[野火]STM32 HAL 库开发实战指南
—基于 F103 系列开发板

EmbedFire
野火电子

2022 年 09 月 13 日

Contents

关于本项目 **1**
如何参与到本项目 1
关于野火 **2**
开源共享，共同进步 2
联系方式 2
TODO 和悬赏任务 **3**
第 1 章 如何使用本书 **4**
1.1 本书的学习顺序 4
1.2 本书的参考资料 4
1.3 本书的编写风格 4
1.4 本书的配套硬件 5
1.5 本书的技术论坛 9
第 2 章 如何安装 KEIL5 . . . . . **10**
2.1 温馨提示 10
2.2 获取 KEIL5 安装包 10
2.3 开始安装 KEIL5 11
2.4 安装 STM32 芯片包 16
第 3 章 如何用 DAP 仿真器下载程序 **19**
3.1 仿真器简介 19
3.2 硬件连接 19
3.3 仿真器配置 21
3.3.1 Debug 选项配置 22
3.3.2 Utilities 选项配置 23
3.3.3 Debug Settings 选项配置 24
3.4 选择目标板 25
3.5 下载程序 26

第 4 章 初识 STM32 27
4.1 什么是 STM32 27
4.1.1 STM32 诞生的背景 27
4.2 STM32 能做什么 28
4.2.1 智能手环 28
4.2.2 微型四轴飞行器 30
4.2.3 淘宝众筹 32
4.3 STM32 怎么选型 33
4.3.1 STM32 分类 33
4.3.2 STM32 命名方法 34
4.3.3 选择合适的 MCU 36
4.3.4 PCB 哪里打样 40
第 5 章 什么是寄存器 41
5.1 什么是寄存器 41
5.2 STM32 长啥样 41
5.3 芯片里面有什么 44
5.3.1 ICode 总线 45
5.3.2 驱动单元 45
5.3.3 被动单元 46
5.4 存储器映射 47
5.4.1 存储器映射 48
5.4.2 存储器区域功能划分 50
5.5 寄存器映射 52
5.5.1 STM32 的外设地址映射 54
5.5.2 C 语言对寄存器的封装 58
5.5.3 修改寄存器的位操作方法 63
第 6 章 新建工程—寄存器版 67
6.1 新建工程 67
6.1.1 新建本地工程文件夹 67
6.1.2 新建工程 68
6.2 下载程序 79

第 7 章 使用寄存器点亮 LED 灯 81
7.1 GPIO 简介 81
7.2 GPIO 框图剖析 82
7.2.1 基本结构分析 82
7.2.2 GPIO 工作模式 87
7.3 实验：使用寄存器点亮 LED 灯 91
7.3.1 硬件连接 92
7.3.2 启动文件 94
7.3.3 stm32f10x.h 文件 96
7.3.4 main 文件 99
7.3.5 下载验证 109
第 8 章 初识 HAL 固件库 110
8.1 CMSIS 标准及库层次关系 110
8.1.1 库目录、文件简介 112
8.1.2 库各文件间的关系 122
8.2 使用帮助文档 123
8.2.1 常用官方资料 123
第 9 章 使用 STM32CubeMX 新建工程 125
9.1 软件安装 125
9.1.1 安装 STM32CubeMX 软件 128
9.2 新建工程 133
9.2.1 新建工程 133
9.2.2 下载验证 144
第 10 章 新建工程—库函数版 145
10.1 新建工程 145
10.1.1 新建本地工程文件夹 145
10.1.2 新建工程 147
第 11 章 GPIO 输出—使用固件库点亮 LED 161
11.1 硬件设计 161
11.2 软件设计 163
11.2.1 编程要点 163

11.2.2 代码分析 . . . 163
11.2.3 下载验证 . . . 179
11.3 STM32 HAL 库补充知识 . . . 180
11.3.1 SystemInit 函数去哪了? . . . 180
11.3.2 断言 . . . 180
11.3.3 Doxygen 注释规范 . . . 182
11.3.4 防止头文件重复包含 . . . 183
第 12 章 GPIO 输入—按键检测 . . . 185
12.1 硬件设计 . . . 185
12.2 软件设计 . . . 187
12.2.1 编程要点 . . . 187
12.2.2 代码分析 . . . 187
12.2.3 下载验证 . . . 192
第 13 章 启动文件详解 . . . 193
13.1 启动文件简介 . . . 193
13.2 查找 ARM 汇编指令 . . . 193
13.3 启动文件代码讲解 . . . 195
13.3.1 Stack—栈 . . . 195
13.3.2 Heap 堆 . . . 196
13.3.3 向量表 . . . 197
13.3.4 复位程序 . . . 200
13.3.5 中断服务程序 . . . 201
13.3.6 用户堆栈初始化 . . . 202
第 14 章 RCC—使用 HSE/HSI 配置时钟 . . . 206
14.1 RCC 主要作用—时钟部分 . . . 206
14.2 RCC 框图剖析—时钟部分 . . . 206
14.2.1 系统时钟 . . . 208
14.2.2 其他时钟 . . . 212
14.3 配置系统时钟实验 . . . 213
14.3.1 使用 HSE . . . 213
14.3.2 使用 HSI . . . 214

14.3.3 硬件设计 . . . 214
14.3.4 软件设计 . . . 214
14.3.5 下载验证 . . . 220
第 15 章 STM32 中断应用概览 . . . 222
15.1 异常类型 . . . 222
15.2 NVIC 简介 . . . 224
15.2.1 NVIC 寄存器简介 . . . 225
15.2.2 NVIC 中断配置固件库 . . . 226
15.3 优先级的定义 . . . 226
15.3.1 优先级定义 . . . 226
15.3.2 优先级分组 . . . 227
15.4 中断编程 . . . 228
第 16 章 EXTI—外部中断/事件控制器 . . . 230
16.1 EXTI 简介 . . . 230
16.2 EXTI 功能框图 . . . 230
16.3 中断/事件线 . . . 233
16.4 EXTI 初始化详解 . . . 235
16.5 外部中断控制实验 . . . 235
16.5.1 硬件设计 . . . 236
16.5.2 软件设计 . . . 237
16.5.3 下载验证 . . . 246
第 17 章 SysTick—系统定时器 . . . 247
17.1 SysTick 简介 . . . 247
17.2 SysTick 寄存器介绍 . . . 247
17.3 SysTick 定时实验 . . . 250
17.3.1 硬件设计 . . . 250
17.3.2 软件设计 . . . 250
第 18 章 通讯的基本概念 . . . 259
18.1 串行通讯与并行通讯 . . . 259
18.2 全双工、半双工及单工通讯 . . . 261
18.3 同步通讯与异步通讯 . . . 263

18.4 通讯速率 264
第 19 章 USART—串口通讯 266
19.1 串口通讯协议简介 266
19.1.1 物理层 266
19.1.2 协议层 272
19.2 STM32 的 USART 简介 274
19.3 USART 功能框图 275
19.3.1 ① 功能引脚 277
19.3.2 ② 数据寄存器 278
19.3.3 ③ 控制器 278
19.3.4 ④ 小数波特率生成 280
19.3.5 校验控制 281
19.3.6 中断控制 281
19.4 USART 初始化结构体详解 282
19.4.1 USART 初始化结构体 282
19.5 USART1 接发通信实验 283
19.5.1 硬件设计 284
19.5.2 软件设计 285
19.5.3 下载验证 293
19.6 USART1 指令控制 LED 灯实验 294
19.6.1 硬件设计 294
19.6.2 软件设计 294
19.6.3 下载验证 302
第 20 章 DMA—直接存储区访问 303
20.1 DMA 简介 303
20.2 DMA 功能框图 303
20.2.1 ①DMA 请求 304
20.2.2 ① 通道 306
20.2.3 ② 仲裁器 306
20.3 DMA 数据配置 306
20.3.1 从哪里来到哪里去 306

20.3.2 要传多少，单位是什么 . . . 307
20.3.3 什么时候传输完成 . . . 308
20.4 DMA 初始化结构体详解 . . . 308
20.4.1 DMA_InitTypeDef 初始化结构体 . . . 309
20.5 DMA 存储器到存储器模式实验 . . . 310
20.5.1 硬件设计 . . . 310
20.5.2 软件设计 . . . 310
20.6 DMA 存储器到外设模式实验 . . . 316
20.6.1 硬件设计 . . . 316
20.6.2 软件设计 . . . 317
第 21 章 常用存储器介绍 . . . 322
21.1 存储器种类 . . . 322
21.2 RAM 存储器 . . . 323
21.2.1 DRAM . . . 323
21.2.2 SRAM . . . 324
21.2.3 DRAM 与 SRAM 的应用场合 . . . 325
21.3 非易失性存储器 . . . 326
21.3.1 ROM 存储器 . . . 326
21.3.2 FLASH 存储器 . . . 327
第 22 章 I2C—读写 EEPROM . . . 329
22.1 I2C 协议简介 . . . 329
22.1.1 I2C 物理层 . . . 329
22.1.2 协议层 . . . 331
22.2 STM32 的 I2C 特性及架构 . . . 336
22.2.1 STM32 的 I^2C 外设简介 . . . 337
22.2.2 STM32 的 I^2C 架构剖析 . . . 338
22.2.3 通讯过程 . . . 341
22.3 I2C 初始化结构体详解 . . . 343
22.4 I2C—读写 EEPROM 实验 . . . 345
22.4.1 硬件设计 . . . 346
22.4.2 软件设计 . . . 348

22.4.3 下载验证 395
第 23 章 SPI—读写串行 FLASH 397
23.1 SPI 协议简介 397
23.1.1 SPI 物理层 398
23.1.2 协议层 399
23.2 STM32 的 SPI 特性及架构 402
23.2.1 STM32 的 SPI 外设简介 403
23.2.2 STM32 的 SPI 架构剖析 403
23.2.3 通讯过程 406
23.3 SPI 初始化结构体详解 407
23.4 SPI—读写串行 FLASH 实验 409
23.4.1 硬件设计 409
23.4.2 软件设计 412
23.4.3 下载验证 446
第 24 章 串行 FLASH 文件系统 FatFs 447
24.1 文件系统 447
24.2 FatFs 文件系统简介 448
24.2.1 FatFs 的目录结构 449
24.2.2 FatFs 帮助文档 450
24.2.3 FATFS 源码 451
24.3 FatFs 文件系统移植实验 452
24.3.1 FatFs 程序结构图 452
24.3.2 硬件设计 453
24.3.3 FatFs 移植步骤 454
24.3.4 FatFs 底层设备驱动函数 457
24.3.5 FatFs 功能配置 467
24.3.6 FatFs 功能测试 467
24.3.7 下载验证 473
24.4 FatFs 功能使用实验 473
24.4.1 硬件设计 474
24.4.2 软件设计 474

24.4.3 下载验证 483
第 25 章 LCD—液晶显示 484
25.1 显示器简介 484
25.1.1 液晶显示器 484
25.1.2 LED 和 OLED 显示器 486
25.1.3 显示器的基本参数 489
25.2 液晶控制原理 490
25.2.1 液晶面板的控制信号 492
25.2.2 液晶数据传输时序 495
25.2.3 显存 497
25.3 野火 3.2 寸液晶屏简介 497
25.3.1 3.2 寸电阻触摸屏实物 497
25.3.2 ILI9341 液晶控制器简介 499
25.3.3 液晶屏的信号线及 8080 时序 501
25.4 使用 STM32 的 FSMC 模拟 8080 接口时序 504
25.4.1 FSMC 简介 505
25.5 FSMC 的地址映射 508
25.6 FSMC 控制异步 NOR FLASH 的时序 510
25.7 用 FSMC 模拟 8080 时序 513
25.8 NOR FLASH 时序结构体 515
25.9 FSMC 初始化结构体 516
25.10 FSMC—液晶显示实验 518
25.10.1 硬件设计 519
25.10.2 软件设计 523
25.10.3 下载验证 556
第 26 章 LCD—液晶显示中英文 557
26.1 字符编码 557
26.1.1 ASCII 编码 557
26.1.2 中文编码 561
26.1.3 Unicode 字符集和编码 567
26.1.4 UTF-32 567

26.1.5 UTF-16 . . . 568
26.1.6 UTF-8 . . . 569
26.1.7 BOM . . . 570
26.2 什么是字模? . . . 570
26.2.1 字模的构成 . . . 571
26.2.2 字模显示原理 . . . 572
26.2.3 如何制作字模 . . . 575
26.2.4 字模寻址公式 . . . 577
26.2.5 存储字模文件 . . . 578
26.3 各种模式的液晶显示字符实验 . . . 578
26.3.1 硬件设计 . . . 579
26.3.2 显示 ASCII 编码的字符 . . . 580
26.3.3 显示 GB2312 编码的字符 . . . 593
26.3.4 显示任意大小的字符 . . . 606
26.3.5 下载验证 . . . 618
第 27 章 电阻触摸屏—触摸画板 . . . 619
27.1 触摸屏简介 . . . 619
27.1.1 电阻式触摸屏检测原理 . . . 621
27.1.2 电阻触摸屏控制芯片 . . . 623
27.1.3 电容式触摸屏检测原理 . . . 624
27.2 电阻触摸屏—触摸画板实验 . . . 627
27.2.1 硬件设计 . . . 627
27.2.2 软件设计 . . . 630
27.2.3 下载验证 . . . 656
第 28 章 ADC—电压采集 . . . 657
28.1 ADC 简介 . . . 657
28.2 ADC 功能框图剖析 . . . 658
28.2.1 ① 电压输入范围 . . . 659
28.2.2 ② 输入通道 . . . 659
28.2.3 ③ 转换顺序 . . . 661
28.2.4 ④ 触发源 . . . 662

28.2.5 ⑤ 转换时间 . . . 663
28.2.6 ⑥ 数据寄存器 . . . 663
28.2.7 ⑦ 中断 . . . 664
28.2.8 ⑧ 电压转换 . . . 665
28.3 ADC 初始化结构体详解 . . . 665
28.3.1 ADC_InitTypeDef 结构体 . . . 666
28.4 独立模式单通道采集实验 . . . 667
28.4.1 硬件设计 . . . 667
28.4.2 软件设计 . . . 668
28.5 独立模式多通道采集实验 . . . 676
28.5.1 硬件设计 . . . 676
28.5.2 软件设计 . . . 676
28.6 双重 ADC 同步规则模式采集实验 . . . 684
28.6.1 硬件设计 . . . 686
28.6.2 软件设计 . . . 686
第 29 章 TIM—基本定时器 . . . 694
29.1 定时器分类 . . . 694
29.2 基本定时器功能框图讲解 . . . 695
29.2.1 ① 时钟源 . . . 695
29.2.2 ② 计数器时钟 . . . 696
29.2.3 ③ 计数器 . . . 696
29.2.4 ④ 自动重装载寄存器 . . . 696
29.2.5 定时时间的计算 . . . 696
29.3 定时器初始化结构体详解 . . . 697
29.4 基本定时器定时实验 . . . 698
29.4.1 硬件设计 . . . 698
29.4.2 软件设计 . . . 698
29.4.3 下载验证 . . . 701
第 30 章 TIM—高级定时器 . . . 702
30.1 高级控制定时器 . . . 702
30.2 高级控制定时器功能框图 . . . 703

30.2.1 ① 时钟源 . . . 704
30.2.2 ② 控制器 . . . 708
30.2.3 ③ 时基单元 . . . 708
30.2.4 ④ 输入捕获 . . . 710
30.2.5 ⑤ 输出比较 . . . 713
30.2.6 ⑥ 断路功能 . . . 717
30.3 输入捕获应用 . . . 718
30.3.1 测量脉宽或者频率 . . . 718
30.3.2 PWM 输入模式 . . . 719
30.4 输出比较应用 . . . 721
30.4.1 PWM 输出模式 . . . 721
30.5 定时器初始化结构体详解 . . . 723
30.5.1 TIM_TimeBaseInitTypeDef . . . 724
30.5.2 TIM_OC_InitTypeDef . . . 725
30.5.3 TIM_IC_InitTypeDef . . . 726
30.5.4 TIM_BreakDeadTimeConfigTypeDef . . . 728
30.6 PWM 互补输出实验 . . . 729
30.6.1 硬件设计 . . . 729
30.6.2 软件设计 . . . 730
30.7 脉宽测量输入捕获实验 . . . 736
30.7.1 硬件设计 . . . 737
30.7.2 软件设计 . . . 737
30.7.3 下载验证 . . . 743
第 31 章 TIM—电容按键检测 . . . 744
31.1 电容按键原理 . . . 744
31.2 电容按键检测实验 . . . 747
31.2.1 硬件设计 . . . 748
31.2.2 软件设计 . . . 748
31.2.3 下载验证 . . . 758
第 32 章 IWDG—独立看门狗 . . . 759
32.1 IWDG 简介 . . . 759

32.2 IWDG 功能框图剖析 . . . 759
32.2.1 ① 独立看门狗时钟 . . . 760
32.2.2 ② 计数器时钟 . . . 760
32.2.3 ③ 计数器 . . . 760
32.2.4 ④ 重装载寄存器 . . . 760
32.2.5 ⑤ 键寄存器 . . . 761
32.2.6 ⑥ 状态寄存器 . . . 761
32.3 怎么用 IWDG . . . 761
32.4 IWDG 超时实验 . . . 762
32.4.1 硬件设计 . . . 762
32.4.2 软件设计 . . . 762
32.4.3 下载验证 . . . 767
第 33 章 WWDG—窗口看门狗 . . . 768
33.1 WWDG 简介 . . . 768
33.2 WWDG 功能框图剖析 . . . 770
33.2.1 ① 窗口看门狗时钟 . . . 770
33.2.2 ② 计数器时钟 . . . 770
33.2.3 ③ 计数器 . . . 771
33.2.4 ④ 窗口值 . . . 771
33.2.5 ⑤ 计算看门狗超时时间 . . . 772
33.3 怎么用 WWDG . . . 773
33.4 WWDG 喂狗实验 . . . 773
33.4.1 硬件设计 . . . 773
33.4.2 软件设计 . . . 773
第 34 章 SDIO—SD 卡读写测试 . . . 779
34.1 SDIO 简介 . . . 779
34.2 SD 卡物理结构 . . . 781
34.3 SDIO 总线 . . . 783
34.3.1 总线拓扑 . . . 783
34.3.2 总线协议 . . . 785
34.3.3 命令 . . . 788

34.3.4 响应 . . . 795
34.4 SD 卡的操作模式及切换 . . . 797
34.4.1 SD 卡的操作模式 . . . 797
34.4.2 卡识别模式 . . . 798
34.4.3 数据传输模式 . . . 799
34.5 STM32 的 SDIO 功能框图 . . . 801
34.5.1 控制单元 . . . 802
34.5.2 命令路径 . . . 803
34.5.3 数据路径 . . . 804
34.5.4 数据 FIFO . . . 806
34.5.5 适配器寄存器 . . . 807
34.6 SDIO 初始化结构体 . . . 807
34.7 SDIO 命令初始化结构体 . . . 808
34.8 SDIO 数据初始化结构体 . . . 809
34.9 SD 卡读写测试实验 . . . 810
34.9.1 硬件设计 . . . 811
34.9.2 软件设计 . . . 812
34.9.3 下载验证 . . . 855
第 35 章 基于 SD 卡的 FatFs 文件系统 . . . 856
35.1 FatFs 移植步骤 . . . 856
35.2 FatFs 接口函数 . . . 859
35.2.1 宏定义和存储设备状态获取函数 . . . 860
35.2.2 存储设备初始化函数 . . . 860
35.2.3 存储设备数据读取函数 . . . 861
35.2.4 存储设备数据写入函数 . . . 862
35.2.5 其他控制函数 . . . 864
35.3 FatFs 功能测试 . . . 865
35.3.1 变量定义 . . . 865
35.3.2 主函数 . . . 866
35.3.3 下载验证 . . . 871

第 36 章 DAC—输出正弦波 872
36.1 DAC 简介 872
36.2 DAC 功能框图剖析 872
36.2.1 参考电压 874
36.2.2 数模转换及输出通道 874
36.2.3 触发源及 DHRx 寄存器 874
36.3 DAC 初始化结构体详解 875
36.4 DAC 输出正弦波实验 876
36.4.1 硬件设计 876
36.4.2 软件设计 877
36.4.3 下载验证 887
36.5 课后练习 888
第 37 章 电源管理—实现低功耗 889
37.1 电源管理相关的库函数及命令 889
37.1.1 配置 PVD 监控功能 889
37.1.2 WFI 与 WFE 命令 889
37.1.3 进入停止模式 890
37.1.4 进入待机模式 892
37.2 PWR—睡眠模式实验 894
37.2.1 硬件设计 894
37.2.2 软件设计 894
37.2.3 下载验证 899
37.3 PWR—停止模式实验 899
37.3.1 硬件设计 899
37.3.2 软件设计 899
37.3.3 下载验证 906
37.4 STM32 的电源管理简介 906
37.4.1 电源监控器 907
37.4.2 STM32 的电源系统 908
37.4.3 STM32 的功耗模式 911
37.5 PWR—待机模式实验 915

37.5.1 硬件设计 . . . 916
37.5.2 软件设计 . . . 916
37.6 PWR—PVD 电源监控实验 . . . 920
37.6.1 硬件设计 . . . 920
37.6.2 软件设计 . . . 922
37.6.3 下载验证 . . . 925
第 38 章 RTC—实时时钟 . . . 927
38.1 RTC 实时时钟简介 . . . 927
38.2 RTC 外设框图剖析 . . . 928
38.3 UNIX 时间戳 . . . 929
38.4 与 RTC 控制相关的库函数 . . . 930
38.4.1 初始化结构体讲解 . . . 931
38.4.2 RTC 时间结构体讲解 . . . 931
38.4.3 RTC 日期结构体讲解 . . . 933
38.4.4 RTC 闹钟结构体 . . . 934
38.5 利用 RTC 提供北京时间 . . . 935
38.5.1 硬件设计 . . . 935
38.5.2 软件设计 . . . 937
38.5.3 下载验证 . . . 947
第 39 章 MDK 的编译过程及文件类型全解 . . . 948
39.1 编译过程 . . . 948
39.1.1 编译过程简介 . . . 948
39.1.2 具体工程中的编译过程 . . . 949
39.2 程序的组成、存储与运行 . . . 953
39.2.1 CODE、RO、RW、ZI Data 域及堆栈空间 . . . 953
39.2.2 程序的存储与运行 . . . 954
39.3 编译工具链 . . . 956
39.3.1 设置环境变量 . . . 956
39.3.2 armcc、armasm 及 armlink . . . 959
39.3.3 armar、fromelf 及用户指令 . . . 967
39.4 MDK 工程的文件类型 . . . 969

39.4.1 uvprojx、uvoptx 及 uvguix 工程文件 . . . 970
39.4.2 源文件 . . . 974
39.4.3 Output 目录下生成的文件 . . . 975
39.4.4 Listing 目录下的文件 . . . 1011
39.4.5 sct 分散加载文件的格式与应用 . . . 1021
39.5 实验自动分配变量到指定的 SRAM 空间 . . . 1036
39.5.1 补充关于“__attribute__”关键字的说明 . . . 1036
39.5.2 硬件设计 . . . 1037
39.5.3 软件设计 . . . 1037
39.6 实验：优先使用内部 SRAM 并把堆区分配到指定空间 . . . 1049
39.6.1 硬件设计 . . . 1049
39.6.2 软件设计 . . . 1050
第 40 章 在 SRAM 中调试代码 . . . 1062
40.1 在 RAM 中调试代码 . . . 1062
40.2 STM32 的启动方式 . . . 1063
40.3 内部 FLASH 的启动过程 . . . 1065
40.4 实验：在内部 SRAM 中调试代码 . . . 1069
40.4.1 硬件设计 . . . 1069
40.4.2 软件设计 . . . 1071
第 41 章 读写内部 FLASH . . . 1088
41.1 STM32 的内部 FLASH 简介 . . . 1088
41.1.1 内部 FLASH 的构成 . . . 1090
41.2 对内部 FLASH 的写入过程 . . . 1092
41.2.1 解锁 . . . 1092
41.2.2 页擦除 . . . 1092
41.2.3 写入数据 . . . 1093
41.3 查看工程的空间分布 . . . 1093
41.3.1 程序 ROM 的加载与执行空间 . . . 1095
41.3.2 ROM 空间分布表 . . . 1096
41.4 操作内部 FLASH 的库函数 . . . 1096
41.4.1 FLASH 解锁、上锁函数 . . . 1096

41.4.2 设置操作位数及页擦除 1098
41.4.3 写入数据 1100
41.5 实验：读写内部 FLASH 1103
41.5.1 硬件设计 1103
41.5.2 软件设计 1104
41.5.3 下载验证 1108
第 42 章 设置 FLASH 的读写保护及解除 1109
42.1 选项字节与读写保护 1109
42.1.1 选项字节的内容 1109
42.1.2 RDP 读保护 1112
42.1.3 WRP 写保护 1113
42.2 修改选项字节的过程 1114
42.3 操作选项字节的库函数 1114
42.3.1 选项字节上锁、解锁 1115
42.3.2 设置选项字节函数 1116
42.3.3 写入选项字节 1119
42.4 实验：设置读写保护及解除 1119
42.4.1 硬件设计 1120
42.4.2 软件设计 1120
42.4.3 下载测试 1127
第 43 章 MPU6050 传感器—姿态检测 1128
43.1 姿态检测 1128
43.1.1 基本认识 1128
43.1.2 坐标系 1129
43.1.3 利用陀螺仪检测角度 1131
43.1.4 利用加速度计检测角度 1133
43.1.5 利用磁场检测角度 1135
43.1.6 利用 GPS 检测角度 1135
43.1.7 姿态融合与四元数 1135
43.2 传感器 1136
43.2.1 传感器工作原理 1136

43.2.2 传感器参数 1136
43.2.3 物理量的表示方法 1137
43.3 MPU6050 模块简介 1138
43.3.1 MPU6050 模块功能及外观 1138
43.3.2 MPU6050 模块的引脚功能说明 1140
43.3.3 MPU6050 模块的硬件原理图 1141
43.4 MPU6050 的特性参数 1142
43.5 MPU6050—获取原始数据实验 1144
43.5.1 硬件设计 1145
43.5.2 软件设计 1146
43.6 MPU6050—利用 DMP 进行姿态解算 1159
43.6.1 硬件设计 1159
43.6.2 软件设计 1159
43.7 MPU6050—使用第三方上位机 1184
43.7.1 硬件设计 1185
43.7.2 软件设计 1185
第 44 章 OV7725 摄像头驱动 1196
44.1 摄像头简介 1196
44.1.1 数字摄像头跟模拟摄像头区别 1196
44.1.2 CCD 与 CMOS 的区别 1197
44.2 OV7725 摄像头 1198
44.2.1 OV7725 传感器简介 1198
44.2.2 OV7725 引脚及功能框架图 1199
44.2.3 SCCB 时序 1201
44.2.4 OV7725 的寄存器 1203
44.2.5 像素数据输出时序 1205
44.2.6 FIFO 读写时序 1208
44.2.7 摄像头的驱动原理 1211
44.3 摄像头驱动实验 1218
44.3.1 硬件设计 1218
44.3.2 软件设计 1220

联系我们 **1256**
rst 投稿示例文件 **1257**
如何参与项目 **1258**
项目源码 1258
文档在线阅读与 pdf 1258
常见问题 **1259**
版权说明 **1260**

关于本项目

点击右侧链接可在线阅读本项目文档：《[野火]STM32 HAL 库开发实战指南——基于 F103 系列开发板》

本书主要面向无 MCU 开发经验，希望从零开始学习 STM32 开发的开发者及在校学生。

如何参与到本项目

如果您想参与到项目，可按 如何参与野火开源项目 的说明操作。

关于野火

开源共享，共同进步

野火在发布第一块 STM32 开发板之初，就喊出 **开源共享，共同进步**的口号，把代码和文档教程都免费提供给用户下载，而我们也一直把这个理念贯穿至今。

目前我们的产品已经包括 *STM32*、*i.MX RT* 系列、*GD32V*、*FPGA*、*Linux*、*emXGUI*、操作系统、网络、下载器等分支，覆盖电子工程应用领域的各种常用技术，其中教学类产品的代码和文档一直保持着开源的姿态发布到网络上，为电子工程师排忧解难，让嵌入式没有难用的技术是我们最大的愿望。

联系方式

- 官网：http://www.embedfire.com
- 论坛：http://www.firebbs.cn
- 资料中心：http://doc.embedfire.com
- github 主页：https://github.com/Embedfire
- gitee 主页：https://gitee.com/Embedfire
- 天猫：https://yehuosm.tmall.com
- 淘宝：https://fire-stm32.taobao.com
- 邮箱：embedfire@embedfire.com
- 电话：0769-33894118

TODO 和悬赏任务

本页记录待改进的事项，以后会在此页发布任务清单，完成任务可获得 **奖励**，敬请期待！

如果您想参与到项目，可按 如何参与野火开源项目 的说明操作。

- 本文档正处于完善优化阶段

野火其它的 STM32 固件库、RTOS 和 GUI 的书籍

第 1 章　如何使用本书

1.1 本书的学习顺序

本书分为基础入门篇和提高篇，入门篇需要按照顺序学习，讲究循序渐进，步步为营。学习完基础篇之后，应该算基本入门 M3。提高篇属于高级例程，学习的时候并不一定要按照书中的章节排序，可根据需要跳跃式的学习。

1.2 本书的参考资料

本书参考资料为：《STM32F10X-中文参考手册》和《Cortex-M3 权威指南》，这两本是 ST 官方的手册，属于精华版，面面俱到，无所不包。限于篇幅问题，本书不可能面面俱到，而是着重于框图分析和代码讲解，有关寄存器的详细描述则略过，在学习本书的时候，涉及到寄存器描述部分还请参考这两本手册，这样学习效果会更佳。

1.3 本书的编写风格

本书着重讲解 F103 的外设以及外设的应用，力争全面分析每个外设的功能框图和外设的使用方法，让读者可以零死角的玩转 STM32 F103 系列。基本每个章节对应一个外设，每章的主要内容大概分为三个部分，第一部分为简介，第二部分为外设功能框图分析，第三部分为代码讲解。

外设简介则是用自己的话把外设概括性的介绍一遍，力图语句简短，通俗易懂，并不会完全照抄数据手册的介绍。

外设功能框图分析则是章节的重点，该部分会详细讲解功能框图的每个部分的作用，这是学习 F103 的精髓所在，掌握了整个外设的框图则可以熟练的使用该外设，熟练的编程，日后学习其他型号的单片机，也将会得心应手。因为即使单片机的型号不同，外设的框图还是基本一样的。这一步的学习比较枯燥，但是必须死磕，方能达成所愿。

代码分析则是讲解使用该外设的实验讲解，主要分析代码流程，和一些编程的注意事项。在掌握了框图之后，代码部分则是手到擒来而已。

1.4 本书的配套硬件

本书配套的硬件平台为野火的 STM32 F1 开发板，包括：F103-MINI、F103-指南者、F103-霸道开发板，还有 F103-拂晓电机开发板，学习的时候如果配套这些当中的其中一种硬件平台做实验，学习必会达到事半功倍的效果，可以省去中间移植时遇到的各种问题。

图 1-1 野火 STM32F103-MINI

Wildfire OS
00:11:01
KEY
Breathing
ADC
EEPROM
Clock
USB
Wifi
Humiture
Camera
Calculator
UserApp
UserApp
RESET
K1
K2
电容按键
WIFI
ON
OFF
JTAG
ARM-OB
USB TO UART
USB DEVICE
SDIO
SPI1
I2C1/2-USART2/3

图 1-2 野火 STM32F103-指南者

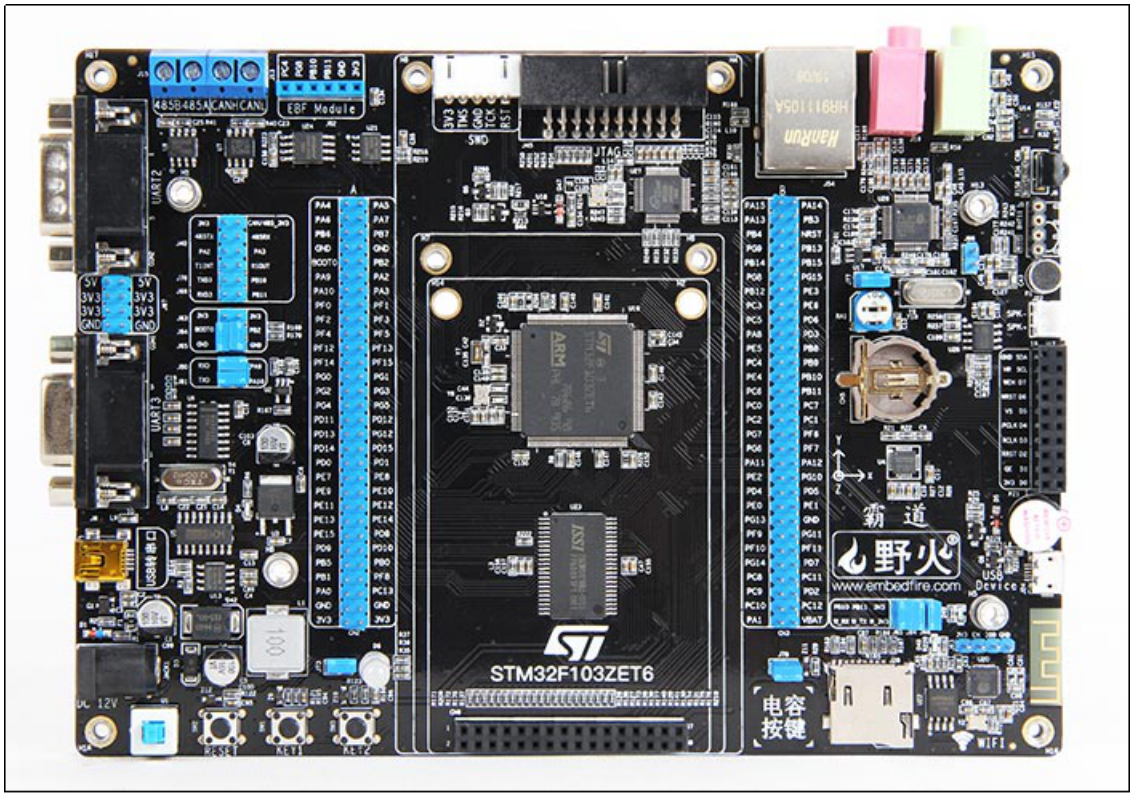


图 1-3 野火 STM32F103-霸道

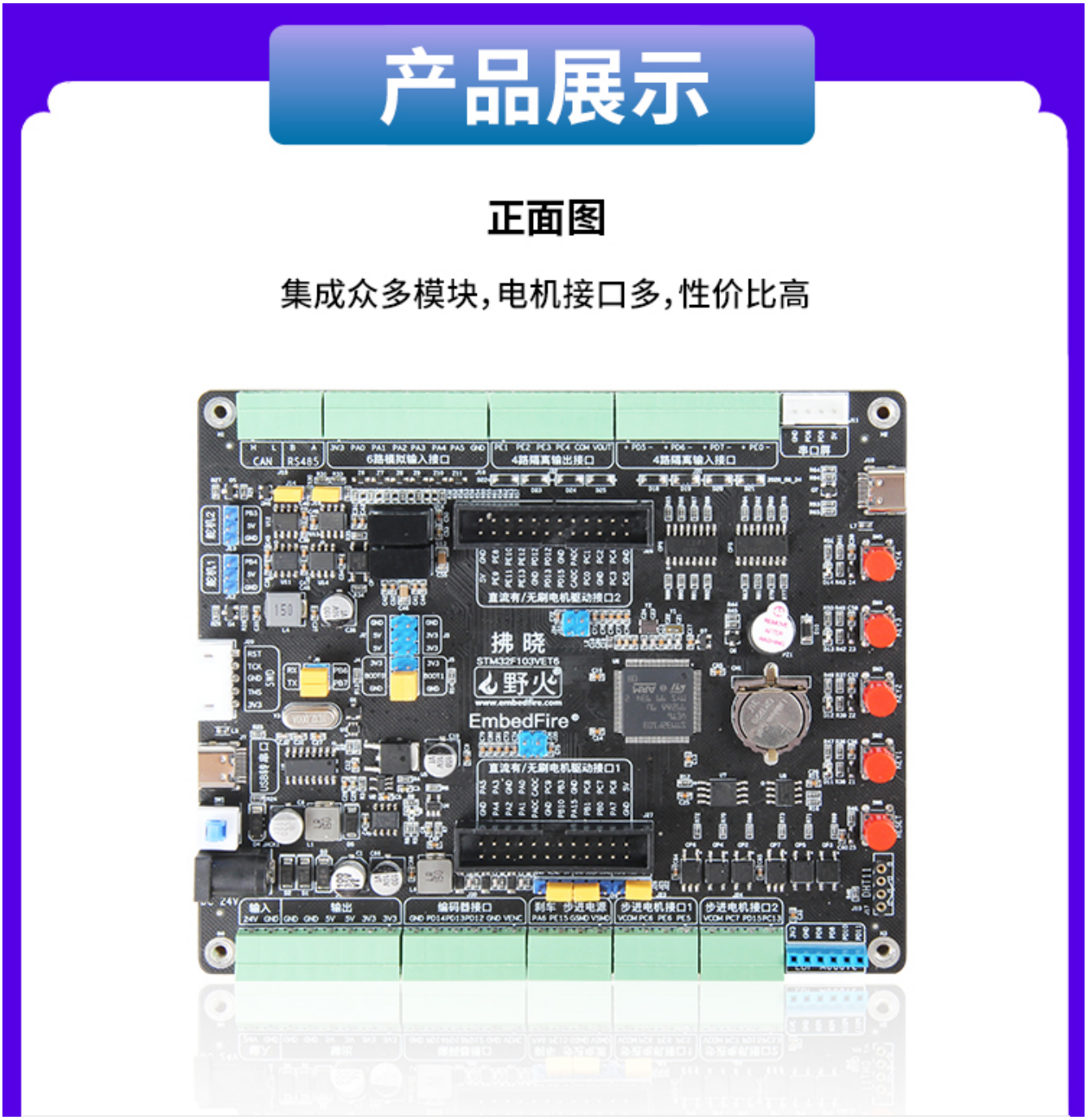


图 1-4 野火 STM32F103-拂晓

1.5 本书的技术论坛

如果在学习过程中遇到问题，可以到论坛：www.firebbs.cn 发帖交流，开源共享，共同进步。

鉴于水平有限，本书难免有纰漏，热心的读者也可把勘误发到论坛好让我们改进做得更好，祝您学习愉快，M3 的世界，野火与您同行。

第 2 章　如何安装 KEIL5

本章内容所涉及的软件只供教学使用，不得用于商业用途。个人或公司因商业用途导致的法律责任，后果自负。

2.1 温馨提示

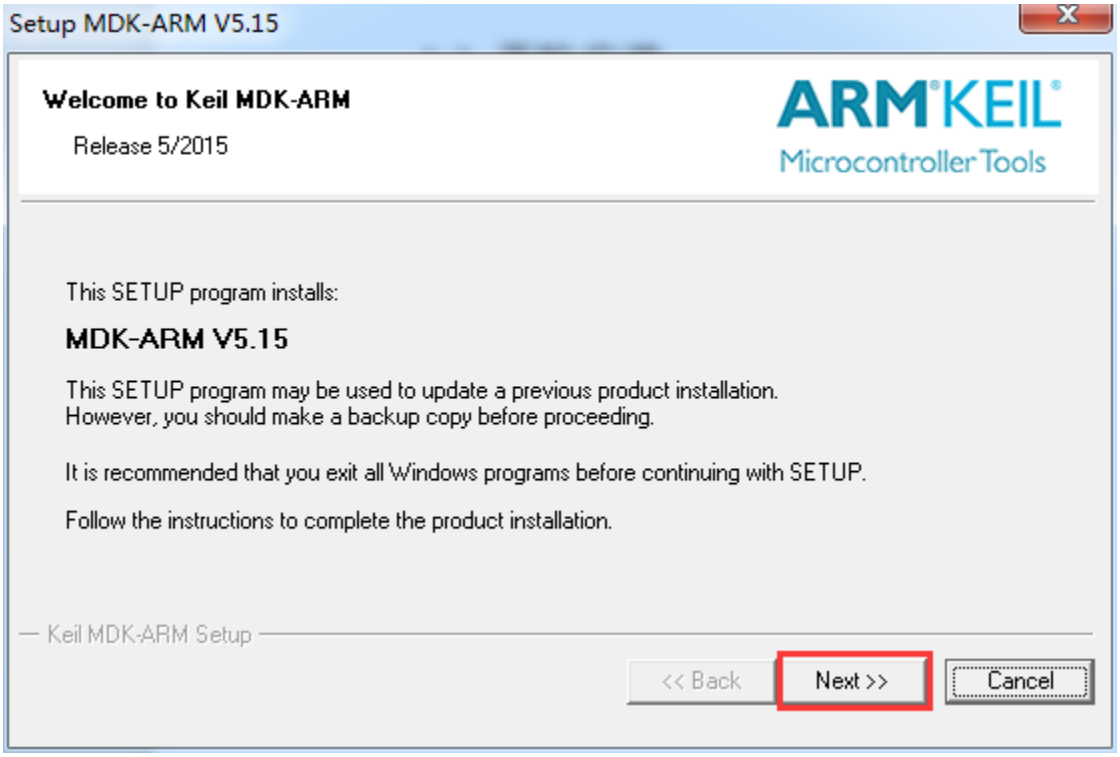
1. 安装路径不能带中文，必须是英文路径
2. 安装目录不能跟 51 的 KEIL 或者 KEIL4 冲突，三者目录必须分开
3. KEIL5 的安装比起 KEIL4 多了一个步骤，必须添加 MCU 库，不然没法使用。
4. 如果使用的时候出现莫名其妙的错误，先百度查找解决方法，莫乱阵脚。

2.2 获取 KEIL5 安装包

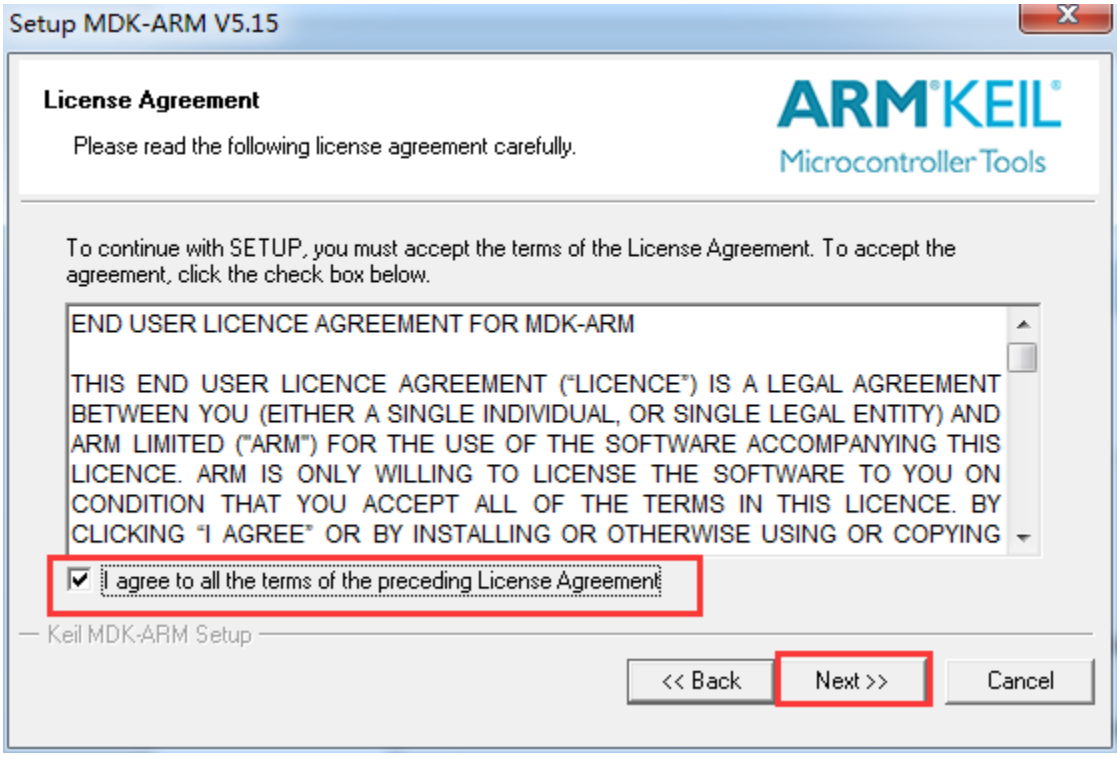
要想获得 KEIL5 的安装包，在百度里面搜索“KEIL5 下载”即可找到很多网友提供的下载文件，或者到 KEIL 的官网下载：https://www.keil.com/download/product/，一大堆注册非常麻烦。我们这里面 KEIL5 的版本是 MDK5.15，以后有新版本大家可使用更高版本。

2.3 开始安装 KEIL5

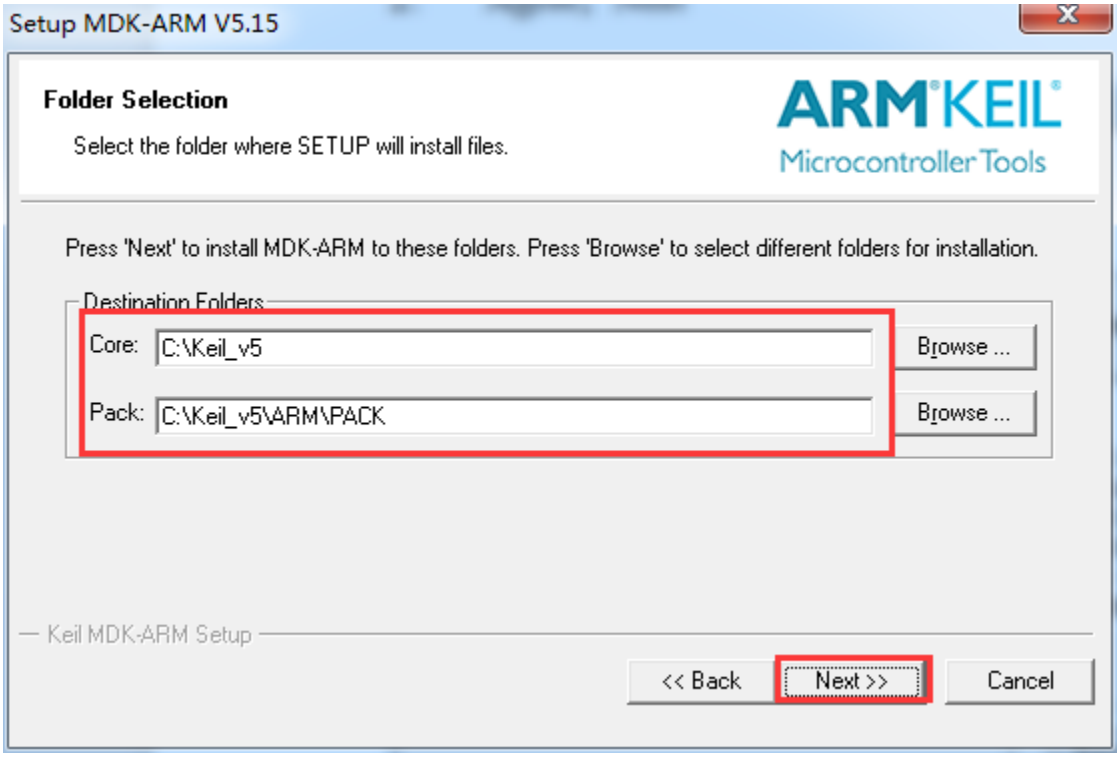
双击 KEIL5 安装包，开始安装，点 Next。



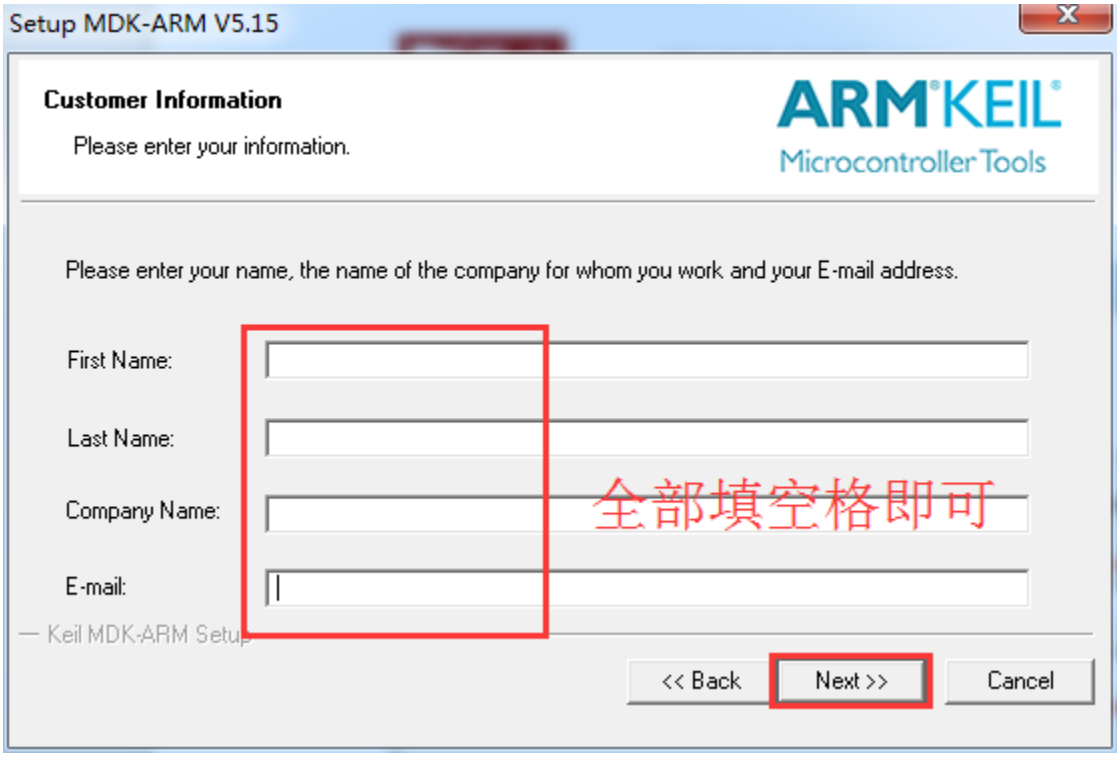
勾选 I agree，Next



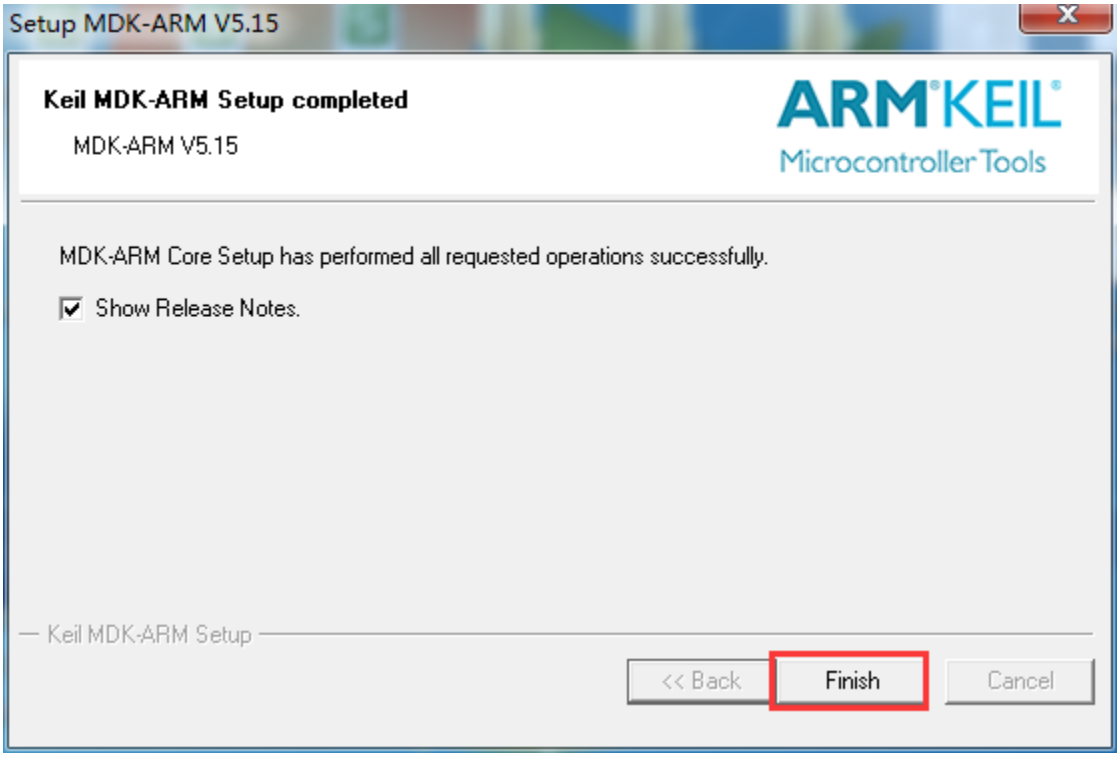
选择安装路径，Core 一栏的路径指的是 MDK-ARM 软件（即 Keil5）的安装根路径；后面我们还要安装芯片包，Pack 一栏的路径指的就是芯片包的安装路径。如果要修改路径的话，路径不能带中文，然后点 Next。



填写用户信息，全部填空格（键盘的 space 键）即可，Next


Setup MDK-ARM V5.15
Customer Information
Please enter your information.
ARM KEIL
Microcontroller Tools
Please enter your name, the name of the company for whom you work and your E-mail address.
First Name:
Last Name:
Company Name:
E-mail:
全部填空格即可
Keil MDK-ARM Setup
<< Back
Next >>
Cancel

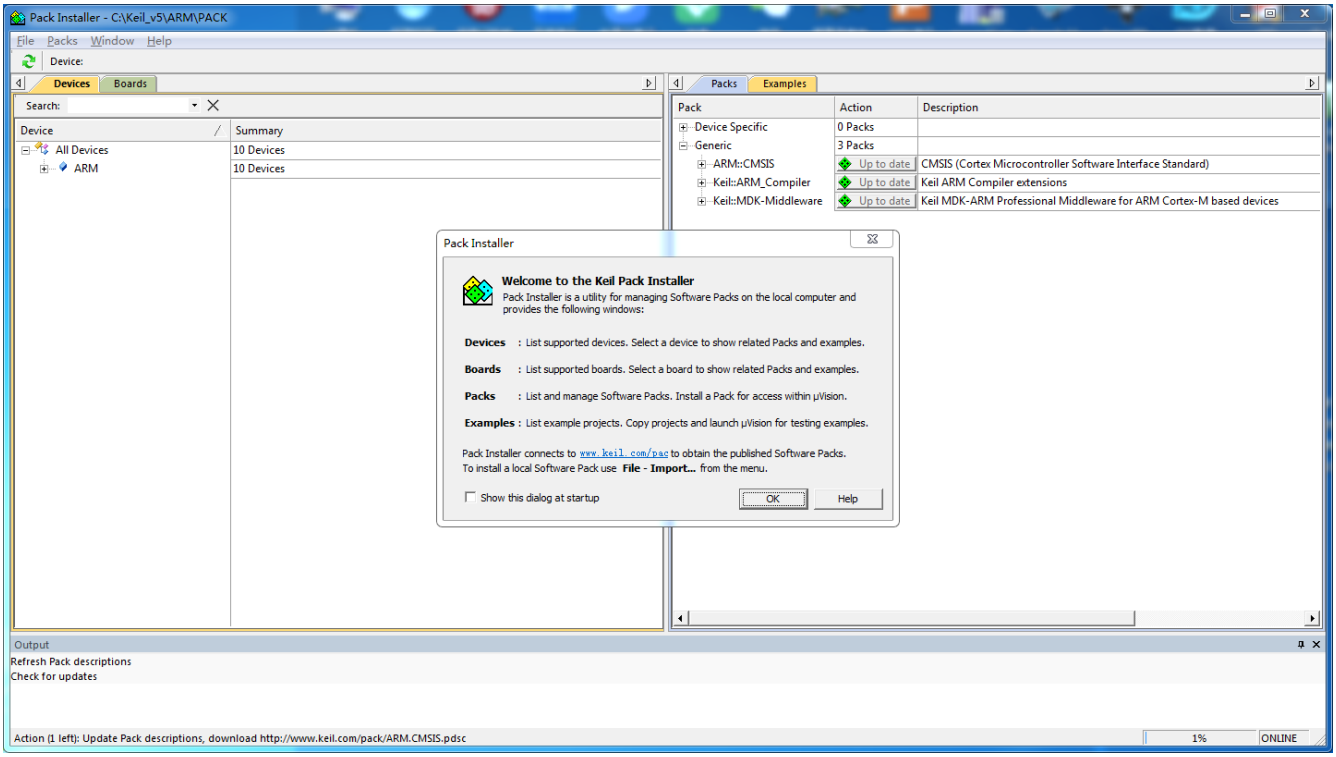
Finish，安装完毕



2.4 安装 STM32 芯片包

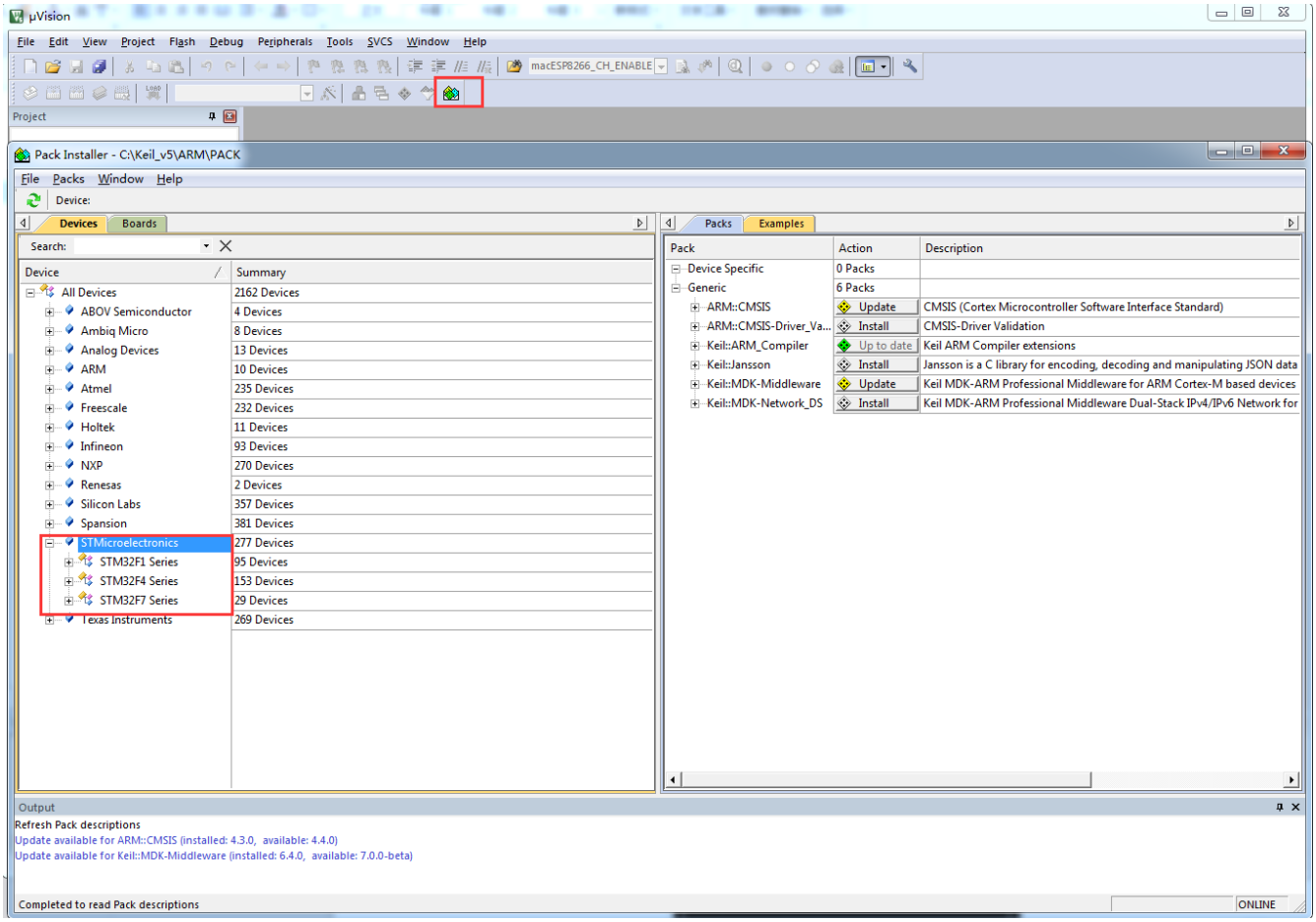
KEIL5 不像 KEIL4 那样自带了很多厂商的 MCU 型号，KEIL5 需要自己安装芯片包。

把下面弹出的界面关掉，我们直接去 keil 的官网下载：http://www.keil.com/dd2/pack/，或者直接用我们下载好的包。



在官网中找到 STM32F1、STM32F4、STM32F7 这 3 个系列的包下载到本地电脑，具体下载哪个系列的根据你使用的型号下载即可，这里我只下载我自己需要使用的 F1/4/7 这三个系列的包，F1 代表 M3，F4 代表 M4，F7 代表 M7。

把下载好的包双击安装即可，安装路径选择跟 KEIL5 一样的安装路径，安装成功之后，在 KEIL5 的 Pack Installer 中就可以看到我们安装的包，以后我们新建工程的时候，就有单片机的型号可选。


µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
macESP8266_CH_ENABLE
Project
Pack Installer - C:\Keil_v5\ARM\PACK
File Packs Window Help
Device:
Devices Boards
Search:
Device
Summary
All Devices 2162 Devices
ABOV Semiconductor 4 Devices
Ambiq Micro 8 Devices
Analog Devices 13 Devices
ARM 10 Devices
Atmel 235 Devices
Freescale 232 Devices
Holtek 11 Devices
Infineon 93 Devices
NXP 270 Devices
Renesas 2 Devices
Silicon Labs 357 Devices
Spansion 381 Devices
STMicroelectronics 277 Devices
STM32F1 Series 95 Devices
STM32F4 Series 153 Devices
STM32F7 Series 29 Devices
Texas Instruments 269 Devices
Packs Examples
Pack
Action
Description
Device Specific 0 Packs
Generic 6 Packs
ARM::CMSIS Update CMSIS (Cortex Microcontroller Software Interface Standard)
ARM::CMSIS-Driver_Va... Install CMSIS-Driver Validation
Keil::ARM_Compiler Up to date Keil ARM Compiler extensions
Keil::Jansson Install Jansson is a C library for encoding, decoding and manipulating JSON data
Keil::MDK-Middleware Update Keil MDK-ARM Professional Middleware for ARM Cortex-M based devices
Keil::MDK-Network_DS Install Keil MDK-ARM Professional Middleware Dual-Stack IPv4/IPv6 Network for
Output
Refresh Pack descriptions
Update available for ARM::CMSIS (installed: 4.3.0, available: 4.4.0)
Update available for Keil::MDK-Middleware (installed: 6.4.0, available: 7.0.0-beta)
Completed to read Pack descriptions
ONLINE

第 3 章　如何用 DAP 仿真器下载程序

3.1 仿真器简介

本书配套的仿真器为 Fire-Debugger，遵循 ARM 公司的 CMSIS-DAP 标准，支持所有基于 Cortex-M 内核的单片机，常见的 M3、M4 和 M7 都可以完美支持。

Fire-Debugger 支持下载和在线仿真程序，支持 XP/WIN7/WIN8/WIN10 这四个操作系统，免驱，不需要安装驱动即可使用，支持 KEIL 和 IAR 直接下载，非常方便。

3.2 硬件连接

把仿真器用 USB 线连接电脑，如果仿真器的灯亮则表示正常，可以使用。然后把仿真器的另外一端连接到开发板，给开发板上电，然后就可以通过软件 KEIL 或者 IAR 给开发板下载程序。

图 3-1 仿真器与电脑和开发板连接方式

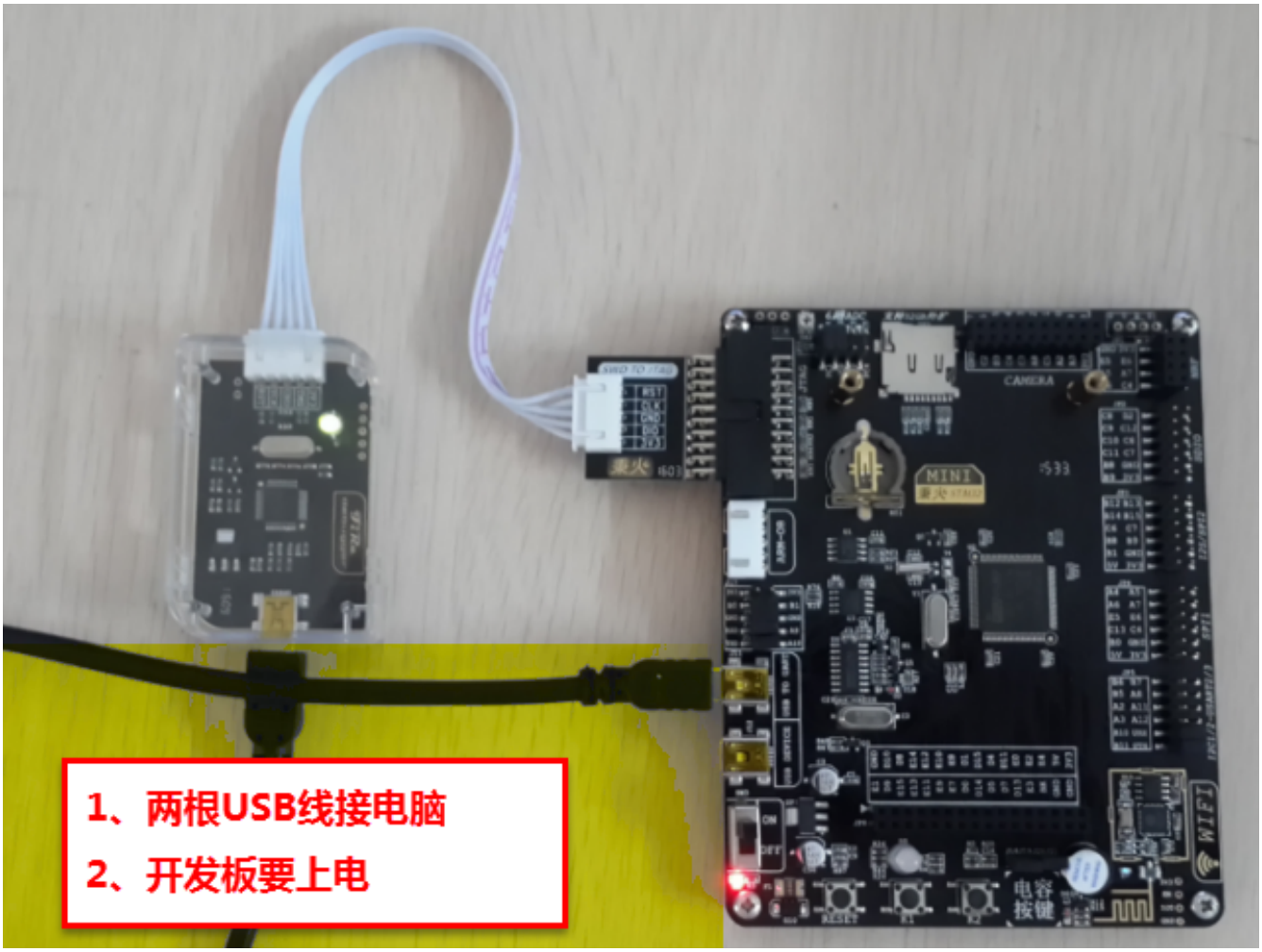


图 3-2 仿真器与指南者连接图

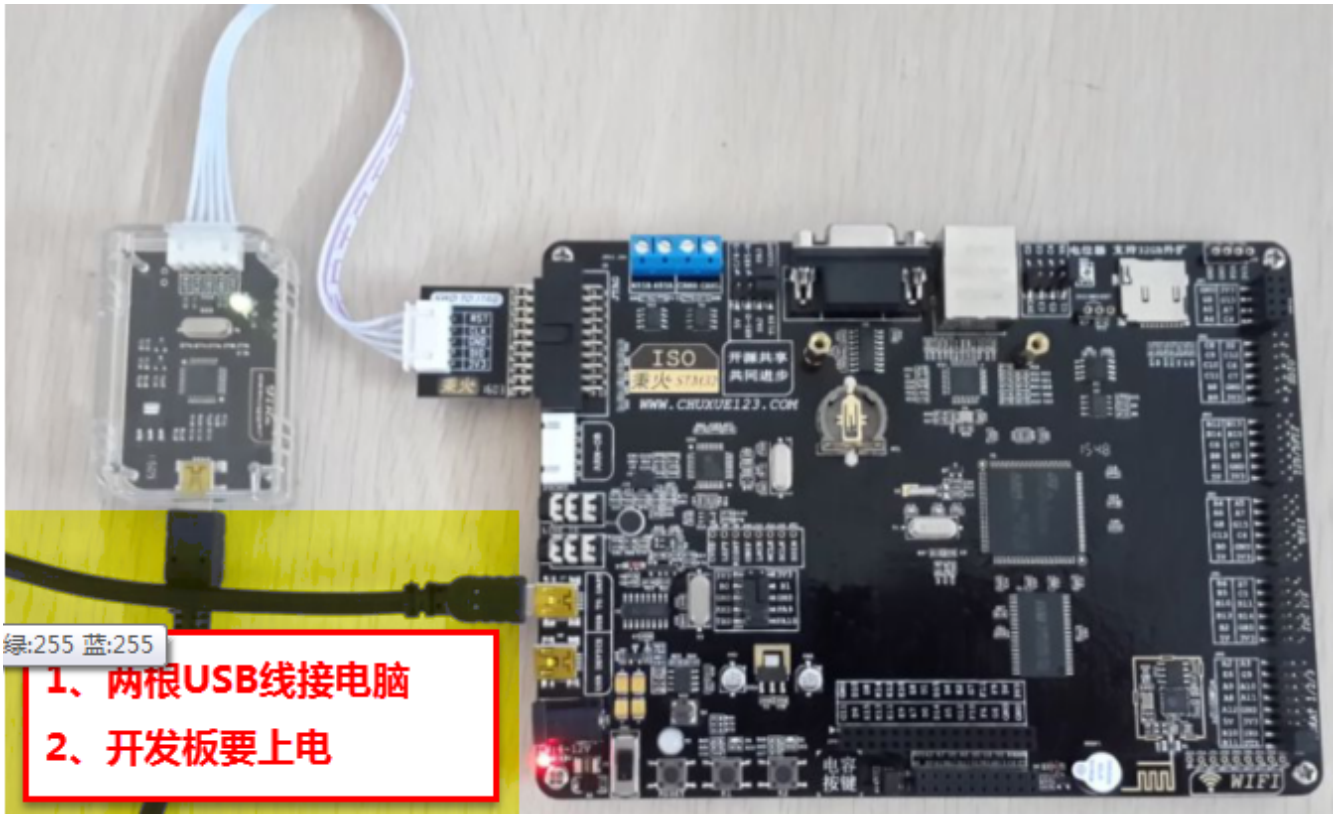


图 3-3 仿真器与指南者连接图

3.3 仿真器配置

在仿真器连接好电脑和开发板且开发板供电正常的情况下，打开编译软件 KEIL，在魔术棒选项卡里面选择仿真器的型号，具体过程看图示：

3.3.1 Debug 选项配置

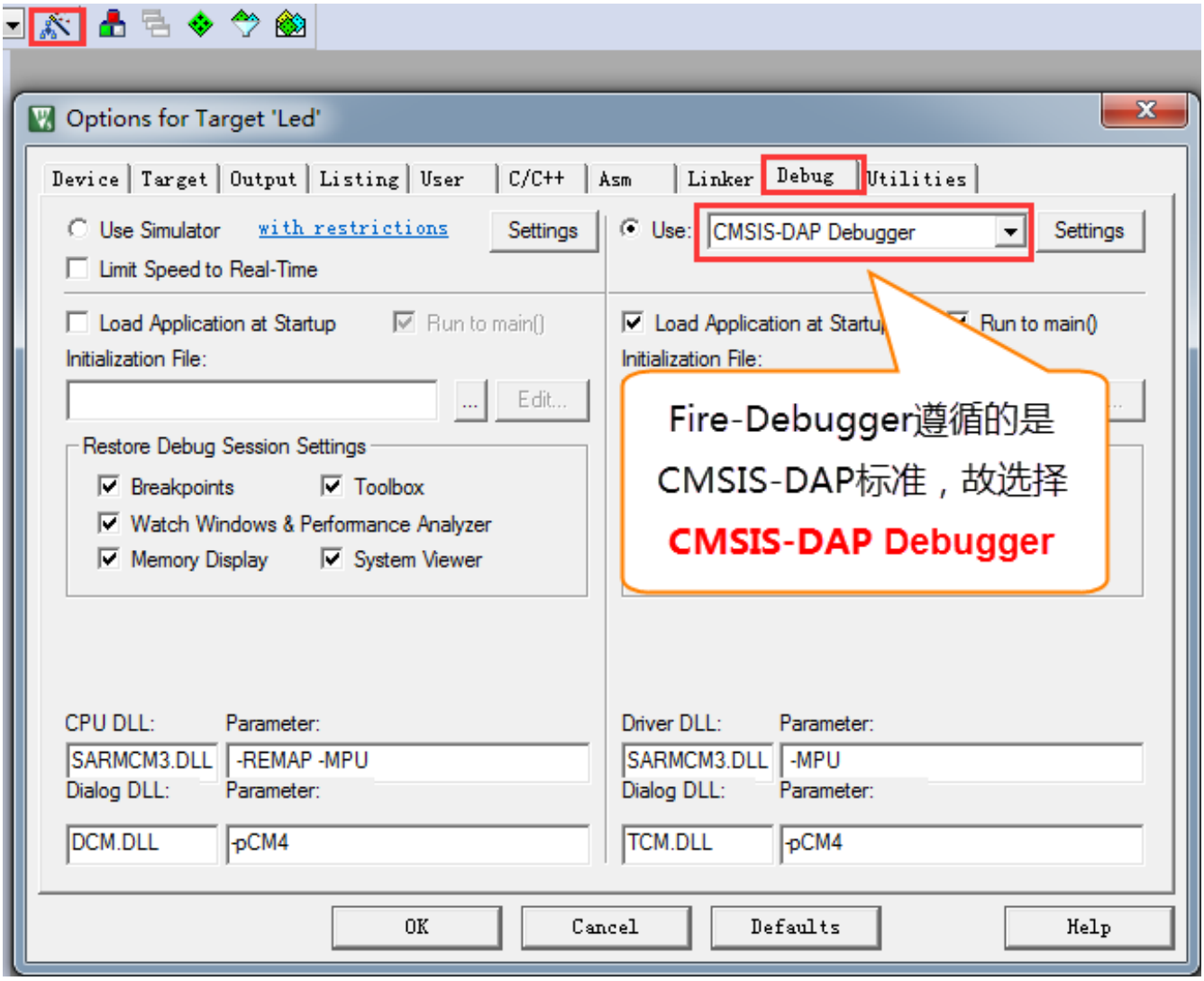


图 3-4 Debug 选择 CMSIS-DAP Debugger

3.3.2 Utilities 选项配置

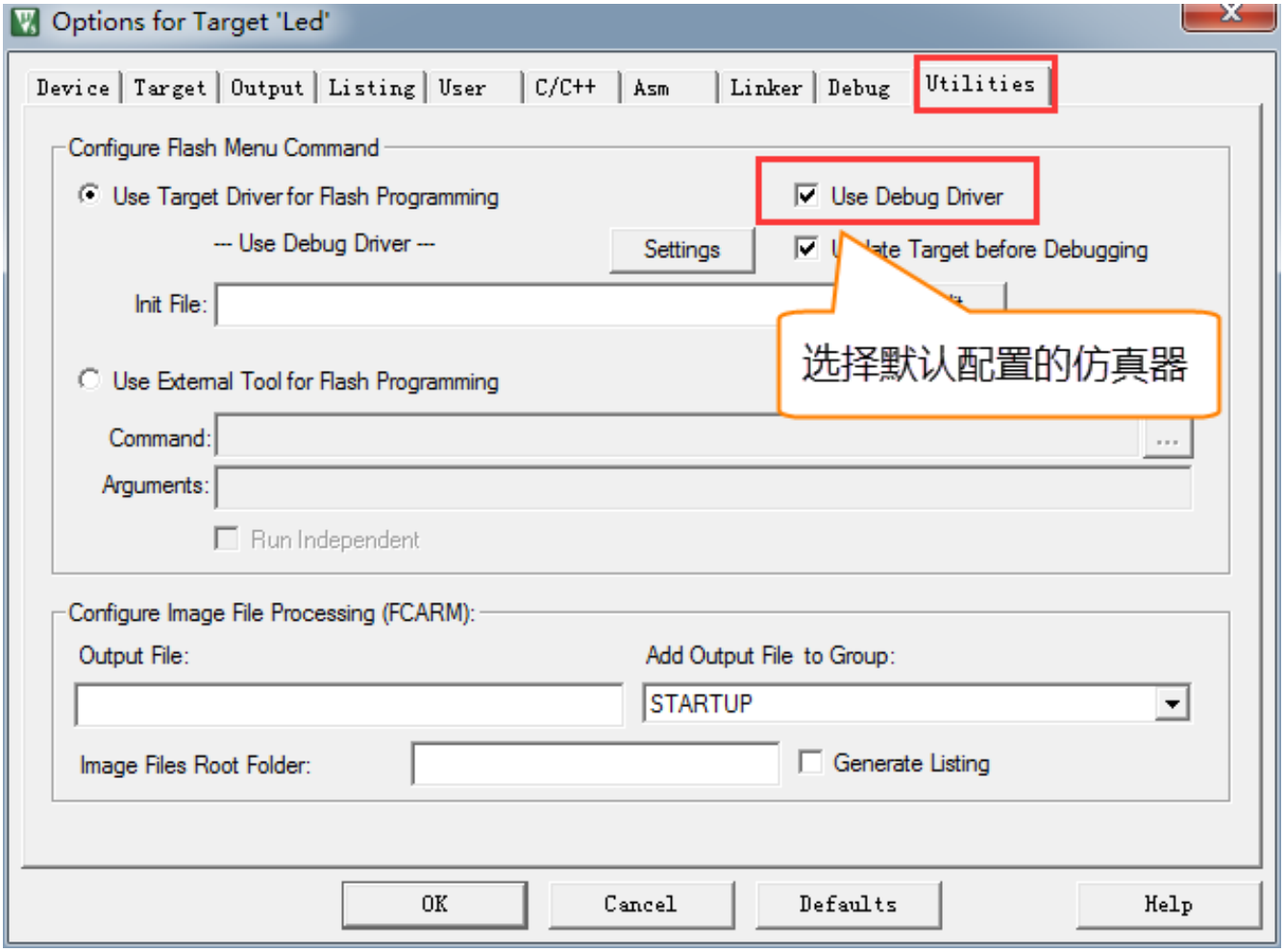


图 3-5 Utilities 选择 Use Debug Driver

3.3.3 Debug Settings 选项配置

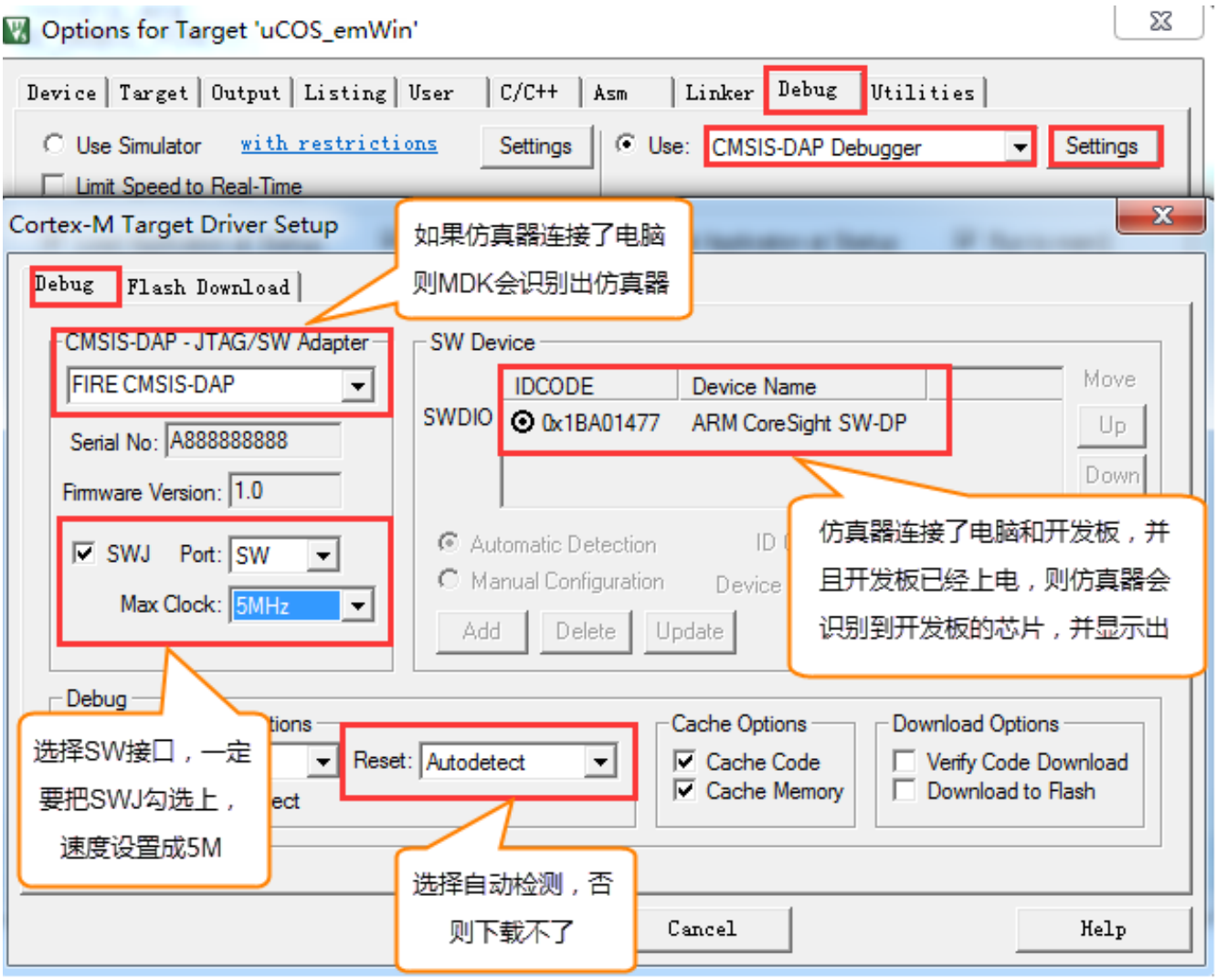


图 3-6 Debug Settings 选项配置

注意：

- 上面，CMSIS-DAP Adapter 一栏下面显示的型号和 Serial No 里显示的和视频如果有差异没有关系，能识别到仿真器和板子就行。
- 底下，Reset 一栏如果没有 Autodetect 选项，可以选择 SYSRESETREQ；如果使用的是普通 DAP 下载器，Connect 一栏先选择：under Reset。

3.4 选择目标板

选择目标板，具体选择多大的 FLASH 要根据板子上的芯片型号决定。野火 STM32 开发板的配置是：F1 选 512K，F4 选 1M。这里面有个小技巧就是把 Reset and Run 也勾选上，这样程序下载完之后就会自动运行，否则需要手动复位。擦除的 FLASH 大小选择 Sectors 即可，不要选择 Full Chip，不然下载会比较慢。

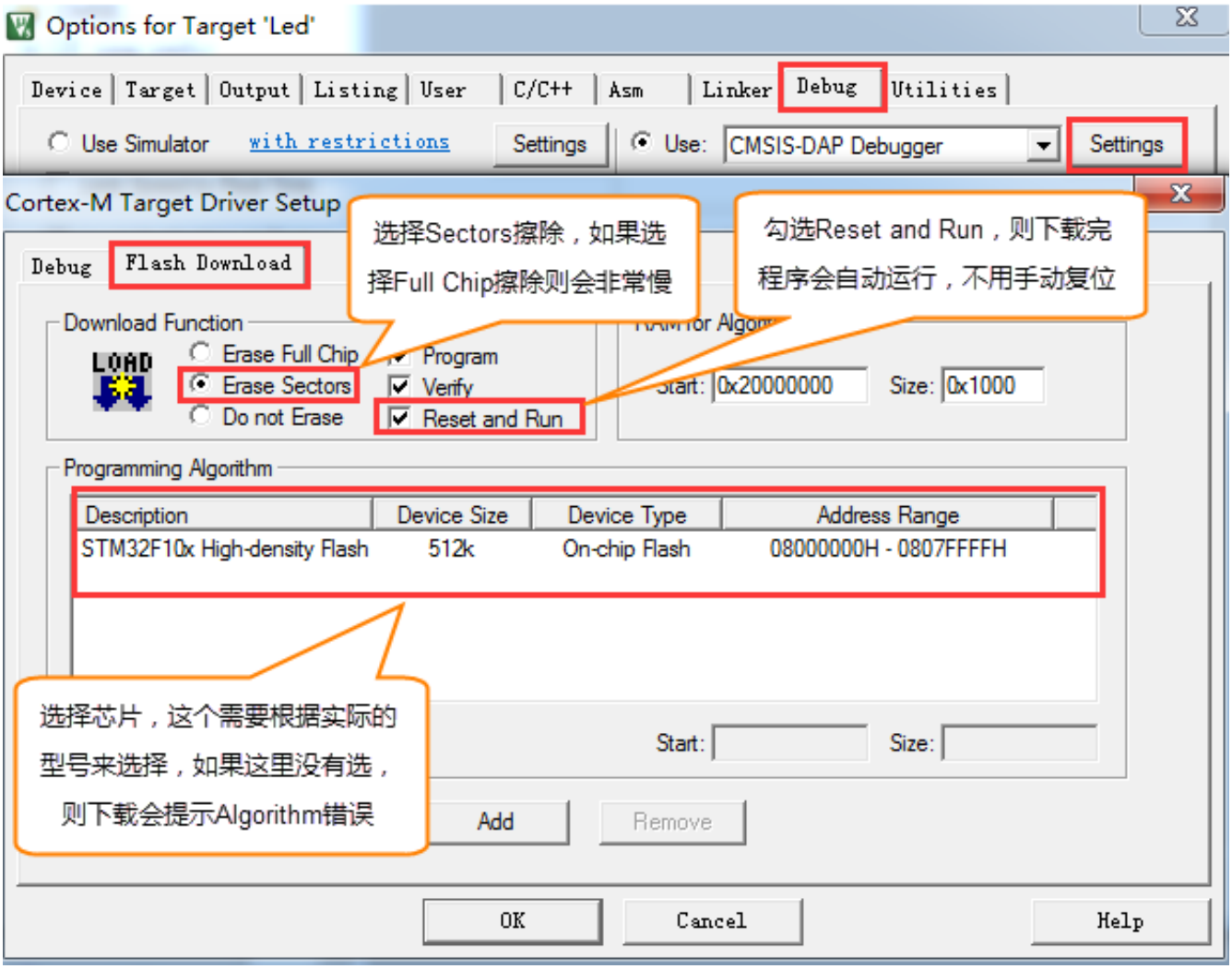
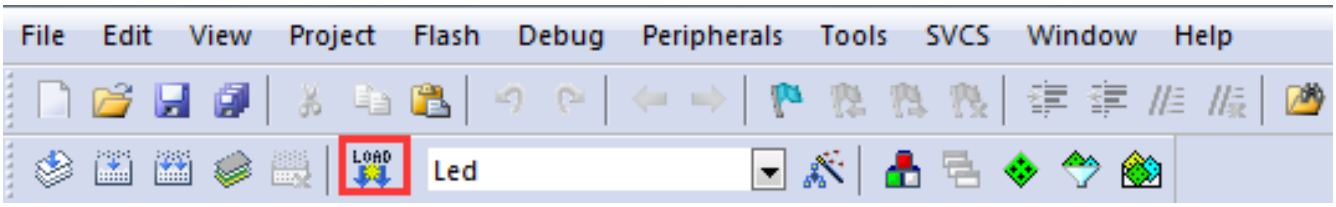


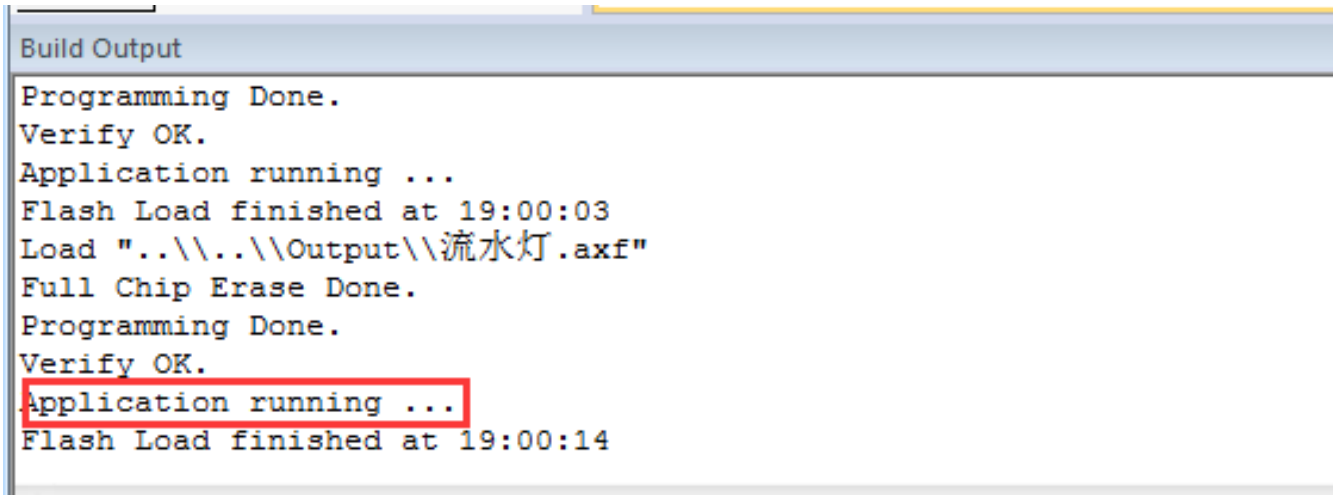
图 3-7 选择目标板

3.5 下载程序

如果前面步骤都成功了，接下来就可以把编译好的程序下载到开发板上运行。下载程序不需要其他额外的软件，直接点击 KEIL 中的 LOAD 按钮即可。



程序下载后，Build Output 选项卡如果打印出 Application running…则表示程序下载成功。如果没有出现实验现象，按复位键试试。



第 4 章　初识 STM32

本章参考资料：1、《STM8 和 STM32 产品选型手册》

4.1 什么是 STM32

STM32，从字面上来理解，ST 是意法半导体，M 是 Microelectronics 的缩写，32 表示 32 位，合起来理解，STM32 就是指 ST 公司开发的 32 位微控制器。在如今的 32 位控制器当中，STM32 可以说是最璀璨的新星，它受宠若娇，大受工程师和市场的青睐，无芯能出其右。

4.1.1 STM32 诞生的背景

51 是嵌入式学习中一款入门级的精典 MCU，因其结构简单，易于教学，且可以通过串口编程而不需要额外的仿真器，所以在教学时被大量采用，至今很多大学在嵌入式教学中用的还是 51。51 诞生于 70 年代，属于传统的 8 位单片机，如今，久经岁月的洗礼，既有其辉煌又有其不足。现在的市场产品竞争越来越激烈，对成本极其敏感，相应地对 MCU 的性能要求也更苛刻：更多功能，更低功耗，易用界面和多任务。面对这些要求，51 现有的资源就显得得抓襟见肘。所以无论是高校教学还是市场需求，都急需一款新的 MCU 来为这个领域注入新的活力。

基于这样的市场需求，ARM 公司推出了其全新的基于 ARMv7 架构的 32 位 Cortex-M3 微控制器内核。紧随其后，ST（意法半导体）公司就推出了基于 Cortex-M3 内核的 MCU——STM32。STM32 凭借其产品线的多样化、极高的性价比、简单易用的库开发方式，迅速在众多 Cortex-M3 MCU 中脱颖而出，成为最闪亮的一颗新星。STM32 一上市就迅速占领了中低端 MCU 市场，受到了市场和工程师的无比青睐，颇有星火燎原之势。

作为一名合格的嵌入式工程师，面对新出现的技术，我们不是充耳不闻，而是要尽快吻合市场的需要，跟上技术的潮流。如今 STM32 的出现就是一种趋势，一种潮流，我们要做的就是搭上这趟快车，让自己的技术更有竞争力。

4.2 STM32 能做什么

STM32 属于一个微控制器，自带了各种常用通信接口，比如 USART、I2C、SPI 等，可接非常多的传感器，可以控制很多的设备。现实生活中，我们接触到的很多电器产品都有 STM32 的身影，比如智能手环，微型四轴飞行器，平衡车、移动 POST 机，智能电饭锅，3D 打印机等等。下面我们以最近最为火爆的两个产品来讲解下，一个是手环，一个是飞行器。

4.2.1 智能手环

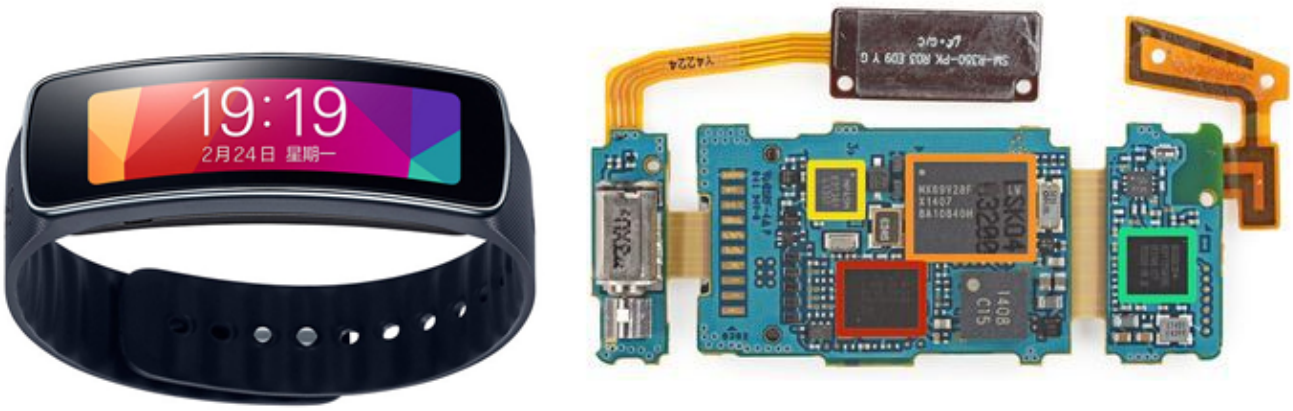


图 4-1 三星 GearFit 智能手环

红圈：STM32F439ZIY6S 处理器（我们简称之为 439），2048KB FLASH ，256KB RAM ,WLCSP143 封装。

橙圈：Macronix MX69V28F64 16 MB 闪存，基于 MCP 封装的存储器，是一种包含了 NOR 和 SRAM 的闪存，这在手环手机这种移动设备中经常使用，优点是体积小，可以减小 PCB 的尺寸。这个闪存用的 439 的 FSMC 接口驱动。

黄圈：InvenSense MPU-6500 陀螺仪/加速度计，用 439 的 I2C 接口驱动。

绿圈：博通 BCM4334WKUBG 芯片，支持 802.11n，蓝牙 4.0+HS 以及 FM 接收芯片，用 439 的 SDIO 或者 SPI 接口驱动。

显示：1.84” 可弯曲屏幕 (Super AMOLED)，432 x 128 像素。触摸部分用 439 的 I2C 接口驱动，

OLED 显示部分用 LTDC 接口驱动。

表格 4-1 三星 Gear Fit 和野火 STM32f103“指南者”资源对比

资源	三星 Gear Fit	指南者
CPU	STM32F439ZIY6S，WLCSP143 封装	STM32F103VET6，LQPF100 封装
存储	NOR+SRAM 16MB，FSMC 接口	串行 FLASH 16MB ，SPI 接口
显示	1.84 寸的 AMOLED，RGB 接口，LTDC 驱动	3.2 寸电阻屏，FSMC 接口
陀螺仪	MPU6050，I2C 接口	可外接 MPU6050 模块，I2C 接口
无线通信	蓝牙: 博通 BCM4334，SDIO 或者 SPI 接口	WIFI：ESP8266，UART 接口

除了这几个重要资源的对比，我们的指南者开发板上还板载了 EEPROM，USB 转串口，蜂鸣器，LED，普通按键，电容按键等外设资源，还可以扩展 VS1053 MP3 模块，W5500 以太网模块，利用这些可以充分的学习 STM32F103VET6 这个芯片。在板子上面，还可以跑系统 ucosiii，学习图形界面 emwin。如果功夫所至，学完之后，自己都可以做一个类似 Gear Fit 这样的手环。可很多人又会说，Gear Fit 涉及硬件和软件，整个系统这么复杂，并不是一个人可以完成的。说的没错，我们可以做不了，但是我们的能力可以无限接近，多学点，技多不压身嘛。

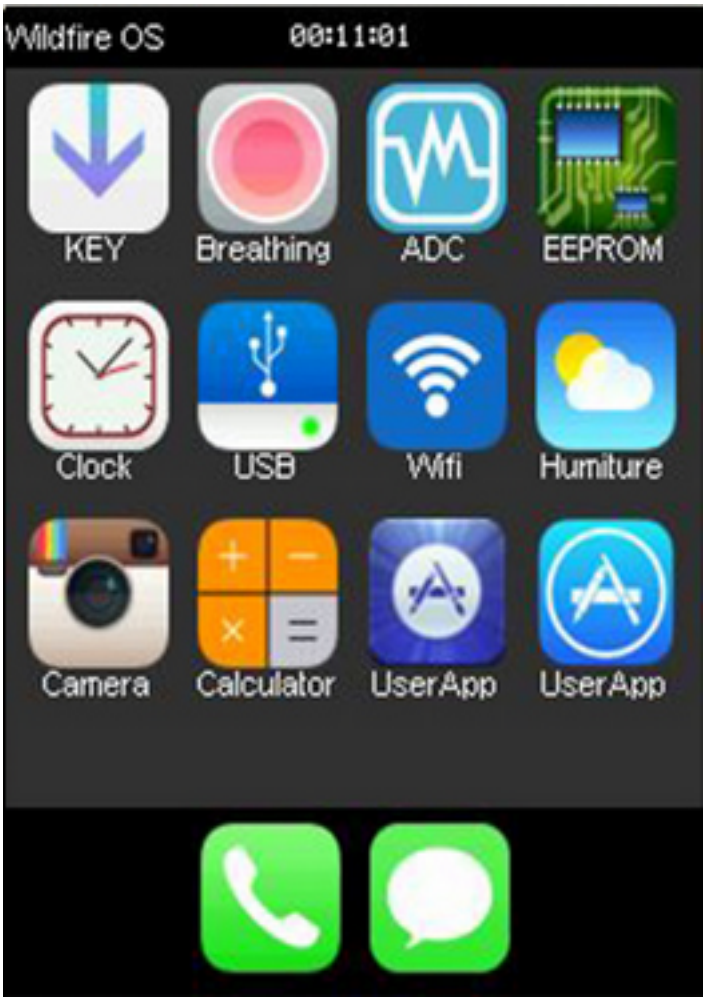


图 4-2 ucosiii+emwin 做的系统界面（指南者的开机界面）

4.2.2 微型四轴飞行器

现在无人机非常火热，高端的无人机用 STM32 做不来，但是小型的四轴飞行器用 STM32 还是绰绰有余的。如图 4-3 所示飞行器的基本都可以用 STM32 搞定。

图 4-3 微型四轴飞行器

上面的是属于产品，如果想自己 DIY，可以在入门 STM32 之后，买一本飞行器 DIY 的书，边做边学。入门级的书籍推荐《四轴飞行器 DIY—基于 STM32 微控制器》，见图 4-4。

图 4-4 四轴飞行器 DIY —基于 STM32 微控制器

4.2.3 淘宝众筹

学会了 STM32，想自己做产品，如何实现自己的梦想，淘宝众筹吧。做出产品原型，用别人的钱为自己的梦想买单。

淘宝众筹科技类网址：这里面有很多小玩意都可以用 STM32 实现，只要你的创意到了，就会有人买单，前提是我们要先学会 STM32。

https://hi.taobao.com/market/hi/list.php?spm=a215p.1596646.1.8.LbVyJk#type=121288001

图 4-5 淘宝众筹科技类

4.3 STM32 怎么选型

4.3.1 STM32 分类

STM32 有很多系列，可以满足市场的各种需求，从内核上分有 Cortex-M0、M3、M4 和 M7 这几种，每个内核又大概分为主流、高性能和低功耗。具体的见表格 *4_2*。

单纯从学习的角度出发，可以选择 F1 和 F4，F1 代表了基础型，基于 Cortex-M3 内核，主频为 72MHZ，F4 代表了高性能，基于 Cortex-M4 内核，主频 180M。

之于 F1，F4（429 系列以上）除了内核不同和主频的提升外，升级的明显特色就是带了 LCD 控制器和摄像头接口，支持 SDRAM，这个区别在项目选型上会被优先考虑。但是从大学教学和用户初学来说，还是首选 F1 系列，目前在市场上资料最多，产品占有量最多的就是 F1 系列的 STM32。

表格 4-2 STM8 和 STM32 分类

CPU 位数	内核	系列	描述
32	Cortex-M0	STM32-F0	入门级
		STM32-L0	低功耗
	Cortex-M3	STM32-F1	基础型，主频 72M
		STM32-F2	高性能
		STM32-L1	低功耗
	Cortex-M4	STM32-F3	混和信号
		STM32-F4	高性能，主频 180M
		STM32-L4	低功耗
	Cortex-M7	STM32-F7	高性能
8	超级版 6502	STM8S	标准系列
		STM8AF	标准系列的汽车应用
		STM8AL	低功耗的汽车应用
		STM8L	低功耗

4.3.2 STM32 命名方法

这里我们以野火 F103 指南者用的型号 STM32F103VET6 来讲解下 STM32 的命名方法。

表格 4-3 STM32F103VET6 命名解释

	STM32F103VET6
家族	STM32, 表示 32bit 的 MCU
产品类型	F 表示基础型
具体特性	基础型
引脚数目	V 表示 100pin，其他常用的为 C 表示 48，R 表示 64，Z 表示 144，Z 表示 144，B 表示 208，N 表示 216
FLASH 大小	E 表示 512KB，其他常用的为 C 表示 256，E 表示 512，I 表示 2048
封装	T 表示 QFP 封装，这个是最常用的封装
温度	6 表示温度等级为 A：-40~85°

有关更详细的命名方法见图 *4_6*。

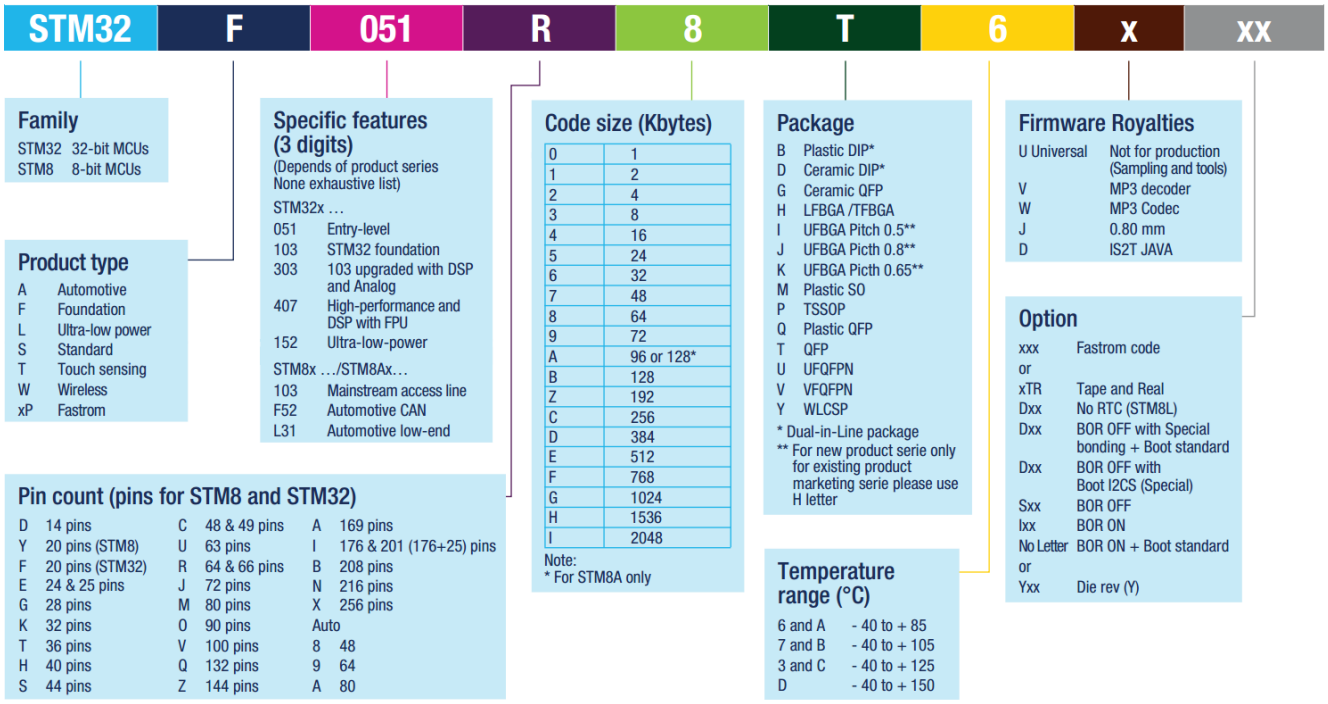


图 4-6 STM8 和 STM32 命名方法，摘自《STM8 和 STM32 选型手册》

4.3.3 选择合适的 MCU

了解了 STM32 的分类和命名方法之后，就可以根据项目的具体需求先大概选择哪类内核的 MCU，普通应用，不需要接大屏幕的一般选择 Cortex-M3 内核的 F1 系列，如果要追求高性能，需要大量的数据运算，且需要外接 RGB 大屏幕的则选择 Cortex-M4 内核的 F429 系列。

明确了大方向之后，接下来就是细分选型，先确定引脚，引脚多的功能就多，价格也贵，具体得根据实际项目中需要使用到什么功能，够用就好。确定好了引脚数目之后再选择 FLASH 大小，相同引脚数的 MCU 会有不同的 FLASH 大小可供选择，这个也是根据实际需要选择，程序大的就选择大点的 FLASH，要是产品一量产，这些省下来的都是钱啊。有些月出货量以 KK（百万数量级）为单位的产品，不仅是 MCU，连电阻电容能少用就少用，更甚者连 PCB 的过孔的多少都有讲究。项目中的元器件的选型的水深的很，很多学问。

4.3.3.1 如何分配原理图 IO

在画原理图之前，一般的做法是先把引脚分类好，然后才开始画原理图，引脚分类具体见*表格 4_4*。

表格 4-4 画原理图时的引脚分类

引脚分类	引脚说明说明
电源	(VBAT)、(VDD VSS)、(VDDA VSSA)、(VREF+ VREF-) 等
晶振 IO	主晶振 IO，RTC 晶振 IO
下载 IO	用于 JTAG 下载的 IO：JTMS、JTCK、JTDI、JTDO、NJTRST
BOOT IO	BOOT0、BOOT1，用于设置系统的启动方式
复位 IO	NRST，用于外部复位
上面 5 部分 IO 组成的系统我们也叫做最小系统	
GPIO	专用器件接到专用的总线，比如 I2C，SPI，SDIO，FSMC，DCMI 这些总线的器件需要接到专用的 IO
	普通的元器件接到 GPIO，比如蜂鸣器，LED，按键等元器件用普通的 GPIO 即
	如果还有剩下的 IO，可根据项目需要引出或者不引出

4.3.3.2 如何寻找 IO 的功能说明

要想根据功能来分配 IO，那就得先知道每个 IO 的功能说明，这个我们可以从官方的数据手册里面找到。在学习的时候，有两个官方资料我们会经常用到，一个是参考手册（英文叫 Reference manual），另外一个是数据手册（英文叫 Data Sheet）。两者的具体区别见*表格 4_5*。

表格 4-5 参考手册和数据手册的内容区别

手册	主要内容	说明
参考手册	片上外设的功能说明和寄存器描述	对片上每一个外设的功能和使用做了详细的说明，包含寄存器的详细描述。编程的时候需要反复查询这个手册。
数据手册	功能概览	主要讲这个芯片有哪些功能，属于概括性的介绍。芯片选型的时候首先看这个部分。
	引脚说明	详细描述每一个引脚的功能，设计原理图的时候和写程序的时候需要参考这部分。
	内存映射	讲解该芯片的内存映射，列举每个总线的地址和包含有哪些外设。
	封装特性	讲解芯片的封装，包含每个引脚的长度宽度等，我们画 PCB 封装的时候需要参考这部分的参数。

一句话概括：数据手册主要用于芯片选型和设计原理图时参考，参考手册主要用于在编程的时候查阅。官方的这两个文档可以从官方网址里面下载：http://www.stmcu.org/document/list/index/category-150，也可以从我们配置的光盘资料里面找到。

在数据手册中，有关引脚定义的部分在“Pinouts and pin description”这个小节中，具体定义见表格 *4_6*。

表格 4-6 数据手册中对引脚定义

Table 5. High-density STM32F103xx pin definitions

Pins (1)						(2) Pin name	(3) Type(1)	(4) I / O Level	(5) Main function(3) (after reset)	(6) Alternate functions(4)	
LFBGA144	LFBGA100	WLCSP64	LQFP64	LQFP100	LQFP144					Default	(7) Remap
A3	A3	-	-	1	1	PE2	I/O	FT	PE2	TRACECK/ FSMC_A23	
A2	B3	-	-	2	2	PE3	I/O	FT	PE3	TRACED0/FSMC_A19	
B2	C3	-	-	3	3	PE4	I/O	FT	PE4	TRACED1/FSMC_A20	
B3	D3	-	-	4	4	PE5	I/O	FT	PE5	TRACED2/FSMC_A21	
B4	E3	-	-	5	5	PE6	I/O	FT	PE6	TRACED3/FSMC_A22	
C2	B2	C6	1	6	6	V_{BAT}	S		V_{BAT}		

表格 4-7 对引脚定义的解读

名称	缩写	说明
1. 引脚序号	阿拉伯数字表示 LQFP 封装，英文字母开头的表示 BGA 封装。引脚序号这里列出了有 6 种封装型号，具体使用哪一种要根据实际情况来选择。	
2. 引脚名称	指复位状态下的引脚名称	
3. 引脚类型	S	电源引脚
	I	输入引脚
	I/O	输入/输出引脚
4.I/O 结构	FT	兼容 5V
	TTa	只支持 3V3，且直接到 ADC
	B	BOOT 引脚
	RST	复位引脚，内部带弱上拉
5. 主功能	每个引脚复位后的功能	
6. 复用功能	这里指的是 IO 的默认的复用功能	
7. 重映射功能	IO 除了默认的复用功能之外，还可以通过重映射的方法映射到其他的 IO，这样可以增加 IO 口功能的多样性和灵活性。	

4.3.3.3 开始分配原理图 IO

比如我们的 F103“指南者”使用的 MCU 型号是 STM32F103VET6，封装为 LQFP100，我们在数据手册中找到这个封装的引脚定义，然后根据引脚序号，一个一个复制出来，整理成 excel 表。具体整理方法按照表格 *4_4* 画原理图时的引脚分类即可。分配好之后就开始画原理图。

4.3.4 PCB 哪里打样

设计好原理图，画好 PCB 之后，需要把板子做出来，进行软硬件联调。首先得 PCB 打样，这里我推荐一家我经常打样的厂家，深圳嘉立创（JLC），行业标杆，良心价格，网址：http://www.sz-jlc.com。一块 10CM*10CM 以内的板子，三天做好，50 块就可以搞定，还包邮，简直便宜到掉渣。如果你足够懒，不想自己焊接电阻电容二三极管什么的，嘉立创还可以帮你把 PCB 样板上的阻容贴好给你，打样贴片一条龙。

样品做好了，软硬件什么都 OK，要小批量怎么办？还是找 JLC。

第 5 章　什么是寄存器

本章参考资料：《STM32F10xx 参考手册》、《STM32F10xx 数据手册》、

学习本章时，配合《STM32F10xx 参考手册》“存储器和总线架构”及“通用 I/O(GPIO)”章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

5.1 什么是寄存器

我们经常说寄存器，那么什么是寄存器？这是我们本章需要讲解的内容，在学习的过程中，大家带着这个疑问好好思考下，到最后看看大家能否用一句话给寄存器下一个定义。

5.2 STM32 长啥样

以我们的 F103-指南者开发板为例，指南者开发板中使用的芯片是 100pin（100 个引脚）的 STM32F103VET6，具体见图 5-1。这个就是我们接下来要学习的 STM32，它将带领我们进入嵌入式的殿堂。

芯片正面是丝印，ARM 应该是表示该芯片使用的是 ARM 的内核，STM32F103VET6 是芯片型号，后面的字应该是跟生产批次相关，最下面的是 ST 的 LOGO。

芯片四周是引脚，左下角的小圆点表示 1 脚，然后从 1 脚起按照逆时针的顺序排列（所有芯片的引脚顺序都是逆时针排列的）。开发板中把芯片的引脚引出来，连接到其他各种芯片上（比如传感器），然后在 STM32 上编程（实际就是通过程序控制这些引脚输出高电平或者低电平）来控制其他芯片工作，通过做实验的方式来学习 STM32 芯片的各个资源。

开发板是一种评估板，主要用于学习与评估，其板载资源非常丰富，引脚复用比较多，力求在一个板子上验证芯片的全部功能。我们的 STM32 F1 的开发板上面的 STM32 型号虽然不完全相同，但实际上学习它们所需要的数据手册和参考手册资料是一样的，初学者不必为此担心。

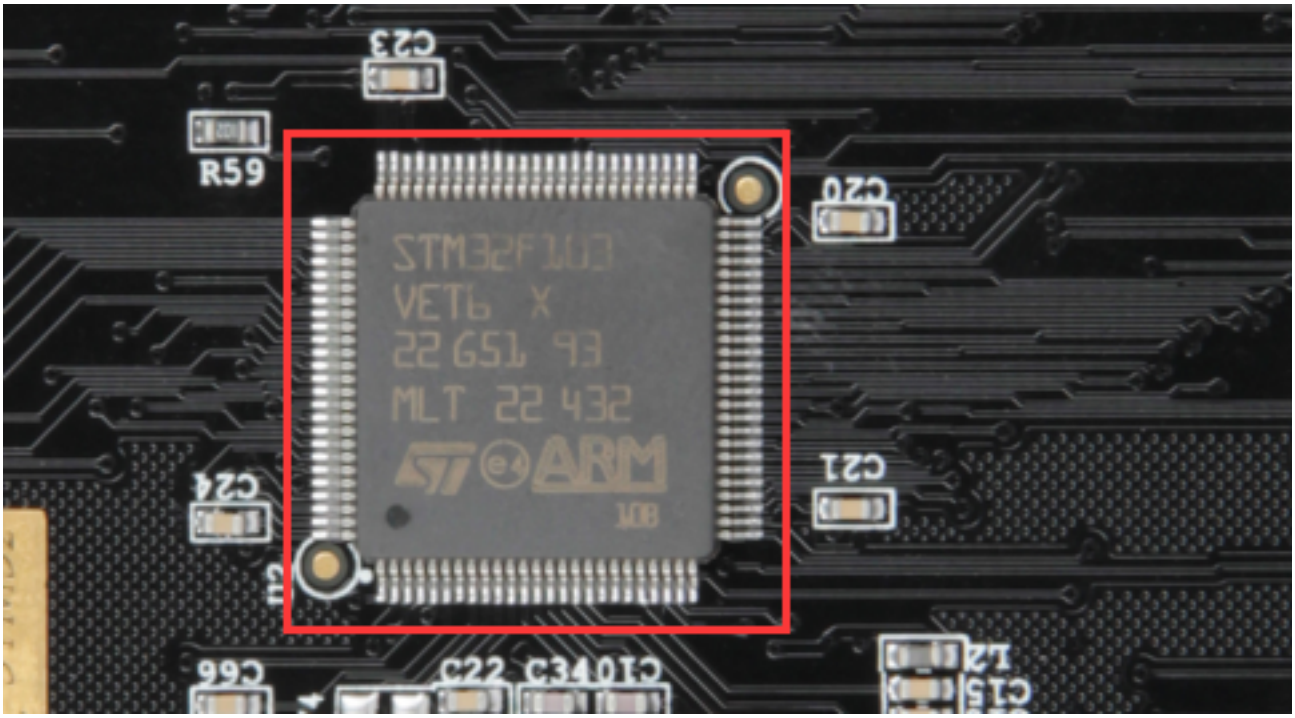


图 5-1 STM32F103VET6 实物图（红色框中部分）

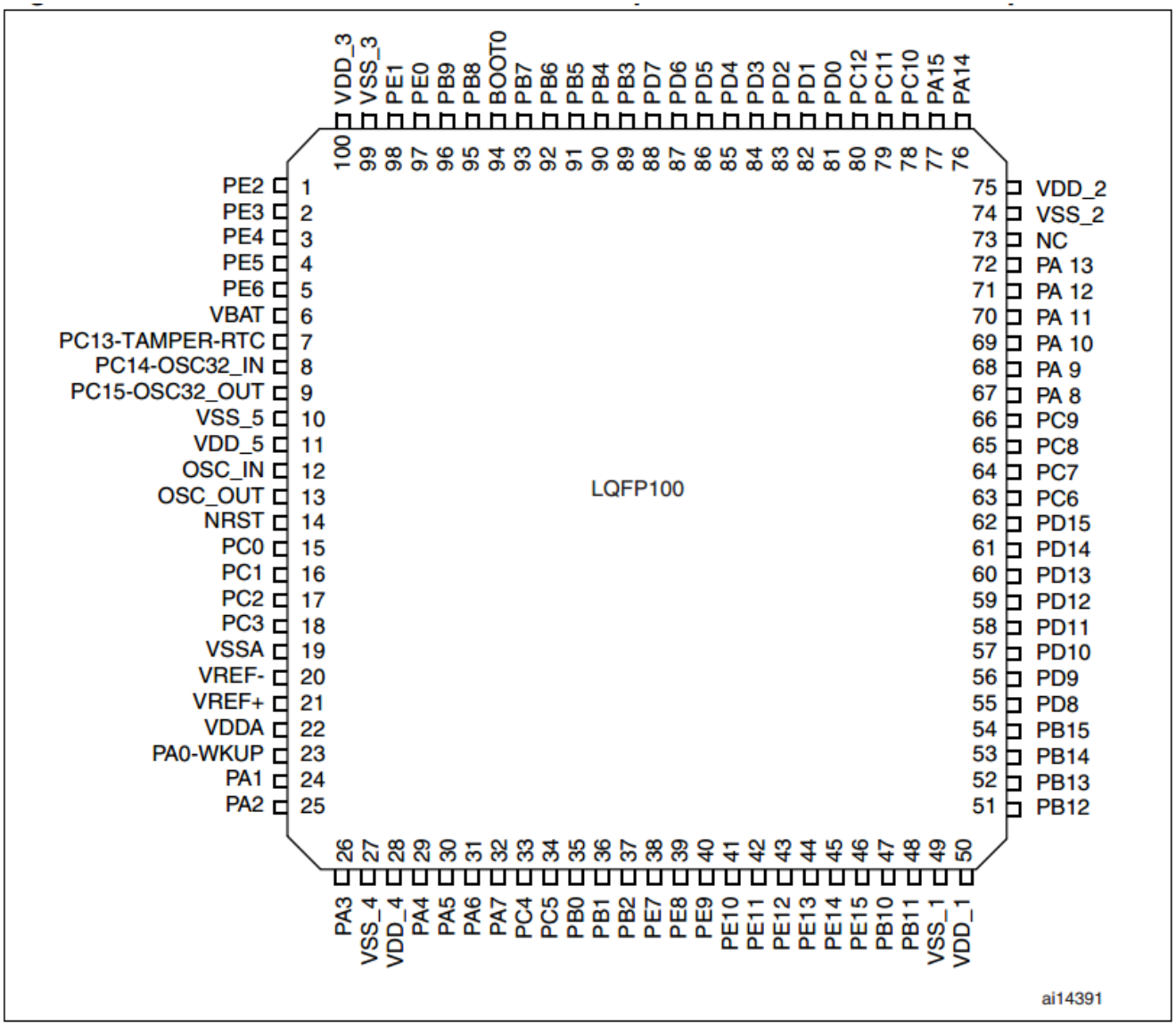


图 5-2 STM32F103VET6 正面引脚图

5.3 芯片里面有什么

我们看到的 STM32 芯片是已经封装好的成品，主要由内核和片上外设组成。若与电脑类比，内核与外设就如同电脑上的 CPU 与主板、内存、显卡、硬盘的关系。

STM32F103 采用的是 Cortex-M3 内核，内核即 CPU，由 ARM 公司设计。ARM 公司并不生产芯片，而是出售其芯片技术授权。芯片生产厂商 (SOC) 如 ST、TI、NXP 等，负责在内核之外设计部件并生产整个芯片，这些内核之外的部件被称为核外外设或片上外设。如 GPIO、USART（串口）、I2C、SPI 等都叫做片上外设。具体见图 5-3。

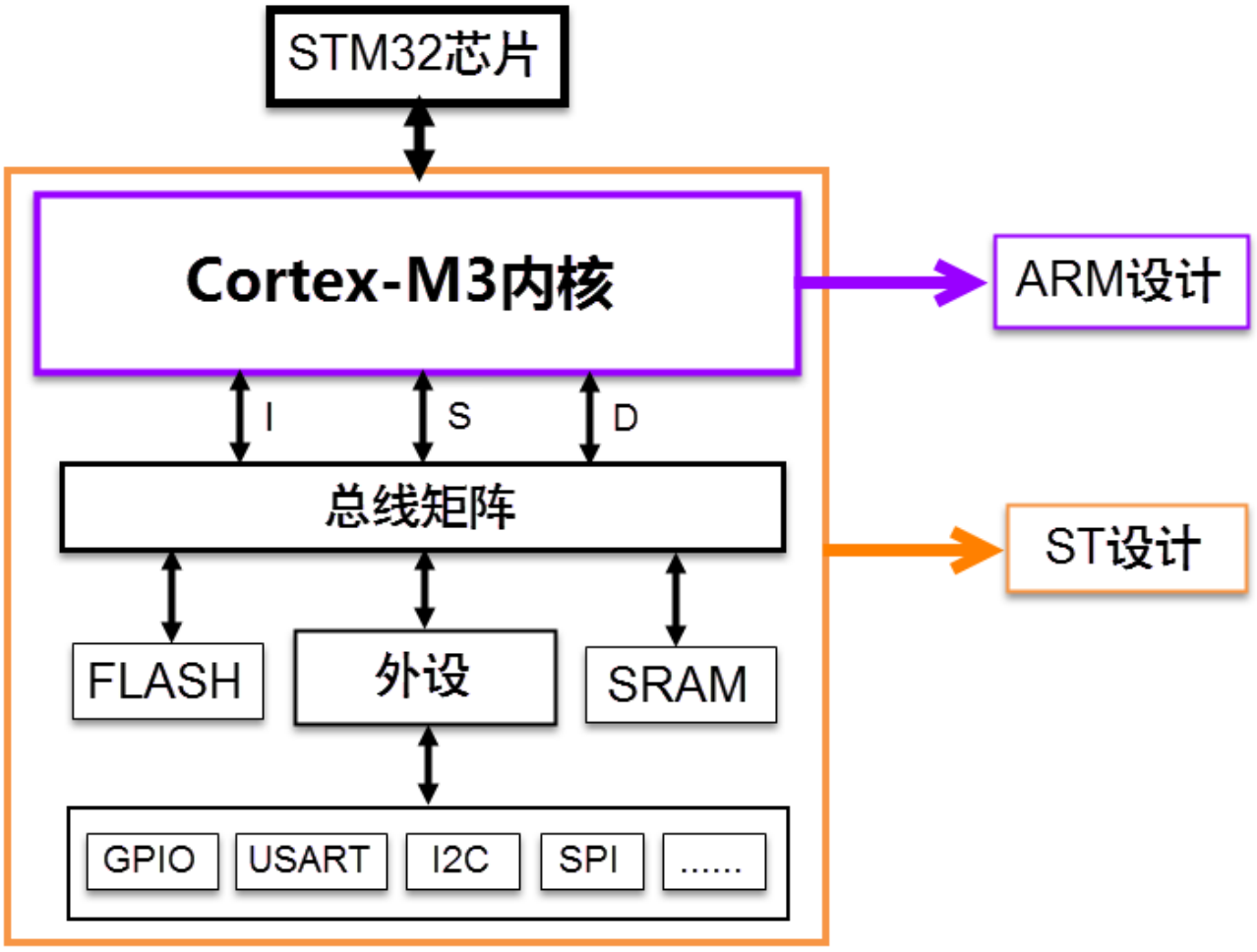


图 5-3 STM32 芯片架构简图

芯片（这里指内核，或者叫 CPU）和外设之间通过各种总线连接，其中驱动单元有 4 个，被动单元也有 4 个，具体见图 5-4。为了方便理解，我们都可以把驱动单元理解成是 CPU 部分，被动单元都理解成外设。下面我们简单介绍下驱动单元和被动单元的各个部件。

5.3.1 ICode 总线

ICode 中的 I 表示 Instruction，即指令。我们写好的程序经过编译之后都是一条条指令，存放在 FLASH 中，内核要读取这些指令来执行程序就必须通过 ICode 总线，它几乎每时每刻都需要被使用，它是专门用来取指的。

5.3.2 驱动单元

5.3.2.1 DCode 总线

DCode 中的 D 表示 Data，即数据，那说明这条总线是用来取数的。我们在写程序的时候，数据有常量和变量两种，常量就是固定不变的，用 C 语言中的 const 关键字修饰，是放到内部的 FLASH 当中的，变量是可变的，不管是全局变量还是局部变量都放在内部的 SRAM。因为数据可以被 Dcode 总线和 DMA 总线访问，所以为了避免访问冲突，在取数的时候需要经过一个总线矩阵来仲裁，决定哪个总线在取数。

5.3.2.2 系统总线

系统总线主要是访问外设的寄存器，我们通常说的寄存器编程，即读写寄存器都是通过这根系统总线来完成的。

5.3.2.3 DMA 总线

DMA 总线也主要是用来传输数据，这个数据可以是在某个外设的数据寄存器，可以在 SRAM，可以在内部的 FLASH。因为数据可以被 Dcode 总线和 DMA 总线访问，所以为了避免访问冲突，在取数的时候需要经过一个总线矩阵来仲裁，决定哪个总线在取数。

5.3.3 被动单元

5.3.3.1 内部的闪存存储器

内部的闪存存储器即 FLASH，我们编写好的程序就放在这个地方。内核通过 ICode 总线来取里面的指令。

5.3.3.2 内部的 SRAM

内部的 SRAM，即我们通常说的 RAM，程序的变量，堆栈等的开销都是基于内部的 SRAM。内核通过 DCode 总线来访问它。

5.3.3.3 FSMC

FSMC 的英文全称是 Flexible static memory controller，叫灵活的静态的存储器控制器，是 STM32F10xx 中一个很有特色的外设，通过 FSMC，我们可以扩展内存，如外部的 SRAM，NAND-FLASH 和 NORFLASH。但有一点我们要注意的是，FSMC 只能扩展静态的内存，即名称里面的 S：static，不能是动态的内存，比如 SDRAM 就不能扩展。

5.3.3.4 AHB 到 APB 的桥

从 AHB 总线延伸出来的两条 APB2 和 APB1 总线，上面挂载着 STM32 各种各样的特色外设。我们经常说的 GPIO、串口、I2C、SPI 这些外设就挂载在这两条总线上，这个是我们学习 STM32 的重点，就是要学会编程这些外设去驱动外部的各种设备。

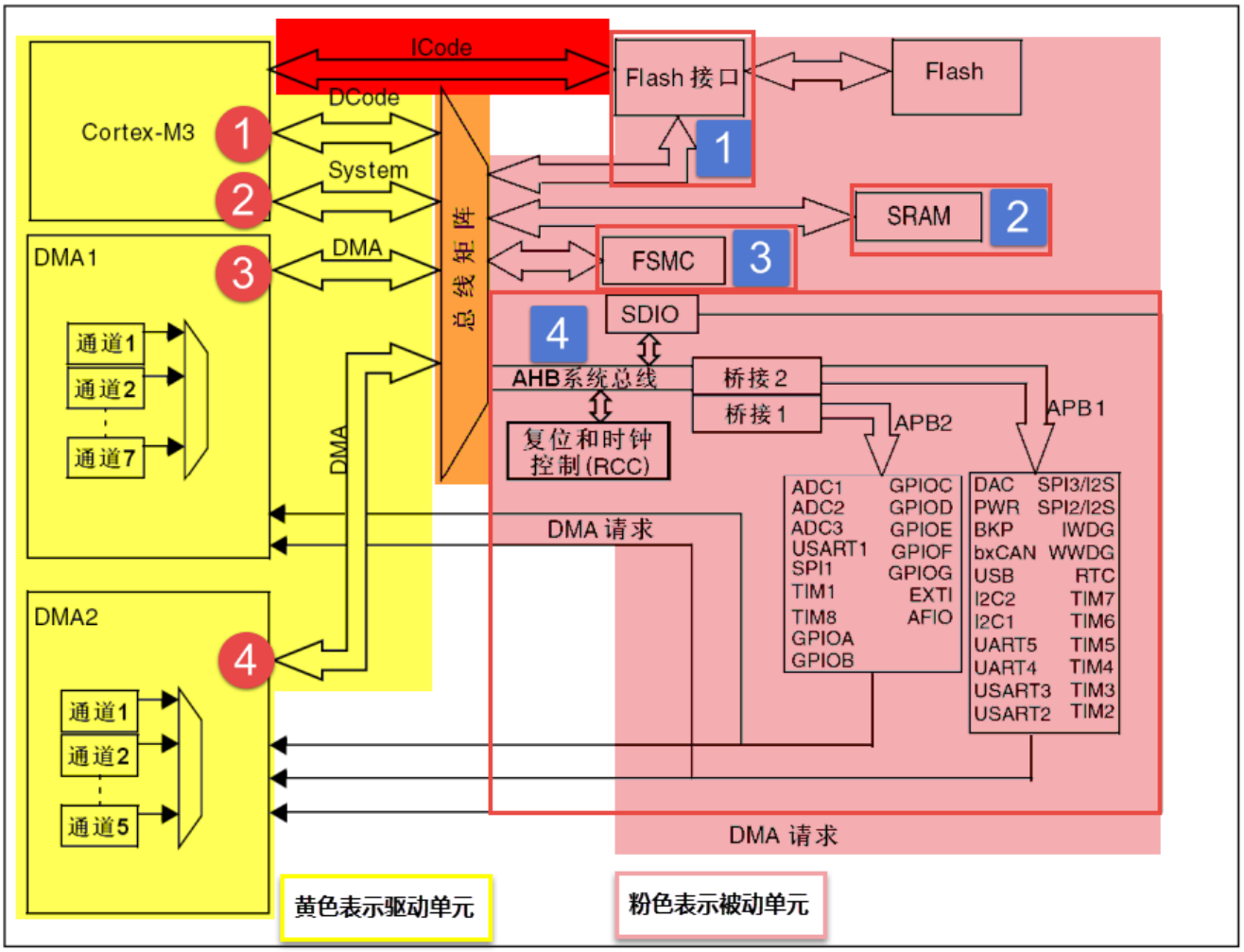


图 5-4 STM32F10xx 系统框图（不包括互联型）

5.4 存储器映射

在图 5-4 中，被控单元的 FLASH，RAM，FSMC 和 AHB 到 APB 的桥（即片上外设），这些功能部件共同排列在一个 4GB 的地址空间内。我们在编程的时候，可以通过他们的地址找到他们，然后来操作他们（通过 C 语言对它们进行数据的读和写）。

5.4.1 存储器映射

存储器本身不具有地址信息，它的地址是由芯片厂商或用户分配的，给存储器分配地址的过程就称为存储器映射，具体见图 5-5。如果给存储器再分配一个地址就叫存储器重映射。

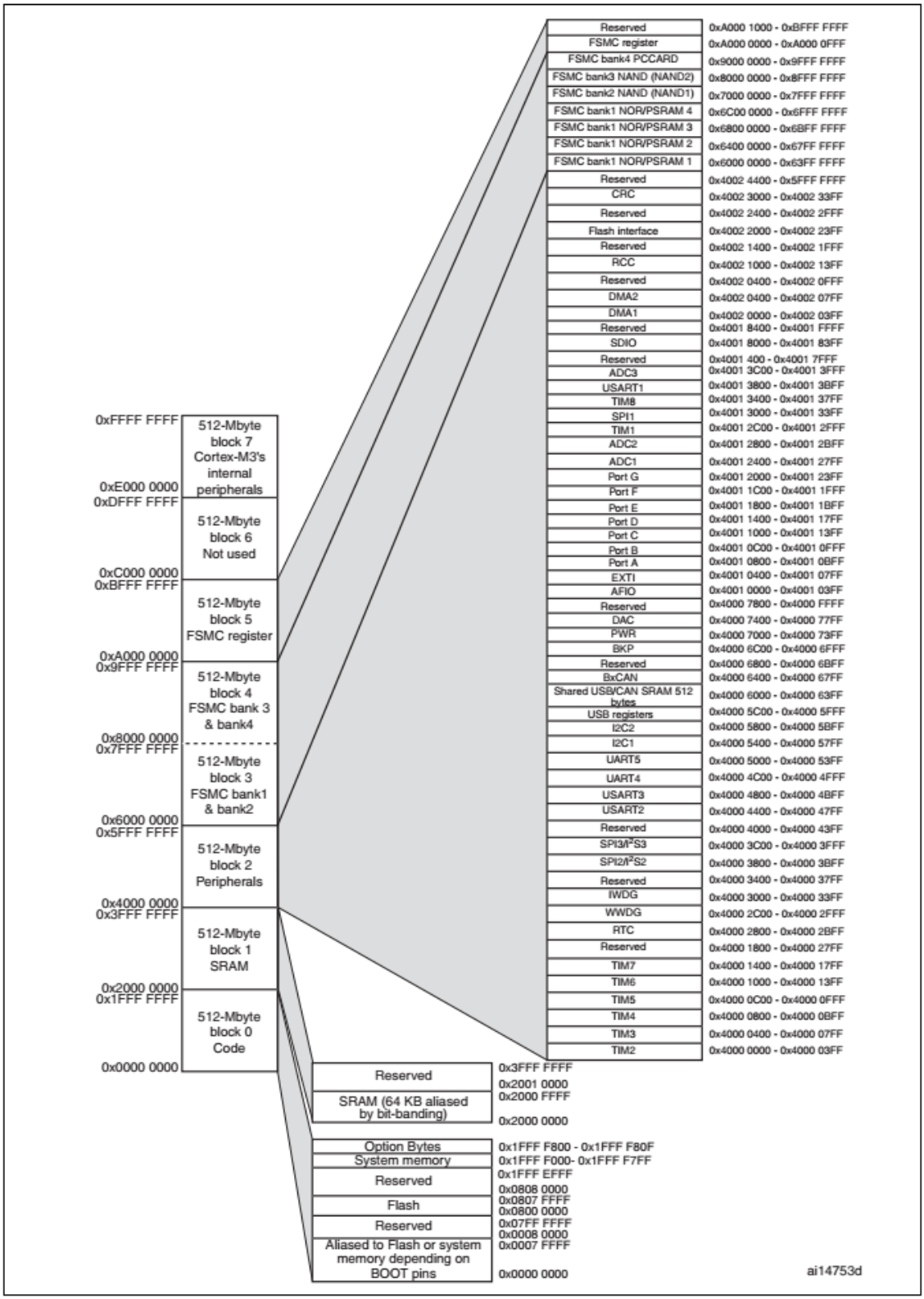

0xFFFF FFFF
512-Mbyte block 7 Cortex-M3's internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
512-Mbyte block 5 FSMC register
0xA000 0000
0x9FFF FFFF
512-Mbyte block 4 FSMC bank 3 & bank4
0x8000 0000
0x7FFF FFFF
512-Mbyte block 3 FSMC bank1 & bank2
0x6000 0000
0x5FFF FFFF
512-Mbyte block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte block 0 Code
0x0000 0000
Reserved
0xA000 1000 - 0xBFFF FFFF
FSMC register
0xA000 0000 - 0xA000 0FFF
FSMC bank4 PCCARD
0x9000 0000 - 0x9FFF FFFF
FSMC bank3 NAND (NAND2)
0x8000 0000 - 0x8FFF FFFF
FSMC bank2 NAND (NAND1)
0x7000 0000 - 0x7FFF FFFF
FSMC bank1 NOR/PSRAM 4
0x6C00 0000 - 0x6FFF FFFF
FSMC bank1 NOR/PSRAM 3
0x6800 0000 - 0x6BFF FFFF
FSMC bank1 NOR/PSRAM 2
0x6400 0000 - 0x67FF FFFF
FSMC bank1 NOR/PSRAM 1
0x6000 0000 - 0x63FF FFFF
Reserved
0x4002 4400 - 0x5FFF FFFF
CRC
0x4002 3000 - 0x4002 33FF
Reserved
0x4002 2400 - 0x4002 2FFF
Flash interface
0x4002 2000 - 0x4002 23FF
Reserved
0x4002 1400 - 0x4002 1FFF
RCC
0x4002 1000 - 0x4002 13FF
Reserved
0x4002 0400 - 0x4002 0FFF
DMA2
0x4002 0400 - 0x4002 07FF
DMA1
0x4002 0000 - 0x4002 03FF
Reserved
0x4001 8400 - 0x4001 FFFF
SDIO
0x4001 8000 - 0x4001 83FF
Reserved
0x4001 400 - 0x4001 7FFF
ADC3
0x4001 3C00 - 0x4001 3FFF
USART1
0x4001 3800 - 0x4001 3BFF
TIM8
0x4001 3400 - 0x4001 37FF
SPI1
0x4001 3000 - 0x4001 33FF
TIM1
0x4001 2C00 - 0x4001 2FFF
ADC2
0x4001 2800 - 0x4001 2BFF
ADC1
0x4001 2400 - 0x4001 27FF
Port G
0x4001 2000 - 0x4001 23FF
Port F
0x4001 1C00 - 0x4001 1FFF
Port E
0x4001 1800 - 0x4001 1BFF
Port D
0x4001 1400 - 0x4001 17FF
Port C
0x4001 1000 - 0x4001 13FF
Port B
0x4001 0C00 - 0x4001 0FFF
Port A
0x4001 0800 - 0x4001 0BFF
EXTI
0x4001 0400 - 0x4001 07FF
AFIO
0x4001 0000 - 0x4001 03FF
Reserved
0x4000 7800 - 0x4000 FFFF
DAC
0x4000 7400 - 0x4000 77FF
PWR
0x4000 7000 - 0x4000 73FF
BKP
0x4000 6C00 - 0x4000 6FFF
Reserved
0x4000 6800 - 0x4000 6BFF
BxCAN
0x4000 6400 - 0x4000 67FF
Shared USB/CAN SRAM 512 bytes
0x4000 6000 - 0x4000 63FF
USB registers
0x4000 5C00 - 0x4000 5FFF
I2C2
0x4000 5800 - 0x4000 5BFF
I2C1
0x4000 5400 - 0x4000 57FF
UART5
0x4000 5000 - 0x4000 53FF
UART4
0x4000 4C00 - 0x4000 4FFF
USART3
0x4000 4800 - 0x4000 4BFF
USART2
0x4000 4400 - 0x4000 47FF
Reserved
0x4000 4000 - 0x4000 43FF
SPI3/I2S3
0x4000 3C00 - 0x4000 3FFF
SPI2/I2S2
0x4000 3800 - 0x4000 3BFF
Reserved
0x4000 3400 - 0x4000 37FF
IWDG
0x4000 3000 - 0x4000 33FF
WWDG
0x4000 2C00 - 0x4000 2FFF
RTC
0x4000 2800 - 0x4000 2BFF
Reserved
0x4000 1800 - 0x4000 27FF
TIM7
0x4000 1400 - 0x4000 17FF
TIM6
0x4000 1000 - 0x4000 13FF
TIM5
0x4000 0C00 - 0x4000 0FFF
TIM4
0x4000 0800 - 0x4000 0BFF
TIM3
0x4000 0400 - 0x4000 07FF
TIM2
0x4000 0000 - 0x4000 03FF
0x3FFF FFFF
Reserved
0x2001 0000
0x2000 FFFF
SRAM (64 KB aliased by bit-banding)
0x2000 0000
Option Bytes
0x1FFF F800 - 0x1FFF F80F
System memory
0x1FFF F000- 0x1FFF F7FF
0x1FFF EFFF
Reserved
0x0808 0000
0x0807 FFFF
Flash
0x0800 0000
0x07FF FFFF
Reserved
0x0008 0000
0x0007 FFFF
Aliased to Flash or system memory depending on BOOT pins
0x0000 0000
ai14753d

图 5-5 存储器映射（摘自参考手册-存储器映射章节）

5.4.2 存储器区域功能划分

在这 4GB 的地址空间中，ARM 已经粗线条的平均分成了 8 个块，每块 512MB，每个块也都规定了用途，具体分类见表格 5-1。每个块的大小都有 512MB，显然这是非常大的，芯片厂商在每个块的范围内设计各具特色的外设时并不一定都用得完，都是只用了其中的一部分而已。

表格 5-1 存储器功能分类

序号	用途	地址范围
Block 0	Code	0x0000 0000 ~ 0x1FFF FFFF(512MB)
Block 1	SRAM	0x2000 0000 ~ 0x3FFF FFFF(512MB)
Block 2	片上外设	0x4000 0000 ~ 0x5FFF FFFF(512MB)
Block 3	FSMC 的 bank1 ~ bank2	0x6000 0000 ~ 0x7FFF FFFF(512MB)
Block 4	FSMC 的 bank3 ~ bank4	0x8000 0000 ~ 0x9FFF FFFF(512MB)
Block 5	FSMC 寄存器	0xA000 0000 ~ 0xCFFF FFFF(512MB)
Block 6	没有使用	0xD000 0000 ~ 0xDFFF FFFF(512MB)
Block 7	Cortex-M3 内部外设	0xE000 0000 ~ 0xFFFF FFFF(512MB)

在这 8 个 Block 里面，有 3 个块非常重要，也是我们最关心的三个块。Block0 用来设计成内部 FLASH，Block1 用来设计成内部 RAM，Block2 用来设计成片上的外设，下面我们简单的介绍下这三个 Block 里面的具体区域的功能划分。

5.4.2.1 存储器 Block0 内部区域功能划分

Block0 主要用于设计片内的 FLASH，我们使用的 STM32F103RCT6（MINI）的 FLASH 为 256KB，而 STM32F103VET6（指南者和拂晓）和 STM32F103ZET6（霸道）的 FLASH 都是 512KB，这三者都属于大容量。要在芯片内部集成更大的 FLASH 或者 SRAM 都意味着芯片成本的增加，往往片内集成的 FLASH 都不会太大，ST 能在追求性价比的同时做到 512KB，实乃良心之举。Block 内部区域的功能划分具体见表格 5-2。

表格 5-2 存储器 Block0 内部区域功能划分

块	用途说明	地址范围
Block0	预留	0x1FFE C008 ~ 0x1FFF FFFF
	选项字节：用于配置读写保护、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。当芯片不小心被锁住之后，我们可以从 RAM 里面启动来修改这部分相应的寄存器位。	0x1FFF F800 - 0x1FFF F80F
	系统存储器：里面存的是 ST 出厂时烧写好的 isp 自举程序（即 Bootloader），用户无法改动。串口下载的时候需要用到这部分程序。	0x1FFF F000- 0x1FFF F7FF
	预留	0x0808 0000 ~ 0x1FFF EFFF
	FLASH：我们的程序就放在这里。	0x0800 0000 ~ 0x0807 FFFF (512KB)
	预留	0x0008 0000 ~ 0x07FF FFFF
	取决于 BOOT 引脚，为 FLASH、系统存储器、SRAM 的别名。	0x0000 0000 ~ 0x0007 FFFF

5.4.2.2 储存器 Block1 内部区域功能划分

Block1 用于设计片内的 SRAM。我们使用的 STM32F103RCT6（MINI）的 SRAM 为 48KB，STM32F103VET6（指南者和拂晓）和 STM32F103VET6（指南者）的 SRAM 都是 64KB，Block 内部区域的功能划分具体见表格 5-3。

表格 5-3 存储器 Block1 内部区域功能划分

块	用途说明	地址范围
Block1	预留	0x2001 0000 ~ 0x3FFF FFFF
	SRAM 64KB	0x2000 0000 ~0x2000 FFFF

5.4.2.3 储存器 Block2 内部区域功能划分

Block2 用于设计片内的外设，根据外设的总线速度不同，Block 被分成了 APB 和 AHB 两部分，其中 APB 又被分为 APB1 和 APB2，具体见表格 5-4。

表格 5-4 存储器 Block2 内部区域功能划分

块	用途说明	地址范围
Block2	APB1 总线外设	0x4000 0000 ~ 0x4000 77FF
	APB2 总线外设	0x4001 0000 ~ 0x4001 3FFF
	AHB 总线外设	0x4001 8000 ~ 0x5003 FFFF

5.5 寄存器映射

我们知道，存储器本身没有地址，给存储器分配地址的过程叫存储器映射，那什么叫寄存器映射？寄存器到底是什么？

在存储器 Block2 这块区域，设计的是片上外设，它们以四个字节为一个单元，共 32bit，每一个单元对应不同的功能，当我们控制这些单元时就可以驱动外设工作。我们可以找到每个单元的起始地址，然后通过 C 语言指针的操作方式来访问这些单元，如果每次都是通过这种地址的方式来访问，不仅不好记忆还容易出错，这时我们可以根据每个单元功能的不同，以功能为名给这个内存单元取一个别名，这个别名就是我们经常说的寄存器，这个给已经分配好地址的有特定功能的内存单元取别名的过程就叫寄存器映射。

比如，我们找到 GPIOB 端口的输出数据寄存器 ODR 的地址是 0x40010C0C（至于这个地址如何找到可以先跳过，后面我们会有详细的讲解），ODR 寄存器是 32bit，低 16bit 有效，对应着 16 个

外部 IO，写 0/1 对应的的 IO 则输出低/高电平。现在我们通过 C 语言指针的操作方式，让 GPIOB 的 16 个 IO 都输出高电平，具体见*代码 5_1*。

列表 1: 代码 5-1 通过绝对地址访问内存单元

```
// GPIOB 端口全部输出 高电平

*(unsigned int*)(0x4001 0C0C) = 0xFFFF;
```

0x40010C0C 在我们看来是 GPIOB 端口 ODR 的地址，但是在编译器看来，这只是一个普通的变量，是一个立即数，要想让编译器也认为是指针，我们得进行强制类型转换，把它转换成指针，即 (unsigned int *)0x4001 0C0C，然后再对这个指针进行 * 操作。

刚刚我们说了，通过绝对地址访问内存单元不好记忆且容易出错，我们可以通过寄存器别名的方式来操作，具体见*代码 5_2*（其中 GPIOB_BASE 指的是 GPIOB 端口的基地址，稍后会讲到）。

列表 2: 代码 5-2 通过寄存器别名方式访问内存单元

```
// GPIOB 端口全部输出 高电平

#define GPIOB_ODR (unsigned int *)(GPIOB_BASE+0x0C)

 * GPIOB_ODR = 0xFF;
```

为了方便操作，我们干脆把指针操作“*”也定义到寄存器别名里面，具体见*代码 5_3*。

列表 3: 代码 5-3 通过寄存器别名访问内存单元

```
//GPIOB 端口全部输出 高电平

#define GPIOB_ODR *(unsigned int *)(GPIOB_BASE+0x0C)

GPIOB_ODR = 0xFF;
```

5.5.1 STM32 的外设地址映射

片上外设区分为三条总线，根据外设速度的不同，不同总线挂载着不同的外设，APB1 挂载低速外设，APB2 和 AHB 挂载高速外设。相应总线的最低地址我们称为该总线的基地址，总线基地址也是挂载在该总线上的首个外设的地址。其中 APB1 总线的地址最低，片上外设从这里开始，也叫外设基地址。

5.5.1.1 总线基地址

表格 5-5 总线基地址

总线名称	总线基地址	相对外设基地址的偏移
APB1	0x4000 0000	0x0
APB2	0x4001 0000	0x0001 0000
AHB	0x4001 8000	0x0001 8000

表格 5-5 的“相对外设基地址偏移”即该总线地址与“片上外设”基地址 0x4000 0000 的差值。关于地址的偏移我们后面还会讲到。

5.5.1.2 外设基地址

总线上挂载着各种外设，这些外设也有自己的地址范围，特定外设的首个地址称为“XX 外设基地址”，也叫 XX 外设的边界地址。具体有关 STM32F10xx 外设的边界地址请参考《STM32F10xx 参考手册》的 2.3 小节的存储器映射的表 1：STM32F10xx 寄存器边界地址。

这里面我们以 GPIO 这个外设来讲解外设的基地址，GPIO 属于高速的外设，挂载到 APB2 总线上，具体见表格 5-6。

表格 5-6 外设 GPIO 基地址

外设名称	外设基地址	相对 APB2 总线的地址偏移
GPIOA	0x4001 0800	0x0000 0800
GPIOB	0x4001 0C00	0x0000 0C00
GPIOC	0x4001 1000	0x0000 1000
GPIOD	0x4001 1400	0x0000 1400
GPIOE	0x4001 1800	0x0000 1800
GPIOF	0x4001 1C00	0x0000 1C00
GPIOG	0x4001 2000	0x0000 2000

5.5.1.3 外设寄存器

在 XX 外设的地址范围内，分布着的就是该外设的寄存器。以 GPIO 外设为例，GPIO 是通用输入输出端口的简称，简单来说就是 STM32 可控制的引脚，基本功能是控制引脚输出高电平或者低电平。最简单的应用就是把 GPIO 的引脚连接到 LED 灯的阴极，LED 灯的阳极接电源，然后通过 STM32 控制该引脚的电平，从而实现控制 LED 灯的亮灭。

GPIO 有很多个寄存器，每一个都有特定的功能。每个寄存器为 32bit，占四个字节，在该外设的基地址上按照顺序排列，寄存器的位置都以相对该外设基地址的偏移地址来描述。这里我们以 GPIOB 端口为例，来说明 GPIO 都有哪些寄存器，具体见表格 5-7。

表格 5-7 GPIOB 端口的寄存器地址列表

寄存器名称	寄存器地址	相对 GPIOB 基址的偏移
GPIOB_CRL	0x4001 0C00	0x00
GPIOB_CRH	0x4001 0C04	0x04
GPIOB_IDR	0x4001 0C08	0x08
GPIOB_ODR	0x4001 0C0C	0x0C
GPIOH_BSRR	0x4001 0C10	0x10
GPIOH_BRR	0x4001 0C14	0x14
GPIOH_LCKR	0x4001 0C18	0x18

有关外设的寄存器说明可参考《STM32F10xx 参考手册》中具体章节的寄存器描述部分，在编程的时候我们需要反复的查阅外设的寄存器说明。

这里我们以“GPIO 端口置位/复位寄存器”为例，教大家如何理解寄存器的说明，具体见图 5-6。

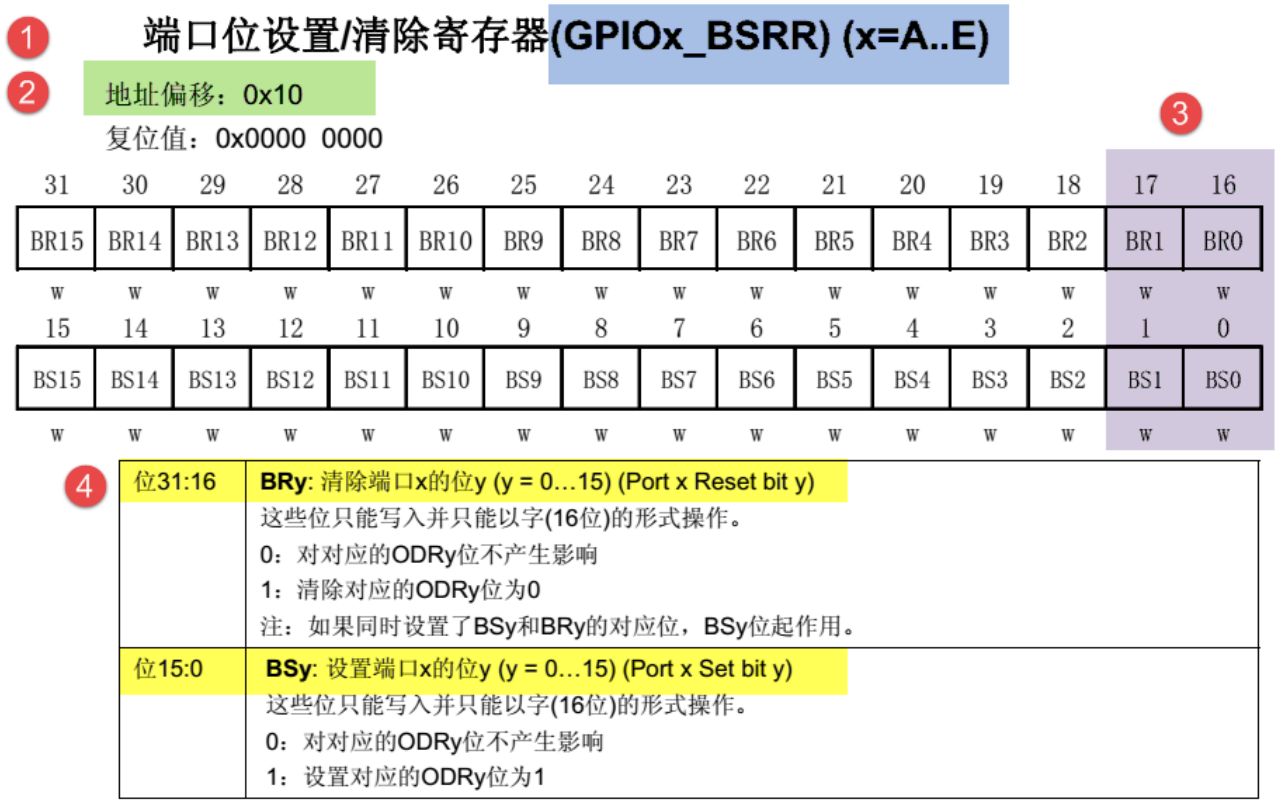


图 5-6 GPIO 端口置位/复位寄存器说明

- ① 名称

寄存器说明中首先列出了该寄存器中的名称，“(GPIOx_BSRR)(x=A…E)”这段的意思是该寄存器名为“GPIOx_BSRR”其中的“x”可以为 A-E，也就是说这个寄存器说明适用于 GPIOA、GPIOB 至 GPIOE，这些 GPIO 端口都有这样的一个寄存器。

- ② 偏移地址

偏移地址是指本寄存器相对于这个外设的基地址的偏移。本寄存器的偏移地址是 0x10，从参考手册中我们可以查到 GPIOA 外设的基地址为 0x4001 0800 ，我们就可以算出 GPIOA 的这个 GPIOA_BSRR 寄存器的地址为：0x4001 0800+0x10 ；同理，由于 GPIOB 的外设基地址为 0x4001 0C00，可算出 GPIOB_BSRR 寄存器的地址为：0x4001 0C00+0x10 。其他 GPIO 端口以此类推即可。

- ③ 寄存器位表

紧接着的是本寄存器的位表，表中列出它的 0-31 位的名称及权限。表上方的数字为位编号，中间为位名称，最下方为读写权限，其中 w 表示只写，r 表示只读，rw 表示可读写。本寄存器中的位权限都是 w，所以只能写，如果读本寄存器，是无法保证读取到它真正内容的。而有的寄存器位只读，一般是用于表示 STM32 外设的某种工作状态的，由 STM32 硬件自动更改，程序通过读取那些寄存器位来判断外设的工作状态。

- ④ 位功能说明

位功能是寄存器说明中最重要的部分，它详细介绍了寄存器每一个位的功能。例如本寄存器中有两种寄存器位，分别为 BRy 及 BSy，其中的 y 数值可以是 0-15，这里的 0-15 表示端口的引脚号，如 BR0、BS0 用于控制 GPIOx 的第 0 个引脚，若 x 表示 GPIOA，那就是控制 GPIOA 的第 0 引脚，而 BR1、BS1 就是控制 GPIOA 第 1 个引脚。

其中 BRy 引脚的说明是“0：不会对相应的 ODRy 位执行任何操作；1：对相应 ODRy 位进行复位”。这里的“复位”是将该位设置为 0 的意思，而“置位”表示将该位设置为 1；说明中的 ODRy 是另一个寄存器的寄存器位，我们只需要知道 ODRy 位为 1 的时候，对应的引脚 y 输出高电平，为 0 的时候对应的引脚输出低电平即可 (感兴趣的读者可以查询该寄存器 GPIOx_ODR 的说明了解)。所以，如果对 BR0 写入“1”的话，那么 GPIOx 的第 0 个引脚就会输出“低电平”，但是对 BR0 写入“0”的话，却不会影响 ODR0 位，所以引脚电平不会改变。要想该引脚输出“高电平”，就需要对“BS0”位写入“1”，寄存器位 BSy 与 BRy 是相反的操作。

5.5.2 C 语言对寄存器的封装

以上所有的关于存储器映射的内容，最终都是为大家更好地理解如何用 C 语言控制读写外设寄存器做准备，此处是本章的重点内容。

5.5.2.1 封装总线和外设基地址

在编程上为了方便理解和记忆，我们把总线基地址和外设基地址都以相应的宏定义起来，总线或者外设都以他们的名字作为宏名，具体见*代码 5_4*。

列表 4: 代码 5-4 总线和外设基址宏定义

```
/* 外设基地址 */

#define PERIPH_BASE ((unsigned int)0x40000000)

/* 总线基地址 */

#define APB1PERIPH_BASE PERIPH_BASE

#define APB2PERIPH_BASE (PERIPH_BASE + 0x00010000)

#define AHBPERIPH_BASE (PERIPH_BASE + 0x00020000)

/* GPIO 外设基地址 */

#define GPIOA_BASE (APB2PERIPH_BASE + 0x0800)

#define GPIOB_BASE (APB2PERIPH_BASE + 0x0C00)

#define GPIOC_BASE (APB2PERIPH_BASE + 0x1000)

#define GPIOD_BASE (APB2PERIPH_BASE + 0x1400)
```

(下页继续)

(续上页)

```
#define GPIOE_BASE (APB2PERIPH_BASE + 0x1800)

#define GPIOF_BASE (APB2PERIPH_BASE + 0x1C00)

#define GPIOG_BASE (APB2PERIPH_BASE + 0x2000)

/* 寄存器基地址，以 GPIOB 为例 */

#define GPIOB_CRL (GPIOB_BASE+0x00)

#define GPIOB_CRH (GPIOB_BASE+0x04)

#define GPIOB_IDR (GPIOB_BASE+0x08)

#define GPIOB_ODR (GPIOB_BASE+0x0C)

#define GPIOB_BSRR (GPIOB_BASE+0x10)

#define GPIOB_BRR (GPIOB_BASE+0x14)

#define GPIOB_LCKR (GPIOB_BASE+0x18)
```

代码 5-4 首先定义了“片上外设”基地址 PERIPH_BASE，接着在 PERIPH_BASE 上加入各个总线的地址偏移，得到 APB1、APB2 总线的地址 APB1PERIPH_BASE、APB2PERIPH_BASE，在其之上加入外设地址的偏移，得到 GPIOA-G 的外设地址，最后在外设地址上加入各寄存器的地址偏移，得到特定寄存器的地址。一旦有了具体地址，就可以用指针读写，具体见代码 5_5。

列表 5: 代码 5-5 使用指针控制 BSRR 寄存器

```
/* 控制 GPIOB 引脚 0 输出低电平 (BSRR 寄存器的 BR0 置 1) */
*(unsigned int *)GPIOB_BSRR = (0x01<<(16+0));
```

(下页继续)

(续上页)

```
/* 控制 GPIOB 引脚 0 输出高电平 (BSRR 寄存器的 BS0 置 1) */
*(unsigned int *)GPIOB_BSRR = 0x01<<0;

unsigned int temp;
/* 读取 GPIOB 端口所有引脚的电平 (读 IDR 寄存器) */
temp = *(unsigned int *)GPIOB_IDR;
```

该代码使用 (unsigned int *) 把 GPIOB_BSRR 宏的数值强制转换成了地址，然后再用“*”号做取指针操作，对该地址的赋值，从而实现了写寄存器的功能。同样，读寄存器也是用取指针操作，把寄存器中的数据取到变量里，从而获取 STM32 外设的状态。

5.5.2.2 封装寄存器列表

用上面的方法去定义地址，还是稍显繁琐，例如 GPIOA-GPIOE 都各有一组功能相同的寄存器，如 GPIOA_ODR/GPIOB_ODR/GPIOC_ODR 等等，它们只是地址不一样，但却要为每个寄存器都定义它的地址。为了更方便地访问寄存器，我们引入 C 语言中的结构体语法对寄存器进行封装，具体见*代码 5_6*。

列表 6: 代码 5-6 使用结构体对 GPIO 寄存器组的封装

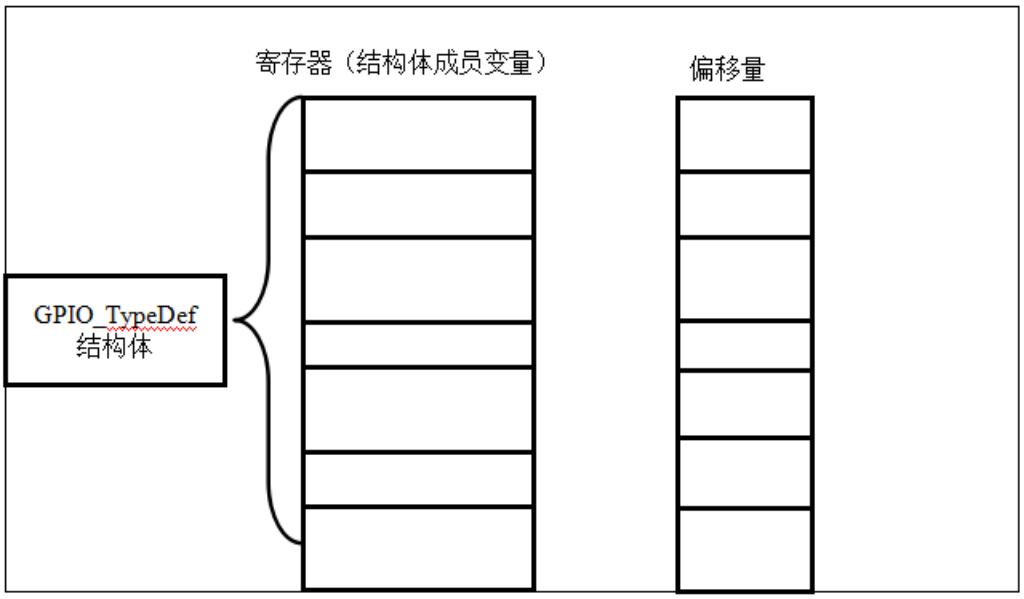
```
typedef unsigned           int uint32_t; /* 无符号 32 位变量 */
typedef unsigned short     int uint16_t; /* 无符号 16 位变量 */

/* GPIO 寄存器列表 */
typedef struct {
    uint32_t CRL;      /*GPIO 端口配置低寄存器     地址偏移: 0x00 */
    uint32_t CRH;      /*GPIO 端口配置高寄存器     地址偏移: 0x04 */
    uint32_t IDR;      /*GPIO 数据输入寄存器       地址偏移: 0x08 */
    uint32_t ODR;      /*GPIO 数据输出寄存器       地址偏移: 0x0C */
    uint32_t BSRR;     /*GPIO 位设置/清除寄存器    地址偏移: 0x10 */
    uint32_t BRR;      /*GPIO 端口位清除寄存器      地址偏移: 0x14 */
```

(下页继续)

(续上页)

```
    uint16_t LCKR;     /*GPIO 端口配置锁定寄存器    地址偏移：0x18 */
} GPIO_TypeDef;
```

这段代码用 typedef 关键字声明了名为 GPIO_TypeDef 的结构体类型，结构体内有 7 个成员变量，变量名正好对应寄存器的名字。C 语言的语法规定，结构体内变量的存储空间是连续的，其中 32 位的变量占用 4 个字节，16 位的变量占用 2 个字节，具体见图 *5_7*。



也就是说，我们定义的这个 GPIO_TypeDef，假如这个结构体的首地址为 0x4001 0C00（这也是第一个成员变量 CRL 的地址），那么结构体中第二个成员变量 CRH 的地址即为 0x4001 0C00 +0x04，加上的这个 0x04，正是代表 CRL 所占用的 4 个字节地址的偏移量，其它成员变量相对于结构体首地址的偏移，在上述代码右侧注释已给。

这样的地址偏移与 STM32 GPIO 外设定义的寄存器地址偏移一一对应，只要给结构体设置好首地址，就能把结构体内成员的地址确定下来，然后就能以结构体的形式访问寄存器，具体见*代码 5_7*。

列表 7: 代码 5-7 通过结构体指针访问寄存器

```
GPIO_TypeDef * GPIOx; //定义一个 GPIO_TypeDef 型结构体指针 GPIOx
GPIOx = GPIOB_BASE; //把指针地址设置为宏 GPIOB_BASE 地址
GPIOx->IDR = 0xFFFF;
GPIOx->ODR = 0xFFFF;
uint32_t temp;
temp = GPIOx->IDR; //读取 GPIOB_IDR 寄存器的值到变量 temp 中
```

这段代码先用 GPIO_TypeDef 类型定义一个结构体指针 GPIOx，并让指针指向地址 GPIOB_BASE(0x4001 0C00)，使用地址确定下来，然后根据 C 语言访问结构体的语法，用 GPIOx->ODR 及 GPIOx->IDR 等方式读写寄存器。

最后，我们更进一步，直接使用宏定义好 GPIO_TypeDef 类型的指针，而且指针指向各个 GPIO 端口的首地址，使用时我们直接用该宏访问寄存器即可，具体代码 5_8。

列表 8: 代码 5-8 定义好 GPIO 端口首地址址针

```
/* 使用 GPIO_TypeDef 把地址强制转换成指针 */
#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)

/* 使用定义好的宏直接访问 */
/* 访问 GPIOB 端口的寄存器 */
GPIOB->BSRR = 0xFFFF;        //通过指针访问并修改 GPIOB_BSRR 寄存器
GPIOB->CRL = 0xFFFF;         //修改 GPIOB_CRL 寄存器
GPIOB->ODR =0xFFFF;          //修改 GPIOB_ODR 寄存器
```

(下页继续)

(续上页)

```
uint32_t temp;
temp = GPIOB->IDR;            //读取 GPIOB_IDR 寄存器的值到变量 temp 中

/* 访问 GPIOA 端口的寄存器 */
GPIOA->BSRR = 0xFFFF;
GPIOA->CRL = 0xFFFF;
GPIOA->ODR =0xFFFF;

//uint32_t temp; //该变量前面已定义
temp = GPIOA->IDR;            //读取 GPIOA_IDR 寄存器的值到变量 temp 中
```

这里我们仅是以 GPIO 这个外设为例，给大家讲解了 C 语言对寄存器的封装。以此类推，其他外设也同样可以用这种方法来封装。好消息是，这部分工作都由固件库帮我们完成了，这里我们只是分析了下这个封装的过程，让大家知其然，也只其所以然。

5.5.3 修改寄存器的位操作方法

使用 C 语言对寄存器赋值时，我们常常要求只修改该寄存器的某一位或者某几位的值，且其它的寄存器位不变，这个时候我们就需要用到 C 语言的位操作方法了。

5.5.3.1 把变量的某位或某几个位进行清零（按位与 &）

此处我们以变量 a 代表一个 8 位的寄存器（STM32 的寄存器是 32 位的），并假设寄存器中本来已有数值，此时我们需要把变量 a 的某一位清零，且其它位不变；由于寄存器中有时也会存在连续几个寄存器位用于控制某个功能的情况，所以我们也假设需要把寄存器的某几个连续位清零，且其它位不变。方法见*代码清单 5_1*。

列表 9: 代码清单 5-1 对某位清零、以及对某几位清零

```
/* 对某位清零 */

 //定义一个变量 a = 1001 1111 b (二进制数)
 unsigned char a = 0x9f;



 //对 bit2 清零
 a &= ~(1<<2);

 //上面一行代码右边括号中的 1 左移两位, (1<<2) 得二进制数: 0000 0100 b (这个便是位 2
的掩码)
 //然后按位取反, ~(1<<2) 得 1111 1011 b
 //假如 a 中原来的值为二进制数: a = 1001 1111 b
 //所得的数与 a 作“位与&”运算: a = (1001 1111 b) & (1111 1011 b)
 //经过运算后, a 的值 a=1001 1011 b
 //这样, a 的 bit2 位就被清零, 而其它位不变。



 /* 下面是对某几个连续位清零 */

 //同样我们首先定义一个变量 a = 1001 1111 b (二进制数)
 unsigned char a = 0x9f;

 //若把 a 中的二进制位分成 2 个一组
 //即 bit0、bit1 为第 0 组, bit2、bit3 为第 1 组,
 //   bit4、bit5 为第 2 组, bit6、bit7 为第 3 组
 //现在, 我们要对第 1 组的 bit2、bit3 清零

 a &= ~(3<<2*1);
```

(下页继续)

(续上页)

```
//括号中的 3 左移两位，(3<<2*1) 得二进制数：0000 1100 b（这个是位 3、位 2 的掩码）
//然后按位取反，~(3<<2*1) 得 1111 0011 b
//假如 a 中原来的值为二进制数： a = 1001 1111 b
//所得的数与 a 作”位与&”运算： a = (1001 1111 b) & (1111 0011 b)
//经过运算后，a 的值 a=1001 0011 b
//最后 a 的第 1 组的 bit2、bit3 就被清零了，而其它位不变。

//上述 (~(3<<2*1)) 中的 1 即为组编号；如清零第 3 组 bit6、bit7 此处应为 3，即要左移
6 位
//括号中的 2 为每组的位数，每组有 2 个二进制位；若分成 4 个一组，此处即为 4
//括号中的 3 是组内所有位都为 1 时的值；若分成 4 个一组，此处即为二进制数“1111 b”

//例如对第 2 组 bit4、bit5 清零，3 就要左移 4 位
a &= ~(3<<2*2);
```

5.5.3.2 对变量的某一位或某几位进行赋值（按位或 |）

寄存器位经过上面的清零操作后，接下来就可以方便地对某几位写入所需要的数值了，且其它位不变，方法见代码清单 5_3，这时候写入的数值一般就是需要设置寄存器的位参数。

列表 10: 代码清单 5-3 对某几位进行赋值

```
/* 对某位赋值 */

//假设 a = 1000 0011 b
a |= (1<<4);
//此时对变量 a 的 bit4 置 1
//置 1 后，即 a = 1001 0011 b
```

(下页继续)

(续上页)

```
/* 对某几位进行赋值 */

 //假设 a = 1000 0011 b
 a |= (1<<2*2);
 //此时对清零后的第 2 组 bit4、bit5 设置成二进制数“01 b ”(也就是“01 b”左移 4 位)
 //即 a = 1001 0011 b，成功设置了第 2 组的值，其它组不变
```

5.5.3.3 对变量的某位取反（按位异或 ^）

某些情况下，我们需要对寄存器的某个位进行取反操作，即 1 变 0 ，0 变 1，这可以直接用如下的操作，其它位不变，见代码清单 5_4。

列表 11: 代码清单 5-4 对某位进行取反操作

```
 //a = 1001 0011 b
 //把 bit6 取反，其它位不变

 a ^=(1<<6);
 //a = 1101 0011 b
```

关于修改寄存器位的这些操作，在下一章中有应用实例代码，可配合阅读。

第 6 章　新建工程—寄存器版

本章内容所涉及的软件只供教学使用，不得用于商业用途。个人或公司因商业用途导致的法律责任，后果自负。

版本说明：MDK5.15 (即 Keil MDK 软件)，如果有更高的版本可使用高版本。

版本号可从 MDK 软件的菜单栏的“Help–>About uVision”选项中查询到。

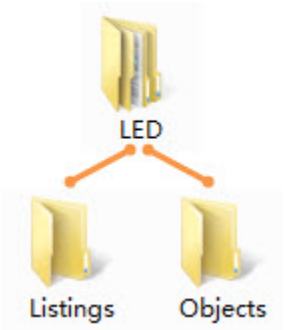
6.1 新建工程

6.1.1 新建本地工程文件夹

为了工程目录更加清晰，我们在本地电脑上新建 1 个文件夹用于存放整个工程，如命名为“LED”，然后在该目录下新建 2 个文件夹，具体如下：

表格 6-1 工程目录文件夹清单

名称	作用
Listing	存放编译器编译时候产生的 c/汇编/链接的列表清单
Output	存放编译产生的调试信息、hex 文件、预览信息、封装库等



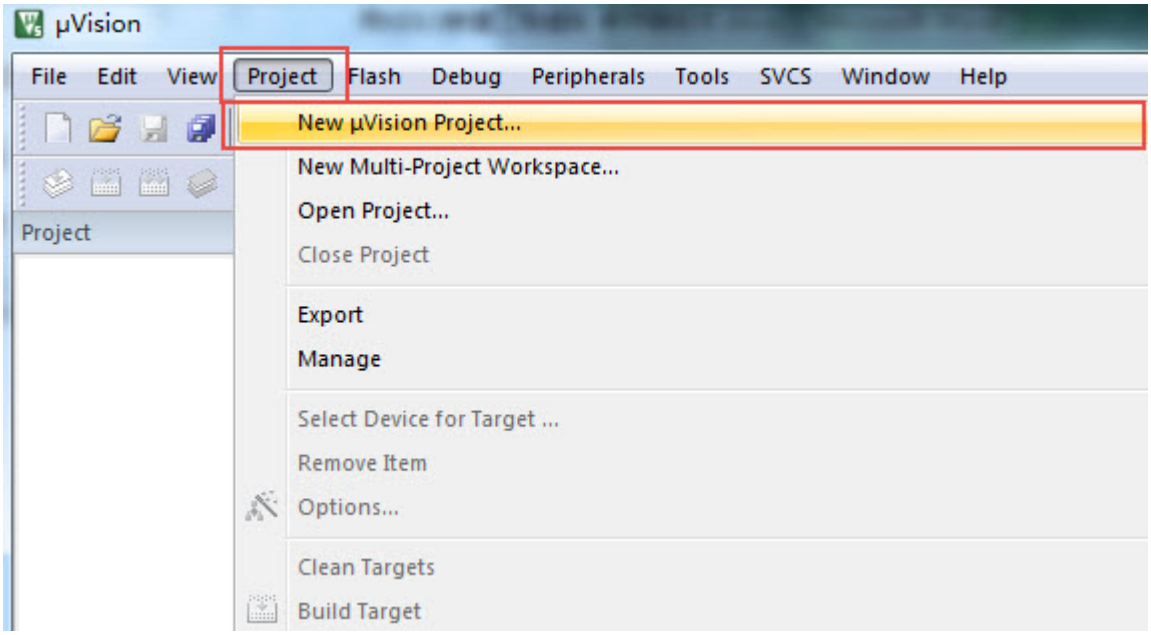
在本地新建好文件夹后，在文件夹下新建一些文件：

表格 6-2 工程目录文件夹内容清单

名称	作用
LED	存放 startup_stm32f10x_hd.s、stm32f10x.h、main.c 文件
Listing	暂时为空
Output	暂时为空

6.1.2 新建工程

打开 KEIL5，新建一个工程，工程名根据喜好命名，我这里取 LED-REG，直接保存在 LED 文件夹下。

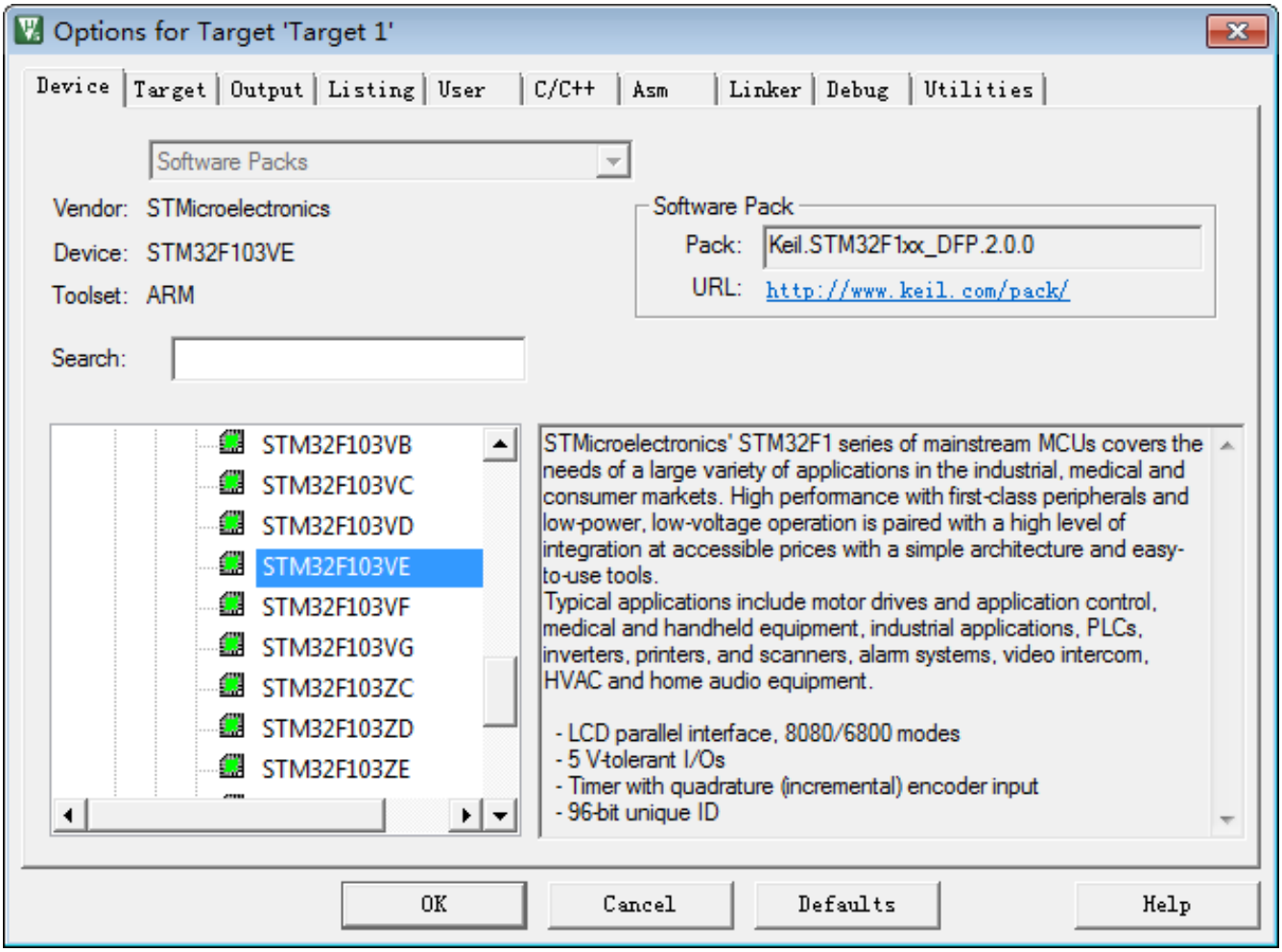


6.1.2.1 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，具体如下：

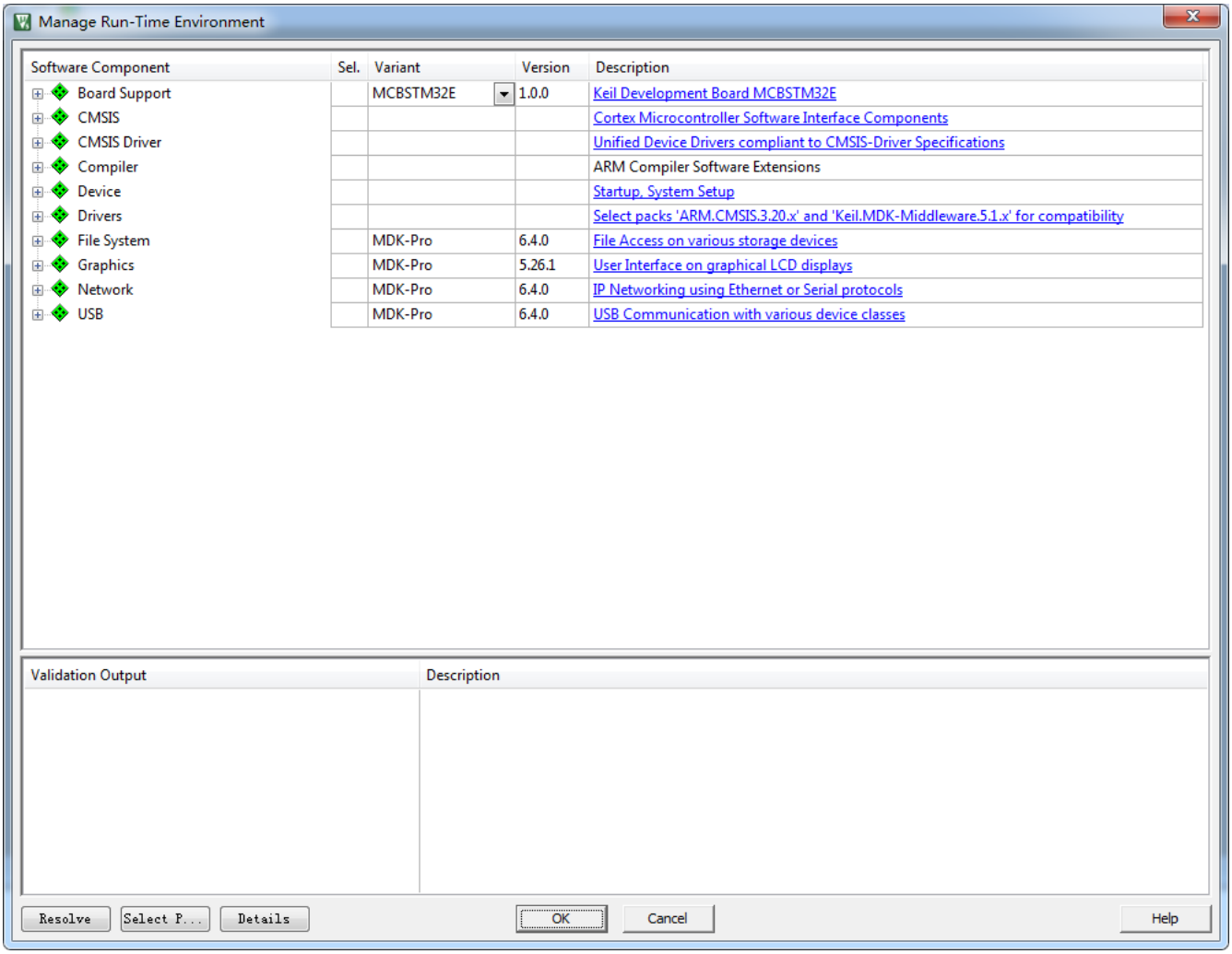
- F103-MINI，选 STM32F103RC 型号
- F103-指南者，选 STM32F103VE 型号
- F103-霸道，选 STM32F103ZE 型号
- F103-拂晓电机开发板，选 STM32F103VE 型号

如果这里没有出现你想要的 CPU 型号，或者一个型号都没有，那么肯定是你的 KEIL5 没有添加 device 库，因为 KEIL5 不像 KEIL4 那样自带了很多 MCU 的型号，KEIL5 需要自己添加，关于如何添加请参考《如何安装 KEIL5》这一章。



6.1.2.2 在线添加库文件

用寄存器控制 STM32 时我们不需要在线添加库文件，这里我们点击关掉。



6.1.2.3 添加文件

在新建的工程中添加文件，文件从本地建好的工程文件夹下获取，双击组文件夹就会出现添加文件的路径，然后选择文件即可。我们对要添加的三个文件说明如下：

startup_stm32f10x_hd.s

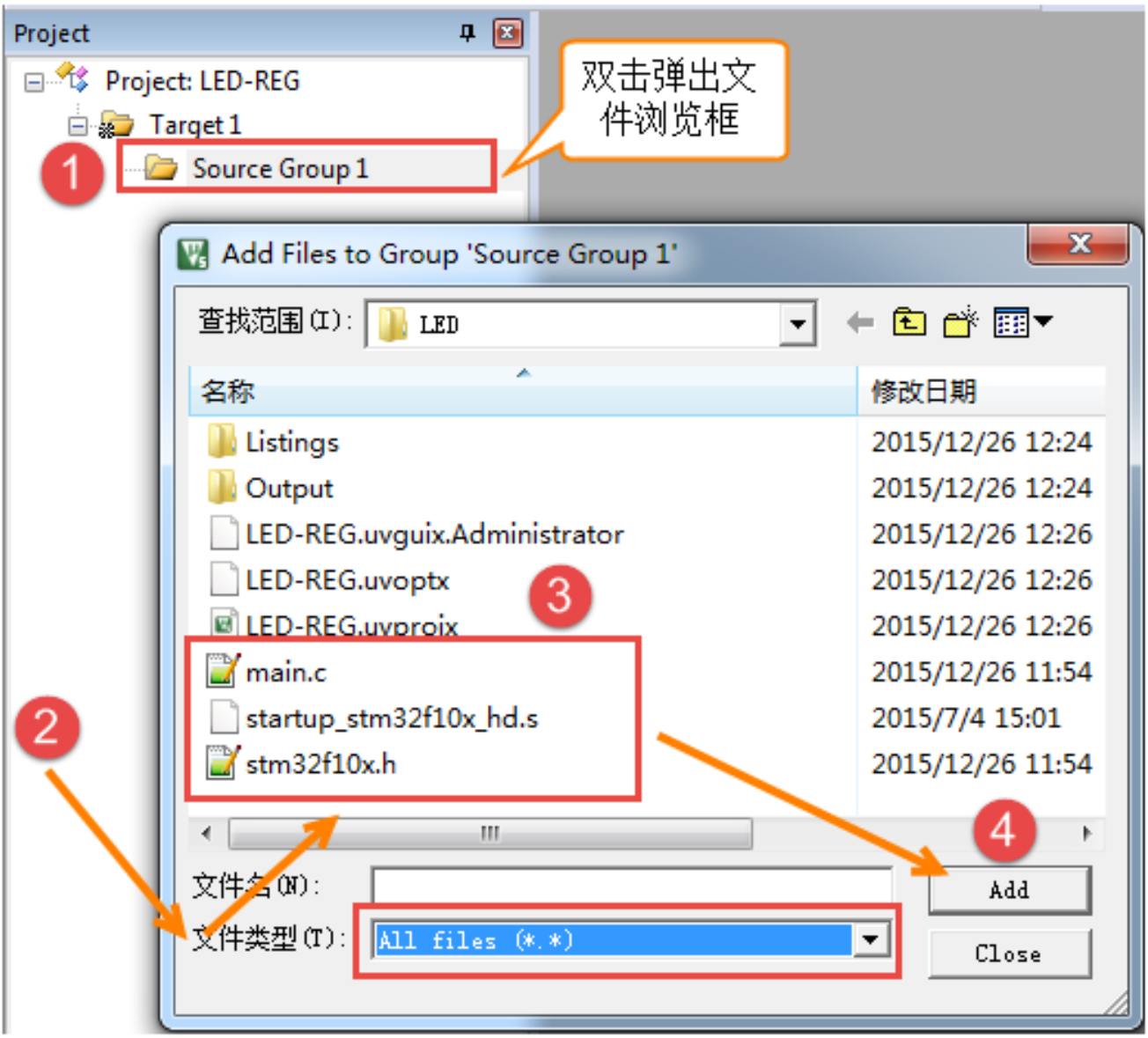
启动文件，系统上电后第一个运行的程序，由汇编编写，C 编程用的比较少，可暂时不管，这个文件从固件库里面拷贝而来，由官方提供。文件在这个目录：STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm\startup_stm32f10x_hd.s。

stm32f10x.h

用户手动新建，用于存放寄存器映射的代码，暂时为空。

main.c

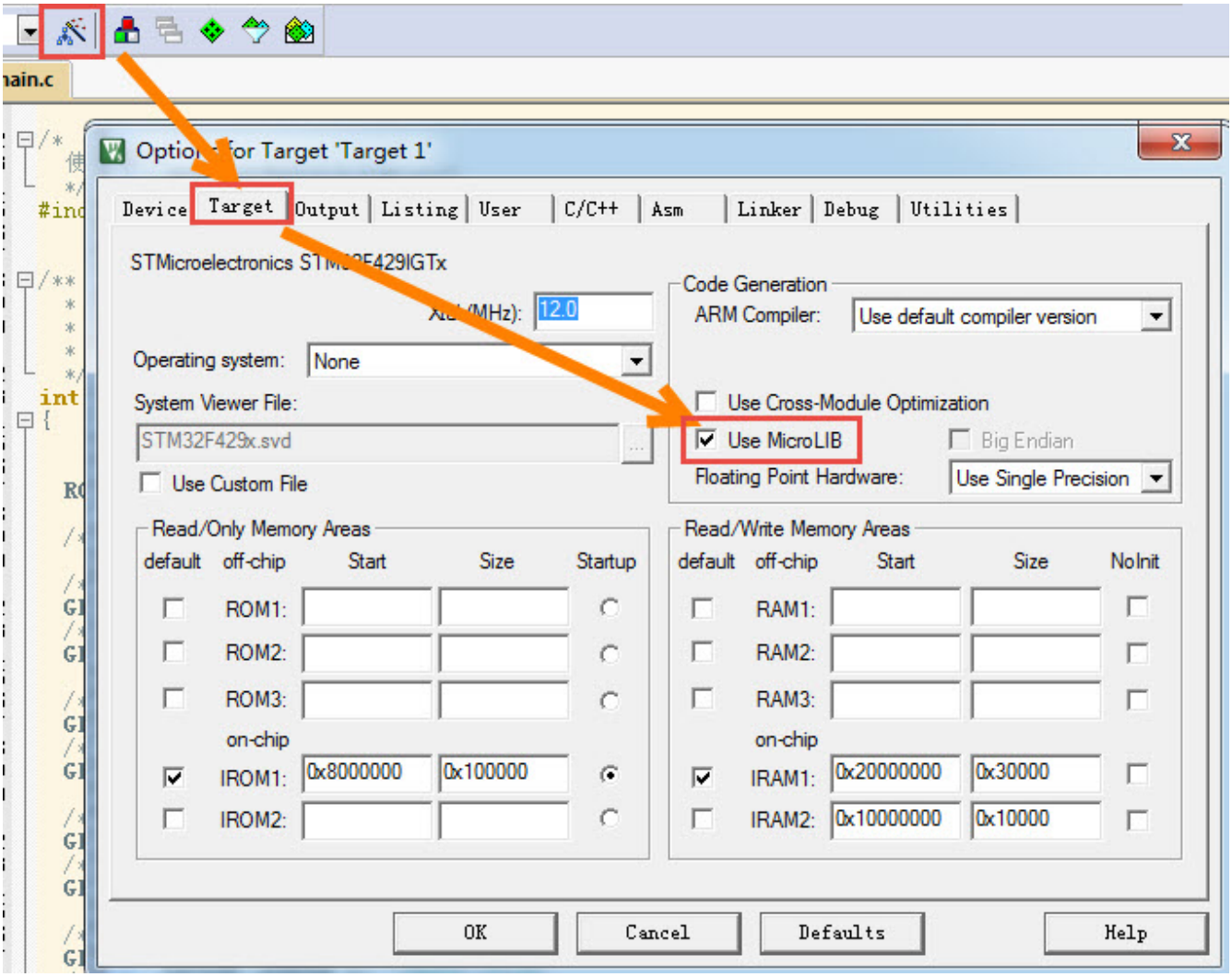
用户手动新建，用于存放 main 函数，暂时为空。



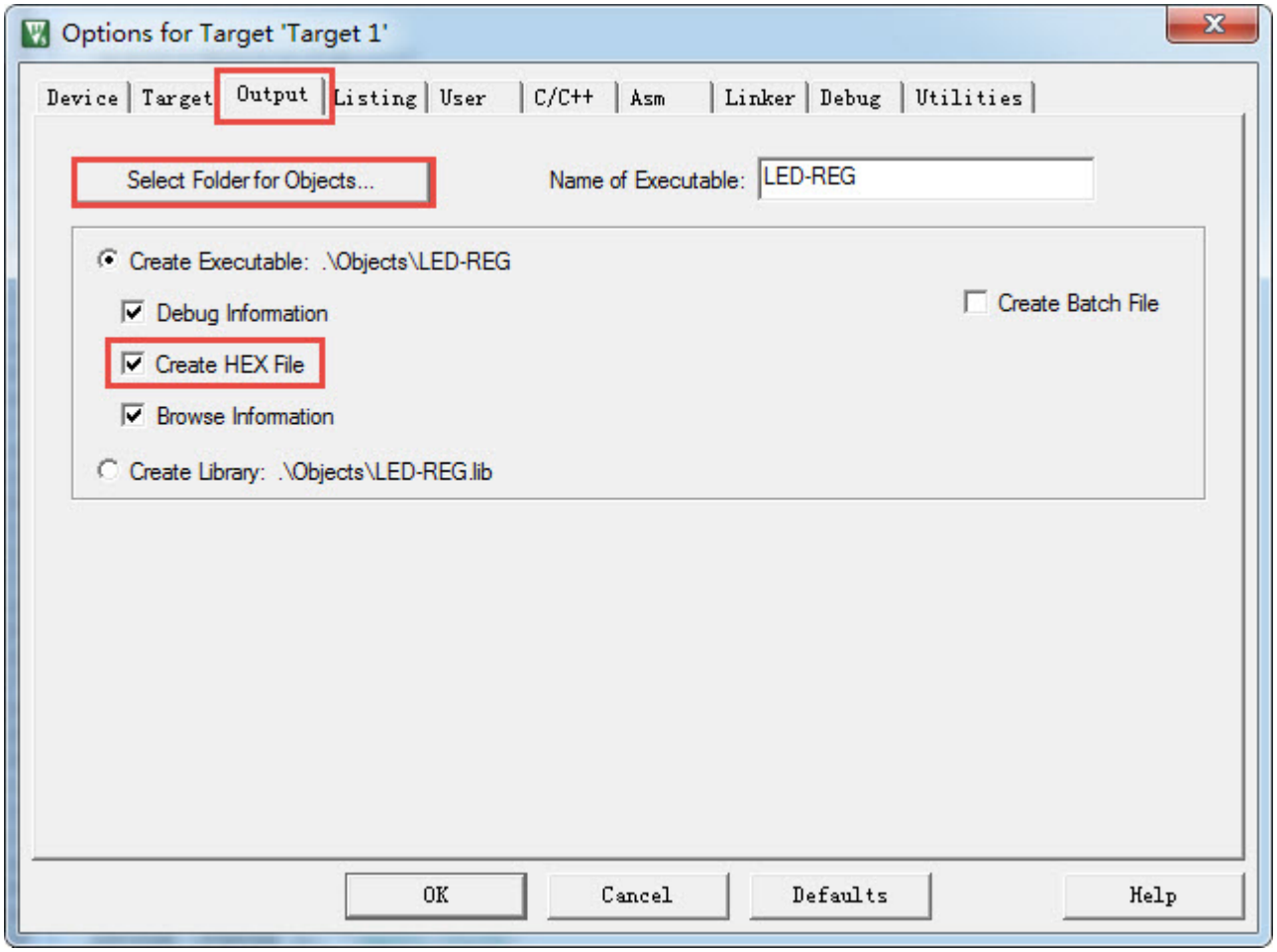
6.1.2.4 配置魔术棒选项卡

这一步的配置工作很重要，很多人串口用不了 printf 函数，编译有问题，下载有问题，都是这个步骤的配置出了错。

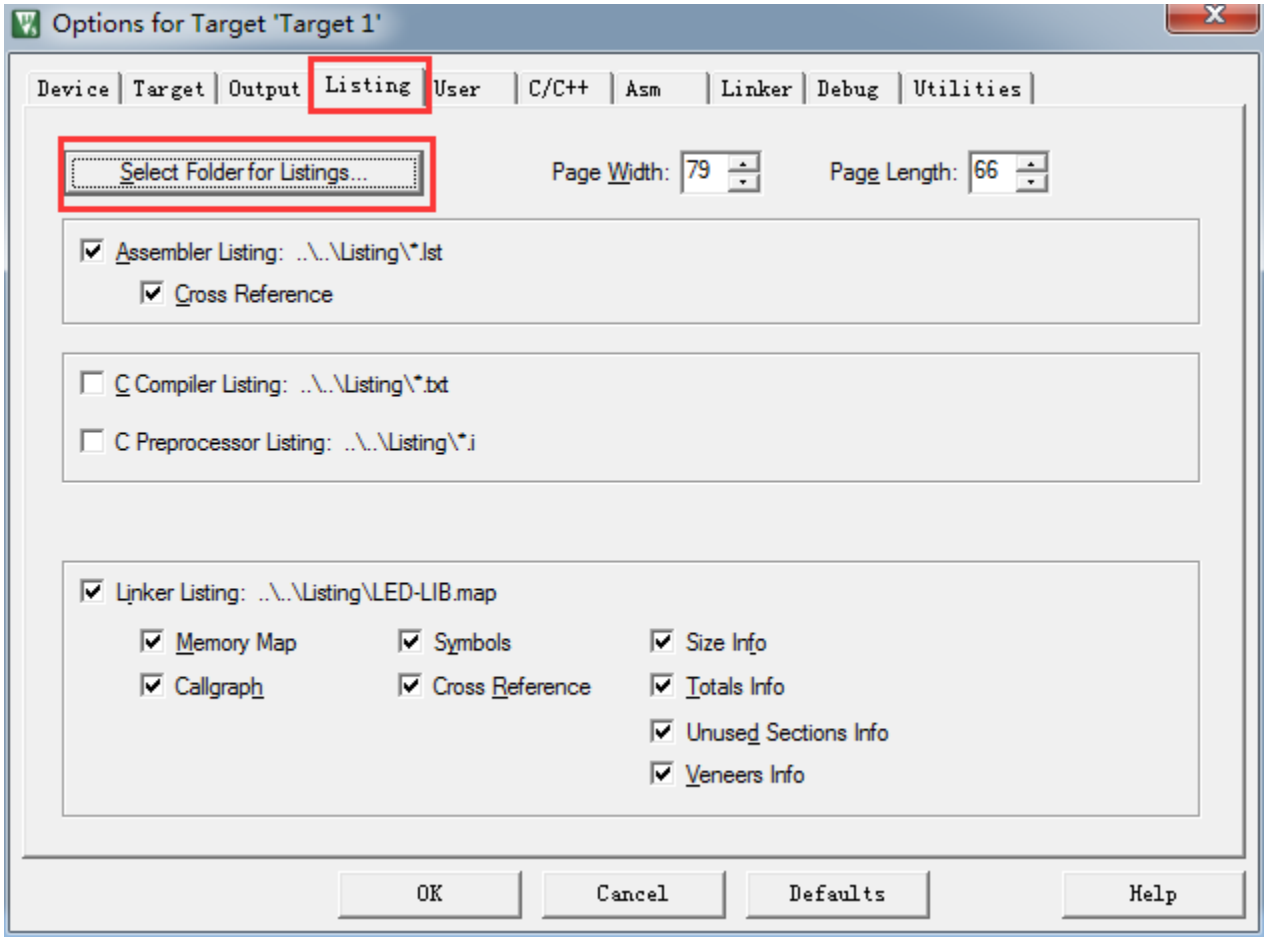
①Target 中选中微库“Use MicroLib”，为的是在日后编写串口驱动的时候可以使用 printf 函数。



②Output 选项卡中把输出文件夹定位到我们工程目录下的 output 文件夹，如果想在编译的过程中生成 hex 文件，那么那 Create HEX File 选项勾上。



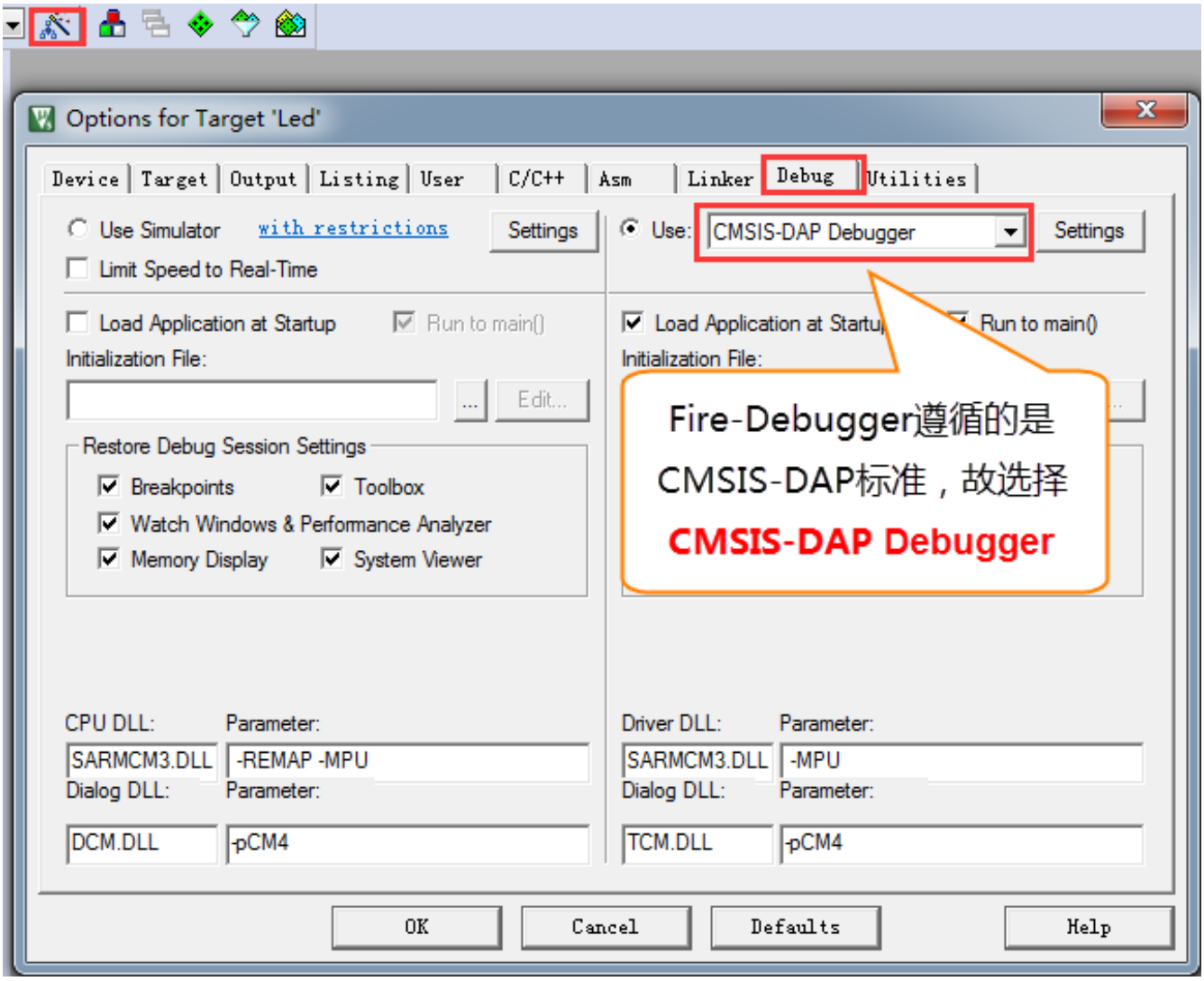
③ 在 Listing 选项卡中把输出文件夹定位到我们工程目录下的 Listing 文件夹。



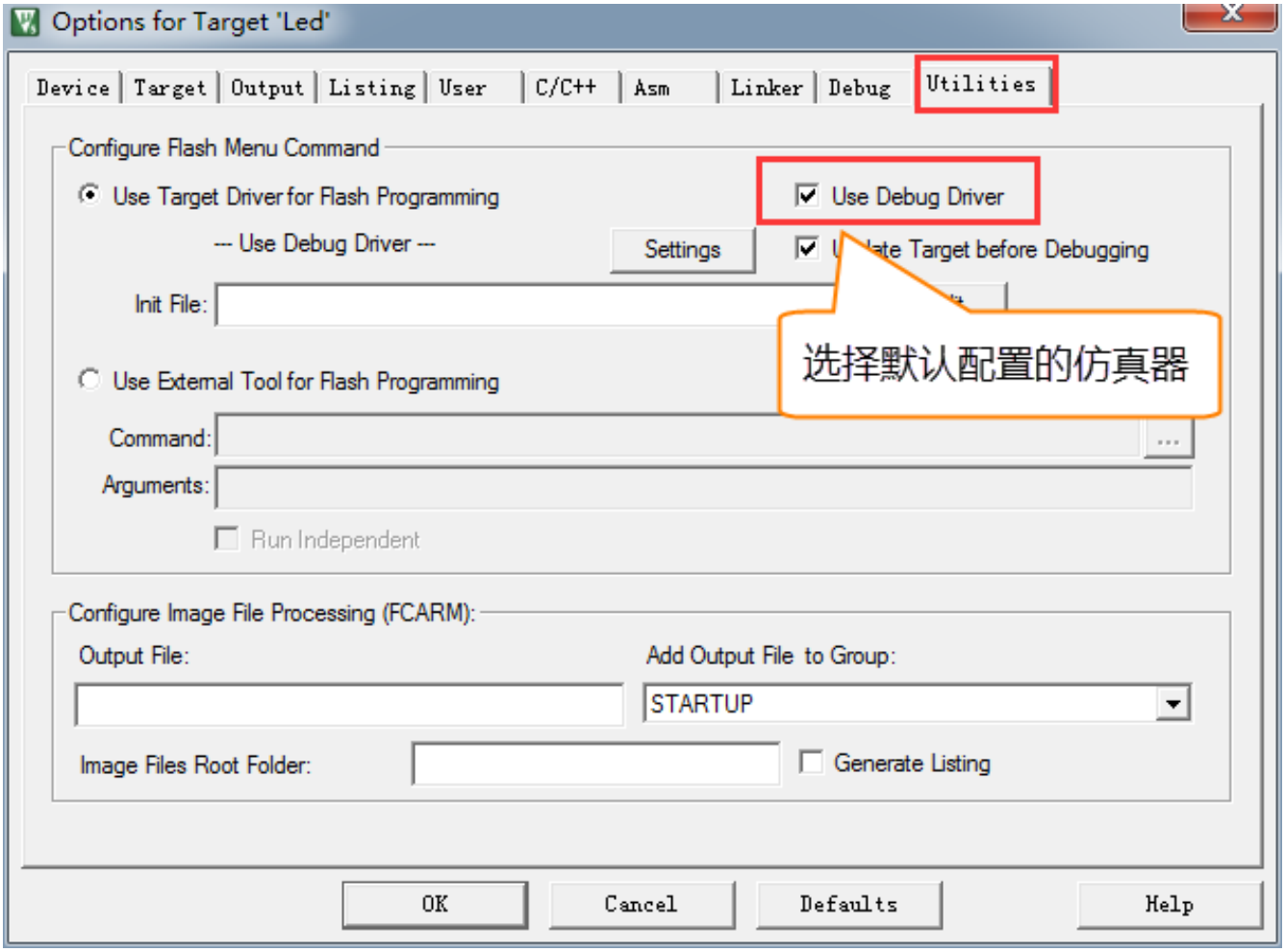
6.1.2.5 下载器配置

在仿真器连接好电脑和开发板且开发板供电正常的情况下，打开编译软件 KEIL，在魔术棒选项卡里面选择仿真器的型号，具体过程看图示：

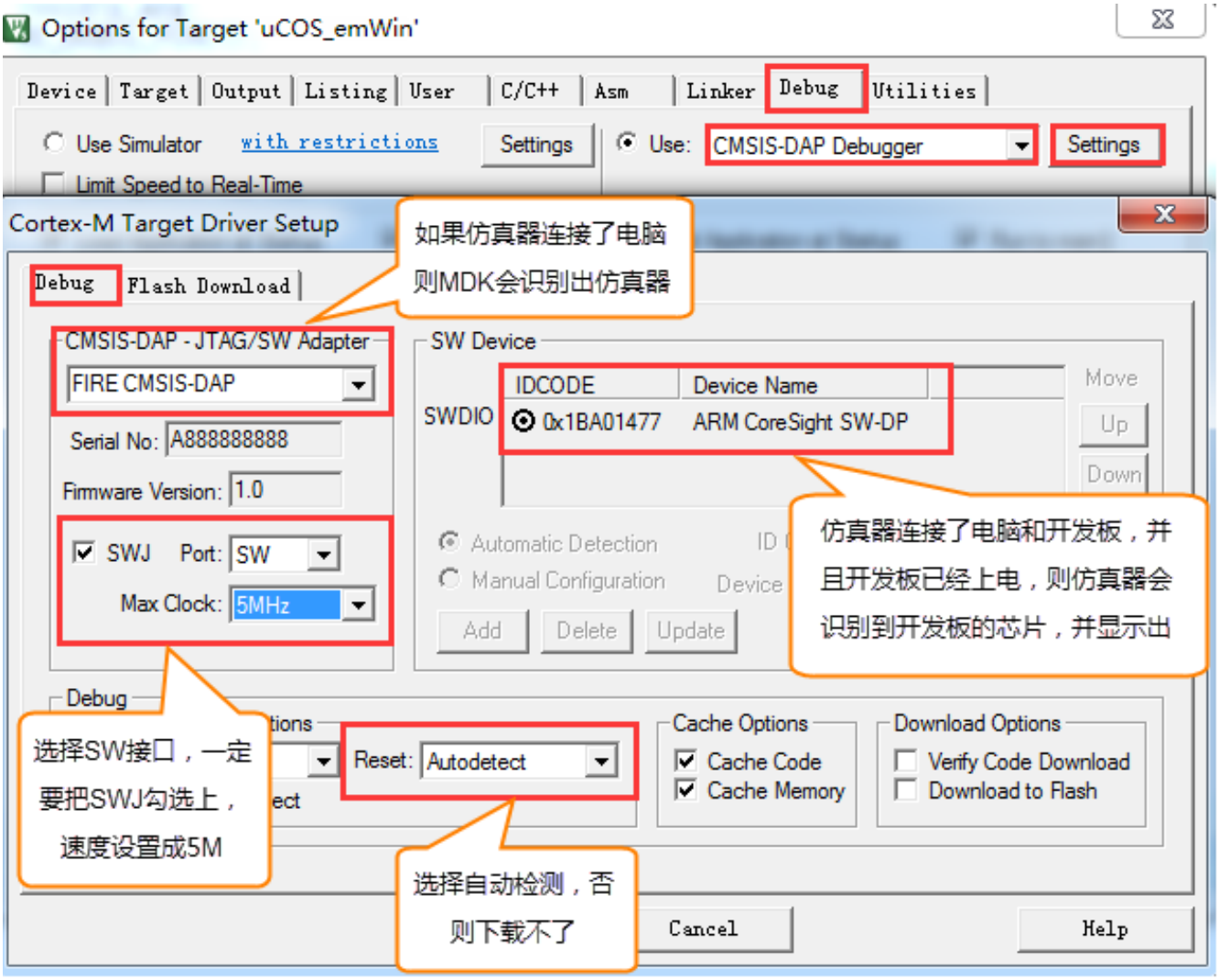
Debug 选项配置



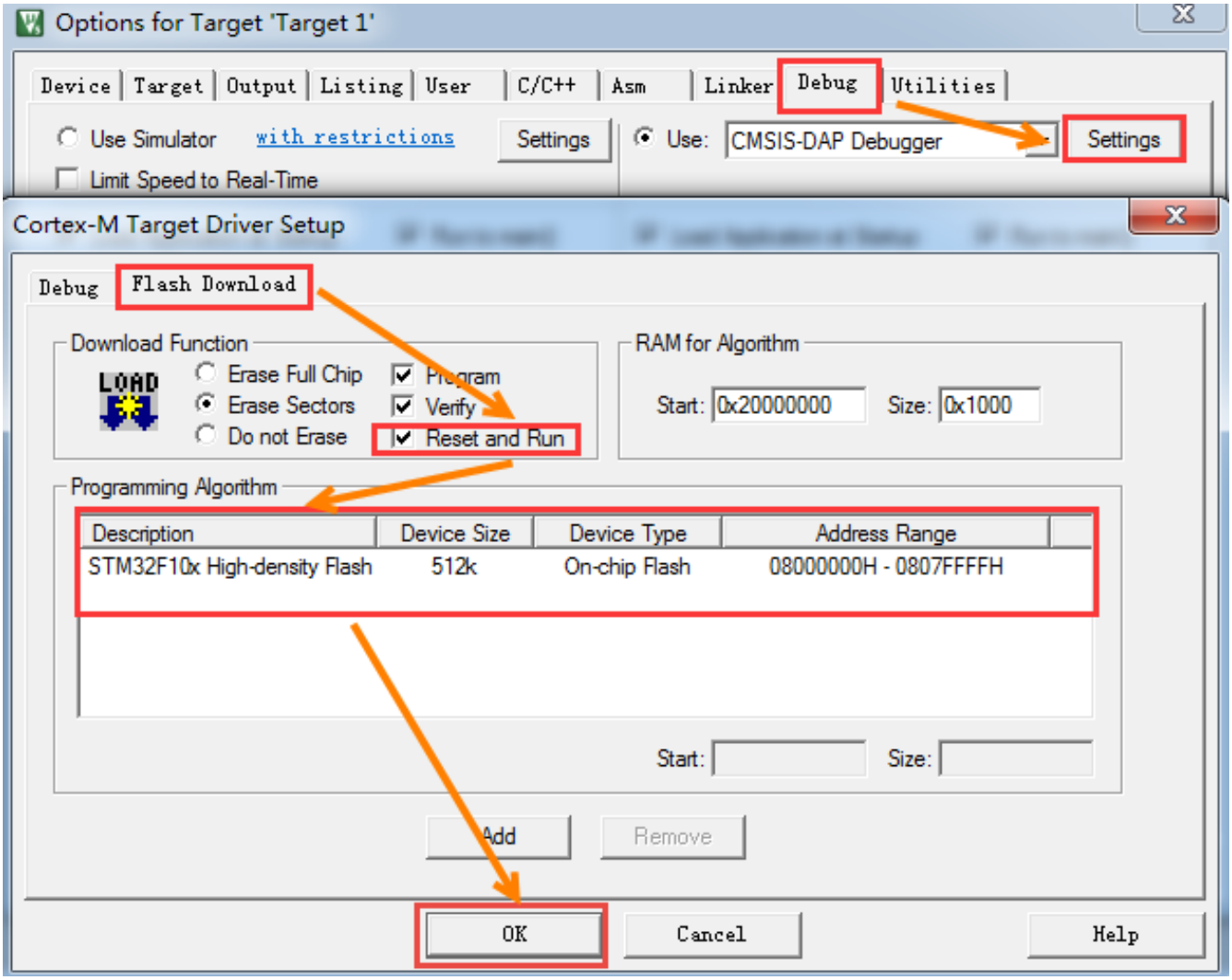
Utilities 选项配置



Debug Settings 选项配置

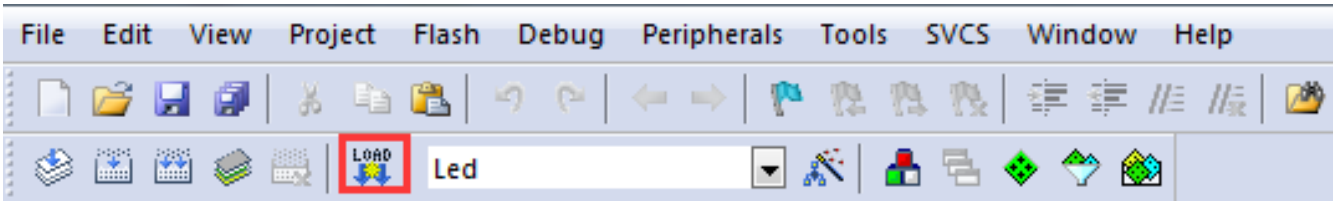


选择目标板，具体选择多大的 FLASH 要根据板子上的芯片型号决定。这里 F103-MINI 选 256K，指南者/霸道/拂晓都选 512K。这里面有个小技巧就是把 Reset and Run 也勾选上，这样程序下载完之后就会自动运行，否则需要手动复位。擦除的 FLASH 大小选择 Sectors 即可，不要选择 Full Chip，不然下载会比较慢。



6.2 下载程序

如果前面步骤都成功了，接下来就可以把编译好的程序下载到开发板上运行。下载程序不需要其他额外的软件，直接点击 KEIL 中的 LOAD 按钮即可。



程序下载后，Build Output 选项卡如果打印出 Application running…则表示程序下载成功。如果没有出现实验现象，按复位键试试。当然，这只是一个工程模版，我们还没写程序，开发板不会有任何现象。

至此，一个新的工程模版新建完毕。

第 7 章　使用寄存器点亮 LED 灯

本章参考资料：《STM32F10X-中文参考手册》GPIO 和 RCC 章节。

学习本章时，配合《STM32F10X-中文参考手册》“通用 I/O(GPIO)”章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。关于建立工程时使用 KEIL5 的基本操作，请参考前面的章节。

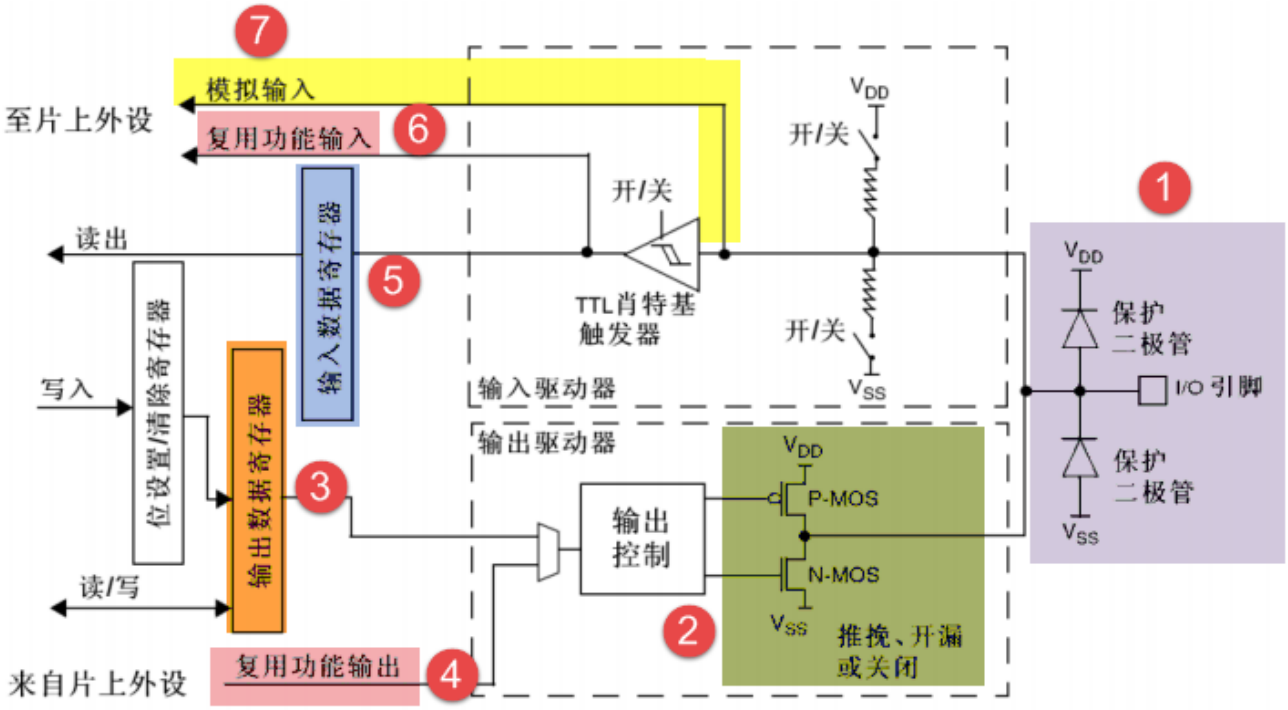
7.1 GPIO 简介

GPIO 是通用输入输出端口的简称，简单来说就是 STM32 可控制的引脚，STM32 芯片的 GPIO 引脚与外部设备连接起来，从而实现与外部通讯、控制以及数据采集的功能。STM32 芯片的 GPIO 被分成很多组，每组有 16 个引脚（也有可能少于 16 个），如型号为 STM32F103VET6 型号的芯片有 GPIOA、GPIOB、GPIOC 至 GPIOE 共 5 组 GPIO，该芯片一共 100 个引脚，其中 GPIO 引脚就占了一大部分，所有的 GPIO 引脚都有基本的输入输出功能。

最基本的输出功能是由 STM32 控制引脚输出高、低电平，实现开关控制，如把 GPIO 引脚接入到 LED 灯，那就可以控制 LED 灯的亮灭，引脚接入到继电器或三极管，那就可以通过继电器或三极管控制外部大功率电路的通断。

最基本的输入功能是检测外部输入电平，如把 GPIO 引脚连接到按键，通过电平高低区分按键是否被按下。

7.2 GPIO 框图剖析



通过 GPIO 硬件结构框图，就可以从整体上深入了解 GPIO 外设及它的各种应用模式。该图从最右端看起，最右端就是代表 STM32 芯片引出的 GPIO 引脚，其余部件都位于芯片内部。

7.2.1 基本结构分析

下面我们按图中的编号对 GPIO 端口的结构部件进行说明。

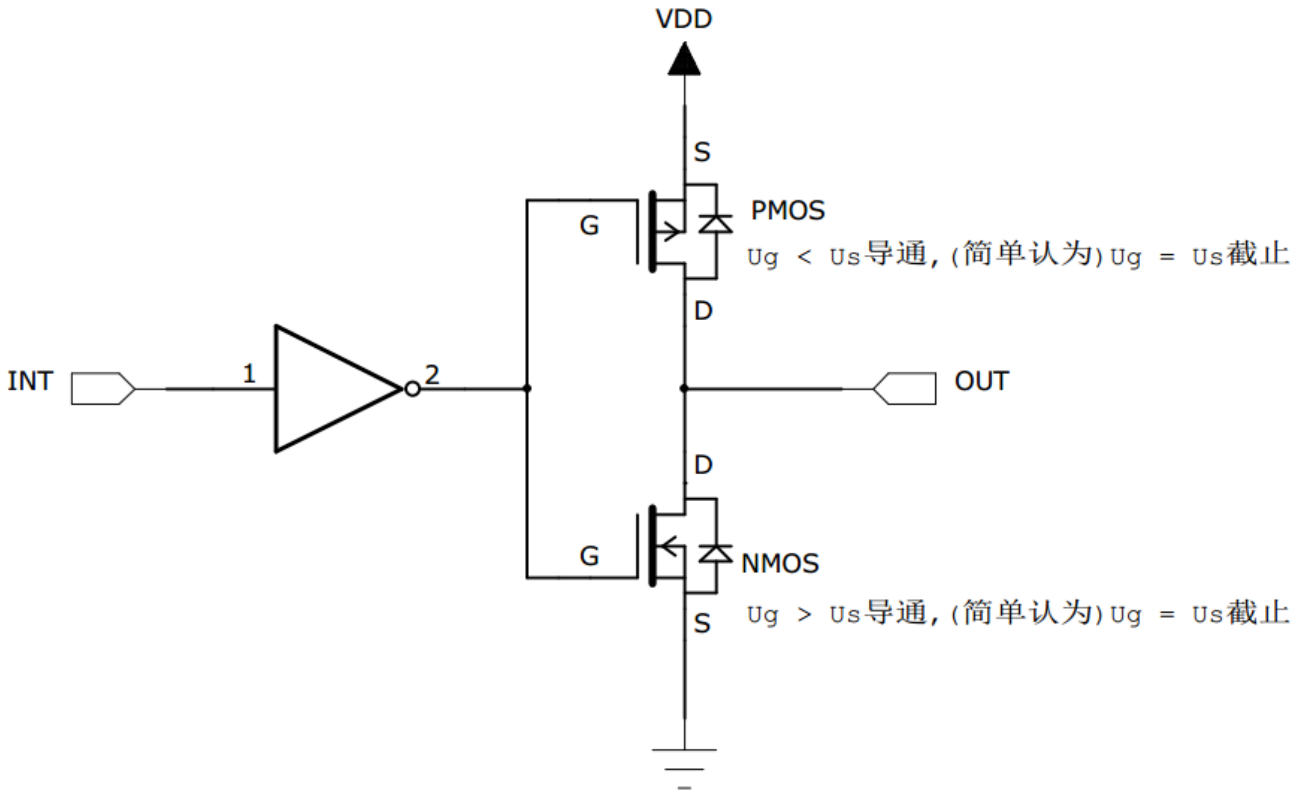
7.2.1.1 保护二极管及上、下拉电阻

引脚的两个保护二级管可以防止引脚外部过高或过低的电压输入，当引脚电压高于 V_{DD} 时，上方的二极管导通，当引脚电压低于 V_{SS} 时，下方的二极管导通，防止不正常电压引入芯片导致芯片烧毁。尽管有这样的保护，并不意味着 STM32 的引脚能直接外接大功率驱动器件，如直接驱动电机，强制驱动要么电机不转，要么导致芯片烧坏，必须要加大功率及隔离电路驱动。

7.2.1.2 P-MOS 管和 N-MOS 管

GPIO 引脚线路经过两个保护二极管后，向上流向“输入模式”结构，向下流向“输出模式”结构。先看输出模式部分，线路经过一个由 P-MOS 和 N-MOS 管组成的单元电路。这个结构使 GPIO 具有了“推挽输出”和“开漏输出”两种模式。

所谓的推挽输出模式，是根据这两个 MOS 管的工作方式来命名的。在该结构中输入高电平时，经过反向后，上方的 P-MOS 导通，下方的 N-MOS 关闭，对外输出高电平；而在该结构中输入低电平时，经过反向后，N-MOS 管导通，P-MOS 关闭，对外输出低电平。当引脚高低电平切换时，两个管子轮流导通，P 管负责灌电流，N 管负责拉电流，使其负载能力和开关速度都比普通的方式有很大的提高。推挽输出的低电平为 0 伏，高电平为 3.3 伏，具体参考图 *7_2*，它是推挽输出模式时的等效电路。

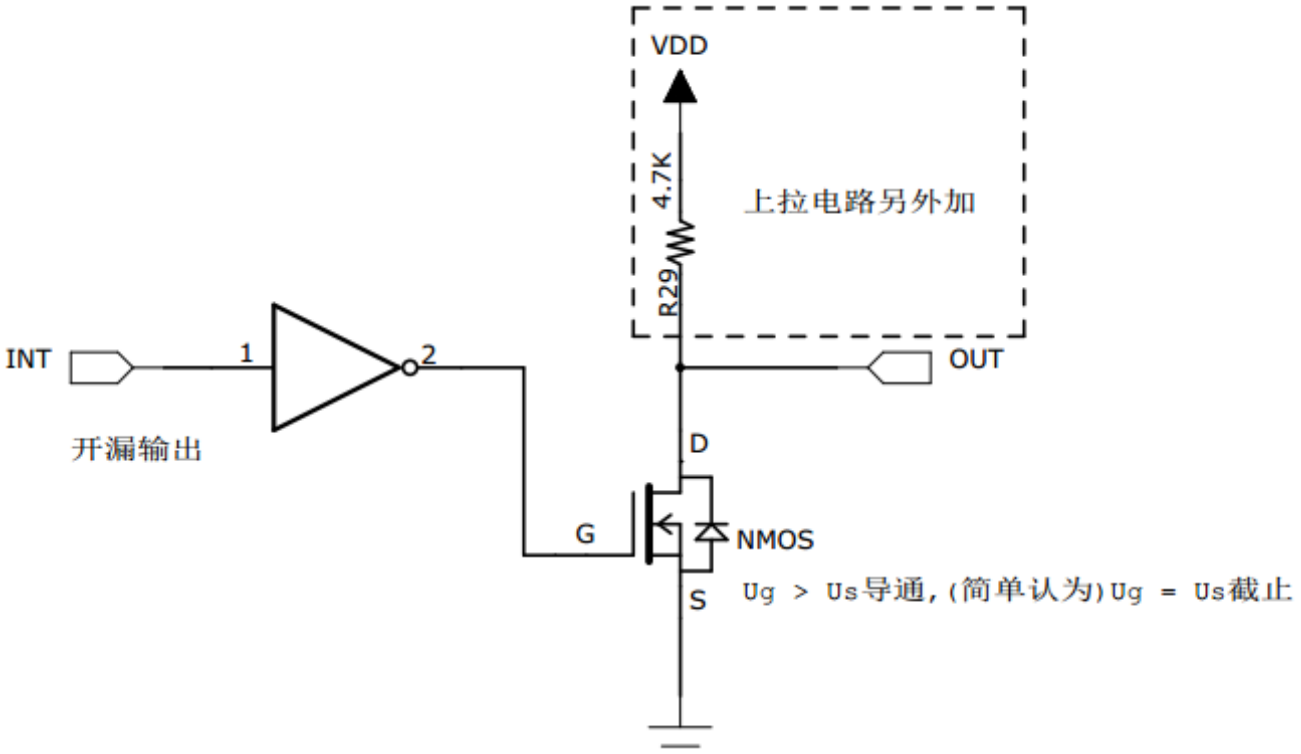


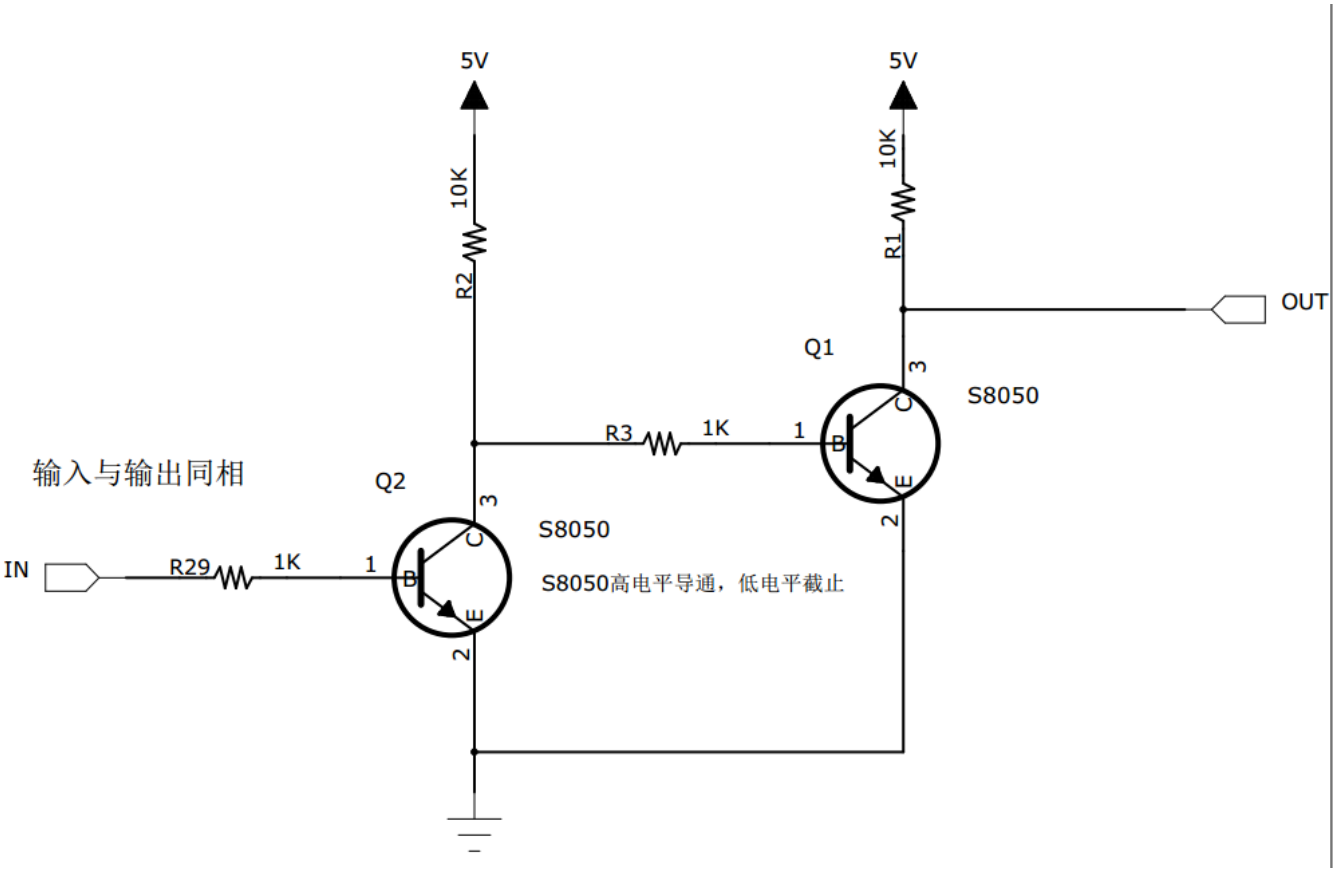
而在开漏输出模式时，上方的 P-MOS 管完全不工作。如果我们控制输出为 0，低电平，则 P-MOS 管关闭，N-MOS 管导通，使输出接地，若控制输出为 1 (它无法直接输出高电平) 时，则 P-MOS 管和 N-MOS 管都关闭，所以引脚既不输出高电平，也不输出低电平，为高阻态。为正常使用时

必须外部接上拉电阻，参考图 *7_3* 中等效电路。它具有“线与”特性，也就是说，若有很多个开漏模式引脚连接到一起时，只有当所有引脚都输出高阻态，才由上拉电阻提供高电平，此高电平的电压为外部上拉电阻所接的电源的电压。若其中一个引脚为低电平，那线路就相当于短路接地，使得整条线路都为低电平，0 伏。

推挽输出模式一般应用在输出电平为 0 和 3.3 伏而且需要高速切换开关状态的场合。在 STM32 的应用中，除了必须用开漏模式的场合，我们都习惯使用推挽输出模式。

开漏输出一般应用在 I2C、SMBUS 通讯等需要“线与”功能的总线电路中。除此之外，还用在电平不匹配的场合，如需要输出 5 伏的高电平，就可以在外部接一个上拉电阻，上拉电源为 5 伏，并且把 GPIO 设置为开漏模式，当输出高阻态时，由上拉电阻和电源向外输出 5 伏的电平，具体见图 *7_4*。





7.2.1.3 输出数据寄存器

前面提到的双 MOS 管结构电路的输入信号，是由 GPIO “输出数据寄存器 GPIOx_ODR” 提供的，因此我们通过修改输出数据寄存器的值就可以修改 GPIO 引脚的输出电平。而 “置位/复位寄存器 GPIOx_BSRR” 可以通过修改输出数据寄存器的值从而影响电路的输出。

7.2.1.4 复用功能输出

“复用功能输出” 中的 “复用” 是指 STM32 的其它片上外设对 GPIO 引脚进行控制，此时 GPIO 引脚用作该外设功能的一部分，算是第二用途。从其它外设引出来的 “复用功能输出信号” 与 GPIO 本身的数据据寄存器都连接到双 MOS 管结构的输入中，通过图中的梯形结构作为开关切换选择。

例如我们使用 USART 串口通讯时，需要用到某个 GPIO 引脚作为通讯发送引脚，这个时候就可

以把该 GPIO 引脚配置成 USART 串口复用功能，由串口外设控制该引脚，发送数据。

列表 1: 代码 7_0_1

```
// GPIOB 16 个 IO 全部输出 0XFF
GPIOB->ODR = 0XFF;
```

7.2.1.5 输入数据寄存器

看 GPIO 结构框图的上半部分，GPIO 引脚经过内部的上、下拉电阻，可以配置成上/下拉输入，然后再连接到施密特触发器，信号经过触发器后，模拟信号转化为 0、1 的数字信号，然后存储在“输入数据寄存器 GPIOx_IDR”中，通过读取该寄存器就可以了解 GPIO 引脚的电平状态。

列表 2: 代码 7_0_2

```
// 读取 GPIOB 端口的 16 位数据值
uint16_t temp;
temp = GPIOB->IDR;
```

7.2.1.6 复用功能输入

与“复用功能输出”模式类似，在“复用功能输入模式”时，GPIO 引脚的信号传输到 STM32 其它片上外设，由该外设读取引脚状态。

同样，如我们使用 USART 串口通讯时，需要用到某个 GPIO 引脚作为通讯接收引脚，这个时候就可以把该 GPIO 引脚配置成 USART 串口复用功能，使 USART 可以通过该通讯引脚的接收远端数据。

7.2.1.7 模拟输入输出

当 GPIO 引脚用于 ADC 采集电压的输入通道时，用作“模拟输入”功能，此时信号是不经过施密特触发器的，因为经过施密特触发器后信号只有 0、1 两种状态，所以 ADC 外设要采集到原始的模拟信号，信号源输入必须在施密特触发器之前。类似地，当 GPIO 引脚用于 DAC 作为模拟电压输出通道时，此时作为“模拟输出”功能，DAC 的模拟信号输出就不经过双 MOS 管结构，模拟信号直接输出到引脚。

7.2.2 GPIO 工作模式

总结一下，由 GPIO 的结构决定了 GPIO 可以配置成以下模式：

列表 3: 代码 7-1 GPIO 8 种工作模式

```
typedef enum
{
```

(下页继续)

(续上页)

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

在固件库中，GPIO 总共有 8 种细分的工作模式，稍加整理可以大致归类为以下三类：

7.2.2.1 输入模式 (模拟/浮空/上拉/下拉)

在输入模式时，施密特触发器打开，输出被禁止，可通过输入数据寄存器 GPIOx_IDR 读取 I/O 状态。其中输入模式，可设置为上拉、下拉、浮空和模拟输入四种。上拉和下拉输入很好理解，默认的电平由上拉或者下拉决定。浮空输入的电平是不确定的，完全由外部的输入决定，一般接按键的时候用的是这个模式。模拟输入则用于 ADC 采集。

7.2.2.2 输出模式 (推挽/开漏)

在输出模式中，推挽模式时双 MOS 管以轮流方式工作，输出数据寄存器 GPIOx_ODR 可控制 I/O 输出高低电平。开漏模式时，只有 N-MOS 管工作，输出数据寄存器可控制 I/O 输出高阻态或低电平。输出速度可配置，有 2MHz/10MHz/50MHz 的选项。此处的输出速度即 I/O 支持的高低电平状态最高切换频率，支持的频率越高，功耗越大，如果功耗要求不严格，把速度设置成最大即可。

在输出模式时施密特触发器是打开的，即输入可用，通过输入数据寄存器 GPIOx_IDR 可读取 I/O 的实际状态。

7.2.2.3 复用功能 (推挽/开漏)

复用功能模式中，输出使能，输出速度可配置，可工作在开漏及推挽模式，但是输出信号源于其它外设，输出数据寄存器 GPIOx_ODR 无效；输入可用，通过输入数据寄存器可获取 I/O 实际状态，但一般直接用外设的寄存器来获取该数据信号。

通过对 GPIO 寄存器写入不同的参数，就可以改变 GPIO 的工作模式，再强调一下，要了解具体寄存器时一定要查阅《STM32F10X-中文参考手册》中对应外设的寄存器说明。在 GPIO 外设中，控制端口高低控制寄存器 CRH 和 CRL 可以配置每个 GPIO 的工作模式和工作的速度，每 4 个位控制一个 IO，CRH 控制端口的高八位，CRL 控制端口的低 8 位，具体的看 CRH 和 CRL 的寄存器描述。

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

CRL控制着端口的低8位IO

偏移地址：0x00

复位值：0x4444 4444

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

4bit分成一组，控制一个IO

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

8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)

CRH控制着端口的高8位IO

偏移地址：0x04

复位值：0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[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
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

4bit分成一组，控制一个IO

7.3 实验：使用寄存器点亮 LED 灯

本小节中，我们以实例讲解如何通过控制寄存器来点亮 LED 灯。此处侧重于讲解原理，请直接用 KEIL5 软件打开我们提供的实验例程配合阅读，先了解原理，学习完本小节后，再尝试自己建立一个同样的工程。本节配套例程名称为“GPIO 输出—使用寄存器点亮 LED 灯”，在工程目

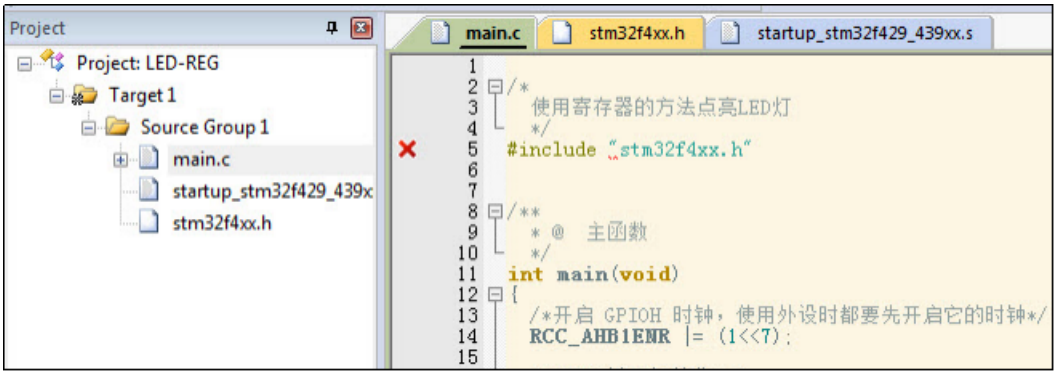
录下找到后缀为“.uvprojx”的文件，用 KEIL5 打开即可。

自己尝试新建工程时，请对照查阅《新建工程—寄存器版》章节。若没有安装 KEIL5 软件，请参考《如何安装 KEIL5》章节。

打开该工程，见图 7_7，可看到一共有三个文件：

- startup_stm32f10x_hd.s
- stm32f10x.h
- main.c

下面我们对这三个文件进行讲解。



7.3.1 硬件连接

我们的 F103-指南者和 F103-霸道开发板中 STM32 芯片与 LED 灯的连接见图 7_8，这是一个 RGB 灯，里面由红绿蓝（Red, Green, Blue）三个小灯构成，使用 PWM 控制时可以混合成 256*256*256 种不同的颜色。

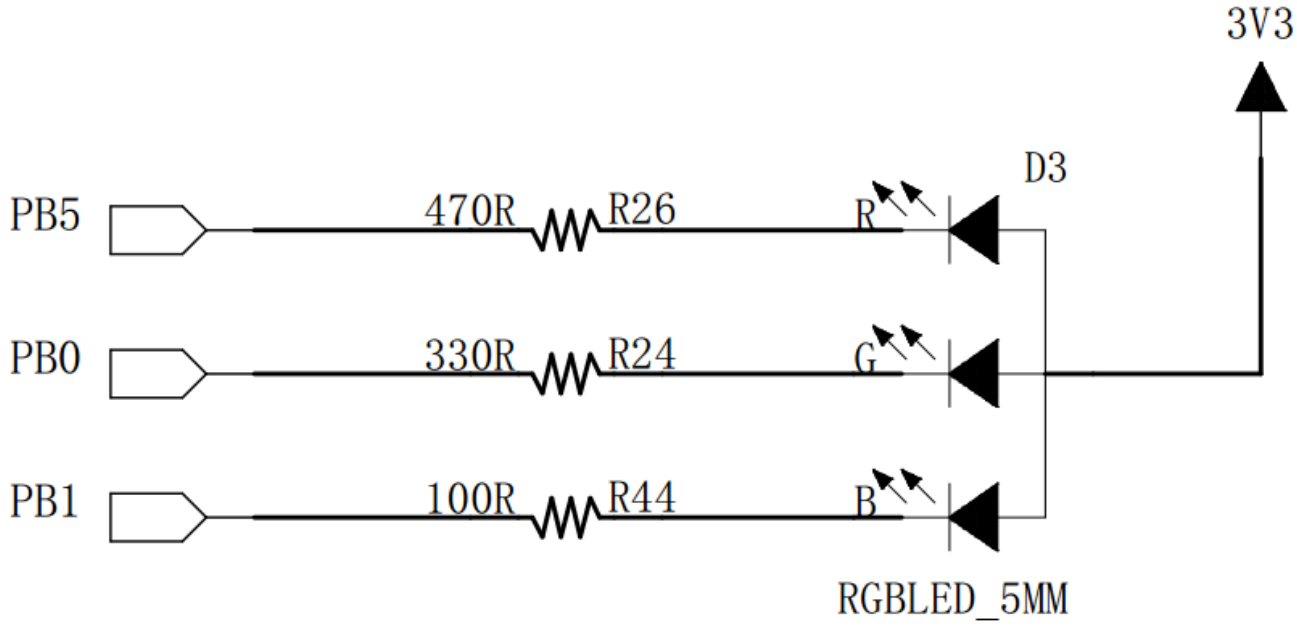


图 7-8 F103-指南者和 F103-霸道开发板的 LED 硬件原理图

而我们的 F103-MINI 开发板和 F103-拂晓电机开发板接的是单色的 LED 灯，其中 STM32 芯片与 LED 灯的连接分别如图 *7_9* 和图 *7_10* 所示。

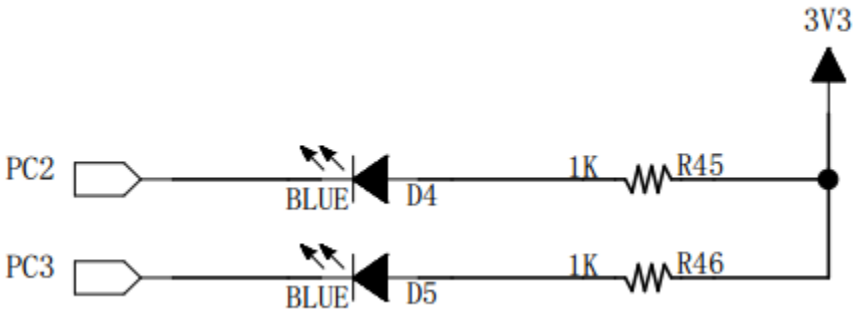


图 7-9 F103-MINI 开发板的 LED 硬件原理图

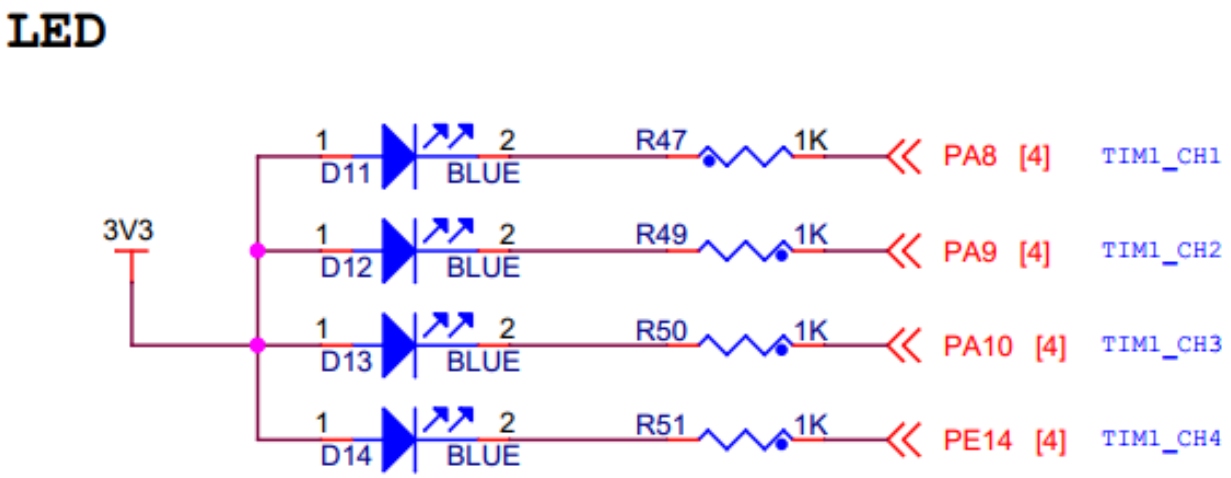


图 7-10 F103-拂晓电机开发板的 LED 硬件原理图

以上所有的这些 LED 灯的阴极都是连接到 STM32 的 GPIO 引脚，阳极接到 3.3V 电源，同时每个灯都分别接了一个限流电阻。只要我们控制 GPIO 引脚的电平输出状态，即可控制 LED 灯的亮灭。若您使用的实验板 LED 灯的连接方式或引脚不一样，只需根据我们的工程修改引脚即可，程序的控制原理相同。

我们的目标是把 GPIO 的引脚设置成推挽输出模式并且默认下拉，输出低电平，这样就能让 LED 灯亮起来了。

7.3.2 启动文件

启动文件在这里只是简要的介绍下，关于这个文件的详解请参考后面的《启动文件详解》章节。

名为“startup_stm32f10x_hd.s”的文件，它里边使用汇编语言写好了基本程序，当 STM32 芯片上电启动的时候，首先会执行这里的汇编程序，从而建立起 C 语言的运行环境，所以我们把这个文件称为启动文件。该文件使用的汇编指令是 Cortex-M3 内核支持的指令，可参考《Cortex-M3 权威指南》中指令集章节。

startup_stm32f10x_hd.s 文件由官方提供，一般有需要也是在官方的基础上修改，不会自己完全重写。该文件从 ST 固件库里面找到，找到该文件后把启动文件添加到工程里面即可。不同型号的芯片以及不同编译环境下使用的汇编文件是不一样的，但功能相同。

对于启动文件这部分我们主要总结它的功能，不详解讲解里面的代码，其功能如下：

- 初始化堆栈指针 SP;
- 初始化程序计数器指针 PC;
- 设置堆、栈的大小;
- 初始化中断向量表;
- 配置外部 SRAM 作为数据存储器（这个由用户配置，一般的开发板可没有外部 SRAM）；
- 调用 SystemIni() 函数配置 STM32 的系统时钟。
- 设置 C 库的分支入口“__main”（最终用来调用 main 函数）；

先去除繁枝细节，挑重点的讲，主要理解最后两点，在启动文件中有一段复位后立即执行的程序，代码见*代码清单 7_2*。在实际工程中阅读时，可使用编辑器的搜索 (Ctrl+F) 功能查找这段代码在文件中的位置，搜索 Reset_Handler 即可找到。

列表 4: 代码清单 7-2 复位后执行的程序

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

开头的是程序注释，在汇编里面注释用的是“;”，相当于 C 语言的“//”注释符

第二行是定义了一个子程序：Reset_Handler。PROC 是子程序定义伪指令。这里就相当于 C 语言里定义了一个函数，函数名为 Reset_Handler。

第三行 EXPORT 表示 Reset_Handler 这个子程序可供其他模块调用。相当于 C 语言的函数声明。

关键字 [WEAK] 表示弱定义，如果编译器发现在别处定义了同名的函数，则在链接时用别处的地址进行链接，如果其它地方没有定义，编译器也不报错，以此处地址进行链接。

第四行和第五行 IMPORT 说明 SystemInit 和 __main 这两个标号在其他文件，在链接的时候需要到其他文件去寻找。相当于 C 语言中，从其它文件引入函数声明。以便下面对外部函数进行调用。

SystemInit 需要由我们自己实现，即我们要编写一个具有该名称的函数，用来初始化 STM32 芯片的时钟，一般包括初始化 AHB、APB 等各总线的时钟，需要经过一系列的配置 STM32 才能达到稳定运行的状态。其实这个函数在固件库里面有提供，官方已经为我们写好。

__main 其实不是我们定义的 (不要与 C 语言中的 main 函数混淆)，这是一个 C 库函数，当编译器编译时，只要遇到这个标号就会定义这个函数，该函数的主要功能是：负责初始化栈、堆，配置系统环境，并在函数的最后调用用户编写的 main 函数，从此来到 C 的世界。

第六行把 SystemInit 的地址加载到寄存器 R0。

第七行程序跳转到 R0 中的地址执行程序，即执行 SystemInit 函数的内容。

第八行把 __main 的地址加载到寄存器 R0。

第九行程序跳转到 R0 中的地址执行程序，即执行 __main 函数，执行完毕之后就去到我们熟知的 C 世界，进入 main 函数。

第十行表示子程序的结束。

总之，看完这段代码后，了解到如下内容即可：我们需要在外部定义一个 SystemInit 函数设置 STM32 的时钟；STM32 上电后，会执行 SystemInit 函数，最后执行我们 C 语言中的 main 函数。

7.3.3 stm32f10x.h 文件

看完启动文件，那我们立即写 SystemInit 和 main 函数吧？别着急，定义好了 SystemInit 函数和 main 我们又能写什么内容？连接 LED 灯的 GPIO 引脚，是要通过读写寄存器来控制的，就这样空着手，如何控制寄存器。我们知道寄存器就是给一个已经分配好地址的特殊的内存空间取的一个别名，这个特殊的内存空间可以通过指针来操作。在编程之前我们要先实现寄存器映射，有关寄存器映射的代码都统一写在 stm32f10x.h 文件中，见*代码清单 7_3a*、*代码清单 7_3b*、*代码清单 7_3c*。

下面是F103-MINI开发板、F103-指南者和F103-霸道开发板、还有F103-拂晓电机开发板中STM32芯片的外设地址定义。

列表 5: 代码清单 7-3a F103-MINI 的外设地址定义

```
/* 片上外设基地址  */
#define PERIPH_BASE           ((unsigned int)0x40000000)

/* 总线基地址，GPIO 都挂载到 APB2 上 */
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
/* AHB 总线基地址 */
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)

/*GPIOB 外设基地址 */
#define GPIOC_BASE            (APB2PERIPH_BASE + 0x1000)

/* GPIOB 寄存器地址，强制转换成指针 */
#define GPIOC_CRL             *(unsigned int*)(GPIOC_BASE+0x00)
#define GPIOC_CRH             *(unsigned int*)(GPIOC_BASE+0x04)
#define GPIOC_IDR             *(unsigned int*)(GPIOC_BASE+0x08)
#define GPIOC_ODR             *(unsigned int*)(GPIOC_BASE+0x0C)
#define GPIOC_BSRR            *(unsigned int*)(GPIOC_BASE+0x10)
#define GPIOC_BRR             *(unsigned int*)(GPIOC_BASE+0x14)
#define GPIOC_LCKR            *(unsigned int*)(GPIOC_BASE+0x18)

/*RCC 外设基地址 */
#define RCC_BASE              (AHBPERIPH_BASE + 0x1000)
/*RCC 的 AHB1 时钟使能寄存器地址，强制转换成指针 */
#define RCC_APB2ENR           *(unsigned int*)(RCC_BASE+0x18)
```

列表 6: 代码清单 7-3b F103-指南者和 F103-霸道的外设地址定义

```
/* 片上外设基地址  */
#define PERIPH_BASE          ((unsigned int)0x40000000)

/* 总线基地址，GPIO 都挂载到 APB2 上 */
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)

/*GPIOB 外设基地址 */
#define GPIOB_BASE           (APB2PERIPH_BASE + 0x0C00)

/* GPIOB 寄存器地址，强制转换成指针 */
#define GPIOB_CRL            *(unsigned int*)(GPIOB_BASE+0x00)
#define GPIOB_CRH            *(unsigned int*)(GPIOB_BASE+0x04)
#define GPIOB_IDR            *(unsigned int*)(GPIOB_BASE+0x08)
#define GPIOB_ODR            *(unsigned int*)(GPIOB_BASE+0x0C)
#define GPIOB_BSRR           *(unsigned int*)(GPIOB_BASE+0x10)
#define GPIOB_BRR            *(unsigned int*)(GPIOB_BASE+0x14)
#define GPIOB_LCKR           *(unsigned int*)(GPIOB_BASE+0x18)

/*RCC 外设基地址 */
#define RCC_BASE             (AHBPERIPH_BASE + 0x1000)
/*RCC 的 AHB1 时钟使能寄存器地址，强制转换成指针 */
#define RCC_APB2ENR          *(unsigned int*)(RCC_BASE+0x18)
```

列表 7: 代码清单 7-3c F103-拂晓的外设地址定义

```
/* 片上外设基地址  */
#define PERIPH_BASE          ((unsigned int)0x40000000)

/* 总线基地址，GPIO 都挂载到 APB2 上 */
```

(下页继续)

(续上页)

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

/*GPIOB 外设基地址 */
#define GPIOA_BASE           (APB2PERIPH_BASE + 0x0800)

/* GPIOB 寄存器地址, 强制转换成指针 */
#define GPIOA_CRL            *(unsigned int*)(GPIOA_BASE+0x00)
#define GPIOA_CRH            *(unsigned int*)(GPIOA_BASE+0x04)
#define GPIOA_IDR            *(unsigned int*)(GPIOA_BASE+0x08)
#define GPIOA_ODR            *(unsigned int*)(GPIOA_BASE+0x0C)
#define GPIOA_BSRR           *(unsigned int*)(GPIOA_BASE+0x10)
#define GPIOA_BRR            *(unsigned int*)(GPIOA_BASE+0x14)
#define GPIOA_LCKR           *(unsigned int*)(GPIOA_BASE+0x18)

/*RCC 外设基地址 */
#define RCC_BASE             (AHBPERIPH_BASE + 0x1000)
/*RCC 的 AHB1 时钟使能寄存器地址, 强制转换成指针 */
#define RCC_APB2ENR          *(unsigned int*)(RCC_BASE+0x18)
```

GPIO 外设的地址跟前面章节讲解的相同，不过此处把寄存器的地址值都直接强制转换成了指针，方便使用。代码的最后两段是 RCC 外设寄存器的地址定义，RCC 外设是用来设置时钟的，以后我们会详细分析，本实验中只要了解到使用 GPIO 外设必须开启它的时钟即可。

7.3.4 main 文件

现在就可以开始编写程序，在 main 文件中先编写一个 main 函数，里面什么都没有，暂时为空。

```
int main (void)
{
}
```

此时直接编译的话，会出现如下错误：

“Error: L6218E: Undefined symbol SystemInit (referred from startup_stm32f10x.o)”

错误提示 SystemInit 没有定义。从分析启动文件时我们知道，Reset_Handler 调用了该函数用来初始化 SMT32 系统时钟，为了简单起见，我们在 main 文件里面定义一个 SystemInit 空函数，什么也不做，为的是骗过编译器，把这个错误去掉。关于配置系统时钟我们在后面再写。当我们不配置系统时钟时，STM32 会把 HSI 当作系统时钟，HSI=8M，由芯片内部的振荡器提供。我们在 main 中添加如下函数：

```
// 函数为空，目的是为了骗过编译器不报错
void SystemInit(void)
{
}
```

这时再编译就没有错了，完美解决。还有一个方法就是在启动文件中把有关 SystemInit 的代码注释掉也可以，见*代码清单 7_4*。

列表 8: 代码清单 7-4 注释掉启动文件中调用 SystemInit 的代码

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

接下来在 main 函数中添加代码，实现我们的点灯之旅。

7.3.4.1 GPIO 模式

首先要根据自己所使用的开发板，来确定有哪些引脚是连接到 LED 灯上的，然后配置这些引脚。每一块开发板上不止一个 LED 小灯（全彩 LED 灯由 3 个 IO 引脚控制其颜色），我们目前仅配置其中一个小灯对应的引脚即可。

- 对于 F103-MINI 开发板，配置的是 PC2
- 对于 F103-指南者和 F103-霸道开发板，配置的是 PB0
- 对于 F103-拂晓电机开发板，配置的是 PA8

我们把以上连接到 LED 灯的 GPIO 引脚配置成通用推挽输出，输出的速度为 10M。即：对于 PC2 和 PB0 引脚，我们要配置 GPIO 的端口配置低寄存器：CRL，见图 *7_11*。CRL 中包含 0-7 号引脚，每个引脚占用 4 个寄存器位。MODE 位用来配置输出的速度，CNF 位用来配置各种输入输出模式。

而对于 PA8 引脚，由于 CRL 仅能配置 GPIO 端口的 0-7 号引脚，所以我们要配置的是端口配置高寄存器：CRH。类似地，CRH 中包含 8-15 号引脚，每个引脚也占用 4 个寄存器位。MODE 位用来

配置输出的速度，CNF 位用来配置各种输入输出模式。这样，每个 GPIO 的端口（GPIOA, GPIOB, GPIOC…等）都分别对应着一个 CRL 和一个 CRH 寄存器，由这两个寄存器配置每个 GPIO 端口的共 16 个引脚的工作模式。

配置的代码具体见代码清单 7_5。

列表 9: 代码清单 7-5 配置输出模式

```
/* F103-MINI 开发板 */
 // 清空控制 PC2 的端口位
 GPIOC_CRL &= ~( 0x0F<< (4*2));
 // 配置 PC2 为通用推挽输出，速度为 10M
 GPIOC_CRL |= (1<<4*2);

 /* F103-指南者和 F103-霸道开发板 */
 // 清空控制 PB0 的端口位
 GPIOB_CRL &= ~( 0x0F<< (4*0));
 // 配置 PB0 为通用推挽输出，速度为 10M
 GPIOB_CRL |= (1<<4*0);

 /* F103-拂晓电机开发板 */
 // 清空控制 PA8 的端口位
 GPIOA_CRH &= ~( 0x0F<< (4*0));
 // 配置 PA8 为通用推挽输出，速度为 10M
 GPIOA_CRH |= (1<<4*0);
```

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址：0x00

复位值：0x4444 4444

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

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

0 0 0 1

CRL低四位配置为：0001，即可设置GPIO端口0为输出模式，速度为10M

以配置 PB0 为例，在代码中，我们先把控制 PB0 的端口位清 0，然后再向它赋值“0001 b”，从而使 GPIOB0 引脚设置成输出模式，速度为 10M。

代码中使用了“&= ~”、“|=”这种操作方法是为了避免影响到寄存器中的其它位，因为寄存器不能按位读写，假如我们直接给 CRL 寄存器赋值：

```
GPIOB_CRL = 0x0000001;
```

这时 CRL 的的低 4 位被设置成“0001”输出模式，但其它 GPIO 引脚就有意见了，因为其它引脚

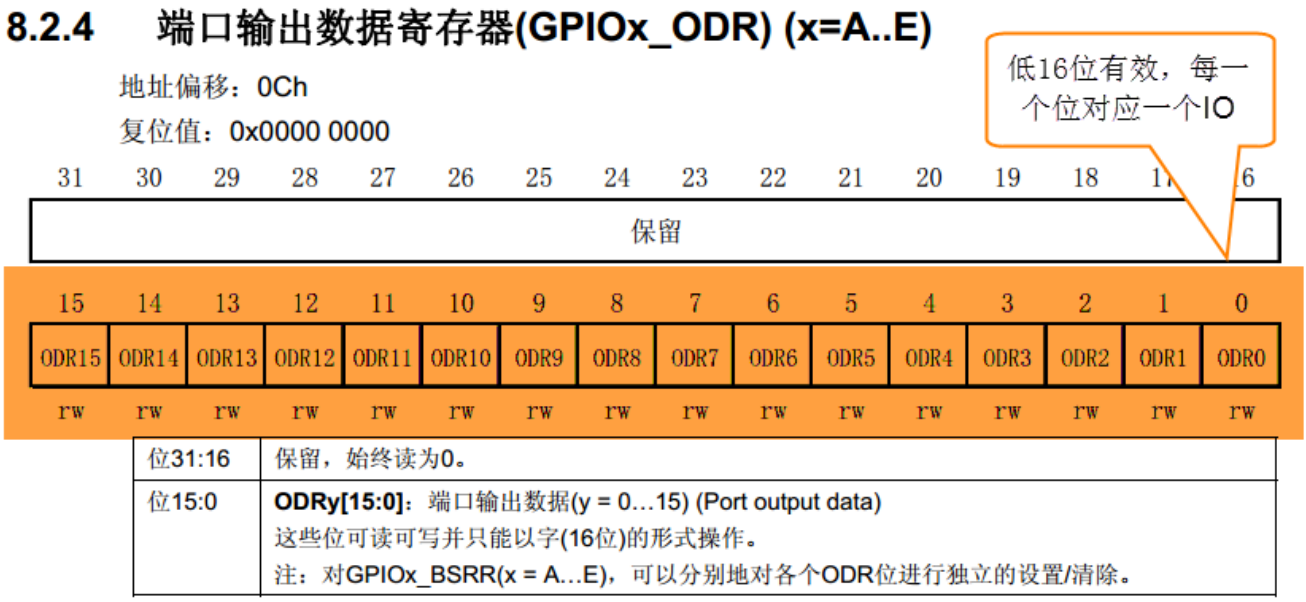
的 MODER 位都已被设置成输入模式。

7.3.4.2 控制引脚输出电平

在输出模式时，对端口位设置/清除寄存器 BSRR 寄存器、端口位清除寄存器 BRR 和 ODR 寄存器写入参数即可控制引脚的电平状态，其中操作 BSRR 和 BRR 最终影响的都是 ODR 寄存器，然后再通过 ODR 寄存器的输出来控制 GPIO。为了一步到位，我们在这里直接操作 ODR 寄存器来控制 GPIO 的电平。具体见*代码清单 7_6*。

列表 10: 代码清单 7-6 控制引脚输出电平

```
// F103-MINI 开发板  PC2 输出低电平
GPIOC_ODR &= ~(1<<2);

// F103-指南者和 F103-霸道开发板  PB0 输出低电平
GPIOB_ODR &= ~(1<<0);

// F103-拂晓电机开发板  PA8 输出低电平
GPIOA_ODR &= ~(1<<8);
```



7.3.4.3 开启外设时钟

设置完 GPIO 的引脚，控制电平输出，以为现在总算可以点亮 LED 了吧，其实还差最后一步。由于 STM32 的外设很多，为了降低功耗，每个外设都对应着一个时钟，在芯片刚上电的时候这些时钟都是被关闭的，如果想要外设工作，必须把相应的时钟打开。

STM32 的所有外设的时钟由一个专门的外设来管理，叫 RCC（reset and clockcontrol），RCC 在《STM32F10X-中文参考手册》的第六章。关于 RCC 外设中的时钟部分，我们在后面的章节《RCC—使用 HSE/HSI 配置》中有详细的讲解，这里我们暂时先了解下。

所有的 GPIO 都挂载到 APB2 总线上，具体的时钟由 APB2 外设时钟使能寄存器 (RCC_ APB2ENR) 来控制，具体见代码清单 7_7。

列表 11: 代码清单 7-7 开启端口时钟

```
// F103-MINI 开发板  开启 GPIOC 端口 时钟
RCC_APB2ENR |= (1<<4);

// F103-指南者和 F103-霸道开发板   开启 GPIOB 端口 时钟
RCC_APB2ENR |= (1<<3);

// F103-拂晓电机开发板 开启 GPIOA 端口 时钟
RCC_APB2ENR |= (1<<2);
```

6.3.7 APB2 外设时钟使能寄存器(RCC_APB2ENR)

偏移地址：0x18

复位值：0x0000 0000

访问：字，半字和字节访问

通常无访问等待周期。但在APB2总线上的外设被访问时，将插入等待状态，直至APB2的外设访问结束。

注：当外设时钟没有启用时，软件不能读出外设寄存器的数值，返回的数值始终是0x0。

控制GPIOB端口时钟位

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC3 EN	USART1 EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	保留	AFIO EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

图 7-11 APB2 外设时钟使能寄存器

7.3.4.4 水到渠成

开启时钟，配置引脚模式，控制电平，经过这三步，我们总算可以控制一个 LED 了。现在我们完整组织下用 STM32 控制一个 LED 的代码，见*代码清单 7_8a*、*代码清单 7_8b*、*代码清单 7_8c*。

列表 12: 代码清单 7-8a F103-MINI 开发板对应的 main 函数代码

```
int main(void)
{
    // 开启 GPIOC 端口 时钟
    RCC_APB2ENR |= (1<<4);

    // 清空控制 PC2 的端口位
    GPIOC_CRL &= ~( 0x0F<< (4*2));
    // 配置 PC2 为通用推挽输出，速度为 10M
    GPIOC_CRL |= (1<<4*2);

    // PC2 输出低电平
    GPIOC_ODR &= ~(1<<2);

    while (1);

}
```

列表 13: 代码清单 7-8b F103-指南者和 F103-霸道开发板对应的 main 函数代码

```
int main(void)
{
    // 开启 GPIOB 端口时钟
    RCC_APB2ENR |= (1<<3);
```

(下页继续)

(续上页)

```
    //清空控制 PB0 的端口位
    GPIOB_CRL &= ~( 0x0F<< (4*0));
    // 配置 PB0 为通用推挽输出，速度为 10M
    GPIOB_CRL |= (1<<4*0);

    // PB0 输出低电平
    GPIOB_ODR &= (0<<0);

    while (1);

}
```

列表 14: 代码清单 7-8c F103-拂晓电机开发板对应的 main 函数代码

```
int main(void)
{
    // 开启 GPIOA 端口 时钟
    RCC_APB2ENR |= (1<<2);

    // 清空控制 PA8 的端口位
    GPIOA_CRH &= ~( 0x0F<< (4*0));
    // 配置 PA8 为通用推挽输出，速度为 10M
    GPIOA_CRH |= (1<<4*0);

    // PA8 输出低电平
    GPIOA_ODR &= ~(1<<8);

    while (1);

}
```

在本章节中，要求完全理解 stm32f10x.h 文件及 main 文件的内容 (RCC 相关的可以除外)。

7.3.5 下载验证

把编译好的程序下载到开发板并复位，可看到板子上的 LED 灯被点亮。

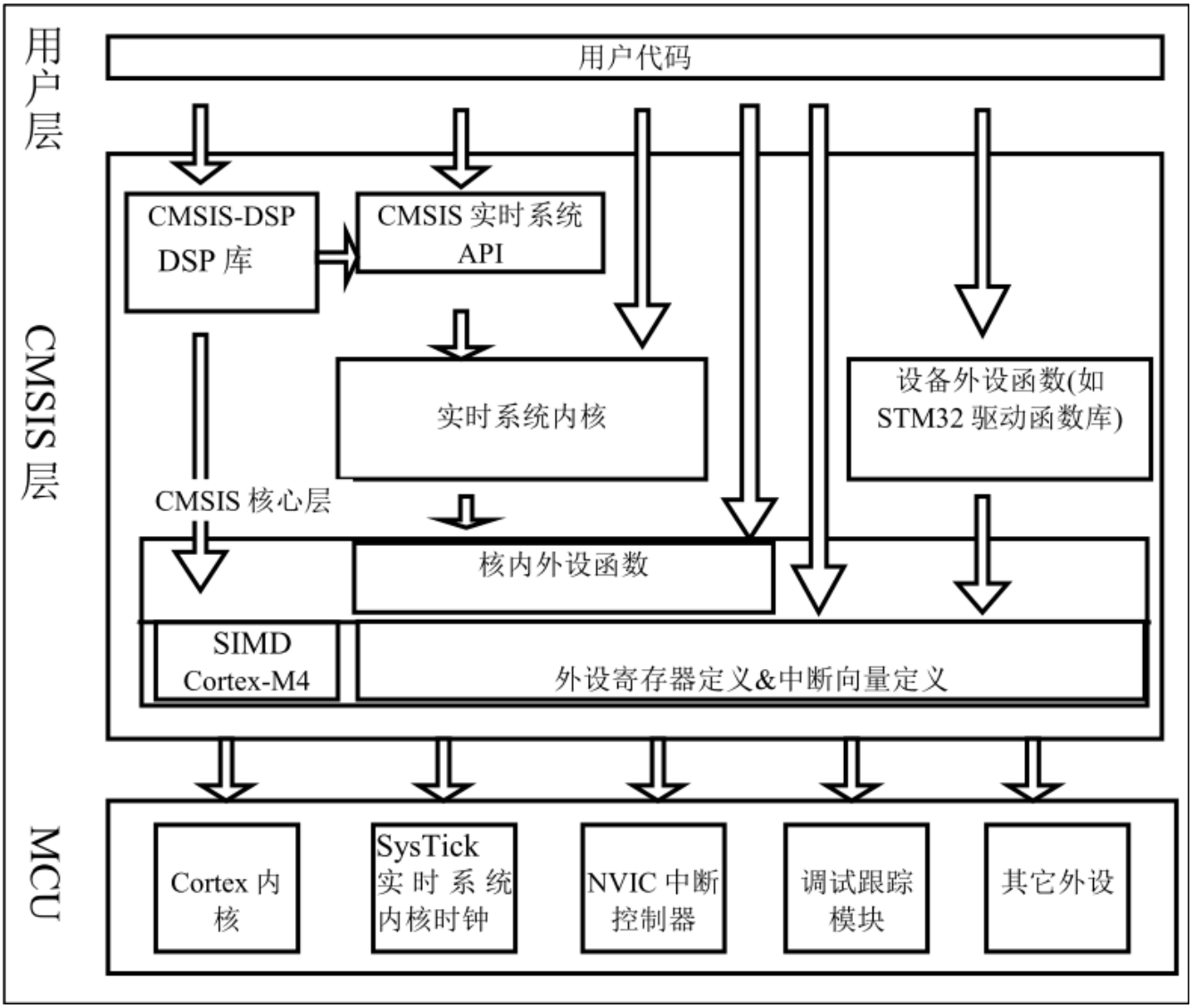
第 8 章　初识 HAL 固件库

在上一章中，我们构建了几个控制 GPIO 外设的函数，算是实现了函数库的雏形，但 GPIO 还有很多功能函数我们没有实现，而且 STM32 芯片不仅仅只有 GPIO 这一个外设。如果我们想要亲自完成这个函数库，工作量是非常巨大的。ST 公司提供的 HAL 软件库，包含了 STM32 芯片所有寄存器的控制操作，我们直接学习如何使用 ST 的 HAL 库，会极大地方便控制 STM32 芯片。。

8.1 CMSIS 标准及库层次关系

因为基于 Cortex 系列芯片采用的内核都是相同的，区别主要为核外的片上外设的差异，这些差异却导致软件在同内核，不同外设的芯片上移植困难。为了解决不同的芯片厂商生产的 Cortex 微控制器软件的兼容性问题，ARM 与芯片厂商建立了 CMSIS 标准 (Cortex MicroController Software Interface Standard)。

所谓 CMSIS 标准，实际是新建了一个软件抽象层。见图 *9_1*。



CMSIS 标准中最主要的为 CMSIS 核心层，它包括了：

- 内核函数层：其中包含用于访问内核寄存器的名称、地址定义，主要由 ARM 公司提供。
- 设备外设访问层：提供了片上的核外外设的地址和中断定义，主要由芯片生产商提供。

可见 CMSIS 层位于硬件层与操作系统或用户层之间，提供了与芯片生产商无关的硬件抽象层，可以为接口外设、实时操作系统提供简单的处理器软件接口，屏蔽了硬件差异，这对软件的移植是有极大的好处的。STM32 的库，就是按照 CMSIS 标准建立的。

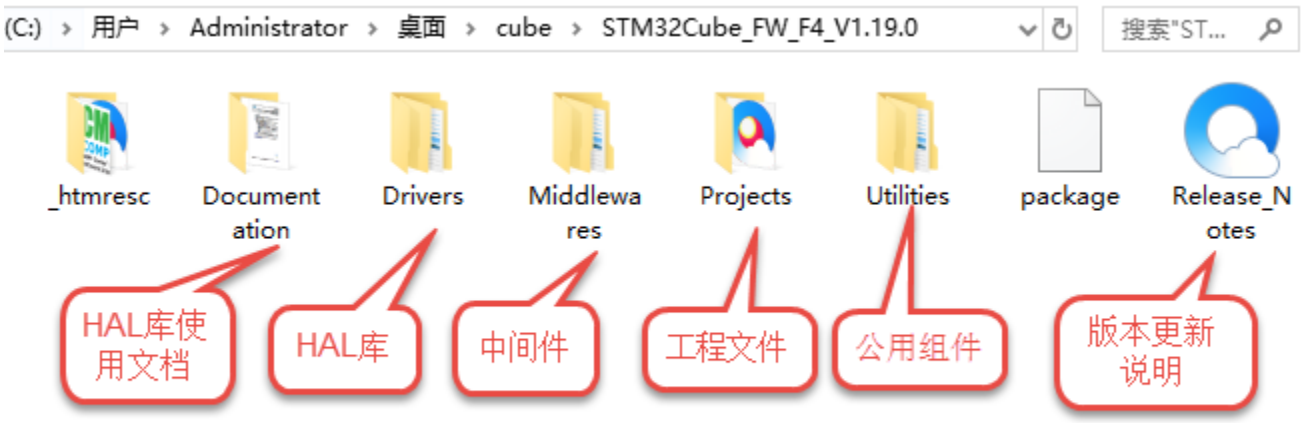
8.1.1 库目录、文件简介

STM32 HAL 库可以从官网获得，也可以直接从本书的配套资料得到。本书讲解的例程全部 HAL 库文件。以下内容请大家打开最新的 STM32HAL 库文件配合阅读。

解压库文件后进入其目录：

“STM32Cube_FW_F1_V1.8.0”

软件库各文件夹的内容说明见图 *9_2*。



目录：STM32Cube_FW_F1_V1.8.0\

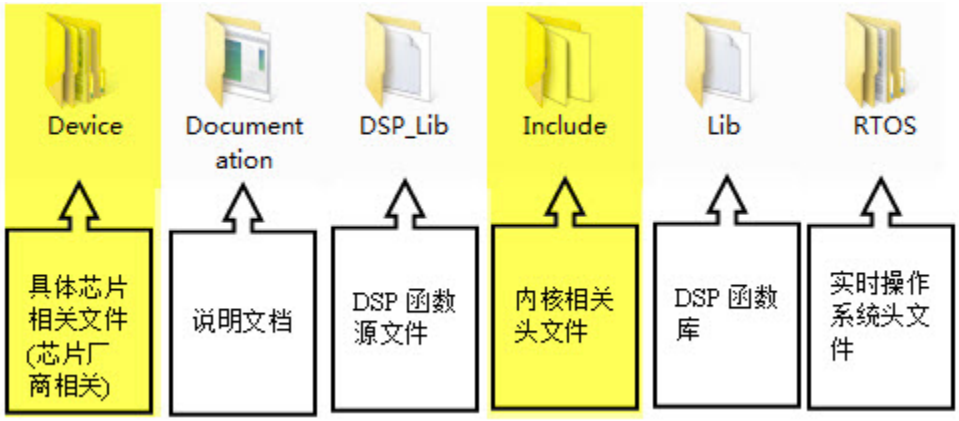
- Documentation：文件夹下是 HAL 库帮助文档，主要讲述如何使用驱动库来编写自己的应用程序。说得形象一点就是告诉我们：ST 公司已经为你写好了每个外设的驱动了，想知道如何运用这些例子就来向我求救吧。不幸的是，这个帮助文档是英文的，这对很多英文不好的朋友来说是一个很大的障碍。但这里要告诉大家，英文仅仅是一种工具，绝对不能让它成为我们学习的障碍。其实这些英文还是很简单的，我们需要的是拿下它的勇气。
- Drivers：文件夹下是官方的 CMSISI 库，HAL 库，板载外设驱动。
- Middlewares：中间件，包含 ST 官方的 STemWin、STM32_Audio、STM32_USB_Device_Library、STM32_USB_Host_Library；也有第三方的 fatfs 文件系统等等。
- *Project*：文件夹下是用驱动库写的针对官方发行 demo 板的例子和工程模板。

- Utilities：实用的公用组件比如 LCD_LOG 实用液晶打印调试信息。
- Release_Note.html:：库的版本更新说明。

在使用库开发时，我们需要把 *Drivers* 目录下的 CMSIS、STM32F1xx_HAL_Driver 内核与外设的库文件添加到工程中，并查阅库帮助文档来了解 ST 提供的库函数，这个文档说明了每一个库函数的使用方法。

先看看 CMSIS 文件夹。

STM32Cube_FW_F1_V1.8.0\Drivers\CMSIS\文件夹下内容。



目录：Drivers *CMSIS*\

其中 Device 与 Include 中的文件是我们使用得最多的，先讲解这两个文件夹中的内容。

8.1.1.1 Include 文件夹

在 *Include* 文件夹中包含了的是位于 CMSIS 标准的核内设备函数层的 Cortex-M 核通用的头文件，它们的作用是为那些采用 Cortex-M 核设计 SOC 的芯片商设计的芯片外设提供一个进入内核的接口，定义了一些内核相关的寄存器 (类似我们前面写的 stm32F103xx.h 文件，但定义的是内核部分的寄存器)。这些文件在其它公司的 Cortex-M 系列芯片也是相同的。至于这些功能是怎样用源码实现的，可以不用管它，只需把这些文件加进我们的工程文件即可，有兴趣的朋友可以深究，关于内核的寄存器说明，需要查阅《cortex_M3_Technical Reference Manual》及《Cortex®-M3 内核编程手册》文档，《STM32F103xxx 参考手册》只包含片上外设说明，不包含内核寄存器。

我们写 STM32F1 的工程，必须用到其中的四个文件：core_cm3.h、core_cmFunc.h、corecmInstr.h、core_cmSimd.h，其它的文件是属于其它内核的，还有几个文件是 DSP 函数库使用的头文件。

core_cM3.c 文件有一些与编译器相关条件编译语句，用于屏蔽不同编译器的差异。里面包含了一些跟编译器相关的信息，如：“__CC_ARM ”(本书采用的 RVMDK、KEIL)，“__GNUC__ ”(GNU 编译器)、“ICC Compiler” (IAR 编译器)。这些不同的编译器对于 C 嵌入汇编或内联函数关键字的语法不一样，这段代码统一使用 “__ASM、__INLINE” 宏来定义，而在不同的编译器下，宏自动更改到相应的值，实现了差异屏蔽，见代码清单 9-1。

列表 1: 代码清单 9-1：core_cm3.c 文件中对编译器差异的屏蔽

```
#if   defined ( __CC_ARM )
#define __ASM            __asm     /*!< asm keyword for ARM Compiler */
#define __INLINE         __inline  /*!< inline keyword for ARM Compiler*/
#define __STATIC_INLINE  static __inline

#elif defined ( __GNUC__ )
#define __ASM            __asm  /*!< asm keyword for GNU Compiler */
#define __INLINE         inline /*!< inline keyword for GNU Compiler */
#define __STATIC_INLINE  static inline

#elif defined ( __ICCARM__ )
#define __ASM            __asm  /*!< asm keyword for IAR Compiler */
/*!< inline keyword for IAR Compiler. */
#define __STATIC_INLINE  static inline
#define __INLINE         inline

#elif defined ( __TMS470__ )
#define __ASM            __asm  /*!< asm keyword for TI CCS Compiler */
#define __STATIC_INLINE  static inline

#elif defined ( __TASKING__ )
```

(下页继续)

(续上页)

```
  #define __ASM            __asm /*!< asm keyword for TASKING Compiler */
  #define __INLINE         inline /*!< inline keyword for TASKING Compiler
↪*/

  #define __STATIC_INLINE  static inline

  #elif defined ( __CSMC__ )
  #define __packed
  #define __ASM            _asm /*!< asm keyword for COSMIC Compiler */
  /*use -pc99 on compile line !< inline keyword for COSMIC Compiler   */
  #define __INLINE         inline
  #define __STATIC_INLINE  static inline

  #endif
```

较重要的是在 core_cm3.c 文件中包含了“stdint.h”这个头文件，这是一个 ANSI C 文件，是独立于处理器之外的，就像我们熟知的 C 语言头文件“stdio.h”文件一样。位于 RVMDK 这个软件的安装目录下，主要作用是提供一些类型定义。见代码清单 9_2。

列表 2: 代码清单 9-2：*stdint.c* 文件中的类型定义

```
/* 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;
```

这些新类型定义屏蔽了在不同芯片平台时，出现的诸如 int 的大小是 16 位，还是 32 位的差异。所以在我们以后的程序中，都将使用新类型如 *uint8_t* 、*uint16_t* 等。

在稍旧版的程序中还经常会出现如 *u8*、*u16*、*u32* 这样的类型，分别表示的无符号的 8 位、16 位、32 位整型。初学者碰到这样的旧类型感觉一头雾水，它们定义的位置在 *Stm32F103xx.h* 文件中。建议在以后的新程序中尽量使用 *uint8_t*、*uint16_t* 类型的定义。

core_cm3.c 跟启动文件一样都是底层文件，都是由 ARM 公司提供的，遵守 CMSIS 标准，即所有 CM3 芯片的库都带有这个文件，这样软件在不同的 CM3 芯片的移植工作就得以简化。

8.1.1.2 Device 文件夹

在 *Device* 文件夹下的是具体芯片直接相关的文件，包含启动文件、芯片外设寄存器定义、系统时钟初始化功能的一些文件，这是由 ST 公司提供的。

- system_stm32f1xx.c 文件

文件目录：\Drivers\CMSIS\ Device\ST\stm32f1xx\Source\Templates

这个文件包含了 *STM32* 芯片上电后初始化系统时钟、扩展外部存储器用的函数，例如我们前两章提到供启动文件调用的“*SystemInit*”函数，用于上电后初始化时钟，该函数的定义就存储在

system_stm32f1xx.c 文件。*STM32F103* 系列的芯片，调用库的这个 *SystemInit* 函数后，系统时钟被初始化为 *72MHz*，如有需要可以修改这个文件的内容，设置成自己所需的时钟频率。

- 启动文件

文件目录：*\Drivers\CMSIS\Device\ST\stm32f1xx\Source\Templates*

在这个目录下，还有很多文件夹，如“*ARM*”、“*gcc*”、“*iar*”等，这些文件夹下包含了对应编译平台的汇编启动文件，在实际使用时要根据编译平台来选择。我们使用的 *MDK* 启动文件在“*ARM*”文件夹中。其中的“*strartup_STM32F103xx.s*”即为 *STM32F103* 芯片的启动文件，前面两章工程中使用的启动文件就是从这里复制过去的，*strartup_STM32F103xe.s(xe* 适用于大容量产品*)* 可兼容 *F103* 的 *ve,ze,rc* 芯片。如果使用其它型号的芯片，要在此处文件夹选择其他对应的启动文件。

- *stm32F103xx.h* 文件

文件目录：*\Drivers\CMSIS\Device\ST\stm32f1xx\Include*

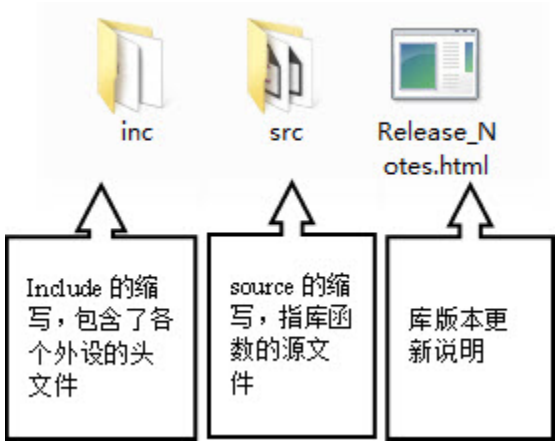
stm32F103xx.h 这个文件非常重要，是一个 STM32 芯片底层相关的文件。它是我们前两章自己定义的“stm32F103xx.h”文件的完整版，包含了 STM32 中所有的外设寄存器地址和结构体类型定义，在使用到 STM32 HAL 库的地方都要包含这个头文件。

CMSIS 文件夹中的主要内容就是这样，接下来我们看看 STM32F1xx_HAL_Driver 文件夹。

8.1.1.3 STM32F1xx_ HAL _Driver 文件夹

文件目录：DriversSTM32F1xx_HAL_Driver

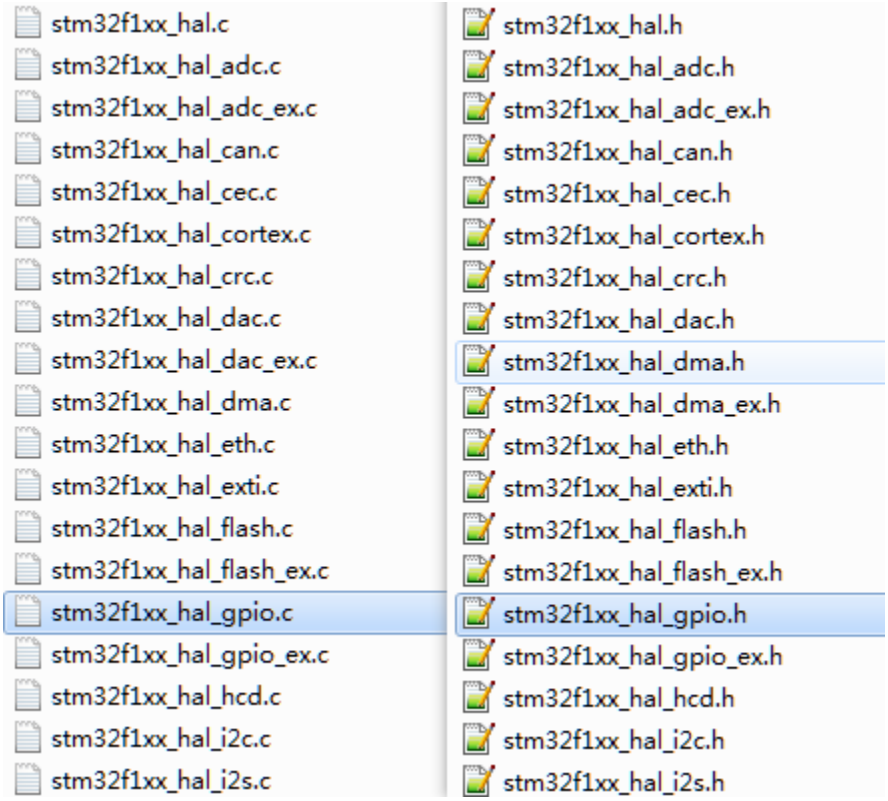
进入 Drivers 目录下的 STM32F1xx_HAL_Driver 文件夹，见图 *9_4*。



STM32F1xx_HAL_Driver 文件夹下有 *inc*（include 的缩写）跟 *src*（source 的简写）这两个文件夹，这里的文件属于 CMSIS 之外的的、芯片片上外设部分。*src* 里面是每个设备外设的驱动源程序，inc 则是相对应的外设头文件。src 及 inc 文件夹是 ST 的 HAL 库的主要内容，甚至不少人直接认为 ST 的 HAL 库就是指这些文件，可见其重要性。

在 *src* 和 *inc* 文件夹里的就是 ST 公司针对每个 STM32 外设而编写的库函数文件，每个外设对应一个 *.c* 和 *.h* 后缀的文件。我们把这类外设文件统称为：*stm32f1xx_hal_ppp.c* 或 *stm32f1xx_hal_ppp.h* 文件，PPP 表示外设名称。如在上一章中我们自建的 stm32f1xx_hal_gpio.c 及 stm32f1xx_hal_gpio.h 文件，就属于这一类。

如针对模数转换 (ADC) 外设，在 *src* 文件夹下有一个 *stm32f1xx_hal_adc.c* 源文件，在 *inc* 文件夹下有一个 *stm32f1xx_hal_adc.h* 头文件，若我们开发的工程中用到了 STM32 内部的 ADC，则至少要把这两个文件包含到工程里。见图 *9_5*。



8.1.1.4 stm32f1xx_it.c、stm32f1xx_hal_conf.h 文件

文件目录：STM32Cube_FW_F4_V1.19.0\Projects\STM32F103ZI-Nucleo\Templates

在这个文件目录下，存放了官方的一个库工程模板，我们在用库建立一个完整的工程时，还需要添加这个目录下 src 文件夹中 *stm32f1xx_it.c* 和 inc 文件夹中和 *stm32f1xx_it.h*、*stm32f1xx_hal_conf.h* 这三个文件。

stm32f1xx_it.c：这个文件是专门用来编写中断服务函数的，在我们修改前，这个文件已经定义了一些系统异常 (特殊中断) 的接口，其它普通中断服务函数由我们自己添加。但是我们怎么知道这些中断服务函数的接口如何写？是不是可以自定义呢？答案当然不是的，这些都有可以在汇编启动文件中找到，在学习中断和启动文件的时候我们会详细介绍

stm32f1xx_hal_conf.h：这个文件被包含进 *stm32f103xx.h* 文件。STM32HAL 库支持所有 STM32F1 型号的芯片，但有的型号芯片外设功能比较多，所以使用这个配置文件根据芯片型号增减 ST 库的外设文件, 另外时钟源配置也是在这里进行设置。见代码清单 *9_3*。

列表 3: 代码清单 9-3 stm32f1xx_hal_conf.h 文件配置软件库

```
/* Includes ------------------------------------------------------*/
/**
* @brief Include module's header file
*/

#ifdef HAL_RCC_MODULE_ENABLED
#include "stm32f1xx_hal_rcc.h"
#endif /* HAL_RCC_MODULE_ENABLED */

#ifdef HAL_GPIO_MODULE_ENABLED
#include "stm32f1xx_hal_gpio.h"
#endif /* HAL_GPIO_MODULE_ENABLED */

#ifdef HAL_DMA_MODULE_ENABLED
#include "stm32f1xx_hal_dma.h"
#endif /* HAL_DMA_MODULE_ENABLED */

#ifdef HAL_CORTEX_MODULE_ENABLED
#include "stm32f1xx_hal_cortex.h"
#endif /* HAL_CORTEX_MODULE_ENABLED */

#ifdef HAL_ADC_MODULE_ENABLED
#include "stm32f1xx_hal_adc.h"
#endif /* HAL_ADC_MODULE_ENABLED */

#ifdef HAL_CAN_MODULE_ENABLED
#include "stm32f1xx_hal_can.h"
#endif /* HAL_CAN_MODULE_ENABLED */
```

stm32f1xx_hal_conf.h 这个文件还可配置是否使用“断言”编译选项，见代码清单 9_4。

列表 4: 代码清单 9-4 断言配置

```
#ifdef  USE_FULL_ASSERT

/**
* @brief  The assert_param macro is used for  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)0 : assert_failed((uint8_t
*)__FILE__, __LINE__))
/* Exported functions ------------------------------------ */
void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif /* USE_FULL_ASSERT */
```

在 ST 的 HAL 库函数中，一般会包含输入参数检查，即上述代码中的“assert_param”宏，当参数不符合要求时，会调用“assert_failed”函数，这个函数默认是空的。

实际开发中使用断言时，先通过定义 USE_FULL_ASSERT 宏来使能断言，然后定义“assert_failed”函数，通常我们会让它调用 printf 函数输出错误说明。使能断言后，程序运行时会检查函数的输入参数，当软件经过测试，可发布时，会取消 USE_FULL_ASSERT 宏来去掉断言功能，使程序全速运行。

8.1.2 库各文件间的关系

前面向大家简单介绍了各个库文件的作用，库文件是直接包含进工程即可，丝毫不用修改，而有的文件就要我们在使用的时候根据具体的需要进行配置。接下来从整体上把握一下各个文件在库工程中的层次或关系，这些文件对应到 CMSIS 标准架构上。见图 *9_6*。

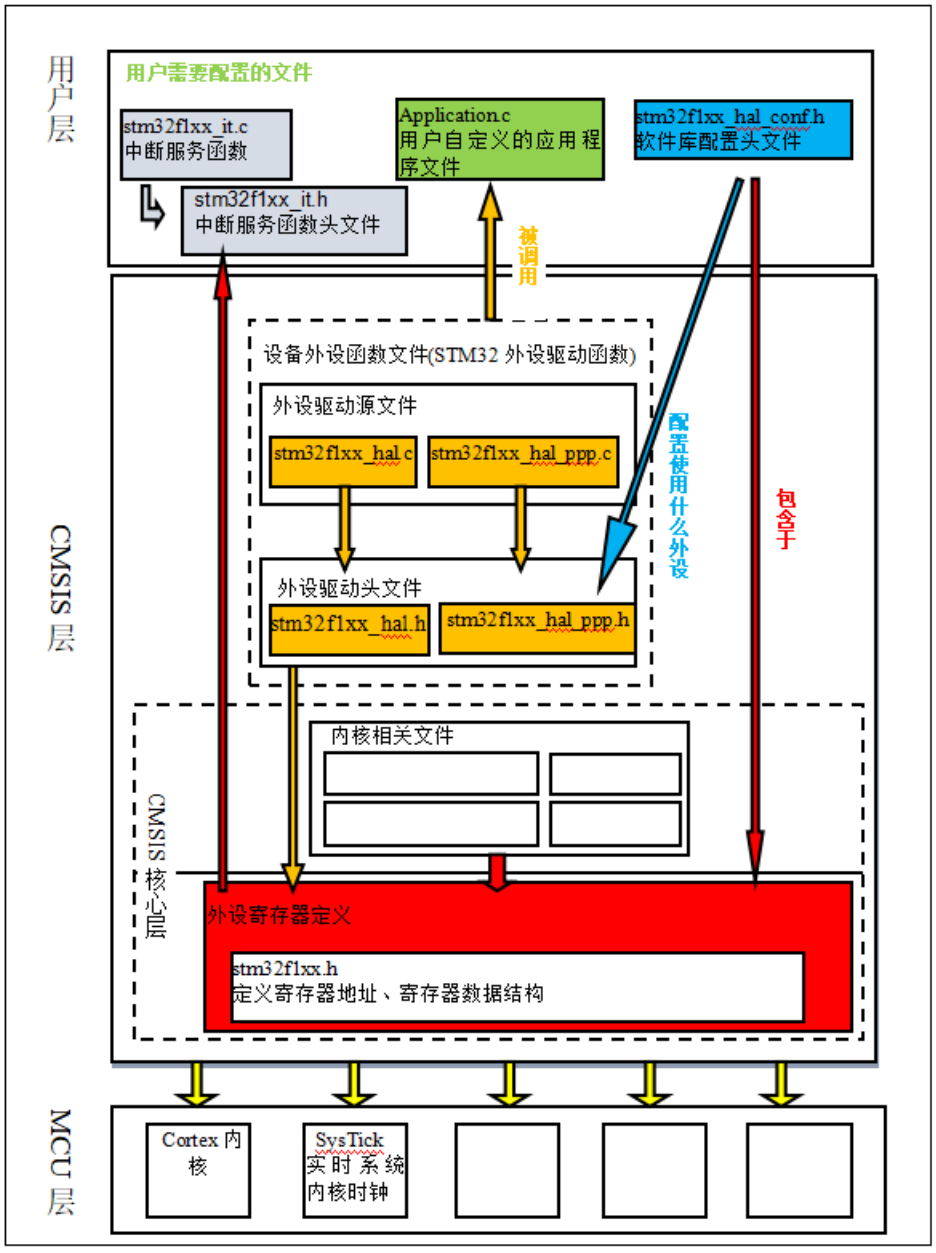


图 9-6 库各文件关系

图 *9_6* 描述了 STM32 库各文件之间的调用关系，这个图省略了 DSP 核和实时系统层部分的文件关系。在实际的使用库开发工程的过程中，我们把位于 CMSIS 层的文件包含进工程，除了特殊系统时钟需要修改 system_stm32f1xx.c，其它文件丝毫不用修改，也不建议修改。

对于位于用户层的几个文件，就是我们在使用库的时候，针对不同的应用对库文件进行增删（用条件编译的方法增删）和改动的文件。

8.2 使用帮助文档

我坚信，授之以鱼不如授之以渔。官方资料是所有关于 STM32 知识的源头，所以在本小节介绍如何使用官方资料。官方的帮助手册，是最好的教程，几乎包含了所有在开发过程中遇到的问题。这些资料已整理到了本书附录资料中。

8.2.1 常用官方资料

- 《STM32F10X-中文参考手册》

这个文件全方位介绍了 STM32 芯片的各种片上外设，它把 STM32 的时钟、存储器架构、及各种外设、寄存器都描述得清清楚楚。当我们对 STM32 的外设感到困惑时，可查阅这个文档。以直接配置寄存器方式开发的话，查阅这个文档寄存器部分的频率会相当高，但这样效率太低了。

- 《STM32 规格书》

本文档相当于 STM32 的 datasheet，包含了 STM32 芯片所有的引脚功能说明及存储器架构、芯片外设架构说明。后面我们使用 STM32 其它外设时，常常需要查找这个手册，了解外设对应到 STM32 的哪个 GPIO 引脚。

- 《Cortex™-M3 内核编程手册》

本文档由 ST 公司提供，主要讲解 STM32 内核寄存器相关的说明，例如系统定时器、NVIC 等核外设的寄存器。这部分的内容是《STM32F10X-中文参考手册》没涉及到的内核部分的补充。相对来说，本文档虽然介绍了内核寄存器，但不如以下两个文档详细，要了解内核时，可作为以下两个手册的配合资料使用。

- 《Cortex-M3 权威指南》。

这个手册是由 ARM 公司提供的，它详细讲解了 Cortex 内核的架构和特性，要深入了解 Cortex-M 内核，这是首选，经典中的经典。这个手册也被翻译成中文，出版成书，我们配套的资料里面有提供中文版的电子版。

- 《STM32F103xG_User_Manual.chm》

这个就是本章提到的库的帮助文档，在使用库函数时，我们最好通过查阅此文件来了解 HAL 库提供了哪些外设、函数原型或库函数的调用的方法。也可以直接阅读源码里面的函数的函数说明。

第 9 章　使用 STM32CubeMX 新建工程

STM32Cube 是一项意法半导体的原创活动，通过减少开发工作、时间和成本，使开发者的开发工作更轻松。STM32Cube 是一个全面的软件平台，包括了 ST 产品的每个系列。(如，STM32CubeF1 是针对 STM32F1 系列)。平台包括了 STM32Cube 硬件抽象层和一套的中间件组件 (RTOS，USB，FS，TCP/IP，Graphics，等等)。

9.1 软件安装

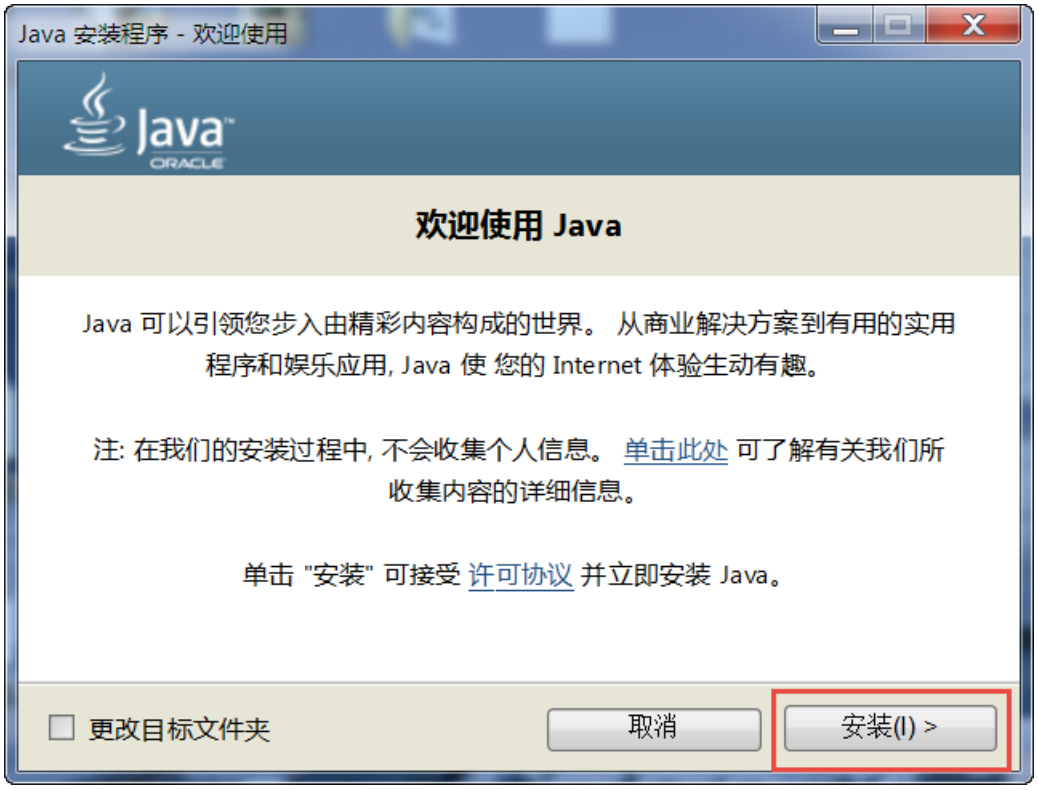
必备软件：Java（V1.7 及以上版本）和 STM32CubeMX（版本 5.3.0）

Java 下载地址：www.java.com/zh_CN/

STM32CubeMX 下载地址：ST 官网搜索 STM32CubeMX

安装 Java 软件

双击安装包“JavaSetup8u151.exe”，具体操作步骤如下：


Java 安装程序 - 欢迎使用
Java
ORACLE
欢迎使用 Java
Java 可以引领您步入由精彩内容构成的世界。 从商业解决方案到有用的实用程序和娱乐应用, Java 使 您的 Internet 体验生动有趣。
注: 在我们的安装过程中, 不会收集个人信息。 单击此处 可了解有关我们所收集内容的详细信息。
单击 "安装" 可接受 许可协议 并立即安装 Java。
更改目标文件夹
取消
安装(I) >

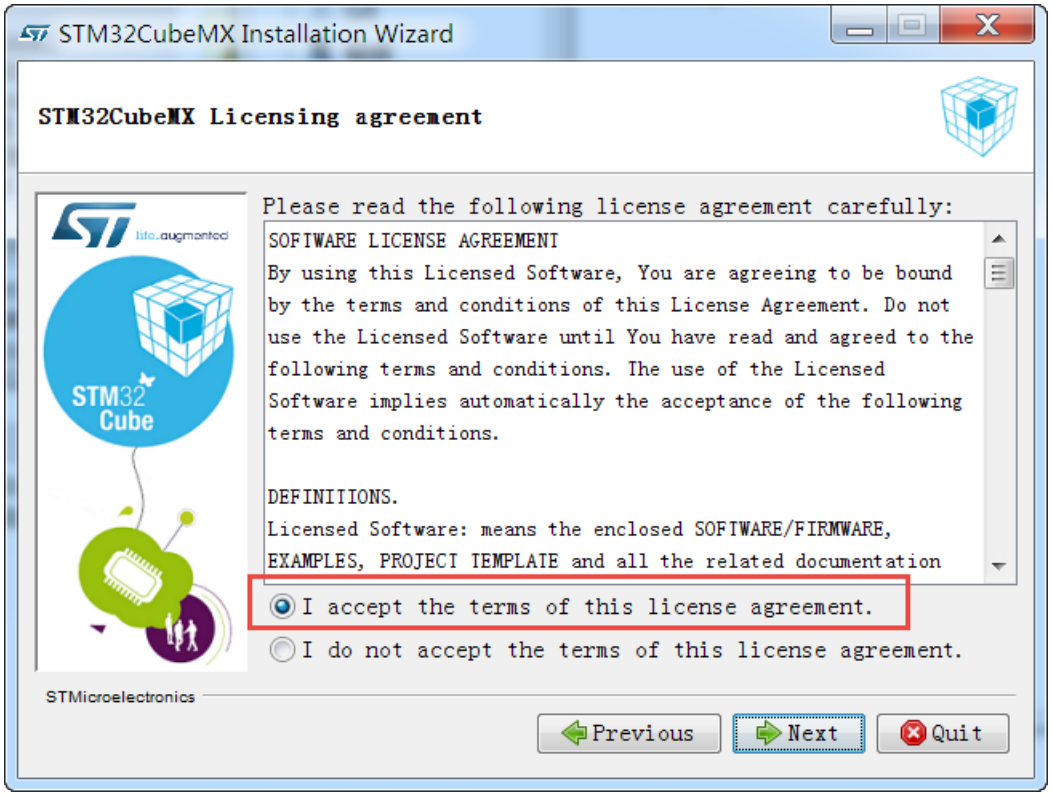
Java 安装 - 完成
Java
ORACLE
您已成功安装 Java
当有 Java 更新可用时，系统将会提示您。 请始终安装更新以获取最新的性能和安全改进。
有关更新设置的详细信息
单击 "关闭" 后，浏览器将会打开以便您可以验证 Java 是否工作正常。
关闭(C)

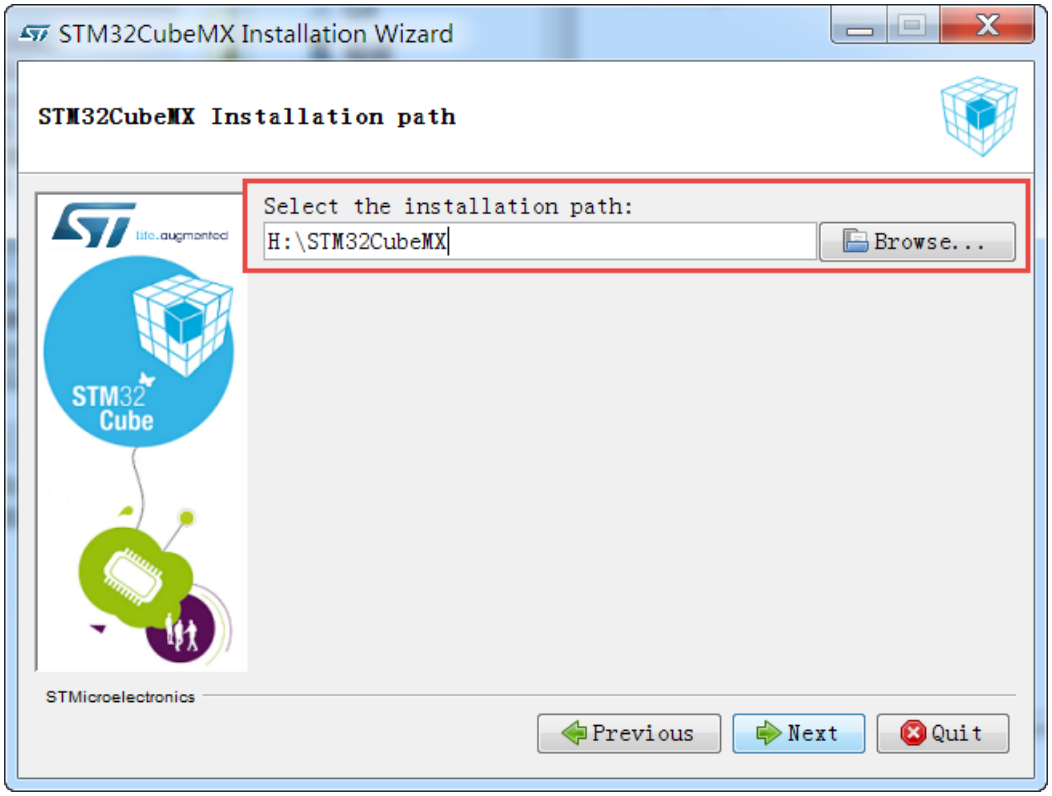
在本地新建好文件夹后，把准备好的库文件添加到相应的文件夹下：

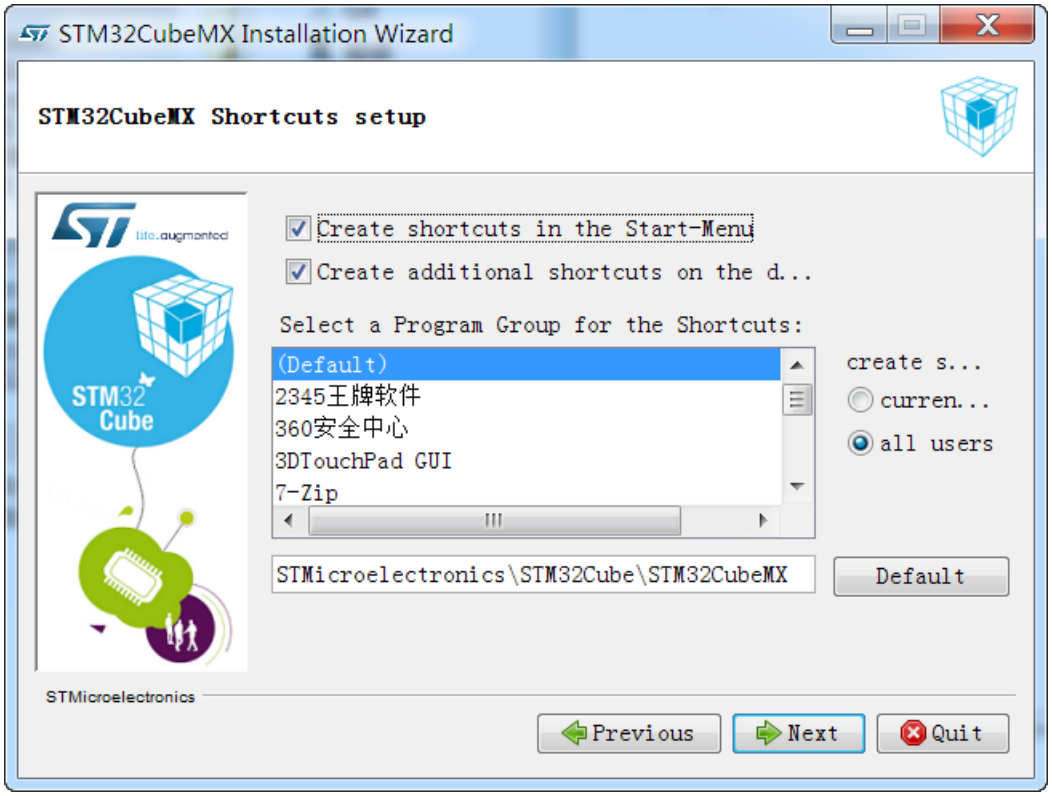
9.1.1 安装 STM32CubeMX 软件

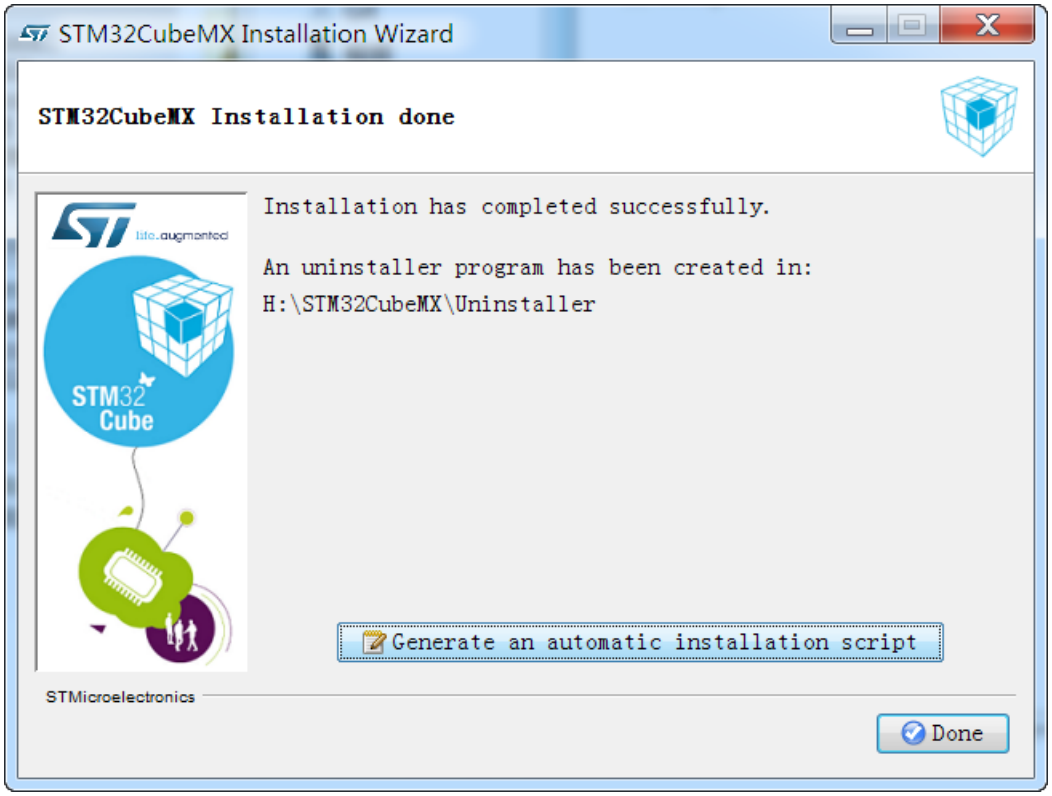
双击安装包“SetupSTM32CubeMX-5.3.0.exe”，具体操作步骤如下：

STM32CubeMX Installation Wizard
Welcome to the Installation of STM32CubeMX 4.22.1
Starting STM32CubeMX 4.22.1 installation
The homepage is at: http://www.st.com/stm32cube
STM32 Cube
STMicroelectronics
Next
Quit


STM32CubeMX Installation Wizard
STM32CubeMX Licensing agreement
Please read the following license agreement carefully:
SOFTWARE LICENSE AGREEMENT
By using this Licensed Software, You are agreeing to be bound by the terms and conditions of this License Agreement. Do not use the Licensed Software until You have read and agreed to the following terms and conditions. The use of the Licensed Software implies automatically the acceptance of the following terms and conditions.
DEFINITIONS.
Licensed Software: means the enclosed SOFTWARE/FIRMWARE, EXAMPLES, PROJECT TEMPLATE and all the related documentation
I accept the terms of this license agreement.
I do not accept the terms of this license agreement.
STMicroelectronics
Previous
Next
Quit


STM32CubeMX Installation Wizard
STM32CubeMX Installation path
Select the installation path:
H:\STM32CubeMX
Browse...
STM32 Cube
STMicroelectronics
Previous
Next
Quit

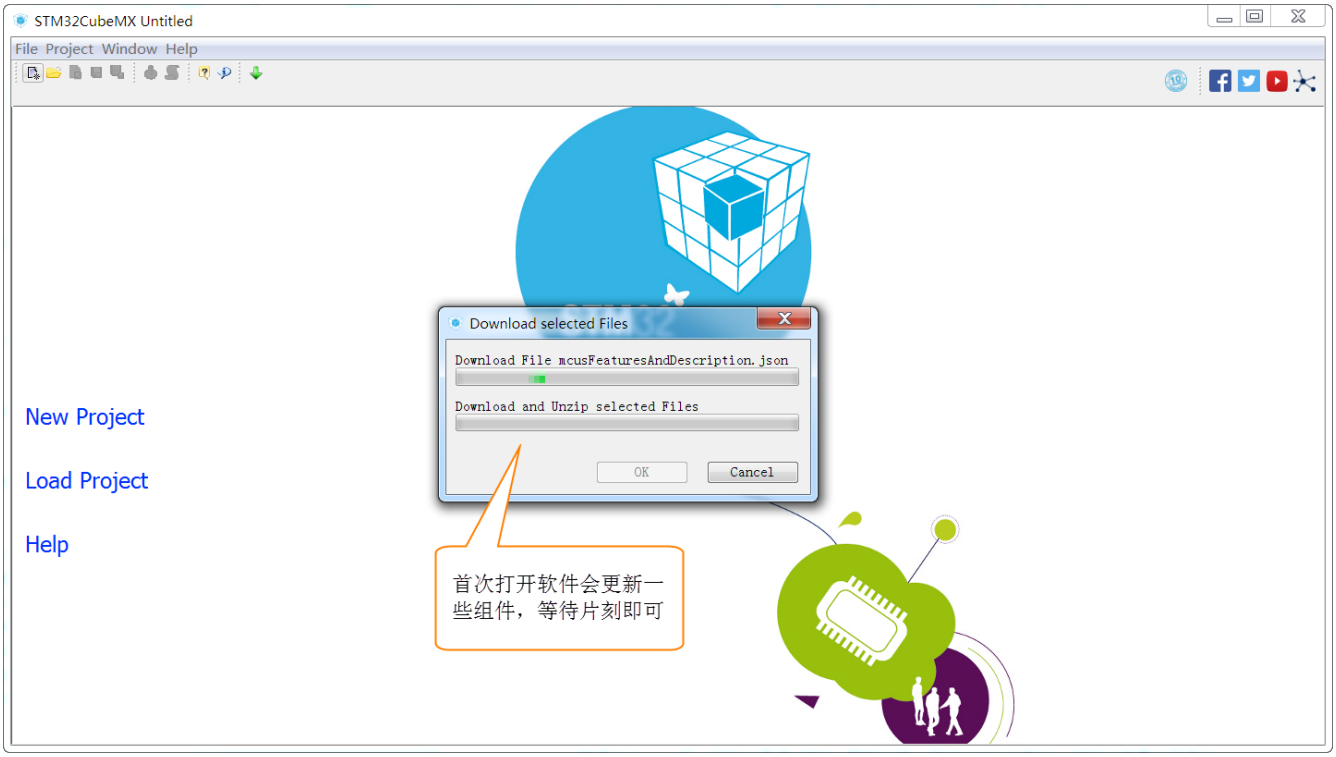

STM32CubeMX Installation Wizard
STM32CubeMX Shortcuts setup
Create shortcuts in the Start-Menu
Create additional shortcuts on the d...
Select a Program Group for the Shortcuts:
(Default)
2345王牌软件
360安全中心
3DTouchPad GUI
7-Zip
create s...
curren...
all users
STMicroelectronics\STM32Cube\STM32CubeMX
Default
STMicroelectronics
Previous
Next
Quit



9.2 新建工程

9.2.1 新建工程

打开 STM32CubeMX，软件会更新一些组件，等待安装完成即可。

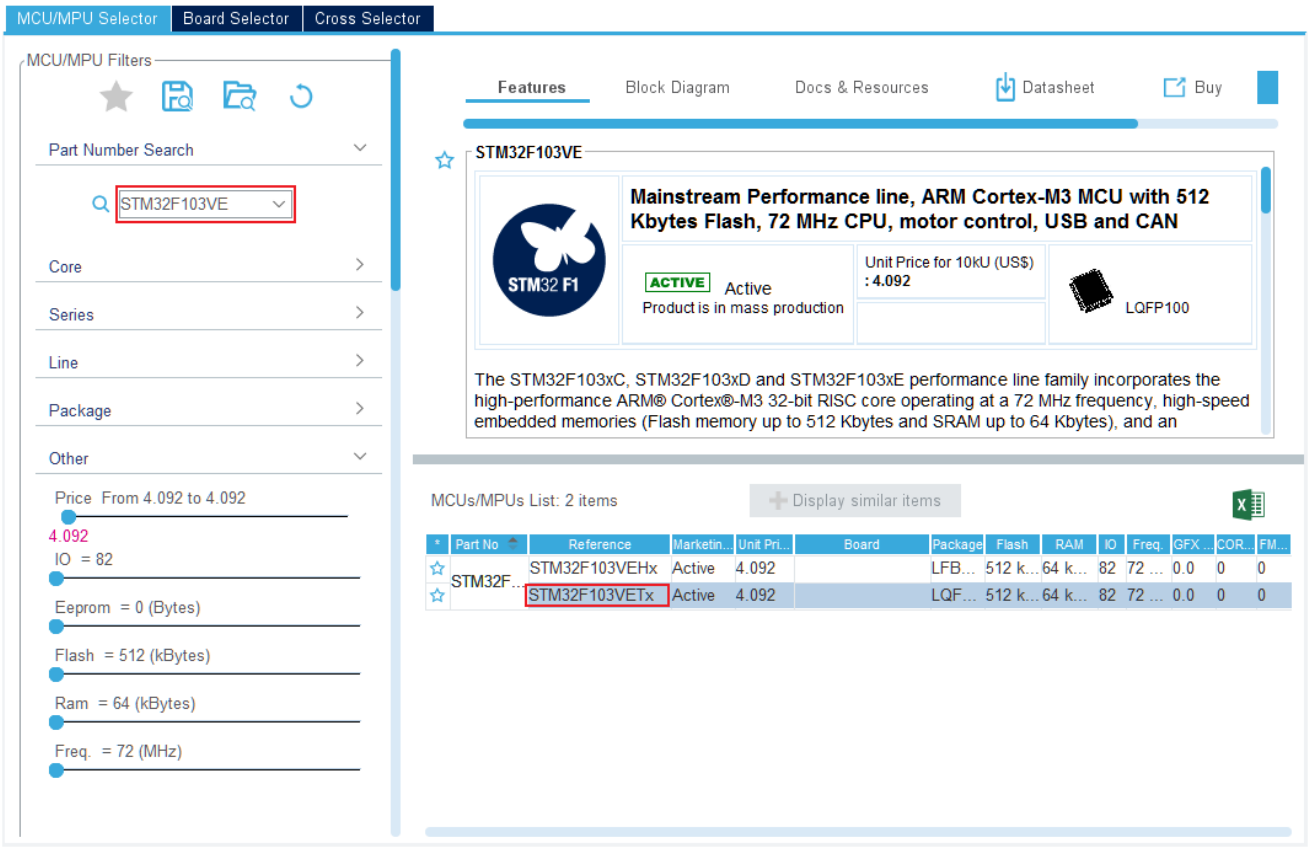


9.2.1.1 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，

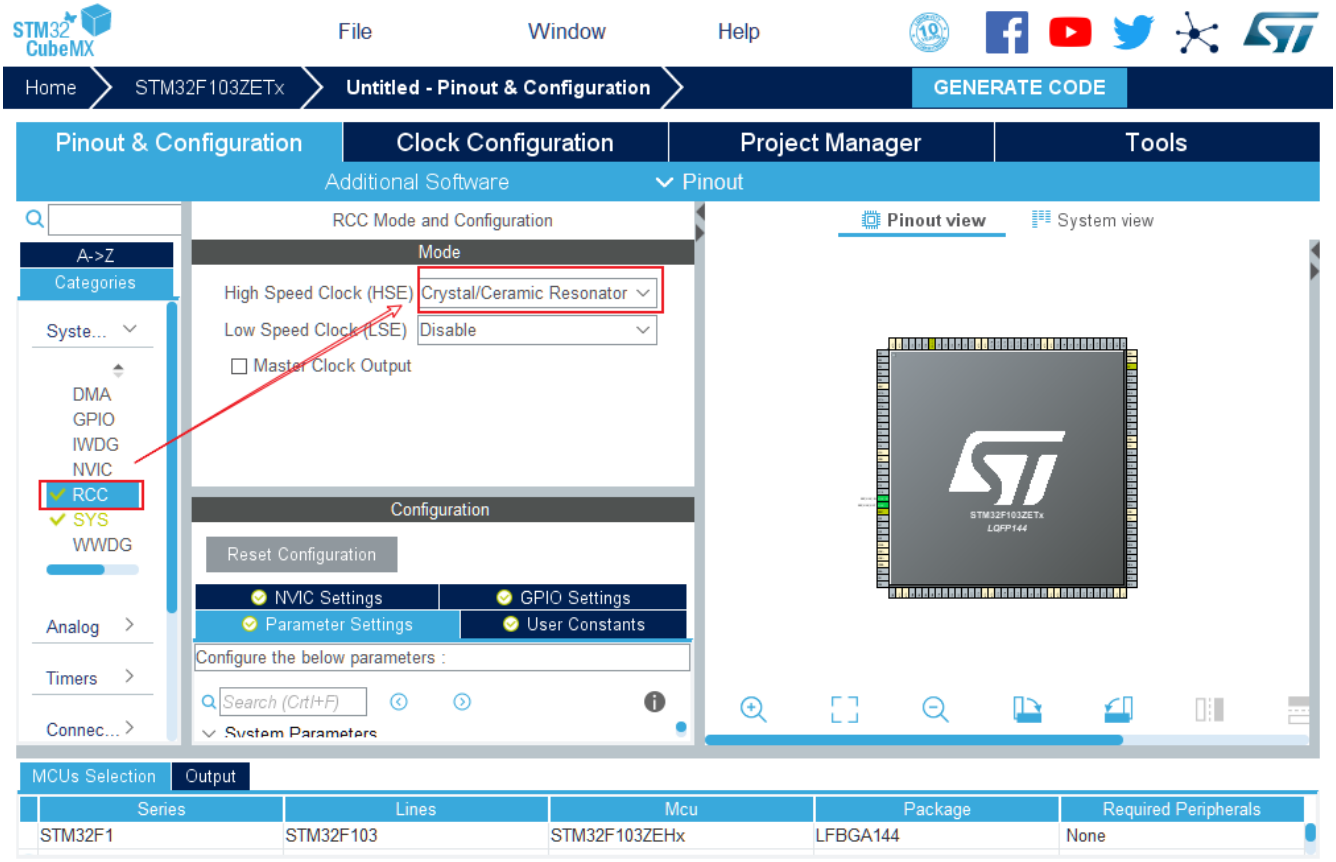
- F103-MINI，选 STM32F103RCTx 型号
- F103-指南者，选 STM32F103VETx 型号
- F103-霸道，选 STM32F103ZETx 型号
- F103-拂晓电机开发板，选 STM32F103VETx 型号

以 F103-指南者为例，我们直接在搜索框输入型号 STM32F103VE 最终确认 STM32F103VETx 为我们实际使用型号。



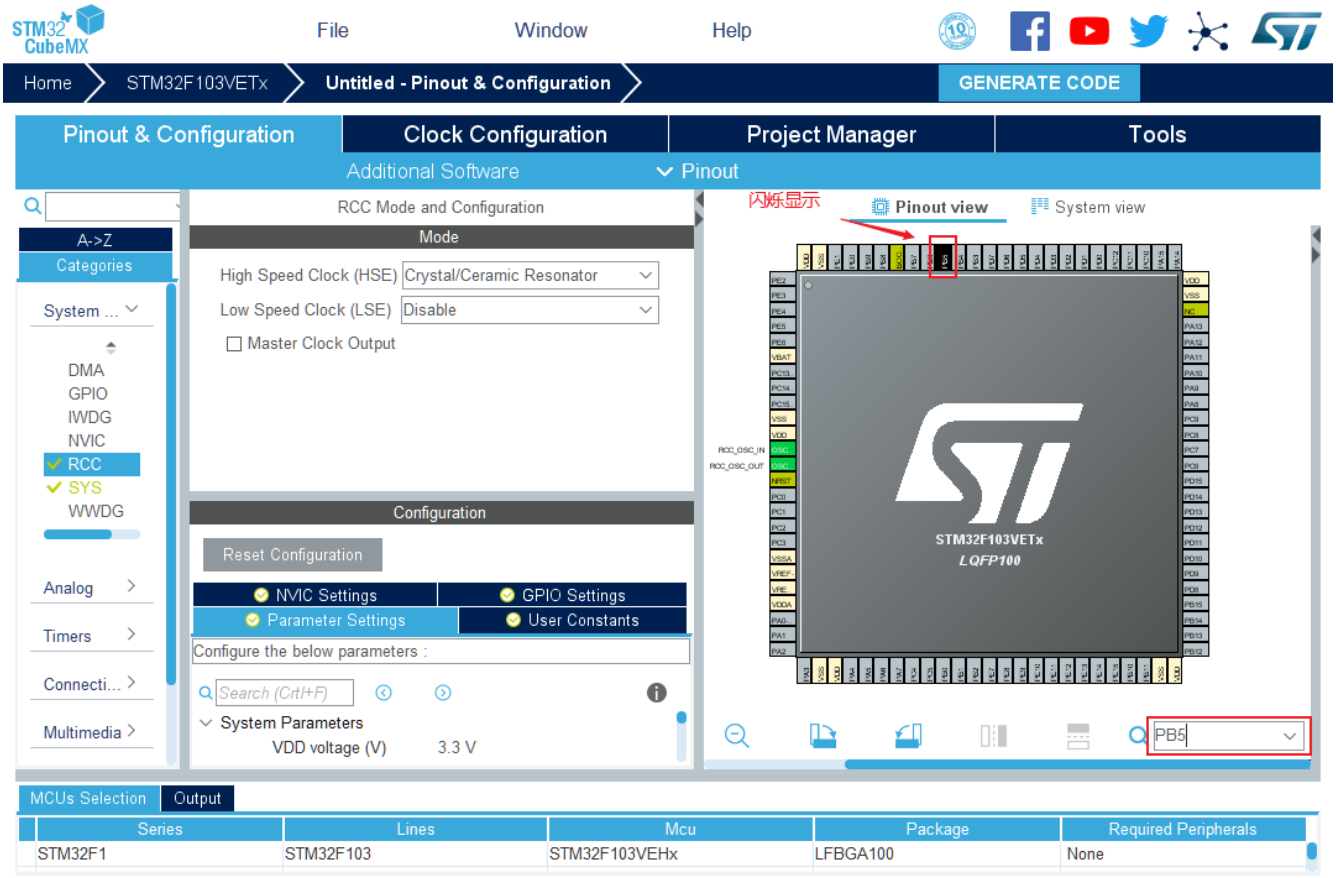
9.2.1.2 确认时钟源

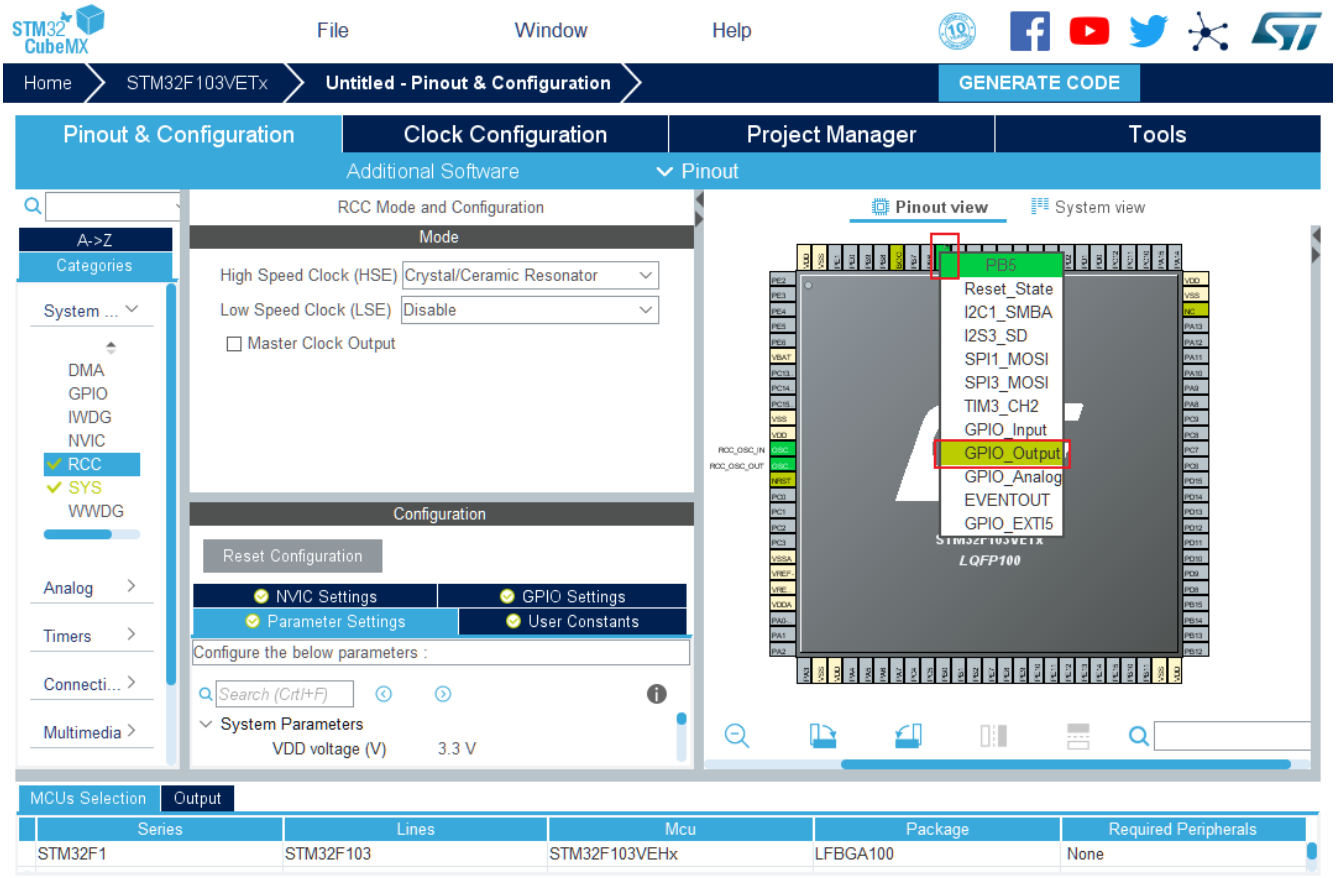
进入工程后打开 RCC 选项，选择 Crystal/Ceramic Resonator，即使用外部晶振作为 HSE 的时钟源。



9.2.1.3 配置 IO 口

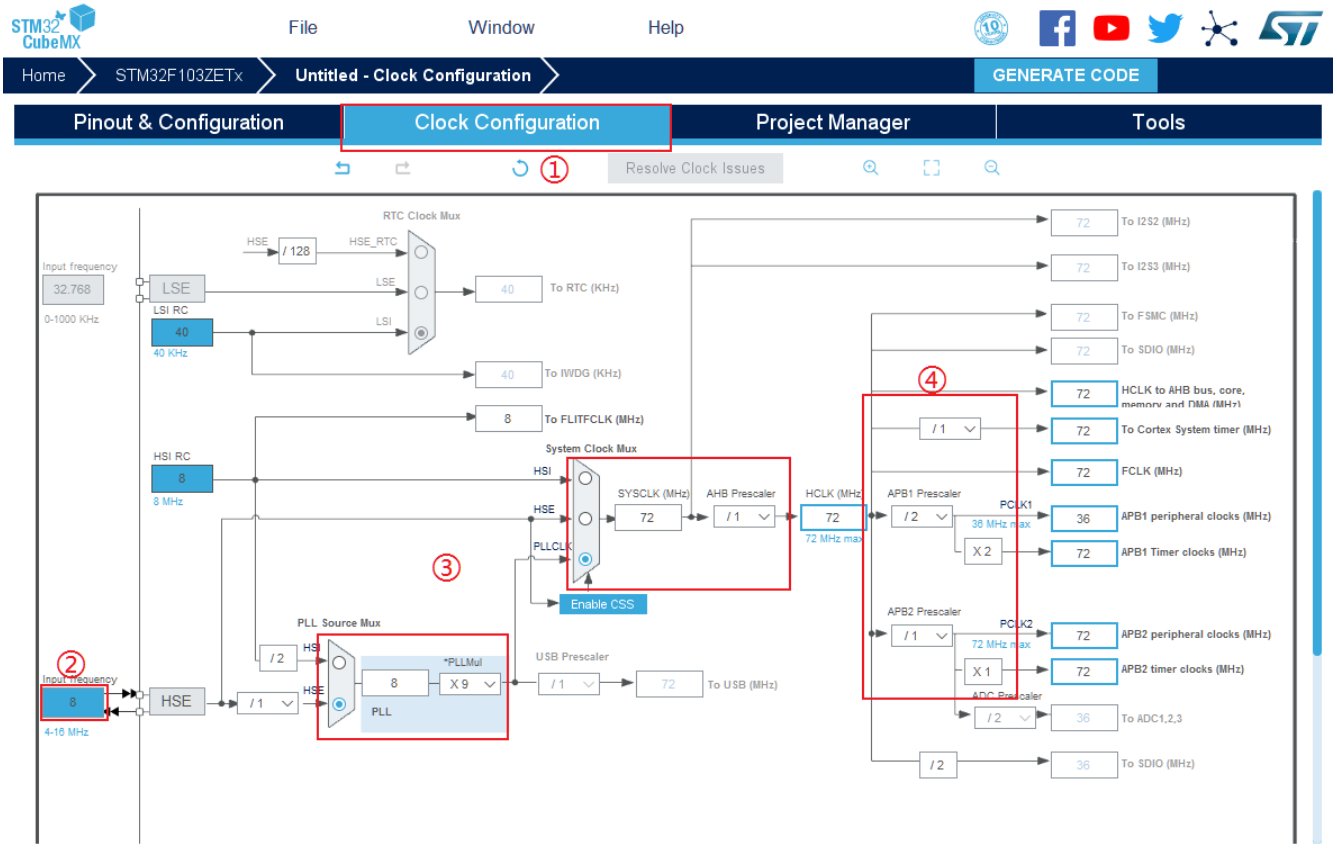
这个工程简单控制一个 LED 周期闪烁，我们只需要配置一个 IO 即可，大家可以根据自己的开发板情况来配置，这里选定控制红色 LED 的引脚 PB5，通过搜索框搜索可以定位 IO 口的引脚位置，图中会闪烁显示，配置 PB5 的属性为 GPIO_Output。


STM32 CubeMX
File
Window
Help
Home
STM32F103VETx
Untitled - Pinout & Configuration
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Additional Software
Pinout
RCC Mode and Configuration
Mode
High Speed Clock (HSE)
Crystal/Ceramic Resonator
Low Speed Clock (LSE)
Disable
Master Clock Output
Configuration
Reset Configuration
NVIC Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Search (Crtl+F)
System Parameters
VDD voltage (V)
3.3 V
A->Z
Categories
System ...
DMA
GPIO
IWDG
NVIC
RCC
SYS
WWDG
Analog
Timers
Connecti...
Multimedia
闪烁显示
Pinout view
System view
STM32F103VETx
LQFP100
PB5
MCUs Selection
Output
Series
Lines
Mcu
Package
Required Peripherals
STM32F1
STM32F103
STM32F103VEHx
LFBGA100
None



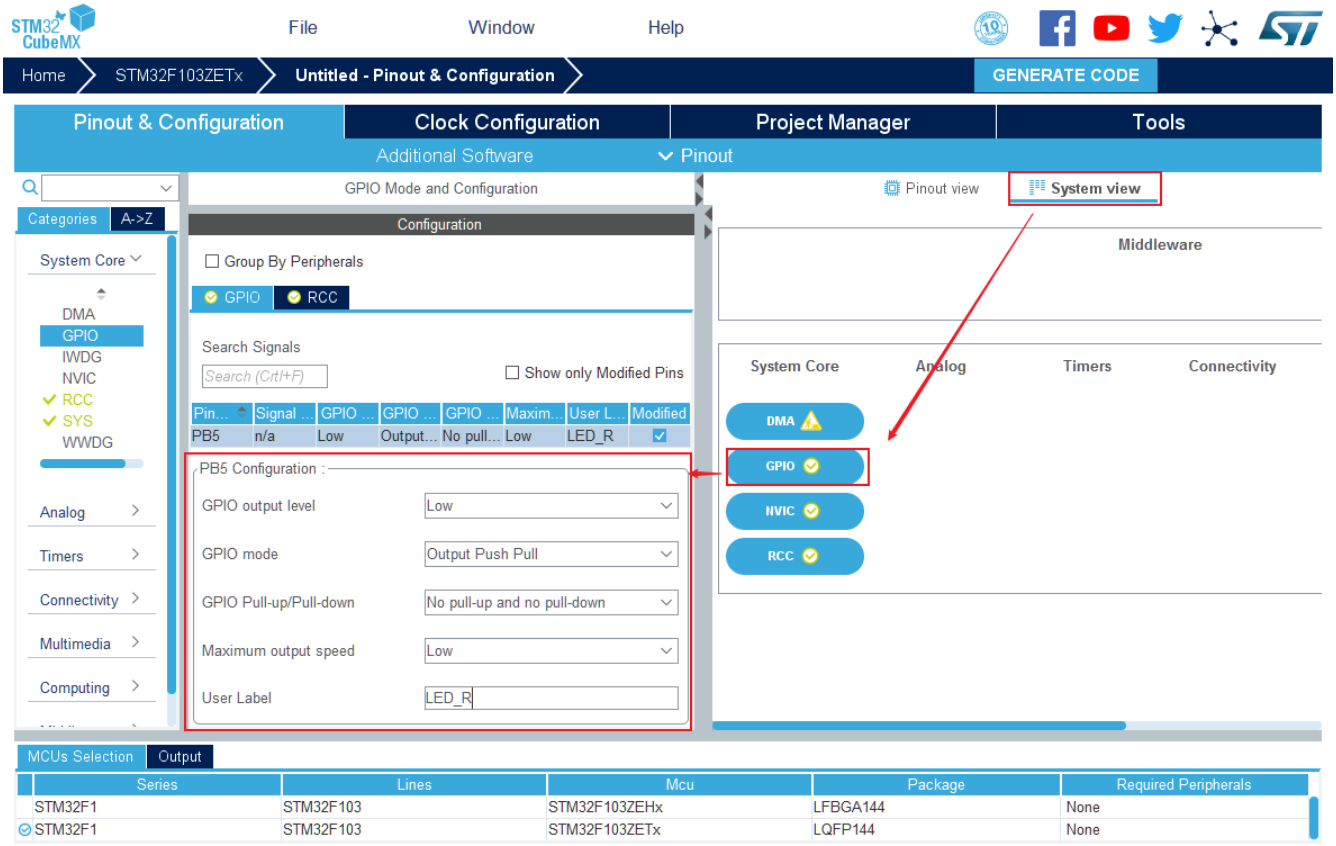
9.2.1.4 配置系统时钟

开发板的外部晶振为 8MHz，我们填入 8；通道选择 LSE；PLLM 选择为/1；倍频系数 N 选择为 x9；系统时钟选择 PLLCLK；系统时钟设定为 72Mz；APB1 分频系数选择为/2 即 PCLK1 位 36MHz；APB2 分频系数选择为/1 即 PCLK2 位 72MHz。



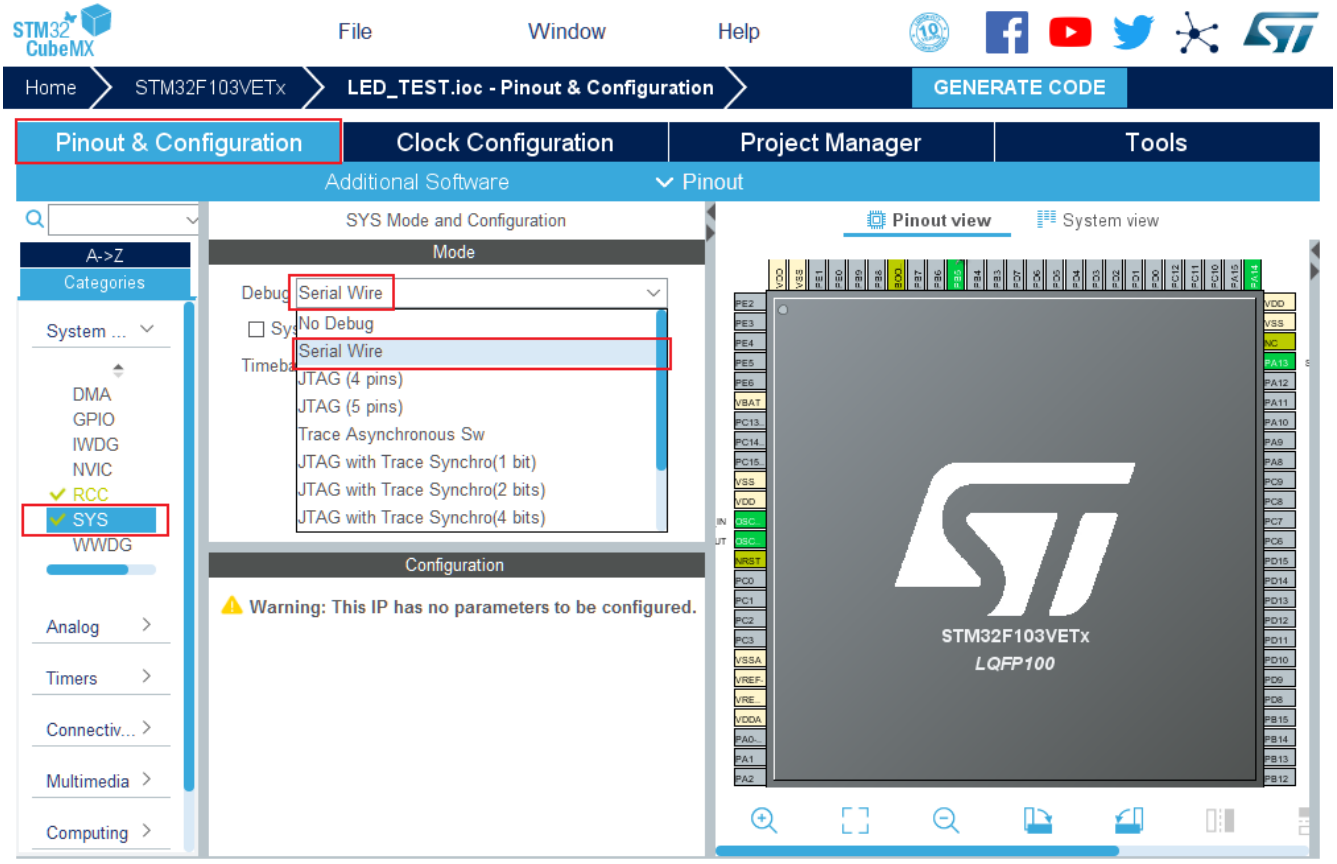
9.2.1.5 进一步配置 IO 的具体属性

点击 Configuration，进入系统详细配置，选择 GPIO，配置 PB5 的默认电平，开漏输出，无上下拉，低速模式。引脚标签为 LED_R。

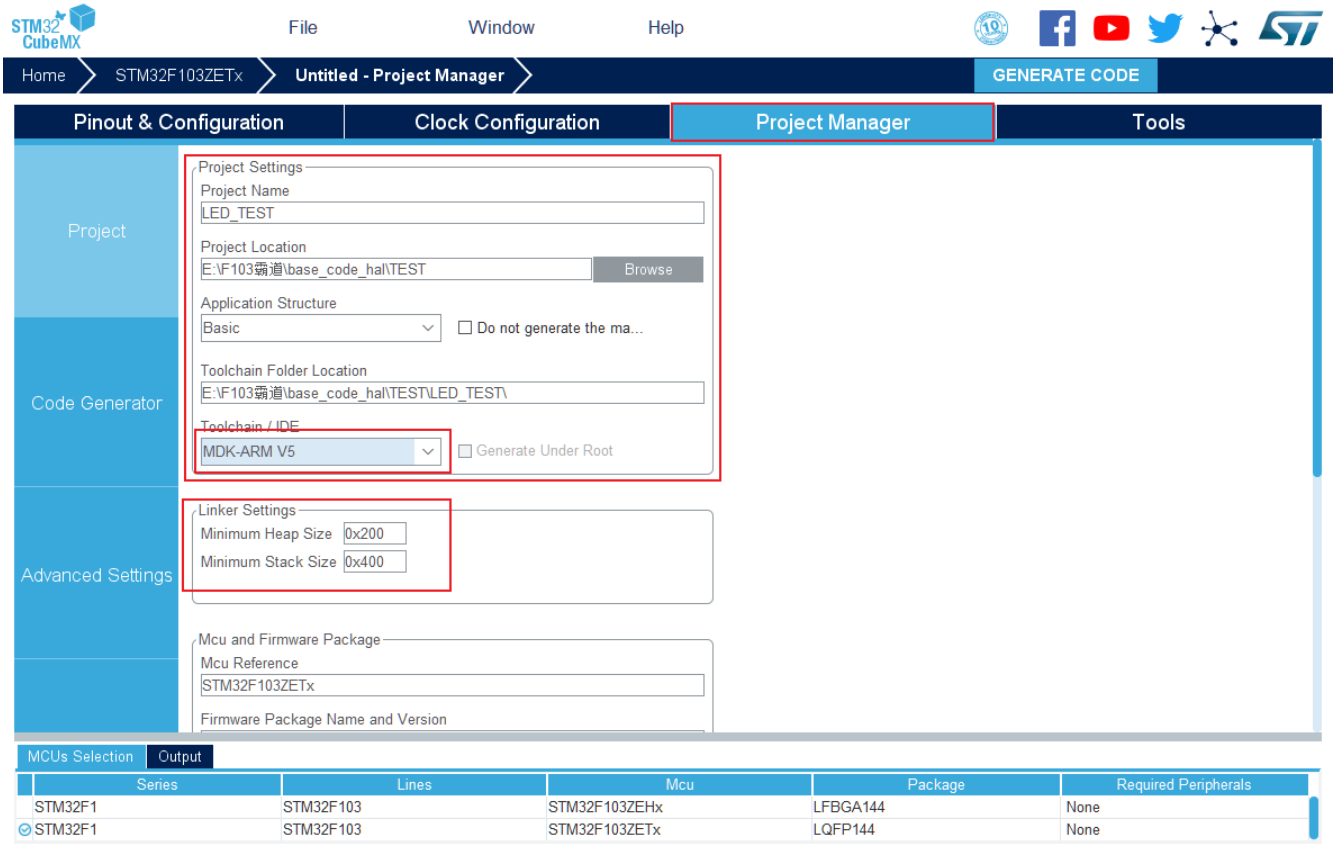


9.2.1.6 配置工程属性

为了防止出现，烧录以后仿真器无法连接的情况，我们在 Pinout 里将 SYS 里面的 Debug 设置成 Serial Wire, 这样问题得到解决。

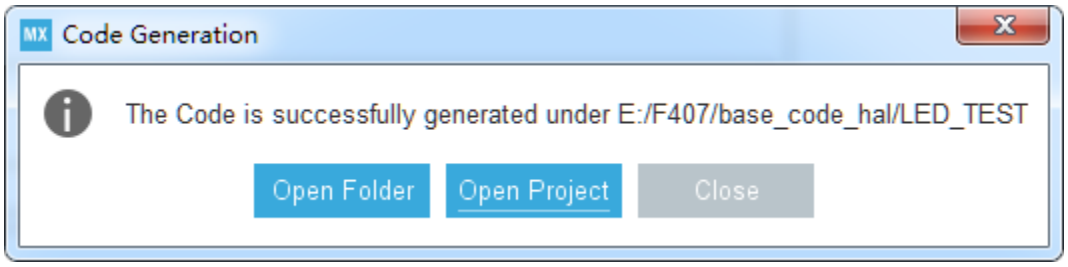


接着选择 Project Manager 选项，配置工程的名称，路径，使用的 IDE 工具，堆栈大小。注意不要使用中文路径和工程名称。



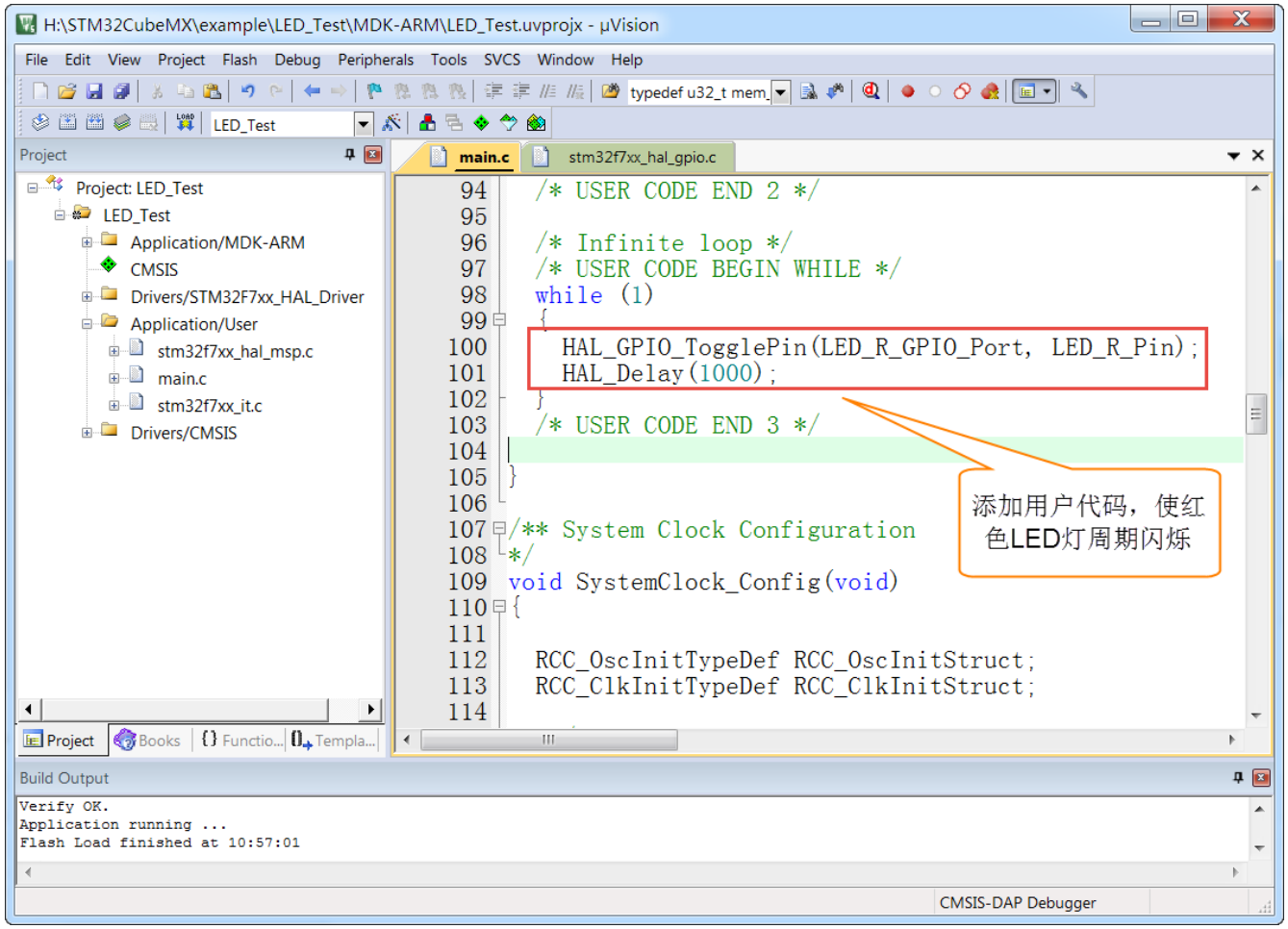
9.2.1.7 生成代码

点击 GENERATE CODE, 在设定的路径成功生成代码，选着打开工程。



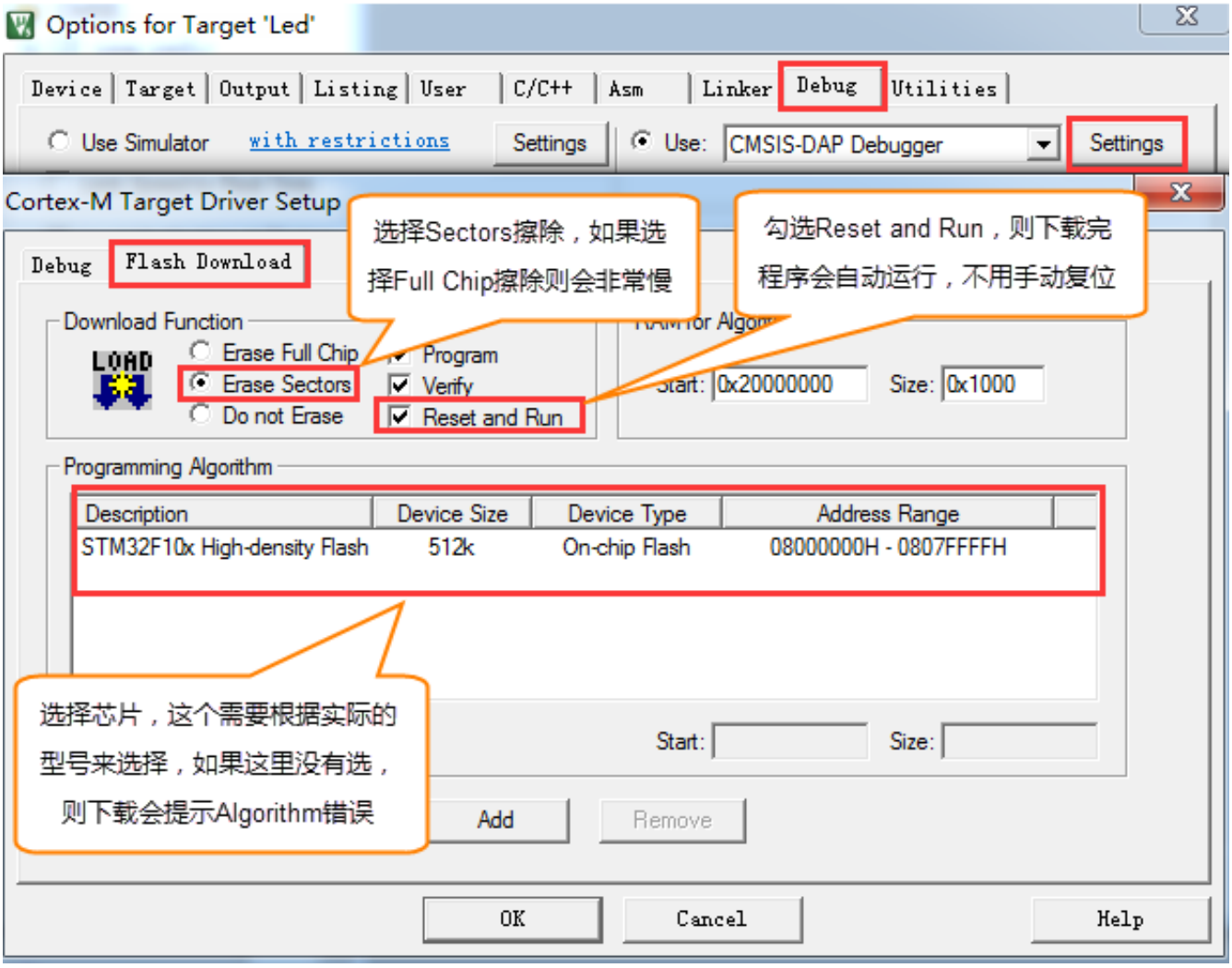
9.2.1.8 添加用户测试代码

打开工程后在 main 函数中的主循环插入用户代码，目的是让红色 LED 周期闪烁。



9.2.1.9 配置下载调试工具

配置下载工具为 CMSIS-DAP，程序下载完后复位并运行。



9.2.2 下载验证

把编译好的程序下载到开发板并复位，可看到板子上的红色灯会周期闪烁。

第 10 章　新建工程—库函数版

了解 STM32 的 HAL 库文件之后，我们就可以使用它来建立工程了，因为用库新建工程的步骤较多，我们一般是使用库建立一个空的工程，作为工程模板。以后直接复制一份工程模板，在它之上进行开发。

本章的“工程模板”范例可在配套资料中找到，自己新建工程模版时可参考该工程。

10.1 新建工程

本章内容所涉及的软件只供教学使用，不得用于商业用途。个人或公司因商业用途导致的法律责任，后果自负。

版本说明：MDK5.15 (即 Keil MDK 软件)，使用最新版本即可

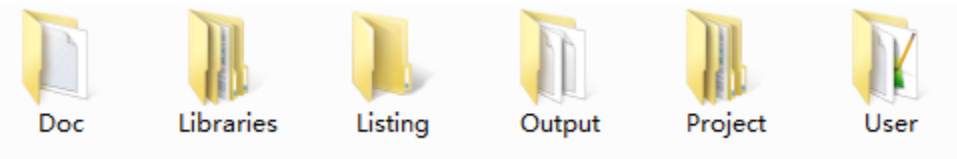
版本号可从 MDK 软件的菜单栏的“Help–>About uVision”选项中查询到。

10.1.1 新建本地工程文件夹

为了工程目录更加清晰，我们在本地电脑上新建一个“工程模板”文件夹，在它之下再新建 6 个文件夹，具体如下：

表 11-1 工程目录文件夹清单

名称	作用
Doc	用来存放程序说明的文件，由写程序的人添加
Libraries	存放的是库文件
Listing	存放编译器编译时候产生的 C/汇编/链接的列表清单
Output	存放编译产生的调试信息、hex 文件、预览信息、封装库等
Project	用来存放工程
User	用户编写的驱动文件



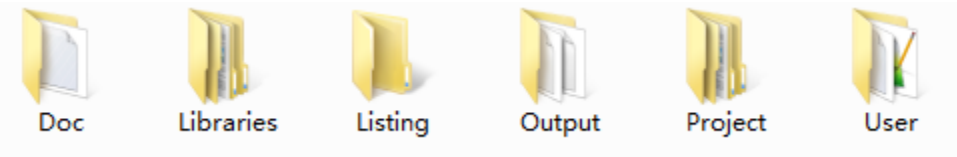
在本地新建好文件夹后，把准备好的库文件添加到相应的文件夹下：

表 11-2 工程目录文件夹内容清单

名称	作用
Doc	工程说明.txt
Libraries	CMSIS：里面放着跟 CM3 内核有关的库文件
	STM32F1xx_HAL_Driver：STM32 外设库文件
Listing	暂时为空
Output	暂时为空
Project	暂时为空
User	stm32f1xx_hal_conf.h：用来配置库的头文件
	stm32f1xx_it.h stm32f1xx_it.c：中断相关的函数都在这个文件编写，暂时为空
	main.c：main 函数文件

10.1.2 新建工程

打开 KEIL5，新建一个工程，工程名根据喜好命名，我这里取 Template（中文是模版的意思），保存在 ProjectRVMDK（uv5）文件夹下。

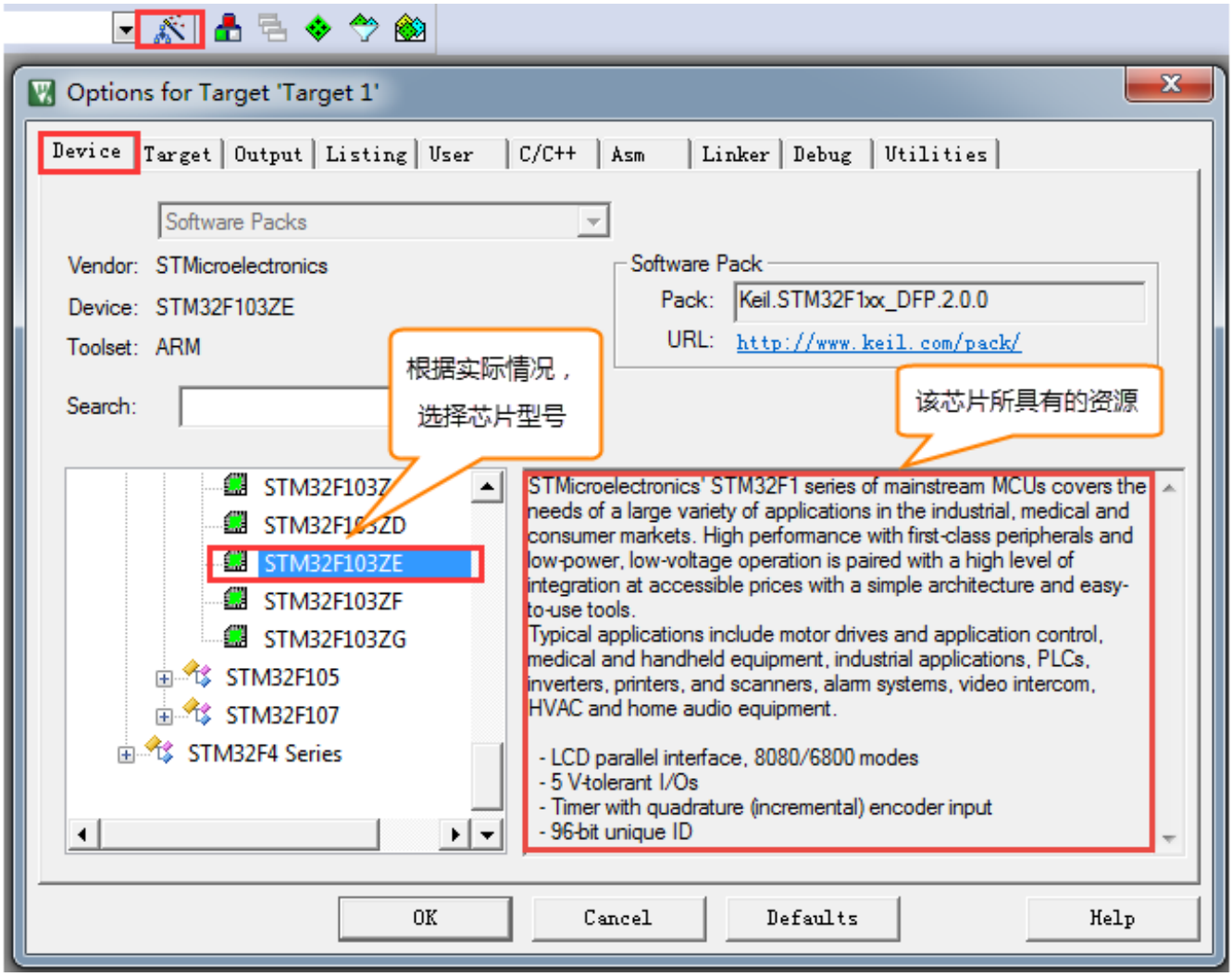


10.1.2.1 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，具体如下：

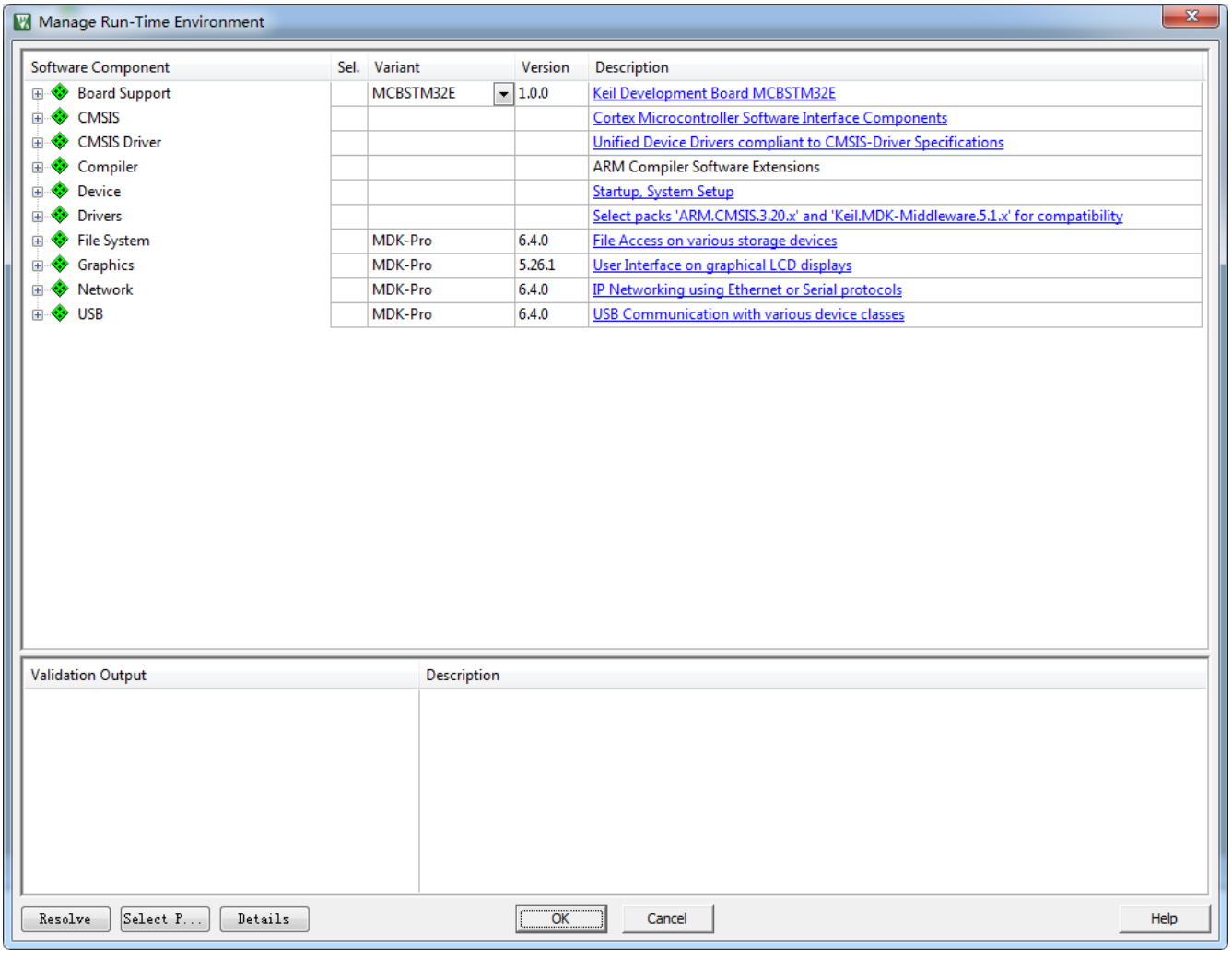
- F103-MINI，选 STM32F103RC 型号
- F103-指南者，选 STM32F103VE 型号
- F103-霸道，选 STM32F103ZE 型号
- F103-拂晓电机开发板，选 STM32F103VE 型号

如果这里没有出现你想要的 CPU 型号，或者一个型号都没有，那么肯定是你的 KEIL5 没有添加 device 库，KEIL5 不像 KEIL4 那样自带了很多 MCU 的型号，KEIL5 需要自己添加，关于如何添加请参考《如何安装 KEIL5》这一章。



10.1.2.2 在线添加库文件

等下我们手动本地添加库文件，这里我们点击关掉在线添加，因为 keil 的服务器在国外，在线添加会非常慢。

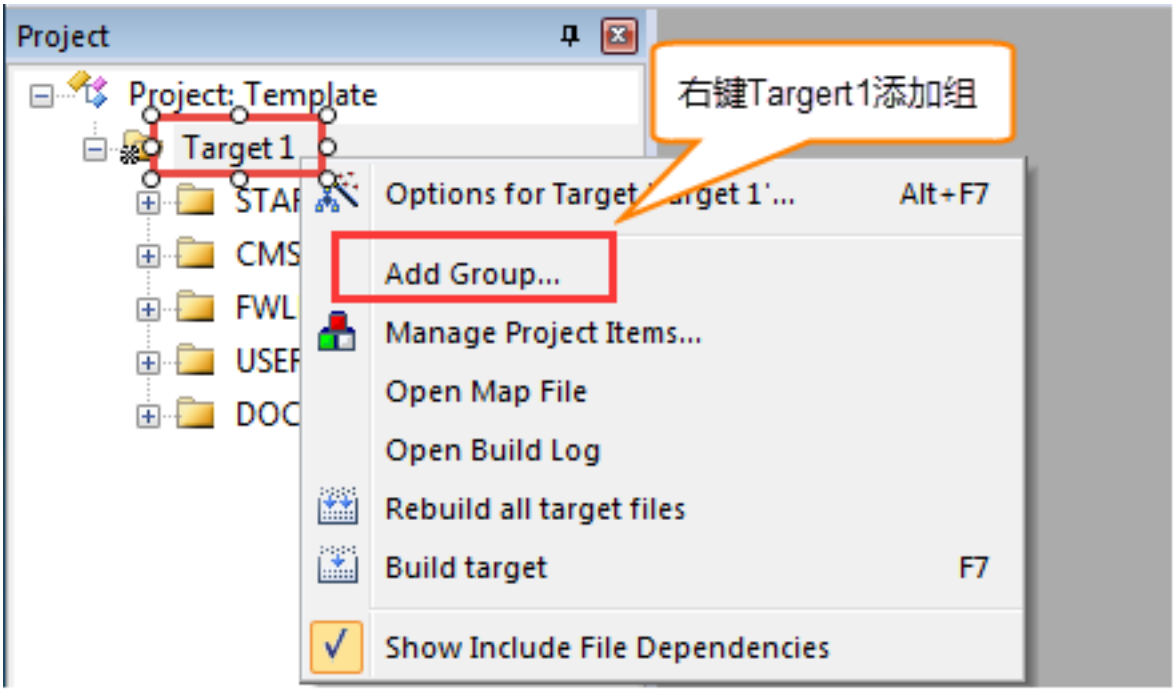


10.1.2.3 添加组文件夹

在新建的工程中添加 5 个组文件夹，用来存放各种不同的文件，文件从本地建好的工程文件夹下获取，双击组文件夹就会出现添加文件的路径，然后选择文件即可。

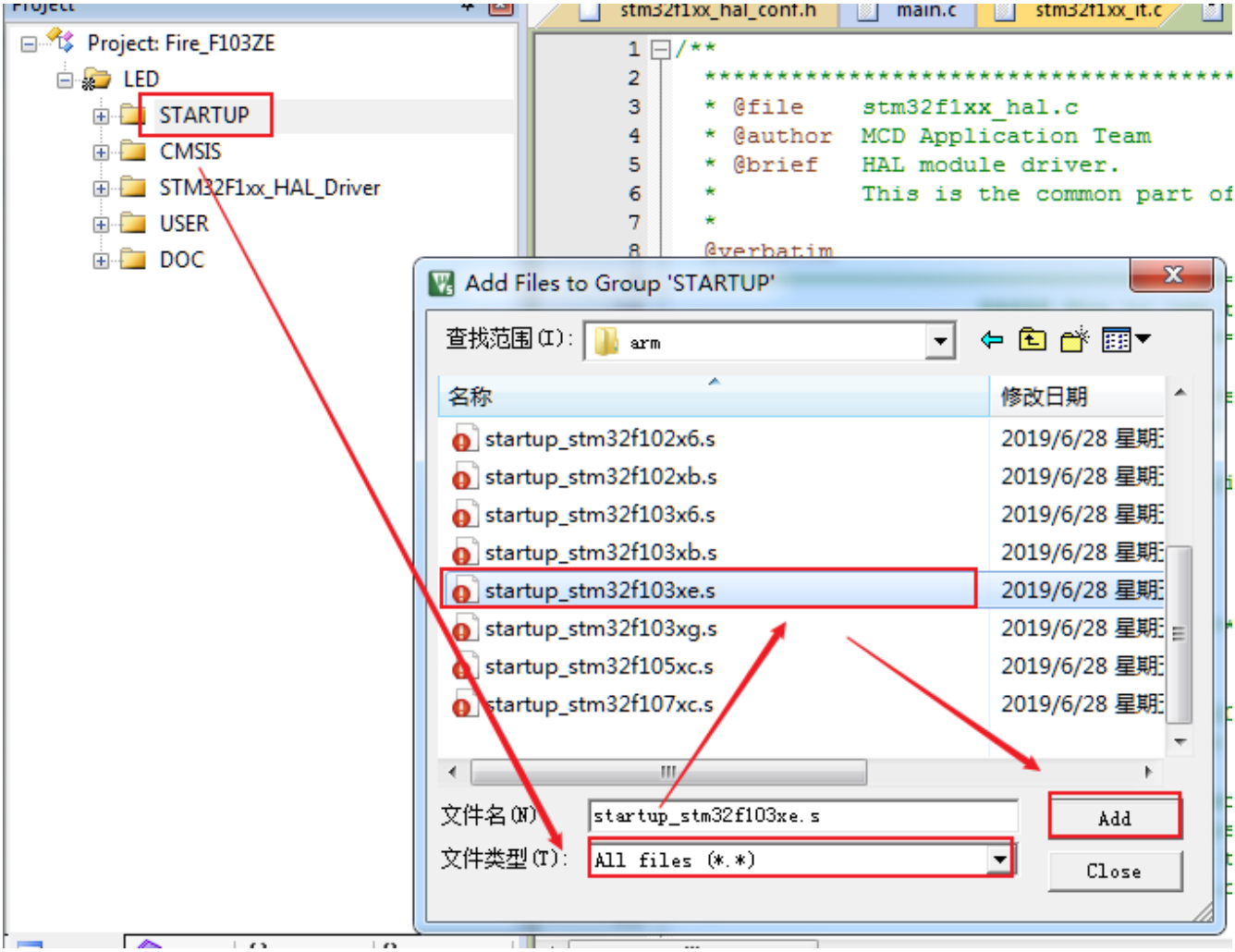
表 11-3 工程内组文件夹内容清单

名称	作用
STARTUP	存放汇编的启动文件：startup_STM32F103xe.s
STM32F1xx_HAL_Driver	与 STM32 外设相关的库文件
	stm32f1xx_hal.c
	stm32f1xx_hal_ppp.c（ppp 代表外设名称）
USER	用户编写的文件：
	main.c：main 函数文件，暂时为空
	stm32f1xx_it.c：跟中断有关的函数都放这个文件，暂时为空
DOC	工程说明.txt：程序说明文件，用于说明程序的功能和注意事项等



10.1.2.4 添加文件

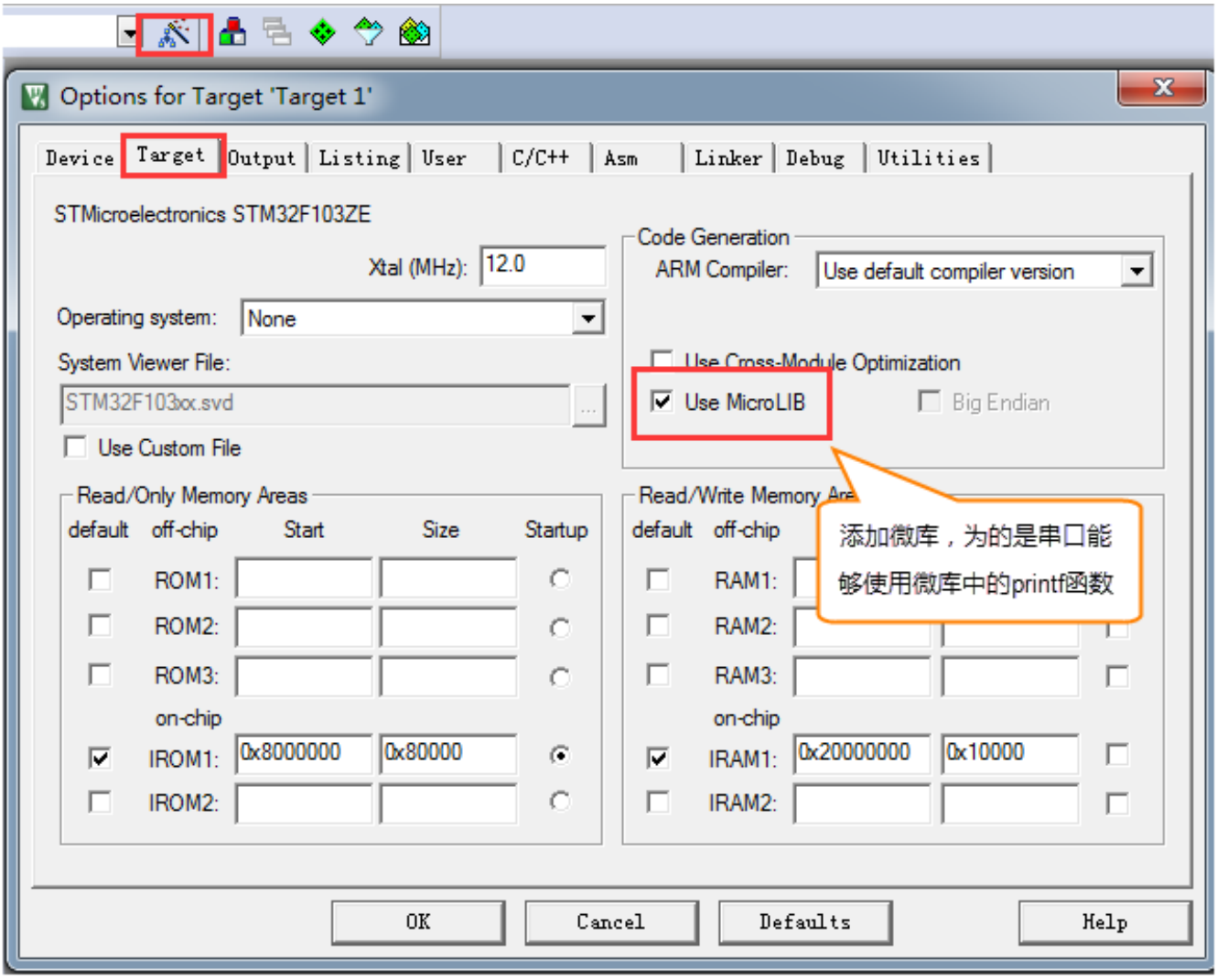
先把上面提到的文件从 STM32HAL 库中复制到工程模版对应文件夹的目录下，然后在新建的工程中添加这些文件，双击组文件夹就会出现添加文件的路径，然后选择文件即可。



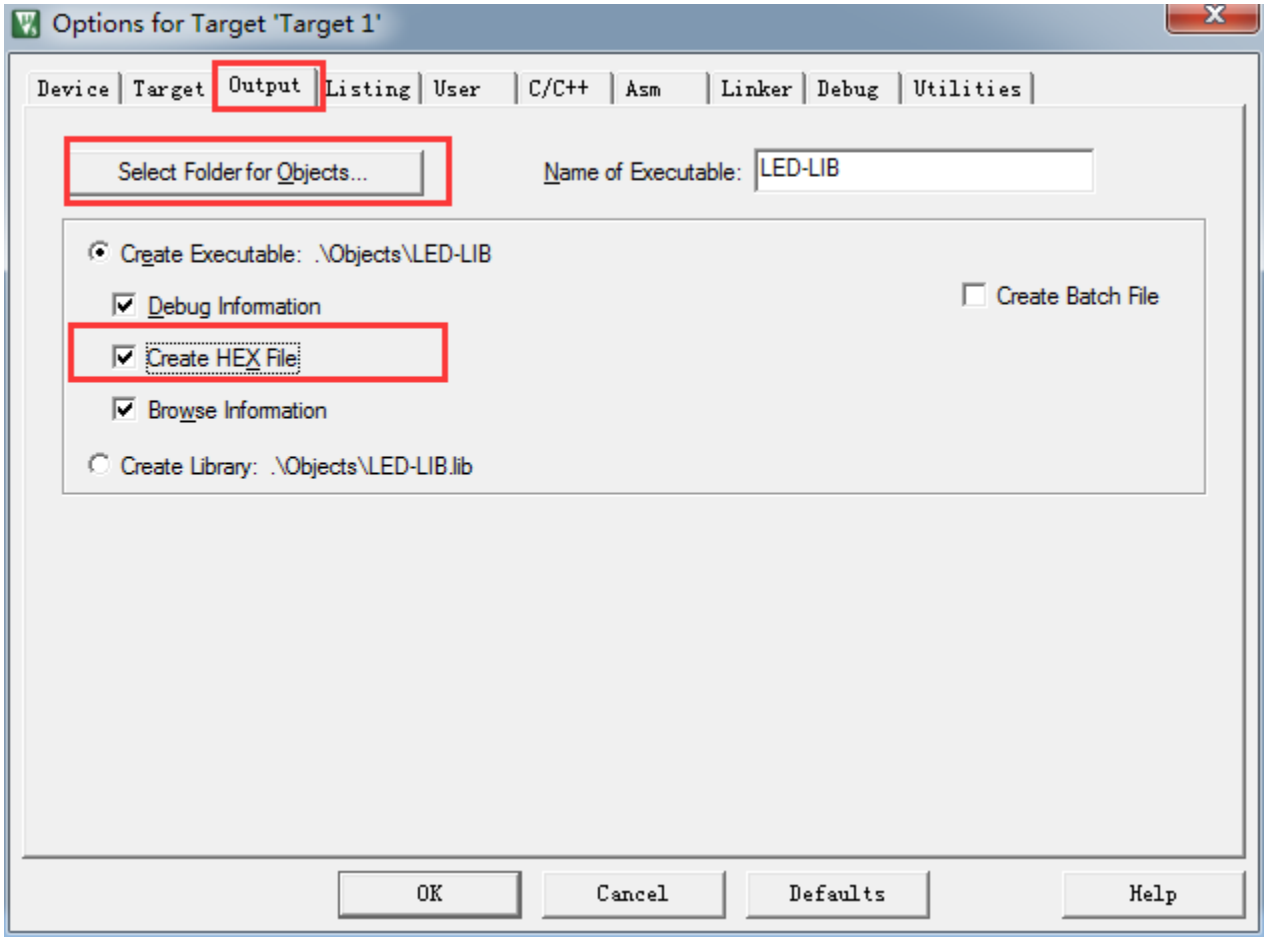
10.1.2.5 配置魔术棒选项卡

这一步的配置工作很重要，很多人串口用不了 printf 函数，编译有问题，下载有问题，都是这个步骤的配置出了错。

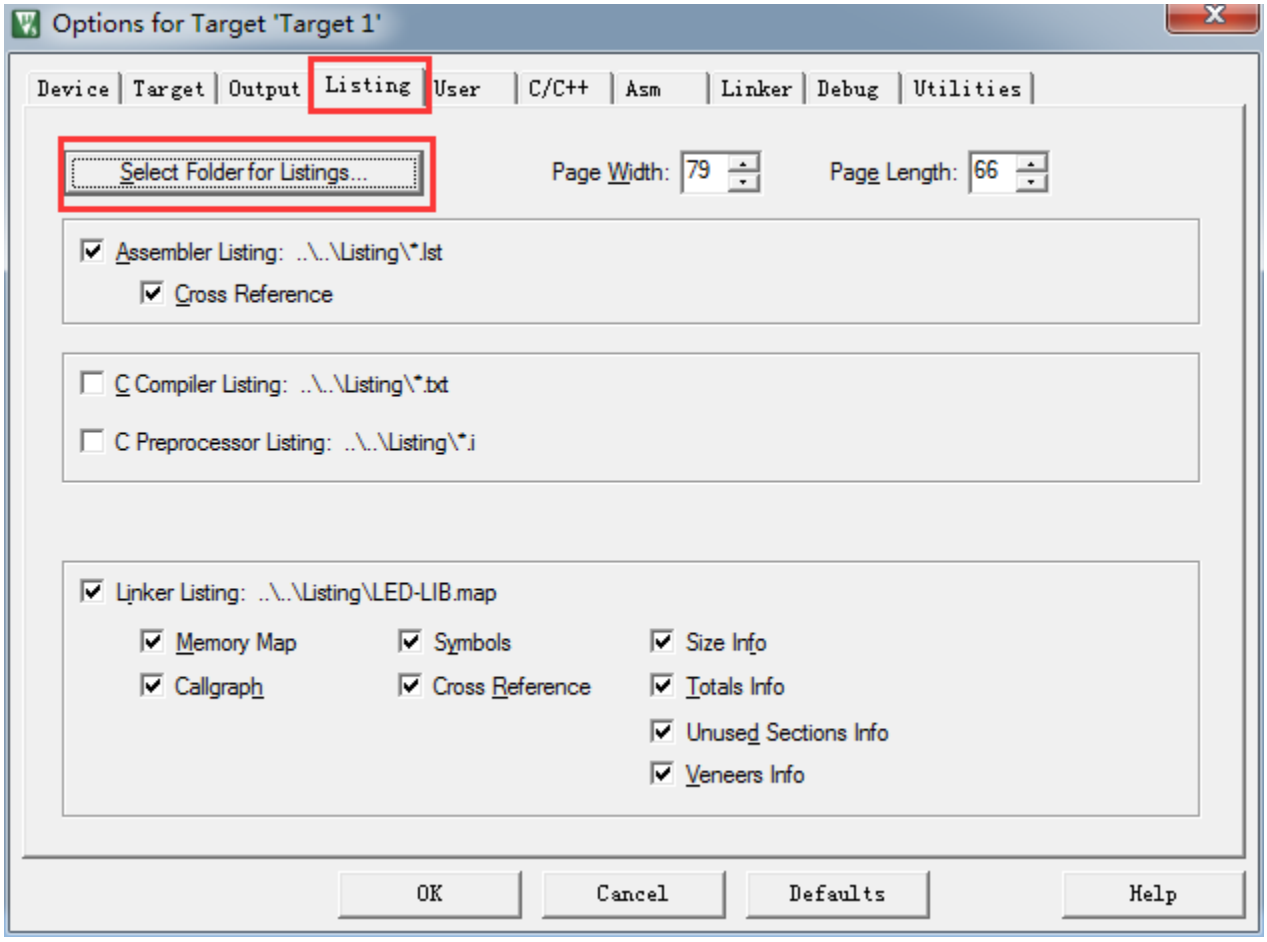
(1) Target 中选中微库 “Use MicroLib”，为的是在日后编写串口驱动的时候可以使用 printf 函数。



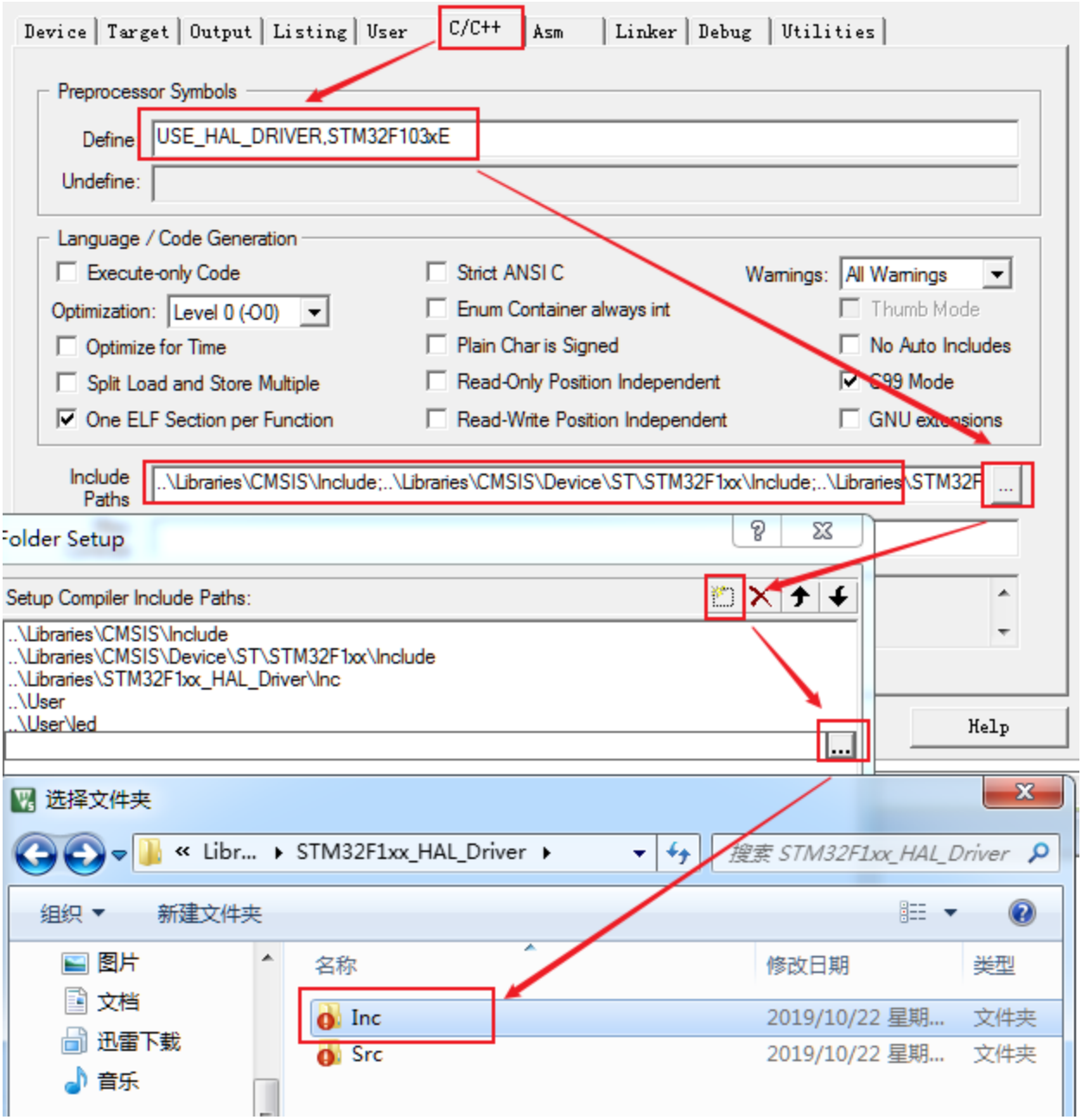
(2) 在 Output 选项卡中把输出文件夹定位到我们工程目录下的“output”文件夹，如果想在编译的过程中生成 hex 文件，那么那 Create HEX File 选项勾上。



(3) 在 Listing 选项卡中把输出文件夹定位到我们工程目录下的“Listing”文件夹。



(4) 在 C/C++ 选项卡中添加处理宏及编译器编译的时候查找的头文件路径。如果头文件路径添加有误，则编译的时候会报错找不到头文件。



在这个选项中添加宏，就相当于我们在文件中使用“#define”语句定义宏一样。在编译器中添加宏的好处就是，只要用了这个模版，就不用源文件中修改代码。

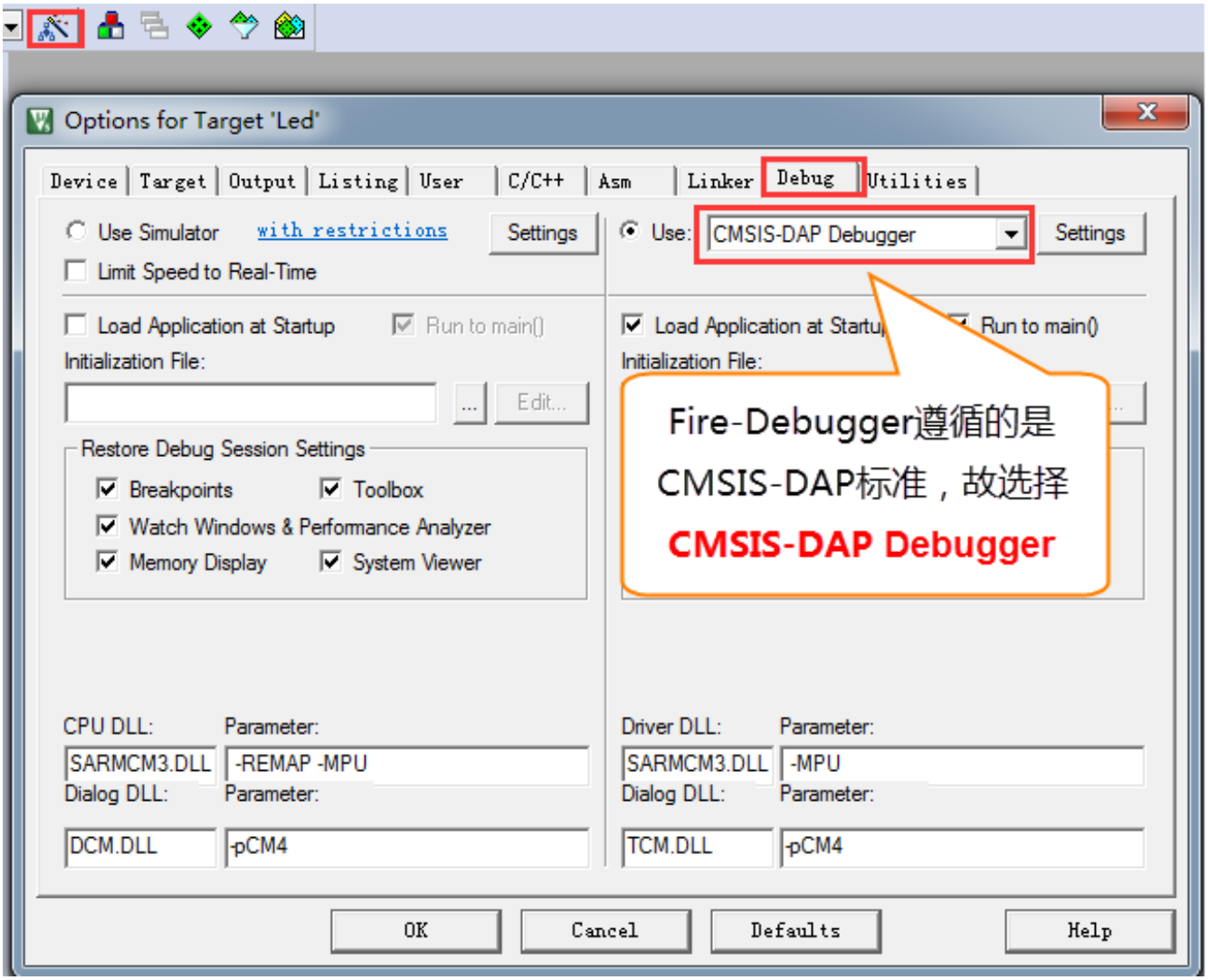
- STM32F103xE 宏：为了告诉 STM32HAL 库，我们使用的芯片类型是 STM32 型号是大容量的，使 STM32HAL 库根据我们选定的芯片型号来配置。

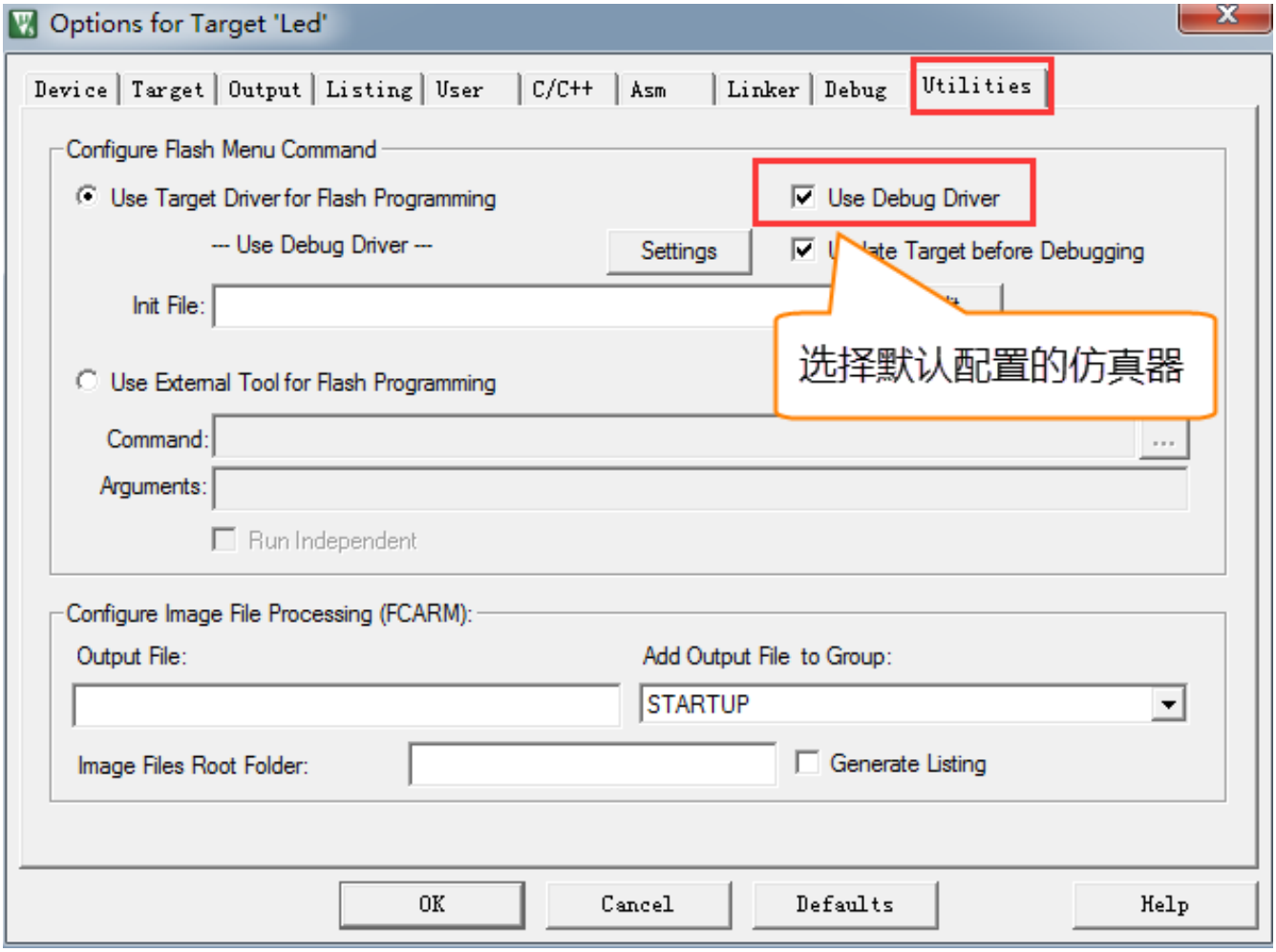
- USE_HAL_DRIVER 宏：为了让 stm32f1xx.h 包含 stm32f1xx_hal_conf.h 这个头文件。

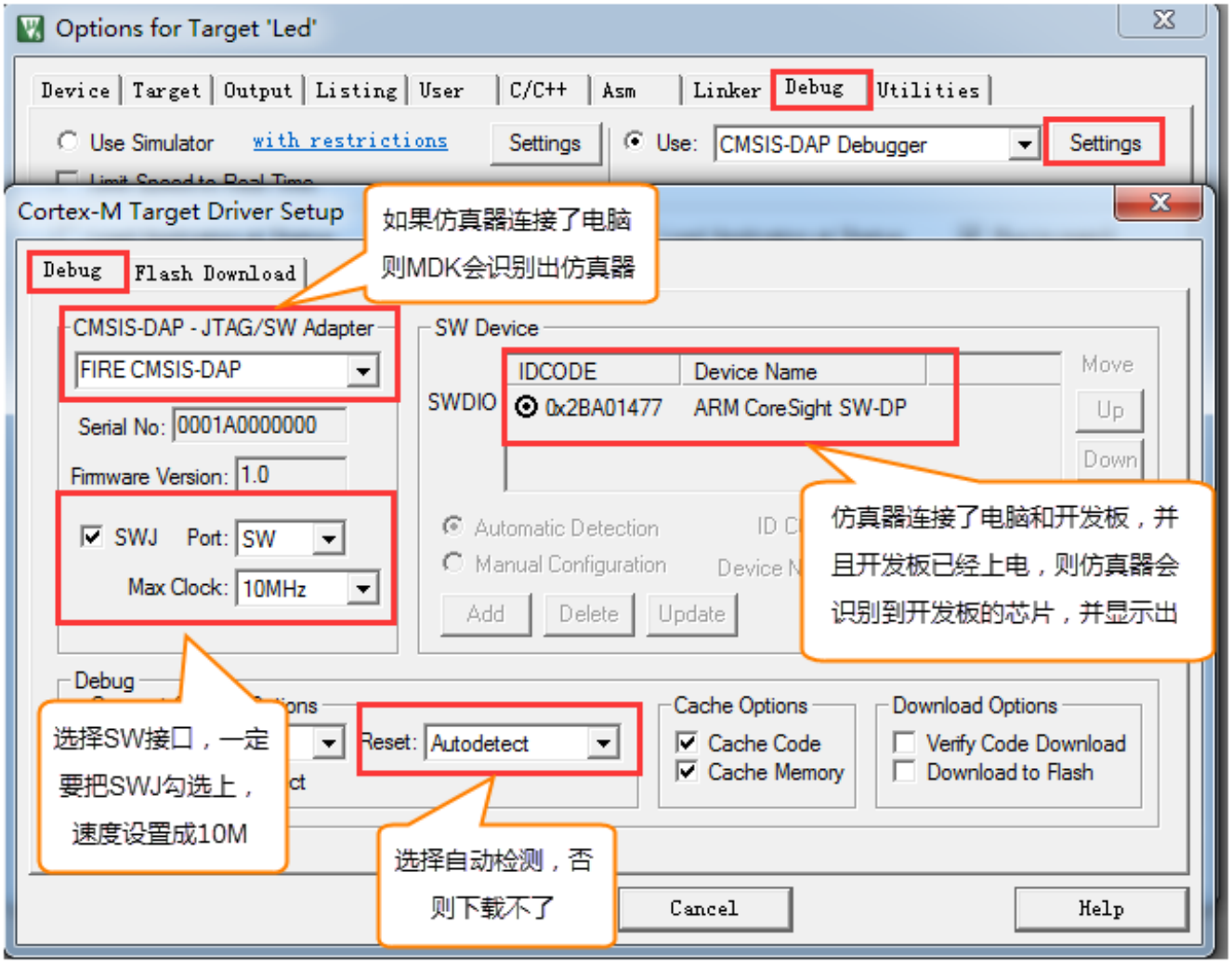
“Include Paths ”这里添加的是头文件的路径，如果编译的时候提示说找不到头文件，一般就是这里配置出了问题。你把头文件放到了哪个文件夹，就把该文件夹添加到这里即可。(请使用图中的方法用文件浏览器去添加路径，不要直接手打路径，容易出错)

10.1.2.6 仿真器配置

本书使用的仿真器是 Fire-Debugger，可下载和仿真程序。Fire-Debugger 支持 XP/WIN7/WIN8/WIN10 这几个操作系统，无需安装驱动，免驱，使用非常方便，具体配置见如下图。

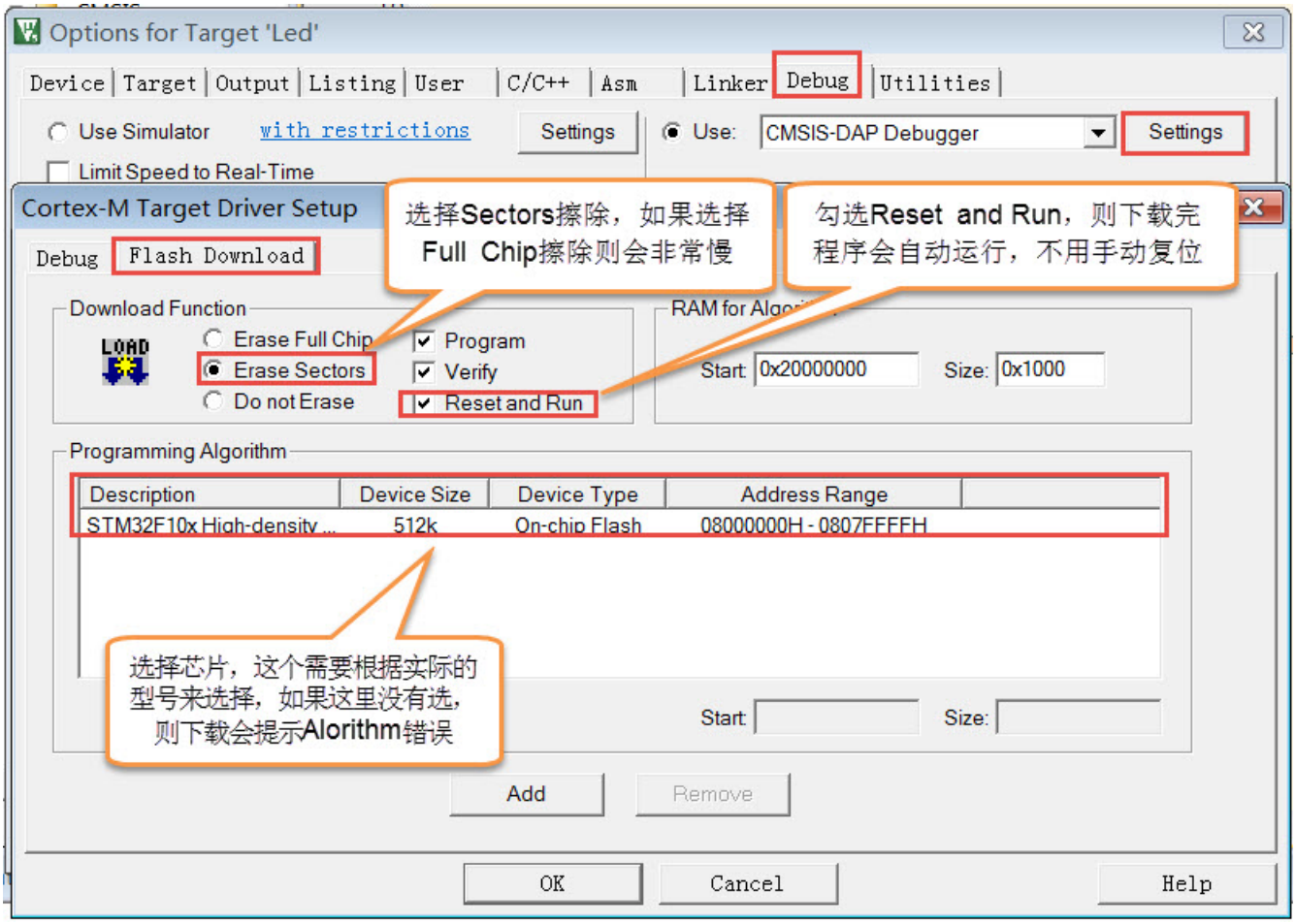

Options for Target 'Led'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Use Simulator
with restrictions
Settings
Limit Speed to Real-Time
Use:
CMSIS-DAP Debugger
Settings
Load Application at Startup
Run to main()
Initialization File:
Edit...
Restore Debug Session Settings
Breakpoints
Toolbox
Watch Windows & Performance Analyzer
Memory Display
System Viewer
Fire-Debugger遵循的是
CMSIS-DAP标准，故选择
CMSIS-DAP Debugger
CPU DLL:
Parameter:
SARMCM3.DLL
-REMAP -MPU
Dialog DLL:
DCM.DLL
-pCM4
Driver DLL:
-MPU
TCM.DLL
OK
Cancel
Defaults
Help


Options for Target 'Led'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Configure Flash Menu Command
Use Target Driver for Flash Programming
Use Debug Driver
--- Use Debug Driver ---
Settings
Update Target before Debugging
Init File:
选择默认配置的仿真器
Use External Tool for Flash Programming
Command:
Arguments:
Run Independent
Configure Image File Processing (FCARM):
Output File:
Add Output File to Group:
STARTUP
Image Files Root Folder:
Generate Listing
OK
Cancel
Defaults
Help



10.1.2.7 选择 CPU 型号

这一步的配置也不是配置一次之后完事，常常会因为各种原因需要重新选择，当你下载的时候，提示说找不到 Device 的时候，请确保该配置是否正确。有时候下载程序之后，不会自动运行，要手动复位的时候，也回来看看这里的“Reset and Run”配置是否失效。指南者/霸道/拂晓开发板上的 STM32 芯片的 FLASH 大小是 512kByte，所以这里选择 512k 的容量，F103-MINI 开发板的要选 256K，如果使用的是其他型号的，要根据实际情况选择。



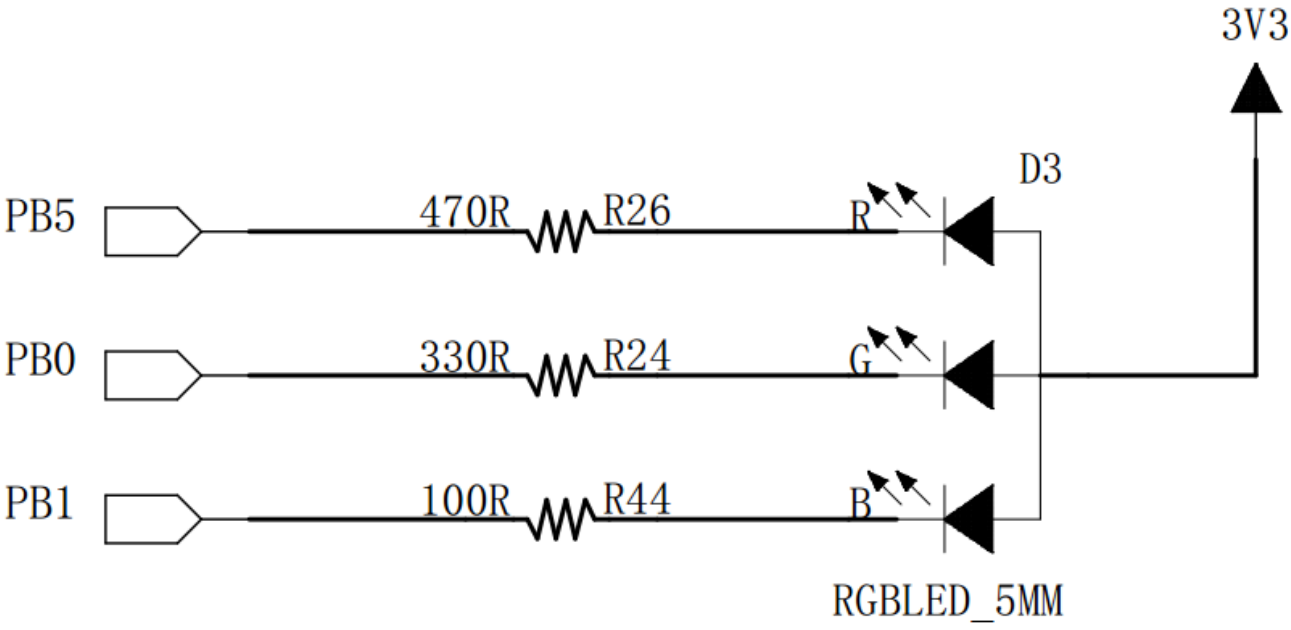
一个新的工程模版新建完毕。

第 11 章　GPIO 输出—使用固件库点亮 LED

利用固件库建立好的工程模板，就可以方便地使用 STM32HAL 库编写应用程序了，可以说从这一章我们才开始迈入 STM32F1 开发的大门。LED 灯的控制使用到 GPIO 外设的基本输出功能，本章中不再赘述 GPIO 外设的概念和功能框图的讲解，而是重点讲解如何使用 STM32HAL 库编程。

11.1 硬件设计

我们的 F103-指南者和 F103-霸道开发板中 STM32 芯片与 LED 灯的连接见图 *12_1*，这是一个 RGB 灯，里面由红绿蓝（Red, Green, Blue）三个小灯构成，使用 PWM 控制时可以混合成 256*256*256 种不同的颜色。



而我们的 F103-MINI 开发板和 F103 拂晓电机开发板接的是单色的 LED 灯，其中 STM32 芯片与

LED 灯的连接分别如图 *12_2* 和图 *12_3* 所示。

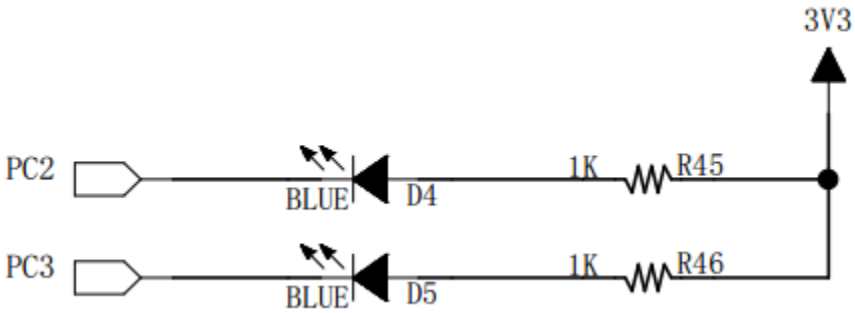


图 12-2 F103-MINI 开发板的 LED 硬件原理图

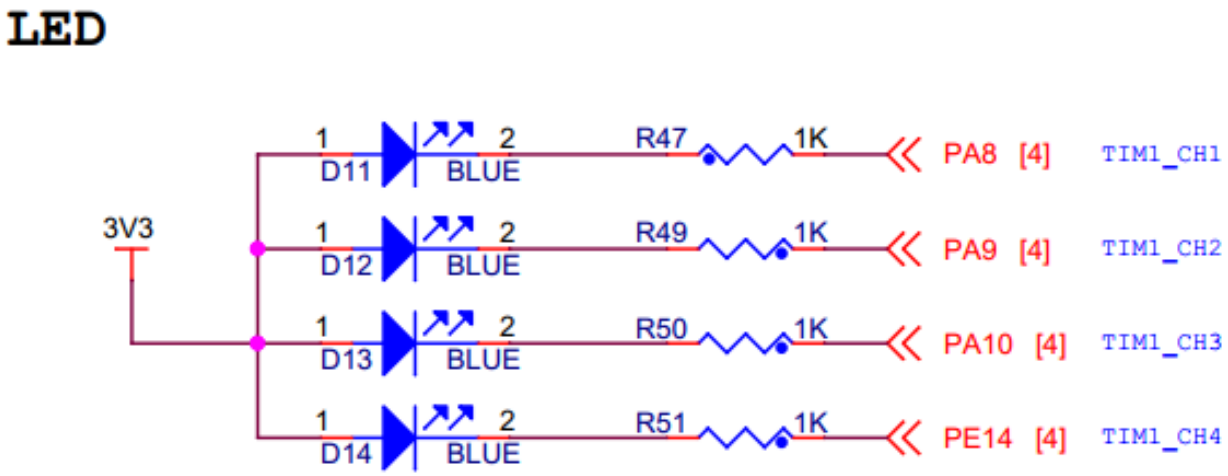


图 12-3 F103 拂晓电机开发板的 LED 硬件原理图

以上所有的这些 LED 灯的阴极都是连接到 STM32 的 GPIO 引脚，阳极接到 3.3V 电源，同时每个灯都分别接了一个限流电阻。只要我们控制 GPIO 引脚的电平输出状态，即可控制 LED 灯的亮灭。若您使用的实验板 LED 灯的连接方式或引脚不一样，只需根据我们的工程修改引脚即可，程序的控制原理相同。

11.2 软件设计

这里只讲解核心部分的代码，有些变量的设置，头文件的包含等可能不会涉及到，完整的代码请参考本章配套的工程。

为了使工程更加有条理，我们把 LED 灯控制相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_led.c”及“bsp_led.h”文件，其中的“bsp”即 Board Support Packet 的缩写 (板级支持包)，这些文件也可根据您的喜好命名，这些文件不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

11.2.1 编程要点

1. 使能 GPIO 端口时钟；
2. 初始化 GPIO 目标引脚为推挽输出模式；
3. 编写简单测试程序，控制 GPIO 引脚输出高、低电平。

11.2.2 代码分析

11.2.2.1 LED 灯引脚宏定义

在编写应用程序的过程中，要考虑更改硬件环境的情况，例如 LED 灯的控制引脚与当前的不一样，我们希望程序只需要做最小的修改即可在新的环境正常运行。这个时候一般把硬件相关的部分使用宏来封装，若更改了硬件环境，只修改这些硬件相关的宏即可，这些定义一般存储在头文件，即本例子中的“bsp_led.h”文件中，见*代码清单 12_1*、*代码清单 12_2*、*代码清单 12_3*。

列表 1: 代码清单 12-1：F103-指南者和 F103-霸道开发板
LED 控制引脚相关的宏

```
//引脚定义
/*******************************************************/
```

(下页继续)

(续上页)

```
//R 红色灯
#define LED1_PIN                  GPIO_PIN_5
#define LED1_GPIO_PORT            GPIOB
#define LED1_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()

//G 绿色灯
#define LED2_PIN                  GPIO_PIN_0
#define LED2_GPIO_PORT            GPIOB
#define LED2_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()

//B 蓝色灯
#define LED3_PIN                  GPIO_PIN_1
#define LED3_GPIO_PORT            GPIOB
#define LED3_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
```

列表 2: 代码清单 12-2：F103-MINI 开发板 LED 控制引脚相关的宏

```
//引脚定义
/*******************************************************/
//R 红色灯
#define LED1_PIN                  GPIO_PIN_2
#define LED1_GPIO_PORT            GPIOC
#define LED1_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOC_CLK_ENABLE()

//G 绿色灯
#define LED2_PIN                  GPIO_PIN_3
#define LED2_GPIO_PORT            GPIOC
#define LED2_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOC_CLK_ENABLE()
```

列表 3: 代码清单 12-3：F103 拂晓电机开发板 LED 控制引脚相关的宏

```
//引脚定义
/*******************************************************/
//LED1
#define LED1_PIN                  GPIO_PIN_8
#define LED1_GPIO_PORT            GPIOA
#define LED1_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOA_CLK_ENABLE()

//LED2
#define LED2_PIN                  GPIO_PIN_9
#define LED2_GPIO_PORT            GPIOA
#define LED2_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOA_CLK_ENABLE()

//LED3
#define LED3_PIN                  GPIO_PIN_10
#define LED3_GPIO_PORT            GPIOA
#define LED3_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOA_CLK_ENABLE()

//LED4
#define LED4_PIN                  GPIO_PIN_14
#define LED4_GPIO_PORT            GPIOE
#define LED4_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOE_CLK_ENABLE()
```

以上代码分别把控制 LED 灯的 GPIO 端口、GPIO 引脚号以及 GPIO 端口时钟封装起来了。在实际控制的时候我们就直接用这些宏，以达到应用代码跟硬件无关的效果。

其中的 GPIO 时钟宏“__GPIOF_CLK_ENABLE()”是 STM32HAL 库定义的 GPIO 端口时钟相关的宏，它的作用与“GPIO_PIN_x”这类宏类似，是用于指示寄存器位的，方便库函数使用。它们指示 GPIOF 时钟，下面初始化 GPIO 时钟的时候可以看到它的用法。

11.2.2.2 控制 LED 灯亮灭状态的宏定义

为了方便控制 LED 灯，我们把 LED 灯常用的亮、灭及状态反转的控制也直接定义成宏，见代码清单 *12_4*、代码清单 *12_5*、代码清单 *12_6*。

列表 4: 代码清单 12-4：F103-指南者和 F103-霸道开发板控制 LED 亮灭的宏

```
/** 控制 LED 灯亮灭的宏，
* LED 低电平亮，设置 ON=0，OFF=1
* 若 LED 高电平亮，把宏设置成 ON=1 ，OFF=0 即可
*/
#define ON  GPIO_PIN_RESET
#define OFF GPIO_PIN_SET

/* 带参宏，可以像内联函数一样使用 */
#define LED1(a) HAL_GPIO_WritePin(LED1_GPIO_PORT,LED1_PIN,a)
#define LED2(a) HAL_GPIO_WritePin(LED2_GPIO_PORT,LED2_PIN,a)
#define LED3(a) HAL_GPIO_WritePin(LED2_GPIO_PORT,LED3_PIN,a)

/* 直接操作寄存器的方法控制 IO */
#define digitalHi(p,i)       {p->BSRR=i;}         //设置为高电平
#define digitalLo(p,i)       {p->BSRR=(uint32_t)i << 16;}
                  //输出低电平
#define digitalToggle(p,i)     {p->ODR ^=i;}      //输出反转状态

/* 定义控制 IO 的宏 */
#define LED1_TOGGLE   digitalToggle(LED1_GPIO_PORT,LED1_PIN)
#define LED1_OFF     digitalHi(LED1_GPIO_PORT,LED1_PIN)
#define LED1_ON      digitalLo(LED1_GPIO_PORT,LED1_PIN)

#define LED2_TOGGLE   digitalToggle(LED2_GPIO_PORT,LED2_PIN)
#define LED2_OFF     digitalHi(LED2_GPIO_PORT,LED2_PIN)
```

(下页继续)

(续上页)

```
#define LED2_ON     digitalLo(LED2_GPIO_PORT,LED2_PIN)

#define LED3_TOGGLE   digitalToggle(LED3_GPIO_PORT,LED3_PIN)
#define LED3_OFF    digitalHi(LED3_GPIO_PORT,LED3_PIN)
#define LED3_ON     digitalLo(LED3_GPIO_PORT,LED3_PIN)

/* 基本混色，后面高级用法使用 PWM 可混出全彩颜色，且效果更好 */

//红
#define LED_RED   \
        LED1_ON;\
        LED2_OFF\
        LED3_OFF
//绿
#define LED_GREEN    \
        LED1_OFF;\
        LED2_ON\
        LED3_OFF
//蓝
#define LED_BLUE   \
        LED1_OFF;\
        LED2_OFF\
        LED3_ON
//黄 (红 + 绿)
#define LED_YELLOW   \
        LED1_ON;\
        LED2_ON\
        LED3_OFF
//紫 (红 + 蓝)
#define LED_PURPLE   \
        LED1_ON;\
        LED2_OFF\
```

(下页继续)

(续上页)

```
        LED3_ON
//青 (绿 + 蓝)
#define LED_CYAN \
        LED1_OFF;\
        LED2_ON\
        LED3_ON
//白 (红 + 绿 + 蓝)
#define LED_WHITE \
        LED1_ON;\
        LED2_ON\
        LED3_ON
//黑 (全部关闭)
#define LED_RGBOFF  \
        LED1_OFF;\
        LED2_OFF\
        LED3_OFF
```

列表 5: 代码清单 12-5：F103-MINI 开发板控制 LED 亮灭的宏

```
1  /** 控制 LED 灯亮灭的宏,
2  * LED 低电平亮, 设置 ON=0, OFF=1
3  * 若 LED 高电平亮, 把宏设置成 ON=1 , OFF=0 即可
4  */
5  #define ON  GPIO_PIN_RESET
6  #define OFF GPIO_PIN_SET
7
8  /* 带参宏, 可以像内联函数一样使用 */
9  #define LED1(a) HAL_GPIO_WritePin(LED1_GPIO_PORT,LED1_PIN,a)
10 #define LED2(a) HAL_GPIO_WritePin(LED2_GPIO_PORT,LED2_PIN,a)
11
12
```

(下页继续)

(续上页)

```
/* 直接操作寄存器的方法控制 IO */
#define digitalHi(p,i)       {p->BSRR=i;}         //设置为高电平
#define digitalLo(p,i)       {p->BSRR=(uint32_t)i << 16;}         //输出低电平
#define digitalToggle(p,i)   {p->ODR ^=i;}      //输出反转状态

/* 定义控制 IO 的宏 */
#define LED1_TOGGLE    digitalToggle(LED1_GPIO_PORT,LED1_PIN)
#define LED1_OFF       digitalHi(LED1_GPIO_PORT,LED1_PIN)
#define LED1_ON        digitalLo(LED1_GPIO_PORT,LED1_PIN)

#define LED2_TOGGLE    digitalToggle(LED2_GPIO_PORT,LED2_PIN)
#define LED2_OFF       digitalHi(LED2_GPIO_PORT,LED2_PIN)
#define LED2_ON        digitalLo(LED2_GPIO_PORT,LED2_PIN)
```

列表 6: 代码清单 12-6：F103 拂晓电机开发板控制 LED 亮灭的宏

```
/** 控制 LED 灯亮灭的宏，
* LED 低电平亮，设置 ON=0，OFF=1
* 若 LED 高电平亮，把宏设置成 ON=1 ，OFF=0 即可
*/
#define ON  GPIO_PIN_RESET
#define OFF GPIO_PIN_SET

/* 带参宏，可以像内联函数一样使用 */
#define LED1(a)      HAL_GPIO_WritePin(LED1_GPIO_PORT,LED1_PIN,a)

#define LED2(a)      HAL_GPIO_WritePin(LED2_GPIO_PORT,LED2_PIN,a)

#define LED3(a)      HAL_GPIO_WritePin(LED3_GPIO_PORT,LED3_PIN,a)

#define LED4(a)      HAL_GPIO_WritePin(LED4_GPIO_PORT,LED4_PIN,a)
```

(下页继续)

(续上页)

```
/* 直接操作寄存器的方法控制 IO */
#define     digitalHi(p,i)                  {p->BSRR=i;}                ␣
↪   //设置为高电平
#define digitalLo(p,i)                      {p->BSRR=(uint32_t)i << 16;} ␣
↪                          //输出低电平
#define digitalToggle(p,i)          {p->ODR ^=i;}                   //输出反
转状态


/* 定义控制 IO 的宏 */
#define LED1_TOGGLE         digitalToggle(LED1_GPIO_PORT,LED1_PIN)
#define LED1_OFF            digitalHi(LED1_GPIO_PORT,LED1_PIN)
#define LED1_ON                     digitalLo(LED1_GPIO_PORT,LED1_PIN)

#define LED2_TOGGLE         digitalToggle(LED2_GPIO_PORT,LED2_PIN)
#define LED2_OFF            digitalHi(LED2_GPIO_PORT,LED2_PIN)
#define LED2_ON                     digitalLo(LED2_GPIO_PORT,LED2_PIN)

#define LED3_TOGGLE         digitalToggle(LED3_GPIO_PORT,LED3_PIN)
#define LED3_OFF            digitalHi(LED3_GPIO_PORT,LED3_PIN)
#define LED3_ON                     digitalLo(LED3_GPIO_PORT,LED3_PIN)

#define LED4_TOGGLE         digitalToggle(LED4_GPIO_PORT,LED4_PIN)
#define LED4_OFF            digitalHi(LED4_GPIO_PORT,LED4_PIN)
#define LED4_ON                     digitalLo(LED4_GPIO_PORT,LED4_PIN)
```

这部分宏控制 LED 亮灭的操作是直接向 BSRR 寄存器写入控制指令来实现的，对 BSRR 低 16 位写 1 输出高电平，对 BSRR 高 16 位写 1 输出低电平，对 ODR 寄存器某位进行异或操作可反转位的状态。

对于 F103-指南者和 F103-霸道开发板上的 RGB 彩灯可以实现混色，如最后一段代码我们控制红

灯和绿灯亮而蓝灯灭，可混出黄色效果。

代码中的“\”是 C 语言中的续行符语法，表示续行符的下一行与续行符所在的代码是同一行。代码中因为宏定义关键字“#define”只是对当前行有效，所以我们使用续行符来连接起来，以下的代码是等效的：

```
#define LED_YELLOW LED1_ON; LED2_ON; LED3_OFF
```

应用续行符的时候要注意，在“\”后面不能有任何字符 (包括注释、空格)，只能直接回车。

11.2.2.3 LED GPIO 初始化函数

利用上面的宏，编写 LED 灯的初始化函数，见*代码清单 12_7*、*代码清单 12_8*、*代码清单 12_9*。

列表 7: 代码清单 12-7：F103-指南者和 F103-霸道开发板的 LED GPIO 初始化函数

```
void LED_GPIO_Config(void)
{
    /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
    GPIO_InitTypeDef  GPIO_InitStruct;
    /* 开启 LED 相关的 GPIO 外设时钟 */
    LED1_GPIO_CLK_ENABLE();
    LED2_GPIO_CLK_ENABLE();
    LED3_GPIO_CLK_ENABLE();
    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED1_PIN;
    /* 设置引脚的输出类型为推挽输出 */
    GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
    /* 设置引脚为上拉模式 */
    GPIO_InitStruct.Pull  = GPIO_PULLUP;
    /* 设置引脚速率为高速 */
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    /* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
```

(下页继续)

(续上页)

```
    HAL_GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStruct);
    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED2_PIN;
    HAL_GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStruct);
    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED3_PIN;
    HAL_GPIO_Init(LED3_GPIO_PORT, &GPIO_InitStruct);
    /* 关闭 RGB 灯 */
    LED_RGBOFF;
}
```

列表 8: 代码清单 12-8：F103-MINI 开发板的 LED GPIO 初始化函数

```
void LED_GPIO_Config(void)
{
    /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
    GPIO_InitTypeDef  GPIO_InitStruct;
    /* 开启 LED 相关的 GPIO 外设时钟 */
    LED1_GPIO_CLK_ENABLE();
    LED2_GPIO_CLK_ENABLE();

    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED1_PIN;
    /* 设置引脚的输出类型为推挽输出 */
    GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
    /* 设置引脚为上拉模式 */
    GPIO_InitStruct.Pull  = GPIO_PULLUP;
    /* 设置引脚速率为高速 */
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    /* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
    HAL_GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStruct);
```

(下页继续)

(续上页)

```
    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED2_PIN;
    HAL_GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStruct);
    /* 关闭 RGB 灯 */
    LED_RGBOFF;
}
```

列表 9: 代码清单 12-9：F103 拂晓电机开发板的 LED GPIO 初始化函数

```
void LED_GPIO_Config(void)
 {
     /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
     GPIO_InitTypeDef  GPIO_InitStruct;

     /* 开启 LED 相关的 GPIO 外设时钟 */
     LED1_GPIO_CLK_ENABLE();
     LED2_GPIO_CLK_ENABLE();
     LED3_GPIO_CLK_ENABLE();
     LED4_GPIO_CLK_ENABLE();

     /* 选择要控制的 GPIO 引脚 */
     GPIO_InitStruct.Pin = LED1_PIN;

     /* 设置引脚的输出类型为推挽输出 */
     GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;

     /* 设置引脚为上拉模式 */
     GPIO_InitStruct.Pull  = GPIO_PULLUP;

     /* 设置引脚速率为高速 */
     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
```

(下页继续)

(续上页)

```
    /* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
    HAL_GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStruct);

    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED2_PIN;
    HAL_GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStruct);

    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED3_PIN;
    HAL_GPIO_Init(LED3_GPIO_PORT, &GPIO_InitStruct);

        /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = LED4_PIN;
    HAL_GPIO_Init(LED4_GPIO_PORT, &GPIO_InitStruct);

    /* 关闭 RGB 灯 */
    LED_RGBOFF;
}
```

整个函数与“构建库函数雏形”章节中的类似，主要区别是硬件相关的部分使用宏来代替，初始化 GPIO 端口时钟时也采用了 STM32 库函数，函数执行流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置。

(2) 调用宏定义函数 LED1_GPIO_CLK_ENABLE() 来使能 LED 灯的 GPIO 端口时钟，在前面的章节中我们是直接向 RCC 寄存器赋值来使能时钟的，不如这样直观。该函数在 HAL 库里边将操作寄存器部分封装起来，直接调用宏即可。

(3) 向 GPIO 初始化结构体赋值，把引脚初始化成推挽输出模式，其中的 GPIO_PIN 使用宏“LEDx_PIN”来赋值，使函数的实现方便移植。

(4) 使用以上初始化结构体的配置，调用 HAL_GPIO_Init 函数向寄存器写入参数，完成 GPIO 的初始化，这里的 GPIO 端口使用“LEDx_GPIO_PORT”宏来赋值，也是为了程序移植方便。

(5) 使用同样的初始化结构体，只修改控制的引脚和端口，初始化其它 LED 灯使用的 GPIO 引脚。

(6) 使用宏控制 RGB 灯默认关闭。

11.2.2.4 主函数

编写完 LED 灯的控制函数后，就可以在 main 函数中测试了，见*代码清单 12_10*、*代码清单 12_11*、*代码清单 12_12*。

列表 10: 代码清单 12-10：F103-指南者和 F103-霸道开发板控制 LED 灯，main 文件

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();

    /* LED 端口初始化 */
    LED_GPIO_Config();

    /* 控制 LED 灯 */
    while (1) {
        LED1( ON );       // 亮
        HAL_Delay(1000);
        LED1( OFF );      // 灭
        HAL_Delay(1000);

        LED2( ON );      // 亮
        HAL_Delay(1000);
        LED2( OFF );      // 灭

        LED3( ON );       // 亮
```

(下页继续)

(续上页)

```
        HAL_Delay(1000);
        LED3( OFF );       // 灭

        /* 轮流显示 红绿蓝黄紫青白 颜色 */
        LED_RED;
        HAL_Delay(1000);

        LED_GREEN;
        HAL_Delay(1000);

        LED_BLUE;
        HAL_Delay(1000);

        LED_YELLOW;
        HAL_Delay(1000);

        LED_PURPLE;
        HAL_Delay(1000);

        LED_CYAN;
        HAL_Delay(1000);

        LED_WHITE;
        HAL_Delay(1000);

        LED_RGBOFF;
        HAL_Delay(1000);
    }
}
```

列表 11: 代码清单 12-11：F103-MINI 开发板控制 LED 灯，main 文件

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();

    /* LED 端口初始化 */
    LED_GPIO_Config();

    /* 控制 LED 灯 */
    while (1) {
        LED1( ON );       // 亮
        HAL_Delay(1000);
        LED1( OFF );      // 灭
        HAL_Delay(1000);

        LED2( ON );      // 亮
        HAL_Delay(1000);
        LED2( OFF );      // 灭
    }
}
```

列表 12: 代码清单 12-12：F103 拂晓电机开发板控制 LED 灯，main 文件

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();
```

(下页继续)

(续上页)

```
    /* LED 端口初始化 */
    LED_GPIO_Config();

    /* 控制 LED 灯 */
    while (1) {
        LED1( ON );       // 亮
        HAL_Delay(1000);
        LED1( OFF );      // 灭
        HAL_Delay(1000);

        LED2( ON );      // 亮
        HAL_Delay(1000);
        LED2( OFF );      // 灭

        LED3( ON );       // 亮
        HAL_Delay(1000);
        LED3( OFF );      // 灭

        /* 轮流显示 红绿蓝黄紫青白 颜色 */
        LED_RED;
        HAL_Delay(1000);

        LED_GREEN;
        HAL_Delay(1000);

        LED_BLUE;
        HAL_Delay(1000);

        LED_YELLOW;
        HAL_Delay(1000);

        LED_PURPLE;
```

(下页继续)

(续上页)

```
        HAL_Delay(1000);

        LED_CYAN;
        HAL_Delay(1000);

        LED_WHITE;
        HAL_Delay(1000);

        LED_RGBOFF;
        HAL_Delay(1000);
    }
}
```

在 main 函数中，调用 SystemClock_Config 函数初始化系统的时钟为 72MHz，所有程序都必须设置好系统的时钟再进行其他操作，具体设置将在 RCC 时钟章节详细讲解，接着调用我们前面定义的 LED_GPIO_Config 初始化好 LED 的控制引脚，然后直接调用各种控制 LED 灯亮灭的宏来实现 LED 灯的控制，延时采用库自带基于滴答时钟延时 HAL_Delay 单位为 ms，直接调用即可，这里 HAL_Delay(1000) 表示延时 1s。

以上，就是一个使用 STM32 HAL 软件库开发应用的流程。

11.2.3 下载验证

把编译好的程序下载到开发板并复位，可看到 RGB 彩灯轮流显示不同的颜色。

11.3 STM32 HAL 库补充知识

11.3.1 SystemInit 函数去哪了？

在前几章我们自己建工程的时候需要定义一个 SystemInit 空函数，但是在这个用 STM32 HAL 库的工程却没有这样做，SystemInit 函数去哪了呢?

这个函数在 STM32 HAL 库的“system_stm32f1xx.c”文件中定义了，而我们的工程已经包含该文件。

11.3.2 断言

细心对比过前几章我们自己定义的 GPIO_Init 函数与 STM32 HAL 库中同名函数的读者，会发现 HAL 库中的函数内容多了一些乱七八糟的东西，见*代码清单 12_13*。

列表 13: 代码清单 12-13：HAL_GPIO_Init 函数的断言部分

```
void HAL_GPIO_Init(GPIO_TypeDef  *GPIOx, GPIO_InitTypeDef *GPIO_Init)
{
    uint32_t position = 0x00;
    uint32_t ioposition = 0x00;
    uint32_t iocurrent = 0x00;
    uint32_t temp = 0x00;

    /* Check the parameters */
    assert_param(IS_GPIO_ALL_INSTANCE(GPIOx));
    assert_param(IS_GPIO_PIN(GPIO_Init->Pin));
    assert_param(IS_GPIO_MODE(GPIO_Init->Mode));
    assert_param(IS_GPIO_PULL(GPIO_Init->Pull));
    /* ------- 以下内容省略，跟前面我们定义的函数内容相同----- */
}
```

基本上每个库函数的开头都会有这样类似的内容，这里的“assert_param”实际是一个宏，在库函数中它用于检查输入参数是否符合要求，若不符合要求则执行某个函数输出警告，“assert_param”的定义见代码清单 *12_14*。

列表 14: 代码清单 12-14：stm32f1xx_hal_conf.h 文件中关于断言的定义

```
#ifdef  USE_FULL_ASSERT
/**
* @brief  assert_param 宏用于函数的输入参数检查
* @param  expr: 若 expr 值为假，则调用 assert_failed 函数
*              报告文件名及错误行号
*              若 expr 值为真，则不执行操作
*/
#define assert_param(expr) \
        ((expr) ? (void)0 : assert_failed((uint8_t *)__FILE__, __LINE__))
/* 错误输出函数 ------------------------------------------------------- */
void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif
```

这段代码的意思是，假如我们不定义“USE_FULL_ASSERT”宏，那么“assert_param”就是一个空的宏 (#else 与 #endif 之间的语句生效)，没有任何操作。从而所有库函数中的 assert_param 实际上都无意义，我们就当看不见好了。

假如我们定义了“USE_FULL_ASSERT”宏，那么“assert_param”就是一个有操作的语句 (#if 与 #else 之间的语句生效)，该宏对参数 expr 使用 C 语言中的问号表达式进行判断，若 expr 值为真，则无操作 (void 0)，若表达式的值为假，则调用“assert_failed”函数，且该函数的输入参数为“__FILE__”及“__LINE__”，这两个参数分别代表“assert_param”宏被调用时所在的“文件名”及“行号”。

但库文件只对“assert_failed”写了函数声明，没有写函数定义，实际用时需要用户来定义，我们一般会用 printf 函数来输出这些信息，见代码清单 *12_15*。

列表 15: 代码清单 12-15：assert_failed 输出错误信息

```
void assert_failed(uint8_t \* file, uint32_t line)
{
    printf( “/r/n 输入参数错误，错误文件名 =%s, 行号 =%s” ,file,line);
}
```

注意在我们的这个 LED 工程中，还不支持 printf 函数 (在 USART 外设章节会讲解)，想测试 assert_failed 输出的读者，可以在这个函数中做点亮红色 LED 灯的操作，作为警告输出测试。

那么为什么函数输入参数不对的时候，assert_param 宏中的 expr 参数值会是假呢? 这要回到 HAL_GPIO_Init 函数，看它对 assert_param 宏的调用，它被调用时分别以“IS_GPIO_ALL_INSTANCE(GPIOx)” 、 “IS_GPIO_PIN(GPIO_Init->Pin)” 等作为输入参数，也就是说被调用时，expr 实际上是一条针对输入参数的判断表达式。例如“IS_GPIO_PIN” 的宏定义：

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

若它的输入参数 PIN 值为 0，则表达式的值为假，PIN 非 0 时表达式的值为真。我们知道用于选择 GPIO 引脚号的宏“GPIO_PIN_x” 的值至少有一个数据位为 1，这样的输入参数才有意义，若 GPIO_InitStruct-> Pin 的值为 0，输入参数就无效了。配合 IS_GPIO_PIN”这句表达式，“assert_param” 就实现了检查输入参数的功能。对 assert_param 宏的其它调用方式类似，大家可以自己看库源码来研究一下。

11.3.3 Doxygen 注释规范

在 STM32 HAL 库以及我们自己编写的“bsp_led.c” 文件中，可以看到一些比较特别的注释，类似*代码清单 12_16*。

列表 16: 代码清单 12-16：Doxygen 注释规范

```
/**
* @brief  初始化控制 LED 的 IO
* @param  无
* @retval 无
*/
```

这是一种名为“Doxygen”的注释规范，如果在工程文件中按照这种规范去注释，可以使用 Doxygen 软件自动根据注释生成帮助文档。我们所说非常重要的库帮助文档《STM32F103xx_User_Manual.chm》，就是由该软件根据库文件的注释生成的。关于 Doxygen 注释规范本教程不作讲解，感兴趣的读者可自行搜索网络上的资料学习。

11.3.4 防止头文件重复包含

在 STM32 HAL 库的所有头文件以及我们自己编写的“bsp_led.h”头文件中，可看到类似*代码清单 12_17* 的宏定义。它的功能是防止头文件被重复包含，避免引起编译错误。

列表 17: 代码清单 12-17：防止头文件重复包含的宏

```
#ifndef __LED_H
#define __LED_H

/* 此处省略头文件的具体内容 */

#endif /* end of __LED_H */
```

在头文件的开头，使用“#ifndef”关键字，判断标号“__LED_H”是否被定义，若没有被定义，则从“#ifndef”至“#endif”关键字之间的内容都有效，也就是说，这个头文件若被其它文件“#include”，它就会被包含到其该文件中了，且头文件中紧接着使用“#define”关键字定义上面判断的标号“__LED_H”。当这个头文件被同一个文件第二次“#include”包含的时候，由于有了第一次包含中的“#define __LED_H”定义，这时再判断“#ifndef __LED_H”，判断的结果就是假了，从“#ifndef”至

“#endif” 之间的内容都无效，从而防止了同一个头文件被包含多次，编译时就不会出现 “redefine（重复定义）” 的错误了。

一般来说，我们不会直接在 C 的源文件写两个 “#include” 来包含同一个头文件，但可能因为头文件内部的包含导致重复，这种代码主要是避免这样的问题。如 “bsp_led.h” 文件中使用了 “#include “stm32F103xx.h”” 语句，按习惯，可能我们写主程序的时候会在 main 文件写 “#include “bsp_led.h” 及 #include “stm32F103xx.h””，这个时候 “stm32F103xx.h” 文件就被包含两次了，如果没有这种机制，就会出错。

至于为什么要用两个下划线来定义 “__LED_H” 标号，其实这只是防止它与其它普通宏定义重复了，如我们用 “GPIO_PIN_0” 来代替这个判断标号，就会因为 stm32F103xx.h 已经定义了 GPIO_PIN_0，结果导致 “bsp_led.h” 文件无效了，“bsp_led.h” 文件一次都没被包含。

第 12 章　GPIO 输入—按键检测

按键检测使用到 GPIO 外设的基本输入功能，本章中不再赘述 GPIO 外设的概念，如您忘记了，可重读前面“GPIO 框图剖析”小节，STM32 HAL 库中 GPIO 初始化结构体 GPIO_TypeDef 的定义与“定义引脚模式的枚举类型”小节中讲解的相同。

12.1 硬件设计

按键机械触点断开、闭合时，由于触点的弹性作用，按键开关不会马上稳定接通或一下子断开，使用按键时会产生图 *13_1* 中的带波纹信号，需要用软件消抖处理滤波，不方便输入检测。本实验板连接的按键带硬件消抖功能，见图 *13_2a* 和图 *13_2b*，它利用电容充放电的延时，消除了波纹，从而简化软件的处理，软件只需要直接检测引脚的电平即可。

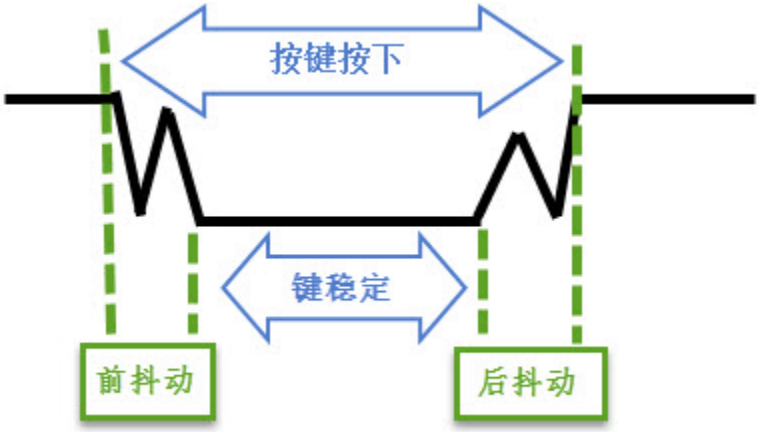


图 13-1 按键抖动说明图

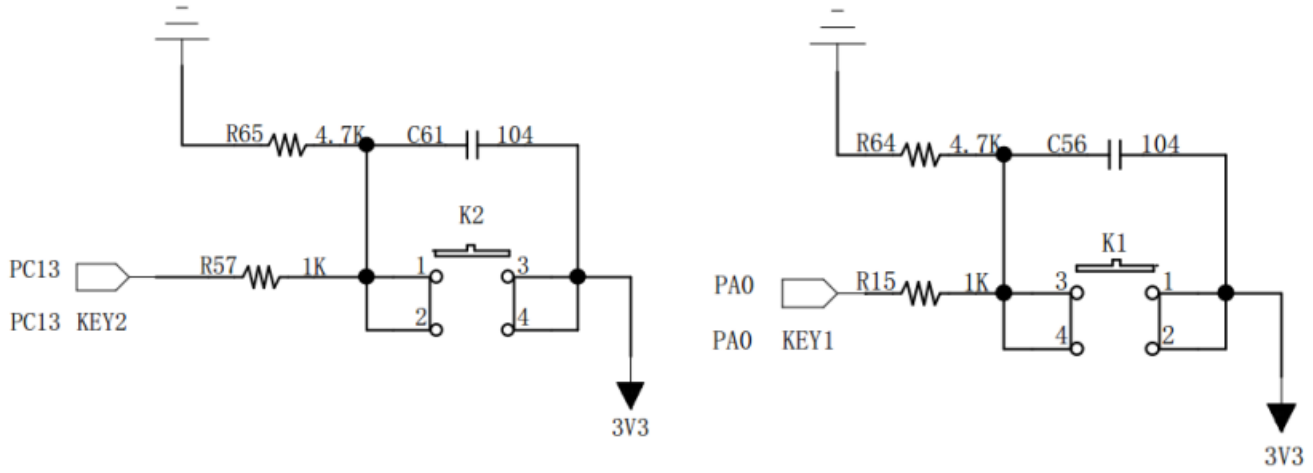
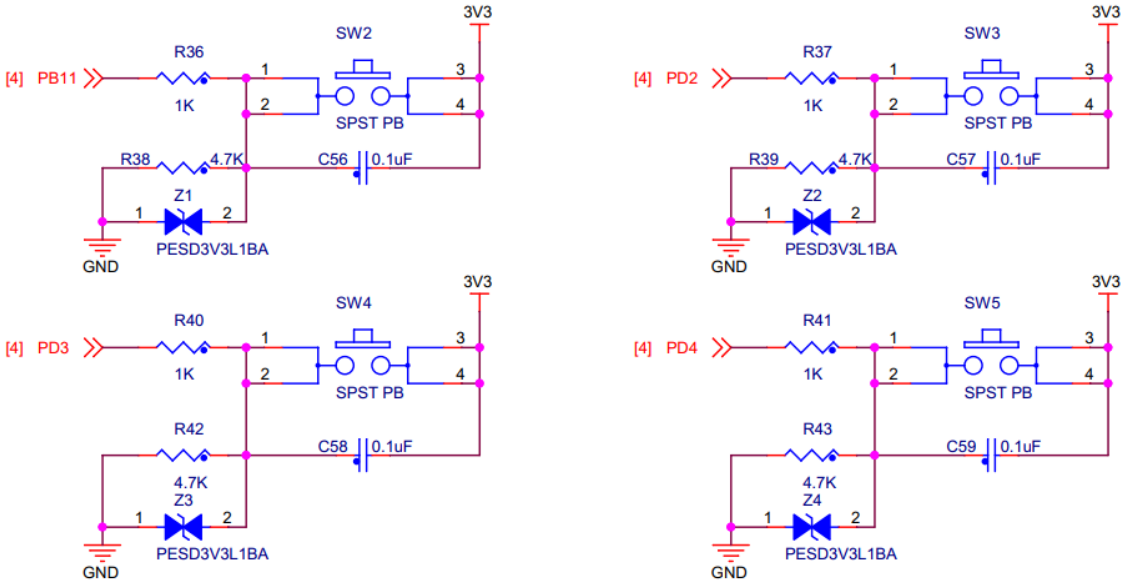


图 13-2a MINI 板、指南者和霸道开发板的按键原理图

按键



复位

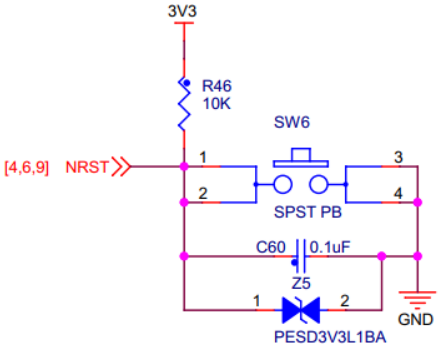


图 13-2b 拂晓电机开发板的按键原理图

从按键的原理图可知，这些按键在没有被按下的时候，GPIO 引脚的输入状态为低电平 (按键所在的电路不通，引脚接地)，当按键按下时，GPIO 引脚的输入状态为高电平 (按键所在的电路导通，引脚接到电源)。只要我们检测引脚的输入电平，即可判断按键是否被按下。

若您使用的实验板按键的连接方式或引脚不一样，只需根据我们的工程修改引脚即可，程序的控制原理相同。

12.2 软件设计

同 LED 的工程，为了使工程更加有条理，我们把按键相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_key.c”及“bsp_key.h”文件，这些文件也可根据您的喜好命名，这些文件不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

12.2.1 编程要点

1. 使能 GPIO 端口时钟；
2. 初始化 GPIO 目标引脚为输入模式 (浮空输入)；
3. 编写简单测试程序，检测按键的状态，实现按键控制 LED 灯。

12.2.2 代码分析

12.2.2.1 按键引脚宏定义

同样，在编写按键驱动时，也要考虑更改硬件环境的情况。我们把按键检测引脚相关的宏定义到“bsp_key.h”文件中，见*代码清单 13_1a*。

列表 1: 代码清单 13-1a MINI 板、指南者和霸道开发板按键检测引脚相关的宏

```
//引脚定义
/*******************************************************/
#define KEY1_PIN                    GPIO_PIN_0
#define KEY1_GPIO_PORT              GPIOA
#define KEY1_GPIO_CLK_ENABLE()      __GPIOA_CLK_ENABLE()

#define KEY2_PIN                    GPIO_PIN_13
#define KEY2_GPIO_PORT              GPIOC
#define KEY2_GPIO_CLK_ENABLE()      __GPIOC_CLK_ENABLE()
/*******************************************************/

/** 按键按下标置宏
* 按键按下为高电平，设置 KEY_ON=1， KEY_OFF=0
* 若按键按下为低电平，把宏设置成 KEY_ON=0 ，KEY_OFF=1 即可
*/
#define KEY_ON  1
#define KEY_OFF 0
```

列表 2: 代码清单 13-1b 拂晓电机开发板按键检测引脚相关的宏

```
//引脚定义
/*******************************************************/
#define KEY1_PIN GPIO_PIN_11
#define KEY1_GPIO_PORT GPIOB
#define KEY1_GPIO_CLK_ENABLE() __HAL_RCC_GPIOB_CLK_ENABLE()

#define KEY2_PIN GPIO_PIN_2
#define KEY2_GPIO_PORT GPIOD
```

(下页继续)

(续上页)

```
#define KEY2_GPIO_CLK_ENABLE() __HAL_RCC_GPIOD_CLK_ENABLE()

#define KEY3_PIN GPIO_PIN_3
#define KEY3_GPIO_PORT GPIOD
#define KEY3_GPIO_CLK_ENABLE() __HAL_RCC_GPIOD_CLK_ENABLE()

#define KEY4_PIN GPIO_PIN_4
#define KEY4_GPIO_PORT GPIOD
#define KEY4_GPIO_CLK_ENABLE() __HAL_RCC_GPIOD_CLK_ENABLE()
/*******************************************************/

/** 按键按下标置宏
* 按键按下为高电平，设置 KEY_ON=1， KEY_OFF=0
* 若按键按下为低电平，把宏设置成 KEY_ON=0 ，KEY_OFF=1 即可
*/
#define KEY_ON  1
#define KEY_OFF 0
```

以上代码根据按键的硬件连接，把检测按键输入的 GPIO 端口、GPIO 引脚号以及 GPIO 端口时钟封装起来了。

12.2.2.2 按键 GPIO 初始化函数

利用上面的宏，编写按键的初始化函数，见*代码清单 13_2*。该函数对于我们所有的 F1 开发板都是通用的，但由于拂晓电机开发板的按键有 4 个，所以该函数仅初始化了其中的 2 个按键，读者可自行添加代码初始化另外两个按键。

列表 3: 代码清单 13-2 按键 GPIO 初始化函数

```
void Key_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
```

(下页继续)

(续上页)

```
    /* 开启按键 GPIO 口的时钟 */
    KEY1_GPIO_CLK_ENABLE();
    KEY2_GPIO_CLK_ENABLE();
    /* 选择按键的引脚 */
    GPIO_InitStructure.Pin = KEY1_PIN;
    /* 设置引脚为输入模式 */
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;
    /* 设置引脚不上拉也不下拉 */
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    /* 使用上面的结构体初始化按键 */
    HAL_GPIO_Init(KEY1_GPIO_PORT, &GPIO_InitStructure);
    /* 选择按键的引脚 */
    GPIO_InitStructure.Pin = KEY2_PIN;
    /* 使用上面的结构体初始化按键 */
    HAL_GPIO_Init(KEY2_GPIO_PORT, &GPIO_InitStructure);
}
```

同为 GPIO 的初始化函数，初始化的流程与前一章初始化 LED 的 GPIO 引脚的流程类似，主要区别是引脚的模式。函数执行流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置。

(2) 用宏定义函数 KEY1_GPIO_CLK_ENABLE()，KEY2_GPIO_CLK_ENABLE() 来使能按键的 GPIO 端口时钟。

(3) 向 GPIO 初始化结构体赋值，把引脚初始化成浮空输入模式，其中的 Pin 使用宏“KEYx_PIN”来赋值，使函数的实现方便移植。由于引脚的默认电平受按键电路影响，所以设置成“浮空/上拉/下拉”模式均没有区别。

(4) 使用以上初始化结构体的配置，调用 HAL_GPIO_Init 函数向寄存器写入参数，完成 GPIO 的初始化，这里的 GPIO 端口使用“KEYx_GPIO_PORT”宏来赋值，也是为了程序移植方便。

(5) 使用同样的初始化结构体，只修改控制的引脚和端口，初始化其它按键检测时使用的 GPIO 引脚。

12.2.2.3 检测按键的状态

初始化按键后，就可以通过检测对应引脚的电平来判断按键状态了，见*代码清单 13_3*。

列表 4: 代码清单 13-3 检测按键的状态

```
uint8_t Key_Scan(GPIO_TypeDef* GPIOx,uint16_t GPIO_Pin)
{
    /* 检测是否有按键按下 */
    if (HAL_GPIO_ReadPin(GPIOx,GPIO_Pin) == KEY_ON ) {
        /* 等待按键释放 */
        while (HAL_GPIO_ReadPin(GPIOx,GPIO_Pin) == KEY_ON);
        return  KEY_ON;
    } else
        return KEY_OFF;
}
```

在这里我们定义了一个 Key_Scan 函数用于扫描指定按键的状态。GPIO 引脚的输入电平可通过读取 IDR 寄存器对应的数据位来感知，而 STM32 HAL 库提供了库函数 HAL_GPIO_ReadPin 来获取位状态，该函数输入 GPIO 端口及引脚号，函数返回该引脚的电平状态，高电平返回 1，低电平返回 0。Key_Scan 函数中以 HAL_GPIO_ReadPin 的返回值与自定义的宏“KEY_ON”对比，若检测到按键按下，则使用 while 循环持续检测按键状态，直到按键释放，按键释放后 Key_Scan 函数返回一个“KEY_ON”值；若没有检测到按键按下，则函数直接返回“KEY_OFF”。若按键的硬件没有做消抖处理，需要在这个 Key_Scan 函数中做软件滤波，防止波纹抖动引起误触发。

12.2.2.4 主函数

接下来我们使用主函数编写按键检测流程，见*代码清单 13_4*。

列表 5: 代码清单 13-4 按键检测主函数

```
int main(void)
{
```

(下页继续)

(续上页)

```
    /* 系统时钟初始化成 72MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 初始化按键 */
    Key_GPIO_Config();
    /* 轮询按键状态，若按键按下则反转 LED */
    while (1) {
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON  ) {
            /*LED1 反转 */
            LED1_TOGGLE;
        }

        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON  ) {
            /*LED2 反转 */
            LED2_TOGGLE;
        }
    }
}
```

代码中初始化 LED 灯及按键后，在 while 函数里不断调用 Key_Scan 函数，并判断其返回值，若返回值表示按键按下，则反转 LED 灯的状态。

12.2.3 下载验证

把编译好的程序下载到开发板并复位，按下按键可以控制 LED 灯亮、灭状态。

第 13 章　启动文件详解

本章参考资料《STM32F10X-中文参考手册》第九章-中断和事件：表 55 其他 STM32F10xxx 产品（小容量、中容量和大容量）的向量表；MDK 中的帮助手册—ARM Development Tools：用来查询 ARM 的汇编指令和编译器相关的指令。

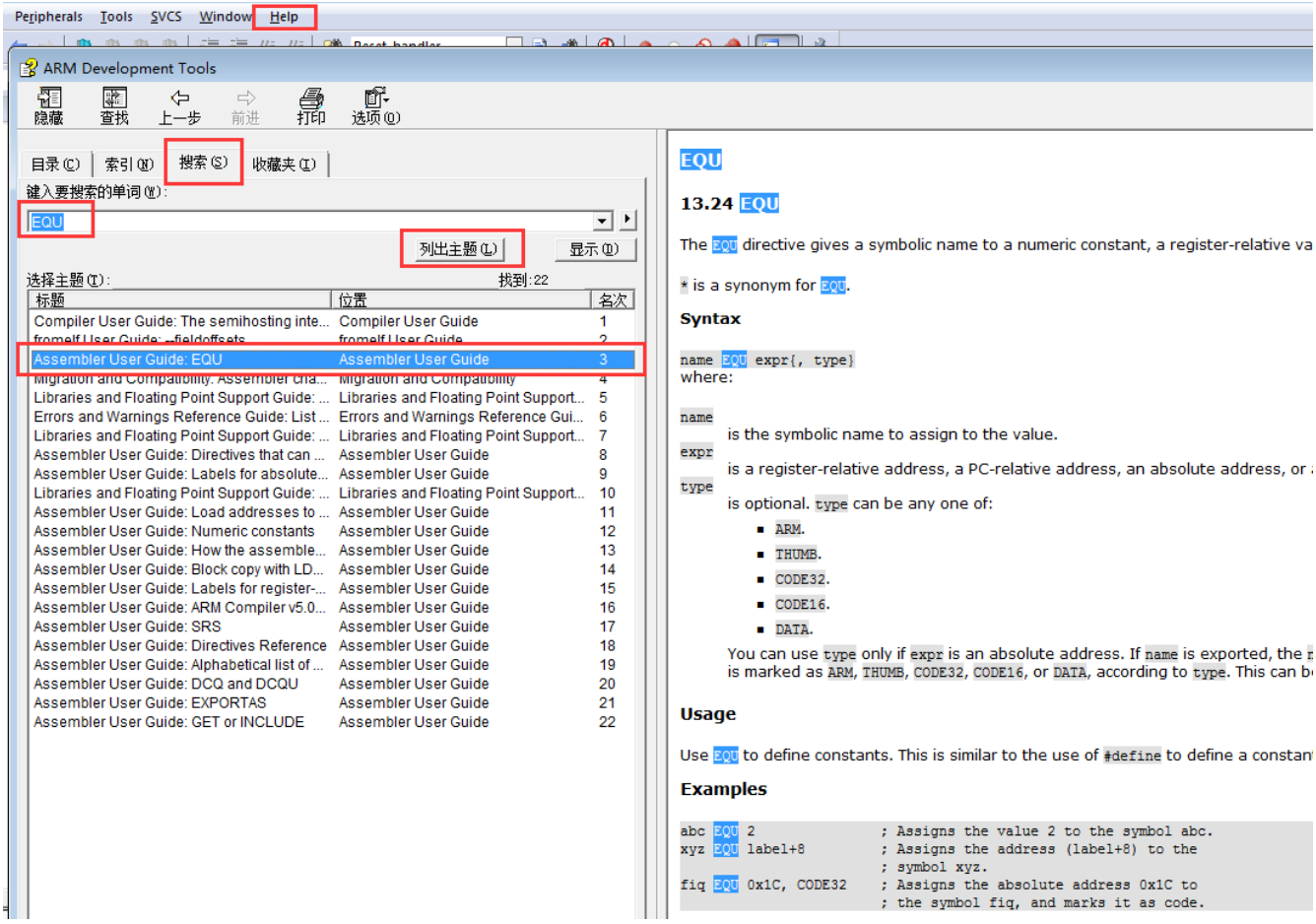
13.1 启动文件简介

启动文件由汇编编写，是系统上电复位后第一个执行的程序。主要做了以下工作：

1. 初始化堆栈指针 SP（__initial_sp）
2. 初始化 PC 指针（Reset_Handler）
3. 初始化中断向量表（__Vectors）
4. 配置系统时钟（SystemInit）
5. 调用 C 库函数 _main 初始化用户堆栈，从而最终调用 main 函数去到 C 的世界

13.2 查找 ARM 汇编指令

在讲解启动代码的时候，会涉及到 ARM 的汇编指令和 Cortex 内核的指令，有关 Cortex 内核的指令我们可以参考《CM3 权威指南 CnR2》第四章：指令集。剩下的 ARM 的汇编指令我们可以在 MDK->Help->Uvision Help 中搜索到，以 EQU 为例，检索如下：



检索出来的结果会有很多，我们只需要看 Assembler User Guide 这部分即可。下面列出了启动文件中使用到的 ARM 汇编指令，该列表的指令全部从 ARM Development Tools 这个帮助文档里面检索而来。其中编译器相关的指令 WEAK 和 ALIGN 为了方便也放在同一个表格了。

表格 14-1 启动文件使用的 ARM 汇编指令汇总

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
SPACE	分配内存空间
PRESERVE8	当前文件堆栈需按照 8 字节对齐
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出错。要注意的是：这个不是 ARM 的指令，是编译器的，这里放在一起只是为了方便。
IMPORT	声明标号来自外部文件，跟 C 语言中的 EXTERN 关键字类似
B	跳转到一个标号
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，这里放在一起只是为了方便。
END	到达文件的末尾，文件结束
IF,ELSE,ENDIF	汇编条件分支语句，跟 C 语言的 if else 类似

13.3 启动文件代码讲解

13.3.1 Stack—栈

```
Stack_Size      EQU     0x00000400
```

(下页继续)

(续上页)

```
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp
```

开辟栈的大小为 0X00000400（1KB），名字为 STACK，NOINIT 即不初始化，可读可写，8（2^3）字节对齐。

栈的作用是用于局部变量，函数调用，函数形参等的开销，栈的大小不能超过内部 SRAM 的大小。如果编写的程序比较大，定义的局部变量很多，那么就需要修改栈的大小。如果某一天，你写的程序出现了莫名奇怪的错误，并进入了硬 fault 的时候，这时你就要考虑下是不是栈不够大，溢出了。

EQU：宏定义的伪指令，相当于等于，类似与 C 中的 define。

AREA：告诉汇编器汇编一个新的代码段或者数据段。STACK 表示段名，这个可以任意命名；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。

SPACE：用于分配一定大小的内存空间，单位为字节。这里指定大小等于 Stack_Size。

标号 **__initial_sp** 紧挨着 SPACE 语句放置，表示栈的结束地址，即栈顶地址，栈是由高向低生长的。

13.3.2 Heap 堆

```
Heap_Size       EQU     0x00000200

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

开辟堆的大小为 0X00000200（512 字节），名字为 HEAP，NOINIT 即不初始化，可读可写，8（2^3）

字节对齐。__heap_base 表示对的起始地址，__heap_limit 表示堆的结束地址。堆是由低向高生长的，跟栈的生长方向相反。

堆主要用来动态内存的分配，像 malloc() 函数申请的内存就在堆上面。这个在 STM32 里面用的比较少。

```
PRESERVE8
THUMB
```

PRESERVE8：指定当前文件的堆栈按照 8 字节对齐。

THUMB：表示后面指令兼容 THUMB 指令。THUBM 是 ARM 以前的指令集，16bit，现在 Cortex-M 系列的都使用 THUMB-2 指令集，THUMB-2 是 32 位的，兼容 16 位和 32 位的指令，是 THUMB 的超集。

13.3.3 向量表

```
AREA    RESET, DATA, READONLY
EXPORT  __Vectors
EXPORT  __Vectors_End
EXPORT  __Vectors_Size
```

定义一个数据段，名字为 RESET，可读。并声明 __Vectors、__Vectors_End 和 __Vectors_Size 这三个标号具有全局属性，可供外部的文件调用。

EXPORT：声明一个标号可被外部的文件使用，使标号具有全局属性。如果是 IAR 编译器，则使用的是 GLOBAL 这个指令。

当内核响应了一个发生的异常后，对应的异常服务例程 (ESR) 就会执行。为了决定 ESR 的入口地址，内核使用了“向量表查表机制”。这里使用一张向量表。向量表其实是一个 WORD（32 位整数）数组，每个下标对应一种异常，该下标元素的值则是该 ESR 的入口地址。向量表在地址空间中的位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0（即 FLASH 地址 0）处必须包含一张向量表，用于初始时的异常分配。要注意的是这里有个另类：0 号类型并不是什么入口地址，而是给出了复位后 MSP 的

初值。

表格 14-2 F103 向量表

编号	优先级	优先级类型	名称	说明	地址
	-	-	-	保留（实际存的是 MSP 地址）	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)连接到 NMI 向量	0X0000 0008
	-1	固定	HardFault	所有类型的错误	0X0000 000C
	0	可编程	MemManage	存储器管理	0X0000 0010
	1	可编程	BusFault	预取指失败，存储器访问失败	0X0000 0014
	2	可编程	UsageFault	未定义的指令或非法状态	0X0000 0018
	-	-	-	保留	0X0000 001C-0X0000 002B
	3	可编程	SVCall	通过 SWI 指令调用的系统服务	0X0000 002C
	4	可编程	Debug Monitor	调试监控器	0X0000 0030
	-	-	-	保留	0X0000 0034
	5	可编程	PendSV	可挂起的系统服务	0X0000 0038
	6	可编程	SysTick	系统嘀嗒定时器	0X0000 003C
0	7	可编程	WWDG	窗口看门狗中断	0X0000 0040
1	8	可编程	PVD	连到 EXTI 的电源电压检测(PVD)中断	0X0000 0044
2	9	可编程	TAMPER	侵入检测中断	0X0000 0048
中间部分省略，详情请参考《STM32 中文参考手册》第九章-中断和事件-向量表部分					
57	64	可编程	DMA2 通道 2	DMA2 通道 2 中断	0X0000 0124
58	65	可编程	DMA2 通道 3	DMA2 通道 3z 中断	0X0000 0128
59	66	可编程	DMA2 通道 4_5	DMA2 通道 4 和通道 5 中断	0X0000 012C

列表 1: 代码 14-1 向量表

```
__Vectors  DCD    __initial_sp          ; 栈顶地址
           DCD    Reset_Handler         ; 复位程序地址
           DCD    NMI_Handler
           DCD    HardFault_Handler
           DCD    MemManage_Handler
```

(下页继续)

(续上页)

```
        DCD    BusFault_Handler
        DCD    UsageFault_Handler
        DCD    0                    ; 0 表示保留
        DCD    0
        DCD    0
        DCD    0
        DCD    SVC_Handler
        DCD    DebugMon_Handler
        DCD    0
        DCD    PendSV_Handler
        DCD    SysTick_Handler

; 外部中断开始
        DCD    WWDG_IRQHandler
        DCD    PVD_IRQHandler
        DCD    TAMPER_IRQHandler

; 限于篇幅，中间代码省略
        DCD    DMA2_Channel2_IRQHandler
        DCD    DMA2_Channel3_IRQHandler
        DCD    DMA2_Channel4_5_IRQHandler
__Vectors_End
__Vectors_Size EQU __Vectors_End - __Vectors
```

__Vectors 为向量表起始地址，__Vectors_End 为向量表结束地址，两个相减即可算出向量表大小。

向量表从 FLASH 的 0 地址开始放置，以 4 个字节为一个单位，地址 0 存放的是栈顶地址，0X04 存放的是复位程序的地址，以此类推。从代码上看，向量表中存放的都是中断服务函数的函数名，可我们知道 C 语言中的函数名就是一个地址。

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。在向量表中，DCD 分配了一堆内存，并且以 ESR 的入口地址初始化它们。

13.3.4 复位程序

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

定义一个名称为.text 的代码段，可读。

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

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

复位子程序是系统上电后第一个执行的程序，调用 SystemInit 函数初始化系统时钟，然后调用 C 库函数 _mian，最终调用 main 函数去到 C 的世界。

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该标号，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，跟 C 语言中的 EXTERN 关键字类似。这里表示 SystemInit 和 __main 这两个函数均来自外部的文件。

SystemInit() 是一个标准的库函数，在 system_stm32f103xe.c 这个库文件中定义。主要作用是配置系统时钟，这里调用这个函数之后，单片机的系统时钟配被配置为 72M。

__main 是一个标准的 C 库函数，主要作用是初始化用户堆栈，并在函数的最后调用 main 函数去到 C 的世界。这就是为什么我们写的程序都有一个 main 函数的原因。

LDR、BLX、BX 是 CM4 内核的指令，可在《CM3 权威指南 CnR2》第四章-指令集里面查询到，具体作用见下表：

指令名称	作用
LDR	从存储器中加载字到一个寄存器中
BL	跳转到由寄存器/标号给出的地址，并把跳转前的下条指令地址保存到 LR
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

13.3.5 中断服务程序

在启动文件里面已经帮我们写好所有中断的中断服务函数，跟我们平时写的中断服务函数不一样的就是这些函数都是空的，真正的中断复服务程序需要我们在外部的 C 文件里面重新实现，这里只是提前占了一个位置而已。

如果我们在使用某个外设的时候，开启了某个中断，但是又忘记编写配套的中断服务程序或者函数名写错，那当中断来临的时，程序就会跳转到启动文件预先写好的空的中断服务程序中，并且在这个空函数中无线循环，即程序就死在这里。

```
NMI_Handler     PROC    ; 系统异常
                EXPORT  NMI_Handler             [WEAK]
                B       .
                ENDP

; 限于篇幅，中间代码省略
SysTick_Handler PROC
                EXPORT  SysTick_Handler         [WEAK]
                B       .
                ENDP

Default_Handler PROC    ; 外部中断
                EXPORT  WWDG_IRQHandler         [WEAK]
```

(下页继续)

(续上页)

```
                EXPORT  PVD_IRQHandler          [WEAK]
                EXPORT  TAMP_STAMP_IRQHandler [WEAK]

; 限于篇幅，中间代码省略
LTDC_IRQHandler
LTDC_ER_IRQHandler
DMA2D_IRQHandler
                B       .
                ENDP
```

B：跳转到一个标号。这里跳转到一个‘.’，即表示无线循环。

13.3.6 用户堆栈初始化

ALIGN：对指令或者数据存放的地址进行对齐，后面会跟一个立即数。缺省表示 4 字节对齐。

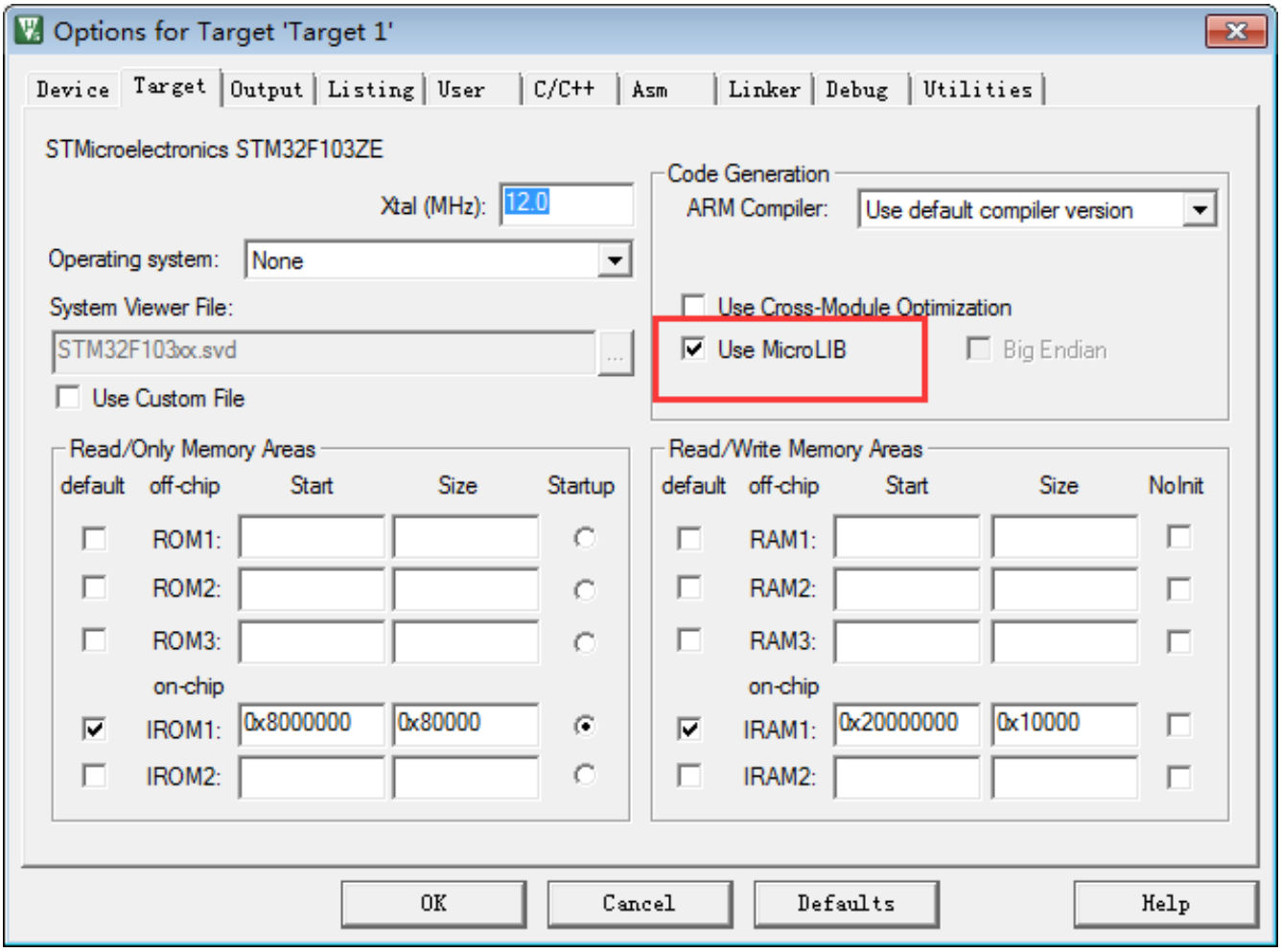
```
; 用户栈和堆初始化，由 C 库函数_main 来完成
    IF      :DEF:__MICROLIB  ; 这个宏在 KEIL 里面开启

    EXPORT  __initial_sp
    EXPORT  __heap_base
    EXPORT  __heap_limit

    ELSE

    IMPORT  __use_two_region_memory ; 这个函数由用户自己实现
    EXPORT  __user_initial_stackheap

__user_initial_stackheap

    LDR     R0, =  Heap_Mem
```

(下页继续)

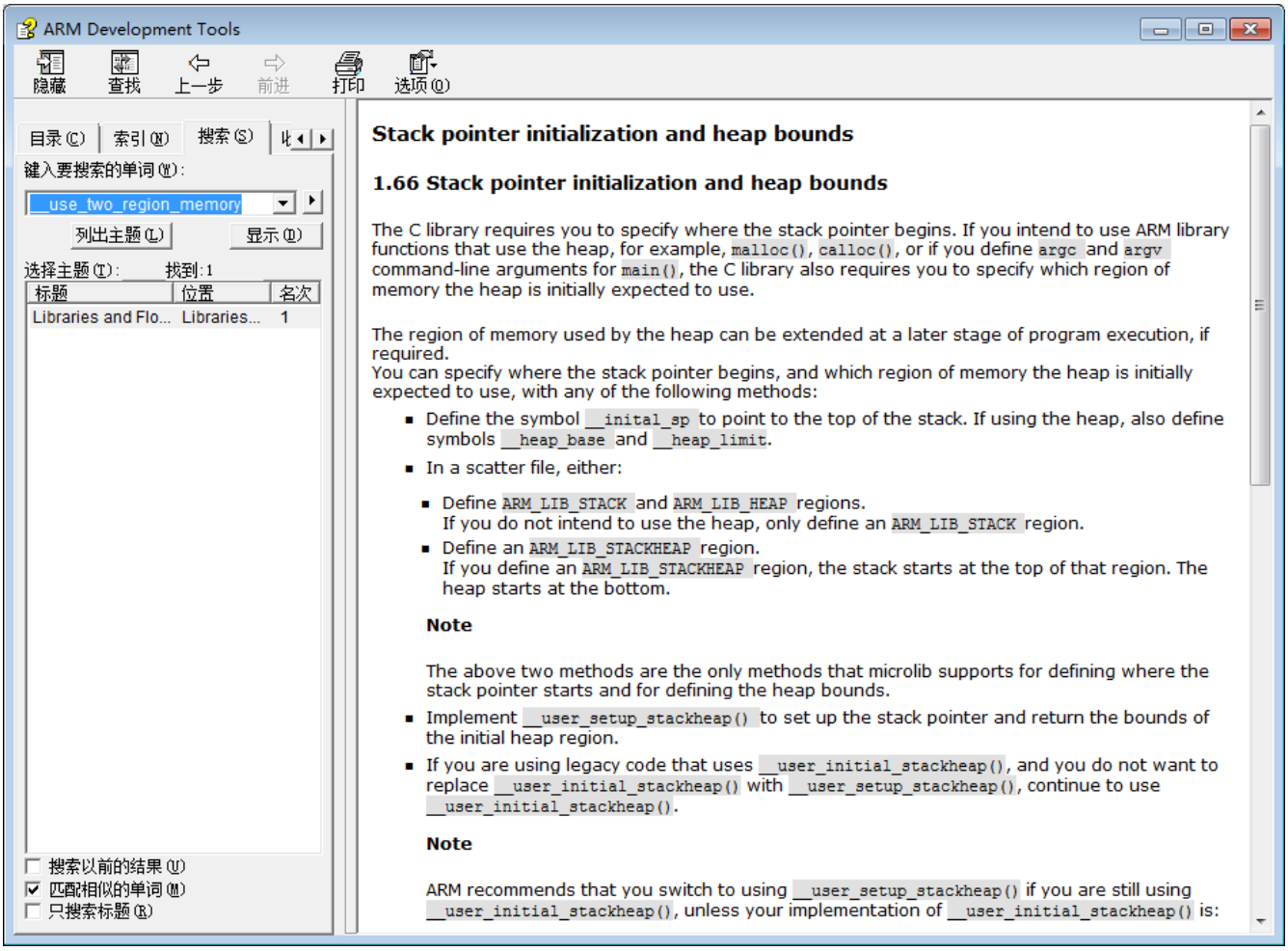
(续上页)

```
    LDR     R1, =(Stack_Mem + Stack_Size)
    LDR     R2, = (Heap_Mem +  Heap_Size)
    LDR     R3, = Stack_Mem
    BX      LR

    ALIGN

    ENDIF
    END
```

首先判断是否定义了 __MICROLIB，如果定义了这个宏则赋予标号 __initial_sp（栈顶地址）、__heap_base（堆起始地址）、__heap_limit（堆结束地址）全局属性，可供外部文件调用。有关这个宏我们在 KEIL 里面配置，具体见图 *14_2*。然后堆栈的初始化就由 C 库函数 _main 来完成。



如果没有定义 __MICROLIB，则插入标号 __use_two_region_memory，这个函数需要用户自己实现，具体要实现成什么样，可在 KEIL 的帮助文档里面查询到，具体见图 *14_3*。



然后声明标号 __user_initial_stackheap 具有全局属性，可供外部文件调用，并实现这个标号的内容。

IF,ELSE,ENDIF： 汇编的条件分支语句，跟 C 语言的 if ,else 类似

END： 文件结束。

第 14 章　RCC—使用 HSE/HSI 配置时钟

本章参考资料：《STM32F10X-中文参考手册》RCC 章节。

学习本章时，配合《STM32F10X-中文参考手册》RCC 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

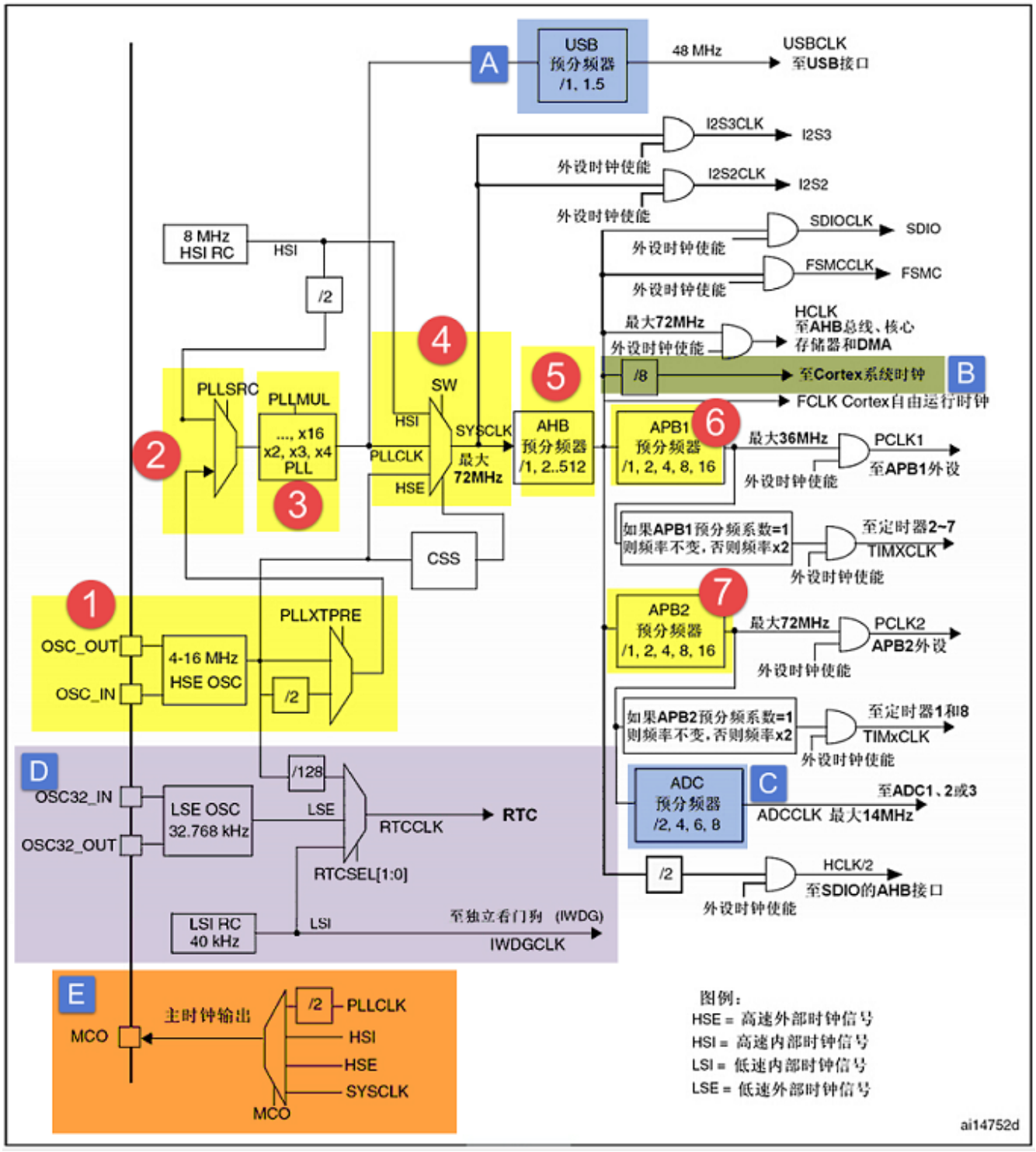
RCC： reset clock control 复位和时钟控制器。本章我们主要讲解时钟部分，特别是要着重理解时钟树，理解了时钟树，STM32 的一切时钟的来龙去脉都会了如指掌。

14.1 RCC 主要作用—时钟部分

设置系统时钟 SYSCLK、设置 AHB 分频因子（决定 HCLK 等于多少）、设置 APB2 分频因子（决定 PCLK2 等于多少）、设置 APB1 分频因子（决定 PCLK1 等于多少）、设置各个外设的分频因子；控制 AHB、APB2 和 APB1 这三条总线时钟的开启、控制每个外设的时钟的开启。对于 SYSCLK、HCLK、PCLK2、PCLK1 这四个时钟的配置一般是：PCLK2 = HCLK = SYSCLK=PLLCLK = 72M，PCLK1=HCLK/2 = 36M。这个时钟配置也是库函数的标准配置，我们用的最多的就是这个。

14.2 RCC 框图剖析—时钟部分

时钟树单纯讲理论的话会比较枯燥，如果选取一条主线，并辅以代码，先主后次讲解的话会很容易，而且记忆还更深刻。我们这里选取库函数时钟系统时钟函数：SetSysClockTo72(); 以这个函数的编写流程来讲解时钟树，这个函数也是我们用库的时候默认的系统时钟设置函数。该函数的功能是利用 HSE 把时钟设置为：PCLK2 = HCLK = SYSCLK = 72M，PCLK1=HCLK/2 = 36M。下面我们就以这个代码的流程为主线，来分析时钟树，对应的是图中的黄色部分，代码流程在时钟树中以数字的大小顺序标识。


A
USB
预分频器
/1, 1.5
48 MHz
USBCLK
至USB接口
I2S3CLK
I2S3
外设时钟使能
I2S2CLK
I2S2
外设时钟使能
SDIOCLK
SDIO
外设时钟使能
FSMCCLK
FSMC
外设时钟使能
最大72MHz
HCLK
至AHB总线、核心
存储器和DMA
外设时钟使能
/8
至Cortex系统时钟
B
FCLK Cortex自由运行时钟
8 MHz
HSI RC
HSI
/2
4
SW
HSI
SYSCLK
PLLCLK
最大
72MHz
HSE
5
AHB
预分频器
/1, 2..512
APB1
预分频器
/1, 2, 4, 8, 16
6
最大36MHz
PCLK1
至APB1外设
外设时钟使能
如果APB1预分频系数=1
则频率不变，否则频率x2
至定时器2~7
TIMXCLK
外设时钟使能
PLLSRC
PLLMUL
..., x16
x2, x3, x4
PLL
2
3
CSS
1
PLLXTPRE
OSC_OUT
OSC_IN
4-16 MHz
HSE OSC
/2
APB2
预分频器
/1, 2, 4, 8, 16
7
最大72MHz
PCLK2
APB2外设
外设时钟使能
如果APB2预分频系数=1
则频率不变，否则频率x2
至定时器1和8
TIMxCLK
外设时钟使能
D
/128
OSC32_IN
OSC32_OUT
LSE OSC
32.768 kHz
LSE
RTCCLK
RTC
RTCSEL[1:0]
ADC
预分频器
/2, 4, 6, 8
C
至ADC1、2或3
ADCCLK 最大14MHz
/2
HCLK/2
至SDIO的AHB接口
外设时钟使能
LSI RC
40 kHz
LSI
至独立看门狗 (IWDG)
IWDGCLK
E
主时钟输出
MCO
/2
PLLCLK
HSI
HSE
SYSCLK
MCO
图例：
HSE = 高速外部时钟信号
HSI = 高速内部时钟信号
LSI = 低速内部时钟信号
LSE = 低速外部时钟信号
ai14752d

14.2.1 系统时钟

14.2.1.1 HSE 高速外部时钟信号

HSE 是高速的外部时钟信号，可以由有源晶振或者无源晶振提供，频率从 4-16MHZ 不等。当使用有源晶振时，时钟从 OSC_IN 引脚进入，OSC_OUT 引脚悬空，当选用无源晶振时，时钟从 OSC_IN 和 OSC_OUT 进入，并且要配谐振电容。

HSE 最常使用的就是 8M 的无源晶振。当确定 PLL 时钟来源的时候，HSE 可以不分频或者 2 分频，这个由时钟配置寄存器 CFGR 的位 17：PLLXTPRE 设置，我们设置为 HSE 不分频。

14.2.1.2 PLL 时钟源

PLL 时钟来源可以有两个，一个来自 HSE，另外一个是 HSI/2，具体用哪个由时钟配置寄存器 CFGR 的位 16：PLLSRC 设置。HSI 是内部高速的时钟信号，频率为 8M，根据温度和环境的情况频率会有漂移，一般不作为 PLL 的时钟来源。这里我们选 HSE 作为 PLL 的时钟来源。

14.2.1.3 PLL 时钟 PLLCLK

通过设置 PLL 的倍频因子，可以对 PLL 的时钟来源进行倍频，倍频因子可以是:[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]，具体设置成多少，由时钟配置寄存器 CFGR 的位 21-18：PLLMUL[3:0] 设置。我们这里设置为 9 倍频，因为上一步我们设置 PLL 的时钟来源为 HSE=8M，所以经过 PLL 倍频之后的 PLL 时钟：PLLCLK = 8M *9 = 72M。72M 是 ST 官方推荐的稳定运行时钟，如果你想超频的话，增大倍频因子即可，最高为 128M。我们这里设置 PLL 时钟：PLLCLK = 8M *9 = 72M。

14.2.1.4 系统时钟 SYSCLK

系统时钟来源可以是：HSI、PLLCLK、HSE，具体的时钟配置寄存器 CFGR 的位 1-0：SW[1:0] 设置。我们这里设置系统时钟：SYSCLK = PLLCLK = 72M。

14.2.1.5 AHB 总线时钟 HCLK

系统时钟 SYSCLK 经过 AHB 预分频器分频之后得到时钟叫 APB 总线时钟，即 HCLK，分频因子可以是:[1,2,4，8，16，64，128，256，512]，具体的由时钟配置寄存器 CFGR 的位 7-4 ：HPRE[3:0] 设置。片上大部分外设的时钟都是经过 HCLK 分频得到，至于 AHB 总线上的外设的时钟设置为多少，得等到我们使用该外设的时候才设置，我们这里只需粗线条的设置好 APB 的时钟即可。我们这里设置为 1 分频，即 HCLK=SYSCLK=72M。

14.2.1.6 APB2 总线时钟 PCLK2

APB2 总线时钟 PCLK2 由 HCLK 经过高速 APB2 预分频器得到，分频因子可以是:[1,2,4，8，16]，具体由时钟配置寄存器 CFGR 的位 13-11：PPRE2[2:0] 决定。HCLK2 属于高速的总线时钟，片上高速的外设就挂载到这条总线上，比如全部的 GPIO、USART1、SPI1 等。至于 APB2 总线上的外设的时钟设置为多少，得等到我们使用该外设的时候才设置，我们这里只需粗线条的设置好 APB2 的时钟即可。我们这里设置为 1 分频，即 PCLK2 = HCLK = 72M。

14.2.1.7 APB1 总线时钟 PCLK1

APB1 总线时钟 PCLK1 由 HCLK 经过低速 APB 预分频器得到，分频因子可以是:[1,2,4，8，16]，具体的由时钟配置寄存器 CFGR 的位 10-8：PRRE1[2:0] 决定。PCLK1 属于低速的总线时钟，最高为 36M，片上低速的外设就挂载到这条总线上，比如 USART2/3/4/5、SPI2/3，I2C1/2 等。至于 APB1 总线上的外设的时钟设置为多少，得等到我们使用该外设的时候才设置，我们这里只需粗线条的设置好 APB1 的时钟即可。我们这里设置为 2 分频，即 PCLK1 = HCLK/2 = 36M。

14.2.1.8 设置系统时钟库函数

上面的 7 个步骤对应的设置系统时钟库函数如下，该函数截取自固件库文件 system_stm32f103xe.c。为了方便阅读，我已把互联型相关的代码删掉，把英文注释翻译成了中文，并把代码标上了序号，总共七个步骤。该函数是直接操作寄存器的，有关寄存器部分请参考数据手册的 RCC 的寄存器描述部分。

列表 1: 代码 15-1 设置系统时钟库函数

```
void HSE_SetSysClock(void)
{
    RCC_ClkInitTypeDef clkinitstruct = {0};
    RCC_OscInitTypeDef oscinitstruct = {0};

    /* Enable HSE Oscillator and activate PLL with HSE as source */
    oscinitstruct.OscillatorType  = RCC_OSCILLATORTYPE_HSE;
    oscinitstruct.HSEState        = RCC_HSE_ON;
    oscinitstruct.HSEPredivValue  = RCC_HSE_PREDIV_DIV1;
    oscinitstruct.PLL.PLLState    = RCC_PLL_ON;
    oscinitstruct.PLL.PLLSource   = RCC_PLLSOURCE_HSE;
    oscinitstruct.PLL.PLLMUL      = RCC_PLL_MUL9;
    if (HAL_RCC_OscConfig(&oscinitstruct)!= HAL_OK) {
        /* Initialization Error */
        while (1);
    }

    /* Select PLL as system clock source and configure the HCLK, PCLK1
    and PCLK2 clocks dividers */
    clkinitstruct.ClockType = (RCC_CLOCKTYPE_SYSCLK |
                           RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1
                           | RCC_CLOCKTYPE_PCLK2);
    clkinitstruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    clkinitstruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    clkinitstruct.APB2CLKDivider = RCC_HCLK_DIV1;
    clkinitstruct.APB1CLKDivider = RCC_HCLK_DIV2;
    if (HAL_RCC_ClockConfig(&clkinitstruct, FLASH_LATENCY_2)!= HAL_OK)
        {
        /* Initialization Error */
        while (1);
```

(下页继续)

(续上页)

```
    }
}

void HSI_SetSysClock(void)
{
    RCC_ClkInitTypeDef clkinitstruct = {0};
    RCC_OscInitTypeDef oscinitstruct = {0};

    /* Enable HSE Oscillator and activate PLL with HSE as source */
    oscinitstruct.OscillatorType  = RCC_OSCILLATORTYPE_HSI;
    oscinitstruct.HSEState        = RCC_HSE_ON;
    oscinitstruct.HSEPredivValue  = RCC_HSE_PREDIV_DIV1;
    oscinitstruct.PLL.PLLState    = RCC_PLL_ON;
    oscinitstruct.PLL.PLLSource   = RCC_PLLSOURCE_HSE;
    oscinitstruct.PLL.PLLMUL      = RCC_PLL_MUL9;
    if (HAL_RCC_OscConfig(&oscinitstruct)!= HAL_OK) {
        /* Initialization Error */
        while (1);
    }

    /* Select PLL as system clock source and configure the HCLK, PCLK1
    and PCLK2 clocks dividers */
    clkinitstruct.ClockType = (RCC_CLOCKTYPE_SYSCLK |
                           RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1
                           | RCC_CLOCKTYPE_PCLK2);
    clkinitstruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    clkinitstruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    clkinitstruct.APB2CLKDivider = RCC_HCLK_DIV1;
    clkinitstruct.APB1CLKDivider = RCC_HCLK_DIV2;
    if (HAL_RCC_ClockConfig(&clkinitstruct, FLASH_LATENCY_2)!= HAL_OK)
        {
        /* Initialization Error */
```

(下页继续)

(续上页)

```
        while (1);
    }
}
```

14.2.2 其他时钟

通过对系统时钟设置的讲解，整个时钟树我们已经把握的有六七成，剩下的时钟部分我们讲解几个重要的。

14.2.2.1 A、USB 时钟

USB 时钟是由 PLLCLK 经过 USB 预分频器得到，分频因子可以是：[1,1.5]，具体的由时钟配置寄存器 CFGR 的位 22：USBPRE 配置。USB 的时钟最高是 48M，根据分频因子反推过来算，PLLCLK 只能是 48M 或者是 72M。一般我们设置 PLLCLK=72M，USBCLK=48M。USB 对时钟要求比较高，所以 PLLCLK 只能是由 HSE 倍频得到，不能使用 HSI 倍频。

14.2.2.2 B、Cortex 系统时钟

Cortex 系统时钟由 HCLK 8 分频得到，等于 9M，Cortex 系统时钟用来驱动内核的系统定时器 SysTick，SysTick 一般用于操作系统的时钟节拍，也可以用做普通的定时。

14.2.2.3 C、ADC 时钟

ADC 时钟由 PCLK2 经过 ADC 预分频器得到，分频因子可以是 [2,4,6,8]，具体的由时钟配置寄存器 CFGR 的位 15-14：ADCPRE[1:0] 决定。很奇怪的是怎么没有 1 分频。ADC 时钟最高只能是 14M，如果采样周期设置成最短的 1.5 个周期的话，ADC 的转换时间可以达到最短的 1us。如果真要达到最短的转换时间 1us 的话，那 ADC 的时钟就得是 14M，反推 PCLK2 的时钟只能是：28M、56M、84M、112M，鉴于 PCLK2 最高是 72M，所以只能取 28M 和 56M。

14.2.2.4 D、RTC 时钟、独立看门狗时钟

RTC 时钟可由 HSE/128 分频得到，也可由低速外部时钟信号 LSE 提供，频率为 32.768KHZ，也可由低速内部时钟信号 LSI 提供，具体选用哪个时钟由备份域控制寄存器 BDCR 的位 9-8：RTCSEL[1:0] 配置。独立看门狗的时钟由 LSI 提供，且只能是由 LSI 提供，LSI 是低速的内部时钟信号，频率为 30~60KHZ 直接不等，一般取 40KHZ。

14.2.2.5 E、MCO 时钟输出

MCO 是 microcontroller clock output 的缩写，是微控制器时钟输出引脚，在 STM32 F1 系列中由 PA8 复用所得，主要作用是可以对外提供时钟，相当于一个有源晶振。MCO 的时钟来源可以是：PLLCLK/2、HSI、HSE、SYSCLK，具体选哪个由时钟配置寄存器 CFGR 的位 26-24：MCO[2:0] 决定。除了对外提供时钟这个作用之外，我们还可以通过示波器监控 MCO 引脚的时钟输出来验证我们的系统时钟配置是否正确。

14.3 配置系统时钟实验

14.3.1 使用 HSE

一般情况下，我们都是使用 HSE，然后 HSE 经过 PLL 倍频之后作为系统时钟。通常的配置是：HSE=8M，PLL 的倍频因子为：9，系统时钟就设置成:SYSCLK = 8M * 9 = 72M。使用 HSE，系统时钟最高为 72M，这个是官方推荐的最高的稳定时钟，如果你想铤而走险，也可以超频，超频最高能到 128M。当程序来到 main 函数之前，启动文件：statup_stm32f103.s 已经调用 SystemInit() 函数把系统时钟初始化成 72MHZ，SystemInit() 在库文件：system_stm32f103xe.c 中定义。如果我们想把系统时钟设置低一点或者超频的话，可以修改底层的库文件，但是为了维持库的完整性，我们可以根据时钟树的流程自行写一个。

14.3.2 使用 HSI

当 HSE 故障的时候，如果 PLL 的时钟来源是 HSE，那么当 HSE 故障的时候，不仅 HSE 不能使用，连 PLL 也会被关闭，这个时候系统会自动切换 HSI 作为系统时钟，此时 SYSCLK=HSI=8M，如果没有开启 CSS 和 CSS 中断的话，那么整个系统就只能在低速率运行，这是系统跟瘫痪没什么两样。如果开启了 CSS 功能的话，那么可以当 HSE 故障时，在 CSS 中断里面采取补救措施，使用 HSI，并把系统时钟设置为更高的频率，最高是 64M，64M 的频率足够一般的外设使用，如：ADC 、SPI、I2C 等。但是这里就又有一个问题了，原来 SYSCLK=72M，现在因为故障改成 64M，那么那些外设的时钟肯定被改变了，那么外设工作就会被打乱，那我们是不是在设置 HSI 时钟的时候，也重新调整外设总线的分频因子，即 AHB，APB2 和 APB1 的分频因子，使外设的时钟达到跟 HSE 没有故障之前一样。但是这个也不是最保障的办法，毕竟不能一直使用 HSI，所以当 HSE 故障时还是要采取报警措施。

还有一种情况是，有些用户不想用 HSE，想用 HSI，但是又不知道怎么用 HSI 来设置系统时钟，因为调用库函数都是使用 HSE，下面我们给出个使用 HSI 配置系统时钟例子，起个抛砖引玉的作用。

14.3.3 硬件设计

1. RCC
2. LED 一个

RCC 是单片机内部资源，不需要外部电路。通过 LED 闪烁的频率来直观的判断不同系统时钟频率对软件延时的效果。

14.3.4 软件设计

我们编写两个 RCC 驱动文件，bsp_clkconfig.h 和 bsp_clkconfig.c，用来存放 RCC 系统时钟配置函数。

14.3.4.1 编程要点

编程要点对应着时钟树图中的序号。

1、开启 HSE/HSI ，并等待 HSE/HSI 稳定

2、设置 AHB、APB2、APB1 的预分频因子

3、设置 PLL 的时钟来源，和 PLL 的倍频因子，设置各种频率主要就是在这里设置

4、开启 PLL，并等待 PLL 稳定

5、把 PLLCK 切换为系统时钟 SYSCLK

6、读取时钟切换状态位，确保 PLLCLK 被选为系统时钟

14.3.4.2 代码分析

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。

14.3.4.2.1 使用 HSE 配置系统时钟

列表 2: 代码 15-2 HSE 作为系统时钟来源

```
void  HSE_SetSysClock(void)
{
    RCC_ClkInitTypeDef clkinitstruct = {0};
    RCC_OscInitTypeDef oscinitstruct = {0};

    /* 使能 HSE，并以 HSE 作为 PLL 时钟源 */
    oscinitstruct.OscillatorType  = RCC_OSCILLATORTYPE_HSE;
    oscinitstruct.HSEState        = RCC_HSE_ON;
    oscinitstruct.HSEPredivValue  = RCC_HSE_PREDIV_DIV1;
    oscinitstruct.PLL.PLLState    = RCC_PLL_ON;
```

(下页继续)

(续上页)

```
    oscinitstruct.PLL.PLLSource    = RCC_PLLSOURCE_HSE;
    oscinitstruct.PLL.PLLMUL       = RCC_PLL_MUL9;
    if (HAL_RCC_OscConfig(&oscinitstruct)!= HAL_OK) {
        /* 初始化错误 */
        while (1);
    }

    /* 选择 PLL 作为系统时钟源并配置 HCLK,
    PCLK1 和 PCLK2 分频系数 */
    clkinitstruct.ClockType = (RCC_CLOCKTYPE_SYSCLK |
                            RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1
                            | RCC_CLOCKTYPE_PCLK2);
    clkinitstruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    clkinitstruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    clkinitstruct.APB2CLKDivider = RCC_HCLK_DIV1;
    clkinitstruct.APB1CLKDivider = RCC_HCLK_DIV2;
    if (HAL_RCC_ClockConfig(&clkinitstruct, FLASH_LATENCY_2)!= HAL_OK)
        {
        /* 初始化错误 */
        while (1);
    }
}
```

这个函数采用库函数编写，这些宏来源于库函数的定义，宏展开是一些 32 位的十六进制数，具体功能是配置了时钟配置寄存器 CFGR 的位 21-18 PLLMUL[3:0]，预先定义好倍频因子，方便调用。

14.3.4.2.2 使用 HSI 配置系统时钟

```
void HSI_SetSysClock(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /* 使能 HSI，并以 HSI 作为 PLL 时钟源 */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI_DIV2;
    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL16;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        while (1);
    }
    /* 选择 PLL 作为系统时钟源并配置 HCLK,
    PCLK1 和 PCLK2 分频系数 */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK| RCC_CLOCKTYPE_SYSCLK|␣
↪RCC_CLOCKTYPE_PCLK1 |RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) !=
        HAL_OK) {
        while (1);
    }
}
```

HSI 设置系统时钟函数跟 HSE 设置系统时钟函数在原理上是一样的，有一个区别的地方就是，

HSI 必须 2 分频之后才能作为 PLL 的时钟来源，所以使用 HSI 时，最大的系统时钟 SYSCLK 只能是 HSI/2*16=4*16=64MHZ。

14.3.4.2.3 软件延时

```
void Delay(__IO uint32_t nCount)
{
    for (; nCount != 0; nCount--);
}
```

软件延时函数，使用不同的系统时钟，延时时间不一样，可以通过 LED 闪烁的频率来判断。

14.3.4.2.4 MCO 输出

在 STM32F103 系列中，PA8 可以复用为 MCO 引脚，对外提供时钟输出，在 HAL 库 stm32f1xx_hal_rcc.c 中已经定义了相关函数，我们可以直接调用，用示波器监控该引脚的输出来判断我们的系统时钟是否设置正确。

列表 3: 代码 15-3 MCO GPIO 初始化

```
1 /*
2  * 初始化 MCO 引脚 PA8
3  * 在 F103 系列中 MCO 引脚只有一个，即 PA8，在 F4 系列中，MCO 引脚有两个
4  */
void HAL_RCC_MCOConfig(uint32_t RCC_MCOx, uint32_t RCC_MCOSource,
                       uint32_t RCC_MCODiv)
{
    GPIO_InitTypeDef gpio = {0U};

    /* Check the parameters */
    assert_param(IS_RCC_MCO(RCC_MCOx));
    assert_param(IS_RCC_MCODIV(RCC_MCODiv));
```

(下页继续)

(续上页)

```
    assert_param(IS_RCC_MCO1SOURCE(RCC_MCOSource));

    /* Prevent unused argument(s) compilation warning */
    UNUSED(RCC_MCOx);
    UNUSED(RCC_MCODiv);

    /* Configure the MCO1 pin in alternate function mode */
    gpio.Mode      = GPIO_MODE_AF_PP;
    gpio.Speed     = GPIO_SPEED_FREQ_HIGH;
    gpio.Pull      = GPIO_NOPULL;
    gpio.Pin       = MCO1_PIN;

    /* MCO1 Clock Enable */
    MCO1_CLK_ENABLE();

    HAL_GPIO_Init(MCO1_GPIO_PORT, &gpio);

    /* Configure the MCO clock source */
    __HAL_RCC_MCO1_CONFIG(RCC_MCOSource, RCC_MCODiv);
}
```

我们初始化 MCO 引脚之后，可以直接调用库函数 HAL_RCC_MCOConf() 来选择 MCO 时钟来源。

14.3.4.2.5 主函数

```
int main(void)
{
    // 使用 HSI，配置系统时钟为 72M
    HSE_SetSysClock();

    // LED 端口初始化
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();

    // MCO1 输出 PLLCLK
    HAL_RCC_MCOConfig(RCC_MCO,RCC_MCO1SOURCE_PLLCLK, RCC_MCODIV_1);

    while (1)
    {
        LED2( ON );       // 亮
        Delay(0x0FFFFF);
        LED2( OFF );      // 灭
        Delay(0x0FFFFF);
    }
}
```

在主函数中，可以调用 HSE_SetSysClock() 或者 HSI_SetSysClock() 这两个函数把系统时钟设置成各种常用的时钟，然后通过 MCO 引脚监控，或者通过 LED 闪烁的快慢体验不同的系统时钟对同一个软件延时函数的影响。

14.3.5 下载验证

把编译好的程序下载到开发板，可以看到设置不同的系统时钟时，LED 闪烁的快慢不一样。更精确的数据我们可以用示波器监控 MCO 引脚看到。

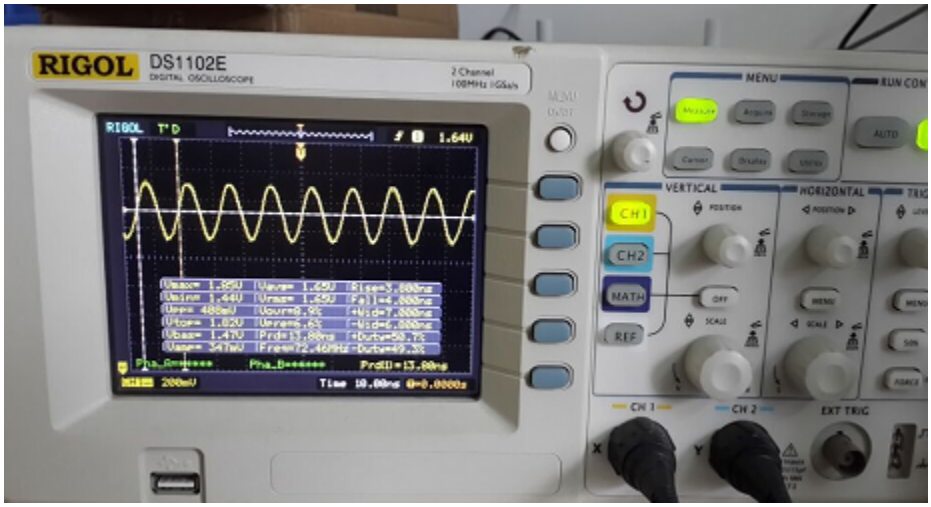


图 15-2 MCO=SYSCLK=72M

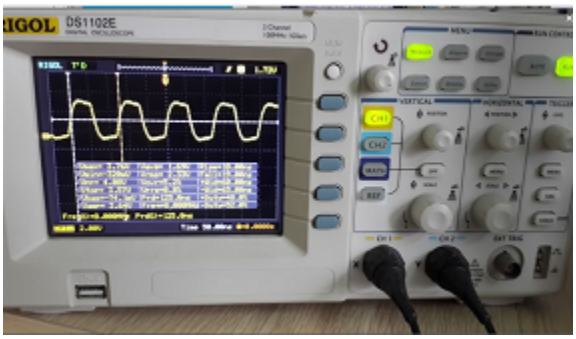


图 15-3 MCO=HSI=8M

第 15 章　STM32 中断应用概览

本章参考资料《STM32F10X-中文参考手册》、《Cortex-M3 内核编程手册》-4.3 章节：NVIC 和 4.4 章节：SCB—4.4.5 的 AIRCR。

STM32 中断非常强大，每个外设都可以产生中断，所以中断的讲解放在哪一个外设里面去讲都不合适，这里单独抽出一章来做一个总结性的介绍，这样在其他章节涉及到中断部分的知识我们就不用费很大的篇幅去讲解，只要示意性带过即可。

本章如无特别说明，异常就是中断，中断就是异常，请不要刻意钻牛角尖较劲。

15.1 异常类型

F103 在内核水平上搭载了一个异常响应系统，支持为数众多的系统异常和外部中断。其中系统异常有 8 个（如果把 Reset 和 HardFault 也算上的话就是 10 个），外部中断有 60 个。除了个别异常的优先级被定死外，其它异常的优先级都是可编程的。有关具体的系统异常和外部中断可在 HAL 库文件 stm32f1xx.h 这个头文件查询到，在 IRQn_Type 这个结构体里面包含了 F103 系列全部的异常声明。

表格 16-1 F103 系统异常清单

编号	优先级	优先级类型	名称	说明	地址
	•	•	•	保留（实际存的是 MSP 地址）	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) 连接到 NMI 向量	0X0000 0008
	-1	固定	HardFault	所有类型的错误	0X0000 000C
	0	可编程	MemManage	存储器管理	0X0000 0010
	1	可编程	BusFault	预取指失败，存储器访问失败	0X0000 0014
	2	可编程	UsageFault	未定义的指令或非法状态	0X0000 0018
	•	•	•	保留	0X0000 001C-0X0000 002B
	3	可编程	SVCall	通过 SWI 指令调用的系统服务	0X0000 002C
	4	可编程	Debug Monitor	调试监控器	0X0000 0030
	•	•	•	保留	0X0000 0034
	5	可编程	PendSV	可挂起的系统服务	0X0000 0038
	6	可编程	SysTick	系统嘀嗒定	0X0000 003C

表格 16-2 F103 外部中断清单

编号	优先级	优先级类型	名称	说明	地址
0	7	可编程	WWDG	窗口看门狗中断	0X00000040
1	8	可编程	PVD	连到 EXTI 的电源电压检测 (PVD) 中断	0X00000044
2	9	可编程	TAMPER	侵入检测中断	0X00000048
中间部分省略，详情请参考《STM32 中文参考手册》第九章-中断和事件-向量表部分					
57	64	可编程	DMA2 通道 2	DMA2 通道 2 中断	0X00000124
58	65	可编程	DMA2 通道 3	DMA2 通道 3z 中断	0X00000128
59	66	可编程	DMA2 通道 4_5	DMA2 通道 4 和通道 5 中断	0X0000012C

15.2 NVIC 简介

在讲如何配置中断优先级之前，我们需要先了解下 NVIC。NVIC 是嵌套向量中断控制器，控制着整个芯片中断相关的功能，它跟内核紧密耦合，是内核里面的一个外设。但是各个芯片厂商在设计芯片的时候会对 Cortex-M3 内核里面的 NVIC 进行裁剪，把不需要的部分去掉，所以说 STM32 的 NVIC 是 Cortex-M3 的 NVIC 的一个子集。

15.2.1 NVIC 寄存器简介

在固件库中，NVIC 的结构体定义可谓是颇有远虑，给每个寄存器都预留了很多位，恐怕为的是日后扩展功能。不过 STM32F103 可用不了这么多，只是用了部分而已，具体使用了多少可参考《Cortex-M3 内核编程手册》-4.3.11:NVIC 寄存器映射。

列表 1: 代码 16-1 NVIC 结构体定义，来自固件库头文件：core_cm3.h

```
typedef struct {
    __IOM uint32_t ISER[8U];              // 中断使能寄存器
    uint32_t RESERVED0[24U];
    __IOM uint32_t ICER[8U];              // 中断清除寄存器
    uint32_t RSERVED1[24U];
    __IOM uint32_t ISPR[8U];              // 中断使能悬起寄存器
    uint32_t RESERVED2[24U];
    __IOM uint32_t ICPR[8U];              // 中断清除悬起寄存器
    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 wide)
    uint32_t RESERVED5[644U];
    __OM  uint32_t STIR;                  // 软件触发中断寄存器
}  NVIC_Type;
```

在配置中断的时候我们一般只用 ISER、ICER 和 IP 这三个寄存器，ISER 用来使能中断，ICER 用来失能中断，IP 用来设置中断优先级。

15.2.2 NVIC 中断配置固件库

固件库文件 core_cm3.h 的最后，还提供了 NVIC 的一些函数，这些函数遵循 CMSIS 规则，只要是 Cortex-M3 的处理器都可以使用，具体如下：

表格 16-3 符合 CMSIS 标准的 NVIC 库函数

NVIC 库函数	描述
void NVIC_EnableIRQ(IRQn_Type IRQn)	使能中断
void NVIC_DisableIRQ(IRQn_Type IRQn)	失能中断
void NVIC_SetPendingIRQ(IRQn_Type IRQn)	设置中断悬起位
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)	清除中断悬起位
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)	获取悬起中断编号
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	设置中断优先级
uint32_t NVIC_GetPriority(IRQn_Type IRQn)	获取中断优先级
void NVIC_SystemReset(void)	系统复位

这些库函数我们在编程的时候用的都比较少，甚至基本都不用。在配置中断的时候我们还有更简洁的方法，请看中断编程小节。

15.3 优先级的定义

15.3.1 优先级定义

在 NVIC 有一个专门的寄存器：中断优先级寄存器 NVIC_IPRx，用来配置外部中断的优先级，IPR 宽度为 8bit，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。但是绝大多数 CM3 芯片都会精简设计，以致实际上支持的优先级数减少，在 F103 中，只使用了高 4bit，如下所示：

表格 16-4 F103 使用 4bit 表达优先级

bit7 bit6 bit5 bit4	bit3 bit2 bit1 bit0
用于表达优先级	未使用，读回为 0

用于表达优先级的这 4bit，又被分组成抢占优先级和子优先级。如果有多个中断同时响应，抢占优先级高的就会抢占抢占优先级低的优先得到执行，如果抢占优先级相同，就比较子优先级。如果抢占优先级和子优先级都相同的话，就比较他们的硬件中断编号，编号越小，优先级越高。

15.3.2 优先级分组

优先级的分组由内核外设 SCB 的应用程序中断及复位控制寄存器 AIRCR 的 PRIGROUP[10:8] 位决定，F103 分为了 5 组，具体如下：主优先级 = 抢占优先级

表 16-5

PRIGROUP[2:0]	中断优先级值 PRI_N[7:4]			级数	
	二进制点	主优先级位	子优先级位	主优先级	子优先级
0b 011	0b xxxx	[7:4]	None	16	None
0b 100	0b xxx.y	[7:5]	[4]	8	2
0b 101	0b xx.yy	[7:6]	[5:4]	4	4
0b 110	0b x.yyy	[7]	[6:4]	2	9
0b 111	0b .yyyy	None	[7:4]	None	16

设置优先级分组可调用库函数 HAL_NVIC_SetPriority() 实现，有关 NVIC 中断相关的库函数都在库文件 stm32f1xx_hal_cortex.c 和 stm32f1xx_hal_cortex.h 中。

列表 2: 代码 16-2 中断优先级分组库函数 HAL_NVIC_SetPriority()

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority,
                          uint32_t SubPriority)
{
    uint32_t prioritygroup = 0x00U;

    /* Check the parameters */
```

(下页继续)

(续上页)

```
    assert_param(IS_NVIC_SUB_PRIORITY(SubPriority));
    assert_param(IS_NVIC_PREEMPTION_PRIORITY(PreemptPriority));

    prioritygroup = NVIC_GetPriorityGrouping();

    NVIC_SetPriority(IRQn, NVIC_EncodePriority(prioritygroup,
                     PreemptPriority, SubPriority));
}
```

15.4 中断编程

在配置每个中断的时候一般有 3 个编程要点：

1. 使能外设某个中断，这个具体由每个外设的相关中断使能位控制。比如串口有发送完成中断，接收完成中断，这两个中断都由串口控制寄存器的相关中断使能位控制。
2. 配置 EXTI 中断源、配置中断优先级。

列表 3: 代码 16-3 NVIC 初始化结构体

```
/* 配置 EXTI 中断源、配置中断优先级 */
HAL_NVIC_SetPriority(IRQn,PreemptPriority, SubPriority)
```

1. IRQn：用来设置中断源，不同的中断中断源不一样，且不可写错，即使写错了程序也不会报错，只会导致不响应中断。具体的成员配置可参考 stm32f103xe.h 头文件里面的 IRQn_Type 结构体定义，这个结构体包含了所有的中断源。

列表 4: 代码 16-4 IRQn_Type 中断源结构体

```
typedef enum IRQn {
    //Cortex-M3 处理器异常编号
    NonMaskableInt_IRQn       = -14,
```

(下页继续)

(续上页)

```
    MemoryManagement_IRQn     = -12,
    BusFault_IRQn             = -11,
    UsageFault_IRQn           = -10,
    SVCall_IRQn               = -5,
    DebugMonitor_IRQn         = -4,
    PendSV_IRQn               = -2,
    SysTick_IRQn              = -1,
    //STM32 外部中断编号
    WWDG_IRQn                 = 0,
    PVD_IRQn                  = 1,
    TAMP_STAMP_IRQn           = 2,

    // 限于篇幅，中间部分代码省略，具体的可查看库文件 stm32f103xe.h

    DMA2_Channel2_IRQn        = 57,
    DMA2_Channel3_IRQn        = 58,
    DMA2_Channel4_5_IRQn      = 59
} IRQn_Type;
```

2. PreemptionPriority：抢占优先级，具体的值要根据优先级分组来确定，。

3. SubPriority：子优先级，具体的值要根据优先级分组来确定。

3. 编写中断服务函数

在启动文件 startup_stm32f103xe.s 中我们预先为每个中断都写了一个中断服务函数，只是这些中断函数都是为空，为的只是初始化中断向量表。实际的中断服务函数都需要我们重新编写，为了方便管理我们把中断服务函数统一写在 stm32f1xx_it.c 这个库文件中。

关于中断服务函数的函数名必须跟启动文件里面预先设置的一样，如果写错，系统就在中断向量表中找不到中断服务函数的入口，直接跳转到启动文件里面预先写好的空函数，并且在里面无限循环，实现不了中断。

第 16 章　EXTI—外部中断/事件控制器

本章参考资料：《STM32F10X-中文参考手册》中断和事件章节和 8.4 章节 AFIO 寄存器描述。

上一章节我们已经详细介绍了 NVIC，对 STM32F10x 系列中断管理系统有个全局的了解，我们这章的内容是 NVIC 的实例应用，也是 STM32F10x 控制器非常重要的一个资源。学习本章时，配合《STM32F10X-中文参考手册》中断和事件章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

特别说明，本书内容是以 STM32F10X 系列控制器资源讲解。

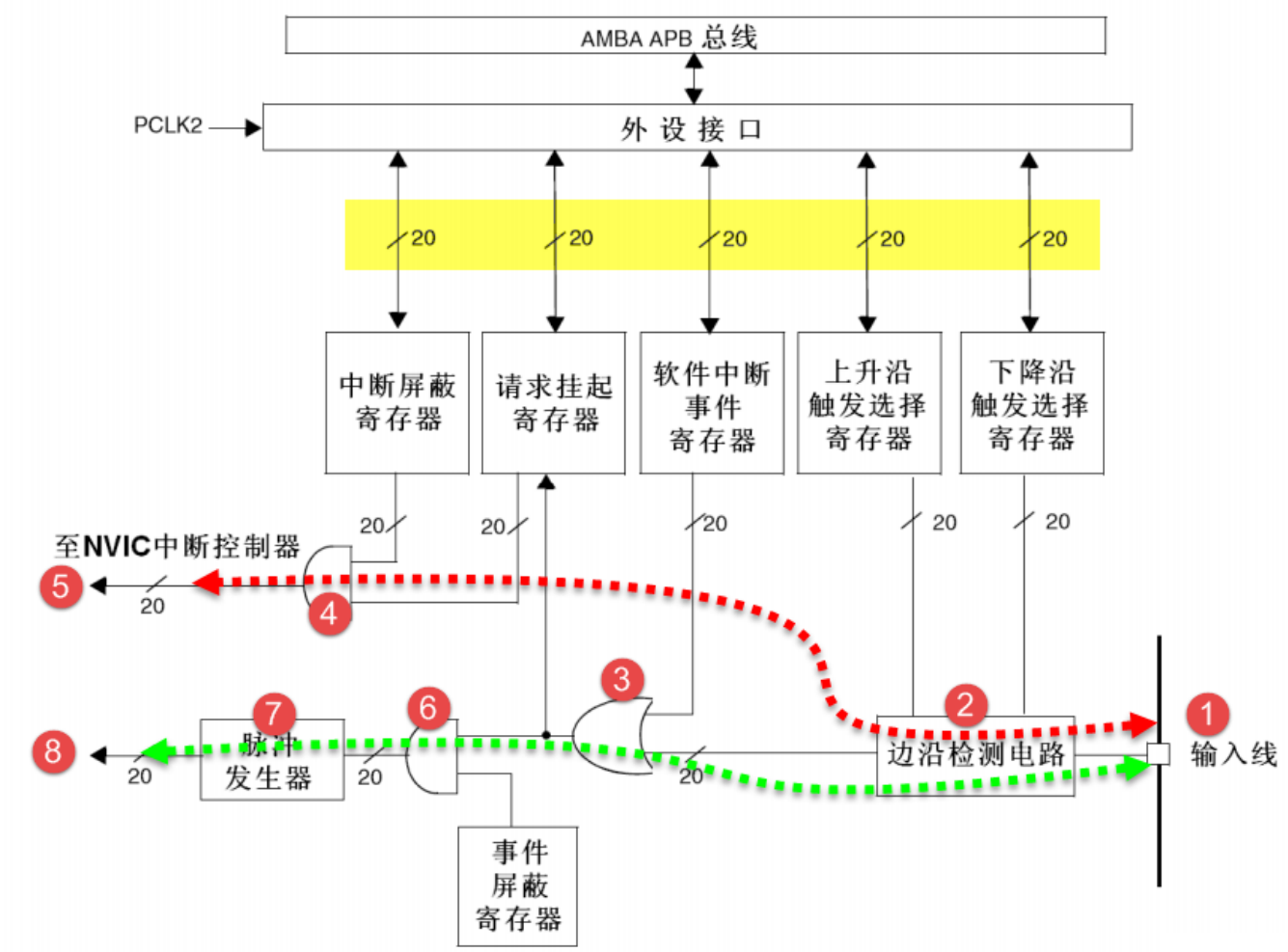
16.1 EXTI 简介

EXTI（External interrupt/event controller）—外部中断/事件控制器，管理了控制器的 20 个中断/事件线。每个中断/事件线都对应有一个边沿检测器，可以实现输入信号的上升沿检测和下降沿的检测。EXTI 可以实现对每个中断/事件线进行单独配置，可以单独配置为中断或者事件，以及触发事件的属性。

16.2 EXTI 功能框图

EXTI 的功能框图包含了 EXTI 最核心内容，掌握了功能框图，对 EXTI 就有一个整体的把握，在编程时思路就非常清晰。EXTI 功能框图见图 *17_1*。

在图 *17_1* 可以看到很多在信号线上打一个斜杠并标注“20”字样，这个表示在控制器内部类似的信号线路有 20 个，这与 EXTI 总共有 20 个中断/事件线是吻合的。所以我们只要明白其中一个的原理，那其他 19 个线路原理也就知道了。



EXTI 可分为两大部分功能，一个是产生中断，另一个是产生事件，这两个功能从硬件上就有所不同。

首先我们来看图 *17_1* 中红色虚线指示的电路流程。它是一个产生中断的线路，最终信号流入到 NVIC 控制器内。

编号 1 是输入线，EXTI 控制器有 19 个中断/事件输入线，这些输入线可以通过寄存器设置为任意一个 GPIO，也可以是一些外设的事件，这部分内容我们将在后面专门讲解。输入线一般是存在电平变化的信号。

编号 2 是一个边沿检测电路，它会根据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) 对应位的设置来控制信号触发。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 给编号 3 电路，否则输出无效信号 0。而 EXTI_RTSR 和

EXTI_FTSR 两个寄存器可以控制器需要检测哪些类型的电平跳变过程，可以是只有上升沿触发、只有下降沿触发或者上升沿和下降沿都触发。

编号 3 电路实际就是一个或门电路，它一个输入来自编号 2 电路，另外一个输入来自软件中断事件寄存器 (EXTI_SWIER)。EXTI_SWIER 允许我们通过程序控制就可以启动中断/事件线，这在某些地方非常有用。我们知道或门的作用就是有 1 就为 1，所以这两个输入随便一个有有效信号 1 就可以输出 1 给编号 4 和编号 6 电路。

编号 4 电路是一个与门电路，它一个输入是编号 3 电路，另外一个输入来自中断屏蔽寄存器 (EXTI_IMR)。与门电路要求输入都为 1 才输出 1，导致的结果是如果 EXTI_IMR 设置为 0 时，那不管编号 3 电路的输出信号是 1 还是 0，最终编号 4 电路输出的信号都为 0；如果 EXTI_IMR 设置为 1 时，最终编号 4 电路输出的信号才由编号 3 电路的输出信号决定，这样我们可以简单的控制 EXTI_IMR 来实现是否产生中断的目的。编号 4 电路输出的信号会被保存到挂起寄存器 (EXTI_PR) 内，如果确定编号 4 电路输出为 1 就会把 EXTI_PR 对应位置 1。

编号 5 是将 EXTI_PR 寄存器内容输出到 NVIC 内，从而实现系统中断事件控制。

接下来我们来看看绿色虚线指示的电路流程。它是一个产生事件的线路，最终输出一个脉冲信号。

产生事件线路是在编号 3 电路之后与中断线路有所不同，之前电路都是共用的。编号 6 电路是一个与门，它一个输入来自编号 3 电路，另外一个输入来自事件屏蔽寄存器 (EXTI_EMR)。如果 EXTI_EMR 设置为 0 时，那不管编号 3 电路的输出信号是 1 还是 0，最终编号 6 电路输出的信号都为 0；如果 EXTI_EMR 设置为 1 时，最终编号 6 电路输出的信号才由编号 3 电路的输出信号决定，这样我们可以简单的控制 EXTI_EMR 来实现是否产生事件的目的。

编号 7 是一个脉冲发生器电路，当它的输入端，即编号 6 电路的输出端，是一个有效信号 1 时就会产生一个脉冲；如果输入端是无效信号就不会输出脉冲。

编号 8 是一个脉冲信号，就是产生事件的线路最终的产物，这个脉冲信号可以给其他外设电路使用，比如定时器 TIM、模拟数字转换器 ADC 等等，这样的脉冲信号一般用来触发 TIM 或者 ADC 开始转换。

产生中断线路目的是把输入信号输入到 NVIC，进一步会运行中断服务函数，实现功能，这样是软件级的。而产生事件线路目的就是传输一个脉冲信号给其他外设使用，并且是电路级别的信号传输，属于硬件级的。

另外，EXTI 是在 APB2 总线上的，在编程时候需要注意到这点。

16.3 中断/事件线

EXTI 有 20 个中断/事件线，每个 GPIO 都可以被设置为输入线，占用 EXTI0 至 EXTI15，还有另外七根用于特定的外设事件，见表 *17_1* 。

4 根特定外设中断/事件线由外设触发，具体用法参考《STM32F10X-中文参考手册》中对外设的具体说明。

表 17-1 EXTI 中断/事件线

中断/事件线	输入源
EXTI0	PX0(X 可为 A,B,C,D,E,F,G,H,I)
EXTI1	PX1(X 可为 A,B,C,D,E,F,G,H,I)
EXTI2	PX2(X 可为 A,B,C,D,E,F,G,H,I)
EXTI3	PX3(X 可为 A,B,C,D,E,F,G,H,I)
EXTI4	PX4(X 可为 A,B,C,D,E,F,G,H,I)
EXTI5	PX5(X 可为 A,B,C,D,E,F,G,H,I)
EXTI6	PX6(X 可为 A,B,C,D,E,F,G,H,I)
EXTI7	PX7(X 可为 A,B,C,D,E,F,G,H,I)
EXTI8	PX8(X 可为 A,B,C,D,E,F,G,H,I)
EXTI9	PX9(X 可为 A,B,C,D,E,F,G,H,I)
EXTI10	PX10(X 可为 A,B,C,D,E,F,G,H,I)
EXTI11	PX11(X 可为 A,B,C,D,E,F,G,H,I)
EXTI12	PX12(X 可为 A,B,C,D,E,F,G,H,I)
EXTI13	PX13(X 可为 A,B,C,D,E,F,G,H,I)
EXTI14	PX14(X 可为 A,B,C,D,E,F,G,H,I)
EXTI15	PX15(X 可为 A,B,C,D,E,F,G,H,I)
EXTI16	PVD 输出
EXTI17	RTC 闹钟事件
EXTI18	USB 唤醒事件
EXTI19	以太网唤醒事件（只适用互联型）

EXTI0 至 EXTI15 用于 GPIO，通过编程控制可以实现任意一个 GPIO 作为 EXTI 的输入源。由表 *17_1* 可知，EXTI0 可以通过 AFIO 的外部中断配置寄存器 1(AFIO_EXTICR1) 的 EXTI0[3:0] 位选择配置为 PA0、PB0、PC0、PD0、PE0、PF0、PG0、PH0 或者 PI0，见图 *17_2*。其他 EXTI 线 (EXTI 中断/事件线) 使用配置都是类似的。

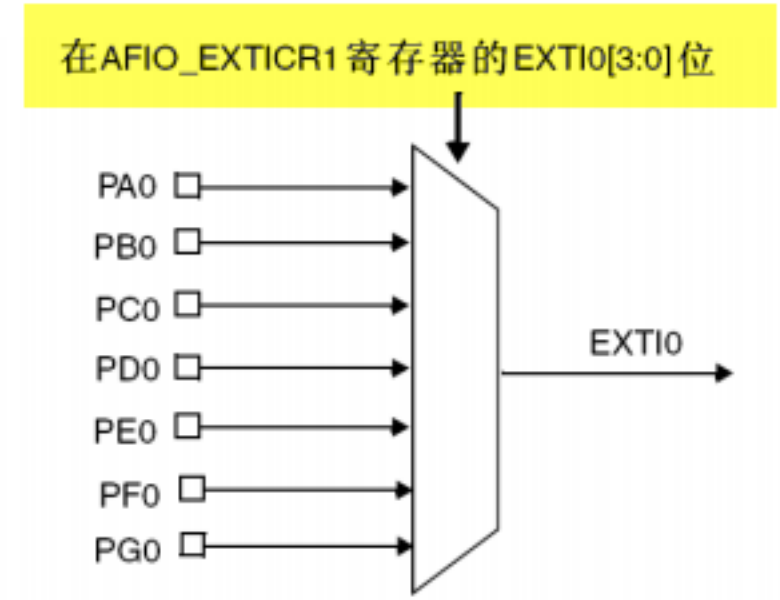


图 17-2 EXTI0 输入源选择

16.4 EXTI 初始化详解

HAL 库函数的 EXIT 初始化非常简单，只需配置好 IO 口的模式，然后配置中断源、中断优先级、使能中断。

1) HAL_NVIC_SetPriority：该函数负责 EXTI 中断/事件线选择，可选 EXTI0 至 EXTI25，可参考表 *17_1* 选择，配置优先级。

2) HAL_NVIC_EnableIRQ：该函数负责控制使能中断。

16.5 外部中断控制实验

中断在嵌入式应用中占有非常重要的地位，几乎每个控制器都有中断功能。中断对保证紧急事件得到第一时间处理是非常重要的。

我们设计使用外接的按键来作为触发源，使得控制器产生中断，并在中断服务函数中实现控制 RGB 彩灯的任务。

16.5.1 硬件设计

轻触按键在按下时会使得引脚接通，通过电路设计可以使得按下时产生电平变化，见图 *17_3a*、图 *17_3b*。

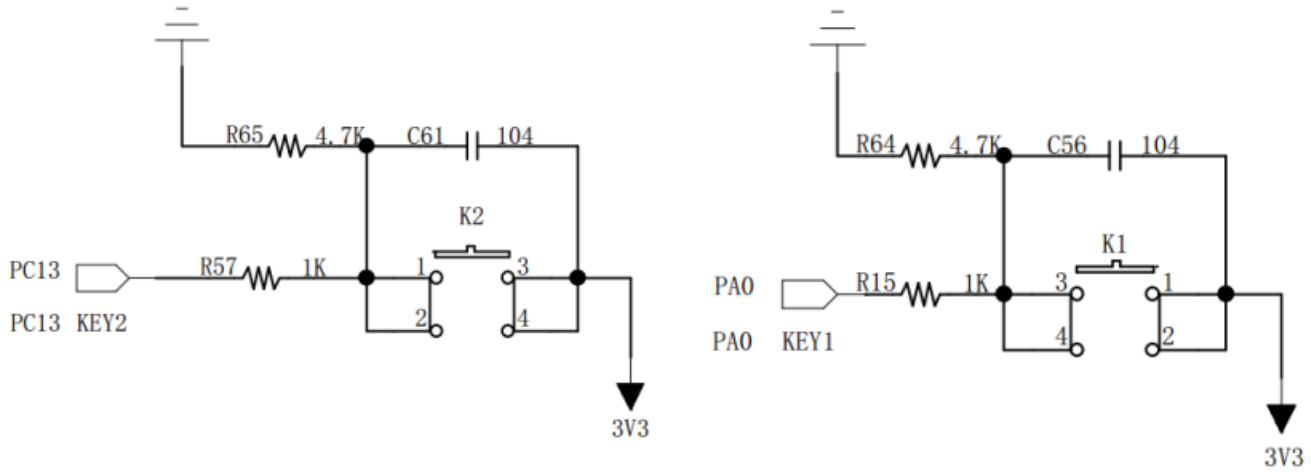


图 17-3a MINI 板、指南者和霸道开发板的按键原理图

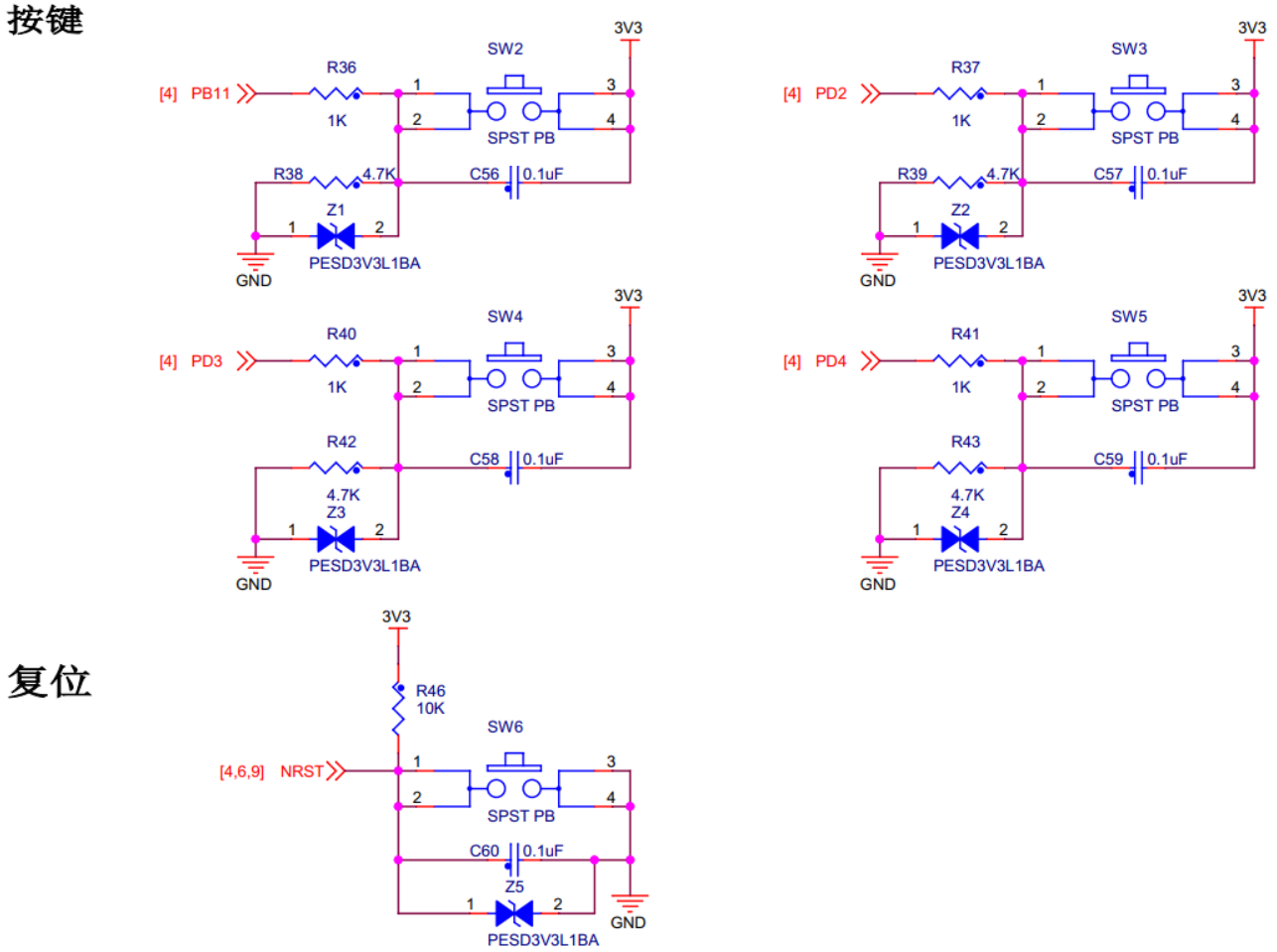


图 17-3b 拂晓电机开发板的按键原理图

16.5.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_exti.c 和 bsp_exti.h 文件用来存放 EXTI 驱动程序及相关宏定义，中断服务函数放在 stm32f1xx_it.h 文件中。

16.5.2.1 编程要点

1) 初始化用来产生中断的 GPIO；

2) 初始化 EXTI；

3) 配置 NVIC；

4) 编写中断服务函数；

16.5.2.2 软件分析

16.5.2.2.1 按键和 EXTI 宏定义

列表 1: 代码清单 17-1a MINI 板、指南者和霸道开发板的按键和 EXTI 宏定义

```
//引脚定义
/*******************************************************/
#define KEY1_INT_GPIO_PORT                GPIOA
#define KEY1_INT_GPIO_CLK_ENABLE()        __GPIOA_CLK_ENABLE();
#define KEY1_INT_GPIO_PIN                 GPIO_PIN_0
#define KEY1_INT_EXTI_IRQ                 EXTI0_IRQn
#define KEY1_IRQHandler                   EXTI0_IRQHandler

#define KEY2_INT_GPIO_PORT                GPIOC
#define KEY2_INT_GPIO_CLK_ENABLE()        __GPIOA_CLK_ENABLE();
#define KEY2_INT_GPIO_PIN                 GPIO_PIN_13
#define KEY2_INT_EXTI_IRQ                 EXTI15_10_IRQn
#define KEY2_IRQHandler                   EXTI15_10_IRQHandler
/*******************************************************/


.. code-block:: c
:caption: 代码清单 17-1b 拂晓电机开发板的按键和 EXTI 宏定义
```

(下页继续)

(续上页)

```
:name: 代码清单 17_1b

//引脚定义
/*******************************************************/
#define KEY1_INT_GPIO_PORT                GPIOA
#define KEY1_INT_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE();
#define KEY1_INT_GPIO_PIN                 GPIO_PIN_15
#define KEY1_INT_EXTI_IRQ                 EXTI15_10_IRQn
#define KEY1_IRQHandler                   EXTI15_10_IRQHandler

#define KEY2_INT_GPIO_PORT                GPIOD
#define KEY2_INT_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOD_CLK_ENABLE();
#define KEY2_INT_GPIO_PIN                 GPIO_PIN_2
#define KEY2_INT_EXTI_IRQ                 EXTI2_IRQn
#define KEY2_IRQHandler                   EXTI2_IRQHandler

#define KEY3_INT_GPIO_PORT                GPIOD
#define KEY3_INT_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOD_CLK_ENABLE();
#define KEY3_INT_GPIO_PIN                 GPIO_PIN_3
#define KEY3_INT_EXTI_IRQ                 EXTI3_IRQn
#define KEY3_IRQHandler                   EXTI3_IRQHandler

#define KEY4_INT_GPIO_PORT                GPIOD
#define KEY4_INT_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOD_CLK_ENABLE();
#define KEY4_INT_GPIO_PIN                 GPIO_PIN_4
#define KEY4_INT_EXTI_IRQ                 EXTI4_IRQn
#define KEY4_IRQHandler                   EXTI4_IRQHandler
/*******************************************************/
```

使用宏定义方法指定与电路设计相关配置，这对于程序移植或升级非常有用的。

16.5.2.2.2 EXTI 中断配置

列表 2: 代码清单 17-2a MINI 板、指南者和霸道开发板的 EXTI 中断配置（bsp_exit.c 文件）

```
void EXTI_Key_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启按键 GPIO 口的时钟 */
    KEY1_INT_GPIO_CLK_ENABLE();
    KEY2_INT_GPIO_CLK_ENABLE();

    /* 选择按键 1 的引脚 */
    GPIO_InitStructure.Pin = KEY1_INT_GPIO_PIN;
    /* 设置引脚为输入模式 */
    GPIO_InitStructure.Mode = GPIO_MODE_IT_RISING;
    /* 设置引脚不上拉也不下拉 */
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    /* 使用上面的结构体初始化按键 */
    HAL_GPIO_Init(KEY1_INT_GPIO_PORT, &GPIO_InitStructure);
    /* 配置 EXTI 中断源 到 key1 引脚、配置中断优先级 */
    HAL_NVIC_SetPriority(KEY1_INT_EXTI_IRQ, 0, 0);
    /* 使能中断 */
    HAL_NVIC_EnableIRQ(KEY1_INT_EXTI_IRQ);

    /* 选择按键 2 的引脚 */
    GPIO_InitStructure.Pin = KEY2_INT_GPIO_PIN;
    /* 其他配置与上面相同 */
    HAL_GPIO_Init(KEY2_INT_GPIO_PORT, &GPIO_InitStructure);
    /* 配置 EXTI 中断源 到 key1 引脚、配置中断优先级 */
    HAL_NVIC_SetPriority(KEY2_INT_EXTI_IRQ, 0, 0);
    /* 使能中断 */
```

(下页继续)

(续上页)

```
    HAL_NVIC_EnableIRQ(KEY2_INT_EXTI_IRQ);
}
```

列表 3: 代码清单 17-2b 拂晓电机开发板的 EXTI 中断配置（bsp_exit.c 文件）

```
void EXTI_Key_Config(void)
{
GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启按键 GPIO 口的时钟 */
    KEY1_INT_GPIO_CLK_ENABLE();
    KEY2_INT_GPIO_CLK_ENABLE();
    KEY3_INT_GPIO_CLK_ENABLE();
    KEY4_INT_GPIO_CLK_ENABLE();

    /* 选择按键 1 的引脚 */
    GPIO_InitStructure.Pin = KEY1_INT_GPIO_PIN;
    /* 设置引脚为输入模式 */
    GPIO_InitStructure.Mode = GPIO_MODE_IT_RISING;
    /* 设置引脚不上拉也不下拉 */
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    /* 使用上面的结构体初始化按键 */
    HAL_GPIO_Init(KEY1_INT_GPIO_PORT, &GPIO_InitStructure);
    /* 配置 EXTI 中断源 到 key1 引脚、配置中断优先级 */
    HAL_NVIC_SetPriority(KEY1_INT_EXTI_IRQ, 0, 0);
    /* 使能中断 */
    HAL_NVIC_EnableIRQ(KEY1_INT_EXTI_IRQ);

    /* 选择按键 2 的引脚 */
    GPIO_InitStructure.Pin = KEY2_INT_GPIO_PIN;
    /* 其他配置与上面相同 */
```

(下页继续)

(续上页)

```
    HAL_GPIO_Init(KEY2_INT_GPIO_PORT, &GPIO_InitStructure);
    /* 配置 EXTI 中断源 到 key1 引脚、配置中断优先级 */
    HAL_NVIC_SetPriority(KEY2_INT_EXTI_IRQ, 0, 0);
    /* 使能中断 */
    HAL_NVIC_EnableIRQ(KEY2_INT_EXTI_IRQ);

        /* 选择按键 3 的引脚 */
    GPIO_InitStructure.Pin = KEY3_INT_GPIO_PIN;
    /* 其他配置与上面相同 */
    HAL_GPIO_Init(KEY3_INT_GPIO_PORT, &GPIO_InitStructure);
    /* 配置 EXTI 中断源 到 key1 引脚、配置中断优先级 */
    HAL_NVIC_SetPriority(KEY3_INT_EXTI_IRQ, 0, 0);
    /* 使能中断 */
    HAL_NVIC_EnableIRQ(KEY3_INT_EXTI_IRQ);

            /* 选择按键 2 的引脚 */
    GPIO_InitStructure.Pin = KEY4_INT_GPIO_PIN;
    /* 其他配置与上面相同 */
    HAL_GPIO_Init(KEY4_INT_GPIO_PORT, &GPIO_InitStructure);
    /* 配置 EXTI 中断源 到 key1 引脚、配置中断优先级 */
    HAL_NVIC_SetPriority(KEY4_INT_EXTI_IRQ, 0, 0);
    /* 使能中断 */
    HAL_NVIC_EnableIRQ(KEY4_INT_EXTI_IRQ);
}
```

首先，使用 GPIO_InitTypeDef 结构体定义用于 GPIO 初始化配置的变量，关于这个结构体前面都已经做了详细的讲解。

使用 GPIO 之前必须开启 GPIO 端口的时钟；

调用 HAL_NVIC_SetPriority 和 HAL_NVIC_EnableIRQ 函数完成对按键 1、按键 2 优先级配置并使能中断通道。

作为中断/时间输入线把 GPIO 配置为中断上升沿触发模式，这里不使用上拉或下拉，有外部电路

完全决定引脚的状态。

我们的目的是产生中断，执行中断服务函数，EXTI 选择中断模式，按键 1 使用下降沿触发方式，并使能 EXTI 线。

按键 2 基本上采用与按键 1 相关参数配置，只是改为上升沿触发方式。

16.5.2.2.3 EXTI 中断服务函数

列表 4: 代码清单 17-3 EXTI 中断服务函数

```
void KEY1_IRQHandler(void)
{
    //确保是否产生了 EXTI Line 中断
    if (__HAL_GPIO_EXTI_GET_IT(KEY1_INT_GPIO_PIN) != RESET) {
        // LED1 取反
        LED1_TOGGLE;
        //清除中断标志位
        __HAL_GPIO_EXTI_CLEAR_IT(KEY1_INT_GPIO_PIN);
    }
}

void KEY2_IRQHandler(void)
{
    //确保是否产生了 EXTI Line 中断
    if (__HAL_GPIO_EXTI_GET_IT(KEY2_INT_GPIO_PIN) != RESET) {
        // LED2 取反
        LED2_TOGGLE;
        //清除中断标志位
        __HAL_GPIO_EXTI_CLEAR_IT(KEY2_INT_GPIO_PIN);
    }
}
```

(下页继续)

(续上页)

```
/* 注：KEY3_IRQHandler 和 KEY4_IRQHandler 这两个不是拂晓电机开发板的不用加 */

void KEY3_IRQHandler(void)
{
//确保是否产生了 EXTI Line 中断
    if(__HAL_GPIO_EXTI_GET_IT(KEY3_INT_GPIO_PIN) != RESET)
    {
        // LED2 取反
        LED3_TOGGLE;
    //清除中断标志位
        __HAL_GPIO_EXTI_CLEAR_IT(KEY3_INT_GPIO_PIN);
    }
}

void KEY4_IRQHandler(void)
{
//确保是否产生了 EXTI Line 中断
    if(__HAL_GPIO_EXTI_GET_IT(KEY4_INT_GPIO_PIN) != RESET)
    {
        // LED2 取反
        LED4_TOGGLE;
    //清除中断标志位
        __HAL_GPIO_EXTI_CLEAR_IT(KEY4_INT_GPIO_PIN);
    }
}
```

当中断发生时，对应的中断服务函数就会被执行，我们可以在中断服务函数实现一些控制。

一般为确保中断确实发生，我们会在中断服务函数调用中断标志位状态读取函数读取外设中断标志位并判断标志位状态。

__HAL_GPIO_EXTI_GET_IT 函数用来获取 EXTI 的中断标志位状态，如果 EXTI 线有中断发生

函数返回“SET”否则返回“RESET”。实际上，__HAL_GPIO_EXTI_GET_IT 函数是通过读取 EXTI_PR 寄存器值来判断 EXTI 线状态的。

按键 1 的中断服务函数我们让 LED1 翻转其状态，按键 2 的中断服务函数我们让 LED2 翻转其状态。执行任务后需要调用 __HAL_GPIO_EXTI_CLEAR_IT 函数清除 EXTI 线的中断标志位。

16.5.2.2.4 主函数

列表 5: 代码清单 17-4 主函数

```
int main(void)
{
    /* 系统时钟初始化成 72MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();

    /* 初始化 EXTI 中断，按下按键会触发中断，
    *  触发中断会进入 stm32f7xx_it.c 文件中的函数
    *
    *  KEY1_IRQHandler 和 KEY2_IRQHandler，处理中断，反转 LED 灯。
    *  KEY3_IRQHandler 和 KEY4_IRQHandler 只有拂晓电机开发板才有
    */
    EXTI_Key_Config();

    /* 等待中断，由于使用中断方式，CPU 不用轮询按键
    */
    while (1) {
    }
}
```

主函数非常简单，只有三个任务函数。SystemClock_Config 初始化系统时钟，LED_GPIO_Config 函数定义在 bsp_led.c 文件内，完成 RGB 彩灯的 GPIO 初始化配置。EXTI_Key_Config 函数完成两个

按键的 GPIO 和 EXTI 配置。

16.5.3 下载验证

保证开发板相关硬件连接正确，把编译好的程序下载到开发板。此时 RGB 彩色灯是暗的，如果我们按下开发板上的按键 1，RGB 彩灯变亮，再按下按键 1，RGB 彩灯又变暗；如果我们按下开发板上的按键 2 并弹开，RGB 彩灯变亮，再按下开发板上的 KEY2 并弹开，RGB 彩灯又变暗。按键按下表示上升沿，按键弹开表示下降沿，这跟我们软件设置是一样的。

第 17 章　SysTick—系统定时器

本章参考资料《Cortex-M3 内核编程手册》-4.5 章节 SysTick Timer(STK)，和 4.48 章节 SHPRx，其中 STK 这个章节有 SysTick 的简介和寄存器的详细描述。因为 SysTick 是属于 CM3 内核的外设，有关寄存器的定义和部分库函数都在 core_CM3.h 这个头文件中实现。所以学习 SysTick 的时候可以参考这两个资料，一个是文档，一个是源码。

17.1 SysTick 简介

SysTick—系统定时器是属于 CM3 内核中的一个外设，内嵌在 NVIC 中。系统定时器是一个 24bit 的向下递减的计数器，计数器每计数一次的时间为 1/SYSCLK，一般我们设置系统时钟 SYSCLK 等于 72M。当重装载数值寄存器的值递减到 0 的时候，系统定时器就产生一次中断，以此循环往复。

因为 SysTick 是属于 CM3 内核的外设，所以所有基于 CM3 内核的单片机都具有这个系统定时器，使得软件在 CM3 单片机中可以很容易的移植。系统定时器一般用于操作系统，用于产生时基，维持操作系统的心跳。

17.2 SysTick 寄存器介绍

SysTick—系统定时器有 4 个寄存器，简要介绍如下。在使用 SysTick 产生定时的时候，只需要配置前三个寄存器，最后一个校准寄存器不需要使用。

表 18-1 SysTick 寄存器汇总

寄存器名称	寄存器描述
CTRL	SysTick 控制及状态寄存器
LOAD	SysTick 重装载数值寄存器
VAL	SysTick 当前数值寄存器
CALIB	SysTick 校准数值寄存器

表 18-2 SysTick 控制及状态寄存器

位段	名称	类型	复位值	描述
16	COUNT-FLAG	R/W	0	如果在上次读取本寄存器后，SysTick 已经计到了 0，则该位为 1。
2	CLK-SOURCE	R/W	0	时钟源选择位，0=AHB/8，1= 处理器时钟 AHB
1	TICK-INT	R/W	0	1=SysTick 倒数计数到 0 时产生 SysTick 异常请求，0= 数到 0 时无动作。也可以通过读取 COUNTFLAG 标志位来确定计数器是否递减到 0
0	EN-ABLE	R/W	0	SysTick 定时器的使能位

表 18-3 SysTick 重装载数值寄存器

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

表 18-4 SysTick 当前数值寄存器

位段	名称	类型	复位值	描述
23:0	CURRENT	R/W	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

表 18-5 SysTick 校准数值寄存器

位段	名称	类型	复位值	描述
31	NOREF	R	0	NOREF flag. Reads as zero. Indicates that a separate reference clock is provided. The frequency of this clock is HCLK/8
30	SKEW	R	1	Reads as one. Calibration value for the 1 ms inexact timing is not known because TENMS is not known. This can affect the suitability of SysTick as a software real time clock
23:0	TENMS	RS	0	Indicates the calibration value when the SysTick counter runs on HCLK max/8 as external clock. The value is product dependent, please refer to the Product Reference Manual, SysTick Calibration Value section. When HCLK is programmed at the maximum frequency, the SysTick period is 1ms. If calibration information is not known, calculate the calibration value required from the frequency of the processor clock or external clock.

系统定时器的校准数值寄存器在定时实验中不需要用到。有关各个位的描述这里引用手册里面的英文版本，比较晦涩难懂，暂时不知道这个寄存器用来干什么。有研究过的朋友可以交流，起个抛砖引玉的作用。

17.3 SysTick 定时实验

利用 SysTick 产生 1s 的时基，LED 以 1s 的频率闪烁。

17.3.1 硬件设计

SysTick 属于单片机内部的外设，不需要额外的硬件电路，剩下的只需一个 LED 灯即可。

17.3.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_SysTick.c 和 bsp_ SysTick.h 文件用来存放 SysTick 驱动程序及相关宏定义，中断服务函数放在 stm32f1xx_it.c 文件中。

17.3.2.1 编程要点

1、设置重装载寄存器的值

2、清除当前数值寄存器的值

3、配置控制与状态寄存器

17.3.2.2 代码分析

SysTick 属于内核的外设，有关的寄存器定义和库函数都在内核相关的库文件 core_cm3.h 中。

17.3.2.2.1 SysTick 配置库函数

列表 1: 代码 18-1SysTick 配置库函数

```
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
    // 不可能的重装载值，超出范围
    if ((ticks - 1UL) > SysTick_LOAD_RELOAD_Msk) {
        return (1UL);
    }

    // 设置重装载寄存器
    SysTick->LOAD  = (uint32_t)(ticks - 1UL);

    // 设置中断优先级
    NVIC_SetPriority (SysTick_IRQn, (1UL << __NVIC_PRIO_BITS) - 1UL);

    // 设置当前数值寄存器
    SysTick->VAL   = 0UL;

    // 设置系统定时器的时钟源为 AHBCLK=72M
    // 使能系统定时器中断
    // 使能定时器
    SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                     SysTick_CTRL_TICKINT_Msk   |
                     SysTick_CTRL_ENABLE_Msk;
    return (0UL);
}
```

用固件库编程的时候我们只需要调用库函数 SysTick_Config() 即可，形参 ticks 用来设置重装载寄存器的值，最大不能超过重装载寄存器的值 2^{24}，当重装载寄存器的值递减到 0 的时候产生中断，然后重装载寄存器的值又重新装载往下递减计数，以此循环往复。紧随其后设置好中断优先级，最后配置系统定时器的时钟等于 AHBCLK=72M，使能定时器和定时器中断，这样系统定时器就

配置好了，一个库函数搞定。

SysTick_Config() 库函数主要配置了 SysTick 中的三个寄存器：LOAD、VAL 和 CTRL，有关具体的部分看代码注释即可。

17.3.2.2.2 配置 SysTick 中断优先级

SysTick_Config() 库函数主要配置了 SysTick 中的三个寄存器：LOAD、VAL 和 CTRL，有关具体的部分看代码注释即可。其中还调用了函数库函数 NVIC_SetPriority() 来配置系统定时器的中断优先级，该库函数也在 core_m4.h 中定义，原型如下：

```
__STATIC_INLINE void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
{
if ((int32_t)IRQn < 0) {
SCB->SHP[(((uint32_t)(int32_t)IRQn) & 0xFUL)-4UL] =
(uint8_t)((priority << (8 - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
} else {
NVIC->IP[((uint32_t)(int32_t)IRQn)] =
(uint8_t)((priority << (8 - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
}
}
```

因为 SysTick 属于内核外设，跟普通外设的中断优先级有些区别，并没有抢占优先级和子优先级的说法。在 STM32F103 中，内核外设的中断优先级由内核 SCB 这个外设的寄存器：SHPRx（x=1.2.3）来配置。有关 SHPRx 寄存器的详细描述可参考《Cortex-M4 内核编程手册》4.4.8 章节。下面我们简单介绍下这个寄存器。

SPRH1-SPRH3 是一个 32 位的寄存器，但是只能通过字节访问，每 8 个字段控制着一个内核外设的中断优先级的配置。在 STM32F103 中，只有位 7:4 这高四位有效，低四位没有用到，所以内核外设的中断优先级可编程为：0~15，只有 16 个可编程优先级，数值越小，优先级越高。如果软件优先级配置相同，那就根据他们在中断向量表里面的位置编号来决定优先级大小，编号越小，优先级越高。

表 18-6 系统异常优先级字段

异常	字段	寄存器描述
Memory management fault	PRI_4	SHPR1
Bus fault	PRI_5	
Usage fault	PRI_6	
SVCall	PRI_11	SHPR2
PendSV	PRI_14	SHPR3
SysTick	PRI_15	

如果要修改内核外设的优先级，只需要修改下面三个寄存器对应的某个字段即可。

System handler priority register 1 (SHPR1)

Address offset: 0x18

Reset value: 0x0000 0000

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								PRI_6[7:4]				PRI_6[3:0]			
								rw	rw	rw	rw	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRI_5[7:4]				PRI_5[3:0]				PRI_4[7:4]				PRI_4[7:4]			
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r

图 18-1 SHPR1 寄存器

System handler priority register 2 (SHPR2)

Address offset: 0x1C

Reset value: 0x0000 0000

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_11[7:4]				PRI_11[3:0]				Reserved							
rw	rw	rw	rw	r	r	r	r								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

图 18-2 SHPR2 寄存器

System handler priority register 3 (SHPR3)

Address: 0xE000 ED20

Reset value: 0x0000 0000

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_15[7:4]				PRI_15[3:0]				PRI_14[7:4]				PRI_14[3:0]			
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

图 18-3 SHPR3 寄存器

在系统定时器中，配置优先级为 (1UL « __NVIC_PRIO_BITS) -1UL)，其中宏 __NVIC_PRIO_BITS 为 4，那计算结果就等于 15，可以看出系统定时器此时设置的优先级在内核外设中是最低的，如果要修改优先级则修改这个值即可，范围为：0~15。

```
// 设置系统定时器中断优先级
NVIC_SetPriority (SysTick_IRQn, (1UL << __NVIC_PRIO_BITS) - 1UL);
```

17.3.2.2.3 SysTick 初始化函数

列表 2: 代码 18-2 SysTick 初始化函数

```
/**
* @brief  启动系统滴答定时器 SysTick
* @param  无
* @retval 无
*/
void SysTick_Init(void)
{
    /* SystemFrequency / 1000    1ms 中断一次
    * SystemFrequency / 100000  10us 中断一次
```

(下页继续)

(续上页)

```
    * SystemFrequency / 1000000 1us 中断一次
    */
    if (HAL_SYSTICK_Config(SystemCoreClock / 100000)) {
        /* Capture error */
        while (1);
    }
}
```

SysTick 初始化函数由用户编写，里面调用了 SysTick_Config() 这个固件库函数，通过设置该固件库函数的形参，就决定了系统定时器经过多少时间就产生一次中断。

17.3.2.2.4 SysTick 中断时间的计算

SysTick 定时器的计数器是向下递减计数的，计数一次的时间 T_{DEC}=1/CLK_{AHB}，当重装载寄存器中的值 $VALUE_{LOAD}$ 减到 0 的时候，产生中断，可知中断一次的时间 T_{INT}=$VALUE_{LOAD}$ * T_{DEC}= VALUE $_{LOAD}$/CLK:sub:*AHB*，其中 CLK_{AHB} =72MHZ。如果设置 $VALUE_{LOAD}$ 为 72，那中断一次的时间 T_{INT}=72/72M=1us。不过 1us 的中断没啥意义，整个程序的重心都花在进出中断上了，根本没有时间处理其他的任务。

```
SysTick_Config(SystemCoreClock / 100000))
```

SysTick_Config （）的形我们配置为 SystemCoreClock / 100000=72M/100000=720，从刚刚分析我们知道这个形参的值最终是写到重装载寄存器 LOAD 中的，从而可知我们现在把 SysTick 定时器中断一次的时间 T_{INT}=720/72M=10us。

17.3.2.2.5 SysTick 定时时间的计算

当设置好中断时间 T_{INT} 后，我们可以设置一个变量 t，用来记录进入中断的次数，那么变量 t 乘以中断的时间 T_{INT} 就可以计算出需要定时的时间。

17.3.2.2.6 SysTick 定时函数

现在我们定义一个微秒级别的延时函数，形参为 nTime，当用这个形参乘以中断时间 T_{INT} 就得出我们需要的延时时间，其中 T_{INT} 我们已经设置好为 10us。关于这个函数的具体调用看注释即可。

```
/**
* @brief   us 延时程序,10us 为一个单位
* @param
*   @arg nTime: Delay_us( 1 ) 则实现的延时为 1 * 10us = 10us
* @retval  无
*/
void Delay_us(__IO u32 nTime)
{
    TimingDelay = nTime;

    while (TimingDelay != 0);
}
```

函数 Delay_us() 中我们等待 TimingDelay 为 0，当 TimingDelay 为 0 的时候表示延时时间到。变量 TimingDelay 在中断函数中递减，即 SysTick 每进一次中断即 10us 的时间 TimingDelay 递减一次。

17.3.2.2.7 SysTick 中断服务函数

```
void SysTick_Handler(void)
{
    TimingDelay_Decrement();
}
```

中断复位函数调用了另外一个函数 TimingDelay_Decrement()，原型如下：

```
/**
* @brief  获取节拍程序
```

(下页继续)

(续上页)

```
* @param  无
* @retval 无
* @attention  在 SysTick 中断函数 SysTick_Handler() 调用
*/
void TimingDelay_Decrement(void)
{
    if (TimingDelay != 0x00) {
        TimingDelay--;
    }
}
```

TimingDelay 的值等于延时函数中传进去的 nTime 的值，比如 nTime=100000，则延时的时间等于 100000*10us=1s。

17.3.2.2.8 主函数

```
int main(void)
{
    /* 系统时钟初始化成 72MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 配置 SysTick 为 10us 中断一次,
    时间到后触发定时中断,
    * 进入 stm32f7xx_it.
    c 文件的 SysTick_Handler 处理，通过数中断次数计时
    */
    SysTick_Init();
    while (1) {
        LED_RED;
        Delay_us(100000);      // 10000 * 10us = 1000ms
```

(下页继续)

(续上页)

```
        LED_GREEN;
        Delay_us(100000);    // 10000 * 10us = 1000ms
        LED_BLUE;
        Delay_us(100000);    // 10000 * 10us = 1000ms
    }
}
```

主函数中初始化了 LED 和 SysTick，然后在一个 while 循环中以 1s 的频率让 LED 闪烁。

第 18 章　通讯的基本概念

在计算机设备与设备之间或集成电路之间常常需要进行数据传输，在本书后面的章节中我们会学习到各种各样的通讯方式，所以在本章中我们先统一介绍这些通讯的基本概念。

18.1 串行通讯与并行通讯

按数据传送的方式，通讯可分为串行通讯与并行通讯，串行通讯是指设备之间通过少量数据信号线 (一般是 8 根以下)，地线以及控制信号线，按数据位形式一位一位地传输数据的通讯方式。而并行通讯一般是指使用 8、16、32 及 64 根或更多的数据线进行传输的通讯方式，它们的通讯传输对比说明见图 *19_1*，并行通讯就像多个车道的公路，可以同时传输多个数据位的数据，而串行通讯，而串行通讯就像单个车道的公路，同一时刻只能传输一个数据位的数据。

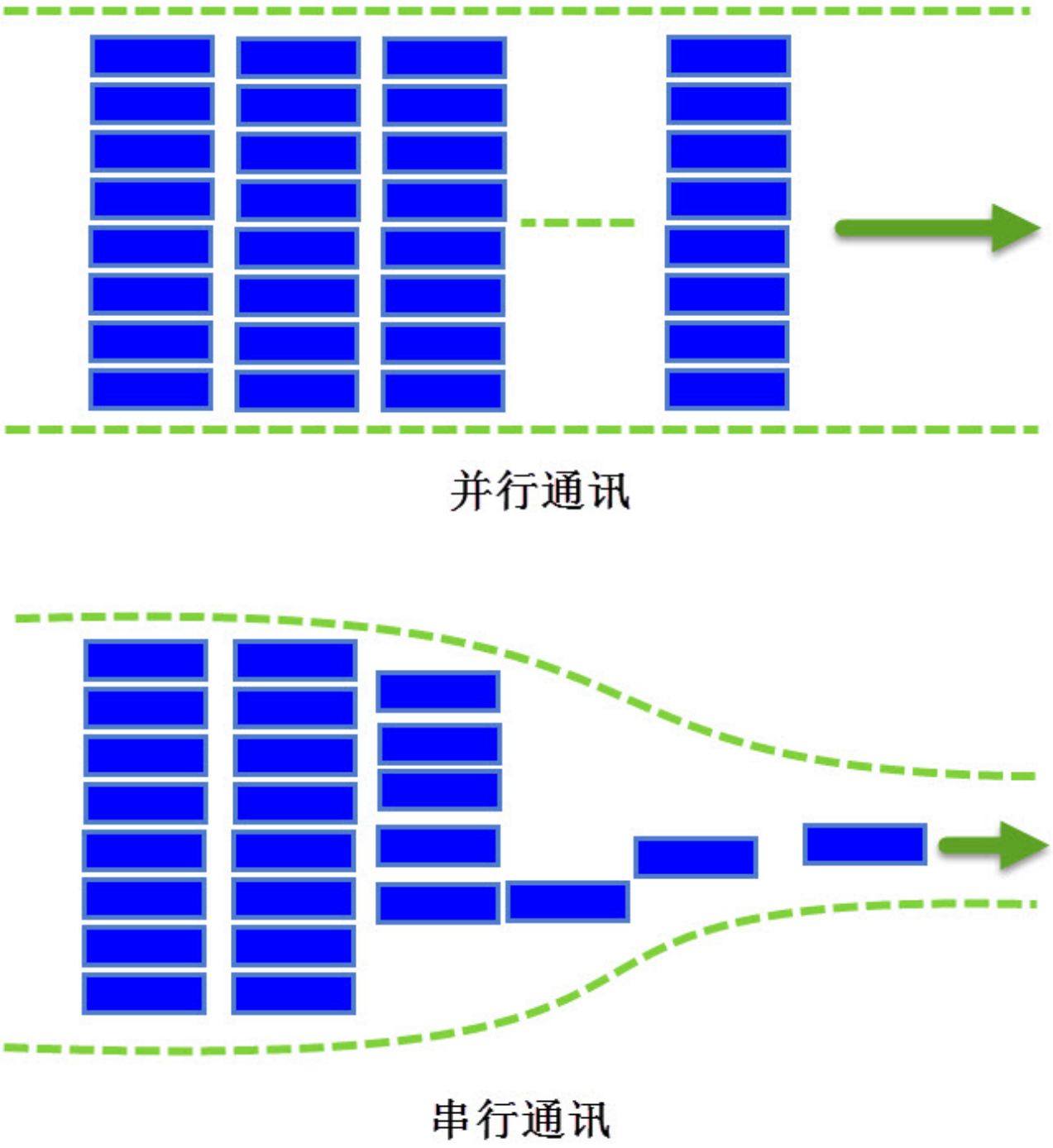


图 19-1 并行通讯与串行通讯的对比图

很明显，因为一次可传输多个数据位的数据，在数据传输速率相同的情况下，并行通讯传输的数据量要大得多，而串行通讯则可以节省数据线的硬件成本 (特别是远距离时) 以及 PCB 的布线面积，串行通讯与并行通讯的特性对比见表 *19_1*。

表 19-1 串行通讯与并行通讯的特性对比

特性	串行通讯	并行通讯
通讯距离	较远	较近
抗干扰能力	较强	较弱
传输速率	较慢	较高
成本	较低	较高

不过由于并行传输对同步要求较高，且随着通讯速率的提高，信号干扰的问题会显著影响通讯性能，现在随着技术的发展，越来越多的应用场合采用高速率的串行差分传输。

18.2 全双工、半双工及单工通讯

根据数据通讯的方向，通讯又分为全双工、半双工及单工通讯，它们主要以信道的方向来区分，见图 *19_2* 及表 *19_2*。

表 19-2 通讯方式说明

通讯方式	说明
全双工	在同一时刻，两个设备之间可以同时收发数据
半双工	两个设备之间可以收发数据，但不能在同一时刻进行
单工	在任何时刻都只能进行一个方向的通讯，即一个固定为发送设备，另一个固定为接收设备

仍以公路来类比，全双工的通讯就是一个双向车道，两个方向上的车流互不相干；半双工则像乡间小道那样，同一时刻只能让一辆小车通过，另一方向的来车只能等待道路空出来时才能经过；而单工则像单行道，另一方向的车辆完全禁止通行。

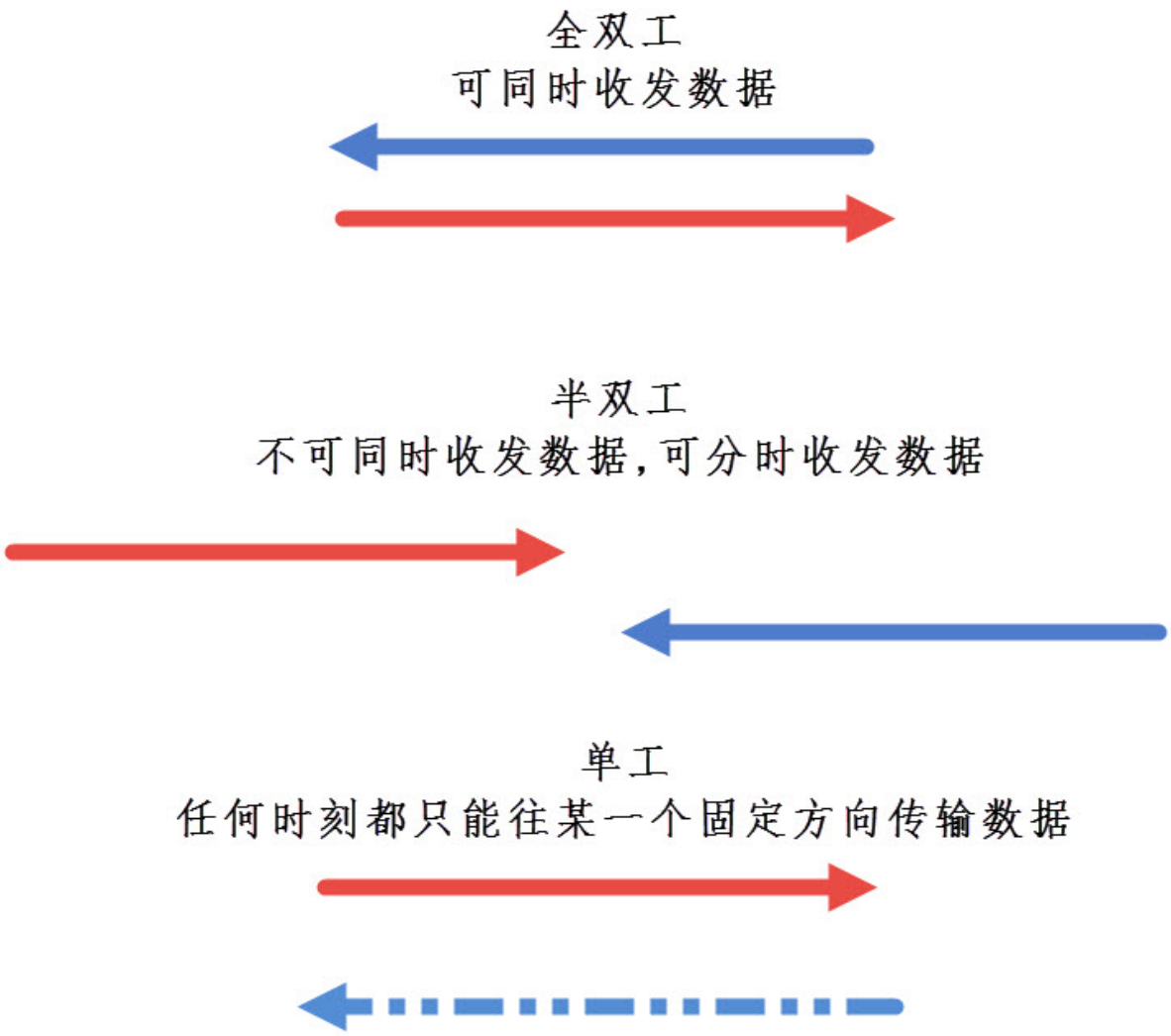


图 19-2 全双工、半双工及单工通讯

18.3 同步通讯与异步通讯

根据通讯的数据同步方式，又分为同步和异步两种，可以根据通讯过程中是否有使用到时钟信号进行简单的区分。

在同步通讯中，收发设备双方会使用一根信号线表示时钟信号，在时钟信号的驱动下双方进行协调，同步数据，见图 *19_3*。通讯中通常双方会统一规定在时钟信号的上升沿或下降沿对数据线进行采样。

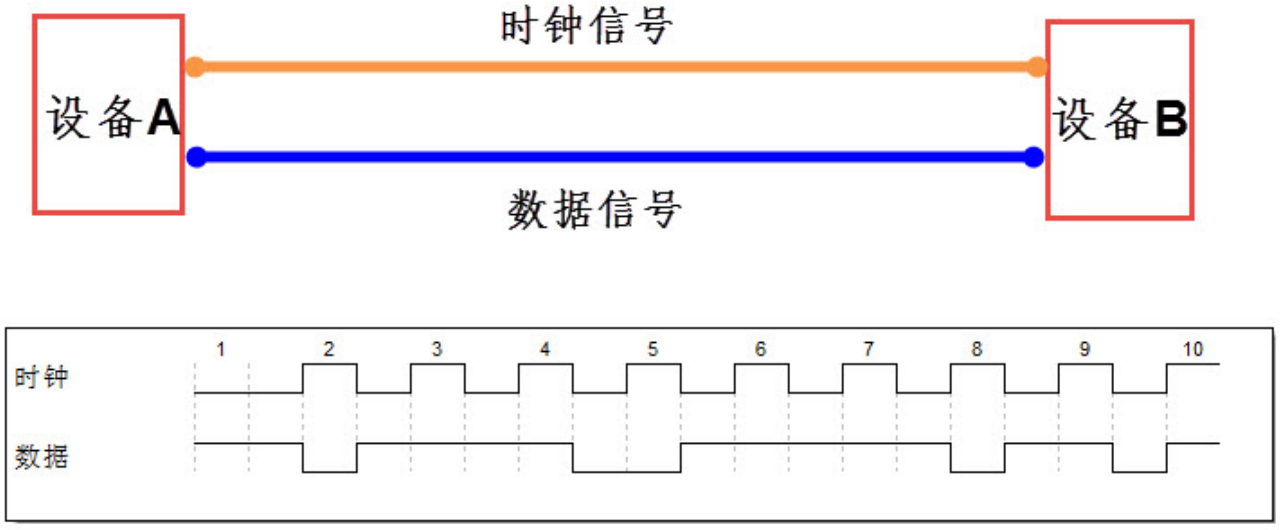


图 19-3 同步通讯

在异步通讯中不使用时钟信号进行数据同步，它们直接在数据信号中穿插一些同步用的信号位，或者把主体数据进行打包，以数据帧的格式传输数据，见图 *19_4*，某些通讯中还需要双方约定数据的传输速率，以便更好地同步。

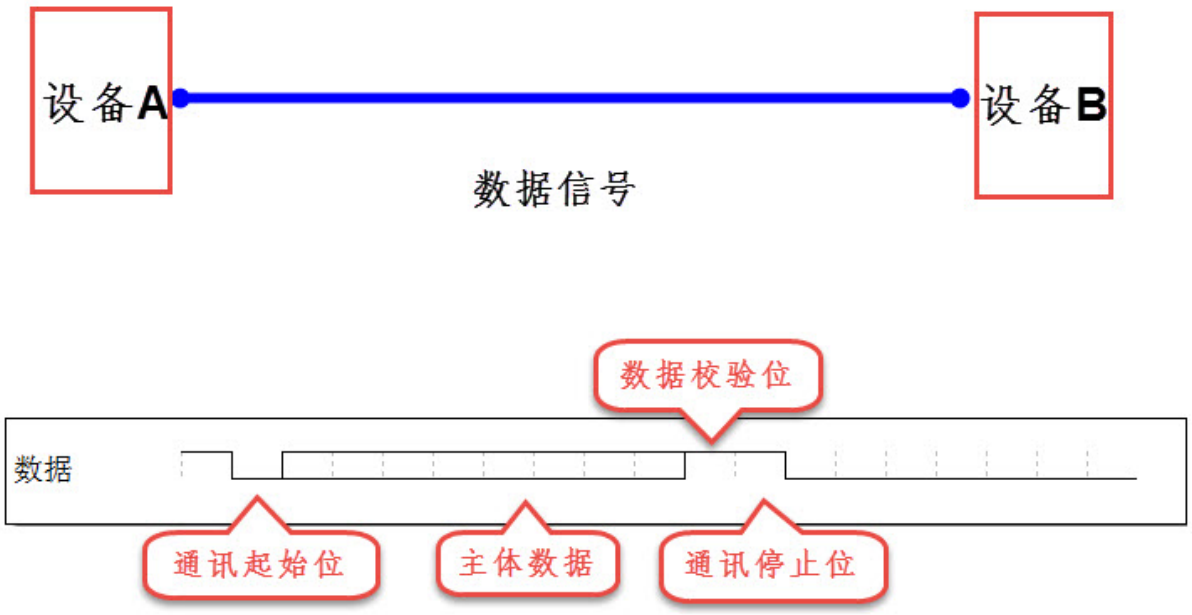


图 19-4 某种异步通讯

在同步通讯中，数据信号所传输的内容绝大部分就是有效数据，而异步通讯中会包含有帧的各种标识符，所以同步通讯的效率更高，但是同步通讯双方的时钟允许误差较小，而异步通讯双方的时钟允许误差较大。

18.4 通讯速率

衡量通讯性能的一个非常重要的参数就是通讯速率，通常以比特率 (Bitrate) 来表示，即每秒钟传输的二进制位数，单位为比特每秒 (bit/s)。容易与比特率混淆的概念是“波特率”(Baudrate)，它表示每秒钟传输了多少个码元。而码元是通讯信号调制的概念，通讯中常用时间间隔相同的符号来表示一个二进制数字，这样的信号称为码元。如常见的通讯传输中，用 0V 表示数字 0，5V 表示数字 1，那么一个码元可以表示两种状态 0 和 1，所以一个码元等于一个二进制比特位，此时波特率的大小与比特率一致；如果在通讯传输中，有 0V、2V、4V 以及 6V 分别表示二进制数 00、01、10、11，那么每个码元可以表示四种状态，即两个二进制比特位，所以码元数是二进制比特位数的一半，这个时候的波特率为比特率的一半。因为很多常见的通讯中一个码元都是表示两种

状态，人们常常直接以波特率来表示比特率，虽然严格来说没什么错误，但希望您能了解它们的区别。

第 19 章　USART—串口通讯

本章参考资料：《STM32F10X-中文参考手册》USART 章节。学习本章时，配合《STM32F10X-中文参考手册》USART 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。特别说明，本书内容是以 STM32F103 系列控制器资源讲解。

19.1 串口通讯协议简介

串口通讯 (Serial Communication) 是一种设备间非常常用的串行通讯方式，因为它简单便捷，因此大部分电子设备都支持该通讯方式，电子工程师在调试设备时也经常使用该通讯方式输出调试信息。

在计算机科学里，大部分复杂的问题都可以通过分层来简化。如芯片被分为内核层和片上外设；STM32HAL 库则是在寄存器与用户代码之间的软件层。对于通讯协议，我们也以分层的方式来理解，最基本的是把它分为物理层和协议层。物理层规定通讯系统中具有机械、电子功能部分的特性，确保原始数据在物理媒体的传输。协议层主要规定通讯逻辑，统一收发双方的数据打包、解包标准。简单来说物理层规定我们用嘴巴还是用肢体来交流，协议层则规定我们用中文还是英文来交流。

下面我们分别对串口通讯协议的物理层及协议层进行讲解。

19.1.1 物理层

串口通讯的物理层有很多标准及变种，我们主要讲解 RS-232 标准，RS-232 标准主要规定了信号的用途、通讯接口以及信号的电平标准。

使用 RS-232 标准的串口设备间常见的通讯结构见图 *20_1*。

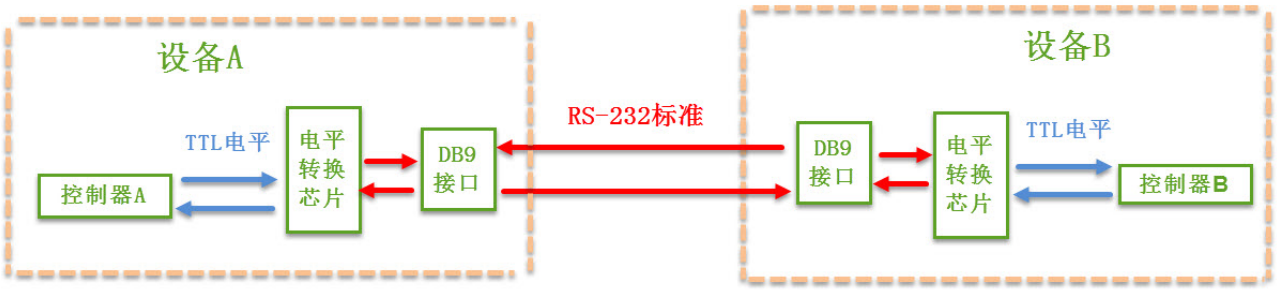


图 20-1 串口通讯结构图

在上面的通讯方式中，两个通讯设备的“DB9 接口”之间通过串口信号线建立起连接，串口信号线中使用“RS-232 标准”传输数据信号。由于 RS-232 电平标准的信号不能直接被控制器直接识别，所以这些信号会经过一个“电平转换芯片”转换成控制器能识别的“TTL 标准”的电平信号，才能实现通讯。

19.1.1.1 电平标准

根据通讯使用的电平标准不同，串口通讯可分为 TTL 标准及 RS-232 标准，见表 20-1。

表 20-1 TTL 电平标准与 RS232 电平标准

通讯标准	电平标准 (发送端)
5V TTL	逻辑 1：2.4V-5V
	逻辑 0：0~0.5V
RS-232	逻辑 1：-15V~-3V
	逻辑 0：+3V~+15V

我们知道常见的电子电路中常使用 TTL 的电平标准，理想状态下，使用 5V 表示二进制逻辑 1，使用 0V 表示逻辑 0；而为了增加串口通讯的远距离传输及抗干扰能力，它使用-15V 表示逻辑 1，+15V 表示逻辑 0。使用 RS232 与 TTL 电平校准表示同一个信号时的对比见图 *20_2*。

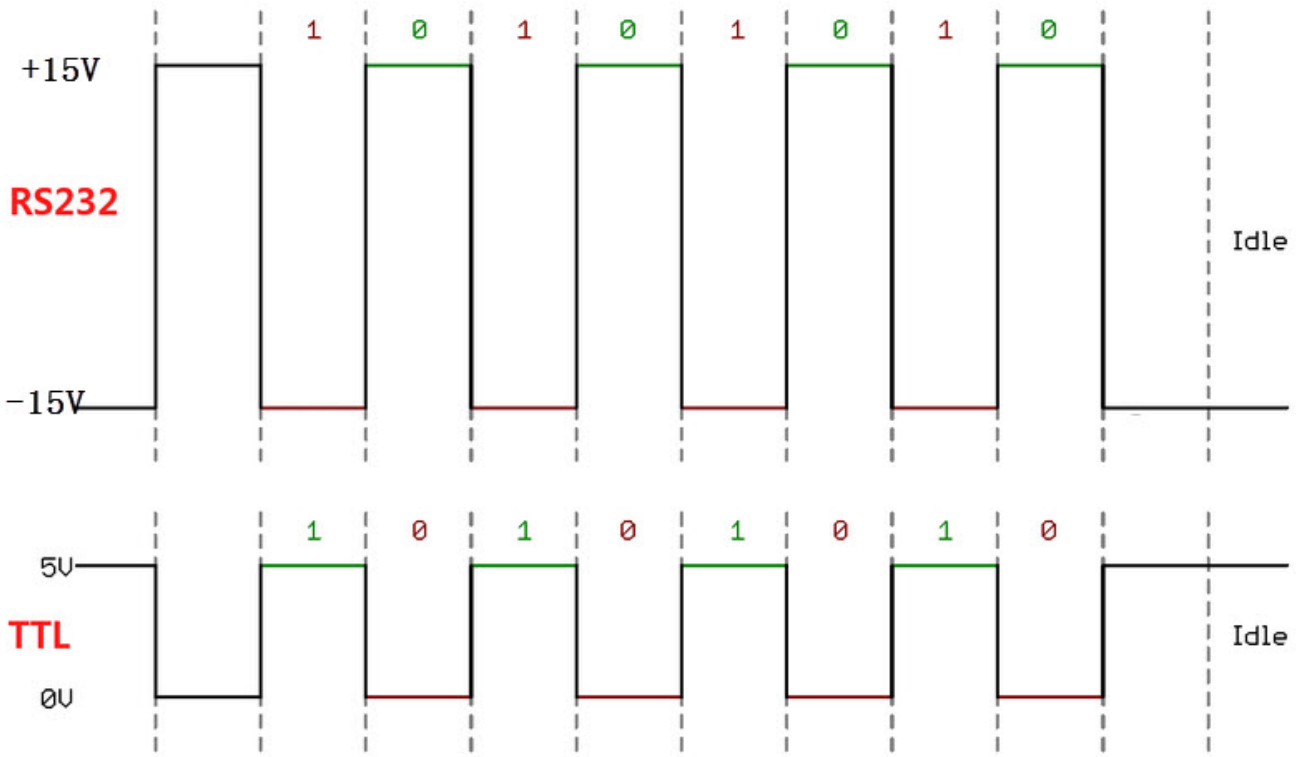


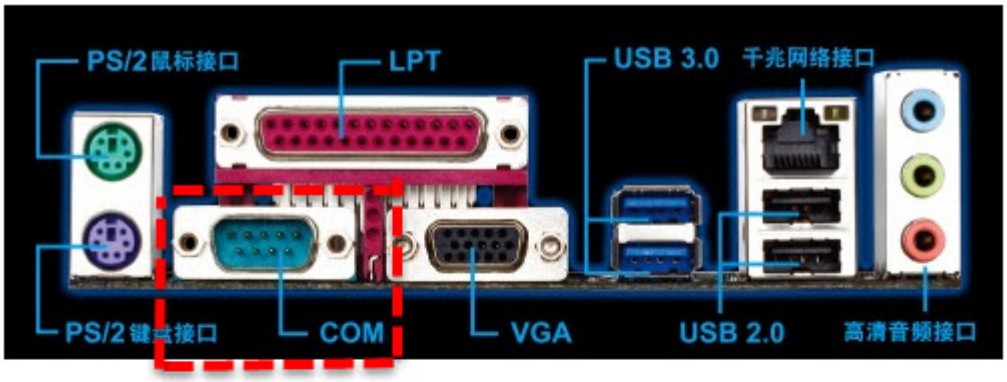
图 20-2 RS-232 与 TTL 电平标准下表示同一个信号

因为控制器一般使用 TTL 电平标准，所以常常会使用 MA3232 芯片对 TTL 及 RS-232 电平的信号进行互相转换。

19.1.1.2 RS-232 信号线

在最初的应用中，RS-232 串口标准常用于计算机、路由与调制调解器 (MODEN，俗称“猫”) 之间的通讯，在这种通讯系统中，设备被分为数据终端设备 DTE(计算机、路由) 和数据通讯设备 DCE(调制调解器)。我们以这种通讯模型讲解它们的信号线连接方式及各个信号线的作用。

在旧式的台式计算机中一般会有 RS-232 标准的 COM 口 (也称 DB9 接口)，见图 *20_3*。



COM口即DB9接口

串口线

图 20-3 电脑主板上的 COM 口及串口线

其中接线口以针式引出信号线的称为公头，以孔式引出信号线的称为母头。在计算机中一般引出公头接口，而在调制调解器设备中引出的一般为母头，使用上图中的串口线即可把它与计算机连接起来。通讯时，串口线中传输的信号就是使用前面讲解的 RS-232 标准调制的。

在这种应用场合下，DB9 接口中的公头及母头的各个引脚的标准信号线接法见图 *20_4* 及表 *20_2*。

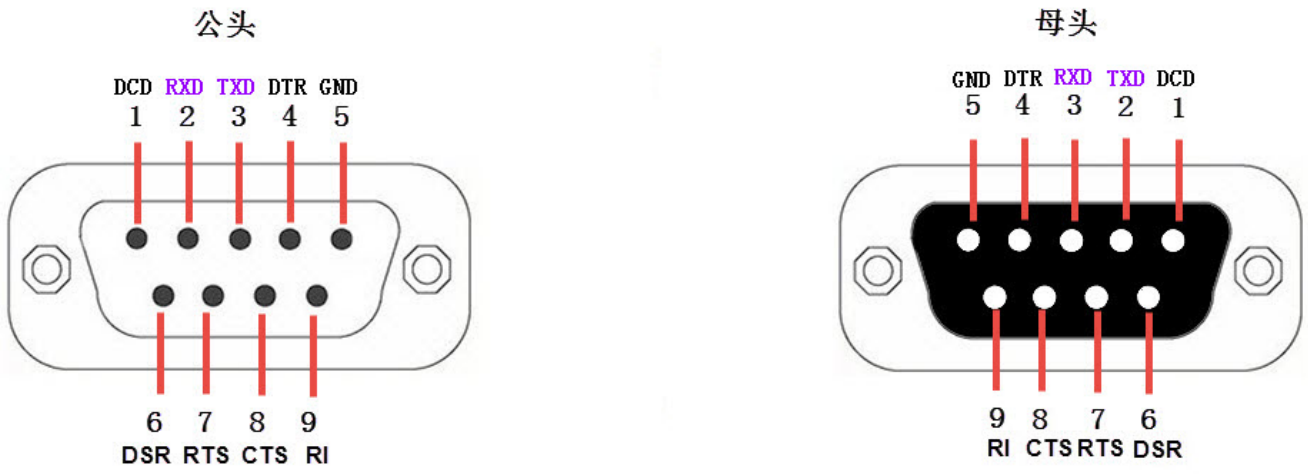


图 20-4 DB9 标准的公头及母头接法

表 20-2 DB9 信号线说明 (公头，为方便理解，可把 DTE 理解为计算机，DCE 理解为调制调解器)

序号	名称	符号	数据方向	说明
1	载波检测	DCD	DTEDCE	Data Carrier Detect，数据载波检测，用于 DTE 告知对方，本机是否收到对方的载波信号
2	接收数据	RXD	DTEDCE	Receive Data，数据接收信号，即输入。
3	发送数据	TXD	DTEDCE	Transmit Data，数据发送信号，即输出。两个设备之间的 TXD 与 RXD 应交叉相连
4	数据终端 (DTE) 就绪	DTR	DTEDCE	Data Terminal Ready，数据终端就绪，用于 DTE 向对方告知本机是否已准备好
5	信号地	GND	•	地线，两个通讯设备之间的地电位可能不一样，这会影响收发双方的电平信号，所以两个串口设备之间必须要使用地线连接，即共地。
6	数据设备 (DCE) 就绪	DSR	DTEDCE	Data Set Ready，数据发送就绪，用于 DCE 告知对方本机是否处于待命状态
7	请求发送	RTS	DTEDCE	Request To Send，请求发送，DTE

上表中的是计算机端的 DB9 公头标准接法，由于两个通讯设备之间的收发信号 (RXD 与 TXD) 应交叉相连，所以调制调解器端的 DB9 母头的收发信号接法一般与公头的相反，两个设备之间连接时，只要使用“直通型”的串口线连接起来即可，见图 *20_5*。

图 20-5 计算机与调制调解器的信号线连接

串口线中的 RTS、CTS、DSR、DTR 及 DCD 信号，使用逻辑 1 表示信号有效，逻辑 0 表示信号无效。例如，当计算机端控制 DTR 信号线表示为逻辑 1 时，它是为了告知远端的调制调解器，本机已准备好接收数据，0 则表示还没准备就绪。

在目前的其它工业控制使用的串口通讯中，一般只使用 RXD、TXD 以及 GND 三条信号线，直接传输数据信号，而 RTS、CTS、DSR、DTR 及 DCD 信号都被裁剪掉了。

19.1.2 协议层

串口通讯的数据包由发送设备通过自身的 TXD 接口传输到接收设备的 RXD 接口。在串口通讯的协议层中，规定了数据包的内容，它由启始位、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，其组成见图 *20_6*。

图 20-6 串口数据包的基本组成

19.1.2.1 波特率

本章中主要讲解的是串口异步通讯，异步通讯中由于没有时钟信号 (如前面讲解的 DB9 接口中是没有时钟信号的)，所以两个通讯设备之间需要约定好波特率，即每个码元的长度，以便对信号进行解码，图 *20_6* 中用虚线分开的每一格就是代表一个码元。常见的波特率为 4800、9600、115200 等。

19.1.2.2 通讯的起始和停止信号

串口通讯的一个数据包从起始信号开始，直到停止信号结束。数据包的起始信号由一个逻辑 0 的数据位表示，而数据包的停止信号可由 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。

19.1.2.3 有效数据

在数据包的起始位之后紧接着的就是要传输的主体数据内容，也称为有效数据，有效数据的长度常被约定为 5、6、7 或 8 位长。

19.1.2.4 数据校验

在有效数据之后，有一个可选的数据校验位。由于数据通信相对更容易受到外部干扰导致传输数据出现偏差，可以在传输过程加上校验位来解决这个问题。校验方法有奇校验 (odd)、偶校验 (even)、0 校验 (space)、1 校验 (mark) 以及无校验 (noparity)。

奇校验要求有效数据和校验位中“1”的个数为奇数，比如一个 8 位长的有效数据为：01101001，此时总共有 4 个“1”，为达到奇校验效果，校验位为“1”，最后传输的数据将是 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”。

19.2 STM32 的 USART 简介

通用同步异步收发器 (Universal Synchronous Asynchronous Receiver and Transmitter) 是一个串行通信设备，可以灵活地与外部设备进行全双工数据交换。有别于 USART 还有一个 UART(Universal Asynchronous Receiver and Transmitter)，它是在 USART 基础上裁剪掉了同步通信功能，只有异步通信。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，我们平时用的串口通信基本都是 UART。

串行通信一般是以帧格式传输数据，即是一帧一帧的传输，每帧包含有起始信号、数据信息、停止信息，可能还有校验信息。USART 就是对这些传输参数有具体规定，当然也不是只有唯一一个参数值，很多参数值都可以自定义设置，只是增强它的兼容性。

USART 满足外部设备对工业标准 NRZ 异步串行数据格式的要求，并且使用了小数波特率发生器，可以提供多种波特率，使得它的应用更加广泛。USART 支持同步单向通信和半双工单线通信；还支持局域互连网络 LIN、智能卡 (SmartCard) 协议与 lrDA(红外线数据协会) SIR ENDEC 规范。

USART 支持使用 DMA，可实现高速数据通信，有关 DMA 具体应用将在 DMA 章节作具体讲解。

USART 在 STM32 应用最多莫过于“打印”程序信息，一般在硬件设计时都会预留一个 USART

通信接口连接电脑，用于在调试程序是可以把一些调试信息“打印”在电脑端的串口调试助手工具上，从而了解程序运行是否正确、如果出错哪具体哪里出错等等。

19.3 USART 功能框图

USART 的功能框图包含了 USART 最核心内容，掌握了功能框图，对 USART 就有一个整体的把握，在编程时就思路就非常清晰。USART 功能框图见图 *20_7*。

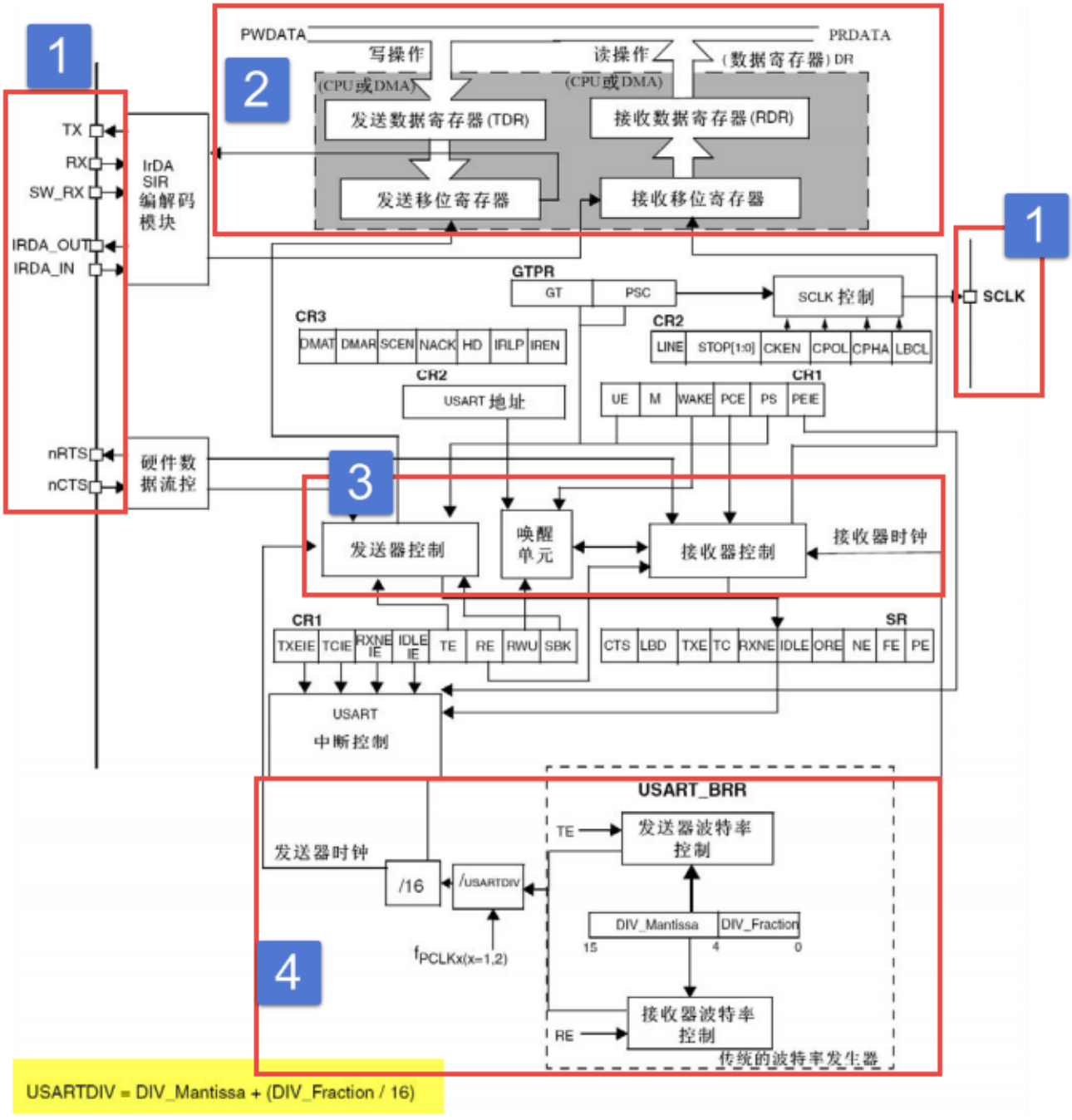


图 20-7 USART 功能框图

19.3.1 □ 功能引脚

TX：发送数据输出引脚。

RX：接收数据输入引脚。

SW_RX：数据接收引脚，只用于单线和智能卡模式，属于内部引脚，没有具体外部引脚。

nRTS：请求以发送 (Request To Send)，n 表示低电平有效。如果使能 RTS 流控制，当 USART 接收器准备好接收新数据时就会将 nRTS 变成低电平；当接收寄存器已满时，nRTS 将被设置为高电平。该引脚只适用于硬件流控制。

nCTS：清除以发送 (Clear To Send)，n 表示低电平有效。如果使能 CTS 流控制，发送器在发送下一帧数据之前会检测 nCTS 引脚，如果为低电平，表示可以发送数据，如果为高电平则在发送完当前数据帧之后停止发送。该引脚只适用于硬件流控制。

SCLK：发送器时钟输出引脚。这个引脚仅适用于同步模式。

USART 引脚在 STM32F103VET6 芯片具体分布见*表 20_3*。

表 20-3 STM32F103VET6 芯片的 USART 引脚

引脚	APB2 总线	APB1 总线			
	USART1	USART2	USART3	UART4	UART5
TX	PA9	PA2	PB10	PC10	PC12
RX	PA10	PA3	PB11	PC11	PD2
SCLK	PA8	PA4	PB12		
nCTS	PA11	PA0	PB13		
nRTS	PA12	PA1	PB14		

STM32F103VET6 系统控制器有三个 USART 和两个 UART，其中 USART1 和时钟来源于 APB2 总线时钟，其最大频率为 72MHz，其他四个的时钟来源于 APB1 总线时钟，其最大频率为 36MHz。UART 只是异步传输功能，所以没有 SCLK、nCTS 和 nRTS 功能引脚。

19.3.2 数据寄存器

USART 数据寄存器 (USART_DR) 只有低 9 位有效，并且第 9 位数据是否有效要取决于 USART 控制寄存器 1(USART_CR1) 的 M 位设置，当 M 位为 0 时表示 8 位数据字长，当 M 位为 1 表示 9 位数据字长，我们一般使用 8 位数据字长。

USART_DR 包含了已发送的数据或者接收到的数据。USART_DR 实际是包含了两个寄存器，一个专门用于发送的可写 TDR，一个专门用于接收的可读 RDR。当进行发送操作时，往 USART_DR 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。

TDR 和 RDR 都是介于系统总线和移位寄存器之间。串行通信是一个位一个位传输的，发送时把 TDR 内容转移到发送移位寄存器，然后把移位寄存器数据每一位发送出去，接收时把接收到的每一位顺序保存在接收移位寄存器内然后才转移到 RDR。

USART 支持 DMA 传输，可以实现高速数据传输，具体 DMA 使用将在 DMA 章节讲解。

19.3.3 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。使用 USART 之前需要向 USART_CR1 寄存器的 UE 位置 1 使能 USART，UE 位用来开启供给给串口的时钟。发送或者接收数据字长可选 8 位或 9 位，由 USART_CR1 的 M 位控制。

发送器

当 USART_CR1 寄存器的发送使能位 TE 置 1 时，启动数据发送，发送移位寄存器的数据会在 TX 引脚输出，低位在前，高位在后。如果是同步模式 SCLK 也输出时钟信号。

一个字符帧发送需要三个部分：起始位 + 数据帧 + 停止位。起始位是一个位周期的低电平，位周期就是每一位占用的时间；数据帧就是我们要发送的 8 位或 9 位数据，数据是从最低位开始传输的；停止位是一定时间周期的高电平。

停止位时间长短是可以通过 USART 控制寄存器 2(USART_CR2) 的 STOP[1:0] 位控制，可选 0.5 个、1 个、1.5 个和 2 个停止位。默认使用 1 个停止位。2 个停止位适用于正常 USART 模式、单线模式和调制解调器模式。0.5 个和 1.5 个停止位用于智能卡模式。

当选择 8 位字长，使用 1 个停止位时，具体发送字符时序图见图 *20_8*。

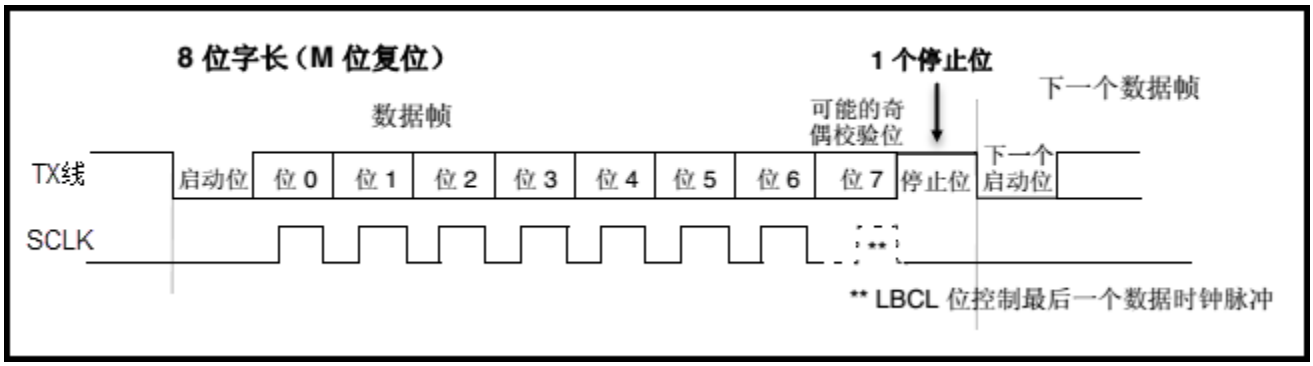


图 20-8 字符发送时序图

当发送使能位 TE 置 1 之后，发送器开始会先发送一个空闲帧 (一个数据帧长度的高电平)，接下来就可以往 USART_DR 寄存器写入要发送的数据。在写入最后一个数据后，需要等待 USART 状态寄存器 (USART_SR) 的 TC 位为 1，表示数据传输完成，如果 USART_CR1 寄存器的 TCIE 位置 1，将产生中断。

在发送数据时，编程的时候有几个比较重要的标志位我们来总结下。

名称	描述
TE	发送使能
TXE	发送寄存器为空，发送单个字节的时候使用
TC	发送完成，发送多个字节数据的时候使用
TXIE	发送完成中断使能

接收器

如果将 USART_CR1 寄存器的 RE 位置 1，使能 USART 接收，使得接收器在 RX 线开始搜索起始位。在确定到起始位后就根据 RX 线电平状态把数据存放在接收移位寄存器内。接收完成后就把接收移位寄存器数据移到 RDR 内，并把 USART_SR 寄存器的 RXNE 位置 1，同时如果 USART_CR2 寄存器的 RXNEIE 置 1 的话可以产生中断。

在接收数据时，编程的时候有几个比较重要的标志位我们来总结下。

名称	描述
RE	接收使能
RXNE	读数据寄存器非空
RXNEIE	发送完成中断使能

19.3.4 小数波特率生成

波特率指数据信号对载波的调制速率，它用单位时间内载波调制状态改变次数来表示，单位为波特。比特率指单位时间内传输的比特数，单位 bit/s(bps)。对于 USART 波特率与比特率相等，以后不区分这两个概念。波特率越大，传输速率越快。

USART 的发送器和接收器使用相同的波特率。计算公式如下：

$$\text{Tx / Rx 波特率} = \frac{f_{CK}}{(16 * USARTDIV)}$$

公式 20-1 波特率计算

其中，f_{PLCK} 为 USART 时钟，USARTDIV 是一个存放在波特率寄存器 (USART_BRR) 的一个无符号定点数。其中 DIV_Mantissa[11:0] 位定义 USARTDIV 的整数部分，DIV_Fraction[3:0] 位定义 USARTDIV 的小数部分。

例如：DIV_Mantissa=24(0x18)，DIV_Fraction=10(0x0A)，此时 USART_BRR 值为 0x18A；那么 USARTDIV 的小数位 10/16=0.625；整数位 24，最终 USARTDIV 的值为 24.625。

如果知道 USARTDIV 值为 27.68，那么 DIV_Fraction=16*0.68=10.88，最接近的正整数为 11，所以 DIV_Fraction[3:0] 为 0xB；DIV_Mantissa= 整数 (27.68)=27，即为 0x1B。

波特率的常用值有 2400、9600、19200、115200。下面以实例讲解如何设定寄存器值得到波特率的值。

我们知道 USART1 使用 APB2 总线时钟，最高可达 72MHz，其他 USART 的最高频率为 36MHz。

我们选取 USART1 作为实例讲解，即 f_{PLCK}=72MHz。为得到 115200bps 的波特率，此时：

$$115200 = \frac{72000000}{16 * \text{USARTDIV}}$$

解得 USARTDIV=39.0625，可算得 DIV_Fraction=0.0625*16=1=0x01，DIV_Mantissa=39=0x27，即应该设置 USART_BRR 的值为 0x271。

19.3.5 校验控制

STM32F103 系列控制器 USART 支持奇偶校验。当使用校验位时，串口传输的长度将是 8 位的数据帧加上 1 位的校验位总共 9 位，此时 USART_CR1 寄存器的 M 位需要设置为 1，即 9 数据位。将 USART_CR1 寄存器的 PCE 位置 1 就可以启动奇偶校验控制，奇偶校验由硬件自动完成。启动了奇偶校验控制之后，在发送数据帧时会自动添加校验位，接收数据时自动验证校验位。接收数据时如果出现奇偶校验位验证失败，会见 USART_SR 寄存器的 PE 位置 1，并可以产生奇偶校验中断。

使能了奇偶校验控制后，每个字符帧的格式将变成：起始位 + 数据帧 + 校验位 + 停止位。

19.3.6 中断控制

USART 有多个中断请求事件，具体见表 20-4。

表 20-4 USART 中断请求

中断事件	事件标志	使能控制位
发送数据寄存器为空	TXE	TXEIE
CTS 标志	CTS	CTSIE
发送完成	TC	TCIE
准备好读取接收到的数据	RXNE	RXNEIE
检测到上溢错误	ORE	RXNEIE
检测到空闲线路	IDLE	IDLEIE
奇偶校验错误	PE	PEIE
断路标志	LBD	LBDIE
多缓冲通信中的噪声标志、上溢错误和帧错误	NF/ORE/FE	EIE

19.4 USART 初始化结构体详解

HAL 库函数对每个外设都建立了一个初始化结构体，比如 USART_InitTypeDef，结构体成员用于设置外设工作参数，并由外设初始化配置函数，比如 USART_Init() 调用，这些设定参数将会设置外设相应的寄存器，达到配置外设工作环境的目的。

初始化结构体和初始化库函数配合使用是 HAL 库精髓所在，理解了初始化结构体每个成员意义基本上就可以对该外设运用自如了。初始化结构体定义在 stm32f1xx_hal_usart.h 文件中，初始化库函数定义在 stm32f1xx_hal_usart.c 文件中，编程时我们可以结合这两个文件内注释使用。

19.4.1 USART 初始化结构体

```
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
```

(下页继续)

(续上页)

```
    uint32_t HwFlowCtl;             //硬件流控制
    uint32_t OverSampling;          // 过采样模式
    uint32_t CLKLastBit;            // 最尾位时钟脉冲
} USART_InitTypeDef;
```

1) BaudRate：波特率设置。一般设置为 2400、9600、19200、115200。HAL 库函数会根据设定值计算得到 UARTDIV 值，见公式 20-1，并设置 UART_BRR 寄存器值。

2) WordLength：数据帧字长，可选 8 位或 9 位。它设定 UART_CR1 寄存器的 M 位的值。如果没有使能奇偶校验控制，一般使用 8 数据位；如果使能了奇偶校验则一般设置为 9 数据位。

3) StopBits：停止位设置，可选 0.5 个、1 个、1.5 个和 2 个停止位，它设定 USART_CR2 寄存器的 STOP[1:0] 位的值，一般我们选择 1 个停止位。

4) Parity：奇偶校验控制选择，可选 USART_PARITY_NONE (无校验)、USART_PARITY_EVEN (偶校验) 以及 USART_PARITY_ODD (奇校验)，它设定 UART_CR1 寄存器的 PCE 位和 PS 位的值。

5) Mode：UART 模式选择，有 USART_MODE_RX 和 USART_MODE_TX，允许使用逻辑或运算选择两个，它设定 USART_CR1 寄存器的 RE 位和 TE 位。

19.5 USART1 接发通信实验

USART 只需两根信号线即可完成双向通信，对硬件要求低，使得很多模块都预留 USART 接口来实现与其他模块或者控制器进行数据传输，比如 GSM 模块，WIFI 模块、蓝牙模块等等。在硬件设计时，注意还需要一根“共地线”。

我们经常使用 USART 来实现控制器与电脑之间的数据传输。这使得我们调试程序非常方便，比如我们可以把一些变量的值、函数的返回值、寄存器标志位等等通过 USART 发送到串口调试助手，这样我们可以非常清楚程序的运行状态，当我们正式发布程序时再把这些调试信息去除即可。

我们不仅仅可以将数据发送到串口调试助手，我们还可以在串口调试助手发送数据给控制器，控制器程序根据接收到的数据进行下一步工作。

首先，我们来编写一个程序实现开发板与电脑通信，在开发板上电时通过 USART 发送一串字符串给电脑，然后开发板进入中断接收等待状态，如果电脑有发送数据过来，开发板就会产生中断，我们在中断服务函数接收数据，并马上把数据返回发送给电脑。

19.5.1 硬件设计

为利用 USART 实现开发板与电脑通信，需要用到一个 USB 转 USART 的 IC，我们选择 CH340G 芯片来实现这个功能，CH340G 是一个 USB 总线的转接芯片，实现 USB 转 USART、USB 转 lrDA 红外或者 USB 转打印机接口，我们使用其 USB 转 USART 功能。具体电路设计见图 *20_9*。

我们将 CH340G 的 TXD 引脚与 USART1 的 RX 引脚连接，CH340G 的 RXD 引脚与 USART1 的 TX 引脚连接。CH340G 芯片集成在开发板上，其地线 (GND) 已与控制器的 GND 连通。

需要注意的是：虽然 MINI 板、指南者和霸道开发板、还有拂晓电机开发板用的都是 USART1，但拂晓电机开发板与其他板子使用的引脚不同：MINI 板、指南者和霸道开发板的引脚为 PA9、PA10，而拂晓开发板的为 PB6、PB7。

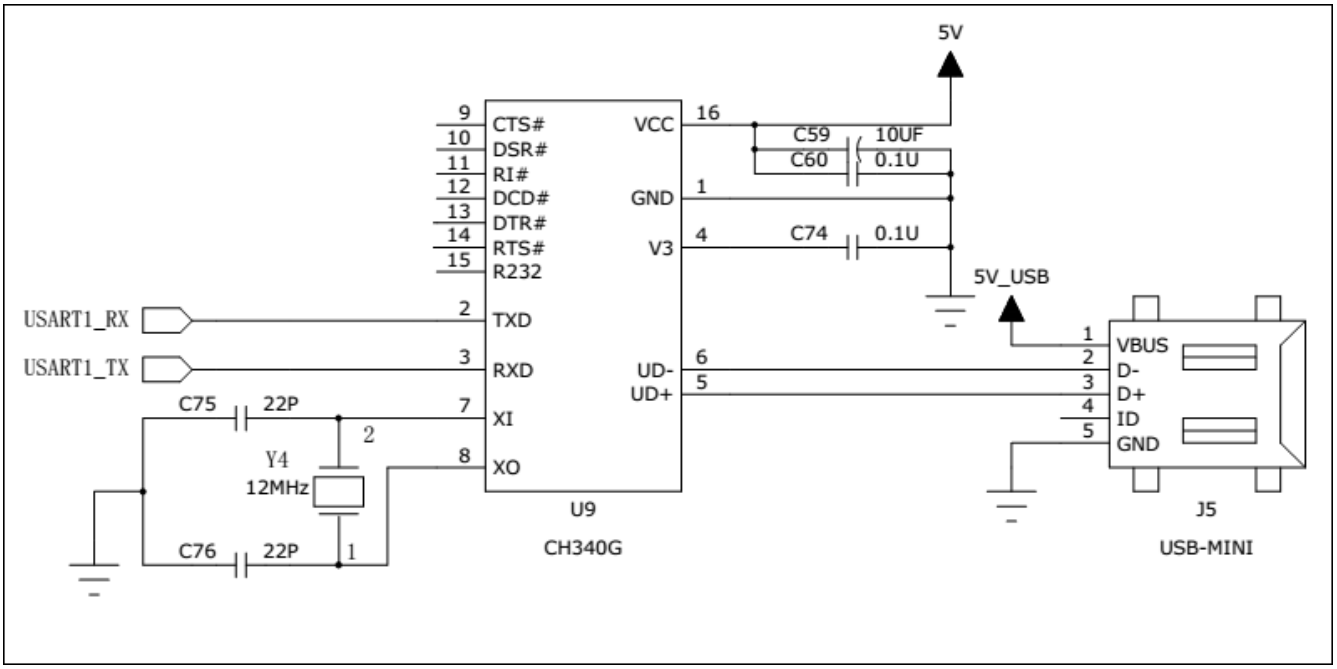


图 20-9 USB 转串口硬件设计

19.5.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_usart.c 和 bsp_usart.h 文件用来存放 USART 驱动程序及相关宏定义。

19.5.2.1 编程要点

1) 使能 RX 和 TX 引脚 GPIO 时钟和 USART 时钟；
2) 初始化 GPIO，并将 GPIO 复用到 USART 上；
3) 配置 USART 参数；
4) 配置中断控制器并使能 USART 接收中断；
5) 使能 USART；
6) 在 USART 接收中断服务函数实现数据接收和发送。

19.5.2.2 代码分析

19.5.2.2.1 GPIO 和 USART 宏定义

列表 1: 代码清单 20-1a MINI 板、指南者和霸道开发板的 GPIO 和 USART 宏定义

```
//串口波特率
#define DEBUG_USART_BAUDRATE                    115200

//引脚定义
/************************************************************/
#define DEBUG_USART                             USART1
#define DEBUG_USART_CLK_ENABLE()                __USART1_CLK_ENABLE();
```

(下页继续)

(续上页)

```
#define RCC_PERIPHCLK_UARTx                          RCC_PERIPHCLK_USART1
#define RCC_UARTxCLKSOURCE_SYSCLK                    RCC_USART1CLKSOURCE_SYSCLK

#define DEBUG_USART_RX_GPIO_PORT                     GPIOA
#define DEBUG_USART_RX_GPIO_CLK_ENABLE()             __GPIOA_CLK_ENABLE()
#define DEBUG_USART_RX_PIN                           GPIO_PIN_10
#define DEBUG_USART_RX_AF                            GPIO_AF7_USART1


#define DEBUG_USART_TX_GPIO_PORT                     GPIOA
#define DEBUG_USART_TX_GPIO_CLK_ENABLE()             __GPIOA_CLK_ENABLE()
#define DEBUG_USART_TX_PIN                           GPIO_PIN_9
#define DEBUG_USART_TX_AF                            GPIO_AF7_USART1

#define DEBUG_USART_IRQHandler                       USART1_IRQHandler
#define DEBUG_USART_IRQ                              USART1_IRQn
/************************************************************/
```

列表 2: 代码清单 20-1b 拂晓电机开发板的 GPIO 和 USART 宏定义

```
//串口波特率
#define DEBUG_USART_BAUDRATE                         115200

//引脚定义
/********************************************************/
#define DEBUG_USART                                  USART1
#define DEBUG_USART_CLK_ENABLE()                     __HAL_RCC_USART1_CLK_
↪ENABLE();

#define DEBUG_USART_RX_GPIO_PORT                     GPIOB
```

(下页继续)

(续上页)

```
#define DEBUG_USART_RX_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOB_CLK_
↪ENABLE()
#define DEBUG_USART_RX_PIN                      GPIO_PIN_7

#define DEBUG_USART_TX_GPIO_PORT                GPIOB
#define DEBUG_USART_TX_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOB_CLK_ENABLE()
#define DEBUG_USART_TX_PIN                      GPIO_PIN_6

#define DEBUG_USART_IRQHandler                  USART1_IRQHandler
#define DEBUG_USART_IRQ                                USART1_IRQn
/************************************************************/
```

使用宏定义方便程序移植和升级。开发板中的 CH340G 的收发引脚默认通过跳帽连接到 USART1，如果想使用其他串口，可以把 CH340G 跟 USART1 直接的连接跳帽拔掉，然后再把其他串口的 IO 用杜邦线接到 CH340G 的收发引脚即可。

这里我们使用 USART1，设定波特率为 115200，选定 USART 的 GPIO：MINI 板、指南者和霸道开发板的为 PA9、PA10，而拂晓开发板的为 PB6、PB7。

19.5.2.2.2 USART 初始化配置

列表 3: 代码清单 20-2 USART 初始化配置

```
/**
* @brief  DEBUG_USART GPIO 配置，工作模式配置。115200 8-N-1
* @param  无
* @retval 无
*/
void DEBUG_USART_Config(void)
{

```

(下页继续)

(续上页)

```
    UartHandle.Instance          = DEBUG_USART; //USART1 句柄
    UartHandle.Init.BaudRate     = DEBUG_USART_BAUDRATE;//波特率
    UartHandle.Init.WordLength   = UART_WORDLENGTH_8B; //8 位字长
    UartHandle.Init.StopBits     = UART_STOPBITS_1;     //一个停止位
    UartHandle.Init.Parity       = UART_PARITY_NONE;    //无奇偶校验
    UartHandle.Init.HwFlowCtl    = UART_HWCONTROL_NONE;//无硬件流控
    UartHandle.Init.Mode         = UART_MODE_TX_RX;    //收发模式

    HAL_UART_Init(&UartHandle);

    /* 使能串口接收断 */
    __HAL_UART_ENABLE_IT(&UartHandle,UART_IT_RXNE);
}
```

函数体中 UartHandle 是定义为 UART_HandleTypeDef 结构体类型的全局变量，它管理着串口的所有配置。该例程中 DEBUG_USART_Config 函数初始化 USART 与 MCU 硬件无关的东西：例如串口协议，其中包括波特率，奇偶校验，停止位等等，这些设置和使用什么样的 MCU 没有任何关系，可以使用 F1 的 MCU，也可以是 F2…F4，甚至是 PC 上的串口。所以就把串口抽像成为一个“串口”。至于对具体 mcu 底层硬件相关的配置如引脚、时钟、DMA、中断等是在 HAL_UART_MspInit(UART_HandleTypeDef *huart) 函数中完成的，该函数被 HAL_UART_Init 函数所调用。

需要特别指出的是在 HAL_UART_Init 调用 HAL_UART_MspInit 库函数的函数原型是：

列表 4: 代码清单 20-3 HAL_UART_MspInit 库函数定义

```
__weak void HAL_UART_MspDeInit(UART_HandleTypeDef *huart)
{
    /* 防止未使用的参数编译警告 */
    UNUSED(huart);
}
```

__weak 表示弱定义，表示如果你自己定义了同名的函数就不用他，如果你没定义就使用这个弱函数。其中 UNUSED 函数只是为了防止未使用的参数编译警告，其实质是什么也不做。下面列举的是 HAL_UART_MspInit 强函数的定义，它是实际被 HAL_UART_Init 函数所调用的函数。

列表 5: 代码清单 20-4 HAL_UART_MspInit 用户强函数定义

```
/**
* @brief UART MSP 初始化
* @param huart: UART handle
* @retval 无
*/
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
    GPIO_InitTypeDef  GPIO_InitStruct;

    DEBUG_USART_CLK_ENABLE();

    DEBUG_USART_RX_GPIO_CLK_ENABLE();
    DEBUG_USART_TX_GPIO_CLK_ENABLE();

    /**USART1 GPIO Configuration
    PA9     ------> USART1_TX
    PA10    ------> USART1_RX
```

(下页继续)

(续上页)

```
    */
    /* 配置 Tx 引脚为复用功能  */
    GPIO_InitStruct.Pin = DEBUG_USART_TX_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;  //复用推挽输出
    GPIO_InitStruct.Pull = GPIO_PULLUP;      //上拉
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH; //高速
    GPIO_InitStruct.Alternate = DEBUG_USART_TX_AF; //复用为 USART1
    HAL_GPIO_Init(DEBUG_USART_TX_GPIO_PORT, &GPIO_InitStruct);
    /* 配置 Rx 引脚为复用功能 */
    GPIO_InitStruct.Pin = DEBUG_USART_RX_PIN;
    GPIO_InitStruct.Alternate = DEBUG_USART_RX_AF;
    HAL_GPIO_Init(DEBUG_USART_RX_GPIO_PORT, &GPIO_InitStruct);
    /* 抢占优先级 0，子优先级 1*/
    HAL_NVIC_SetPriority(DEBUG_USART_IRQ ,0,1);
    HAL_NVIC_EnableIRQ(DEBUG_USART_IRQ ); /* 使能 USART1 中断通道 */
}
```

在 HAL 库中 IO 口初始化参数设置和复用映射配置是在函数 HAL_GPIO_Init 中一次性完成的。需要注意，我们要复用 PA9 和 PA10 为串口发送接收相关引脚，我们需要配置 IO 口为复用，同时复用映射到串口 1。对于中断优先级配置，方法就非常简单只需要调用函数 HAL_NVIC_SetPriority 中断优先级配置和函数 HAL_NVIC_EnableIRQ 中断使能即可。

19.5.2.2.3 字符发送

列表 6: 代码清单 20-5 字符发送函数

```
/*****************  发送字符串 **********************/
void Usart_SendString(uint8_t *str)
{
    unsigned int k=0;
    do {
```

(下页继续)

(续上页)

```
        HAL_UART_Transmit( &UartHandle,(uint8_t *)(str + k) ,1,1000);
        k++;
    } while (*(str + k)!='\0');
}
```

Usart_SendString 函数用来发送一个字符串，它实际是调用 HAL_UART_Transmit 函数（这是一个阻塞的发送函数，无需重复判断串口是否发送完成）发送每个字符，直到遇到空字符才停止发送。最后使用循环检测发送完成的事件标志来实现保证数据发送完成后才退出函数。

19.5.2.2.4 USART 中断服务函数

列表 7: 代码清单 20-6 USART 中断服务函数

```
void  DEBUG_USART_IRQHandler(void)
{
    uint8_t ch=a;

    if (__HAL_UART_GET_FLAG( &UartHandle, UART_FLAG_RXNE ) != RESET) {
        ch=( uint16_t)READ_REG(UartHandle.Instance->DR);
        WRITE_REG ( UartHandle.Instance->DR,ch);
    }
}
```

这段代码是存放在 stm32f1xx_it.c 文件中的，该文件用来集中存放外设中断服务函数。当我们使能了中断并且中断发生时就会执行中断服务函数。

我们在*代码清单 20_2* 使能了 USART 接收中断，当 USART 有接收到数据就会执行 DEBUG_USART_IRQHandler 函数。__HAL_UART_GET_FLAG 函数用来获取中断事件标志。使用 if 语句来判断是否是真的产生 USART 数据接收这个中断事件，如果是真的就使用 USART 数据读取函数 READ_REG 读取数据赋值给 ch, 读取过程会软件清除 UART_FLAG_RXNE 标志位。最后再调用 USART 写函数 WRITE_REG 把数据又发送给源设备。

19.5.2.2.5 主函数

列表 8: 代码清单 20-7 主函数

```
int main(void)
{
    HAL_Init();
    /* 配置系统时钟为 72MHz */
    SystemClock_Config();

    /* 初始化 USART 配置模式为 115200 8-N-1, 中断接收 */
    DEBUG_USART_Config();

    /* 调用 printf 函数, 因为重定向了 fputc, printf 的内容会
    输出到串口 */
    printf("欢迎使用野火开发板\n");

    /* 自定义函数方式 */
    Usart_SendString( (uint8_t *)"自定义函数输出：这是一个串口中断接收回显实验\n"␣
↪);

    while (1) {
    }
}
```

首先我们需要调用 SystemClock_Config 函数配置系统时钟，调用 Debug_USART_Config 函数完成 USART 初始化配置，包括 GPIO 配置，USART 配置，接收中断使用等等信息。数据的回传在中断函数实现。

19.5.3 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板的 USB 转串口跟电脑，在电脑端打开串口调试助手并配置好相关参数：115200 8-N-1，把编译好的程序下载到开发板，此时串口调试助手即可收到开发板发过来的数据。我们在串口调试助手发送区域输入任意字符，点击发送按钮，马上在串口调试助手接收区即可看到相同的字符。

图 20-10 实验现象

19.6 USART1 指令控制 LED 灯实验

在学习 C 语言时我们经常使用 C 语言标准函数库输入输出函数，比如 printf、scanf、getchar 等等。为让开发板也支持这些函数需要把 USART 发送和接收函数添加到这些函数的内部函数内。

正如之前所讲，可以在串口调试助手输入指令，让开发板根据这些指令执行一些任务，现在我们编写程序让开发板接收 USART 数据，然后根据数据内容控制 LED 灯的亮灭，指南者和霸道开发板上的 LED 灯是一个 RGB 彩灯，所以也可以通过 RGB 彩灯的 3 个引脚来控制灯的颜色。

19.6.1 硬件设计

硬件设计同前一个实验。LED 灯的接线引脚可参考前面点亮 LED 灯的章节。

19.6.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp _usart.c 和 bsp _usart.h 文件用来存放 USART 驱动程序及相关宏定义。

19.6.2.1 编程要点

1) 初始化配置 RGB 彩色灯 GPIO；
2) 使能 RX 和 TX 引脚 GPIO 时钟和 USART 时钟；
3) 初始化 GPIO，并将 GPIO 复用到 USART 上；
4) 配置 USART 参数；
5) 使能 USART；
6) 获取指令输入，根据指令控制 RGB 彩色灯。

与上一个实验不同的是我们这里不使用接收中断，而是靠查询标志位的方式来实现接收。

19.6.2.2 代码分析

19.6.2.2.1 GPIO 和 USART 宏定义

列表 9: 代码清单 20-8a MINI 板、指南者和霸道开发板的 GPIO 和 USART 宏定义

```
//串口波特率
#define DEBUG_USART_BAUDRATE                    115200

//引脚定义
/************************************************************/
#define DEBUG_USART                             USART1
#define DEBUG_USART_CLK_ENABLE()                __USART1_CLK_ENABLE();

#define RCC_PERIPHCLK_UARTx                     RCC_PERIPHCLK_USART1
#define RCC_UARTxCLKSOURCE_SYSCLK               RCC_USART1CLKSOURCE_SYSCLK

#define DEBUG_USART_RX_GPIO_PORT                GPIOA
#define DEBUG_USART_RX_GPIO_CLK_ENABLE()        __GPIOA_CLK_ENABLE()
#define DEBUG_USART_RX_PIN                      GPIO_PIN_10
#define DEBUG_USART_RX_AF                       GPIO_AF7_USART1

#define DEBUG_USART_TX_GPIO_PORT                GPIOA
#define DEBUG_USART_TX_GPIO_CLK_ENABLE()        __GPIOA_CLK_ENABLE()
#define DEBUG_USART_TX_PIN                      GPIO_PIN_9
#define DEBUG_USART_TX_AF                       GPIO_AF7_USART1

#define DEBUG_USART_IRQHandler                  USART1_IRQHandler
#define DEBUG_USART_IRQ                         USART1_IRQn
/************************************************************/
```

列表 10: 代码清单 20-8b 拂晓电机开发板的 GPIO 和 USART 宏定义

```
//串口波特率
#define DEBUG_USART_BAUDRATE                    115200

//引脚定义
/************************************************************/
#define DEBUG_USART                             USART1
#define DEBUG_USART_CLK_ENABLE()                __HAL_RCC_USART1_CLK_
↪ENABLE();

#define DEBUG_USART_RX_GPIO_PORT                GPIOB
#define DEBUG_USART_RX_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOB_CLK_
↪ENABLE()
#define DEBUG_USART_RX_PIN                      GPIO_PIN_7

#define DEBUG_USART_TX_GPIO_PORT                GPIOB
#define DEBUG_USART_TX_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOB_CLK_ENABLE()
#define DEBUG_USART_TX_PIN                      GPIO_PIN_6

#define DEBUG_USART_IRQHandler                  USART1_IRQHandler
#define DEBUG_USART_IRQ                             USART1_IRQn
/************************************************************/
```

使用宏定义方便程序移植和升级，这里我们可以 USART1，设定波特率为 115200。

19.6.2.2.2 USART 初始化配置

列表 11: 代码清单 20-9 USART 初始化配置

```
void UARTx_Config(void)
{
GPIO_InitTypeDef GPIO_InitStruct;

UARTx_RX_GPIO_CLK_ENABLE();
UARTx_TX_GPIO_CLK_ENABLE();

/* 使能 UART 时钟 */
UARTx_CLK_ENABLE();

/**USART1 GPIO Configuration
PA9      ------> USART1_TX
PA10     ------> USART1_RX
*/
/* 配置 Tx 引脚为复用功能  */
GPIO_InitStruct.Pin = UARTx_TX_PIN;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = UARTx_TX_AF;
HAL_GPIO_Init(UARTx_TX_GPIO_PORT, &GPIO_InitStruct);

/* 配置 Rx 引脚为复用功能 */
GPIO_InitStruct.Pin = UARTx_RX_PIN;
GPIO_InitStruct.Alternate = UARTx_RX_AF;
HAL_GPIO_Init(UARTx_RX_GPIO_PORT, &GPIO_InitStruct);

/* 配置串 USARTx 模式 */
UartHandle.Instance = UARTx;
```

(下页继续)

(续上页)

```
UartHandle.Init.BaudRate = UARTx_BAUDRATE;
UartHandle.Init.WordLength = UART_WORDLENGTH_8B;
UartHandle.Init.StopBits = UART_STOPBITS_1;
UartHandle.Init.Parity = UART_PARITY_NONE;
UartHandle.Init.Mode = UART_MODE_TX_RX;
HAL_UART_Init(&UartHandle);

}
```

在这个函数中我们并没有使用 HAL_UART_MspInit 函数，为了直观与简洁，而是直接在 UARTx_Config 函数中进行引脚、时钟等配置，在以后的部分例程我们也会大量使用这种方式。

使用 GPIO_InitTypeDef 和 USART_InitTypeDef 结构体定义一个 GPIO 初始化变量以及一个 USART 初始化变量，这两个结构体内容我们之前已经有详细讲解。

调用 UARTx_RX_GPIO_CLK_ENABLE 和 UARTx_TX_GPIO_CLK_ENABLE 函数开启 GPIO 端口时钟，使用 GPIO 之前必须开启对应端口的时钟。

初始化配置 RX 线和 TX 线引脚为复用功能，并将指定的 GPIO 连接至 USART1，然后配置串口的工作参数为 115200-8-N-1。最后调用 HAL_UART_Init 函数初始化 USART。

19.6.2.2.3 重定向 printf 和 scanf 函数

列表 12: 代码清单 20-10 重定向输入输出函数

```
//重定向 c 库函数 printf 到串口 USARTx，重定向后可使用 printf 函数
int fputc(int ch, FILE *f)
{
    /* 发送一个字节数据到串口 USARTx */
    HAL_UART_Transmit(&UartHandle, (uint8_t *)&ch, 1, 0xFFFF);
    return (ch);
}
```

(下页继续)

(续上页)

```
//重定向 c 库函数 scanf 到串口 USARTx，重写向后可使用 scanf、getchar 等函数
int fgetc(FILE *f)
{
    int ch;
    /* 等待串口输入数据 */
    while (__HAL_UART_GET_FLAG(&UartHandle, UART_FLAG_RXNE) == RESET);
    HAL_UART_Receive(&UartHandle, (uint8_t *)&ch, 1, 0xFFFF);
    return (ch);
}
```

在 C 语言 HAL 库中，fputc 函数是 printf 函数内部的一个函数，功能是将字符 ch 写入到文件指针 f 所指向文件的当前写指针位置，简单理解就是把字符写入到特定文件中。我们使用 USART 函数重新修改 fputc 函数内容，达到类似“写入”的功能。

fgetc 函数与 fputc 函数非常相似，实现字符读取功能。在使用 scanf 函数时需要注意字符输入格式。

还有一点需要注意的，使用 fput 和 fgetc 函数达到重定向 C 语言 HAL 库输入输出函数必须在 MDK 的工程选项把“Use MicroLIB”勾选上，MicoroLIB 是缺省 C 库的备选库，它对标准 C 库进行了高度优化使代码更少，占用更少资源。

为使用 printf、scanf 函数需要在文件中包含 stdio.h 头文件。

19.6.2.2.4 输出提示信息

列表 13: 代码清单 20-11 输出提示信息

```
static void Show_Message(void)
{
    printf("\r\n   这是一个通过串口通信指令控制 RGB 彩灯实验 \n");
    printf("使用  USART  参数为：%d 8-N-1 \n",USART_BAUDRATE);
```

(下页继续)

(续上页)

```
    printf("开发板接到指令后控制 RGB 彩灯颜色，指令对应如下：\n");
    printf("   指令   ------ 彩灯颜色 \n");
    printf("     1     ------    红 \n");
    printf("     2     ------    绿 \n");
    printf("     3     ------    蓝 \n");
    printf("     4     ------    黄 \n");
    printf("     5     ------    紫 \n");
    printf("     6     ------    青 \n");
    printf("     7     ------    白 \n");
    printf("     8     ------    灭 \n");
}
```

Show_Message 函数全部是调用 printf 函数，“打印”实验操作信息到串口调试助手。

19.6.2.2.5 主函数

列表 14: 代码清单 20-12 主函数

```
int main(void)
{
    char ch;
    HAL_Init();
    /* 配置系统时钟为 72MHz */
    SystemClock_Config();
    /* 初始化 RGB 彩灯 */
    LED_GPIO_Config();
    /* 初始化 USART 配置模式为 115200 8-N-1，中断接收 */
    DEBUG_USART_Config();

    /* 打印指令输入提示信息 */
    Show_Message();
```

(下页继续)

(续上页)

```
    while (1) {
        /* 获取字符指令 */
        ch=getchar();
        printf("接收到字符: %c\n",ch);

        /* 根据字符指令控制 RGB 彩灯颜色 */
        switch (ch) {
        case '1':
            LED_RED;
            break;
        case '2':
            LED_GREEN;
            break;
        case '3':
            LED_BLUE;
            break;
        case '4':
            LED_YELLOW;
            break;
        case '5':
            LED_PURPLE;
            break;
        case '6':
            LED_CYAN;
            break;
        case '7':
            LED_WHITE;
            break;
        case '8':
            LED_RGBOFF;
            break;
```

(下页继续)

(续上页)

```
            default:
                /* 如果不是指定指令字符，打印提示信息 */
                Show_Message();
                break;
            }
        }
}
```

首先我们定义一个字符变量来存放接收到的字符。

接下来调用 SystemClock_Config 函数初始化系统时钟，调用 LED_GPIO_Config 函数完成 RGB 彩色 GPIO 初始化配置，该函数定义在 bsp_led.c 文件内。

调用 USARTx_Config 函完成 USART 初始化配置。

Show_Message 函数使用 printf 函数打印实验指令说明信息。

getchar 函数用于等待获取一个字符，并返回字符。我们使用 ch 变量保持返回的字符，接下来判断 ch 内容执行对应的程序了。

我们使用 switch 语句判断 ch 变量内容，并执行对应的功能程序。

19.6.3 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板，此时串口调试助手即可收到开发板发过来的数据。我们在串口调试助手发送区域输入一个特定字符，点击发送按钮，RGB 彩色灯状态随之改变。

第 20 章　DMA—直接存储区访问

本章参考资料：《STM32F10X-中文参考手册》DMA 控制器章节。

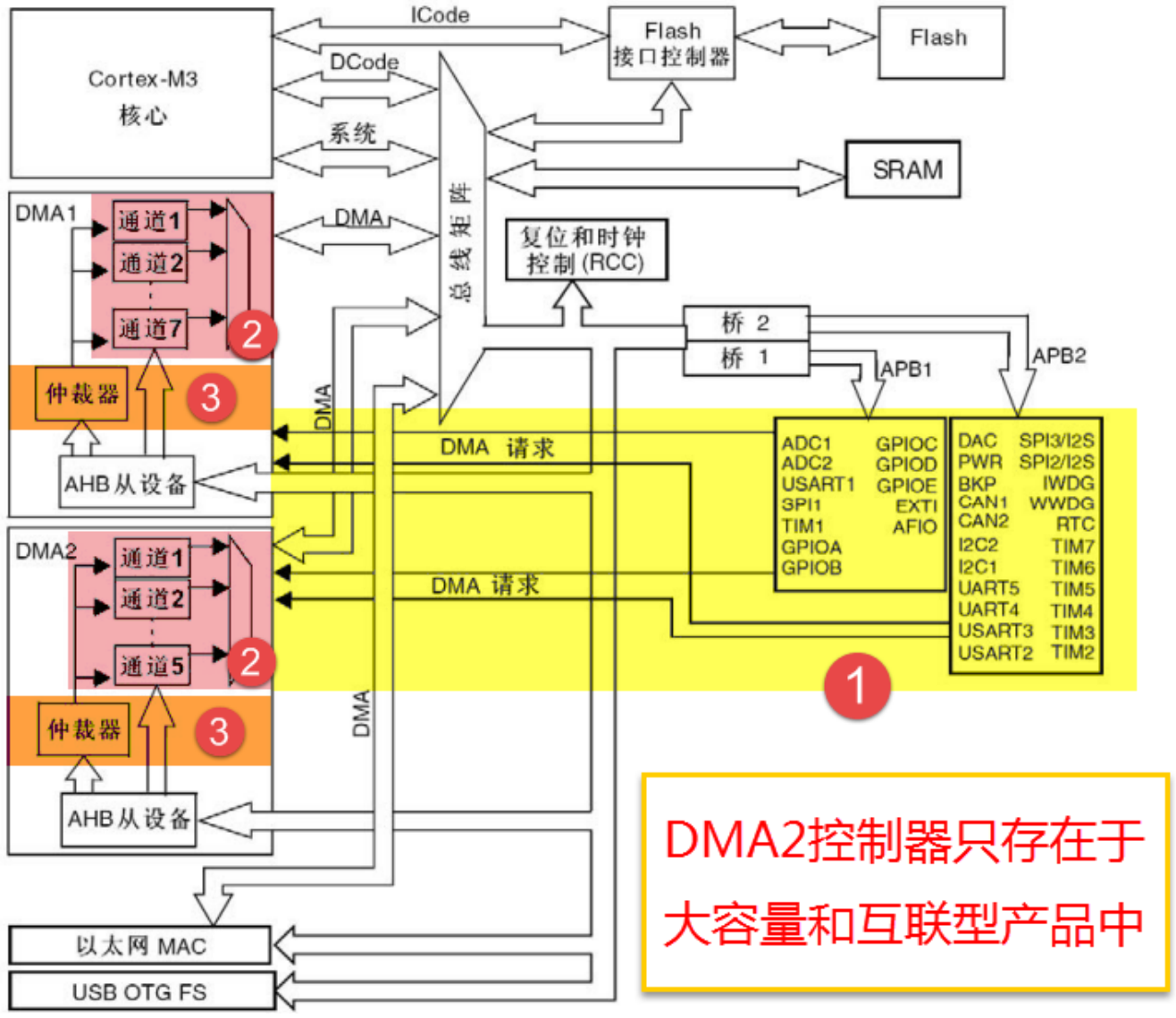
学习本章时，配合《STM32F10X-中文参考手册》DMA 控制器章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

20.1 DMA 简介

DMA(Direct Memory Access)—直接存储器存取，是单片机的一个外设，它的主要功能是用来搬数据，但是不需要占用 CPU，即在传输数据的时候，CPU 可以干其他的事情，好像是多线程一样。数据传输支持从外设到存储器或者存储器到存储器，这里的存储器可以是 SRAM 或者是 FLASH。DMA 控制器包含了 DMA1 和 DMA2，其中 DMA1 有 7 个通道，DMA2 有 5 个通道，这里的通道可以理解为传输数据的一种管道。要注意的是 DMA2 只存在于大容量的单片机中。

20.2 DMA 功能框图

DMA 控制器独立于内核，属于一个单独的外设，结构比较简单，从编程的角度来看，我们只需掌握功能框图中的三部分内容即可，具体见图 *21_1*：DMA 控制器的框图。



20.2.1 DMA 请求

如果外设要想通过 DMA 来传输数据，必须先给 DMA 控制器发送 DMA 请求，DMA 收到请求信号之后，控制器会给外设一个应答信号，当外设应答后且 DMA 控制器收到应答信号之后，就会启动 DMA 的传输，直到传输完毕。

DMA 有 DMA1 和 DMA2 两个控制器，DMA1 有 7 个通道，DMA2 有 5 个通道，不同的 DMA 控制器的通道对应着不同的外设请求，这决定了我们在软件编程上该怎么设置，具体见 DMA 请求

映像表。

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I^2S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

图 21-2 DMA1 各个通道的请求映像

外设	通道1	通道2	通道3	通道4	通道5
ADC3(1)					ADC3
SPI/I2S3	SPI/I2S3_RX	SPI/I2S3_TX			
UART4			UART4_RX		UART4_TX
SDIO(1)				SDIO	
TIM5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP		TIM5_CH2	TIM5_CH1
TIM6/ DAC通道1			TIM6_UP/ DAC通道1		
TIM7/ DAC通道2				TIM7_UP/ DAC通道2	
TIM8(1)	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	TIM8_CH1		TIM8_CH2

图 21-3 DMA2 各个通道的请求映像

其中 ADC3、SDIO 和 TIM8 的 DMA 请求只在大容量产品中存在，这个在具体项目时要注意。

20.2.2 通道

DMA 具有 12 个独立可编程的通道，其中 DMA1 有 7 个通道，DMA2 有 5 个通道，每个通道对应不同的外设的 DMA 请求。虽然每个通道可以接收多个外设的请求，但是同一时间只能接收一个，不能同时接收多个。

20.2.3 仲裁器

当发生多个 DMA 通道请求时，就意味着有先后响应处理的顺序问题，这个就由仲裁器也管理。仲裁器管理 DMA 通道请求分为两个阶段。第一阶段属于软件阶段，可以在 DMA_CCRx 寄存器中设置，有 4 个等级：非常高、高、中和低四个优先级。第二阶段属于硬件阶段，如果两个或以上的 DMA 通道请求设置的优先级一样，则他们优先级取决于通道编号，编号越低优先权越高，比如通道 0 高于通道 1。在大容量产品和互联型产品中，DMA1 控制器拥有高于 DMA2 控制器的优先级。

20.3 DMA 数据配置

使用 DMA，最核心就是配置要传输的数据，包括数据从哪里来，要到哪里去，传输的数据的单位是什么，要传多少数据，是一次传输还是循环传输等等。

20.3.1 从哪里来到哪里去

我们知道 DMA 传输数据的方向有三个：从外设到存储器，从存储器到外设，从存储器到存储器。具体的方向 DMA_CCR 位 4 DIR 配置：0 表示从外设到存储器，1 表示从存储器到外设。这里面涉及到的外设地址由 DMA_CPAR 配置，存储器地址由 DMA_CMAR 配置。

20.3.1.1 外设到存储器

当我们使用从外设到存储器传输时，以 ADC 采集为例。DMA 外设寄存器的地址对应的就是 ADC 数据寄存器的地址，DMA 存储器的地址就是我们自定义的变量（用来接收存储 AD 采集的数据）的地址。方向我们设置外设为源地址。

20.3.1.2 存储器到外设

当我们使用从存储器到外设传输时，以串口向电脑端发送数据为例。DMA 外设寄存器的地址对应的就是串口数据寄存器的地址，DMA 存储器的地址就是我们自定义的变量（相当于一个缓冲区，用来存储通过串口发送到电脑的数据）的地址。方向我们设置外设为目标地址。

20.3.1.3 存储器到存储器

当我们使用从存储器到存储器传输时，以内部 FLASH 向内部 SRAM 复制数据为例。DMA 外设寄存器的地址对应的就是内部 FLASH（我们这里把内部 FALSH 当作一个外设来看）的地址，DMA 存储器的地址就是我们自定义的变量（相当于一个缓冲区，用来存储来自内部 FLASH 的数据）的地址。方向我们设置外设（即内部 FLASH）为源地址。跟上面两个不一样的是，这里需要把 DMA_CCR 位 14：MEM2MEM：存储器到存储器模式配置为 1，启动 M2M 模式。

20.3.2 要传多少，单位是什么

当我们配置好数据要从哪里来到哪里去之后，我们还需要知道我们要传输的数据是多少，数据的单位是什么。

以串口向电脑发送数据为例，我们可以一次性给电脑发送很多数据，具体多少由 DMA_CNDTR 配置，这是一个 32 位的寄存器，一次最多只能传输 65535 个数据。

要想数据传输正确，源和目标地址存储的数据宽度还必须一致，串口数据寄存器是 8 位的，所以我们定义的要发送的数据也必须是 8 位。外设的数据宽度由 DMA_CCR 的 PSIZE[1:0] 配置，可以是 8/16/32 位，存储器的数据宽度由 DMA_CCR 的 MSIZE[1:0] 配置，可以是 8/16/32 位。

在 DMA 控制器的控制下，数据要想有条不紊的从一个地方搬到另外一个地方，还必须正确设置两边数据指针的增量模式。外设的地址指针由 DMA_CCRx 的 PINC 配置，存储器的地址指针由 MINC 配置。以串口向电脑发送数据为例，要发送的数据很多，每发送完一个，那么存储器的地址指针就应该加 1，而串口数据寄存器只有一个，那么外设的地址指针就固定不变。具体的数据指针的增量模式由实际情况决定。

20.3.3 什么时候传输完成

数据什么时候传输完成，我们可以通过查询标志位或者通过中断的方式来鉴别。每个 DMA 通道在 DMA 传输过半、传输完成和传输错误时都会有相应的标志位，如果使能了该类型的中断后，则会产生中断。有关各个标志位的详细描述请参考 DMA 中断状态寄存器 DMA_ISR 的详细描述。

传输完成还分两种模式，是一次传输还是循环传输，一次传输很好理解，即是传输一次之后就停止，要想再传输的话，必须关断 DMA 使能后再重新配置后才能继续传输。循环传输则是一次传输完成之后又恢复第一次传输时的配置循环传输，不断的重复。具体的由 DMA_CCR 寄存器的 CIRC 循环模式位控制。

20.4 DMA 初始化结构体详解

HAL 库函数对每个外设都建立了一个初始化结构体 xxx_InitTypeDef(xxx 为外设名称)，结构体成员用于设置外设工作参数，并由 HAL 库函数 xxx_Init() 调用这些设定参数进入设置外设相应的寄存器，达到配置外设工作环境的目的。

结构体 xxx_InitTypeDef 和库函数 xxx_Init 配合使用是 HAL 库精髓所在，理解了结构体 xxx_InitTypeDef 每个成员意义基本上就可以对该外设运用自如。结构体 xxx_InitTypeDef 定义在 stm32f1xx_hal_xxx.h(后面 xxx 为外设名称) 文件中，库函数 xxx_Init 定义在 stm32f1xx_xxx.c 文件中，编程时我们可以结合这两个文件内注释使用。

20.4.1 DMA_InitTypeDef 初始化结构体

```
typedef struct {

    uint32_t Direction;              //传输方向
    uint32_t PeriphInc;              //外设递增
    uint32_t MemInc;                 //存储器递增
    uint32_t PeriphDataAlignment;    //外设数据宽度
    uint32_t MemDataAlignment;       //存储器数据宽度
    uint32_t Mode;                   //模式选择
    uint32_t Priority;               //优先级

} DMA_InitTypeDef;
```

1) Direction：传输方向选择，可选外设到存储器、存储器到外设以及存储器到存储器。它设定 DMA_SxCR 寄存器的 DIR[1:0] 位的值。ADC 采集显然使用外设到存储器模式。

2) PeripheralInc：如果配置为 DMA_PINC_ENABLE，使能外设地址自动递增功能，它设定 DMA_CCR 寄存器的 PINC 位的值；一般外设都是只有一个数据寄存器，所以一般不会使能该位。

3) MemoryInc：如果配置为 DMA_MINC_ENABLE，使能存储器地址自动递增功能，它设定 DMA_CCR 寄存器的 MINC 位的值；我们自定义的存储区一般都是存放多个数据的，所以要使能存储器地址自动递增功能。

4) PeriphDataAlignment：外设数据宽度，可选字节 (8 位)、半字 (16 位) 和字 (32 位)，它设定 DMA_SxCR 寄存器的 PSIZE[1:0] 位的值。ADC 数据寄存器只有低 16 位数据有效，使用半字数据宽度。

5) Mode：DMA 传输模式选择，可选一次传输或者循环传输，它设定 DMA_SxCR 寄存器的 CIRC 位的值。我们希望 ADC 采集是持续循环进行的，所以使用循环传输模式。

6) 软件设置数据流的优先级，有 4 个可选优先级分别为非常高、高、中和低，它设定 DMA_SxCR 寄存器的 PL[1:0] 位的值。DMA 优先级只有在多个 DMA 数据流同时使用时才有意义，这里我们设置为非常高优先级就可以了。

20.5 DMA 存储器到存储器模式实验

本章只讲解存储器到存储器和存储器到外设这两种模式，其他功能模式在其他章节使用到的时候再讲。存储器到存储器模式可以实现数据在两个内存的快速拷贝。我们先定义一个静态的源数据，存放在内部 FLASH，然后使用 DMA 传输把源数据拷贝到目标地址上（内部 SRAM），最后对比源数据和目标地址的数据，看看是否传输准确。

20.5.1 硬件设计

DMA 存储器到存储器实验不需要其他硬件要求，只用到 RGB 彩色灯用于指示程序状态。

20.5.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。这个实验代码比较简单，主要程序代码都在 main.c 文件中。

20.5.2.1 编程要点

1) 使能 DMA 时钟；
2) 配置 DMA 数据参数；
3) 使能 DMA，进行传输；
4) 等待传输完成，并对源数据和目标地址数据进行比较。

20.5.2.2 代码分析

20.5.2.2.1 DMA 宏定义及相关变量定义

列表 1: 代码清单 21-1 DMA 数据流和相关变量定义

```
/* 相关宏定义，使用存储器到存储器传输必须使用 DMA2 */
DMA_HandleTypeDef DMA_Handle;

#define DMA_STREAM                  DMA2_Stream0
#define DMA_CHANNEL                 DMA_CHANNEL_0
#define DMA_STREAM_CLOCK()          __DMA2_CLK_ENABLE()

#define BUFFER_SIZE                 32

/* 定义 aSRC_Const_Buffer 数组作为 DMA 传输数据源
const 关键字将 aSRC_Const_Buffer 数组变量定义为常量类型 */
const uint32_t aSRC_Const_Buffer[BUFFER_SIZE]= {
    0x01020304,0x05060708,0x090A0B0C,0x0D0E0F10,
    0x11121314,0x15161718,0x191A1B1C,0x1D1E1F20,
    0x21222324,0x25262728,0x292A2B2C,0x2D2E2F30,
    0x31323334,0x35363738,0x393A3B3C,0x3D3E3F40,
    0x41424344,0x44564748,0x494A4B4C,0x4D4E4F50,
    0x51525345,0x55565758,0x595A5B5C,0x5D5E5F60,
    0x61626364,0x65666768,0x696A6B6C,0x6D6E6F70,
    0x71727374,0x75767778,0x797A7B7C,0x7D7E7F80
};
/* 定义 DMA 传输目标存储器 */
uint32_t aDST_Buffer[BUFFER_SIZE];
```

使用宏定义设置外设配置方便程序修改和升级。

存储器到存储器传输通道没有硬性规定，可以随意选择。

aSRC_Const_Buffer[BUFFER_SIZE] 定义用来存放源数据，并且使用了 const 关键字修饰，即常量类型，使得变量是存储在内部 flash 空间上。

20.5.2.2.2 DMA 数据配置

列表 2: 代码清单 21-2 DMA 传输参数配置

```
static void DMA_Config(void)
{
    HAL_StatusTypeDef DMA_status = HAL_ERROR;

    DMA_STREAM_CLOCK();
    //数据流选择
    DMA_Handle.Instance=DMA_STREAM;
    //存储器到外设 HAL_DMA_Init(&DMA_Handle);
    DMA_Handle.Init.Direction=DMA_MEMORY_TO_MEMORY;
    //外设非增量模式/* Associate the DMA handle */
    DMA_Handle.Init.PeriphInc=DMA_PINC_ENABLE;
    //存储器增量模式__HAL_LINKDMA(&UartHandle, hdmatx,DMA_Handle);
    DMA_Handle.Init.MemInc=DMA_MINC_ENABLE;
    //外设数据长度:8 位
    DMA_Handle.Init.PeriphDataAlignment=DMA_PDATAALIGN_WORD;
    //存储器数据长度:8 位
    DMA_Handle.Init.MemDataAlignment=DMA_MDATAALIGN_WORD;
    //外设普通模式
    DMA_Handle.Init.Mode=DMA_NORMAL;
    //中等优先级
    DMA_Handle.Init.Priority=DMA_PRIORITY_MEDIUM;

    /* 完成 DMA 数据流参数配置 */
    HAL_DMA_Init(&DMA_Handle);
    DMA_status = HAL_DMA_Start(&DMA_Handle,(uint32_t)aSRC_Const_Buffer,
                (uint32_t)aDST_Buffer,BUFFER_SIZE);
    /* 判断 DMA 状态 */
    if (DMA_status != HAL_OK) {
        /* DMA 出错就让程序运行下面循环: RGB 彩色灯闪烁 */
```

(下页继续)

(续上页)

```
        while (1) {
            LED_RED;
            Delay(0xFFFFFF);
            LED_RGBOFF;
            Delay(0xFFFFFF);
        }
    }
}
```

使用 DMA_DMA_HandleTypeDef 结构体定义一个 DMA 数据流初始化变量，这个结构体内容我们之前已经有详细讲解。

调用 DMA_STREAM_CLOCK 函数开启 DMA 数据流时钟，使用 DMA 控制器之前必须开启对应的时钟。

存储器到存储器模式通道选择没有具体规定，只能使用一次传输模式不能循环传输，最后我调用 HAL_DMA_Init 函数完成 DMA 数据流的初始化配置。

HAL_DMA_Start 函数用于启动 DMA 数据流传输，源地址和目标地址使用之前定义的数组首地址，返回 DMA 传输状态。

如果 DMA 传输没有就绪就会闪烁 RGB 彩灯提示。

20.5.2.2.3 存储器数据对比

列表 3: 代码清单 21-3 源数据与目标地址数据对比

```
uint8_t Buffercmp(const uint32_t* pBuffer,
                  uint32_t* pBuffer1, uint16_t BufferLength)
{
    /* 数据长度递减 */
    while (BufferLength--) {
        /* 判断两个数据源是否对应相等 */
```

(下页继续)

(续上页)

```
        if (*pBuffer != *pBuffer1) {
            /* 对应数据源不相等马上退出函数，并返回 0 */
            return 0;
        }
        /* 递增两个数据源的地址指针 */
        pBuffer++;
        pBuffer1++;
    }
    /* 完成判断并且对应数据相对 */
    return 1;
}
```

判断指定长度的两个数据源是否完全相等，如果完全相等返回 1；只要其中一对数据不相等返回 0。它需要三个形参，前两个是两个数据源的地址，第三个是要比较数据长度。

20.5.2.2.4 主函数

列表 4: 代码清单 21-4 存储器到存储器模式主函数

```
int main(void)
{
    /* 定义存放比较结果变量 */
    uint8_t TransferStatus;
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 设置 RGB 彩色灯为紫色 */
    LED_PURPLE;

    /* 简单延时函数 */
```

(下页继续)

(续上页)

```
    Delay(0xFFFFFF);

    /* DMA 传输配置 */
    DMA_Config();

    /* 等待 DMA 传输完成 */
    while (__HAL_DMA_GET_FLAG(&DMA_Handle,DMA_FLAG_TC6)==DISABLE) {
    }
    /* 比较源数据与传输后数据 */
    TransferStatus=Buffercmp(aSRC_Const_Buffer, aDST_Buffer,BUFFER_SIZE);
    /* 判断源数据与传输后数据比较结果 */
    if (TransferStatus==0) {
        /*
        源数据与传输后数据不相等时 RGB 彩色灯显示红色
        */
        LED_RED;
    } else {
        /*
        源数据与传输后数据相等时 RGB 彩色灯显示蓝色
        */
        LED_BLUE;
    }
    while (1) {
    }
}
```

首先定义一个变量用来保存存储器数据比较结果。

SystemClock_Config 函数初始化系统时钟。

RGB 彩色灯用来指示程序进程，使用之前需要初始化它，LED_GPIO_Config 定义在 bsp_led.c 文件中。开始设置 RGB 彩色灯为紫色，LED_PURPLE 是定义在 bsp_led.h 文件的一个宏定义。

Delay 函数只是一个简单的延时函数。

调用 DMA_Config 函数完成 DMA 数据流配置并启动 DMA 数据传输。

__HAL_DMA_GET_FLAG 函数获取 DMA 数据流事件标志位的当前状态，这里获取 DMA 数据传输完成这个标志位，使用循环持续等待直到该标志位被置位，即 DMA 传输完成这个事件发生，然后退出循环，运行之后程序。

确定 DMA 传输完成之后就可以调用 Buffercmp 函数比较源数据与 DMA 传输后目标地址的数据是否一一对应。TransferStatus 保存比较结果，如果为 1 表示两个数据源一一对应相等说明 DMA 传输成功；相反，如果为 0 表示两个数据源数据存在不等情况，说明 DMA 传输出错。

如果 DMA 传输成功设置 RGB 彩色灯为蓝色，如果 DMA 传输出错设置 RGB 彩色灯为红色。

20.5.2.3 下载验证

确保开发板供电正常，编译程序并下载。观察 RGB 彩色灯变化情况。正常情况下 RGB 彩色灯先为紫色，然后变成蓝色。如果 DMA 传输出错才会为红色。

20.6 DMA 存储器到外设模式实验

上个实验我们讲了 DMA 存储器到存储器模式，接下来我们再讲一个存储器到外设的实验。我们先定义一个数据变量，存于 SRAM 中，然后通过 DMA 的方式传输到串口的数据寄存器，然后通过串口把这些数据发送到电脑的上位机显示出来。

20.6.1 硬件设计

存储器到外设模式使用到 USART1 功能，具体电路设置参考 USART 章节，无需其他硬件设计。

20.6.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们编写两个串口驱动文件 bsp_usart_dma.c 和 bsp_usart_dma.h，有关串口和 DMA 的宏定义以及驱动函数都在里边。

20.6.2.1 编程要点

1) 配置 USART 通信功能；
2) 设置串口 DMA 工作参数；
3) 使能 DMA；
4) DMA 传输同时 CPU 可以运行其他任务。

20.6.2.2 代码分析

20.6.2.2.1 USART 和 DMA 宏定义

列表 5: 代码清单 21-5 USART 和 DMA 相关宏定义

```
//串口波特率
#define DEBUG_USART_BAUDRATE                    115200
//引脚定义
/************************************************************/
#define DEBUG_USART                             USART1
#define DEBUG_USART_CLK_ENABLE()                __USART1_CLK_ENABLE();

#define DEBUG_USART_RX_GPIO_PORT                GPIOA
#define DEBUG_USART_RX_GPIO_CLK_ENABLE()        __GPIOA_CLK_ENABLE()
#define DEBUG_USART_RX_PIN                      GPIO_PIN_10
#define DEBUG_USART_RX_AF                       GPIO_AF7_USART1
```

(下页继续)

(续上页)

```
#define DEBUG_USART_TX_GPIO_PORT                    GPIOA
#define DEBUG_USART_TX_GPIO_CLK_ENABLE()            __GPIOA_CLK_ENABLE()
#define DEBUG_USART_TX_PIN                          GPIO_PIN_9
#define DEBUG_USART_TX_AF                           GPIO_AF7_USART1

#define DEBUG_USART_IRQHandler                      USART1_IRQHandler
#define DEBUG_USART_IRQ                             USART1_IRQn
/************************************************************/
//DMA
#define SENDBUFF_SIZE                             10//发送的数据量
#define DEBUG_USART_DMA_CLK_ENABLE()              __DMA2_CLK_ENABLE()
#define DEBUG_USART_DMA_CHANNEL                   DMA_CHANNEL_4
#define DEBUG_USART_DMA_STREAM                    DMA2_Stream7
```

使用宏定义设置外设配置方便程序修改和升级。

USART 部分设置与 USART 章节内容相同，可以参考 USART 章节内容理解。

查阅图 *21_3* 可知、DMA2 的数据流 7 通道 4。

20.6.2.2.2 串口 DMA 传输配置

列表 6: 代码清单 21-6 USART1 发送请求 DMA 设置

```
void USART_DMA_Config(void)
{
    DEBUG_USART_DMA_CLK_ENABLE();
    //数据流选择
    DMA_Handle.Instance=DEBUG_USART_DMA_STREAM;
    //存储器到外设 HAL_DMA_Init(&DMA_Handle);
    DMA_Handle.Init.Direction=DMA_MEMORY_TO_PERIPH;
```

(下页继续)

(续上页)

```
    //外设非增量模式/* Associate the DMA handle */
    DMA_Handle.Init.PeriphInc=DMA_PINC_DISABLE;
    //存储器增量模式__HAL_LINKDMA(&UartHandle, hdmatx, DMA_Handle);
    DMA_Handle.Init.MemInc=DMA_MINC_ENABLE;
    //外设数据长度:8 位
    DMA_Handle.Init.PeriphDataAlignment=DMA_PDATAALIGN_BYTE;
    //存储器数据长度:8 位
    DMA_Handle.Init.MemDataAlignment=DMA_MDATAALIGN_BYTE;
    //外设普通模式
    DMA_Handle.Init.Mode=DMA_NORMAL;
    //中等优先级
    DMA_Handle.Init.Priority=DMA_PRIORITY_MEDIUM;
    HAL_DMA_Init(&DMA_Handle);
    /* Associate the DMA handle */
    __HAL_LINKDMA(&UartHandle, hdmatx, DMA_Handle);
}
```

首先定义一个 DMA 初始化变量，用来填充 DMA 的参数，然后使能 DMA 时钟。

因为数据是从存储器到串口，所以设置存储器为源地址，串口的数据寄存器为目标地址，要发送的数据有很多且都先存储在存储器中，则存储器地址指针递增，串口数据寄存器只有一个，则外设地址地址不变，两边数据单位设置成一致，传输模式可选一次或者循环传输，只有一个 DMA 请求，优先级随便设，最后调用 HAL_DMA_Init 函数把这些参数写到 DMA 的寄存器中，然后使能 DMA 开始传输。

__HAL_LINKDMA 函数用于链接 DMA 数据流及通道到串口外设通道上。

20.6.2.2.3 主函数

列表 7: 代码清单 21-7 存储器到外设模式主函数

```
int main(void)
{
    uint16_t i;
    SystemClock_Config();
    /* 初始化 USART */
    Debug_USART_Config();
    /* 配置使用 DMA 模式 */
    USART_DMA_Config();
    /* 配置 RGB 彩色灯 */
    LED_GPIO_Config();
    printf("\r\n USART1 DMA TX 测试 \r\n");
    /* 填充将要发送的数据 */
    for (i=0; i<SENDBUFF_SIZE; i++) {
        SendBuff[i]  = 'A';
    }
    /* 为演示 DMA 持续运行而 CPU 还能处理其它事情，持续使用

    DMA 发送数据，量非常大，长时间运行可能会导致电端串口

    口调试助手会卡死，鼠标乱飞的情况，或把 DMA 配置中的循

    环模式改为单次模式 */

    /* USART1 向 DMA 发出 TX 请求 */
    HAL_UART_Transmit_DMA(&UartHandle, (uint8_t *)SendBuff ,
                          SENDBUFF_SIZE);

    /* 此时 CPU 是空闲的，可以干其他的事情 */
    //例如同时控制 LED
```

(下页继续)

(续上页)

```
    while (1) {
        LED1_TOGGLE
        Delay(0xFFFFF);
    }
}
```

SystemClock_Config 函数初始化系统时钟。

Debug_USART_Config 函数定义在 bsp_usart_dma.c 中，它完成 USART 初始化配置，包括 GPIO 初始化，USART 通信参数设置等等，具体可参考 USART 章节讲解。

USART_DMA_Config 函数也是定义在 bsp_usart_dma.c 中，之前我们已经详细分析了。

LED_GPIO_Config 函数定义在 bsp_led.c 中，它完成 RGB 彩色灯初始化配置，具体可参考 GPIO 章节讲解。

使用 for 循环填充源数据，SendBuff[SENDBUFF_SIZE] 是一个全局无符号 8 位整数数组，是 DMA 传输的源数据。

HAL_UART_Transmit_DMA 函数用于启动 USART 的 DMA 传输。只需要指定源数据地址及长度，运行该函数后 USART 的 DMA 发送传输就开始了，根据配置它会通过 USART 循环发送数据。

DMA 传输过程是不占用 CPU 资源的，可以一边传输一次运行其他任务。

20.6.2.3 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板的 USB 转串口和电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。程序运行后在串口调试助手可接收到大量的数据，同时开发板上 RGB 彩色灯不断闪烁。

这里要注意为演示 DMA 持续运行并且 CPU 还能处理其它事情，持续使用 DMA 发送数据，量非常大，长时间运行可能会导致电脑端串口调试助手会卡死，鼠标乱飞的情况，所以在测试时最好把串口调试助手的自动清除接收区数据功能勾选上或把 DMA 配置中的循环模式改为单次模式。

第 21 章　常用存储器介绍

21.1 存储器种类

存储器是计算机结构的重要组成部分。存储器是用来存储程序代码和数据的部件，有了存储器计算机才具有记忆功能。基本的存储器种类见图 *22_1*。

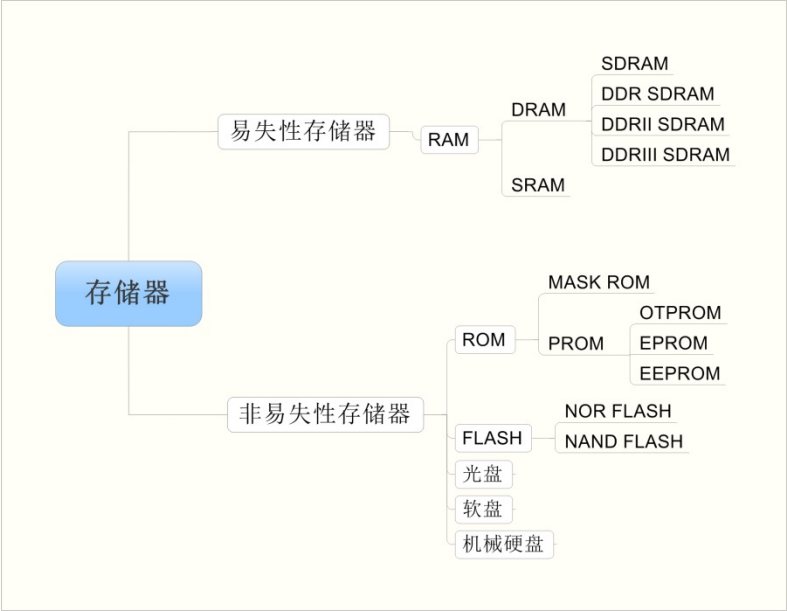


图 22-1 基本存储器种类

存储器按其存储介质特性主要分为“易失性存储器”和“非易失性存储器”两大类。其中的“易失/非易失”是指存储器断电后，它存储的数据内容是否会丢失的特性。由于一般易失性存储器存取速度快，而非易失性存储器可长期保存数据，它们都在计算机中占据着重要角色。在计算机中易失性存储器最典型的代表是内存，非易失性存储器的代表则是硬盘。

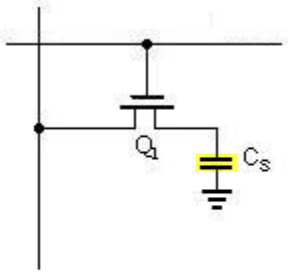
21.2 RAM 存储器

RAM 是“Random Access Memory”的缩写，被译为随机存储器。所谓“随机存取”，指的是当存储器中的消息被读取或写入时，所需要的时间与这段信息所在的位置无关。这个词的由来是因为早期计算机曾使用磁鼓作为存储器，磁鼓是顺序读写设备，而 RAM 可随读取其内部任意地址的数据，时间都是相同的，因此得名。实际上现在 RAM 已经专门用于指代作为计算机内存的易失性半导体存储器。

根据 RAM 的存储机制，又分为动态随机存储器 DRAM(Dynamic RAM) 以及静态随机存储器 SRAM(Static RAM) 两种。

21.2.1 DRAM

动态随机存储器 DRAM 的存储单元以电容的电荷来表示数据，有电荷代表 1，无电荷代表 0，见图 *22_2*。但时间一长，代表 1 的电容会放电，代表 0 的电容会吸收电荷，因此它需要定期刷新操作，这就是“动态 (Dynamic)”一词所形容的特性。刷新操作会对电容进行检查，若电量大于满电量的 1/2，则认为其代表 1，并把电容充满电；若电量小于 1/2，则认为其代表 0，并把电容放电，藉此来保证数据的正确性。



DRAM存储单元

图 22-2 DRAM 存储单元

21.2.1.1 SDRAM

根据 DRAM 的通讯方式，又分为同步和异步两种，这两种方式根据通讯时是否需要使用时钟信号来区分。图 *22_3* 是一种利用时钟进行同步的通讯时序，它在时钟的上升沿表示有效数据。

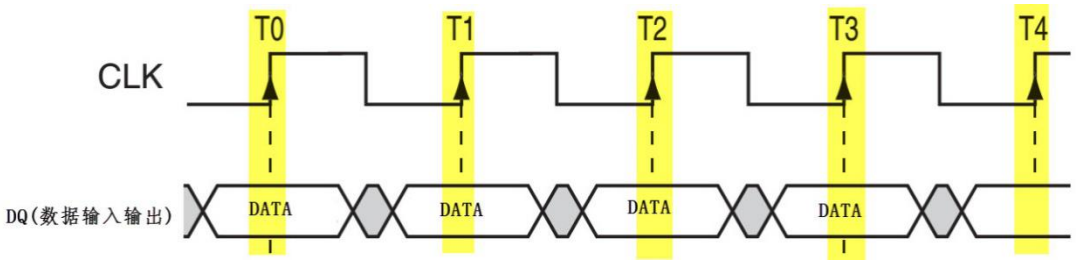


图 22-3 同步通讯时序图

由于使用时钟同步的通讯速度更快，所以同步 DRAM 使用更为广泛，这种 DRAM 被称为 SDRAM(Synchronous DRAM)。

21.2.1.2 DDR SDRAM

为了进一步提高 SDRAM 的通讯速度，人们设计了 DDR SDRAM 存储器 (Double Data Rate SDRAM)。它的存储特性与 SDRAM 没有区别，但 SDRAM 只在上升沿表示有效数据，在 1 个时钟周期内，只能表示 1 个有数据；而 DDR SDRAM 在时钟的上升沿及下降沿各表示一个数据，也就是说在 1 个时钟周期内可以表示 2 位数据，在时钟频率同样的情况下，提高了一倍的速度。至于 DDRII 和 DDRIII，它们的通讯方式并没有区别，主要是通讯同步时钟的频率提高了。

当前个人计算机常用的内存条是 DDRIII SDRAM 存储器，在一个内存条上包含多个 DDRIII SDRAM 芯片。

21.2.2 SRAM

静态随机存储器 SRAM 的存储单元以锁存器来存储数据，见图 *22_4*。这种电路结构不需要定时刷新充电，就能保持状态 (当然，如果断电了，数据还是会丢失的)，所以这种存储器被称为“静态 (Static)” RAM。部分批次 SRAM 芯片替换为 AS6C8016 型号时序与 ISSI 的芯片兼容

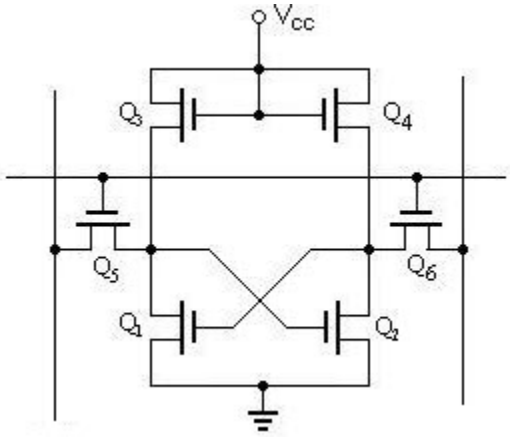


图 22-4 SRAM 存储单元

同样地，SRAM 根据其通讯方式也分为同步 (SSRAM) 和异步 SRAM，相对来说，异步 SRAM 用得比较广泛。

21.2.3 DRAM 与 SRAM 的应用场合

对比 DRAM 与 SRAM 的结构，可知 DRAM 的结构简单得多，所以生产相同容量的存储器，DRAM 的成本要更低，且集成度更高。而 DRAM 中的电容结构则决定了它的存取速度不如 SRAM，特性对比见表 22-1。

表 22-1 DRAM 与 SRAM 对比

特性	DRAM	SRAM
存取速度	较慢	较快
集成度	较高	较低
生产成本	较低	较高
是否需要刷新	是	否

所以在实际应用场合中，SRAM 一般只用于 CPU 内部的高速缓存 (Cache)，而外部扩展的内存一般使用 DRAM。在 STM32 系统的控制器中，只有 STM32F429 型号或更高级的芯片才支持扩展 SDRAM，其它型号如 STM32F1、STM32F2 及 STM32F103 等型号只能扩展 SRAM。

21.3 非易失性存储器

非易失性存储器种类非常多，半导体类的有 ROM 和 FLASH，而其它的则包括光盘、软盘及机械硬盘。

21.3.1 ROM 存储器

ROM 是“Read Only Memory”的缩写，意为只能读的存储器。由于技术的发展，后来设计出了可以方便写入数据的 ROM，而这个“Read Only Memory”的名称被沿用下来了，现在一般用于指代非易失性半导体存储器，包括后面介绍的 FLASH 存储器，有些人也把它归到 ROM 类里边。

21.3.1.1 MASK ROM

MASK(掩膜) ROM 就是正宗的“Read Only Memory”，存储在它内部的数据是在出厂时使用特殊工艺固化的，生产后就不可修改，其主要优势是大批量生产时成本低。当前在生产量大，数据不需要修改的场合，还有应用。

21.3.1.2 OTPROM

OTPROM(One Time Programable ROM) 是一次可编程存储器。这种存储器出厂时内部并没有资料，用户可以使用专用的编程器将自己的资料写入，但只能写入一次，被写入过后，它的内容也不可再修改。在 NXP 公司生产的控制器芯片中常使用 OTPROM 来存储密钥；在 STM32F429 芯片中也具有一部分 OTPROM 空间。

21.3.1.3 EPROM

EPROM(Erasable Programmable ROM) 是可重复擦写的存储器，它解决了 PROM 芯片只能写入一次的问题。这种存储器使用紫外线照射芯片内部擦除数据，擦除和写入都要专用的设备。现在这种存储器基本淘汰，被 EEPROM 取代。

21.3.1.4 EEPROM

EEPROM(Electrically Erasable Programmable ROM) 是电可擦除存储器。EEPROM 可以重复擦写，它的擦除和写入都是直接使用电路控制，不需要再使用外部设备来擦写。而且可以按字节为单位修改数据，无需整个芯片擦除。现在主要使用的 ROM 芯片都是 EEPROM。

21.3.2 FLASH 存储器

FLASH 存储器又称为闪存，它也是可重复擦写的储器，部分书籍会把 FLASH 存储器称为 FLASH ROM，但它的容量一般比 EEPROM 大得多，且在擦除时，一般以多个字节为单位。如有的 FLASH 存储器以 4096 个字节为扇区，最小的擦除单位为一个扇区。根据存储单元电路的不同，FLASH 存储器又分为 NOR FLASH 和 NAND FLASH，见表 22-2。

表 22-2 NOR FLASH 与 NAND FLASH 特性对比

特性	NOR FLASH	NAND FLASH
同容量存储器成本	较贵	较便宜
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
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
坏块	较少	较多
是否支持 XIP	支持	不支持

NOR 与 NAND 的共性是在数据写入前都需要有擦除操作，而擦除操作一般是以“扇区/块”为单位的。而 NOR 与 NAND 特性的差别，主要是由于其内部“地址/数据线”是否分开导致的。

由于 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。

而由于 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。表 22-2 中的最后一项“是否支持 XIP”描述的就是这种立即执行的特性 (eXecute In Place)。

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。

另外，FLASH 的擦除次数都是有限的 (现在普遍是 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，而且由于擦写过程复杂，从整体来说 NOR 块块更少，寿命更长。由于可能存在坏块，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。

由于两种 FLASH 存储器特性的差异，NOR FLASH 一般应用在代码存储的场合，如嵌入式控制器内部的程序存储空间。而 NAND FLASH 一般应用在大数据量存储的场合，包括 SD 卡、U 盘以及固态硬盘等，都是 NAND FLASH 类型的。

在本教程中会对如何使用 RAM、EEPROM、FLASH 存储器进行实例讲解。

第 22 章　I2C—读写 EEPROM

本章参考资料：《STM32F10X-中文参考手册》I2C 章节及《I2C 总线协议》。

若对 I2C 通讯协议不了解，可先阅读《I2C 总线协议》文档的内容学习。

关于 EEPROM 存储器，请参考“常用存储器介绍”章节，实验中的 EEPROM，请参考其规格书《AT24C02》来了解。

22.1 I2C 协议简介

I2C 通讯协议 (Inter - Integrated Circuit) 是由 Phiilps 公司开发的，由于它引脚少，硬件实现简单，可扩展性强，不需要 USART、CAN 等通讯协议的外部收发设备，现在被广泛地使用在系统内多个集成电路 (IC) 间的通讯。

在计算机科学里，大部分复杂的问题都可以通过分层来简化。如芯片被分为内核层和片上外设；STM32HAL 库则是在寄存器与用户代码之间的软件层。对于通讯协议，我们也以分层的方式来理解，最基本的是把它分为物理层和协议层。物理层规定通讯系统中具有机械、电子功能部分的特性，确保原始数据在物理媒体的传输。协议层主要规定通讯逻辑，统一收发双方的数据打包、解包标准。简单来说物理层规定我们用嘴巴还是用肢体来交流，协议层则规定我们用中文还是英文来交流。

下面我们分别对 I2C 协议的物理层及协议层进行讲解。

22.1.1 I2C 物理层

I2C 通讯设备之间的常用连接方式见图 *23_1*。

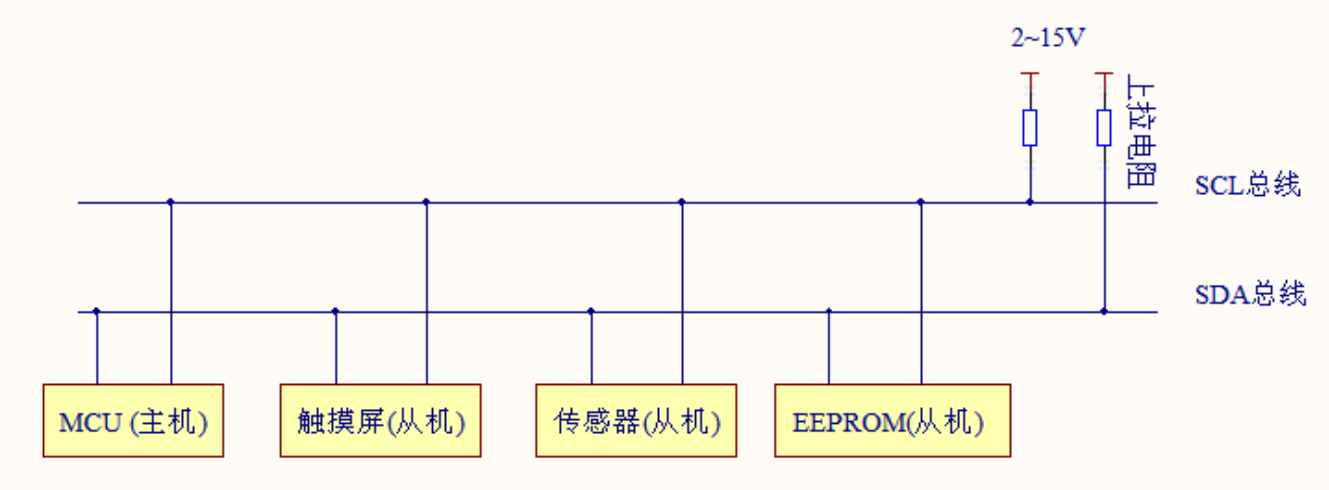


图 23-1 常见的 I2C 通讯系统

它的物理层有如下特点：

(1) 它是一个支持设备的总线。“总线”指多个设备共用的信号线。在一个 I2C 通讯总线中，可连接多个 I2C 通讯设备，支持多个通讯主机及多个通讯从机。

(2) 一个 I2C 总线只使用两条总线线路，一条双向串行数据线 *(SDA)*，一条串行时钟线 *(SCL)*。数据线即用来表示数据，时钟线用于数据收发同步。

(3) 每个连接到总线的设备都有一个独立的地址，主机可以利用这个地址进行不同设备之间的访问。

(4) 总线通过上拉电阻接到电源。当 I2C 设备空闲时，会输出高阻态，而当所有设备都空闲，都输出高阻态时，由上拉电阻把总线拉成高电平。

(5) 多个主机同时使用总线时，为了防止数据冲突，会利用仲裁方式决定由哪个设备占用总线。

(6) 具有三种传输模式：标准模式传输速率为 100kbit/s，快速模式为 400kbit/s，高速模式下可达 3.4Mbit/s，但目前大多 I^2C 设备尚不支持高速模式。

(7) 连接到相同总线的 IC 数量受到总线的最大电容 400pF 限制。

22.1.2 协议层

I2C 的协议定义了通讯的起始和停止信号、数据有效性、响应、仲裁、时钟同步和地址广播等环节。

22.1.2.1 I2C 基本读写过程

先看看 I2C 通讯过程的基本结构，它的通讯过程见图 *23_2*、图 *23_3* 及图 *23_4*。

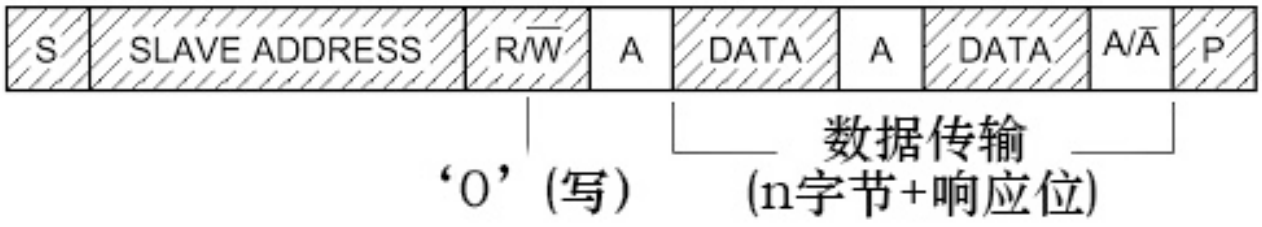


图 23-2 主机写数据到从机

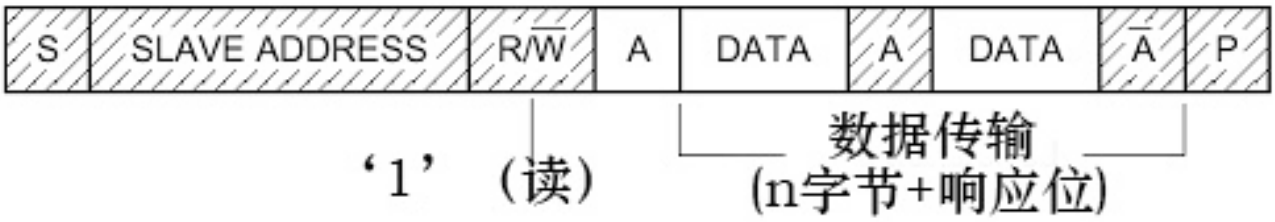


图 23-3 主机由从机中读数据

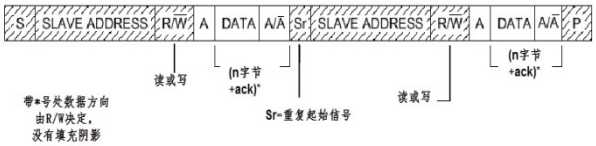


图 23-4 I2C 通讯复合格式

图例：

数据由主机传输至从机

S： 传输开始信号

SLAVE_ADDRESS: 从机地址

数据由从机传输至主机

R/W-： 传输方向选择位，1 为读，0 为写

A/A-： 应答 (ACK) 或非应答 (NACK) 信号

P： 停止传输信号

这些图表示的是主机和从机通讯时，SDA 线的数据包序列。

其中 S 表示由主机的 I2C 接口产生的*传输起始信号 (S)*，这时连接到 I2C 总线上的所有从机都会接收到这个信号。

起始信号产生后，所有从机就开始等待主机紧接下来广播的*从机地址信号 (SLAVE_ADDRESS)*。在 I2C 总线上，每个设备的地址都是唯一的，当主机广播的地址与某个设备地址相同时，这个设备就被选中了，没被选中的设备将会忽略之后的数据信号。根据 I2C 协议，这个从机地址可以是 *7 位*或 *10 位*。

在地址位之后，是传输方向的选择位，该位为 0 时，表示后面的数据传输方向是由主机传输至从机，即主机向从机写数据。该位为 1 时，则相反，即主机由从机读数据。

从机接收到匹配的地址后，主机或从机会返回一个应答 (ACK) 或非应答 (NACK) 信号，只有接收到应答信号后，主机才能继续发送或接收数据。

写数据

若配置的方向传输位为“*写数据*”*方向*，即第一幅图的情况，广播完地址，接收到应答信号后，*主机开始正式向从机传输数据 (DATA)*，数据包的大小为 *8 位*，主机每发送完一个字节数据，都要等待从机的*应答信号 (ACK)*，重复这个过程，可以向从机传输 N 个数据，这个 N 没有大小限制。当数据传输结束时，主机向从机发送一个*停止传输信号 (P)*，表示不再传输数据。

读数据

若配置的方向传输位为“读数据”方向，即第二幅图的情况，广播完地址，接收到应答信号后，从机开始向主机返回数据 *(DATA)*，数据包大小也为 *8* 位，从机每发送完一个数据，都会等待主机的应答信号 *(ACK)*，重复这个过程，可以返回 N 个数据，这个 N 也没有大小限制。当主机希望停止接收数据时，就向从机返回一个非应答信号 *(NACK)*，则从机自动停止数据传输。

读和写数据

除了基本的读写，I2C 通讯更常用的是复合格式，即第三幅图的情况，该传输过程有两次起始信号 (S)。一般在第一次传输中，主机通过 SLAVE_ADDRESS 寻找到从设备后，发送一段“数据”，这段数据通常用于表示从设备内部的寄存器或存储器地址 (注意区分它与 SLAVE_ADDRESS 的区别)；在第二次的传输中，对该地址的内容进行读或写。也就是说，第一次通讯是告诉从机读写地址，第二次则是读写的实际内容。

以上通讯流程中包含的各个信号分解如下：

22.1.2.2 通讯的起始和停止信号

前文中提到的起始 (S) 和停止 (P) 信号是两种特殊的状态，见图 23-5。当 SCL 线是高电平时 SDA 线从高电平向低电平切换，这个情况表示通讯的起始。当 SCL 是高电平时 SDA 线由低电平向高电平切换，表示通讯的停止。起始和停止信号一般由主机产生。

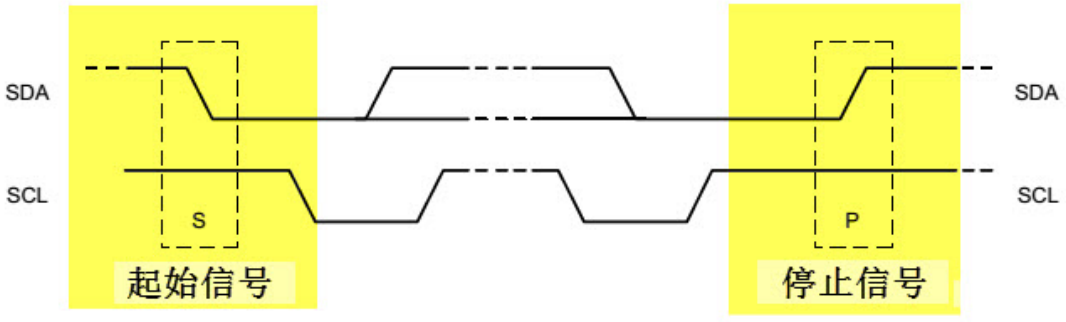


图 23-5 起始和停止信号

22.1.2.3 数据有效性

I2C 使用 SDA 信号线来传输数据，使用 SCL 信号线进行数据同步。见图 *23_6*。SDA 数据线在 SCL 的每个时钟周期传输一位数据。传输时，SCL 为高电平的时候 SDA 表示的数据有效，即此时的 SDA 为高电平时表示数据“1”，为低电平时表示数据“0”。当 SCL 为低电平时，SDA 的数据无效，一般在这个时候 SDA 进行电平切换，为下一次表示数据做好准备。

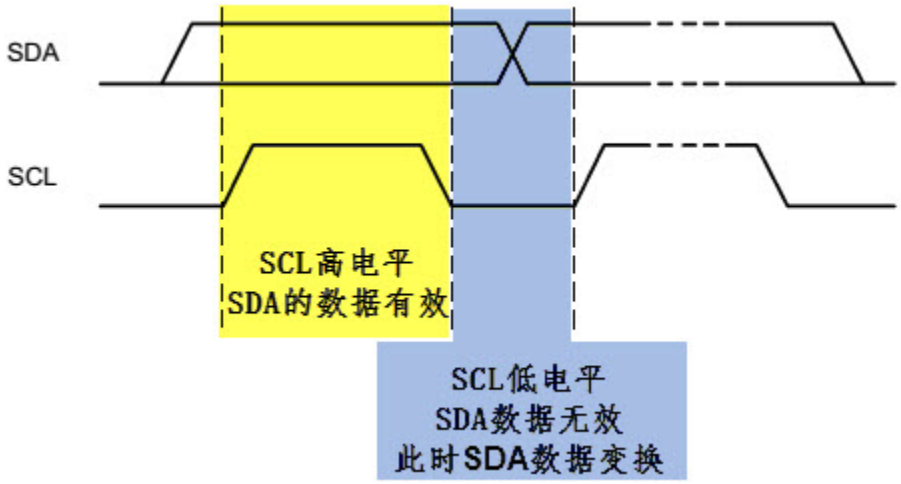


图 23-6 数据有效性

每次数据传输都以字节为单位，每次传输的字节数不受限制。

22.1.2.4 地址及数据方向

I2C 总线上的每个设备都有自己的独立地址，主机发起通讯时，通过 SDA 信号线发送设备地址 (SLAVE_ADDRESS) 来查找从机。I2C 协议规定设备地址可以是 7 位或 10 位，实际中 7 位的地址应用比较广泛。紧跟设备地址的一个数据位用来表示数据传输方向，它是数据方向位 (R/)，第 8 位或第 11 位。数据方向位为“1”时表示主机由从机读数据，该位为“0”时表示主机向从机写数据。见图 *23_7*。

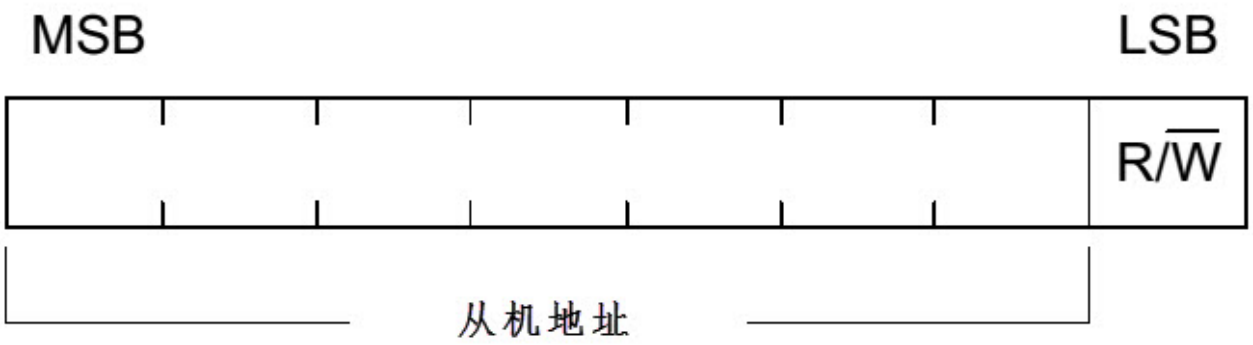


图 23-7 设备地址 (7 位) 及数据传输方向

读数据方向时，主机会释放对 SDA 信号线的控制，由从机控制 SDA 信号线，主机接收信号，写数据方向时，SDA 由主机控制，从机接收信号。

22.1.2.5 响应

I2C 的数据和地址传输都带响应。响应包括“应答 (ACK)”和“非应答 (NACK)”两种信号。作为数据接收端时，当设备 (无论主从机) 接收到 I2C 传输的一个字节数据或地址后，若希望对方继续发送数据，则需要向对方发送“应答 (ACK)”信号，发送方会继续发送下一个数据；若接收端希望结束数据传输，则向对方发送“非应答 (NACK)”信号，发送方接收到该信号后会产生一个停止信号，结束信号传输。见图 *23_8*。

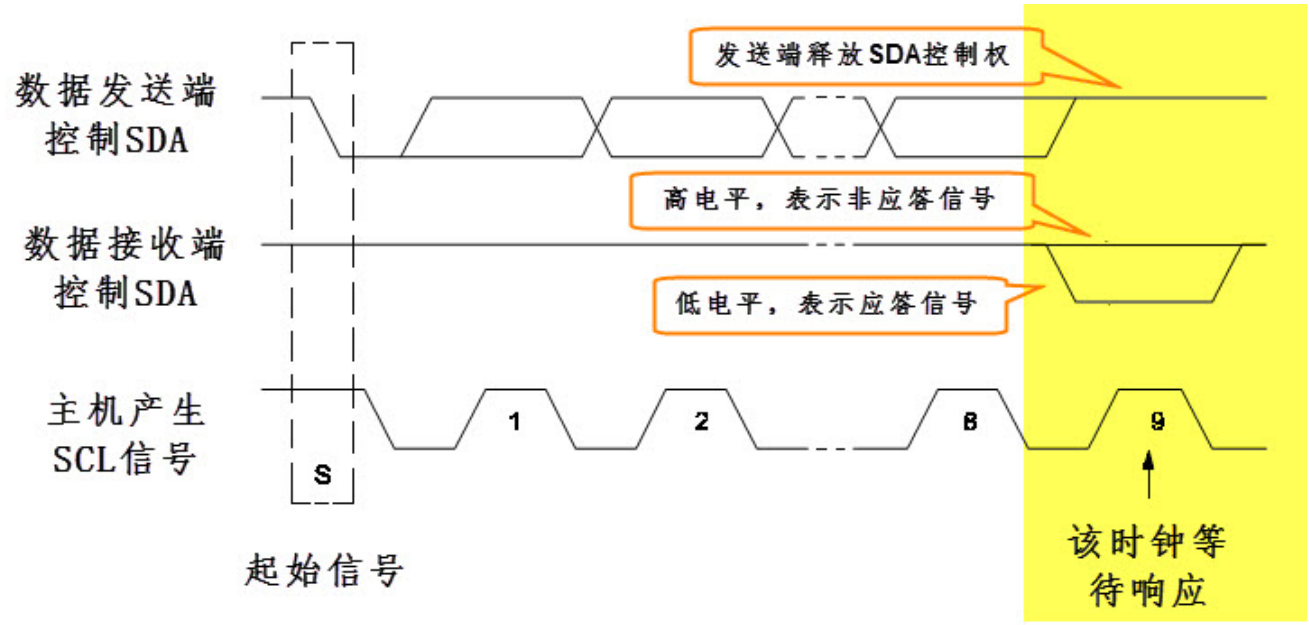


图 23-8 响应与非响应信号

传输时主机产生时钟，在第 9 个时钟时，数据发送端会释放 SDA 的控制权，由数据接收端控制 SDA，若 SDA 为高电平，表示非应答信号 (NACK)，低电平表示应答信号 (ACK)。

22.2 STM32 的 I2C 特性及架构

如果我们直接控制 STM32 的两个 GPIO 引脚，分别用作 SCL 及 SDA，按照上述信号的时序要求，直接像控制 LED 灯那样控制引脚的输出 (若是接收数据时则读取 SDA 电平)，就可以实现 I2C 通讯。同样，假如我们按照 USART 的要求去控制引脚，也能实现 USART 通讯。所以只要遵守协议，就是标准的通讯，不管您如何实现它，不管是 ST 生产的控制器还是 ATMEL 生产的存储器，都能按通讯标准交互。

由于直接控制 GPIO 引脚电平产生通讯时序时，需要由 CPU 控制每个时刻的引脚状态，所以称之为“软件模拟协议”方式。

相对地，还有“硬件协议”方式，STM32 的 I2C 片上外设专门负责实现 I2C 通讯协议，只要配置好该外设，它就会自动根据协议要求产生通讯信号，收发数据并缓存起来，CPU 只要检测该外设的状态和访问数据寄存器，就能完成数据收发。这种由硬件外设处理 I2C 协议的方式减轻了

CPU 的工作，且使软件设计更加简单。

22.2.1 STM32 的 I²C 外设简介

STM32 的 I2C 外设可用作通讯的主机及从机，支持 100Kbit/s 和 400Kbit/s 的速率，支持 7 位、10 位设备地址，支持 DMA 数据传输，并具有数据校验功能。它的 I2C 外设还支持 SMBus2.0 协议，SMBus 协议与 I2C 类似，主要应用于笔记本电脑的电池管理中，本教程不展开，感兴趣的读者可参考《SMBus20》文档了解。

22.2.2 STM32 的 I²C 架构剖析

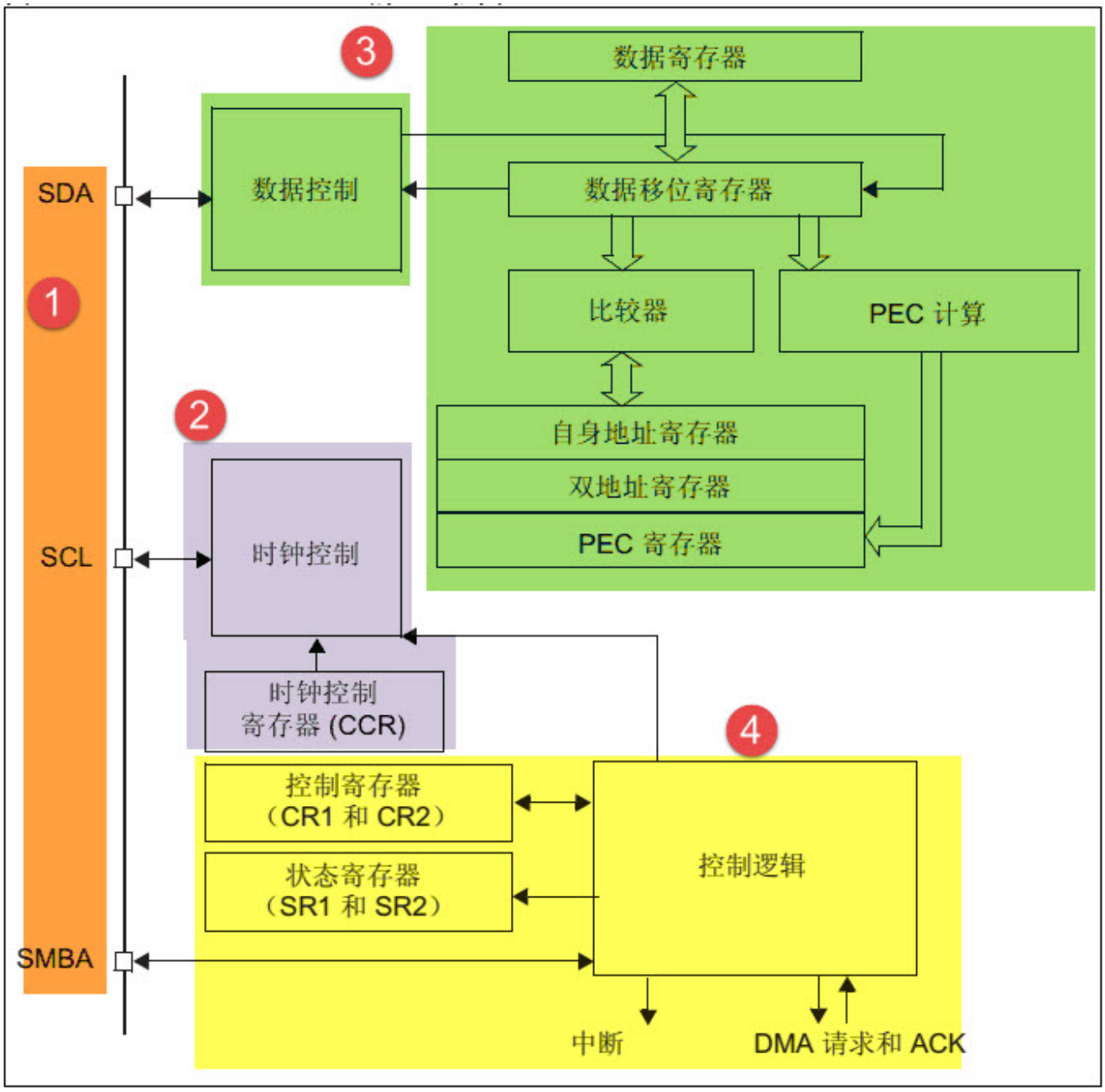


图 23-9 I2C 架构图

22.2.2.1 通讯引脚

I²C 的所有硬件架构都是根据图中左侧 SCL 线和 SDA 线展开的 (其中的 SMBA 线用于 SMBUS 的警告信号，I2C 通讯没有使用)。STM32 芯片有多个 I2C 外设，它们的 I2C 通讯信号引出到不同的 GPIO 引脚上，使用时必须配置到这些指定的引脚，见表 23-1。关于 GPIO 引脚的复用功能，以规格书为准。

表 23-1 STM32F10x 的 I2C 引脚

引脚	I2C1	I2C2
SCL	PB6 / PB8(重映射)	PB10
SDA	PB7 / PB9(重映射)	PB11

22.2.2.2 时钟控制逻辑

SCL 线的时钟信号，由 I²C 接口根据时钟控制寄存器 *(CCR)* 控制，控制的参数主要为时钟频率。配置 I2C 的 CCR 寄存器可修改通讯速率相关的参数：

- 可选择 I2C 通讯的 “标准/快速” 模式，这两个模式分别 I2C 对应 100/400Kbit/s 的通讯速率。
- 在快速模式下可选择 SCL 时钟的占空比，可选 Tlow/Thigh=2 或 Tlow/Thigh=16/9 模式，我们知道 I2C 协议在 SCL 高电平时对 SDA 信号采样，SCL 低电平时 SDA 准备下一个数据，修改 SCL 的高低电平比会影响数据采样，但其实这两个模式的比例差别并不大，若不是要求非常严格，这里随便选就可以了。
- CCR 寄存器中还有一个 12 位的配置因子 CCR，它与 I2C 外设的输入时钟源共同作用，产生 SCL 时钟，STM32 的 I2C 外设都挂载在 APB1 总线上，使用 APB1 的时钟源 PCLK1，SCL 信号线的输出时钟公式如下：

标准模式：

T_{high}=CCR*T_{PCKL1} T_{low} = CCR*T_{PCLK1}

快速模式中 T_{low}/T_{high}=2 时：

T_{high} = CCR*T_{PCKL1} T_{low} = 2*CCR*T_{PCKL1}

快速模式中 T_{low}/T_{high}=16/9 时：

T_{high} = 9*CCR*T_{PCKL1} T_{low} = 16*CCR*T_{PCKL1}

例如，我们的 PCLK1=36MHz，想要配置 400Kbit/s 的速率，计算方式如下：

PCLK 时钟周期：TPCLK1 = 1/36000000

目标 SCL 时钟周期：TSCL = 1/400000

SCL 时钟周期内的高电平时间：THIGH = TSCL/3

SCL 时钟周期内的低电平时间：TLOW = 2*TSCL/3

计算 CCR 的值：CCR = THIGH/TPCLK1 = 30

计算结果得出 CCR 为 30，向该寄存器位写入此值则可以控制 IIC 的通讯速率为 400KHz，其实即使配置出来的 SCL 时钟不完全等于标准的 400KHz，IIC 通讯的正确性也不会受到影响，因为所有数据通讯都是由 SCL 协调的，只要它的时钟频率不远高于标准即可。

22.2.2.3 数据控制逻辑

I2C 的 SDA 信号主要连接到数据移位寄存器上，数据移位寄存器的数据来源及目标是数据寄存器 (DR)、地址寄存器 (OAR)、PEC 寄存器以及 SDA 数据线。当向外发送数据的时候，数据移位寄存器以“数据寄存器”为数据源，把数据一位一位地通过 SDA 信号线发送出去；当从外部接收数据的时候，数据移位寄存器把 SDA 信号线采样到的数据一位一位地存储到“数据寄存器”中。若使能了数据校验，接收到的数据会经过 PCE 计算器运算，运算结果存储在“PEC 寄存器”中。当 STM32 的 I2C 工作在从机模式的时候，接收到设备地址信号时，数据移位寄存器会把接收到的地址与 STM32 的自身的“I2C 地址寄存器”的值作比较，以便响应主机的寻址。STM32 的自身 I2C 地址可通过修改“自身地址寄存器”修改，支持同时使用两个 I2C 设备地址，两个地址分别存储在 OAR1 和 OAR2 中。

22.2.2.4 整体控制逻辑

整体控制逻辑负责协调整个 I2C 外设，控制逻辑的工作模式根据我们配置的“控制寄存器(CR1/CR2)”的参数而改变。在外设工作时，控制逻辑会根据外设的工作状态修改“状态寄存器 (SR1 和 SR2)”，我们只要读取这些寄存器相关的寄存器位，就可以了解 I2C 的工作状态。除此之外，控制逻辑还根据要求，负责控制产生 I2C 中断信号、DMA 请求及各种 I2C 的通讯信号(起始、停止、响应信号等)。

22.2.3 通讯过程

使用 I2C 外设通讯时，在通讯的不同阶段它会对“状态寄存器 (SR1 及 SR2)”的不同数据位写入参数，我们通过读取这些寄存器标志来了解通讯状态。

22.2.3.1 主发送器

见图 *23_10* 。图中的是“主发送器”流程，即作为 I2C 通讯的主机端时，向外发送数据时的过程。

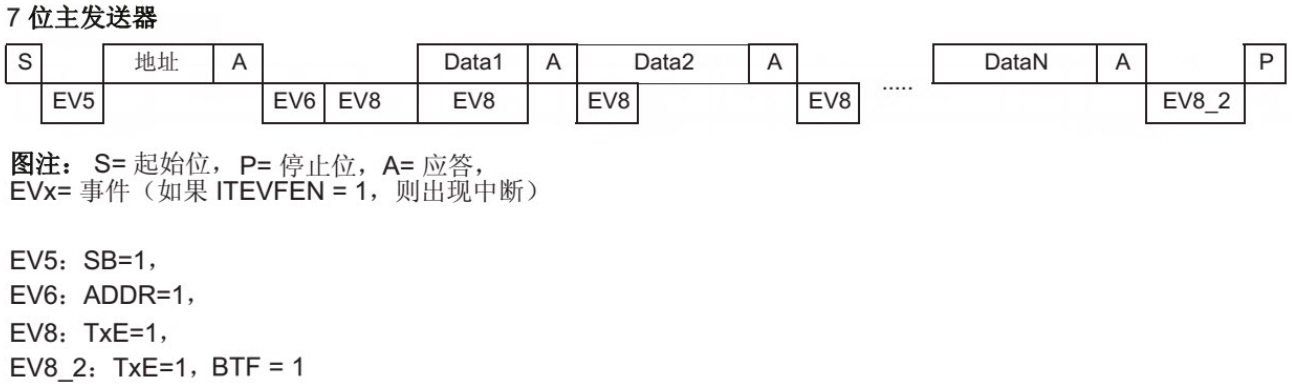


图 23-10 主发送器通讯过程

主发送器发送流程及事件说明如下：

(1) 控制产生起始信号 (S)，当发生起始信号后，它产生事件“EV5”，并会对 SR1 寄存器的“SB”位置 1，表示起始信号已经发送；

(2) 紧接着发送设备地址并等待应答信号，若有从机应答，则产生事件“EV6”及“EV8”，这时 SR1 寄存器的“ADDR”位及“TXE”位被置 1，ADDR 为 1 表示地址已经发送，TXE 为 1 表示数据寄存器为空；

(3) 以上步骤正常执行并对 ADDR 位清零后，我们往 I2C 的“数据寄存器 DR”写入要发送的数据，这时 TXE 位会被重置 0，表示数据寄存器非空，I2C 外设通过 SDA 信号线一位位把数据发送出去后，又会产生“EV8”事件，即 TXE 位被置 1，重复这个过程，就可以发送多个字节数据了；

(4) 当我们发送数据完成后，控制 I2C 设备产生一个停止信号 (P)，这个时候会产生 EV8_2 事件，SR1 的 TXE 位及 BTF 位都被置 1，表示通讯结束。

假如我们使能了 I2C 中断，以上所有事件产生时，都会产生 I2C 中断信号，进入同一个中断服务函数，到 I2C 中断服务程序后，再通过检查寄存器位来判断是哪一个事件。

22.2.3.2 主接收器

再来分析主接收器过程，即作为 I2C 通讯的主机端时，从外部接收数据的过程，见图 *23_11*。

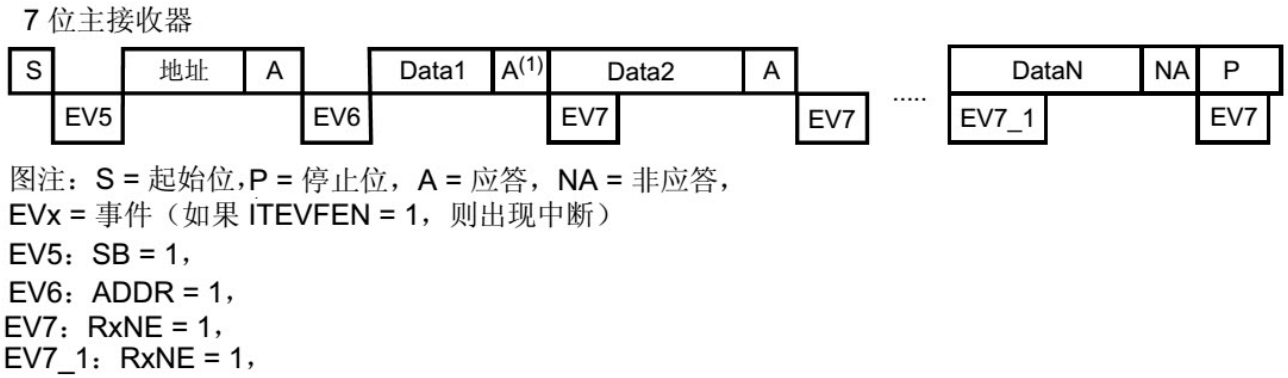


图 23-11 主接收器过程

主接收器接收流程及事件说明如下：

(1) 同主发送流程，起始信号 (S) 是由主机端产生的，控制发生起始信号后，它产生事件“EV5”，并会对 SR1 寄存器的“SB”位置 1，表示起始信号已经发送；

(2) 紧接着发送设备地址并等待应答信号，若有从机应答，则产生事件“EV6”这时 SR1 寄存器的“ADDR”位被置 1，表示地址已经发送。

(3) 从机端接收到地址后，开始向主机端发送数据。当主机接收到这些数据后，会产生“EV7”事件，SR1 寄存器的 RXNE 被置 1，表示接收数据寄存器非空，我们读取该寄存器后，可对数据寄存器清空，以便接收下一次数据。此时我们可以控制 I2C 发送应答信号 (ACK) 或非应答信号 (NACK)，若应答，则重复以上步骤接收数据，若非应答，则停止传输；

(4) 发送非应答信号后，产生停止信号 (P)，结束传输。

在发送和接收过程中，有的事件不只是标志了我们上面提到的状态位，还可能同时标志主机状态之类的状态位，而且读了之后还需要清除标志位，比较复杂。我们可使用 STM32HAL 库函数来直接检测这些事件的复合标志，降低编程难度。

22.3 I2C 初始化结构体详解

跟其它外设一样，STM32 HAL 库提供了 I2C 初始化结构体及初始化函数来配置 I2C 外设。初始化结构体及函数定义在库文件“stm32f4xx_hal_i2c.h”及“stm32f4xx_hal_i2c.c”中，编程时我们可以结合这两个文件内的注释使用或参考库帮助文档。了解初始化结构体后我们就能对 I2C 外设运用自如了，见*代码清单 23_1*。

列表 1: 代码清单 23-1 I2C 初始化结构体

```
typedef struct {
    uint32_t ClockSpeed; /*!< 设置 SCL 时钟频率，此值要低于 40 0000*/
    uint32_t DutyCycle;  /* 指定时钟占空比，可选 low/high = 2:1 及 16:9 模式 */
    uint32_t OwnAddress1; /* 指定自身的 I2C 设备地址 1，可以是 7-bit 或者 10-
↪bit*/
    uint32_t AddressingMode; /* 指定地址的长度模式，可以是 7bit 模式或者 10bit 模
式 */

    uint32_t DualAddressMode; /* 设置双地址模式 */
    uint32_t OwnAddress2;   /* 指定自身的 I2C 设备地址 2，只能是 7-bit */
    uint32_t GeneralCallMode; /* 指定广播呼叫模式 */
    uint32_t NoStretchMode; /* 指定禁止时钟延长模式 */
} I2C_InitTypeDef;
```

这些结构体成员说明如下，其中括号内的文字是对应参数在 STM32 HAL 库中定义的宏：

(1) *ClockSpeed*

本成员设置的是 I2C 的传输速率，在调用初始化函数时，函数会根据我们输入的数值写入到 I2C 的时钟控制寄存器 CCR。这个数值的计算上一节已经说明。

(2) *DutyCycle*

本成员设置的是 I2C 的 SCL 线时钟的占空比。该配置有两个选择，分别为低电平时间比高电平时间为 2：1 (I2C_DUTYCYCLE_2) 和 16：9 (I2C_DUTYCYCLE_16_9)。其实这两个模式的比例差别并不大，一般要求都不会如此严格，这里随便选就可以了。

(3) *OwnAddress1*

本成员配置的是 STM32 的 I2C 设备自身地址 *1*，每个连接到 I2C 总线上的设备都要有一个自己的地址，作为主机也不例外。地址可设置为 7 位或 10 位 (受下面 (3) AddressingMode 成员决定)，只要该地址是 I2C 总线上唯一的即可。

STM32 的 I2C 外设可同时使用两个地址，即同时对两个地址作出响应，这个结构成员 OwnAddress1 配置的是默认的、OAR1 寄存器存储的地址，若需要设置第二个地址寄存器 OAR2，可使用 DualAddressMode 成员使能，然后设置 OwnAddress2 成员即可，OAR2 不支持 10 位地址。

(4) *AddressingMode*

本成员选择 I2C 的寻址模式是 7 位还是 10 位地址。这需要根据实际连接到 I2C 总线上设备的地址进行选择，这个成员的配置也影响到 OwnAddress1 成员，只有这里设置成 10 位模式时，OwnAddress1 才支持 10 位地址。

(5) *DualAddressMode*

本成员配置的是 STM32 的 I2C 设备自己的地址，每个连接到 I2C 总线上的设备都要有一个自己的地址，作为主机也不例外。地址可设置为 7 位或 10 位 (受下面 I2C_dual_addressing_mode 成员决定)，只要该地址是 I2C 总线上唯一的即可。

STM32 的 I2C 外设可同时使用两个地址，即同时对两个地址作出响应，这个结构成员 I2C_OwnAddress1 配置的是默认的、OAR1 寄存器存储的地址，若需要设置第二个地址寄存器 OAR2，可使用 I2C_OwnAddress2Config 函数来配置，OAR2 不支持 10 位地址。

(6) *OwnAddress2*

本成员配置的是 STM32 的 I2C 设备自身地址 *2*，每个连接到 I2C 总线上的设备都要有一个自己的地址，作为主机也不例外。地址可设置为 7 位，只要该地址是 I2C 总线上唯一的即可。

(7) *GeneralCallMode*

本成员是关于 I^2C 从模式时的广播呼叫模式设置。

(8) *NoStretchMode*

本成员是关于 I^2C 禁止时钟延长模式设置，用于在从模式下禁止时钟延长。它在主模式下必须保持关闭。

配置完这些结构体成员值，调用库函数 HAL_I2C_Init 即可把结构体的配置写入到寄存器中。

22.4 I2C—读写 EEPROM 实验

EEPROM 是一种掉电后数据不丢失的存储器，常用来存储一些配置信息，以便系统重新上电的时候加载之。EEPOM 芯片最常用的通讯方式就是 I^2C 协议，本小节以 EEPROM 的读写实验为大家讲解 STM32 的 I^2C 使用方法。实验中 STM32 的 I2C 外设采用主模式，分别用作主发送器和主接收器，通过查询事件的方式来确保正常通讯。

22.4.1 硬件设计

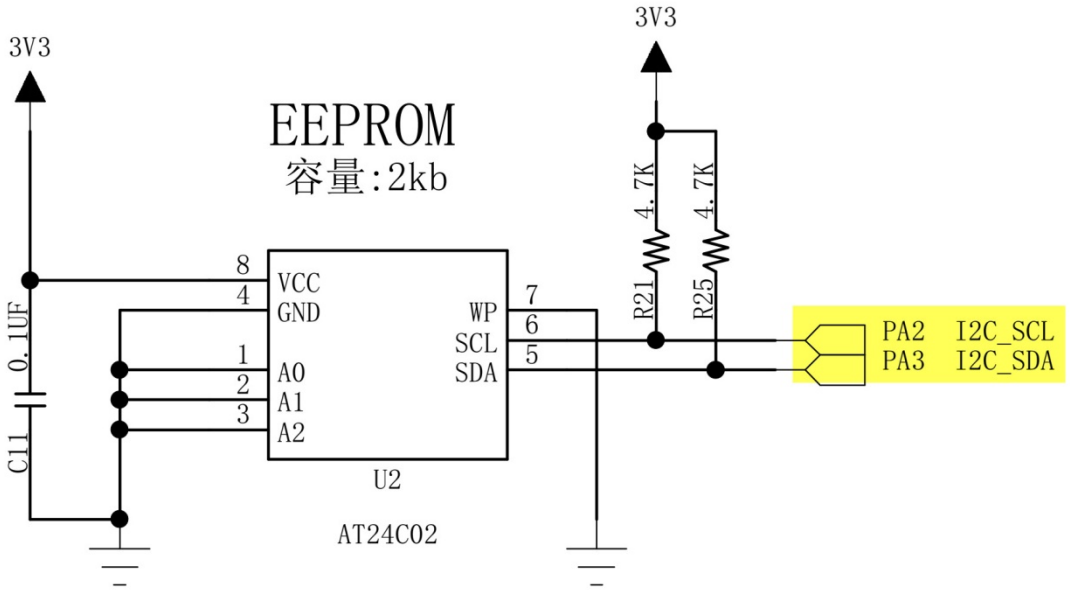


图 23-12a MINI 开发板的 EEPROM 硬件连接图

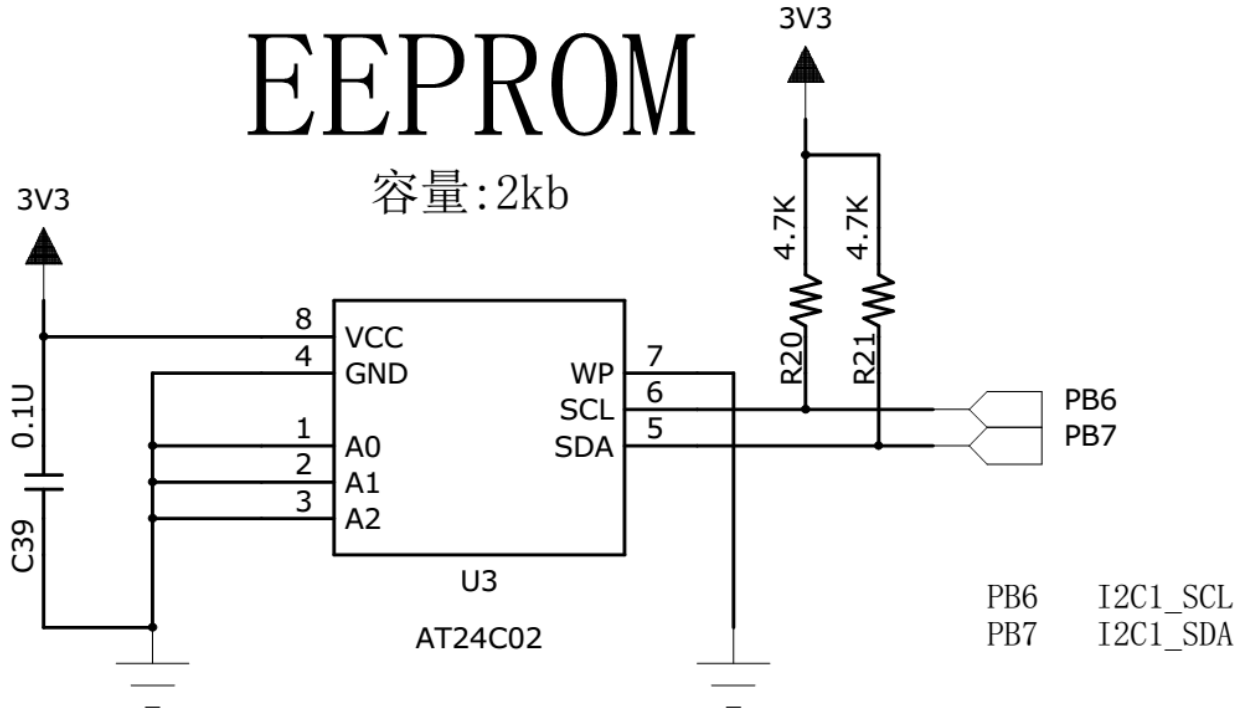


图 23-12b 指南者和霸道开发板的 EEPROM 硬件连接图

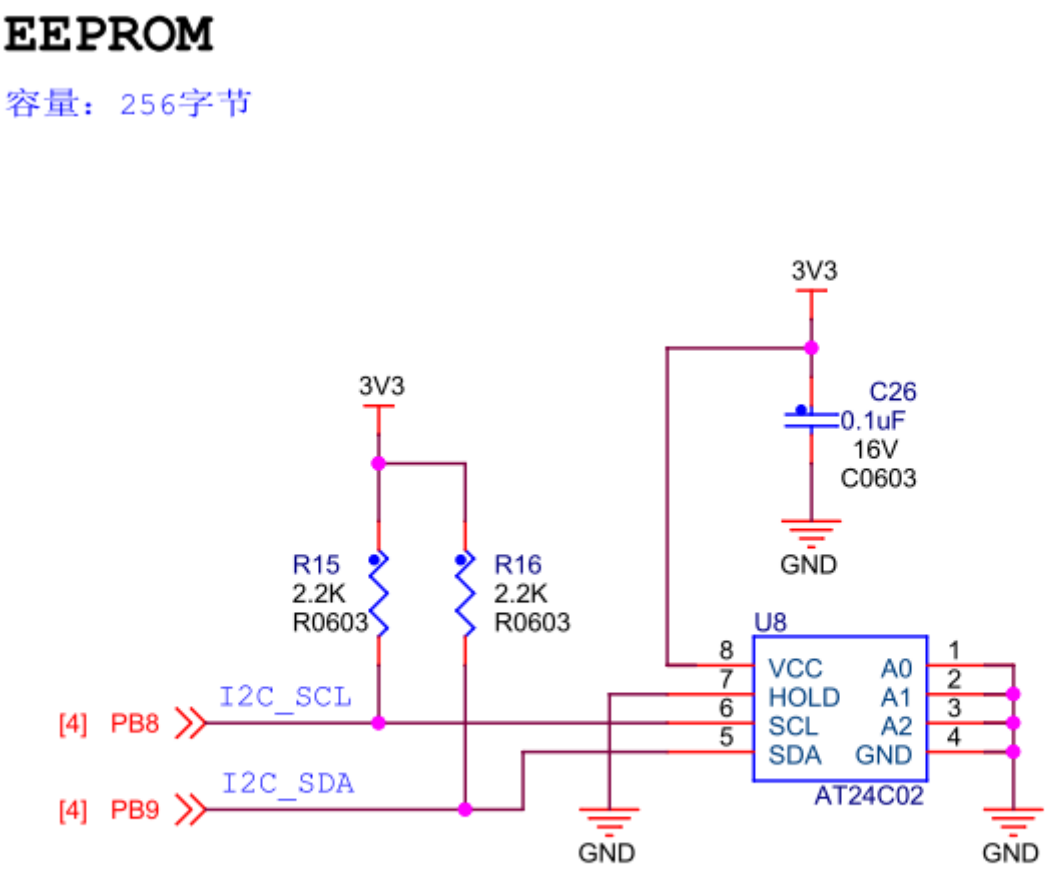


图 23-12c 拂晓电机开发板的 EEPROM 硬件连接图

本实验中开发板上的 EEPROM 芯片 (型号都是：AT24C02) 的 SCL 及 SDA 引脚连接到了 STM32 对应的 I2C 引脚中，结合上拉电阻，构成了 I2C 通讯总线，它们通过 I2C 总线交互。EEPROM 芯片的设备地址一共有 7 位，其中高 4 位固定为：1010 b，低 3 位则由 A0/A1/A2 信号线的电平决定，见图 *23_13*，图中的 R/W 是读写方向位，与地址无关。

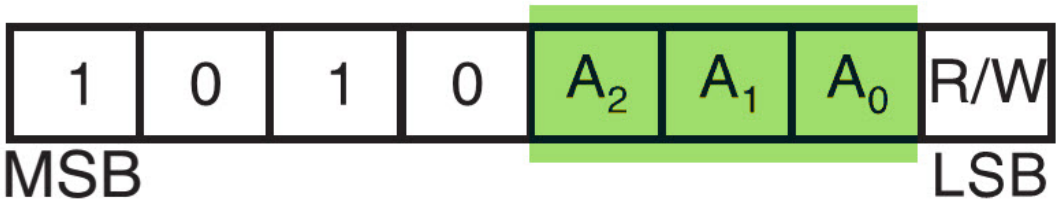


图 23-13 EEPROM 设备地址 (摘自《AT24C02》规格书)

按照我们此处的连接，A0/A1/A2 均为 0，所以 EEPROM 的 7 位设备地址是：101 0000b ，即 0xA0。由于 I2C 通讯时常常是地址跟读写方向连在一起构成一个 8 位数，且当 R/W 位为 0 时，表示写方向，所以加上 7 位地址，其值为“0xA0”，常称该值为 I2C 设备的“写地址”；当 R/W 位为 1 时，表示读方向，加上 7 位地址，其值为“0xA1”，常称该值为“读地址”。

EEPROM 芯片中还有一个 WP 引脚，具有写保护功能，当该引脚电平为高时，禁止写入数据，当引脚为低电平时，可写入数据，我们直接接地，不使用写保护功能。

关于 EEPROM 的更多信息，可参考其数据手册《AT24C02》来了解。若您使用的实验板 EEPROM 的型号、设备地址或控制引脚不一样，只需根据我们的工程修改即可，程序的控制原理相同。

22.4.2 软件设计

为了使工程更加有条理，我们把读写 EEPROM 相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_i2c_ee.c”及“bsp_i2c_ee.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

22.4.2.1 编程要点

(1) 配置通讯使用的目标引脚为开漏模式；

(2) 使能 I2C 外设的时钟；

(3) 配置 I2C 外设的模式、地址、速率等参数并使能 I2C 外设；

(4) 编写基本 I2C 按字节收发的函数；

(5) 编写读写 EEPROM 存储内容的函数；

(6) 编写测试程序，对读写数据进行校验。

22.4.2.2 代码分析

22.4.2.2.1 I2C 硬件相关宏定义

我们把 I2C 硬件相关的配置都以宏的形式定义到相应的头文件当中，MINI 开发板的定义在“bsp_i2c_gpio.h”中，见*代码清单 23_2a*，指南者和霸道、拂晓电机开发板的定义在“bsp_i2c_ee.h”文件中，见*代码清单 23_2b*、*代码清单 23_2c*。

使用 MINI 开发板的朋友需要注意的是，I2C 引脚 PA2、PA3 没有硬件自带的 I2C 复用功能，所以无法通过 HAL 库去进行 I2C 功能的配置，我们需要使用软件控制 IO 口的方式去模拟 I2C 协议进行通讯。

列表 2: 代码清单 23-2a MINI 开发板 I2C 硬件配置相关的宏（bsp_i2c_gpio.h 文件）

```
#define EEPROM_I2C_WR 0    /* 写控制 bit */
#define EEPROM_I2C_RD 1    /* 读控制 bit */


/* 定义 I2C 总线连接的 GPIO 端口,
用户只需要修改下面 4 行代码即可任意改变 SCL 和 SDA 的引脚
*/

#define I2Cx_SDA_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define I2Cx_SCL_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()

#define EEPROM_GPIO_PORT_I2C  GPIOA      /* GPIO 端口 */
#define EEPROM_I2C_SCL_PIN    GPIO_PIN_2       /* 连接到 SCL 时钟线的 GPIO */
#define EEPROM_I2C_SDA_PIN    GPIO_PIN_3       /* 连接到 SDA 数据线的 GPIO */
```

(下页继续)

(续上页)

```
/* 定义读写 SCL 和 SDA 的宏，已增加代码的可移植性和可阅读性 */
#if 0 /* 条件编译： 1 选择 GPIO 的库函数实现 IO 读写 */

#define EEPROM_I2C_SCL_1() digitalH(EEPROM_GPIO_PORT_I2C,EEPROM_I2C_SCL_
↪PIN)   /* SCL = 1 */
#define EEPROM_I2C_SCL_0()  digitalL(EEPROM_GPIO_PORT_I2C,EEPROM_I2C_SCL_
↪PIN)    /* SCL = 0 */

#define EEPROM_I2C_SDA_1() digitalH(EEPROM_GPIO_PORT_I2C,EEPROM_I2C_SDA_
↪PIN)   /* SDA = 1 */
#define EEPROM_I2C_SDA_0()  digitalL(EEPROM_GPIO_PORT_I2C,EEPROM_I2C_SDA_
↪PIN)    /* SDA = 0 */

#define EEPROM_I2C_SDA_READ()  GPIO_ReadInputDataBit(EEPROM_GPIO_PORT_I2C,␣
↪EEPROM_I2C_SDA_PIN) /* 读 SDA 口线状态 */

#else /* 这个分支选择直接寄存器操作实现 IO 读写 */

/*   注意：如下写法，在 IAR 最高级别优化时，会被编译器错误优化 */
#define EEPROM_I2C_SCL_1()  EEPROM_GPIO_PORT_I2C->BSRR =EEPROM_I2C_SCL_PIN␣
↪      /* SCL = 1 */
#define EEPROM_I2C_SCL_0()  EEPROM_GPIO_PORT_I2C->BRR =EEPROM_I2C_SCL_PIN ␣
↪      /* SCL = 0 */

#define EEPROM_I2C_SDA_1()  EEPROM_GPIO_PORT_I2C->BSRR =EEPROM_I2C_SDA_PIN␣
↪      /* SDA = 1 */
#define EEPROM_I2C_SDA_0()  EEPROM_GPIO_PORT_I2C->BRR =EEPROM_I2C_SDA_PIN ␣
↪      /* SDA = 0 */

#define EEPROM_I2C_SDA_READ()  ((EEPROM_GPIO_PORT_I2C->IDR &EEPROM_I2C_SDA_
↪PIN) != 0)  /* 读 SDA 口线状态 */
```

(下页继续)

(续上页)

```
#endif

/* 直接操作寄存器的方法控制 IO */
#define digitalH(p,i)      {p->BSRR=i;}          //设置为高电平
#define digitalL(p,i)      {p->BSRR=(uint32_t)i << 16;}//输出低电平
```

以上代码的前 6 行，根据硬件连接，把与 EEPROM 通讯使用的 GPIO 引脚号都以宏封装起来，以便初始化 GPIO 模式的时候使用。

第 6 行后的代码，被条件编译宏 “#if…#else…#endif” 分成了两部分，当 “#if ” 中的条件成立，使用第一部分代码，否则使用第二部分代码。这两部分代码实现的功能完全一样，都是定义了一些宏，用于控制作为 SCL、SDA 线的 GPIO 输出高低电平，以及读取 SDA 线引脚的电平，这些功能在软件模拟 I2C 的时候用于产生时序，收发数据，实现 I2C 协议。这两部分的主要区别是第一部分使用库函数来控制引脚，代码比较易于理解，而第二部分则直接通过寄存器来控制引脚，它减少了函数调用的开销，在软件模拟 I2C 时，这些宏会被频繁调用，所以在本工程中使用第二部分的代码。

列表 3: 代码清单 23-2b 指南者和霸道开发板 I2C 硬件配置相关的宏（bsp_i2c_ee.h 文件）

```
/* 这个地址只要与 STM32 外挂的 I2C 器件地址不一样即可 */
#define I2C_OWN_ADDRESS7      0X0A

#define I2Cx                                I2C1
#define I2Cx_CLK_ENABLE()                   __HAL_RCC_I2C1_CLK_ENABLE()
#define I2Cx_SDA_GPIO_CLK_ENABLE()          __HAL_RCC_GPIOB_CLK_ENABLE()
#define I2Cx_SCL_GPIO_CLK_ENABLE()          __HAL_RCC_GPIOB_CLK_ENABLE()

#define I2Cx_FORCE_RESET()                  __HAL_RCC_I2C1_FORCE_RESET()
#define I2Cx_RELEASE_RESET()                __HAL_RCC_I2C1_RELEASE_RESET()

/* Definition for I2Cx Pins */
```

(下页继续)

(续上页)

```
#define I2Cx_SCL_PIN                    GPIO_PIN_6
#define I2Cx_SCL_GPIO_PORT              GPIOB
#define I2Cx_SCL_AF                     GPIO_AF4_I2C1
#define I2Cx_SDA_PIN                    GPIO_PIN_7
#define I2Cx_SDA_GPIO_PORT              GPIOB
#define I2Cx_SDA_AF                     GPIO_AF4_I2C1
```

列表 4: 代码清单 23-2c 拂晓电机开发板 I2C 硬件配置相关的宏（bsp_i2c_ee.h 文件）

```
/* 这个地址只要与 STM32 外挂的 I2C 器件地址不一样即可 */
#define I2C_OWN_ADDRESS7      0X0A

#define I2Cx                             I2C1
#define I2Cx_CLK_ENABLE()                __HAL_RCC_I2C1_CLK_ENABLE()
#define I2Cx_SDA_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOB_CLK_ENABLE()
#define I2Cx_SCL_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOB_CLK_ENABLE()

#define I2Cx_FORCE_RESET()               __HAL_RCC_I2C1_FORCE_RESET()
#define I2Cx_RELEASE_RESET()             __HAL_RCC_I2C1_RELEASE_RESET()

/* Definition for I2Cx Pins */
#define I2Cx_SCL_PIN                    GPIO_PIN_8
#define I2Cx_SCL_GPIO_PORT              GPIOB
#define I2Cx_SCL_AF                     GPIO_AF4_I2C1
#define I2Cx_SDA_PIN                    GPIO_PIN_9
#define I2Cx_SDA_GPIO_PORT              GPIOB
#define I2Cx_SDA_AF                     GPIO_AF4_I2C1
```

以上代码根据硬件连接，把与 EEPROM 通讯使用的 I2C 号、引脚号都以宏封装起来，并且定义了自身的 I2C 地址及通讯速率，以便配置模式的时候使用。

22.4.2.2.2 初始化 I2C 的 GPIO

利用上面的宏，编写 I2C GPIO 引脚的初始化函数，MINI 开发板的见*代码清单 23_3a*，指南者和霸道、拂晓电机开发板的见*代码清单 23_3b*。

列表 5: 代码清单 23-3a MINI 开发板的 I2C GPIO 初始化函数（bsp_i2c_gpio.c 文件）

```
void i2c_CfgGpio(void)
 {
     GPIO_InitTypeDef  GPIO_InitStruct;
     /* Enable GPIO TX/RX clock */
     I2Cx_SCL_GPIO_CLK_ENABLE();
     I2Cx_SDA_GPIO_CLK_ENABLE();

     /* I2C TX GPIO pin configuration  */
     GPIO_InitStruct.Pin       = EEPROM_I2C_SCL_PIN |
                                 EEPROM_I2C_SDA_PIN;
     GPIO_InitStruct.Mode      = GPIO_MODE_OUTPUT_OD;
     GPIO_InitStruct.Pull      = GPIO_NOPULL;
     GPIO_InitStruct.Speed     = GPIO_SPEED_FREQ_HIGH;

     HAL_GPIO_Init(EEPROM_GPIO_PORT_I2C, &GPIO_InitStruct);

     /* 给一个停止信号,
     复位 I2C 总线上的所有设备到待机模式 */
     i2c_Stop();
 }
```

列表 6: 代码清单 23-3b 指南者和霸道、拂晓电机开发板的 I2C GPIO 初始化函数

```
void HAL_I2C_MspInit(I2C_HandleTypeDef *hi2c)
{
    GPIO_InitTypeDef  GPIO_InitStruct;

    /* ## 使能使能外设和 GPIO 时钟 */
    /* 使能 GPIO 时钟 */
    I2Cx_SCL_GPIO_CLK_ENABLE();
    I2Cx_SDA_GPIO_CLK_ENABLE();
    /* 使能 I2C1 使能时钟 */
    I2Cx_CLK_ENABLE();

    /*## 配置外设引脚 */
    /* I2C TX GPIO 配置  */
    GPIO_InitStruct.Pin       = I2Cx_SCL_PIN;
    GPIO_InitStruct.Mode      = GPIO_MODE_AF_OD;
    GPIO_InitStruct.Pull      = GPIO_NOPULL;
    GPIO_InitStruct.Speed     = GPIO_SPEED_FAST;
    GPIO_InitStruct.Alternate = I2Cx_SCL_AF;

    HAL_GPIO_Init(I2Cx_SCL_GPIO_PORT, &GPIO_InitStruct);

    /* I2C RX GPIO 配置  */
    GPIO_InitStruct.Pin = I2Cx_SDA_PIN;
    GPIO_InitStruct.Alternate = I2Cx_SDA_AF;

    HAL_GPIO_Init(I2Cx_SDA_GPIO_PORT, &GPIO_InitStruct);

    /* 强制 I2C 外设时钟复位 */
    I2Cx_FORCE_RESET() ;
```

(下页继续)

(续上页)

```
    /* 释放 I2C 外设时钟复位 */
    I2Cx_RELEASE_RESET();
}
```

同为外设使用的 GPIO 引脚初始化，初始化的流程与“串口初始化函数”章节中的类似，主要区别是引脚的模式。函数执行流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置；

(2) 调用宏 I2Cx_CLK_ENABLE() 使能 I2C 外设时钟，调用宏定义 I2Cx_SCL_GPIO_CLK_ENABLE() 和 I2Cx_SDA_GPIO_CLK_ENABLE() 来使能 I2C 引脚使用的 GPIO 端口时钟。

(3) 向 GPIO 初始化结构体赋值，把引脚初始化成复用开漏模式，要注意 I2C 的引脚必须使用这种模式。

(4) 使用以上初始化结构体的配置，调用 HAL_GPIO_Init 函数向寄存器写入参数，完成 GPIO 的初始化。

22.4.2.2.3 配置 I2C 的模式

以上只是配置了 I2C 所使用的引脚，还不算对 I2C 模式的配置。

对于 MINI 开发板，我们要自己手动去模拟 I2C 通信协议，所以需要编写几个函数去实现 I2C 的启动、停止、发送单字节、接收单字节、应答、还有延时的功能。这个便是我们针对于 MINI 开发板的软件模拟 I2C 的配置过程。见*代码清单 23_4a*、*代码清单 23_4a_ 续 1* 、*代码清单 23_4a_ 续 2*。

而对于指南者和霸道开发板和拂晓电机开发板，与 EEPROM 芯片相连接的两个 I2C 引脚是有硬件 I2C 的，我们可以直接使用 HAL 库去配置 STM32 的硬件 I2C，非常地方便。见*代码清单 23_4b*。

软件模拟 I2C：起始停止信号及速率配置

列表 7: 代码清单 23_4a 起始停止信号及速率配置（bsp_i2c_gpio.c 文件）

```
/*
*********************************************
* 函 数 名: i2c_Delay
* 功能说明: I2C 总线位延迟, 最快 400KHz
* 形    参: 无
* 返 回 值: 无
*********************************************
*/
static void i2c_Delay(void)
{
    uint8_t i;

    /*
    下面的时间是通过逻辑分析仪测试得到的。
    工作条件: CPU 主频 72MHz , MDK 编译环境, 1 级优化
    循环次数为 10 时, SCL 频率 = 205KHz
    循环次数为 7 时, SCL 频率 = 347KHz, SCL 高电平时间 1.5us, SCL 低电平时间 2.87us
    循环次数为 5 时, SCL 频率 = 421KHz, SCL 高电平时间 1.25us, SCL 低电平时间 2.375us
    */
    for (i = 0; i < 10; i++);
}

/*
*********************************************
* 函 数 名: i2c_Start
* 功能说明: CPU 发起 I2C 总线启动信号
* 形    参: 无
* 返 回 值: 无
```

(下页继续)

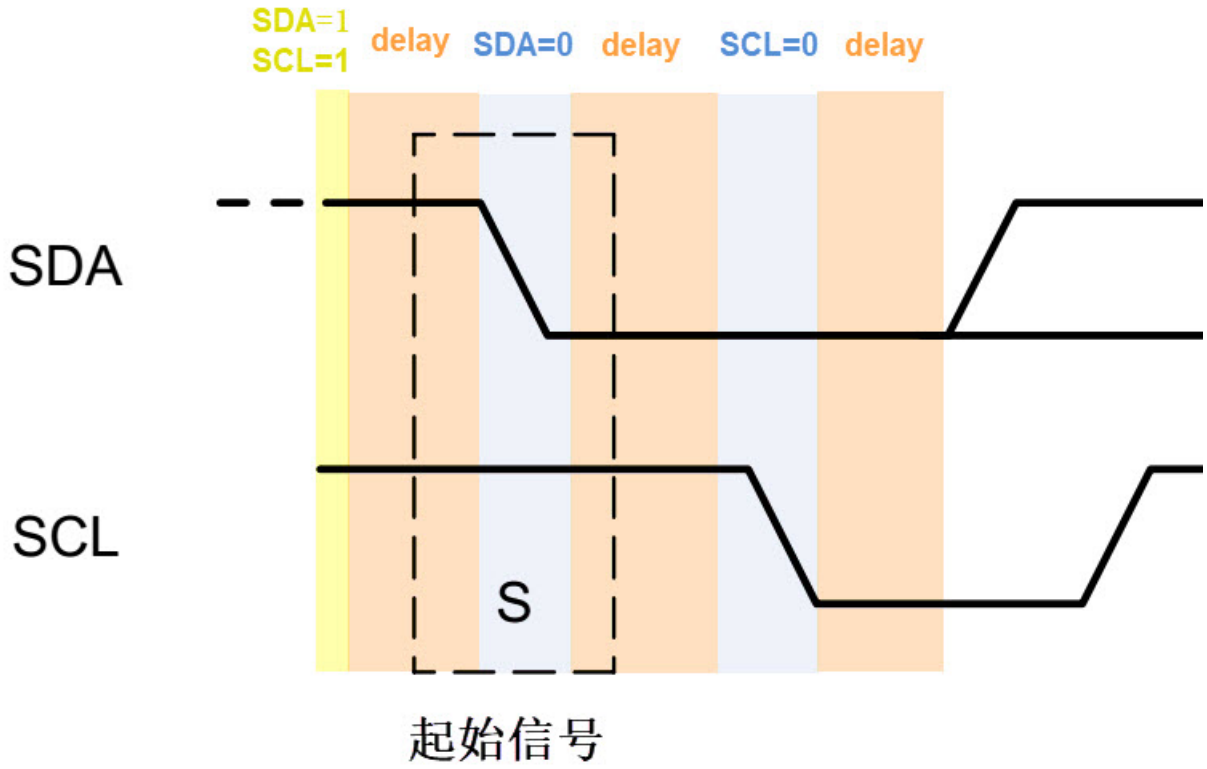
(续上页)

```
29  ************************************************
30  */
31  void i2c_Start(void)
32  {
33      /* 当 SCL 高电平时，SDA 出现一个下跳沿表示 I2C 总线启动信号 */
34      EEPROM_I2C_SDA_1();
35      EEPROM_I2C_SCL_1();
36      i2c_Delay();
37      EEPROM_I2C_SDA_0();
38      i2c_Delay();
39      EEPROM_I2C_SCL_0();
40      i2c_Delay();
41  }
42
43  /*
44  ************************************************
45  * 函 数 名: i2c_Start
46  * 功能说明: CPU 发起 I2C 总线停止信号
47  * 形    参: 无
48  * 返 回 值: 无
49  ************************************************
50  */
51  void i2c_Stop(void)
52  {
53      /* 当 SCL 高电平时，SDA 出现一个上跳沿表示 I2C 总线停止信号 */
54      EEPROM_I2C_SDA_0();
55      EEPROM_I2C_SCL_1();
56      i2c_Delay();
57      EEPROM_I2C_SDA_1();
58  }
```

上述代码首先定义了一个 i2c_Delay 函数，该函数主要用于 I2C 模拟输出信号电平的延时，如控

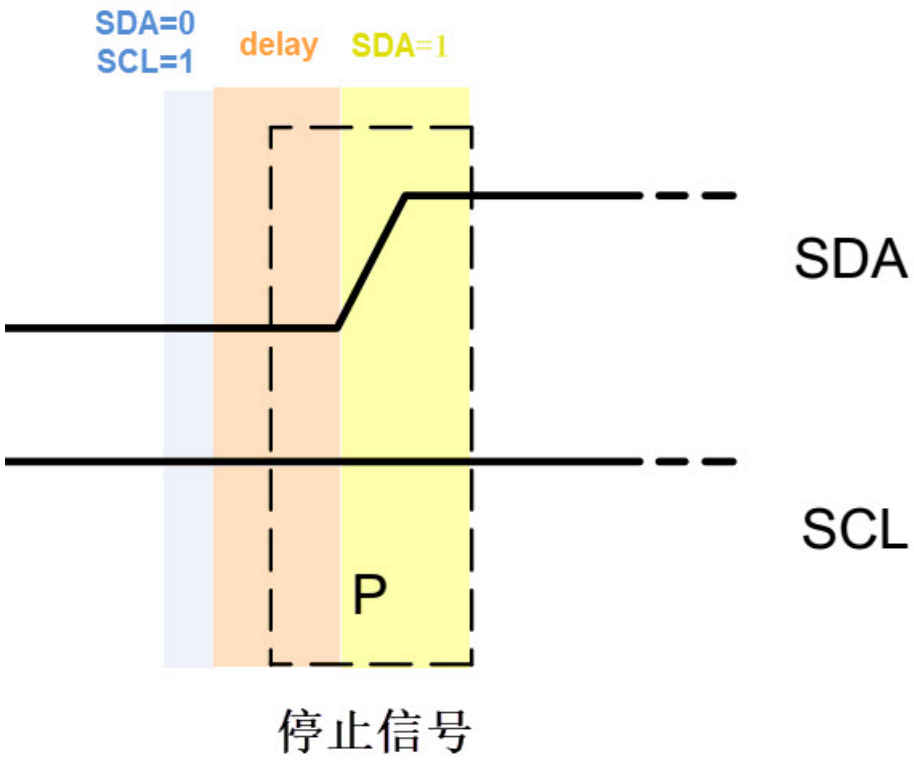
制 SDA 输出高电平后调用 i2c_Delay 延时，再控制 SDA 输出低电平，就达到了 SDA 维持高电平一段时间后切换的效果，而电平切换的延时会影响到 I2C 时序的通讯速率。在 i2c_Delay 函数内部，实际是直接使用了一个无实际操作、纯粹消耗时间的 for 循环，以此达到延时目的。该延时函数配置的循环次数是个经验值，在不同的芯片平台或主频不同的情况下，延时的时间会存在差异，我们使用逻辑分析仪测出了 i2c_Delay 函数在不同情况下对整个模拟 I2C 时序波特率的影响，具体的波特率见函数中的注释。

代码中的 i2c_Start 函数，用于控制 SCL、SDA 线产生 I2C 协议的起始信号，它利用“bsp_i2c_gpio.h”文件中的宏控制 SDA、SCL 线输出高低电平，而这输出高低电平的逻辑完全与图 起始信号分解图 中解说的 I2C 协议起始信号的时序一样，见图 起始信号分解图 对起始信号的分解，分解后的每个步骤实际就是 i2c_Start 函数内部对 GPIO 引脚的操作。



起始信号

类似地，代码中的 i2c_Stop 函数用于控制 SCL、SDA 线产生 I2C 协议的停止信号，停止信号的分解见图 停止信号分解图 ，i2c_Stop 函数就是按该分解步骤控制 GPIO 的电平逻辑，实现停止信号。



软件模拟 I2C：发送和接收一个字节

列表 8: 代码清单 23_4a(续 1) 收发数据（bsp_i2c_gpio.c 文件）

```
1  /*
2  *********************************************************
3  * 函 数 名: i2c_SendByte
4  * 功能说明: CPU 向 I2C 总线设备发送 8bit 数据
5  * 形    参: _ucByte : 等待发送的字节
6  * 返 回 值: 无
7  *********************************************************
8  */
9  void i2c_SendByte(uint8_t _ucByte)
10 {
```

(下页继续)

(续上页)

```
    uint8_t i;

    /* 先发送字节的高位 bit7 */
    for (i = 0; i < 8; i++) {
        if (_ucByte & 0x80) {
            EEPROM_I2C_SDA_1();
        } else {
            EEPROM_I2C_SDA_0();
        }
        i2c_Delay();
        EEPROM_I2C_SCL_1();
        i2c_Delay();
        EEPROM_I2C_SCL_0();
        if (i == 7) {
            EEPROM_I2C_SDA_1(); // 释放总线
        }
        _ucByte <<= 1;  /* 左移一个 bit */
        i2c_Delay();
    }
}

/*
*********************************************
* 函 数 名: i2c_ReadByte
* 功能说明: CPU 从 I2C 总线设备读取 8bit 数据
* 形    参: 无
* 返 回 值: 读到的数据
*********************************************
*/
uint8_t i2c_ReadByte(void)
{
    uint8_t i;
```

(下页继续)

(续上页)

```
43      uint8_t value;
44
45      /* 读到第 1 个 bit 为数据的 bit7 */
46      value = 0;
47      for (i = 0; i < 8; i++) {
48          value <<= 1;
49          EEPROM_I2C_SCL_1();
50          i2c_Delay();
51          if (EEPROM_I2C_SDA_READ()) {
52              value++;
53          }
54          EEPROM_I2C_SCL_0();
55          i2c_Delay();
56      }
57      return value;
58  }
```

I2C 通讯的主体是收发数据，实现时主要是根据 I2C 协议数据有效性的要求，读取或控制 SDA 信号线的电平，见上述的代码。

以发送函数 i2c_SendByte 为例，该函数以其输入参数作为要使用 I2C 协议输出的数据，该数据大小为一字节。函数的主体是一个 8 次的 for 循环，循环体执行一次将会对外发送一个数据位，循环结束时刚好发送完该字节数据。

循环体每位数据的发送逻辑见图数据有效性分解图，步骤分解如下：

(1) 程序对输入参数 _ucByte 和 0x80“与”运算，判断其最高位的逻辑值，为 1 时控制 SDA 输出高电平，为 0 则控制 SDA 输出低电平；

(2) 延时，以此保证 SDA 线输出的电平已稳定，再进行后续操作；

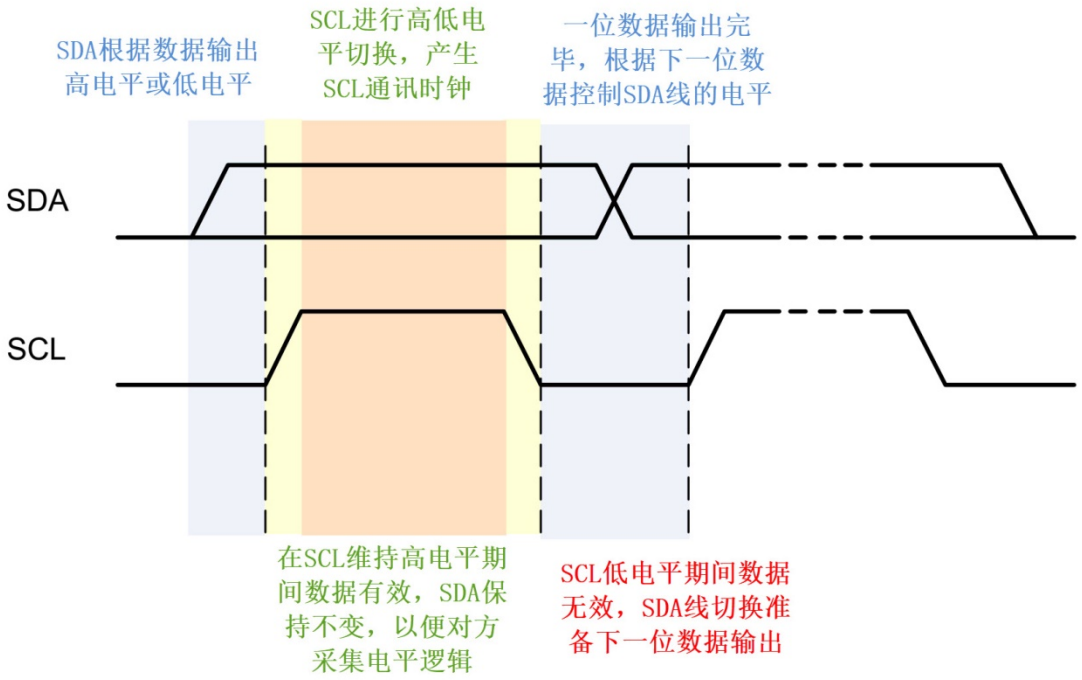
(3) 控制 SCL 线产生高低电平跳变，也就是产生 I2C 协议中 SCL 线的通讯时钟；

(4) 在 SCL 线高低电平之间有个延时，该延时期间 SCL 线维持高电平，根据 I2C 协议，此时数据有效，通讯的另一方会在此时读取 SDA 线的电平逻辑，高电平时接收到该位为数据 1，否则为 0；

(5) 一次循环体执行结束，_ucByte 左移一位以便下次循环发送下一位的数据；

(6) 如次循环 8 次，把 _ucByte 中的 8 位个数据位发送完毕，在最后一位发送完成后（此时循环计数器 i=7），控制 SDA 线输出 1（即高阻态），也就是说发送方释放 SDA 总线，等待接收方的应答信号；

(7) 本驱动程序中，检测应答信号由另一个独立函数 i2c_WaitAck 完成（后面再分析），此处释放 SDA 总线后直接退出循环和函数。



i2c_ReadByte 函数的控制过程刚好反过来，不过由于本程序 STM32 都是作为 I2C 协议的通讯主机，所以 SCL 线的时钟信号依然是由 STM32 来控制产生的，即图*数据有效性分解图* 中的 SDA 线将由另一端的发送方控制，STM32 控制 SCL 产生通讯时钟，并在数据有效时读取 SDA 线的信号获得数据。

i2c_ReadByte 函数也是以 for 循环为主体，循环体会被执行 8 次，执行完毕后将会接收到一个字节的数据，循环体接收数据的流程如下：

(1) 使用一个变量 value 暂存要接收的数据，每次循环开始前先对 value 的值左移 1 位，以给 value 变量的 bit0 腾出空间，bit0 将用于缓存最新接收到的数据位，一位一位地接收并移位，最后拼出完整的 8 位数据；

(2) 控制 SCL 线进行高低电平切换，输出 I2C 协议通讯用的时钟；

(3) 在 SCL 线高低电平切换之间，有个延时，该延时确保给予了足够的时间让数据发送方进行处理，即发送方在 SCL 时钟驱动下通过 SDA 信号线发出电平逻辑信号，而这个延时之后，作为数据接收端的 STM32 使用宏 EEPROM_I2C_SDA_READ 读取 SDA 信号线的电平，若信号线为 1，则 value++，即把它的 bit0 置 1，否则不操作（这样该位将保持为 0），这样就读取到了一位的数据；

(4) SCL 线切换成低电平后，加入延时，以便接收端根据需要切换 SDA 线输出数据；

(5) 循环结束后，value 变量中包含有 1 个字节数据，使用 return 把它作为函数返回值返回；

(6) 本驱动程序中，作为接收端回复发送端的应答和非应答信号由 i2c_Ack 及 i2c_NAck 函数完成（后面再分析），此处接收完 8 位数据后直接返回数据退出函数。

软件模拟 I2C：响应处理及返回

一次基本的 I2C 协议通讯过程包含起始信号、收或发 8 个数据位的数据、1 个数据位的响应以及结束信号信号组成，这些环节中剩下最后的响应处理及返回将由下面的代码完成。

列表 9: 代码清单 23_4a(续 2) 响应处理及返回（bsp_i2c_gpio.c 文件）

```
1  /*
2  *********************************************************
3  * 函 数 名: i2c_WaitAck
4  * 功能说明:
5  CPU 产生一个时钟，并读取器件的 ACK 应答信号
6  * 形    参: 无
7  * 返 回 值: 返回 0 表示正确应答，1 表示无器件响应
8  *********************************************************
9  */
10 uint8_t i2c_WaitAck(void)
11 {
12     uint8_t re;
13
```

(下页继续)

(续上页)

```
14      EEPROM_I2C_SDA_1(); /* CPU 释放 SDA 总线 */
15      i2c_Delay();
16      EEPROM_I2C_SCL_1(); /* CPU 驱动 SCL = 1,
17                             此时器件会返回 ACK 应答 */
18      i2c_Delay();
19      if (EEPROM_I2C_SDA_READ()) {  /* CPU 读取 SDA 口线状态 */
20          re = 1;
21      } else {
22          re = 0;
23      }
24      EEPROM_I2C_SCL_0();
25      i2c_Delay();
26      return re;
27  }
28
29  /*
30  *********************************************
31  * 函 数 名: i2c_Ack
32  * 功能说明: CPU 产生一个 ACK 信号
33  * 形    参: 无
34  * 返 回 值: 无
35  *********************************************
36  */
37  void i2c_Ack(void)
38  {
39      EEPROM_I2C_SDA_0(); /* CPU 驱动 SDA = 0 */
40      i2c_Delay();
41      EEPROM_I2C_SCL_1(); /* CPU 产生 1 个时钟 */
42      i2c_Delay();
43      EEPROM_I2C_SCL_0();
44      i2c_Delay();
45      EEPROM_I2C_SDA_1(); /* CPU 释放 SDA 总线 */
```

(下页继续)

(续上页)

```
46  }
47
48  /*
49  *********************************************
50  * 函 数 名: i2c_NAck
51  * 功能说明: CPU 产生 1 个 NACK 信号
52  * 形    参: 无
53  * 返 回 值: 无
54  *********************************************
55  */
56  void i2c_NAck(void)
57  {
58      EEPROM_I2C_SDA_1(); /* CPU 驱动 SDA = 1 */
59      i2c_Delay();
60      EEPROM_I2C_SCL_1(); /* CPU 产生 1 个时钟 */
61      i2c_Delay();
62      EEPROM_I2C_SCL_0();
63      i2c_Delay();
64  }
```

阅读下面的解释时请配合前面的图 *23_8* 响应与非响应信号来理解，该图描述了 I2C 协议的响应信号时序。

首先是 i2c_WaitAck 函数，该函数用于 STM32 作为发送方时，等待及处理接收方传来的响应或非响应信号，即一般调用前面的 i2c_SendByte 函数后，再调用本函数检测响应。函数的实现如下：

(1) STM32 控制 SDA 信号线输出高阻态，释放它对 SDA 的控制权，由接收方控制；

(2) 控制 SCL 信号线切换高低电平，产生一个时钟信号，根据 I2C 协议，此时接收方若把 SDA 设置为低电平，就表示返回一个“应答”信号，若 SDA 保持为高电平，则表示返回一个“非应答”信号；

(3) 在 SCL 切换高低电平之间，有个延时确保给予了足够的时间让接收方返回应答信号，延时后使用宏 EEPROM_I2C_SDA_READ 读取 SDA 线的电平，根据电平值赋予 re 变量的值；

(4) 函数的最后返回 re 的值，接收到响应时返回 0，未接收到响应时返回 1。

当 STM32 作为数据接收端，调用 i2c_ReadByte 函数后，需要给发送端返回应答或非应答信号，此时可使用 i2c_Ack 及 i2c_Nack 函数处理，该处理与 i2c_WaitAck 函数相反，此时 SDA 线也由 STM32 控制，说明如下：

(1) 根据要返回“应答”还是“非应答”信号，先准备好 SDA 线的电平，i2c_Ack 函数中把 SDA 线设置为低电平，表示“应答”信号，i2c_Nack 函数中把 SDA 线设置为高电平，表示“非应答”信号；

(2) 控制 SCL 线进行高低电平切换，产生一个时钟信号，在 SCL 线的高低电平之间加入一个延时，确保有足够的时间让通讯的另一方接收到 SDA 信号线的电平；

(3) 在 i2c_Ack 函数的末尾，响应信号发送结束后，重新把 SDA 线设置为高电平以释放总线的控制权，方便后续的通讯。

以上就是软件模拟 I2C 协议的各个环节，可以看到，各个环节都是配合延时，使用内核直接控制 SDA、SCL 信号线 GPIO 的电平逻辑，产生符合 I2C 通讯协议的时序。若使用 STM32 的硬件 I2C，初始化 I2C 外设后，调用库函数设置 I2C 外设的寄存器位即可产生以上各环节的起始、停止、应答及非应答信号，收发数据时也可以直接通过读写 I2C 的数据寄存器完成，相对而言减轻了内核的负担。

到这里，我们就完成了在 MINI 开发板上实现的软件模拟 I2C。

下面是对指南者和霸道开发板、拂晓电机开发板使用 HAL 库对 I2C 模式的配置。

列表 10: 代码清单 23-4b 指南者和霸道开发板、拂晓电机开发板：配置 I2C 模式

```
/**
* @brief  I2C 工作模式配置
* @param  无
* @retval 无
*/
static void I2C_Mode_Config(void)
{
```

(下页继续)

(续上页)

```
    I2C_Handle.Instance             = I2Cx;

    I2C_Handle.Init.AddressingMode  = I2C_ADDRESSINGMODE_7BIT;
    I2C_Handle.Init.ClockSpeed      = 400000;
    I2C_Handle.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
    I2C_Handle.Init.DutyCycle       = I2C_DUTYCYCLE_2;
    I2C_Handle.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
    I2C_Handle.Init.NoStretchMode   = I2C_NOSTRETCH_DISABLE;
    I2C_Handle.Init.OwnAddress1     = I2C_OWN_ADDRESS7 ;
    I2C_Handle.Init.OwnAddress2     = 0;
    /* Init the I2C */
    HAL_I2C_Init(&I2C_Handle);

    HAL_I2CEx_AnalogFilter_Config(&I2C_Handle, I2C_ANALOGFILTER_ENABLE);
}

/**
* @brief  I2C 外设 (EEPROM) 初始化
* @param  无
* @retval 无
*/
void I2C_EE_Init(void)
{
    I2C_Mode_Config();
}
```

熟悉 STM32 I2C 结构的话，这段初始化程序就十分好理解了，指定连接 EEPROM 的 I2C 为 EEPROM_I2C 这里是 I2C4，时序配置为上面用工具计算出来的值，自身地址为 0，地址设置为 7bit 模式，关闭双地址模式，自身地址 2 也为 0，禁止通用广播模式，禁止时钟延长模式。最后调用库函数 HAL_I2C_Init 把这些配置写入寄存器。

为方便调用，我们把 I2C 的 GPIO 及模式配置都用 I2C_EE_Init 函数封装起来。

22.4.2.2.4 向 EEPROM 写入一个字节的数据

初始化好 I2C 外设后，就可以使用 I2C 通讯了，我们看看如何向 EEPROM 写入一个字节的数据，对于 MINI 开发板，见代码清单 *23_5a*，对于指南者和霸道开发板和拂晓电机开发板，见代码清单 *23_5b*。

检测 EEPROM 的响应及单字节写入

列表 11: 代码清单 23-5a 检测 EEPROM 的响应

```
1  /**bsp_i2c_ee.h 文件 **/
2  /* 24xx02 的设备地址 */
3  #define EEPROM_DEV_ADDR      0xA0
4
5  /**bsp_i2c_gpio.h 文件 **/
6  #define EEPROM_I2C_WR 0    /* 写控制 bit */
7  #define EEPROM_I2C_RD 1    /* 读控制 bit */
8
9  /**bsp_i2c_ee.c 文件 **/
10 /*
11 *********************************************************
12 * 函 数 名: i2c_CheckDevice
13 * 功能说明: 检测 I2C 总线设备, CPU 向发送设备地址,
14 *             然后读取设备应答来判断该设备是否存在
15 * 形    参: _Address: 设备的 I2C 总线地址
16 * 返 回 值: 返回值 0 表示正确, 返回 1 表示未探测到
17 *********************************************************
18 */
19 uint8_t ee_CheckDevice(uint8_t _Address)
20 {
21     uint8_t ucAck;
22
23     i2c_Start();     /* 发送启动信号 */
```

(下页继续)

(续上页)

```
24
25      /* 发送设备地址 + 读写控制 bit (0 = w,  1 = r) bit7
26                                                  先传 */
27      i2c_SendByte(_Address | EEPROM_I2C_WR);
28      ucAck = i2c_WaitAck();  /* 检测设备的 ACK 应答 */
29
30      i2c_Stop();      /* 发送停止信号 */
31
32      return ucAck;
33  }
```

以上代码涉及到几个宏，首先是 EEPROM_DEV_ADDR（宏值为 0XA0），这是 EERPOM 的设备地址，该地址与 EEPROM 的硬件接法有关，可参考前面图图 *23_13* EEPROM 设备地址中的说明理解，此设备地址是一个 8 位写地址，即 bit0 代表写方向。在应用时，一般还与下面的宏 EEPROM_I2C_WR（宏值为 0）及 EEPROM_I2C_RD（宏值为 1）配合，如“EEPROM_DEV_ADDR | EEPROM_I2C_WR”这个或运算仍然表示 8 位的写方向地址（运算结果的 bit0=0），而“EEPROM_DEV_ADDR | EEPROM_I2C_RD”则用来表示 8 位的读方向地址（运算结果的 bit0=1）。

这段做代码的核心是 ee_CheckDevice 函数，它接收一个 _Address 参数作为要访问的设备地址，其函数内部过程如下：

(1) 使用 i2c_Start 函数产生一个 I2C 起始信号；

(2) 调用 i2c_SendByte 函数把设备地址 _Address 发送出去，发送时使用宏 EEPROM_I2C_WR 与 _Address 作“或”运算，表示它向 I2C 总线上广播寻找地址为 _Address 的设备，若总线上有该设备且工作正常，该设备将按照 I2C 协议返回一个应答信号；

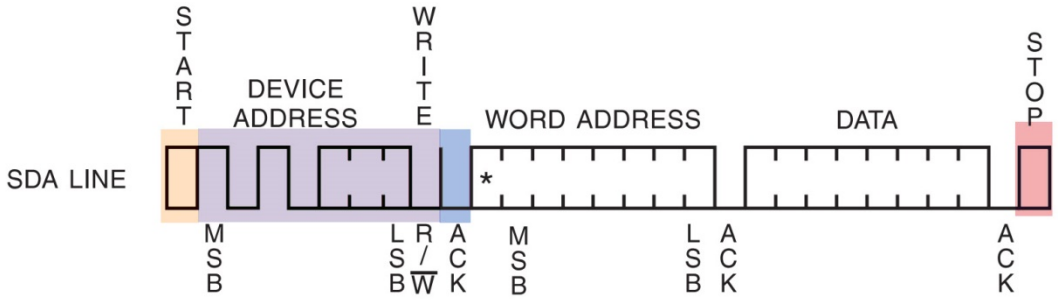
(3) 调用 i2c_WaitAck 函数检测应答信号，若检测到，说明 EEPROM 工作正常；

(4) 检测响应后调用 i2c_Stop 发送停止信号，结束本次通讯；

(5) 把响应结果作为 EEPROM 检测结果返回，为 0 时表示 EEPROM 正常，为 1 表示异常。

这个函数实现的过程其实是下图 *EEPROM 单字节写入时序* 中的一部分，该图展示了如何向 ERPOM 写入一个字节的数据，此处我们仅使用了开头发送设备地址的一段，主要是为了获取 EEP-

ROM 响应以判断 EEPROM 是否正常，不需要后续的数据写入过程，所以在响应之后就发送结束信号关闭此次通讯。



在完整的单字节写入通讯过程中，I2C 主机（此处为 STM32）发送的第 1 个字节为 EEPROM 的设备地址，用于从 I2C 总线上的众多设备中寻找到 EEPROM，得到 EEPROM 的响应后，再发送第 2 个字节，当 EEPROM 接收到这个字节后，会把它理解成将要写入的 EEPROM 内存地址（即 EEPROM 内部的 0-255 个存储单元，它指定要存储在哪），第 3 个字节是要写入的数据内容。

对于 STM32 来说，无论是 EEPROM 的设备地址、内存地址还是要写入的数据，都是一个数值而已，本质都是一样的。它们都可使用 i2c_SendByte 函数一个字节一个字节地发送出去，而在 EEPROM 端，对数值的解释不同，它就分别代表了设备地址、内存地址及要写入的数据。

关于 EEPROM 的单字节写入，由于它比后面要介绍的“页写入”效率要低得多，所以在本工程中都使用“页写入”函数来代替，就没有编写专门的单字节写入的代码了。

列表 12: 代码清单 23-5b 向 EEPROM 写入一个字节的数据

```
/**
* @brief   写一个字节到 I2C EEPROM 中
* @param
*   @arg pBuffer: 缓冲区指针
*   @arg WriteAddr: 写地址
* @retval  无
*/
uint32_t I2C_EE_ByteWrite(uint8_t* pBuffer, uint8_t WriteAddr)
{
```

(下页继续)

(续上页)

```
    HAL_StatusTypeDef status = HAL_OK;

    status = HAL_I2C_Mem_Write(&I2C_Handle, EEPROM_ADDRESS, (uint16_t)
            WriteAddr, I2C_MEMADD_SIZE_8BIT, pBuffer, 1, 100);

    /* Check the communication status */
    if (status != HAL_OK) {
        /* Execute user timeout callback */
        //I2Cx_Error(Addr);
    }
    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {

    }

    /* Check if the EEPROM is ready for a new operation */
    while (HAL_I2C_IsDeviceReady(&I2C_Handle, EEPROM_ADDRESS,
            EEPROM_MAX_TRIALS, I2Cx_TIMEOUT_MAX) == HAL_TIMEOUT);

    /* Wait for the end of the transfer */
    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {

    }
    return status;
}
```

这里我们只是简单调用库函数 HAL_I2C_Mem_Write 就可以实现，通过封装一次使用更方。

在这个通讯过程中，STM32 实际上通过 I2C 向 EEPROM 发送了两个数据，但为何第一个数据被解释为 EEPROM 的内存地址？这是由 EEPROM 的自己定义的单字节写入时序，见图 *23_14*.

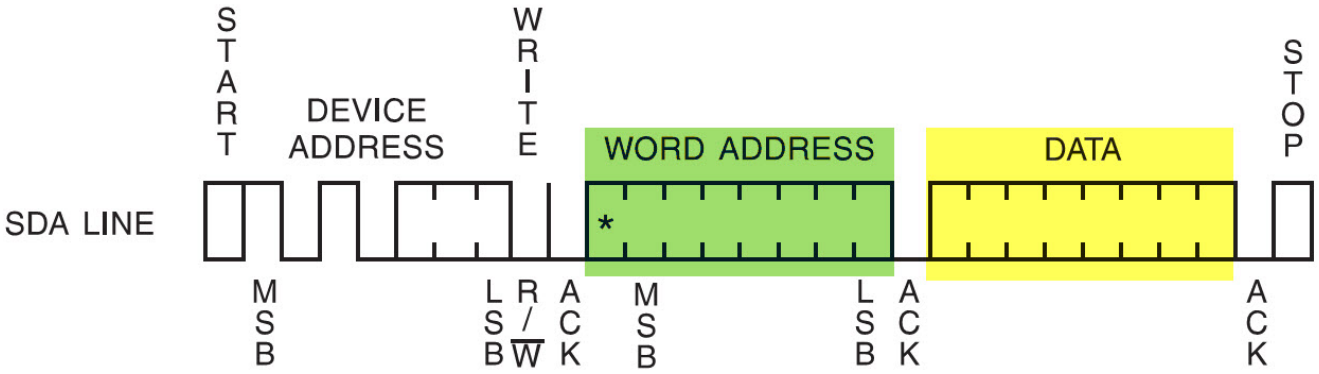


图 23-14 EEPROM 单字节写入时序 (摘自《AT24C02》规格书)

EEPROM 的单字节时序规定，向它写入数据的时候，第一个字节为内存地址，第二个字节是要写入的数据内容。所以我们需要理解：命令、地址的本质都是数据，对数据的解释不同，它就有了不同的功能。

22.4.2.2.5 EEPROM 的页写入

在以上的数据通讯中，每写入一个数据都需要向 EEPROM 发送写入的地址，我们希望向连续地址写入多个数据的时候，只要告诉 EEPROM 第一个内存地址 address1，后面的数据按次序写入到 address2、address3…这样可以节省通讯的内容，加快速度。为应对这种需求，EEPROM 定义了一种页写入时序，见图 *23_15*。

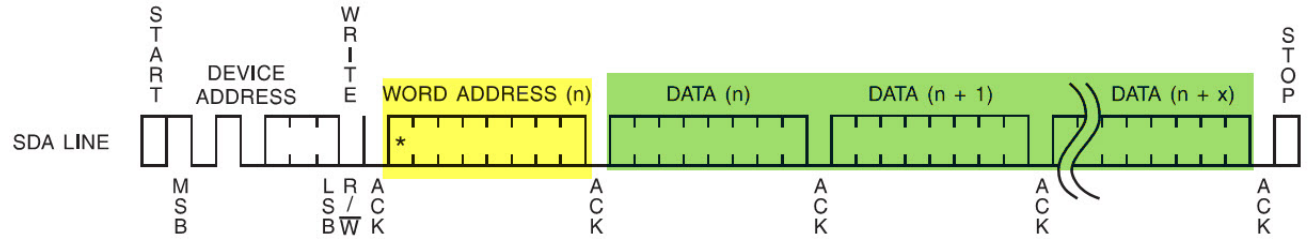


图 23-15 EEPROM 页写入时序 (摘自《AT24C02》规格书)

根据页写入时序，第一个数据被解释为要写入的内存地址 address1，后续可连续发送 n 个数据，这些数据会依次写入到内存中。其中 AT24C02 型号的芯片页写入时序最多可以一次发送 8 个数据 (即 n = 8)，该值也称为页大小，某些型号的芯片每个页写入时序最多可传输 16 个数据。

EEPROM 的页写入代码实现，对于 MINI 开发板，见*代码清单 23_6a*，对于指南者和霸道开发板、拂晓电机开发板，见*代码清单 23_6b*。

EEPROM 的页写入及状态等待

列表 13: 代码清单 23_6a MINI 开发板：EEPROM 的页写入及状态等待

```
1  /*bsp_i2c_ee.h 文件 */
2  /* 24xx02 的页面大小 */
3  #define EEPROM_PAGE_SIZE      8
4
5  /*bsp_i2c_ee.c 文件 */
6  /*
7  *********************************************************
8  * 函 数 名: ee_WaitStandby
9  * 功能说明: 等待 EEPROM 到准备状态，在写入数据后，必须调用本函数
10 *             写入操作时，使用 I2C 把数据传输到 EEPROM 后，
11 *             EEPROM 会向内部空间写入数据需要一定的时间，
12 *             当 EEPROM 内部写入完成后会对 I2C 的设备寻址有响应，
13 *             调用本函数可等待至 EEPROM 内部时序写入完毕
14 * 形    参: 无
15 * 返 回 值: 0 表示正常，1 表示等待超时
16 *********************************************************
17 */
18 uint8_t ee_WaitStandby(void)
19 {
20     uint32_t wait_count = 0;
21
22     while (ee_CheckDevice(EEPROM_DEV_ADDR)) {
23         //若检测超过次数，退出循环
24         if (wait_count++>0xFFFF) {
25             //等待超时
26             return 1;
```

(下页继续)

(续上页)

```
27          }
28      }
29      //等待完成
30      return 0;
31  }
32
33  /*
34  *********************************************************
35  * 函 数 名: ee_WriteBytes
36  * 功能说明: 向串行 EEPROM 指定地址写入若干数据，采用页写操作提高写入效率
37  * 形    参: _usAddress : 起始地址
38  *           _usSize     : 数据长度，单位为字节
39  *           _pWriteBuf  : 存放读到的数据的缓冲区指针
40  * 返 回 值: 0 表示失败，1 表示成功
41  *********************************************************
42  */
43  uint8_t ee_WriteBytes(uint8_t *_pWriteBuf, uint16_t _usAddress,
44  uint16_t _usSize)
45  {
46      uint16_t i,m;
47      uint16_t usAddr;
48
49      /*
50      写串行 EEPROM 不像读操作可以连续读取很多字节，每次写操作只能在同一个 page。
51      对于 24xx02，page size = 8
52      简单的处理方法为：按字节写操作模式，每写 1 个字节，都发送地址
53      为了提高连续写的效率：本函数采用 page wirte 操作。
54      */
55
56      usAddr = _usAddress;
57      for (i = 0; i < _usSize; i++) {
58          /* 当发送第 1 个字节或是页面首地址时，需要重新发起启动信号和地址 */
```

(下页继续)

(续上页)

```
59          if ((i == 0) || (usAddr & (EEPROM_PAGE_SIZE - 1)) == 0) {
60              /*   第 0 步：发停止信号，结束上一页的通讯，准备下一次通讯   */
61              i2c_Stop();
62
63              /* 通过检查器件应答的方式，判断内部写操作是否完成，一般小于 10ms
64              CLK 频率为 200KHz 时，查询次数为 30 次左右
65              原理同 ee_WaitStandby 函数，但该函数检查完成后会产生停止信号，
66              不适用于此处 */
67              for (m = 0; m < 1000; m++) {
68                  /* 第 1 步：发起 I2C 总线启动信号 */
69                  i2c_Start();
70
71                  /* 第 2 步：发起控制字节，高 7bit 是地址，bit0 是读写控制位，0 表示写，1
    表示读 */
72                  i2c_SendByte(EEPROM_DEV_ADDR | EEPROM_I2C_WR);  /* 此处是写指
    令 */
73
74                  /* 第 3 步：发送一个时钟，判断器件是否正确应答 */
75                  if (i2c_WaitAck() == 0) {
76                      break;
77                  }
78              }
79              if (m  == 1000) {
80                  goto cmd_fail;  /* EEPROM 器件写超时 */
81              }
82
83              /* 第 4 步：发送字节地址，24C02 只有 256 字节，因此 1 个字节就够了，
84              如果是 24C04 以上，那么此处需要连发多个地址 */
85              i2c_SendByte((uint8_t)usAddr);
86
87              /* 第 5 步：等待 ACK */
88              if (i2c_WaitAck() != 0) {
```

(下页继续)

(续上页)

```
89                  goto cmd_fail;  /* EEPROM 器件无应答 */
90              }
91          }
92
93          /* 第 6 步：开始写入数据 */
94          i2c_SendByte(_pWriteBuf[i]);
95
96          /* 第 7 步：发送 ACK */
97          if (i2c_WaitAck() != 0) {
98              goto cmd_fail;  /* EEPROM 器件无应答 */
99          }
100
101         usAddr++; /* 地址增 1 */
102     }
103
104     /* 命令执行成功，发送 I2C 总线停止信号 */
105     i2c_Stop();
106
107     //等待最后一次 EEPROM 内部写入完成
108     if (ee_WaitStandby() == 1)
109         goto cmd_fail;
110
111     return 1;
112
113 cmd_fail: /* 命令执行失败后，切记发送停止信号，避免影响 I2C 总线上其他设备 */
114     /* 发送 I2C 总线停止信号 */
115     i2c_Stop();
116     return 0;
117 }
```

这段代码中包含了一个 ee_WaitStandby 函数，它的作用是等待 EEPROM 准备就绪。当 EEPROM 接收完 STM32 传来要写入的数据后（即一次 I2C 通讯完毕），它会启动内部时序，往存储单元

内写入接收到的数据，而这个过程需要花费一点时间，在此期间 EEPROM 不会响应外部的任何 I2C 信号，所以下一次操作前需要确保 EEPROM 内部写入时序完成。ee_WaitStandby 函数就实现了这样的功能，它直接循环调用 ee_CheckDevice 函数，前面已经讲解了 ee_CheckDevice 函数的功能是发送 EEPROM 设备地址然后检查响应，所以此处实质是循环等待直至获取到 EEPROM 的响应信号，或检测超过次数认为响应失败，一旦获取到 EEPROM 的响应即可认为它的内部写入时序完成，可以进行后续操作。

接下来分析 ee_WriteBytes 函数，它接收三个输入参数：_pWriteBuf、_usAddress 和 _usSize 分别表示要存储到 EEPROM 的数据，要存储到 EEPROM 的存储单元起始地址及要存储多少个数据，函数的执行流程如下：

(1) 整个函数的操作主要由第 58 行的 for 循环完成，它循环 _usSize 次，即有多少个要发送的数据它就执行多少次，每次循环会发送一个字节的数据；

(2) 由于本函数采用的是页写入方式，一个发送过程最多可以发送 8 个字节的数据，所以每发送 8 个数据后发送一次起始信号、设备地址及内存地址即可，第 60 行的 if 就是作这样的判断，每发送完一页数据 EEPROM_PAGE_SIZE（8 个字节），那么条件成立，执行第 60-90 行的 if 分支；

(3) 在这个分支中，先调用 i2c_Stop 函数，结束上一次的页写入过程，准备下一次通讯；

(4) 在第 68 行的 for 循环，使用 i2c_Start、i2c_SendByte 及 i2c_WaitAck 函数产生新的一次起始信号、对 EEPROM 设备寻址并检测 EEPROM 是否有响应，若有响应跳出这个循环，若尝试了 1000 次还没接收到响应则认为 EEPROM 寻址超时，可能存在硬件错误，使用 goto 跳转到函数尾的错误处理并退出函数。此处使用 for 循环跟前面等待 EEPROM 内部写入时序完成的 ee_WaitStandby 函数思路是一样的，它等待至上一次写入完成后才进行下次的操作，不直接调用 ee_WaitStandby 函数是因为该函数检测后会发送 I2C 停止信号，而此处希望等待完成后进行后续的页写入操作；

(5) 若检测到 EEPROM 响应，本函数将执行到第 86 行，这里调用 i2c_SendByte 函数发送将要写入的 EEPROM 内部存储单元地址，发送后检查应答；

(6) 若一切顺利，就可以开始发送数据了，程序执行到第 95 行，注意这里的代码已经不属于前面第 60 行 if 语句每 8 个数据执行一次的条件执行区域了，所以这个 i2c_SendByte 将在每个循环执行一次，它把输入参数 _pWriteBuf 存储的数据通过 I2C 协议发送出去，而 EEPROM 也将把这些数值写入到对应的存储单元中；

(7) 第 102 行的 usAddr 地址变量自增 1，该变量参与到前面第 60 行 if 语句判断中，每 8 个循环执

行一次，并把该地址发送给 EEPORM；

(8) 第 106 行，已经退出发送数据循环，所有数据已经发送完成，这里调用 i2c_Stop 结束最后一次页写入；

(9) 第 109 行，调用 ee_WaitStandby 函数等待最后一次页写入操作的内部时序完成，防止调用本函数后用户立刻执行其它 EEPROM 操作时无响应，等待结束后使用 return 退出函数；

(10) 第 114 行是标号 cmd_fail 的处理，前面写入过程产生各种错误后都跳转至此处，调用 i2c_Stop 函数发送停止信号，释放 I2C 总线，然后使用 return 退出函数。

列表 14: 代码清单 23_6b 指南者和霸道开发板、拂晓电机开发板：EEPROM 的页写入

```
/**
* @brief  在 EEPROM 的一个写循环中可以写多个字节，但一次写入的字节数
*         不能超过 EEPROM 页的大小，AT24C02 每页有 8 个字节
* @param
*   @arg pBuffer: 缓冲区指针
*   @arg WriteAddr: 写地址
*     @arg NumByteToWrite: 写的字节数
* @retval  无
*/
uint32_t I2C_EE_PageWrite(uint8_t* pBuffer, uint8_t WriteAddr,
uint8_t NumByteToWrite)
{
    HAL_StatusTypeDef status = HAL_OK;
    /* Write EEPROM_PAGESIZE */
    status=HAL_I2C_Mem_Write(&I2C_Handle, EEPROM_ADDRESS,WriteAddr,
I2C_MEMADD_SIZE_8BIT, (uint8_t*)(pBuffer),NumByteToWrite, 100);

    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {

    }
```

(下页继续)

(续上页)

```
    /* Check if the EEPROM is ready for a new operation */
    while (HAL_I2C_IsDeviceReady(&I2C_Handle, EEPROM_ADDRESS,
EEPROM_MAX_TRIALS, I2Cx_TIMEOUT_MAX) == HAL_TIMEOUT);
    /* Wait for the end of the transfer */
    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {


    }
    return status;
}
```

这段页写入函数主体跟单字节写入函数是一样的，只是它在发送数据的时候，使用 while 循环控制发送多个数据，发送完多个数据后才产生 I2C 停止信号，只要每次传输的数据小于等于 EEPROM 时序规定的页大小，就能正常传输。

22.4.2.2.6 多字节写入

多次写入数据时，利用 EEPROM 的页写入方式，避免单字节读写时候的等待。多个数据写入过程见*代码清单 23_7*。

列表 15: 代码清单 23-7 指南者和霸道开发板、拂晓电机开发板：多字节写入

```
/**
* @brief   将缓冲区中的数据写到 I2C EEPROM 中
* @param
*   @arg pBuffer: 缓冲区指针
*   @arg WriteAddr: 写地址
*     @arg NumByteToWrite: 写的字节数
* @retval  无
*/
void I2C_EE_BufferWrite(uint8_t* pBuffer, uint8_t WriteAddr,
```

(下页继续)

(续上页)

```
uint16_t NumByteToWrite)
{
    uint8_t NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0;

    Addr = WriteAddr % EEPROM_PAGESIZE;
    count = EEPROM_PAGESIZE - Addr;
    NumOfPage =  NumByteToWrite / EEPROM_PAGESIZE;
    NumOfSingle = NumByteToWrite % EEPROM_PAGESIZE;

    /* If WriteAddr is I2C_PageSize aligned  */
    if (Addr == 0) {
        /* If NumByteToWrite < I2C_PageSize */
        if(NumByteToWrite <= count)
        {
            I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
        }
        /* If NumByteToWrite > I2C_PageSize */
        else {
            while (NumOfPage--) {
                I2C_EE_PageWrite(pBuffer, WriteAddr, EEPROM_PAGESIZE);
                WriteAddr +=  EEPROM_PAGESIZE;
                pBuffer += EEPROM_PAGESIZE;
            }

            if (NumOfSingle!=0) {
                I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
            }
        }
    }
    /* If WriteAddr is not I2C_PageSize aligned  */
    else {
        /* If NumByteToWrite < I2C_PageSize */
```

(下页继续)

(续上页)

```
        if (NumOfPage== 0) {
            I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
        }
        /* If NumByteToWrite > I2C_PageSize */
        else {
            NumByteToWrite -= count;
            NumOfPage =  NumByteToWrite / EEPROM_PAGESIZE;
            NumOfSingle = NumByteToWrite % EEPROM_PAGESIZE;

            if (count != 0) {
                I2C_EE_PageWrite(pBuffer, WriteAddr, count);
                WriteAddr += count;
                pBuffer += count;
            }

            while (NumOfPage--) {
                I2C_EE_PageWrite(pBuffer, WriteAddr, EEPROM_PAGESIZE);
                WriteAddr +=  EEPROM_PAGESIZE;
                pBuffer += EEPROM_PAGESIZE;
            }
            if (NumOfSingle != 0) {
                I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
            }
        }
    }
}
```

很多读者觉得这段代码的运算很复杂，看不懂，其实它的主旨就是对输入的数据进行分页 (本型号芯片每页 8 个字节)，见表 23-2。通过“整除”计算要写入的数据 NumByteToWrite 能写满多少“完整的页”，计算得的值存储在 NumOfPage 中，但有时数据不是刚好能写满完整页的，会多一点出来，通过“求余”计算得出“不满一页的数据个数”就存储在 NumOfSingle 中。计算后通过按页传输 NumOfPage 次整页数据及最后的 NumOfSing 个数据，使用页传输，比之前的单个字节

数据传输要快很多。

除了基本的分页传输，还要考虑首地址的问题，见表 23-3。若首地址不是刚好对齐到页的首地址，会需要一个 count 值，用于存储从该首地址开始写满该地址所在的页，还能写多少个数据。实际传输时，先把这部分 count 个数据先写入，填满该页，然后把剩余的数据 (NumByteToWrite-count)，再重复上述求出 NumOPage 及 NumOfSingle 的过程，按页传输到 EEPROM。

1. 若 writeAddress=16，计算得 Addr=16%8= 0 ，count=8-0= 8；
2. 同时，若 NumOfPage=22，计算得 NumOfPage=22/8= 2，NumOfSingle=22%8= 6。
3. 数据传输情况如表 23-2

表 23-2 首地址对齐到页时的情况

不影响	0	1	2	3	4	5	6	7
不影响	8	9	10	11	12	13	14	15
第 1 页	16	17	18	19	20	21	22	23
第 2 页	24	25	26	27	28	29	30	31
NumOfSingle=6	32	33	34	35	36	37	38	39

4. 若 writeAddress=17，计算得 Addr=17%8= 1，count=8-1= 7；
5. 同时，若 NumOfPage=22，
6. 先把 count 去掉，特殊处理，计算得新的 NumOfPage=22-7= 15
7. 计算得 NumOfPage=15/8= 1，NumOfSingle=15%8= 7。
8. 数据传输情况如表 23-3

表 23-3 首地址未对齐到页时的情况

不影响	0	1	2	3	4	5	6	7
不影响	8	9	10	11	12	13	14	15
count=7	16	17	18	19	20	21	22	23
第 1 页	24	25	26	27	28	29	30	31
NumOfSingle=7	32	33	34	35	36	37	38	39

最后，强调一下，EEPROM 支持的页写入只是一种加速的 I2C 的传输时序，实际上并不要求每次都以页为单位进行读写，EEPROM 是支持随机访问的 (直接读写任意一个地址)，如前面的单个字节写入。在某些存储器，如 NAND FLASH，它是必须按照 Block 写入的，例如每个 Block 为 512 或 4096 字节，数据写入的最小单位是 Block，写入前都需要擦除整个 Block；NOR FLASH 则是写入前必须以 Sector/Block 为单位擦除，然后才可以按字节写入。而我们的 EEPROM 数据写入和擦除的最小单位是“字节”而不是“页”，数据写入前不需要擦除整页。

22.4.2.2.7 从 EEPROM 读取数据

从 EEPROM 读取数据是一个复合的 I2C 时序，它实际上包含一个写过程和一个读过程，见图 *23_16*。

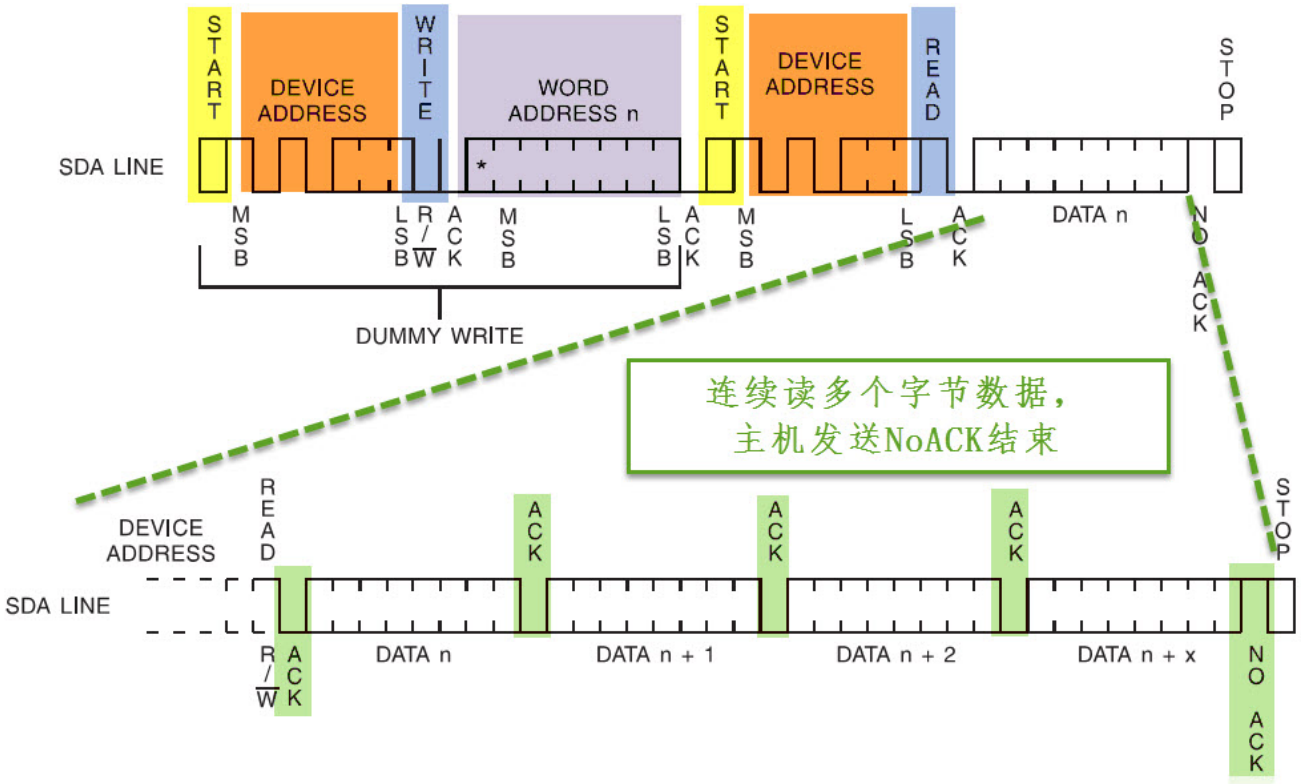


图 23-16 EEPROM 数据读取时序

读时序的第一个通讯过程中，使用 I2C 发送设备地址寻址 (写方向)，接着发送要读取的“内存地

址”；第二个通讯过程中，再次使用 I2C 发送设备地址寻址，但这个时候的数据方向是读方向；在这个过程之后，EEPROM 会向主机返回从“内存地址”开始的数据，一个字节一个字节地传输，只要主机的响应为“应答信号”，它就会一直传输下去，主机想结束传输时，就发送“非应答信号”，并以“停止信号”结束通讯，作为从机的 EEPROM 也会停止传输。HAL 库已经帮我们实现了这一个过程，我们只是简单封装一下就可以直接使用，实现代码对于 MINI 开发板，见代码清单 23_8a，对于指南者和霸道开发板、拂晓电机开发板，见代码清单 23_8b。

列表 16: 代码清单 23-8a MINI 开发板：从 EEPROM 读取数据

```
1  /*
2  *********************************************************
3  * 函 数 名: ee_ReadBytes
4  * 功能说明: 从串行 EEPROM 指定地址处开始读取若干数据
5  * 形    参: _usAddress : 起始地址
6      *       _usSize     : 数据长度，单位为字节
7      *       _pReadBuf   : 存放读到的数据的缓冲区指针
8  * 返 回 值: 0 表示失败，1 表示成功
9  *********************************************************
10 */
11 uint8_t ee_ReadBytes(uint8_t *_pReadBuf, uint16_t _usAddress,
12                       uint16_t _usSize)
13 {
14     uint16_t i;
15
16     /* 采用串行 EEPROM 随即读取指令序列，连续读取若干字节 */
17
18     /* 第 1 步: 发起 I2C 总线启动信号 */
19     i2c_Start();
20
21     /* 第 2 步: 发起控制字节，高 7bit 是地址，bit0 是读写控制位，0 表示写，1 表示读 */
22     i2c_SendByte(EEPROM_DEV_ADDR | EEPROM_I2C_WR);  /* 此处是写方向 */
23
```

(下页继续)

(续上页)

```
    /* 第 3 步：等待 ACK */
    if (i2c_WaitAck() != 0) {
        goto cmd_fail;  /* EEPROM 器件无应答 */
    }

    /* 第 4 步：发送字节地址，24C02 只有 256 字节，因此 1 个字节就够了，
    如果是 24C04 以上，那么此处需要连发多个地址 */
    i2c_SendByte((uint8_t)_usAddress);

    /* 第 5 步：等待 ACK */
    if (i2c_WaitAck() != 0) {
        goto cmd_fail;  /* EEPROM 器件无应答 */
    }

    /* 第 6 步：重新启动 I2C 总线。
    前面的代码的目的向 EEPROM 传送地址，下面开始读取数据 */
    i2c_Start();

    /* 第 7 步：发起控制字节，高 7bit 是地址，bit0 是读写控制位，
    0 表示写，1 表示读 */
    i2c_SendByte(EEPROM_DEV_ADDR | EEPROM_I2C_RD);  /* 此处是读指令 */

    /* 第 8 步：发送 ACK */
    if (i2c_WaitAck() != 0) {
        goto cmd_fail;  /* EEPROM 器件无应答 */
    }

    /* 第 9 步：循环读取数据 */
    for (i = 0; i < _usSize; i++) {
        _pReadBuf[i] = i2c_ReadByte();  /* 读 1 个字节 */

        /* 每读完 1 个字节后，需要发送 Ack， 最后一个字节不需要 Ack，发 Nack */
```

(下页继续)

(续上页)

```
56          if (i != _usSize - 1) {
57              i2c_Ack();  /* 中间字节读完后，CPU 产生 ACK 信号 (驱动 SDA = 0) */
58          } else {
59              i2c_NAck(); /* 最后 1 个字节读完后，CPU 产生 NACK 信号 (驱动 SDA = 1)
60
61          }
62      }
63      /* 发送 I2C 总线停止信号 */
64      i2c_Stop();
65      return 1; /* 执行成功 */
66
67  cmd_fail: /* 命令执行失败后，切记发送停止信号，避免影响 I2C 总线上其他设备 */
68      /* 发送 I2C 总线停止信号 */
69      i2c_Stop();
70      return 0;
71  }
```

这段读取 EEPROM 内容的 ee_ReadBytes 函数写过程跟前面的写入函数类似：

(1) 产生第一次起始信号，并发送 EEPROM 设备地址，地址中的 bit0 表示写方向；

(2) 接收到应答后，向 EEPROM 发送要读取的存储单元地址；

(3) 接收到应答后，产生第二次起始信号，发送 EEPROM 的设备地址，地址中的 bit0 表示读方向，注意两次起始信号间没有停止信号，这依然是同一次通讯过程；

(4) 在 for 循环中，连续调用 i2c_ReadByte 函数一个字节一个字节地接收 EEPROM 返回的数据内容并存储到数组中；

(5) 接收到足够的数据后，产生停止信号，退出函数。

列表 17: 代码清单 23-8b 指南者和霸道开发板、拂晓电机开发板：从 EEPROM 读取数据

```
/**
* @brief   从 EEPROM 里面读取一块数据
* @param
*   @arg pBuffer: 存放从 EEPROM 读取的数据的缓冲区指针
*   @arg WriteAddr: 接收数据的 EEPROM 的地址
*     @arg NumByteToWrite: 要从 EEPROM 读取的字节数
* @retval  无
*/
uint32_t I2C_EE_BufferRead(uint8_t* pBuffer, uint8_t ReadAddr, uint16_t␣
↪NumByteToRead)
{
    HAL_StatusTypeDef status = HAL_OK;

    status=HAL_I2C_Mem_Read(&I2C_Handle,EEPROM_ADDRESS,ReadAddr,
    I2C_MEMADD_SIZE_8BIT, (uint8_t *)pBuffer, NumByteToRead, 1000);

    return status;
}
```

这里代码非常简单，我们只需要确定 I2C 的地址，数据格式，数据存储指针，数据大小，超时设置就可以把想要的数据读回来。

22.4.2.3 main 文件

22.4.2.3.1 EEPROM 读写测试函数

完成基本的读写函数后，接下来我们编写一个读写测试函数来检验驱动程序。对于 MINI 开发板，见*代码清单 23_9a*，对于指南者和霸道开发板、拂晓电机开发板，见*代码清单 23_9b*。

列表 18: 代码清单 23-9a MINI 开发板的 EEPROM 读写测试函数

```
/*
* eeprom AT24C02 读写测试
* 正常返回 1，异常返回 0
*/
uint8_t ee_Test(void)
{
    uint16_t i;
    uint8_t write_buf[EEPROM_SIZE];
    uint8_t read_buf[EEPROM_SIZE];

    /*-----------------------------------------------------------*/
    if (ee_CheckDevice(EEPROM_DEV_ADDR) == 1) {
        /* 没有检测到 EEPROM */
        printf("没有检测到串行 EEPROM!\r\n");

        return 0;
    }
    /*-----------------------------------------------------------*/
    /* 填充测试缓冲区 */
    for (i = 0; i < EEPROM_SIZE; i++) {
        write_buf[i] = i;
    }
    /*-----------------------------------------------------------*/
    if (ee_WriteBytes(write_buf, 0, EEPROM_SIZE) == 0) {
        printf("写 eeprom 出错! \r\n");
        return 0;
    } else {
        printf("写 eeprom 成功! \r\n");
    }
```

(下页继续)

(续上页)

```
30
31      /*-------------------------------------------------------*/
32      if (ee_ReadBytes(read_buf, 0, EEPROM_SIZE) == 0) {
33          printf("读 eeprom 出错! \r\n");
34          return 0;
35      } else {
36          printf("读 eeprom 成功, 数据如下: \r\n");
37      }
38      /*-------------------------------------------------------*/
39      for (i = 0; i < EEPROM_SIZE; i++) {
40          if (read_buf[i] != write_buf[i]) {
41              printf("0x%02X ", read_buf[i]);
42              printf("错误:EEPROM 读出与写入的数据不一致");
43              return 0;
44          }
45          printf(" %02X", read_buf[i]);
46
47          if ((i & 15) == 15) {
48              printf("\r\n");
49          }
50      }
51      printf("eeprom 读写测试成功\r\n");
52      return 1;
53  }
```

在这个测试函数中先调用了 ee_CheckDevice 函数检测 EEPROM 的响应，通过是否接收到响应来判断 EEPROM 是否工作正常；接着填充一个数组，数组的内容为 0，1，2，3 至 N，然后把这个数组的内容写入到 EEPROM 中，写入完毕后再从 EEPROM 的地址中读取数据，把读取得到的与写入的数据进行校验，若一致说明读写正常，否则读写过程有问题或者 EEPROM 芯片不正常。

列表 19: 代码清单 23-9b 指南者和霸道开发板、拂晓电机开发板的 EEPROM 读写测试函数

```
/**
* @brief  I2C(AT24C02) 读写测试
* @param  无
* @retval 正常返回 1 , 不正常返回 0
*/
uint8_t I2C_Test(void)
{
    uint16_t i;

    EEPROM_INFO("写入的数据");

    for ( i=0; i<DATA_Size; i++ ) { //填充缓冲
        I2c_Buf_Write[i] =i;
        printf("0x%02X ", I2c_Buf_Write[i]);
        if (i%16 == 15)
            printf("\n\r");
    }

    //将 I2c_Buf_Write 中顺序递增的数据写入 EERPOM 中
    I2C_EE_BufferWrite( I2c_Buf_Write, EEP_Firstpage, DATA_Size);

    EEPROM_INFO("读出的数据");
    //将 EEPROM 读出数据顺序保持到 I2c_Buf_Read 中
    I2C_EE_BufferRead(I2c_Buf_Read, EEP_Firstpage, DATA_Size);
    //将 I2c_Buf_Read 中的数据通过串口打印
    for (i=0; i<DATA_Size; i++) {
        if (I2c_Buf_Read[i] != I2c_Buf_Write[i]) {
            printf("0x%02X ", I2c_Buf_Read[i]);
            EEPROM_ERROR("错误:I2C EEPROM 写入与读出的数据不一致");
```

(下页继续)

(续上页)

```
            return 0;
        }
        printf("0x%02X ", I2c_Buf_Read[i]);
        if (i%16 == 15)
            printf("\n\r");

    }
    EEPROM_INFO("I2C(AT24C02) 读写测试成功");
    return 1;
}
```

代码中先填充一个数组，数组的内容为 1,2,3 至 N，接着把这个数组的内容写入到 EEPROM 中，写入时采用页写入的方式。写入完毕后再从 EEPROM 的地址中读取数据，把读取得到的与写入的数据进行校验，若一致说明读写正常，否则读写过程有问题或者 EEPROM 芯片不正常。其中代码用到的 EEPROM_INFO 跟 EEPROM_ERROR 宏类似，都是对 printf 函数的封装，使用和阅读代码时把它直接当成 printf 函数就好。具体的宏定义在“bsp_i2c_ee.h 文件中”，在以后的代码我们常常会用类似的宏来输出调试信息。

22.4.2.3.2 main 函数

最后编写 main 函数，函数中初始化串口、I2C 外设，然后调用上面的 I2C_Test 函数进行读写测试，对于 MINI 开发板，见*代码清单 23_10a*，对于指南者和霸道开发板、拂晓电机开发板，见*代码清单 23_10b*。

列表 20: 代码清单 23-10a MINI 开发板的 main 函数

```
1 int main(void)
2 {
3     HAL_Init();
4     /* 配置系统时钟为 72 MHz */
5     SystemClock_Config();
```

(下页继续)

(续上页)

```
6
7       /* 初始化 USART 配置模式为 115200 8-N-1, 中断接收 */
8       DEBUG_USART_Config();
9
10      printf("\r\n 这是一个 EEPROM 读写小数和长整数实验\r\n");
11
12      /* I2C 外设初 (AT24C02) 始化 */
13      i2c_CfgGpio();
14
15      /* 读取数据标志位 */
16      ee_ReadBytes(&cal_flag, 0, 1);
17
18      if ( cal_flag != 0xCD ) {
19          /* 若标志等于 0xcd, 表示之前已有写入数据 */
20          printf("\r\n没有检测到数据标志, FLASH 没有存储数据, 即将进行小数写入实验\r\n
    ↪");
21          cal_flag =0xCD;
22
23          /* 写入标志到 0 地址 */
24          ee_WriteBytes(&cal_flag, 0, 1);
25
26          /* 生成要写入的数据 */
27          for ( k=0; k<7; k++ ) {
28              double_buffer[k] = k +0.1;
29              int_bufffer[k]=k*500+1 ;
30          }
31
32          /* 写入小数数据到地址 10*/
33          ee_WriteBytes((void*)double_buffer,DOUBLE_ADDR, sizeof(double_
    ↪buffer));
34          /* 写入整数数据到地址 60*/
35          ee_WriteBytes((void*)int_bufffer, LONGINT_ADDR, sizeof(int_
    ↪bufffer));
```

(下页继续)

(续上页)

```
36
37          printf("向芯片写入数据: ");
38          /* 打印到串口 */
39          for ( k=0; k<7; k++ ) {
40              printf("小数 tx = %LF\r\n",double_buffer[k]);
41              printf("整数 tx = %d\r\n",int_bufffer[k]);
42          }
43
44          printf("\r\n请复位开发板, 以读取数据进行检验\r\n");
45
46      } else {
47          printf("\r\n检测到数据标志\r\n");
48
49          /* 读取小数数据 */
50          ee_ReadBytes((void*)double_buffer, DOUBLE_ADDR, sizeof(double_
    ↪buffer));
51
52          /* 读取整数数据 */
53          ee_ReadBytes((void*)int_bufffer, LONGINT_ADDR, sizeof(int_bufffer));
54
55          printf("\r\n从芯片读到数据: \r\n");
56          for ( k=0; k<7; k++ ) {
57              printf("小数 rx = %LF \r\n",double_buffer[k]);
58              printf("整数 rx = %d \r\n",int_bufffer[k]);
59          }
60
61      }
62
63      while (1) {
64      }
65  }
```

列表 21: 代码清单 23-10b 指南者和霸道开发板、拂晓电机开发板的 main 函数

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /* 配置系统时钟为 72 MHz */
    SystemClock_Config();

    /* 初始化 RGB 彩灯 */
    LED_GPIO_Config();

    LED_BLUE;
    /* 初始化 USART1*/
    UARTx_Config();

    printf("\r\n 欢迎使用野火  STM32 F103 开发板。\r\n");

    printf("\r\n 这是一个 I2C 外设 (AT24C02) 读写测试例程 \r\n");

    /* I2C 外设初 (AT24C02) 始化 */
    I2C_EE_Init();

    if (I2C_Test() ==1) {
        LED_GREEN;
    } else {
        LED_RED;
    }
```

(下页继续)

(续上页)

```
    while (1) {
    }
```

22.4.3 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到 EEPROM 测试的调试信息。

图 23-17 MINI 开发板 EEPROM 测试成功

图 23-17 指南者和霸道开发板、拂晓电机开发板 EEPROM 测试成功

第 23 章　SPI—读写串行 FLASH

本章参考资料：《STM32F10X-中文参考手册》SPI 章节及《SPI 总线协议介绍》。

若对 SPI 通讯协议不了解，可先阅读《SPI 总线协议介绍》文档的内容学习。

关于 FLASH 存储器，请参考“常用存储器介绍”章节，实验中 FLASH 芯片的具体参数，请参考其规格书《W25Q64》来了解。

23.1 SPI 协议简介

SPI 协议是由摩托罗拉公司提出的通讯协议 (Serial Peripheral Interface)，即串行外围设备接口，是一种高速全双工的通信总线。它被广泛地使用在 ADC、LCD 等设备与 MCU 间，要求通讯速率较高的场合。

学习本章时，可与 I2C 章节对比阅读，体会两种通讯总线的差异以及 EEPROM 存储器与 FLASH 存储器的区别。下面我们分别对 SPI 协议的物理层及协议层进行讲解。

23.1.1 SPI 物理层

SPI 通讯设备之间的常用连接方式见图 *24_1*。

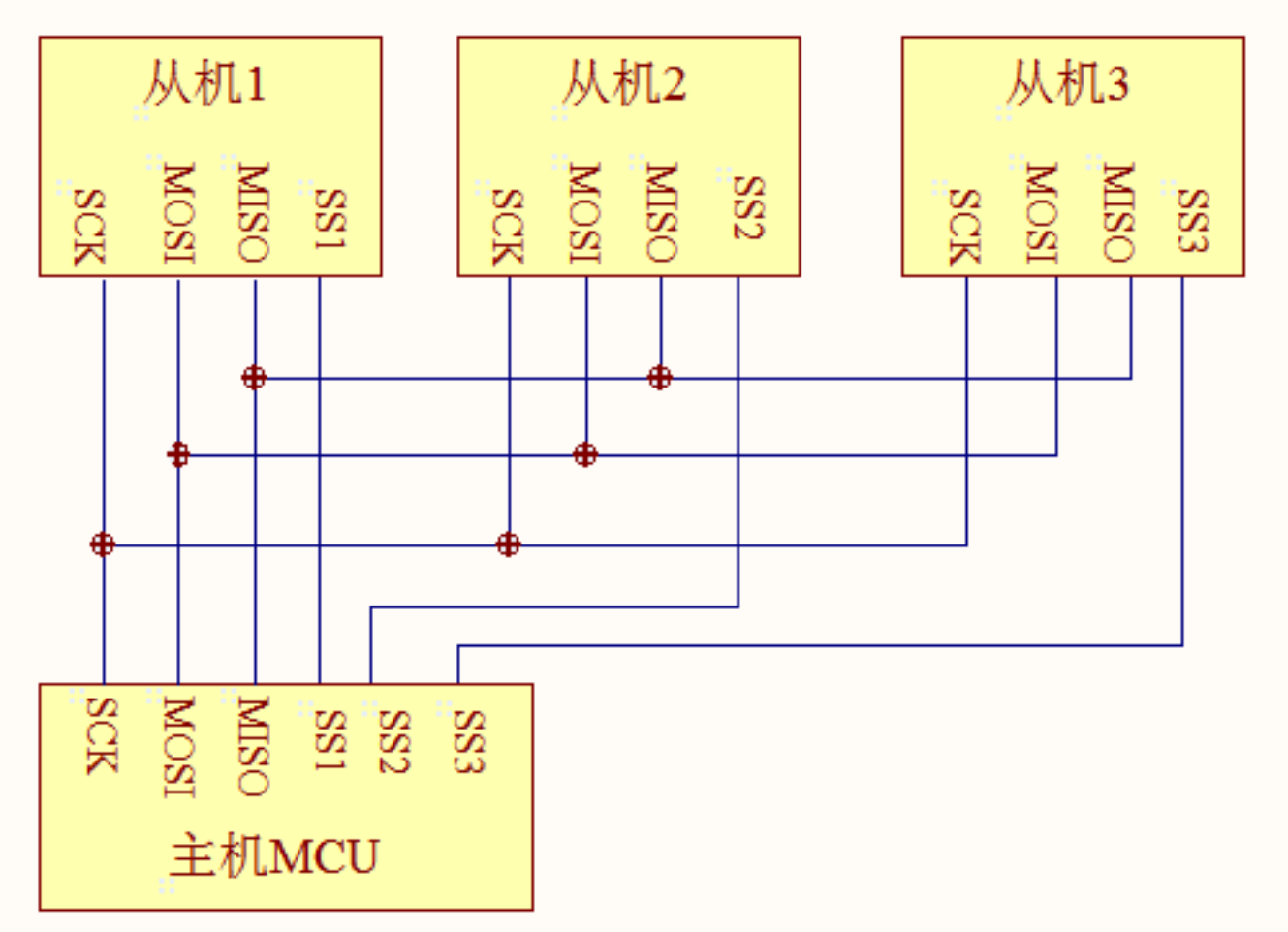


图 24-1 常见的 SPI 通讯系统

SPI 通讯使用 3 条总线及片选线，3 条总线分别为 SCK、MOSI、MISO，片选线为，它们的作用介绍如下：

(1) *(Slave Select)*：从设备选择信号线，常称为片选信号线，也称为 *NSS*、*CS*，以下用 *NSS* 表示。当有多个 SPI 从设备与 SPI 主机相连时，设备的其它信号线 SCK、MOSI 及 MIS O 同时并联到相同的 SPI 总线上，即无论有多少个从设备，都共同只使用这 3 条总线；而每个从设备都有独立的这一条 NSS 信号线，本信号线独占主机的一个引脚，即有多少个从设备，就有多少条片选信号线。I2C 协议中通过设备地址来寻址、选中总线上的某个设备并与其进

行通讯；而 SPI 协议中没有设备地址，它使用 NSS 信号线来寻址，当主机要选择从设备时，把该从设备的 NSS 信号线设置为低电平，该从设备即被选中，即片选有效，接着主机开始与被选中的从设备进行 SPI 通讯。所以 SPI 通讯以 NSS 线置低电平为开始信号，以 NSS 线被拉高作为结束信号。

(2) *SCK (Serial Clock)*：时钟信号线，用于通讯数据同步。它由通讯主机产生，决定了通讯的速率，不同的设备支持的最高时钟频率不一样，如 STM32 的 SPI 时钟频率最大为 $f_{pclk}/2$，两个设备之间通讯时，通讯速率受限于低速设备。

(3) *MOSI (Master Output，Slave Input)*：主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入主机发送的数据，即这条线上数据的方向为主机到从机。

(4) *MISO(Master Input,，Slave Output)*：主设备输入/从设备输出引脚。主机从这条信线读入数据，从机的数据由这条信号线输出到主机，即在这条线上数据的方向为从机到主机。

23.1.2 协议层

与 I2C 的类似，SPI 协议定义了通讯的起始和停止信号、数据有效性、时钟同步等环节。

23.1.2.1 SPI 基本通讯过程

先看看 SPI 通讯的通讯时序，见图 *24_2*。

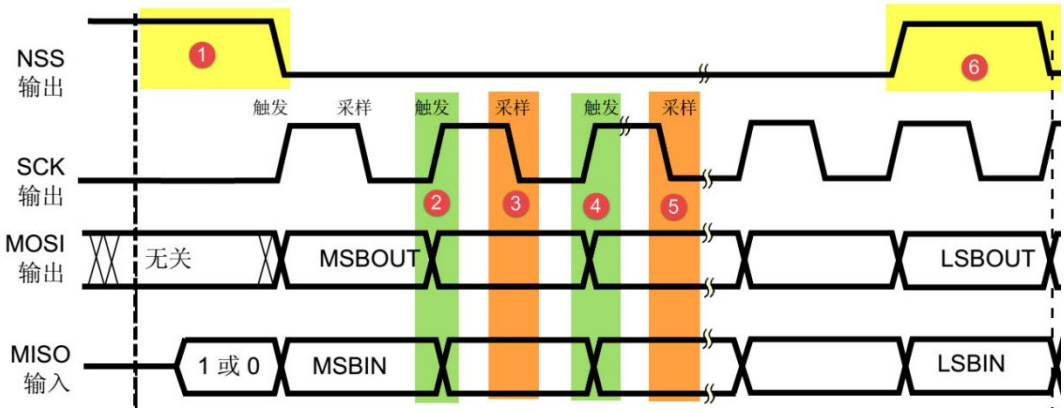


图 24-2 SPI 通讯时序

这是一个主机的通讯时序。NSS、SCK、MOSI 信号都由主机控制产生，而 MISO 的信号由从机产生，主机通过该信号线读取从机的数据。MOSI 与 MISO 的信号只在 NSS 为低电平的时候才有效，在 SCK 的每个时钟周期 MOSI 和 MISO 传输一位数据。

以上通讯流程中包含的各个信号分解如下：

23.1.2.2 通讯的起始和停止信号

在图 *24_2* 中的标号处，NSS 信号线由高变低，是 SPI 通讯的起始信号。NSS 是每个从机各自独占的信号线，当从机在自己的 NSS 线检测到起始信号后，就知道自己被主机选中了，开始准备与主机通讯。在图中的标号处，NSS 信号由低变高，是 SPI 通讯的停止信号，表示本次通讯结束，从机的选中状态被取消。

23.1.2.3 数据有效性

SPI 使用 MOSI 及 MISO 信号线来传输数据，使用 SCK 信号线进行数据同步。MOSI 及 MISO 数据线在 SCK 的每个时钟周期传输一位数据，且数据输入输出是同时进行的。数据传输时，MSB 先行或 LSB 先行并没有作硬性规定，但要保证两个 SPI 通讯设备之间使用同样的协定，一般都会采用图 24-2 中的 MSB 先行模式。

观察图中的标号处，MOSI 及 MISO 的数据在 SCK 的上升沿期间变化输出，在 SCK 的下降沿时被采样。即在 SCK 的下降沿时刻，MOSI 及 MISO 的数据有效，高电平时表示数据“1”，为低电平时表示数据“0”。在其它时刻，数据无效，MOSI 及 MISO 为下一次表示数据做准备。

SPI 每次数据传输可以 8 位或 16 位为单位，每次传输的单位数不受限制。

23.1.2.4 CPOL/CPHA 及通讯模式

上面讲述的图 24-2 中的时序只是 SPI 中的其中一种通讯模式，SPI 一共有四种通讯模式，它们的主要区别是总线空闲时 SCK 的时钟状态以及数据采样时刻。为方便说明，在此引入“时钟极性 CPOL”和“时钟相位 CPHA”的概念。

时钟极性 CPOL 是指 SPI 通讯设备处于空闲状态时，SCK 信号线的电平信号 (即 SPI 通讯开始前、NSS 线为高电平时 SCK 的状态)。CPOL=0 时，SCK 在空闲状态时为低电平，CPOL=1 时，则相

反。

时钟相位 CPHA 是指数据的采样的时刻，当 CPHA=0 时，MOSI 或 MISO 数据线上的信号将会在 SCK 时钟线的“奇数边沿”被采样。当 CPHA=1 时，数据线在 SCK 的“偶数边沿”采样。见图 *24_3* 及图 *24_4*。

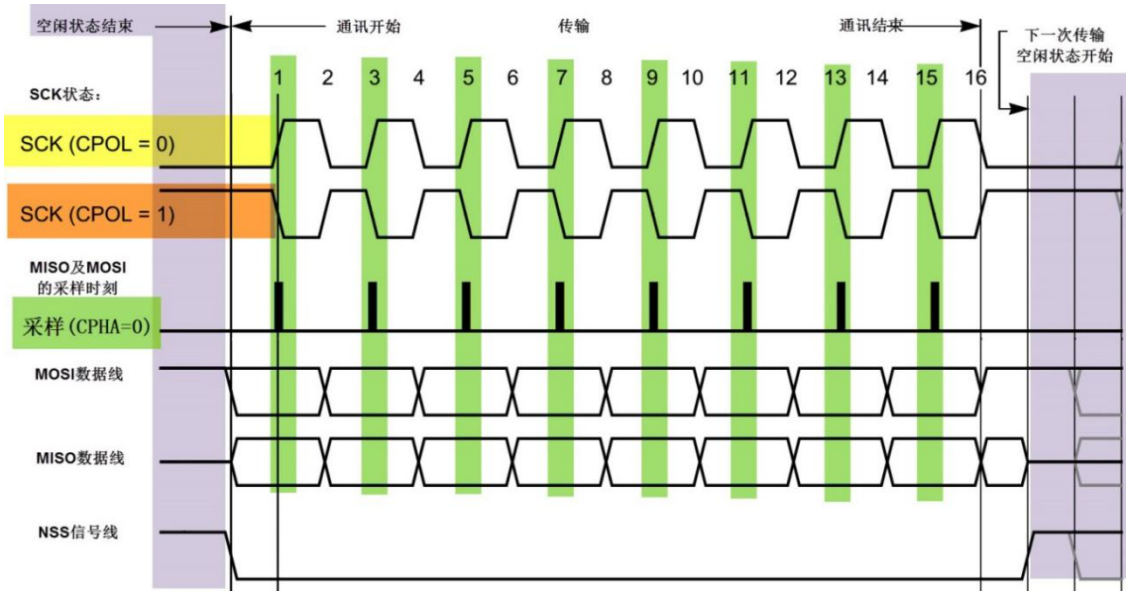


图 24-3 CPHA=0 时的 SPI 通讯模式

我们来分析这个 CPHA=0 的时序图。首先，根据 SCK 在空闲状态时的电平，分为两种情况。SCK 信号线在空闲状态为低电平时，CPOL=0；空闲状态为高电平时，CPOL=1。

无论 CPOL=0 还是 =1，因为我们配置的时钟相位 CPHA=0，在图中可以看到，采样时刻都是在 SCK 的奇数边沿。注意当 CPOL=0 的时候，时钟的奇数边沿是上升沿，而 CPOL=1 的时候，时钟的奇数边沿是下降沿。所以 SPI 的采样时刻不是由上升/下降沿决定的。MOSI 和 MISO 数据线的有效信号在 SCK 的奇数边沿保持不变，数据信号将在 SCK 奇数边沿时被采样，在非采样时刻，MOSI 和 MISO 的有效信号才发生切换。

类似地，当 CPHA=1 时，不受 CPOL 的影响，数据信号在 SCK 的偶数边沿被采样，见图 *24_4*。

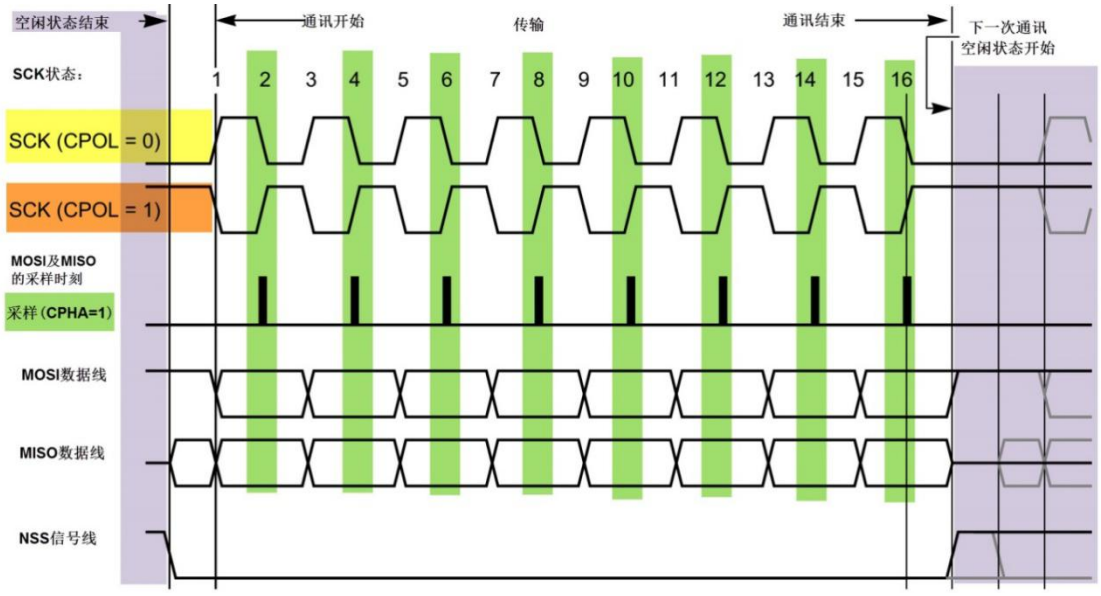


图 24-4 CPHA=1 时的 SPI 通讯模式

由 CPOL 及 CPHA 的不同状态，SPI 分成了四种模式，见表 24-1，主机与从机需要工作在相同的模式下才可以正常通讯，实际中采用较多的是“模式 0”与“模式 3”。

表 24-1 SPI 的四种模式

SPI 模式	CPOL	CPHA	空闲时 SCK 时钟	采样时刻
0	0	0	低电平	奇数边沿
1	0	1	低电平	偶数边沿
2	1	0	高电平	奇数边沿
3	1	1	高电平	偶数边沿

23.2 STM32 的 SPI 特性及架构

与 I2C 外设一样，STM32 芯片也集成了专门用于 SPI 协议通讯的外设。

23.2.1 STM32 的 SPI 外设简介

STM32 的 SPI 外设可用作通讯的主机及从机，支持最高的 SCK 时钟频率为 $f_{pclk}/2$ (STM32F103 型号的芯片默认 f:sub:*pclk1* 为 36MHz，f_{pclk2} 为 72MHz)，完全支持 SPI 协议的 4 种模式，数据帧长度可设置为 8 位或 16 位，可设置数据 MSB 先行或 LSB 先行。它还支持双线全双工 (前面小节说明的都是这种模式)、双线单向以及单线模式。其中双线单向模式可以同时使用 MOSI 及 MISO 数据线向一个方向传输数据，可以加快一倍的传输速度。而单线模式则可以减少硬件接线，当然这样速率会受到影响。我们只讲解双线全双工模式。

23.2.2 STM32 的 SPI 架构剖析

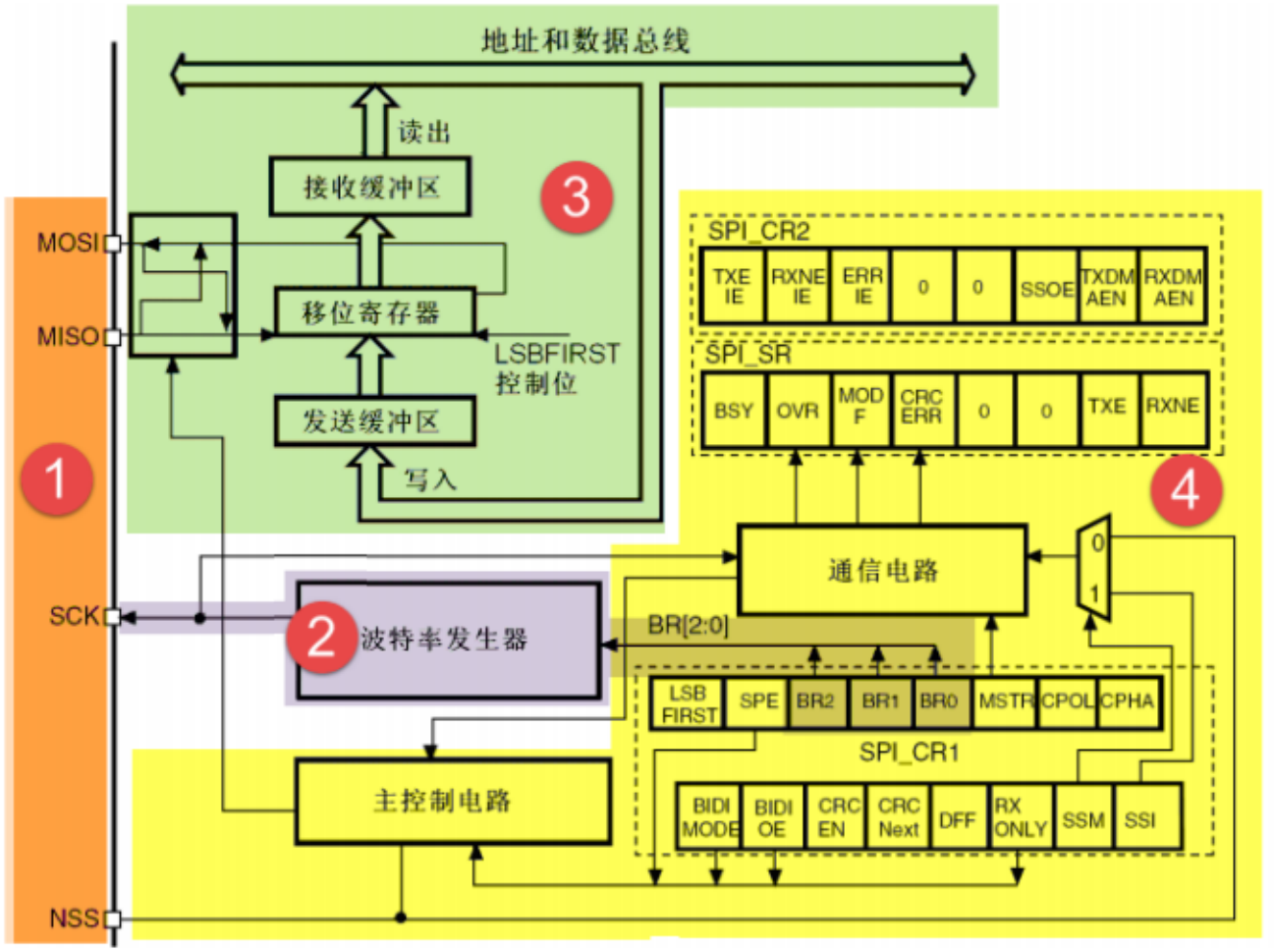


图 24-5 SPI 架构图

23.2.2.1 通讯引脚

SPI 的所有硬件架构都从图 24-5 中左侧 MOSI、MISO、SCK 及 NSS 线展开的。STM32 芯片有多个 SPI 外设，它们的 SPI 通讯信号引出到不同的 GPIO 引脚上，使用时必须配置到这些指定的引脚，见表 24-2。关于 GPIO 引脚的复用功能，可查阅《STM32F10x 规格书》，以它为准。

表 24-2 STM32F10x 的 SPI 引脚 (整理自《STM32F10x 规格书》)

引脚	SPI 编号		
	SPI1	SPI2	SPI3
NSS	PA4	PB12	PA15 下载口的 TDI
CLK	PA5	PB13	PB3 下载口的 TDO
MISO	PA6	PB14	PB4 下载口的 NTRST
MOSI	PA7	PB15	PB5

其中 SPI1 是 APB2 上的设备，最高通信速率达 36Mbtis/s，SPI2、SPI3 是 APB1 上的设备，最高通信速率为 18Mbits/s。除了通讯速率，在其它功能上没有差异。其中 SPI3 用到了下载接口的引脚，这几个引脚默认功能是下载，第二功能才是 IO 口，如果想使用 SPI3 接口，则程序上必须先禁用掉这几个 IO 口的下载功能。一般在资源不是十分紧张的情况下，这几个 IO 口是专门用于下载和调试程序，不会复用为 SPI3。

23.2.2.2 时钟控制逻辑

SCK 线的时钟信号，由波特率发生器根据“控制寄存器 *CR1*”中的 *BR[0:2]* 位控制，该位是对 f_{pclk} 时钟的分频因子，对 f_{pclk} 的分频结果就是 SCK 引脚的输出时钟频率，计算方法见表 24-3。

表 24-3 BR 位对 fpclk 的分频

BR[0:2]	分频结果 (SCK 频率)	BR[0:2]	分频结果 (SCK 频率)		
000	f	$_{pclk}$/2	100	f	$_{pclk}$/32
001	f	$_{pclk}$/4	101	f	$_{pclk}$/64
010	f	$_{pclk}$/8	110	f	$_{pclk}$/128
011	f	$_{pclk}$/16	111	f	$_{pclk}$/256

其中的 f_{pclk} 频率是指 SPI 所在的 APB 总线频率，APB1 为 f_{pclk1}，APB2 为 f_{pckl2}。

通过配置“控制寄存器 CR”的“CPOL 位”及“CPHA”位可以把 SPI 设置成前面分析的4 种 SPI 模式。

23.2.2.3 数据控制逻辑

SPI 的 MOSI 及 MISO 都连接到数据移位寄存器上，数据移位寄存器的数据来源及目标接收、发送缓冲区以及 MISO、MOSI 线。当向外发送数据的时候，数据移位寄存器以“发送缓冲区”为数据源，把数据一位一位地通过数据线发送出去；当从外部接收数据的时候，数据移位寄存器把数据线采样到的数据一位一位地存储到“接收缓冲区”中。通过写 SPI 的“数据寄存器 DR”把数据填充到发送缓冲区中，通讯读“数据寄存器 DR”，可以获取接收缓冲区中的内容。其中数据帧长度可以通过“控制寄存器 CR1”的“DFF 位”配置成 8 位及 16 位模式；配置“LSBFIRST 位”可选择 MSB 先行还是 LSB 先行。

23.2.2.4 整体控制逻辑

整体控制逻辑负责协调整个 SPI 外设，控制逻辑的工作模式根据我们配置的“控制寄存器 (CR1/CR2)”的参数而改变，基本的控制参数包括前面提到的 SPI 模式、波特率、LSB 先行、主从模式、单双向模式等等。在外设工作时，控制逻辑会根据外设的工作状态修改“状态寄存器 (SR)”，我们只要读取状态寄存器相关的寄存器位，就可以了解 SPI 的工作状态了。除此之外，控制逻辑还根据要求，负责控制产生 SPI 中断信号、DMA 请求及控制 NSS 信号线。

实际应用中，我们一般不使用 STM32 SPI 外设的标准 NSS 信号线，而是更简单地使用普通的 GPIO，软件控制它的电平输出，从而产生通讯起始和停止信号。

23.2.3 通讯过程

STM32 使用 SPI 外设通讯时，在通讯的不同阶段它会对“状态寄存器 SR”的不同数据位写入参数，我们通过读取这些寄存器标志来了解通讯状态。

图 *24_6* 中的是“主模式”流程，即 STM32 作为 SPI 通讯的主机端时的数据收发过程。

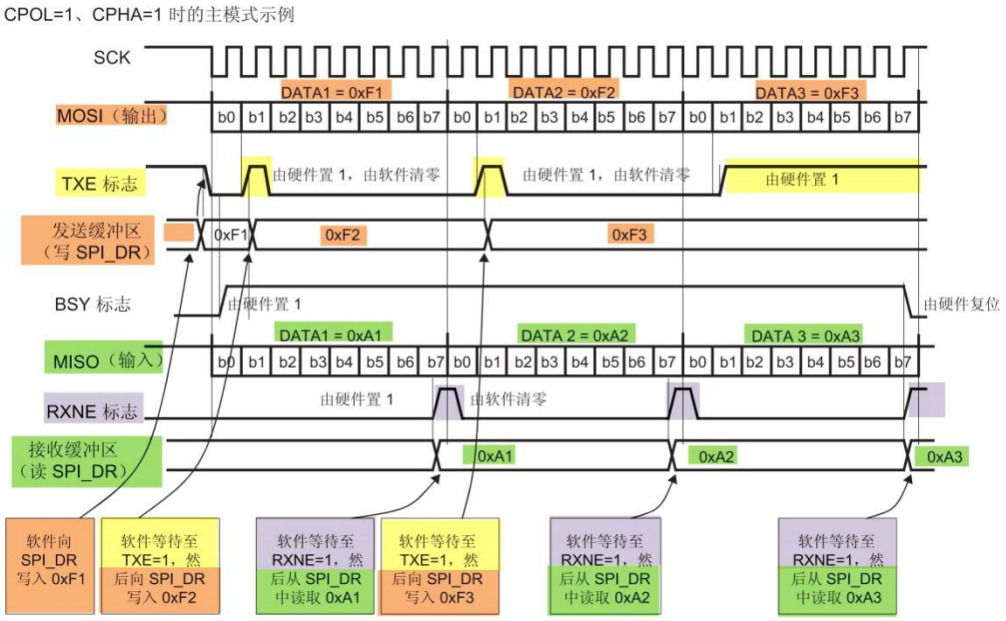


图 24-6 主发送器通讯过程

主模式收发流程及事件说明如下：

(1) 控制 NSS 信号线，产生起始信号 (图中没有画出)；

(2) 把要发送的数据写入到“数据寄存器 DR”中，该数据会被存储到发送缓冲区；

(3) 通讯开始，SCK 时钟开始运行。MOSI 把发送缓冲区中的数据一位一位地传输出去；MISO 则把数据一位一位地存储进接收缓冲区中；

(4) 当发送完一帧数据的时候，“状态寄存器 SR”中的“TXE 标志位”会被置 1，表示传输完一帧，发送缓冲区已空；类似地，当接收完一帧数据的时候，“RXNE 标志位”会被置 1，表示传输完一帧，接收缓冲区非空；

(5) 等待到“TXE 标志位”为 1 时，若还要继续发送数据，则再次往“数据寄存器 DR”写入数据

即可；等待到“RXNE 标志位”为 1 时，通过读取“数据寄存器 DR”可以获取接收缓冲区中的内容。

假如我们使能了 TXE 或 RXNE 中断，TXE 或 RXNE 置 1 时会产生 SPI 中断信号，进入同一个中断服务函数，到 SPI 中断服务程序后，可通过检查寄存器位来了解是哪一个事件，再分别进行处理。也可以使用 DMA 方式来收发“数据寄存器 DR”中的数据。

23.3 SPI 初始化结构体详解

跟其它外设一样，STM32HAL 库提供了 SPI 初始化结构体及初始化函数来配置 SPI 外设。初始化结构体及函数定义在库文件“stm32f4xx_hal_spi.h”中，编程时我们可以结合这两个文件内的注释使用或参考库帮助文档。了解初始化结构体后我们就能对 SPI 外设运用自如了，见*代码清单 24_1*。

列表 1: 代码清单 24-1 SPI 初始化结构体

```
typedef struct {
    uint32_t Mode;       /* 设置 SPI 的主/从机端模式 */
    uint32_t Direction; /* 设置 SPI 的单双向模式 */
    uint32_t DataSize;  /* 设置 SPI 的数据帧长度，可选 8/16 位 */
    uint32_t CLKPolarity;/* 设置时钟极性 CPOL，可选高/低电平 */
    uint32_t CLKPhase; /* 设置时钟相位，可选奇/偶数边沿采样 */
    uint32_t NSS;        /* 设置 NSS 引脚由 SPI 硬件控制还是软件控制 */
    uint32_t BaudRatePrescaler; /* 设置时钟分频因子，fpclk/分频数 =fSCK */
    uint32_t FirstBit; /* 设置 MSB/LSB 先行 */
    uint32_t TIMode;    /* 指定是否启用 TI 模式 */
    uint32_t CRCCalculation; /* 指定是否启用 CRC 计算 */
    uint32_t CRCPolynomial;  /* 设置 CRC 校验的表达式 */
} SPI_InitTypeDef;
```

这些结构体成员说明如下，其中括号内的文字是对应参数在 STM32 标准库中定义的宏：

(1) *Mode*：本成员设置 SPI 工作在主机模式 *(SPI_MODE_MASTER)* 或从机模式 *(SPI_MODE*

_SLAVE)，这两个模式的最大区别为 SPI 的 SCK 信号线的时序，*SCK* 的时序是由通讯中的主机产生的。若被配置为从机模式，STM32 的 SPI 外设将接受外来的 SCK 信号。

(2) *Direction*：本成员设置 *SPI* 的通讯方向，可设置为双线全双工 *(SPI_DIRECTION_2LINES)*，双线只接收 *(SPI_DIRECTION_2LINES_RXONLY)*，单线 *SPI_DIRECTION_1LINE*。

(3) *DataSize*：本成员可以选择 SPI 通讯的数据帧大小是为 8 位 (SPI_DATASIZE_8BIT) 还是 16 位 (SPI_DATASIZE_16BIT)。

(4) CLKPolarity 和 CLKPhase：这两个成员配置 SPI 的时钟极性 CLKPolarity 和时钟相位 CLKPhase，这两个配置影响到 SPI 的通讯模式，关于 CLKPolarity 和 CLKPhase 的说明参考前面“通讯模式”小节。时钟极性 CLKPolarity 成员，可设置为高电平 (SPI_POLARITY_HIGH) 或低电平 (SPI_POLARITY_LOW)。时钟相位 CPHA 则可以设置为 SPI_PHASE_1EDGE(在 SCK 的奇数边沿采集数据) 或 SPI_P HASE_2EDGE(在 SCK 的偶数边沿采集数据) 。

(5) *NSS*：本成员配置 NSS 引脚的使用模式，可以选择为硬件模式 *(SPI_NSS_HARD)* 与软件模式 *(SPI_NSS_SOFT)*，在硬件模式中的 SPI 片选信号由 SPI 硬件自动产生，而软件模式则需要我们亲自把相应的 GPIO 端口拉高或置低产生非片选和片选信号。实际中软件模式应用比较多。

(6) *BaudRatePrescaler*：本成员设置波特率分频因子，分频后的时钟即为 SPI 的 SCK 信号线的时钟频率。这个成员参数可设置为 fpclk 的 2、4、6、8、16、32、64、128、256 分频。

(7) *FirstBit*：所有串行的通讯协议都会有 *MSB* 先行 *(高位数据在前)* 还是 *LSB* 先行 *(低位数据在前)* 的问题，而 STM32 的 SPI 模块可以通过这个结构体成员，对这个特性编程控制。

(8) TIMode ： 指定是否启用 TI 模式。可选择为使能 *(SPI_TIMO DE_ENABLE)* 与不是能 *(SPI_TIMODE_DISABLE)*。

(9) CRCCalculation ： 指定是否启用 CRC 计算

(10) *SPI_CRCPolynomial*：这是 SPI 的 *CRC* 校验中的多项式，若我们使用 CRC 校验时，就使用这个成员的参数 (多项式)，来计算 CRC 的值。

配置完这些结构体成员后，我们要调用 *HAL_SPI_Init* 函数把这些参数写入到寄存器中，实现 SPI 的初始化，然后调用 *__HAL_SPI_ENABLE* 来使能 SPI 外设。

23.4 SPI—读写串行 FLASH 实验

FLSAH 存储器又称闪存，它与 EEPROM 都是掉电后数据不丢失的存储器，但 FLASH 存储器容量普遍大于 EEPROM，现在基本取代了它的地位。我们生活中常用的 U 盘、SD 卡、SSD 固态硬盘以及我们 STM32 芯片内部用于存储程序的设备，都是 FLASH 类型的存储器。在存储控制上，最主要的区别是 FLASH 芯片只能一大片一大片地擦写，而在“I2C 章节”中我们了解到 EEPROM 可以单个字节擦写。

本小节以一种使用 SPI 通讯的串行 FLASH 存储芯片的读写实验为大家讲解 STM32 的 SPI 使用方法。实验中 STM32 的 SPI 外设采用主模式，通过查询事件的方式来确保正常通讯。

23.4.1 硬件设计

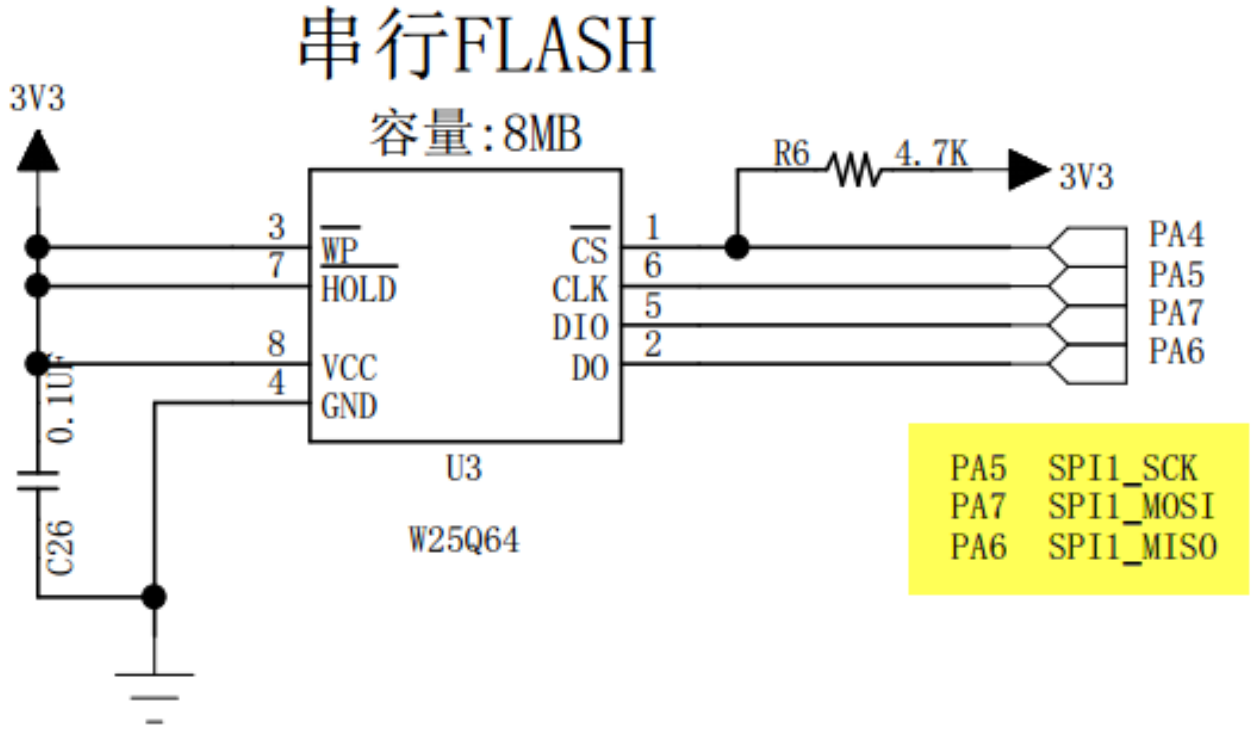


图 24-7a 指南者开发板：SPI 串行 FLASH 硬件连接图

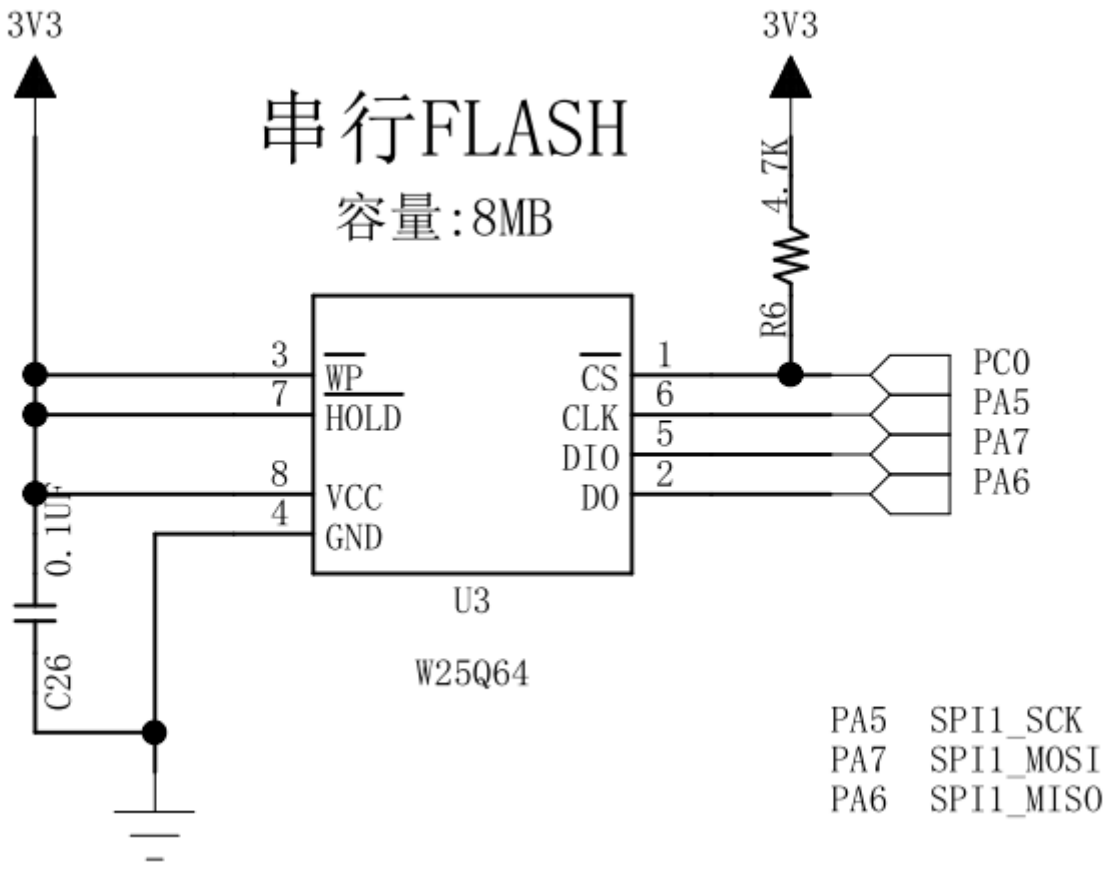


图 24-7b MINI、霸道开发板：SPI 串行 FLASH 硬件连接图

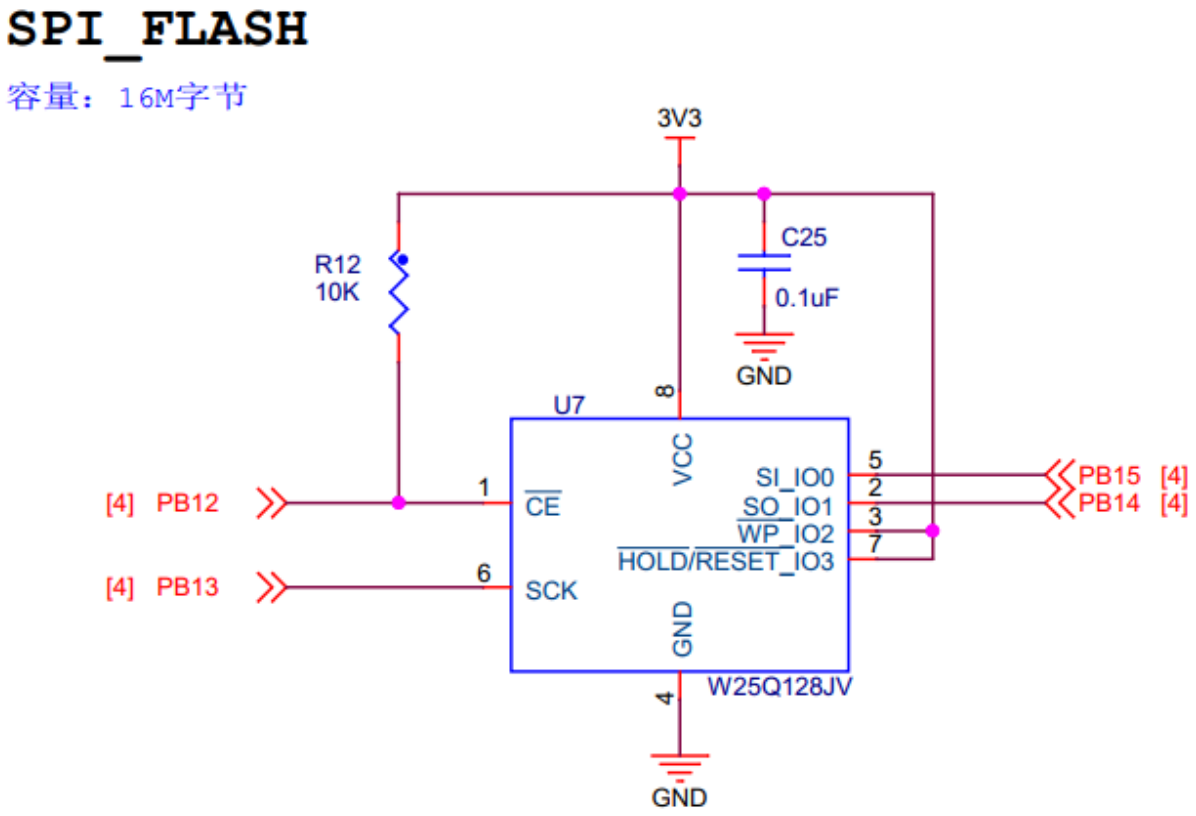


图 24-7c 拂晓电机开发板：SPI 串行 FLASH 硬件连接图

本实验板中的 FLASH 芯片 (型号：W25Q64) 是一种使用 SPI 通讯协议的 NOR FLASH 存储器，它的 CS/CLK/DIO/DO 引脚分别连接到了 STM32 对应的 SPI 引脚 NSS/SCK/MOSI/MISO 上，其中 STM32 的 NSS 引脚虽然是其片上 SPI 外设的硬件引脚，但实际上后面的程序只是把它当成一个普通的 GPIO，使用软件的方式控制 NSS 信号，所以在 SPI 的硬件设计中，NSS 可以随便选择普通的 GPIO，不必纠结于选择硬件 NSS 信号。

FLASH 芯片中还有 WP 和 HOLD 引脚。WP 引脚可控制写保护功能，当该引脚为低电平时，禁止写入数据。我们直接接电源，不使用写保护功能。HOLD 引脚可用于暂停通讯，该引脚为低电平时，通讯暂停，数据输出引脚输出高阻抗状态，时钟和数据输入引脚无效。我们直接接电源，不使用通讯暂停功能。

关于 FLASH 芯片的更多信息，可参考其数据手册《W25Q64》来了解。若您使用的实验板 FLASH 的型号或控制引脚不一样，只需根据我们的工程修改即可，程序的控制原理相同。

23.4.2 软件设计

为了使工程更加有条理，我们把读写 FLASH 相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_spi_flash.c”及“bsp_spi_flash.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

23.4.2.1 编程要点

(1) 初始化通讯使用的目标引脚及端口时钟；

(2) 使能 SPI 外设的时钟；

(3) 配置 SPI 外设的模式、地址、速率等参数并使能 SPI 外设；

(4) 编写基本 SPI 按字节收发的函数；

(5) 编写对 FLASH 擦除及读写操作的的函数；

(6) 编写测试程序，对读写数据进行校验。

23.4.2.2 代码分析

23.4.2.2.1 SPI 硬件相关宏定义

我们把 SPI 硬件相关的配置都以宏的形式定义到“bsp_spi_flash.h”文件中，见*代码清单 24_2a*、*代码清单 24_2b*、*代码清单 24_2c*。

列表 2: 代码清单 24-2a MINI、霸道开发板 SPI 硬件配置相关的宏

```
#define SPIx                             SPI1
#define SPIx_CLK_ENABLE()                __HAL_RCC_SPI1_CLK_ENABLE()
#define SPIx_SCK_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOA_CLK_ENABLE()
#define SPIx_MISO_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOA_CLK_ENABLE()
#define SPIx_MOSI_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOA_CLK_ENABLE()
```

(下页继续)

(续上页)

```
  #define SPIx_CS_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOC_CLK_ENABLE()

  #define SPIx_FORCE_RESET()              __HAL_RCC_SPI1_FORCE_RESET()
  #define SPIx_RELEASE_RESET()            __HAL_RCC_SPI1_RELEASE_RESET()

  /* Definition for SPIx Pins */
  #define SPIx_SCK_PIN                    GPIO_PIN_5
  #define SPIx_SCK_GPIO_PORT              GPIOA

  #define SPIx_MISO_PIN                   GPIO_PIN_6
  #define SPIx_MISO_GPIO_PORT             GPIOA

  #define SPIx_MOSI_PIN                   GPIO_PIN_7
  #define SPIx_MOSI_GPIO_PORT             GPIOA

  #define FLASH_CS_PIN                    GPIO_PIN_4
  #define FLASH_CS_GPIO_PORT              GPIOA

  #define digitalHi(p,i)          {p->BSRR=i;}                    //设置为高电
平
  #define digitalLo(p,i)          {p->BSRR=(uint32_t)i << 16;}    //输出低电平

  #define SPI_FLASH_CS_LOW()      digitalLo(FLASH_CS_GPIO_PORT, FLASH_CS_PIN␣
↪)
  #define SPI_FLASH_CS_HIGH()     digitalHi(FLASH_CS_GPIO_PORT, FLASH_CS_PIN␣
↪)
```

列表 3: 代码清单 24-2b 指南者开发板 SPI 硬件配置相关的宏

```
#define SPIx                            SPI1
#define SPIx_CLK_ENABLE()               __HAL_RCC_SPI1_CLK_ENABLE()
```

(下页继续)

(续上页)

```
#define SPIx_SCK_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define SPIx_MISO_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOA_CLK_ENABLE()
#define SPIx_MOSI_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOA_CLK_ENABLE()
#define SPIx_CS_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()

#define SPIx_FORCE_RESET()                __HAL_RCC_SPI1_FORCE_RESET()
#define SPIx_RELEASE_RESET()              __HAL_RCC_SPI1_RELEASE_RESET()

/* Definition for SPIx Pins */
#define SPIx_SCK_PIN                      GPIO_PIN_5
#define SPIx_SCK_GPIO_PORT                GPIOA

#define SPIx_MISO_PIN                     GPIO_PIN_6
#define SPIx_MISO_GPIO_PORT               GPIOA

#define SPIx_MOSI_PIN                     GPIO_PIN_7
#define SPIx_MOSI_GPIO_PORT               GPIOA

#define FLASH_CS_PIN                      GPIO_PIN_0
#define FLASH_CS_GPIO_PORT                GPIOC

#define digitalHi(p,i)           {p->BSRR=i;}                    //设置为高电
平
#define digitalLo(p,i)           {p->BSRR=(uint32_t)i << 16;}    //输出低电平

#define SPI_FLASH_CS_LOW()       digitalLo(FLASH_CS_GPIO_PORT, FLASH_CS_PIN␣
↪)
#define SPI_FLASH_CS_HIGH()      digitalHi(FLASH_CS_GPIO_PORT, FLASH_CS_PIN␣
↪)
```

列表 4: 代码清单 24-2c 拂晓电机开发板 SPI 硬件配置相关的宏

```
#define SPIx                                SPI2
#define SPIx_CLK_ENABLE()                   __HAL_RCC_SPI2_CLK_ENABLE()
#define SPIx_SCK_GPIO_CLK_ENABLE()          __HAL_RCC_GPIOB_CLK_ENABLE()
#define SPIx_MISO_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOB_CLK_ENABLE()
#define SPIx_MOSI_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOB_CLK_ENABLE()
#define SPIx_CS_GPIO_CLK_ENABLE()           __HAL_RCC_GPIOB_CLK_ENABLE()

#define SPIx_FORCE_RESET()                  __HAL_RCC_SPI2_FORCE_RESET()
#define SPIx_RELEASE_RESET()                __HAL_RCC_SPI2_RELEASE_RESET()

/* Definition for SPIx Pins */
#define SPIx_SCK_PIN                        GPIO_PIN_13
#define SPIx_SCK_GPIO_PORT                  GPIOB

#define SPIx_MISO_PIN                       GPIO_PIN_14
#define SPIx_MISO_GPIO_PORT                 GPIOB

#define SPIx_MOSI_PIN                       GPIO_PIN_15
#define SPIx_MOSI_GPIO_PORT                 GPIOB

#define FLASH_CS_PIN                        GPIO_PIN_12
#define FLASH_CS_GPIO_PORT                  GPIOB


#define     digitalHi(p,i)           {p->BSRR=i;}                                ␣
↪ //设置为高电平
#define digitalLo(p,i)           {p->BSRR=(uint32_t)i << 16;}     //输出低电平
#define SPI_FLASH_CS_LOW()       digitalLo(FLASH_CS_GPIO_PORT,FLASH_CS_PIN )
#define SPI_FLASH_CS_HIGH()      digitalHi(FLASH_CS_GPIO_PORT,FLASH_CS_PIN )
```

以上代码根据硬件连接，把与 FLASH 通讯使用的 SPI 号、GPIO 等都以宏封装起来，并且定义了控制 CS(NSS) 引脚输出电平的宏，以便配置产生起始和停止信号时使用。

23.4.2.2.2 初始化 SPI 的 GPIO

利用上面的宏，编写 SPI 的初始化函数，该函数对于所有的开发板都是通用的，见*代码清单 24_3*。

列表 5: 代码清单 24-3 SPI 的初始化函数 (GPIO 初始化部分)

```
void HAL_SPI_MspInit(SPI_HandleTypeDef *hspi)
{
    GPIO_InitTypeDef  GPIO_InitStruct;

    /*##-1- Enable peripherals and GPIO Clocks
    #################################*/
    /* Enable GPIO TX/RX clock */
    SPIx_SCK_GPIO_CLK_ENABLE();
    SPIx_MISO_GPIO_CLK_ENABLE();
    SPIx_MOSI_GPIO_CLK_ENABLE();
    SPIx_CS_GPIO_CLK_ENABLE();
    /* Enable SPI clock */
    SPIx_CLK_ENABLE();

    /*##-2- Configure peripheral GPIO
    ########################################*/
    /* SPI SCK GPIO pin configuration  */
    GPIO_InitStruct.Pin       = SPIx_SCK_PIN;
    GPIO_InitStruct.Mode      = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull      = GPIO_PULLUP;
    GPIO_InitStruct.Speed     = GPIO_SPEED_FREQ_HIGH;

    HAL_GPIO_Init(SPIx_SCK_GPIO_PORT, &GPIO_InitStruct);
```

(下页继续)

(续上页)

```
    /* SPI MISO GPIO pin configuration  */
    GPIO_InitStruct.Pin = SPIx_MISO_PIN;
    HAL_GPIO_Init(SPIx_MISO_GPIO_PORT, &GPIO_InitStruct);

    /* SPI MOSI GPIO pin configuration  */
    GPIO_InitStruct.Pin = SPIx_MOSI_PIN;
    HAL_GPIO_Init(SPIx_MOSI_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = FLASH_CS_PIN ;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    HAL_GPIO_Init(FLASH_CS_GPIO_PORT, &GPIO_InitStruct);
}
```

与所有使用到 GPIO 的外设一样，都要先把使用到的 GPIO 引脚模式初始化，配置好复用功能。GPIO 初始化流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置；

(2) 调用库函数 SPIx_SCK_GPIO_CLK_ENABLE()，SPIx_MISO_GPIO_CLK_ENA BLE() 等完成 SPI 相关引脚的时钟使能。调用库函数 SPIx_CLK_ENABLE() 完成 SPI 外设的使能。

(3) 向 GPIO 初始化结构体赋值，把 SCK/MOSI/MISO 引脚初始化成复用推挽模式。而 CS(NSS) 引脚由于使用软件控制，我们把它配置为普通的推挽输出模式。

(4) 使用以上初始化结构体的配置，调用 HAL_GPIO_Init 函数向分别寄存器写入参数，完成 GPIO 的初始化。

23.4.2.2.3 配置 SPI 的模式

以上只是配置了 SPI 使用的引脚，对 SPI 外设模式的配置。在配置 STM32 的 SPI 模式前，我们要先了解从机端的 SPI 模式。本例子中可通过查阅 FLASH 数据手册《W25Q64》获取。根据 FLASH 芯片的说明，它支持 SPI 模式 0 及模式 3，支持双线全双工，使用 MSB 先行模式，支持最高通讯时钟为 104MHz，数据帧长度为 8 位。我们要把 STM32 的 SPI 外设中的这些参数配置一致。见*代码清单 24_4*。

列表 6: 代码清单 24-4 配置 SPI 模式

```
void SPI_FLASH_Init(void)
{
    SpiHandle.Instance               = SPIx;
    SpiHandle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_4;
    SpiHandle.Init.Direction         = SPI_DIRECTION_2LINES;
    SpiHandle.Init.CLKPhase          = SPI_PHASE_2EDGE;
    SpiHandle.Init.CLKPolarity       = SPI_POLARITY_HIGH;
    SpiHandle.Init.CRCCalculation    = SPI_CRCCALCULATION_DISABLE;
    SpiHandle.Init.CRCPolynomial     = 7;
    SpiHandle.Init.DataSize          = SPI_DATASIZE_8BIT;
    SpiHandle.Init.FirstBit          = SPI_FIRSTBIT_MSB;
    SpiHandle.Init.NSS               = SPI_NSS_SOFT;
    SpiHandle.Init.TIMode            = SPI_TIMODE_DISABLE;

    SpiHandle.Init.Mode = SPI_MODE_MASTER;

    HAL_SPI_Init(&SpiHandle);

    __HAL_SPI_ENABLE(&SpiHandle);
}
```

这段代码中，把 STM32 的 SPI 外设配置为主机端，双线全双工模式，数据帧长度为 8 位，使用 SPI 模式 3(CLKPolarity =1，CLKPhase =1)，NSS 引脚由软件控制以及 MSB 先行模式。代码中把

SPI 的时钟频率配置成了 4 分频，实际上可以配置成 2 分频以提高通讯速率，读者可亲自尝试一下。最后一个成员为 CRC 计算式，由于我们与 FLASH 芯片通讯不需要 CRC 校验，并没有使能 SPI 的 CRC 功能，这时 CRC 计算式的成员值是无效的。

赋值结束后调用库函数 HAL_SPI_Init 把这些配置写入寄存器，并调用 __HAL_SPI_ENABLE 函数使能外设。

23.4.2.2.4 使用 SPI 发送和接收一个字节的数据

初始化好 SPI 外设后，就可以使用 SPI 通讯了，复杂的数据通讯都是由单个字节数据收发组成的，我们看看它的代码实现，见*代码清单 24_5*。

列表 7: 代码清单 24-5 使用 SPI 发送和接收一个字节的数据

```
#define Dummy_Byte 0xFF
/**
* @brief  使用 SPI 发送一个字节的数据
* @param  byte: 要发送的数据
* @retval 返回接收到的数据
*/
u8 SPI_FLASH_SendByte(u8 byte)
{
    SPITimeout = SPIT_FLAG_TIMEOUT;

    /* 等待发送缓冲区为空，TXE 事件 */
    while (__HAL_SPI_GET_FLAG( &SpiHandle, SPI_FLAG_TXE ) == RESET)
    {
        if ((SPITimeout--) == 0) return SPI_TIMEOUT_UserCallback(0);
    }

    /* 写入数据寄存器，把要写入的数据写入发送缓冲区 */
    SPI_I2S_SendData(FLASH_SPI, byte);
```

(下页继续)

(续上页)

```
    SPITimeout = SPIT_FLAG_TIMEOUT;

    /* 等待接收缓冲区非空，RXNE 事件 */
    while (__HAL_SPI_GET_FLAG( &SpiHandle, SPI_FLAG_RXNE ) == RESET)
    {
        if ((SPITimeout--) == 0) return SPI_TIMEOUT_UserCallback(1);
    }

    /* 读取数据寄存器，获取接收缓冲区数据 */
    return READ_REG(SpiHandle.Instance->DR);
}

/**
* @brief  使用 SPI 读取一个字节的数据
* @param  无
* @retval 返回接收到的数据
*/
u8 SPI_FLASH_ReadByte(void)
{
    return (SPI_FLASH_SendByte(Dummy_Byte));
}
```

SPI_FLASH_SendByte 发送单字节函数中包含了等待事件的超时处理，这部分原理跟 I2C 中的一样，在此不再赘述。

SPI_FLASH_SendByte 函数实现了前面讲的“*SPI 通讯过程*”：

(1) 本函数中不包含 SPI 起始和停止信号，只是收发的主要过程，所以在调用本函数前后要做好起始和停止信号的操作；

(2) 对 SPITimeout 变量赋值为宏 SPIT_FLAG_TIMEOUT。这个 SPITimeout 变量在下面的 while 循中每次循环减 1，该循环通过调用库函数 SPI_I2S_GetFlagStatus 检测事件，若检测到事件，则进入通讯的下一阶段，若未检测到事件则停留在此处一直

检测，当检测 SPIT_FLAG_TIMEOUT 次都还没等待到事件则认为通讯失败，调用的 SPI_TIMEOUT_UserCallback 输出调试信息，并退出通讯；

(3) 通过检测 TXE 标志，获取发送缓冲区的状态，若发送缓冲区为空，则表示可能存在的上一个数据已经发送完毕；

(4) 等待至发送缓冲区为空后，调用库函数 SPI_I2S_SendData 把要发送的数据“byte”写入到 SPI 的数据寄存器 DR，写入 SPI 数据寄存器的数据会存储到发送缓冲区，由 SPI 外设发送出去；

(5) 写入完毕后等待 RXNE 事件，即接收缓冲区非空事件。由于 SPI 双线全双工模式下 MOSI 与 MISO 数据传输是同步的 (请对比“*SPI 通讯过程*”阅读)，当接收缓冲区非空时，表示上面的数据发送完毕，且接收缓冲区也收到新的数据；

(6) 等待至接收缓冲区非空时，通过调用库函数 SPI_I2S_ReceiveData 读取 SPI 的数据寄存器 DR，就可以获取接收缓冲区中的新数据了。代码中使用关键字“return”把接收到的这个数据作为 SPI_FLASH_SendByte 函数的返回值，所以我们可以看到在下面定义的 SPI 接收数据函数 SPI_FLASH_ReadByte，它只是简单地调用了 SPI_FLASH_SendByte 函数发送数据“Dummy_Byte”，然后获取其返回值 (因为不关注发送的数据，所以此时的输入参数“Dummy_Byte”可以为任意值)。可以这样做的原因是 SPI 的接收过程和发送过程实质是一样的，收发同步进行，关键在于我们的上层应用中，关注的是发送还是接收的数据。

23.4.2.2.5 控制 FLASH 的指令

搞定 SPI 的基本收发单元后，还需要了解如何对 FLASH 芯片进行读写。FLASH 芯片自定义了很多指令，我们通过控制 STM32 利用 SPI 总线向 FLASH 芯片发送指令，FLASH 芯片收到后就会执行相应的操作。

而这些指令，对主机端 (STM32) 来说，只是它遵守最基本的 SPI 通讯协议发送出的数据，但在设备端 (FLASH 芯片) 把这些数据解释成不同的意义，所以才成为指令。查看 FLASH 芯片的数据手册《W25Q64》，可了解各种它定义的各种指令的功能及指令格式，见表 24-4。

表 24-4 FLASH 常用芯片指令表 (摘自规格书《W25Q64》)

指令	第一字节 (指令编码)	第二字节	第三字节	第四字节	第五字节	第六字节	第七-N 字节
Write Enable	06h						
Write Disable	04h						
Read Status Register	05h	(S7–S0)					
Write Status Register	01h	(S7–S0)					
Read Data	03h	A23–A16	A15–A8	A7–A0	(D7–D0)	(Next byte)	continuous
Fast Read	0Bh	A23–A16	A15–A8	A7–A0	dummy	(D7–D0)	(Next Byte) continuous
Fast Read Dual Output	3Bh	A23–A16	A15–A8	A7–A0	dummy	I/O = (D6,D4,D2,D0) O = (D7,D5,D3,D1)	(one byte per 4 clocks, continuous)
Page Program	02h	A23–A16	A15–A8	A7–A0	D7–D0	Next byte	Up to 256 bytes
Block Erase(64KB)	D8h	A23–A16	A15–A8	A7–A0			
Sector Erase(4KB)	20h	A23–A16	A15–A8	A7–A0			
Chip Erase	C7h						
Power-down	B9h						
Release Power- down / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0)		
Manufac-turer/ Device ID	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)	
JEDEC ID	9Fh	(M7-M0) 生产厂商	(ID15-ID8) 存储器类型	(ID7-ID0) 容量			

该表中的第一列为指令名，第二列为指令编码，第三至第 N 列的具体内容根据指令的不同而有不同的含义。其中带括号的字节参数，方向为 FLASH 向主机传输，即命令响应，不带括号的则为主机向 FLASH 传输。表中“*A0~A23*”指 FLASH 芯片内部存储器组织的地址；“*M0~M7*”为厂商号（MANUFACTURER ID）；“ID0-ID15”为 FLASH 芯片的 ID；“dummy”指该处可为任意数据；“*D0~D7*”为 FLASH 内部存储矩阵的内容。

在 FLSAH 芯片内部，存储有固定的厂商编号 (M7-M0) 和不同类型 FLASH 芯片独有的编号 (ID15-ID0)，见表 24-5。

表 24-5 FLASH 数据手册的设备 ID 说明

FLASH 型号	厂商号 (M7-M0)	FLASH 型号 (ID15-ID0)
W25Q64	EF h	4017 h
W25Q128	EF h	4018 h

通过指令表中的读 ID 指令“JEDEC ID”可以获取这两个编号，该指令编码为“9F h”，其中“9F h”是指 16 进制数“9F”(相当于 C 语言中的 0x9F)。紧跟指令编码的三个字节分别为 FLASH 芯片输出的“(M7-M0)”、“(ID15-ID8)”及“(ID7-ID0)”。

此处我们以该指令为例，配合其指令时序图进行讲解，见图 *24_8*。

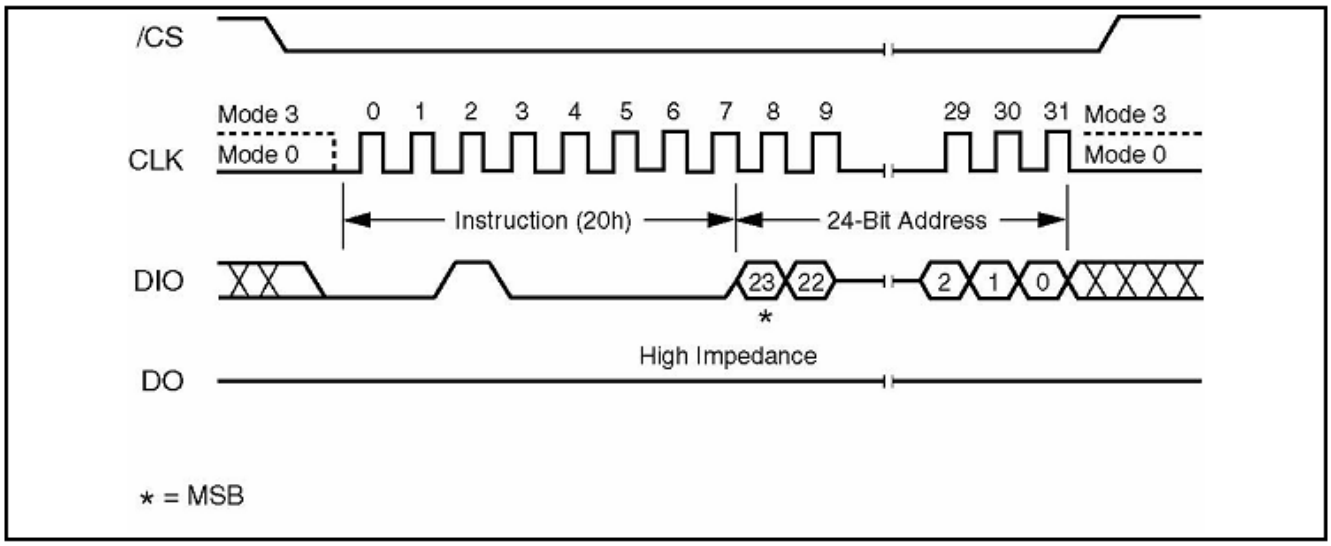


图 24-8 FLASH 读 ID 指令“JEDEC ID”的时序 (摘自规格书《W25Q64》)

主机首先通过 *MOSI* 线向 *FLASH* 芯片发送第一个字节数据为“*9Fh*”，当 FLASH 芯片收到该数据后，它会解读成主机向它发送了“JEDEC 指令”，然后它就作出该命令的响应：通过 *MISO* 线把它的厂商 ID(M7-M0) 及芯片类型 (ID15-0) 发送给主机，主机接收到指令响应后可进行校验。常见的应用是主机端通过读取设备 ID 来测试硬件是否连接正常，或用于识别设备。

对于 FLASH 芯片的其它指令，都是类似的，只是有的指令包含多个字节，或者响应包含更多的数据。

实际上，编写设备驱动都是有一定的规律可循的。首先我们要确定设备使用的是什么通讯协议。如上一章的 EEPROM 使用的是 I^2C，本章的 FLASH 使用的是 SPI。那么我们就先根据它的通讯协议，选择好 STM32 的硬件模块，并进行相应的 I^2C 或 SPI 模块初始化。接着，我们要了解目标设备的相关指令，因为不同的设备，都会有相应的不同的指令。如 EEPROM 中会把第一个数据解释为内部存储矩阵的地址 (实质就是指令)。而 FLASH 则定义了更多的指令，有写指令，读指令，读 ID 指令等等。最后，我们根据这些指令的格式要求，使用通讯协议向设备发送指令，达到控制设备的目标。

23.4.2.2.6 定义 FLASH 指令编码表

为了方便使用，我们把 FLASH 芯片的常用指令编码使用宏来封装起来，后面需要发送指令编码的时候我们直接使用这些宏即可，见代码清单 24_6。

列表 8: 代码清单 24-6 FLASH 指令编码表

```
 /*FLASH 常用命令 */
#define W25X_WriteEnable 0x06
#define W25X_WriteDisable 0x04
#define W25X_ReadStatusReg 0x05
#define W25X_WriteStatusReg 0x01
#define W25X_ReadData 0x03
#define W25X_FastReadData 0x0B
#define W25X_FastReadDual 0x3B
#define W25X_PageProgram 0x02
#define W25X_BlockErase 0xD8
```

(下页继续)

(续上页)

```
#define W25X_SectorErase 0x20
#define W25X_ChipErase 0xC7
#define W25X_PowerDown 0xB9
#define W25X_ReleasePowerDown 0xAB
#define W25X_DeviceID 0xAB
#define W25X_ManufactDeviceID 0x90
#define W25X_JedecDeviceID 0x9F
/* 其它 */
#define sFLASH_ID 0XEF4017
#define Dummy_Byte 0xFF
```

23.4.2.2.7 读取 FLASH 芯片 ID

根据“JEDEC”指令的时序，我们把读取 FLASH ID 的过程编写成一个函数，见代码清单 24_7。

列表 9: 代码清单 24-7 读取 FLASH 芯片 ID

```
/**
* @brief  读取 FLASH ID
* @param  无
* @retval FLASH ID
*/
u32 SPI_FLASH_ReadID(void)
{
    u32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;

    /* 开始通讯: CS 低电平 */
    SPI_FLASH_CS_LOW();

    /* 发送 JEDEC 指令，读取 ID */
    SPI_FLASH_SendByte(W25X_JedecDeviceID);
```

(下页继续)

(续上页)

```
    /* 读取一个字节数据 */
    Temp0 = SPI_FLASH_SendByte(Dummy_Byte);

    /* 读取一个字节数据 */
    Temp1 = SPI_FLASH_SendByte(Dummy_Byte);

    /* 读取一个字节数据 */
    Temp2 = SPI_FLASH_SendByte(Dummy_Byte);

    /* 停止通讯: CS 高电平 */
    SPI_FLASH_CS_HIGH();

    /* 把数据组合起来，作为函数的返回值 */
    Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;

    return Temp;
}
```

这段代码利用控制 CS 引脚电平的宏“SPI_FLASH_CS_LOW/HIGH”以及前面编写的单字节收发函数 SPI_FLASH_SendByte，很清晰地实现了“JEDEC ID”指令的时序：发送一个字节的指令编码“W25X_JedecDeviceID”，然后读取 3 个字节，获取 FLASH 芯片对该指令的响应，最后把读取到的这 3 个数据合并到一个变量 Temp 中，然后作为函数返回值，把该返回值与我们定义的宏“sFLASH_ID”对比，即可知道 FLASH 芯片是否正常。

23.4.2.2.8 FLASH 写使能以及读取当前状态

在向 FLASH 芯片存储矩阵写入数据前，首先要使能写操作，通过“Write Enable”命令即可写使能，见*代码清单 24_8*。

列表 10: 代码清单 24-8 写使能命令

```
/**
* @brief  向 FLASH 发送 写使能 命令
* @param  none
* @retval none
*/
void SPI_FLASH_WriteEnable(void)
{
    /* 通讯开始: CS 低 */
    SPI_FLASH_CS_LOW();

    /* 发送写使能命令 */
    SPI_FLASH_SendByte(W25X_WriteEnable);

    /* 通讯结束: CS 高 */
    SPI_FLASH_CS_HIGH();
}
```

与 EEPROM 一样，由于 FLASH 芯片向内部存储矩阵写入数据需要消耗一定的时间，并不是在总线通讯结束的一瞬间完成的，所以在写操作后需要确认 FLASH 芯片“空闲”时才能进行再次写入。为了表示自己的工作状态，FLASH 芯片定义了一个状态寄存器，见图 *24_9*。

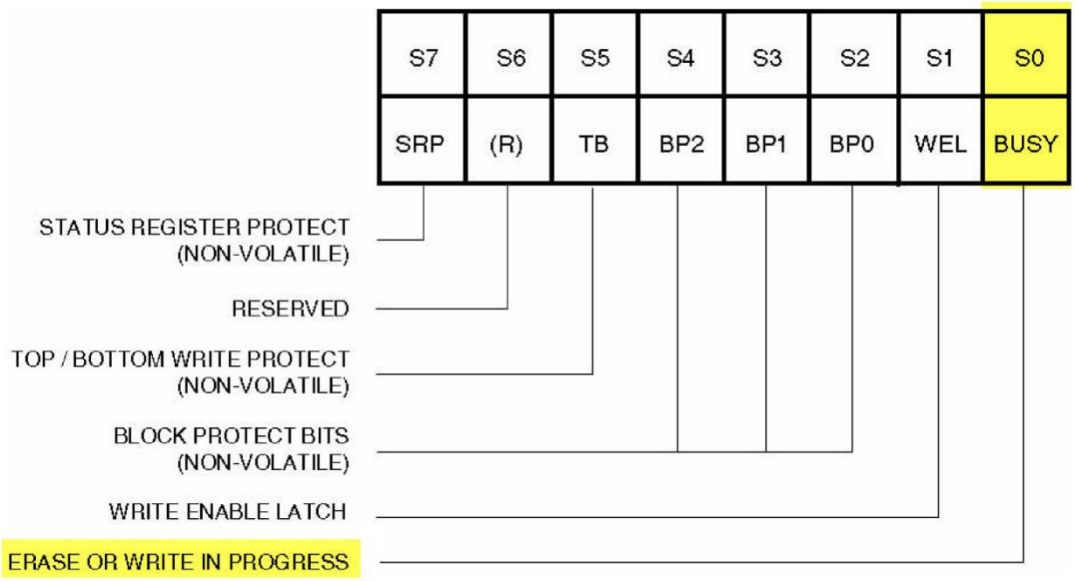


图 24-9 FLASH 芯片的状态寄存器

我们只关注这个状态寄存器的第 0 位“BUSY”，当这个位为“1”时，表明 FLASH 芯片处于忙碌状态，它可能正在对内部的存储矩阵进行“擦除”或“数据写入”的操作。

利用指令表中的“Read Status Register”指令可以获取 FLASH 芯片状态寄存器的内容，其时序见图 *24_10*。

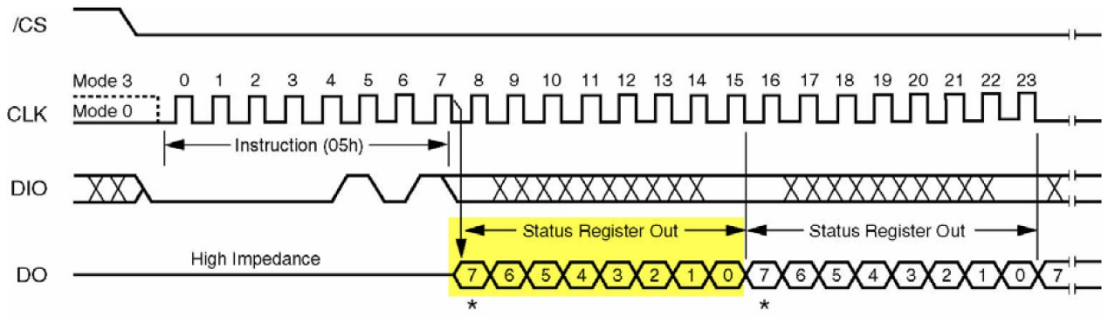


图 24-10 读取状态寄存器的时序

只要向 FLASH 芯片发送了读状态寄存器的指令，FLASH 芯片就会持续向主机返回最新的状态寄存器内容，直到收到 SPI 通讯的停止信号。据此我们编写了具有等待 FLASH 芯片写入结束功能的函数，见*代码清单 24_9*。

列表 11: 代码清单 24-9 通过读状态寄存器等待 FLASH 芯片空闲

```
/* WIP(busy) 标志，FLASH 内部正在写入 */
#define WIP_Flag                  0x01

/**
* @brief  等待 WIP(BUSY) 标志被置 0，即等待到 FLASH 内部数据写入完毕
* @param  none
* @retval none
*/
void SPI_FLASH_WaitForWriteEnd(void)
{
    u8 FLASH_Status = 0;

    /* 选择 FLASH: CS 低 */
    SPI_FLASH_CS_LOW();

    /* 发送 读状态寄存器 命令 */
    SPI_FLASH_SendByte(W25X_ReadStatusReg);

    /* 若 FLASH 忙碌，则等待 */
    do
    {
        /* 读取 FLASH 芯片的状态寄存器 */
        FLASH_Status = SPI_FLASH_SendByte(Dummy_Byte);
    }
    while ((FLASH_Status & WIP_Flag) == SET);  /* 正在写入标志 */

    /* 停止信号  FLASH: CS 高 */
    SPI_FLASH_CS_HIGH();
}
```

这段代码发送读状态寄存器的指令编码“W25X_ReadStatusReg”后，在 while 循环里持续获取寄存器的内容并检验它的“WIP_Flag 标志”(即 BUSY 位)，一直等待到该标志表示写入结束时才退出本函数，以便继续后面与 FLASH 芯片的数据通讯。

23.4.2.2.9 FLASH 扇区擦除

由于 FLASH 存储器的特性决定了它只能把原来为“1”的数据位改写成“0”，而原来为“0”的数据位不能直接改写为“1”。所以这里涉及到数据“擦除”的概念，在写入前，必须要对目标存储矩阵进行擦除操作，把矩阵中的数据位擦除为“1”，在数据写入的时候，如果要存储数据“1”，那就不修改存储矩阵，在要存储数据“0”时，才更改该位。

通常，对存储矩阵擦除的基本操作单位都是多个字节进行，如本例子中的 FLASH 芯片支持“扇区擦除”、“块擦除”以及“整片擦除”，见表 24-6。

表 24-6 本实验 FLASH 芯片的擦除单位

擦除单位	大小
扇区擦除 Sector Erase	4KB
块擦除 Block Erase	64KB
整片擦除 Chip Erase	整个芯片完全擦除

FLASH 芯片的最小擦除单位为扇区 (Sector)，而一个块 (Block) 包含 16 个扇区，其内部存储矩阵分布见图 *24_11*。

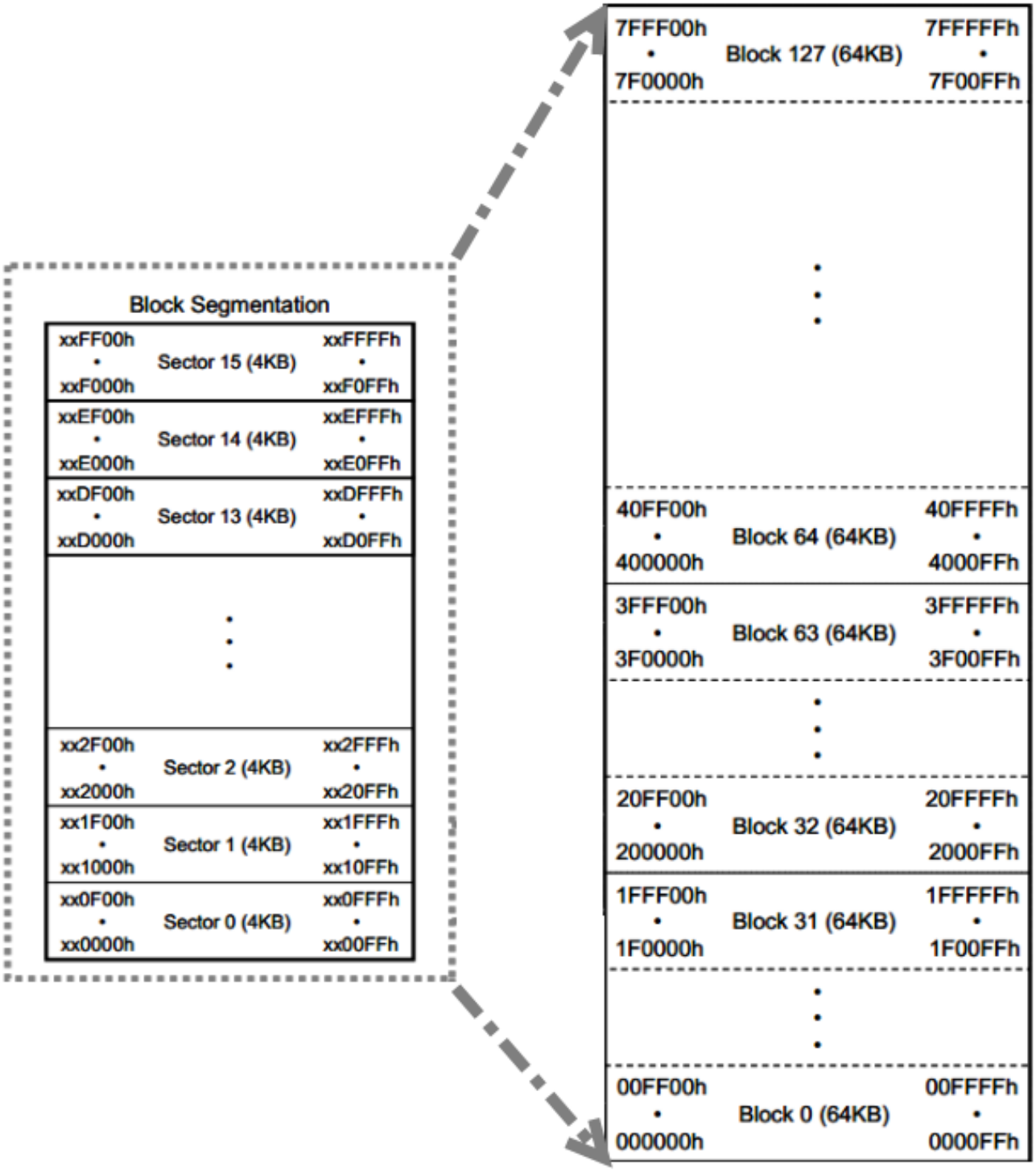


图 24-11 FLASH 芯片的存储矩阵

使用扇区擦除指令“Sector Erase”可控制 FLASH 芯片开始擦写，其指令时序见图 *24_12*。

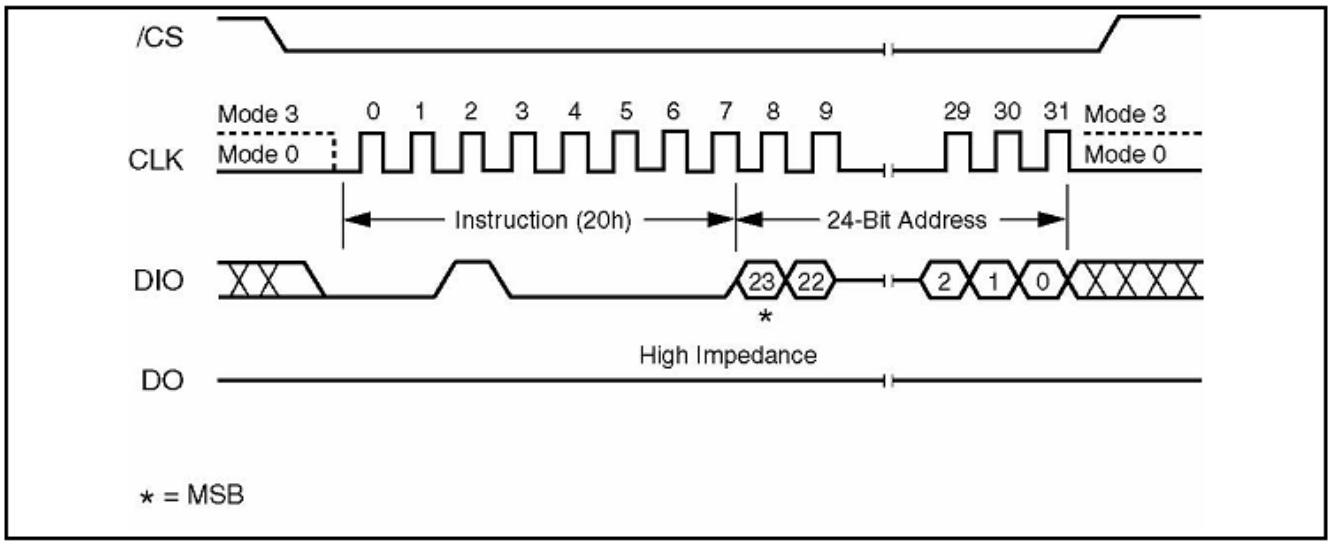


图 24-12 扇区擦除时序

扇区擦除指令的第一个字节为指令编码，紧接着发送的 3 个字节用于表示要擦除的存储矩阵地址。要注意的是在扇区擦除指令前，还需要先发送“写使能”指令，发送扇区擦除指令后，通过读取寄存器状态等待扇区擦除操作完毕，代码实现见*代码清单 24_10*。

列表 12: 代码清单 24-10 擦除扇区

```
/**
* @brief  擦除 FLASH 扇区
* @param  SectorAddr: 要擦除的扇区地址
* @retval 无
*/
void SPI_FLASH_SectorErase(u32 SectorAddr)
{
    /* 发送 FLASH 写使能命令 */
    SPI_FLASH_WriteEnable();
    SPI_FLASH_WaitForWriteEnd();
    /* 擦除扇区 */
    /* 选择 FLASH: CS 低电平 */
```

(下页继续)

(续上页)

```
    SPI_FLASH_CS_LOW();
    /* 发送扇区擦除指令 */
    SPI_FLASH_SendByte(W25X_SectorErase);
    /* 发送擦除扇区地址的高位 */
    SPI_FLASH_SendByte((SectorAddr & 0xFF0000) >> 16);
    /* 发送擦除扇区地址的中位 */
    SPI_FLASH_SendByte((SectorAddr & 0xFF00) >> 8);
    /* 发送擦除扇区地址的低位 */
    SPI_FLASH_SendByte(SectorAddr & 0xFF);
    /* 停止信号 FLASH: CS 高电平 */
    SPI_FLASH_CS_HIGH();
    /* 等待擦除完毕 */
    SPI_FLASH_WaitForWriteEnd();
}
```

这段代码调用的函数在前面都已讲解，只要注意发送擦除地址时高位在前即可。调用扇区擦除指令时注意输入的地址要对齐到 4KB。

23.4.2.2.10 FLASH 的页写入

目标扇区被擦除完毕后，就可以向它写入数据了。与 EEPROM 类似，FLASH 芯片也有页写入命令，使用页写入命令最多可以一次向 FLASH 传输 256 个字节的数据，我们把这个单位为页大小。FLASH 页写入的时序见图 *24_13*。

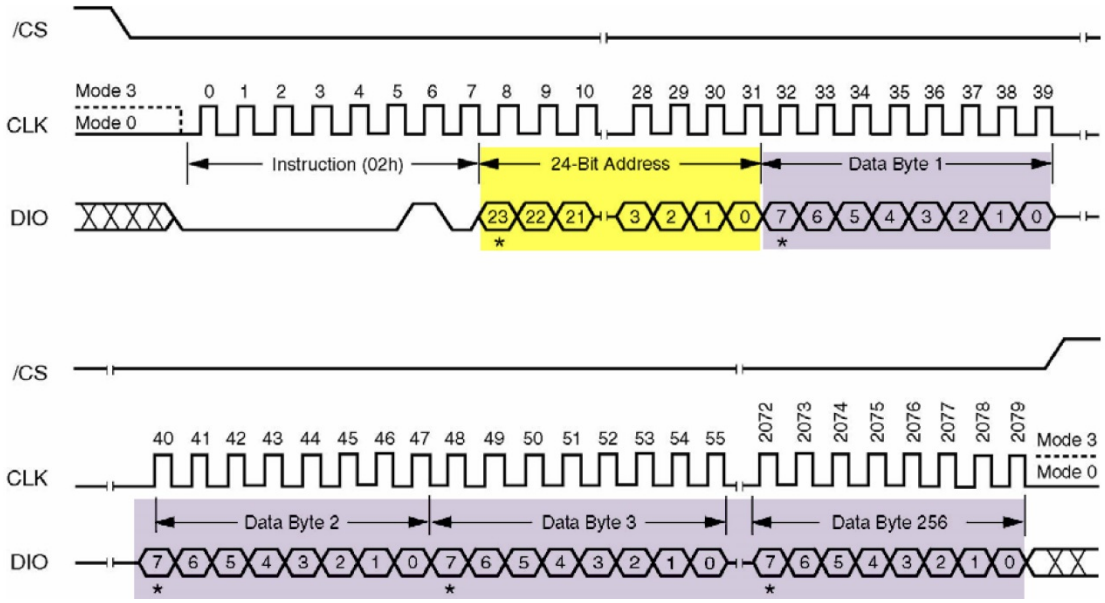


图 24-13 FLASH 芯片页写入

从时序图可知，第 1 个字节为“页写入指令”编码，2-4 字节为要写入的“地址 A”，接着的是要写入的内容，最多个可以发送 256 字节数据，这些数据将会从“地址 A”开始，按顺序写入到 FLASH 的存储矩阵。若发送的数据超出 256 个，则会覆盖前面发送的数据。

与擦除指令不一样，页写入指令的地址并不要求按 256 字节对齐，只要确认目标存储单元是擦除状态即可 (即被擦除后没有被写入过)。所以，若对“地址 x”执行页写入指令后，发送了 200 个字节数据后终止通讯，下一次再执行页写入指令，从“地址 (x+200)”开始写入 200 个字节也是没有问题的 (小于 256 均可)。只是在实际应用中由于基本擦除单元是 4KB，一般都以扇区为单位进行读写，想深入了解，可学习我们的“FLASH 文件系统”相关的例子。

把页写入时序封装成函数，其实现见*代码清单 24_11*。

列表 13: 代码清单 24-11 FLASH 的页写入

```
/**
* @brief  对 FLASH 按页写入数据，调用本函数写入数据前需要先擦除扇区
* @param  pBuffer，要写入数据的指针
* @param WriteAddr，写入地址
```

(下页继续)

(续上页)

```
* @param  NumByteToWrite, 写入数据长度, 必须小于等于页大小
* @retval 无
*/
void SPI_FLASH_PageWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
    /* 发送 FLASH 写使能命令 */
    SPI_FLASH_WriteEnable();

    /* 选择 FLASH: CS 低电平 */
    SPI_FLASH_CS_LOW();
    /* 写送写指令 */
    SPI_FLASH_SendByte(W25X_PageProgram);
    /* 发送写地址的高位 */
    SPI_FLASH_SendByte((WriteAddr & 0xFF0000) >> 16);
    /* 发送写地址的中位 */
    SPI_FLASH_SendByte((WriteAddr & 0xFF00) >> 8);
    /* 发送写地址的低位 */
    SPI_FLASH_SendByte(WriteAddr & 0xFF);

    if (NumByteToWrite > SPI_FLASH_PerWritePageSize)
    {
        NumByteToWrite = SPI_FLASH_PerWritePageSize;
        FLASH_ERROR("SPI_FLASH_PageWrite too large!");
    }

    /* 写入数据 */
    while (NumByteToWrite--)
    {
        /* 发送当前要写入的字节数据 */
        SPI_FLASH_SendByte(*pBuffer);
        /* 指向下一字节数据 */
        pBuffer++;
```

(下页继续)

(续上页)

```
    }

    /* 停止信号 FLASH: CS 高电平 */
    SPI_FLASH_CS_HIGH();

    /* 等待写入完毕 */
    SPI_FLASH_WaitForWriteEnd();
}
```

这段代码的内容为：先发送“写使能”命令，接着才开始页写入时序，然后发送指令编码、地址，再把要写入的数据一个接一个地发送出去，发送完后结束通讯，检查 FLASH 状态寄存器，等待 FLASH 内部写入结束。

23.4.2.2.11 不定量数据写入

应用的时候我们常常要写入不定量的数据，直接调用“页写入”函数并不是特别方便，所以我们在它的基础上编写了“不定量数据写入”的函数，基实现见代码清单 *24_12*。

列表 14: 代码清单 24-12 不定量数据写入

```
/**
* @brief  对 FLASH 写入数据，调用本函数写入数据前需要先擦除扇区
* @param   pBuffer，要写入数据的指针
* @param  WriteAddr，写入地址
* @param  NumByteToWrite，写入数据长度
* @retval 无
*/
void SPI_FLASH_BufferWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
    u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0, temp = 0;
```

(下页继续)

(续上页)

```
    /*mod 运算求余，若 writeAddr 是 SPI_FLASH_PageSize 整数倍，
    运算结果 Addr 值为 0*/
    Addr = WriteAddr % SPI_FLASH_PageSize;

    /* 差 count 个数据值，刚好可以对齐到页地址 */
    count = SPI_FLASH_PageSize - Addr;
    /* 计算出要写多少整数页 */
    NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
    /*mod 运算求余，计算出剩余不满一页的字节数 */
    NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;

    /* Addr=0，则 WriteAddr 刚好按页对齐 aligned  */
    if (Addr == 0)
    {
        /* NumByteToWrite < SPI_FLASH_PageSize */
        if (NumOfPage == 0)
        {
            SPI_FLASH_PageWrite(pBuffer, WriteAddr,
                                NumByteToWrite);
        }
        else /* NumByteToWrite > SPI_FLASH_PageSize */
        {
            /* 先把整数页都写了 */
            while (NumOfPage--)
            {
                SPI_FLASH_PageWrite(pBuffer, WriteAddr,
                                    SPI_FLASH_PageSize);
                WriteAddr +=  SPI_FLASH_PageSize;
                pBuffer += SPI_FLASH_PageSize;
            }
            /* 若有多余的不满一页的数据，把它写完 */
            SPI_FLASH_PageWrite(pBuffer, WriteAddr,
```

(下页继续)

(续上页)

```
                                NumOfSingle);
        }
    }
    /* 若地址与 SPI_FLASH_PageSize 不对齐  */
    else
    {
        /* NumByteToWrite < SPI_FLASH_PageSize */
        if (NumOfPage == 0)
        {
            /* 当前页剩余的 count 个位置比 NumOfSingle 小, 一页写不完 */
            if (NumOfSingle > count)
            {
                temp = NumOfSingle - count;
                /* 先写满当前页 */
                SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);

                WriteAddr +=  count;
                pBuffer += count;
                /* 再写剩余的数据 */
                SPI_FLASH_PageWrite(pBuffer, WriteAddr, temp);
            }
            else /* 当前页剩余的 count 个位置能写完 NumOfSingle 个数据 */
            {
                SPI_FLASH_PageWrite(pBuffer, WriteAddr,
                                    NumByteToWrite);
            }
        }
        else /* NumByteToWrite > SPI_FLASH_PageSize */
        {
            /* 地址不对齐多出的 count 分开处理, 不加入这个运算 */
            NumByteToWrite -= count;
            NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
```

(下页继续)

(续上页)

```
            NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;

            /* 先写完 count 个数据，为的是让下一次要写的地址对齐 */
            SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);

            /* 接下来就重复地址对齐的情况 */
            WriteAddr +=  count;
            pBuffer += count;
            /* 把整数页都写了 */
            while (NumOfPage--)
            {
                SPI_FLASH_PageWrite(pBuffer, WriteAddr,
                                    SPI_FLASH_PageSize);
                WriteAddr +=  SPI_FLASH_PageSize;
                pBuffer += SPI_FLASH_PageSize;
            }
            /* 若有多余的不满一页的数据，把它写完 */
            if (NumOfSingle != 0)
            {
                SPI_FLASH_PageWrite(pBuffer, WriteAddr,
                                    NumOfSingle);
            }
        }
    }
}
```

这段代码与 EEPROM 章节中的“快速写入多字节”函数原理是一样的，运算过程在此不再赘述。区别是页的大小以及实际数据写入的时候，使用的是针对 FLASH 芯片的页写入函数，且在实际调用这个“不定量数据写入”函数时，还要注意确保目标扇区处于擦除状态。

23.4.2.2.12 从 FLASH 读取数据

相对于写入，FLASH 芯片的数据读取要简单得多，使用读取指令“Read Data”即可，其指令时序见图 *24_14*。

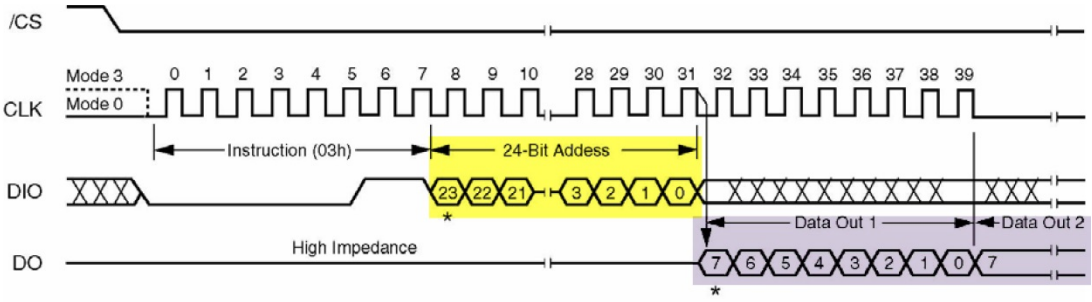


图 24-14 SPI FLASH 读取数据时序

发送了指令编码及要读的起始地址后，FLASH 芯片就会按地址递增的方式返回存储矩阵的内容，读取的数据量没有限制，只要没有停止通讯，FLASH 芯片就会一直返回数据。代码实现见*代码清单 24_13*。

列表 15: 代码清单 24-13 从 FLASH 读取数据

```
/**
* @brief  读取 FLASH 数据
* @param  pBuffer, 存储读出数据的指针
* @param   ReadAddr, 读取地址
* @param   NumByteToRead, 读取数据长度
* @retval 无
*/
void SPI_FLASH_BufferRead(u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
{
    /* 选择 FLASH: CS 低电平 */
    SPI_FLASH_CS_LOW();
    /* 发送 读 指令 */
    SPI_FLASH_SendByte(W25X_ReadData);
    /* 发送 读 地址高位 */
```

(下页继续)

(续上页)

```
    SPI_FLASH_SendByte((ReadAddr & 0xFF0000) >> 16);
    /* 发送 读 地址中位 */
    SPI_FLASH_SendByte((ReadAddr& 0xFF00) >> 8);
    /* 发送 读 地址低位 */
    SPI_FLASH_SendByte(ReadAddr & 0xFF);

    /* 读取数据 */
    while (NumByteToRead--)
    {
        /* 读取一个字节 */
        *pBuffer = SPI_FLASH_SendByte(Dummy_Byte);
        /* 指向下一个字节缓冲区 */
        pBuffer++;
    }

    /* 停止信号 FLASH: CS 高电平 */
    SPI_FLASH_CS_HIGH();
}
```

由于读取的数据量没有限制，所以发送读命令后一直接收 NumByteToRead 个数据到结束即可。

23.4.2.3 main 函数

最后我们来编写 main 函数，进行 FLASH 芯片读写校验。由于 MINI 开发板和拂晓电机开发板没有 RGB 全彩灯，我们把 main 函数分开成两个，见*代码清单 24_14a* 和*代码清单 24_14b*。

列表 16: 代码清单 24-14a MINI 开发板：main 函数

```
1 int main(void)
2 {
3     /* 设定系统时钟为 72MHz */
4     SystemClock_Config();
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();

    /* 配置串口 1 为: 115200 8-N-1 */
    DEBUG_USART_Config();
    printf("\r\n这是一个 8M 串行 flash(W25Q64) 实验 (SPI 驱动)\r\n");

    /* 8M 串行 flash W25Q64 初始化 */
    SPI_FLASH_Init();
    /* 获取 Flash Device ID */
    DeviceID = SPI_FLASH_ReadDeviceID();

    Delay( 200 );

    /* 获取 SPI Flash ID */
    FlashID = SPI_FLASH_ReadID();

    printf("\r\nFlashID is 0x%X,  Manufacturer Device ID is 0x%X\r\n",
        FlashID, DeviceID);

    /* 检验 SPI Flash ID */
    if (FlashID == sFLASH_ID) {
        printf("\r\n检测到 SPI FLASH W25Q128 !\r\n");

        /* 擦除将要写入的 SPI FLASH
        扇区, FLASH 写入前要先擦除 */
        SPI_FLASH_SectorErase(FLASH_SectorToErase);

        /* 将发送缓冲区的数据写到 flash 中 */
        SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress,BufferSize);
        printf("\r\n写入的数据为: \r\n%s", Tx_Buffer);

        /* 将刚刚写入的数据读出来放到接收缓冲区中 */
```

(下页继续)

(续上页)

```
37          SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);
38          printf("\r\n读出的数据为: \r\n%s", Rx_Buffer);
39
40          /* 检查写入的数据与读出的数据是否相等 */
41          TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);
42
43          if ( PASSED == TransferStatus1 ) {
44              LED2_ON;
45              printf("\r\n8M 串行 flash(W25Q64) 测试成功!\n\r");
46          } else {
47              LED1_ON;
48              printf("\r\n8M 串行 flash(W25Q64) 测试失败!\n\r");
49          }
50      }// if (FlashID == sFLASH_ID)
51      else {
52          LED1_ON;
53          printf("\r\n获取不到 W25Q64 ID!\n\r");
54      }
55
56      SPI_Flash_PowerDown();
57      while (1);
58  }
```

列表 17: 代码清单 24-14b 指南者和霸道开发板：main 函数

```
int main(void)
{
    /* 设定系统时钟为 72MHz */
    SystemClock_Config();

    LED_GPIO_Config();
```

(下页继续)

(续上页)

```
    LED_BLUE;

    /* 配置串口 1 为: 115200 8-N-1 */
    DEBUG_USART_Config();

    printf("\r\n这是一个 8M 串行 flash(W25Q64) 实验 (SPI 驱动) \r\n");

    /* 16M 串行 flash W25Q64 初始化 */
    SPI_FLASH_Init();
    /* 获取 Flash Device ID */
    DeviceID = SPI_FLASH_ReadDeviceID();

    Delay( 200 );

    /* 获取 SPI Flash ID */
    FlashID = SPI_FLASH_ReadID();

    printf("\r\nFlashID is 0x%X,  Manufacturer Device ID is 0x%X\r\n",␣
↪FlashID, DeviceID);

    /* 检验 SPI Flash ID */
    if (FlashID == sFLASH_ID) {
        printf("\r\n检测到 SPI FLASH W25Q128 !\r\n");

        /* 擦除将要写入的 SPI FLASH  扇区, FLASH 写入前要先擦除 */
        SPI_FLASH_SectorErase(FLASH_SectorToErase);

        /* 将发送缓冲区的数据写到 flash 中 */
        SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, BufferSize);
        printf("\r\n写入的数据为: \r\n%s", Tx_Buffer);

        /* 将刚刚写入的数据读出来放到接收缓冲区中  */
```

(下页继续)

(续上页)

```
        SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);
        printf("\r\n读出的数据为：\r\n%s", Rx_Buffer);

        /* 检查写入的数据与读出的数据是否相等 */
        TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);

        if ( PASSED == TransferStatus1 ) {
            LED_GREEN;
            printf("\r\n8M 串行 flash(W25Q64) 测试成功！\n\r");
        } else {
            LED_RED;
            printf("\r\n8M 串行 flash(W25Q64) 测试失败！\n\r");
        }
    }// if (FlashID == sFLASH_ID)
    else {
        LED_RED;
        printf("\r\n获取不到 W25Q64 ID！\n\r");
    }

    SPI_Flash_PowerDown();
    while (1);
}
```

函数中初始化了 LED、串口、SPI 外设，然后读取 FLASH 芯片的 ID 进行校验，若 ID 校验通过则向 FLASH 的特定地址写入测试数据，然后再从该地址读取数据，测试读写是否正常。

注意：

由于实验板上的 FLASH 芯片默认已经存储了特定用途的数据，如擦除了这些数据会影响到某些程序的运行。所以我们预留了 FLASH 芯片的“第 0 扇区 (0-4096 地址)”专用于本实验，如非必要，请勿擦除其它地址的内容。如已擦除，可在配套资料里找到“刷外部 FLASH 内容”程序，根据其说明给 FLASH 重新写入出厂内容。

23.4.3 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到 FLASH 测试的调试信息。

第 24 章　串行 FLASH 文件系统 FatFs

本章参考资料：《00index_e.html》，这是 FatFs 官方的编译好的 HTML 文档，里面有 FatFs 所有函数的介绍和函数的应用示例，学习 FatFs 看这个官方的文档即可。

图 25-1 FatFs 参考资料

24.1 文件系统

即使读者可能不了解文件系统，读者也一定对“文件”这个概念十分熟悉。数据在 PC 上是以文件的形式储存在磁盘中的，这些数据的形式一般为 ASCII 码或二进制形式。在上一章我们已经写好了 SPI Flash 芯片的驱动函数，我们可以非常方便的在 SPI Flash 芯片上读写数据。如需要记录本书的书名“零死角玩转 STM32-F103 系列”，可以把这些文字转化成 ASCII 码，存储在数组中，然后调用 SPI_FLASH_BufferWrite 函数，把数组内容写入到 SPI Flash 芯片的指定地址上，在需要的时候从该地址把数据读取出来，再对读出来的数据以 ASCII 码的格式进行解读。

但是，这样直接存储数据会带来极大的不便，如难以记录有效数据的位置，难以确定存储介质的剩余空间，以及应以何种格式来解读数据。就如同一个巨大的图书馆无人管理，杂乱无章地存放着各种书籍，难以查找所需的文档。想象一下图书馆的采购人员购书后，把书籍往馆内一扔，拍拍屁股走人，当有人来借阅某本书的时候，就不得不一本本地查找。这样直接存储数据的方式对于小容量的存储介质如 EEPROM 还可以接受，但对于 SPI Flash 芯片或者 SD 卡之类的大容量设备，我们需要一种高效的方式来管理它的存储内容。

这些管理方式即为文件系统，它是为了存储和管理数据，而在存储介质建立的一种组织结构，这些结构包括操作系统引导区、目录和文件。常见的 windows 下的文件系统格式包括 FAT32、NTFS、exFAT。在使用文件系统前，要先对存储介质进行格式化。格式化先擦除原来内容，在存储介质上新建一个文件分配表和目录。这样，文件系统就可以记录数据存放的物理地址，剩余空间。

使用文件系统时，数据都以文件的形式存储。写入新文件时，先在目录中创建一个文件索引，它指示了文件存放的物理地址，再把数据存储到该地址中。当需要读取数据时，可以从目录中找到该文件的索引，进而在相应的地址中读取出数据。具体还涉及到逻辑地址、簇大小、不连续存储等一系列辅助结构或处理过程。

文件系统的存在使我们在存取数据时，不再是简单地向某物理地址直接读写，而是要遵循它的读写格式。如经过逻辑转换，一个完整的文件可能被分开成多段存储到不连续的物理地址，使用目录或链表的方式来获知下一段的位置。

上一章的 SPI Flash 芯片驱动只完成了向物理地址写入数据的工作，而根据文件系统格式的逻辑转换部分则需要额外的代码来完成。实质上，这个逻辑转换部分可以理解为当我们需要写入一段数据时，由它来求解向什么物理地址写入数据、以什么格式写入及写入一些原始数据以外的信息(如目录)。这个逻辑转换部分代码我们也习惯称之为文件系统。

24.2 FatFs 文件系统简介

上面提到的逻辑转换部分代码 (文件系统) 即为本章的要点，文件系统庞大而复杂，它需要根据应用的文件系统格式而编写，而且一般与驱动层分离开来，很方便移植，所以工程应用中一般是移植现成的文件系统源码。

FatFs 是面向小型嵌入式系统的一种通用的 FAT 文件系统。它完全是由 ANSI C 语言编写并且完全独立于底层的 I/O 介质。因此它可以很容易地不加修改地移植到其他的处理器当中，如 8051、

PIC、AVR、SH、Z80、H8、ARM 等。FatFs 支持 FAT12、FAT16、FAT32 等格式，所以我们利用前面写好的 SPI Flash 芯片驱动，把 FatFs 文件系统代码移植到工程之中，就可以利用文件系统的各种函数，对 SPI Flash 芯片以“文件”格式进行读写操作了。

FatFs 文件系统的源码可以从 fatfs 官网下载:

http://elm-chan.org/fsw/ff/00index_e.html

24.2.1 FatFs 的目录结构

在移植 FatFs 文件系统到开发板之前，我们先要到 FatFs 的官网获取源码，官网有对 FatFs 做详细的介绍，有兴趣可以了解。解压之后可看到里面有 doc 和 src 这两个文件夹，见图 *25_2*。doc 文件夹里面是一些使用帮助文档；src 才是 FatFs 文件系统的源码。

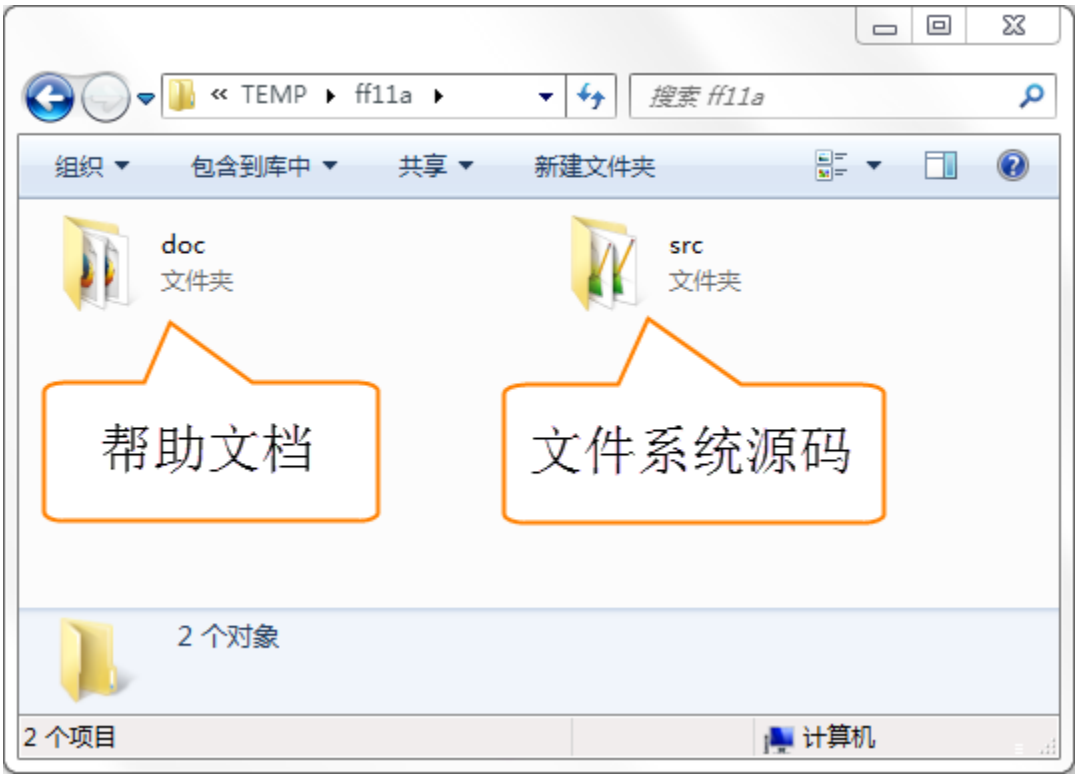


图 25-2 FatFs 文件目录

24.2.2 FatFs 帮助文档

打开 doc 文件夹，可看到如图 *25_3* 的文件目录：

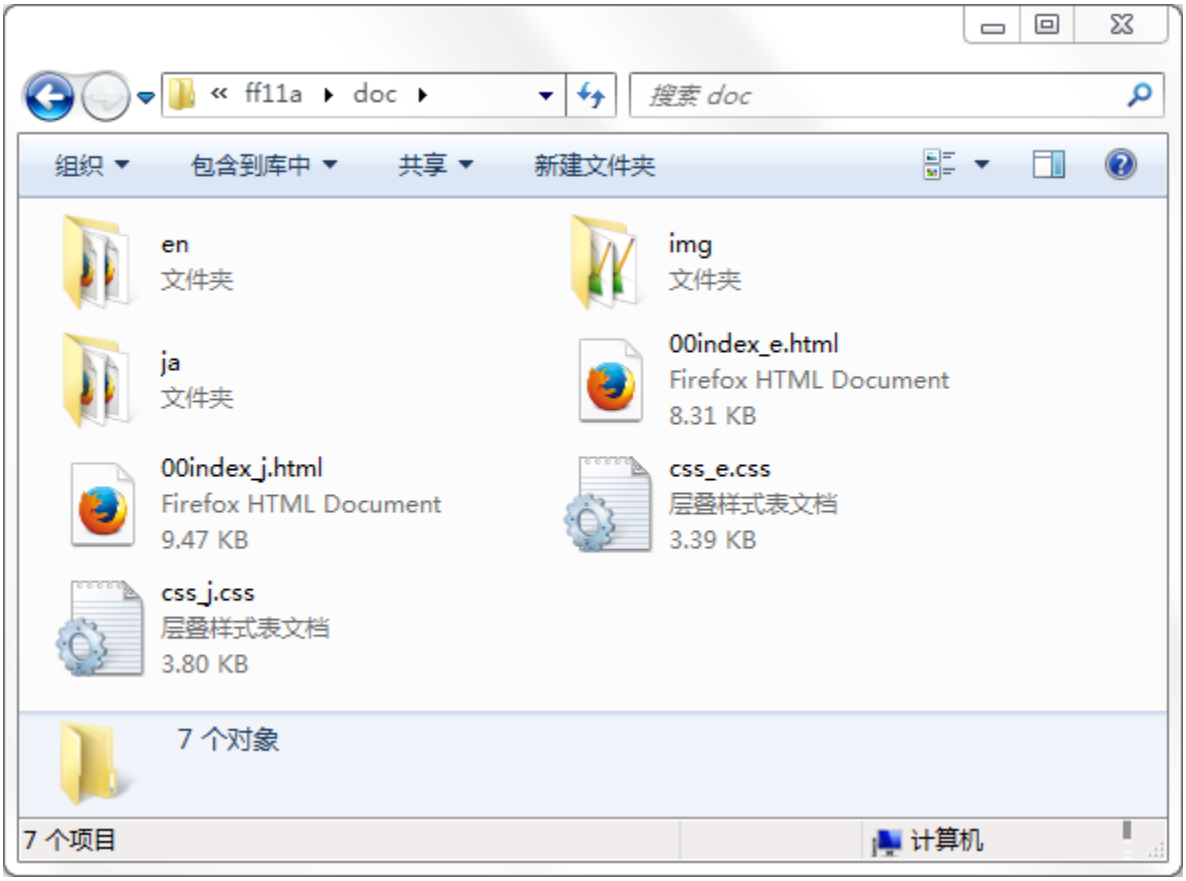


图 25-3 doc 文件夹的文件目录

其中 en 和 ja 这两个文件夹里面是编译好的 html 文档，讲的是 FATFS 里面各个函数的使用方法，这些函数都是封装得非常好的函数，利用这些函数我们就可以操作 SPI Flash 芯片。有关具体的函数我们在用到的时候再讲解。这两个文件夹的唯一区别就是 en 文件夹下的文档是英文的，ja 文件夹下的是日文的。img 文件夹包含 en 和 ja 文件夹下文件需要用到的图片，还有四个名为 app.c 文件，内容都是 FatFs 具体应用例程。00index_e.html 和 00index_j.html 是一些关于 FATFS 的简介，至于另外两个文件可以不看。

24.2.3 FATFS 源码

打开 src 文件夹，可看到如图 *25_4* 的文件目录：

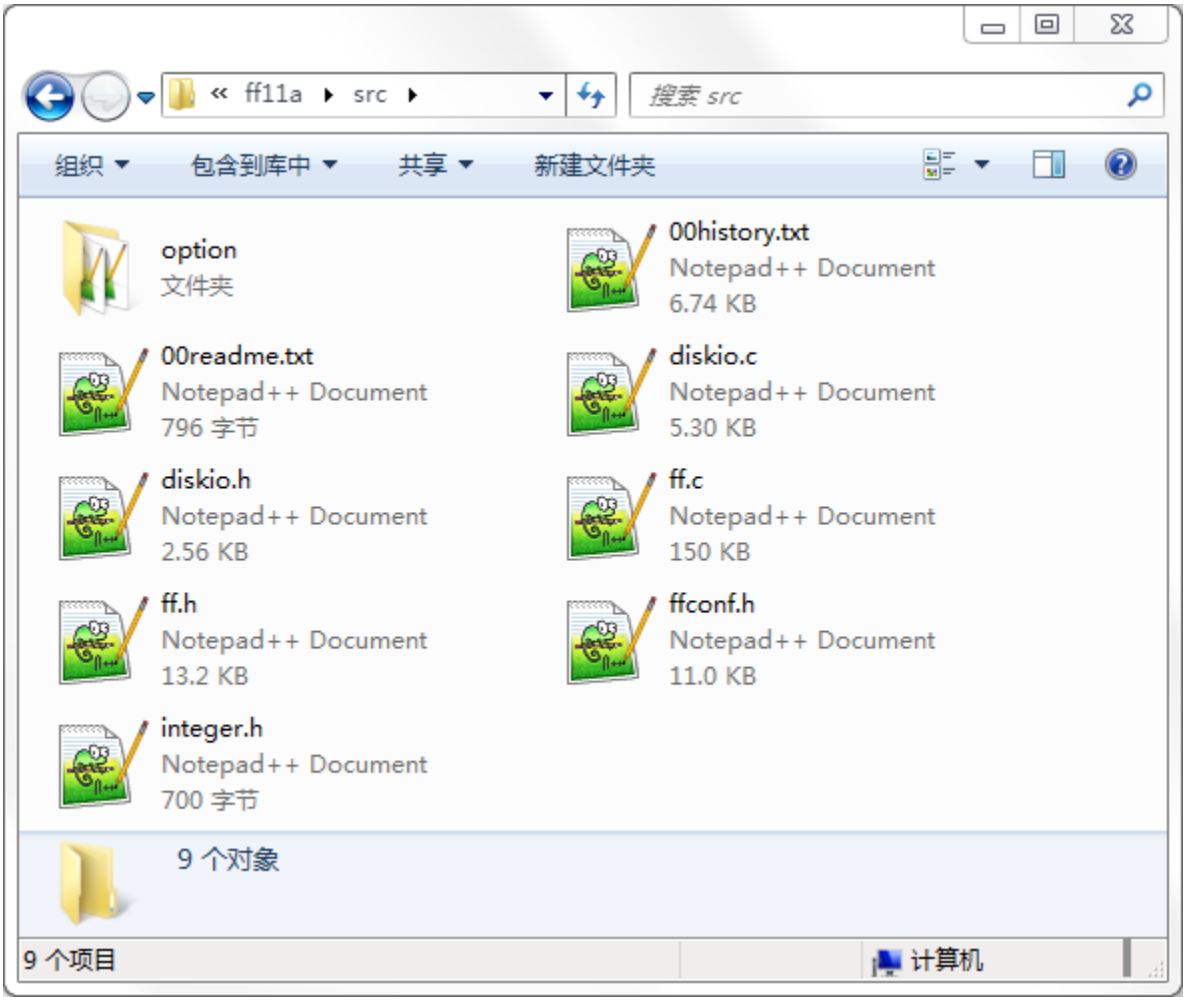


图 25-4 src 文件夹的文件目录

option 文件夹下是一些可选的外部 c 文件，包含了多语言支持需要用到的文件和转换函数。

diskio.c 文件是 FatFs 移植最关键的文件，它为文件系统提供了最底层的访问 SPI Flash 芯片的方法，FatFs 有且仅有它需要用到与 SPI Flash 芯片相关的函数。diskio.h 定义了 FatFs 用到的宏，以及 diskio.c 文件内与底层硬件接口相关的函数声明。

00history.txt 介绍了 FatFs 的版本更新情况。

00readme.txt 说明了当前目录下 diskio.c 、diskio.h、ff.c、ff.h、integer.h 的功能。

src 文件夹下的源码文件功能简介如下：

- integer.h：文件中包含了一些数值类型定义。
- diskio.c：包含底层存储介质的操作函数，这些函数需要用户自己实现，主要添加底层驱动函数。
- ff.c：*FatFs* 核心文件，文件管理的实现方法。该文件独立于底层介质操作文件的函数，利用这些函数实现文件的读写。
- cc936.c：本文件在 *option* 目录下，是简体中文支持所需要添加的文件，包含了简体中文的 *GBK* 和 *Unicode* 相互转换功能函数。
- ffconf.h: 这个头文件包含了对 *FatFs* 功能配置的宏定义，通过修改这些宏定义就可以裁剪 *FatFs* 的功能。如需要支持简体中文，需要把 ffconf.h 中的 *_CODE_PAGE* 的宏改成 *936* 并把上面的 *cc936.c* 文件加入到工程之中。

建议阅读这些源码的顺序为：integer.h –> diskio.c –> ff.c 。

阅读文件系统源码 ff.c 文件需要一定的功底，建议读者先阅读 FAT32 的文件格式，再去分析 ff.c 文件。若仅为使用文件系统，则只需要理解 integer.h 及 diskio.c 文件并会调用 ff.c 文件中的函数就可以了。本章主要讲解如何把 FATFS 文件系统移植到开发板上，并编写一个简单读写操作范例。

24.3 FatFs 文件系统移植实验

24.3.1 FatFs 程序结构图

移植 FatFs 之前我们先通过 FatFs 的程序结构图了解 FatFs 在程序中的关系网络，见图 *25_5*。

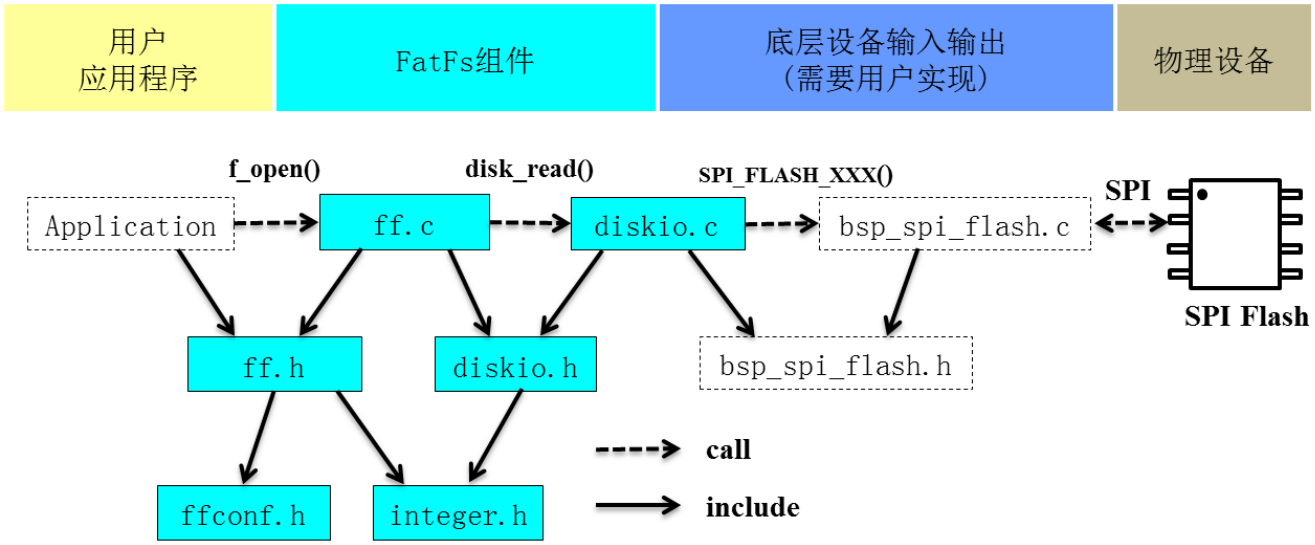


图 25-5 FatFs 程序结构图

用户应用程序需要由用户编写，想实现什么功能就编写什么的程序，一般我们只用到 f_mount()、f_open()、f_write()、f_read() 就可以实现文件的读写操作。

FatFs 组件是 FatFs 的主体，文件都在源码 src 文件夹中，其中 ff.c、ff.h、integer.h 以及 diskio.h 四个文件我们不需要改动，只需要修改 ffconf.h 和 diskio.c 两个文件。

底层设备输入输出要求实现存储设备的读写操作函数、存储设备信息获取函数等等。我们使用 SPI Flash 芯片作为物理设备，在上一章节已经编写好了 SPI Flash 芯片的驱动程序，这里我们就直接使用。

24.3.2 硬件设计

FatFs 属于软件组件，不需要附带其他硬件电路。我们使用 SPI Flash 芯片作为物理存储设备，其硬件电路在上一章已经做了分析，这里就直接使用。

24.3.3 FatFs 移植步骤

上一章我们已经实现了 SPI Flash 芯片驱动程序，并实现了读写测试，为移植 FatFs 方便，我们直接拷贝一份工程，我们在工程基础上添加 FatFs 组件，并修改 main 函数的用户程序即可。

1) 先拷贝一份 SPI Flash 芯片测试的工程文件 (整个文件夹)，并修改文件夹名为“SPI—FatFs 文件系统”。将 FatFs 源码中的 src 文件夹整个文件夹拷贝一份至“SPI—FatFs 文件系统 USER”文件夹下并修改名为“FATFS”，见图 *25_6*。

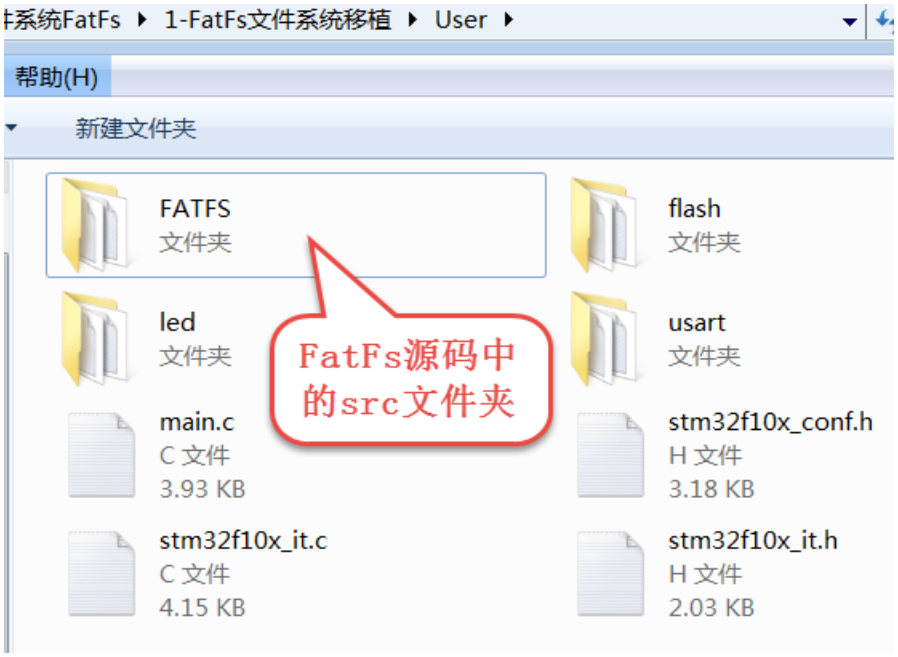


图 25-6 拷贝 FatFs 源码到工程

2) 使用 KEIL 软件打开工程文件 (..SPI—FatFs 文件系统 ProjectRVMDK(uv5)\ BH-F103.uvprojx)，并将 FatFs 组件文件添加到工程中，需要添加有 ff.c、diskio.c 和 cc936.c 三个文件，见图 *25_7*。

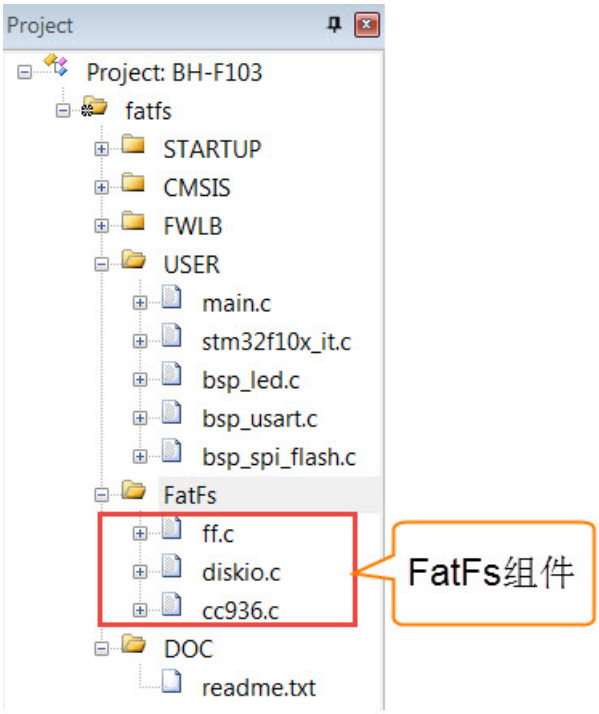


图 25-7 添加 FatFS 文件到工程

3) 添加 FATFS 文件夹到工程的 include 选项中。打开工程选项对话框，选择“C/C++”选项下的“Include Paths”项目，在弹出路径设置对话框中选择添加“FATFS”文件夹，见图 *25_8*。

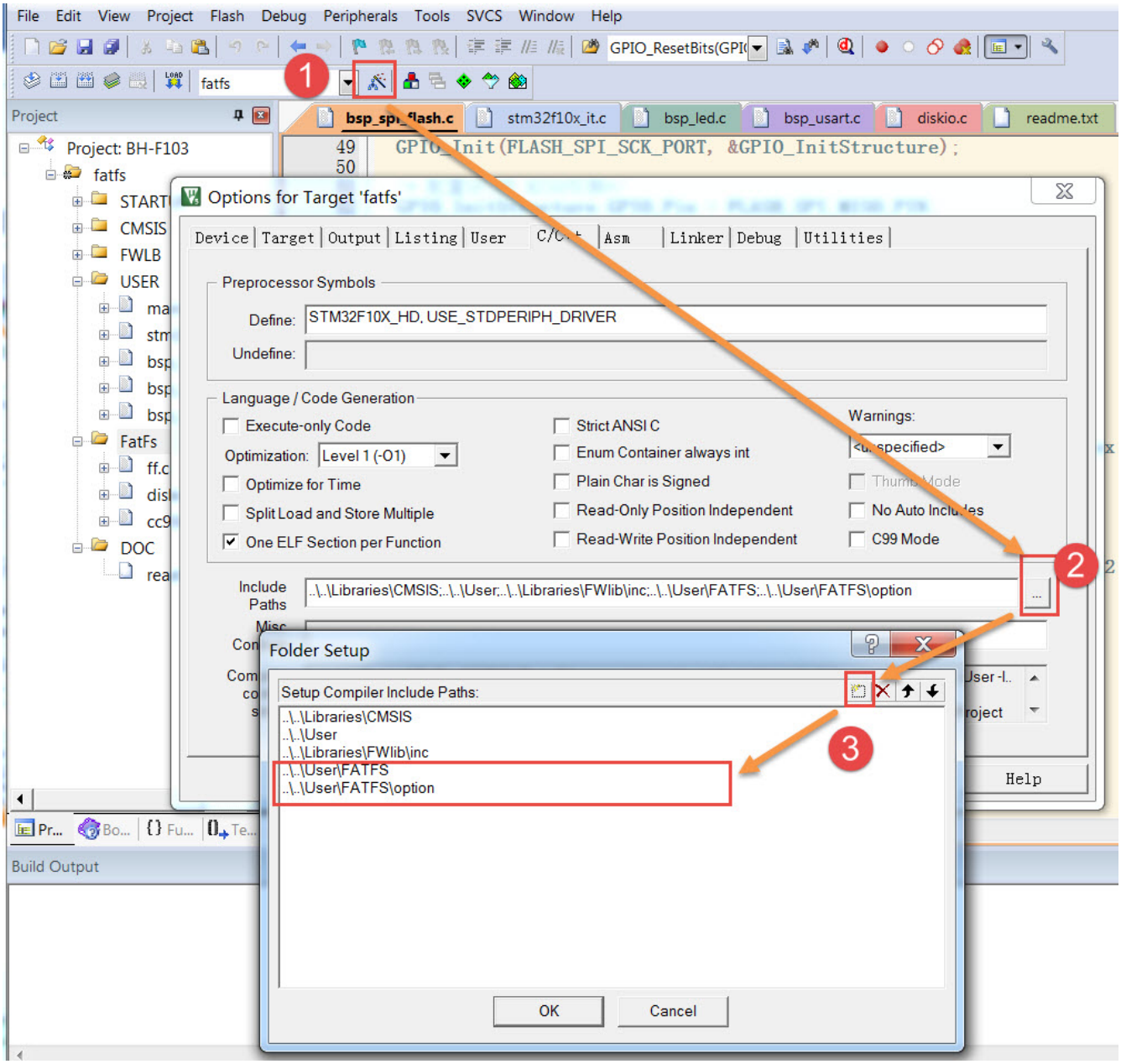


图 25-8 添加 FATFS 路径到工程选项

4) 如果现在编译工程，可以发现有两个错误，一个是来自 diskio.c 文件，提示有一些头文件没找到，diskio.c 文件内容是与底层设备输入输出接口函数文件，不同硬件设计驱动就不同，需要的文件也不同；另外一个错误来自 cc936.c 文件，提示该文件不是工程所必需的，这是因为 FatFs 默认使用日语，我们想要支持简体中文需要修改 FatFs 的配置，即修改 ffconf.h

文件。至此，将 FatFs 添加到工程的框架已经操作完成，接下来要做的就是修改 diskio.c 文件和 ffconf.h 文件。

24.3.4 FatFs 底层设备驱动函数

FatFs 文件系统与底层介质的驱动分离开来，对底层介质的操作都要交给用户去实现，它仅仅是提供了一个函数接口而已。表 25_1 为 FatFs 移植时用户必须支持的函数。通过表 25_1 我们可以清晰知道很多函数是在一定条件下才需要添加的，只有前三个函数是必须添加的。我们完全可以根据实际需求选择实现用到的函数。

前三个函数是实现读文件最基本需求。接下来三个函数是实现创建文件、修改文件需要的。为实现格式化功能，需要在 disk_ioctl 添加两个获取物理设备信息选项。我们一般只要实现前面六个函数就可以了，已经足够满足大部分功能。

为支持简体中文长文件名称需要添加 ff_convert 和 ff_wtoupper 函数，实际这两个已经在 cc936.c 文件中实现，我们只要直接把 cc936.c 文件添加到工程中就可以。

后面六个函数一般都不用。如真有需要可以参考 syscall.c 文件 (srcoption 文件夹内)。

表 25-1 FatFs 移植需要用户支持函数

函数	条件(ffconf.h)	备注
disk_status disk_initialize disk_read	总是需要	底层设备驱动函数
disk_write get_fattime disk_ioctl (CTRL_SYNC)	_FS_READONLY == 0	
disk_ioctl (GET_SECTOR_COUNT) disk_ioctl (GET_BLOCK_SIZE)	_USE_MKFS == 1	
disk_ioctl (GET_SECTOR_SIZE)	_MAX_SS != _MIN_SS	
disk_ioctl (CTRL_TRIM)	_USE_TRIM == 1	
ff_convert ff_wtoupper	_USE_LFN != 0	Unicode 支持，为支持简体中文，添加 cc936.c 到工程即可
ff_cre_syncobj ff_del_syncobj ff_req_grant ff_rel_grant	_FS_REENTRANT == 1	FatFs 可重入配置，需要多任务系统支持(一般不需要)
ff_mem_alloc ff_mem_free	_USE_LFN == 3	长文件名支持，缓冲区设置在堆空间(一般设置_USE_LFN = 2)

底层设备驱动函数是存放在 diskio.c 文件，我们的目的就是把 diskio.c 中的函数接口与 SPI Flash 芯片驱动连接起来。总共有五个函数，分别为设备状态获取 (disk_status)、设备初始化 (disk_initialize)、扇区读取 (disk_read)、扇区写入 (disk_write)、其他控制 (disk_ioctl)。

接下来，我们对每个函数结合 SPI Flash 芯片驱动做详细讲解。

24.3.4.1 宏定义

列表 1: 代码清单 25-1 物理编号宏定义

```
/* 为每个设备定义一个物理编号 */
#define ATA          0      // 预留 SD 卡使用
#define SPI_FLASH    1      // 外部 SPI Flash
```

这两个宏定义在 FatFs 中非常重要，FatFs 是支持多物理设备的，必须为每个物理设备定义一个不同的编号。

SD 卡是预留接口，在讲解 SDIO 接口相关章节后会用到，可以实现使用读写 SD 卡内文件。

24.3.4.2 设备状态获取

列表 2: 代码清单 25-2 设备状态获取

```
DSTATUS disk_status (
    BYTE pdrv    /* 物理编号 */
)
{

    DSTATUS status = STA_NOINIT;

    switch (pdrv) {
    case ATA: /* SD CARD */
        break;

    case SPI_FLASH:
        /* SPI Flash 状态检测：读取 SPI Flash 设备 ID */
        if (sFLASH_ID == SPI_FLASH_ReadID()) {
            /* 设备 ID 读取结果正确 */
            status &= ~STA_NOINIT;
        } else {
```

(下页继续)

(续上页)

```
            /* 设备 ID 读取结果错误 */
            status = STA_NOINIT;;
        }
        break;

    default:
        status = STA_NOINIT;
    }
    return status;
}
```

disk_status 函数只有一个参数 pdrv，表示物理编号。一般我们都是使用 switch 函数实现对 pdrv 的分支判断。对于 SD 卡只是预留接口，留空即可。对于 SPI Flash 芯片，我们直接调用在 SPI_FLASH_ReadID() 获取设备 ID，然后判断是否正确，如果正确，函数返回正常标准；如果错误，函数返回异常标志。SPI_FLASH_ReadID() 是定义在 bsp_spi_flash.c 文件中，上一章节已做了分析。

24.3.4.3 设备初始化

列表 3: 代码清单 25-3 设备初始化

```
DSTATUS disk_initialize (
    BYTE pdrv        /* 物理编号 */
)
{
    uint16_t i;
    DSTATUS status = STA_NOINIT;
    switch (pdrv) {
    case ATA:           /* SD CARD */
        break;
```

(下页继续)

(续上页)

```
    case SPI_FLASH:     /* SPI Flash */
        /* 初始化 SPI Flash */
        SPI_FLASH_Init();
        /* 延时一小段时间 */
        i=500;
        while (--i);
        /* 唤醒 SPI Flash */
        SPI_Flash_WAKEUP();
        /* 获取 SPI Flash 芯片状态 */
        status=disk_status(SPI_FLASH);
        break;

    default:
        status = STA_NOINIT;
    }
    return status;
}
```

disk_initialize 函数也是有一个参数 pdrv，用来指定设备物理编号。对于 SPI Flash 芯片我们调用 SPI_FLASH_Init() 函数实现对 SPI Flash 芯片引脚 GPIO 初始化配置以及 SPI 通信参数配置。SPI_Flash_WAKEUP() 函数唤醒 SPI Flash 芯片，当 SPI Flash 芯片处于睡眠模式时需要唤醒芯片才可以进行读写操作。

最后调用 disk_status 函数获取 SPI Flash 芯片状态，并返回状态值。

24.3.4.4 读取扇区

列表 4: 代码清单 25-4 扇区读取

```
DRESULT disk_read (
    BYTE pdrv,      /* 设备物理编号 (0..) */
    BYTE *buff,     /* 数据缓存区 */
```

(下页继续)

(续上页)

```
    DWORD sector, /* 扇区首地址 */
    UINT count    /* 扇区个数 (1..128) */
)
{
    DRESULT status = RES_PARERR;
    switch (pdrv) {
    case ATA: /* SD CARD */
        break;

    case SPI_FLASH:
        /* 扇区偏移 2MB，外部 Flash 文件系统空间放在 SPI Flash 后面 6MB 空间 */
        sector+=512;
        SPI_FLASH_BufferRead(buff, sector <<12, count<<12);
        status = RES_OK;
        break;

    default:
        status = RES_PARERR;
    }
    return status;
}
```

disk_read 函数有四个形参。pdrv 为设备物理编号。buff 是一个 BYTE 类型指针变量，buff 指向用来存放读取到数据的存储区首地址。sector 是一个 DWORD 类型变量，指定要读取数据的扇区首地址。count 是一个 UINT 类型变量，指定扇区数量。

BYTE 类型实际是 unsigned char 类型，DWORD 类型实际是 unsigned long 类型，UINT 类型实际是 unsigned int 类型，类型定义在 integer.h 文件中。

开发板使用的 SPI Flash 芯片型号为 W25Q64FV，每个扇区大小为 4096 个字节 (4KB)，总共有 8M 字节空间，为兼容后面实验程序，我们只将后部分 6MB 空间分配给 FatFs 使用，前部分 2MB 空间用于其他实验需要，即 FatFs 是从 2MB 空间开始，为实现这个效果需要将所有的读写地址都偏移 512 个扇区空间。

对于 SPI Flash 芯片，主要是使用 SPI_FLASH_BufferRead() 实现在指定地址读取指定长度的数据，它接收三个参数，第一个参数为指定数据存放地址指针。第二个参数为指定数据读取地址，这里使用左移运算符，左移 12 位实际是乘以 4096，这与每个扇区大小是息息相关的。第三个参数为读取数据个数，也是需要使用左移运算符。

24.3.4.5 扇区写入

列表 5: 代码清单 25-5 扇区写入

```
DRESULT disk_write (
    BYTE pdrv,          /* 设备物理编号 (0..) */
    const BYTE *buff, /* 欲写入数据的缓存区 */
    DWORD sector,       /* 扇区首地址 */
    UINT count          /* 扇区个数 (1..128) */
)
{
    uint32_t write_addr;
    DRESULT status = RES_PARERR;
    if (!count) {
        return RES_PARERR;     /* Check parameter */
    }

    switch (pdrv) {
    case ATA: /* SD CARD */
        break;

    case SPI_FLASH:
        /* 扇区偏移 2MB，外部 Flash 文件系统空间放在 SPI Flash 后面 6MB 空间 */
        sector+=512;
        write_addr = sector<<12;
        SPI_FLASH_SectorErase(write_addr);
        SPI_FLASH_BufferWrite((u8 *)buff,write_addr,count<<12);
```

(下页继续)

(续上页)

```
        status = RES_OK;
        break;

    default:
        status = RES_PARERR;
    }
    return status;
}
```

disk_write 函数有四个形参，pdrv 为设备物理编号。buff 指向待写入扇区数据的首地址。sector，指定要写入数据的扇区首地址。count 指定扇区数量。对于 SPI Flash 芯片，在写入数据之前需要先擦除，所以用到扇区擦除函数 (SPI_FLASH_SectorErase)。然后就是在调用数据写入函数 (SPI_FLASH_BufferWrite) 把数据写入到指定位置内。

24.3.4.6 其他控制

列表 6: 代码清单 25-6 其他控制

```
DRESULT disk_ioctl (
    BYTE pdrv,      /* 物理编号 */
    BYTE cmd,       /* 控制指令 */
    void *buff      /* 写入或者读取数据地址指针 */
)
{
    DRESULT status = RES_PARERR;
    switch (pdrv) {
    case ATA: /* SD CARD */
        break;

    case SPI_FLASH:
        switch (cmd) {
```

(下页继续)

(续上页)

```
        /* 扇区数量: 1536*4096/1024/1024=6(MB) */
        case GET_SECTOR_COUNT:
            *(DWORD * )buff = 1536;
            break;
        /* 扇区大小  */
        case GET_SECTOR_SIZE :
            *(WORD * )buff = 4096;
            break;
        /* 同时擦除扇区个数 */
        case GET_BLOCK_SIZE :
            *(DWORD * )buff = 1;
            break;
        }
        status = RES_OK;
        break;

    default:
        status = RES_PARERR;
    }
    return status;
}
```

disk_ioctl 函数有三个形参，pdrv 为设备物理编号，cmd 为控制指令，包括发出同步信号、获取扇区数目、获取扇区大小、获取擦除块数量等等指令，buff 为指令对应的数据指针。

对于 SPI Flash 芯片，为支持 FatFs 格式化功能，需要用到获取扇区数量 (GET_SECTOR_COUNT) 指令和获取擦除块数量 (GET_BLOCK_SIZE)。另外，SD 卡扇区大小为 512 字节，SPI Flash 芯片一般设置扇区大小为 4096 字节，所以需要用到获取扇区大小 (GET_SECTOR_SIZE) 指令。

24.3.4.7 时间戳获取

列表 7: 代码清单 25-7 时间戳获取

```
__weak DWORD get_fattime(void)
{
    /* 返回当前时间戳 */
    return     ((DWORD)(2015 - 1980) << 25)  /* Year 2015 */
             | ((DWORD)1 << 21)          /* Month 1 */
             | ((DWORD)1 << 16)          /* Mday 1 */
             | ((DWORD)0 << 11)          /* Hour 0 */
             | ((DWORD)0 << 5)           /* Min 0 */
             | ((DWORD)0 >> 1);          /* Sec 0 */
}
```

get_fattime 函数用于获取当前时间戳，在 ff.c 文件中被调用。FatFs 在文件创建、被修改时会记录时间，这里我们直接使用赋值方法设定时间戳。为更好的记录时间，可以使用控制器 RTC 功能，具体要求返回值格式为：

- bit31:25 ——从 1980 至今是多少年，范围是 (0..127)；
- bit24:21 ——月份，范围为 (1..12)；
- bit20:16 ——该月份中的第几日，范围为 (1..31)；
- bit15:11——时，范围为 (0..23)；
- bit10:5 ——分，范围为 (0..59)；
- bit4:0 ——秒/ 2，范围为 (0..29) 。

24.3.5 FatFs 功能配置

ffconf.h 文件是 FatFs 功能配置文件，我们可以对文件内容进行修改，使得 FatFs 更符合我们的要求。ffconf.h 对每个配置选项都做了详细的使用情况说明。下面只列出修改的配置，其他配置采用默认即可。

列表 8: 代码清单 25-8 FatFs 功能配置选项

```
#define _USE_MKFS 1
#define _CODE_PAGE 936
#define _USE_LFN 2
#define _VOLUMES 2
#define _MIN_SS 512
#define _MAX_SS 4096
```

1) _USE_MKFS：格式化功能选择，为使用 FatFs 格式化功能，需要把它设置为 1。
2) _CODE_PAGE：语言功能选择，并要求把相关语言文件添加到工程宏。为支持简体中文文件名需要使用“936”，正如在图 25-7 的操作，我们已经把 cc936.c 文件添加到工程中。
3) _USE_LFN：长文件名支持，默认不支持长文件名，这里配置为 2，支持长文件名，并指定使用栈空间为缓冲区。
4) _VOLUMES：指定物理设备数量，这里设置为 2，包括预留 SD 卡和 SPI Flash 芯片。
5) _MIN_SS 、_MAX_SS：指定扇区大小的最小值和最大值。SD 卡扇区大小一般都为 512 字节，SPI Flash 芯片扇区大小一般设置为 4096 字节，所以需要把 _MAX_SS 改为 4096。

24.3.6 FatFs 功能测试

移植操作到此，就已经把 FatFs 全部添加到我们的工程了，这时我们编译功能，顺利编译通过，没有错误。接下来，我们就可以使用编写图 *25_5* 中用户应用程序了。

主要的测试包括格式化测试、文件写入测试和文件读取测试三个部分，主要程序都在 main.c 文件中实现。

24.3.6.1 变量定义

列表 9: 代码清单 25-9 变量定义

```
FATFS fs;                                /* FatFs 文件系统对象 */
FIL fnew;                                /* 文件对象 */
FRESULT res_flash;                       /* 文件操作结果 */
UINT fnum;                               /* 文件成功读写数量 */
BYTE buffer[1024]= {0};                  /* 读缓冲区 */
BYTE textFileBuffer[] =                  /* 写缓冲区 */
    "欢迎使用野火 STM32 开发板 今天是个好日子，新建文件系统测试文件\r\n";
```

FATFS 是在 ff.h 文件定义的一个结构体类型，针对的对象是物理设备，包含了物理设备的物理编号、扇区大小等等信息，一般我们都需要为每个物理设备定义一个 FATFS 变量。

FIL 也是在 ff.h 文件定义的一个结构体类型，针对的对象是文件系统内具体的文件，包含了文件很多基本属性，比如文件大小、路径、当前读写地址等等。如果需要在同一时间打开多个文件进行读写，才需要定义多个 FIL 变量，不然一般定义一个 FIL 变量即可。

FRESULT 是也在 ff.h 文件定义的一个枚举类型，作为 FatFs 函数的返回值类型，主要管理 FatFs 运行中出现的错误。总共有 19 种错误类型，包括物理设备读写错误、找不到文件、没有挂载工作空间等等错误。这在实际编程中非常重要，当有错误出现时我们要停止文件读写，通过返回值我们可以快速定位到错误发生的可能地点。如果运行没有错误才返回 FR_OK。

fnum 是个 32 位无符号整形变量，用来记录实际读取或者写入数据的数组。

buffer 和 textFileBuffer 分别对应读取和写入数据缓存区，都是 8 位无符号整形数组。

24.3.6.2 主函数

列表 10: 代码清单 25-10 主函数

```
int main(void)
{
```

(下页继续)

(续上页)

```
    SystemClock_Config();
    /* 初始化 LED */
    LED_GPIO_Config();
    LED_BLUE;

    /* 初始化调试串口，一般为串口 1 */
    DEBUG_USART_Config();
    printf("****** 这是一个 SPI FLASH 文件系统实验 ******\r\n");

    //在外部 SPI Flash 挂载文件系统，文件系统挂载时会对 SPI 设备初化

    res_flash = f_mount(&fs,"1:",1);

    /*----------------------- 格式化测试 ---------------------------*/
    /* 如果没有文件系统就格式化创建创建文件系统 */
    if (res_flash == FR_NO_FILESYSTEM) {
        printf("》FLASH 还没有文件系统，即将进行格式化...\r\n");
        /* 格式化 */
        res_flash=f_mkfs("1:",0,0);

        if (res_flash == FR_OK) {
            printf("》FLASH 已成功格式化文件系统。\r\n");
            /* 格式化后，先取消挂载 */
            res_flash = f_mount(NULL,"1:",1);
            /* 重新挂载 */
            res_flash = f_mount(&fs,"1:",1);
        } else {
            LED_RED;
            printf("《《格式化失败。》》\r\n");
            while (1);
        }
    } else if (res_flash!=FR_OK) {
```

(下页继续)

(续上页)

```
        printf("!! 外部 Flash 挂载文件系统失败。(%d)\r\n", res_flash);
        printf("!! 可能原因: SPI Flash 初始化不成功。\r\n");
        while (1);
    } else {
        printf("》文件系统挂载成功，可以进行读写测试\r\n");
    }

    /*---------------------- 文件系统测试: 写测试 ---------------------------
↪--*/
    /* 打开文件，如果文件不存在则创建它 */
    printf("\r\n****** 即将进行文件写入测试... ******\r\n");
    res_flash = f_open(&fnew, "1:FatFs 读写测试文件.txt",FA_CREATE_ALWAYS |␣
↪FA_WRITE );
    if ( res_flash == FR_OK ) {
        printf("》打开/创建 FatFs 读写测试文件.txt 文件成功，向文件写入数据。\r\n");
        /* 将指定存储区内容写入到文件内 */
        res_flash=f_write(&fnew,WriteBuffer,sizeof(WriteBuffer),&fnum);
        if (res_flash==FR_OK) {
            printf("》文件写入成功，写入字节数据: %d\n", fnum);
            printf("》向文件写入的数据为: \r\n%s\r\n", WriteBuffer);
        } else {
            printf("!! 文件写入失败: (%d)\n",res_flash);
        }
        /* 不再读写，关闭文件 */
        f_close(&fnew);
    } else {
        LED_RED;
        printf("!! 打开/创建文件失败。\r\n");
    }

    /*------------------- 文件系统测试: 读测试 ------------------------------
↪-----*/
```

(下页继续)

(续上页)

```
    printf("****** 即将进行文件读取测试... ******\r\n");
    res_flash = f_open(&fnew, "1:FatFs 读写测试文件.txt",  FA_OPEN_EXISTING␣
↪| FA_READ);
    if (res_flash == FR_OK) {
        LED_GREEN;
        printf("》打开文件成功。\r\n");
        res_flash = f_read(&fnew, ReadBuffer, sizeof(ReadBuffer), &fnum);
        if (res_flash==FR_OK) {
            printf("》文件读取成功，读到字节数据：%d\r\n",
                fnum);
            printf("》读取得的文件数据为：\r\n%s \r\n",  ReadBuffer);
        } else {
            printf("!! 文件读取失败：(%d)\n",res_flash);
        }
    } else {
        LED_RED;
        printf("!! 打开文件失败。\r\n");
    }
    /* 不再读写，关闭文件 */
    f_close(&fnew);

    /* 不再使用文件系统，取消挂载文件系统 */
    f_mount(NULL,"1:",1);

    /* 操作完成，停机 */
    while (1) {
    }
}
```

首先，初始化 RGB 彩灯和调试串口，用来指示程序进程。

FatFs 的第一步工作就是使用 f_mount 函数挂载工作区。f_mount 函数有三个形参，第一个参数是指向 FATFS 变量指针，如果赋值为 NULL 可以取消物理设备挂载。第二个参数为逻辑设备编号，

使用设备根路径表示，与物理设备编号挂钩，在代码清单 25_1 中我们定义 SPI Flash 芯片物理编号为 1，所以这里使用“1:”。第三个参数可选 0 或 1，1 表示立即挂载，0 表示不立即挂载，延迟挂载。f_mount 函数会返回一个 FRESULT 类型值，指示运行情况。

如果 f_mount 函数返回值为 FR_NO_FILESYSTEM，说明没有 FAT 文件系统，比如新出厂的 SPI Flash 芯片就没有 FAT 文件系统。我们就必须对物理设备进行格式化处理。使用 f_mkfs 函数可以实现格式化操作。f_mkfs 函数有三个形参，第一个参数为逻辑设备编号；第二参数可选 0 或者 1，0 表示设备为一般硬盘，1 表示设备为软盘。第三个参数指定扇区大小，如果为 0，表示通过代码清单 25_6 中 disk_ioctl 函数获取。格式化成功后需要先取消挂载原来设备，再重新挂载设备。

在设备正常挂载后，就可以进行文件读写操作了。使用文件之前，必须使用 f_open 函数打开文件，不再使用文件必须使用 f_close 函数关闭文件，这个跟电脑端操作文件步骤类似。f_open 函数有三个形参，第一个参数为文件对象指针。第二参数为目标文件，包含绝对路径的文件名称和后缀名。第三个参数为访问文件模式选择，可以是打开已经存在的文件模式、读模式、写模式、新建模式、总是新建模式等的或运行结果。比如对于写测试，使用 FA_CREATE_ALWAYS 和 FA_WRITE 组合模式，就是总是新建文件并进行写模式。

f_close 函数用于不再对文件进行读写操作关闭文件，f_close 函数只要一个形参，为文件对象指针。f_close 函数运行可以确保缓冲区完全写入到文件内。

成功打开文件之后就可以使用 f_write 函数和 f_read 函数对文件进行写操作和读操作。这两个函数用到的参数是一致的，只不过一个是数据写入，一个是数据读取。f_write 函数第一个形参为文件对象指针，使用与 f_open 函数一致即可。第二个参数为待写入数据的首地址，对于 f_read 函数就是用来存放读出数据的首地址。第三个参数为写入数据的字节数，对于 f_read 函数就是欲读取数据的字节数。第四个参数为 32 位无符号整形指针，这里使用 fnum 变量地址赋值给它，在运行读写操作函数后，fnum 变量指示成功读取或者写入的字节个数。

最后，不再使用文件系统时，使用 f_mount 函数取消挂载。

24.3.7 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。程序开始运行后，RGB 彩灯为蓝色，在串口调试助手可看到格式化测试、写文件检测和读文件检测三个过程；最后如果所有读写操作都正常，RGB 彩灯会指示为绿色，如果在运行中 FatFs 出现错误 RGB 彩灯指示为红色。

虽然我们通过 RGB 彩灯指示和串口调试助手信息打印方法来说明 FatFs 移植成功，并顺利通过测试，但心底总是很踏实，所谓眼见为实，虽然我们创建了“FatFs 读写测试文件.txt”这个文件，却完全看不到实体。这个确实是个问题，因为我们这里使用 SPI Flash 芯片作为物理设备，并不像 SD 卡那么方便直接用读卡器就可以在电脑端打开验证。另外一个问题，就目前来说，在 SPI Flash 芯片上挂载 FatFs 好像没有实际意义，无法发挥文件系统功能。

实际上，这里归根到底就是我们目前没办法在电脑端查看 SPI Flash 芯片内 FatFs 的内容，没办法非常方便拷贝、删除文件。我们当然不会做无用功，STM32 控制器还有一个硬件资源可以解决上面的问题，就是 USB！我们可以通过编程把整个开发板变成一个 U 盘，而 U 盘存储空间就是 SPI Flash 芯片的空间。这样非常方便实现文件读写。至于 USB 内容将在 USB 相关章节讲解。

24.4 FatFs 功能使用实验

上个实验我们实现了 FatFs 的格式化、读文件和写文件功能，这个已经满足很多部分的运用需要。有时，我们需要更多的文件操作功能，FatFs 还是提供了不少的功能的，比如设备存储空间信息获取、读写文件指针定位、创建目录、文件移动和重命名、文件或目录信息获取等等功能。我们接下来这个实验内容就是展示 FatFs 众多功能，提供一个很好了范例，以后有用到相关内容，参考使用非常方便。

24.4.1 硬件设计

本实验主要使用 FatFs 软件功能，不需要其他硬件模块，使用与 FatFs 移植实验相同硬件配置即可。

24.4.2 软件设计

上个实验我们已经移植好了 FatFs，这个例程主要是应用，所以简单起见，直接拷贝上个实验的工程文件，保持 FatFs 底层驱动程序，我们只改 main.c 文件内容，实现应用程序。

24.4.2.1 FatFs 多项功能测试

列表 11: 代码清单 25-11 FatFs 多项功能测试

```
/* FatFs 多项功能测试 */
static FRESULT miscellaneous(void)
{
    DIR dir;
    FATFS *pfs;
    DWORD fre_clust, fre_sect, tot_sect;

    printf("\n*************** 设备信息获取 ***************\r\n");
    /* 获取设备信息和空簇大小 */
    res_flash = f_getfree("1:", &fre_clust, &pfs);

    /* 计算得到总的扇区个数和空扇区个数 */
    tot_sect = (pfs->n_fatent - 2) * pfs->csize;
    fre_sect = fre_clust * pfs->csize;

    /* 打印信息 (4096 字节/扇区) */
    printf("》设备总空间: %10lu KB。\n》可用空间:  %10lu KB。\n", tot_sect *4,␣
↪fre_sect *4);
```

(下页继续)

(续上页)

```
printf("\n******** 文件定位和格式化写入功能测试 ********\r\n");
res_flash = f_open(&fnew, "1:FatFs 功能测试文件.txt",
                   FA_CREATE_ALWAYS| FA_WRITE|FA_READ );
res_flash=f_write(&fnew,"欢迎使用野火 STM32 开发板",50,&fnum);
if ( res_flash == FR_OK )
{
    /*  文件定位，定位到文件的末尾 */
    res_flash = f_lseek(&fnew,f_size(&fnew)-1);
    if (res_flash == FR_OK)
    {
        /* 格式化写入，参数格式类似 printf 函数 */
        f_printf(&fnew,"\n在原来文件新添加一行内容\n");
        f_printf(&fnew,"》设备总空间：%10lu KB。\n》可用空间;%10lu KB。\n",
                 tot_sect *4, fre_sect *4);
        /*  文件定位到文件起始位置 */
        res_flash = f_lseek(&fnew,0);
        /* 读取文件所有内容到缓存区 */
        res_flash = f_read(&fnew,readbuffer,f_size(&fnew),&fnum);
        if (res_flash == FR_OK)
        {
            printf("》文件内容：\n%s\n",readbuffer);
        }
    }
    f_close(&fnew);

    printf("\n********** 目录创建和重命名功能测试 **********\r\n");
    /* 尝试打开目录 */
    res_flash=f_opendir(&dir,"1:TestDir");
    if (res_flash!=FR_OK)
    {
        /* 打开目录失败，就创建目录 */
```

(下页继续)

(续上页)

```
            res_flash=f_mkdir("1:TestDir");
        }
        else
        {
            /* 如果目录已经存在，关闭它 */
            res_flash=f_closedir(&dir);
            /* 删除文件 */
            f_unlink("1:TestDir/testdir.txt");
        }
        if (res_flash==FR_OK)
        {
            /* 重命名并移动文件 */
            res_flash=f_rename("1:FatFs 功能测试文件.txt","1:TestDir/testdir.
↪txt");
            if (res_flash==FR_OK)
            {
                printf("》重命名并移动文件操作成功\n");
            }
            else
            {
                printf("》重命名并移动文件操作失败：%d\n",res_flash);
            }
        }
    }
    else
    {
        printf("!! 打开文件失败：%d\n",res_flash);
        printf("!! 或许需要再次运行“FatFs 移植与读写测试”工程\n");
    }
    return res_flash;
}
```

首先是设备存储信息获取，目的是获取设备总容量和剩余可用空间。f_getfree 函数是设备空闲簇信息获取函数，有三个形参，第一个参数为逻辑设备编号；第二个参数为返回空闲簇数量，这里 1 簇等于 1 个扇区；第三个参数为返回指向文件系统对象的指针。通过计算可得到设备总的扇区个数以及空闲扇区个数，对于 SPI Flash 芯片我们设置每个扇区为 4096 字节大小，即 4KB。这样很容易就算出设备存储信息。

接下来是文件读写指针定位和格式化输入功能测试。文件定位在一些场合非常有用，比如我们需要记录多项数据，但每项数据长度不确定，但有个最长长度，我们就可以使用文件定位 lseek 函数功能把数据存放在规定好的地址空间上。当我们需要读取文件内容时就使用文件定位函数定位到对应地址读取。

使用文件读写操作之前都必须使用 f_open 函数打开文件，开始的时候读写指针是在文件起始位置的，马上写入数据的话会覆盖原来文件内容。这里，我们使用 f_lseek 函数定位到文件末尾位置，再写入内容。f_lseek 函数有两个形参，第一个参数为文件对象指针，第二个参数为需要定位的字节数，这个字节数是相对文件起始位置的，比如设置为 0，则将文件读写指针定位到文件起始位置。

f_printf 函数是格式化写入函数，需要把 ffconf.h 文件中的 _USE_STRFUNC 配置为 1 才支持。f_printf 函数用法类似 C 库函数 printf 函数，只是它将数据直接写入到文件中。

最后是目录创建和文件移动和重命名功能。使用 f_opendir 函数可以打开路径 (这里不区分目录和路径概念，下同)，如果路径不存在则创建，如果存在则使用 f_closedir 函数关闭已经打开的路径，并把以前创建的文件删除。新版的 FatFs 支持相对路径功能，使路径操作更加灵活。f_opendir 函数有两个形参，第一个参数为指向路径对象的指针，第二个参数为路径。f_closedir 函数只需要指向路径对象的指针一个形参。

f_mkdir 函数用于创建路径，如果指定的路径不存在就创建它，创建的路径存在形式就是文件夹。f_mkdir 函数只要一个形参，就是指定路径。

f_rename 函数是带有移动功能的重命名函数，它有两个形参，第一个参数为源文件名称，第二个参数为目标名称。目标名称可附带路径，如果路径与源文件路径不同则移动文件到目标路径下。

24.4.2.2 文件信息获取

列表 12: 代码清单 25-12 文件信息获取

```
static FRESULT file_check(void)
{
    static FILINFO fno;

    /* 获取文件信息，必须确保文件存在 */
    res_flash=f_stat("1:TestDir/testdir.txt",&fno);
    if (res_flash==FR_OK) {
        printf(" “testdir.txt” 文件信息: \n");
        printf("》文件大小: %ld(字节)\n", fno.fsize);
        printf("》时间戳: %u/%02u/%02u, %02u:%02u\n",
            (fno.fdate >> 9) + 1980, fno.fdate >> 5 & 15, fno.fdate & 31,
            fno.ftime >> 11, fno.ftime >> 5 & 63);
        printf("》属性: %c%c%c%c%c\n\n",
            (fno.fattrib & AM_DIR) ? 'D' : '-',      // 是一个目录
            (fno.fattrib & AM_RDO) ? 'R' : '-',      // 只读文件
            (fno.fattrib & AM_HID) ? 'H' : '-',      // 隐藏文件
            (fno.fattrib & AM_SYS) ? 'S' : '-',      // 系统文件
            (fno.fattrib & AM_ARC) ? 'A' : '-');     // 档案文件
    }
    return res_flash;
}
```

f_stat 函数用于获取文件的属性，有两个形参，第一个参数为文件路径，第二个参数为返回指向文件信息结构体变量的指针。文件信息结构体变量包含文件的大小、最后修改时间和日期、文件属性、短文件名以及长文件名等信息。

24.4.2.3 路径扫描

列表 13: 代码清单 25-13 路径扫描

```
static FRESULT scan_files (char* path)
{
    FRESULT res;     //部分在递归过程被修改的变量，不用全局变量
    FILINFO fno;
    DIR dir;
    int i;
    char *fn;         // 文件名

#if _USE_LFN
    /* 长文件名支持 */
    /* 简体中文需要 2 个字节保存一个“字” */
    static char lfn[_MAX_LFN*2 + 1];
    fno.lfname = lfn;
    fno.lfsize = sizeof(lfn);
#endif
    //打开目录
    res = f_opendir(&dir, path);
    if (res == FR_OK) {
        i = strlen(path);
        for (;;) {
            //读取目录下的内容，再读会自动读下一个文件
            res = f_readdir(&dir, &fno);
            //为空时表示所有项目读取完毕，跳出
            if (res != FR_OK || fno.fname[0] == 0) break;
#if _USE_LFN
            fn = *fno.lfname ? fno.lfname : fno.fname;
#else
            fn = fno.fname;
#endif
```

(下页继续)

(续上页)

```
            //点表示当前目录，跳过
            if (*fn == '.') continue;
            //目录，递归读取
            if (fno.fattrib & AM_DIR) {
                //合成完整目录名
                sprintf(&path[i], "/%s", fn);
                //递归遍历
                res = scan_files(path);
                path[i] = 0;
                //打开失败，跳出循环
                if (res != FR_OK)
                    break;
            } else {
                printf("%s/%s\r\n", path, fn);                    //输出文件名
                /* 可以在这里提取特定格式的文件路径 */
            }//else
        } //for
    }
    return res;
}
```

scan_files 函数用来扫描指定路径下的文件。比如我们设计一个 mp3 播放器，我们需要提取 mp3 格式文件，诸如 *.txt、.c 文件我们统统不可要的，这时我们就必须扫描路径下所有文件并把.mp3 或 *.MP3 格式文件提取出来。这里我们不提取特定格式文件，而是把所有文件名称都通过串口打印出来。

我们在 ffconf.h 文件中定义了长文件名称支持 (_USE_LFN=2)，一般有用到简体中文文件名称的都要长文件名支持。短文件名称是 8.3 格式，即名称是 8 个字节，后缀名是 3 个字节，对于使用英文名称还可以，使用中文名称就很容易长度不够了。使能了长文件名支持后，使用之前需要指定文件名的存储区还有存储区的大小。

接下来就是使用 f_opendir 函数打开指定的路径。如果路径存在就使用 f_readdir 函数读取路径下内容，f_readdir 函数可以读取路径下的文件或者文件夹，并保存信息到文件信息对象变量内。

f_readdir 函数有两个形参，第一个参数为指向路径对象变量的指针，第二个参数为指向文件信息对象的指针。f_readdir 函数另外一个特性是自动读取下一个文件对象，即循序运行该函数可以读取该路径下的所有文件。所以，在程序中，我们使用 for 循环让 f_readdir 函数读取所有文件，并在读取所有文件之后退出循环。

在 f_readdir 函数成功读取到一个对象时，我们还不清楚它是一个文件还是一个文件夹，此时我们就可以使用文件信息对象变量的文件属性来判断了，如果判断得出是个文件那我们就直接通过串口打印出来就好了。如果是个文件夹，我们就要进入该文件夹扫描，这时就重新调用扫描函数 scan_files 就可以了，形成一个递归调用结构，只是我们这次用的参数与最开始时候是不同的，现在是使用子文件夹名称。

24.4.2.4 主函数

列表 14: 代码清单 25-14 主函数

```
int main(void)
{
    SystemClock_Config();
    /* 初始化调试串口，一般为串口 1 */
    DEBUG_USART_Config();
    printf("******** 这是一个 SPI FLASH 文件系统实验 *******\r\n");

    //在外部 SPI Flash 挂载文件系统，文件系统挂载时会对 SPI 设备初始化

    res_flash = f_mount(&fs,"1:",1);
    if (res_flash!=FR_OK) {
        printf("!! 外部 Flash 挂载文件系统失败。(%d)\r\n",
                res_flash);
        printf("!! 可能原因: SPI Flash 初始化不成功。\r\n");
        while (1);
    } else {
        printf("》文件系统挂载成功，可以进行测试\r\n");
    }
```

(下页继续)

(续上页)

```
    /* FatFs 多项功能测试 */
    res_flash = miscellaneous();
    printf("\n*************** 文件信息获取测试 **************\r\n");
    res_flash = file_check();
    printf("***************** 文件扫描测试 ****************\r\n")
            ;
    strcpy(fpath,"1:");
    scan_files(fpath);
    /* 不再使用文件系统，取消挂载文件系统 */
    f_mount(NULL,"1:",1);

    /* 操作完成，停机 */
    while (1) {
    }
}
```

串口在程序调试中经常使用，可以把变量值直观打印到串口调试助手，这个信息非常重要，同样在使用之前需要调用 Debug_USART_Config 函数完成调试串口初始化。

使用 FatFs 进行文件操作之前都使用 f_mount 函数挂载物理设备，这里我们使用 SPI Flash 芯片上的 FAT 文件系统。

接下来我们直接调用 miscellaneous 函数进行 FatFs 设备信息获取、文件定位和格式化写入功能以及目录创建和重命名功能测试。调用 file_check 函数进行文件信息获取测试。

scan_files 函数用来扫描路径下的所有文件，fpath 是我们定义的一个包含 100 个元素的字符型数组，并将其赋值为 SPI Flash 芯片物理编号对于的根目录。这样允许 scan_files 函数见打印 SPI Flash 芯片内 FatFs 所有文件到串口调试助手。注意，这里的定义 fpath 数组是必不可少的，因为 scan_files 函数本身是个递归函数，要求实际参数有较大空间的缓存区。

24.4.3 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。程序开始运行，在串口调试助手可看到每个阶段测试相关信息情况。

第 25 章　LCD—液晶显示

本章参考资料：《STM32F10X-中文参考手册》、《STM32F103 增强型系列数据手册》、库帮助文档《STM32F103xG_User_Manual.chm》。

关于开发板配套的液晶屏控制器参数可查阅《ILI9341.pdf》资料获知。

本章讲解的内容涉及对 FSMC 的控制，若您不了解 FSMC 外设，请先学习前面的《FSMC—扩展外部 SRAM》章节。

25.1 显示器简介

显示器属于计算机的 I/O 设备，即输入输出设备。它是一种将特定电子信息输出到屏幕上再反射到人眼的显示工具。常见的有 CRT 显示器、液晶显示器、LED 点阵显示器及 OLED 显示器。

25.1.1 液晶显示器

液晶显示器，简称 LCD(Liquid Crystal Display)，相对于上一代 CRT 显示器 (阴极射线管显示器)，LCD 显示器具有功耗低、体积小、承载的信息量大及不伤眼的优点，因而它成为了现在的主流电子显示设备，其中包括电视、电脑显示器、手机屏幕及各种嵌入式设备的显示器。图 26-1 是液晶电视与 CRT 电视的外观对比，很明显液晶电视更薄，“时尚”是液晶电视给人的第一印象，而 CRT 电视则感觉很“笨重”。

图 26-1 液晶电视及 CRT 电视

液晶是一种介于固体和液体之间的特殊物质，它是一种有机化合物，常态下呈液态，但是它的分子排列却和固体晶体一样非常规则，因此取名液晶。如果给液晶施加电场，会改变它的分子排列，从而改变光线的传播方向，配合偏振光片，它就具有控制光线透过率的作用，再配合彩色滤光片，改变加给液晶电压大小，就能改变某一颜色透光量的多少，图 *26_2* 中的就是绿色显示结构。利用这种原理，做出可控红、绿、蓝光输出强度的显示结构，把三种显示结构组成一个显示单位，通过控制红绿蓝的强度，可以使该单位混合输出不同的色彩，这样的一个显示单位被称为像素。

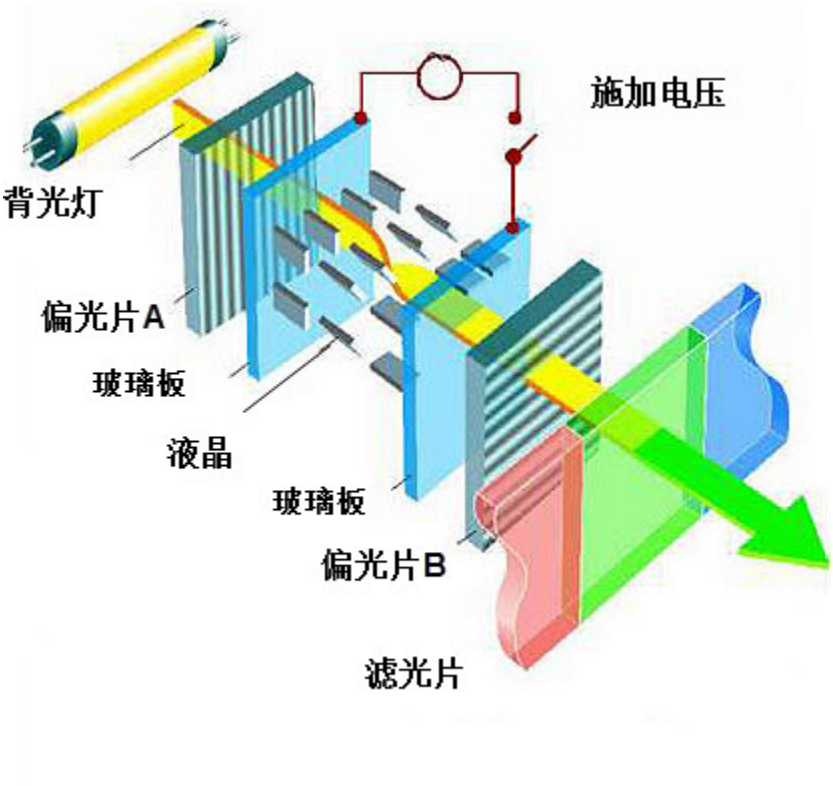


图 26-2 液晶屏的绿色显示结构

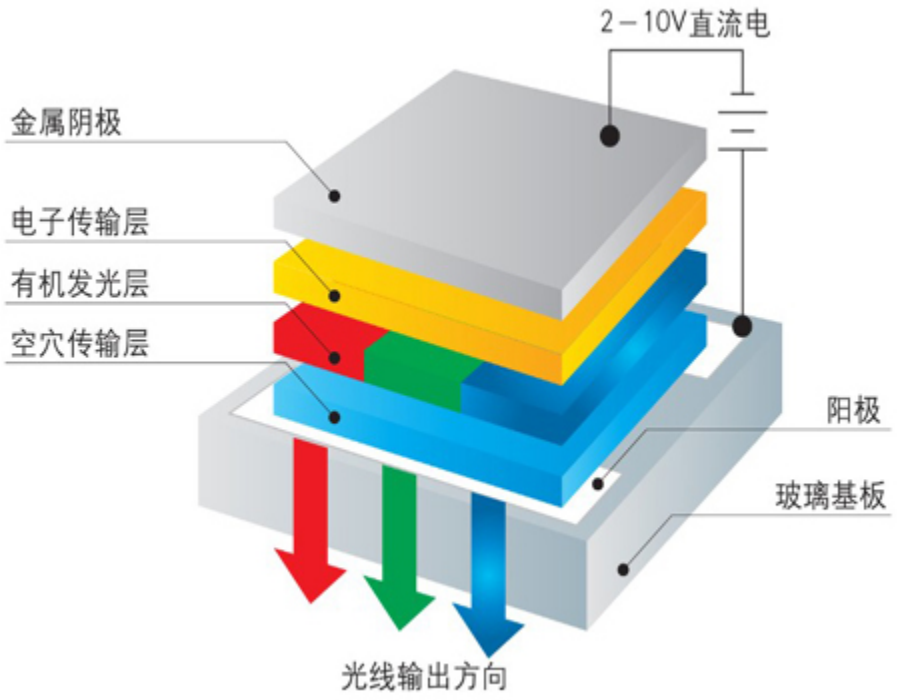
注意液晶本身是不发光的，所以需要有一个背光灯提供光源，光线经过一系列处理过程才到输出，所以输出的光线强度是要比光源的强度低很多的，比较浪费能源 (当然，比 CRT 显示器还是节能多了)。而且这些处理过程会导致显示方向比较窄，也就是它的视角较小，从侧面看屏幕会看不清它的显示内容。另外，输出的色彩变换时，液晶分子转动也需要消耗一定的时间，导致屏幕的响应速度低。

25.1.2 LED 和 OLED 显示器

LED 点阵显示器不存在以上液晶显示器的问题，LED 点阵彩色显示器的单个像素点内包含红绿蓝三色 LED 灯，显示原理类似我们实验板上的 LED 彩灯，通过控制红绿蓝颜色的强度进行混色，实现全彩颜色输出，多个像素点构成一个屏幕。由于每个像素点都是 LED 灯自发光的，所以在户外白天也显示得非常清晰，但由于 LED 灯体积较大，导致屏幕的像素密度低，所以它一般只适合用于广场上的巨型显示器。相对来说，单色的 LED 点阵显示器应用得更广泛，如公交车上

的信息展示牌、店招等，见图 *26_3*。

新一代的 OLED 显示器与 LED 点阵彩色显示器的原理类似，但由于它采用的像素单元是“有机发光二极管”(Organic Light Emitting Diode)，所以像素密度比普通 LED 点阵显示器高得多，见图 *26_4*。



OLED 显示器不需要背光源、对比度高、轻薄、视角广及响应速度快等优点。待到生产工艺更加成熟时，必将取代现在液晶显示器的地位，图 *26_5*。

25.1.3 显示器的基本参数

不管是哪一种显示器，都有一定的参数用于描述它们的特性，各个参数介绍如下：

(1) 像素

像素是组成图像的最基本单元要素，显示器的像素指它成像最小的点，即前面讲解液晶原理中提到的一个显示单元。

(1) 分辨率

一些嵌入式设备的显示器常常以“行像素值 x 列像素值”表示屏幕的分辨率。如分辨率 800x480 表示该显示器的每一行有 800 个像素点，每一列有 480 个像素点，也可理解为有 800 列，480 行。

(2) 色彩深度

色彩深度指显示器的每个像素点能表示多少种颜色，一般用“位”(bit) 来表示。如单色屏的每个像素点能表示亮或灭两种状态 (即实际上能显示 2 种颜色)，用 1 个数据位就可以表示像素点的所有状态，所以它的色彩深度为 1bit，其它常见的显示屏色深为

16bit、24bit。

(3) 显示器尺寸

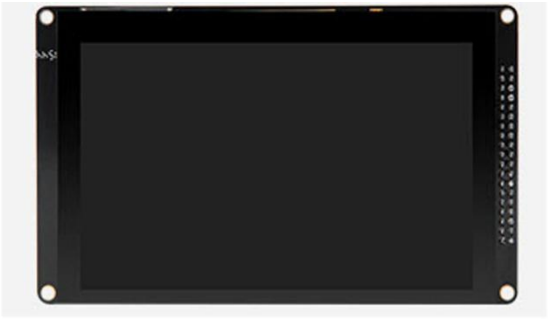
显示器的大小一般以英寸表示，如 5 英寸、21 英寸、24 英寸等，这个长度是指屏幕对角线的长度，通过显示器的对角线长度及长宽比可确定显示器的实际长宽尺寸。

(4) 点距

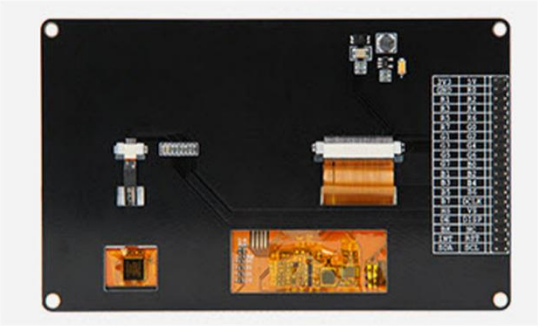
点距指两个相邻像素点之间的距离，它会影响画质的细腻度及观看距离，相同尺寸的屏幕，若分辨率越高，则点距越小，画质越细腻。如现在有些手机的屏幕分辨率比电脑显示器的还大，这是手机屏幕点距小的原因；LED 点阵显示屏的点距一般都比较大，所以适合远距离观看。

25.2 液晶控制原理

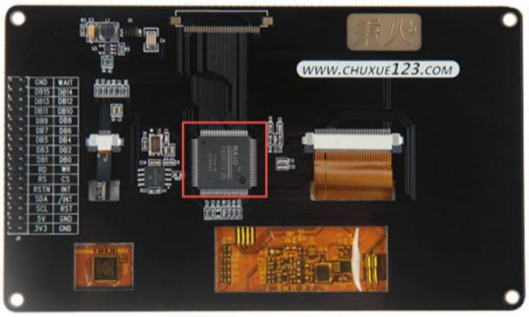
图 *26_6* 是两种适合于 STM32 芯片使用的显示屏，我们以它为例讲解控制液晶屏的基本原理。



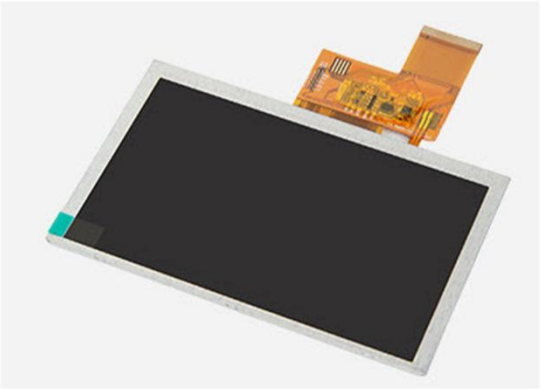
触摸显示屏正面



触摸显示屏背面
（不带液晶控制器）



触摸显示屏背面
（红框中的是液晶控制器）



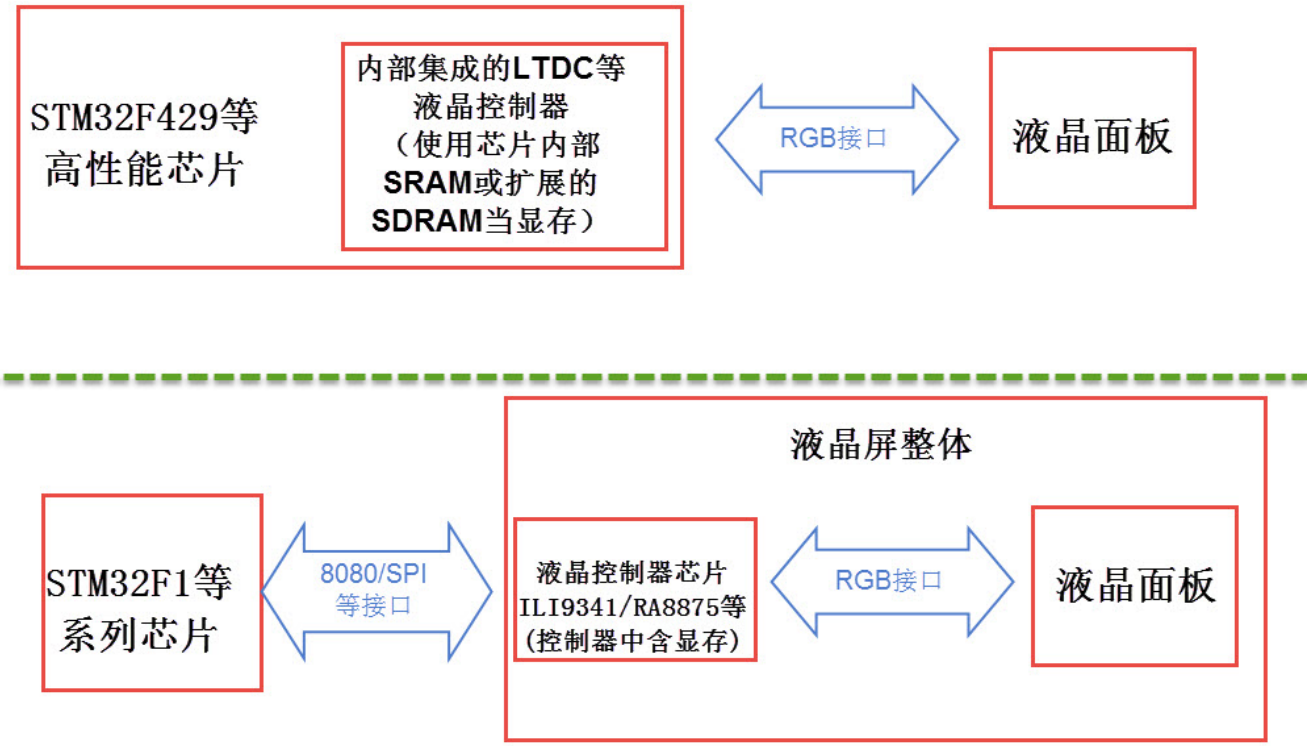
液晶显示器面板

电容触摸屏面板
（带触摸控制器）

这个完整的显示屏由液晶显示面板、电容触摸面板以及 PCB 底板构成。图中的触摸面板带有触摸控制芯片，该芯片处理触摸信号并通过引出的信号线与外部器件通讯，触摸面板中间是透明的，它贴在液晶面板上面，一起构成屏幕的主体，触摸面板与液晶面板引出的排线连接到 PCB 底板上，根据实际需要，PCB 底板上可能会带有“液晶控制器芯片”，图中右侧的液晶屏 PCB 上带有 RA8875 液晶控制器。因为控制液晶面板需要比较多的资源，所以大部分低级微控制器都不

能直接控制液晶面板，需要额外配套一个专用液晶控制器来处理显示过程，外部微控制器只要把它希望显示的数据直接交给液晶控制器即可。而不带液晶控制器的 PCB 底板，只有小部分的电源管理电路，液晶面板的信号线与外部微控制器相连，直接控制。STM32F429 系列的芯片不需要额外的液晶控制器，也就是说它把专用液晶控制器的功能集成到 STM32F429 芯片内部了，可以理解为电脑的 CPU 集成显卡，它节约了额外的控制器成本。而 STM32F1 系列的芯片由于没有集成液晶控制器到芯片内部，所以它只能驱动自带控制器的屏幕，可以理解为电脑的外置显卡。

总的来说，这两类屏幕的控制框图如图 *26_7* 所示。



25.2.1 液晶面板的控制信号

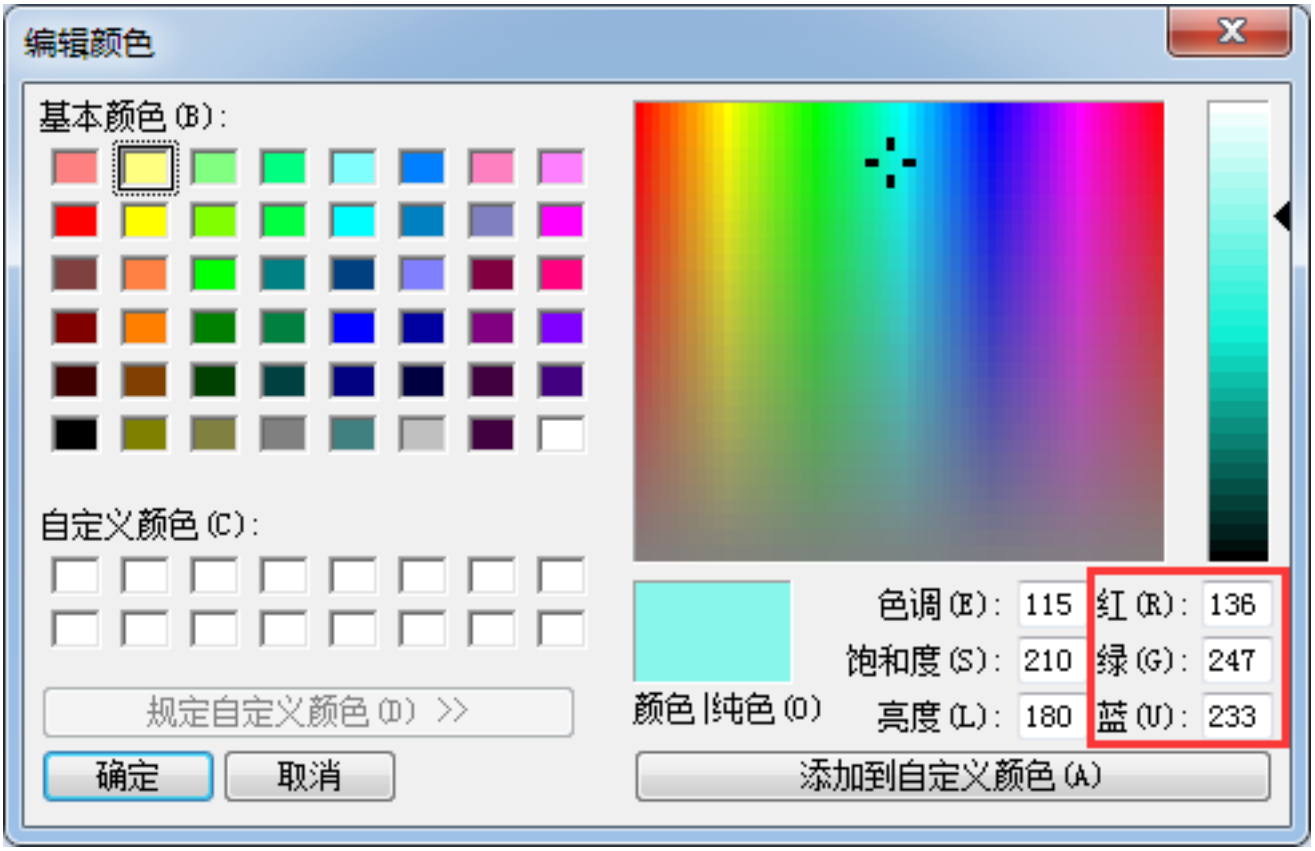
本章我们主要讲解如何控制液晶面板，液晶面板的控制信号线即图 *26_6* 中液晶面板引出的 FPC 排线，其说明见表 26-1，液晶面板通过这些信号线与液晶控制器通讯，使用这种通讯信号的被称为 RGB 接口 (RGB Interface)。

表 26-1 液晶面板的信号线

信号名称	说明
R[7:0]	红色数据
G[7:0]	绿色数据
B[7:0]	蓝色数据
CLK	像素同步时钟信号
HSYNC	水平同步信号
VSYNC	垂直同步信号
DE	数据使能信号

(1) RGB 信号线

RGB 信号线各有 8 根，分别用于表示液晶屏一个像素点的红、绿、蓝颜色分量。使用红绿蓝颜色分量来表示颜色是一种通用的做法，打开 Windows 系统自带的画板调色工具，可看到颜色的红绿蓝分量值，见图 *26_8*。常见的颜色表示会在 “RGB” 后面附带各个颜色分量值的数据位数，如 RGB565 表示红绿蓝的数据线数分别为 5、6、5 根，一共为 16 个数据位，可表示 2^{16} 种颜色；而这个液晶屏的种颜色分量的数据线都有 8 根，所以它支持 RGB888 格式，一共 24 位数据线，可表示的颜色为 2^{24} 种。



(2) 同步时钟信号 CLK

液晶屏与外部使用同步通讯方式，以 CLK 信号作为同步时钟，在同步时钟的驱动下，每个时钟传输一个像素点数据。

(3) 水平同步信号 HSYNC

水平同步信号 HSYNC(Horizontal Sync) 用于表示液晶屏一行像素数据的传输结束，每传输完成液晶屏的一行像素数据时，HSYNC 会发生电平跳变，如分辨率为 800x480 的显示屏 (800 列，480 行)，传输一帧的图像 HSYNC 的电平会跳变 480 次。

(4) 垂直同步信号 VSYNC

垂直同步信号 VSYNC(Vertical Sync) 用于表示液晶屏一帧像素数据的传输结束，每传输完成一帧像素数据时，VSYNC 会发生电平跳变。其中“帧”是图像的单位，一幅图像称为一帧，在液晶屏中，一帧指一个完整屏液晶像素点。人们常常用“帧/秒”来表示液晶屏的刷新特性，即液晶屏每秒可以显示多少帧图像，如液晶屏以 60 帧/秒的

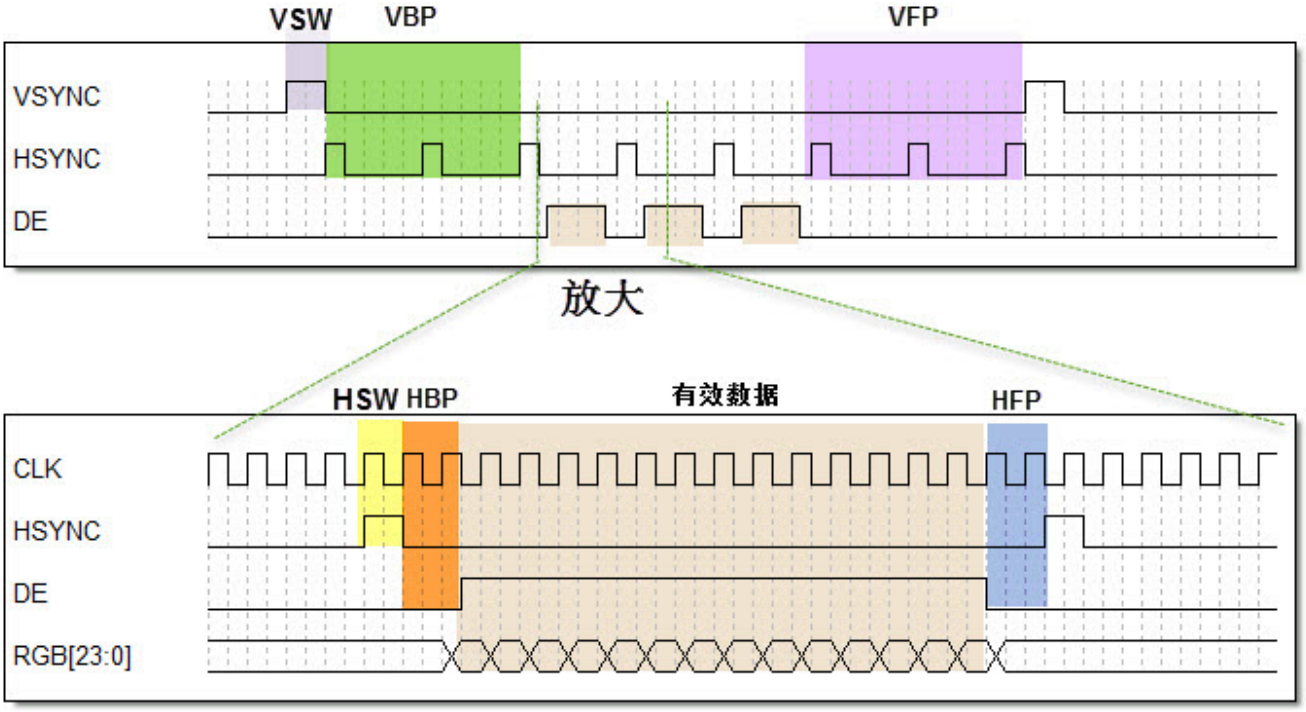
速率运行时，VSYNC 每秒钟电平会跳变 60 次。

(5) 数据使能信号 DE

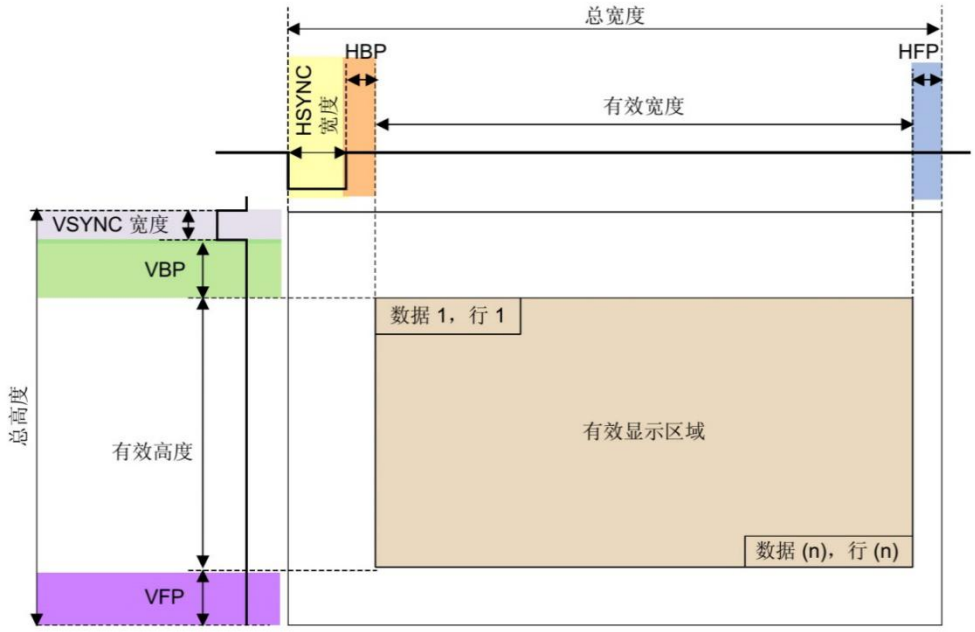
数据使能信号 DE(Data Enable) 用于表示数据的有效性，当 DE 信号线为高电平时，RGB 信号线表示的数据有效。

25.2.2 液晶数据传输时序

通过上述信号线向液晶屏传输像素数据时，各信号线的时序见图 *26_9*。图中表示的是向液晶屏传输一帧图像数据的时序，中间省略了多行及多个像素点。



液晶屏显示的图像可看作一个矩形，结合图 *26_10* 来理解。液晶屏有一个显示指针，它指向将要显示的像素。显示指针的扫描方向方向从左到右、从上到下，一个像素点一个像素点地描绘图形。这些像素点的数据通过 RGB 数据线传输至液晶屏，它们在同步时钟 CLK 的驱动下一个一个地传输到液晶屏中，交给显示指针，传输完成一行时，水平同步信号 HSYNC 电平跳变一次，而传输完一帧时 VSYNC 电平跳变一次。



但是，液晶显示指针在行与行之间，帧与帧之间切换时需要延时，而且 HSYNC 及 VSYNC 信号本身也有宽度，这些时间参数说明见表 26-2。

表 26-2 液晶通讯中的时间参数

时间参数	参数说明
VBP (vertical back porch)	表示在一帧图像开始时，垂直同步信号以后的无效的行数
VFP (vertical front porch)	表示在一帧图像结束后，垂直同步信号以前的无效的行数
HBP (horizontal back porch)	表示从水平同步信号开始到一行的有效数据开始之间的 CLK 的个数
HFP (horizontal front porth)	表示一行的有效数据结束到下一个水平同步信号开始之间的 CLK 的个数
VSW (vertical sync width)	表示垂直同步信号的宽度，单位为行
HSW (horizontal sync width)	表示水平同步信号的宽度，单位为同步时钟 CLK 的个数

在这些时间参数控制的区域，数据使能信号线“DE”都为低电平，RGB 数据线的信号无效，当“DE”为高电平时，表示的数据有效，传输的数据会直接影响液晶屏的显示区域。

25.2.3 显存

液晶屏中的每个像素点都是数据，在实际应用中需要把每个像素点的数据缓存起来，再传输给液晶屏，一般会使用 SRAM 或 SDRAM 性质的存储器，而这些专门用于存储显示数据的存储器，则被称为显存。显存一般至少要能存储液晶屏的一帧显示数据，如分辨率为 800x480 的液晶屏，使用 RGB888 格式显示，它的一帧显示数据大小为：3x800x480=1152000 字节；若使用 RGB565 格式显示，一帧显示数据大小为：2x800x480=768000 字节。

一般来说，外置的液晶控制器会自带显存，而像 STM32F429 等集成液晶控制器的芯片可使用内部 SRAM 或外扩 SDRAM 用于显存空间。

25.3 野火 3.2 寸液晶屏简介

25.3.1 3.2 寸电阻触摸屏实物

上面讲解的屏幕其液晶控制器与液晶屏是完全分离的，且具有带控制器和不带控制器的版本，易于理解，下面我们再来分析实验板标配的分辨率为 320*240 的 3.2 寸电阻触摸液晶屏，见图 *26_11*。

1
带触摸的液晶面板
（带ILI9341液晶控制器）
触摸控制器
2
液晶PCB底板
带触摸控制器
3
触摸液晶屏幕整体

图 26-11 实验板标配的 3.2 寸电阻触摸屏

图中的标号部分是液晶屏幕的整体，通过引出的排针接入到实验板上可对它进行控制，它分为标号的液晶触摸面板和标号的 PCB 底板两部分。

标号处的液晶触摸面板由液晶屏和触摸屏组成，屏幕表面的灰色线框即为电阻触摸屏的信号线，触摸屏的下方即为液晶面板，在它的内部包含了一个型号为 ILI9341 的液晶控制器芯片 (由于集成度高，所以图中无法看见)，该液晶控制器使用 8080 接口与单片机通讯，图中液晶面板引出的 FPC 信号线即 8080 接口 (RGB 接口已在内部直接与 ILI9341 相连)，且控制器中包含有显存，单片机把要显示的数据通过引出的 8080 接口发送到液晶控制器，这些数据会被存储到它内部的显存中，然后液晶控制器不断把显存的内容刷新到液晶面板，显示内容。

标号处的是 PCB 底板，它主要包含了一个电阻触摸屏的控制器 XPT2046，电阻触摸屏控制器实质上是一个 ADC 芯片，通过检测电压值来计算触摸坐标。PCB 底板与液晶触摸面板通过 FPC 排线座连接，然后引出到排针，方便与实验板的排母连接。

25.3.2 ILI9341 液晶控制器简介

本液晶屏内部包含有一个液晶控制芯片 ILI9341，它的内部结构非常复杂，见图 26-12。该芯片最主核心部分是位于中间的 GRAM(Graphics RAM)，它就是显存。GRAM 中每个存储单元都对应着液晶面板的一个像素点。它右侧的各种模块共同作用把 GRAM 存储单元的数据转化成液晶面板的控制信号，使像素点呈现特定的颜色，而像素点组合起来则成为一幅完整的图像。

框图的左上角为 ILI9341 的主要控制信号线和配置引脚，根据其不同状态设置可以使芯片工作在不同的模式，如每个像素点的位数是 6、16 还是 18 位；可配置使用 SPI 接口、8080 接口还是 RGB 接口与 MCU 进行通讯。MCU 通过 SPI、8080 接口或 RGB 接口与 ILI9341 进行通讯，从而访问它的控制寄存器 (CR)、地址计数器 (AC)、及 GRAM。

在 GRAM 的左侧还有一个 LED 控制器 (LED Controller)。LCD 为非发光性的显示装置，它需要借助背光源才能达到显示功能，LED 控制器就是用来控制液晶屏中的 LED 背光源。

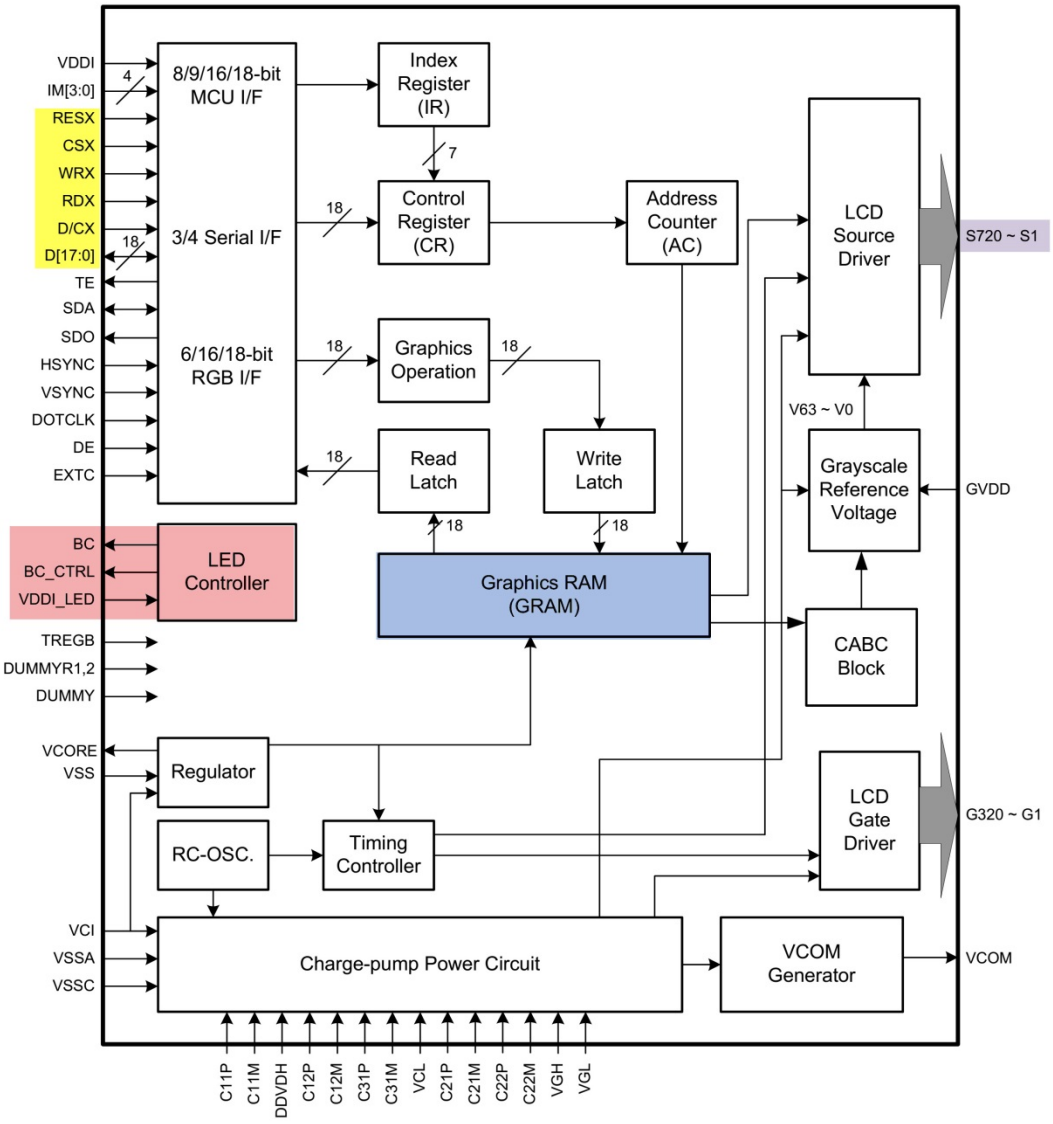


图 26-12 ILI9341 控制器内部框图

提示： 部分出厂的野火 3.2 寸液晶屏其内部所使用的 LCD 控制芯片的型号不是 ILI9341 而是 ST7789V。实际上这两个芯片是差不多的。我们在所有的 3.2 寸屏相关的例程中做了程序兼容处理，具体做法是在 LCD 初始化时在 ILI9341_REG_Config() 函数里先读取 LCD 控制芯片的 ID，并使用一个变量 lcdid 来保存它，接着具体配置 LCD 控制芯片时会根据这个 ID 的不同来进行不同的配置；有一个比较重要的函数是 ILI9341_GramScan()，在这个函数里面我们也根据变量 lcdid 做

了不同的配置，比如配置 LCD 显示的颜色模式。另外，例程中的 LCD 函数命名仍是以 ILI9341 为前缀没有修改但不影响。

25.3.3 液晶屏的信号线及 8080 时序

ILI9341 控制器根据自身的 IM[3:0] 信号线电平决定它与 MCU 的通讯方式，它本身支持 SPI 及 8080 通讯方式，本示例中液晶屏的 ILI9341 控制器在出厂前就已经按固定配置好 (内部已连接硬件电路)，它被配置为通过 8080 接口通讯，使用 16 根数据线的 RGB565 格式。内部硬件电路连接完，剩下的其它信号线被引出到 FPC 排线，最后该排线由 PCB 底板引出到排针，排针再与实验板上的 STM32 芯片连接，引出的排针信号线见图 *26_13*。

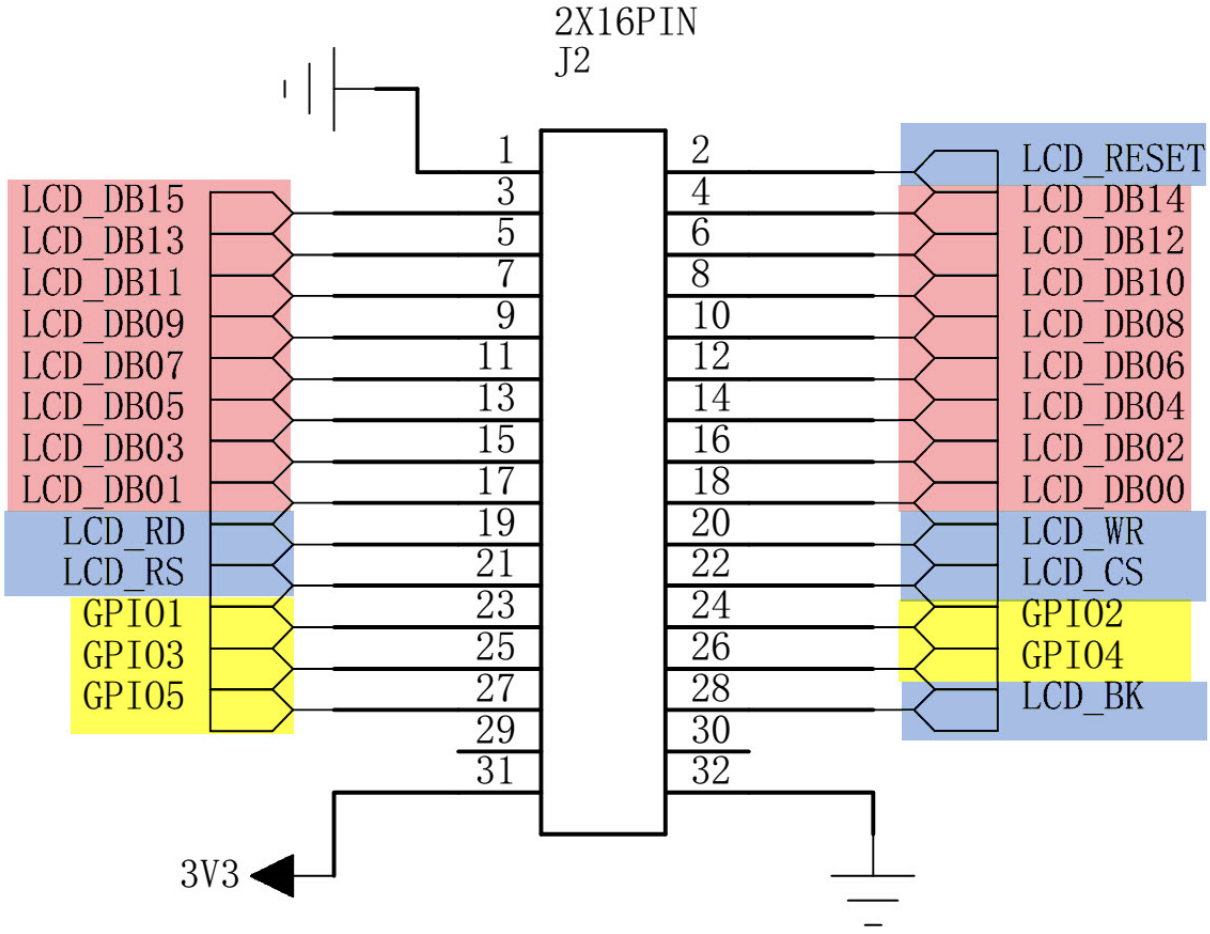


图 26-13 液晶屏引出的信号线

这些信号线的说明见表 26-3。

表 26-3 液晶屏引出的信号线说明

信号线	ILI9341 对应的信号线	说明
LCD_DB[15:0]	D[15:0]	数据信号
LCD_RD	RDX	读数据信号，低电平有效
LCD_RS	D/CX	数据/命令信号，高电平时，D[15:0] 表示的是数据 (RGB 像素数据或命令数据)，低电平时 D[15:0] 表示控制命令
LCD_RESET	RESX	复位信号，低电平有效
LCD_WR	WRX	写数据信号，低电平有效
LCD_CS	CSX	片选信号，低电平有效
LCD_BK	•	背光信号，低电平点亮
GPIO[5:1]	•	触摸屏的控制信号线，下一章再介绍

这些信号线即 8080 通讯接口，带 X 的表示低电平有效，STM32 通过该接口与 ILI9341 芯片进行通讯，实现对液晶屏的控制。通讯的内容主要包括命令和显存数据，显存数据即各个像素点的 RGB565 内容；命令是指对 ILI9341 的控制指令，MCU 可通过 8080 接口发送命令编码控制 ILI9341 的工作方式，例如复位指令、设置光标指令、睡眠模式指令等等，具体的指令在《ILI9341.pdf》数据手册均有详细说明。写命令时序图见图 *26_14*。

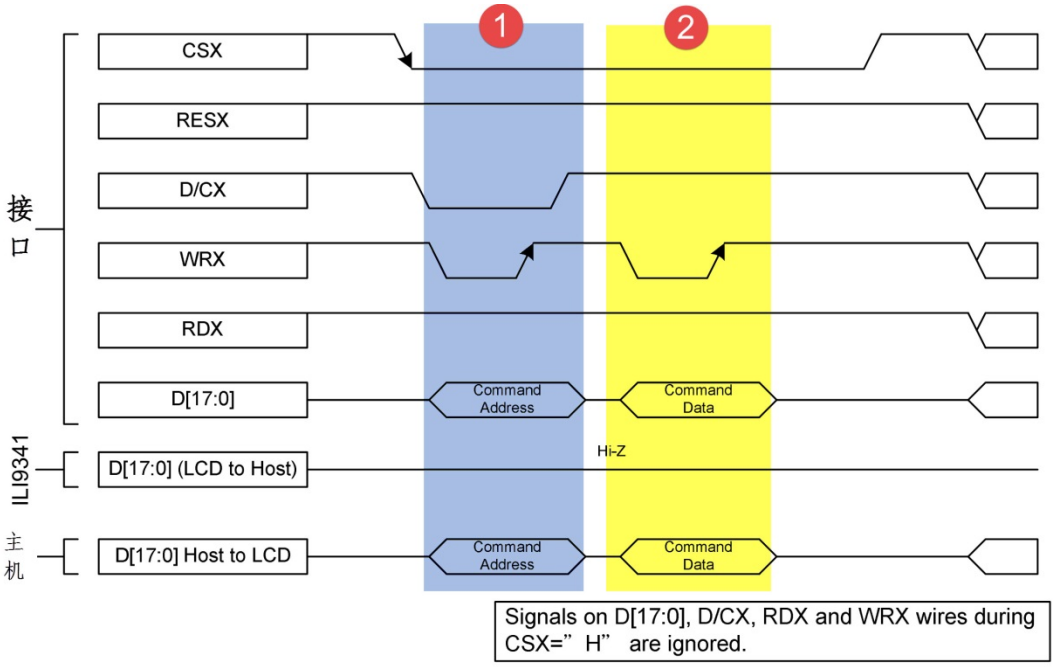


图 26-14 使用 18 条数据线的 8080 接口写命令时序

由图可知，写命令时序由片选信号 CSX 拉低开始，对数据/命令选择信号线 *D/CX* 也置低电平表示写入的是命令地址 (可理解为命令编码，如软件复位命令：0x01)，以写信号 WRX 为低，读信号 RDX 为高表示数据传输方向为写入，同时，在数据线 D[17:0](或 D[15:0]) 输出命令地址，在第二个传输阶段传送的是命令的参数，所以 *D/CX 要置高电平 *，表示写入的是命令数据，命令数据是某些指令带有的参数，如复位指令编码为 0x01，它后面可以带一个参数，该参数表示多少秒后复位 (实际的复位命令不含参数，此处只是为了讲解指令编码与参数的区别)。

当需要把像素数据写入 GRAM 时，过程很类似，把片选信号 CSX 拉低后，再把数据/命令选择信号线 *D/CX* 置为高电平，这时由 D[17:0] 传输的数据则会被 ILI9341 保存至它的 GRAM 中。

25.4 使用 STM32 的 FSMC 模拟 8080 接口时序

ILI9341 的 8080 通讯接口时序可以由 STM32 使用普通 I/O 接口进行模拟，但这样效率太低，STM32 提供了一种特别的控制方法——使用 FSMC 接口实现 8080 时序。

25.4.1 FSMC 简介

STM32F1 系列芯片使用 FSMC 外设来管理扩展的存储器，FSMC 是 Flexible Static Memory Controller 的缩写，译为灵活的静态存储控制器。它可以用于驱动包括 SRAM、NOR FLASH 以及 NAND FLSAH 类型的存储器，不能驱动如 SDRAM 这种动态的存储器而在 STM32F429 系列的控制器中，它具有 FMC 外设，支持控制 SDRAM 存储器。

由于 FSMC 外设可以用于控制扩展的外部存储器，而 MCU 对液晶屏的操作实际上就是把显示数据写入到显存中，与控制存储器非常类似，且 8080 接口的通讯时序完全可以使用 FSMC 外设产生，因而非常适合使用 FSMC 控制液晶屏。

FSMC 外设的结构见图 *27_15*。

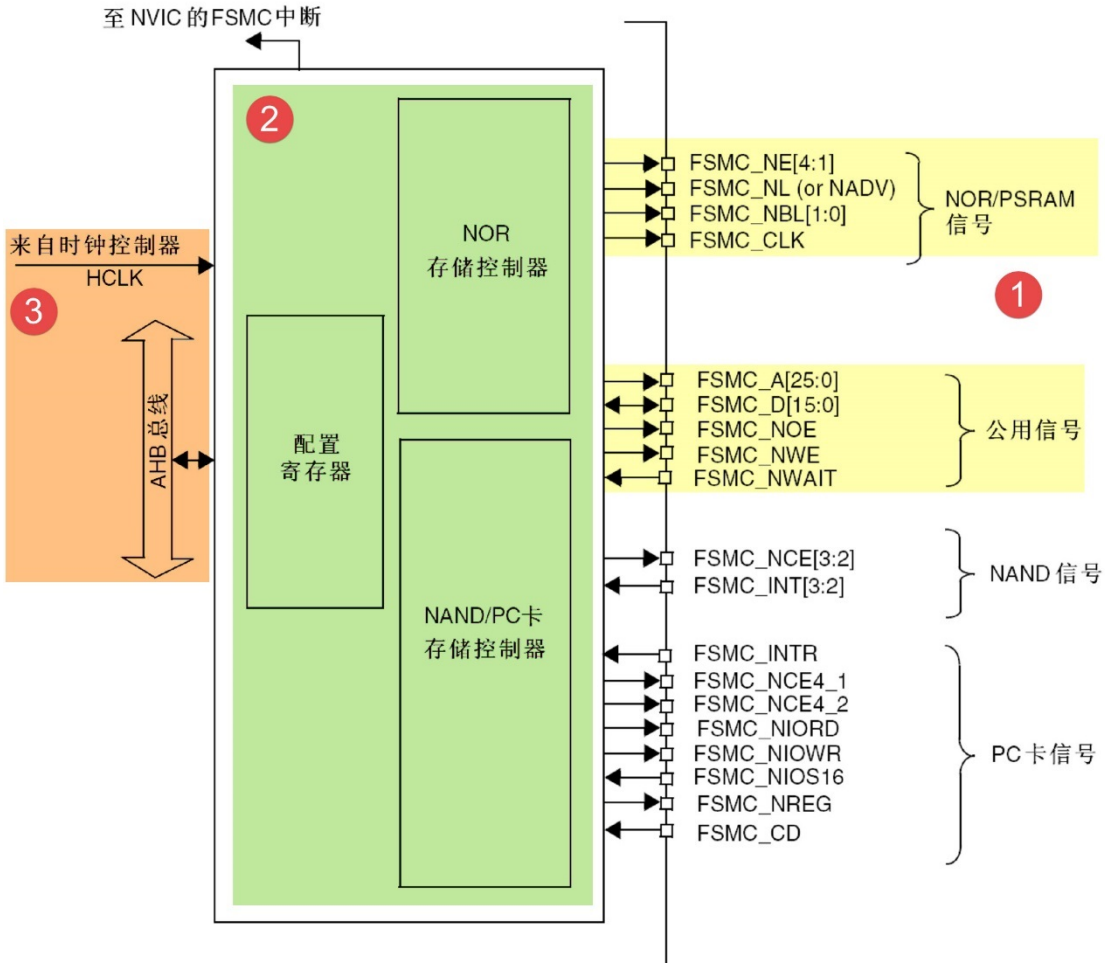


图 27-15 FSMC 结构图

25.4.1.1 通讯引脚

在框图的右侧是 FSMC 外设相关的控制引脚，由于控制不同类型存储器的时候会有一些不同的引脚，看起来有非常多，其中地址线 FSMC_A 和数据线 FSMC_D 是所有控制器都共用的。这些 FSMC 引脚具体对应的 GPIO 端口及引脚号可在《STM32F103 规格书》中搜索查找到，不在此列出。

在本章示例中，控制 LCD 时，是使用 FSMC 的 NORPSRAM 模式的，而且控制 LCD 时使用的是 NOR FLASH 类型的模式 B，所以我们重点分析框图中 NOR FLASH 控制信号线部分，控制 NOR FLASH 主要使用到表 27-4 的信号线：

表 27-4 FSMC 控制 NOR FLASH 的信号线

FSMC 信号名称	信号方向	功能
CLK	输出	时钟 (同步突发模式使用)
A[25:0]	输出	地址总线
D[15:0]	输入/输出	双向数据总线
NE[x]	输出	片选，x = 1…4
NOE	输出	输出使能
NWE	输出	写使能
NWAIT	输入	NOR 闪存要求 FSMC 等待的信号
NADV	输出	地址、数据线复用时作锁存信号

在控制 LCD 时，使用的是类似异步、地址与数据线独立的 NOR FLASH 控制方式，所以实际上 CLK、NWAIT、NADV 引脚并没有使用到。

其中比较特殊的 FSMC_NE 是用于控制存储器芯片的片选控制信号线，STM32 具有 FSMC_NE1/2/3/4 号引脚，不同的引脚对应 STM32 内部不同的地址区域。例如，当 STM32 访问 0x68000000-0x6BFFFFFF 地址空间时，FSMC_NE3 引脚会自动设置为低电平，由于它一般连接到外部存储器的片选引脚且低电平有效，所以外部存储器的片选被使能，而访问 0x60000000-0x63FFFFFF 地址时，FSMC_NE1 会输出低电平。当使用不同的 FSMC_NE 引脚连接外部存储器

时，STM32 访问外部存储的地址不一样，从而达到控制多个外部存储器芯片的目的。各引脚对应的地址会在后面“FSMC 的地址映射”小节讲解。

25.4.1.2 存储器控制器

上面不同类型的引脚是连接到 FSMC 内部对应的存储控制器中的。NOR/PSRAM/SRAM 设备使用相同的控制器，NAND/PC 卡设备使用相同的控制器，不同的控制器有专用的寄存器用于配置其工作模式。

控制 NOR FLASH 的有 FSMC_BCR1/2/3/4 控制寄存器、FSMC_BTR1/2/3/4 片选时序寄存器以及 FSMC_BWTR1/2/3/4 写时序寄存器。每种寄存器都有 4 个，分别对应于 4 个不同的存储区域，各种寄存器介绍如下：

- FSMC_BCR 控制寄存器可配置要控制的存储器类型、数据线宽度以及信号有效极性能参数。
- FMC_BTR 时序寄存器用于配置 SRAM 访问时的各种时间延迟，如数据保持时间、地址保持时间等。
- FMC_BWTR 写时序寄存器与 FMC_BTR 寄存器控制的参数类似，它专门用于控制写时序的时间参数。

25.4.1.3 时钟控制逻辑

FSMC 外设挂载在 AHB 总线上，时钟信号来自于 HCLK(默认 72MHz)，控制器的同步时钟输出就是由它分频得到。例如，NOR 控制器的 FSMC_CLK 引脚输出的时钟，它可用于与同步类型的 NOR FLASH 芯片进行同步通讯，它的时钟频率可通过 FSMC_BTR 寄存器的 CLKDIV 位配置，可以配置为 HCLK 的 1/2 或 1/3，也就是说，若它与同步类型的 NOR FLASH 通讯时，同步时钟最高频率为 36MHz。本示例中的 NOR FLASH 为异步类型的存储器，不使用同步时钟信号，所以时钟分频配置不起作用。

25.5 FSMC 的地址映射

FSMC 连接好外部的存储器并初始化后，就可以直接通过访问地址来读写数据，这种地址访问与 I2C EEPROM、SPI FLASH 的不一样，后两种方式都需要控制 I2C 或 SPI 总线给存储器发送地址，然后获取数据；在程序里，这个地址和数据都需要分开使用不同的变量存储，并且访问时还需要使用代码控制发送读写命令。而使用 FSMC 外接存储器时，其存储单元是映射到 STM3 2 的内部寻址空间的；在程序里，定义一个指向这些地址的指针，然后就可以通过指针直接修改该存储单元的内容，FSMC 外设会自动完成数据访问过程，读写命令之类的操作不需要程序控制，访问示例代码见*代码清单 26_7_1*。

列表 1: 代码清单 27-1 使用 FSMC 访问外部存储器示例代码

```
#define Bank1_SRAM3_ADDR    ((uint32_t)(0x68000000))

/* 写/读 16 位数据 */
*( uint16_t*) (Bank1_SRAM3_ADDR+10 ) = (uint16_t)0xBBBB;
printf("指针访问 SRAM, 写入数据 0xBBBB \r\n");

temp =  *( uint16_t*) (Bank1_SRAM3_ADDR+10 );
printf("读取数据: 0x%X \r\n",temp);
```

以上代码实际上就是标准的 C 语言对特定地址的指针式访问，只是由于该地址被 STM32 映射到 FSMC 外设，所以访问这些地址时，FSMC 会自动输出地址、数据等访问时序。

FSMC 的地址映射见图 *27_16*。

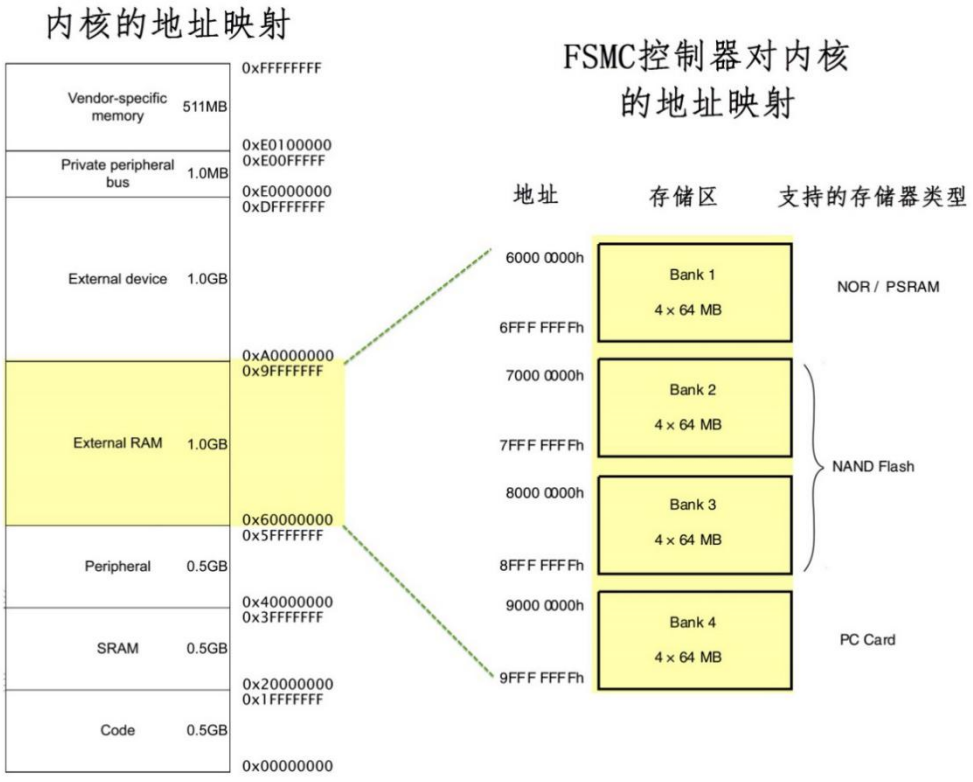


图 27-16 FSMC 的地址映射

图中左侧的是 Cortex-M3 内核的存储空间分配，右侧是 STM32 FSMC 外设的地址映射。可以看到 FSMC 的 NOR/PSRAM/SRAM/NAND FLASH 以及 PC 卡的地址都在 External RAM 地址空间内。正是因为存在这样的地址映射，使得访问 FSMC 控制的存储器时，就跟访问 STM32 的片上外设寄存器一样 (片上外设的地址映射即图中左侧的“Peripheral”区域)。

FSMC 把整个 External RAM 存储区域分成了 4 个 Bank 区域，并分配了地址范围及适用的存储器类型，如 NOR 及 SRAM 存储器只能使用 Bank1 的地址。在每个 Bank 的内部又分成了 4 个小块，每个小块有相应的控制引脚用于连接片选信号，如 FSMC_NE[4:1] 信号线可用于选择 BANK1 内部的 4 小块地址区域，见图 *27_17*，当 STM32 访问 0x6C000000-0x6FFFFFFF 地址空间时，会访问到 Bank1 的第 1 小块区域，相应的 FSMC_NE1 信号线会输出控制信号。

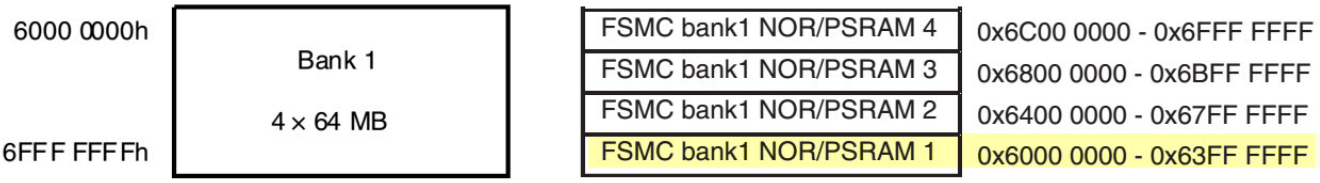


图 27-17 Bank1 内部的小块地址分配

25.6 FSMC 控制异步 NOR FLASH 的时序

FSMC 外设支持输出多种不同的时序以便于控制不同的存储器，它具有 ABCD 四种模式，下面我们仅针对控制异步 NOR FLASH 使用的模式 B 进行讲解，见图 *27_19*。

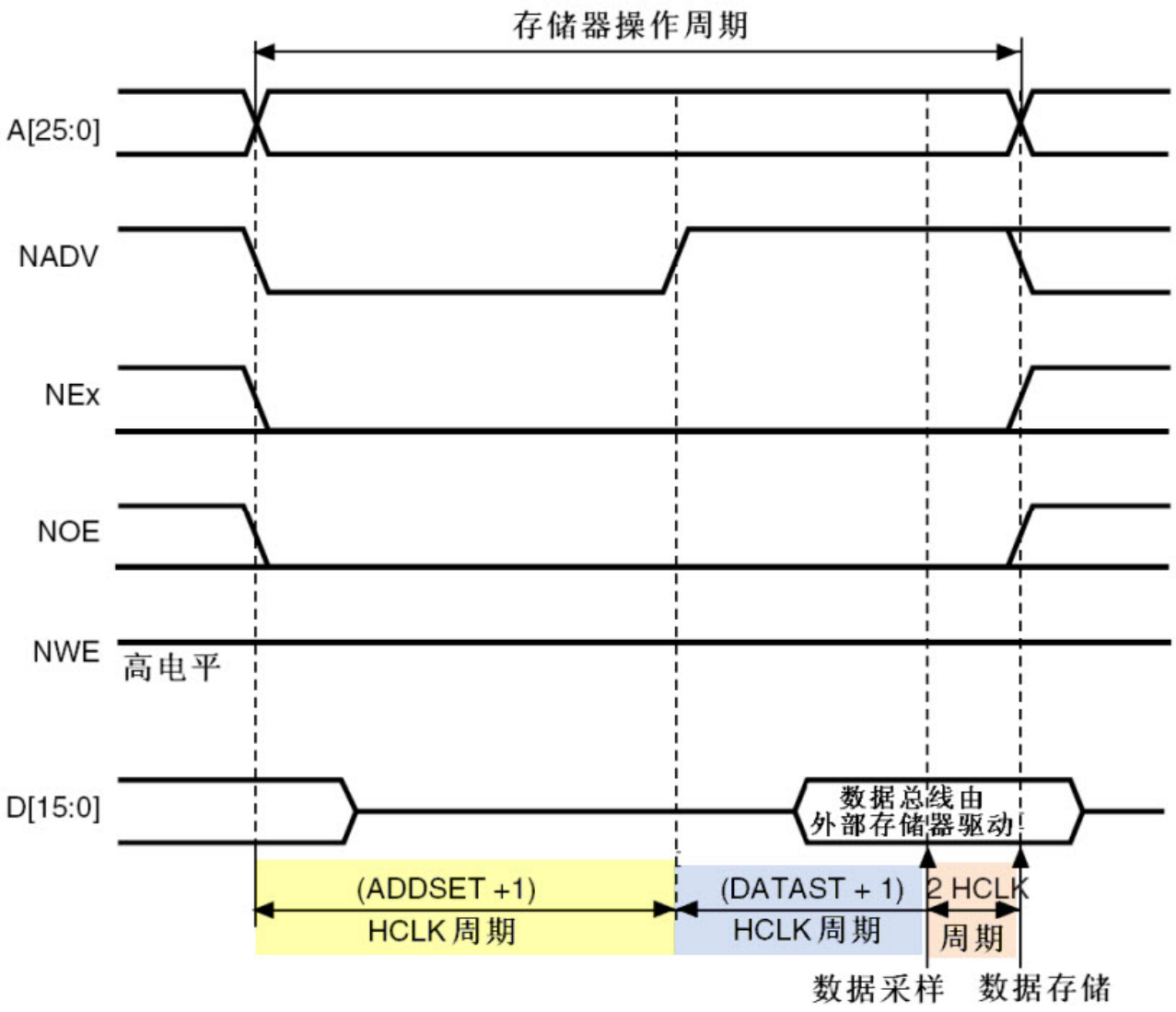


图 27-18 FSMC 读 NOR FLASH 的时序图（模式 B）

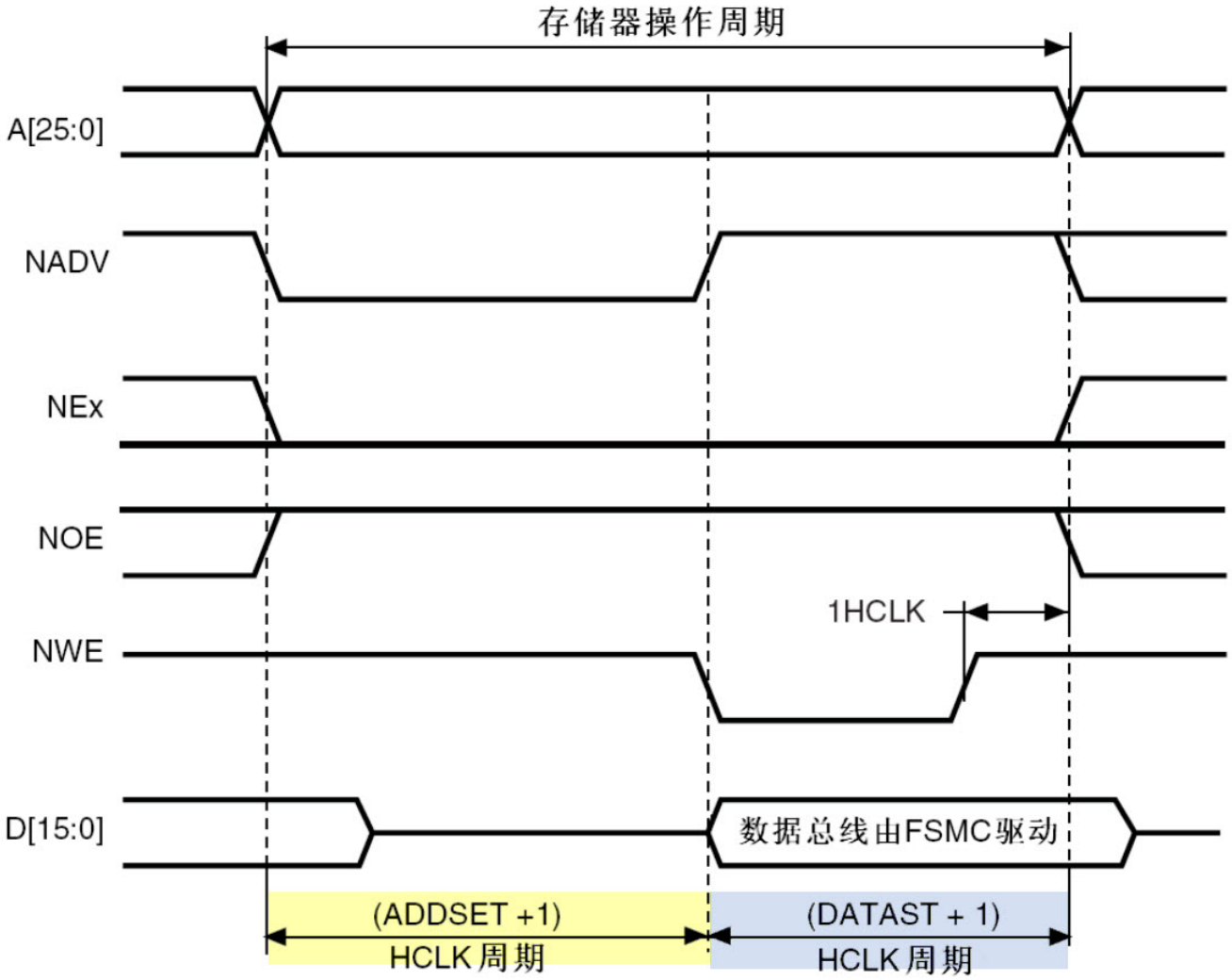


图 27-19 FSMC 写 NOR FLASH 的时序图（模式 B）

当内核发出访问某个指向外部存储器地址时，FSMC 外设会根据配置控制信号线产生时序访问存储器，上图中的是访问外部异步 NOR FLASH（模式 B）时 FSMC 外设的读写时序。

以读时序为例，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) 以及 2 个 HCLK 周期组成。在地址建立周期中，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。

写时序类似，区别是它的一个存储器操作周期仅由地址建立周期 (ADDSET) 和数据建立周期

(DATAST) 组成，且在数据建立周期期间写使能信号线发出写信号，接着 FSMC 把数据通过数据线传输到存储器中。

根据 STM32 对寻址空间的地址映射，地址 *0x6000 0000 ~0x9FFF FFFF* 是映射到外部存储器的，而其中的 0x6000 0000 ~0x6FFF FFFF 则是分配给 NOR FLASH、PSRAM 这类可直接寻址的器件。

当 FSMC 外设被配置成正常工作，并且外部接了 NOR FLASH 时，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* 选择相应的 NOR 芯片，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 NOR FLASH 中了。

25.7 用 FSMC 模拟 8080 时序

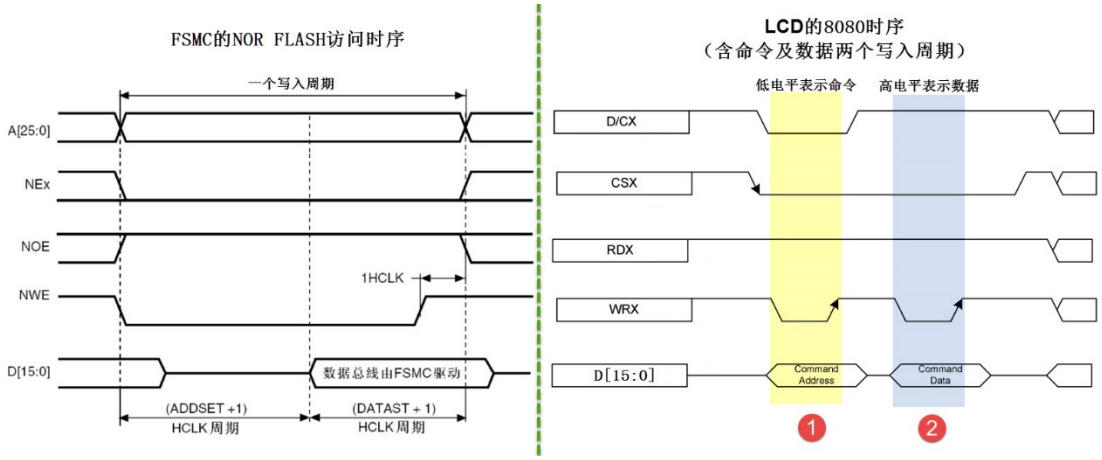


图 27-20FSMC 模式 B 时序与 8080 时序对比（写过程）

见表 27-5，对比 FSMC NOR/PSRAM 中的模式 B 时序与 ILI9341 液晶控制器芯片使用的 8080 时序可发现，这两个时序是十分相似的 (除了 FSMC 的地址线 A 和 8080 的 D/CX 线，可以说是完全一样)，它们的信号线对比见表 27-5。

表 27-5 FSMC 的 NOR 与 8080 信号线对比

FSMC-NOR 信号线	功能	8080 信号线	功能
NEx	片选信号	CSX	片选信号
NWR	写使能	WRX	写使能
NOE	读使能	RDX	读使能
D[15:0]	数据信号	D[15:0]	数据信号
A[25:0]	地址信号	D/CX	数据/命令选择

对于 FSMC 和 8080 接口，前四种信号线都是完全一样的，仅仅是 FSMC 的地址信号线 A[25:0] 与 8080 的数据/命令选择线 D/CX 有区别。而对于 D/CX 线，它为高电平的时候表示数值，为低电平的时候表示命令，如果能使用 FSMC 的 A 地址线根据不同的情况产生对应的电平，那么就完全可以使用 FSMC 来产生 8080 接口需要的时序了。

为了模拟出 8080 时序，我们可以把 FSMC 的 *A0* 地址线 (也可以使用其它 A1/A2 等地址线) 与 ILI9341 芯片 8080 接口的 *D/CX 信号线连接 *，那么当 A0 为高电平时 (即 D/CX 为高电平)，数据线 D[15:0] 的信号会被 ILI9341 理解为数值，若 A0 为低电平时 (即 D/CX 为低电平)，传输的信号则会被理解为命令。

由于 FSMC 会自动产生地址信号，当使用 FSMC 向 0x6xxx xxx1、0x6xxx xxx3、0x6xxx xxx5…这些奇数地址写入数据时，地址最低位的值均为 1，所以它会控制地址线 A0(D/CX) 输出高电平，那么这时通过数据线传输的信号会被理解为数值；若向 0x6xxx xxx0 、0x6xxx xxx2、0x6xxx xxx4…这些偶数地址写入数据时，地址最低位的值均为 0，所以它会控制地址线 A0(D/CX) 输出低电平，因此这时通过数据线传输的信号会被理解为命令，见表 27-6。

表 27-6 使用 FSMC 输出地址示例

地址	地址的二进制值 (仅列出低四位)	A0(D/CX) 的电平	控制 ILI9341 时的意义
0x6xxx xxx1	000 1	1 高电平	D 数值
0x6xxx xxx3	001 1	1 高电平	D 数值
0x6xxx xxx5	010 1	1 高电平	D 数值
0x6xxx xxx0	000 0	0 低电平	C 命令
0x6xxx xxx2	001 0	0 低电平	C 命令
0x6xxx xxx4	010 0	0 低电平	C 命令

有了这个基础，只要配置好 FSMC 外设，然后在代码中利用指针变量，向不同的地址单元写入数据，就能够由 FSMC 模拟出的 8080 接口向 ILI9341 写入控制命令或 GRAM 的数据了。

注意：在实际控制时，以上地址计算方式还不完整，还需要注意 HADDR 内部地址与 FSMC 地址信号线的转换，关于这部分内容在代码讲解时再详细举例说明。

25.8 NOR FLASH 时序结构体

在讲解程序前，再来了解一下与 FSMC NOR FLASH 控制相关的结构体。

与控制 SRAM 时一样，控制 FSMC 使用 NOR FLASH 存储器时主要是配置时序寄存器以及控制寄存器，利用 HAL 库的时序结构体以及初始化结构体可以很方便地写入参数。

NOR FLASH 时序结构体的成员见*代码清单 26_1*。

列表 2: 代码清单 26_1 NOR FLASH 时序结构体
FSMC_NORSRAMTimingInitTypeDef

```
typedef struct
{
uint32_t AddressSetupTime;        /* 地址建立时间，0-0xF 个 HCLK 周期 */
uint32_t AddressHoldTime;         /* 地址保持时间，0-0xF 个 HCLK 周期 */
uint32_t DataSetupTime;           /* 地址建立时间，0-0xF 个 HCLK 周期 */
uint32_t BusTurnAroundDuration; /* 总线转换周期,0-0xF 个 HCLK 周期，在 NOR␣
↪FLASH */
uint32_t CLKDivision;   /* 时钟分频因子,1-0xF，若控制异步存储器，本参数无效 */
uint32_t DataLatency;     /* 数据延迟时间，若控制异步存储器，本参数无效 */
uint32_t AccessMode;              /* 设置访问模式 */
}FSMC_NORSRAMTimingInitTypeDef;
```

(1) AddressSetupTime

本成员设置地址建立时间，即 FSMC 读写时序图 26-16 中的 ADDSET 值，它可以被设置为 0-0xF 个 HCLK 周期数，按 STM32HAL 库的默认配置，HCLK 的时钟频率为 72MHz，即一个 HCLK 周期为 1/72 微秒。

(2) DataSetupTime

本成员设置数据建立时间，即 FSMC 读写时序图 26-16 中的 DATAST 值，它可以被设置为 0-0xF 个 HCLK 周期数。

(3) AccessMode

本成员设置存储器访问模式，不同的模式下 FSMC 访问存储器地址时引脚输出的时序不一样，可选 FSMC_ACCESS_MODE_A/B/C/D 模式。控制异步 NOR FLASH 时使用 B 模式。

这个 FSMC_NORSRAM_TimingTypeDef 时序结构体配置的延时参数，将作为下一节的 FSMC NOR FLASH 初始化结构体的一个成员。

25.9 FSMC 初始化结构体

FSMC 控制 NOR FLASH 相关的结构体，初始化结构体见代码清单 *26_2*。

列表 3: 代码清单 26_2 NOR FLASH 初始化结构体 FSMC_NORSRAMInitTypeDef

```
  /**
  * @brief  FSMC NOR/SRAM Init structure definition
  */
  typedef struct
  {
  uint32_t NSBank;                  /* 设置要控制的 Bank 区域 */
  uint32_t DataAddressMux;          /* 设置地址总线与数据总线是否复用 */
  uint32_t MemoryType;             /* 设置存储器的类型 */
  uint32_t MemoryDataWidth;        /* 设置存储器的数据宽度 */
  uint32_t BurstAccessMode;          /* 设置是否支持突发访问模式，只支持同步类型的存
储器 */
  uint32_t WaitSignalPolarity;    /* 设置等待信号的极性 */
  uint32_t WrapMode;               /* 设置是否支持对齐的突发模式 */
  uint32_t WaitSignalActive;       /* 配置等待信号在等待前有效还是等待期间有效 */
```

(下页继续)

(续上页)

```
  uint32_t WriteOperation;        /* 设置是否写使能 */
  uint32_t WaitSignal;            /* 设置是否使能等待状态插入 */
  uint32_t ExtendedMode;          /* 设置是否使能扩展模式 */
  uint32_t WriteBurst;            /* 设置是否使能写突发操作 */
  uint32_t AsynchronousWait;      /* 设置是否使能等待信号 */
  uint32_t ContinuousClock;       /* 设置是否使能 FMC 时钟输出到外部存储设备 */
  uint32_t WriteFifo;             /* 设置是否使能 FIFO*/
  uint32_t PageSize;              /* 指定页的大小 */
  /* 当不使用扩展模式时，本参数用于配置读写时序，否则用于配置读时序 */
  FSMC_NORSRAM_TimingTypeDef* FSMC_ReadWriteTimingStruct;
  /* 当使用扩展模式时，本参数用于配置写时序 */
  FSMC_NORSRAM_TimingTypeDef * FSMC_WriteTimingStruct;
  } FSMC_NORSRAM_InitTypeDef;
```

这部分内容与 FSMC 控制 SRAM 时完全一致，此处仅列出与 NOR FLASH 模式 B 相关的一些结构体成员的说明：

(1) NSBank

本成员用于选择 FSMC 映射的存储区域，它的可选参数以及相应的内核地址映射范围见**错误! 未找到引用源**。。

表 26-7 可以选择的存储器区域及区域对应的地址范围

可以输入的宏	对应的地址区域
FSMC_Bank1_NORSRAM1	0x60000000-0x63FFFFFF
FSMC_Bank1_NORSRAM2	0x64000000-0x67FFFFFF
FSMC_Bank1_NORSRAM3	0x68000000-0x6BFFFFFF
FSMC_Bank1_NORSRAM4	0x6C000000-0x6FFFFFFF

(2) MemoryType

本成员用于设置要控制的存储器类型，它支持控制的存储器类型为 SRAM、PSRAM 以及 NOR FLASH。

(3) MemoryDataWidth

本成员用于设置要控制的存储器的数据宽度，可选择设置成 8 或 16 位。

(4) WriteOperation

这个成员用于设置是否写使能，禁止写使能的话 FSMC 只能从存储器中读取数据，不能写入。

(5) ExtendedMode

本成员用于设置是否使用扩展模式，在非扩展模式下，对存储器读写的时序都只使用 FSMC_BCR 寄存器中的配；在扩展模式下，对存储器的读写时序可以分开配置，读时序使用 FSMC_BCR 寄存器，写时序使用 FSMC_BWTR 寄存器的配置。

25.10 FSMC—液晶显示实验

本小节讲解如何使用 FSMC 外设控制实验板配套的 3.2 寸 ILI9341 液晶屏，见图 *26_18*，该液晶屏的分辨率为 320x240，支持 RGB565 格式。

学习本小节内容时，请打开配套的“固件库例程液晶显示程序液晶显示”工程配合阅读。

25.10.1 硬件设计

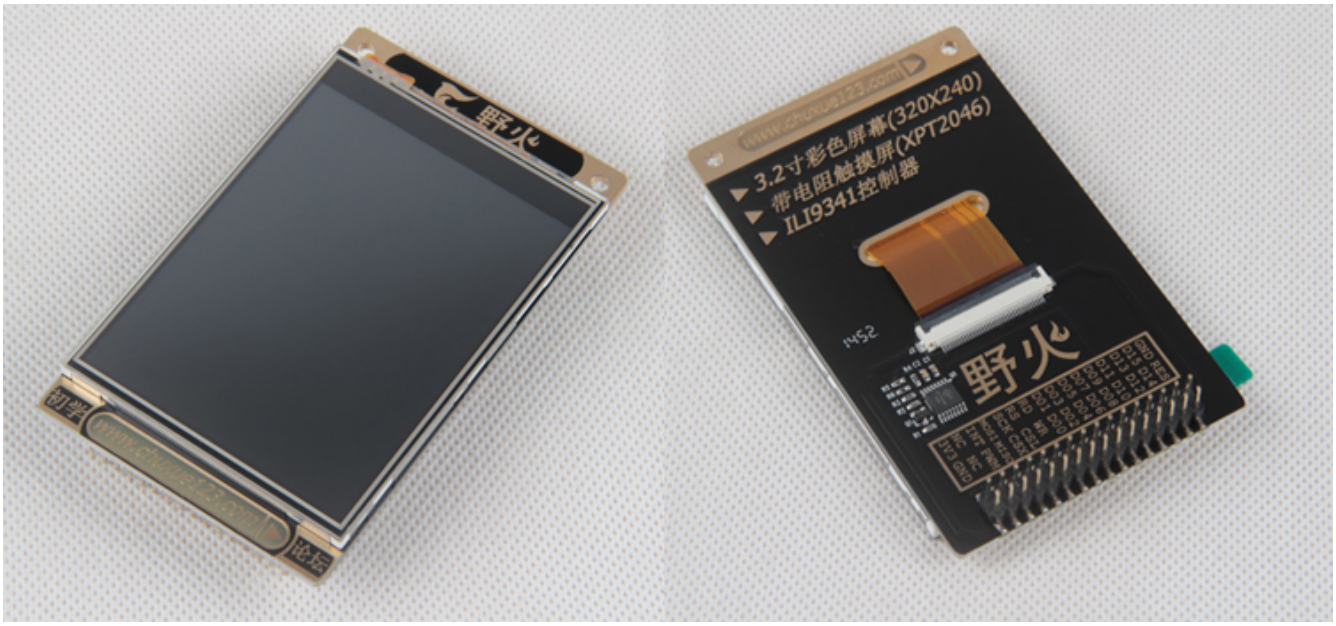


图 26-18 液晶屏实物图

图 26-18 液晶屏背面的 PCB 电路对应图 *26_19*、图 *26_20*、图 *26_21* 中的原理图，分别是屏幕 PCB 底板原理图、触摸部分原理图、液晶排针接口线序图。

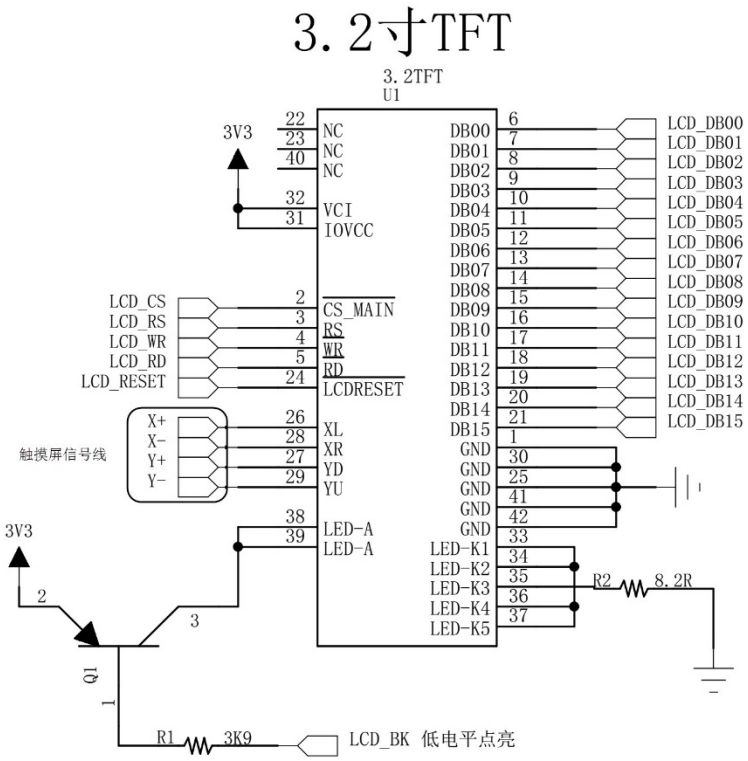


图 26-19 屏幕 PCB 底板原理图 (截图于《3.2 寸液晶原理图.pdf》)

屏幕的 PCB 底板引出的信号线会通过 PCB 底板上的 FPC 接口与液晶面板连接，这些信包括液晶控制相关的 CS、RS 等信号及 DB0-DB15 数据线，其中 RS 引脚以高电平表示传输数据，低电平表示传输命令；另外还有引出 LCD_BK 引脚用于控制屏幕的背光供电，可以通过该引脚控制背光的强度，该引脚为低电平时打开背光。图中的 X+/X-/Y+/Y-引脚是液晶面板上触摸屏引出的信号线，它们会被连接到 PCB 底板的电阻触摸屏控制器，用于检测触摸信号，其原理图见图 *26_20*

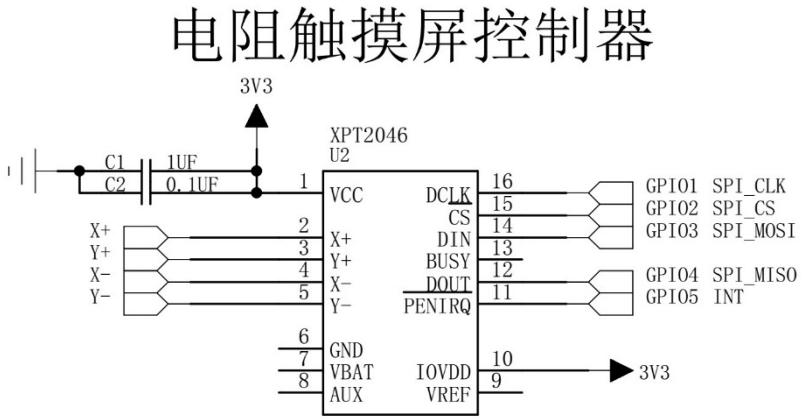


图 26-20 屏幕 PCB 底板的触摸部分原理图 (截图于《3.2 寸液晶原理图.pdf》)

触摸检测的主体是型号为 XPT2046 的芯片，它接收触摸屏的 X+/X-/Y+/Y-信号进行处理，把触摸信息使用 SPI 接口输出到 STM32 等控制器，在触摸屏章节将会详细讲解其检测原理。

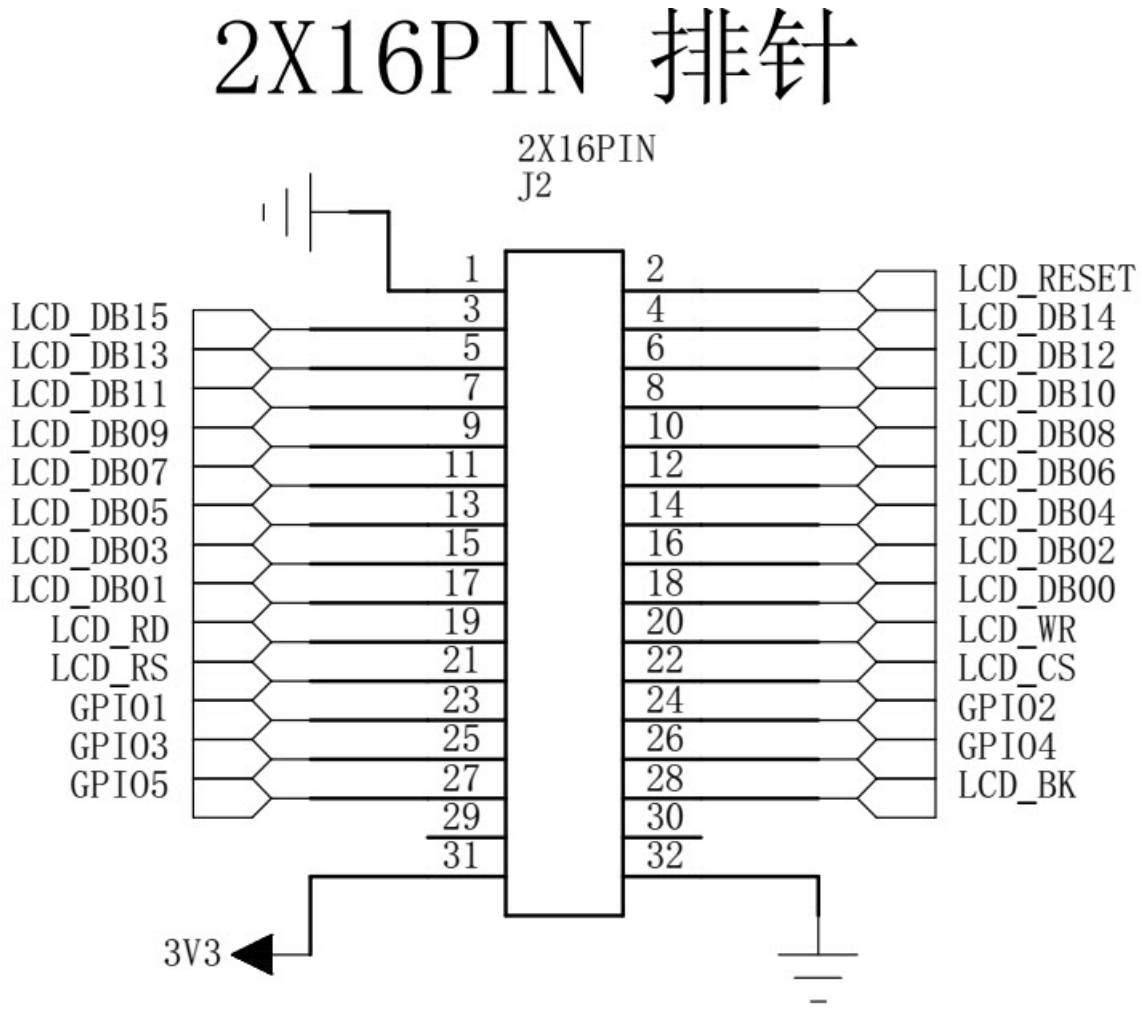


图 26-21 液晶屏接口 (截图于《3.2 寸液晶原理图.pdf》)

图 *26_22* 表示的是 PCB 底板引出的排针线序，屏幕整体通过这些引出的排针与开发板或其它控制器连接。

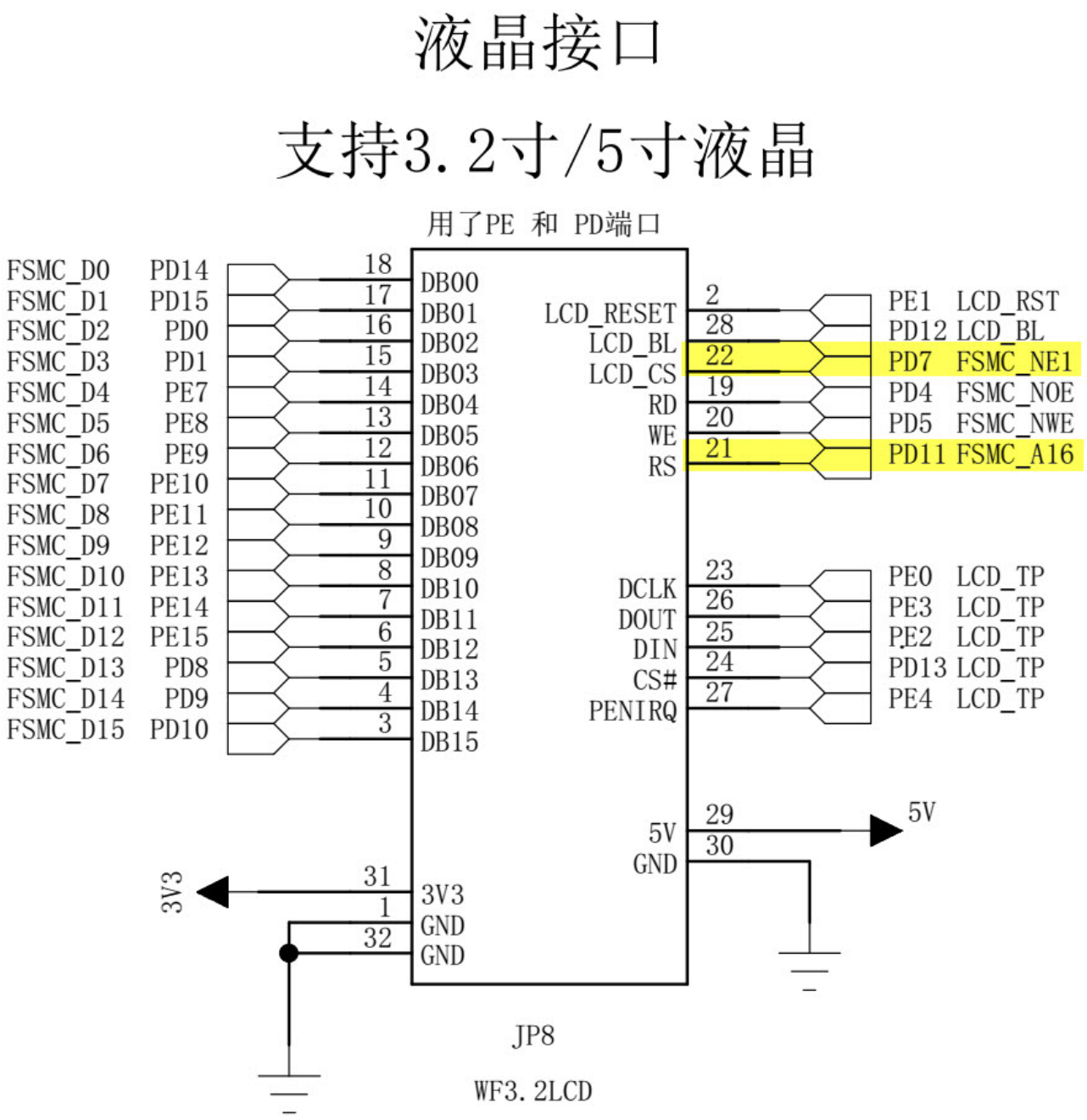


图 26-22 开发板与屏幕的连接的信号说明 (截图于《指南者开发板原理图.pdf》)

图 *26_22* 是指南者开发板上的液晶排母接口原理图，它说明了配套的 3.2 寸屏幕接入到开发板上时的信号连接关系。其中请着重关注图中液晶屏 LCD_CS 及 LCD_RS(即 DC 引脚) 与 FSMC 存储区选择引脚 FSMC_NE 及地址信号 FSMC_A 的编号，它们会决定 STM32 要使用什么内存地址来

控制与液晶屏的通讯。

以上原理图可查阅《3.2 寸液晶原理图.pdf》及《指南者开发板原理图.pdf》文档获知，若您使用的液晶屏或实验板不一样，请根据实际连接的引脚修改程序。

25.10.2 软件设计

为了使工程更加有条理，我们把 LCD 控制相关的代码独立分开存储，方便以后移植。在“USART—串口通讯”工程的基础上新建“bsp_ili9341_lcd.c”及“bsp_ili9341_lcd.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

25.10.2.1 编程要点

(1) 初始化通讯使用的目标引脚及端口时钟；

(2) 使能 FSMC 外设的时钟；

(3) 配置 FSMC 为异步 NOR FLASH 模式以仿真 8080 时序；

(4) 建立机制使用 FSMC 向液晶屏发送命令及数据；

(5) 发送控制命令初始化液晶屏；

(6) 编写液晶屏的绘制像素点函数；

(7) 利用描点函数制作各种不同的液晶显示应用。

25.10.2.2 代码分析

25.10.2.2.1 液晶 LCD 硬件相关宏定义

我们把 FSMC 控制液晶屏硬件相关的配置都以宏的形式定义到“bsp_ili9341_lcd.h”文件中，见*代码清单 26_3*。

列表 4: 代码清单 26-3 FSMC 硬件配置相关的宏 (省略了部分数据线)

```
/****** 控制信号线 ******/
//片选，选择 NOR/SRAM 块
#define      ILI9341_CS_PORT                 GPIOD
#define      ILI9341_CS_PIN                  GPIO_PIN_7

//DC 引脚，使用 FSMC 的地址信号控制，本引脚决定了访问 LCD 时使用的地址

#define      ILI9341_DC_PORT                 GPIOD
#define      ILI9341_DC_PIN                  GPIO_PIN_11

//写使能
#define      ILI9341_WR_PORT                 GPIOD
#define      ILI9341_WR_PIN                  GPIO_PIN_5

//读使能
#define      ILI9341_RD_PORT                 GPIOD
#define      ILI9341_RD_PIN                  GPIO_PIN_4

//复位引脚
#define      ILI9341_RST_PORT                GPIOE
#define      ILI9341_RST_PIN                 GPIO_PIN_1

//背光引脚
#define      ILI9341_BK_PORT                 GPIOD
#define      ILI9341_BK_PIN                  GPIO_PIN_12

/******** 数据信号线 ***************/
#define      ILI9341_D0_PORT                 GPIOD
#define      ILI9341_D0_PIN                  GPIO_PIN_14
```

(下页继续)

(续上页)

```
/******* 此处省略其它数据线 ********/
```

以上代码根据硬件的连接，把与 FSMC 与液晶屏通讯使用的引脚号、引脚源以及复用功能映射都以宏封装起来。其中请着重关注代码中液晶屏 LCD_CS 及 LCD_RS(DC 引脚) 与 FSMC 存储区选择引脚 FSMC_NE 及地址信号 FSMC_A 的编号，它们会决定 STM32 要使用什么内存地址来控制与液晶屏的通讯。

25.10.2.2.2 初始化 FSMC 的 GPIO

利用上面的宏，编写 FSMC 的 GPIO 引脚初始化函数，见*代码清单 26_4*。

列表 5: 代码清单 26-4 FSMC 的 GPIO 初始化函数 (省略了部分数据线)

```
static void ILI9341_GPIO_Config ( void )
{
   GPIO_InitTypeDef GPIO_Initure;

   /* Enable GPIOs clock */
   __HAL_RCC_GPIOD_CLK_ENABLE();
   __HAL_RCC_GPIOE_CLK_ENABLE();
   __HAL_RCC_GPIOF_CLK_ENABLE();
   __HAL_RCC_GPIOG_CLK_ENABLE();
   __HAL_RCC_FSMC_CLK_ENABLE();       //使能 FSMC 时钟

   /* Common GPIO configuration */
   GPIO_Initure.Mode       = GPIO_MODE_OUTPUT_PP;  //推挽输出
   GPIO_Initure.Pull       = GPIO_PULLUP;
   GPIO_Initure.Speed      = GPIO_SPEED_FREQ_HIGH;
   GPIO_Initure.Pin=GPIO_PIN_12;
```

(下页继续)

(续上页)

```
    HAL_GPIO_Init(GPIOD,&GPIO_Initure);

    //初始化复位引脚 G11
    GPIO_Initure.Pin=GPIO_PIN_1;
    HAL_GPIO_Init(GPIOE,&GPIO_Initure);

    GPIO_Initure.Mode=GPIO_MODE_AF_PP;
//  GPIO_Initure.Alternate=GPIO_AF12_FSMC;  //复用为 FSMC

    //初始化 PD0,1,4,5,8,9,10,14,15
    GPIO_Initure.Pin=GPIO_PIN_0| GPIO_PIN_1| GPIO_PIN_4| GPIO_PIN_5|GPIO_P
                     IN_7| GPIO_PIN_8|\
                     GPIO_PIN_9| GPIO_PIN_10| GPIO_PIN_11| GPIO_PIN_14|GPI
                     O_PIN_15;
    HAL_GPIO_Init(GPIOD, &GPIO_Initure);

    //初始化 PE2,7,8,9,10,11,12,13,14,15
    GPIO_Initure.Pin=  GPIO_PIN_7| GPIO_PIN_8| GPIO_PIN_9| GPIO_PIN_10|GPI
                       O_PIN_11|\
                       GPIO_PIN_12| GPIO_PIN_13| GPIO_PIN_14|GPIO_PIN_15;
    HAL_GPIO_Init(GPIOE,&GPIO_Initure);

}
```

与控制 SRAM 中的 GPIO 初始化类似，对于 FSMC 引脚，全部直接初始化为复用推挽输出模式即可，而背光 BK 引脚及液晶复信 RST 信号则被初始化成普通的推挽输出模式，这两个液晶控制信号直接输出普通的电平控制即可。

25.10.2.2.3 配置 FSMC 的模式

接下来需要配置 FSMC 的工作模式，见*代码清单 26_5*。

列表 6: 代码清单 26-5 配置 FSMC 的模式

```
static void ILI9341_FSMC_Config ( void )
{
    SRAM_Handler.Instance = FSMC_NORSRAM_DEVICE;
    SRAM_Handler.Extended = FSMC_NORSRAM_EXTENDED_DEVICE;
    /* SRAM device configuration */
    Timing.AddressSetupTime      = 0x00;
    Timing.AddressHoldTime       = 0x00;
    Timing.DataSetupTime         = 0x08;
    Timing.BusTurnAroundDuration = 0x00;
    Timing.CLKDivision           = 0x00;
    Timing.DataLatency           = 0x00;
    Timing.AccessMode            = FSMC_ACCESS_MODE_B;
    //使用 NE4
    SRAM_Handler.Init.NSBank=FSMC_Bank1_NORSRAMx;
    //地址/数据线不复用
    SRAM_Handler.Init.DataAddressMux=FSMC_DATA_ADDRESS_MUX_DISABLE;
    //NOR
    SRAM_Handler.Init.MemoryType=FSMC_MEMORY_TYPE_NOR;
    //16 位数据宽度
    SRAM_Handler.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
    SRAM_Handler.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
    SRAM_Handler.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
    SRAM_Handler.Init.WaitSignalActive=FSMC_WAIT_TIMING_BEFORE_WS;
    //存储器写使能
```

(下页继续)

(续上页)

```
    SRAM_Handler.Init.WriteOperation=FSMC_WRITE_OPERATION_ENABLE;
    //等待使能位，此处未用到
    SRAM_Handler.Init.WaitSignal=FSMC_WAIT_SIGNAL_DISABLE;
    //读写使用相同的时序
    SRAM_Handler.Init.ExtendedMode=FSMC_EXTENDED_MODE_DISABLE;
    //是否使能同步传输模式下的等待信号，此处未用到
    SRAM_Handler.Init.AsynchronousWait=FSMC_ASYNCHRONOUS_WAIT_DISABLE;
    //禁止突发写
    SRAM_Handler.Init.WriteBurst=FSMC_WRITE_BURST_DISABLE;

    /* SRAM controller initialization */
    ILI9341_GPIO_Config();
    HAL_SRAM_Init(& SRAM_Handler, &Timing, &Timing);
}
```

这个函数的主体是把 FSMC 配置成异步 NOR FLASH 使用的模式 B，使用该方式模拟 8080 时序控制液晶屏，执行流程如下：

初始化 FSMC 时钟

函数开头使用 __HAL_RCC_FSMC_CLK_ENABLE(); 使能 FSMC 外设的时钟。

时序结构体赋值

接下来对时序结构体 FSMC_NORSRAM_TimingInitTypeDef 赋值。在这个时序结构体配置中，由于我们要使用异步 NOR FLASH 的方式模拟 8080 时序，所以选择 FSMC 为模式 B，在该模式下配置 FSMC 的控制时序结构体中，实际上只有地址建立时间 AddressSetupTime（即 ADDSET 的值）以及数据建立时间 DataSetupTime（即 DATAST 的值）成员的配置值是有效的，其它异步 NOR FLASH 没使用到的成员值全配置为 0 即可。而且，这些成员值使用的单位为：1 个 HCLK 的时钟周期，而 HCLK 的时钟频率为 72MHz，对应每个时钟周期为 1/72 微秒。

由图 *26_23* 及图 *26_24* 中的 ILI9341 时序参数说明及要求可大致得知 ILI9341 的写周期为最小 t_{wc} = 66ns，而读周期最小为 $t_{rdl}+t_{rod}$=45+20=65ns。（对于读周期表中有参数要一个要求为 trcfm 和 trc 分别为 450ns 及 160ns，但经过测试并不需要遵照它们的指标要求）

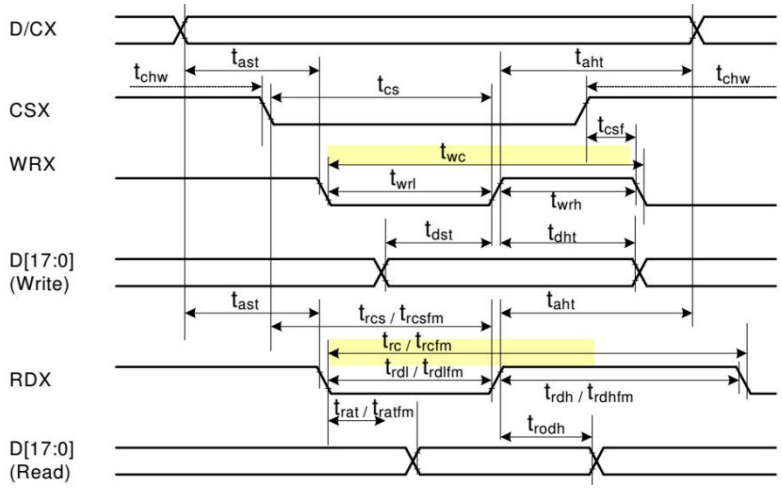


图 26-23 ILI9341 时序参数说明图 (摘自 ILI9341 数据手册)

Signal	Symbol	Parameter	min	max	Unit	Description
DCX	tast	Address setup time	0	-	ns	
	taht	Address hold time (Write/Read)	0	-	ns	
CSX	tchw	CSX "H" pulse width	0	-	ns	
	tcs	Chip Select setup time (Write)	15	-	ns	
	trcs	Chip Select setup time (Read ID)	45	-	ns	
	trcsfm	Chip Select setup time (Read FM)	355	-	ns	
	tcsf	Chip Select Wait time (Write/Read)	10	-	ns	
WRX	twc	Write cycle	66	-	ns	
	twrh	Write Control pulse H duration	15	-	ns	
	twrl	Write Control pulse L duration	15	-	ns	
RDX (FM)	trcfm	Read Cycle (FM)	450	-	ns	
	trdhfm	Read Control H duration (FM)	90	-	ns	
	trdlfm	Read Control L duration (FM)	355	-	ns	
RDX (ID)	trc	Read cycle (ID)	160	-	ns	
	trdh	Read Control pulse H duration	90	-	ns	
	trdl	Read Control pulse L duration	45	-	ns	
D[17:0], D[15:0], D[8:0], D[7:0]	tdst	Write data setup time	10	-	ns	For maximum CL=30pF For minimum CL=8pF
	tdht	Write data hold time	10	-	ns	
	trat	Read access time	-	40	ns	
	tratfm	Read access time	-	340	ns	
	trod	Read output disable time	20	80	ns	

图 26-24 ILI9341 的时序参数要求 (摘自 ILI9341 数据手册)

在 FSMC 代码中使用结构体中的 AddressSetupTime（即 ADDSET 的值）及 DataSetupTime（即 DATAST 的值）成员控制 FSMC 的读写周期，见图 *26_25*。

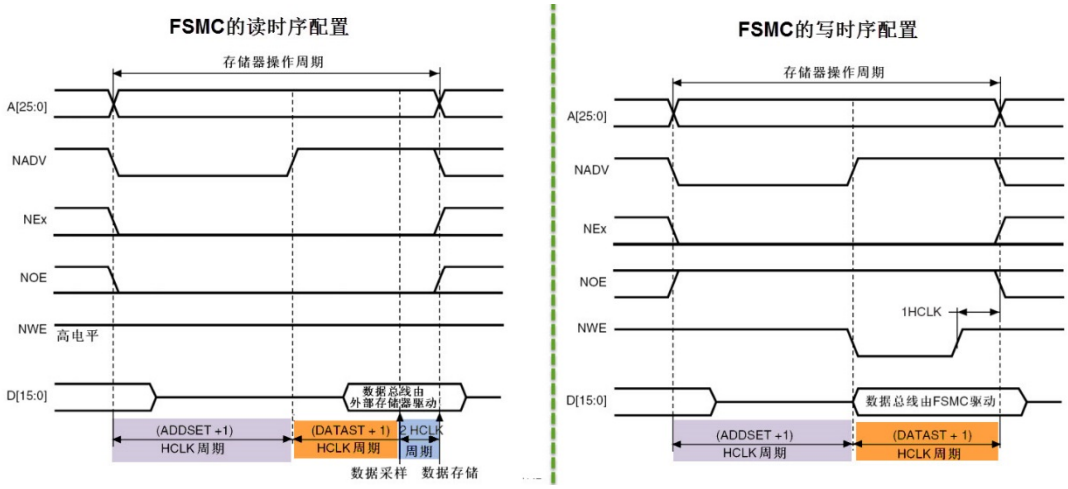


图 26-25FSMC 的读写时序

结合 ILI9341 的时序要求和 FSMC 的配置图，代码中按照读写时序周期均要求至少 66ns 来计算，配置结果为 ADDSET = 1 及 DATST = 4，把时间单位 1/72 微秒 (即 1000/72 纳秒) 代入，因此读写周期的时间被配置为：

读周期：t_{rc} =((ADDSET+1)+(DATST+1)+2) *1000/72 = ((1+1)+(4+1)+2)*1000/72 = 125ns

写周期：t_{wc} =((ADDSET+1)+(DATST+1)) *1000/72 = ((1+1)+(4+1))*1000/72 = 97ns

所以把这两个参数值写入到 FSMC 后，它控制的读写周期都比 ILI9341 的最低要求值大。（经测试，这两个参数值也可以适当减小，您可以亲自试一下）

把计算得的参数赋值到时序结构体中的 AddressSetupTime（即 ADDSET 的值）及 DataSetupTime（即 DATAST 的值）中。

(1) 配置 FSMC 初始化结构体

函数接下来对 FSMC SRAM 的初始化结构体赋值。主要包括存储映射区域、存储器类型以及数据线宽度等，这些是根据外部电路设置的。

设置存储区域 NSBank

- 设置存储区域 NSBank

  FSMC_Bank 成员设置 FSMC 的 NOR FLASH 存储区域映射选择为宏 FSMC_Bank1_NORSRAMx (即 FSMC_NORSRAM_BANK4)，；

- 存储器类型 MemoryType

  由于使用异步 NOR FLASH 模式模拟 8080 时序，所以 MemoryType 成员要选择相应的 FSMC_MEMORY_TYPE_NOR；

- 数据线宽度 MemoryDataWidth

  根据硬件的数据线连接，数据线宽度被配置为 16 位宽 FSMC_NORSRAM_MEM_BUS_WIDTH_16；

- 写使能 WriteOperation

  WriteOperation 用于设置写使能，只有使能了才能正常使用 FSMC 向外部存储器写入数据；

- 扩展模式以及读写时序

  在 _ExtendedMode 成员中可以配置是否使用扩展模式，当设置扩展模式时，读时序使用 FSMC_ReadWriteTimingStruct 中的配置，写时序使用 FSMC_WriteTimingStruct 中的配置，两种配置互相独立，可以赋值为不同的读写时序结构体。在本实例中不使用扩展模式，即读写时序使用相同的配置，都是赋值为前面的 readWriteTiming 结构体；

- 其它

  配置 FSMC 还涉及到其它的结构体成员，但这些结构体成员与异步 NOR FLASH 控制不相关，都被设置为 Disable 了；

赋值完成后调用库函数 HAL_SRAM_Init 把初始化结构体配置的各种参数写入到 FSMC_BCR 控制寄存器及 FSMC_BTR 时序寄存器中。

25.10.2.2.4 计算控制液晶屏时使用的地址

初始化完 FSMC 后，即可使用类似扩展外部 SRAM 中的读取方式：通过访问某个地址，由 FSMC 产生时序与外部存储器通讯，进行读写。

同样地，当访问特定地址时，FSMC 会产生相应的模拟 8080 时序，控制地址线输出要访问的内存地址，使用数据信号线接收或发送数据，其它片选信号 NE、读使能信号 NOE、写使能信号 NWE 辅助产生完整的时序，而由于控制液晶屏的硬件连接中，使用如图 *26_26* 中的连接来模拟

8080 时序，所以 FSMC 产生的这些信号会被 ILI9341 接收，并且使用其中一根 FSMC_Ax 地址控制命令/数据选择引脚 RS(即 D/CX)，因此，需要重点理解下当 STM32 访问什么地址时，对应的 FSMC_Ax 引脚会输出高电平表示传输的是数据，访问什么地址时，对应的 FSMC_Ax 引脚会输出低电平表示传输的是命令。若理解了计算过程，以后就可以根据自己制作的硬件电路来计算访问地址了。

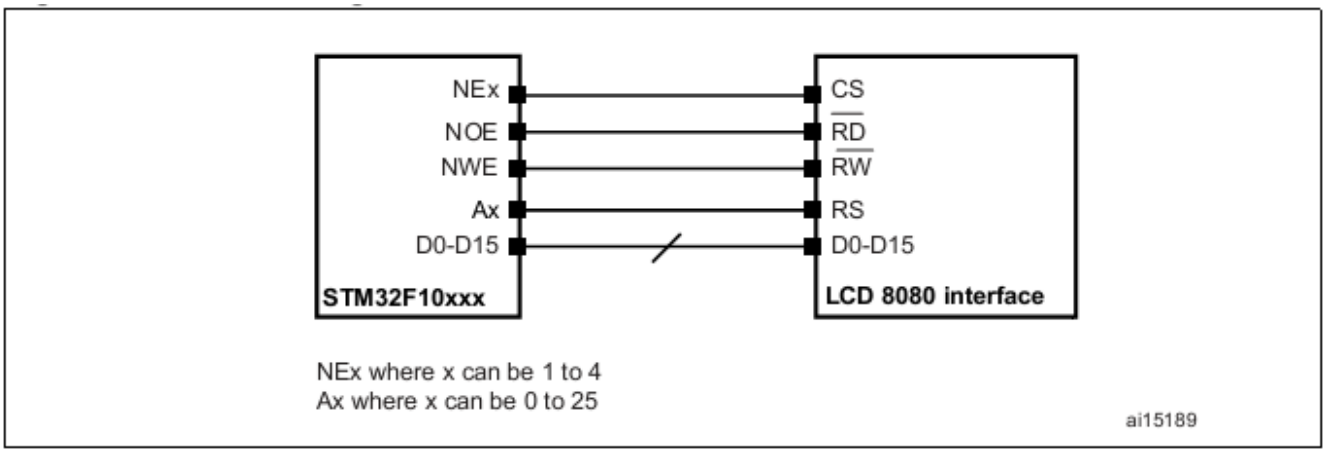


图 26-26 FSMC 与 8080 端口连接简图

计算地址的过程如下：

(1) 本工程中使用的是 FSMC_NE4 作为 8080_CS 片选信号，所以首先可以确认地址范围，当访问 0X6C00 0000 ~ 0X6FFF FFFF 地址时，FSMC 均会对外产生片选有效的访问时序；

(2) 本工程中使用 FSMC_A23 地址线作为命令/数据选择线 RS 信号，所以在以上地址范围内，再选择出使得 FSMC_A23 输出高电平的地址，即可控制表示数据，选择出使得 FSMC_A23 输出低电平的地址，即可控制表示命令。

- 要使 FSMC_A23 地址线为高电平，实质是输出地址信号的第 23 位为 1 即可，使用 0X6C00 0000~0X6FFF FFFF 内的任意地址，作如下运算：

  设置地址的第 23 位为 1：0X6C00 0000 |= (1«23) = 0x6C80 0000

- 要使 FSMC_A23 地址线为低电平，实质是输出地址信号的第 23 位为 0 即可，使用 0X6C00 0000~0X6FFF FFFF 内的任意地址，作如下运算：

  设置地址的第 23 位为 0：0X6C00 0000 &= ~ (1«23) = 0x6C00 0000

(3) 但是，以上方法计算的地址还不完全正确，根据《STM32 参考手册》对 FSMC 访问 NOR

FLASH 的说明，见图 *26_27*，STM32 内部访问地址时使用的是内部 HADDR 总线，它是需要转换到外部存储器的内部 AHB 地址线，它是字节地址 (8 位)，而存储器访问不都是按字节访问，因此接到存储器的地址线依存储器的数据宽度有所不同。

数据宽度(1)	连到存储器的地址线	最大访问存储器空间(位)
8位	HADDR[25:0]与FSMC_A[25:0]对应相连	64M字节 x 8 = 512 M位
16位	HADDR[25:1]与FSMC_A[24:0]对应相连，HADDR[0]未接	64M字节/2 x 16 = 512 M位

(1) 对于16位宽度的外部存储器，FSMC将在内部使用HADDR[25:1]产生外部存储器的地址FSMC_A[24:0]。不论外部存储器的宽度是多少(16位或8位)，FSMC_A[0]始终应该连到外部存储器的地址线A[0]。

图 26-27《STM32 参考手册》中对 HADDR 与 FSMC 地址线的说明

在本工程中使用的是 16 位的数据访问方式，所以 HADDR 与 FSMC_A 的地址线连接关系会左移一位，如 HADDR1 与 FSMC_A0 对应、HADDR2 与 FSMC_A1 对应。因此，当 FSMC_A0 地址线为 1 时，实际上内部地址的第 1 位为 1，FSMC_A1 地址线为 1 时，实际上内部地址的第 2 位为 1。同样地，当希望 FSMC_A23 地址输出高电平或低电平时，需要重新调整计算公式：

- 要使 FSMC_A23 地址线为高电平，实质是访问内部 HADDR 地址的第 (23+1) 位为 1 即可，使用 0X6C00 0000~0X6FFF FFFF 内的任意地址，作如下运算：

  使 FSMC_A23 地址线为高电平：0X6C00 0000 |= (1«(23+1)) = 0x6D00 0000

- 要使 FSMC_A23 地址线为低电平，实质是访问内部 HADDR 地址的第 (23+1) 位为 0 即可，使用 0X6C00 0000~0X6FFF FFFF 内的任意地址，作如下运算：

  使 FSMC_A23 地址线为低电平：0X6C00 0000 &= ~ (1«(23+1)) = 0x6C00 0000

根据最终的计算结果，总结如下：当 STM32 访问内部的 0x6D00 0000 地址时，FSMC 自动输出时序，且使得与液晶屏的数据/命令选择线 RS(即 D/CX) 相连的 FSMC_A23 输出高电平，使得液晶屏会把传输过程理解为数据传输；类似地，当 STM32 访问内部的 0X6C00 0000 地址时，FSMC 自动输出时序，且使得与液晶屏的数据/命令选择线 RS(即 D/CX) 相连的 FSMC_A23 输出低电平，使得液晶屏会把传输过程理解为命令传输。

在工程代码中，把以上计算结果封装成了宏，见*代码清单 26_6*

列表 7: 代码清单 26_6 使用 FSMC 访问数据及访问命令的地址（bsp_ili9341_lcd.h 文件）

```
/*************************************************************
2^26 =0X0400 0000 = 64MB, 每个 BANK 有 4*64MB = 256MB
64MB:FSMC_Bank1_NORSRAM1:0X6000 0000 ~ 0X63FF FFFF
64MB:FSMC_Bank1_NORSRAM2:0X6400 0000 ~ 0X67FF FFFF
64MB:FSMC_Bank1_NORSRAM3:0X6800 0000 ~ 0X6BFF FFFF
64MB:FSMC_Bank1_NORSRAM4:0X6C00 0000 ~ 0X6FFF FFFF

选择 BANK1-BORSRAM4 连接 TFT, 地址范围为 0X6C00 0000 ~ 0X6FFF FFFF
FSMC_A23 接 LCD 的 DC(寄存器/数据选择) 脚
寄存器基地址 = 0X6C00 0000
RAM 基地址 = 0X6D00 0000 = 0X6C00 0000+2^23*2 = 0X6C00 0000 + 0X100 0000 =␣
↪0X6D00 0000
当选择不同的地址线时, 地址要重新计算
*************************************************************/

/*************** ILI9341 显示屏的 FSMC 参数定义 ***************/
//FSMC_Bank1_NORSRAM 用于 LCD 命令操作的地址
#define      FSMC_Addr_ILI9341_CMD         ( ( uint32_t ) 0x6C000000 )

//FSMC_Bank1_NORSRAM 用于 LCD 数据操作的地址
#define      FSMC_Addr_ILI9341_DATA        ( ( uint32_t ) 0x6D000000 )
```

利用这样的宏，再使用指针的形式访问其地址，即可控制 FSMC 产生相应的时序，工程代码中把发送命令及发送数据的操作封装成了内联函数，方便后面调用，见代码清单 26_7。

列表 8: 代码清单 26-7 向液晶屏发送命令及发送数据的操作

```
/**
```

(下页继续)

(续上页)

```
* @brief  向 ILI9341 写入命令
* @param  usCmd : 要写入的命令(表寄存器地址)
* @retval 无
*/
__inline void ILI9341_Write_Cmd ( uint16_t usCmd )
{
    * ( __IO uint16_t * ) ( FSMC_Addr_ILI9341_CMD ) = usCmd;

}

/**
* @brief  向 ILI9341 写入数据
* @param  usData : 要写入的数据
* @retval 无
*/
__inline void ILI9341_Write_Data ( uint16_t usData )
{
    * ( __IO uint16_t * ) ( FSMC_Addr_ILI9341_DATA ) = usData;

}
```

需要写操作时，只要把要发送的命令代码或数据作为参数输入到函数然后调用即可，对于液晶屏的读操作，把向指针赋值的过程改为读取指针内容即可。

25.10.2.2.5 向液晶屏写入初始化配置

利用上面的发送命令及数据操作，可以向液晶屏写入一些初始化配置，见代码清单 *26_8*。

列表 9: 代码清单 26-8 向液晶屏写入初始化配置

```
/**
* @brief  初始化 ILI9341 寄存器
* @param  无
* @retval 无
*/
static void ILI9341_REG_Config ( void )
{
    /*  Power control B (CFh)  */
    DEBUG_DELAY  ();
    ILI9341_Write_Cmd ( 0xCF  );
    ILI9341_Write_Data ( 0x00  );
    ILI9341_Write_Data ( 0x81  );
    ILI9341_Write_Data ( 0x30  );

    /*  Power on sequence control (EDh) */
    DEBUG_DELAY ();
    ILI9341_Write_Cmd ( 0xED );
    ILI9341_Write_Data ( 0x64 );
    ILI9341_Write_Data ( 0x03 );
    ILI9341_Write_Data ( 0x12 );
    ILI9341_Write_Data ( 0x81 );

    /* 以下省略大量配置内容 */
}
```

以上列出的代码中省略了大量的配置内容，本质上它们都是使用 ILI9341_Write_Cmd 发送代码，然后使用 ILI9341_Write_Data 函数发送命令对应的参数对液晶屏进行配置。

这个初始化过程中发送的代码及参数主要是配置了液晶屏的上电过程、显示屏的伽玛参数、分辨率、像素格式等内容，这些配置主要由液晶屏生产厂家提供，本教程后面只针对常用命令进行讲解，此处不作详细说明，关于命令及参数可以查询《ILI9341 数据手册》获知，在该文档中搜

索命令代码即可方便定位到相应的说明。例如，要查找代码中的 0xCF 命令说明，在文档中搜索“CFh”即可，见图 26_28_。

8.4.2 Power control B (CFh)

CFh	Power control B												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	1	1	0	0	1	1	1	1	CFh
1st Parameter	1	1	↑	XX	0	0	0	0	0	0	0	0	00
2nd Parameter	1	1	↑	XX	1	0	0	Power control[1:0]		0	0	1	81
3rd Parameter	1	1	↑	XX	0	0	1	DC_ena	0	0	0	0	30

Description

2nd parameter: power control[1:0]

Only setting power control [1:0]=11, the VGH and VGL voltage level follow the table below.

BT [2:0]			DDVDH	VGH	VGL
0	0	0	VCI x 2	VCI x 7	-VCI x 4
0	0	1			-VCI x 3
0	1	0		VCI x 6	-VCI x 4
0	1	1			-VCI x 3

3rd parameter: Discharge path enable. Enable high for ESD protection

图 26-28 在《ILI9341 数据手册》中 CFh 命令的部分说明

提示： 可能有部分批次的 ILI9341 液晶屏在使用过程中会出现莫名其妙的条纹，遇到这种情况可以试着去改下原本的初始化配置，就在 ILI9341_REG_Config 这个函数里面、把这个参数的值调大（最大为 0x06，参考 ILI9341 的数据手册里的“8.4.1 Power control A (CBh)”章节）：

```
220     /*  Power control A (CBh) */
221     macDEBUG_DELAY ();
222     ILI9341_Write_Cmd ( 0xCB );
223     ILI9341_Write_Data ( 0x39 );
224     ILI9341_Write_Data ( 0x2C );
225     ILI9341_Write_Data ( 0x00 );
226     ILI9341_Write_Data ( 0x34 );
227     ILI9341_Write_Data ( 0x03 );
```

设置液晶显示窗口

根据液晶屏的要求，在发送显示数据前，需要先设置显示窗口确定后面发送的像素数据的显示区域，见代码清单 26_9

列表 10: 代码清单 26-9 设置液晶显示窗口

```
/********** ILI934 命令 ********************************/
#define      CMD_SetCoordinateX            0x2A       //设置 X 坐标
#define      CMD_SetCoordinateY            0x2B       //设置 Y 坐标

/**
* @brief  在 ILI9341 显示器上开辟一个窗口
* @param  usX ：在特定扫描方向下窗口的起点 X 坐标
* @param  usY ：在特定扫描方向下窗口的起点 Y 坐标
* @param  usWidth ：窗口的宽度
* @param  usHeight ：窗口的高度
* @retval 无
*/
void ILI9341_OpenWindow ( uint16_t usX, uint16_t usY,
uint16_t usWidth, uint16_t usHeight )
{
   ILI9341_Write_Cmd ( CMD_SetCoordinateX );           /* 设置 X 坐标 */
   ILI9341_Write_Data ( usX >> 8  );  /* 先高 8 位，然后低 8 位 */
   ILI9341_Write_Data ( usX & 0xff  );  /* 设置起始点和结束点 */
   ILI9341_Write_Data ( ( usX + usWidth - 1 ) >> 8  );
   ILI9341_Write_Data ( ( usX + usWidth - 1 ) & 0xff  );

   ILI9341_Write_Cmd ( CMD_SetCoordinateY );            /* 设置 Y 坐标 */
   ILI9341_Write_Data ( usY >> 8  );
   ILI9341_Write_Data ( usY & 0xff  );
   ILI9341_Write_Data ( ( usY + usHeight - 1 ) >> 8 );
   ILI9341_Write_Data ( ( usY + usHeight - 1) & 0xff );
}
```

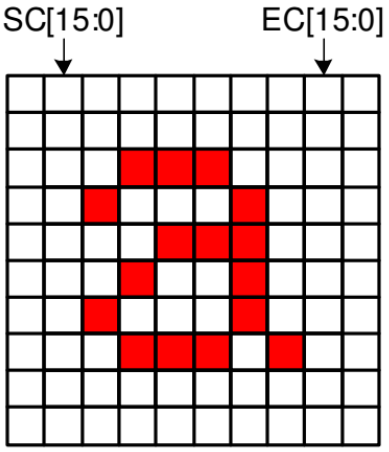
2Ah	CASET (Column Address Set)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	0	0	1	0	1	0	1	0	2Ah
1st Parameter	1	1	↑	XX	SC15	SC14	SC13	SC12	SC11	SC10	SC9	SC8	Note1
2ndParameter	1	1	↑	XX	SC7	SC6	SC5	SC4	SC3	SC2	SC1	SC0	
3rd Parameter	1	1	↑	XX	EC15	EC14	EC13	EC12	EC11	EC10	EC9	EC8	Note1
4th Parameter	1	1	↑	XX	EC7	EC6	EC5	EC4	EC3	EC2	EC1	EC0	
Description	This command is used to define area of frame memory where MCU can access. This command makes no change on the other driver status. The values of SC [15:0] and EC [15:0] are referred when RAMWR command comes. Each value represents one column line in the Frame Memory.  X = Don't care												

图 26-29 设置显示窗口的 X 坐标 (2Ah 命令)（摘自《ILI9341 数据手册》）

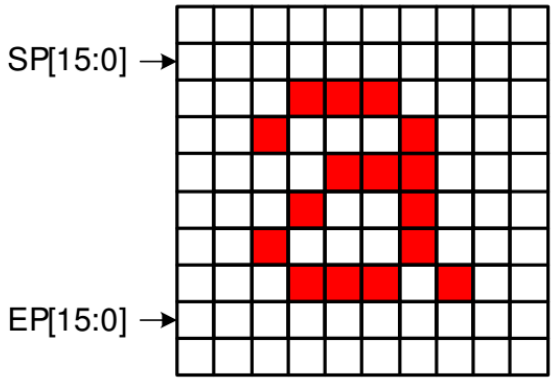
2Bh	PASET (Page Address Set)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	0	0	1	0	1	0	1	1	2Bh
1st Parameter	1	1	↑	XX	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	Note1
2ndParameter	1	1	↑	XX	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	
3rdParameter	1	1	↑	XX	EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	Note1
4th Parameter	1	1	↑	XX	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0	
Description	This command is used to define area of frame memory where MCU can access. This command makes no change on the other driver status. The values of SP [15:0] and EP [15:0] are referred when RAMWR command comes. Each value represents one Page line in the Frame Memory.  X = Don't care												

图 26-30 设置液晶显示窗口的 Y 坐标（2Bh 命令）（摘自《ILI9341 数据手册》）

代码中定义的 ILI9341_OpenWindow 函数实现了图 *26_29* 及图 *26_30* 中的 0x2A 和 0x2B 命令，它们分别用于设置显示窗口的起始及结束的 X 坐标和 Y 坐标，每个命令后包含 4 个 8 位的参数，这些参数组合后成起始坐标和结束坐标各 1 个用 16 位表示的值。ILI9341_OpenWindow 把它的四个函数输入参数 X、Y 起始坐标，宽度、高度转化成命令参数的格式，写入到液晶屏中，从而设置出一个显示窗口。

25.10.2.2.6 发送像素数据

调用上面的 ILI9341_OpenWindow 函数设置显示窗口后，再向液晶屏发送像素数据时，这些数据就会直接显示在它设定的窗口位置中。发送像素数据的操作见*代码清单 26_10*。

列表 11: 代码清单 26_10 发送像素数据

```
#define      CMD_SetPixel                    0x2C        //填充像素

/**
* @brief  在 ILI9341 显示器上以某一颜色填充像素点
* @param  ulAmout_Point ：要填充颜色的像素点的总数目
* @param  usColor ：颜色
* @retval 无
*/
static __inline void ILI9341_FillColor ( uint32_t ulAmout_Point,
uint16_t usColor )
{
   uint32_t i = 0;
   /* memory write */
   ILI9341_Write_Cmd ( CMD_SetPixel );

   for ( i = 0; i < ulAmout_Point; i ++ )
      ILI9341_Write_Data ( usColor );
}
```

2Ch	RAMWR (Memory Write)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	0	0	1	0	1	1	0	0	2Ch
1st Parameter	1	1	↑	D1 [17:0]									XX
:	1	1	↑	Dx [17:0]									XX
Nth Parameter	1	1	↑	Dn [17:0]									XX
Description	This command is used to transfer data from MCU to frame memory. This command makes no change to the other driver status. When this command is accepted, the column register and the page register are reset to the Start Column/Start Page positions. The Start Column/Start Page positions are different in accordance with MADCTL setting.) Then D [17:0] is stored in frame memory and the column register and the page register incremented. Sending any other command can stop frame Write. X = Don't care.												

图 26-31 发送像素数据（2Ch 命令）（摘自《ILI9341 数据手册》）

发送像素数据的命令非常简单，首先发送命令代码 0x2C，然后后面紧跟着要传输的像素数据即可。按照本液晶屏的配置，像素点的格式为 RGB565，所以像素数据就是要显示的 RGB565 格式的颜色值。

本 ILI9341_FillColor 函数包含两个输入参数，分别用于设置要发送的像素数据个数 ulAmout_Point 及像素点的颜色值 usColor，在代码实现中它调用 ILI9341_Write_Cmd 发送一次命令代码，接着使用 for 循环调用 ILI9341_Write_Data 写入 ulAmout_Point 个同样的颜色值。

这些颜色值会按顺序填充到前面使用 ILI9341_OpenWindow 函数设置的显示窗口中，例如，若设置了一个 usX=10，usY=30，usWidth=50，usHeight=20 的窗口，然后再连续填充 50*20 个颜色值为 0XFFFF 的像素数据，即可在 (10,30) 的起始坐标处显示一个宽 50 像素高 20 像素的白色矩形。

25.10.2.2.7 绘制单个像素点

利用前面的 ILI9341_OpenWindow 和 ILI9341_FillColor 函数，可以正式开始控制液晶屏绘制特定的图像，而所有图像都是由多个像素点组成的，单个像素点的绘制函数见*代码清单 26_11*。

列表 12: 代码清单 26-11 绘制单个像素点

```
static uint16_t CurrentTextColor   = BLACK;//前景色
static uint16_t CurrentBackColor   = WHITE;//背景色

/**
* @brief  设定 ILI9341 的光标坐标
* @param  usX : 在特定扫描方向下光标的 X 坐标
* @param  usY : 在特定扫描方向下光标的 Y 坐标
* @retval 无
*/
static void ILI9341_SetCursor ( uint16_t usX, uint16_t usY )
{
   ILI9341_OpenWindow ( usX, usY, 1, 1 );
}
```

(下页继续)

(续上页)

```
/**
* @brief  对 ILI9341 显示器的某一点以某种颜色进行填充
* @param  usX ：在特定扫描方向下该点的 X 坐标
* @param  usY ：在特定扫描方向下该点的 Y 坐标
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
* @retval 无
*/
void ILI9341_SetPointPixel ( uint16_t usX, uint16_t usY )
{
   if ( ( usX < LCD_X_LENGTH ) && ( usY < LCD_Y_LENGTH ) ) {
      ILI9341_SetCursor ( usX, usY );

      ILI9341_FillColor ( 1, CurrentTextColor );
   }
}
```

ILI9341_SetPointPixel 函数直接调用了 ILI9341_SetCursor（实质上是 ILI9341_OpenWindow 函数的封装）设置单个像素点的绘制窗口，然后调用 ILI9341_FillColor 填充单个像素点，而像素点的颜色由全局变量 CurrentTextColor 表示。

利用这个 ILI9341_SetPointPixel 函数，可以向液晶屏指定的 XY 坐标描绘单个像素点。

绘制矩形

类似地，使用 ILI9341_OpenWindow 和 ILI9341_FillColor 制作的绘制矩形操作见*代码清单 26_12*。

列表 13: 代码清单 26-12 绘制矩形

```
/**
* @brief  在 ILI9341 显示器上画一个矩形
* @param  usX_Start ：在特定扫描方向下矩形的起始点 X 坐标
* @param  usY_Start ：在特定扫描方向下矩形的起始点 Y 坐标
* @param  usWidth：矩形的宽度（单位：像素）
* @param  usHeight：矩形的高度（单位：像素）
```

(下页继续)

(续上页)

```
* @param  ucFilled ：选择是否填充该矩形
*   该参数为以下值之一：
*     @arg 0 ：空心矩形
*     @arg 1 ：实心矩形
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
* @retval 无
*/
void ILI9341_DrawRectangle ( uint16_t usX_Start, uint16_t usY_Start,
                uint16_t usWidth, uint16_t usHeight, uint8_t ucFilled )
{
   if ( ucFilled ) {
      ILI9341_OpenWindow ( usX_Start, usY_Start, usWidth, usHeight );
      ILI9341_FillColor ( usWidth * usHeight ,CurrentTextColor);
   } else {
      ILI9341_DrawLine ( usX_Start, usY_Start,
                           usX_Start + usWidth - 1, usY_Start );
      ILI9341_DrawLine ( usX_Start, usY_Start + usHeight - 1,
                     usX_Start + usWidth - 1, usY_Start + usHeight - 1 );
      ILI9341_DrawLine ( usX_Start, usY_Start, usX_Start,
                           usY_Start + usHeight - 1 );
      ILI9341_DrawLine ( usX_Start + usWidth - 1, usY_Start,
                     usX_Start + usWidth - 1, usY_Start + usHeight - 1 );
   }
}
```

ILI9341_DrawRectangle 函数分成两部分，它根据输入参数 ucFilled 是否为真值决定绘制的是实心矩形还是只有边框的矩形。绘制实心矩形时，直接使用 ILI9341_OpenWindow 函数根据输入参数设置显示矩形窗口，然后根据实心矩形的像素点个数调用 ILI9341_FillColor 即可完成绘制；而绘制空心矩形时，实质上是绘制四条边框线，它调用 ILI9341_DrawLine 函数绘制，ILI9341_DrawLine 函数的输入参数是用于表示直接的两个坐标点 (x1,y1) 与 (x2,y2)，该函数内部根据数据关系，使用这两个点确定一条直线，最后调用 ILI9341_SetPointPixel 函数一点一点地绘制成完整的直线。

关于 ILI9341_DrawLine 画线函数、ILI9341_DrawCircle 画圆函数等代码不再讲解，它们都是根据

数学关系在特点的位置显示坐标点而已。另外关于工程中的显示字符串的原理将在下一个章节详细说明。

25.10.2.2.8 设置液晶的扫描方向

控制液晶屏时，还有一个非常重要的参数，就是设置液晶屏的扫描方向，见*代码清单 26_13*。

列表 14: 代码清单 26-13 设置液晶的扫描方向

```
#define        ILI9341_LESS_PIXEL          240      //液晶屏较短方向的像素宽度
#define        ILI9341_MORE_PIXEL         320      //液晶屏较长方向的像素宽度

//根据液晶扫描方向而变化的 XY 像素宽度
//调用 ILI9341_GramScan 函数设置方向时会自动更改
uint16_t LCD_X_LENGTH = ILI9341_LESS_PIXEL;
uint16_t LCD_Y_LENGTH = ILI9341_MORE_PIXEL;

//液晶屏扫描模式，本变量主要用于方便选择触摸屏的计算参数
//参数可选值为 0-7
//调用 ILI9341_GramScan 函数设置方向时会自动更改
//LCD 刚初始化完成时会使用本默认值
uint8_t LCD_SCAN_MODE = 6;

/**
* @brief  设置 ILI9341 的 GRAM 的扫描方向
* @param  ucOption ：选择 GRAM 的扫描方向
*     @arg 0-7 : 参数可选值为 0-7 这八个方向
*
*  !!! 其中 0、3、5、6 模式适合从左至右显示文字，
*         不推荐使用其它模式显示文字 其它模式显示文字会有镜像效果
*
*  其中 0、2、4、6 模式的 X 方向像素为 240，Y 方向像素为 320
*  其中 1、3、5、7 模式下 X 方向像素为 320，Y 方向像素为 240
```

(下页继续)

(续上页)

```
*
*  其中 6 模式为大部分液晶例程的默认显示方向
*  其中 3 模式为摄像头例程使用的方向
*  其中 0 模式为 BMP 图片显示例程使用的方向
*
* @retval 无
* @note   坐标图例：A 表示向上，V 表示向下，< 表示向左，> 表示向右
          X 表示 X 轴，Y 表示 Y 轴


------------------------------------------------------------
模式 0:          .   模式 1:      . 模式 2:        . 模式 3:
          A      .          A     .   A           .    A
          |      .          |     .   |           .    |
          Y      .          X     .   Y           .    X
          0      .          1     .   2           .    3
<--- X0 o     . <----Y1 o     .    o 2X--->   .    o 3Y--->
------------------------------------------------------------
模式 4:          . 模式 5:        . 模式 6:        . 模式 7:
<--- X4 o     . <--- Y5 o     .    o 6X--->   .    o 7Y--->
          4      .          5     .   6           .    7
          Y      .          X     .   Y           .    X
          |      .          |     .   |           .    |
          V      .          V     .   V           .    V
------------------------------------------------------------
                      LCD 屏示例
                 |-----------------|
                 |     野火 Logo    |
                 |                 |
                 |                 |
                 |                 |
                 |                 |
                 |                 |
```

(下页继续)

(续上页)

```
                |                  |
                |                  |
                |                  |
                |------------------|
                屏幕正面（宽 240，高 320）

*******************************************************/
void ILI9341_GramScan ( uint8_t ucOption )
{
   //参数检查，只可输入 0-7
   if (ucOption >7 )
      return;

   //根据模式更新 LCD_SCAN_MODE 的值，主要用于触摸屏选择计算参数
   LCD_SCAN_MODE = ucOption;

   //根据模式更新 XY 方向的像素宽度
   if (ucOption%2 == 0) {
      //0 2 4 6 模式下 X 方向像素宽度为 240，Y 方向为 320
      LCD_X_LENGTH = ILI9341_LESS_PIXEL;
      LCD_Y_LENGTH =  ILI9341_MORE_PIXEL;
   } else {
      //1 3 5 7 模式下 X 方向像素宽度为 320，Y 方向为 240
      LCD_X_LENGTH = ILI9341_MORE_PIXEL;
      LCD_Y_LENGTH =  ILI9341_LESS_PIXEL;
   }

   //0x36 命令参数的高 3 位可用于设置 GRAM 扫描方向
   ILI9341_Write_Cmd ( 0x36 );
ILI9341_Write_Data ( 0x08 |(ucOption<<5));//根据 ucOption 的值设置 LCD 参数，共
0-7 种模式
   ILI9341_Write_Cmd ( CMD_SetCoordinateX );
```

(下页继续)

(续上页)

```
    ILI9341_Write_Data ( 0x00 );     /* x 起始坐标高 8 位 */
    ILI9341_Write_Data ( 0x00 );     /* x 起始坐标低 8 位 */
    ILI9341_Write_Data ( ((LCD_X_LENGTH-1)>>8)&0xFF );/* x 结束坐标高 8 位 */
    ILI9341_Write_Data ( (LCD_X_LENGTH-1)&0xFF );     /* x 结束坐标低 8 位 */

    ILI9341_Write_Cmd ( CMD_SetCoordinateY );
    ILI9341_Write_Data ( 0x00 );     /* y 起始坐标高 8 位 */
    ILI9341_Write_Data ( 0x00 );     /* y 起始坐标低 8 位 */
    ILI9341_Write_Data ( ((LCD_Y_LENGTH-1)>>8)&0xFF );/*y 结束坐标高 8 位 */
    ILI9341_Write_Data ( (LCD_Y_LENGTH-1)&0xFF );     /*y 结束坐标低 8 位 */

    /* write gram start */
    ILI9341_Write_Cmd ( CMD_SetPixel );
}
```

当设置了液晶显示窗口，再连续向液晶屏写入像素点时，它会一个点一个点地往液晶屏的 X 方向填充，填充完一行 X 方向的像素点后，向 Y 方向下移一行，X 坐标回到起始位置，再往 X 方向一个点一个点地填充，如此循环直至填充完整个显示窗口。

而屏幕的坐标原点和 XY 方向都可以根据实际需要使用 0X36 命令来配置的，该命令的说明见图 *26_32*。

36h	MADCTL (Memory Access Control)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	0	0	1	1	0	1	1	0	36h
Parameter	1	1	↑	XX	MY	MX	MV	ML	BGR	MH	0	0	00

This command defines read/write scanning direction of frame memory.

This command makes no change on the other driver status.

Bit	Name	Description
MY	Row Address Order	These 3 bits control MCU to memory write/read direction.
MX	Column Address Order	
MV	Row / Column Exchange	
ML	Vertical Refresh Order	LCD vertical refresh direction control.
BGR	RGB-BGR Order	Color selector switch control (0=RGB color filter panel, 1=BGR color filter panel)
MH	Horizontal Refresh ORDER	LCD horizontal refreshing direction control.

图 26-32 液晶扫描模式命令（摘自《ILI9341 数据手册》）

0X36 命令参数中的 MY、MX、MV 这三个数据位用于配置扫描方向，因此一共有 $2^3 = 8$ 种模式。

ILI9341_GramScan 函数就是根据输入的模式设置这三个数据位，并且根据相应的模式更改 XY 方向的分辨率 LCD_X_LENGTH 和 LCD_Y_LENGTH，使得其它函数可以利用这两个全局变量获屏幕实际的 XY 方向分辨率信息；同时，函数内还设置了全局变量 LCD_SCAN_MODE 的值用于记录当前的屏幕扫描模式，这在后面计算触摸屏坐标的时候会使用到。设置完扫描方向后，代码中还调用设置液晶显示窗口的命令 CMD_SetCoordinateX/Y（0X2A/0X2B 命令）默认打开一个与屏幕大小一致的显示窗口，方便后续的显示操作。

调用 ILI9341_GramScan 函数设置 0-7 模式时，各个模式的坐标原点及 XY 方向如图 *26_33* 所示。

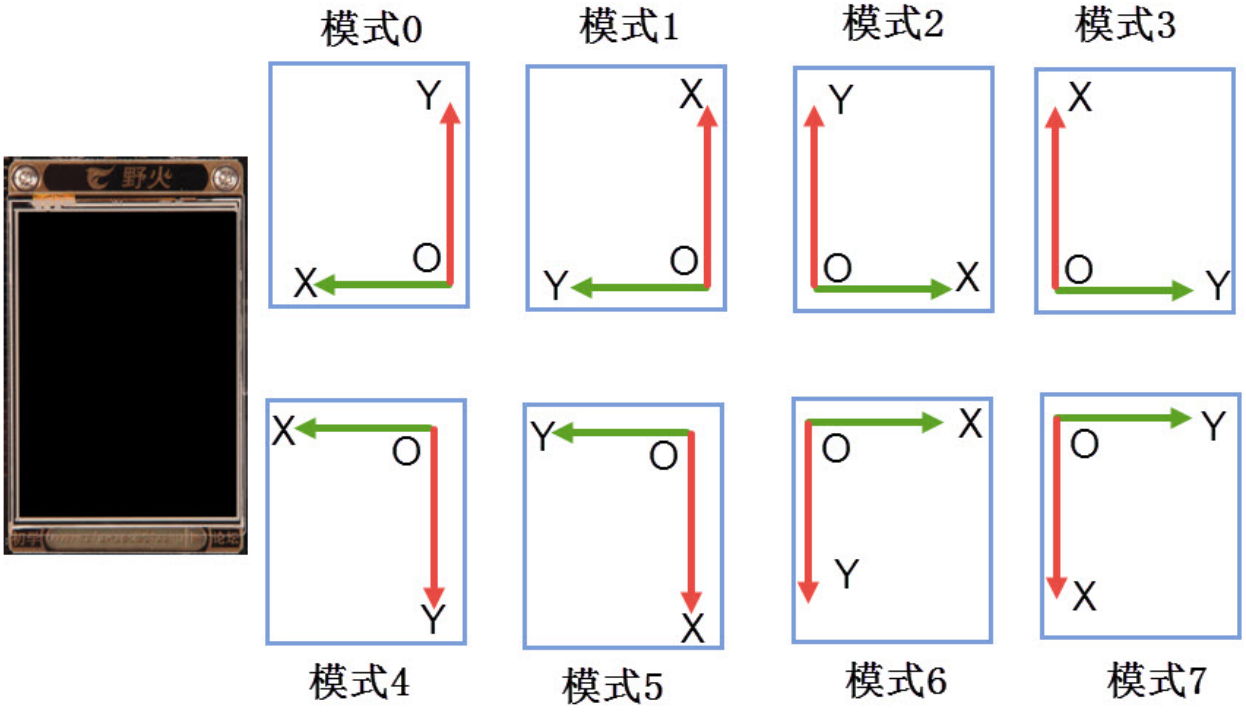


图 26-33 液晶屏的 8 种扫描模式

其中模式 6 最符合我们的阅读习惯，扫描方向与文字方向一致，都是从左到右，从上到下，所以本开发板中的大部分液晶程序都是默认使用模式 6。

其实模式 0、3、5、6 的液晶扫描方向都与文字方向一致，比较适合显示文字，只要适当旋转屏幕即可，使得用屏幕四个边沿作为正面看去都有适合的文字显示模式。而其它模式由于扫描方向与文字方向不一致，要想实现同样的效果非常麻烦，也没有实现的必要。

25.10.2.2.9 液晶屏全局初始化

利用前面介绍的各种函数，我们把它封装成 ILI9341_Init 函数，见*代码清单 26_14*。

列表 15: 代码清单 26-14 液晶屏全局初始化函数

```
//液晶屏扫描模式，本变量主要用于方便选择触摸屏的计算参数
//参数可选值为 0-7
//调用 ILI9341_GramScan 函数设置方向时会自动更改
//LCD 刚初始化完成时会使用本默认值
uint8_t LCD_SCAN_MODE = 6;

/**
* @brief  ILI9341 初始化函数，如果要用到 lcd，一定要调用这个函数
* @param  无
* @retval 无
*/
void ILI9341_Init ( void )
{
   ILI9341_GPIO_Config ();
   ILI9341_FSMC_Config ();

   ILI9341_BackLed_Control ( ENABLE );       //点亮 LCD 背光灯
   ILI9341_Rst ();//复位液晶屏
   ILI9341_REG_Config ();//写入寄存器配置

   //设置默认扫描方向，其中 6 模式为大部分液晶例程的默认显示方向
   ILI9341_GramScan(LCD_SCAN_MODE);
}
```

本函数初始化 GPIO、FSMC 外设，然后开启液晶屏的背光，复位液晶屏，并且写入基本的液晶屏配置，最后调用 ILI9341_GramScan 函数设置默认的液晶扫描方向。在需要使用液晶屏的时候，直接调用本函数即可完成初始化。

25.10.2.3 基本液晶显示例程的 main 函数

本章内容中配套了两个工程进行演示，它们的液晶驱动完全一样，仅是 main 函数里的应用层展示不同效果时稍有区别，先讲解基本的《液晶显示》例程，其 main 函数内容见*代码清单 26_15*。

列表 16: 代码清单 26-15 main 函数

```
int main(void)
{
   /* 系统时钟初始化成 72MHz */
   SystemClock_Config();
   /* LED 端口初始化 */
   LED_GPIO_Config();
   /* 初始化串口 */
   DEBUG_USART_Config();

   ILI9341_Init ();              //LCD 初始化

   printf("\r\n ********** 液晶屏英文显示程序 ********** \r\n");
   printf("\r\n 本程序不支持中文，显示中文的程序请学习下一章 \r\n");

   //其中 0、3、5、6 模式适合从左至右显示文字，
   //不推荐使用其它模式显示文字其它模式显示文字会有镜像效果
   //其中 6 模式为大部分液晶例程的默认显示方向
   ILI9341_GramScan( 6 );
   while ( 1 ) {
         LCD_Test();
   }
}
```

程序中，调用了 ILI9341_Init 函数初始化液晶屏，然后再初始化串口。(在实际测试中，若先初始化串口再初始化液晶屏，会导致错误，原因不明。所以在应用时，注意先初始化液晶屏再初始化串口)

初始化完成后，调用 LCD_Test 函数显示各种图形进行测试 (如直线、矩形、圆形)，见代码清单 26_16。具体内容请直接在工程中阅读源码。LCD_Test 中还调用了文字显示函数，关于文字显示的原理在下一章再详细说明。

列表 17: 代码清单 26-16 液晶效果演示测试

```
/* 用于测试各种液晶的函数 */
void LCD_Test(void)
{
    /* 演示显示变量 */
    static uint8_t testCNT = 0;
    char dispBuff[100];

    testCNT++;

    LCD_SetFont(&Font8x16);
    LCD_SetColors(RED,BLACK);

    ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */
    /******** 显示字符串示例 *******/
    ILI9341_DispStringLine_EN(LINE(0),"BH 3.2 inch LCD para:");
    ILI9341_DispStringLine_EN(LINE(1),"Image resolution:240x320 px");
    ILI9341_DispStringLine_EN(LINE(2),"ILI9341 LCD driver");
    ILI9341_DispStringLine_EN(LINE(3),"XPT2046 Touch Pad driver");

    /******** 显示变量示例 *******/
    LCD_SetFont(&Font16x24);
    LCD_SetTextColor(GREEN);

    /* 使用 cHAL 库把变量转化成字符串 */
    sprintf(dispBuff,"Count : %d ",testCNT);
    LCD_ClearLine(LINE(4)); /* 清除单行文字 */
```

(下页继续)

(续上页)

```
/* 然后显示该字符串即可，其它变量也是这样处理 */
ILI9341_DispStringLine_EN(LINE(4),dispBuff);

/******* 显示图形示例 ******/
LCD_SetFont(&Font24x32);
/* 画直线 */

LCD_ClearLine(LINE(4));/* 清除单行文字 */
LCD_SetTextColor(BLUE);

ILI9341_DispStringLine_EN(LINE(4),"Draw line:");

LCD_SetTextColor(RED);
ILI9341_DrawLine(50,170,210,230);
ILI9341_DrawLine(50,200,210,240);
/* 省略部分内容 */

 ILI9341_Clear(0,16*8,LCD_X_LENGTH,LCD_Y_LENGTH-16*8); /* 清屏，显示全黑 */
/* 画矩形 */

LCD_ClearLine(LINE(4)); /* 清除单行文字 */
LCD_SetTextColor(BLUE);

ILI9341_DispStringLine_EN(LINE(4),"Draw Rect:");

LCD_SetTextColor(RED);
ILI9341_DrawRectangle(50,200,100,30,1);
/* 省略部分内容 */

Delay(0xFFFFFF);

ILI9341_Clear(0,16*8,LCD_X_LENGTH,LCD_Y_LENGTH-16*8); /* 清屏，显示全黑 */
```

(下页继续)

(续上页)

```
    /* 画圆 */
    LCD_ClearLine(LINE(4)); /* 清除单行文字 */
    LCD_SetTextColor(BLUE);

    ILI9341_DispStringLine_EN(LINE(4),"Draw Cir:");

    LCD_SetTextColor(RED);
    ILI9341_DrawCircle(100,200,20,0);
    /* 省略部分内容 */
    Delay(0xFFFFFF);

ILI9341_Clear(0,16*8,LCD_X_LENGTH,LCD_Y_LENGTH-16*8); /* 清屏，显示全黑 */

}
```

25.10.2.4 液晶坐标方向演示的 main 函数

打开《液晶坐标方向演示》的例程，可看到其 main 函数与上一个工程有区别，见*代码清单 26_17*

列表 18: 代码清单 26-17 液晶坐标方向演示例程的 main 函数

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 初始化串口 */
    DEBUG_USART_Config();
```

(下页继续)

(续上页)

```
    ILI9341_Init ();                  //LCD 初始化

    printf("\r\n ********** 液晶屏显示方向说明程序 ********** \r\n");
    printf("\r\n 本程序不支持中文，显示中文的程序请学习下一章 \r\n");

    while ( 1 ) {
          //展示 LCD 的八种方向模式
          LCD_Direction_Show();
    }
}

/* 用于展示 LCD 的八种方向模式 */
void LCD_Direction_Show(void)
{

    uint8_t i = 0;
    char dispBuff[100];

    //轮流展示各个方向模式
    for (i=0; i<8; i++) {
          LCD_SetFont(&Font16x24);
          LCD_SetColors(RED,BLACK);

          ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */

          //其中 0、3、5、6 模式适合从左至右显示文字，
          //不推荐使用其它模式显示文字其它模式显示文字会有镜像效果
          //其中 6 模式为大部分液晶例程的默认显示方向
          ILI9341_GramScan ( i );

          sprintf(dispBuff,"o%d. X --->",i);
```

(下页继续)

(续上页)

```
        ILI9341_DispStringLine_EN(LINE(0),dispBuff);
                                    //沿 X 方向显示文字

        sprintf(dispBuff,"o%d.Y|V",i);
        ILI9341_DispString_EN_YDir(0,0,dispBuff);
                                    //沿 Y 方向显示文字
        Delay(0xFFFFFF);

        //显示测试
        // *  !!! 其中 0、3、5、6 模式适合从左至右显示文字，不推荐使用其它模式显示文字
        //其它模式显示文字会有镜像效果
        LCD_Test();
    }
}
```

本工程的 main 了函数中主要调用了 LCD_Direction_Show 函数，该函数主要添加了液晶屏在不同扫描模式下的显示效果演示，请直接观看程序的演示效果，了解液晶屏的各个扫描模式。注意：其中部分模式显示文字时会因为镜像效果导致无法阅读，这是由扫描模式决定的，并不是代码错误，只要使用适当的模式即可实现正常的文字显示效果。

25.10.3 下载验证

用 USB 线连接开发板，编译程序下载到实验板，并上电复位，液晶屏会显示各种内容。

第 26 章　LCD—液晶显示中英文

本章参考资料：《STM32F10X-中文参考手册》、《STM32F103 增强型系列数据手册》。

在前面我们学习了如何使用 FSMC 外设控制液晶屏并用它显示各种图形，本章讲解如何控制液晶屏显示文字。使用液晶屏显示文字时，涉及到字符编码与字模的知识。

26.1 字符编码

由于计算机只能识别 0 和 1，文字也只能以 0 和 1 的形式在计算机里存储，所以我们需要对文字进行编码才能让计算机处理，编码的过程就是规定特定的 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。

26.1.1 ASCII 编码

学习 C 语言时，我们知道在程序设计中使用 ASCII 编码表约定了一些控制字符、英文及数字。它们在存储器中，本质也是二进制数，只是我们约定这些二进制数可以表示某些特殊意义，如以 ASCII 编码解释数字“0x41”时，它表示英文字符“A”。ASC II 码表分为两部分，第一部分是控制字符或通讯专用字符，它们的数字编码从 0~31，见表 27-1，它们并没有特定的图形显示，但会根据不同的应用程序，而对文本显示有不同的影响。ASCII 码的第二部分包括空格、阿拉伯数字、标点符号、大小写英文字母以及“DEL（删除控制）”，这部分符号的数字编码从 32~127，除最后一个 DEL 符号外，都能以图形的方式来表示，它们属于传统文字书写系统的一部分。

表 27-1 ASCII 码中的控制字符或通讯专用字符

十进制	十六进制	缩写/字符	解释
0	0	NUL(null)	空字符
1	1	SOH(start of headline)	标题开始

下页继续

表 1 – 续上页

2	2	STX (start of text)	正文开始
3	3	ETX (end of text)	正文结束
4	4	EOT (end of transmission)	传输结束
5	5	ENQ (enquiry)	请求
6	6	ACK (acknowledge)	收到通知
7	7	BEL (bell)	响铃
8	8	BS (backspace)	退格
9	9	HT (horizontal tab)	水平制表符
10	0A	LF (NL line feed, new line)	换行键
11	0B	VT (vertical tab)	垂直制表符
12	0C	FF (NP form feed, new page)	换页键
13	0D	CR (carriage return)	回车键
14	0E	SO (shift out)	不用切换
15	0F	SI (shift in)	启用切换
16	10	DLE (data link escape)	数据链路转义
17	11	DC1 (device control 1)	设备控制 1
18	12	DC2 (device control 2)	设备控制 2
19	13	DC3 (device control 3)	设备控制 3
20	14	DC4 (device control 4)	设备控制 4
21	15	NAK (negative acknowledge)	拒绝接收
22	16	SYN (synchronous idle)	同步空闲
23	17	ETB (end of trans. block)	传输块结束
24	18	CAN (cancel)	取消
25	19	EM (end of medium)	介质中断
26	1A	SUB (substitute)	替补
27	1B	ESC (escape)	换码 (溢出)
28	1C	FS (file separator)	文件分割符
29	1D	GS (group separator)	分组符
30	1E	RS (record separator)	记录分离符

下页继续

表 1 – 续上页

31	1F	US (unit separator)	单元分隔符

表 27-2 ASCII 码中的字符及数字

十进制	十六进制	缩　写/字符	十进制	十六进制	缩　写/字符		
32	20	(space) 空格	80	50	P		
33	21	!	81	51	Q		
34	22	“	82	52	R		
35	23	#	83	53	S		
36	24	$	84	54	T		
37	25	%	85	55	U		
38	26	&	86	56	V		
39	27	‘	87	57	W		
40	28	(	88	58	X		
41	29	)	89	59	Y		
42	2A	*	90	5A	Z		
43	2B	•	91	5B	[		
44	2C	,	92	5C			
45	2D	•	93	5D	]		
46	2E	.	94	5E	^		
47	2F	/	95	5F	_		
48	30	0	96	60	‘		
49	31	1	97	61	a		

下页继续

表 2 – 续上页

50	32	2	98	62	b		
51	33	3	99	63	c		
52	34	4	100	64	d		
53	35	5	101	65	e		
54	36	6	102	66	f		
55	37	7	103	67	g		
56	38	8	104	68	h		
57	39	9	105	69	i		
58	3A	:	106	6A	j		
59	3B	;	107	6B	k		
60	3C	<	108	6C	l		
61	3D	=	109	6D	m		
62	3E	>	110	6E	n		
63	3F	?	111	6F	o		
64	40	@	112	70	p		
65	41	A	113	71	q		
66	42	B	114	72	r		
67	43	C	115	73	s		
68	44	D	116	74	t		
69	45	E	117	75	u		
70	46	F	118	76	v		
71	47	G	119	77	w		
72	48	H	120	78	x		
73	49	I	121	79	y		
74	4A	J	122	7A	z		
75	4B	K	123	7B	{		
76	4C	L	124	7C			

下页继续

表 2 – 续上页

77	4D	M	125	7D	}		
78	4E	N	126	7E	~		
79	4F	O	127	7F	DEL	(delete)	删除

后来，计算机引进到其它国家的时候，由于他们使用的不是英语，他们使用的字母在 ASCII 码表中没有定义，所以他们采用 127 号之后的位来表示这些新的字母，还加入了各种形状，一直编号到 255。从 128 到 255 这些字符被称为 ASCII 扩展字符集。至此基本存储单位 Byte(char) 能表示的编号都被用完了。

26.1.2 中文编码

由于英文书写系统都是由 26 个基本字母组成，利用 26 个字母组可合出不同的单词，所以用 ASCII 码表就能表达整个英文书写系统。而中文书写系统中的汉字是独立的方块，若参考单词拆解成字母的表示方式，汉字可以拆解成部首、笔画来表示，但这样会非常复杂 (可参考五笔输入法编码)，所以中文编码直接对方块字进行编码，一个汉字使用一个号码。

由于汉字非常多，常用字就有 6000 多个，如果像 ASCII 编码表那样只使用 1 个字节最多只能表示 256 个汉字，所以我们使用 2 个字节来编码。

26.1.2.1 GB2312 标准

我们首先定义的是 GB2312 标准。它把 ASCII 码表 127 号之后的扩展字符集直接取消掉，并规定小于 127 的编码按原来 ASCII 标准解释字符。当 2 个大于 127 的字符连在一起时，就表示 1 个汉字，第 1 个字节使用 (0xA1-0xFE) 编码，第 2 个字节使用 (0xA1-0xFE) 编码，这样的编码组合起来可以表示了 7000 多个符号，其中包含 6763 个汉字。在这些编码里，我们还把数学符号、罗马字母、日文假名等都编进表中，就连原来在 ASCII 里原本就有的数字、标点以及字母也重新编了 2 个字节长的编码，这就是平时在输入法里可切换的“全角”字符，而标准的 ASCII 码表中 127 号以下的就被称为“半角”字符。

表 27-3 说明了 GB2312 是如何兼容 ASCII 码的，当我们设定系统使用 GB2312 标准的时候，它遇到一个字符串时，会按字节检测字符值的大小，若遇到连续两个字节的数值都大于 127 时就把这

两个连续的字节合在一起，用 GB2312 解码，若遇到的数值小于 127，就直接用 ASCII 把它解码。

表 27-3 GB2312 兼容 ASCII 码的原理

第 1 字节	第 2 字节	表示的字符	说明
0x68	0x69	(hi)	两个字节的值都小于 127(0x7F)，使用 ASCII 解码
0xB0	0xA1	(啊)	两个字节的值都大于 127(0x7F)，使用 GB2312 解码

26.1.2.1.1 区位码

在 GB2312 编码的实际使用中，有时会用到区位码的概念，见图 *27_1*。GB2312 编码对所收录字符进行了“分区”处理，共 94 个区，每区含有 94 个位，共 8836 个码位。而区位码实际是 GB2312 编码的内部形式，它规定对收录的每个字符采用两个字节表示，第一个字节为“高字节”，对应 94 个区；第二个字节为“低字节”，对应 94 个位。所以它的区位码范围是：0101-9494。为兼容 ASCII 码，区号和位号分别加上 0xA0 偏移就得到 GB2312 编码。在区位码上加上 0xA0 偏移，可求得 GB2312 编码范围：0xA1A1 - 0xFEFE，其中汉字的编码范围为 0xB0A1-0xF7FE，第一字节 0xB0-0xF7（对应区号：16 - 87），第二个字节 0xA1-0xFE（对应位号：01 - 94）。

例如，“啊”字是 GB2312 编码中的第一个汉字，它位于 16 区的 01 位，所以它的区位码就是 1601，加上 0xA0 偏移，其 GB2312 编码为 0xB0A1。其中区位码为 0101 的码位表示的是“空格”符。

图 27-1 GB2312 的部分区位码

26.1.2.2 GBK 编码

据统计，GB2312 编码中表示的 6763 个汉字已经覆盖中国大陆 99.75% 的使用率，单看这个数字已经很令人满意了，但是我们不能因为那些文字不常用就不让它进入信息时代，而且生僻字在人名、文言文中的出现频率是非常高的。为此我们在 GB2312 标准的基础上又增加了 14240 个新汉字 (包括所有后面介绍的 Big5 中的所有汉字) 和符号，这个方案被称为 GBK 标准。增加这么多字符，按照 GB2312 原来的格式来编码，2 个字节已经没有足够的编码，我们聪明的程序员修改了一下格式，不再要求第 2 个字节的编码值必须大于 127，只要第 1 个字节大于 127 就表示这是一个汉字的开始，这样就做到了兼容 ASCII 和 GB2312 标准。

表 27-4 说明了 GBK 是如何兼容 ASCII 和 GB2312 标准的，当我们设定系统使用 GBK 标准的时

候，它按顺序遍历字符串，按字节检测字符值的大小，若遇到一个字符的值大于 127 时，就再读取它后面的一个字符，把这两个字符值合在一起，用 GBK 解码，解码完后，再读取第 3 个字符，重新开始以上过程，若该字符值小于 127，则直接用 ASCII 解码。

表 27-4 GBK 兼容 ASCII 和 GB2312 的原理

第 1 字节	第 2 字节	第 3 字节	表示的字符	说明
0x68(<7F)	0xB0(>7F)	0xA1(>7F)	h 啊)	第 1 个字节小于 127，使用 ASCII 解码，每 2 个字节大于 127，直接使用 GBK 解码，兼容 GB2312
0xB0(>7F)	0xA1(>7F)	0x68(<7F)	啊 h)	第 1 个字节大于 127，直接使用 GBK 码解释，第 3 个字节小于 127，使用 ASCII 解码
0xB0(>7F)	0x56(<7F)	0x68(<7F)	□ h)	第 1 个字节大于 127，第 2 个字节虽然小于 127，直接使用 GBK 解码，第 3 个字节小于 127，使用 ASCII 解码

26.1.2.3 GB18030

随着计算机技术的普及，我们后来又在 GBK 的标准上不断扩展字符，这些标准被称为 GB18030，如 GB18030-2000、GB18030-2005 等 (“-” 号后面的数字是制定标准时的年号)，GB18030 的编码使用 4 个字节，它利用前面标准中的第 2 个字节未使用的 “0x30-0x39” 编码表示扩充四字节的后缀，兼容 GBK、GB2312 及 ASCII 标准。

GB18030-2000 主要在 GBK 基础上增加了 “CJK(中日韩) 统一汉字扩充 A” 的汉字。加上前面 GBK 的内容，GB18030-2000 一共规定了 27533 个汉字（包括部首、部件等）的编码，还有一些常用非汉字符号。

GB18030-2005 的主要特点是在 GB18030-2000 基础上增加了 “CJK(中日韩) 统一汉字扩充 B” 的汉字。增加了 42711 个汉字和多种我国少数民族文字的编码（如藏、蒙古、傣、彝、朝鲜、维吾尔文等）。加上前面 GB18030-2000 的内容，一共收录了 70244 个汉字。

GB2312、GBK 及 GB18030 是汉字的国家标准编码，新版向下兼容旧版，各个标准简要说明见表 27-5，目前比较流行的是 GBK 编码，因为每个汉字只占用 2 个字节，而且它编码的字符已经能

满足大部分的需求，但国家要求一些产品必须支持 GB18030 标准。

表 27-5 汉字国家标准

类别	编码范围	汉字编码范围	扩充汉字数	说明
GB2312	第一字节 0xA1-0xFE	第一字节 0xB0-0xF7	6763	除汉字外，还包括拉丁字母、希腊字母、日文平假名及片假名字母、俄语西里尔字母在内的 682 个全角字符
	第二字节 0xA1-0xFE	第二字节 0xA1-0xFE		
GBK	第一字节 0x81-0xFE	第一字节 0x81-0xA0	6080	包括部首和构件, 中日韩汉字, 包含了 BIG5 编码中的所有汉字, 加上 GB2312 的原内容，一共有 21003 个汉字
	第二字节 0x40-0xFE	第二字节 0x40-0xFE		
		第一字节 0xAA-0xFE	8160	
		第二字节 0x40-0xA0		
GB18030-2000	第一字节 0x81-0xFE	第一字节 0x81-0x82	6530	在 GBK 基础上增加了中日韩统一汉字扩充 A 的汉字，加上 GB2312、GBK 的内容，一共有 27533 个汉字
	第二字节	第二字节		

26.1.2.4 Big5 编码

在台湾、香港等地区，使用较多的是 Big5 编码，它的主要特点是收录了繁体字。而从 GBK 编码开始，已经把 Big5 中的所有汉字收录进编码了。即对于汉字部分，GBK 是 Big5 的超集，Big5 能表示的汉字，在 GBK 都能找到那些字相应的编码，但他们的编码是不一样的，两个标准不兼容，如 GBK 中的“啊”字编码是“0xB0A1”，而 Big5 标准中的编码为“0xB0DA”。

26.1.3 Unicode 字符集和编码

由于各个国家或地区都根据使用自己的文字系统制定标准，同一个编码在不同的标准里表示不一样的字符，各个标准互不兼容，而又没有一个标准能够囊括所有的字符，即无法用一个标准表达所有字符。国际标准化组织 (ISO) 为解决这一问题，它舍弃了地区性的方案，重新给全球上所有文化使用的字母和符号进行编号，对每个字符指定一个唯一的编号 (ASCII 中原有的字符编号不变)，这些字符的号码从 0x000000 到 0x10FFFF，该编号集被称为 Universal Multiple-Octet Coded Character Set，简称 UCS，也被称为 Unicode。最新版的 Unicode 标准还包含了表情符号 (聊天软件中的部分 emoji 表情)，可访问 Unicode 官网了解：http://www.unicode.org。

Unicode 字符集只是对字符进行编号，但具体怎么对每个字符进行编码，Unicode 并没指定，因此也衍生出了如下几种 unicode 编码方案 (Unicode Transformation Format)。

26.1.4 UTF-32

对 Unicode 字符集编码，最自然的就是 UTF-32 方式了。编码时，它直接对 Unicode 字符集里的每个字符都用 4 字节来表示，转换方式很简单，直接将字符对应的编号数字转换为 4 字节的二进制数。如表 27-6，由于 UTF-32 把每个字符都用要 4 字节来存储，因此 UTF-32 不兼容 ASCII 编码，也就是说 ASCII 编码的文件用 UTF-32 标准来打开会成为乱码。

表 27-6 UTF-32 编码示例

字符	GBK 编码	Unicode 编号	UTF-32 编码
A	0x41	0x0000 0041	大端格式 0x0000 0041
啊	0xB0A1	0x0000 554A	大端格式 0x0000 554A

对 UTF-32 数据进行解码的时候，以 4 个字节为单位进行解析即可，根据编码可直接找到 Unicode 字符集中对应编号的字符。

UTF-32 的优点是编码简单，解码也很方便，读取编码的时候每次都直接读 4 个字节，不需要加其它的判断。它的缺点是浪费存储空间，大量常用字符的编号只需要 2 个字节就能表示。其次，在存储的时候需要指定字节顺序，是高位字节存储在前 (大端格式)，还是低位字节存储在前 (小端格式)。

26.1.5 UTF-16

针对 UTF-32 的缺点，人们改进出了 UTF-16 的编码方式，它采用 2 字节或 4 字节的变长编码方式 (UTF-32 定长为 4 字节)。对 Unicode 字符编号在 0 到 65535 的统一用 2 个字节来表示，将每个字符的编号转换为 2 字节的二进制数，即从 0x0000 到 0xFFFF。而由于 Unicode 字符集在 0xD800-0xDBFF 这个区间是没有表示任何字符的，所以 UTF-16 就利用这段空间，对 Unicode 中编号超出 0xFFFF 的字符，利用它们的编号做某种运算与该空间建立映射关系，从而利用该空间表示 4 字节扩展，感兴趣的读者可查阅相关资料了解具体的映射过程。

表 27-7 UTF-16 编码示例

字符	GB18030 编码	Unicode 编号	UTF-16 编码
A	0x41	0x0000 0041	大端格式 0x0041
啊	0xB0A1	0x0000 554A	大端格式 0x554A
🄵	0x9735 F832	0x0002 75CC	大端格式 0xD85D DDCC

注：🄵五笔：TLHH(不支持 GB18030 码的输入法无法找到该字，感兴趣可搜索它的 Unicode 编号找到)

UTF-16 解码时，按两个字节去读取，如果这两个字节不在 0xD800 到 0xDFFF 范围内，那就是双字节编码的字符，以双字节进行解析，找到对应编号的字符。如果这两个字节在 0xD800 到 0xDFFF 之间，那它就是四字节编码的字符，以四字节进行解析，找到对应编号的字符。

UTF-16 编码的优点是相对 UTF-32 节约了存储空间，缺点是仍不兼容 ASCII 码，仍有大小端格式问题。

26.1.6 UTF-8

UTF-8 是目前 Unicode 字符集中使用得最广的编码方式，目前大部分网页文件已使用 UTF-8 编码，如使用浏览器查看百度首页源文件，可以在前几行 HTML 代码中找到如下代码：

1 <meta http-equiv=Content-Type content=”text/html;charset=utf-8”>

其中“charset”等号后面的“utf-8”即表示该网页字符的编码方式 UTF-8。

UTF-8 也是一种变长的编码方式，它的编码有 1、2、3、4 字节长度的方式，每个 Unicode 字符根据自己的编号范围去进行对应的编码，见表 27-8。它的编码符合以下规律：

- 对于 UTF-8 单字节的编码，该字节的第 1 位设为 0(从左边数起第 1 位，即最高位)，剩余的位用来写入字符的 Unicode 编号。即对于 Unicode 编号从 0x0000 0000-0x0000 007F 的字符，UTF-8 编码只需要 1 个字节，因为这个范围 Unicode 编号的字符与 ASCII 码完全相同，所以 UTF-8 兼容了 ASCII 码表。
- 对于 UTF-8 使用 N 个字节的编码 (N>1)，第一个字节的前 N 位设为 1，第 N+1 位设为 0，后面字节的前两位都设为 10，这 N 个字节的其余空位填充该字符的 Unicode 编号，高位用 0 补足。

表 27-8 UTF-8 编码原理 (x 的位置用于填充 Unicode 编号)

Unicode(16 进制)	UTF-8（2 进制）				
编号范围	第一字节	第二字节	第三字节	第四字节	第五字节
00000000-0000007F	0xxxxxxx				
00000080-000007FF	110xxxxx	10xxxxxx			
00000800-0000FFFF	1110xxxx	10xxxxxx	10xxxxxx		
00010000-0010FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	
…	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

注：实际上 utf-8 编码长度最大为四个字节，所以最多只能表示 Unicode 编码值的二进制数为 21 位的 Unicode 字符。但是已经能表示所有的 Unicode 字符，因为 Unicode 的最大码位 0x10FFFF 也只有 21 位。

UTF-8 解码的时候以字节为单位去看，如果第一个字节的 bit 位以 0 开头，那就是 ASCII 字符，以

单字节进行解析。如果第一个字节的数据位以“110”开头，就按双字节进行解析，3、4 字节的解析方法类似。

UTF-8 的优点是兼容了 ASCII 码，节约空间，且没有字节顺序的问题，它直接根据第 1 个字节前面数据位中连续的 1 个数决定后面有多少个字节。不过使用 UTF-8 编码汉字平均需要 3 个字节，比 GBK 编码要多一个字节。

26.1.7 BOM

由于 UTF 系列有多种编码方式，而且对于 UTF-16 和 UTF-32 还有大小端的区分，那么计算机软件在打开文档的时候到底应该用什么编码方式去解码呢？有的人就想到在文档最前面加标记，一种标记对应一种编码方式，这些标记就叫做 BOM(Byte Order Mark)，它们位于文本文件的开头，见表 27-9。注意 BOM 是对 Unicode 的几种编码而言的，ANSI 编码没有 BOM。

表 27-9 BOM 标记

BOM 标记	表示的编码
0xEF 0xBB 0xBF	UTF-8
0xFF 0xFE	UTF-16 小端格式
0xFE 0xFF	UTF-16 大端格式
0xFF 0xFE 0x00 0x00	UTF-32 小端格式
0x00 0x00 0xFE 0xFF	UTF-32 大端格式

但由于带 BOM 的设计很多规范不兼容，不能跨平台，所以这种带 BOM 的设计没有流行起来。Linux 系统下默认不带 BOM。

26.2 什么是字模？

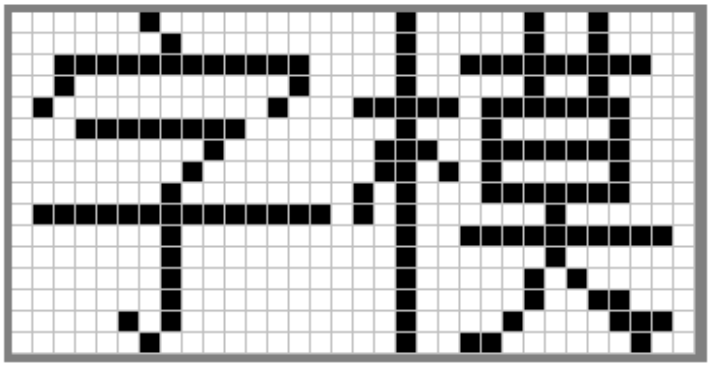
有了编码，我们就能在计算机中处理、存储字符了，但是如果计算机处理完字符后直接以编码的形式输出，人类将难以识别。来，在 2 秒内告诉我 ASCII 编码的“0x25”表示什么字符？不容易

吧？要是觉得容易，再来告诉我 GBK 编码的 “0xBCC6” 表示什么字符。因此计算机与人交互时，一般会把字符转化成人类习惯的表现形式进行输出，如显示、打印的时候。

但是如果仅有字符编码，计算机还不知道该如何表达该字符，因为字符实际上是一个个独特的图形，计算机必须把字符编码转化成对应的字符图形人类才能正常识别，因此我们要给计算机提供字符的图形数据，这些数据就是字模，多个字模数据组成的文件也被称为字库。计算机显示字符时，根据字符编码与字模数据的映射关系找到它相应的字模数据，液晶屏根据字模数据显示该字符。

26.2.1 字模的构成

已知字模是图形数据，而图形在计算机中是由一个个像素点组成的，所以字模实质是一个个像素点数据。为方便处理，我们把字模定义成方块形的像素点阵，且每个像素点只有 0 和 1 这两种状态 (可以理解为单色图像数据)。见图 *27_2*，这是两个宽、高为 16x 16 的像素点阵组成的两个汉字图形，其中的黑色像素点即为文字的笔迹。计算机要表示这样的图形，只需使用 16x16 个二进制数据位，每个数据位记录一个像素点的状态，把黑色像素点以 “1” 表示，无色像素点以 “0” 表示即可。这样的一个汉字图形，使用 16x16/8=32 个字节来就可以记录下来。



16x16 的“字”的字模数据以 C 语言数组的方式表示，见代码清单 27_1。在这样的字模中，以两个字节表示一行像素点，16 行构成一个字模。

列表 1: 代码清单 27-1：“字”的字模数据

```
/* 字 */
unsigned char code Bmp003[]=
{
/*--------------------------------------------------------------
;  源文件 / 文字 : 字
;  宽 × 高（像素）: 16×16
;  字模格式/大小 : 单色点阵液晶字模，横向取模，字节正序/32 字节
--------------------------------------------------------------*/

0x02,0x00,0x01,0x00,0x3F,0xFC,0x20,0x04,0x40,0x08,0x1F,0xE0,0x00,0x40,
0x00,0x80,
0xFF,0xFF,0x7F,0xFE,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x05,0x00,
0x02,0x00,
};
```

26.2.2 字模显示原理

如果使用 LCD 的画点函数，按位来扫描这些字模数据，把为 1 的位以黑色来显示 (也可以使用其它颜色)，为 0 的数据位以白色来显示，即可把整个点阵还原出来，显示在液晶屏上。

为便于理解，我们编写了一个使用串口 printf 利用字模打印字符到串口上位机，见代码清单 27_2 中演示的字模显示原理。

列表 2: 代码清单 27-2 使用串口利用字模打印字到上位机

```
/*" 当" 字符的字模 16x16 */
unsigned char charater_matrix[] = {
```

(下页继续)

(续上页)

```
    /*" 当",0*/
    0x01,0x00,0x21,0x08,0x11,0x08,0x09,0x10,
    0x09,0x20,0x01,0x00,0x7F,0xF8,0x00,0x08,
    0x00,0x08,0x00,0x08,0x3F,0xF8,0x00,0x08,
    0x00,0x08,0x00,0x08,0x7F,0xF8,0x00,0x08,
};

/**
* @brief  使用串口在上位机打印字模
*         演示字模显示原理
* @retval 无
*/
void Printf_Charater(void)
{
    int i,j;
    unsigned char kk;

    /*i 用作行计数 */
    for ( i=0; i<16; i++) {
        /*j 用作一字节内数据的移位计数 */
        /* 一行像素的第一个字节 */
        for (j=0; j<8; j++) {
              /* 一个数据位一个数据位地处理 */
              kk = charater_matrix[2*i] << j ;  //左移 J 位
              if ( kk & 0x80) {
                 printf("*"); //如果最高位为 1，输出 * 号，表示笔迹
              } else {
                 printf(" "); //如果最高位为 0，输出空格，表示空白
              }
        }
        /* 一行像素的第二个字节 */
        for (j=0; j<8; j++) {
```

(下页继续)

(续上页)

```
            kk = charater_matrix[2*i+1] << j ;  //左移 J 位
            if ( kk & 0x80) {
               printf("*"); //如果最高位为 1，输出 * 号，表示笔迹
            } else {
               printf(" "); //如果最高位为 0，输出空格，表示空白
            }
      }
      printf("\n");     //输出完一行像素，换行
   }
   printf("\n\n");      //一个字输出完毕
}
```

在 main 函数中运行这段代码，连接好开发板到上位机，可以看到图 *27_3* 中的现象。该函数中利用 printf 函数对字模数据中为 1 的数据位打印“*”号，为 0 的数据位打印出“空格”，从而在串口接收区域中使用“*”号表达出了一个“当”字。

26.2.3 如何制作字模

以上只是某几个字符的字模，为方便使用，我们需要制作所有常用字符的字模，如程序只需要英文显示，那就需要制作包含 ASCII 码表 27-2 中所有字符的字模，如程序只需要使用一些常用汉字，我们可以选择制作 GB2312 编码里所有字符的字模，而且希望字模数据与字符编码有固定的映射关系，以便在程序中使用字符编码作为索引，查找字模。在网上搜索可找到一些制作字模的软件工具，可满足这些需求。在我们提供的《液晶显示中英文》的工程目录下提供了一个取模软件“PCtoLCD”，这里以它为例讲解如何制作字模，其它字模软件也是类似的。

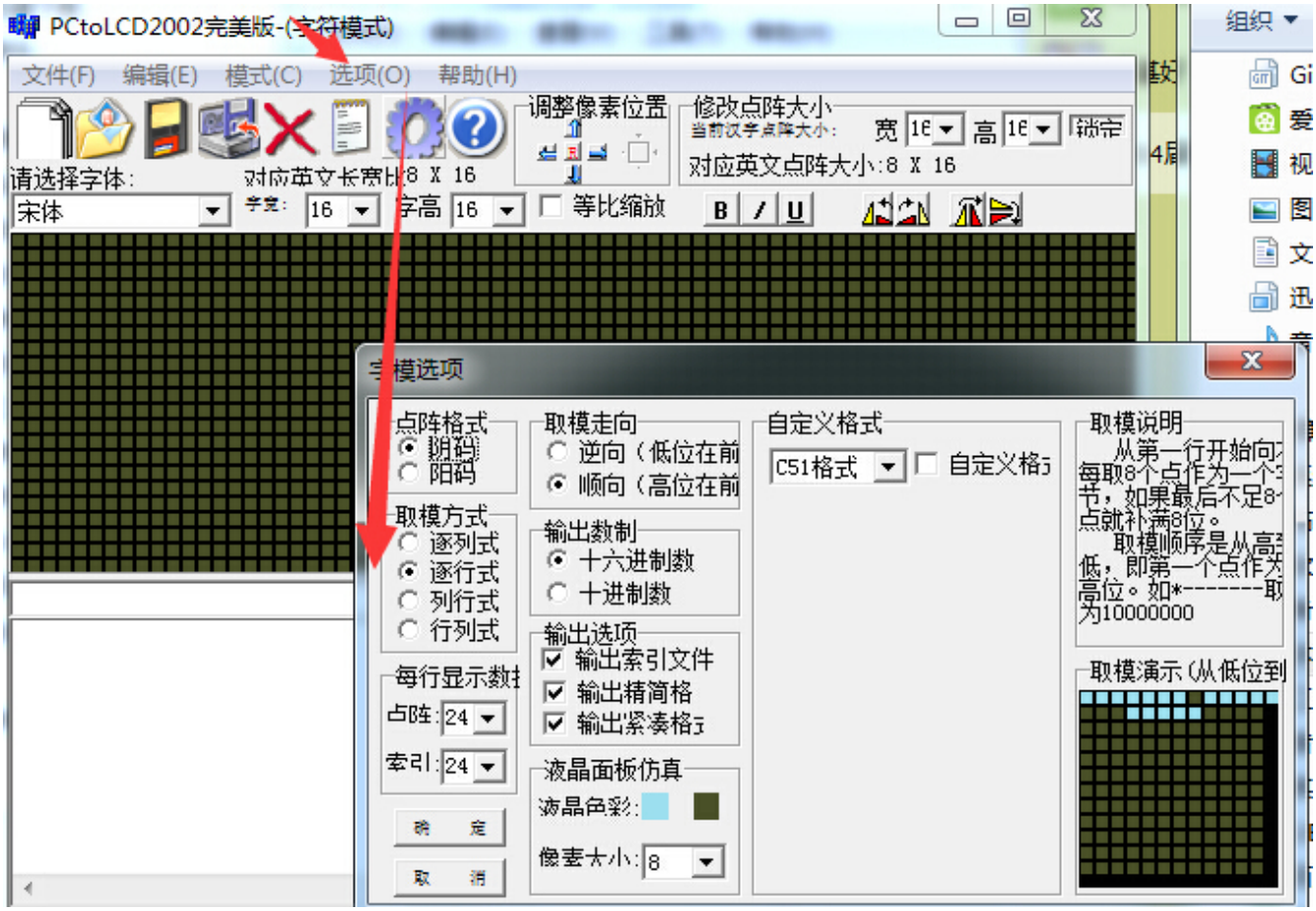
(1) 配置字模格式

打开取模软件，点击“选项”菜单，会弹出一个对话框，见图 27-4。

- 选项“点阵格式”中的阴、阳码是指字模点阵中有笔迹像素位的状态是“1”还是“0”，像我们前文介绍的那种就是阴码，反过来就是阳码。本工程中使用阴码。

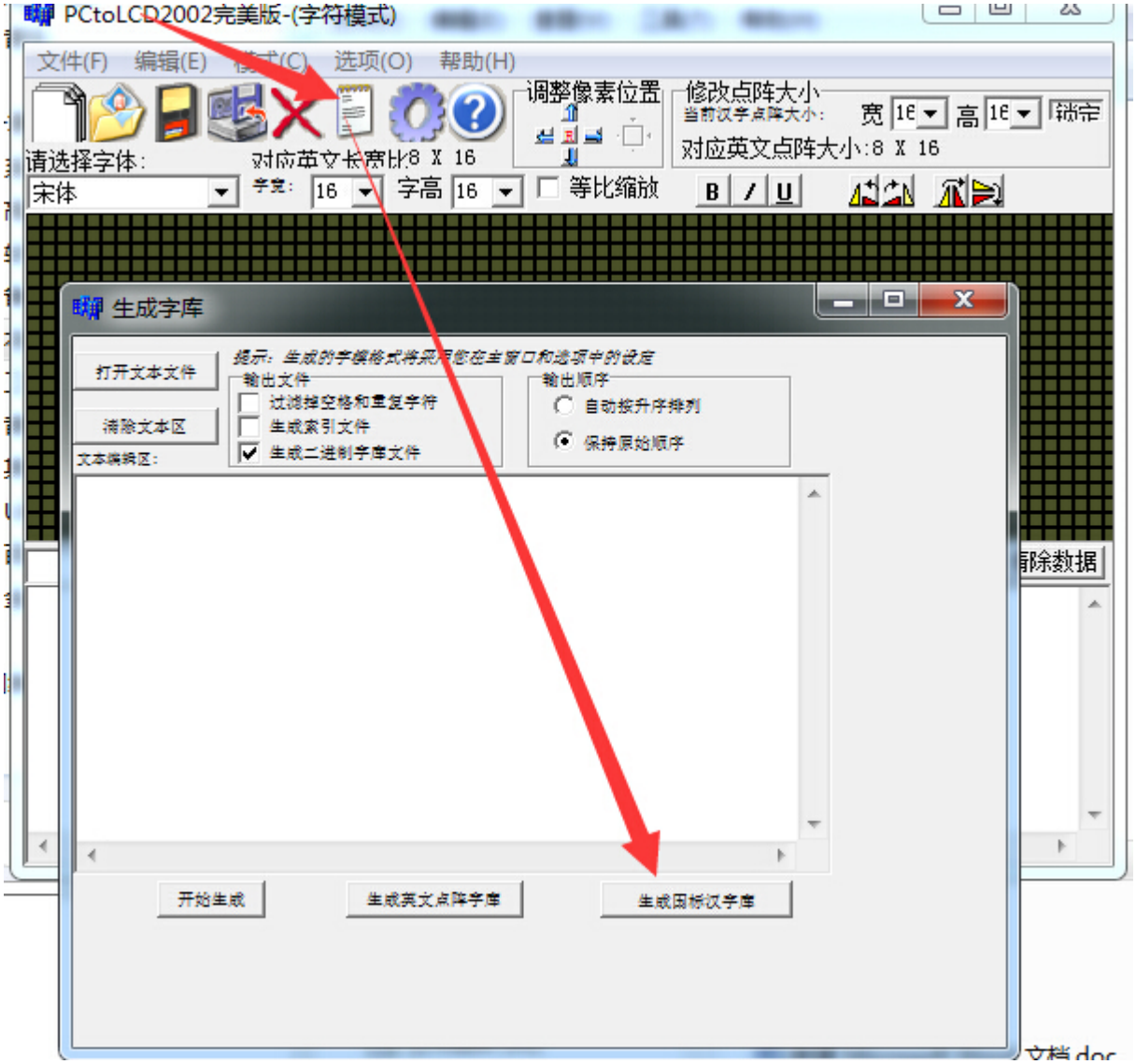
- 选项“取模方式”是指字模图形的扫描方向，修改这部分的设置后，选项框的右侧会有相应的说明及动画显示，这里我们依然按前文介绍的字模类型，把它配置成“逐行式”
- 选项“每行显示的数据”里我们把点阵和索引都配置成 16，设置这个点阵的像素大小为 16x16。

字模选项的格式保持不变，设置完我们点击确定即可，字模选项的这些配置会影响到显示代码的编写方式，即类似前文*代码清单 27_2* 中的程序。



(2) 生成 GB2312 字模

配置完字模选项后，点击软件中的导入文本图标，会弹出一个“生成字库”的对话框，点击右下角的生成国标汉字库按钮即可生成包含了 GB2312 编码里所有字符的字模文件。在《液晶显示中英文》的工程目录下的《GB2312_H1616.FON》是我用这个取模软件生成的字模原文件，若不想自己制作字模，可直接使用该文件。



26.2.4 字模寻址公式

使用字模软件制作的字模数据一般会按照编码格式排列。如我们利用以上软件生成的字模文件《GB2312_H2424.FON》中的数据，是根据 GB2312 的区位码表的顺序存储的，它存储了区位码为 0101-9494 的字符，每个字模的大小为 16x16/8=32 字节。其中第一个字符“空格”的区位码为 0101，它是首个字符，所以文件的前 32 字节存储的是它的字模数据；同理，32-64 字节存储的则是 0102 字符“、”的字模数据。所以我们可以导出任意字符的寻址公式:

Addr = (((Code:sub:*H*-0xA0-1)*94) +($Code_L$-0xA0-1))*16*16/8

其中 $Code_H$ 和 $Code_L$ 分别是 GB2312 编码的第一字节和第二字节；94 是指一个区中有 94 个位 (即 94 个字符)。公式的实质是根据字符的 GB2312 编码，求出区位码，然后区位码乘以每个字符占据的字节数，求出地址偏移。

26.2.5 存储字模文件

上面生成的《GB2312_H1616.FON》文件的大小为 256KB，比很多 STM32 芯片内部的所有 FLASH 空间都大，如果我们还是在程序中直接以 C 语言数组的方式存储字模数据，STM32 芯片的程序空间会非常紧张，一般的做法是把字模数据存储到外部存储器，如 SD 卡或 SPI-FLASH 芯片，当需要显示某个字符时，控制器根据字符的编码算好字模的存储地址，再从存储器中读取，而 FLASH 芯片在生产前就固化好字模内容，然后直接把 FLASH 芯片贴到电路板上，作为整个系统的一部分。

26.3 各种模式的液晶显示字符实验

本小节讲解如何利用字模使用在液晶屏上显示字符。

根据编码或字模存储位置、使用方式的不同，讲解中涉及到多个工程，见表 27-10 中的说明，在讲解特定实验的时候，请读者打开相应的工程来阅读

表 27-10 各种模式的液晶显示字符实验

工程名称	说明
液晶显示	仅包含 ASCII 码字符显示功能，字库直接以 C 语言常量数组的方式存储在 STM32 芯片的内部 FLASH 空间
液晶显示中英文（字库在外部 FLASH）	包含 ASCII 码字符及 GB2312 码字符的显示功能，ASCII 码字符存储在 STM32 内部 FLASH，GB2312 码字符存储在外部 SPI-FLASH 芯片
液晶显示中英文（字库在 SD 卡）	包含 ASCII 码字符及 GB2312 码字符的显示功能，ASCII 码字符存储在 STM32 内部 FLASH，GB2312 码字符直接以文件的格式存储在 SD 卡中
液晶显示中英文（任意大小）	在基础字库的支持下，使用字库缩放函数，使得只用一种字库，就能显示任意大小的字符。包含 ASCII 码字符及 GB2312 码字符的显示功能，ASCII 码字符存储在 STM32 内部 FLASH，GB2312 码字符存储在外部 SPI-FLASH 芯片

这些实验是在《液晶显示》工程的基础上修改的，主要添加字符显示相关的内容，本小节只讲解这部分新增的函数。关于液晶驱动的原理在此不再赘述，不理解这部分的可阅读前面的章节。

26.3.1 硬件设计

针对不同模式的液晶显示字符工程，需要有不同的硬件支持。字库存储在 STM32 芯片内部 FLASH 的工程跟普通液晶显示的硬件需求无异。需要外部字库的工程，要有额外的 SPI-FLASH、SD 支持，使用外部 FLASH 时，我们的实验板上直接用板子上的 SPI-FLASH 芯片存储字库，出厂前我们已给 FLASH 芯片烧录了前面的《GB2312_H1616.FON》字库文件，如果您想把我们的程序移植到您自己设计产品上，请确保该系统包含有存储了字库的 FLASH 芯片，才能正常显示汉字使用 SD 卡时，需要给板子接入存储有《GB2312_H1616.FON》字库文件的 MicroSD 卡，SD 卡的文件系统格式需要是 FAT 格式，且字库文件所在的目录需要跟程序里使用的文件目录一致。

关于 SPI-FLASH 和 SD 卡的原理图及驱动说明可参考其他的章节。给外部 SPI-FLASH 和 SD 卡存储字库的操作我们将在另一个文档中说明，本章的教程默认您已配置好 SPI-FLASH 和 SD 卡相关的字库环境。

26.3.2 显示 ASCII 编码的字符

我们先来看如何显示 ASCII 码表中的字符，请打开“液晶显示”（即）的工程文件。本工程中我们把字库数据相关的函数代码写在“fonts.c”及“fonts.h”文件中，字符显示的函数仍存储在 LCD 驱动文件“bsp_ili9341_lcd.c”及“bsp_ ili9341_lcd.h”中。

26.3.2.1 编程要点

(1) 获取字模数据；

(2) 根据字模格式，编写液晶显示函数；

(3) 编写测试程序，控制液晶英文。

26.3.2.2 代码分析

26.3.2.2.1 ASCII 字模数据

要显示字符首先要有字库数据，在工程的“fonts.c”文件中我们定义了一系列大小为 24x32、16x24、8x16 的 ASCII 码表的字模数据，其形式见*代码清单 27_3*。

列表 3: 代码清单 27-3 部分英文字库 16x24 大小 (fonts.c 文件)

```
/*
* 常用 ASCII 表，偏移量 32，大小:24（高度）* 16 （宽度）
*/
//@conslons 字体，阴码点阵格式，逐行顺向取摸
const uint8_t ASCII16x24_Table [ ] = {
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

(下页继续)

(续上页)

```
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x03,0x80,0x01,0x80,0x01,0x80,
    0x01,0x80,0x01,0x80,0x01,0x80,0x01,0x80,
    0x01,0x80,0x01,0x80,0x01,0x80,0x00,0x00,
    0x00,0x00,0x03,0xc0,0x03,0xc0,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

    /* 以下部分省略....., 包含从 空格 至 波浪号 的 ASCII 码图形字模数据 */
```

由于 ASCII 中的字符并不多，所以本工程中直接以 C 语言数组的方式存储这些字模数据，C 语言的 const 数组是作为常量直接存储到 STM32 芯片的内部 FLASH 中的，所以如果您不需要显示中文，可以不用外部的 SPI-FLASH 芯片，可省去烧录字库的麻烦。以上代码定义的 ASCII16x24_Table 数组是 16x24 大小的 ASCII 字库。

26.3.2.2.2 管理英文字模的结构体

为了方便使用各种不同的字体，工程中定义了一个“sFont”结构体类型，并利用它定义存储了不同字体信息的结构体变量，见*代码清单 27_4*。

列表 4: 代码清单 27-4 管理英文字模的结构体 (fonts.c 文件)

```
/* 字体格式 */
typedef struct _tFont {
    const uint8_t *table; /* 指向字模数据的指针 */
    uint16_t Width;    /* 字模的像素宽度 */
    uint16_t Height;     /* 字模的像素高度 */

} sFONT;
```

(下页继续)

(续上页)

```
sFONT Font8x16 = {
   ASCII8x16_Table,
   8, /* 字模宽 */
   16, /* 字模高 */
};

sFONT Font16x24 = {
   ASCII16x24_Table,
   16, /* 字模宽 */
   24, /* 字模高 */
};

sFONT Font24x32 = {
   ASCII24x32_Table,
   24, /* 字模宽 */
   32, /* 字模高 */
};
```

这个结构体类型定义了三个变量，第一个是指向字模数据的指针，即前面提到的 C 语言数组，每二、三个变量存储了该字模单个字符的像素宽度和高度。利用这个类型定义了 Font8x16、Font16x24 之类的变量，方便显示时寻址。

26.3.2.2.3 切换字体

在程序中若要方便切换字体，还需要定义一个存储了当前选择字体的变量 LCD_Currentfonts，*代码清单 27_5*。

列表 5: 代码清单 27-5 切换字体 (bsp_lcd.c 文件)

```
/* 用于存储当前选择的字体格式的全局变量 */
static sFONT *LCD_Currentfonts;
/**
* @brief  设置字体格式 (英文)
* @param  fonts: 选择要设置的字体格式
* @retval None
*/
void LCD_SetFont(sFONT *fonts)
{
   LCD_Currentfonts = fonts;
}
```

使用 LCD_SetFont 可以切换 LCD_Currentfonts 指向的字体类型，函数的可输入参数即前面的 Font8x16、Font16x24 之类的变量。

26.3.2.2.4 ASCII 字符显示函数

利用字模数据以及上述结构体变量，我们可以编写一个能显示各种字体的通用函数，见代码清单 27_6。

列表 6: 代码清单 27-6 ASCII 字符显示函数

```
/**
* @brief  在 ILI9341 显示器上显示一个英文字符
* @param  usX : 在特定扫描方向下字符的起始 X 坐标
* @param  usY : 在特定扫描方向下该点的起始 Y 坐标
* @param  cChar : 要显示的英文字符
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
* @retval 无
*/
```

(下页继续)

(续上页)

```
void ILI9341_DispChar_EN ( uint16_t usX, uint16_t usY, const char cChar )
{
   uint8_t  byteCount, bitCount,fontLength;
   uint16_t ucRelativePositon;
   uint8_t *Pfont;

   //对 ascii 码表偏移 (字模表不包含 ASCII 表的前 32 个非图形符号)
   ucRelativePositon = cChar - ' ';

   //每个字模的字节数
   fontLength = (LCD_Currentfonts->Width*LCD_Currentfonts->Height)/8;

   //字模首地址
   /*ascii 码表偏移值乘以每个字模的字节数，求出字模的偏移位置 */
   Pfont = (uint8_t *)&LCD_Currentfonts->table[ucRelativePositon *␣
↪fontLength];

   //设置显示窗口
ILI9341_OpenWindow ( usX, usY, LCD_Currentfonts->Width, LCD_Currentfonts->
↪Height);

   ILI9341_Write_Cmd ( CMD_SetPixel );

   //按字节读取字模数据
   //由于前面直接设置了显示窗口，显示数据会自动换行
   for ( byteCount = 0; byteCount < fontLength; byteCount++ ) {
      //一位一位处理要显示的颜色
      for ( bitCount = 0; bitCount < 8; bitCount++ ) {
            if ( Pfont[byteCount] & (0x80>>bitCount) )
               ILI9341_Write_Data ( CurrentTextColor );
            else
               ILI9341_Write_Data ( CurrentBackColor );
```

(下页继续)

(续上页)

```
        }
    }
}
```

这个函数与前文中的串口打印字模到上位机的那个函数原理是一样的，只是这个函数要使用液晶显示，且字模数据并不是一个个独立的数组，而是所有字符的字模都放到同一个数组里，所以显示时，要根据字符编码计算字模数据的偏移，并把串口打印的处理改成像素点显示。该函数的说明如下：

(1) 输入参数

这个字符显示函数有 usX、usY 及 cChar 参数。其中 usX 和 usY 分别表示字符显示位置的 (X,Y) 坐标；而输入参数 cChar 是要显示的英文字符，如字符’A’，字符’空格’等。

(2) 根据字符计算字模的数组偏移

在显示前，首先要提取出字符相应的字模数据。由于 ASCII 码中的编码 0~31(NULL~US 符号) 是不存在图形表示的，为节省空间，字模表中只包含图形符号相关的数据，例如，对于 Font8x16 的字模表，每个字符的字模数据长度 =8*16/8=16 字节，那么 0~15 字节表示的是’空格’字模，16~31 字节表示的是’!’号字模，32~47 字节表示的是’“’号字模。

因此，显示函数中通过 cChar 获知要显示的字符后，使用 ucRelativePositon 变量存储 cChar 减去字符’空格’的 ASCII 码值，即求出图形 ASCII 码的偏移值；然后使用 fontLength 变量存储根据当前选择的字体宽度与高度计算出的单个字模数据长度；最后根据它在字模数组表中的偏移值（ucRelativePositon 与 fontLength 的乘积）求出该字模的地址指针，存储在指针变量 Pfont 中，后面就可以直接使用该指针获取字模数据了，从 Pfont 至 (Pfont+fontLength-1) 范围内的都是输入字符 cChar 的字模数据。

(3) 设置显示窗口并发送显示命令

计算出字模数据的指针后，可以准备开始显示，函数中使用 ILI9341_OpenWindow 函数根据输入的显示坐标及字模的宽高设置一个字符的显示窗口，并使用 ILI9341_Write_Cmd 函数发送设置像素点的命令（CMD_SetPixel），有了这两个操作

后，下面使用的 ILI9341_Write_Data 函数发送的像素点数据将会一行一行地显示到窗口中（沿 X 方向，到达 X 结尾后沿 Y 方向显示下一行），见图 *27_6*。

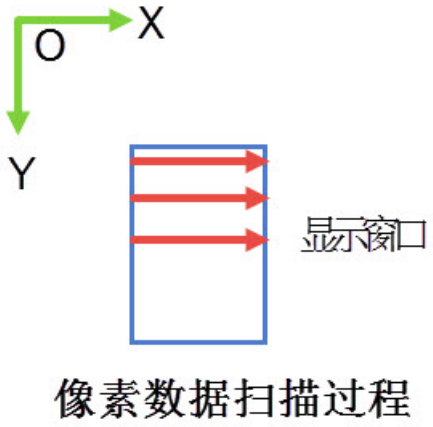


图 27-6 设置显示窗口后的像素数据扫描过程

(4) 行循环与列循环

由于根据字模大小设置了显示窗口，使用 ILI9341_Write_Data 函数发送像素数据时到达单行的结尾它会自动换行，所以在发送数据时不需要再考虑换行。代码中直接使用两层循环处理字模数据，其中外层 for 循环用于遍历字模的字节数据，一个字节一个字节地读取，而字节数据的处理则交给内层 for 循环，当外层 for 循环遍历完 fontLength 个字节表示处理完一个字符的字模，即显示完一个字符；内层 for 循环用于遍历字模单个字节数据的每个数据位，数据位为 1 时就发送一个点的字体颜色（CurrentTextColor），数据位为 0 时就发送背景颜色（CurrentBackColor）。

经过 ILI9341_DispChar_EN 函数的处理，可显示一个英文字符，要显示字符串时，重复调用本函数即可。

26.3.2.2.5 显示字符串

对 ILI9341_DispChar_EN 函数进行封装，我们可以得到 ASCII 字符的字符串显示函数，见*代码清单 27_7*。

列表 7: 代码清单 27-7 字符串显示函数

```
/**
* @brief  在 ILI9341 显示器上显示英文字符串
* @param  usX ：在特定扫描方向下字符的起始 X 坐标
* @param  usY ：在特定扫描方向下字符的起始 Y 坐标
* @param  pStr ：要显示的英文字符串的首地址
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
* @retval 无
*/
void ILI9341_DispString_EN (  uint16_t usX ,uint16_t usY,  char * pStr )
{
   while ( * pStr != '\0' ) {
if ( ( usX - ILI9341_DispWindow_X_Star + LCD_Currentfonts->Width )> LCD_X_
↪LENGTH) {
            usX = ILI9341_DispWindow_X_Star;
            usY += LCD_Currentfonts->Height;
         }

if ( ( usY -ILI9341_DispWindow_Y_Star + LCD_Currentfonts->Height )>LCD_Y_
↪LENGTH ) {
            usX = ILI9341_DispWindow_X_Star;
            usY = ILI9341_DispWindow_Y_Star;
         }

         ILI9341_DispChar_EN ( usX, usY, * pStr);

         pStr ++;
```

(下页继续)

(续上页)

```
            usX += LCD_Currentfonts->Width;
    }
}
```

本函数中的输入参数 pStr 为指向要显示的字符串的指针，在函数的内部利用 while 循环把字符串中的字符一个个地利用 ILI9341_DispChar_EN 函数显示到液晶屏上，当遇到字符串结束符’0’时完成显示，退出 while 循环，结束函数。在 while 循环的开头，有两个 if 判断操作，它们分别用于判断显示字符时的 X 及 Y 坐标是否超出屏幕的边沿，若超出了则换到下一行，使用这个函数，可以很方便地利用“ILI9341_DispString_EN (10，20,”test”)”这样的格式在液晶屏上直接显示一串字符。

26.3.2.2.6 使用宏计算 Y 坐标

使用 ILI9341_DispString_EN 函数显示时，需要注意字 Y 方向字符覆盖的问题，例如在 (10,20) 坐标处显示了一行字体高度为 16 像素的字符串，如果再显示另一串字符时指定的坐标为 (10,25)，那么由于高度预留不足，会出现字符覆盖的现象，因此调用时需要小心计算 Y 方向的坐标。为了简化操作，代码中定义了一个宏 LINE 及函数 ILI9341_DispStringLine_EN 来显示字符串，见*代码清单 27_8*。

列表 8: 代码清单 27-8 使用宏计算 Y 坐标

```
#define LINE(x) ((x) * (((sFONT *)LCD_GetFont())->Height))

/**
* @brief  在 ILI9341 显示器上显示英文字符串
* @param  line ：在特定扫描方向下字符串的起始 Y 坐标
*   本参数可使用宏 LINE(0)、LINE(1) 等方式指定文字坐标，
*   宏 LINE(x) 会根据当前选择的字体来计算 Y 坐标值。
*   显示中文且使用 LINE 宏时，需要把英文字体设置成 Font8x16
* @param  pStr ：要显示的英文字符串的首地址
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
```

(下页继续)

(续上页)

```
* @retval 无
*/
void ILI9341_DispStringLine_EN (  uint16_t line,  char * pStr )
{
   uint16_t usX = 0;

   while ( * pStr != '\0' ) {
if(( usX - ILI9341_DispWindow_X_Star + LCD_Currentfonts->Width ) > LCD_X_
↪LENGTH ) {
            usX = ILI9341_DispWindow_X_Star;
            line += LCD_Currentfonts->Height;
      }

if((line - ILI9341_DispWindow_Y_Star + LCD_Currentfonts->Height )> LCD_Y_
↪LENGTH ) {
            usX = ILI9341_DispWindow_X_Star;
            line = ILI9341_DispWindow_Y_Star;
      }

      ILI9341_DispChar_EN ( usX, line, * pStr);

      pStr ++;

      usX += LCD_Currentfonts->Width;
   }
}
```

本函数主体与 ILI9341_DispString_EN 差异不大，主要是删减了 X 方向的坐标，另外使用输入参数 line 来指定 Y 方向的坐标，调用本函数时，一般配合上面的 LINE 宏来使用，该宏会根据当前选择的字体高度来计算字符 Y 方向的间隔，如当前字体为 8x16 时，字体高度为 16 像素，所以 LINE(1) 会返回数值 16，LINE(3) 会返回数值 48。利用它配合本函数可以使用 “ILI9341_DispStringLine_EN (LINE(1),” test”); ” 的形式来使字符串显示在第 1 行，即 Y 方向为 16 像素处，当然，使用前面

的函数“ILI9341_DispString_EN (10,LINE(1),” test”); ”同样也是可以的，并且可指定 X 坐标。

26.3.2.2.7 清除屏幕字符

在实际应用中，还经常需要把当前屏幕显示的内容清除掉，这可以使用代码清单 27_9 中的函数。

列表 9: 代码清单 27-9 清除屏幕字符

```
/**
* @brief  对 ILI9341 显示器的某一窗口以某种颜色进行清屏
* @param  usX ：在特定扫描方向下窗口的起点 X 坐标
* @param  usY ：在特定扫描方向下窗口的起点 Y 坐标
* @param  usWidth ：窗口的宽度
* @param  usHeight ：窗口的高度
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
* @retval 无
*/
void ILI9341_Clear ( uint16_t usX, uint16_t usY,
{
   ILI9341_OpenWindow ( usX, usY, usWidth, usHeight );

   ILI9341_FillColor ( usWidth * usHeight, CurrentBackColor );

}

/**
* @brief  清除某行文字
* @param  Line: 指定要删除的行
*   本参数可使用宏 LINE(0)、LINE(1) 等方式指定要删除的行，
*   宏 LINE(x) 会根据当前选择的字体来计算 Y 坐标值，并删除当前字体高度的第 x 行。
* @retval None
*/
void LCD_ClearLine(uint16_t Line)
```

(下页继续)

(续上页)

```
{/* 清屏，显示背景颜色 */
   ILI9341_Clear(0,Line,LCD_X_LENGTH,((sFONT *)LCD_GetFont())->Height);
}
```

代码中的 ILI9341_Clear 函数可以直接清除一个指定的矩形，它会把该矩形显示成当前设置的背景颜色 CurrentBackColor，实现清除图像的效果，而 LCD_ClearLine 函数对它进行了封装，使用 LINE 宏配合，可以比较方便地清除单行字符串，如调用 LCD_ClearLine(LINE(1)) 可以清除第 1 行的字符串。

26.3.2.2.8 显示 ASCII 码示例

下面我们再来看 main 文件是如何利用这些函数显示 ASCII 码字符的，见*代码清单 27_10*。

列表 10: 代码清单 27-10 显示 ASCII 码的 main 函数

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main ( void )
{
   ILI9341_Init ();          //LCD 初始化

   USART_Config();

   printf("\r\n ********** 液晶屏英文显示程序 ********** \r\n");
   printf("\r\n 本程序不支持中文，显示中文的程序请学习下一章 \r\n");

   //其中 0、3、5、6 模式适合从左至右显示文字，
   //不推荐使用其它模式显示文字 其它模式显示文字会有镜像效果
```

(下页继续)

(续上页)

```
    //其中 6 模式为大部分液晶例程的默认显示方向
    ILI9341_GramScan ( 6 );
    while ( 1 ) {
        LCD_Test();
    }
}
```

main 函数中主要是对液晶屏初始化，初始化完成后就能够显示 ASCII 码字符了，无需利用 SPI-FLASH 及 SD 卡。在 while 循环中调用的 LCD_Test 函数包含了显示字符的函数调用示例，见*代码清单 27_11*。

列表 11: 代码清单 27-11 LCD_Test 函数中的 ASCII 码显示示例

```
/* 用于测试各种液晶的函数 */
void LCD_Test(void)
{
    /* 演示显示变量 */
    static uint8_t testCNT = 0;
    char dispBuff[100];

    testCNT++;

    LCD_SetFont(&Font8x16);
    LCD_SetColors(RED,BLACK);

    ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */
    /******** 显示字符串示例 *******/
    ILI9341_DispStringLine_EN(LINE(0),"BH 3.2 inch LCD para:");
    ILI9341_DispStringLine_EN(LINE(1),"Image resolution:240x320 px");
    ILI9341_DispStringLine_EN(LINE(2),"ILI9341 LCD driver");
    ILI9341_DispStringLine_EN(LINE(3),"XPT2046 Touch Pad driver");
```

(下页继续)

(续上页)

```
    /******** 显示变量示例 *******/
    LCD_SetFont(&Font16x24);
    LCD_SetTextColor(GREEN);

    /* 使用 cHAL 库把变量转化成字符串 */
    sprintf(dispBuff,"Count : %d ",testCNT);
    LCD_ClearLine(LINE(4)); /* 清除单行文字 */

    /* 然后显示该字符串即可，其它变量也是这样处理 */
    ILI9341_DispStringLine_EN(LINE(4),dispBuff);
    /* 以下省略其它液晶演示示例 */
}
```

这段代码包含了使用字符串显示函数显示常量字符和变量的示例。显示常量字符串时，直接使用双引号括起要显示的字符串即可，根据 C 语言的语法，这些字符串会被转化成常量数组，数组内存储对应字符的 ASCII 码，然后存储到 STM32 的 FLASH 空间，函数调用时通过指针来找到对应的 ASCII 码，液晶显示函数使用前面分析过的流程，转换成液晶显示输出。

在很多场合下，我们可能需要使用液晶屏显示代码中变量的内容，这时很多用户就不知道该如何解决了，上面的 LCD_Test 函数结尾处演示了如何处理。它主要是使用一个 C 语言 HAL 库里的函数 sprintf，把变量转化成 ASCII 码字符串，转化后的字符串存储到一个数组中，然后我们再利用液晶显示字符串的函数显示该数组的内容即可。spritnf 函数的用法与 printf 函数类似，使用它时需要包含头文件 string.h。

26.3.3 显示 GB2312 编码的字符

显示 ASCII 编码比较简单，由于字库文件小，甚至都不需要使用外部的存储器，而显示汉字时，由于我们的字库是存储到外部存储器上的，这涉及到额外的获取字模数据的操作。

我们分别制作了两个工程来演示如何显示汉字，以下部分的内容请打开“液晶显示中英文（字库在外部 FLASH）”和“液晶显示中英文（字库在 SD 卡）”工程阅读理解。这两个工程使用的字

库文件内容相同，只是字库存储的位置不一样，工程中我们把获取字库数据相关的函数代码写在“fonts.c”及“fonts.h”文件中，字符显示的函数仍存储在 LCD 驱动文件“bsp_ili9341_lcd.c”及“bsp_ili9341_lcd.h”中。

26.3.3.1 编程要点

(1) 获取字模数据；

(2) 根据字模格式，编写液晶显示函数；

(3) 编写测试程序，控制液晶汉字。

26.3.3.2 代码分析

26.3.3.2.1 显示汉字字符

由于我们的 GB2312 字库文件与 ASCII 字库文件是使用同一种方式生成的，但字符的编码不同导致字模偏移地址计算有区别，且字模数据存储的位置不同，所以为了显示汉字，需要另外编写一个字符显示函数，它利用前文生成的《GB2312_H1616.FON》字库显示 GB2312 编码里的字符，见*代码清单 27_12*。

列表 12: 代码清单 27-12 显示 GB2312 编码字符的函数
(bsp_ili9341_ldc.c 文件)

```
#define      WIDTH_CH_CHAR                  16      //中文字符宽度
#define      HEIGHT_CH_CHAR                 16      //中文字符高度

/**
* @brief  在 ILI9341 显示器上显示一个中文字符
* @param  usX ：在特定扫描方向下字符的起始 X 坐标
* @param  usY ：在特定扫描方向下字符的起始 Y 坐标
* @param  usChar ：要显示的中文字符（国标码）
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
```

(下页继续)

(续上页)

```
* @retval 无
*/
void ILI9341_DispChar_CH ( uint16_t usX, uint16_t usY, uint16_t usChar )
{
   uint8_t rowCount, bitCount;
   uint8_t ucBuffer [ WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8 ];
   uint16_t usTemp;

   //设置显示窗口
   ILI9341_OpenWindow ( usX, usY, WIDTH_CH_CHAR, HEIGHT_CH_CHAR );

   ILI9341_Write_Cmd ( CMD_SetPixel );

   //取字模数据
   GetGBKCode ( ucBuffer, usChar );

   for ( rowCount = 0; rowCount < HEIGHT_CH_CHAR; rowCount++ ) {
      /* 取出两个字节的数据，在 lcd 上即是一个汉字的一行 */
      usTemp = ucBuffer [ rowCount * 2 ];
      usTemp = ( usTemp << 8 );
      usTemp |= ucBuffer [ rowCount * 2 + 1 ];

      for ( bitCount = 0; bitCount < WIDTH_CH_CHAR; bitCount ++ ) {
            if ( usTemp & ( 0x8000 >> bitCount ) )  //高位在前
               ILI9341_Write_Data ( CurrentTextColor );
            else
               ILI9341_Write_Data ( CurrentBackColor );
      }
   }
}
```

这个 GB2312 码的显示函数与 ASCII 码的显示函数是很类似的，它的输入参数有 usX，usY 及

usChar。其中 usX 和 usY 用于设定字符的显示位置，usChar 是字符的编码，这是一个 16 位的变量，因为 GB2312 编码中每个字符是 2 个字节的。函数的执行流程介绍如下：

(1) 使用 ILI9341_OpenWindow 和 ILI9341_Write_Cmd 来设置显示窗口并发送显示像素命令。

(2) 使用 GetGBKCode 函数获取字模数据，向该函数输入 usChar 参数 (字符的编码)，它会从外部 SPI-FLASH 芯片或 SD 卡中读取该字符的字模数据，读取得的数据被存储到数组 ucBuffer 中。关于 GetGBKCode 函数我们在后面详细讲解。

(3) 遍历像素点。这个代码在遍历时还使用了 usTemp 变量用来缓存一行的字模数据 (本字模一行有 2 个字节)，然后一位一位地判断这些数据，数据位为 1 的时候，像素点就显示字体颜色，否则显示背景颜色。原理是跟 ASCII 字符显示一样的。

26.3.3.2.2 显示中英文字符串

类似地，我们希望希望汉字也能直接以字符串的形式来调用函数显示，而且最好是中英文字符可以混在一个字符串里。为此，我们编写了 LCD_DisplayStringLine_EN_CH 函数，*代码清单 27_13*。

列表 13: 代码清单 27-13 显示中英文的字符串

```
/**
* @brief  在 ILI9341 显示器上显示中英文字符串
* @param  usX ：在特定扫描方向下字符的起始 X 坐标
* @param  usY ：在特定扫描方向下字符的起始 Y 坐标
* @param  pStr ：要显示的字符串的首地址
* @note 可使用 LCD_SetBackColor、LCD_SetTextColor、LCD_SetColors 函数设置颜色
* @retval 无
*/
void ILI9341_DispString_EN_CH (uint16_t usX , uint16_t usY, char * pStr )
{
   uint16_t usCh;

   while ( * pStr != '\0' ) {
        if ( * pStr <= 126 ) {            //英文字符
```

(下页继续)

(续上页)

```
if(( usX - ILI9341_DispWindow_X_Star + LCD_Currentfonts->Width ) > LCD_X_
↪LENGTH ) {
                usX = ILI9341_DispWindow_X_Star;
                usY += LCD_Currentfonts->Height;
             }

if(( usY - ILI9341_DispWindow_Y_Star + LCD_Currentfonts->Height )> LCD_Y_
↪LENGTH ) {
                usX = ILI9341_DispWindow_X_Star;
                usY = ILI9341_DispWindow_Y_Star;
             }

             ILI9341_DispChar_EN ( usX, usY, * pStr );

             usX +=  LCD_Currentfonts->Width;
             pStr ++;
       }

       else {                                  //汉字字符
if ( ( usX - ILI9341_DispWindow_X_Star + WIDTH_CH_CHAR ) > LCD_X_LENGTH ) {
                usX = ILI9341_DispWindow_X_Star;
                usY += HEIGHT_CH_CHAR;
             }

if ( ( usY - ILI9341_DispWindow_Y_Star + HEIGHT_CH_CHAR ) > LCD_Y_LENGTH ) {
                usX = ILI9341_DispWindow_X_Star;
                usY = ILI9341_DispWindow_Y_Star;
             }

             usCh = * ( uint16_t * ) pStr;

             usCh = ( usCh << 8 ) + ( usCh >> 8 );
```

(下页继续)

(续上页)

```
            ILI9341_DispChar_CH ( usX, usY, usCh );

            usX += WIDTH_CH_CHAR;
            pStr += 2;            //一个汉字两个字节
        }
    }
}
```

这个函数根据字符串中的编码值，判断它是 ASCII 码还是国标码中的字符，然后作不同处理。英文部分与前方中的英文字符串显示函数是一样的，中文部分也很类似，需要注意的是中文字符每个占 2 个字节，而且由于 STM32 芯片的数据是小端格式存储的，国标码是大端格式存储的，所以函数中对输入参数 pStr 指针获取的编码 usCh 交换了字节顺序，再输入到单个字符的显示函数 LCD_DispChar_CH 中。

26.3.3.2.3 获取 SPI-FLASH 中的字模数据

前面提到的 GetGBKCode 函数用于获取汉字字模数据，它根据字库文件的存储位置，有 SPI-FLASH 和 SD 卡两个版本，我们先来分析比较简单的 SPI-FLASH 版本，*代码清单 27_14*。该函数定义在“液晶显示中英文（字库在外部 FLASH）”工程的“fonts.c”和“fonts.h”文件中。

列表 14: 代码清单 27-14 从 SPI-FLASH 获取字模数据 (“液晶显示中英文（字库在外部 FLASH）“工程)

```
/*************fonts.h 文件中的定义 *********************************/

/* 使用 FLASH 字模 */
/* 中文字库存储在 FLASH 的起始地址 */
/*FLASH*/
#define GBKCODE_START_ADDRESS   387*4096
```

(下页继续)

(续上页)

```
/* 获取字库的函数 */
//定义获取中文字符字模数组的函数名，ucBuffer 为存放字模数组名，usChar 为中文字符（国标码）

#define GetGBKCode( ucBuffer, usChar )  GetGBKCode_from_EXFlash( ucBuffer, usChar )
int GetGBKCode_from_EXFlash( uint8_t * pBuffer, uint16_t c);

/*****************************************************************/

/************fonts.c 文件中的字义 ************************************/

/* 使用 FLASH 字模 */
//字模 GB2312_H1616 配套的函数

/**
* @brief  获取 FLASH 中文显示字库数据
* @param  pBuffer: 存储字库矩阵的缓冲区
* @param  c :  要获取的文字
* @retval None.
*/
int GetGBKCode_from_EXFlash( uint8_t * pBuffer, uint16_t c)
{
   unsigned char High8bit,Low8bit;
   unsigned int pos;

   static uint8_t everRead=0;

   /* 第一次使用，初始化 FLASH*/
   if (everRead == 0) {
      SPI_FLASH_Init();
      everRead = 1;
```

(下页继续)

(续上页)

```
    }

    High8bit= c >> 8;      /* 取高 8 位数据 */
    Low8bit= c & 0x00FF;   /* 取低 8 位数据 */

    /*GB2312 公式 */
    pos = ((High8bit-0xa1)*94+Low8bit-0xa1)*WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8;
    //读取字库数据
    SPI_FLASH_BufferRead(pBuffer,GBKCODE_START_ADDRESS+pos, WIDTH_CH_
↪CHAR*HEIGHT_CH_CHAR/8);
    return 0;
}
```

这个 GetGBKCode 实质上是一个宏，当使用 SPI-FLASH 作为字库数据源时，它等效于 GetGBKCode_from_EXFlash 函数，它的执行过程如下：

(1) 初始化 SPI 外设，以确保后面使用 SPI 读取 FLASH 内容时 SPI 已正常工作，并且初始化后做一个标记，以后再读取字模数据的时候就不需要再次初始化 SPI 了；

(2) 取出要显示字符的 GB2312 编码的高位字节和低位字节，以便后面用于计算字符的字模地址偏移；

(3) 根据字符的编码及字模的大小导出的寻址公式，计算当前要显示字模数据在字库中的地址偏移；

(4) 利用 SPI_FLASH_BufferRead 函数，从 SPI-FLASH 中读取该字模的数据，输入参数中的 GBKCODE_START_ADDRESS 是在代码头部定义的一个宏，它是字库文件存储在 SPI-FLASH 芯片的基地址，该基地址加上字模在字库中的地址偏移，即可求出字模在 SPI-FLASH 中存储的实际位置。这个基地址具体数值是在我们烧录 FLASH 字库时决定的，程序中定义的是实验板出厂时默认烧录的位置。

(5) 获取到的字模数据存储到 pBuffer 指针指向的存储空间，显示汉字的函数直接利用它来显示字符。

26.3.3.2.4 获取 SD 卡中的字模数据

类似地，从 SD 卡中获取字模数据时，使用 GetGBKCode_from_sd 函数，见*代码清单 27_15*。该函数定义在“液晶显示中英文（字库在 SD 卡）”工程的“fonts.c”和“fonts.h”文件中。

列表 15: 代码清单 27-15 从 SD 卡中获取字模数据 (“液晶显示中英文（字库在 SD 卡）”工程)

```
/*************fonts.h 文件中的定义 *********************************/

/* 使用 SD 字模 */

/*SD 卡字模路径 */
#define GBKCODE_FILE_NAME      "0:/srcdata/GB2312_H1616.FON"

/* 获取字库的函数 */
//定义中文字符字模数组的函数名，ucBuffer 为存放字模数组名，usChar 为中文字符（国标码）

#define GetGBKCode( ucBuffer, usChar )  GetGBKCode_from_sd( ucBuffer,␣
↪usChar )
int GetGBKCode_from_sd ( uint8_t * pBuffer, uint16_t c);

/*****************************************************************/

/*************fonts.c 文件中的字义 *********************************/

/* 使用 SD 字模 */

static FIL fnew;                                  /* 文件句柄 */
static FATFS fs;                                  /* 文件系统句柄 */
static FRESULT res_sd;
static UINT br;                                 /* 文件 R/W 计数 */
```

(下页继续)

(续上页)

```
/**
* @brief  获取 SD 卡中文显示字库数据
* @param  pBuffer: 存储字库矩阵的缓冲区
* @param  c :  要获取的文字
* @retval None.
*/
int GetGBKCode_from_sd ( uint8_t * pBuffer, uint16_t c)
{
   unsigned char High8bit,Low8bit;
   unsigned int pos;

   static uint8_t everRead = 0;

   High8bit= c >> 8;      /* 取高 8 位数据 */
   Low8bit= c & 0x00FF;   /* 取低 8 位数据 */

   pos = ((High8bit-0xa1)*94+Low8bit-0xa1)*WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8;

   /* 第一次使用，挂载文件系统，初始化 sd*/
   if (everRead == 0) {
      res_sd = f_mount(&fs,"0:",1);
      everRead = 1;

   }

res_sd = f_open(&fnew , GBKCODE_FILE_NAME, FA_OPEN_EXISTING | FA_READ);

   if ( res_sd == FR_OK ) {
      f_lseek (&fnew, pos);    //指针偏移

      //16*16 大小的汉字 其字模 占用 16*16/8 个字节
res_sd = f_read( &fnew, pBuffer, WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8, &br );
```

(下页继续)

(续上页)

```
      f_close(&fnew);

      return 0;
   } else
      return -1;
}
```

当字库的数据源在 SD 卡时，GetGBKCode 宏指向的是这个 GetGBKCode_from_sd 函数。由于字库是使用 SD 卡的文件系统存储的，从 SD 卡中获取字模数据实质上是直接读取字库文件，利用 f_lseek 函数偏移文件的读取指针，使它能够读取特定字符的字模数据。

由于使用文件系统的方式读取数据比较慢，而 SD 卡大多数都会使用文件系统，所以我们一般使用 SPI-FLASH 直接存储字库 (不带文件系统地使用)，市场上有一些厂商直接生产专用的字库芯片，可以直接使用，省去自己制作字库的麻烦。

26.3.3.2.5 显示 GB2312 字符示例

下面我们再来看 main 文件是如何利用这些函数显示 GB2312 的字符，由于我们用 GetGBKCode 宏屏蔽了差异，所以在上层使用字符串函数时，不需要针对不同的字库来源写不同的代码，见*代码清单 27_16*。

列表 16: 代码清单 27-16 main 函数

```
int main(void)
{
   ILI9341_Init ();          //LCD 初始化

   /* USART config */
   USART_Config();

   printf("\r\n ****** 液晶屏中文显示程序（字模文件在 SD 卡）**** \r\n");
```

(下页继续)

(续上页)

```
    printf("\r\n 实验前请阅读工程中的 readme.txt 文件说明，存储字模数据到 SPI-FLASH
或 SD 卡\r\n");

    //其中 0、3、5、6 模式适合从左至右显示文字，
    //不推荐使用其它模式显示文字 其它模式显示文字会有镜像效果
    //其中 6 模式为大部分液晶例程的默认显示方向
    ILI9341_GramScan ( 6 );

    while ( 1 ) {
          LCD_Test();
    }
}
```

main 文件中的初始化流程与普通的液晶初始化没有区别，这里也不需要初始化 SPI 或 SDIO，因为我们在获取字库的函数中包含了相应的初始化流程。在 while 循环里调用的 LCD_Test 包含了显示 GB2312 字符串的示例，见*代码清单 27_17*。

列表 17: 代码清单 27-17 显示 GB2312 字符示例

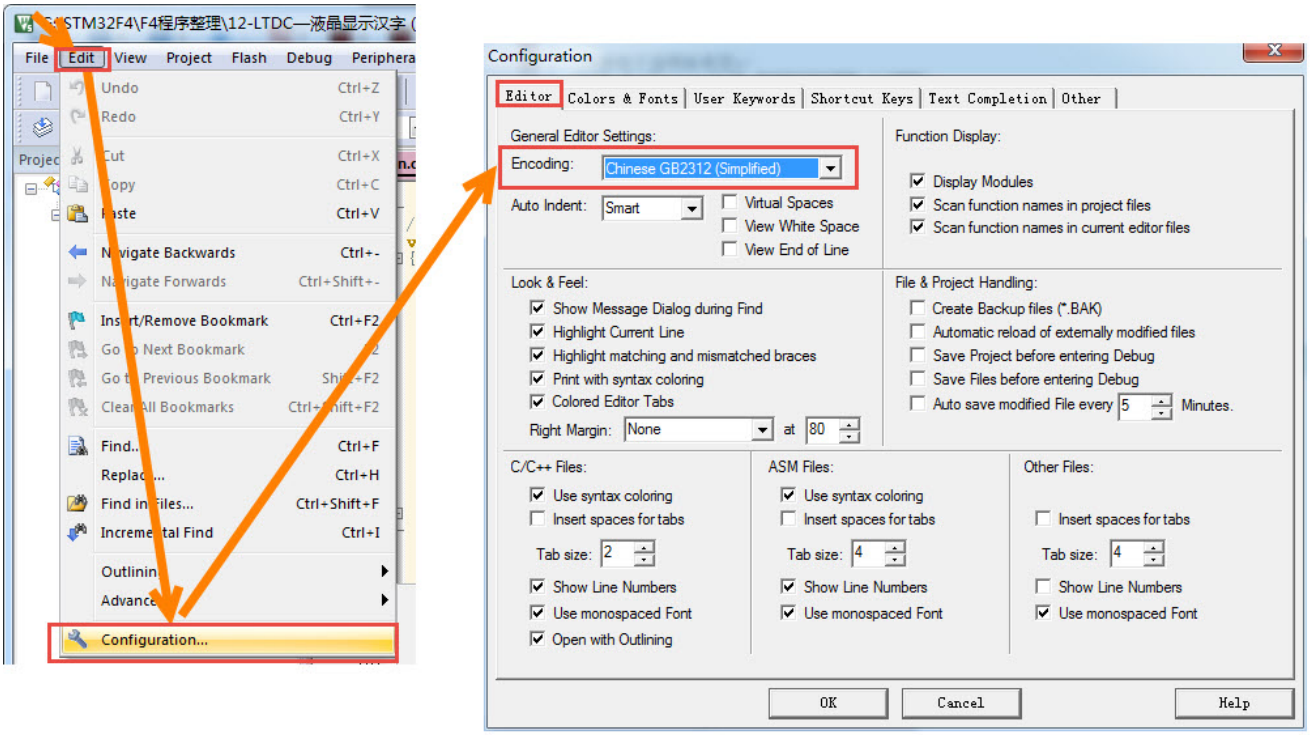
```
/* 用于测试各种液晶的函数 */
void LCD_Test(void)
{
    /* 演示显示变量 */
    static uint8_t testCNT = 0;
    char dispBuff[100];

    testCNT++;

    LCD_SetFont(&Font8x16);
    LCD_SetColors(RED,BLACK);

    ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */
```

(下页继续)

(续上页)

```
/******** 显示字符串示例 *******/
ILI9341_DispStringLine_EN_CH(LINE(0),"野火 3.2 寸 LCD 参数: ");
ILI9341_DispStringLine_EN_CH(LINE(1),"分辨率: 240x320 px");
ILI9341_DispStringLine_EN_CH(LINE(2),"ILI9341 液晶驱动");
ILI9341_DispStringLine_EN_CH(LINE(3),"XPT2046 触摸屏驱动");

/******** 显示变量示例 *******/
LCD_SetTextColor(GREEN);

/* 使用 cHAL 库把变量转化成字符串 */
sprintf(dispBuff,"显示变量:  %d ",testCNT);
LCD_ClearLine(LINE(5)); /* 清除单行文字 */

/* 然后显示该字符串即可，其它变量也是这样处理 */
ILI9341_DispStringLine_EN_CH(LINE(5),dispBuff);

/****** 其它液晶演示示例省略 ******/
```

在调用字符串显示函数的时候，我们也是直接使用双引号括起要显示的中文字符即可，为什么这样就能正常显示呢？我们的字符串显示函数需要的输入参数是字符的 GB2312 编码，编译器会自动转化这些中文字符成相应的 GB2312 编码吗？为什么编译器不把它转化成 UTF-8 编码？这跟我们的开发环境配置有关，在 MDK 软件中，可在“Edit->Configuration->Editor->Encoding”选项设定编码，见图 *27_7*。



编译环境会把文件中的字符串转换成这里配置的编码，然后存储到 STM32 的程序空间中，所以这里的设定要与您的字库编码格式一样。如果您的实验板显示的时候出现乱码，请确保以下所有环节都正常：

- SPI-FLASH 或 SD 卡中是否有字库文件？
- 文件存储的位置或路径是否与程序的配置一致？
- 开发环境中的字符编码选项是否与字库的编码一致？

26.3.4 显示任意大小的字符

前文中无论是 ASCII 字符还是 GB2312 的字符，都只能显示字库中设定的字体大小，例如，我们想显示一些像素大小为 48x48 的字符，那我们又得制作相应的字库，非常麻烦。为此我们编写了一些函数，简便地实现显示任意大小字符的目的。本小节的内容请打开“液晶显示中英文（任意大小）”工程来配合阅读。

26.3.4.1 编程要点

(1) 编写缩放字模数据的函数；

(2) 编写利用缩放字模的结果进行字符显示的函数；

(3) 编写测试程序，控制显示不同大小的字符。

26.3.4.2 代码分析

26.3.4.2.1 缩放字模数据

显示任意大小字符的功能，其核心是缩放字模，通过 LCD_zoomChar 函数对原始字模数据进行缩放，见代码清单 27-1。

列表 18: 代码清单 27-18 缩放字模数据

```
/************************ 缩放字体 ****************************/
#define ZOOMMAXBUFF 16384
uint8_t zoomBuff[ZOOMMAXBUFF] = {0};  //用于缩放的缓存，最大支持到 128*128
uint8_t zoomTempBuff[1024] = {0};

/**
* @brief  缩放字模，缩放后的字模由 1 个像素点由 8 个数据位来表示
                    0x01 表示笔迹，0x00 表示空白区
* @param  in_width ：原始字符宽度
* @param  in_heig ：原始字符高度
* @param  out_width ：缩放后的字符宽度
* @param  out_heig：缩放后的字符高度
* @param  in_ptr ：字库输入指针 注意：1pixel 1bit
* @param  out_ptr ：缩放后的字符输出指针 注意：1pixel 8bit
*    out_ptr 实际上没有正常输出，改成了直接输出到全局指针 zoomBuff 中
* @param  en_cn ：0 为英文，1 为中文
* @retval 无
```

(下页继续)

(续上页)

```
*/
void ILI9341_zoomChar(uint16_t in_width,  //原始字符宽度
                      uint16_t in_heig,   //原始字符高度
                      uint16_t out_width, //缩放后的字符宽度
                      uint16_t out_heig,  //缩放后的字符高度
                      uint8_t *in_ptr,  //字库输入指针  注意: 1pixel 1bit
                    uint8_t *out_ptr, //缩放后的字符输出指针 注意: 1pixel 8bit
                      uint8_t en_cn)     //0 为英文, 1 为中文
{
   uint8_t *pts,*ots;
   //根据源字模及目标字模大小, 设定运算比例因子, 左移 16 是为了把浮点运算转成定点运算

   unsigned int xrIntFloat_16=(in_width<<16)/out_width+1;
   unsigned int yrIntFloat_16=(in_heig<<16)/out_heig+1;

   unsigned int srcy_16=0;
   unsigned int y,x;
   uint8_t *pSrcLine;

   uint16_t byteCount,bitCount;

   //检查参数是否合法
   if (in_width >= 32) return;                 //源字库不允许超过 32 像素
   if (in_width * in_heig == 0) return;
   if (in_width * in_heig >= 1024 ) return;           //限制输入最大 32*32

   if (out_width * out_heig == 0) return;
if (out_width * out_heig >= ZOOMMAXBUFF ) return; //限制最大缩放 128*128
   pts = (uint8_t*)&zoomTempBuff;

   //为方便运算, 字库的数据由 1 pixel/1bit 映射到 1pixel/8bit
   //0x01 表示笔迹, 0x00 表示空白区
```

(下页继续)

(续上页)

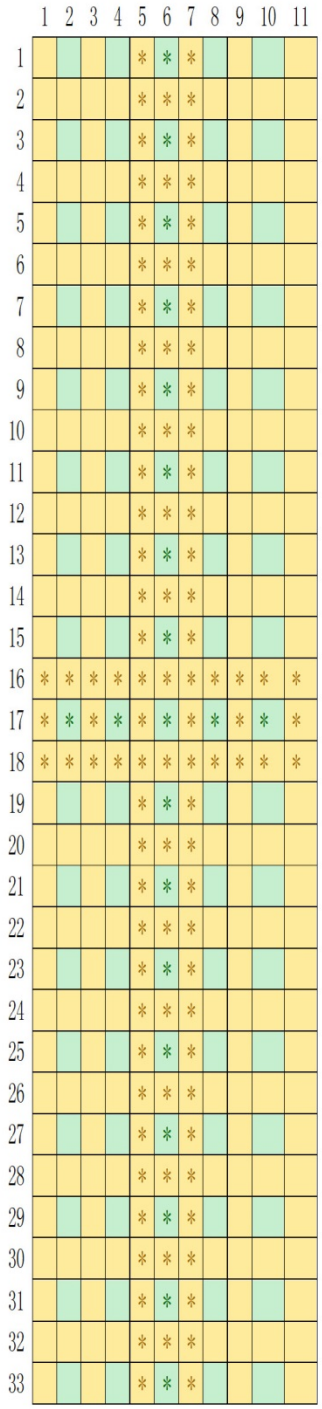
```
if (en_cn == 0x00) { //英文
      //英文和中文字库上下边界不对，可在此处调整。需要注意 tempBuff 防止溢出
      for (byteCount=0; byteCount<in_heig*in_width/8; byteCount++) {
         for (bitCount=0; bitCount<8; bitCount++) {
            //把源字模数据由位映射到字节
            //in_ptr 里 bitX 为 1，则 pts 里整个字节值为 1
            //in_ptr 里 bitX 为 0，则 pts 里整个字节值为 0
            *pts++ = (in_ptr[byteCount] & (0x80>>bitCount))?1:0;
         }
      }
} else { //中文
      for (byteCount=0; byteCount<in_heig*in_width/8; byteCount++) {
         for (bitCount=0; bitCount<8; bitCount++) {
            //把源字模数据由位映射到字节
            //in_ptr 里 bitX 为 1，则 pts 里整个字节值为 1
            //in_ptr 里 bitX 为 0，则 pts 里整个字节值为 0
            *pts++ = (in_ptr[byteCount] & (0x80>>bitCount))?1:0;
         }
      }
}

//zoom 过程
pts = (uint8_t*)&zoomTempBuff;  //映射后的源数据指针
ots = (uint8_t*)&zoomBuff;  //输出数据的指针
for (y=0; y<out_heig; y++) {  /* 行遍历 */
      unsigned int srcx_16=0;
      pSrcLine=pts+in_width*(srcy_16>>16);
      for (x=0; x<out_width; x++) { /* 行内像素遍历 */
         ots[x]=pSrcLine[srcx_16>>16]; //把源字模数据复制到目标指针中
         srcx_16+=xrIntFloat_16;      //按比例偏移源像素点
      }
      srcy_16+=yrIntFloat_16;          //按比例偏移源像素点
```

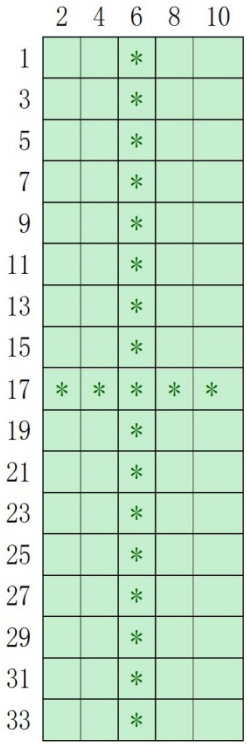
(下页继续)

(续上页)

```
            ots+=out_width;
    }
    /*!!! 缩放后的字模数据直接存储到全局指针 zoomBuff 里了 */
    //out_ptr 没有正确传出，后面调用直接改成了全局变量指针!
        out_ptr = (uint8_t*)&zoomBuff;
    /* 实际中如果使用 out_ptr 不需要下面这一句!!!
        只是因为 out_ptr 没有使用，会导致 warning。强迫症 */
    out_ptr++;
}
```

缩放字模的本质是按照缩放比例，减少或增加矩阵中的像素点，见图 *27_8*，只要把左侧的矩阵隔一行、隔一列地取出像素点，即可得到右侧按比例缩小了的矩阵，而右侧的小矩阵按比例填复制像素点即可得到左侧放大的矩阵，上述函数就是完成了这样的工作。





该函数的说明如下：

(1) 输入参数

函数包含输入参数源字模、缩放后字模的宽度及高度：in_width、inheig、out_width、out_heig。源字模数据指针 in_ptr，缩放后的字符指针 out_ptr 以及用于指示字模是英文还是中文的标志 en_cn。其中 out_ptr 指针实质上没有用到，这个函数缩放后的数据最后直接存储在全局变量 zoomBuff 中了。

(2) 计算缩放比例

根据输入字模与要求的输出字模大小，计算出缩放比例到 xrIntFloat_16 及 yrIntFloat_16 变量中，运算式中的左移 16 位是典型的把浮点型运算转换成定点运算的处理方式。理解的时候可把左移 16 位的运算去掉，把它当成一个自然的数学小数运算即可。

(3) 检查输入参数

由于运算变量及数组的一些限制，函数中要检查输入参数的范围，本函数限制最大输出字模的大小为 128*128 像素，输入字模限制不可以超过 32*32 像素。

(4) 映射字模

输入源的字模都是 1 个数据位表示 1 个像素点的，为方便后面的运算，函数把输入字模转化成 1 个字节 (8 个数据位) 表示 1 个像素点，该字节的值为 0x01 表示笔迹像素，0x00 表示空白像素。把字模数据的 1 个数据位映射为 1 个字节，可以方便后面直接使用指针和数组索引运算。

(5) 缩放字符

缩放字符这部分代码比较难理解，但总的来说它就是利用前面计算得的比例因子，以它为步长复制源字模的数据到目标字模的缓冲区中，具体的抽象运算只能意会了。其中的右移 16 位是把比例因子由定点数转换回原始的数值。如果还是觉得难以理解，可以把函数的宽度及高度输入参数 in_width、inheig、out_width 及 out_heig 都设置成 16，然后代入运算来阅读这段代码。

(6) 缩放结果

经过运算，缩放的结果存储在 zoomBuff 中，它只是存储了一个字模的缩放结果，所以每显示一个字模都需要先调用这个函数更新 zoomBuff 中的字模数据，而且它也是

用 1 个字节表示 1 个像素位的。

26.3.4.2.2 利用缩放的字模数据显示字符

由于缩放后的字模数据格式与我们原来用的字模数据格式不一样，所以我们也要重新编写字符显示函数，见*代码清单 27_19*。

列表 19: 代码清单 27-19 利用缩放的字模显示字符

```
/**
* @brief  利用缩放后的字模显示字符
* @param  Xpos ：字符显示位置 x
* @param  Ypos ：字符显示位置 y
* @param  Font_width ：字符宽度
* @param  Font_Heig: 字符高度
* @param  c: 要显示的字模数据
* @param  DrawModel ：是否反色显示
* @retval 无
*/
void ILI9341_DrawChar_Ex(uint16_t usX, //字符显示位置 x
                         uint16_t usY, //字符显示位置 y
                         uint16_t Font_width, //字符宽度
                         uint16_t Font_Height,  //字符高度
                         uint8_t *c,            //字模数据
                         uint16_t DrawModel)    //是否反色显示
{
   uint32_t index = 0, counter = 0;

   //设置显示窗口
   ILI9341_OpenWindow ( usX, usY, Font_width, Font_Height);

   ILI9341_Write_Cmd ( CMD_SetPixel );
```

(下页继续)

(续上页)

```
    //按字节读取字模数据
    //由于前面直接设置了显示窗口，显示数据会自动换行
    for ( index = 0; index < Font_Height; index++ ) {
        //一位一位处理要显示的颜色
        for ( counter = 0; counter < Font_width; counter++ ) {
            //缩放后的字模数据，以一个字节表示一个像素位
            //整个字节值为 1 表示该像素为笔迹
            //整个字节值为 0 表示该像素为背景
            if ( *c++ == DrawModel )
                ILI9341_Write_Data ( CurrentBackColor );
            else
                ILI9341_Write_Data ( CurrentTextColor );
        }
    }
}
```

注意在这个函数中，它并没有对中英文模区分显示代码，因为本函数的字模是由输入参数 c 指针中获取的，在调用本函数时，需要输入要显示的字模数据指针，而不是字符编码。在其它方面这个函数主体与前面介绍的字符显示函数都很类似，只是它在判断字模数据位的时候，直接用一整个字节来判断，区分显示分支，而且还支持了反色显示模式。

26.3.4.2.3 利用缩放的字模显示字符串

单个字符显示的函数并不包含字模的获取过程，为便于使用，我们把它直接封装成字符串显示函数，见*代码清单 27_20*。

列表 20: 代码清单 27-20 利用缩放的字模显示字符串

```
/**
* @brief  利用缩放后的字模显示字符串
* @param  Xpos ：字符显示位置 x
```

(下页继续)

(续上页)

```
* @param  Ypos ：字符显示位置 y
* @param  Font_width ：字符宽度，英文字符在此基础上/2。注意为偶数
* @param  Font_Heig: 字符高度，注意为偶数
* @param  c ：要显示的字符串
* @param  DrawModel ：是否反色显示
* @retval 无
*/
void ILI9341_DisplayStringEx(uint16_t x,     //字符显示位置 x
                             uint16_t y,         //字符显示位置 y
                             uint16_t Font_width,
                             //要显示的字体宽度，英文字符在此基础上/2。注意为偶数
                        uint16_t Font_Height,  //要显示的字体高度，注意为偶数
                             uint8_t *ptr,           //显示的字符内容
                             uint16_t DrawModel)   //是否反色显示

{
uint16_t Charwidth = Font_width; //默认为 Font_width，英文宽度为中文宽度的一半
   uint8_t *psr;
   uint8_t Ascii;  //英文
   uint16_t usCh;  //中文
   uint8_t ucBuffer [ WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8 ];

   while ( *ptr != '\0') {
         /**** 处理换行 *****/
if ( ( x - ILI9341_DispWindow_X_Star + Charwidth ) > LCD_X_LENGTH ) {
            x = ILI9341_DispWindow_X_Star;
            y += Font_Height;
         }

if ( ( y - ILI9341_DispWindow_Y_Star + Font_Height ) > LCD_Y_LENGTH ) {
            x = ILI9341_DispWindow_X_Star;
            y = ILI9341_DispWindow_Y_Star;
```

(下页继续)

(续上页)

```
        }

        if (*ptr > 0x80) { //如果是中文
            Charwidth = Font_width;
            usCh = * ( uint16_t * ) ptr;
            usCh = ( usCh << 8 ) + ( usCh >> 8 );
            GetGBKCode ( ucBuffer, usCh );  //取字模数据
            //缩放字模数据，源字模为 16*16
            ILI9341_zoomChar(WIDTH_CH_CHAR,HEIGHT_CH_CHAR,Charwidth,
Font_Height,(uint8_t *)&ucBuffer,psr,1);
            //显示单个字符
            ILI9341_DrawChar_Ex(x,y,Charwidth,Font_Height,
(uint8_t*)&zoomBuff,DrawModel);
            x+=Charwidth;
            ptr+=2;
        } else {//英文
            Charwidth = Font_width / 2;
            Ascii = *ptr - 32;
            //使用 16*24 字体缩放字模数据
        ILI9341_zoomChar(16,24,Charwidth,Font_Height,
   (uint8_t *)&Font16x24.table[Ascii * Font16x24.Height*Font16x24.Width/8],
↪psr,0);
            //显示单个字符
ILI9341_DrawChar_Ex(x,y,Charwidth,Font_Height,(uint8_t*)&zoomBuff,
↪DrawModel);
            x+=Charwidth;
            ptr++;
        }
    }
}
```

这个函数包含了从字符编码到源字模获取、字模缩放及单个字符显示的过程，多个这样的过程

组合起来，就实现了简单易用的字符串显示函数。而且可了解到它使用的英文源字模数据是 Font16x24 字体，而中文源字模数据仍是采用 GetGBKCode 函数获取，使得数据源的获取与上层分离，支持从 SPI FLASH 及 SD 卡中获取数据源。

26.3.4.2.4 利用缩放的字模显示示例

利用缩放的字模显示时，液晶的初始化过程与前面的工程无异，以下我们给出 LCD_Test 函数中调用字符串函数显示不同字符时的示例，见*代码清单 27_21*。

列表 21: 代码清单 27-21 利用缩放的字模显示示例

```
/* 用于测试各种液晶的函数 */
void LCD_Test(void)
{
    /* 演示显示变量 */
    static uint8_t testCNT = 0;
    char dispBuff[100];
    testCNT++;

    LCD_SetFont(&Font8x16);
    LCD_SetColors(RED,BLACK);

    ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */
    /******** 显示字符串示例 *******/
    ILI9341_DispStringLine_EN_CH(LINE(0),"野火 BH");
    //显示指定大小的字符
    ILI9341_DisplayStringEx(0,1*24,24,24,(uint8_t *)"野火 BH",0);
    ILI9341_DisplayStringEx(2*48,0*48,48,48,(uint8_t *)"野火 BH",0);

    /******** 显示变量示例 *******/
    LCD_SetTextColor(GREEN);

    /* 使用 cHAL 库把变量转化成字符串 */
```

(下页继续)

(续上页)

```
    sprintf(dispBuff,"显示变量：  %d ",testCNT);
    LCD_ClearLine(LINE(5)); /* 清除单行文字 */

    /* 然后显示该字符串即可，其它变量也是这样处理 */
    ILI9341_DispStringLine_EN_CH(LINE(5),dispBuff);
    /*... 以下部分省略 */
}
```

26.3.5 下载验证

用 USB 线连接开发板，编译程序下载到实验板，并上电复位，各个不同的工程会有不同的的液晶屏显示字符示例。

第 27 章　电阻触摸屏—触摸画板

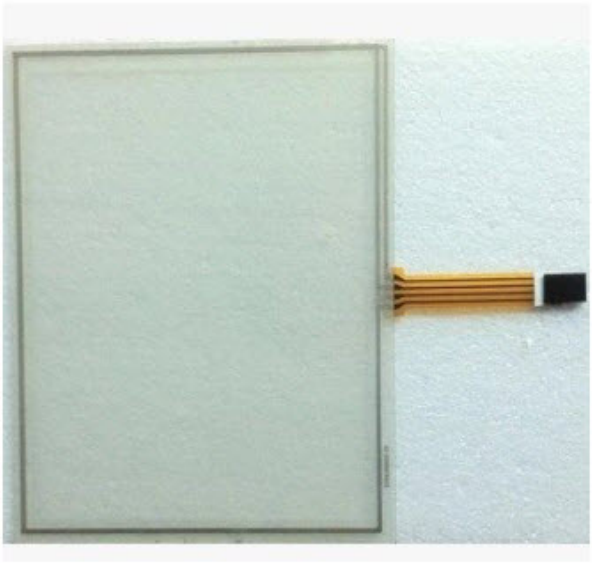
本章参考资料：《STM32F10X-中文参考手册》、《STM32F103 增强型系列数据手册》。

关于开发板配套的触摸面板配套的触摸控制芯片资料可查阅《XPT2046》数据手册获知。

在前面我们学习了如何使用 FSMC 外设控制液晶屏并用它显示各种图形及文字，利用液晶屏，STM32 的系统具有了高级信息输出功能，然而我们还希望有用户友好的输入设备，触摸屏是不二之选，目前大部分电子设备都使用触摸屏配合液晶显示器组成人机交互系统。

27.1 触摸屏简介

触摸屏又称触控面板，它是一种把触摸位置转化成坐标数据的输入设备，根据触摸屏的检测原理，主要分为电阻式触摸屏和电容式触摸屏。相对来说，电阻屏造价便宜，能适应较恶劣的环境，但它只支持单点触控 (一次只能检测面板上的一个触摸位置)，触摸时需要一定的压力，使用久了容易造成表面磨损，影响寿命；而电容屏具有支持多点触控、检测精度高的特点，电容屏通过与导电物体产生的电容效应来检测触摸动作，只能感应导电物体的触摸，湿度较大或屏幕表面有水珠时会影响电容屏的检测效果。



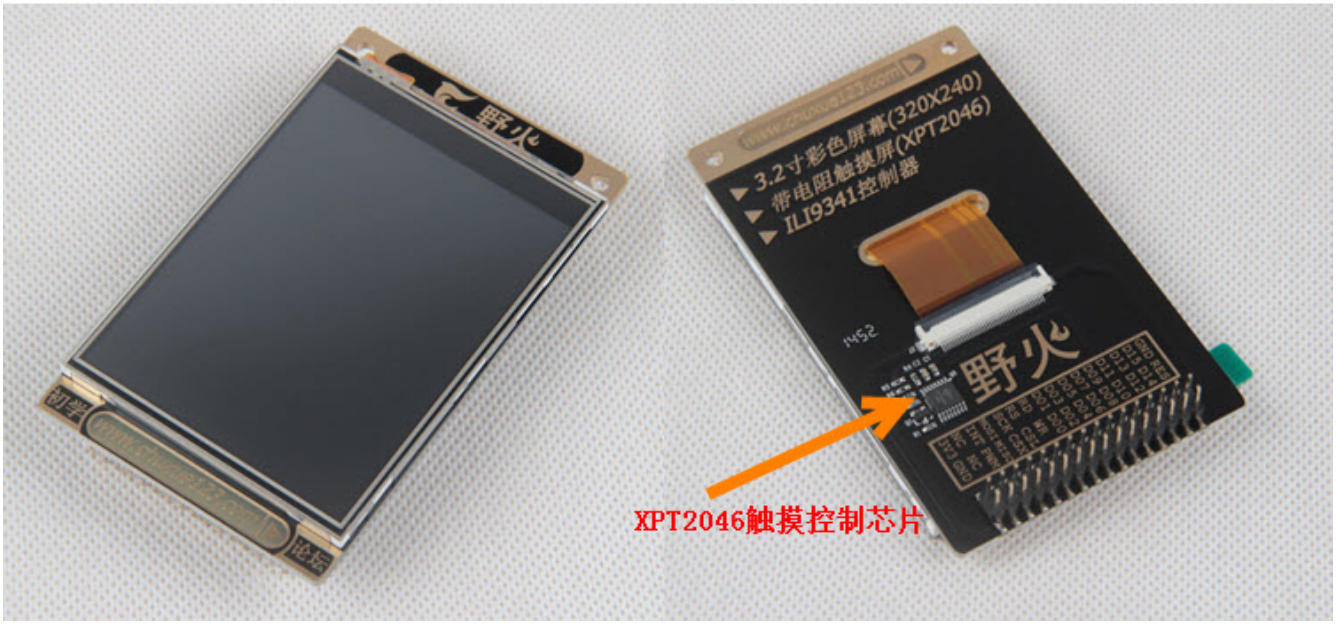
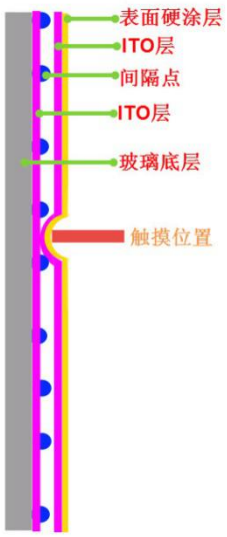

3.2寸彩色屏幕(320X240)
带电阻触摸屏(XPT2046)
ILI9341控制器
野火
XPT2046触摸控制芯片

图 *28_1* 和图 *28_2* 分别是带电阻触摸屏及电容触摸屏的两种屏幕，从外观上并没有明显的区别，区分电阻屏与电容屏最直接的方法就是使用绝缘物体点击屏幕，因为电阻屏通过压力能正常检测触摸动作，而该绝缘物体无法影响电容屏所检测的信号，因而无法检测到触摸动作。目前电容式触摸屏被大部分应用在智能手机、平板电脑等电子设备中，而在汽车导航、工控机等设备中电阻式触摸屏仍占主流。

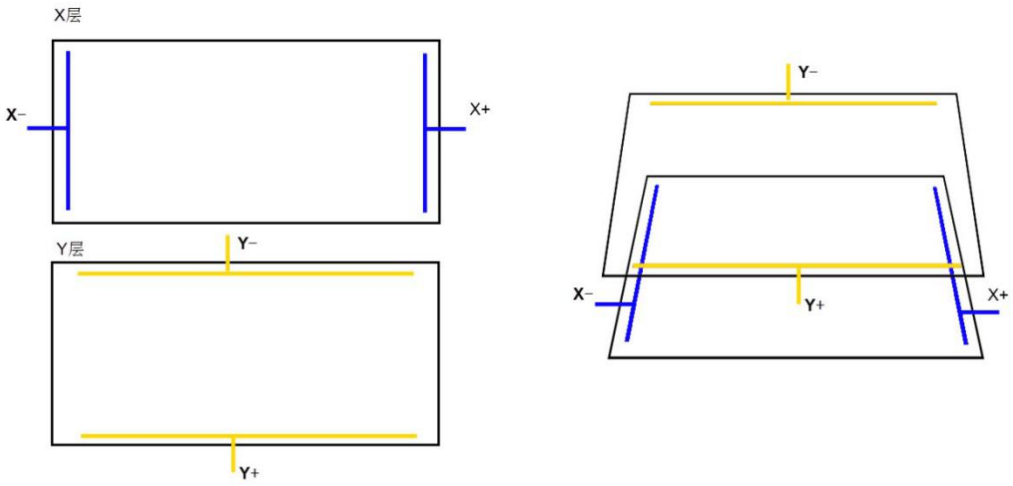
27.1.1 电阻式触摸屏检测原理

电阻式的触摸屏结构见图 *28_3*。它主要由表面硬涂层、两个 ITO 层、间隔点以及玻璃底层构成，这些结构层都是透明的，整个触摸屏覆盖在液晶面板上，透过触摸屏可看到液晶面板。表面涂层起到保护作用，玻璃底层起承载的作用，而两个 ITO 层是触摸屏的关键结构，它们是涂有铟锡金属氧化物的导电层。两个 ITO 层之间使用间隔点使两层分开，当触摸屏表面受到压力时，表面弯

曲使得上层 ITO 与下层 ITO 接触，在触点处连通电路。



两个 ITO 涂层的两端分别引出 X-、X+、Y-、Y+ 四个电极，见图 *28_4* ，这是电阻屏最常见的四线结构，通过这些电极，外部电路向这两个涂层可以施加匀强电场或检测电压。



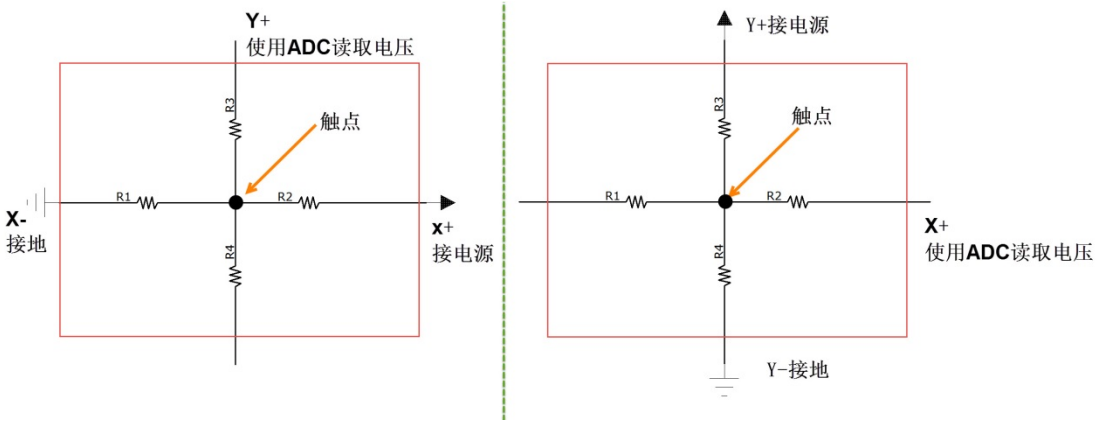
当触摸屏被按下时，两个 ITO 层相互接触，从触点处把 ITO 层分为两个电阻，且由于 ITO 层均匀导电，两个电阻的大小与触点离两电极的距离成比例关系，利用这个特性，可通过以下过程来检测坐标，这也正是电阻触摸屏名称的由来，见图 *28_5*。

- 计算 X 坐标时，在 X+ 电极施加驱动电压 V_{ref}，X-极接地，所以 X+ 与 X-处形成了匀强电场，而触点处的电压通过 Y+ 电极采集得到，由于 ITO 层均匀导电，触点电压与 V_{ref} 之比等于触点 X 坐标与屏宽度之比，从而：

$$x = \frac{V_{Y+}}{V_{\text{ref}}} \times Width$$

- 计算 Y 坐标时，在 Y+ 电极施加驱动电压 V_{ref}，Y-极接地，所以 Y+ 与 Y-处形成了匀强电场，而触点处的电压通过 X+ 电极采集得到，由于 ITO 层均匀导电，触点电压与 V_{ref} 之比等于触点 Y 坐标与屏高度之比，从而：

$$y = \frac{V_{Y+}}{V_{\text{ref}}} \times Height$$



27.1.2 电阻触摸屏控制芯片

为了方便检测触摸的坐标，一些芯片厂商制作了电阻屏专用的控制芯片，控制上述采集过程、采集电压，外部微控制器直接与触摸控制芯片通讯直接获得触点的电压或坐标。如图 28-1 中我们生产的这款 3.2 寸电阻触摸屏就是采用 XPT2046 芯片作为触摸控制芯片，XPT2046 芯片控制 4 线电阻触摸屏，STM32 与 XPT2046 采用 SPI 通讯获取采集得的电压，然后转换成坐标。

XPT2046 是专用在四线电阻屏的触摸屏控制器，STM32 可通过 SPI 接口向它写入控制字，由它测得 X、Y 方向的触点电压返回给 STM32。见图 *28_6*。

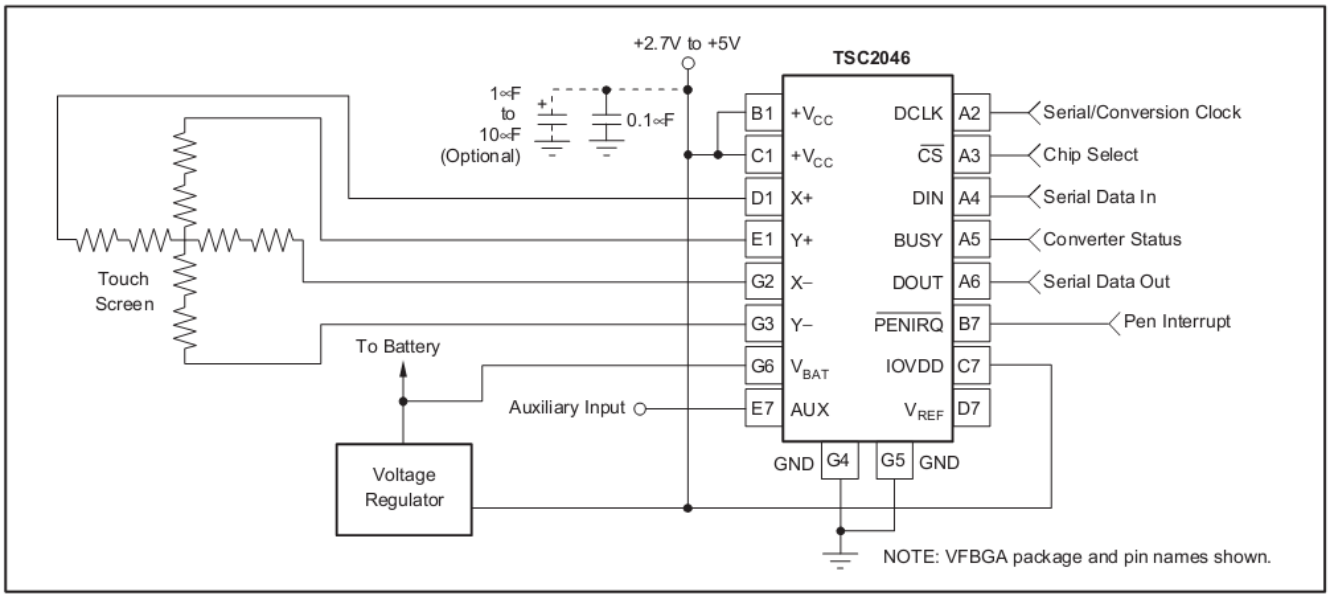


图 28-6 TSC2046 与电阻屏的连接图

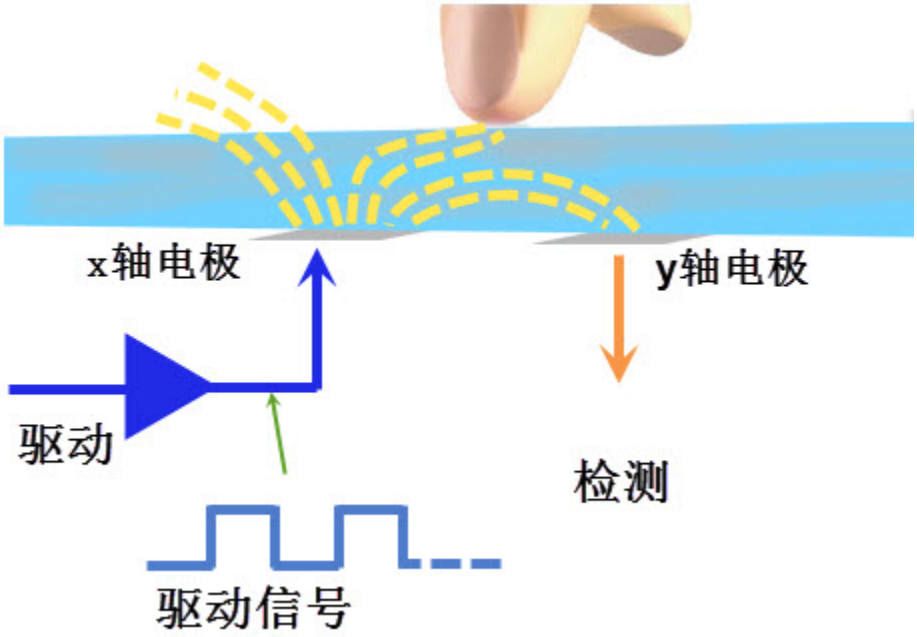
图中，电阻屏两层阻性材料的两端分别接入到 XPT2046 的 X+、X-和 Y+、Y-。当要测量 X 坐标时，STM32 通过 SPI 接口写命令到 XPT2046，使它通过内部的模拟开关使 X+、X-接通电源，于是在电阻屏的 X 方向上产生一个匀强电场；把 Y+、Y-连接到 XPT2046 的 ADC。当电阻屏被触摸时，上、下两层的阻性材料接触，在 *PENIRQ* 引脚产生一个中断信号，通知 STM32。该触点的电压由 Y+ 或 Y-(此时的 Y+Y-电阻很小，可忽略) 引入到 ADC 进行测量，STM32 读取该电压，进行软件转换，就可以测得触点 X 方向的坐标。同理可以测得 Y 方向的坐标。

XPT2046 输出的 ADC 电压值是 12 位的，这也是它型号中 2046 名称的来源。

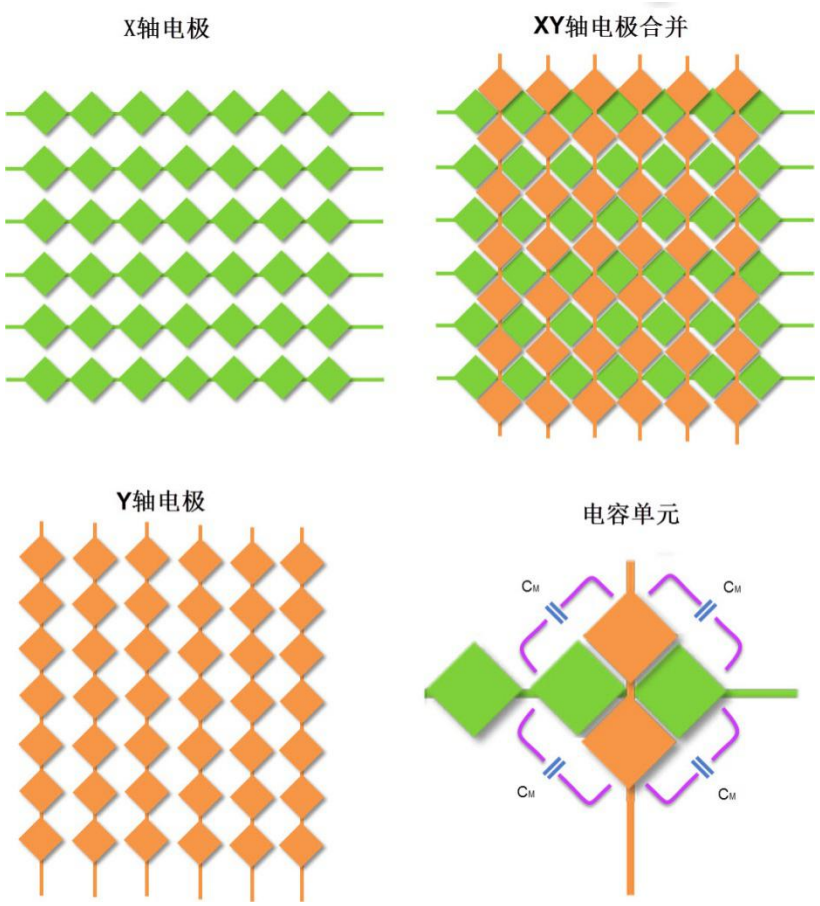
27.1.3 电容式触摸屏检测原理

与电阻式触摸屏不同，电容式触摸屏不需要通过压力使触点变形，再通过触点处电压值来检测坐标，它的基本原理和前面定时器章节中介绍的电容按键类似，都是利用充电时间检测电容大小，从而通过检测出电容值的变化来获知触摸信号。见图 *28_7*，电容屏的最上层是玻璃 (不会像电阻屏那样形变)，核心层部分也是由 ITO 材料构成的，这些导电材料在屏幕里构成了人眼看不见的静电网，静电网由多行 X 轴电极和多列 Y 轴电极构成，两个电极之间会形成电容。触摸屏工作时，X 轴电极发出 AC 交流信号，而交流信号能穿过电容，即通过 Y 轴能感应出该信号，当交流电穿越时电容会有充放电过程，检测该充电时间可获知电容量。若手指触摸屏幕，会影响触摸点

附近两个电极之间的耦合，从而改变两个电极之间的电容量，若检测到某电容的电容量发生了改变，即可获知该电容处有触摸动作（这就是为什么它被称为电容式触摸屏以及绝缘体触摸没有反应的原因）。



电容屏 ITO 层的结构见图 *28_8*，这是比较常见的形式，电极由多个菱形导体组成，生产时使用蚀刻工艺在 ITO 层生成这样的结构。



X 轴电极与 Y 轴电极在交叉处形成电容，即这两组电极构成了电容的两极，这样的结构覆盖了整个电容屏，每个电容单元在触摸屏中都有其特定的物理位置，即电容的位置就是它在触摸屏的 XY 坐标。检测触摸的坐标时，第 1 条 X 轴的电极发出激励信号，而所有 Y 轴的电极同时接收信号，通过检测充电时间可检测出各个 Y 轴与第 1 条 X 轴相交的各个互电容的大小，各个 X 轴依次发出激励信号，重复上述步骤，即可得到整个触摸屏二维平面的所有电容大小。当手指接近时，会导致局部电容改变，根据得到的触摸屏电容量变化的二维数据表，可以得知每个触摸点的坐标，因此电容触摸屏支持多点触控。

其实电容触摸屏可看作是多个电容按键组合而成，就像机械按键中独立按键和矩阵按键的关系一样，甚至电容触摸屏的坐标扫描方式与矩阵按键都是很相似的。

在野火的 F4 系列产品中有使用到电容触摸屏，感兴趣可以去了解一下。

27.2 电阻触摸屏—触摸画板实验

本小节讲解如何驱动电阻触摸屏，并利用触摸屏制作一个简易的触摸画板应用。

学习本小节内容时，请打开配套的“电阻触摸屏—触摸画板”工程配合阅读。

27.2.1 硬件设计

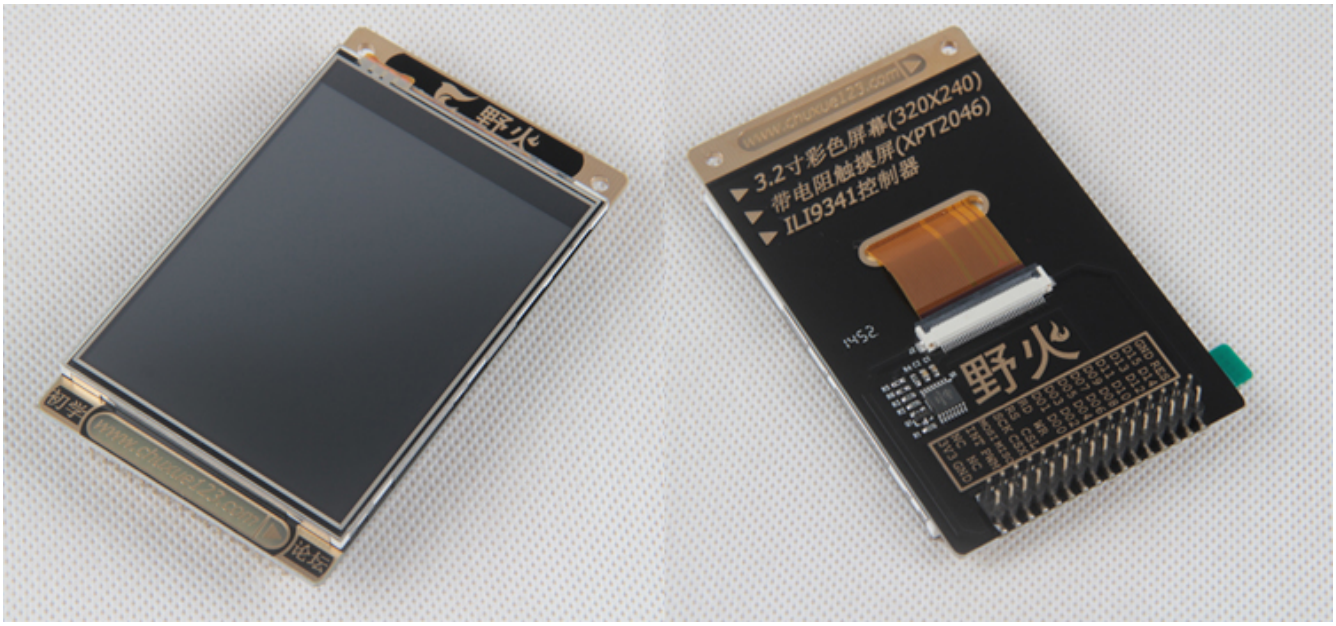
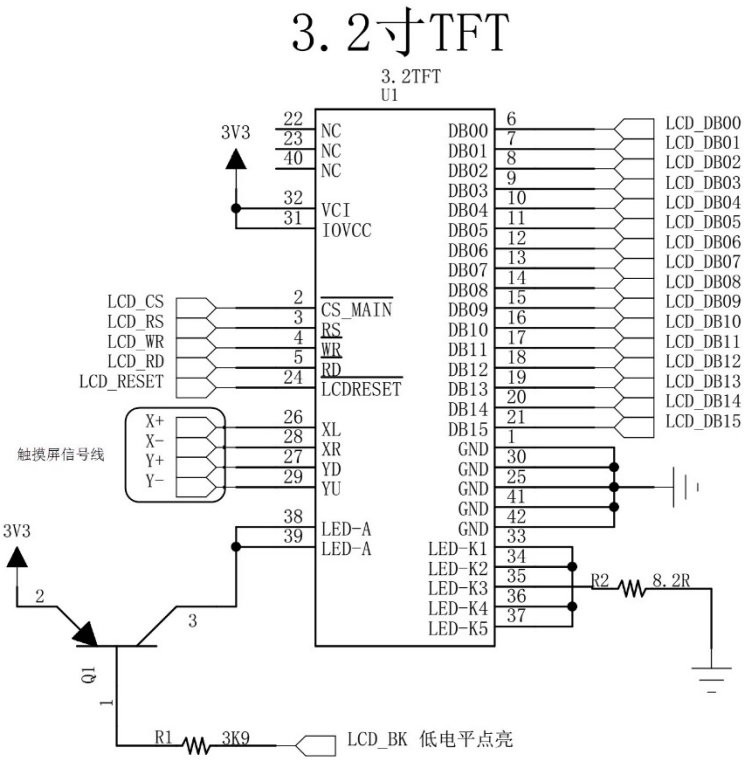
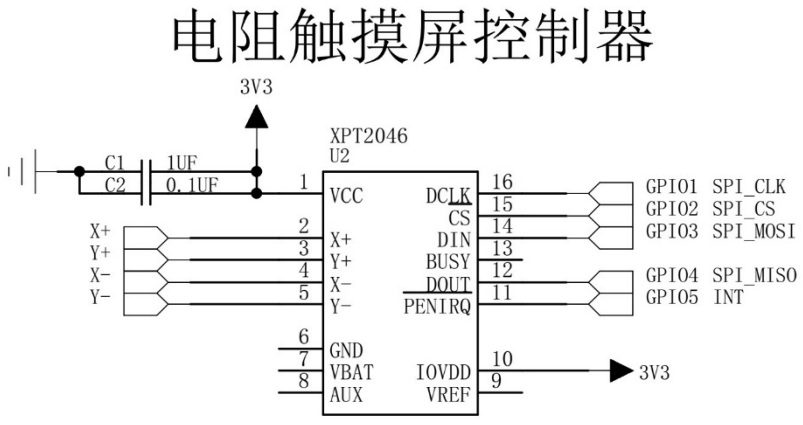


图 *28_9* 液晶屏背面的 PCB 电路对应图 *28_10*、图 *28_11*、图 *28_12* 中的原理图，分别是屏幕 PCB 底板原理图、触摸部分原理图、液晶排针接口线序图。



屏幕的 PCB 底板引出的信号线会通过 PCB 底板上的 FPC 接口与液晶面板连接，这些信包括液晶控制相关的 CS、RS 等信号及 DB0-DB15 数据线，其中 RS 引脚以高电平表示传输数据，低电平表示传输命令；另外还有引出 LCD_BK 引脚用于控制屏幕的背光供电，可以通过该引脚控制背光的强度，该引脚为低电平时打开背光。图中的 X+/X-/Y+/Y-引脚是液晶面板上触摸屏引出的信号线，它们会被连接到 PCB 底板的电阻触摸屏控制器，用于检测触摸信号，其原理图见图 *28_11*。



触摸检测的主体是型号为 XPT2046 的芯片，它接收触摸屏的 X+/X-/Y+/Y-信号进行处理，把触摸

信息使用 SPI 接口输出到 STM32 等控制器，注意，由于控制 XPT2046 芯片的并不是 STM32 专用的硬件 SPI 接口，所以在编写程序时，需要使用软件模拟 SPI 时序与触摸芯片进行通讯。

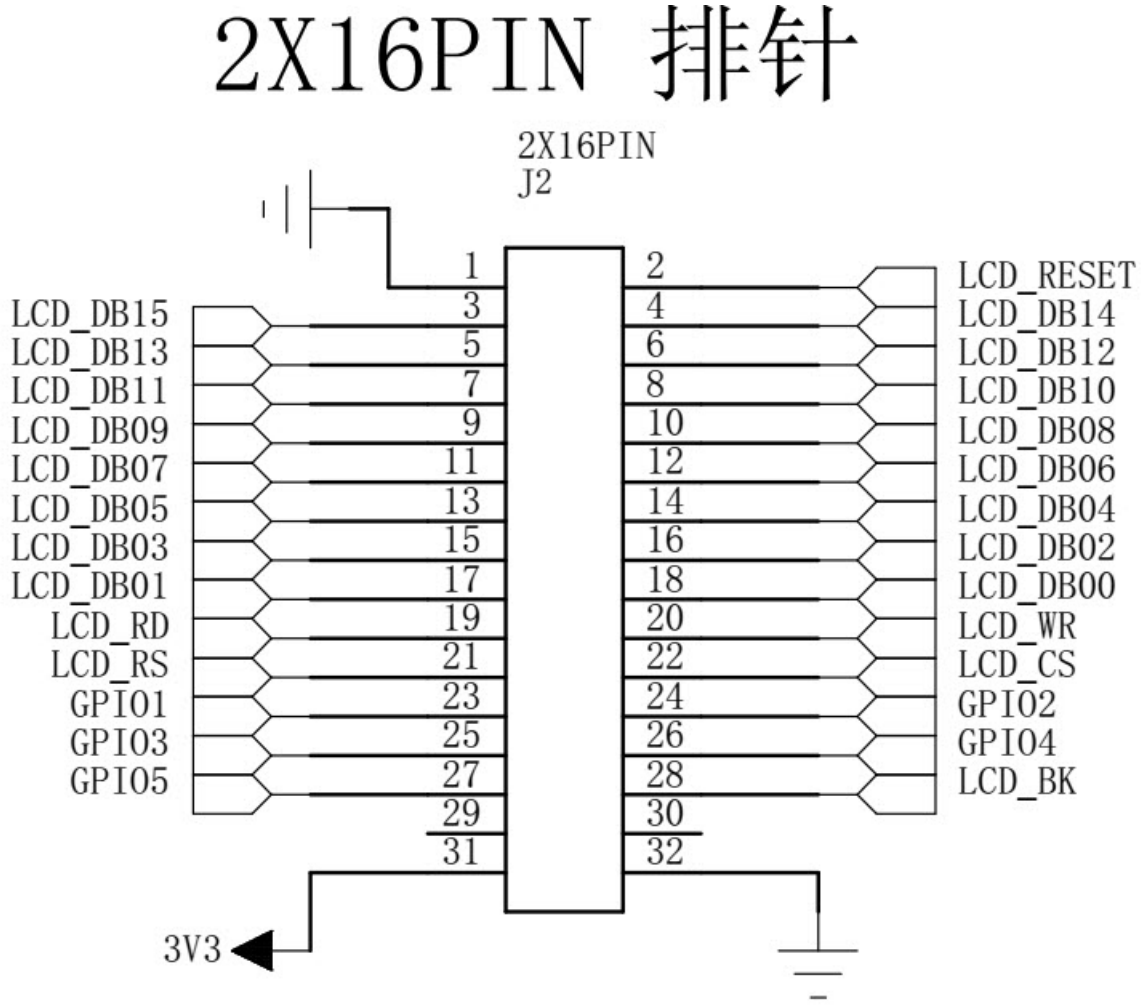


图 *28_12* 表示的是 PCB 底板引出的排针线序，屏幕整体通过这些引出的排针与开发板或其它控制器连接。

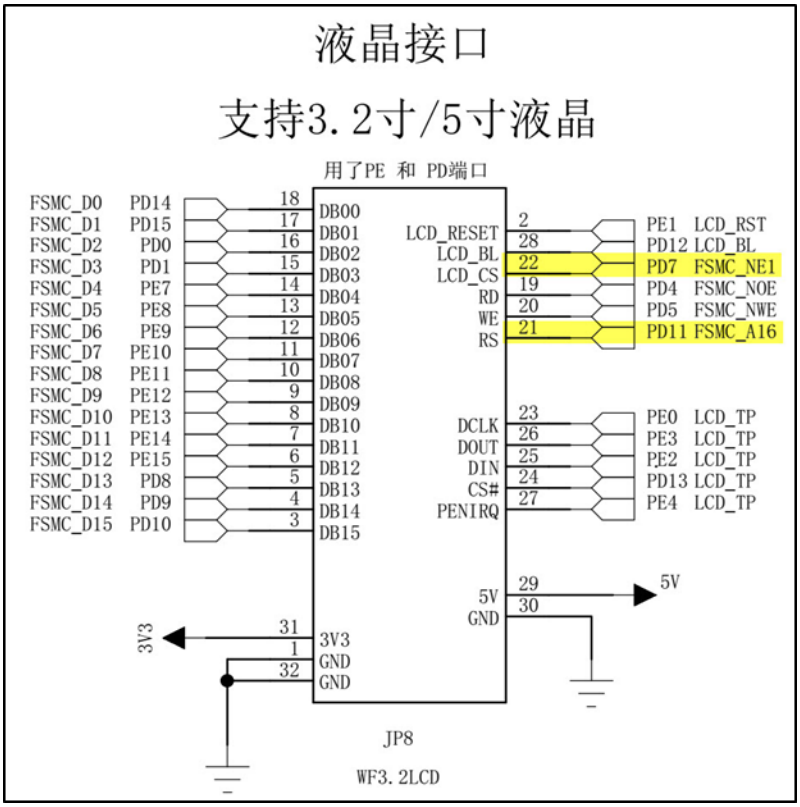


图 *28_13* 是指南者开发板上的液晶排母接口原理图，它说明了配套的 3.2 寸屏幕接入到开发板上时的信号连接关系。其中请着重关注图中液晶屏 LCD_CS 及 LCD_RS(即 DC 引脚) 与 FSMC 存储区选择引脚 FSMC_NE 及地址信号 FSMC_A 的编号，它们会决定 STM32 要使用什么内存地址来控制与液晶屏的通讯。

以上原理图可查阅《3.2 寸液晶原理图.pdf》及《指南者开发板原理图.pdf》文档获知，若您使用的液晶屏或实验板不一样，请根据实际连接的引脚修改程序。

27.2.2 软件设计

本工程中的把触摸屏相关的控制代码都存储到了“bsp_xpt2046_lcd.c”及“bsp_xpt2046_lcd.h”文件中，这些文件也可根据您的喜好命名，它们不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

27.2.2.1 编程要点

(1) 编写软件模拟 SPI 协议的驱动；

(2) 编写触摸芯片的控制驱动，如发送命令字，获取触摸坐标等；

(3) 编写触摸校正程序；

(4) 编写测试程序检验驱动。

27.2.2.2 代码分析

27.2.2.2.1 触摸屏硬件相关宏定义

根据触摸屏与 STM32 芯片的硬件连接，我们把触摸屏硬件相关的配置都以宏的形式定义到“bsp_xpt2046_lcd.h”文件中，见*代码清单 28_1*。

列表 1: 代码清单 28-1 触摸屏硬件配置相关的宏 (bsp_xpt2046_lcd.h 文件)

```
/* XPT2046 触摸屏触摸信号指示引脚定义 (不使用中断)*/
#define     XPT2046_PENIRQ_GPIO_PORT    GPIOE
#define     XPT2046_PENIRQ_GPIO_PIN     GPIO_PIN_4

//触屏信号有效电平
#define   XPT2046_PENIRQ_ActiveLevel        0
#define   XPT2046_PENIRQ_Read()         HAL_GPIO_ReadPin (
                            XPT2046_PENIRQ_GPIO_PORT,
                            XPT2046_PENIRQ_GPIO_PIN )

/*XPT2046 触摸屏模拟 SPI 引脚定义 */

#define     XPT2046_SPI_CS_PIN              GPIO_PIN_13
#define     XPT2046_SPI_CS_PORT             GPIOD
```

(下页继续)

(续上页)

```
#define    XPT2046_SPI_CLK_PIN             GPIO_PIN_0
#define    XPT2046_SPI_CLK_PORT            GPIOE

#define    XPT2046_SPI_MOSI_PIN            GPIO_PIN_2
#define    XPT2046_SPI_MOSI_PORT           GPIOE

#define    XPT2046_SPI_MISO_PIN            GPIO_PIN_3
#define    XPT2046_SPI_MISO_PORT           GPIOE

/* 直接操作寄存器的方法控制 IO */
#define digitalHi(p,i)       {p->BSRR=i;}  //设置为高电平
#define digitalLo(p,i)       {p->BSRR=(uint32_t)i << 16;}//输出低电平

#define   XPT2046_CS_ENABLE()       digitalHi( XPT2046_SPI_CS_PORT,
                            XPT2046_SPI_CS_PIN )
#define   XPT2046_CS_DISABLE()      digitalLo ( XPT2046_SPI_CS_PORT,
                            XPT2046_SPI_CS_PIN )

#define   XPT2046_CLK_HIGH()        digitalHi ( XPT2046_SPI_CLK_PORT,
                            XPT2046_SPI_CLK_PIN )
#define   XPT2046_CLK_LOW()         digitalLo ( XPT2046_SPI_CLK_PORT,
                          XPT2046_SPI_CLK_PIN )

#define   XPT2046_MOSI_1()          digitalHi ( XPT2046_SPI_MOSI_PORT,
                          XPT2046_SPI_MOSI_PIN )
#define   XPT2046_MOSI_0()          digitalLo ( XPT2046_SPI_MOSI_PORT,
                          XPT2046_SPI_MOSI_PIN )

#define   XPT2046_MISO()            HAL_GPIO_ReadPin (
                          XPT2046_SPI_MISO_PORT, XPT2046_SPI_MISO_PIN )
```

以上代码根据硬件的连接，把 STM32 与触摸屏通讯使用的引脚号和控制 CS/CLK/MOSI 引脚输出高低电平的操作、读取 MISO 引脚电平状态的操作都使用宏封装了起来，以便后面制作模拟 SPI 时序的驱动。另外，本驱动中 XPT2046 的 PENIRQ 触摸信号并没有使用中断检测，而是使用普通的引脚电平轮询获取状态的。

27.2.2.2.2 初始化触摸屏控制引脚

利用上面的宏，编写触摸屏控制引脚的初始化函数，见*代码清单 28_2*。

列表 2: 代码清单 28-2 触摸屏控制引脚的 GPIO 初始化函数 (bsp_xpt2046_lcd.c 文件)

```
void XPT2046_Init ( void )
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    /* 开启 GPIO 时钟 */
    __HAL_RCC_GPIOF_CLK_ENABLE();
    __HAL_RCC_GPIOG_CLK_ENABLE();

    /* 模拟 SPI GPIO 初始化 */
    GPIO_InitStructure.Pin=XPT2046_SPI_CLK_PIN;
    GPIO_InitStructure.Speed=GPIO_SPEED_FREQ_HIGH ;
    GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP;
    HAL_GPIO_Init(XPT2046_SPI_CLK_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = XPT2046_SPI_MOSI_PIN;
    HAL_GPIO_Init(XPT2046_SPI_MOSI_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = XPT2046_SPI_CS_PIN;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH ;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    HAL_GPIO_Init(XPT2046_SPI_CS_PORT, &GPIO_InitStructure);
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin = XPT2046_SPI_MISO_PIN;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH ;
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;   //上拉输入
    GPIO_InitStructure.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(XPT2046_SPI_MISO_PORT, &GPIO_InitStructure);

    /* 拉低片选，选择 XPT2046 */
    XPT2046_CS_DISABLE();

    //触摸屏触摸信号指示引脚，不使用中断
    GPIO_InitStructure.Pin = XPT2046_PENIRQ_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;   //上拉输入
    GPIO_InitStructure.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(XPT2046_PENIRQ_GPIO_PORT, &GPIO_InitStructure);
}
```

以上函数直接初始化了触摸屏用到的 SPI 信号线，由于使用软件模拟 SPI 协议的方式，所以它把 MISO 设置为输入，其余的 MOSI、CLK、CS 引脚均配置为普通的推挽输出模式，而 PENIRQ 作为触摸信号的输入检测，它也被设置成输入模式。

27.2.2.2.3 模拟 SPI 协议的读写时序

初始化完引脚，即可编写 SPI 协议的模拟时序，见*代码清单 28_3*。

列表 3: 代码清单 28-3 模拟 SPI 的读写时序 (bsp_xpt2046_lcd.c 文件)

```
/**
* @brief  用于 XPT2046 的简单微秒级延时函数
* @param  nCount ：延时计数值，单位为微妙
```

(下页继续)

(续上页)

```
* @retval 无
*/
static void XPT2046_DelayUS ( __IO uint32_t ulCount )
{
    uint32_t i;
    for ( i = 0; i < ulCount; i ++ ) {
        uint8_t uc = 12;      //设置值为 12，大约延 1 微秒
        while ( uc -- );      //延 1 微秒
    }
}

/**
* @brief  XPT2046 的写入命令
* @param  ucCmd ：命令
*   该参数为以下值之一：
*     @arg 0x90 ：通道 Y+ 的选择控制字
*     @arg 0xd0 ：通道 X+ 的选择控制字
* @retval 无
*/
static void XPT2046_WriteCMD ( uint8_t ucCmd )
{
    uint8_t i;
    XPT2046_MOSI_0();
    XPT2046_CLK_LOW();
    for ( i = 0; i < 8; i ++ ) {
        if ( ( ucCmd >> ( 7 - i ) ) & 0x01 ) {
            XPT2046_MOSI_1()
        } else {
            XPT2046_MOSI_0();
        }
        XPT2046_DelayUS ( 5 );
        XPT2046_CLK_HIGH();
```

(下页继续)

(续上页)

```
        XPT2046_DelayUS ( 5 );
        XPT2046_CLK_LOW();
    }
}

/**
* @brief  XPT2046 的读取命令
* @param  无
* @retval 读取到的数据
*/
static uint16_t XPT2046_ReadCMD ( void )
{
    uint8_t i;
    uint16_t usBuf=0, usTemp;
    XPT2046_MOSI_0();
    XPT2046_CLK_HIGH();
    for ( i=0; i<12; i++ ) {
        XPT2046_CLK_LOW();
        usTemp = XPT2046_MISO();
        usBuf |= usTemp << ( 11 - i );
        XPT2046_CLK_HIGH();
    }
    return usBuf;
}
```

SPI 协议的读写时序都比较简单，只要驱动好一个时钟信号传输一个数据位即可，发送数据时使用 MOSI 引脚输出电平，读取数据时从 MISO 引脚获取状态。

代码中的 XPT2046_WriteCMD 函数主要在后面用于发送控制触摸芯片的命令代码，发送不同的命令可以控制触摸芯片检测 X 坐标或 Y 坐标的触摸信号，该命令代码一般为 8 个数据位；而 XPT2046_ReadCMD 函数主要在后面用于读取触摸芯片输出的 ADC 电压值，这些 ADC 电压值一般为 12 个数据位。

27.2.2.2.4 采集触摸原始数据

利用 XPT2046_WriteCMD 及 XPT2046_ReadCMD 函数，可控制触摸屏检测并获取触摸的原始 ADC 数据，见*代码清单 28_4* 。

列表 4: 代码清单 28-4 采集触摸原始数据 (bsp_xpt2046_lcd.c 文件)

```
#define  XPT2046_CHANNEL_X     0x90            //通道 Y+ 的选择控制字
#define  XPT2046_CHANNEL_Y     0xd0            //通道 X+ 的选择控制字

/**
* @brief  对 XPT2046 选择一个模拟通道后，启动 ADC，并返回 ADC 采样结果
* @param  ucChannel
*   该参数为以下值之一：
*     @arg 0x90 : 通道 Y+ 的选择控制字
*     @arg 0xd0 : 通道 X+ 的选择控制字
* @retval 该通道的 ADC 采样结果
*/
static uint16_t XPT2046_ReadAdc ( uint8_t ucChannel )
{
    XPT2046_WriteCMD ( ucChannel );
    return  XPT2046_ReadCMD ();
}



/**
* @brief  读取 XPT2046 的 X 通道和 Y 通道的 AD 值 (12 bit，最大是 4096)
* @param  sX_Ad ：存放 X 通道 AD 值的地址
* @param  sY_Ad ：存放 Y 通道 AD 值的地址
* @retval 无
*/
static void XPT2046_ReadAdc_XY ( int16_t * sX_Ad, int16_t * sY_Ad )
```

(下页继续)

(续上页)

```
{
    int16_t sX_Ad_Temp, sY_Ad_Temp;

    sX_Ad_Temp = XPT2046_ReadAdc ( XPT2046_CHANNEL_X );
    XPT2046_DelayUS ( 1 );
    sY_Ad_Temp = XPT2046_ReadAdc ( XPT2046_CHANNEL_Y );

    * sX_Ad = sX_Ad_Temp;
    * sY_Ad = sY_Ad_Temp;
}
```

根据触摸芯片的要求，发送命令代码 XPT2046_CHANNEL_X(0x90) 后，电阻屏的 X 方向会通电，然后触摸屏使用 Y 通道检测得电压，获取到触摸点 X 方向的 ADC 原始值；发送命令代码 XPT2046_CHANNEL_Y(0xd0) 后，电阻屏的 Y 方向会通电，然后触摸屏使用 X 通道检测得电压，获取到触摸点 Y 方向的 ADC 原始值。把该过程封装起来，即可得到 XPT2046_ReadAdc 及 XPT2046_ReadAdc_XY 函数，实际应用中通常直接调用 XPT2046_ReadAdc_XY 函数以检测两个方向的触摸数据。

27.2.2.2.5 多次采样求平均值

为了使得采样更精确，工程中使用代码清单 *28_5* 中的函数来采集最终使用的数据。

列表 5: 代码清单 28-5 多次采样求平均值
(bsp_xpt2046_lcd.c 文件)

```
static uint8_t XPT2046_ReadAdc_Smooth_XY ( strType_XPT2046_Coordinate
                                           * pScreenCoordinate )
{
    uint8_t ucCount = 0, i;

    int16_t sAD_X, sAD_Y;
```

(下页继续)

(续上页)

```
    int16_t sBufferArray [ 2 ] [ 10 ] = { { 0 },{ 0 } };//坐标 X 和 Y 进行多次采样

    //存储采样中的最小值、最大值
    int32_t lX_Min, lX_Max, lY_Min, lY_Max;

    /* 循环采样 10 次 */
    do {
        XPT2046_ReadAdc_XY ( & sAD_X, & sAD_Y );

        sBufferArray [ 0 ] [ ucCount ] = sAD_X;
        sBufferArray [ 1 ] [ ucCount ] = sAD_Y;

        ucCount ++;

    } while ( ( XPT2046_PENIRQ_Read() == XPT2046_PENIRQ_ActiveLevel )
            && ( ucCount < 10 ) );
            //用户点击触摸屏时即 TP_INT_IN 信号为低并且 ucCount<10

    /* 如果触笔弹起 */
    if ( XPT2046_PENIRQ_Read() != XPT2046_PENIRQ_ActiveLevel )
        ucXPT2046_TouchFlag = 0;        //中断标志复位

    /* 如果成功采样 10 个样本 */
    if ( ucCount ==10 ) {
        lX_Max = lX_Min = sBufferArray [ 0 ] [ 0 ];
        lY_Max = lY_Min = sBufferArray [ 1 ] [ 0 ];

        for ( i = 1; i < 10; i ++ ) {
            if ( sBufferArray[ 0 ] [ i ] < lX_Min )
                lX_Min = sBufferArray [ 0 ] [ i ];
```

(下页继续)

(续上页)

```
        else if ( sBufferArray [ 0 ] [ i ] > lX_Max )
            lX_Max = sBufferArray [ 0 ] [ i ];

    }

    for ( i = 1; i < 10; i ++ ) {
        if ( sBufferArray [ 1 ] [ i ] < lY_Min )
            lY_Min = sBufferArray [ 1 ] [ i ];

        else if ( sBufferArray [ 1 ] [ i ] > lY_Max )
            lY_Max = sBufferArray [ 1 ] [ i ];
    }

    /* 去除最小值和最大值之后求平均值 */
    pScreenCoordinate ->x =  ( sBufferArray [ 0 ] [ 0 ] +
                             sBufferArray [ 0 ] [ 1 ] +
                             sBufferArray [ 0 ] [ 2 ] +
                             sBufferArray [ 0 ] [ 3 ] +
                             sBufferArray [ 0 ] [ 4 ] +
                             sBufferArray [ 0 ] [ 5 ] +
                             sBufferArray [ 0 ] [ 6 ] +
                             sBufferArray [ 0 ] [ 7 ] +
                             sBufferArray [ 0 ] [ 8 ] +
                             sBufferArray [ 0 ] [ 9 ] - lX_Min-
                             lX_Max ) >> 3;

    pScreenCoordinate ->y =  ( sBufferArray [ 1 ] [ 0 ] +
                             sBufferArray [ 1 ] [ 1 ] +
                             sBufferArray [ 1 ] [ 2 ] +
                             sBufferArray [ 1 ] [ 3 ] +
                             sBufferArray [ 1 ] [ 4 ] +
                             sBufferArray [ 1 ] [ 5 ] +
```

(下页继续)

(续上页)

```
                                    sBufferArray [ 1 ] [ 6 ] +
                                    sBufferArray [ 1 ] [ 7 ] +
                                    sBufferArray [ 1 ] [ 8 ] +
                                    sBufferArray [ 1 ] [ 9 ] - lY_Min-
                                    lY_Max ) >> 3;
        return 1;
    }
    return 0;
}
```

本函数有一个输入参数 strType_XPT2046_Coordinate 类型的结构体，它主要包含 x/y/pre_x/pre_y 四个结构体成员，其中 x/y 是用来存储最新的触摸参数值的，而 pre_x/pre_y 用于存储上一次的触摸点。本函数中仅使用了 x/y 结构体成员值，且使用它存储的是触摸屏的原始触摸数据，即 ADC 值。

代码中对 X、Y 坐标各采样 10 次，然后去除极大极小值后再取平均，计算结果即存储在结构体中的 x/y 成员值中。

27.2.2.2.6 根据原始数据计算坐标值

由 XPT2046_ReadAdc_Smooth_XY 函数得到触摸原始数据后，再使用*代码清单 28_6* 中的 XPT2046_Get_TouchedPoint 即可计算出对应的触摸坐标。

列表 6: 代码清单 28-6 根据原始数据计算坐标值
(bsp_xpt2046_lcd.c 文件)

```
typedef struct {        //校准系数结构体（最终使用）
    float dX_X,
        dX_Y,
        dX,
        dY_X,
        dY_Y,
```

(下页继续)

(续上页)

```
        dY;
} strType_XPT2046_TouchPara;

//默认触摸参数，不同的屏幕稍有差异，可重新调用触摸校准函数获取
strType_XPT2046_TouchPara strXPT2046_TouchPara[] = {
-0.006464,   -0.073259,  280.358032,    0.074878,    0.002052,   -6.545977,
↪//扫描方式 0
0.086314,    0.001891,  -12.836658,   -0.003722,   -0.065799,  254.715714,/
↪/扫描方式 1
0.002782,    0.061522,  -11.595689,    0.083393,    0.005159,  -15.650089,/
↪/扫描方式 2
0.089743,   -0.000289,  -20.612209,   -0.001374,    0.064451,  -16.054003,/
↪/扫描方式 3
0.000767,   -0.068258,  250.891769,   -0.085559,   -0.000195,  334.747650,/
↪/扫描方式 4
-0.084744,    0.000047,  323.163147,   -0.002109,   -0.066371,  260.985809,
↪//扫描方式 5
-0.001848,    0.066984,  -12.807136,   -0.084858,   -0.000805,  333.395386,
↪//扫描方式 6
-0.085470,   -0.000876,  334.023163,   -0.003390,    0.064725,   -6.211169,
↪//扫描方式 7
};

//液晶屏扫描模式，本变量主要用于方便选择触摸屏的计算参数
//参数可选值为 0-7
//调用 ILI9341_GramScan 函数设置方向时会自动更改
//LCD 刚初始化完成时会使用本默认值
uint8_t LCD_SCAN_MODE = 6;

/**
* @brief  获取 XPT2046 触摸点（校准后）的坐标
* @param  pDisplayCoordinate : 该指针存放获取到的触摸点坐标
```

(下页继续)

(续上页)

```
* @param  pTouchPara: 坐标校准系数
* @retval 获取情况
*   该返回值为以下值之一:
*     @arg 1 : 获取成功
*     @arg 0 : 获取失败
*/
uint8_t XPT2046_Get_TouchedPoint ( strType_XPT2046_Coordinate *
pDisplayCoordinate,
                          strType_XPT2046_TouchPara * pTouchPara )
{
    uint8_t ucRet = 1;           //若正常，则返回 0
    strType_XPT2046_Coordinate strScreenCoordinate;

    if ( XPT2046_ReadAdc_Smooth_XY ( & strScreenCoordinate ) ) {
        pDisplayCoordinate ->x = ( ( pTouchPara[LCD_SCAN_MODE].dX_X *
                                     strScreenCoordinate.x ) +
            ( pTouchPara[LCD_SCAN_MODE].dX_Y * strScreenCoordinate.y )

                                  pTouchPara[LCD_SCAN_MODE].dX );

        pDisplayCoordinate ->y = ( ( pTouchPara[LCD_SCAN_MODE].dY_X *
                                     strScreenCoordinate.x ) +
            ( pTouchPara[LCD_SCAN_MODE].dY_Y * strScreenCoordinate.y ) +
                                  pTouchPara[LCD_SCAN_MODE].dY );
    } else ucRet = 0;           //如果获取的触点信息有误，则返回 0

    return ucRet;
}
```

在实际应用中，并不会使用前面介绍触摸原理时讲解的直接按比例运算把**触摸原始数据物理坐标**转换成与**液晶屏像素对应的 XY 逻辑坐标**（如触摸屏输出的原始数据范围为 0-2045，液晶屏的像素 XY 坐标为 0-239 及 0-319），那种直接转换的方式误差比较大，所以通常会采用“多点触摸

校正法”来转换坐标，使用这种方式时，在应用前需要校正屏幕。校正时，使用液晶屏在特定的位置显示几个点要求用户点击，根据触摸校准算法的数学关系把逻辑坐标与物理坐标转换公式的各个系数计算出来。

这些触摸转换系数，在我们上述代码中使用 strType_XPT2046_TouchPara 类型来存储，一共有 6 个系数。利用这个数据类型，代码中定义了一个数组 strXPT2046_TouchPara，它存储了液晶屏在 8 个扫描方向时使用的转换系数，这些系数是我编写代码时使用某个液晶屏测试出来的，作为默认转换系数，不同的液晶屏这些转换系数稍有差异，若在实际使用中你感觉触摸不准确，可以使用校准函数 XPT2046_Touch_Calibrate 来重新计算自己屏幕的转换系数。

而本代码中列出的 XPT2046_Get_TouchedPoint 本函数可利用两用的转换系数计算出当前的触摸逻辑坐标。它有两个输入参数，一个参数 pDisplayCoordinate 用于存储计算后得到的触摸逻辑坐标，作为计算输出，这坐标与液晶屏对应；而参数 pTouchPara 即为校准系数，作为计算输入。在函数的内部，它先调用 XPT2046_ReadAdc_Smooth_XY 检测触摸点的原始数据物理坐标，然后代入公式中计算输出逻辑坐标。

27.2.2.2.7 触摸校正

触摸校正函数 XPT2046_Touch_Calibrate 的代码涉及到的都是数学函数映射关系的运算，比较复杂，此处作原理讲解，在工程应用需要校正时，可采用*代码清单 28_7* 中的 Calibrate_or_Get_TouchParaWithFlash 函数。

列表 7: 代码清单 28-7 校正并存储转换系数到 SPI FLASH(bsp_xpt2046_lcd.c 文件)

```
//触摸参数写到 FLASH 里的标志
#define    FLASH_TOUCH_PARA_FLAG_VALUE    0xA5

//触摸标志写到 FLASH 里的地址
#define    FLASH_TOUCH_PARA_FLAG_ADDR     (1*1024)

//触摸参数写到 FLASH 里的地址
#define    FLASH_TOUCH_PARA_ADDR      (2*1024)
```

(下页继续)

(续上页)

```
/**
* @brief  从 FLASH 中获取 或 重新校正触摸参数 (校正后会写入到 SPI FLASH 中)
* @note   若 FLASH 中从未写入过触摸参数,
*           会触发校正程序校正 LCD_Mode 指定模式的触摸参数，此时其它模式写入默认值
*
*         若 FLASH 中已有触摸参数，且不强制重新校正
*           会直接使用 FLASH 里的触摸参数值
*
*         每次校正时只会更新指定的 LCD_Mode 模式的触摸参数，其它模式的不变
* @note  本函数调用后会把液晶模式设置为 LCD_Mode
*
* @param  LCD_Mode: 要校正触摸参数的液晶模式
* @param  forceCal: 是否强制重新校正参数，可以为以下值:
*   @arg 1: 强制重新校正
*   @arg 0: 只有当 FLASH 中不存在触摸参数标志时才重新校正
* @retval 无
*/
void Calibrate_or_Get_TouchParaWithFlash(uint8_t LCD_Mode,
uint8_t forceCal)
{
    uint8_t para_flag=0;

    //初始化 FLASH
    SPI_FLASH_Init();

    //读取触摸参数标志
    SPI_FLASH_BufferRead(&para_flag,FLASH_TOUCH_PARA_FLAG_ADDR,1);

    //若不存在标志或 florceCal=1 时，重新校正参数
    if (para_flag != FLASH_TOUCH_PARA_FLAG_VALUE | forceCal ==1) {
        //若标志存在，说明原本 FLASH 内有触摸参数，
```

(下页继续)

(续上页)

```
        //先读回所有 LCD 模式的参数值，以便稍后强制更新时只更新指定 LCD 模式的参数，
其它模式的不变
    if ( para_flag == FLASH_TOUCH_PARA_FLAG_VALUE && forceCal == 1) {
SPI_FLASH_BufferRead((uint8_t *)&strXPT2046_TouchPara,FLASH_TOUCH_PARA_
↪ADDR,4*6*8);
        }

        //等待触摸屏校正完毕，更新指定 LCD 模式的触摸参数值
        while ( ! XPT2046_Touch_Calibrate (LCD_Mode) );

        //擦除扇区
        SPI_FLASH_SectorErase(0);

        //设置触摸参数标志
        para_flag = FLASH_TOUCH_PARA_FLAG_VALUE;
        //写入触摸参数标志
        SPI_FLASH_BufferWrite(&para_flag,FLASH_TOUCH_PARA_FLAG_ADDR,1);
        //写入最新的触摸参数
SPI_FLASH_BufferWrite((uint8_t*)&strXPT2046_TouchPara,FLASH_TOUCH_PARA_
↪ADDR,4*6*8);
    } else {  //若标志存在且不强制校正，则直接从 FLASH 中读取
SPI_FLASH_BufferRead((uint8_t *)&strXPT2046_TouchPara,FLASH_TOUCH_PARA_
↪ADDR,4*6*8);
    }
}
```

本函数实际上对触摸校正函数 XPT2046_Touch_Calibrate 做了封装，加入了把转换系数存储在外部 SPI FLASH 的功能，以便下次板子重新上电也能使用上一次的校正得到的系数。在调用时，使用 LCD_Mode 参数选择要校正的液晶扫描模式，使用 forceCal 选择是否要强制校正；该函数调用后会触发 XPT2046_Touch_Calibrate 的校正函数，在屏幕上显示几个触点提示用户点击，若校正成功，则会把转换系数写入到外部的 SPI FLASH 空间中并加入记录标志，下次再调用本函数的时候，若不是使用 forceCal 设置成强制校正是不会触发重新校正的过程的。

27.2.2.2.8 触摸检测状态机

前面介绍的函数都是获取坐标相关的，然而那些函数并不需要长期调用，只有当检测到触摸信号的时候，再去检测坐标即可，检测触摸信号可以使用代码清单 28_8 中的触摸状态机检测。

列表 8: 代码清单 28-8 触摸检测状态机 (bsp_xpt2046_lcd.c 文件)

```
/****** 触摸状态机相关 ******/

//触屏信号有效电平
#define  XPT2046_PENIRQ_ActiveLevel  0
#define  XPT2046_PENIRQ_Read()   GPIO_ReadInputDataBit ( XPT2046_PENIRQ_
↪GPIO_PORT,
                                                          XPT2046_PENIRQ_
↪GPIO_PIN )

typedef enum {
    XPT2046_STATE_RELEASE  = 0, //触摸释放
    XPT2046_STATE_WAITING,      //触摸按下
    XPT2046_STATE_PRESSED,      //触摸按下
} enumTouchState  ;

#define TOUCH_PRESSED          1
#define TOUCH_NOT_PRESSED      0

//触摸消抖阈值
#define DURIATION_TIME         2

/**
* @brief  触摸屏检测状态机
* @retval 触摸状态
*   该返回值为以下值之一:
```

(下页继续)

(续上页)

```
*     @arg TOUCH_PRESSED : 触摸按下
*     @arg TOUCH_NOT_PRESSED : 无触摸
*/
uint8_t XPT2046_TouchDetect(void)
{
    static enumTouchState touch_state = XPT2046_STATE_RELEASE;
    static uint32_t i;
    uint8_t detectResult = TOUCH_NOT_PRESSED;

    switch (touch_state) {
    case XPT2046_STATE_RELEASE:
        if (XPT2046_PENIRQ_Read() == XPT2046_PENIRQ_ActiveLevel) {
            //第一次出现触摸信号
            touch_state = XPT2046_STATE_WAITING;
            detectResult =TOUCH_NOT_PRESSED;
        } else {  //无触摸
            touch_state = XPT2046_STATE_RELEASE;
            detectResult =TOUCH_NOT_PRESSED;
        }
        break;

    case XPT2046_STATE_WAITING:
        if (XPT2046_PENIRQ_Read() == XPT2046_PENIRQ_ActiveLevel) {
            i++;
            //等待时间大于阈值则认为触摸被按下
            //消抖时间 = DURIATION_TIME * 本函数被调用的时间间隔
//如在定时器中调用，每 10ms 调用一次，则消抖时间为：DURIATION_TIME*10ms
            if (i > DURIATION_TIME) {
                i=0;
                touch_state = XPT2046_STATE_PRESSED;
                detectResult = TOUCH_PRESSED;
            } else {       //等待时间累加
```

(下页继续)

(续上页)

```
                touch_state = XPT2046_STATE_WAITING;
                detectResult =   TOUCH_NOT_PRESSED;
            }
        } else {  //等待时间值未达到阈值就为无效电平，当成抖动处理
            i = 0;
            touch_state = XPT2046_STATE_RELEASE;
            detectResult = TOUCH_NOT_PRESSED;
        }
        break;

    case XPT2046_STATE_PRESSED:
        if (XPT2046_PENIRQ_Read() == XPT2046_PENIRQ_ActiveLevel) {
            //触摸持续按下
            touch_state = XPT2046_STATE_PRESSED;
            detectResult = TOUCH_PRESSED;
        } else {  //触摸释放
            touch_state = XPT2046_STATE_RELEASE;
            detectResult = TOUCH_NOT_PRESSED;
        }
        break;

    default:
        touch_state = XPT2046_STATE_RELEASE;
        detectResult = TOUCH_NOT_PRESSED;
        break;
    }
    return detectResult;
}
```

当触摸屏有触点按下时，PENIRQ 引脚会输出低电平，直到没有触摸的时候，它才会输出高电平；而且 STM32 的中断只支持边沿触发（上升沿或下降沿），不支持电平触发，在触摸屏上存在类似机械按键的信号抖动，所以如果使用中断的方式来检测触摸状态并不适合，难以辨别触摸按下及

释放的情况。

状态机编程是一种非常高效的编程方式，它非常适合应用在涉及状态转换的过程控制中，上述代码采用状态机的编程方式对触摸状态进行检测，主要涉及触摸的按下、消抖及释放这三种状态转换。在应用时，本函数需要在循环体里调用，或定时调用（如每隔 10ms 调用一次），其状态转换关系见图 *28_14*。

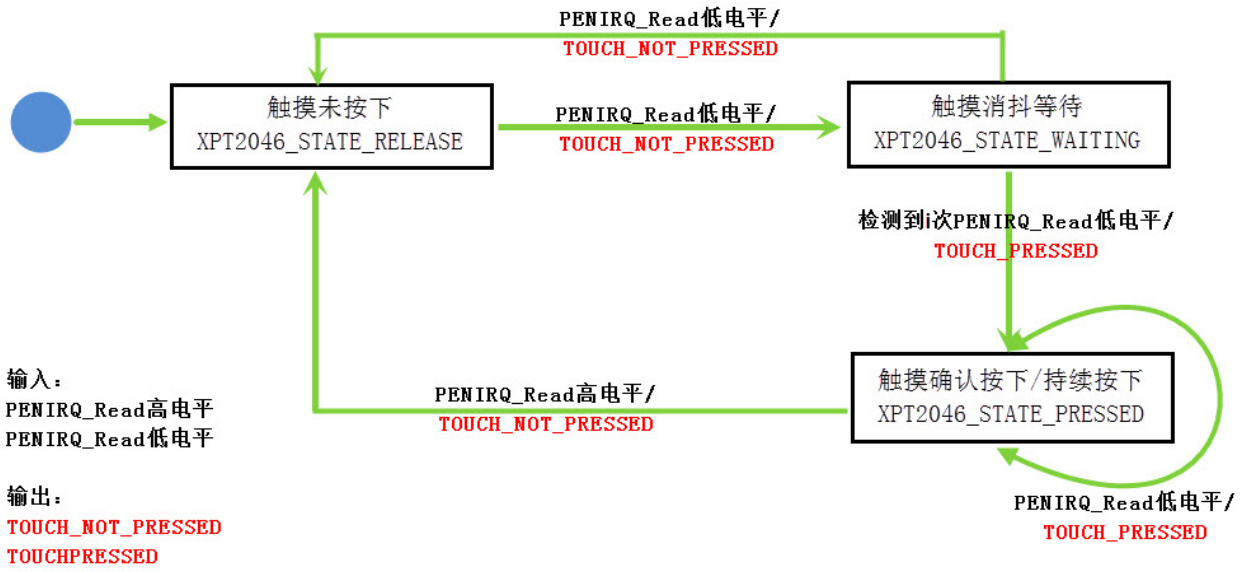


图 28-14 触摸检测状态转换图

在代码中，通过使用 XPT2046_PENIRQ_Read 函数获取当前 PENIRQ 引脚的电平，再根据当前的状态决定是否转换进入下一个状态，若经过消抖处理后进入“触摸确认按下/持续按下(XPT2046_STATE_PRESSED)”状态时，函数会返回 TOUCH_PRESSED 表示触摸被按下，其余状态返回 TOUCH_NOT_PRESSED 表示触摸无按下或释放状态。代码中的触摸消抖等待状态中，实质是通过延时、多次检测 PENIRQ 引脚的电平达到消抖的目的，若 XPT2046_TouchDetect 函数每隔 10ms 被调用一次，那么消抖的延时值则为 DURIATION_TIME*10 毫秒，可以根据实际情况适当调整该消抖阈值。

27.2.2.2.9 触摸坐标获取及处理

XPT2046_TouchDetect 函数只是检测了触摸是否被按下的状态，当触摸被按下时，还要调用前面介绍的 XPT2046_Get_TouchedPoint 函数获取触摸点的坐标，然后再处理，为便于使用，我们把这方面的操作封装到*代码清单 28_9* 中的 XPT2046_TouchEvenHandler 函数中。

列表 9: 代码清单 28-9 触摸坐标获取及处理 (bsp_xpt2046_lcd.c 文件)

```
typedef struct {         //液晶坐标结构体
    /* 负数值表示无新数据 */
    int16_t x;       //记录最新的触摸参数值
    int16_t y;

    /* 用于记录连续触摸时 (长按) 的上一次触摸位置 */
    int16_t pre_x;
    int16_t pre_y;

} strType_XPT2046_Coordinate;

/**
* @brief   检测到触摸中断时调用的处理函数，通过它调用 tp_down 和 tp_up 汇报触摸点
* @note    本函数需要在 while 循环里被调用，也可使用定时器定时调用
*
* 例如，可以每隔 5ms 调用一次，消抖阈值宏 DURIATION_TIME 可设置为 2，这样每秒最多可以检测 100 个点。
*           可在 XPT2046_TouchDown 及 XPT2046_TouchUp 函数中编写自己的触摸应用
* @param   none
* @retval  none
*/
void XPT2046_TouchEvenHandler(void )
{
    static strType_XPT2046_Coordinate cinfo= {-1,-1,-1,-1};
```

(下页继续)

(续上页)

```
    if (XPT2046_TouchDetect() == TOUCH_PRESSED) {
        LED_GREEN;

        //获取触摸坐标
        XPT2046_Get_TouchedPoint(&cinfo,strXPT2046_TouchPara);

        //输出调试信息到串口
        XPT2046_DEBUG("x=%d,y=%d",cinfo.x,cinfo.y);

        //调用触摸被按下时的处理函数，可在该函数编写自己的触摸按下处理过程
        XPT2046_TouchDown(&cinfo);

        /* 更新触摸信息到 pre xy*/
        cinfo.pre_x = cinfo.x;
        cinfo.pre_y = cinfo.y;
    } else {
        LED_BLUE;

        //调用触摸被释放时的处理函数，可在该函数编写自己的触摸释放处理过程
        XPT2046_TouchUp(&cinfo);

        /* 触笔释放，把 xy 重置为负 */
        cinfo.x = -1;
        cinfo.y = -1;
        cinfo.pre_x = -1;
        cinfo.pre_y = -1;
    }
}

/**
* @brief   触摸屏被按下的时候会调用本函数
```

(下页继续)

(续上页)

```
* @param  touch 包含触摸坐标的结构体
* @note  请在本函数中编写自己的触摸按下处理应用
* @retval 无
*/
void XPT2046_TouchDown(strType_XPT2046_Coordinate * touch)
{
    //若为负值表示之前已处理过
    if (touch->pre_x == -1 && touch->pre_x == -1)
        return;

    /*** 在此处编写自己的触摸按下处理应用 ***/

    /* 处理触摸画板的选择按钮 */
    Touch_Button_Down(touch->x,touch->y);

    /* 处理描绘轨迹 */
    Draw_Trail(touch->pre_x,touch->pre_y,touch->x,touch->y,&brush);

    /*** 在上面编写自己的触摸按下处理应用 ***/

}

/**
* @brief   触摸屏释放的时候会调用本函数
* @param  touch 包含触摸坐标的结构体
* @note  请在本函数中编写自己的触摸释放处理应用
* @retval 无
*/
void XPT2046_TouchUp(strType_XPT2046_Coordinate * touch)
{
    //若为负值表示之前已处理过
    if (touch->pre_x == -1 && touch->pre_x == -1)
```

(下页继续)

(续上页)

```
        return;

    /*** 在此处编写自己的触摸释放处理应用 ***/

    /* 处理触摸画板的选择按钮 */
    Touch_Button_Up(touch->pre_x,touch->pre_y);

    /*** 在上面编写自己的触摸释放处理应用 ***/
}
```

由于 XPT2046_TouchEvenHandler 函数带有 XPT2046_TouchDetect 状态机检测，所以它需要被循环或定时调用，以实现状态转换。当确认有触摸按下时，它会调用 XPT2046_Get_TouchedPoint 获取当前触摸坐标，并使用 XPT2046_TouchDown 函数根据触摸坐标进行处理，当触摸释放时，调用 XPT2046_TouchUp 函数处理释放坐标。

XPT2046_TouchDown 和 XPT2046_TouchUp 函数是一个接口，用户可以根据自己的应用编写相应的触摸处理程序，把触摸按下和释放的处理加入到上述函数即可。在本工程中加入了触摸画板的按钮处理（Touch_Button_Down/Up）和绘制触摸笔迹（Draw_Trail）的操作，关于触摸画板应用的内容在“palette.c”及“palette.h”文件中，这些都是与 STM32 无关上层应用，感兴趣的读者可在工程中阅读，本教程就不讲解这些内容了。。

27.2.2.3 main 函数

完成了触摸屏的驱动，就可以应用了，以下我们来看工程的主体 main 函数，见*代码清单 28_14*。

列表 10: 代码清单 28-14 main 函数

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();
    /* 初始化串口 */
    DEBUG_USART_Config();

    ILI9341_Init ();              //LCD 初始化
    XPT2046_Init();

    printf("\r\n ********** 触摸画板程序 ********** \r\n");
    printf("\r\n 若汉字显示不正常，请阅读工程中的 readme.txt 文件说明，根据要求给
FLASH 重刷字模数据\r\n ");

    //其中 0、3、5、6 模式适合从左至右显示文字，
    //不推荐使用其它模式显示文字其它模式显示文字会有镜像效果
    //其中 6 模式为大部分液晶例程的默认显示方向
    ILI9341_GramScan ( 3  );

    //绘制触摸画板界面
    Palette_Init(LCD_SCAN_MODE);

    while ( 1 ) {
        //触摸检测函数，本函数至少 10ms 调用一次
        XPT2046_TouchEvenHandler();
    }
}
```

main 函数中使用 XPT2046_Init 初始化触摸屏相关的引脚，然后调用 Calibrate_or_Get_TouchParaWithFlash 进行触摸校正；关于触摸画板应用程序的初始化，都包含在 Palette_Init 函数中，它会绘制触摸画板的按钮和白板界面；在 main 函数的 while 循环里调用了 XPT2046_TouchEvenHandler 函数，以实现状态机检测和对触摸进行处理，当有触摸按下和释放时，都通过其内部调用的 XPT2046_TouchDown 和 XPT2046_TouchUp 函数完成画板相关的操作。

27.2.3 下载验证

编译程序下载到实验板，并上电复位，液晶屏会显示出触摸画板的界面，点击屏幕可以在该界面画出简单的图形。

第 28 章　ADC—电压采集

本章参考资料：《STM32F10X-中文参考手册》ADC 章节。

学习本章时，配合《STM32F10X-中文参考手册》ADC 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

28.1 ADC 简介

STM32f103 系列有 3 个 ADC，精度为 12 位，每个 ADC 最多有 16 个外部通道。其中 ADC1 和 ADC2 都有 16 个外部通道，ADC3 根据 CPU 引脚的不同通道数也不同，一般都有 8 个外部通道。ADC 的模式非常多，功能非常强大，具体的我们在功能框图中分析每个部分的功能。

28.2 ADC 功能框图剖析

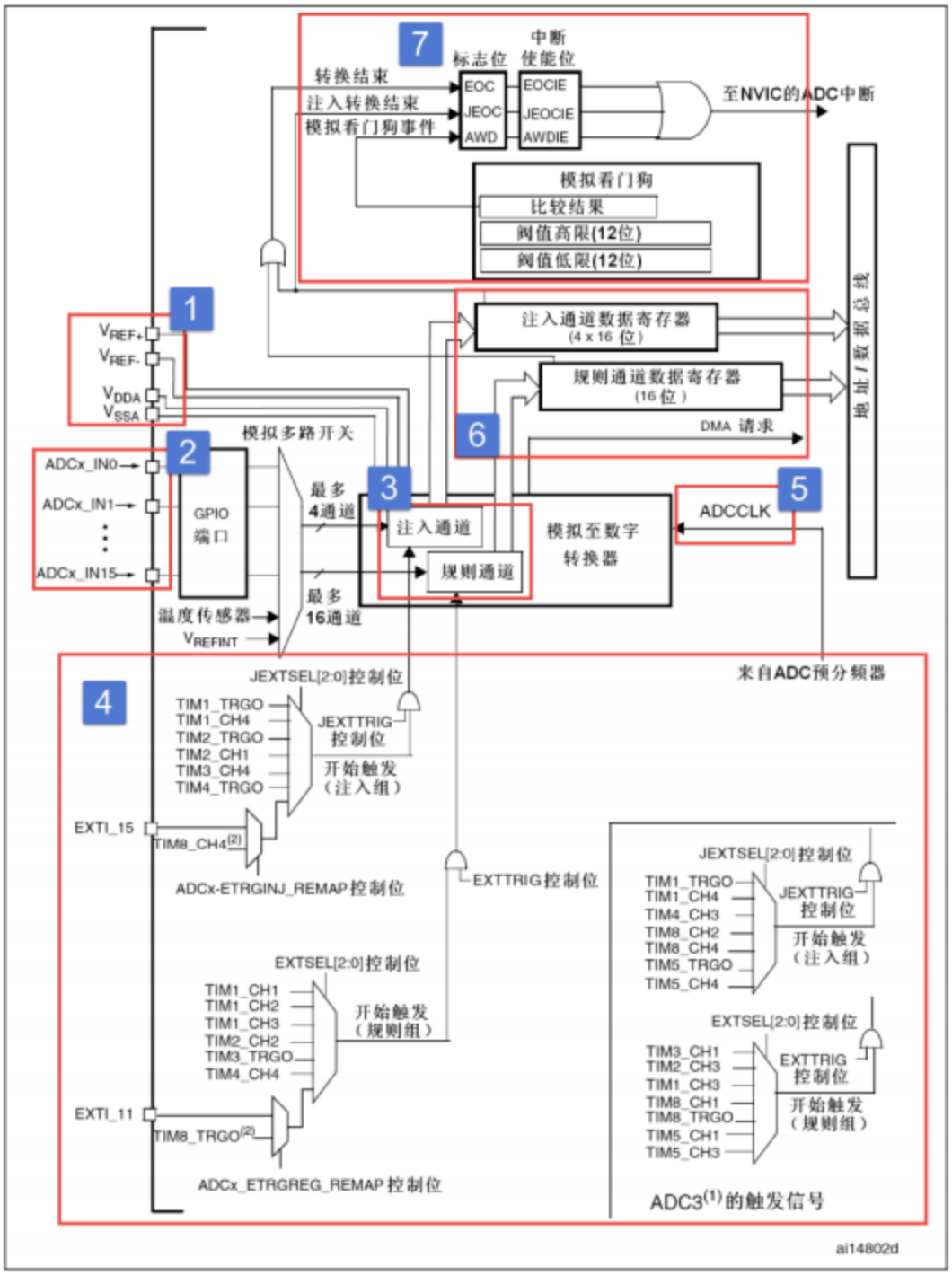


图 29-1 单个 ADC 功能框图

掌握了 ADC 的功能框图，就可以对 ADC 有一个整体的把握，在编程的时候可以做到了然如胸，不会一知半解。框图讲解采用从左到右的方式，跟 ADC 采集数据，转换数据，传输数据的方向大概一致。

28.2.1 电压输入范围

ADC 输入范围为：$V_{REF-} \leq V_{IN} \leq V_{REF+}$。由 V_{REF-}、V_{REF+}、V_{DDA}、V_{SSA}、这四个外部引脚决定。

我们在设计原理图的时候一般把 V_{SSA} 和 V_{REF-} 接地，把 V_{REF+} 和 V_{DDA} 接 3V3，得到 ADC 的输入电压范围为：0~3.3V。

如果我们想让输入的电压范围变宽，去到可以测试负电压或者更高的正电压，我们可以在外部加一个电压调理电路，把需要转换的电压抬升或者降压到 0~3.3V，这样 ADC 就可以测量了。

28.2.2 输入通道

我们确定好 ADC 输入电压之后，那么电压怎么输入到 ADC？这里我们引入通道的概念，STM32 的 ADC 多达 18 个通道，其中外部的 16 个通道就是框图中的 ADCx_IN0、ADCx_IN1…ADCx_IN5。这 16 个通道对应着不同的 IO 口，具体是哪一个 IO 口可以从手册查询到。其中 ADC1/2/3 还有内部通道：ADC1 的通道 16 连接到了芯片内部的温度传感器，Vrefint 连接到了通道 17。ADC2 的模拟通道 16 和 17 连接到了内部的 VSS。

ADC3 的模拟通道 9、14、15、16 和 17 连接到了内部的 VSS。

STM32F103ZET6 ADC IO 分配					
ADC1	IO	ADC2	IO	ADC3	IO
通道0	PA0	通道0	PA0	通道0	PA0
通道1	PA1	通道1	PA1	通道1	PA1
通道2	PA2	通道2	PA2	通道2	PA2
通道3	PA3	通道3	PA3	通道3	PA3
通道4	PA4	通道4	PA4	通道4	PF6
通道5	PA5	通道5	PA5	通道5	PF7
通道6	PA6	通道6	PA6	通道6	PF8
通道7	PA7	通道7	PA7	通道7	PF9
通道8	PB0	通道8	PB0	通道8	PF10
通道9	PB1	通道9	PB1	通道9	连接内部VSS
通道10	PC0	通道10	PC0	通道10	PC0
通道11	PC1	通道11	PC1	通道11	PC1
通道12	PC2	通道12	PC2	通道12	PC2
通道13	PC3	通道13	PC3	通道13	PC3
通道14	PC4	通道14	PC4	通道14	连接内部VSS
通道15	PC5	通道15	PC5	通道15	连接内部VSS
通道16	连接内部温度传感器	通道16	连接内部VSS	通道16	连接内部VSS
通道17	连接内部Vrefint	通道17	连接内部VSS	通道17	连接内部VSS

图 29-2 STM32F103VET6 ADC 通道

外部的 16 个通道在转换的时候又分为规则通道和注入通道，其中规则通道最多有 16 路，注入通道最多有 4 路。那这两个通道有什么区别？在什么时候使用？

28.2.2.1 规则通道

规则通道：顾名思意，规则通道就是很规矩的意思，我们平时一般使用的就是这个通道，或者应该说我们用到的都是这个通道，没有什么特别要注意的可讲。

28.2.2.2 注入通道

注入，可以理解为插入，插队的意思，是一种不安分的通道。它是一种在规则通道转换的时候强行插入要转换的一种通道。如果在规则通道转换过程中，有注入通道插队，那么就要先转换完注入通道，等注入通道转换完成后，再回到规则通道的转换流程。这点跟中断程序很像，都是不安分的主。所以，注入通道只有在规则通道存在时才会出现。

28.2.3 ADC 转换顺序

28.2.3.1 规则序列

规则序列寄存器有 3 个，分别为 SQR3、SQR2、SQR1。SQR3 控制着规则序列中的第一个到第六个转换，对应的位为：SQ1[4:0]~SQ6[4:0]，第一次转换的是位 4:0 SQ1[4:0]，如果通道 16 想第一次转换，那么在 SQ1[4:0] 写 16 即可。SQR2 控制着规则序列中的第 7 到第 12 个转换，对应的位为：SQ7[4:0]~SQ12[4:0]，如果通道 1 想第 8 个转换，则 SQ8[4:0] 写 1 即可。SQR1 控制着规则序列中的第 13 到第 16 个转换，对应位为：SQ13[4:0]~SQ16[4:0]，如果通道 6 想第 10 个转换，则 SQ10[4:0] 写 6 即可。具体使用多少个通道，由 SQR1 的位 L[3:0] 决定，最多 16 个通道。

规则序列寄存器 SQRx, x (1, 2, 3)			
寄存器	寄存器位	功能	取值
SQR3	SQ1[4:0]	设置第1个转换的通道	通道1~16
	SQ2[4:0]	设置第2个转换的通道	通道1~16
	SQ3[4:0]	设置第3个转换的通道	通道1~16
	SQ4[4:0]	设置第4个转换的通道	通道1~16
	SQ5[4:0]	设置第5个转换的通道	通道1~16
	SQ6[4:0]	设置第6个转换的通道	通道1~16
SQR2	SQ7[4:0]	设置第7个转换的通道	通道1~16
	SQ8[4:0]	设置第8个转换的通道	通道1~16
	SQ9[4:0]	设置第9个转换的通道	通道1~16
	SQ10[4:0]	设置第10个转换的通道	通道1~16
	SQ11[4:0]	设置第11个转换的通道	通道1~16
	SQ12[4:0]	设置第12个转换的通道	通道1~16
SQR1	SQ13[4:0]	设置第13个转换的通道	通道1~16
	SQ14[4:0]	设置第14个转换的通道	通道1~16
	SQ15[4:0]	设置第15个转换的通道	通道1~16
	SQ16[4:0]	设置第16个转换的通道	通道1~16
	SQL[3:0]	需要转换多少个通道	1~16

图 29-3 规则序列寄存器

28.2.3.2 注入序列

注入序列寄存器 JSQR 只有一个，最多支持 4 个通道，具体多少个由 JSQR 的 JL[2:0] 决定。如果 JL 的值小于 4 的话，则 JSQR 跟 SQR 决定转换顺序的设置不一样，第一次转换的不是 JSQR1[4:0]，而是 JCQRx[4:0]，x = （4-JL），跟 SQR 刚好相反。如果 JL=00（1 个转换），那么转换的顺序是从 JSQR4[4:0] 开始，而不是从 JSQR1[4:0] 开始，这个要注意，编程的时候不要搞错。当 JL 等于 4 时，跟 SQR 一样。

注入序列寄存器 JSQR			
寄存器	寄存器位	功能	取值
JSQR	JSQ1[4:0]	设置第1个转换的通道	通道1~4
	JSQ2[4:0]	设置第2个转换的通道	通道1~4
	JSQ3[4:0]	设置第3个转换的通道	通道1~4
	JSQ4[4:0]	设置第4个转换的通道	通道1~4
	JL[1:0]	需要转换多少个通道	1~4

图 29-4 注入序列寄存器

28.2.4 □ 触发源

通道选好了，转换的顺序也设置好了，那接下来就该开始转换了。ADC 转换可以由 ADC 控制寄存器 2: ADC_CR2 的 ADON 这个位来控制，写 1 的时候开始转换，写 0 的时候停止转换，这个是最简单也是最好理解的开启 ADC 转换的控制方式，理解起来没啥技术含量。

除了这种庶民式的控制方法，ADC 还支持触发转换，这个触发包括内部定时器触发和外部 IO 触发。触发源有很多，具体选择哪一种触发源，由 ADC 控制寄存器 2:ADC_CR2 的 EXTSEL[2:0] 和 JEXTSEL[2:0] 位来控制。EXTSEL[2:0] 用于选择规则通道的触发源，JEXTSEL[2:0] 用于选择注入通道的触发源。选定好触发源之后，触发源是否要激活，则由 ADC 控制寄存器 2:ADC_CR2 的 EXTTRIG 和 JEXTTRIG 这两位来激活。其中 ADC3 的规则转换和注入转换的触发源与 ADC1/2 的有所不同，在框图上已经表示出来。

28.2.5 转换时间

28.2.5.1 ADC 时钟

ADC 输入时钟 ADC_CLK 由 PCLK2 经过分频产生，最大是 14M，分频因子由 RCC 时钟配置寄存器 RCC_CFGR 的位 15:14 ADCPRE[1:0] 设置，可以是 2/4/6/8 分频，注意这里没有 1 分频。一般我们设置 PCLK2=HCLK=72M。

28.2.5.2 采样时间

ADC 使用若干个 ADC_CLK 周期对输入的电压进行采样，采样的周期数可通过 ADC 采样时间寄存器 ADC_SMPR1 和 ADC_SMPR2 中的 SMP[2:0] 位设置，ADC_SMPR2 控制的是通道 0~9，ADC_SMPR1 控制的是通道 10~17。每个通道可以分别用不同的时间采样。其中采样周期最小是 1.5 个，即如果我们要达到最快的采样，那么应该设置采样周期为 1.5 个周期，这里说的周期就是 1/ADC_CLK。

ADC 的转换时间跟 ADC 的输入时钟和采样时间有关，公式为：Tconv = 采样时间 + 12.5 个周期。当 ADCLK = 14MHZ（最高），采样时间设置为 1.5 周期（最快），那么总的转换时间（最短）Tconv = 1.5 周期 + 12.5 周期 = 14 周期 = 1us。

一般我们设置 PCLK2=72M，经过 ADC 预分频器能分频到最大的时钟只能是 12M，采样周期设置为 1.5 个周期，算出最短的转换时间为 1.17us，这个才是最常用的。

28.2.6 数据寄存器

一切准备就绪后，ADC 转换后的数据根据转换组的不同，规则组的数据放在 ADC_DR 寄存器，注入组的数据放在 JDRx。

28.2.6.1 规则数据寄存器

ADC 规则组数据寄存器 ADC_DR 只有一个，是一个 32 位的寄存器，低 16 位在单 ADC 时使用，高 16 位是在 ADC1 中双模式下保存 ADC2 转换的规则数据，双模式就是 ADC1 和 ADC2 同时使用。在单模式下，ADC1/2/3 都不使用高 16 位。因为 ADC 的精度是 12 位，无论 ADC_DR 的高 16 或者低 16 位都放不满，只能左对齐或者右对齐，具体是以哪一种方式存放，由 ADC_CR2 的 11 位 ALIGN 设置。

规则通道可以有 16 个这么多，可规则数据寄存器只有一个，如果使用多通道转换，那转换的数据就全部都挤在了 DR 里面，前一个时间点转换的通道数据，就会被下一个时间点的另外一个通道转换的数据覆盖掉，所以当通道转换完成后就应该把数据取走，或者开启 DMA 模式，把数据传输到内存里面，不然就会造成数据的覆盖。最常用的做法就是开启 DMA 传输。

28.2.6.2 注入数据寄存器

ADC 注入组最多有 4 个通道，刚好注入数据寄存器也有 4 个，每个通道对应着自己的寄存器，不会跟规则寄存器那样产生数据覆盖的问题。ADC_JDRx 是 32 位的，低 16 位有效，高 16 位保留，数据同样分为左对齐和右对齐，具体是以哪一种方式存放，由 ADC_CR2 的 11 位 ALIGN 设置。

28.2.7 □ 中断

28.2.7.1 转换结束中断

数据转换结束后，可以产生中断，中断分为三种：规则通道转换结束中断，注入转换通道转换结束中断，模拟看门狗中断。其中转换结束中断很好理解，跟我们平时接触的中断一样，有相应的中断标志位和中断使能位，我们还可以根据中断类型写相应配套的中断服务程序。

28.2.7.2 模拟看门狗中断

当被 ADC 转换的模拟电压低于低阈值或者高于高阈值时，就会产生中断，前提是我们开启了模拟看门狗中断，其中低阈值和高阈值由 ADC_LTR 和 ADC_HTR 设置。例如我们设置高阈值是 2.5V，那么模拟电压超过 2.5V 的时候，就会产生模拟看门狗中断，反之低阈值也一样。

28.2.7.3 DMA 请求

规则和注入通道转换结束后，除了产生中断外，还可以产生 DMA 请求，把转换好的数据直接存储在内存里面。要注意的是只有 ADC1 和 ADC3 可以产生 DMA 请求。有关 DMA 请求需要配合《STM32F10X-中文参考手册》DMA 控制器这一章节来学习。一般我们在使用 ADC 的时候都会开启 DMA 传输。

28.2.8 □ 电压转换

模拟电压经过 ADC 转换后，是一个 12 位的数字值，如果通过串口以 16 进制打印出来的话，可读性比较差，那么有时候我们就需要把数字电压转换成模拟电压，也可以跟实际的模拟电压（用万用表测）对比，看看转换是否准确。

我们一般在设计原理图的时候会把 ADC 的输入电压范围设定在：0~3.3v，因为 ADC 是 12 位的，那么 12 位满量程对应的就是 3.3V，12 位满量程对应的数字值是：2^12。数值 0 对应的就是 0V。如果转换后的数值为 X，X 对应的模拟电压为 Y，那么会有这么一个等式成立： 2^12 / 3.3 = X / Y，=> Y = (3.3 * X) / 2^12。

28.3 ADC 初始化结构体详解

HAL 库函数对每个外设都建立了一个初始化结构体 xxx_InitTypeDef(xxx 为外设名称)，结构体成员用于设置外设工作参数，并由 HAL 库函数 xxx_Init() 调用这些设定参数进入设置外设相应的寄存器，达到配置外设工作环境的目的。

结构体 xxx_InitTypeDef 和库函数 xxx_Init 配合使用是 HAL 库精髓所在，理解了结构体 xxx_InitTypeDef 每个成员意义基本上就可以对该外设运用自如了。结构体 xxx_InitTypeDef 定义

在 stm32f1xx_hal_xxx.h 文件中，库函数 xxx_Init 定义在 stm32f1xx_hal_xxx.c 文件中，编程时我们可以结合这两个文件内注释使用。

28.3.1 ADC_InitTypeDef 结构体

ADC_InitTypeDef 结构体定义在 stm32f1xx_hal_adc.h 文件内，具体定义如下：

```
typedef struct
{
    uint32_t Mode;                          // ADC 工作模式选择
    FunctionalState ScanConvMode;           /* ADC 扫描（多通道）
                                               或者单次（单通道）模式选择 */
    FunctionalState ContinuousConvMode; // ADC 单次转换或者连续转换选择
    uint32_t ExternalTrigConv;              // ADC 转换触发信号选择
    uint32_t DataAlign;                     // ADC 数据寄存器对齐格式
    uint8_t NbrOfChannel;                   // ADC 采集通道数
} ADC_InitTypeDef;
```

Mode：配置 ADC 的模式，当使用一个 ADC 时是独立模式，使用两个 ADC 时是双模式，在双模式下还有很多细分模式可选，具体配置 ADC_CR1:DUALMOD 位。

ScanConvMode：可选参数为 ENABLE 和 DISABLE，配置是否使用扫描。如果是单通道 AD 转换使用 DISABLE，如果是多通道 AD 转换使用 ENABLE，具体配置 ADC_CR1:SCAN 位。

ContinuousConvMode：可选参数为 ENABLE 和 DISABLE，配置是启动自动连续转换还是单次转换。使用 ENABLE 配置为使能自动连续转换；使用 DISABLE 配置为单次转换，转换一次后停止需要手动控制才重新启动转换，具体配置 ADC_CR2:CON 位。

ExternalTrigConv：外部触发选择，图 29-1 中列举了很多外部触发条件，可根据项目需求配置触发来源。实际上，我们一般使用软件自动触发。

DataAlign：转换结果数据对齐模式，可选右对齐 ADC_DataAlign_Right 或者左对齐 ADC_DataAlign_Left。一般我们选择右对齐模式。

NbrOfChannel：AD 转换通道数目，根据实际设置即可。具体的通道数和通道的转换顺序是配置

规则序列或注入序列寄存器。

28.4 独立模式单通道采集实验

STM32 的 ADC 功能繁多，我们设计三个实验尽量完整的展示 ADC 的功能。首先是比较基础实用的单通道采集，实现开发板上电位器电压的采集，并通过串口打印至 PC 端串口调试助手。单通道采集适用 AD 转换完成中断，在中断服务函数中读取数据，不使用 DMA 传输，在多通道采集时才使用 DMA 传输。

28.4.1 硬件设计

我们的 MINI、指南者和霸道开发板板载一个贴片滑动变阻器，滑动变阻器的滑片端连接到了 STM32 的 PC1 引脚，电路设计见图 *29_5*。

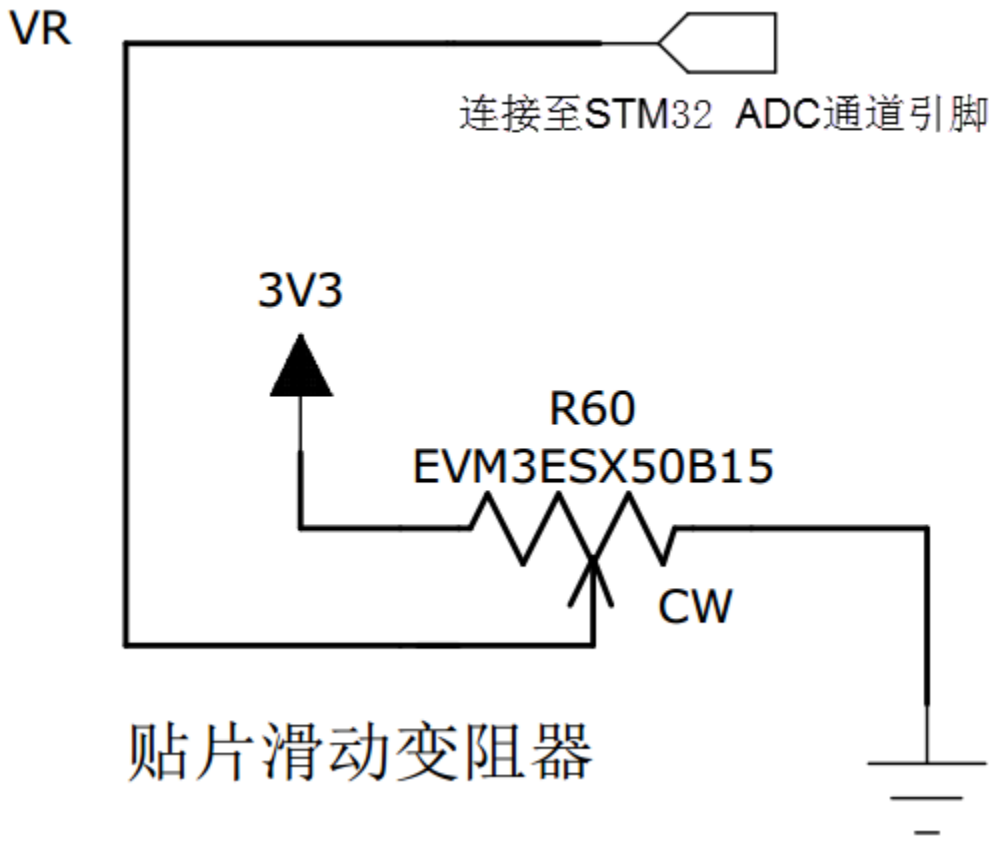


图 29-5 开发板电位器部分原理图

贴片滑动变阻器的动触点通过连接至 STM32 芯片的 ADC 通道引脚。当我们旋转滑动变阻器调节旋钮时，其动触点电压也会随之改变，电压变化范围为 0~3.3V，亦是开发板默认的 ADC 电压采集范围。

28.4.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。

我们编写两个 ADC 驱动文件，bsp_adc.h 和 bsp_adc.c，用来存放 ADC 所用 IO 引脚的初始化函数以及 ADC 配置相关函数。

28.4.2.1 编程要点

1) 初始 ADC 用到的 GPIO；
2) 设置 ADC 的工作参数并初始化；
3) 设置 ADC 工作时钟；
4) 设置 ADC 转换通道顺序及采样时间；
5) 配置使能 ADC 转换完成中断，在中断内读取转换完数据；
6) 使能 ADC；
7) 使能软件触发 ADC 转换。

ADC 转换结果数据使用中断方式读取，这里没有使用 DMA 进行数据传输。

28.4.2.2 代码分析

28.4.2.2.1 ADC 宏定义

代码清单 29-1 ADC 宏定义

```
// ADC GPIO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT                GPIOC
#define RHEOSTAT_ADC_GPIO_PIN                 GPIO_PIN_1
#define RHEOSTAT_ADC_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOC_CLK_ENABLE()

// ADC 序号宏定义
#define RHEOSTAT_ADC                          ADC1
#define RHEOSTAT_ADC_CLK_ENABLE()             __HAL_RCC_ADC1_CLK_ENABLE();
#define RHEOSTAT_ADC_CHANNEL                  ADC_CHANNEL_11

// ADC DMA 通道宏定义，这里我们使用 DMA 传输
```

(下页继续)

(续上页)

```
#define RHEOSTAT_ADC_DMA_CLK_ENABLE()        __HAL_RCC_DMA1_CLK_ENABLE();
#define RHEOSTAT_ADC_DMA_STREAM              DMA1_Channel1
```

使用宏定义引脚信息方便硬件电路改动时程序移植。

28.4.2.2.2 ADC GPIO 初始化函数

列表 1: 代码清单 29-2 ADC GPIO 初始化

```
static void Rheostat_ADC_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // 使能 GPIO 时钟
    RHEOSTAT_ADC_GPIO_CLK_ENABLE();

    // 配置 IO
    GPIO_InitStructure.Pin = RHEOSTAT_ADC_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStructure.Pull = GPIO_NOPULL ; //不上拉不下拉
    HAL_GPIO_Init(RHEOSTAT_ADC_GPIO_PORT, &GPIO_InitStructure);
}
```

使用到 GPIO 时候都必须开启对应的 GPIO 时钟，GPIO 用于 AD 转换功能必须配置为模拟输入模式。

28.4.2.2.3 配置 ADC 工作模式

列表 2: 代码清单 29-3 ADC 工作模式配置

```
static void Rheostat_ADC_Mode_Config(void)
{

    // 开启 ADC 时钟
    RHEOSTAT_ADC_CLK_ENABLE();
    // -------------------ADC Init 结构体 参数 初始化-----------------------
    // ADC1
    ADC_Handle.Instance = RHEOSTAT_ADC;
    // 时钟为 fpclk 4 分频
```

(下页继续)

(续上页)

```
    ADC_Handle.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
    // ADC 分辨率
    ADC_Handle.Init.Resolution = ADC_RESOLUTION_12B;
    // 禁止扫描模式，多通道采集才需要
    ADC_Handle.Init.ScanConvMode = DISABLE;
    // 连续转换
    ADC_Handle.Init.ContinuousConvMode = ENABLE;
    // 非连续转换
    ADC_Handle.Init.DiscontinuousConvMode = DISABLE;
    // 非连续转换个数
    ADC_Handle.Init.NbrOfDiscConversion   = 0;
    //禁止外部边沿触发
    ADC_Handle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    //使用软件触发，外部触发不用配置，注释掉即可
    //ADC_Handle.Init.ExternalTrigConv      = ADC_EXTERNALTRIGCONV_T1_CC1;
    //数据右对齐
    ADC_Handle.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    //转换通道 1 个
    ADC_Handle.Init.NbrOfConversion = 1;
    //使能连续转换请求
    ADC_Handle.Init.DMAContinuousRequests = ENABLE;
    //转换完成标志
    ADC_Handle.Init.EOCSelection          = DISABLE;
    // 初始化 ADC
    HAL_ADC_Init(&ADC_Handle);
    //---------------------------------------------------------------------
    ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL;
    ADC_Config.Rank         = 1;
    // 采样时间间隔
    ADC_Config.SamplingTime = ADC_SAMPLETIME_56CYCLES;
    ADC_Config.Offset       = 0;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时钟周期
```

(下页继续)

(续上页)

```
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);

    HAL_ADC_Start_IT(&ADC_Handle);
}
```

首先，使用 ADC_HandleTypeDef 和 ADC_ChannelConfTypeDef 结构体分别定义一个 ADC 初始化和 ADC 通道配置变量，这两个结构体我们之前已经有详细讲解。

我们调用 RHEOSTAT_ADC_CLK_ENABLE() 开启 ADC 时钟。

接下来我们使用 ADC_HandleTypeDef 结构体变量 ADC_Handle 来配置 ADC 的寄存器基地址指针、分频系数为 4、ADC1 为 12 位分辨率、单通道采集不需要扫描、启动连续转换、使用内部软件触发无需外部触发事件、使用右对齐数据格式、转换通道为 1，并调用 HAL_ADC_Init 函数完成 ADC1 工作环境配置。

使用 ADC_ChannelConfTypeDef 结构体变量 ADC_Config 来配置 ADC 的通道、转换顺序，可选为 1 到 16；采样周期选择，采样周期越短，ADC 转换数据输出周期就越短但数据精度也越低，采样周期越长，ADC 转换数据输出周期就越长同时数据精度越高。PC3 对应 ADC 通道 ADC_Channel_13，这里我们选择 ADC_SampleTime_56Cycles 即 56 周期的采样时间，调用 HAL_ADC_ConfigChannel 函数完成 ADC1 的配置。

利用 ADC 转换完成中断可以非常方便的保证我们读取到的数据是转换完成后的数据而不用担心该数据可能是 ADC 正在转换时“不稳定”的数据。我们使用 HAL_ADC_Start_IT 函数使能 ADC 转换完成中断，并在中断服务函数中读取转换结果数据。

28.4.2.2.4 ADC 中断配置

列表 3: 代码清单 29-4 ADC 中断配置

```
// 配置中断优先级
static void Rheostat_ADC_NVIC_Config(void)
{
    HAL_NVIC_SetPriority(Rheostat_ADC_IRQ, 0, 0);
```

(下页继续)

(续上页)

```
    HAL_NVIC_EnableIRQ(Rheostat_ADC_IRQ);
}
```

我们使能了 ADC 转换完成中断，需要配置中断源和中断优先级。

28.4.2.2.5 ADC 中断服务函数

列表 4: 代码清单 29-5 ADC 中断服务函数

```
void ADC_IRQHandler(void)
{
    HAL_ADC_IRQHandler(&ADC_Handle);
}
/**
* @brief  转换完成中断回调函数（非阻塞模式）
* @param  AdcHandle : ADC 句柄
* @retval 无
*/
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* AdcHandle)
{
    /* 获取结果 */
    ADC_ConvertedValue = HAL_ADC_GetValue(AdcHandle); }
}
```

中断服务函数一般定义在 stm32f1xx_it.c 文件内，HAL_ADC_IRQHandler 是 HAL 中自带的一个中断服务函数，他处理过程中会指向一个回调函数给我们去添加用户代码，这里我们使用 HAL_ADC_ConvCpltCallback 转换完成中断，在 ADC 转换完成后就会进入中断服务函数，在进入回调函数，我们在回调函数内直接读取 ADC 转换结果保存在变量 ADC_ConvertedValue(在 main.c 中定义) 中。

ADC_GetConversionValue 函数是获取 ADC 转换结果值的库函数，只有一个形参为 ADC 句柄，该函数还返回一个 16 位的 ADC 转换结果值

28.4.2.2.6 主函数

列表 5: 代码清单 29-6 主函数

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /* 配置系统时钟为 180 MHz */
    SystemClock_Config();
    /* 初始化 USART1 配置模式为 115200 8-N-1 */
    UARTx_Config();
    Rheostat_Init();
    while (1) {
    ADC_Vol =(float)ADC_ConvertedValue/4096*(float)3.3;//读取转换的 AD 值
        printf("\r\n The current AD value = 0x%04X \r\n", ADC_
↪ConvertedValue);
    printf("\r\n The current AD value = %f V \r\n",ADC_Vol);
        Delay(0x8fffff);
    }
}
```

主函数先调用 USARTx_Config 函数配置调试串口相关参数，函数定义在 bsp_debug_usart.c 文件中。

接下来调用 Rheostat _Init 函数进行 ADC 初始化配置并启动 ADC。Rheostat _Init 函数是定义在 bsp_adc.c 文件中，它只是简单的分别调用 Rheostat_ADC_GPIO_Config ()、Rheostat_ADC_Mode_Config () 和 Rheostat_ADC_NVIC_Config()。

Delay 函数只是一个简单的延时函数。

在 ADC 中断服务函数中我们把 AD 转换结果保存在变量 ADC_ConvertedValue 中，根据我们之前

的分析可以非常清楚的计算出对应的电位器动触点的电压值。

最后就是把相关数据打印至串口调试助手.

28.4.2.3 下载验证

用 USB 线连接开发板的“USB 转串口”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到不断有数据从开发板传输过来，此时我们旋转电位器改变其电阻值，那么对应的数据也会有变化。

28.5 独立模式多通道采集实验

28.5.1 硬件设计

开发板已通过排针接口把部分 ADC 通道引脚引出，其中电位器通过跳线帽默认接了一个 ADC 的 IO，其他的 ADC IO 在做实验的时候可以用杜邦线连接到开发板中的 GND 或者 3V3 来获取模拟信号。要注意的是，用来做 ADC 输入的 IO 不能被复用，否则会导致采集到的信号不准确。

28.5.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。

跟单通道例程一样，我们编写两个 ADC 驱动文件，bsp_adc.h 和 bsp_adc.c，用来存放 ADC 所用 IO 引脚的初始化函数以及 ADC 配置相关函数，实际上这两个文件跟单通道实验的文件是非常相似的。

28.5.2.1 编程要点

1) 初始化 ADC GPIO；

2) 初始化 ADC 工作参数；

3) 配置 DMA 工作参数；

4) 读取 ADC 采集的数据；

ADC 转换结果数据使用 DMA 方式传输至指定的存储区，这样取代单通道实验使用中断服务的读取方法。实际上，多通道 ADC 采集一般使用 DMA 数据传输方式更加高效方便。

28.5.2.2 代码分析

28.5.2.2.1 ADC 宏定义

列表 6: 代码清单 29-7 多通道 ADC 相关宏定义

```
#define     NOFCHANEL                       6
#define     ADC_PIN1                        GPIO_PIN_0
#define     ADC_CHANNEL1                    ADC_CHANNEL_10

#define     ADC_PIN2                        GPIO_PIN_1
#define     ADC_CHANNEL2                    ADC_CHANNEL_11

#define     ADC_PIN3                        GPIO_PIN_2
#define     ADC_CHANNEL3                    ADC_CHANNEL_12

#define     ADC_PIN4                        GPIO_PIN_3
#define     ADC_CHANNEL4                    ADC_CHANNEL_13

#define     ADC_PIN5                        GPIO_PIN_4
#define     ADC_CHANNEL5                    ADC_CHANNEL_14
```

(下页继续)

(续上页)

```
#define     ADC_PIN6                        GPIO_PIN_5
#define     ADC_CHANNEL6                    ADC_CHANNEL_15

// ADC GPIO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT              GPIOC
#define RHEOSTAT_ADC_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOC_CLK_ENABLE()

// ADC 序号宏定义
#define RHEOSTAT_ADC                        ADC1
#define RHEOSTAT_ADC_CLK_ENABLE()           __HAL_RCC_ADC1_CLK_ENABLE();

// ADC DMA 通道宏定义，这里我们使用 DMA 传输
#define RHEOSTAT_ADC_DMA_CLK_ENABLE()       __HAL_RCC_DMA1_CLK_ENABLE();
#define RHEOSTAT_ADC_DMA_STREAM             DMA1_Channel1
```

定义 NOFCHANEL 个通道进行多通道 ADC 实验，并且定义 DMA 相关配置。

28.5.2.2.2 ADC GPIO 初始化函数

列表 7: 代码清单 29-8 ADC GPIO 初始化

```
static void Rheostat_ADC_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RHEOSTAT_ADC_CLK_ENABLE();
    // 使能 GPIO 时钟
    RHEOSTAT_ADC_GPIO_CLK_ENABLE();

    // 配置 IO
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin =ADC_PIN1|
                            ADC_PIN2|
                            ADC_PIN3|
                            ADC_PIN4|
                            ADC_PIN5|
                            ADC_PIN6;
    GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStructure.Pull = GPIO_NOPULL ; //不上拉不下拉
    HAL_GPIO_Init(RHEOSTAT_ADC_GPIO_PORT, &GPIO_InitStructure);
}
```

使用到 GPIO 时候都必须开启对应的 GPIO 时钟，GPIO 用于 AD 转换功能必须配置为模拟输入模式。

28.5.2.2.3 配置 ADC 工作模式

列表 8: 代码清单 29-9 ADC 工作模式配置

```
static void Rheostat_ADC_Mode_Config(void)
{
    // ------------------DMA Init 结构体参数 初始化--------------------------
    // 开启 DMA 时钟
    RHEOSTAT_ADC_DMA_CLK_ENABLE();
    // 数据传输通道
    hdma_adcx.Instance = RHEOSTAT_ADC_DMA_STREAM;
    hdma_adcx.Init.Direction=DMA_PERIPH_TO_MEMORY;
    //存储器到外设
    hdma_adcx.Init.PeriphInc=DMA_PINC_DISABLE;
    //外设非增量模式
    hdma_adcx.Init.MemInc=DMA_MINC_ENABLE;
    //存储器增量模式
```

(下页继续)

(续上页)

```
    hdma_adcx.Init.PeriphDataAlignment=DMA_PDATAALIGN_HALFWORD;
    //外设数据长度:16 位
    hdma_adcx.Init.MemDataAlignment=DMA_PDATAALIGN_HALFWORD;
    //存储器数据长度:16 位
    hdma_adcx.Init.Mode= DMA_CIRCULAR;
    //外设普通模式
    hdma_adcx.Init.Priority=DMA_PRIORITY_MEDIUM;
    //中等优先级

    HAL_DMA_Init(&hdma_adcx);

    __HAL_LINKDMA( &ADC_Handle,DMA_Handle,hdma_adcx);

    //------------------------------------------------------------------------
↪------
    RCC_PeriphCLKInitTypeDef ADC_CLKInit;
    // 开启 ADC 时钟
    ADC_CLKInit.PeriphClockSelection=RCC_PERIPHCLK_ADC;
    //ADC 外设时钟
    ADC_CLKInit.AdcClockSelection=RCC_ADCPCLK2_DIV6;
    //分频因子 6 时钟为 72M/6=12MHz
    HAL_RCCEx_PeriphCLKConfig(&ADC_CLKInit);
    //设置 ADC 时钟

    ADC_Handle.Instance=RHEOSTAT_ADC;
    ADC_Handle.Init.DataAlign=ADC_DATAALIGN_RIGHT;
    //右对齐
    ADC_Handle.Init.ScanConvMode=ENABLE;
    //非扫描模式
    ADC_Handle.Init.ContinuousConvMode=ENABLE;
    //连续转换
    ADC_Handle.Init.NbrOfConversion=NOFCHANEL ;
```

(下页继续)

(续上页)

```
    //1 个转换在规则序列中 也就是只转换规则序列 1
    ADC_Handle.Init.DiscontinuousConvMode=DISABLE;
    //禁止不连续采样模式
    ADC_Handle.Init.NbrOfDiscConversion=0;
    //不连续采样通道数为 0
    ADC_Handle.Init.ExternalTrigConv=ADC_SOFTWARE_START;
    //软件触发
    HAL_ADC_Init(&ADC_Handle);
    //初始化
    //-------------------------------------------------------------------
↪------
    ADC_Config.Channel      = ADC_CHANNEL1;
    ADC_Config.Rank         = 1;
    // 采样时间间隔
    ADC_Config.SamplingTime = ADC_SAMPLETIME_55CYCLES_5 ;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时时钟周期
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    //-------------------------------------------------------------------
↪------
    ADC_Config.Channel      = ADC_CHANNEL2;
    ADC_Config.Rank         = 2;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时时钟周期
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    //-------------------------------------------------------------------
↪------
    ADC_Config.Channel      = ADC_CHANNEL3;
    ADC_Config.Rank         = 3;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时时钟周期
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    //-------------------------------------------------------------------
↪------
    ADC_Config.Channel      = ADC_CHANNEL4;
```

(下页继续)

(续上页)

```
    ADC_Config.Rank          = 4;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时时钟周期
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    //-------------------------------------------------------------------
↪------
    ADC_Config.Channel       = ADC_CHANNEL5;
    ADC_Config.Rank          = 5;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时时钟周期
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    //-------------------------------------------------------------------
↪------
    ADC_Config.Channel       = ADC_CHANNEL6;
    ADC_Config.Rank          = 6;
    // 配置 ADC 通道转换顺序为 1，第一个转换，采样时间为 3 个时时钟周期
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    HAL_ADC_Start_DMA(&ADC_Handle, (uint32_t*)&ADC_ConvertedValue, 1);
}
```

ADCx_Mode_Config 函数主要做了两个工作，一个是配置 ADC 的工作参数，另外一个是配置 DMA 的工作参数。

ADC 的工作参数具体如下：打开 ADC 外设时钟；因为只是使用一个 ADC，所有模式配置为独立模式；多通道采集，开启扫描模式；需要不断的采集外部的模拟数据，所有使能连续转换模式；不使用外部触发转换信号；转换结果右对齐；设置需要转换的通道的个数，最后调用 ADC_Init() 函数把这些参数写入 ADC 的寄存器完成配置。因为是多通道采集，所以调用 ADC_RegularChannelConfig() 函数设置每个通道的转换顺序和采样实际。

DMA 的工作参数具体如下：我们是把 ADC 采集到的数据通过 DMA 传输到存储器上，则外设地址为 ADC 的数据寄存器；存储器的地址是我们定义的用来存放 ADC 数据的数组的地址；传输方向为外设到存储器；缓冲区大小等于等于我们定义的存储 ADC 数据的数组大小；所有通道转换的数据都放在一个数据寄存器中，则外设地址不变；采集存储的数据有多个，则存储器地址递增；外设和存储器单位均为两个字节；开启循环传输模式；只有一个 DMA 通道工作，优先级随便设置；禁用存储器到存储器模式，最后最后调用 DMA_Init() 函数把这些参数写入 DMA 的寄存

器完成配置。

完成配置之后则使能 ADC 和 DMA，开启软件触发，让 ADC 开始采集数据。

28.5.2.2.4 主函数

列表 9: 代码清单 29-10 主函数

```
int main(void)
{
    SystemClock_Config();
    USART_Config();

    // ADC 初始化
    ADCx_Init();

    printf("\r\n ----这是一个 ADC 多通道采集 DMA 读取实验----\r\n");

    while (1)
    {

        ADC_ConvertedValueLocal[0] =(float)
                                    ADC_ConvertedValue[0]/4096*3.3;
        ADC_ConvertedValueLocal[1] =(float)
                                    ADC_ConvertedValue[1]/4096*3.3;
        ADC_ConvertedValueLocal[2] =(float)
                                    ADC_ConvertedValue[2]/4096*3.3;
        ADC_ConvertedValueLocal[3] =(float)
                                    ADC_ConvertedValue[3]/4096*3.3;
        ADC_ConvertedValueLocal[4] =(float)
                                    ADC_ConvertedValue[4]/4096*3.3;

        printf("\r\n CH1 value = %f V \r\n",ADC_ConvertedValueLocal[0]);
```

(下页继续)

(续上页)

```
        printf("\r\n CH2 value = %f V \r\n",ADC_ConvertedValueLocal[1]);
        printf("\r\n CH3 value = %f V \r\n",ADC_ConvertedValueLocal[2]);
        printf("\r\n CH2 value = %f V \r\n",ADC_ConvertedValueLocal[3]);
        printf("\r\n CH3 value = %f V \r\n",ADC_ConvertedValueLocal[4]);

        printf("\r\n\r\n");
        Delay(0xffffee);
    }
}
```

主函数中我们配置好串口，初始化好 ADC 之后，把采集到的电压经过转换之后通过串口打印到电脑的调试助手显示，要注意的是在做实验时需要给每个 ADC 通道提供模拟电源，可以用杜邦线从开发板的 GND 或者 3V3 取信号来做实验。

28.5.2.3 下载验证

将待测电压通过杜邦线接在对应引脚上，用 USB 线连接开发板的“USB 转串口”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到不断有数据从开发板传输过来，此时我们改变输入电压值，那么对应的数据也会有变化。

28.6 双重 ADC 同步规则模式采集实验

AD 转换包括采样阶段和转换阶段，在采样阶段才对通道数据进行采集；而在转换阶段只是将采集到的数据进行转换为数字量输出，此刻通道数据变化不会改变转换结果。

独立模式的 ADC 采集需要在一个通道采集并且转换完成后才会进行下一个通道的采集。而双重 ADC 的机制就是使用两个 ADC 同时采样一个或者多个通道。双重 ADC 模式较独立模式一个最大的优势就是提高了采样率，弥补了单个 ADC 采样不够快的缺点。

启用双 ADC 模式的时候，通过配置 ADC_CR1 寄存器的 DUALMOD[3:0] 位，可以有几种不同的模式，具体见表格 29-1：

表格 29-1 双 ADC 模式的各种模式汇总

模式	简要说明
同步注入模式	ADC1 和 ADC2 同时转换一个注入通道组，其中 ADC1 为主，ADC2 为从。转换的数据存储在每个 ADC 接口的 ADC_JDRx 寄存器中。
同步规则模式	ADC1 和 ADC2 同时转换一个规则通道组，其中 ADC1 为主，ADC2 为从。ADC1 转换的结果放在 ADC1_DR 的低 16 位，ADC2 转换的结果放在 ADC1_DR 的高十六位。
快速交叉模式	ADC1 和 ADC2 交替采集一个规则通道组（通常为一个通道）。当 ADC2 触发之后，ADC1 需要等待 7 个 ADCCLK 之后才能触发。
慢速交叉模式	ADC1 和 ADC2 交替采集一个规则通道组（只能为一个通道）。当 ADC2 触发之后，ADC1 需要等待 14 个 ADCCLK 之后才能触发。
交替触发模式	ADC1 和 ADC2 轮流采集注入通道组，当 ADC1 所有通道采集完毕之后再采集 ADC2 的通道，如此循环。跟交叉采集不一样。
混合的规则/注入同步模式	规则组同步转换被中断，以启动注入组的同步转换。分开两个模式来理解就可以了，区别就是注入组可以中断规则组的转换。
混合的同步规则 + 交替触发模式	规则组同步转换被中断，以启动注入组交替触发转换。分开两个模式来理解就可以了，区别就是注入组可以中断规则组的转换。
混合同步注入 + 交叉模式	交叉转换可以被同步注入模式中断。这种情况下，交叉转换被中断，注入转换被启动。

这里只对这些模式做了简要的说明，更具体的信息请参考数据手册 ADC 章节的双 ADC 模式小节。

这里我们选取同步规则模式来作为实验讲解。同步规则模式是 ADC1 和 ADC2 同时转换一个规则通道组，ADC1 是主，ADC2 是从，ADC1 转换的结果放在 ADC1_DR 的低 16 位，ADC2 转换的结果放在 ADC1_DR 的高十六位。并且必须开启 DMA 功能。

外部触发来自 ADC1 的规则组多路开关 (由 ADC1_CR2 寄存器的 EXTSEL[2:0] 选择)，它同时给 ADC2 提供同步触发。为了简单起见，ADC1 我们选择软件触发，ADC2 必须选择外部触发，这个外部触发来自于 ADC1 的规则组多路开关。

为了实验的简单起见，实验中我们选取 ADC1 和 ADC2 各采集一个通道。

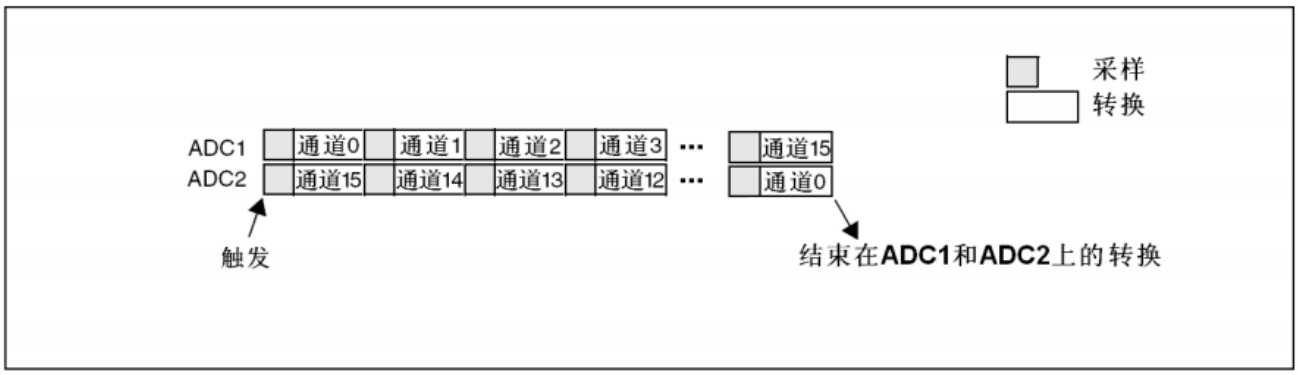


图 29-6 双重 ADC 同步规则模式

28.6.1 硬件设计

开发板用引出了 6 路 ADC，实验中选取其中的两路（根据软件配置来选择）接开发板中的电位器/GND/3V3 来获取模拟信号即可做实验。

28.6.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。

跟单通道例程一样，我们编写两个 ADC 驱动文件，bsp_adc.h 和 bsp_adc.c，用来存放 ADC 所用 IO 引脚的初始化函数以及 ADC 配置相关函数，实际上这两个文件跟单通道实验的文件非常相似。

28.6.2.1 编程要点

1) 初始化 ADC GPIO；

2) 初始化 DMA 配置；

3) 初始化 ADC 参数；

4) 读取 ADC 采集的数据，并打印出来校正；

28.6.2.2 代码分析

28.6.2.3 ADC 宏定义

列表 10: 代码清单 29-11 多通道 ADC 相关宏定义

```
#define ADCx1_RCC_CLK_ENABLE()              __HAL_RCC_ADC1_CLK_ENABLE()
#define ADCx1_RCC_CLK_DISABLE()             __HAL_RCC_ADC1_CLK_DISABLE()
#define DMAx1_RCC_CLK_ENABLE()              __HAL_RCC_DMA1_CLK_ENABLE()
#define ADCx1                               ADC1
#define ADC_DMAx_CHANNELn                   DMA1_Channel1
#define ADC_DMAx_CHANNELn_IRQn              DMA1_Channel1_IRQn
#define ADC_DMAx_CHANNELn_IRQHANDLER        DMA1_Channel1_IRQHandler

#define ADCx2_RCC_CLK_ENABLE()              __HAL_RCC_ADC2_CLK_ENABLE()
#define ADCx2_RCC_CLK_DISABLE()             __HAL_RCC_ADC2_CLK_DISABLE()
#define ADCx2                               ADC2

#define ADC_GPIO_ClK_ENABLE()               __HAL_RCC_GPIOC_CLK_ENABLE()
#define ADC_GPIO                            GPIOC
#define ADC_GPIO_PIN                        GPIO_PIN_1
#define ADC_CHANNEL                         ADC_CHANNEL_11
```

ADC1 和 ADC2 的通道由宏来定义，当硬件环境改变时，方便修改。

28.6.2.4 ADC GPIO 初始化函数

列表 11: 代码清单 29-12 ADC GPIO 初始化

```
void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if (hadc->Instance==ADCx1) {
        /* 外设时钟使能 */
        ADCx1_RCC_CLK_ENABLE();

        /* AD 转换通道引脚时钟使能 */
        ADC_GPIO_ClK_ENABLE();

        /* DMA 时钟使能 */
        DMAx1_RCC_CLK_ENABLE();

        /* AD 转换通道引脚初始化 */
        GPIO_InitStruct.Pin = ADC_GPIO_PIN;
        GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
        HAL_GPIO_Init(ADC_GPIO, &GPIO_InitStruct);

        /* DMA 外设初始化配置 */
        hdma_adcx.Instance = ADC_DMAx_CHANNELn;
        hdma_adcx.Init.Direction = DMA_PERIPH_TO_MEMORY;
        hdma_adcx.Init.PeriphInc = DMA_PINC_DISABLE;
        hdma_adcx.Init.MemInc = DMA_MINC_DISABLE;
        hdma_adcx.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
        hdma_adcx.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
        hdma_adcx.Init.Mode = DMA_CIRCULAR;
        hdma_adcx.Init.Priority = DMA_PRIORITY_HIGH;
        HAL_DMA_Init(&hdma_adcx);
        /* 连接 DMA */
```

(下页继续)

(续上页)

```
        __HAL_LINKDMA(hadc,DMA_Handle,hdma_adcx);

        /* 外设中断优先级配置和使能中断 */
        HAL_NVIC_SetPriority(ADC_DMAx_CHANNELn_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(ADC_DMAx_CHANNELn_IRQn);
    } else if (hadc->Instance==ADCx2) {
        /* 外设时钟使能 */
        ADCx2_RCC_CLK_ENABLE();

        /* AD 转换通道引脚时钟使能 */
        ADC_GPIO_ClK_ENABLE();

        /* AD 转换通道引脚初始化 */
        GPIO_InitStruct.Pin = ADC_GPIO_PIN;
        GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
        HAL_GPIO_Init(ADC_GPIO, &GPIO_InitStruct);
    }
}
```

使用到 GPIO 时候都必须开启对应的 GPIO 时钟，GPIO 用于 AD 转换功能必须配置为模拟输入模式。

28.6.2.5 配置双重 ADC 规则同步模式

列表 12: 代码清单 29-13 规则同步模式配置

```
void ADCx_Init(void)
{
    ADC_MultiModeTypeDef multimode;
    ADC_ChannelConfTypeDef sConfig;
```

(下页继续)

(续上页)

```
    // ADC 功能配置
    hadcx1.Instance = ADCx1;
    hadcx1.Init.ScanConvMode = ADC_SCAN_DISABLE;
    hadcx1.Init.ContinuousConvMode = ENABLE;
    hadcx1.Init.DiscontinuousConvMode = DISABLE;
    hadcx1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadcx1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadcx1.Init.NbrOfConversion = 1;
    HAL_ADC_Init(&hadcx1);

    /**Configure the ADC multi-mode */
    multimode.Mode = ADC_DUALMODE_INTERLSLOW;
    HAL_ADCEx_MultiModeConfigChannel(&hadcx1, &multimode);

    // 配置采样通道
    sConfig.Channel = ADC_CHANNEL;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_55CYCLES_5;
    HAL_ADC_ConfigChannel(&hadcx1, &sConfig);

    // ADC 功能配置
    hadcx2.Instance = ADCx2;
    hadcx2.Init.ScanConvMode = ADC_SCAN_DISABLE;
    hadcx2.Init.ContinuousConvMode = ENABLE;
    hadcx2.Init.DiscontinuousConvMode = DISABLE;
    hadcx2.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadcx2.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadcx2.Init.NbrOfConversion = 1;
    HAL_ADC_Init(&hadcx2);

    /**Configure the ADC multi-mode */
    multimode.Mode = ADC_DUALMODE_INTERLSLOW;
```

(下页继续)

(续上页)

```
    HAL_ADCEx_MultiModeConfigChannel(&hadcx2, &multimode);

    // 配置采样通道
    sConfig.Channel = ADC_CHANNEL;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_55CYCLES_5;
    HAL_ADC_ConfigChannel(&hadcx2, &sConfig);
}
```

ADCx_Init () 与独立模式多通道配置基本一样，只是有几点需要注意：

ADC 工作模式要设置为同步规则模式；两个 ADC 的通道的采样时间需要一致；ADC1 设置为软件触发；ADC2 设置为外部触发。其他的基本一样，看代码注释理解即可。

28.6.2.6 主函数

列表 13: 代码清单 29-14 主函数

```
int main(void)
{
    /* 配置系统时钟为 72MHz */
    SystemClock_Config();

    /* 初始化 USART1 配置模式为 115200 8-N-1 */
    DEBUG_USART_Config();

    ADCx_Init();
    HAL_ADCEx_Calibration_Start(&hadcx1);
    HAL_ADCEx_Calibration_Start(&hadcx2);

    /* 启动 AD 转换并使能 DMA 传输和中断 */
    HAL_ADC_Start(&hadcx2);
```

(下页继续)

(续上页)

```
    HAL_ADCEx_MultiModeStart_DMA(&hadcx1,&ADC_ConvertedValue,sizeof(
                                ADC_ConvertedValue));

    /* 无限循环 */
    while (1) {
        HAL_Delay(1000);
        /* 3.
        3 为 AD 转换的参考电压值，stm32 的 AD 转换为 12bit，2^1
        2=4096，
            即当输入为 3.3V 时，AD 转换结果为 4096 */
        ADC_ConvertedValueLocal[0] =(float)(ADC_ConvertedValue&0xFFF)
                                    *3.3/4096;
        ADC_ConvertedValueLocal[1] =(float)((ADC_ConvertedValue>>16)
                                    &0xFFF)*3.3/4096;

        printf("ADC1 转换原始值 = 0x%04X --> 电压值 = %f V \n",
                ADC_ConvertedValue&0xFFFF,ADC_ConvertedValueLocal[0]);
        printf("ADC2 转换原始值 = 0x%04X --> 电压值 = %f V \n",
                (ADC_ConvertedValue>>16)&0xFFFF,ADC_ConvertedValueLocal[
                1]);
        printf("\r\n");
        DMA_Transfer_Complete_Count=0;
    }
}
```

配置好串口，初始化好 ADC，然后把 ADC1 和 ADC2 采集的数据分离出来，最后调用 printf 函数打印到电脑的串口调试助手。

28.6.2.7 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板“USB 转串口”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到不断有数据从开发板传输过来，此时我们旋转电位器改变其电阻值，那么对应的数据也会有变化。

第 29 章　TIM—基本定时器

本章参考资料：《STM32F10X-中文参考手册》基本定时器章节。学习本章时，配合参考资料一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

29.1 定时器分类

STM32F1 系列中，除了互联型的产品，共有 8 个定时器，分为基本定时器，通用定时器和高级定时器。基本定时器 TIM6 和 TIM7 是一个 16 位的只能向上计数的定时器，只能定时，没有外部 IO。通用定时器 TIM2/3/4/5 是一个 16 位的可以向上/下计数的定时器，可以定时，可以输出比较，可以输入捕捉，每个定时器有四个外部 IO。高级定时器 TIM1/8 是一个 16 位的可以向上/下计数的定时器，可以定时，可以输出比较，可以输入捕捉，还可以有三相电机互补输出信号，每个定时器有 8 个外部 IO。更加具体的分类详情见图 *30_1*。

	定时器	计数器分辨率	计数器类型	预分频系数	产生DMA	捕获/比较通道	互补输出
高级定时器	TIM1	16位	向上/向下	1~65535	可以	4	有
	TIM8	16位	向上/向下	1~65535	可以	4	有
通用定时器	TIM2	16位	向上/向下	1~65535	可以	4	没有
	TIM3	16位	向上/向下	1~65535	可以	4	没有
	TIM4	16位	向上/向下	1~65535	可以	4	没有
	TIM5	16位	向上/向下	1~65535	可以	4	没有
基本定时器	TIM6	16位	向上	1~65535	可以	0	没有
	TIM7	16位	向上	1~65535	可以	0	没有

图 30-1 定时器分类

29.2 基本定时器功能框图讲解

基本定时器的核心是时基，不仅基本定时器有，通用定时器和高级定时器也有。学习定时器时，我们先从简单的基本定时器学起，到了后面的通用和高级定时器的学习中，我们直接跳过时基部分的讲解即可。基本定时器的功能框图见图 *30_2*。

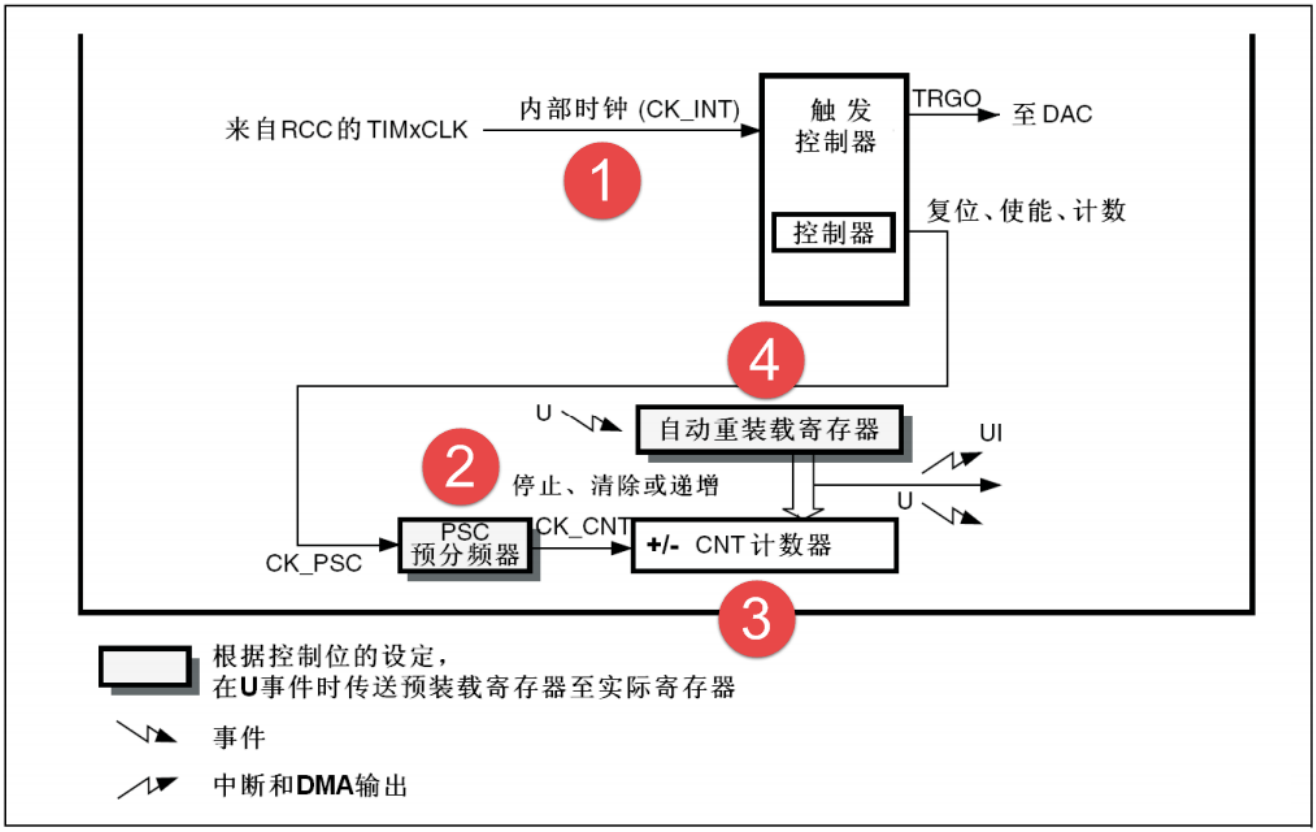


图 30-2 基本定时器功能框图

29.2.1 □ 时钟源

定时器时钟 TIMxCLK，即内部时钟 CK_INT，经 APB1 预分频器后分频提供，如果 APB1 预分频系数等于 1，则频率不变，否则频率乘以 2，库函数中 APB1 预分频的系数是 2，即 PCLK1=36M，所以定时器时钟 TIMxCLK=36*2=72M。

29.2.2 □ 计数器时钟

定时器时钟经过 PSC 预分频器之后，即计数器时钟 CK_CNT，用来驱动计数器计数。PSC 是一个

16 位的预分频器，可以对定时器时钟 TIMxCLK 进行 1~65536 之间的任何一个数进行分频。具体计算方式为：CK_CNT=TIMxCLK/(PSC+1)。

29.2.3 □ 计数器

计数器 CNT 是一个 16 位的计数器，只能往上计数，最大计数值为 65535。当计数达到自动重装载寄存器的时候产生更新事件，并清零从头开始计数。

29.2.4 □ 自动重装载寄存器

自动重装载寄存器 ARR 是一个 16 位的寄存器，这里面装着计数器能计数的最大数值。当计数到这个值的时候，如果使能了中断的话，定时器就产生溢出中断。

29.2.5 定时时间的计算

定时器的定时时间等于计数器的中断周期乘以中断的次数。计数器在 CK_CNT 的驱动下，计一个数的时间则是 CK_CNT 的倒数，等于：1/（TIMxCLK/(PSC+1)），产生一次中断的时间则等于：(1/CK_CNT) * (ARR+1)。如果在中断服务程序里面设置一个变量 time，用来记录中断的次数，那么就可以计算出我们需要的定时时间等于：(1/CK_CNT) * (ARR+1) * time。

29.3 定时器初始化结构体详解

在 HAL 库函数头文件 stm32f1xx_hal_tim.h 中对定时器外设建立了四个初始化结构体，基本定时器只用到其中一个即 TIM_TimeBaseInitTypeDef，具体的见*代码清单 30_1* ，其他三个我们在高级定时器章节讲解。

列表 1: 代码清单 30-1 定时器基本初始化结构体

```
typedef struct {
    uint32_t Prescaler;           // 预分频器
    uint32_t CounterMode;         // 计数模式
    uint32_t Period;              // 定时器周期
    uint32_t ClockDivision;       // 时钟分频
    uint32_t RepetitionCounter;    // 重复计算器
} TIM_TimeBaseInitTypeDef;
```

(1) Prescaler：定时器预分频器设置，时钟源经该预分频器才是定时器时钟，它设定 TIMx_PSC 寄存器的值。可设置范围为 0 至 65535，实现 1 至 65536 分频。

(2) CounterMode：定时器计数方式，可是在为向上计数、向下计数以及三种中心对齐模式。基本定时器只能是向上计数，即 TIMx_CNT 只能从 0 开始递增，并且无需初始化。

(3) Period：定时器周期，实际就是设定自动重载寄存器的值，在事件生成时更新到影子寄存器。可设置范围为 0 至 65535。

(4) ClockDivision：时钟分频，设置定时器时钟 CK_INT 频率与数字滤波器采样时钟频率分频比，基本定时器没有此功能，不用设置。

(5) RepetitionCounter：重复计数器，属于高级控制寄存器专用寄存器位，利用它可以非常容易控制输出 PWM 的个数。这里不用设置。

虽然定时器基本初始化结构体有 5 个成员，但对于基本定时器只需设置其中两个就可以，想想使用基本定时器就是简单。

29.4 基本定时器定时实验

29.4.1 硬件设计

本实验利用基本定时器 TIM6/7 定时 1s，1s 时间到 LED 翻转一次。基本定时器是单片机内部的资源，没有外部 IO，不需要接外部电路，现只需要一个 LED 即可。

29.4.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们编写两个定时器驱动文件，bsp_TiMbase.h 和 bsp_TiMbase.h，用来配置定时器中断优先级和和初始化定时器。

29.4.2.1 编程要点

(1) 开定时器时钟 TIMx_CLK, x[6,7]；

(2) 初始化时基初始化结构体；

(3) 使能 TIMx, x[6,7] update 中断；

(4) 打开定时器；

(5) 编写中断服务程序

通用定时器和高级定时器的定时编程要点跟基本定时器差不多，只是还要再选择下计数器的计数模式，是向上还是向下。因为基本定时器只能向上计数，且没有配置计数模式的寄存器，默认是向上。

29.4.2.2 软件分析

29.4.2.2.1 基本定时器宏定义

列表 2: 代码清单 30-2 宏定义

```
#define BASIC_TIM TIM6
#define BASIC_TIM_CLK_ENABLE()    __TIM6_CLK_ENABLE()

#define BASIC_TIM_IRQn            TIM6_DAC_IRQn
#define BASIC_TIM_IRQHandler      TIM6_DAC_IRQHandler
```

使用宏定义非常方便程序升级、移植。

29.4.2.2.2 基本定时器配置

列表 3: 代码清单 30-3 基本定时器模式配置

```
void BASIC_TIMx_Init(void)
{
    TIM_MasterConfigTypeDef sMasterConfig;

    htimx.Instance = BASIC_TIMx;
    htimx.Init.Prescaler = BASIC_TIMx_PRESCALER;
    htimx.Init.CounterMode = TIM_COUNTERMODE_UP;
    htimx.Init.Period = BASIC_TIMx_PERIOD;
    HAL_TIM_Base_Init(&htimx);

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&htimx, &sMasterConfig);
}
```

29.4.2.2.3 定时器中断优先级配置

```
// 中断优先级配置
void BASIC_TIM_NVIC_Config(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    // 设置中断组为 0
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    // 设置中断来源
    NVIC_InitStructure.NVIC_IRQChannel = BASIC_TIM_IRQ ;
    // 设置主优先级为 0
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    // 设置抢占优先级为 3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

我们设置中断分组为 0，主优先级为 0，抢占优先级为 3。

29.4.2.2.4 定时器中断服务程序

```
void  BASIC_TIM_IRQHandler (void)
{
    if ( TIM_GetITStatus( BASIC_TIM, TIM_IT_Update) != RESET ) {
        time++;
        TIM_ClearITPendingBit(BASIC_TIM , TIM_FLAG_Update);
    }
}
```

定时器中断一次的时间是 1ms，我们定义一个全局变量 time，每当进一次中断的时候，让 time 来记录进入中断的次数。如果我们想实现一个 1s 的定时，我们只需要判断 time 是否等于 1000 即

可，1000 个 1ms 就是 1s。然后把 time 清 0，重新计数，以此循环往复。在中断服务程序的最后，要把相应的中断标志位清除掉，切记。

29.4.2.2.5 主函数

```
void  BASIC_TIM_IRQHandler (void)
{
    HAL_TIM_IRQHandler(&TIM_TimeBaseStructure);
}
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim==(&TIM_TimeBaseStructure)) {
        LED1_TOGGLE;  //红灯周期闪烁
    }
}
```

我们在 TIM_Mode_Config 函数启动了定时器更新中断，在发生中断时，中断服务函数就得到运行。在服务函数内直接调用库函数 HAL_TIM_IRQHandler 函数，它会产生一个中断回调函数 HAL_TIM_PeriodElapsedCallback，用来添加用户代码，确定是 TIM6 产生中断后才运行 RGB 彩灯翻转动作。

29.4.3 下载验证

把编写好的程序下载到开发板，可以看到 LED1 以 1s 的频率闪烁一次。

第 30 章　TIM—高级定时器

本章参考资料：《STM32F10X-中文参考手册》高级定时器章节。学习本章时，配合参考资料一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

30.1 高级控制定时器

高级控制定时器 (TIM1 和 TIM8) 和通用定时器在基本定时器的基础上引入了外部引脚，可以实现输入捕获和输出比较功能。高级控制定时器比通用定时器增加了可编程死区互补输出、重复计数器、带刹车 (断路) 功能，这些功能都是针对工业电机控制方面。这几个功能在本书不做详细的介绍，主要介绍常用的输入捕获和输出比较功能。

高级控制定时器时基单元包含一个 16 位自动重装载寄存器 ARR，一个 16 位的计数器 CNT，可向上/下计数，一个 16 位可编程预分频器 PSC，预分频器时钟源有多种可选，有内部的时钟、外部时钟。还有一个 8 位的重复计数器 RCR，这样最高可实现 40 位的可编程定时。

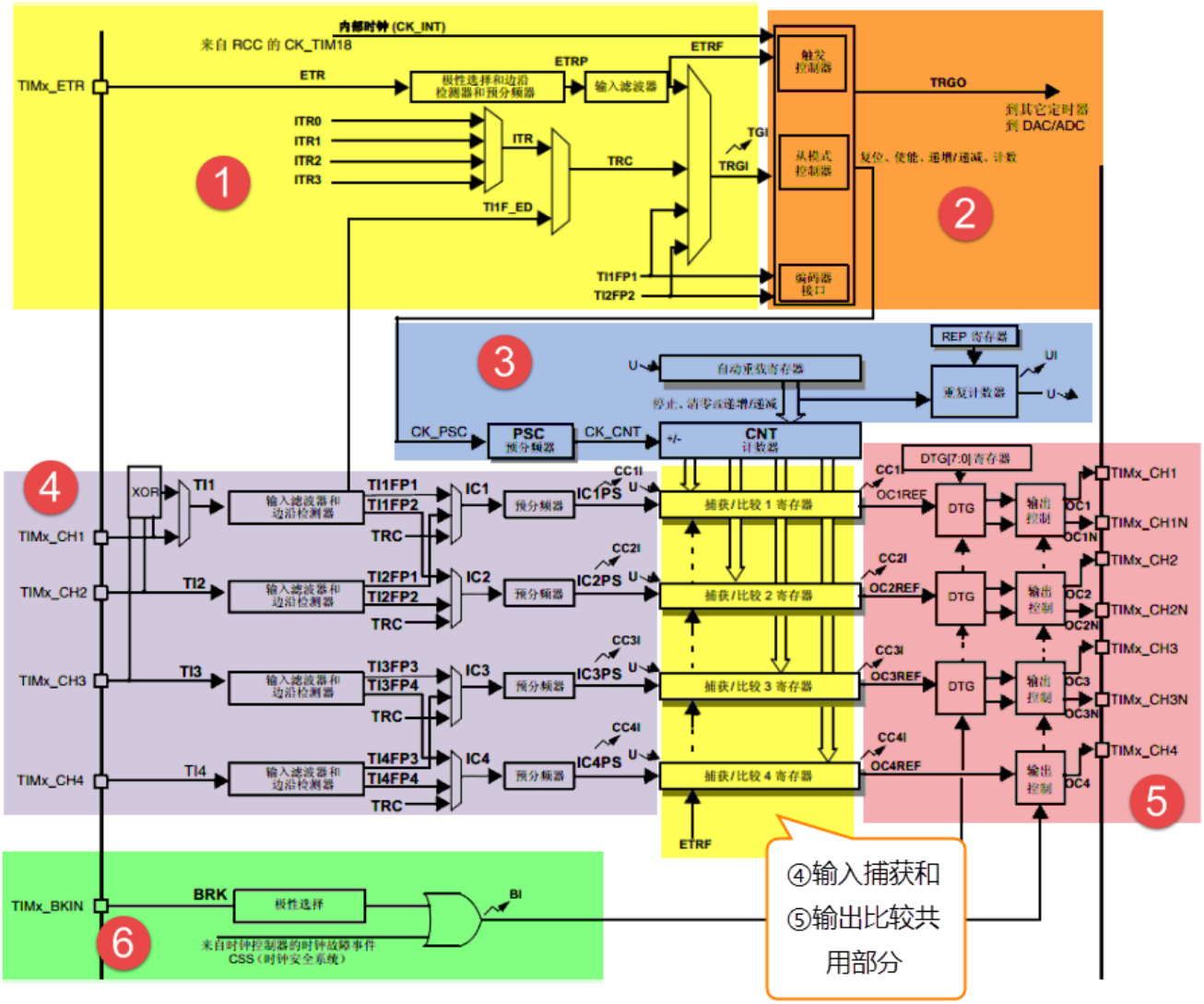
STM32F103VET6 的高级/通用定时器的 IO 分配具体见表 31-1。配套开发板因为 IO 资源紧缺，定时器的 IO 很多已经复用它途，故下表中的 IO 只有部分可用于定时器的实验。

表 31-1 高级控制和通用定时器通道引脚分布

	高级定时器	通用定时器				
	TIM1	TIM8	TIM2	TIM5	TIM3	TIM4
CH1	PA8/PE9	PC6	PA0/PA15	PA0	PA6/PC6/PB4	PB6/PD12
CH1N	PB13/PA7/PE8	PA7				
CH2	PA9/PE11	PC7	PA1/PB3	PA1	PA7/PC7/PB5	PB7/PD13
CH2N	PB14/PB0/PE10	PB0				
CH3	PA10/PE13	PC8	PA2/PB10	PA2	PB0/PC8	PB8/PD14
CH3N	PB15/PB1/PE12	PB1				
CH4	PA11/PE14	PC9	PA3/PB11	PA3	PB1/PC9	PB9/PD15
ETR	PA12/PE7	PA0	PA0/PA15		PD2	PE0
BKIN	PB12/PA6/PE15	PA6				

30.2 高级控制定时器功能框图

高级控制定时器功能框图包含了高级控制定时器最核心内容，掌握了功能框图，对高级控制定时器就有一个整体的把握，在编程时思路就非常清晰，见，图中有些寄存器是带影子的，表示其有影子寄存器。



30.2.1 时钟源

高级控制定时器有四个时钟源可选：

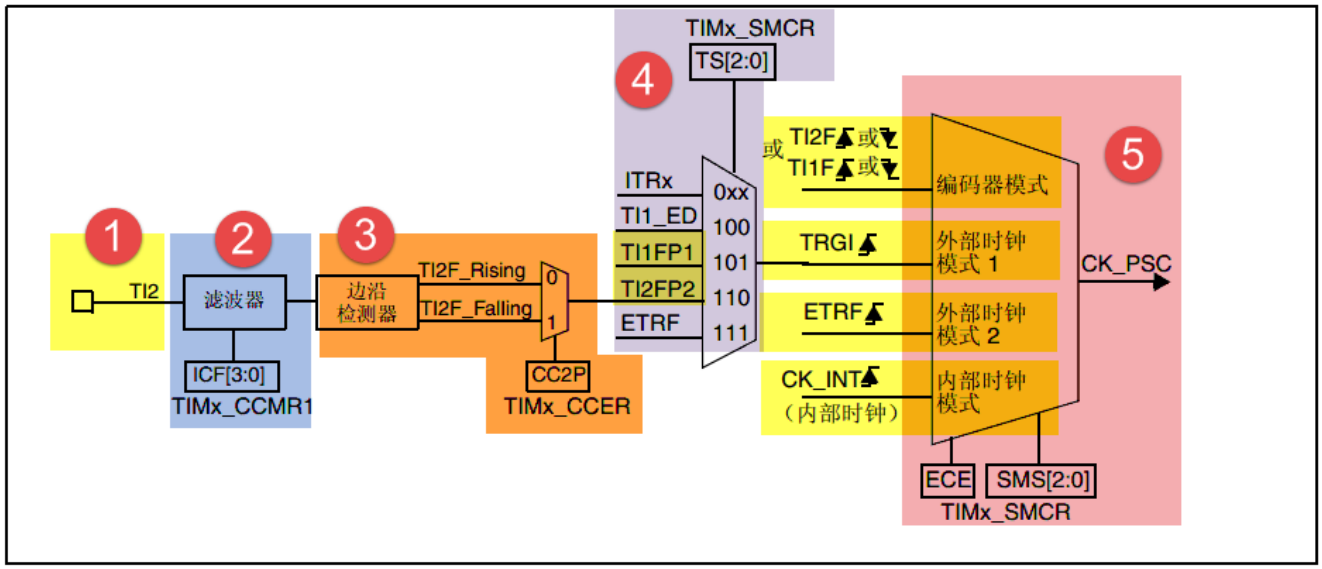
- 内部时钟源 CK_INT
- 外部时钟模式 1：外部输入引脚 TIx（x=1,2,3,4）
- 外部时钟模式 2：外部触发输入 ETR

- 内部触发输入 (ITRx)

30.2.1.1 内部时钟源 (CK_INT)

内部时钟 CK_INT 即来自于芯片内部，等于 72M，一般情况下，我们都是使用内部时钟。当从模式控制寄存器 TIMx_SMCR 的 SMS 位等于 000 时，则使用内部时钟。

30.2.1.2 外部时钟模式 1



口：时钟信号输入引脚

当使用外部时钟模式 1 的时候，时钟信号来自于定时器的输入通道，总共有 4 个，分别为 TI1/2/3/4，即 TIMx_CH1/2/3/4。具体使用哪一路信号，由 TIM_CCMRx 的位 CCxS[1:0] 配置，其中 CCMR1 控制 TI1/2，CCMR2 控制 TI3/4。

口：滤波器

如果来自外部的时钟信号的频率过高或者混杂有高频干扰信号的话，我们就需要使用滤波器对信号重新采样，来达到降频或者去除高频干扰的目的，具体的由 TIMx_CCMRx 的位 ICxF[3:0] 配置。

口：边沿检测

边沿检测的信号来自于滤波器的输出，在成为触发信号之前，需要进行边沿检测，决定是上升沿有效还是下降沿有效，具体的由 TIMx_CCER 的位 CCxP 和 CCxNP 配置。

□：触发选择

当使用外部时钟模式 1 时，触发源有两个，一个是滤波后的定时器输入 1（TI1FP1）和滤波后的定时器输入 2（TI2FP2），具体的由 TIMxSMCR 的位 TS[2:0] 配置。

□：从模式选择

选定了触发源信号后，最后我们需把信号连接到 TRGI 引脚，让触发信号成为外部时钟模式 1 的输入，最终等于 CK_PSC，然后驱动计数器 CNT 计数。具体的配置 TIMx_SMCR 的位 SMS[2:0] 为 000 即可选择外部时钟模式 1。

□：使能计数器

经过上面的 5 个步骤之后，最后我们只需使能计数器开始计数，外部时钟模式 1 的配置就算完成。使能计数器由 TIMx_CR1 的位 CEN 配置。

30.2.1.3 外部时钟模式 2

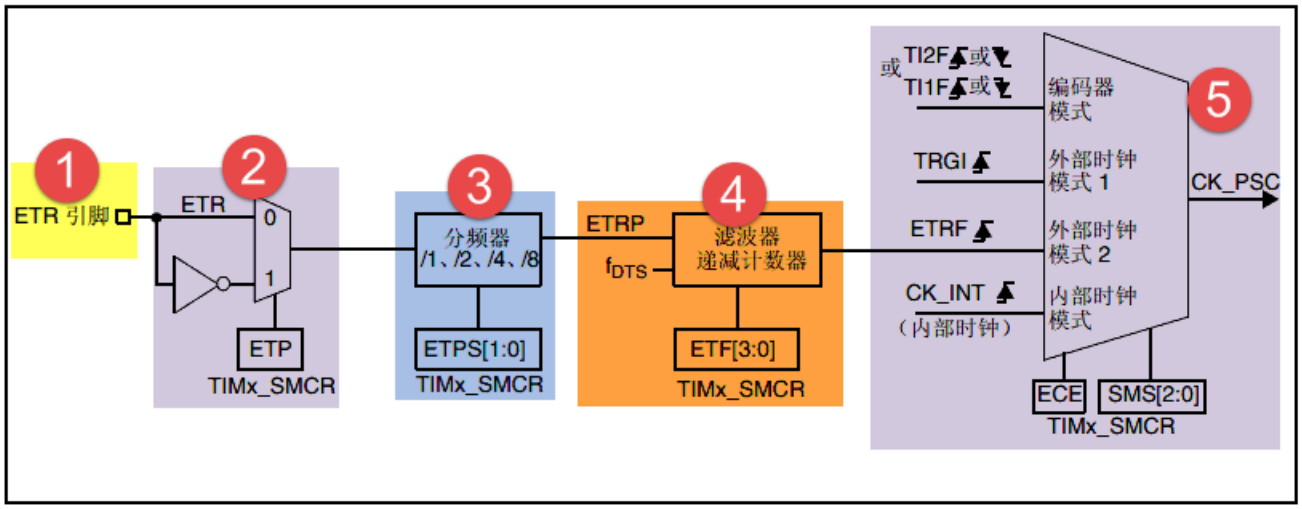


图 31-3 外部时钟模式 2 框图

□：时钟信号输入引脚

当使用外部时钟模式 2 的时候，时钟信号来自于定时器的特定输入通道 TIMx_ETR，只有 1 个。

□：外部触发极性

来自 ETR 引脚输入的信号可以选择为上升沿或者下降沿有效，具体的由 TIMx_SMCR 的位 ETP 配置。

□：外部触发预分频器

由于 ETRP 的信号的频率不能超过 TIMx_CLK（72M）的 1/4，当触发信号的频率很高的情况下，就必须使用分频器来降频，具体的由 TIMx_SMCR 的位 ETPS[1:0] 配置。

□：滤波器

如果 ETRP 的信号的频率过高或者混杂有高频干扰信号的话，我们就需要使用滤波器对 ETRP 信号重新采样，来达到降频或者去除高频干扰的目的。具体的由 TIMx_SMCR 的位 ETF[3:0] 配置，其中的 f_{DTS} 是由内部时钟 CK_INT 分频得到，具体的由 TIMx_CR1 的位 CKD[1:0] 配置。

□：从模式选择

经过滤波器滤波的信号连接到 ETRF 引脚后，触发信号成为外部时钟模式 2 的输入，最终等于 CK_PSC，然后驱动计数器 CNT 计数。具体的配置 TIMx_SMCR 的位 ECE 为 1 即可选择外部时钟模式 2。

□：使能计数器

经过上面的 5 个步骤之后，最后我们只需使能计数器开始计数，外部时钟模式 2 的配置就算完成。使能计数器由 TIMx_CR1 的位 CEN 配置。

30.2.1.4 内部触发输入

内部触发输入是使用一个定时器作为另一个定时器的预分频器。硬件上高级控制定时器和通用定时器在内部连接在一起，可以实现定时器同步或级联。主模式的定时器可以对从模式定时器执行复位、启动、停止或提供时钟。

30.2.2 控制器

高级控制定时器控制器部分包括触发控制器、从模式控制器以及编码器接口。触发控制器用来针对片内外设输出触发信号，比如为其它定时器提供时钟和触发 DAC/ADC 转换。

编码器接口专门针对编码器计数而设计。从模式控制器可以控制计数器复位、启动、递增/递减、计数。有关控制器部分只需熟练阅读寄存器描述即可。

30.2.3 时基单元

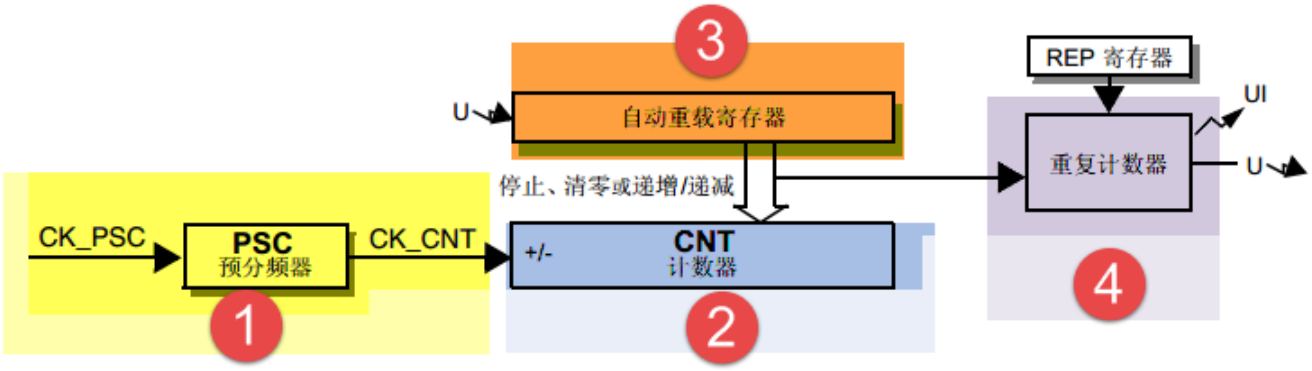


图 31-4 高级定时器时基单元

高级控制定时器时基单元功能包括四个寄存器，分别是计数器寄存器 (CNT)、预分频器寄存器 (PSC)、自动重载寄存器 (ARR) 和重复计数器寄存器 (RCR)。其中重复计数器 RCR 是高级定时器独有，通用和基本定时器没有。前面三个寄存器都是 16 位有效，TIMx_RCR 寄存器是 8 位有效。

30.2.3.1 预分频器 PSC

预分频器 PSC，有一个输入时钟 CK_PSC 和一个输出时钟 CK_CNT。输入时钟 CK_PSC 就是上面时钟源的输出，输出 CK_CNT 则用来驱动计数器 CNT 计数。通过设置预分频器 PSC 的值可以得到不同的 CK_CNT，实际计算为：f_{CK_CNT} 等于 f_{CK_PSC}/(PSC[15:0]+1)，可以实现 1 至 65536 分频。

30.2.3.2 计数器 CNT

高级控制定时器的计数器有三种计数模式，分别为递增计数模式、递减计数模式和递增/递减 (中心对齐) 计数模式。

(1) 递增计数模式下，计数器从 0 开始计数，每来一个 CK_CNT 脉冲计数器就增加 1，直到计数器的值与自动重载寄存器 ARR 值相等，然后计数器又从 0 开始计数并生成计数器上溢事件，计数器总是如此循环计数。如果禁用重复计数器，在计数器生成上溢事件就马上生成更新事件 (UEV)；如果使能重复计数器，每生成一次上溢事件重复计数器内容就减 1，直到重复计数器内容为 0 时才会生成更新事件。

(2) 递减计数模式下，计数器从自动重载寄存器 ARR 值开始计数，每来一个 CK_CNT 脉冲计数器就减 1，直到计数器值为 0，然后计数器又从自动重载寄存器 ARR 值开始递减计数并生成计数器下溢事件，计数器总是如此循环计数。如果禁用重复计数器，在计数器生成下溢事件就马上生成更新事件；如果使能重复计数器，每生成一次下溢事件重复计数器内容就减 1，直到重复计数器内容为 0 时才会生成更新事件。

(3) 中心对齐模式下，计数器从 0 开始递增计数，直到计数值等于 (ARR-1) 值生成计数器上溢事件，然后从 ARR 值开始递减计数直到 1 生成计数器下溢事件。然后又从 0 开始计数，如此循环。每次发生计数器上溢和下溢事件都会生成更新事件。

30.2.3.3 自动重载寄存器 ARR

自动重载寄存器 ARR 用来存放与计数器 CNT 比较的值，如果两个值相等就递减重复计数器。可以通过 TIMx_CR1 寄存器的 ARPE 位控制自动重载影子寄存器功能，如果 ARPE 位置 1，自动重载影子寄存器有效，只有在事件更新时才把 TIMx_ARR 值赋给影子寄存器。如果 ARPE 位为 0，则修改 TIMx_ARR 值马上有效。

30.2.3.4 重复计数器 RCR

在基本/通用定时器发生上/下溢事件时直接就生成更新事件，但对于高级控制定时器却不是这样，高级控制定时器在硬件结构上多出了重复计数器，在定时器发生上溢或下溢事件是递减重复计数器的值，只有当重复计数器为 0 时才会生成更新事件。在发生 N+1 个上溢或下溢事件 (N 为 RCR 的值) 时产生更新事件。

30.2.4 □ 输入捕获

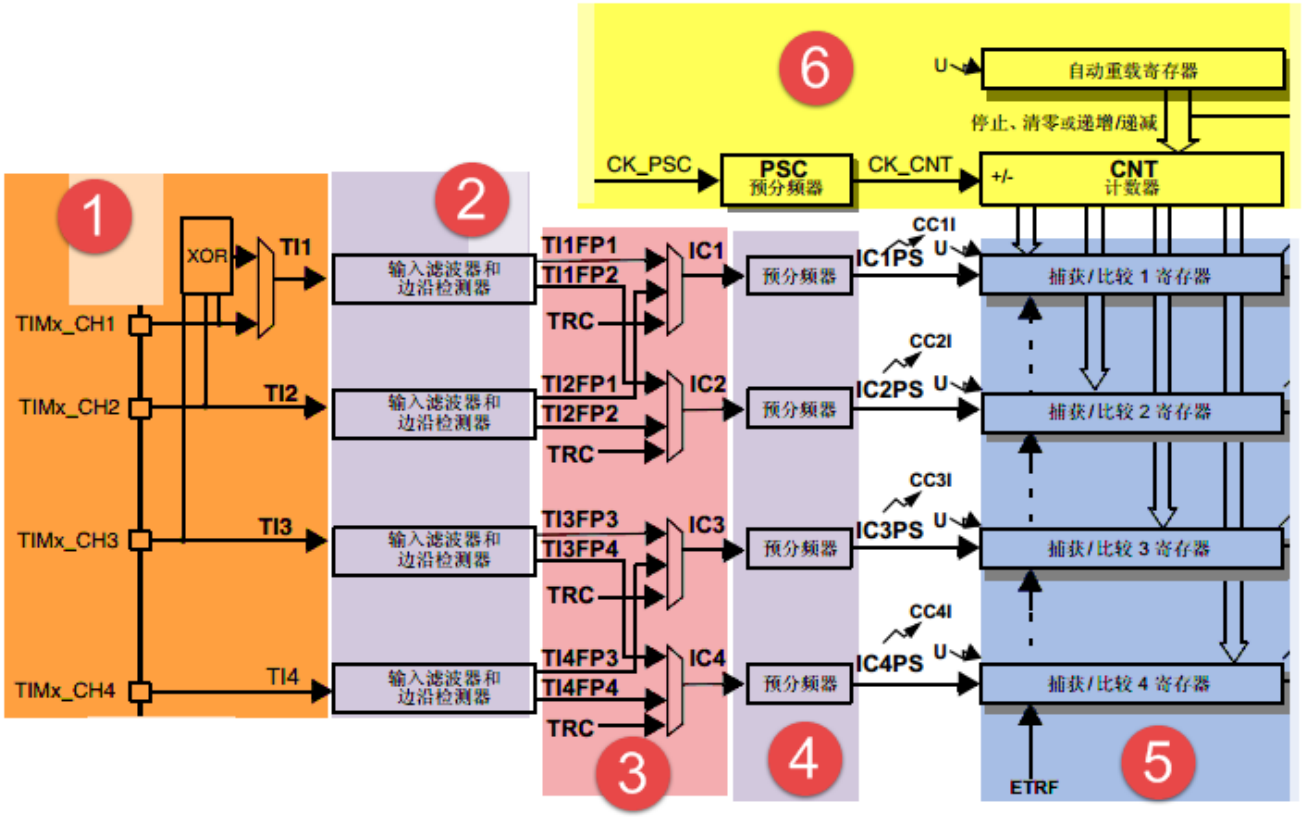


图 31-5 输入捕获功能框图

输入捕获可以对输入的信号的上升沿、下降沿或者双边沿进行捕获，常用的有测量输入信号的脉宽和测量 PWM 输入信号的频率和占空比这两种。

输入捕获的大概的原理就是，当捕获到信号的跳变沿的时候，把计数器 CNT 的值锁存到捕获寄

存器 CCR 中，把前后两次捕获到的 CCR 寄存器中的值相减，就可以算出脉宽或者频率。如果捕获的脉宽的时间长度超过你的捕获定时器的周期，就会发生溢出，这个我们需要做额外的处理。

30.2.4.1 输入通道

需要被测量的信号从定时器的外部引脚 TIMx_CH1/2/3/4 进入，通常叫 TI1/2/3/4，在后面的捕获讲解中对于要被测量的信号我们都以 TIx 为标准叫法。

30.2.4.2 输入滤波器和边沿检测器

当输入的信号存在高频干扰的时候，我们需要对输入信号进行滤波，即进行重新采样，根据采样定律，采样的频率必须大于等于两倍的输入信号。比如输入的信号为 1M，又存在高频的信号干扰，那么此时就很有必要进行滤波，我们可以设置采样频率为 2M，这样可以在保证采样到有效信号的基础上把高于 2M 的高频干扰信号过滤掉。

滤波器的配置由 CR1 寄存器的位 CKD[1:0] 和 CCMR1/2 的位 ICxF[3:0] 控制。从 ICxF 位的描述可知，采样频率 f_{SAMPLE} 可以由 f_{CK_INT} 和 f_{DTS} 分频后的时钟提供，其中是 f_{CK_INT} 内部时钟，f_{DTS} 是 f_{CK_INT} 经过分频后得到的频率，分频因子由 CKD[1:0] 决定，可以是不分频，2 分频或者是 4 分频。

边沿检测器用来设置信号在捕获的时候是什么边沿有效，可以是上升沿，下降沿，或者是双边沿，具体的由 CCER 寄存器的位 CCxP 和 CCxNP 决定。

30.2.4.3 捕获通道

捕获通道就是图中的 IC1/2/3/4，每个捕获通道都有相对应的捕获寄存器 CCR1/2/3/4，当发生捕获的时候，计数器 CNT 的值就会被锁存到捕获寄存器中。

这里我们要搞清楚输入通道和捕获通道的区别，输入通道是用来输入信号的，捕获通道是用来捕获输入信号的通道，一个输入通道的信号可以同时输入给两个捕获通道。比如输入通道 TI1 的信号经过滤波边沿检测器之后的 TI1FP1 和 TI1FP2 可以进入到捕获通道 IC1 和 IC2，其实这就是我们后面要讲的 PWM 输入捕获，只有一路输入信号（TI1）却占用了两个捕获通道（IC1 和 IC2）。

当只需要测量输入信号的脉宽时候，用一个捕获通道即可。输入通道和捕获通道的映射关系具体由寄存器 CCMRx 的位 CCxS[1:0] 配置。

30.2.4.4 预分频器

ICx 的输出信号会经过一个预分频器，用于决定发生多少个事件时进行一次捕获。具体的由寄存器 CCMRx 的位 ICxPSC 配置，如果希望捕获信号的每一个边沿，则不分频。

30.2.4.5 捕获寄存器

经过预分频器的信号 ICxPS 是最终被捕获的信号，当发生捕获时（第一次），计数器 CNT 的值会被锁存到捕获寄存器 CCR 中，还会产生 CCxI 中断，相应的中断位 CCxIF（在 SR 寄存器中）会被置位，通过软件或者读取 CCR 中的值可以将 CCxIF 清 0。如果发生第二次捕获（即重复捕获：CCR 寄存器中已捕获到计数器值且 CCxIF 标志已置 1），则捕获溢出标志位 CCxOF（在 SR 寄存器中）会被置位，CCxOF 只能通过软件清零。

30.2.5 □ 输出比较

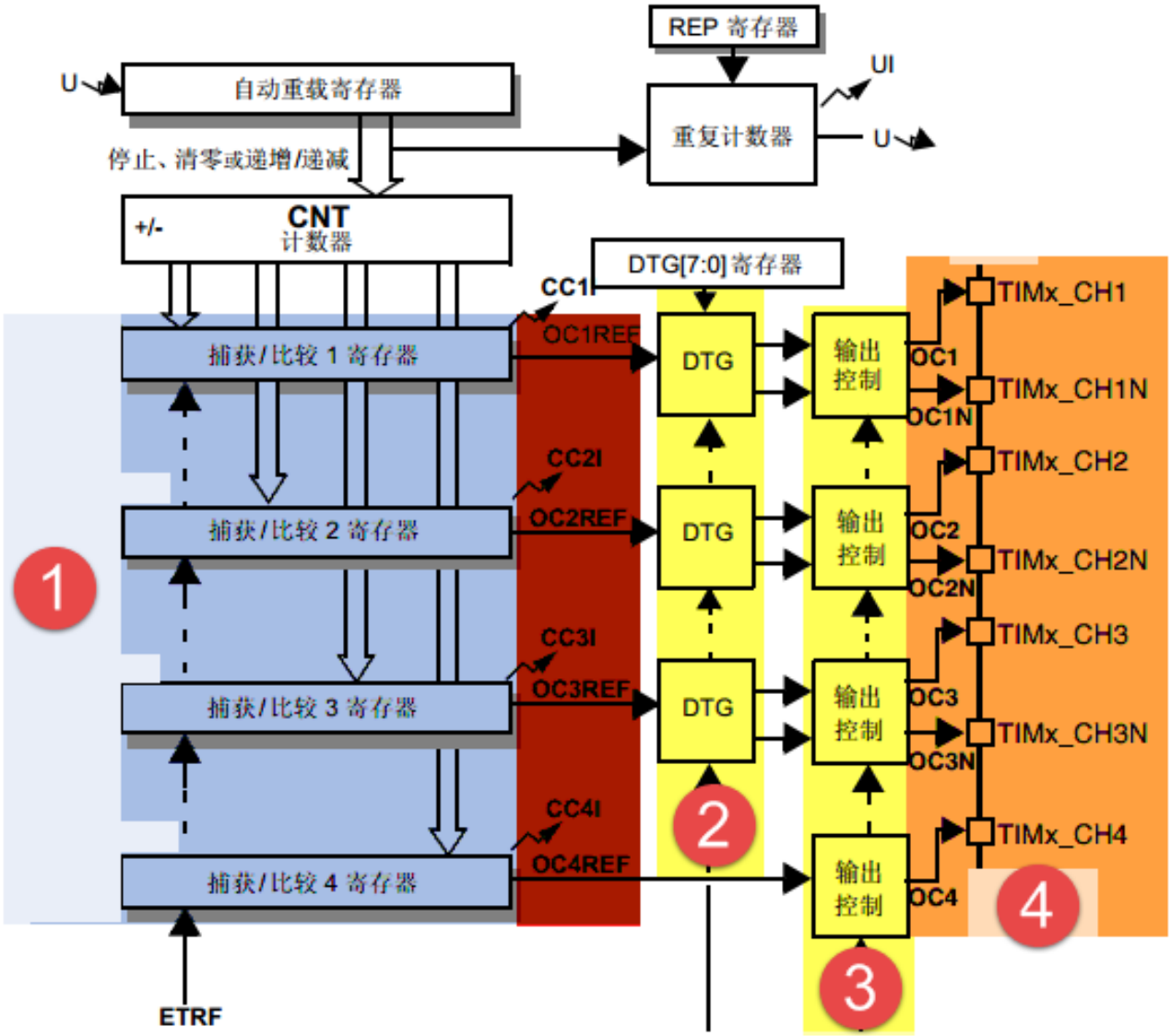


图 31-6 输出比较功能框图

输出比较就是通过定时器的外部引脚对外输出控制信号，有冻结、将通道 X（x=1,2,3,4）设置为匹配时输出有效电平、将通道 X 设置为匹配时输出无效电平、翻转、强制变为无效电平、强制变为有效电平、PWM1 和 PWM2 这八种模式，具体使用哪种模式由寄存器 CCMRx 的位 OCxM[2:0]配置。其中 PWM 模式是输出比较中的特例，使用的也最多。

30.2.5.1 比较寄存器

当计数器 CNT 的值跟比较寄存器 CCR 的值相等的时候，输出参考信号 OCxREF 的信号的极性就会改变，其中 OCxREF=1（高电平）称之为有效电平，OCxREF=0（低电平）称之为无效电平，并且会产生比较中断 CCxI，相应的标志位 CCxIF（SR 寄存器中）会置位。然后 OCxREF 再经过一系列的控制之后就成为真正的输出信号 OCx/OCxN。

30.2.5.2 死区发生器

在生成的参考波形 OCxREF 的基础上，可以插入死区时间，用于生成两路互补的输出信号 OCx 和 OCxN，死区时间的大小具体由 BDTR 寄存器的位 DTG[7:0] 配置。死区时间的大小必须根据与输出信号相连接的器件及其特性来调整。下面我们简单举例说明下带死区的 PWM 信号的应用，我们以一个板桥驱动电路为例。

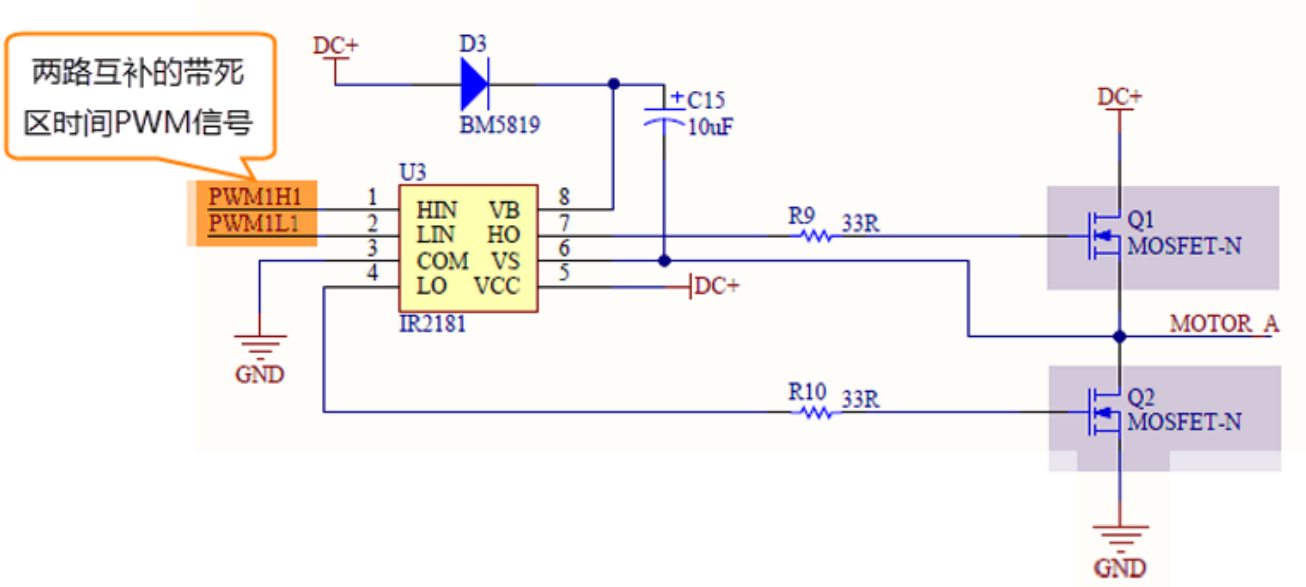


图 31-7 半桥驱动电路

在这个半桥驱动电路中，Q1 导通，Q2 截止，此时我想让 Q1 截止 Q2 导通，肯定是要先让 Q1 截止一段时间之后，再等一段时间才让 Q2 导通，那么这段等待的时间就称为死区时间，因为 Q1 关闭需要时间（由 MOS 管的工艺决定）。如果 Q1 关闭之后，马上打开 Q2，那么此时一段时间内相当于 Q1 和 Q2 都导通了，这样电路会短路。

图 *31_8* 是针对上面的半桥驱动电路而画的带死区插入的 PWM 信号，图中的死区时间要根据 MOS 管的工艺来调节。

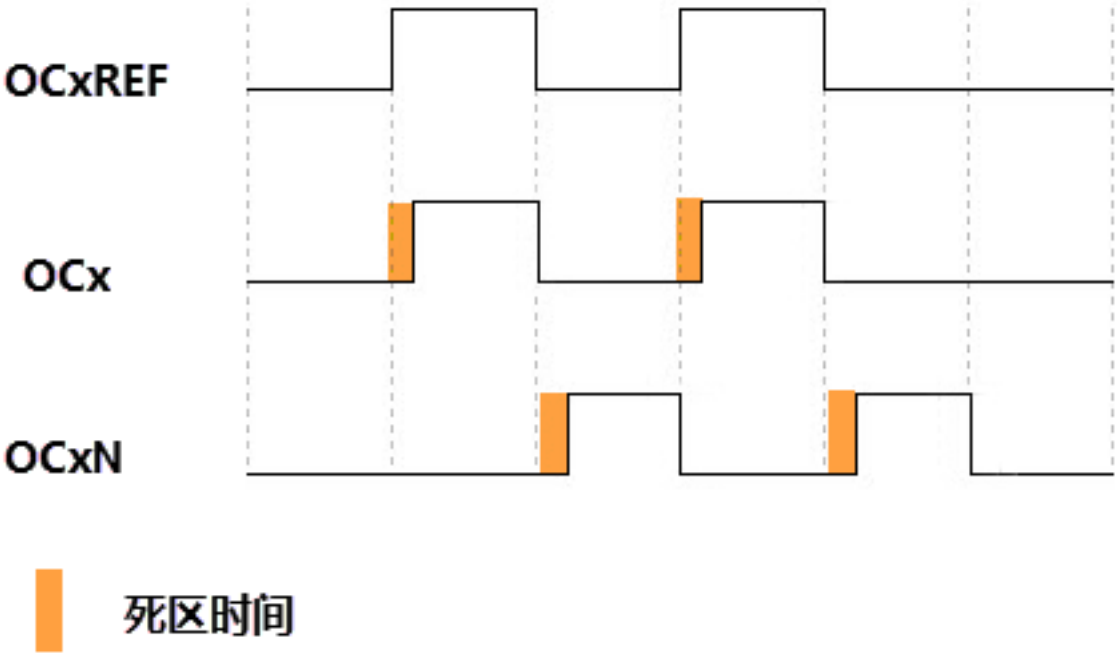


图 31-8 带死区插入的互补输出

30.2.5.3 输出控制

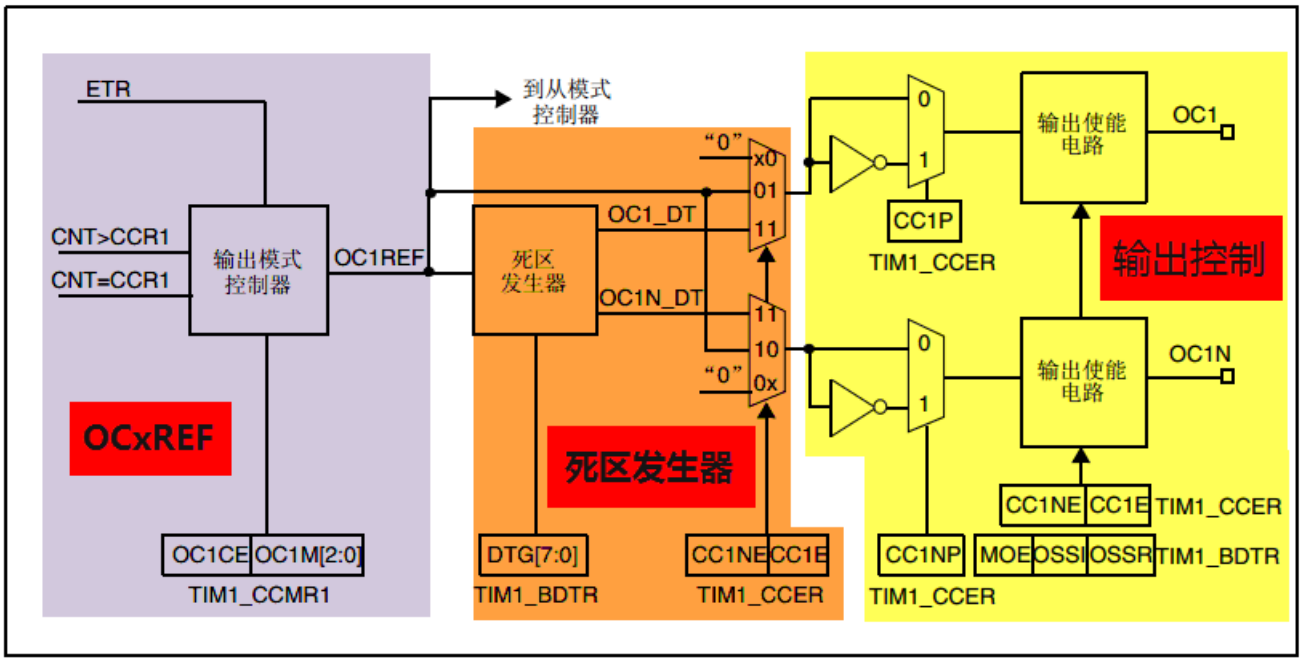


图 31-9 输出比较（通道 1~3）的输出控制框图

在输出比较的输出控制中，参考信号 OCxREF 在经过死区发生器之后会产生两路带死区的互补信号 OCx_DT 和 OCxN_DT（通道 1~3 才有互补信号，通道 4 没有，其余跟通道 1~3 一样），这两路带死区的互补信号然后就进入输出控制电路，如果没有加入死区控制，那么进入输出控制电路的信号就直接是 OCxREF。

进入输出控制电路的信号会被分成两路，一路是原始信号，一路是被反向的信号，具体的由寄存器 CCER 的位 CCxP 和 CCxNP 控制。经过极性选择的信号是否由 OCx 引脚输出到外部引脚 CHx/CHxN 则由寄存器 CCER 的位 CxE/CxNE 配置。

如果加入了断路（刹车）功能，则断路和死区寄存器 BDTR 的 MOE、OSSI 和 OSSR 这三个位会共同影响输出的信号。

30.2.5.4 输出引脚

输出比较的输出信号最终是通过定时器的外部 IO 来输出的，分别为 CH1/2/3/4，其中前面三个通道还有互补的输出通道 CH1/2/3N。更加详细的 IO 说明还请查阅相关的数据手册。

30.2.6 断路功能

断路功能就是电机控制的刹车功能，使能断路功能时，根据相关控制位状态修改输出信号电平。在任何情况下，OCx 和 OCxN 输出都不能同时为有效电平，这关系到电机控制常用的 H 桥电路结构原因。

断路源可以是时钟故障事件，由内部复位时钟控制器中的时钟安全系统 (CSS) 生成，也可以是外部断路输入 IO，两者是或运算关系。

系统复位启动都默认关闭断路功能，将断路和死区寄存器 (TIMx_BDTR) 的 BKE 为置 1，使能断路功能。可通过 TIMx_BDTR 寄存器的 BKP 位设置设置断路输入引脚的有效电平，设置为 1 时输入 BRK 为高电平有效，否则低电平有效。

发送断路时，将产生以下效果：

- TIMx_BDTR 寄存器中主输出模式使能 (MOE) 位被清零，输出处于无效、空闲或复位状态；
- 根据相关控制位状态控制输出通道引脚电平；当使能通道互补输出时，会根据情况自动控制输出通道电平；
- 将 TIMx_SR 寄存器中的 BIF 位置 1，并可产生中断和 DMA 传输请求。
- 如果 TIMx_BDTR 寄存器中的自动输出使能 (AOE) 位置 1，则 MOE 位会在发生下一个 UEV 事件时自动再次置 1。

30.3 输入捕获应用

输入捕获一般应用在两个方面，一个方面是脉冲跳变沿时间测量，另一方面是 PWM 输入测量。

30.3.1 测量脉宽或者频率

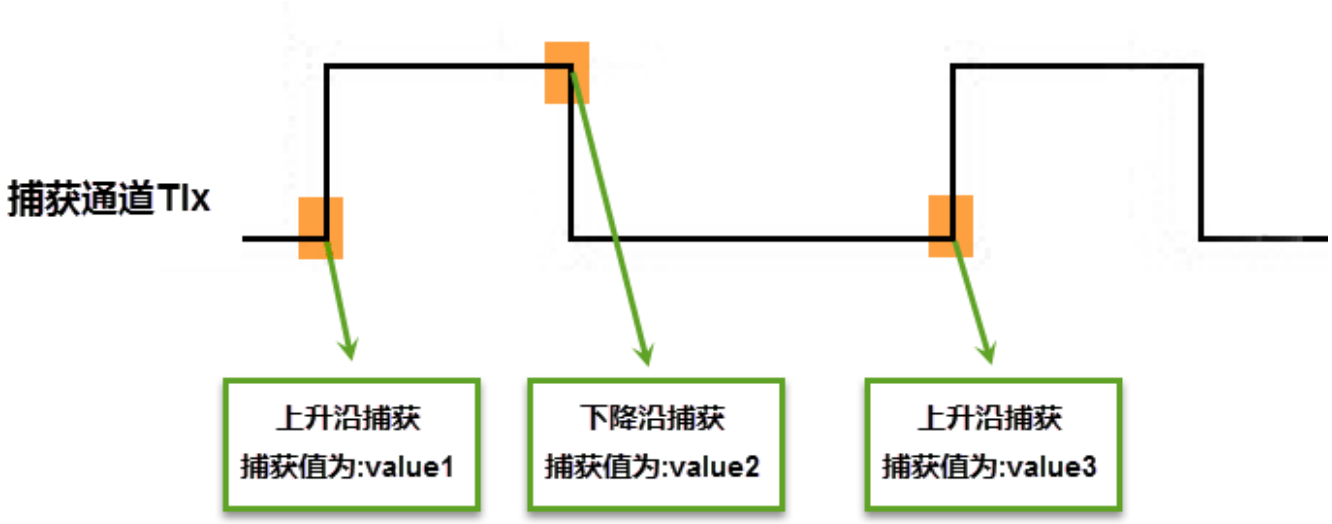


图 31-10 脉宽/频率测量示意图

30.3.1.1 测量频率

当捕获通道 TIx 上出现上升沿时，发生第一次捕获，计数器 CNT 的值会被锁存到捕获寄存器 CCR 中，而且还会进入捕获中断，在中断服务程序中记录一次捕获（可以用一个标志变量来记录），并把捕获寄存器中的值读取到 value1 中。当出现第二次上升沿时，发生第二次捕获，计数器 CNT 的值会再次被锁存到捕获寄存器 CCR 中，并再次进入捕获中断，在捕获中断中，把捕获寄存器的值读取到 value3 中，并清除捕获记录标志。利用 value3 和 value1 的差值我们就可以算出信号的周期（频率）。

30.3.1.2 测量脉宽

当捕获通道 TIx 上出现上升沿时，发生第一次捕获，计数器 CNT 的值会被锁存到捕获寄存器 CCR 中，而且还会进入捕获中断，在中断服务程序中记录一次捕获（可以用一个标志变量来记录），并把捕获寄存器中的值读取到 value1 中。然后把捕获边沿改变为下降沿捕获，目的是捕获后面的下降沿。当下降沿到来的时候，发生第二次捕获，计数器 CNT 的值会再次被锁存到捕获寄存器 CCR 中，并再次进入捕获中断，在捕获中断中，把捕获寄存器的值读取到 value3 中，并清除捕获记录标志。然后把捕获边沿设置为上升沿捕获。

在测量脉宽过程中需要来回的切换捕获边沿的极性，如果测量的脉宽时间比较长，定时器就会发生溢出，溢出的时候会产生更新中断，我们可以在中断里面对溢出进行记录处理。

30.3.2 PWM 输入模式

测量脉宽和频率还有一个更简便的方法就是使用 PWM 输入模式，该模式是输入捕获的特例，只能使用通道 1 和通道 2，通道 3 和通道 4 使用不了。与上面那种只使用一个捕获寄存器测量脉宽和频率的方法相比，PWM 输入模式需要占用两个捕获寄存器。

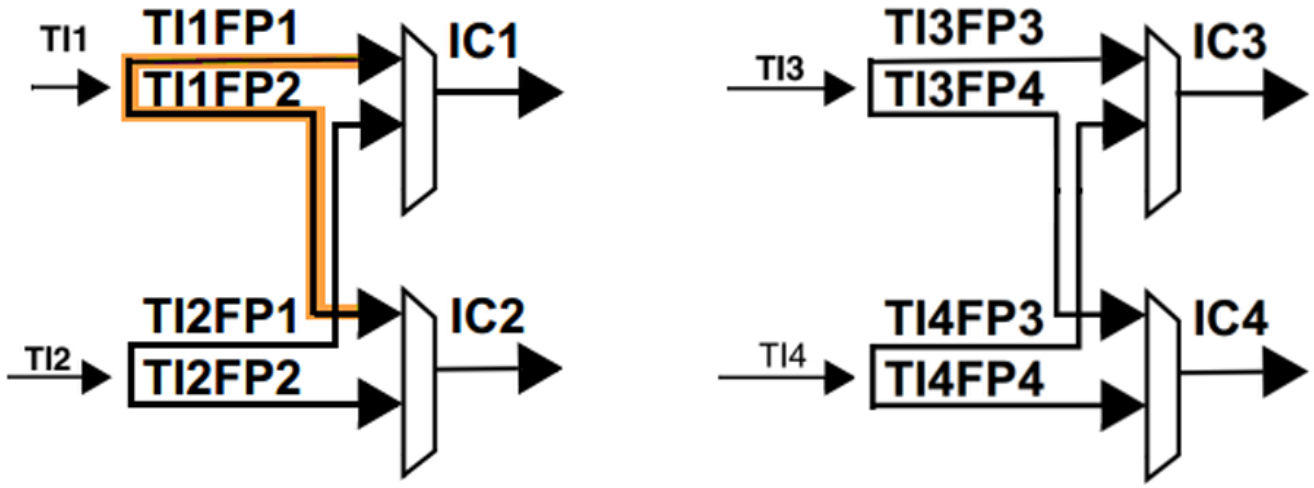


图 31-11 输入通道和捕获通道的关系映射图

当使用 PWM 输入模式的时候，因为一个输入通道 (TIx) 会占用两个捕获通道 (ICx)，所以一个定时器在使用 PWM 输入的时候最多只能使用两个输入通道 (TIx)。

我们以输入通道 TI1 工作在 PWM 输入模式为例来讲解下具体的工作原理，其他通道以此类推即可。

PWM 信号由输入通道 TI1 进入，因为是 PWM 输入模式的缘故，信号会被分为两路，一路是 TI1FP1，另外一路是 TI2FP2。其中一路是周期，另一路是占空比，具体哪一路信号对应周期还是占空比，得从程序上设置哪一路信号作为触发输入，作为触发输入的哪一路信号对应的就是周期，另一路就是对应占空比。作为触发输入的那一路信号还需要设置极性，是上升沿还是下降沿捕获，一旦设置好触发输入的极性，另外一路硬件就会自动配置为相反的极性捕获，无需软件配置。一句话概括就是：选定输入通道，确定触发信号，然后设置触发信号的极性即可，因为是 PWM 输入的缘故，另一路信号则由硬件配置，无需软件配置。

当使用 PWM 输入模式的时候必须将从模式控制器配置为复位模式（配置寄存器 SMCR 的位 SMS[2:0] 来实现），即当我们启动触发信号开始进行捕获的时候，同时把计数器 CNT 复位清零。

下面我们以一个更加具体的时序图来分析下 PWM 输入模式。

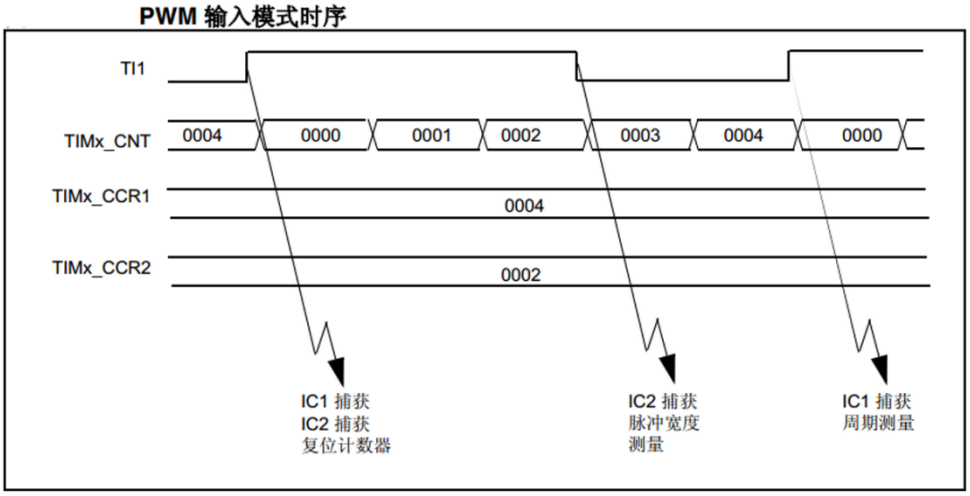


图 31-12 PWM 输入模式时序

PWM 信号由输入通道 TI1 进入，配置 TI1FP1 为触发信号，上升沿捕获。当上升沿的时候 IC1 和 IC2 同时捕获，计数器 CNT 清零，到了下降沿的时候，IC2 捕获，此时计数器 CNT 的值被锁存到捕获寄存器 CCR2 中，到了下一个上升沿的时候，IC1 捕获，计数器 CNT 的值被锁存到捕获寄存器 CCR1 中。其中 CCR2+1 测量的是脉宽，CCR1+1 测量的是周期。这里要注意的是 CCR2 和 CCR1 的值在计算占空比和频率的时候都必须加 1，因为计数器是从 0 开始计数的。

从软件上来说，用 PWM 输入模式测量脉宽和周期更容易，付出的代价是需要占用两个捕获寄存

器。

30.4 输出比较应用

输出比较模式总共有 8 种，具体的由寄存器 CCMRx 的位 OCxM[2:0] 配置。我们这里只讲解最常用的 PWM 模式，其他几种模式具体的看数据手册即可。

30.4.1 PWM 输出模式

PWM 输出就是对外输出脉宽（即占空比）可调的方波信号，信号频率由自动重装寄存器 ARR 的值决定，占空比由比较寄存器 CCR 的值决定。

PWM 模式分为两种，PWM1 和 PWM2，总得来说是差不多，就看你怎么用而已，具体的区别见表格 31-1。

表格 31-1 PWM1 与 PWM2 模式的区别

模式	计数器 CNT 计算方式	说明
PWM1	递增	CNT<CCR，通道 CH 为有效，否则为无效
	递减	CNT>CCR，通道 CH 为无效，否则为有效
PWM2	递增	CNT<CCR，通道 CH 为无效，否则为有效
	递减	CNT>CCR，通道 CH 为有效，否则为无效

下面我们以 PWM1 模式来讲解，以计数器 CNT 计数的方向不同还分为边沿对齐模式和中心对齐模式。PWM 信号主要都是用来控制电机，一般的电机控制用的都是边沿对齐模式，FOC 电机一般用中心对齐模式。我们这里只分析这两种模式在信号感官上（即信号波形）的区别，具体在电机控制中的区别不做讨论，到了你真正需要使用的时候就会知道了。

30.4.1.1 PWM 边沿对齐模式

在递增计数模式下，计数器从 0 计数到自动重载值（TIMx_ARR 寄存器的内容），然后重新从 0 开始计数并生成计数器上溢事件

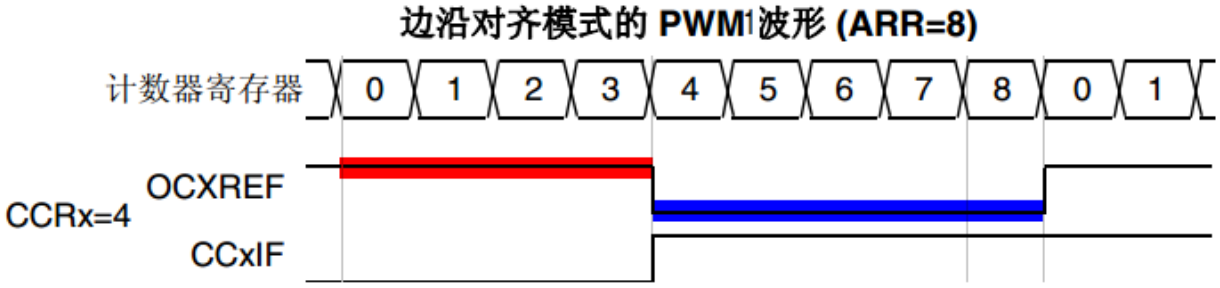


图 31-13 PWM1 模式的边沿对齐波形

在边沿对齐模式下，计数器 CNT 只工作在一种模式，递增或者递减模式。这里我们以 CNT 工作在递增模式为例，在中，ARR=8，CCR=4，CNT 从 0 开始计数，当 CNT<CCR 的值时，OCxREF 为有效的高电平，于此同时，比较中断寄存器 CCxIF 置位。当 CCR=<CNT<=ARR 时，OCxREF 为无效的低电平。然后 CNT 又从 0 开始计数并生成计数器上溢事件，以此循环往复。

30.4.1.2 PWM 中心对齐模式

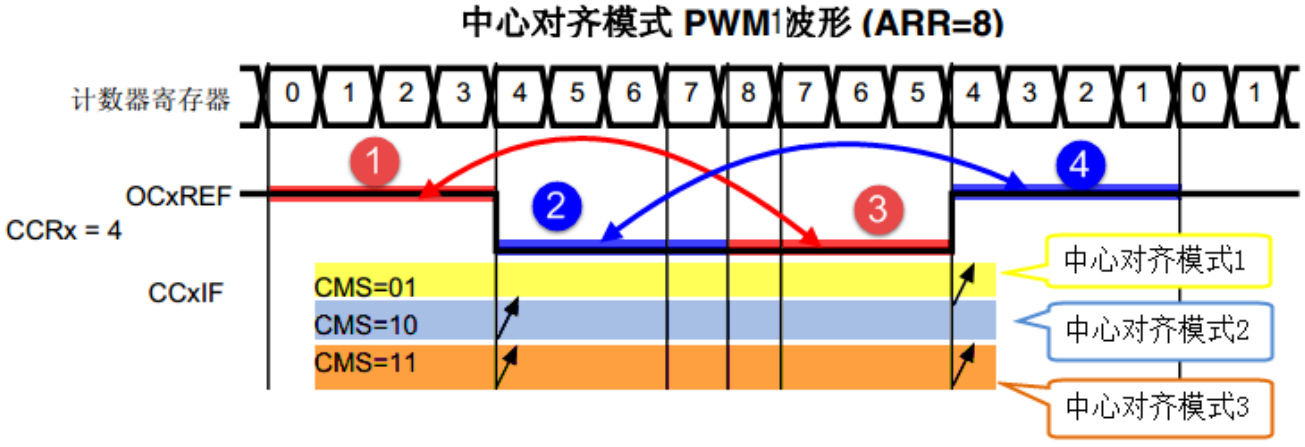


图 31-14 PWM1 模式的中心对齐波形

在中心对齐模式下，计数器 CNT 是工作做递增/递减模式下。开始的时候，计数器 CNT 从 0 开始计数到自动重载值减 1(ARR-1)，生成计数器上溢事件；然后从自动重载值开始向下计数到 1 并生成计数器下溢事件。之后从 0 开始重新计数。

图 *31_14* 是 PWM1 模式的中心对齐波形，ARR=8，CCR=4。第一阶段计数器 CNT 工作在递增模式下，从 0 开始计数，当 CNT<CCR 的值时，OCxREF 为有效的高电平，当 CCR=<CNT«ARR 时，OCxREF 为无效的低电平。第二阶段计数器 CNT 工作在递减模式，从 ARR 的值开始递减，当 CNT>CCR 时，OCxREF 为无效的低电平，当 CCR=>CNT>=1 时，OCxREF 为有效的高电平。

在波形图上我们把波形分为两个阶段，第一个阶段是计数器 CNT 工作在递增模式的波形，这个阶段我们又分为 ① 和 ② 两个阶段，第二个阶段是计数器 CNT 工作在递减模式的波形，这个阶段我们又分为 ③ 和 ④ 两个阶段。要说中心对齐模式下的波形有什么特征的话，那就是 ① 和 ③ 阶段的时间相等，② 和 ④ 阶段的时间相等。

中心对齐模式又分为中心对齐模式 1/2/3 三种，具体由寄存器 CR1 位 CMS[1:0] 配置。具体的区别就是比较中断中断标志位 CCxIF 在何时置 1：中心模式 1 在 CNT 递减计数的时候置 1，中心对齐模式 2 在 CNT 递增计数时置 1，中心模式 3 在 CNT 递增和递减计数时都置 1。

30.5 定时器初始化结构体详解

在 HAL 库函数头文件 stm32f1xx_hal_tim.h 中对定时器外设建立了四个初始化结构体，分别为时基初始化结构体 TIM_TimeBaseInitTypeDef、输出比较初始化结构体 TIM_OCInitTypeDef、输入捕获初始化结构体 TIM_ICInitTypeDef 和断路和死区初始化结构体 TIM_BDTRInitTypeDef，高级控制定时器可以用到所有初始化结构体，通用定时器不能使用 TIM_BDTRInitTypeDef 结构体，基本定时器只能使用时基结构体。接下来我们具体讲解下这四个结构体。

30.5.1 TIM_TimeBaseInitTypeDef

时基结构体 TIM_TimeBaseInitTypeDef 用于定时器基础参数设置，与 TIM_TimeBaseInit 函数配合使用完成配置。

列表 1: 代码清单 31-1 定时器基本初始化结构体

```
typedef struct {
    uint32_t Prescaler;           // 预分频器
    uint32_t CounterMode;         // 计数模式
    uint32_t Period;              // 定时器周期
    uint32_t ClockDivision;       // 时钟分频
    uint32_t RepetitionCounter;    // 重复计算器
} TIM_TimeBaseInitTypeDef;
```

(1) Prescaler：定时器预分频器设置，时钟源经该预分频器才是定时器计数时钟 CK_CNT，它设定 PSC 寄存器的值。计算公式为：计数器时钟频率 (f:sub:*CK_CNT*) 等于 f_{CK_PSC} / (PSC[15:0] + 1)，可实现 1 至 65536 分频。

(2) CounterMode：定时器计数方式，可设置为向上计数、向下计数以及中心对齐。高级控制定时器允许选择任意一种。

(3) Period：定时器周期，实际就是设定自动重载寄存器 ARR 的值，ARR 为要装载到实际自动重载寄存器（即影子寄存器）的值，可设置范围为 0 至 65535。

(4) ClockDivision：时钟分频，设置定时器时钟 CK_INT 频率与死区发生器以及数字滤波器采样时钟频率分频比。可以选择 1、2、4 分频。

(5) RepetitionCounter：重复计数器，只有 8 位，只存在于高级定时器。

30.5.2 TIM_OC_InitTypeDef

输出比较结构体 TIM_OC_InitTypeDef 用于输出比较模式，与 TIM_OCxInit 函数配合使用完成指定定时器输出通道初始化配置。高级控制定时器有四个定时器通道，使用时都必须单独设置。

列表 2: 代码清单 31-2 定时器比较输出初始化结构体

```
typedef struct {
    uint32_t OCMode;                    // 比较输出模式
    uint32_t Pulse;                     // 脉冲宽度
    uint32_t OCPolarity;      // 输出极性
    uint32_t OCNPolarity;       // 互补输出极性
    uint32_t OCFastMode;                // 比较输出模式快速使能
    uint32_t OCIdleState;               // 空闲状态下比较输出状态
    uint32_t OCNIdleState;              // 空闲状态下比较互补输出状态
} TIM_OCInitTypeDef;
```

(1) OCMode：比较输出模式选择，总共有八种，常用的为 PWM1/PWM2。它设定 CCMRx 寄存器 OCxM[2:0] 位的值。

(2) Pulse：比较输出脉冲宽度，实际设定比较寄存器 CCR 的值，决定脉冲宽度。可设置范围为 0 至 65535。

(3) OCPolarity：比较输出极性，可选 OCx 为高电平有效或低电平有效。它决定着定时器通道有效电平。它设定 CCER 寄存器的 CCxP 位的值。

(4) OCNPolarity：比较互补输出极性，可选 OCxN 为高电平有效或低电平有效。它设定 TIMx_CCER 寄存器的 CCxNP 位的值。

(5) OCFastMode：比较输出模式快速使能。它设定 TIMx_CCMR 寄存器的，OCxFE 位的值可以快速使能或者禁能输出。

(6) OCIdleState：空闲状态时通道输出电平设置，可选输出 1 或输出 0，即在空闲状态 (BDTR_MOE 位为 0) 时，经过死区时间后定时器通道输出高电平或低电平。它设定 CR2 寄存器的 OISx 位的值。

(7) OCNIdleState：空闲状态时互补通道输出电平设置，可选输出 1 或输出 0，即在空闲状态

(BDTR_MOE 位为 0) 时，经过死区时间后定时器互补通道输出高电平或低电平，设定值必须与 OCIdleState 相反。它设定是 CR2 寄存器的 OISxN 位的值。

30.5.3 TIM_IC_InitTypeDef

输入捕获结构体 TIM_IC_InitTypeDef 用于输入捕获模式，与 HAL_TIM_IC_ConfigChannel 函数配合使用完成定时器输入通道初始化配置。如果使用 PWM 输入模式需要与 HAL_TIM_PWM_ConfigChannel 函数配合使用完成定时器输入通道初始化配置。

列表 3: 代码清单 31-3 定时器输入捕获初始化结构体

```
typedef struct {
    uint32_t ICPolarity;    // 输入捕获触发选择
    uint32_t ICSelection;   // 输入捕获选择
    uint32_t ICPrescaler;   // 输入捕获预分频器
    uint32_t ICFilter;      // 输入捕获滤波器
} TIM_IC_InitTypeDef;
```

(1) ICPolarity：输入捕获边沿触发选择，可选上升沿触发、下降沿触发或边沿跳变触发。它设定 CCER 寄存器 CCxP 位和 CCxNP 位的值。

(2) ICSelection：输入通道选择，捕获通道 ICx 的信号可来自三个输入通道，分别为 TIM_ICSELECTION_DIRECTTI、TIM_ICSELECTION_INDIRECTTI 或 TIM_ICSELECTION_TRC，具体的区别见图 31-16。它设定 CCRMx 寄存器的 CCxS[1:0] 位的值。

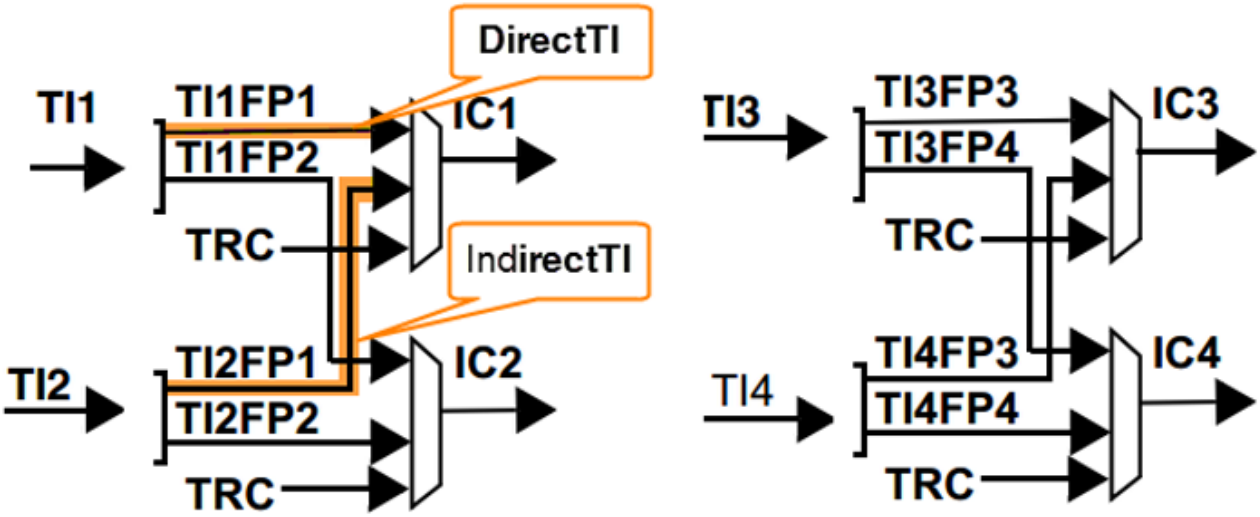


图 31-15 输入通道与捕获通道 IC 的映射图

(3) ICPrescaler：输入捕获通道预分频器，可设置 1、2、4、8 分频，它设定 CCMRx 寄存器的 ICxPSC[1:0] 位的值。如果需要捕获输入信号的每个有效边沿，则设置 1 分频即可。

(4) ICFilter：输入捕获滤波器设置，可选设置 0x0 至 0x0F。它设定 CCMRx 寄存器 ICxF[3:0] 位的值。一般我们不使用滤波器，即设置为 0。

30.5.4 TIM_BreakDeadTimeConfigTypeDef

断路和死区结构体 TIM_BreakDeadTimeConfigTypeDef 用于断路和死区参数的设置，属于高级定时器专用，用于配置断路时通道输出状态，以及死区时间。它与 HAL_TIMEx_ConfigBreakDeadTime 函数配置使用完成参数配置。这个结构体的成员只对应 BDTR 这个寄存器，有关成员的具体使用配置请参考手册 BDTR 寄存器的详细描述。

列表 4: 代码清单 31-4 断路和死区初始化结构体

```
typedef struct {
    uint32_t OffStateRunMode;          // 运行模式下的关闭状态选择
    uint32_t OffStateIDLEMode;         // 空闲模式下的关闭状态选择
    uint32_t LockLevel;               // 锁定配置
    uint32_t DeadTime;                // 死区时间
    uint32_t BreakState;               // 断路输入使能控制
    uint32_t BreakPolarity;           // 断路输入极性
    uint32_t BreakFilter;                    // 断路输入滤波器
    uint32_t Break2State;                    // 断路 2 输入使能控制
    uint32_t Break2Polarity;          // 断路 2 输入极性
    uint32_t Break2Filter;                   // 断路 2 输入滤波器
    uint32_t AutomaticOutput;          // 自动输出使能
} TIM_BreakDeadTimeConfigTypeDef;
```

(1) OffStateRunMode：运行模式下的关闭状态选择，它设定 BDTR 寄存器 OSSR 位的值。

(2) OffStateIDLEMode：空闲模式下的关闭状态选择，它设定 BDTR 寄存器 OSSI 位的值。

(3) LockLevel：锁定级别配置，BDTR 寄存器 LOCK[1:0] 位的值。

(4) DeadTime：配置死区发生器，定义死区持续时间，可选设置范围为 0x0 至 0xFF。它设定 BDTR 寄存器 DTG[7:0] 位的值。

(5) BreakState：断路输入功能选择，可选使能或禁止。它设定 BDTR 寄存器 BKE 位的值。

(6) BreakPolarity：断路输入通道 BRK 极性选择，可选高电平有效或低电平有效。它设定 BDTR 寄存器 BKP 位的值。

(7) BreakFilter：断路输入滤波器，定义 BRK 输入的采样频率和适用于 BRK 的数字滤波器带宽。它设定 BDTR 寄存器 BKF[3:0] 位的值。

(8) Break2State：断路 2 输入功能选择，可选使能或禁止。它设定 BDTR 寄存器 BK2E 位的值。

(9) Break2Polarity：断路 2 输入通道 BRK2 极性选择，可选高电平有效或低电平有效。它设定 BDTR 寄存器 BK2P 位的值。

(10) Break2Filter：断路 2 输入滤波器，定义 BRK2 输入的采样频率和适用于 BRK2 的数字滤波器带宽。它设定 BDTR 寄存器 BK2F[3:0] 位的值。

(11) AutomaticOutput：自动输出使能，可选使能或禁止，它设定 BDTR 寄存器 AOE 位的值。

30.6 PWM 互补输出实验

输出比较模式比较多，这里我们以 PWM 输出为例讲解，并通过示波器来观察波形。实验中不仅在主输出通道输出波形，还在互补通道输出与主通道互补的的波形，并且添加了断路和死区功能。

30.6.1 硬件设计

根据开发板引脚使用情况，并且参考表 31-1 中定时器引脚信息，使用高级定时器 TIM1 的通道 1 及其互补通道作为本实验的波形输出通道，对应选择 PA8 和 PB13 引脚。将示波器的两个输入通道分别与 PA8 和 PB13 引脚连接，用于观察波形，还有注意共地。

为增加断路功能，需要用到 TIM1_BKIN 引脚，这里选择 PB12 引脚。程序我们设置该引脚为高电平有效，当 BKIN 引脚被置高低电平的时候，两路互补的 PWM 输出就被停止，就好像是刹车一样。

30.6.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_AdvanceTim.c 和 bsp_AdvanceTim.h 文件用来存定时器驱动程序及相关宏定义。

30.6.2.1 编程要点

(1) 定时器用到的 GP IO 初始化

(2) 定时器时基结构体 TIM_TimeBaseInitTypeDef 初始化

(3) 定时器输出比较结构体 TIM_OCInitTypeDef 初始化

(4) 定时器刹车和死区结构体 TIM_BDTRInitTypeDef 初始化

30.6.2.2 软件分析

30.6.2.2.1 宏定义

列表 5: 代码清单 31-5 宏定义

```
/* 定时器 */
#define ADVANCED_TIMx                           TIM1
#define ADVANCED_TIM_RCC_CLK_ENABLE()           __HAL_RCC_TIM1_CLK_ENABLE()
#define ADVANCED_TIM_RCC_CLK_DISABLE()          __HAL_RCC_TIM1_CLK_DISABLE()
#define ADVANCED_TIM_GPIO_RCC_CLK_ENABLE()      {
                                                  __HAL_RCC_GPIOA_CLK_ENABLE();
                                                  __HAL_RCC_GPIOC_CLK_ENABLE();}

/* TIM8 通道 1 输出引脚 */
#define ADVANCE_OCPWM_PIN                       GPIO_PIN_6
#define ADVANCE_OCPWM_GPIO_PORT                 GPIOC
```

(下页继续)

(续上页)

```
/* TIM8 通道 1 互补输出引脚 */
#define ADVANCE_OCNPWM_PIN                GPIO_PIN_5
#define ADVANCE_OCNPWM_GPIO_PORT          GPIOA
```

使用宏定义非常方便程序升级、移植。有关每个宏的具体含义看程序注释即可。

30.6.2.2.2 定时器复用功能引脚初始化

列表 6: 代码清单 31-6 定时器复用功能引脚初始化

```
void HAL_TIM_MspPostInit(TIM_HandleTypeDef* htim)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* 定时器通道功能引脚端口时钟使能 */
    ADVANCED_TIM_GPIO_RCC_CLK_ENABLE();

    /* 定时器通道 1 功能引脚 IO 初始化 */
    GPIO_InitStruct.Pin =  ADVANCE_OCPWM_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(ADVANCE_OCPWM_GPIO_PORT, &GPIO_InitStruct);
    /* 定时器通道 1 互补通道功能引脚 IO 初始化 */
    GPIO_InitStruct.Pin = ADVANCE_OCNPWM_PIN;
    HAL_GPIO_Init(ADVANCE_OCNPWM_GPIO_PORT, &GPIO_InitStruct);
}
```

HAL_TIM_MspPostInit() 函数初始化了定时器用到的相关的 GPIO，当使用不同的 GP IO 的时候，只需要修改头文件里面的宏定义即可，而不需要修改这个函数。

30.6.2.2.3 定时器模式配置

列表 7: 代码清单 31-7 定时器模式配置

```
void ADVANCE_TIM_Mode_Config(void)
{
    TIM_ClockConfigTypeDef Clock_Config;
    TIM_MasterConfigTypeDef Master_Config;
    TIM_BreakDeadTimeConfigTypeDef BreakDeadTime_Config;
    TIM_OC_InitTypeDef OC_Config;

    TIM_TimeBaseStructure.Instance = ADVANCED_TIMx;
    TIM_TimeBaseStructure.Init.Prescaler = ADVANCED_TIM_PRESCALER;
    TIM_TimeBaseStructure.Init.CounterMode = TIM_COUNTERMODE_UP;
    TIM_TimeBaseStructure.Init.Period = ADVANCED_TIM_PERIOD;
    TIM_TimeBaseStructure.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    TIM_TimeBaseStructure.Init.RepetitionCounter = ADVANCED_TIM_
↪REPETITIONCOUNTER;
    HAL_TIM_Base_Init(&TIM_TimeBaseStructure);

    Clock_Config.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    HAL_TIM_ConfigClockSource(&TIM_TimeBaseStructure, &Clock_Config);

    HAL_TIM_PWM_Init(&TIM_TimeBaseStructure);

    Master_Config.MasterOutputTrigger = TIM_TRGO_RESET;
    Master_Config.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&TIM_TimeBaseStructure,
                                          &Master_Config);

    BreakDeadTime_Config.OffStateRunMode = TIM_OSSR_DISABLE;
    BreakDeadTime_Config.OffStateIDLEMode = TIM_OSSI_DISABLE;
    BreakDeadTime_Config.LockLevel = TIM_LOCKLEVEL_OFF;
```

(下页继续)

(续上页)

```
    BreakDeadTime_Config.DeadTime = 0;
    BreakDeadTime_Config.BreakState = TIM_BREAK_DISABLE;
    BreakDeadTime_Config.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
    BreakDeadTime_Config.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
    HAL_TIMEx_ConfigBreakDeadTime(&TIM_TimeBaseStructure,
                                  &BreakDeadTime_Config);

    OC_Config.OCMode = TIM_OCMODE_PWM1;
    OC_Config.Pulse = 900;
    OC_Config.OCPolarity = TIM_OCPOLARITY_HIGH;
    OC_Config.OCNPolarity = TIM_OCNPOLARITY_LOW;
    OC_Config.OCFastMode = TIM_OCFAST_DISABLE;
    OC_Config.OCIdleState = TIM_OCIDLESTATE_RESET;
    OC_Config.OCNIdleState = TIM_OCNIDLESTATE_RESET;

    HAL_TIM_PWM_ConfigChannel(&TIM_TimeBaseStructure, &OC_Config,
                              TIM_CHANNEL_1);
}
```

ADVANCE_TIM_Mode_Config() 函数中初始化了三个结构体，有关这三个结构体成员的具体含义可参考“定时器初始化结构体详解”小节，剩下的程序参考注释阅读即可。如果需要修改 PWM 的周期和占空比，修改头文件里面的 ADVANCE_TIM_PERIOD、ADVANCE_TIM_PSC 和 ADVANCE_TIM_PULSE 这三个宏即可。PWM 信号的频率的计算公司为：F = TIM_CLK/{(ARR+1)*(PSC+1)}，其中 TIM_CLK 等于 72MHZ，ARR 即自动重装载寄存器的值，对应 ADVANCE_TIM_PERIOD 这个宏，PSC 即计数器时钟的分频因子，对应 ADVANCE_TIM_PSC 这个宏。

30.6.2.2.4 主函数

列表 8: 代码清单 31-8 main 函数

```
int main(void)
{
    /* 初始化系统时钟为 72MHz */
    SystemClock_Config();
    /* 初始化 LED */
    DEBUG_USART_Config();

    printf("\r\nPWM 波输出实验\r\n");

    ADVANCE_TIM_Mode_Config();
    /* 启动通道 PWM 输出 */
    HAL_TIM_PWM_Start(&TIM_TimeBaseStructure,TIM_CHANNEL_1);
    /* 启动定时器互补通道 PWM 输出 */
    HAL_TIMEx_PWMN_Start(&TIM_TimeBaseStructure,TIM_CHANNEL_1);
    /* 无限循环 */
    while (1) {
    }
}
```

Main 函数很简单，调用了 ADVANCE_TIM_Init() 函数，该函数调用了 ADVANCE_TIM_GPIO_Config() 和 ADVANCE_TIM_Mode_Config() 这两个函数完成了定时器 GPIO 引脚和工作模式的初始化，这时，相应的 GPIO 引脚上就可以检测到互补输出的 PWM 信号，而且带死区时间，如果程序运行的过程中，BKIN 引脚被拉高的话，PWM 输出会被禁止，就好像是断路或者刹车一样。

30.6.2.3 下载验证

根据实验的硬件设计内容接好示波器输入通道和开发板引脚连接，，编译实验程序并下载到开发板上，调整示波器到合适参数，在示波器显示屏和看到一路互补的带死区时间的 PWM 波形，参考图 *31_16*。至于图中的信号有毛刺是因为信号的输出引脚还接了其他的芯片，受到了影响。

图 31-16 PWM 互补带死区时间波形输出

当 BKIN 引脚接高电平时，PWM 输出被禁止，就好像是刹车一样，具体见图 *31_17*。

图 31-17 PWM 刹车输出

30.7 脉宽测量输入捕获实验

上一节我们讲了输出比较，这一节我们讲输入捕获。输入捕获有常见的测量脉宽和特殊的 PWM 输入，这节我们先讲测量一个信号的脉宽，下一节再讲 PWM 输入。

30.7.1 硬件设计

根据开发板引脚使用情况，我们选用通用定时器 TIM5 的 CH1，就 PA0 这个 GPIO 来测量信号的脉宽。在开发板中 PA0 接的是一个按键，默认接 GND，当按键按下的时候 IO 口会被拉高，这个时候我们可以利用定时器的输入捕获功能来测量按键按下的这段高电平的时间，按键的具体原理图见图 *31_18*。

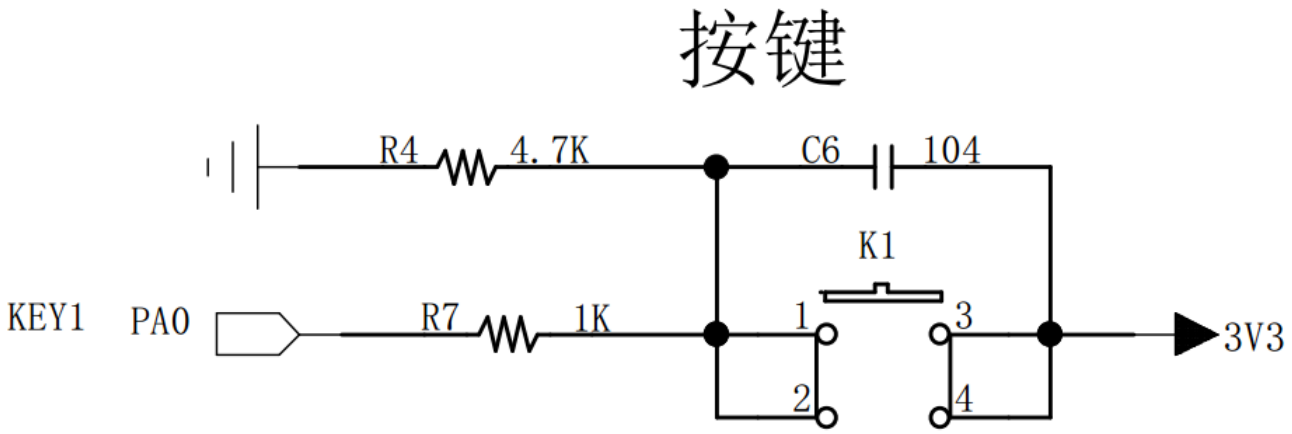


图 31-18 按键原理图。

30.7.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_GeneralTim.c 和 bsp_GeneralTim.h 文件用来存定时器驱动程序及相关宏定义。

30.7.2.1 编程要点

(1) 定时器用到的 GP IO 初始化

(2) 定时器时基结构体 TIM_TimeBaseInitTypeDef 初始化

(3) 定时器输入捕获结构体 TIM_ICInitTypeDef 初始化

(4) 编写中断服务函数，读取捕获值，计算出脉宽的时间

30.7.2.2 软件分析

30.7.2.2.1 宏定义

列表 9: 代码清单 31-9 宏定义

```
/************ 通用定时器 TIM 参数定义，只限 TIM2、3、4、5************/
// 当使用不同的定时器的时候，对应的 GPIO 是不一样的，这一点要注意
// 我们这里默认使用 TIM5

#define GENERAL_TIMx                                TIM5
#define GENERAL_TIM_RCC_CLK_ENABLE()                __HAL_RCC_TIM5_CLK_ENABLE()
#define GENERAL_TIM_RCC_CLK_DISABLE()               __HAL_RCC_TIM5_CLK_DISABLE(
                                                    )
#define GENERAL_TIM_GPIO_RCC_CLK_ENABLE()           __HAL_RCC_GPIOA_CLK_ENABLE();
#define GENERAL_TIM_CHANNELx                        TIM_CHANNEL_1
#define GENERAL_TIM_IT_CCx                          TIM_IT_CC1

#define GENERAL_TIM_IRQn                            TIM5_IRQn
#define GENERAL_TIM_IRQHANDLER                      TIM5_IRQHandler

#define KEY1_RCC_CLK_ENABLE()                       __HAL_RCC_GPIOA_CLK_ENABLE()
#define KEY1_GPIO_PIN                               GPIO_PIN_0
#define KEY1_GPIO                                   GPIOA

//------------------------------------------------------------------
// 测量的起始边沿
#define             GENERAL_TIM_STRAT_ICPolarityTIM_INPUTCHANNELPOLARITY_
↪RISING
// 测量的结束边沿
#define             GENERAL_TIM_END_ICPolarityTIM_INPUTCHANNELPOLARITY_
↪FALLING
```

使用宏定义非常方便程序升级、移植。有关具体每个宏的含义看注释即可。

30.7.2.2.2 定时器复用功能引脚初始化

列表 10: 代码清单 31-10 定时器复用功能引脚初始化

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* 基本定时器外设时钟使能 */
    GENERAL_TIM_RCC_CLK_ENABLE();

    KEY1_RCC_CLK_ENABLE();

    GPIO_InitStruct.Pin = KEY1_GPIO_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    HAL_GPIO_Init(KEY1_GPIO, &GPIO_InitStruct);

    /* 配置定时器中断优先级并使能 */
    HAL_NVIC_SetPriority(GENERAL_TIM_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(GENERAL_TIM_IRQn);
}
```

GENERAL_TIM_GPIO_Config() 函数初始化了定时器用到的相关的 GPIO，当使用不同的 GPIO 的时候，只需要修改头文件里面的宏定义即可，而不需要修改这个函数。

30.7.2.2.3 定时器模式配置

列表 11: 代码清单 31-11 定时器模式配置

```
void GENERAL_TIM_Mode_Config(void)
{
    TIM_ClockConfigTypeDef CLKSource;
    TIM_MasterConfigTypeDef MasterConfig;
    TIM_IC_InitTypeDef ICConfig;

    TIM_TimeBase.Instance = GENERAL_TIMx;
    // 驱动 CNT 计数器的时钟 = Fck_int/(psc+1)
    TIM_TimeBase.Init.Prescaler = GENERAL_TIM_PRESCALER;
    // 计数器计数模式，设置为向上计数
    TIM_TimeBase.Init.CounterMode = TIM_COUNTERMODE_UP;
    // 自动重装载寄存器的值，累计 TIM_Period+1 个频率后产生一个更新或者中断
    TIM_TimeBase.Init.Period = GENERAL_TIM_PERIOD;
    // 时钟分频因子 ，配置死区时间时需要用到
    TIM_TimeBase.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init(& TIM_TimeBase);

    CLKSource.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    HAL_TIM_ConfigClockSource(& TIM_TimeBase, &CLKSource);

    MasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    MasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(& TIM_TimeBase,
                                          &MasterConfig);

    ICConfig.ICPolarity = GENERAL_TIM_STRAT_ICPolarity;
    ICConfig.ICSelection = TIM_ICSELECTION_DIRECTTI;
    ICConfig.ICPrescaler = TIM_ICPSC_DIV1;
    ICConfig.ICFilter = 0;
```

(下页继续)

(续上页)

```
    HAL_TIM_IC_ConfigChannel(& TIM_TimeBase, &ICConfig,
                             GENERAL_TIM_CHANNELx);
}
```

ADVANCE_TIM_Mode_Config() 函数中初始化了两个结构体，有关这两个结构体成员的具体含义可参考“定时器初始化结构体详解”小节，剩下的程序参考注释阅读即可。

在初始化时基结构体的周期和时钟分频因子这两个成员时，我们使用了两个宏 GENERAL_TIM_PERIOD 和 GENERAL_TIM_PSC。GENERAL_TIM_PERIOD 配置的是 ARR 寄存器的值，决定了计数器一个周期的计数时间，默认我们配置为 0XFFFF，即最大。GENERAL_TIM_PSC 配置的是分频因子，默认配置为 72-1，则可以计算出计数器的计数周期为 (GENERAL_TIM_PSC+1)/72M = 1US。所以输入捕获能捕获的最小的时间为 1us，最长的时间为 1us*(0Xffff+1)=65536us=65.536ms，当超过这个计数周期的时候，就会产生中断，然后在中断里面做额外的处理，需要记录好产生了多少次更新中断，最后把这个更新时间加入到脉宽的时间里面。

30.7.2.2.4 主函数

列表 12: 代码清单 31-12 main 函数

```
int main(void)
{
    uint32_t time,TIM_PscCLK;
    /* 初始化系统时钟为 72MHz */
    SystemClock_Config();
    /* 初始化 LED */
    LED_GPIO_Config();

    DEBUG_USART_Config();
```

(下页继续)

(续上页)

```
    printf("\r\n按下 KEY1，测试 KEY1 按下的时间\r\n");

    GENERAL_TIM_Mode_Config();
    // TIM 计数器的驱动时钟
    TIM_PscCLK= HAL_RCC_GetHCLKFreq()/GENERAL_TIM_PRESCALER;

    /* 启动定时器 */
    HAL_TIM_Base_Start_IT(&TIM_TimeBase);
    /* 启动定时器通道输入捕获并开启中断 */
    HAL_TIM_IC_Start_IT(&TIM_TimeBase,GENERAL_TIM_CHANNELx);

    while (1) {
        /* 完成测量高电平脉宽 */
        if (TIM_ICUserValueStructure.ucFinishFlag == 1 ) {
            /* 计算高电平计数值 */
            time = TIM_ICUserValueStructure .usPeriod *
                    GENERAL_TIM_PERIOD + TIM_ICUserValueStructure .
                    usCtr;
            /* 打印高电平脉宽时间 */
            printf ( "测得高电平脉宽时间: %d.%d s\n", time /
                    TIM_PscCLK, time % TIM_PscCLK );
            TIM_ICUserValueStructure .ucFinishFlag = 0;
        }
    }
}
```

主函数很简单，主要是一些初始化，然后在一个 while 循环中打印测量的脉宽时间。

30.7.3 下载验证

把编译好的程序烧写进开发板，用 USB 线连接好电脑与开发板的 USB 转串口接口，按下 K1 按键，电脑的串口调试助手就会打印出按键按下的时间，具体见图 *31_19*。

图 31-19 测试按键按下的时间

第 31 章　TIM—电容按键检测

前面章节我们讲解了基本定时器和高级控制定时器的功能，这一章我们将介绍定时器输入捕获一个应用实例—电容按键检测，帮助我们更加深入理解定时器。

31.1 电容按键原理

电容器 (简称为电容) 就是可以容纳电荷的器件，两个金属块中间隔一层绝缘体就可以构成一个最简单的电容。如图 *32_1* (俯视图)，有两个金属片，之间有一个绝缘介质，这样就构成了一个电容。这样一个电容在电路板上非常容易实现，一般设计四周的铜片与电路板地信号连通，这样一种结构就是电容按键的模型。当电路板形状固定之后，该电容的容量也是相对稳定的。

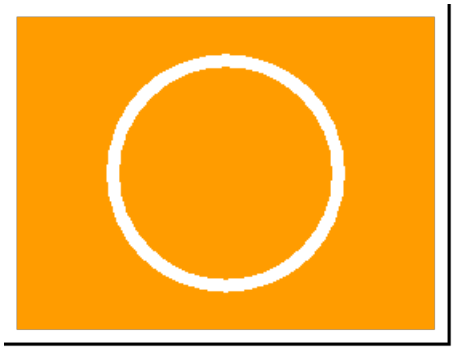


图 32-1 片状电容器

电路板制作时都会在表面上覆盖一层绝缘层，用于防腐蚀和绝缘，所以实际电路板设计时情况如图 *32_2*。电路板最上层是绝缘材料，下面一层是导电铜箔，我们根据电路走线情况设计决定铜箔的形状，再下面一层一般是 FR-4 板材。金属感应片与地信号之间有绝缘材料隔着，整个可以等效为一个电容 Cx。一般在设计时候，把金属感应片设计成方便手指触摸大小。

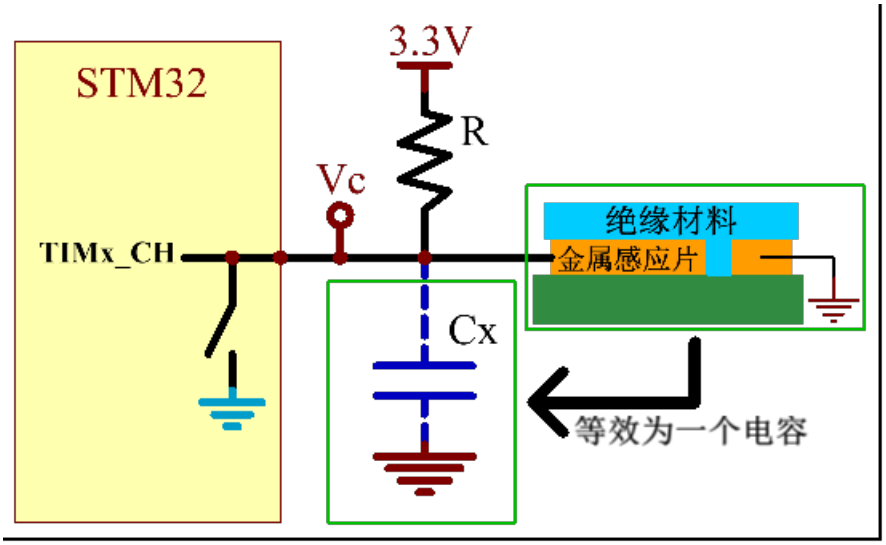


图 32-2 无手指触摸情况

在电路板未上电时，可以认为电容 Cx 是没有电荷的，在上电时，在电阻作用下，电容 Cx 就会有一个充电过程，直到电容充满，即 Vc 电压值为 3.3V，这个充电过程的时间长短受到电阻 R 阻值和电容 Cx 容值的直接影响。但是在我们选择合适电阻 R 并焊接固定到电路板上后，这个充电时间就基本上不会变了，因为此时电阻 R 已经是固定的，电容 Cx 在无外界明显干扰情况下基本上也是保持不变的。

现在，我们来看看当我们用手指触摸时会是怎样一个情况？如图 *32_3*，当我们用手指触摸时，金属感应片除了与地信号形成一个等效电容 Cx 外，还会与手指形成一个 Cs 等效电容。

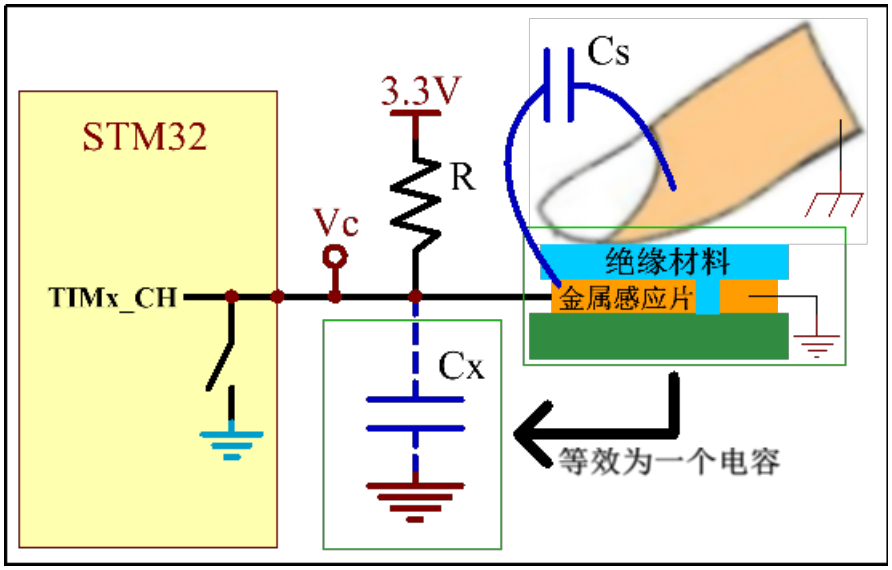


图 32-3 有手指触摸情况

此时整个电容按键可以容纳的电荷数量就比没有手指触摸时要多了，可以看成是 Cx 和 Cs 叠加的效果。在相同的电阻 R 情况下，因为电容容值增大了，导致需要更长的充电时间。也就是这个充电时间变长使得我们区分有无手指触摸，也就是电容按键是否被按下。

现在最主要的任务就是测量充电时间。充电过程可以看出是一个信号从低电平变成高电平的过程，现在就是要求出这个变化过程的时间。我们可以利用定时器输入捕获功能计算充电时间，即设置 TIMx_CH 为定时器输入捕获模式通道。这样先测量得到无触摸时的充电时间作为比较基准，然后再定时循环测量充电时间与无触摸时的充电时间作比较，如果超过一定的阈值就认为是有手指触摸。

图 *32_4* 为 Vc 跟随时间变化情况，可以看出在无触摸情况下，电压变化较快；而在有触摸时，总的电容量增大了，电压变化缓慢一些。

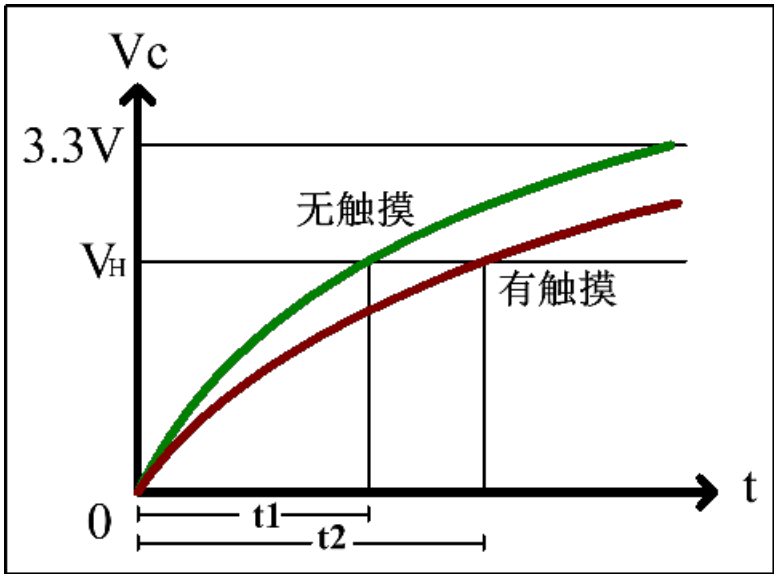


图 32-4 Vc 电压与充电时间关系

为测量充电时间，我们需要设置定时器输入捕获功能为上升沿触发，图 *32_4* 中 V_H 就是被触发上升沿的电压值，也是 STM32 认为是高电平的最低电压值，大约为 1.8V。t1 和 t2 可以通过定时器捕获/比较寄存器获取得到。

不过，在测量充电时间之前，我们必须想办法制作这个充电过程。之前的分析是在电路板上电时会有充电过程，现在我们要求在程序运行中循环检测按键，所以必须可以控制充电过程的生成。我们可以控制 TIMx_CH 引脚作为普通的 GPIO 使用，使其输出一小段时间的低电平，为电容 Cx 放电，即 Vc 为 0V。当我们重新配置 TIMx_CH 为输入捕获时电容 Cx 在电阻 R 的作用下就可以产生充电过程。

31.2 电容按键检测实验

电容按键不需要任何外部机械部件，使用方便，成本低，很容易制成与周围环境相密封的键盘，以起到防潮防湿的作用。电容按键优势突出使得越来越多电子产品使用它代替传统的机械按键。

本实验实现电容按键状态检测方法，提供一个编程实例。

31.2.1 硬件设计

MINI、霸道开发板指南者开发板板载一个电容按键，原理图设计参考图 32-5。

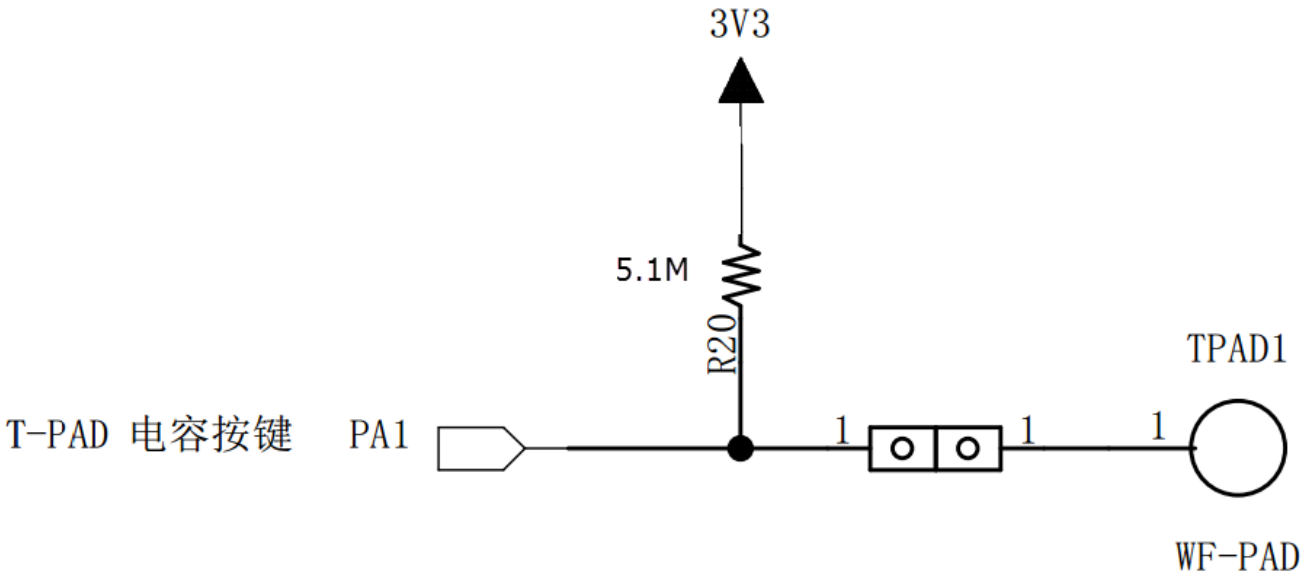


图 32-5 电容按键电路设计

标示 TPAD1 在电路板上就是电容按键实体，默认通过一个调帽连接到 PA1，即通用定时器 TIM5 的通道 2。充电电容的阻值为 5.1M，电阻的大小决定了电容按键充电的时间。

实验还用到调试串口和蜂鸣器功能，用来打印输入捕获信息和指示按键状态，这两个模块电路可参考之前相关章节。

31.2.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_tpad.c 和 bsp_tpad.h 文件用来存放电容按键检测相关函数和宏定义。

31.2.2.1 编程要点

(1) 编写定时器输入捕获相关函数

(2) 测量电容按键空载的充电时间 T1

(3) 测量电容按键有手触摸的充电时间 T2

(4) 只需要比较 T2 与 T1 的时间即可检测出按键是否有手指触摸

31.2.2.2 软件分析

31.2.2.2.1 电容按键宏定义

列表 1: 代码清单 32-1 宏定义

```
#define TOUCHPAD_GATE_VAL    100
// 定义定时器预分频，定时器实际时钟频率为：72MHz/（TOUCHPAD_TIMx_PRESCALER+1）
#define TOUCHPAD_TIM_PRESCALER               47 //1.5MHz
// 定义定时器周期
#define TOUCHPAD_TIM_ARR                      0xFFFF
/******************* TPAD 引脚配置参数定义 ************************/
#define TOUCHPAD_TIMx                         TIM5
#define TOUCHPAD_TIM_RCC_CLK_ENABLE()         __HAL_RCC_TIM5_CLK_ENABLE(
                                             )
#define TOUCHPAD_TIM_RCC_CLK_DISABLE()        __HAL_RCC_TIM5_CLK_DISABLE()
#define TOUCHPAD_GPIO_RCC_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define TOUCHPAD_GPIO_PIN                     GPIO_PIN_1
#define TOUCHPAD_GPIO                         GPIOA
#define TOUCHPAD_TIM_CHANNEL                  TIM_CHANNEL_2
#define TOUCHPAD_TIM_FLAG_CCR                 TIM_FLAG_CC2
```

有关宏的具体含义配套注释阅读即可。

31.2.2.2.2 电容按键 GPIO 配置

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* 基本定时器外设时钟使能 */
    TOUCHPAD_TIM_RCC_CLK_ENABLE();
    /* 定时器通道引脚时钟使能 */
    TOUCHPAD_GPIO_RCC_CLK_ENABLE();
    /* 定时器通道引脚配置：捕获功能设置为输入模式 */
    GPIO_InitStruct.Pin = TOUCHPAD_GPIO_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    HAL_GPIO_Init(TOUCHPAD_GPIO, &GPIO_InitStruct);
}
```

HAL_TIM_Base_MspInit () 函数初始化了定时器用到的相关的 GPIO，当使用不同的 GPIO 的时候，只需要修改头文件里面的宏定义即可，而不需要修改这个函数。

31.2.2.2.3 电容按键 TIM 模式配置

```
static void TPAD_TIM_Mode_Config(void)
{
    TIM_ClockConfigTypeDef sClockSourceConfig;
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_IC_InitTypeDef sConfigIC;

    TIM_Handle.Instance = TOUCHPAD_TIMx;
    TIM_Handle.Init.Prescaler = TOUCHPAD_TIM_PRESCALER;
    TIM_Handle.Init.CounterMode = TIM_COUNTERMODE_UP;
```

(下页继续)

(续上页)

```
    TIM_Handle.Init.Period = TOUCHPAD_TIM_ARR;
    TIM_Handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init(&TIM_Handle);

    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    HAL_TIM_ConfigClockSource(&TIM_Handle, &sClockSourceConfig);

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&TIM_Handle, &sMasterConfig);

    sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
    sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
    sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
    sConfigIC.ICFilter = 3;
    HAL_TIM_IC_ConfigChannel(&TIM_Handle, &sConfigIC,
                             TOUCHPAD_TIM_CHANNEL);
}
```

TPAD_TIM_Mode_Config() 函数中初始化了两个结构体，有关这两个结构体成员的具体含义可参考“定时器初始化结构体详解”小节，剩下的程序参考注释阅读即可。有个地方要注意的是捕获信号的极性配置，需要配置为上升沿。因为电容按键在放电之后再充电的时候是一个电平由低到高的过程。

31.2.2.2.4 电容按键复位

```
static void TPAD_Reset(void)
{
    /* 定义 IO 硬件初始化结构体变量 */
    GPIO_InitTypeDef GPIO_InitStruct;
```

(下页继续)

(续上页)

```
    /* 使能电容按键引脚对应 IO 端口时钟 */
    TOUCHPAD_GPIO_RCC_CLK_ENABLE();

    /* 设置引脚输出为低电平 */
    HAL_GPIO_WritePin(TOUCHPAD_GPIO, TOUCHPAD_GPIO_PIN, GPIO_PIN_RESET);

    /* 设定电容按键对应引脚 IO 编号 */
    GPIO_InitStruct.Pin = TOUCHPAD_GPIO_PIN;
    /* 设定电容按键对应引脚 IO 为输出模式 */
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    /* 设定电容按键对应引脚 IO 操作速度 */
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    /* 初始化电容按键对应引脚 IO */
    HAL_GPIO_Init(TOUCHPAD_GPIO, &GPIO_InitStruct);

    HAL_Delay(5);

    __HAL_TIM_SET_COUNTER(&TIM_Handle,0); // 清零定时器计数
    __HAL_TIM_CLEAR_FLAG(&TIM_Handle, TIM_FLAG_UPDATE|TIM_FLAG_CC2);//清除中
断标志

    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(TOUCHPAD_GPIO, &GPIO_InitStruct);
    HAL_TIM_IC_Start(&TIM_Handle,TOUCHPAD_TIM_CHANNEL);
}
```

开发板上电之后，电容按键默认已经充满了电，要想测得电容按键的充电时间就必须先把电容按键的电放掉，方法为让接电容按键的 IO 输出低电平即可，这个放电的时间一般都是 us 级别，我们可以稍微延时一下即可。放电完毕之后，再把连接电容按键的 IO 配置为输入，然后通过输入捕获的方法测量电容按键的充电时间，这个充电时间 T1 是没有手指触摸的情况下的充电时间，而且这个空载的充电时间非常稳定，因为电路板的硬件已经确定了。当有手指触摸的情况下，相

当于电容变大，充电时间 T2 会变长，我们只需要对比这两个时间就可以知道电容按键是否有手指触摸。

31.2.2.2.5 电容按键初始化

```
/**
* @brief  初始化触摸按键，获得空载的时候触摸按键的充电时间
* @param  无
* @retval 0：成功，1：失败
* @note   空载值一般很稳定，由硬件电路决定，该函数只需要调用一次即可
*         而且这个空载的充电时间每个硬件都不一样，最好实际测试下
*/
uint8_t TPAD_Init(void)
{
    uint16_t temp;
    // 电容按键用到的输入捕获的 IO 和捕获模式参数初始化
    TPAD_TIM_Init();

    temp = TPAD_Get_Val();

    // 电容按键空载的充电时间非常稳定，不同的硬件充电时间不一样
    // 需要实际测试所得，指南者 上的电容按键空载充电时间稳定在 218
    // 如果你觉得单次测量不准确，你可以多次测量然后取个平均值
    if ( (TPAD_DEFAULT_VAL_MIN<temp) && (temp<TPAD_DEFAULT_VAL_MAX) ) {
        tpad_default_val = temp;
        // 调试的时候可以把捕获的值打印出来，看看默认的充电时间是多少
        printf("电容按键默认充电时间为：%d us\n",tpad_default_val);
        return 0;  // 成功
    } else {
        return 1; // 失败
    }
}
```

TPAD_Init() 函数用来获取电容按键空载的充电时间，当获取到之后，把值存在 tpad_default_val 这个全局变量当中。这个空载的充电时间不同的硬件是不一样的，需要实际测试，在调试的过程中，可把捕获到的值打印出来看看。指南者开发板上这个值稳定在 218。

在 TPAD_Init() 函数中，我们是通过调用 TPAD_Get_Val() 函数来获取电容按键的充电时间的。当电容按键从 0 开始充电到 STM32 能够识别的高电平时，定时器则发生捕获，此时计数器的值会被锁存到输入捕获寄存器，我们只需要读取输入捕获寄存器的值，就可以算出这个充电的时间。通过 TPAD_Get_Val() 这个函数，我们可以测出电容按键的空载充电时间 T1 和有手触摸的情况下的充电时间 T2。

31.2.2.2.6 获取定时器输入捕获值

```
/**
* @brief  获取定时器捕获值
* @param  无
* @retval 定时器捕获值。如果超时，则直接返回定时器的计数值。
*/
uint16_t TPAD_Get_Val(void)
{
    // 每次捕获的时候，必须先复位放电
    TPAD_Reset();

    // 当电容按键复位放电之后，计数器清 0 开始计数
    TIM_SetCounter (TPAD_TIM,0);
    // 清除相关的标志位
    TIM_ClearITPendingBit (TPAD_TIM, TPAD_TIM_IT_CCx | TIM_IT_Update);

    // 等待捕获上升沿，当电容按键充电到 1.8V 左右的时候，就会被认为是上升沿
    while (TIM_GetFlagStatus (TPAD_TIM, TPAD_TIM_IT_CCx) == RESET) {
        // 如果超时了，直接返回 CNT 的值
        // 一般充电时间都是在 ms 级别以内，很少会超过定时器的最大计数值
        if (TIM_GetCounter(TPAD_TIM) > TPAD_TIM_Period-100) {
```

(下页继续)

(续上页)

```
            return TIM_GetCounter (TPAD_TIM);
        }
    }

    // 获取捕获比较寄存器的值
    return TPAD_TIM_GetCapturex_FUN(TPAD_TIM);
}
```

31.2.2.2.7 获取最大输入捕获值

```
static uint16_t TPAD_Get_Val(void)
{
    TPAD_Reset();
    while (__HAL_TIM_GET_FLAG(&TIM_Handle,TOUCHPAD_TIM_FLAG_CCR)
            ==RESET) {
        uint16_t count;
        count=__HAL_TIM_GET_COUNTER(&TIM_Handle);
        if (count>(TOUCHPAD_TIM_ARR-500))
            return count;//超时了，直接返回 CNT 的值
    };
    return HAL_TIM_ReadCapturedValue(&TIM_Handle,TOUCHPAD_TIM_CHANNEL);
}
```

该函数接收一个参数，用来指定获取电容按键捕获值的循环次数，函数的返回值则为 num 次发生捕获中最大的捕获值。

当我们用手指触摸电容按键的时候，常常会有干扰或者是误触发，所以我们一般选取最大的值为有效值。

31.2.2.2.8 电容按键状态扫描

```
/**
* @brief  按键扫描函数
* @param  无
* @retval 1: 按键有效，0: 按键无效
*/
uint8_t TPAD_Scan(void)
{
    // keyen: 按键检测使能标志
    // 0: 可以开始检测
    // >0: 还不能开始检测，表示按键一直被按下
    // 注意: keytn 这个变量由 static 修饰，相当于一个全局变量，但是因为是在函数内部定义，

    // 所以是相当于这个函数的全局变量，每次修改之前保留的是上一次的值
    static uint8_t keyen=0;

    uint8_t res=0,sample=3;
    uint16_t scan_val;

    // 根据 sample 值采样多次，并取最大值，小的一般是干扰或者是误触摸
    scan_val = TPAD_Get_MaxVal(sample);

    // 当扫描的值大于空载值加上默认的门限值之后，表示按键按下
    // 这个 TPAD_GATE_VAL 根据硬件决定，需要实际测试
    if (scan_val > (tpad_default_val+TPAD_GATE_VAL)) {
        // 再次检测，类似于机械按键的去抖
        scan_val = TPAD_Get_MaxVal(sample);
        if ( ( keyen == 0 )&& (scan_val > (tpad_default_val+TPAD_GATE_VAL)))
            res = 1;  // 有效的按键

        // 如果按键一直被按下，keyen 的值会一直在 keyen 的初始值和 keyen-1 之间循环，
永远不会等于 0
```

(下页继续)

(续上页)

```
        keyen = 2;
    }

    // 当按键没有被按下或者 keyen>0 时，会执行 keyen--
    if ( keyen > 0)
        keyen--;
    return res;
}
```

按键扫描函数不断的检测充电时间，当大于 tpad_default_val+TPAD_GATE_VAL 时，表示按键被按下，其中 TPAD_GATE_VAL 是一个宏，具体多大需要实际测试。具体的我们可以通过调用 TPAD_Get_Val() 函数来测试按键有手触摸的情况下的充电值，然后再减去 tpad_default_val 的值就可以得到 TPAD_GATE_VAL，当减小这个门限值的时候可以提高按键的灵敏度。

在按键扫描函数中，我们引入了一个按键检测标志 keyen，其由关键字 static 修饰，相当于一个全局变量，每次修改这个变量的时候其保留的都是上一次的值。引入一个按键检测标志是为了消除按键是否一直按下的情况，如果按键一直被按下 keyen 的值会一直在 keyen 的初始值和 keyen-1 之间循环，永远不会等于 0，则永远都不会被认为按键按下，需要等待释放。有关函数更加详细的说明看程序的注释即可。

31.2.2.2.9 主函数

```
int main(void)
{
    /* 配置系统时钟为 72 MHz */
    SystemClock_Config();

    /* 蜂鸣器初始化 */
    BEEP_GPIO_Config();

    /* 初始化 USART1 配置模式为 115200 8-N-1 */
```

(下页继续)

(续上页)

```
    DEBUG_USART_Config();

    printf ( "\r\n野火 STM32 输入捕获电容按键检测实验\r\n" );
    printf ( "\r\n触摸电容按键，蜂鸣器则会响\r\n" );

    /* 初始化电容按键获取未被触摸时参数 */
    TPAD_Init();

    while (1) {
        if ( TPAD_Scan(0)  ) {
            BEEP(ON);
            HAL_Delay(30);
            BEEP(OFF);
        }
    }
}
```

主函数初始化了蜂鸣器和串口，然后等待电容按键初始化成功，如果不成功则会一直等待。初始化成功之后，在一个 while 无限循环中不断的扫描按键，当按键按下之后蜂鸣器响 25ms，然后关掉。

31.2.3 下载验证

把编译好的程序烧写到开发板，手按电容按键，蜂鸣器则会响。

第 32 章　IWDG—独立看门狗

本章参考资料：《STM32F10X-中文参考手册》IWDG 章节。

学习本章时，配合《STM32F10X-中文参考手册》IWDG 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

32.1 IWDG 简介

STM32 有两个看门狗，一个是独立看门狗另外一个是窗口看门狗，独立看门狗号称宠物狗，窗口看门狗号称警犬，本章我们主要分析独立看门狗的功能框图和它的应用。独立看门狗用通俗一点的话来解释就是一个 12 位的递减计数器，当计数器的值从某个值一直减到 0 的时候，系统就会产生一个复位信号，即 IWDG_RESET。如果在计数没减到 0 之前，刷新了计数器的值的话，那么就不会产生复位信号，这个动作就是我们经常说的喂狗。看门狗功能由 VDD 电压域供电，在停止模式和待机模式下仍能工作。

32.2 IWDG 功能框图剖析

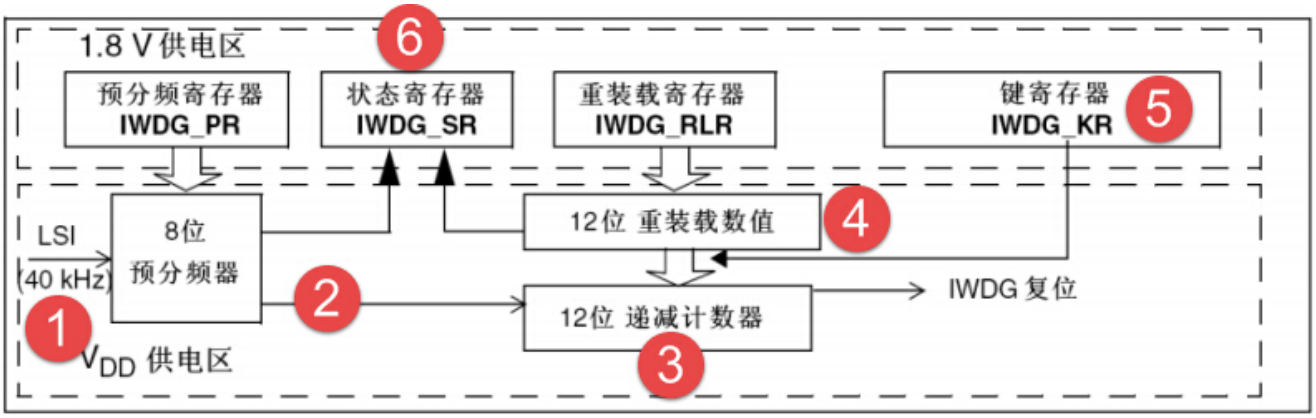


图 33-1 IWDG 功能框图

32.2.1 独立看门狗时钟

独立看门狗的时钟由独立的 RC 振荡器 LSI 提供，即使主时钟发生故障它仍然有效，非常独立。LSI 的频率一般在 30~60KHZ 之间，根据温度和工作场合会有一定的漂移，我们一般取 40KHZ，所以独立看门狗的定时时间并不一定非常精确，只适用于对时间精度
要求比较低的场合。

32.2.2 计数器时钟

递减计数器的时钟由 LSI 经过一个 8 位的预分频器得到，我们可以操作预分频器寄存器 IWDG_PR 来设置分频因子，分频因子可以是：[4,8,16,32,64,128,256,256]，计数器时钟 CK_CNT= 40/ 4*2^PRV，一个计数器时钟计数器就减一。

32.2.3 计数器

独立看门狗的计数器是一个 12 位的递减计数器，最大值为 0XFFF，当计数器减到 0 时，会产生一个复位信号:IWDG_RESET，让程序重新启动运行，如果在计数器减到 0 之前刷新了计数器的值的话，就不会产生复位信号，重新刷新计数器值的这个动作我们俗称喂狗。

32.2.4 重装载寄存器

重装载寄存器是一个 12 位的寄存器，里面装着要刷新到计数器的值，这个值的大小决定着独立看门狗的溢出时间。超时时间 Tout = (4*2^prv) / 40 * rlv (s) ，prv 是预分频器寄存器的值，rlv 是重装载寄存器的值。

32.2.5 键寄存器

键寄存器 IWDG_KR 可以说是独立看门狗的一个控制寄存器，主要有三种控制方式，往这个寄存器写入下面三个不同的值有不同的效果。

表格 33-1 键寄存器取值枚举

键值	键值作用
0XAAAA	把 RLR 的值重装载到 CNT
0X5555	PR 和 RLR 这两个寄存器可写
0XCCCC	启动 IWDG

通过写往键寄存器写 0XCCC 来启动看门狗是属于软件启动的方式，一旦独立看门狗启动，它就关不掉，只有复位才能关掉。

32.2.6 状态寄存器

状态寄存器 SR 只有位 0：PVU 和位 1：RVU 有效，这两位只能由硬件操作，软件操作不了。RVU：看门狗计数器重装载值更新，硬件置 1 表示重装载值的更新正在进行中，更新完毕之后由硬件清 0。PVU: 看门狗预分频值更新，硬件置’1’指示预分频值的更新正在进行中，当更新完成后，由硬件清 0。所以只有当 RVU/PVU 等于 0 的时候才可以更新重装载寄存器/预分频寄存器。

32.3 怎么用 IWDG

独立看门狗一般用来检测和解决由程序引起的故障，比如一个程序正常运行的时间是 50ms，在运行完这个段程序之后紧接着进行喂狗，我们设置独立看门狗的定时溢出时间为 60ms，比我们需要监控的程序 50ms 多一点，如果超过 60ms 还没有喂狗，那就说明我们监控的程序出故障了，跑飞了，那么就会产生系统复位，让程序重新运行。

32.4 IWDG 超时实验

32.4.1 硬件设计

1. IWDG 一个
2. 按键一个
3. LED 一个

IWDG 属于单片机内部资源，不需要外部电路，需要一个外部的按键和 LED，通过按键来喂狗，喂狗成功 LED 亮，喂狗失败，程序重启，LED 灭一次。

32.4.2 软件设计

我们编写两个 IWDG 驱动文件，bsp_iwdg.h 和 bsp_iwdg.c，用来存放 IWDG 的初始化配置函数。

32.4.2.1 代码分析

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。

32.4.2.1.1 IWDG 配置函数

列表 1: 代码 33-1 IWDG 配置函数

```
void IWDG_Config(uint8_t prv ,uint16_t rlv)
{
    // 使能 预分频寄存器 PR 和重装载寄存器 RLR 可写
    IWDG_Handle.Instance = IWDG;

    // 设置预分频器值
```

(下页继续)

(续上页)

```
    IWDG_SetPrescaler = prv;

    // 设置重装载寄存器值
    IWDG_SetReload = rlv;

    // 初始化 IWDG
    HAL_IWDG_Init(&IWDG_Handle);

    // 启动 IWDG
    __HAL_IWDG_START(&IWDG_Handle);
}
```

IWDG 配置函数有两个形参，prv 用来设置预分频的值，取值可以是：

列表 2: 代码 33-2 形参 prv 取值

```
/*
*     @arg IWDG_Prescaler_4:    IWDG prescaler set to 4
*     @arg IWDG_Prescaler_8:    IWDG prescaler set to 8
*     @arg IWDG_Prescaler_16:   IWDG prescaler set to 16
*     @arg IWDG_Prescaler_32:   IWDG prescaler set to 32
*     @arg IWDG_Prescaler_64:   IWDG prescaler set to 64
*     @arg IWDG_Prescaler_128:  IWDG prescaler set to 128
*     @arg IWDG_Prescaler_256:  IWDG prescaler set to 256
*/
```

这些宏在 stm32f4xx_hal_iwdg.h 中定义，宏展开是 8 位的 16 进制数，具体作用是配置配置预分频寄存器 IWDG_PR，获得各种分频系数。形参 rlv 用来设置重装载寄存器 IWDG_RLR 的值，取值范围为 0~0XFFF。溢出时间 Tout = prv/40 * rlv (s)，prv 可以是 [4,8,16,32,64,128,256]。如果我们需要设置 1s 的超时溢出，prv 可以取 IWDG_ PRESCALER _64，rlv 取 625，即调用:IWDG_Config(IWDG_Prescaler_64 ,625)。Tout=64/40*625=1s。

32.4.2.1.2 喂狗函数

列表 3: 代码 33-3 喂狗函数

```
void IWDG_Feed(void)
{
    // 把重装载寄存器的值放到计数器中，喂狗，防止 IWDG 复位
    // 当计数器的值减到 0 的时候会产生系统复位
    HAL_IWDG_Refresh(&IWDG_Handle);
}
```

32.4.2.1.3 主函数

列表 4: 代码清单 33-4

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();

    HAL_Delay(1000);
    /* 检查是否为独立看门狗复位 */
    if (__HAL_RCC_GET_FLAG(RCC_FLAG_IWDGRST) != RESET) {
        /* 独立看门狗复位 */
        /*  亮红灯 */
        LED_RED;

        /* 清除标志 */
        __HAL_RCC_CLEAR_RESET_FLAGS();

        /* 如果一直不喂狗，会一直复位，加上前面的延时，会看到红灯闪烁
```

(下页继续)

(续上页)

```
        在 1s 时间内喂狗的话，则会持续亮绿灯 */
    } else {
        /* 不是独立看门狗复位（可能为上电复位或者手动按键复位之类的） */
        /* 亮蓝灯 */
        LED_BLUE;
    }

    /* 初始化按键 */
    Key_GPIO_Config();

    // IWDG 1s 超时溢出
    IWDG_Config(IWDG_PRESCALER_64 ,625);

    //while 部分是我们在项目中具体需要写的代码，这部分的程序可以用独立看门狗来监控
    //如果我们知道这部分代码的执行时间，比如是 500ms，那么我们可以设置独立看门狗的
    //溢出时间是 600ms，比 500ms 多一点，如果要被监控的程序没有跑飞正常执行的话，那么
    //执行完毕之后就会执行喂狗的程序，如果程序跑飞了那程序就会超时，到达不了喂狗的
    //程序，此时就会产生系统复位。但是也不排除程序跑飞了又跑回来了，刚好喂狗了，
    //歪打正着。所以要想更精确的监控程序，可以使用窗口看门狗，窗口看门狗规定必须
    //在规定的窗口时间内喂狗。
    while (1) {
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON  ) {
            // 喂狗，如果不喂狗，系统则会复位，复位后亮红灯，如果在 1s
            // 时间内准时喂狗的话，则会亮绿灯
            IWDG_Feed();
            //喂狗后亮绿灯
            LED_GREEN;
        }

    }
}
```

主函数中我们初始化好系统时钟、LED 和按键相关的配置，设置 IWDG 1s 超时溢出之后，进入 while 死循环，通过按键来喂狗，如果喂狗成功，则亮绿灯，如果喂狗失败的话，系统重启，程序重新执行，当执行到 __HAL_RCC_GET_FLAG 函数的时候，则会检测到是 IWDG 复位，然后让红灯亮。如果喂狗一直失败的话，则会一直产生系统复位，加上前面延时的效果，则会看到红灯一直闪烁。

我们这里是通过按键来模拟一个喂狗程序，真正的项目中则不是这样使用。while 部分是我们在项目中具体需要写的代码，这部分的程序可以用独立看门狗来监控，如果我们知道这部分代码的执行时间，比如是 500ms，那么我们可以设置独立看门狗的溢出时间是 510ms，比 500ms 多一点，如果要被监控的程序没有跑飞正常执行的话，那么执行完毕之后就会执行喂狗的程序，如果程序跑飞了那程序就会超时，到达不了喂狗的程序，此时就会产生系统复位，但是也不排除程序跑飞了又跑回来了，刚好喂狗了，歪打正着。所以要想更精确的监控程序，可以使用窗口看门狗，窗口看门狗规定必须在规定的窗口时间内喂狗，早了不行，晚了也不行。

32.4.3 下载验证

把编译好的程序下载到开发板，在 1s 的时间内通过按键来不断的喂狗，如果喂狗失败，红灯闪烁。如果一直喂狗成功，则绿灯常亮。

第 33 章　WWDG—窗口看门狗

本章参考资料：《STM32F10X-中文参考手册》WWDG 章节。

学习本章时，配合《STM32F10X-中文参考手册》WWDG 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

33.1 WWDG 简介

STM32 有两个看门狗，一个是独立看门狗，一个是窗口看门狗。我们知道独立看门狗的工作原理就是一个递减计数器不断的往下递减计数，当减到 0 之前如果没有喂狗的话，产生复位。窗口看门狗跟独立看门狗一样，也是一个递减计数器不断的往下递减计数，当减到一个固定值 0X40 时还不喂狗的话，产生复位，这个值叫窗口的下限，是固定的值，不能改变。这个是跟独立看门狗类似的地方，不同的地方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，上限值由用户独立设置。窗口看门狗计数器的值必须在上窗口和下窗口之间才可以喂狗，这就是窗口看门狗中窗口两个字的含义

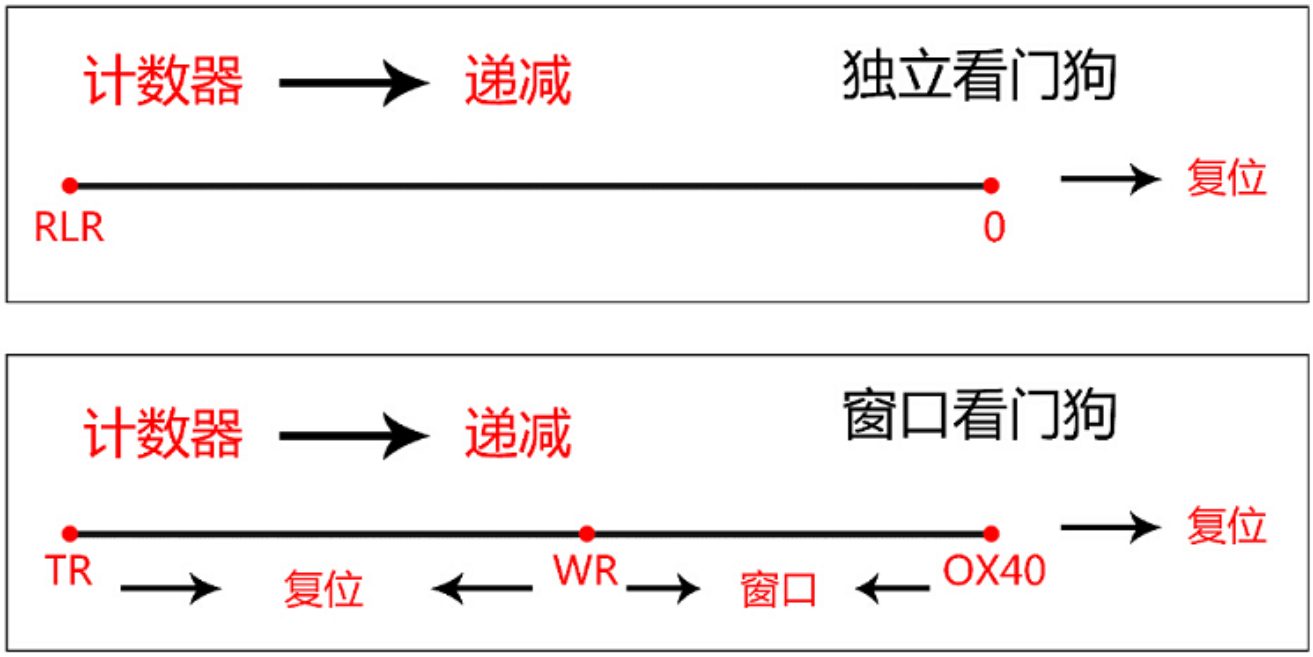


图 34-1 IWDG 与 WWDG 区别

RLR 是重装载寄存器，用来设置独立看门狗的计数器的值。TR 是窗口看门狗的计数器的值，由用户独立设置，WR 是窗口看门狗的上窗口值，由用户独立设置。

33.2 WWDG 功能框图剖析

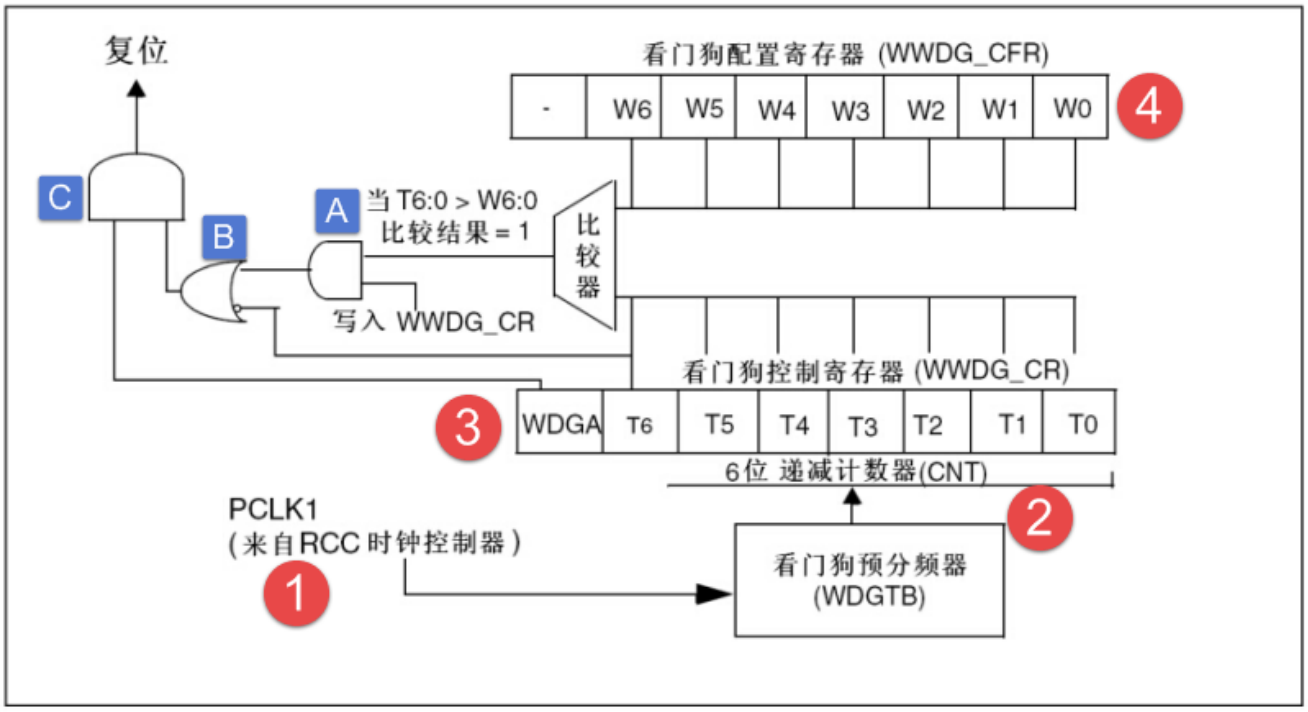


图 34-2 窗口看门狗功能框图

33.2.1 窗口看门狗时钟

窗口看门狗时钟来自 PCLK1，PCLK1 最大是 36M，由 RCC 时钟控制器开启。

33.2.2 计数器时钟

计数器时钟由 CK 计时器时钟经过预分频器分频得到，分频系数由配置寄存器 CFR 的位 8:7 WDGTB[1:0] 配置，可以是 [0,1,2,3]，其中 CK 计时器时钟 =PCLK1/4096，除以 4096 是手册规定的，没有为什么。所以计数器的时钟 CNT_CK=PCLK1/4096/(2^WDGTB)，
这就可以算出计数器减一个数的时间 T= 1/CNT_CK = Tpclk1 * 4096 * (2^WDGTB)。

33.2.3 窗口计数器

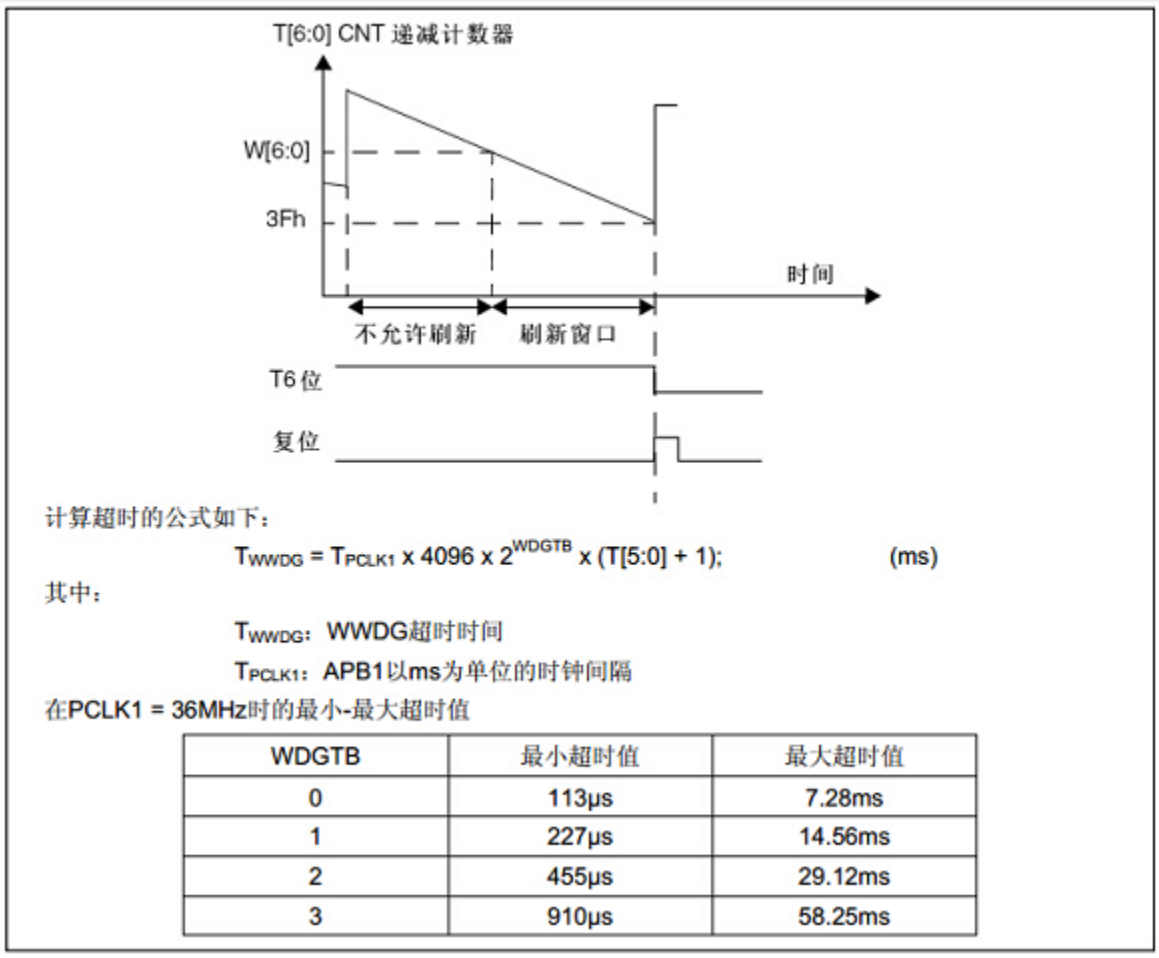
窗口看门狗的计数器是一个递减计数器，共有 7 位，其值存在控制寄存器 CR 的位 6:0，即 T[6:0]，当 7 个位全部为 1 时是 0X7F，这个是最大值，当递减到 T6 位变成 0 时，即从 0X40 变为 0X3F 时候，会产生看门狗复位。这个值 0X40 是看门狗能够递减到的最小值，所以计数器的值只能是：0X40~0X7F 之间，实际上真正用来计数的是 T[5:0]。当递减计数器递减到 0X40 的时候，还不会马上产生复位，如果使能了提前唤醒中断：CFR 位 9 EWI 置 1，则产生提前唤醒中断，如果真进入了这个中断的话，就说明程序肯定是出问题了，

那么在中断服务程序里面我们就需要做最重要的工作，比如保存重要数据，或者报警等，这个中断我们也叫它死前中断。

33.2.4 窗口值

我们知道窗口看门狗必须在计数器的值在一个范围内才可以喂狗，其中下窗口的值是固定的 0X40，上窗口的值可以改变，具体的由配置寄存器 CFR 的位 6:0 W[6:0] 设置。其值必须大于 0X40，如果小于或者等于 0X40 就是失去了窗口的价值，而且也不能大于计数器的值，所以必须得小于 0X7F。那窗口值具体要设置成多大？这个得根据我们需要监控的程序的运行时间来决定。如果我们要监控的程序段 A 运行的时间为 Ta，当执行完这段程序之后就要进行喂狗，如果在窗口时间内没有喂狗的话，那程序就肯定是出问题了。一般计数器的值 TR 设置成最大 0X7F，窗口值为 WR，计数器减一个数的时间为 T，那么时间：(TR-WR)*T 应该稍微大于 Ta 即可，这样就能做到刚执行完程序段 A 之后喂狗，起到监控的作用，这样也就可以算出 WR 的值是多少。

33.2.5 □ 计算看门狗超时时间



计算超时的公式如下：

$$T_{WWDG} = T_{PCLK1} \times 4096 \times 2^{WDGTB} \times (T[5:0] + 1); \quad (ms)$$

其中：

T_{WWDG}：WWDG超时时间

T_{PCLK1}：APB1以ms为单位的时钟间隔

在PCLK1 = 36MHz时的最小-最大超时值

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

图 34-3 窗口看门狗时序图

这个图来自数据手册，从图我们知道看门狗超时时间：Twwdg = Tpclk1 x 4096 x 2^wdgtb x (T[5:0] + 1) ms，当 PCLK1 = 36MHZ 时，WDGTB 取不同的值时有最小和最大的超时时间，那这个最小和最大的超时时间该怎么理解，又是怎么算出来的？讲起来有点绕，这里我稍微讲解下 WDGTB=0 时是怎么算的。递减计数器有 7 位 T[6:0]，当位 6 变为 0 的时候就会产生复位，实际上有效的计数位是 T[5:0]，而且 T6 必须先设置为 1。如果 T[5:0]=0 时，递减计数器再减一次，就产生复位了，那这减一的时间就等于计数器的周期 =1/CNT_CK = Tpclk1 * 4096 * (2^WDGTB) = 1/36 * 4096 *2^0 = 113.7us，这个就是最短的超时时间。如果 T[5:0] 全部装满为 1，即 63，当他减到 0X40 变成 0X3F 时，所需的时间就是最大的超时时间 =113.7*2^5=113.7*64=7.2768ms。同理，当 WDGTB

等于 1/2/3 时，代入公式即可。

33.3 怎么用 WWDG

WWDG 一般被用来监测，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。比如一个程序段正常运行的时间是 50ms，在运行完这个段程序之后紧接着进行喂狗，如果在规定的时间窗口内还没有喂狗，那就说明我们监控的程序出故障了，跑飞了，那么就会产生系统复位，让程序重新运行。

33.4 WWDG 喂狗实验

33.4.1 硬件设计

1. WWDG 一个
2. LED 两个

WWDG 属于单片机内部资源，不需要外部电路，需要两个 LED 来指示程序的运行状态。

33.4.2 软件设计

我们编写两个 WWDG 驱动文件，bsp_wwdg.h 和 bsp_wwdg.c，用来存放 WWDG 的初始化配置函数。

33.4.2.1 代码分析

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。

33.4.2.2 WWDG 配置函数

列表 1: 代码 34-1 WWDG 配置函数

```
void WWDG_Config(uint8_t tr, uint8_t wr, uint32_t prv)
{
    // 开启 WWDG 时钟
    __HAL_RCC_WWDG_CLK_ENABLE();
    // 配置 WWDG 中断优先级
    WWDG_NVIC_Config();
    // 配置 WWDG 句柄即寄存器基地址
    WWDG_Handle.Instance = WWDG;
    // 设置预分频器值
    WWDG_Handle.Init.Prescaler = prv;
    // 设置上窗口值
    WWDG_Handle.Init.Window = wr;
    // 设置计数器的值
    WWDG_Handle.Init.Counter = tr;
    // 使能提前唤醒中断
    WWDG_Handle.Init.EWIMode = WWDG_EWI_ENABLE;
    // 初始化 WWDG
    HAL_WWDG_Init(&WWDG_Handle);
}
```

WWDG 配置函数有三个形参，tr 是计数器的值，一般我们设置成最大 0X7F，wr 是上窗口的值，这个我们要根据监控的程序的运行时间来设置，但是值必须在 0X40 和计数器的值之间，prv 用来设置预分频的值，取值可以是：

列表 2: 代码 34-2 形参 prv 取值

```
/*
*     @arg WWDG_Prescaler_1: WWDG counter clock = (PCLK1/4096)/1
*     @arg WWDG_Prescaler_2: WWDG counter clock = (PCLK1/4096)/2
```

(下页继续)

(续上页)

```
 *      @arg WWDG_Prescaler_4: WWDG counter clock = (PCLK1/4096)/4
 *      @arg WWDG_Prescaler_8: WWDG counter clock = (PCLK1/4096)/8
 */
```

这些宏在 stm32f4xx_hal_wwdg.h 中定义，宏展开是 32 位的 16 进制数，具体作用是设置配置寄存器 CFR 的位 8:7 WDGTB[1:0]，获得各种分频系数。

33.4.2.3 WWDG 中断优先级函数

```
// WWDG 中断优先级初始化
static void WWDG_NVIC_Config(void)
{
    HAL_NVIC_SetPriority(WWDG_IRQn,0,0);
    HAL_NVIC_EnableIRQ(WWDG_IRQn);
}
```

在递减计数器减到 0X40 的时候，我们开启了提前唤醒中断，这个中断我们称它为死前中断或者叫遗嘱中断，在中断函数里面我们应该出来最重要的事情，而且必须得快，因为递减计数器再减一次，就会产生系统复位。

33.4.2.4 提前唤醒中断复位程序

列表 3: 代码 34-3 提前唤醒中断服务程序

```
// WWDG 中断服务程序，如果发生了此中断，表示程序已经出现了故障，
// 这是一个死前中断。在此中断服务程序中应该干最重要的事，
// 比如保存重要的数据等
void WWDG_IRQHandler(void)
{
    //WWDG 中断服务处理函数，用户代码在提前唤醒中断回调函数数中添加
    HAL_WWDG_IRQHandler(&WWDG_Handle);
```

(下页继续)

(续上页)

```
}

void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg)
{
    //黄灯亮，点亮 LED 只是示意性的操作，
    //真正使用的时候，这里应该是做最重要的事情
    LED_YELLOW;
}
```

33.4.2.5 喂狗函数

列表 4: 代码 34-4 喂狗函数

```
// 喂狗
void WWDG_Feed(void)
{
    // 喂狗，刷新递减计数器的值，设置成最大 WDG_CNT=0X7F
    HAL_WWDG_Refresh(&WWDG_Handle);
}
```

喂狗就是重新刷新递减计数器的值防止系统复位，喂狗一般是在主函数中喂。

33.4.2.6 主函数

列表 5: 代码 34-5 主函数

```
int main(void)
{
    uint8_t wwdg_tr, wwdg_wr;
    /* 系统时钟初始化成 72MHz */
```

(下页继续)

(续上页)

```
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();

    //检查窗口看门狗复位标志位
    if (__HAL_RCC_GET_FLAG(RCC_FLAG_WWDGRST) != RESET) {
        // 看门狗复位启动，红色灯亮
        LED_RED;

        //清除复位标志位
        __HAL_RCC_CLEAR_RESET_FLAGS();
    } else {
        // 正常启动，蓝色灯亮
        LED_BLUE;
    }
    HAL_Delay(500);
    LED_RGBOFF;
    HAL_Delay(500);

    // WWDG 配置
    // 初始化 WWDG: 配置计数器初始值，配置上窗口值，启动 WWDG，使能提前唤醒中断
    WWDG_Config(127,80,WWDG_PRESCALER_8);

    // 窗口值我们在初始化的时候设置成 0X5F，这个值不会改变
    wwdg_wr = WWDG->CFR & 0X7F;

    while (1) {
        //-----------------------------------------------------
        // 这部分应该写需要被 WWDG 监控的程序，这段程序运行行的时间
        // 决定了窗口值应该设置成多大。
        //-----------------------------------------------------
        // 计时器值，初始化成最大 0X7F，当开启 WWDG 时候，这个值会不断减小
```

(下页继续)

(续上页)

```
        // 当计数器的值大于窗口值时喂狗的话，会复位，当计数器减少到 0X40
        // 还没有喂狗的话就非常非常危险了，计数器再减一次到了 0X3F 时就复位
        // 所以要当计数器的值在窗口值和 0X40 之间的时候喂狗，其中 0X40 是固定的。
        wwdg_tr = WWDG->CR & 0X7F;
        if ( wwdg_tr == wwdg_wr) {
            // 喂狗，重新设置计数器的值为最大 0X7F
            WWDG_Feed();
            // 正常喂狗，绿色灯闪烁
            LED2_TOGGLE;
        }
    }
}
```

主函数中我们把 WWDG 的计数器的值设置为 0X7F，上窗口值设置为 0X5F，分频系数为 8 分频。在 while 死循环中，我们不断读取计数器的值，当计数器的值减小到小于上窗口值的时候，我们喂狗，让计数器重新计数。

在 while 死循环中，一般是我们需要监控的程序，这部分代码的运行时间，决定了上窗口值应该设置为多少，当监控的程序运行完毕之后，我们需要执行喂狗程序，比起独立看门狗，这个喂狗的窗口时间是非常短的，对时间要求很精确。如果没有在这个窗口时间内喂狗的话，那就说明程序出故障了，会产生提前唤醒中断，最后系统复位。

33.4.2.6.1 下载验证

把编译好的程序下载到开发板，LED_BLUE 被点亮，一段时间之后熄灭，之后 LED_BLUE 一直就没有被点亮过，说明系统没有产生复位，如果产生复位的话 LED_BLUE 会再被点亮一次。中断服务程序中的 LED_YELLOW 也没被点亮过，LED_GREEN 闪烁，说明喂狗正常。

第 34 章　SDIO—SD 卡读写测试

本章参考资料：《STM32F10X-中文参考手册》、《STM32F103 增强型系列数据手册》以及 SD 简易规格文件《Physical Layer Simplified Specification V2.0》(版本号：2.00)。

阅读本章内容之前，建议先阅读 SD 简易规格文件。

34.1 SDIO 简介

SD 卡 (Secure Digital Memory Card) 在我们生活中已经非常普遍了，控制器对 SD 卡进行读写通信操作一般有两种通信接口可选，一种是 SPI 接口，另外一种就是 SDIO 接口。SDIO 全称是安全数字输入/输出接口，多媒体卡 (MMC)、SD 卡、SD I/O 卡都有 SDIO 接口。STM32F10x 系列控制器有一个 SDIO 主机接口，它可以与 MMC 卡、SD 卡、SD I/O 卡以及 CE-ATA 设备进行数据传输。MMC 卡可以说是 SD 卡的前身，现阶段已经用得很少。SD I/O 卡本身不是用于存储的卡，它是指利用 SDIO 传输协议的一种外设。比如 Wi-Fi Card，它主要是提供 Wi-Fi 功能，有些 Wi-Fi 模块是使用串口或者 SPI 接口进行通信的，但 Wi-Fi SDIO Card 是使用 SDIO 接口进行通信的。并且一般设计 SD I/O 卡是可以插入到 SD 的插槽。CE-ATA 是专为轻薄笔记本硬盘设计的硬盘高速通讯接口。

多媒体卡协会网站www.mmca.org中提供了有 MMCA 技术委员会发布的多媒体卡系统规范。

SD 卡协会网站 www.sdcard.org 中提供了 SD 存储卡和 SDIO 卡系统规范。

CE-ATA 工作组网站www.ce-ata.org中提供了 CE_ATA 系统规范。

随之科技发展，SD 卡容量需求越来越大，SD 卡发展到现在也是有几个版本的，关于 SDIO 接口的设备整体概括见图 *35_1*。

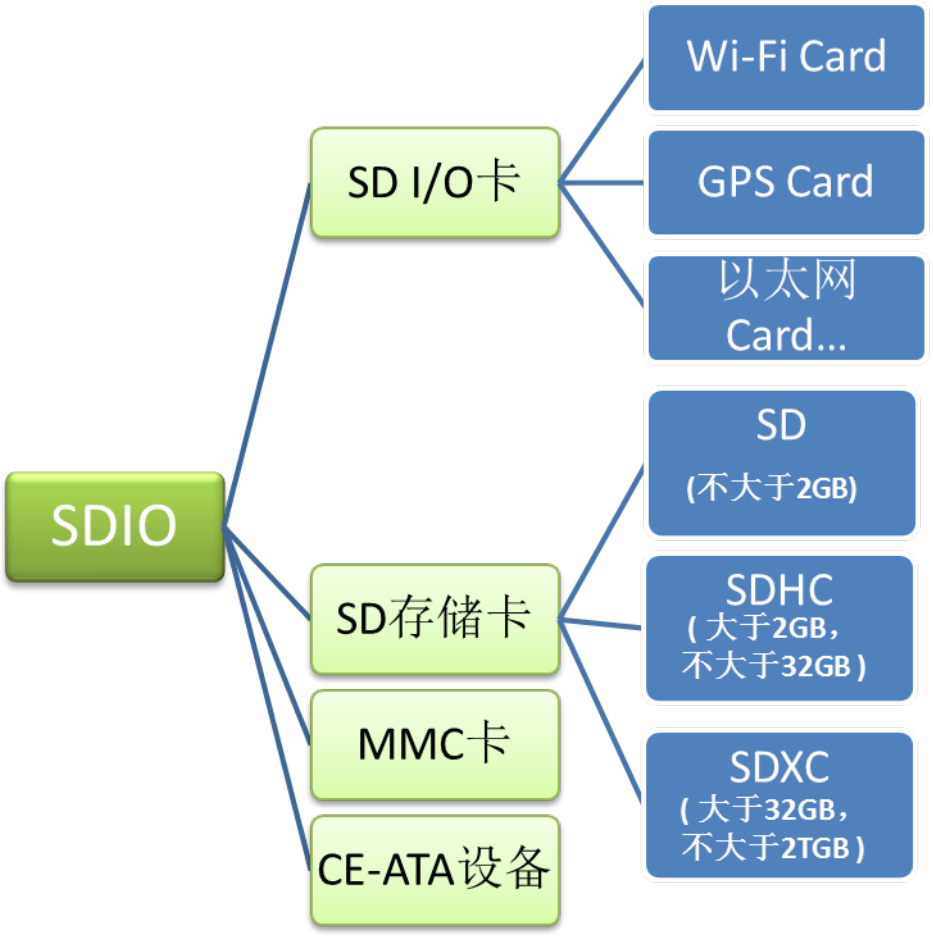


图 35-1 SDIO 接口的设备

关于 SD 卡和 SD I/O 部分内容可以在 SD 协会网站获取到详细的介绍，比如各种 SD 卡尺寸规则、读写速度标示方法、应用扩展等等信息。

本章内容针对 SD 卡使用讲解，对于其他类型卡的应用可以参考相关系统规范实现，所以对于控制器中针对其他类型卡的内容可能在本章中简单提及或者被忽略，本章内容不区分 SDIO 和 SD 卡这两个概念。即使目前 SD 协议提供的 SD 卡规范版本最新是 4.01 版本，但 STM32F10x 系列控制器只支持 SD 卡规范版本 2.0，即只支持标准容量 SD 和高容量 SDHC 标准卡，不支持超大容量 SDXC 标准卡，所以可以支持的最高卡容量是 32GB。

34.2 SD 卡物理结构

一张 SD 卡包括有存储单元、存储单元接口、电源检测、卡及接口控制器和接口驱动器 5 个部分，见图 35-2。存储单元是存储数据部件，存储单元通过存储单元接口与卡控制单元进行数据传输；电源检测单元保证 SD 卡工作在合适的电压下，如出现掉电或上状态时，它会使控制单元和存储单元接口复位；卡及接口控制单元控制 SD 卡的运行状态，它包括有 8 个寄存器；接口驱动器控制 SD 卡引脚的输入输出。

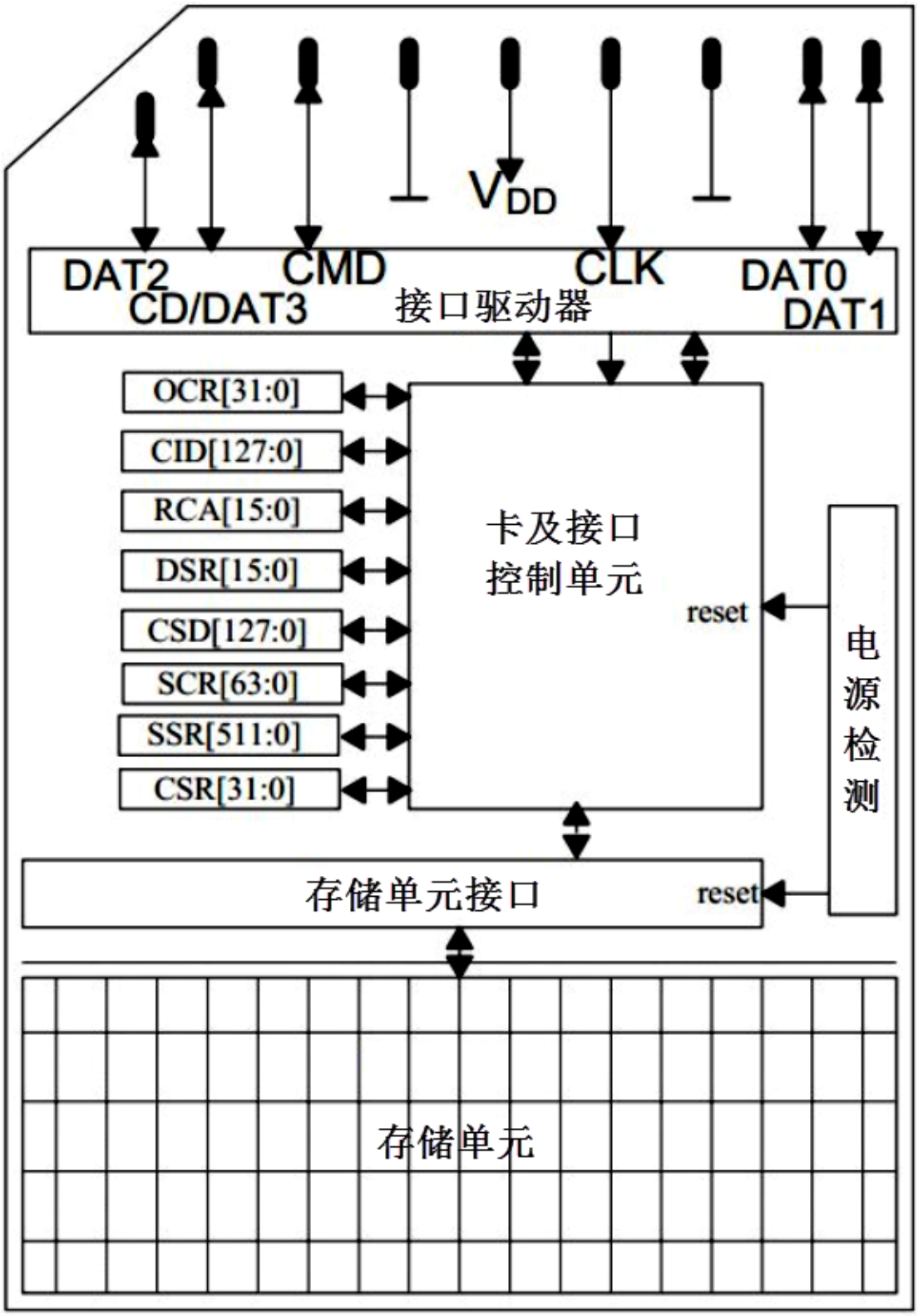

VDD
DAT2
CD/DAT3
CMD
接口驱动器
CLK
DAT0
DAT1
OCR[31:0]
CID[127:0]
RCA[15:0]
DSR[15:0]
CSD[127:0]
SCR[63:0]
SSR[511:0]
CSR[31:0]
卡及接口
控制单元
reset
电源检测
存储单元接口
reset
存储单元

图 35-2 SD 卡物理结构

SD 卡总共有 8 个寄存器，用于设定或表示 SD 卡信息，参考表 35-1。这些寄存器只能通过对应的命令访问，对 SD 卡进行控制操作并不是像操作控制器 GPIO 相关寄存器那样一次读写一个寄存器的，它是通过命令来控制，SDIO 定义了 64 个命令，每个命令都有特殊意义，可以实现某一特定功能，SD 卡接收到命令后，根据命令要求对 SD 卡内部寄存器进行修改，程序控制中只需要发送组合命令就可以实现 SD 卡的控制以及读写操作。

表 35-1 SD 卡寄存器

名称	bit 宽度	描述
CID	128	卡识别号 (Card identification number): 用来识别的卡的个体号码 (唯一的)
RCA	16	相对地址 (Relative card address): 卡的本地系统地址，初始化时，动态地由卡建议，主机核准。
DSR	16	驱动级寄存器 (Driver Stage Register): 配置卡的输出驱动
CSD	128	卡的特定数据 (Card Specific Data): 卡的操作条件信息
SCR	64	SD 配置寄存器 (SD Configuration Register):SD 卡特殊特性信息
OCR	32	操作条件寄存器 (Operation conditions register)
SSR	512	SD 状态 (SD Status):SD 卡专有特征的信息
CSR	32	卡状态 (Card Status): 卡状态信息

每个寄存器位的含义可以参考 SD 简易规格文件《Physical Layer Simplified Specification V2.0》第 5 章内容。

34.3 SDIO 总线

34.3.1 总线拓扑

SD 卡一般都支持 SDIO 和 SPI 这两种接口，本章内容只介绍 SDIO 接口操作方式，如果需要使用 SPI 操作方式可以参考 SPI 相关章节。另外，STM32F10x 系列控制器的 SDIO 是不支持 SPI 通信

模式的，如果需要用到 SPI 通信只能使用 SPI 外设。

SD 卡总线拓扑参考图 35-3。虽然可以共用总线，但不推荐多卡槽共用总线信号，要求一个单独 SD 总线应该连接一个单独的 SD 卡。

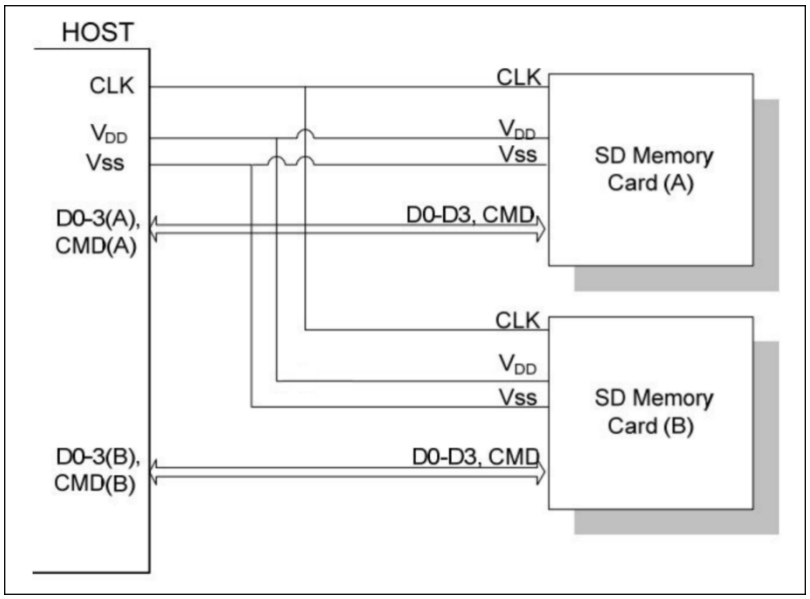


图 35-3 SD 卡总线拓扑

SD 卡使用 9-pin 接口通信，其中 3 根电源线、1 根时钟线、1 根命令线和 4 根数据线，具体说明如下：

- **CLK：**时钟线，由 SDIO 主机产生，即由 STM32 控制器输出；
- **CMD：**命令控制线，SDIO 主机通过该线发送命令控制 SD 卡，如果命令要求 SD 卡提供应答 (响应)，SD 卡也是通过该线传输应答信息；
- **D0-3：**数据线，传输读写数据；SD 卡可将 D0 拉低表示忙状态；
- **VDD、VSS1、VSS2：**电源和地信号。

在之前的 I2C 以及 SPI 章节都有详细讲解了对应的通信时序，实际上，SDIO 的通信时序简单许多，SDIO 不管是从主机控制器向 SD 卡传输，还是 SD 卡向主机控制器传输都只以 CLK 时钟线的**上升沿**为有效。SD 卡操作过程会使用两种不同频率的时钟同步数据，一个是识别卡阶段时钟频率 FOD，最高为 400kHz，另外一个是数据传输模式下时钟频率 FPP，默认最高为 25MHz，如果通过相关寄存器配置使 SDIO 工作在高速模式，此时数据传输模式最高频率为 50MHz。

对于 STM32 控制器只有一个 SDIO 主机，所以只能连接一个 SDIO 设备，开发板上集成了一个 Micro SD 卡槽和 SDIO 接口的 WiFi 模块，要求只能使用其中一个设备。SDIO 接口的 WiFi 模块一般集成有使能线，如果需要用到 SD 卡需要先控制该使能线禁用 WiFi 模块。

34.3.2 总线协议

SD 总线通信是基于命令和数据传输的。通讯由一个起始位（“0”），由一个停止位（“1”）终止。SD 通信一般是主机发送一个命令 (Command)，从设备在接收到命令后作出响应 (Response)，如有需要会有数据 (Data) 传输参与。

SD 总线的基本交互是命令与响应交互，见图 35-4。

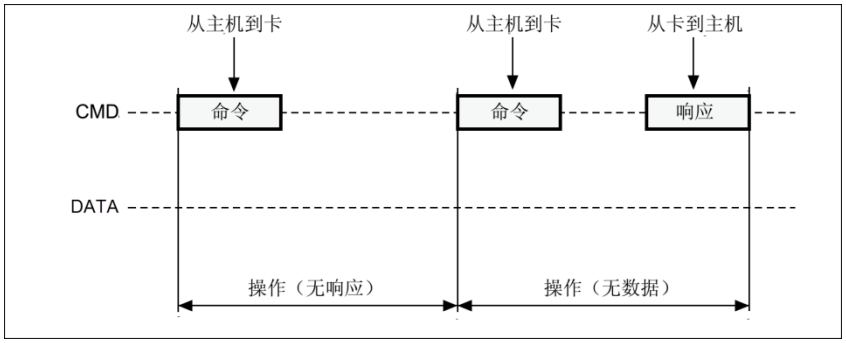


图 35-4 命令与响应交互

SD 数据是以块 (Black) 形式传输的，SDHC 卡数据块长度一般为 512 字节，数据可以从主机到卡，也可以是从卡到主机。数据块需要 CRC 位来保证数据传输成功。CRC 位由 SD 卡系统硬件生成。STM32 控制器可以控制使用单线或 4 线传输，本开发板设计使用 4 线传输。图 *35_5* 为主机向 SD 卡写入数据块操作示意。

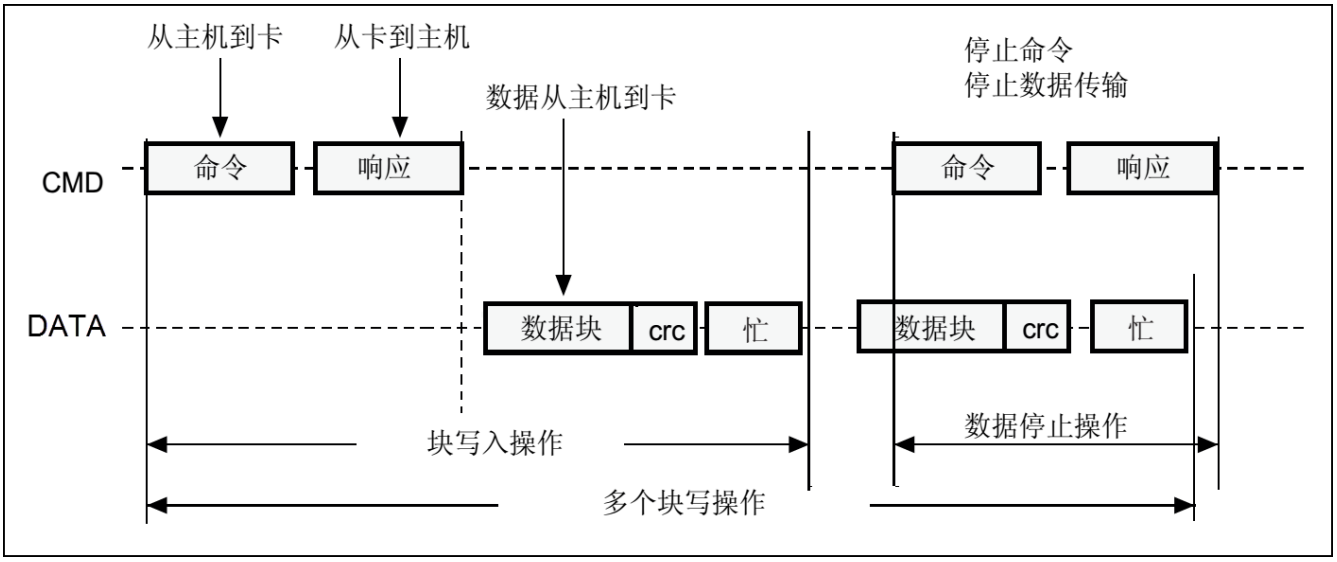


图 35-5 多块写入操作

SD 数据传输支持单块和多块读写，它们分别对应不同的操作命令，多块写入还需要使用命令来停止整个写入操作。数据写入前需要检测 SD 卡忙状态，因为 SD 卡在接收到数据后编程到存储区过程需要一定操作时间。SD 卡忙状态通过把 D0 线拉低表示。

数据块读操作与之类似，只是无需忙状态检测。

使用 4 数据线传输时，每次传输 4bit 数据，每根数据线都必须有起始位、终止位以及 CRC 位，CRC 位每根数据线都要分别检查，并把检查结果汇总然后在数据传输完后通过 D0 线反馈给主机。

SD 卡数据包有两种格式，一种是常规数据 (8bit 宽)，它先发低字节再发高字节，而每个字节则是先发高位再发低位，4 线传输示意如图 *35_6*。

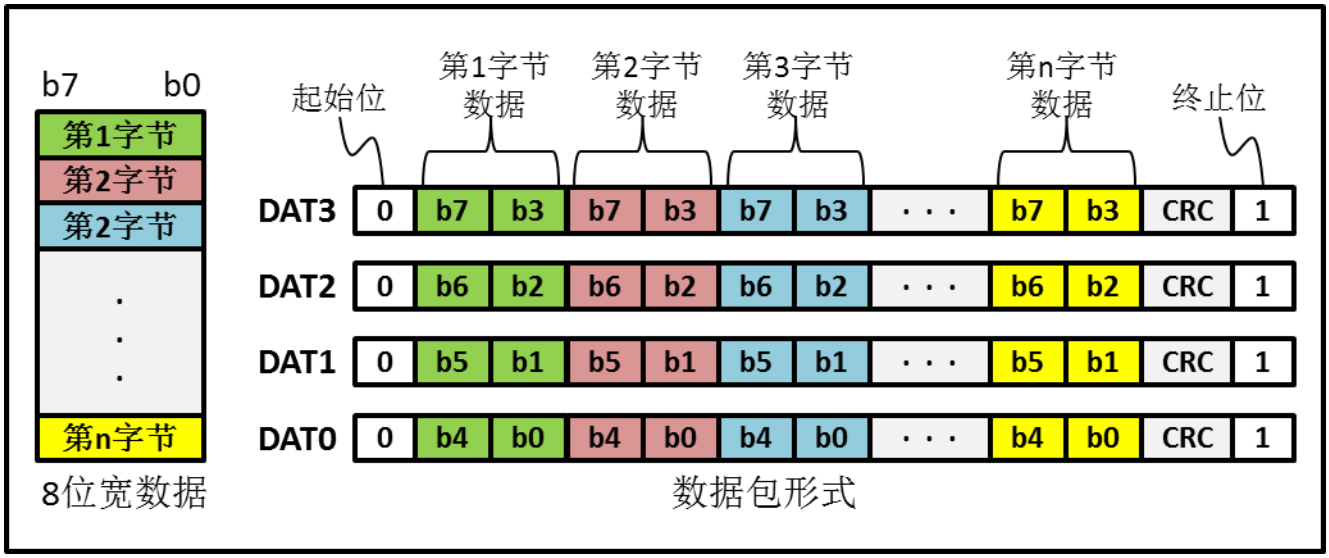


图 35-6 8 位宽数据包传输

4 线同步发送，每根线发送一个字节的其中两个位，数据位在四线顺序排列发送，DAT3 数据线发较高位，DAT0 数据线发较低位。

另外一种数据包发送格式是宽位数据包格式，对 SD 卡而言宽位数据包发送方式是针对 SD 卡 SSR(SD 状态) 寄存器内容发送的，SSR 寄存器总共有 512bit，在主机发出 ACMD13 命令后 SD 卡将 SSR 寄存器内容通过 DAT 线发送给主机。宽位数据包格式示意见图 *35_7*。

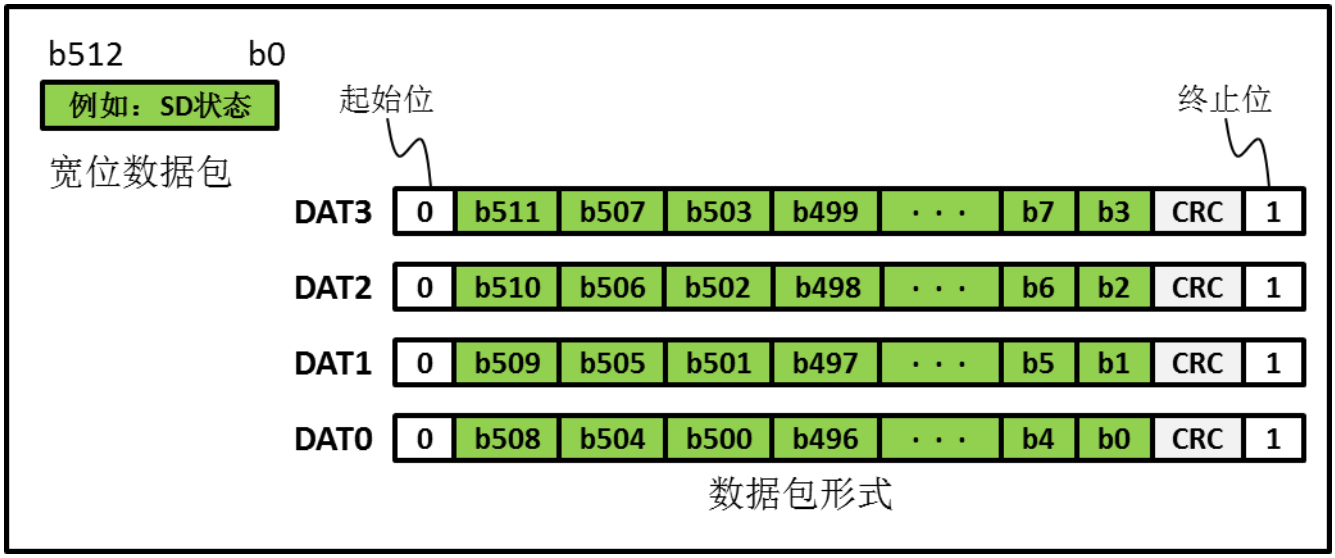


图 35-7 宽位数据包传输

34.3.3 命令

SD 命令由主机发出，以广播命令和寻址命令为例，广播命令是针对与 SD 主机总线连接的所有从设备发送的，寻址命令是指定某个地址设备进行命令传输。

34.3.3.1 命令格式

SD 命令格式固定为 48bit，都是通过 CMD 线连续传输的（数据线不参与），见图 *35_8*。

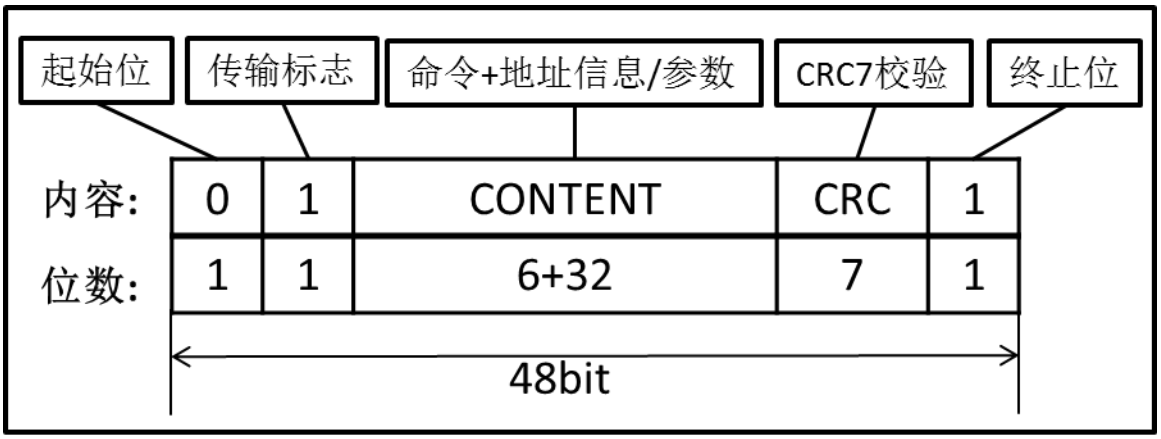


图 35-8 SD 命令格式

SD 命令的组成如下：

- 起始位和终止位：命令的主体包含在起始位与终止位之间，它们都只包含一个数据位，起始位为 0，终止位为 1。
- 传输标志：用于区分传输方向，该位为 1 时表示命令，方向为主机传输到 SD 卡，该位为 0 时表示响应，方向为 SD 卡传输到主机。

命令主体内容包括命令、地址信息/参数和 CRC 校验三个部分。

- 命令号：它固定占用 6bit，所以总共有 64 个命令 (代号：CMD0~CMD63)，每个命令都有特定的用途，部分命令不适用于 SD 卡操作，只是专门用于 MMC 卡或者 SD I/O 卡。
- 地址/参数：每个命令有 32bit 地址信息/参数用于命令附加内容，例如，广播命令没有地址信息，这 32bit 用于指定参数，而寻址命令这 32bit 用于指定目标 SD 卡的地址。

- CRC7 校验：长度为 7bit 的校验位用于验证命令传输内容正确性，如果发生外部干扰导致传输数据个别位状态改变将导致校准失败，也意味着命令传输失败，SD 卡不执行命令。

34.3.3.2 命令类型

SD 命令有 4 种类型：

- 无响应广播命令 (bc)，发送到所有卡，不返回任务响应；
- 带响应广播命令 (bcr)，发送到所有卡，同时接收来自所有卡响应；
- 寻址命令 (ac)，发送到选定卡，DAT 线无数据传输；
- 寻址数据传输命令 (adtc)，发送到选定卡，DAT 线有数据传输。

另外，SD 卡主机模块系统旨在为各种应用程序类型提供一个标准接口。在此环境中，需要有特定的客户/应用程序功能。为实现这些功能，在标准中定义了两种类型的通用命令：特定应用命令 (ACMD) 和常规命令 (GEN_CMD)。要使用 SD 卡制造商特定的 ACMD 命令如 ACMD6，需要在发送该命令之前无发送 CMD55 命令，告知 SD 卡接下来的命令为特定应用命令。CMD55 命令只对紧接的第一个命令有效，SD 卡如果检测到 CMD55 之后的第一条命令为 ACMD 则执行其特定应用功能，如果检测发现不是 ACMD 命令，则执行标准命令。

34.3.3.3 命令描述

SD 卡系统的命令被分为多个类，每个类支持一种 “卡的功能设置”。表 35-2 列举了 SD 卡部分命令信息，更多详细信息可以参考 SD 简易规格文件说明，表中填充位和保留位都必须被设置为 0。

虽然没有必须完全记住每个命令详细信息，但越熟悉命令对后面编程理解非常有帮助。

表 35-2 SD 部分命令描述

命令序号	类型	参数	响应	缩写	描述	

下页继续

表 1 – 续上页

基本命令 (Class 0)						
CMD0	bc	[31:0] 填充位	•	GO_IDLE_STATE	复位所有的卡到 idle 状态。	
CMD2	bcr	[31:0] 填充位	R2	ALL_SEND_CID	通知所有卡通过 CMD 线返回 CID 值。	
CMD3	bcr	[31:0] 填充位	R6	SEND_RELATIVE_ADDR	通知所有卡发布新 RCA。	
CMD4	bc	[31:16]DSR[15:0] 填充位	•	SET_DSR	编程所有卡的 DSR。	
CMD7	ac	[31:16]RCA[15:0] 填充位	R1b	SELECT/DESELECT_CARD	选择/取消选择 RCA 地址卡。	
CMD8	bcr	[31:12] 保留位 [11:8]VHS[7:0] 检查模式	R7	SEND_IF_COND	发送 SD 卡接口条件，包含主机支持的电压信息，并询问卡是否支持。	

下页继续

表 1 – 续上页

CMD9	ac	[31:16]RCA[15:0] 填充位	R2	SEND_CSD	选定卡通过 CMD 线发送 CSD 内容	
CMD10	ac	[31:16]RCA[15:0] 填充位	R2	SEND_CID	选定卡通过 CMD 线发送 CID 内容	
CMD12	ac	[31:0] 填充位	R1b	STOP_TRANSMISSION	强制停止传输	
CMD13	ac	[31:16]RCA[15:0] 填充位	R1	SEND_STATUS	选定卡通过 CMD 线发送它状态寄存器	
CMD15	ac	[31:16]RCA[15:0] 填充位	•	GO_INACTIVE_STATE	使选定卡进入“inactive”状态	
面向块的读操作 (Class	2)					

下页继续

表 1 – 续上页

CMD16	ac	[31:0] 块长度	R1	SET_BLOCK_LEN	设置块长度	
CMD17	adtc	[31:0] 数据地址	R1	READ_SINGLE_BLOCK	读单个块	
CMD18	adtc	[31:0] 数据地址	R1	READ_MULTIPLE_BLOCK	连续从 SD 卡读取数据块，直到被 CMD12 中断。块长度同 CMD17。	
面向块的写操作 (Class	4)					

下页继续

表 1 – 续上页

CMD24	adtc	[31:0] 数据地址	R1	WRITE_BLOCK	对于标准卡，写入 SEL_BLOCK_LEN 长度字节的块；对于 SDHC 卡，写入 512 字节的块。	
CMD25	adtc	[31:0] 数据地址	R1	WRITE_MILTIPLE_BLOCK	连续向 SD 卡写入数据块，直到被 CMD12 中断。每块长度同 CMD17。	
CMD27	adtc	[31:0] 填充位	R1	PROGRAM_CSD	对 CSD 的可编程位进行编程	
擦除命令 (Class 5)						
CMD32	ac	[31:0] 数据地址	R1	ERASE_WR_BLK_START	设置擦除的起始块地址	
CMD33	ac	[31:0] 数据地址	R1	ERASE_WR_BLK_END	设置擦除的结束块地址	
CMD38	ac	[31:0] 填充位	R1b	ERASE	擦除预先选定的块	

下页继续

表 1 – 续上页

加锁命令 (Class	7)					
CMD42	adtc	[31:0] 保留	R1	LOCK_UNLOCK	加/解锁 SD 卡	
特定应用命令 (Class	8)					
CMD55	ac	[31:16]RCA[15:0] 填充位	R1	APP_CMD	指定下个命令为特定应用命令，不是标准命令	
CMD56	adtc	[31:1] 填充位 [0] 读/写	R1	GEN_CMD	通用命令，或者特定应用命令中，用于传输一个数据块，最低位为 1 表示读数据，为 0 表示写数据	
SD 卡特定应用命令						
ACMD6	ac	[31:2] 填充位 [1:0] 总线宽度	R1	SET_BUS_WIDTH	定义数据总线宽度（‘00’=1bit,’10’=4bit)。	

下页继续

表 1 – 续上页

ACMD13	adtc	[31:0] 填充位	R1	SD_STATUS	发送 SD 状态	
ACMD41	Bcr	[32] 保留位 [30]HCS(OCR[30])	[29:24] 保留位 [23:0]VDD 电压 (OCR[23:0])	R3	SD_SEND_OP_COND	主机要求卡发送它的支持信息 (HCS) 和 OCR 寄存器内容。
ACMD51	adtc	[31:0] 填充位	R1	SEND_SCR	读取配置寄存器 SCR	

34.3.4 响应

响应由 SD 卡向主机发出，部分命令要求 SD 卡作出响应，这些响应多用于反馈 SD 卡的状态。SDIO 总共有 7 个响应类型 (代号：R1~R7)，其中 SD 卡没有 R4、R5 类型响应。特定的命令对应有特定的响应类型，比如当主机发送 CMD3 命令时，可以得到响应 R6。与命令一样，SD 卡的响应也是通过 CMD 线连续传输的。根据响应内容大小可以分为短响应和长响应。短响应是 48bit 长度，只有 R2 类型是长响应，其长度为 136bit。各个类型响应具体情况如表 35-3。

除了 R3 类型之外，其他响应都使用 CRC7 校验来校验，对于 R2 类型是使用 CID 和 CSD 寄存器内部 CRC7。

表 35-3 SD 卡响应类型

R1(正常响应命令)								
描述	起始位	传输位	命令号	卡状态	CRC7	终止位		
Bit	47	46	[45:40]	[39:8]	[7:1]	0		
位宽	1	1	6	32	7	1		
值	“0”	“0”	x	x	x	“1”		
备注	如果有传输到卡的数据，那么在数据线可能有 busy 信号							
R2(CID,CSD 寄存器)								
描述	起始位	传输位	保留	[127:1]	终止位			
Bit	135	134	[133:128]	[127:1]	0			
位宽	1	1	6	x	1			
值	“0”	“0”	“111111”	CID 或者 CSD 寄存器 [127:1] 位的值	“1”			
备注	CID 寄存器内容作为 CMD2 和 CMD10 响应，CSD 寄存器内容作为 CMD9 响应。							
R3(OCR 寄存器)								
描述	起始位	传输位	保留	OCR 寄存器	保留	终止位		
Bit	47	46	[45:40]	[39:8]	[7:1]	0		
位宽	1	1	6	32	7	1		
值	“0”	“0”	“111111”	x	“1111111”	“1”		
备注	OCR 寄存器的值作为 ACMD41 的响应							
R6(发布的 RCA 寄存								

34.4 SD 卡的操作模式及切换

34.4.1 SD 卡的操作模式

SD 卡有多个版本，STM32 控制器目前最高支持《Physical Layer Simplified Specification V2.0》定义的 SD 卡，STM32 控制器对 SD 卡进行数据读写之前需要识别卡的种类：V1.0 标准卡、V2.0 标准卡、V2.0 高容量卡或者不被识别卡。

SD 卡系统 (包括主机和 SD 卡) 定义了两种操作模式：卡识别模式和数据传输模式。在系统复位后，主机处于卡识别模式，寻找总线上可用的 SDIO 设备；同时，SD 卡也处于卡识别模式，直到被主机识别到，即当 SD 卡接收到 SEND_RCA(CMD3) 命令后，SD 卡就会进入数据传输模式，而主机在总线上所有卡被识别后也进入数据传输模式。在每个操作模式下，SD 卡都有几种状态，参考表 35-4，通过命令控制实现卡状态的切换。

表 35-4 SD 卡状态与操作模式

操作模式	SD 卡状态
无效模式 (Inactive)	无效状态 (Inactive State)
卡识别模式 (Card identification mode)	空闲状态 (Idle State)
	准备状态 (Ready State)
	识别状态 (Identification State)
数据传输模式 (Data transfer mode)	待机状态 (Stand-by State)
	传输状态 (Transfer State)
	发送数据状态 (Sending-data State)
	接收数据状态 (Receive-data State)
	编程状态 (Programming State)
	断开连接状态 (Disconnect State)

34.4.2 卡识别模式

在卡识别模式下，主机会复位所有处于“卡识别模式”的 SD 卡，确认其工作电压范围，识别 SD 卡类型，并且获取 SD 卡的相对地址 (卡相对地址较短，便于寻址)。在卡识别过程中，要求 SD 卡工作在识别时钟频率 FOD 的状态下。卡识别模式下 SD 卡状态转换如图 *35_9*。

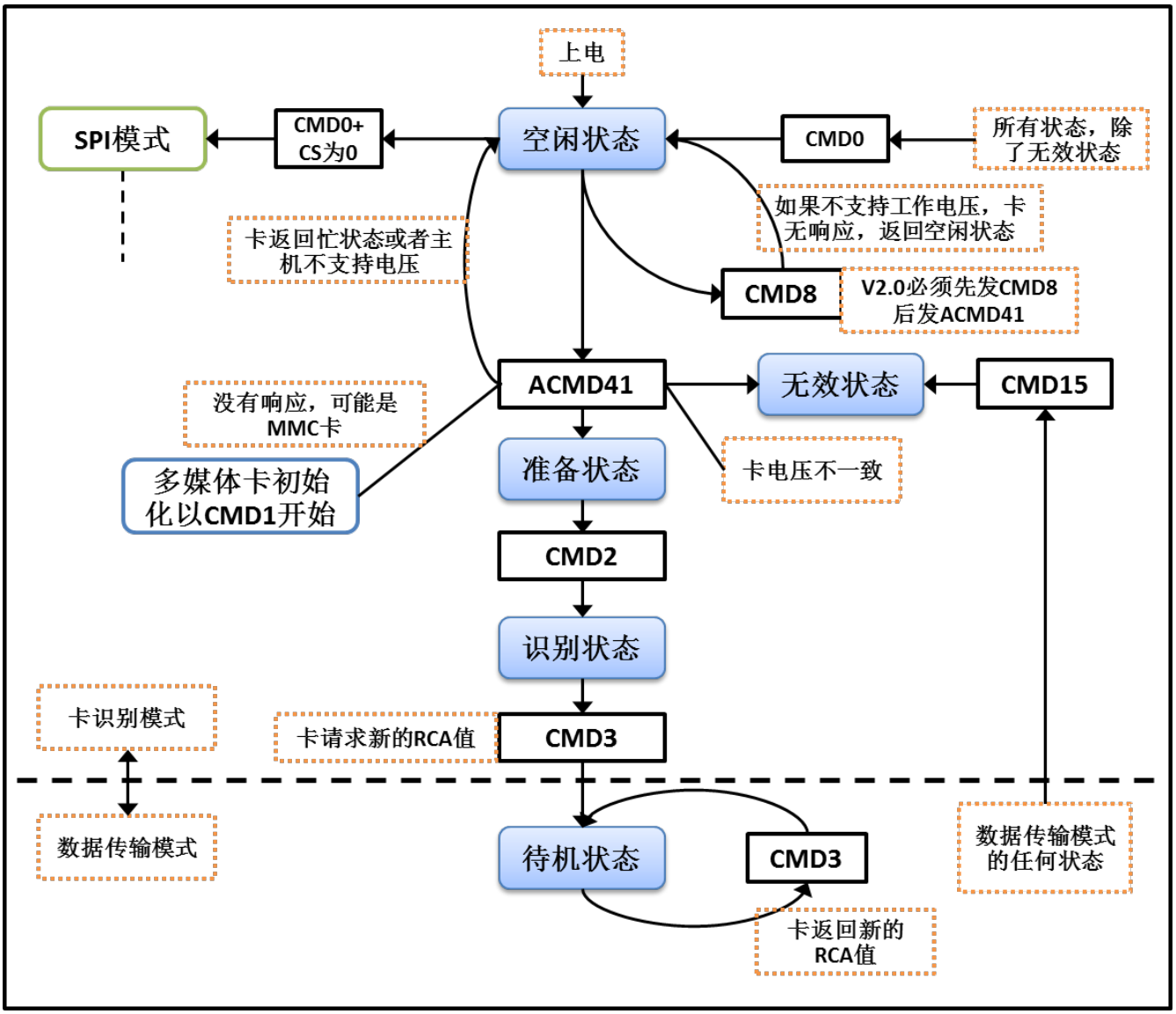


图 35-9 卡识别模式状态转换图

主机上电后，所有卡处于空闲状态，包括当前处于无效状态的卡。主机也可以发送

GO_IDLE_STATE(CMD0) 让所有卡软复位从而进入空闲状态，但当前处于无效状态的卡并不会复位。

主机在开始与卡通信前，需要先确定双方在互相支持的电压范围内。SD 卡有一个电压支持范围，主机当前电压必须在该范围可能才能与卡正常通信。SEND_IF_COND(CMD8) 命令就是用于验证卡接口操作条件的 (主要是电压支持)。卡会根据命令的参数来检测操作条件匹配性，如果卡支持主机电压就产生响应，否则不响应。而主机则根据响应内容确定卡的电压匹配性。CMD8 是 SD 卡标准 V2.0 版本才有的新命令，所以如果主机有接收到响应，可以判断卡为 V2.0 或更高版本 SD 卡。

SD_SEND_OP_COND(ACMD41) 命令可以识别或拒绝不匹配它的电压范围的卡。ACMD41 命令的 VDD 电压参数用于设置主机支持电压范围，卡响应会返回卡支持的电压范围。对于对 CMD8 有响应的卡，把 ACMD41 命令的 HCS 位设置为 1，可以测试卡的容量类型，如果卡响应的 CCS 位为 1 说明为高容量 SD 卡，否则为标准卡。卡在响应 ACMD41 之后进入准备状态，不响应 ACMD41 的卡为不可用卡，进入无效状态。ACMD41 是应用特定命令，发送该命令之前必须先发 CMD55。

ALL_SEND_CID(CMD2) 用来控制所有卡返回它们的卡识别号 (CID)，处于准备状态的卡在发送 CID 之后就进入识别状态。之后主机就发送 SEND_RELATIVE_ADDR(CMD3) 命令，让卡自己推荐一个相对地址 (RCA) 并响应命令。这个 RCA 是 16bit 地址，而 CID 是 128bit 地址，使用 RCA 简化通信。卡在接收到 CMD3 并发出响应后就进入数据传输模式，并处于待机状态，主机在获取所有卡 RCA 之后也进入数据传输模式。

34.4.3 数据传输模式

只有 SD 卡系统处于数据传输模式下才可以进行数据读写操作。数据传输模式下可以将主机 SD 时钟频率设置为 FPP，默认最高为 25MHz，频率切换可以通过 CMD4 命令来实现。数据传输模式下，SD 卡状态转换过程见图 *35_10*。

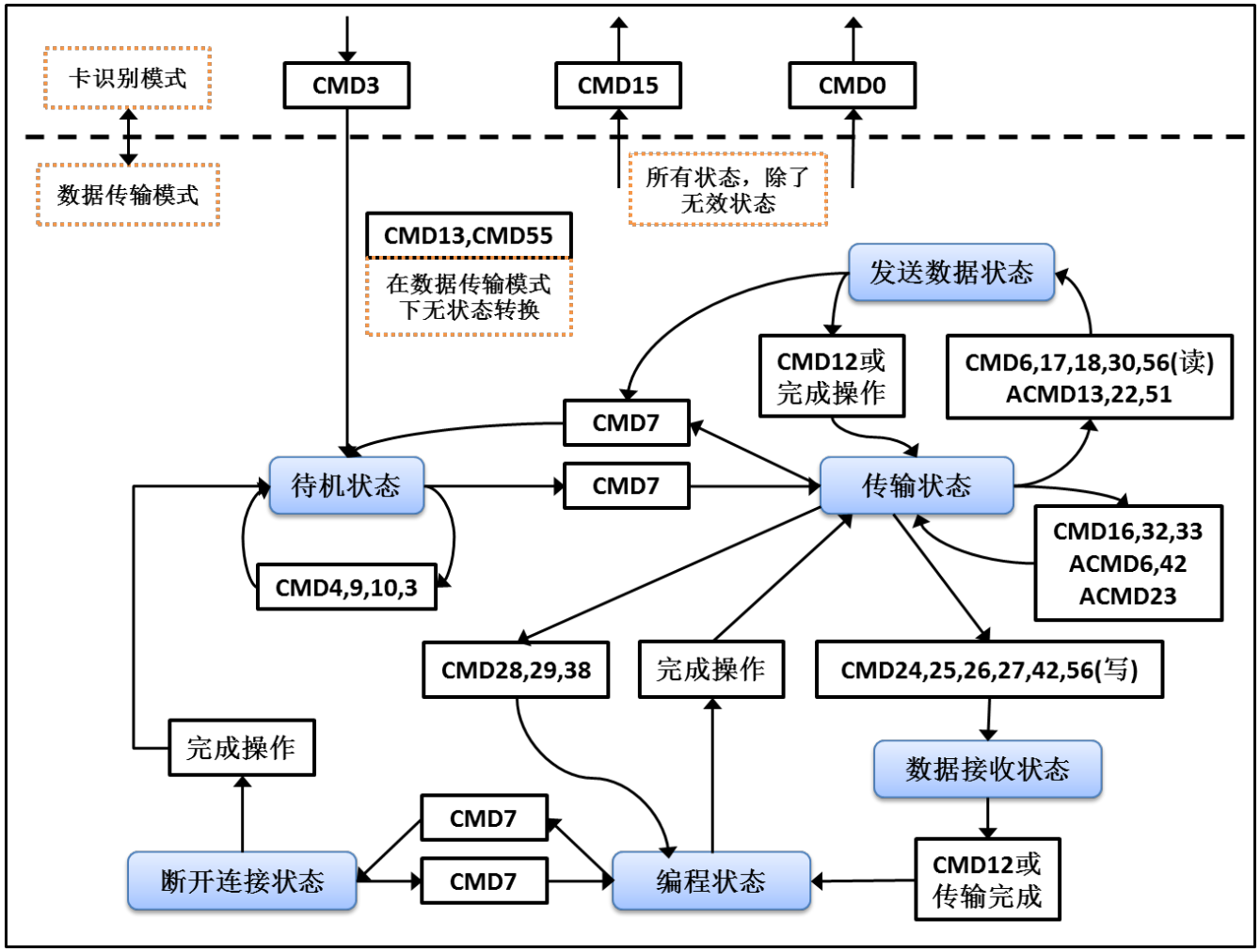


图 35-10 数据传输模式卡状态转换

CMD7 用来选定和取消指定的卡，卡在待机状态下还不能进行数据通信，因为总线上可能有多个卡都是出于待机状态，必须选择一个 RCA 地址目标卡使其进入传输状态才可以进行数据通信。同时通过 CMD7 命令也可以让已经被选择的目标卡返回到待机状态。

数据传输模式下的数据通信都是主机和目标卡之间通过寻址命令点对点进行的。卡处于传输状态下可以使用表 35-2 中面向块的读写以及擦除命令对卡进行数据读写、擦除。CMD12 可以中断正在进行的数据通信，让卡返回到传输状态。CMD0 和 CMD15 会中止任何数据编程操作，返回卡识别模式，这可能导致卡数据被损坏。

34.5 STM32 的 SDIO 功能框图

STM32 控制器有一个 SDIO，由两部分组成：SDIO 适配器和 AHB 接口，见图 35-11。SDIO 适配器提供 SDIO 主机功能，可以提供 SD 时钟、发送命令和进行数据传输。AHB 接口用于控制器访问 SDIO 适配器寄存器并且可以产生中断和 DMA 请求信号。

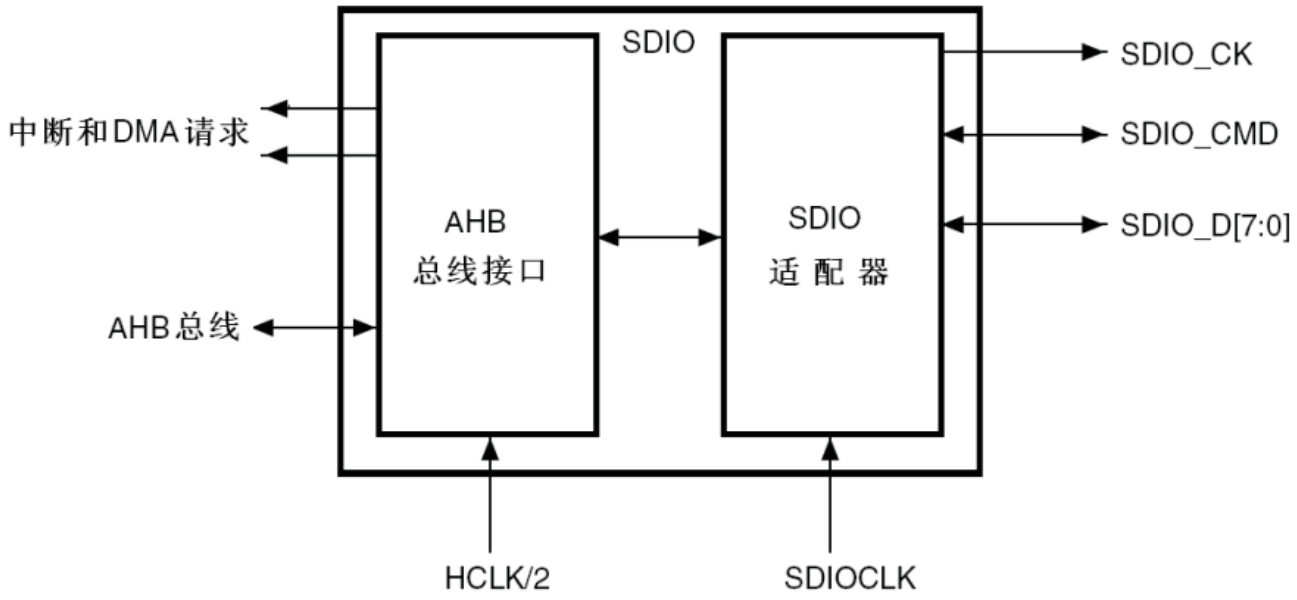


图 35-11 SDIO 功能框图

SDIO 使用两个时钟信号，一个是 SDIO 适配器时钟 (SDIOCLK=HCLK=72MHz)，另外一个是 AHB 总线时钟的二分频 (HCLK/2，一般为 36MHz)。适配器寄存器和 *FIFO* 使用 *AHB* 总线一侧的时钟 *(HCLK/2)*，控制单元、命令通道和数据通道使用 *SDIO* 适配器一侧的时钟 *(SDIOCLK)*。

SDIO_CK 是 SDIO 接口与 SD 卡用于同步的时钟信号。它使用 *SDIOCLK* 作为 *SDIO_CK* 的时钟来源，可以通过设置 *BYPASS* 模式直接得到，这时 *SDIO_CK* = *SDIOCLK=HCLK*。若禁止 BYPASS 模式，可以通过配置时钟寄存器的 *CLKDIV* 位控制分频因子，即 *SDIO_CK=SDIOCLK/*（*2+CLKDIV*）= *HCLK/*（*2+CLKDIV*）。配置时钟时要注意，*SD* 卡普遍要求 *SDIO_CK* 时钟频率不能超过 *25MHz*。

STM32 控制器的 SDIO 是针对 MMC 卡和 SD 卡的主设备，所以预留有 8 根数据线，对于 SD 卡最多用四根数据线。

SDIO 适配器是 SD 卡系统主机部分，是 STM32 控制器与 SD 卡数据通信中间设备。SDIO 适配器

由五个单元组成，分别是控制单元、命令路径单元、数据路径单元、寄存器单元以及 FIFO，见图 35-12。

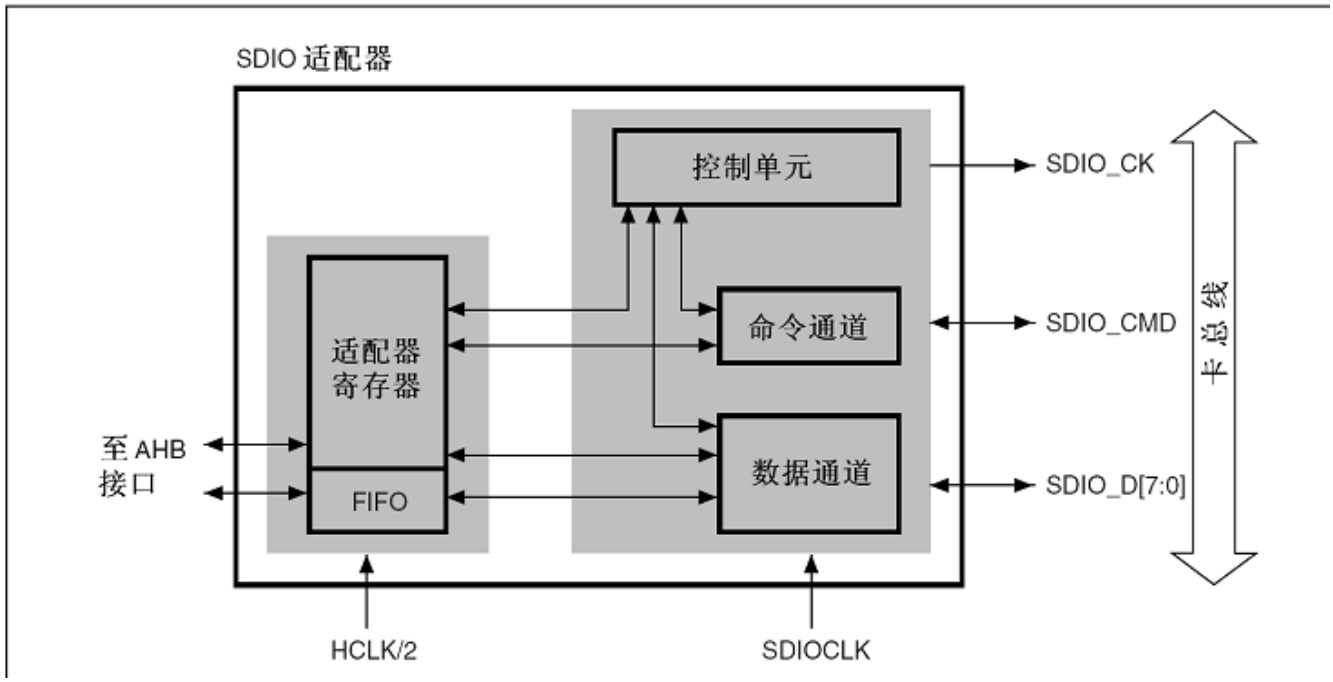


图 35-12 SDIO 适配器框图

34.5.1 控制单元

控制单元包含电源管理和时钟管理功能，结构如图 35-13。电源管理部件会在系统断电和上电阶段禁止 SD 卡总线输出信号。时钟管理部件控制 CLK 线时钟信号生成。一般使用 SDIOCLK 分频得到。

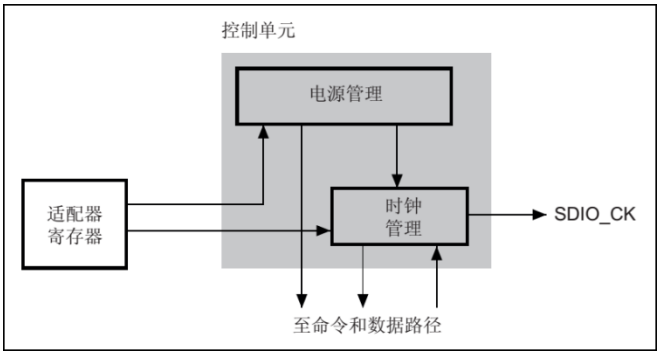


图 35-13 SDIO 适配器控制单元

34.5.2 命令路径

命令路径控制命令发送，并接收卡的响应，结构见图 *35_14*。

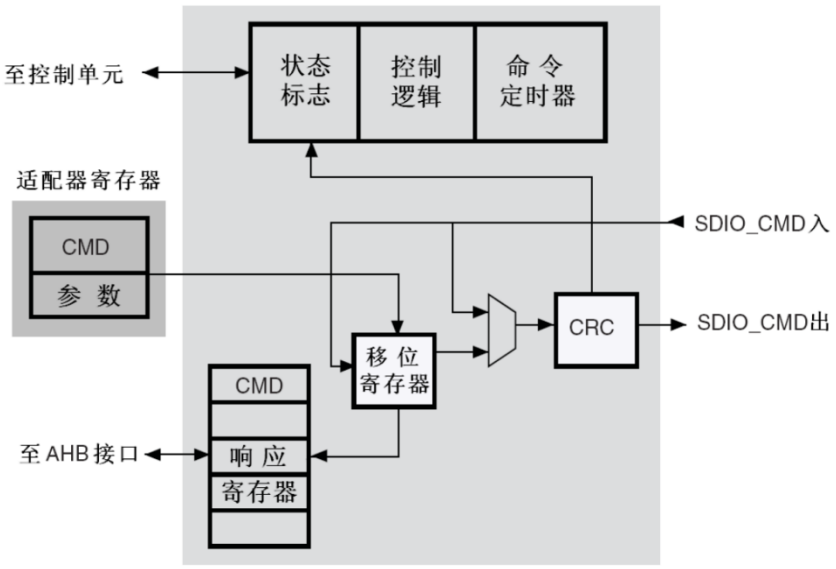


图 35-14 SDIO 适配器命令路径

关于 SDIO 适配器状态转换流程可以参考图 *35_9*，当 SD 卡处于某一状态时，SDIO 适配器必然处于特定状态与之对应。STM32 控制器以命令路径状态机 (CPSM) 来描述 SDIO 适配器的状态变化，并加入了等待超时检测功能，以便退出永久等待的情况。CPSM 的描述见图 *35_15*。

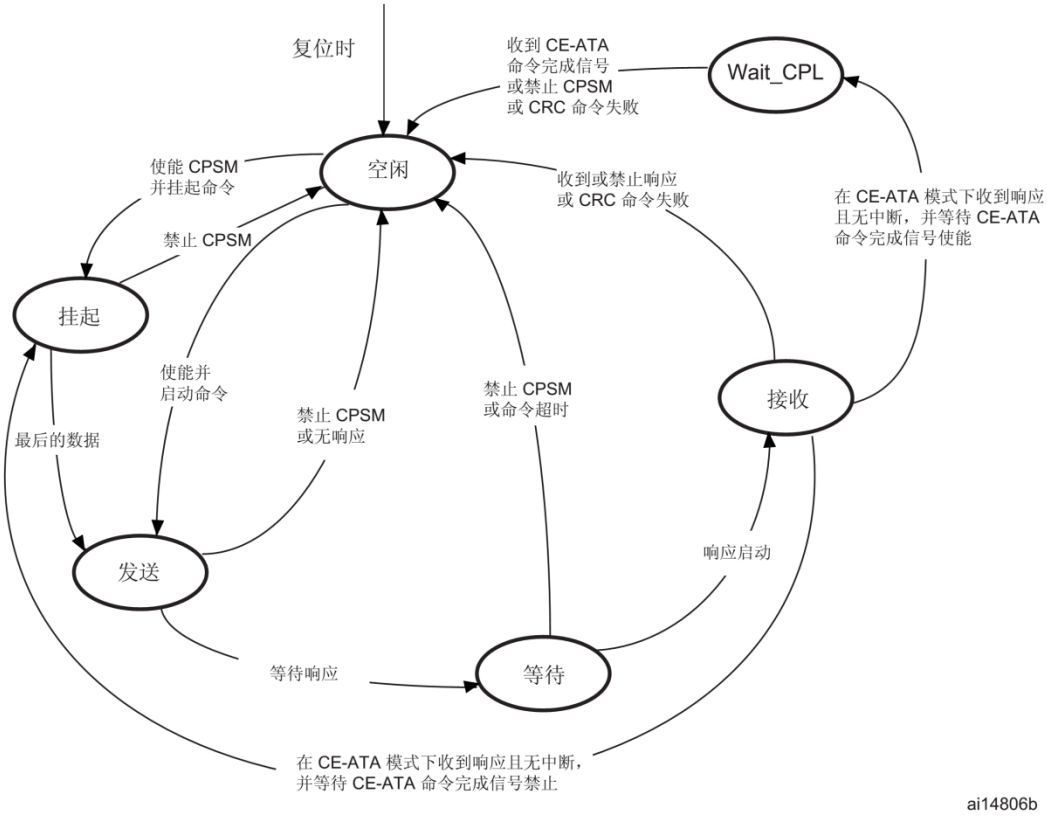


图 35-15 CPSM 状态机描述图

34.5.3 数据路径

数据路径部件负责与 SD 卡相互数据传输，内部结构见图 *35_16*。

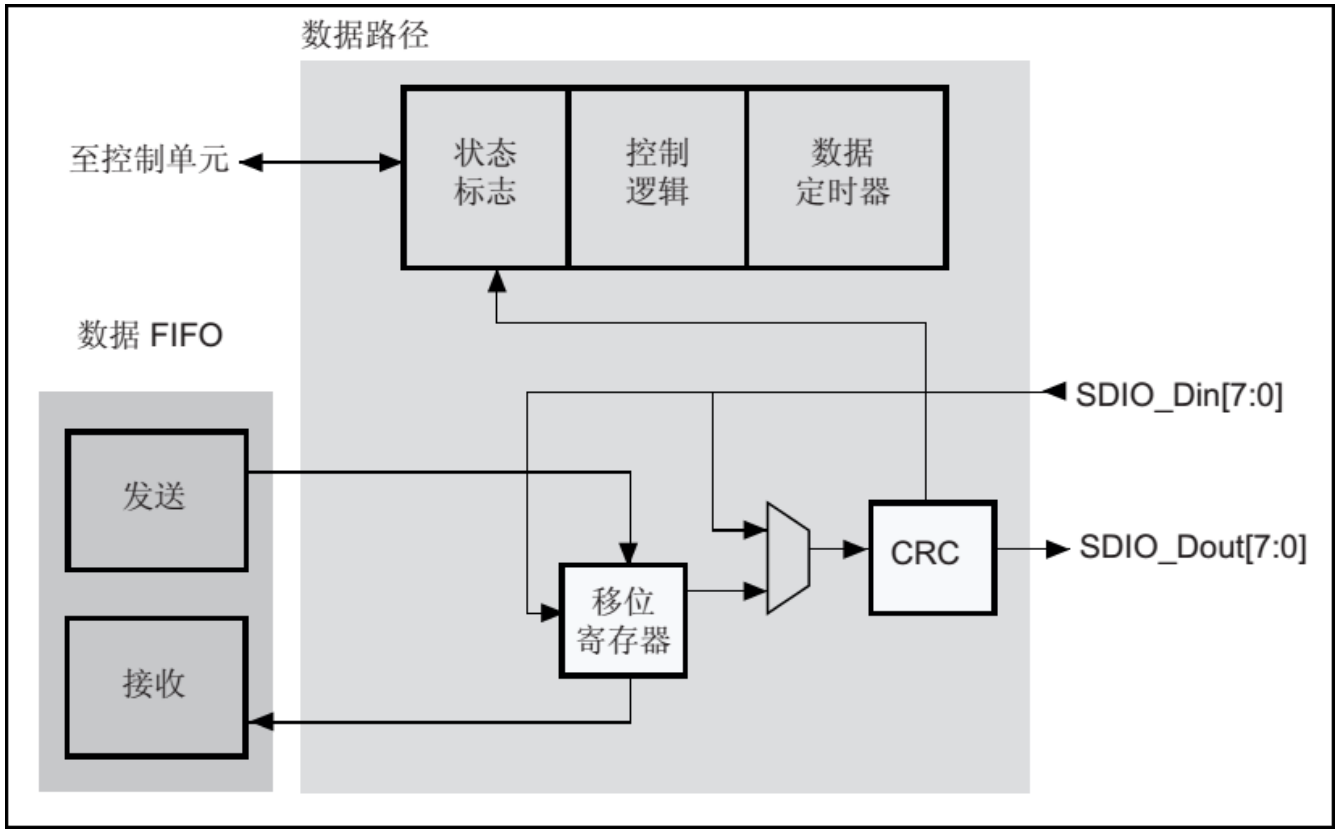


图 35-16 SDIO 适配器数据路径

SD 卡系统数据传输状态转换参考图 *35_10*，SDIO 适配器以数据路径状态机 (DPSM) 来描述 SDIO 适配器状态变化情况。并加入了等待超时检测功能，以便退出永久等待情况。发送数据时，DPSM 处于等待发送 (Wait_S) 状态，如果数据 FIFO 不为空，DPSM 变成发送状态并且数据路径部件启动向卡发送数据。接收数据时，DPSM 处于等待接收状态，当 DPSM 收到起始位时变成接收状态，并且数据路径部件开始从卡接收数据。DPSM 状态机描述见图 *35_17*。

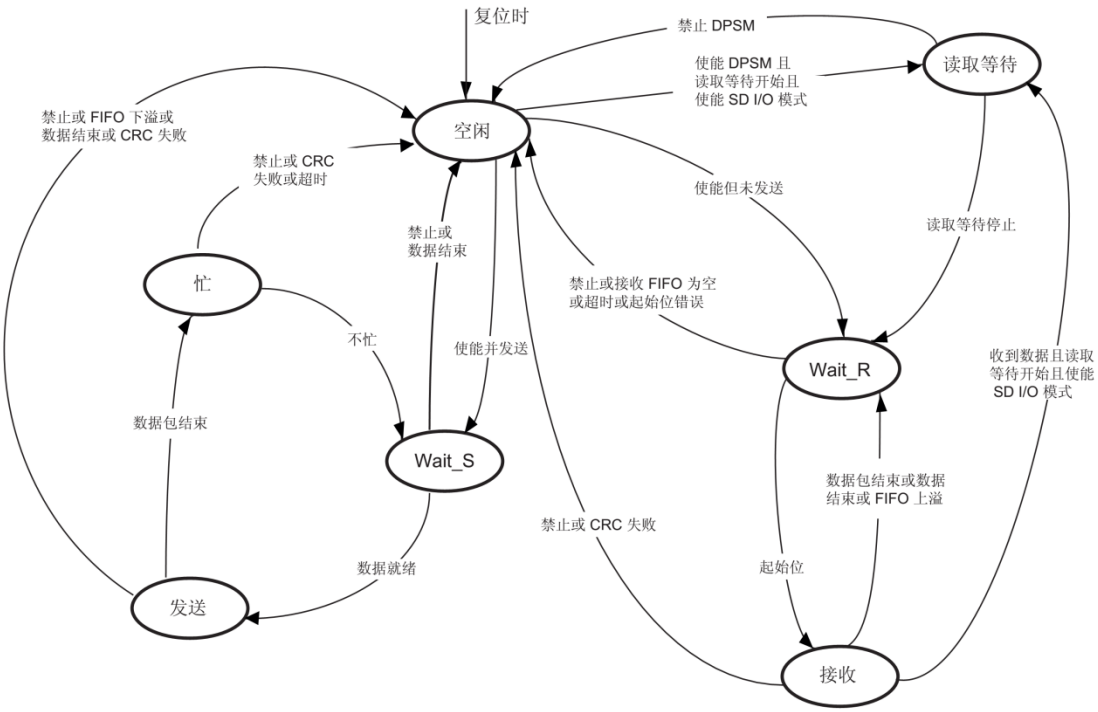


图 35-17 DPSM 状态机描述图

34.5.4 数据 FIFO

数据 FIFO(先进先出) 部件是一个数据缓冲器，带发送和接收单元。控制器的 FIFO 包含宽度为 32bit、深度为 32 字的数据缓冲器和发送/接收逻辑。其中 SDIO 状态寄存器 (SDIO_STA) 的 TXACT 位用于指示当前正在发送数据，RXACT 位指示当前正在接收数据，这两个位不可能同时为 1。

- 当 TXACT 为 1 时，可以通过 AHB 接口将数据写入到传输 FIFO。
- 当 RXACT 为 1 时，接收 FIFO 存放从数据路径部件接收到的数据。

根据 FIFO 空或满状态会把 SDIO_STA 寄存器位值 1，并可以产生中断和 DMA 请求。

34.5.5 适配器寄存器

适配器寄存器包含了控制 SDIO 外设的各种控制寄存器及状态寄存器，内容较多，可以通过 SDIO 提供的各种结构体来了解，这些寄存器的功能都被整合到了结构体或 STM32HAL 库之中。

34.6 SDIO 初始化结构体

HAL 库函数对 SDIO 外设建立了三个初始化结构体，分别为 SDIO 初始化结构体 SDIO_InitTypeDef、SDIO 命令初始化结构体 SDIO_CmdInitTypeDef 和 SDIO 数据初始化结构体 SDIO_DataInitTypeDef。这些结构体成员用于设置 SDIO 工作环境参数，并由 SDIO 相应初始化配置函数或功能函数调用，这些参数将会被写入到 SDIO 相应的寄存器，达到配置 SDIO 工作环境的目的。

初始化结构体和初始化库函数配合使用是 HAL 库精髓所在，理解了初始化结构体每个成员意义基本上就可以对该外设运用自如了。初始化结构体定义在 stm32f1xx_hal_sd.h 文件中，初始化库函数定义在 stm32f1xx_hal_sdio.c 文件中，编程时我们可以结合这两个文件内注释使用。

SDIO 初始化结构体用于配置 SDIO 基本工作环境，比如时钟分频、时钟沿、数据宽度等等。它被 SDIO_Init 函数使用。

列表 1: 代码清单 35-1 SDIO 初始化结构体

```
typedef struct {
    uint32_t ClockEdge;              // 时钟沿
    uint32_t ClockBypass;            // 旁路时钟
    uint32_t ClockPowerSave;         // 节能模式
    uint32_t BusWide;                // 数据宽度
    uint32_t HardwareFlowControl;    // 硬件流控制
    uint32_t ClockDiv;               // 时钟分频
} SDIO_InitTypeDef;
```

各结构体成员的作用介绍如下：

(1) ClockEdge：主时钟 SDIOCLK 产生 CLK 引脚时钟有效沿选择，可选上升沿或下降沿，它设定 SDIO 时钟控制寄存器 (SDIO_CLKCR) 的 NEGEDGE 位的值，一般选择设置为高电平。

(2) ClockBypass：时钟分频旁路使用，可选使能或禁用，它设定 SDIO_CLKCR 寄存器的 BYPASS 位。如果使能旁路，SDIOCLK 直接驱动 CLK 线输出时钟；如果禁用，使用 SDIO_CLKCR 寄存器的 CLKDIV 位值分频 SDIOCLK，然后输出到 CLK 线。一般选择禁用时钟分频旁路。

(3) ClockPowerSave：节能模式选择，可选使能或禁用，它设定 SDIO_CLKCR 寄存器的 PWRSAV 位的值。如果使能节能模式，CLK 线只有在总线激活时才有时钟输出；如果禁用节能模式，始终使能 CLK 线输出时钟。

(4) BusWide：数据线宽度选择，可选 1 位数据总线、4 位数据总线或 8 为数据总线，系统默认使用 1 位数据总线，操作 SD 卡时在数据传输模式下一般选择 4 位数据总线。它设定 SDIO_CLKCR 寄存器的 WIDBUS 位的值。

(5) HardwareFlowControl：硬件流控制选择，可选使能或禁用，它设定 SDIO_CLKCR 寄存器的 HWFC_EN 位的值。硬件流控制功能可以避免 FIFO 发送上溢和下溢错误。

(6) ClockDiv：时钟分频系数，它设定 SDIO_CLKCR 寄存器的 CLKDIV 位的值，设置 SDIOCLK 与 CLK 线输出时钟分频系数：

CLK 线时钟频率 =SDIOCLK/([CLKDIV+2])。

34.7 SDIO 命令初始化结构体

SDIO 命令初始化结构体用于设置命令相关内容，比如命令号、命令参数、响应类型等等。

列表 2: 代码清单 35-2 SDIO 命令初始化接口

```
typedef struct {
    uint32_t Argument; // 命令参数
    uint32_t CmdIndex; // 命令号
    uint32_t Response; // 响应类型
    uint32_t WaitForInterrupt; // 等待使能
```

(下页继续)

(续上页)

```
    uint32_t CPSM;      // 命令路径状态机
} SDIO_CmdInitTypeDef;
```

各个结构体成员介绍如下：

(1) Argument：作为命令的一部分发送到卡的命令参数，它设定 SDIO 参数寄存器 (SDIO_ARG) 的值。

(2) CmdIndex：命令号选择，它设定 SDIO 命令寄存器 (SDIO_CMD) 的 CMDINDEX 位的值。

(3) Response：响应类型，SDIO 定义两个响应类型：长响应和短响应。根据命令号选择对应的响应类型。SDIO 定义了四个 32 位的 SDIO 响应寄存器 (SDIO_RESPx,x=1..4)，短响应只用到 SDIO_RESP1。

(4) Wait：等待类型选择，有三种状态可选，一种是无等待状态，超时检测功能启动；一种是等待中断，另外一种是等待传输完成。它设定 SDIO_CMD 寄存器的 WAITPEND 位和 WAITINT 位的值。

(5) CPSM：命令路径状态机控制，可选使能或禁用 CPSM。它设定 SDIO_CMD 寄存器的 CPSMEN 位的值。

34.8 SDIO 数据初始化结构体

SDIO 数据初始化结构体用于配置数据发送和接收参数，比如传输超时、数据长度、传输模式等等。

列表 3: 代码清单 35-3 SDIO 数据初始化结构体

```
typedef struct {
    uint32_t DataTimeOut;     // 数据传输超时
    uint32_t DataLength;      // 数据长度
    uint32_t DataBlockSize;   // 数据块大小
    uint32_t TransferDir;     // 数据传输方向
```

(下页继续)

(续上页)

```
    uint32_t TransferMode;    // 数据传输模式
    uint32_t DPSM;            // 数据路径状态机
} SDIO_DataInitTypeDef;
```

各结构体成员介绍如下：

(1) DataTimeOut：设置数据传输以卡总线时钟周期表示的超时周期，它设定 SDIO 数据定时器寄存器 (SDIO_DTIMER) 的值。在 DPSM 进入 Wait_R 或繁忙状态后开始递减，直到 0 还处于以上两种状态则将超时状态标志置 1.

(2) DataLength：设置传输数据长度，它设定 SDIO 数据长度寄存器 (SDIO_DLEN) 的值。

(3) DataBlockSize：设置数据块大小，有多种尺寸可选，不同命令要求的数据块可能不同。它设定 SDIO 数据控制寄存器 (SDIO_DCTRL) 寄存器的 DBLOCKSIZE 位的值。

(4) TransferDir：数据传输方向，可选从主机到卡的写操作，或从卡到主机的读操作。它设定 SDIO_DCTRL 寄存器的 DTDIR 位的值。

(5) TransferMode：数据传输模式，可选数据块或数据流模式。对于 SD 卡操作使用数据块类型。它设定 SDIO_DCTRL 寄存器的 DTMODE 位的值。

(6) DPSM：数据路径状态机控制，可选使能或禁用 DPSM。它设定 SDIO_DCTRL 寄存器的 DTEN 位的值。要实现数据传输都必须使能 SDIO_DPSM。

34.9 SD 卡读写测试实验

SD 卡广泛用于便携式设备上，比如数码相机、手机、多媒体播放器等。对于嵌入式设备来说是一种重要的存储数据部件。类似于 SPI Flash 芯片数据操作，可以直接进行读写，也可以写入文件系统，然后使用文件系统读写函数，使用文件系统操作。本实验是进行 SD 卡最底层的数据读写操作，直接使用 SDIO 对 SD 卡进行读写，会损坏 SD 卡原本内容，导致数据丢失，实验前请注意备份 SD 卡的原内容。由于 SD 卡容量很大，我们平时使用的 SD 卡都是已经包含有文件系统的，一般不会使用本章的操作方式编写 SD 卡的应用，但它是 SD 卡操作的基础，对于原理学习是非常有必要的，在它的基础上移植文件系统到 SD 卡的应用将在下一章讲解。

34.9.1 硬件设计

STM32 控制器的 SDIO 引脚是被设计固定不变的，开发板设计采用四根数据线模式。对于命令线和数据线须需要加一个上拉电阻。

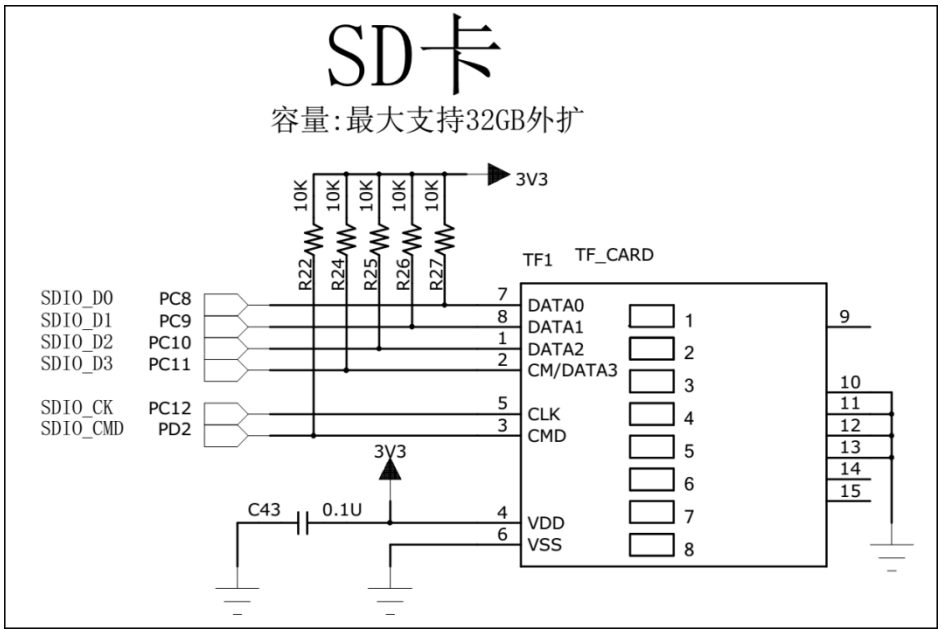


图 35-18a 指南者和霸道开发板的 SD 卡硬件设计（SDIO 接口）

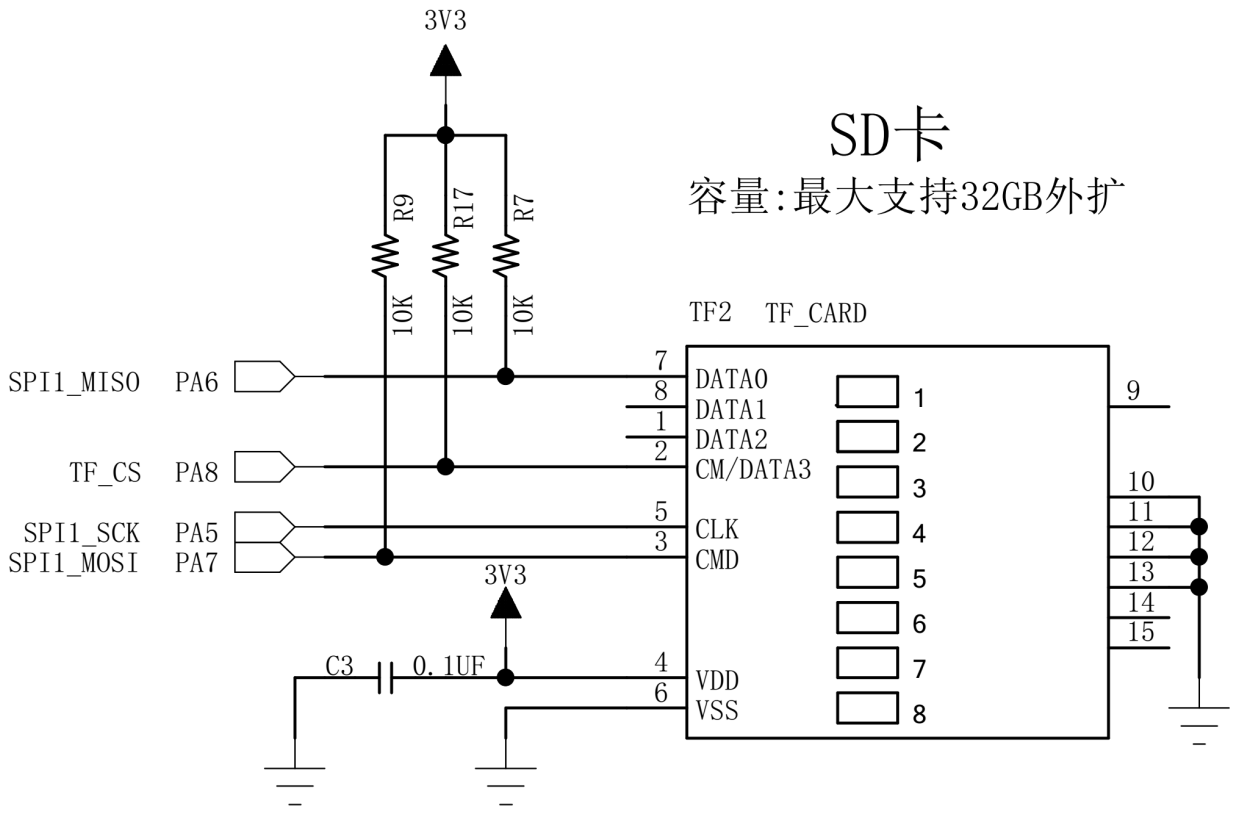


图 35-18b MINI 开发板的 SD 卡硬件设计（SPI 接口）

34.9.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等没有全部罗列出来，完整的代码请参考本章配套的工程。有了之前相关 SDIO 知识基础，我们就可以着手开始编写 SD 卡驱动程序了，根据之前内容，可了解操作的大概流程：

- 初始化相关 GPIO 及 SDIO 外设；
- 配置 SDIO 基本通信环境进入卡识别模式，通过几个命令处理后得到卡类型；
- 如果是可用卡就进入数据传输模式，接下来就可以进行读、写、擦除的操作。

虽然看起来只有三步，但它们有非常多的细节需要处理。实际上，SD 卡是非常常用外设部件，ST 公司在其测试板上也有板子 SD 卡卡槽，并提供了完整的驱动程序，我们直接参考移植使用即可。

类似 SDIO、USB 这些复杂的外设，它们的通信协议相当庞大，要自行编写完整、严谨的驱动不是一件轻松的事情，这时我们就可以利用 ST 官方例程的驱动文件，根据自己硬件移植到自己开发平台即可。

在“初识 STM32 HAL 库”章节我们重点讲解了 HAL 库的源代码及启动文件和库使用帮助文档这两部分内容，实际上“Utilities”文件夹内容是非常有参考价值的，该文件夹包含了基于 ST 官方实验板的驱动文件，比如 LCD、SDRAM、SD 卡、音频解码 IC 等等底层驱动程序，另外还有第三方软件库，如 emWin 图像软件库和 FatFs 文件系统。虽然，我们的开发平台跟 ST 官方实验平台硬件设计略有差别，但移植程序方法是完全可行的。学会移植程序可以减少很多工作量，加快项目进程，更何况 ST 官方的驱动代码是经过严格验证的。

在“STM32Cube_FW_F4_V1.19.0DriversBSP”文件路径下可以知道 SD 卡驱动文件，见图 35-19。我们需要 stm32746g_discovery_sd.c 和 stm32746g_discovery_sd.h 两个文件的完整内容。另外还需要 stm32746g_discovery.c 和 stm32746g_discovery.h 两个文件的部分代码内容，为简化工程，本章配置工程代码是将这两个文件需要用到的内容移植到 stm32746g_discovery_sd.c 文件中，具体可以参考工程文件。

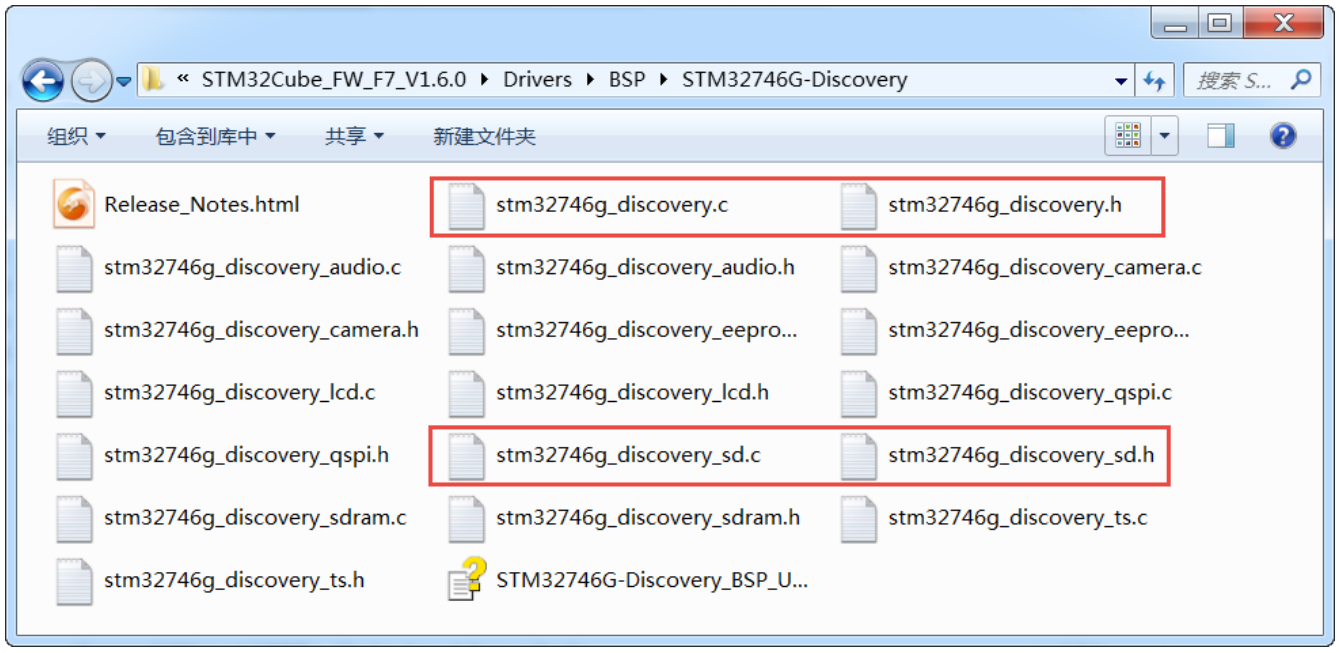


图 35-19 ST 官方实验板 SD 卡驱动文件

我们把 stm32746g_discovery_sd.c 和 stm32746g_discovery_sd.h 两个文件拷贝到我们的工程文件夹中，并将其对应改名为 bsp_sdio_sd.c 和 bsp_sdio_sd.h，见图 *35_20*。另外，sdio_test.c 和 sdio_test.h

文件包含了 SD 卡读、写、擦除测试代码。

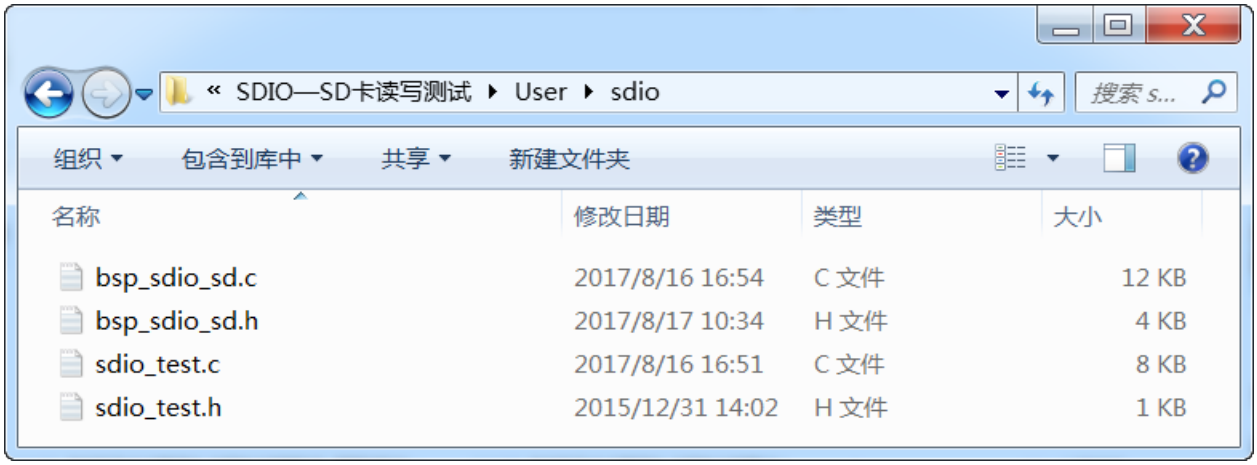


图 35-20 SD 卡驱动文件

34.9.2.1 GPIO 初始化和 DMA 配置

SDIO 用到 CLK 线、CMD 线和 4 根 DAT 线，使用之前必须初始化相关的 GPIO，并设置复用模式为 SDIO 的类型。而 SDIO 外设又支持生成 DMA 请求，使用 DMA 传输可以提高数据传输效率，因此在 SDIO 的控制代码中，可以把它设置为 DMA 传输模式或轮询模式，STM32HAL 库提供 SDIO 示例中针对这两个模式做了区分处理。由于应用中一般都使用 DMA 传输模式，所以接下来代码分析都以 DMA 传输模式介绍。

34.9.2.1.1 SDIO 模式、地址及时钟分频配置相关的宏定义

列表 4: 代码清单 35-4 SDIO 模式、地址及时钟分频配置相关的宏定义（bsp_sdio_sdcard.h 文件）

```
#define BSP_SD_CardInfo HAL_SD_CardInfoTypeDef

#define MSD_OK          ((uint8_t)0x00)
#define MSD_ERROR       ((uint8_t)0x01)
```

(下页继续)

(续上页)

```
/**
* @brief  SD transfer state definition
*/
#define  SD_TRANSFER_OK                 ((uint8_t)0x00)
#define  SD_TRANSFER_BUSY               ((uint8_t)0x01)

#define SD_DETECT_PIN                     GPIO_PIN_11
#define SD_DETECT_GPIO_PORT               GPIOF
#define __SD_DETECT_GPIO_CLK_ENABLE()     __HAL_RCC_GPIOF_CLK_ENABLE()
#define SD_DETECT_IRQn                    EXTI15_10_IRQn

#define SD_DATATIMEOUT           100000000U

#define SD_PRESENT               ((uint8_t)0x01)
#define SD_NOT_PRESENT           ((uint8_t)0x00)

/* DMA definitions for SD DMA transfer */
#define __DMAx_TxRx_CLK_ENABLE            __HAL_RCC_DMA2_CLK_ENABLE
#define SD_DMAx_Tx_INSTANCE               DMA2_Channel4
#define SD_DMAx_Rx_INSTANCE               DMA2_Channel4
#define SD_DMAx_Tx_IRQn                   DMA2_Channel4_5_IRQn
#define SD_DMAx_Rx_IRQn                   DMA2_Channel4_5_IRQn
#define BSP_SD_IRQHandler                 SDIO_IRQHandler
#define SD_DMAx_Tx_IRQHandler             DMA2_Channel4_5_IRQHandler
#define SD_DMAx_Rx_IRQHandler             DMA2_Channel4_5_IRQHandler
```

34.9.2.1.2 DMA 传输配置

列表 5: 代码清单 35-5 DMA 传输配置（bsp_sdio_sdcard.c 文件）

```
uint8_t SD_DMAConfigRx(SD_HandleTypeDef *hsd)
{
    static DMA_HandleTypeDef hdma_rx;
    HAL_StatusTypeDef status = HAL_ERROR;

    if (hsd->hdmarx == NULL) {
        /* Configure DMA Rx parameters */
        hdma_rx.Init.Direction           = DMA_PERIPH_TO_MEMORY;
        hdma_rx.Init.PeriphInc           = DMA_PINC_DISABLE;
        hdma_rx.Init.MemInc              = DMA_MINC_ENABLE;
        hdma_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
        hdma_rx.Init.MemDataAlignment    = DMA_MDATAALIGN_WORD;
        hdma_rx.Init.Priority            = DMA_PRIORITY_VERY_HIGH;

        hdma_rx.Instance = SD_DMAx_Rx_INSTANCE;

        /* Associate the DMA handle */
        __HAL_LINKDMA(hsd, hdmarx, hdma_rx);

        /* Stop any ongoing transfer and reset the state*/
        HAL_DMA_Abort(&hdma_rx);

        /* Deinitialize the Channel for new transfer */
        HAL_DMA_DeInit(&hdma_rx);

        /* Configure the DMA Channel */
        status = HAL_DMA_Init(&hdma_rx);
```

（下页继续）

(续上页)

```
        /* NVIC configuration for DMA transfer complete interrupt */
        HAL_NVIC_SetPriority(SD_DMAx_Rx_IRQn, 0xD, 0);
        HAL_NVIC_EnableIRQ(SD_DMAx_Rx_IRQn);
    } else {
        status = HAL_OK;
    }
    return (status != HAL_OK? MSD_ERROR : MMSD_OK);
}

/**
* @brief SD_DMAConfigTx
* @par Function Description
*   This function configure the DMA to transmit data to the SD card
* @retval
*  SD_ERROR or MSD_OK
*/
uint8_t SD_DMAConfigTx(SD_HandleTypeDef *hsd)
{
    static DMA_HandleTypeDef hdma_tx;
    HAL_StatusTypeDef status;

    if (hsd->hdmatx == NULL) {
        /* Configure DMA Tx parameters */
        hdma_tx.Init.Direction           = DMA_MEMORY_TO_PERIPH;
        hdma_tx.Init.PeriphInc           = DMA_PINC_DISABLE;
        hdma_tx.Init.MemInc              = DMA_MINC_ENABLE;
        hdma_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
        hdma_tx.Init.MemDataAlignment    = DMA_MDATAALIGN_WORD;
        hdma_tx.Init.Priority            = DMA_PRIORITY_VERY_HIGH;

        hdma_tx.Instance = SD_DMAx_Tx_INSTANCE;
```

(下页继续)

(续上页)

```
        /* Associate the DMA handle */
        __HAL_LINKDMA(hsd, hdmatx, hdma_tx);

        /* Stop any ongoing transfer and reset the state*/
        HAL_DMA_Abort(&hdma_tx);

        /* Deinitialize the Channel for new transfer */
        HAL_DMA_DeInit(&hdma_tx);

        /* Configure the DMA Channel */
        status = HAL_DMA_Init(&hdma_tx);

        /* NVIC configuration for DMA transfer complete interrupt */
        HAL_NVIC_SetPriority(SD_DMAx_Tx_IRQn, 0xD, 0);
        HAL_NVIC_EnableIRQ(SD_DMAx_Tx_IRQn);
    } else {
        status = HAL_OK;
    }
    return (status != HAL_OK? MSD_ERROR : MMSD_OK);
}
```

SD_DMA_RxConfig 函数用于配置 DMA 的 SDIO 接收请求参数，并指定接收存储器地址和大小。SD_DMA_TxConfig 函数用于配置 DMA 的 SDIO 发送请求参数，并指定发送存储器地址和大小。这两个函数在 SDIO 数据传输时会被调用来对 DMA 进行配置，使得传输过程采用 DMA 搬运数据。函数有两个输入参数，BufferSRC 在接收时用于指定接收到的数据存储的内存地址，发送时用于指定要对外发送的数据的内存地址；而 BufferSize 用于指定数据的大小，它是 32 个字节的，所以在配置 DMA 传输的数据大小时，要把它由字的单位转换成字节，也就是 1/4 大小；另外，DMA 配置中的外设地址在接收和发送时都是 SDIO 的 FIFO，在代码中使用了宏 SDIO_FIFO_ADDRESS 来定义。接收和发送函数非常类似，只是数据的方向和来源不一样而已，对于 DMA 相关配置的详细解释可以参考 DMA 章节内容。

34.9.2.2 相关类型定义

打开 bsp_sdio_sdcard.h 文件可以发现有非常多的枚举类型定义、结构体类型定义以及宏定义，把所有的定义在这里罗列出来不太现实，这部分代码内容请直接打开工程查看，针对这些内容在此处简要介绍如下：

- 枚举类型定义：有 MSD_ERROR、SDTransferState 和 SDCardState 三个。MSD_ERROR 是列举了控制器可能出现的错误、比如 CRC 校验错误、CRC 校验错误、通信等待超时、FIFO 上溢或下溢、擦除命令错误等等。这些错误类型部分是控制器系统寄存器的标志位，部分是通过命令的响应内容得到的。SDTransferState 定义了 SDIO 传输状态，有传输正常状态、传输忙状态和传输错误状态。SDCardState 定义卡的当前状态，比如准备状态、识别状态、待机状态、传输状态等等，具体状态转换过程参考图 35-9 和图 35-10。
- 结构体类型定义：有 SD_CSD、SD_CID、SD_CardStatus 以及 SD_CardInfo。SD_CSD 定义了 SD 卡的特定数据 (CSD) 寄存器位，一般提供 R2 类型的响应可以获取得到 CSD 寄存器内容。SD_CID 结构体类似 SD_CSD 结构体，它定义 SD 卡 CID 寄存器内容，也是通过 R2 响应类型获取得到。SD_CardStatus 结构体定义了 SD 卡状态，有数据宽度、卡类型、速度等级、擦除宽度、传输偏移地址等等 SD 卡状态。SD_CardInfo 结构体定义了 SD 卡信息，包括了 SD_CSD 类型和 SD_CID 类型成员，还有定义了卡容量、卡块大小、卡相对地址 RCA 和卡类型成员。
- 宏定义内容：包含有命令号定义、SDIO 传输方式、SD 卡插入状态以及 SD 卡类型定义。参考表 35-2 列举了描述了部分命令，文件中为每个命令号定义一个宏，比如将复位 CMD0 定义为 SD_CMD_GO_IDLE_STATE，这与表 35-2 中缩写部分是类似的，所以熟悉命名用途可以更好理解 SD 卡操作过程。SDIO 数据传输可以选择是否使用 DMA 传输，前面提到的 SD_DMA_MODE 和 SD_POLLING_MODE 就定义在这里，两种方式只能二选一使用，为提高系统性能，一般使用 DMA 传输模式。接下来还定义了检测 SD 卡是否正确插入的宏 SD_PRESENT 和 SD_NOT_PRESENT，ST 官方的原 SD 卡驱动是以一个输入引脚电平判断 SD 卡是否正确插入，由于我们的硬件没有使用该引脚，所以我们的程序里把 ST 驱动中原来的引脚检测部分代码删除掉了，但保留了 SD_PRESENT 和 SD_NOT_PRESENT 两个宏定义。最后定义 SD 卡具体的类型，有 V1.1 版本标准卡、V2.0 版本标准卡、高容量 SD 卡以及其他类型卡，前面三个是常用的类型。

在 bsp_sdio_sdcard.c 文件也有部分宏定义，这部分宏定义只能在该文件中使用。这部分宏定义包

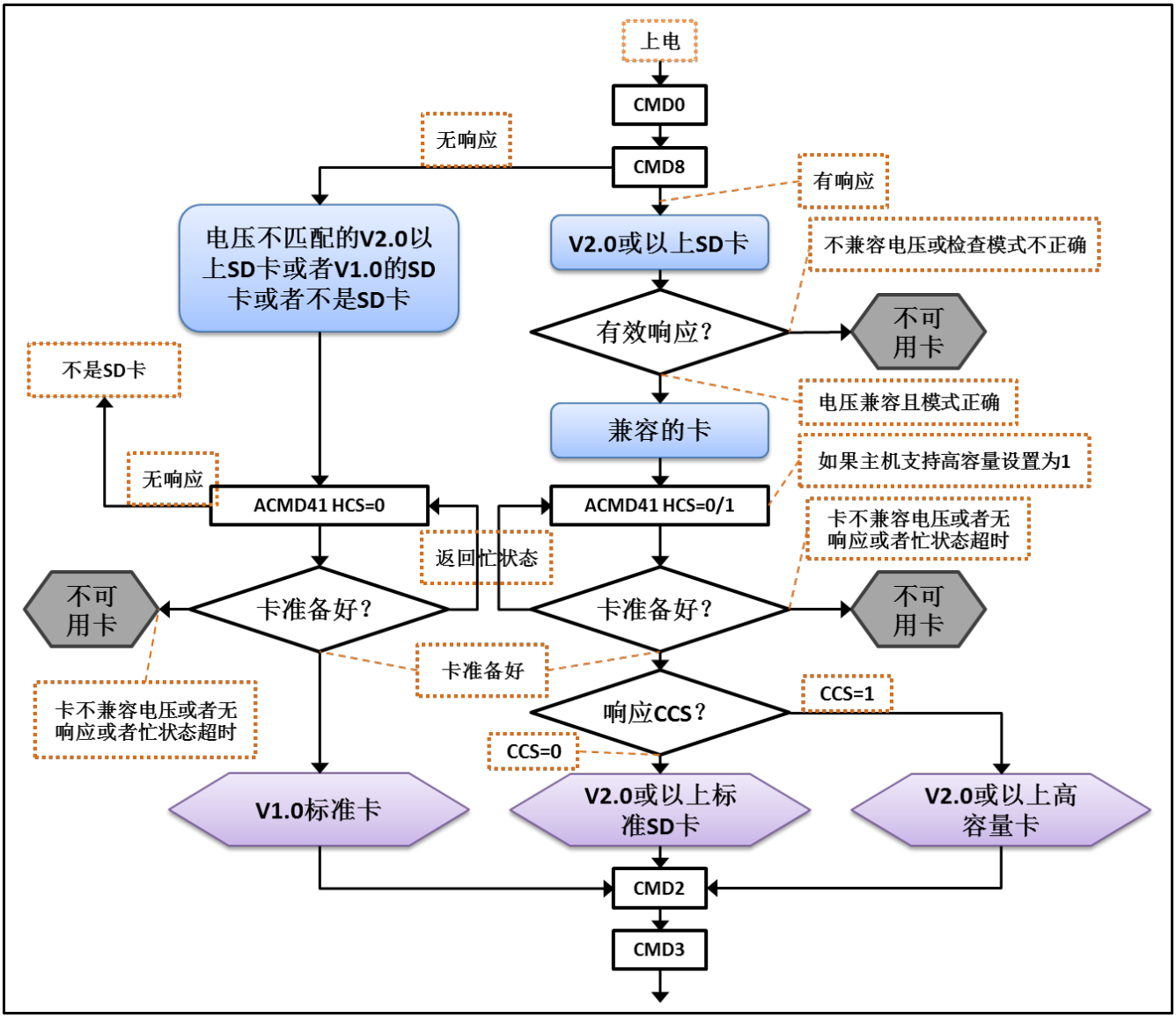
括命令超时时间定义、OCR 寄存器位掩码、R6 响应位掩码等等，这些定义更多是为提取特定响应位内容而设计的掩码。

因为类型定义和宏定义内容没有在本文中列举出来，读者有必要使用 KEIL 工具打开本章配套例程理解清楚。同时了解 bsp_sdio_sdcard.c 文件中定义的多个不同类型变量。

接下来我们就开始根据 SD 卡识别过程和数据传输过程理解 SD 卡驱动函数代码。这部分代码内容也是非常庞大，不可能全部在文档中全部列出，对于部分函数只介绍其功能。

34.9.2.3 SD 卡初始化

SD 卡初始化过程主要是卡识别和相关 SD 卡状态获取。整个初始化函数可以实现图 *35_21* 中的功能。



34.9.2.3.1 SD 卡初始化函数

列表 6: 代码清单 35-6 SD_Init 函数

```
uint8_t BSP_SD_Init(void)
{
```

(下页继续)

(续上页)

```
    uint8_t state = MMSD_OK;

    /*SD 设备接口配置 */
    uSdHandle.Instance = SDIO;
    uSdHandle.Init.ClockEdge           = SDIO_CLOCK_EDGE_RISING;
    uSdHandle.Init.ClockBypass         = SDIO_CLOCK_BYPASS_DISABLE;
    uSdHandle.Init.ClockPowerSave      = SDIO_CLOCK_POWER_SAVE_DISABLE;
    uSdHandle.Init.BusWide             = SDIO_BUS_WIDE_1B;
    uSdHandle.Init.HardwareFlowControl = SDIO_HARDWARE_FLOW_CONTROL_
↪DISABLE;
    uSdHandle.Init.ClockDiv            = SDIO_TRANSFER_CLK_DIV;

    /*  SD 初始化 */
    BSP_SD_MspInit(NULL);

    if (HAL_SD_Init(&uSdHandle) != HAL_OK) {
        state = MSD_ERROR;
    }

    /* 配置 SD 总线宽度 */
    if (state == MMSD_OK) {
        if (HAL_SD_ConfigWideBusOperation(&uSdHandle, SDIO_BUS_WIDE_4B)
            != HAL_OK) {
            state = MSD_ERROR;
        } else {
            state = MMSD_OK;
        }
    }
    return  state;
}
```

该函数的部分执行流程如下：

(1) 配置 NVIC，SD 卡通信用到 SDIO 中断，如果用到 DMA 传输还需要配置 DMA 中断。中断服务函数 SDIO_IRQHandler 定义在 stm32f1xx_it.c 文件的，为了移植方便，您也可以把它直接定义在 bsp_sdio_sdcard.c 文件中，中断服务函数定义在哪个文件问题都不大，只要定义正确就可以的，编译器会自动寻找，只是在移植的时候，要注意别漏掉该函数。

(2) 执行 GPIO_Configuration 函数，其功能是对底层 SDIO 引脚进行初始化以及开启相关时钟，该函数在之前已经讲解。

(3) SDIO_DeInit 函数用于解除初始化 SDIO 接口，它是与 GPIO_Configuration 函数相反功能，关闭相关时钟，关闭 SDIO 电源，让 SDIO 接近上电复位状态。恢复复位状态后再进行相关配置，可以防止部分没有配置的参数采用非默认值而导致错误，这是 ST 官方驱动常用的一种初始化方式。

(4) 调用 SD_PowerON 函数，它用于查询卡的工作电压和时钟控制配置，并返回 MSD_ERROR 类型错误，该函数是整个 SD 识别精髓，有必要详细分析。

34.9.2.3.2 SD_POWERON 函数

列表 7: 代码清单 35-7 SD_POWERON 函数

```
/*
* 函数名: SD_PowerON
* 描述  : 确保 SD 卡的工作电压和配置控制时钟
* 输入  : 无
* 输出  : -MSD_ERROR SD 卡错误代码
*          成功时则为 MSD_OK
* 调用  : 在 SD_Init() 调用
*/
MSD_ERROR SD_PowerON(void)
{
    MSD_ERROR errorstatus = MMSD_OK;
    uint32_t response = 0, count = 0, validvoltage = 0;
    uint32_t SDType = SD_STD_CAPACITY;
```

(下页继续)

(续上页)

```
    /*****************************************************************/
    /* 上电初始化
    * 配置 SDIO 的外设
    * SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_INIT_CLK_DIV)
    * 初始化时的时钟不能大于 400KHz
    */
    /* HCLK = 72MHz, SDIOCLK = 72MHz, SDIO_CK = HCLK/(178 + 2) = 400 KHz

    SDIO_InitStructure.SDIO_ClockDiv = SDIO_INIT_CLK_DIV;

    SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;

    /* 不使用 bypass 模式，直接用 HCLK 进行分频得到 SDIO_CK */
    SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;

    /* 空闲时不关闭时钟电源 */
    SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;

    /* 初始化的时候暂时先把数据线配置成 1 根 */
    SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;

    /* 禁止能硬件流控制 */
    SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
↪Disable;

    SDIO_Init(&SDIO_InitStructure);

    /* 开启 SDIO 外设的电源 */
    SDIO_SetPowerState(SDIO_PowerState_ON);

    /* 使能 SDIO 时钟 */
```

(下页继续)

(续上页)

```
    SDIO_ClockCmd(ENABLE);
    /*****************************************************************/
    /* 下面发送一系列命令，开始卡识别流程
    * CMD0: GO_IDLE_STATE(复位所以 SD 卡进入空闲状态)
    * 没有响应
    */
    SDIO_CmdInitStructure.SDIO_Argument = 0x0;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_GO_IDLE_STATE;

    /* 没有响应 */
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_No;

    /* 关闭等待中断 */
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;

    /* CPSM 在开始发送命令之前等待数据传输结束 */
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    /* 检测是否正确接收到 cmd0 */
    errorstatus = CmdError();

    /* 命令发送出错，返回 */
    if (errorstatus != MSD_OK) {          /* CMD 响应超时   */
        return (errorstatus);
    }
    /*****************************************************************/
    /* CMD8: SEND_IF_COND
    * 发送 CMD8 检查 SD 卡的电压操作条件
    *
    * 参数: - [31:12]: 保留 (要被设置为 '0')
    *       - [11:8] : 支持的电压 (VHS) 0x1 (范围: 2.7-3.6 V)
```

(下页继续)

(续上页)

```
*       - [7:0]  : 校验模式 (推荐 0xAA)
* 响应类型: R7
*/
/* 接收到命令 sd 会返回这个参数 */
SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN;

SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);

/* 检查是否接收到命令 */
errorstatus = CmdResp7Error();

/* 有响应则 card 遵循 sd 协议 2.0 版本 */
if (errorstatus == MSD_OK) {
    /* SD Card 2.0 ，先把它定义会 sdsc 类型的卡 */
    CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0;

    /* 这个变量用作 ACMD41 的参数，用来询问是 sdsc 卡还是 sdhc 卡 */
    SDType = SD_HIGH_CAPACITY;
} else {  /* 无响应，说明是 1.x 的或 mmc 的卡 */
    /* 发命令 CMD55 */
    SDIO_CmdInitStructure.SDIO_Argument = 0x00;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);
    errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
}
```

(下页继续)

(续上页)

```
    /* CMD55
    * 发送 cmd55，用于检测是 sd 卡还是 mmc 卡，或是不支持的卡
    * CMD 响应：R1
    */
    SDIO_CmdInitStructure.SDIO_Argument = 0x00;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    /* 是否响应，没响应的是 mmc 或不支持的卡 */
    errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
    /*********************************************************************/
    /* 若 errorstatus 为 Command TimeOut，说明是 MMC 卡
    * 若 errorstatus 为 MSD_OK ，说明是 SD card：SD 卡 2.0 (电压范围不匹配)
    * 或 SD 卡 1.x
    */
    if (errorstatus == MSD_OK) { //响应了 cmd55，是 sd 卡，可能为 1.x，可能为
2.0
        /* 下面开始循环地发送 sdio 支持的电压范围，循环一定次数 */

        /* SD CARD
        * 发送 ACMD41 SD_APP_OP_COND ，带参数 0x80100000
            */
        while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL)) {
            /* 在发送 ACMD 命令前都要先向卡发送 CMD55
            * 发送 CMD55 APP_CMD ， RCA 为 0
            */
            SDIO_CmdInitStructure.SDIO_Argument = 0x00;
            SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
```

(下页继续)

(续上页)

```
            SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
            SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
            SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
            SDIO_SendCommand(&SDIO_CmdInitStructure);

            errorstatus = CmdResp1Error(SD_CMD_APP_CMD);

            if (errorstatus != MSD_OK) {
                return (errorstatus);
            }

            /* ACMD41
          * 命令参数由支持的电压范围及 HCS 位组成，HCS 位置一来区分卡是 SDSC 还是 SDHC
            * 0:SDSC
            * 1:SDHC
            * 响应：R3，对应的是 OCR 寄存器
            */
            SDIO_CmdInitStructure.SDIO_Argument = SD_VOLTAGE_WINDOW_SD |␣
↪SDType;
            SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_APP_OP_COND;
            SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
            SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
            SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
            SDIO_SendCommand(&SDIO_CmdInitStructure);

            errorstatus = CmdResp3Error();

            if (errorstatus != MSD_OK) {
                return (errorstatus);
            }

            /* 若卡需求电压在 SDIO 的供电电压范围内，会自动上电并标志 pwr_up 位
```

(下页继续)

(续上页)

```
            * 读取卡寄存器，卡状态
            */
            response = SDIO_GetResponse(SDIO_RESP1);

            /* 读取卡的 ocr 寄存器的 pwr_up 位，看是否已工作在正常电压 */
            validvoltage = (((response >> 31) == 1) ? 1 : 0);
            count++;         /* 计算循环次数 */
        }

        if (count >= SD_MAX_VOLT_TRIAL) {  /* 循环检测超过一定次数还没上电 */
        errorstatus = SD_INVALID_VOLTRANGE; /* SDIO 不支持 card 的供电电压 */
            return (errorstatus);
        }

        /* 检查卡返回信息中的 HCS 位 */
        /* 判断 ocr 中的 ccs 位 ，如果是 sdsc 卡则不执行下面的语句 */
        if (response &= SD_HIGH_CAPACITY) {
    CardType = SDIO_HIGH_CAPACITY_SD_CARD; /* 把卡类型从初始化的 sdsc 型改为
sdhc 型 */
        }

    }/* else MMC Card */

    return (errorstatus);
}
```

SD_POWERON 函数执行流程如下：

(1) 配置 SDIO_InitStructure 结构体变量成员并调用 SDIO_Init 库函数完成 SDIO 外设的基本配置，注意此处的 SDIO 时钟分频，由于处于卡识别阶段，其时钟不能超过 400KHz。

(2) 调用 SDIO_SetPowerState 库函数控制 SDIO 的电源状态，给 SDIO 提供电源，并调用 SDIO_ClockCmd 库函数使能 SDIO 时钟。

(3) 发送命令给 SD 卡，首先发送 CMD0，复位所有 SD 卡，CMD0 命令无需响应，所以调用 CmdError 函数检测错误即可。CmdError 函数用于无需响应的命令发送检测，带有等待超时检测功能，它通过不断检测 SDIO_STA 寄存器的 CMDSENT 位即可知道命令发送成功与否。如果遇到超时错误则直接退出 SD_PowerON 函数。如果无错误则执行下面程序。

(4) 发送 CMD8 命令，检测 SD 卡支持的操作条件，主要就是电压匹配，CMD8 的响应类型是 R7，使用 CmdResp7Error 函数可获取得到 R7 响应结果，它是通过检测 SDIO_STA 寄存器相关位完成的，并具有等待超时检测功能。如果 CmdResp7Error 函数返回值为 MSD_OK，即 CMD8 有响应，可以判定 SD 卡为 V2.0 及以上的高容量 SD 卡，如果没有响应可能是 V1.1 版本卡或者是不可用卡。

(5) 使用 ACMD41 命令判断卡的具体类型。因为是 A 类命令，所以在发送 ACMD41 之前必须先发送 CMD55，CMD55 命令的响应类型的 R1。如果 CMD55 命令都没有响应说明是 MMC 卡或不可用卡。在正确发送 CMD55 之后就可以发送 ACMD41，并根据响应判断卡类型，ACMD41 的响应号为 R3，CmdResp3Error 函数用于检测命令正确发送并带有超时检测功能，但并不具备响应内容接收功能，需要在判定命令正确发送之后调用 SDIO_GetResponse 函数才能获取响应的内容。实际上，在有响应时，SDIO 外设会自动把响应存放在 SDIO_RESPx 寄存器中，SDIO_GetResponse 函数只是根据形参返回对应响应寄存器的值。通过判定响应内容值即可确定 SD 卡类型。

(6) 执行 SD_PowerON 函数无错误后就已经确定了 SD 卡类型，并说明卡和主机电压是匹配的，SD 卡处于卡识别模式下的准备状态。退出 SD_PowerON 函数返回 SD_Init 函数，执行接下来代码。判断执行 SD_PowerON 函数无错误后，执行下面的 SD_InitializeCards 函数进行与 SD 卡相关的初始化，使得卡进入数据传输模式下的待机模式。

34.9.2.3.3 SD_InitializeCards 函数

列表 8: 代码清单 35-8 SD_InitializeCards 函数

```
/*
* 函数名: SD_InitializeCards
* 描述   : 初始化所有的卡或者单个卡进入就绪状态
* 输入   : 无
```

(下页继续)

(续上页)

```
* 输出  : -MSD_ERROR SD 卡错误代码
*          成功时则为 MSD_OK
* 调用: 在 SD_Init() 调用, 在调用 power_on () 上电卡识别完毕后, 调用此函数进行卡初始化
*/
MSD_ERROR SD_InitializeCards(void)
{
    MSD_ERROR errorstatus = MSD_OK;
    uint16_t rca = 0x01;

    if (SDIO_GetPowerState() == SDIO_PowerState_OFF) {
        errorstatus = SD_REQUEST_NOT_APPLICABLE;
        return (errorstatus);
    }

    /* 判断卡的类型 */
    if (SDIO_SECURE_DIGITAL_IO_CARD != CardType) {
        /* Send CMD2 ALL_SEND_CID
        * 响应: R2, 对应 CID 寄存器
        */
        SDIO_CmdInitStructure.SDIO_Argument = 0x0;
        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ALL_SEND_CID;
        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
        SDIO_SendCommand(&SDIO_CmdInitStructure);

        errorstatus = CmdResp2Error();

        if (MSD_OK != errorstatus) {
            return (errorstatus);
        }
```

(下页继续)

(续上页)

```
        /* 将返回的 CID 信息存储起来 */
        CID_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
        CID_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
        CID_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
        CID_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
    }

/*****************************************************************/
    if (   (SDIO_STD_CAPACITY_SD_CARD_V1_1 == CardType)
        ||(SDIO_STD_CAPACITY_SD_CARD_V2_0 == CardType)
        ||(SDIO_SECURE_DIGITAL_IO_COMBO_CARD == CardType)
    ||(SDIO_HIGH_CAPACITY_SD_CARD == CardType) ) { /* 使用的是 2.0 的卡 */
        /* 发送 CMD3 SET_REL_ADDR , 带参数 0
        * 要求各个 SD 卡返回自身的 RCA 地址.
        * 响应: R6, 对应 RCA 寄存器
        */
        SDIO_CmdInitStructure.SDIO_Argument = 0x00;
        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR;
        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
        SDIO_SendCommand(&SDIO_CmdInitStructure);

        /* 把接收到的卡相对地址存起来 */
        errorstatus = CmdResp6Error(SD_CMD_SET_REL_ADDR, &rca);

        if (MSD_OK != errorstatus) {
            return (errorstatus);
        }
    }
    /*****************************************************************/
    if (SDIO_SECURE_DIGITAL_IO_CARD != CardType) {
```

(下页继续)

(续上页)

```
        RCA = rca;

        /* Send CMD9 SEND_CSD with argument as card's RCA
            * 响应:R2  对应寄存器 CSD(Card-Specific Data)
            */
        SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16);
        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
        SDIO_SendCommand(&SDIO_CmdInitStructure);

        errorstatus = CmdResp2Error();

        if (MSD_OK != errorstatus) {
            return (errorstatus);
        }

        CSD_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
        CSD_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
        CSD_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
        CSD_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
    }
    /*****************************************************************/
    /* 全部卡初始化成功 */
    errorstatus = MSD_OK;
    return (errorstatus);
}
```

SD_InitializeCards 函数执行流程如下：

(1) 判断 SDIO 电源是否启动，如果没有启动电源返回错误。

(2) SD 卡不是 SD I/O 卡时会进入 if 判断，执行发送 CMD2，CMD2 是用于通知所有卡通过 CMD 线返回 CID 值，执行 CMD2 发送之后就可以使用 CmdResp2Error 函数获取 CMD2 命令发送情况，发送无错误后即可以使用 SDIO_GetResponse 函数获取响应内容，它是个长响应，我们把 CMD2 响应内容存放在 CID_Tab 数组内。

(3) 发送 CMD2 之后紧接着就发送 CMD3，用于指示 SD 卡自行推荐 RCA 地址，CMD3 的响应为 R6 类型，CmdResp6Error 函数用于检查 R6 响应错误，它有两个形参，一个是命令号，这里为 CMD3，另外一个是 RCA 数据指针，这里使用 rca 变量的地址赋值给它，使得在 CMD3 正确响应之后 rca 变量即存放 SD 卡的 RCA。R6 响应还有一部分位用于指示卡的状态，CmdResp6Error 函数通用会对每个错误位进行必要的检测，如果发现有错误存在则直接返回对应错误类型。执行完 CmdResp6Error 函数之后返回到 SD_InitializeCards 函数中，如果判断无错误说明此刻 SD 卡已经处于数据传输模式。

(4) 发送 CMD9 给指定 RCA 的 SD 卡使其发送返回其 CSD 寄存器内容，这里的 RCA 就是在 CmdResp6Error 函数获取得到的 rca。最后把响应内容存放在 CSD_Tab 数组中。

执行 SD_InitializeCards 函数无错误后 SD 卡就已经处于数据传输模式下的待机状态，退出 SD_InitializeCards 后会返回前面的 SD_Init 函数，执行接下来代码，以下是 SD_Init 函数的后续执行过程：

(1) 重新配置 SDIO 外设，提高时钟频率，之前的卡识别模式都设定 CMD 线时钟为小于 400KHz，进入数据传输模式可以把时钟设置为小于 25MHz，以便提高数据传输速率。

(2) 调用 SD_GetCardInfo 函数获取 SD 卡信息，它需要一个指向 SD_CardInfo 类型变量地址的指针形参，这里赋值为 SDCardInfo 变量的地址。SD 卡信息主要是 CID 和 CSD 寄存器内容，这两个寄存器内容在 SD_InitializeCards 函数中都完成读取过程并将其分别存放在 CID_Tab 数组和 CSD_Tab 数组中，所以 SD_GetCardInfo 函数只是简单的把这两个数组内容整合复制到 SDCardInfo 变量对应成员内。正确执行 SD_GetCardInfo 函数后，SDCardInfo 变量就存放了 SD 卡的很多状态信息，这在之后应用中使用频率是很高的。

(3) 调用 SD_SelectDeselect 函数用于选择特定 RCA 的 SD 卡，它实际是向 SD 卡发送 CMD7。执行之后，卡就从待机状态转变为传输模式，可以说数据传输已经是万事俱备了。

(4) 扩展数据线宽度，之前的所有操作都是使用一根数据线传输完成的，使用 4 根数据线可以提高传输性能，调用可以设置数据线宽度，函数只有一个形参，用于指定数据线宽度。在 SD_EnableWideBusOperation 函数中，调用了 SDEnWideBus 函数使能使用宽数据线，然后传

输 SDIO_InitTypeDef 类型变量并使用 SDIO_Init 函数完成使用 4 根数据线配置。

至此，SD_Init 函数已经全部执行完成。如果程序可以正确执行，接下来就可以进行 SD 卡读写以及擦除等操作。虽然 bsp_sdio_sd.c 文件看起来非常长，但在 SD_Init 函数分析过程就已经涉及到它差不多一半内容了，另外一半内容主要就是读、写或擦除相关函数。

34.9.2.4 SD 卡数据操作

SD 卡数据操作一般包括数据读取、数据写入以及存储区擦除。数据读取和写入都可以分为单块操作和多块操作。

34.9.2.4.1 擦除函数

列表 9: 代码清单 35-9 SD_Erase 函数

```
/**
* @brief  控制 SD 卡擦除指定的数据区域
* @param  startaddr: 擦除的开始地址
* @param  endaddr: 擦除的结束地址
* @retval MSD_ERROR: SD 返回的错误代码.
*/
MSD_ERROR SD_Erase(uint32_t startaddr, uint32_t endaddr)
{
    MSD_ERROR errorstatus = MSD_OK;
    uint32_t delay = 0;
    __IO uint32_t maxdelay = 0;
    uint8_t cardstate = 0;

    /*!< 检查 SD 卡是否支持擦除操作 */
    if (((CSD_Tab[1] >> 20) & SD_CCCC_ERASE) == 0) {
        errorstatus = SD_REQUEST_NOT_APPLICABLE;
        return (errorstatus);
```

(下页继续)

(续上页)

```
    }
    //延时，根据时钟分频设置来计算
    maxdelay = 120000 / ((SDIO->CLKCR & 0xFF) + 2);

    if (SDIO_GetResponse(SDIO_RESP1) & SD_CARD_LOCKED) { //卡已上锁
        errorstatus = SD_LOCK_UNLOCK_FAILED;
        return (errorstatus);
    }

    if (CardType == SDIO_HIGH_CAPACITY_SD_CARD) { //sdhc 卡
        //在 sdhc 卡中，地址参数为块地址，每块 512 字节，而 sdsc 卡地址为字节地址
        //所以若是 sdhc 卡要对地址/512 进行转换
        startaddr /= 512;
        endaddr /= 512;
    }

    /*!<  ERASE_GROUP_START (CMD32) 设置擦除的起始地址,
    erase_group_end(CMD33) 设置擦除的结束地址,  */
    if ((SDIO_STD_CAPACITY_SD_CARD_V1_1 == CardType) ||
        (SDIO_STD_CAPACITY_SD_CARD_V2_0 == CardType) ||
        (SDIO_HIGH_CAPACITY_SD_CARD == CardType)) {
        /*!< 发送命令 CMD32 SD_ERASE_GRP_START , 带参数 startaddr  */
        SDIO_CmdInitStructure.SDIO_Argument = startaddr;
        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_ERASE_GRP_START;
        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;  //R1
        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
        SDIO_SendCommand(&SDIO_CmdInitStructure);

        errorstatus = CmdResp1Error(SD_CMD_SD_ERASE_GRP_START);
        if (errorstatus != MSD_OK) {
            return (errorstatus);
```

(下页继续)

(续上页)

```
        }

        /*!< 发送命令 CMD33 SD_ERASE_GRP_END , 带参数 endaddr  */
        SDIO_CmdInitStructure.SDIO_Argument = endaddr;
        SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_ERASE_GRP_END;
        SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
        SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
        SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
        SDIO_SendCommand(&SDIO_CmdInitStructure);

        errorstatus = CmdResp1Error(SD_CMD_SD_ERASE_GRP_END);
        if (errorstatus != MSD_OK) {
            return (errorstatus);
        }
    }

    /*!< 发送 CMD38 ERASE 命令, 开始擦除 */
    SDIO_CmdInitStructure.SDIO_Argument = 0;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ERASE;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    errorstatus = CmdResp1Error(SD_CMD_ERASE);

    if (errorstatus != MSD_OK) {
        return (errorstatus);
    }

    for (delay = 0; delay < maxdelay; delay++) {
    }
```

(下页继续)

(续上页)

```
    /*!< 等待 SD 卡的内部时序操作完成 */
    errorstatus = IsCardProgramming(&cardstate);

    while ((errorstatus == MSD_OK) && ((SD_CARD_PROGRAMMING == cardstate)␣
↪|| (SD_CARD_RECEIVING == cardstate)))
    {
        errorstatus = IsCardProgramming(&cardstate);
    }

    return (errorstatus);
}
```

SD_Erase 函数用于擦除 SD 卡指定地址范围内的数据。该函数接收两个参数，一个是擦除的起始地址，另外一个是擦除的结束地址。对于高容量 SD 卡都是以块大小为 512 字节进行擦除的，所以保证字节对齐是程序员的责任。SD_Erase 函数的执行流程如下：

(1) 检查 SD 卡是否支持擦除功能，如果不支持则直接返回错误。为保证擦除指令正常进行，要求主机一个遵循下面的命令序列发送指令：CMD32->CMD33->CMD38。如果发送顺序不对，SD 卡会设置 ERASE_SEQ_ERROR 位到状态寄存器。

(2) SD_Erase 函数发送 CMD32 指令用于设定擦除块开始地址，在执行无错误后发送 CMD33 设置擦除块的结束地址。

(3) 发送擦除命令 CMD38，使得 SD 卡进行擦除操作。SD 卡擦除操作由 SD 卡内部控制完成，不同卡擦除后是 0xff 还是 0x00 由厂家决定。擦除操作需要花费一定时间，这段时间不能对 SD 卡进行其他操作。

(4) 通过 IsCardProgramming 函数可以检测 SD 卡是否处于编程状态 (即卡内部的擦写状态)，需要确保 SD 卡擦除完成才退出 SD_Erase 函数。IsCardProgramming 函数先通过发送 CMD13 命令 SD 卡发送它的状态寄存器内容，并对响应内容进行分析得出当前 SD 卡的状态以及可能发送的错误。

34.9.2.4.2 数据写入操作

数据写入可分为单块数据写入和多块数据写入，这里只分析单块数据写入，多块的与之类似。SD 卡数据写入之前并没有硬性要求擦除写入块，这与 SPI Flash 芯片写入是不同的。ST 官方的 SD 卡写入函数包括扫描查询方式和 DMA 传输方式，我们这里只介绍 DMA 传输模式。

列表 10: 代码清单 35-10 SD_WriteBlock 函数

```
/**
* @brief  向 sd 卡写入一个 BLOCK 的数据 (512 字节)
* @note   本函数使用后需要调用如下两个函数来等待数据传输完成
*          - SD_WaitWriteOperation(): 确认 DMA 已把数据传输到 SDIO 接口
*          - SD_GetStatus(): 确认 SD 卡内部已经把数据写入完毕
* @param  writebuff: 指向要写入的数据
* @param  WriteAddr: 要把数据写入到 sd 卡的地址
* @param  BlockSize: 块大小, sdhc 卡为 512 字节
* @retval MSD_ERROR: 返回的 sd 错误代码
*/
MSD_ERROR SD_WriteBlock(uint8_t *writebuff, uint32_t WriteAddr, uint16_t␣
↪BlockSize)
{
    MSD_ERROR errorstatus = MSD_OK;

    TransferError = MSD_OK;
    TransferEnd = 0;
    StopCondition = 0;

    SDIO->DCTRL = 0x0;

    if (CardType == SDIO_HIGH_CAPACITY_SD_CARD) {
        BlockSize = 512;
        WriteAddr /= 512;
    }
```

(下页继续)

(续上页)

```
/*----以下这段是在 st 驱动库上添加的 , 没有这一段容易卡死在 DMA 检测中 ---*/
/* 设置块 BLOCK 的大小, cmd16,
* 若是 sdsc 卡, 可以用来设置块大小,
* 若是 sdhc 卡, 块大小为 512 字节, 不受 cmd16 影响
*/
SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);

errorstatus = CmdResp1Error(SD_CMD_SET_BLOCKLEN);

if (MSD_OK != errorstatus) {
    return (errorstatus);
}
/*************************************************************/

/*!< 发送 CMD24 WRITE_SINGLE_BLOCK 写入 */
SDIO_CmdInitStructure.SDIO_Argument = WriteAddr;     //写入地址
SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_WRITE_SINGLE_BLOCK;
SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;    //r1
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);

errorstatus = CmdResp1Error(SD_CMD_WRITE_SINGLE_BLOCK);

if (errorstatus != MSD_OK) {
    return (errorstatus);
```

(下页继续)

(续上页)

```
    }

    //配置 sdio 的写数据寄存器
    SDIO_DataInitStructure.SDIO_DataTimeOut = SD_DATATIMEOUT;
    SDIO_DataInitStructure.SDIO_DataLength = BlockSize;
    SDIO_DataInitStructure.SDIO_DataBlockSize = (uint32_t) 9 << 4;  //512
字节
    SDIO_DataInitStructure.SDIO_TransferDir = SDIO_TransferDir_ToCard;//写
数据
    SDIO_DataInitStructure.SDIO_TransferMode = SDIO_TransferMode_Block;
    SDIO_DataInitStructure.SDIO_DPSM = SDIO_DPSM_Enable;   //开启数据通道状态
机
    SDIO_DataConfig(&SDIO_DataInitStructure);

    SDIO_ITConfig(SDIO_IT_DATAEND, ENABLE);  //数据传输结束中断
    SD_DMA_TxConfig((uint32_t *)writebuff, BlockSize); //配置 dma，跟 rx 类似
    SDIO_DMACmd(ENABLE);   // 使能 sdio 的 dma 请求

    return (errorstatus);
}
```

SD_WriteBlock 函数用于向指定的目标地址写入一个块的数据，它有三个形参，分别为指向待写入数据的首地址的指针变量、目标写入地址和块大小。块大小一般都设置为 512 字节。SD_WriteBlock 写入函数的执行流程如下：

(1) SD_WriteBlock 函数开始时将 SDIO 数据控制寄存器 (SDIO_DCTRL) 清零，复位之前的传输设置。

(2) 对 SD 卡进行数据读写之前，都必须发送 CMD16 指定块的大小，对于标准卡，要写入 Block-Size 长度字节的块；对于 SDHC 卡，写入固定为 512 字节的块。接下来就可以发送块写入命令 CMD24 通知 SD 卡要进行数据写入操作，并指定待写入数据的目标地址。

(3) 利用 SDIO_DataInitTypeDef 结构体类型变量配置数据传输的超时、块数量、数据块大小、数据传输方向等参数并使用 SDIO_DataConfig 函数完成数据传输环境配置。

(4) 调用 SDIO_ITConfig 函数使能 SDIO 数据结束传输结束中断，传输结束时，会跳转到 SDIO 的中断服务函数运行。

(5) 调用前面讲解的 SD_DMA_TxConfig 函数，配置使能 SDIO 数据向 SD 卡的数据传输的 DMA 请求，该函数可以参考*代码清单 35_5*。为使 SDIO 发送 DMA 请求，需要调用 SDIO_DMACmd 函数使能。对于高容量的 SD 卡要求块大小必须为 512 字节，程序员有责任保证数据写入地址与块大小的字节对齐问题。

执行完以上代码后，SDIO 外设会自动生成 DMA 发送请求，将指定数据使用 DMA 传输写入到 SD 卡内。

34.9.2.4.3 写入操作等待函数

SD_WaitWriteOperation 函数用于检测和等待数据写入完成，在调用数据写入函数之后一般都需要调用，SD_WaitWriteOperation 函数适用于单块及多块写入函数。

列表 11: 代码清单 35-11 SD_WaitWriteOperation 函数

```
/**
* @brief  本函数会一直等待到 DMA 传输结束
*         在 SDIO_WriteBlock() 和 SDIO_WriteMultiBlocks() 函数后
    必须被调用以确保 DMA 数据传输完成
* @param  None.
* @retval MSD_ERROR: 返回的 sd 错误代码.
*/
MSD_ERROR SD_WaitWriteOperation(void)
{
    MSD_ERROR errorstatus = MSD_OK;
    //等待 dma 是否传输结束
    while ((SD_DMAEndOfTransferStatus() == RESET) &&
        (TransferEnd == 0) && (TransferError == MSD_OK)) {
    }
```

(下页继续)

(续上页)

```
    if (TransferError != MSD_OK) {
        return (TransferError);
    }

    /*!< 清除标志 */
    SDIO_ClearFlag(SDIO_STATIC_FLAGS);

    return (errorstatus);
}
```

上述代码调用库函数 SD_DMAEndOfTransferStatus 一直检测 DMA 的传输完成标志，当 DMA 传输结束时，该函数会返回 SET 值。另外，while 循环中的判断条件使用的 TransferEnd 和 TransferError 是全局变量，它们会在 SDIO 的中断服务函数根据传输情况被设置，传输结束后，根据 TransferError 的值来确认是否正确传输，若不正确则直接返回错误代码。SD_WaitWriteOperation 函数最后是清除相关标志位并返回错误。由于这个函数里的 while 循环的存在，它会确保 DMA 的传输结束。

34.9.2.4.4 数据读取操作

同向 SD 卡写入数据类似，从 SD 卡读取数据可分为单块读取和多块读取。这里仅介绍单块读操作函数，多块读操作类似。

列表 12: 代码清单 35-12 SD_ReadBlock 函数

```
/**
* @brief  从 sd 卡读取一个 BLOCK 的数据 (512 字节)
* @note   本函数使用后需要调用如下两个函数来等待数据传输完成
*          - SD_ReadWaitOperation(): 确认 DMA 已从 SDIO 传输到数据到内存
*          - SD_GetStatus(): 确认 SD 卡传输完成
* @param  writebuff: 指向要接收数据的缓冲区
* @param  WriteAddr: 要把数据写入到 sd 卡的地址
* @param  BlockSize: 块大小, sdhc 卡为 512 字节
```

(下页继续)

(续上页)

```
* @retval MSD_ERROR: 返回的 sd 错误代码
*/
MSD_ERROR SD_ReadBlock(uint8_t *readbuff, uint32_t ReadAddr, uint16_t␣
↪BlockSize)
{
    MSD_ERROR errorstatus = MSD_OK;

    TransferError = MSD_OK;
    TransferEnd = 0;    //传输结束标置位，在中断服务置 1
    StopCondition = 0;

    SDIO->DCTRL = 0x0;

    if (CardType == SDIO_HIGH_CAPACITY_SD_CARD) {
        BlockSize = 512;
        ReadAddr /= 512;
    }
    /**********************add，没有这一段容易卡死在 DMA 检测中 *************/
    /* Set Block Size for Card，cmd16，
    * 若是 sdsc 卡，可以用来设置块大小，
    * 若是 sdhc 卡，块大小为 512 字节，不受 cmd16 影响
    */
    SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;    //r1
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    errorstatus = CmdResp1Error(SD_CMD_SET_BLOCKLEN);

    if (MSD_OK != errorstatus) {
```

(下页继续)

(续上页)

```
        return (errorstatus);
    }
    /*************************************************************/
    SDIO_DataInitStructure.SDIO_DataTimeOut = SD_DATATIMEOUT;
    SDIO_DataInitStructure.SDIO_DataLength = BlockSize;
    SDIO_DataInitStructure.SDIO_DataBlockSize = (uint32_t) 9 << 4;
    SDIO_DataInitStructure.SDIO_TransferDir = SDIO_TransferDir_ToSDIO;
    SDIO_DataInitStructure.SDIO_TransferMode = SDIO_TransferMode_Block;
    SDIO_DataInitStructure.SDIO_DPSM = SDIO_DPSM_Enable;
    SDIO_DataConfig(&SDIO_DataInitStructure);

    /*!< 发送 CMD17 READ_SINGLE_BLOCK 命令 */
    SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)ReadAddr;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_READ_SINGLE_BLOCK;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    errorstatus = CmdResp1Error(SD_CMD_READ_SINGLE_BLOCK);

    if (errorstatus != MSD_OK) {
        return (errorstatus);
    }

    SDIO_ITConfig(SDIO_IT_DATAEND, ENABLE);
    SDIO_DMACmd(ENABLE);
    SD_DMA_RxConfig((uint32_t *)readbuff, BlockSize);

    return (errorstatus);
}
```

数据读取操作与数据写入操作编程流程是类似，只是数据传输方向改变，使用到的 SD 命令号也有所不同而已。SD_ReadBlock 函数有三个形参，分别为数据读取存储器的指针、数据读取起始目标地址和单块长度。SD_ReadBlock 函数执行流程如下：

(1) 将 SDIO 外设的数据控制寄存器 (SDIO_DCTRL) 清零，复位之前的传输设置。

(2) 对 SD 卡进行数据读写之前，都必须发送 CMD16 指定块的大小，对于标准卡，读取 BlockSize 长度字节的块；对于 SDHC 卡，固定读取 512 字节的块。

(3) 利用 SDIO_DataInitTypeDef 结构体类型变量配置数据传输的超时、块数量、数据块大小、数据传输方向等参数并使用 SDIO_DataConfig 函数完成数据传输环境配置。

(4) 向 SD 卡发送单块读数据命令 CMD17，SD 卡在接收到命令后就会通过数据线把数据传输到 SDIO 的数据 FIFO 内。

(5) 调用 SDIO_ITConfig 函数使能相关中断数据结束中断，当数据传输完成时会进入 SDIO 的中断服务函数。

(6) 最后调用 SD_DMA_RxConfig 函数，配置使能 SDIO 从 SD 卡读取数据的 DMA 请求，该函数可以参考代码清单 35-5。为使 SDIO 发送 DMA 请求，需要调用 SDIO_DMACmd 函数使能，配置完成后，SD 卡发出的数据将会传输到 STM32 的 SDIO 外设，而 SDIO 外设激发 DMA 请求，把数据搬运到内存中。

对于高容量的 SD 卡要求块大小必须为 512 字节，程序员有责任保证目标读取地址与块大小的字节对齐问题。

34.9.2.4.5 读取操作等待函数

SD_WaitReadOperation 函数用于等待数据读取操作完成，只有在确保数据读取完成了我们才可以放心使用数据。SD_WaitReadOperation 函数适用于单块及多亏读取函数。

列表 13: 代码清单 35-13 SD_WaitReadOperation 函数

```
/**
* @brief  本函数会一直等待到 DMA 传输结束
*         SDIO_ReadMultiBlocks() 函数后
```

(下页继续)

(续上页)

```
    必须被调用以确保 DMA 数据传输完成
* @param  None.
* @retval MSD_ERROR: 返回的 sd 错误代码.
*/
MSD_ERROR SD_WaitReadOperation(void)
{
    MSD_ERROR errorstatus = MSD_OK;
    //等待 dma 传输结束
    while ((SD_DMAEndOfTransferStatus() == RESET) &&
           (TransferEnd == 0) && (TransferError == MSD_OK)) {
    }

    if (TransferError != MSD_OK) {
        return (TransferError);
    }
    return (errorstatus);
}
```

本代码与写入等待函数类似，利用 SD_DMAEndOfTransferStatus 函数及 TransferEnd 和 TransferError 全局变量确认是否传输完成，并检查传输是否正常结束，若不正常则直接返回错误代码。SD_WaitReadOperation 函数最后是清除相关标志位并返回错误。由于这个函数里的 while 循环的存在，它会确保 DMA 的传输结束。

34.9.2.5 SDIO 中断服务函数

在进行数据传输操作时都会使能相关标志中断，用于跟踪传输进程和错误检测。中断服务函数的 stm32f1xx_it.c 文件，在移植 SDIO 驱动时要注意添加。

列表 14: 代码清单 35-14 SDIO 中断服务函数

```
/*
* 函数名: SDIO_IRQHandler
* 描述  : 在 SDIO_ITConfig() 这个函数开启了 sdio 中断 ，
    数据传输结束时产生中断
* 输入  : 无
* 输出  : 无
*/
void SDIO_IRQHandler(void)
{
    /* SDIO 中断相关的处理 * /
    SD_ProcessIRQSrc();
}

/*
* 函数名: SD_ProcessIRQSrc
* 描述  : 数据传输结束中断
* 输入  : 无
* 输出  : SD 错误类型
*/
MSD_ERROR SD_ProcessIRQSrc(void)
{
    if (StopCondition == 1) { //发送读取、多块读写命令时置 1
        SDIO->ARG = 0x0;    //命令参数寄存器
        SDIO->CMD = 0x44C;     // 命令寄存器:  0100   01     001100
        //              [7:6]    [5:0]
        //         CPSMEN   WAITRESP  CMDINDEX
        //     开启命令状态机 短响应    cmd12 STOP_TRANSMISSION
        TransferError = CmdResp1Error(SD_CMD_STOP_TRANSMISSION);
    } else {
        TransferError = MSD_OK;
```

(下页继续)

(续上页)

```
    }
    SDIO_ClearITPendingBit(SDIO_IT_DATAEND); //清中断
    SDIO_ITConfig(SDIO_IT_DATAEND, DISABLE); //关闭 sdio 中断使能
    TransferEnd = 1;
    return (TransferError);
}
```

SDIO 中断服务函数 SDIO_IRQHandler 会直接调用 SD_ProcessIRQSrc 函数执行。SD_ProcessIRQSrc 函数首先判断全局变量 StopCondition 变量是否为 1，该全局变量在 SDIO 的多块读写函数中被置 1（前面分析的单块读写函数中 StopCondition 均为 0），因为根据 SD 卡的要求，多块读写命令由 CMD12 结束，SD 卡在接收到该命令时才停止多块的传输，此处正是根据 StopCondition 的情况控制是否发送 CMD12 命令，它发送命令时直接采用往寄存器写入命令和参数的方式。

中断服务函数的其它部分，根据传输情况设置全局变量 TransferError 和 TransferEnd。

至此，我们已经介绍了 SD 卡初始化、SD 卡数据操作的基础功能函数以及 SDIO 相关中断服务函数内容，利用这个 SDIO 驱动，可以编写一些简单的 SD 卡读写测试程序。

34.9.2.6 测试函数

测试 SD 卡部分的函数是我们自己编写的，存放在 sdio_test.c 文件中。

34.9.2.6.1 SD 卡测试函数

列表 15: 代码清单 35-15 SD_Test

```
void SD_Test(void)
{
    LED_BLUE;
    /*---------------------------- SD Init ---------------------------- */
    /* SD 卡使用 SDIO 中断及 DMA 中断接收数据，中断服务程序位于 bsp_sdio_sd.c 文件
尾
```

(下页继续)

(续上页)

```
    if ((Status = SD_Init()) != MSD_OK) {
        LED_RED;
        printf("SD 卡初始化失败，请确保 SD 卡已正确接入开发板，或换一张 SD 卡测试！\
↪n");
    } else {
        printf("SD 卡初始化成功！\n");
    }
    if (Status == MSD_OK) {
        LED_BLUE;
        /* 擦除测试 */
        SD_EraseTest();

        LED_BLUE;
        /*single block 读写测试 */
        SD_SingleBlockTest();

        LED_BLUE;
    }
}
```

测试程序以开发板上 LED 灯指示测试结果，同时打印相关测试结果到串口调试助手。测试程序先调用 SD_Init 函数完成 SD 卡初始化，该函数具体代码参考代码清单 35-6，如果初始化成功就可以进行数据操作测试。

34.9.2.6.2 SD 卡擦除测试

列表 16: 代码清单 35-16 SD_EraseTest

```
void SD_EraseTest(void)
{
```

(下页继续)

(续上页)

```
    /*------------------- 块擦除 ------------------------------*/
    if (Status == MSD_OK) {
        /* 擦除 NumberOfBlocks 个块每个块长度为 512 字节 */
        Status = SD_Erase(0x00, (BLOCK_SIZE * NUMBER_OF_BLOCKS));
    }

    if (Status == MSD_OK) {
        Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00,
                                    BLOCK_SIZE, NUMBER_OF_BLOCKS);

        /* 等待传输完成 */
        Status = SD_WaitReadOperation();

        /* 检查传输是否正常 */
        while (SD_GetStatus() != SD_TRANSFER_OK);
    }

    /* 校验数据 */
    if (Status == MSD_OK) {
        EraseStatus = eBuffercmp(Buffer_MultiBlock_Rx, MULTI_BUFFER_SIZE);
    }

    if (EraseStatus == PASSED) {
        LED_GREEN;
        printf("SD 卡擦除测试成功! \n");
    } else {
        LED_BLUE;
        printf("SD 卡擦除测试失败! \n");
        printf("温馨提示: 部分 SD 卡不支持擦除测试, 若 SD 卡能通过下面的 single 读写
测试, 即表示 SD 卡能够正常使用。\n");
    }
}
```

SD_EraseTest 函数主要编程思路是擦除一定数量的数据块，接着读取已擦除块的数据，把读取到的数据与 0xff 或者 0x00 比较，得出擦除结果。

SD_Erase 函数用于擦除指定地址空间，源代码参考*代码清单 35_9*，它接收两个参数指定擦除空间的起始地址和终止地址。如果 SD_Erase 函数返回正确，表示擦除成功则执行数据块读取；如果 SD_Erase 函数返回错误，表示 SD 卡擦除失败，并不是所有卡都能擦除成功的，部分卡虽然擦除失败，但数据读写操作也是可以正常执行的。这里使用多块读取函数 SD_ReadMultiBlocks，它有四个形参，分别为读取数据存储器、读取数据目标地址、块大小以及块数量，函数后面都会跟随等待数据传输完成相关处理代码。接下来会调用 eBuffercmp 函数判断擦除结果，它有两个形参，分别为数据指针和数据字节长度，它实际上是把数据存储器内所有数据都与 0xff 或 0x00 做比较，只有出现这两个数之外就报错退出。

34.9.2.6.3 单块读写测试

列表 17: 代码清单 35-17 SD_SingleBlockTest 函数

```
void SD_SingleBlockTest(void)
{
    /*------------------- 块 读写 ---------------------------*/
    /* 向数组填充要写入的数据 */
    Fill_Buffer(Buffer_Block_Tx, BLOCK_SIZE, 0x320F);

    if (Status == MSD_OK) {
        /* 把 512 个字节写入到 SD 卡的 0 地址 */
        Status = SD_WriteBlock(Buffer_Block_Tx, 0x00, BLOCK_SIZE);
        /* 检查传输 */
        Status = SD_WaitWriteOperation();
        while (SD_GetStatus() != SD_TRANSFER_OK);
    }
    if (Status == MSD_OK) {
        /* 从 SD 卡的 0 地址读取 512 个字节 */
        Status = SD_ReadBlock(Buffer_Block_Rx, 0x00, BLOCK_SIZE);
```

(下页继续)

(续上页)

```
        /* 检查传输 */
        Status = SD_WaitReadOperation();
        while (SD_GetStatus() != SD_TRANSFER_OK);
    }
    /* 校验读出的数据是否与写入的数据一致 */
    if (Status == MSD_OK) {
        TransferStatus1 = Buffercmp(Buffer_Block_Tx,
                                    Buffer_Block_Rx, BLOCK_SIZE);
    }
    if (TransferStatus1 == PASSED) {
        LED_GREEN;
        printf("Single block 测试成功! \n");
    } else {
    LED_RED;
printf("Single block 测试失败，请确保 SD 卡正确接入开发板，或换一张 SD 卡测试! \n");
    }
}
```

SD_SingleBlockTest 函数主要编程思想是首先填充一个块大小的存储器，通过写入操作把数据写入到 SD 卡内，然后通过读取操作读取数据到另外的存储器，然后在对比存储器内容得出读写操作是否正确。

SD_SingleBlockTest 函数一开始调用 Fill_Buffer 函数用于填充存储器内容，它只是简单实用 for 循环赋值方法给存储区填充数据，它有三个形参，分别为存储区指针、填充字节数和起始数选择，这里的起始数选择参数对本测试没有实际意义。SD_WriteBlock 函数和 SD_ReadBlock 函数分别执行数据写入和读取操作，具体可以参考*代码清单 35_10* 和*代码清单 35_12*。Buffercmp 函数用于比较两个存储区内容是否完全相等，它有三个形参，分别为第一个存储区指针、第二个存储区指针和存储器长度，该函数只是循环比较两个存储区对应位置的两个数据是否相等，只有发现存在不相等就报错退出。

SD_MultiBlockTest 函数与 SD_SingleBlockTest 函数执行过程类似，这里就不做详细分析。

34.9.2.6.4 主函数

列表 18: 代码清单 35-18 main 函数

```
int main(void)
{
    HAL_Init();
    /* 配置系统时钟为 72 MHz */
    SystemClock_Config();
    /* 初始化 USART 配置模式为 115200 8-N-1, 中断接收 */
    DEBUG_USART_Config();

    /* 初始化 LED 灯 */
    LED_GPIO_Config();
    LED_BLUE;
    /* 初始化独立按键 */
    Key_GPIO_Config();

    /* 初始化 USART1*/
    DEBUG_USART_Config();

    printf("\r\n欢迎使用野火  STM32 开发板。\r\n");

    printf("在开始进行 SD 卡基本测试前，请给开发板插入 32G 以内的 SD 卡\r\n");
    printf("本程序会对 SD 卡进行 非文件系统方式读写，会删除 SD 卡的文件系统\r\n");
    printf("实验后可通过电脑格式化或使用 SD 卡文件系统的例程恢复 SD 卡文件系统\r\n
↪");
    printf("\r\n 但 sd 卡内的原文件不可恢复，实验前务必备份 SD 卡内的原文件!!! \r\n
↪");
    printf("\r\n 若已确认，请按开发板的 KEY1 按键，开始 SD 卡测试实验....\r\n");

    /* Infinite loop */
    while (1) {
```

(下页继续)

(续上页)

```
        /* 按下按键开始进行 SD 卡读写实验，会损坏 SD 卡原文件 */
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON) {
            printf("\r\n开始进行 SD 卡读写实验\r\n");
            SD_Test();
        }
    }
}
```

测试过程中有用到 LED 灯、独立按键和调试串口，所以需要对这些模块进行初始化配置。在无限循环中不断检测按键状态，如果有被按下就执行 SD 卡测试函数。由于本实验尚未移植文件系统，所以运行后会破坏原 SD 卡存储的内容，所以实验前请注意备份 SD 卡的数据。

34.9.3 下载验证

把 Micro SD 卡插入到开发板右侧的卡槽内，使用 USB 线连接开发板上的“USB TO UART”接口到电脑，电脑端配置好串口调试助手参数。编译实验程序并下载到开发板上，程序运行后在串口调试助手可接收到开发板发过来的提示信息，按下开发板左下边沿的 K1 按键，开始执行 SD 卡测试，测试结果在串口调试助手可观察到，板子上 LED 灯也可以指示测试结果。

第 35 章　基于 SD 卡的 FatFs 文件系统

上一章我们已经全面介绍了 SD 卡的识别和简单的数据读写，也进行了简单的读写测试，不过像这样直接操作 SD 卡存储单元，在实际应用中是不现实的。SD 卡一般用来存放文件，所以都需要加载文件系统到里面。类似于串行 Flash 芯片，我们移植 FatFs 文件系统到 SD 卡内。

对于 FatFs 文件系统的介绍和具体移植过程参考“基于串行 Flash 的 FatFs 文件系统”，这里就不做过多介绍，重点放在 SD 卡与 FatFs 接口函数编写上。与串行 Flash 的 FatFs 文件系统移植例程相比，FatFs 文件系统部分的代码只有 diskio.c 文件有所不同，其他的不用修改，所以一个简易的移植方法是利用原来工程进行修改。下面讲解利用原来工程实现 SD 卡的 FatFs 文件系统。

35.1 FatFs 移植步骤

上一章我们已经完成了 SD 卡驱动程序以及进行了简单的读写测试。该工程有很多东西是现在可以使用的，所以我们先把上一章的工程文件完整的拷贝一份，并修改文件夹名为“SDIO-FatFs 移植与读写测试”，如果此时使用 KEIL 软件打开该工程，应该是编译无错误并实现上一章的测试功能。

接下来，我们到串行 Flash 文件系统移植工程文件的“SPI—FatFs 移植与读写测试 User”文件夹下拷贝“FATFS”整个文件夹到现在工程文件的“SDIO—FatFs 移植与读写测试 User”文件夹下，如图 *36_1*。该文件夹是 FatFs 文件系统的所有代码文件，在串行 Flash 移植 FatFs 文件系统时我们对部分文件做了修改，这里主要是想要保留之前的配置，而不是使用 FatFs 官方源码还需要重新配置。

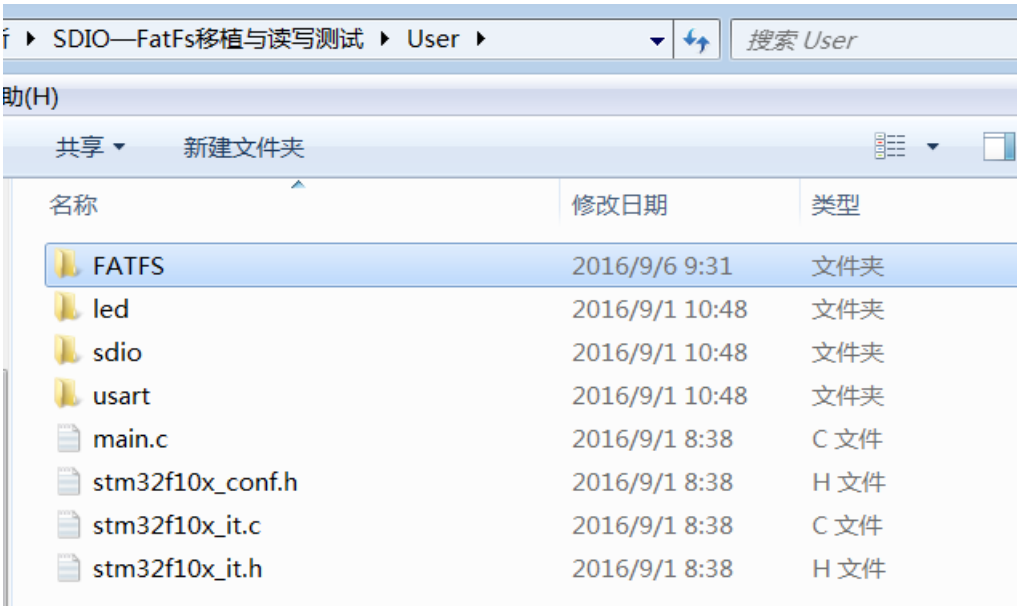
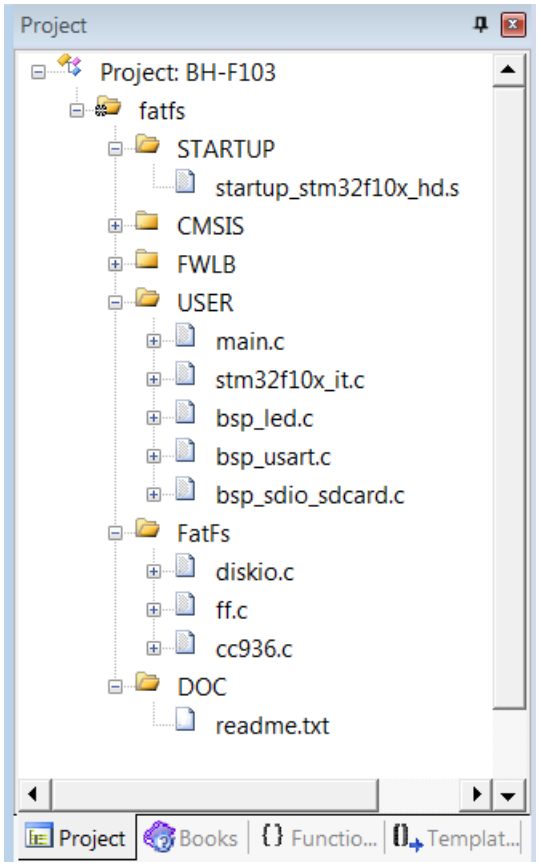
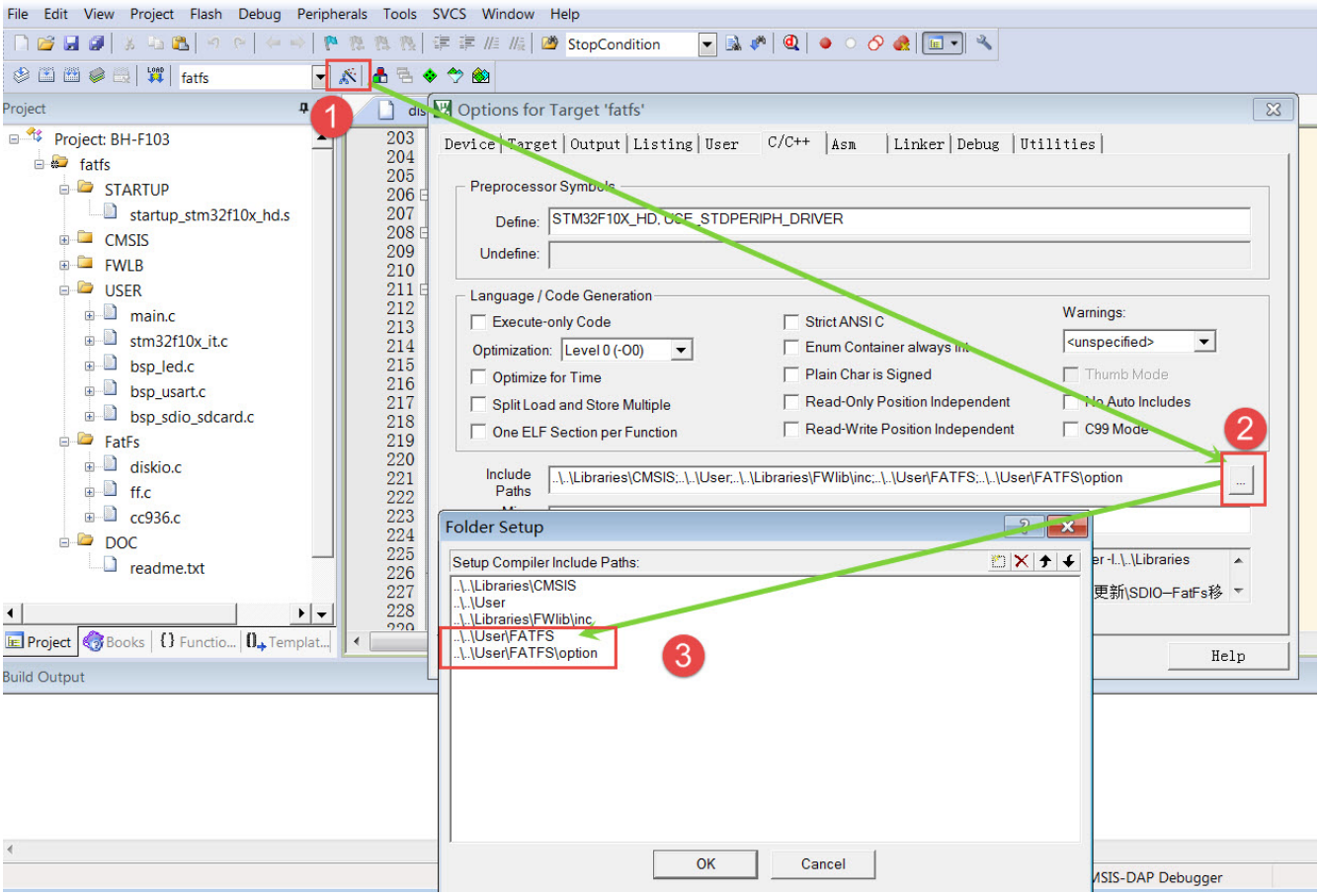


图 36-1 拷贝 FatFs 文件夹

现在就可以使用 KEIL 软件打开“SDIO-FatFs 移植与读写测试”工程文件，并把 FatFs 相关文件添加到工程内，同时把 sdio_test.c 文件移除，参考图 *36_2*。



添加文件之后还必须打开工程选项对话框添加文件系统的头文件路径，参考图 *36_3*。



操作到这来，工程文件结构就算完整了，接下来就是修改文件代码了。这来有两个文件需要修改，为 diskio.c 文件和 main.c 文件。main.c 文件内容可以参考“SPI—FatFs 移植与读写测试”工程中的 main.c 文件，只有做小细节修改而已。这来重点讲解 diskio.c 文件，也是整个移植的重点。

35.2 FatFs 接口函数

FatFs 文件系统与存储设备的连接函数在 diskio.c 文件中，主要有 5 个函数需要我们编写的。

35.2.1 宏定义和存储设备状态获取函数

列表 1: 代码清单 36-1 宏定义和 SD_status 函数

```
DSTATUS SD_status(BYTE lun)
{
    Stat = STA_NOINIT;

    if (BSP_SD_GetStatus() == MSD_OK) {
        Stat &= ~STA_NOINIT;
    }

    return Stat;
}
```

SD_status 函数要求返回存储设备的当前状态，对于 SD 卡一般返回 SD 卡插入状态，这里直接返回正常状态。

35.2.2 存储设备初始化函数

列表 2: 代码清单 36-2 disk_initialize 函数

```
DSTATUS SD_initialize(BYTE lun)
{
    Stat = STA_NOINIT;

    /* Configure the uSD device */
    if (BSP_SD_Init() == MSD_OK) {
        Stat &= ~STA_NOINIT;
    }

    return Stat;
```

(下页继续)

(续上页)

```
}
```

该函数用于初始化存储设备，一般包括相关 GPIO 初始化、外设环境初始化、中断配置等等。对于 SD 卡，直接调用 BSP_SD_Init 函数实现对 SD 卡初始化，如果函数返回 SD_OK 说明 SD 卡正确插入，并且控制器可以与之正常通信。

35.2.3 存储设备数据读取函数

列表 3: 代码清单 36-3 SD_read 函数

```
DRESULT SD_read(BYTE lun, BYTE *buff, DWORD sector, UINT count)
{
    DRESULT res = RES_OK;

    if ((DWORD)buff & 3) {
        DWORD scratch[BLOCK_SIZE / 4];

        while (count--) {
            memcpy(scratch, buff, BLOCK_SIZE);
            res = SD_read(lun,(void *)scratch, sector++, 1);

            if (res != RES_OK) {
                break;
            }
            buff += BLOCK_SIZE;
        }

        return (res);
    }
    if (BSP_SD_ReadBlocks_DMA((uint32_t*)buff,
                        (uint64_t) (sector * BLOCK_SIZE),
```

(下页继续)

(续上页)

```
                        BLOCK_SIZE,
                        count) != MSD_OK) {
        res = RES_ERROR;
    }

    return res;
}
```

SD_read 函数用于从存储设备指定地址开始读取一定的数量的数据到指定存储区内。对于 SD 卡，最重要是使用 BSP_SD_ReadBlocks_DMA 函数读取多块数据到存储区。这里需要注意的地方是 SD 卡数据操作是使用 DMA 传输的，并设置数据尺寸为 32 位大小，为实现数据正确传输，要求存储区是 4 字节对齐。在某些情况下，FatFs 提供的 buff 地址不是 4 字节对齐，这会导致 DMA 数据传输失败，所以为保证数据传输正确，可以先判断存储区地址是否是 4 字节对齐，如果存储区地址已经是 4 字节对齐，无需其他处理，直接使用 BSP_SD_ReadBlocks_DMA 函数执行多块读取即可。如果判断得到地址不是 4 字节对齐，则先申请一个 4 字节对齐的临时缓冲区，即局部数组变量 scratch，通过定义为 DWORD 类型可以使得其自动 4 字节对齐，scratch 所占的总存储空间也是一个块大小，这样把一个块数据读取到 scratch 内，然后把 scratch 存储器内容拷贝到 buff 地址空间上就可以了。

BSP_SD_ReadBlocks_DMA 函数用于从 SD 卡内读取多个块数据，它有四个形参，分别为存储区地址指针、起始块地址、块大小以及块数量。

35.2.4 存储设备数据写入函数

列表 4: 代码清单 36-4 disk_write 函数

```
DRESULT SD_write(BYTE lun, const BYTE *buff, DWORD sector, UINT count)
{
    DRESULT res = RES_OK;

    if ((DWORD)buff & 3) {
```

(下页继续)

(续上页)

```
        DWORD scratch[BLOCK_SIZE / 4];

        while (count--) {
            memcpy(scratch, buff, BLOCK_SIZE);
            res = SD_write(lun,(void *)scratch, sector++, 1);

            if (res != RES_OK) {
                break;
            }
            buff += BLOCK_SIZE;
        }

        return (res);
    }
    if (BSP_SD_WriteBlocks_DMA((uint32_t*)buff,
                        (uint64_t)(sector * BLOCK_SIZE),
                        BLOCK_SIZE, count) != MSD_OK) {
        res = RES_ERROR;
    }

    return res;
}
```

SD_write 函数用于向存储设备指定地址写入指定数量的数据。对于 SD 卡，执行过程与 SD_read 函数是非常相似，也必须先检测存储区地址是否是 4 字节对齐，如果是 4 字节对齐则直接调用 BSP_SD_WriteBlocks_DMA 函数完成多块数据写入操作。如果不是 4 字节对齐，申请一个 4 字节对齐的临时缓冲区，先把待写入的数据拷贝到该临时缓冲区内，然后才写入到 SD 卡。

BSP_SD_WriteBlocks_DMA 函数是向 SD 卡写入多个块数据，它有四个形参，分别为存储区地址指针、起始块地址、块大小以及块数量，它与 BSP_SD_ReadBlocks_DMA 函数执行相互过程。最后也是需要使用相关函数保存数据写入完整才退出 SD_write 函数。

35.2.5 其他控制函数

列表 5: 代码清单 36-5 disk_ioctl 函数

```
DRESULT SD_ioctl(BYTE lun, BYTE cmd, void *buff)
{
    DRESULT res = RES_ERROR;
    SD_CardInfo CardInfo;

    if (Stat & STA_NOINIT) return RES_NOTRDY;
    switch (cmd) {
    /* Make sure that no pending write process */
    case CTRL_SYNC :
        res = RES_OK;
        break;

    /* Get number of sectors on the disk (DWORD) */
    case GET_SECTOR_COUNT :
        BSP_SD_GetCardInfo(&CardInfo);
        *(DWORD*)buff = CardInfo.CardCapacity / BLOCK_SIZE;
        res = RES_OK;
        break;

    /* Get R/W sector size (WORD) */
    case GET_SECTOR_SIZE :
        *(WORD*)buff = BLOCK_SIZE;
        res = RES_OK;
        break;

    /* Get erase block size in unit of sector (DWORD) */
    case GET_BLOCK_SIZE :
        *(DWORD*)buff = BLOCK_SIZE;
        break;
```

(下页继续)

(续上页)

```
    default:
        res = RES_PARERR;
    }

    return res;
}
```

SD_ioctl 函数有三个形参，lun 为设备物理编号，cmd 为控制指令，包括发出同步信号、获取扇区数目、获取扇区大小、获取擦除块数量等等指令，buff 为指令对应的数据指针。

对于 SD 卡，为支持格式化功能，需要用到获取扇区数量 (GET_SECTOR_COUNT) 指令和获取块尺寸 (GET_BLOCK_SIZE)。另外，SD 卡扇区大小为 512 字节，串行 Flash 芯片一般设置扇区大小为 4096 字节，所以需要用到获取扇区大小 (GET_SECTOR_SIZE) 指令。

至此，基于 SD 卡的 FatFs 文件系统移植就已经完成了，最重要就是 sd_diskio.c 文件中 5 个函数的编写。接下来就编写 FatFs 基本的文件操作检测移植代码是否可以正确执行。

35.3 FatFs 功能测试

要的测试包括格式化测试、文件写入测试和文件读取测试三个部分，主要程序都在 main.c 文件中实现。

35.3.1 变量定义

列表 6: 代码清单 36-6 变量定义

```
char SDPath[4];             /* SD 逻辑驱动器路径 */
FATFS fs;                       /* FatFs 文件系统对象 */
FIL fnew;                       /* 文件对象 */
FRESULT res_sd;                /* 文件操作结果 */
```

(下页继续)

(续上页)

```
UINT fnum;                          /* 文件成功读写数量 */
BYTE ReadBuffer[1024]= {0};         /* 读缓冲区 */
BYTE WriteBuffer[] =                /* 写缓冲区 */
    "欢迎使用野火 STM32 F103 开发板 今天是个好日子，新建文件系统测试文件\r\n";
```

SDPath[4] 为存储 SD 逻辑驱动器路径的一个数组，存储的内容是“0:/”。

FATFS 是在 ff.h 文件定义的一个结构体类型，针对的对象是物理设备，包含了物理设备的物理编号、扇区大小等等信息，一般我们都需要为每个物理设备定义一个 FATFS 变量。

FIL 也是在 ff.h 文件定义的一个结构体类型，针对的对象是文件系统内具体的文件，包含了文件很多基本属性，比如文件大小、路径、当前读写地址等等。如果需要在同一时间打开多个文件进行读写，才需要定义多个 FIL 变量，不然一般定义一个 FIL 变量即可。

FRESULT 是也在 ff.h 文件定义的一个枚举类型，作为 FatFs 函数的返回值类型，主要管理 FatFs 运行中出现的错误。总共有 19 种错误类型，包括物理设备读写错误、找不到文件、没有挂载工作空间等等错误。这在实际编程中非常重要，当有错误出现是我们要停止文件读写，通过返回值我们可以快速定位到错误发生的可能地点。如果运行没有错误才返回 FR_OK。

fnum 是个 32 位无符号整形变量，用来记录实际读取或者写入数据的数组。

buffer 和 textFileBuffer 分别对应读取和写入数据缓存区，都是 8 位无符号整形数组。

35.3.2 主函数

列表 7: 代码清单 36-7 main 函数

```
int main(void)
{
    /* 配置系统时钟为 72 MHz */
    SystemClock_Config();
    /* 初始化 USART 配置模式为 115200 8-N-1，中断接收 */
    DEBUG_USART_Config();
```

(下页继续)

(续上页)

```
/* 初始化 LED 灯 */
LED_GPIO_Config();
LED_BLUE;

/* 初始化 USART1*/
DEBUG_USART_Config();

/* 初始化调试串口，一般为串口 1 */
DEBUG_USART_Config();
printf("\r\n****** 这是一个 SD 卡 文件系统实验 ******\r\n");

//链接驱动器，创建盘符
FATFS_LinkDriver(&SD_Driver, SDPath);
//在外部 SD 卡挂载文件系统，文件系统挂载时会对 SD 卡初始化
f_res = f_mount(&fs,"0:",1);

/*----------------------- 格式化测试 ---------------------------*/
/* 如果没有文件系统就格式化创建创建文件系统 */
if(f_res == FR_NO_FILESYSTEM)
{
printf("》SD 卡还没有文件系统，即将进行格式化...\r\n");
/* 格式化 */
f_res=f_mkfs("0:",0,0);
if(f_res == FR_OK)
{
    printf("》SD 卡已成功格式化文件系统。\r\n");
    /* 格式化后，先取消挂载 */
    f_res = f_mount(NULL,"0:",1);
    /* 重新挂载 */
    f_res = f_mount(&fs,"0:",1);
}
```

(下页继续)

(续上页)

```
    else
    {
        LED_RED;
        printf("《《格式化失败。》》\r\n");
        while(1);
    }
    }
    else if(f_res!=FR_OK)
    {
        printf("!! SD 卡挂载文件系统失败。(%d)\r\n",f_res);
        printf("!! 可能原因: SD 卡初始化不成功。\r\n");
        while(1);
    }
    else
    {
        printf("》文件系统挂载成功，可以进行读写测试\r\n");
    }
    /*----------------------- 文件系统测试: 写测试 -----------------------------
↪--*/
    /* 打开文件，如果文件不存在则创建它 */
    printf("\r\n****** 即将进行文件写入测试... ******\r\n");
    f_res = f_open(&file, "0:FatFs 读写测试文件.txt",FA_CREATE_ALWAYS | FA_
↪WRITE );
    if ( f_res == FR_OK )
    {
    printf("》打开/创建 FatFs 读写测试文件.txt 文件成功，向文件写入数据。\r\n");
    /* 将指定存储区内容写入到文件内 */
    f_res=f_write(&file,WriteBuffer,sizeof(WriteBuffer),&fnum);
    if(f_res==FR_OK)
    {
        printf("》文件写入成功，写入字节数据: %d\n",fnum);
        printf("》向文件写入的数据为: \r\n%s\r\n",WriteBuffer);
```

(下页继续)

(续上页)

```
    }
    else
    {
        printf("!! 文件写入失败: (%d)\n",f_res);
    }
    /* 不再读写，关闭文件 */
    f_close(&file);
    }
    else
    {
    LED_RED;
    printf("!! 打开/创建文件失败。\r\n");
    }
    /*------------------- 文件系统测试: 读测试 ------------------------------
↪-----*/
    printf("****** 即将进行文件读取测试... ******\r\n");
    f_res = f_open(&file, "0:FatFs 读写测试文件.txt", FA_OPEN_EXISTING | FA_
↪READ);
    if(f_res == FR_OK)
    {
    LED_GREEN;
    printf("》打开文件成功。\r\n");
    f_res = f_read(&file, ReadBuffer, sizeof(ReadBuffer), &fnum);
    if(f_res==FR_OK)
    {
        printf("》文件读取成功，读到字节数据: %d\r\n", fnum);
        printf("》读取得的文件数据为: \r\n%s \r\n",  ReadBuffer);
    }
    else
    {
        printf("!! 文件读取失败: (%d)\n",f_res);
    }
```

(下页继续)

(续上页)

```
    }
    else
    {
    LED_RED;
    printf("!! 打开文件失败。\r\n");
    }
    /* 不再读写，关闭文件 */
    f_close(&file);
    /* 不再使用文件系统，取消挂载文件系统 */
    f_mount(NULL,"0:",1);
    /* 操作完成，停机 */
    while(1)
    {
    }
}
```

程序的开头首先初始化 RGB 彩灯和调试串口，用来指示程序进程。

FatFs 的第一步工作就是使用 f_mount 函数挂载工作区。f_mount 函数有三个形参，第一个参数是指向 FATFS 变量指针，如果赋值为 NULL 可以取消物理设备挂载。第二个参数为逻辑设备编号，使用设备根路径表示，与物理设备编号挂钩，在代码清单 38-1 中我们定义 SD 卡物理编号为 0，所以这里使用“0:”。第三个参数可选 0 或 1，1 表示立即挂载，0 表示不立即挂载，延迟挂载。f_mount 函数会返回一个 FRESULT 类型值，指示运行情况。

如果 f_mount 函数返回值为 FR_NO_FILESYSTEM，说明 SD 卡没有 FAT 文件系统。我们就必须对 SD 卡进行格式化处理。使用 f_mkfs 函数可以实现格式化操作。f_mkfs 函数有三个形参，第一个参数为逻辑设备编号；第二参数可选 0 或者 1，0 表示设备为一般硬盘，1 表示设备为软盘。第三个参数指定扇区大小，如果为 0，表示通过代码清单 38-5 中 disk_ioctl 函数获取。格式化成功后需要先取消挂载原来设备，再重新挂载设备。

在设备正常挂载后，就可以进行文件读写操作了。使用文件之前，必须使用 f_open 函数打开文件，不再使用文件必须使用 f_close 函数关闭文件，这个跟电脑端操作文件步骤类似。f_open 函数有三个形参，第一个参数为文件对象指针。第二参数为目标文件，包含绝对路径的文件名称

和后缀名。第三个参数为访问文件模式选择，可以是打开已经存在的文件模式、读模式、写模式、新建模式、总是新建模式等的或运行结果。比如对于写测试，使用 FA_CREATE_ALWAYS 和 FA_WRITE 组合模式，就是总是新建文件并进行写模式。

f_close 函数用于不再对文件进行读写操作关闭文件，f_close 函数只要一个形参，为文件对象指针。f_close 函数运行可以确保缓冲区完全写入到文件内。

成功打开文件之后就可以使用 f_write 函数和 f_read 函数对文件进行写操作和读操作。这两个函数用到的参数是一致的，只不过一个是数据写入，一个是数据读取。f_write 函数第一个形参为文件对象指针，使用与 f_open 函数一致即可。第二个参数为待写入数据的首地址，对于 f_read 函数就是用来存放读出数据的首地址。第三个参数为写入数据的字节数，对于 f_read 函数就是欲读取数据的字节数。第四个参数为 32 位无符号整形指针，这里使用 fnum 变量地址赋值给它，在运行读写操作函数后，fnum 变量指示成功读取或者写入的字节个数。

最后，不再使用文件系统时，使用 f_mount 函数取消挂载。

35.3.3 下载验证

保证开发板相关硬件连接正确，用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。程序开始运行后，RGB 彩灯为蓝色，在串口调试助手可看到格式化测试、写文件检测和读文件检测三个过程；最后如果所有读写操作都正常，RGB 彩灯会指示为绿色，如果在运行中 FatFs 出现错误 RGB 彩灯指示为红色。正确执行例程程序后可以使用读卡器将 SD 卡在电脑端打开，我们可以在 SD 卡根目录下看到“FatFs 读写测试文件.txt”文件，这与程序设计是相吻合的。

第 36 章　DAC—输出正弦波

本章参考资料：《STM32F10X-中文参考手册》DAC 章节。

学习本章时，配合《STM32F10X-中文参考手册》DAC 章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

36.1 DAC 简介

DAC 为数字/模拟转换模块，故名思议，它的作用就是把输入的数字编码，转换成对应的模拟电压输出，它的功能与 ADC 相反。在常见的数字信号系统中，大部分传感器信号被化成电压信号，而 ADC 把电压模拟信号转换成易于计算机存储、处理的数字编码，由计算机处理完成后，再由 DAC 输出电压模拟信号，该电压模拟信号常常用来驱动某些执行器件，使人类易于感知。如音频信号的采集及还原就是这样一个过程。

STM32 具有片上 DAC 外设，它的分辨率可配置为 8 位或 12 位的数字输入信号，具有两个 DAC 输出通道，这两个通道互不影响，每个通道都可以使用 DMA 功能，都具有出错检测能力，可外部触发。

36.2 DAC 功能框图剖析

STM32 的 DAC 模块框图见图 *37_1*。

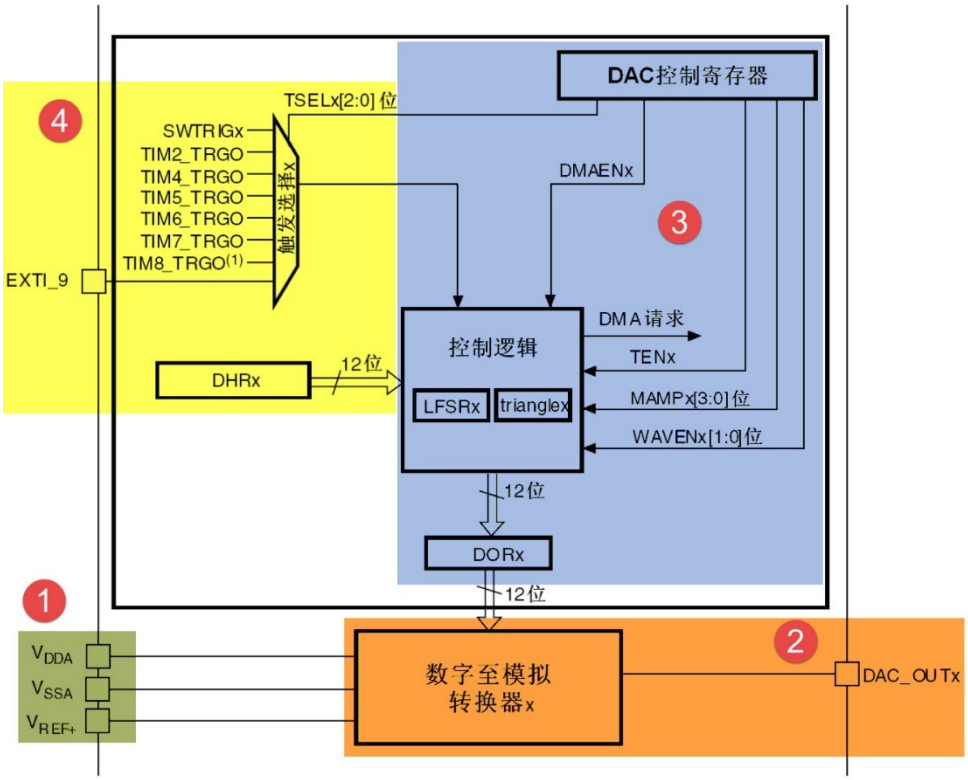


图 37-1 DAC 功能框图

整个 DAC 模块围绕框图下方的“数字至模拟转换器 x”展开，它的左边分别是参考电源的引脚：V_{DDA}、V_{SSA} 及 V_{ref+}，其中 STM32 的 DAC 规定了它的参考电压 V_{ref+} 输入范围为 2.4——3.3V。“数字至模拟转换器 x”的输入为 DAC 的数据寄存器“DORx”的数字编码，经过它转换得的模拟信号由图中右侧的“DAC_OUTx”输出。而数据寄存器“DORx”又受“控制逻辑”支配，它可以控制数据寄存器加入一些伪噪声信号或配置产生三角波信号。图中的左上角为 DAC 的触发源，DAC 根据触发源的信号来进行 DAC 转换，其作用就相当于 DAC 转换器的开关，它可以配置的触发源为外部中断源触发、定时器触发或软件控制触发。如本章实验中需要控制正弦波的频率，就需要定时器定时触发 DAC 进行数据转换。

36.2.1 参考电压

与 ADC 外设类似，DAC 也使用 V_{REF+} 引脚作为参考电压，在设计原理图的时候一般把 V_{SSA} 接地，把 V_{REF+} 和 V_{DDA} 接 3.3V，可得到 DAC 的输出电压范围为：0~3.3V。

如果想让输出的电压范围变宽，可以在外部加一个电压调理电路，把 0~3.3V 的 DAC 输出抬升到特定的范围即可。

36.2.2 数模转换及输出通道

框图中的“数字至模拟转换器 x”是核心部件，整个 DAC 外设都围绕它而展开。它以左边的 V_{REF+} 作为参考电源，以 DAC 的数据寄存器“DORx”的数字编码作为输入，经过它转换得的模拟信号由右侧的“DAC_OUTx”通道输出。其中各个部件中的“x”是指设备的标号，在 STM32 中具有 2 个这样的 DAC 部件，每个 DAC 有 1 个对应的输出通道连接到特定的引脚，即：PA4-通道 1，PA5-通道 2，为避免干扰，使用 DAC 功能时，DAC 通道引脚需要被配置成模拟输入功能（AIN）。

36.2.3 触发源及 DHRx 寄存器

在使用 DAC 时，不能直接对上述 DORx 寄存器写入数据，任何输出到 DAC 通道 x 的数据都必须写入到 DHRx 寄存器中（其中包含 DHR8Rx、DHR12Lx 等，根据数据对齐方向和分辨率的情况写入到对应的寄存器中）。

数据被写入到 DHRx 寄存器后，DAC 会根据触发配置进行处理，若使用硬件触发，则 DHRx 中的数据会在 3 个 APB1 时钟周期后传输至 DORx，DORx 随之输出相应的模拟电压到输出通道；若 DAC 设置为外部事件触发，可以使用定时器（TIMx_TRGO）、EXTI_9 信号或软件触发（SWTRIGx）这几种方式控制数据 DAC 转换的时机，例如使用定时器触发，配合不同时刻的 DHRx 数据，可实现 DAC 输出正弦波的功能。

36.3 DAC 初始化结构体详解

在 ST 的 HAL 库中，把控制 DAC 相关的各种配置封装到了结构体 DAC_InitTypeDef 中，它主要包含了 DAC_CR 控制寄存器的各寄存器位的配置，见*代码清单 37_1*。

列表 1: 代码清单 37-1 DAC_InitTypeDef 结构体

```
typedef struct {
   uint32_t DAC_Trigger;
   /*DAC 触发方式 */
   uint32_t DAC_OutputBuffer;
   /* 选择是否使能输出缓冲器 */
} DAC_ChannelConfTypeDef;
```

各个结构体成员的介绍如下，解说中各模式后括号内的英文为该模式在 HAL 库中使用宏：

(1) DAC_Trigger

本成员用于配置 DAC 的触发模式，当 DAC 产生相应的触发事件时，才会把 DHRx 寄存器的值转移到 DORx 寄存器中进行转换。本结构体成员可以选择的触发模式如下：定时器触发模式（AC_TRIGGER_T2/4//6/7_TRGO），使用定时器 2、4、6、7 控制 DHRx 寄存器的数据按时间转移到 DORx 中进行转换，利用这种方式可以输出特定的波形；EXTI_9 触发方式（DAC_TRIGGER_EXT_IT9），当产生 EXTI_9 事件时（如 GPIO 中断事件），触发转换；软件触发模式（DAC_TRIGGER_SOFTWARE），在本模式下，向 DAC_SWTRIGR 寄存器写入配置即可触发信号进行转换。

(2) DAC_OutputBuffer

本结构体成员用于控制是否使能 DAC 的输出缓冲（DAC_OUTPUTBUFFER_ENABLE/DISABLE），使能了 DAC 的输出缓冲后可以减小输出阻抗，适合直接驱动一些外部负载。

36.4 DAC 输出正弦波实验

利用 STM32 的 DAC 配合 TIM 定时器，可以输出随时间变化的电压，本章的实验以输出正弦波为例，演示如何控制输出电压波形。

36.4.1 硬件设计

STM32 的 DAC 外设有固定的输出通道，分别为 PA4 和 PA5，不过，指南者开发板已经在板载 SPI-FLASH 芯片中使用了这两个引脚，所以用作 DAC 通道输出电压时会受到干扰，影响实验，见图 *38_3*。

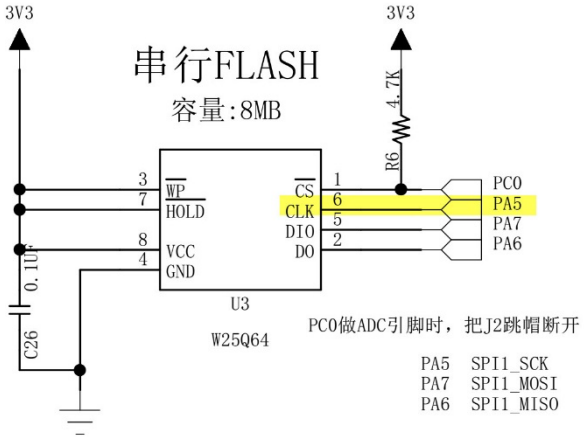


图 38-3 SPI-FLASH 占用了 DAC 使用的输出通道

PA5 引脚连接到 FLASH 芯片的 CLK 引脚中，这可能会干扰 DAC 实验输出的电压信号，导致得不到正确的波形。经测试，在进行本实验时，只要不使用 SPI-FLASH，DAC 可以输出正常的波形，所以 DAC 和 SPI-FLASH 不在同一个实验中使用即可。

在设计 DAC 专门的实际应用时，DAC 的输出通道应独占，不与其它设备共用。本开发板的设计是考虑到各种资源分配，才不得不占用 DAC 通道的。

实验时直接使用示波器测量 PA4 和 PA5 引脚的输出即可。

36.4.2 软件设计

为了使工程更加有条理，我们把 DAC 控制相关的代码独立分开存储，方便以后移植。新建“bsp_dac.c”及“bsp_dac.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32HAL 库的内容，是由我们自己根据应用需要编写的。

36.4.2.1 编程要点

1) 计算获取正弦波数据表；
2) 根据正弦波数据表的周期内点数和周期计算定时器触发间隔；
3) 初始化 DAC 输出通道，初始化 DAC 工作模式；
4) 配置触发 DAC 用的定时器；
5) 配置 DMA 自动转运正弦波数据表。

配置完成后，即可在 PA4、PA5 引脚中检测到信号输出。

36.4.2.2 代码分析

36.4.2.2.1 生成正弦波数据表

要输出正弦波，实质是要控制 DAC 以 v=sin(t) 的正弦函数关系输出电压，其中 v 为电压输出，t 为时间。

而由于模拟信号连续而数字信号是离散的，所以使用 DAC 产生正弦波时，只能按一定时间间隔输出正弦曲线上的点，在该时间段内输出相同的电压值，若缩短时间间隔，提高单个周期内的输出点数，可以得到逼近连续正弦波的图形，见图 *37_3*，若在外部电路加上适当的电容滤波，可得到更完美的图形。

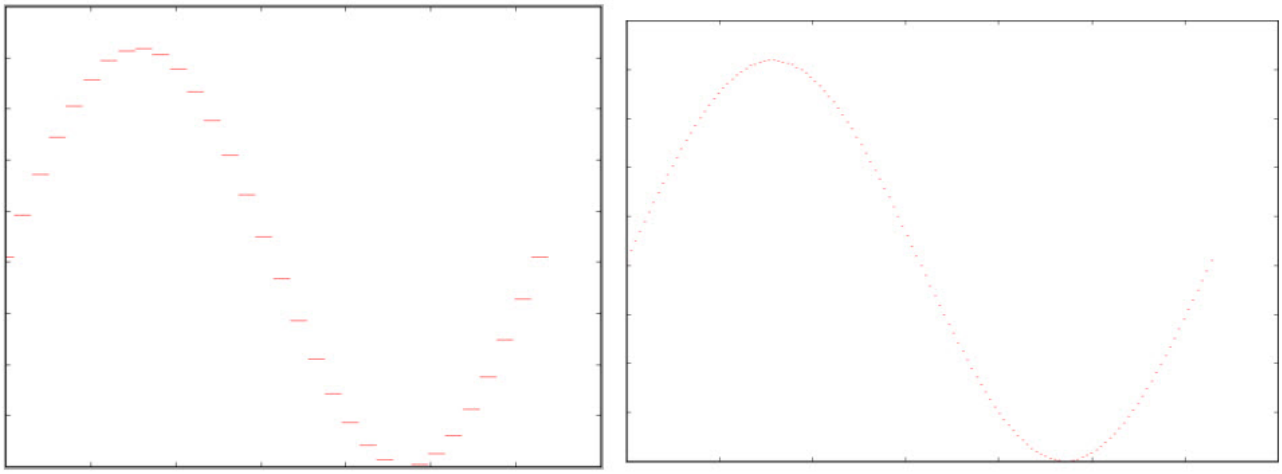


图 37-3 DAC 按点输出正弦波数据 (左：32 个点，右：128 个点)

由于正弦曲线是周期函数，所以只需要得到单个周期内的数据后按周期重复即可，而单个周期内取样输出的点数又是有限的，所以为了得到呈 v=sin(t) 函数关系电压值的数据通常不会实时计算获取，而是预先计算好函数单个周期内的电压数据表，并且转化成以 DAC 寄存器表示的值。

如 sin 函数值的范围为 [-1: +1]，而 STM32 的 DAC 输出电压范围为 [0~3.3]V，按 12 位 DAC 分辨率表示的方法，可写入寄存器的最大值为 $2^{12} = 4096$，即范围为 [0:4096]。所以，实际输出时，会进行如下处理：

1) 抬升 sin 函数的输出为正值：v = sin(t)+1 ，此时，v 的输出范围为 [0:2]；
2) 扩展输出至 DAC 的全电压范围：v = 3.3*(sin(t)+1)/2 ，此时，v 的输出范围为 [0:3.3]，正是 DAC 的电压输出范围，扩展至全电压范围可以充分利用 DAC 的分辨率；
3) 把电压值以 DAC 寄存器的形式表示：Reg_val = 2^{12}/3.3 * v = 2^{11}*(sin(t)+1)，此时，存储到 DAC 寄存器的值范围为 [0:4096]；
4) 实践证明，在 sin(t) 的单个周期内，取 32 个点进行电压输出已经能较好地还原正弦波形，所以在 t∈[0:2π] 区间内等间距根据上述 Reg_val 公式运算得到 32 个寄存器值，即可得到正弦波表；
5) 控制 DAC 输出时，每隔一段相同的时间从上述正弦波表中取出一个新数据进行输出，即可输出正弦波。改变间隔时间的单位长度，可以改变正弦波曲线的周期。

为方便起见，我们使用了 Python 和 Matlab 脚本制作正弦波表，脚本的代码存储在本工程的目录

下，感兴趣可以打开文件查看，以下列出 Python 脚本代码，见代码清单 37-2。

列表 2: 代码清单 37-2 制作正弦波数据表的 python 脚本
（工程目录下的 sinWave.py 文件）

```
#! python3
#coding=utf-8

"""
Python 版本: 3.x
外部库: matplotlib1.5.3、numpy1.11.2

运行方式:
在命令行中输入: python sinWave.py

运行结果:
命令行中会打印计算得的各点数据,
在当前目录下会生成 py_dac_sinWav.c 文件, 包含上述数据,
并且会弹出描绘曲线的对话框。
"""

import matplotlib.pyplot as plt
import numpy as np
import math

# 修改本变量可以更改点数, 如 16、32、64 等
POINT_NUM = 32

pi = math.pi

# 一个周期 POINT_NUM 个点
n = np.linspace(0,2*pi,POINT_NUM)
```

(下页继续)

(续上页)

```
# 计算 POINT_NUM 个点的正弦值
a = map(math.sin,n)

r =[]
for i in a:
    # 调整幅值至在 0~1 区间
    i+=1

    # 按 3.3V 电压调整幅值
    i*= 3.3/2

    # 求取 dac 数值, 12 位 dac LSB = 3.3V/2**12
    ri = round(i*2**12/3.3)

    # 检查参数
    if ri >= 4095:
        ri = 4095

    # 得到 dac 数值序列
    r.append( ri )

print(list(map(int,r)))

# 写入序列到文件
with open("py_dac_sinWav.c",'w',encoding= 'gb2312') as f:
    print(list(map(int,r)),file= f)

# 绘图
plt.plot(n,r,"-o")
plt.show()
```

Python 脚本的实现原理就是前面介绍的正弦波数据表的制作过程，运行后，该脚本把得到的正弦

波表数据输出到目录下的 py_dac_sinWav.c 文件中，见代码清单 37-3，并且根据取样点描绘出示意图，见图 37-4。Matlab 脚本原理相同，此处不再列出，实际上使用 C 语言也能制作正弦波表，只是画图不方便而已。

列表 3: 代码清单 37-3 生成的正弦波数据表

```
[2048, 2460, 2856, 3218, 3532, 3786, 3969, 4072, 4093, 4031, 3887, 3668,
3382, 3042, 2661, 2255, 1841, 1435, 1054, 714, 428, 209, 65, 3, 24, 127,
310, 564, 878, 1240, 1636, 2048]
```

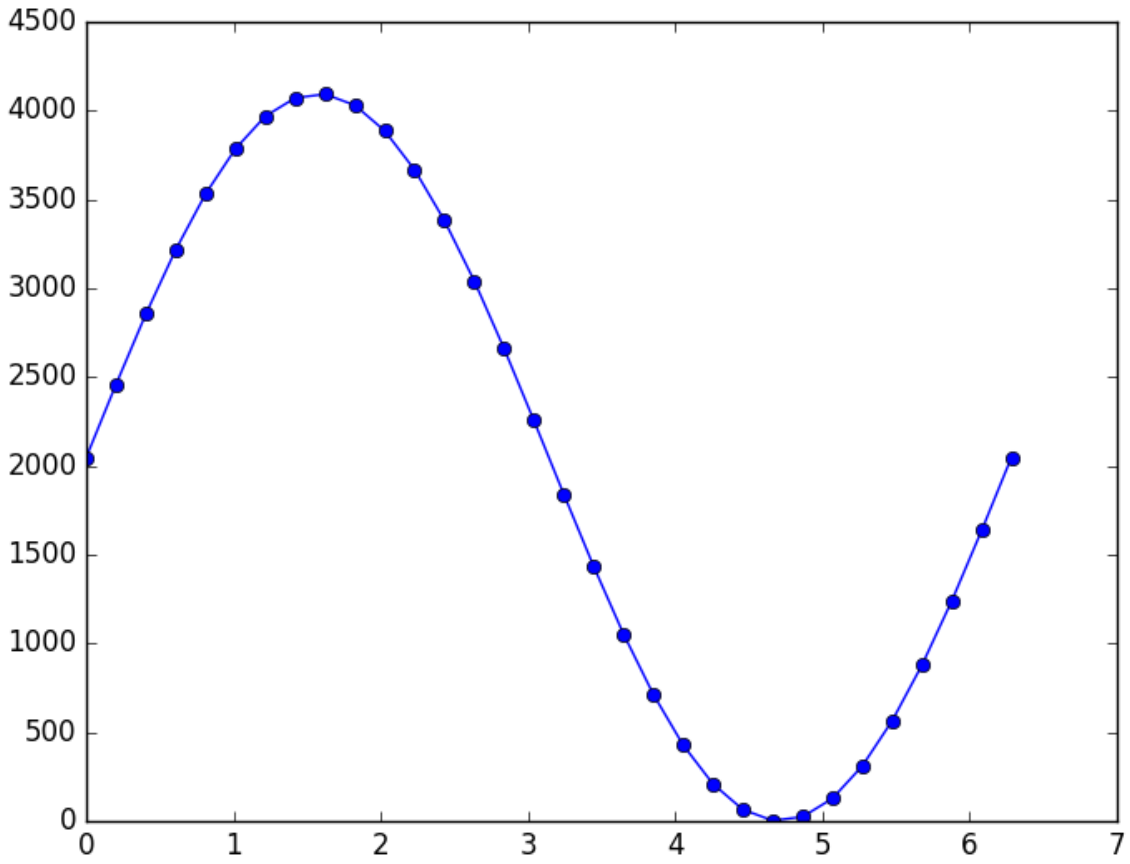


图 37-4 python 脚本根据正弦波表描绘的曲线图

36.4.2.2.2 DAC 宏定义

制作好正弦波数据表后，开始使用 MDK 编写 STM32 的 DAC 工程，首先设置好相关的宏，见*代码清单 37_4*。

列表 4: 代码清单 37-4 DAC 宏定义（bsp_dac.h 文件）

```
//DAC DHR12RD 寄存器，12 位、右对齐、双通道

#define DAC_DHR12RD_ADDRESS (DAC_BASE+0x20)
```

此处定义的宏 DAC_DHR12RD_ADDRESS 是寄存器 DHR12RD 的地址，该寄存器是 12 位右对齐的双通道寄存器，见图 *37_5*。在本实验中将会使用 DMA 把正弦波数据表的点数据赋值到该寄存器中，往该寄存器赋值后的数据会在 DAC 被触发的时候搬运到 2 个 DAC 转换器，然后在这 2 个通道中输出以 12 位右对齐表示的这两个通道的电压。DAC 中还有其它寄存器，它们的功能类似，可以在《STM32 中文参考手册》中了解到。

地址偏移：0x20
复位值：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				DACC2DHR[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留				DACC1DHR[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留。
位27:16	**DACC2DHR[11:0]**：DAC通道2的12位右对齐数据 (DAC channel2 12-bit right-aligned data) 该位由软件写入，表示DAC通道2的12位数据。
位15:12	保留。
位11:0	**DACC1DHR[11:0]**：DAC通道1的12位右对齐数据 (DAC channel1 12-bit right-aligned data) 该位由软件写入，表示DAC通道2的12位数据。

图 37-5 DHR12RD 寄存器说明

与 DAC 控制相关的引脚固定是 PA4 和 PA5，就不使用宏定义了，在源代码中会直接使用引脚号操作。

36.4.2.2.3 DAC GPIO，DMA 和模式配置，

列表 5: 代码清单 37-5 DAC GPIO 和模式配置

```
void HAL_DAC_MspInit(DAC_HandleTypeDef*  DAC_InitStructure)
{
   GPIO_InitTypeDef GPIO_InitStruct;

   /* DAC 外设时钟使能 */
   DACx_CLK_ENABLE();

   /* DMA 控制器时钟使能 */
   __HAL_RCC_DMA2_CLK_ENABLE();

   /* DAC 通道引脚端口时钟使能 */
   DACx_CHANNEL_GPIO_CLK_ENABLE();

   /* DAC 通道引脚配置 */
   GPIO_InitStruct.Pin = DACx_CHANNEL_PIN;
   GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   HAL_GPIO_Init(DACx_CHANNEL_GPIO_PORT, &GPIO_InitStruct);

   /* DMA 控制器初始化 */
   DMA_InitStructure.Instance = DACx_DMAx_CHANNELn;
   DMA_InitStructure.Init.Direction = DMA_MEMORY_TO_PERIPH;
   DMA_InitStructure.Init.PeriphInc = DMA_PINC_DISABLE;
   DMA_InitStructure.Init.MemInc = DMA_MINC_ENABLE;
   DMA_InitStructure.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
   DMA_InitStructure.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
   DMA_InitStructure.Init.Mode = DMA_CIRCULAR;
   DMA_InitStructure.Init.Priority = DMA_PRIORITY_HIGH;
   HAL_DMA_Init(&DMA_InitStructure);
```

(下页继续)

(续上页)

```
    __HAL_LINKDMA( DAC_InitStructure,DMA_Handle2,DMA_InitStructure);
}
```

在 HAL_DAC_MspInit 函数中，完成了 *DAC* 通道的 *GPIO* 的初始化和 *DAC* 模式配置。其中 GPIO 按照要求被配置为模拟输入模式（没有模拟输出模式），在该模式下才能正常输出模拟信号。

配置 DAC 工作模式时，则使用了 DAC_ChannelConfTypeDef 类型的初始化结构体，把 DAC 通道 1 和 2 都配置成了使用定时器 TIM6 触发、不使用波形发生器以及不使用 DAC 输出缓冲的模式。

初始化完 GPIO 和 DAC 模式后，还使用了 __HAL_LINKDMA 函数使能了通道以及 DMA 的请求。由于本实验中对 DAC1 和 2 的操作是同步的，所以只要把 DMA 与 DAC 通道 2 关联起来即可，当使用 DMA 设置通道 2 的数据值时，同时更新通道 1 的内容。

36.4.2.2.4 定时器配置及计算正弦波的频率

初始化完 DAC 后，需要配置触发用的定时器，设定每次触发的间隔，以达到控制正弦波周期的目的。

列表 6: 代码清单 37-6 配置定时器

```
void DAC_Config(void)
{
    DAC_ChannelConfTypeDef sConfig;
    TIM_MasterConfigTypeDef sMasterConfig;

    __HAL_RCC_TIM6_CLK_ENABLE();

    /* DMA 中断配置 */
    HAL_NVIC_SetPriority(DACx_DMAx_CHANNELn_IRQn, 1, 0);
    HAL_NVIC_EnableIRQ(DACx_DMAx_CHANNELn_IRQn);

    /* DAC 初始化 */
```

(下页继续)

(续上页)

```
    DAC_InitStructure.Instance = DACx;
    HAL_DAC_Init(& DAC_InitStructure);

    /* DAC 通道输出配置 */
    sConfig.DAC_Trigger = DAC_TRIGGER_T6_TRGO;
    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE;
    HAL_DAC_ConfigChannel(& DAC_InitStructure, &sConfig, DACx_CHANNEL);

    /* 初始化定时器，用于触发 DAC 更新 */
    TIM_Time.Instance = TIM6;
    TIM_Time.Init.Prescaler = 71;
    TIM_Time.Init.CounterMode = TIM_COUNTERMODE_UP;
    TIM_Time.Init.Period = 20;
    HAL_TIM_Base_Init(&TIM_Time);

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&TIM_Time, &sMasterConfig);
}
```

因为前面的 DAC 配置了 TIM6 当触发源，所以这里将对 TIM 进行配置。TIM 的定时周期被配置为 20，向上计数，不分频。即 TIM2 每隔 20*（1/72M）秒就会触发一次 DAC 事件，作 DAC 触发源使用的定时器并不需要设置中断，当定时器计数器向上计数至指定的值时，产生 Update 事件，同时触发 DAC 把 DHRx 寄存器的数据转移到 DORx，从而开始进行转换。

根据定时器的配置，可推算出正弦波频率的计算方式：

按默认配置，STM32 系统时钟周期为：

$T_{\text{systick}} = 1/72000000$，

定时器 TIM2 的单个时钟周期：

$T_{\text{tim}} = (TIM_Prescaler + 1) \times T_{\text{systick}}$，

定时器触发周期：

$T_{\text{update}} = (TIM_Period + 1) \times T_{\text{tim}}$，

根据正弦波单个周期的点数 N，求出正弦波单个周期时间为：

$T_{\text{sin}} = T_{\text{update}} \times N$，

对应正弦波的频率为：

$$f_{\text{sin}} = \frac{1}{T_{\text{sin}}} = \frac{1}{T_{\text{systick}} \times (TIM_Prescaler + 1) \times (TIM_\text{Period} + 1) \times N}$$

根据上述公式，代入本工程的配置，可得本实验中的正弦波频率为 112500：

$$f_{\text{sin}} = \frac{1}{T_{\text{sin}}} = \frac{72000000}{(0+1) \times (19+1) \times 32} = 112500$$

在实际应用中，可以根据工程里的正弦波点数和定时器配置生成特定频率的正弦波。

36.4.2.2.5 主函数

列表 7: 代码清单 37-7 主函数

```
int main(void)
{
   /* 初始化系统时钟为 72MHz */
   SystemClock_Config();
   /* 初始化 LED */
   DAC_Config();

   /* 启动定时器 */
   HAL_TIM_Base_Start(&TIM_Time);
   /* 启动 DACx DMA 功能 */
   HAL_DAC_Start_DMA(& DAC_InitStructure,DACx_CHANNEL,(uint32_t *)
                     Sine12bit,32,DAC_ALIGN_12B_R);
   /* 无限循环 */
```

(下页继续)

(续上页)

```
    while (1) {

    }
}
```

本工程的主函数非常简单，调用 DAC_Config() 即可完成所有的配置，启动定时器，使能 DMA 之后，使用示波器测量 PA4、PA5 引脚可查看其输出的波形。。

36.4.3 下载验证

注意：参考本章的硬件设计部分说明，实验前需要拆掉 SPI-FLASH 芯片附近的电阻 R6。

用 USB 线连接开发板的“USB 转串口”接口跟电脑，把编译好的程序下载到开发板，使用示波器测量 PA4、PA5 的引脚可看到正弦波形（示波器使用 x10 倍档测量更加准确），见图 37-6，注意观察图中示波器测量出的频率值和电压峰值。

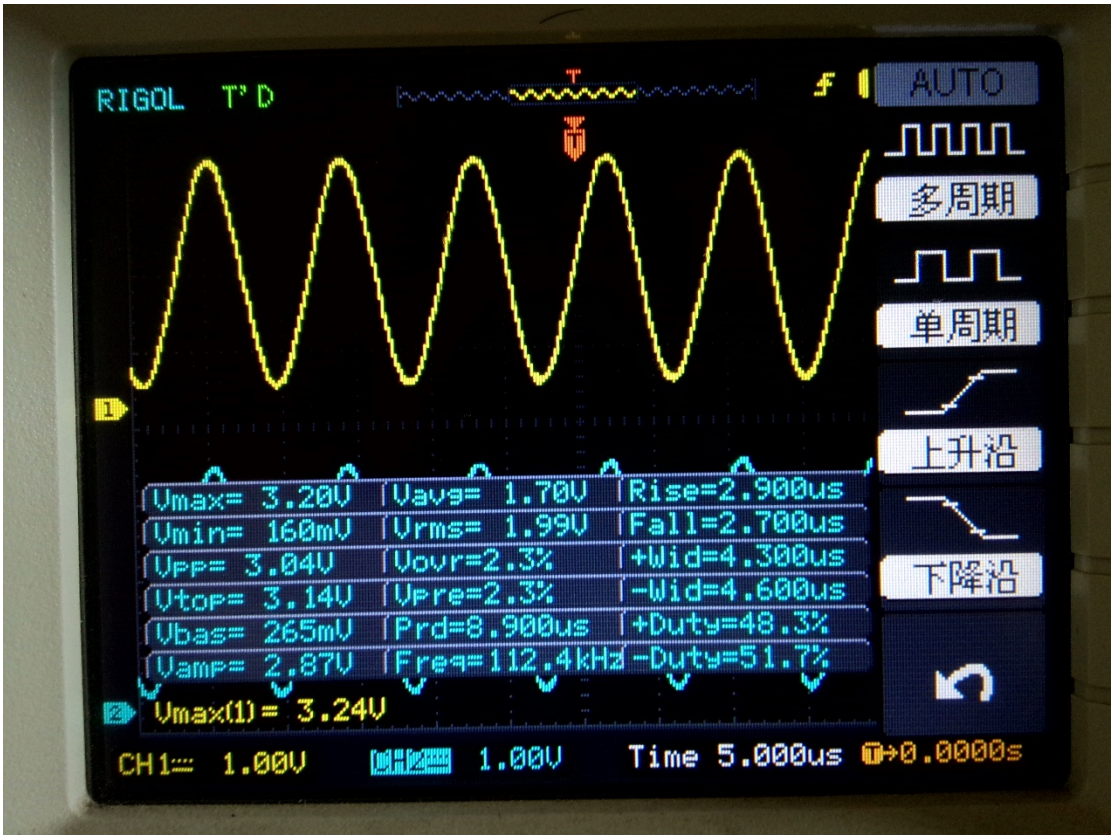


图 37-6 使用示波器测量出的电压波形

36.5 课后练习

1、编程实现：使用 DAC 输出频率为 1000Hz 的正弦波。

第 37 章　电源管理—实现低功耗

本章参考资料：《STM32 参考手册》、《STM32F103CDE 增强型系列数据手册》。

37.1 电源管理相关的库函数及命令

STM32HAL 库对电源管理提供了完善的函数及命令，使用它们可以方便地进行控制，本小节对这些内容进行讲解。

37.1.1 配置 PVD 监控功能

PVD 可监控 VDD 的电压，当它低于阈值时可产生 PVD 中断以让系统进行紧急处理，这个阈值可以直接使用库函数 PWR_PVDLevelConfig 配置成前面表 *38_1* 中说明的阈值等级。

37.1.2 WFI 与 WFE 命令

我们了解到进入各种低功耗模式时都需要调用 WFI 或 WFE 命令，它们实质上都是内核指令，在库文件 core_cm3.h 中把这些指令封装成了函数，见代码清单 *38_1*。

列表 1: 代码清单 38-1 WFI 与 WFE 的指令定义
(core_cm3.h 文件)

```
/** brief  等待中断

    等待中断 是一个暂停执行指令
    暂停至任意中断产生后被唤醒
*/
#define __WFI                                   __wfi
```

(下页继续)

(续上页)

```
/** brief  等待事件

    等待事件 是一个暂停执行指令
    暂停至任意事件产生后被唤醒
*/
#define __WFE                                   __wfe
```

对于这两个指令，我们应用时一般只需要知道，调用它们都能进入低功耗模式，需要使用函数的格式“__WFI();”和“__WFE();”来调用 (因为 __wfi 及 __wfe 是编译器内置的函数，函数内部调用了相应的汇编指令)。其中 WFI 指令决定了它需要用中断唤醒，而 WFE 则决定了它可用事件来唤醒，关于它们更详细的区别可查阅《cortex-CM3/CM4 权威指南》了解。

37.1.3 进入停止模式

直接调用 WFI 和 WFE 指令可以进入睡眠模式，而进入停止模式则还需要在调用指令前设置一些寄存器位，STM32HAL 库把这部分的操作封装到 HAL_PWR_EnterSTOPMode 函数中了，它的定义见代码清单 *38_2*。

列表 2: 代码清单 40-2 进入停止模式

```
/**
* @brief 进入停止模式
* @note 在停止模式下所有 I/O 都会保持在停止前的状态
* @note 从停止模式唤醒后，会使用 HSI 作为时钟源
* @note 调压器若工作在低功耗模式，可减少功耗，但唤醒时会增加延迟
* @param Regulator: 设置停止模式时调压器的工作模式
*        @arg PWR_MAINREGULATOR_ON: 调压器正常运行
*        @arg PWR_LOWPOWERREGULATOR_ON: 调压器低功耗运行
* @param STOPEntry: 设置使用 WFI 还是 WFE 进入停止模式
```

(下页继续)

(续上页)

```
*         @arg PWR_STOPENTRY_WFI: WFI 进入停止模式
*         @arg PWR_STOPENTRY_WFE: WFE 进入停止模式
* @retval None
*/
void HAL_PWR_EnterSTOPMode(uint32_t Regulator, uint8_t STOPEntry)
{
    uint32_t tmpreg = 0;

    /* 检查参数是否合法 */
    assert_param(IS_PWR_REGULATOR(Regulator));
    assert_param(IS_PWR_STOP_ENTRY(STOPEntry));

    /* 设置调压器的模式 ---------------------------------*/
    tmpreg = PWR->CR1;
    /* 清除 PDDS 及 LPDS 位 */
    tmpreg &= (uint32_t)~(PWR_CR1_PDDS | PWR_CR1_LPDS);

    /* 根据 PWR_Regulator 的值 (调压器工作模式) 配置 LPDS,MRLVDS 及 LPLVDS 位 */
    tmpreg |= Regulator;

    /* 写入参数值到寄存器 */
    PWR->CR1 = tmpreg;

    /* 设置内核寄存器的 SLEEPDEEP 位 */
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;

    /* 设置进入停止模式的方式 ------------------------------------------------
↪*/
    if (STOPEntry == PWR_STOPENTRY_WFI) {
        /* 需要中断唤醒 */
        __WFI();
    } else {
```

(下页继续)

(续上页)

```
        /* 需要事件唤醒 */
        __SEV();
        __WFE();
        __WFE();
    }
    /* 以下的程序是当重新唤醒时才执行的，清除 SLEEPDEEP 位的状态 */
    SCB->SCR &= (uint32_t)~((uint32_t)SCB_SCR_SLEEPDEEP_Msk);
}
```

这个函数有两个输入参数，分别用于控制调压器的模式及选择使用 WFI 或 WFE 停止，代码中先是根据调压器的模式配置 PWR_CR 寄存器，再把内核寄存器的 SLEEPDEEP 位置 1，这样再调用 WFI 或 WFE 命令时，STM32 就不是睡眠，而是进入停止模式了。函数结尾处的语句用于复位 SLEEPDEEP 位的状态，由于它是在 WFI 及 WFE 指令之后的，所以这部分代码是在 STM32 被唤醒的时候才会执行。

要注意的是进入停止模式后，STM32 的所有 I/O 都保持在停止前的状态，而当它被唤醒时，STM32 使用 HSI 作为系统时钟 (8MHz) 运行，由于系统时钟会影响很多外设的工作状态，所以一般我们在唤醒后会重新开启 HSE，把系统时钟设置回原来的状态。

37.1.4 进入待机模式

类似地，STM32 HAL 库也提供了控制进入待机模式的函数，其定义见。

列表 3: 代码清单 38-3 进入待机模式

```
/**
* @brief 进入待机模式
* @note 待机模式时，除了以下引脚，其余引脚都在高阻态：
*          - 复位引脚
*          - RTC_AF1 引脚 (PC13)(需要使能侵入检测、时间戳事件或 RTC 闹钟事件)
*          - RTC_AF2 引脚 (PI8) (需要使能侵入检测或时间戳事件)
*          - WKUP 引脚 (PA0) (需要使能 WKUP 唤醒功能)
```

(下页继续)

(续上页)

```
* @retval None
*/
void HAL_PWR_EnterSTANDBYMode(void)
{
    /* 选择待机模式 */
    PWR->CR1 |= PWR_CR1_PDDS;

    /* 设置内核寄存器的 SLEEPDEEP 位 */
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;

    /* 存储操作完毕时才能进入待机模式，使用以下语句确保存储操作执行完毕 */
#if defined ( __CC_ARM)
    __force_stores();
#endif
    /* 等待中断唤醒 */
    __WFI();
}
```

该函数中先配置了 PDDS 寄存器位及 SLEEPDEEP 寄存器位，接着调用 __force_stores 函数确保存储操作完毕后再调用 WFI 指令，从而进入待机模式。这里值得注意的是，待机模式也可以使用 WFE 指令进入的，如果您有需要可以自行修改。

在进入待机模式后，除了被使能了的用于唤醒的 I/O，其余 I/O 都进入高阻态，而从待机模式唤醒后，相当于复位 STM32 芯片，程序重新从头开始执行。

37.2 PWR—睡眠模式实验

在本小节中，我们以实验的形式讲解如何控制 STM32 进入低功耗睡眠模式。

37.2.1 硬件设计

实验中的硬件主要使用到了按键、LED 彩灯以及使用串口输出调试信息，这些硬件都与前面相应实验中的一致，涉及到硬件设计的可参考原理图或前面章节中的内容。

37.2.2 软件设计

本小节讲解的是“PWR—睡眠模式”实验，请打开配套的代码工程阅读理解。

37.2.2.1 程序设计要点

(1) 初始化用于唤醒的中断按键；

(2) 进入睡眠状态；

(3) 使用按键中断唤醒芯片；

37.2.2.2 代码分析

37.2.2.2.1 main 函数

睡眠模式的程序比较简单，我们直接阅读它的 main 函数了解执行流程，见*代码清单 38_4*。

列表 4: 代码清单 38-4 睡眠模式的 main 函数 (main.c 文件)

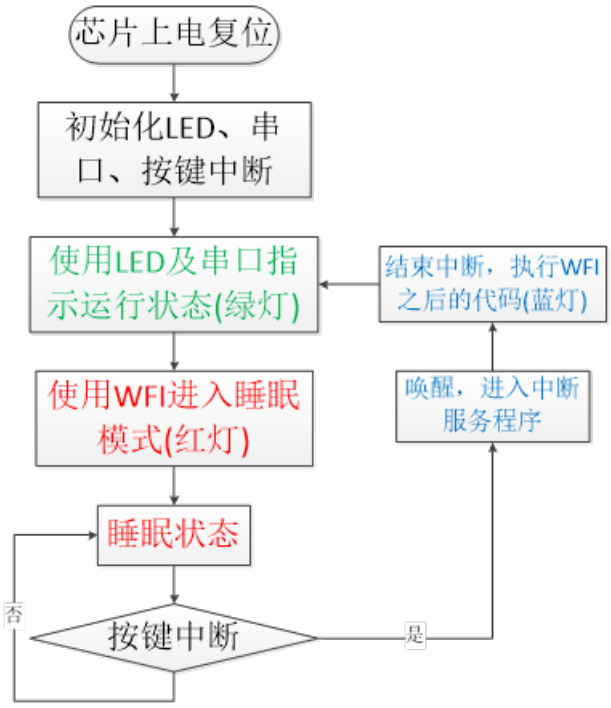
```
int main(void)
{
    /* 初始化系统时钟为 72MHz */
    SystemClock_Config();
    /* 初始化 LED */
    LED_GPIO_Config();
    /* 初始化调试串口，一般为串口 1 */
    DEBUG_USART_Config();
    /*
    初始化按键为中断模式，按下中断后会进入中断
    服务函数  */
    EXTI_Key_Config();

    printf("\r\n 欢迎使用野火  STM32 F103 开发板。\r\n");
    printf("\r\n 野火 F103 睡眠模式例程\r\n");

    printf("\r\n 实验说明：\r\n");

    printf("\r\n 1. 本程序中，绿灯表示 STM32 正常运行，红灯表示睡眠状态，蓝灯表示刚从睡眠状态被唤醒\r\n");
    printf("\r\n 2. 程序运行一段时间后自动进入睡眠状态，在睡眠状态下，可使用 KEY1 或 KEY2 唤醒\r\n");
    printf("\r\n 3. 本实验执行这样一个循环：\r\n ------》亮绿灯 （正常运行）-> 亮红灯（睡眠模式）-> 按 KEY1 或 KEY2 唤醒-> 亮蓝灯 （刚被唤醒）-----》\r\n");
    printf("\r\n 4. 在睡眠状态下，DAP 下载器无法给 STM32 下载程序\r\n 可按 KEY1、KEY2 唤醒后下载，\r\n 或按复位键使芯片处于复位状态，然后在电脑上点击下载按钮，再释放复位按键，即可下载 \r\n");

    while (1) {
        /********** 执行任务 ***************************/
```

（下页继续）

(续上页)

```
        printf("\r\n STM32 正常运行，亮绿灯\r\n");

        LED_GREEN;
        HAL_Delay(2000);
        /***** 任务执行完毕，进入睡眠降低功耗 ***********/

        printf("\r\n 进入睡眠模式，亮红灯，按 KEY1 或 KEY2 按键可唤醒\r\n");

        //使用红灯指示，进入睡眠状态
        LED_RED;
        //暂停滴答时钟，防止通过滴答时钟中断唤醒
        HAL_SuspendTick();
        //进入睡眠模式
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI);
        //等待中断唤醒  K1 或 K2 按键中断
        /*** 被唤醒，亮蓝灯指示 ***/
        LED_BLUE;
        //被唤醒后，恢复滴答时钟
        HAL_ResumeTick();
        HAL_Delay(2000);

        printf("\r\n 已退出睡眠模式\r\n");
        //继续执行 while 循环

    }
}
```

这个 main 函数的执行流程见图 *38_1*。



(1) 程序中首先初始化了 LED 灯及串口以便用于指示芯片的运行状态，并且把实验板上的两个按键都初始化成了中断模式，以便当系统进入睡眠模式的时候可以通过按键来唤醒。这些硬件的初始化过程都跟前面章节中的一模一样。

(2) 初始化完成后使用 LED 及串口表示运行状态，在本实验中，LED 彩灯为绿色时表示正常运行，红灯时表示睡眠状态，蓝灯时表示刚从睡眠状态中被唤醒。

(3) 程序执行一段时间后，直接使用 WFI 指令进入睡眠模式，由于 WFI 睡眠模式可以使用任意中断唤醒，所以我们可以使用按键中断唤醒。在实际应用中，您也可以把它改成串口中断、定时器中断等。

(4) 当系统进入停止状态后，我们按下实验板上的 KEY1 或 KEY2 按键，即可使系统回到正常运行的状态，当执行完中断服务函数后，会继续执行 WFI 指令后的代码。

37.2.2.2.2 中断服务函数

系统刚被唤醒时会进入中断服务函数，见*代码清单 38_5*。

列表 5: 代码清单 38-5 按键中断的服务函数 (stm32f1xx_it.c 文件)

```
void KEY1_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(KEY1_INT_GPIO_PIN);
}

void KEY2_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(KEY2_INT_GPIO_PIN);
}
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    LED_BLUE;
    if (GPIO_Pin==KEY1_INT_GPIO_PIN)
        printf("\r\n KEY1 按键中断唤醒 \r\n");
    else if (GPIO_Pin==KEY2_INT_GPIO_PIN)
        printf("\r\n KEY2 按键中断唤醒 \r\n");
    else {
    }
}
```

用于唤醒睡眠模式的中断，其中断服务函数也没有特殊要求，跟普通的应用一样。

37.2.3 下载验证

下载这个实验测试时，可连接上串口，在电脑端的串口调试助手获知调试信息。当系统进入睡眠状态的时候，可以按 KEY1 或 KEY2 按键唤醒系统。

注意：

当系统处于睡眠模式低功耗状态时 (包括后面讲解的停止模式及待机模式)，使用 DAP 下载器是无法给芯片下载程序的，所以下载程序时要先把系统唤醒。或者使用如下方法：按着板子的复位按键，使系统处于复位状态，然后点击电脑端的下载按钮下载程序，这时再释放复位按键，就能正常给板子下载程序了。

37.3 PWR—停止模式实验

在睡眠模式实验的基础上，我们进一步讲解如何进入停止模式及唤醒后的状态恢复。

37.3.1 硬件设计

本实验中的硬件与睡眠模式中的一致，主要使用到了按键、LED 彩灯以及使用串口输出调试信息。

37.3.2 软件设计

本小节讲解的是“PWR—停止模式”实验，请打开配套的代码工程阅读理解。

37.3.2.1 程序设计要点

(1) 初始化用于唤醒的中断按键；

(2) 选择电压调节器的工作模式并进入停止状态；

(3) 使用按键中断唤醒芯片；

(4) 重启 HSE 时钟，使系统完全恢复停止前的状态。

37.3.2.2 代码分析

37.3.2.2.1 重启 HSE 时钟

与睡眠模式不一样，系统从停止模式被唤醒时，是使用 HSI 作为系统时钟的，在 STM32F103 中，HSI 时钟一般为 8MHz，与我们常用的 72MHz 相关太远，它会影响各种外设的工作频率。所以在系统从停止模式唤醒后，若希望各种外设恢复正常的工作状态，就要恢复停止模式前使用的系统时钟，本实验中定义了一个 SYSCLKConfig_STOP 函数，用于恢复系统时钟，它的定义见*代码清单 38_6*。

列表 6: 代码清单 38-6 恢复系统时钟 (main.c 文件)

```
static void SYSCLKConfig_STOP(void)
{
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    uint32_t pFLatency = 0;

    /* 启用电源控制时钟 */
    __HAL_RCC_PWR_CLK_ENABLE();

    /* 根据内部 RCC 寄存器获取振荡器配置 */
    HAL_RCC_GetOscConfig(&RCC_OscInitStruct);

    /* 从停止模式唤醒后重新配置系统时钟:
    启用 HSE 和 PLL */
    RCC_OscInitStruct.OscillatorType  = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState        = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState    = RCC_PLL_ON;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        while (1) {
```

(下页继续)

(续上页)

```
            ;
        }
    }

    /* 根据内部 RCC 寄存器获取时钟配置 */
    HAL_RCC_GetClockConfig(&RCC_ClkInitStruct, &pFLatency);

    /* 选择 PLL 作为系统时钟源，并配置 HCLK、PCLK1 和 PCLK2 时钟分频系数 */
    RCC_ClkInitStruct.ClockType      = RCC_CLOCKTYPE_SYSCLK;
    RCC_ClkInitStruct.SYSCLKSource   = RCC_SYSCLKSOURCE_PLLCLK;
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, pFLatency) != HAL_OK) {
        while (1) {
            ;
        }
    }
}
```

这个函数主要是调用了各种 RCC 相关的库函数，开启了 HSE 时钟、使能 PLL 并且选择 PLL 作为时钟源，从而恢复停止前的时钟状态。

37.3.2.2.2 main 函数

停止模式实验的 main 函数流程与睡眠模式的类似，主要是调用指令方式的不同及唤醒后增加了恢复时钟的操作，见*代码清单 38_7*。

列表 7: 代码清单 38-7 停止模式的 main 函数 (main.c 文件)

```
int main(void)
{
    uint32_t SYSCLK_Frequency=0;
```

(下页继续)

(续上页)

```
    uint32_t HCLK_Frequency=0;
    uint32_t PCLK1_Frequency=0;
    uint32_t PCLK2_Frequency=0;
    uint32_t SYSCLK_Source=0;

    /* 初始化系统时钟为 72MHz */
    SystemClock_Config();
    /* 初始化 LED */
    LED_GPIO_Config();
    /* 初始化调试串口，一般为串口 1 */
    DEBUG_USART_Config();
    /*
    初始化按键为中断模式，按下中断后会进入中断
    服务函数  */
    EXTI_Key_Config();

    printf("\r\n 欢迎使用野火  STM32 F103 开发板。\r\n");
    printf("\r\n 野火 F103 停止模式例程\r\n");

    printf("\r\n 实验说明：\r\n");

    printf("\r\n 1. 本程序中，绿灯表示 STM32 正常运行，红灯表示睡眠状态，蓝灯表示刚从
停止状态被唤醒\r\n");
    printf("\r\n 2. 程序运行一段时间后自动进入停止状态，在停止状态下，可使用 KEY1 或
KEY2 唤醒\r\n");
    printf("\r\n 3. 本实验执行这样一个循环：\r\n ------》亮绿灯 （正常运行）-> 亮红灯
（停止模式）-> 按 KEY1 或 KEY2 唤醒-> 亮蓝灯 （刚被唤醒）-----》\r\n");
    printf("\r\n 4. 在停止状态下，DAP 下载器无法给 STM32 下载程序\r\n 可按 KEY1、
KEY2 唤醒后下载，\r\n 或按复位键使芯片处于复位状态，然后在电脑上点击下载按钮，再释放复位
按键，即可下载\r\n");

    while (1) {
```

(下页继续)

(续上页)

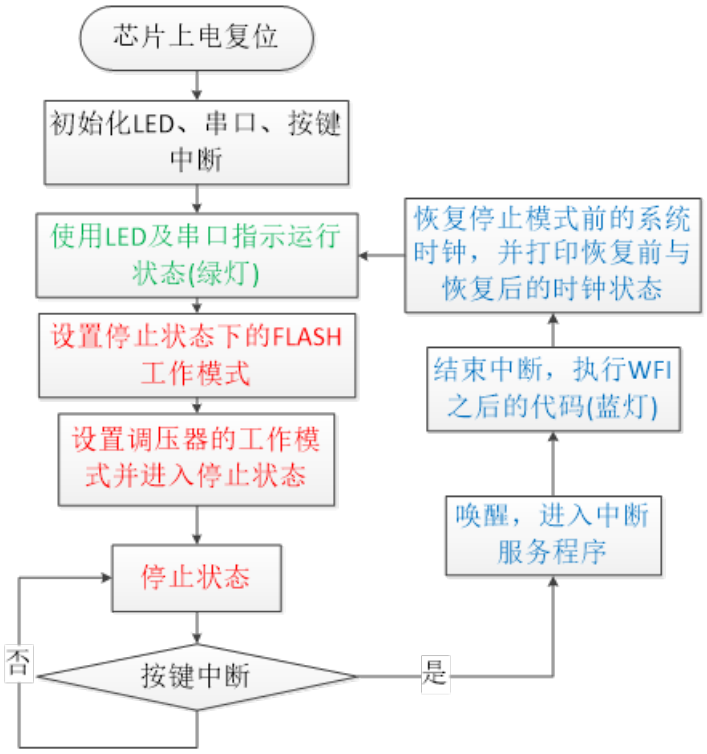
```
/********** 执行任务 ***************************/
printf("\r\n STM32 正常运行，亮绿灯\r\n");

LED_GREEN;
HAL_Delay(2000);
/***** 任务执行完毕，进入睡眠降低功耗 ***********/

printf("\r\n 进入停止模式，亮红灯，按 KEY1 或 KEY2 按键可唤醒\r\n");

//使用红灯指示，进入睡眠状态
LED_RED;
//暂停滴答时钟，防止通过滴答时钟中断唤醒
HAL_SuspendTick();
/* 设置停止模式时，FLASH 进入掉电状态 */
HAL_PWREx_EnableFlashPowerDown();
/*
进入停止模式，设置电压调节器为低功耗模式，
等待中断唤醒 */
HAL_PWR_EnterSTOPMode(PWR_MAINREGULATOR_ON,PWR_STOPENTRY_WFI);
//等待中断唤醒  K1 或 K2 按键中断
/*** 被唤醒，亮蓝灯指示 ***/
LED_BLUE;
//根据时钟寄存器的值更新 SystemCoreClock 变量
SystemCoreClockUpdate();
//获取唤醒后的时钟状态
SYSCLK_Frequency = HAL_RCC_GetSysClockFreq();
HCLK_Frequency   = HAL_RCC_GetHCLKFreq();
PCLK1_Frequency  = HAL_RCC_GetPCLK1Freq();
PCLK2_Frequency  = HAL_RCC_GetPCLK2Freq();
SYSCLK_Source    = __HAL_RCC_GET_SYSCLK_SOURCE();
//这里由于串口直接使用 HSI 时钟，不会会影响串口波特率
printf("\r\n刚唤醒的时钟状态：\r\n");
```

(下页继续)

(续上页)

```
        printf(" SYSCLK 频率:%d,\r\n HCLK 频率:%d,\r\n PCLK1 频率:%d,\r\n␣
↪PCLK2 频率:%d,\r\n 时钟源:%d (0 表示 HSI, 8 表示 PLLCLK)\n",SYSCLK_Frequency,
↪HCLK_Frequency,PCLK1_Frequency,PCLK2_Frequency,SYSCLK_Source);
        /* 从停止模式唤醒后配置系统时钟: 启用 HSE、PLL*/
        /* 选择 PLL 作为系统时钟源 (HSE 和 PLL 在停止模式下禁用)*/

        SYSCLKConfig_STOP();
        //被唤醒后, 恢复滴答时钟
        HAL_ResumeTick();
        //获取重新配置后的时钟状态
        SYSCLK_Frequency = HAL_RCC_GetSysClockFreq();
        HCLK_Frequency   = HAL_RCC_GetHCLKFreq();
        PCLK1_Frequency  = HAL_RCC_GetPCLK1Freq();
        PCLK2_Frequency  = HAL_RCC_GetPCLK2Freq();
        SYSCLK_Source    = __HAL_RCC_GET_SYSCLK_SOURCE();

        //重新配置时钟源后始终状态
        printf("\r\n重新配置后的时钟状态: \r\n");
        printf(" SYSCLK 频率:%d,\r\n HCLK 频率:%d,\r\n PCLK1 频率:%d,\r\n␣
↪PCLK2 频率:%d,\r\n 时钟源:%d (0 表示 HSI, 8 表示 PLLCLK)\n",SYSCLK_Frequency,
↪HCLK_Frequency,PCLK1_Frequency,PCLK2_Frequency,SYSCLK_Source);
        HAL_Delay(2000);
        printf("\r\n 已退出停止模式\r\n");
        //继续执行 while 循环
    }
}
```

这个 main 函数的执行流程见图 *38_2*。



(1) 程序中首先初始化了 LED 灯及串口以便用于指示芯片的运行状态，并且把实验板上的两个按键都初始化成了中断模式，以便当系统进入停止模式的时候可以通过按键来唤醒。这些硬件的初始化过程都跟前面章节中的一模一样。

(2) 初始化完成后使用 LED 及串口表示运行状态，在本实验中，LED 彩灯为绿色时表示正常运行，红灯时表示停止状态，蓝灯时表示刚从停止状态中被唤醒。在停止模式下，I/O 口会保持停止前的状态，所以 LED 彩灯在停止模式时也会保持亮红灯。

(3) 程序执行一段时间后，我们先用库函数 HAL_PWREx_EnableFlashPowerDown 设置 FLASH 的在停止状态时使用掉电模式，接着调用库函数 HAL_PWR_EnterSTOPMode 把调压器设置在低功耗模式，进入停止状态。由于 WFI 停止模式可以使用任意 EXTI 的中断唤醒，所以我们可以使用按键中断唤醒。

(4) 当系统进入睡眠状态后，我们按下实验板上的 KEY1 或 KEY2 按键，即可唤醒系统，当执行完中断服务函数后，会继续执行 HAL_PWR_EnterSTOPMode 函数后的代码。

(5) 为了更清晰地展示停止模式的影响，在刚唤醒后，我们调用了库函数 SystemCoreClock-Update、HAL_RCC_GetSysClockFreq、HAL_RCC_GetHCLKFreq、HAL_RCC_GetPCLK1Freq、

HAL_RCC_GetPCLK2Freq、__HAL_RCC_GET_SYSCLK_SOURCE 函数获取刚唤醒后的系统的时钟源以及时钟频率，并通过串口打印出来。在使用 SYSCLKConfig_STOP 函数恢复时钟后，我们再次获取这些时频率，最后再通过串口打印出来。

(6) 通过串口调试信息我们会知道刚唤醒时系统时钟使用的是 HSI 时钟，频率为 8MHz，恢复后的系统时钟采用 HSE 倍频后的 PLL 时钟，时钟频率为 72MHz。

37.3.3 下载验证

下载这个实验测试时，可连接上串口，在电脑端的串口调试助手获知调试信息。当系统进入停止状态的时候，可以按 KEY1 或 KEY2 按键唤醒系统。

注意：

当系统处于停止模式低功耗状态时 (包括睡眠模式及待机模式)，使用 DAP 下载器是无法给芯片下载程序的，所以下载程序时要先把系统唤醒。或者使用如下方法：按着板子的复位按键，使系统处于复位状态，然后点击电脑端的下载按钮下载程序，这时再释放复位按键，就能正常给板子下载程序了。

37.4 STM32 的电源管理简介

电源对电子设备的重要性不言而喻，它是保证系统稳定运行的基础，而保证系统能稳定运行后，又有低功耗的要求。在很多应用场合中都对电子设备的功耗要求非常苛刻，如某些传感器信息采集设备，仅靠小型的电池提供电源，要求工作长达数年之久，且期间不需要任何维护；由于智慧穿戴设备的小型化要求，电池体积不能太大导致容量也比较小，所以也很有必要从控制功耗入手，提高设备的续行时间。因此，STM32 有专门的电源管理外设监控电源并管理设备的运行模式，确保系统正常运行，并尽量降低器件的功耗。

37.4.1 电源监控器

STM32 芯片主要通过引脚 VDD 从外部获取电源，在它的内部具有电源监控器用于检测 VDD 的电压，以实现复位功能及掉电紧急处理功能，保证系统可靠地运行。

37.4.1.1 上电复位与掉电复位 (POR 与 PDR)

当检测到 VDD 的电压低于阈值 VPOR 及 VPDR 时，无需外部电路辅助，STM32 芯片会自动保持在复位状态，防止因电压不足强行工作而带来严重的后果。见图 *38_3*，在刚开始电压低于 VPOR 时 (约 1.92V)，STM32 保持在上电复位状态 (POR，Power On Reset)，当 VDD 电压持续上升至大于 VPOR 时，芯片开始正常运行，而在芯片正常运行的时候，当检测到 VDD 电压下降至低于 VPDR 阈值 (约 1.88V)，会进入掉电复位状态 (PDR，Power Down Reset)。

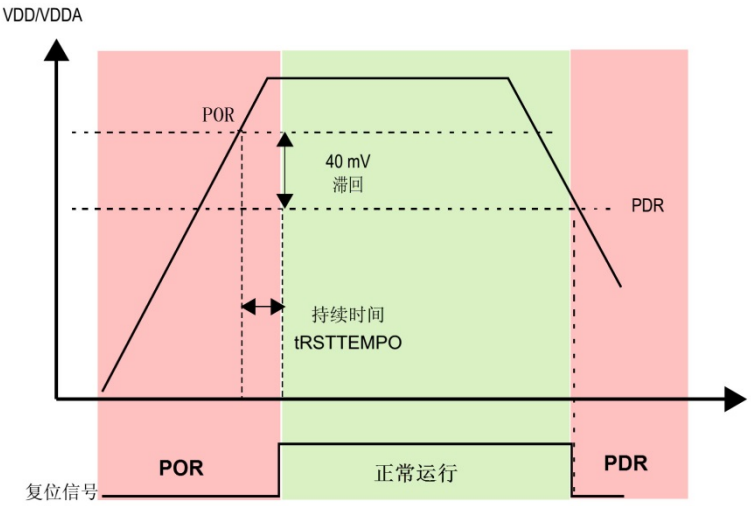


图 38-3 POR 与 PDR

37.4.1.2 可编程电压检测器 PVD

上述 POR、PDR 功能是使用其电压阈值与外部供电电压 VDD 比较，当低于工作阈值时，会直接进入复位状态，这可防止电压不足导致的误操作。除此之外，STM32 还提供了可编程电压检测器 PVD，它也是实时检测 VDD 的电压，当检测到电压低于编程的 VPVD 阈值时，会向内核产生

一个 PVD 中断 (EXTI16 线中断) 以使内核在复位前进行紧急处理。该电压阈值可通过电源控制寄存器 PWR_CSR 设置。

使用 PVD 可配置 8 个等级，见表 *38_1*。其中的上升沿和下降沿分别表示类似图 *38_3* 中 VDD 电压上升过程及下降过程的阈值。

表 38-1 PVD 的阈值等级

阈值等级	条件	最小值	典型值	最大值	单位
级别 0	上升沿	2.1	2.18	2.26	V
	下降沿	2	2.08	2.16	V
级别 1	上升沿	2.19	2.28	2.37	V
	下降沿	2.09	2.18	2.27	V
级别 2	上升沿	2.28	2.38	2.48	V
	下降沿	2.18	2.28	2.38	V
级别 3	上升沿	2.38	2.48	2.58	V
	下降沿	2.28	2.38	2.48	V
级别 4	上升沿	2.47	2.58	2.69	V
	下降沿	2.37	2.48	2.59	V
级别 5	上升沿	2.57	2.68	2.79	V
	下降沿	2.47	2.58	2.69	V
级别 6	上升沿	2.66	2.78	2.9	V
	下降沿	2.56	2.68	2.8	V
级别 7	上升沿	2.76	2.88	3	V
	下降沿	2.66	2.78	2.9	V

37.4.2 STM32 的电源系统

为了方便进行电源管理，STM32 把它的外设、内核等模块跟据功能划分了供电区域，其内部电源区域划分见图 *38_4*。

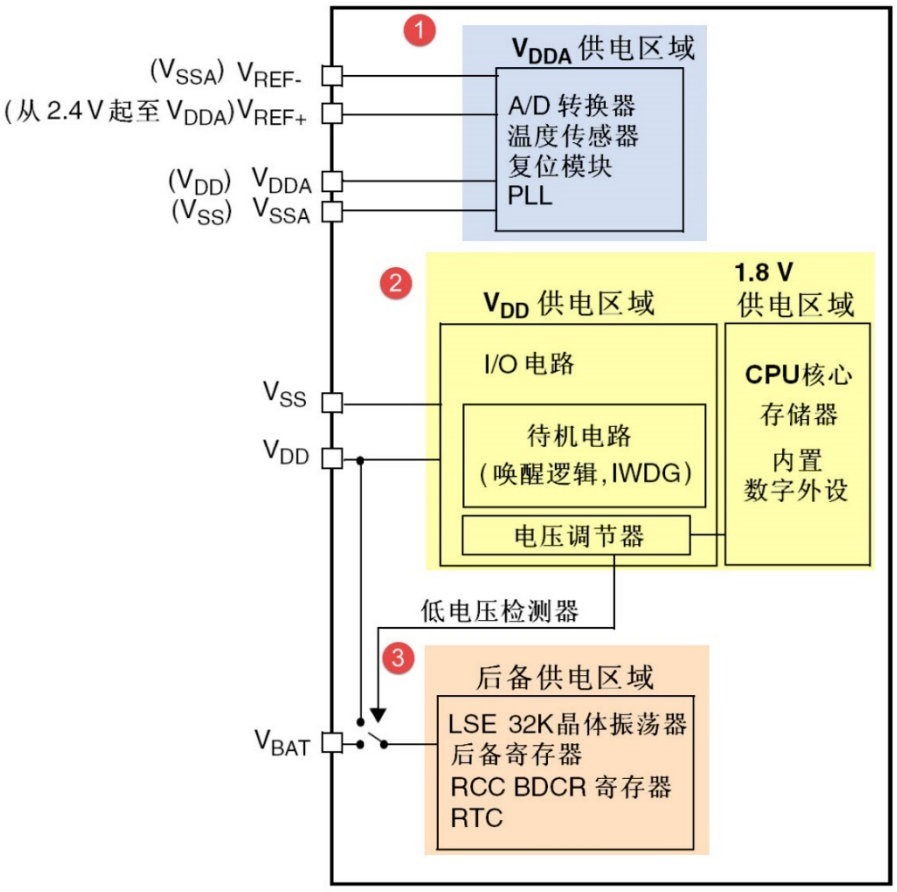


图 38-4 STM32 的电源系统

从框图了解到，STM32 的电源系统主要分为备份域电路、内核电路以及 ADC 电路三部分，介绍如下：

- ADC 电源及参考电压（V_{DDA} 供电区域）

  为了提高转换精度，STM32 的 ADC 配有独立的电源接口，方便进行单独的滤波。ADC 的工作电源使用 V_{DDA} 引脚输入，使用 V_{SSA} 作为独立的地连接，V_{REF} 引脚则为 ADC 提供测量使用的参考电压。

- 调压器供电电路（V_{DD}/1.8V 供电区域）

  在 STM32 的电源系统中调压器供电的电路是最主要的部分，调压器为备份域及待机电路以外的所有数字电路供电，其中包括内核、数字外设以及 RAM，调压器的输出电压约为 1.8V，因而使用调压器供电的这些电路区域被称为 1.8V 域。

调压器可以运行在“运行模式”、“停止模式”以及“待机模式”。在运行模式下，1.8V 域全功率运行；在停止模式下 1.8V 域运行在低功耗状态，1.8V 区域的所有时钟都被关闭，相应的外设都停止了工作，但它会保留内核寄存器以及 SRAM 的内容；在待机模式下，整个 1.8V 域都断电，该区域的内核寄存器及 SRAM 内容都会丢失 (备份区域的寄存器不受影响)。

- 备份域电路（后备供电区域）

  STM32 的 LSE 振荡器、RTC 及备份寄存器这些器件被包含进备份域电路中，这部分的电路可以通过 STM32 的 V_{BAT} 引脚获取供电电源，在实际应用中一般会使用 3V 的钮扣电池对该引脚供电。

  在图中备份域电路的左侧有一个电源开关结构，它的功能类似图 40-5 中的双二极管，在它的“1”处连接了 V_{BAT} 电源，“2”处连接了 V_{DD} 主电源 (一般为 3.3V)，右侧“3”处引出到备份域电路中。当 V_{DD} 主电源存在时，由于 V_{DD} 电压较高，备份域电路通过 V_{DD} 供电，节省钮扣电池的电源，仅当 V_{DD} 掉电时，备份域电路由钮扣电池通过 V_{BAT} 供电，保证电路能持续运行，从而可利用它保留关键数据。

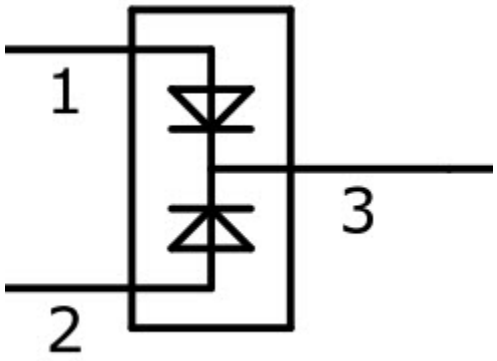


图 38-5 双二极管结构

37.4.3 STM32 的功耗模式

按功耗由高到低排列，STM32 具有运行、睡眠、停止和待机四种工作模式。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低功耗模式降低功耗，这三种模式中，电源消耗不同、唤醒时间不同、唤醒源不同，用户需要根据应用需求，选择最佳的低功耗模式。三种低功耗的模式说明见表 40-2。

表 40-2STM32 的低功耗模式说明

模式	说明	进入方式	唤醒方式	对 1.8V 区域时钟的影响	对 VDD 区域时钟的影响	调压器
睡眠	内核停止，所有外设包括 M3 核心的外设，如 NVIC、系统时钟(SysTick) 等仍在运行	调用 WFI 命令	任一中断	内核时钟关，对其他时钟和 ADC 时钟无影响	无	开
		调用 WFE 命令	唤醒事件			
停止	所有的时钟都已停止	配置 PWR_CR 寄存器的 PDDS +LPDS 位 +SLEEPDEEP 位 +WFI 或 WFE 命令	任一外部中断（在外部中断寄存器中设置)	关闭所有 1.8V 区域的时钟	HSI 和 HSE 的振荡器关闭	开启或处于低功耗模式（依据电源控制寄存器的设定)
待机	1.8V 电源关闭	配置 PWR_CR 寄存器的 PDDS +SLEEPDEEP 位 +WFI 或 WFE 命令	WKUP 引脚的上升沿、RTC 闹钟事件、NRST 引脚上的外部复位、IWDG 复位			关

从表中可以看到，这三种低功耗模式层层递进，运行的时钟或芯片功能越来越少，因而功耗越来越低。

37.4.3.1 睡眠模式

在睡眠模式中，仅关闭了内核时钟，内核停止运行，但其片上外设，CM3 核心的外设全都还照常运行。有两种方式进入睡眠模式，它的进入方式决定了从睡眠唤醒的方式，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。睡眠模式的各种特性见表 40-3。

表 40-3 睡眠模式的各种特性

特性	说明
立即睡眠	在执行 WFI 或 WFE 指令时立即进入睡眠模式。
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。
进入方式	内核寄存器的 SLEEPDEEP = 0 ，然后调用 WFI 或 WFE 指令即可进入睡眠模式； 另外若内核寄存器的 SLEEPONEXIT=0 时，进入“立即睡眠”模式，SLEEPONEXIT=1 时，进入“退出时睡眠”模式。
唤醒方式	如果是使用 WFI 指令睡眠的，则可使用任意中断唤醒； 如果是使用 WFE 指令睡眠的，则由事件唤醒。
睡眠时	关闭内核时钟，内核停止，而外设正常运行，在软件上表现为不再执行新的代码。这个状态会保留睡眠前的内核寄存器、内存的数据。
唤醒延迟	无延迟。
唤醒后	若由中断唤醒，先进入中断，退出中断服务程序后，接着执行 WFI 指令后的程序；若由事件唤醒，直接接着执行 WFE 后的程序。

37.4.3.2 停止模式

在停止模式中，进一步关闭了其它所有的时钟，于是所有的外设都停止了工作，但由于其 1.8V 区域的部分电源没有关闭，还保留了内核的寄存器、内存的信息，所以从停止模式唤醒，并重新开启时钟后，还可以从上次停止处继续执行代码。停止模式可以由任意一个外部中断 (EXTI) 唤醒，在停止模式中可以选择电压调节器为开模式或低功耗模式。停止模式的各种特性见表 40-4。

表 40-4 停止模式的各种特性

特性	说明
调压器低功耗模式	在停止模式下调压器可工作在正常模式或低功耗模式，可进一步降低功耗
进入方式	内核寄存器的 SLEEPDEEP =1 , PWR_CR 寄存器中的 PDDS=0 , 然后调用 WFI 或 WFE 指令即可进入停止模式； PWR_CR 寄存器的 LPDS=0 时，调压器工作在正常模式，LPDS=1 时工作在低功耗模式；
唤醒方式	如果是使用 WFI 指令睡眠的，可使用任意 EXTI 线的中断唤醒； 如果是使用 WFE 指令睡眠的，可使用任意配置为事件模式的 EXTI 线事件唤醒。
停止时	内核停止，片上外设也停止。这个状态会保留停止前的内核寄存器、内存的数据。
唤醒延迟	基础延迟为 HSI 振荡器的启动时间，若调压器工作在低功耗模式，还需要加上调压器从低功耗切换至正常模式下的时间。
唤醒后	若由中断唤醒，先进入中断，退出中断服务程序后，接着执行 WFI 指令后的程序； 若由事件唤醒，直接接着执行 WFE 后的程序。唤醒后，STM32 会使用 HSI 作为系统时钟。

37.4.3.3 待机模式

待机模式，它除了关闭所有的时钟，还把 1.8V 区域的电源也完全关闭了，也就是说，从待机模式唤醒后，由于没有之前代码的运行记录，只能对芯片复位，重新检测 boot 条件，从头开始执行程序。它有四种唤醒方式，分别是 WKUP(PA0) 引脚的上升沿，RTC 闹钟事件，NRST 引脚的复位和 IWDG(独立看门狗) 复位。

表 40-5 待机模式的各种特性

特性	说明
进入方式	内核寄存器的 SLEEPDEEP =1，PWR_CR 寄存器中的 PDDS=1，PWR_CR 寄存器中的唤醒状态位 WUF=0，然后调用 WFI 或 WFE 指令即可进入待机模式；
唤醒方式	通过 WKUP 引脚的上升沿，RTC 闹钟、唤醒、入侵、时间戳事件或 NRST 引脚外部复位及 IWDG 复位唤醒。
待机时	内核停止，片上外设也停止；内核寄存器、内存的数据会丢失；除复位引脚、RTC_AF1 引脚及 WKUP 引脚，其它 I/O 口均工作在高阻态。
唤醒延迟	芯片复位的时间
唤醒后	相当于芯片复位，在程序表现为从头开始执行代码。

在以上讲解的睡眠模式、停止模式及待机模式中，若备份域电源正常供电，备份域内的 RTC 都可以正常运行，备份域内的寄存器的数据会被保存，不受功耗模式影响。

37.5 PWR—待机模式实验

最后我们来学习最低功耗的待机模式。

37.5.1 硬件设计

本实验中的硬件与睡眠模式、停止模式中的一致，主要使用到了按键、LED 彩灯以及使用串口输出调试信息。要强调的是，由于 WKUP 引脚 (PA0) 必须使用上升沿才能唤醒待机状态的系统，所以我们硬件设计的 PA0 引脚连接到按键 KEY1，且按下按键的时候会在 PA0 引脚产生上升沿，从而可实现唤醒的功能，按键的具体电路请查看配套的原理图。

37.5.2 软件设计

本小节讲解的是“PWR—待机模式”实验，请打开配套的代码工程阅读理解。

37.5.2.1 程序设计要点

(1) 清除 WUF 标志位；

(2) 使能 WKUP 唤醒功能；

(3) 进入待机状态。

37.5.2.2 代码分析

37.5.2.2.1 main 函数

待机模式实验的执行流程比较简单，见*代码清单 38_8*。

列表 8: 代码清单 38-8 停止模式的 main 函数 (main.c 文件)

```
int main(void)
{
    /* 初始化系统时钟为 72MHz */
    SystemClock_Config();
```

(下页继续)

(续上页)

```
    /* 初始化 LED */
    LED_GPIO_Config();
    /* 初始化调试串口，一般为串口 1 */
    DEBUG_USART_Config();
    /* 初始化按键，不需要中断，
    仅初始化 KEY2 即可，只用于唤醒的 PA0 引脚不需要这样
    初始化 */
    Key_GPIO_Config();

    printf("\r\n 欢迎使用野火  STM32 F103 开发板。\r\n");
    printf("\r\n 野火 F103 待机模式例程\r\n");

    printf("\r\n 实验说明：\r\n");

    printf("\r\n 1. 本程序中，绿灯表示本次复位是上电或引脚复位，红灯表示即将进入待机状
态，蓝灯表示本次是待机唤醒的复位\r\n");
    printf("\r\n 2. 长按 KEY2 按键后，会进入待机模式\r\n");
    printf("\r\n 3. 在待机模式下，按 KEY1 按键可唤醒，唤醒后系统会进行复位，程序从头
开始执行\r\n");
    printf("\r\n 4. 可通过检测 WU 标志位确定复位来源\r\n");

    printf("\r\n 5. 在待机状态下，DAP 下载器无法给 STM32 下载程序需要唤醒后才能下载
↪");

    //检测复位来源
    if (__HAL_PWR_GET_FLAG(PWR_FLAG_SB) == SET) {
        __HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);
        LED_BLUE;
        printf("\r\n 待机唤醒复位 \r\n");
    } else {
        LED_GREEN;
        printf("\r\n 非待机唤醒复位 \r\n");
```

(下页继续)

(续上页)

```
    }

    while (1) {
        // K2 按键长按进入待机模式
        if (KEY2_LongPress()) {

            printf("\r\n即将进入待机模式，进入待机模式后可按 KEY1 唤醒，唤醒后会进行
复位，程序从头开始执行\r\n");
            LED_RED;
            HAL_Delay(1000);

            /* 清除 WU 状态位 */
            __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);

            /* 使能 WKUP 引脚的唤醒功能 ，使能 PA0*/
            HAL_PWR_EnableWakeUpPin( 0x00000100U);

            //暂停滴答时钟，防止通过滴答时钟中断唤醒
            HAL_SuspendTick();
            /* 进入待机模式 */
            HAL_PWR_EnterSTANDBYMode();
        }
    }
}
```

这个 main 函数的执行流程见图 40-1。

图 40-6 待机模式实验流程图

(1) 程序中首先初始化了系统时钟、LED 灯及串口以便用于指示芯片的运行状态，由于待机模式唤醒使用 WKUP 引脚并不需要特别的引脚初始化，所以我们调用的按键初始化函数 Key_GPIO_Config 它的内部只初始化了 KEY2 按键，而且是普通的输入模式，对唤醒用的 PA0 引脚可以不初始化。当然，如果不初始化 PA0 的话，在正常运行模式中 KEY1 按键是不能正常运

行的，我们这里只是强调待机模式的 WKUP 唤醒不需要中断，也不需要像按键那样初始化。本工程中使用的 Key_GPIO_Config 函数定义如代码清单 38_9 所示。

列表 9: 代码清单 38-9 Key_GPIO_Config 函数 (bsp_key.c 文件)

```
void Key_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启按键 GPIO 口的时钟 */
    KEY2_GPIO_CLK_ENABLE();
    /* 选择按键的引脚 */
    GPIO_InitStructure.Pin = KEY2_PIN;

    /* 设置引脚为输入模式 */
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;

    /* 设置引脚不上拉也不下拉 */
    GPIO_InitStructure.Pull = GPIO_NOPULL;

    /* 使用上面的结构体初始化按键 */
    HAL_GPIO_Init(KEY2_GPIO_PORT, &GPIO_InitStructure);
}
```

(2) 使使用库函数 __HAL_PWR_GET_FLAG 检测 PWR_FLAG_SB 标志位，当这个标志位为 SET 状态的时候，表示本次系统是从待机模式唤醒的复位，否则可能是上电复位。我们利用这个区分两种复位形式，分别使用蓝色 LED 灯或绿色 LED 灯来指示。

(3) 在 while 循环中，使用自定义的函数 KEY2_LongPress 来检测 KEY2 按键是否被长时间按下，若长时间按下则进入待机模式，否则继续 while 循环。KEY2_LongPress 函数不是本章分析的重点，感兴趣的读者请自行查阅工程中的代码。

(4) 检测到 KEY2 按键被长时间按下，要进入待机模式。在使用库函数 HAL_PWR_EnableWakeUpPin 发送待机命令前，要先使用库函数

__HAL_PWR_CLEAR_FLAG 清除 PWR_FLAG_WU 标志位，并且使用库函数 HAL_PWR_EnableWakeUpPin 使能 WKUP 唤醒功能，这样进入待机模式后才能使用 WKUP 唤醒。

(5) 在进入待机模式前我们控制了 LED 彩灯为红色，但在待机状态时，由于 I/O 口会处于高阻态，所以 LED 灯会熄灭。

(6) 按下 KEY1 按键，会使 PA0 引脚产生一个上升沿，从而唤醒系统。

(7) 系统唤醒后会进行复位，从头开始执行上述过程，与第一次上电时不同的是，这样的复位会使 PWR_FLAG_SB 标志位改为 SET 状态，所以这个时候 LED 彩灯会亮蓝色。

(1) 下载验证

下载这个实验测试时，可连接上串口，在电脑端的串口调试助手获知调试信息。长按实验板上的 KEY2 按键，系统会进入待机模式，按 KEY1 按键可唤醒系统。

注意：

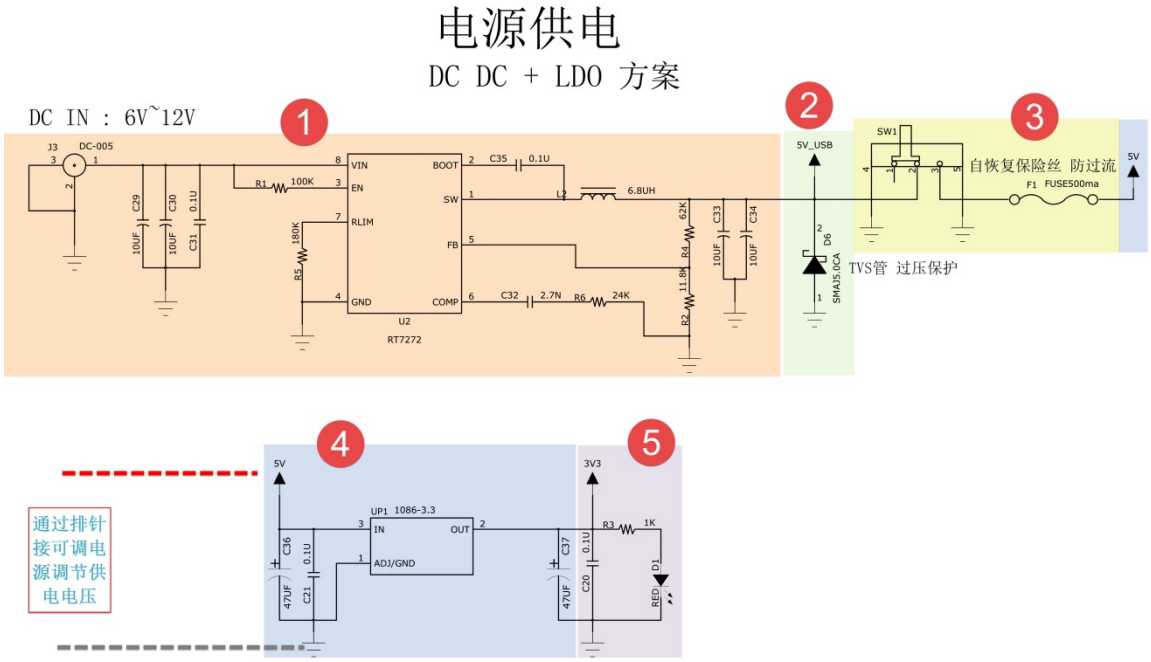
当系统处于待机模式低功耗状态时 (包括睡眠模式及停止模式)，使用 DAP 下载器是无法给芯片下载程序的，所以下载程序时要先把系统唤醒。或者使用如下方法：按着板子的复位按键，使系统处于复位状态，然后点击电脑端的下载按钮下载程序，这时再释放复位按键，就能正常给板子下载程序了。

37.6 PWR—PVD 电源监控实验

这一小节我们学习如何使用 PVD 监控供电电源，增强系统的鲁棒性。

37.6.1 硬件设计

本实验中使用 PVD 监控 STM32 芯片的 VDD 引脚，当监测到供电电压低于阈值时会产生 PVD 中断，系统进入中断服务函数进入紧急处理过程。所以进行这个实验时需要使用一个可调的电压源给实验板供电，改变给 STM32 芯片的供电电压，为此我们需要先了解实验板的电源供电系统，见图 *38_7*。



整个电源供电系统主要分为以下五部分：

(1) 6-12V 的 DC 电源供电系统，这部分使用 DC 电源接口引入 6-12V 的电源，经过 RT7272 进行电压转换成 5V 电源，再与第二部分的“5V_USB”电源线连接在一起。

(2) 第二部分使用 USB 接口，使用 USB 线从外部引入 5V 电源，引入的电源经过电源开关及保险丝连接到“5V”电源线。

(3) 第三部分的是电源开关及保险丝，即当我们的实验板使用 DC 电源或“5V_USB”线供电时，可用电源开关控制通断，保险丝也会起保护作用。

(4) “5V”电源线遍布整个板子，板子上各个位置引出的标有“5V”丝印的排针都与这个电源线直接相连。5V 电源线给板子上的某些工作电压为 5V 的芯片供电。5V 电源还经过 LDO 稳压芯片，输出 3.3V 电源连接到“3.3V”电源线。

(5) 同样地，“3.3V”电源线也遍布整个板子，各个引出的标有“3.3V”丝印的排针都与它直接相连，3.3V 电源给工作电压为 3.3V 的各种芯片供电。STM32 芯片的 VDD 引脚就是直接与这个 3.3V 电源相连的，所以通过 STM32 的 PVD 监控的就是这个“3.3V”电源线的电压。

当我们进行这个 PVD 实验时，为方便改变“3.3V”电源线的电压，我们可以把可调电源通过实验板上引出的“5V”及“GND”排针给实验板供电，当可调电源电压降低时，LDO 在“3.3V”电

源线的供电电压会随之降低，即 STM32 的 PVD 监控的 VDD 引脚电压会降低，这样我们就可以模拟 VDD 电压下降的实验条件，对 PVD 进行测试了。不过，由于这样供电不经过保险丝，所以在调节电压的时候要小心，不要给它供电远高于 5V，否则可能会烧坏实验板上的芯片。

37.6.2 软件设计

本小节讲解的是“PWR—睡眠模式”实验，请打开配套的代码工程阅读理解。为了方便把这个工程的 PVD 监控功能移植到其它应用，我们把 PVD 电压监控相关的主要代码编都写到“bsp_pvd.c”及“bsp_pvd.h”文件中，这些文件是我们自己编写的，不属于 HAL 库的内容，可根据您的喜好命名文件。

37.6.2.1 程序设计要点

(1) 初始化 PVD 中断；

(2) 设置 PVD 电压监控等级并使能 PVD；

(3) 编写 PVD 中断服务函数，处理紧急任务。

37.6.2.2 代码分析

37.6.2.2.1 初始化 PVD

使用 PVD 功能前需要先初始化，我们把这部分代码封装到 PVD_Config 函数中，见*代码清单 38_10*。

列表 10: 代码清单 38-10 初始化 PVD(bsp_pvd.c 文件)

```
void PVD_Config(void)
{
    PWR_PVDTypeDef sConfigPVD;
```

(下页继续)

(续上页)

```
    /* 使能 PWR 时钟 */
    __PWR_CLK_ENABLE();
    /* 配置 PVD 中断 */
    /* 中断设置，抢占优先级 0，子优先级为 0*/
    HAL_NVIC_SetPriority(PVD_IRQn, 0 ,0);
    HAL_NVIC_EnableIRQ(PVD_IRQn);

    /* 配置 PVD 级别 5 (PVD 检测电压的阈值为 2.8V，
    VDD 电压低于 2.8V 时产生 PVD 中断，具体数据
    可查询数据手册获知) 具体级别根据自己的
    实际应用要求配置 */
    sConfigPVD.PVDLevel = PWR_PVDLEVEL_5;
    sConfigPVD.Mode = PWR_PVD_MODE_IT_RISING_FALLING;
    HAL_PWR_ConfigPVD(&sConfigPVD);
    /* 使能 PVD 输出 */
    HAL_PWR_EnablePVD();
}
```

21 }

在这段代码中，执行的流程如下：

(1) 使能电源管理时钟。

(2) 配置 PVD 的中断优先级。由于电压下降是非常危急的状态，所以请尽量把它配置成最高优先级。

(3) 使用库函数 HAL_PWR_ConfigPVD 设置 PVD 监控的电压阈值等级，各个阈值等级表示的电压值请查阅表 *38_1* 或 STM32 的数据手册。

(4) 最后使用库函数 HAL_PWR_EnablePVD 使能 PVD 功能。

37.6.2.2.2 PVD 中断服务函数

配置完成 PVD 后，还需要编写中断服务函数，在其中处理紧急任务，本工程的 PVD 中断服务函数见*代码清单 38_11*。

列表 11: 代码清单 38-11 PVD 中断服务函数 (stm32f1xx_it.c 文件)

```
void PVD_IRQHandler(void)
{
    HAL_PWR_PVD_IRQHandler();
}
/**
* @brief  PWR PVD interrupt callback
* @param  None
* @retval None
*/
void HAL_PWR_PVDCallback(void)
{
    /* 亮红灯，实际应用中应进入紧急状态处理 */
    LED_RED;
}
```

注意这个中断服务函数的名是 PVD_IRQHandler 而不是 EXTI16_IRQHandler(STM32 没有这样的中断函数名)，示例中我们仅点亮了 LED 红灯，不同的应用中要根据需求进行相应的紧急处理。

37.6.2.2.3 main 函数

本电源监控实验的 main 函数执行流程比较简单，仅调用了 PVD_Config 配置监控功能，当 VDD 供电电压正常时，板子亮绿灯，当电压低于阈值时，会跳转到中断服务函数中，板子亮红灯，见*代码清单 38_12*。

列表 12: 代码清单 38-12 停止模式的 main 函数 (main.c 文件)

```
int main(void)
{
    /* 配置系统时钟为 180 MHz */
    SystemClock_Config();
    /* 初始化 LED */
    LED_GPIO_Config();
    //亮绿灯，表示正常运行
    LED_GREEN;

    //配置 PVD，当电压过低时，会进入中断服务函数，亮红灯
    PVD_Config();

    while (1) {
        /* 正常运行的程序 */
    }
}
```

37.6.3 下载验证

本工程的验证步骤如下：

(1) 通过电脑把本工程编译并下载到实验板；

(2) 把下载器、USB 及 DC 电源等外部供电设备都拔掉；

(3) 按“硬件设计”小节中的说明，使用可调电源通过“5V”及“GND”排针给实验板供 5V 电源；(注意要先调好可调电源的电压再连接，防止烧坏实验板)

(4) 复位实验板，确认板子亮绿灯，表示正常状态；

(5) 持续降低可调电源的输出电压，直到实验板亮红灯，这时表示 PVD 检测到电压低于阈值。

本工程中，我们实测 PVD 阈值等级为“PWR_PVDLevel_2V6”时，当可调电源电压降至 4.1V 时，板子亮红灯，此时的“3.3V”电源引脚的实测电压为 2.55V，您可以设置其它电压阈值来进行测试。

注意：

由于这样使用可调电源供电不经过保险丝，所以在调节电压的时候要小心，不要给它供电远高于 5V，否则可能会烧坏实验板上的芯片。

第 38 章　RTC—实时时钟

本章参考资料：《STM32F103 增强型系列数据手册》以及《STM32F10X-中文参考手册》的《电源控制 PWR》及《实时时钟 RTC》章节。

38.1 RTC 实时时钟简介

STM32 的 RTC 外设（Real Time Clock），实质是一个 *掉电后还继续运行的定时器*。从定时器的角度来说，相对于通用定时器 TIM 外设，它十分简单，只有很纯粹的计时和触发中断的功能；但从 *掉电还继续运行的角度来说*，它却是 STM32 中唯一一个具有如此强大功能的外设。所以 RTC 外设的复杂之处并不在于它的定时功能，而在于它掉电还继续运行的特性。

以上所说的掉电，是指 *主电源 VDD* 断开的情况，为了 RTC 外设 *掉电继续运行*，必须接上锂电池给 STM32 的 RTC、备份发卡通过 V_{BAT} 引脚供电。当主电源 V_{DD} 有效时，由 V_{DD} 给 RTC 外设供电；而当 V_{DD} 掉电后，由 V_{BAT} 给 RTC 外设供电。但无论由什么电源供电，RTC 中的数据都保存在属于 RTC 的 *备份域中*，若主电源 V_{DD} 和 V_{BAT} 都掉电，那么 *备份域中保存的所有数据将丢失*。*备份域*除了 RTC 模块的寄存器，还有 42 个 16 位的寄存器可以在 V_{DD} 掉电的情况下保存用户程序的数据，系统复位或电源复位时，这些数据也不会被复位。

从 RTC 的定时器特性来说，它是一个 *32 位*的计数器，只能向上计数。它使用的时钟源有三种，分别为高速外部时钟的 128 分频（HSE/128）、低速内部时钟 LSI 以及低速外部时钟 LSE；使 HSE 分频时钟或 LSI 的话，在主电源 V_{DD} 掉电的情况下，这两个时钟来源都会受到影响，因此没法保证 RTC 正常工作。因此 RTC 一般使用 *低速外部时钟 LSE*，在设计中，频率通常为实时时钟模块中常用的 32.768KHz，这是因为 $32768 = 2^{15}$，分频容易实现，所以它被广泛应用到 RTC 模块。在主电源 V_{DD} 有效的情况下 (待机)，RTC 还可以配置 *闹钟事件*使 STM32 退出待机模式。

38.2 RTC 外设框图剖析

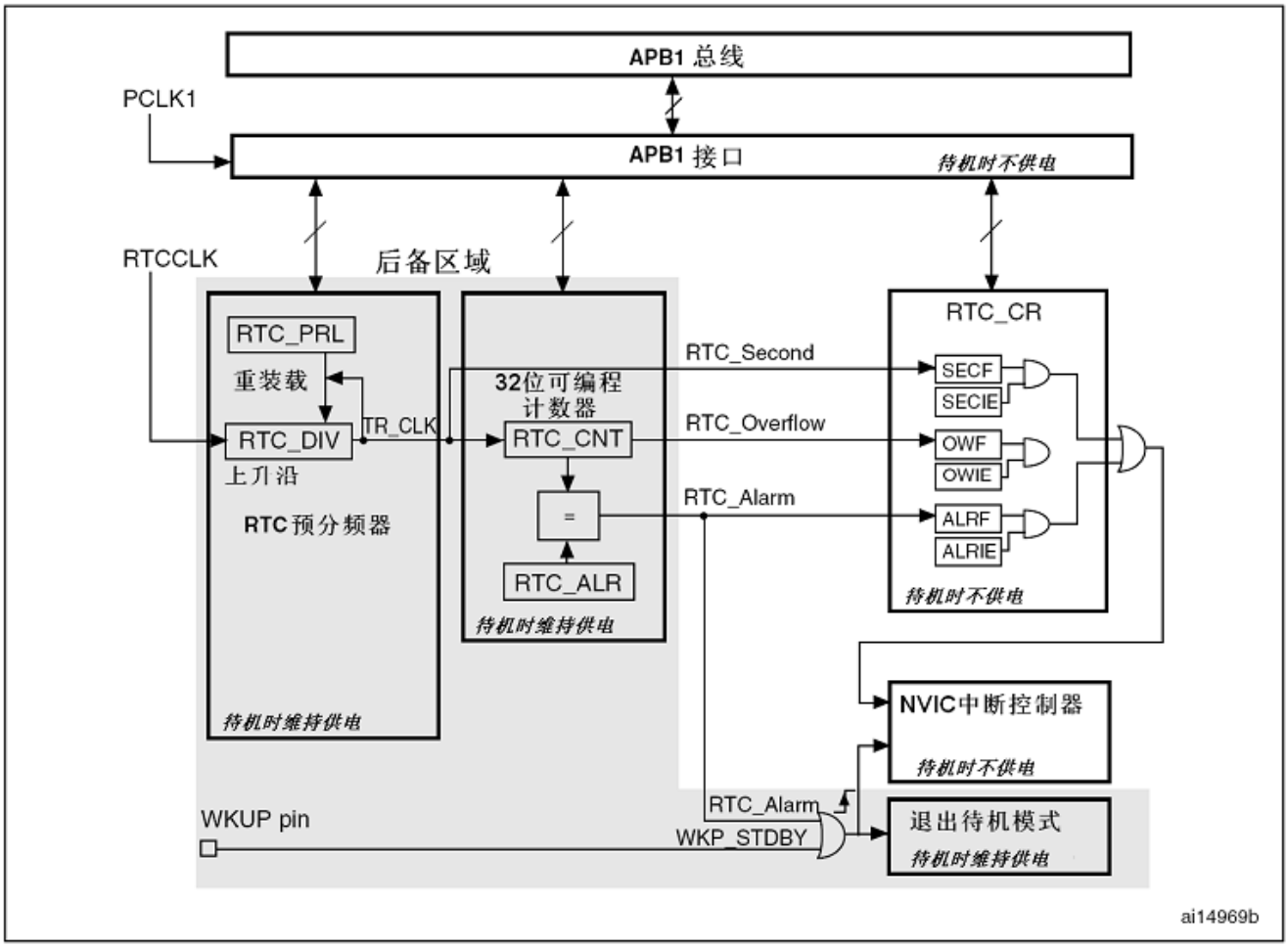


图 39-1 RTC 架构图

框图中浅灰色的部分都是属于备份域的，在 V_{DD} 掉电时可在 V_{BAT} 的驱动下继续运行。这部分仅包括 RTC 的分频器，计数器，和闹钟控制器。若 V_{DD} 电源有效，RTC 可以触发 *RTC_Second(秒中断)*、*RTC_Overflow(溢出事件)* 和 *RTC_Alarm(闹钟中断)*。从结构图可以分析到，其中的定时器溢出事件无法被配置为中断。若 STM32 原本处于待机状态，可由闹钟事件或 *WKUP* 事件 (外部唤醒事件，属于 EXTI 模块，不属于 RTC) 使它退出待机模式。闹钟事件是在 * 计数器 RTC_CNT* 的值等于闹钟寄存器 *RTC_ALR* 的值时触发的。

在备份域中所有寄存器都是 16 位的，RTC 控制相关的寄存器也不例外。它的计数器 *RTC_CNT* 的

32 位由 *RTC_CNTL* 和 *RTC_CNTH* 两个寄存器组成，分别保存定时计数值的低 16 位和高 16 位。在配置 RTC 模块的时钟时，通常把输入的 32768Hz 的 *RTCCLK* 进行 32768 分频得到实际驱动计数器的时钟 *TR_CLK* = RTCCLK/32768= 1 Hz，计时周期为 1 秒，计时器在 *TR_CLK* 的驱动下计数，即每秒计数器 *RTC_CNT* 的值加 1。

由于备份域的存在，使得 RTC 核具有了完全独立于 APB1 接口的特性，也因此对 RTC 寄存器的访问要遵守一定的规则。

系统复位后，默认禁止访问后备寄存器和 RTC，防止对后备区域 (BKP) 的意外写操作。执行以下操作使能对后备寄存器和 RTC 的访问：

(1) 设置 *RCC_APB1ENR* 寄存器的 *PWREN* 和 *BKPEN* 位来使能电源和后备接口时钟。

(2) 设置 *PWR_CR* 寄存器的 *DBP* 位使能对后备寄存器和 *RTC* 的访问。

设置后备寄存器为可访问后，在第一次通过 APB1 接口访问 RTC 时，因为时钟频率的差异，所以必须等待 APB1 与 RTC 外设同步，确保被读取出来的 RTC 寄存器值是正确的。若在同步之后，一直没有关闭 APB1 的 RTC 外设接口，就不需要再次同步了。

如果内核要对 RTC 寄存器进行任何的写操作，在内核发出写指令后，RTC 模块在 3 个 *RTCCLK* 时钟之后，才开始正式的写 RTC 寄存器操作。由于 *RTCCLK* 的频率比内核主频低得多，所以每次操作后必须要检查 *RTC* 关闭操作标志位 *RTOFF*，当这个标志被置 1 时，写操作才正式完成。

当然，以上的操作都具有库函数，读者不必具体地查阅寄存器。

38.3 UNIX 时间戳

在使用 RTC 外设前，还需要引入 UNIX 时间戳的概念。

如果从现在起，把计数器 *RTC_CNT* 的计数值置 0，然后每秒加 1，*RTC_CNT* 什么时候会溢出呢？由于 RTC_CNT 是 32 位寄存器，可存储的最大值为 (2^{32}-1)，即这样计时的话，在 2:sup:*32* 秒后溢出，即它将在今后的 136 年时溢出：

N = 2^{32}/365/24/60/60 ≈136 年

假如某个时刻读取到计数器的数值为 X = 60*60*24*2，即两天时间的秒数，而假设又知道计数器是在 2011 年 1 月 1 日的 0 时 0 分 0 秒置 0 的，那么就可以根据计数器的这个相对时间数值，计

算得这个 X 时刻是 2011 年 1 月 3 日的 0 时 0 分 0 秒了。而计数器则会在 (2011+136) 年左右溢出，也就是说到了（2011+136）年时，如果我们还在使用这个计数器提供时间的话就会出现问题。在这个例子中，定时器被置 0 的这个时间被称为计时元年，相对计时元年经过的秒数称为时间戳，也就是计数器中的值。

大多数操作系统都是利用时间戳和计时元年来计算当前时间的，而这个时间戳和计时元年大家都取了同一个标准——*UNIX* 时间戳和 *UNIX* 计时元年。*UNIX* 计时元年被设置为格林威治时间 1970 年 1 月 1 日 0 时 0 分 0 秒，大概是为了纪念 UNIX 的诞生的时代吧，而 *UNIX* 时间戳即为当前时间相对于 UNIX 计时元年经过的秒数。因为 unix 时间戳主要用来表示当前时间或者和电脑有关的日志时间（如文件创立时间，log 发生时间等），考虑到所有电脑文件不可能在 1970 年前创立，所以用 unix 时间戳很少用来表示 1970 前的时间。

在这个计时系统中，使用的是有符号的 32 位整型变量来保存 UNIX 时间戳的，即实际可用计数位数比我们上面例子中的少了一位，少了这一位，*UNIX* 计时元年也相对提前了，这个计时方法在 2038 年 1 月 19 日 03 时 14 分 07 秒将会发生溢出，这个时间离我们并不远。由于 UNIX 时间戳被广泛应用到各种系统中，溢出可能会导致系统发生严重错误，届时，很可能会重演一次“千年虫”的问题，所以在设计预期寿命较长的设备需要注意。

在网络上搜索“UNIX 时间戳”可找到一些网站提供当前实时的 UNIX 时间戳，见图 *39_2*。

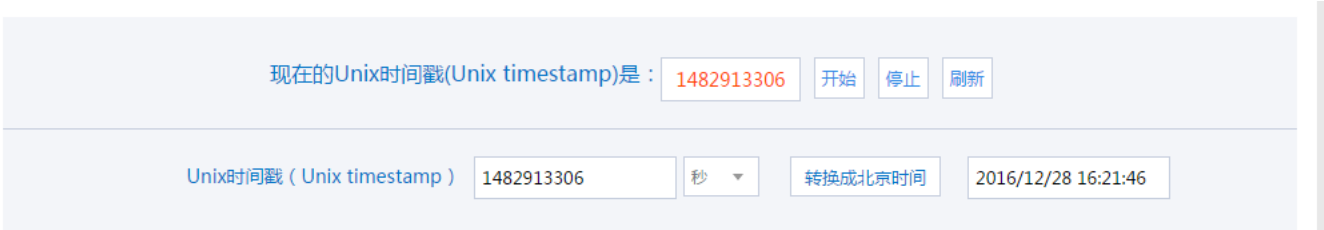


图 39-2 某些网站显示的实时 UNIX 时间戳

38.4 与 RTC 控制相关的库函数

STM32HAL 库对 RTC 控制提供了完善的函数，使用它们可以方便地进行控制，本小节对这些内容进行讲解。

38.4.1 初始化结构体讲解

HAL 库函数对每个外设都建立了一个初始化结构体，比如 RTC_InitTypeDef，结构体成员用于设置外设工作参数，并由外设初始化配置函数，比如 RTC_Init() 调用，这些配置好的参数将会设置外设相应的寄存器，达到配置外设工作环境的目的。

初始化结构体和初始化库函数配合使用是 HAL 库精髓所在，理解了初始化结构体每个成员意义基本上就可以对该外设运用自如。初始化结构体定义在 stm32f1xx_hal_rtc.h 头文件中，初始化库函数定义在 stm32f1xx_hal_rtc.c 文件中，编程时我们可以结合这两个文件内注释使用。

RTC 初始化结构体用来设置 RTC 小时的格式和 RTC_CLK 的分频系数。

列表 1: RTC 初始化结构体

```
typedef struct {
    uint32_t AsynchPrediv;     /* 配置 RTC_CLK 的异步分频因子 (0x00~0x7F ) */

    uint32_t OutPut;           /* 指定哪一路信号作为 RTC 的输出 */
} RTC_InitTypeDef;
```

1) AsynchPrediv：RTC_CLK 异步分频因子设置，7 位有效，具体由 RTC 预分频器寄存器 RTC_PRER 的 PREDIV_A[6:0] 位配置。

2) OutPut：RTCEx 输出通道设置，可以是 RTC_OUTPUT_DISABLE 禁止输出、RTC_OUTPUT_ALARMA 闹钟 A 输出、RTC_OUTPUT_ALARMB 闹钟 B 输出、RTC_OUTPUT_WAKEUP 唤醒输出。

38.4.2 RTC 时间结构体讲解

RTC 时间初始化结构体用来设置初始时间，配置的是 RTC 时间寄存器 RTC_TR。

列表 2: RTC 时间结构体

```
typedef struct {
    uint8_t Hours;     /* 小时设置 */
    uint8_t Minutes;   /* 分钟设置 */
    uint8_t Seconds;   /* 秒设置 */
} RTC_TimeTypeDef;
```

1) Hours：小时设置，12 小时制式时，取值范围为 0~11，24 小时制式时，取值范围为 0~23。

2) Minutes：分钟设置，取值范围为 0~59。

3) Seconds：秒钟设置，取值范围为 0~59。

列表 3: 代码清单 39-1 进入和退出 RTC 配置模式

```
/**
* @brief  进入 RTC 配置模式 .
* @param  None
* @retval None
*/
void RTC_EnterConfigMode(void)
{
    /* 设置 CNF 位进入配置模式 */
    RTC->CRL |= RTC_CRL_CNF;
}

/*
* @brief  退出 RTC 配置模式 .
* @param  None
* @retval None
*/
void RTC_ExitConfigMode(void)
{
```

(下页继续)

(续上页)

```
    /* 清空  CNF 位退出配置模式 */
    RTC->CRL &= (uint16_t)~((uint16_t)RTC_CRL_CNF);
}
```

这两个库函数分别提供了进入和退出 RTC 寄存器的配置模式，一般情况下它们由库函数调用。

38.4.3 RTC 日期结构体讲解

RTC 日期初始化结构体用来设置初始日期，配置的是 RTC 日期寄存器 RTC_DR。

列表 4: RTC 日期结构体

```
typedef struct {
    uint8_t WeekDay; /* 星期几设置 */

    uint8_t Month;   /* 月份设置 */

    uint8_t Date;    /* 日期设置 */

    uint8_t Year;    /* 年份设置 */
} RTC_DateTypeDef;
```

1) WeekDay：星期几设置，取值范围为 1~7，对应星期一 ~ 星期日。

2) Month：月份设置，取值范围为 1~12。

3) Date：日期设置，取值范围为 1~31。

4) Year：年份设置，取值范围为 0~99。

38.4.4 RTC 闹钟结构体

RTC 闹钟结构体主要用来设置闹钟时间，设置的格式为 [星期/日期]-[时]-[分]-[秒]，共四个字段，每个字段都可以设置为有效或者无效，即可 MASK。如果 MASK 掉 [星期/日期] 字段，则每天闹钟都会响。

列表 5: RTC 闹钟结构体

```
typedef struct {
    RTC_TimeTypeDef AlarmTime;      /* 设定 RTC 时间寄存器的值：时/分/秒 */
    uint32_t Alarm;                 /* RTC 闹钟选择 */
} RTC_AlarmTypeDef;
```

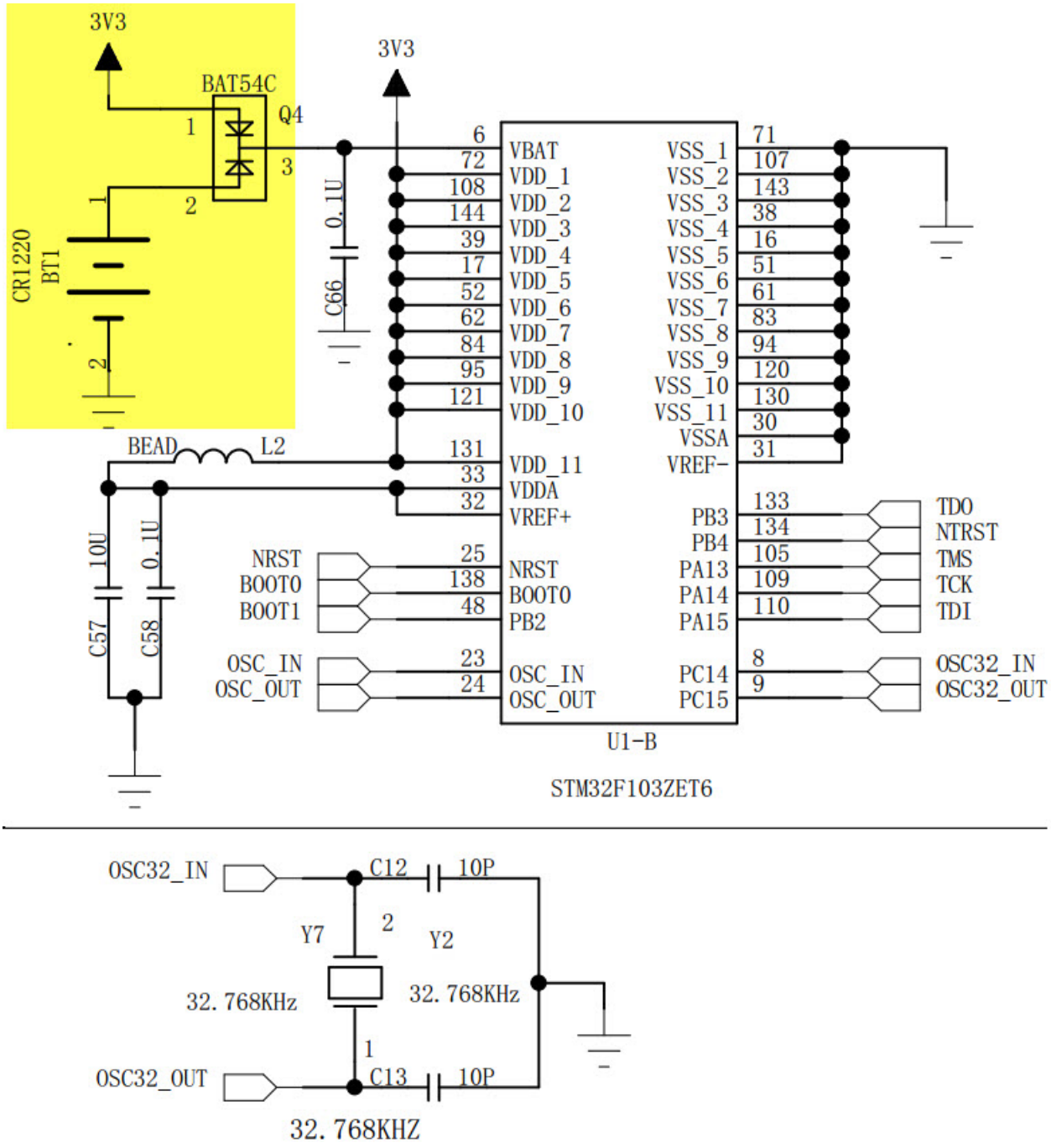
1. Alarm：RTC 闹钟选择，即选择闹钟 A 或者闹钟 B。

38.5 利用 RTC 提供北京时间

从上面的分析可知，RTC 外设是个连续计数的计数器，利用它提供的时间戳，可通过程序转换输出实时时钟和日历的功能，修改计数器的值则可以重新设置系统当前的时间和日期。由于它的时钟配置系统 (*RCC_BDCR* 寄存器) 是在备份域，在系统复位或从待机模式唤醒后 RTC 的设置维持不变，而且使用备份域电源可以全 RTC 计时器在主电源关掉的情况下仍然运行，保证时间的正确。

38.5.1 硬件设计

开发板中与 RTC 相关的硬件设计见图 *39_3*。



原理图中的右上角是备份域的供电电路，在本开发板中提供了一个钮扣电池插槽，可以接入型号为 CR1220 的钮扣电池，该型号的钮扣电池电压为 3.2V，图中的 BAT54C 双向二极管可切换输入到 STM32 备份域电源引脚 VBAT 的供电，当主电源正常供电时，由稳压器输出的 3.3V 供电，当

主电源掉电时，由钮扣电池供电。

原理图下方的是本开发板采用的 LSE 晶振电路，此处使用的晶振频率为 32.768KHz，RTC 外设可以使用 LSE 作为时钟，把它进行分频得到 1Hz 的 RTC 计时时钟。

注意：本实验默认使用 LSI 内部时钟，使用内部时钟时，即使安装了钮扣电池，主电源掉电后时间是不会继续走的，只会保留上次断电的时间。若要持续运行，需要修改 bsp_rtc.h 文件，使用 RTC_CLOCK_SOURCE_LSE 宏，切换成使用 LSE 外部时钟。

38.5.2 软件设计

本小节讲解的是“RTC—实时时钟/RTC—万年历实验”实验，请打开配套的代码工程阅读理解，本工程与 RTC 底层驱动相关的文件为 bsp_rtc.c/h，在底层驱动之上我们添加了从开源代码中获取的 bsp_calendar.c/h 和 bsp_date.c/h 文件，用于万年历的计算。

38.5.2.1 程序设计要点

(1) 初始化 RTC 外设；

(2) 设置时间以及添加配置标志；

(3) 获取当前时间；

38.5.2.2 代码分析

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，完整的代码请参考本章配套的工程。我们创建了两个文件：bsp_rtc.c 和 bsp_rtc.h 文件用来存 RTC 驱动程序及相关宏定义，中断服务函数则放在 stm32f4xx_it.h 文件中。

38.5.2.2.1 宏定义

列表 6: 代码清单 39-2 宏定义

```
//是否使用 LCD 显示
//#define USE_LCD_DISPLAY

// 时钟源宏定义
//#define RTC_CLOCK_SOURCE_LSE
#define RTC_CLOCK_SOURCE_LSI

// 异步分频因子
#define ASYNCHPREDIV          0X7F
// 同步分频因子
#define SYNCHPREDIV           0XFF

// 时间宏定义
#define RTC_H12_AMorPM        RTC_H12_AM
#define HOURS                 1           // 0~23
#define MINUTES               1           // 0~59
#define SECONDS               1           // 0~59

// 日期宏定义
#define WEEKDAY               1          // 1~7
#define DATE                  1          // 1~31
#define MONTH                 1          // 1~12
#define YEAR                  1          // 0~99

// 时间格式宏定义
#define RTC_Format_BINorBCD   RTC_Format_BIN

// 备份域寄存器宏定义
#define RTC_BKP_DRX           RTC_BKP_DR0
```

(下页继续)

(续上页)

```
// 写入到备份寄存器的数据宏定义
#define RTC_BKP_DATA          0X32F2
```

为了方便程序移植，我们把移植时需要修改的代码部分都通过宏定义来实现。具体的配合注释看代码即可。

38.5.2.2.2 RTC 时钟配置函数

列表 7: 代码清单 39-3 RTC 时钟配置函数

```
void RTC_CLK_Config(void)
{
    RCC_OscInitTypeDef        RCC_OscInitStruct;
    RCC_PeriphCLKInitTypeDef  PeriphClkInitStruct;

    // 配置 RTC 外设
    Rtc_Handle.Instance = RTC;

    /* 使能 PWR 时钟 */
    __HAL_RCC_PWR_CLK_ENABLE();
    /* PWR_CR:
    DBF 置 1，使能 RTC、RTC 备份寄存器和备份 SRAM 的访问 */
    HAL_PWR_EnableBkUpAccess();

#if defined (RTC_CLOCK_SOURCE_LSI)
    /* 使用 LSI 作为 RTC 时钟源会有误差
    * 默认选择 LSE 作为 RTC 的时钟源
    */
    /* 初始化 LSI */
    RCC_OscInitStruct.OscillatorType =  RCC_OSCILLATORTYPE_LSI |
                                        RCC_OSCILLATORTYPE_LSE;
```

(下页继续)

(续上页)

```
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    RCC_OscInitStruct.LSIState = RCC_LSI_ON;
    RCC_OscInitStruct.LSEState = RCC_LSE_OFF;
    HAL_RCC_OscConfig(&RCC_OscInitStruct);
    /* 选择 LSI 做为 RTC 的时钟源 */
    PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_RTC;
    PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSI;
    HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);

#elif defined (RTC_CLOCK_SOURCE_LSE)
    /* 初始化 LSE */
    RCC_OscInitStruct.OscillatorType =  RCC_OSCILLATORTYPE_LSI |
                                        RCC_OSCILLATORTYPE_LSE;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    RCC_OscInitStruct.LSEState = RCC_LSE_ON;
    RCC_OscInitStruct.LSIState = RCC_LSI_OFF;
    HAL_RCC_OscConfig(&RCC_OscInitStruct);
    /* 选择 LSE 做为 RTC 的时钟源 */
    PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_RTC;
    PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;
    HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);

//  /* Configures the External Low Speed oscillator (LSE) drive capability␣
↪*/
//  __HAL_RCC_LSEDRIVE_CONFIG(RCC_LSEDRIVE_HIGH);

#endif /* RTC_CLOCK_SOURCE_LSI */

    /* 使能 RTC 时钟 */
    __HAL_RCC_RTC_ENABLE();

    /* 等待 RTC APB 寄存器同步 */
```

(下页继续)

(续上页)

```
    HAL_RTC_WaitForSynchro(&Rtc_Handle);

    /*========== 初始化同步/异步预分频器的值 ==========*/

    /* 驱动日历的时钟 ck_spare = LSE/[(255+1)*(127+1)] = 1HZ */

    /* 设置异步预分频器的值 */
    Rtc_Handle.Init.AsynchPrediv = ASYNCHPREDIV;
    /* 设置同步预分频器的值 */
    Rtc_Handle.Init.SynchPrediv  = SYNCHPREDIV;
    Rtc_Handle.Init.HourFormat   = RTC_HOURFORMAT_24;
    /* 用 RTC_InitStructure 的内容初始化 RTC 寄存器 */
    if (HAL_RTC_Init(&Rtc_Handle) != HAL_OK) {
        printf("\n\r RTC 时钟初始化失败 \r\n");
    }
}
```

RTC 时钟配置函数主要实现两个功能，一是选择 RTC_CLK 的时钟源，根据宏定义来决定使用 LSE 还是 LSI 作为 RTC_CLK 的时钟源，这里为了方便我们选择 LSI；二是设置 RTC_CLK 的预分频系数，包括异步和同步两个，这里设置异步分频因子为 ASYNCHPREDIV（127），同步分频因子为 ASYNCHPREDIV（255），则产生的最终驱动日历的时钟 CK_SPRE=32.768/(127+1)*(255+1)=1HZ，则每秒更新一次。

38.5.2.2.3 RTC 时间初始化函数

列表 8: RTC 时间和日期设置函数

```
/**
* @brief  设置时间和日期
* @param  无
* @retval 无
```

(下页继续)

(续上页)

```
*/
void RTC_TimeAndDate_Set(void)
{
    RTC_DateTypeDef  RTC_DateStructure;
    RTC_TimeTypeDef  RTC_TimeStructure;
    // 初始化时间
    RTC_TimeStructure.TimeFormat = RTC_H12_AMorPM;
    RTC_TimeStructure.Hours = HOURS;
    RTC_TimeStructure.Minutes = MINUTES;
    RTC_TimeStructure.Seconds = SECONDS;
    HAL_RTC_SetTime(&Rtc_Handle,&RTC_TimeStructure, RTC_FORMAT_BIN);
    // 初始化日期
    RTC_DateStructure.WeekDay = WEEKDAY;
    RTC_DateStructure.Date = DATE;
    RTC_DateStructure.Month = MONTH;
    RTC_DateStructure.Year = YEAR;
    HAL_RTC_SetDate(&Rtc_Handle,&RTC_DateStructure, RTC_FORMAT_BIN);

    HAL_RTCEx_BKUPWrite(&Rtc_Handle,RTC_BKP_DRX, RTC_BKP_DATA);
}
```

RTC 时间和日期设置函数主要是设置时间和日期这两个结构体，然后调相应的 HAL_RTC_SetTime 和 HAL_RTC_SetDate 函数把初始化好的时间写到相应的寄存器，每当写完之后都会在备份寄存器里面写入一个数，以作标记，为的是程序开始运行的时候检测 RTC 的时间是否已经配置过。

具体的时间、日期、备份寄存器和写入备份寄存器的值都在头文件的宏定义里面，要修改这些值只需修改头文件的宏定义即可。

38.5.2.2.4 RTC 时间显示函数

列表 9: RTC 时间显示函数

```
/**
* @brief  显示时间和日期
* @param  无
* @retval 无
*/
void RTC_TimeAndDate_Show(void)
{
    uint8_t Rtctmp=0;
    char LCDTemp[100];
    RTC_TimeTypeDef RTC_TimeStructure;
    RTC_DateTypeDef RTC_DateStructure;
    while (1) {
        // 获取日历
        HAL_RTC_GetTime(&Rtc_Handle, &RTC_TimeStructure, RTC_FORMAT_BIN);
        HAL_RTC_GetDate(&Rtc_Handle, &RTC_DateStructure, RTC_FORMAT_BIN);

        // 每秒打印一次
        if (Rtctmp != RTC_TimeStructure.Seconds) {

            // 打印日期
        printf("The Date :  Y:20%0.2d - M:%0.2d - D:%0.2d - W:%0.2d\r\n",
                    RTC_DateStructure.Year,
                    RTC_DateStructure.Month,
                    RTC_DateStructure.Date,
                    RTC_DateStructure.WeekDay);

            //液晶显示日期
            //先把要显示的数据用 sprintf 函数转换为字符串，然后才能用液晶显示函数显示
    sprintf(LCDTemp,"The Date :  Y:20%0.2d - M:%0.2d - D:%0.2d - W:%0.2d",
```

(下页继续)

(续上页)

```
                    RTC_DateStructure.Year,
                    RTC_DateStructure.Month,
                    RTC_DateStructure.Date,
                    RTC_DateStructure.WeekDay);

            LCD_SetColors(LCD_COLOR_RED,LCD_COLOR_BLACK);
            LCD_DisplayStringLine_EN_CH(8,(uint8_t *)LCDTemp);

            // 打印时间
            printf("The Time :  %0.2d:%0.2d:%0.2d \r\n\r\n",
                    RTC_TimeStructure.Hours,
                    RTC_TimeStructure.Minutes,
                    RTC_TimeStructure.Seconds);

            //液晶显示时间
            sprintf(LCDTemp,"The Time :  %0.2d:%0.2d:%0.2d",
                    RTC_TimeStructure.Hours,
                    RTC_TimeStructure.Minutes,
                    RTC_TimeStructure.Seconds);

            LCD_DisplayStringLine_EN_CH(10,(uint8_t *)LCDTemp);

            (void)RTC->DR;
        }
        Rtctmp = RTC_TimeStructure.Seconds;
    }
}
```

RTC 时间和日期显示函数中，通过调用 HAL_RTC_GetTime() 和 HAL_RTC_GetDate() 这两个库函数，把时间和日期都读取保存到时间和日期结构体中，然后以 1s 为频率，把时间显示出来。

在使用液晶显示时间的时候，需要先调用标准的 C 库函数 sprintf() 把数据转换成字符串，然后才能调用液晶显示函数来显示，因为液晶显示时处理的都是字符串。

38.5.2.2.5 主函数

列表 10: 代码清单 39-4 main 函数

```
int main(void)
{
    /* 系统时钟初始化成 72 MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 串口初始化 */
    DEBUG_USART_Config();
    /* 蜂鸣器端口初始化 */
    BEEP_GPIO_Config();
    printf("\n\r这是一个 RTC 闹钟实验 \r\n");

#ifdef USE_LCD_DISPLAY
    /*================ 液晶初始化开始 ================*/
    ILI9806G_Init ();          //LCD 初始化

    //其中 0、3、5、6 模式适合从左至右显示文字,
    //不推荐使用其它模式显示文字其它模式显示文字会有镜像效果
    //其中 6 模式为大部分液晶例程的默认显示方向
    ILI9806G_GramScan ( 6 );
    /*================ 液晶初始化结束 ================*/
#endif
    /*
    *
    * 当我们配置过 RTC 时间之后就往备份寄存器 0 写入数据做标记
    *
    * 所以每次程序重新运行的时候就通过检测备份寄存器 0 的值来判断 RTC
    * 是否已经配置过，如果配置过那就继续运行，如果没有配置过
    * 就初始化 RTC，配置 RTC 的时间。
```

(下页继续)

(续上页)

```
    */

    /* RTC 配置：选择时钟源，设置 RTC_CLK 的分频系数 */
    RTC_CLK_Config();

    if (HAL_RTCEx_BKUPRead(&Rtc_Handle,RTC_BKP_DRX) != 0X32F3) {
        /* 设置时间和日期 */
        RTC_TimeAndDate_Set();
    } else {
        /* 检查是否电源复位 */
        if (__HAL_RCC_GET_FLAG(RCC_FLAG_PORRST) != RESET) {
            printf("\r\n 发生电源复位....\r\n");
        }
        /* 检查是否外部复位 */
        else if (__HAL_RCC_GET_FLAG(RCC_FLAG_PINRST) != RESET) {
            printf("\r\n 发生外部复位....\r\n");
        }
        printf("\r\n 不需要重新配置 RTC....\r\n");
        /* 使能 PWR 时钟 */
        __HAL_RCC_PWR_CLK_ENABLE();
        /* PWR_CR:DBF 置 1，使能 RTC、RTC 备份寄存器和备份 SRAM 的访问 */
        HAL_PWR_EnableBkUpAccess();
        /* 等待 RTC APB 寄存器同步 */
        HAL_RTC_WaitForSynchro(&Rtc_Handle);
    }
    /* 显示时间和日期 */
    RTC_TimeAndDate_Show();
}
```

主函数中，我们调用 HAL_RTCEx_BKUPRead () 库函数来读取备份寄存器的值是否等于我们预设的那个值，因为当我们初始化完 RTC 的时间之后就往备份寄存器写入一个数据做标记，所以每次程序重新运行的时候就通过检测备份寄存器的值来判断，RTC 是否已经配置过，如果配置过则

判断是电源复位还是外部引脚复位且继续运行显示时间，如果没有配置过，就初始化 RTC，配置 RTC 的时间，然后显示。

如果想每次程序运行时都重新配置 RTC，则用一个异于写入的值来做判断即可。

38.5.3 下载验证

把程序编译好下载到开发板，通过电脑端口的串口调试助手或者液晶可以看到时间正常运行。当 VDD 不断电的情况下，发生外部引脚复位，时间不会丢失。当 VDD 断电或者发生外部引脚复位，VBT 有电池供电时，时间不会丢失。当 VDD 断电且 VBAT 也不供电的情况下，时间会丢失，然后根据程序预设的初始时间重新启动。

第 39 章　MDK 的编译过程及文件类型全解

本章参考资料：MDK 的帮助手册《ARM Development Tools》，点击 MDK 界面的“help->uVision Help”菜单可打开该文件。关于 ELF 文件格式，参考配套资料里的《ELF 文件格式》文件。

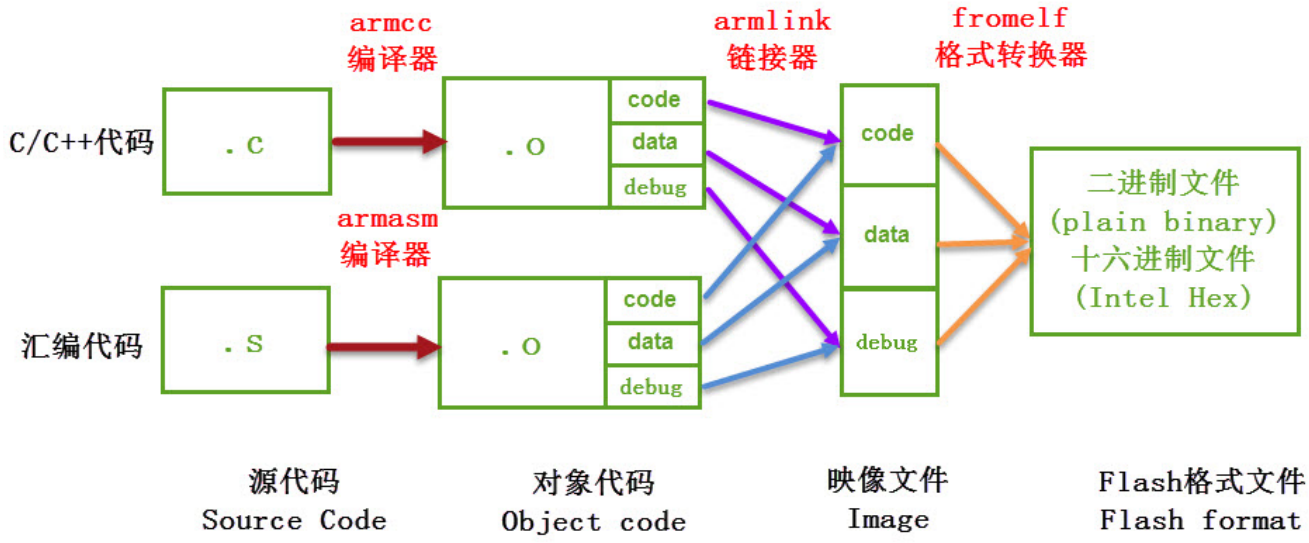
在本章中讲解了非常多的文件类型，学习时请跟着教程的节奏，打开实际工程中的文件来了解。

相信您已经非常熟练地使用 MDK 创建应用程序了，平时使用 MDK 编写源代码，然后编译生成机器码，再把机器码下载到 STM32 芯片上运行，但是这个编译、下载的过程 MDK 究竟做了什么工作？它编译后生成的各种文件又有什么作用？本章节将对这些过程进行讲解，了解编译及下载过程有助于理解芯片的工作原理，这些知识对制作 IAP(bootloader) 以及读写控制器内部 FLASH 的应用时非常重要。

39.1 编译过程

39.1.1 编译过程简介

首先我们简单了解下 MDK 的编译过程，它与其它编译器的工作过程是类似的，该过程见图 *40_1*。



编译过程生成的不同文件将在后面的小节详细说明，此处先抓住主要流程来理解。

(1) 编译，MDK 软件使用的编译器是 armcc 和 armasm，它们根据每个 c/c++ 和汇编源文件编译成对应的以“.o”为后缀名的对象文件 (Object Code，也称目标文件)，其内容主要是从源文件编译得到的机器码，包含了代码、数据以及调试使用的信息；

(2) 链接，链接器 armlink 把各个.o 文件及库文件链接成一个映像文件“.axf”或“.elf”；

(3) 格式转换，一般来说 Windows 或 Linux 系统使用链接器直接生成可执行映像文件 elf 后，内核根据该文件的信息加载后，就可以运行程序了，但在单片机平台上，需要把该文件的内容加载到芯片上，所以还需要对链接器生成的 elf 映像文件利用格式转换器 fromelf 转换成“.bin”或“.hex”文件，交给下载器下载到芯片的 FLASH 或 ROM 中。

39.1.2 具体工程中的编译过程

下面我们打开“多彩流水灯”的工程，以它为例进行讲解，其它工程的编译过程也是一样的，只是文件有差异。打开工程后，点击 MDK 的“rebuild”按钮，它会重新构建整个工程，构建的过程会在 MDK 下方的“Build Output”窗口输出提示信息，见图 *40_2*。

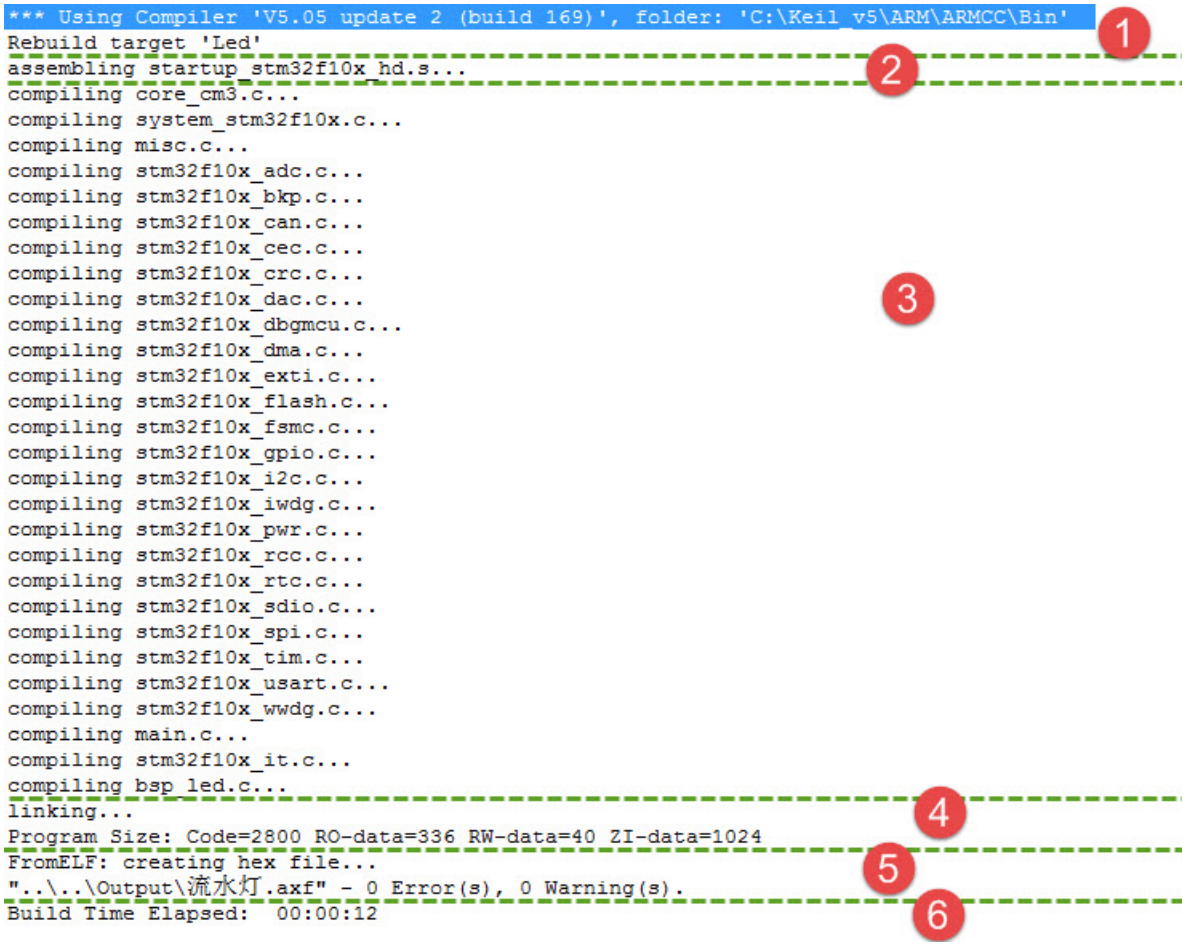


图 40-2 编译工程时的编译提示

构建工程的提示输出主要分 6 个部分，说明如下：

(1) 提示信息的第一部分说明构建过程调用的编译器。图中的编译器名字是“V5.06(build 20)”，后面附带了该编译器所在的文件夹。在电脑上打开该路径，可看到该编译器包含图 *40_3* 中的各个编译工具，如 armar、armasm、armcc、armlink 及 fromelf，后面四个工具已在图 *40_1* 中已讲解，而 armar 是用于把.o 文件打包成 lib 文件的。

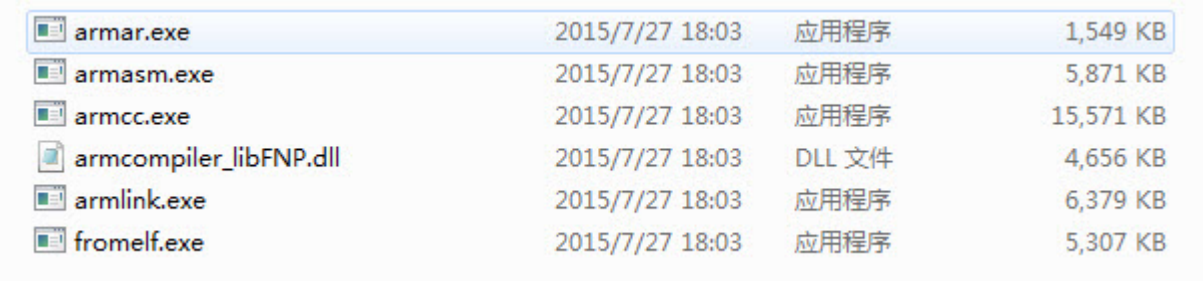


图 40-3 编译工具

(2) 使用 armasm 编译汇编文件。图中列出了编译 startup 启动文件时的提示，编译后每个汇编源文件都对应有一个独立的.o 文件。

(3) 使用 armcc 编译 c/c++ 文件。图中列出了工程中所有的 c/c++ 文件的提示，同样地，编译后每个 c/c++ 源文件都对应有一个独立的.o 文件。

(4) 使用 armlink 链接对象文件，根据程序的调用把各个.o 文件的内容链接起来，最后生成程序的 axf 映像文件，并附带程序各个域大小的说明，包括 Code、RO-data、RW-data 及 ZI-data 的大小。

(5) 使用 fromelf 生成下载格式文件，它根据 axf 映像文件转化成 hex 文件，并列出编译过程出现的错误 (Error) 和警告 (Warning) 数量。

(6) 最后一段提示给出了整个构建过程消耗的时间。

构建完成后，可在工程的“Output”及“Listing”目录下找到由以上过程生成的各种文件，见图 *40_4*。

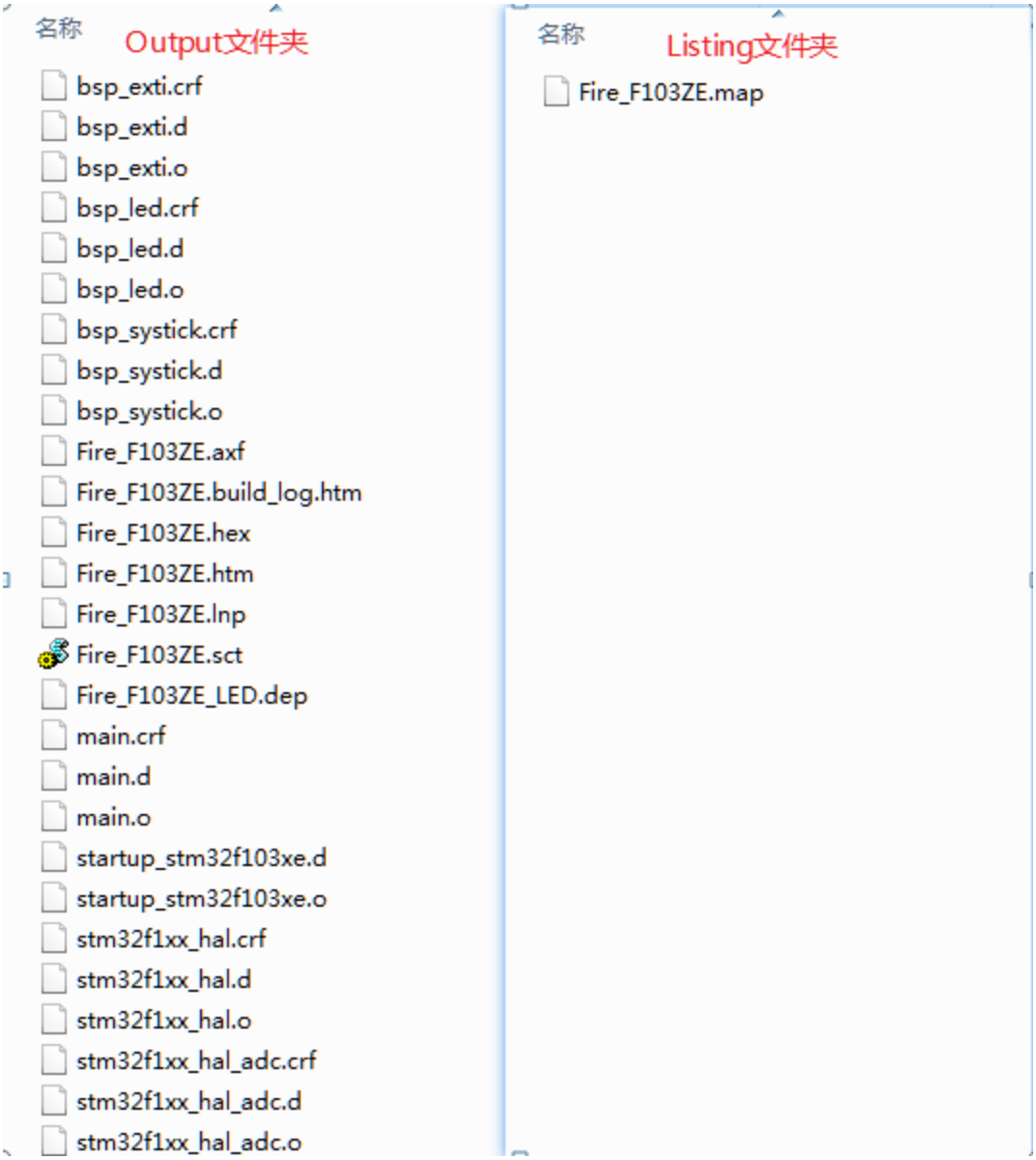


图 40-4 编译后 Output 及 Listing 文件夹中的内容

可以看到，每个 C 源文件都对应生成了.o、.d 及.crf 后缀的文件，还有一些额外的.dep、.hex、.axf、.htm、.lnp、.sct、.lst 及.map 文件。

39.2 程序的组成、存储与运行

39.2.1 CODE、RO、RW、ZI Data 域及堆栈空间

在工程的编译提示输出信息中有一个语句“Program Size：Code=xx RO-data=xx RW-data=xx ZI-data=xx”，它说明了程序各个域的大小，编译后，应用程序中所有具有同一性质的数据 (包括代码) 被归到一个域，程序在存储或运行的时候，不同的域会呈现不同的状态，这些域的意义如下：

- Code：即代码域，它指的是编译器生成的机器指令，这些内容被存储到 ROM 区。
- RO-data：Read Only data，即只读数据域，它指程序中用到的只读数据，这些数据被存储在 ROM 区，因而程序不能修改其内容。例如 C 语言中 const 关键字定义的变量就是典型的 RO-data。
- RW-data：Read Write data，即可读写数据域，它指初始化为“非 0 值”的可读写数据，程序刚运行时，这些数据具有非 0 的初始值，且运行的时候它们会常驻在 RAM 区，因而应用程序可以修改其内容。例如 C 语言中使用定义的全局变量，且定义时赋予“非 0 值”给该变量进行初始化。
- ZI-data：Zero Initialie data，即 0 初始化数据，它指初始化为“0 值”的可读写数据域，它与 RW-data 的区别是程序刚运行时这些数据初始值全都为 0，而后续运行过程与 RW-data 的性质一样，它们也常驻在 RAM 区，因而应用程序可以更改其内容。例如 C 语言中使用定义的全局变量，且定义时赋予“0 值”给该变量进行初始化 (若定义该变量时没有赋予初始值，编译器会把它当 ZI-data 来对待，初始化为 0)；
- ZI-data 的栈空间 (Stack) 及堆空间 (Heap)：在 C 语言中，函数内部定义的局部变量属于栈空间，进入函数的时候从向栈空间申请内存给局部变量，退出时释放局部变量，归还内存空间。而使用 malloc 动态分配的变量属于堆空间。在程序中的栈空间和堆空间都是属于 ZI-data 区域的，这些空间都会被初始值化为 0 值。编译器给出的 ZI-data 占用的空间值中包含了堆栈的大小 (经实际测试，若程序中完全没有使用 malloc 动态申请堆空间，编译器会优化，不把堆空间计算在内)。

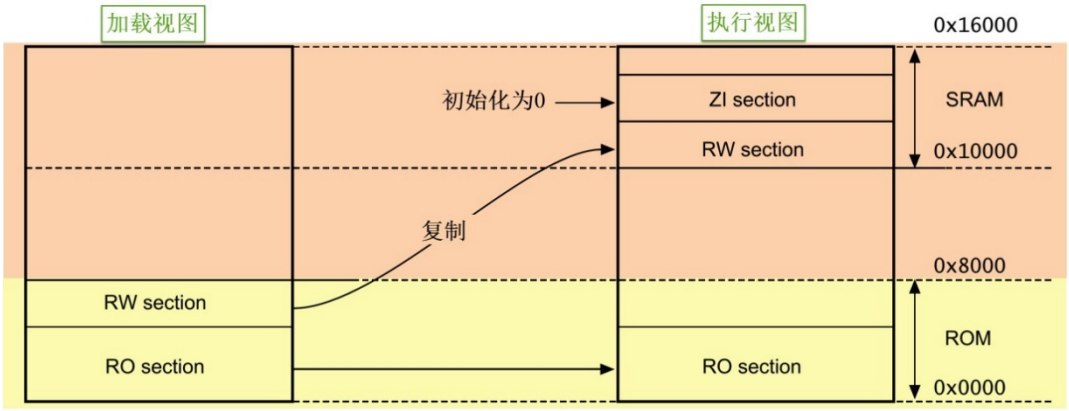
  综上所述，以程序的组成构件为例，它们所属的区域类别见表 40-1。

  表 40-1 程序组件所属的区域

程序组件	所属类别
机器代码指令	Code
常量	RO-data
初值非 0 的全局变量	RW-data
初值为 0 的全局变量	ZI-data
局部变量	ZI-data 栈空间
使用 malloc 动态分配的空间	ZI-data 堆空间

39.2.2 程序的存储与运行

RW-data 和 ZI-data 它们仅仅是初始值不一样而已，为什么编译器非要把它们区分开？这就涉及到程序的存储状态了，应用程序具有静止状态和运行状态。静止态的程序被存储在非易失存储器中，如 STM32 的内部 FLASH，因而系统掉电后也能正常保存。但是当程序在运行状态的时候，程序常常需要修改一些暂存数据，由于运行速度的要求，这些数据往往存放在内存中 (RAM)，掉电后这些数据会丢失。因此，程序在静止与运行的时候它在存储器中的表现是不一样的，见图 *40_5*。



图中的左侧是应用程序的存储状态，右侧是运行状态，而上方是 RAM 存储器区域，下方是 ROM 存储器区域。

程序在存储状态时，RO 节 (RO section) 及 RW 节都被保存在 ROM 区。当程序开始运行时，内核直接从 ROM 中读取代码，并且在执行主体代码前，会先执行一段加载代码，它把 RW 节数据从

ROM 复制到 RAM，并且在 RAM 加入 ZI 节，ZI 节的数据都被初始化为 0。加载完后 RAM 区准备完毕，正式开始执行主体程序。

编译生成的 RW-data 的数据属于图中的 RW 节，ZI-data 的数据属于图中的 ZI 节。是否需要掉电保存，这就是把 RW-data 与 ZI-data 区别开来的原因，因为在 RAM 创建数据的时候，默认值为 0，但如果有的数据要求初值非 0，那就需要使用 ROM 记录该初始值，运行时再复制到 RAM。

STM32 的 RO 区域不需要加载到 SRAM，内核直接从 FLASH 读取指令运行。计算机系统的应用程序运行过程很类似，不过计算机系统的程序在存储状态时位于硬盘，执行的时候甚至会把上述的 RO 区域 (代码、只读数据) 加载到内存，加快运行速度，还有虚拟内存管理单元 (MMU) 辅助加载数据，使得可以运行比物理内存还大的应用程序。而 STM32 没有 MMU，所以无法支持 Linux 和 Windows 系统。

当程序存储到 STM32 芯片的内部 FLASH 时 (即 ROM 区)，它占用的空间是 Code、RO-data 及 RW-data 的总和，所以如果这些内容比 STM32 芯片的 FLASH 空间大，程序就无法被正常保存了。当程序在执行的时候，需要占用内部 SRAM 空间 (即 RAM 区)，占用的空间包括 RW-data 和 ZI-data。应用程序在各个状态时各区域的组成见表 40-2。

表 40-2 程序状态区域的组成

程序状态与区域	组成
程序执行时的只读区域 (RO)	Code + RO data
程序执行时的可读写区域 (RW)	RW data + ZI data
程序存储时占用的 ROM 区	Code + RO data + RW data

在 MDK 中，我们建立的工程一般会选择芯片型号，选择后就有确定的 FLASH 及 SRAM 大小，若代码超出了芯片的存储器的极限，编译器会提示错误，这时就需要裁剪程序了，裁剪时可针对超出的区域来优化。

39.3 编译工具链

在前面编译过程中，MDK 调用了各种编译工具，平时我们直接配置 MDK，不需要学习如何使用它们，但了解它们是非常有好处的。例如，若希望使用 MDK 编译生成 bin 文件的，需要在 MDK 中输入指令控制 fromelf 工具；在本章后面讲解 AXF 及 O 文件的时候，需要利用 fromelf 工具查看其文件信息，这都是无法直接通过 MDK 做到的。关于这些工具链的说明，在 MDK 的帮助手册《ARM Development Tools》都有详细讲解，点击 MDK 界面的“help->uVision Help”菜单可打开该文件。

39.3.1 设置环境变量

调用这些编译工具，需要用到 Windows 的“命令行提示符工具”，为了让命令行方便地找到这些工具，我们先把工具链的目录添加到系统的环境变量中。查看本机工具链所在的具体目录可根据上一小节讲解的工程编译提示输出信息中找到，如本机的路径为“D:workkeil5ARMARMCCbin”。

39.3.1.1 添加路径到 PATH 环境变量

本文以 Win7 系统为例添加工具链的路径到 PATH 环境变量，其它系统是类似的。

(1) 右键电脑系统的“计算机图标”，在弹出的菜单中选择“属性”，见图 *40_6*；

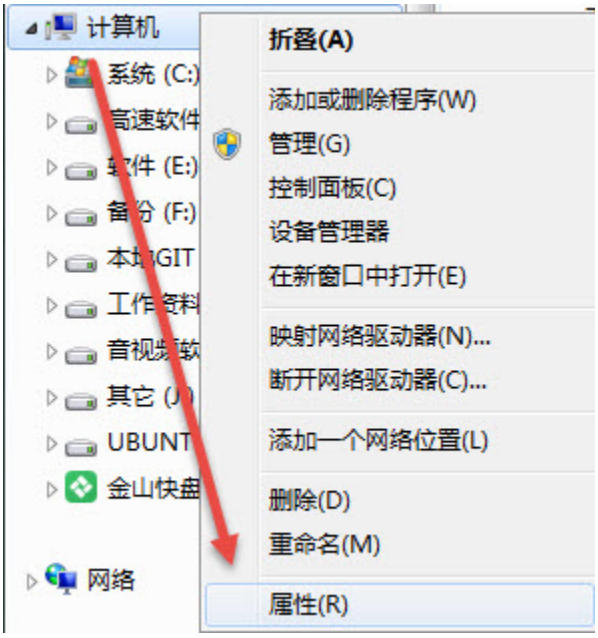


图 40-6 计算机属性页面

(2) 在弹出的属性页面依次点击“高级系统设置”-> “环境变量”，在用户变量一栏中找到名为“PATH”的变量，若没有该变量，则新建一个。编辑“PATH”变量，在它的变量值中输入工具链的路径，如本机的是“;D:workkeil5ARMARMCCbin”，注意要使用“分号;”让它与其它路径分隔开，输入完毕后依次点确定，见图 *40_7*；

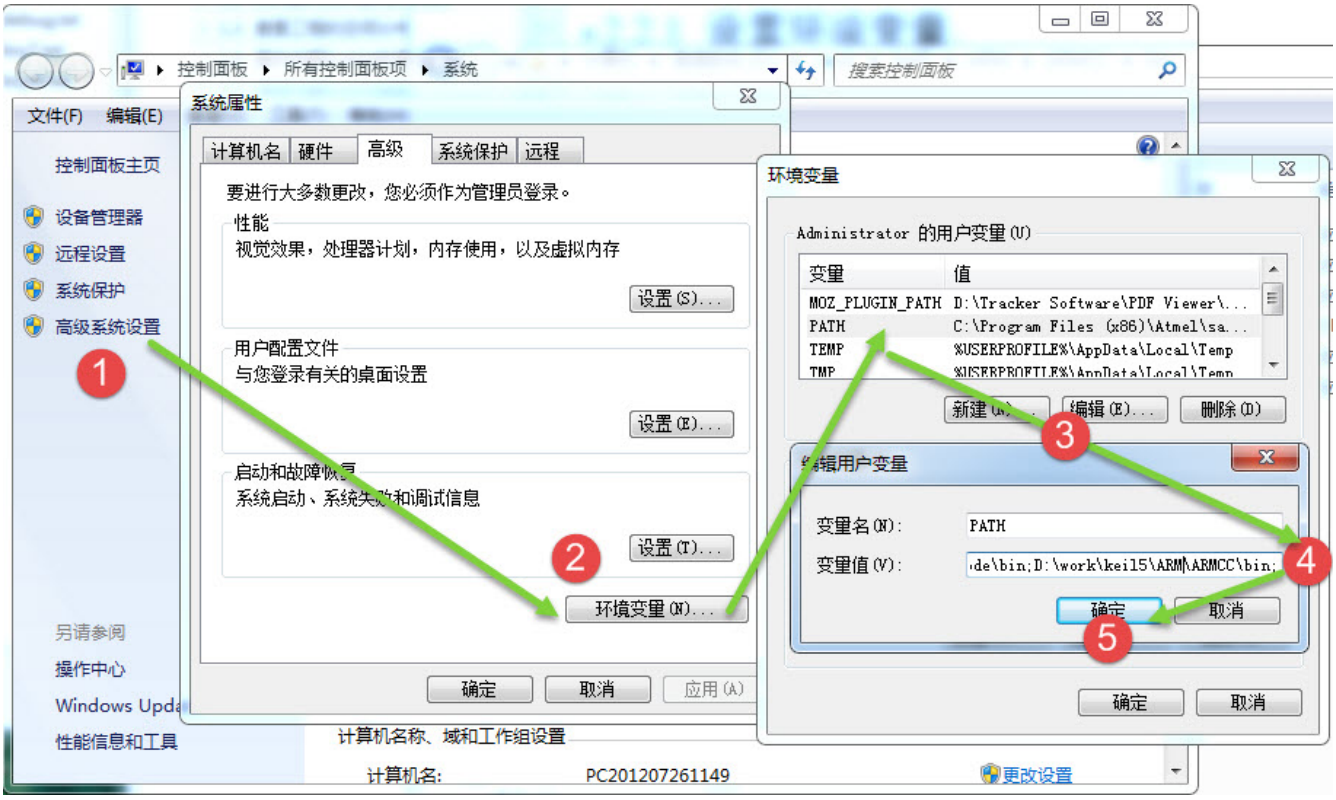


图 40-7 添加工具链路径到 PATH 变量

(3) 打开 Windows 的命令行，点击系统的“开始菜单”，在搜索框输入“cmd”，在搜索结果中点击“cmd.exe”即可打开命令行，见图 *40_8*；

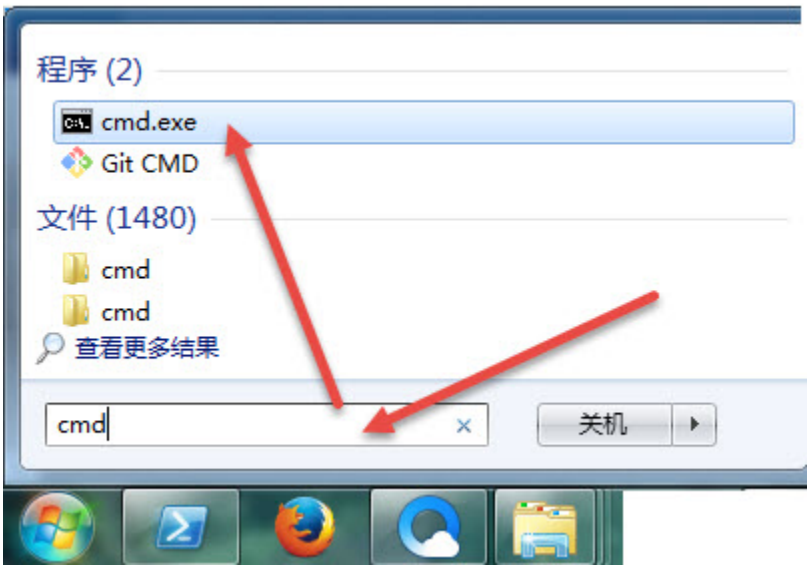


图 40-8 打开命令行

(4) 在弹出的命令行窗口中输入“fromelf”回车，若窗口打印出 formelf 的帮助说明，那么路径正常，就可以开始后面的工作了；若提示“不是内部名外部命令，也不是可运行的程序…”信息，说明路径不对，请重新配置环境变量，并确认该工作目录下有编译工具链。

这个过程本质就是让命令行通过“PATH”路径找到“fromelf.exe”程序运行，默认运行“fromelf.exe”时它会输出自己的帮助信息，这就是工具链的调用过程，MDK 本质上也是如此调用工具链的，只是它集成为 GUI，相对于命令行对用户更友好，毕竟上述配置环境变量的过程已经让新手烦躁了。

39.3.2 armcc、armasm 及 armlink

接下来我们看看各个工具链的具体用法，主要以 armcc 为例。

39.3.2.1 armcc

armcc 用于把 c/c++ 文件编译成 ARM 指令代码，编译后会输出 ELF 格式的 O 文件 (对象、目标文件)，在命令行中输入“armcc”回车可调用该工具，它会打印帮助说明，见图 *40_9*

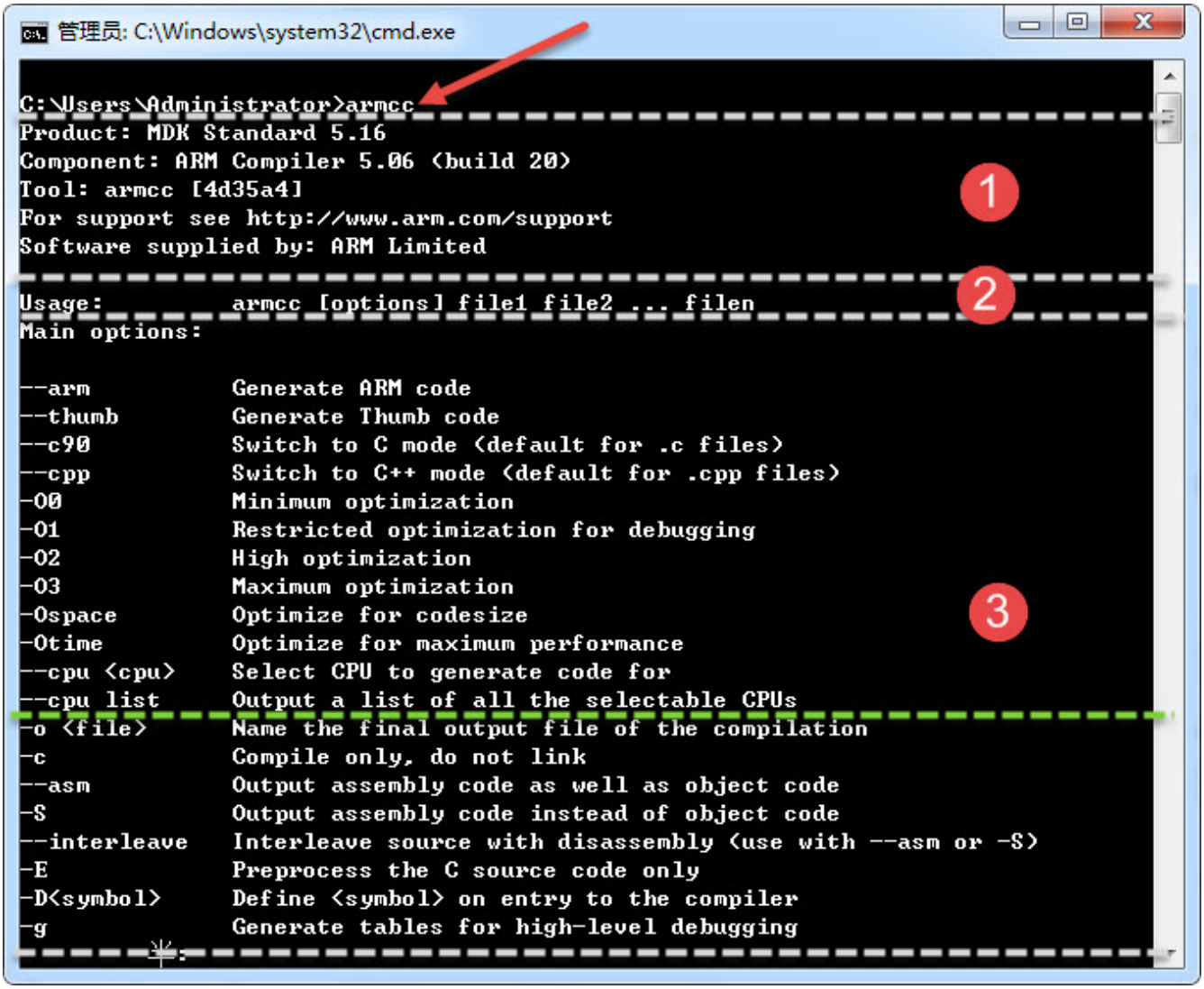


图 40-9 armcc 的帮助提示

帮助提示中分三部分，第一部分是 armcc 版本信息，第二部分是命令的用法，第三部分是主要命令选项。

根据命令用法：armcc [options] file1 file2 …filen ，在 [option] 位置可输入下面的 “–arm”、“–cpu list”

选项，若选项带文件输入，则把文件名填充在 file1 file2…的位置，这些文件一般是 c/c++ 文件。

例如根据它的帮助说明，“–cpu list”可列出编译器支持的所有 cpu，我们在命令行中输入“armcc –cpu list”，可查看图 *40_10* 中的 cpu 列表。

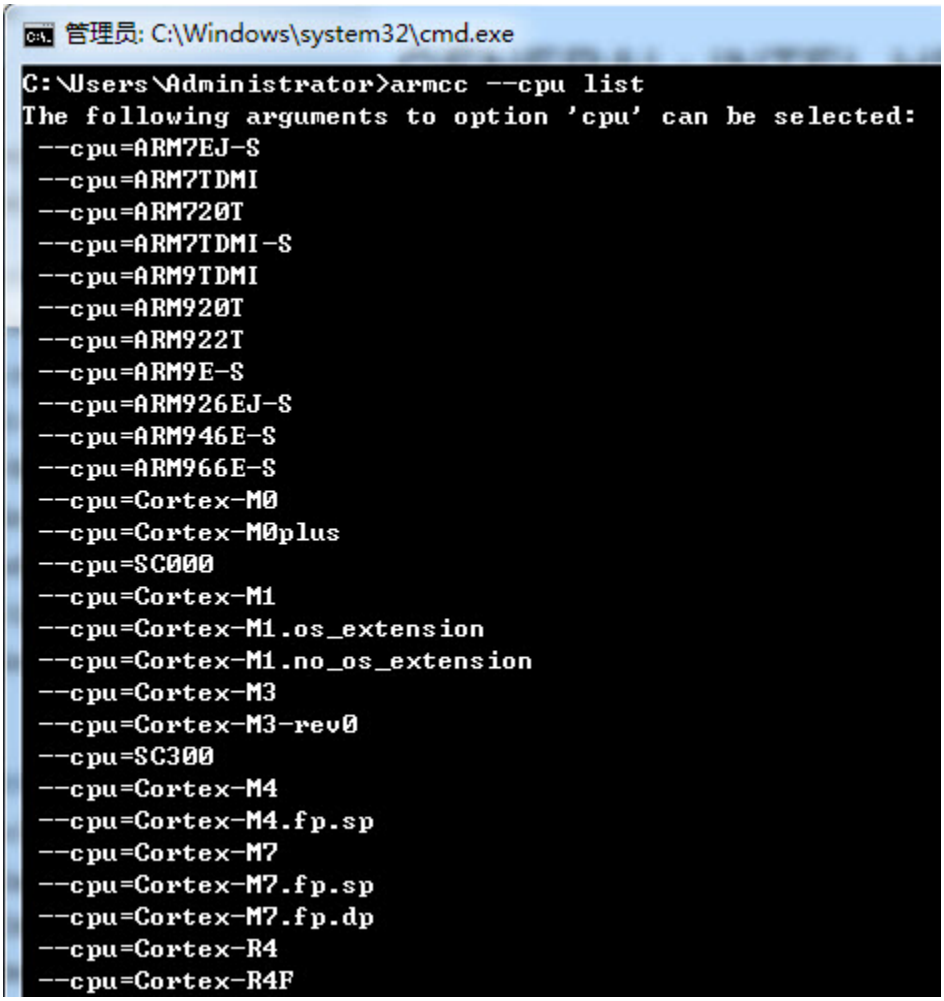


图 40-10 cpulist

打开 MDK 的 Options for Targe->c/c++ 菜单，可看到 MDK 对编译器的控制命令，见图 *40_11*。

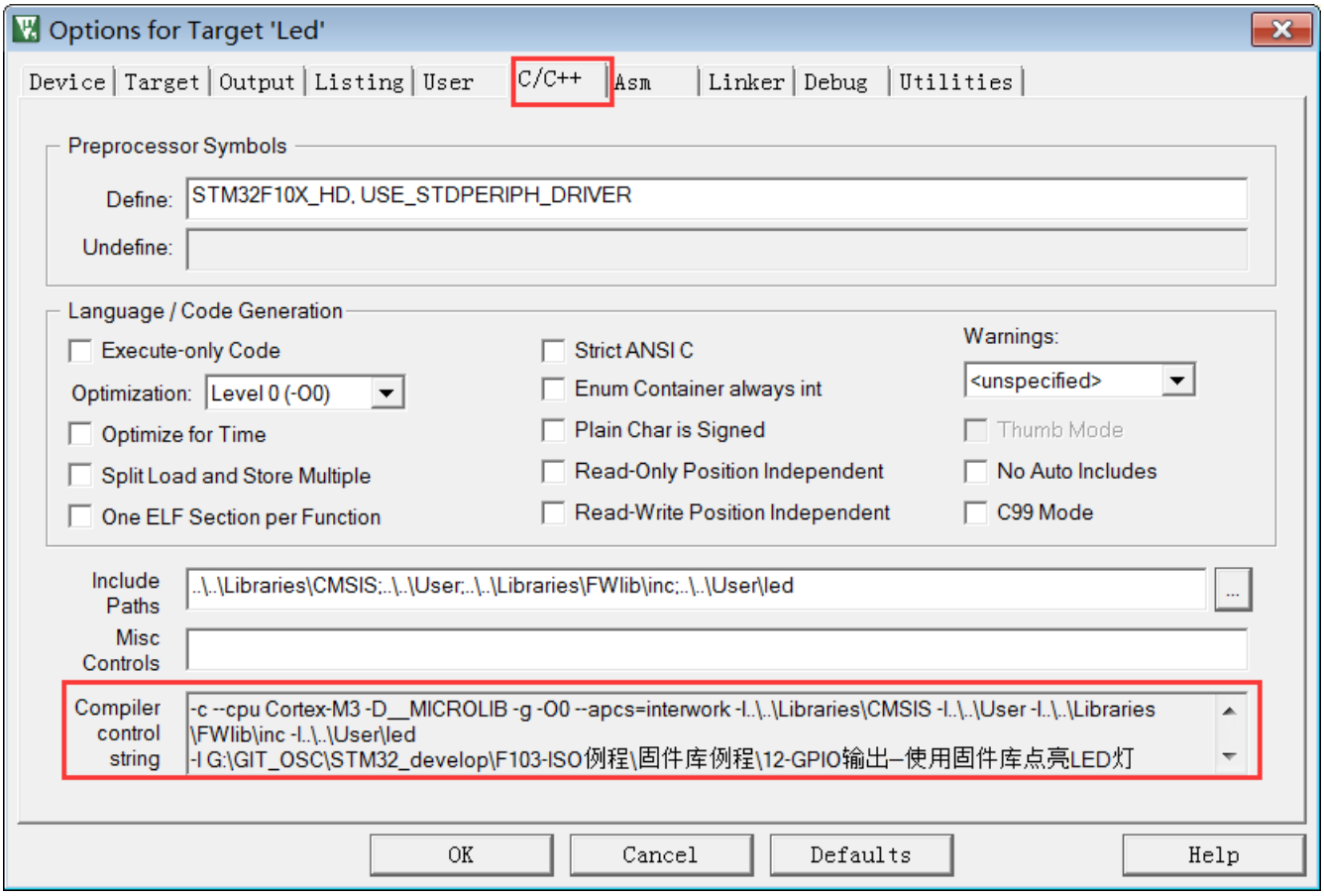


图 40-11 MDK 的 ARMCC 编译选项

从该图中的命令可看到，它调用了-c、-cpu –D –g –O1 等编译选项，当我们修改 MDK 的编译配置时，可看到该控制命令也会有相应的变化。然而我们无法在该编译选项框中输入命令，只能通过 MDK 提供的选项修改。

了解这些，我们就可以查询具体的 MDK 编译选项的具体信息了，如 c/c++ 选项中的“Optimization：Leve 1（-O1）”是什么功能呢？首先可了解到它是“-O”命令，命令后还带个数字，查看 MDK 的帮助手册，在 armcc 编译器说明章节，可详细了解，如图 *40_9*。

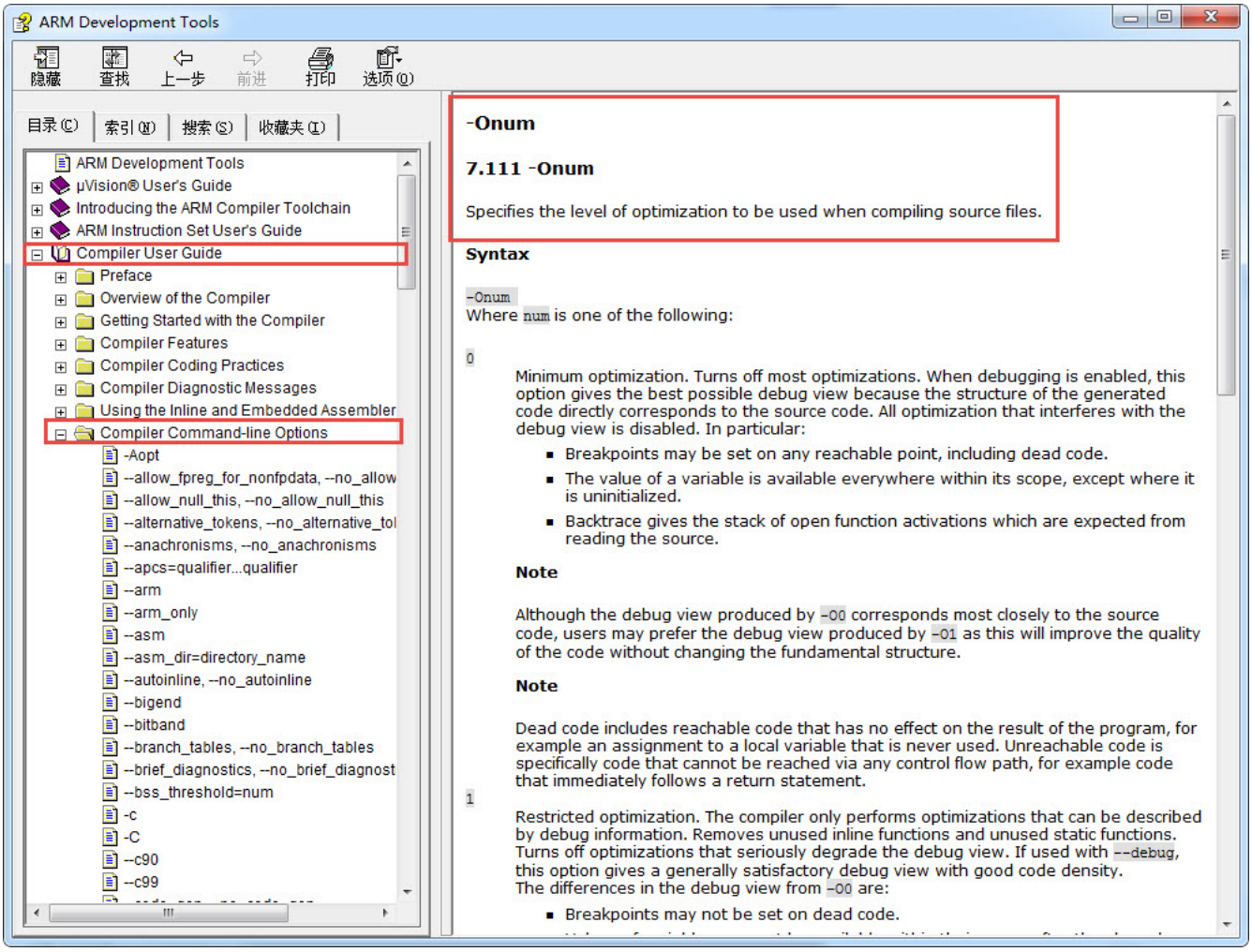
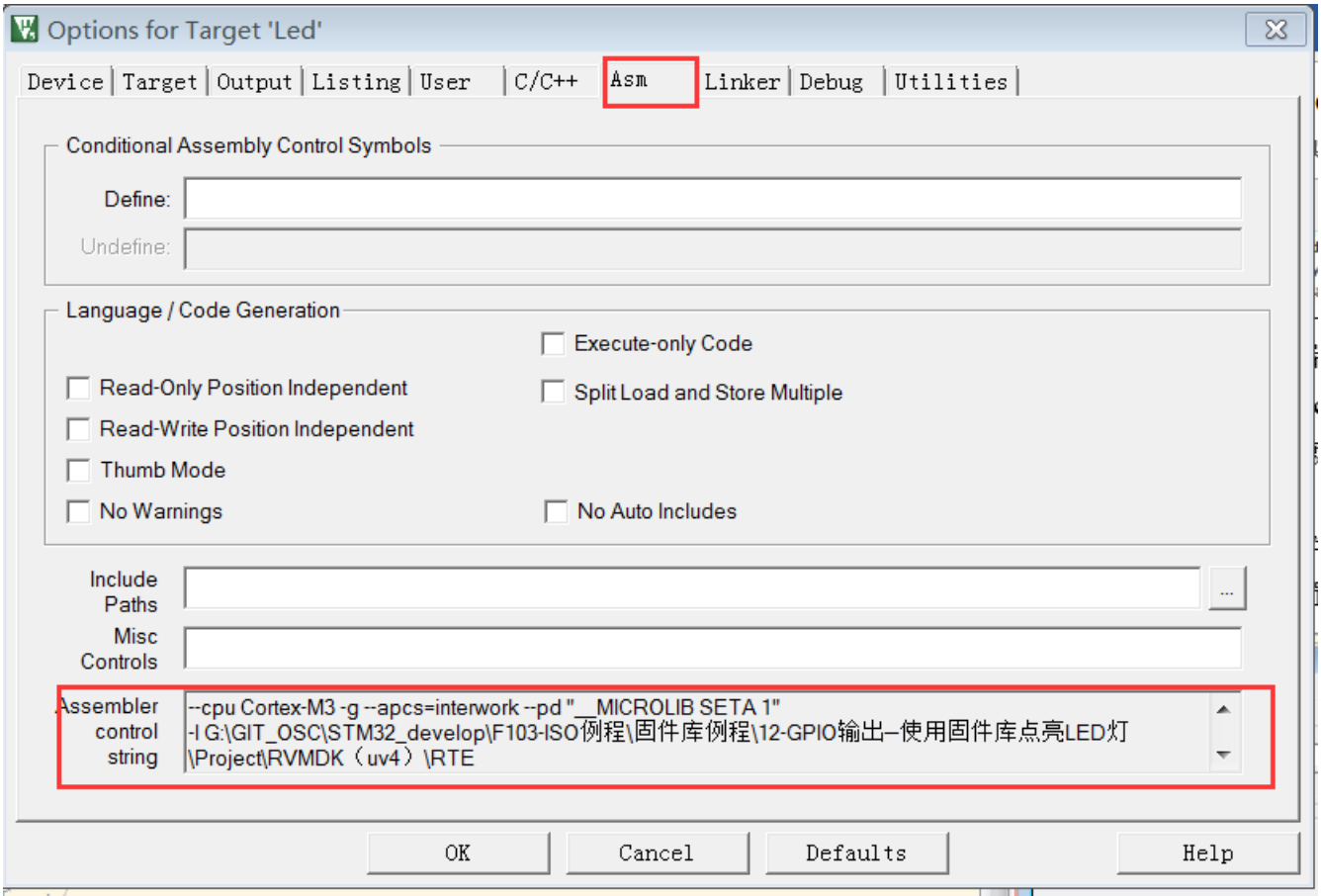


图 40-12 编译器选项说明

利用 MDK，我们一般不需要自己调用 armcc 工具，但经过这样的过程我们就会对 MDK 有更深入的认识，面对它的各种编译选项，就不会那么头疼了。

39.3.2.2 armasm

armasm 是汇编器，它把汇编文件编译成 O 文件。与 armcc 类似，MDK 对 armasm 的调用选项可在“Option for Target->Asm”页面进行配置，见图 *40_13*。


Options for Target 'Led'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Conditional Assembly Control Symbols
Define:
Undefine:
Language / Code Generation
Execute-only Code
Read-Only Position Independent
Split Load and Store Multiple
Read-Write Position Independent
Thumb Mode
No Warnings
No Auto Includes
Include Paths
Misc Controls
Assembler control string
--cpu Cortex-M3 -g --apcs=interwork --pd "__MICROLIB SETA 1"
-I G:\GIT_OSC\STM32_develop\F103-ISO例程\固件库例程\12-GPIO输出-使用固件库点亮LED灯
\Project\RVMDK（uv4）\RTE
OK
Cancel
Defaults
Help

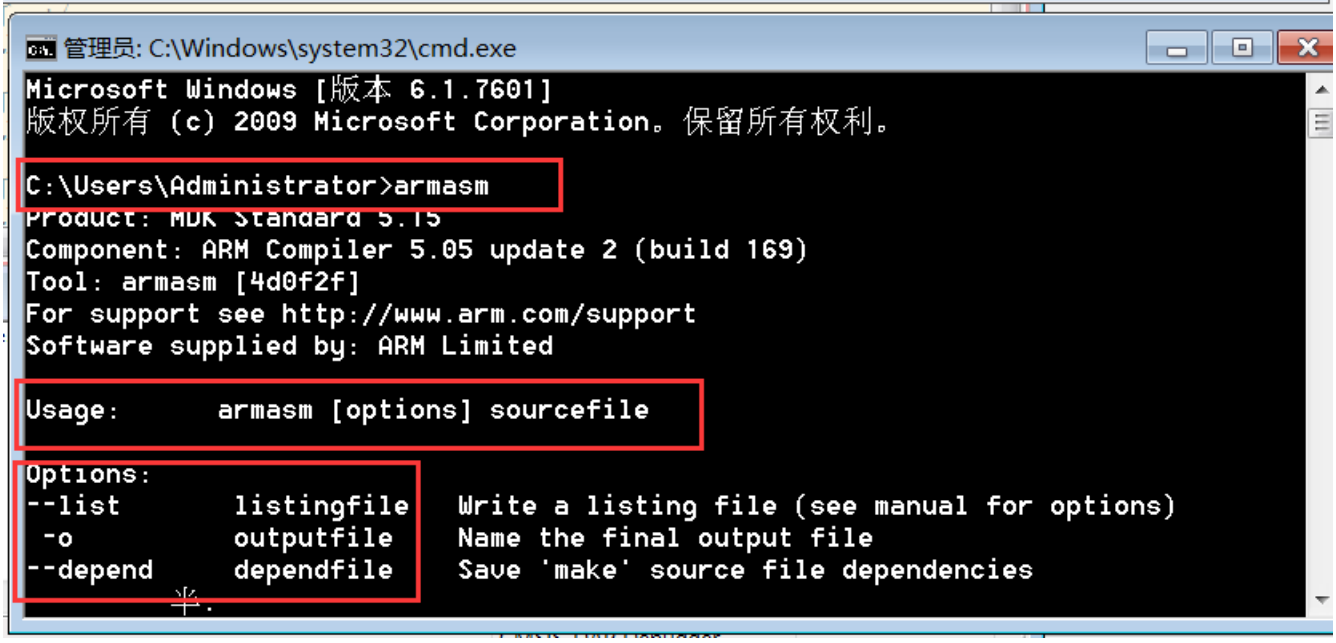

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
C:\Users\Administrator>armasm
Component: ARM Compiler 5.05 update 2 (build 169)
Tool: armasm [4d0f2f]
For support see http://www.arm.com/support
Software supplied by: ARM Limited
Usage: armasm [options] sourcefile
Options:
--list listingfile Write a listing file (see manual for options)
-o outputfile Name the final output file
--depend dependfile Save 'make' source file dependencies

图 40-13 armasm 与 MDK 的编译选项

39.3.2.3 armlink

armlink 是链接器，它把各个 O 文件链接组合在一起生成 ELF 格式的 AXF 文件，AXF 文件是可执行的，下载器把该文件中的指令代码下载到芯片后，该芯片就能运行程序了；利用 armlink 还可以控制程序存储到指定的 ROM 或 RAM 地址。在 MDK 中可在“Option for Target->Linker”页面配置 armlink 选项，见图 *40_14*。

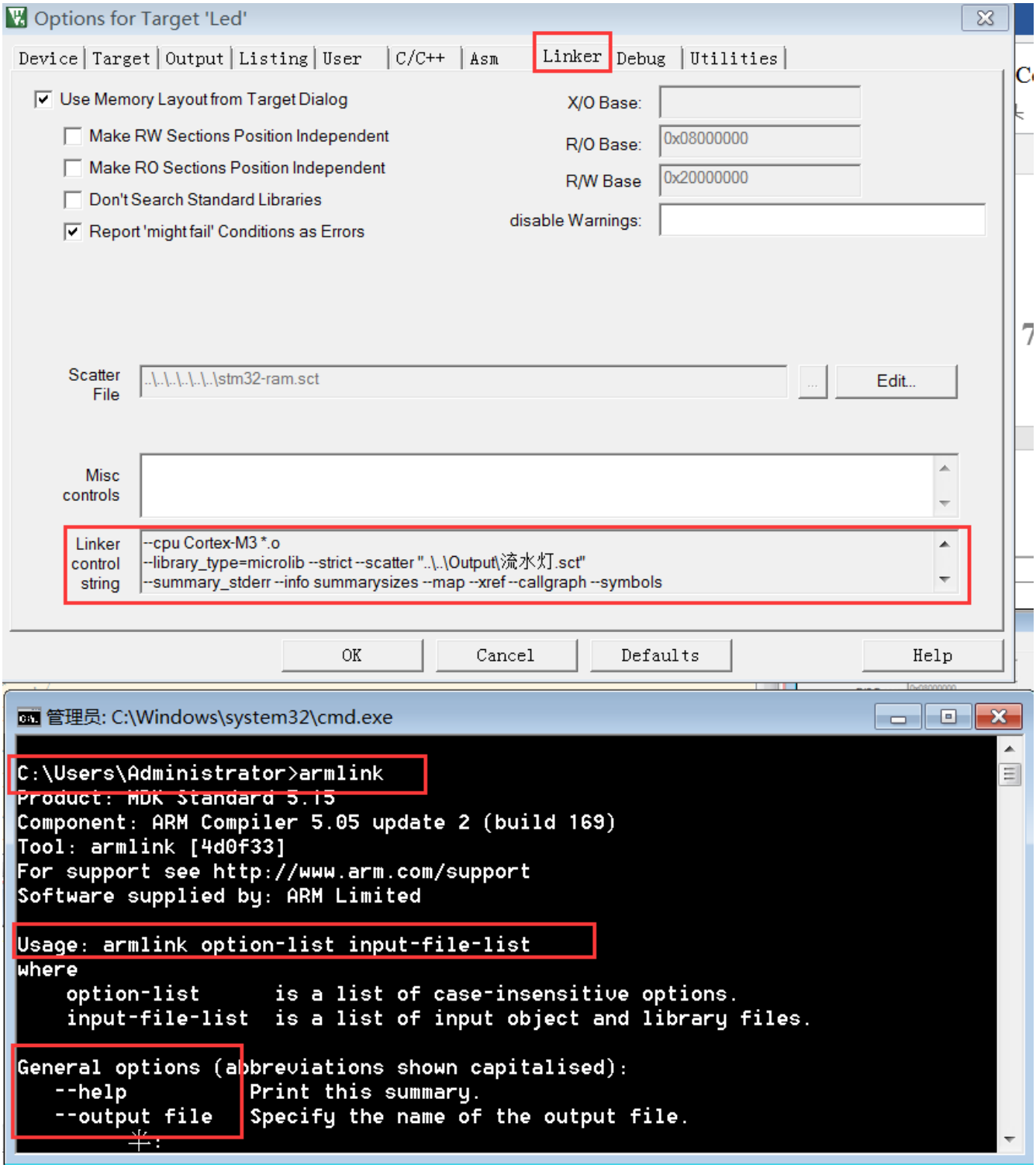


图 40-14 armlink 与 MDK 的配置选项

链接器默认是根据芯片类型的存储器分布来生成程序的，该存储器分布被记录在工程里的 sct 后缀的文件中，有特殊需要的话可自行编辑该文件，改变链接器的链接方式，具体后面我们会详细讲解。

39.3.3 armar、fromelf 及用户指令

armar 工具用于把工程打包成库文件，fromelf 可根据 axf 文件生成 hex、bin 文件，hex 和 bin 文件是大多数下载器支持的下载文件格式。

在 MDK 中，针对 armar 和 fromelf 工具的选项几乎没有，仅集成了生成 HEX 或 Lib 的选项，见图 *40_15*。

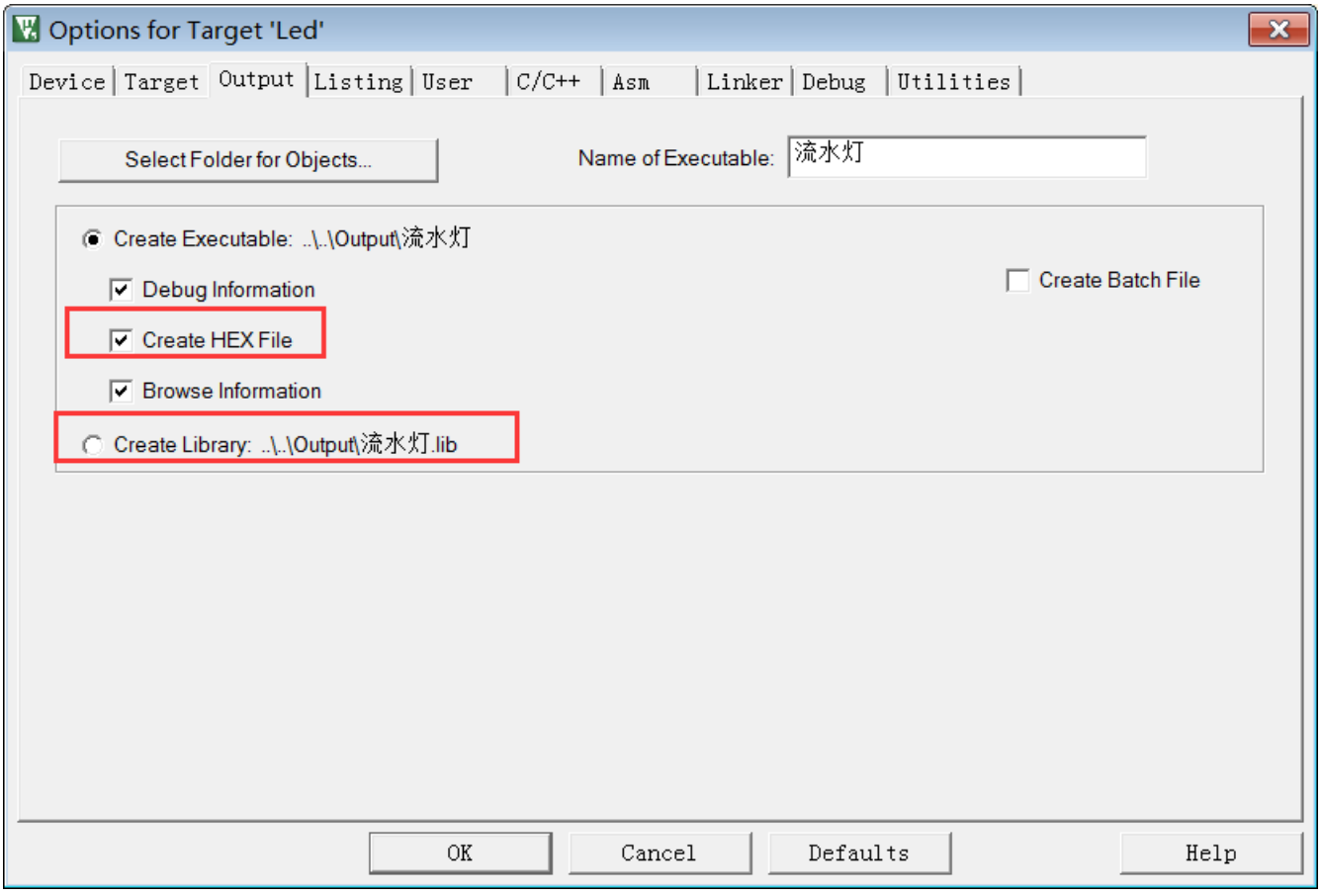


图 40-15 MDK 中，控制 fromelf 生成 hex 及控制 armar 生成 lib 的配置

例如如果我们想利用 fromelf 生成 bin 文件，可以在 MDK 的“Option for Target->User”页中添加

调用 fromelf 的指令，见图 *40_16*。

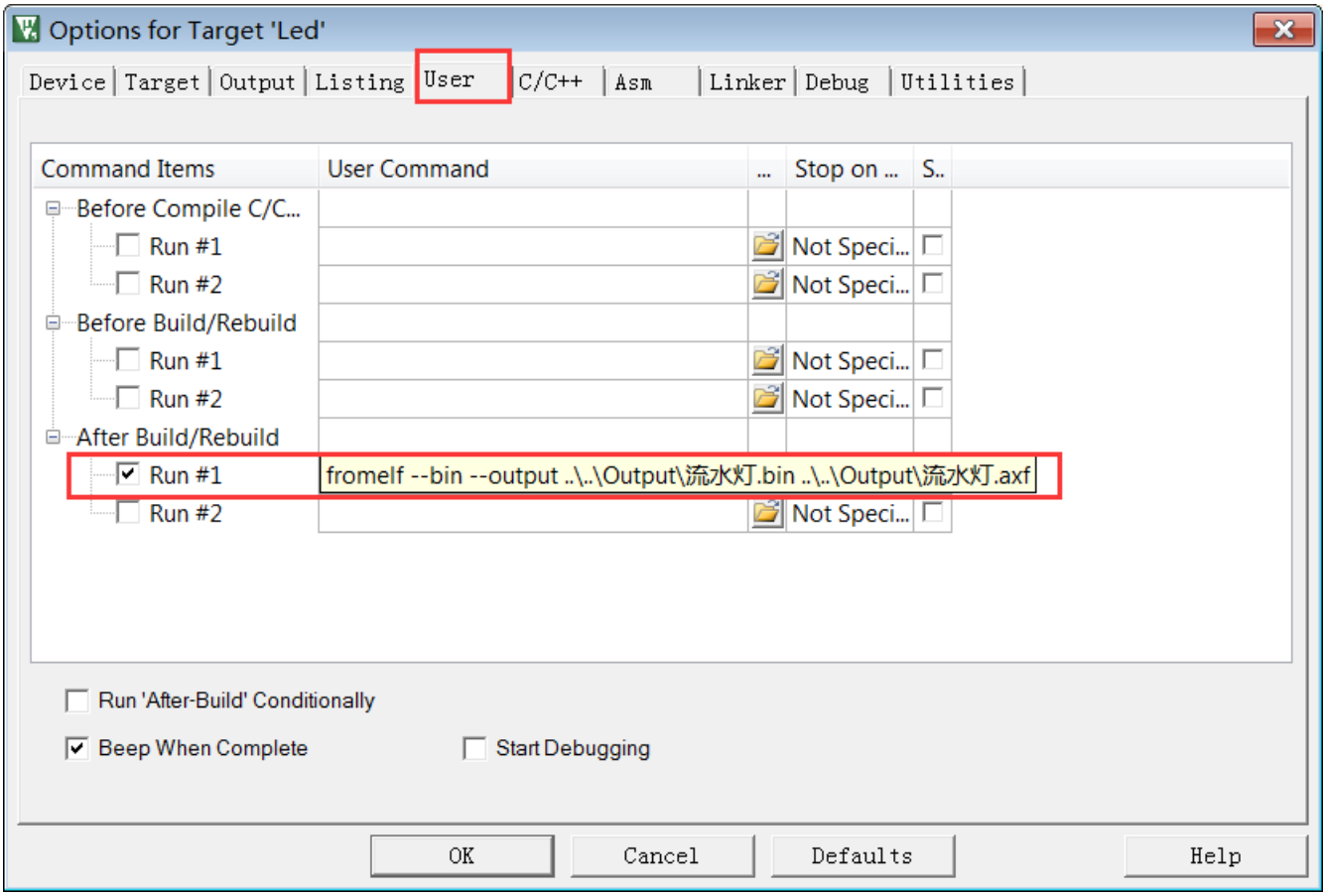


图 40-16 在 MDK 中添加指令

在 User 配置页面中，提供了三种类型的用户指令输入框，在不同组的框输入指令，可控制指令的执行时间，分别是编译前 (Before Compile c/c++ file)、构建前 (Before Build/Rebuild) 及构建后 (After Build/Rebuild) 执行。这些指令并没有限制必须是 arm 的编译工具链，例如如果您自己编写了 python 脚本，也可以在这里输入用户指令执行该脚本。

图中的生成 bin 文件指令调用了 fromelf 工具，紧跟后面的是工具的选项及输出文件名、输入文件名。由于 fromelf 是根据 axf 文件生成 bin 的，而 axf 文件又是构建 (build) 工程后才生成，所以我们把该指令放到“After Build/Rebuild”一栏。

39.4 MDK 工程的文件类型

除了上述编译过程生成的文件，MDK 工程中还包含了各种各样的文件，下面我们统一介绍，MDK 工程的常见文件类型见表 40-3。

表 40-3 MDK 常见的文件类型 (不分大小写)

后缀	说明
Project 目录下的工程文件	
*.uvguix	MDK5 工程的窗口布局文件，在 MDK4 中 *.UVGUI 后缀的文件功能相同
*.uvprojx	MDK5 的工程文件，它使用了 XML 格式记录了工程结构，双击它可以打开整
*.uvoptx	MDK5 的工程配置选项，包含 debugger、trace configuration、breakpooints 以及当
*.ini	某些下载器的配置记录文件
源文件	
*.c	C 语言源文件
*.cpp	C++ 语言源文件
*.h	C/C++ 的头文件
*.s	汇编语言的源文件
*.inc	汇编语言的头文件 (使用 “$include” 来包含)
Output 目录下的文件	
*.lib	库文件
*.dep	整个工程的依赖文件
*.d	描述了对应.o 的依赖的文件
*.crf	交叉引用文件，包含了浏览信息 (定义、引用及标识符)
*.o	可重定位的对象文件 (目标文件)
*.bin	二进制格式的映像文件，是纯粹的 FLASH 映像，不含任何额外信息
*.hex	Intel Hex 格式的映像文件，可理解为带存储地址描述格式的 bin 文件
*.elf	由 GCC 编译生成的文件，功能跟 axf 文件一样，该文件不可重定位
*.axf	由 ARMCC 编译生成的可执行对象文件，可用于调试，该文件不可重定位
*.sct	链接器控制文件 (分散加载)

表 1 – 续上页

*.scr	链接器产生的分散加载文件
*.lnp	MDK 生成的链接输入文件，用于调用链接器时的命令输入
*.htm	链接器生成的静态调用图文件
*.build_log.htm	构建工程的日志记录文件
Listing 目录下的文件	
*.lst	C 及汇编编译器产生的列表文件
*.map	链接器生成的列表文件，包含存储器映像分布
其它	
*.ini	仿真、下载器的脚本文件

这些文件主要分为 MDK 相关文件、源文件以及编译、链接器生成的文件。我们以“多彩流水灯”工程为例讲解各种文件的功能。

39.4.1 uvprojx、uvoptx 及 uvguix 工程文件

在工程的“Project”目录下主要是 MDK 工程相关的文件，见图 *40_17*。

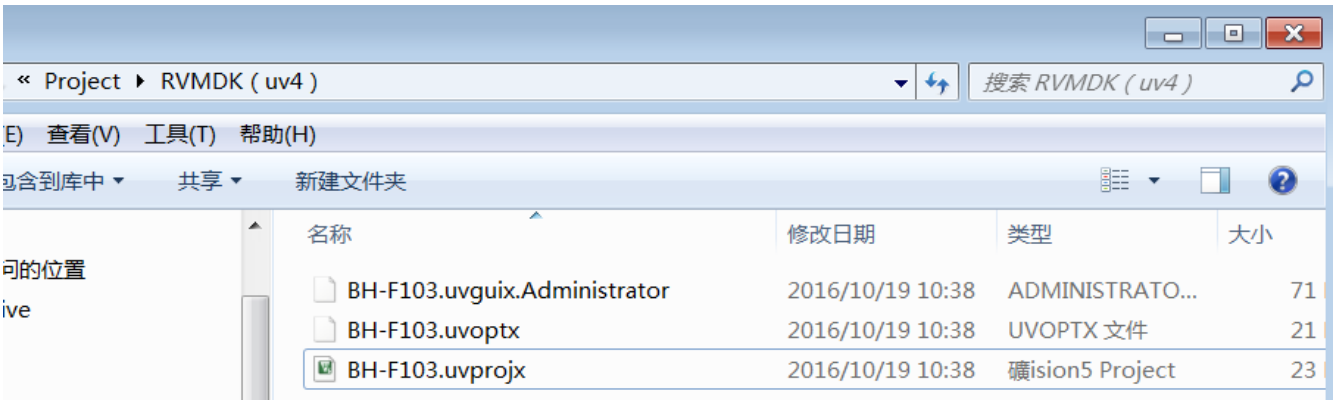


图 40-17 Project 目录下的 uvprojx、uvoptx 及 uvgui 文件

39.4.1.1 uvprojx 文件

uvprojx 文件就是我们平时双击打开的工程文件，它记录了整个工程的结构，如芯片类型、工程包含了哪些源文件等内容，见图 *40_18*。

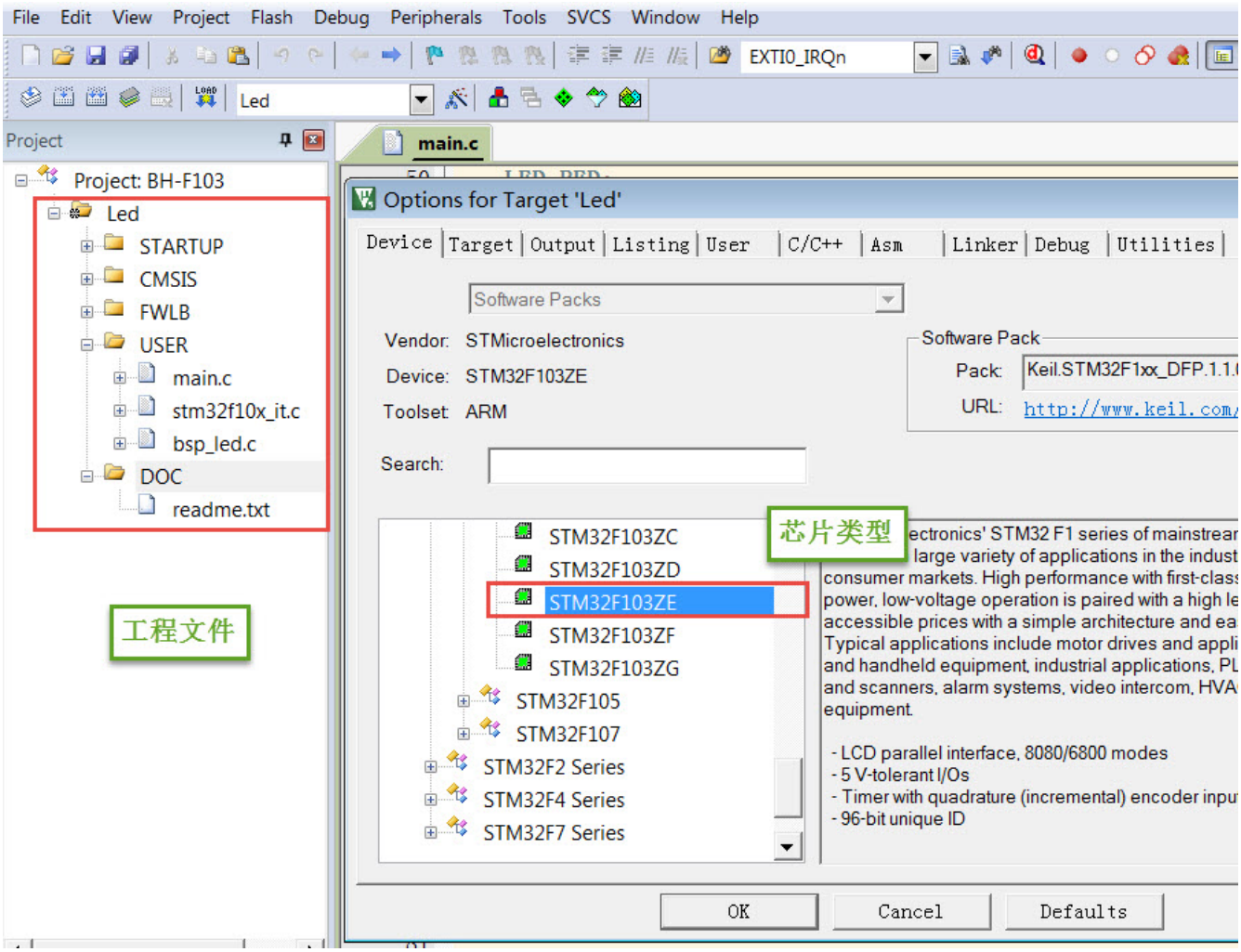


图 40-18 工程包含的文件、芯片类型等内容

39.4.1.2 uvoptx 文件

uvoptx 文件记录了工程的配置选项，如下载器的类型、变量跟踪配置、断点位置以及当前已打开的文件等等，见图 *40_19*。

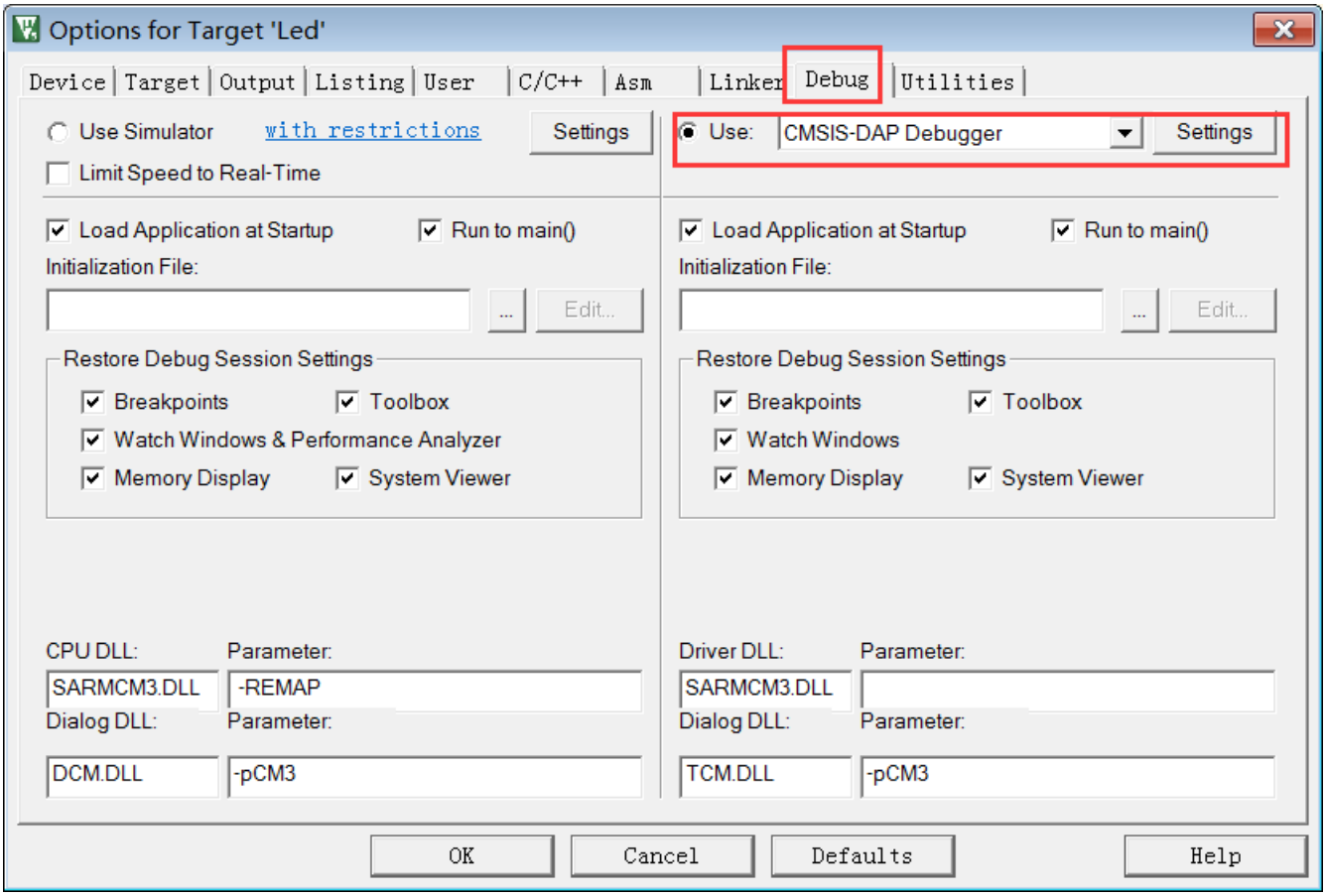


图 40-19 工程中选择使用的下载器类型

39.4.1.3 uvguix 文件

uvguix 文件记录了 MDK 软件的 GUI 布局，如代码编辑区窗口的大小、编译输出提示窗口的位置等等。

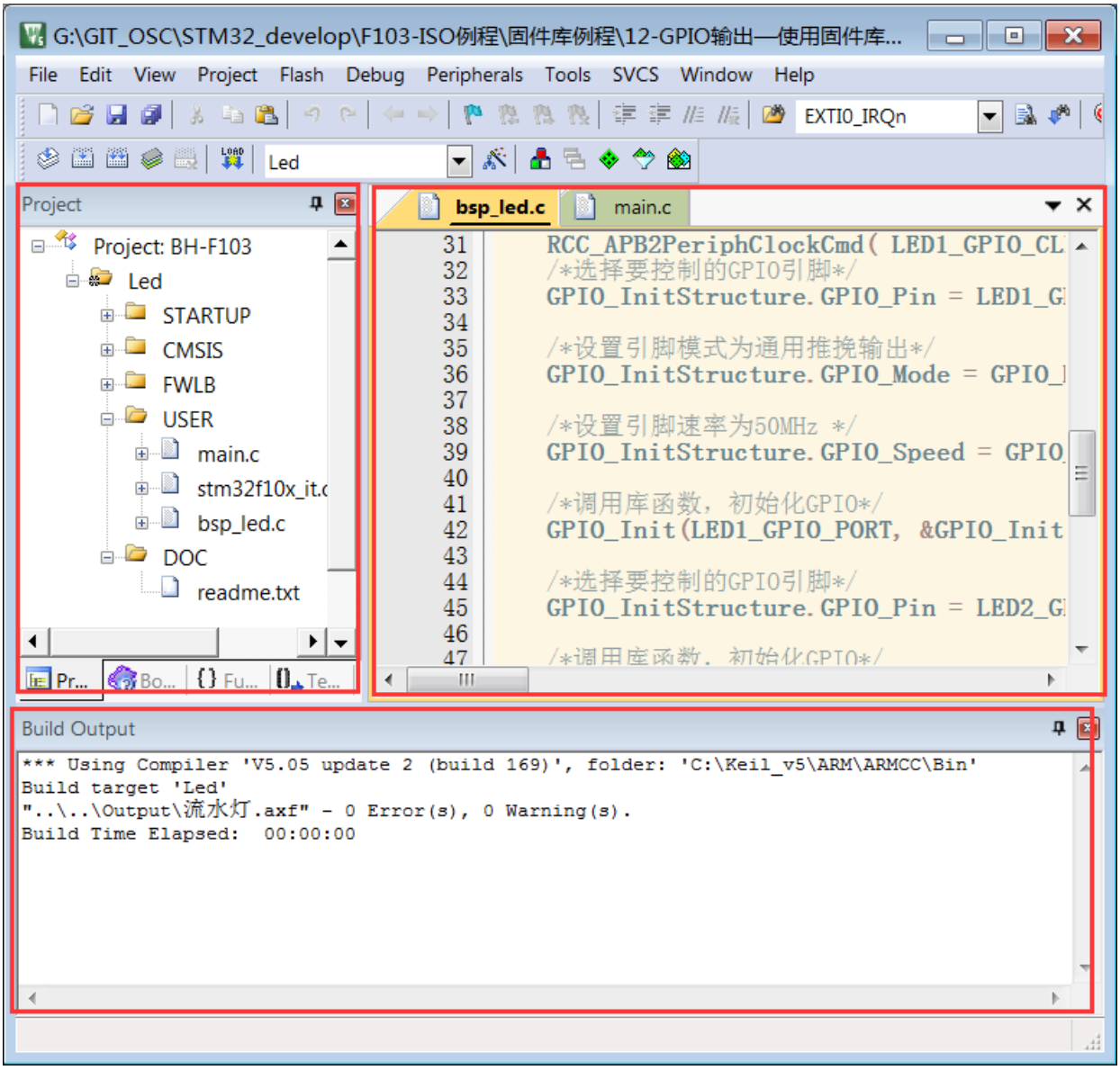


图 40-20 记录 MDK 工作环境中各个窗口的大小

uvprojx、uvoptx 及 uvguix 都是使用 XML 格式记录的文件，若使用记事本打开可以看到 XML 代码，见图 *40_21*。而当使用 MDK 软件打开时，它根据这些文件的 XML 记录加载工程的各种参数，使得我们每次重新打开工程时，都能恢复上一次的工作环境。

图 40-21 使用记事本打开 uvprojx、uvoptx 及 uvguix 文件可看到 XML 格式的记录

这些工程参数都是当 MDK 正常退出时才会被写入保存，所以若 MDK 错误退出时 (如使用 Windows 的任务管理器强制关闭)，工程配置参数的最新更改是不会被记录的，重新打开工程时要再次配置。根据这几个文件的记录类型，可以知道 uvprojx 文件是最重要的，删掉它我们就无法再正常打开工程了，而 uvoptx 及 uvguix 文件并不是必须的，可以删除，重新使用 MDK 打开 uvprojx 工程文件后，会以默认参数重新创建 uvoptx 及 uvguix 文件。(所以当使用 Git/SVN 等代码管理的时候，往往只保留 uvprojx 文件)

39.4.2 源文件

源文件是工程中我们最熟悉的内容了，它们就是我们编写的各种源代码，MDK 支持 c、cpp、h、s、inc 类型的源代码文件，其中 c、cpp 分别是 c/c++ 语言的源代码，h 是它们的头文件，s 是汇编文件，inc 是汇编文件的头文件，可使用 “$include” 语法包含。编译器根据工程中的源文件最终生成机器码。

39.4.3 Output 目录下生成的文件

点击 MDK 中的编译按钮，它会根据工程的配置及工程中的源文件输出各种对象和列表文件，在工程的“Options for Targe->Output->Select Folder for Objects”和“Options for Targe->Listing->Select Folder for Listings”选项配置它们的输出路径，见图 *40_22* 和图 *40_23*。

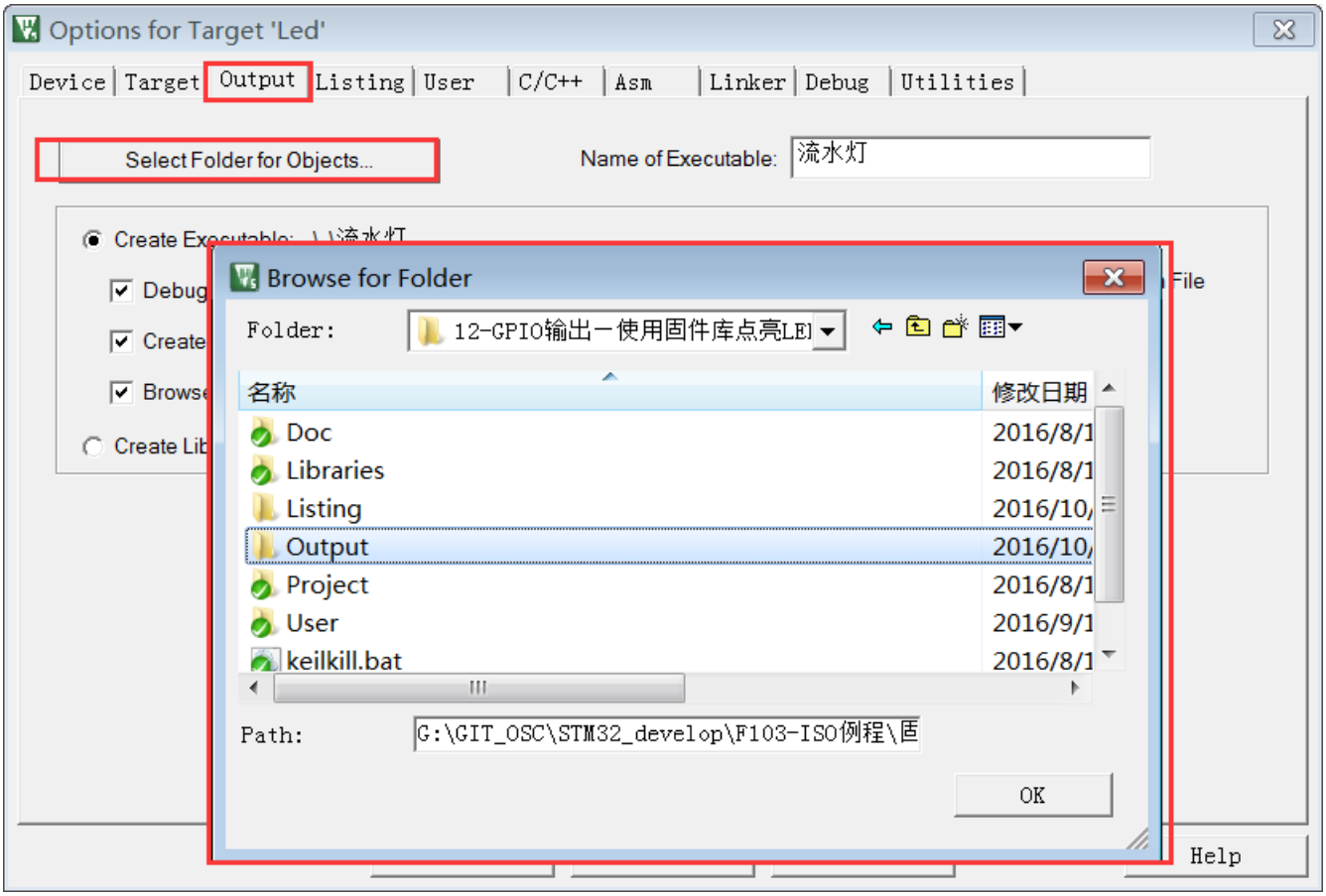


图 40-22 设置 Output 输出路径

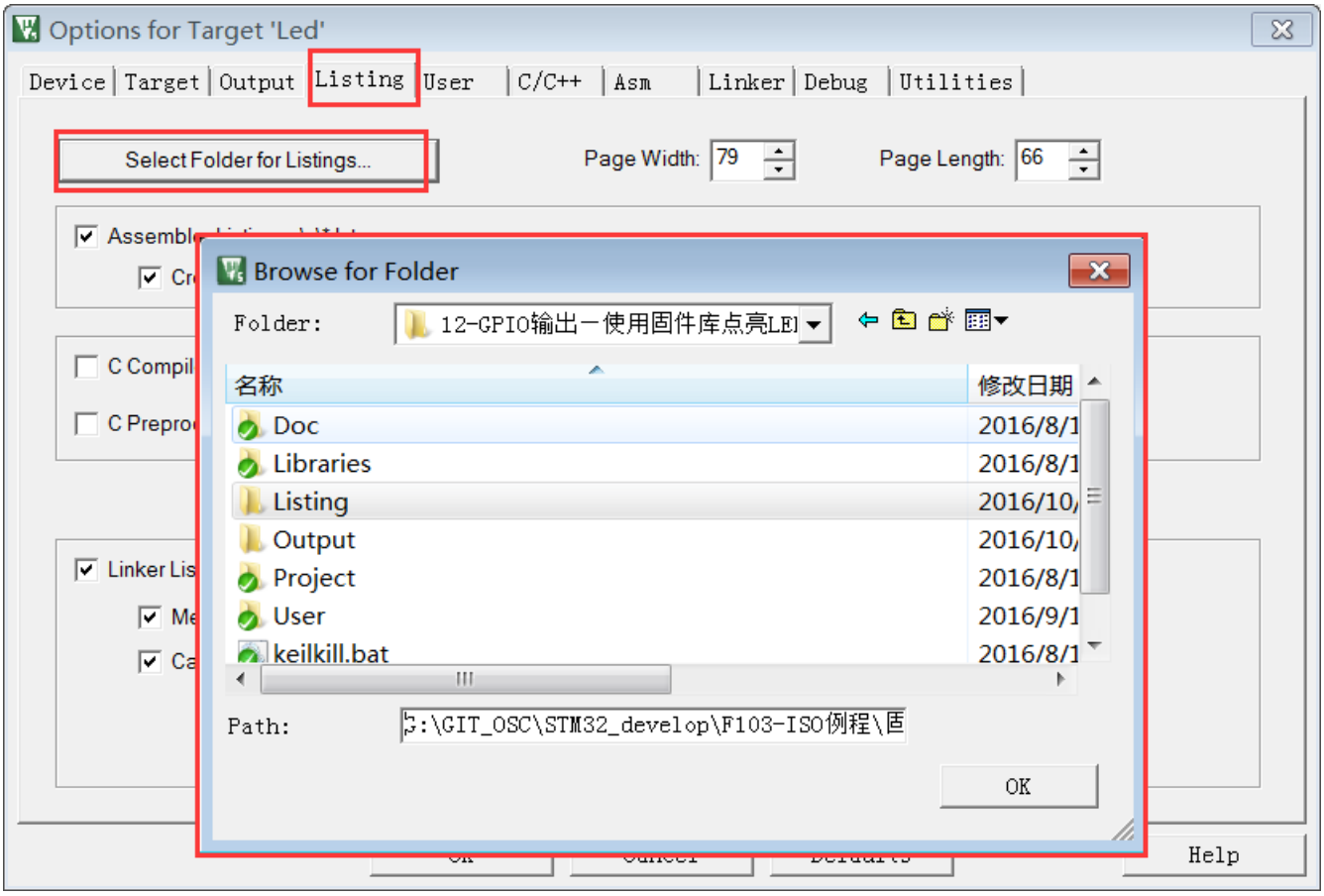


图 40-23 设置 Listing 输出路径

编译后 Output 和 Listing 目录下生成的文件见图 *40_24*。

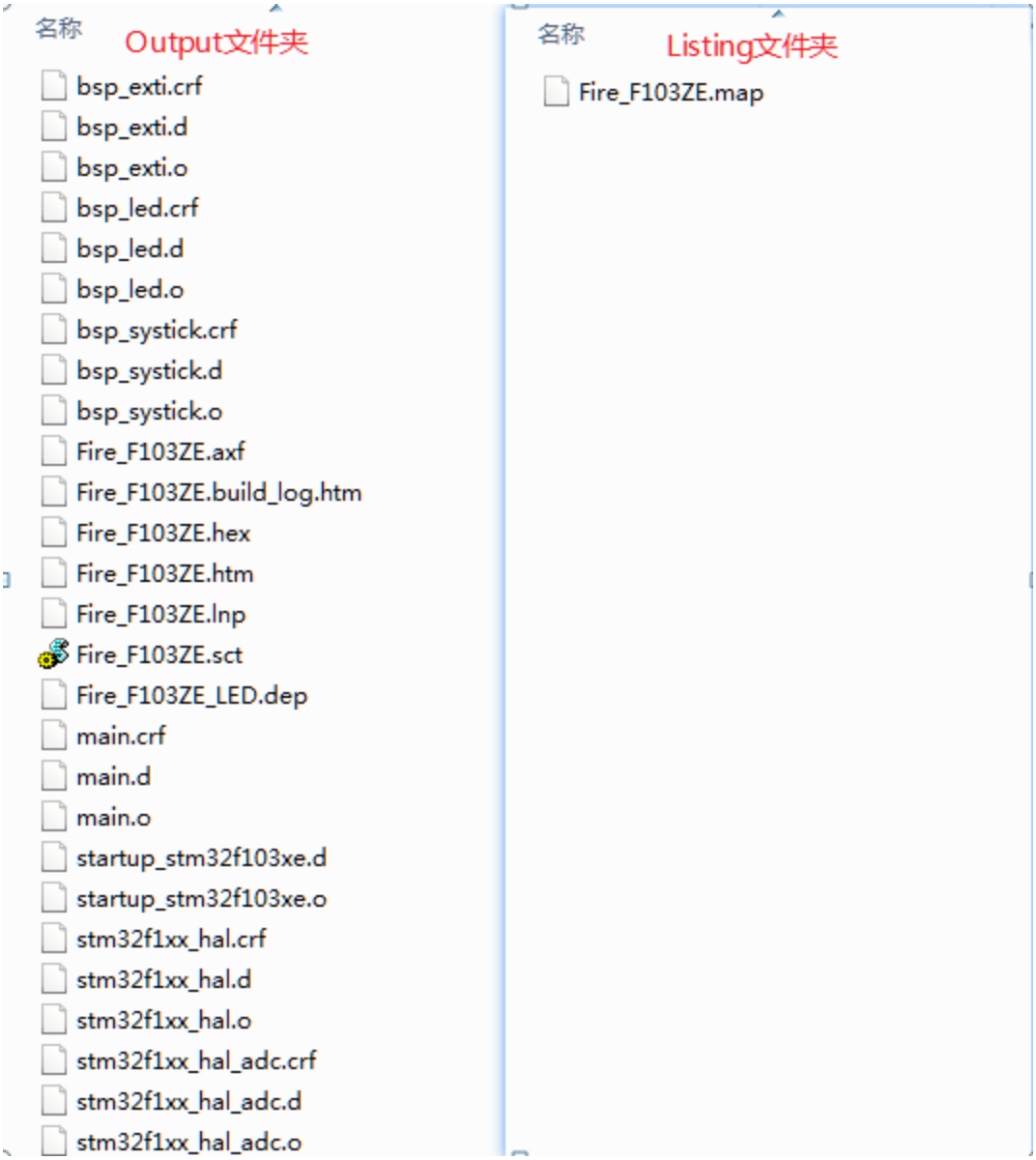


图 40-24 编译后 Output 及 Listing 文件夹中的内容

接下来我们讲解 Output 路径下的文件。

39.4.3.1 lib 库文件

在某些场合下我们希望提供给第三方一个可用的代码库，但不希望对方看到源码，这个时候我们就可以把工程生成 lib 文件 (Library file) 提供给对方，在 MDK 中可配置 “Options for Target->Create Library” 选项把工程编译成库文件，见图 *40_25*。

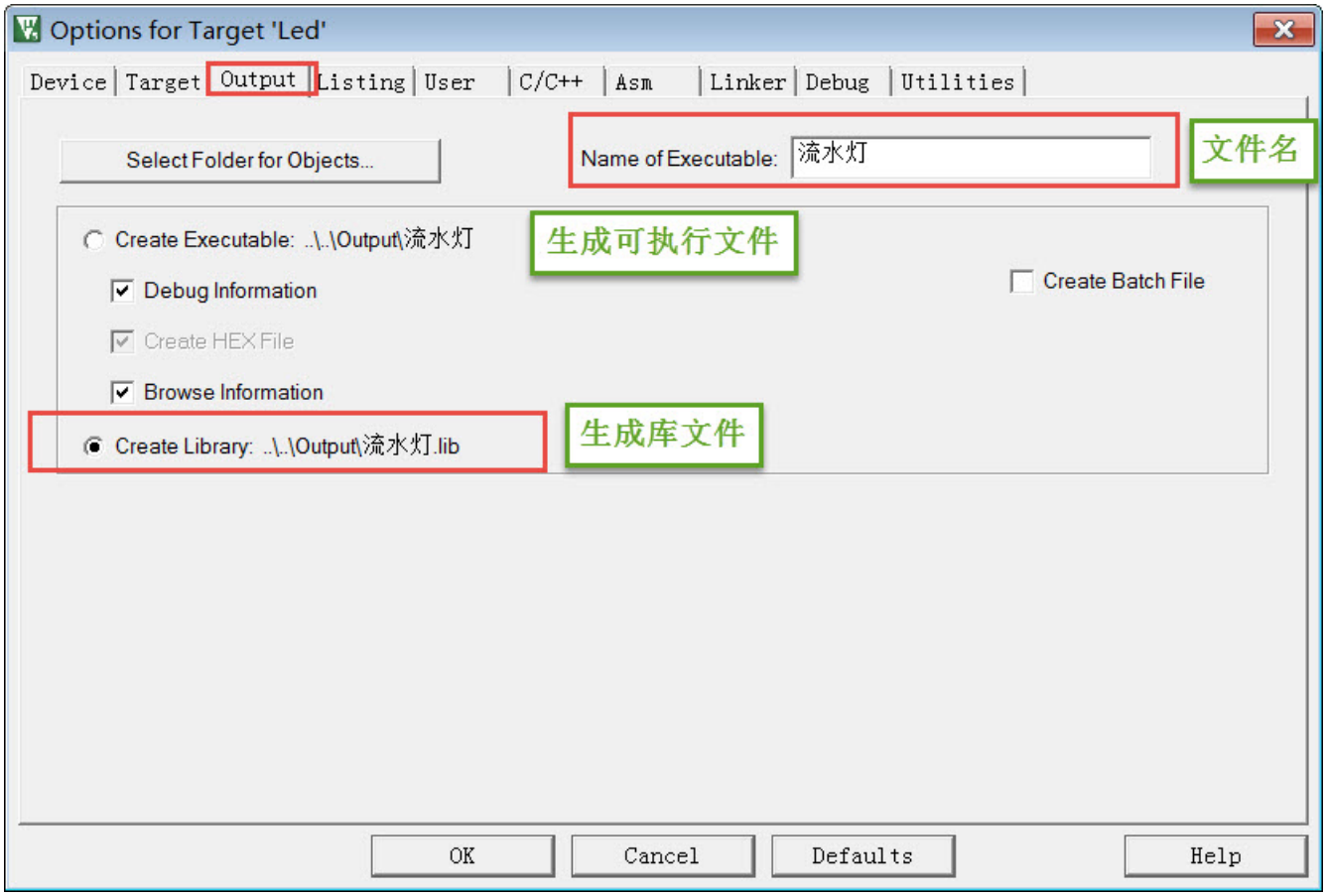


图 40-25 生成库文件或可执行文件

工程中生成可执行文件或库文件只能二选一，默认编译是生成可执行文件的，可执行文件即我们下载到芯片上直接运行的机器码。

得到生成的 *.lib 文件后，可把它像 C 文件一样添加到其它工程中，并在该工程调用 lib 提供的函数接口，除了不能看到 *.lib 文件的源码，在应用方面它跟 C 源文件没有区别。

39.4.3.2 dep、d 依赖文件

*.dep 和 *.d 文件 (Dependency file) 记录的是工程或其它文件的依赖，主要记录了引用的头文件路径，其中 *.dep 是整个工程的依赖，它以工程名命名，而 *.d 是单个源文件的依赖，它们以对应的源文件名命名。这些记录使用文本格式存储，我们可直接使用记事本打开，见图 *40_26* 和图 *40_27*。

BH-F103_Led.dep | bsp_led.d

```
1  Dependencies for Project 'BH-F103', Target 'Led': (DO NOT MODIFY !)
2  F (..\..\Libraries\CMSIS\startup\startup_stm32f10x_hd.s)(0x57B3E0C6)(--cpu Cortex-M3 -g --apcs=inte
3
4  -I G:\GIT_OSC\STM32_develop\F103-ISO例程\固件库例程\MDK编译过程及文件全解\MDK文件详解-GPIO输出-使用
5
6  -I C:\Keil_v5\ARM\PACK\Keil\STM32F1xx_DFP\1.1.0\Device\Include
7
8  -I C:\Keil_v5\ARM\CMSIS\Include
9
10 --pd "__UVISION_VERSION SETA 515" --pd "STM32F10X_HD SETA 1"
11
12 --list ..\..\listing\startup_stm32f10x_hd.lst --xref -o ..\..\output\startup_stm32f10x_hd.o --deper
```

图 40-26 工程的 dep 文件内容

BH-F103_Led.dep | bsp_led.d

```
1  ..\..\output\bsp_led.o: ..\..\User\Led\bsp_led.c
2  ..\..\output\bsp_led.o: ..\..\User\Led\bsp_led.h
3  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\stm32f10x.h
4  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\core_cm3.h
5  ..\..\output\bsp_led.o: C:\Keil_v5\ARM\ARMCC\Bin\..\include\stdint.h
6  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\system_stm32f10x.h
7  ..\..\output\bsp_led.o: ..\..\User\stm32f10x_conf.h
8  ..\..\output\bsp_led.o: ..\..\Libraries\FWlib\inc\stm32f10x_adc.h
9  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\stm32f10x.h
10 ..\..\output\bsp_led.o: ..\..\Libraries\FWlib\inc\stm32f10x_bkp.h
11 ..\..\output\bsp_led.o: ..\..\Libraries\FWlib\inc\stm32f10x_can.h
12 ..\..\output\bsp_led.o: ..\..\Libraries\FWlib\inc\stm32f10x_cec.h
```

图 40-27 bsp_led.d 文件的内容

39.4.3.3 crf 交叉引用文件

*.crf 是交叉引用文件 (Cross-Reference file)，它主要包含了浏览信息 (browse information)，即源代码中的宏定义、变量及函数的定义和声明的位置。

我们在代码编辑器中点击“Go To Definition Of ‘xxxx’”可实现浏览跳转，见图 *40_28*，跳转的时候，MDK 就是通过 *.crf 文件查找出跳转位置的。

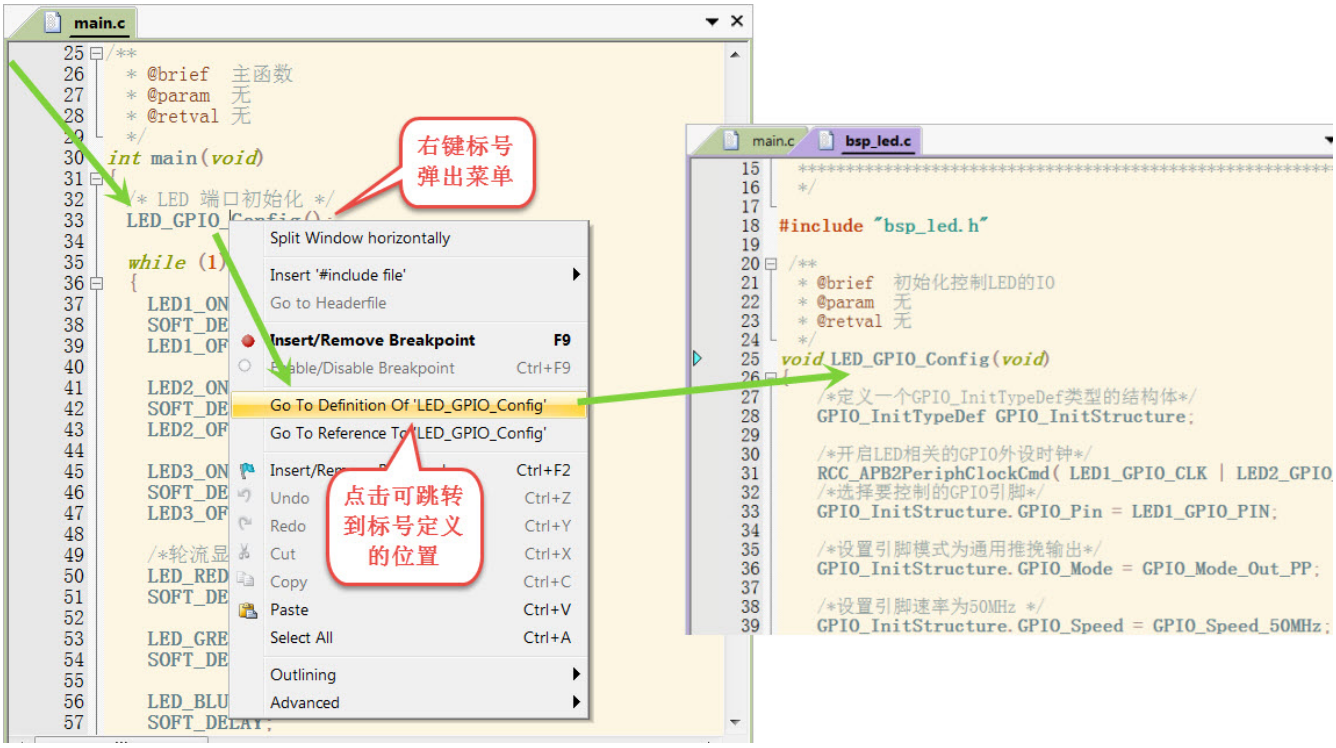


图 40-28 浏览信息

通过配置 MDK 中的“Option for Target->Output->Browse Information”选项可以设置编译时是否生成浏览信息，见图 *40_29*。只有勾选该选项并编译后，才能实现上面的浏览跳转功能。

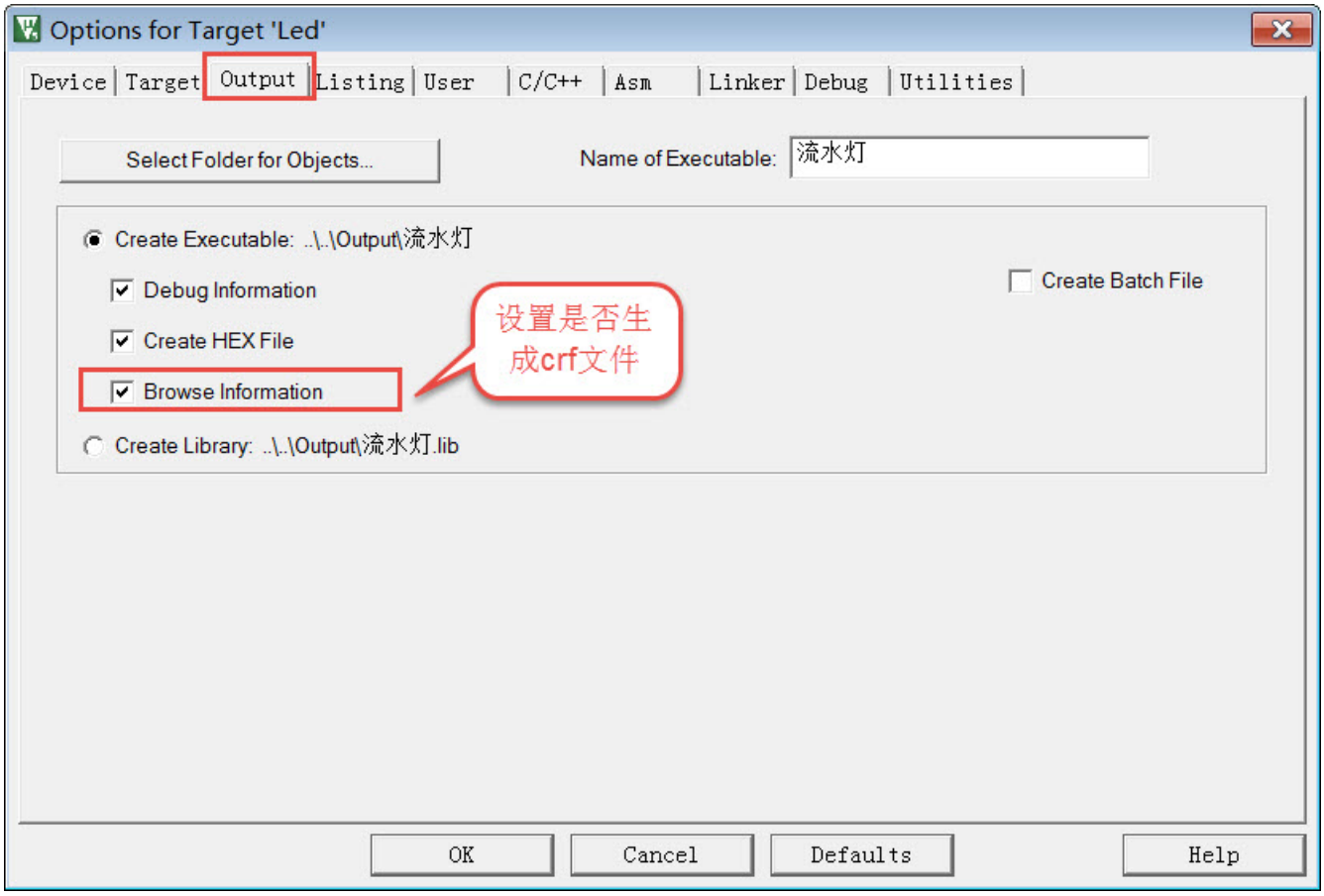


图 40-29 在 Options forTarget 中设置是否生成浏览信息

*.crf 文件使用了特定的格式表示，直接用文本编辑器打开会看到大部分乱码，见图 *40_30*，我们不作深入研究。

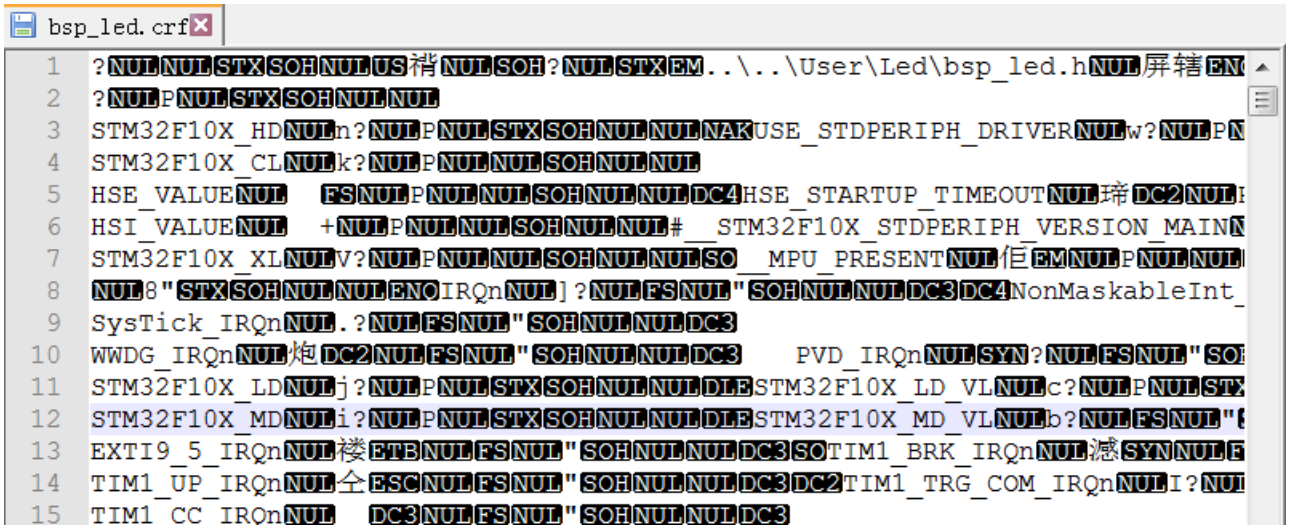


图 40-30 crf 文件内容

39.4.3.4 o、axf 及 elf 文件

.o、.elf*、.axf、*.bin* 及.hex 文件都存储了编译器根据源代码生成的机器码，根据应用场合的不同，它们又有所区别。

39.4.3.4.1 ELF 文件说明

.o、.elf*、.axf 以及前面提到的 lib 文件都是属于目标文件，它们都是使用 ELF 格式来存储的，关于 ELF 格式的详细内容请参考配套资料里的《ELF 文件格式》文档了解，它讲解的是 Linux 下的 ELF 格式，与 MDK 使用的格式有小区别，但大致相同。在本教程中，仅讲解 ELF 文件的核心概念。

ELF 是 Executable and Linking Format 的缩写，译为可执行链接格式，该格式用于记录目标文件的内容。在 Linux 及 Windows 系统下都有使用该格式的文件 (或类似格式) 用于记录应用程序的内容，告诉操作系统如何链接、加载及执行该应用程序。

目标文件主要有如下三种类型：

(1) 可重定位的文件 (Relocatable File)，包含基础代码和数据，但它的代码及数据都没有指定绝对地址，因此它适合于与其他目标文件链接来创建可执行文件或者共享目标文件。 这种文

件一般由编译器根据源代码生成。

例如 MDK 的 armcc 和 armasm 生成的 *.o 文件就是这一类，另外还有 Linux 的 *.o 文件，Windows 的 *.obj 文件。

(2) 可执行文件 (Executable File)，它包含适合于执行的程序，它内部组织的代码数据都有固定的地址 (或相对于基地址的偏移)，系统可根据这些地址信息把程序加载到内存执行。这种文件一般由链接器根据可重定位文件链接而成，它主要是组织各个可重定位文件，给它们的代码及数据一一打上地址标号，固定其在程序内部的位置，链接后，程序内部各种代码及数据段不可再重定位 (即不能再参与链接器的链接)。

例如 MDK 的 armlink 生成的 *.elf 及 *.axf 文件，(使用 gcc 编译工具可生成 *.elf 文件，用 armlink 生成的是 *.axf 文件，*.axf* 文件在.elf 之外，增加了调试使用的信息，其余区别不大，后面我们仅讲解 *.axf 文件)，另外还有 Linux 的/bin/bash 文件，Windows 的 *.exe 文件。

(3) 共享目标文件 (Shared Object File)，它的定义比较难理解，我们直接举例，MDK 生成的 *.lib 文件就属于共享目标文件，它可以继续参与链接，加入到可执行文件之中。另外，Linux 的.so，如/lib/ glibc-2.5.so，Windows 的 DLL 都属于这一类。

39.4.3.4.2 o 文件与 axf 文件的关系

根据上面的分类，我们了解到，*.axf* 文件是由多个.o 文件链接而成的，而 *.o 文件由相应的源文件编译而成，一个源文件对应一个 *.o 文件。它们的关系见图 *40_31*。

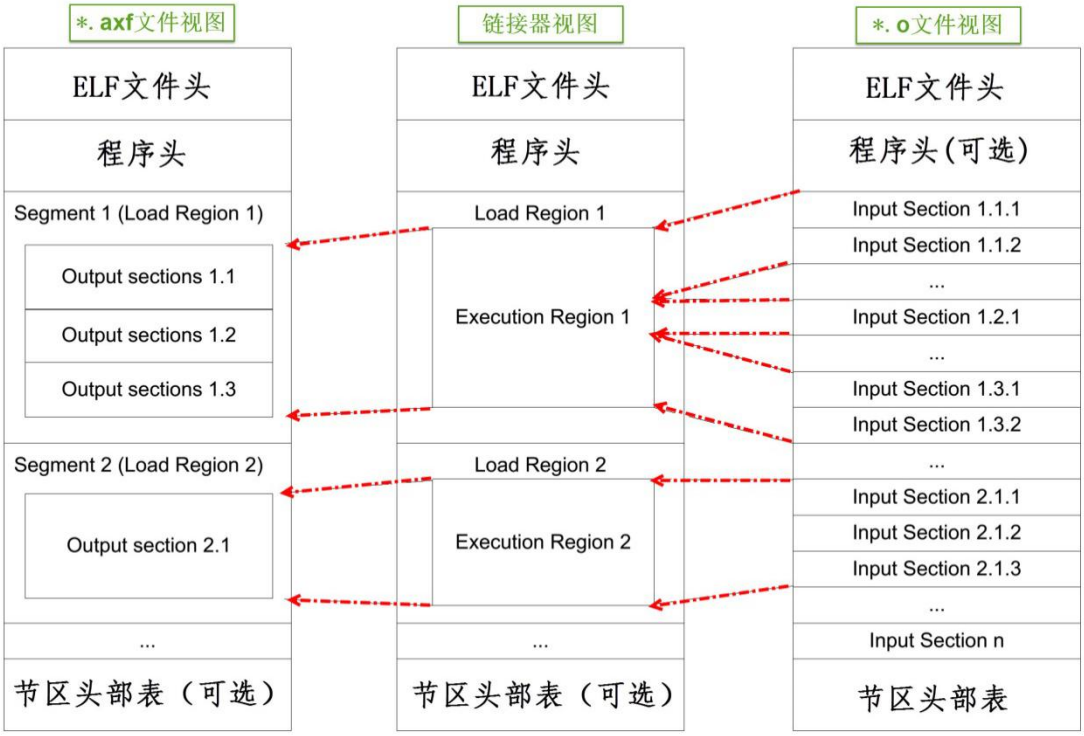


图 40-31*.axf 文件与 *.o 文件的关系

图中的中间代表的是 armlink 链接器，在它的右侧是输入链接器的 *.o 文件，左侧是它输出的 *axf 文件。

可以看到，由于都使用 ELF 文件格式，*.o* 与.axf 文件的结构是类似的，它们包含 ELF 文件头、程序头、节区 (section) 以及节区头部表。各个部分的功能说明如下：

- ELF 文件头用来描述整个文件的组织，例如数据的大小端格式，程序头、节区头在文件中的位置等。
- 程序头告诉系统如何加载程序，例如程序主体存储在本文件的哪个位置，程序的大小，程序要加载到内存什么地址等等。MDK 的可重定位文件 *.o 不包含这部分内容，因为它还不是可执行文件，而 armlink 输出的 *.axf 文件就包含该内容了。
- 节区是 *.o 文件的独立数据区域，它包含提供给链接视图使用的大量信息，如指令 (Code)、数据 (RO、RW、ZI-data)、符号表 (函数、变量名等)、重定位信息等，例如每个由 C 语言定义的函数在 *.o 文件中都会有一个独立的节区；
- 存储在最后的节区头则包含了本文件节区的信息，如节区名称、大小等等。

总的来说，链接器把各个 *.o 文件的节区归类、排列，根据目标器件的情况编排地址生成输出，汇总到 *.axf 文件。例如，见图 *40_32*，“多彩流水灯”工程中在“bsp_led.c”文件中有一个 LED_GPIO_Config 函数，而它内部调用了“stm32f1xx_hal_gpio.c”的 GPIO_Init 函数，经过 armcc 编译后，LED_GPIO_Config 及 GPIO_Iint 函数都成了指令代码，分别存储在 bsp_led.o 及 stm32f1xx_hal_gpio.o 文件中，这些指令在 *.o 文件都没有指定地址，仅包含了内容、大小以及调用的链接信息，而经过链接器后，链接器给它们都分配了特定的地址，并且把地址根据调用指向链接起来。

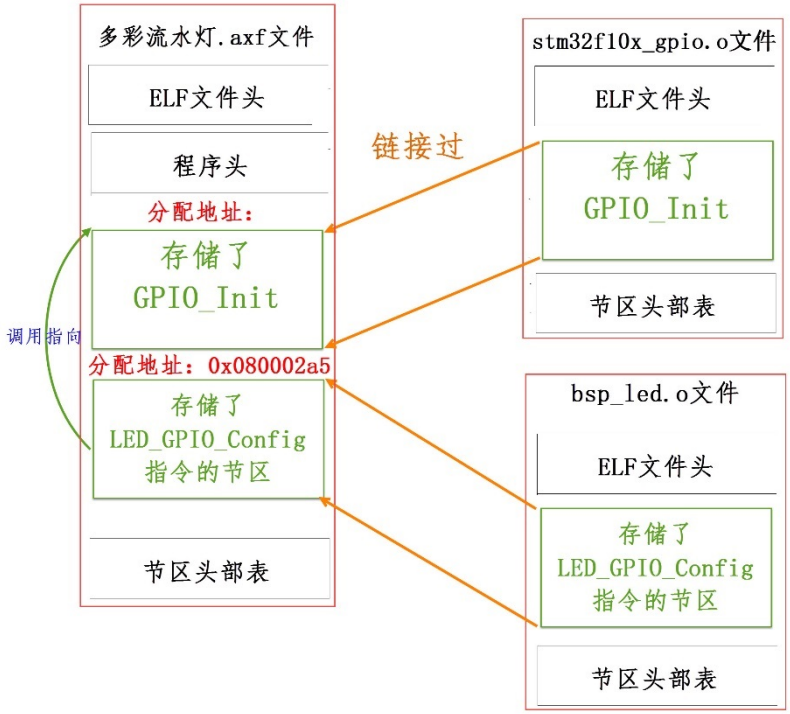


图 40-32 具体的链接过程

39.4.3.4.3 ELF 文件头

接下来我们看看具体文件的内容，使用 fromelf 文件可以查看 *.o、*.axf* 及.lib 文件的 ELF 信息。

使用命令行，切换到文件所在的目录，输入“fromelf –text –v bsp_led.o”命令，可控制输出 bsp_led.o 的详细信息，见图 *40_33*。利用“-c、-z”等选项还可输出反汇编指令文件、代码及数据文件等信息，请亲手尝试一下。

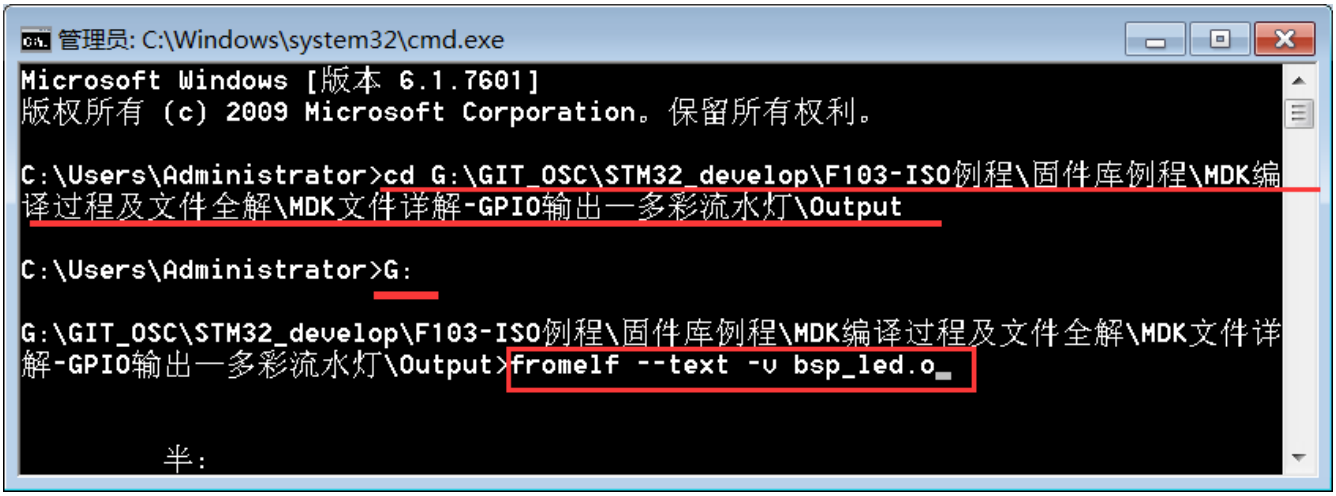


图 40-33 使用 fromelf 查看 o 文件信息

为了便于阅读，我已使用 fromelf 指令生成了“多彩流水灯.axf”、“bsp_led.o”及“多彩流水灯.lib”的 ELF 信息，并已把这些信息保存在独立的文件中，在配套资料的“elf 信息输出”文件夹下可查看，见表 40-4。

表 40-4 配套资料里使用 fromelf 生成的文件

fromelf 选项	可查看的信息	生成到配套资料里相应的文件
-v	详细信息	bsp_led_o_elfInfo_v.txt/多彩流水灯 _axf_elfInfo_v.txt
-a	数据的地址	bsp_led_o_elfInfo_a.txt/多彩流水灯 _axf_elfInfo_a.txt
-c	反汇编代码	bsp_led_o_elfInfo_c.txt/多彩流水灯 _axf_elfInfo_c.txt
-d	data section 的内容	bsp_led_o_elfInfo_d.txt/多彩流水灯 _axf_elfInfo_d.txt
-e	异常表	bsp_led_o_elfInfo_e.txt/多彩流水灯 _axf_elfInfo_e.txt
-g	调试表	bsp_led_o_elfInfo_g.txt/多彩流水灯 _axf_elfInfo_g.txt
-r	重定位信息	bsp_led_o_elfInfo_r.txt/多彩流水灯 _axf_elfInfo_r.txt
-s	符号表	bsp_led_o_elfInfo_s.txt/多彩流水灯 _axf_elfInfo_s.txt
-t	字符串表	bsp_led_o_elfInfo_t.txt/多彩流水灯 _axf_elfInfo_t.txt
-y	动态段内容	bsp_led_o_elfInfo_y.txt/多彩流水灯 _axf_elfInfo_y.txt
-z	代码及数据的大小信息	bsp_led_o_elfInfo_z.txt/多彩流水灯 _axf_elfInfo_z.txt

直接打开“elf 信息输出”目录下的 bsp_led_o_elfInfo_v.txt 文件，可看到*代码清单 40_1* 中的内容。

列表 1: 代码清单 40-1 bsp_led.o 文件的 ELF 文件头 (可到“bsp_led_o_elfInfo_v.txt”文件查看)

```
===============================================================================

** ELF Header Information

File Name:
bsp_led.o       //bsp_led.o 文件

Machine class: ELFCLASS32 (32-bit)  //32 位机
   Data encoding: ELFDATA2LSB (Little endian) //小端格式
   Header version: EV_CURRENT (Current version)
   Operating System ABI: none
   ABI Version: 0
   File Type: ET_REL (Relocatable object) (1) //可重定位类型
   Machine: EM_ARM (ARM)

   Entry offset (in SHF_ENTRYSECT section): 0x00000000
   Flags: None (0x05000000)

   ARM ELF revision: 5 (ABI version 2)

   Header size: 52 bytes (0x34)
   Program header entry size: 0 bytes (0x0)  //程序头大小
   Section header entry size: 40 bytes (0x28)

   Program header entries: 0
   Section header entries: 178

   Program header offset: 0 (0x00000000) //程序头在文件中的位置 (没有程序头)
   Section header offset: 378972 (0x0005c85c)//节区头在文件中的位置
```

(下页继续)

(续上页)

```
   Section header string table index: 175

========================================================================
```

在上述代码中已加入了部分注释，解释了相应项的意义，值得一提的是在这个 *.o 文件中，它的 ELF 文件头中告诉我们它的程序头 (Program header) 大小为“0 bytes”，且程序头所在的文件位置偏移也为“0”，这说明它是没有程序头的。

39.4.3.4.4 程序头

接下来打开“多彩流水灯 _axf_elfInfo_v.txt”文件，查看工程的 *.axf 文件的详细信息，见*代码清单 40_2*。

列表 2: 代码清单 40-2 *.axf 文件中的 elf 文件头及程序头 (可到“流水灯 _axf_elfInfo_v.txt”文件查看)

```
========================================================================

** ELF Header Information

File Name:
流水灯.axf            //流水灯.axf 文件

Machine class: ELFCLASS32 (32-bit)       //32 位机
   Data encoding: ELFDATA2LSB (Little endian)  //小端格式
   Header version: EV_CURRENT (Current version)
   Operating System ABI: none
   ABI Version: 0
   File Type: ET_EXEC (Executable) (2)      //可执行文件类型
   Machine: EM_ARM (ARM)
```

(下页继续)

(续上页)

```
   Image Entry point: 0x08000131
   Flags: EF_ARM_HASENTRY (0x05000002)

   ARM ELF revision: 5 (ABI version 2)

   Built with
Component: ARM Compiler 5.05 update 2 (build 169) Tool: armasm [4d0f2f]
Component: ARM Compiler 5.05 update 2 (build 169) Tool: armlink [4d0f33]

   Header size: 52 bytes (0x34)
   Program header entry size: 32 bytes (0x20)  //程序头大小
   Section header entry size: 40 bytes (0x28)

   Program header entries: 1
   Section header entries: 16

   Program header offset: 279836 (0x0004451c)  //程序头在文件中的位置
   Section header offset: 279868 (0x0004453c)  //节区头在文件中的位置

   Section header string table index: 15

   ============================================================================

   ** Program header #0

   Type           : PT_LOAD (1)      //表示这是可加载的内容
   File Offset    : 52 (0x34)        //在文件中的偏移
   Virtual Addr   : 0x08000000       //虚拟地址（此处等于物理地址）
   Physical Addr  : 0x08000000       //物理地址
   Size in file   : 3176 bytes (0xc68)  //程序在文件中占据的大小
   Size in memory: 4200 bytes (0x1068) //若程序加载到内存，占据的内存空间
```

(下页继续)

(续上页)

```
   Flags          : PF_X + PF_W + PF_R + PF_ARM_ENTRY (0x80000007)
   Alignment      : 8                      //地址对齐


==============================================================================
```

对比之下，可发现 *.axf 文件的 ELF 文件头对程序头的大小说明为非 0 值，且给出了它在文件的偏移地址，在输出信息之中，包含了程序头的详细信息。可看到，程序头的“Physical Addr”描述了本程序要加载到的内存地址“0x0800 0000”，正好是 STM32 内部 FLASH 的首地址；“size in file”描述了本程序占据的空间大小为“3176 bytes”，它正是程序烧录到 FLASH 中需要占据的空间。

39.4.3.4.5 节区头

在 ELF 的原文件中，紧接着程序头的一般是节区的主体信息，在节区主体信息之后是描述节区主体信息的节区头，我们先来看看节区头中的信息了解概况。通过对比 *.o 文件及 *.axf 文件的节区头部信息，可以清楚地看出这两种文件的区别，见*代码清单 40_3*。

列表 3: 代码清单 40-3 *.o 文件的节区信息（“bsp_led_o_elfInfo_v.txt”文件)

```
==============================================================================
** Section #1

Name         :
i.LED_GPIO_Config      //节区名
//此节区包含程序定义的信息，其格式和含义都由程序来解释。
Type         : SHT_PROGBITS (0x00000001)
//此节区在进程执行过程中占用内存。 节区包含可执行的机器指令。
Flags        : SHF_ALLOC + SHF_EXECINSTR (0x00000006)
Addr         : 0x00000000     //地址
File Offset : 52 (0x34)      //在文件中的偏移
```

(下页继续)

(续上页)

```
Size         : 96 bytes (0x60)    //大小
Link         :
SHN_UNDEF
Info         : 0
Alignment    : 4          //字节对齐
Entry Size   : 0
====================================
```

这个节区的名称为 LED_GPIO_Config，它正好是我们在 bsp_led.c 文件中定义的函数名。注意：编译时要勾选“Options for Target ->C/C++ -> One ELF Section per Function”中的选项，生成的 *.o 文件内部的代码区域才会与 C 文件中定义的函数名一致，否则它会把多个函数合成一个代码段，名字一般跟 C 文件中的函数名不同。见图 *40_34*。

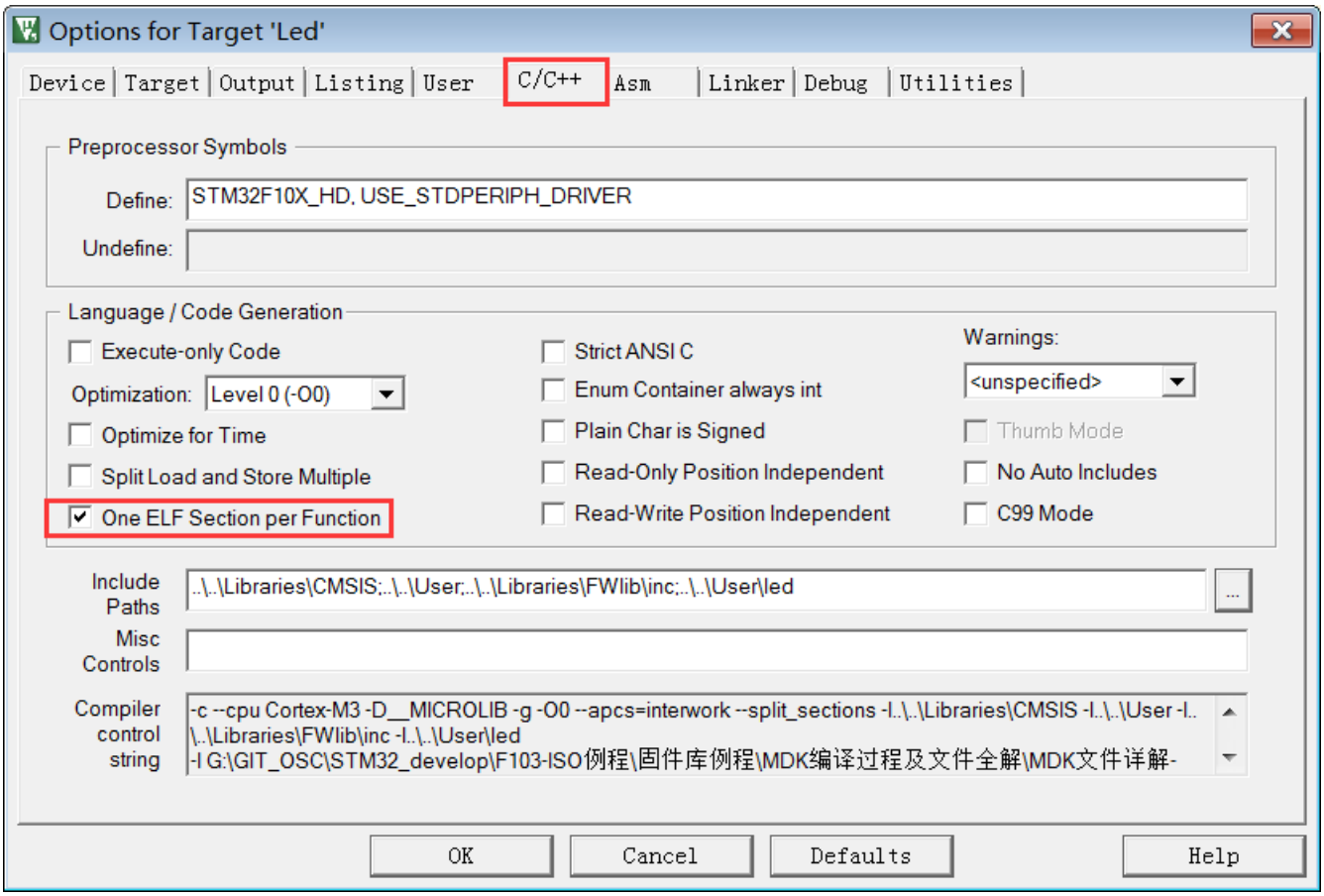


图 40-34 勾选 One ELF Section per Function

这个节区头描述的是该函数被编译后的节区信息，其中包含了节区的类型（指令类型 SHT_PROGBITS）、节区应存储到的地址 (0x00000000)、它主体信息在文件位置中的偏移 (52) 以及节区的大小 (96 bytes)。

由于 *.o 文件是可重定位文件，所以它的地址并没有被分配，是 0x00000000（假如文件中还有其它函数，该函数生成的节区中，对应的地址描述也都是 0）。当链接器链接时，根据这个节区头信息，在文件中找到它的主体内容，并根据它的类型，把它加入到主程序中，并分配实际地址，链接后生成的 *.axf 文件，我们再来看看它的内容，见*代码清单 40_4*。

列表 4: 代码清单 40-4 *.axf 文件的节区信息（“流水灯_axf_elfInfo_v.txt”文件）

```
============================================================================
** Section #1

Name         : ER_IROM1       //节区名
//此节区包含程序定义的信息，其格式和含义都由程序来解释。
Type         : SHT_PROGBITS (0x00000001)
//此节区在进程执行过程中占用内存。 节区包含可执行的机器指令
Flags        :
SHF_ALLOC + SHF_EXECINSTR (0x00000006)
Addr         : 0x08000000     //地址
File Offset  : 52 (0x34)
Size         : 3136 bytes (0xc40)//大小
Link         :
SHN_UNDEF
Info         : 0
Alignment    : 4
Entry Size   : 0
====================================
** Section #2
```

（下页继续）

(续上页)

```
Name         : RW_IRAM1//节区名
//包含将出现在程序的内存映像中的为初始
//化数据。 根据定义， 当程序开始执行， 系统
//将把这些数据初始化为 0。

Type         : SHT_PROGBITS (0x00000001)
//此节区在进程执行过程中占用内存。 节区包含进程执行过程中将可写的数据。
Flags        :
SHF_ALLOC + SHF_WRITE (0x00000003)
Addr         : 0x20000000      //地址
File Offset  : 3188 (0xc74)     //大小
Size         : 40 bytes (0x28)
Link         :
SHN_UNDEF
Info         : 0
Alignment    : 4
Entry Size   : 0
====================================
```

在 *.axf 文件中，主要包含了两个节区，一个名为 ER_IROM1，一个名为 RW_IRAM1，这些节区头信息中除了具有 *.o 文件中节区头描述的节区类型、文件位置偏移、大小之外，更重要的是它们都有具体的地址描述，其中 ER_IROM1 的地址为 0x08000000，而 RW_IRAM1 的地址为 0x20000000，它们正好是 STM32 内部 FLASH 及 SRAM 的首地址，对应节区的大小就是程序需要占用 FLASH 及 SRAM 空间的实际大小。

也就是说，经过链接器后，它生成的 *.axf 文件已经汇总了其它 *.o 文件的所有内容，生成的 ER_IROM1 节区内容可直接写入到 STM32 内部 FLASH 的具体位置。例如，前面 *.o 文件中的 i.LED_GPIO_Config 节区已经被加入到 *.axf 文件的 ER_IROM1 节区的某地址。

39.4.3.4.6 节区主体及反汇编代码

使用 fromelf 的-c 选项可以查看部分节区的主体信息，对于指令节区，可根据其内容查看相应的反汇编代码，打开“bsp_led_o_elfInfo_c.txt”文件可查看这些信息，见*代码清单 40_5*。

列表 5: 代码清单 40-5 *.o 文件的 LED_GPIO_Config 节区及反汇编代码 (bsp_led_o_elfInfo_c.txt 文件)

```
==============================================================================

** Section #1 'i.LED_GPIO_Config' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size   : 96 bytes (alignment 4)
   Address: 0x00000000

   $t
   i.LED_GPIO_Config
   LED_GPIO_Config
//  地址         内容   [ASCII 码 (无意义)]   内容对应的代码
0x00000000:    b508        ..      PUSH     {r3,lr}
0x00000002:    2101        .!      MOVS     r1,#1
0x00000004:    2008        .       MOVS     r0,#8
0x00000006:    f7fffffe    ....    BL       RCC_APB2PeriphClockCmd
0x0000000a:    2020                MOVS     r0,#0x20
0x0000000c:    f8ad0000    ....    STRH     r0,[sp,#0]
0x00000010:    2010        .       MOVS     r0,#0x10
0x00000012:    f88d0003    ....    STRB     r0,[sp,#3]
0x00000016:    2003        .       MOVS     r0,#3
0x00000018:    f88d0002    ....    STRB     r0,[sp,#2]
0x0000001c:    4669        iF      MOV      r1,sp
0x0000001e:       480f        .H        LDR    r0,[pc,#60] ; [0x5c] =␣
 ↪0x40010c00
0x00000020:    f7fffffe    ....    BL       GPIO_Init
0x00000024:    2001        .       MOVS     r0,#1
```

(下页继续)

(续上页)

```
/* 以下内容省略...*/
```

可看到，由于这是 *.o 文件，它的节区地址还是没有分配的，基地址为 0x00000000，接着在 LED_GPIO_Config 标号之后，列出了一个表，表中包含了地址偏移、相应地址中的内容以及根据内容反汇编得到的指令。细看汇编指令，还可看到它包含了跳转到 RCC_APB2PeriphClockCmd 及 GPIO_Init 标号的语句，而且这两个跳转语句原来的内容都是“f7fffffe”，这是因为还 *.o 文件中并没有 RCC_APB2PeriphClockCmd 及 GPIO_Init 标号的具体地址索引，在 *.axf 文件中，这是不一样的。

接下来我们打开“流水灯 _axf_elfInfo_c.txt”文件，查看 *.axf 文件中，ER_IROM1 节区中对应 LED_GPIO_Config 的内容，见*代码清单 40_6*。

列表 6: 代码清单 40-6*.axf 文件的 LED_GPIO_Config 反汇编代码 (流水灯 _axf_elfInfo_c.txt 文件)

```
LED_GPIO_Config
   0x08000b7c:    b508        ..      PUSH     {r3,lr}
   0x08000b7e:    2101        .!      MOVS     r1,#1
   0x08000b80:    2008        .       MOVS     r0,#8
   0x08000b82:    f7fffefd    ....    BL       RCC_APB2PeriphClockCmd ;␣
↪0x8000980
   0x08000b86:    2020                MOVS     r0,#0x20
   0x08000b88:    f8ad0000    ....    STRH     r0,[sp,#0]
   0x08000b8c:    2010        .       MOVS     r0,#0x10
   0x08000b8e:    f88d0003    ....    STRB     r0,[sp,#3]
   0x08000b92:    2003        .       MOVS     r0,#3
   0x08000b94:    f88d0002    ....    STRB     r0,[sp,#2]
   0x08000b98:    4669        iF      MOV      r1,sp
   0x08000b9a:      480f        .H        LDR  r0,[pc,#60] ; [0x8000bd8] =␣
↪0x40010c00
   0x08000b9c:    f7fffc34    ..4.    BL       GPIO_Init ; 0x8000408
   0x08000ba0:    2001        .       MOVS     r0,#1
```

(下页继续)

(续上页)

```
   0x08000ba2:    f8ad0000    ....    STRH     r0,[sp,#0]
   0x08000ba6:    4669        iF      MOV      r1,sp
   0x08000ba8:      480b        .H      LDR      r0,[pc,#44] ; [0x8000bd8] =␣
↪0x40010c00
   /* 以下内容省略...*/
```

可看到，除了基地址以及跳转地址不同之外，LED_GPIO_Config 中的内容跟 *.o 文件中的一样。另外，由于 *.o 是独立的文件，而 *.axf 是整个工程汇总的文件，所以在 *.axf 中包含了所有调用到 *.o 文件节区的内容。例如，在“bsp_led_o_elfInfo_c.txt”(bsp_led.o 文件的反汇编信息) 中不包含 RCC_APB2PeriphClockCmd 及 GPIO_Init 的内容，而在“流水灯 _axf_elfInfo_c.txt”(流水灯.axf 文件的反汇编信息) 中则可找到它们的具体信息，且它们也有具体的地址空间。

在 *.axf 文件中，跳转到 RCC_APB2PeriphClockCmd 及 GPIO_Init 标号的这两个指令后都有注释，分别是“; 0x8000980”及“; 0x8000408”，它们是这两个标号所在的具体地址，而且这两个跳转语句的跟 *.o 中的也有区别，内容分别为“f7fffefd”及“f7fffc34”(*.o 中的均为 f7fffffe)。这就是链接器链接的含义，它把不同.o 中的内容链接起来了。

39.4.3.4.7 分散加载代码

学习至此，还有一个疑问，前面提到程序有存储态及运行态，它们之间应有一个转化过程，把存储在 FLASH 中的 RW-data 数据拷贝至 SRAM。然而我们的工程中并没有编写这样的代码，在汇编文件中也查不到该过程，芯片是如何知道 FLASH 的哪些数据应拷贝到 SRAM 的哪些区域呢?

通过查看“流水灯 _axf_elfInfo_c.txt”的反汇编信息，了解到程序中具有一段名为“__scatterload”的分散加载代码，见*代码清单 40_7*，它是由 armlink 链接器自动生成的。

列表 7: 代码清单 40-7 分散加载代码 (多彩流水灯 _axf_elfInfo_c.txt 文件)

```
.text
   __scatterload
   __scatterload_rt2
```

(下页继续)

(续上页)

```
    0x08000bdc:    4c06        .L      LDR      r4,[pc,#24] ; [0x8000bf8]␣
↪= 0x8000c20
    0x08000bde:    4d07        .M      LDR      r5,[pc,#28] ; [0x8000bfc]␣
↪= 0x8000c40
    0x08000be0:    e006        ..      B        0x8000bf0 ; __scatterload␣
↪+ 20
    0x08000be2:    68e0        .h      LDR      r0,[r4,#0xc]
    0x08000be4:    f0400301    @...    ORR      r3,r0,#1
    0x08000be8:    e8940007    ....    LDM      r4,{r0-r2}
    0x08000bec:    4798        .G      BLX      r3
    0x08000bee:    3410        .4      ADDS     r4,r4,#0x10
    0x08000bf0:    42ac        .B      CMP      r4,r5
    0x08000bf2:    d3f6        ..      BCC      0x8000be2 ; __scatterload␣
↪+ 6
    0x08000bf4:    f7fffaa0    ....    BL       __main_after_scatterload ;
↪ 0x8000138
  $d
    0x08000bf8:    08000c20     ...    DCD    134220832
    0x08000bfc:    08000c40    @...    DCD    134220864
  $t
  i.__scatterload_copy
  __scatterload_copy
    0x08000c00:    e002        ..      B        0x8000c08 ; __scatterload_
↪copy + 8
    0x08000c02:    c808        ..      LDM      r0!,{r3}
    0x08000c04:    1f12        ..      SUBS     r2,r2,#4
    0x08000c06:    c108        ..      STM      r1!,{r3}
    0x08000c08:    2a00        .*      CMP      r2,#0
    0x08000c0a:    d1fa        ..      BNE      0x8000c02 ; __scatterload_
↪copy + 2
    0x08000c0c:    4770        pG      BX       lr
  i.__scatterload_null
```

(下页继续)

(续上页)

```
   __scatterload_null
      0x08000c0e:    4770        pG      BX       lr
   i.__scatterload_zeroinit
   __scatterload_zeroinit
      0x08000c10:    2000        .       MOVS     r0,#0
      0x08000c12:    e001        ..      B        0x8000c18 ; __scatterload_
↪zeroinit + 8
      0x08000c14:    c101        ..      STM      r1!,{r0}
      0x08000c16:    1f12        ..      SUBS     r2,r2,#4
      0x08000c18:    2a00        .*      CMP      r2,#0
      0x08000c1a:    d1fb        ..      BNE      0x8000c14 ; __scatterload_
↪zeroinit + 4
      0x08000c1c:    4770        pG      BX       lr
      0x08000c1e:    0000        ..      MOVS     r0,r0
```

这段分散加载代码包含了拷贝过程 (主要使用 LDM 复制指令)，而 LDM 指令的操作数中包含了加载的源地址，这些地址中包含了内部 FLASH 存储的 RW-data 数据，执行这些指令后数据就会从 FLASH 地址加载到内部 SRAM 的地址。而“__scatterload ”的代码会被“__main”函数调用，见*代码清单 40_8*，__main 在启动文件中的“Reset_Handler”会被调用，因而，在主体程序执行前，已经完成了分散加载过程。

列表 8: 代码清单 40-8 __main 的反汇编代码（部分，流水灯 _axf_elfInfo_c.txt 文件）

```
$t
.ARM.Collect$$$$00000000
.ARM.Collect$$$$00000001
__Vectors_End
__main
_main_stk
0x08000130:  f8dfd00c  ....  LDR    sp,__lit__00000000 ; [0x8000140] =␣
↪0x20000428
```

(下页继续)

(续上页)

```
.ARM.Collect$$$$00000004
_main_scatterload
0x08000134:    f000fd52    ..R.    BL        __scatterload ; 0x8000bdc
```

39.4.3.5 hex 文件及 bin 文件

若编译过程无误，即可把工程生成前面对应的 *.axf 文件，而在 MDK 中使用下载器 (DAP/JLINK/ULINK 等) 下载程序或仿真的时候，MDK 调用的就是 *.axf 文件，它解释该文件，然后控制下载器把 *.axf 中的代码内容下载到 STM32 芯片对应的存储空间，然后复位后芯片就开始执行代码了。

然而，脱离了 MDK 或 IAR 等工具，下载器就无法直接使用 *.axf 文件下载代码了，它们一般仅支持 hex 和 bin 格式的代码数据文件。默认情况下 MDK 都不会生成 hex 及 bin 文件，需要配置工程选项或使用 fromelf 命令。

39.4.3.5.1 生成 hex 文件

生成 hex 文件的配置比较简单，在“Options for Target->Output->Create Hex File”中勾选该选项，然后编译工程即可，见图 *40_35*。

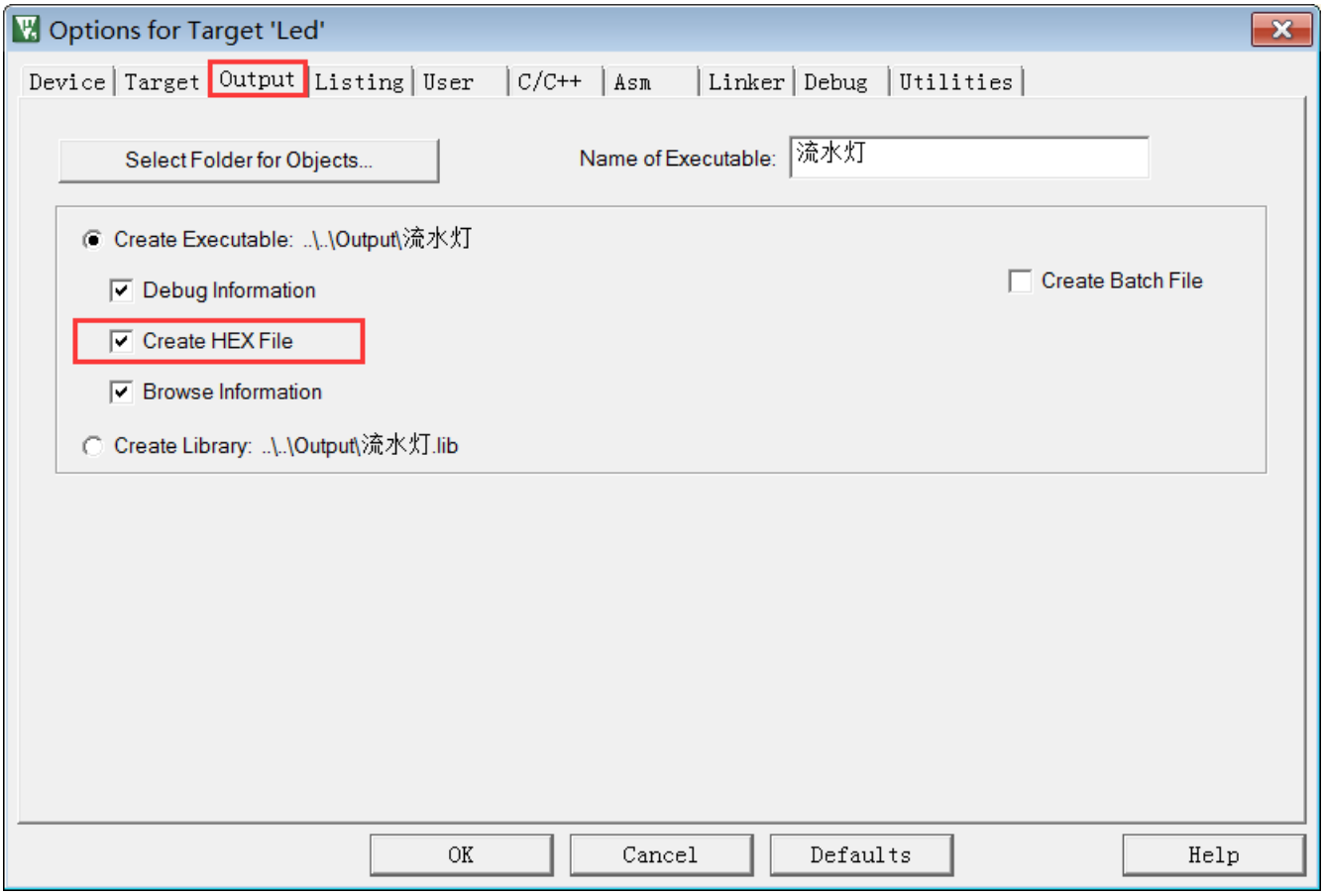


图 40-35 生成 hex 文件的配置

39.4.3.5.2 生成 bin 文件

使用 MDK 生成 bin 文件需要使用 fromelf 命令，在 MDK 的“Options For Target->Users”中加入图 *40_36* 中的命令。

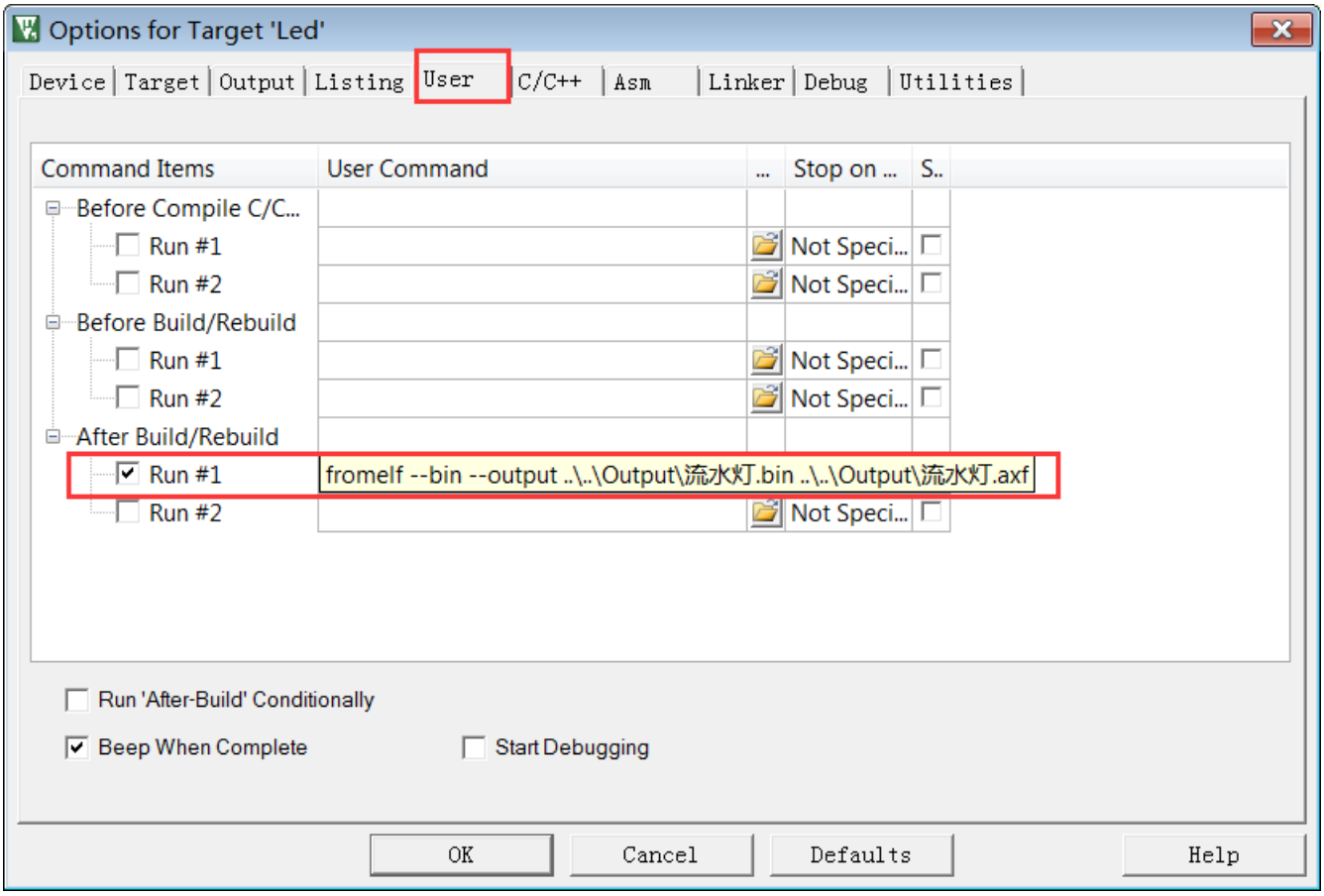


图 40-36 使用 fromelf 指令生成 bin 文件

图中的指令内容为：

“fromelf –bin –output ....Output 流水灯.bin ....Output 流水灯.axf”

该指令是根据本机及工程的配置而写的，在不同的系统环境或不同的工程中，指令内容都不一样，我们需要理解它，才能为自己的工程定制指令，首先看看 fromelf 的帮助，见图 *40_37*。

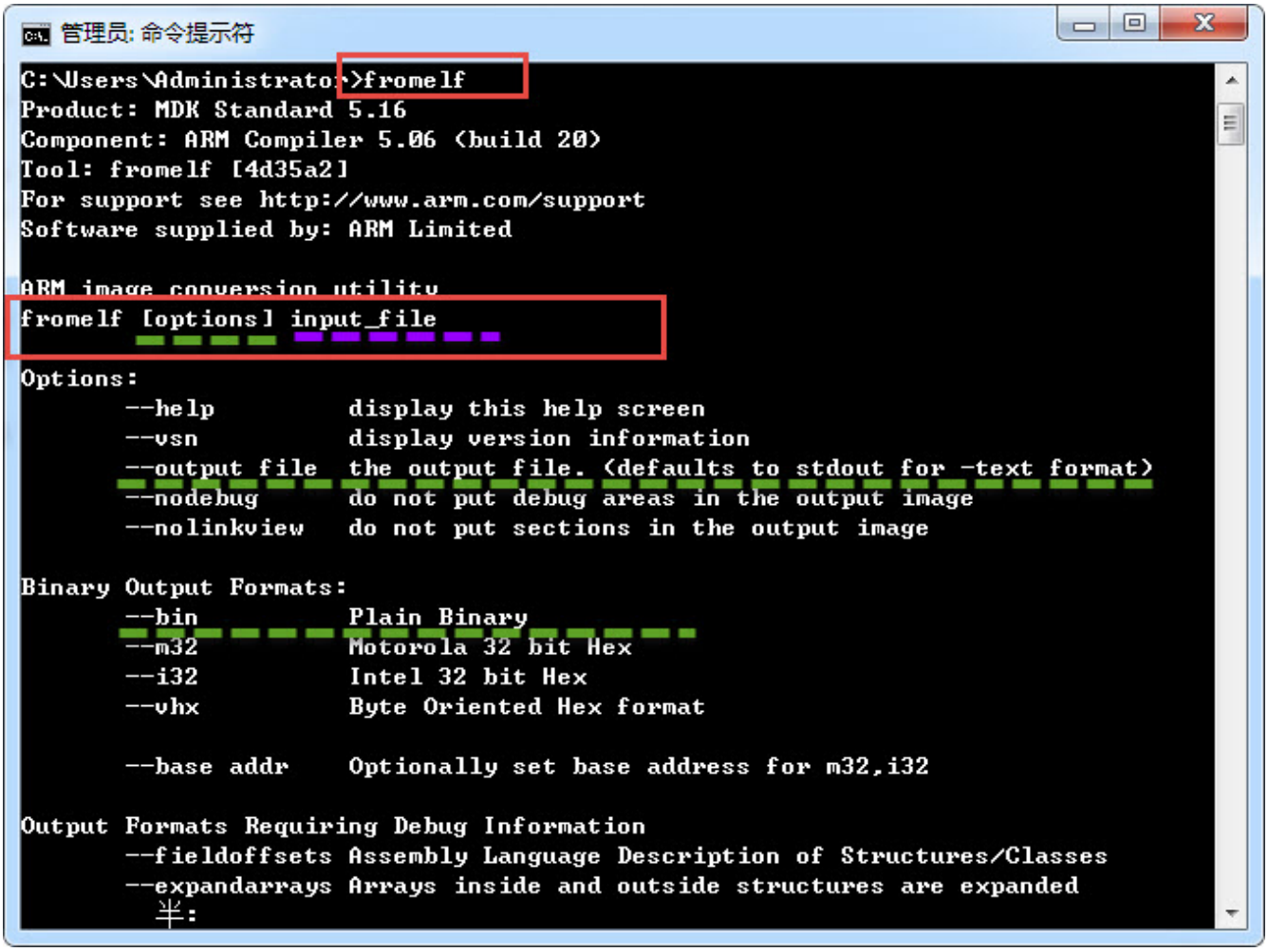


图 40-37 fromelf 的帮助

我们在 MDK 输入的指令格式是遵守 fromelf 帮助里的指令格式说明的，其格式为：

“fromelf [options] input_file”

其中 optinos 是指令选项，一个指令支持输入多个选项，每个选项之间使用空格隔开，我们的实例中使用“–bin”选项设置输出 bin 文件，使用“–output file”选项设置输出文件的名字为“....Output 流水灯.bin”，这个名字是一个相对路径格式，一个“..”表示当前目录的上一层，两个“..”表示上两层目录（当前目录是指 uvprojx 工程文件所在的位置）。如果不了解如何使用“..”表示路径，可使用 MDK 命令输入框后面的文件夹图标打开文件浏览器选择文件，加入绝对路径，在命令的最后使用“....Output 流水灯.axf”作为命令的输入文件。具体的格式分解见图 *40_38*。

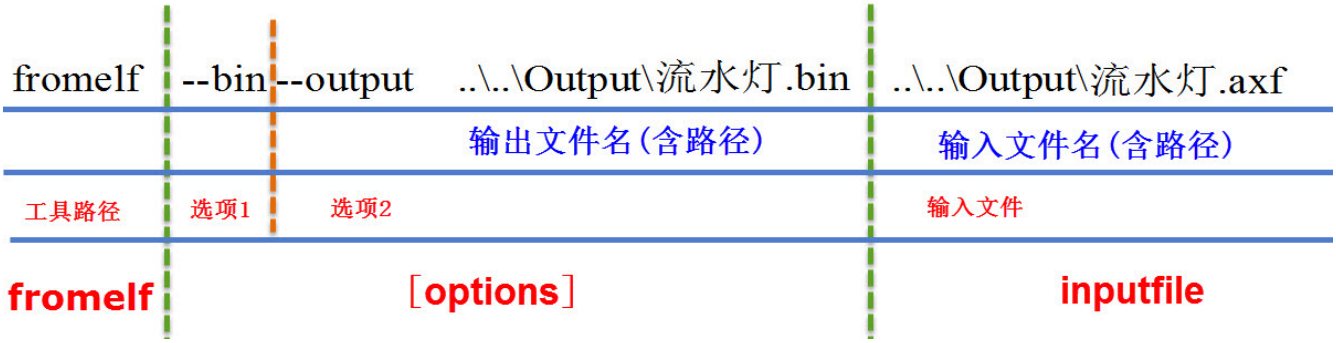


图 40-38 fromelf 命令格式分解

fromelf 需要根据工程的 *.axf 文件输入来转换得到 bin 文件，所以在命令的输入文件参数中要选择本工程对应的 *.axf 文件，在 MDK 命令输入栏中，我们把 fromelf 指令放置在“After Build/Rebuild”(工程构建完成后执行) 一栏也是基于这个考虑，这样设置后，工程构建完成生成了最新的 *.axf 文件，MDK 再执行 fromelf 指令，从而得到最新的 bin 文件。

设置完成生成 hex 的选项或添加了生成 bin 的用户指令后，点击工程的编译 (build) 按钮，重新编译工程，成功后可看到图 *40_39* 中的输出。打开相应的目录即可找到文件，若找不到 bin 文件，请查看提示输出栏执行指令的信息，根据信息改正 fromelf 指令。

```
Build Output
*** Using Compiler 'V5.05 update 2 (build 169)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Led'
After Build - User command #1: fromelf --bin --output ..\..\Output\流水灯.bin ..\..\Output\流水灯.axf
"..\..\Output\流水灯.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

图 40-39 fromelf 生成 hxe 及 bin 文件的提示

其中 bin 文件是纯二进制数据，无特殊格式，接下来我们了解一下 hex 文件格式。

39.4.3.5.3 hex 文件格式

hex 是 Intel 公司制定的一种使用 ASCII 文本记录机器码或常量数据的文件格式，这种文件常常用来记录将要存储到 ROM 中的数据，绝大多数下载器支持该格式。

一个 hex 文件由多条记录组成，而每条记录由五个部分组成，格式形如“**:llaaaatt[dd…]**cc”，例如本“多彩流水灯”工程生成的 hex 文件前几条记录见*代码清单 40_9* 。

列表 9: 代码清单 40-9 Hex 文件实例 (流水灯.hex 文件，可直接用记事本打开)

```
:020000040800F2

:10000000000400204501000829030008BF02000881

:1000100025030008D0100089D0400080000000071

:10002000000000000000000000000004D03000878

:100030009101000800000002B03000839040008AB

:100040005F0100085F0100085F0100085F01000810
```

记录的各个部分介绍如下：

- “:”：每条记录的开头都使用冒号来表示一条记录的开始；
- ll： 以 16 进制数表示这条记录的主体数据区的长度 (即后面 [dd…] 的长度)；
- aaaa: 表示这条记录中的内容应存放到 FLASH 中的起始地址；
- tt：表示这条记录的类型，它包含中的各种类型；

  表 40-5 tt 值所代表的类型说明

tt 的值	代表的类型
00	数据记录
01	本文件结束记录
02	扩展地址记录
04	扩展线性地址记录 (表示后面的记录按个这地址递增)
05	表示一个线性地址记录的起始 (只适用于 ARM)

- dd： 表示一个字节的数据，一条记录中可以有多个字节数据，ll 区表示了它有多少个字节

的数据；

- cc ： 表示本条记录的校验和，它是前面所有 16 进制数据 (除冒号外，两个为一组) 的和对 256 取模运算的结果的补码。

例如，*代码清单 40_9* 中的第一条记录解释如下：

(1) 02：表示这条记录数据区的长度为 2 字节；

(2) 0000：表示这条记录要存储到的地址；

(3) 04：表示这是一条扩展线性地址记录；

(4) 0800：由于这是一条扩展线性地址记录，所以这部分表示地址的高 16 位，与前面的 “0000” 结合在一起，表示要扩展的线性地址为 “0x0800 0000”，这正好是 STM32 内部 FLASH 的首地址；

(5) F2：表示校验和，它的值为 (0x02+0x00+0x00+0x04+0x08+0x00)%256 的值再取补码。

再来看第二条记录：

(1) 10：表示这条记录数据区的长度为 2 字节；

(2) 0000：表示这条记录所在的地址，与前面的扩展记录结合，表示这条记录要存储的 FLASH 首地址为 (0x0800 0000+0x0000)；

(3) 00：表示这是一条数据记录，数据区的是地址；

(4) 00040020450100082903000 8BF020008：这是要按地址存储的数据；

(5) 81: 校验和

为了更清楚地对比 bin、hex 及 axf 文件的差异，我们来查看这些文件内部记录的信息来进行对比。

39.4.3.5.4 hex、bin 及 axf 文件的区别与联系

bin、hex 及 axf 文件都包含了指令代码，但它们的信息丰富程度是不一样的。

- bin 文件是最直接的代码映像，它记录的内容就是要存储到 FLASH 的二进制数据 (机器码本质上就是二进制数据)，在 FLASH 中是什么形式它就是什么形式，没有任何辅助信息，包

括大小端格式也没有，因此下载器需要有针对芯片 FLASH 平台的辅助文件才能正常下载(一般下载器程序会有匹配的这些信息)；

- hex 文件是一种使用十六进制符号表示的代码记录，记录了代码应该存储到 FLASH 的哪个地址，下载器可以根据这些信息辅助下载；
- axf 文件在前文已经解释，它不仅包含代码数据，还包含了工程的各种信息，因此它也是三个文件中最大的。

同一个工程生成的 bin、hex 及 axf 文件的大小见图 *40_40*。

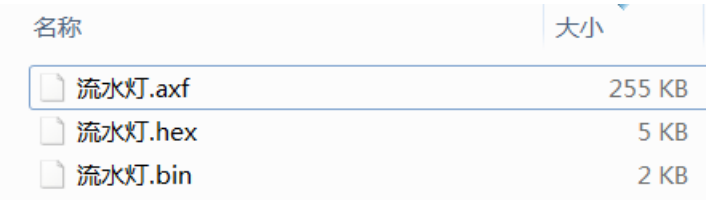


图 40-40 同一个工程的 bin、bex 及 axf 文件大小

实际上，这个工程要烧写到 FLASH 的内容总大小为 1492 字节，然而在 Windows 中查看的 bin 文件却比它大 (bin 文件是 FLASH 的代码映像，大小应一致)，这是因为 Windows 文件显示单位的原因，使用右键查看文件的属性，可以查看它实际记录内容的大小，见图 *40_41*。

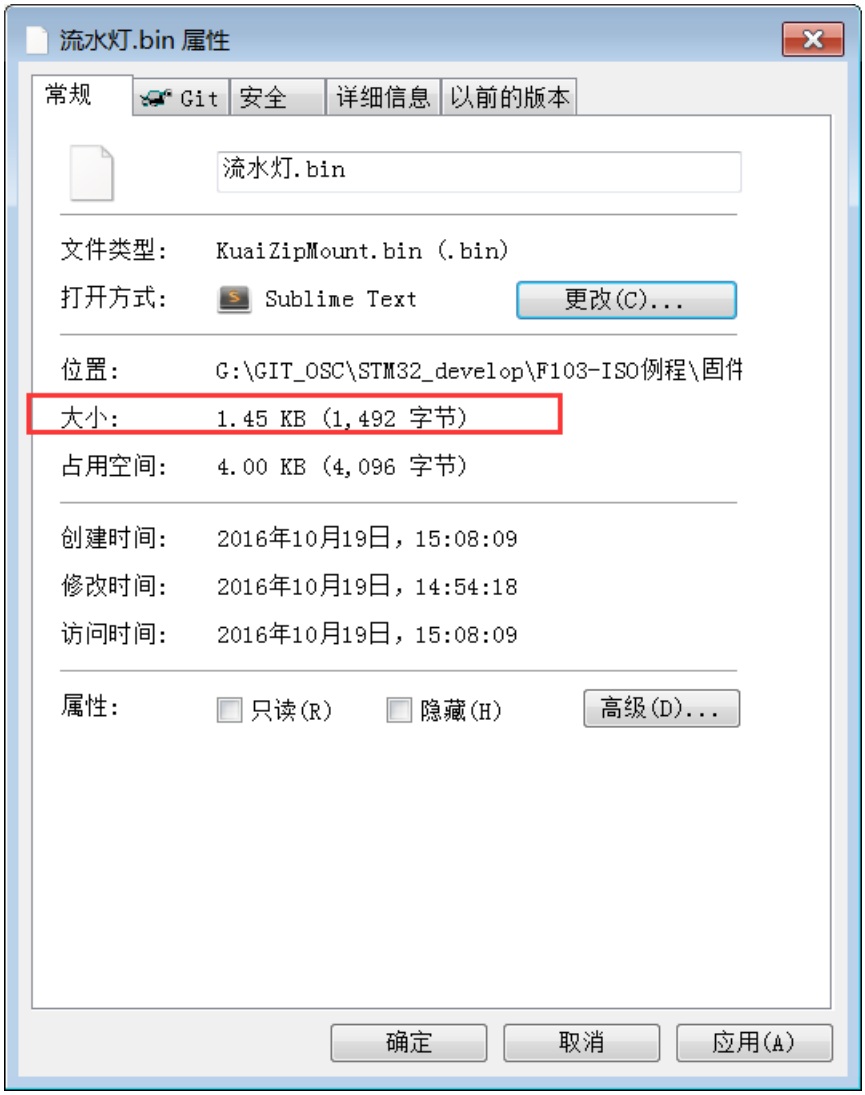


图 40-41 bin 文件大小

接下来我们打开本工程的“流水灯.bin”、“流水灯.hex”及由“流水灯.axf”使用 fromelf 工具输出的反汇编文件“流水灯 _axf_elfInfo_c.txt”文件，清晰地对比它们的差异，见图 *40_42*。如果您想要亲自阅读自己电脑上的 bin 文件，推荐使用 sublime 软件打开，它可以把二进制数以 ASCII 码呈现出来，便于阅读。

流水灯.bin

```
0004 0020 4501 0008 2903 0008 bf02 0008
2503 0008 8d01 0008 9d04 0008 0000 0000
0000 0000 0000 0000 0000 0000 4d03 0008
9101 0008 0000 0000 2b03 0008 3904 0008
5f01 0008 5f01 0008 5f01 0008 5f01 0008
5f01 0008 5f01 0008 5f01 0008 5f01 0008
5f01 0008 5f01 0008 5f01 0008 5f01 0008
5f01 0008 5f01 0008 5f01 0008 5f01 0008
5f01 0008 5f01 0008 5f01 0008 5f01 0008
```

流水灯.hex

```
:020000040800F2
:10000000000040020450100082903000 8BF02000881
:1000100025030008 8D0100089D0400080000000071
:1000200000000000000000000000000004D03000878
:1000300091010008000000002B0300083904000 8AB
:100040005F0100085F0100085F0100085F01000810
:100050005F0100085F0100085F0100085F01000800
:100060005F0100085F0100085F0100085F010008F0
:100070005F0100085F0100085F0100085F010008E0
```

流水灯_axf_elfInfo_c.txt

```
43
44  ================================================================
45
46  ** Section #1 'ER_IROM1' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXEC
47      Size   : 1492 bytes (alignment 4)
48      Address: 0x08000000
49
50      $d.realdata
51      RESET
52      __Vectors
53          0x08000000:    20000400    ...     DCD    536871936
54          0x08000004:    08000145    E...    DCD    134218053
55          0x08000008:    08000329    )...    DCD    134218537
56          0x0800000c:    080002bf    ....    DCD    134218431
57          0x08000010:    08000325    %...    DCD    134218533
58          0x08000014:    0800018d    ....    DCD    134218125
59          0x08000018:    0800049d    ....    DCD    134218909
```

图 40-42 同一个工程的 bin、hex 及 axf 文件对代码的记录

在“流水灯 _axf_elfInfo_c.txt”文件中不仅可以看到代码数据，还有具体的标号、地址以及反汇编得到的代码，虽然它不是 *.axf 文件的原始内容，但因为它是通过 *.axf 文件 fromelf 工具生成的，我们可认为 *.axf 文件本身记录了大量这些信息，它的内容非常丰富，熟悉汇编语言的人可轻松阅读。

在 hex 文件中包含了地址信息以及地址中的内容，而在 bin 文件中仅包含了内容，连存储的地址信息都没有。观察可知，bin、hex 及 axf 文件中的数据内容都是相同的，它们存储的都是机器码。这就是它们三都之间的区别与联系。

由于文件中存储的都是机器码，见图 *40_43*，该图是我根据 axf 文件的 GPIO_Init 函数的机器码，在 bin 及 hex 中找到的对应位置。所以经验丰富的人是有可能从 bin 或 hex 文件中恢复出汇编代码的，只是成本较高，但不是不可能。

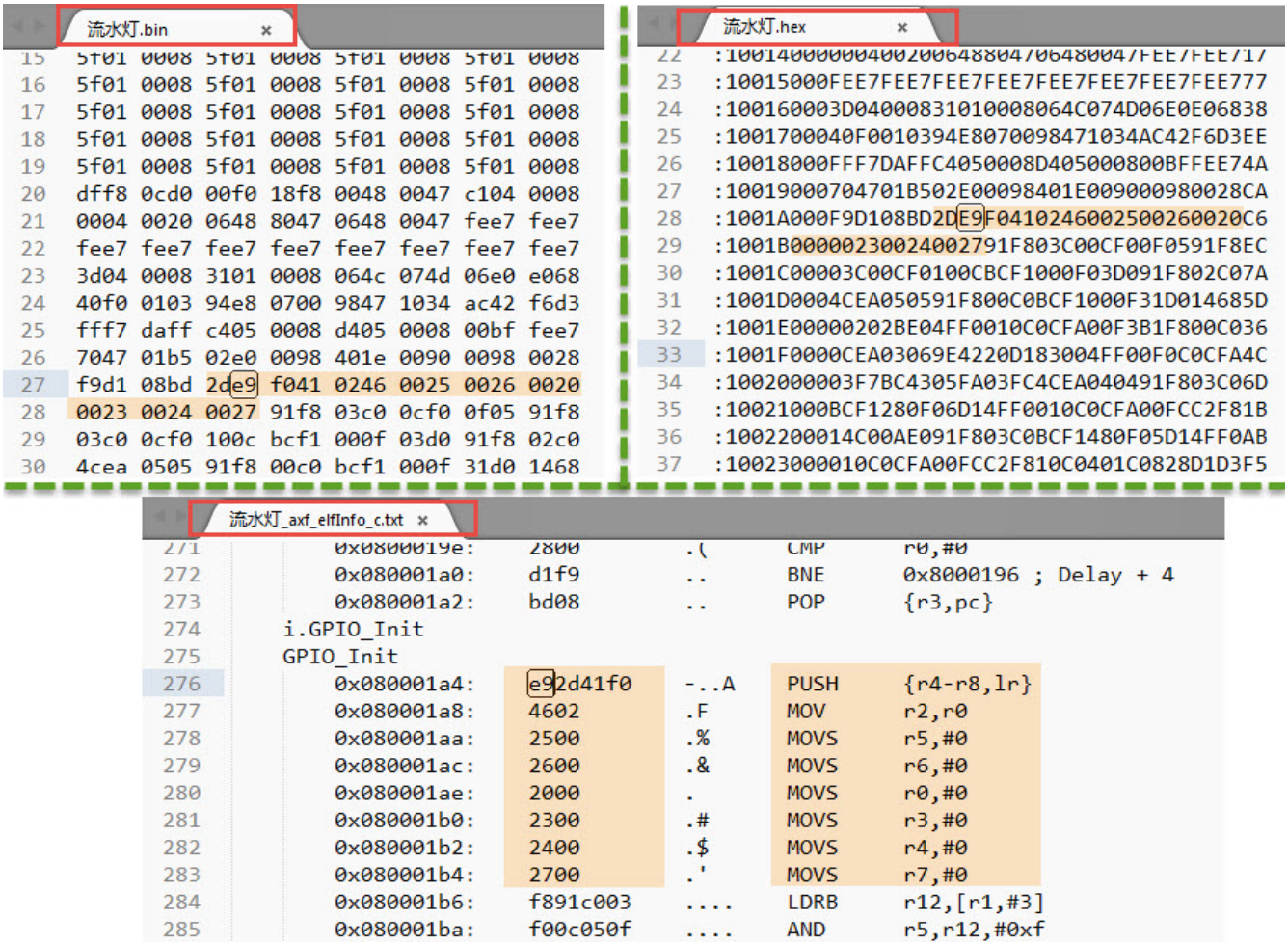


图 40-43 GPIO_Init 函数的代码数据在三个文件中的表示

如果芯片没有做任何加密措施，使用下载器可以直接从芯片读回它存储在 FLASH 中的数据，从而得到 bin 映像文件，根据芯片型号还原出部分代码即可进行修改，甚至不用修改代码，直接根据目标产品的硬件 PCB，抄出一样的板子，再把 bin 映像下载芯片，直接山寨出目标产品，所以在实际的生产中，一定要注意做好加密措施。由于 axf 文件中含有大量的信息，且直接使用 fromelf 即可反汇编代码，所以更不要随便泄露 axf 文件。lib 文件也能反使用 fromelf 文件反汇编代码，不过它不能还原出 C 代码，由于 lib 文件的主要目的是为了保护 C 源代码，也算是达到了它的要求。

39.4.3.6 htm 静态调用图文件

在 Output 目录下，有一个以工程文件命名的后缀为 *.bulid_log.htm 及 *.htm 文件，如“流水灯.bulid_log.htm”及“流水灯.htm”，它们都可以使用浏览器打开。其中 *.build_log.htm 是工程的构建过程日志，而 *.htm 是链接器生成的静态调用图文件。

在静态调用图文件中包含了整个工程各种函数之间互相调用的关系图，而且它还给出了静态占用最深的栈空间数量以及它对应的调用关系链。

例如图 *40_44* 是“流水灯.htm”文件顶部的说明。

Static Call Graph for image ..\..\Output\流水灯.axf

```
#<CALLGRAPH># ARM Linker, 5050169: Last Updated: Wed Oct 19 11:55:41 2016

Maximum Stack Usage = 32 bytes + Unknown(Cycles, Untraceable Function Pointers)

Call chain for Maximum Stack Depth:

main ⇒ LED_GPIO_Config ⇒ GPIO_Init
```

图 40-44 “流水灯.htm”中的静态占用最深的栈空间说明

该文件说明了本工程的静态栈空间最大占用 32 字节 (Maximum Stack Usage:32bytes)，这个占用最深的静态调用为“main->LED_GPIO_Config->GPIO_Init”。注意这里给出的空间只是静态的栈使用统计，链接器无法统计动态使用情况，例如链接器无法知道递归函数的递归深度。在本文件的后面还可查询到其它函数的调用情况及其它细节。

利用这些信息，我们可以大致了解工程中应该分配多少空间给栈，有空间余量的情况下，一般会设置比这个静态最深栈使用量大一倍，在 STM32 中可修改启动文件改变堆栈的大小；如果空间不足，可从该文件中了解到调用深度的信息，然后优化该代码。

注意：

查看了各个工程的静态调用图文件统计后，我们发现本书提供的一些比较大规模的工程例子，静态栈调用最大深度都已超出 STM32 启动文件默认的栈空间大小 0x00000400，即 1024 字节，但在当时的调试过程中却没有发现错误，所以当时没有修改栈的默认大小 (有一些工程调试时已发现

问题，它们的栈空间就已经被我们改大了)，虽然这些工程实际运行并没有错误，但这可能只是因为它使用的栈溢出 RAM 空间恰好没被程序其它部分修改而已。所以，建议您在实际的大型工程应用中 (特别是使用了各种外部库时，如 Lwip/emWin/Fatfs 等)，要查看本静态调用图文件，了解程序的栈使用情况，给程序分配合适的栈空间。

39.4.4 Listing 目录下的文件

在 Listing 目录下包含了 *.map 及 *.lst 文件，它们都是文本格式的，可使用 Windows 的记事本软件打开。其中 lst 文件仅包含了一些汇编符号的链接信息，我们重点分析 map 文件。

39.4.4.1 map 文件说明

map 文件是由链接器生成的，它主要包含交叉链接信息，查看该文件可以了解工程中各种符号之间的引用以及整个工程的 Code、RO-data、RW-data 以及 ZI-data 的详细及汇总信息。它的内容中主要包含了“节区的跨文件引用”、“删除无用节区”、“符号映像表”、“存储器映像索引”以及“映像组件大小”，各部分介绍如下：

39.4.4.1.1 节区的跨文件引用

打开“流水灯.map”文件，可看到它的第一部分——节区的跨文件引用 (Section Cross References)，见*代码清单 40_10*。

列表 10: 代码清单 40-10 节区的跨文件引用 (部分，流水灯.map 文件)

```
==============================================================

Section Cross References

   startup_stm32f10x_hd.o(RESET) refers to startup_stm32f10x_hd.o(STACK)␣
↪for __initial_sp
```

(下页继续)

(续上页)

```
   startup_stm32f10x_hd.o(RESET) refers to startup_stm32f10x_hd.o(.text)␣
↪for Reset_Handler
   startup_stm32f10x_hd.o(RESET) refers to stm32f1xx_it.o(i.SysTick_
↪Handler) for SysTick_Handler
   /**... 以下部分省略 ****/
   main.o(i.main) refers to bsp_led.o(i.LED_GPIO_Config) for LED_GPIO_Config
   main.o(i.main) refers to main.o(i.Delay) for Delay
   bsp_led.o(i.LED_GPIO_Config) refers to stm32f10x_rcc.o(i.RCC_
↪APB2PeriphClockCmd) for
   RCC_APB2PeriphClockCmd
   bsp_led.o(i.LED_GPIO_Config) refers to stm32f10x_gpio.o(i.GPIO_Init) for␣
↪GPIO_Init
   bsp_led.o(i.LED_GPIO_Config) refers to stm32f10x_gpio.o(i.GPIO_SetBits)␣
↪for GPIO_SetBits
   /**... 以下部分省略 ****/
```

在这部分中，详细列出了各个 *.o 文件之间的符号引用。由于 *.o 文件是由 asm 或 c/c++ 源文件编译后生成的，各个文件及文件内的节区间互相独立，链接器根据它们之间的互相引用链接起来，链接的详细信息在这个“Section Cross References”一一列出。

例如，开头部分说明的是 startup_stm32f10x.o 文件中的“RESET”节区分为它使用的“__initial_sp”符号引用了同文件“STACK”节区。

也许我们对启动文件不熟悉，不清楚这究竟是什么，那我们继续浏览，可看到 main.o 文件的引用说明，如说明 main.o 文件的 i.main 节区为它使用的 LED_GPIO_Config 符号引用了 bsp_led.o 文件的 i.LED_GPIO_Config 节区。

同样地，下面还有 bsp_led.o 文件的引用说明，如说明了 bsp_led.o 文件的 i.LED_GPIO_Config 节区为它使用的 GPIO_Init 符号引用了 stm32f10x_gpio.o 文件的 i.GPIO_Init 节区。

可以了解到，这些跨文件引用的符号其实就是源文件中的函数名、变量名。有时在构建工程的时候，编译器会输出“Undefined symbol xxx (referred from xxx.o)”这样的提示，该提示的原因就是在链接过程中，某个文件无法在外部找到它引用的标号，因而产生链接错误。例如，见图 *40_45*，我们把 bsp_led.c 文件中定义的函数 LED_GPIO_Config 改名为 LED_GPIO_ConfigABCD，而不修

改 main.c 文件中的调用，就会出现 main 文件无法找到 LED_GPIO_Config 符号的提示（Undefined symbol xxxx from xxx.o）。

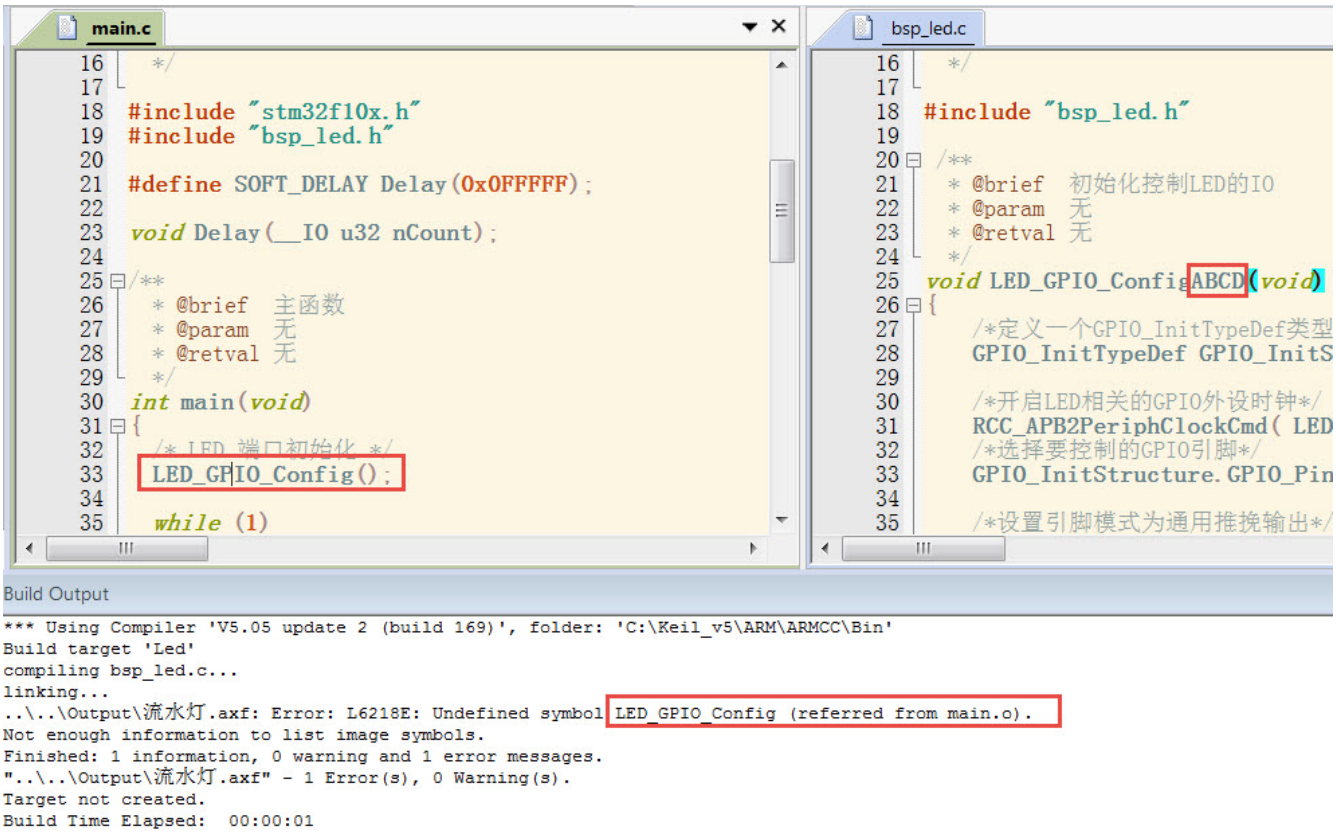


图 40-45 找不到符号的错误提示

39.4.4.1.2 删除无用节区

map 文件的第二部分是删除无用节区的说明 (Removing Unused input sections from the image.)，见*代码清单 40_11* 。

列表 11: 代码清单 40-11 删除无用节区 (部分，流水灯.map 文件)

```
==============================================================================
```

(下页继续)

(续上页)

```
Removing Unused input sections from the image.

   Removing startup_stm32f10x_hd.o(HEAP), (512 bytes).
   Removing core_cm3.o(.emb_text), (32 bytes).
   Removing system_stm32f10x.o(i.SystemCoreClockUpdate), (164 bytes).
   Removing system_stm32f10x.o(.data), (20 bytes).
   Removing misc.o(i.NVIC_Init), (112 bytes).
   Removing misc.o(i.NVIC_PriorityGroupConfig), (20 bytes).
   Removing misc.o(i.NVIC_SetVectorTable), (20 bytes).
   Removing misc.o(i.NVIC_SystemLPConfig), (32 bytes).
   Removing misc.o(i.SysTick_CLKSourceConfig), (40 bytes).
   Removing stm32f10x_adc.o(i.ADC_AnalogWatchdogCmd), (20 bytes).
   Removing stm32f10x_adc.o(i.ADC_AnalogWatchdogSingleChannelConfig), (16␣
↪bytes).
   Removing stm32f10x_adc.o(i.ADC_AnalogWatchdogThresholdsConfig), (6␣
↪bytes).
   Removing stm32f10x_adc.o(i.ADC_AutoInjectedConvCmd), (22 bytes).
   Removing stm32f10x_adc.o(i.ADC_ClearFlag), (6 bytes).
   Removing stm32f10x_adc.o(i.ADC_ClearITPendingBit), (10 bytes).
   Removing stm32f10x_adc.o(i.ADC_Cmd), (22 bytes).
   Removing stm32f10x_adc.o(i.ADC_DMACmd), (22 bytes).
   Removing stm32f10x_adc.o(i.ADC_DeInit), (92 bytes).
   Removing stm32f10x_adc.o(i.ADC_DiscModeChannelCountConfig), (24 bytes).
   /*…以下部分省略 */
```

这部分列出了在链接过程它发现工程中未被引用的节区，这些未被引用的节区将会被删除 (指不加入到 *.axf 文件，不是指在 *.o 文件删除)，这样可以防止这些无用数据占用程序空间。

例如，上面的信息中说明 startup_stm32f10x.o 中的 HEAP(在启动文件中定义的用于动态分配的“堆”区) 以及 stm32f10x_adc.o 的各个节区都被删除了，因为在我们这个工程中没有使用动态内存分配，也没有引用任何 stm32f10x_adc.c 中的内容。由此也可以知道，虽然我们把 STM32HAL 库的各个外设对应的 c 库文件都添加到了工程，但不必担心这会使工程变得臃肿，因为未被引用的节区内容不会被加入到最终的机器码文件中。

39.4.4.1.3 符号映像表

map 文件的第三部分是符号映像表 (Image Symbol Table)，见*代码清单 40_12*。

列表 12: 代码清单 40-12 符号映像表 (部分，流水灯.map 文件)

```
Image Symbol Table

   Local Symbols

   Symbol Name                     Value     Ov Type        Size ␣
↪Object(Section)
   /**... 省略部分 ****/
   ../clib/microlib/init/entry.s   0x00000000   Number         0  entry7b.o␣
↪ABSOLUTE
   ../clib/microlib/init/entry.s   0x00000000   Number         0  entry11b.
↪o ABSOLUTE
   ../clib/microlib/init/entry.s   0x00000000   Number         0  entry11a.
↪o ABSOLUTE
   ../clib/microlib/init/entry.s   0x00000000   Number         0  entry10b.
↪o ABSOLUTE

   i.DebugMon_Handler              0x08000190   Section        0  stm32f1xx_
↪it.o(i.DebugMon_Handler)
   i.Delay                         0x08000192   Section        0  main.o(i.
↪Delay)
   i.GPIO_Init                     0x080001a4   Section        0  stm32f10x_
↪gpio.o(i.GPIO_Init)
   i.GPIO_SetBits                  0x080002ba   Section        0  stm32f10x_
↪gpio.o(i.GPIO_SetBits)
   i.HardFault_Handler             0x080002be   Section        0  stm32f1xx_
↪it.o(i.HardFault_Handler)
```

(下页继续)

(续上页)

```
   i.LED_GPIO_Config                0x080002c4   Section        0  bsp_led.
↪o(i.LED_GPIO_Config)
   i.RCC_APB2PeriphClockCmd    0x0800032c   Section        0  stm32f10x_rcc.
↪o(i.RCC_APB2PeriphClockCmd)
   i.main                           0x080004c0   Section        0  main.o(i.
↪main)
   STACK                            0x20000000   Section     1024  startup_
↪stm32f10x_hd.o(STACK)

   Global Symbols

   Symbol Name                              Value     Ov Type         Size ␣
↪Object(Section)
   /**... 省略部分 ****/
LED_GPIO_Config                  0x080002c5   Thumb Code    90  bsp_led.o(i.
↪LED_GPIO_Config)
   RCC_APB2PeriphClockCmd      0x0800032d   Thumb Code    26  stm32f10x_rcc.
↪o(i.RCC_APB2PeriphClockCmd)
   SVC_Handler                      0x0800034d   Thumb Code     2  stm32f1xx_
↪it.o(i.SVC_Handler)
   SysTick_Handler                  0x08000439   Thumb Code     2  stm32f1xx_
↪it.o(i.SysTick_Handler)
   SystemInit                       0x0800043d   Thumb Code    78  system_
↪stm32f10x.o(i.SystemInit)
   main                             0x080004c1   Thumb Code   252  main.o(i.
↪main)
   Region$$Table$$Base              0x080005c4   Number         0  anon$$obj.
↪o(Region$$Table)
   Region$$Table$$Limit             0x080005d4   Number         0  anon$$obj.
↪o(Region$$Table)
   __initial_sp                     0x20000400   Data           0  startup_
↪stm32f10x_hd.o(STACK)
```

(下页继续)

(续上页)

```
   /**... 以下部分省略 ****/
```

这个表列出了被引用的各个符号在存储器中的具体地址、占据的空间大小等信息。如我们可以查到 LED_GPIO_Config 符号存储在 0x080002c4 地址，它属于 Thumb Code 类型，大小为 90 字节，它所在的节区为 bsp_led.o 文件的 i.LED_GPIO_Config 节区。

39.4.4.1.4 存储器映像索引

map 文件的第四部分是存储器映像索引 (Memory Map of the image)，见*代码清单 40_13*。

列表 13: 代码清单 40-13 存储器映像索引 (部分，流水灯.map 文件)

```
Memory Map of the image

   Image Entry point : 0x08000131

   Load Region LR_IROM1 (Base: 0x08000000, Size: 0x000005d4, Max:␣
↪0x00080000, ABSOLUTE)

   Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x000005d4, Max:␣
↪0x00080000, ABSOLUTE)

   Base Addr    Size         Type   Attr      Idx    E Section Name       ␣
↪Object


   0x08000190   0x00000002   Code   RO         3130    i.DebugMon_Handler ␣
↪stm32f1xx_it.o
   0x08000192   0x00000012   Code   RO         3108    i.Delay            ␣
↪main.o
```

(下页继续)

(续上页)

```
    0x080001a4   0x00000116   Code   RO          1292    i.GPIO_Init          ␣
↪stm32f10x_gpio.o
    0x080002ba   0x00000004   Code   RO          1300    i.GPIO_SetBits       ␣
↪stm32f10x_gpio.o
    0x080002be   0x00000004   Code   RO          3131    i.HardFault_Handler ␣
↪stm32f1xx_it.o
    0x080002c2   0x00000002   PAD
    0x080002c4   0x00000060   Code   RO          3192    i.LED_GPIO_Config    ␣
↪bsp_led.o
    0x08000324   0x00000004   Code   RO          3132    i.MemManage_Handler ␣
↪stm32f1xx_it.o
    0x08000328   0x00000002   Code   RO          3133    i.NMI_Handler        ␣
↪stm32f1xx_it.o
    0x0800032a   0x00000002   Code   RO          3134    i.PendSV_Handler     ␣
↪stm32f1xx_it.o
    0x0800032c   0x00000020   Code   RO          1710    i.RCC_
↪APB2PeriphClockCmd  stm32f10x_rcc.o

    0x080004be   0x00000002   PAD
    0x080004c0   0x00000104   Code   RO          3109    i.main               ␣
↪main.o
    0x080005c4   0x00000010   Data   RO          3226    Region$$Table        ␣
↪anon$$obj.o


    Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x00000400, Max:␣
↪0x00010000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        ␣
↪Object

    0x20000000   0x00000400   Zero   RW           1    STACK                ␣
↪startup_stm32f10x_hd.o
```

(下页继续)

(续上页)

本工程的存储器映像索引分为 ER_IROM1 及 RW_IRAM1 部分，它们分别对应 STM32 内部 FLASH 及 SRAM 的空间。相对于符号映像表，这个索引表描述的单位是节区，而且它描述的主要信息中包含了节区的类型及属性，由此可以区分 Code、RO-data、RW-data 及 ZI-data。

例如，从上面的表中我们可以看到 i.LED_GPIO_Config 节区存储在内部 FLASH 的 0x080002c4 地址，大小为 0x00000060，类型为 Code，属性为 RO。而程序的 STACK 节区 (栈空间) 存储在 SRAM 的 0x20000000 地址，大小为 0x00000400，类型为 Zero，属性为 RW（即 RW-data）。

39.4.4.1.5 映像组件大小

map 文件的最后一部分是包含映像组件大小的信息 (Image component sizes)，这也是最常查询的内容，见*代码清单 40_14*。

列表 14: 代码清单 40-14 映像组件大小 (部分，多彩流水灯.map 文件)

```
Image component sizes

      Code (inc. data)    RO Data    RW Data    ZI Data      Debug   Object␣
↪Name

      96           6           0          0          0        622   bsp_led.o
         0           0           0          0          0       4504   core_
↪cm3.o
      278          8           0          0          0       1675   main.o
      36           8         304          0       1024        932   startup_
↪stm32f10x_hd.o
      282          0           0          0          0       2771   ␣
↪stm32f10x_gpio.o
      26           0           0          0          0       4726   stm32f1xx_
↪it.o
```

(下页继续)

(续上页)

```
      32           6           0           0           0         665   stm32f10x_
↪rcc.o
      328          28           0           0           0      214041   system_
↪stm32f10x.o
   ----------------------------------------------------------------------
      1084          56         320           0        1024      229936   Object␣
↪Totals
         0           0          16           0           0           0   (incl.␣
↪Generated)
         6           0           0           0           0           0   (incl.␣
↪Padding)
   ----------------------------------------------------------------------
/*... 省略部分 */
==============================================================================
      Code (inc. data)    RO Data    RW Data    ZI Data      Debug

      1172          72         320           0        1024      229428   Grand␣
↪Totals
      1172          72         320           0        1024      229428   ELF␣
↪Image Totals
      1172          72         320           0           0           0   ROM␣
↪Totals
==============================================================================
   Total RO  Size (Code + RO Data)                 1492 (   1.46kB)
   Total RW  Size (RW Data + ZI Data)              1024 (   1.00kB)
   Total ROM Size (Code + RO Data + RW Data)       1492 (   1.46kB)
==============================================================
```

这部分包含了各个使用到的 *.o 文件的空间汇总信息、整个工程的空间汇总信息以及占用不同类型存储器的空间汇总信息，它们分类描述了具体占据的 Code、RO-data、RW-data 及 ZI-data 的大小，并根据这些大小统计出占据的 ROM 总空间。

我们仅分析最后两部分信息，如 Grand Totals 一项，它表示整个代码占据的所有空间信息，其中

Code 类型的数据大小为 1172 字节，这部分包含了 72 字节的指令数据 (inc .data) 已算在内，另外 RO-data 占 320 字节，RW-data 占 0 字节，ZI-data 占 1024 字节。在它的下面两行有一项 ROM Totals 信息，它列出了各个段所占据的 ROM 空间，除了 ZI-data 不占 ROM 空间外，其余项都与 Grand Totals 中相等 (RW-data 也占据 ROM 空间，只是本工程中没有 RW-data 类型的数据而已)。

最后一部分列出了只读数据 (RO)、可读写数据 (RW) 及占据的 ROM 大小。其中只读数据大小为 1492 字节，它包含 Code 段及 RO-data 段; 可读写数据大小为 1024 字节，它包含 RW-data 及 ZI-data 段；占据的 ROM 大小为 1492 字节，它除了 Code 段和 RO-data 段，还包含了运行时需要从 ROM 加载到 RAM 的 RW-data 数据（本工程中 RW-data 数据为 0 字节）。

综合整个 map 文件的信息，可以分析出，当程序下载到 STM32 的内部 FLASH 时，需要使用的内部 FLASH 是从 0x0800 0000 地址开始的大小为 1492 字节的空间；当程序运行时，需要使用的内部 SRAM 是从 0x20000000 地址开始的大小为 1024 字节的空间。

粗略一看，发现这个小程序竟然需要 1024 字节的 SRAM，实在说不过去，但仔细分析 map 文件后，可了解到这 1024 字节都是 STACK 节区的空间 (即栈空间)，栈空间大小是在启动文件中定义的，这 1024 字节是默认值 (0x00000400)。它是提供给 C 语言程序局部变量申请使用的空间，若我们确认自己的应用程序不需要这么大的栈，完全可以修改启动文件，把它改小一点，查看前面讲解的 htm 静态调用图文件可了解静态的栈调用情况，可以用它作为参考。

39.4.5 sct 分散加载文件的格式与应用

39.4.5.1 sct 分散加载文件简介

当工程按默认配置构建时，MDK 会根据我们选择的芯片型号，获知芯片的内部 FLASH 及内部 SRAM 存储器概况，生成一个以工程名命名的后缀为 *.sct 的分散加载文件 (Linker Control File，scatter loading)，链接器根据该文件的配置分配各个节区地址，生成分散加载代码，因此我们通过修改该文件可以定制具体节区的存储位置。

例如可以设置源文件中定义的所有变量自动按地址分配到外部 SRAM，这样就不需要再使用关键字 “__attribute__” 按具体地址来指定了；利用它还可以控制代码的加载区与执行区的位置，例如可以把程序代码存储到单位容量价格便宜的 NAND-FLASH 中，但在 NAND-FLASH 中的代码是不能像内部 FLASH 的代码那样直接提供给内核运行的，这时可通过修改分散加载文件，把代码加载区设定为 NAND-FLASH 的程序位置，而程序的执行区设定为 SRAM 中的位置，这样链接

器就会生成一个配套的分散加载代码，该代码会把 NAND-FLASH 中的代码加载到 SRAM 中，内核再从 SRAM 中运行主体代码，大部分运行 Linux 系统的代码都是这样加载的。

39.4.5.2 分散加载文件的格式

下面先来看看 MDK 默认使用的 sct 文件，在 Output 目录下可找到“流水灯.sct”，该文件记录的内容见*代码清单 40_15*。

列表 15: 代码清单 40-15 默认的分散加载文件内容 (“流水灯.sct”)

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00100000  {    ; 注释: 加载域，基地址 空间大小
ER_IROM1 0x08000000 0x00100000  { ; 注释: 加载地址 = 执行地址
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00030000  {  ; 注释: 可读写数据
   .ANY (+RW +ZI)
}
}
```

在默认的 sct 文件配置中仅分配了 Code、RO-data、RW-data 及 ZI-data 这些大区域的地址，链接时各个节区 (函数、变量等) 直接根据属性排列到具体的地址空间。

sct 文件中主要包含描述加载域及执行域的部分，一个文件中可包含有多个加载域，而一个加载域可由多个部分的执行域组成。同等级的域之间使用花括号“{}”分隔开，最外层的是加载域，第二层“{}”内的是执行域，其整体结构见图 *40_46*。

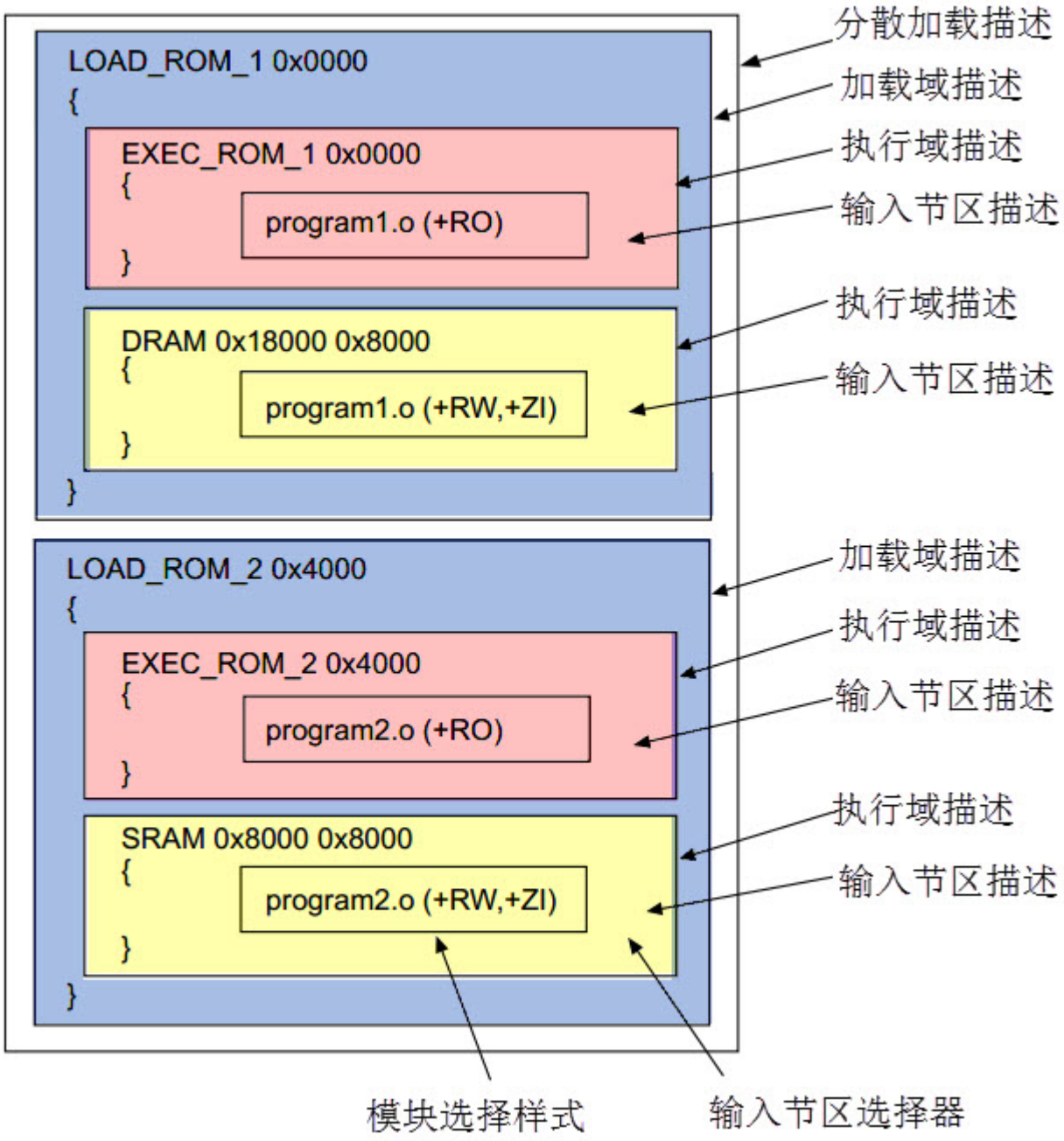


图 40-46 分散加载文件的整体结构

39.4.5.2.1 加载域

sct 文件的加载域格式见*代码清单 40_16*。

列表 16: 代码清单 40-16 加载域格式

```
//方括号中的为选填内容
加载域名 （基地址 | ("+" 地址偏移)）[属性列表] [最大容量]
"{"
    执行区域描述 +
"}"
```

配合前面代码清单 42-15 中的分散加载文件内容，各部分介绍如下：

- 加载域名：名称，在 map 文件中的描述会使用该名称来标识空间。如本例中只有一个加载域，该域名为 LR_IROM1。
- 基地址 + 地址偏移：这部分说明了本加载域的基地址，可以使用 + 号连接一个地址偏移，算进基地址中，整个加载域以它们的结果为基地址。如本例中的加载域基地址为 0x08000000，刚好是 STM32 内部 FLASH 的基地址。
- 属性列表：属性列表说明了加载域的是否为绝对地址、N 字节对齐等属性，该配置是可选的。本例中没有描述加载域的属性。
- 最大容量：最大容量说明了这个加载域可使用的最大空间，该配置也是可选的，如果加上这个配置后，当链接器发现工程要分配到该区域的空间比容量还大，它会在工程构建过程给出提示。本例中的加载域最大容量为 0x00080000，即 512KB，正是本型号 STM32 内部 FLASH 的空间大小。

39.4.5.2.2 执行域

sct 文件的执行域格式见*代码清单 40_17*。

列表 17: 代码清单 40-17 执行域格式

```
//方括号中的为选填内容
执行域名 （基地址 | "+" 地址偏移）［属性列表］［最大容量 ］
"{"
    输入节区描述
"}"
```

执行域的格式与加载域是类似的，区别只是输入节区的描述有所不同，在代码清单 *40_15* 的例子中包含了 ER_IROM1 及 RW_IRAM 两个执行域，它们分别对应描述了 STM32 的内部 FLASH 及内部 SRAM 的基地址及空间大小。而它们内部的“输入节区描述”说明了哪些节区要存储到这些空间，链接器会根据它来处理编排这些节区。

39.4.5.2.3 输入节区描述

配合加载域及执行域的配置，在相应的域配置“输入节区描述”即可控制该节区存储到域中，其格式见代码清单 *40_18*。

列表 18: 代码清单 40-18 输入节区描述的几种格式

```
//除模块选择样式部分外，其余部分都可选选填
模块选择样式"(" 输入节区样式","+" 输入节区属性")"
模块选择样式"(" 输入节区样式","+" 节区特性")"

模块选择样式"(" 输入符号样式","+" 节区特性")"
模块选择样式"(" 输入符号样式","+" 输入节区属性")"
```

配合前面代码清单 *40_15* 中的分散加载文件内容，各部分介绍如下：

- 模块选择样式：模块选择样式可用于选择 o 及 lib 目标文件作为输入节区，它可以直接使用目标文件名或“*”通配符，也可以使用“.ANY”。例如，使用语句“bsp_led.o”可以选择 bsp_led.o 文件，使用语句“*.o*”可以选择所有 *o* 文件，使用“.lib”可以选择所有 lib 文件，使用“*”或“.ANY”可以选择所有的 o 文件及 lib 文件。其中“.ANY”选择语句的优先级

是最低的，所有其它选择语句选择完剩下的数据才会被“.ANY”语句选中。

- 输入节区样式：我们知道在目标文件中会包含多个节区或符号，通过输入节区样式可以选择要控制的节区。

  示例文件中“(RESET, +First)”语句的 RESET 就是输入节区样式，它选择了名为 RESET 的节区，并使用后面介绍的节区特性控制字“+First”表示它要存储到本区域的第一个地址。示例文件中的“*(InRoot$$Sections)*”是一个链接器支持的特殊选择符号，它可以选择所有 *HAL* 库里要求存储到 *root* 区域的节区，如 *__main.o*、*__scatter*.o 等内容。

- 输入符号样式：同样地，使用输入符号样式可以选择要控制的符号，符号样式需要使用“:gdef:”来修饰。例如可以使用“*(:gdef:Value_Test)”来控制选择符号“Value_Test”。

- 输入节区属性：通过在模块选择样式后面加入输入节区属性，可以选择样式中不同的内容，每个节区属性描述符前要写一个“+”号，使用空格或“,”号分隔开，可以使用的节区属性描述符见表 40-6。

  表 40-6 属性描述符及其意义

节区属性描述符	说明
RO-CODE 及 CODE	只读代码段
RO-DATA 及 CONST	只读数据段
RO 及 TEXT	包括 RO-CODE 及 RO-DATA
RW-DATA	可读写数据段
RW-CODE	可读写代码段
RW 及 DATA	包括 RW-DATA 及 RW-CODE
ZI 及 BSS	初始化为 0 的可读写数据段
XO	只可执行的区域
ENTRY	节区的入口点

  例如，示例文件中使用“.ANY(+RO)”选择剩余所有节区 RO 属性的内容都分配到执行域 ER_IROM1 中，使用“.ANY(+RW +ZI)”选择剩余所有节区 RW 及 ZI 属性的内容都分配到执行域 RW_IRAM1 中。

- 节区特性：节区特性可以使用“+FIRST”或“+LAST”选项配置它要存储到的位置，FIRST

存储到区域的头部，LAST 存储到尾部。通常重要的节区会放在头部，而 CheckSum(校验和)之类的数据会放在尾部。

例如示例文件中使用“(RESET,+First)”选择了 RESET 节区，并要求把它放置到本区域第一个位置，而 RESET 是工程启动代码中定义的向量表，见*代码清单 40_19*，该向量表中定义的堆栈顶和复位向量指针必须要存储在内部 FLASH 的前两个地址，这样 STM32 才能正常启动，所以必须使用 FIRST 控制它们存储到首地址。

列表 19: 代码清单 40-19 startup_stm32f10x.s 文件中定义的 RESET 区 (部分)

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors     DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
```

总的来说，我们的 sct 示例文件配置如下：程序的加载域为内部 FLASH 的 0x08000000，最大空间为 0x00080000；程序的执行基地址与加载基地址相同，其中 RESET 节区定义的向量表要存储在内部 FLASH 的首地址，且所有 o 文件及 lib 文件的 RO 属性内容都存储在内部 FLASH 中；程序执行时 RW 及 ZI 区域都存储在以 0x20000000 为基地址，大小为 0x00010000 的空间 (64KB)，这部分正好是 STM32 内部主 SRAM 的大小。

链接器根据 sct 文件链接，链接后各个节区、符号的具体地址信息可以在 map 文件中查看。

39.4.5.3 通过 MDK 配置选项来修改 sct 文件

了解 sct 文件的格式后，可以手动编辑该文件控制整个工程的分散加载配置，但 sct 文件格式比较复杂，所以 MDK 提供了相应的配置选项可以方便地修改该文件，这些选项配置能满足基本的使用需求，本小节将对这些选项进行说明。

39.4.5.3.1 选择 sct 文件的产生方式

首先需要选择 sct 文件产生的方式，选择使用 MDK 生成还是使用用户自定义的 sct 文件。在 MDK 的“Options for Target->Linker->Use Memory Layout from Target Dialog”选项即可配置该选择，见图 *40_47*。

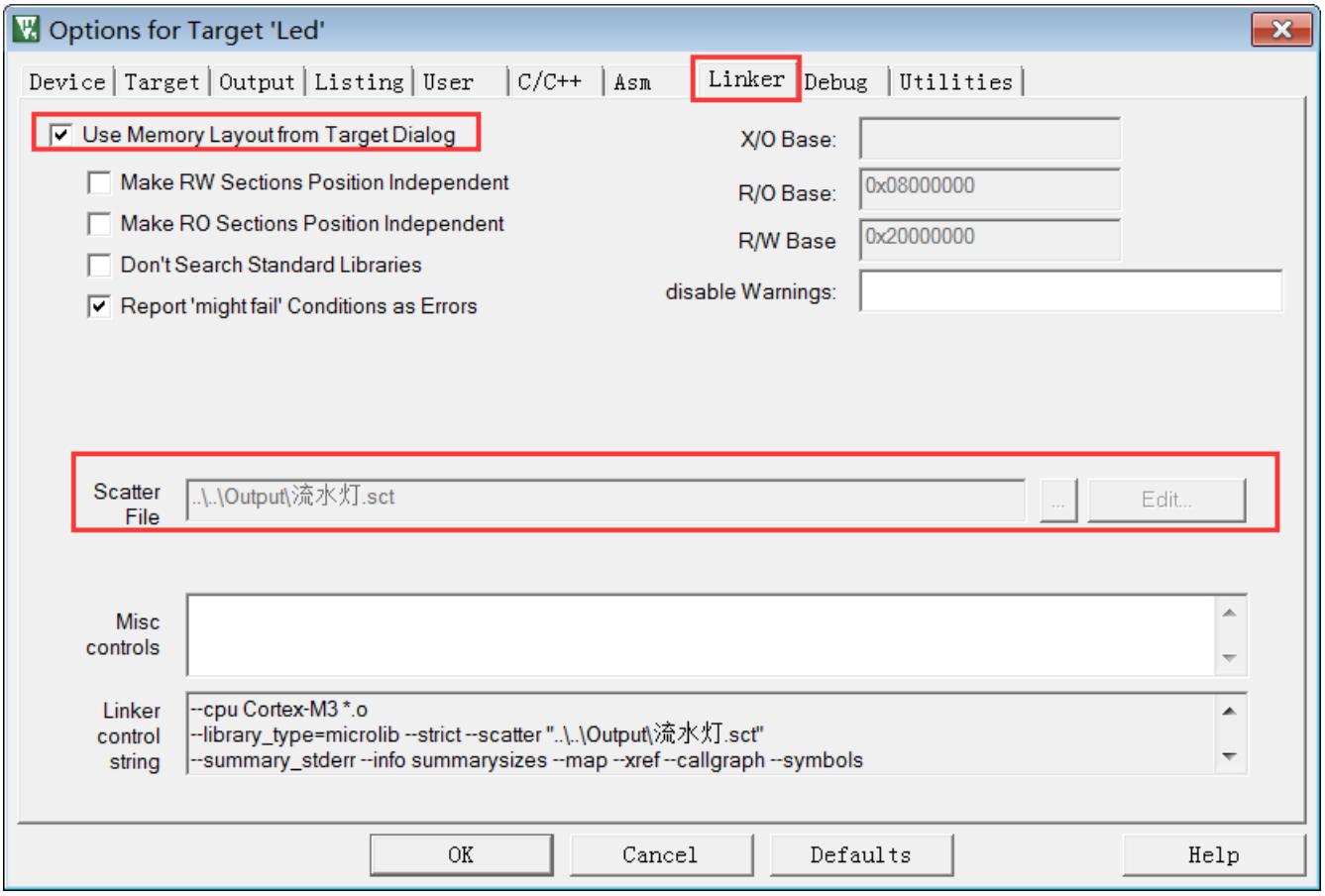


图 40-47 选择使用 MDK 生成的 sct 文件

该选项的译文为“是否使用 Target 对话框中的存储器分布配置”，勾选后，它会根据“Options for Target”对话框中的选项生成 sct 文件，这种情况下，即使我们手动打开它生成的 sct 文件编辑也是无效的，因为每次构建工程的时候，MDK 都会生成新的 sct 文件覆盖旧文件。该选项在 MDK 中是默认勾选的，若希望 MDK 使用我们手动编辑的 sct 文件构建工程，需要取消勾选，并通过 Scatter File 框中指定 sct 文件的路径，见图 *40_48*。

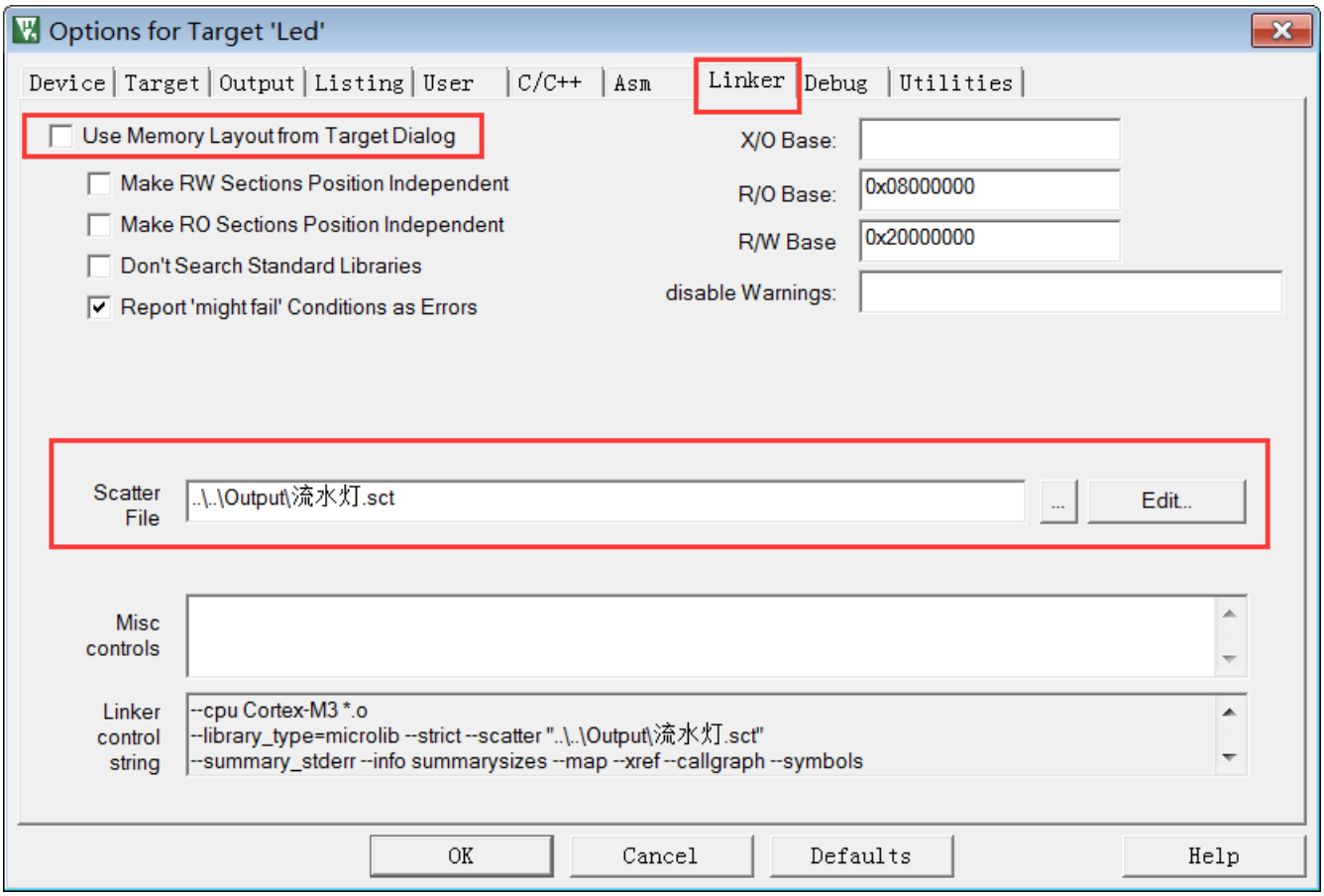


图 40-48 使用指定的 sct 文件构建工程

39.4.5.3.2 通过 Target 对话框控制存储器分配

若我们在 Linker 中勾选了“使用 Target 对话框的存储器布局”选项，那么“Options for Target”对话框中的存储器配置就生效了。主要配置是在 Device 标签页中选择芯片的类型，设定芯片基本的内部存储器信息以及在 Target 标签页中细化具体的存储器配置 (包括外部存储器)，见图 *40_49* 及图 *40_50*。

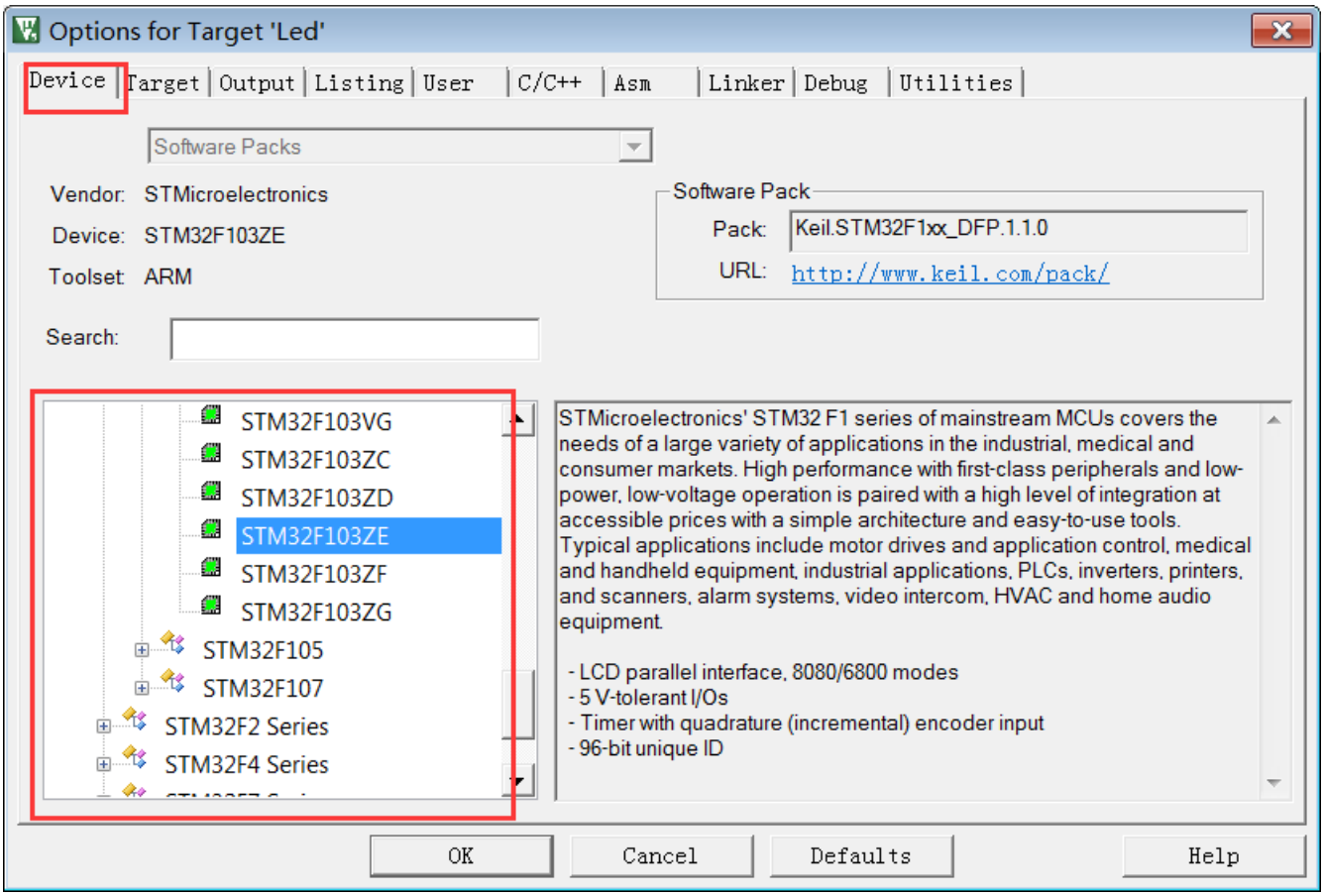


图 40-49 选择芯片类型

图中 Device 标签页中选定了芯片的型号为 STM32F103VE，选中后，在 Target 标签页中的存储器信息会根据芯片更新。

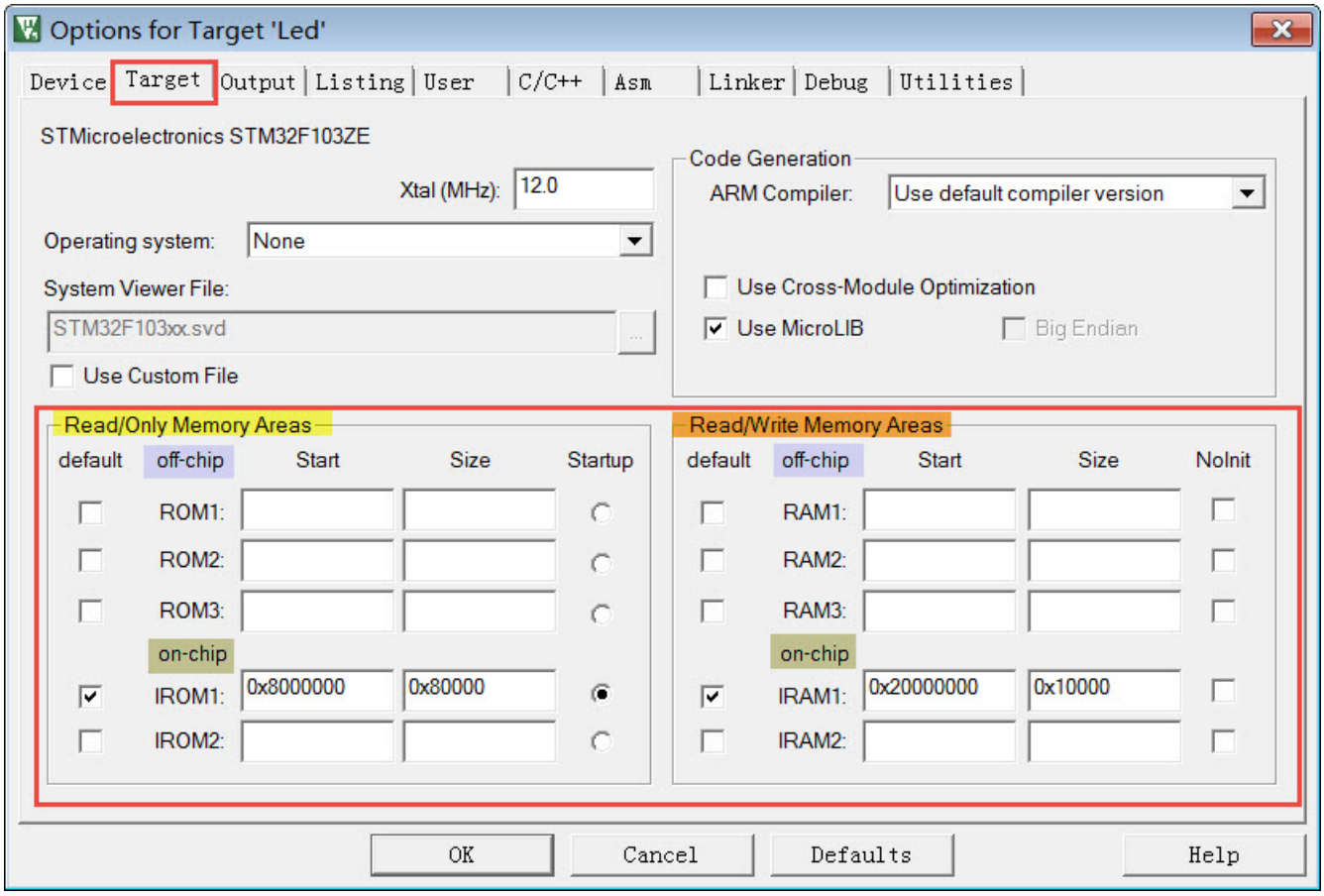


图 40-50 Target 对话框中的存储器分配

在 Target 标签页中存储器信息分成只读存储器 (Read/Only Memory Areas) 和可读写存储器 (Read/Write Memory Areas) 两类，即 ROM 和 RAM，而且它们又细分成了片外存储器 (off-chip) 和片内存储器 (on-chip) 两类。

例如，由于我们已经选定了芯片的型号，MDK 会自动根据芯片型号填充片内的 ROM 及 RAM 信息，其中的 IROM1 起始地址为 0x80000000，大小为 0x80000，正是该 STM32 型号的内部 FLASH 地址及大小；而 IRAM1 起始地址为 0x20000000，大小为 0x10000，正是该 STM32 内部主 SRAM 的地址及大小。图中的 IROM1 及 IRAM1 前面都打上了勾，表示这个配置信息会被采用，若取消勾选，则该存储配置信息是不会被使用的。

在某些芯片，会有多个内部 SRAM 空间，如 STM32F429 系列。它会在标签页中的 IRAM2 一栏默认也填写了配置信息，设置 STM32F4 系列特有的内部高速 SRAM(被称为 CCM)。

而如果希望设置外部 SRAM 空间，可以把外部 SRAM 的信息写到对话框里 “off-chip” 的 “RAM1” 配置中。

下面我们尝试修改 Target 标签页中的这些存储信息，例如，把 STM32 内部的 SRAM 分成两等份，按照图 *40_51* 中的 1 配置，把 IRAM1 的基地址设置为 0x20000000，大小改为 0x8000，把 IRAM2 的基地址设置为 0x20008000，大小为 0x8000，然后编译工程，查看到工程的 sct 文件如代码清单 *40_20* 所示。虽然修改后 IRAM1 和 IRAM2 加起来还是原来的内部 SRAM 空间，但它演示了对 Target 选项的修改是如何影响 sct 文件的，您也可以尝试其它配置，观察 sct 文件，以学习 sct 文件的语法。

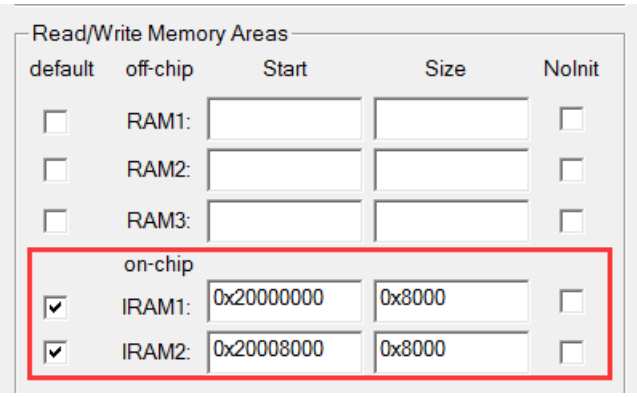


图 40-51 修改 IRAM1 的基地址及仅使用 IRAM2 的配置

列表 20: 代码清单 40-20 修改了 IRAM1 基地址后的 sct 文件内容

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00080000  {    ; load region size_region
ER_IROM1 0x08000000 0x00080000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
```

(下页继续)

(续上页)

```
RW_IRAM1 0x20000000 0x00008000  {  ; RW data
    .ANY (+RW +ZI)
}
RW_IRAM2 0x20008000 0x00008000  {
    .ANY (+RW +ZI)
}
}
```

可以发现，sct 文件根据 Target 标签页做出了相应的改变，除了这种修改外，在 Target 标签页上还控制同时使用 IRAM1 和 IRAM2、加入外部 RAM(如外接的 SRAM)，外部 FLASH 等。

39.4.5.3.3 控制文件分配到指定的存储空间

设定好存储器的信息后，可以控制各个源文件定制到哪个部分存储器，在 MDK 的工程文件栏中，选中要配置的文件，右键，并在弹出的菜单中选择“Options for File xxxx”即可弹出一个文件配置对话框，在该对话框中进行存储器定制，见图 *40_52*。

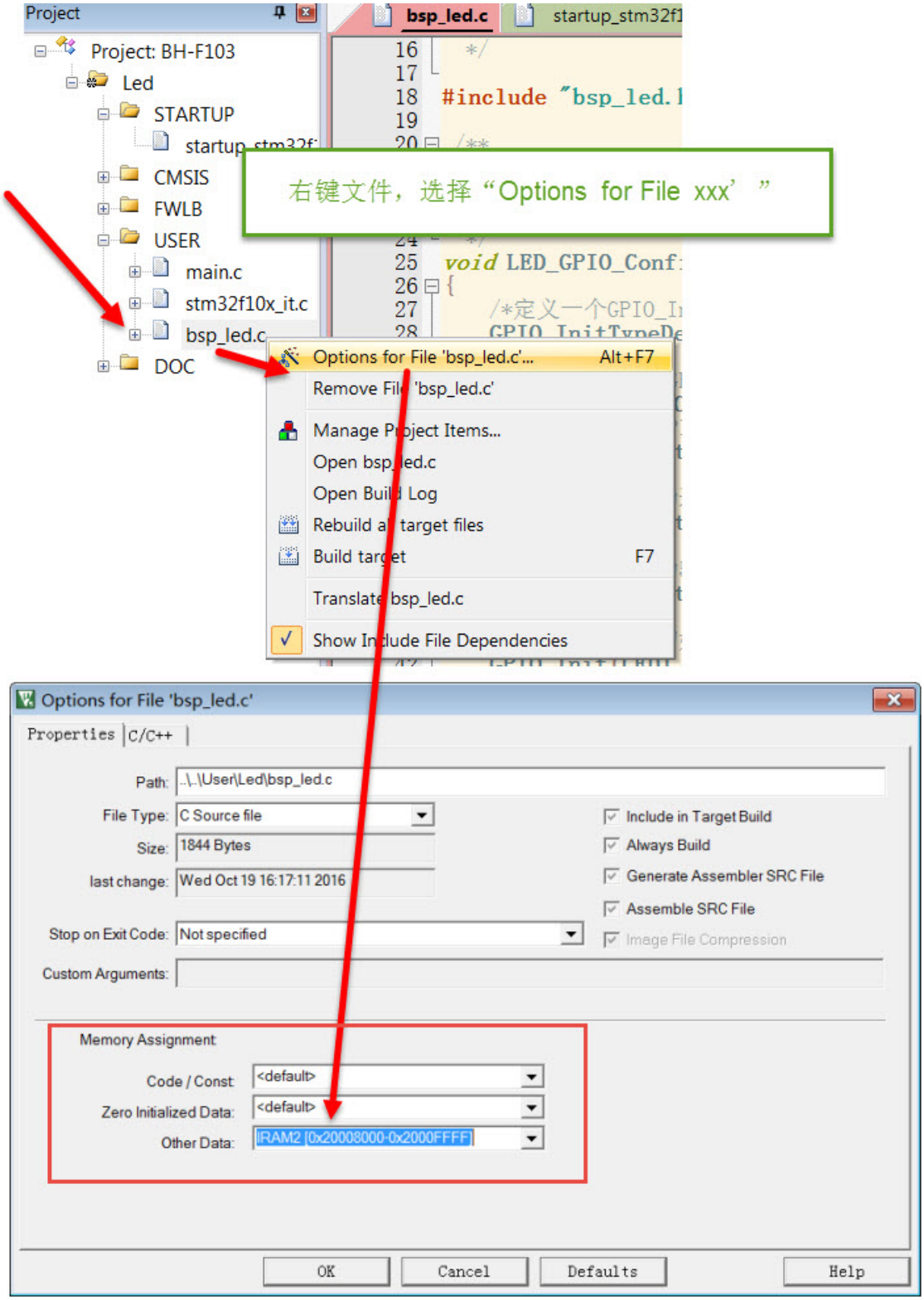

Project
Project: BH-F103
Led
STARTUP
CMSIS
FWLB
USER
main.c
stm32f10x_it.c
bsp_led.c
DOC
bsp_led.c
startup_stm32f1
右键文件，选择“Options for File xxx’ ”
Options for File 'bsp_led.c'... Alt+F7
Remove File 'bsp_led.c'
Manage Project Items...
Open bsp_led.c
Open Build Log
Rebuild all target files
Build target F7
Translate bsp_led.c
Show Include File Dependencies
Options for File 'bsp_led.c'
Properties
C/C++
Path: ..\..\User\Led\bsp_led.c
File Type: C Source file
Size: 1844 Bytes
last change: Wed Oct 19 16:17:11 2016
Include in Target Build
Always Build
Generate Assembler SRC File
Assemble SRC File
Image File Compression
Stop on Exit Code: Not specified
Custom Arguments:
Memory Assignment
Code / Const: <default>
Zero Initialized Data: <default>
Other Data: IRAM2 [0x20008000-0x2000FFFF]
OK
Cancel
Defaults
Help

图 40-52 使用右键打开文件配置并把它的 RW 区配置成使用 IRAM2

在弹出的对话框中有一个 “Memory Assignment” 区域 (存储器分配)，在该区域中可以针对文件的各种属性内容进行分配，如 Code/Const 内容 (RO)、Zero Initialized Data 内容 (ZI-data) 以及 Other Data 内容 (RW-data)，点击下拉菜单可以找到在前面 Target 页面配置的 IROM1、IRAM1、IRAM2 等存储器。例如图中我们把这个 bsp_led.c 文件的 Other Data 属性的内容分配到了 IRAM2 存储器 (在 Target 标签页中我们勾选了 IRAM1 及 IRAM2)，当在 bsp_led.c 文件定义了一些 RW-data 内容时 (如初值非 0 的全局变量)，该变量将会被分配到 IRAM2 空间，配置完成后点击 OK，然后编译工程，查看到的 sct 文件内容见*代码清单 40_21*。

列表 21: 代码清单 40-21 修改 bsp_led.c 配置后的 sct 文件

```
LR_IROM1 0x08000000 0x00080000{    ; load region size_region
ER_IROM1 0x08000000 0x00080000{  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00008000  {  ; RW data
   .ANY (+RW +ZI)
}
RW_IRAM2 0x20008000 0x00008000  {
   bsp_led.o (+RW)
   .ANY (+RW +ZI)
}
}
```

可以看到在 sct 文件中的 RW_IRAM2 执行域中增加了一个选择 bsp_led.o 中 RW 内容的语句。

类似地，我们还可以设置某些文件的代码段被存储到特定的 ROM 中，或者设置某些文件使用的 ZI-data 或 RW-data 存储到外部 SRAM 中 (控制 ZI-data 到 SDRAM 时注意还需要修改启动文件设置堆栈对应的地址，原启动文件中的地址是指向内部 SRAM 的)。

虽然 MDK 的这些存储器配置选项很方便，但有很多高级的配置还是需要手动编写 sct 文件实现

的，例如 MDK 选项中的内部 ROM 选项最多只可以填充两个选项位置，若想把内部 ROM 分成多片地址管理就无法实现了；另外 MDK 配置可控的最小粒度为文件，若想控制特定的节区也需要直接编辑 sct 文件。

接下来我们将讲解几个实验，通过编写 sct 文件定制存储空间。

39.5 实验自动分配变量到指定的 SRAM 空间

最常见的 sct 文件通常应用于管理外部扩展的存储器空间，如野火 F103-霸道开发板扩展的外部 SRAM，野火 F429-挑战者开发板扩展的外部 SDRAM，使用 sct 文件可以轻松分配堆、栈以及各种变量自动分配到这些扩展的空间。由于本书介绍的 F103-指南者开发板使用的 STM32VET6 型号芯片无法扩展外部 SRAM，所以在本教程中使用内部 SRAM 进行演示，在内部 SRAM 划分一块区域进行管理。这些 sct 文件管理方法的核心与扩展外部存储空间的使用情况完全一致，只是管理的存储器地址不同，以及外部存储器需要额外的初始化而已，不会影响我们学习 sct 文件的原理，如果您需要完全针对于外部存储器空间管理的说明，可以查询上述开发板的教程学习。

由于内存管理对应用程序非常重要，若修改 sct 文件，不使用默认配置，对工程影响非常大，容易导致出错，所以我们使用两个实验配置来讲解 sct 文件的应用细节，希望您学习后不仅知其然而且知其所以然，清楚地了解修改后对应用程序的影响，还可以举一反三根据自己的需求进行各种存储器定制。

39.5.1 补充关于“__attribute__”关键字的说明

在程序中，当需要指定某个变量的内存地址时，MDK 提供了一个关键字“__attribute__”实现该功能，这种用法通常也是为了把变量指定到外部扩展的存储器，而 sct 文件存储器管理取代或改进了这种地址分配方式。在此处先补充一下关键字“__attribute__”的使用说明，见代码清单 40_22。

列表 22: 代码清单 40_22 直接指定变量地址

```
/* 定义一个要指定的地址 */
#define USER_ADDR     ((uint32_t)0x20005000)
/* 使用 atribute 指定该变量存储到 USER_ADDR, 这种方式必须定义成全局变量 */
```

(下页继续)

(续上页)

```
uint8_t testValue __attribute__((at(USER_ADDR)));
testValue = 0xDD;
```

这种方式使用“__attribute__((at()))”来指定变量的地址，代码中指定 testValue 存储到 USER_ADDR 地址 0x20005000 中，若把该地址改为外部存储器 SRAM 的地址，变量就会被存储到外部 SRAM 了，因而利用该关键字在一定程度上可以定制各种存储器的空间分配。要注意使用这种方法定义变量时，必须在函数外把它定义成全局变量，才可以存储到指定地址上。

然而，当有多个这样的变量时，为了防止变量占用的空间重叠，或减少碎片空间进行充分利用，就需要手动计算各个变量的地址了，非常麻烦。在本实验中我们将修改 sct 文件，让链接器自动分配全局变量到指定的存储区域并进行管理，使得利用指定存储区域时就跟普通的变量定义一样简单。

上文中提到的“指定存储空间”可以是内部 SRAM，也可以是外部扩展的 SRAM、SDRAM 存储器，它们的管理方法都一样，只是空间地址不同而已，下文将不再强调它们的区别，当您使用不同的存储器时，只要指定不同的地址即可。

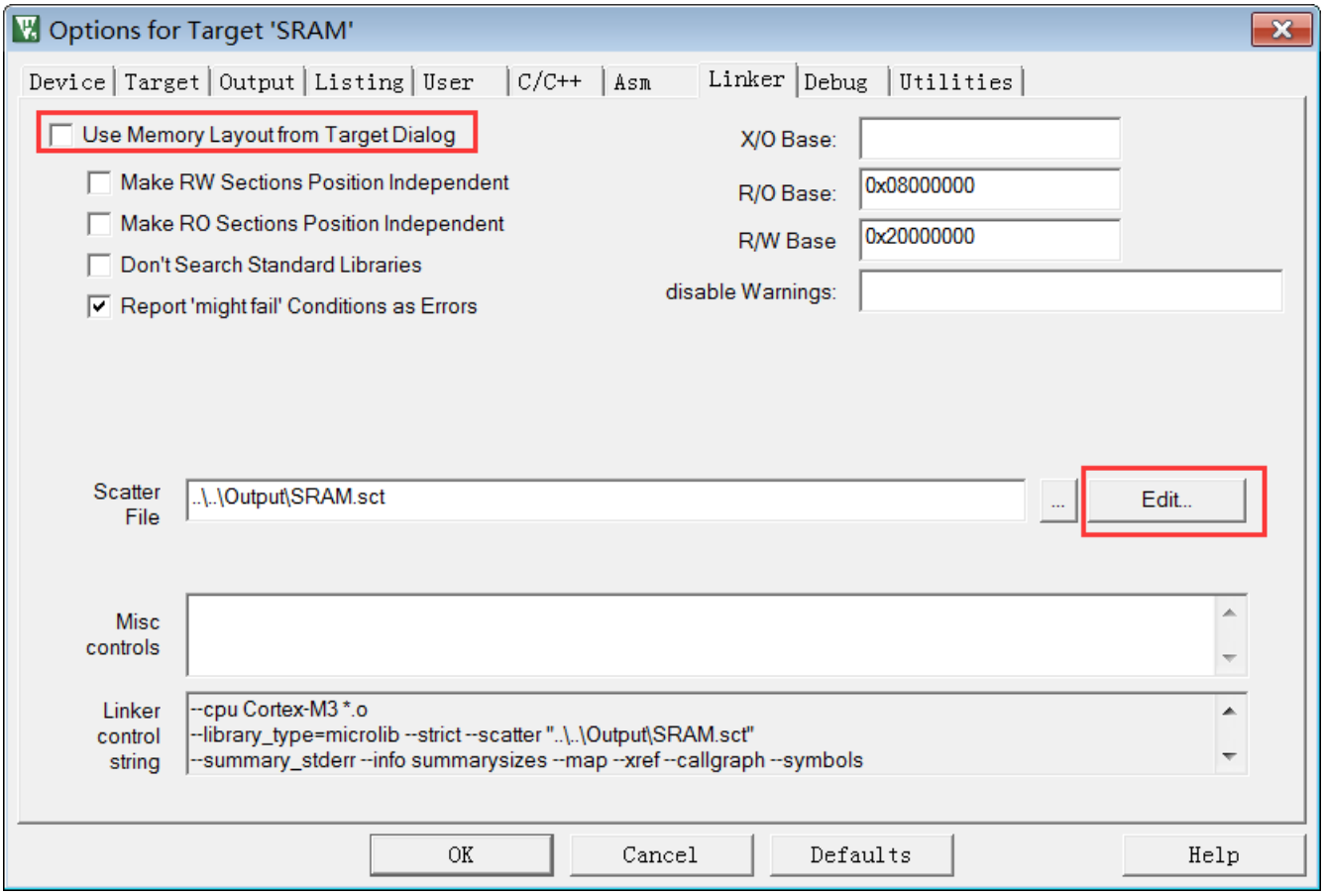
39.5.2 硬件设计

本小节使用内部 SRAM 的 0x20005000 至 0x2000C000 地址，大小为 28K 字节的区域作为“指定的存储空间”，演示如何使用 sct 控制变量变动分配到指定的存储空间，无需使用其它硬件。

39.5.3 软件设计

本小节中提供的例程名为“SCT 文件应用—自动分配变量到指定的空间”，学习时请打开该工程来理解，该工程是由基础实验“点亮 LED 灯”改写而来的。

为方便讲解，本实验直接使用手动编写的 sct 文件，所以在 MDK 的“Options for Target->Linker->Use Memory Layout from Target Dialog”选项被取消勾选，取消勾选后可直接点击“Edit”按钮编辑工程的 sct 文件，也可到工程目录下打开编辑，见图 *40_53*。



取消了这个勾选后，在 MDK 的 Target 对话框及文件配置的存储器分布选项都会失效，仅以 sct 文件中的为准，更改对话框及文件配置选项都不会影响 sct 文件的内容。

39.5.3.1 编程要点

(1) 修改启动文件，在 __main 执行之前初始化“指定的存储空间”的硬件；

(2) 在 sct 文件中增加“指定的存储空间”对应的执行域；

(3) 使用节区选择语句选择要分配到“指定的存储空间”的内容；

(4) 编写测试程序，编译正常后，查看 map 文件的空间分配情况。

39.5.3.2 代码分析

39.5.3.2.1 在 __main 之前初始化外部“指定的存储空间”的硬件

在前面讲解 ELF 文件格式的小节中我们了解到，芯片启动后，会通过 __main 函数调用分散加载代码 __scatterload，分散加载代码会把存储在 FLASH 中的 RW-data 复制到 RAM 中，然后在 RAM 区开辟一块 ZI-data 的空间，并将其初始化为 0 值。因此，为了保证在程序中定义到“指定的存储空间”中的变量能被正常初始化，我们需要在系统执行分散加载代码之前使该空间使用的存储器正常运转，使它能够正常保存数据。如果把存储器的初始化放到 C 语言的 main 函数才执行，那么由于 __scatterload 时存储器没有正常工作，所以拷贝过程无效，那么 RW-data 类型变量的初值都会不正常，这会导致程序运行不符合预期。

为了解决这个问题，可修改工程的 startup_stm32f10x.s 启动文件，见*代码清单 40_23*。

列表 23: 代码清单 40-23 修改启动文件中的 Reset_handler 函数 (startup_stm32f10x.s 文件)

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

       ; 从外部文件引入声明, 格式: IMPORT 要调用的初始化函数名
; 以下语句仅作演示，使用外部存储器时请去掉注释用的“;”号，本工程使用内部 SRAM，无需初始化
                 ;IMPORT FSMC_SRAM_Init
                 LDR     R0, =SystemInit
                 BLX     R0

    ; 在__main 之前调用 FSMC_SRAM_Init 进行初始化
; 以下语句仅作演示，使用外部存储器时请去掉注释用的“;”号，本工程使用内部 SRAM，无需初始化
              ;LDR   R0, =FSMC_SRAM_Init
              ;BLX   R0
```

(下页继续)

(续上页)

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

在原来的启动文件中可以增加上述加粗表示的代码，增加的代码中使用到汇编语法 IMPOR，引入用户在其它 C 语言文件中定义的名为 FSMC_SRAM_Init 的函数（函数名要根据具体的程序来改），接着使用 LDR 指令加载函数的代码地址到寄存器 R0，最后使用 BLX R0 指令跳转到 FSMC_SRAM_Init 的代码地址执行。

加入的代码实现了 Reset_handler 在执行 __main 函数前先调用了我们自定义的 FSMC_SRAM_Init 函数，从而为分散加载代码准备好正常的硬件工作环境。

注意：

在本工程中，由于使用内部 SRAM 空间，不需要初始化，所以实际上不需要修改启动文件，保持与普通工程的一致即可。

39.5.3.2.2 sct 文件初步应用

接下来修改 sct 文件，控制使得在 C 源文件中定义的全局变量都自动由链接器分配到“指定的存储空间”（0x20005000 至 0x2000C000 地址的内部 SRAM 空间），见*代码清单 40_24*。

列表 24: 代码清单 40-24 配置 sct 文件 (SRAM.sct 文件)

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00080000{     ; 加载域
ER_IROM1 0x08000000 0x00080000{   ; 加载地址 = 执行地址
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
```

(下页继续)

(续上页)

```
}

RW_IRAM1 0x20000000 0x00005000 {  ; 内部 SRAM
*.o(STACK)                        ; 选择 STACK 节区，栈
stm32f10x_rcc.o(+RW)              ; 选择 stm32f10x_rcc 的 RW 内容
.ANY (+RW +ZI)                    ; 其余的 RW/ZI-data 都分配到这里
}

RW_ERAM1 0x20005000 0x00007000{  ; “指定的存储空间”

   .ANY (+RW +ZI)                    ; 其余的 RW/ZI-data 都分配到这里
}
}
```

加粗部分是本例子中增加的代码，我们从后面开始，先分析比较简单的外部 SRAM 执行域部分。

- RW_ERAM1 0x20005000 0x00007000{}

  RW_ERAM1 是我们配置的“指定的存储空间”的执行域。该执行域的名字是可以随便取的，最重要的是它的基地址及空间大小，在本工程中，这两个值均指向内部 SRAM，正是我们预设的 0x20005000 至 0x2000C000 地址区域，若使用外扩存储器，把它设置为该存储器的基地址及大小即可。在 RW_ERAM1 执行域内部，它使用“.ANY(+RW +ZI)”语句，选择了所有的 RW/ZI 类型的数据都分配到这个“指定的存储空间”，所以我们在工程中的 C 文件定义全局变量时，它都会被分配到该区域，若使用外扩存储器，这些数据就会存储到相应的存储器中。

- RW_IRAM1 执行域

  RW_IRAM1 是 STM32 原内部 SRAM 的执行域，因为实验需要，工程中只给它分配了 0x5000 字节的空间。我们在它的执行域配置中增加了“*.o(STACK) 及 stm32f10x_rcc.o(+RW)”语句。本来上面配置外部 SRAM 执行域后已经达到使全局变量分配的目的，为何还要修改原内部 SRAM 的执行域呢?

  这是由于如果在 __main 之前调用的 FSMC_SRAM_Init 外部存储器初始化函数调用了

很多库函数，且这些函数内部定义了一些局部变量，而函数内的局部变量是需要分配到“栈”空间 (STACK)，见表 40-1，所以在 FSMC_SRAM_Init 函数执行之前，栈空间必须要被准备好，然而在 FSMC_SRAM_Init 函数执行之前，外部存储器却并未正常工作，这样的矛盾导致栈空间不能被分配到外部存储器区域。

虽然内部 SRAM 的执行域 RW_IRAM1 及“指定的存储空间”执行域 RW_ERAM1 中都使用“.ANY(+RW +ZI)”语句选择了所有 RW 及 ZI 属性的内容，但对于符合两个相同选择语句的内容，链接器会优先选择使用空间较大的执行域，即这种情况下只有当“指定的存储空间”执行域的空间使用完了，RW/ZI 属性的内容才会被分配到内部 SRAM。

所以在大部分情况下，内部 SRAM 执行域中的“.ANY(+RW +ZI)”语句是不起作用的，而栈节区 (STACK) 又属于 ZI-data 类，如果我们的内部 SRAM 执行域还是按原来的默认配置的话，栈节区会被分配到“指定的存储空间”，若此时“指定的存储空间”使用的是外部存储器，将会导致出错。为了避免这个问题，我们把栈节区使用“*.o(STACK)”语句分配到内部 SRAM 的执行域。

增加“stm32f10x_rcc.o(+RW)”语句是因为初始化外部存储器的 FSMC_SRAM_Init 函数可能会调用 stm32f10x_rcc.c 文件中的 RCC_AHBPeriphClockCmd 函数，而查看 map 文件后了解到 stm32f10x_rcc.c 定义了一些 RW-data 类型的变量，见图 *40_54*。不管这些数据是否在 FSMC_SRAM_Init 调用过程中使用到，保险起见，我们直接把这部分内容也分配到内部 SRAM 的执行区。

SRAM.map

```
799  ================================================================================
800
801  Image component sizes
802
803
804        Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
805
806          96           6          0          0          0        634   bsp_led.o
807         188          10          0          0          0       1091   bsp_usart.o
808           0           0          0          0          0       4504   core_cm3.o
809         436         242          0          0          1       2271   main.o
810        1172         188          0          0          0       2764   sram.o
811          44          12        304          0       1024        944   startup_stm32f10x_hd.o
812        1548          18          0          0          0       5567   stm32f10x_fsmc.o
813         860          38          0          0          0       5869   stm32f10x_gpio.o
814          26           0          0          0          0       1318   stm32f10x_it.o
815         932          36          0         20          0       9144   stm32f10x_rcc.o
816        1032          22          0          0          0       8596   stm32f10x_usart.o
817         480          38          0         20          0     236987   system_stm32f10x.o
818
819      --------------------------------------------------------------------------------
820        6816         610        336         40       1028     279689   Object Totals
821           0           0         32          0          0          0   (incl. Generated)
822           2           0          0          0          3          0   (incl. Padding)
823
824      --------------------------------------------------------------------------------
```

图 40_54 SRAM.map 文件中查看到 stm32f10x_rcc.o 文件的 RW-data 使用统计信息

注意：

由于本工程“指定的存储空间”也是属于内部 SRAM，所以“*.o(STACK) 及 stm32f10x_rcc.o(+RW)”语句不添加也是没有问题的，此处为了兼容当该空间为外部存储器时的情况。

39.5.3.2.3 变量分配测试及结果

接下来查看本工程中的 main 文件，它定义了各种变量测试空间分配，见*代码清单 40_25*。

列表 25: 代码清单 40-25 main 文件

```
//定义变量到“指定的存储空间”
uint32_t testValue  =7 ;
//定义变量到“指定的存储空间”
```

(下页继续)

(续上页)

```
uint32_t testValue2  =0;

//定义数组到“指定的存储空间”
uint8_t testGrup[100]  = {0};
//定义数组到“指定的存储空间”
uint8_t testGrup2[100] = {1,2,3};

/* 本实验中的 sct 配置，若使用外部存储器时，堆区工作可能不正常，
使用 malloc 无法得到正常的地址，不推荐在实际工程应用 */
/* 另一种我们推荐的配置请参考教程中的说明 */

/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
   uint32_t inerTestValue =10;

   /* LED 端口初始化 */
   LED_GPIO_Config();

   /* 初始化串口 */
   USART_Config();

   printf("\r\nSCT 文件应用——自动分配变量到“指定的存储空间”实验\r\n");

   printf("\r\n使用“ uint32_t inerTestValue =10; ”语句定义的局部变量：\r\n");
   printf("结果：它的地址为：0x%x，变量值为：%d\r\n",(uint32_t)&inerTestValue,
↪inerTestValue);
```

(下页继续)

(续上页)

```
  printf("\r\n使用 “uint32_t testValue  =7 ;” 语句定义的全局变量: \r\n");
  printf("结果: 它的地址为: 0x%x, 变量值为: %d\r\n", (uint32_t)&testValue,
↪testValue);

  printf("\r\n使用 “uint32_t testValue2  =0 ; ” 语句定义的全局变量: \r\n");
  printf("结果: 它的地址为: 0x%x, 变量值为: %d\r\n", (uint32_t)&testValue2,
↪testValue2);



  printf("\r\n使用 “uint8_t testGrup[100]  ={0};” 语句定义的全局数组: \r\n");
  printf("结果: 它的地址为: 0x%x, 变量值为: %d,%d,%d\r\n",(uint32_t)&testGrup,
↪testGrup[0],testGrup[1],testGrup[2]);

  printf("\r\n使用 “uint8_t testGrup2[100] ={1,2,3};” 语句定义的全局数组: \r\n
↪");
  printf("结果: 它的地址为: 0x%x, 变量值为: %d, %d,%d\r\n",(uint32_t)&testGrup2,
↪testGrup2[0],testGrup2[1],testGrup2[2]);



  /* 本实验中的 sct 配置, 若使用外部存储器时, 堆区工作可能不正常,
     使用 malloc 无法得到正常的地址, 不推荐在实际工程应用 */
  /* 另一种我们推荐的配置请参考教程中的说明 */

  uint32_t * pointer = (uint32_t*)malloc(sizeof(uint32_t)*3);
  if (pointer != NULL) {
     *(pointer)=1;
     *(++pointer)=2;
     *(++pointer)=3;

     printf("\r\n使用 “ uint32_t *pointer = (uint32_t*)malloc(sizeof(uint32_
↪t)*3); ” 动态分配的变量\r\n");
     printf("\r\n定义后的操作为: \r\n*(pointer++)=1;\r\n*(pointer++)=2;\r\
↪n*pointer=3;\r\n\r\n");
```

(下页继续)

(续上页)

```
      printf("结果: 操作后它的地址为: 0x%x, 查看变量值操作: \r\n",(uint32_
↪t)pointer);
      printf("*(pointer--)=%d, \r\n",*(pointer--));
      printf("*(pointer--)=%d, \r\n",*(pointer--));
      printf("*(pointer)=%d, \r\n",*(pointer));

      free(pointer);
   } else {
      printf("\r\n使用 malloc 动态分配变量出错!!! \r\n");
   }
   LED_BLUE;

   while (1);
}
```

代码中定义了局部变量、初值非 0 的全局变量及数组、初值为 0 的全局变量及数组以及动态分配内存，并把它们的值和地址通过串口打印到上位机，通过这些变量，我们可以测试栈、ZI/RW-data 及堆区的变量是否能正常分配。构建工程后，首先查看工程的 map 文件观察变量的分配情况，见图 *40_55* 及图 *40_56*。

SRAM.map

		地址		大小	节区
	i.__scatterload_zeroinit)				
737	free	0x0800169d	Thumb Code	76	malloc.o(i.free)
738	malloc	0x080016ed	Thumb Code	92	malloc.o(i.malloc)
739	Region$$Table$$Base	0x0800183c	Number	0	anon$$obj.o(Region$$Table)
740	Region$$Table$$Limit	0x0800187c	Number	0	anon$$obj.o(Region$$Table)
741	__initial_sp	0x20000418	Data	0	startup_stm32f10x_hd.o(STACK)
742	SystemCoreClock	0x20005000	Data	4	system_stm32f10x.o(.data)
743	AHBPrescTable	0x20005004	Data	16	system_stm32f10x.o(.data)
744	testValue	0x20005014	Data	4	main.o(.data)
745	testValue2	0x20005018	Data	4	main.o(.data)
746	testGrup2	0x2000501c	Data	100	main.o(.data)
747	__stdout	0x20005080	Data	4	stdout.o(.data)
748	__microlib_freelist	0x20005084	Data	4	mvars.o(.data)
749	__microlib_freelist_initialised	0x20005088	Data	4	mvars.o(.data)
750	testGrup	0x2000508c	Data	100	main.o(.bss)
751	__heap_base	0x200050f0	Data	0	startup_stm32f10x_hd.o(HEAP)
752	__heap_limit	0x200052f0	Data	0	startup_stm32f10x_hd.o(HEAP)

图 40-55 在 map 文件中查看工程的存储分布 1(SRAM.map 文件)

SRAM.map

```
    Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x00000418, Max: 0x00005000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x20000000   0x00000014   Data   RW          318    .data               stm32f10x_rcc.o
    0x20000014   0x00000004   PAD
    0x20000018   0x00000400   Zero   RW            1    STACK               startup_stm32f10x_hd.o


    Execution Region RW_ERAM1 (Base: 0x20005000, Size: 0x000002f0, Max: 0x00007000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x20005000   0x00000014   Data   RW           25    .data               system_stm32f10x.o
    0x20005014   0x0000006c   Data   RW          406    .data               main.o
    0x20005080   0x00000004   Data   RW          781    .data               mc_w.l(stdout.o)
    0x20005084   0x00000004   Data   RW          790    .data               mc_w.l(mvars.o)
    0x20005088   0x00000004   Data   RW          791    .data               mc_w.l(mvars.o)
    0x2000508c   0x00000064   Zero   RW          404    .bss                main.o
    0x200050f0   0x00000200   Zero   RW            2    HEAP                startup_stm32f10x_hd.o


==============================================================================

Image component sizes


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

          96           6          0          0          0        638   bsp_led.o
         188          10          0          0          0       1095   bsp_usart.o
           0           0          0          0          0       4504   core_cm3.o
         844         588        227        108        100       3183   main.o
          36           8        304          0       1536        948   startup_stm32f10x_hd.o
         860          38          0          0          0       5877   stm32f10x_gpio.o
          26           0          0          0          0       1322   stm32f10x_it.o
         932          36          0         20          0       9160   stm32f10x_rcc.o
        1032          22          0          0          0       8608   stm32f10x_usart.o
         480          38          0         20          0     228979   system_stm32f10x.o

    ------------------------------------------------------------------------------
        4496         746        596        148       1640     264314   Object Totals
           0           0         64          0          0          0   (incl. Generated)
           2           0          1          0          4          0   (incl. Padding)
```

图 40-56 在 map 文件中查看工程的存储分布 2(SDRAM.map 文件)

上述 map 文件中的“.data”表示 RW-data 类型的节区，“.bss”表示 ZI-data 类型的节区。

从 map 文件中，可看到 stm32f10x_rcc 的 RW-data 及栈空间节区 (STACK) 都被分配到了 RW_IRAM1 区域，即 STM32 的内部 SRAM 空间中；而 main 文件中定义的 RW-data、ZI-data 以及堆空间节区 (HEAP) 都被分配到了 RW_ERAM1 区域，即我们“指定的存储空间”中。map

文件显示，这些变量的分配与 sct 文件配置的目标一致。(堆空间属于 ZI-data，由于没有像控制栈节区那样指定到内部 SRAM，所以它被默认分配到“指定的存储空间”了；在 main 文件中我们定义了一个初值为 0 的全局变量 testValue2 及初值为 0 的数组 testGrup[100]，它们本应占用的是 104 字节的 ZI-data 空间，但在 map 文件中却查看到它仅使用了 100 字节的 RW-data 空间，这是因为链接器把 testValue2 分配为 RW-data 类型的变量了，这是链接器本身的特性，它对像 testGrup[100] 这样的数组才优化作为 ZI-data 分配，这不是我们 sct 文件导致的空间分配错误。)

接下来把程序下载到实验板进行测试，串口打印的调试信息如图 *40_57*。

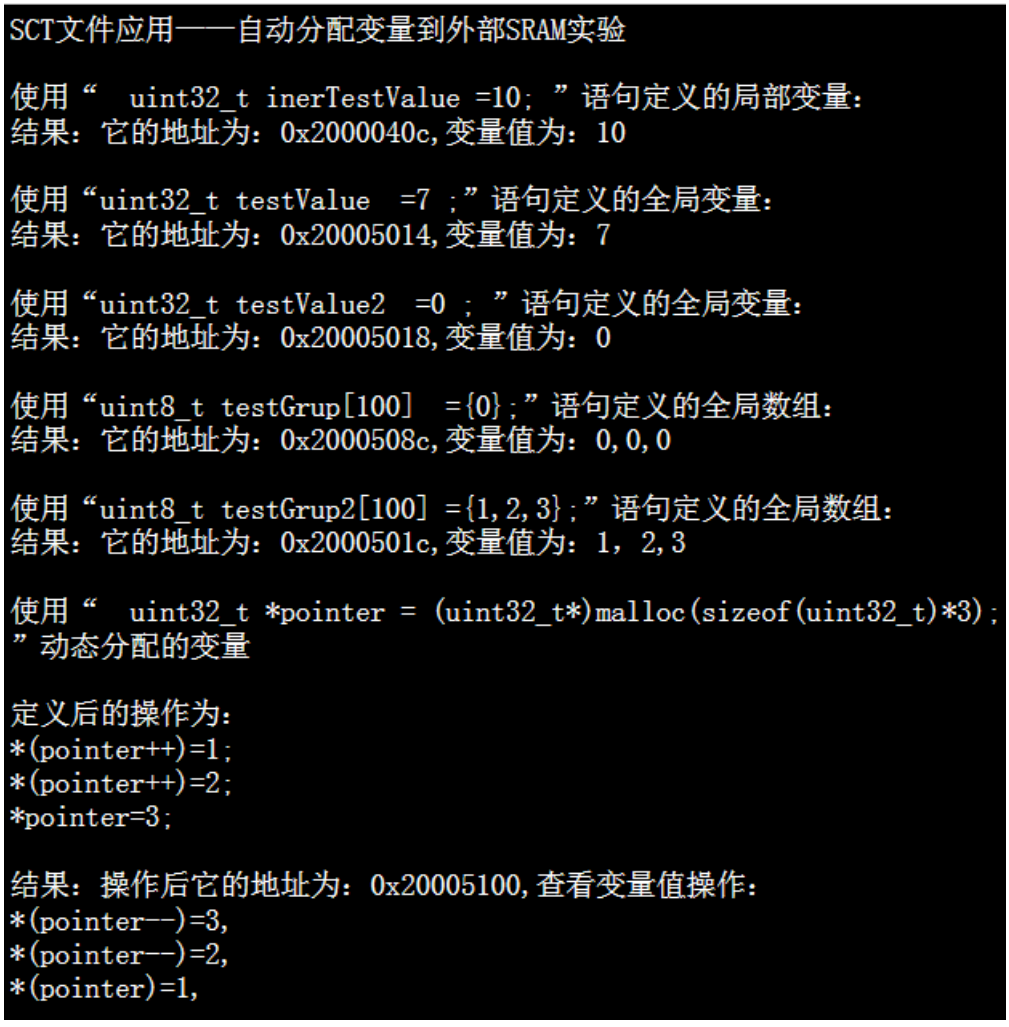


图 40-57 空间分配实验实测结果

从调试信息中可发现，变量都定义到了正确的位置，如内部变量定义在内部 SRAM 的栈区域，全

局变量定义到了“指定的存储空间”。

经过我的测试，使用外部存储器即外扩 SRAM 或 SDRAM 时，曾出现过堆空间分配错误的情况，该情况下即使在 sct 文件中使用“*.o(HEAP)”语句指定堆区到内部 SRAM 或外部外部存储器区域，都无法正常使用 malloc 分配空间。另外，由于外部存储器的读写速度比内部 SRAM 的速度慢，所以我们更希望默认定义的变量先使用内部 SRAM，当它的空间使用完毕后再把变量分配到外部存储器。

在下一小节中我们将改进 sct 的文件配置，解决这两个问题。

39.5.3.3 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手查看各个变量输出的变量值及地址值是否与 sct 文件配置的目标一致。

39.6 实验：优先使用内部 SRAM 并把堆区分配到指定空间

本实验使用另一种方案配置 sct 文件，使得默认情况下优先使用内部 SRAM 空间，在需要的时候使用一个关键字指定变量存储到“指定的存储空间”，另外，我们还把系统默认的堆空间 (HEAP) 映射到“指定的存储空间”，从而可以使用 C 语言标准库的 malloc 函数动态从中分配变量，利用标准库对该空间内存管理，这在外部内存管理中非常有用。

39.6.1 硬件设计

本小节与上一个实验一样，使用内部 SRAM 的 0x20005000 至 0x2000C000 地址，大小为 28K 字节的区域作为“指定的存储空间”，无需使用其它硬件。。

39.6.2 软件设计

本小节中提供的例程名为“SCT 文件应用—优先使用内部 SRAM 并把堆分配到指定空间”，学习时请打开该工程来理解，该工程从上一小节的实验改写而来的，同样地，本工程只使用手动编辑的 sct 文件配置，不使用 MDK 选项配置，在“Options for Target->linker”的选项见图 *40_58*。

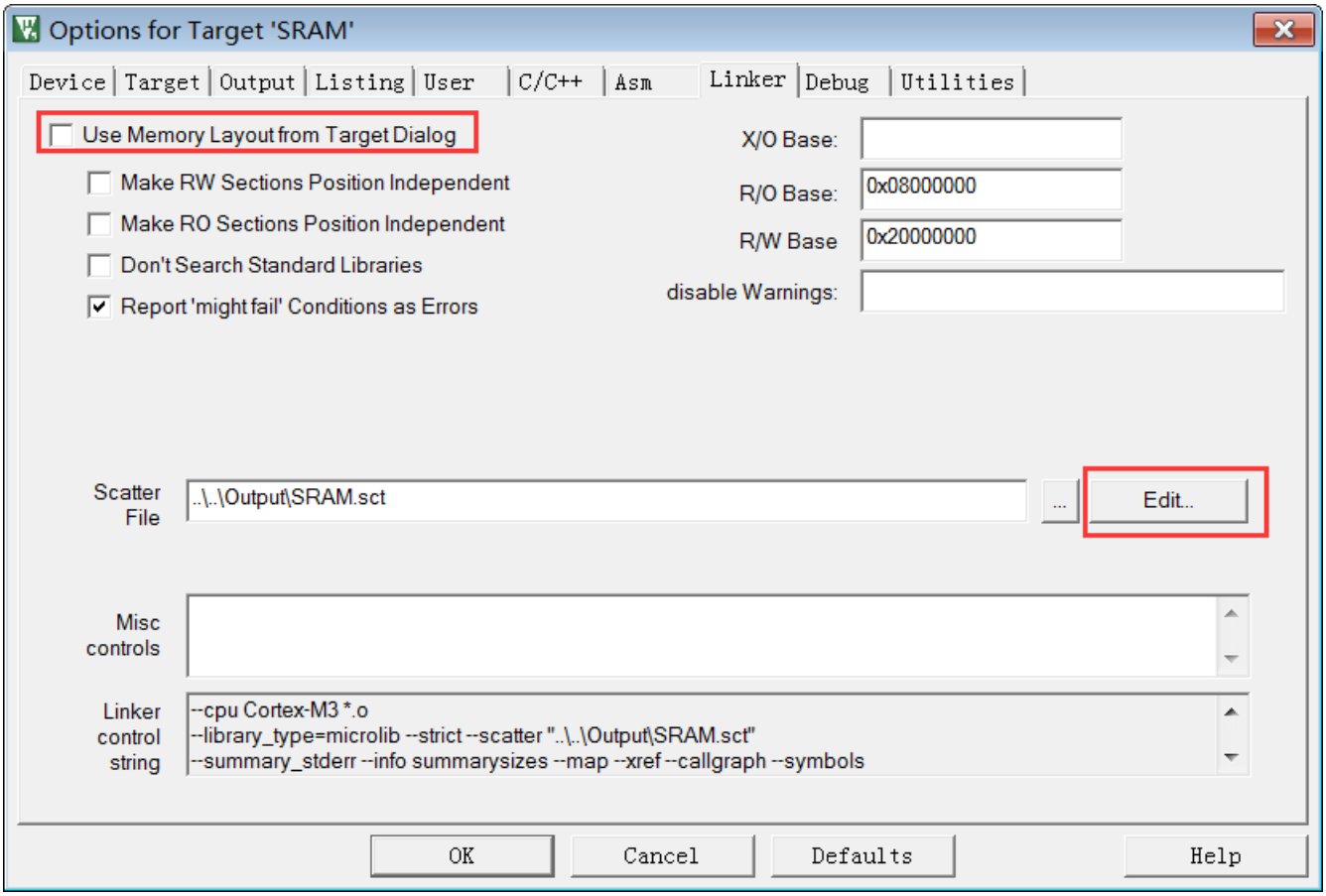


图 40-58 使用手动编写的 sct 文件

取消了这个默认的“Use Memory Layout from Target Dialog”勾选后，在 MDK 的 Target 对话框及文件配置的存储器分布选项都会失效，仅以 sct 文件中的为准，更改对话框及文件配置选项都不会影响 sct 文件的内容。

39.6.2.1 编程要点

(1) 修改启动文件，在 __main 执行之前初始化“指定的存储空间”的硬件；

(2) 在 sct 文件中增加“指定的存储空间”对应的执行域；

(3) 在“指定的存储空间”的执行域中选择一个自定义节区“EXRAM”；

(4) 使用 __attribute__ 关键字指定变量分配到节区“EXRAM”；

(5) 使用宏封装 __attribute__ 关键字，简化变量定义；

(6) 根据需要，把堆区分配到内部 SRAM 或“指定的存储空间”；

(7) 编写测试程序，编译正常后，查看 map 文件的空间分配情况。

39.6.2.2 代码分析

39.6.2.2.1 在 __main 之前初始化外部“指定的存储空间”的硬件

同样地，若“指定的存储空间”是外部存储器，为了使定义到该空间的变量能被正常初始化，需要修改工程 startup_stm32f10x.s 启动文件中的 Reset_handler 函数，在 __main 函数之前调用该存储器的初始化函数使硬件正常运转，见*代码清单 40_26*。

列表 26: 代码清单 40-26 修改启动文件中的 Reset_handler 函数 (startup_stm32f10x.s 文件)

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

       ; 从外部文件引入声明, 格式: IMPORT 要调用的初始化函数名
; 以下语句仅作演示，使用外部存储器时请去掉注释用的“;”号，本工程使用内部 SRAM，无需初始化
                 ;IMPORT FSMC_SRAM_Init
```

(下页继续)

(续上页)

```
                LDR     R0, =SystemInit
                BLX     R0

   ; 在__main 之前调用 FSMC_SRAM_Init 进行初始化
; 以下语句仅作演示，使用外部存储器时请去掉注释用的“;”号，本工程使用内部 SRAM，无需初始化
            ;LDR   R0, =FSMC_SRAM_Init
            ;BLX   R0

                LDR     R0, =__main
                BX      R0
                ENDP
```

它与上一小节中一样，当芯片上电运行 Reset_handler 函数时，在执行 __main 函数前先调用了自定义的外部存储器初始化函数 FSMC_SRAM_Init，从而为分散加载代码准备好正常的硬件工作环境。本工程使用的演示空间为内部 SRAM 区域，不需要初始化，因此代码该部分额外的代码都注释掉了。

39.6.2.2.2 sct 文件配置

接下来分析本实验中的 sct 文件配置与上一小节有什么差异，见*代码清单 40_27*。

列表 27: 代码清单 40-27 本实验的 sct 文件内容 (SRAM.sct)

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************
LR_IROM1 0x08000000 0x00080000{    ; load region size_region
ER_IROM1 0x08000000 0x00080000{  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
```

(下页继续)

(续上页)

```
    .ANY (+RO)
}


RW_IRAM1 0x20000000 0x00005000 {  ; 内部 SRAM
.ANY (+RW +ZI)       ; 其余的 RW/ZI-data 都分配到这里
}

RW_ERAM1 0x20005000 0x00007000{  ; 指定的存储空间
*.o(HEAP)              ; 选择堆区
    .ANY (EXRAM)        ; 选择 EXRAM 节区
}
}
```

本实验的 sct 文件中的 RW_IRAM1 执行域是 STM32 原内部 SRAM 的执行域，因为实验需要，工程中只给它分配了 0x5000 字节的空间。另外新增了一个“指定的存储空间”执行域 RW_ERAM1，它的基地址和大小正是我们预设的 0x20005000 至 0x2000C000 地址区域。代码中还使用了“*.o(HEAP)”语句把堆区分配到了 RW_ERAM1，使用“.ANY(EXRAM)”语句把名为“EXRAM”的节区分也配到 RW_ERAM1。

这个“EXRAM”节区是由我们自定义的，在语法上就跟在 C 文件中定义全局变量类似，只要它跟工程中的其它原有节区名不一样即可。有了这个节区选择配置，当我们需要定义变量到“指定的存储空间”时，只需要指定该变量分配到该节区，它就会被分配到该空间中。

本实验中的 sct 配置就是这么简单，接下来直接使用就可以了。

39.6.2.2.3 指定变量分配到节区

当我们需要把变量分配到“指定的存储空间”时，需要使用 __attribute__ 关键字指定节区，它的语法见*代码清单 40_28*。

列表 28: 代码清单 40-28 指定变量定义到某节区的语法

```
//使用 __attribute__ 关键字定义指定变量定义到某节区
//语法: 变量定义 __attribute__ ((section (" 节区名"))) = 变量值;
uint32_t testValue  __attribute__ ((section ("EXRAM"))) =7 ;

//使用宏封装
//设置变量定义到“EXRAM”节区的宏
#define __EXRAM  __attribute__ ((section ("EXRAM")))

//使用该宏定义变量到“指定的存储空间”
uint32_t testValue __EXRAM =7 ;
```

上述代码介绍了基本的指定节区语法：“变量定义 __attribute__ ((section (“节区名”))) = 变量值;”，它的主体跟普通的 C 语言变量定义语法无异，在赋值“=”号前 (可以不赋初值)，加了个“__attribute__ ((section (“节区名”)))”描述它要分配到的节区。本例中的节区名为“EXRAM”，即我们在 sct 文件中选择分配到“指定的存储空间”执行域的节区，所以该变量就被分配到该存储器中了。

由于“__attribute__”关键字写起来比较繁琐，我们可以使用宏定义把它封装起来，简化代码。本例中我们把指定到“EXRAM”的描述语句“__attribute__ ((section (“EXRAM”)))”封装成了宏“__EXRAM”，应用时只需要使用宏的名字替换原来“__attribute__”关键字的位置即可，如“uint32_t testValue __EXRAM =7 ;”。有 51 单片机使用经验的读者会发现，这种变量定义方法就跟使用 keil 51 特有的关键字“xdata”定义变量到外部 RAM 空间差不多。

类似地，如果工程中还使用了其它存储器也可以用这样的方法实现变量分配，例如 STM32F429 的高速内部 SRAM(CCM)，可以在 sct 文件增加该高速 SRAM 的执行域，然后在执行域中选择一个自定义节区，在工程源文件中使用“__attribute__”关键字指定变量到该节区，就可以可把变量分配到高速内部 SRAM 了。

根据我们 sct 文件的配置，如果定义变量时没有指定节区，它会默认优先使用内部 SRAM，把变量定义到内部 SRAM 空间，而且由于局部变量是属于栈节区 (STACK)，它不能使用“__attribute__”关键字指定节区。在本例中的栈节区被分配到内部 SRAM 空间。

39.6.2.2.4 变量分配测试及结果

接下来查看本工程中的 main 文件，它定义了各种变量测试空间分配，见*代码清单 40_25*。

列表 29: 代码清单 40-29 main 文件

```
//设置变量定义到“EXRAM”节区的宏
#define __EXRAM  __attribute__ ((section ("EXRAM")))

//定义变量到“指定的存储空间”
uint32_t testValue __EXRAM =7 ;
//上述语句等效于:
//uint32_t testValue  __attribute__ ((section ("EXRAM"))) =7 ;

//定义变量到 SRAM
uint32_t testValue2  =7 ;

//定义数组到“指定的存储空间”
uint8_t testGrup[3] __EXRAM = {1,2,3};
//定义数组到 SRAM
uint8_t testGrup2[3] = {1,2,3};

/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
   uint32_t inerTestValue =10;

   /* LED 端口初始化 */
   LED_GPIO_Config();
```

(下页继续)

(续上页)

```
  /* 初始化串口 */
  USART_Config();
  printf("\r\nSCT 文件应用——自动分配变量到“指定的存储空间”实验\r\n");

  printf("\r\n使用 “ uint32_t inerTestValue =10; ”语句定义的局部变量：\r\n");
  printf("结果：它的地址为：0x%x，变量值为：%d\r\n", (uint32_t)&inerTestValue,
↪inerTestValue);

  printf("\r\n使用 “uint32_t testValue __EXRAM =7 ;”语句定义的全局变量：\r\n");
  printf("结果：它的地址为：0x%x，变量值为：%d\r\n", (uint32_t)&testValue,
↪testValue);

  printf("\r\n使用 “uint32_t testValue2  =7 ; ”语句定义的全局变量：\r\n");
  printf("结果：它的地址为：0x%x，变量值为：%d\r\n", (uint32_t)&testValue2,
↪testValue2);


  printf("\r\n使用 “uint8_t testGrup[3] __EXRAM ={1,2,3};”语句定义的全局数组：\
↪r\n");
  printf("结果：它的地址为：0x%x，变量值为：%d,%d,%d\r\n", (uint32_t)&testGrup,
↪testGrup[0],testGrup[1],testGrup[2]);

  printf("\r\n使用 “uint8_t testGrup2[3] ={1,2,3};”语句定义的全局数组：\r\n");
  printf("结果：它的地址为：0x%x，变量值为：%d，%d,%d\r\n", (uint32_t)&
↪testGrup2,testGrup2[0],testGrup2[1],testGrup2[2]);

  /* 使用 malloc 从外部 SRAM 中分配空间 */
  uint32_t *pointer = (uint32_t*)malloc(sizeof(uint32_t)*3);

  if (pointer != NULL) {
        *(pointer)=1;
```

(下页继续)

(续上页)

```
        *(++pointer)=2;
        *(++pointer)=3;

        printf("\r\n使用 “ uint32_t *pointer = (uint32_
↪t*)malloc(sizeof(uint32_t)*3); ” 动态分配的变量\r\n");
        printf("\r\n定义后的操作为: \r\n*(pointer++)=1;\r\n*(pointer++)=2;\r\
↪n *pointer=3;\r\n\r\n");
        printf("结果: 操作后它的地址为: 0x%x, 查看变量值操作: \r\n",(uint32_
↪t)pointer);
        printf("*(pointer--)=%d, \r\n",*(pointer--));
        printf("*(pointer--)=%d, \r\n",*(pointer--));
        printf("*(pointer)=%d, \r\n",*(pointer));

        free(pointer);
    } else {
        printf("\r\n使用 malloc 动态分配变量出错!!! \r\n");
    }
    /* 蓝灯亮 */
    LED_BLUE;
    while (1);
}
```

代码中定义了普通变量、指定到 EXRAM 节区的变量并使用动态分配内存，还把它们的值和地址通过串口打印到上位机，通过这些变量，我们可以检查变量是否能正常分配。

构建工程后，查看工程的 map 文件观察变量的分配情况，见图 *40_59*。

SRAM.map

		地址		大小	节区
737	free	0x080016a9	Thumb Code	76	malloc.o(i.free)
738	malloc	0x080016f9	Thumb Code	92	malloc.o(i.malloc)
739	Region$$Table$$Base	0x0800184c	Number	0	anon$$obj.o(Region$$Table)
740	Region$$Table$$Limit	0x0800188c	Number	0	anon$$obj.o(Region$$Table)
741	SystemCoreClock	0x20000000	Data	4	system_stm32f10x.o(.data)
742	AHBPrescTable	0x20000004	Data	16	system_stm32f10x.o(.data)
743	testValue2	0x20000028	Data	4	main.o(.data)
744	testGrup2	0x2000002c	Data	3	main.o(.data)
745	__stdout	0x20000030	Data	4	stdout.o(.data)
746	__microlib_freelist	0x20000034	Data	4	mvars.o(.data)
747	__microlib_freelist_initialised	0x20000038	Data	4	mvars.o(.data)
748	__initial_sp	0x20000440	Data	0	startup_stm32f10x_hd.o(STACK)
749	testValue	0x20005000	Data	4	main.o(EXRAM)
750	testGrup	0x20005004	Data	3	main.o(EXRAM)
751	__heap_base	0x20005008	Data	0	startup_stm32f10x_hd.o(HEAP)
752	__heap_limit	0x20005208	Data	0	startup_stm32f10x_hd.o(HEAP)

图 40-59 在 map 文件中查看工程的存储分布 (SRAM.map 文件)

从 map 文件中可看到普通变量及栈节区都被分配到了内部 SRAM 的地址区域，而指定到 EXRAM 节区的变量及堆空间都被分配到了“指定的存储空间”地址区域，与我们的要求一致。

再把程序下载到实验板进行测试，串口打印的调试信息如图 *40_60*。

```
SCT文件应用——自动分配变量到“指定的存储空间”实验

使用“  uint32_t inerTestValue =10; ”语句定义的局部变量:
结果: 它的地址为: 0x20000434,变量值为: 10

使用“uint32_t testValue __EXRAM =7 ;”语句定义的全局变量:
结果: 它的地址为: 0x20005000,变量值为: 7

使用“uint32_t testValue2  =7 ; ”语句定义的全局变量:
结果: 它的地址为: 0x20000028,变量值为: 7

使用“uint8_t testGrup[3] __EXRAM ={1,2,3};”语句定义的全局数组:
结果: 它的地址为: 0x20005004,变量值为: 1,2,3

使用“uint8_t testGrup2[3] ={1,2,3};”语句定义的全局数组:
结果: 它的地址为: 0x2000002c,变量值为: 1, 2,3

使用“  uint32_t *pointer = (uint32_t*)malloc(sizeof(uint32_t)*3);
”动态分配的变量

定义后的操作为:
*(pointer++)=1;
*(pointer++)=2;
*pointer=3;

结果: 操作后它的地址为: 0x20005018,查看变量值操作:
*(pointer--)=3,
*(pointer--)=2,
*(pointer)=1,
```

图 40-60 空间分配实验实测结果

从调试信息中可发现，实际运行结果也完全正常，本实验中的 sct 文件配置达到了优先分配变量到内部 SRAM 的目的，而且堆区也能使用 malloc 函数正常分配空间。

39.6.2.2.5 把堆区分配到内部 SRAM 空间。

若您希望堆区 (HEAP) 按照默认配置，使它还是分配到内部 SRAM 空间，只要把“*.o(HEAP)”选择语句从“指定的存储空间”的执行域删除掉即可，堆节区就会默认分配到内部 SRAM，“指定的存储空间”仅选择 EXRAM 节区的内容进行分配，见*代码清单 40_30*。

列表 30: 代码清单 40-30 按默认配置分配堆区到内部 SRAM 的 sct 文件范例

```
LR_IROM1 0x08000000 0x00080000{    ; load region size_region
ER_IROM1 0x08000000 0x00080000{  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00005000 {  ; 内部 SRAM
.ANY (+RW +ZI)       ; 其余的 RW/ZI-data 都分配到这里
}

RW_ERAM1 0x20005000 0x00007000 {  ; “指定的存储空间”
   .ANY (EXRAM)        ; 选择 EXRAM 节区
}
}
```

39.6.2.2.6 屏蔽链接过程的 warning

在我们的实验配置的 sct 文件中使用了“*.o(HEAP)”语句选择堆区，但有时我们的工程完全没有使用堆 (如整个工程都没有使用 malloc)，这时链接器会把堆占用的空间删除，构建工程后会输出警告提示该语句仅匹配到无用节区，见图 *40_61*。

```
Build Output
Build target 'SDRAM'
compiling main.c...
linking...
..\..\Output\SDRAM.sct(24): warning: L6329W: Pattern *.o(HEAP) only matches removed unused sections.
Program Size: Code=4580 RO-data=532 RW-data=36 ZI-data=1028
Finished: 0 information, 1 warning and 0 error messages.
FromELF: creating hex file...
"..\..\Output\SDRAM.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:00:01
```

这并无什么大碍，但强迫症患者不希望看到 warning，可以在“Options for Target->Linker->disable

Warnings” 中输入 warning 号屏蔽它。warning 号可在提示信息中找到，如上图提示信息中 “warning: L6329W” 表示它的 warning 号为 6329，把它输入到图 *40_62* 中的对话框中即可。

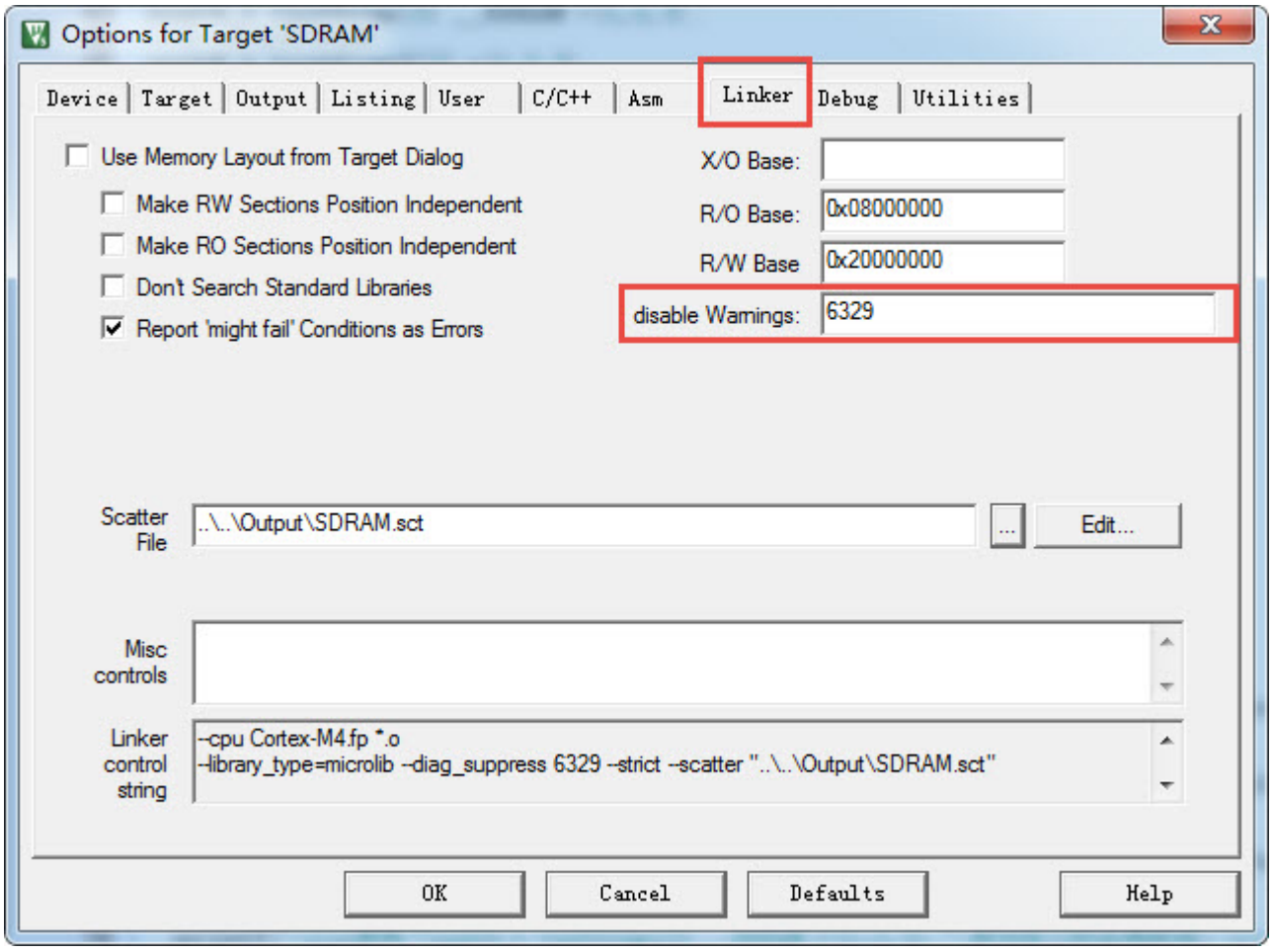


图 40-62 屏蔽链接过程的 warning

39.6.2.3 下载验证

用 USB 线连接开发板 “USB TO UART” 接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手查看各个变量输出的变量值及地址值是否与 sct 文件配置的目标一致。

第 40 章　在 SRAM 中调试代码

本章参考资料：《STM32F10x 中文参考手册》、《STM32F10x 规格书》、《Cortex-M3 权威指南》。

学习本章时，配合《STM32F10x 中文参考手册》“存储器和总线结构”章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

40.1 在 RAM 中调试代码

一般情况下，我们在 MDK 中编写工程应用后，调试时都是把程序下载到芯片的内部 FLASH 运行测试的，代码的 CODE 及 RW-data 的内容被写入到内部 FLASH 中存储。但在某些应用场合下却不希望或不能修改内部 FLASH 的内容，这时就可以使用 RAM 调试功能了，它的本质是把原来存储在内部 FLASH 的代码 (CODE 及 RW-data 的内容) 改为存储到 SRAM 中 (内部 SRAM 或外部 SDRAM 均可)，芯片复位后从 SRAM 中加载代码并运行。

把代码下载到 RAM 中调试有如下优点：

- 下载程序非常快。RAM 存储器的写入速度比在内部 FLASH 中要快得多，且没有擦除过程，因此在 RAM 上调试程序时程序几乎是秒下的，对于需要频繁改动代码的调试过程，能节约很多时间，省去了烦人的擦除与写入 FLASH 过程。另外，STM32 的内部 FLASH 可擦除次数为 1 万次，虽然一般的调试过程都不会擦除这么多次导致 FLASH 失效，但这确实也是一个考虑使用 RAM 的因素。
- 不改写内部 FLASH 的原有程序。
- 对于内部 FLASH 被锁定的芯片，可以把解锁程序下载到 RAM 上，进行解锁。

相对地，把代码下载到 RAM 中调试有如下缺点：

- 存储在 RAM 上的程序掉电后会丢失，不能像 FLASH 那样保存。
- 若使用 STM32 的内部 SRAM 存储程序，程序的执行速度与在 FLASH 上执行速度无异，但 SRAM 空间较小。

- 若使用外部扩展的 SRAM 存储程序，程序空间非常大，但 STM32 读取外部 SRAM 的速度比读取内部 FLASH 慢，这会导致程序总执行时间增加，因此在外部 SRAM 中调试的程序无法完美仿真在内部 FLASH 运行时的环境。另外，由于 STM32 无法直接从外部 SRAM 中启动且应用程序复制到外部 SRAM 的过程比较复杂 (下载程序前需要使 STM32 能正常控制外部 SRAM)，所以在很少会在 STM32 的外部 SRAM 中调试程序。

40.2 STM32 的启动方式

在前面讲解的 STM32 启动代码章节了解到 CM-3 内核在离开复位状态后的工作过程如下，见图 *43_1*：

(1) 从地址 0x00000000 处取出栈指针 MSP 的初始值，该值就是栈顶的地址。

(2) 从地址 0x00000004 处取出程序指针 PC 的初始值，该值指向复位后应执行的第一条指令。

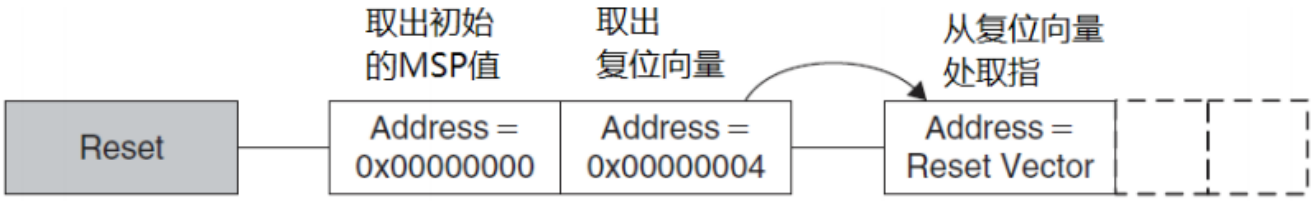


图 43-1 复位序列

上述过程由内核自动设置运行环境并执行主体程序，因此它被称为自举过程。

虽然内核是固定访问 0x00000000 和 0x00000004 地址的，但实际上这两个地址可以被重映射到其它地址空间。以 STM32F103 为例，根据芯片引出的 BOOT0 及 BOOT1 引脚的电平情况，这两个地址可以被映射到内部 FLASH、内部 SRAM 以及系统存储器中，不同的映射配置见表 43-1。

表 43-1 BOOT 引脚的不同设置对 0 地址的映射

BOOT1	BOOT0	映射到的存储器	0x00000000 地址映射到	0x00000004 地址映射到
x	0	内部 FLASH	0x08000000	0x08000004
1	1	内部 SRAM	0x20000000	0x20000004
0	1	系统存储器	0x1FFFB000	0x1FFFB004

内核在离开复位状态后会从映射的地址中取值给栈指针 MSP 及程序指针 PC，然后执行指令，我们一般以存储器的类型来区分自举过程，例如内部 FLASH 启动方式、内部 SRAM 启动方式以及系统存储器启动方式。

(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。具备这两个条件后，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

类似地，当芯片上电后采样到 BOOT0 和 BOOT1 引脚均为高电平时，0x00000000 和 0x00000004 地址被映射到内部 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 startup_stm32f103xe.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。（下一小节将以实例讲解）

(3) 系统存储器启动方式

当芯片上电后采样到 BOOT0 引脚为高电平，BOOT1 为低电平时，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 提供支持 (In System Program)，如检测 USART1/2、CAN2 及 USB 通讯接口传输过来的信息，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。

40.3 内部 FLASH 的启动过程

下面我们以最常规的内部 FLASH 启动方式来分析自举过程，主要理解 MSP 和 PC 内容是怎样被存储到 0x08000000 和 0x08000004 这两个地址的。

见图 *43_2* ，这是 STM32F103 默认的启动文件的代码，启动文件的开头定义了一个大小为 0x400 的栈空间，且栈顶的地址使用标号“__initial_sp”来表示；在图下方定义了一个名为“Reset_Handler”的子程序，它就是我们总是提到的在芯片启动后第一个执行的代码。在汇编语法中，程序的名字和标号都包含它所在的地址，因此，我们的目标是把“__initial_sp”和“Reset_Handler”赋值到 0x08000000 和 0x08000004 地址空间存储，这样内核自举的时候就可以获得栈顶地址以及第一条要执行的指令了。在启动代码的中间部分，使用了汇编关键字“DCD”把“__initial_sp”和“Reset_Handler”定义到了最前面的地址空间。

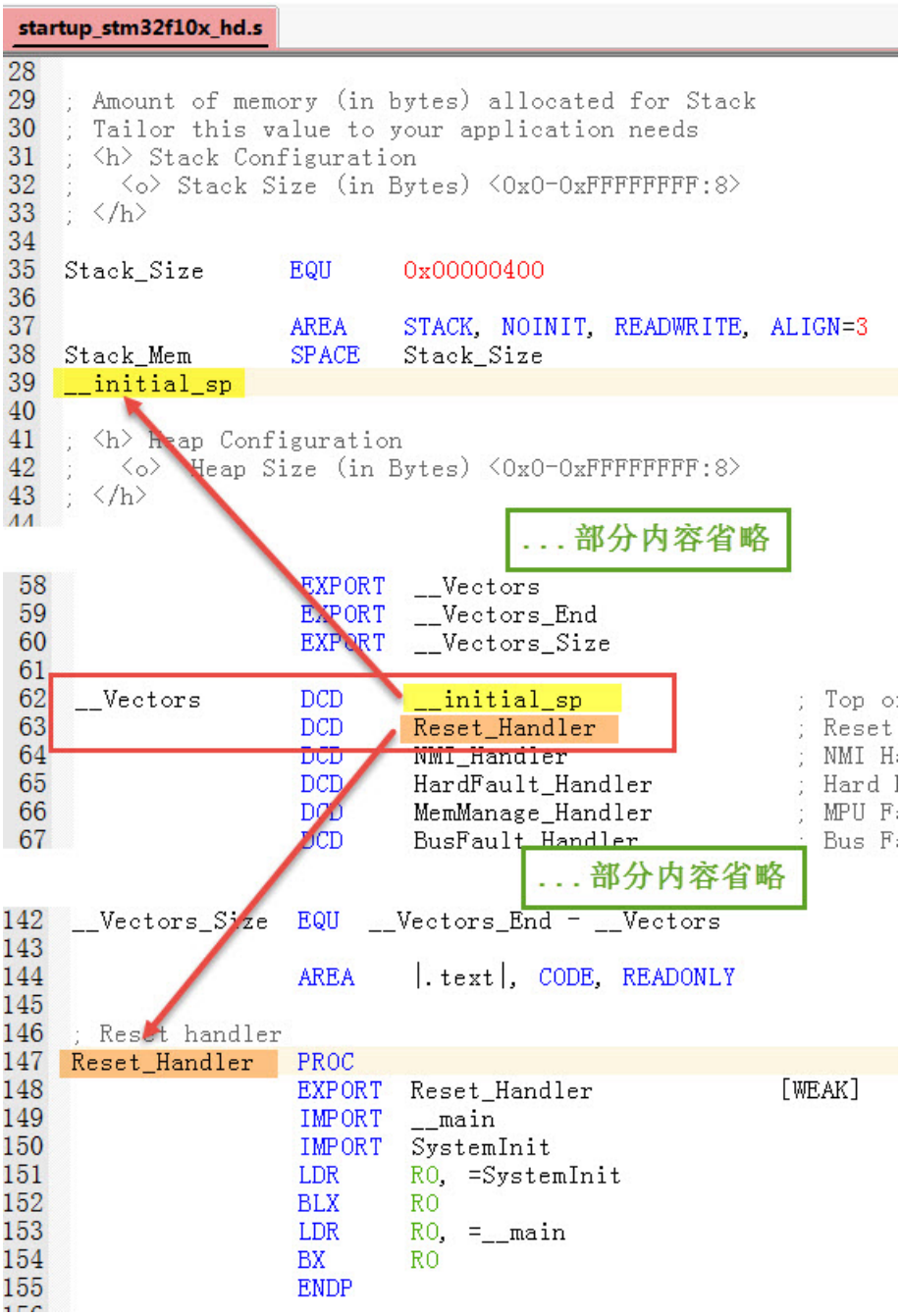


图 43-2 启动代码中存储的 MSP 及 PC 指针内容

在启动文件中把设置栈顶及首条指令地址到了最前面的地址空间，但这并没有指定绝对地址，各种内容的绝对地址是由链接器根据分散加载文件 (*.sct) 分配的，STM32F103 的默认分散加载文件配置见*代码清单 43_1*。

列表 1: 代码清单 43-1 默认分散加载文件的空间配置

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00080000 {    ; load region size_region
  ER_IROM1 0x08000000 0x00080000 {  ; load address = execution address
  *.o (RESET, +First)
  *(InRoot$$Sections)
  .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00010000  {  ; RW data
  .ANY (+RW +ZI)
  }
}
```

分散加载文件把加载区和执行区的首地址都设置为 0x08000000，正好是内部 FLASH 的首地址，因此汇编文件中定义的栈顶及首条指令地址会被存储到 0x08000000 和 0x08000004 的地址空间。

类似地，如果我们修改分散加载文件，把加载区和执行区的首地址设置为内部 SRAM 的首地址 0x20000000，那么栈顶和首条指令地址将会被存储到 0x20000000 和 0x20000004 的地址空间了。

为了进一步消除疑虑，我们可以查看反汇编代码及 map 文件信息来了解各个地址空间存储的内容，见图 *43_3*，这是多彩流水灯工程编译后的信息，它的启动文件及分散加载文件都按默认配置。其中反汇编代码是使用 fromelf 工具从 axf 文件生成的，具体过程可参考前面的章节了解。

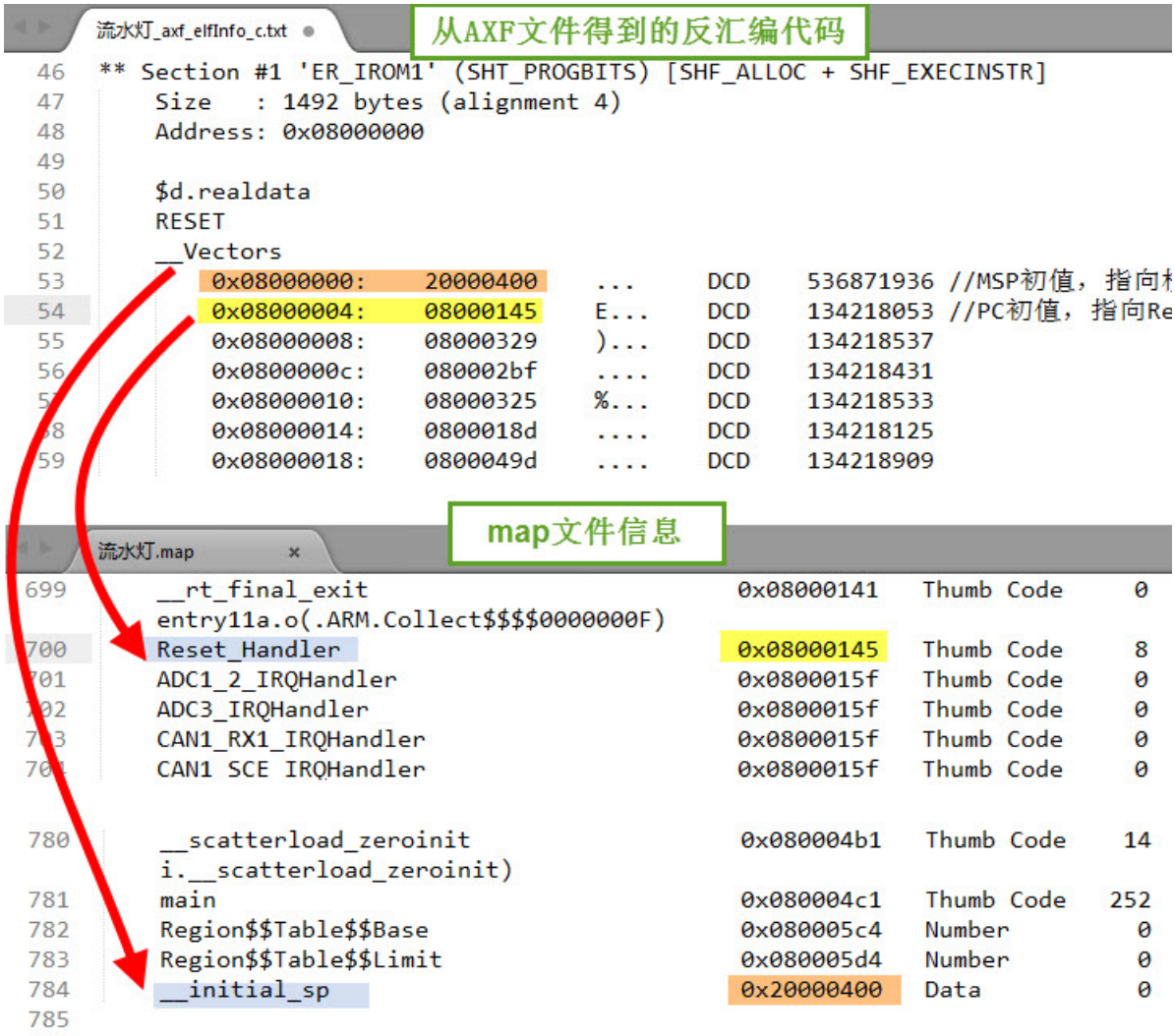


图 43-3 从反汇编代码及 map 文件查看存储器的内容

从反汇编代码可了解到，这个工程的 0x08000000 地址存储的值为 0x20000400，0x08000004 地址存储的值为 0x08000145，查看 map 文件，这两个值正好是栈顶地址 __initial_sp 以及首条指令 Reset_Handler 的地址。下载器会根据 axf 文件 (bin、hex 类似) 存储相应的内容到内部 FLASH 中。

由此可知，BOOT0 为低电平时，内核复位后，从 0x08000000 读取到栈顶地址为 0x20000400，了解到子程序的栈空间范围，再从 0x08000004 读取到第一条指令的存储地址为 0x08000145，于是跳转到该地址执行代码，即从 ResetHandler 开始运行，运行 SystemInit、__main(包含分散加载代码)，最后跳转到 C 语言的 main 函数。

对比在内部 FLASH 中运行代码的过程，可了解到若希望在内部 SRAM 中调试代码，需要设置启动方式为从内部 SRAM 启动，修改分散加载文件控制代码空间到内部 SRAM 地址以及把生成程序下载到芯片的内部 SRAM 中。

40.4 实验：在内部 SRAM 中调试代码

本实验将演示如何设置工程选项实现在内部 SRAM 中调试代码，实验的示例代码名为“RAM 调试—多彩流水灯”，学习以下内容时请打开该工程来理解，它是从普通的多彩流水灯例程改造而来的。

40.4.1 硬件设计

本小节中使用到的流水灯硬件不再介绍，主要讲解与 SRAM 调试相关的硬件配置。在 SRAM 上调试程序，需要修改 STM32 芯片的启动方式，见图 *43_4*。

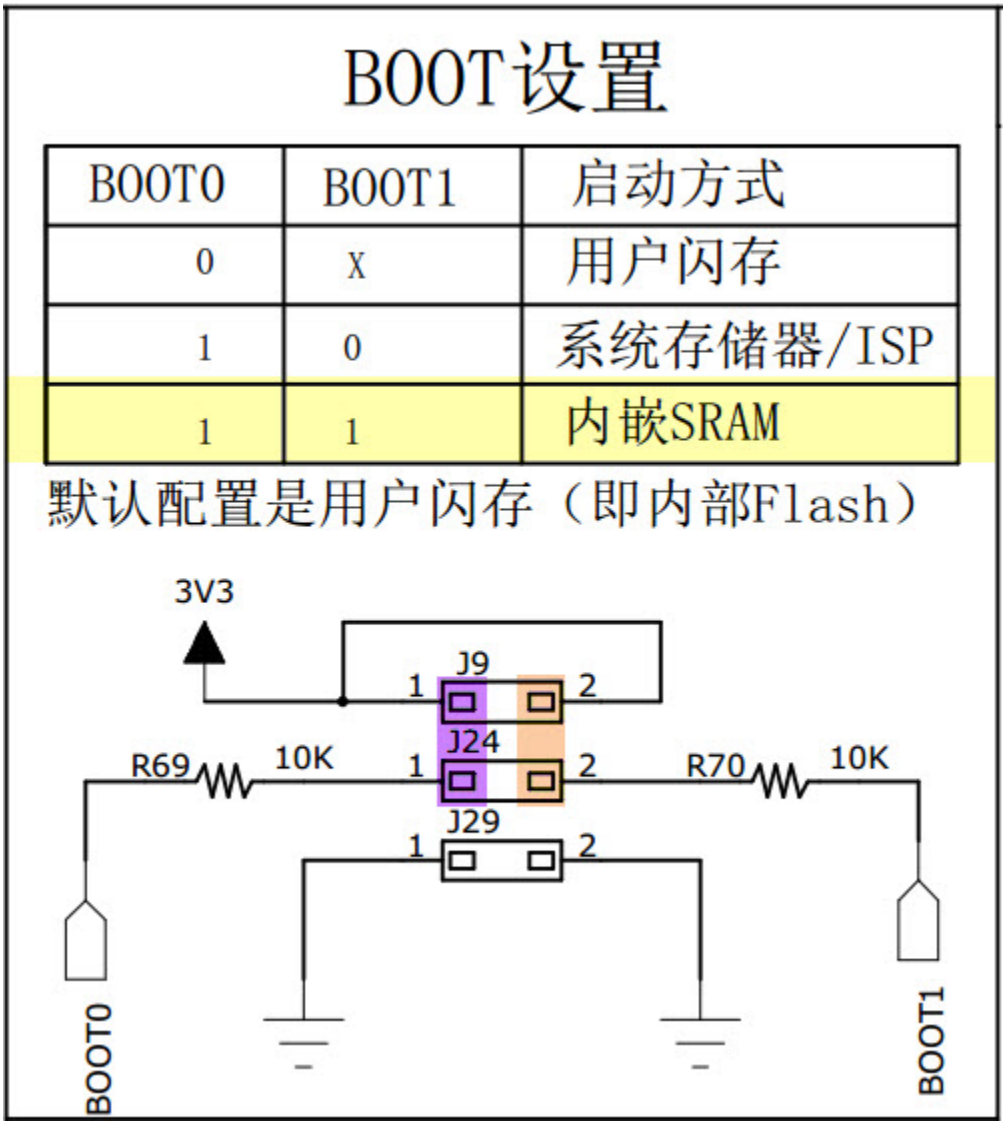


图 43-4 实验板的 boot 引脚配置

在我们的实验板左侧有引出 STM32 芯片的 BOOT0 和 BOOT1 引脚，可使用跳线帽设置它们的电平从而控制芯片的启动方式，它支持从内部 FLASH 启动、系统存储器启动以及内部 SRAM 启动方式。

本实验在 SRAM 中调试代码，因此把 BOOT0 和 BOOT1 引脚都使用跳线帽连接到 3.3V，使芯片从 SRAM 中启动。

40.4.2 软件设计

本实验的工程从普通的多彩流水灯工程改写而来，主要修改了分散加载文件及一些程序的下载选项。

40.4.2.1 主要步骤

(1) 在原工程的基础上创建一个调试版本；

(2) 修改分散加载文件，使链接器把代码分配到内部 SRAM 空间；

(3) 添加宏修改 STM32 的向量表地址；

(4) 修改仿真器和下载器的配置，使程序能通过下载器存储到内部 SRAM；

(5) 根据使用情况选择是否需要使用仿真器命令脚本文件 *.ini；

(6) 尝试给 SRAM 下载程序或仿真调试。

40.4.2.2 创建工程的调试版本

由于在 SRAM 中运行的代码一般只是用于调试，调试完毕后，在实际生产环境中仍然使用在内部 FLASH 中运行的代码，因此我们希望能够便捷地在调试版和发布版代码之间切换。MDK 的“Manage Project Items”可实现这样的功能，使用它可管理多个不同配置的工程，见图 43-5，点击“Manage Project Items”按钮，在弹出对话框左侧的“Project Target”一栏包含了原工程的名字，如图中的原工程名为“多彩流水灯”，右侧是该工程包含的文件。为了便于调试，我们在左侧的“Project Target”一栏添加一个工程名，如图中输入“SRAM_ 调试”，输入后点击 OK 即可，这个“SRAM_ 调试”版本的工程会复制原“多彩流水灯”工程的配置，后面我们再进行修改。

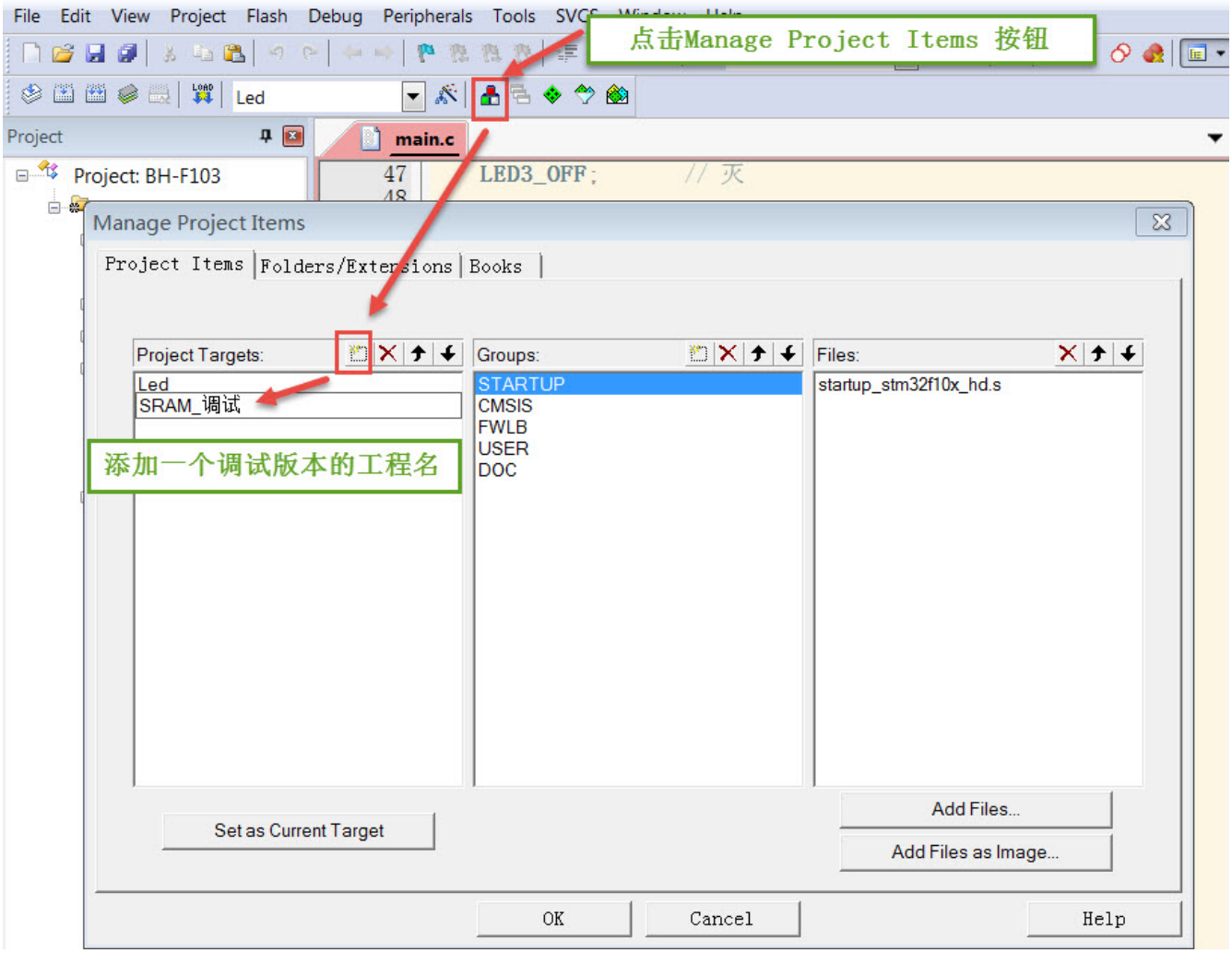


图 43-5 使用 Manage Project Items 添加一个工程配置

当需要切换工程版本时，点击 MDK 工程名的下拉菜单可选择目标工程，在不同的工程中，所有配置都是独立的，例如芯片型号、下载配置等等，但如果两个工程共用了同一个文件，对该文件的修改会同时影响两个工程，例如这两个工程都使用同一个 main 文件，我们在 main 文件修改代码，两个工程都会被修改。

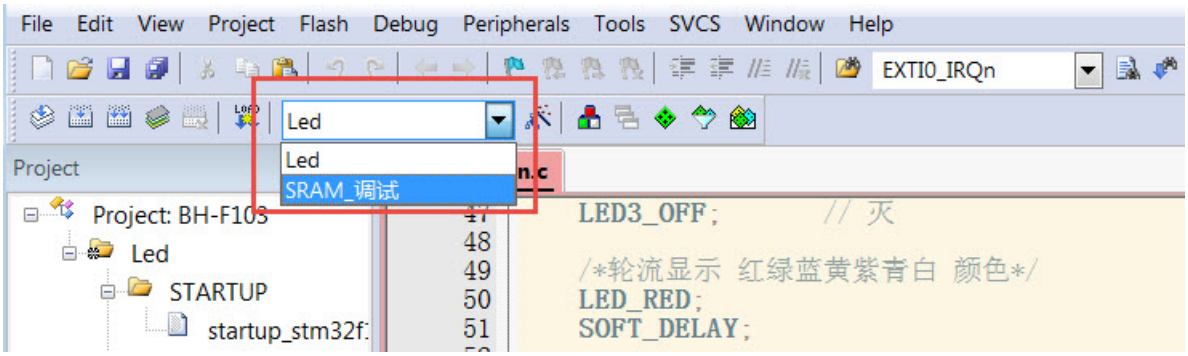


图 43-6 切换工程

在下面的教程中我们将切换到“SRAM_ 调试”版本的工程，配置出一个代码会被存储到 SRAM 的多彩流水灯工程。

40.4.2.3 配置分散加载文件

为方便讲解，本工程的分散加载只使用手动编辑的 sct 文件配置，不使用 MDK 的对话框选项配置，在“Options for Target->linker”的选项见图 *43_7*。

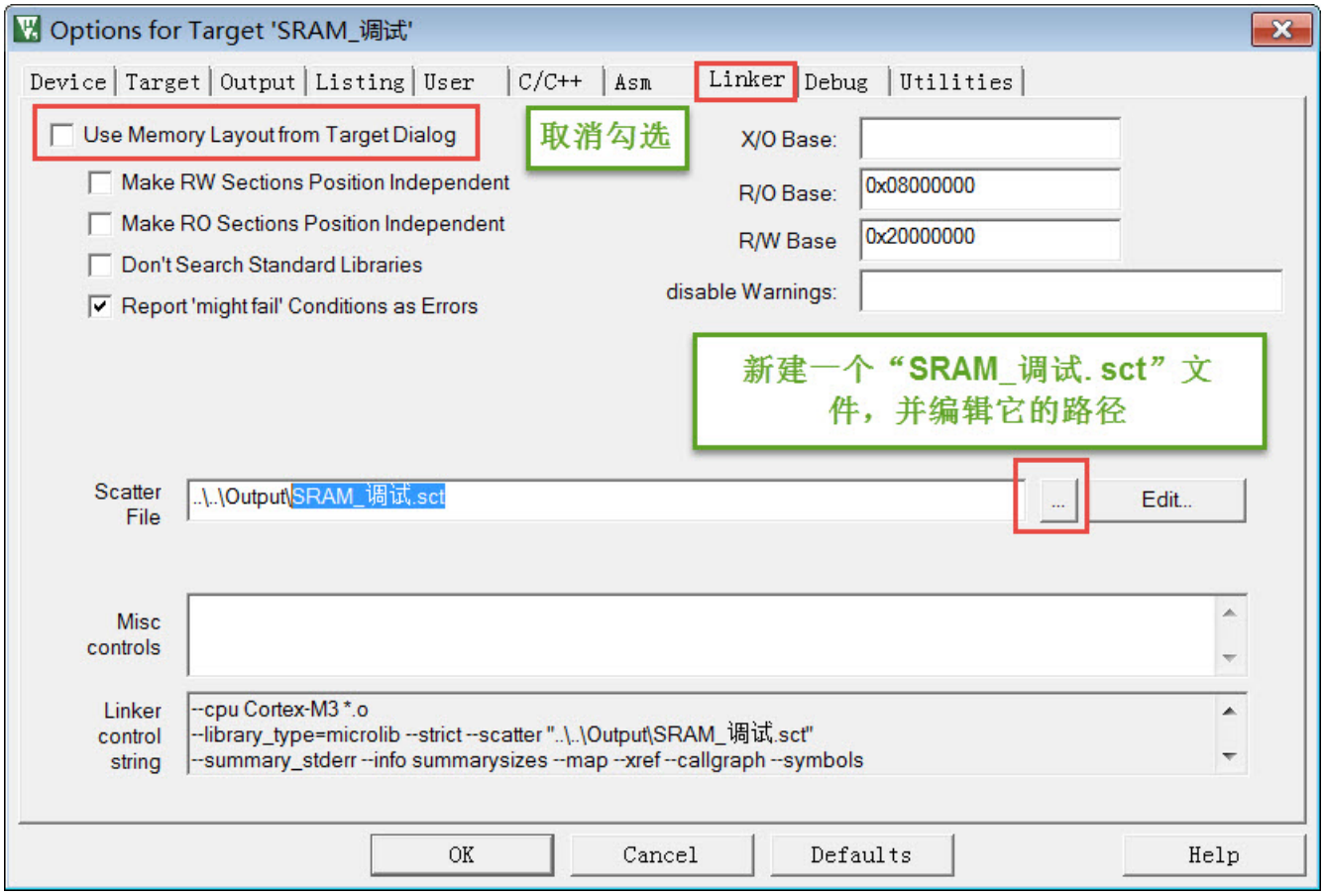


图 43-7 使用新建的“SRAM_ 调试.sct”文件

为了防止“多彩流水灯”工程的分散加载文件被影响，我们在工程的 Output 路径下新建了一个名为“SRAM_ 调试.sct”的文件，并在上图中把它配置为“SRAM_ 调试”工程专用的分散加载文件，该文件的内容见*代码清单 43_2*，若不了解分散加载文件的使用，请参考前面的章节。

列表 2: 代码清单 43-2 分散加载文件配置 (SRAM_ 调试.sct)

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x20000000 0x00008000{     ; load region size_region
```

(下页继续)

(续上页)

```
ER_IROM1 0x20000000 0x00008000{  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20008000 0x00008000{  ; RW data
   .ANY (+RW +ZI)
}
}
```

在这个分散加载文件配置中，把原本分配到内部 FLASH 空间的加载域和执行域改到了以地址 0x20000000 开始的 32KB(0x00008000) 空间，而 RW data 空间改到了以地址 0x20008000 开始的 32KB 空间 (0x00008000)。也就是说，它把 STM32 的内部 SRAM 分成了虚拟 ROM 区域以及 RW data 数据区域，链接器会根据它的配置给工程中的各种内容分配到 SRAM 地址。

在具体的应用中，虚拟 ROM 及 RW 区域的大小可根据自己的程序定制，配置完毕编译工程后可在 map 文件中查看具体的空间地址分配。

40.4.2.4 配置中断向量表

由于 startup_stm32f13xe.s 文件中的启动代码不是指定到绝对地址的，经过它由链接器决定应存储到内部 FLASH 还是 SRAM，所以 SRAM 版本工程中的启动文件不需要作任何修改。

重点在于启动文件定义的中断向量表被存储到内部 FLASH 和内部 SRAM 时，这两种情况对内核的影响是不同的，内核会根据它的“向量表偏移寄存器 VTOR”配置来获取向量表，即中断服务函数的入口。VTOR 寄存器是由启动文件中 Reset_Handle 中调用的库函数 SystemInit 配置的，见*代码清单 43_3*。

列表 3: 代码清单 43-3 SystemInit 函数 (system_stm32f103xe.c 文件)

```
void SystemInit(void)
```

(下页继续)

(续上页)

```
{
/* .. 其它代码部分省略 */

/* 配置向量表添加的偏移地址 ----*/
#ifdef VECT_TAB_SRAM
SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* 向量表存储在 SRAM */
#else
SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;/* 向量表存储在内部 FLASH */

#endif
}
```

代码中根据是否存储宏定义 VECT_TAB_SRAM 来决定 VTOR 的配置，默认情况下代码中没有定义宏 VECT_TAB_SRAM，所以 VTOR 默认情况下指示向量表是存储在内部 FLASH 空间的。

由于本工程的分散加载文件配置，在启动文件中定义的中断向量表会被分配到 SRAM 空间，所以我们要定义这个宏，使得 SystemInit 函数修改 VTOR 寄存器，向内核指示向量表被存储到内部 SRAM 空间了，见图 *43_8*，在 “Options for Target-> c/c++ ->Define” 框中输入宏 VECT_TAB_SRAM，注意它与其它宏之间要使用英文逗号分隔开。

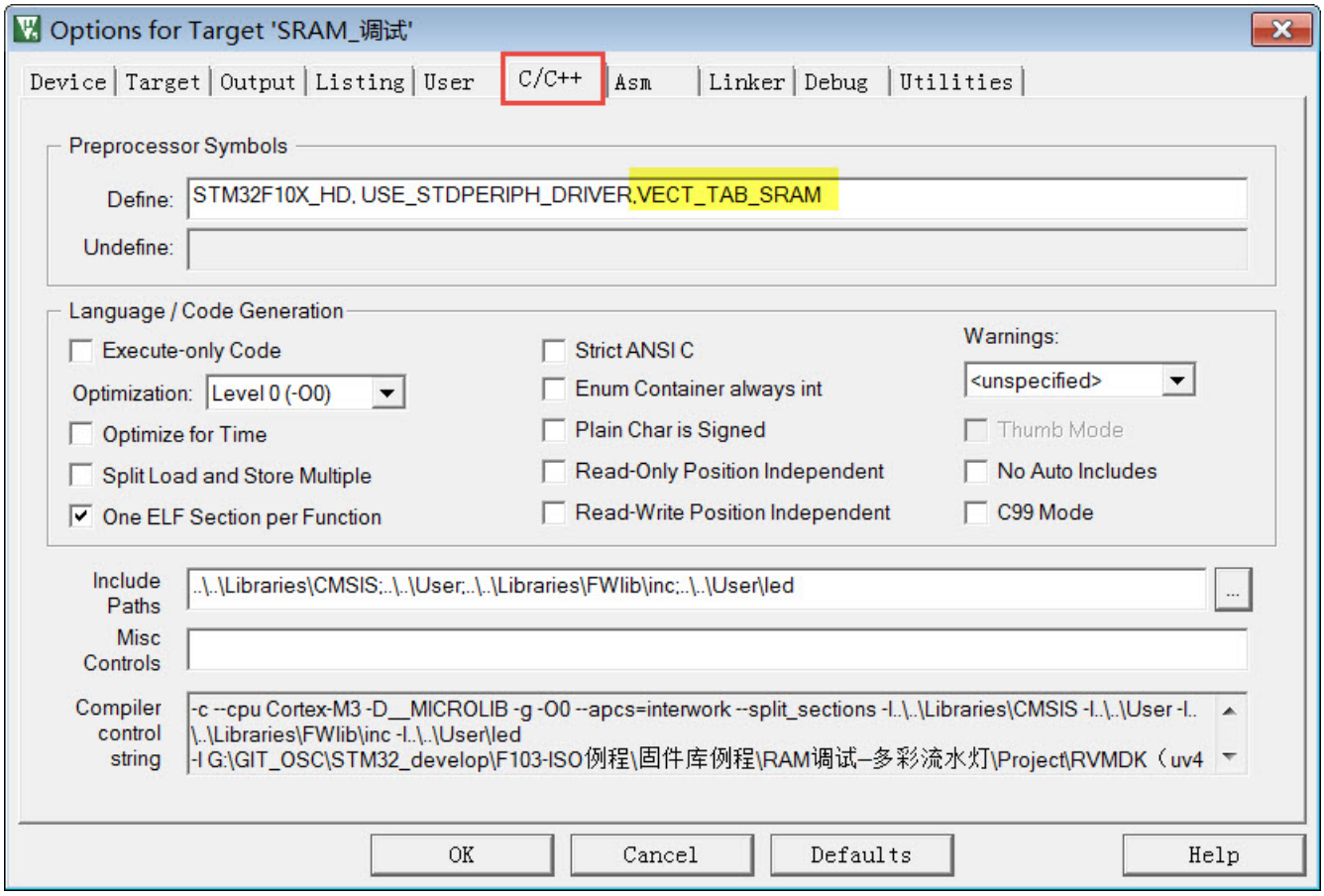


图 43-8 在 c/c++ 编译选项中加入宏 VECT_TAB_SRAM

配置完成后重新编译工程，即可生成存储到 SRAM 空间地址的代码指令。

相对于直接在文件中定义宏，使用这种方式定义该宏的好处是，若切换回 FLASH 版本的 LED 工程，该工程不受影响，若把宏定义到文件中，那么由于两个版本共用文件而受到影响。

40.4.2.5 修改 FLASH 下载配置

得到 SRAM 版本的代码指令后，为了把它下载到芯片的 SRAM 中，还需要修改下载器的配置，见图 43-9，“Options for Target->Utilities->Settings”中的选项。

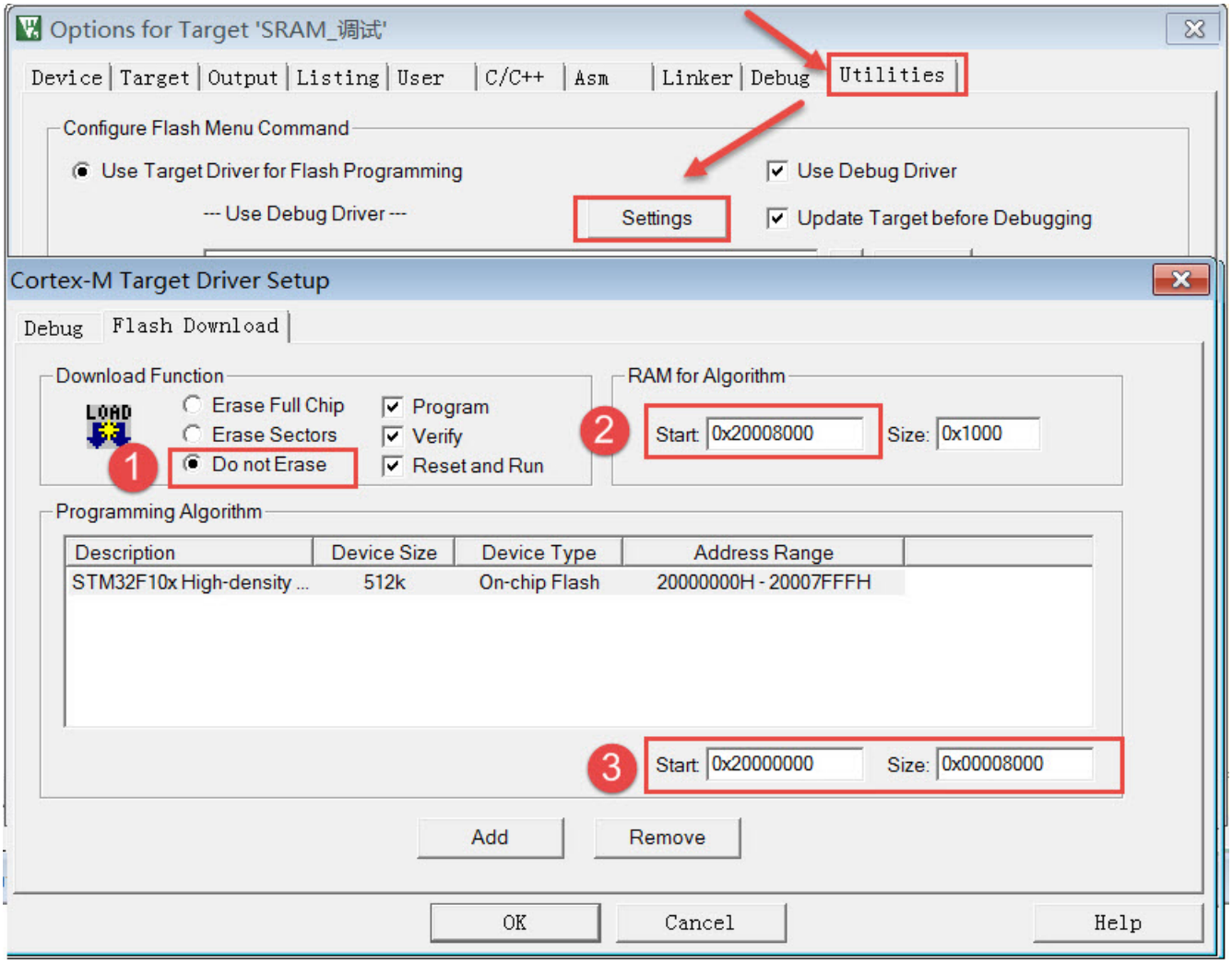


图 43-9 下载配置

这个配置对话框原本是用于设置芯片内部 FLASH 信息的，当我们点击 MDK 的 （下载、LOAD）或 （调试）按钮时，它会从此处加载配置然后下载程序到 FLASH 中，而在上图中我们把它的配置修改成下载到内部 SRAM 了，各个配置的解释如下：

- 把“Download Function”中的擦除选项配置为“Do not Erase”。这是因为数据写入到内部 SRAM 中不需要像 FLASH 那样先擦除后写入。在本工程中，如果我们不选择“Do not Erase”的话，会因为擦除过程导致下载出错。
- “RAM for Algorithm”一栏是指“编程算法”(Programming Algorithm) 可使用的 RAM 空间，

下载程序到 FLASH 时运行的编程算法需要使用 RAM 空间，在默认配置中它的首地址为 0x20000000，即内部 SRAM 的首地址，但由于我们的分散加载文件配置，0x20000000 地址开始的 32KB 实际为虚拟 ROM 空间，实际的 RAM 空间是从地址 0x20008000 开始的，所以这里把算法 RAM 首地址更改为本工程中实际作为 RAM 使用的地址。若编程算法使用的 RAM 地址与虚拟 ROM 空间地址重合的话，会导致下载出错。

- “Programming Algorithm” 一栏中是设置内部 FLASH 的编程算法，编程算法主要描述了 FLASH 的地址、大小以及扇区等信息，MDK 根据这些信息把程序下载到芯片的 FLASH 中，不同的控制器芯片一般会有不同的编程算法。由于 MDK 没有内置 SRAM 的编程算法，所以我们直接在原来的基础上修改它的基地址和空间大小，把它改成虚拟 ROM 的空间信息。

从这个例子可了解到，这里的配置是跟我们的分散加载文件的实际 RAM 空间和虚拟 ROM 空间信息是一致的，若您的分散加载文件采用不同的配置，这个下载选项也要作出相应的修改，不能照抄本例子的空间信息。

这个配置是针对程序下载的，配置完成后点击 MDK 的

按钮（下载、LOAD），程序会被下载到 STM32 的内部 SRAM 中。根据前面介绍的理论知识，若给 STM32 的 BOOT0 和 BOOT1 引脚都接到高电平，那么 STM32 将被设置为 SRAM 启动，按下板子的复位键后，程序会从内部 SRAM 中加载运行。

注意：非常遗憾的是，我们在各种平台做了大量测试，发现程序虽然被下载到 SRAM 了，但复位后 STM32 的程序 PC 指针和 SP 指针却莫名奇妙地指向非预设的 ResetHandler 及栈顶位置，导致程序无法正常运行（测试时，均有使用电压表直接测量 STM32 芯片 BOOT 引脚的电压确认它们都是高电平，后面小节有给出测试得的不正常情况下，PC 和 SP 指针的值）。另外，当使用 STM32F429 芯片时，根据前面介绍的理论作类似的配置，程序下载到 SRAM 后，完全能正常运行，而在 STM32F1 系列各型号的芯片上，均无法实现。

由于直接下载到芯片上复位运行的方式无法正常工作，所以下面介绍另一种折衷的解决办法，即使用仿真器强制设置 PC 指针及 SP 指针。

40.4.2.6 指定 PC 及 SP 指针值的仿真器配置

前面介绍的下载配置主要指定了程序的下载位置，使得程序能够加载到 SRAM，而由于实际应用在 SRAM 启动方式时 PC 和 SP 指针加载不正常，因此需要使用仿真器辅助修改 PC 及 SP 指针，然后在仿真器的控制下在 SRAM 中调试运行，即在 MDK 中使用 按钮 (调试、debug) 时进行的硬件在线调试、单步运行等功能，该功能与在 FLASH 中的硬件调试一样，但针对本实验的在 SRAM 运行环境，需要对配置进行修改。配置如下：

- 添加“Download options”配置。在“Options for Target->Debug->Settings”中进行勾选“Verify Code Download”及“Download to FLASH”配置，见图 *43_10*。

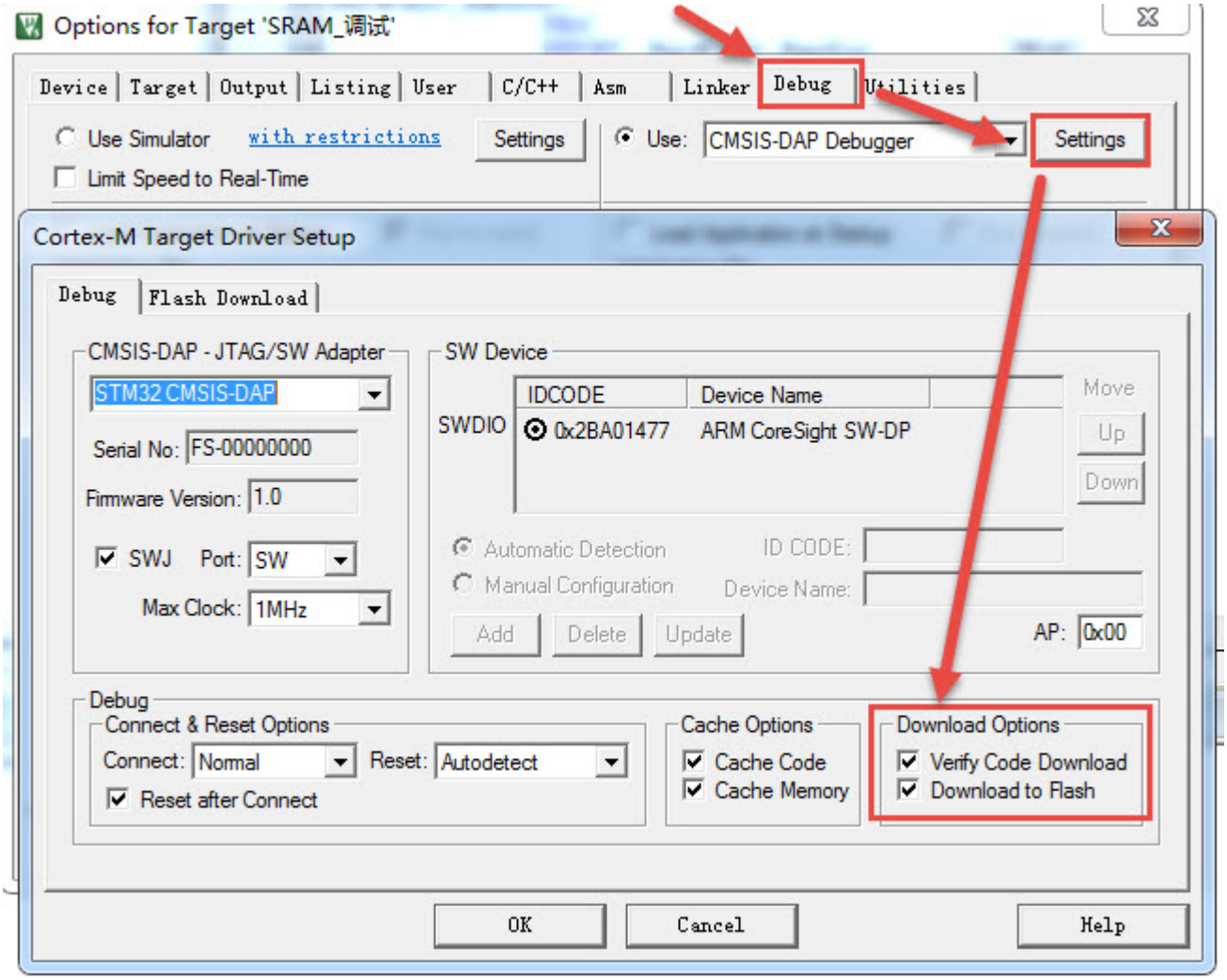


图 43-10 设置仿真前检查代码并下载程序到 FLASH 中

在图中勾选“Verify Code Download”及“Download to FLASH”选项后，当点击调试按钮后，本工程的程序会被下载到内部 SRAM 中，只有勾选了这两个选项才能正常仿真。

- 添加仿真器加载指令。见图 *43_11*，在“Options for Target->Debug”对话框中取消勾选“Load Application at startup”选项。点击“Initialization File”文本框右侧的文件浏览按钮，在弹出的对话框中新建一个名为“Debug_RAM.ini”的文件；

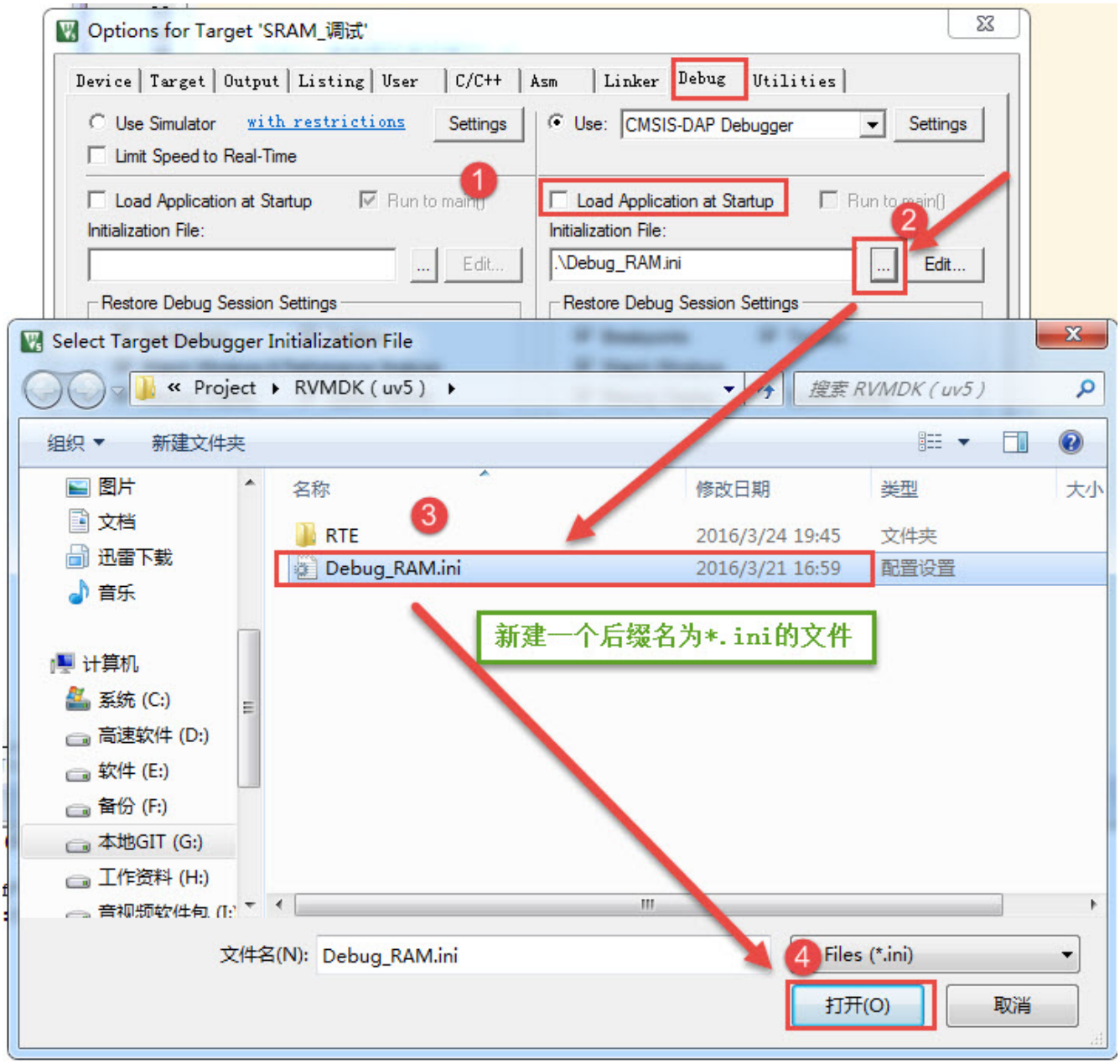


图 43-11 新建一个 ini 文件

在 Debug_RAM.ini 文件中输入如*代码清单 43_4* 中的内容。

列表 4: 代码清单 43-4 Debug_RAM.ini 文件内容

```
/*****************************************************/
/* Debug_RAM.ini: Initialization File for Debugging from Internal RAM
 ↪  */
/*****************************************************/
/* This file is part of the uVision/ARM development tools.
 ↪  */
/* Copyright (c) 2005-2014 Keil Software. All rights reserved.
 ↪  */
/* This software may only be used under the terms of a valid, current,
 ↪  */
/* end user licence from KEIL for a compatible version of KEIL software
 ↪  */
/*development tools. Nothing else gives you the right to use this software
 ↪*/
/*****************************************************/

FUNC void Setup (void) {
SP = _RDWORD(0x20000000); // 设置栈指针 SP，把 0x20000000 地址中的内容赋值到 SP。
PC = _RDWORD(0x20000004); // 设置程序指针 PC，把 0x20000004 地址中的内容赋值到 PC。

_WDWORD(0xE000ED08, 0x20000000);  // Setup Vector Table Offset Register
}

LOAD %L INCREMENTAL                    // 下载 axf 文件到 RAM
Setup();                               //调用上面定义的 setup 函数设置运行环境

//g, main   //跳转到 main 函数，本示例调试时不需要从 main 函数执行，注释掉了，程序从启
动代码开始执行
```

上述配置过程是控制 MDK 执行仿真器的脚本文件 Debug_RAM.ini，而该脚本文件在下载了程序到 SRAM 后，初始化了 SP 指针 (即 MSP) 和 PC 指针分别指向了 0x20000000 和 0x20000004，这

样的操作强制芯片上电后从该地址获取 SP 和 PC 的内容，而根据程序配置，该地址分别存储了栈顶和 ResetHandler 的地址值。

有了这样的配置，就能解决 STM32F1 系列芯片 SRAM 启动方式 SP 和 PC 指针乱指的问题了，点击调试按钮 ，即可启动仿真过程，由于强制配置了指针，所以即使 BOOT0 和 BOOT1 引脚不设置为 SRAM 启动也能正常仿真，但点击下载按钮 把程序下载到 SRAM 然后按复位依然是不能全速运行的 (这种运行方式脱离了仿真器的控制，SP 和 PC 指针无法被初始化指向正确的位置)。

经过这样的配置后，硬件仿真时与在 FLASH 仿真程序基本无异、单步运行、全速运行以及查看各种变量值等，但由于上述仿真加载的指令只在点击调试按钮后才会运行一次，所以在调试时如果点击复位，程序的 SP 和 PC 仍然会指向错误的位置，所以每次希望复位程序时，都需要重新点击 按钮加载调试。

上述 Debug_RAM.ini 文件是从 STM32F1 的 MDK 芯片包里复制过来的，若您感兴趣可到 MDK 安装目录搜索该文件名，该文件的语法可以从 MDK 的帮助手册的“µVision User’s Guide->Debug Commands”章节学习。

40.4.2.7 关于复位后 PC 和 SP 指针的调试情况

为了更好地了解 RAM 调试的运行情况，在仿真时，可以点击 MDK 仿真环境左栏底部的“Registers”按钮查看内核寄存器的情况。

当仿真器配置使用“Debug_RAM.ini”文件强制设置 SP 和 PC 寄存器的加载地址时，它们都获取到了正常的栈顶和 ResetHandler 的地址值，见图 *43_12*。

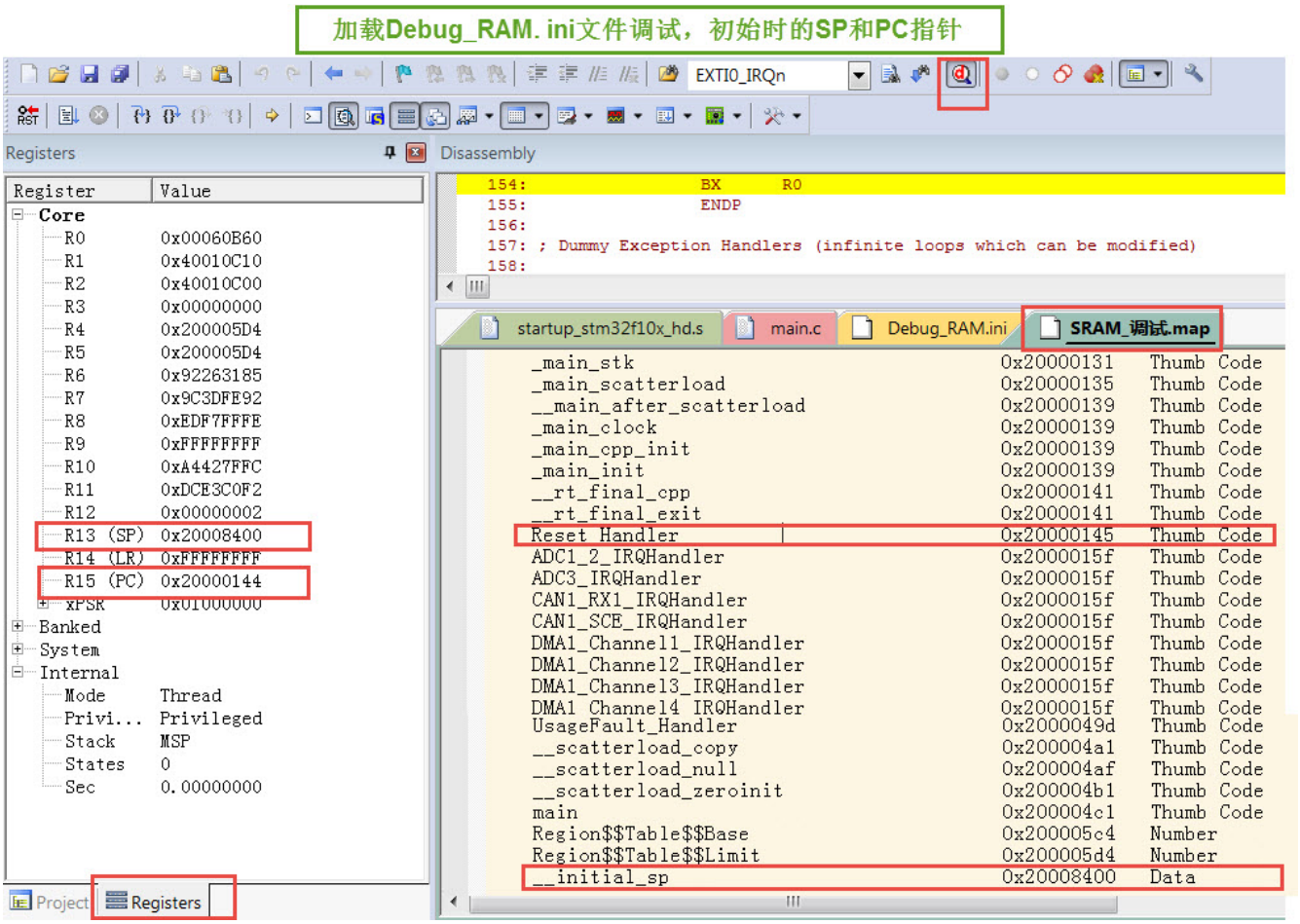
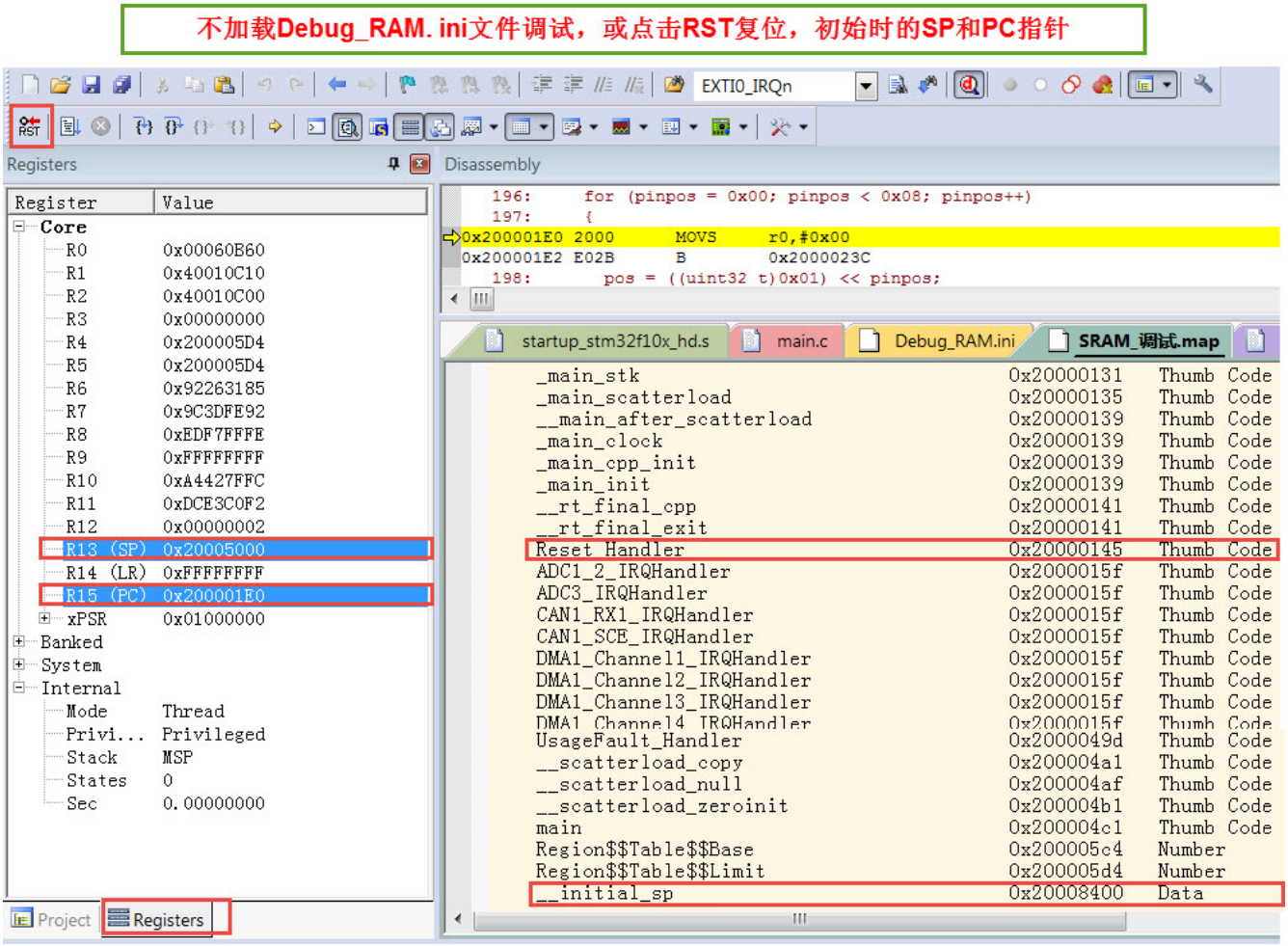


图 43-12 加载 Debug_RAM.ini 文件调试，初始时的 SP 和 PC 指针正常

从图中的 map 文件可以了解到，Reset_Handler 程序存储的地址值为 0x20000145（PC 指针加载时会减 1，即从 0x20000144 可加载到正常的 ResetHandler 代码），栈顶指针 __initial_sp 的地址值为 0x20008400。

查看图中左栏的 PC 与 SP 寄存器的值，正好是 0x20000144 和 0x20008400，也正因如此，使用这种方式调试时，SRAM 中的程序能正常运行。

反观后面的图 *43_13*，它呈现的是当仿真器 ** 不使用 ** “Debug_RAM.ini” 文件强制设置 SP 和 PC 寄存器的加载地址，或者点击复位键时的情况，这时 PC 寄存器的值为 0x200001e0，经查询该地址存储的是 GPIO_Init 函数的部分指令，而 SP 寄存器的值为 0x20005000，指向未知的存储区域。由于 PC 寄存器存储的地址不是 ResetHandler，当程序运行时，就无法按照预定的设计运行，导致出错。



这种情况，不使用仿真器强制配置，PC 和 SP 寄存器的值是由芯片复位后自动加载的，而图中的实验运行平台 BOOT0 和 BOOT1 已设置为高电平，即 SRAM 启动方式，根据 STM32 的说明，它本应直接从 0x20000000 及 0x20000004 地址加载到正常的栈顶和 ResetHandler 地址，而且经我们调试查询到在 SRAM 的这两个地址中确实存储了正确的栈顶和 ResetHandler 地址。所以无法得知 STM32F1 系列的芯片为何加载到错误的数据。

由于存在这个无法解决的问题，在 STM32F1 系列的芯片只能将就使用这种调试方式来使程序在 SRAM 中运行了。

40.4.2.8 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，把 BOOT0 及 BOOT1 引脚使用跳帽连接到低电平，在电脑端打开串口调试助手，点击下载按钮，把编译好的程序下载到芯片的内部 SRAM 中，复位运行，观察流水灯是否正常闪烁；给开发板断电再重新上电，观察程序是否还能正常运行。

第 41 章　读写内部 FLASH

本章参考资料：《STM32F10x 闪存编程参考手册》《STM32F10x 中文参考手册》、《STM32F10x 规格书》、《Cortex-M3 权威指南》。

41.1 STM32 的内部 FLASH 简介

在 STM32 芯片内部有一个 FLASH 存储器，它主要用于存储代码，我们在电脑上编写好应用程序后，使用下载器把编译后的代码文件烧录到该内部 FLASH 中，由于 FLASH 存储器的内容在掉电后不会丢失，芯片重新上电复位后，内核可从内部 FLASH 中加载代码并运行，见图 *44_1*。

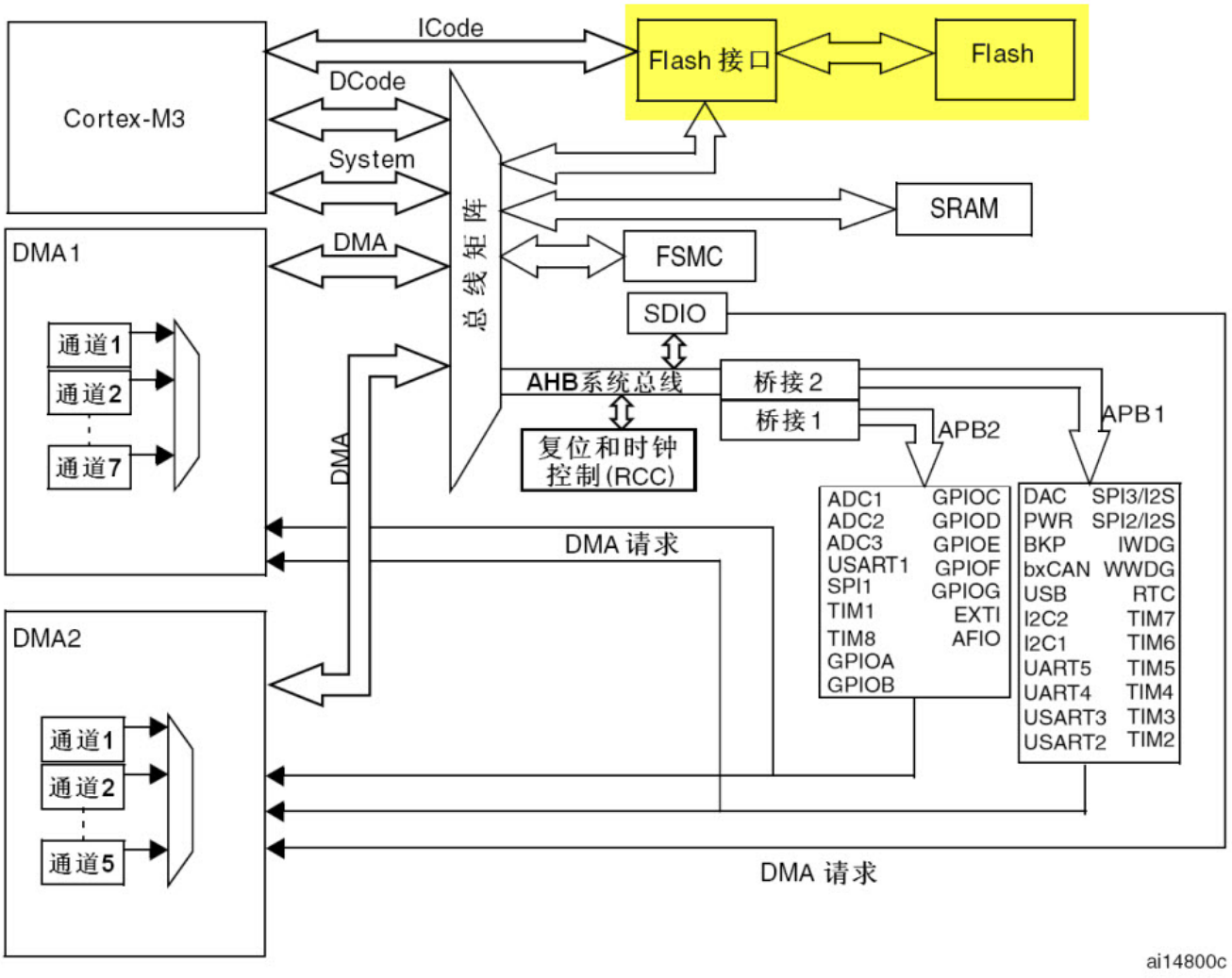


图 44-1 STM32 的内部框架图

除了使用外部的工具（如下载器）读写内部 FLASH 外，STM32 芯片在运行的时候，也能对自身的内部 FLASH 进行读写，因此，若内部 FLASH 存储了应用程序后还有剩余的空间，我们可以把它像外部 SPI-FLASH 那样利用起来，存储一些程序运行时产生的需要掉电保存的数据。

由于访问内部 FLASH 的速度要比外部的 SPI-FLASH 快得多，所以在紧急状态下常常会使用内部 FLASH 存储关键记录；为了防止应用程序被抄袭，有的应用会禁止读写内部 FLASH 中的内容，或者在第一次运行时计算加密信息并记录到某些区域，然后删除自身的部分加密代码，这些应用都涉及到内部 FLASH 的操作。

41.1.1 内部 FLASH 的构成

STM32 的内部 FLASH 包含主存储器、系统存储器以及选项字节区域，它们的地址分布及大小见表 44-1（在《STM32 参考手册》中没有关于其内部 FLASH 的说明，需要了解这些内容时，要查阅《STM32F10x 闪存编程参考手册》）。

表 44-1 STM32 大容量产品内部 FLASH 的构成（摘自《STM32F10x 闪存编程参考手册》）

区域	名称	块地址	大小
主存储器	页 0	0x0800 0000 - 0x0800 07FF	2 Kbytes
	页 1	0x0800 0800 - 0x0800 0FFF	2 Kbytes
	页 2	0x0800 1000 - 0x0800 17FF	2 Kbytes
	页 3	0x0800 1800 - 0x0800 FFFF	2 Kbytes
	.	.	.
	.	.	.
	.	.	.
	页 255	0x0807 F800 - 0x0807 FFFF	2 Kbytes
系统存储区	0x1FFF F000 - 0x1FFF F7FF	2 Kbytes	
选项字节	0x1FFF F800 - 0x1FFF F80F	16 bytes	

各个存储区域的说明如下：

- 主存储器

  一般我们说 STM32 内部 FLASH 的时候，都是指这个主存储器区域，它是存储用户应用程序的空间，芯片型号说明中的 256K FLASH、512K FLASH 都是指这个区域的大小。

  主存储器分为 256 页，每页大小为 2KB，共 512KB。这个分页的概念，实质就是 FLASH 存储器的扇区，与其它 FLASH 一样，在写入数据前，要先按页（扇区）擦除。

  注意上表中的主存储器是本实验板使用的 STM32ZET6 型号芯片的参数，即 STM32F1 大容量产品。若使用超大容量、中容量或小容量产品，它们主存储器的页数量、页大小均有不同，使用的时候要注意区分。

关于 STM32 内部 FLASH 的容量类型可根据它的型号名获知，见表 44-2。

表 44-2 STM32 芯片的命名规则

型号范例	STM32 F 103 Z E T 6
家族	“STM32 “表示 32bit 的 MCU
产品类型	“F” 表示基础型
具体特性	“103” 基础型
引脚数目	“Z” 表示 144 个引脚，其他常用的为：C 表示 48 引脚，R 表示 64 引脚，V 表示 100 引脚，“Z” 表示 144 个引脚，B 表示 208 引脚，N 表示 216 引脚
FLASH 大小	E 表示 512KB，其他常用的为：4 表示 16KB（小容量 ld），6 表示 32KB（小容量 ld），8 表示 64KB（中容量 md），B 表示 128KB（中容量 md），C 表示 256 KB（大容量 hd），E 表示 512 KB（大容量 hd），F 表示 768KB（超大容量 xl），G 表示 1024KB（超大容量 xl），
封装	“T” 表示 QFP 封装，这个是最常用的封装
温度	“6” 表示温度等级为 A： -40~85°

- 系统存储区

  系统存储区是用户不能访问的区域，它在芯片出厂时已经固化了启动代码，它负责实

现串口、USB 以及 CAN 等 ISP 烧录功能。

- 选项字节

  选项字节用于配置 FLASH 的读写保护、待机/停机复位、软件/硬件看门狗等功能，这部分共 16 字节。可以通过修改 FLASH 的选项控制寄存器修改。

41.2 对内部 FLASH 的写入过程

41.2.1 解锁

由于内部 FLASH 空间主要存储的是应用程序，是非常关键的数据，为了防止误操作修改了这些内容，芯片复位后默认会给控制寄存器 FLASH_CR 上锁，这个时候不允许设置 FLASH 的控制寄存器，从而不能修改 FLASH 中的内容。

所以对 FLASH 写入数据前，需要先给它解锁。解锁的操作步骤如下：

(1) 往 FPEC 键寄存器 FLASH_KEYR 中写入 KEY1 = 0x45670123

(2) 再往 FPEC 键寄存器 FLASH_KEYR 中写入 KEY2 = 0xCDEF89AB

41.2.2 页擦除

在写入新的数据前，需要先擦除存储区域，STM32 提供了页（扇区）擦除指令和整个 FLASH 擦除 (批量擦除) 的指令，批量擦除指令仅针对主存储区。

页擦除的过程如下：

(1) 检查 FLASH_SR 寄存器中的“忙碌寄存器位 BSY”，以确认当前未执行任何 Flash 操作；

(2) 在 FLASH_CR 寄存器中，将“激活页擦除寄存器位 PER ”置 1，

(3) 用 FLASH_AR 寄存器选择要擦除的页；

(4) 将 FLASH_CR 寄存器中的“开始擦除寄存器位 STRT ”置 1，开始擦除；

(5) 等待 BSY 位被清零时，表示擦除完成。

41.2.3 写入数据

擦除完毕后即可写入数据，写入数据的过程并不是仅仅使用指针向地址赋值，赋值前还还需要配置一系列的寄存器，步骤如下：

(1) 检查 FLASH_SR 中的 BSY 位，以确认当前未执行任何其它的内部 Flash 操作；

(2) 将 FLASH_CR 寄存器中的“激活编程寄存器位 PG”置 1；

(3) 向指定的 FLASH 存储器地址执行数据写入操作，每次只能以 16 位的方式写入；

(4) 等待 BSY 位被清零时，表示写入完成。

41.3 查看工程的空间分布

由于内部 FLASH 本身存储有程序数据，若不是有意删除某段程序代码，一般不应修改程序空间的内容，所以在使用内部 FLASH 存储其它数据前需要了解哪一些空间已经写入了程序代码，存储了程序代码的扇区都不应作任何修改。通过查询应用程序编译时产生的“*.map”后缀文件，可以了解程序存储到了哪些区域，它在工程中的打开方式见图 *44_2*，也可以到工程目录中的“Listing”文件夹中找到，关于 map 文件的详细说明可参考前面的《MDK 的编译过程及文件详解》章节。

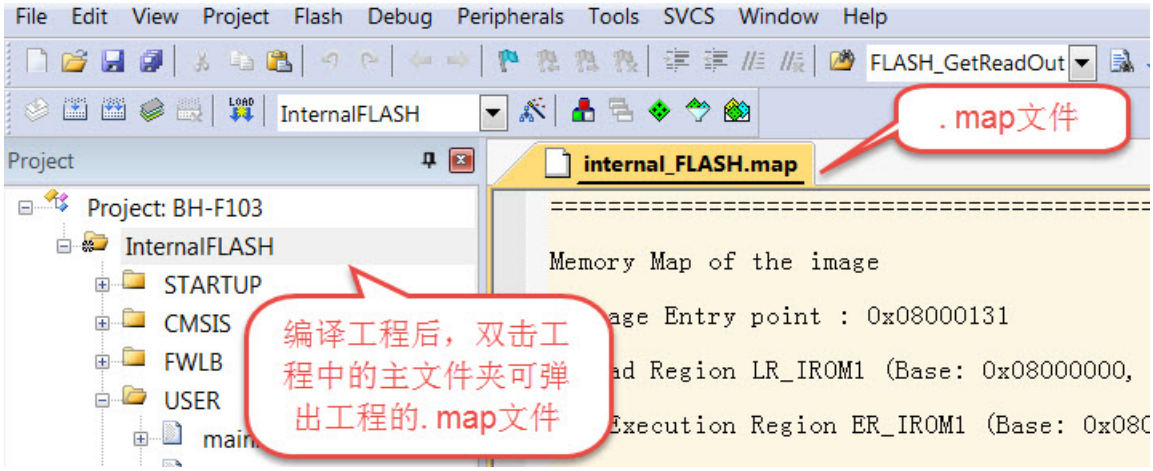


图 44-2 打开工程的.map 文件

打开 map 文件后，查看文件最后部分的区域，可以看到一段以“Memory Map of the image”开头

的记录 (若找不到可用查找功能定位)，见*代码清单 44_1*。

列表 1: 代码清单 44-1 map 文件中的存储映像分布说明

```
============================================================
Memory Map of the image //存储分布映像

Image Entry point : 0x08000131
/* 程序 ROM 加载空间 */
Load Region LR_IROM1 (Base: 0x08000000, Size: 0x000017a8, Max: 0x00080000,␣
↪ABSOLUTE)
/* 程序 ROM 执行空间 */
Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x0000177c, Max:␣
↪0x00080000, ABSOLUTE)
/* 地址分布列表 */
Base Addr    Size         Type   Attr      Idx    E Section Name       ␣
↪Object

0x08000000   0x00000130   Data   RO            3    RESET              ␣
↪startup_stm32f10x_hd.o
0x08000130   0x00000000   Code   RO          479  * .ARM.Collect$$$
↪$00000000  mc_w.l(entry.o)
0x08000130   0x00000004   Code   RO          742    .ARM.Collect$$$
↪$00000001  mc_w.l(entry2.o)
0x08000134   0x00000004   Code   RO          745    .ARM.Collect$$$
↪$00000004  mc_w.l(entry5.o)
/*... 此处省略大部分内容 */
0x080016e8   0x00000024   Code   RO          772    .text              mc_
↪w.l(init.o)
0x0800170c   0x00000010   Code   RO          483    i.__0printf$bare   mc_
↪w.l(printfb.o)
0x0800171c   0x0000000e   Code   RO          784    i.__scatterload_copy ␣
↪mc_w.l(handlers.o)
```

(下页继续)

(续上页)

```
0x0800172a   0x00000002   Code   RO          785    i.__scatterload_null ␣
↪mc_w.l(handlers.o)
0x0800172c   0x0000000e   Code   RO          786    i.__scatterload_
↪zeroinit  mc_w.l(handlers.o)
0x0800173a   0x00000022   Code   RO          490    i._printf_core       mc_
↪w.l(printfb.o)
0x0800175c   0x00000020   Data   RO          782    Region$$Table        ␣
↪anon$$obj.o
```

这一段是某工程的 ROM 存储器分布映像，在 STM32 芯片中，ROM 区域的内容就是指存储到内部 FLASH 的代码。

41.3.1 程序 ROM 的加载与执行空间

上述说明中有两段分别以“Load Region LR_ROM1”及“Execution Region ER_IROM1”开头的内容，它们分别描述程序的加载及执行空间。在芯片刚上电运行时，会加载程序及数据，例如它会从程序的存储区域加载到程序的执行区域，还把一些已初始化的全局变量从 ROM 复制到 RAM 空间，以便程序运行时可以修改变量的内容。加载完成后，程序开始从执行区域开始执行。

在上面 map 文件的描述中，我们了解到加载及执行空间的基地址 (Base) 都是 0x08000000，它正好是 STM32 内部 FLASH 的首地址，即 STM32 的程序存储空间就直接是执行空间；它们的大小 (Size) 分别为 0x000017a8 及 0x0000177c，执行空间的 ROM 比较小的原因就是因为部分 RW-data 类型的变量被拷贝到 RAM 空间了；它们的最大空间 (Max) 均为 0x00080000，即 512K 字节，它指的是内部 FLASH 的最大空间。

计算程序占用的空间时，需要使用加载区域的大小进行计算，本例子中应用程序使用的内部 FLASH 是从 0x08000000 至 (0x08000000+0x000017a8) 地址的空间区域。

41.3.2 ROM 空间分布表

在加载及执行空间总体描述之后，紧接着一个 ROM 详细地址分布表，它列出了工程中的各个段(如函数、常量数据) 所在的地址 Base Addr 及占用的空间 Size，列表中的 Type 说明了该段的类型，CODE 表示代码，DATA 表示数据，而 PAD 表示段之间的填充区域，它是无效的内容，PAD 区域往往是为了解决地址对齐的问题。

观察表中的最后一项，它的基地址是 0x0800175c，大小为 0x00000020，可知它占用的最高的地址空间为 0x0800177c，跟执行区域的最高地址 0x0000177c 一样，但它们比加载区域说明中的最高地址 0x80017a8 要小，所以我们以加载区域的大小为准。对比表 44-1 的内部 FLASH 页地址分布表，可知仅使用页 0 至页 2 就可以完全存储本应用程序，所以从页 3(地址 0x08001800) 后的存储空间都可以作其它用途，使用这些存储空间时不会篡改应用程序空间的数据。

41.4 操作内部 FLASH 的库函数

为简化编程，STM32HAL 库提供了一些库函数，它们封装了对内部 FLASH 写入数据操作寄存器的过程。

41.4.1 FLASH 解锁、上锁函数

对内部 FLASH 解锁、上锁的函数见*代码清单 44_2*。

列表 2: 代码清单 44-2 FLASH 解锁、上锁

```
/** @defgroup FLASH_Keys FLASH Keys
* @{
*/
#define FLASH_KEY1               ((uint32_t)0x45670123U)
#define FLASH_KEY2               ((uint32_t)0xCDEF89ABU)
/**
* @brief  Unlock the FLASH control register access
```

(下页继续)

(续上页)

```
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASH_Unlock(void)
{
    if ((FLASH->CR & FLASH_CR_LOCK) != RESET) {
        /* Authorize the FLASH Registers access */
        FLASH->KEYR = FLASH_KEY1;
        FLASH->KEYR = FLASH_KEY2;
    } else {
        return HAL_ERROR;
    }

    return HAL_OK;
}

/**
* @brief  Locks the FLASH control register access
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASH_Lock(void)
{
    /* Set the LOCK Bit to lock the FLASH Registers access */
    FLASH->CR |= FLASH_CR_LOCK;

    return HAL_OK;
}
```

解锁的时候，它对 FLASH_KEYR 寄存器写入两个解锁参数，上锁的时候，对 FLASH_CR 寄存器的 FLASH_CR_LOCK 位置 1。

41.4.2 设置操作位数及页擦除

解锁后擦除扇区时可调用 FLASH_EraseSector 完成，见*代码清单 44_3*。

列表 3: 代码清单 44-3 擦除扇区

```
/**
* @brief  擦除指定的页
* @param  Page_Address: 要擦除的页地址.
* @retval FLASH Status:
            可能的返回值: FLASH_BUSY, FLASH_ERROR_PG,
*           FLASH_ERROR_WRP, FLASH_COMPLETE or FLASH_TIMEOUT.
*
*
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
uint32_t *SectorError)
{
    HAL_StatusTypeDef status = HAL_ERROR;
    uint32_t index = 0;

    /* Process Locked */
    __HAL_LOCK(&pFlash);

    /* Check the parameters */
    assert_param(IS_FLASH_TYPEERASE(pEraseInit->TypeErase));

    /* Wait for last operation to be completed */
    status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

    if (status == HAL_OK) {
        /*Initialization of SectorError variable*/
```

(下页继续)

(续上页)

```
        *SectorError = 0xFFFFFFFFU;

        if (pEraseInit->TypeErase == FLASH_TYPEERASE_MASSERASE) {
            /*Mass erase to be done*/
#if defined (FLASH_OPTCR_nDBANK)
FLASH_MassErase((uint8_t) pEraseInit->VoltageRange, pEraseInit->Banks);
#else
            FLASH_MassErase((uint8_t) pEraseInit->VoltageRange);
#endif /* FLASH_OPTCR_nDBANK */

            /* Wait for last operation to be completed */
status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

    /* if the erase operation is completed, disable the MER Bit */
            FLASH->CR &= (~FLASH_MER_BIT);
        } else {
            /* Check the parameters */
        assert_param(IS_FLASH_NBSECTORS(pEraseInit->NbSectors +
pEraseInit->Sector));

            /* Erase by sector by sector to be done*/
for (index = pEraseInit->Sector; index < (pEraseInit->NbSectors +
pEraseInit->Sector); index++) {
        FLASH_Erase_Sector(index, (uint8_t) pEraseInit->VoltageRange);

                /* Wait for last operation to be completed */
status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

/*If the erase operation is completed, disable the SER Bit and SNB Bits*/
                CLEAR_BIT(FLASH->CR, (FLASH_CR_SER | FLASH_CR_SNB));

                if (status != HAL_OK) {
```

(下页继续)

(续上页)

```
/* In case of error, stop erase procedure and return the faulty sector*/
                    *SectorError = index;
                    break;
                  }
              }
          }
    }

    /* Process Unlocked */
    __HAL_UNLOCK(&pFlash);

    return status;
}
```

本函数包含两个输入参数，分别是擦除 flash 初始化结构体和返回擦除出错编码，FLASH_EraseInitTypeDef 擦除 flash 初始化结构体主要包含擦除的方式，是扇区擦除还是批量擦除，选择不同电压时实质是选择不同的数据操作位数，并且确定擦除首地址即擦除的扇区个数。函数根据输入参数配置 PSIZE 位，然后擦除扇区，擦除扇区的时候需要等待一段时间，它使用 FLASH_WaitForLastOperation 等待，擦除完成的时候才会退出 HAL_FLASHEx_Erase 函数。

41.4.3 写入数据

对内部 FLASH 写入数据不像对 SDRAM 操作那样直接指针操作就完成了，还要设置一系列的寄存器，利用 FLASH_TYPEPROGRAM_DOUBLEWORD、FLASH_TYPEPROGRAM_WORD、FLASH_TYPEPROGRAM_HALFWORD 和 FLASH_TYPEPROGRAM_BYTE 函数

可按字、半字的单位单位写入数据，见*代码清单 44_4*。

列表 4: 代码清单 44-4 写入数据

```
HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t
                                    Address, uint64_t Data)
{
    HAL_StatusTypeDef status = HAL_ERROR;
    uint8_t index = 0;
    uint8_t nbiterations = 0;

    /* Process Locked */
    __HAL_LOCK(&pFlash);

    /* Check the parameters */
    assert_param(IS_FLASH_TYPEPROGRAM(TypeProgram));
    assert_param(IS_FLASH_PROGRAM_ADDRESS(Address));

#if defined(FLASH_BANK2_END)
    if (Address <= FLASH_BANK1_END) {
#endif /* FLASH_BANK2_END */
        /* Wait for last operation to be completed */
        status = FLASH_WaitForLastOperation(FLASH_TIMEOUT_VALUE);
#if defined(FLASH_BANK2_END)
    } else {
        /* Wait for last operation to be completed */
        status = FLASH_WaitForLastOperationBank2(FLASH_TIMEOUT_VALUE);
    }
#endif /* FLASH_BANK2_END */

    if (status == HAL_OK) {
        if (TypeProgram == FLASH_TYPEPROGRAM_HALFWORD) {
            /* Program halfword (16-bit) at a specified address. */
            nbiterations = 1U;
```

(下页继续)

(续上页)

```
        } else if (TypeProgram == FLASH_TYPEPROGRAM_WORD) {
            /* Program word (32-bit = 2*16-bit) at a specified address.
                            */
            nbiterations = 2U;
        } else {
            /* Program double word (64-bit = 4*16-bit) at a specified␣
↪address. */
            nbiterations = 4U;
        }

        for (index = 0U; index < nbiterations; index++) {
            FLASH_Program_HalfWord((Address + (2U*index)), (uint16_t)(
                                Data >> (16U*index)));

#if defined(FLASH_BANK2_END)
            if (Address <= FLASH_BANK1_END) {
#endif /* FLASH_BANK2_END */
                /* Wait for last operation to be completed */
                status = FLASH_WaitForLastOperation(
                        FLASH_TIMEOUT_VALUE);

                /* If the program operation is completed, disable the PG␣
↪Bit */
                CLEAR_BIT(FLASH->CR, FLASH_CR_PG);
#if defined(FLASH_BANK2_END)
            } else {
                /* Wait for last operation to be completed */
                status = FLASH_WaitForLastOperationBank2(
                        FLASH_TIMEOUT_VALUE);

                /* If the program operation is completed, disable the PG␣
↪Bit */
```

(下页继续)

(续上页)

```
                CLEAR_BIT(FLASH->CR2, FLASH_CR2_PG);
            }
#endif /* FLASH_BANK2_END */
            /* In case of error, stop programation procedure */
            if (status != HAL_OK) {
                break;
            }
        }
    }

    /* Process Unlocked */
    __HAL_UNLOCK(&pFlash);

    return status;
}
```

看函数代码可了解到，形参依次设置了数据操作宽度，写入数据地址，写入的数据。在赋值操作后，调用了 FLASH_WaitForLastOperation 函数等待写操作完毕。

41.5 实验：读写内部 FLASH

在本小节中我们以实例讲解如何使用内部 FLASH 存储数据。

41.5.1 硬件设计

本实验仅操作了 STM32 芯片内部的 FLASH 空间，无需额外的硬件。

41.5.2 软件设计

本小节讲解的是“内部 FLASH 编程”实验，请打开配套的代码工程阅读理解。为了方便展示及移植，我们把操作内部 FLASH 相关的代码都编写到“bsp_internal_Flash.c”及“bsp_internal_Flash.h”文件中，这些文件是我们自己编写的，不属于 HAL 库的内容，可根据您的喜好命名文件。

41.5.2.1 程序设计要点

(1) 对内部 FLASH 解锁；

(2) 找出空闲页，擦除目标页；

(3) 进行读写测试。

41.5.2.2 代码分析

41.5.2.2.1 硬件定义

读写内部 FLASH 不需要用到任何外部硬件，不过在编写测试时我们要先确定内部 FLASH 的页大小以及要往哪些地址写入数据，在本工程中这些定义在 bsp_internal_Flash.h 头文件中，见*代码清单 44_5*。

列表 5: 代码清单 44-5 各个扇区的基地址 (bsp_internal_Flash.h 文件)

```
//写入的起始地址与结束地址

#define WRITE_START_ADDR ((uint32_t)0x08008000)

#define WRITE_END_ADDR ((uint32_t)0x0800C000)
```

代码中首先根据芯片类型定义了宏 FLASH_PAGE_SIZE，由于本工程使用的是 STM32ZET6 芯片，所以 FLASH_PAGE_SIZE 已经 stm32f1xx_hal_flash_ex.c 中被定义成 0x800，即 2048 字节。

另外，WRITE_START_ADDR 和 WRITE_END_ADDR 定义了后面本工程测试读写内部 FLASH 的起始地址与结束地址，这部分区域与 map 文件指示的程序本身占用的空间不重合，所以在后面修改这些地址的内容时，它不会把自身的程序修改掉。

41.5.2.2.2 读写内部 FLASH

一切准备就绪，可以开始对内部 FLASH 进行擦写，这个过程不需要初始化任何外设，只要按解锁、擦除及写入的流程走就可以了，见*代码清单 44_6*。

列表 6: 代码清单 44-6 对内部地 FLASH 进行读写测试 (bsp_internal_Flash.c 文件)

```
int InternalFlash_Test(void)
{

    uint32_t Address = 0x00;          //记录写入的地址
    uint32_t DATA_32 = 0x3210ABCD;        //记录写入的数据
    uint32_t NbrOfPage = 0x00;        //记录写入多少页
    __IO uint32_t Data32 = 0;

    uint32_t SECTORError = 0;
    TestStatus MemoryProgramStatus = PASSED;//记录整个测试结果

    static FLASH_EraseInitTypeDef EraseInitStruct;
    /* 解锁 */
    HAL_FLASH_Unlock();

    /* 计算要擦除多少页 */
    NbrOfPage = (WRITE_END_ADDR - WRITE_START_ADDR) / FLASH_PAGE_SIZE;
    EraseInitStruct.TypeErase     = FLASH_TYPEERASE_PAGES;
    EraseInitStruct.NbPages       = NbrOfPage;
    EraseInitStruct.PageAddress   = WRITE_START_ADDR;
```

(下页继续)

(续上页)

```
    if (HAL_FLASHEx_Erase(&EraseInitStruct, &SECTORError) != HAL_OK) {
        /* 擦除出错，返回，实际应用中可加入处理 */
        return -1;
    }
    /* 向内部 FLASH 写入数据 */
    Address = WRITE_START_ADDR;
    while (Address < WRITE_END_ADDR) {
        if (HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, Address, DATA_32)
            == HAL_OK) {
            Address = Address + 4;
        } else {
            /* 写入出错，返回，实际应用中可加入处理
            */
            return -1;
        }
    }

    HAL_FLASH_Lock();

    /* 检查写入的数据是否正确 */
    Address = WRITE_START_ADDR;

    while ((Address < WRITE_END_ADDR) && (MemoryProgramStatus !=
        FAILED)) {
        if ((*(__IO uint32_t*) Address) != DATA_32) {
            MemoryProgramStatus = FAILED;
        }
        Address += 4;
    }
    return MemoryProgramStatus;
}
```

该函数的执行过程如下：

(1) 调用 HAL_FLASH_Unlock 解锁；

(2) 根据起始地址及结束地址计算要擦除多少页；

(3) 使用循环调用 FLASH_ HAL_FLASHEx_Erase 擦除页，每次擦除一页；

(4) 使用循环调用 HAL_FLASH_Program 函数向起始地址至结束地址的存储区域都写入变量“Data”存储的数值数值；

(5) 调用 HAL_FLASH_Lock 上锁；

(6) 使用指针读取写入的数据内容并校验。

41.5.2.2.3 main 函数

最后我们来看看 main 函数的执行流程，见*代码清单 44_7*。

列表 7: 代码清单 44-7 main 函数 (main.c 文件)

```
int main(void)
{
    /* 配置系统时钟为 72 MHz */
    SystemClock_Config();

    /* 初始化 USART 配置模式为 115200 8-N-1*/
    DEBUG_USART_Config();
    /* 初始化 LED*/
    LED_GPIO_Config();

    LED_BLUE;
    printf("\r\n 欢迎使用野火  STM32  开发板。\r\n");
    printf("正在进行读写内部 FLASH 实验，请耐心等待\r\n");

    if (InternalFlash_Test()== PASSED) {
```

(下页继续)

(续上页)

```
        LED_GREEN;
        printf("读写内部 FLASH 测试成功\r\n");
    } else {
        printf("读写内部 FLASH 测试失败\r\n");
        LED_RED;
    }
    while (1) {
    }
}
```

main 函数中初始化了用于指示调试信息的 LED 及串口后，直接调用了 InternalFlash_Test 函数，进行读写测试并根据测试结果输出调试信息。

41.5.3 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到擦写内部 FLASH 的调试信息。

第 42 章　设置 FLASH 的读写保护及解除

本章参考资料：《STM32F10x 闪存编程参考手册》《STM32F10x 中文参考手册》、《STM32F10x 规格书》、《Cortex-M3 权威指南》。

42.1 选项字节与读写保护

在实际发布的产品中，在 STM32 芯片的内部 FLASH 存储了控制程序，如果不作任何保护措施的话，可以使用下载器直接把内部 FLASH 的内容读取回来，得到 bin 或 hex 文件格式的代码拷贝，别有用心的厂商会利用该方法山寨产品。为此，STM32 芯片提供了多种方式保护内部 FLASH 的程序不被非法读取，但在默认情况下该保护功能是不开启的，若要开启该功能，需要改写内部 FLASH 选项字节 (Option Bytes) 中的配置。

42.1.1 选项字节的内容

选项字节是一段特殊的 FLASH 空间，STM32 芯片会根据它的内容进行读写保护配置，选项字节的构成见表 45-1。

表 45-1 选项字节的构成

地址	[31:24]	[23:16]	[15:8]	[7:0]
0x1FFF F800	nUSER	USER	nRDP	RDP
0x1FFF F804	nData1	Data1	nData0	Data0
0x1FFF F808	nWRP1	WRP1	nWRP0	WRP0
0x1FFF F80C	nWRP3	WRP3	nWRP2	WRP2

STM32F103 系列芯片的选项字节有 8 个配置项，即上表中的 USER、RDP、DATA0/1 及 WRP0/1/2/3，而表中带 n 的同类项是该项的反码，即 nUSER 的值等于 (~USER)、nRDP 的值等于 (~RDP)，STM32

利用反码来确保选项字节内容的正确性。

选项字节的 8 个配置项具体的数据位配置说明见表 45-2。

表 45-2 选项字节具体的数据位配置说明

选项字节	
地址 0x1FFF F800	
位[7:0] RDP：读保护选项字节。 读保护用于保护 Flash 中存储的软件代码。 -把RDP配置为值0xA5时，内部FLASH处于无读保护状态 -把RDP配置为其它非0xA5的值时，内部的FLASH处于读保护状态	
位[23:16] USER:用户选项字节 这个字节用于配置下列功能： - 选择看门狗事件：硬件或软件 - 进入停机(STOP)模式时的复位事件 - 进入待机模式时的复位事件	
位19:23	0xF8：不用
位 18	nRST_STDBY待机模式复位事件 0：当进入待机模式时产生复位 1：进入待机模式时不产生复位
位 17	nRST_STOP停机复位事件 0：当进入停机(STOP)模式时产生复位 1：进入停机(STOP)模式时不产生复位
位 16	WDG_SW门狗事件 1：硬件看门狗 0：软件看门狗
地址0x1FFF F804	
Datax：2个字节的用户数据 这个地址可以使用选项字节的编程方式编程。	
地址0x1FFF F808-0x1FFE F80C	
WRPx(0-3)：FLASH写保护选项字节 对于小容量产品，选项字节WRPx中的每一个比特位用于保护主存储器中4个存储页 - 0：实施写保护 -1：不实施写保护 每个用户选项字节用于保护32K字节的主存储器。 WRP0：第0~31页的写保护 对于中容量产品，选项字节WRPx中的每一个比特位用于保护主存储器中4个存储页 -0：实施写保护 - 1：不实施写保护 四个用户选择字节用于保护总共128K字节的主存储器。 WRP0：第0~31页的写保护 WRP1：第32~63页的写保护 WRP2：第64~95页的写保护 WRP3：第96~127页的写保护 对于大容量产品，选择字节WRPx中的每一个比特位用于保护主存储器中2个存储页 - 0：实施写保护 - 1：不实施写保护 四个用户选择字节用于保护总共512K字节的主存储器。 WRP0：第0~15页的写保护 WRP1：第16~31页的写保护 WRP2：第32~47页的写保护 WRP3：位0~6提供第48~61页的写保护；位7提供第62~255页的写保护	

我们主要讲解选项字节配置中的 RDP 位和 WRP 位，它们分别用于配置读保护和写保护。

42.1.2 RDP 读保护

修改选项字节的 RDP 位的值可设置内部 FLASH 为以下保护级别：

- 0xA5：无保护

  这是 STM32 的默认保护级别，它没有任何读保护，读取内部 FLASH 的内容都没有任何限制。也就是说，第三方可以使用调试器等工具，获取该芯片 FLASH 中存储的程序，然后可以把获得的程序以 bin 和 hex 的格式下载到另一块 STM32 芯片中，加上 PCB 抄板技术，轻易复制出同样的产品。

- 其它值：使能读保护

  把 RDP 配置成除 0xA5 外的任意数值，都会使能读保护。在这种情况下，若使用调试功能 (使用下载器、仿真器) 或者从内部 SRAM 自举时都不能对内部 FLASH 作任何访问 (读写、擦除都被禁止)；而如果 STM32 是从内部 FLASH 自举时，它允许对内部 FLASH 的任意访问。也就是说，任何尝试从外部访问内部 FLASH 内容的操作都被禁止。例如，无法通过下载器读取它的内容，或编写一个从内部 SRAM 启动的程序，若该 SRAM 启动的程序读取内部 FLASH，会被禁止。而如果是芯片原本的内部 FLASH 程序自己访问内部 FLASH（即从 FLASH 自举的程序），是完全没有问题的，例如芯片本身的程序，若包含有指针对内部 FLASH 某个地址进行的读取操作，它能获取正常的数据。

  另外，被设置成读保护后，FLASH 前 4K 字节的空间会强制加上写保护，也就是说，即使是从 FLASH 启动的程序，也无法擦写这 4K 字节空间的内容；而对于前 4K 字节以外的空间，读保护并不影响它对其它空间的擦除/写入操作。利用这个特性，可以编写 IAP 代码 (In Application Program) 更新 FLASH 中的程序，它的原理是通过某个通讯接口获取将要更新的程序内容，然后利用内部 FLASH 擦写操作把这些内容烧录到自己的内部 FLASH 中，实现应用程序的更新，该原理类似串口 ISP 程序下载功能，只不过 ISP 这个接收数据并更新的代码由 ST 提供，且存放在系统存储区域，而 IAP 是由用户自行编写的，存放在用户自定义的 FLASH 区域，且通讯方式可根据用户自身的需求定制，如 IIC、SPI 等，只要能接收到数据均可。

- 解除保护

  当需要解除芯片的读保护时，要把选项字节的 RDP 位重新设置为 0xA5。在解除保护前，芯片会自动触发擦除主 FLASH 存储器的全部内容，即解除保护后原内部 FLASH 的代码会丢失，从而防止降级后原内容被读取到。

芯片被配置成读保护后根据不同的使用情况，访问权限不同，总结。

表 45-3 读保护模式下不同区域的访问限制

存储区	保护设置	从 RAM 自举（SRAM 调试模式）	从内部 FLASH 自举				
	读	写	擦除	读	写	擦除	
主 FLASH 前 4K 字节	读保护	否	是	否	否		
主 FLASH 前 4K 字节外的空间	读保护	否	是				
选项字节	读保护	是	是				

42.1.3 WRP 写保护

使用选项字节的 WRP0/1/2/3 可以设置主 FLASH 的写保护，防止它存储的程序内容被修改。

- 设置写保护

  写保护的配置一般以 4K 字节为单位，除 WRP3 的最后一位比较特殊外，每个 WRP 选项字节的一位用于控制 4K 字节的写访问权限，把对应 WRP 的位置 0 即可把它匹配的空间加入写保护。被设置成写保护后，主 FLASH 中的内容使用任何方式都不能被擦除和写入，写保护不会影响读访问权限，读访问权限完全由前面介绍的读保护设置限制。

- 解除写保护

  解除写保护是逆过程，把对应 WRP 的位置 1 即可把它匹配的空间解除写保护。解除写保护后，主 FLASH 中的内容不会像解读保护那样丢失，它会被原样保留。

42.2 修改选项字节的过程

根据前面的说明，修改选项字节的内容可修改读写保护配置，不过选项字节复位后的默认状态是始终可以读但被写保护的，因此它具有类似前面《读写内部 FLASH》章节提到的 FLASH_CR 寄存器的访问限制，要想修改，需要先对 FLASH_OPTKEYR 寄存器写入解锁编码。由于修改选项字节时也需要访问 FLASH_CR 寄存器，所以同样也要对 FLASH_KEYR 写入解锁编码。

修改选项字节的整个过程总结如下：

1. 解除 FLASH_CR 寄存器的访问限制
   - 往 FPEC 键寄存器 FLASH_KEYR 中写入 KEY1 = 0x45670123
   - 再往 FPEC 键寄存器 FLASH_KEYR 中写入 KEY2 = 0xCDEF89AB
2. 解除对选项字节的访问限制
   - 往 FLASH_OPTKEYR 中写入 KEY1 = 0x45670123
   - 再往 FLASH_OPTKEYR 中写入 KEY2 = 0xCDEF89AB
3. 配置 FLASH_CR 的 OPTPG 位，准备修改选项字节
4. 直接使用指针操作修改选项字节的内容，根据需要修改 RDP、WRP 等内容
5. 对于读保护的解除，由于它会擦除 FLASH 的内容，所以需要检测状态寄存器标志位以确认 FLASH 擦除操作完成。
6. 若是设置读保护及其解除，需要给芯片重新 **上电复位**，以使新配置的选项字节生效；对于设置写保护及其解除，需要给芯片进行 **系统复位**，以使新配置的选项字节生效。

42.3 操作选项字节的库函数

为简化编程，STM32HAL 库提供了一些库函数，它们封装了前面介绍的修改选项字节时的操作过程。

42.3.1 选项字节上锁、解锁

简化编程，STM32 HAL 库提供了一些库函数，它们封装了修改选项字节时操作寄存器的过程，代码清单 45_1。

列表 1: 代码清单 45-1 选项字节解锁、上锁

```
#define FLASH_OPT_KEY1           ((uint32_t)0x08192A3BU)
#define FLASH_OPT_KEY2           ((uint32_t)0x4C5D6E7FU)
/**
* @brief  Unlock the FLASH Option Control Registers access.
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASH_OB_Unlock(void)
{
   if ((FLASH->OPTCR & FLASH_OPTCR_OPTLOCK) != RESET) {
      /* Authorizes the Option Byte register programming */
      FLASH->OPTKEYR = FLASH_OPT_KEY1;
      FLASH->OPTKEYR = FLASH_OPT_KEY2;
   } else {
      return HAL_ERROR;
   }

   return HAL_OK;
}

/**
* @brief  Lock the FLASH Option Control Registers access.
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASH_OB_Lock(void)
{
/* Set the OPTLOCK Bit to lock the FLASH Option Byte Registers access */
```

(下页继续)

(续上页)

```
    FLASH->OPTCR |= FLASH_OPTCR_OPTLOCK;

    return HAL_OK;
}
```

解锁的时候，它对 FLASH_OPTCR 寄存器写入两个解锁参数，上锁的时候，对 FLASH_OPTCR 寄存器的 FLASH_OPTCR_OPTLOCK 位置 1。

42.3.2 设置选项字节函数

解锁后设置选项字节寄存器可以先初始化 FLASH_OBProgramInitTypeDef 结构体，然后调用 HAL_FLASHEx_OBProgram 完成。

列表 2: 代码清单 45-2 设置选项字节函数

```
/**
* @brief  Program option bytes
* @param  pOBInit: pointer to an FLASH_OBInitStruct structure that
*         contains the configuration information for the programming.
*
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASHEx_OBProgram(
FLASH_OBProgramInitTypeDef *pOBInit)
{
    HAL_StatusTypeDef status = HAL_ERROR;

    /* Process Locked */
    __HAL_LOCK(&pFlash);

    /* Check the parameters */
```

(下页继续)

(续上页)

```
  assert_param(IS_OPTIONBYTE(pOBInit->OptionType));

  /* Write protection configuration */
  if ((pOBInit->OptionType & OPTIONBYTE_WRP) == OPTIONBYTE_WRP) {
    assert_param(IS_WRPSTATE(pOBInit->WRPState));
    if (pOBInit->WRPState == OB_WRPSTATE_ENABLE) {
        /*Enable of Write protection on the selected Sector*/
        status = FLASH_OB_EnableWRP(pOBInit->WRPSector);
      } else {
        /*Disable of Write protection on the selected Sector*/
        status = FLASH_OB_DisableWRP(pOBInit->WRPSector);
      }
  }

  /* Read protection configuration */
  if ((pOBInit->OptionType & OPTIONBYTE_RDP) == OPTIONBYTE_RDP) {
      status = FLASH_OB_RDP_LevelConfig(pOBInit->RDPLevel);
  }

  /* USER  configuration */
  if ((pOBInit->OptionType & OPTIONBYTE_USER) == OPTIONBYTE_USER) {
#if defined (FLASH_OPTCR_nDBANK)
      status = FLASH_OB_UserConfig(pOBInit->USERConfig & OB_WWDG_SW,
                                   pOBInit->USERConfig & OB_IWDG_SW,
                                   pOBInit->USERConfig & OB_STOP_NO_RST,
                                   pOBInit->USERConfig & OB_STDBY_NO_RST,
                                   pOBInit->USERConfig & OB_IWDG_STOP_ACTIVE,
                                   pOBInit->USERConfig & OB_IWDG_STDBY_ACTIVE,
                                   pOBInit->USERConfig & OB_NDBANK_SINGLE_BANK,
                                   pOBInit->USERConfig & OB_DUAL_BOOT_DISABLE);
#else
      status = FLASH_OB_UserConfig(pOBInit->USERConfig & OB_WWDG_SW,
```

(下页继续)

(续上页)

```
                                        pOBInit->USERConfig & OB_IWDG_SW,
                                        pOBInit->USERConfig & OB_STOP_NO_RST,
                            pOBInit->USERConfig & OB_STDBY_NO_RST,
                            pOBInit->USERConfig & OB_IWDG_STOP_ACTIVE,
                            pOBInit->USERConfig & OB_IWDG_STDBY_ACTIVE);
#endif /* FLASH_OPTCR_nDBANK */
   }

   /* BOR Level  configuration */
   if ((pOBInit->OptionType & OPTIONBYTE_BOR) == OPTIONBYTE_BOR) {
         status = FLASH_OB_BOR_LevelConfig(pOBInit->BORLevel);
   }

   /* Boot 0 Address configuration */
   if ((pOBInit->OptionType & OPTIONBYTE_BOOTADDR_0) ==
OPTIONBYTE_BOOTADDR_0) {
         status = FLASH_OB_BootAddressConfig(OPTIONBYTE_BOOTADDR_0,
pOBInit->BootAddr0);
   }

   /* Boot 1 Address configuration */
   if ((pOBInit->OptionType & OPTIONBYTE_BOOTADDR_1) ==
OPTIONBYTE_BOOTADDR_1) {
         status = FLASH_OB_BootAddressConfig(OPTIONBYTE_BOOTADDR_1,
pOBInit->BootAddr1);
   }

   /* Process Unlocked */
   __HAL_UNLOCK(&pFlash);
   return status;
}
```

该函数根据输入选项字节结构体 FLASH_OBProgramInitTypeDef 参数设置寄存器响应的位，特别注意，其注释警告了若 RDPLevel 位配置成 OB_RDP_LEVEL_2 会无法恢复。

42.3.3 写入选项字节

调用上一步骤中的函数配置寄存器后，还要调用*代码清单 45_03* 中的 HAL_FLASH_OB_Launch 函数把寄存器的内容写入到选项字节中。

列表 3: 代码清单 45-03 写入选项字节

```
/**
* @brief  Launch the option byte loading.
* @retval HAL Status
*/
HAL_StatusTypeDef HAL_FLASH_OB_Launch(void)
{
   /* Set the OPTSTRT bit in OPTCR register */
   FLASH->OPTCR |= FLASH_OPTCR_OPTSTRT;

   /* Wait for last operation to be completed */
   return (FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE));
}
```

该函数设置 FLASH_OPTCR_OPTSTRT 位后调用了 FLASH_WaitForLastOperation 函数等待写入完成，并返回写入状态，若操作正常，它会返回 FLASH_COMPLETE。

42.4 实验：设置读写保护及解除

在本实验中我们将以实例讲解如何修改选项字节的配置，设置读写保护及解除。

本实验要进行的操作比较特殊，由于设置成读写保护状态后，若不解除保护状态或者解除代码工作不正常，将无法给芯片的 FLASH 下载新的程序，所以本程序在开发过程中使用内部 SRAM

调试的方式开发，便于测试程序（读写保护只影响 FLASH，SRAM 调试时程序下载到 SRAM 中，不受影响）。工程中，提供了 FLASH 和 SRAM 调试的版本，见图 *45_1*，

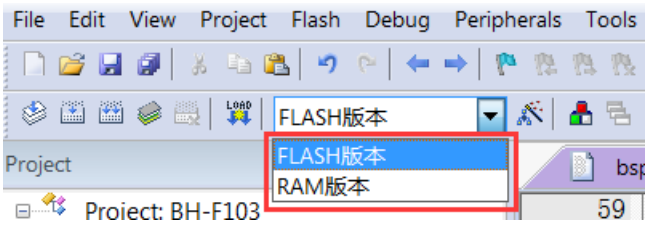


图 45-1 两种版本的程序

工程的 FLASH 版本程序包含完整的保护及解除方案，程序下载到内部 FLASH 后，它自身可以正常地进行保护及解除。另外，在学习过程中如果您想亲自修改该代码进行测试，也不用担心把解除操作的代码修改至工作不正常而导致芯片无法解锁报废，处于这种情况时，只要使用本工程的 SRAM 版本下载到芯片中，即可实现解锁。只要具备前面章节介绍的 SRAM 调试知识并备份了 SRAM 版本的工程即可大胆尝试。

42.4.1 硬件设计

本实验完全针对内部 FLASH 的操作，对外部硬件无特殊要求。即使是在 SRAM 调试模式下，由于是使用 Debug 强制加载 PC 和 SP 指针，所以也无需设置 BOOT0 和 BOOT1 的引脚。

42.4.2 软件设计

本实验的工程名称为“设置读写保护与解除”，学习时请打开该工程配合阅读。为了方便展示及移植，我们把读写保护相关的代码都编写到“bsp_readWriteProtect.c”及“bsp_readWriteProtect.h”文件中，这些文件是我们自己编写的，不属于 HAL 库的内容，可根据您的喜好命名文件。

42.4.2.1 代码分析

42.4.2.1.1 设置写保护及解除

针对读保护及其解除，本工程定义了 FLASH_Test 函数，见*代码清单 45_3*。

列表 4: 代码清单 45-3 配

```
void FLASH_Test(void)
{
   printf("测试开始\r\n");
   /* 初始化测试状态 */
   MemoryProgramStatus = PASSED;

   /* 解锁 Flash 以启用闪存控制寄存器访问 */
   HAL_FLASH_Unlock();

   /* 解锁选项字节 */
   HAL_FLASH_OB_Unlock();

   /* 获取页面写保护状态 */
   HAL_FLASHEx_OBGetConfig(&OptionsBytesStruct);

#ifdef WRITE_PROTECTION_DISABLE
   /* 检查所需页面是否已被写保护 */
   if ((OptionsBytesStruct.WRPPage & FLASH_PAGE_TO_BE_PROTECTED) !=
         FLASH_PAGE_TO_BE_PROTECTED) {
         /* 恢复写保护页面 */
         OptionsBytesStruct.OptionType   = OPTIONBYTE_WRP;
         OptionsBytesStruct.WRPState     = OB_WRPSTATE_DISABLE;
         OptionsBytesStruct.WRPPage = FLASH_PAGE_TO_BE_PROTECTED;
         if (HAL_FLASHEx_OBProgram(&OptionsBytesStruct) != HAL_OK) {
            while (1) {
```

(下页继续)

(续上页)

```
                LED1_ON
                printf("运行失败\r\n");
            }
        }

        /* 生成系统重置以加载新选项字节值 */
        HAL_FLASH_OB_Launch();
    }
#elif defined WRITE_PROTECTION_ENABLE
    /* 检查所需页面是否尚未写保护 */
    if (((~OptionsBytesStruct.WRPPage) & FLASH_PAGE_TO_BE_PROTECTED )
        != FLASH_PAGE_TO_BE_PROTECTED) {
        /* 启用页面写保护 */
        OptionsBytesStruct.OptionType = OPTIONBYTE_WRP;
        OptionsBytesStruct.WRPState   = OB_WRPSTATE_ENABLE;
        OptionsBytesStruct.WRPPage    = FLASH_PAGE_TO_BE_PROTECTED;
        if (HAL_FLASHEx_OBProgram(&OptionsBytesStruct) != HAL_OK) {
            while (1) {
                LED1_ON
                printf("运行失败\r\n");
            }
        }

        /* 生成系统重置以加载新选项字节值 */
        HAL_FLASH_OB_Launch();
    }
#endif /* WRITE_PROTECTION_DISABLE */

    /* 锁定选项字节 */
    HAL_FLASH_OB_Lock();

    /* 所选页面未被写保护 */
```

(下页继续)

(续上页)

```
  if ((OptionsBytesStruct.WRPPage & FLASH_PAGE_TO_BE_PROTECTED) !=
        0x00) {
        /* 填写 EraseInit 结构 */
        EraseInitStruct.TypeErase   = FLASH_TYPEERASE_PAGES;
        EraseInitStruct.PageAddress = FLASH_USER_START_ADDR;
        EraseInitStruct.NbPages     = (FLASH_USER_END_ADDR -
                                      FLASH_USER_START_ADDR)
                                      /FLASH_PAGE_SIZE;

        if (HAL_FLASHEx_Erase(&EraseInitStruct, &PageError) != HAL_OK)
           {
           /*
           页面擦除时发生错误。
           用户可以在这里添加一些代码来处理这个错误

           PageError 将包含有问题的页面，然后知道此页面
           上的代码错误，
           用户可以调用函数'HAL_FLASH_GetError ()'
           */
           while (1) {
              LED1_ON
              printf("运行失败\r\n");
           }
        }

        /* 由 FLASH_USER_START_ADDR 和 FLASH_USER_END_ADDR 定义的地址
        处的 DATA_32 FLASH 字程序 */
        Address = FLASH_USER_START_ADDR;
        while (Address < FLASH_USER_END_ADDR) {
           if (HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, Address,
              DATA_32) == HAL_OK) {
              Address = Address + 4;
```

(下页继续)

(续上页)

```
            } else {
                while (1) {
                      LED1_ON
                      printf("运行失败\r\n");
                }
            }
        }

        /* 检查书面数据的正确性 */
    Address = FLASH_USER_START_ADDR;

        while (Address < FLASH_USER_END_ADDR) {
            if ((*(__IO uint32_t*) Address) != DATA_32) {
                MemoryProgramStatus = FAILED;
            }
            Address += 4;
        }
} else {
        /* 所需页面受写保护 */
        /* 检查是否允许在此页面中写入 */
        Address = FLASH_USER_START_ADDR;
        if (HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, Address, DATA_32)
            != HAL_OK) {
            /* 编程期间返回错误。 */
            /* 检查 WRPERR 标志是否设置良好 */
            if (HAL_FLASH_GetError() == HAL_FLASH_ERROR_WRP) {
                MemoryProgramStatus = FAILED;
            } else {
                while (1) {
                      LED1_ON;
                      printf("运行失败\r\n");
                }
```

(下页继续)

(续上页)

```
            }
        } else {
            while (1) {
                LED1_ON
                printf("运行失败\r\n");
            }
        }
    }
    HAL_FLASH_Lock();

    /* 检查程序数据是否存在问题 */
    if (MemoryProgramStatus == PASSED) {
          LED2_ON
    } else {
          while (1) {
              LED1_ON
              printf("运行失败\r\n");
          }
    }
    while (1) {
    }
}
```

本函数主要演示写保护和解除功能，若芯片本身处于读保护状态，则解除保护，若芯片本身处于无读保护状态，则设置加入读保护。

42.4.2.1.2 main 函数

最后来看看本实验的 main 函数，见*代码清单 45_4*。

列表 5: 代码清单 45-4 main 函数

```
//【 !!】注意事项:
//1. 当芯片处于读写保护状态时，均无法下载新的程序，需要先解除保护状态后再下载
//2. 本工程包含两个版本，可在 MDK 的“Load”下载按钮旁边的下拉框选择:
//  FLASH 版本: 程序下载到 STM32 的 FLASH 中，与普通的程序无异
//  RAM 版本  : 程序下载到 STM32 的内部 SRAM 中，需要使用 RAM 调试方式，
//                只能点 Debug 按钮运行（该运行方法可参考 SRAM 调试章节的说明）

//3. 若自己修改程序导致使芯片处于读写保护状态而无法下载，
//  且 FALSH 程序自身又不包含自解除状态的程序，
//  可以按 SRAM 调试的方式运行本工程的“RAM 版本”解除，解除即可重新下载。

/*
* 函数名: main
* 描述  : 主函数
* 输入  : 无
* 输出  : 无
*/
int main(void)
{
   HAL_Init();
   /* 配置系统时钟为 72 MHz */
   SystemClock_Config();
   /* 初始化 RGB 彩灯 */
   LED_GPIO_Config();
   /* 初始化 USART 配置模式为 115200 8-N-1，中断接收 */
   DEBUG_USART_Config();
   FLASH_Test();
}
```

在 main 函数中，初始化了串口、LED、按键等外设后，调用 FLASH_Test()，运行本函数后会给扇区 FLASH_WRP_SECTORS 进行写保护，再重复一次会进行解写保护。

42.4.3 下载测试

接上串口调试助手后，直接点击 MDK 的“Load”按钮把程序下载到 STM32 的 FLASH 中，复位运行。切换写保护状态时，芯片会自动复位，程序重新执行；切换读保护状态时，按键后需要重新给开发板上电复位，配置才会有效（断电时，串口与电脑的连接会断开，所以上电后注意重新打开串口调试助手），若是执行解除读保护过程，运行后芯片 FLASH 中自身的代码都会消失，所以要重新给开发板下载程序。

第 43 章　MPU6050 传感器—姿态检测

本章参考数据：《STM32F10x 参考手册》、《STM32F10x 数据手册》。

关于 MPU6050 的参考资料：《MPU-60X0 寄存器》、《MPU6050 数据手册》以及官方驱动《motion_driver_6.12》。

本章讲解的内容跨领域的知识较多，若您感兴趣，请自行查阅各方面的资料，对比学习。

43.1 姿态检测

43.1.1 基本认识

在飞行器中，飞行姿态是非常重要的参数，见图 *46_1*，以飞机自身的中心建立坐标系，当飞机绕坐标轴旋转的时候，会分别影响偏航角、横滚角及俯仰角。

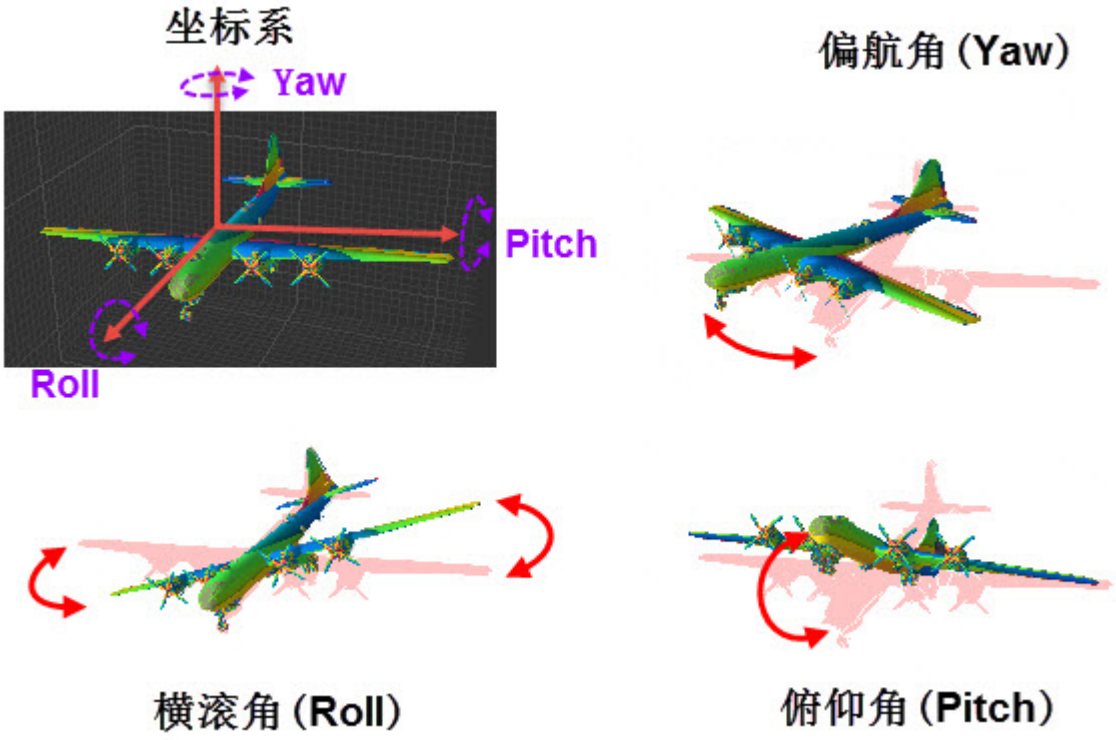


图 46-1 表示飞机姿态的偏航角、横滚角及俯仰角

假如我们知道飞机初始时是左上角的状态，只要想办法测量出基于原始状态的三个姿态角的变化量，再进行叠加，就可以获知它的实时姿态了。

43.1.2 坐标系

抽象来说，姿态是“载体坐标系”与“地理坐标系”之间的转换关系。

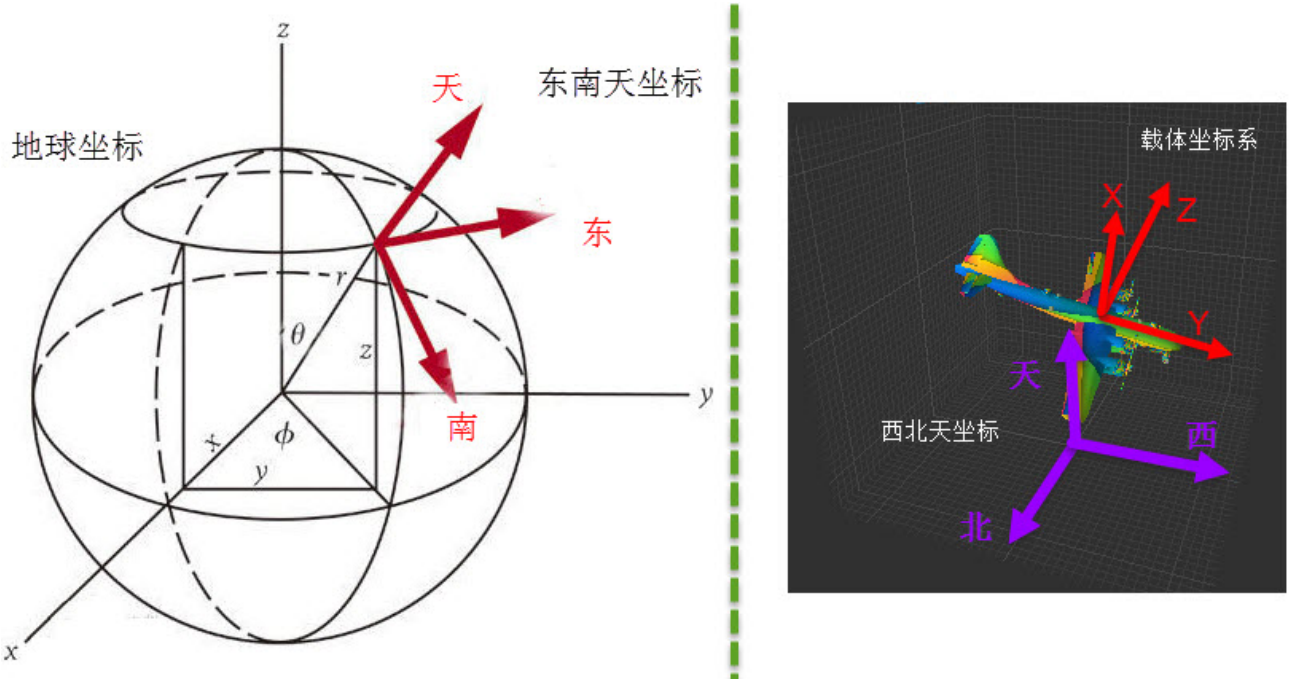


图 46-2 地球坐标系、地理坐标系与载体坐标系

我们先来了解三种常用的坐标系：

- 地球坐标系：以地球球心为原点，Z 轴沿地球自转轴方向，X、Y 轴在赤道平面内的坐标系。
- 地理坐标系：它的原点在地球表面 (或运载体所在的点)，Z 轴沿当地地理垂线的方向 (重力加速度方向)，XY 轴沿当地经纬线的切线方向。根据各个轴方向的不同，可选为“东北天”、“东南天”、“西北天”等坐标系。这是我们日常生活中使用的坐标系，平时说的东南西北方向与这个坐标系东南西北的概念一致。
- 载体坐标系：载体坐标系以运载体的质心为原点，一般根据运载体自身结构方向构成坐标系，如 Z 轴上由原点指向载体顶部，Y 轴指向载体头部，X 轴沿载体两侧方向。上面说基于飞机建立的坐标系就是一种载体坐标系，可类比到汽车、舰船、人体、动物或手机等各种物体。

地理坐标系与载体坐标系都以载体为原点，所以它们可以经过简单的旋转进行转换，载体的姿态角就是根据载体坐标系与地理坐标系的夹角来确定的。配合图 *46_1*，发挥您的空间想象力，假设初始状态中，飞机的 Z 轴、X 轴及 Y 轴分别与地理坐标系的天轴、北轴、东轴平行。如当飞机绕自身的“Z”轴旋转，它会使自身的“Y”轴方向与地理坐标系的“南北”方向偏离一定角度，

该角度就称为偏航角 (Yaw)；当载体绕自身的“X”轴旋转，它会使自身的“Z”轴方向与地理坐标系的“天地”方向偏离一定角度，该角度称为俯仰角 (Pitch)；当载体绕自身的“Y”轴旋转，它会使自身的“X”轴方向与地理坐标系的“东西”方向偏离一定角度，该角度称为横滚角。

表 46-1 姿态角的关系

坐标系间的旋转角度	说明	载体自身旋转
偏航角 (Yaw)	Y 轴与北轴的夹角	绕载体 Z 轴旋转可改变
俯仰角 (Pitch)	Z 轴与天轴的夹角	绕载体 X 轴旋转可改变
横滚角 (Roll)	X 轴与东轴的夹角	绕载体 Y 轴旋转可改变

这些角度也称欧拉角，是用于描述姿态的非常直观的角度。

43.1.3 利用陀螺仪检测角度

最直观的角度检测器就是陀螺仪了，见图 *46_3*，它可以检测物体绕坐标轴转动的“角速度”，如同将速度对时间积分可以求出路程一样，将角速度对时间积分就可以计算出旋转的“角度”。

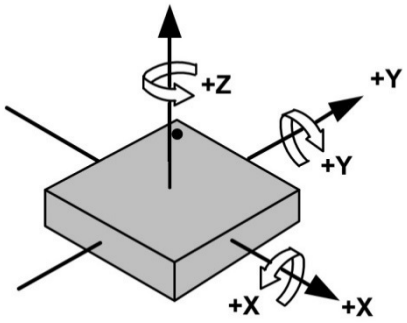


图 46-3 陀螺仪检测示意图

43.1.3.1 陀螺仪检测的缺陷

由于陀螺仪测量角度时使用积分，会存在积分误差，见图 *46_4*，若积分时间 Dt 越小，误差就越小。这十分容易理解，例如计算路程时，假设行车时间为 1 小时，我们随机选择行车过程某个时刻的速度 Vt 乘以 1 小时，求出的路程误差是极大的，因为行车的过程中并不是每个时刻都等于该时刻速度的，如果我们每 5 分钟检测一次车速，可得到 Vt1、Vt2、Vt3-Vt12 这 12 个时刻的车速，对各个时刻的速度乘以时间间隔 (5 分钟)，并对这 12 个结果求和，就可得出一个相对精确的行车路程了，不断提高采样频率，就可以使积分时间 Dt 变小，降低误差。

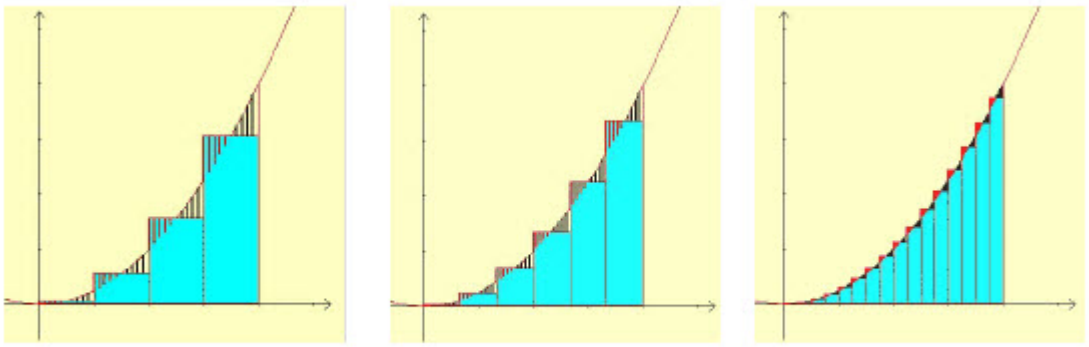


图 46-4 积分误差

同样地，提高陀螺仪传感器的采样频率，即可减少积分误差，目前非常普通的陀螺仪传感器的采样频率都可以达到 8KHz，已能满足大部分应用的精度要求。

更难以解决的是器件本身误差带来的问题。例如，某种陀螺仪的误差是 0.1 度/秒，当陀螺仪静止不动时，理想的角速度应为 0，无论它静止多久，对它进行积分测量得的旋转角度都是 0，这是理想的状态；而由于存在 0.1 度/秒的误差，当陀螺仪静止不动时，它采样得的角速度一直为 0.1 度/秒，若静止了 1 分钟，对它进行积分测量得的旋转角度为 6 度，若静止了 1 小时，陀螺仪进行积分测量得的旋转角度就是 360 度，即转过了一整圈，这就变得无法忍受了。只有当正方向误差和负方向误差能正好互相抵消的时候，才能消除这种累计误差。

43.1.4 利用加速度计检测角度

由于直接用陀螺仪测量角度在长时间测量时会产生累计误差，因而我们又引入了检测倾角的传感器。

图 46-5 T 字型水平仪

测量倾角最常见的例子是建筑中使用的水平仪，在重力的影响下，水平仪内的气泡能大致反映水柱所在直线与重力方向的夹角关系，利用图 *46_5* 中的 T 字型水平仪，可以检测出图 *46_1* 中说明的横滚角与俯仰角，但是偏航角是无法以这样的方式检测的。

在电子设备中，一般使用加速度传感器来检测倾角，它通过检测器件在各个方向的形变情况而采样得到受力数据，根据 F=ma 转换，传感器直接输出加速度数据，因而被称为加速度传感器。由于地球存在重力场，所以重力在任何时刻都会作用于传感器，当传感器静止的时候 (实际上加速度为 0)，传感器会在该方向检测出加速度 g，不能认为重力方向测出的加速度为 g，就表示传感器在该方向作加速度为 g 的运动。

当传感器的姿态不同时，它在自身各个坐标轴检测到的重力加速度是不一样的，利用各方向的测量结果，根据力的分解原理，可求出各个坐标轴与重力之间的夹角，见图 *46_6*。

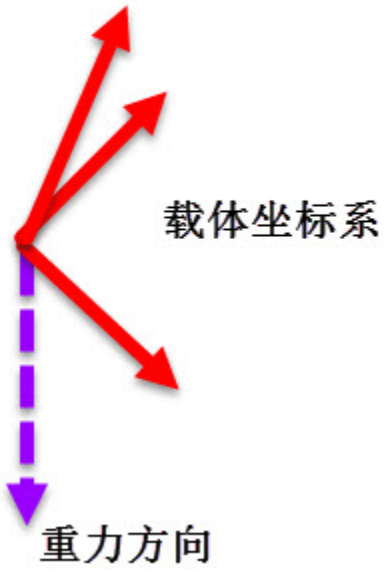


图 46-6 重力检测

因为重力方向是与地理坐标系的“天地”轴固连的，所以通过测量载体坐标系各轴与重力方向的夹角即可求得它与地理坐标系的角度旋转关系，从而获知载体姿态。

43.1.4.1 加速度传感器检测的缺陷

由于这种倾角检测方式是利用重力进行检测的，它无法检测到偏航角 (Yaw)，原理跟 T 字型水平仪一样，无论如何设计水平仪，水泡都无法指示这样的角度。

另一个缺陷是加速度传感器并不会区分重力加速度与外力加速度，当物体运动的时候，它也会在运动的方向检测出加速度，特别在震动的状态下，传感器的数据会有非常大的数据变化，此时难以反应重力的实际值。

43.1.5 利用磁场检测角度

为了弥补加速度传感器无法检测偏航角 (Yaw) 的问题，我们再引入磁场检测传感器，它可以检测出各个方向上的磁场大小，通过检测地球磁场，它可实现指南针的功能，所以也被称为电子罗盘。由于地磁场与地理坐标系的“南北”轴固联，利用磁场检测传感器的指南针功能，就可以测量出偏航角 (Yaw) 了。

43.1.5.1 磁场检测器的缺陷

与指南针的缺陷一样，使用磁场传感器会受到外部磁场干扰，如载体本身的电磁场干扰，不同地理环境的磁铁矿干扰等等。

43.1.6 利用 GPS 检测角度

使用 GPS 可以直接检测出载体在地球上的坐标，假如载体在某时刻测得坐标为 A，另一时刻测得坐标为 B，利用两个坐标即可求出它的航向，即可以确定偏航角，且不受磁场的影响，但这种检测方式只有当载体产生大范围位移的时候才有效 (GPS 民用精度大概为 10 米级)。

43.1.7 姿态融合与四元数

可以发现，使用陀螺仪检测角度时，在静止状态下存在缺陷，且受时间影响，而加速度传感器检测角度时，在运动状态下存在缺陷，且不受时间影响，刚好互补。假如我们同时使用这两种传感器，并设计一个滤波算法，当物体处于静止状态时，增大加速度数据的权重，当物体处于运动状时，增大陀螺仪数据的权重，从而获得更准确的姿态数据。同理，检测偏航角，当载体在静止状态时，可增大磁场检测器数据的权重，当载体在运动状态时，增大陀螺仪和 GPS 检测数据的权重。这些采用多种传感器数据来检测姿态的处理算法被称为姿态融合。

在姿态融合解算的时候常常使用“四元数”来表示姿态，它由三个实数及一个虚数组成，因而被称之为四元数。使用四元数表示姿态并不直观，但因为使用欧拉角 (即前面说的偏航角、横滚角及俯仰角) 表示姿态的时候会有“万向节死锁”问题，且运算比较复杂，所以一般在数据处理的

时候会使用四元数，处理完毕后再把四元数转换成欧拉角。在这里我们只要了解四元数是姿态的另一种表示方式即可，感兴趣的话可自行查阅相关资料。

43.2 传感器

43.2.1 传感器工作原理

前文提到了各种传感器，在这里大致讲解一下传感器的工作原理。我们讲的传感器一般是指把物理量转化成电信号量的装置，见图 *46_7*。

图 46-7 传感器工作原理

敏感元件直接感受被测物理量，并输出与该物理量有确定关系的信号，经过转换元件将该物理量信号转换为电信号，变换电路对转换元件输出的电信号进行放大调制，最后输出容易检测的电信号量。例如，温度传感器可把温度量转化成电压信号量输出，且温度值与电压值成比例关系，我们只要使用 ADC 测量出电压值，并根据转换关系即可求得实际温度值。而前文提到的陀螺仪、加速度及磁场传感器也是类似的，它们检测的角速度、加速度及磁场强度与电压值有确定的转换关系。

43.2.2 传感器参数

传感器一般使用精度、分辨率及采样频率这些参数来进行比较，衡量它的性能，见表 46-2。

表 46-2 传感器参数

参数	说明
线性误差	指传感器测量值与真实物理量值之间的拟合度误差。
分辨率	指传感器可检测到的最小物理量的单位。
采样频率	指在单位时间内的采样次数。

其中误差与分辨率是比较容易混淆的概念，以使用尺子测量长度为例，误差就是指尺子准不准，使用它测量出 10 厘米，与计量机构标准的 10 厘米有多大区别，若区别在 5 毫米以内，我们则称这把尺子的误差为 5 毫米。而分辨率是指尺子的最小刻度值，假如尺子的最小刻度值为 1 厘米，我们称这把尺子的分辨率为 1 厘米，它只能用于测量厘米级的尺寸，对于毫米级的长度，这就无法用这把尺子进行测量了。如果把尺子加热拉长，尺子的误差会大于 5 毫米，但它的分辨率仍为 1 厘米，只是它测出的 1 厘米值与真实值之间差得更远了。

43.2.3 物理量的表示方法

大部分传感器的输出都是与电压成比例关系的，电压值一般采用 ADC 来测量，而 ADC 一般有固定的位数，如 8 位 ADC、12 位 ADC 等，ADC 的位数会影响测量的分辨率及量程。例如图 46-8，假设用一个 2 位的 ADC 来测量长度，2 位的 ADC 最多只能表示 0、1、2、3 这四个数，假如它的分辨率为 20 厘米，那么它最大的测量长度为 60 厘米，假如它的分辨率为 10 厘米，那么它的最大测量长度为 30 厘米，由此可知，对于特定位数的 ADC，量程和分辨率不可兼得。

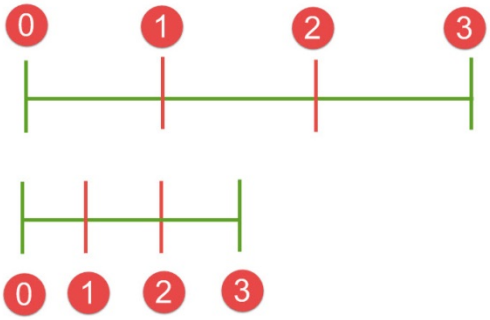


图 46-8 ADC 表示的物理量范围

在实际应用中，常常直接用 ADC 每位表征的物理量值来表示分辨率，如每位代表 20 厘米，我们称它的分辨率为 1LSB/20cm，它等效于 5 位表示 1 米：5LSB/m。其中的 LSB（Least Significant

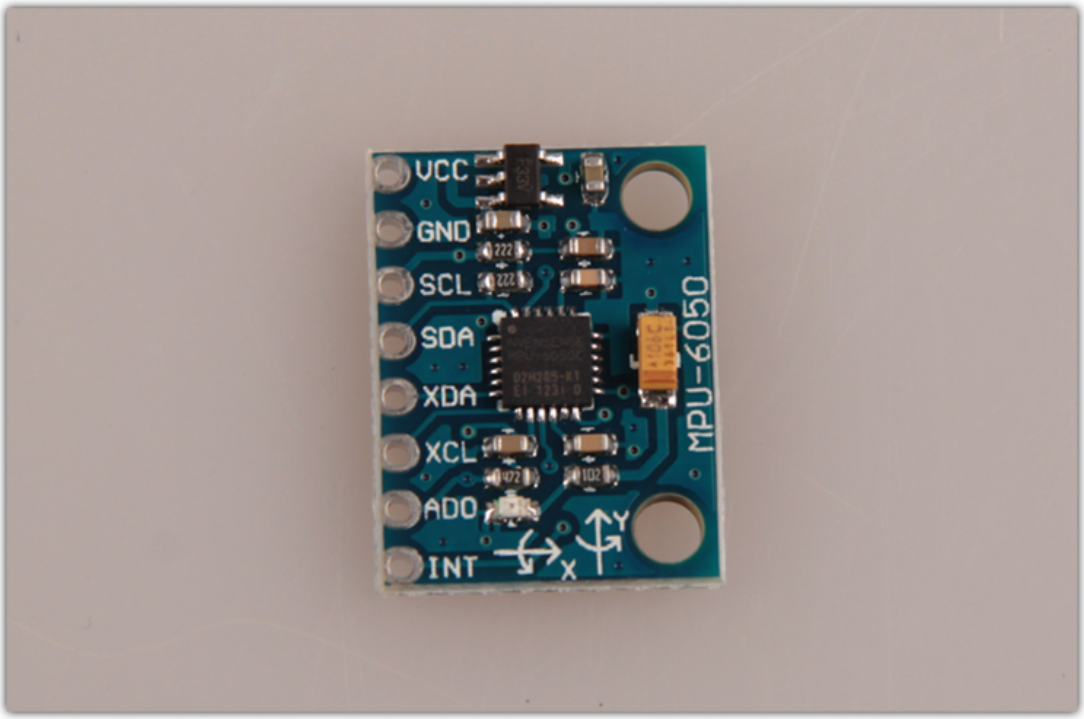
Bit），意为最 ADC 的低有效位。

使用采样得到的 ADC 数值，除以分辨率，即可求取得到物理量。例如使用分辨率为 5LSB/m、线性误差为 0.1m 的传感器进行长度测量，其 ADC 采样得到数据值为“20”，可计算知道该传感器的测量值为 4 米，而该长度的真实值介于 3.9-4.1 米之间。

43.3 MPU6050 模块简介

43.3.1 MPU6050 模块功能及外观

接下来我们使用传感器实例来讲解如何检测物体的姿态。在我们的开发板上有引出 I2C 总线接口，方便用于扩展使用 I2C 协议通讯的传感器模块，本节中我们将在板子上扩展 MPU6050 陀螺仪模块，见图 *47_9*。它是一种六轴传感器模块，采用 InvenSense 公司的 MPU6050 作为主芯片，能同时检测三轴加速度、三轴陀螺仪 (三轴角速度) 的运动数据以及温度数据。利用 MPU6050 芯片内部的 DMP 模块（Digital Motion Processor 数字运动处理器），可对传感器数据进行滤波、融合处理，它直接通过 I2C 接口向主控器输出姿态解算后的姿态数据，降低主控器的运算量。其姿态解算频率最高可达 200Hz，非常适合用于对姿态控制实时要求较高的领域。常见应用于手机、智能手环、四轴飞行器及计步器等的姿态检测。


VCC
GND
SCL
SDA
XDA
XCL
ADO
INT
MPU-6050

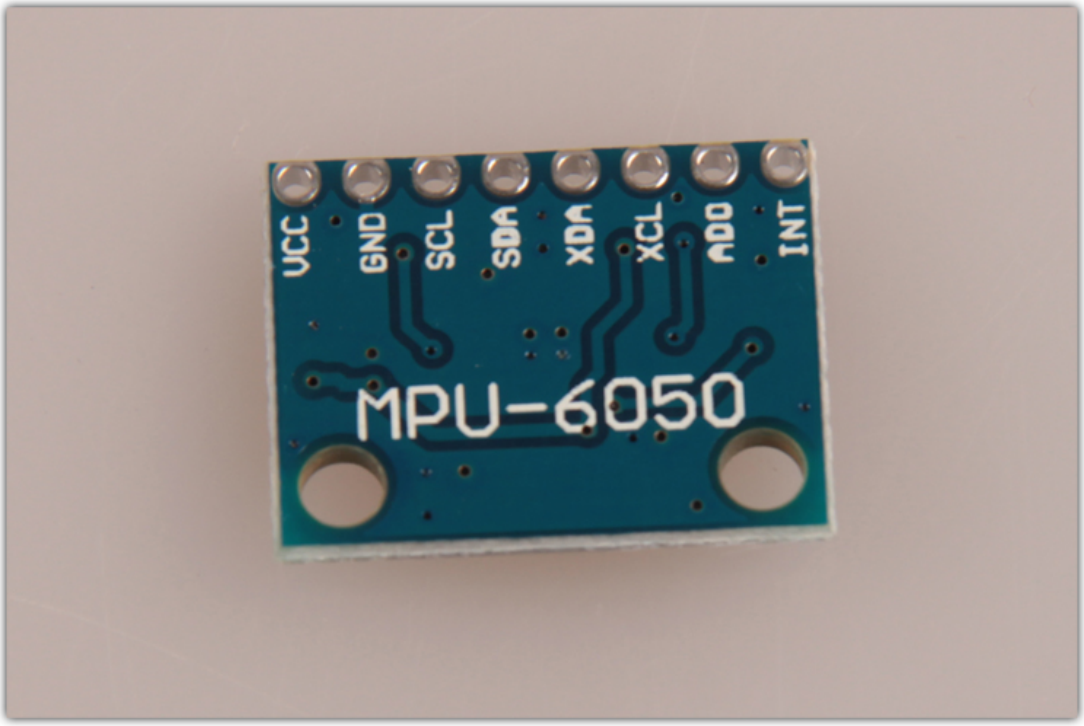

VCC
GND
SCL
SDA
XDA
XCL
ADO
INT
MPU-6050

图 47-9 MPU6050 模块外观

43.3.2 MPU6050 模块的引脚功能说明

该模块引出的 8 个引脚功能说明见表 47-3。

表 47-3 MPU6050 模块引脚说明

序号	引脚名称	说明
1	VCC	3.3/5V 电源输入
2	GND	地线
3	SCL	I2C 从时钟信号线 SCL (模块上已接上拉电阻)
4	SDA	I2C 从数据信号线 SDA (模块上已接上拉电阻)
5	XDA	I2C 主串行数据信号线，用于外接传感器 (模块上已接上拉电阻)
6	XCL	I2C 主串行时钟信号线，用于外接传感器 (模块上已接上拉电阻)
7	AD0	从机地址设置引脚 • 接地或悬空时，地址为:0x68 • 接 VCC 时，地址为:0x69
8	INT	中断输出引脚

其中的 SDA/SCL、XDA/XCL 通讯引脚分别为两组 I2C 信号线。当模块与外部主机通讯时，使用 SDA/SCL，如与 STM32 芯片通讯；而 XDA/XCL 则用于 MPU6050 芯片与其它 I2C 传感器通讯时使用，例如使用它与磁场传感器连接，MPU6050 模块可以把从主机 SDA/SCL 接收的数据或命令通过 XDA/XCL 引脚转发到磁场传感器中。但实际上这种功能比较鸡肋，控制麻烦且效率低，

一般会直接把磁场传感器之类的 I2C 传感器直接与 MPU6050 挂载在同一条总线上（即都连接到 SDA/SCL），使用主机直接控制。

43.3.3 MPU6050 模块的硬件原理图

MPU6050 模块的硬件原理图见图 *47_10*。

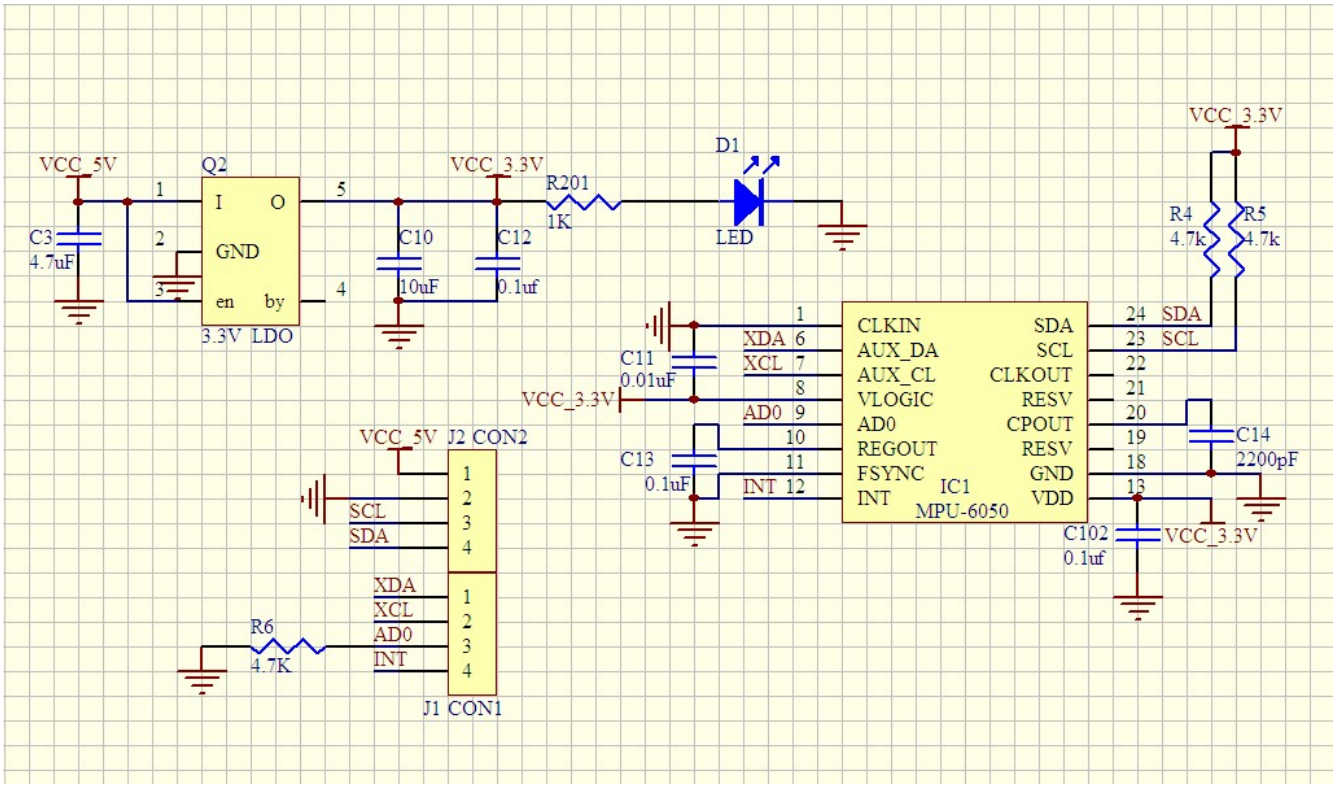


图 47-10 MPU6050 模块原理图

它的硬件非常简单，SDA 与 SCL 被引出方便与外部 I2C 主机连接，看图中的右上角，可知该模块的 I2C 通讯引脚 SDA 及 SCL 已经连接了上拉电阻，因此它与外部 I2C 通讯主机通讯时直接使用导线连接起来即可；而 MPU6050 模块与其它传感器通讯使用的 XDA、XCL 引脚没有接上拉电阻，要使用时需要注意。模块自身的 I2C 设备地址可通过 AD0 引脚的电平控制，当 AD0 接地时，设备地址为 0x68(七位地址)，当 AD0 接电源时，设备地址为 0x69(七位地址)。另外，当传感器有新数据的时候会通过 INT 引脚通知 STM32。

由于 MPU6050 检测时是基于自身中心坐标系的，见图 *46_9*，它表示的坐标系及旋转符号标出了

MPU6050 传感器的 XYZ 轴的加速度有角速度的正方向。所以在安装模块时，您需要考虑它与所在设备的坐标系统的关系。

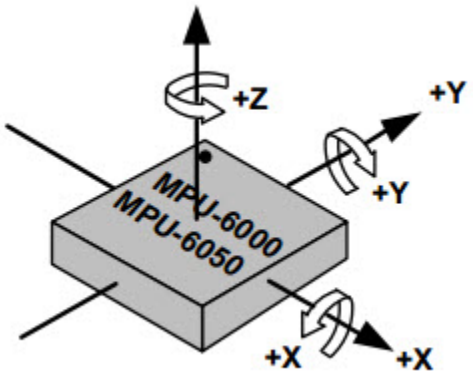


图 46-9 MPU6050 传感器的坐标及方向

图 *46_9* 中表示的坐标系及旋转符号标出了 MPU6050 传感器的 XYZ 轴的加速度有角速度的正方向。

43.4 MPU6050 的特性参数

实验板中使用的 MPU6050 传感器参数见表 46-3。

表 46-3 MPU6050 的特性参数

参数	说明
供电	3.3V-5V
通讯接口	I2C 协议，支持的 I2C 时钟最高频率为 400KHz
测量维度	加速度：3 维陀螺仪：3 维
ADC 分辨率	加速度：16 位陀螺仪：16 位
加速度测量范围	±2g、±4g、±8g、±16g 其中 g 为重力加速度常数，g=9.8m/s ²
加速度最高分辨率	16384 LSB/g
加速度线性误差	0.1g
加速度输出频率	最高 1000Hz
陀螺仪测量范围	±250 °/s 、±500 °/s 、±1000 °/s、±2000 °/s、
陀螺仪最高分辨率	131 LSB/(°/s)
陀螺仪线性误差	0.1 °/s
陀螺仪输出频率	最高 8000Hz
DMP 姿态解算频率	最高 200Hz
温度传感器测量范围	-40~ +85℃
温度传感器分辨率	340 LSB/℃
温度传感器线性误差	±1℃
工作温度	-40~ +85℃
功耗	500uA~3.9mA (工作电压 3.3V)

该表说明，加速度与陀螺仪传感器的 ADC 均为 16 位，它们的量程及分辨率可选多种模式，见图 *46_11*，量程越大，分辨率越低。

AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2*g*	16384 LSB/*g*
1	±4*g*	8192 LSB/*g*
2	±8*g*	4096 LSB/*g*
3	±16*g*	2048 LSB/*g*

图 46-10 加速度配置跟量程的关系

FS_SEL selects the full scale range of the gyroscope outputs according to the following table.

FS_SEL	Full Scale Range
0	± 250 °/s
1	± 500 °/s
2	± 1000 °/s
3	± 2000 °/s

图 46-11 陀螺仪的几种量程配置

从表中还可了解到传感器的加速度及陀螺仪的采样频率分别为 1000Hz 及 8000Hz，它们是指加速度及角速度数据的采样频率，我们可以使用 STM32 控制器把这些数据读取出来然后进行姿态融合解算，以求出传感器当前的姿态 (即求出偏航角、横滚角、俯仰角)。而如果我们使用传感器内部的 DMP 单元进行解算，它可以直接对采样得到的加速度及角速度进行姿态解算，解算得到的结果再输出给 STM32 控制器，即 STM32 无需自己计算，可直接获取偏航角、横滚角及俯仰角，该 DMP 每秒可输出 200 次姿态数据。

43.5 MPU6050—获取原始数据实验

这一小节我们学习如何使用 STM32 控制 MPU6050 传感器读取加速度、角速度及温度数据。在控制传感器时，使用到了 STM32 的 I2C 驱动，就如同控制 STM32 一样，对 MPU6050 传感器的不同寄存器写入不同内容可以实现不同模式的控制，从特定的寄存器读取内容则可获取测量数据，这部分关于 MPU6050 具体寄存器的内容我们不再展开，请您查阅《MPU-60X0 寄存器》手册获

知。

43.5.1 硬件设计

在实验前，我们先用杜邦线把 STM32 开发板与该 MPU6050 模块连接起来，见图 *47_14* 及表 47-5。

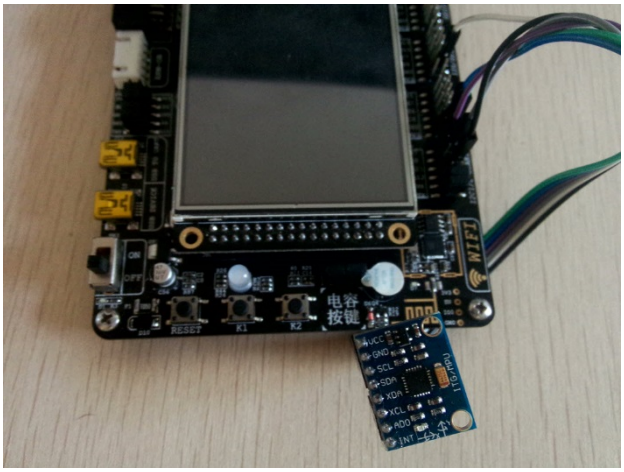


图 47-14 STM32 与 MPU6050 的硬件连接

表 47-5 MPU6050 模块引脚说明

序号	引脚名称	与 STM32 开发板连接
1	VCC	接 3.3V 或 5V
2	GND	GND
3	SCL	PB6
4	SDA	PB7
5	AD0	悬空或接地
6	INT	PA11

43.5.2 软件设计

本小节讲解的是“硬件 STM32-MPU6050”实验，请打开配套的代码工程阅读理解。为了方便展示及移植，我们把 STM32 的 I2C 驱动相关的代码都编写到“bsp_i2c.c”及“bsp_i2c.h”文件中，与 MPU6050 传感器相关的代码都写到“mpu6050.c”及“mpu6050.h”文件中，这些文件是我们自己编写的，不属于 HAL 库的内容，可根据您的喜好命名文件。

43.5.2.1 程序设计要点

(1) 初始化 STM32 的 I2C；

(2) 使用 I2C 向 MPU6050 写入控制参数；

(3) 定时读取加速度、角速度及温度数据。

43.5.2.2 代码分析

43.5.2.2.1 I2C 的硬件定义

本实验中的 I2C 驱动与 MPU6050 驱动分开主要是考虑到扩展其它传感器时的通用性，如使用磁场传感器、气压传感器都可以使用同样一个 I2C 驱动，这个驱动只要给出针对不同传感器时的不同读写接口即可。关于 STM32 的 I2C 驱动原理请参考读写 EEPROM 的章节，本章讲解的 I2C 驱动主要针对接口封装讲解，细节不再赘述。本实验中的 I2C 硬件定义见*代码清单 46_1*。

列表 1: 代码清单 46-1 I2C 的硬件定义 (bsp_i2c.h 文件)

```
/* 引脚定义 */
#define SENSORS_I2C_SCL_GPIO_PORT GPIOB
#define SENSORS_I2C_SCL_GPIO_CLK_ENABLE() __HAL_RCC_GPIOB_CLK_ENABLE()
#define SENSORS_I2C_SCL_GPIO_PIN GPIO_PIN_6
#define SENSORS_I2C_SDA_GPIO_PORT GPIOB
#define SENSORS_I2C_SDA_GPIO_CLK_ENABLE() __HAL_RCC_GPIOB_CLK_ENABLE()
#define SENSORS_I2C_SDA_GPIO_PIN GPIO_PIN_7
```

(下页继续)

(续上页)

```
#define SENSORS_I2C I2C1
#define SENSORS_I2C_RCC_CLK_ENABLE() __HAL_RCC_I2C1_CLK_ENABLE()
#define SENSORS_I2C_FORCE_RESET() __HAL_RCC_I2C1_FORCE_RESET()
#define SENSORS_I2C_RELEASE_RESET() __HAL_RCC_I2C1_RELEASE_RESET()
```

这些宏根据传感器使用的 I2C 硬件封装起来了。

43.5.2.2.2 初始化 I2C

接下来利用这些宏对 I2C 进行初始化，初始化过程与 I2C 读写 EEPROM 中的无异，见代码清单 46_2。

列表 2: 代码清单 46-2 初始化 I2C（bsp_i2c.c 文件）

```
void I2cMaster_Init(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;

   /* 使能 I2Cx 时钟 */
   SENSORS_I2C_RCC_CLK_ENABLE();

   /* 使能 I2C GPIO 时钟 */
   SENSORS_I2C_SCL_GPIO_CLK_ENABLE();
   SENSORS_I2C_SDA_GPIO_CLK_ENABLE();

   /* 配置 I2Cx 引脚: SCL ----------------------------------------*/
   GPIO_InitStructure.Pin =  SENSORS_I2C_SCL_GPIO_PIN;
   GPIO_InitStructure.Mode = GPIO_MODE_AF_OD;
   GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
   GPIO_InitStructure.Pull= GPIO_NOPULL;
//  GPIO_InitStructure.Alternate=SENSORS_I2C_AF;
   HAL_GPIO_Init(SENSORS_I2C_SCL_GPIO_PORT, &GPIO_InitStructure);
```

(下页继续)

(续上页)

```
  /* 配置 I2Cx 引脚: SDA ----------------------------------------*/
  GPIO_InitStructure.Pin = SENSORS_I2C_SDA_GPIO_PIN;
  HAL_GPIO_Init(SENSORS_I2C_SDA_GPIO_PORT, &GPIO_InitStructure);

  if (HAL_I2C_GetState(&I2C_Handle) == HAL_I2C_STATE_RESET) {
        /* 强制复位 I2C 外设时钟 */
        SENSORS_I2C_FORCE_RESET();

        /* 释放 I2C 外设时钟复位 */
        SENSORS_I2C_RELEASE_RESET();

        /* I2C 配置 */
        I2C_Handle.Instance = SENSORS_I2C;
        I2C_Handle.Init.AddressingMode  = I2C_ADDRESSINGMODE_7BIT;
        I2C_Handle.Init.ClockSpeed      = 400000;
        I2C_Handle.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
        I2C_Handle.Init.DutyCycle       = I2C_DUTYCYCLE_2;
        I2C_Handle.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
        I2C_Handle.Init.NoStretchMode   = I2C_NOSTRETCH_DISABLE;
        I2C_Handle.Init.OwnAddress1     = 0;
        I2C_Handle.Init.OwnAddress2     = 0;
        /* 初始化 I2C */
        HAL_I2C_Init(&I2C_Handle);
//    /* 使能模拟滤波器 */
//    HAL_I2CEx_AnalogFilter_Config(&I2C_Handle, I2C_ANALOGFILTER_ENABLE);
  }
}
```

43.5.2.2.3 对读写函数的封装

初始化完成后就是编写 I2C 读写函数了，这部分跟 EERPOM 的一样，主要是调用 STM32 HAL 库函数读写数据寄存器及标志位，本实验的这部分被编写进 ST_Sensors_I2C_WriteRegister 及 ST_Sensors_I2C_ReadRegister 中了，在它们之上，再封装成了 Sensors_I2C_WriteRegister 及 Sensors_I2C_ReadRegister，见*代码清单 46_3*。

列表 3: 代码清单 46-3 对读写函数的封装 (i2c.c 文件)

```
/**
* @brief  写寄存器，这是提供给上层的接口
* @param  slave_addr: 从机地址
* @param  reg_addr: 寄存器地址
* @param len: 写入的长度
* @param data_ptr: 指向要写入的数据
* @retval 正常为 0，不正常为非 0
*/
int Sensors_I2C_WriteRegister(unsigned char slave_addr,
                              unsigned char reg_addr,
                              unsigned short len,
                              unsigned char *data_ptr)
{
   HAL_StatusTypeDef status = HAL_OK;
   status = HAL_I2C_Mem_Write(&I2C_Handle, slave_addr, reg_addr,
   I2C_MEMADD_SIZE_8BIT,data_ptr, len,I2Cx_FLAG_TIMEOUT);
   if (status != HAL_OK) {/* 检查通讯状态 */
      /* 总线出错处理 */
      I2Cx_Error(slave_addr);
   }
   while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {
}
   /* 检查 SENSOR 是否就绪进行下一次读写操作 */
   while (HAL_I2C_IsDeviceReady(&I2C_Handle, slave_addr,
```

(下页继续)

(续上页)

```
    I2Cx_FLAG_TIMEOUT, I2Cx_FLAG_TIMEOUT) == HAL_TIMEOUT);
    /* 等待传输结束 */
    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {

    }
    return status;
}

/**
* @brief  读寄存器，这是提供给上层的接口
* @param  slave_addr: 从机地址
* @param  reg_addr: 寄存器地址
* @param len: 要读取的长度
* @param data_ptr: 指向要存储数据的指针
* @retval 正常为 0，不正常为非 0
*/
int Sensors_I2C_ReadRegister(unsigned char slave_addr,
                             unsigned char reg_addr,
                             unsigned short len,
                             unsigned char *data_ptr)
{
    HAL_StatusTypeDef status = HAL_OK;
    status =HAL_I2C_Mem_Read(&I2C_Handle,slave_addr
    ,reg_addr,I2C_MEMADD_SIZE_8BIT,data_ptr,len,I2Cx_FLAG_TIMEOUT);
    if (status != HAL_OK) {/* 检查通讯状态 */
        /* 总线出错处理 */
        I2Cx_Error(slave_addr);
    }
    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {
    }
    /* 检查 SENSOR 是否就绪进行下一次读写操作 */
    while (HAL_I2C_IsDeviceReady(&I2C_Handle, slave_addr,
```

(下页继续)

(续上页)

```
    I2Cx_FLAG_TIMEOUT, I2Cx_FLAG_TIMEOUT) == HAL_TIMEOUT);
    /* 等待传输结束 */
    while (HAL_I2C_GetState(&I2C_Handle) != HAL_I2C_STATE_READY) {


    }
    return status;
}
```

封装后的函数主要是增加了错误重试机制，若读写出现错误，则会进行多次尝试，多次尝试均失败后会返回错误代码。这个函数作为 I2C 驱动对外的接口，其它使用 I2C 的传感器调用这个函数进行读写寄存器。

43.5.2.2.4 MPU6050 的寄存器定义

MPU6050 有各种各样的寄存器用于控制工作模式，我们把这些寄存器的地址、寄存器位使用宏定义到了 mpu6050.h 文件中了，见*代码清单 46_4*。

列表 4: 代码清单 46-4 MPU6050 的寄存器定义(mpu6050.h)

```
//模块的 A0 引脚接 GND, IIC 的 7 位地址为 0x68, 若接到 VCC, 需要改为 0x69
#define MPU6050_SLAVE_ADDRESS   (0x68<<1)       //MPU6050 器件读地址
#define MPU6050_WHO_AM_I         0x75
#define MPU6050_SMPLRT_DIV       0  //8000Hz
#define MPU6050_DLPF_CFG         0
#define MPU6050_GYRO_OUT         0x43      //MPU6050 陀螺仪数据寄存器地址
#define MPU6050_ACC_OUT          0x3B      //MPU6050 加速度数据寄存器地址
#define MPU6050_RA_XG_OFFS_TC        0x00 //[7] PWR_MODE, [6:1]
/*......... 以下部分省略 */
```

43.5.2.2.5 初始化 MPU6050

根据 MPU6050 的寄存器功能定义，我们使用 I2C 往寄存器写入特定的控制参数，见*代码清单 46_5*。

列表 5: 代码清单 46-5 初始化 MPU6050(mpu6050.c)

```
/**
* @brief    写数据到 MPU6050 寄存器
* @param    reg_add: 寄存器地址
* @param    reg_data: 要写入的数据
* @retval
*/
void MPU6050_WriteReg(u8 reg_add,u8 reg_dat)
{
   Sensors_I2C_WriteRegister(MPU6050_ADDRESS,reg_add,1,&reg_dat);
}
```

(下页继续)

(续上页)

```
/**
* @brief   从 MPU6050 寄存器读取数据
* @param   reg_add: 寄存器地址
* @param   Read: 存储数据的缓冲区
* @param   num: 要读取的数据量
* @retval
*/
void MPU6050_ReadData(u8 reg_add,unsigned char* Read,u8 num)
{
   Sensors_I2C_ReadRegister(MPU6050_ADDRESS,reg_add,num,Read);
}

/**
* @brief   初始化 MPU6050 芯片
* @param
* @retval
*/
void MPU6050_Init(void)
{
   int i=0,j=0;
   //在初始化之前要延时一段时间，若没有延时，则断电后再上电数据可能会出错
   for (i=0; i<1000; i++) {
      for (j=0; j<1000; j++) {
            ;
      }
   }
   //解除休眠状态
   MPU6050_WriteReg(MPU6050_RA_PWR_MGMT_1, 0x00);
   //陀螺仪采样率
   MPU6050_WriteReg(MPU6050_RA_SMPLRT_DIV , 0x07);
   MPU6050_WriteReg(MPU6050_RA_CONFIG , 0x06);
   //配置加速度传感器工作在 16G 模式
```

(下页继续)

(续上页)

```
    MPU6050_WriteReg(MPU6050_RA_ACCEL_CONFIG , 0x01);
    //陀螺仪自检及测量范围，典型值：0x18(不自检，2000deg/s)
    MPU6050_WriteReg(MPU6050_RA_GYRO_CONFIG, 0x18);
}
```

这段代码首先使用 MPU6050_ReadData 及 MPU6050_WriteRed 函数封装了 I2C 的底层读写驱动，接下来用它们在 MPU6050_Init 函数中向 MPU6050 寄存器写入控制参数，设置了 MPU6050 的采样率、量程 (分辨率)。

43.5.2.2.6 读传感器 ID

初始化后，可通过读取它的“WHO AM I”寄存器内容来检测硬件是否正常，该寄存器存储了 ID 号 0x68，见*代码清单 46_6*。

列表 6: 代码清单 46-6 读取传感器 ID(mpu6050.c)

```
/**
    * @brief   读取 MPU6050 的 ID
    * @param
    * @retval  正常返回 1，异常返回 0
    */
uint8_t MPU6050ReadID(void)
{
    unsigned char Re = 0;
    MPU6050_ReadData(MPU6050_RA_WHO_AM_I,&Re,1);     //读器件地址
    if (Re != 0x68) {
        MPU_ERROR("检测不到 MPU6050 模块，请检查模块与开发板的接线");
        return 0;
    } else {
        MPU_INFO("MPU6050 ID = %d\r\n",Re);
        return 1;
```

(下页继续)

(续上页)

```
    }
}
```

43.5.2.2.7 读取原始数据

若传感器检测正常，就可以读取它数据寄存器获取采样数据了，见*代码清单 46_7*。

列表 7: 代码清单 46-7 读取传感器数据 (mpu6050.c)

```
/**
* @brief    读取 MPU6050 的加速度数据
* @param
* @retval
*/
void MPU6050ReadAcc(short *accData)
{
    u8 buf[6];
    MPU6050_ReadData(MPU6050_ACC_OUT, buf, 6);
    accData[0] = (buf[0] << 8) | buf[1];
    accData[1] = (buf[2] << 8) | buf[3];
    accData[2] = (buf[4] << 8) | buf[5];
}

/**
* @brief    读取 MPU6050 的角加速度数据
* @param
* @retval
*/
void MPU6050ReadGyro(short *gyroData)
{
    u8 buf[6];
```

(下页继续)

(续上页)

```
    MPU6050_ReadData(MPU6050_GYRO_OUT,buf,6);
    gyroData[0] = (buf[0] << 8) | buf[1];
    gyroData[1] = (buf[2] << 8) | buf[3];
    gyroData[2] = (buf[4] << 8) | buf[5];
}

/**
* @brief    读取 MPU6050 的原始温度数据
* @param
* @retval
*/
void MPU6050ReadTemp(short *tempData)
{
    u8 buf[2];
    MPU6050_ReadData(MPU6050_RA_TEMP_OUT_H,buf,2);      //读取温度值
    *tempData = (buf[0] << 8) | buf[1];
}

/**
* @brief    读取 MPU6050 的温度数据，转化成摄氏度
* @param
* @retval
*/
void MPU6050_ReturnTemp(float*Temperature)
{
    short temp3;
    u8 buf[2];

    MPU6050_ReadData(MPU6050_RA_TEMP_OUT_H,buf,2);      //读取温度值
    temp3= (buf[0] << 8) | buf[1];
    *Temperature=((double) (temp3 /340.0))+36.53;
}
```

其中前以上三个函数分别用于读取三轴加速度、角速度及温度值，这些都是原始的 ADC 数值 (16 位长)，对于加速度和角速度，把读取得的 ADC 值除以分辨率，即可求得实际物理量数值。最后一个函数 MPU6050_ReturnTemp 展示了温度 ADC 值与实际温度值间的转换，它是根据 MPU6050 的说明给出的转换公式进行换算的，注意陀螺仪检测的温度会受自身芯片发热的影响，严格来说它测量的是自身芯片的温度，所以用它来测量气温是不太准确的。对于加速度和角速度值我们没有进行转换，在下一小节中我们直接利用这些数据交给 DMP 单元，求解出姿态角。

43.5.2.2.8 main 函数

最后我们来看看本实验的 main 函数，见*代码清单 46_8*。

列表 8: 代码清单 46-8 main 函数

```
int main(void)
{

   short Acel[3];
   short Gyro[3];
   float Temp;

   SystemClock_Config();

   /* LED 端口初始化 */
   LED_GPIO_Config();
   /* 串口通信初始化 */
   DEBUG_USART_Config();

   //I2C 初始化
   I2cMaster_Init();
   //MPU6050 初始化
   MPU6050_Init();
```

(下页继续)

(续上页)

```
    //检测 MPU6050
    if (MPU6050ReadID() == 1) {
          while (1) {
             if (Task_Delay[0]==TASK_ENABLE) {
                LED2_TOGGLE;
                Task_Delay[0]=1000;
             }

             if (Task_Delay[1]==0) {
                MPU6050ReadAcc(Acel);
                printf("加速度: %8d%8d%8d",Acel[0],Acel[1],Acel[2]);
                MPU6050ReadGyro(Gyro);
                printf("    陀螺仪%8d%8d%8d",Gyro[0],Gyro[1],Gyro[2]);
                MPU6050_ReturnTemp(&Temp);
                printf("    温度%8.2f\r\n",Temp);

                Task_Delay[1]=500;//更新一次数据，可根据自己的需求，提高采样频率，如
100ms 采样一次
             }
          }
    } else {
          printf("\r\n没有检测到 MPU6050 传感器! \r\n");
          LED_RED;
          while (1);
    }
}
```

本实验中控制 MPU6050 并没有使用中断检测，我们是利用 Systick 定时器进行计时，隔一段时间读取 MPU6050 的数据寄存器获取采样数据的，代码中使用 Task_Delay 变量来控制定时时间，在 Systick 中断里会每隔 1ms 对该变量值减 1，所以当它的值为 0 时表示定时时间到。

在 main 函数里，调用 I2cMaster_Init、MPU6050_Init 及 MPU6050ReadID 函数后，就在 whlie 循环里判断定时时间，定时时间到后就读取加速度、角速度及温度值，并使用串口打印信息到电脑端。

通过宏定义 USE_LCD_DISPLAY 来确定是否在 LCD 液晶上显示传感器的数据。

43.5.2.3 下载验证

使用杜邦线连接好开发板和模块，用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到 MPU6050 采样得到的调试信息。

43.6 MPU6050—利用 DMP 进行姿态解算

上一小节我们仅利用 MPU6050 采集了原始的数据，如果您对姿态解算的算法深有研究，可以自行编写姿态解算的算法，并利用这些数据，使用 STM32 进行姿态解算，解算后输出姿态角。而由于 MPU6050 内部集成了 DMP，不需要 STM32 参与解算，可直接输出姿态角，也不需要对解算算法作深入研究，非常方便，本章讲解如何使用 DMP 进行解算。

实验中使用的代码主体是从 MPU6050 官方提供的驱动《motion_driver_6.12》移植过来的，该资料包里提供了基于 STM32F1 控制器的源代码（本工程正是利用该代码移植到 STM32F1 上的）及使用 python 语言编写的上位机，资料中还附带了说明文档，请您充分利用官方自带的资料学习。

43.6.1 硬件设计

硬件设计与上一小节实验中的完全一样，且软件中使用了 INT 引脚产生的中断信号，本小节中的代码默认使用软件 I2C。

43.6.2 软件设计

本小节讲解的是“MPU6050_python 上位机”实验，请打开配套的代码工程阅读理解。本工程是从官方代码移植过来的 (IAR 工程移植至 MDK)，改动并不多，我们主要给读者讲解一下该驱动的设计思路，方便应用。由于本工程的代码十分庞大，在讲解到某些函数时，请善用 MDK 的搜索功能，从而在工程中查找出对应的代码。

43.6.2.1 程序设计要点

(1) 提供 I2C 读写接口、定时服务及 INT 中断处理；

(2) 从陀螺仪中获取原始数据并处理；

(3) 更新数据并输出。

43.6.2.2 代码分析

官方的驱动主要是了 MPL 软件库 (Motion Processing Library)，要移植该软件库我们需要为它提供 I2C 读写接口、定时服务以及 MPU6050 的数据更新标志。若需要输出调试信息到上位机，还需要提供串口接口。

43.6.2.2.1 I2C 读写接口

MPL 库的内部对 I2C 读写时都使用 i2c_write 及 i2c_read 函数，在文件“inv_mpu.c”中给出了它们的接口格式，见*代码清单 46_1*。

列表 9: 代码清单 46-9 I2C 读写接口 (inv_mpu.c 文件)

```
/* 以下函数需要定义成如下格式:
* i2c_write(unsigned char slave_addr, unsigned char reg_addr,
*      unsigned char length, unsigned char const *data)
* i2c_read(unsigned char slave_addr, unsigned char reg_addr,
*      unsigned char length, unsigned char *data)
*/

#define i2c_write   Sensors_I2C_WriteRegister
#define i2c_read    Sensors_I2C_ReadRegister
```

这些接口的格式与我们上一小节写的 I2C 读写函数 Sensors_I2C_ReadRegister 及 Sensors_I2C_WriteRegister 一致，所以可直接使用宏替换。

43.6.2.2.2 提供定时服务

MPL 软件库中使用到了延时及时间戳功能，要求需要提供 delay_ms 函数实现毫秒级延时，提供 get_ms 获取毫秒级的时间戳，它们的接口格式也在 “inv_mpu.c” 文件中给出，见代码清单 *46_10*。

列表 10: 代码清单 46-10 定时服务接口（inv_mpu.c 文件）

```
/*
* delay_ms(unsigned long num_ms)
* get_ms(unsigned long *count)
*/

#define delay_ms    Delay_ms
#define get_ms      get_tick_count
```

我们为接口提供的 Delay_ms 及 get_tick_count 函数定义在 bsp_SysTick.c 文件，我们使用 SysTick 每毫秒产生一次中断，进行计时，见代码清单 *46_11*。

列表 11: 代码清单 46-11 使用 Systick 进行定时（bsp_SysTick.c）

```
static __IO u32 TimingDelay;
static __IO uint32_t g_ul_ms_ticks=0;
/**
   * @brief   us 延时程序,1ms 为一个单位
   * @param
   *   @arg nTime: Delay_ms( 1 ) 则实现的延时为 1 ms
   * @retval  无
   */
void Delay_ms(__IO u32 nTime)
{
   TimingDelay = nTime;
```

(下页继续)

(续上页)

```
    while (TimingDelay != 0);
}

/**
    * @brief  获取节拍程序
    * @param  无
    * @retval 无
    * @attention  在 SysTick 中断函数 SysTick_Handler() 调用
    */
void TimingDelay_Decrement(void)
{
    if (TimingDelay != 0x00) {
          TimingDelay--;
    }
}

/**
    * @brief  获取当前毫秒值
    * @param  存储最新毫秒值的变量
    * @retval 无
    */
int get_tick_count(unsigned long *count)
{
    count[0] = g_ul_ms_ticks;
    return 0;
}

/**
    * @brief  毫秒累加器，在中断里每毫秒加 1
    * @param  无
    * @retval 无
    */
```

(下页继续)

(续上页)

```
void TimeStamp_Increment (void)
{
   g_ul_ms_ticks++;
}
```

上述代码中的 TimingDelay_Decrement 和 TimeStamp_Increment 函数是在 Systick 的中断服务函数中被调用的，见*代码清单 46_12*。systick 被配置为每毫秒产生一次中断，而每次中断中会对 TimingDelay 变量减 1，对 g_ul_ms_ticks 变量加 1。它们分别用于 Delay_ms 函数利用 TimingDelay 的值进行阻塞延迟，而 get_tick_count 函数获取的时间戳即 g_ul_ms_ticks 的值。

列表 12: 代码清单 46-12 Systick 的中断服务函数 (stm32f1xx_it.c 文件)

```
/**
* @brief  SysTick 中断服务函数.
* @param  None
* @retval None
*/
void SysTick_Handler(void)
{
   TimingDelay_Decrement();
   TimeStamp_Increment();
}
```

43.6.2.2.3 提供串口调试接口

MPL 代码库的调试信息输出函数都集中到了 log_stm32.c 文件中，我们可以为这些函数提供串口输出接口，以便把这些信息输出到上位机，见*代码清单 46_13*。

列表 13: 代码清单 46-13 串口调试接口 (log_stm32.c 文件)

```
/* 串口输出接口 */
int fputcc(int ch)
{
   /* 发送一个字节数据到 USART1 */
   USART_SendData(DEBUG_USART, (uint8_t) ch);

   /* 等待发送完毕 */
   while (USART_GetFlagStatus(DEBUG_USART, USART_FLAG_TXE) == RESET);

   return (ch);
}

/* 输出四元数数据 */
void eMPL_send_quat(long *quat)
{
   char out[PACKET_LENGTH];
   int i;
   if (!quat)
      return;
   memset(out, 0, PACKET_LENGTH);
   out[0] = '$';
   out[1] = PACKET_QUAT;
   out[3] = (char)(quat[0] >> 24);
   out[4] = (char)(quat[0] >> 16);
   out[5] = (char)(quat[0] >> 8);
   out[6] = (char)quat[0];
   out[7] = (char)(quat[1] >> 24);
   out[8] = (char)(quat[1] >> 16);
   out[9] = (char)(quat[1] >> 8);
```

(下页继续)

(续上页)

```
    out[10] = (char)quat[1];
    out[11] = (char)(quat[2] >> 24);
    out[12] = (char)(quat[2] >> 16);
    out[13] = (char)(quat[2] >> 8);
    out[14] = (char)quat[2];
    out[15] = (char)(quat[3] >> 24);
    out[16] = (char)(quat[3] >> 16);
    out[17] = (char)(quat[3] >> 8);
    out[18] = (char)quat[3];
    out[21] = '\r';
    out[22] = '\n';

    for (i=0; i<PACKET_LENGTH; i++) {
        fputcc(out[i]);
    }
}
```

上述代码中的 fputcc 函数是我们自己编写的串口输出接口，它与我们重定向 printf 函数定义的 fputc 函数功能很类似。下面的 eMPL_send_quat 函数是 MPL 库中的原函数，它用于打印“四元数信息”，在这个 log_stm32.c 文件中还有输出日志信息的 _MLPrintLog 函数，输出原始信息到专用上位机的 eMPL_send_data 函数，它们都调用了 fputcc 进行输出。

43.6.2.2.4 MPU6050 的中断接口

与上一小节中的基础实验不同，为了高效处理采样数据，MPL 代码库使用了 MPU6050 的 INT 中断信号，为此我们要给提供中断接口，见*代码清单 46_14*。

列表 14: 代码清单 46-14 中断接口 (stm32f1xx_it.c 文件)

```
#define MPU_IRQHandler                          EXTI9_5_IRQHandler
```

(下页继续)

(续上页)

```
void MPU_IRQHandler(void)
{
if (__HAL_GPIO_EXTI_GET_IT(MPU_INT_GPIO_PIN) != RESET) {
//确保是否产生了 EXTI Line 中断
      /* Handle new gyro*/
      gyro_data_ready_cb();

      __HAL_GPIO_EXTI_CLEAR_IT(MPU_INT_GPIO_PIN);      //清除中断标志位
   }
}
```

在工程中我们把 MPU6050 与 STM32 相连的引脚配置成了中断模式，上述代码是该引脚的中断服务函数，在中断里调用了 MPL 代码库的 gyro_data_ready_cb 函数，它设置了标志变量 hal.new_gyro，以通知 MPL 库有新的数据，其函数定义见*代码清单 46_15*。

列表 15: 代码清单 46-15 设置标志变量 (main.c 文件)

```
/* 每当有新的数据产生时，本函数会被中断服务函数调用，
* 在本工程中，它设置标志位用于指示及保护 FIFO 缓冲区
*/
void gyro_data_ready_cb(void)
{
   hal.new_gyro = 1;
}
```

43.6.2.2.5 main 函数执行流程

了解 MPL 移植需要提供的接口后，我们直接看 main 函数了解如何利用 MPL 库获取姿态数据，见*代码清单 46_16*。

列表 16: 代码清单 46-16 使用 MPL 进行姿态解算的过程

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
   inv_error_t result;
   unsigned char accel_fsr,  new_temp = 0;
   unsigned short gyro_rate, gyro_fsr;
   unsigned long timestamp;
   struct int_param_s int_param;

   SysTick_Init();

   /* LED 端口初始化 */
   LED_GPIO_Config();
   LED_BLUE;

   /* 串口通信初始化 */
   USART_Config();

   //MPU6050 中断引脚
   EXTI_Pxy_Config();
   //I2C 初始化
   I2cMaster_Init();

   printf("mpu 6050 test start");

   result = mpu_init(&int_param);
```

(下页继续)

(续上页)

```
  if (result) {
     LED_RED;
     MPL_LOGE("Could not initialize gyro.result =  %d\n",result);
  } else {
     LED_GREEN;
  }

  /* If you're not using an MPU9150 AND you're not using DMP features,
  * this function will place all slaves on the primary bus.
  * mpu_set_bypass(1);
  */

  result = inv_init_mpl();
  if (result) {
     MPL_LOGE("Could not initialize MPL.\n");
  }

  /* 计算 6 轴和 9 轴传感器的四元数 */
  inv_enable_quaternion();
  inv_enable_9x_sensor_fusion();

  /* 无运动状态时更新陀螺仪
  * WARNING: These algorithms are mutually exclusive.
  */
  inv_enable_fast_nomot();
  /* inv_enable_motion_no_motion(); */
  /* inv_set_no_motion_time(1000); */

  /* 当温度变化时更新 陀螺仪 */
  inv_enable_gyro_tc();

  /* 允许 read_from_mpl 使用 MPL APIs. */
```

(下页继续)

(续上页)

```
    inv_enable_eMPL_outputs();

    result = inv_start_mpl();
    if (result == INV_ERROR_NOT_AUTHORIZED) {
        while (1) {
            MPL_LOGE("Not authorized.\n");
        }
    }
    if (result) {
        MPL_LOGE("Could not start the MPL.\n");
    }

    /* 设置寄存器，开启陀螺仪 */
    /* 唤醒所有传感器 */

    mpu_set_sensors(INV_XYZ_GYRO | INV_XYZ_ACCEL);
    /* 把陀螺仪及加速度数据放进 FIFO */
    mpu_configure_fifo(INV_XYZ_GYRO | INV_XYZ_ACCEL);
    mpu_set_sample_rate(DEFAULT_MPU_HZ);

    /* 重新读取配置，确认前面的设置成功  */
    mpu_get_sample_rate(&gyro_rate);
    mpu_get_gyro_fsr(&gyro_fsr);
    mpu_get_accel_fsr(&accel_fsr);

    /* 使用 MPL 同步配置 */
    /* 设置每毫秒的采样率 */
    inv_set_gyro_sample_rate(1000000L / gyro_rate);
    inv_set_accel_sample_rate(1000000L / gyro_rate);

    /* 设置 chip-to-body 原点矩阵.
    * 设置硬件单位为 dps/g's/degrees 因子.
```

(下页继续)

(续上页)

```
    */
    inv_set_gyro_orientation_and_scale(
        inv_orientation_matrix_to_scalar(gyro_pdata.orientation),
        (long)gyro_fsr<<15);
    inv_set_accel_orientation_and_scale(
            inv_orientation_matrix_to_scalar(gyro_pdata.orientation),
            (long)accel_fsr<<15);

    /* 初始化硬件状态相关的变量. */

    hal.sensors = ACCEL_ON | GYRO_ON;

    hal.dmp_on = 0;
    hal.report = 0;
    hal.rx.cmd = 0;
    hal.next_pedo_ms = 0;
    hal.next_compass_ms = 0;
    hal.next_temp_ms = 0;

    /* 获取时间戳 */
    get_tick_count(&timestamp);

    /* 初始化 DMP 步骤:
        * 1. 调用 dmp_load_motion_driver_firmware().
            它会把 inv_mpu_dmp_motion_driver.h 文件中的 DMP 固件写入到 MPU 的存储空
间
        * 2. 把陀螺仪和加速度的原始数据矩阵送入 DMP.
        * 3. 注册姿态回调函数. 除非相应的特性使能了, 否则该回调函数不会被执行
        * 4. 调用 dmp_enable_feature(mask) 使能不同的特性.
        * 5. 调用 dmp_set_fifo_rate(freq) 设置 DMP 输出频率.
        * 6. 调用特定的特性控制相关的函数.
        *
```

(下页继续)

(续上页)

```
      * 调用 mpu_set_dmp_state(1) 使能 DMP. 该函数可在 DMP 运行时被
      * 重复调用设置使能或关闭
      * 以下是 inv_mpu_dmp_motion_driver.c 文件中的 DMP 固件提供的特性的简介:
      * DMP_FEATURE_LP_QUAT: 使用 DMP 以 200Hz 的频率产生一个只包含陀螺仪的四元
      数数据以高速的状态解算陀螺仪数据，减少错误
      * DMP_FEATURE_6X_LP_QUAT: 使用 DMP 以 200Hz 的频率产生陀螺仪/加速度，四元数␣
↪.
      它不能与前面的 DMP_FEATURE_LP_QUAT 同时使用
      * DMP_FEATURE_TAP: 检测 X, Y, 和 Z 轴.
      * DMP_FEATURE_ANDROID_ORIENT: 当屏幕翻转时，在四个方向产生一个事件
      * DMP_FEATURE_GYRO_CAL: 若 8s 内都没有运动，计算陀螺仪的数据
      * DMP_FEATURE_SEND_RAW_ACCEL: 添加原始 加速度 数据到 FIFO.
      * DMP_FEATURE_SEND_RAW_GYRO: 添加原始 陀螺仪 数据到 FIFO.
      * DMP_FEATURE_SEND_CAL_GYRO: 添加 校准后的 陀螺仪 数据到 FIFO.
      它不能与 DMP_FEATURE_SEND_RAW_GYRO. 同时使用
      */

  dmp_load_motion_driver_firmware();
  dmp_set_orientation(
        inv_orientation_matrix_to_scalar(gyro_pdata.orientation));
  dmp_register_tap_cb(tap_cb);

  dmp_register_android_orient_cb(android_orient_cb);
  /*
     * DMP 传感器整合只能工作在陀螺仪 +-2000dps 及加速度 +-2G 的量程上。
     */
  hal.dmp_features = DMP_FEATURE_6X_LP_QUAT | DMP_FEATURE_TAP |
     DMP_FEATURE_ANDROID_ORIENT | DMP_FEATURE_SEND_RAW_ACCEL |
           DMP_FEATURE_SEND_CAL_GYRO |DMP_FEATURE_GYRO_CAL;
  dmp_enable_feature(hal.dmp_features);
  dmp_set_fifo_rate(DEFAULT_MPU_HZ);
  mpu_set_dmp_state(1);
```

(下页继续)

(续上页)

```
hal.dmp_on = 1;

while (1) {

        unsigned long sensor_timestamp;
        int new_data = 0;
        if (USART_GetFlagStatus (DEBUG_USARTx, USART_FLAG_RXNE)) {
            /* 已通过串口接收到数据. 使用 handle_input 来处理串口接收到的命令
            * 这部分是处理 python 官方上位机串口发送的指令的。
            */
            USART_ClearFlag(DEBUG_USARTx, USART_FLAG_RXNE);

            handle_input();
        }
        get_tick_count(&timestamp);

        /* 温度数据不需要与陀螺仪数据那每次都采样，这里设置隔一段时间采样
        */
        if (timestamp > hal.next_temp_ms) {
            hal.next_temp_ms = timestamp + TEMP_READ_MS;
            new_temp = 1;
        }
        if (hal.new_gyro && hal.dmp_on) {
            short gyro[3], accel_short[3], sensors;
            unsigned char more;
            long accel[3], quat[4], temperature;
            /* 当使用 DMP 时，本函数从 FIFO 读取新的数据
            FIFO 中存储了陀螺仪、加速度、四元数及手势数据.
            传感器参数可告知调用者哪种有新数据
    * 例如,if sensors==(INV_XYZ_GYRO|INV_WXYZ_QUAT),FIFO 中就不包含加速度数据
            * 手势数据的解算由是否产生了手势运动事件来触发
            * 若产生了事件，应用函数会使用回调函数来通知
```

(下页继续)

(续上页)

```
        */
dmp_read_fifo(gyro,accel_short,quat,&sensor_timestamp,&sensors, &more);
        if (!more)
            hal.new_gyro = 0;
        if (sensors & INV_XYZ_GYRO) {
            /* 把新数据送入 MPL. */
            inv_build_gyro(gyro, sensor_timestamp);
            new_data = 1;
            if (new_temp) {
                new_temp = 0;
                /* 温度数据只用于陀螺仪的暂时计算 */
                mpu_get_temperature(&temperature, &sensor_timestamp);
                inv_build_temp(temperature, sensor_timestamp);
            }
        }
        if (sensors & INV_XYZ_ACCEL) {
            accel[0] = (long)accel_short[0];
            accel[1] = (long)accel_short[1];
            accel[2] = (long)accel_short[2];
            inv_build_accel(accel, 0, sensor_timestamp);
            new_data = 1;
        }
        if (sensors & INV_WXYZ_QUAT) {
            inv_build_quat(quat, 0, sensor_timestamp);
            new_data = 1;
        }
    }
    if (new_data) {
        inv_execute_on_data();
        /* 本函数读取补偿的后的传感器数据和经过 MPL 传感器融合后输出的数据
           输出的格式见 in eMPL_outputs.c 文件.
           这个函数在主机需要数据的时候调用即可，对调用频率无要求。
```

(下页继续)

(续上页)

```
            */
            read_from_mpl();
        }
    }
```

如您所见，main 函数非常长，而且我们只是摘抄了部分，在原工程代码中还有很多代码，以及不同模式下的条件判断分支，例如加入磁场数据使用 9 轴数据进行解算的功能 (这是 MPU9150 的功能，MPU6050 不支持) 以及其它工作模式相关的控制示例。上述 main 函数的主要执行流程概括如下：

(1) 初始化 STM32 的硬件，如 Systick、LED、调试串口、INT 中断引脚以及 I2C 外设的初始化；

(2) 调用 MPL 库函数 mpu_init 初始化传感器的基本工作模式 (以下过程调用的大部分都是 MPL 库函数，不再强调)；

(3) 调用 inv_init_mpl 函数初始化 MPL 软件库，初始化后才能正常进行解算；

(4) 设置各种运算参数，如四元数运算 (inv_enable_quaternion)、6 轴或 9 轴数据融合 (inv_enable_9x_sensor_fusion) 等等；

(5) 设置传感器的工作模式 (mpu_set_sensors)、采样率 (mpu_set_sample_rate)、分辨率 (inv_set_gyro_orientation_and_scale) 等等；

(6) 当 STM32 驱动、MPL 库、传感器工作模式、DMP 工作模式等所有初始化工作都完成后进行 while 循环；

(7) 在 while 循环中检测串口的输入，若串口有输入，则调用 handle_input 根据串口输入的字符 (命令)，切换工作方式。这部分主要是为了支持上位机通过输入命令，根据进行不同的处理，如开、关加速度信息的采集或调试信息的输出等；

(8) 在 while 循环中检测是否有数据更新 (if (hal.new_gyro && hal.dmp_on))，当有数据更新的时候产生 INT 中断，会使 hal.new_gyro 置 1 的，从而执行 if 里的条件代码；

(9) 使用 dmp_read_fifo 把数据读取到 FIFO，这个 FIFO 是指 MPL 软件库定义的一个缓冲区，用来缓冲最新采集得的数据；

(10) 调用 inv_build_gyro、inv_build_temp、inv_build_accel 及 inv_build_quat 函数处理数据角速度、

温度、加速度及四元数数据，并对标志变量 new_data 置 1；

(11) 在 while 循环中检测 new_data 标志位，当有新的数据时执行 if 里的条件代码；

(12) 调用 inv_execute_on_data 函数更新所有数据及状态；

(13) 调用 read_from_mpl 函数向主机输出最新的数据。

43.6.2.2.6 数据输出接口

在上面 main 中最后调用的 read_from_mpl 函数演示了如何调用 MPL 数据输出接口，通过这些接口我们可以获得想要的数据，其函数定义见*代码清单 46_17*。

列表 17: 代码清单 46-17 MPL 的数据输出接口 (main.c)

```
/* 从 MOL 中获取数据 MPL.
*/
static void read_from_mpl(void)
{
   long msg, data[9];
   int8_t accuracy;
   unsigned long timestamp;
   float float_data[3] = {0};

   MPU_DEBUG_FUNC();
   if (inv_get_sensor_type_quat(data, &accuracy, (inv_time_t*)&timestamp)) {
         /* 发送四元数数据包到 PC. 这些数据由 python 上位机构建 3D 模型姿态,
         因此每当有新数据就发送上去
         */
         eMPL_send_quat(data);

         /* 使用命令, 可以控制它发送指定的数据包 */
         if (hal.report & PRINT_QUAT)
            eMPL_send_data(PACKET_DATA_QUAT, data);
```

(下页继续)

(续上页)

```
    }

    if (hal.report & PRINT_ACCEL) {
        if (inv_get_sensor_type_accel(data, &accuracy,
                                      (inv_time_t*)&timestamp))
            eMPL_send_data(PACKET_DATA_ACCEL, data);

    }
    if (hal.report & PRINT_GYRO) {
        if (inv_get_sensor_type_gyro(data, &accuracy,
                                     (inv_time_t*)&timestamp))
            eMPL_send_data(PACKET_DATA_GYRO, data);

    }

    if (hal.report & PRINT_EULER) {
        if (inv_get_sensor_type_euler(data, &accuracy,
                                      (inv_time_t*)&timestamp))
            eMPL_send_data(PACKET_DATA_EULER, data);

    }

    /********************* 使用液晶屏显示数据 ***************************/
#ifdef USE_LCD_DISPLAY
    if (1) {
        char cStr [ 70 ];
        unsigned long timestamp,step_count,walk_time;

        /* 获取欧拉角 */
        if (inv_get_sensor_type_euler(data, &accuracy,(inv_time_t*)&
↪timestamp)) {
            //inv_get_sensor_type_euler 读出的数据是 Q16 格式，所以左移 16 位.
```

(下页继续)

(续上页)

```
            sprintf ( cStr, "Pitch :  %.4f  ", data[0]*1.0/(1<<16) );
            ILI9341_DispString_EN(30,90,cStr);

            //inv_get_sensor_type_euler 读出的数据是 Q16 格式，所以左移 16 位.
            sprintf ( cStr, "Roll :  %.4f  ", data[1]*1.0/(1<<16) );
            ILI9341_DispString_EN(30,110,cStr);

            //inv_get_sensor_type_euler 读出的数据是 Q16 格式，所以左移 16 位.
            sprintf ( cStr, "Yaw :  %.4f  ", data[2]*1.0/(1<<16) );
            ILI9341_DispString_EN(30,130,cStr);

            /* 温度 */
            mpu_get_temperature(data,(inv_time_t*)&timestamp);

            //inv_get_sensor_type_euler 读出的数据是 Q16 格式，所以左移 16 位.
            sprintf ( cStr, "Temperature :  %.2f  ", data[0]*1.0/(1<<16) );
            ILI9341_DispString_EN(30,150,cStr);

      }
      /* 获取步数 */
      get_tick_count(&timestamp);
      if (timestamp > hal.next_pedo_ms) {

            hal.next_pedo_ms = timestamp + PEDO_READ_MS;
            dmp_get_pedometer_step_count(&step_count);
            dmp_get_pedometer_walk_time(&walk_time);

            sprintf(cStr, "Walked steps :  %ld  steps over  %ld
↪milliseconds..",step_count,walk_time);

            ILI9341_DispString_EN(0,180,cStr);
```

(下页继续)

(续上页)

```
        }
    }
#endif
    /* 以下省略 */
}
```

上述代码展示了使用 inv_get_sensor_type_quat、inv_get_sensor_type_accel、inv_get_sensor_type_gyro、inv_get_sensor_type_euler 及 dmp_get_pedometer_step_count 函数分别获取四元数、加速度、角速度、欧拉角及计步器数据。

代码中的 eMPL_send_data 函数是使用串口按照 PYTHON 上位机格式进行提交数据，上位机根据这些数据对三维模型作相应的旋转。

另外我们自己在代码中加入了液晶显示的代码 (#ifdef USE_LCD_DISPLAY 宏内的代码)，它把这些数据输出到实验板上的液晶屏上。

您可根据自己的数据使用需求，参考这个 read_from_mpl 函数对数据输出接口的调用方式，编写自己的应用。

43.6.2.3 下载验证

直接下载本程序到开发板，在液晶屏上会观察到姿态角、温度、计步器数据，改变开发板的姿态，数据会更新 (计步器数据要模拟走路才会更新)，若直接连接串口调试助手，会接收到一系列的乱码信息，这是正常的，这些数据需要使用官方的 Python 上位机解码。

本实验适用于官方提供的 Python 上位机，它可以把采样的数据传送到上位机，上位机会显示三维模式的姿态。

注意：以下内容仅针对有 Python 编程语言基础的用户，若您不会 Python，而又希望观察到三维模型的姿态，请参考下一小节的实验，它的使用更为简单。

43.6.2.3.1 Python 上位机源代码及说明

MPU6050 官方提供的上位机的使用说明可在配套资料《motion_driver6.12》源码包 documentation 文件夹里的《Motion Driver 6.12 –Getting Started Guide》找到。上位机的源码在《motion_driver6.12》源码包的 eMPL-pythonclient 文件夹，里边有三个 python 文件，见图 *46_13*。

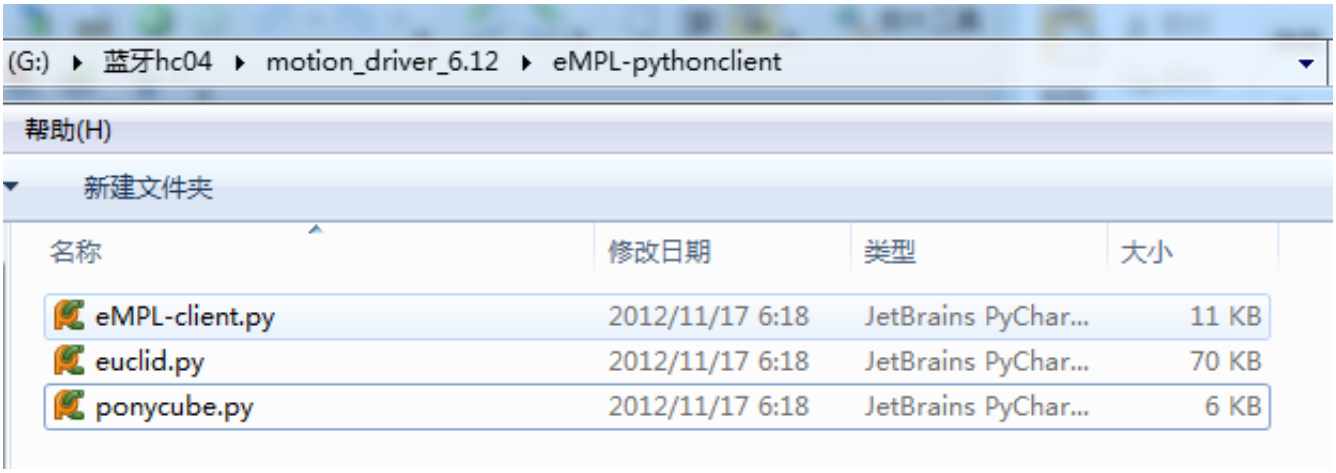


图 46-13 源码包里的 python 上位机源码

43.6.2.3.2 安装 Python 环境

要利用上面的源码，需要先安装 Python 环境，该上位机支持 python2.7 环境 (仅支持 32 位)，并且需要安装 Pyserial 库 (仅支持 Pyserial2.6，不支持 3 版本)、Pygame 库。

可通过如下网址找到安装包。

Python: https://www.python.org/downloads/

Pyserial: https://pypi.python.org/pypi/pyserial

Pygame: http://www.pygame.org/download.shtml

43.6.2.3.3 Python 上位机的使用步骤

- 先把本 STM32 工程代码编译后下载到开发板上运行，确认开发板的 USB TO USART 接口已与电脑相连，正常时开发板的液晶屏现象跟上一章例程的现象一样。
- 使用命令行切换到 python 上位机的目录，执行如下命令：

  python eMPL-client.py <COM PORT NUMBER>

  其中 <COM PORT NUMBER> 参数是 STM32 开发板在电脑端的串口设备号，运行命令后会弹出一个 3D 图形窗口，显示陀螺仪的姿态，见图 46-14。(图中的 “python2_32” 是本机的 python2.7-32 位 python 命令的名字，用户默认用 “python” 命令即可。)

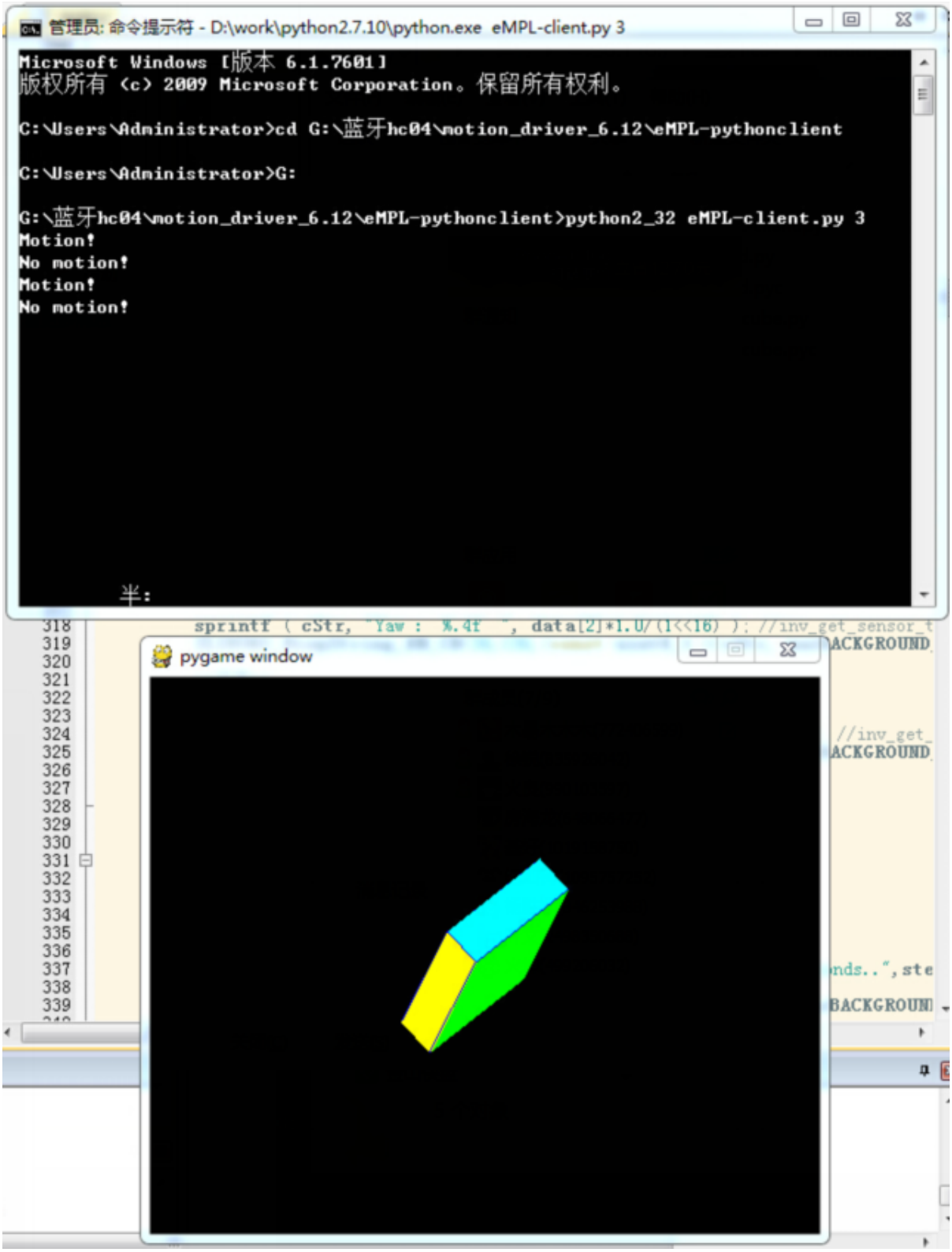

管理员: 命令提示符 - D:\work\python2.7.10\python.exe eMPL-client.py 3
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
C:\Users\Administrator>cd G:\蓝牙hc04\motion_driver_6.12\eMPL-pythonclient
C:\Users\Administrator>G:
G:\蓝牙hc04\motion_driver_6.12\eMPL-pythonclient>python2_32 eMPL-client.py 3
Motion!
No motion!
Motion!
No motion!
半:
pygame window

图 46-14 运行 python 上位机

- 这个上位机还可以接收命令来控制 STM32 进行数据输出，选中图中的 pygame window 窗口(弹出来的 3D 图形窗口)，然后按下键盘的字母“a ”键，命令行窗口就会输出加速度信息，按下“g”键，就会输出陀螺仪信息。命令集说明如下：

- ‘8’：Toggles Accel Sensor
- ‘9’：Toggles Gyro Sensor
- ‘0’：Toggles Compass Sensor
- ‘a’：Prints Accel Data
- ‘g’：Prints Gyro Data
- ‘c’：Prints Compass Data
- ‘e’：Prints Eular Data in radius
- ‘r’：Prints Rotational Matrix Data
- ‘q’：Prints Quaternions
- ‘h’：Prints Heading Data in degrees
- ‘i’：Prints Linear Acceleration data
- ‘o’：Prints Gravity Vector data
- ‘w’：Get compass accuracy and status
- ‘d’：Register Dump
- ‘p’：Turn on Low Power Accel Mode at 20Hz sampling
- ‘l’：Load calibration data from flash memory
- ‘s’：Save calibration data to flash memory
- ‘t’：run factory self test and calibration routine
- ‘1’：Change sensor output data rate to 10Hz

- ‘2’：Change sensor output data rate to 20Hz
- ‘3’：Change sensor output data rate to 40Hz
- ‘4’：Change sensor output data rate to 50Hz
- ‘5’：Change sensor output data rate to 100Hz
- ‘,’：set interrupts to DMP gestures only
- ‘.’：set interrupts to DMP data ready
- ‘6’：Print Pedometer data
- ‘7’：Reset Pedometer data
- ‘f’：Toggle DMP on/off
- ‘m’：Enter Low Power Interrupt Mode
- ‘x’：Reset the MSP430
- ‘v’：Toggle DMP Low Power Quaternion Generation

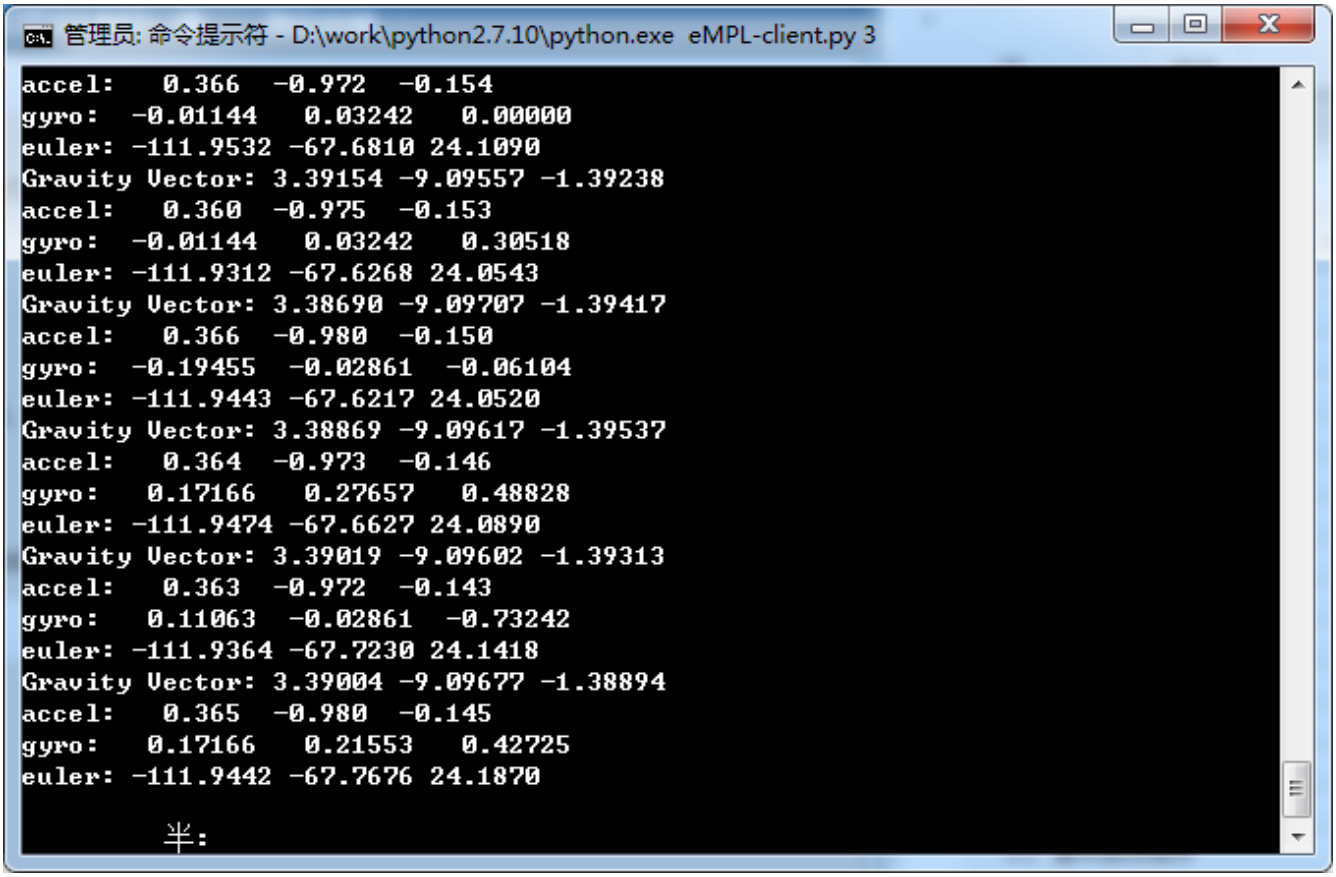


图 46-15 在 3D 窗口输入命令后的命令行窗口输出

在前面提到的 STM32 代码 read_from_mpl 函数，根据标志位决定是否获取欧拉角、加速度、陀螺仪等数据并上传操作，就是配合这个 python 上位机执行的。(在 main 函数里有个检测串口输入的代码，若检测到串口输入，则设置相应的标志位。)

有兴趣的读者可以根据这个官方的 python 上位机，自己编写上位机控制程序。

43.7 MPU6050—使用第三方上位机

上一小节中的实验必须配合使用官方提供的上位机才能看到三维模型，而且功能比较简单，所以在小节中我们演示如何把数据输出到第三方的上位机，直观地观察设备的姿态。

实验中我们使用的是“匿名飞控地面站 0512”版本的上位机，关于上位机的通讯协议可查阅《飞

控通信协议》文档，或到他们的官方网站了解。

43.7.1 硬件设计

硬件设计与上一小节实验中的完全一样，同样使用了中断 INT 引脚获取数据状态，默认使用软件 I2C 通讯。

43.7.2 软件设计

本小节讲解的是“MPU6050_DMP 测试例程”实验，请打开配套的代码工程阅读理解。本小节的内容主体跟上一小节一样，区别主要是当获取得到数据后，本实验根据“匿名飞控”上位机的数据格式要求上传数据。

43.7.2.1 程序设计要点

(1) 了解上位机的通讯协议；

(2) 根据协议格式上传数据到上位机；

43.7.2.2 代码分析

43.7.2.2.1 通讯协议

要按照上位机的格式上传数据，首先要了解它的通讯协议，本实验中的上位机协议说明见表 46-4。

表 46-4 匿名上位机的通讯协议 (部分)

帧	帧头	功能字	长度	数据	校验
STATUS	AAAA	01	LEN	int16 ROL*100 int16 PIT*100 int16 YAW*100 int32 ALT_USE u8 ARMED : A0 加锁 A1 解锁	SUM
SENSER	AAAA	02	LEN	int16 ACC_X int16 ACC_Y int16 ACC_Z int16 GYRO_X int16 GYRO_Y int16 GYRO_Z int16 MAG_X int16 MAG_Y int16 MAG_Z	SUM

表中说明了两种数据帧，分别是 STATUS 帧及 SENSER 帧，数据帧中包含帧头、功能字、长度、主体数据及校验和。“帧头”用于表示数据包的开始，均使用两个字节的 0xAA 表示；“功能字”用于区分数据帧的类型，0x01 表示 STATUS 帧，0x02 表示 SENSER 帧；“长度”表示后面主体数据内容的字节数；“校验和”用于校验，它是前面所有内容的和。

其中的 STATUS 帧用于向上位机传输横滚角、俯仰角及偏航角的值 (100 倍)，SENSER 帧用于传

输加速度、角速度及磁场强度的原始数据。

43.7.2.2.2 发送数据包

根据以上数据格式的要求，我们定义了两个函数，分别用于发送 STATUS 帧及 SENSER 帧，见*代码清单 46_18*。

列表 18: 代码清单 46-18 发送数据包（main.c 文件）

```
#define BYTE0(dwTemp)       (*(char *)(&dwTemp))
#define BYTE1(dwTemp)       (*((char *)(&dwTemp) + 1))
#define BYTE2(dwTemp)       (*((char *)(&dwTemp) + 2))
#define BYTE3(dwTemp)       (*((char *)(&dwTemp) + 3))

/**
* @brief  控制串口发送 1 个字符
* @param  c: 要发送的字符
* @retval none
*/
void usart_send_char(uint8_t c)
{  //循环发送，直到发送完毕
   while (USART_GetFlagStatus(DEBUG_USART,USART_FLAG_TXE)==RESET);
   USART_SendData(DEBUG_USART,c);
}
/* 函数功能：根据匿名最新上位机协议写的显示姿态的程序（上位机 0512 版本）
* 具体协议说明请查看上位机软件的帮助说明。
*/
void Data_Send_Status(float Pitch,float Roll,float Yaw)
{
   unsigned char i=0;
   unsigned char _cnt=0,sum = 0;
   unsigned int _temp;
   u8 data_to_send[50];
```

(下页继续)

(续上页)

```
data_to_send[_cnt++]=0xAA;
data_to_send[_cnt++]=0xAA;
data_to_send[_cnt++]=0x01;
data_to_send[_cnt++]=0;

_temp = (int)(Roll*100);
data_to_send[_cnt++]=BYTE1(_temp);
data_to_send[_cnt++]=BYTE0(_temp);
_temp = 0-(int)(Pitch*100);
data_to_send[_cnt++]=BYTE1(_temp);
data_to_send[_cnt++]=BYTE0(_temp);
_temp = (int)(Yaw*100);
data_to_send[_cnt++]=BYTE1(_temp);
data_to_send[_cnt++]=BYTE0(_temp);
_temp = 0;
data_to_send[_cnt++]=BYTE3(_temp);
data_to_send[_cnt++]=BYTE2(_temp);
data_to_send[_cnt++]=BYTE1(_temp);
data_to_send[_cnt++]=BYTE0(_temp);

data_to_send[_cnt++]=0xA0;

data_to_send[3] = _cnt-4;
//和校验
for (i=0; i<_cnt; i++)
      sum+= data_to_send[i];
data_to_send[_cnt++]=sum;

//串口发送数据
for (i=0; i<_cnt; i++)
      usart_send_char(data_to_send[i]);
```

(下页继续)

(续上页)

```
}

/* 函数功能：根据匿名最新上位机协议写的显示传感器数据（上位机 0512 版本）
* 具体协议说明请查看上位机软件的帮助说明。
*/
void Send_Data(int16_t *Gyro,int16_t *Accel)
{
   unsigned char i=0;
   unsigned char _cnt=0,sum = 0;
//  unsigned int _temp;
   u8 data_to_send[50];

   data_to_send[_cnt++]=0xAA;
   data_to_send[_cnt++]=0xAA;
   data_to_send[_cnt++]=0x02;
   data_to_send[_cnt++]=0;

   data_to_send[_cnt++]=BYTE1(Accel[0]);
   data_to_send[_cnt++]=BYTE0(Accel[0]);
   data_to_send[_cnt++]=BYTE1(Accel[1]);
   data_to_send[_cnt++]=BYTE0(Accel[1]);
   data_to_send[_cnt++]=BYTE1(Accel[2]);
   data_to_send[_cnt++]=BYTE0(Accel[2]);

   data_to_send[_cnt++]=BYTE1(Gyro[0]);
   data_to_send[_cnt++]=BYTE0(Gyro[0]);
   data_to_send[_cnt++]=BYTE1(Gyro[1]);
   data_to_send[_cnt++]=BYTE0(Gyro[1]);
   data_to_send[_cnt++]=BYTE1(Gyro[2]);
   data_to_send[_cnt++]=BYTE0(Gyro[2]);
   data_to_send[_cnt++]=0;
   data_to_send[_cnt++]=0;
```

(下页继续)

(续上页)

```
    data_to_send[_cnt++]=0;
    data_to_send[_cnt++]=0;
    data_to_send[_cnt++]=0;
    data_to_send[_cnt++]=0;

    data_to_send[3] = _cnt-4;
    //和校验
    for (i=0; i<_cnt; i++)
          sum+= data_to_send[i];
    data_to_send[_cnt++]=sum;

    //串口发送数据
    for (i=0; i<_cnt; i++)
          usart_send_char(data_to_send[i]);
}
```

函数比较简单，就是根据输入的内容，一字节一字节地按格式封装好，然后调用串口发送到上位机。

43.7.2.2.3 发送数据

与上一小节一样，我们使用 read_from_mpl 函数输出数据，由于使用了不同的上位机，所以我们修改了它的具体内容，见代码清单 46_19。

列表 19: 代码清单 46-19 read_from_mpl 函数 (main.c 文件)

```
extern struct inv_sensor_cal_t sensors;
/*
* 从 MPL 中获取数据.
*/
```

(下页继续)

(续上页)

```
static void read_from_mpl(void)
{
    float Pitch,Roll,Yaw;
    int8_t accuracy;
    unsigned long timestamp;
    long  data[3];
    /* 获取欧拉角 */
    inv_get_sensor_type_euler(data, &accuracy,(inv_time_t*)&timestamp);

    //inv_get_sensor_type_euler 读出的数据是 Q16 格式，所以左移 16 位.
    Pitch =data[0]*1.0/(1<<16) ;
    Roll = data[1]*1.0/(1<<16);
    Yaw = data[2]*1.0/(1<<16);
    /* 向匿名上位机发送姿态 */
    Data_Send_Status(Pitch,Roll,Yaw);
    /* 向匿名上位机发送原始数据 */
    Send_Data((int16_t *)&sensors.gyro.raw,(int16_t *)&sensors.accel.raw);
}
```

代码中调用 inv_get_sensor_type_euler 获取欧拉角，然后调用 Data_Send_Status 格式上传到上位机，而加速度及角速度的原始数据直接从 sensors 结构体变量即可获取，获取后调用 Send_Data 发送出去。

43.7.2.3 下载验证

直接下载本程序到开发板，在液晶屏上会观察到姿态角、温度、计步器数据，改变开发板的姿态，数据会更新 (计步器数据要模拟走路才会更新)，若直接连接串口调试助手，会接收到一系列的乱码信息，这是正常的，这些数据需要使用“匿名飞控地面站”上位机解码。

若通过液晶屏的信息了解到 MPU6050 模块已正常工作，则可进一步在电脑上使用“ANO_TC 匿名飞控地面站-0512.exe”(以下简称“匿名上位机”) 软件查看可视化数据。

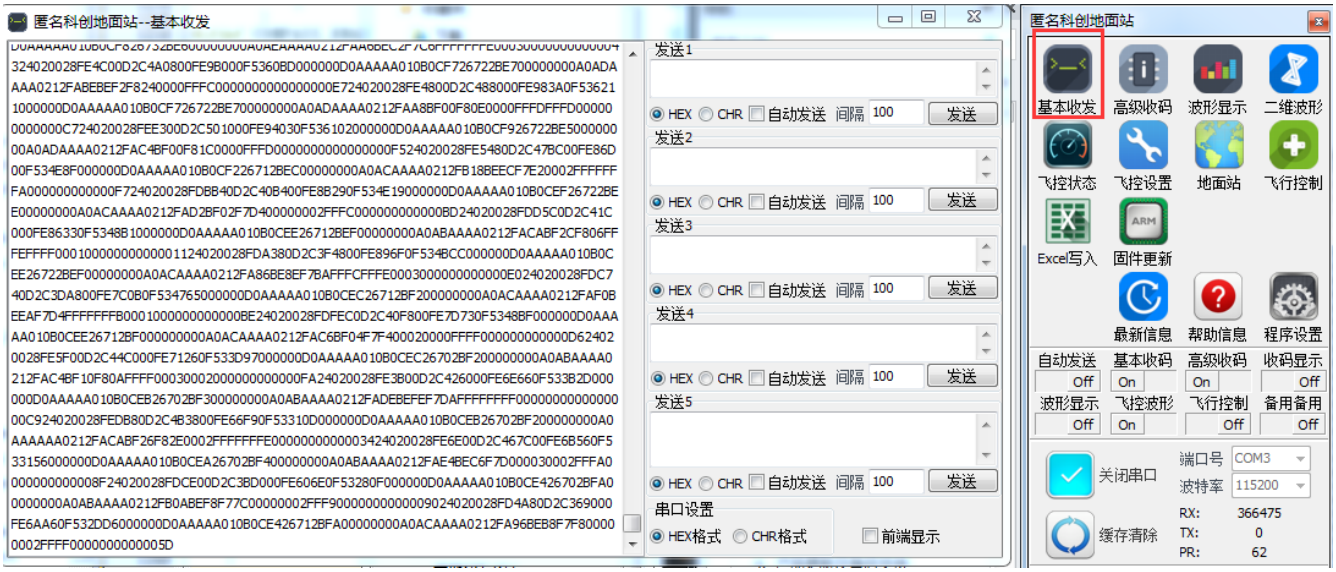
实验步骤如下：

(1) 确认开发板的 USB TO USART 接口已与电脑相连，确认电脑端能查看到该串口设备。

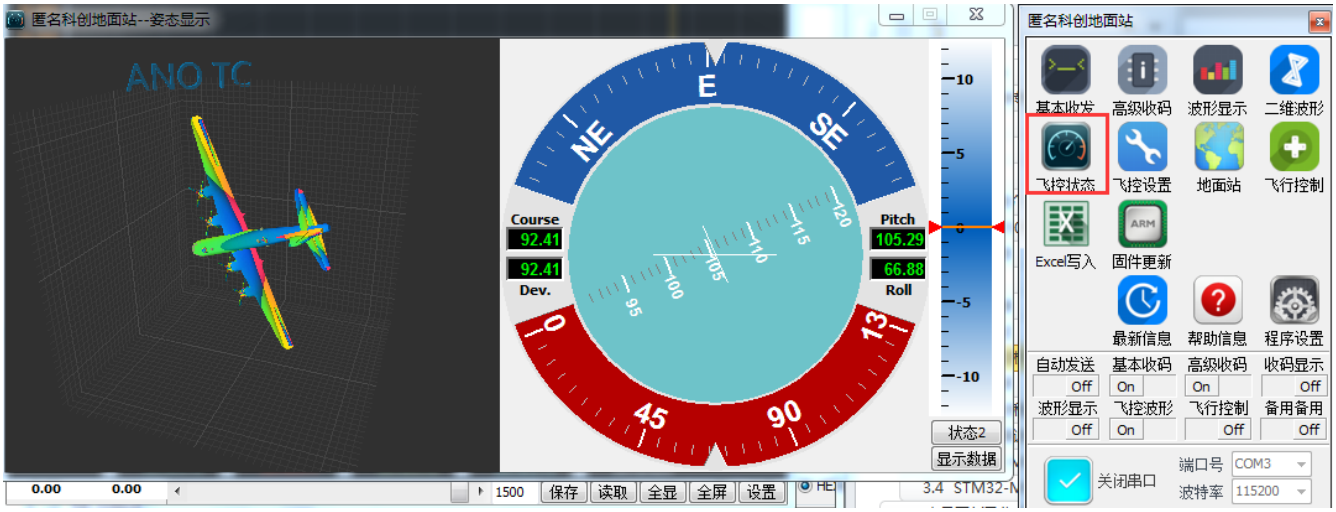
(2) 打开配套资料里的“匿名上位机”软件，在软件界面打开开发板对应的串口 (波特率为 115200)，把“基本收码”、“高级收码”、“飞控波形”功能设置为 on 状态。点击上方图中的基本收发、波形显示、飞控状态图标，会弹出窗口。具体见下文软件配置图。

(3) 在软件的“基本收发”、“波形显示”、“飞控状态”页面可看到滚动数据、随着模块晃动而变化的波形以及模块姿态的 3D 可视化图形。

匿名科创地面站
基本收发
高级收码
波形显示
二维波形
飞控状态
飞控设置
地面站
飞行控制
Excel写入
固件更新
最新信息
帮助信息
程序设置
自动发送
Off
基本收码
On
高级收码
On
收码显示
Off
波形显示
Off
飞控波形
On
飞行控制
Off
备用备用
Off
关闭串口
端口号
COM3
波特率
115200
缓存清除
RX:
49947
TX:
0
PR:
62


匿名科创地面站--基本收发
发送1
发送2
发送3
发送4
发送5
HEX CHR 自动发送 间隔 100 发送
串口设置
HEX格式 CHR格式 前端显示
匿名科创地面站
基本收发
高级收码
波形显示
二维波形
飞控状态
飞控设置
地面站
飞行控制
Excel写入
固件更新
最新信息
帮助信息
程序设置
自动发送
基本收码
高级收码
收码显示
波形显示
飞控波形
飞行控制
备用备用
关闭串口
端口号 COM3
波特率 115200
缓存清除
RX: 366475
TX: 0
PR: 62

匿名科创地面站--波形显示
20,000
15,000
10,000
5,000
0
-5,000
-10,000
-15,000
-20,000
-25,000
-30,000
ACC_X
ACC_Y
ACC_Z
GYRO_X
GYRO_Y
GYRO_Z
MAG_X
MAG_Y
MAG_Z
ROL
PIT
YAW
ALT
LAT
LNG
Series16
Series17
Series18
Series19
Series20
0.00
0.00
1500
保存
读取
全显
全屏
设置
匿名科创地面站
基本收发
高级收码
波形显示
二维波形
飞控状态
飞控设置
地面站
飞行控制
Excel写入
固件更新
最新信息
帮助信息
程序设置
自动发送
基本收码
高级收码
收码显示
波形显示
飞控波形
飞行控制
备用备用
关闭串口
端口号 COM3
波特率 115200
缓存清除
RX: 25923
TX: 0
PR: 62


匿名科创地面站--姿态显示
ANO TC
E
NE
SE
Course
92.41
92.41
Dev.
Pitch
105.29
66.88
Roll
状态2
显示数据
0.00
0.00
1500
保存
读取
全显
全屏
设置
匿名科创地面站
基本收发
高级收码
波形显示
二维波形
飞控状态
飞控设置
地面站
飞行控制
Excel写入
固件更新
最新信息
帮助信息
程序设置
自动发送
基本收码
高级收码
收码显示
Off
On
On
Off
波形显示
飞控波形
飞行控制
备用备用
Off
On
Off
Off
关闭串口
端口号
COM3
波特率
115200

第 44 章　OV7725 摄像头驱动

本章参考资料：《STM32F10x 参考手册》、《STM32F10x 数据手册》。

关于开发板配套的 OV7725 摄像头参数可查阅《ov7725datasheet》配套资料获知。

STM32 的处理速度比传统的 8、16 位机快得多，所以使用它驱动摄像头采集图像信息并进行基本的加工处理非常适合，本章讲解使用 STM32 驱动 OV7725 型号的摄像头。

44.1 摄像头简介

在各类信息中，图像含有最丰富的信息，作为机器视觉领域的核心部件，摄像头被广泛地应用在安防、探险以及车牌检测等场合。摄像头按输出信号的类型来看可以分为数字摄像头和模拟摄像头，按照摄像头图像传感器材料构成来看可以分为 CCD 和 CMOS。现在智能手机的摄像头绝大部分都是 CMOS 类型的数字摄像头。

44.1.1 数字摄像头跟模拟摄像头区别

- 输出信号类型：数字摄像头输出信号为数字信号，模拟摄像头输出信号为标准的模拟信号。
- 接口类型：数字摄像头有 usb 接口 (比如常见的 pc 端免驱摄像头)、IEE1394 火线接口 (由苹果公司领导的开发联盟开发的一种高速度传送接口，数据传输率高达 800Mbps)、千兆网接口（网络摄像头）。模拟摄像头多采用 AV 视频端子（信号线 + 地线）或 S-VIDEO（即莲花头–SUPER VIDEO，是一种五芯的接口，由两路视频亮度信号、两路视频色度信号和一路公共屏蔽地线共五条芯线组成）。
- 分辨率：模拟摄像头的感光器件，其像素指标一般维持在 752(H)*582(V) 左右的水平，像素数一般情况下维持在 41W 左右。数字摄像头分辨率一般从数十万到数百万甚至数千万。但这并不能说明数字摄像头的成像分辨率就比模拟摄像头的高，原因在于模拟摄像头输出的是模拟视频信号，一般直接输入至电视或监视器，其感光器件的分辨率与电视信号的扫

描数呈一定的换算关系，图像的显示介质已经确定，因此模拟摄像头的感光器件分辨率不是不能做高，而是依据于实际情况没必要做这么高。

44.1.2 CCD 与 CMOS 的区别

摄像头的图像传感器 CCD 与 CMOS 传感器主要区别如下：

- 成像材料

  CCD 与 CMOS 的名称跟它们成像使用的材料有关，CCD 是“电荷耦合器件”(Charge Coupled Device) 的简称，而 CMOS 是“互补金属氧化物半导体”(Complementary Metal Oxide Semiconductor) 的简称。

- 功耗

  由于 CCD 的像素由 MOS 电容构成，读取电荷信号时需使用电压相当大 (至少 12V) 的二相或三相或四相时序脉冲信号，才能有效地传输电荷。因此 CCD 的取像系统除了要有多个电源外，其外设电路也会消耗相当大的功率。有的 CCD 取像系统需消耗 2~5W 的功率。而 CMOS 光电传感器件只需使用一个单电源 5V 或 3V，耗电量非常小，仅为 CCD 的 1/8~1/10，有的 CMOS 取像系统只消耗 20~50mW 的功率。

- 成像质量

  CCD 传感器件制作技术起步早，技术成熟，采用 PN 结或二氧化硅 (sio2) 隔离层隔离噪声，所以噪声低，成像质量好。与 CCD 相比，CMOS 的主要缺点是噪声高及灵敏度低，不过现在随着 CMOS 电路消噪技术的不断发展，为生产高密度优质的 CMOS 传感器件提供了良好的条件，现在的 CMOS 传感器已经占领了大部分的市场，主流的单反相机、智能手机都已普遍采用 CMOS 传感器。

44.2 OV7725 摄像头

本章主要讲解实验板配套的摄像头，它的实物见图 *47_1*，该摄像头主要由镜头、图像传感器、板载电路、FIFO 缓存及下方的信号引脚组成。

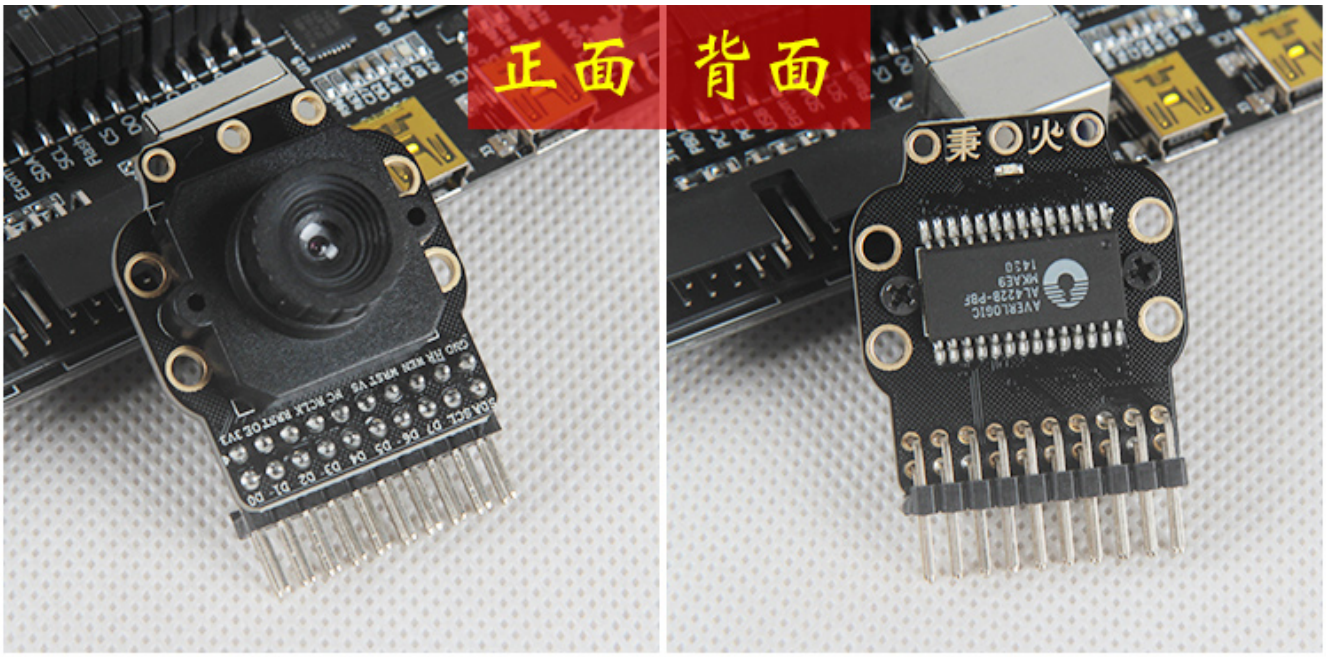


图 47-1 实验板配套的 OV7725 摄像头

镜头部件包含一个镜头座和一个可旋转调节距离的凸透镜，通过旋转可以调节焦距，正常使用时，镜头座覆盖在电路板上遮光，光线只能经过镜头传输到正中央的图像传感器，它采集光线信号，采集得的数据被缓存到摄像头背面的 FIFO 缓存中，然后外部器件通过下方的信号引脚获取拍摄得到的图像数据。

44.2.1 OV7725 传感器简介

若拆开摄像头座，在摄像头的正下方可看到 PCB 板上的一个方形器件，它是摄像头的核心部件，型号为 OV7725 的 CMOS 类型数字图像传感器。该传感器支持输出最大为 30 万像素的图像(640x480 分辨率)，它的体积小，工作电压低，支持使用 VGA 时序输出图像数据，输出图像的数据格式支持 YUV(422/420)、YCbCr422 以及 RGB565 格式。它还可以对采集得的图像进行补偿，

支持伽玛曲线、白平衡、饱和度、色度等基础处理。

44.2.2 OV7725 引脚及功能框架图

OV7725 传感器采用 BGA 封装，它的前端是采光窗口，引脚都在背面引出，引脚的分布见图 *47_2*。

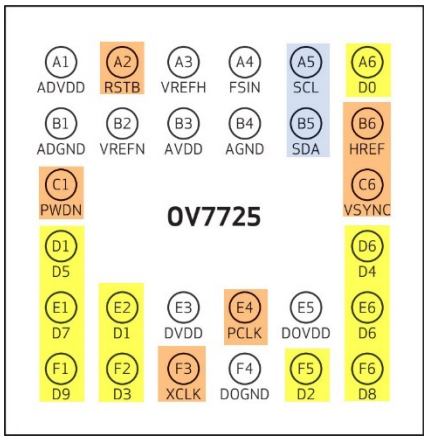


图 47-2 OV7725 管脚图

图中的非彩色部分是电源相关的引脚，彩色部分是主要的信号引脚，其介绍如表 47-1。

表 47-1 OV7725 管脚

管脚名称	管脚类型	管脚描述
RSTB	输入	系统复位管脚，低电平有效
PWDN	输入	掉电/省电模式 (高电平有效)
HREF	输出	行同步信号
VSYNC	输出	场同步信号
PCLK	输出	像素时钟
XCLK	输入	系统时钟输入端口
SCL	输入	SCCB 总线的时钟线
SDA	I/O	SCCB 总线的数据线
D0…D9	输出	像素数据端口

下面我们配合图 47-3 中的 OV7725 功能框图讲解这些信号引脚。

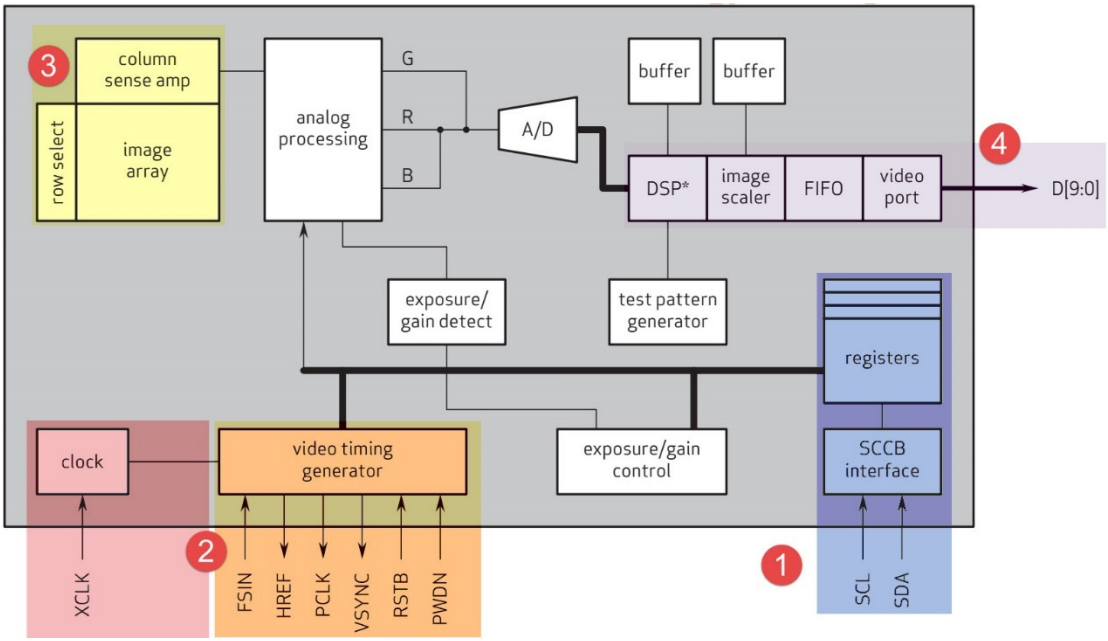


图 47-3 OV7725 功能框图

(1) 控制寄存器

标号处的是 OV7725 的控制寄存器，它根据这些寄存器配置的参数来运行，而这些参数是由外部控制器通过 SCL 和 SDA 引脚写入的，SCL 与 SDA 使用的通讯协议 SCCB 跟 I2C 十分类似，在 STM32 中我们完全可以直接用 I2C 硬件外设来控制。

(2) 通信、控制信号及时钟

标号处包含了 OV7725 的通信、控制信号及外部时钟，其中 PCLK、HREF 及 VSYNC 分别是像素同步时钟、行同步信号以及帧同步信号，这与液晶屏控制中的 VGA 信号是很类似的。RSTB 引脚为低电平时，用于复位整个传感器芯片，PWDN 用于控制芯片进入低功耗模式。注意最后的一个 XCLK 引脚，它跟 PCLK 是完全不同的，XCLK 是用于驱动整个传感器芯片的时钟信号，是外部输入到 OV7725 的信号；而 PCLK 是 OV7725 输出数据时的同步信号，它是由 OV7725 输出的信号。XCLK 可以外接晶振或由外部控制器提供，若要类比 XCLK 之于 OV7725 就相当于 HSE 时钟输入引脚与 STM32 芯片的关系，PCLK 引脚可类比 STM32 的 I2C 外设的 SCL 引脚。

(3) 感光矩阵

标号处的是感光矩阵，光信号在这里转化成电信号，经过各种处理，这些信号存储成由一个个像素点表示的数字图像。

(4) 数据输出信号

标号处包含了 DSP 处理单元，它会根据控制寄存器的配置做一些基本的图像处理运算。这部分还包含了图像格式转换单元及压缩单元，转换出的数据最终通过 D0-D9 引脚输出，一般来说我们使用 8 根据数据线来传输，这时仅使用 D2-D9 引脚。

44.2.3 SCCB 时序

外部控制器对 OV7725 寄存器的配置参数是通过 SCCB 总线传输过去的，而 SCCB 总线跟 I2C 十分类似，所以在 STM32 驱动中可以直接使用片上 I2C 外设与它通讯。关于 SCCB 协议的完整内容可查看配套资料里的《SCCB 协议》文档，下面进行简单介绍。

44.2.3.1 SCCB 的起始、停止信号及数据有效性

SCCB 的起始信号、停止信号及数据有效性与 I2C 完全一样，见图 *47_4* 及图 *47_5*。

- 起始信号：在 SCL（图中为 SIO_C）为高电平时，SDA(图中为 SIO_D) 出现一个下降沿，则 SCCB 开始传输。
- 停止信号：在 SCL 为高电平时，SDA 出现一个上升沿，则 SCCB 停止传输。
- 数据有效性：除了开始和停止状态，在数据传输过程中，当 SCL 为高电平时，必须保证 SDA 上的数据稳定，也就是说，SDA 上的电平变换只能发生在 SCL 为低电平的时候，SDA 的信号在 SCL 为高电平时被采集。

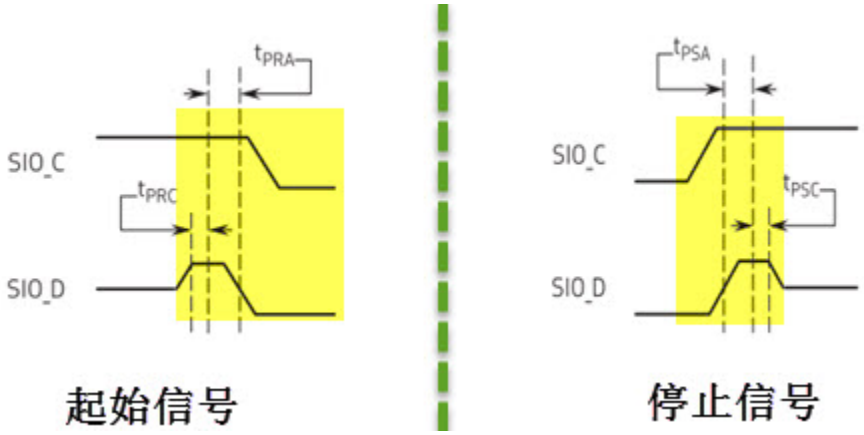


图 47-4 SCCB 停止信号

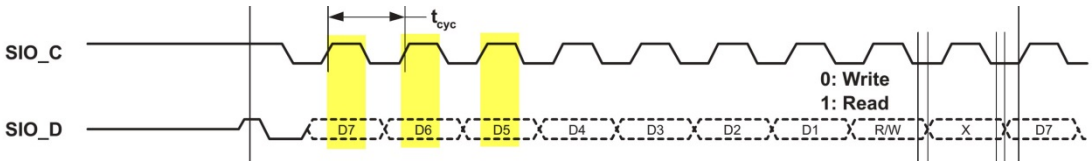


图 47-5 SCCB 的数据有效性

44.2.3.2 SCCB 数据读写过程

在 SCCB 协议中定义的读写操作与 I2C 也是一样的，只是换了一种说法。它定义了两种写操作，即三步写操作和两步写操作。三步写操作可向从设备的一个目的寄存器中写入数据，见图 *47_6*。在三步写操作中，第一阶段发送从设备的 ID 地址 +W 标志 (等于 I2C 的设备地址：7 位设备地址 + 读写方向标志)，第二阶段发送从设备目标寄存器的 8 位地址，第三阶段发送要写入寄存器的 8 位数据。图中的“X”数据位可写入 1 或 0，对通讯无影响。

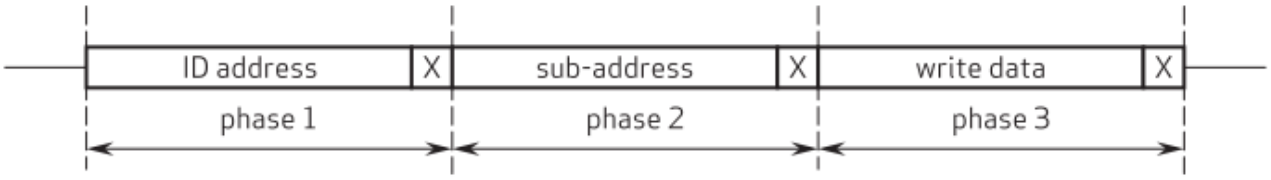


图 47-6 SCCB 的三步写操作

而两步写操作没有第三阶段，即只向从器件传输了设备 ID+W 标志和目的寄存器的地址，见图 *47_7*。两步写操作是用来配合后面的读寄存器数据操作的，它与读操作一起使用，实现 I2C 的复

合过程。

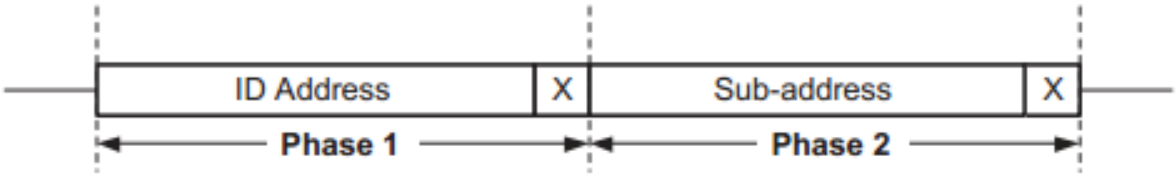


图 47-7 SCCB 的两步写操作

两步读操作，它用于读取从设备目的寄存器中的数据，见图 *47_8*。在第一阶段中发送从设备的设备 ID+R 标志 (设备地址 + 读方向标志) 和自由位，在第二阶段中读取寄存器中的 8 位数据和写 NA 位 (非应答信号)。由于两步读操作没有确定目的寄存器的地址，所以在读操作前，必需有一个两步写操作，以提供读操作中的寄存器地址。

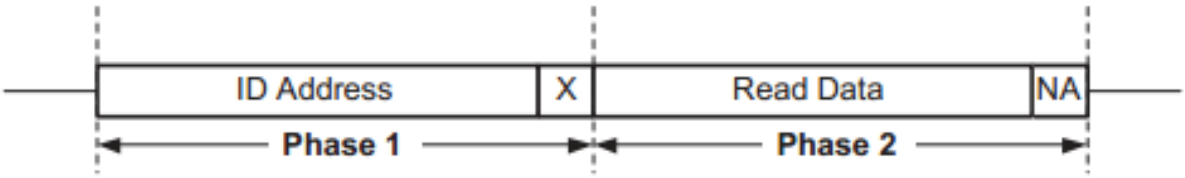


图 47-8 SCCB 的两步读操作

可以看到，以上介绍的 SCCB 特性都与 I2C 无区别，完全可以使用 STM32 的 I2C 外设来与 OV7725 进行 SCCB 通讯。

44.2.4 OV7725 的寄存器

控制 OV7725 涉及到它很多的寄存器，可直接查询《OV7725datasheet》了解，通过这些寄存器的配置，可以控制它输出图像的分辨率大小、图像格式、图像处理及图像方向等。见图 *47_9*。

Address (Hex)	Register Name	Default (Hex)	R/W	Description
10	AEC	40	RW	Exposure Value Bit[7:0]: AEC[7:0] (see register AECH for AEC[15:8])
11	CLKRC	80	RW	Internal Clock Bit[7]: Reserved Bit[6]: Use external clock directly (no clock pre-scale available) Bit[5:0]: Internal clock pre-scalar F(internal clock) = F(input clock)/(Bit[5:0]+1)/2 • Range: [0 0000] to [1 1111]
12	COM7	00	RW	Common Control 7 Bit[7]: SCCB Register Reset 0: No change 1: Resets all registers to default values Bit[6]: Resolution selection 0: VGA 1: QVGA Bit[5]: BT.656 protocol ON/OFF selection Bit[4]: Sensor RAW Bit[3:2]: RGB output format control 00: GBR4:2:2 01: RGB565 10: RGB555 11: RGB444 Bit[1:0]: Output format control 00: YUV 01: Processed Bayer RAW 10: RGB 11: Bayer RAW

图 47-9 0xFF=0 时的 DSP 相关寄存器说明 (部分)

官方还提供了一个《OV7725 Software Application Note》的文档，它针对不同的配置需求，提供了配置范例，见图 *47_10*。其中 write_SCCB 是一个利用 SCCB 向寄存器写入数据的函数，第一个参数为要写入的寄存器的地址，第二个参数为要写入的内容。

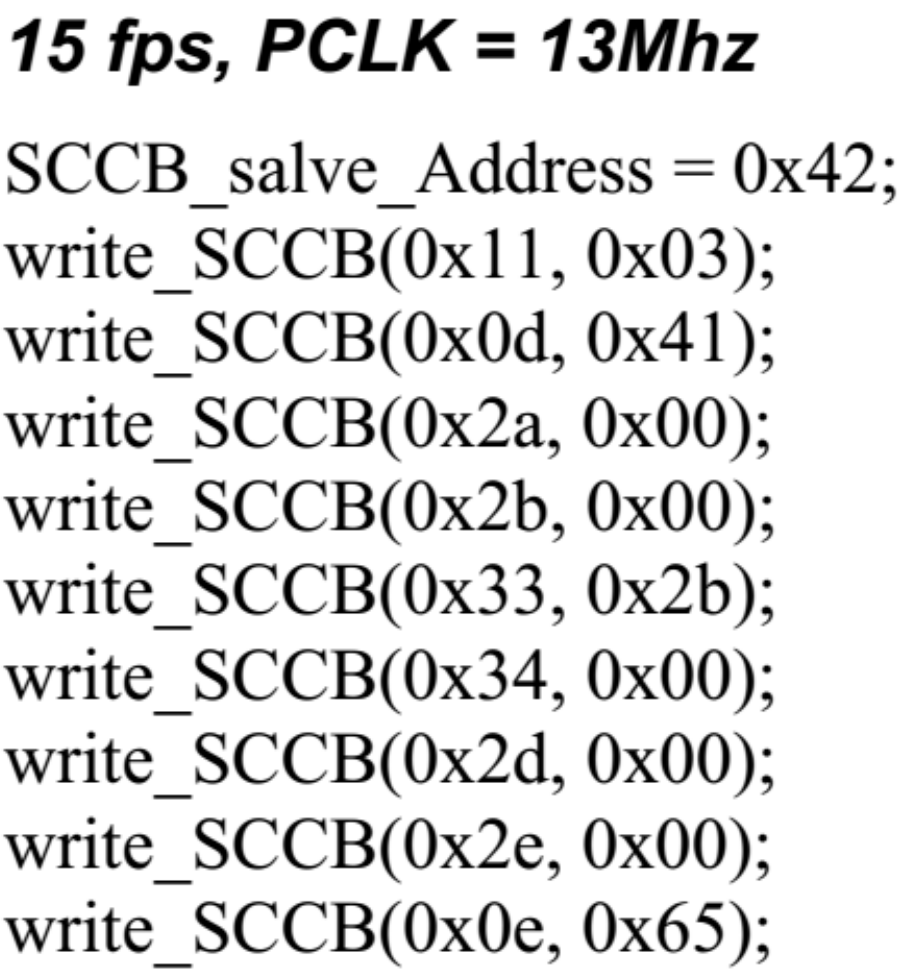


图 47-10 调节帧率的寄存器配置范例

44.2.5 像素数据输出时序

主控器控制 OV7725 时采用 SCCB 协议读写其寄存器，而它输出图像时则使用 VGA 或 QVGA 时序，其中 VGA 在输出图像分辨率为 480*640 时采用，QVGA 是 Quarter VGA，其输出分辨率为 240*320，这些时序跟控制液晶屏输出图像数据时十分类似。

OV7725 传感器输出图像时，一帧帧地输出，在帧内的数据一般从左到右，从上到下，一个像素一个像素地输出 (也可通过寄存器修改方向)，见图 *47_11*。

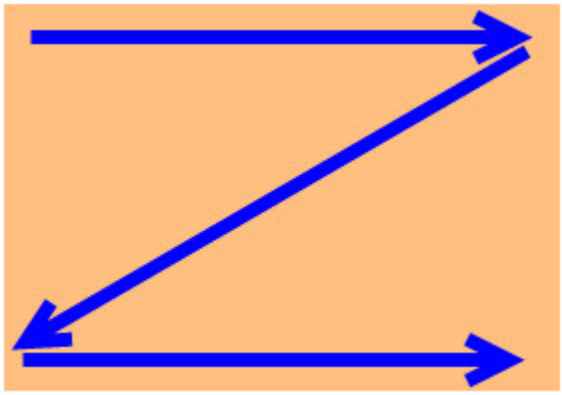


图 47-11 摄像头数据输出

例如，见图 *47_12* 和图 *47_13*，若我们使用 D2-D9 数据线，图像格式设置为 RGB565，进行数据输出时，D2-D9 数据线在 PCLK 在上升沿阶段维持稳定，并且会在 1 个像素同步时钟 PCLK 的驱动下发送 1 字节的数据信号，所以 2 个 PCLK 时钟可发送 1 个 RGB565 格式的像素数据。当 HREF 为高电平时，像素数据依次传输，每传输完一行数据时，行同步信号 HREF 会输出一个电平跳变信号间隔开当前行和下一行的数据；一帧的图像由 N 行数据组成，当 VSYNC 为低电平时，各行的像素数据依次传输，每传输完一帧图像时，VSYNC 会输出一个电平跳变信号。

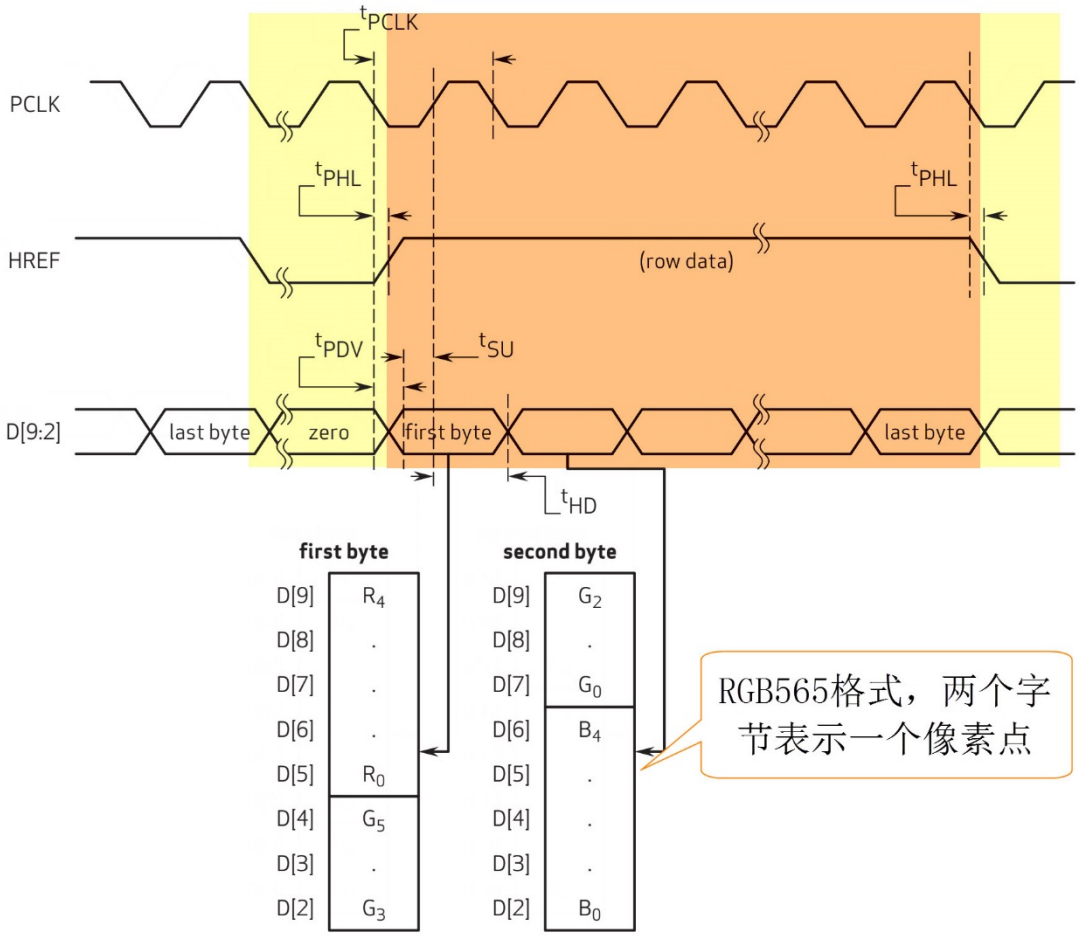


图 47-12 像素同步时序

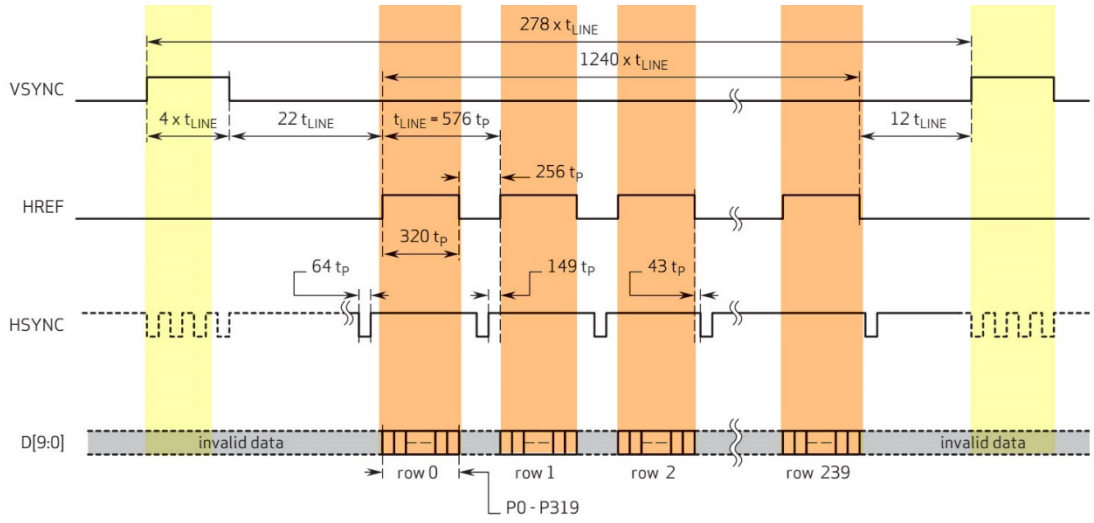


图 47-13 QVGA 帧图像同步时序

44.2.6 FIFO 读写时序

STM32F1 系列的控制器主频高、一般会扩展外部 SRAM、SDRAM 等存储器，且具有 DCMI 外设，可以直接根据 VGA 时序接收并存储摄像头输出的图像数据；而 STM32F1 系列的控制器一般主频较低、为节省成本可能不扩展 SRAM 存储器，而且不具 DCMI 外设，难以直接接收和存储 OV7725 图像传感器输出的数据。

为了解决上述问题，针对类似 STM32F1 或更低级的控制器，野火的 OV7725 摄像头在图像传感器之外还添加了一个型号为 AL422B 的 FIFO，用于缓冲数据。AL422B 的本质是一种 RAM 存储器，见图 *47_14*，它的容量大小为 393216 字节，支持同时写入和读出数据，这正是专门用于 FIFO 缓冲功能而设计的，关于它的详细说明可查阅《AL422_datasheet》文档。

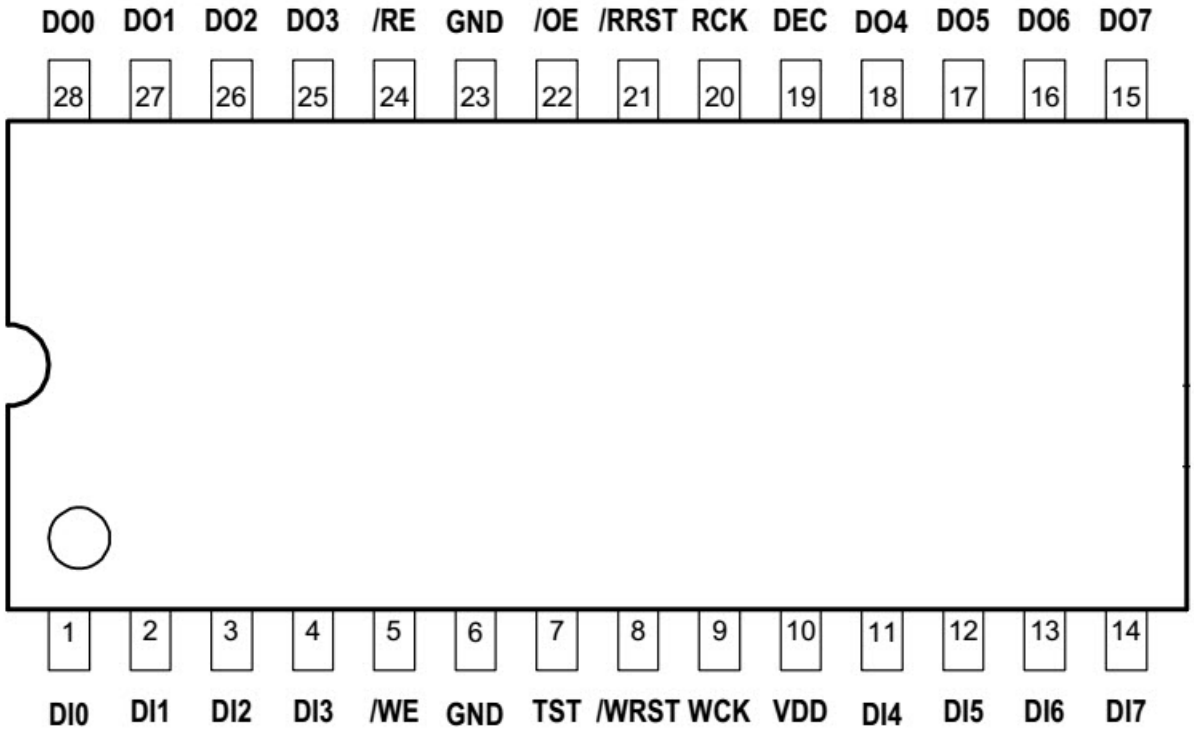


图 47-14 AL422B FIFO 引脚图

AL422B 的各引脚功能介绍见表 47-2。

表 47-2 AL422B 引脚功能说明

管脚名称	管脚类型	管脚描述
DI[0:7]	输入	数据输入引脚
WCK	输入	数据输入同步时钟
/WE	输入	写使能信号，低电平有效
/WRST	输入	写指针复位信号，低电平有效
DO[0:7]	输出	数据输出引脚
RCK	输入	数据输出同步时钟
/RE	输入	读使能信号，低电平有效
/RRST	输入	读指针复位信号，低电平有效
/OE	输入	数据输出使能，低电平有效
TST	输入	测试引脚，实际使用时设置为低电平

由于 AL422B 支持同时写入和读出数据，所以它的输入和输出的控制信号线都是互相独立的。写入和读出数据的时序类似，跟 VGA 的像素输出时序一致，读写时序介绍如下：

- 写时序

  写 FIFO（AL422B，下面统称为 FIFO）时序见图 *47_15*。

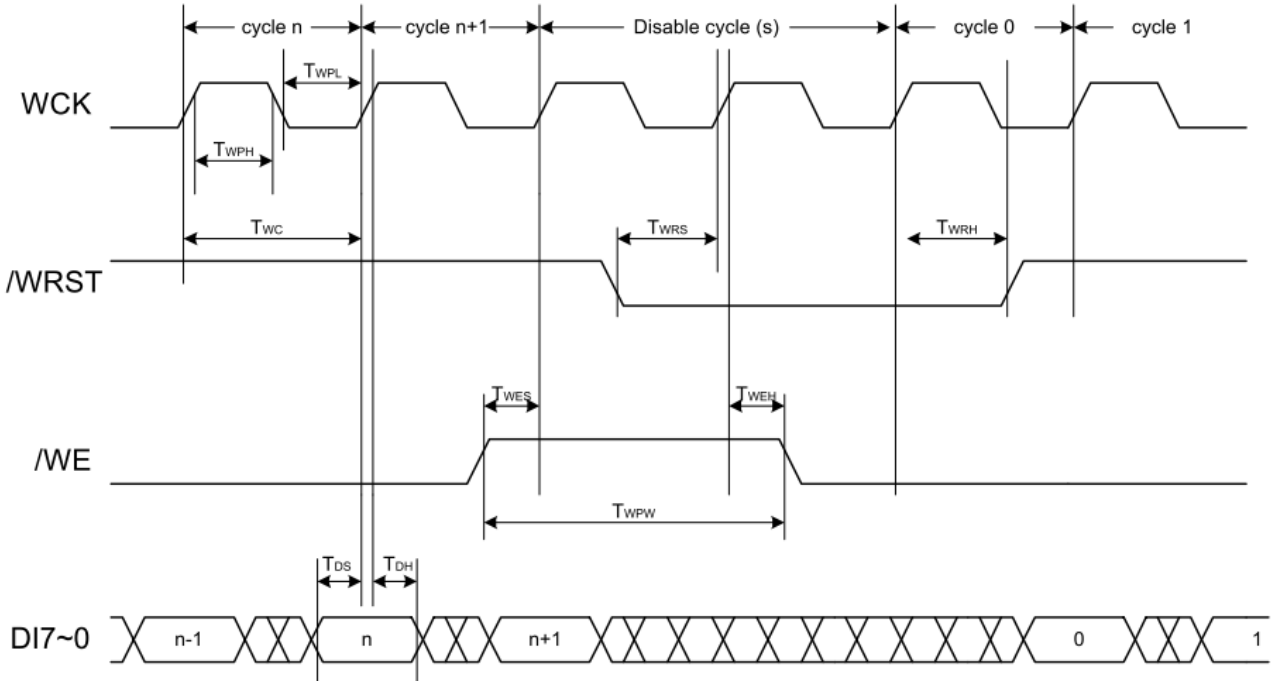


图 47-15 写 FIFO 时序

在写时序中，当 WE 管脚为低电平时，FIFO 写入处于使能状态，随着读时钟 WCK 的运转，DI[0:7] 表示的数据将会就会按地址递增的方式存入 FIFO；当 WE 管脚为高电平时，关闭输入，DI[0:7] 的数据不会被写入 FIFO。

在控制写入数据时，一般会先控制写指针作一个复位操作：把 WRST 设置为低电平，写指针会复位到 FIFO 的 0 地址，然后 FIFO 接收到的数据会从该地址开始按自增的方式写入。

- 读时序

  读 FIFO 时序见图 *47_16*。

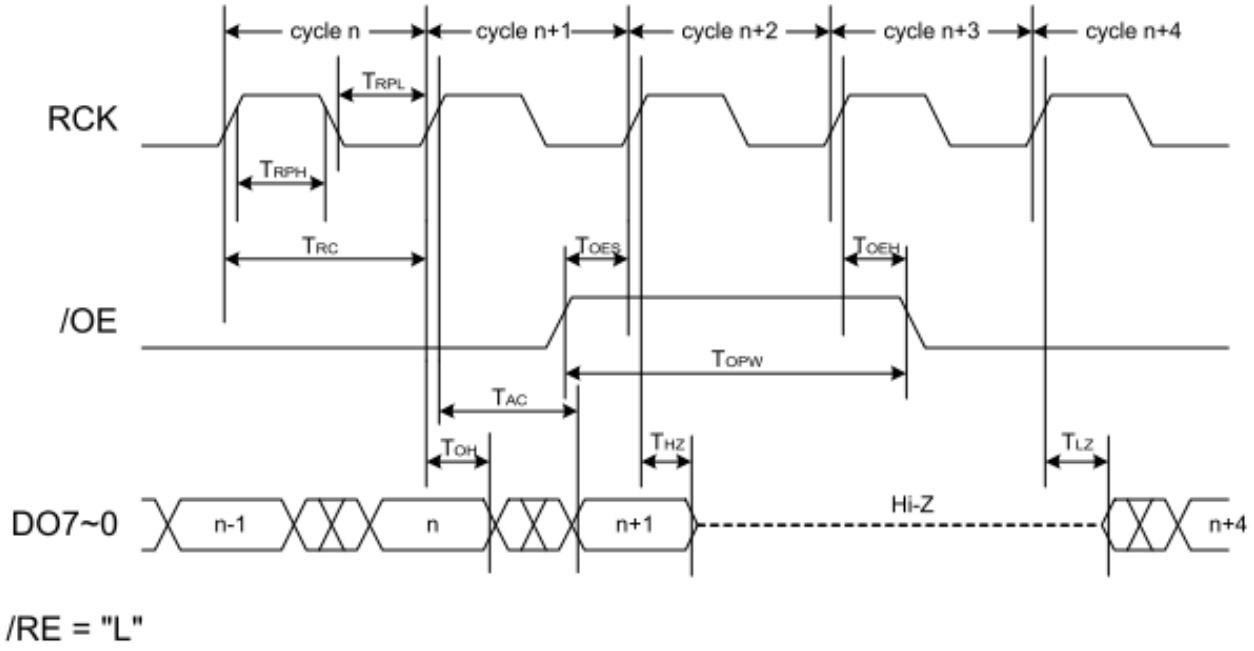


图 47-16 读 FIFO 时序

FIFO 的读时序类似，不过读使能由两个引脚共同控制，即 OE 和 RE 引脚均为低电平时，输出处于使能状态，随着读时钟 RCK 的运转，在数据输出管脚 DO[0:7] 就会按地址递增的方式输出数据。

类似地，在控制读出数据时，一般会先控制读指针作一个复位操作：把 RRST 设置为低电平，读指针会复位到 FIFO 的 0 地址，然后 FIFO 数据从该地址开始按自增的方式输出。

44.2.7 摄像头的驱动原理

野火 OV7725 摄像头中包含有 FIFO，所以外部控制器驱动摄像头时，需要协调好 FIFO 与 OV7725 传感器的关系，下面配合摄像头的原理图讲解其驱动原理。

原理图主要分为外部引出接口、OV7725 及 FIFO 部分，见图 *47_17*。

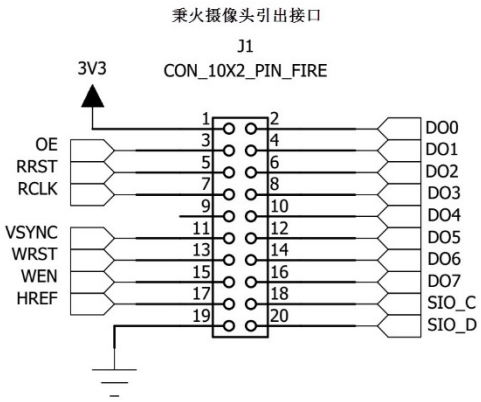


图 47-17 野火摄像头引出的排母接口

摄像头引出的接口包含了 OV7725 传感器及 FIFO 的混合引脚，外部的控制器使用这些引脚即可驱动摄像头，其说明见表 47-3。

表 47-3 摄像头引脚列表

管脚名称	管脚关系	管脚描述
OE	FIFO 的 OE 引脚	数据输出使能，低电平有效
RRST	FIFO 的 RRST 引脚	读指针复位信号，低电平有效
RCLK	FIFO 的 RCK 引脚	数据输出同步时钟
VSYNC	OV7725 的 VSYNC 引脚	场同步信号
WRST	FIFO 的 WRST 引脚	写指针复位信号，低电平有效
WEN	与下面的 HREF 共同组成与非门的输入	与 HREF 共同控制 FIFO 的 WE 引脚，WEN 与 HREF 同时为高电平时，WE 为低电平，OV7725 可以向 FIFO 写入数据
HREF	OV7725 的 HREF 引脚	行同步信号
DO[0:7]	FIFO 的 DO[0:7] 引脚	数据输出引脚
SIO_C	OV7725 的 SCL 引脚	SCCB 总线的时钟线
SIO_D	OV7725 的 SDA 引脚	SCCB 总线的数据线

从上述列表与下面的图 *47_18* 和图 *47_19*，可了解到，与 OV7725 传感器像素输出相关的 PCLK 和 D[0:7] 并没有引出，因为这些引脚被连接到了 FIFO 的输入部分，OV7725 的像素输出时序与 FIFO 的写入数据时序是一致的，所以在 OV7725 时钟 PCLK 的驱动下，它输出的数据会一个字节一个字节地被 FIFO 接收并存储起来。

其中最为特殊的是 WEN 引脚，它与 OV7725 的 HREF 连接到一个与非门的输入，与非门的输出连接到 FIFO 的 WE 引脚，因此，当 WEN 与 HREF 均为高电平时，FIFO 的 WE 为低电平，此时允许 OV7725 向 FIFO 写入数据。

外部控制器通过控制 WEN 引脚，可防止 OV7725 覆盖了还未被控制器读出的旧 FIFO 数据。另外，在 OV7725 输出时序中，只有当 HREF 为高电平时，PCLK 驱动下 D[0:7] 线表示的才是有效像素数据，因此，利用 HREF 控制 FIFO 的 WE 可以确保只有有效数据才被写入到 FIFO 中。

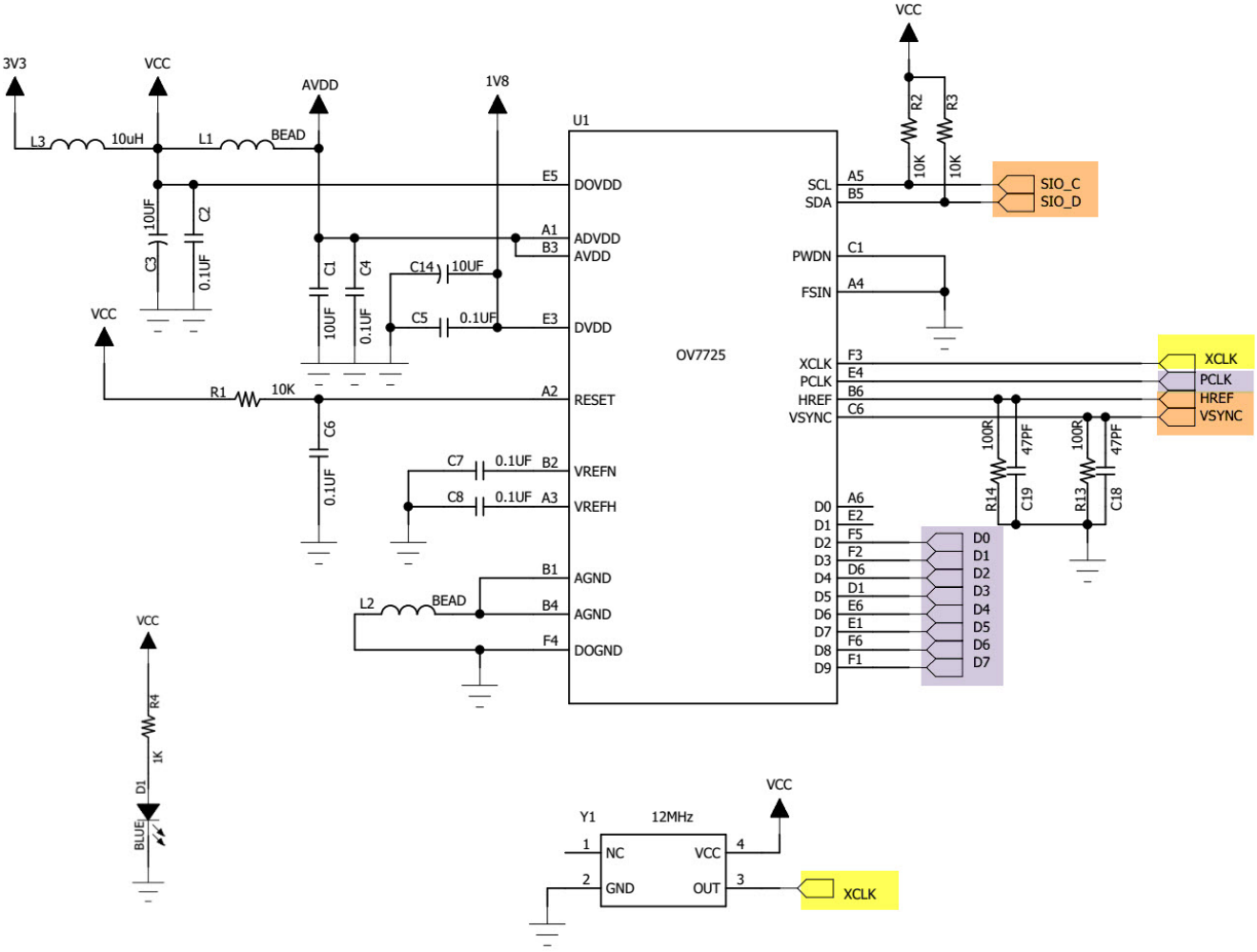


图 47-18 摄像头的 OV7725 部分硬件连接图

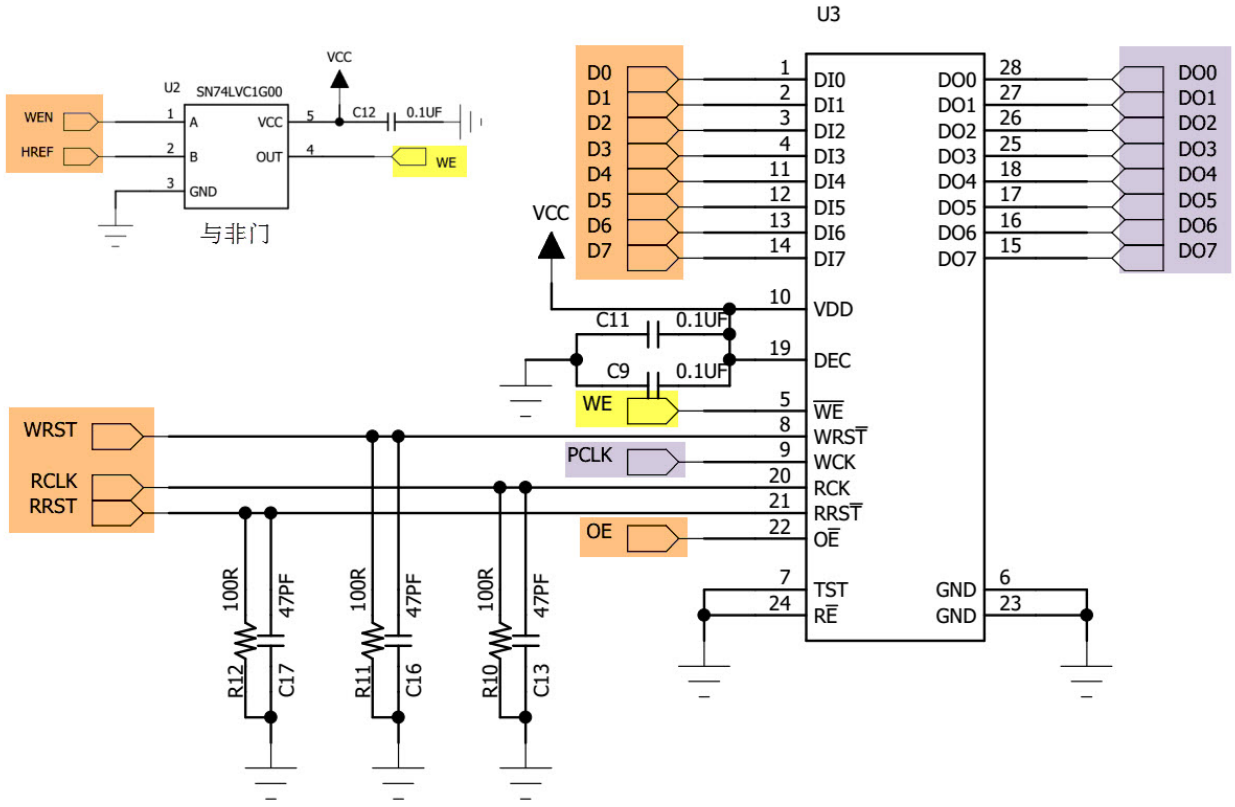


图 47-19 摄像头的 FIFO 部分硬件连接图

配合摄像头的原理图、OV7725、FIFO 的时序图，可总结出摄像头采集数据的过程如下：

(1) 利用 SIO_C、SIO_D 引脚通过 SCCB 协议向 OV7725 的寄存器写入初始化配置；

(2) 初始化完成后，OV7725 传感器会使用 VGA 时序输出图像数据，它的 VSYNC 会首先输出帧有效信号（低电平跳变），当外部的控制器（如 STM32）检测到该信号时，把 WEN 引脚设置为高电平，并且使用 WRST 引脚复位 FIFO 的写指针到 0 地址；

(3) 随着 OV7725 继续按 VGA 时序输出图像数据，它在传输每行有效数据时，HREF 引脚都会持续输出高电平，由于 WEN 和 HREF 同时为高电平输入至与非门，使得其连接到 FIFO WE 引脚的输出为低电平，允许向 FIFO 写入数据，所以在这期间，OV7725 通过它的 PCLK 和 D[0:7] 信号线把图像数据存储到 FIFO 中，由于前面复位了写指针，所以图像数据是从 FIFO 的 0 地址开始记录的；

(4) 各行图像数据持续传输至 FIFO，受 HREF 控制的 WE 引脚确保了写入到 FIFO 中的都是有

效的图像数据，OV7725 输出完一帧数据时，VSYNC 会再次输出帧有效信号，表示一帧图像已输出完成；

(5) 控制器检测到上述 VSYNC 信号后，可知 FIFO 中已存储好一帧图像数据，这时控制 WEN 引脚为低电平，使得 FIFO 禁止写入，防止 OV7725 持续输出的下一帧数据覆盖当前 FIFO 数据；

(6) 控制器使用 RRST 复位读指针到 FIFO 的 0 地址，然后通过 FIFO 的 RCLK 和 DO[0:7] 引脚，从 0 地址开始把 FIFO 缓存的整帧图像数据读取出来。在这期间，OV7725 是持续输出它采集到的图像数据的，但由于禁止写入 FIFO，这些数据被丢弃了；

(7) 控制器使用 WRST 复位写指针到 FIFO 的 0 地址，然后等待新的 VSYNC 有效信号到来，检测到后把 WEN 引脚设置为高电平，恢复 OV7725 向 FIFO 的写入权限，OV7725 输出的新一帧图像数据会被写入到 FIFO 的 0 地址中，重复上述过程。

摄像头的整个控制过程见图 *47_20*。

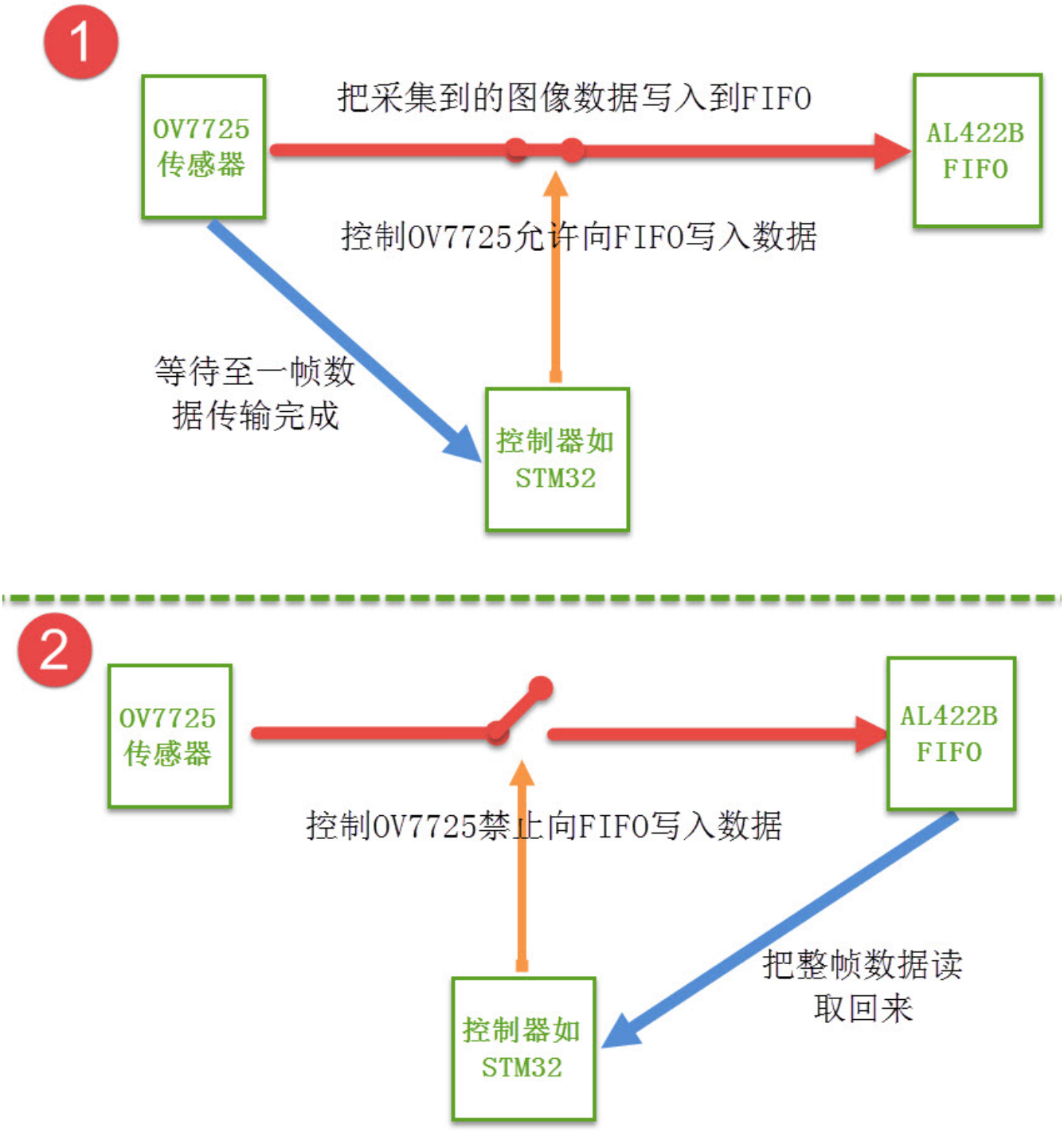


图 47-20 摄像头控制过程

在使用本摄像头时，我们一般配套开发板的液晶屏，把 OV7725 配置为 240*320 分辨率（QVGA），

RGB565 格式，那么 OV7725 输出一帧的图像大小为 240*320*2=153600 字节，而本摄像头采用的 FIFO 型号 AL422B 容量为 393216 字节，最多可以缓存 2 帧这样的图像，通过这样的方式，STM32 无需直接处理 OV7725 高速输出的数据。但是，如果配置 OV7725 为 480*640 分辨率时，其一帧图像大小为 480*640*2=614400 字节，FIFO 的容量不足以直接存储一帧这样的图像，因此，当 OV7725 往 FIFO 写数据的时候，STM32 端要同时读取数据，确保在 OV7725 覆盖旧数据的之前，STM32 端已经把这部分数据读取出来了。

44.3 摄像头驱动实验

本小节讲解如何使用如何利用 OV7725 摄像头采集 RGB565 格式的图像数据，并把这些数据实时显示到液晶屏上。

学习本小节内容时，请打开配套的“摄像头-OV7725-液晶实时显示”工程配合阅读。

44.3.1 硬件设计

关于摄像头的原理图此处不再分析。在我们的实验板上有引出一个摄像头专用的排母，可直接与摄像头引出的引脚连，接入后它与 STM32 引脚的连接关系见图 *47_21*。

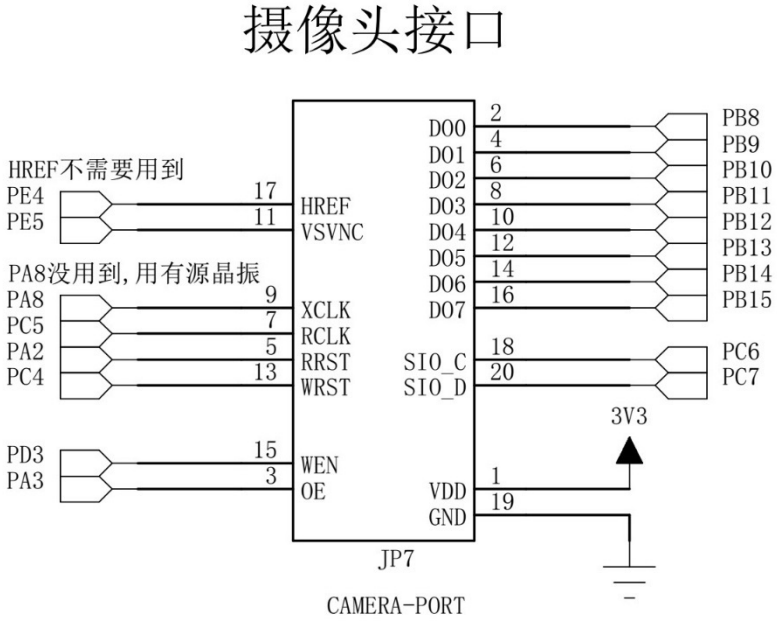


图 47-21 STM32 实验板引出的摄像头接口

摄像头与 STM32 连接关系中主要分为 SCCB 控制、VGA 时序控制、FIFO 数据读取部分，介绍如下：

(1) SCCB 控制相关

摄像头中的 SIO_C 和 SIO_D 引脚直接连接到 STM32 普通的 GPIO，它们不具有硬件 I2C 的功能，所以在后面的代码中采用模拟 I2C 时序，实际上直接使用硬件 I2C 是完全可以实现 SCCB 协议的，本设计采用模拟 I2C 是芯片资源分配妥协的结果。

(2) VGA 时序相关

检测 VGA 时序的 HREF、VSYNC 引脚，它们与 STM32 连接的 GPIO 均设置为输入模式，其中 HREF 在本实验中并没有使用，它已经通过摄像头内部的与非门控制了 FIFO 的写使能；VSYNC 与 STM32 连接的 GPIO 引脚会在程序中配置成中断模式，STM32 利用该中断信号获知新的图像是否采集完成，从而控制 FIFO 是否写使能。

(3) FIFO 相关

与 FIFO 控制相关的 RCLK、RRST、WRST、WEN 及 OE 与 STM32 连接的引脚均直接配置成推挽输出，STM32 根据图像的采集情况利用这些引脚控制 FIFO；读取 FIFO 数据内容使用的数据引脚 DO[0:7] 均连接到 STM32 同一个 GPIO 端口连续的高 8 位引脚 PB[8:15]，这些引脚使用时均配置成输入，程序设计中直接读取 GPIO 端口的高 8 位状态直接获取一个字节的 FIFO 内容，建议在连接这部分数据信号时，参考本设计采用同一个 GPIO 端口连续的 8 位（高 8 位或低 8 位均可），否则会导致读取数据的程序非常复杂。

(4) XCLK 信号

本设计中 STM32 的摄像头接口还预留了 PA8 引脚用于与摄像头的 XCLK 连接，STM32 的 PA8 可以对外输出时钟信号，所以在使用不带晶振的摄像头时，可以通过该引脚给摄像头提供时钟，野火摄像头内部已自带晶振，在程序中没有使用 PA8 引脚。

以上原理图可查阅《指南者开发板—原理图》文档获知，若您使用的摄像头或实验板不一样，请根据实际连接的引脚修改程序。

44.3.2 软件设计

本实验的工程名称为“液晶实时显示”，学习时请打开该工程配合阅读。为了方便展示及移植，我们把模拟 SCCB 时序相关的代码写到 bsp_sccb.c 及 bsp_sccb.h 文件中，而摄像头模式控制相关的代码都编写到“bsp_ov7725.c”、“bsp_ov7725.h”文件中，这些文件是我们自己编写的，不属于 HAL 库的内容，可根据您的喜好命名文件。

44.3.2.1 编程要点

(1) 初始化 SCCB 通讯使用的目标引脚及端口时钟；

(2) 初始化 OV7725 的 VGA 和 FIFO 控制相关的引脚和时钟；

(3) 使用 SCCB 协议向 OV7725 写入初始化配置；

(4) 配置筛选器的工作方式；

(5) 编写测试程序，收发报文并校验。

44.3.2.2 代码分析

44.3.2.2.1 摄像头硬件相关宏定义

我们把摄像头控制硬件相关的配置以宏的形式定义到“bsp_ov7725.h”及“bsp_sccb.h”文件中，其中包括 VGA 部分接口、FIFO 控制及 SCCB(即模拟 I2C) 相关的引脚，见*代码清单 47_1*。

列表 1: 代码清单 47-1 摄像头硬件配置相关的宏 (bsp_ov7725.h 文件)

```
// FIFO 输出使能，即模块中的 OE
#define      OV7725_OE_GPIO_PORT          GPIOA
#define      OV7725_OE_GPIO_PIN           GPIO_PIN_3

// FIFO 写复位
```

(下页继续)

(续上页)

```
#define      OV7725_WRST_GPIO_PORT        GPIOC
#define      OV7725_WRST_GPIO_PIN         GPIO_PIN_4

// FIFO 读复位
#define      OV7725_RRST_GPIO_PORT        GPIOA
#define      OV7725_RRST_GPIO_PIN         GPIO_PIN_2

// FIFO 读时钟
#define      OV7725_RCLK_GPIO_PORT        GPIOC
#define      OV7725_RCLK_GPIO_PIN         GPIO_PIN_5

// FIFO 写使能
#define      OV7725_WE_GPIO_PORT          GPIOD
#define      OV7725_WE_GPIO_PIN           GPIO_PIN_3

// 8 位数据口
#define      OV7725_DATA_GPIO_PORT        GPIOB
#define      OV7725_DATA_0_GPIO_PIN       GPIO_PIN_8
#define      OV7725_DATA_1_GPIO_PIN       GPIO_PIN_9
#define      OV7725_DATA_2_GPIO_PIN       GPIO_PIN_10
#define      OV7725_DATA_3_GPIO_PIN       GPIO_PIN_11
#define      OV7725_DATA_4_GPIO_PIN       GPIO_PIN_12
#define      OV7725_DATA_5_GPIO_PIN       GPIO_PIN_13
#define      OV7725_DATA_6_GPIO_PIN       GPIO_PIN_14
#define      OV7725_DATA_7_GPIO_PIN       GPIO_PIN_15

// OV7725 场中断
#define      OV7725_VSYNC_GPIO_PORT       GPIOC
#define      OV7725_VSYNC_GPIO_PIN        GPIO_PIN_3

#define      OV7725_VSYNC_EXTI_IRQ              EXTI3_IRQn
#define      OV7725_VSYNC_EXTI_INT_FUNCTION     EXTI3_IRQHandler
```

列表 2: 代码清单 47-2 SCCB 接口相关的宏定义

```
/* 直接操作寄存器的方法控制 IO */
#define digitalHi(p,i)       {p->BSRR=i;}  //设置为高电平
#define digitalLo(p,i)       {p->BSRR=(uint32_t)i << 16;}
                 //输出低电平

#define SCL_H          digitalHi(GPIOC , GPIO_PIN_6)
#define SCL_L          digitalLo(GPIOC , GPIO_PIN_6)

#define SDA_H          digitalHi(GPIOC , GPIO_PIN_7)
#define SDA_L          digitalLo(GPIOC , GPIO_PIN_7)

#define SCL_read       HAL_GPIO_ReadPin(GPIOC , GPIO_PIN_6)
#define SDA_read       HAL_GPIO_ReadPin(GPIOC , GPIO_PIN_7)

#define ADDR_OV7725    0x42
```

以上代码根据硬件的连接，使用宏封装了各种控制信号，包括控制输出电平或读取输入时使用的库函数操作。若使用 STM32 与摄像头的引脚连接与我们的设计的不同，修改这两个文件的引脚连接关系即可。

44.3.2.2.2 SCCB 总线的软件实现

本设计中使用普通 GPIO 来模拟 SCCB 时序，需要根据 SCCB 时序，编写读、写字节的模拟函数，在后面的 OV7725_Init 会利用这些函数向 OV7725 相应的寄存器写入配置参数，初始化摄像头。本小节介绍的与 SCCB 时序相关函数都位于 bsp_sccb.c 文件中，这些函数跟模拟 I2C 的基本一致。

(1) 初始化 SCCB 用到的 GPIO

在本实验中，使用 SCCB_GPIO_Config 函数初始化 SCCB 使用的两个通讯引脚，把它们初始化成普通开漏输出模式，其代码见*代码清单 47_3*。

列表 3: 代码清单 47-3：SCCB_GPIO_Config 函数

```
void SCCB_GPIO_Config(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    /* SCL(PC6)、SDA(PC7) 管脚配置 */

    __HAL_RCC_GPIOC_CLK_ENABLE();
    GPIO_InitStructure.Pin =   GPIO_PIN_7;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_MEDIUM;//
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_OD;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.Pin =  GPIO_PIN_6;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

(2) SCCB 起始与结束时序

SCCB 通讯需要有起始与结束信号，这分别由 SCCB_Start 和 SCCB_Stop 函数实现。

SCCB_Start 代码见*代码清单 47_4*。

列表 4: 代码清单 47-4：SCCB_Start 函数

```
/*****************************************************************
* 函数名: SCCB_Start
* 描述  : SCCB 起始信号
* 输入  : 无
* 输出  : 无
* 注意  : 内部调用
*****************************************************************/
static int SCCB_Start(void)
{
```

(下页继续)

(续上页)

```
    SDA_H;
    SCL_H;
    SCCB_delay();
    if (!SDA_read)
        return DISABLE; /* SDA 线为低电平则总线忙，退出 */
    SDA_L;
    SCCB_delay();
    if (SDA_read)
        return DISABLE; /* SDA 线为高电平则总线出错，退出 */
    SDA_L;
    SCCB_delay();
    return ENABLE;
}
```

参照前面介绍的 SCCB 时序，当 SCL 线为高电平时，SDA 线出现下降沿，表示 SCCB 时序的起始信号，SCCB_Start 函数就是实现了这样功能，其中的 SDA_H 和 SCL_H 是用于控制 SDA 和 SCL 引脚电平的宏。为了提高程序的健壮性，还使用 SDA_read 宏检测 SDA 线是否忙碌或是否正常。

SCCB 结束信号的函数实现类似，其 SCCB_Stop 代码见*代码清单 47_5*。

列表 5: 代码清单 47-5：SCCB_Stop 函数

```
/*****************************************************************
* 函数名：SCCB_Stop
* 描述  ：SCCB 停止信号
* 输入  ：无
* 输出  ：无
* 注意  ：内部调用
*****************************************************************/
static void SCCB_Stop(void)
{
    SCL_L;
    SCCB_delay();
```

(下页继续)

(续上页)

```
    SDA_L;
    SCCB_delay();
    SCL_H;
    SCCB_delay();
    SDA_H;
    SCCB_delay();
}
```

参照前面 SCCB 时序的介绍，当 SCL 线为高电平的时候，使 SDA 线出现一个上升沿，表示 SCCB 时序的结束信号。

(3) 写寄器与读寄存器

与 I2C 时序类似，在 SCCB 时序也使用自由位 (Don’t care bit) 和非应答 (NA) 信号来保证正常通讯。自由位和非应答信号位于 SCCB 每个传输阶段中的第九位。

在写数据的第一个传输阶段中，第 9 位为自由位，在一般的正常通讯中，第 9 位时，主机的 SDA 线输出高电平，而从机把 SDA 线拉低作为响应，第二、三阶段类似，只是传输的内容分别为目的寄存器地址和要写入的数据。见图 *47_22*。

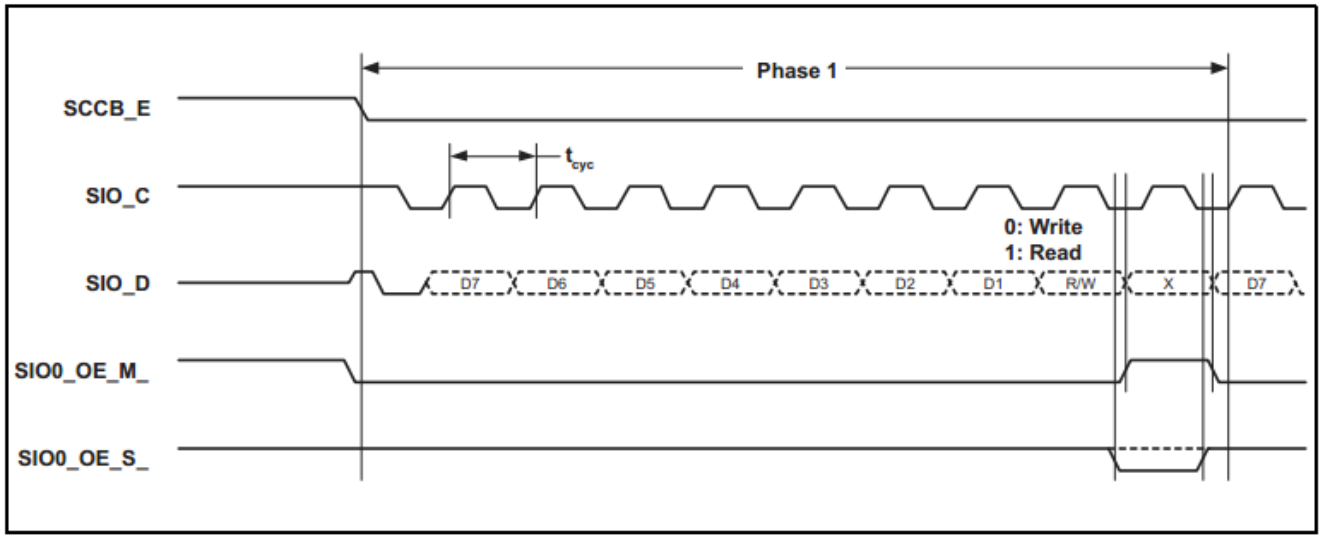


图 47-22 写操作第一阶段 (传输器件地址)

因此，在数据传输的第 9 位，主机使用 SCCB_WaitAck 函数来等待从机的应答，代码见代码清单 47_6。

列表 6: 代码清单 47-6：SCCB_WaitAck 函数

```
/*****************************************************************
* 函数名：SCCB_WaitAck
* 描述  ：SCCB 等待应答
* 输入  ：无
* 输出  ：返回为:=1 有 ACK,=0 无 ACK
* 注意  ：内部调用
*****************************************************************/
static int SCCB_WaitAck(void)
{
    SCL_L;
    SCCB_delay();
    SDA_H;
    SCCB_delay();
    SCL_H;
    SCCB_delay();
    if (SDA_read) {
        SCL_L;
        return DISABLE;
    }
    SCL_L;
    return ENABLE;
}
```

该函数让主机把 SDA 线设为高电平，延时一段时间后再检测 SDA 线的电平，若为低则返回 ENABLE 表示接收到从机的应答，反之返回 DISABLE。

最后，整个三相写过程由函数 SCCB_WriteByte 实现，它的具体代码见代码清单 47_7。

列表 7: 代码清单 47-7：SCCB_WriteByte 函数

```
/* 以下宏位于 bsp_ov7725.h 文件 */
#define ADDR_OV7725   0x42
/* 以下宏位于 bsp_sccb.h 文件 */
#define DEV_ADR  ADDR_OV7725

/*****************************************************************************
* 函数名: SCCB_WriteByte
* 描述  : 写一字节数据
* 输入: -WriteAddress: 待写入地址- SendByte: 待写入数据 - DeviceAddress: 器件类型
* 输出  : 返回为:=1 成功写入,=0 失败
* 注意  : 无
*****************************************************************/
int SCCB_WriteByte( uint16_t WriteAddress , uint8_t SendByte )
{
    if (!SCCB_Start()) {
        return DISABLE;
    }
    SCCB_SendByte( DEV_ADR );                    /* 器件地址 */
    if ( !SCCB_WaitAck() ) {
        SCCB_Stop();
        return DISABLE;
    }
    SCCB_SendByte((uint8_t)(WriteAddress & 0x00FF));   /* 设置低起始地址 */
    SCCB_WaitAck();
    SCCB_SendByte(SendByte);
    SCCB_WaitAck();
    SCCB_Stop();
    return ENABLE;
}
```

SCCB_WriteByte 函数调用 SCCB_Start 产生起始信号，接着调用 SCCB_SendByte 把器件地址

DEV_ADR(这是一个宏，数值是 0x42) 一位一位地发送出去，在第 9 位时，调用 SCCB_WaitAck 函数检测从机的应答，若接收到应答则进入第二阶段——发送目的寄存器地址，再进入第三阶段——发送要写入的内容。在第二、三阶段没有加条件判断语句判断是否接收到从机的应答，这是因为 SCCB 规定在数据传输阶段允许从机不应答 (实际上，OV7725 芯片在这两个阶段都会有应答讯号。在最后，三阶段都传输结束时，要调用 SCCB_Stop 函数结束本次 SCCB 传输。

与自由位相对应的非应答信号用在两相读操作的第二阶段的第 9 位，见图 *47_23*。在这第 9 位中，从机把 SDA 线置为高电平，而主机把 SDA 线拉低表示非应答，接着本次读数据的操作就结束了。

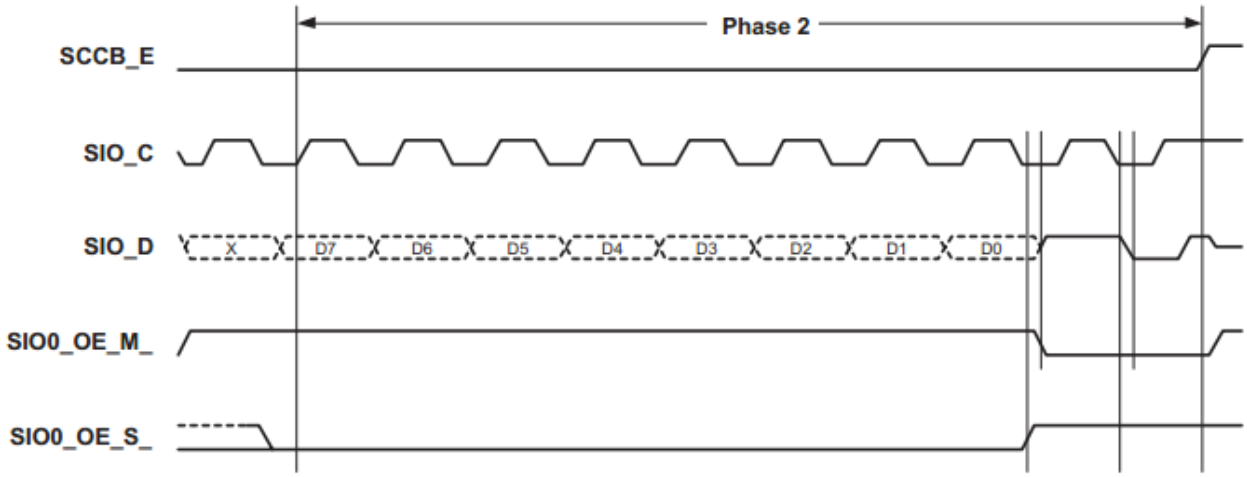


图 47-23 两相读操作第二阶段 (读寄存器内容)

主机的非应答信号，由 SCCB_NoAck 函数实现，其代码见*代码清单 47_8*。

列表 8: 代码清单 47-8：SCCB_NoAck 函数

```
/*****************************************************************
* 函数名：SCCB_NoAck
* 描述   ：SCCB 无应答方式
* 输入   ：无
* 输出   ：无
* 注意   ：内部调用
*****************************************************************/
static void SCCB_NoAck(void)
```

(下页继续)

(续上页)

```
{
    SCL_L;
    SCCB_delay();
    SDA_H;
    SCCB_delay();
    SCL_H;
    SCCB_delay();
    SCL_L;
    SCCB_delay();
}
```

最后，整个读寄存器的过程由 SCCB_ReadByte 函数完成，其代码见*代码清单 47_9*。

列表 9: 代码清单 47-9：SCCB_ReadByte 函数

```
/*****************************************************************************
* 函数名: SCCB_ReadByte
* 描述   : 读取一串数据
* 输入: - pBuffer: 存放读出数据- length: 待读出长度- ReadAddress: 待读出地址 -
DeviceAddress: 器件类型
* 输出   : 返回为:=1 成功读入,=0 失败
* 注意   : 无
*****************************************************************************/
int SCCB_ReadByte(uint8_t* pBuffer, uint16_t length, uint8_t ReadAddress)
{
    if (!SCCB_Start()) {
        return DISABLE;
    }
    SCCB_SendByte( DEV_ADR );          /* 器件地址 */
    if ( !SCCB_WaitAck() ) {
        SCCB_Stop();
        return DISABLE;
```

(下页继续)

(续上页)

```
    }
    SCCB_SendByte( ReadAddress );      /* 设置低起始地址 */
    SCCB_WaitAck();
    SCCB_Stop();

    if (!SCCB_Start()) {
        return DISABLE;
    }
    SCCB_SendByte( DEV_ADR + 1 );      /* 器件地址 */
    if (!SCCB_WaitAck()) {
        SCCB_Stop();
        return DISABLE;
    }
    while (length) {
        *pBuffer = SCCB_ReceiveByte();
        if (length == 1) {
            SCCB_NoAck();
        } else {
            SCCB_Ack();
        }
        pBuffer++;
        length--;
    }
    SCCB_Stop();
    return ENABLE;
}
```

本函数在两相读操作前，加入了一个两相写操作，用于向从机发送要读取的寄存器地址。在两相写操作的第一个阶段 (第 26 行)，使用 SCCB_SendByte 函数发送的数据是 DEV_ADR+1 (即 0x43)，这与写操作中发送的 DEV_ADR 有区别，这是因为在第一阶段发送的这个器件地址的最低位是用于表示数据传送方向的，最低位为 0 时表示主机写数据，最低位为 1 时表示主机读数据，所以 DEV_ADR+1 就表示读数据了。读操作的第二阶段使用 SCCB_ReceiveByte 函数，一位一位地接

收数据，然后存放到 PBuffer 指向的单元中的。接收完 8 位数据后，主机调用 SCCB_NoAck 发送非应答位，最后调用 SCCB_Stop 结束本次读操作。

44.3.2.2.3 初始化 OV7725

在上一小节编写了模拟 SCCB 的读写寄存器的时序后，就可以向 OV7725 的寄存器发送配置参数，对 OV7725 进行初始化了。该过程由 bsp_ov7725.c 文件中的 OV7725_Init 函数完成，代码见*代码清单 47_10*。

列表 10: 代码清单 47_10

```
/*****************************************************
* 函数名: Sensor_Init
* 描述   : Sensor 初始化
* 输入   : 无
* 输出   : 返回 1 成功，返回 0 失败
* 注意   : 无
*****************************************************/
ErrorStatus OV7725_Init(void)
{
    uint16_t i = 0;
    uint8_t Sensor_IDCode = 0;

    //开始配置 ov7725
    if ( 0 == SCCB_WriteByte ( 0x12, 0x80 ) ) { /* 复位 ov7725  */
        //sccb 写数据错误
        return ERROR ;
    }
        /* 读取 ov7725  ID 号 */
    if ( 0 == SCCB_ReadByte( &Sensor_IDCode, 1, 0x0b ) ) {
        //读取 ov7725  ID 失败
        return ERROR;
    }
```

(下页继续)

(续上页)

```
    if (Sensor_IDCode == OV7725_ID) {
        for ( i = 0 ; i < OV7725_REG_NUM ; i++ ) {
            if ( 0 == SCCB_WriteByte(Sensor_Config[i].Address,Sensor_
↪Config[i].Value) )
            {   //DEBUG("write reg faild", Sensor_Config[i].Address);
                return ERROR;
            }
        }
    } else {
        return ERROR;
    }
    //ov7725 寄存器配置成功
    return SUCCESS;
}
```

这个函数的执行流程如下：

(1) 调用 SCCB_WriteByte 向地址为 0x12 的寄存器写入数据 0x80，进行复位操作。根据 OV7725 的数据手册，把该寄存器的位 7 置 1，可控制它对寄存器进行复位。

(2) 调用 SCCB_ReadByte 函数从地址为 0x0b 的寄存器读取出 OV7725 芯片的 ID 号，并与默认值进行对比，这个操作可以用来确保 OV7725 是否正常工作。

(3) 利用 for 语句循环调用 SCCB_WriteByte 函数，向各个寄存器写入配置参数，其中 SCCB_WriteByte 的输入参数为 Sensor_Config[i].Address 和 Sensor_Config[i].Value，这是自定义的结构体数组，分别对应于要配置的寄存器地址和寄存器配置参数。Sensor_Config 数组是在 bsp_ov7725.c 文件定义的，文件中首先定义了 Reg_Info 结构体类型，它包含地址和寄存器两个结构体成员，见*代码清单 47_11*。

列表 11: 代码清单 47-11：Register_Info 结构体

```
typedef struct Reg {
    uint8_t Address;   /* 寄存器地址 */
    uint8_t Value;     /* 寄存器值 */
} Reg_Info;
```

再利用这个自定义的结构体，定义结构体数组，每组的内容就表示了寄存器地址及相应的配置参数。若要修改对 OV7725 的配置，可参考 OV7725 数据手册的说明，修改相应地址的内容即可，SCCB_WriteByte 函数会在 for 循环中把这些参数写入 OV7725 芯片，下面是结构体数组的部分代码，它省略了部分寄存器，完整部分请参考源代码，见*代码清单 47_12*。

列表 12: 代码清单 47-12：部分寄存器控制参数

```
/* (bsp_ov7725.h 文件) 寄存器地址宏定义 */
#define REG_GAIN       0x00
#define REG_BLUE       0x01
#define REG_RED        0x02
#define REG_GREEN      0x03
#define REG_BAVG       0x05
#define REG_GAVG       0x06
#define REG_RAVG       0x07
#define REG_AECH       0x08
#define REG_COM2       0x09
/*... 以下省略大部分寄存器地址 */

/* (bsp_ov7725.c 文件) 寄存器参数配置，左侧为地址，右侧为要写入的值 */
Reg_Info Sensor_Config[] = {
    {REG_CLKRC,      0x00}, /* 时钟配置 */
    {REG_COM7,       0x46}, /*QVGA RGB565 */
    {REG_HSTART,     0x3f}, /* 水平图像开始 */
    {REG_HSIZE,      0x50}, /* 水平图像宽度 */
```

(下页继续)

(续上页)

```
    {REG_VSTRT,      0x03}, /* 垂直开始 */
    {REG_VSIZE,      0x78}, /* 垂直高度 */
    {REG_HREF,       0x00}, /* 杂项 */
    {REG_HOutSize,   0x50}, /* 水平输出宽度 */
    {REG_VOutSize,   0x78}, /* 垂直输出高度 */
    {REG_EXHCH,      0x00}, /* 杂项 */

    /*... 以下省略大部分内容.*/
};
```

44.3.2.2.4 采集并显示图像

OV7725 初始始化完成后，该芯片开始正常工作，由于 OV7725 采集得的图像保存到 FIFO，我们使用 STM32 只需要检测摄像头模块的 VSYNC 输出的帧结束信号，然后从 FIFO 中读取图像数据即可。

(1) 初始化 VSYNC 引脚

由于使用中断的方式来检测 VSYNC 的信号，所以要把相应的引脚初始化并为它配置 EXTI 中断，本实验使用 VSYNC_GPIO_Config 函数完成该工作，见*代码清单 47_13*。。

列表 13：代码清单 47-13：VSYNC_GPIO_Config 函数（bsp_ov7725.c 文件）

```
static void VSYNC_GPIO_Config(void)
{

    GPIO_InitTypeDef GPIO_InitStructure;
    /* 初始化时钟，注意中断要开 AFIO*/
    __HAL_RCC_AFIO_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    /* 初始化引脚 */
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin =  OV7725_VSYNC_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_IT_RISING;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(OV7725_VSYNC_GPIO_PORT, &GPIO_InitStructure);

    HAL_NVIC_SetPriority(OV7725_VSYNC_EXTI_IRQ, 0, 3);
    HAL_NVIC_EnableIRQ(OV7725_VSYNC_EXTI_IRQ);
}
```

代码中把 VSYNC 引脚配置为浮空模式，并使用下降沿中断（配置成上升沿中断也是可以的），正好对应 VGA 时序中 VSYNC 输出信号时电平跳变产生的下降沿。

(2) 编写检测 VSYNC 的中断服务函数

由于 VSYNC 出现两次下降沿，才表示 FIFO 保存了一幅图像，所以在检测 VSYNC 下降沿的中断服务函数中，使用一个变量 Ov7725_vsync 作为标志。Ov7725_vsync 标志的初始值为 0，当检测到第一次上升沿时，控制 FIFO 的相应 GPIO 引脚，允许 OV7725 向 FIFO 写入图像数据，并把标志值设置为 1；检测到第二次上升沿时，禁止 OV7725 写 FIFO，把标志设置为 2，而我们将会在 main 函数的循环中对该标志进行判断，当 Ov7725_vsync=2 时，STM32 开始从 FIFO 读取数据并显示，读取完毕后把 Ov7725_vsync 标置复位为 0，重新开始下一幅图像的采集。

中断服务函数位于 stm32f1xx_it.c 文件，见*代码清单 47_14*。

列表 14: 代码清单 47-14：VSYNC 的中断服务函数

```
/* ov7725 场中断 服务程序 */
void OV7725_VSYNC_EXTI_INT_FUNCTION ( void )
{
    if ( __HAL_GPIO_EXTI_GET_IT(OV7725_VSYNC_GPIO_PIN) != RESET ) {
        if ( Ov7725_vsync == 0 ) {
            FIFO_WRST_L();//拉低使 FIFO 写 (数据 from 摄像头) 指针复位
            FIFO_WE_H();  //拉高使 FIFO 写允许
```

(下页继续)

(续上页)

```
            Ov7725_vsync = 1;
            FIFO_WE_H(); //使 FIFO 写允许
            FIFO_WRST_H();//允许使 FIFO 写 (数据 from 摄像头) 指针运动
        } else if ( Ov7725_vsync == 1 ) {
            FIFO_WE_L(); //拉低使 FIFO 写暂停
            Ov7725_vsync = 2;
        }
        __HAL_GPIO_EXTI_CLEAR_IT(OV7725_VSYNC_GPIO_PIN);
    }
}
```

(3) 读 FIFO 并显示图像

采集得的图像数据都保存到摄像头模块的 FIFO 中，在 Ov7725_vsync 标志变为 2 的时候，STM32 即可读取它并显示到 LCD 上。与 FIFO 相关的函数有 FIFO_GPIO_Config、FIFO_PREPARE 和 ImagDisp 。

- FIFO_GPIO_Config 类似 SCCB_GPIO_Config 函数，完成了基本的 GPIO 初始化，见*代码清单 47_15*。

列表 15: 代码清单 47-15：FIFO_GPIO_Config 函数

```
static void FIFO_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启时钟 */

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
```

(下页继续)

(续上页)

```
/*(FIFO_OE--FIFO 输出使能)*/
GPIO_InitStructure.Mode  = GPIO_MODE_OUTPUT_PP;
GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
GPIO_InitStructure.Pin = OV7725_OE_GPIO_PIN;
HAL_GPIO_Init(OV7725_OE_GPIO_PORT, &GPIO_InitStructure);

/*(FIFO_WRST--FIFO 写复位)*/
GPIO_InitStructure.Pin = OV7725_WRST_GPIO_PIN;
HAL_GPIO_Init(OV7725_WRST_GPIO_PORT, &GPIO_InitStructure);

/*(FIFO_RRST--FIFO 读复位) */
GPIO_InitStructure.Pin = OV7725_RRST_GPIO_PIN;
HAL_GPIO_Init(OV7725_RRST_GPIO_PORT, &GPIO_InitStructure);

/*(FIFO_RCLK-FIFO 读时钟)*/
GPIO_InitStructure.Pin = OV7725_RCLK_GPIO_PIN;
HAL_GPIO_Init(OV7725_RCLK_GPIO_PORT, &GPIO_InitStructure);

/*(FIFO_WE--FIFO 写使能)*/
GPIO_InitStructure.Pin = OV7725_WE_GPIO_PIN;
HAL_GPIO_Init(OV7725_WE_GPIO_PORT, &GPIO_InitStructure);

/*(FIFO_DATA--FIFO 输出数据)*/
GPIO_InitStructure.Pin =  OV7725_DATA_0_GPIO_PIN |
                          OV7725_DATA_1_GPIO_PIN |
                              OV7725_DATA_2_GPIO_PIN |
                              OV7725_DATA_3_GPIO_PIN |
                              OV7725_DATA_4_GPIO_PIN |
                              OV7725_DATA_5_GPIO_PIN |
                              OV7725_DATA_6_GPIO_PIN |
                              OV7725_DATA_7_GPIO_PIN;
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(OV7725_DATA_GPIO_PORT, &GPIO_InitStructure);

    FIFO_OE_L();                /* 拉低使 FIFO 输出使能 */
    FIFO_WE_H();                /* 拉高使 FIFO 写允许 */
}
```

- FIFO_PREPAGE 实际是一个宏，它是在 main 函数中，判断到接收完成一幅图像后被调用的，它的作用是把 FIFO 读指针复位，使后面的数据读取从 FIFO 的 0 地址开始，其代码见代码清单 47_16 (在工程中，要把宏定义写在同一行或用续行符)。

列表 16: 代码清单 47-16：FIFO_PREPAGE 宏

```
#define FIFO_PREPARE                    do{\
                                        FIFO_RRST_L();\
                                        FIFO_RCLK_L();\
                                        FIFO_RCLK_H();\
                                        FIFO_RRST_H();\
                                        FIFO_RCLK_L();\
                                        FIFO_RCLK_H();\
                                        }while(0)
```

- ImagDisp 函数完成了从 FIFO 读取图像及显示到 LCD 的工作，每当 OV7725 输出完一幅图像，它被调用一次，在调用前，要用上面的 FIFO_PREAGE 宏复位 FIFO 读指针。它的代码见代码清单 47_17。

列表 17: 代码清单 47-17：ImagDisp 函数

```
/**
* @brief  设置显示位置
* @param  sx:x 起始显示位置
```

(下页继续)

(续上页)

```
* @param  sy:y 起始显示位置
* @param  width: 显示窗口宽度, 要求跟 OV7725_Window_Set 函数中的 width 一致
* @param  height: 显示窗口高度, 要求跟 OV7725_Window_Set 函数中的 height 一致
* @retval 无
*/
void ImagDisp(uint16_t sx,uint16_t sy,uint16_t width,uint16_t height)
{
    uint16_t i, j;
    uint16_t Camera_Data;

    ILI9341_OpenWindow(sx,sy,width,height);
    ILI9341_Write_Cmd ( CMD_SetPixel );

    for (i = 0; i < width; i++) {
        for (j = 0; j < height; j++) {
            /* 从 FIFO 读出一个 rgb565 像素到 Camera_Data 变量 */
            READ_FIFO_PIXEL(Camera_Data);
            ILI9341_Write_Data(Camera_Data);
        }
    }
}
```

在代码中，先根据输入参数在液晶屏设置了显示窗口，然后循环调用宏 READ_FIFO_PIXEL 读取 FIFO 数据，循环的次数就是摄像头输出的像素个数，代码中使用 width 和 height 控制，最后使用 LCD_WR_Data 函数把该图像数据显示到 LCD 上。宏 READ_FIFO_PIXEL 代码见*代码清单 47_18* (在工程中，要把宏定义写在同一行或用续行符)。

列表 18: 代码清单 47-18：READ_FIFO_PIXEL 宏

```
#define READ_FIFO_PIXEL(RGB565)     do{\
                                    RGB565=0;\
                                    FIFO_RCLK_L();\
```

(下页继续)

(续上页)

```
                RGB565 = (OV7725_DATA_GPIO_PORT->IDR) & 0xff00;\
                                FIFO_RCLK_H();\
                                FIFO_RCLK_L();\
            RGB565 |= (OV7725_DATA_GPIO_PORT->IDR >>8) & 0x00ff;\
                                FIFO_RCLK_H();\
                                }while(0)
```

这个宏把 FIFO 读取到的数据按 RGB565 的处理，保存到一个 16 位的变量中，LCD_WR_Data 函数可以直接利用这个数据，显示一个像素点到 LCD 上。

44.3.2.2.5 main 文件

利用前面介绍的函数，就可以驱动摄像头采集并显示图像了，关于摄像头模式或分辨率的配置本工程还提供了其它的函数进行修改，首先来看了解实现了采集流程的最基本的 main 函数，见*代码清单 47_19*。

列表 19: 代码清单 47-19：摄像头例程的 main 函数（main.c 文件）

```c
int main(void)
{
    float frame_count = 0;
    uint8_t retry = 0;
    /* 系统时钟初始化成 72MHz */
    SystemClock_Config();
    ILI9341_Init ();              //LCD 初始化
    ILI9341_GramScan( 3 );
    LCD_SetFont(&Font8x16);
    LCD_SetColors(RED,BLACK);
    ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */
    /******** 显示字符串示例 *******/
```

(下页继续)

(续上页)

```
    ILI9341_DispStringLine_EN(LINE(0),"BH OV7725 Test Demo");
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 初始化串口 */
    DEBUG_USART_Config();
    Key_GPIO_Config();
    SysTick_Init();

    printf("\r\n ** OV7725 摄像头实时液晶显示例程 ** \r\n");

    /* ov7725 gpio 初始化 */
    OV7725_GPIO_Config();

    LED_BLUE;
    /* ov7725 寄存器默认配置初始化 */
    while (OV7725_Init() != SUCCESS) {
        retry++;
        if (retry>5) {
            printf("\r\n没有检测到 OV7725 摄像头\r\n");
            ILI9341_DispStringLine_EN(LINE(2),"No OV7725 module detected!
↪");
            while (1);
        }
    }
    /* 根据摄像头参数组配置模式 */
    OV7725_Special_Effect(cam_mode.effect);
    /* 光照模式 */
    OV7725_Light_Mode(cam_mode.light_mode);
    /* 饱和度 */
    OV7725_Color_Saturation(cam_mode.saturation);
    /* 光照度 */
    OV7725_Brightness(cam_mode.brightness);
```

(下页继续)

(续上页)

```
    /* 对比度 */
    OV7725_Contrast(cam_mode.contrast);
    /* 特殊效果 */
    OV7725_Special_Effect(cam_mode.effect);

    /* 设置图像采样及模式大小 */
    OV7725_Window_Set(cam_mode.cam_sx,
                      cam_mode.cam_sy,
                      cam_mode.cam_width,
                      cam_mode.cam_height,
                      cam_mode.QVGA_VGA);

    /* 设置液晶扫描模式 */
    ILI9341_GramScan( cam_mode.lcd_scan );
    ILI9341_DispStringLine_EN(LINE(2),"OV7725 initialize success!");
    printf("\r\nOV7725 摄像头初始化完成\r\n");

    Ov7725_vsync = 0;

    while (1) {
        /* 接收到新图像进行显示 */
        if ( Ov7725_vsync == 2 ) {
            frame_count++;
            FIFO_PREPARE;           /*FIFO 准备 */
            ImagDisp(cam_mode.lcd_sx,
                     cam_mode.lcd_sy,
                     cam_mode.cam_width,
                     cam_mode.cam_height);       /* 采集并显示 */

            Ov7725_vsync = 0;
            LED1_TOGGLE;
```

(下页继续)

(续上页)

```
        }

        /* 每隔一段时间计算一次帧率 */
        if (Task_Delay[0] == 0) {
            printf("\r\nframe_ate = %.2f fps\r\n",frame_count/10);
            frame_count = 0;
            Task_Delay[0] = 10000;
        }
    }
}
```

main 函数的执行流程说明如下：

(1) main 函数首先调用了 ILI9341_Init、USART_Config 、SysTick_Init 和 LED_GPIO_Config 等函数初始化液晶、串口、Systick 定时器和 LED 外设，其中 Systick 每 ms 中断一次，为下面计算帧率的代码提供时间。

(2) 接下来 OV7725_GPIO_Config 函数，该函数内部封装了前面介绍的 SCCB_GPIO_Config、FIFO_GPIO_Config 及 VSYNC_GPIO_Config 函数，对控制摄像头使用的相关引脚都进行了初始化。

(3) 初始化好 SCCB 相关的引脚，就可通过 OV7725_Init 向 OV7725 芯片写入配置参数，代码中使用 while 循环在初始化失败时进行多次尝试。

(4) 调用 ILI9341_GramScan 设置液晶屏的扫描方向，使得液晶屏与摄像头的分辨率一致，做好显示的准备。

(5) 在 while 循环中，根据 Ov7725_vsync 标志，判断 FIFO 是否接收完了一幅图像。在中断服务程序中，若检测到两次 VSYNC 的下降沿 (表示接收完一幅图像)，会把 Ov7725_vsync 变量设置为 2。

(6) 判断接收完成一幅图像后，调用宏 FIFO_PREPARE 使读 FIFO 指针复位，使 ImagDisp 读取 FIFO 时，能读取得正确的数据并显示到液晶屏。

(7) 记录帧数目的变量 frame_count 加 1，这个变量用来统计帧率，每过一段时间后计算帧率通

过串口输出到上位机。最后把 Ov7725_vsync 置 0，使重新开始计数。

44.3.2.2.6 OV7725 的其它模式配置

以上是最基本的摄像头采集过程，而提供的工程在以上基础还增加了一些摄像头的配置，包括分辨率、光照度、饱和度、对比度及特殊模式等，如 OV7725_Window_Set、OV7725_Brightness、OV7725_Color_Saturation、OV7725_Contrast 和 OV7725_Special_Effect 等函数，这些函数的本质都是根据函数的输入参数，转化成对应的配置写入到 OV7725 摄像头的寄存器中，完成相应的配置，下面仅以 OV7725_Special_Effect 函数为例进行介绍，见*代码清单 47_20*。

列表 20: 代码清单 47-20 OV7725_Special_Effect 函数

```
/**
* @brief  设置特殊效果
* @param  eff: 特殊效果，参数范围 [0~6]:
    @arg 0: 正常
    @arg 1: 黑白
    @arg 2: 偏蓝
    @arg 3: 复古
    @arg 4: 偏红
    @arg 5: 偏绿
    @arg 6: 反相
* @retval 无
*/
void OV7725_Special_Effect(uint8_t eff)
{
    switch (eff) {
    case 0://正常
        SCCB_WriteByte(0xa6, 0x06);
        SCCB_WriteByte(0x60, 0x80);
        SCCB_WriteByte(0x61, 0x80);
        break;
```

(下页继续)

(续上页)

```
    case 1://黑白
        SCCB_WriteByte(0xa6, 0x26);
        SCCB_WriteByte(0x60, 0x80);
        SCCB_WriteByte(0x61, 0x80);
        break;

    case 2://偏蓝
        SCCB_WriteByte(0xa6, 0x1e);
        SCCB_WriteByte(0x60, 0xa0);
        SCCB_WriteByte(0x61, 0x40);
        break;

    case 3://复古
        SCCB_WriteByte(0xa6, 0x1e);
        SCCB_WriteByte(0x60, 0x40);
        SCCB_WriteByte(0x61, 0xa0);
        break;

    case 4://偏红
        SCCB_WriteByte(0xa6, 0x1e);
        SCCB_WriteByte(0x60, 0x80);
        SCCB_WriteByte(0x61, 0xc0);
        break;

    case 5://偏绿
        SCCB_WriteByte(0xa6, 0x1e);
        SCCB_WriteByte(0x60, 0x60);
        SCCB_WriteByte(0x61, 0x60);
        break;

    case 6://反相
```

(下页继续)

(续上页)

```
        SCCB_WriteByte(0xa6, 0x46);
        break;

    default:
        OV7725_DEBUG("Special Effect error!");
        break;
    }
}
```

从代码中可了解到，函数支持 0~6 作为输入参数，分别对应不同的模式，在函数内部根据不同的输入对寄存器写入相应配置。

44.3.2.2.7 摄像头配置结构体

由于分辨率、光照度、饱和度、对比度及特殊模式等摄像头配置涉及众多内容，特别是关于分辨率的配置，需要与液晶扫描方向匹配，否则容易出现显示错误的现象，为了方便使用，工程中定义了一个结构体类型专门用于设置摄像头的这些配置，在初始化摄像头或想修改配置的时候，修改该变量的内容，然后把它作为参数输入到各种配置函数调用即可。该摄像头配置结构体类型见*代码清单 47_21*。

列表 21: 代码清单 47-21 摄像头配置结构体 (bsp_ov7725.h 文件)

```
/* 摄像头配置结构体 */
typedef struct {
    uint8_t QVGA_VGA; //0: QVGA 模式, 1: VGA 模式

    /*VGA:sx + width <= 320 或 240 ,sy+height <= 320 或 240*/
    /*QVGA:sx + width <= 320 ,sy+height <= 240*/
    uint16_t cam_sx; //摄像头窗口 X 起始位置
    uint16_t cam_sy; //摄像头窗口 Y 起始位置
```

(下页继续)

(续上页)

```
    uint16_t cam_width;//图像分辨率，宽
    uint16_t cam_height;//图像分辨率，高

    uint16_t lcd_sx;//图像显示在液晶屏的 X 起始位置
    uint16_t lcd_sy;//图像显示在液晶屏的 Y 起始位置
    uint8_t lcd_scan;//液晶屏的扫描模式 (0-7)

    uint8_t light_mode;//光照模式，参数范围 [0~5]
    int8_t saturation;//饱和度，参数范围 [-4 ~ +4]
    int8_t brightness;//光照度，参数范围 [-4~+4]
    int8_t contrast;//对比度，参数范围 [-4~+4]
    uint8_t effect; //特殊效果，参数范围 [0~6]:

} OV7725_MODE_PARAM;
```

结构体类型定义中的代码注释有注明各种配置参数的范围，其中 QVGA_VGA 可以配置采样图像的模式，cam_sx/y 可以设置摄像头采样坐标的原点，cam_width/ height 可设置图像分辨率，分辨率调小时，可以提高图像的采集帧率，lcd_sx/y 可以配置图像显示在液晶屏的起始位置，而 lcd_scan 可以设置液晶屏的扫描模式，其余的配置如光照度、饱和度等可以按需求设置。

在配置分辨率时，必须注意调节范围，如 QVGA 模式下最大为 320*240（宽 * 高），若设置分辨率为 240*320（宽 * 高）时，由于设置分辨率的高度超出 QVGA 的 240 限制，会导致出错，此时把对应的模式改成 VGA 模式即可，因为 VGA 模式的最大分辨率为 640*480（宽 * 高），除了不超过 QVGA、VGA 模式的极限外，还要注意它们在液晶屏的扫描模式和起始位置的配置不会超出液晶显示范围。

在工程中，提供了三组摄像头及显示配置范例，实验时可以亲自尝试一下来了解，见*代码清单 47_22*。

列表 22: 代码清单 47-22 三组摄像头初始化配置（bsp_ov7725.c 文件）

```
//摄像头初始化配置
//注意：使用这种方式初始化结构体，要在 c/c++ 选项中选择 C99 mode
OV7725_MODE_PARAM cam_mode = {

    /* 以下包含几组摄像头配置，可自行测试，保留一组，把其余配置注释掉即可 */
    /************* 配置 1********* 横屏显示 ******************************/

    .QVGA_VGA = 0,  //QVGA 模式
    .cam_sx = 0,
    .cam_sy = 0,

    .cam_width = 320,
    .cam_height = 240,

    .lcd_sx = 0,
    .lcd_sy = 0,
    .lcd_scan = 3, //LCD 扫描模式，本横屏配置可用 1、3、5、7 模式

    //以下可根据自己的需要调整，参数范围见结构体类型定义
    .light_mode = 0,//自动光照模式
    .saturation = 0,
    .brightness = 0,
    .contrast = 0,
    .effect = 0,    //正常模式

    /********** 配置 2********* 竖屏显示 ******************************/
    /* 竖屏显示需要 VGA 模式，同分辨率情况下，比 QVGA 帧率稍低 */
    /*VGA 模式分辨率为 640*480，从中取出 240*320 的图像进行竖屏显示 */
    /* 本工程不支持超过 320*240 或 240*320 的分辨率配置 */
```

(下页继续)

(续上页)

```
//  .QVGA_VGA = 1,  //VGA 模式
//  //取 VGA 模式居中的窗口，可根据实际需要调整
//  .cam_sx = (640-240)/2,
//  .cam_sy = (480-320)/2,
//
//  .cam_width = 240,
//  .cam_height = 320, //在 VGA 模式下，此值才可以大于 240
//
//  .lcd_sx = 0,
//  .lcd_sy = 0,
//  .lcd_scan = 0, //LCD 扫描模式，本竖屏配置可用 0、2、4、6 模式
//
//  //以下可根据自己的需要调整，参数范围见结构体类型定义
//  .light_mode = 0,//自动光照模式
//  .saturation = 0,
//  .brightness = 0,
//  .contrast = 0,
//  .effect = 0,    //正常模式

    /******* 配置 3************ 小分辨率 ****************************/
    /* 小于 320*240 分辨率的，可使用 QVGA 模式，设置的时候注意液晶屏边界 */

//  .QVGA_VGA = 0,  //QVGA 模式
//  //取 QVGA 模式居中的窗口，可根据实际需要调整
//  .cam_sx = (320-100)/2,
//  .cam_sy = (240-150)/2,
//
//  .cam_width = 100,
//  .cam_height = 150,
//
//  /* 液晶屏的显示位置也可以根据需要调整，注意不要超过边界即可 */
```

(下页继续)

(续上页)

```
//  .lcd_sx = 50,
//  .lcd_sy = 50,
//  .lcd_scan = 3, //LCD 扫描模式，0-7 模式都支持，注意不要超过边界即可

//  //以下可根据自己的需要调整，参数范围见结构体类型定义
//  .light_mode = 0,//自动光照模式
//  .saturation = 0,
//  .brightness = 0,
//  .contrast = 0,
//  .effect = 0,     //正常模式
};
```

代码中使用 OV7725_MODE_PARAM 类型定义了一个 cam_mode 变量并对其结构体成员赋予了初始值。本工程默认使用以上第一组配置，采集 320*240 的 QVGA 图像在液晶屏上横屏显示；而第二组配置是 240*320 的 VGA 图像在液晶屏上竖屏显示，注意两组配置中 QVGA_VGA 和 lcd_mode 变量值的区别；第三组配置是 50*50 的分辨率，由于分辨率比较小，其宽和高都没有超出 QVGA 及液晶屏显示范围，所以液晶 0-7 的扫描方式都支持。可亲自尝试以上各组配置，使用时注释掉其余两组即可，也可以在范例的基础上，自己进行修改测试。

使用摄像头配置结构体时，在初始化摄像头时要调用相应的函数对寄存器进行赋值，所以，main 函数需要作出相应的修改，见*代码清单 47_23*。

列表 23: 代码清单 47-23 根据摄像头配置结构体初始化的 main 函数

```
int main(void)
{
float frame_count = 0;
uint8_t retry = 0;
/* 系统时钟初始化成 72MHz */
SystemClock_Config();
ILI9341_Init ();              //LCD 初始化
```

(下页继续)

(续上页)

```
ILI9341_GramScan( 3 );
LCD_SetFont(&Font8x16);
LCD_SetColors(RED,BLACK);
ILI9341_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH); /* 清屏，显示全黑 */
/******** 显示字符串示例 *******/
ILI9341_DispStringLine_EN(LINE(0),"BH OV7725 Test Demo");
/* LED 端口初始化 */
LED_GPIO_Config();
/* 初始化串口 */
DEBUG_USART_Config();
Key_GPIO_Config();
SysTick_Init();

printf("\r\n ** OV7725 摄像头实时液晶显示例程 ** \r\n");

/* ov7725 gpio 初始化 */
OV7725_GPIO_Config();

LED_BLUE;
/* ov7725 寄存器默认配置初始化 */
while(OV7725_Init() != SUCCESS)
{
    retry++;
    if(retry>5)
    {
    printf("\r\n没有检测到 OV7725 摄像头\r\n");
    ILI9341_DispStringLine_EN(LINE(2),"No OV7725 module detected!");
    while(1);
    }
}
/* 根据摄像头参数组配置模式 */
OV7725_Special_Effect(cam_mode.effect);
```

(下页继续)

(续上页)

```
/* 光照模式 */
OV7725_Light_Mode(cam_mode.light_mode);
/* 饱和度 */
OV7725_Color_Saturation(cam_mode.saturation);
/* 光照度 */
OV7725_Brightness(cam_mode.brightness);
/* 对比度 */
OV7725_Contrast(cam_mode.contrast);
/* 特殊效果 */
OV7725_Special_Effect(cam_mode.effect);

/* 设置图像采样及模式大小 */
OV7725_Window_Set(cam_mode.cam_sx,
                  cam_mode.cam_sy,
                  cam_mode.cam_width,
                  cam_mode.cam_height,
                  cam_mode.QVGA_VGA);

/* 设置液晶扫描模式 */
ILI9341_GramScan( cam_mode.lcd_scan );
ILI9341_DispStringLine_EN(LINE(2),"OV7725 initialize success!");
printf("\r\nOV7725 摄像头初始化完成\r\n");

Ov7725_vsync = 0;

while(1)
{
    /* 接收到新图像进行显示 */
    if( Ov7725_vsync == 2 )
    {
    frame_count++;
    FIFO_PREPARE;          /*FIFO 准备 */
```

(下页继续)

(续上页)

```
    ImagDisp(cam_mode.lcd_sx,
                cam_mode.lcd_sy,
                cam_mode.cam_width,
                cam_mode.cam_height);      /* 采集并显示 */

    Ov7725_vsync = 0;
    LED1_TOGGLE;

    }

    /* 检测按键 */
    if( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON  )
    {
    /*LED 反转 */
    LED2_TOGGLE;

    }
    /* 检测按键 */
    if( Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON  )
    {
    /*LED 反转 */
    LED3_TOGGLE;

    /* 动态配置摄像头的模式，
    有需要可以添加使用串口、用户界面下拉选择框
    方式修改这些变量，
    达到程序运行时更改摄像头模式的目的 */

    cam_mode.QVGA_VGA = 0,   //QVGA 模式
    cam_mode.cam_sx = 0,
    cam_mode.cam_sy = 0,
```

(下页继续)

(续上页)

```
    cam_mode.cam_width = 320,
    cam_mode.cam_height = 240,

    cam_mode.lcd_sx = 0,
    cam_mode.lcd_sy = 0,
    cam_mode.lcd_scan = 3,
    //LCD 扫描模式，本横屏配置可用 1、3、5、7 模式

    //以下可根据自己的需要调整，参数范围见结构体类型定义
    cam_mode.light_mode = 0,//自动光照模式
    cam_mode.saturation = 0,
    cam_mode.brightness = 0,
    cam_mode.contrast = 0,
    cam_mode.effect = 1,     //黑白模式

    /* 根据摄像头参数写入配置 */
    OV7725_Special_Effect(cam_mode.effect);
    /* 光照模式 */
    OV7725_Light_Mode(cam_mode.light_mode);
    /* 饱和度 */
    OV7725_Color_Saturation(cam_mode.saturation);
    /* 光照度 */
    OV7725_Brightness(cam_mode.brightness);
    /* 对比度 */
    OV7725_Contrast(cam_mode.contrast);
    /* 特殊效果 */
    OV7725_Special_Effect(cam_mode.effect);

    /* 设置图像采样及模式大小 */
    OV7725_Window_Set(cam_mode.cam_sx,
                      cam_mode.cam_sy,
                      cam_mode.cam_width,
```

(下页继续)

(续上页)

```
                    cam_mode.cam_height,
                    cam_mode.QVGA_VGA);

    /* 设置液晶扫描模式 */
    ILI9341_GramScan( cam_mode.lcd_scan );
    }

    /* 每隔一段时间计算一次帧率 */
    if(Task_Delay[0] == 0)
    {
    printf("\r\nframe_ate = %.2f fps\r\n",frame_count/10);
    frame_count = 0;
    Task_Delay[0] = 10000;
    }
}
}
```

相对于前面介绍的摄像头基本初始化过程，本 main 函数主要增加了对 OV7725_Window_Set、OV7725_Brightness、OV7725_Color_Saturation、OV7725_Contrast 和 OV7725_Special_Effect 等函数的调用，根据摄像头配置结构体 cam_mode 向 OV7725 写入寄存器内容，初始化好后，摄像头图像的采集和显示与普通方式无异。

代码中还增加了按键检测，按下了开发板的 KEY2 按键后，会向摄像头配置结构体 cam_mode 赋予新的配置并写入到 OV7725 寄存器中，以上代码把采集的图像设置成了黑白模式。

44.3.2.3 下载验证

把摄像头模块接入到开发板的摄像头接口上，用 USB 线连接开发板，编译程序下载到实验板并上电复位。即可看到 LCD 上输出摄像头拍到的图像，若图片显示不够清晰，可调整镜头进行调焦，使得到清晰的图像。

联系我们

东莞野火电子技术有限公司

官网：www.embedfire.com

论坛：www.firebbs.cn

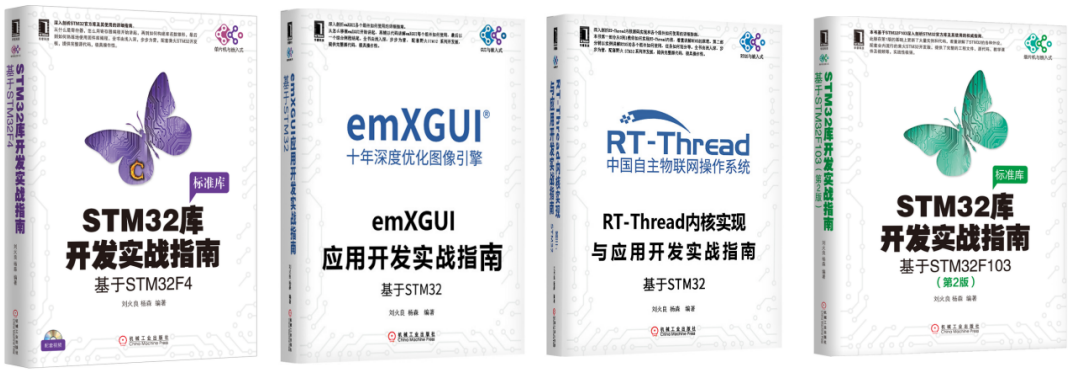
资料中心：doc.embedfire.com

淘宝：fire-stm32.taobao.com

邮箱：firege@embedfire.com

电话：0769-33894118

关注野火公众号，可下载野火全部产品的资料（书籍、视频、程序）



野火 STM32 系列图书，关注公众号即可下载电子版

rst 投稿示例文件

如何参与项目

本项目从代码至文档全部开源，期待大家的参与！

无论是小到发现错别字、发现代码写得不规范、有 bug 抑或是想参与到文档的编辑、投稿完整的文章，我们都热烈欢迎。

具体参与方式请阅读如下说明：如何参与野火开源项目

项目源码

本项目代码及文档源码托管在 github 和 gitee，维护时主要以 github 开发，gitee 主要是方便国内用户下载，可在这两个平台提交 **pull request** 参与到本项目。

- github 仓库地址：https://github.com/有待补充
- gitee 仓库地址：https://gitee.com/有待补充

文档在线阅读与 pdf

本项目的文档是托管在 **readthedocs** 网站的，可在该网站在线浏览文档或者生成 **pdf** 文档离线阅读。

常见问题

版权说明

野火电子保留本项目的所有版权。