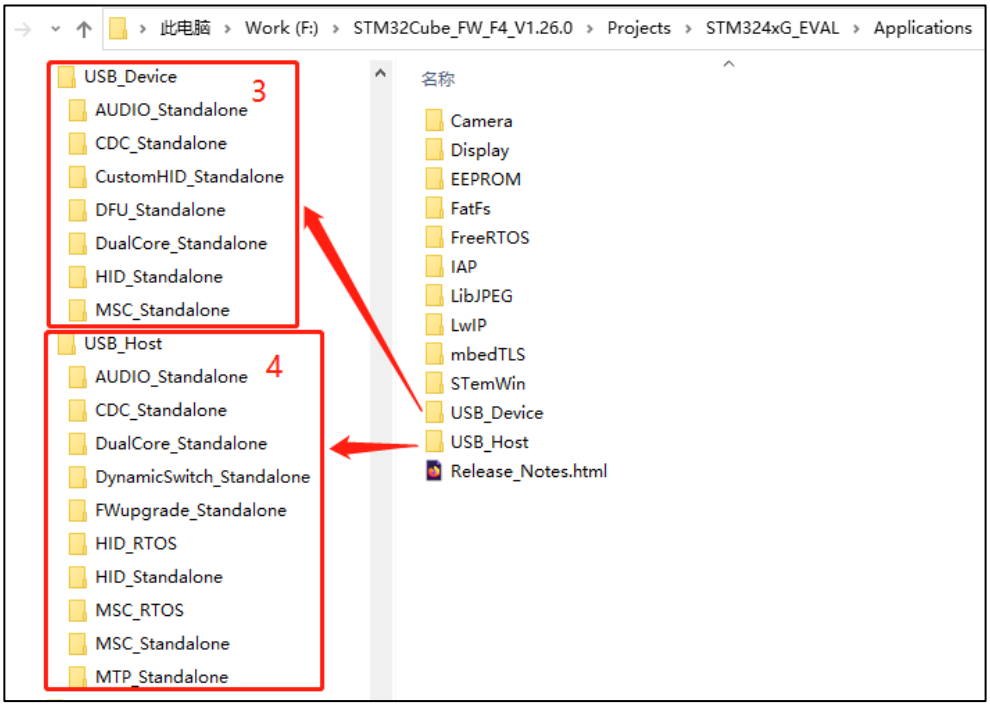


图 39.1.3 ST 提供的 USB OTG 例程

标号 1 是 STM32F4xG USB 从机驱动库；
标号 2 是 STM32F4xG USB 主机驱动库；
标号 3 是 STM32F4xG USB 从机例程（共 7 个）；
标号 4 是 STM32F4xG USB 主机例程（共 10 个）；

整个 USB OTG 库的使用和例程说明，可以参考 ST 官方提供的：UM1734（从机）和 UM1720（主机）这两个文档（在光盘：8，STM32 参考资料→2，STM32 USB 学习资料 文件夹），这两个文档详细介绍了 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 17 个例程，虽然都是基于官方 STM324xG_EVAL 板，但是很容易移植到我们的电机开发板上。

本实验我们移植：STM32Cube_FW_F4_V1.26.0\Projects\STM324xG_EVAL\Applications\USB_Device\MSC_Standalone 这个例程，实现 USB 读卡器功能。

39.2 硬件设计

1. 例程功能

本实验代码,开机的时候先检测 SPI FLASH 是否存在，如果存在则获取其容量，并显示在 LCD 上面（如果不存在，则报错）。之后开始 USB 配置，在配置成功之后就可以在电脑上发现一个可移动磁盘。USB 正在读写会在液晶上显示出来。

LED0 闪烁，提示程序运行。USB 读写 SPI FLASH 时，LED1 闪烁。

2. 硬件资源

1）LED 灯
LED0 – PE0
LED1 – PE1
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）USB_SLAVE 接口(D-/D+连接在 PA11/PA12 上)
5）SPI2(PI0/PI1/PI2/PI3)
6）norflash(本例程使用的是 W25Q128，连接在 SPI2)

3. 原理图

开发板采用的是 16PIN 的 Type-C 接口，用来和电脑的 USB 相连接，Type-C 接口与 STM32 的连接电路图，如下图所示：

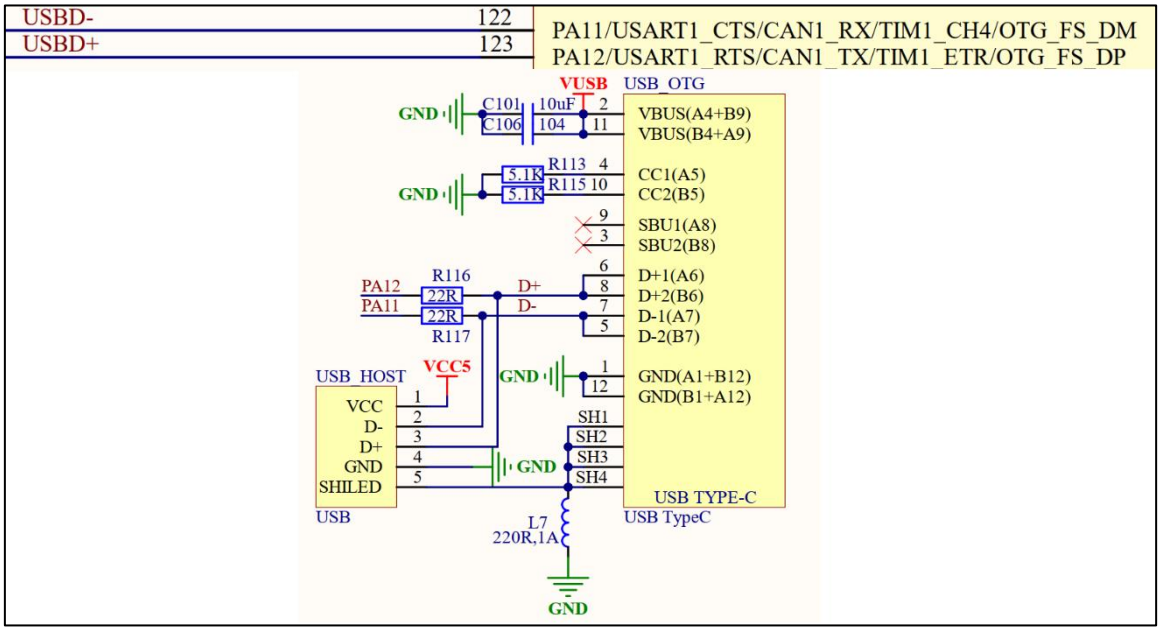


图 39.2.1 Type-C 接口与 STM32 的连接电路图

从上图可以看出，USB Type-C 座是直接连接到 STM32F407 上面的，所以硬件上不需要做

什么操作，可直接使用。

需要注意的是：这个 Type-C 座和 USB 母座（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，Type-C 数据线是通过下图框里的 USB_OTG 接口连接到电脑的，不要连错。

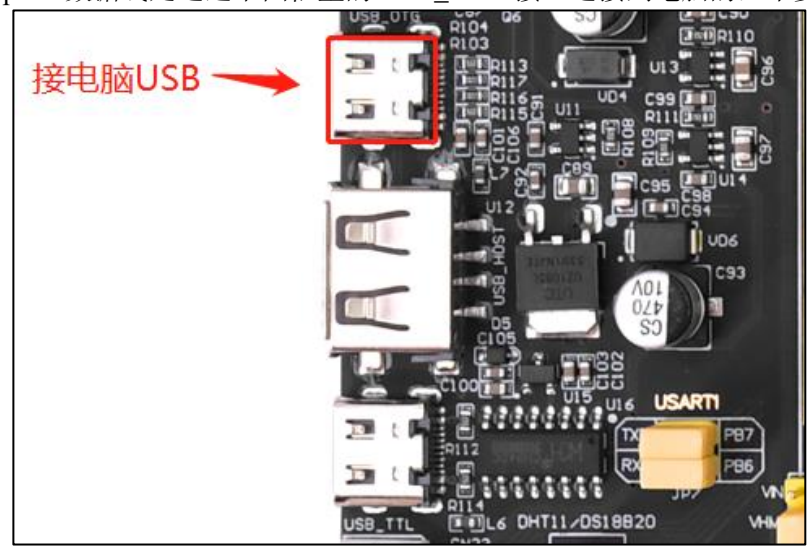


图 39.2.2 Type-C 接口与电脑连接示意图

39.3 程序设计

39.3.1 程序流程图

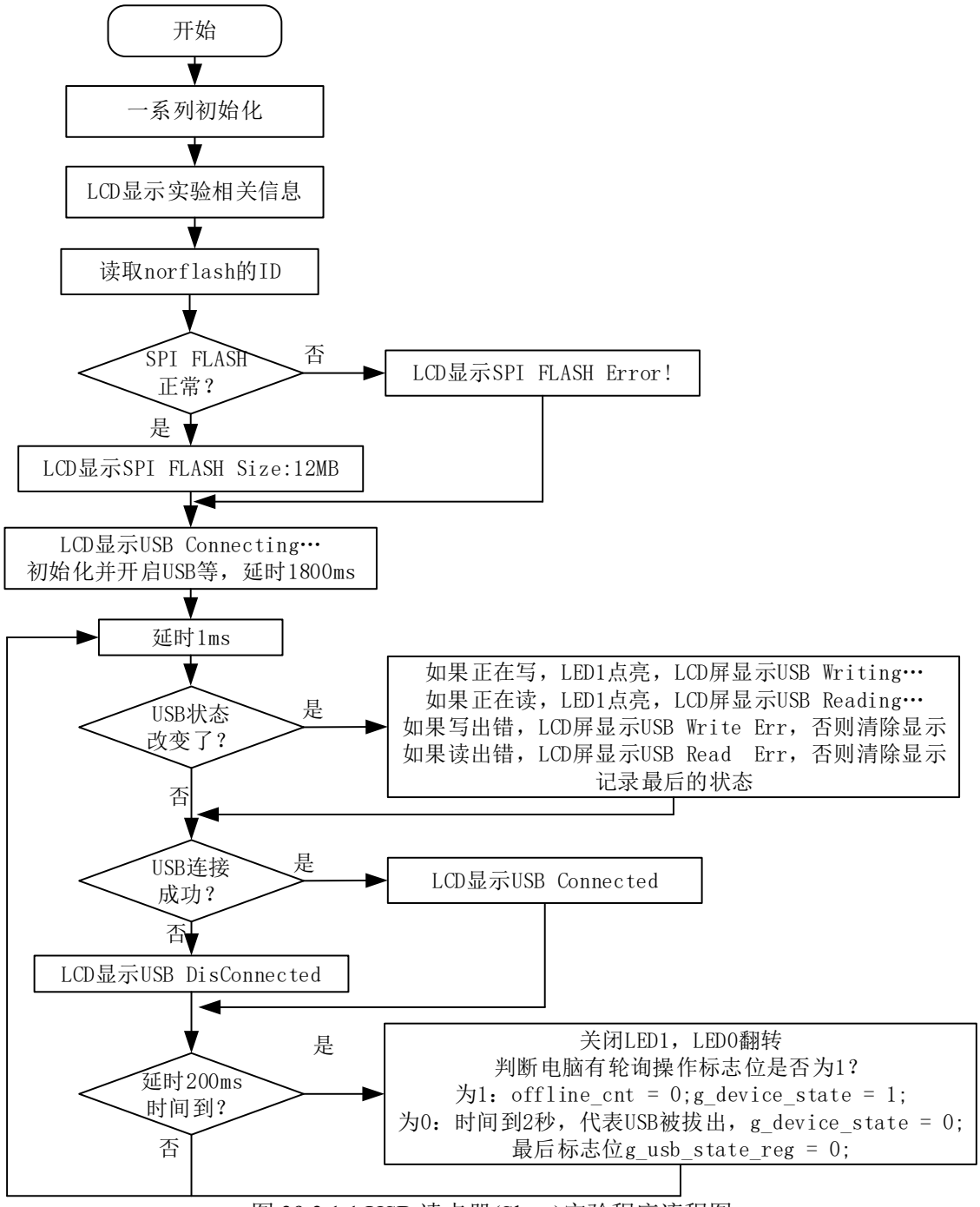


图 39.3.1.1 USB 读卡器(Slave)实验程序流程图

39.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。

本实验，我们在：实验 15 SPI 实验的基础上修改，USB 的代码从 ST 官方例程移植：STM32Cube_FW_F4_V1.26.0\Projects\STM324xG_EVAL\Applications\USB_Device\MSC_Standalone。该目录下提供了三种开发环境的工程：IAR、MDK 和 SW4STM32，我们使用的是 MDK，打开 MDK 工程，然后就可以知道和 USB 相关的代码有哪些，如图 39.3.2.1 所示：

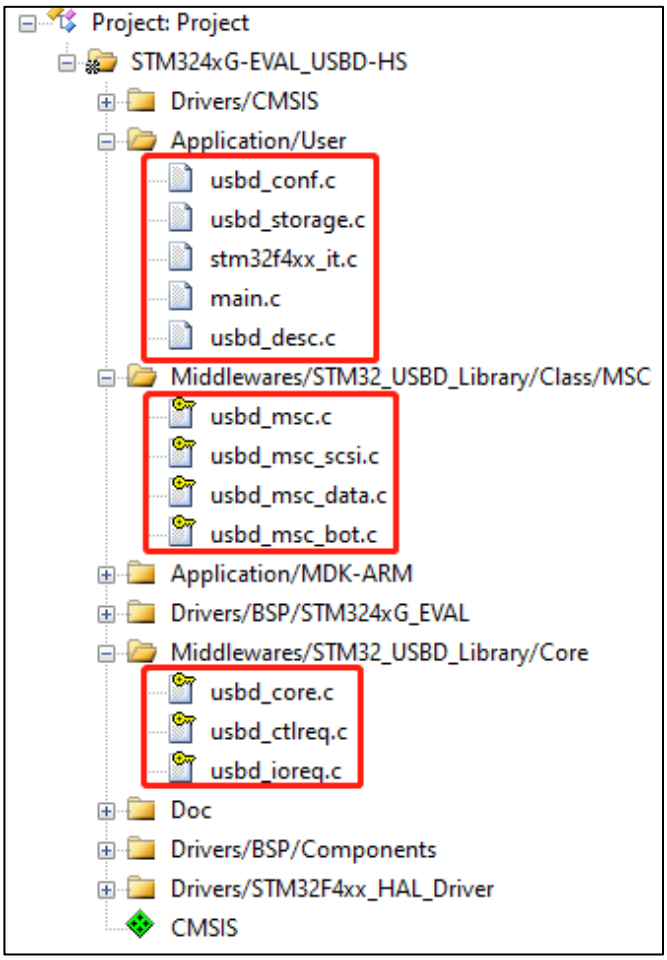


图 39.3.2.1 ST 官方例程 USB 相关代码

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，在工程的 Middlewares 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→1，STM32CubeF4 固件包→STM32Cube_FW_F4_V1.26.0→Middlewares→ST 文件夹下的： STM32_USB_Device_Library、STM32_USB_HOST_Library 两个文件夹及源码拷贝到该文件夹下面。

然后，在 USB 文件夹下，新建一个 USB_APP 文件夹用于存放 MSC 实现相关代码，即：STM32Cube_FW_F4_V1.26.0→Projects→STM324xG_EVAL→Applications→USB_Device→MSC_Standalone→Src 下的部分代码：usbd_conf.c、usbd_storage.c 和 usbd_desc.c 等 3 个.c 文件，同时拷贝：STM32Cube_FW_F4_V1.26.0 → Projects → STM324xG_EVAL → Applications → USB_Device→MSC_Standalone→Inc 下面的：usbd_conf.h、usbd_storage.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 39.3.2.2 所示：

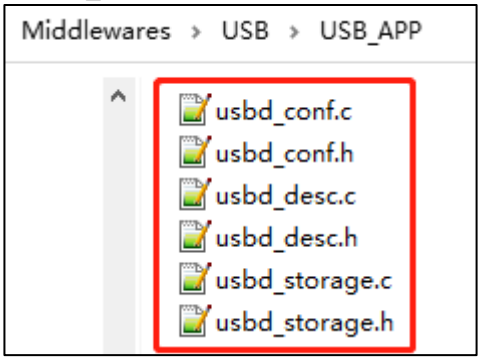


图 39.3.2.2 USB_APP 代码

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

节请参考例程，这里就不详细介绍了，添加好之后，如图 39.3.2.3 所示：

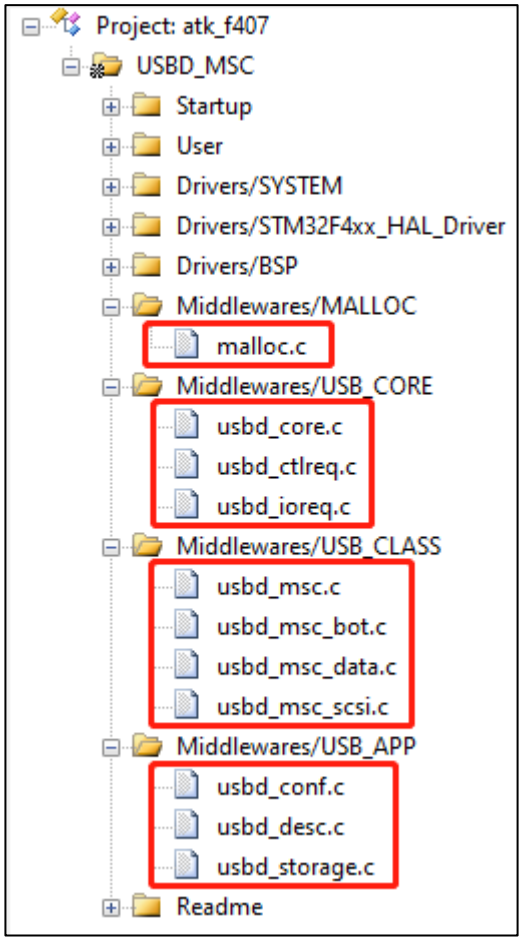


图 39.3.2.3 添加 USB 驱动、malloc 等相关代码

1. USB 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。接下来我们看看 USB_APP 里面的几个.c 文件：

usbd_conf.c 提供了 USB 设备库（从机库，下同）的回调及 MSP 初始化函数，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_conf.c 我们重点介绍 3 个函数，首先是 HAL_PCD_MspInit 和 OTG_FS_IRQHandler 函数，它们的定义如下：

```
/**
 * @brief       初始化 PCD MSP
 * @param       hpcd : PCD 结构体指针
 * @retval      无
 */
void HAL_PCD_MspInit(PCD_HandleTypeDef *hpcd)
{
    GPIO_InitTypeDef gpio_init_struct = {0};

    if (hpcd->Instance == USB_OTG_FS)
    {

        __HAL_RCC_GPIOA_CLK_ENABLE();                           /* 使能 GPIOA 时钟 */
        gpio_init_struct.Pin = GPIO_PIN_11 | GPIO_PIN_12;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用 */
        gpio_init_struct.Pull = GPIO_NOPULL;                    /* 浮空 */
```

```
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;      /* 高速 */
        gpio_init_struct.Alternate = GPIO_AF10_OTG_FS;           /* 复用为 OTG_FS */
        HAL_GPIO_Init(GPIOA, &gpio_init_struct);    /* 初始化 PA11 和 PA12 引脚 */

        __HAL_RCC_USB_OTG_FS_CLK_ENABLE();          /* 使能 OTG FS 时钟 */

        HAL_NVIC_SetPriority(OTG_FS_IRQn, 1, 0);   /* 优先级设置为抢占 1,子优先级 0 */
        HAL_NVIC_EnableIRQ(OTG_FS_IRQn);            /* 使能 OTG FS 中断 */
    }
    else if (hpcd->Instance == USB_OTG_HS)
    {
        /* USB OTG HS 本例程没用到,故不做处理 */
    }
}

/**
 * @brief       USB OTG 中断服务函数
 *   @note      处理所有 USB 中断
 * @param       无
 * @retval      无
 */
void OTG_FS_IRQHandler(void)
{
    HAL_PCD_IRQHandler(&g_pcd_usb_otg_fs);
}
```

HAL_PCD_MspInit 函数，用于使能 USB 时钟，初始化 IO 口，设置中断等。该函数在 HAL_PCD_Init 函数里面被调用。

OTG_FS_IRQHandler 函数，是 USB 的中断服务函数，通过调用 HAL_PCD_IRQHandler 函数，实现对 USB 各种事务的处理。

下面介绍的 USBD 底层初始化函数，其定义如下：

```
/**
 * @brief       USBD 底层初始化函数
 * @param       pdev    : USBD 句柄指针
 * @retval      USB 状态
 *   @arg       USBD_OK(0)   , 正常;
 *   @arg       USBD_BUSY(1) , 忙;
 *   @arg       USBD_FAIL(2) , 失败;
 */
USBD_StatusTypeDef USBD_LL_Init(USBD_HandleTypeDef *pdev)
{
    if (pdev->id == DEVICE_FS)
    {
        g_pcd_usb_otg_fs.pData = pdev;      /* g_pcd_usb_otg_fs 的 pData 指向 pdev */
        pdev->pData = &g_pcd_usb_otg_fs;    /* pdev 的 pData 指向 g_pcd_usb_otg_fs */

        g_pcd_usb_otg_fs.Instance = USB_OTG_FS;          /* 使用 USB OTG FS*/
        g_pcd_usb_otg_fs.Init.dev_endpoints = 4;         /* 端点数为 4 */
        g_pcd_usb_otg_fs.Init.speed = PCD_SPEED_FULL;    /* USB 全速(12Mbps) */
        g_pcd_usb_otg_fs.Init.dma_enable = DISABLE;      /* 不使能 DMA */
        g_pcd_usb_otg_fs.Init.phy_itface = PCD_PHY_EMBEDDED;  /* 使用内部 PHY */
        g_pcd_usb_otg_fs.Init.Sof_enable = DISABLE;           /* 不使能 SOF 中断 */
        g_pcd_usb_otg_fs.Init.low_power_enable = DISABLE;     /* 不使能低功耗模式 */
        g_pcd_usb_otg_fs.Init.lpm_enable = DISABLE;           /* 不使能电源管理 */
        g_pcd_usb_otg_fs.Init.vbus_sensing_enable = DISABLE;  /* 不使能 VBUS 检测 */
        /* 禁止 EP1 dedicated 中断 */
        g_pcd_usb_otg_fs.Init.use_dedicated_ep1 = DISABLE;

        HAL_PCD_Init(&g_pcd_usb_otg_fs);
```

```
        /* 设置接收 FIFO 大小为 0x80(128 字节) */
        HAL_PCDEx_SetRxFiFo(&g_pcd_usb_otg_fs, 0x80);
        /* 设置发送 FIFO 0 的大小为 0x40(64 字节) */
        HAL_PCDEx_SetTxFiFo(&g_pcd_usb_otg_fs, 0, 0x40);
        /* 设置发送 FIFO 1 的大小为 0x80(128 字节) */
        HAL_PCDEx_SetTxFiFo(&g_pcd_usb_otg_fs, 1, 0x80);
    }

    return USBD_OK;
}
```

USBD_LL_Init 函数，用于初始化 USB 底层设置，因为我们定义的是：USE_USB_FS，因此会设置 USB OTG 使用 USB_OTG_FS，然后完成各种设置，比如，使用内部 PHY，使用全速模式，不使能 VBUS 检测等，详见以上代码。该函数在 USBD_Init 函数里面被调用。

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、端点、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就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_storage.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 1 个磁盘：SPI FLASH。usb_storage.c 我们重点介绍 3 个函数，首先是初始化存储设备函数，其定义如下：

```
/**
 * @brief       初始化存储设备
 * @param       lun       : 逻辑单元编号
 * @arg         0, SPI FLASH
 * @retval      操作结果
 * @arg         0    , 成功
 * @arg         其他 , 错误代码
 */
int8_t STORAGE_Init_FS(uint8_t lun)
{
    uint8_t res;
    switch(lun)
    {
        case 0:     /* SPI FLASH */
            norflash_init();
            res = USBD_OK;
            break;
        default :
            res = USBD_FAIL;
    }

    return res;
}
```

STORAGE_Init 函数，用于初始化存储设备，我们这里只需要初始化 NOR FLASH。

下面要介绍的是从存储设备读取数据函数。其定义如下：

```
/**
 * @brief       从存储设备读取数据
 * @param       lun       : 逻辑单元编号
 *   @arg                  0, SPI FLASH
 * @param       buf       : 数据存储区首地址指针
 * @param       blk_addr  : 要读取的地址(扇区地址)
 * @param       blk_len   : 要读取的块数(扇区数)
 * @retval      操作结果
 *   @arg       0    , 成功
 *   @arg       其他 , 错误代码
 */
int8_t STORAGE_Read_FS(uint8_t lun, uint8_t *buf, uint32_t blk_addr,
                       uint16_t blk_len)
{
```

```
    uint8_t res;

    g_usb_state_reg |= 0x02;    /* 标记正在读数据 */

    switch(lun)
    {
        case 0:     /* SPI FLASH */
            norflash_read(buf, USB_STORAGE_FLASH_BASE + blk_addr * 512,
                          blk_len * 512);
            res = USBD_OK;
            break;
        default :
            res = USBD_FAIL;
    }

    return res;
}
```

STORAGE_Read_FS 函数，用于从存储设备读取数据，同样是根据存储设备（lun）的不同，调用不同的读取函数，完成数据读取。

下面要介绍的是向存储设备写数据函数。其定义如下：

```
/**
 * @brief       向存储设备写数据
 * @param       lun       : 逻辑单元编号
 *   @arg                   0, SPI FLASH
 * @param       buf       : 数据存储区首地址指针
 * @param       blk_addr  : 要写入的地址(扇区地址)
 * @param       blk_len   : 要写入的块数(扇区数)
 * @retval      操作结果
 *   @arg       0    , 成功
 *   @arg       其他 , 错误代码
 */
int8_t STORAGE_Write_FS(uint8_t lun, uint8_t *buf, uint32_t blk_addr,
                        uint16_t blk_len)
{
    uint8_t res;

    g_usb_state_reg |= 0x01;    /* 标记正在写数据 */

    switch(lun)
    {
        case 0:     /* SPI FLASH */
            norflash_write(buf, USB_STORAGE_FLASH_BASE + blk_addr * 512,
                           blk_len * 512);
            res = USBD_OK;
            break;
        default :
            res = USBD_FAIL;
    }

    return res;
}
```

STORAGE_Write_FS 函数，用于往存储设备写入数据，也是根据存储设备（lun）的不同，调用不同的写入函数（F407 电机开发板只有 NOR FLASH），完成数据写入。

以上 3 个.c 文件和对应.h 文件的详细代码和修改方法，我们就不详细介绍了，请大家参考光盘本例程源码。

下面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，添加宏定义：USE_USB_FS，具体添加情况如图 39.3.2.4 所示：

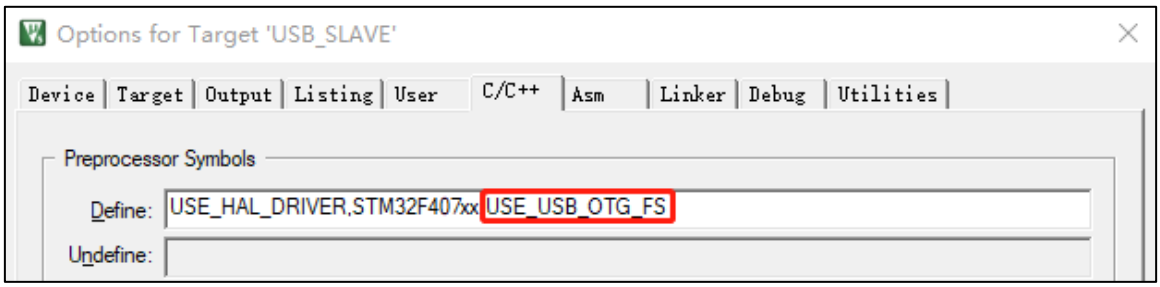


图 39.3.2.4 定义全局宏 USE_USB_FS

2、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。另外，我们通过修改：STORAGE_LUN_NBR 宏定义的值为 1，可以支持 2 个磁盘（电机开发板只支持 1 个磁盘）。

3、官方例程在 2 个或以上磁盘支持的时候，存在 bug，我们需要修改 usbd_msc.h 里面 USBD_MSC_BOT_HandleTypeDef 结构体的 scsi_blk_nbr 参数，将其改为数组形式：uint32_t scsi_blk_nbr[STORAGE_LUN_NBR];数组大小由 STORAGE_LUN_NBR 指定，本实验我们定义的是 1，因此可以支持最多 2 个磁盘，修改 STORAGE_LUN_NBR 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。

以上 3 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：UM1734(STM32Cube USB device library).pdf 这个文档）。

2. main.c 代码

下面是 main.c 的程序，具体如下：

```
USBD_HandleTypeDef USBD_Device;             /* USB Device 处理结构体 */
extern volatile uint8_t g_usb_state_reg;    /* USB 状态 */
extern volatile uint8_t g_device_state;     /* USB 连接情况 */

int main(void)
{
    uint8_t offline_cnt = 0;
    uint8_t tct = 0;
    uint8_t usb_sta;
    uint8_t device_sta;
    uint16_t id;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    norflash_init();                        /* 初始化 norflash */
    my_mem_init(SRAMIN);                    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM);                   /* 初始化内部 SRAMCCM 内存池 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USB Card Reader TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    id = norflash_read_id();

    if ((id == 0) || (id == 0xFFFF))
    {
        /* 检测 NorFlash 错误 */
        lcd_show_string(30, 130, 200, 16, 16, "NorFlash Error!", RED);
    }
    else   /* SPI FLASH 正常 */
    {
```

```
    lcd_show_string(30, 150, 200, 16, 16, "SPI FLASH Size:12MB", RED);
}
/* 提示正在建立连接 */
lcd_show_string(30, 190, 200, 16, 16, "USB Connecting...", RED);

USBD_Init(&USBD_Device, &FS_Desc, DEVICE_FS);   /* 初始化 USB */
USBD_RegisterClass(&USBD_Device, &USBD_MSC);    /* 添加类 */
/* 为 MSC 类添加回调函数 */
USBD_MSC_RegisterStorage(&USBD_Device, &USBD_Storage_Interface_fops_FS);
USBD_Start(&USBD_Device);    /* 开启 USB */
delay_ms(1800);

while (1)
{
    delay_ms(1);

    if (usb_sta != g_usb_state_reg) /* 状态改变了 */
    {
        lcd_fill(30, 210, 240, 210 + 16, WHITE);    /* 清除显示 */

        if (g_usb_state_reg & 0x01)  /* 正在写 */
        {
            LED1(0);
            /* 提示 USB 正在写入数据 */
            lcd_show_string(30, 210, 200, 16, 16, "USB Writing...", RED);
        }

        if (g_usb_state_reg & 0x02)  /* 正在读 */
        {
            LED1(0);
            /* 提示 USB 正在读出数据 */
            lcd_show_string(30, 210, 200, 16, 16, "USB Reading...", RED);
        }

        if (g_usb_state_reg & 0x04)
        {
            /* 提示写入错误 */
            lcd_show_string(30, 230, 200, 16, 16, "USB Write Err ", RED);
        }
        else
        {
            lcd_fill(30, 230, 240, 230 + 16, WHITE); /* 清除显示 */
        }

        if (g_usb_state_reg & 0x08)
        {
            /* 提示读出错误 */
            lcd_show_string(30, 250, 200, 16, 16, "USB Read  Err ", RED);
        }
        else
        {
            lcd_fill(30, 250, 240, 250 + 16, WHITE); /* 清除显示 */
        }

        usb_sta = g_usb_state_reg;                   /* 记录最后的状态 */
    }

    if (device_sta != g_device_state)
    {
        if (g_device_state == 1)
        {
            /* 提示 USB 连接已经建立 */
            lcd_show_string(30, 190, 200, 16, 16, "USB Connected    ", RED);
        }
```

```
            else
            {
                /* 提示 USB 被拔出了 */
                lcd_show_string(30, 190, 200, 16, 16, "USB DisConnected ", RED);
            }

            device_sta = g_device_state;
        }

        tct++;

        if (tct == 200)
        {
            tct = 0;
            LED1(1);           /* 关闭 LED1 */
            LED0_TOGGLE();     /* LED0 闪烁 */

            if (g_usb_state_reg & 0x10)
            {
                offline_cnt = 0;    /* USB 连接了,则清除 offline 计数器 */
                g_device_state = 1;
            }
            else    /* 没有得到轮询 */
            {
                offline_cnt++;

                if (offline_cnt > 10)
                {
                    g_device_state = 0;    /* 2s 内没收到在线标记,代表 USB 被拔出了 */
                }
            }

            g_usb_state_reg = 0;
        }
    }
}
```

其中，USBD_HandleTypeDef 是一个用于处理 USB 设备类通信处理的结构体类型，它包含了 USB 设备类通信的各种变量、结构体参数、传输状态和端点信息等。凡是 USB 设备类通信，都必须要用定义一个这样的结构体，这里定义成：USBD_Device。

使用 ST 官方提供的 USB 库以后，整个 USB 初始化就变得比较简单了：

1， 调用 USBD_Init 函数，初始化 USB 从机内核；

2， 调用 USBD_RegisterClass 函数，链接 MSC 设备类驱动程序到设备内核；

3， 调用 USBD_MSC_RegisterStorage 函数，为 MSC 设备类驱动添加回调函数；

4， 调用 USBD_Start 函数，启动 USB 通信；

经过以上四步处理，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_conf.c 里面：

```
/**
 * @brief       USB OTG 中断服务函数
 *   @note      处理所有 USB 中断
 * @param       无
 * @retval      无
 */
void OTG_FS_IRQHandler(void)
{
    HAL_PCD_IRQHandler(&g_pcd_usb_otg_fs);
}
```

该函数调用 HAL_PCD_IRQHandler 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（g_usb_state_reg 和 g_device_state），来判断 USB 状态，并在 LCD 上面显示相关提示信息。

g_usb_state_reg 在 usbd_storage.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。

g_device_state 是在 usbd_conf.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。

39.4 下载验证

将程序下载到开发板后，在 USB 配置成功后（注意：USB 数据线，要插在 USB_OTG 口！而不是 USB_UART 端口！），如图 39.4.1 所示：

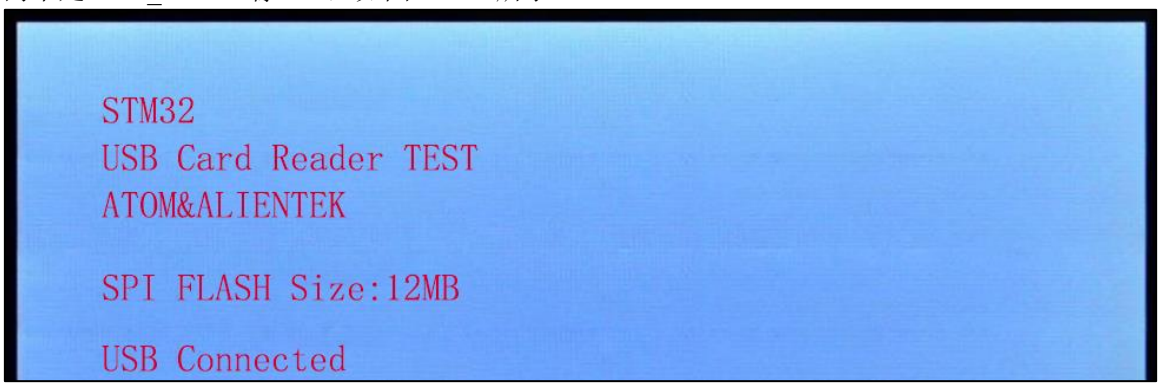


图 39.4.1 USB 连接成功显示界面

此时，电脑提示发现新硬件，并开始自动安装驱动，如图 39.4.2 所示：



图 39.4.2 USB 读卡器被电脑找到

USB 配置成功后，LED1 熄灭，LED0 闪烁，在电脑上可以看到磁盘，如图 39.4.3 所示：



图 39.4.3 电脑找到 USB 读卡器的盘符

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一个磁盘，如图 39.4.4 所示：

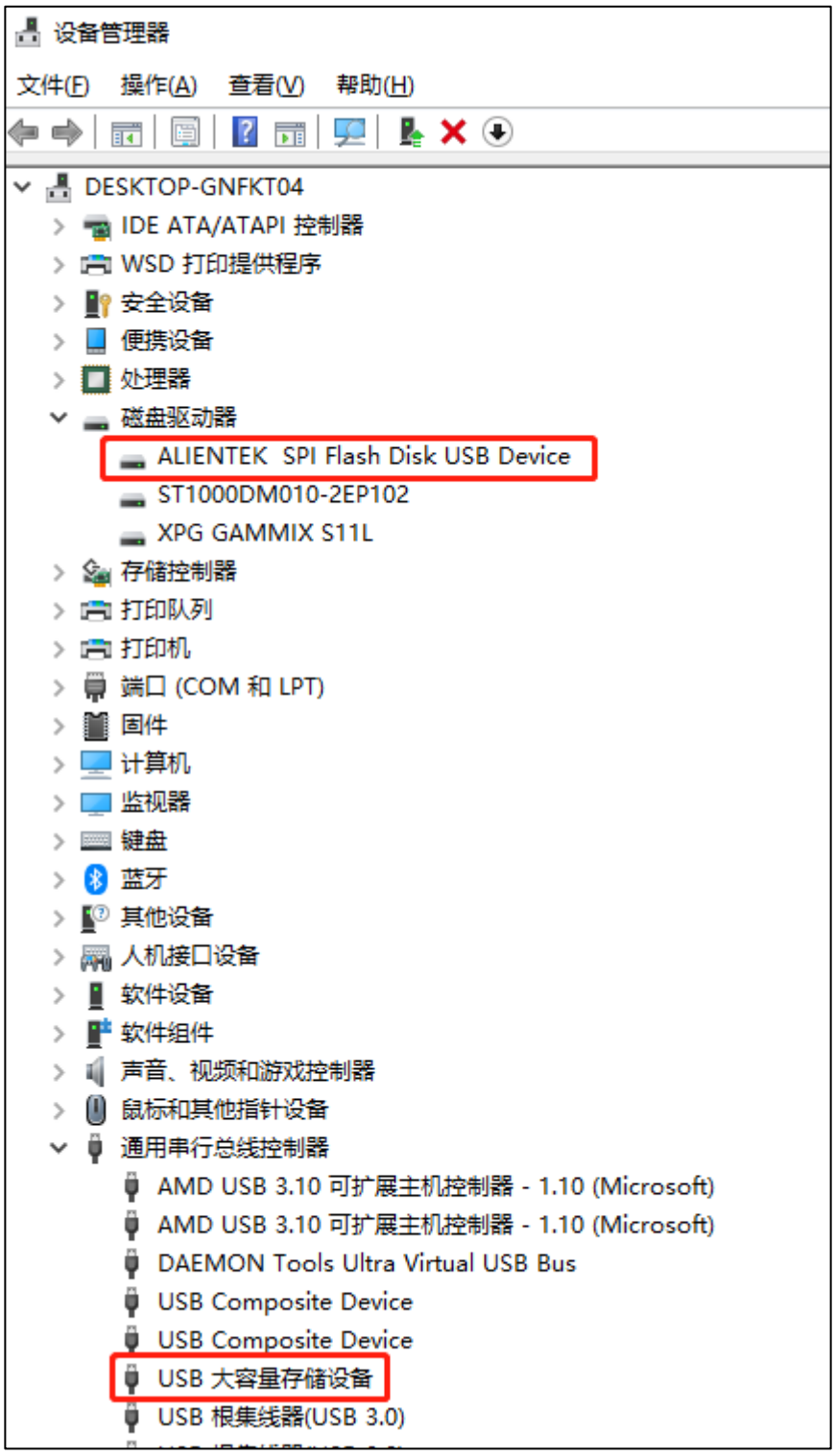


图 39.4.4 通过设备管理器查看磁盘驱动器

此时，我们就可以通过电脑读写 SPI FLASH 里面的内容了。在执行读写操作的时候，就可以看到 LED1 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！

第四十章 USB 虚拟串口(Slave)实验

本章，我们将向大家介绍如何利用 USB 在开发板实现一个 USB 虚拟串口，通过 USB 与电脑数据数据交互。

本章分为如下几个小节：

40.1 USB 虚拟串口简介
40.2 硬件设计
40.3 程序设计
40.4 下载验证

40.1 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→1，STM32CubeF4 固件包 →STM32Cube_FW_F4_V1.26.0→Projects→STM324xG_EVAL→Applications→USB_Device→CDC_Standalone，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。

40.2 硬件设计

1. 例程功能

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 5（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\5，其他小工具\STM32 USB 虚拟串口驱动\VCP_V1.4.0_Setup.exe 这个驱动软件，虚拟串口驱动我们还可以在论坛上下载，链接是：http://www.openedv.com/thread-284178-1-1.html）。

LED0 闪烁，提示程序运行。USB 和电脑连接成功后，LED1 常亮。

2. 硬件资源

1）LED 灯
LED0 – PE0
LED1 – PE1
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）USB_SLAVE 接口(D-/D+连接在 PA11/PA12 上)

40.3 程序设计

40.3.1 程序流程图

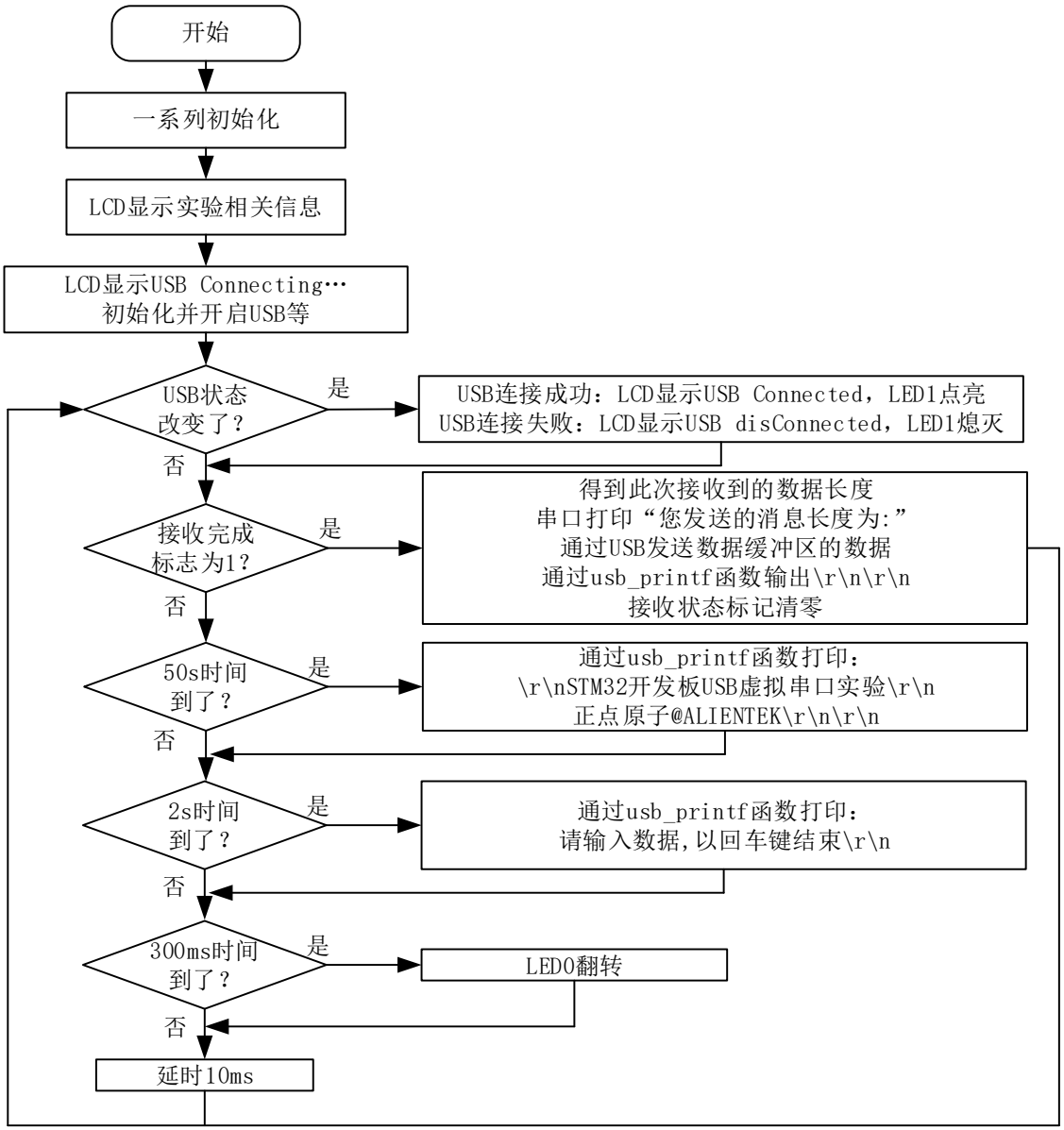


图 40.3.1.1 USB 虚拟串口(Slave)实验程序流程图

40.3.2 程序解析

这里我们只讲解核心代码，详细的源码本实验，在上一个实验的基础上，把不需要的文件从工程中移除，并对照官方 VCP 例子，将相关文件拷贝到 USB 文件夹下。然后，添加 USB 相关代码到工程中，最终得到如图 40.3.2.1 所示的工程：

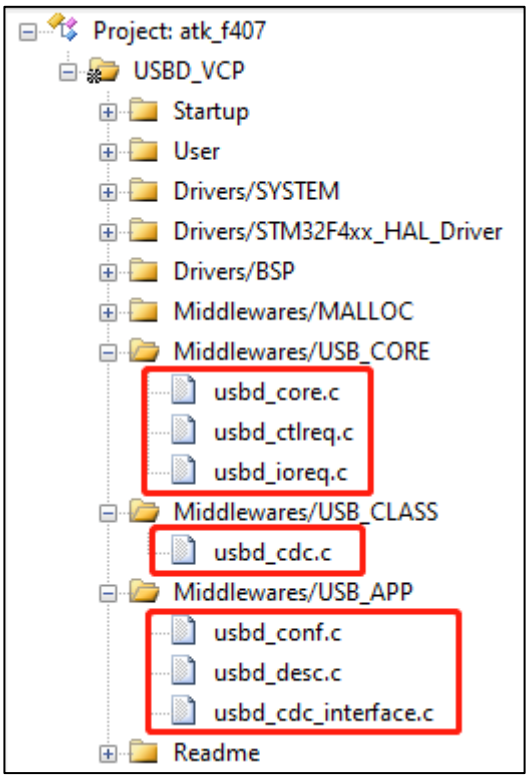


图 40.3.2.1 USB 虚拟串口工程分组

注意：因为 USB 驱动库需要用到内存管理，因此我们保留了相关代码（malloc.c）。

1. USB 驱动代码

可以看到，USB 部分代码，同上一个实验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 里面的这三个.c 文件了。

usbd_conf.c 代码，和上一个实验一样，不需要修改，可以直接使用上一个实验的代码。

usbd_desc.c 代码，同上一个实验不一样，上一个实验描述符是 USB Device 设备，本实验变成了 USB 虚拟串口了（CDC），所以直接用 ST 官方的就行。

usbd_cdc_interface.c 代码，是重点要修改的，首先介绍 usbd_cdc_interface.h 文件的相关宏定义，具体如下：

```
#define USB_USART_REC_LEN       200     /* USB 串口接收缓冲区最大字节数 */

/* 轮询周期，最大 65ms，最小 1ms */
#define CDC_POLLING_INTERVAL    1       /* 轮询周期，最大 65ms，最小 1ms */
```

USB_USART_REC_LEN 宏定义是用于定义 USB 串口接收缓冲区最大字节数，这里设置为 200。CDC_POLLING_INTERVAL 宏定义是用于定义 USB 发送数据轮询周期，作为 delay_ms 函数的参数，最大 65ms，最小 1ms，这里设置为最小值即可。

下面重点介绍 usbd_cdc_interface.c 文件，首先是一些结构体变量、数组和变量的定义，具体如下：

```
/* USB 虚拟串口相关配置参数 */
USBD_CDC_LineCodingTypeDef 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 */
};

/* usb_printf 发送缓冲区，用于 vsprintf */
uint8_t g_usb_usart_printf_buffer[USB_USART_REC_LEN];
```

```
/* USB 接收的数据缓冲区,最大 USART_REC_LEN 个字节,用于 USBD_CDC_SetRxBuffer 函数 */
uint8_t g_usb_rx_buffer[USB_USART_REC_LEN];

/* 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 */
uint8_t g_usb_usart_rx_buffer[USB_USART_REC_LEN];

/* 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
 */
uint16_t g_usb_usart_rx_sta=0;    /* 接收状态标记 */

extern USBD_HandleTypeDef USBD_Device;
static int8_t CDC_Itf_Init(void);
static int8_t CDC_Itf_DeInit(void);
static int8_t CDC_Itf_Control(uint8_t cmd, uint8_t *pbuf, uint16_t length);
static int8_t CDC_Itf_Receive(uint8_t *pbuf, uint32_t *Len);

/* 虚拟串口配置函数(供 USB 内核调用) */
USBD_CDC_ItfTypeDef USBD_CDC_fops =
{
    CDC_Itf_Init,
    CDC_Itf_DeInit,
    CDC_Itf_Control,
    CDC_Itf_Receive
};
```

首先是定义一个 USBD_CDC_LineCodingTypeDef 结构体类型的变量 LineCoding，并赋值。波特率为 115200，停止位和校验位都为 0，数据位,默认 8 位。

g_usb_usart_printf_buffer 是发送缓冲区，大小由 USB_USART_REC_LEN 宏来定义，数组是 uint8_t 类型，所以数字大小为 200 字节。

g_usb_rx_buffer 则是 USB 接收的数据缓冲区，用于 USBD_CDC_SetRxBuffer 函数，大小也是 200 字节。

g_usb_usart_rx_buffer 是用做类似串口 1 接收数据的方法，来处理 USB 虚拟串口接收到的数据，在 cdc_vcp_data_rx 函数中被调用，大小也是 200 字节。

g_usb_usart_rx_sta 变量用于表示接收状态，位 15 表示接收完成标志，位 14 表示接收到 0x0d，位 13~位 0 表示接收到的有效字节数目。

最后定义一个 USBD_CDC_ItfTypeDef 结构体类型的变量 USBD_CDC_fops，供 USB 内核调用，并把四个函数的首地址赋值给其成员。下面会介绍到这几个函数，以及一些其它的函数。

首先是初始化 CDC 函数，其定义如下：

```
/**
 * @brief       初始化 CDC
 * @param       无
 * @retval      USB 状态
 *   @arg       USBD_OK(0)   , 正常;
 *   @arg       USBD_BUSY(1) , 忙;
 *   @arg       USBD_FAIL(2) , 失败;
 */
static int8_t CDC_Itf_Init(void)
{
    USBD_CDC_SetRxBuffer(&USBD_Device, g_usb_rx_buffer);
    return USBD_OK;
}
```

CDC_Itf_Init 用于初始化 VCP，在初始化的时候由 USB 内核调用，这里我们调用函数：USBD_CDC_SetRxBuffer，设置 USB 接收数据缓冲区。USB 虚拟串口收到的数据，会先缓存在这个 buf 里面。

下面介绍的是复位 CDC 函数，其定义如下：

```
/**
```

```
 * @brief       复位 CDC
 * @param       无
 * @retval      USB 状态
 *   @arg       USBD_OK(0)   , 正常;
 *   @arg       USBD_BUSY(1) , 忙;
 *   @arg       USBD_FAIL(2) , 失败;
 */
static int8_t CDC_Itf_DeInit(void)
{
    return USBD_OK;
}
```

CDC_Itf_DeInit 用于复位 VCP，我们用不到，所以直接返回 USBD_OK 即可。

下面介绍的是控制 CDC 的设置函数，其定义如下：

```
/**
 * @brief       控制 CDC 的设置
 * @param       cmd     : 控制命令
 * @param       buf     : 命令数据缓冲区/参数保存缓冲区
 * @param       length  : 数据长度
 * @retval      USB 状态
 *   @arg       USBD_OK(0)   , 正常;
 *   @arg       USBD_BUSY(1) , 忙;
 *   @arg       USBD_FAIL(2) , 失败;
 */
static int8_t CDC_Itf_Control(uint8_t cmd, uint8_t *pbuf, uint16_t length)
{
    switch (cmd)
    {
        case CDC_SEND_ENCAPSULATED_COMMAND:
            break;
        case CDC_GET_ENCAPSULATED_RESPONSE:
            break;
        case CDC_SET_COMM_FEATURE:
            break;
        case CDC_GET_COMM_FEATURE:
            break;
        case CDC_CLEAR_COMM_FEATURE:
            break;
        case CDC_SET_LINE_CODING:
            LineCoding.bitrate = (uint32_t) (pbuf[0] | (pbuf[1] << 8) |
                                             (pbuf[2] << 16) | (pbuf[3] << 24));
            LineCoding.format = pbuf[4];
            LineCoding.paritytype = pbuf[5];
            LineCoding.datatype = pbuf[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 CDC_GET_LINE_CODING:
            pbuf[0] = (uint8_t) (LineCoding.bitrate);
            pbuf[1] = (uint8_t) (LineCoding.bitrate >> 8);
            pbuf[2] = (uint8_t) (LineCoding.bitrate >> 16);
            pbuf[3] = (uint8_t) (LineCoding.bitrate >> 24);
            pbuf[4] = LineCoding.format;
            pbuf[5] = LineCoding.paritytype;
            pbuf[6] = LineCoding.datatype;
            break;
        case CDC_SET_CONTROL_LINE_STATE:
            break;
        case CDC_SEND_BREAK:
            break;
        default:
            break;
```

```
    }
    return USBD_OK;
}
```

CDC_Itf_Control 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 CDC_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。

下面介绍的是 CDC 数据接收函数和处理从 USB 虚拟串口接收到的数据函数，它们的定义如下：

```
/**
 * @brief       CDC 数据接收函数
 * @param       buf     : 接收数据缓冲区
 * @param       len     : 接收到的数据长度
 * @retval      USB 状态
 *   @arg       USBD_OK(0)   , 正常;
 *   @arg       USBD_BUSY(1) , 忙;
 *   @arg       USBD_FAIL(2) , 失败;
 */
static int8_t CDC_Itf_Receive(uint8_t *buf, uint32_t *len)
{
    SCB_CleanDCache_by_Addr((uint32_t *)buf, *len);
    USBD_CDC_ReceivePacket(&USBD_Device);
    cdc_vcp_data_rx(buf, *len);
    return USBD_OK;
}

/**
 * @brief       处理从 USB 虚拟串口接收到的数据
 * @param       buf     : 接收数据缓冲区
 * @param       len     : 接收到的数据长度
 * @retval      无
 */
void cdc_vcp_data_rx(uint8_t *buf, uint32_t Len)
{
    uint8_t i;
    uint8_t res;

    for (i = 0; i < Len; i++)
    {
        res = buf[i];
        if ((g_usb_usart_rx_sta & 0x8000) == 0)      /* 接收未完成 */
        {
            if (g_usb_usart_rx_sta & 0x4000)         /* 接收到了 0x0d */
            {
                if (res != 0x0a)
                {
                    g_usb_usart_rx_sta = 0;          /* 接收错误,重新开始 */
                }
                else
                {
                    g_usb_usart_rx_sta |= 0x8000;    /* 接收完成了 */
                }
            }
            else    /* 还没收到 0x0D */
            {
                if (res == 0x0D)
                {
                    g_usb_usart_rx_sta |= 0x4000;    /* 标记接收到了 0x0D */
                }
                else
                {
                    g_usb_usart_rx_buffer[g_usb_usart_rx_sta & 0x3FFF] = res;
```

```
                g_usb_usart_rx_sta++;
                if (g_usb_usart_rx_sta > (USB_USART_REC_LEN - 1))
                {
                    g_usb_usart_rx_sta = 0;        /* 接收数据溢出 重新开始接收 */
                }
            }
        }
    }
}
```

CDC_Itf_Receive 和 cdc_vcp_data_rx，这两个函数一起，用于 VCP 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 CDC_Itf_Receive，然后在该函数里面再调用 cdc_vcp_data_rx 函数，实现 VCP 的数据接收，只需要在该函数里面，将接收到的数据，保存起来即可，接收的原理和 实验 5 串口通信实验 完全一样。

下面介绍的是通过 USB 发送数据函数，其定义如下：

```
/**
 * @brief       通过 USB 发送数据
 * @param       buf     : 要发送的数据缓冲区
 * @param       len     : 数据长度
 * @retval      无
 */
void cdc_vcp_data_tx(uint8_t *data, uint32_t Len)
{
    USBD_CDC_SetTxBuffer(&USBD_Device, data, Len);
    USBD_CDC_TransmitPacket(&USBD_Device);
    delay_ms(CDC_POLLING_INTERVAL);
}
```

cdc_vcp_data_rx 用于发送 Len 个字节的数据给 VCP，由 VCP 通过 USB 传输给电脑，实现 VCP 的数据发送。

下面介绍的是通过 USB 格式化输出函数，其定义如下：

```
/**
 * @brief       通过 USB 格式化输出函数
 *   @note      通过 USB VCP 实现 printf 输出
 *              确保一次发送数据长度不超 USB_USART_REC_LEN 字节
 * @param       格式化输出
 * @retval      无
 */
void usb_printf(char *fmt, ...)
{
    uint16_t i;
    va_list ap;
    va_start(ap, fmt);
    vsprintf((char *)g_usb_usart_printf_buffer, fmt, ap);
    va_end(ap);
    i = strlen((const char *)g_usb_usart_printf_buffer);    /* 此次发送数据的长度 */
    cdc_vcp_data_tx(g_usb_usart_printf_buffer, i);          /* 发送数据 */
    SCB_CleanDCache_by_Addr((uint32_t *)g_usb_usart_printf_buffer, i);
}
```

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：UM1734(STM32Cube USB device library).pdf 这个文档。

2. main.c 代码

下面是 main.c 的程序，具体如下：

```
USBD_HandleTypeDef USBD_Device;             /* USB Device 处理结构体 */
extern volatile uint8_t g_device_state;     /* USB 连接 情况 */

int main(void)
```

```
{
    uint16_t len;
    uint16_t times = 0;
    uint8_t usbstatus = 0;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */
    my_mem_init(SRAMIN);                    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM);                   /* 初始化内部 SRAMCCM 内存池 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USB Virtual USART TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    /* 提示 USB 开始连接 */
    lcd_show_string(30, 110, 200, 16, 16, "USB Connecting...", RED);

    USBD_Init(&USBD_Device, &VCP_Desc, DEVICE_FS);      /* 初始化 USB */
    USBD_RegisterClass(&USBD_Device, USBD_CDC_CLASS);   /* 添加类 */
    /* 为 MSC 类添加回调函数 */
    USBD_CDC_RegisterInterface(&USBD_Device, &USBD_CDC_fops);
    USBD_Start(&USBD_Device);   /* 开启 USB */
    delay_ms(1800);

    while (1)
    {
        delay_ms(1);
        if (usbstatus != g_device_state)     /* USB 连接状态发生了改变 */
        {
            usbstatus = g_device_state;      /* 记录新的状态 */

            if (usbstatus == 1)
            {
                /* 提示 USB 连接成功 */
                lcd_show_string(30, 110, 200, 16, 16, "USB Connected    ", RED);
                LED1(0);    /* 绿灯亮 */
            }
            else
            {
                /* 提示 USB 断开 */
                lcd_show_string(30, 110, 200, 16, 16, "USB disConnected ", RED);
                LED1(1);    /* 绿灯灭 */
            }
        }

        if (g_usb_usart_rx_sta & 0x8000)
        {
            len = g_usb_usart_rx_sta & 0x3FFF;  /* 得到此次接收到的数据长度 */
            usb_printf("\r\n 您发送的消息长度为:%d\r\n\r\n", len);
            cdc_vcp_data_tx(g_usb_usart_rx_buffer, len);
            usb_printf("\r\n\r\n");              /* 插入换行 */
            g_usb_usart_rx_sta = 0;
        }
        else
        {
            times++;

            if (times % 5000 == 0)
            {
                usb_printf("\r\nSTM32 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，首先定义了 USBD_Device 结构体，然后通过 USBD_Init 等函数初始化 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。

40.4 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（V1.5.0 版）下载地址：http://www.openedv.com/thread-284178-1-1.html，下载完以后，根据自己电脑的系统，选择合适的驱动安装即可。

将程序下载到开发板后（注意：USB 数据线，要插在 USB_SLAVE 口！而不是 USB_UART 端口！），我们打开设备管理器（我用的是 WIN10），在端口（COM 和 LPT）里面可以发现多出了一个 COM4 的设备，这就是 USB 虚拟的串口设备端口，如图 40.4.1 所示：

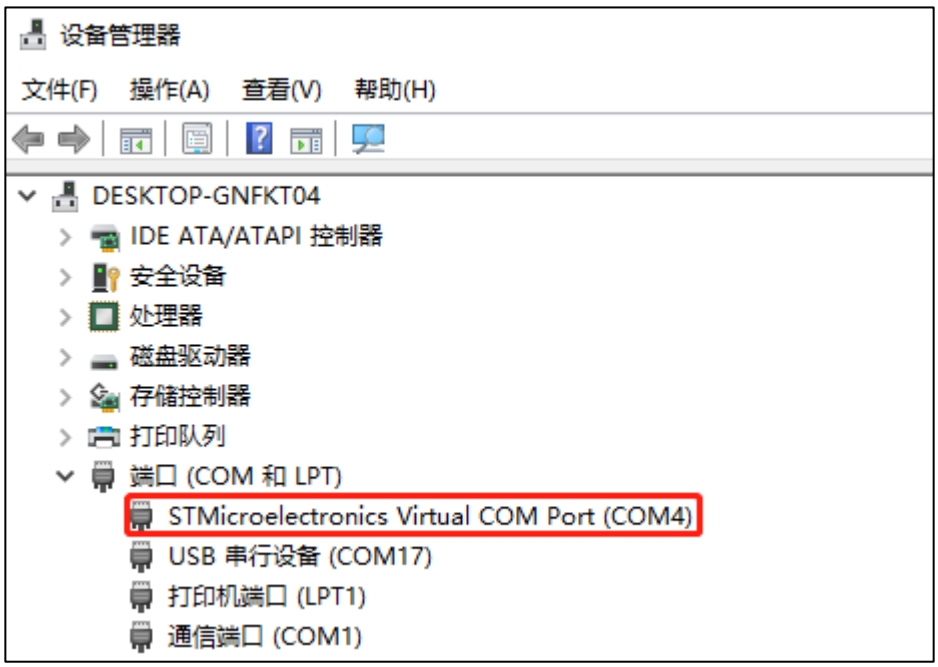


图 40.4.1 通过设备管理器查看 USB 虚拟的串口设备端口

如图 40.4.1，STM32 通过 USB 虚拟的串口，被电脑识别了，端口号为：COM4（可变），字符串名字为：STMicroelectronics Virtual COM Port（COM4）。此时，开发板的 LDE1 常亮，同时，LED0 在闪烁，提示程序运行。开发板的 LCD 显示 USB Connected，如图 40.4.2 所示：

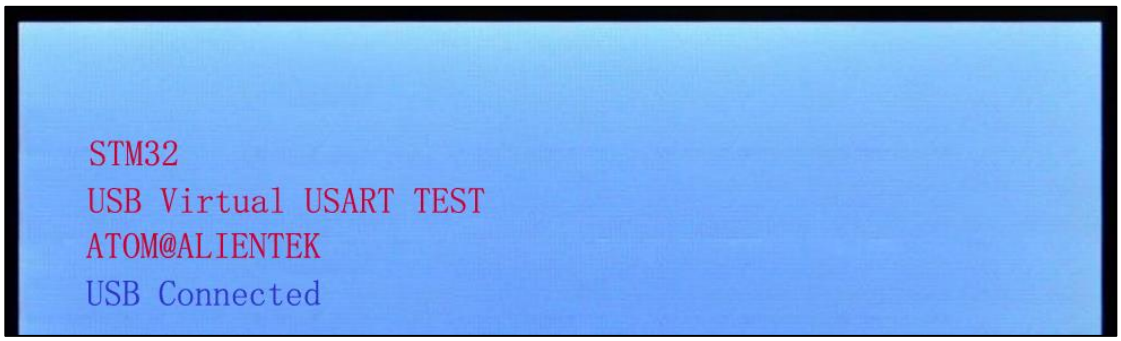


图 40.4.2 USB 虚拟串口连接成功

然后我们打开 XCOM，选择 COM4（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 40.4.3 所示：

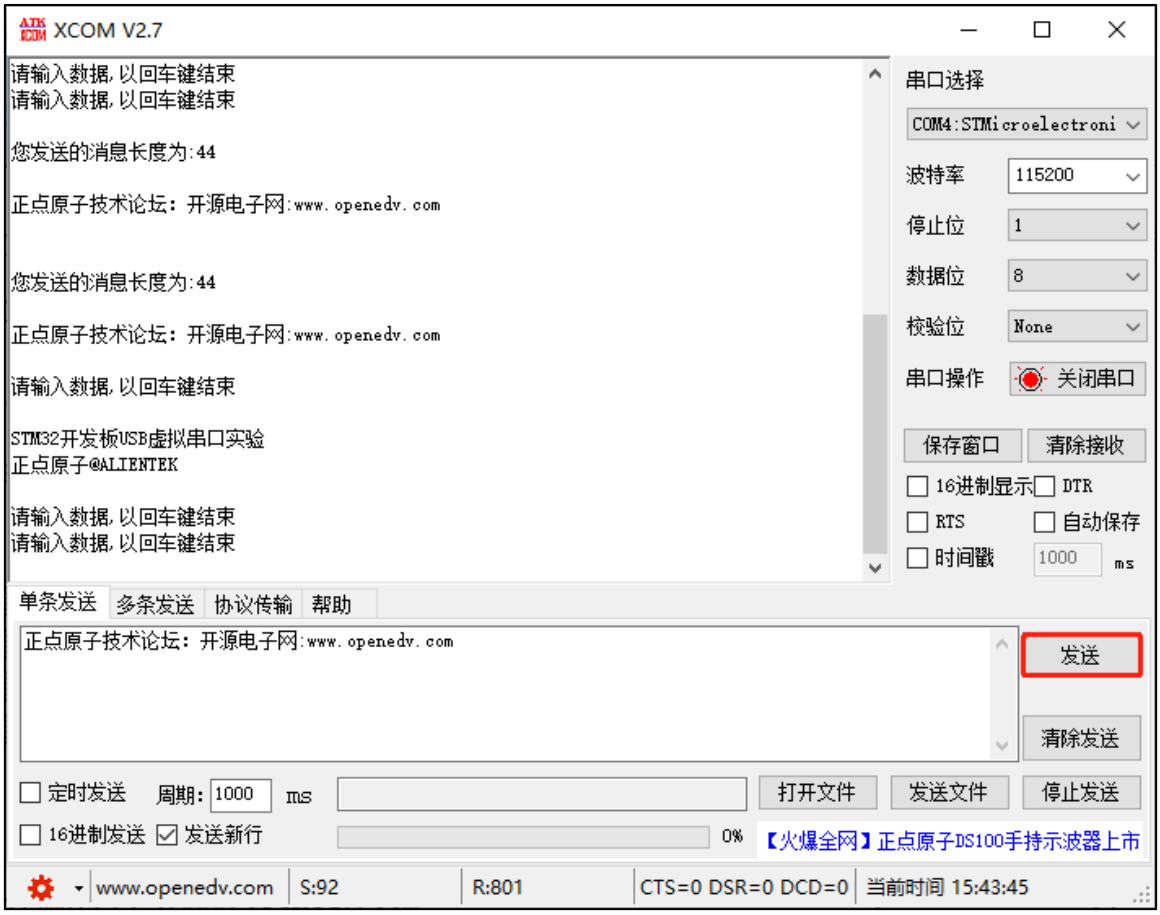


图 40.4.3 STM32 虚拟串口通信测试

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。

至此，USB 虚拟串口实验就完成了，通过本实验，我们就可以利用 STM32 的 USB，直接和电脑进行数据互传了，具有广泛的应用前景。

第四十一章 USB 鼠标键盘(Host)实验

本章我们介绍如何使用 STM32F407 的 USB HOST 来驱动 USB 鼠标/键盘。
本章分为如下几个小节：
41.1 USB 鼠标键盘简介
41.2 硬件设计
41.3 程序设计
41.4 下载验证

41.1 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。

我们直接移植官方的 USB HID 例程，官方例程路径：8，STM32 参考资料 → 1，STM32CubeF4 固件包 → STM32Cube_FW_F4_V1.26.0 → Projects → STM324xG_EVAL → Applications → USB_ Host → HID_Standalone，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我们的开发板上。

41.2 硬件设计

1. 例程功能

本实验代码，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）。

2. 硬件资源

1）LED 灯
LED0 – PE0
LED1 – PE1
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）USB_HOST 接口(D-/D+连接在 PA11/PA12 上)

3. 原理图

本开发板的 USB HOST 接口采用的是贴片 USB 母座，它和 USB SLAVE 的 Type-C 接头是共用 USB_DM 和 USB_DP 信号的，所以 USB HOST 和 USB SLAVE 功能不能同时使用。USB HOST 和 STM32 的连接原理图，如下图所示：

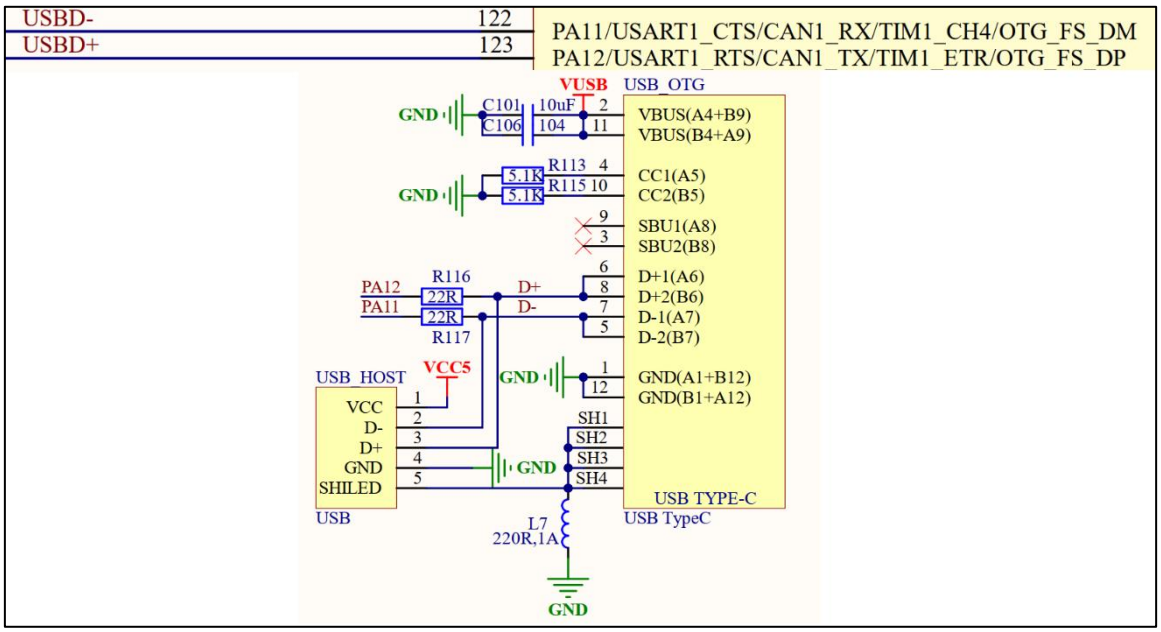


图 41.2.1 USB 母座与 STM32 的连接电路图

从上图可以看出，USB 座是直接连接到 STM32F407 上面的，所以硬件上不需要我们做什么操作，可直接使用。

下面，看看开发板上的 USB 母座（USB_HOST）的位置，请将键盘鼠标插入这个接口上。

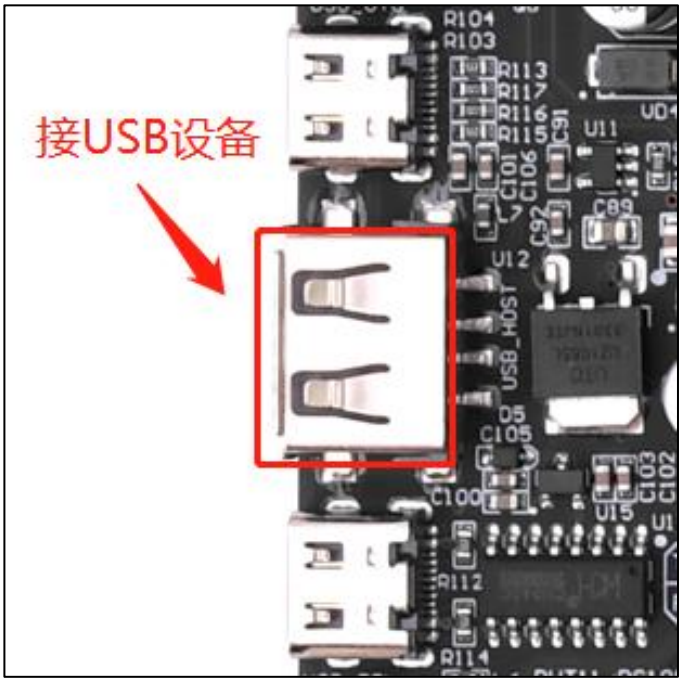


图 41.2.2 USB_HOST 与开发板的连接示意图

41.3 程序设计

41.3.1 程序流程图

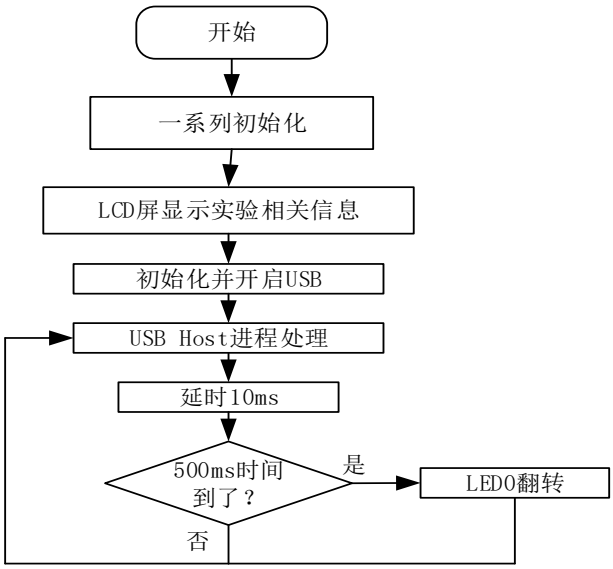


图 41.3.1.1 USB 鼠标键盘实验(Host)程序流程图

41.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。

本实验，我们在上一个实验的基础上修改，将 USB Device 换成 USB Host，然后，对照官方 HID 例子，我们在工程里面添加 USB HID 相关代码，如图 41.3.2.1 所示：

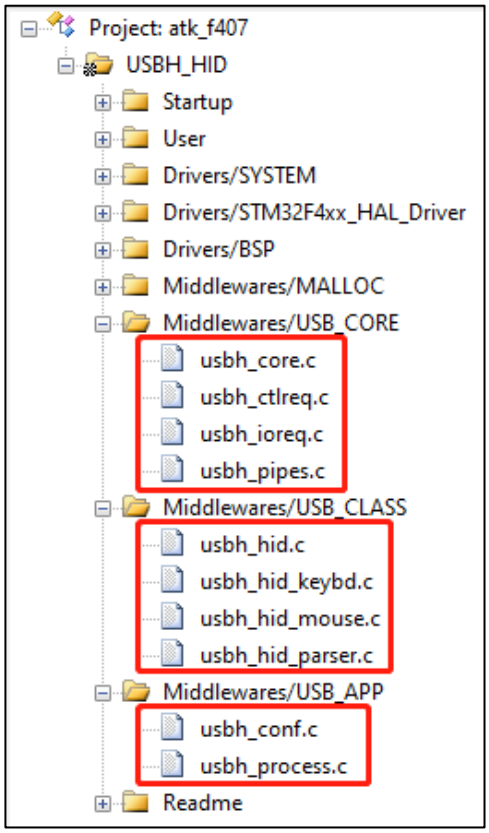


图 41.3.2.1 USB 鼠标键盘工程分组

上图工程分组中的Middlewares/USB_CORE和Middlewares/USB_CLASS分组下的.c文件，直接拷贝 ST 官方 USB HOST 库。我们重点介绍下 USB_APP 里面的 usbh_hid_mouse.c 和 usbh_processs 文件，usbh_process 文件是我们自己写的 demo 文件，usbh_hid_mouse.c 文件拷贝自 ST 官方例程，但是 ST 官方例程对鼠标的支持不好（兼容性差，不支持滚轮等），我们重写了该文件，下面介绍这两个文件和 main.c。

1. USB 驱动代码

usbh_conf.c 提供了 USB 主机库的回调及 MSP 初始化函数，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。该.c 文件我们重点介绍 3 个函数，首先是初始化 PCD MSP 函数，其定义如下：

```
/**
 * @brief       初始化 PCD MSP
 * @param       hpcd:PCD 结构体指针
 * @retval      无
 */
void HAL_PCD_MspInit(PCD_HandleTypeDef * hpcd)
{
    GPIO_InitTypeDef gpio_init_struct = { 0 };

    if (hpcd->Instance == USB_OTG_FS)
    {
        __HAL_RCC_GPIOA_CLK_ENABLE();                           /* 使能 GPIOA 时钟 */

        gpio_init_struct.Pin = GPIO_PIN_11 | GPIO_PIN_12;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用 */
        gpio_init_struct.Pull = GPIO_NOPULL;                    /* 浮空 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;     /* 高速 */
        gpio_init_struct.Alternate = GPIO_AF10_OTG_FS;          /* 复用为 OTG_FS */
        HAL_GPIO_Init(GPIOA, &gpio_init_struct);/* 初始化 PA11 和 PA12 引脚 */

        __HAL_RCC_USB_OTG_FS_CLK_ENABLE();          /* 使能 OTG FS 时钟 */
        HAL_NVIC_SetPriority(OTG_FS_IRQn, 1, 0);/* 优先级设置为抢占 1,子优先级 0 */
        HAL_NVIC_EnableIRQ(OTG_FS_IRQn);            /* 使能 OTG FS 中断 */
    }
    else if (hpcd->Instance == USB_OTG_HS)
    {
        /* USB_OTG_HS 本例程没用到,故不做处理 */
    }
}
```

HAL_HCD_MspInit 函数，用于使能 USB 时钟，选择内核时钟源，初始化 IO 口，设置中断等。该函数在 HAL_HCD_Init 函数里面被调用。

接下来介绍的是 USB OTG 中断服务函数，其定义如下：

```
/**
 * @brief       USB OTG 中断服务函数
 *   @note      处理所有 USB 中断
 * @param       无
 * @retval      无
 */
void OTG_FS_IRQHandler(void)
{
    HAL_HCD_IRQHandler(&hhcd);
}
```

OTG_FS_IRQHandler 函数，是 USB 的中断服务函数，通过调用 HAL_HCD_IRQHandler 函数，实现对 USB 各种事务的处理。

usbh_hid_mouse.h，首先介绍 usbh_hid_mouse.h 里面定义的一个结构体，具体如下：

```
/* 鼠标信息结构体*/
typedef struct _HID_MOUSE_Info
```

```
{
  uint8_t             x;          /* x 轴增量（强制转换成 signed char 后使用）*/
  uint8_t             y;          /* y 轴增量（强制转换成 signed char 后使用）*/
  uint8_t             z;          /* z 轴增量（强制转换成 signed char 后使用）*/
  uint8_t             button;     /* 将 buttons 修改为 button，存储按键状态 */
}
HID_MOUSE_Info_TypeDef;
```

HID_MOUSE_Info_TypeDef 结构体中，成员 x 和 y 表示鼠标的横向/纵向移动的增量值，成员 z 表示鼠标的滚轮的滚动增量值，在使用的时候，需要强制转换成 unsigned char 类型后使用（有±）。button 的 bit0、bit1、bit2 分别表示鼠标的：左键、右键和滚轮键的按下状态。我们通过这个结构体，就可以获得当前鼠标的输入状态。

usbh_hid_mouse.c，总共有三个函数，首先介绍的是 USBH 鼠标初始化函数，其定义如下：

```
/* 鼠标信息(坐标+按钮状态) */
HID_MOUSE_Info_TypeDef    mouse_info;
/* 鼠标上报数据长度,最多 HID_QUEUE_SIZE 个字节 */
uint8_t mouse_report_data[HID_QUEUE_SIZE];

/**
 * @brief       USBH 鼠标初始化
 * @param       phost       : USBH 句柄指针
 * @retval      USB 状态
 *   @arg       USBH_OK(0)       , 正常;
 *   @arg       USBH_BUSY(1)     , 忙;
 *   @arg       其他             , 失败;
 */
USBH_StatusTypeDef USBH_HID_MouseInit(USBH_HandleTypeDef *phost)
{
    HID_HandleTypeDef *HID_Handle =
                      (HID_HandleTypeDef *) phost->pActiveClass->pData;
    mouse_info.x = 0;
    mouse_info.y = 0;
    mouse_info.button = 0;
    if (HID_Handle->length > sizeof(mouse_report_data))
    {
        HID_Handle->length = sizeof(mouse_report_data);
    }
    HID_Handle->pData = (uint8_t *)mouse_report_data;
    USBH_HID_FifoInit(&HID_Handle->fifo, phost->device.Data, HID_QUEUE_SIZE);
    return USBH_OK;
}
```

USBH_HID_MouseInit 函数，用于初始化鼠标，指定 USB 鼠标输出数据的缓存，初始化 FIFO 等，该函数在 USBH_HID_Process 函数里面被调用。

接下来介绍的是 USBH 获取鼠标信息函数，其定义如下：

```
/**
 * @brief       USBH 获取鼠标信息
 * @param       phost       : USBH 句柄指针
 * @retval      鼠标信息(HID_MOUSE_Info_TypeDef)
 */
HID_MOUSE_Info_TypeDef *USBH_HID_GetMouseInfo(USBH_HandleTypeDef *phost)
{
    if (USBH_HID_MouseDecode(phost) == USBH_OK)
    {
        return &mouse_info;
    }
    else
    {
        return NULL;
    }
}
```

USBH_HID_GetMouseInfo 函数，用于获取鼠标信息，通过 USBH_HID_MouseDecode 函数获取当前鼠标传回来的数据。该函数在 USB_Demo 函数里面被调用。

最后介绍的是 USBH 鼠标数据解析函数，其定义如下：

```
/**
 * @brief       USBH 鼠标数据解析函数
 * @param       phost       : USBH 句柄指针
 * @retval      USB 状态
 *   @arg       USBH_OK(0)      , 正常;
 *   @arg       USBH_BUSY(1)    , 忙;
 *   @arg       其他            , 失败;
 */
USBH_StatusTypeDef USBH_HID_MouseDecode(USBH_HandleTypeDef *phost)
{
    HID_HandleTypeDef *HID_Handle =
                      (HID_HandleTypeDef *) phost->pActiveClass->pData;

    if (HID_Handle->length == 0)return USBH_FAIL;

    if (USBH_HID_FifoRead(&HID_Handle->fifo, &mouse_report_data,
                      HID_Handle->length) == HID_Handle->length) /* 读取 FIFO */
    {
        mouse_info.button = mouse_report_data[0];
        mouse_info.x      = mouse_report_data[1];
        mouse_info.y      = mouse_report_data[2];
        mouse_info.z      = mouse_report_data[3];
        return USBH_OK;
    }
    return  USBH_FAIL;
}
```

USBH_HID_MouseDecode 函数，用于解析 USB HOST 获取到的 USB 鼠标数据，并存放到 mouse_info 结构体里面。该函数被 USBH_HID_GetMouseInfo 函数调用。

2. usbh_process.c 代码

```
USBH_HandleTypeDef hUSBHost;    /* USB Host 处理结构体 */

/* 应用状态结构体类型定义 */
typedef enum
{
    APPLICATION_IDLE = 0,       /* 挂起状态 */
    APPLICATION_DISCONNECT,     /* 断开连接 */
    APPLICATION_START,          /* 应用开始（连接上了） */
    APPLICATION_READY,          /* 准备完成（可以执行相关应用代码了） */
    APPLICATION_RUNNING,        /* 运行中 */
} HID_ApplicationTypeDef;


extern HID_MOUSE_Info_TypeDef mouse_info;
HID_ApplicationTypeDef App_State = APPLICATION_IDLE;

uint8_t g_usb_first_plugin_flag = 0; /* USB 第一次插入标志,如果为 1,说明是第一次插入 */


/**
 * @brief       USB 信息显示
 * @param       msg   : 信息类型
 *   @arg             0, USB 无连接
 *   @arg             1, USB 键盘
 *   @arg             2, USB 鼠标
 *   @arg             3, 不支持的 USB 设备
 * @retval      无
 */
void usbh_msg_show(uint8_t msgx)
{
```

```
    switch (msgx)
    {
        case 0:  /* USB 无连接 */
            lcd_show_string(30, 130, 200, 16, 16, "USB Connecting...", RED);
            lcd_fill(0, 150, lcddev.width, lcddev.height, WHITE);
            break;

        case 1:  /* USB 键盘 */
            lcd_show_string(30, 130, 200, 16, 16, "USB Connected    ", RED);
            lcd_show_string(30, 150, 200, 16, 16, "USB KeyBoard", RED);
            lcd_show_string(30, 180, 210, 16, 16, "KEYVAL:", RED);
            lcd_show_string(30, 200, 210, 16, 16, "INPUT STRING:", RED);
            break;

        case 2:  /* USB 鼠标 */
            lcd_show_string(30, 130, 200, 16, 16, "USB Connected    ", RED);
            lcd_show_string(30, 150, 200, 16, 16, "USB Mouse", RED);
            lcd_show_string(30, 180, 210, 16, 16, "BUTTON:", RED);
            lcd_show_string(30, 200, 210, 16, 16, "X POS:", RED);
            lcd_show_string(30, 220, 210, 16, 16, "Y POS:", RED);
            lcd_show_string(30, 240, 210, 16, 16, "Z POS:", RED);
            break;

        case 3:  /* 不支持的 USB 设备 */
            lcd_show_string(30, 130, 200, 16, 16, "USB Connected    ", RED);
            lcd_show_string(30, 150, 200, 16, 16, "Unknow Device", RED);
            break;
    }
}

/**
 * @brief       完成 USBH 不同用户状态下的数据处理
 * @param       phost : USBH 句柄指针
 * @param       id    : USBH 当前的用户状态
 * @retval      无
 */
void USBH_UserProcess(USBH_HandleTypeDef *phost, uint8_t id)
{
    uint8_t dev_type = 0xFF;    /* 设备类型,1,键盘;2,鼠标;其他,不支持的设备 */

    switch (id)
    {
        case HOST_USER_SELECT_CONFIGURATION:
            break;

        case HOST_USER_DISCONNECTION:
            usbh_msg_show(0);    /* 显示已经断开连接,准备重新连接 */
            App_State = APPLICATION_DISCONNECT;
            break;

        case HOST_USER_CLASS_ACTIVE:
            App_State = APPLICATION_READY;
            dev_type = phost->device.CfgDesc.Itf_Desc[phost->
                                device.current_interface].bInterfaceProtocol;

            if (dev_type == HID_KEYBRD_BOOT_CODE)        /* 键盘设备 */
            {
                g_usb_first_plugin_flag = 1;             /* 标记第一次插入 */
                usbh_msg_show(1);                        /* 显示键盘界面 */
            }
            else if (dev_type == HID_MOUSE_BOOT_CODE)    /* 鼠标设备 */
            {
                g_usb_first_plugin_flag = 1;             /* 标记第一次插入 */
                usbh_msg_show(2);                        /* 显示鼠标界面 */
```

```
            }
            else
            {
                usbh_msg_show(3);                           /* 显示不支持 */
            }

            break;

        case HOST_USER_CONNECTION:
            App_State = APPLICATION_START;
            break;

        default:
            break;
    }
}

/* 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) */
#if !(__ARMCC_VERSION >= 6010050)    /* 不是 AC6 编译器，即使用 AC5 编译器时 */
__align(4) uint8_t g_temp_buffer[2017];
#else    /* 使用 AC6 编译器时 */
__ALIGNED(4) uint8_t g_temp_buffer[2017];
#endif

/**
 * @brief       USB 鼠标数据处理
 * @param       data  : USB 鼠标数据结构体指针
 * @retval      无
 */
void mouse_data_process(HID_MOUSE_Info_TypeDef *data)
{
    static signed short x, y, z;

    if (g_usb_first_plugin_flag)    /* 第一次插入,将数据清零 */
    {
        g_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;
    }

    sprintf((char *)g_temp_buffer, "BUTTON:");

    if (data->button & 0x01)
    {
        strcat((char *)g_temp_buffer, "LEFT ");
    }

    if ((data->button & 0x03) == 0x02)
    {
        strcat((char *)g_temp_buffer, "RIGHT");
    }
    else
    {
        if ((data->button & 0x03) == 0x03)
        {
            strcat((char *)g_temp_buffer,"+RIGHT");
        }
    }

    if ((data->button & 0x07) == 0x04)
    {
        strcat((char *)g_temp_buffer, "MID  ");
    }
    else
    {
        if ((data->button & 0x07) > 0x04)
        {
            strcat((char *)g_temp_buffer, "+MID");
        }
    }

    lcd_fill(30 + 56, 180, lcddev.width - 1, 180 + 16, WHITE);
    lcd_show_string(30, 180, 210, 16, 16, (char *)g_temp_buffer, BLUE);
    sprintf((char *)g_temp_buffer, "X POS:%05d", x);
    lcd_show_string(30, 200, 200, 16, 16, (char *)g_temp_buffer, BLUE);
    sprintf((char *)g_temp_buffer, "Y POS:%05d", y);
    lcd_show_string(30, 220, 200, 16, 16, (char *)g_temp_buffer, BLUE);
    sprintf((char *)g_temp_buffer, "Z POS:%05d", z);
    lcd_show_string(30, 240, 200, 16, 16, (char *)g_temp_buffer, BLUE);
    //printf("btn,X,Y,Z:0x%x,%d,%d,%d\r\n",data->button,
    (signed char)data->x,(signed char)data->y,(signed char)data->z);
}

/**
 * @brief       USB 键盘数据处理
 * @param       data  : USB 键盘键值
 * @retval      无
 */
void  keybrd_data_process(uint8_t data)
{
    static uint16_t pos;
    static uint16_t endx, endy;
    static uint16_t maxinputchar;

    uint8_t buf[4];
```

```
    if (g_usb_first_plugin_flag)    /* 第一次插入,将数据清零 */
    {
        g_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;
    }

    sprintf((char *)buf, "%02X", data);
    lcd_show_string(30 + 56, 180, 200, 16, 16, (char *)buf, BLUE);  /* 显示键值 */

    if (data >= ' ' && data <= '~')
    {
        g_temp_buffer[pos++] = data;
        g_temp_buffer[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--;
        }
        g_temp_buffer[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_show_string(30, 220, endx - 30, endy - 220, 16,
                        (char *)g_temp_buffer, BLUE);
    }

    //printf("KEY Board Value:%02X\r\n", data);
    //printf("KEY Board Char:%c\r\n", data);
}

/**
 * @brief       USB 键盘/鼠标演示 demo 数据处理
 * @param       phost : USBH 句柄指针
 * @retval      无
 */
void usb_demo(USBH_HandleTypeDef *phost)
{
    char c;
    HID_KEYBD_Info_TypeDef *k_pinfo;
    HID_MOUSE_Info_TypeDef *m_pinfo;

    if (App_State == APPLICATION_READY)
    {
        if (USBH_HID_GetDeviceType(phost) == HID_KEYBOARD)  /* 键盘设备 */
        {
            k_pinfo = USBH_HID_GetKeybdInfo(phost);          /* 获取键盘信息 */

            if (k_pinfo != NULL)
            {
                c = USBH_HID_GetASCIICode(k_pinfo);          /* 转换成 ASCII 码 */
                keybrd_data_process(c);    /* 在 LCD 上显示出键盘字符 */
```

```
            }
        }
        else if (USBH_HID_GetDeviceType(phost) == HID_MOUSE)/* 鼠标设备 */
        {
            m_pinfo = USBH_HID_GetMouseInfo(phost);          /* 获取鼠标信息 */

            if (m_pinfo != NULL)
            {
                mouse_data_process(&mouse_info);             /* LCD 上显示鼠标信息 */
            }
        }
    }
}


/*****************************************************************************/
/* 以下代码 usbh_check_enume_dead 和 usbh_hid_reconnect 用于 HID 死机处理,
   提高代码健壮性 */

/**
 * @brief       USB 枚举状态死机检测,防止 USB 枚举失败导致的死机
 * @param       phost : USB HOST 结构体指针
 * @retval      0, 没有死机
 *              1, 死机了,外部必须重新启动 USB 连接.
 */
uint8_t usbh_check_enume_dead(USBH_HandleTypeDef *phost)
{
    static uint16_t errcnt = 0;

    /* 这个状态,如果持续存在,则说明 USB 死机了 */
    if (phost->gState == HOST_ENUMERATION && (phost->EnumState == ENUM_IDLE ||
        phost->EnumState == ENUM_GET_FULL_DEV_DESC))
    {
        errcnt++;

        if (errcnt > 2000)    /* 死机了 */
        {
            errcnt = 0;
            return 1;
        }
    }
    else
    {
        errcnt = 0;
    }

    return 0;
}

/**
 * @brief       USB HID 重新连接
 * @param       无
 * @retval      无
 */
void usbh_hid_reconnect(void)
{
    /* 关闭之前的连接 */
    HID_Class.DeInit(&hUSBHost);    /* 复位 HID */
    USBH_DeInit(&hUSBHost);         /* 复位 USB HOST */

    /* 重新复位 USB */
    RCC->AHB2RSTR |= 1 << 7;        /* USB OTG FS 复位 */
    delay_ms(5);
```

```
    RCC->AHB2RSTR &= ~(1 << 7);      /* 复位结束 */

    memset(&hUSBHost, 0, sizeof(hUSBHost));    /* 清零数据 */

    /* 重新连接 USB HID 设备 */
    USBH_Init(&hUSBHost, USBH_UserProcess, 0);
    USBH_RegisterClass(&hUSBHost, USBH_HID_CLASS);
    USBH_Start(&hUSBHost);
}
```

usbh_msg_show 函数，用于显示一些提示信息，在 USBH_UserProcess 函数里面被调用。

USBH_UserProcess 函数，用于执行 USB 主机不同用户状态下的数据处理，在 usbh_core.c 等相关文件里面被调用，该函数仅针对两个状态进行处理：

1）HOST_USER_DISCONNECTION，表示连接断开了，我们在屏幕显示连接断开，并设置 App_State 状态为 APPLICATION_DISCONNECT；

2）HOST_USER_CLASS_ACTIVE，表示 USB 主机类激活，即接入 USB 设备已经准备就绪，可以正常工作了，我们设置 App_State 状态为：APPLICATION_READY，并根据接入设备的不同（鼠标/键盘）显示不同的信息。

mouse_data_process 函数，用于处理 USB 鼠标数据，在液晶屏上面显示鼠标的移动坐标值和按键状态等信息，该函数在 USB_Demo 函数里面被调用。

keybrd_data_process 函数，用于处理键盘数据，在液晶屏上面显示键盘的输入内容，最多支持 2017 个字符的输入显示。该函数在 USB_Demo 函数里面被调用。

usb_demo 函数，根据当前用户状态和 USB 接入设备的类型，执行不同的数据处理。该函数在 main 函数里面被调用。

usbh_check_enume_dead 函数，获取 USB 枚举状态，防止 USB 枚举失败导致的死机。该函数在 main 函数里面的 while 循环里面被调用。

usbh_hid_reconnect 函数，重新连接 HID 设备。当 USB 枚举失败后，会调用该函数，重新连接 HID 设备。

3. main.c 代码

下面是 main.c 的程序，具体如下：

```
int main(void)
{
    HAL_Init();                          /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                     /* 延时初始化 */
    usart_init(115200);                  /* 串口初始化为 115200 */
    led_init();                          /* 初始化 LED */
    lcd_init();                          /* 初始化 LCD */
    key_init();                          /* 初始化按键 */

    my_mem_init(SRAMIN);                 /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM);                /* 初始化内部 SRAMCCM 内存池 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USB MOUSE/KEYBOARD TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "USB Connecting...", RED);

    USBH_Init(&g_hUSBHost, USBH_UserProcess, 1);
    USBH_RegisterClass(&g_hUSBHost, USBH_HID_CLASS);
    USBH_Start(&g_hUSBHost);

    while (1)
    {
        USBH_Process(&g_hUSBHost);
        usb_demo(&g_hUSBHost);
        if(usbh_check_enume_dead(&g_hUSBHost))
```

```
        /* 检测 USB HOST 枚举是否死机了?死机了,则重新初始化 */
        {
            printf("Error! USB HID reconnect!\r\n");
            usbh_hid_reconnect();       /* 重连 */
        }
    }
}
```

main 函数，初始化各个外设后，通过 USBH_Init 等 3 个函数初始化并启动 USB 主机通信，然后进入死循环，不停的调用 USBH_Process、usb_demo 和判断 USB HOST 枚举是否死机，执行各种 USB 处理。

41.4 下载验证

将程序下载到开发板后，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 41.4.1 和图 41.4.2 所示：

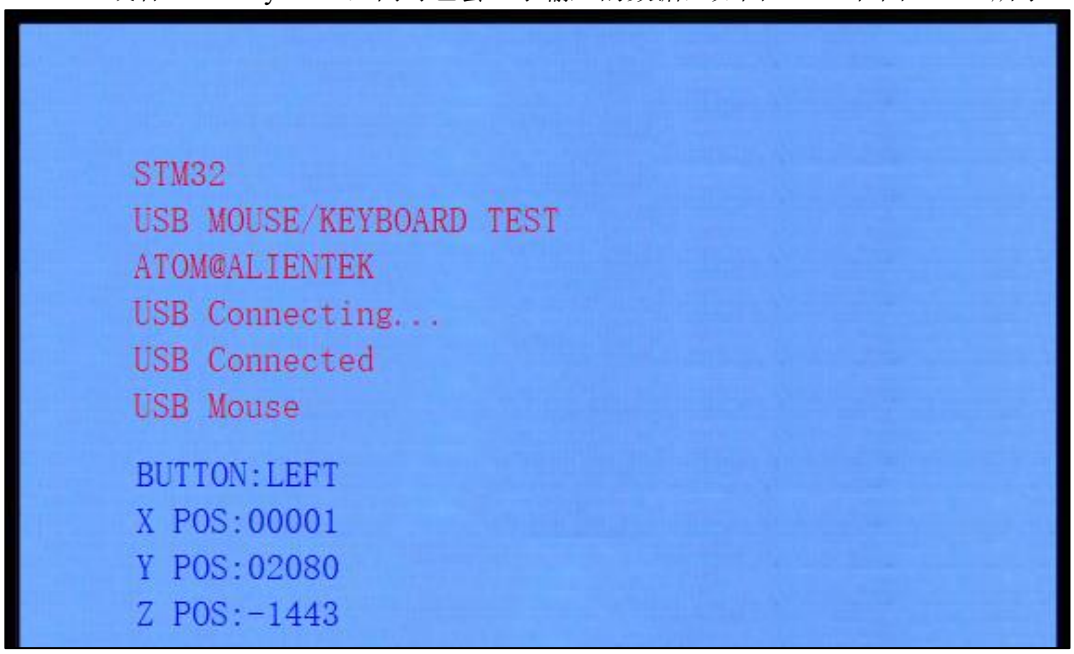


图 41.4.1 USB 鼠标测试

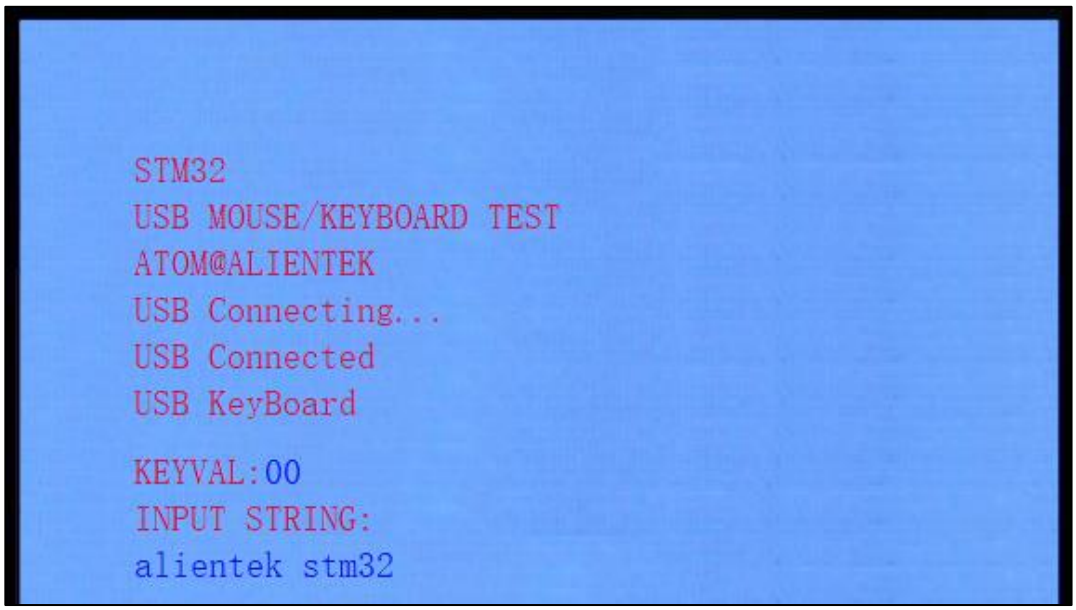


图 41.4.2 USB 键盘测试

其中，图 41.4.1 是 USB 鼠标测试界面，图 41.4.2 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。

第四十二章 USB U 盘(Host)实验

本章我们先介绍 FATFS 这个软件工具，利用它在 STM32 上实现类似电脑上的文件管理功能，通过 USB HOST 功能，实现读写 U 盘/读卡器等大容量 USB 存储设备。

本章分为如下几个小节：

42.1 FATFS 简介
42.2 U 盘简介
42.3 硬件设计
42.4 程序设计
42.5 下载验证

42.1 FATFS 简介

FATFS 是一个完全免费开源的 FAT/exFAT 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（ANSI C C89）编写，所以具有良好的硬件平台独立性，只需做简单的修改就可以移植到 8051、PIC、AVR、ARM、Z80、RX 等系列单片机上。它支持 FATl2、FATl6 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读 / 写，并特别对 8 位单片机和 16 位单片机做了优化。FATFS 的特点有：

- Windows/dos 系统兼容的 FAT/exFAT 文件系统
- 独立于硬件平台，方便跨硬件平台移植
- 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
  - ✧ 新版的 exFAT 文件系统，突破了原来 FAT32 对容量管理 32Gb 的上限，可支持更巨大容量的存储器

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 42.1.1 所示：

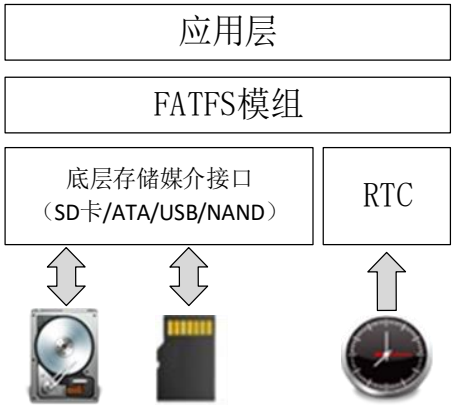


图 42.1.1 FATFS 层次结构图

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（diskI/O）和供给文件创建修改时间的实时时钟。

FATFS 的源码及英文详述，大家可以在：http://elm-chan.org/fsw/ff/00index_e.html 这个网站下载到，教程目前使用的版本为 R0.14b。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：documents 和 source。documents 里面主要是对 FATFS 的介绍，而 source 里面才是我们需要的源码。source 文件夹详情表，如表 42.1.1.1 所示：

文件名	作用简述	备注
diskio.h	FATFS 和 diskI/O 模块公用的包含文件	与硬件平台无关
ff.c	FATFS 模块	
ff.h	FATFS 和应用模块公用的包含文件	
ffconf.h	FATFS 模块配置文件，宏定义对应的功能，代码中都有说明，具体的配置范围可以见官方配置说明 http://elm-chan.org/fsw/ff/doc/config.html	
ffsystem.c	根据是否有操作系统来修改这个文件	
ffunicode.c	可选，根据 ffconf.h 的配置，进行 Unicode 编码转换	
diskio.c	FATFS 和 diskI/O 模块接口层文件，需要根据硬件修改这部分的代码	硬件平台相关代码

表 42.1.1.1 source 文件夹详情表

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） FF_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 TinyFATFS），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） FF_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） FF_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） FF_USE_MKFS。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） FF_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） FF_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） FF_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，同一个文件夹下的 ffunicode.c 根据这个宏选择对应的语言设置）。

8） FF_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） FF_VOLUMES。用于设置 FATFS 支持的逻辑设备数目，我们设置为 2，即支持 2 个设备。

10）FF_MAX_SS。扇区缓冲的最大值，一般设置为 512。

11）FF_FS_EXFAT。新版本增加的功能，使用 exFAT 文件系统，用于支持超过 32Gb 的超大存储。它们使用的是 exFAT 文件系统，使用它时必须要根据设置 FF_USE_LFN 这个参数的值以决定 exFATs 系统使用的内存来自堆栈还是静态数组。

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 http://elm-chan.org/fsw/ff/doc/config.html 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 5 个接口函数，如图 42.1.2 所示：

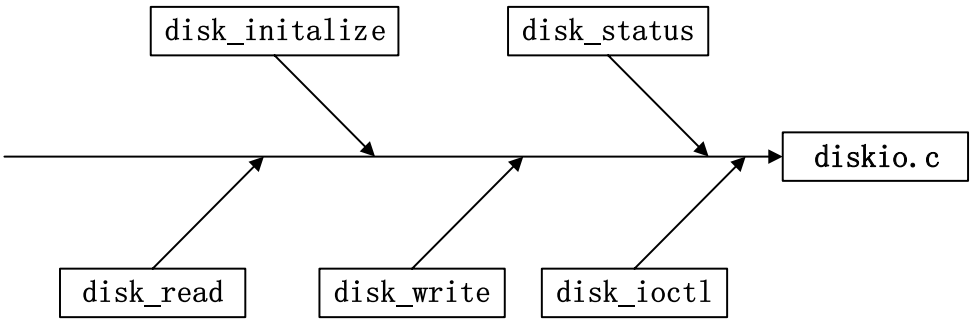


图 42.1.2 diskio 需要实现的函数

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 MDK5.34 编译器，其数据类型和 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 5 个，在 diskio.c 里面。

首先是 disk_initialize 函数，该函数介绍如表 42.1.2 所示：

函数名称	disk_initialize
函数原型	DSTATUS disk_initialize(BYTE Drive)
功能描述	初始化磁盘驱动器
函数参数	Drive：指定要初始化的逻辑驱动器号，即盘符，应当取值 0~9
返回值	函数返回一个磁盘状态作为结果，对于磁盘状态的细节信息，请参考 disk_status 函数
所在文件	ff.c
实例	disk_initialize(0);　　/* 初始化驱动器 0 */
注意事项	disk_initialize 函数初始化一个逻辑驱动器为读/写做准备，函数成功时，返回值的 STA_NOINIT 标志被清零； 应用程序不应调用此函数，否则卷上的 FAT 结构可能会损坏； 如果需要重新初始化文件系统，可使用 f_mount 函数； 在 FatFs 模块上卷注册处理时调用该函数可控制设备的改变； 此函数在 FatFs 挂在卷时调用，应用程序不应该在 FatFs 活动时使用此函数

表 42.1.2 disk_initialize 函数介绍

第二个函数是 disk_status 函数，该函数介绍如图 42.1.3 所示：

函数名称	disk_status
函数原型	DRESULT disk_status (BYTE Drive)
功能描述	返回当前磁盘驱动器的状态
函数参数	Drive：指定要确认的逻辑驱动器号，即盘符，应当取值 0~9
返回值	磁盘状态返回下列标志的组合，FatFs 只使用 STA_NOINIT 和 STA_PROTECTED STA_NOINIT：表明磁盘驱动未初始化，下面列出了产生该标志置位或清零的原因： 置位：系统复位，磁盘被移除和磁盘初始化函数失败 清零：磁盘初始化函数成功

	STA_NODISK：表明驱动器中没有设备，安装磁盘驱动器后总为 0 STA_PROTECTED：表明设备被写保护，不支持写保护的设备总为 0，当 STA_NODISK 置位时非法
所在文件	ff.c
实例	disk_status(0); /* 获取驱动器 0 的状态 */

表 42.1.3 disk_status 函数介绍

第三个函数是 disk_read 函数，该函数介绍如表 42.1.4 所示：

函数名称	disk_read
函数原型	DRESULT disk_read (BYTE Drive, BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount)
功能描述	从磁盘驱动器上读取扇区
函数参数	Drive：指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：指向存储读取数据字节数组的指针，需要为所读取字节数的大小，扇区统计的扇区大小是需要的 （注：FaFts 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理） SectorNumber：指定起始扇区的逻辑块（LBA）上的地址 SectorCount：指定要读取的扇区数，取值 1~128
返回值	RES_OK(0)：函数成功 RES_ERROR：读操作期间产生了任何错误且不能恢复它 RES_PARERR：非法参数 RES_NOTRDY：磁盘驱动器没有初始化
所在文件	ff.c

表 42.1.4 disk_read 函数介绍

第四个函数是 disk_write 函数，该函数介绍如表 42.1.5 所示：

函数名称	disk_write
函数原型	DRESULT disk_write (BYTE Drive, const BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount)
功能描述	向磁盘写入一个或多个扇区
函数参数	Drive：指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：指向要写入字节数组的指针 （注：FaFts 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理） SectorNumber：指定起始扇区的逻辑块（LBA）上的地址 SectorCount：指定要写入的扇区数，取值 1~128
返回值	RES_OK(0)：函数成功 RES_ERROR：写操作期间产生了任何错误且不能恢复它 RES_WRPER：媒体被写保护 RES_PARERR：非法参数 RES_NOTRDY：磁盘驱动器没有初始化
所在文件	ff.c
注意事项	只读配置中不需要此函数

表 42.1.5 disk_write 函数介绍

第五个函数是 disk_ioctl 函数，该函数介绍如表 42.1.6 所示：

函数名称	disk_ioctl
函数原型	DRESULT disk_ioctl (BYTE Drive, BYTE Command, void* Buffer)
功能描述	控制设备指定特性和除了读/写外的杂项功能
函数参数	Drive：指定逻辑驱动器号，即盘符，应当取值 0~9 Command：指定命令代码

	Buffer：指向参数缓冲区的指针，取决于命令代码，不使用时，指定一个 NULL 指针
返回值	RES_OK(0)：函数成功 RES_ERROR：写操作期间产生了任何错误且不能恢复它 RES_PARERR：非法参数 RES_NOTRDY：磁盘驱动器没有初始化
所在文件	ff.c
注意事项	CTRL_SYNC：确保磁盘驱动器已经完成了写处理，当磁盘 I/O 有一个写回缓存，立即刷新原扇区，只读配置下不适用此命令 GET_SECTOR_SIZE：返回磁盘的扇区大小，只用于 f_mkfs() GET_SECTOR_COUNT：返回可利用的扇区数，_MAX_SS ≥ 1024 时可用 GET_BLOCK_SIZE：获得擦除块大小，只用于 f_mkfs() CTRL_ERASE_SECTOR：强制擦除一块的扇区，_USE_ERASE ＞ 0 时可用

表 42.1.6 disk_ioctl 函数介绍

以上五个函数，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。

42.2 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。

STM32F407 的 USB OTG HS 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 →1，STM32CubeF4 固件包 → STM32Cube_FW_F4_V1.26.0 → Projects → STM324xG_EVAL → Applications → USB_Host →MSC_Standalone。本实验，我们就要移植该例程到开发板上，以通过 STM32F407 的 USB HOST 接口，读写 U 盘或 SD 卡读卡器等设备。

42.3 硬件设计

1. 例程功能

本实验代码，开机后，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。

LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
LED1 – PE1
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）USB_HOST 接口(D-/D+连接在 PA11/PA12 上)

3. 原理图

本开发板的 USB HOST 接口采用的是贴片 USB 母座，它和 USB SLAVE 的 Type-C 接头是共用 USB_DM 和 USB_DP 信号的，所以 USB HOST 和 USB SLAVE 功能不能同时使用。USB HOST 和 STM32 的连接原理图，如下图所示：

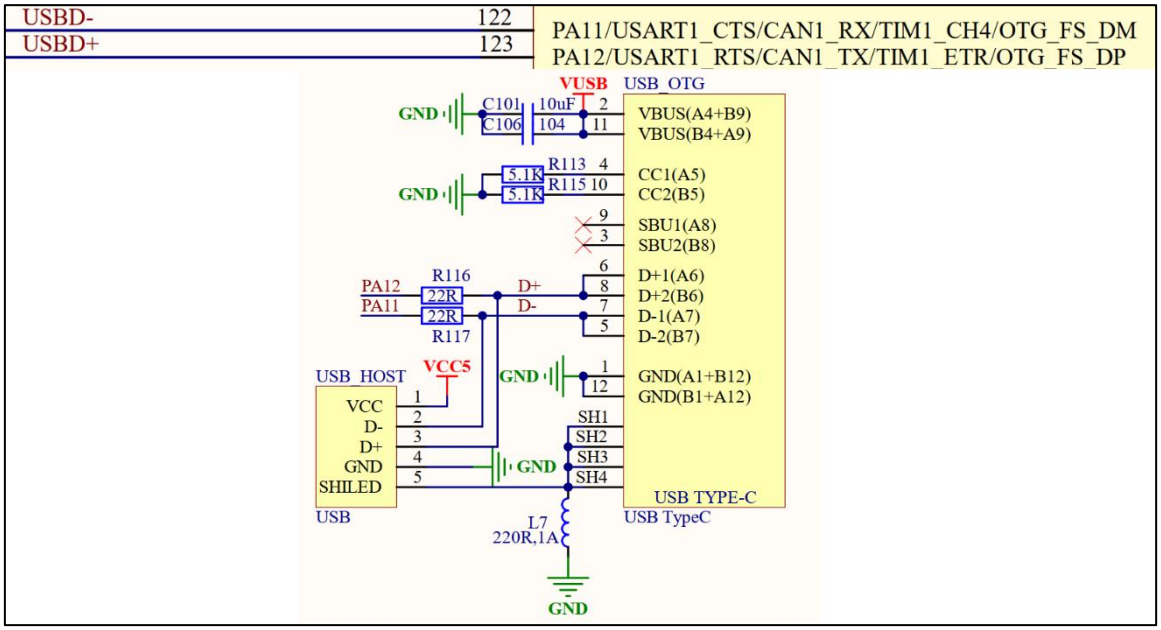


图 42.3.1 USB 母座与 STM32 的连接电路图

从上图可以看出，USB 座是直接连接到 STM32F407 上面的，所以硬件上不需要我们做什么操作，可直接使用。

需要注意的是：这个 USB 母座（USB_HOST）和 Type-C 接口是共用 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，数据线要用 USB_UART 接口连接到电脑。

下面，看看开发板上的 USB 母座（USB_HOST）的位置，请将 U 盘插入这个接口上。

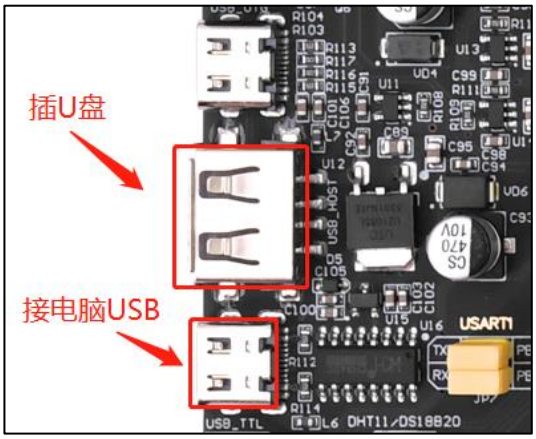


图 42.3.2 U 盘与开发板的连接示意图

42.4 程序设计

42.4.1 程序流程图

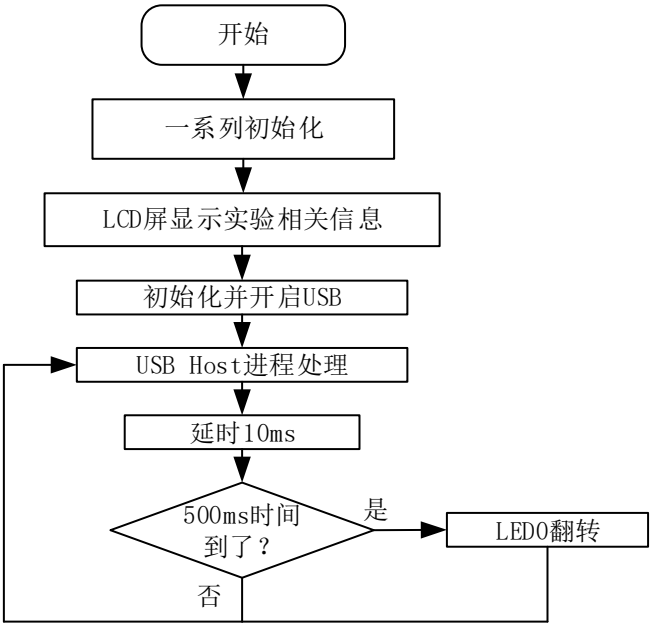


图 42.4.1.1 USB U 盘(Host)实验程序流程图

42.4.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。

本实验，在上一章的 USB 鼠标键盘实验上修改，代码移植自 ST 官方例程：STM32Cube_FW_F4_V1.26.0\Projects\STM324xG_EVAL\Applications\USB_Host\MSC_Standalone。有了这个官方例程做指引，我们就知道具体需要哪些文件，从而实现本实验。

FATFS 的驱动为一个硬件独立的组件，因此我们把 FATFS 的移植代码放到“Middlewares”文件夹下。

在“Middlewares”下新建一个 FATFS 的文件夹，然后将 ff14b.zip 程序包解压到该文件夹下。同时，我们在 FATFS 文件夹里面新建一个 exfuns 的文件夹，用于存放我们针对 FATFS 做的一些扩展代码（这个我们后面再讲）。操作结果如图 42.4.2.1 所示：

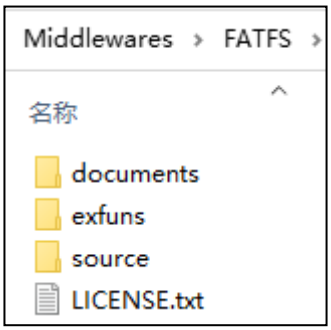


图 42.4.2.1 FATFS 文件夹子目录

本实验的具体移植步骤，我们这里就不一一介绍了，最终移植好之后的工程分组截图，如图 42.4.2.2 所示：

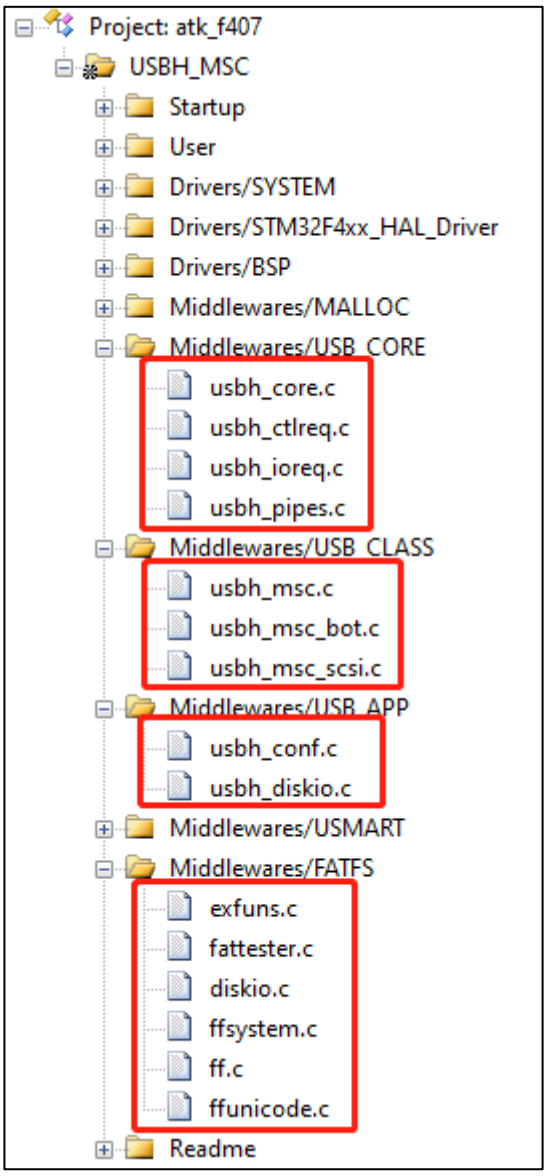


图 42.4.2.2 USB U 盘(Host)实验工程分组

上图工程分组中的 Middlewares/USB_CORE 和 Middlewares/USB_CLASS 分组下的.c 文件，直接拷贝 ST 官方 USB HOST 库，我们重点要修改的是 USB_APP 文件夹下面的代码和 FATFS 文件夹下面的代码，详细的源码请参考光盘本实验对应的源码。

1. FATFS 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。

diskio.c/.h 为我们提供了规定好的底层驱动接口的返回值。这个函数需要使用到我们的硬件接口，所以需要把使用到的硬件驱动的头文件包进来。

```
#include "./MALLOC/malloc.h"
#include "./FATFS/source/diskio.h"
#include "./BSP/NORFLASH/norflash.h"
#include "usbh_diskio.h"
```

接下来我们来对这几个接口进行补充实现。

```
#define EX_FLASH    0       /* 外部 spi flash,卷标为 0 */
#define USB_DISK    1       /* U 盘,卷标为 1 */

/**
 * 对于 25Q128 FLASH 芯片, 我们规定前 12M 给 FATFS 使用, 12M 以后
```

```
 * 紧跟字库, 3 个字库 + UNIGBK.BIN, 总大小 3.09M, 共占用 15.09M
 * 15.09M 以后的存储空间大家可以随便使用.
 */
#define SPI_FLASH_SECTOR_SIZE   512
#define SPI_FLASH_SECTOR_COUNT  12 * 1024 * 2  /* 25Q128,前 12M 字节给 FATFS 占用 */
#define SPI_FLASH_BLOCK_SIZE    8              /* 每个 BLOCK 有 8 个扇区 */
#define SPI_FLASH_FATFS_BASE    0          /* FATFS 在外部 FLASH 的起始地址从 0 开始 */
```

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，我们就通过 switch 来判断，到底要操作 U 盘，还是 SPI FLASH，然后，分别执行对应设备的相关操作。以此实现磁盘编号和磁盘的关联。

(1) disk_initialize 函数

要使用 FAFTS 管理，首先要对它进行初始化，所以先看磁盘的初始化函数，其声明如下：

```
DSTATUS disk_initialize ( BYTE pdrv)
```

- **函数描述：**

  初始化指定编号的磁盘，磁盘所指定的存储区。使用每个磁盘前进行初始化，那在代码中直接根据编号调用硬件的初始化接口即可，这样也能保证代码的扩展性，硬件的顺序可以根据自己的喜好定义。

- **函数形参：**

  形参 1 是 FATFS 管理的磁盘编号 pdrv：磁盘编号 0~9，我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。

  代码实现如下：

```
#define EX_FLASH    0       /* 外部 spi flash, 卷标为 0 */
#define USB_DISK    1       /* U 盘, 卷标为 1 */

/**
 * @brief       初始化磁盘
 * @param       pdrv : 磁盘编号 0~9
 * @retval      无
 */
DSTATUS disk_initialize (
    BYTE pdrv    /* Physical drive nmuber to identify the drive */
)
{
    uint8_t res = 0;

    switch (pdrv)
    {
        case EX_FLASH:       /* 外部 flash */
            norflash_init(); /* 外部 flash 初始化 */
            break;
        case USB_DISK:               /* U 盘 */
            res = USBH_initialize(); /* U 盘初始化 */
            break;
        default:
            res = 1;
            break;
    }

    if (res)
    {
        return  STA_NOINIT;
    }
    else
    {
        return 0;  /* 初始化成功*/
    }
}
```

- **函数返回值：**

  DSTATUS 枚举类型的值，FATFS 规定了自己的返回值来管理各接口函数的操作结果,方便

后续函数的操作和判断，我们看看它的定义：

```
/* Status of Disk Functions */
typedef BYTE    DSTATUS;
/* Disk Status Bits (DSTATUS) */
#define STA_NOINIT      0x01    /* Drive not initialized */
#define STA_NODISK      0x02    /* No medium in the drive */
#define STA_PROTECT     0x04    /* Write protected */
```

定义时也写出了各个参数的含义，根据 ff.c 中的调用实例可知操作返回 0 才是正常的状态，其它情况发生的话就需要结合硬件进行分析了。

(2) disk_status 函数

便于我们知道当前磁盘驱动器的状态，所以 FATFS 也有 disk_status 函数提供给我们使用，其声明如下：

```
DSTATUS disk_status (BYTE pdrv)
```

- **函数描述：**
  返回当前磁盘驱动器的状态。
- **函数形参：**
  FATFS 管理的磁盘编号 pdrv：磁盘编号 0~9。我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。为了简单测试，所以我们这里没有加入硬件状态的判断，测试代码很简单，这里就不贴出来了。
- **函数返回值：**
  直接返回 RES_OK。

(3) disk_read 函数

disk_read 实现直接从硬件接口读取数据，这个函数接口是给 FATFS 的其它读操作接口函数调用，其声明如下：

```
DRESULT disk_read (BYTE pdrv, BYTE *buff, DWORD sector, UINT count)
```

- **函数描述：**
  从磁盘驱动器上读取扇区数据。
- **函数形参：**
  形参 1：是 FATFS 管理的磁盘编号 pdrv：磁盘编号 0~9，我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。
  形参 2：buff 指向要保存数据的内存区域指针，为字节类型。
  形参 3：sector 为实际物理操作时要访问的扇区地址。
  形参 4：count 为本次要读取的数据量，最长为 unsigned int，读到的数量为字节数。

代码实现如下，同样要根据我们定义的设备标号，在 switch-case 中添加对应硬件的驱动：

```
DRESULT disk_read (
    BYTE pdrv,      /* Physical drive nmuber to identify the drive */
    BYTE *buff,     /* Data buffer to store read data */
    DWORD sector,   /* Sector address in LBA */
    UINT count      /* Number of sectors to read */)
{
    uint8_t res = 0;

    if (!count)
    {
        return RES_PARERR;   /* count 不能等于 0，否则返回参数错误 */
    }


    switch (pdrv)
    {
        case EX_FLASH:       /* 外部 flash */
            for (; count > 0; count--)
            {
                norflash_read(buff, SPI_FLASH_FATFS_BASE + sector *
                                    SPI_FLASH_SECTOR_SIZE, SPI_FLASH_SECTOR_SIZE);
                sector++;
                buff += SPI_FLASH_SECTOR_SIZE;
            }
```

```
            res = 0;
            break;

        case USB_DISK:          /* U盘 */
            res = USBH_read(buff, sector, count);
            break;

        default:
            res = 1;
            break;
    }
    /* 处理返回值，将返回值转成 ff.c 的返回值 */
    if (res == 0x00)
    {
        return RES_OK;
    }
    else
    {
        return RES_ERROR;
    }
}
```

- **函数返回值：**

DRESULT 为枚举类型，diskio.h 中有其定义，根据返回值的含义确认操作结果即可。在这里也把该结果展示出来，如下所示：

```
/* Results of Disk Functions */
typedef enum
{
    RES_OK = 0,         /* 0: 操作成功 */
    RES_ERROR,          /* 1: 读/写错误 */
    RES_WRPRT,          /* 2: 写保护状态 */
    RES_NOTRDY,         /* 3: 设备忙 */
    RES_PARERR          /* 4: 其它情形 */
} DRESULT;
```

根据返回值的含义确认操作结果即可。

(4) disk_write 函数

disk_write 实现直接从硬件接口写入数据，这个函数接口给 FATFS 的其它写操作接口函数调用，其声明如下：

```
DRESULT disk_write ( BYTE pdrv, const BYTE *buff, DWORD sector, UINT count)
```

- **函数描述：**

初始化指定编号的磁盘，磁盘所指定的存储区。

- **函数形参：**

形参 1：是 FATFS 管理的磁盘编号 pdrv：磁盘编号 0~9，我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。

形参 2：buff 指向要保存数据的内存区域指针，为字节类型。

形参 3：sector 为实际物理操作时要访问的扇区地址。

形参 4：count 为本次要读取的数据量，最长为 unsigned int，读到的数量为字节数。

代码实现如下，同样要根据我们定义的设备标号，在 switch-case 中添加对应硬件的驱动：

```
DRESULT disk_write (
    BYTE pdrv,              /* Physical drive nmuber to identify the drive */
    const BYTE *buff,       /* Data to be written */
    DWORD sector,           /* Sector address in LBA */
    UINT count              /* Number of sectors to write */
)
{
    uint8_t res = 0;

    if (!count)
    {
        return RES_PARERR;   /* count 不能等于 0，否则返回参数错误 */
```

```
    }

    switch (pdrv)
    {
        case EX_FLASH:        /* 外部 flash */
            for (; count > 0; count--)
            {
                norflash_write((uint8_t *)buff, SPI_FLASH_FATFS_BASE + sector *
                                SPI_FLASH_SECTOR_SIZE, SPI_FLASH_SECTOR_SIZE);
                sector++;
                buff += SPI_FLASH_SECTOR_SIZE;
            }

            res = 0;
            break;

        case USB_DISK:        /* U 盘 */
            res = USBH_write(buff, sector, count);
            break;

        default:
            res = 1;
            break;
    }
    /* 处理返回值，将返回值转成 ff.c 的返回值 */
    if (res == 0x00)
    {
        return RES_OK;
    }
    else
    {
        return RES_ERROR;
    }
}
```

- **函数返回值：**

DRESULT 为枚举类型，diskio.h 中有其定义，编写读函数时已经介绍了，注意要把返回值转成这个枚举类型的参数。

(5) disk_ioctl 函数

disk_ioctl 实现一些控制命令，这个接口为 FATFS 提供一些硬件操作信息，其声明如下：

```
DRESULT disk_ioctl (BYTE pdrv, BYTE cmd, void *buff)
```

- **函数描述：**

控制设备指定特性和除了读/写外得杂项功能。

- **函数形参：**

形参 1：FATFS 管理的磁盘编号 pdrv :磁盘编号 0~9，我们配置 FF_VOLUMES 为 2 来支持两个磁盘，因此可选值为 0 和 1。

形参 2：cmd 是 FATFS 定义好的一些宏，用于访问硬盘设备的一些状态。我们实现几个简单的操作接口，用于获取磁盘容量这些基础信息（diskio.h 中已经定义好了），为方便，我们先只实现几个标准的应用接口，关于 SDIO/MMC 的一些扩展命令我们再根据需要进行支持。

```
/* Command code for disk_ioctrl fucntion */
/* Generic command (Used by FatFs) */
#define CTRL_SYNC           0    /* 完成挂起的写入过程(当 FF_FS_READONLY == 0) */
#define GET_SECTOR_COUNT    1    /* 获取磁盘扇区数(当 FF_USE_MKFS == 1) */
#define GET_SECTOR_SIZE     2    /* 获取磁盘存储空间大小(当 FF_MAX_SS != FF_MIN_SS) */
#define GET_BLOCK_SIZE      3    /* 每个扇区块的大小(当 FF_USE_MKFS == 1) */
```

下面是从 http://elm-chan.org/fsw/ff/doc/dioctl.html 得到的参数实现效果，我们也可以参考原有的 disk_ioctl 的实现来理解这几个参数。

命令	说明
CTRL_SYNC	确保设备已完成挂起的写入过程。如果磁盘 I/O 层或存储设备具有回写缓存，则必须立即将缓存数据提交到介质。如果对介质的每个写入操作都正常完成，则此命令无任何动作。
GET_SECTOR_COUNT	返回对应标号的硬盘的可用扇区数。此命令由 f_mkfs\f_fdisk 函数用于确定要创建的卷/分区的大小。使用时需要设置 FF_USE_MKFS 为 1。
GET_SECTOR_SIZE	将用于读/写函数的扇区大小检索到 buff 指向的 WORD 变量中。有效扇区大小为 512、1024、2048 和 4096。只有在 FF_MAX_SS>FF_MIN_SS 执行此命令时。当 FF_MAX_SS=FF_MIN_SS 时，将永远不会使用此命令，并且读/写函数必须仅在 FF_MAX_SSbytes/扇区中工作。
GET_BLOCK_SIZE	将扇区单位中闪存介质的块大小以 DWORD 指针存到 buff 中。允许的值为 1 到 32768。如果擦除块大小未知或非闪存介质，则返回 1。此命令仅由 f_mkfs 函数使用，并尝试对齐擦除块边界上的数据区域。使用时需要设置 FF_USE_MKFS 为 1。

形参 3：buff 为 void 形指针，根据命令的格式和需要，我们把对应的值转成对应的形式传给它。

参考原有的 disk_ioctl 的实现，我们的函数实现如下。

```
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 == USB_DISK)       /* U盘 */
    {
        res = USBH_ioctl(cmd, buff);
    }
    else if (pdrv == EX_FLASH)  /* 外部FLASH */
    {
        switch (cmd)
        {
            case CTRL_SYNC:
                res = RES_OK;
                break;

            case GET_SECTOR_SIZE:
                *(WORD *)buff = SPI_FLASH_SECTOR_SIZE;
                res = RES_OK;
                break;

            case GET_BLOCK_SIZE:
                *(WORD *)buff = SPI_FLASH_BLOCK_SIZE;
                res = RES_OK;
                break;

            case GET_SECTOR_COUNT:
                *(DWORD *)buff = SPI_FLASH_SECTOR_COUNT;
                res = RES_OK;
                break;

            default:
                res = RES_PARERR;
                break;
        }
    }
```

```
    else
    {
        res = RES_ERROR;    /* 其他的不支持 */
    }
    return res;
}
```

- **函数返回值：**

DRESULT 为枚举类型，diskio.h 中有其定义，编写读函数时已经介绍了，注意要把返回值转成这个枚举类型的参数。

以上实现了我们在 42.1 节提到的 5 个函数，ff.c 中需要实现 get_fattime (void)，同时因为在 ffconf.h 里面设置对长文件名的支持为方法 3，所以必须在 ffsystem.c 中实现 get_fattime、ff_memalloc 和 ff_memfree 这三个函数。这部分比较简单，直接参考我们修改后的 ffsystem.c 的源码。

至此，我们已经可以直接使用 FATFS 下的 ff.c 下的 f_mount 的接口挂载磁盘，然后使用类似标准 C 的文件操作函数，就可以实现文件操作。但 f_mount 还需要一些文件操作的内存，为了方便操作，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针对 FATFS 的扩展代码，如刚才提到的 FATFS 相关函数的内存申请方法等。

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，大家可以直接使用本例程源码，我们将 exfuns.c/.h 和 fattester.c/.h 加入 FATFS 组下的 exfuns 文件下，直接使用即可。

(6) exfuns_init 函数

我们在使用文件操作前，需要用 f_mount 函数挂载磁盘，我们在挂载 SD 卡前需要一些文件系统的内存，为了方便管理，我们定义一个全局的 fs[FF_VOLUMES]指针，定义的数组是我们要管理多个磁盘，而 f_mount 也需要一个 FATFS 类型的指针，定义如下：

```
/* 逻辑磁盘工作区(在调用任何 FATFS 相关函数之前,必须先给 fs 申请内存) */
FATFS *fs[FF_VOLUMES];
```

接下来只要用内存管理部分的知识来实现对 fs 指针的内存申请即可。

```
/**
 * @brief       为 exfuns 申请内存
 * @param       无
 * @retval      0, 成功; 1, 失败.
 */
uint8_t exfuns_init(void)
{
    uint8_t i;
    uint8_t res = 0;

    for (i = 0; i < FF_VOLUMES; i++)
    {
        /* 为磁盘 i 工作区申请内存 */
        fs[i] = (FATFS *)mymalloc(SRAMIN, sizeof(FATFS));
        if (!fs[i])
        {
            break;
        }
    }

#if USE_FATTESTER == 1      /* 如果使能了文件系统测试 */
    res = mf_init();        /* 初始化文件系统测试(申请内存) */
#endif

    if (i == FF_VOLUMES && res == 0)
    {
        return 0;    /* 申请有一个失败,即失败. */
    }
```

```
    else
    {
        return 1;
    }
}
```

其它的函数我们暂时没有使用到，这里先不介绍了，大家使用时查阅源码即可。

2. USB 驱动代码

usbh_conf.c 提供了 USB 主机库的回调及 MSP 初始化函数，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。该.c 文件我们重点介绍 3 个函数，首先是初始化 PCD MSP 函数，其定义如下：

```
/**
 * @brief       初始化 PCD MSP
 * @param       hpcd:PCD 结构体指针
 * @retval      无
 */
void HAL_PCD_MspInit(PCD_HandleTypeDef * hpcd)
{
    GPIO_InitTypeDef gpio_init_struct = { 0 };

    if (hpcd->Instance == USB_OTG_FS)
    {
        __HAL_RCC_GPIOA_CLK_ENABLE();                           /* 使能 GPIOA 时钟 */

        gpio_init_struct.Pin = GPIO_PIN_11 | GPIO_PIN_12;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用 */
        gpio_init_struct.Pull = GPIO_NOPULL;                    /* 浮空 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;     /* 高速 */
        gpio_init_struct.Alternate = GPIO_AF10_OTG_FS;          /* 复用为 OTG_FS */
        HAL_GPIO_Init(GPIOA, &gpio_init_struct);/* 初始化 PA11 和 PA12 引脚 */

        __HAL_RCC_USB_OTG_FS_CLK_ENABLE();          /* 使能 OTG FS 时钟 */
        HAL_NVIC_SetPriority(OTG_FS_IRQn, 1, 0);/* 优先级设置为抢占 1,子优先级 0 */
        HAL_NVIC_EnableIRQ(OTG_FS_IRQn);            /* 使能 OTG FS 中断 */
    }
    else if (hpcd->Instance == USB_OTG_HS)
    {
        /* USB_OTG_HS 本例程没用到,故不做处理 */
    }
}
```

HAL_HCD_MspInit 函数，用于使能 USB 时钟，选择内核时钟源，初始化 IO 口，设置中断等。该函数在 HAL_HCD_Init 函数里面被调用。

接下来介绍的是 USB OTG 中断服务函数，其定义如下：

```
/**
 * @brief       USB OTG 中断服务函数
 *   @note      处理所有 USB 中断
 * @param       无
 * @retval      无
 */
void OTG_FS_IRQHandler(void)
{
    HAL_HCD_IRQHandler(&hhcd);
}
```

OTG_FS_IRQHandler 函数，是 USB 的中断服务函数，通过调用 HAL_HCD_IRQHandler 函数，实现对 USB 各种事务的处理。

最后介绍的是 USBH 底层初始化函数，其定义如下：

```
/**
 * @brief       USBH 底层初始化函数
```

```
 * @param       phost      : USBH 句柄指针
 * @retval      USB 状态
 *   @arg       USBH_OK(0)   , 正常;
 *   @arg       USBH_BUSY(1) , 忙;
 *   @arg       其他         , 失败;
 */
USBH_StatusTypeDef USBH_LL_Init(USBH_HandleTypeDef *phost)
{
    /* Init USB_IP */
    if (phost->id == HOST_FS)
    {
        /* Link the driver to the stack. */
        g_hhcd_USB_OTG_FS.Instance = USB_OTG_FS;     /* 使用 USB_OTG */
        g_hhcd_USB_OTG_FS.Init.Host_channels = 8;    /* 主机通道数为 8 个 */
        g_hhcd_USB_OTG_FS.Init.speed = HCD_SPEED_FULL;  /* USB 全速 12M */
        g_hhcd_USB_OTG_FS.Init.dma_enable = DISABLE;     /* 不使用 DMA */
        g_hhcd_USB_OTG_FS.Init.phy_itface = HCD_PHY_EMBEDDED;   /* 使能内部 PHY */
        g_hhcd_USB_OTG_FS.Init.Sof_enable = DISABLE;             /* 禁止 SOF 中断 */
        g_hhcd_USB_OTG_FS.Init.vbus_sensing_enable = 0;      /* 不使能 VBUS 检测 */
        g_hhcd_USB_OTG_FS.Init.lpm_enable = 0;               /* 不使能连接电源管理 */
        g_hhcd_USB_OTG_FS.Init.low_power_enable = 0;         /* 不使能低功耗模式 */

        g_hhcd_USB_OTG_FS.pData = phost; /* g_hhcd_USB_OTG_FS 的 pData 指向 phost */
        /* phost 的 pData 指向 g_hhcd_USB_OTG_FS */
        phost->pData = &g_hhcd_USB_OTG_FS;

        if (HAL_HCD_Init(&g_hhcd_USB_OTG_FS) != HAL_OK)      /* 初始化 LL 驱动 */
        {
            printf("USB Init ERROR \r\n");
        }
        else
        {
            printf("USB Init OK \r\n");
        }
        USBH_LL_SetTimer(phost, HAL_HCD_GetCurrentFrame(&g_hhcd_USB_OTG_FS));
    }
    return USBH_OK;
}
```

USBH_LL_Init 函数，用于初始化 USB 底层设置，因为我们定义的是：USE_USB_FS，因此会设置 USB OTG 使用 USB_OTG_FS，然后完成各种设置，比如，使用内部 PHY，使用全速模式，不使能 VBUS 检测等，详见以上代码。该函数在 USBH_Init 函数里面被调用。

usbh_diskio.c 提供 U 盘和 FATFS 文件系统之间的输入/输出接口函数，这里总共有 5 个函数，首先介绍的是初始化 USBH 函数，其定义如下：

```
/**
 * @brief       初始化 USBH
 * @param       无
 * @retval      无
 */
DSTATUS USBH_initialize(void)
{
    return RES_OK;
}
```

USBH_initialize 函数，用于初始化 U 盘，我们不需要做任何事情，直接返回 OK 即可。

下面介绍的是获取 U 盘状态函数，其定义如下：

```
/**
 * @brief       获取 U 盘状态
 * @param       无
 * @retval      无
 */
DSTATUS USBH_status(void)
{
```

```
    DRESULT res = RES_ERROR;
    MSC_HandleTypeDef *MSC_Handle = hUSBHost.pActiveClass->pData;
    if(USBH_MSC_UnitIsReady(&g_hUSBHost, MSC_Handle->current_lun))
    {
        printf("U 盘状态查询成功\r\n");
        res=RES_OK;
    }else
    {
        printf("U 盘状态查询失败\r\n");
        res=RES_ERROR;
    }
    return res;
}
```

USBH_status 函数，用于获取 U 盘状态，本实验暂时用不到。

下面介绍的是 U 盘读扇区操作函数，其定义如下：

```
/**
 * @brief       U 盘读扇区操作
 * @param       buff    : 数据缓冲首地址
 * @param       sector  : 扇区地址
 * @param       count   : 需要读取的扇区数
 * @retval      执行结果(详见 DRESULT 定义)
 */
DRESULT USBH_read(BYTE *buff, DWORD sector, UINT count)
{
    DRESULT res = RES_ERROR;
    MSC_LUNTypeDef info;
    MSC_HandleTypeDef *MSC_Handle=hUSBHost.pActiveClass->pData;

    if (USBH_MSC_Read(&hUSBHost, MSC_Handle->current_lun, sector, buff, count)
                      == USBH_OK)
    {
        res = RES_OK;
    }
    else
    {
        printf("U 盘读取失败\r\n");
        USBH_MSC_GetLUNInfo(&hUSBHost, MSC_Handle->current_lun, &info);
        switch (info.sense.asc)
        {
            case SCSI_ASC_LOGICAL_UNIT_NOT_READY:
            case SCSI_ASC_MEDIUM_NOT_PRESENT:
            case SCSI_ASC_NOT_READY_TO_READY_CHANGE:
                USBH_ErrLog("USB Disk is not ready!");
                res = RES_NOTRDY;
                break;
            default:
                res = RES_ERROR;
                break;
        }
    }
    return res;
}
```

USBH_read 函数，用于从 U 盘指定位置，读取指定长度的数据。

下面介绍的是 U 盘写扇区操作函数，其定义如下：

```
/**
 * @brief       U 盘写扇区操作
 * @param       buff    : 数据缓冲首地址
 * @param       sector  : 扇区地址
 * @param       count   : 需要写入的扇区数
 * @retval      执行结果(详见 DRESULT 定义)
 */
DRESULT USBH_write(const BYTE *buff, DWORD sector, UINT count)
{
```

```
    DRESULT res = RES_ERROR;
    MSC_LUNTypeDef info;
    MSC_HandleTypeDef *MSC_Handle=hUSBHost.pActiveClass->pData;
    if(USBH_MSC_Write(&hUSBHost, MSC_Handle->current_lun, sector,
                                                (BYTE *)buff, count) == USBH_OK)
    {
        res=RES_OK;
    }else
    {
        printf("U 盘写入失败\r\n");
        USBH_MSC_GetLUNInfo(&hUSBHost, MSC_Handle->current_lun, &info);
        switch (info.sense.asc)
        {
            case SCSI_ASC_WRITE_PROTECTED:
                USBH_ErrLog("USB Disk is Write protected!");
                res = RES_WRPRT;
                break;
            case SCSI_ASC_LOGICAL_UNIT_NOT_READY:
            case SCSI_ASC_MEDIUM_NOT_PRESENT:
            case SCSI_ASC_NOT_READY_TO_READY_CHANGE:
                USBH_ErrLog("USB Disk is not ready!");
                res = RES_NOTRDY;
                break;
            default:
                res = RES_ERROR;
                break;
        }
    }
    return res;
}
```

USBH_write 函数，用于往 U 盘指定位置，写入指定长度的数据。

最后介绍的是 U 盘 IO 控制操作函数，其定义如下：

```
/**
 * @brief       U 盘 IO 控制操作
 * @param       cmd    : 控制命令
 * @param       buff   : 控制数据
 * @retval      执行结果(详见 DRESULT 定义)
 */
DRESULT USBH_ioctl(BYTE cmd,void *buff)
{
    DRESULT res = RES_ERROR;
    MSC_LUNTypeDef info;
    MSC_HandleTypeDef *MSC_Handle = hUSBHost.pActiveClass->pData;
    switch(cmd)
    {
        case CTRL_SYNC:
            res = RES_OK;
            break;
        case GET_SECTOR_COUNT : /* 获取扇区数量 */
            if(USBH_MSC_GetLUNInfo(&hUSBHost, MSC_Handle->current_lun,
                                   &info) == USBH_OK)
            {
                *(DWORD*)buff = info.capacity.block_nbr;
                res = RES_OK;
                printf("扇区数量:%d\r\n", info.capacity.block_nbr);
            }
            else
            {
                res = RES_ERROR;
            }
            break;
        case GET_SECTOR_SIZE : /* 获取扇区大小 */
            if(USBH_MSC_GetLUNInfo(&hUSBHost,MSC_Handle->current_lun, &info)
                                   == USBH_OK)
            {
```

```
            *(DWORD*)buff = info.capacity.block_size;
            res = RES_OK;
            printf("扇区大小:%d\r\n", info.capacity.block_size);
        }else
        {
            res = RES_ERROR;
        }
        break;
    case GET_BLOCK_SIZE :   /* 获取一个扇区里面擦除块的大小 */
        if(USBH_MSC_GetLUNInfo(&hUSBHost, MSC_Handle->current_lun, &info)
                               ==USBH_OK)
        {
            *(DWORD*)buff = info.capacity.block_size / USB_DEFAULT_BLOCK_SIZE;
            printf("每个扇区擦除块:%d\r\n",
                           info.capacity.block_size / USB_DEFAULT_BLOCK_SIZE);
            res = RES_OK;
        }
        else
        {
            res = RES_ERROR;
        }
        break;
    default:
        res = RES_PARERR;
        break;
  }
  return res;
}
```

USBH_ioctl 函数，可以用于获取 U 盘扇区数量、扇区大小和块大小等信息。

我们将这 5 个函数在 diskio.c 里面和 FATFS 完成对接，同时需要设置 FATFS 支持 2 个磁盘（SPI FLASH 和 U 盘），需要在 ffconf.h 文件里面将 FF_VOLUMES 的宏定义值改成 2，以支持 FATFS 操作 U 盘。

3. main.c 代码

下面是 main.c 的程序，具体如下：

```
USBH_HandleTypeDef g_hUSBHost;    /* USB Host 处理结构体 */

static void USBH_UserProcess(USBH_HandleTypeDef *phost, uint8_t id)
{
  uint32_t total, free;
  uint8_t res = 0;
  printf("id:%d\r\n", id);

  switch (id)
  {
    case HOST_USER_SELECT_CONFIGURATION:
      break;

    case HOST_USER_DISCONNECTION:
      f_mount(0, "1:", 1);          /* 卸载 U 盘 */
      lcd_show_string(30, 140, 200, 16, 16, "U Disk connecting...", BLUE);
      lcd_fill(30, 160, 239, 220, WHITE);
      break;

    case HOST_USER_CLASS_ACTIVE:
      lcd_show_string(30, 140, 200, 16, 16, "U Disk connect successful!",
                                                              BLUE);
      f_mount(fs[1], "1:", 1); /* 重新挂载 U 盘 */
      res = exfuns_get_free("1:", &total, &free);

      if (res == 0)
      {
        lcd_show_string(30, 160, 200, 16, 16, "FATFS OK!", BLUE);
```

```
            lcd_show_string(30, 180, 200, 16, 16, "U Disk Total Size:     MB",
                                                                    BLUE);
            lcd_show_string(30, 200, 200, 16, 16, "U Disk  Free Size:     MB",
                                                                    BLUE);
            /* 显示 U 盘总容量 MB */
            lcd_show_num(174, 180, total >> 10, 5, 16, BLUE);
            lcd_show_num(174, 200, free >> 10, 5, 16, BLUE);
        }
        else
        {
            lcd_show_string(30, 160, 200, 16, 16, "FATFS ERROR!", BLUE);
            printf("U 盘存储空间获取失败\r\n");
        }
        break;

    case HOST_USER_CONNECTION:
        break;

    default:
        break;
    }
}

int main(void)
{
    uint8_t t = 0;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    usmart_dev.init(84);                    /* USMART 初始化 */
    led_init();                             /* 初始化 LED */
    lcd_init();                             /* 初始化 LCD */

    my_mem_init(SRAMIN);                    /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM);                   /* 初始化内部 SRAMCCM 内存池 */

    exfuns_init();                          /* 为 fatfs 相关变量申请内存 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "USB U Disk TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    /* 提示 USB 开始连接 */
    lcd_show_string(30, 110, 200, 16, 16, "USB Connecting...", RED);

    USBH_Init(&g_hUSBHost, USBH_UserProcess, HOST_FS);
    USBH_RegisterClass(&g_hUSBHost, USBH_MSC_CLASS);
    USBH_Start(&g_hUSBHost);

    while (1)
    {
        USBH_Process(&g_hUSBHost);
        delay_ms(10);
        t++;

        if (t == 50)
        {
            t = 0;
            LED0_TOGGLE();
        }
    }
}
```

其中，USBH_HandleTypeDef 是一个用于 USB 主机类通信处理的结构体类型，它包含了

USB 主机通信的各种变量、结构体参数、传输状态和管道信息等。凡是 USB 主机类通信，都必须要用定义一个这样的结构体，这里定义成：g_hUSBHost。

同 USB Device 类通信类似，USB Host 的初始化过程如下：

1， 调用 USBH_Init 函数，初始化 USB 主机内核；

2， 调用 USBH_RegisterClass 函数，链接 MSC 主机类驱动程序到主机内核；

3， 调用 USBH_Start 函数，启动 USB 通信；

经过以上三步处理，USB 主机就启动了，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 usbh_conf.c 里面：

```
/**
 * @brief       USB OTG 中断服务函数
 *   @note      处理所有 USB 中断
 * @param       无
 * @retval      无
 */
void OTG_FS_IRQHandler(void)
{
    HAL_HCD_IRQHandler(&g_hhcd_USB_OTG_FS);
}
```

该函数调用 HAL_HCD_IRQHandler 函数来处理各种 USB 中断请求。

整个 main 函数代码比较简单，不过我们在 main 函数里面，必须不停的调用：USBH_Process 函数，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。

USBH_UserProcess 函数是 USB 主机用户处理回调函数，参数 id 表示 USB 主机当前的一些状态，总共有 6 种状态：

```
#define HOST_USER_SELECT_CONFIGURATION   0x01U      /* USB 进入配置状态 */
#define HOST_USER_CLASS_ACTIVE           0x02U      /* USB 初始化配置完成 */
#define HOST_USER_CLASS_SELECTED         0x03U      /* USB 选择了一个类 */
#define HOST_USER_CONNECTION             0x04U      /* USB 连接成功 */
#define HOST_USER_DISCONNECTION          0x05U      /* USB 连接断开 */
#define HOST_USER_UNRECOVERED_ERROR      0x06U      /* USB 发生了不可恢复错误 */
```

本例程我们只用了：HOST_USER_DISCONNECTION 和 HOST_USER_CLASS_ACTIVE 两个状态。程序运行后，如果没有 U 盘插入，则不会执行到 USBH_UserProcess 函数，当插入 U 盘，并成功识别之后，USBH_UserProcess 函数会进入：HOST_USER_CLASS_ACTIVE 状态，我们就可以挂载 U 盘，并显示容量等信息，表示 U 盘识别完成。此时，如果我们把 U 盘拔出，则会进入：HOST_USER_DISCONNECTION 状态，表示 U 盘拔出，我们取消 U 盘的挂载，并显示设备连接中…（表示当前正在连接设备）。

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，这里和 FATFS 实验一模一样，可以参考该实验的方法操作。

42.5 下载验证

将程序下载到开发板后，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 42.5.1 所示：

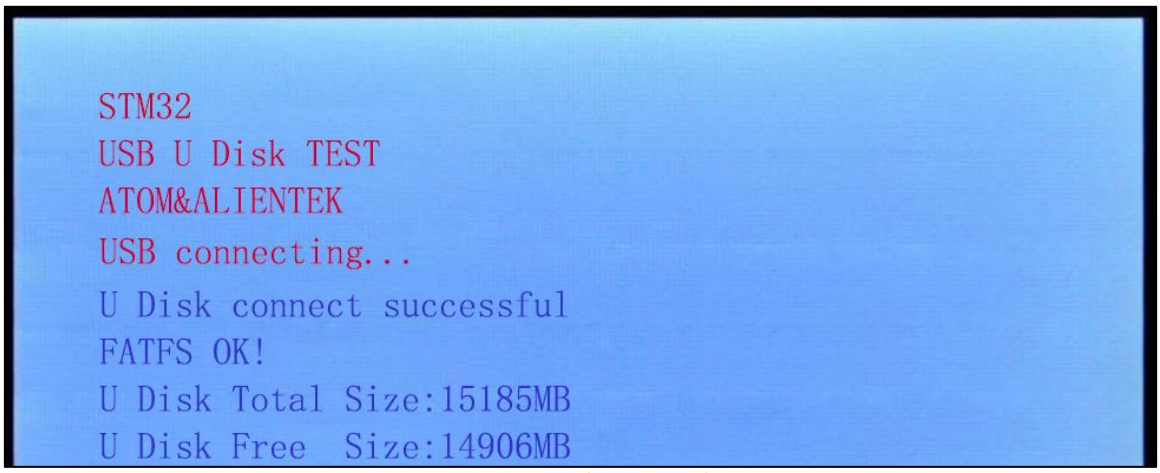


图 42.5.1 U 盘识别成功

此时，我们便可以通过 USMART 来测试 U 盘读写了，如图 42.5.2 和图 42.5.3 所示：

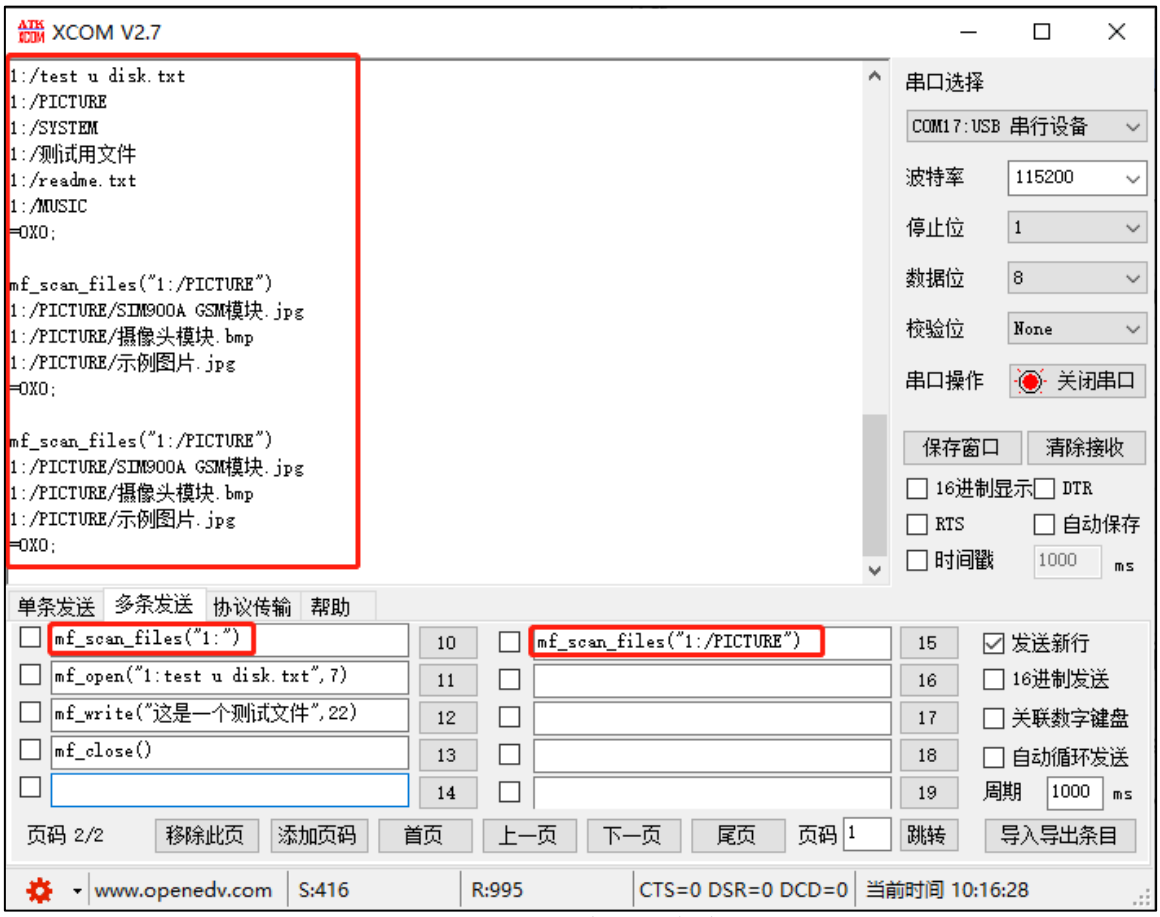


图 42.5.2 测试读取 U 盘读取

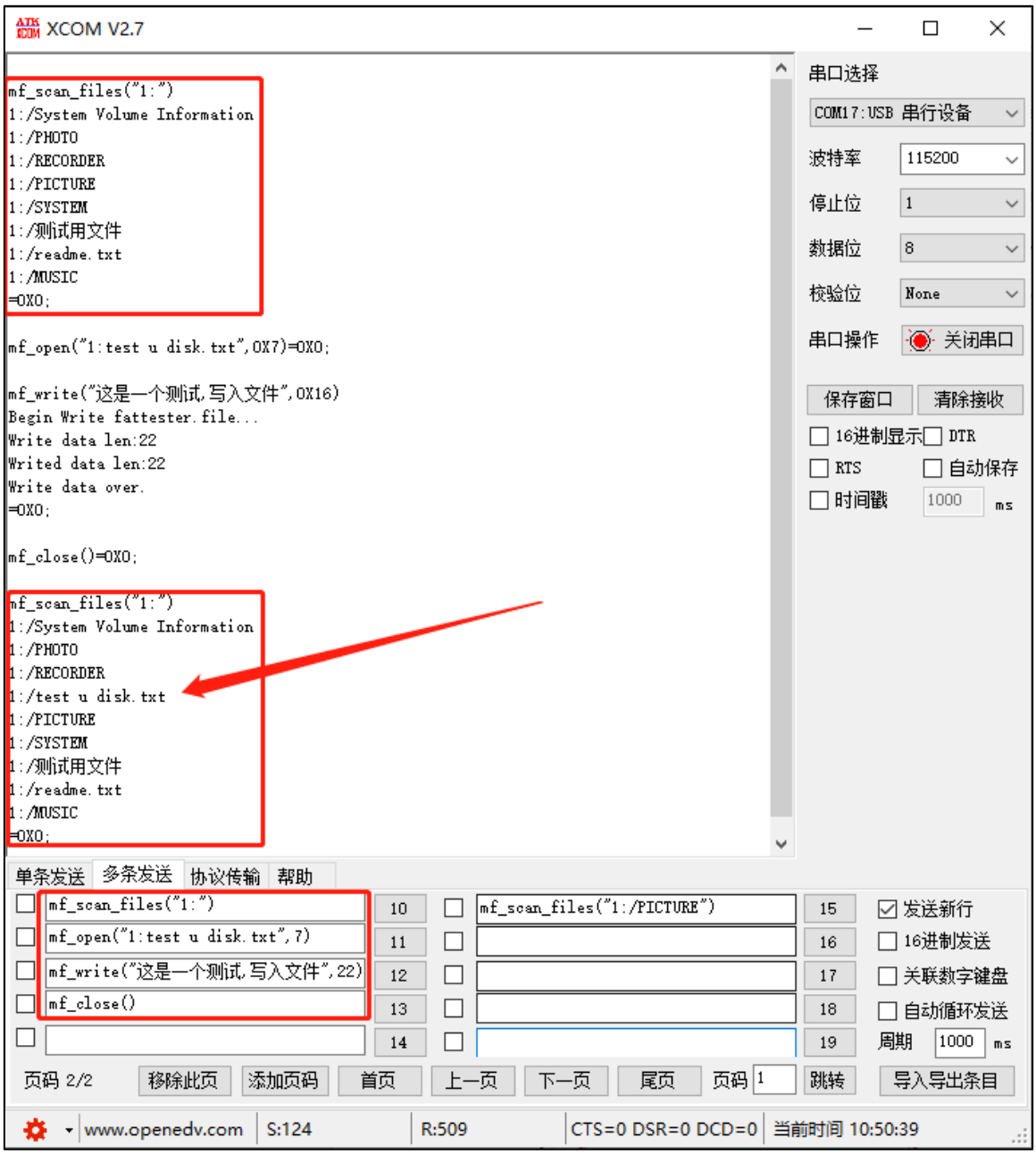


图 42.5.3 测试 U 盘写入

图 42.5.2 分别是通过发送：mf_scan_files("1:")和 mf_scan_files("1:/PICTURE")，扫描 U 盘根目录所有文件和 PICTURE 目录下的所有文件，说明读 U 盘是没问题的。

图 42.5.3 通过发送：mf_open("1: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("1:")，扫描 U 盘根目录文件，发现比图 42.5.3 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。

第四十三章 汉字显示实验

本章，我们将介绍如何使用 STM32 控制 LCD 显示汉字。在本章中，我们将使用外部 FLASH 来存储字库，并可以通过 U 盘更新字库。STM32 读取存在 FLASH 里面的字库，然后将汉字显示在 LCD 上面。本章分为如下几个小节：

43.1 汉字显示介绍
43.2 硬件设计
43.3 程序设计
43.4 下载验证

43.1 汉字显示介绍

43.1.1 字符编码介绍

众所周知，计算机只能识别 0 和 1，所有信息都是以 0 和 1 的形式在计算机里面进行存储，文字信息也是一样。那么如何进行区分文字信息呢？这里就引出了字符编码这个概念，就是对字符进行编码，用规定的 01 数字串来表示特定的文字，最简单的字符编码例子是 ASCII 码。此外还有中文编码，中文编码又可以细分 GB2312，GB13000，GBK，BIG5（繁体）等几种。

1.ASCII 编码

ASCII 编码可以用来表示一些控制字符、英文以及数字，本质上都是二进制数，遵循一定的约定，所以就有了 ASCII 码表。ASCII 码表分为两个部分，第一部分数字编码范围 0~31 是控制字符或者是通讯专用字符，是不具备图形显示但对文本显示有影响的编码；第二部分数字编码范围 32~127 是包含我们日常用到的符号信息，空格、阿拉伯数字、标点符号、大小写英文以及“DEL（删除控制）”，除了 DEL 不具备图形显示外，其他都具备。工程代码中也有 ASCII 的编码表。下面看一下 ASCII 编码表第二部分中的可显示字符如图 43.1.1.1 所示：

二进制	十进制	十六进制	图形	二进制	十进制	十六进制	图形	二进制	十进制	十六进制	图形
0010 0000	32	20	(space)	0100 0000	64	40	@	0110 0000	96	60	`
0010 0001	33	21	!	0100 0001	65	41	A	0110 0001	97	61	a
0010 0010	34	22	"	0100 0010	66	42	B	0110 0010	98	62	b
0010 0011	35	23	#	0100 0011	67	43	C	0110 0011	99	63	c
0010 0100	36	24	$	0100 0100	68	44	D	0110 0100	100	64	d
0010 0101	37	25	%	0100 0101	69	45	E	0110 0101	101	65	e
0010 0110	38	26	&	0100 0110	70	46	F	0110 0110	102	66	f
0010 0111	39	27	'	0100 0111	71	47	G	0110 0111	103	67	g
0010 1000	40	28	(	0100 1000	72	48	H	0110 1000	104	68	h
0010 1001	41	29	)	0100 1001	73	49	I	0110 1001	105	69	i
0010 1010	42	2A	*	0100 1010	74	4A	J	0110 1010	106	6A	j
0010 1011	43	2B	+	0100 1011	75	4B	K	0110 1011	107	6B	k
0010 1100	44	2C	,	0100 1100	76	4C	L	0110 1100	108	6C	l
0010 1101	45	2D	-	0100 1101	77	4D	M	0110 1101	109	6D	m
0010 1110	46	2E	.	0100 1110	78	4E	N	0110 1110	110	6E	n
0010 1111	47	2F	/	0100 1111	79	4F	O	0110 1111	111	6F	o
0011 0000	48	30	0	0101 0000	80	50	P	0111 0000	112	70	p
0011 0001	49	31	1	0101 0001	81	51	Q	0111 0001	113	71	q

0011 0010	50	32	2	0101 0010	82	52	R	0111 0010	114	72	r
0011 0011	51	33	3	0101 0011	83	53	S	0111 0011	115	73	s
0011 0100	52	34	4	0101 0100	84	54	T	0111 0100	116	74	t
0011 0101	53	35	5	0101 0101	85	55	U	0111 0101	117	75	u
0011 0110	54	36	6	0101 0110	86	56	V	0111 0110	118	76	v
0011 0111	55	37	7	0101 0111	87	57	W	0111 0111	119	77	w
0011 1000	56	38	8	0101 1000	88	58	X	0111 1000	120	78	x
0011 1001	57	39	9	0101 1001	89	59	Y	0111 1001	121	79	y
0011 1010	58	3A	:	0101 1010	90	5A	Z	0111 1010	122	7A	z
0011 1011	59	3B	;	0101 1011	91	5B	[	0111 1011	123	7B	{
0011 1100	60	3C	<	0101 1100	92	5C	\	0111 1100	124	7C	|
0011 1101	61	3D	=	0101 1101	93	5D	]	0111 1101	125	7D	}
0011 1110	62	3E	>	0101 1110	94	5E	^	0111 1110	126	7E	~
0011 1111	63	3F	?	0101 1111	95	5F	_				

图 43.1.1.1 可显示字符

2. 中文编码

由于英文文字是由 26 个字母排列组合而成的，所以 ASCII 表就可以适用于表达英文词典库。在汉字系统中，每一个汉字都是一个独立的个体，但是汉字可以以偏旁以及笔画进行划分，不过也是十分杂乱，毕竟汉字现在已经有 8 万多个了，常用的只有 3500 个。所以中文编码是直接对方块字进行编码的，一个汉字使用一个编码。

汉字库十分庞大，需要使用 2 个字节进行编码，假如和 ASCII 码一样只使用 1 字节去编码，那只能表示 256 个汉字，太少了。在前面提及到 GB2312，GB13000，GBK，BIG5(繁体)等，它们都是一个标准。下面对具有代表性的 GB2312 和 GBK 进行讲解。

GB2312 编码

GB2312 是一个简体中文字符集的中国国家标准，也是我们通常说到的国标码。GB2312 由 6763 个常用汉字和 682 个全角的非汉字字符组成。其中根据汉字使用的频率分为两级。一级汉字 3755 个，二级汉字 3008 个。由于字符数量比较大，GB2312 采用了二维矩阵编码法对所有字符进行编码。构造一个 94 行 94 列的方阵，对每一行称为一个“区”，每一列称为一个“位”，然后把所有的字符都依照“GB2312 字符编码规则”，写入方阵中，这样子所有的字符在方阵中都有一个唯一的位置，这个位置就可以用区号和位号组合表示，称为字符的区位码。

因为 GB2312 与西文的存储存在冲突，所以 GB2312 字符在进行存储时，将原来的每一个字节第 8 位设置为 1 和西文加以区分，如果第 8bit 为 0，则表示西文字符，否则就是 GB2312 字符。在实际存储时，采用将区位码的每个字节分别加上 A0H(160)的方法转换为储存码，计算机存储规则则是此编码的补码，而且是位码在前，区码在后。举个例子：汉字“啊”的区位码为 1601，其存储码为 B0A1H，转换过程为：

区位码	区码转换	位码转换	储存码
1001H	10H+A0H=B0H	01H+A0H=A1H	B0A1H

GBK 编码

GBK 编码即汉字内码扩展规范，完全兼容 GB2312，在 GB2312 的基础上，支持繁体字、人名、古汉语等方面出现的罕用字。GBK 采用的是双字节表示，总体编码范围为 0x8140~0xFEFE，第一个字节在 0x81~0xFE 之间，第二个字节分为两个部分，一是 0x40~0x7E，二是 0x80~0xFE。其中和 GB2312 相同的区域，字完全相同，可表示的汉字数达到了 2 万多个，完全能满足我们的一般应用的要求。如图 43.1.1.2 GBK 码位分布图：

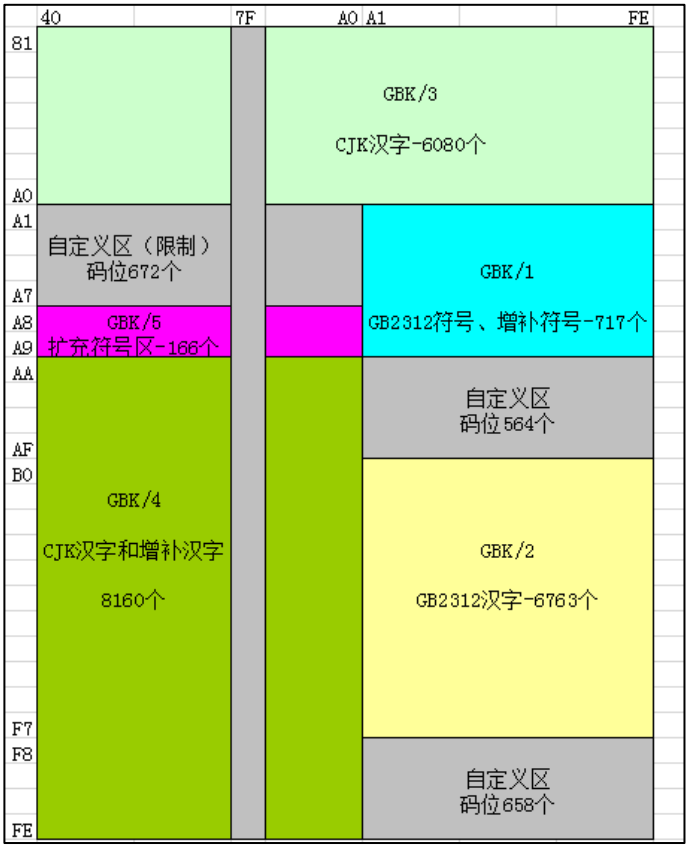


图 43.1.1.2 GBK 码位分布图

在前面 GB2312 也说到，第一字节称为区，那么 GBK 里面总共就有 126 个区（0xFE-0x81+1），第二个字节称为位，我们也可以理解为每个区里面包含的汉字即 190 个汉字（0xFE-0x80+0x7E-0x40+2），GBK 字库总共就 126*190=23940 个汉字。

3.全球统一编码

前一小节讲解的都是中国标准，只有中国使用，并没有表示大多数其他国家的编码。而其他国家又陆续推出各自的编码标准，互不兼容，非常不利于国际交流。所以在后来国际标准化组织(ISO)发行了一个全球统一编码表，把全球各国文字都统一在一个编码标准里，这个全球统一编码表就叫做 Unicode。Unicode 字符集对世界上各国文化使用到的字母和符号进行标号，对每一个字符都分配一个唯一的编号，字符的编号从 0x000000 到 0x10FFFF。

Unicode 没有规定字符对应的二进制码如何存储，只是对每一个字符进行编号。为了解决 Unicode 编码问题，UTF-8、UTF-16 和 UTF-32 的编码方式诞生了。

UTF-8 编码

UTF-8 是一个非常常用的编码方式，漂亮的实现了对 ASCII 码的向后兼容，它是目前互联网上使用最广泛的一种 Unicode 编码方式，它的最大特点就是可变长。它可以使用 1-4 个字节表示一个字符，根据字符的不同变换长度。编码规则如下：

1.对于单字节的字符，第一位设为 0，后面 7 位对应这个字符的 Unicode 码点（即编号）。因此英文中 0~127 字符与 ASCII 码完全相同。这意味着可以使用 UTF-8 编码格式打开用 ASCII 码编写的文档。

2.对于需要使用 N 个字节来表示的字符（N>1），第一个字节的前 N 位都设为 1，第 N+1 位设为 0，剩下的 N-1 个字节的前两位都设为 10，剩下的二进制位则使用这个字符的 Unicode 的码点进行填充。

上面的表述可能比较难接受，下面看一下编码规则表就清晰很多，如表 43.1.1.1 所示：

Unicode 十六进制码点范围	UTF-8 二进制
0000 0000 - 0000 007F	0xxxxxxx

0000 0080 - 0000 07FF	110xxxxx 10xxxxxx
0000 0800 - 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000 - 0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

表 43.1.1.1 UTF-8 编码规则表

有了上面这个编码规则表，进行 UTF-8 编码和解码就简单多了。以汉字“汉”为例，它的 Unicode 的码点是 0x6c49(110 1100 0100 1001)，通过查表发现它属于第三行的规则，具体格式：1110xxxx 10xxxxxx 10xxxxxx。接着我们就可以从“汉”的二进制数的最后一位开始，从后向前依次填充对应的格式中的 x，多出的 x 用 0 补充。这样就可以的到“汉”的 UTF-8 编码：11100110 10110001 10001001，转换成十六进制就是 0xE6 0xB1 0x89。

解码的过程也是十分简单，通过判断一个字节的第一位，假如是 0，说明这个字节对应一个字符；假如是 1 则需要往后数有多少个连续不间断的 1，就表示该字符占用多少个字节。

UTF-16

UTF-16 采用的是 2 字节或 4 字节编码方式。对于 Unicode 编码范围在 0x0000~0xFFFF 之间字符，UTF-16 使用两个字节存储，并直接存储 Unicode 编号，不用编码转换。对于 Unicode 编号范围在 0x10000~0x10FFFF 之间的字符，UTF-16 使用四个字节存储，具体来说是：将字符编号的所有比特位分成两部分，较高的一些比特位用一个值介于 0xD800 和 0xDBFF 之间的双字节存储，较低的一些比特位（剩下的比特位），用一个值介于 0xDC00 和 DFFF 之间的双字节存储。

上面表述的可能有点模糊，请看下面的表格 43.1.1.2 所示：

Unicode 编码范围 (Hex)	具体的 Unicode 编号（BIN）	UTF-16 编码	编码后的字节数
0000 0000~0000 FFFF	xxxxxxxx xxxxxxxx	xxxxxxxx xxxxxxxx	2
0001 0000~0010 FFFF	yyyy yyyy yyxx xxxx xxxx	110110yyyyyyyyyy 110111xx xxxxxxxx	4

表 43.1.1.2 UTF-16 编码规则表

举个例子，用 UTF-16 两个字节去表示“汉”字，它的 Unicode 码点是 0x6C49，那么遵行上面的表格规则，那么用 UTF-16 表示的话就是 01101100 01001001（共 16bit，两个字节）。

现在以第二种情况讲解，汉字“？”为例，它的 Unicode 码点为 0x20BB7，该码点显然超出了基本平面的范围(0x0000 – 0xFFFF)，因此需要使用四个字节进行表示，首先用 0x00BB7–0x10000 计算超出部分，然后将其用 20 个二进制位表示(不足的前面补 0)，结果为 0001000010 1110110111。接着将高 10 位映射到 U+D800 到 U+DBFF 之间，低 10 位映射到 U+DC00 到 U+DFFF 即可。那么换算一下即（110110 0000000000 + 0001000010），高位转换一下 16 进制即 0xD842，同理低位也是这样运算，得到 0xDFB7。因此，得出汉字“？”的 UTF-16 编码为 0xD842 0xDFB7。

UTF-32

UTF-32 是固定长度的编码，始终占用 4 个字节，足以容纳所有的 Unicode 字符，所以直接存储 Unicode 编码即可，不需要任何编码转换。浪费了空间，提供了效率。

大小端模式

在了解了前面的各种解码之后，就知道一个字符可能占用多个字节，那么多个字节在计算机中如何进行存储呢？比如字符 0xABCD，它的存储格式到底时 AB CD，还是 CD AB 呢？

实际上两者都有可能，并且分别有不同的名字。如果存储为 AB CD，则称为 Big Endian 即大端；如果存储为 CD AB，则称为 Little Endian 即小端。

Big Endian：高字节在前，低字节在后，详细一点，就是将高位的字节放在低地址表示。

Little Endian：低字节在前，高字节在后，详细一点，就是将高位的字节放在高地址表示。

在了解大小端的意思之后，我们稍微科普一下 BOM。

BOM

BOM 是文档最前面的标记，位于文本文件的开头，一种标记对应一种编码方式。为什么会需要这个东西呢？因为计算机打开文档时，它首先需要获取一些信息，知道用什么编码方式去解码，才能知道文档写的是什么内容。注意：BOM 是对 Unicode 的几种编码而言的。下面我们看一下 BOM 的标记表，如表 43.1.1.3 所示：

BOM 标记	表示的编码
0xEF 0xBB 0xBF	UTF-8
0xFF 0xFE	UTF-16 小端模式
0xFE 0xFF	UTF-16 大端模式
0xFF 0xFE 0x00 0x00	UTF-32 小端模式
0x00 0x00 0xFE 0xFF	UTF-32 大端模式

表 43.1.1.3 BOM 标记表

43.1.2 字库的生成

有了编码，我们就能可以在计算机上对字符进行操作，但是如果计算机处理完字符后直接以编码的形式输出，我们很难一下子知道究竟是什么图形的字符。因为我们平常看到的都是字符的图形化显示，这里呢，就需要我们提供配套的字符图形。字符图形又称为字模，多个字模组成的文件就叫做字库。当我们为计算机提供了编码和字库的前提下，计算机就可以把字符编码转化成对应的字符图形，我们才能清晰识别。

字模的结构

在前面 OLED 显示实验章节也有说到字模相关知识，这里我们继续展开详细讲解一下，字模是字符的图形结构，字模的实质就是一个个像素点数据。为了方便处理，我们把字模定义成方块形状的像素点阵，像素点只有 0 和 1 两种状态。在单色图像数据中，像素点数值置 1 时，点亮了该像素点，若像素点数值置 0 时，熄灭该像素点。“汉字”字模图，如图 43.1.2.1 所示：

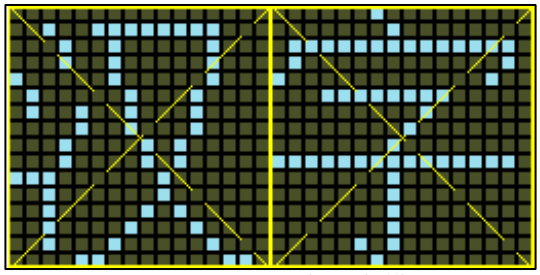


图 43.1.2.1 “汉字”字模图

这两个字模的大小都是 16*16，计算机要表示这样的图形，只需要 16x16 个二进制数据位即可，需要使用 16x16/8=32 个字节保存字模数据。这里存在一个问题，字模数据是从哪里标记为第一个字节呢？我们知道了第一个字节以及字模的走向，我们就可以把生成的点阵数据和字模图像上的小方块进行匹配，更好了解两者的关系。下面我们解密一下，如图 43.1.2.2 取模方式（从上到下，从左到右）进行取模。

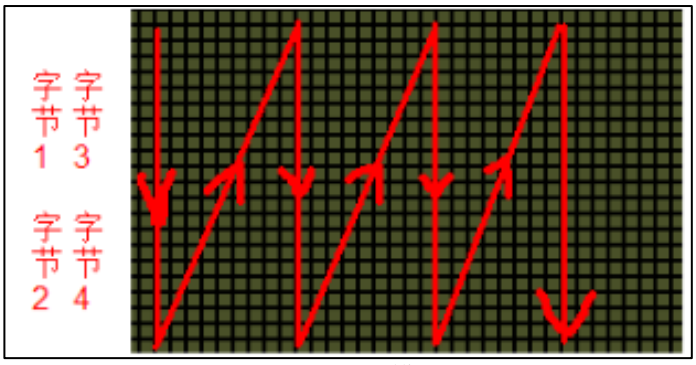


图 43.1.2.2 取模方式

设置好后，生成的数据如下：

```
{0x08,0x20,0x06,0x20,0x40,0x3E,0x30,0xC0,0x03,0x01,0x40,0x01,0x78,0x02,
 0x47,0x04,0x40,0xC8,0x40,0x30,0x40,0xC8,0x47,0x04,0x78,0x02,0x00,0x01,
 0x00,0x01,0x00,0x00},/*"汉",0*/
```

便于大家更好地将点阵数据和图形连接起来，请看图 43.1.2.3 所示：

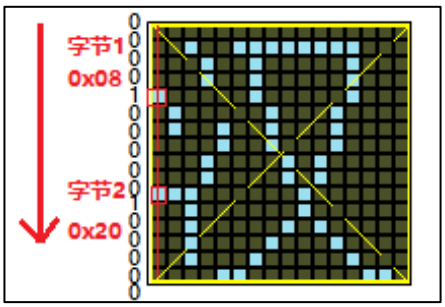


图 43.1.2.3 点阵数据和图形匹配图

了解了字模的结构之后，我们就需要去生成字库了。我们用到一款由易木雨软件工作室涉及的点阵字库生成器 V3.8 软件。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)和Unicode 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，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 43.1.2.4 所示：

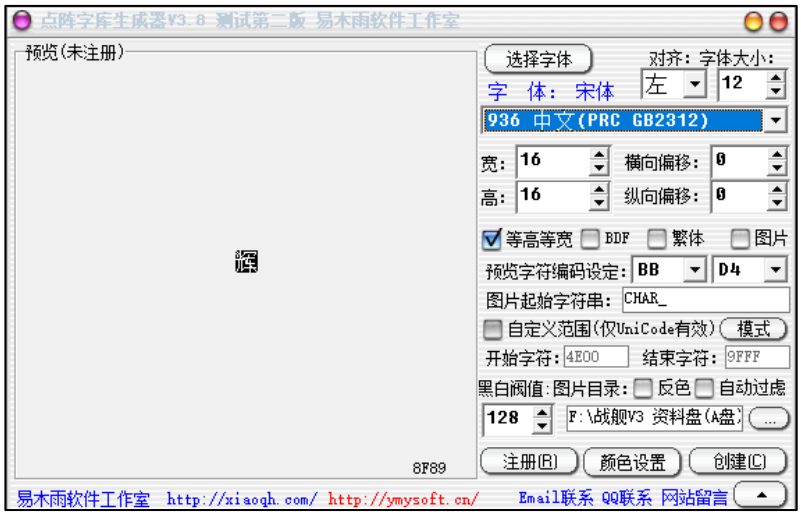


图 43.1.2.4 点阵字库生成器默认界面

本实验，我们总共要生成 3 个字库：12*12 字库、16*16 字库和 24*24 字库。这里以 16*16 字库为例进行介绍，其他两个字库的制作方法类似。要生成 16*16 的 GBK 字库，则需要选择字体大小、选择编码字库、选择字宽和高度、取模方式、存放路径等操作，最后点击创建，就可以开始生成我们需要的字库（.DZK 文件），具体设置如图 43.1.2.5 所示：

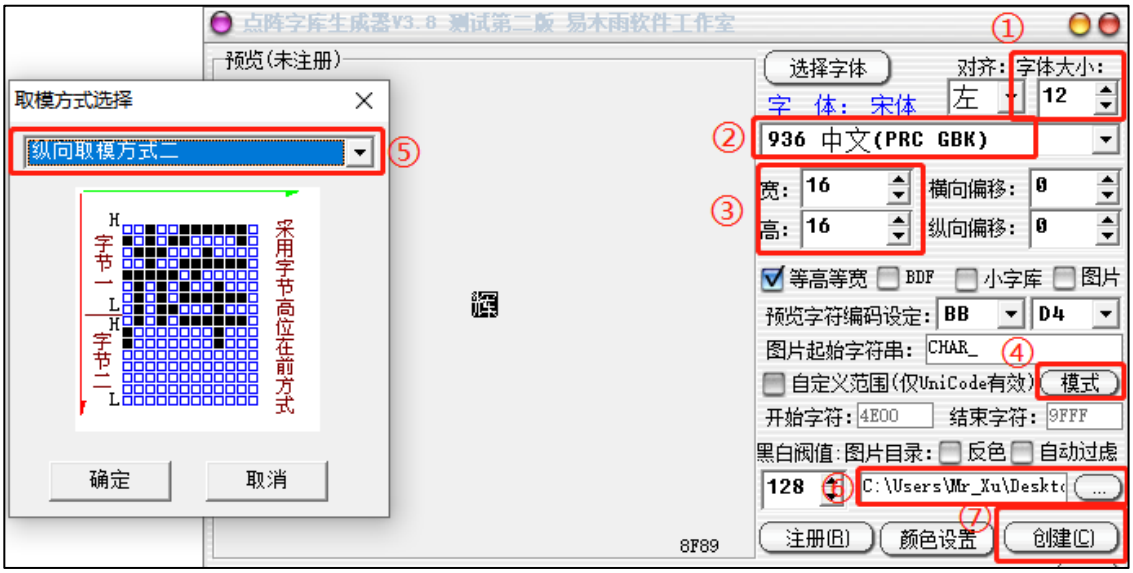


图 43.1.2.5 生成 GBK16*16 字库的设置方法

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 是电脑端字体的大小，dsize 是点阵大小（12、16、24 等）。所以 16*16 点阵大小对应的是 12 号字体。

生成完之后，我们把文件名和后缀改成：GBK16.FON。同样的方法，生成 12*12 的点阵库（GBK12.FON）和 24*24 的点阵库（GBK24.FON），总共制作 3 个字库。

构建字库完成，那我们怎么把字库跟编码连接起来呢？以 GBK16.FON 为例子，它是根据 GBK 的编码规则从 0X8140 开始，逐一建立。因为 16*16 的点阵大小为 32 字节，所以在 FON 文件中第一个 32 字节的数据就是“ 丐”的字模数据，它的编码是 0x8140，而第二个 32 字节的数据就是“ 丒”的字模数据，它的编码就是 0x8141。在前面讲解 GBK 的时候，也提到了 GBK 具有 126 区，每个区具有 190 个汉字，那么我们就可以通过计算地址偏移，最终在这个字库里面定位汉字了：

当 GBKL<0x7F 时：Hp=((GBKH-0x81)*190+GBKL-0x40)*csize；

当 GBKL>0x80 时：Hp=((GBKH-0x81)*190+GBKL-0x41)*csize；

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)，Hp 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)，csize 代表一个汉字点阵所占的字节数。

这样我们只要得到了汉字的 GBK 码，就可以得到该汉字点阵在点阵库里面的位置，从而获取其点阵数据，显示这个汉字了。

43.1.3 汉字显示原理

汉字在液晶上的显示原理与前面 OLED 实验中显示字符的是一样的。汉字在液晶上的显示其实就是一些点的显示与不显示，这就相当于我们的笔一样，有笔经过的地方就画出来，没经过的地方就不画。所以要显示汉字，我们首先要知道汉字的点阵数据，这些数据可以由专门的软件来生成，在前面字库生成已经有讲到如何生成字模与字库了。

在我们前面制作的三个 GBK 字库的基础上，将它们放在 U 盘里，然后通过 U 盘，将字库文件复制到外部 NORFLASH 里，这样，NORFLASH 就相当于一个汉字字库芯片了。

单片机要显示汉字的步骤：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，就可以把字符图形在单片机上显示出来了。

43.2 硬件设计

1. 例程功能

本实验开机的时候先检测 NORFLASH 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 U 盘和文件系统，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。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。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
LED1 – PE1
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）按键
KEY0 – PE2
4）正点原子 2.8/3.5/4.3/7 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
5）USB_HOST 接口(D-/D+连接在 PA11/PA12 上)

43.3 程序设计

43.3.1 程序流程图

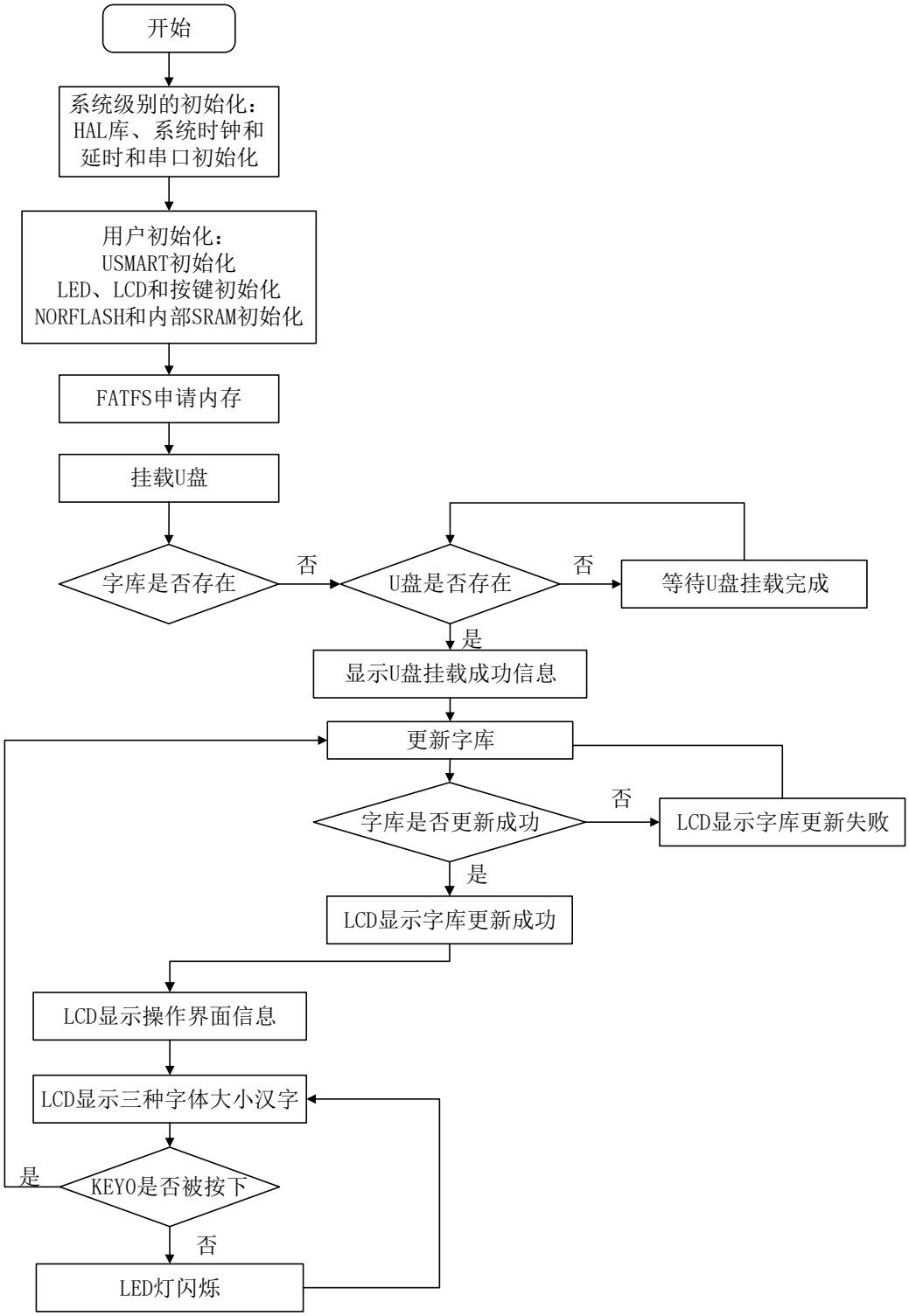


图 43.3.1.1 汉字显示实验程序流程图

43.3.2 程序解析

1. TEXT 代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。TEXT 驱动源码包括四个文件：text.c、text.h、fonts.c 和 fonts.h。

汉字显示实验代码主要分为两部分：一部分就是对字库的更新，另一部分就是对汉字的显示。字库的更新代码放在 font.c 和 font.h 文件中，汉字的显示代码就放在 text.c 和 text.h 中。

下面我们介绍一下有关字库操作的代码，首先我们先看一下 fonts.h 文件中字库信息结构体定义，其代码如下：

```
/* 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
{
    uint8_t  fontok;              /* 字库存在标志，0XAA，字库正常；其他，字库不存在 */
    uint32_t ugbkaddr;            /* unigbk 的地址 */
    uint32_t ugbksize;            /* unigbk 的大小 */
    uint32_t f12addr;             /* gbk12 地址 */
    uint32_t gbk12size;           /* gbk12 的大小 */
    uint32_t f16addr;             /* gbk16 地址 */
    uint32_t gbk16size;           /* gbk16 的大小 */
    uint32_t f24addr;             /* gbk24 地址 */
    uint32_t gbk24size;           /* gbk24 的大小 */
} _font_info;
```

这个结构体用于记录字库的首地址以及字库大小等信息，总共占用 33 个字节，第一个字节用来标识字库是否 OK，其他的用来记录地址和文件大小。因为我们将 NORFLASH（25Q128）的前 12M 字节给了 FATFS 管理（用做本地磁盘），12M 字节后紧跟 3 个字库+UNIGBK.BIN 总大小 3.09M 字节 791 个扇区，在 15.10M 字节后，预留了 100K 字节给用户自己使用。所以，我们的存储地址是从 12*1024*1024 处开始的。最开始的 33 个字节给_font_info 用，用于保存_font_info 结构体数据，之后是 UNIGBK.BIN、GBK12.FON、GBK16.FON 和 GBK24.FON。

下面介绍 font.c 文件几个重要的函数。

字库初始化函数也是利用其存储顺序，进行检查字库，其定义如下：

```
/**
 * @brief       初始化字体
 * @param       无
 * @retval      0, 字库完好; 其他, 字库丢失;
 */
uint8_t fonts_init(void)
{
    uint8_t t = 0;

    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++;
        /* 读出 ftinfo 结构体数据 */
        norflash_read((uint8_t *)&ftinfo, FONTINFOADDR, sizeof(ftinfo));

        if (ftinfo.fontok == 0xAA)
        {
            break;
        }

        delay_ms(20);
    }
    if (ftinfo.fontok != 0xAA)
    {
        return 1;
```

```
    }
    return 0;
}
```

该函数就是从 NORFLASH 芯片中，701*4096 地址开始往后的 41 个字节的数据读取出来，进而判断字库结构体 ftinfo 的字库标记 fontok 是否为 AA，确定字库是否完好。

有人会有疑问，ftinfo.fontok 在哪里赋值 AA 呢？肯定是字库更新完毕后，给该标记赋值的，那下面就来看一下是不是这样子，字库更新函数定义如下：

```
/**
 * @brief       更新字体文件
 *   @note      所有字库一起更新(UNIGBK,GBK12,GBK16,GBK24,GBK32)
 * @param       x, y    : 提示信息的显示地址
 * @param       size    : 提示信息字体大小
 * @param       src     : 字库来源磁盘
 *   @arg               "0:", SD 卡;
 *   @Arg               "1:", FLASH 盘
 *   @arg               "2:", U 盘
 * @param       color   : 字体颜色
 * @retval      0, 成功; 其他, 错误代码;
 */
uint8_t fonts_update_font(uint16_t x, uint16_t y, uint8_t size,
                          uint8_t *src, uint16_t color)
{
    uint8_t *pname;
    uint32_t *buf;
    uint8_t res = 0;
    uint16_t i, j;
    FIL *fftemp;
    uint8_t 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 < 4; 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 *)FONT_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. */
    {
        /* 提示正在擦除扇区 */
        lcd_show_string(x, y, 240, 320, size, "Erasing sectors... ", color);
```

```
        for (i = 0; i < FONTSECSIZE; i++)        /* 先擦除字库区域,提高写入速度 */
        {
            fonts_progress_show(x + 20 * size / 2, y, size, FONTSECSIZE, i,
                                color);          /* 进度显示 */
            norflash_ex_read((uint8_t *)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)
            {
                /* 需要擦除的扇区 */
                norflash_ex_erase_sector((FONTINFOADDR / 4096) + i);
            }
        }

        for (i = 0; i < 4; i++) /* 依次更新 UNIGBK,GBK12,GBK16,GBK24 */
        {
            lcd_show_string(x, y, 240, 320, size, FONT_UPDATE_REMIND_TBL[i],
                            color);
            strcpy((char *)pname, (char *)src);      /* copy src 内容到 pname */
            strcat((char *)pname, (char *)FONT_GBK_PATH[i]);/* 追加具体文件路径 */
            res = fonts_update_fontx(x + 20 * size / 2, y, size, pname, i,
                                     color);         /* 更新字库 */

            if (res)
            {
                myfree(SRAMIN, buf);
                myfree(SRAMIN, pname);
                return 1 + i;
            }
        }

        /* 全部更新好了 */
        ftinfo.fontok = 0xAA;
        /* 保存字库信息 */
        norflash_ex_write((uint8_t *)&ftinfo, FONTINFOADDR, sizeof(ftinfo));
    }

    myfree(SRAMIN, pname);  /* 释放内存 */
    myfree(SRAMIN, buf);
    return rval;            /* 无错误. */
}
```

函数的实现：动态申请内存→尝试打开文件(UNIGBK、GBK12、GBK16 和 GBK24)，确定文件是否存在→擦除字库→依次更新 UNIGBK、GBK12、GBK16 和 GBK24→写入 ftinfo 结构体信息。

在字库更新函数中能直接看到的是 ftinfo.fontok 成员被赋值，而其他成员在单个字库更新函数中被赋值，接下来分析一下更新某个字库函数，其代码如下：

```
/**
 * @brief       更新某一个字库
 * @param       x, y    : 提示信息的显示地址
 * @param       size    : 提示信息字体大小
 * @param       fpath   : 字体路径
 * @param       fx      : 更新的内容
 *   @arg               0, ungbk;
 *   @Arg               1, gbk12;
 *   @arg               2, gbk16;
 *   @arg               3, gbk24;
 * @param       color   : 字体颜色
```

```
 * @retval      0, 成功; 其他, 错误代码;
 */
static uint8_t fonts_update_fontx(uint16_t x, uint16_t y, uint8_t size,
                                  uint8_t *fpath, uint8_t fx, uint16_t color)
{
    uint32_t flashaddr = 0;
    FIL *fftemp;
    uint8_t *tempbuf;
    uint8_t res;
    uint16_t bread;
    uint32_t offx = 0;
    uint8_t 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 *)fpath, FA_READ);

    if (res)rval = 2;    /* 打开文件失败 */

    if (rval == 0)
    {
        switch (fx)
        {
            case 0:       /* 更新 UNIGBK.BIN */
                /* 信息头之后，紧跟 UNIGBK 转换码表 */
                ftinfo.ugbkaddr = FONTINFOADDR + sizeof(ftinfo);
                ftinfo.ugbksize = fftemp->obj.objsize;    /* UNIGBK 大小 */
                flashaddr = ftinfo.ugbkaddr;
                break;

            case 1:       /* 更新 GBK12.BIN */
                /* UNIGBK 之后，紧跟 GBK12 字库 */
                ftinfo.f12addr = ftinfo.ugbkaddr + ftinfo.ugbksize;
                ftinfo.gbk12size = fftemp->obj.objsize;   /* GBK12 字库大小 */
                flashaddr = ftinfo.f12addr;               /* GBK12 的起始地址 */
                break;

            case 2:       /* 更新 GBK16.BIN */
                /* GBK12 之后，紧跟 GBK16 字库 */
                ftinfo.f16addr = ftinfo.f12addr + ftinfo.gbk12size;
                ftinfo.gbk16size = fftemp->obj.objsize;  /* GBK16 字库大小 */
                flashaddr = ftinfo.f16addr;              /* GBK16 的起始地址 */
                break;

            case 3:       /* 更新 GBK24.BIN */
                /* GBK16 之后，紧跟 GBK24 字库 */
                ftinfo.f24addr = ftinfo.f16addr + ftinfo.gbk16size;
                ftinfo.gbk24size = fftemp->obj.objsize; /* GBK24 字库大小 */
                flashaddr = ftinfo.f24addr;             /* GBK24 的起始地址 */
                break;
        }

        while (res == FR_OK)   /* 死循环执行 */
        {
            res = f_read(fftemp, tempbuf, 4096, (UINT *)&bread);    /* 读取数据 */

            if (res != FR_OK)break;     /* 执行错误 */
            /* 从 0 开始写入 bread 个数据 */
```

```
            norflash_ex_write(tempbuf, offx + flashaddr, bread);
            offx += bread;
            /* 进度显示 */
            fonts_progress_show(x, y, size, fftemp->obj.objsize, offx, color);

            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;
}
```

单个字库更新函数，主要是对把字库从 U 盘中读取出数据，写入 NORFLASH。同时把字库大小和起始地址保存在 ftinfo 结构体里，在前面的整个字库更新函数中使用函数：

```
norflash_write((uint8_t *)&ftinfo,FONTINFOADDR,sizeof(ftinfo)); /*保存字库信息*/
```

结构体的所有成员一并写入到那 41 个字节。有了这个字库信息结构体，就能很容易进行定位。结合前面的说到的根据地址偏移寻找汉字的点阵数据，我们就可以开始真正把汉字搬上屏幕中去了。

首先我们肯定需要获得汉字的 GBK 码，这里 MDK 已经帮我们实现了。这里用一个例子说明：

```
char* HZ_str = "正点原子";
printf("正点原子的'正'字GBK高位码: %#x \r\n",*HZ_str);
printf("正点原子的'正'字GBK低位码: %#x \r\n",*(HZ_str+1));
```

串口打印

```
正点原子的'正'字GBK高位码: 0xd5
正点原子的'正'字GBK低位码: 0xfd
```

在这里可以看出 MDK 识别汉字的方式是 GBK 码，换句话来说就是 MDK 自动会把汉字看成是两个字节表示的东西。知道了要表示的汉字和其 GBK 码，那么就可以去找对应的点阵数据。在 text.c 文件中，我们就定义了一个获取汉字点阵数据的函数，其定义如下：

```
/**
 * @brief       获取汉字点阵数据
 * @param       code  : 当前汉字编码(GBK 码)
 * @param       mat   : 当前汉字点阵数据存放地址
 * @param       size  : 字体大小
 *   @note      size 大小的字体,其点阵数据大小为: (size / 8 + ((size % 8) ? 1 : 0))
                                                  * (size)  字节
 * @retval      无
 */
static void text_get_hz_mat(unsigned char *code, unsigned char *mat,
                            uint8_t size)
{
    unsigned char qh, ql;
    unsigned char i;
    unsigned long foffset;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    uint8_t csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size);
    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:
            norflash_ex_read(mat, foffset + ftinfo.f12addr, csize);
            break;

        case 16:
            norflash_ex_read(mat, foffset + ftinfo.f16addr, csize);
            break;

        case 24:
            norflash_ex_read(mat, foffset + ftinfo.f24addr, csize);
            break;

        case 32:
            norflash_ex_read(mat, foffset + ftinfo.f32addr, csize);
            break;
    }
}
```

函数实现的依据就是前面 43.1.2 小节讲到的两条公式：

当 GBKL<0x7F 时：Hp=((GBKH-0x81)*190+GBKL-0x40)*csize；

当 GBKL>0x80 时：Hp=((GBKH-0x81)*190+GBKL-0x41)*csize；

目标汉字的 GBK 码满足上面两条公式其一，就会得出与一个 GBK 对应的汉字点阵数据的偏移。在这个基础上，通过判断汉字的大小，就可以从对应的字库提取目标汉字点阵数据。

在获取到点阵数据后，接下来就可以进行汉字显示，下面看一下汉字显示函数，其定义如下：

```
/**
 * @brief       显示一个指定大小的汉字
 * @param       x,y   : 汉字的坐标
 * @param       font  : 汉字 GBK 码
 * @param       size  : 字体大小
 * @param       mode  : 显示模式
 *   @note            0, 正常显示(不需要显示的点,用 LCD 背景色填充,即 g_back_color)
 *   @note            1, 叠加显示(仅显示需要显示的点, 不需要显示的点, 不做处理)
 * @param       color : 字体颜色
 * @retval      无
 */
void text_show_font(uint16_t x, uint16_t y, uint8_t *font, uint8_t size,
                                                uint8_t mode, uint16_t color)
{
    uint8_t temp, t, t1;
    uint16_t y0 = y;
    uint8_t *dzk;
    /* 得到字体一个字符对应点阵集所占的字节数 */
    uint8_t csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size);
    if (size != 12 && size != 16 && size != 24)
```

```
    {
        return;     /* 不支持的 size */
    }

    dzk = mymalloc(SRAMIN, size);        /* 申请内存 */
    if (dzk == 0) return;                /* 内存不够了 */

    text_get_hz_mat(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_draw_point(x, y, color);          /* 画需要显示的点 */
            }
            else if (mode == 0)     /* 如果非叠加模式，不需要显示的点,用背景色填充 */
            {
                lcd_draw_point(x, y, g_back_color);  /* 填充背景色 */
            }

            temp <<= 1;
            y++;
            if ((y - y0) == size)
            {
                y = y0;
                x++;
                break;
            }
        }
    }
    myfree(SRAMIN, dzk);    /* 释放内存 */
}
```

汉字显示函数通过调用获取汉字点阵数据函数 text_get_hz_mat 就获取到点阵数据，使用 lcd 画点函数把点阵数据中为“1”的点都画出来，最终会在 LCD 显示你所要表示的汉字。

其他函数就不多讲解，大家可以自行消化。

2. main.c 代码

main.c 代码如下：

```
int main(void)
{
    uint8_t i, j;
    uint8_t fontx[2];                    /* GBK 码 */
    uint8_t key = 0, t = 0;
    uint32_t fontcnt = 0;

    HAL_Init();                          /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟,168Mhz */
    delay_init(168);                     /* 延时初始化 */
    usart_init(115200);                  /* 串口初始化为 115200 */
    usmart_dev.init(84);                 /* USMART 初始化 */
    led_init();                          /* 初始化 LED */
    lcd_init();                          /* 初始化 LCD */
    key_init();                          /* 初始化按键 */
    norflash_init();

    my_mem_init(SRAMIN);                 /* 初始化内部 SRAM 内存池 */
    my_mem_init(SRAMCCM);                /* 初始化内部 SRAMCCM 内存池 */
```

```
exfuns_init();                          /* 为 fatfs 相关变量申请内存 */

USBH_Init(&g_hUSBHost, USBH_UserProcess, HOST_FS);
USBH_RegisterClass(&g_hUSBHost, USBH_MSC_CLASS);
USBH_Start(&g_hUSBHost);
delay_ms(1800);
btim_timx_int_init(1000, 840);          /* 84 000 000 / 840 = 100 000 100Khz */

while(usb_mount_status==0)              /* 等待 USB 设备挂载完成 */
{
    lcd_show_string(30, 30, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 50, 200, 16, 16, "USB Connecting...", RED);
    delay_ms(200);
    delay_ms(200);
}
while (fonts_init())                    /* 检查字库 */
{
UPD:
    printf("update font\r\n");
    usb_mount_status = 0;
    lcd_clear(WHITE);                   /* 清屏 */

    USBH_Init(&g_hUSBHost, USBH_UserProcess, HOST_FS);
    USBH_RegisterClass(&g_hUSBHost, USBH_MSC_CLASS);
    USBH_Start(&g_hUSBHost);
    delay_ms(1800);
    btim_timx_int_init(1000, 840);      /* 84 000 000 / 840 = 100 000 100Khz */

    while(usb_mount_status==0)          /* 等待 USB 设备挂载完成 */
    {
        lcd_show_string(30, 30, 200, 16, 16, "STM32", RED);
        lcd_show_string(30, 50, 200, 16, 16, "USB Connecting...", RED);
        delay_ms(200);
        delay_ms(200);
    }

    lcd_clear(WHITE);                   /* 清屏 */
    lcd_show_string(30, 30, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 50, 200, 16, 16, "Font Updating...", RED);

    key = fonts_update_font(20, 90, 16, (uint8_t *)"1:", RED);  /* 更新字库 */
    while (key)                         /* 更新失败 */
    {
        lcd_show_string(30, 90, 200, 16, 16, "Font Update Failed!", RED);
        delay_ms(200);
        lcd_fill(20, 90, 200 + 20, 90 + 16, WHITE);
        delay_ms(200);
    }
    TIM3->CR1&=~(1<<0);                 /* 关闭定时器 3 */
    lcd_show_string(30, 90, 200, 16, 16, "Font Update Success!   ", RED);
    delay_ms(1500);
    lcd_clear(WHITE);                   /* 清屏 */
}

TIM3->CR1&=~(1<<0);                     /* 关闭定时器 3 */

lcd_clear(WHITE);                       /* 清屏 */
text_show_string(30, 30, 200, 16, "正点原子 STM32 开发板", 16, 0, RED);
text_show_string(30, 50, 200, 16, "GBK 字库测试程序", 16, 0, RED);
text_show_string(30, 70, 200, 16, "正点原子@ALIENTEK", 16, 0, RED);
text_show_string(30, 90, 200, 16, "按 KEY0,更新字库", 16, 0, RED);

text_show_string(30, 110, 200, 16, "内码高字节:", 16, 0, BLUE);
```

```
    text_show_string(30, 130, 200, 16, "内码低字节:", 16, 0, BLUE);
    text_show_string(30, 150, 200, 16, "汉字计数器:", 16, 0, BLUE);

    text_show_string(30, 180, 200, 24, "对应汉字为:", 24, 0, BLUE);
    text_show_string(30, 204, 200, 16, "对应汉字(16*16)为:", 16, 0, BLUE);
    text_show_string(30, 220, 200, 16, "对应汉字(12*12)为:", 12, 0, BLUE);

    while (1)
    {
        for (i = 0x81; i < 0xff; i++)    /* GBK 内码高字节范围为 0x81~0xFE */
        {
            fontx[0] = i;
            lcd_show_num(118, 110, i, 3, 16, BLUE); /* 显示内码高字节 */

            for (j = 0x40; j < 0xfe; j++)/*GBK 内码低字节范围 0x40~0x7E, 0x80~0xFE)*/
            {
                if (j == 0x7f)
                    continue;

                fontcnt++;
                lcd_show_num(118, 130, j, 3, 16, BLUE);       /* 显示内码低字节 */
                lcd_show_num(118, 150, fontcnt, 5, 16, BLUE);/* 汉字计数显示 */
                fontx[1] = j;
                text_show_font(30 + 132, 180, fontx, 24, 0, BLUE);
                text_show_font(30 + 144, 204, fontx, 16, 0, BLUE);
                text_show_font(30 + 108, 220, fontx, 12, 0, BLUE);
                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;           /* 跳转到 UPD 位置(强制更新字库) */
                    }
                }
                LED0_TOGGLE();
            }
        }
    }
}
```

main 函数实现了我们在硬件设计例程功能所表述的一致，至此整个软件设计就完成了。

43.4 下载验证

本例程支持 12*12、16*16、24*24 和 32*32 四种字体的显示，将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 开始显示四种大小的汉字及内码如图 43.4.1 所示：

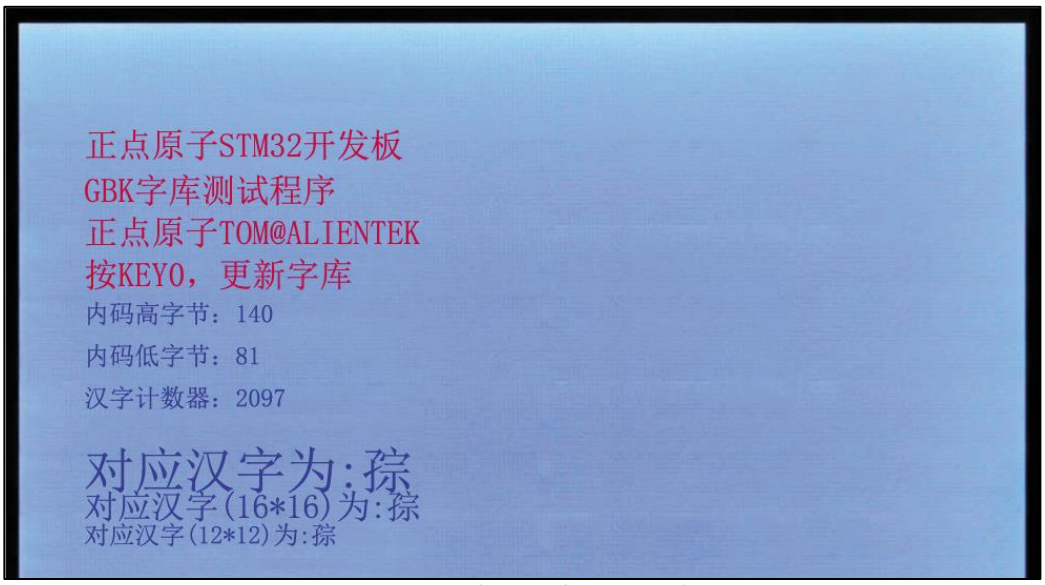


图 43.4.1 汉字显示实验显示效果

一开始如果没有显示汉字，代表 NORFLASH 里边没有字库文件，需要先找一个 U 盘，把 A 盘资料\5，U 盘根目录文件，文件夹下面的 SYSTEM 文件夹拷贝到 U 盘的根目录下，插入开发板，并按复位，就可以实现字库的更新，并且也支持在显示汉字的时候，按下 KEY0，可以从新开始更新字库。字库更新界面如图 43.4.2 所示：

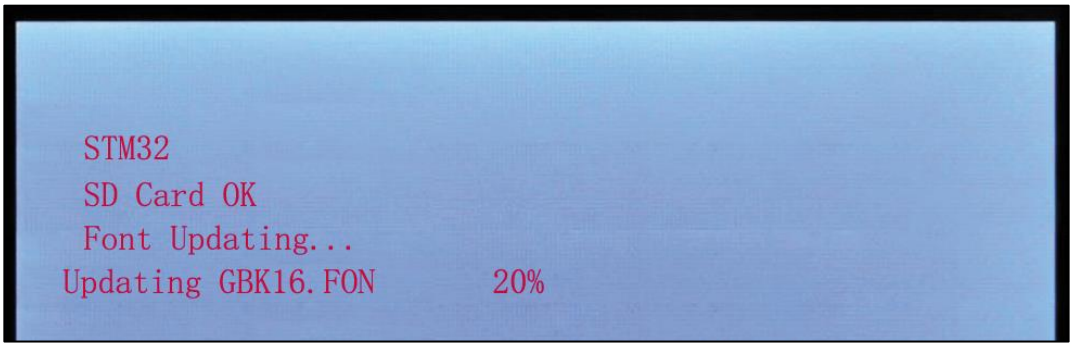


图 43.4.2 汉字字库更新界面

此外我们还可以使用 USMART 来测试该实验。通过 USMART 调用 text_show_string 或者 text_show_string_middle 来实现任意位置显示任何字符串，有兴趣的朋友可以尝试一下。