[野火]STM32 HAL 库开发实战指南
—基于野火霸天虎开发板

EmbedFire
野火电子

2022 年 02 月 19 日

Contents

关于本项目 **2**
关于野火 **3**
开源共享，共同进步 3
联系方式 3
TODO 和悬赏任务 **4**
第 1 章 如何使用本书 **5**
1.1 本书的参考资料 5
1.2 本书的编写风格 5
1.3 本书的配套硬件 6
1.4 本书的技术论坛 6
第 2 章 如何安装 KEIL5 . . . . . **7**
2.1 温馨提示 7
2.2 获取 KEIL5 安装包 7
2.3 开始安装 KEIL5 8
2.4 安装 STM32 芯片包 13
第 3 章 如何用 DAP 仿真器下载程序 **16**
3.1 仿真器简介 16
3.2 硬件连接 17
3.3 仿真器配置 17
3.3.1 Debug 选项配置 18
3.3.2 Utilities 选项配置 19
3.3.3 Debug Settings 选项配置 20
3.4 选择目标板 21
3.5 下载程序 22
第 4 章 初识 STM32 . . . . . **23**
4.1 什么是 STM32 23
4.1.1 STM32 诞生的背景 23

4.2 STM32 能做什么 24
4.2.1 智能手环 24
4.2.2 微型四轴飞行器 27
4.2.3 淘宝众筹 28
4.3 STM32 怎么选型 29
4.3.1 STM32 分类 29
4.3.2 STM32 命名方法 30
4.3.3 选择合适的 MCU 32
4.3.4 PCB 哪里打样 36
第 5 章 什么是寄存器 37
5.1 什么是寄存器 37
5.2 STM32 长啥样 37
5.3 芯片里面有什么 40
5.4 存储器映射 41
5.4.1 存储器映射 42
5.4.2 存储器区域功能划分 44
5.5 寄存器映射 46
5.5.1 STM32 的外设地址映射 48
5.5.2 C 语言对寄存器的封装 52
5.5.3 修改寄存器的位操作方法 58
第 6 章 新建工程—寄存器版 62
6.1 新建工程 62
6.1.1 新建本地工程文件夹 62
6.2 开始新建工程 63
6.2.1 选择 CPU 型号 64
6.2.2 在线添加库文件 65
6.2.3 添加文件 66
6.2.4 配置魔术棒选项卡 68
6.2.5 下载器配置 71
6.3 下载程序 75

第 7 章 使用寄存器点亮 LED 灯 77
7.1 GPIO 简介 77
7.2 GPIO 框图剖析 78
7.2.1 基本结构分析 78
7.2.2 GPIO 工作模式 84
7.3 实验：使用寄存器点亮 LED 灯 85
7.3.1 硬件连接 86
7.3.2 启动文件 87
7.3.3 stm32f4xx.h 文件 89
7.3.4 main 文件 91
7.3.5 下载验证 100
第 8 章 自己写库—构建库函数雏形 101
8.1 什么是 STM32 函数库 101
8.2 为什么采用库来开发及学习？ 102
8.3 实验：构建库函数雏形 103
8.3.1 修改寄存器地址封装 104
8.3.2 定义访问外设的结构体指针 106
8.3.3 定义位操作函数 110
8.3.4 定义初始化结构体 GPIO_InitTypeDef 115
8.3.5 定义引脚模式的枚举类型 116
8.3.6 定义 GPIO 初始化函数 118
8.3.7 全新面貌，使用函数点亮 LED 灯 121
8.3.8 下载验证 123
8.3.9 总结 123
第 9 章 初识 HAL 固件库 125
9.1 CMSIS 标准及库层次关系 125
9.1.1 库目录、文件简介 127
9.1.2 库各文件间的关系 137
9.2 使帮助文档 139
9.2.1 常用官方资料 139

第 10 章　使用 STM32CubeMX 新建工程 141
10.1　软件安装 141
10.1.1　安装 STM32CubeMX 软件 144
10.2　新建工程 149
10.2.1　新建工程 149
10.2.2　下载验证 160
第 11 章　新建工程—库函数版 161
11.1　新建工程 161
11.1.1　新建本地工程文件夹 161
11.1.2　新建工程 162
第 12 章　GPIO 输出—使用固件库点亮 LED 176
12.1　硬件设计 176
12.2　软件设计 177
12.2.1　编程要点 177
12.2.2　代码分析 177
12.2.3　下载验证 188
12.3　STM32 HAL 库补充知识 189
12.3.1　SystemInit 函数去哪了? 189
12.3.2　断言 189
12.3.3　Doxygen 注释规范 192
12.3.4　防止头文件重复包含 193
第 13 章　GPIO 输入—按键检测 195
13.1　硬件设计 195
13.2　软件设计 196
13.2.1　编程要点 196
13.2.2　代码分析 197
13.2.3　下载验证 202
第 14 章　启动文件详解 203
14.1　启动文件简介 203
14.2　查找 ARM 汇编指令 203
14.3　启动文件代码讲解 205

14.3.1 Stack—栈 . . . 205
14.3.2 Heap 堆 . . . 206
14.3.3 向量表 . . . 207
14.3.4 复位程序 . . . 211
14.3.5 中断服务程序 . . . 213
14.3.6 用户堆栈初始化 . . . 214
第 15 章 RCC—使用 HSE/HSI 配置时钟 . . . 218
15.1 RCC 主要作用—时钟部分 . . . 218
15.2 RCC 框图剖析—时钟部分 . . . 218
15.2.1 系统时钟 . . . 220
15.2.2 其他时钟 . . . 224
15.3 配置系统时钟实验 . . . 226
15.3.1 使用 HSE . . . 226
15.3.2 使用 HSI . . . 226
15.3.3 硬件设计 . . . 227
15.3.4 软件设计 . . . 227
15.3.5 下载验证 . . . 237
第 16 章 STM32 中断应用概览 . . . 238
16.1 异常类型 . . . 238
16.2 NVIC 简介 . . . 240
16.2.1 NVIC 寄存器简介 . . . 240
16.2.2 NVIC 中断配置固件库 . . . 241
16.3 优先级的定义 . . . 242
16.3.1 优先级定义 . . . 242
16.3.2 优先级分组 . . . 243
16.4 中断编程 . . . 244
第 17 章 EXTI—外部中断/事件控制器 . . . 246
17.1 EXTI 简介 . . . 246
17.2 EXTI 功能框图 . . . 246
17.3 中断/事件线 . . . 249
17.4 外部中断控制实验 . . . 251

17.4.1 硬件设计 . . . 251
17.4.2 软件设计 . . . 252
17.4.3 下载验证 . . . 257
第 18 章 SysTick—系统定时器 . . . 258
18.1 SysTick 简介 . . . 258
18.2 SysTick 寄存器介绍 . . . 258
18.3 SysTick 定时实验 . . . 261
18.3.1 硬件设计 . . . 261
18.3.2 软件设计 . . . 261
第 19 章 通讯的基本概念 . . . 272
19.1 串行通讯与并行通讯 . . . 272
19.2 全双工、半双工及单工通讯 . . . 274
19.3 同步通讯与异步通讯 . . . 276
19.4 通讯速率 . . . 277
第 20 章 USART—串口通讯 . . . 279
20.1 串口通讯协议简介 . . . 279
20.1.1 物理层 . . . 279
20.1.2 协议层 . . . 285
20.2 STM32 的 USART 简介 . . . 287
20.3 USART 功能框图 . . . 288
20.3.1 ① 功能引脚 . . . 290
20.3.2 ② 数据寄存器 . . . 291
20.3.3 ③ 控制器 . . . 291
20.3.4 ④ 小数波特率生成 . . . 294
20.3.5 校验控制 . . . 295
20.3.6 中断控制 . . . 296
20.4 USART 初始化结构体详解 . . . 296
20.4.1 USART 初始化结构体 . . . 297
20.5 USART1 接发通信实验 . . . 298
20.5.1 硬件设计 . . . 298
20.5.2 软件设计 . . . 299

20.5.3 下载验证 . . . 308
20.6 USART1 指令控制 RGB 彩灯实验 . . . 309
20.6.1 硬件设计 . . . 310
20.6.2 软件设计 . . . 310
20.6.3 下载验证 . . . 319
第 21 章 DMA—直接存储区访问 . . . 320
21.1 DMA 简介 . . . 320
21.2 DMA 功能框图 . . . 321
21.2.1 ① 外设通道选择 . . . 322
21.2.2 ② 仲裁器 . . . 324
21.2.3 ③FIFO . . . 324
21.2.4 ④ 存储器端口、⑤ 外设端口 . . . 325
21.3 DMA 数据配置 . . . 327
21.3.1 DMA 传输模式 . . . 327
21.3.2 源地址和目标地址 . . . 328
21.3.3 流控制器 . . . 328
21.3.4 循环模式 . . . 329
21.3.5 传输类型 . . . 329
21.3.6 直接模式 . . . 331
21.3.7 双缓冲模式 . . . 331
21.3.8 DMA 中断 . . . 332
21.4 DMA 初始化结构体详解 . . . 332
21.4.1 DMA_InitTypeDef 初始化结构体 . . . 333
21.4.2 DMA_HandleTypeDef 初始化结构体 . . . 334
21.5 DMA 存储器到存储器模式实验 . . . 336
21.5.1 硬件设计 . . . 336
21.5.2 软件设计 . . . 337
21.6 DMA 存储器到外设模式实验 . . . 343
21.6.1 硬件设计 . . . 343
21.6.2 软件设计 . . . 344

第 22 章　常用存储器介绍 350
22.1　存储器种类 350
22.2　RAM 存储器 351
22.2.1　DRAM 351
22.2.2　SRAM 352
22.2.3　DRAM 与 SRAM 的应用场合 353
22.3　非易失性存储器 354
22.3.1　ROM 存储器 354
22.3.2　FLASH 存储器 355
第 23 章　I2C—读写 EEPROM 357
23.1　I2C 协议简介 357
23.1.1　I2C 物理层 358
23.1.2　协议层 359
23.2　STM32 的 I2C 特性及架构 364
23.2.1　STM32 的 I^2C 外设简介 365
23.2.2　STM32 的 I^2C 架构剖析 366
23.2.3　通讯过程 369
23.3　I2C 初始化结构体详解 371
23.4　I2C—读写 EEPROM 实验 373
23.4.1　硬件设计 374
23.4.2　软件设计 375
23.4.3　下载验证 392
第 24 章　SPI—读写串行 FLASH 393
24.1　SPI 协议简介 393
24.1.1　SPI 物理层 394
24.1.2　协议层 395
24.2　STM32 的 SPI 特性及架构 398
24.2.1　STM32 的 SPI 外设简介 399
24.2.2　STM32 的 SPI 架构剖析 399
24.2.3　通讯过程 402
24.3　SPI 初始化结构体详解 403

24.4 SPI—读写串行 FLASH 实验 . . . 405
24.4.1 硬件设计 . . . 406
24.4.2 软件设计 . . . 407
24.4.3 下载验证 . . . 441
第 25 章 串行 FLASH 文件系统 FatFs . . . 442
25.1 文件系统 . . . 442
25.2 FatFs 文件系统简介 . . . 443
25.2.1 FatFs 的目录结构 . . . 443
25.2.2 FatFs 帮助文档 . . . 444
25.2.3 FATFS 源码 . . . 446
25.3 FatFs 文件系统移植实验 . . . 447
25.3.1 FatFs 程序结构图 . . . 447
25.3.2 硬件设计 . . . 448
25.3.3 FatFs 移植步骤 . . . 449
25.3.4 FatFs 底层设备驱动函数 . . . 452
25.3.5 FatFs 功能配置 . . . 461
25.3.6 FatFs 功能测试 . . . 462
25.3.7 下载验证 . . . 468
25.4 FatFs 功能使用实验 . . . 468
25.4.1 硬件设计 . . . 469
25.4.2 软件设计 . . . 469
25.4.3 下载验证 . . . 477
第 26 章 FSMC—扩展外部 SRAM . . . 478
26.1 SRAM 控制原理 . . . 478
26.1.1 SRAM 信号线 . . . 481
26.1.2 存储器矩阵 . . . 481
26.1.3 地址译码器、列 I/O 及 I/O 数据电路 . . . 482
26.1.4 控制电路 . . . 483
26.1.5 SRAM 的读写流程 . . . 483
26.2 FSMC 简介 . . . 485
26.3 FSMC 框图剖析 . . . 485

26.3.1 通讯引脚 . . . 486
26.3.2 存储器控制器 . . . 487
26.3.3 时钟控制逻辑 . . . 488
26.4 FSMC 的地址映射 . . . 488
26.5 FSMC 控制 SRAM 的时序 . . . 490
26.6 SRAM 时序结构体 . . . 492
26.7 SRAM 初始化结构体 . . . 494
26.8 FSMC—扩展外部 SRAM 实验 . . . 497
26.8.1 硬件设计 . . . 498
26.8.2 软件设计 . . . 499
26.8.3 下载验证 . . . 513
第 27 章 LCD—液晶显示 . . . 514
27.1 显示器简介 . . . 514
27.1.1 液晶显示器 . . . 514
27.1.2 LED 和 OLED 显示器 . . . 516
27.1.3 显示器的基本参数 . . . 519
27.2 液晶控制原理 . . . 520
27.2.1 液晶面板的控制信号 . . . 522
27.2.2 液晶数据传输时序 . . . 525
27.2.3 显存 . . . 527
27.3 野火 4.3 寸液晶屏简介 . . . 527
27.3.1 4.3 寸电容触摸屏实物 . . . 527
27.3.2 NT35510 液晶控制器简介 . . . 529
27.3.3 液晶屏的信号线及 8080 时序 . . . 531
27.4 使用 STM32 的 FSMC 模拟 8080 接口时序 . . . 534
27.4.1 FSMC 简介 . . . 534
27.4.2 用 FSMC 模拟 8080 时序 . . . 538
27.5 NOR FLASH 时序结构体 . . . 539
27.6 FSMC 初始化结构体 . . . 541
27.7 FSMC—液晶显示实验 . . . 544
27.7.1 硬件设计 . . . 546

27.7.2 软件设计 . . . 551
27.7.3 下载验证 . . . 585
第 28 章 LCD—液晶显示中英文 . . . 586
28.1 字符编码 . . . 586
28.1.1 ASCII 编码 . . . 586
28.1.2 中文编码 . . . 590
28.1.3 Unicode 字符集和编码 . . . 596
28.1.4 UTF-32 . . . 596
28.1.5 UTF-16 . . . 597
28.1.6 UTF-8 . . . 598
28.1.7 BOM . . . 599
28.2 什么是字模? . . . 599
28.2.1 字模的构成 . . . 600
28.2.2 字模显示原理 . . . 601
28.2.3 如何制作字模 . . . 604
28.2.4 字模寻址公式 . . . 606
28.2.5 存储字模文件 . . . 607
28.3 各种模式的液晶显示字符实验 . . . 607
28.3.1 硬件设计 . . . 608
28.3.2 显示 ASCII 编码的字符 . . . 609
28.3.3 显示 GB2312 编码的字符 . . . 624
28.3.4 显示任意大小的字符 . . . 638
28.3.5 下载验证 . . . 650
第 29 章 电容触摸屏—触摸画板 . . . 651
29.1 触摸屏简介 . . . 651
29.1.1 电阻式触摸屏检测原理 . . . 653
29.1.2 电容式触摸屏检测原理 . . . 656
29.2 电容触摸屏控制芯片 . . . 658
29.2.1 GT917S 芯片的引脚 . . . 658
29.2.2 上电时序与 I2C 设备地址 . . . 659
29.2.3 寄存器配置 . . . 660

29.2.4 读取坐标信息 . . . 662
29.3 电容触摸屏—触摸画板实验 . . . 665
29.3.1 硬件设计 . . . 667
29.3.2 软件设计 . . . 669
29.3.3 下载验证 . . . 709
第 30 章 ADC—电压采集 . . . 710
30.1 ADC 简介 . . . 710
30.2 ADC 功能框图剖析 . . . 712
30.2.1 ① 电压输入范围 . . . 713
30.2.2 ② 输入通道 . . . 713
30.2.3 ③ 转换顺序 . . . 715
30.2.4 ④ 触发源 . . . 716
30.2.5 ⑤ 转换时间 . . . 717
30.2.6 ⑥ 数据寄存器 . . . 717
30.2.7 ⑦ 中断 . . . 719
30.2.8 ⑧ 电压转换 . . . 720
30.3 ADC 初始化结构体详解 . . . 720
30.3.1 ADC_HandleTypeDef 结构体 . . . 720
30.3.2 ADC_InitTypeDef 结构体 . . . 721
30.3.3 ADC_ChannelConfTypeDef 结构体 . . . 723
30.4 独立模式单通道采集实验 . . . 723
30.4.1 硬件设计 . . . 723
30.4.2 软件设计 . . . 724
30.5 独立模式多通道采集实验 . . . 731
30.5.1 硬件设计 . . . 731
30.5.2 软件设计 . . . 731
30.6 三重 ADC 交替模式采集实验 . . . 740
30.6.1 硬件设计 . . . 741
30.6.2 软件设计 . . . 741
第 31 章 TIM—基本定时器 . . . 752
31.1 TIM 简介 . . . 752

31.2 基本定时器 . . . 754
31.3 基本定时器功能框图 . . . 754
31.3.1 ① 时钟源 . . . 755
31.3.2 ② 控制器 . . . 756
31.3.3 ③ 计数器 . . . 756
31.3.4 定时器周期计算 . . . 757
31.4 定时器初始化结构体详解 . . . 758
31.5 基本定时器定时实验 . . . 759
31.5.1 硬件设计 . . . 759
第 32 章 TIM—高级定时器 . . . 764
32.1 高级控制定时器 . . . 764
32.2 高级控制定时器功能框图 . . . 765
32.2.1 ① 时钟源 . . . 766
32.2.2 ② 控制器 . . . 770
32.2.3 ③ 时基单元 . . . 770
32.2.4 ④ 输入捕获 . . . 773
32.2.5 ⑤ 输出比较 . . . 776
32.2.6 ⑥ 断路功能 . . . 780
32.3 输入捕获应用 . . . 781
32.3.1 测量脉宽或者频率 . . . 781
32.3.2 PWM 输入模式 . . . 782
32.4 输出比较应用 . . . 784
32.4.1 PWM 输出模式 . . . 784
32.5 定时器初始化结构体详解 . . . 786
32.5.1 TIM_TimeBaseInitTypeDef . . . 787
32.5.2 TIM_OC_InitTypeDef . . . 788
32.5.3 TIM_IC_InitTypeDef . . . 789
32.5.4 TIM_BreakDeadTimeConfigTypeDef . . . 791
32.6 PWM 互补输出实验 . . . 792
32.6.1 硬件设计 . . . 792
32.6.2 软件设计 . . . 793

32.7 PWM 输入捕获实验 . . . 800
32.7.1 硬件设计 . . . 800
32.7.2 软件设计 . . . 801
32.7.3 下载验证 . . . 810
第 33 章 TIM—电容按键检测 . . . 812
33.1 电容按键原理 . . . 812
33.2 电容按键检测实验 . . . 815
33.2.1 硬件设计 . . . 816
33.2.2 软件设计 . . . 816
33.2.3 下载验证 . . . 828
第 34 章 IWDG—独立看门狗 . . . 829
34.1 IWDG 简介 . . . 829
34.2 IWDG 功能框图剖析 . . . 829
34.2.1 ① 独立看门狗时钟 . . . 830
34.2.2 ② 计数器时钟 . . . 830
34.2.3 ③ 计数器 . . . 830
34.2.4 ④ 重装载寄存器 . . . 830
34.2.5 ⑤ 键寄存器 . . . 831
34.2.6 ⑥ 状态寄存器 . . . 831
34.3 怎么用 IWDG . . . 831
34.4 IWDG 超时实验 . . . 832
34.4.1 硬件设计 . . . 832
34.4.2 软件设计 . . . 832
34.4.3 下载验证 . . . 837
第 35 章 WWDG—窗口看门狗 . . . 838
35.1 WWDG 简介 . . . 838
35.2 WWDG 功能框图剖析 . . . 840
35.2.1 ① 窗口看门狗时钟 . . . 840
35.2.2 ② 计数器时钟 . . . 840
35.2.3 ③ 计数器 . . . 841
35.2.4 ④ 窗口值 . . . 841

35.2.5 ⑤ 计算看门狗超时时间 . . . 843
35.3 怎么用 WWDG . . . 844
35.4 WWDG 喂狗实验 . . . 844
35.4.1 硬件设计 . . . 844
35.4.2 软件设计 . . . 845
第 36 章 SDIO—SD 卡读写测试 . . . 851
36.1 SDIO 简介 . . . 851
36.2 SD 卡物理结构 . . . 853
36.3 SDIO 总线 . . . 855
36.3.1 总线拓扑 . . . 855
36.3.2 总线协议 . . . 857
36.3.3 命令 . . . 860
36.3.4 响应 . . . 863
36.4 SD 卡的操作模式及切换 . . . 864
36.4.1 SD 卡的操作模式 . . . 864
36.4.2 卡识别模式 . . . 865
36.4.3 数据传输模式 . . . 867
36.5 STM32 的 SDIO 功能框图 . . . 869
36.5.1 控制单元 . . . 870
36.5.2 命令路径 . . . 871
36.5.3 数据路径 . . . 872
36.5.4 数据 FIFO . . . 874
36.5.5 适配器寄存器 . . . 875
36.6 SDMMC 初始化结构体 . . . 875
36.7 SDMMC 命令初始化结构体 . . . 877
36.8 SDMMC 数据初始化结构体 . . . 878
36.9 SD 卡读写测试实验 . . . 879
36.9.1 硬件设计 . . . 879
36.9.2 软件设计 . . . 880
36.9.3 下载验证 . . . 923

第 37 章　基于 SD 卡的 FatFs 文件系统 924
37.1　FatFs 移植步骤 924
37.2　FatFs 接口函数 928
37.2.1　宏定义和存储设备状态获取函数 928
37.2.2　存储设备初始化函数 928
37.2.3　存储设备数据读取函数 929
37.2.4　存储设备数据写入函数 931
37.2.5　其他控制函数 932
37.3　FatFs 功能测试 934
37.3.1　变量定义 934
37.3.2　主函数 935
37.3.3　下载验证 939
第 38 章　I2S—音频播放与录音输入 940
38.1　I2S 简介 940
38.1.1　数字音频技术 940
38.1.2　I2S 总线接口 942
38.1.3　音频数据传输协议标准 942
38.2　I2S 功能框图 946
38.2.1　功能引脚 948
38.2.2　数据寄存器 948
38.2.3　逻辑控制 948
38.2.4　时钟发生器 949
38.3　WM8978 音频编译码器 950
38.3.1　输入部分 951
38.3.2　输出部分 951
38.3.3　ADC 和 DAC 951
38.3.4　通信接口 951
38.3.5　其他部分 952
38.4　WAV 格式文件 952
38.4.1　RIFF 文件规范 952
38.4.2　WAVE 文件 953

38.4.3 WAVE 文件实例分析 . . . 955
38.5 I2S 初始化结构体详解 . . . 956
38.6 录音与回放实验 . . . 958
38.6.1 硬件设计 . . . 958
38.6.2 软件设计 . . . 959
38.6.3 下载验证 . . . 1003
38.7 MP3 播放器 . . . 1004
38.7.1 MP3 文件结构 . . . 1004
38.7.2 MP3 解码库 . . . 1009
38.7.3 Helix 解码库移植 . . . 1012
38.7.4 MP3 播放器功能实现 . . . 1014
38.7.5 下载验证 . . . 1025
第 39 章 ETH—Lwip 以太网通信 . . . 1026
39.1 互联网模型 . . . 1026
39.2 以太网 . . . 1028
39.2.1 PHY 层 . . . 1028
39.2.2 MAC 子层 . . . 1030
39.3 TCP/IP 协议栈 . . . 1032
39.3.1 为什么需要协议栈 . . . 1033
39.3.2 各网络层的功能 . . . 1033
39.4 以太网外设 (ETH) . . . 1035
39.4.1 SMI 接口 . . . 1036
39.4.2 MII 和 RMII 接口 . . . 1038
39.4.3 MAC 数据包发送和接收 . . . 1041
39.4.4 MAC 过滤 . . . 1043
39.5 PHY：LAN8720A . . . 1043
39.6 LwIP：轻型 TCP/IP 协议栈 . . . 1046
39.7 ETH 初始化结构体详解 . . . 1047
39.8 以太网通信实验：无操作系统 LwIP 移植 . . . 1055
39.8.1 硬件设计 . . . 1055
39.8.2 移植步骤 . . . 1056

39.9 基于 uCOS-III 移植 LwIP 实验 1080
39.9.1 第一步：文件拷贝 1081
39.9.2 第二步：为工程添加文件 1082
39.9.3 第三步：文件修改 1083
39.9.4 下载验证 1104
第 40 章 CAN—通讯实验 1107
40.1 CAN 协议简介 1107
40.1.1 CAN 物理层 1107
40.1.2 协议层 1113
40.2 STM32 的 CAN 外设简介 1123
40.2.1 STM32 的 CAN 架构剖析 1124
40.3 CAN 初始化结构体 1135
40.4 CAN 发送及接收结构体 1138
40.5 CAN 筛选器结构体 1140
40.6 CAN—双机通讯实验 1143
40.6.1 硬件设计 1143
40.6.2 软件设计 1144
40.6.3 下载验证 1156
第 41 章 RS-485 通讯实验 1157
41.1 RS-485 通讯协议简介 1157
41.2 RS-485 的物理层 1157
41.3 RS-485—双机通讯实验 1159
41.3.1 硬件设计 1160
41.3.2 软件设计 1161
41.3.3 下载验证 1171
第 42 章 电源管理—实现低功耗 1173
42.1 STM32 的电源管理简介 1173
42.1.1 电源监控器 1173
42.1.2 STM32 的电源系统 1176
42.1.3 STM32 的功耗模式 1179
42.2 电源管理相关的库函数及命令 1184

42.2.1 配置 PVD 监控功能 1184
42.2.2 WFI 与 WFE 命令 1185
42.2.3 进入停止模式 1185
42.2.4 进入待机模式 1188
42.3 PWR—睡眠模式实验 1190
42.3.1 硬件设计 1190
42.3.2 软件设计 1190
42.3.3 下载验证 1195
42.4 PWR—停止模式实验 1195
42.4.1 硬件设计 1195
42.4.2 软件设计 1195
42.4.3 下载验证 1202
42.5 PWR—待机模式实验 1202
42.5.1 硬件设计 1203
42.5.2 软件设计 1203
42.5.3 下载验证 1208
42.6 PWR—PVD 电源监控实验 1208
42.6.1 硬件设计 1208
42.6.2 软件设计 1210
42.6.3 下载验证 1213
第 43 章 RTC—实时时钟 1215
43.1 RTC 简介 1215
43.2 RTC 功能框图解析 1216
43.2.1 时钟源 1216
43.2.2 预分频器 1217
43.2.3 实时时钟和日历 1217
43.2.4 闹钟 1219
43.2.5 时间戳 1220
43.2.6 入侵检测 1220
43.3 RTC 初始化结构体讲解 1220
43.4 RTC 时间结构体讲解 1222

43.5 RTC 日期结构体讲解 1223
43.6 RTC 闹钟结构体讲解 1223
43.7 RTC—日历实验 1225
43.7.1 硬件设计 1225
43.7.2 软件设计 1225
43.7.3 下载验证 1235
43.8 RTC—闹钟实验 1235
43.8.1 硬件设计 1236
43.8.2 软件设计 1236
43.8.3 代码分析 1237
43.8.4 下载验证 1242
第 44 章 MPU6050 传感器—姿态检测 1243
44.1 姿态检测 1243
44.1.1 基本认识 1243
44.1.2 坐标系 1244
44.1.3 利用陀螺仪检测角度 1246
44.1.4 利用加速度计检测角度 1248
44.1.5 利用磁场检测角度 1250
44.1.6 利用 GPS 检测角度 1250
44.1.7 姿态融合与四元数 1250
44.2 传感器 1251
44.2.1 传感器工作原理 1251
44.2.2 传感器参数 1251
44.2.3 物理量的表示方法 1252
44.3 MPU6050 简介 1253
44.4 MPU6050 的特性参数 1254
44.5 MPU6050—获取原始数据实验 1256
44.5.1 硬件设计 1256
44.5.2 软件设计 1257
44.6 MPU6050—利用 DMP 进行姿态解算 1272
44.6.1 硬件设计 1272

44.6.2 软件设计 1273
44.7 MPU6050—使用第三方上位机 1296
44.7.1 硬件设计 1296
44.7.2 软件设计 1297
第 45 章 DCMI—OV2640 摄像头 1308
45.1 摄像头简介 1308
45.1.1 数字摄像头跟模拟摄像头区别 1308
45.1.2 CCD 与 CMOS 的区别 1309
45.2 OV2640 摄像头 1310
45.2.1 OV2640 传感器简介 1311
45.2.2 OV2640 引脚及功能框图 1311
45.2.3 SCCB 时序 1314
45.2.4 OV2640 的寄存器 1317
45.2.5 像素数据输出时序 1318
45.3 STM32 的 DCMI 接口简介 1320
45.3.1 DCMI 整体框图 1320
45.3.2 DCMI 接口内部结构 1323
45.3.3 同步方式 1324
45.3.4 捕获模式及捕获率 1324
45.4 DCMI 初始化结构体 1325
45.5 DCMI—OV2640 摄像头实验 1327
45.5.1 硬件设计 1327
45.5.2 软件设计 1329
45.5.3 下载验证 1348
第 46 章 DCMI—OV5640 摄像头 1349
46.1 摄像头简介 1349
46.1.1 数字摄像头跟模拟摄像头区别 1349
46.1.2 CCD 与 CMOS 的区别 1350
46.2 OV5640 摄像头 1351
46.2.1 OV5640 传感器简介 1352
46.2.2 OV5640 引脚及功能框图 1352

46.2.3 SCCB 时序 1358
46.2.4 OV5640 的寄存器 1360
46.2.5 像素数据输出时序 1361
46.3 STM32 的 DCMI 接口简介 1362
46.3.1 DCMI 整体框图 1363
46.3.2 DCMI 接口内部结构 1364
46.3.3 同步方式 1366
46.3.4 捕获模式及捕获率 1366
46.4 DCMI 初始化结构体 1366
46.5 DCMI—OV5640 摄像头实验 1369
46.5.1 硬件设计 1369
46.5.2 软件设计 1371
46.5.3 下载验证 1398
第 47 章 MDK 的编译过程及文件类型全解 1399
47.1 编译过程 1399
47.1.1 编译过程简介 1399
47.1.2 具体工程中的编译过程 1400
47.2 程序的组成、存储与运行 1404
47.2.1 CODE、RO、RW、ZI Data 域及堆栈空间 1404
47.2.2 程序的存储与运行 1405
47.3 编译工具链 1407
47.3.1 设置环境变量 1407
47.3.2 armcc、armasm 及 armlink 1410
47.3.3 armar、fromelf 及用户指令 1416
47.4 MDK 工程的文件类型 1418
47.4.1 uvprojx、uvoptx 及 uvguix 工程文件 1419
47.4.2 源文件 1423
47.4.3 Output 目录下生成的文件 1424
47.4.4 Listing 目录下的文件 1459
47.4.5 sct 分散加载文件的格式与应用 1469
47.5 实验：自动分配变量到外部 SRAM 空间 1484

47.5.1 补充关于“__attribute__”关键字的说明 1484
47.5.2 硬件设计 1485
47.5.3 软件设计 1485
47.6 实验：优先使用内部 SRAM 并把堆区分配到外部 SRAM 1497
47.6.1 硬件设计 1497
47.6.2 软件设计 1497
第 48 章 在 SRAM 中调试代码 1514
48.1 在 RAM 中调试代码 1514
48.2 STM32 的启动方式 1515
48.3 内部 FLASH 的启动过程 1517
48.4 实验：在内部 SRAM 中调试代码 1521
48.4.1 硬件设计 1521
48.4.2 软件设计 1523
第 49 章 读写内部 FLASH 1537
49.1 STM32 的内部 FLASH 简介 1537
49.1.1 内部 FLASH 的构成 1538
49.2 对内部 FLASH 的写入过程 1541
49.2.1 解锁 1541
49.2.2 数据操作位数 1541
49.2.3 擦除扇区 1542
49.2.4 写入数据 1542
49.3 查看工程的空间分布 1542
49.3.1 程序 ROM 的加载与执行空间 1545
49.3.2 ROM 空间分布表 1545
49.4 操作内部 FLASH 的库函数 1546
49.4.1 FLASH 解锁、上锁函数 1546
49.4.2 设置操作位数及页擦除 1547
49.4.3 写入数据 1550
49.5 实验：读写内部 FLASH 1553
49.5.1 硬件设计 1553
49.5.2 软件设计 1553

49.5.3 下载验证 1562
第 50 章 设置 FLASH 的读写保护及解除 1563
50.1 选项字节与读写保护 1563
50.1.1 选项字节的内容 1563
50.1.2 RDP 读保护级别 1565
50.2 修改选项字节的过程 1567
50.3 操作选项字节的库函数 1568
50.3.1 选项字节上锁、解锁 1568
50.3.2 设置选项字节函数 1570
50.3.3 写入选项字节 1573
50.4 实验：设置读写保护及解除 1573
50.4.1 硬件设计 1574
50.4.2 软件设计 1574
50.4.3 下载测试 1582
联系我们 1583
rst 投稿示例文件 1584
如何参与项目 1585
项目源码 1585
文档在线阅读与 pdf 1585
常见问题 1586
版权说明 1587

doc_hal_rst

关于本项目

本项目通过 git 开源：

- github 仓库地址：https://github.com/Embedfire-stm32f407-batianhu/ebf_stm32f407_batianhu_hal_tutorial
- gitee 仓库地址：待添加

点击右侧链接可在线阅读本项目文档：《STM32 HAL 库开发实战指南——基于野火 F407 霸天虎开发板 》

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
- 淘宝：https://yehuosm.tmall.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。

- **格式问题**
  - 代码改成使用 rst 的格式
  - 图片的引用需要修改
  - 把图片改成居中形式，去掉缩放
- **主体内容**
  - i2c 章节
  - 其余部分章节的内容转 rst
- 悬赏任务

若想领取悬赏任务，请发送要参与的任务和预计时间到邮箱：embedfire@embedfire.com。

任务名称	奖励	具体内容
i2c-oled	待定	参考 i2c 章节编写驱动 oled 的示例代码和笔记

第 1 章　如何使用本书

1.1 本书的参考资料

本书参考资料为：《STM32F4xx 中文参考手册》和《Cortex®-M4 内核编程手册》，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，本书不可能面面具到，着重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。

1.2 本书的编写风格

本书着重讲解 F407 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—F407。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 F407 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，外设的框图分析的方法还是基本一样的。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。

1.3 本书的配套硬件

本书配套的硬件平台为：野火 STM32-F407 霸天虎开发板，具体见图 *1_1*。学习的时候如果配套该硬件平台做实验，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 STM32—F407 霸天虎硬件资源

1.4 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，M4 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。

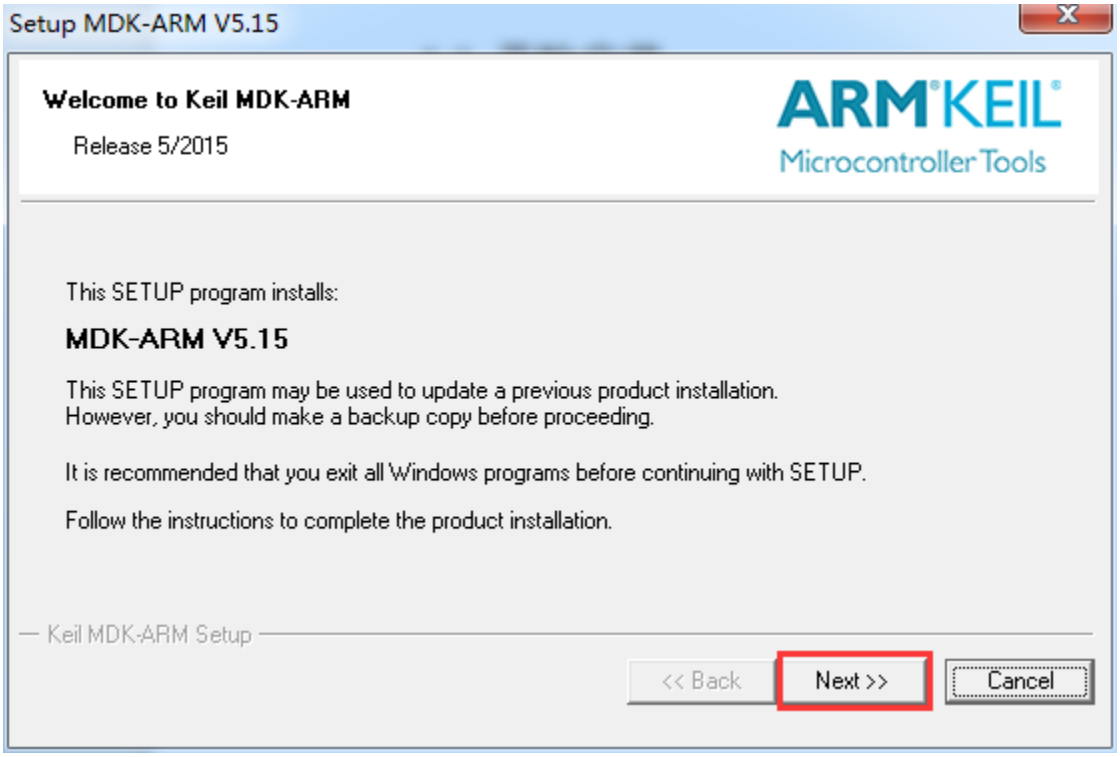
2.2 获取 KEIL5 安装包

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。

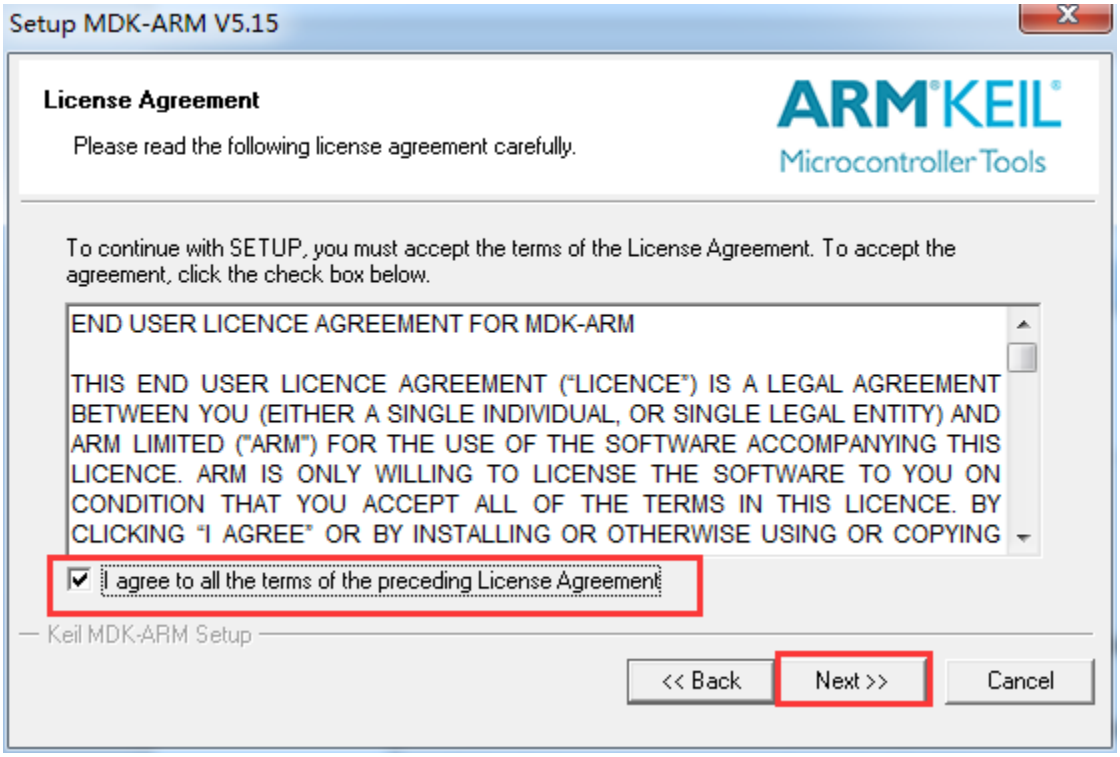


2.3 开始安装 KEIL5

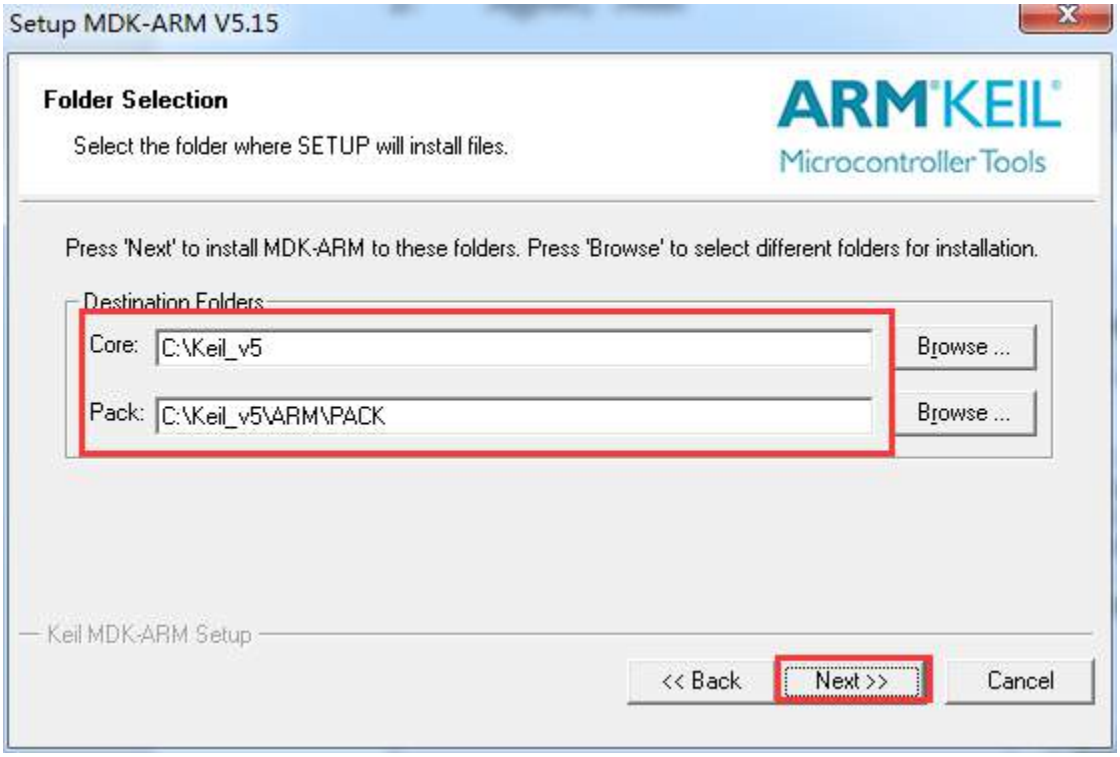
双击 KEIL5 安装包，开始安装，next。


Setup MDK-ARM V5.15
Welcome to Keil MDK-ARM
Release 5/2015
ARM KEIL
Microcontroller Tools
This SETUP program installs:
MDK-ARM V5.15
This SETUP program may be used to update a previous product installation.
However, you should make a backup copy before proceeding.
It is recommended that you exit all Windows programs before continuing with SETUP.
Follow the instructions to complete the product installation.
Keil MDK-ARM Setup
<< Back
Next >>
Cancel

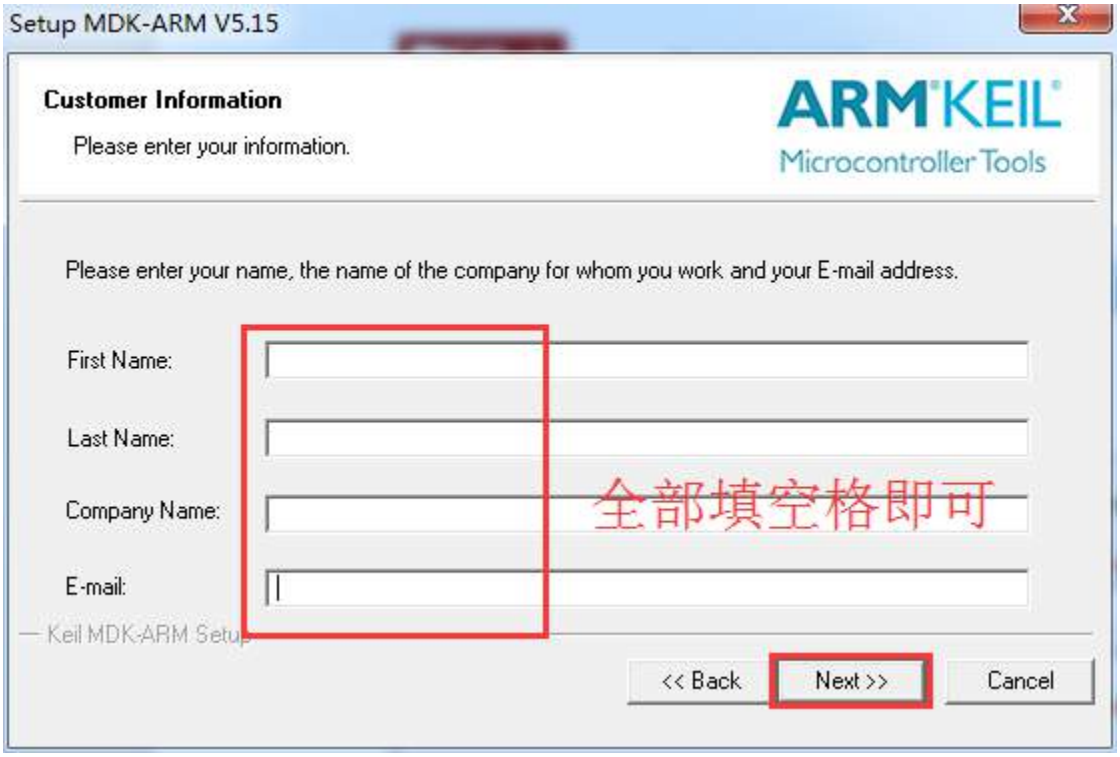
Agree，Nest



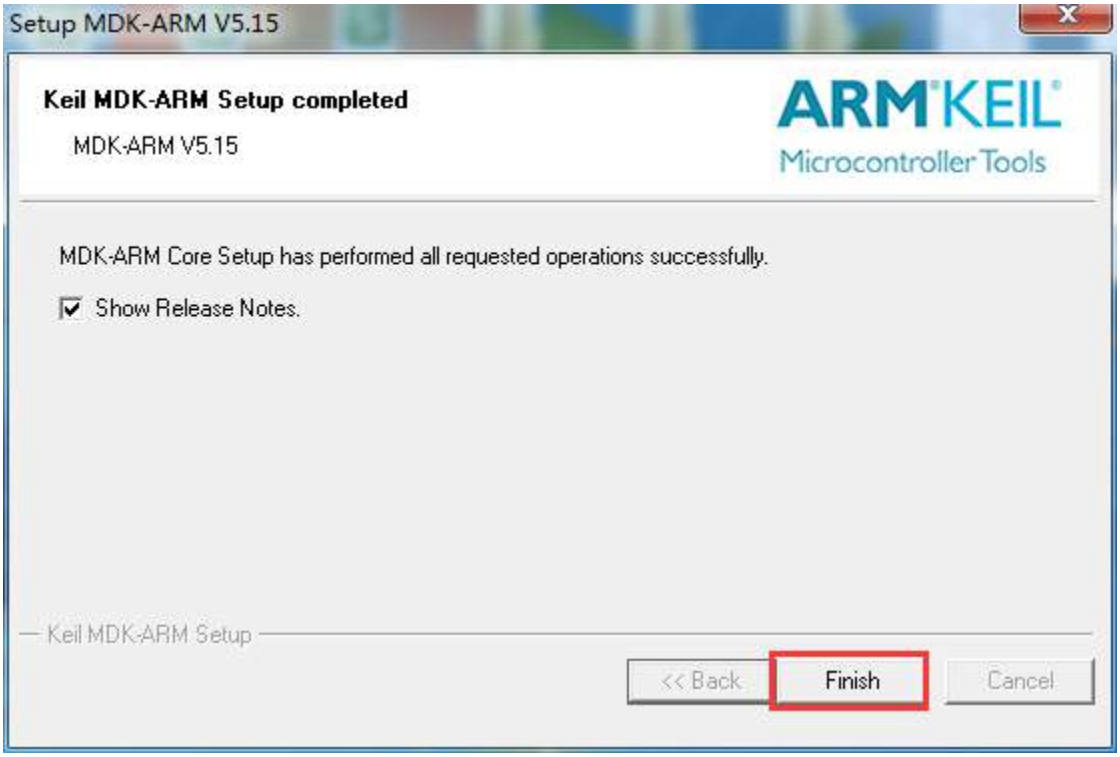
选择安装路径，路径不能带中文，next



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

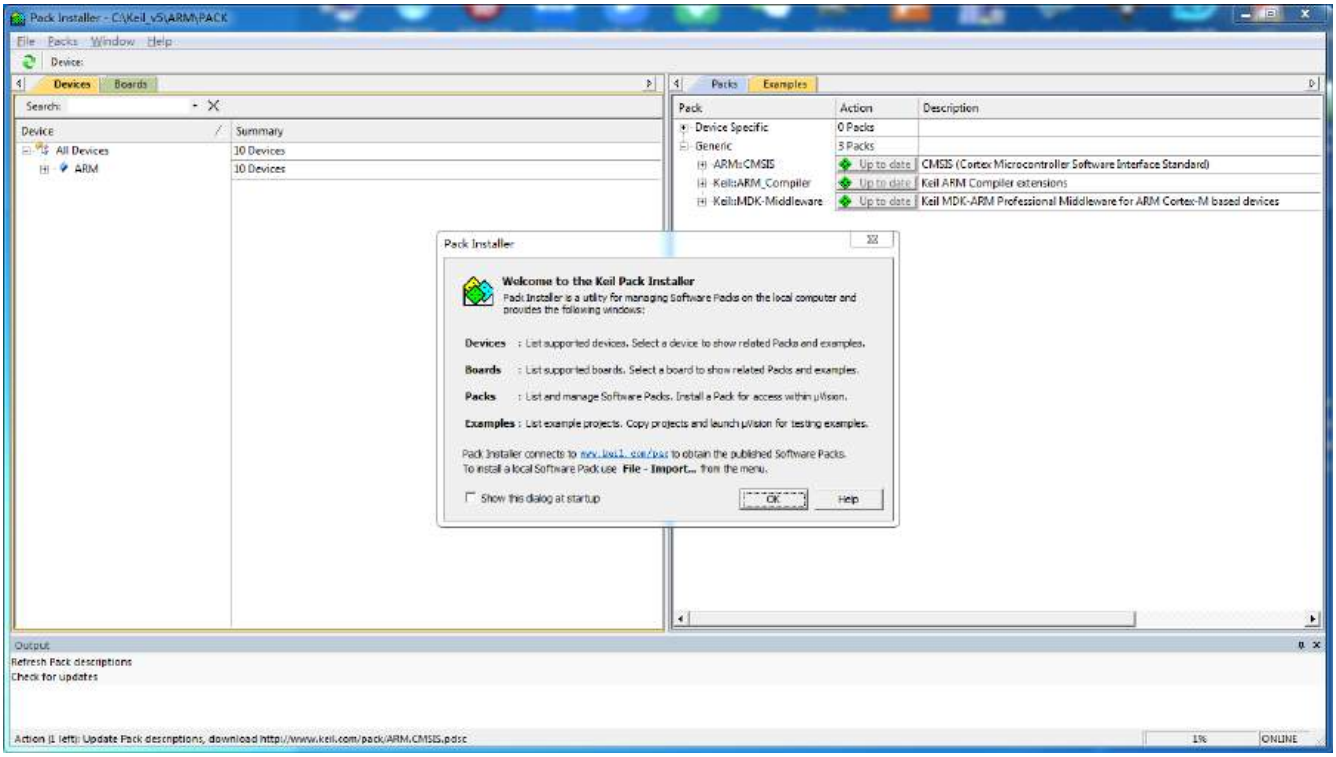
Finish，安装完毕



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。

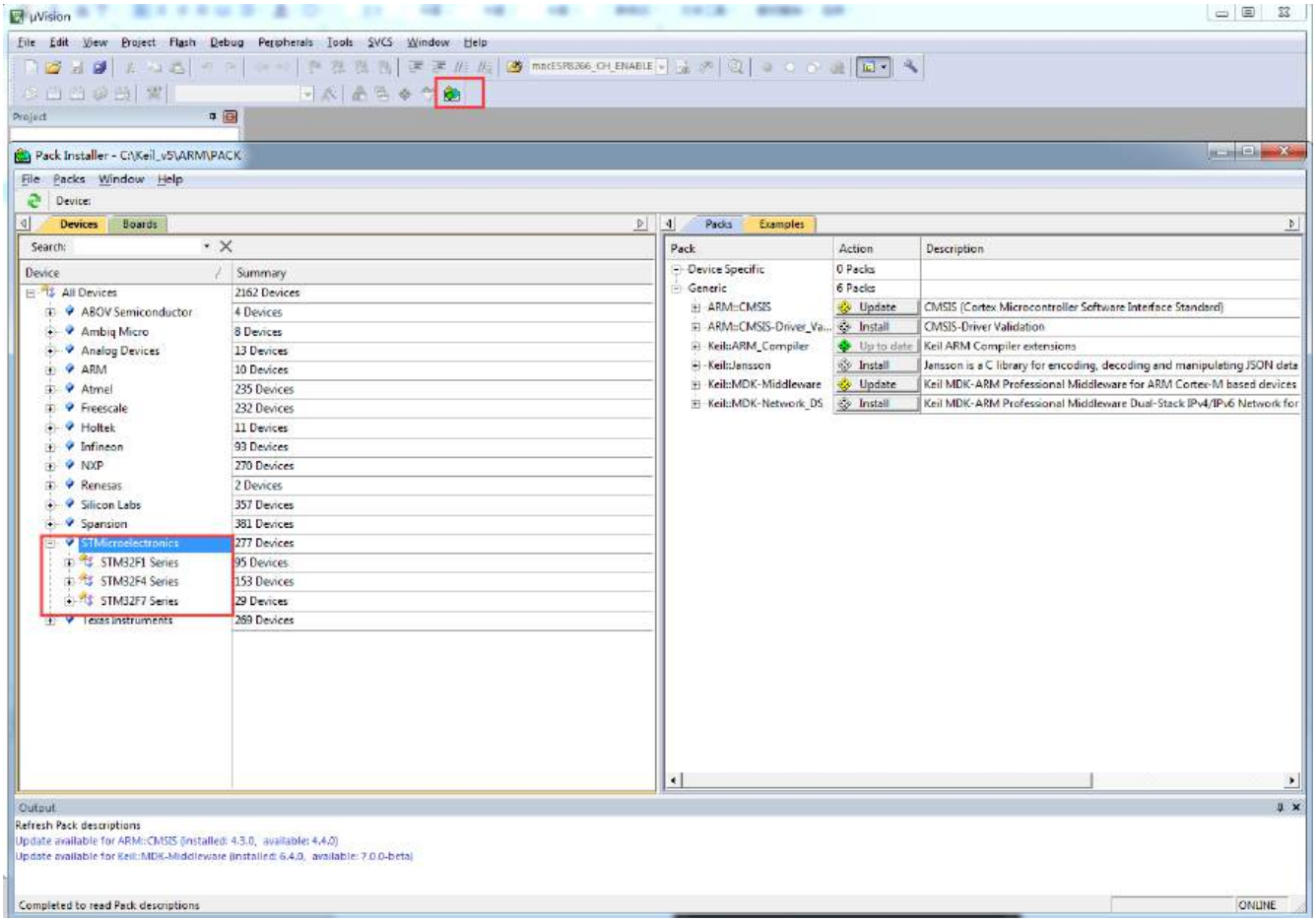
把下面弹出的界面关掉，我们直接去 keil 的官网下载：http://www.keil.com/dd2/pack/，或者直接用我们下载好的包。



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。



到这里 KEIL5 就安装完毕。要特别注意的是 KEIL5 的安装目录不能跟其他版本的 KEIL 冲突，如果冲突了则卸载后重新安装。最后还要记得安装 STM32 芯片包。

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 内核的单片机，常见的 M3、M4 和 M7 都可以完美支持，其外观见图 *3_1*。

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 3-2 仿真器与电脑和开发板连接方式

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

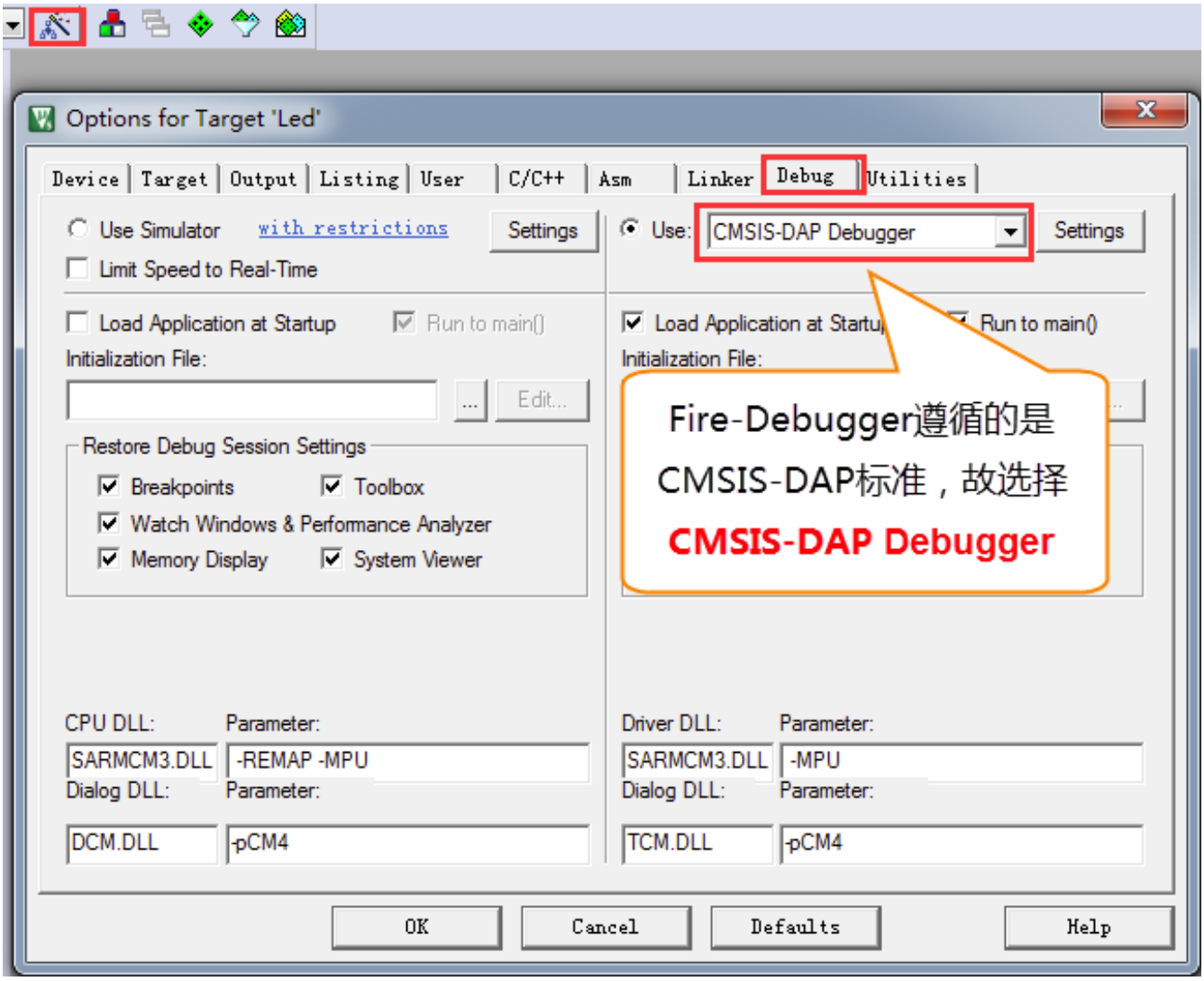


图 3-3 Debug 选择 CMSIS-DAP Debugger

3.3.2 Utilities 选项配置

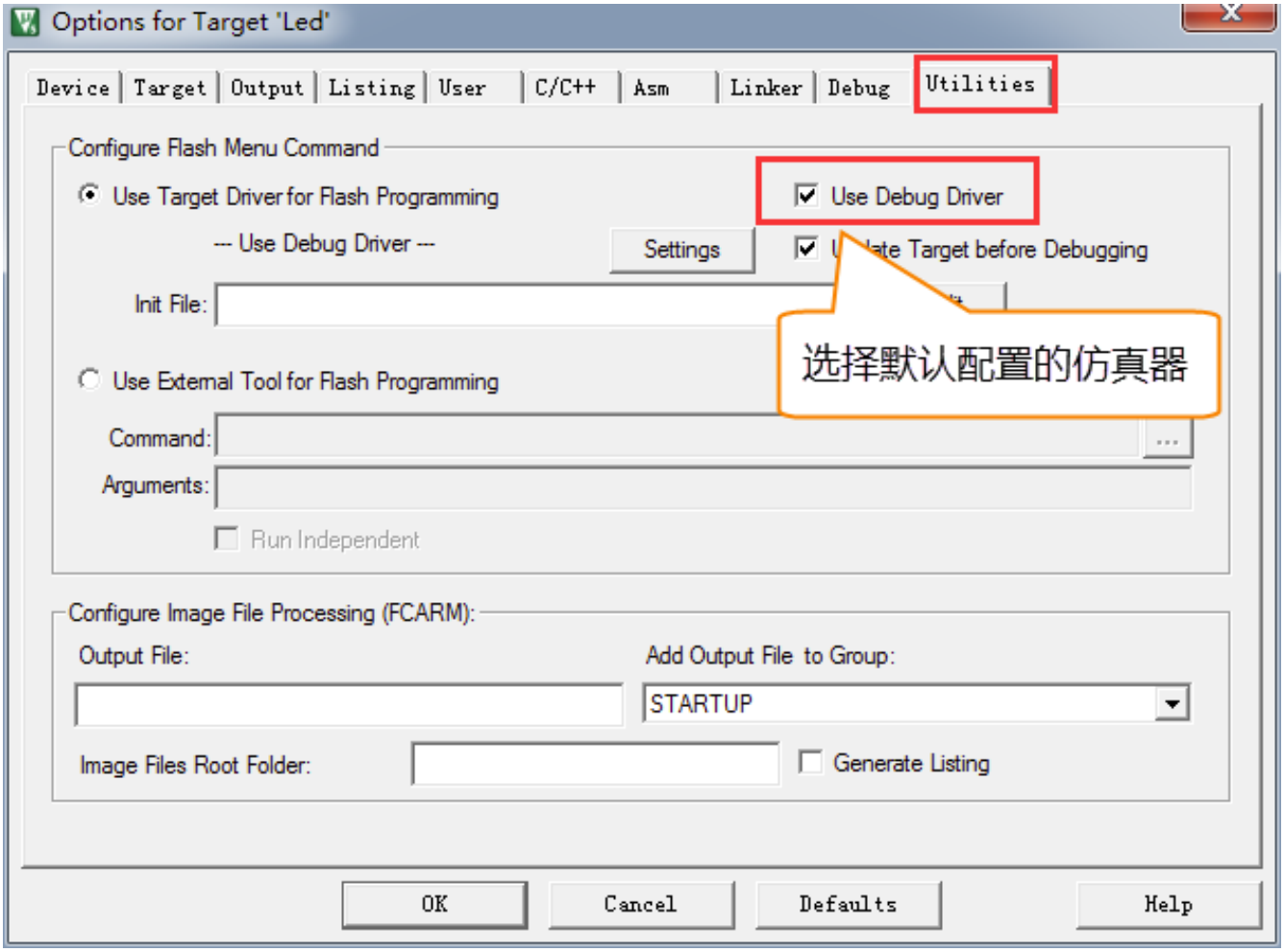


图 3-4 Utilities 选择 Use Debug Driver

3.3.3 Debug Settings 选项配置

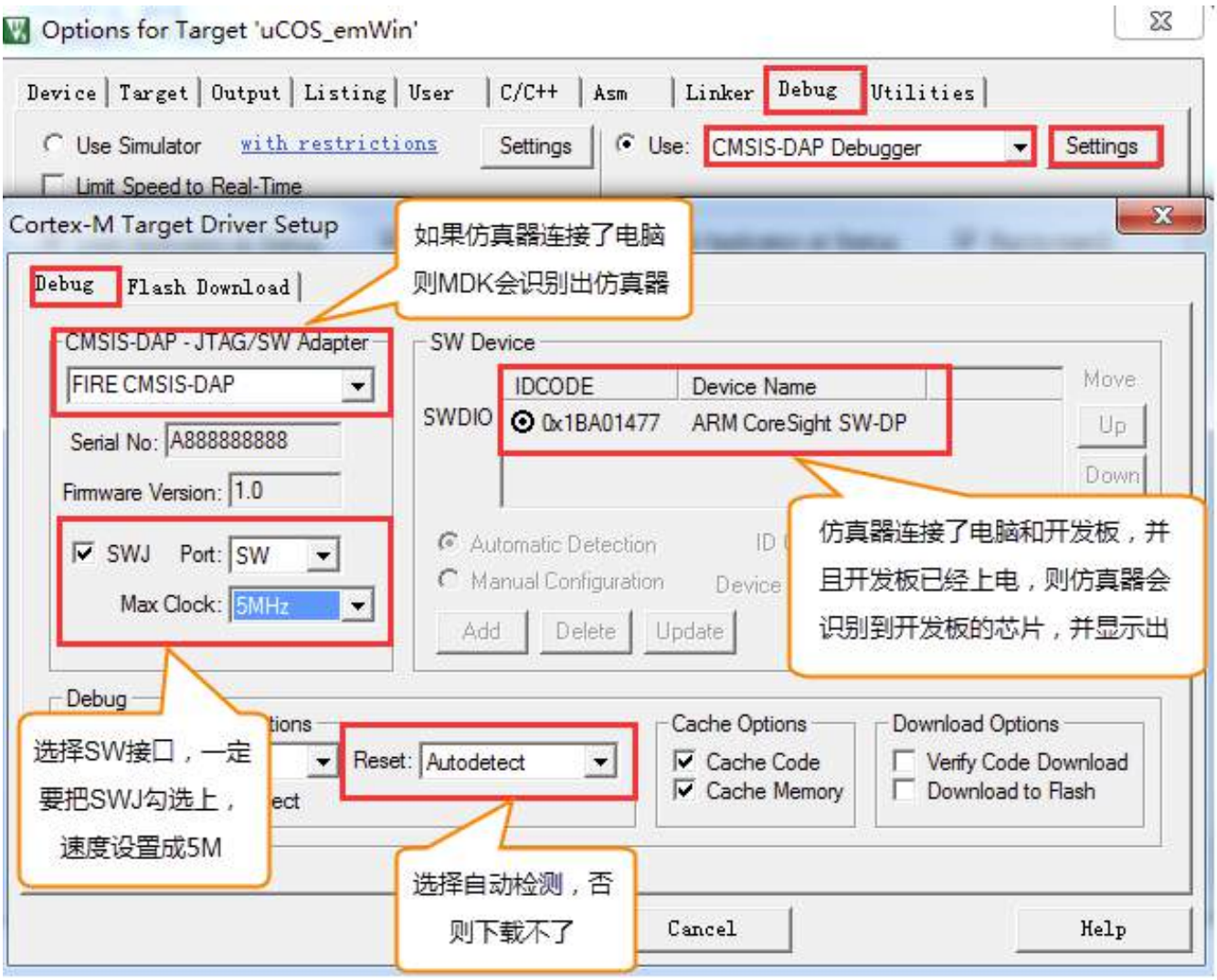


图 3-5 Debug Settings 选项配置

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。

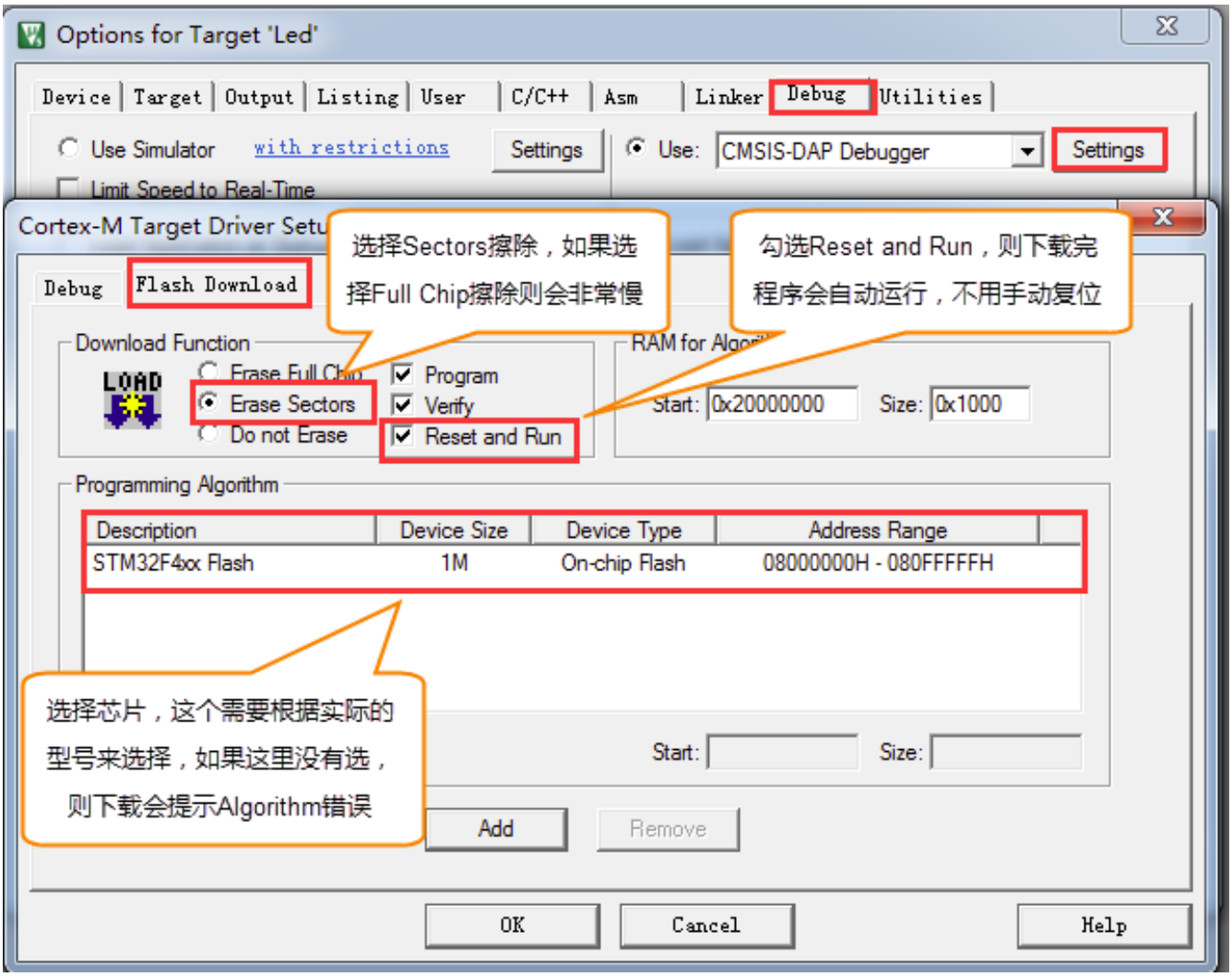


图 3-6 选择目标板

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。



图 3-7 下载程序

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。

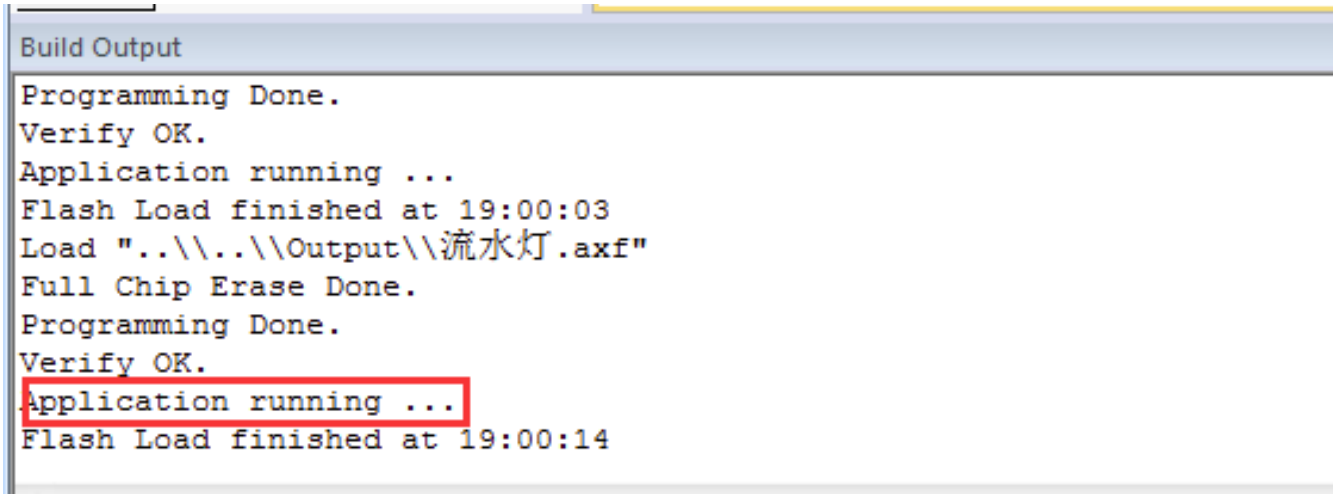


图 3-8 程序运行成功

第 4 章　初识 STM32

本章参考资料：1、《STM8 和 STM32 产品选型手册》2、SetupSTM32CubeMX-4.11.0.exe

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，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	野火 STM32F407 霸天虎
CPU	STM32F439ZIY6S，WLCSP143 封装	STM32F407ZGT6，LQPF144 封装
存储	NOR+SRAM 16MB，FSMC 接口	SRAM 1MB，FSMC 接口
显示	1.84 寸的 AMOLED，RGB 接口，LTDC 驱动	4.5 寸电容屏，8080 接口，FSMC 驱动
陀螺仪	MPU6050，I2C 接口	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 407 开发板上还集成了以太网，音频，CAN，485，232，USB 转串口，蜂鸣器，LED，电容按键等外设资源，可以充分的学习 407 这个芯片。在板子上面，还可以跑系统 FreeRTOS/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。

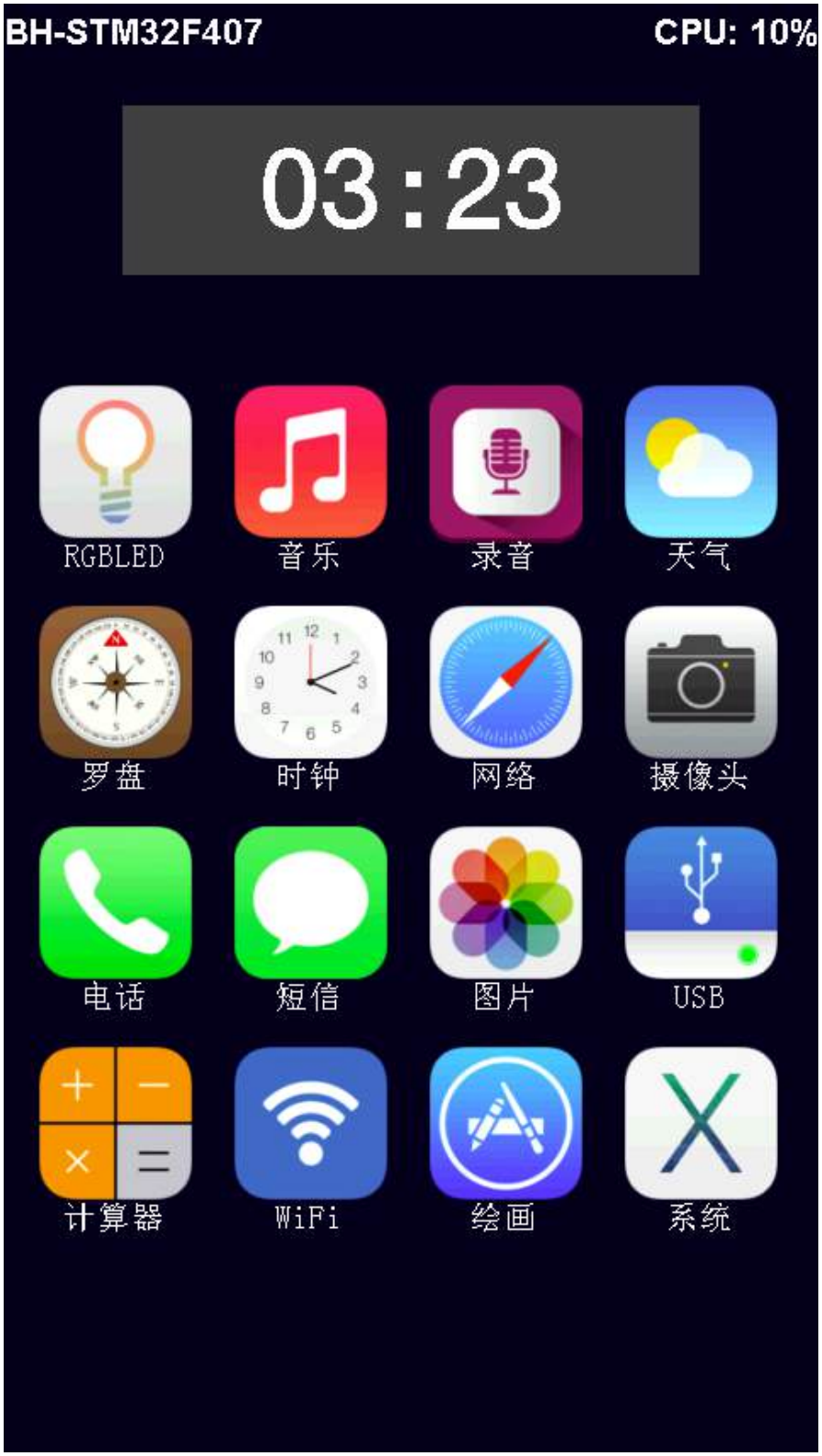

BH-STM32F407
CPU: 10%
03:23
RGBLED
音乐
录音
天气
罗盘
时钟
网络
摄像头
电话
短信
图片
USB
计算器
WiFi
绘画
系统

图 4-2 ucosiii+emwin 做的系统界面（407 开发板的开机界面）

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。小米的四轴飞行器就是使用 STM32F407 做的，如果自己不能完成整个产品，那也可以加入这些科技类的公司负责该产品的某一个部分，技术够硬了，何愁月薪不过万。

淘宝众筹科技类网址：这里面有很多产品都可以用 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 F407 霸天虎用的型号 STM32F407ZGT6 来讲解下 STM32 的命名方法。

表格 4-3 STM32F407ZGT6 命名解释

	STM32F407ZGT6
家族	STM32, 表示 32bit 的 MCU
产品类型	F 表示基础型
具体特性	407 表示高性能且带 DSP 和 FPU
引脚数目	Z 表示 144pin，其他常用的为 C 表示 48，R 表示 64，V 表示 100，Z 表示 144，B 表示 208，N 表示 216
FLASH 大小	G 表示 1024KB，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。

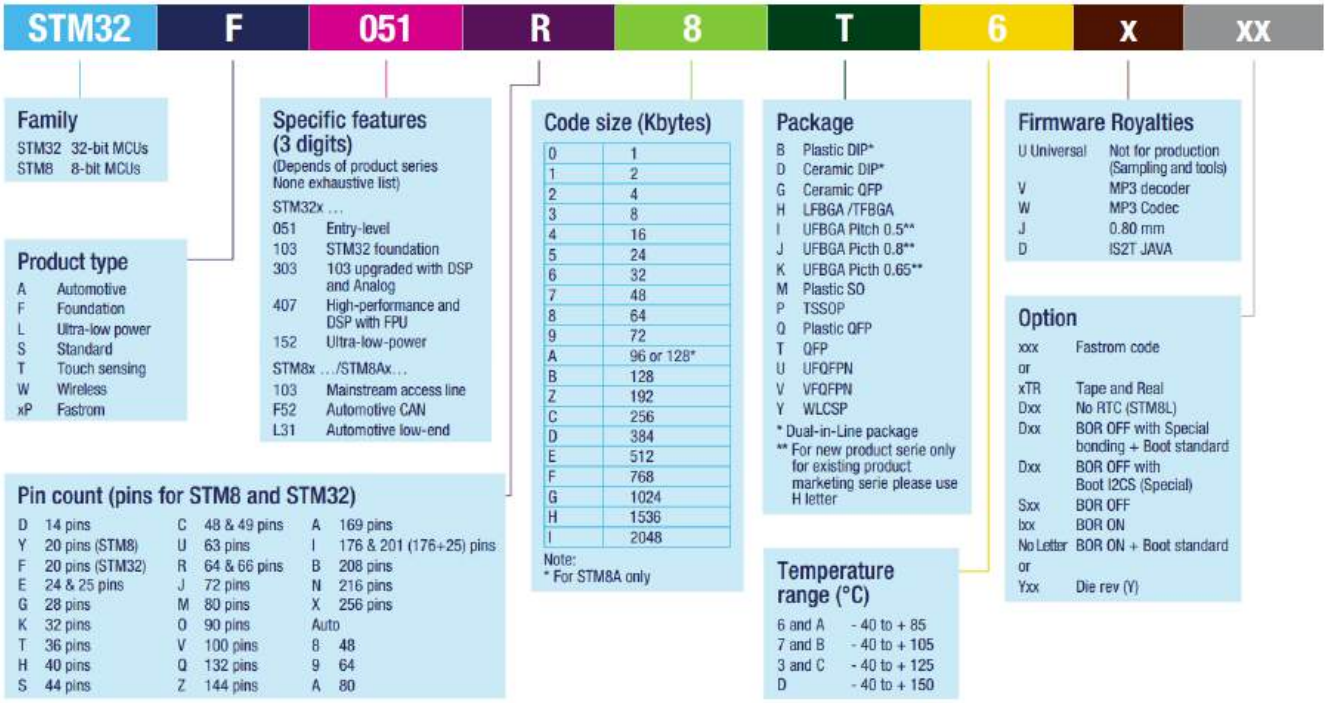


图 4-6 STM8 和 STM32 命名方法，摘自《STM8 和 STM32 选型手册》

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，要做图形界面的则选择 Cortex-M4 内核的 F407 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。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很深，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)	(3)	(4)	(5)	(6) Alternate functions(4)	
LFBGA144	LFBGA100	WLCSP64	LQFP64	LQFP100	LQFP144	Pin name	Type(1)	I / O Level	Main function(3) (after reset)	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 F407 霸天虎使用的 MCU 型号是 STM32F407ZGT6，封装为 LQFP144，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：《STM32F4xx 中文参考手册》、《STM32F4xx 英文数据手册》

学习本章时，配合《STM32F4xx 中文参考手册》“存储器和总线架构”、“嵌入式 FLASH 接口”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

在学习 STM32 编程时，通常有两种编程方式，一种是寄存器编程，另外一种是函数库编程，其中寄存器编程是基础，而函数库编程是在寄存器编程的基础上升级而来的一种易于学习和开发的编程方式，是我们学习 STM32 编程的时候需要重点掌握的一种编程方法。函数库编程对于项目开发固然简单和快速，但是从学习的角度开发，学习寄存器编程的方法我们也不能丢。其实，我们在学习 8 位或者 16 位单片机的时候，居多都是采用寄存器编程，那么这个寄存器编程里面的寄存器是什么东西？它在芯片的什么地方？在探索 STM32 寄存器深层次的问题时，我们从 STM32 表面的外观开始，一层层的往里面深入学习。最后，在学习完本章内容之后，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

我们 F407 霸天虎开发板中使用的芯片是 144pin 的 STM32F407ZGT6，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，STM32F407ZGT6 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，连接到各种传感器上，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）来控制各种传感器工

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，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。

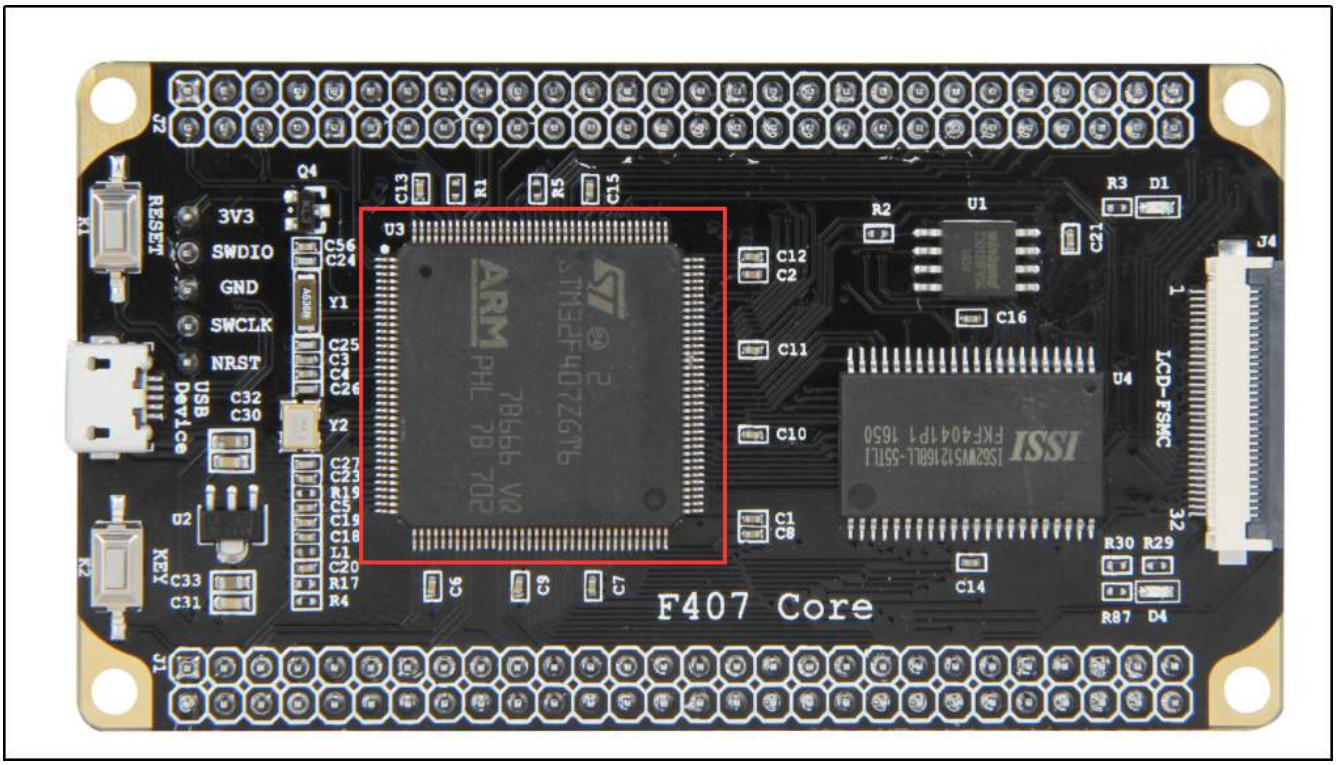


图 5-1 STM32F407ZGT6 实物图 (图中方框部分的芯片)

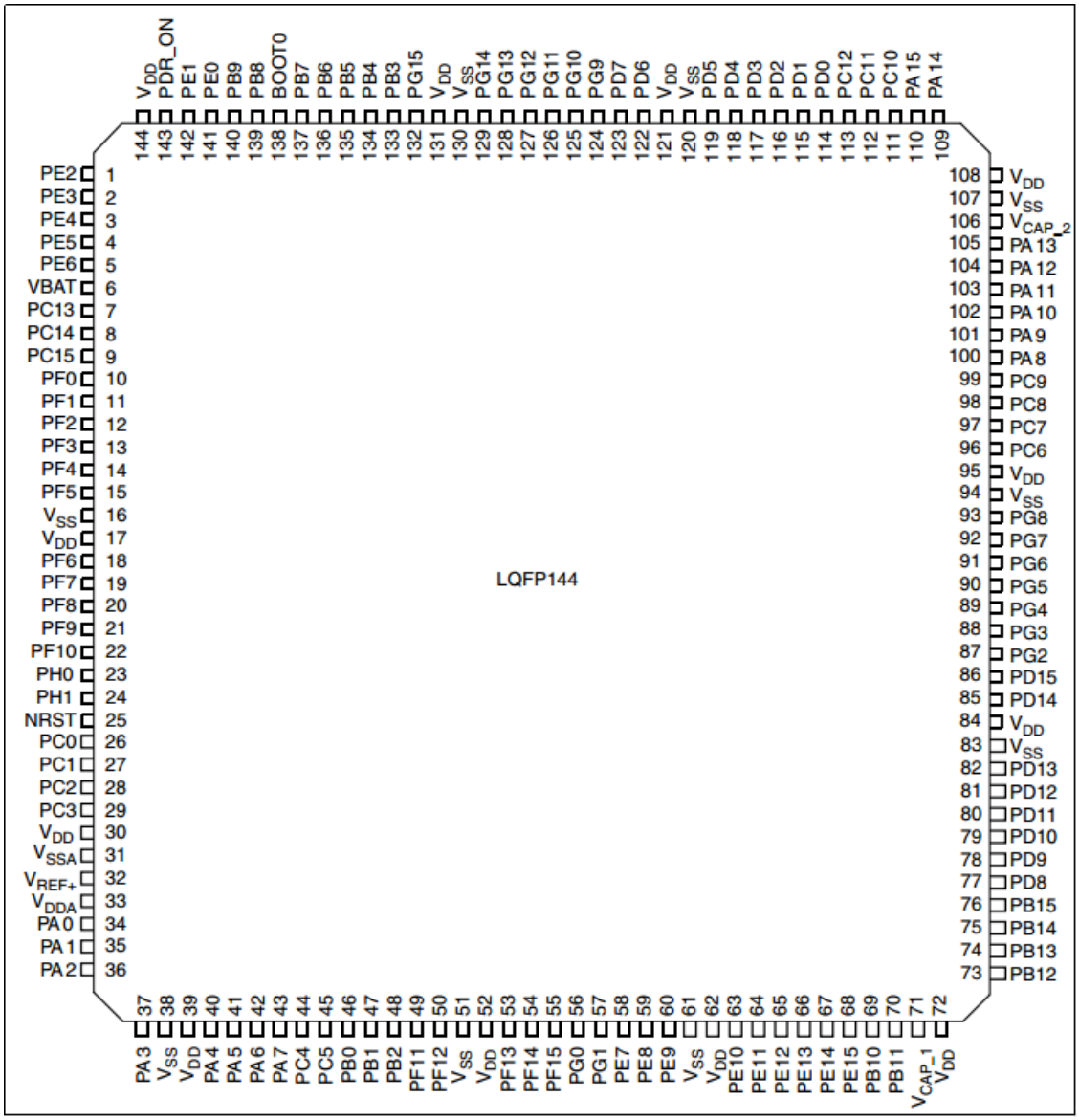


图 5-2 STM32F407ZGT6 正面引脚图

5.3 芯片里面有什么

我们看到的 STM32 芯片已经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。

STM32F407 采用的是 Cortex-M4 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、Freescale，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*。

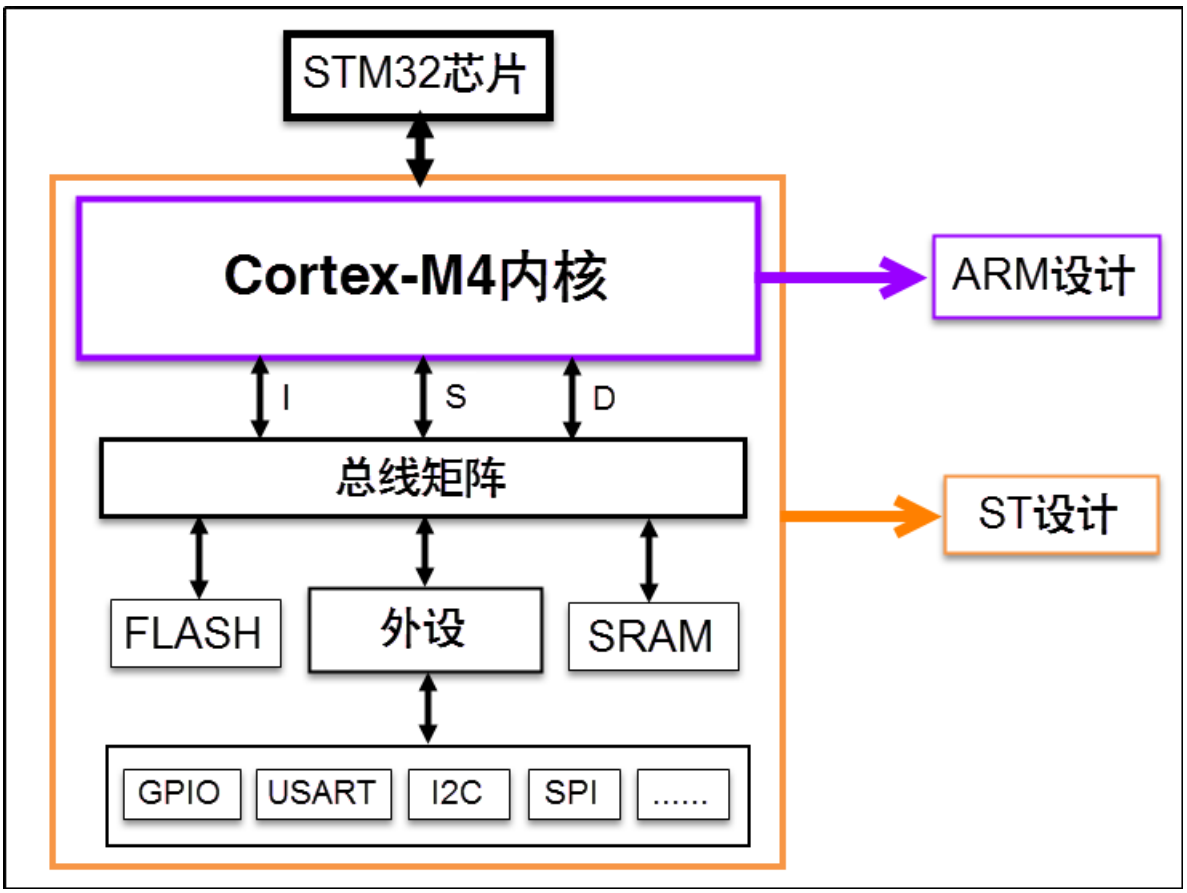
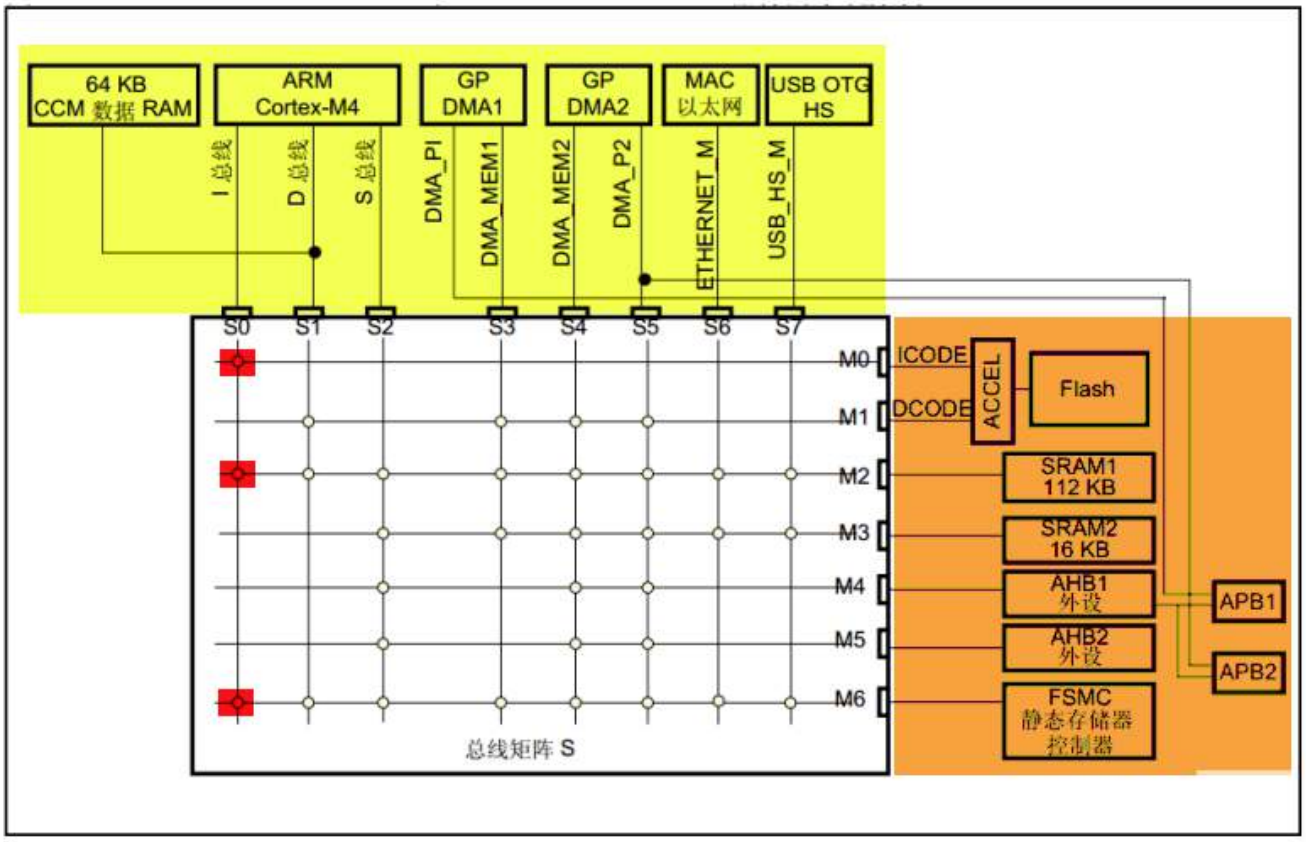


图 5-3 STM32 芯片架构简图

芯片内核和外设之间通过各种总线连接，其中主控总线有 8 条，被控总线有 7 条，具体见图 *5_4*。主控总线通过一个总线矩阵来连接被控总线，总线矩阵用于主控总线之间的访问仲裁管理，仲裁采用循环调度算法。总线之间交叉的时候如果有个圆圈则表示可以通信，没有圆圈则表示不可以

通信。比如 S0：I 总线只有跟 M0、M2 和 M6 这三根被控总线交叉的时候才有圆圈，就表示 S0 只能跟这三根被控总线通信。从功能上来理解，I 总线是指令总线，用来取指，指令指的是编译好的程序指令。我们知道 STM32 有三种启动方式，从 FLASH 启动（包含系统存储器），从内部 SRAM 启动，从外部 RAM 启动，这三种存储器刚好对应的就是 M0、M2 和 M6 这三条总线。



5.4 存储器映射

在图 *5_4* 中，连接被控总线的是 FLASH，RAM 和片上外设，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，操作的也正是这些功能部件。

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，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。

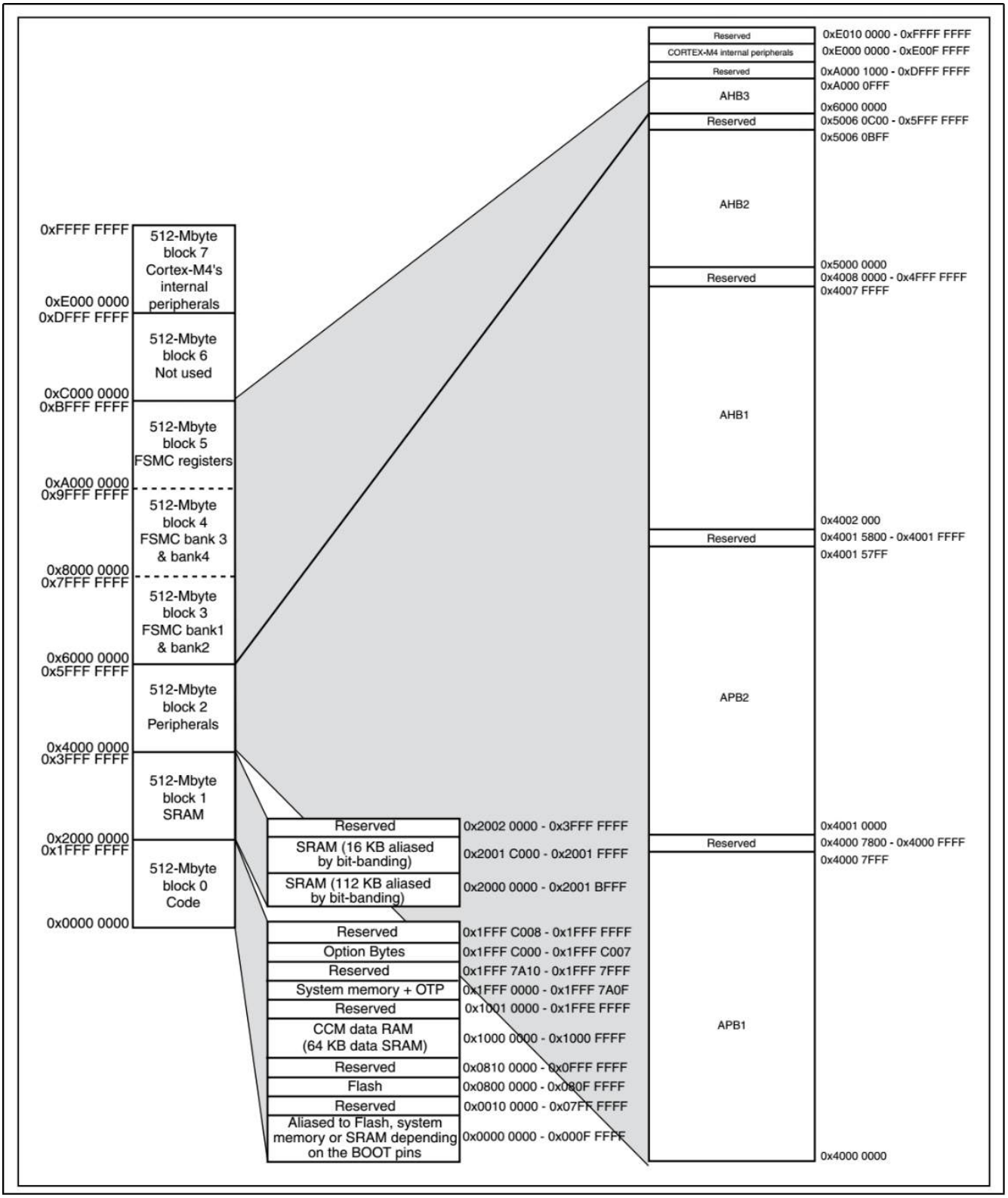

0xFFFF FFFF
512-Mbyte block 7 Cortex-M4's internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
512-Mbyte block 5 FSMC registers
0xA000 0000
0x9FFF FFFF
512-Mbyte block 4 FSMC bank 3 & bank4
0x8000 0000
0x7FFF FFFF
512-Mbyte block 3 FSMC bank1 & bank2
0x6000 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
0x2002 0000 - 0x3FFF FFFF
SRAM (16 KB aliased by bit-banding)
0x2001 C000 - 0x2001 FFFF
SRAM (112 KB aliased by bit-banding)
0x2000 0000 - 0x2001 BFFF
Reserved
0x1FFF C008 - 0x1FFF FFFF
Option Bytes
0x1FFF C000 - 0x1FFF C007
Reserved
0x1FFF 7A10 - 0x1FFF 7FFF
System memory + OTP
0x1FFF 0000 - 0x1FFF 7A0F
Reserved
0x1001 0000 - 0x1FFE FFFF
CCM data RAM (64 KB data SRAM)
0x1000 0000 - 0x1000 FFFF
Reserved
0x0810 0000 - 0x0FFF FFFF
Flash
0x0800 0000 - 0x080F FFFF
Reserved
0x0010 0000 - 0x07FF FFFF
Aliased to Flash, system memory or SRAM depending on the BOOT pins
0x0000 0000 - 0x000F FFFF
Reserved
0xE010 0000 - 0xFFFF FFFF
CORTEX-M4 internal peripherals
0xE000 0000 - 0xE00F FFFF
Reserved
0xA000 1000 - 0xDFFF FFFF
AHB3
0xA000 0FFF
0x6000 0000
Reserved
0x5006 0C00 - 0x5FFF FFFF
AHB2
0x5006 0BFF
0x5000 0000
Reserved
0x4008 0000 - 0x4FFF FFFF
AHB1
0x4007 FFFF
0x4002 000
Reserved
0x4001 5800 - 0x4001 FFFF
APB2
0x4001 57FF
0x4001 0000
Reserved
0x4000 7800 - 0x4000 FFFF
APB1
0x4000 7FFF
0x4000 0000

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-M4 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，F407 系列片内部 FLASH 最大是 1MB，我们使用的 STM32F407ZGT6 的 FLASH 就是 1MB。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 1MB 以上，实乃良心之举。Block0 内部区域的功能划分具体见表 5_2 存储器 Block0 内部区域功能划分

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。F407 内部 SRAM 的大小为 128KB，其中 SRAM1 为 112KB，SRAM2 为 16KB。Block1 内部区域的功能划分具体见表 5-3 存储器 Block1 内部区域功能划分。

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	0x20000000 ~ 0x3FFFFFFF
	SRAM2 16KB	0x2001C000 ~ 0x2001FFFF
	SRAM1 112KB	0x20000000 ~ 0x2001FFFF

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，AHB 分为 AHB1 和 AHB2，具体见表 5-4 存储器 Block2 内部区域功能划分。还有一个 AHB3 包含了 Block3/4/5，AHB3 包含的 3 个 Block 用于扩展外部存储器，如 SRAM，NORFLASH 和 NANDFLASH 等。

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
	预留	0x4000 7800 ~ 0x4000 FFFF
	APB2 总线外设	0x4001 0000 ~ 0x4001 57FF
	预留	0x4001 5800 ~ 0x4001 FFFF
	AHB1 总线外设	0x4002 0000 ~ 0x4007 FFFF
	预留	0x4008 0000 ~ 0x4FFF FFFF
	AHB2 总线外设	0x5000 0000 ~ 0x5006 0BFF
	预留	0x5006 0C00 ~ 0x5FFF 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 GPIOF 端口的输出数据寄存器 ODR 的地址是 0x4002 1414（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 GPIOF 的 16 个 IO 都输出高电平，具体见代码*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*)(0x4002 1414) = 0xFFFF;
```

0x40021414 在我们看来是 GPIOF 端口数据输出寄存器 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 *)0x40021414，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的方式来操作，具体见*代码 5_2* 。

列表 2: 代码 5-2 通过寄存器别名方式访问内存单元

```
// GPIOF 端口全部输出 高电平
#define GPIOF_BASE              0x40021400
#define GPIOF_ODR               (unsigned int*)(GPIOF_BASE+0x14)
*GPIOF_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

```
// GPIOF 端口全部输出 高电平
#define GPIOF_BASE              0x40021400
#define GPIOF_ODR               *(unsigned int*)(GPIOF_BASE+0x14)
GPIOF_ODR = 0xFF;
```

5.5.1 STM32 的外设地址映射

片上外设区分为四条总线，根据外设速度的不同，不同总线挂载着不同的外设，APB 挂载低速外设，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
AHB1	0x4002 0000	0x0001 0000
AHB2	0x5000 0000	0x0001 0000
AHB1	0x6000 0000	已不属于片上外设

表 5-5 总线基地址中的“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 STM32F4xx 外设的边界地址请参考《STM32F4xx 参考手册》的 2.3 小节的存储器映射的表 2：STM32F4xx 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。

表格 5-6 外设 GPIO 基地址

外设名称	外设基地址	相对 AHB1 总线的地址偏移
GPIOA	0x4002 0000	0x0
GPIOB	0x4002 0400	0x0000 0400
GPIOC	0x4002 0800	0x0000 0800
GPIOD	0x4002 0C00	0x0000 0C00
GPIOE	0x4002 1000	0x0000 1000
GPIOF	0x4002 1400	0x0000 1400
GPIOG	0x4002 1800	0x0000 1800
GPIOH	0x4002 1C00	0x0000 1C00

从表 5-6 外设 GPIO 基地址看到，GPIOA 的基址相对于 AHB1 总线的地址偏移为 0，我们应该就可以猜到，AHB1 总线的第一个外设就是 GPIOA。

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 GPIOF 端口为例，来说明 GPIO 都有哪些寄存器，具体见表 5-7 GPIOF 端口的寄存器地址列表。

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
GPIOF_MODER	0x4002 1400	0x00
GPIOF_OTYPER	0x4002 1404	0x04
GPIOF_OSPEEDR	0x4002 1408	0x08
GPIOF_PUPDR	0x4002 140C	0x0C
GPIOF_IDR	0x4002 1410	0x10
GPIOF_ODR	0x4002 1414	0x14
GPIOF_BSRR	0x4002 1418	0x18
GPIOF_LCKR	0x4002 141C	0x1C
GPIOF_AFRL	0x4002 1420	0x20
GPIOF_AFRH	0x4002 1424	0x24

有关外设的寄存器说明可参考《STM32F4xx 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…I)”这段的意思是该寄存器名为“GPIOx_BSRR”其中的“x”可以为 A-I，也就是说这个寄存器说明适用于 GPIOA、GPIOB 至 GPIOI，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 0x18，从参考手册中我们可以查到 GPIOA 外设的基地址为 0x4002 0000，我们就可以算出 GPIOA 的这个 GPIOA_BSRR 寄存器的地址为：0x4002 0000+0x18；同理，由于 GPIOB 的外设基地址为 0x4002 0400，可算出 GPIOB_BSRR 寄存器的地址为：0x4002 0400+0x18。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 ODRx 位执行任何操作；1：对相应 ODRx 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 ODRx 是另一个寄存器的寄存器位，我们只需要知道 ODRx 位为 1 的时候，对应的引脚 x 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)
```

(下页继续)

(续上页)

```
/* 总线基地址 */
#define APB1PERIPH_BASE        PERIPH_BASE
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x00010000)
#define AHB1PERIPH_BASE        (PERIPH_BASE + 0x00020000)
#define AHB2PERIPH_BASE        (PERIPH_BASE + 0x10000000)

/* GPIO 外设基地址 */
#define GPIOA_BASE             (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE             (AHB1PERIPH_BASE + 0x0400)
#define GPIOC_BASE             (AHB1PERIPH_BASE + 0x0800)
#define GPIOD_BASE             (AHB1PERIPH_BASE + 0x0C00)
#define GPIOE_BASE             (AHB1PERIPH_BASE + 0x1000)
#define GPIOF_BASE             (AHB1PERIPH_BASE + 0x1400)
#define GPIOG_BASE             (AHB1PERIPH_BASE + 0x1800)
#define GPIOH_BASE             (AHB1PERIPH_BASE + 0x1C00)

/* 寄存器基地址，以 GPIOF 为例 */
#define GPIOF_MODER              (GPIOF_BASE+0x00)
#define GPIOF_OTYPER             (GPIOF_BASE+0x04)
#define GPIOF_OSPEEDR            (GPIOF_BASE+0x08)
#define GPIOF_PUPDR              (GPIOF_BASE+0x0C)
#define GPIOF_IDR                (GPIOF_BASE+0x10)
#define GPIOF_ODR                (GPIOF_BASE+0x14)
#define GPIOF_BSRR               (GPIOF_BASE+0x18)
#define GPIOF_LCKR               (GPIOF_BASE+0x1C)
#define GPIOF_AFRL               (GPIOF_BASE+0x20)
#define GPIOF_AFRH               (GPIOF_BASE+0x24)
```

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、AHB1 和 AHB2 总线的基地址 APB1PERIPH_BASE、APB2PERIPH_BASE，AHB1PERIPH_BASE 和 AHB2PERIPH_BASE。然后在 AHB1 总线基地址

上加上 GPIO 外设的地址偏移，得到 GPIOA~GPIOH 的外设基地址，最后在外设基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，就可以用指针进行读写操作，具体*代码 5_5*。

列表 5: 代码 5-5 使用指针控制 BSRR 寄存器

```
/* 控制 GPIOF 引脚 6 输出低电平 (BSRR 寄存器的 BR6 置 1) */
*(unsigned int *)GPIOF_BSRR = (0x01<<(16+6));

/* 控制 GPIOF 引脚 6 输出高电平 (BSRR 寄存器的 BS6 置 1) */
*(unsigned int *)GPIOF_BSRR = 0x01<<6;

unsigned int temp;
/* 控制 GPIOF 端口所有引脚的电平 (读 IDR 寄存器) */
temp = *(unsigned int *)GPIOF_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~GPIOH 都各有一组功能相同的寄存器，如 GPIOA_MODER/GPIOB_MODER/GPIOC_MODER 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 */

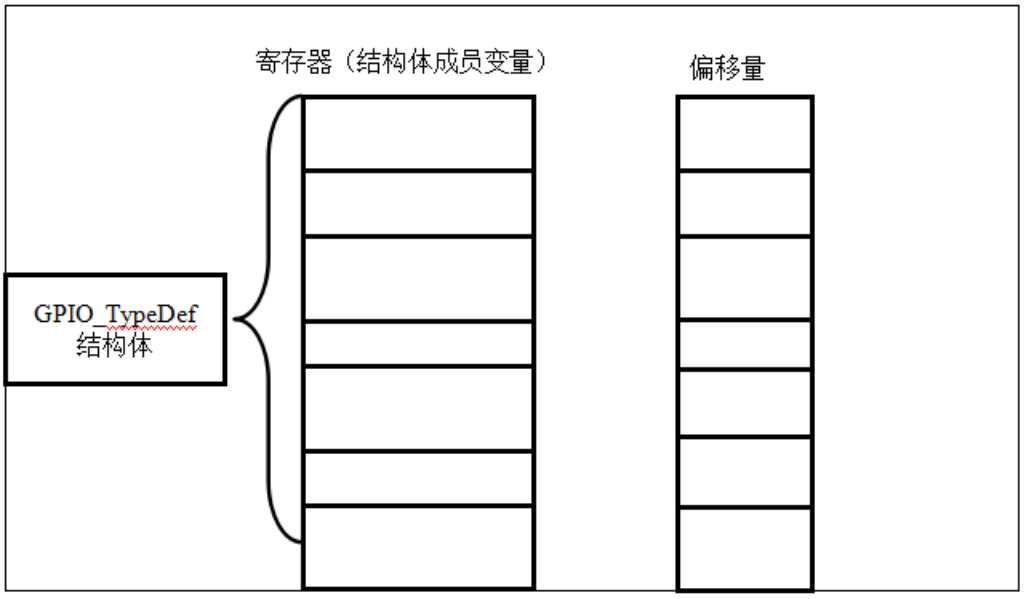
/* GPIO 寄存器列表 */
```

(下页继续)

(续上页)

```
typedef struct {
 uint32_t MODER;     /*GPIO 模式寄存器                地址偏移：0x00      */
 uint32_t OTYPER;    /*GPIO 输出类型寄存器            地址偏移：0x04      */
 uint32_t OSPEEDR;   /*GPIO 输出速度寄存器            地址偏移：0x08      */
 uint32_t PUPDR;     /*GPIO 上拉/下拉寄存器           地址偏移：0x0C      */
 uint32_t IDR;       /*GPIO 输入数据寄存器            地址偏移：0x10      */
 uint32_t ODR;       /*GPIO 输出数据寄存器            地址偏移：0x14      */
 uint16_t BSRRL;     /*GPIO 置位/复位寄存器低 16 位部分 地址偏移：0x18     */
 uint16_t BSRRH;     /*GPIO 置位/复位寄存器高 16 位部分 地址偏移：0x1A     */
 uint32_t LCKR;      /*GPIO 配置锁定寄存器            地址偏移：0x1C      */
 uint32_t AFR[2];    /*GPIO 复用功能配置寄存器        地址偏移：0x20-0x24  */
} 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 8 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，假如我们定义一个 GPIO_TypeDef 类型的结构体，且结构体的首地址为 0x4002 1400（这也是第一个成员变量 MODER 的地址），那么结构体中第二个成员变量 OTYPER 的地址即为

0x4002 1400 +0x04，加上的这个 0x04，正是代表 MODER 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出，其中的 BSRR 寄存器分成了低 16 位 BSRRL 和高 16 位 BSRRH，BSRRL 置 1 引脚输出高电平，BSRRH 置 1 引脚输出低电平，这里分开只是为了方便操作。

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了，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 = GPIOF_BASE;            //把指针地址设置为宏 GPIOF_BASE 地址
GPIOx->BSRRL = 0xFFFF;         //通过指针访问并修改 GPIOF_BSRRL 寄存器
GPIOx->MODER = 0xFFFFFFFF;      //修改 GPIOF_MODER 寄存器
GPIOx->OTYPER =0xFFFFFFFF;      //修改 GPIOF_OTYPER 寄存器

uint32_t temp;
temp = GPIOx->IDR;            //读取 GPIOF_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 GPIOF_BASE(0x4002 1400)，使地址确定下来，然后根据 C 语言访问结构体的语法，用 GPIOx->BSRRL、GPIOx->MODER 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)
```

(下页继续)

(续上页)

```
#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 GPIOF 端口的寄存器 */
GPIOF->BSRRL = 0xFFFF;          //通过指针访问并修改 GPIOF_BSRRL 寄存器
GPIOF->MODER = 0xFFFFFFFF;      //修改 GPIOF_MODER 寄存器
GPIOF->OTYPER =0xFFFFFFFF;      //修改 GPIOF_OTYPER 寄存器

uint32_t temp;
temp = GPIOF->IDR;              //读取 GPIOF_IDR 寄存器的值到变量 temp 中

/* 访问 GPIOA 端口的寄存器 */
GPIOA->BSRRL = 0xFFFF;          //通过指针访问并修改 GPIOA_BSRRL 寄存器
GPIOA->MODER = 0xFFFFFFFF;      //修改 GPIOA_MODER 寄存器
GPIOA->OTYPER =0xFFFFFFFF;      //修改 GPIOA_OTYPER 寄存器

uint32_t temp;
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 把变量的某位清零

此处我们以变量 a 代表寄存器，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，方法见*代码清单 5_1*。

列表 9: 代码清单 5-1 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);

//括号中的 1 左移两位，(1<<2) 得二进制数：0000 0100 b

//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

// a 的 bit2 位被被零，而其它位不变。
```

5.5.3.2 把变量的某几个连续位清零

由于寄存器中有时会有连续几个寄存器位用于控制某个功能，现假设我们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_2。

列表 10: 代码清单 5-2 对某几个连续位清零

```
//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

//要对第 1 组的 bit2、bit3 清零

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

//括号中的 3 左移两位，(3<<2*1) 得二进制数：0000 1100 b

//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

// a 的第 1 组的 bit2、bit3 被清零，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

//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
```

(下页继续)

(续上页)

```
//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“1111b”

//例如对第 2 组 bit4、bit5 清零

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

5.5.3.3 对变量的某几位进行赋值。

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 11: 代码清单 5-3 对某几位进行赋值

```
//a = 1000 0011 b

//此时对清零后的第 2 组 bit4、bit5 设置成二进制数“01 b ”

a |= (1<<2*2);

//a = 1001 0011 b，成功设置了第 2 组的值，其它组不变
```

5.5.3.4 对变量的某位取反

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，这可以直接用如下操作，其它位不变，见代码清单 5_4。

列表 12: 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
```

(下页继续)

(续上页)

```
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，如果有更高的版本可使用高版本。

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

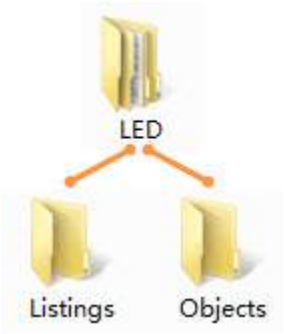
6.1 新建工程

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”，工程名可随意命名。然后在该目录下新建 2 个文件夹 Listings 和 Objects，在使用 KEIL5 新建工程的时候，Listings 和 Objects 会自动生成，具体见表 6-1 工程目录文件夹清单。使用 KEIL5 以下版本则要手动新建。

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



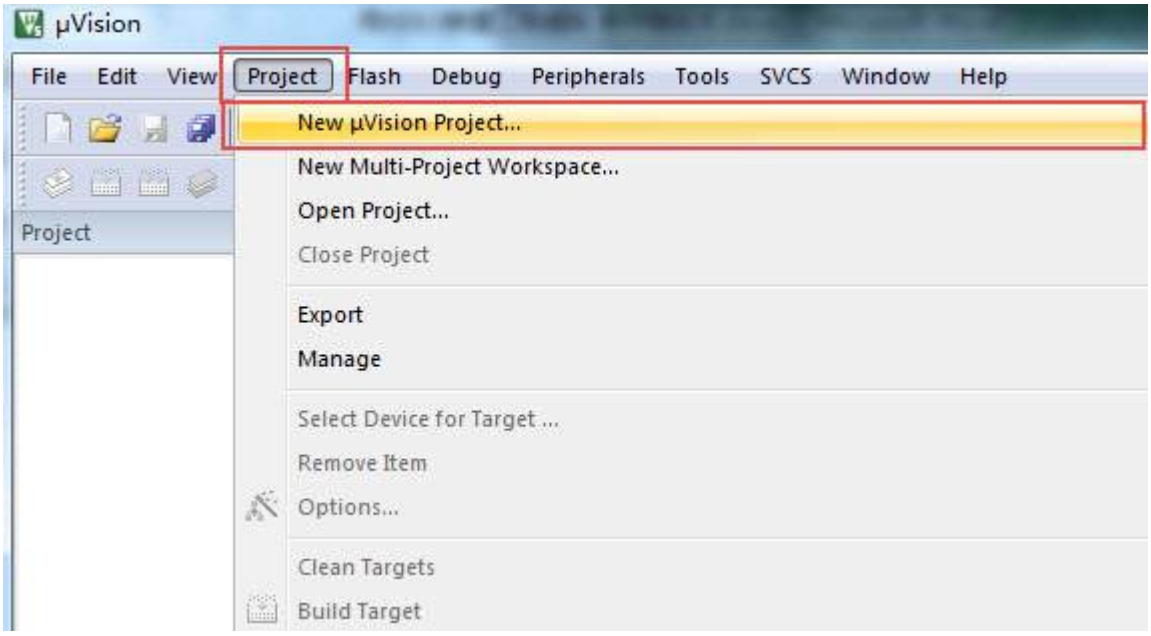
在本地新建好工程文件夹后，然后在文件夹里面新建一些文件，在开始编程之前，这些文件都为空。

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

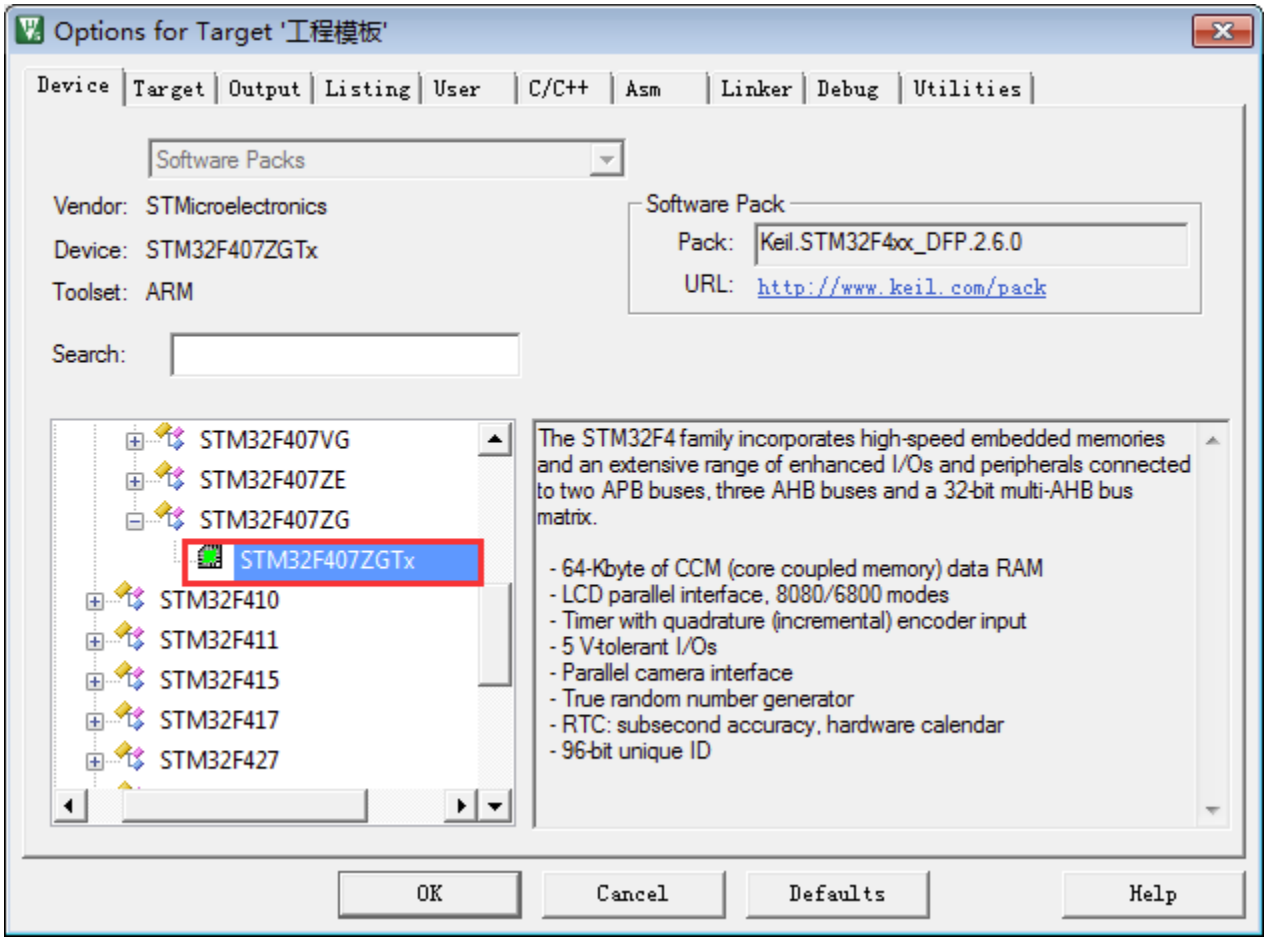
6.2 开始新建工程

打开 KEIL5 软件，新建一个工程，具体见图 *6_2*。工程名根据喜好命名，我这里取 LED-REG（REG 是寄存器的英文 register 的缩写），直接保存在 LED 文件夹下。



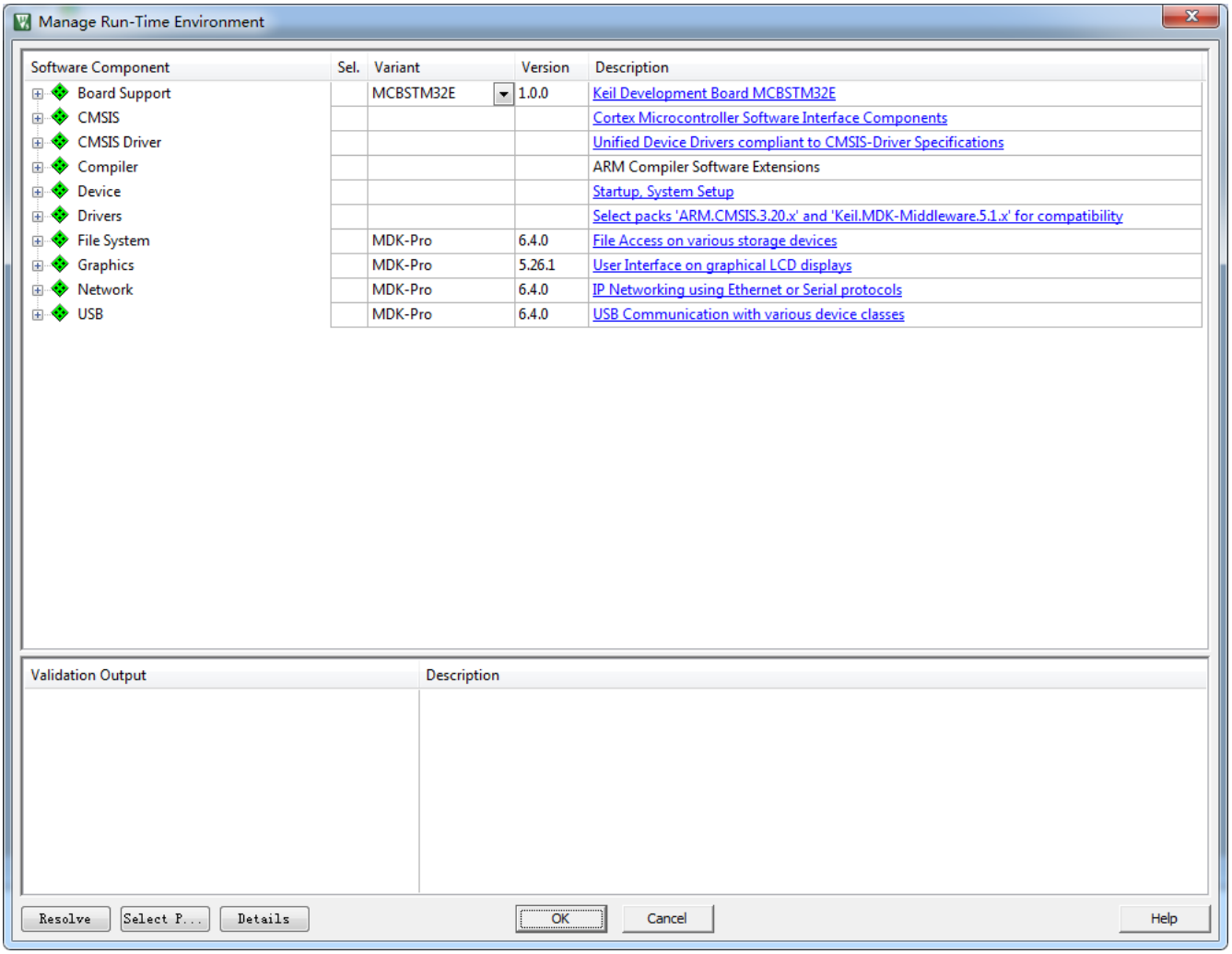
6.2.1 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，F407-霸天虎选 STM32F407ZGTx，具体见图 *6_3*。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 STM32 芯片包，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》这一章的安装 STM32 芯片包小节。



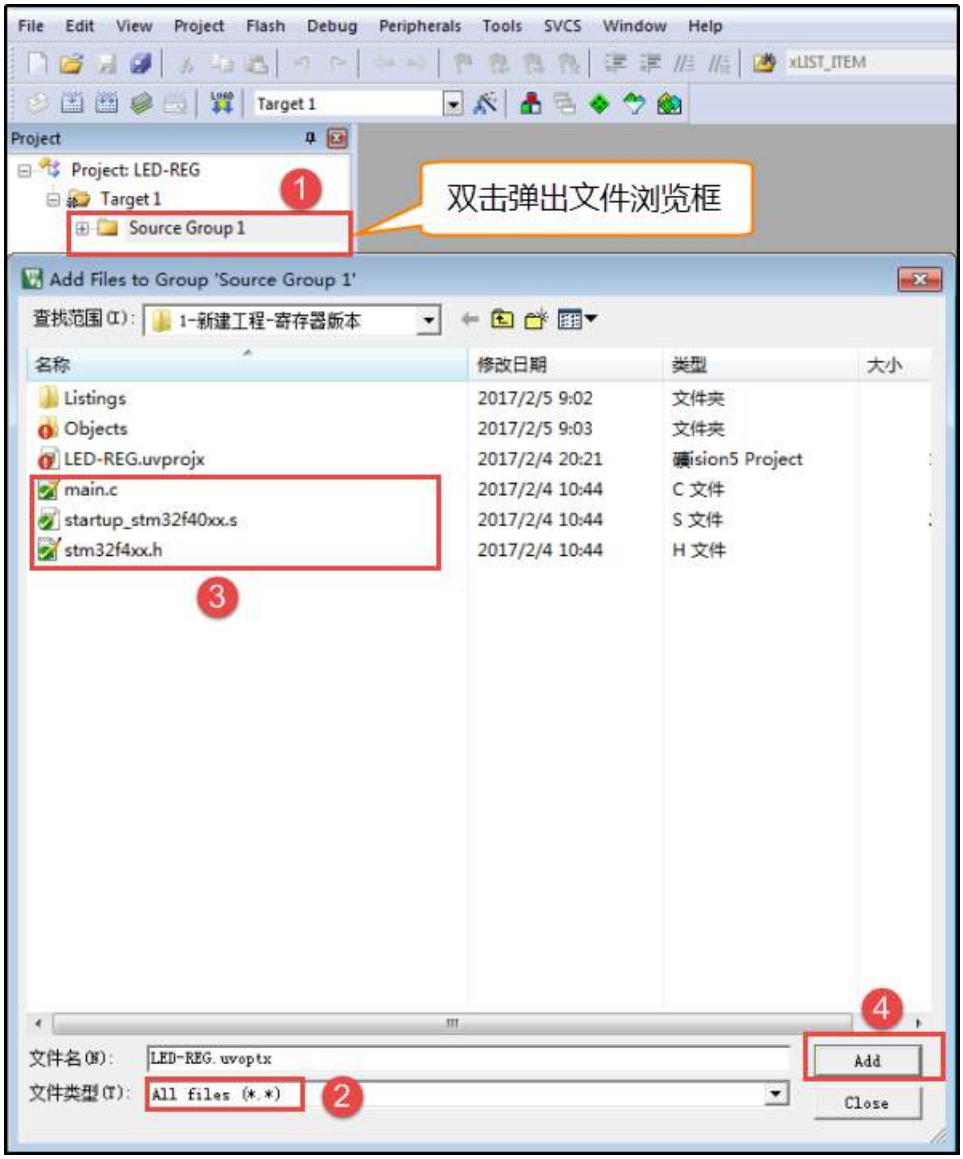
6.2.2 在线添加库文件

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，图 *6_4*。



6.2.3 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，具体见图 *6_5*，然后选择文件 Add 即可。



现在我们对添加的三个文件简单说明如下：

1.startup_stm32f40xx.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，这个文件从函数库里面拷贝而来，由官方提供。文件在这个目录：F4 函数库\STM32Cube_FW_F4_V1.24.0\Drivers\CMSIS\Device\ST\stm32f4xx\Source \Templates\arm\startup_stm32f40xx.s

2.stm32f4xx.h

用户手动新建，用于存放寄存器映射的代码，暂时为空。

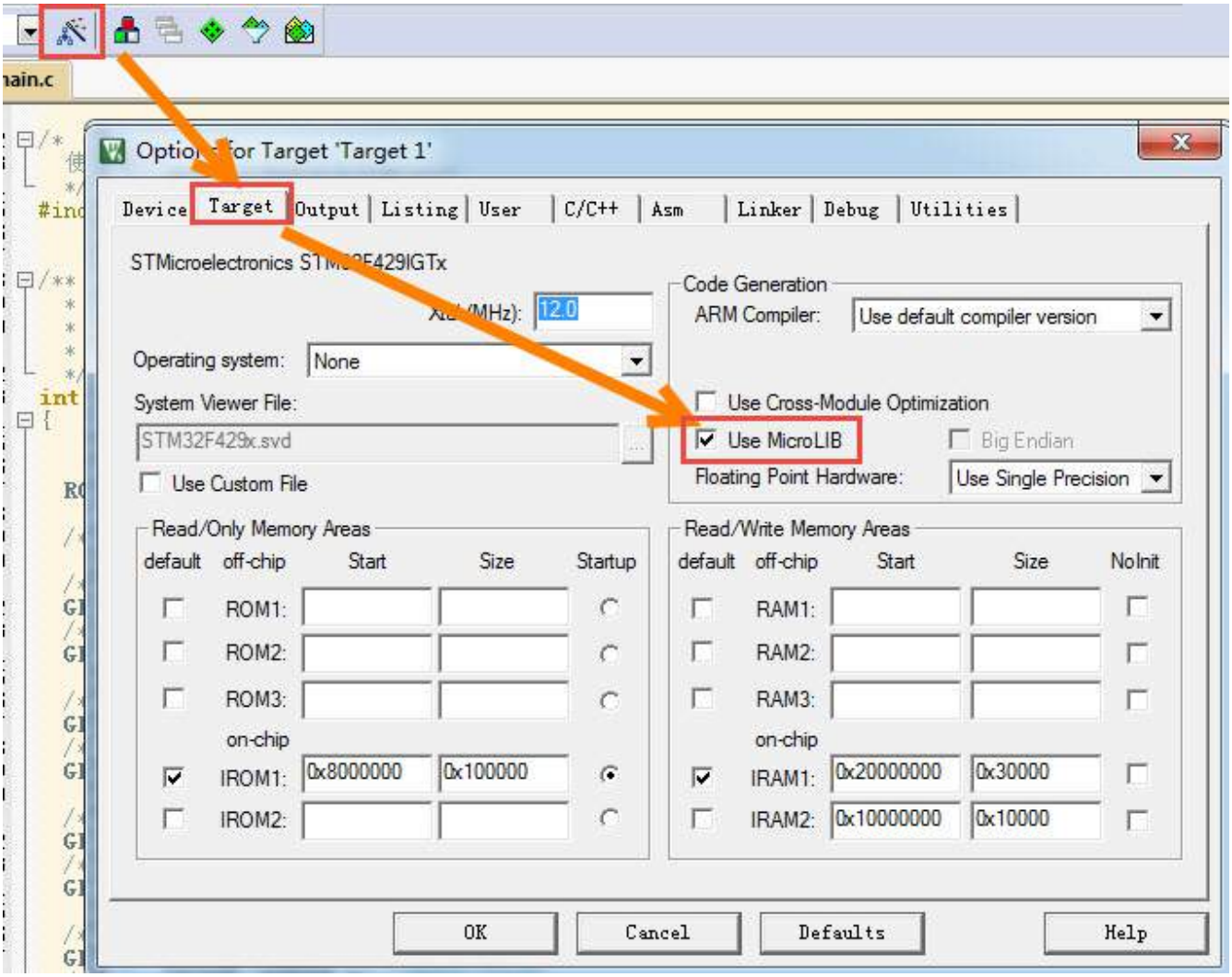
3.main.c

用户手动新建，用于存放 main 函数，暂时为空。

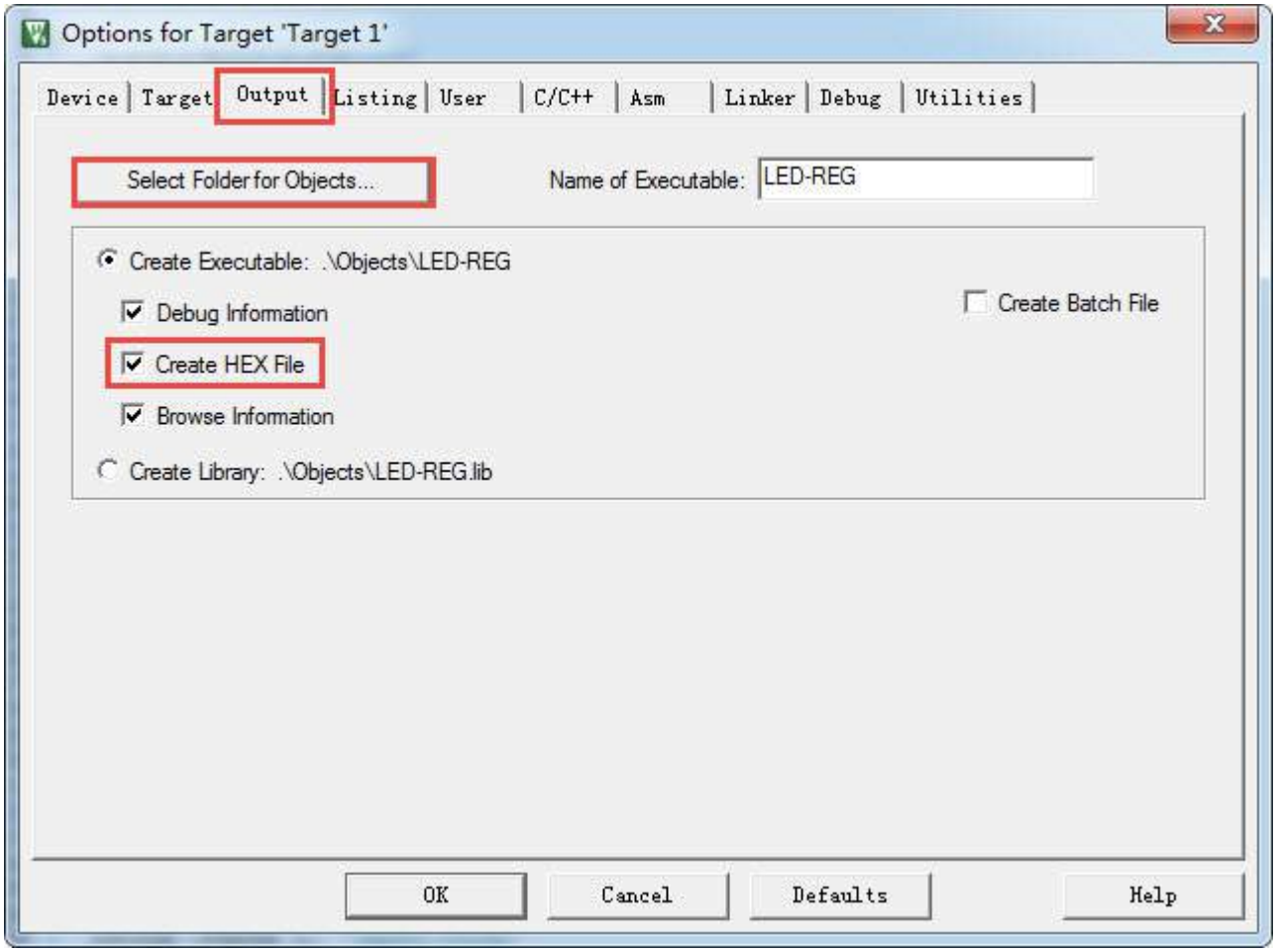
6.2.4 配置魔术棒选项卡

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。

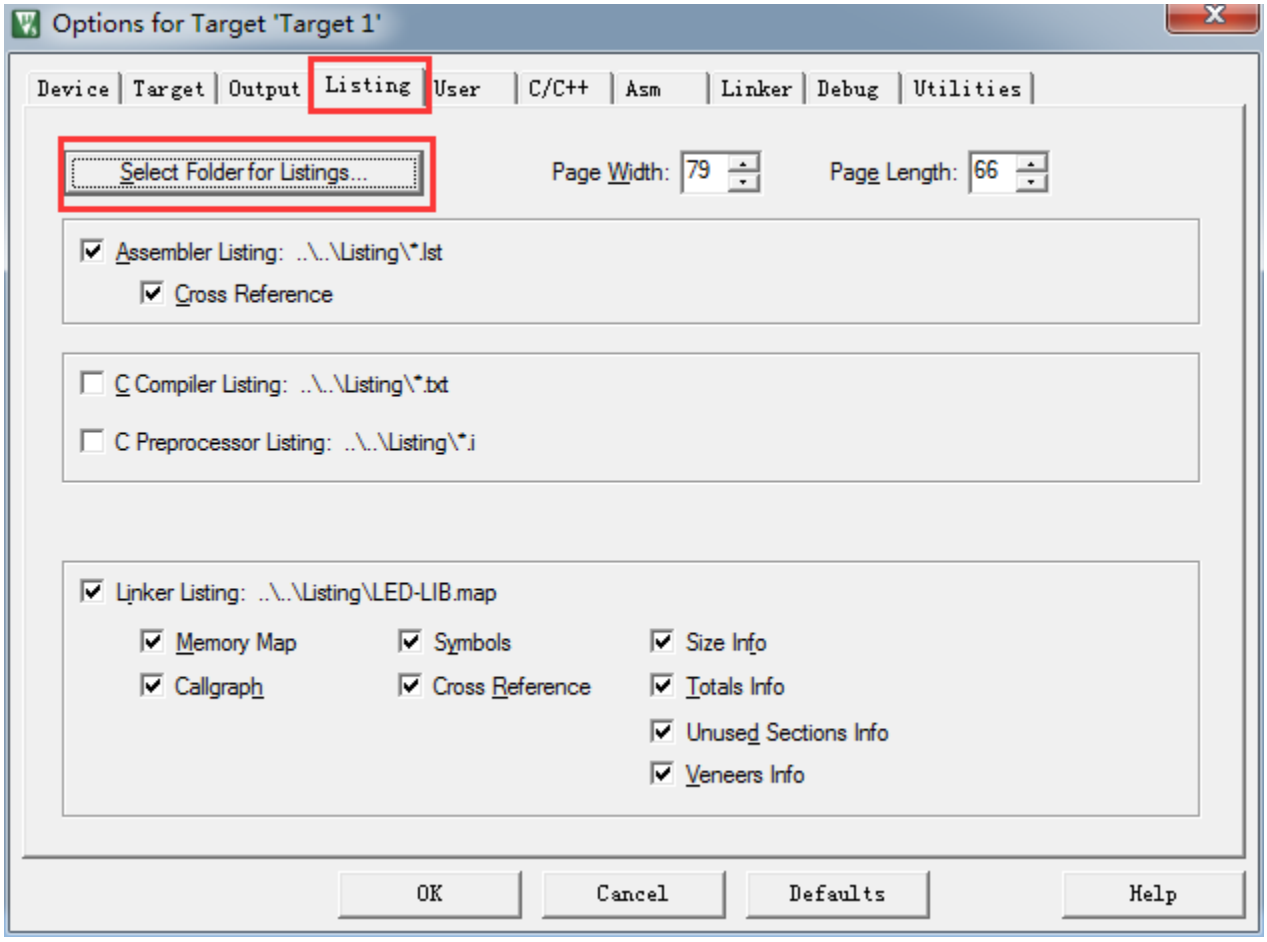
a. Target 中选中微库“Use MicroLib”，具体见图 *6_6*。选择微库的目的是为了在日后编写串口驱动的时候可以使用 printf 函数。而且有些应用中如果用了 STM32 的浮点运算单元 FPU，一定要同时开微库，不然有时会出现各种奇怪的现象。FPU 的开关选项在微库配置选项下方的“Use Single Precision”中，默认是开的。



b. Output 选项卡中把输出文件夹定位到我们工程目录下的 Objects 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，具体见图 *6_7*。



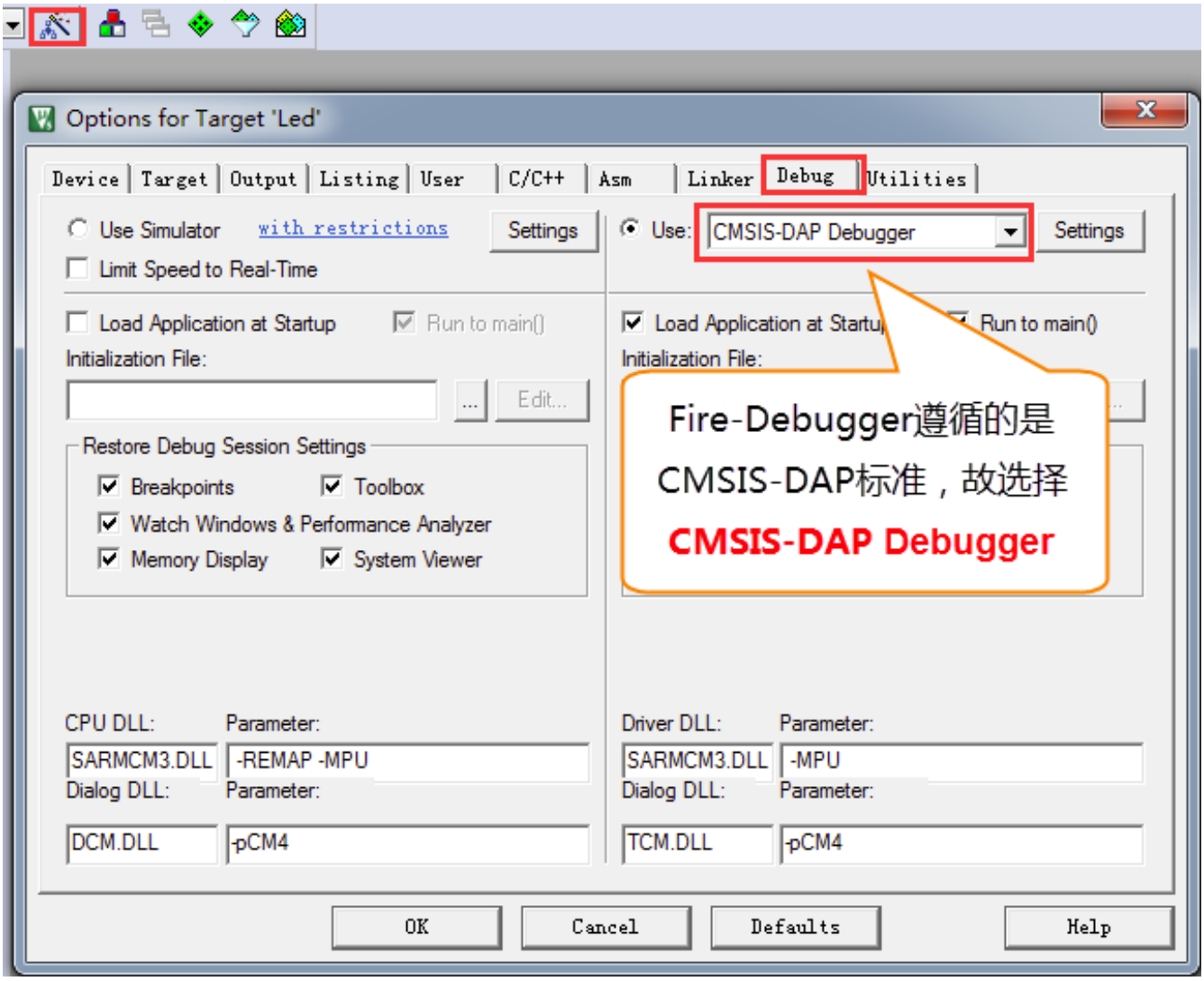
③ 在 Listing 选项卡中把输出文件夹定位到我们工程目录下的 Listing 文件夹。



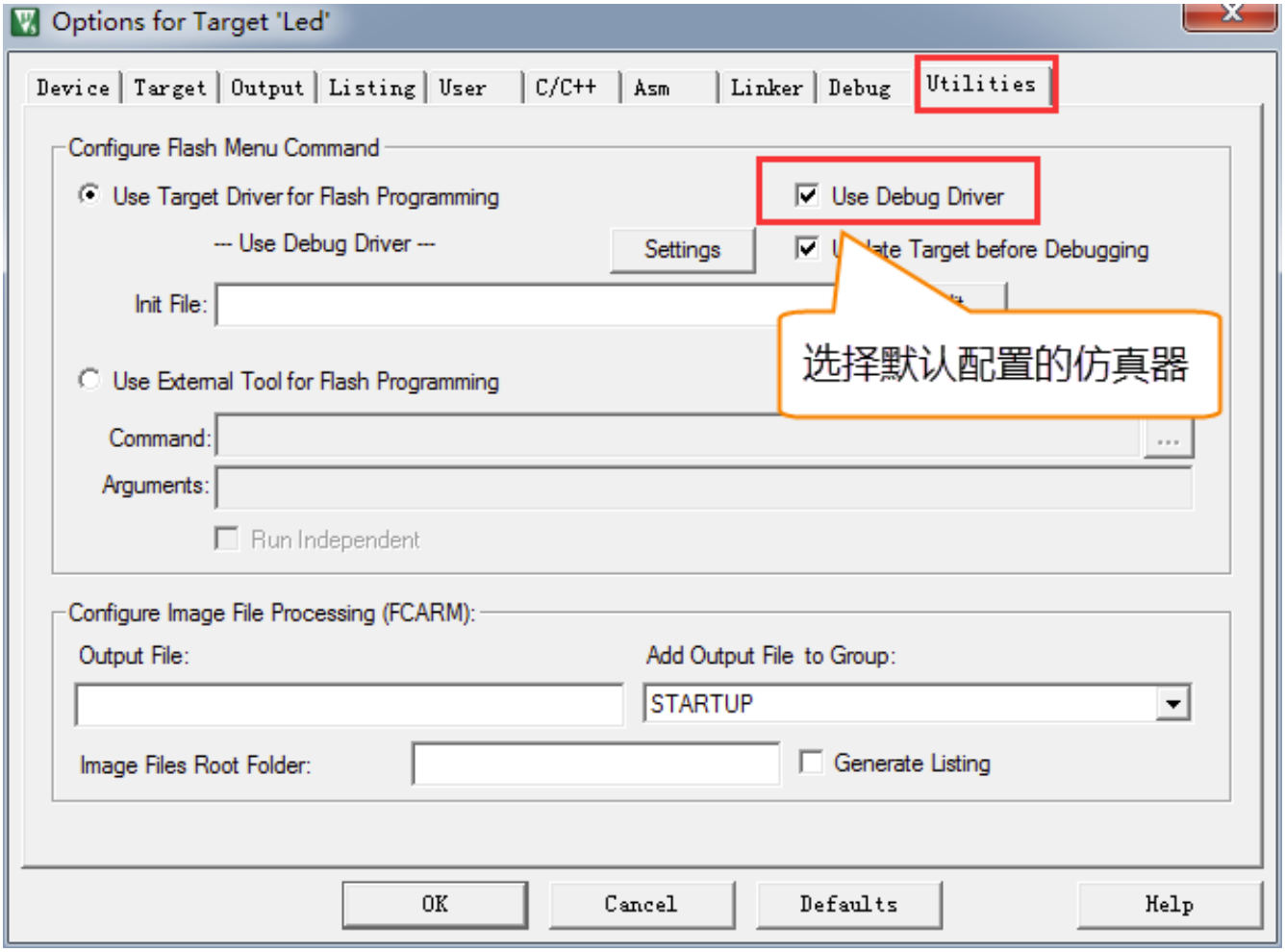
6.2.5 下载器配置

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：

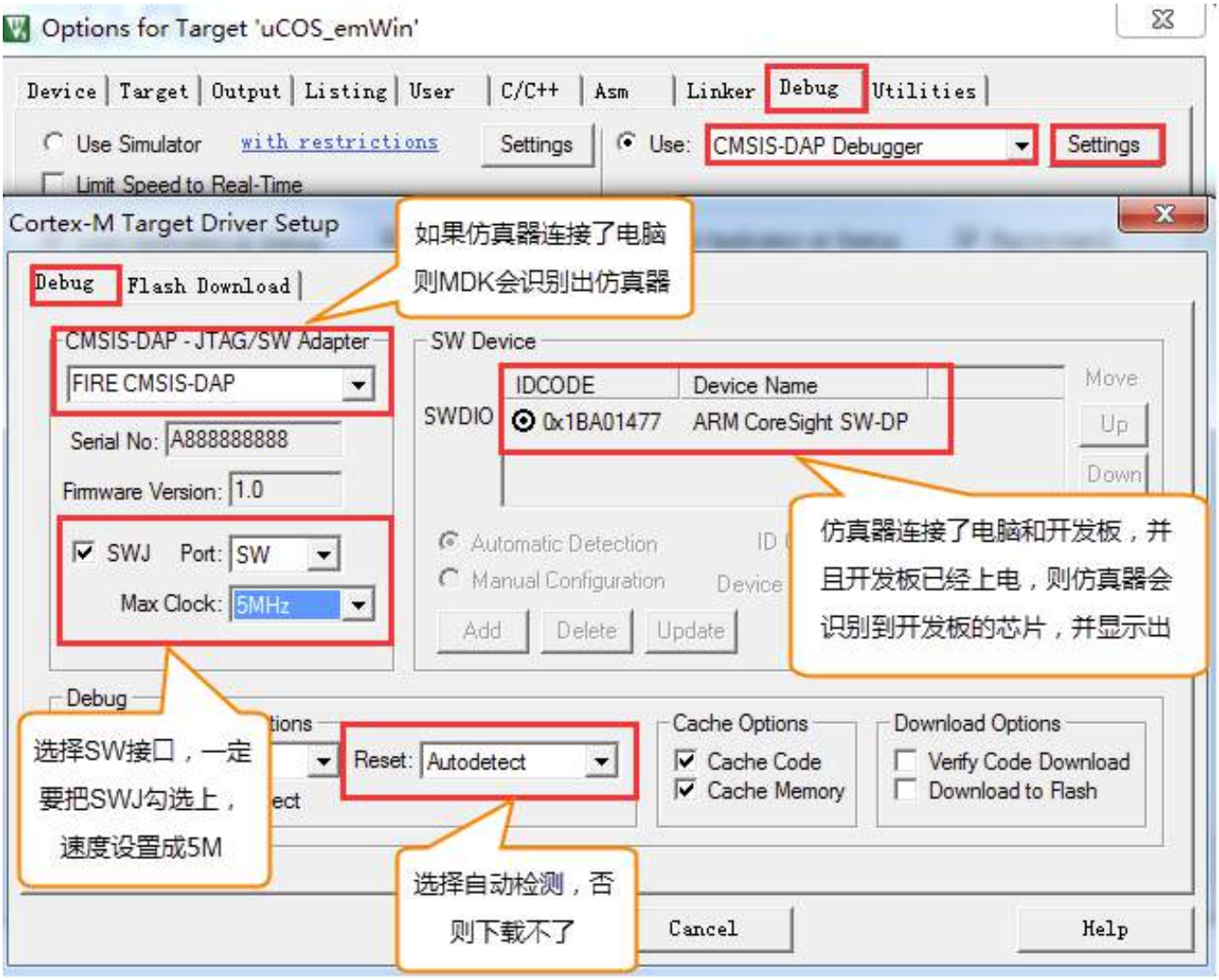
Debug 选项配置



Utilities 选项配置

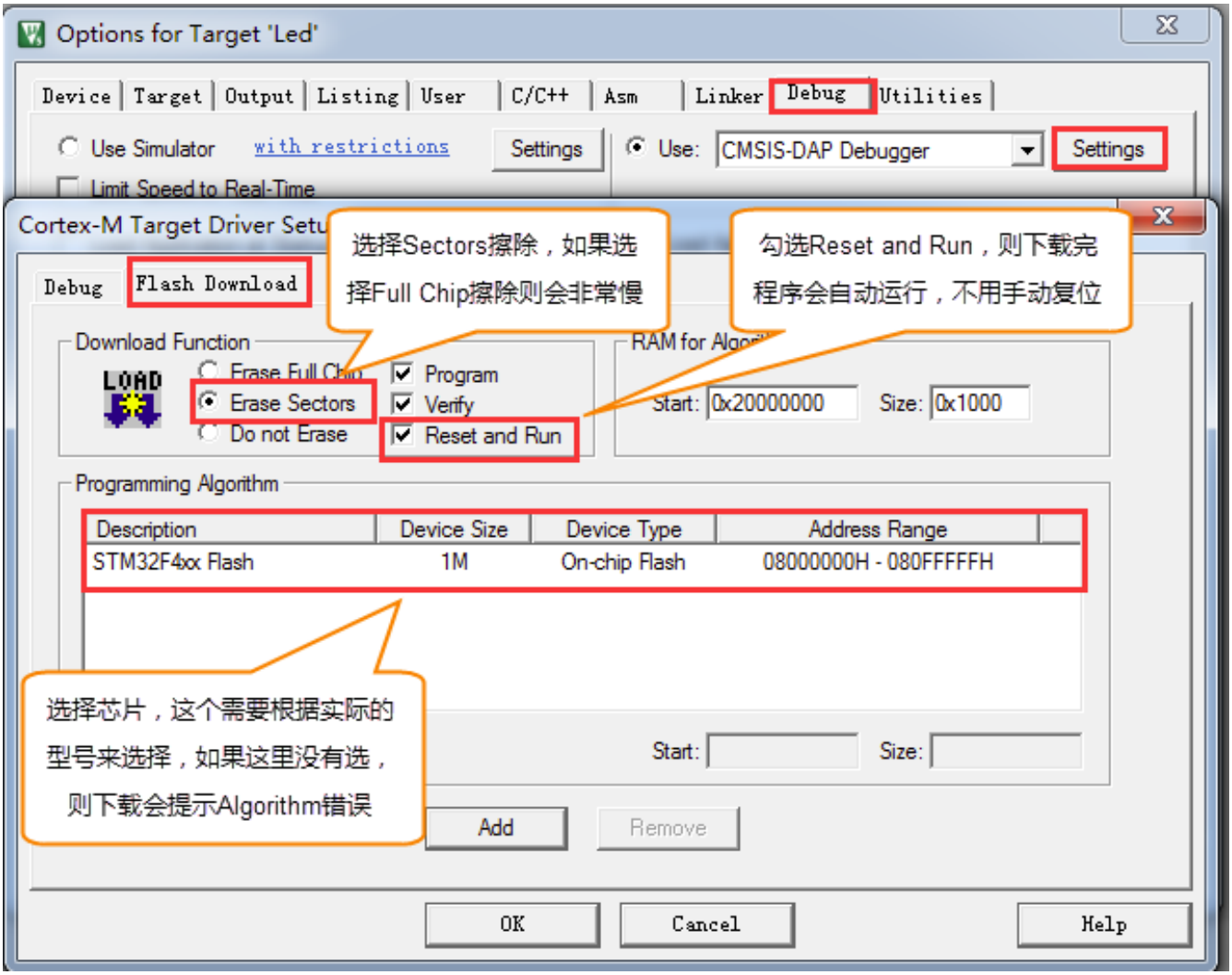


Debug Settings 选项配置



选择目标板，

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。F407-霸天虎选 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，具体设置见图 *6_12*。



6.3 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。下载程序的时候需要用仿真器连接电脑和开发板且开发板要供电。



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：《STM32F4xx 中文参考手册》，学习本章时，配合《STM32F4xx 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。

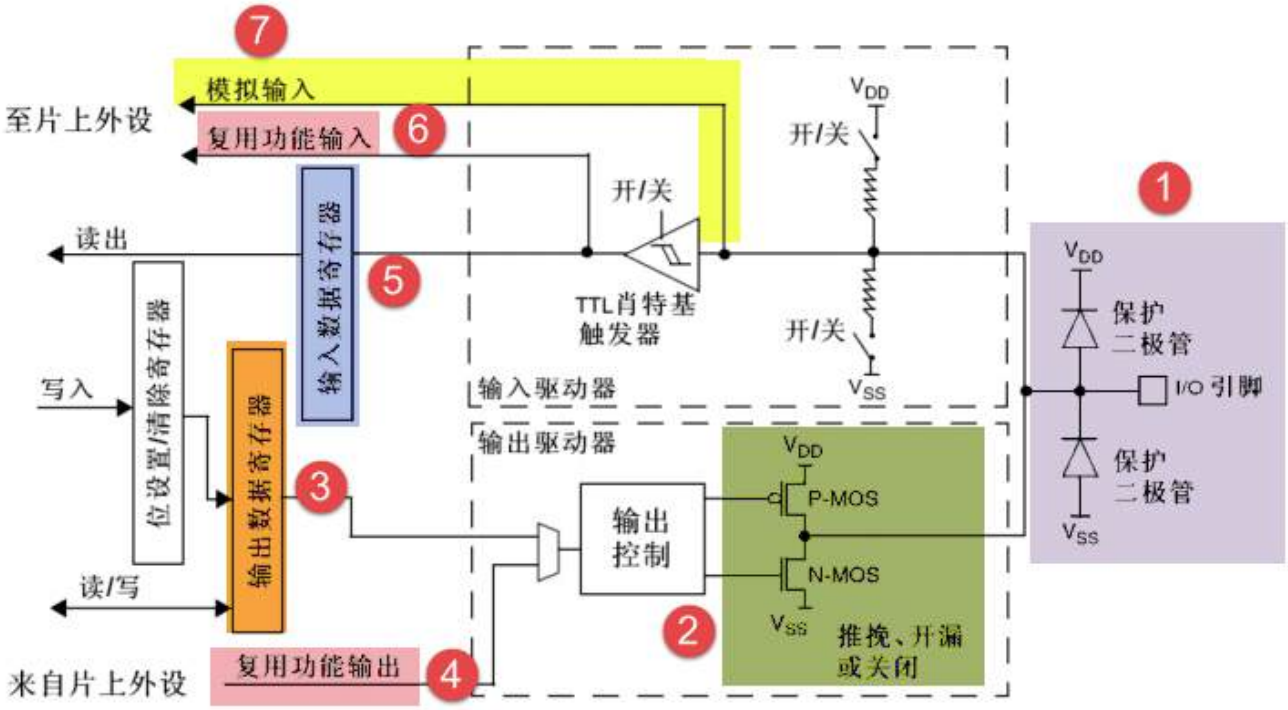
7.1 GPIO 简介

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，如型号为 STM32F4ZGT6 型号的芯片有 GPIOA、GPIOB、GPIOC 至 GPIOG 共 7 组 GPIO，芯片一共 144 个引脚，其中 GPIO 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。具体电压、电流范围可查阅《STM32F4xx 英文数据手册》。

上拉、下拉电阻，从它的结构我们可以看出，通过上、下拉对应的开关配置，我们可以控制引脚默认状态的电压，开启上拉的时候引脚电压为高电平，开启下拉的时候引脚电压为低电平，这样可以消除引脚不定状态的影响。如引脚外部没有外接器件，或者外部的器件不干扰该引脚电压时，STM32 的引脚都会有这个默认状态。

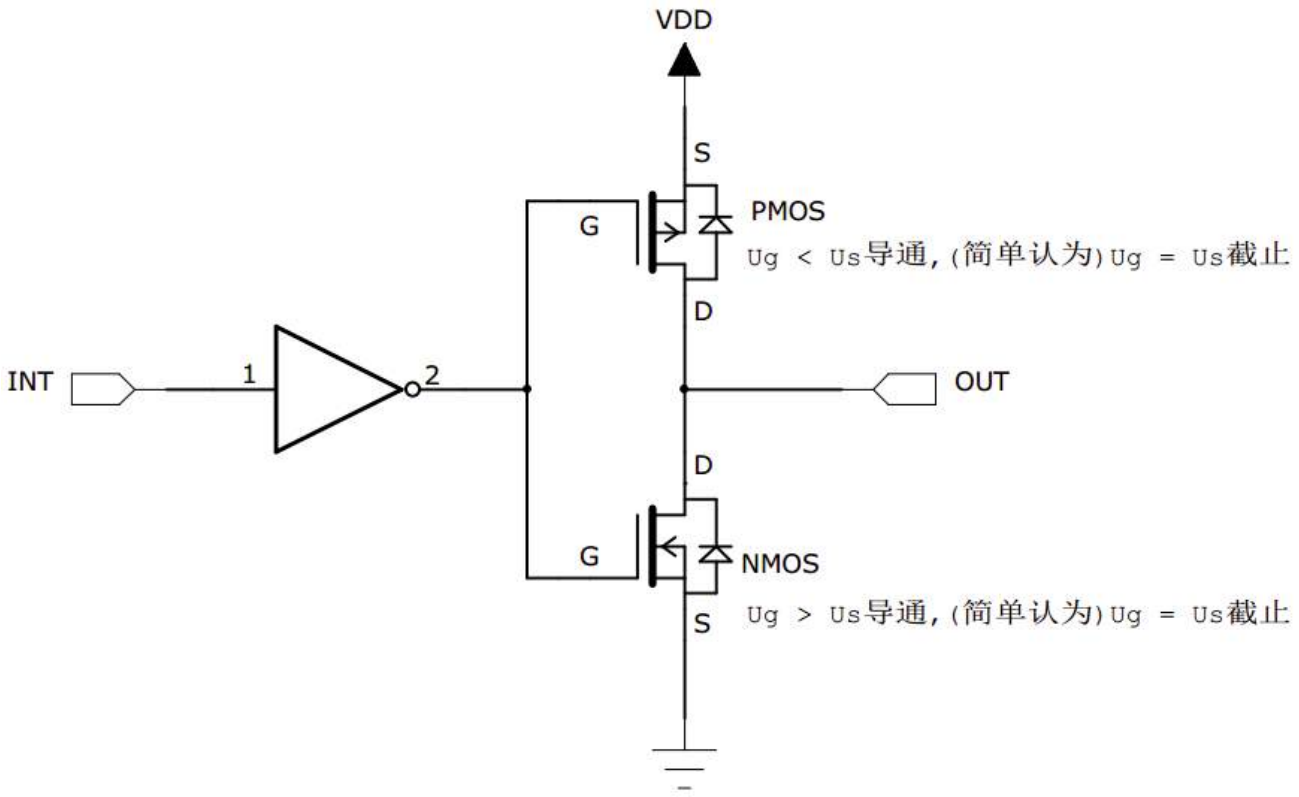
也可以设置“既不上拉也不下拉模式”，我们也把这种状态称为浮空模式，配置成这个模式时，直接用电压表测量其引脚电压为 1 点几伏，这是个不确定值。所以一般来说我们都会选择给引脚设置“上拉模式”或“下拉模式”使它有默认状态。

STM32 的内部上拉是“弱上拉”，即通过此上拉输出的电流是很弱的，如要求大电流还是需要外部上拉。通过“上拉/下拉寄存器 GPIOx_PUPDR”控制引脚的上、下拉以及浮空模式。

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*。

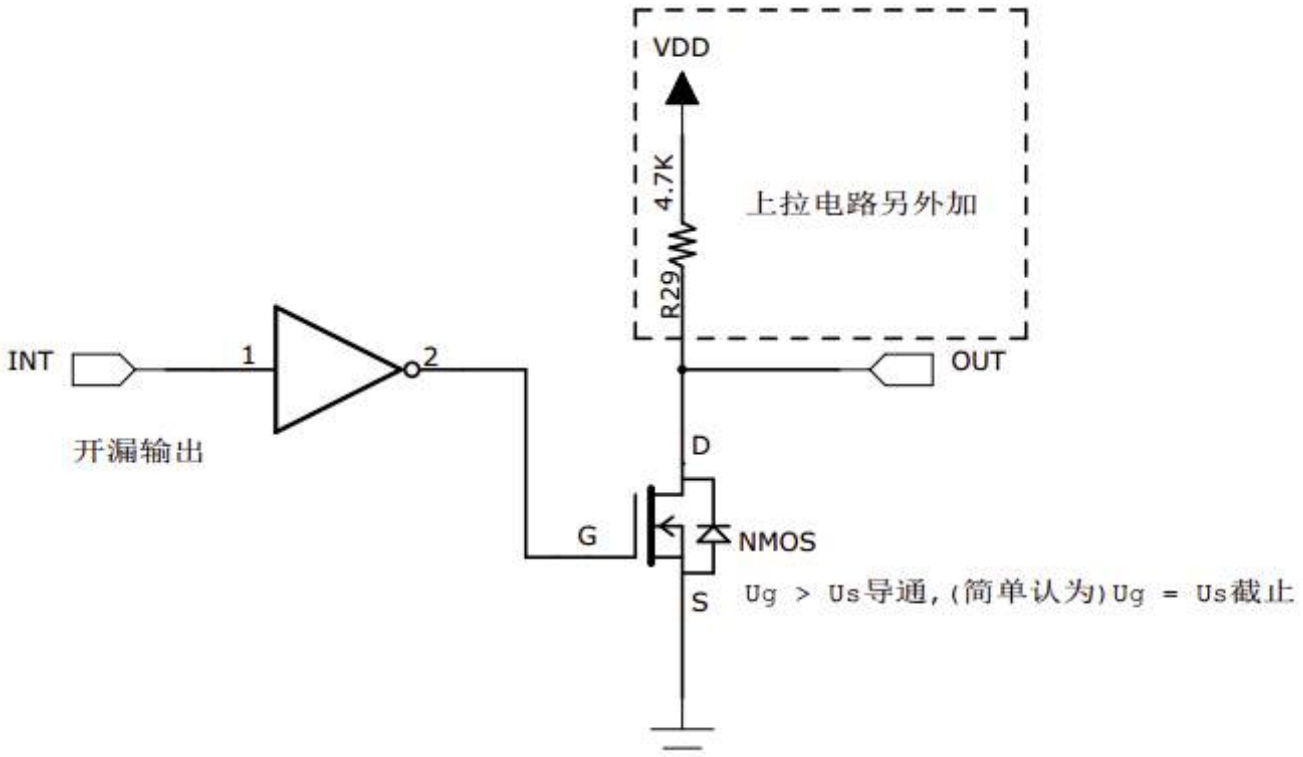

VDD
4.7K
R29
上拉电路另外加
INT
1
2
OUT
开漏输出
D
G
NMOS
S
Ug > Us导通,(简单认为)Ug = Us截止

图 7-3 开漏电路

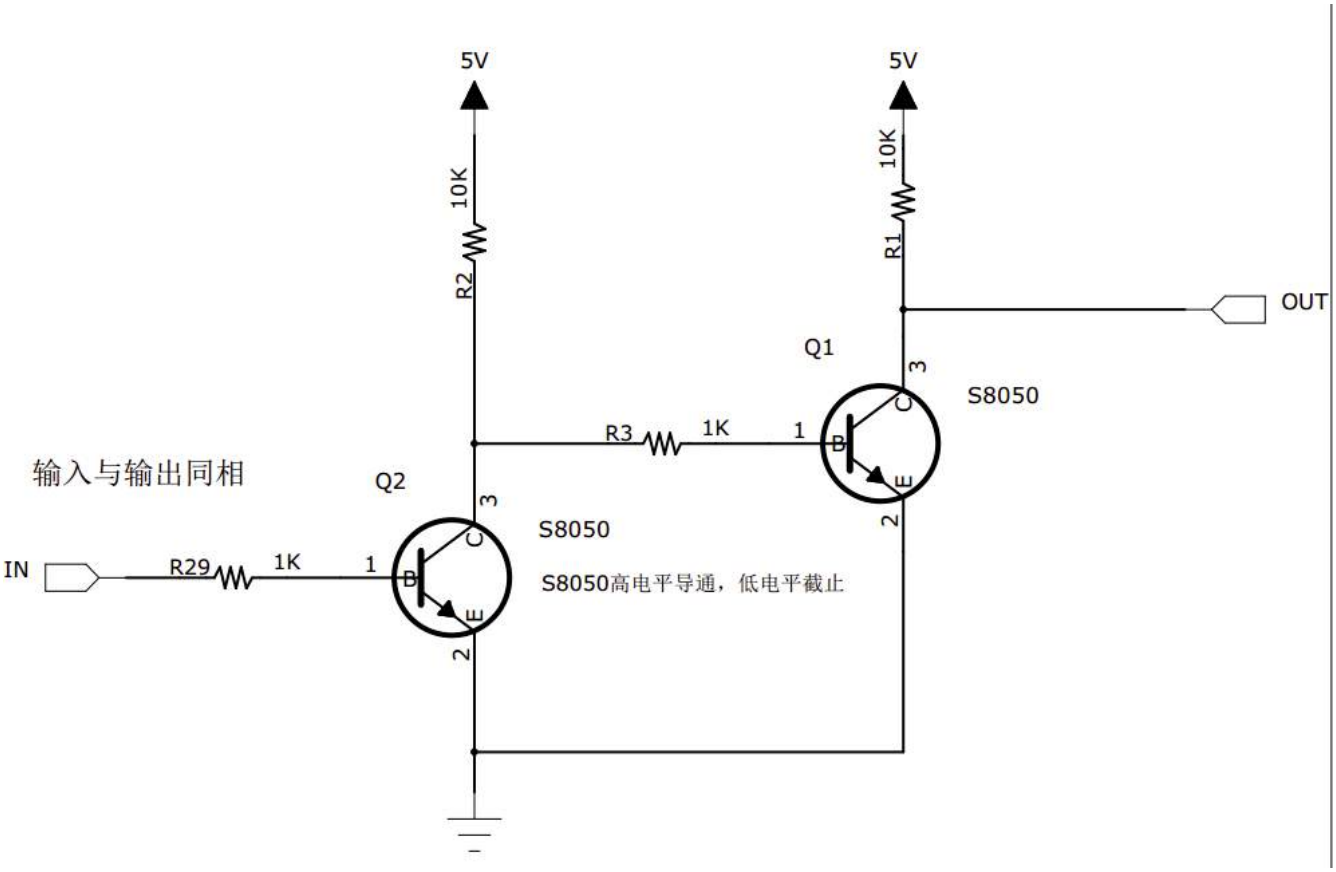


图 7-4 STM32 IO 对外输出 5V 电平

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。

7.2.1.5 输入数据寄存器

看 GPIO 结构框图的上半部分，它是 GPIO 引脚经过上、下拉电阻后引入的，它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。

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：

7.2.2.1 输入模式 (上拉/下拉/浮空)

在输入模式时，施密特触发器打开，输出被禁止。数据寄存器每隔 1 个 AHB1 时钟周期更新一次，可通过输入数据寄存器 GPIOx_IDR 读取 I/O 状态。其中 AHB1 的时钟如按默认配置一般为 168MHz。

用于输入模式时，可设置为上拉、下拉或浮空模式。

7.2.2.2 输出模式 (推挽/开漏，上拉/下拉)

在输出模式中，输出使能，推挽模式时双 MOS 管以方式工作，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 2MHz25MHz50MHz100MHz 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。

此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。

用于输出模式时，可使用上拉、下拉模式或浮空模式。但此时由于输出模式时引脚电平会受到 ODR 寄存器影响，而 ODR 寄存器对应引脚的位为 0，即引脚初始化后默认输出低电平，所以在这种情况下，上拉只起到小幅提高输出电流能力，但不会影响引脚的默认状态。。

7.2.2.3 复用功能 (推挽/开漏，上拉/下拉)

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。

用于复用功能时，可使用上拉、下拉模式或浮空模式。同输出模式，在这种情况下，初始化后引脚默认输出低电平，上拉只起到小幅提高输出电流能力，但不会影响引脚的默认状态。

7.2.2.4 模拟输入输出

模拟输入输出模式中，双 MOS 管结构被关闭，施密特触发器停用，上/下拉也被禁止。其它外设通过模拟通道进行输入输出。

通过对 GPIO 寄存器写入不同的参数，就可以改变 GPIO 的应用模式，再强调一下，要了解具体寄存器时一定要查阅《STM32F4xx 参考手册》中对应外设的寄存器说明。在 GPIO 外设中，通过设置“模式寄存器 GPIOx_MODER”可配置 GPIO 的输入/输出/复用/模拟模式，“输出类型寄存器 GPIOx_OTYPER”配置推挽/开漏模式，配置“输出速度寄存器 GPIOx_OSPEEDR”可选 2/25/50/100MHz 输出速度，“上/下拉寄存器 GPIOx_PUPDR”可配置上拉/下拉/浮空模式，各寄存器的具体参数值见表 7-1 GPIO 寄存器的参数配置。

表 7-1 GPIO 寄存器的参数配置。

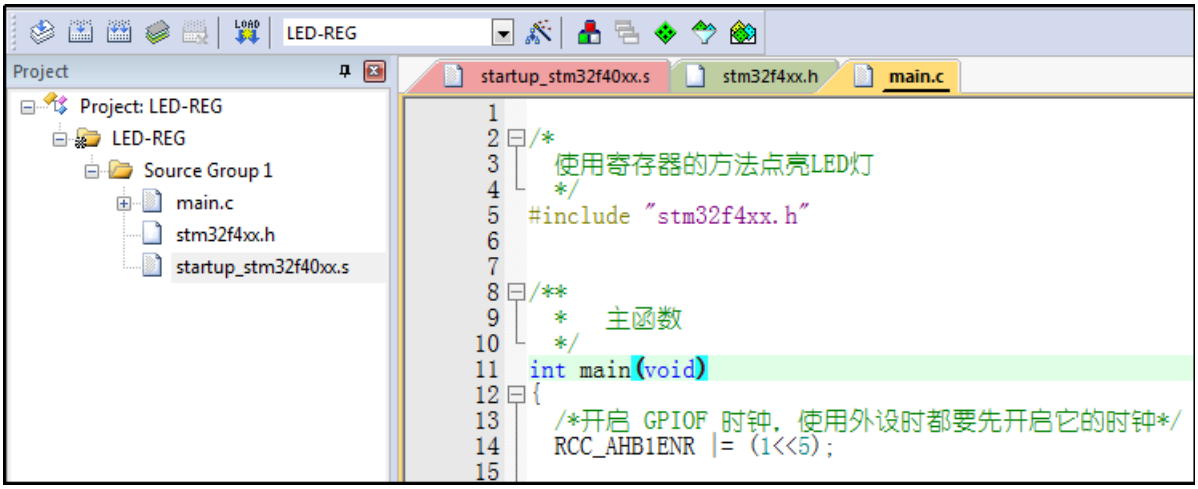
模式寄存器的 MODER 位[0:1]	输出类型寄存器的 OTYPER 位	输出速度寄存器的 OSPEEDR	上/下拉寄存器的 PUPDR 位[0:1]
01 -输出模式	0 -推挽模式 1 -开漏模式	00 -速度 2MHz 01 -速度 25MHz 10 -速度 50MHz 11 -速度 100MHz	00 -无上拉无下拉 01 -上拉 10 -下拉 11 -保留
10 -复用模式			
00 -输入模式	不可用	不可用	
11 -模拟功能	不可用	不可用	00 -无上拉无下拉 01 -保留 10 -保留 11 -保留

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，请您直接用 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“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。

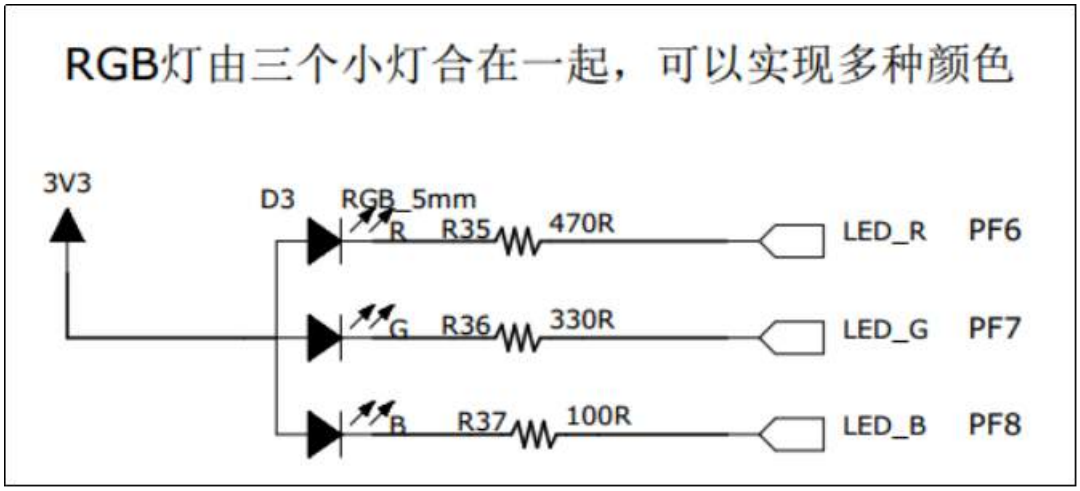
自己尝试新建工程时，请对照查阅《新建工程—寄存器版》章节。若没有安装 KEIL5 软件，请参考《如何安装 KEIL5》章节。

打开该工程，见图 7_7，可看到一共有三个文件，分别 startup_stm32f40xx.s 、stm32f4xx.h 以及 main.c，接下来我们讲会对这三个工程文件进行讲解。



7.3.1 硬件连接

在本教程中 STM32 芯片的 PF6、PF7 和 PF8 引脚分别与一个 RGB 灯的 R 灯（红灯）、G 灯（蓝灯）和 B 灯（绿灯）连接，具体见图 7_8，RGB 灯里面的三个小灯都可以单独控制，如果有两个或者三个灯同时亮的话就会混合成其它的颜色。



图中从 3 个 LED 灯的阳极引出连接到 3.3V 电源，阴极各经过 1 个限流电阻电阻引入至 STM32 的 3 个 GPIO 引脚 PF6、PF7 和 PF8 中，所以我们只要控制这三个引脚输出高低电平，即可控制其所连接 LED 灯的亮灭。如果您的实验板 STM32 连接到 LED 灯的引脚或极性不一样，只需要修改程序到对应的 GPIO 引脚即可，工作原理都是一样的。

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

名为“startup_stm32f40xx.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-M4 内核支持的指令，可从《Cortex-M4 Technical Reference Manual》查到，也可参考《Cortex-M3 权威指南中文》，M3 跟 M4 大部分汇编指令相同。

startup_stm32f40xx.s 文件是由官方提供的，一般有需要也是在官方的基础上修改，不会自己完全重写。该文件可以从 KEIL5 安装目录找到，也可以从 ST 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_1*。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 1: 代码清单 7-1 复位后执行的程序

```
;Reset handler

Reset_Handler PROC

EXPORT Reset_Handler [WEAK]

IMPORT SystemInit

IMPORT __main

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。

第七行把 SystemInit 的地址加载到寄存器 R0。

第八行程序跳转到 R0 中的地址执行程序，即执行 SystemInit 函数的内容。

第九行把 __main 的地址加载到寄存器 R0。

第十行程序跳转到 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。

第十一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 stm32f4xx.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。在上一章，我们知道寄存器就是特殊的内存空间，可以通过指针操作访问寄存器。所以此处我们根据 STM32 的存储器映射先定义好各个寄存器的地址，把这些地址定义都统一写在 stm32f4xx.h 文件中，见代码清单 7_2。

列表 2: 代码清单 7-2 外设地址定义

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

(下页继续)

(续上页)

```
/* 总线基地址 */
#define AHB1PERIPH_BASE        (PERIPH_BASE + 0x00020000)
/*GPIO 外设基地址 */
#define GPIOF_BASE             (AHB1PERIPH_BASE + 0x1400)

/* GPIOH 寄存器地址，强制转换成指针 */
#define GPIOH_MODER              *(unsigned int*)(GPIOH_BASE+0x00)
#define GPIOH_OTYPER             *(unsigned int*)(GPIOH_BASE+0x04)
#define GPIOH_OSPEEDR            *(unsigned int*)(GPIOH_BASE+0x08)
#define GPIOH_PUPDR              *(unsigned int*)(GPIOH_BASE+0x0C)
#define GPIOH_IDR                *(unsigned int*)(GPIOH_BASE+0x10)
#define GPIOH_ODR                *(unsigned int*)(GPIOH_BASE+0x14)
#define GPIOH_BSRR               *(unsigned int*)(GPIOH_BASE+0x18)
#define GPIOH_LCKR               *(unsigned int*)(GPIOH_BASE+0x1C)
#define GPIOH_AFRL               *(unsigned int*)(GPIOH_BASE+0x20)
#define GPIOH_AFRH               *(unsigned int*)(GPIOH_BASE+0x24)

/*RCC 外设基地址 */
#define RCC_BASE                (AHB1PERIPH_BASE + 0x3800)
/*RCC 的 AHB1 时钟使能寄存器地址，强制转换成指针 */
#define RCC_AHB1ENR              *(unsigned int*)(RCC_BASE+0x30)
```

GPIO 外设的地址跟上一章讲解的相同，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_stm32f40xx.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 芯片会自动将内部的高速时钟 HIS（16M）作为系统时钟，程序还是能跑的，只是跑的比较慢（通常情况下，我们会把系统时钟设置为 168M，远远大于 HIS 的 16M）。我们在 main 中添加 SystemInit 函数：

```
// 函数为空，目的是为了骗过编译器不报错

void SystemInit(void)

{

}
```

接下来在 main 函数中添加代码，对寄存器进行控制，有关 GPIO 寄存器的详细描述请参考《STM32F4xx 中文参考手册》“通用 I/O(GPIO)”章节的寄存器描述部分。

7.3.4.1 GPIO 模式

首先我们把连接到 RGB 红灯的 PF6 引脚配置成输出模式，即配置 GPIO 的 MODER 寄存器，具体见图 7_9。MODER 中包含 0-15 号引脚，每个引脚占用 2 个寄存器位。这两个寄存器位设置成“01”时即为 GPIO 的输出模式，具体见代码清单 7_4。

列表 3: 代码清单 7-4 配置输出模式

```
/*GPIOF MODER6 清空 */
GPIOF_MODER  &= ~( 0x03<< (2*6));
/*PF6 MODER6 = 01b 输出模式 */
GPIOH_MODER |= (1<<2*6);
```

GPIO 端口模式寄存器 (GPIOx_MODER) (x = A..I)

GPIO port mode register

偏移地址：0x00

复位值：

- 0xA800 0000（端口 A）
- 0x0000 0280（端口 B）
- 0x0000 0000（其它端口）

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

位 2y:2y+1 **MODERy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 方向模式。

00：输入（复位状态）

01：通用输出模式

10：复用功能模式

11：模拟模式

图 7-7 MODER 寄存器说明 (摘自《STM32F4xx 参考手册》)

在代码中，我们先把 GPIOH MODER 寄存器的 MODER6 对应位清 0，然后再向它赋值“01”，从

而使 GPIOF6 引脚设置成输出模式。

代码中使用了“&=~”、“|=”这种位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 MODER 寄存器赋值：

```
GPIOF_MODER = 0x0004 0000;
```

这时 MODER6 的两个位被设置成“01”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 00 的输入模式。所以为了不影响寄存器的其它位，必须使用“&=~”（清 0）、“|=”（置位）这种位操作方法来实现对寄存器的写操作。

7.3.4.2 输出类型

GPIO 输出有推挽和开漏两种类型，我们了解到开漏类型不能直接输出高电平，要输出高电平还要在芯片外部接上拉电阻，不符合我们的硬件设计，所以我们直接使用推挽模式。配置 OTYPER 寄存中的 OTYPER6 寄存器位，具体见图 *7_12*。该位设置为 0 时 PF6 引脚即为推挽模式，具体见*代码清单 7_8*。

7.4.2 GPIO 端口输出类型寄存器 (GPIOx_OTYPER) (x = A..I)

GPIO port output type register

偏移地址：0x04

复位值：0x0000 0000

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

位 31:16 保留，必须保持复位值。

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

列表 4: 代码清单 7-8 设置为推挽模式

```
/*GPIOF OTYPER6 清空 */
GPIOH_OTYPER &= ~(1<<1*6);
/*PF6 OTYPER6 = 0b 推挽模式 */
GPIOH_OTYPER |= (0<<1*6);
```

7.3.4.3 输出速度

GPIO 引脚的输出速度是引脚支持高低电平切换的最高频率，本实验可以随便设置。此处我们配置 OSPEEDR 寄存器中的寄存器位 OSPEEDR6 即可控制 PF6 的输出速度，寄存器描述见图 *7_13*，具体代码见*代码清单 7_9*。

7.4.3 GPIO 端口输出速度寄存器 (GPIOx_OSPEEDR) (x = A..I/)

GPIO port output speed register

偏移地址：0x08

复位值：

- 0x0000 00C0（端口 B）
- 0x0000 0000（其它端口）

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

位 2y:2y+1 **OSPEEDRy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)
这些位通过软件写入，用于配置 I/O 输出速度。
00：2 MHz（低速）
01：25 MHz（中速）
10：50 MHz（快速）
11：30 pF 时为 100 MHz（高速）（15 pF 时为 80 MHz 输出（最大速度））

列表 5: 代码清单 7-9 设置为推挽模式

```
/*GPIOF OSPEEDR6 清空 */
GPIOH_OSPEEDR &= ~(0x03<<2*6);
/*PF6 OSPEEDR6 = 0b 速率 2MHz*/
GPIOH_OSPEEDR |= (0<<2*6);
```

7.3.4.4 上/下拉模式

当 GPIO 引脚用于输入时，引脚的上/下拉模式可以控制引脚的默认状态。但现在我们的 GPIO 引脚用于输出，引脚受 ODR 寄存器（数据输出寄存器）影响，ODR 寄存器对应引脚位初始初始化后默认值为 0，引脚输出低电平，所以这时我们配置上/下拉模式都不会影响引脚电平状态。但因此处上拉能小幅提高电流输出能力，我们配置它为上拉模式，即配置 PUPDR 寄存器的 PUPDR6 位，寄存器描述具体见图 *7_14*，设置该位为二进制值 “01”，具体代码见*代码清单 7_10*。

7.4.4 GPIO 端口上拉/下拉寄存器 (GPIOx_PUPDR) (x = A..I/)

GPIO port pull-up/pull-down register

偏移地址：0x0C

复位值：

- 0x6400 0000（端口 A）
- 0x0000 0100（端口 B）
- 0x0000 0000（其它端口）

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

位 2y:2y+1 **PUPDRy[1:0]**：端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 上拉或下拉。

00：无上拉或下拉

01：上拉

10：下拉

11：保留

列表 6: 代码清单 7-9 设置为下拉模式

```
/*GPIOF PUPDR6 清空 */
GPIOF_PUPDR &= ~(0x03<<2*6);
/*PF6 PUPDR6 = 01b 上拉模式 */
GPIOF_PUPDR |= (1<<2*6);
```

7.3.4.5 控制引脚输出电平

在输出模式时，对 BSRR 寄存器和 ODR 寄存器写入参数即可控制引脚的电平状态。简单起见，此处我们使用 BSRR 寄存器控制，对相应的 BR6 位设置为 1 时 PF6 即为低电平，点亮 LED 灯，对它的 BS6 位设置为 1 时 PF6 即为高电平，关闭 LED 灯。寄存器 BSRR 的具体描述见图 *7_10*，具体代码见*代码清单 7_5*。

7.4.7 GPIO 端口置位/复位寄存器 (GPIOx_BSRR) (x = A..I)

GPIO port bit set/reset register

偏移地址：0x18

复位值：0x0000 0000

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

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

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

列表 7: 代码清单 7-5 控制引脚输出电平

```
/*PF6 BSRR 寄存器的 BR6 置 1，使引脚输出低电平 */
GPIOF_BSRR |= (1<<16<<6);

/*PF6 BSRR 寄存器的 BS6 置 1，使引脚输出高电平 */
GPIOF_BSRR |= (1<<6);
```

7.3.4.6 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。因为 STM32 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《STM32F4xx 中文参考手册》的第六章有详细的讲解。

所有的 GPIO 都挂载到 AHB1 总线上，所以它们的时钟由 AHB1 外设时钟使能寄存器 (RCC_AHB1ENR) 来控制，有关该寄存器的详细描述请参考《STM32F4xx 中文参考手册》的 RCC 章节的寄存器描述部分。其中 GPIOF 端口的时钟由该寄存器的位 5 写 1 使能来开启。有关 STM32 的时钟系统我们在往后的 RCC 章节会详细的讲解，此处我们只需知道在访问 GPIO 这个外设的寄存器之前，要先开启它的时钟。具体代码见*代码清单 7_6*。

列表 8: 代码清单 7-6 开启端口时钟

```
// 开启 GPIOB 端口 时钟

RCC_AHB1ENR |= (1<<5);
```

7.3.4.7 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_7。

列表 9: 代码清单 7-7 main 文件中控制 LED 灯的代码

```
/*
  使用寄存器的方法点亮 LED 灯
  */
#include "stm32f4xx.h"

/**
  *  主函数
  */
 int main(void)
 {
     /* 开启 GPIOF 时钟，使用外设时都要先开启它的时钟 */
     RCC_AHB1ENR |= (1<<5);

     /* LED 端口初始化 */

     /*GPIOF MODER6 清空 */
     GPIOF_MODER  &= ~( 0x03<< (2*6));
     /*PF6 MODER6 = 01b 输出模式 */
     GPIOF_MODER |= (1<<2*6);
```

(下页继续)

(续上页)

```
    /*GPIOF OTYPER6 清空 */
    GPIOF_OTYPER &= ~(1<<1*6);
    /*PF6 OTYPER6 = 0b 推挽模式 */
    GPIOF_OTYPER |= (0<<1*6);

    /*GPIOF OSPEEDR6 清空 */
    GPIOF_OSPEEDR &= ~(0x03<<2*6);
    /*PF6 OSPEEDR6 = 0b 速率 2MHz*/
    GPIOF_OSPEEDR |= (0<<2*6);

    /*GPIOF PUPDR6 清空 */
    GPIOF_PUPDR &= ~(0x03<<2*6);
    /*PF6 PUPDR6 = 01b 上拉模式 */
    GPIOF_PUPDR |= (1<<2*6);

    /*PF6 BSRR 寄存器的 BR6 置 1，使引脚输出低电平 */
    GPIOF_BSRR |= (1<<16<<6);

    /*PF6 BSRR 寄存器的 BS6 置 1，使引脚输出高电平 */
    //GPIOF_BSRR |= (1<<6);

    while (1);
}

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

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　自己写库—构建库函数雏形

本章参考资料：《STM32F4xx 中文参考手册》、《STM32F4xx 英文数据手册》。

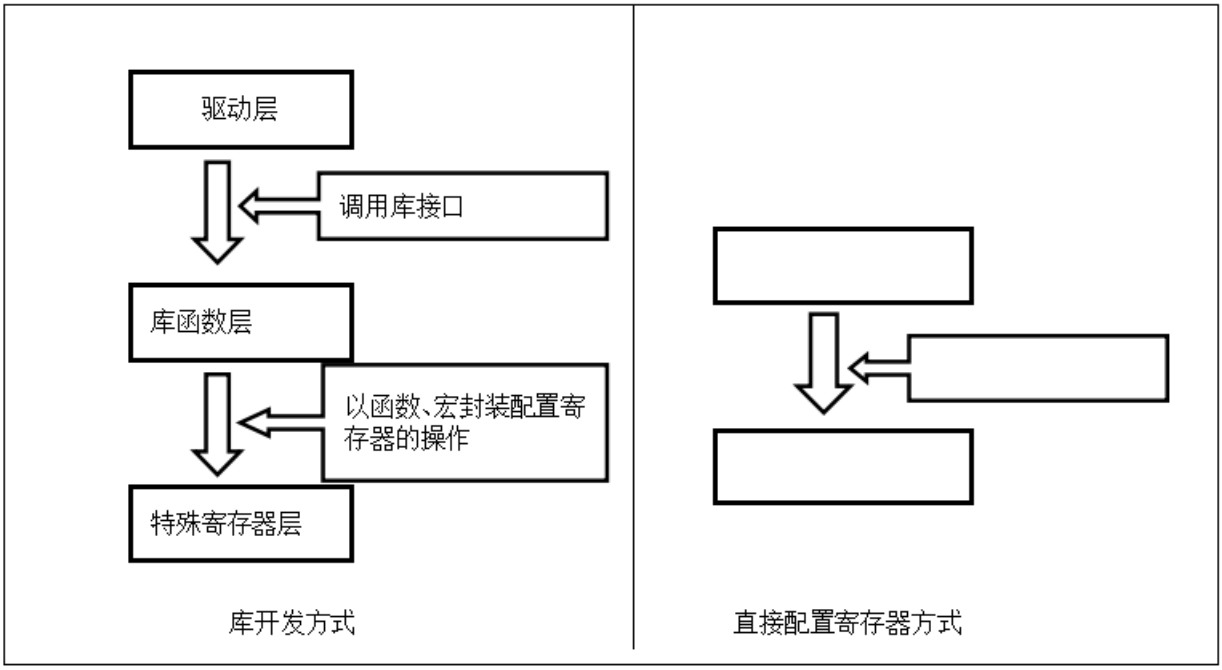
虽然我们上面用寄存器编程的方法点亮了 LED，乍看一下好像代码也很简单，但是我们别侥幸以后就可以一直用寄存器开发。在用寄存器点亮 LED 的时候，我们会发现 STM32 的寄存器都是 32 位的，每次配置的时候都要对照着《STM32F4xx 中文参考手册》中寄存器的说明，然后根据说明对每个控制的寄存器位写入特定参数，因此在配置的时候非常容易出错，而且代码还很不好理解，不便于维护。所以学习 STM32 最好的方法是用官方提供的函数库来开发，然后在函数库的基础上了解底层，学遍所有寄存器。

8.1 什么是 STM32 函数库

ST 官方为 F4 提供了 3 种函数库，即函数库、HAL 库以及 LL 库，它们是由 ST 公司针对 STM32 提供的函数接口，即 API (Application Program Interface)，开发者可调用这些函数接口来配置 STM32 的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速，易于阅读，维护成本低等优点。在本册教材中我们主要针对 HAL 库进行编程。

当我们调用库 API 的时候不需要挖空心思去了解库底层的寄存器操作，就像当年我们刚开始学习 C 语言的时候，用 prinft() 函数时只是学习它的使用格式，并没有去研究它的源码实现，但需要深入研究的时候，经过千锤百炼的库源码就是最佳学习范例。

实际上，库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口。库开发方式与直接配置寄存器方式的区别见图 *8_1*。



8.2 为什么采用库来开发及学习？

在以前 8 位机时代的程序开发中，一般直接配置芯片的寄存器，控制芯片的工作方式，如中断，定时器等。配置的时候，常常要查阅寄存器表，看用到哪些配置位，为了配置某功能，该置 1 还是置 0。这些都是很琐碎的、机械的工作，因为 8 位机的软件相对来说较简单，而且资源很有限，所以可以直接配置寄存器的方式来开发。

对于 STM32，因为外设资源丰富，带来的必然是寄存器的数量和复杂度的增加，这时直接配置寄存器方式的缺陷就突显出来了：

(1) **开发速度慢**

(2) **程序可读性差**

(3) **维护复杂**

这些缺陷直接影响了开发效率，程序维护成本，交流成本。库开发方式则正好弥补了这些缺陷。

而坚持采用直接配置寄存器的方式开发的程序员，会列举以下原因：

(1) **具体参数更直观**

(2) **程序运行占用资源少**

相对于库开发的方式，直接配置寄存器方式生成的代码量的确会少一点，但因为 STM32 有充足的资源，权衡库的优势与不足，绝大部分时候，我们愿意牺牲一点 CPU 资源，选择库开发。一般只有在对代码运行时间要求极苛刻的地方，才用直接配置寄存器的方式代替，如频繁调用的中断服务函数。

对于库开发与直接配置寄存器的方式，就好比编程是用汇编好还是用 C 好一样。在 STM32F1 系列刚推出函数库时引起程序员的激烈争论，但是，随着 ST 库的完善与大家对库的了解，更多的程序员选择了库开发。现在 STM32F1 系列和 STM32F4 系列各有一套自己的函数库，但是它们大部分是兼容的，F1 和 F4 之间的程序移植，只需要小修改即可。而如果要移植用寄存器写的程序，那简直跟脱胎换骨差不多。

用库来进行开发，市场已有定论，用户群说明了一切，但对于 STM32 的学习仍然有人认为用寄存器好，而且汇编不是还没退出大学教材么？认为这种方法直观，能够了解到是配置了哪些寄存器，怎样配置寄存器。事实上，库函数的底层实现恰恰是直接配置寄存器方式的最佳例子，它代替我们完成了寄存器配置的工作，而想深入了解芯片是如何工作的话，只要直接查看库函数的最底层实现就能理解，相信你会为它严谨、优美的实现方式而陶醉，要想修炼 C 语言，就从 ST 的库开始吧。所以在以后的章节中，使用软件库是我们的重点，而且我们通过讲解库 API 去高效地学习 STM32 的寄存器，并不至于因为用库学习，就不会用寄存器控制 STM32 芯片。

8.3 实验：构建库函数雏形

虽然库的优点多多，但很多人对库还是很忌惮，因为一开始用库的时候有很多代码，很多文件，不知道如何入手。不知道您是否认同这么一句话：一切的恐惧都来源于无知。我们对库忌惮那是因为我们不知道什么是库，不知道库是怎么实现的。

接下来，我们在寄存器点亮 LED 的代码上继续完善，把代码一层层封装，实现库的最初的雏形，相信经过这一步的学习后，您对库的运用会游刃有余。这里我们只讲如何实现 GPIO 函数库，其他外设的我们直接参考 STM32HAL 库学习即可，不必自己写。

下面请打开本章配套例程“自己写库—构建库函数雏形”来阅读理解，该例程是在上一章的基础

上修改得来的。

8.3.1 修改寄存器地址封装

上一章中我们在操作寄存器的时候，操作的都是寄存器的绝对地址，如果每个外设寄存器都这样操作，那将非常麻烦。我们考虑到外设寄存器的地址都是基于外设基地址的偏移地址，都是在外设基地址上逐个连续递增的，每个寄存器占 32 个字节，这种方式跟结构体里面的成员类似。所以我们可以定义一种外设结构体，结构体的地址等于外设的基地址，结构体的成员等于寄存器，成员的排列顺序跟寄存器的顺序一样。这样我们操作寄存器的时候就不用每次都找到绝对地址，只要知道外设的基地址就可以操作外设的全部寄存器，即操作结构体的成员即可。

在工程中的“stm32f4xx.h”文件中，我们使用结构体封装 GPIO 及 RCC 外设的的寄存器，具体见*代码清单 8_1*。结构体成员的顺序按照寄存器的偏移地址从低到高排列，成员类型跟寄存器类型一样。

列表 1: 代码清单 8-1 封装寄存器列表

```
//寄存器的值常常是芯片外设自动更改的，即使 CPU 没有执行程序，也有可能发生变化
//编译器有可能会对没有执行程序的变量进行优化

//volatile 表示易变的变量，防止编译器优化，
#define     __IO    volatile
typedef unsigned int uint32_t;
typedef unsigned short uint16_t;

/* GPIO 寄存器列表 */
typedef struct {
 __IO  uint32_t MODER;     /*GPIO 模式寄存器          地址偏移：0x00          */
 __IO  uint32_t OTYPER;    /*GPIO 输出类型寄存器      地址偏移：0x04          */
 __IO  uint32_t OSPEEDR;   /*GPIO 输出速度寄存器      地址偏移：0x08          */
 __IO  uint32_t PUPDR;     /*GPIO 上拉/下拉寄存器     地址偏移：0x0C          */
 __IO  uint32_t IDR;       /*GPIO 输入数据寄存器      地址偏移：0x10          */
 __IO  uint32_t ODR;       /*GPIO 输出数据寄存器      地址偏移：0x14          */
```

(下页继续)

(续上页)

```
  __IO  uint16_t BSRRL;     /*GPIO 置位/复位寄存器 低 16 位部分  地址偏移: 0x18  */
  __IO  uint16_t BSRRH;     /*GPIO 置位/复位寄存器 高 16 位部分  地址偏移: 0x1A  */
  __IO  uint32_t LCKR;      /*GPIO 配置锁定寄存器        地址偏移: 0x1C          */
  __IO  uint32_t AFR[2];    /*GPIO 复用功能配置寄存器     地址偏移: 0x20-0x24     */
} GPIO_TypeDef;

/*RCC 寄存器列表 */
typedef struct {
  __IO  uint32_t CR;            /*!< RCC 时钟控制寄存器,          地址偏移: 0x00 */
  __IO  uint32_t PLLCFGR;       /*!< RCC PLL 配置寄存器,          地址偏移: 0x04 */
  __IO  uint32_t CFGR;          /*!< RCC 时钟配置寄存器,          地址偏移: 0x08 */
  __IO  uint32_t CIR;           /*!< RCC 时钟中断寄存器,          地址偏移: 0x0C */
  __IO  uint32_t AHB1RSTR;      /*!< RCC AHB1 外设复位寄存器,     地址偏移: 0x10 */
  __IO  uint32_t AHB2RSTR;      /*!< RCC AHB2 外设复位寄存器,     地址偏移: 0x14 */
  __IO  uint32_t AHB3RSTR;      /*!< RCC AHB3 外设复位寄存器,     地址偏移: 0x18 */
  __IO  uint32_t RESERVED0;     /*!< 保留,                        地址偏移: 0x1C */
  __IO  uint32_t APB1RSTR;      /*!< RCC APB1 外设复位寄存器,     地址偏移: 0x20 */
  __IO  uint32_t APB2RSTR;      /*!< RCC APB2 外设复位寄存器,     地址偏移: 0x24 */
  __IO  uint32_t RESERVED1[2];  /*!< 保留,                        地址偏移: 0x28-
 ↪0x2C*/
  __IO  uint32_t AHB1ENR;       /*!< RCC AHB1 外设时钟寄存器,     地址偏移: 0x30 */
  __IO  uint32_t AHB2ENR;       /*!< RCC AHB2 外设时钟寄存器,     地址偏移: 0x34 */
  __IO  uint32_t AHB3ENR;       /*!< RCC AHB3 外设时钟寄存器,     地址偏移: 0x38 */
/*RCC 后面还有很多寄存器,此处省略 */
} RCC_TypeDef;

/* 定义 GPIOA-H 寄存器结构体指针 */
#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)
```

(下页继续)

(续上页)

```
#define GPIOF                   ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG                   ((GPIO_TypeDef *) GPIOG_BASE)
#define GPIOH                   ((GPIO_TypeDef *) GPIOH_BASE)

/* 定义 RCC 外设 寄存器结构体指针 */
#define RCC                     ((RCC_TypeDef *) RCC_BASE)
```

这段代码在每个结构体成员前增加了一个“__IO”前缀，它的原型在这段代码的开头，代表了 C 语言中的关键字“volatile”，在 C 语言中该关键字用于修饰易变的变量，要求编译器不要优化。这些结构体内的成员，都代表着寄存器，而寄存器很多时候是由外设或 STM32 芯片状态修改的，也就是说即使 CPU 不执行代码修改这些变量，变量的值也有可能被外设修改、更新，所以每次使用这些变量的时候，我们都要求 CPU 去该变量的地址重新访问。若没有这个关键字修饰，在某些情况下，编译器认为没有代码修改该变量，就直接从 CPU 的某个缓存获取该变量值，这时可以加快执行速度，但该缓存中的是陈旧数据，与我们要求的寄存器最新状态可能会有出入。

8.3.2 定义访问外设的结构体指针

以结构体的形式定义好了外设寄存器后，使用结构体前还需要给结构体的首地址赋值，才能访问到需要的寄存器。为方便操作，我们给每个外设都定义好指向它地址的结构体指针，具体见代码清单

```
/* 定义 GPIOA-H 寄存器结构体指针 */
#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)
/* 定义 RCC 外设 寄存器结构体指针 */
#define RCC                     ((RCC_TypeDef *) RCC_BASE)
```

这些宏通过强制类型转换把外设的基地址转换成 GPIO_TypeDef 类型的指针，从而得到 GPIOA、GPIOB 等直接指向对应外设的指针，通过对结构体指针的操作，即可访问对应外设的寄存器。

利用这些指针访问寄存器，我们把 main 文件里对应的代码修改掉，具体见代码清单

```
/*
* 该章节代码是从 使用寄存器点亮 LED 灯章节代码修改过来
* 属于循序渐进的一章，每一个步骤都对应一部分代码，所以
* 该章节代码有好多个部分，不同部分都是使用条件编译来控制
* 最后只保留了最后的代码部分。
*
* 条件编译跟 C 语言的 if else 类似，只有当条件为真的情况下的
* 代码才会被编译，所以不要看到有多个 main 函数就以为不正确，
* 其实最终编译的只有一个
*/

#if 0
/* 使用寄存器点亮 LED 灯 章节代码
现在已经使用条件编译注释掉 */
int main(void)
{
 /* 开启 GPIOF 时钟，使用外设时都要先开启它的时钟 */
 RCC_AHB1ENR |= (1<<5);

 /* LED 端口初始化 */

 /*GPIOF MODER6 清空 */
 GPIOF_MODER  &= ~( 0x03<< (2*6));
 /*PF6 MODER6 = 01b 输出模式 */
 GPIOF_MODER |= (1<<2*6);

 /*GPIOF OTYPER6 清空 */
 GPIOF_OTYPER &= ~(1<<1*6);
 /*PF6 OTYPER6 = 0b 推挽模式 */
```

(下页继续)

(续上页)

```
 GPIOF_OTYPER |= (0<<1*6);

 /*GPIOF OSPEEDR6 清空 */
 GPIOF_OSPEEDR &= ~(0x03<<2*6);
 /*PF6 OSPEEDR6 = 0b 速率 2MHz*/
 GPIOF_OSPEEDR |= (0<<2*6);

 /*GPIOF PUPDR6 清空 */
 GPIOF_PUPDR &= ~(0x03<<2*6);
 /*PF6 PUPDR6 = 01b 上拉模式 */
 GPIOF_PUPDR |= (1<<2*6);

 /*PF6 BSRR 寄存器的 BR6 置 1，使引脚输出低电平 */
 GPIOF_BSRR |= (1<<16<<6);

 /*PF6 BSRR 寄存器的 BS6 置 1，使引脚输出高电平 */
 //GPIOF_BSRR |= (1<<6);

 while (1);
}

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

#elif 1

/* 自己写库—构建库函数雏形 章节代码
现在已经使用条件编译注释掉 */
int main(void)
{
```

(下页继续)

(续上页)

```
/* 开启 GPIOF 时钟，使用外设时都要先开启它的时钟 */
RCC->AHB1ENR |= (1<<5);

/* LED 端口初始化 */

/*GPIOF MODER6 清空 */
GPIOF->MODER  &= ~( 0x03<< (2*6));
/*PF6 MODER6 = 01b 输出模式 */
GPIOF->MODER |= (1<<2*6);

/*GPIOF OTYPER6 清空 */
GPIOF->OTYPER &= ~(1<<1*6);
/*PF6 OTYPER6 = 0b 推挽模式 */
GPIOF->OTYPER |= (0<<1*6);

/*GPIOF OSPEEDR6 清空 */
GPIOF->OSPEEDR &= ~(0x03<<2*6);
/*PF6 OSPEEDR6 = 0b 速率 2MHz*/
GPIOF->OSPEEDR |= (0<<2*6);

/*GPIOF PUPDR6 清空 */
GPIOF->PUPDR &= ~(0x03<<2*6);
/*PF6 PUPDR6 = 01b 上拉模式 */
GPIOF->PUPDR |= (1<<2*6);

/*PF6 BSRR 寄存器的 BR6 置 1，使引脚输出低电平 */
GPIOF->BSRRH |= (1<<16<<6);

/*PF6 BSRR 寄存器的 BS6 置 1，使引脚输出高电平 */
//GPIOF->BSRRL |= (1<<6);

while (1);
```

(下页继续)

(续上页)

```
}

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

乍一看，除了把寄存器名字的“-”改成了“->”之外，其他都没有变，但就是这个小变化里面蕴藏这大变化，带“-”的操作是直接操作内存，需要对寄存器的地址一个个进行定义，带“->”的操作是使用外设结构体指针的方式来操作，这为我们继续编写库函数打下了基础。还有一个不同是我们把 BSRR 寄存器分成 BSRRH 和 BSRRL 两段，各 16 位，高 16 位控制复位，低 16 位控制置位，都是写 1 有效。

打好了地基，下面我们就来建高楼。接下来使用函数来封装 GPIO 的基本操作，方便以后应用的时候不需要再查询寄存器，而是直接通过调用这里定义的函数来实现。我们把针对 GPIO 外设操作的函数及其宏定义分别存放在“stm32h7xx_hal_gpio.c”和“stm32h7xx_hal_gpio.h”文件中。

8.3.3 定义位操作函数

在“stm32f4xx_hal_gpio.c”文件定义两个位操作函数，分别用于控制引脚输出高电平和低电平，见代码清单 8_3。

列表 2: 代码清单 8-3 GPIO 置位函数与复位函数的定义

```
/**
* 函数功能：设置引脚电平
* 参数说明：GPIOx：该参数为 GPIO_TypeDef 类型的指针，指向 GPIO 端口的地址
*       GPIO_Pin：选择要设置的 GPIO 端口引脚，可输入宏 GPIO_Pin_0-15，
*             表示 GPIOx 端口的 0-15 号引脚
*       PinState：设置所选引脚的电平
*           @arg GPIO_PIN_RESET：设置低电平
```

(下页继续)

(续上页)

```
*             @arg GPIO_PIN_SET: 设置高电平
* 返回值:  无
*/
void HAL_GPIO_WritePin( GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_
↪PinState PinState)
{
 /* 设置 GPIOx 端口 BSRR 寄存器的低 16 位对应第 GPIO_Pin 位，使其输出高电平 */
 /* 设置 GPIOx 端口 BSRR 寄存器的高 16 位对应第 GPIO_Pin 位，使其输出低电平 */
 /* 因为 BSRR 寄存器写 0 不影响，GPIO_Pin 只是对应位为 1，其它位均为 0，所以可以直接赋
值 */

 if (PinState != GPIO_PIN_RESET) {
     GPIOx->BSRRL = GPIO_Pin;
 } else {
     GPIOx->BSRRH = GPIO_Pin ;
 }
}
```

函数体内根据需要的电平，对 GPIOx 的 BSRR 寄存器低 16 位或者高 16 位赋值，从而设置引脚为高电平或低电平。其中 GPIOx 是一个指针变量，通过函数的输入参数我们可以修改它的值，如给它赋予 GPIOA、GPIOB、GPIOH 等结构体指针值，这个函数就可以控制相应的 GPIOA、GPIOB、GPIOH 等端口的输出。

对比我们前面对 BSRR 寄存器的赋值，都是用“|=”操作来防止对其它数据位产生干扰的，为何此函数里的操作却直接用“=”号赋值，这样不怕干扰其它数据位吗？见代码清单 8_4。

列表 3: 代码清单 8-4 赋值方式对比

```
/* 使用 “|=” 来赋值 */
GPIOF->BSRR |= (1<<16<<6);
/* 直接使用 "=" 号赋值 */
GPIOF->BSRR = GPIO_Pin;
```

根据 BSRR 寄存器的特性，对它的数据位写 “0”，是不会影响输出的，只有对它的数据位写 “1”，才会控制引脚输出。对低 16 位写 “1” 输出高电平，对高 16 位写 “1” 输出低电平。也就是说，假如我们对 BSRR(高 16 位) 直接用 “=” 操作赋二进制值 “0000 0000 0000 0001 b”，它会控制 GPIO 的引脚 0 输出低电平，赋二进制值 “0000 0000 0001 0000 b”，它会控制 GPIO 引脚 4 输出低电平，而其它数据位由于是 0，所以不会受到干扰。同理，对 BSRR (低 16 位) 直接赋值也是如此，数据位为 1 的位输出高电平。代码清单 8_5 中的两种方式赋值，功能相同。

列表 4: 代码清单 8-5 BSRR 寄存器赋值等效代码

```
/* 使用 “|=” 来赋值 */
GPIOF->BSRR |= (uint16_t)(1<<6);
/* 直接使用 “=” 来赋值，二进制数 (0000 0000 1000 0000)*/
GPIOF->BSRR =  (uint16_t)(1<<6);
```

这两行代码功能等效，都把 BSRR 的 bit10 设置为 1，控制引脚 10 输出低电平，且其它引脚状态不变。但第二个语句操作效率是比较高的，因为 “|=” 号包含了读写操作，而 “=” 号只需要一个写操作。因此在定义位操作函数中我们使用后者。

利用这两个位操作函数，就可以方便地操作各种 GPIO 的引脚电平了，控制各种端口引脚的范例见代码清单 8_6。

列表 5: 代码清单 8-6 位操作函数使用范例

```
/* 控制 GPIOF 的引脚 6 输出高电平 */
HAL_GPIO_WritePin(GPIOF,(uint16_t)(1<<6), GPIO_PIN_SET);
/* 控制 GPIOF 的引脚 6 输出低电平 */
```

(下页继续)

(续上页)

```
HAL_GPIO_WritePin(GPIOF,(uint16_t)(1<<6), GPIO_PIN_RESET);

/* 控制 GPIOF 的引脚 7、引脚 8 输出高电平，使用“|”同时控制多个引脚 */
HAL_GPIO_WritePin(GPIOF,(uint16_t)(1<<7)| (uint16_t)(1<<8), GPIO_PIN_SET);
/* 控制 GPIOF 的引脚 7、引脚 8 输出低电平 */
HAL_GPIO_WritePin(GPIOF,(uint16_t)(1<<7)| (uint16_t)(1<<8), GPIO_PIN_RESET);

/* 控制 GPIOF 的引脚 7 输出高电平 */
HAL_GPIO_WritePin(GPIOF,(uint16_t)(1<<7), GPIO_PIN_SET);
/* 控制 GPIOF 的引脚 8 输出低电平 */
HAL_GPIO_WritePin(GPIOF,(uint16_t)(1<<8), GPIO_PIN_RESET);
```

使用以上函数输入参数，设置引脚号时，还是很不方便，且可读性差，为此我们把表示 16 个引脚号的操作数都定义成宏，具体见代码清单 *a*

```
/*GPIO 引脚号定义 */
#define GPIO_PIN_0                 (uint16_t)0x0001)  /*!< 选择 Pin0 (1<<0) */
#define GPIO_PIN_1                 ((uint16_t)0x0002)  /*!< 选择 Pin1 (1<<1)*/
#define GPIO_PIN_2                 ((uint16_t)0x0004)  /*!< 选择 Pin2 (1<<2)*/
#define GPIO_PIN_3                 ((uint16_t)0x0008)  /*!< 选择 Pin3 (1<<3)*/
#define GPIO_PIN_4                 ((uint16_t)0x0010)  /*!< 选择 Pin4 */
#define GPIO_PIN_5                 ((uint16_t)0x0020)  /*!< 选择 Pin5 */
#define GPIO_PIN_6                 ((uint16_t)0x0040)  /*!< 选择 Pin6 */
#define GPIO_PIN_7                 ((uint16_t)0x0080)  /*!< 选择 Pin7 */
#define GPIO_PIN_8                 ((uint16_t)0x0100)  /*!< 选择 Pin8 */
#define GPIO_PIN_9                 ((uint16_t)0x0200)  /*!< 选择 Pin9 */
#define GPIO_PIN_10                ((uint16_t)0x0400)  /*!< 选择 Pin10 */
#define GPIO_PIN_11                ((uint16_t)0x0800)  /*!< 选择 Pin11 */
#define GPIO_PIN_12                ((uint16_t)0x1000)  /*!< 选择 Pin12 */
#define GPIO_PIN_13                ((uint16_t)0x2000)  /*!< 选择 Pin13 */
#define GPIO_PIN_14                ((uint16_t)0x4000)  /*!< 选择 Pin14 */
#define GPIO_PIN_15                ((uint16_t)0x8000)  /*!< 选择 Pin15 */
```

(下页继续)

(续上页)

```
#define GPIO_PIN_All                ((uint16_t)0xFFFF)  /*!< 选择全部引脚 */
```

这些宏代表的参数是某位置“1”其它位置“0”的数值，其中最后一个“GPIO_PIN_ALL”是所有数据位都为“1”，所以用它可以一次控制设置整个端口的 0-15 所有引脚。利用这些宏，GPIO 的控制代码可改为代码清单

```
/* 控制 GPIOF 的引脚 6 输出高电平 */
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_6, GPIO_PIN_SET);
/* 控制 GPIOF 的引脚 6 输出低电平 */
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_6, GPIO_PIN_RESET);

/* 控制 GPIOF 的引脚 7、引脚 8 输出高电平，使用“|”，同时控制多个引脚 */
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_7| GPIO_PIN_8, GPIO_PIN_SET);
/* 控制 GPIOF 的引脚 7、引脚 8 输出低电平 */
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_7| GPIO_PIN_8, GPIO_PIN_RESET);
/* 控制 GPIOF 的所有输出低电平 */
HAL_GPIO_WritePin(GPIOF, GPIO_PIN_All, GPIO_PIN_RESET);

/* 控制 GPIOA 的引脚 8 输出高电平 */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_SET);
/* 控制 GPIOB 的引脚 9 输出低电平 */
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, GPIO_PIN_RESET);
```

使用以上代码控制 GPIO，我们就不需要再看寄存器了，直接从函数名和输入参数就可以直观看出这个语句要实现什么操作 (英文中“Set”表示“置位”，即高电平，“Reset”表示“复位”，即低电平)。

8.3.4 定义初始化结构体 GPIO_InitTypeDef

定义位操作函数后，控制 GPIO 输出电平的代码得到了简化，但在控制 GPIO 输出电平前还需要初始化 GPIO 引脚的各种模式，这部分代码涉及的寄存器有很多，我们希望初始化 GPIO 也能以如此简单的方法去实现。为此，我们先根据 GPIO 初始化时涉及到的初始化参数以结构体的形式封装起来，声明一个名为 GPIO_InitTypeDef 的结构体类型，见*代码 8_3*

列表 6: 代码 8-3 定义 GPIO 初始化结构体

```
/**
* @brief GPIO 初始化结构体类型定义
*/
typedef struct {
 uint32_t Pin;        /* 指定要配置的 GPIO 引脚 */

 uint32_t Mode;       /* 指定所选引脚的工作模式 */

 uint32_t Pull;       /* 指定所选引脚的上拉或下拉激活 */

 uint32_t Speed;      /* 指定所选引脚的速度 */

 uint32_t Alternate;  /* 要连接到所选引脚的外设 */
} GPIO_InitTypeDef;
```

这个结构体中包含了初始化 GPIO 所需要的信息，包括引脚号、工作模式、输出速率、输出类型以及上/下拉模式。设计这个结构体的思路是：初始化 GPIO 前，先定义一个这样的结构体变量，根据需要配置 GPIO 的模式，对这个结构体的各个成员进行赋值，然后把这个变量作为“GPIO 初始化函数”的输入参数，该函数能根据这个变量值中的内容去配置寄存器，从而实现 GPIO 的初始化。

8.3.5 定义引脚模式的枚举类型

上面定义的结构体很直接，美中不足的是在对结构体中各个成员赋值时还需要看具体哪个模式对应哪个数值，如 GPIO_Mode 成员的“输入/输出/复用/模拟”模式对应二进制值“00 、01、10、11”，我们不希望每次用到都要去查找这些索引值，所以使用 C 语言中的枚举语法定义这些参数，具体见代码 8_4。

列表 7: 代码 8-4 GPIO 配置参数的枚举定义

```
#define  GPIO_MODE_INPUT        ((uint32_t)0x00000000U)   /*!< 浮空输入 */
#define  GPIO_MODE_OUTPUT_PP    ((uint32_t)0x00000001U)   /*!< 推挽输出 */
#define  GPIO_MODE_OUTPUT_OD    ((uint32_t)0x00000011U)   /*!< 开漏输出 */
#define  GPIO_MODE_AF_PP     ((uint32_t)0x00000002U)   /*!< 推挽复用输出 */
#define  GPIO_MODE_AF_OD    ((uint32_t)0x00000012U)   /*!< 开漏复用输出 */

#define  GPIO_MODE_ANALOG      ((uint32_t)0x00000003U)   /*!< 模拟模式 */

#define  GPIO_SPEED_FREQ_LOW          ((uint32_t)0x00000000U)  /*!< 低速 */
#define  GPIO_SPEED_FREQ_MEDIUM       ((uint32_t)0x00000001U)  /*!< 中速 */
#define  GPIO_SPEED_FREQ_HIGH         ((uint32_t)0x00000002U)  /*!< 快速 */
#define  GPIO_SPEED_FREQ_VERY_HIGH    ((uint32_t)0x00000003U)  /*!< 高速 */

#define  GPIO_NOPULL         ((uint32_t)0x00000000U)   /*!< 无上下拉  */
#define  GPIO_PULLUP         ((uint32_t)0x00000001U)   /*!< 上拉    */
#define  GPIO_PULLDOWN       ((uint32_t)0x00000002U)   /*!< 下拉    */
```

有了这些枚举定义，我们的 GPIO_InitTypeDef 结构体也可以使用枚举类型来限定输入了，具体见代码

列表 8: 代码 8-4 使用枚举类型定义的 GPIO_InitTypeDef 结构体成员

```
/**
```

(下页继续)

(续上页)

```
* @brief GPIO 初始化结构体类型定义
*/
typedef struct {
 uint32_t Pin;        /* 指定要配置的 GPIO 引脚 */

 uint32_t Mode;       /* 指定所选引脚的工作模式 */

 uint32_t Pull;       /* 指定所选引脚的上拉或下拉激活 */

 uint32_t Speed;      /* 指定所选引脚的速度 */

 uint32_t Alternate;  /* 要连接到所选引脚的外设 */
} GPIO_InitTypeDef;
```

如果不使用枚举类型，仍使用“uint8_t”类型来定义结构体成员，那么成员值的范围就是 0-255 了，而实际上这些成员都只能输入几个数值。所以使用枚举类型可以对结构体成员起到限定输入的作用，只能输入相应已定义的枚举值。

利用这些枚举定义，给 GPIO_InitTypeDef 结构体类型赋值配置就非常直观了，具体见代码

列表 9: 代码 8-4 给 GPIO_InitTypeDef 初始化结构体赋值范例

```
GPIO_InitTypeDef InitStruct;

/* LED 端口初始化 */

/* 初始化 PF6 引脚 */
/* 选择要控制的 GPIO 引脚 */
InitStruct.Pin = GPIO_PIN_6;
/* 设置引脚的输出类型为推挽输出 */
InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
/* 设置引脚为上拉模式 */
```

(下页继续)

(续上页)

```
InitStruct.Pull = GPIO_PULLUP;
/* 设置引脚速率为低速模式 */
InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
/* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
HAL_GPIO_Init(GPIOF, &InitStruct);
```

8.3.6 定义 GPIO 初始化函数

接着前面的思路，对初始化结构体赋值后，把它输入到 GPIO 初始化函数，由它来实现寄存器配置。我们的 GPIO 初始化函数实现具体见*代码 8_5*

列表 10: 代码 8-5 GPIO 初始化函数

```
/**
* 函数功能：初始化引脚模式
* 参数说明：GPIOx，该参数为 GPIO_TypeDef 类型的指针，指向 GPIO 端口的地址
*          GPIO_InitTypeDef:GPIO_InitTypeDef 结构体指针，指向初始化变量
*/
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;

 /* Configure the port pins */
 for (position = 0; position < 16; position++) {
 /* 以下运算是为了通过 GPIO_InitStruct->GPIO_Pin 算出引脚号 0-15*/
 /* 经过运算后 pos 的 pinpos 位为 1，其余为 0，与 GPIO_Pin_x 宏对应。pinpos 变量每次循环加 1，*/
 ioposition = ((uint32_t)0x01) << position;
```

(下页继续)

(续上页)

```
 /* pos 与 GPIO_InitStruct->Pin 做 & 运算，若运算结果 currentpin == pos，
 则表示 GPIO_InitStruct->Pin 的 pinpos 位也为 1，
 从而可知 pinpos 就是 GPIO_InitStruct->Pin 对应的引脚号：0-15*/
 iocurrent = (uint32_t)(GPIO_Init->Pin) & ioposition;

 if (iocurrent == ioposition) {
 /*------------- GPIO Mode Configuration -----------------*/
 /* 在复用功能模式选择的情况下 */
 if ((GPIO_Init->Mode == GPIO_MODE_AF_PP) || (GPIO_Init->Mode == GPIO_MODE_
↪AF_OD)) {
 /* 配置与当前 IO 映射的备用功能 */
 temp = GPIOx->AFR[position >> 3];
 temp &= ~((uint32_t)0xF << ((uint32_t)(position & (uint32_t)0x07) * 4)) ;
 temp |= ((uint32_t)(GPIO_Init->Alternate) << (((uint32_t)position &␣
↪(uint32_t)0x07) * 4));
 GPIOx->AFR[position >> 3] = temp;
}

/* 配置 IO 方向模式（输入，输出，复用或模拟） */
temp = GPIOx->MODER;
temp &= ~(GPIO_MODER_MODER0 << (position * 2));
temp |= ((GPIO_Init->Mode & GPIO_MODE) << (position * 2));
GPIOx->MODER = temp;

/* 在输出或复用功能模式选择的情况下 */
if ((GPIO_Init->Mode == GPIO_MODE_OUTPUT_PP) || (GPIO_Init->Mode == GPIO_
↪MODE_AF_PP) ||
(GPIO_Init->Mode == GPIO_MODE_OUTPUT_OD) || (GPIO_Init->Mode == GPIO_MODE_
↪AF_OD)) {

 /* 配置速度参数 */
 temp = GPIOx->OSPEEDR;
```

(下页继续)

(续上页)

```
  temp &= ~(GPIO_OSPEEDER_OSPEEDR0 << (position * 2));
  temp |= (GPIO_Init->Speed << (position * 2));
  GPIOx->OSPEEDR = temp;

  /* 配置 IO 输出类型 */
  temp = GPIOx->OTYPER;
  temp &= ~(GPIO_OTYPER_OT_0 << position) ;
  temp |= (((GPIO_Init->Mode & GPIO_OUTPUT_TYPE) >> 4) << position);
  GPIOx->OTYPER = temp;
}

  /* 激活当前 IO 的上拉或下拉电阻 */
  temp = GPIOx->PUPDR;
  temp &= ~(GPIO_PUPDR_PUPDR0 << (position * 2));
  temp |= ((GPIO_Init->Pull) << (position * 2));
  GPIOx->PUPDR = temp;
}
}
}
```

这个函数有 GPIOx 和 GPIO_InitStruct 两个输入参数，分别是 GPIO 外设指针和 GPIO 初始化结构体指针。分别用来指定要初始化的 GPIO 端口及引脚的工作模式。

函数实现主要分两个环节：

(1) 利用 for 循环，根据 GPIO_InitStruct 的结构体成员 GPIO_PIN 计算出要初始化的引脚号。这段看起来复杂的运算实际上可以这样理解：它要通过宏“GPIO_PIN_x”的参数计算出 x 值 (宏的参数值是第 x 数据位为 1，其余为 0，参考代码清单 *a*)，计算得的引脚号结果存储在 pinpos 变量中。

(2) 得到引脚号 pinpos 后，利用初始化结构体各个成员的值，对相应寄存器进行配置，这部分与我们前面直接配置寄存器的操作是类似的，先对引脚号 pinpos 相应的配置位清空，后根据结构体成员对配置位赋值 (GPIO_Mode 成员对应 MODER 寄存器的配置，GPIO_PuPd 成员对应 PUPDR 寄存器的配置等)。区别是这里的寄存器配置值及引脚号都是由变量存储的。

8.3.7 全新面貌，使用函数点亮 LED 灯

完成以上的准备后，我们就可以用自己定义的函数来点亮 LED 灯，见*代码 8_7* 使用函数点亮 LED

列表 11: 代码 8_7 使用函数点亮 LED

```
void Delay( uint32_t nCount);
/**
  *   主函数
  */
int main(void)
{
   GPIO_InitTypeDef InitStruct;

   /* 开启 GPIOHF 时钟，使用外设时都要先开启它的时钟 */
    RCC->AHB1ENR |= (1<<5);

    /* LED 端口初始化 */

    /* 初始化 PH6 引脚 */
    /* 选择要控制的 GPIO 引脚 */
    InitStruct.Pin = GPIO_PIN_6;
    /* 设置引脚的输出类型为推挽输出 */
    InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    /* 设置引脚为上拉模式 */
    InitStruct.Pull = GPIO_PULLUP;
    /* 设置引脚速率为低速模式 */
    InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    /* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
    HAL_GPIO_Init(GPIOF, &InitStruct);

    /* 使引脚输出低电平，点亮 LED1*/
    HAL_GPIO_WritePin(GPIOF,GPIO_PIN_6,GPIO_PIN_RESET);
```

(下页继续)

(续上页)

```
    /* 延时一段时间 */
    Delay(0xFFFFFF);

    /* 使引脚输出高电平，关闭 LED1*/
    HAL_GPIO_WritePin(GPIOF,GPIO_PIN_6,GPIO_PIN_SET);

    /* 初始化 PH11 引脚 */
    InitStruct.Pin = GPIO_PIN_7;
    HAL_GPIO_Init(GPIOF,&InitStruct);

    /* 使引脚输出低电平，点亮 LED2*/
    HAL_GPIO_WritePin(GPIOF,GPIO_PIN_7,GPIO_PIN_RESET);

    while (1);

}

//简单的延时函数，让 cpu 执行无意义指令，消耗时间
//具体延时时间难以计算，以后我们可使用定时器精确延时
void Delay( uint32_t nCount)
{
    for (; nCount != 0; nCount--);
}
// 函数为空，目的是为了骗过编译器不报错
void SystemInit(void)
{
}
```

现在看起来，使用函数来控制 LED 灯与之前直接控制寄存器已经有了很大的区别：main 函数中先定义了一个初始化结构体变量 InitStruct，然后对该变量的各个成员按点亮 LED 灯所需要的 GPIO 配置模式进行赋值，赋值后，调用 HAL_GPIO_Init 函数，让它根据结构体成员值对 GPIO

寄存器写入控制参数，完成 GPIO 引脚初始化。控制电平时，直接使用 HAL_GPIO_WritePin 函数控制输出。如若对其它引脚进行不同模式的初始化，只要修改初始化结构体 InitStruct 的成员值，把新的参数值输入到 HAL_GPIO_Init 函数再调用即可。

代码中新增的 Delay 函数，主要功能是延时，让我们可以看清楚实验现象 (不延时的话指令执行太快，肉眼看不出来)，它的实现原理是让 CPU 执行无意义的指令，消耗时间，在此不要纠结它的延时时间，写一个大概输入参数值，下载到实验板实测，觉得太久了就把参数值改小，短了就改大即可。需要精确延时的时候我们会用 STM32 的定时器外设进行精确延时的。

8.3.8 下载验证

把编译好的程序下载到开发板并复位，可看到板子上的灯先亮红色 (LED1)，后亮绿色 (LED2)。

8.3.9 总结

什么是 ST 函数库？我们上面写的就是，相对于官方的完整版的函数库，我们写的只是一个雏形，写这个函数库的雏形目的是为了帮助我们从寄存器编程顺利地过度到到函数库编程的，让我们知道什么是函数库，为以后能够熟练的使用函数库编程打下基础。

我们从寄存器映射开始，把内存跟寄存器建立起一一对应的关系，然后操作寄存器点亮 LED，再把寄存器操作封装成一个个函数。一步一步走来，我们实现了库最简单的雏形，如果我们不断地增加操作外设的函数，并且把所有的外设都写完，一个完整的库就实现了。

本章中的 GPIO 相关库函数及结构体定义，实际上都是从 ST 的 HAL 库搬过来的。这样分析它纯粹是为了满足自己的求知欲，学习其编程的方式、思想，这对提高我们的编程水平是很有好处的，顺便感受一下 ST 库设计的严谨性，我认为这样的代码不仅严谨且华丽优美，不知您是否也有这样的感受。

与直接配置寄存器相比，从执行效率上看会有额外的消耗：初始化变量赋值的过程、库函数在被调用的时候要耗费调用时间；在函数内部，对输入参数转换所需要的额外运算也消耗一些时间 (如 GPIO 中运算求出引脚号时)。而其它的宏、枚举等解释操作是作编译过程完成的，这部分并不消耗内核的时间。那么函数库的优点呢？是我们可以快速上手 STM32 控制器；配置外设状态时，不需要再纠结要向寄存器写入什么数值；交流方便，查错简单。这就是我们选择库的原因。

现在的处理器的主频是越来越高，我们不需要担心 CPU 耗费那么多时间来干活会不会被累倒，库主要应用是在初始化过程，而初始化过程一般是芯片刚上电或在核心运算之前的执行的，这段时间的等待是 0.02us 还是 0.01us 在很多时候并没有什么区别。相对来说，我们还是担心一下如果都用寄存器操作，每行代码都要查《STM32F4xx 规格书》中的说明，自己会不会被累倒吧。

在以后开发的工程中，一般不会去分析 ST 的库函数的实现了。因为外设的库函数是很类似的，库外设都包含初始化结构体，以及特定的宏或枚举标识符，这些封装被库函数这些转化成相应的值，写入到寄存器之中，函数内部的具体实现是十分枯燥和机械的工作。如果您有兴趣，在您掌握了如何使用外设的库函数之后，可以查看一下它的源码实现。

通常我们只需要通过了解每种外设的“初始化结构体”就能够通过它去了解 STM32 的外设功能及控制了。

第 9 章　初识 HAL 固件库

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、《Cortex-M4 权威指南》。

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。

9.1 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*。

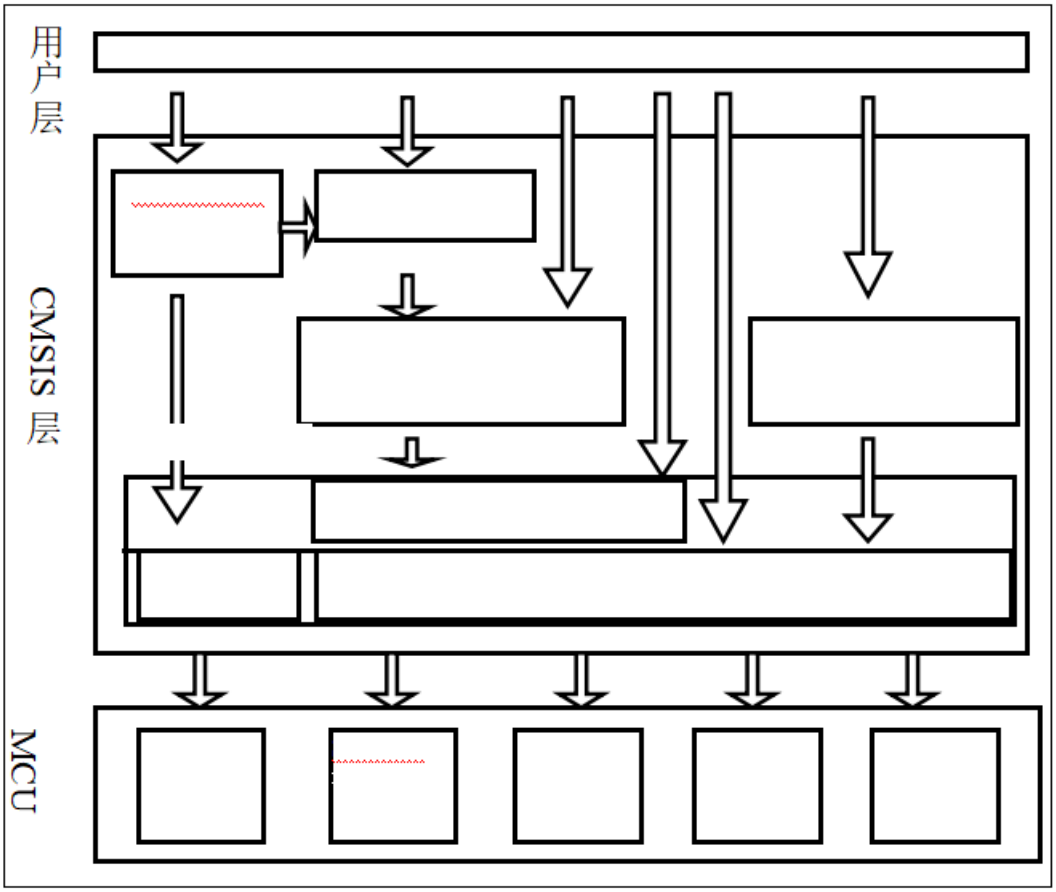


图 9- 1 CMSIS 架构

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。

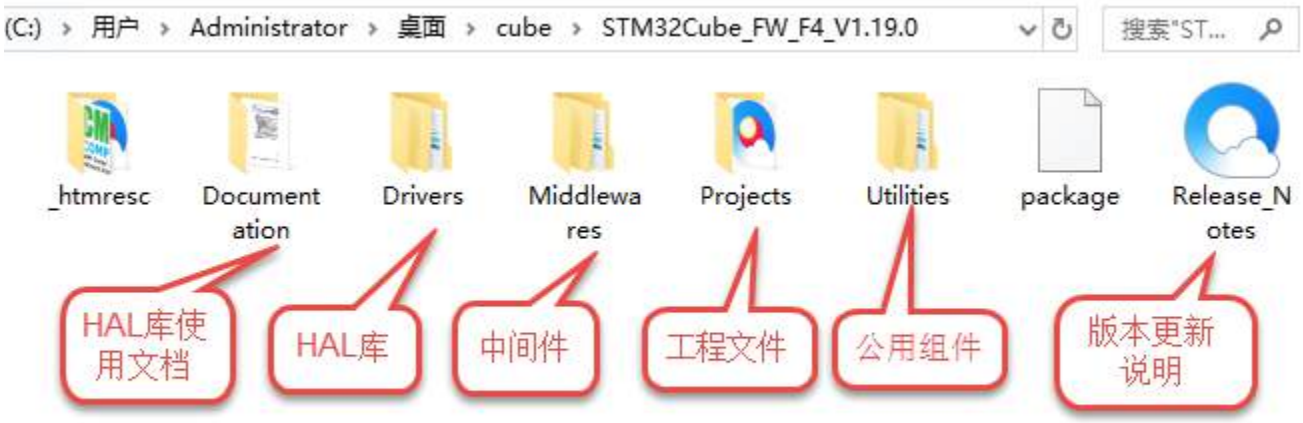
9.1.1 库目录、文件简介

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采用 1.5.1 库文件。以下内容请大家打开 STM32HAL 库文件配合阅读。

解压库文件后进入其目录：

“STM32Cube_FW_F4_V1.19.0”

软件库各文件夹的内容说明见图 *9_2*。



目录：STM32Cube_FW_F4_V1.19.0\

- 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、STM32F4xx_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_F4_V1.19.0DriversCMSIS 文件夹下内容见图 *9_3*。

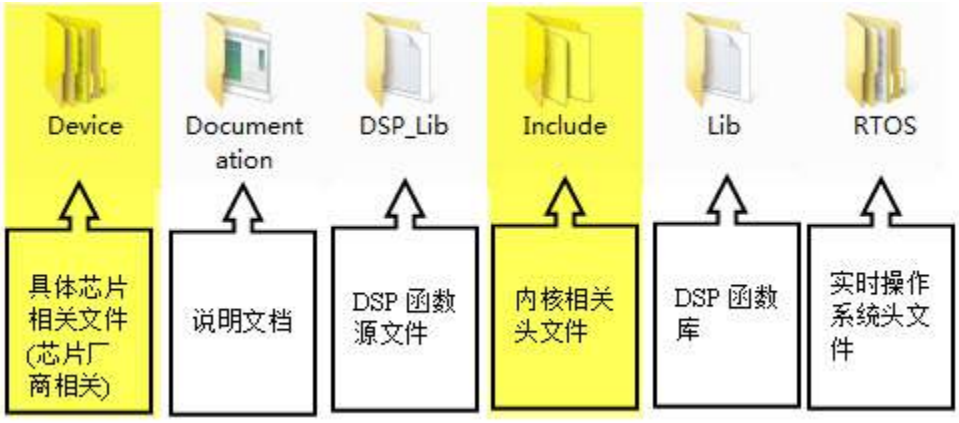


图 9-3 CMSIS 文件夹内容目录：Drivers\CMSIS\

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。

9.1.1.1 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 stm32F407xx.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_M4_Technical Reference Manual》及《Cortex®-M4 内核编程手册》文档，《STM32F407xxx 参考手册》只包含片上外设说明，不包含内核寄存器。

我们写 STM32F4 的工程，必须用到其中的四个文件：core_cm4.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_cM4.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. */
```

(下页继续)

(续上页)

```
        #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__ )

        #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_cm4.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 Stm32F407xx.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_cm4.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 CM4 芯片的库都带有这个文件，这样软件在不同的 CM4 芯片的移植工作就得以简化。

9.1.1.2 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_stm32f4xx.c 文件

文件目录：\ Drivers \CMSIS\Device\ST\stm32f4xx\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_stm32f4xx.c 文件。STM32F407 系列的芯片，调用库的这个 SystemInit 函数后，系统时钟被初始化为 168MHz，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\stm32f4xx\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_STM32F407xx.s”即为 STM32F407 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。如果使用其它型号的芯片，要在此处选择对应的启动文件，如 STM32F407 型号使用“startup_stm32F407xx.s”文件。

- stm32F407xx.h 文件

文件目录：Drivers \CMSIS\Device\ST\stm32f4xx\Include

stm32F407xx.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“stm32F407xx.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 STM32F4xx_HAL_Driver 文件夹。

9.1.1.3 STM32F4xx_HAL_Driver 文件夹

文件目录：Drivers\STM32F4xx_HAL_Driver

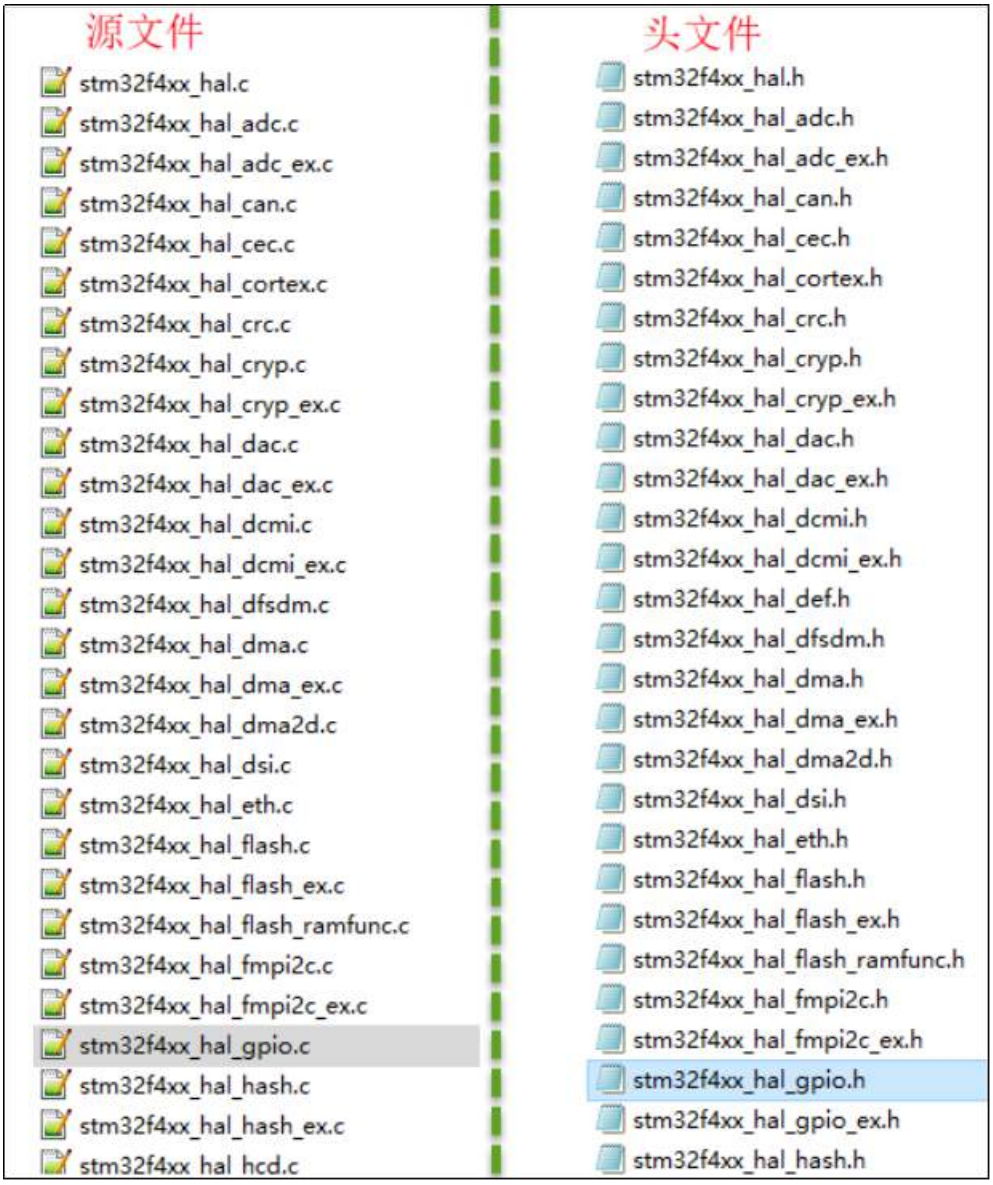
进入 Drivers 目录下的 STM32F4xx_HAL_Driver 文件夹，见图 *9_4*。



STM32F4xx_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：stm32f4xx_hal_ppp.c 或 stm32f4xx_hal_ppp.h 文件，PPP 表示外设名称。如在上一章中我们自建的 stm32f4xx_hal_gpio.c 及 stm32f4xx_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 stm32f4xx_hal_adc.c 源文件，在 inc 文件夹下有一个 stm32f4xx_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*。



9.1.1.4 stm32f4xx_it.c、stm32f4xx_hal_conf.h 文件

文件目录：STM32Cube_FW_F4_V1.19.0\Projects\STM32F407ZI-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 stm32f4xx_it.c 和 inc 文件夹中和 stm32f4xx_it.h、stm32f4xx_hal_conf.h 这三个文件。

stm32f4xx_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

stm32f4xx_hal_conf.h：这个文件被包含进 stm32f407xx.h 文件。ST HAL 库支持所有 STM32F4 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 "stm32f4xx_hal_rcc.h"
#endif /* HAL_RCC_MODULE_ENABLED */

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

#ifdef HAL_DMA_MODULE_ENABLED
   #include "stm32f4xx_hal_dma.h"
#endif /* HAL_DMA_MODULE_ENABLED */

#ifdef HAL_CORTEX_MODULE_ENABLED
   #include "stm32f4xx_hal_cortex.h"
#endif /* HAL_CORTEX_MODULE_ENABLED */
```

(下页继续)

(续上页)

```
#ifdef HAL_ADC_MODULE_ENABLED
   #include "stm32f4xx_hal_adc.h"
#endif /* HAL_ADC_MODULE_ENABLED */

#ifdef HAL_CAN_MODULE_ENABLED
   #include "stm32f4xx_hal_can.h"
#endif /* HAL_CAN_MODULE_ENABLED */
```

stm32f4xx_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。

9.1.2 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*。

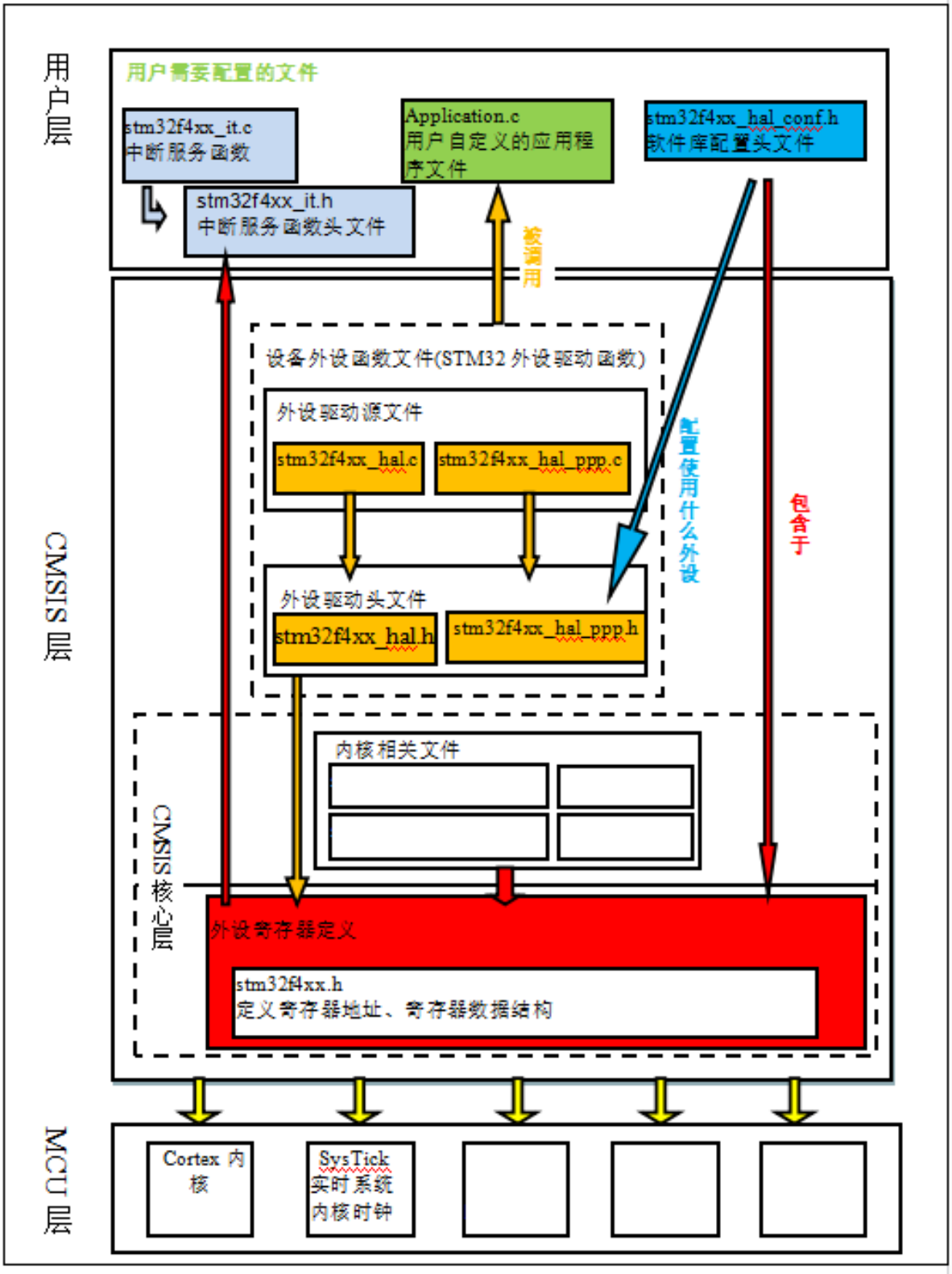

用户层
CMSIS层
MCU层
用户需要配置的文件
stm32f4xx_it.c
中断服务函数
stm32f4xx_it.h
中断服务函数头文件
Application.c
用户自定义的应用程序文件
stm32f4xx_hal_conf.h
软件库配置头文件
被调用
设备外设函数文件(STM32 外设驱动函数)
外设驱动源文件
stm32f4xx_hal.c
stm32f4xx_hal_ppp.c
外设驱动头文件
stm32f4xx_hal.h
stm32f4xx_hal_ppp.h
配置使用什么外设
包含于
内核相关文件
CMSIS核心层
外设寄存器定义
stm32f4xx.h
定义寄存器地址、寄存器数据结构
Cortex 内核
SysTick
实时系统
内核时钟

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_stm32f4xx.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。

9.2 使帮助文档

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。

9.2.1 常用官方资料

- 《STM32F4XX-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。

- 《STM32F4xx 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™-M4 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、中断等寄存器。这部分的内容是《STM32F4xx 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。

这两个手册是由 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，其中 Cortex-M3 版本有中文版，方便学习。因为 Cortex-M4 内核与 Cortex-M3 内核大部分相同，可用它来学习，而 Cortex-M4 新增的特性，则必须参考《cortex_M4_Technical Reference Manual》文档了，目前只有英文版。

- 《STM32F417xx_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 10 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，STM32CubeF4 是针对 STM32F4 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)。

10.1 软件安装

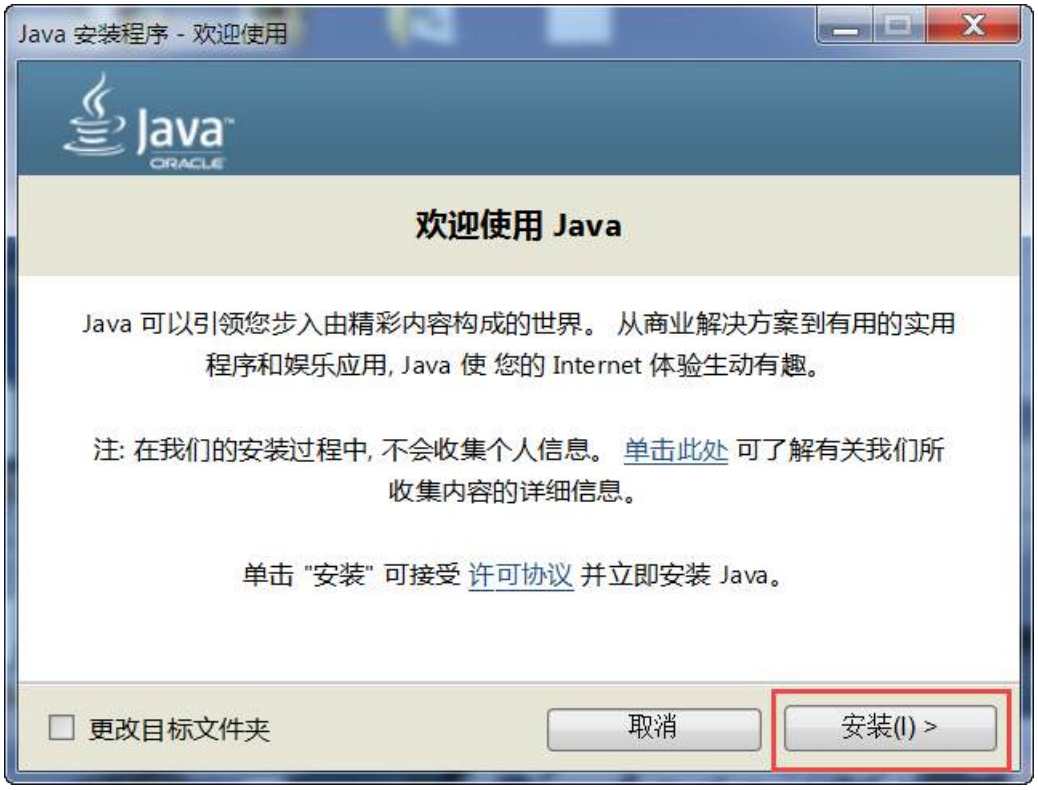
必备软件：Java（V1.7 及以上版本）和 STM32CubeMX（版本 5.3.0）

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)

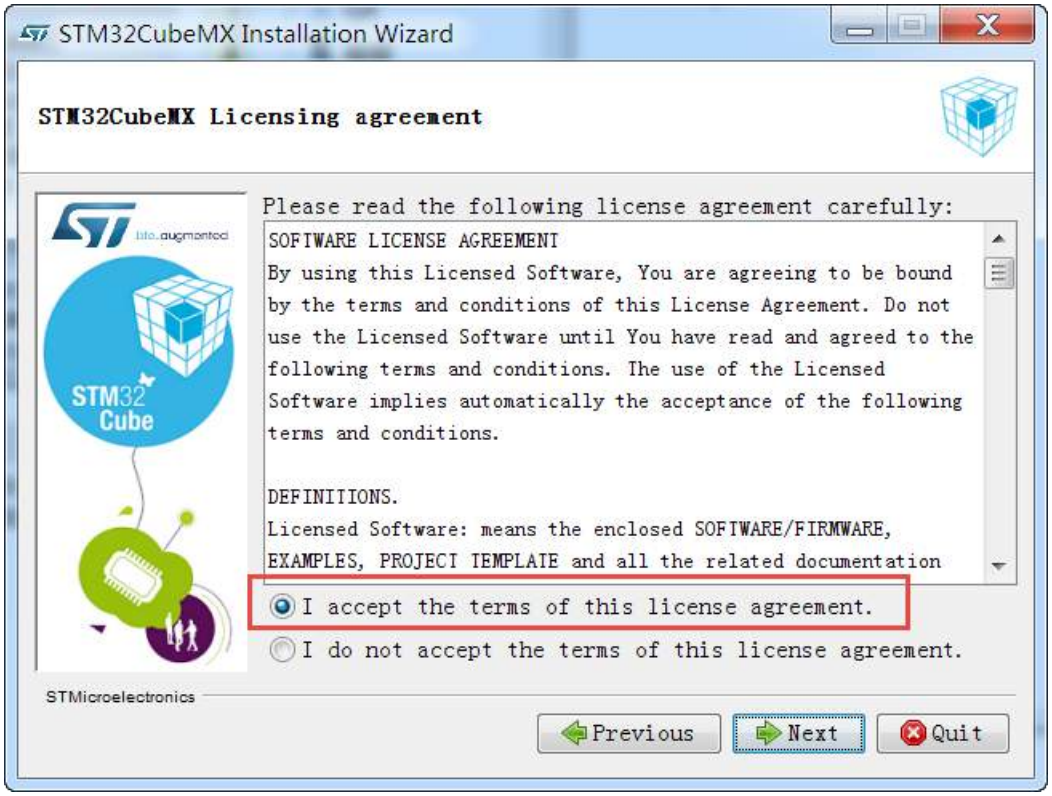


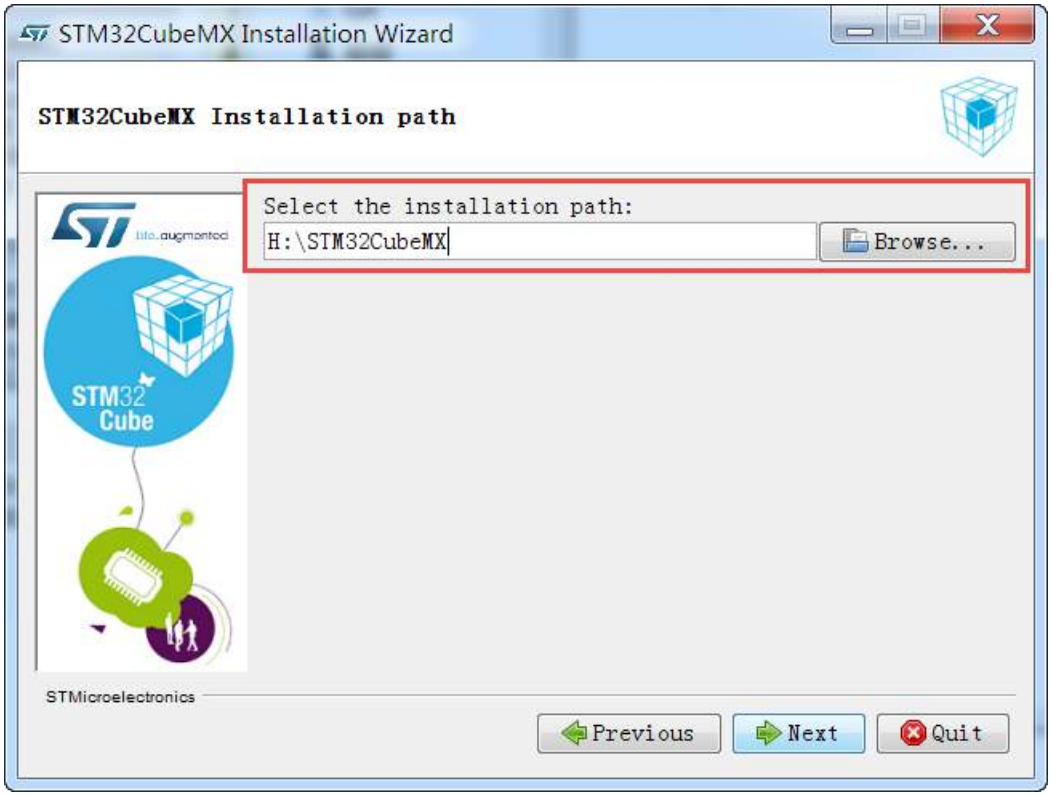
在本地新建好文件夹后，把准备好的库文件添加到相应的文件夹下：

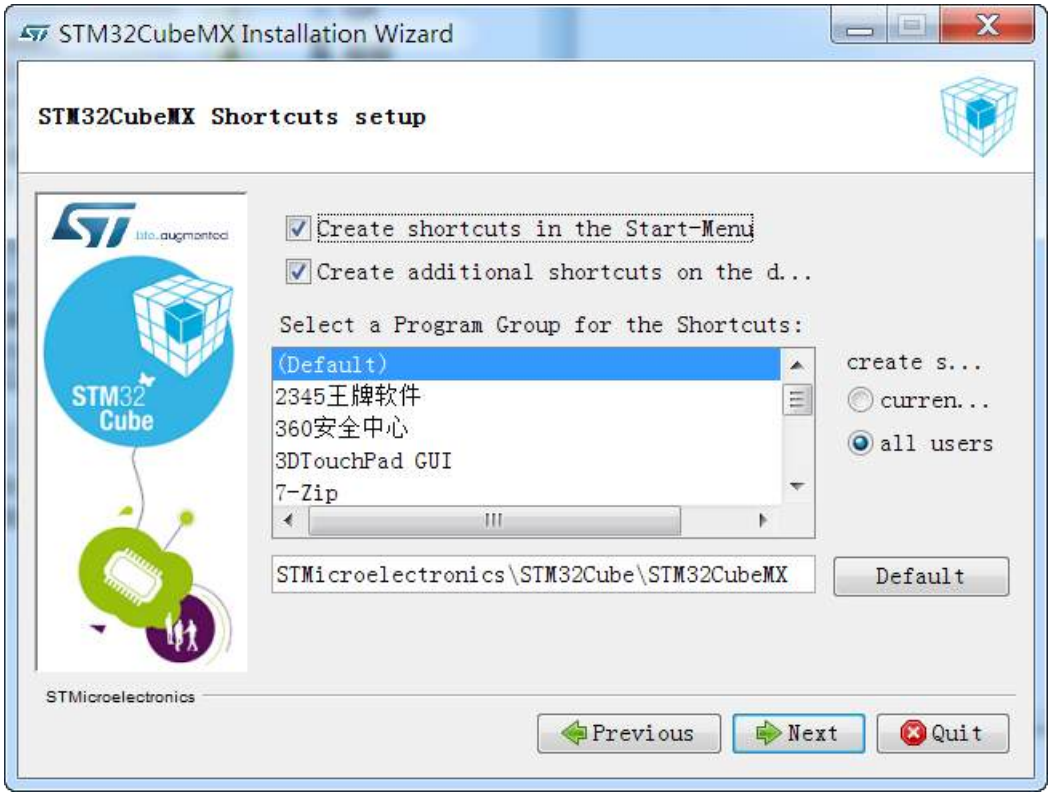
10.1.1 安装 STM32CubeMX 软件

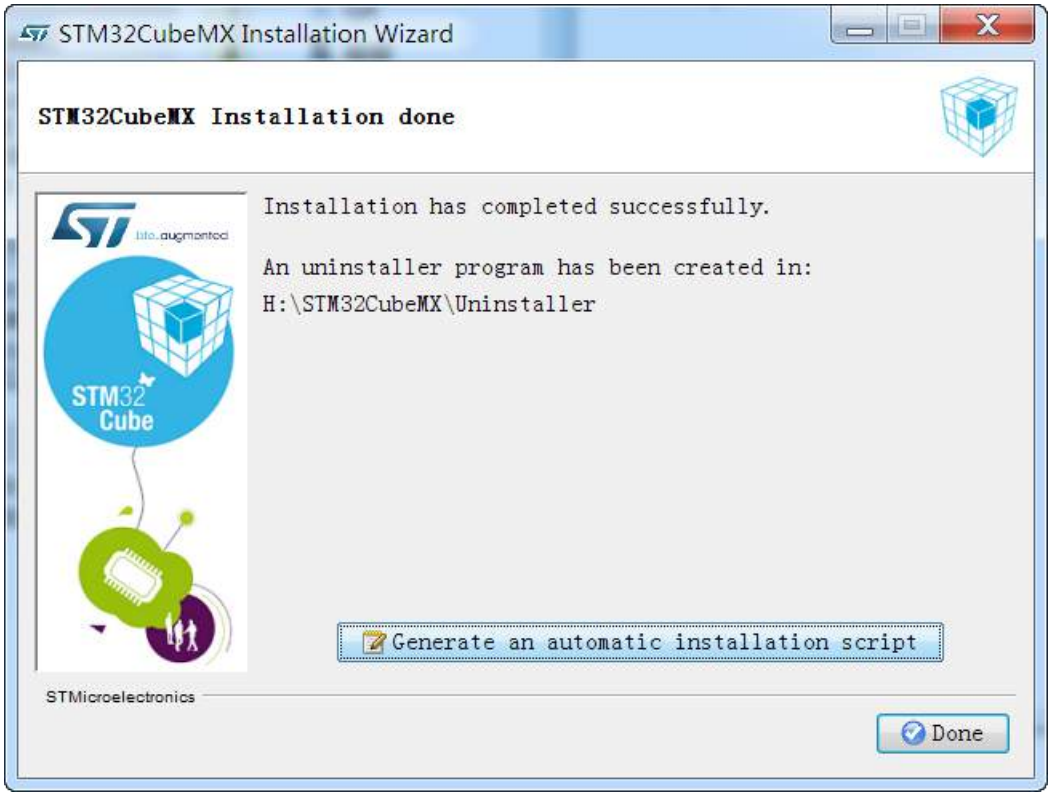
双击安装包“SetupSTM32CubeMX-5.3.0.exe”，具体操作步骤如下：


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

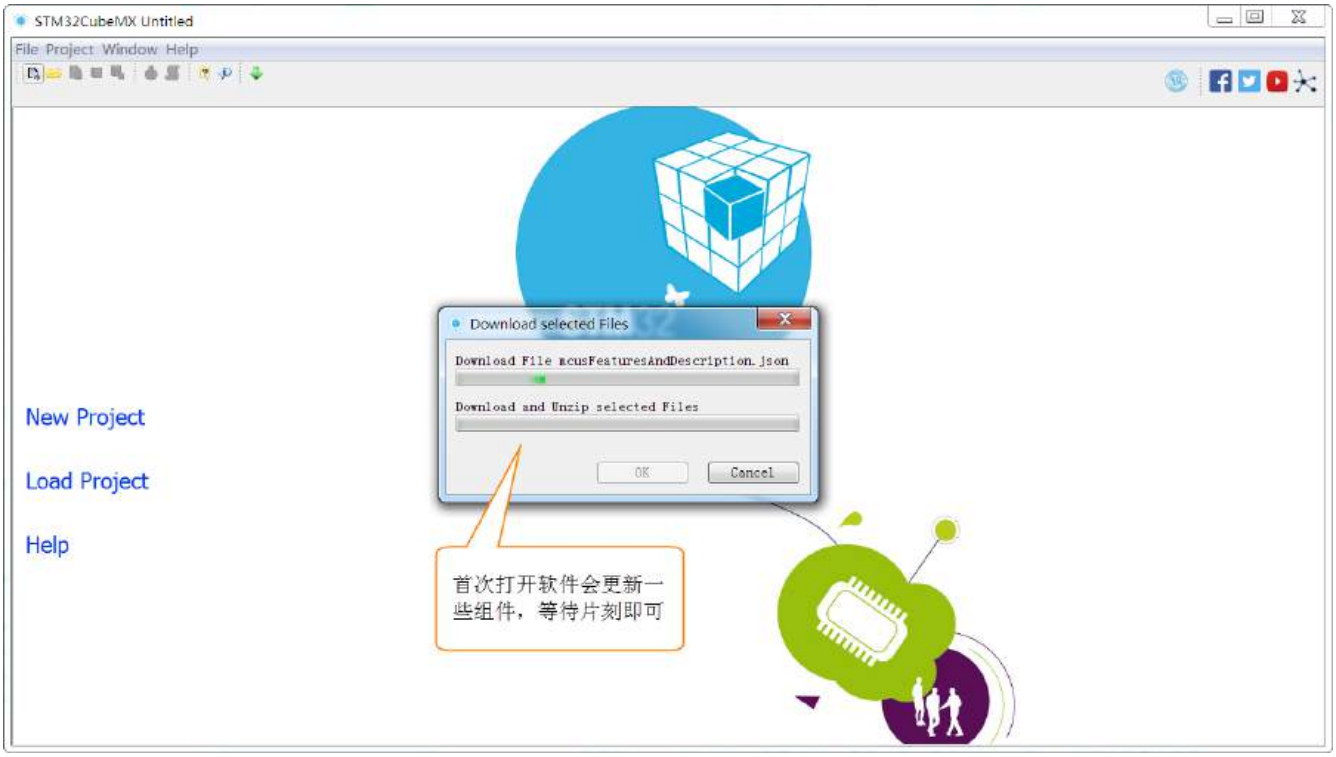

STM32CubeMX Installation Wizard
STM32CubeMX Shortcuts setup
Create shortcuts in the Start-Menu
Create additional shortcuts on the d...
Select a Program Group for the Shortcuts:
(Default)
2345王牌软件
360安全中心
3DTouchPad GUI
7-Zip
create s...
curren...
all users
STMicroelectronics\STM32Cube\STM32CubeMX
Default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



10.2 新建工程

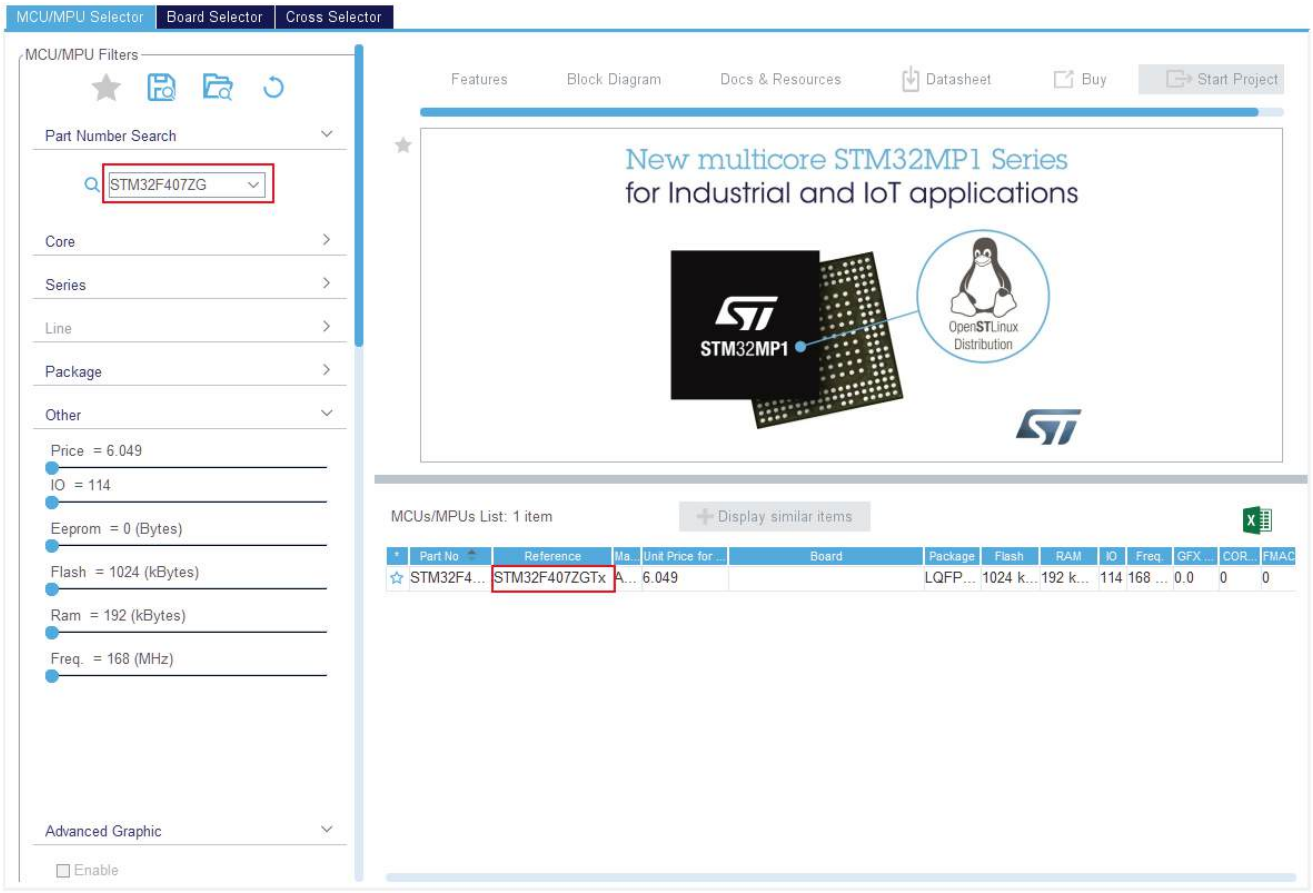
10.2.1 新建工程

打开 STM32CubeMX，软件会更新一些组件，等待安装完成即可。



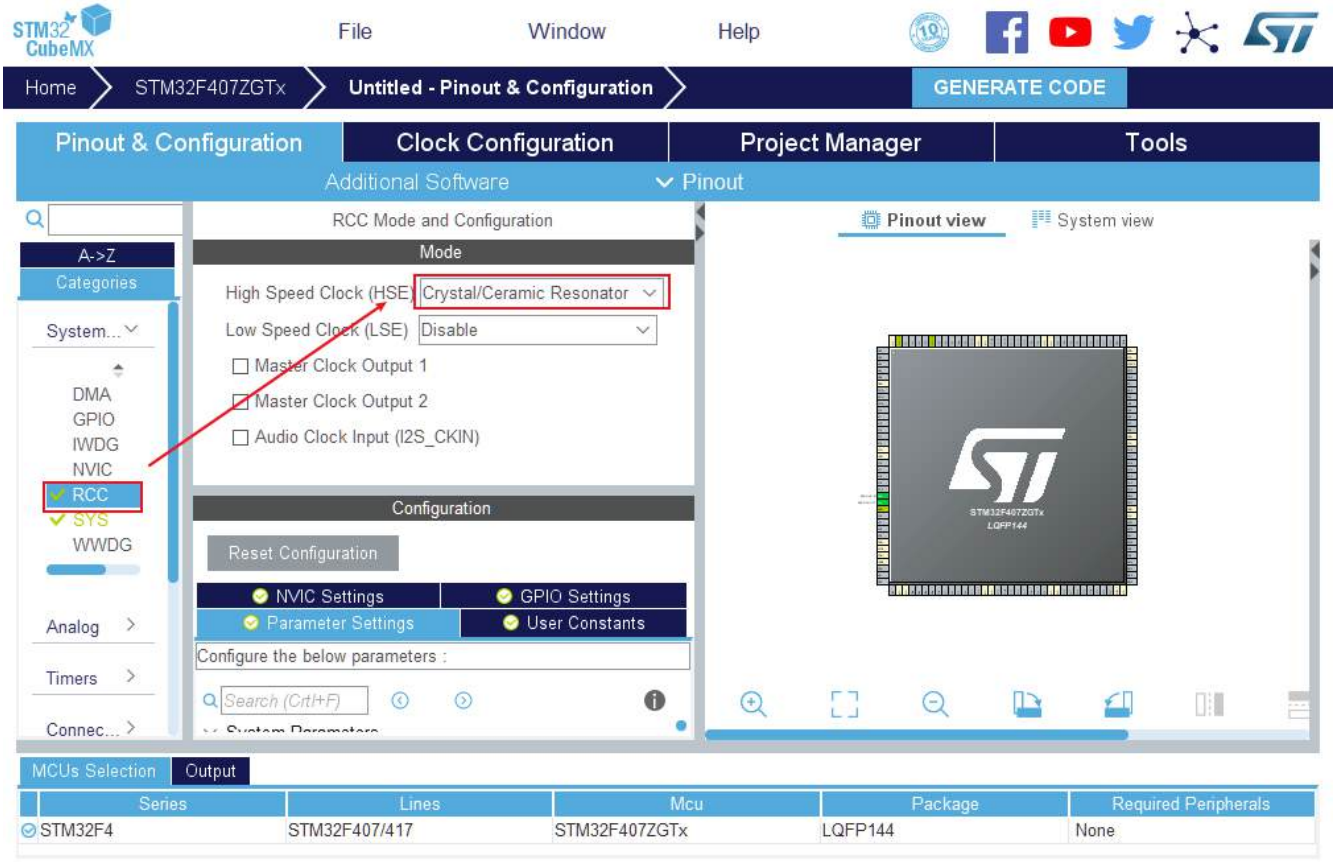
10.2.1.1 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，M4 挑战者选 STM32F407ZGT 型号。我们直接在搜索框输入型号 STM32F407ZG 最终确认 STM32F407ZGTx 为我们实际使用型号。



10.2.1.2 确认时钟源

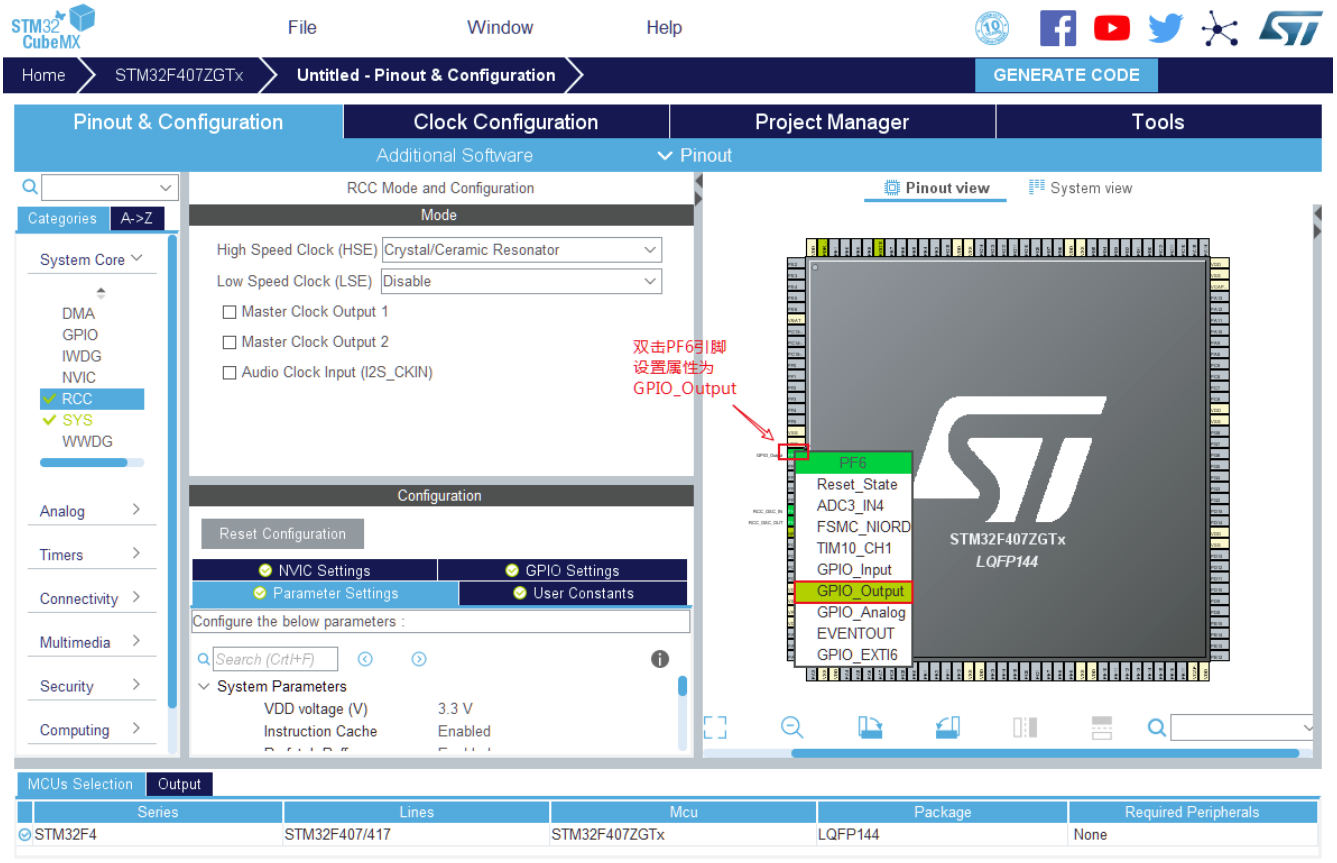
进入工程后打开 RCC 选项，选择 Crystal/Ceramic Resonator，即使用外部晶振作为 HSE 的时钟源。



10.2.1.3 配置 IO 口

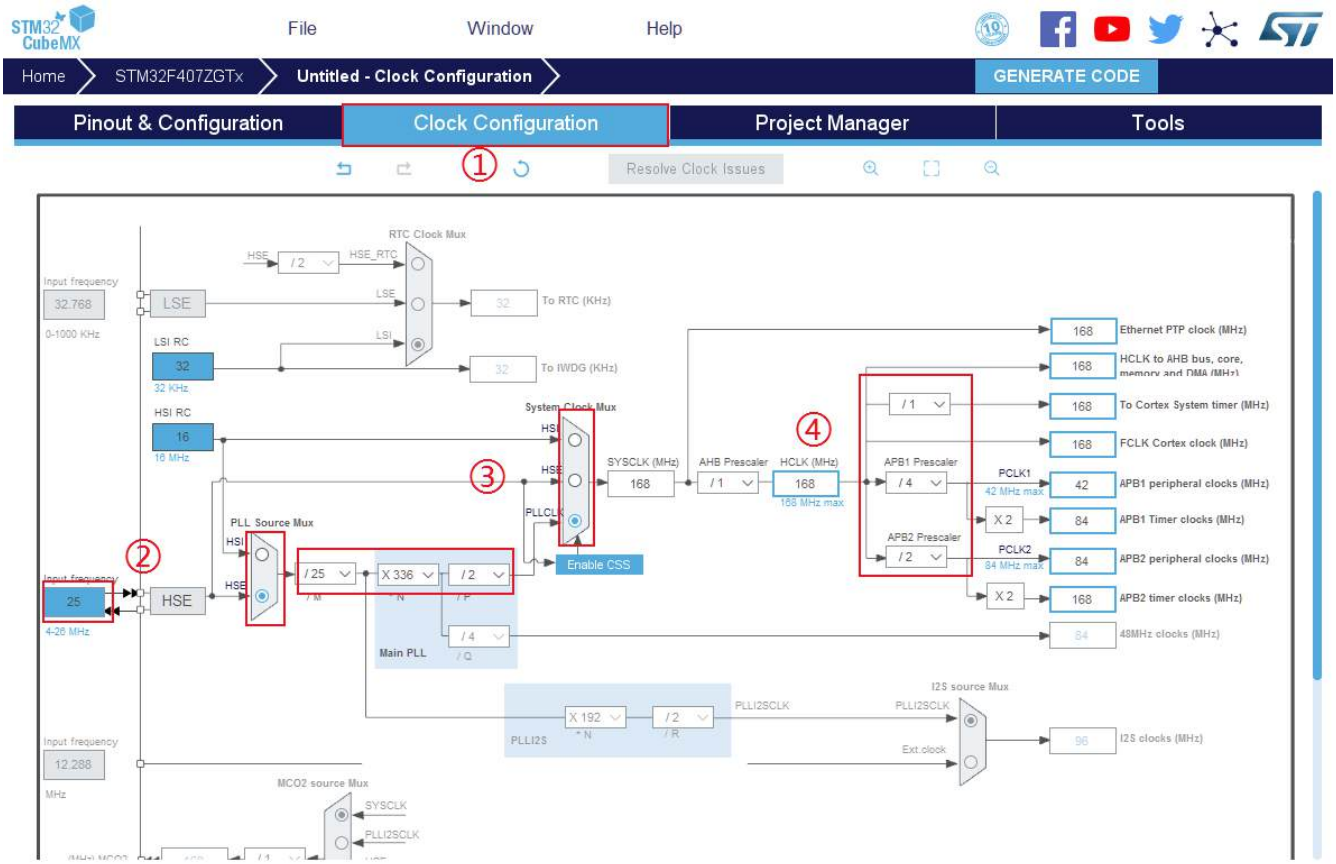
这个工程简单控制一个 LED 周期闪烁，我们只需要配置一个 IO 即可，这里选定控制红色 LED 的引脚 PF6，通过搜索框搜索可以定位 IO 口的引脚位置，图中会闪烁显示，配置 PF6 的属性为 GPIO_Output。


STM32 CubeMX
File
Window
Help
Home
STM32F407ZGTx
Untitled - Pinout & Configuration
GENERATE CODE
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Additional Software
Pinout
Categories
A->Z
System Core
DMA
GPIO
IWDG
NVIC
RCC
SYS
WWDG
Analog
Timers
Connectivity
Multimedia
Security
Computing
RCC Mode and Configuration
Mode
High Speed Clock (HSE)
Crystal/Ceramic Resonator
Low Speed Clock (LSE)
Disable
Master Clock Output 1
Master Clock Output 2
Audio Clock Input (I2S_CKIN)
Configuration
Reset Configuration
NVIC Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Search (Crt/+F)
System Parameters
VDD voltage (V)
3.3 V
Instruction Cache
Enabled
Pinout view
System view
闪烁显示
STM32F407ZGTx
LQFP144
PF6
MCUs Selection
Output
Series
Lines
Mcu
Package
Required Peripherals
STM32F4
STM32F407/417
STM32F407ZGTx
LQFP144
None



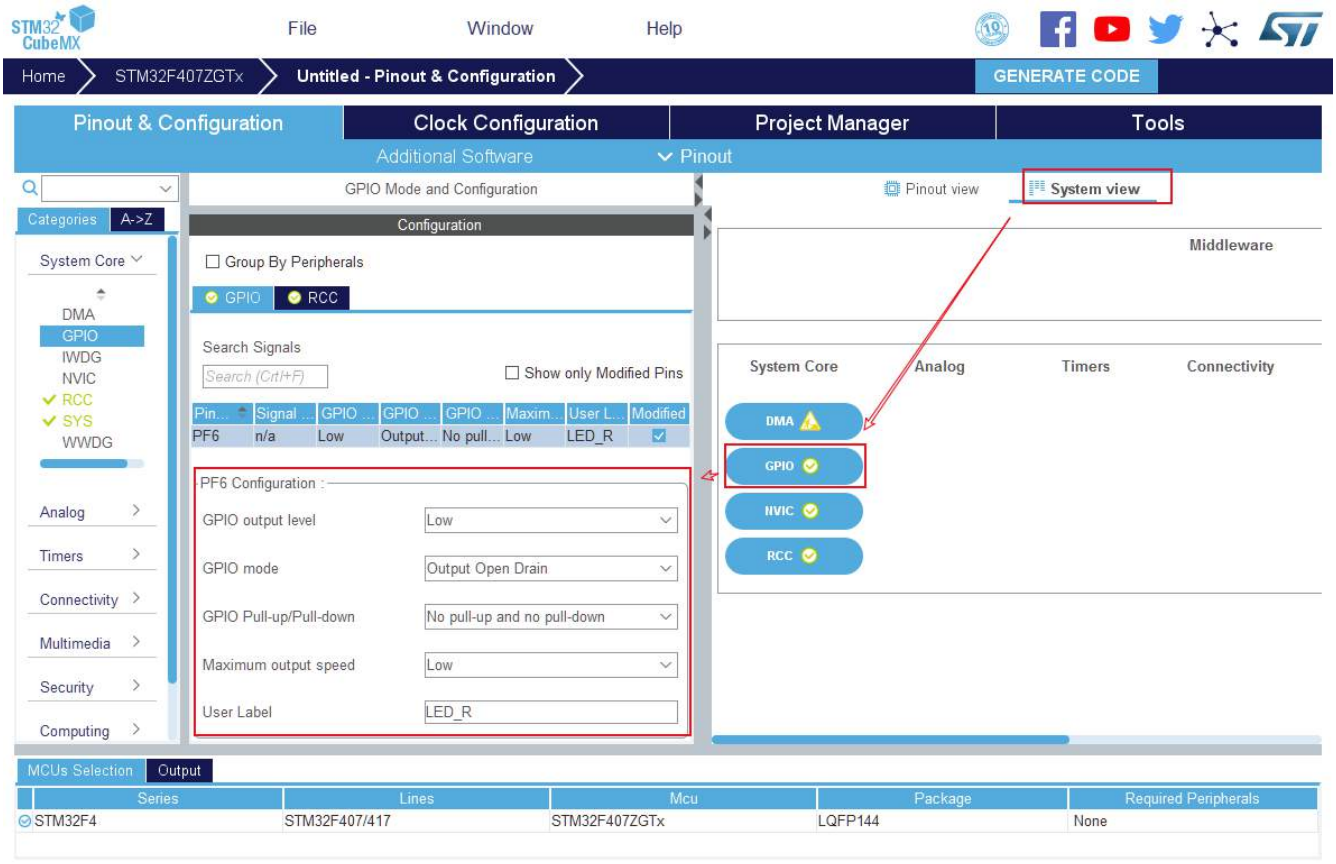
10.2.1.4 配置系统时钟

开发板的外部晶振为 25MHz，我们填入 25；通道选择 LSE；PLLM 选择为/25；倍频系数 N 选择为 x336；系统时钟选择 PLLCLK；系统时钟设定为 168Mz；APB1 分频系数选择为/4 即 PCLK1 位 42MHz；APB2 分频系数选择为/2 即 PCLK2 位 84MHz。



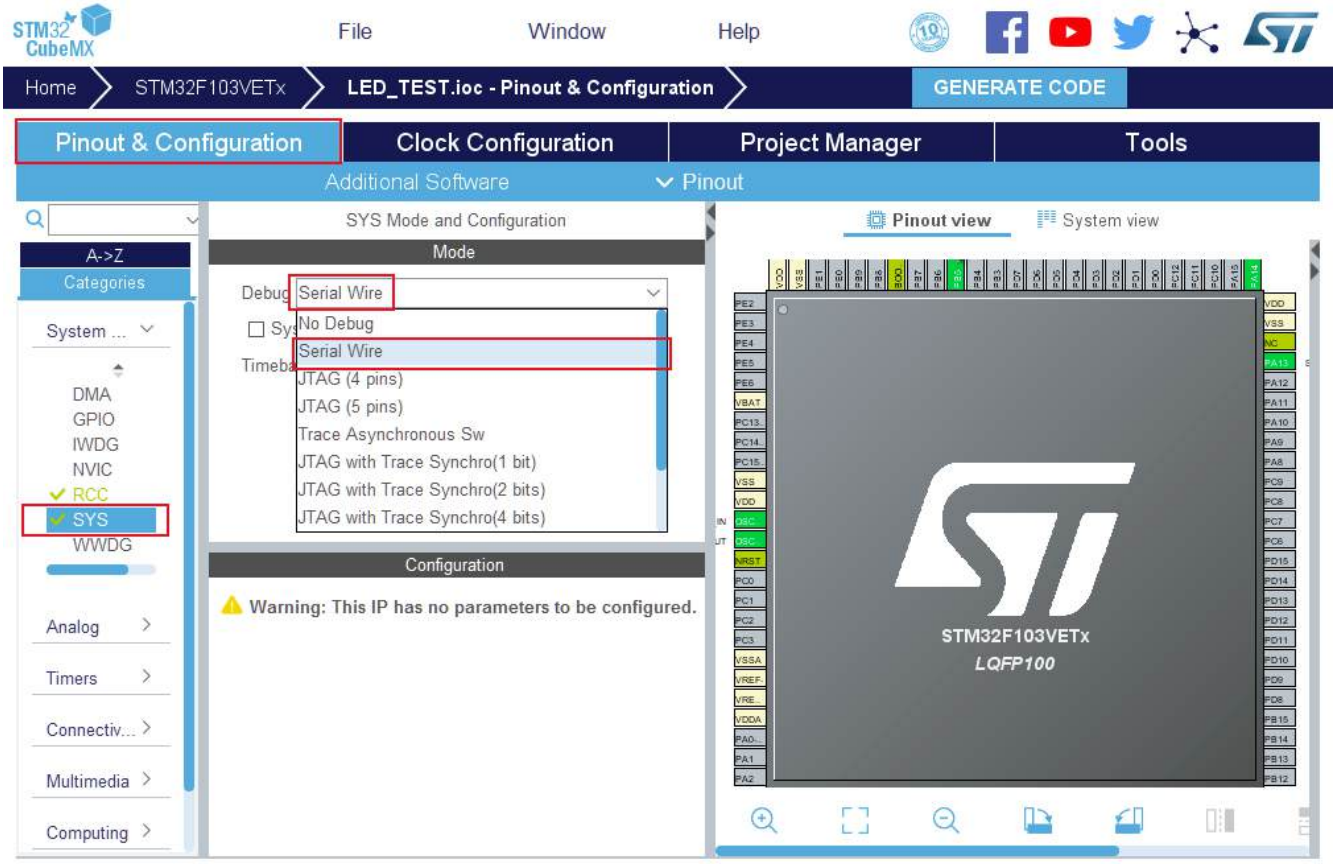
10.2.1.5 进一步配置 IO 的具体属性

点击 Configuration，进入系统详细配置，选着 GPIO，配置 PH10 的默认电平，开漏输出，无上下拉，低速模式。引脚标签为 LED_R。



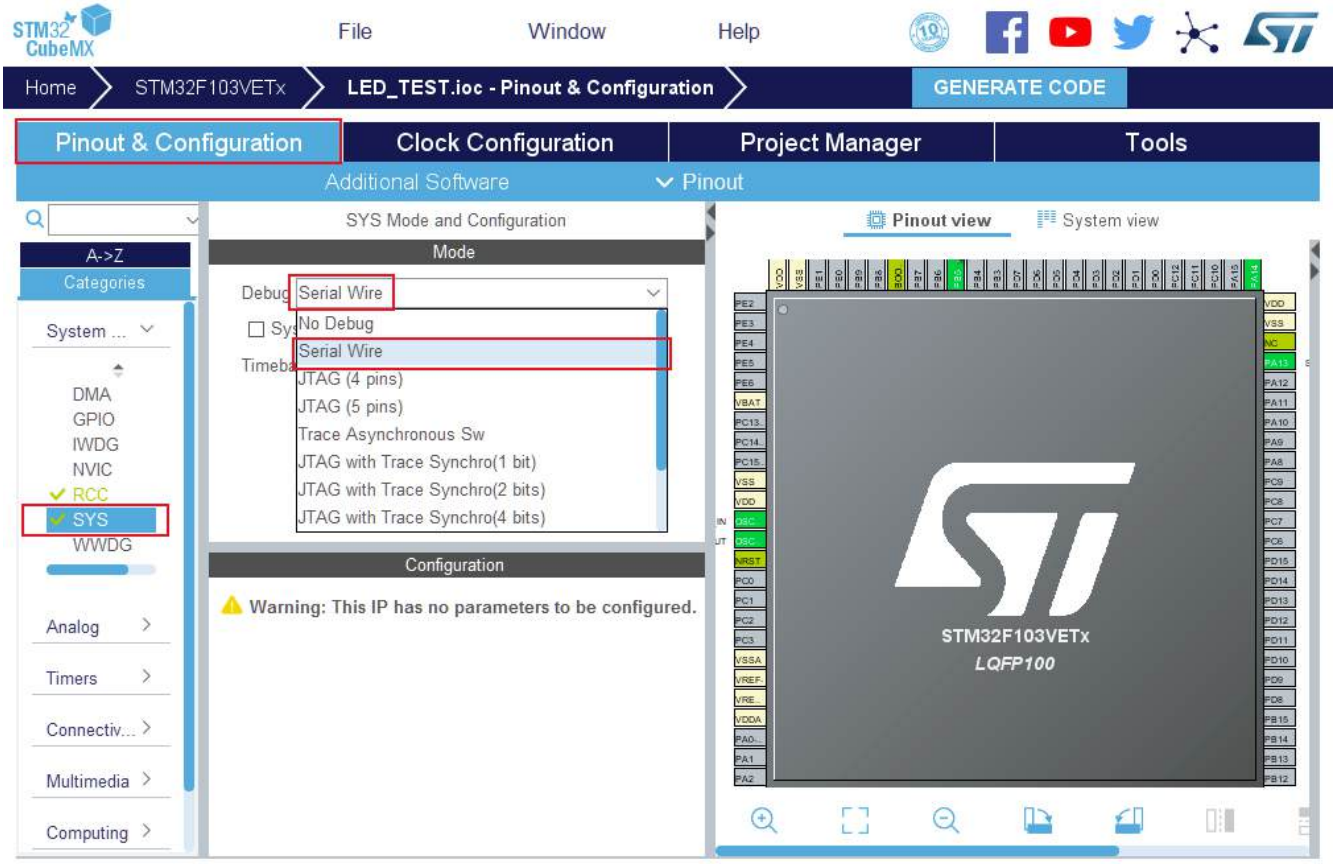
10.2.1.6 配置工程属性

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。



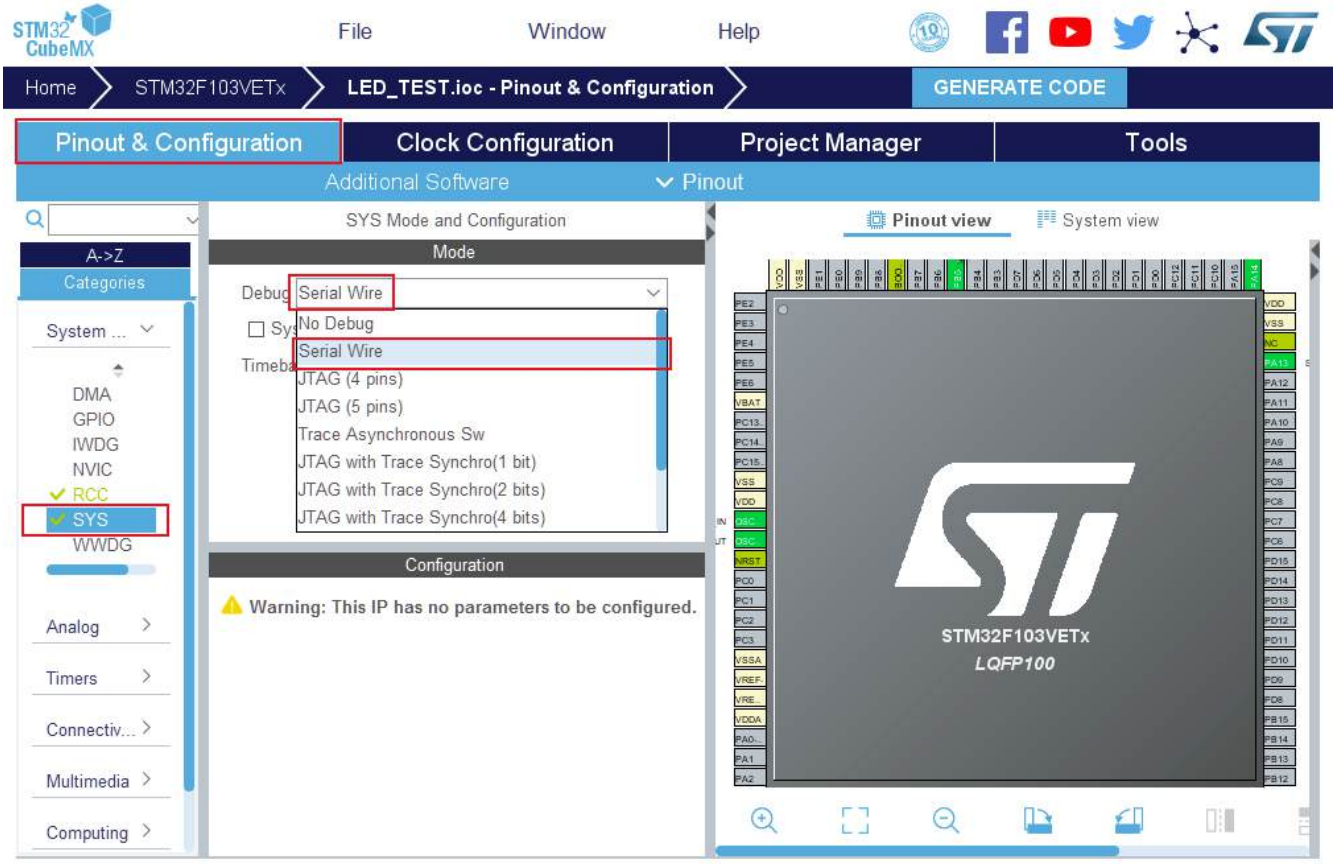
10.2.1.7 生成代码

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



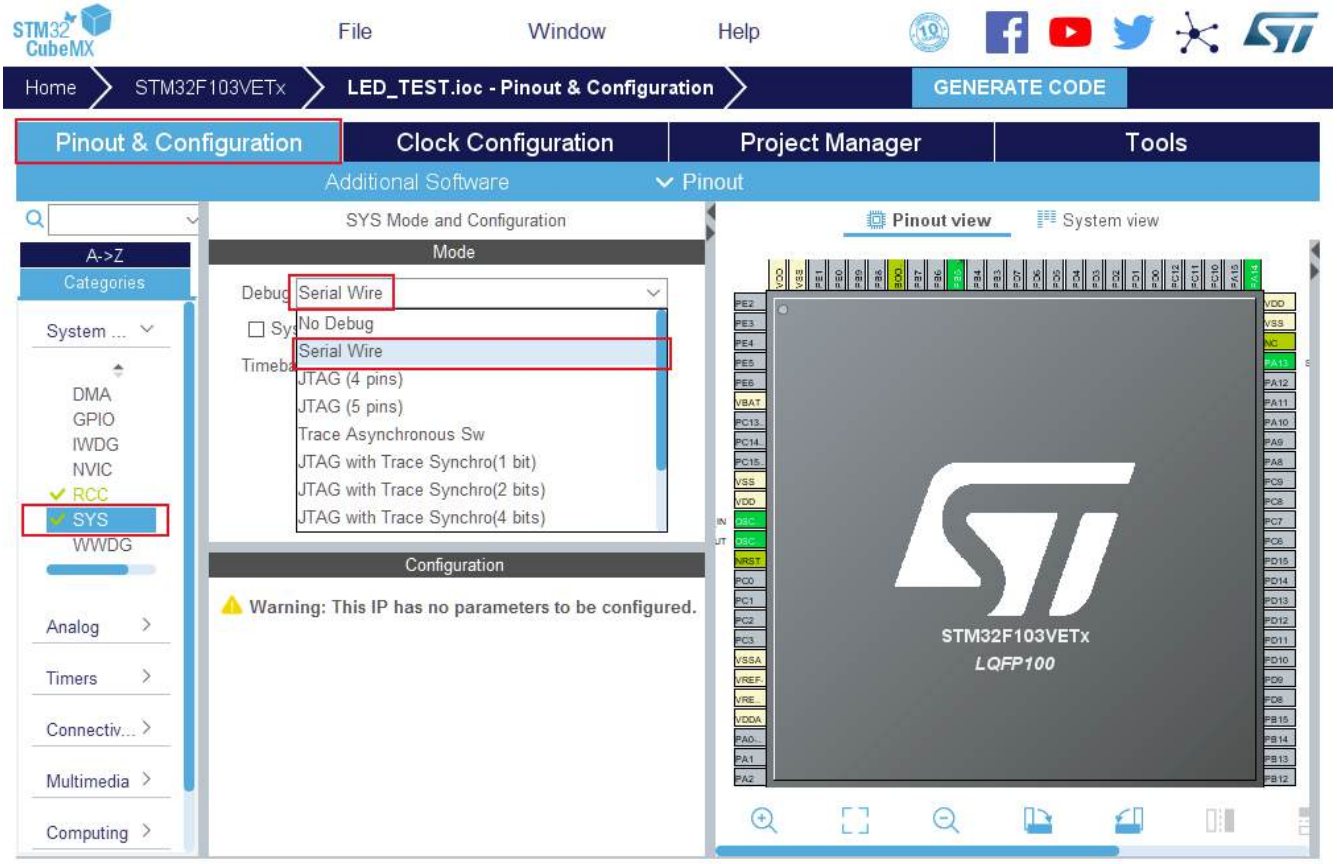
10.2.1.8 添加用户测试代码

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。



10.2.1.9 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。



10.2.2 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 11 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。

11.1 新建工程

版本说明：MDK5.15 (MDK 即 KEIL 软件)

版本号可从 MDK 软件的“Help->About uVision”选项中查询到。

11.1.1 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 CM4 内核有关的库文件
	STM32F4xx_HAL_Driver：STM32 外设库文件
Listing	暂时为空
Output	暂时为空
Project	暂时为空
User	stm32f4xx_hal_conf.h：用来配置库的头文件
	stm32f4xx_it.h stm32f4xx_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

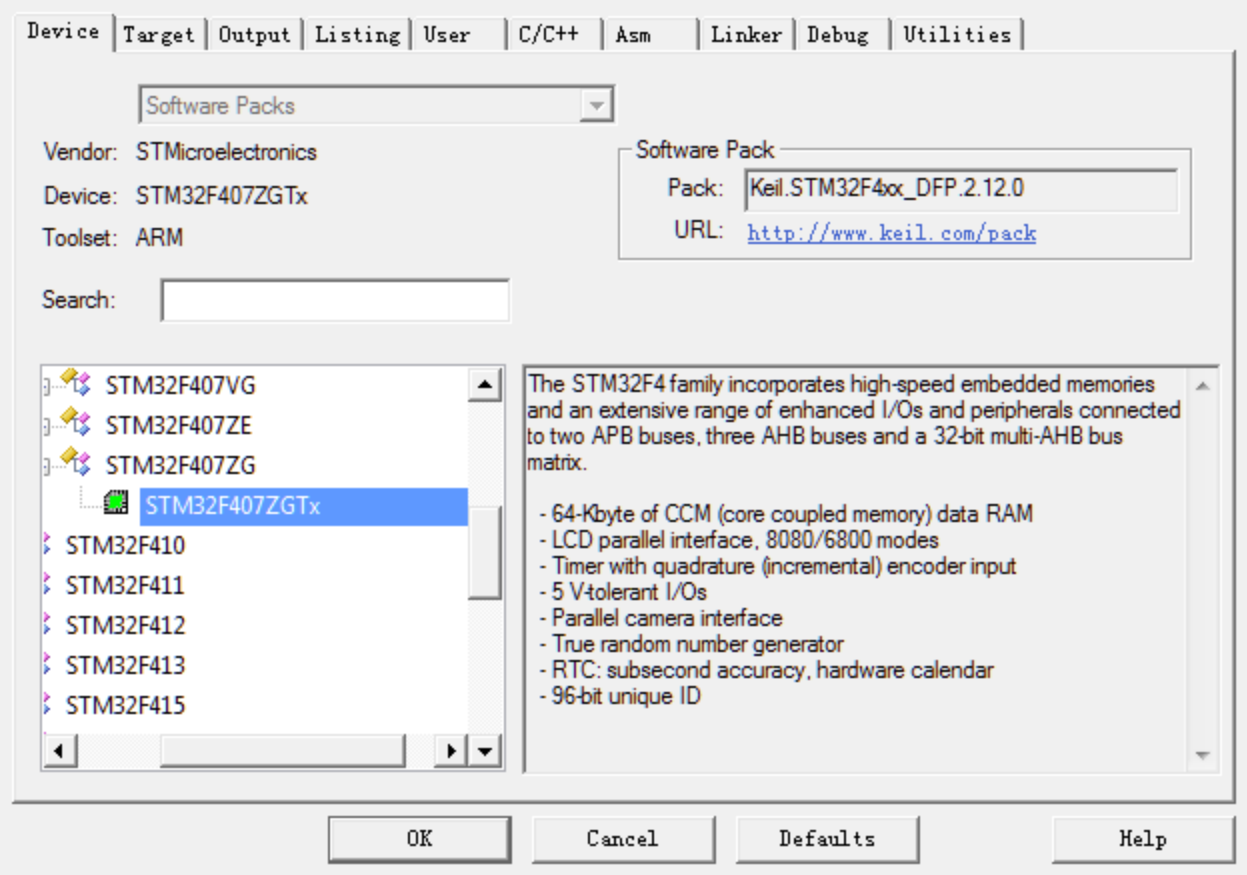
11.1.2 新建工程

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。



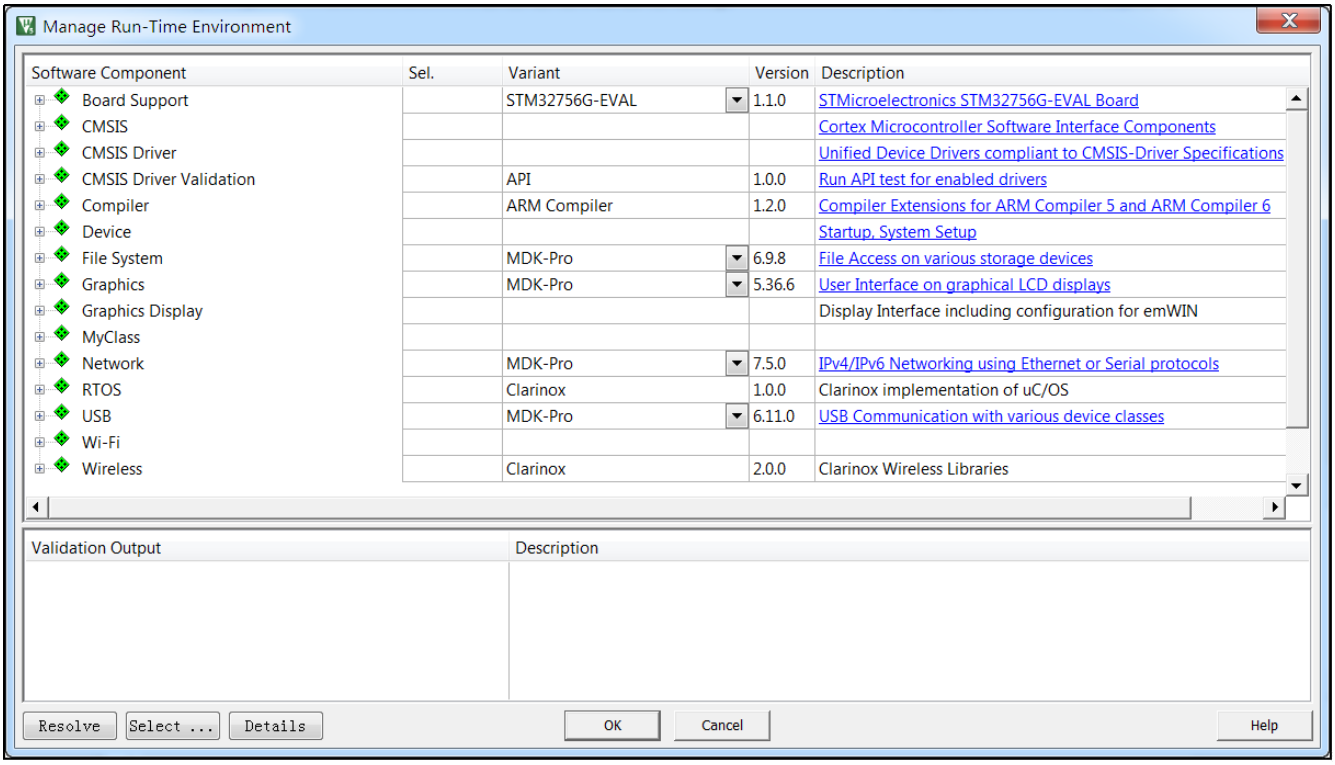
11.1.2.1 选择 CPU 型号

这个根据你开发板使用的 CPU 具体的型号来选择，M4 挑战者选 STM32F407ZGT 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。



11.1.2.2 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。

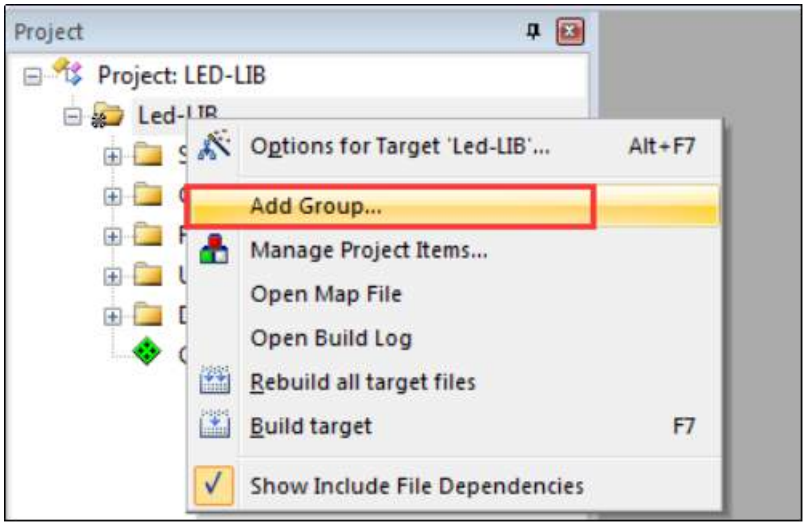


11.1.2.3 添加组文件夹

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。

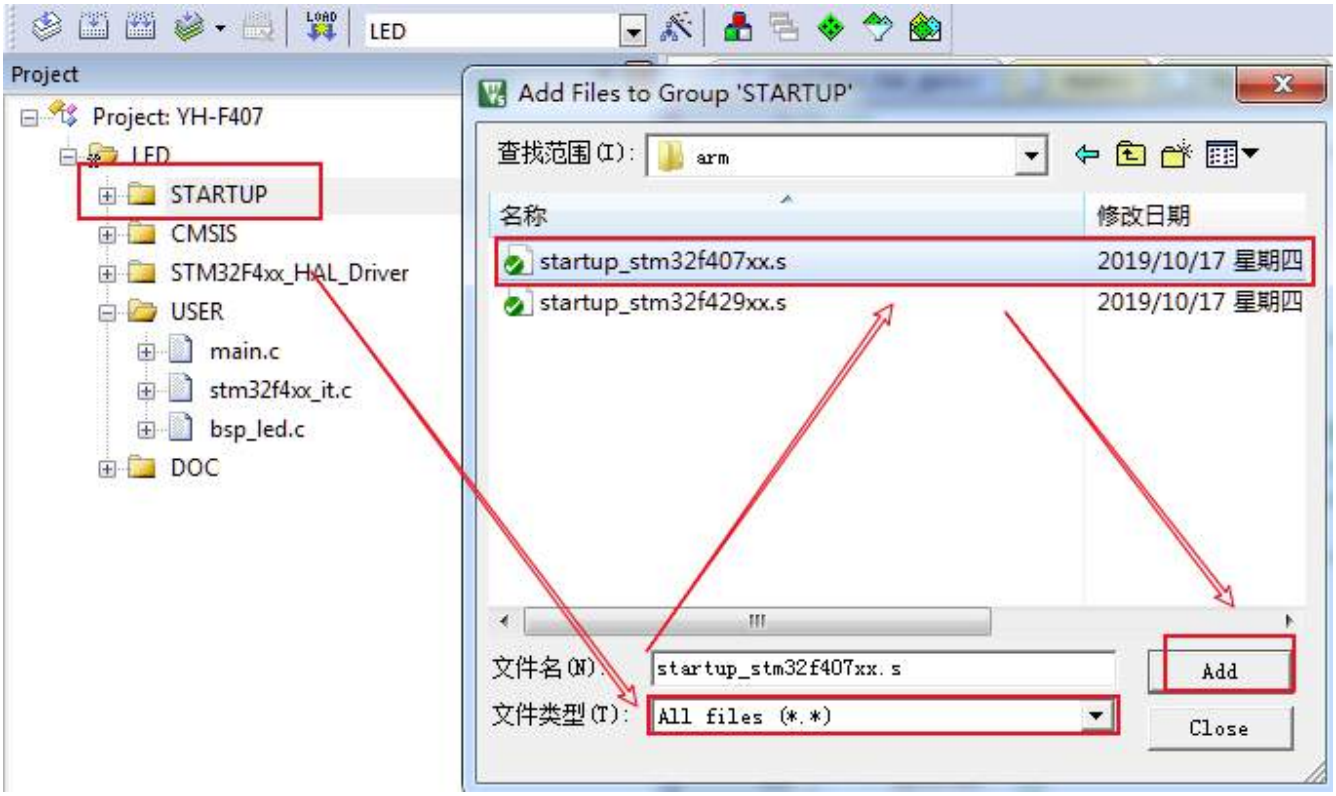
表 11-3 工程内组文件夹内容清单

名称	作用
STARTUP	存放汇编的启动文件：startup_STM32F407xx.s
STM32F4xx_HAL_Driver	与 STM32 外设相关的库文件
	stm32f4xx_hal.c
	stm32f4xx_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
	stm32f4xx_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



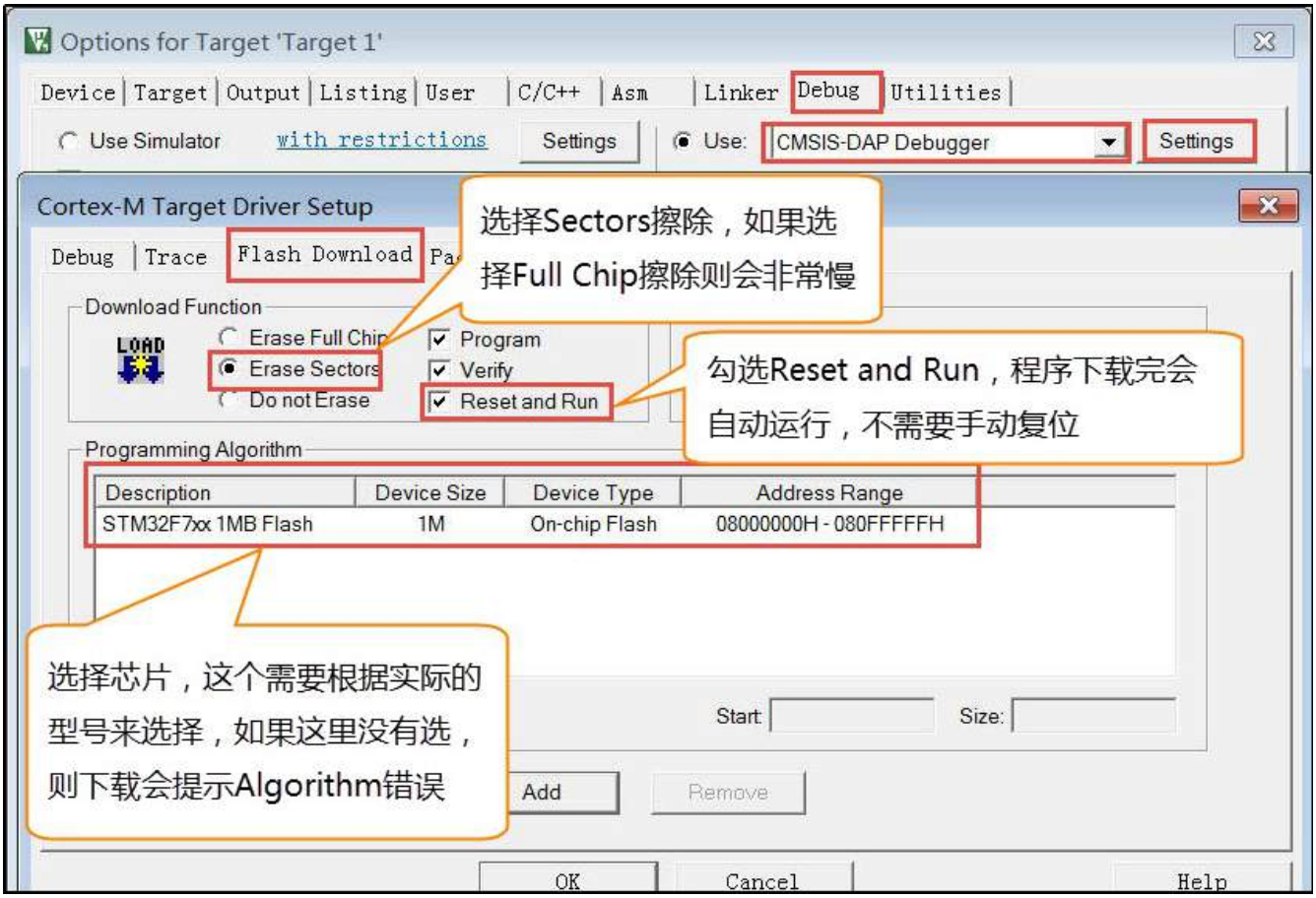
11.1.2.4 添加文件

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。



11.1.2.5 设置文件是否加入编译

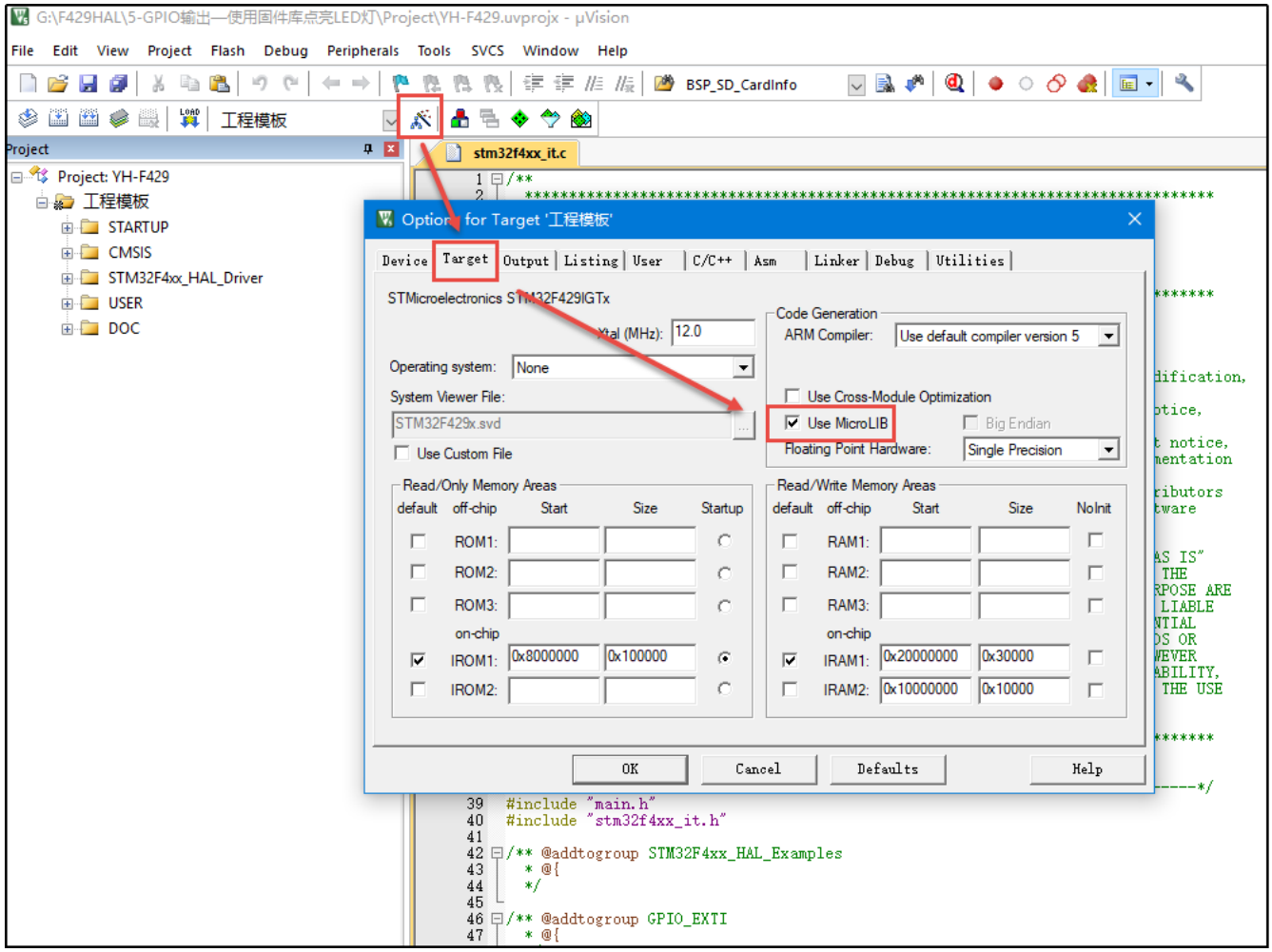
STM32F407 外设比较丰富，它的库文件比较庞大，在添加外设文件时，为了减少编译时间。我们把外设库的所有文件都添加进工程，使用下面的方法把暂时没有用到的库文件，设置为不加入编译，这样就不会对特定文件进行编译。这种设置在开发时也很常用，暂时不把文件加进编译，方便调试，加快开发进度。



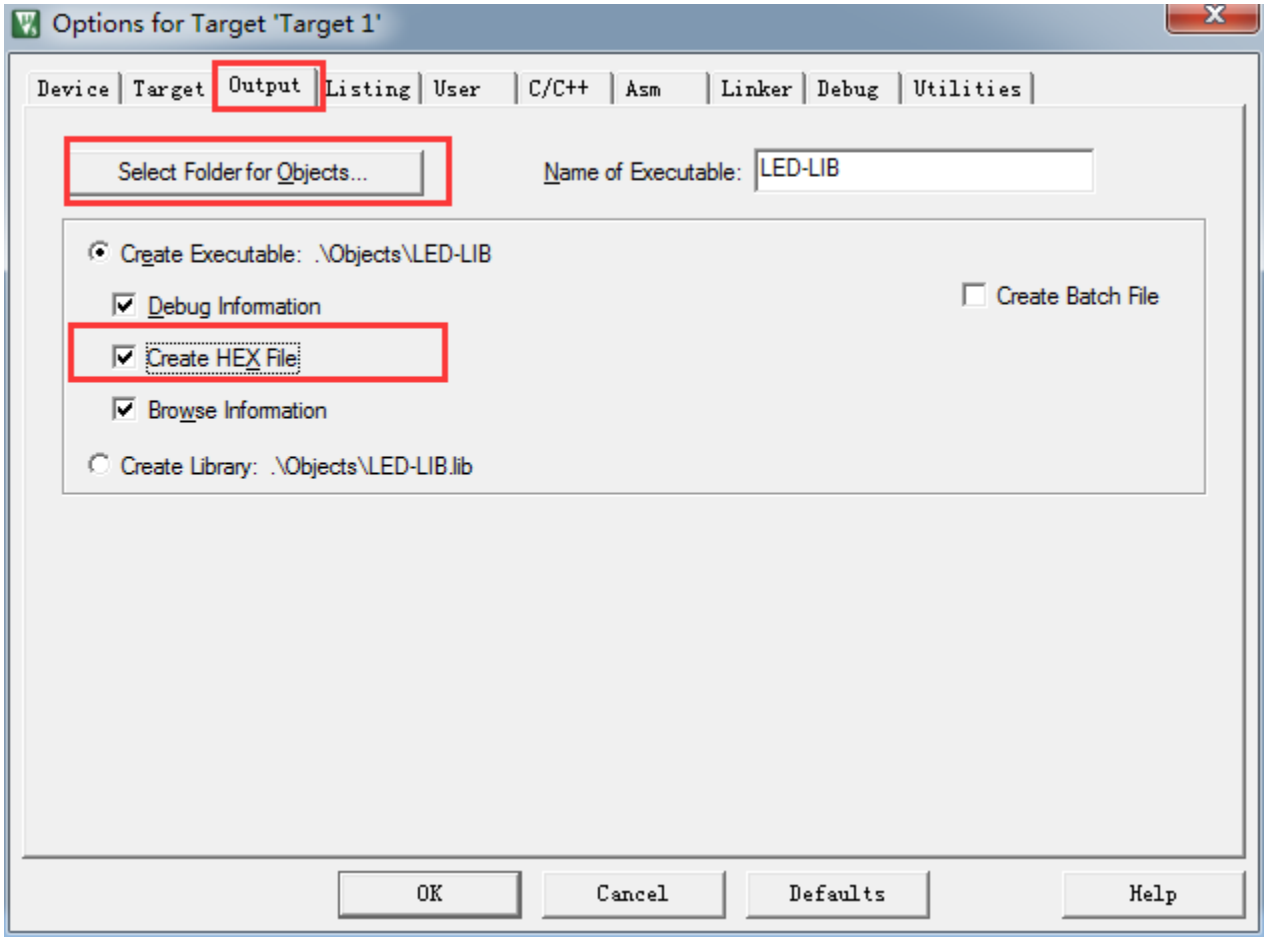
11.1.2.6 配置魔术棒选项卡

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。

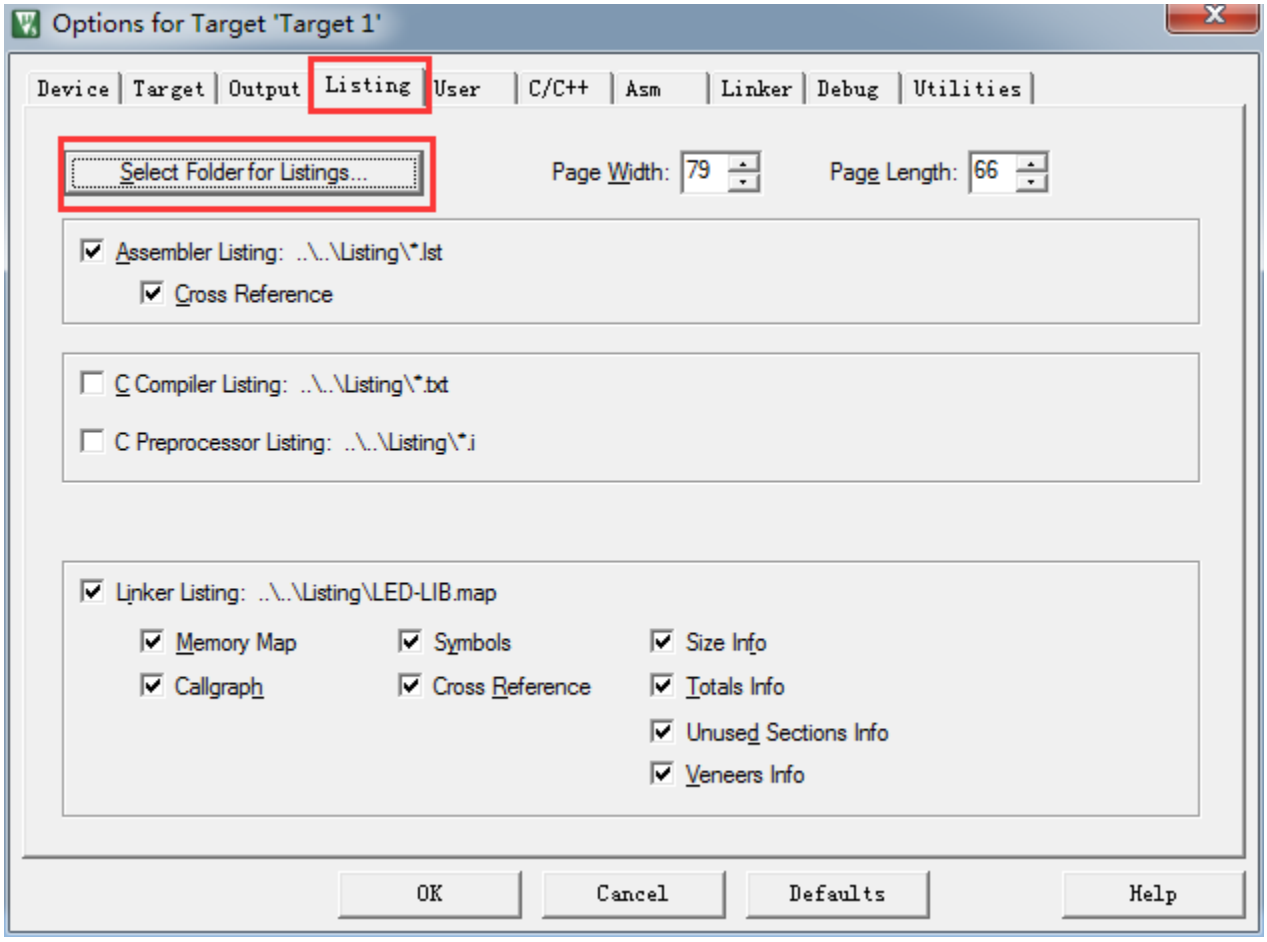
(1) Target 中选中微库“Use MicroLib”，为的是在日后编写串口驱动的时候可以使用 printf 函数。而且有些应用中如果用了 STM32 的浮点运算单元 FPU，一定要同时开微库，不然有时会出现各种奇怪的现象。FPU 的开关选项在微库配置选项下方的“Use double Precision”中，默认是开的。



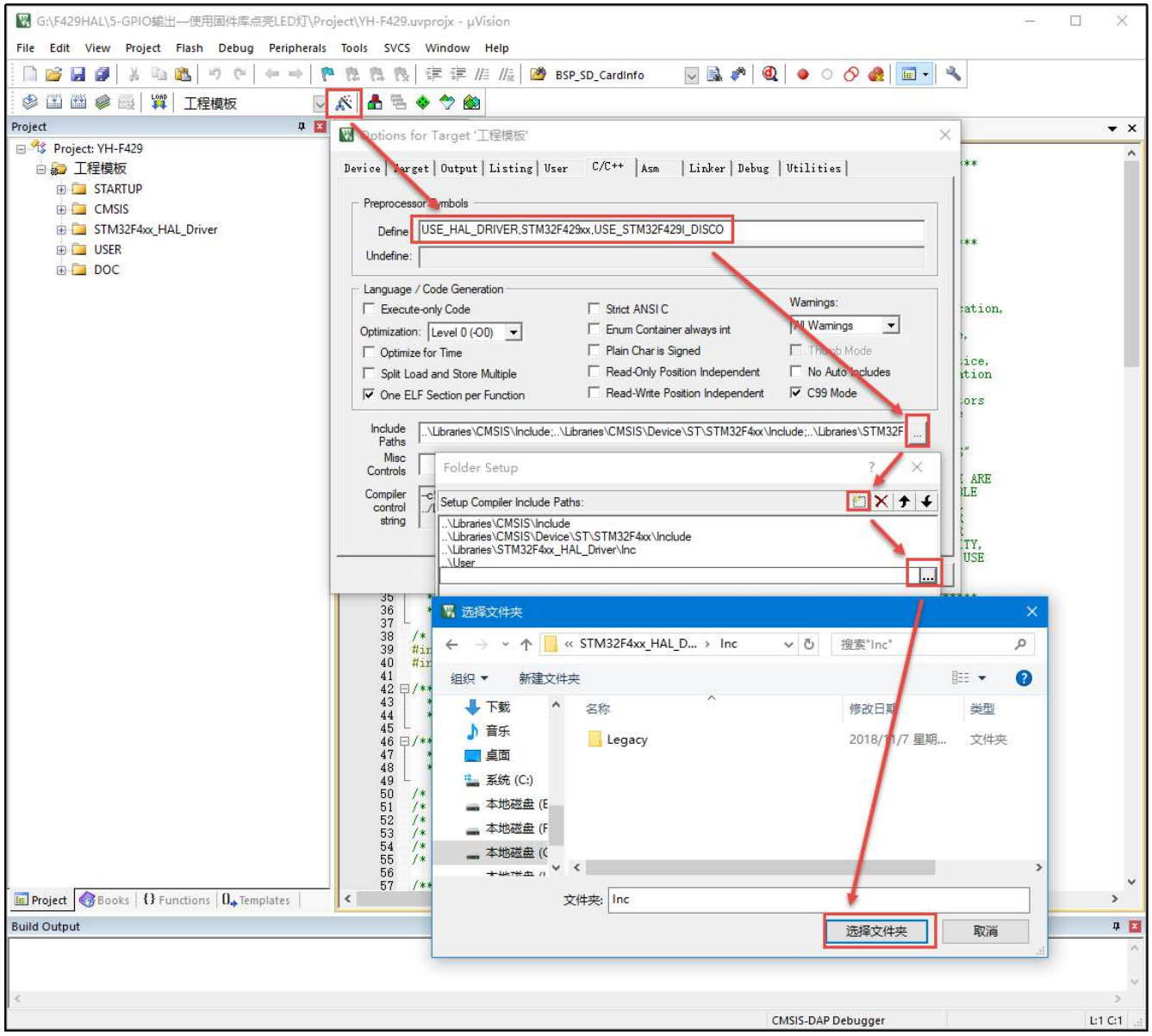
(2) 在 Output 选项卡中把输出文件夹定位到我们工程目录下的“output”文件夹，如果想在编译的过程中生成 hex 文件，那么那 Create HEX File 选项勾上。



(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。

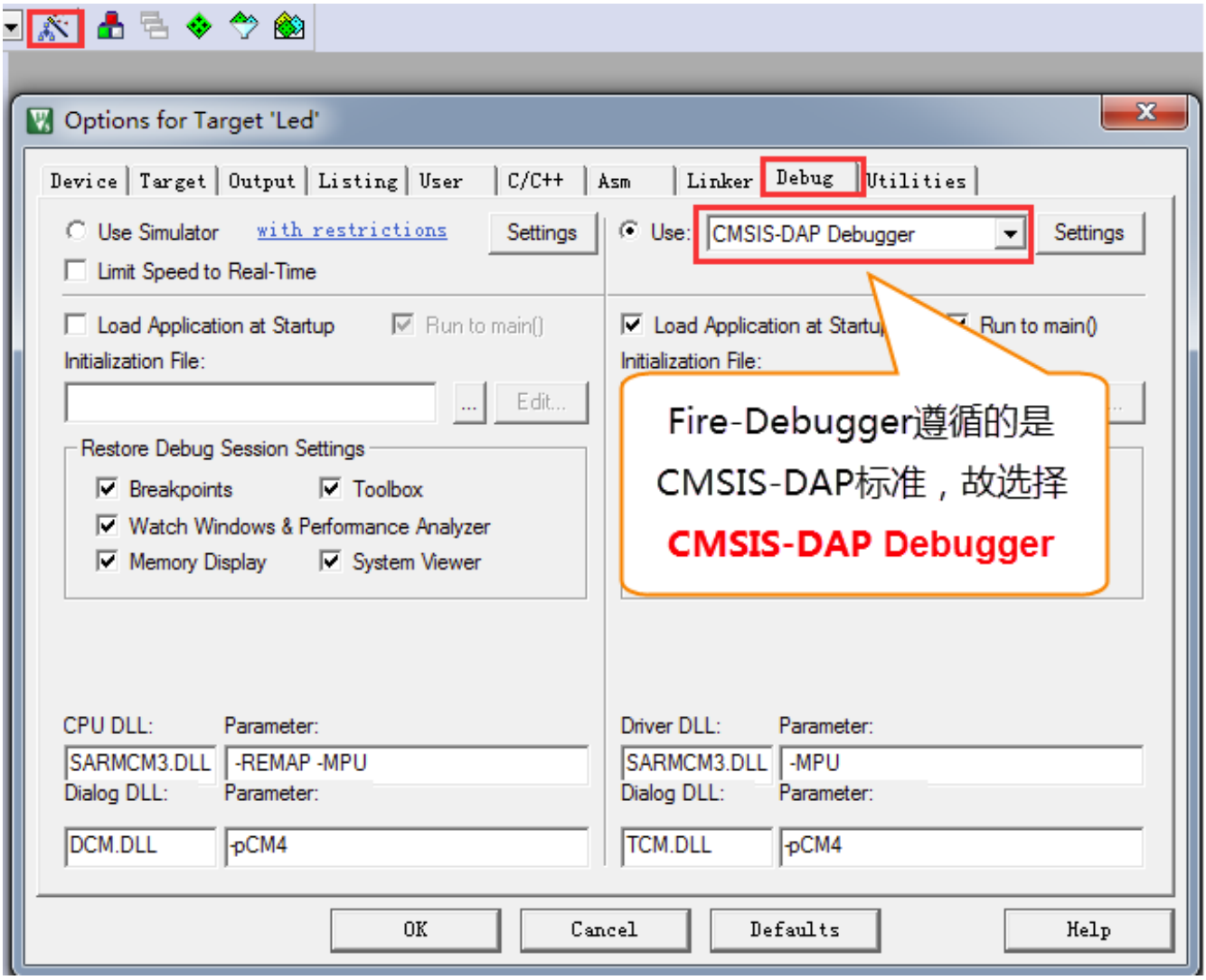
- STM32F407xx 宏：为了告诉 STM32 HAL 库，我们使用的芯片是 STM32F407 型号，使 STM32 HAL 库根据我们选定的芯片型号来配置。
- USE_HAL_DRIVER 宏：为了让 stm32F407xx.h 包含 stm32f4xx_hal_conf.h 这个头文件。

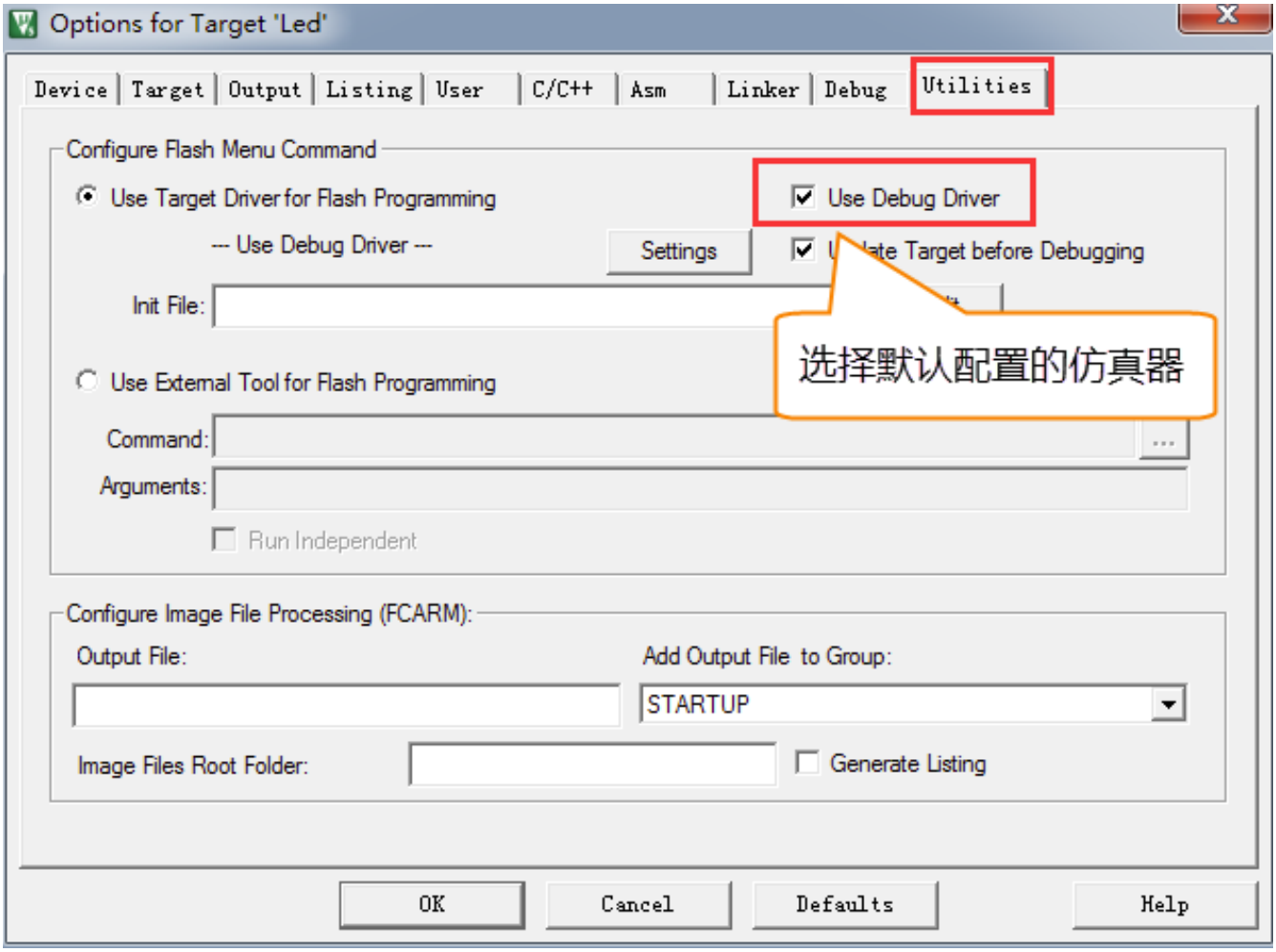
“Include Paths ”这里添加的是头文件的路径，如果编译的时候提示说找不到头文件，一般就是这

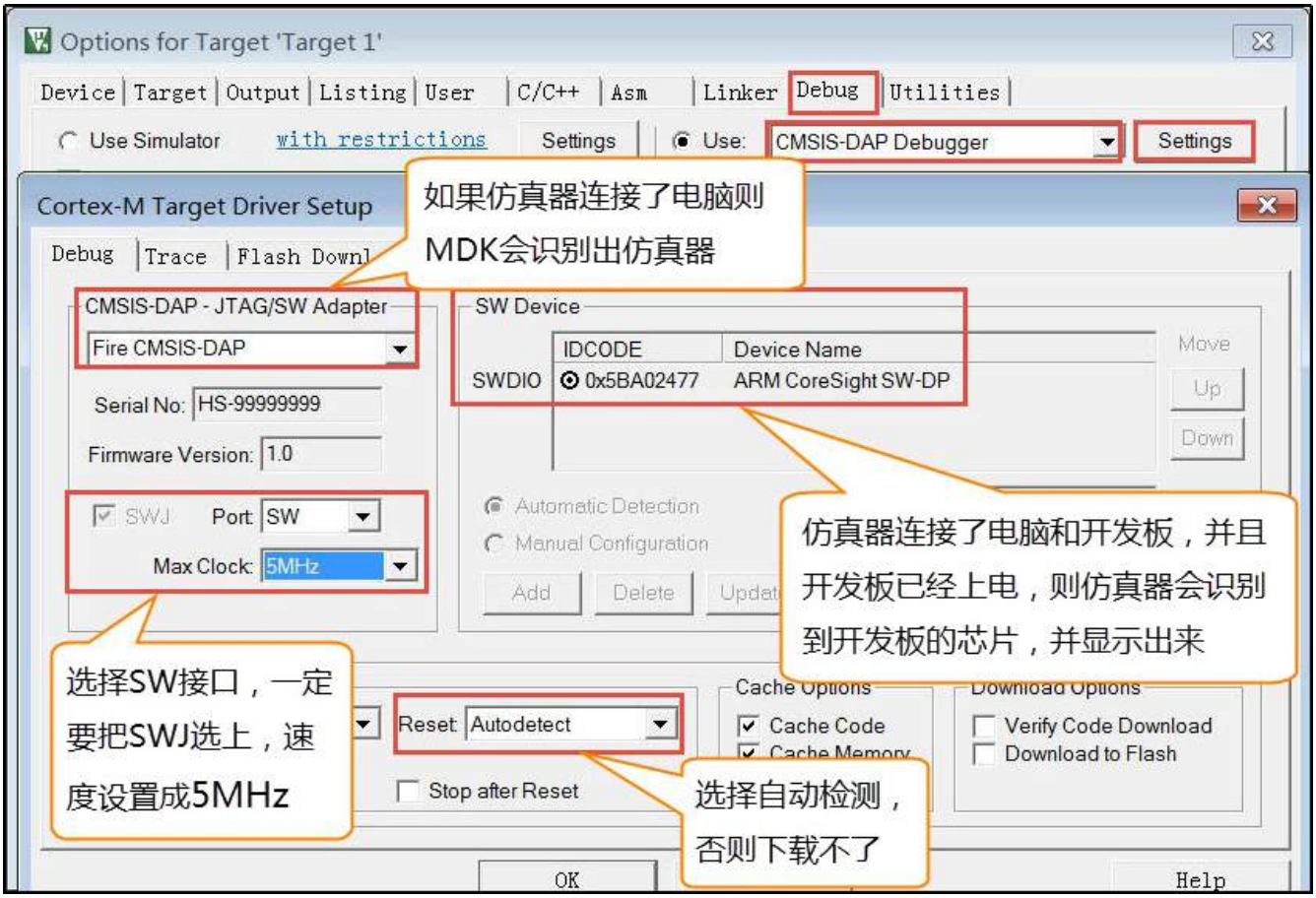
里配置出了问题。你把头文件放到了哪个文件夹，就把该文件夹添加到这里即可。(请使用图中的方法用文件浏览器去添加路径，不要直接手打路径，容易出错)

11.1.2.7 下载器配置

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。

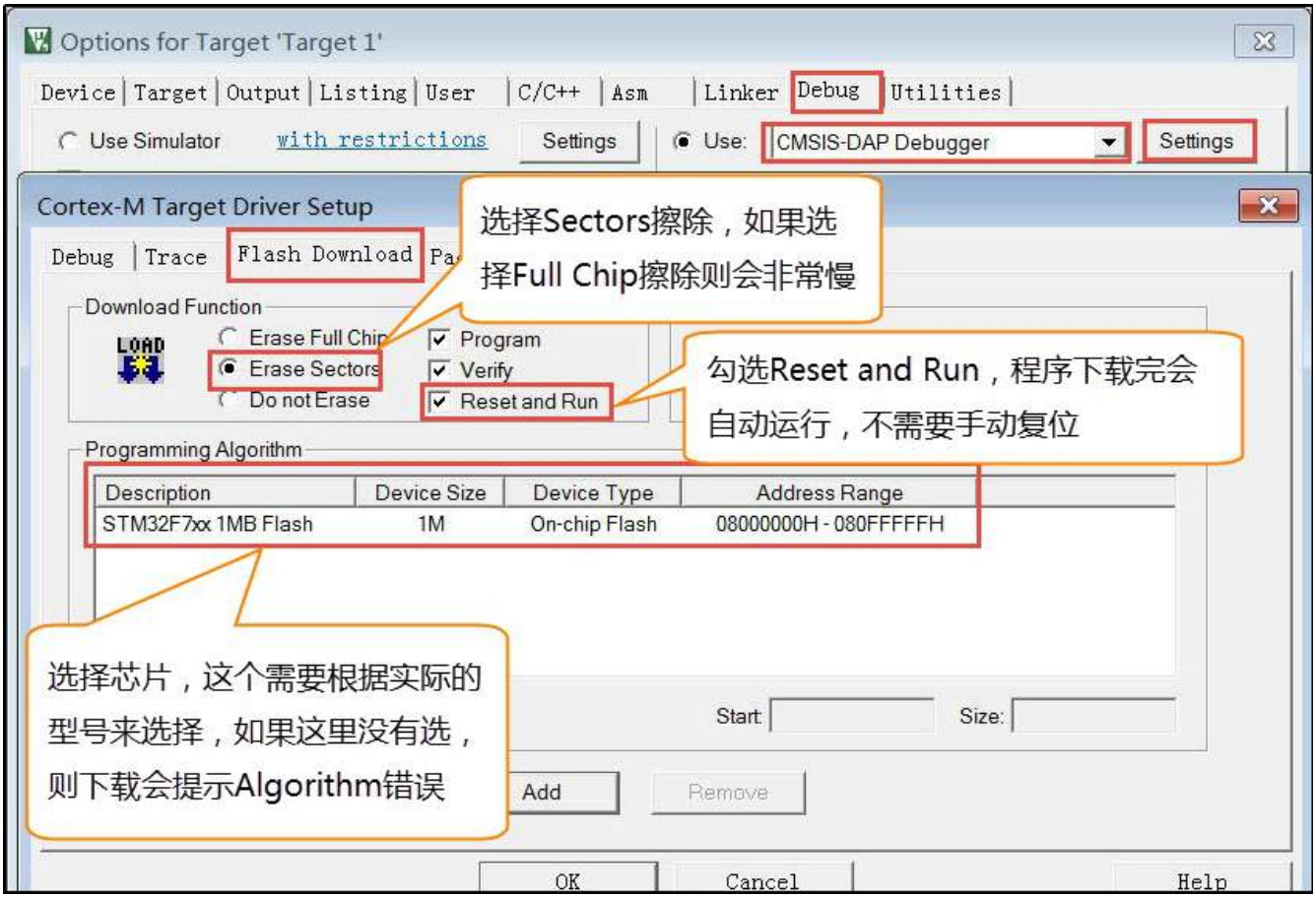



Options for Target 'Led'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Configure Flash Menu Command
Use Target Driver for Flash Programming
Use Debug Driver
--- Use Debug Driver ---
Settings
Init File:
选择默认配置的仿真器
Use External Tool for Flash Programming
Command:
Arguments:
Run Independent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



11.1.2.8 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。M4 用的 STM32 的 FLASH 大小是 1M，所以这里选择 1M 的容量，如果使用的是其他型号的，要根据实际情况选择。



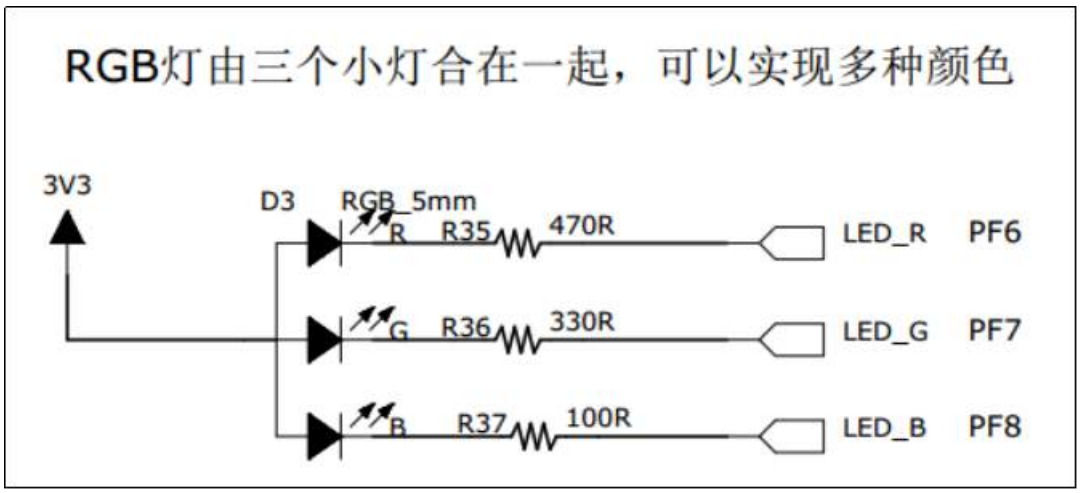
一个新的工程模版新建完毕。

第 12 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 STM32F4 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。

12.1 硬件设计

在本教程中 STM32 芯片的 PF6、PF7 和 PF8 引脚分别与一个 RGB 灯的 R 灯（红灯）、G 灯（蓝灯）和 B 灯（绿灯）连接，具体见图 *12_1*，RGB 灯里面的三个小灯都可以单独控制，如果有两个或者三个灯同时亮的话就会混合成其它的颜色。



这些 LED 灯的阴极都是连接到 STM32 的 GPIO 引脚，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。若你使用的实验板 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。

12.2 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。

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。

12.2.2 代码分析

12.2.2.1 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*。

列表 1: 代码清单 12-1 LED 控制引脚相关的宏

```
//引脚定义
/*******************************************************/
//R 红色灯
```

(下页继续)

(续上页)

```
#define LED1_PIN                  GPIO_PIN_6
#define LED1_GPIO_PORT            GPIOF
#define LED1_GPIO_CLK_ENABLE()    __GPIOF_CLK_ENABLE()

//G 绿色灯
#define LED2_PIN                  GPIO_PIN_7
#define LED2_GPIO_PORT            GPIOF
#define LED2_GPIO_CLK_ENABLE()    __GPIOF_CLK_ENABLE()

//B 蓝色灯
#define LED3_PIN                  GPIO_PIN_8
#define LED3_GPIO_PORT            GPIOF
#define LED3_GPIO_CLK_ENABLE()    __GPIOF_CLK_ENABLE()

/************************************************************/
```

以上代码分别把控制三盏 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。

12.2.2.2 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_2*。

列表 2: 代码清单 12-2 控制 LED 亮灭的宏

```
/* 控制 LED 灯亮灭的宏，
```

(下页继续)

(续上页)

```
* 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)
```

(下页继续)

(续上页)

```
#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)

/* 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 \
```

(下页继续)

(续上页)

```
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;\
```

(下页继续)

(续上页)

```
LED2_OFF\

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\
```

(下页继续)

(续上页)

```
LED3_OFF
```

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。

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。

12.2.2.3 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_3*。

列表 3: 代码清单 12-3 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();
```

(下页继续)

(续上页)

```
    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*/

    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 */
```

(下页继续)

(续上页)

```
    GPIO_InitStruct.Pin = LED3_PIN;

    HAL_GPIO_Init(LED3_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。

12.2.2.4 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_4*。

列表 4: 代码清单 12-4 控制 LED 灯，main 文件

```
int main(void)

{

    /* 系统时钟初始化成 216 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);
```

(下页继续)

(续上页)

```
    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;

    HAL_Delay(1000);

    LED_CYAN;
```

(下页继续)

(续上页)

```
        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 168MHz，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。

12.2.3 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。

12.3 STM32 HAL 库补充知识

12.3.1 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_stm32f4xx.c”文件中定义了，而我们的工程已经包含该文件。

12.3.2 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_5*。

列表 5: 代码清单 12-5 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));
```

(下页继续)

(续上页)

```
    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_6*。

列表 6: 代码清单 12-6 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)
```

(下页继续)

(续上页)

```
    ((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_7。

列表 7: 代码清单 12-7 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);
```

(下页继续)

(续上页)

```
}
```

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。

12.3.3 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_8*。

列表 8: 代码清单 12-8 Doxygen 注释规范

```
/*

* @brief 初始化控制 LED 的 IO

* @param 无

```

(下页继续)

(续上页)

```
* @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。

12.3.4 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_9 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 9: 代码清单 12-9 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 “stm32F407xx.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 “stm32F407xx.h””，这个时候 “stm32F407xx.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 stm32F407xx.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。

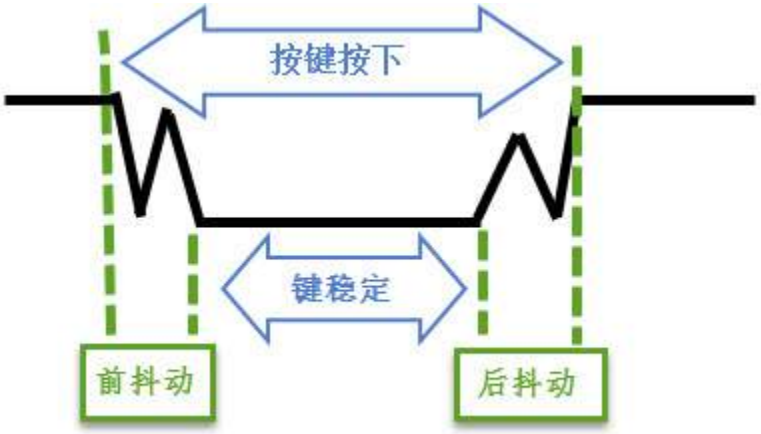
第 13 章　GPIO 输入—按键检测

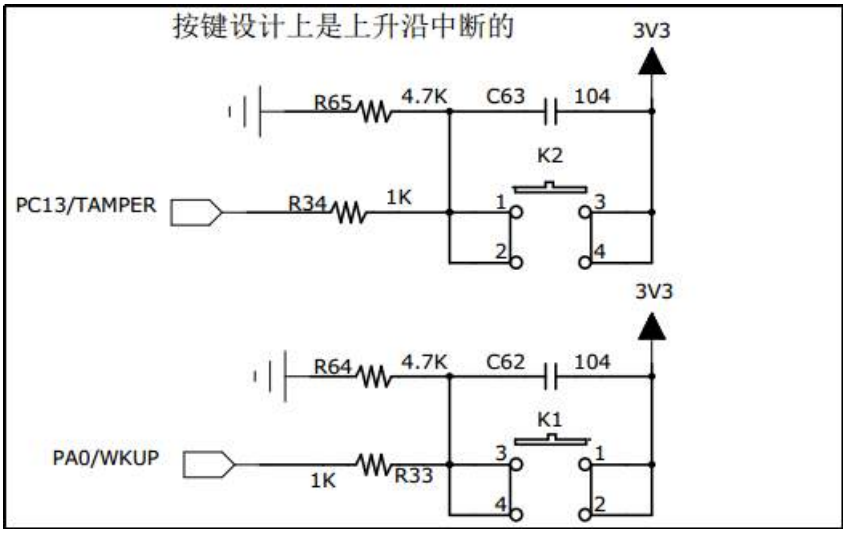
本章参考资料：《STM32F4xx 参考手册》、《Cortex®-M4 编程手册》。

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。

13.1 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_2*，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。





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。

13.2 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。

13.2.1 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。

13.2.2 代码分析

13.2.2.1 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_1*。

列表 1: 代码清单 13-1 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
```

(下页继续)

(续上页)

```
*/

 #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。

13.2.2.2 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*。

列表 2: 代码清单 13-2 按键 GPIO 初始化函数

```
void Key_GPIO_Config(void)

{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 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 */
```

(下页继续)

(续上页)

```
    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“LED GPIO 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) 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。

13.2.2.3 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 3: 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 函数用于扫描按键状态。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。

13.2.2.4 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 4: 代码清单 13-4 按键检测主函数

```
int main(void)

{

    /* 系统时钟初始化成 168MHz */

    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 ) {
```

(下页继续)

(续上页)

```
            /*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。

13.2.3 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 14 章　启动文件详解

本章参考资料《STM32F4xx 中文参考手册》的第十章-中断和事件：表 45. STM32F405xx/07xx 和 STM32F415xx/17xx 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。

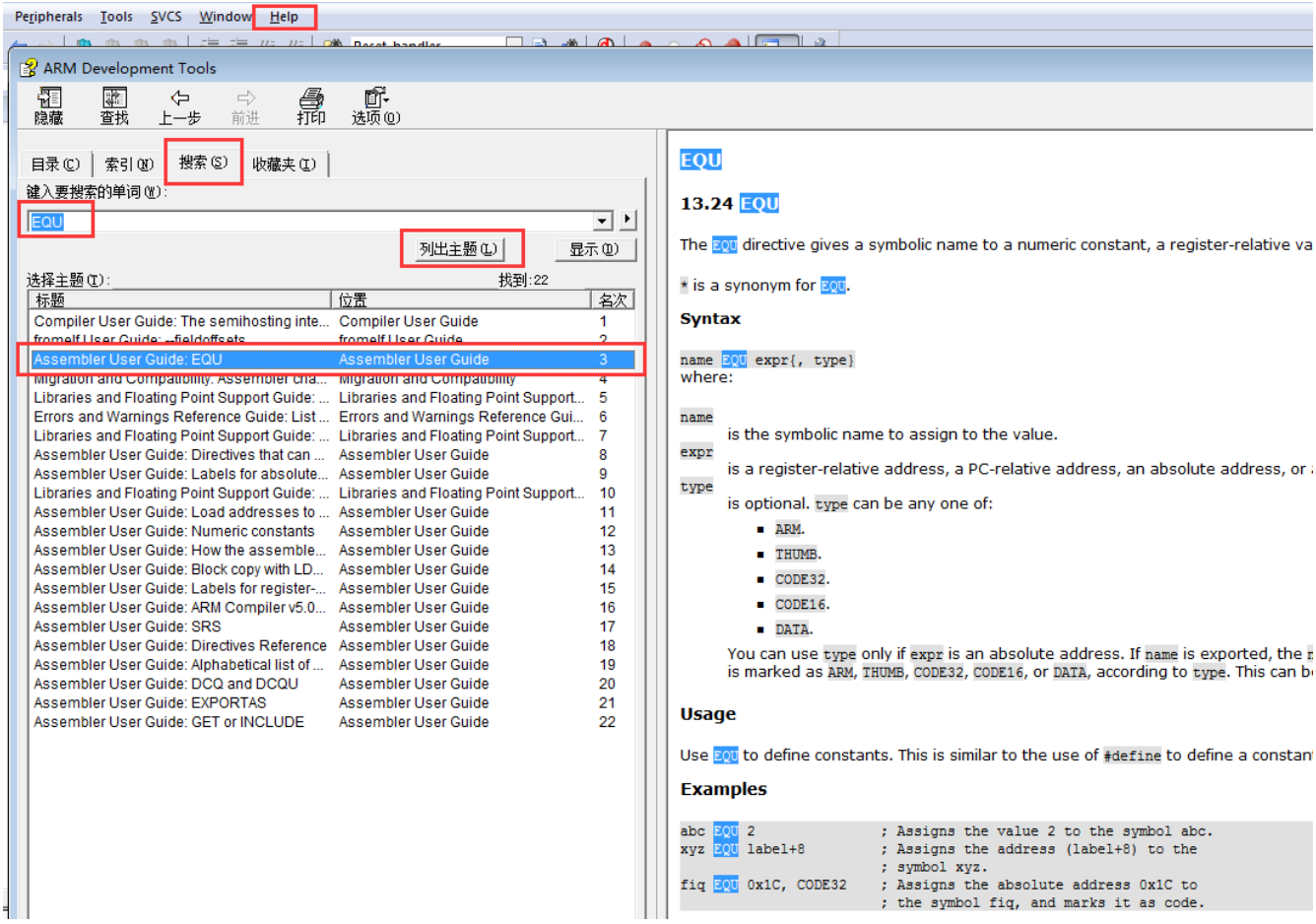
14.1 启动文件简介

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
4. 配置系统时钟
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

14.2 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
PRE-SERVE8	当前文件堆栈需按照 8 字节对齐
EX-PORT	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。
IM-PORT	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

14.3 启动文件代码讲解

14.3.1 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。

14.3.2 Heap 堆

```
Heap_Size       EQU 0x00000200

            AREA HEAP, NOINIT, READWRITE, ALIGN=3

__heap_base
```

(下页继续)

(续上页)

```
            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。

1 PRESERVE8

2 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。

14.3.3 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 14-2 F407 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	-	窗口看门狗中断	0X0000 0040
1	8	可编程	PVD	连接 EXTI 线的可编程电压检测中断	0X0000 0044
2	9	可编程	TAMP_STAMP	连接 EXTI 线的入侵和时间戳中断	0X0000 0048
中间部分省略，详情请参考 STM32F4xx 中文参考手册》第十章-中断和事件-向量表部分					
79	86	可编程	CRYP	CRYP 加密全局中断	0X 0000 017C
80	87	可编程	HASH_RNG	哈希和随机数发生器全局中断	0X 0000 0180
81	88	可编程	FPU	FPU 全局中断	0X 0000 0184

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

          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
```

(下页继续)

(续上页)

```
        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。

14.3.4 复位程序

```
1 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，最终调用 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。如果我们在这里不调用 __main，那么程序最终就不会调用我们 C 文件里面的 main，如果是调皮的用户就可以修改主函数的名称，然后在这里面 IMPORT 你写的主函数名称即可。

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

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

这个时候你在 C 文件里面写的主函数名称就不是 main 了，而是 user_main 了。

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

14.3.5 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。

14.3.6 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
```

(下页继续)

(续上页)

```
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

    LDR R1, =(Stack_Mem + Stack_Size)

    LDR R2, = (Heap_Mem + Heap_Size)

    LDR R3, = Stack_Mem

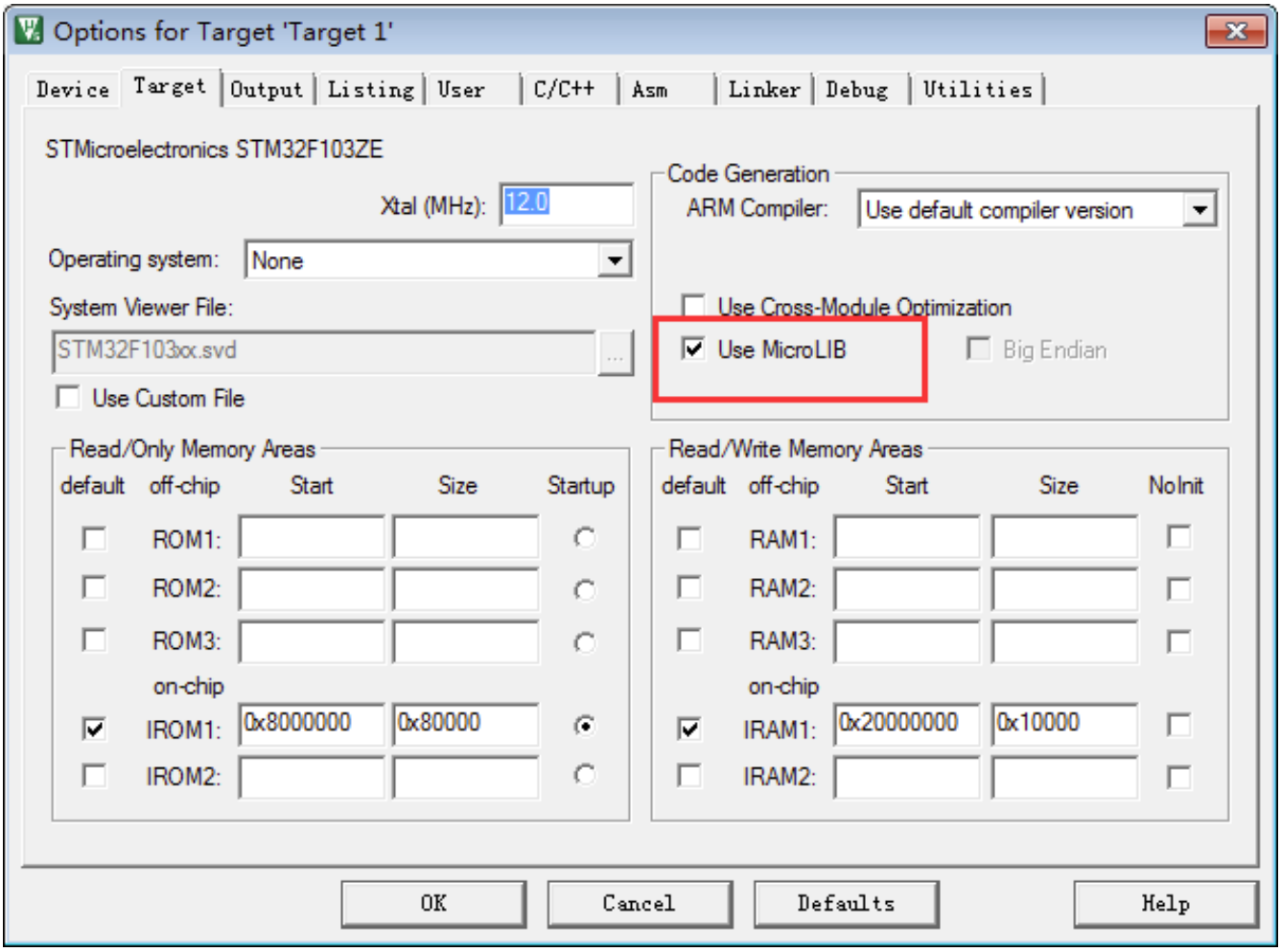
    BX LR

    ALIGN

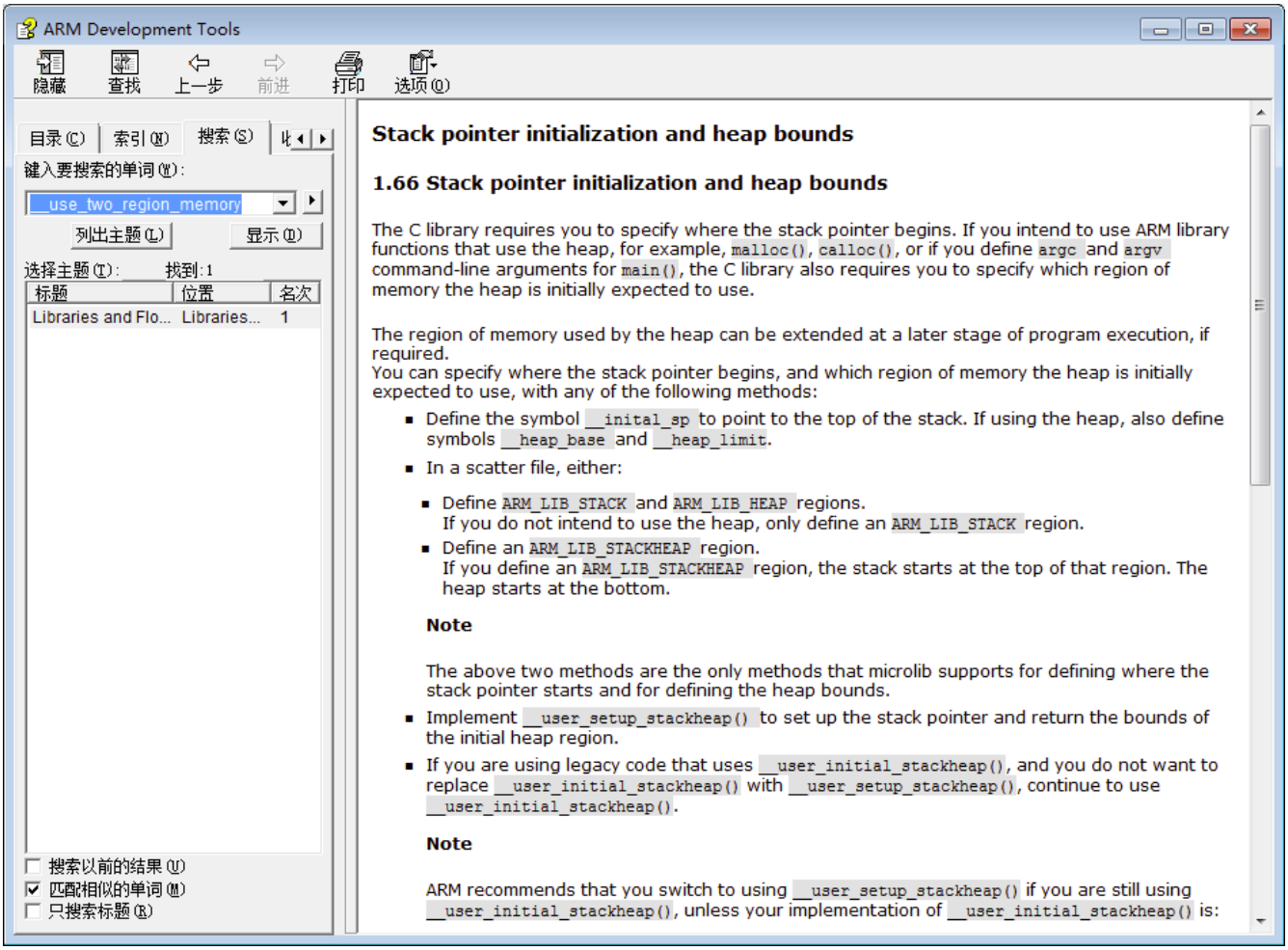
    ENDIF

    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 15 章　RCC—使用 HSE/HSI 配置时钟

本章参考资料：《STM32F4xx 中文参考手册》RCC 章节。

学习本章时，配合《STM32F4xx 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。

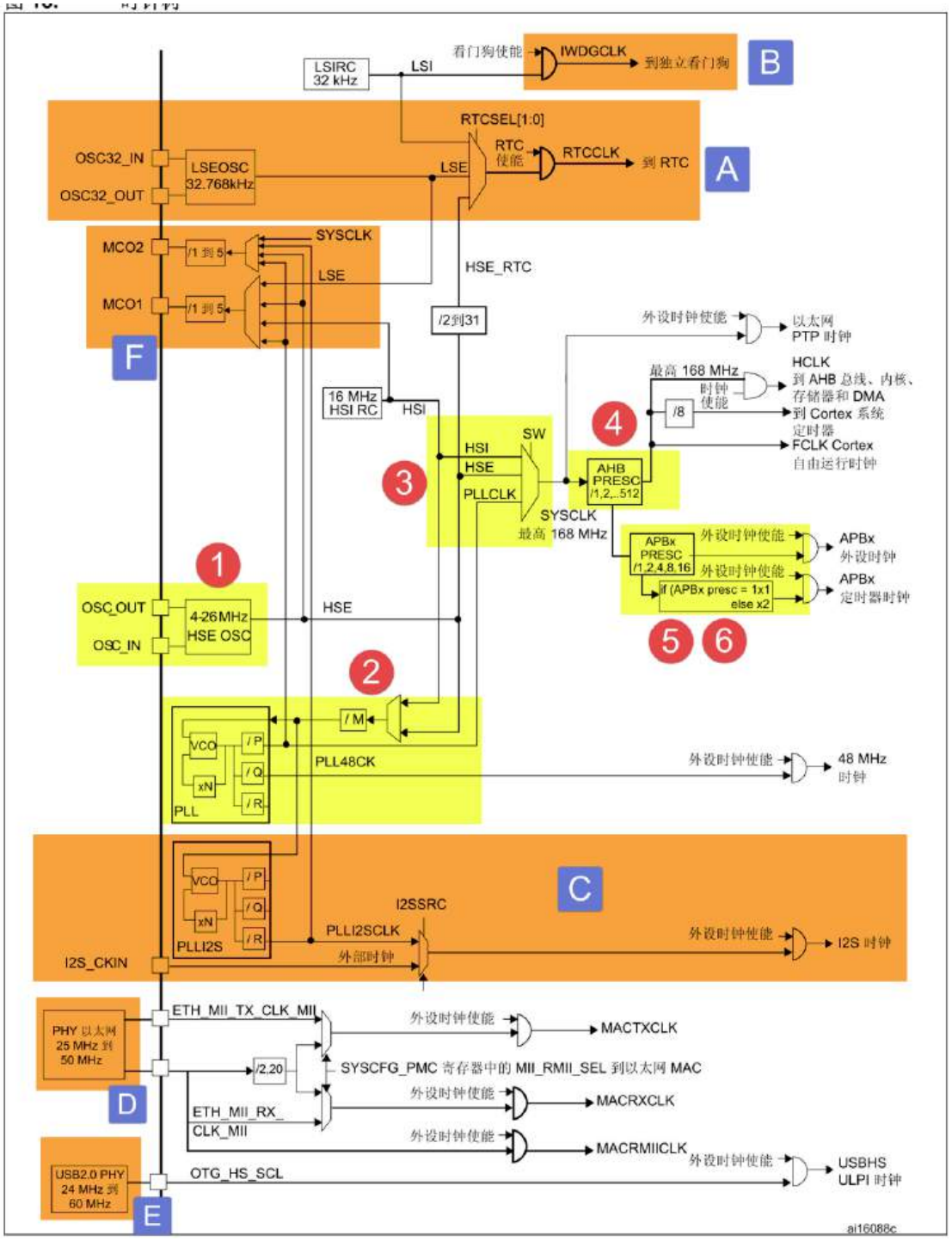
RCC： reset clock control 复位和时钟控制器。本章我们主要讲解时钟部分，特别是要着重理解时钟树，理解了时钟树，F407 的一切时钟的来龙去脉都会了如指掌。

15.1 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：HCLK = SYSCLK=PLLCLK = 168M，PCLK1=HCLK/2 = 84M，PCLK1=HCLK/4 = 42M。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。

15.2 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：SetSysClock(); 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：HCLK = SYSCLK=PLLCLK = 168M，PCLK1=HCLK/2 = 84M，PCLK1=HCLK/4 = 42M 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。


LSIRC 32 kHz
LSI
看门狗使能
IWDGCLK
到独立看门狗
B
RTCSEL[1:0]
OSC32_IN
OSC32_OUT
LSEOSC 32.768kHz
LSE
RTC 使能
RTCCLK
到 RTC
A
MCO2
MCO1
/1 到 5
SYSCLK
LSE
HSE_RTC
/2到31
F
外设时钟使能
以太网 PTP 时钟
最高 168 MHz
时钟使能
/8
HCLK 到 AHB 总线、内核、存储器和 DMA
到 Cortex 系统定时器
FCLK Cortex 自由运行时钟
16 MHz HSI RC
HSI
4
SW
HSI
HSE
PLLCLK
3
AHB PRESC /1,2,..512
SYSCLK 最高 168 MHz
APBx PRESC /1,2,4,8,16
外设时钟使能
APBx 外设时钟
if (APBx presc = 1x1 else x2
APBx 定时器时钟
5
6
1
OSC_OUT
OSC_IN
4-26MHz HSE OSC
HSE
2
/ M
VCO
/ P
/ Q
/ R
xN
PLL
PLL48CK
外设时钟使能
48 MHz 时钟
C
I2SSRC
PLLI2S
PLLI2SCLK
外部时钟
I2S_CKIN
外设时钟使能
I2S 时钟
PHY 以太网 25 MHz 到 50 MHz
D
ETH_MII_TX_CLK_MII
外设时钟使能
MACTXCLK
/2,20
SYSCFG_PMC 寄存器中的 MII_RMII_SEL 到以太网 MAC
外设时钟使能
MACRXCLK
ETH_MII_RX_CLK_MII
外设时钟使能
MACRMIICLK
USB2.0 PHY 24 MHz 到 60 MHz
E
OTG_HS_SCL
外设时钟使能
USBHS ULPI 时钟
ai16088c

15.2.1 系统时钟

15.2.1.1 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 4-26MHZ 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 我们使用 25M 的无源晶振。如果我们使用 HSE 或者 HSE 经过 PLL 倍频之后的时钟作为系统时钟 SYSCLK，当 HSE 故障时候，不仅 HSE 会被关闭，PLL 也会被关闭，此时高速的内部时钟时钟信号 HSI 会作为备用的系统时钟，直到 HSE 恢复正常，HSI=16M。

15.2.1.2 锁相环 PLL

PLL 的主要作用是对时钟进行倍频，然后把时钟输出到各个功能部件。PLL 有两个，一个是主 PLL，另外一个是专用的 PLLI2S，他们均由 HSE 或者 HSI 提供时钟输入信号。

主 PLL 有两路的时钟输出，第一个输出时钟 PLLCLK 用于系统时钟，F407 里面最高是 168M，第二个输出用于 USB OTG FS 的时钟（48M）、RNG 和 SDIO 时钟（<=48M）。专用的 PLLI2S 用于生成精确时钟，给 I2S 提供时钟。

HSE 或者 HSI 经过 PLL 时钟输入分频因子 M（2~63）分频后，成为 VCO 的时钟输入，VCO 的时钟必须在 1~2M 之间，我们选择 HSE=25M 作为 PLL 的时钟输入，M 设置为 25，那么 VCO 输入时钟就等于 1M。

VCO 输入时钟经过 VCO 倍频因子 N 倍频之后，成为 VCO 时钟输出，VCO 时钟必须在 192~432M 之间。我们配置 N 为 336，则 VCO 的输出时钟等于 336M。如果要把系统时钟超频，就得在 VCO 倍频系数 N 这里做手脚。PLLCLK_OUTMAX = VCOCLK_OUTMAX/P_MIN = 432/2=216M，即 F407 最高可超频到 216M。

VCO 输出时钟之后有三个分频因子：PLLCLK 分频因子 p，USB OTG FS/RNG/SDIO 时钟分频因子 Q，分频因子 R（F446 才有，F407 没有）。p 可以取值 2、4、6、8, 我们配置为 2，则得到 PLLCLK=168M。Q 可以取值 4~15，但是 USB OTG FS 必须使用 48M，Q=VCO 输出时钟 336/48=7。有关 PLL 的配置有一个专门的 RCC PLL 配置寄存器 RCC_PLLCFGR，具体描述看手册即可。

PLL 的时钟配置经过，稍微整理下可由如下公式表达：

VCOCLK_IN = PLLCLK_IN / M = HSE / 25 = 1M

VCOCLK_OUT = VCOCLK_IN * N = 1M * 336 = 336M

PLLCLK_OUT=VCOCLK_OUT/P=336/2=168M

USBCLK = VCOCLK_OUT/Q=336/7=48。。

15.2.1.3 3. 系统时钟 SYSCLK

系统时钟来源可以是：HSI、PLLCLK、HSE，具体的由时钟配置寄存器 RCC_CFGR 的 SW 位配置。我们这里设置系统时钟：SYSCLK = PLLCLK = 168M。如果系统时钟是由 HSE 经过 PLL 倍频之后的 PLLCLK 得到，当 HSE 出现故障的时候，系统时钟会切换为 HSI=16M，直到 HSE 恢复正常为止。

15.2.1.4 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 AHB 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 RCC_CFGR 的 HPRE 位设置。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=168M。

15.2.1.5 APB2 总线时钟 HCLK2

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 RCC_CFGR 的 PPRE2 位设置。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 2 分频，即 PCLK2 = HCLK /2= 84M。

15.2.1.6 APB1 总线时钟 HCLK1

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]，具体由时钟配置寄存器 RCC_CFGR 的 PPRE1 位设置。HCLK1 属于低速的总线时钟，最高为 42M，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 4 分频，即 PCLK1 = HCLK/4 = 42M。

15.2.1.7 设置系统时钟库函数

上面的 6 个步骤对应的设置系统时钟库函数具体见代码清单 15 1，为了方便阅读，已经把跟 407 不相关的代码删掉，把英文注释翻译成了中文，并把代码标上了序号，总共 6 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

```
/**
* @brief  System Clock Configuration
*         The system Clock is configured as follow :
*            System Clock source            = PLL (HSE)
*            SYSCLK(Hz)                     = 168000000
*            HCLK(Hz)                       = 168000000
*            AHB Prescaler                  = 1
*            APB1 Prescaler                 = 4
*            APB2 Prescaler                 = 2
*            HSE Frequency(Hz)              = 8000000
*            PLL_M                          = 25
*            PLL_N                          = 336
*            PLL_P                          = 2
*            PLL_Q                          = 7
*            VDD(V)                         = 3.3
*            Main regulator output voltage  = Scale1 mode
*            Flash Latency(WS)              = 5
```

(下页继续)

(续上页)

```
* @param  None
* @retval None
*/
static void SystemClock_Config(void)
{
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_OscInitTypeDef RCC_OscInitStruct;

    /* Enable Power Control clock */
    __HAL_RCC_PWR_CLK_ENABLE();

    /* The voltage scaling allows optimizing the power consumptio
    when the device is clocked below the maximum system frequency,
    to update the voltage scaling value
        regarding system frequency refer to product datasheet.  */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /* Enable HSE Oscillator and activate PLL with HSE as source */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 25;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        while (1) {};
    }

    /* Select PLL as system clock source and configure the HCLK,
    PCLK1 and PCLK2 clocks dividers */
```

(下页继续)

(续上页)

```
    RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK |
    RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1| RCC_CLOCKTYPE_PCLK2);
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct,FLASH_LATENCY_5) != HAL_OK)
↪{
        while (1) {};
    }

    /* STM32F405x/407x/415x/417x Revision Z devices:prefetch is supported ␣
↪*/
    if (HAL_GetREVID() == 0x1001) {
        /* Enable the Flash prefetch */
        __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
    }
}
```

15.2.2 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。

15.2.2.1 A、RTC 时钟

RTCCLK 时钟源可以是 HSE 1 MHz（HSE 由一个可编程的预分频器分频）、LSE 或者 LSI 时钟。选择方式是编程 RCC 备份域控制寄存器 (RCC_BDCR) 中的 RTCSEL[1:0] 位和 RCC 时钟配置寄存器 (RCC_CFGR) 中的 RTCPRE[4:0] 位。所做的选择只能通过复位备份域的方式修改。我们通常的做法是由 LSE 给 RTC 提供时钟，大小为 32.768KHZ。LSE 由外接的晶体谐振器产生，所配的谐振电容精度要求高，不然很容易不起震。

15.2.2.2 B、独立看门狗时钟

独立看门狗时钟由内部的低速时钟 LSI 提供，大小为 32KHZ。

15.2.2.3 C、I2S 时钟

I2S 时钟可由外部的时钟引脚 I2S_CKIN 输入，也可由专用的 PLLI2SCLK 提供，具体的由 RCC 时钟配置寄存器 (RCC_CFGR) 的 I2SSCR 位配置。我们在使用 I2S 外设驱动 W8978 的时候，使用的时钟是 PLLI2SCLK，这样就可以省掉一个有源晶振。

15.2.2.4 D、PHY 以太网时钟

F407 要想实现以太网功能，除了有本身内置的 MAC 之外，还需要外接一个 PHY 芯片，常见的 PHY 芯片有 DP83848 和 LAN8720，其中 DP83848 支持 MII 和 RMII 接口，LAN8720 只支持 RMII 接口。野火 F407 开发板用的是 RMII 接口，选择的 PHY 芯片是 LAB8720。使用 RMII 接口的好处是使用的 IO 减少了一半，速度还是跟 MII 接口一样。当使用 RMII 接口时，PHY 芯片只需输出一路时钟给 MCU 即可，如果是 MII 接口，PHY 芯片则需要提供两路时钟给 MCU。

15.2.2.5 E、USB PHY 时钟

F407 的 USB 没有集成 PHY，要想实现 USB 高速传输的话，必须外置 USB PHY 芯片，常用的芯片是 USB3300。当外接 USB PHY 芯片时，PHY 芯片需要给 MCU 提供一个时钟。

外扩 USB3300 会占用非常多的 IO，跟 SDRAM 和 RGB888 的 IO 会复用的很厉害，鉴于 USB 高速传输用的比较少，野火 F407 霸天虎就没有外扩这个芯片。

15.2.2.6 F、MCO 时钟输出

MCO 是 microcontroller clock output 的缩写，是微控制器时钟输出引脚，主要作用是可以对外提供时钟，相当于一个有源晶振。F407 中有两个 MCO，由 PA8/PC9 复用所得。MCO1 所需的时钟源通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 MCO1PRE[2:0] 和 MCO1[1:0] 位选择。MCO2 所

需的时钟源通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 MCO2PRE[2:0] 和 MCO2 位选择。有关 MCO 的 IO、时钟选择和输出速率的具体信息如下表所示：

时钟输出	IO	时钟来源	最大输出速率
MCO1	PA8	HSI、LSE、HSE、PLLCLK	100M
MCO2	PC9	HSE、PLLCLK、SYSCLK、PLLI2SCLK	100M

15.3 配置系统时钟实验

15.3.1 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。F407 系统时钟最高为 168M，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 216M。

如果我们使用库函数编程，当程序来到 main 函数之前，启动文件：startup_stm32f40xxx.s 已经调用 SystemInit() 函数把系统时钟初始化成 168MHZ，SystemInit() 在库文件：system_stm32f4xx.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。

15.3.2 使用 HSI

当 HSE 直接或者间接（HSE 经过 PLL 倍频）的作为系统时钟的时候，如果 HSE 发生故障，不仅 HSE 会被关闭，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=16M，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，重新设置系统频率为 168M，让系统恢复正常使用。但这只是权宜之计，并非万全之策，最好的方法还是要采取相应的补救措施并报警，然后修复 HSE。临时使用 HSI 只是为了把损失降低到最小，毕竟 HSI 较于 HSE 精度还是要低点。

F103 系列中，使用 HSI 最大只能把系统设置为 64M，并不能跟使用 HSE 一样把系统时钟设置为 72M，究其原因是 HSI 在进入 PLL 倍频的时候必须 2 分频，导致 PLL 倍频因子调到最大也只能到 64M，而 HSE 进入 PLL 倍频的时候则不用 2 分频。

在 F407 中，无论是使用 HSI 还是 HSE 都可以把系统时钟设置为 168M，因为 HSE 或者 HSI 在进入 PLL 倍频的时候都会被分频为 1M 之后再倍频。

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。

15.3.3 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。

15.3.4 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。

15.3.4.1 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，设置 VCO 输入时钟分频因子 PLL_M，设置 VCO 输出时钟倍频因子 PLL_N，设置 PLLCLK 时钟分频因子 PLL_P，设置 OTG FS,SDIO,RNG 时钟分频因子 PLL_Q。

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

15.3.4.2 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。

15.3.4.2.1 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

```
/*
* 使用 HSE 时，设置系统时钟的步骤
* 1、开启 HSE ，并等待 HSE 稳定
* 2、设置 AHB、APB2、APB1 的预分频因子
* 3、设置 PLL 的时钟来源
*     设置 VCO 输入时钟 分频因子          m
*     设置 VCO 输出时钟 倍频因子          n
*     设置 PLLCLK 时钟分频因子             p
*     设置 OTG FS,SDIO,RNG 时钟分频因子 q
* 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
*/
/*
* m: VCO 输入时钟 分频因子，取值 2~63
* n: VCO 输出时钟 倍频因子，取值 50~432
* p: PLLCLK 时钟分频因子  ，取值 2，4，6，8
* q: OTG FS,SDIO,RNG 时钟分频因子，取值 4~15
```

(下页继续)

(续上页)

```
* 函数调用举例，使用 HSE 设置时钟
* SYSCLK=HCLK=168MHz,PCLK2=HCLK/2=84MHz,PCLK1=HCLK/4=42MHz
* HSE_SetSysClock(25, 336, 2, 7);
*
HSE 作为时钟来源，经过 PLL 倍频作为系统时钟，这是通常的做法

*/
void HSE_SetSysClock(uint32_t m, uint32_t n, uint32_t p, uint32_t q)
{
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_OscInitTypeDef RCC_OscInitStruct;

    /* Enable Power Control clock */
    __HAL_RCC_PWR_CLK_ENABLE();

    /* The voltage scaling allows optimizing the power consumption when
    the device is clocked below the maximum system frequency, to update
    the voltage scaling value regarding system frequency refer to product
    datasheet.  */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /* Enable HSE Oscillator and activate PLL with HSE as source */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 25;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        while (1) {};
```

(下页继续)

(续上页)

```
    }

    /* Select PLL as system clock source and configure the HCLK,
    PCLK1 and PCLK2 clocks dividers */
    RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK |
                                   RCC_CLOCKTYPE_HCLK |
                                   RCC_CLOCKTYPE_PCLK1 |
                                   RCC_CLOCKTYPE_PCLK2);
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) !=
                            HAL_OK) {
        while (1) {};
    }

    /* STM32F405x/407x/415x/417x Revision Z devices: prefetch is supported␣
↪ */
    if (HAL_GetREVID() == 0x1001) {
        /* Enable the Flash prefetch */
        __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
    }
}
```

这个函数采用库函数编写，代码理解参考注释即可。函数有 4 个形参 m、n、p、q，具体说明见表 15-1 PLL 配置因子。

表 15-1 PLL 配置因子

形参	形参说明	取值范围
m	VCO 输入时钟分频因子	2~63
n	VCO 输出时钟倍频因子	192~432
p	PLLCLK 时钟分频因子	2/4/6/8
q	OTGFS,SDIO,RNG 时钟分频因子	4~15

HSE 我们使用 25M，参数 m 我们一般也设置为 25，所以我们需要修改系统时钟的时候只需要修改参数 n 和 p 即可，SYSCLK=PLLCLK=HSE/m*n/p。

函数调用举例：HSE_SetSysClock(25, 336, 2, 7) 把系统时钟设置为 168M，这个跟库里面的系统时钟配置是一样的。HSE_SetSysClock(25, 432, 2, 9) 把系统时钟设置为 216M，这个是超频，要慎用。

15.3.4.2.2 使用 HSI 配置系统时钟

```
/*
* 使用 HSI 时，设置系统时钟的步骤
* 1、开启 HSI ，并等待 HSI 稳定
* 2、设置 AHB、APB2、APB1 的预分频因子
* 3、设置 PLL 的时钟来源
*     设置 VCO 输入时钟 分频因子          m
*     设置 VCO 输出时钟 倍频因子          n
*     设置 SYSCLK 时钟分频因子             p
*     设置 OTG FS,SDIO,RNG 时钟分频因子 q
* 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
*/

/*
* m: VCO 输入时钟 分频因子，取值 2~63
* n: VCO 输出时钟 倍频因子，取值 192~432
```

(下页继续)

(续上页)

```
* p: PLLCLK 时钟分频因子  , 取值 2, 4, 6, 8
* q: OTG FS,SDIO,RNG 时钟分频因子, 取值 4~15
* 函数调用举例, 使用 HSI 设置时钟
* SYSCLK=HCLK=168M,PCLK2=HCLK/2=84M,PCLK1=HCLK/4=42M
* HSI_SetSysClock(16, 336, 2, 7);
* HSE 作为时钟来源, 经过 PLL 倍频作为系统时钟, 这是通常的做法
void HSI_SetSysClock(uint32_t m, uint32_t n, uint32_t p, uint32_t q)
{
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_OscInitTypeDef RCC_OscInitStruct;

    /* Enable Power Control clock */
    __HAL_RCC_PWR_CLK_ENABLE();

    /* The voltage scaling allows optimizing the power consumption
    when the device is clocked below the maximum system frequency,
    to update the voltage scaling value  regarding system
    frequency refer toproduct datasheet.  */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /* Enable HSE Oscillator and activate PLL with HSE as source */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 25;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        while (1) {};
    }
```

(下页继续)

(续上页)

```
    /* Select PLL as system clock source and configure the HCLK,
    PCLK1 and PCLK2 clocks dividers */
    RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK |
                                   RCC_CLOCKTYPE_HCLK |
                                   RCC_CLOCKTYPE_PCLK1 |
                                   RCC_CLOCKTYPE_PCLK2);
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) !=
        HAL_OK) {
        while (1) {};
    }

    /* STM32F405x/407x/415x/417x Revision Z devices: prefetch is supported ␣
↪*/
    if (HAL_GetREVID() == 0x1001) {
        /* Enable the Flash prefetch */
        __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
    }
}
```

这个函数采用库函数编写，代码理解参考注释即可。函数有 4 个形参 m、n、p、q，具体说明见表 15-2 PLL 配置因子。

表 15-2 PLL 配置因子

形参	形参说明	取值范围
m	VCO 输入时钟分频因子	2~63
n	VCO 输出时钟倍频因子	192~432
p	PLLCLK 时钟分频因子	2/4/6/8
q	OTGFS,SDIO,RNG 时钟分频因子	4~15

HSI 为 16M，参数 m 我们一般也设置为 16，所以我们需要修改系统时钟的时候只需要修改参数 n 和 p 即可，SYSCLK=PLLCLK=HSI/m*n/p。

函数调用举例：HSI_SetSysClock(16, 336, 2, 7) 把系统时钟设置为 168M，这个跟库里面的系统时钟配置是一样的。HSI_SetSysClock(16, 432, 2, 9) 把系统时钟设置为 216M，这个是超频，要慎用。

15.3.4.2.3 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。

15.3.4.2.4 MCO 输出

在 F407 中，PA8/PC9 可以复用为 MCO1/2 引脚，对外提供时钟输出，我们也可以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野火 F407 开发板中 PA8 与 PC9 都引出来了，在摄像头接口位置，都可以使用示波器来监控波形。

15.3.4.2.5 主函数

```
 // 使用 HSE 或者 HSI 配置系统时钟
#include "stm32f4xx.h"
#include "./rcc/bsp_clkconfig.h"
#include "./led/bsp_led.h"

 void Delay(__IO u32 nCount);

/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    // 程序来到 main 函数之前，启动文件：statup_stm32f4xx.s 已经调用
    // SystemInit() 函数把系统时钟初始化成 168MHZ
    // SystemInit() 在 system_stm32f4xx.c 中定义
    // 如果用户想修改系统时钟，可自行编写程序修改
    // 重新设置系统时钟，这时候可以选择使用 HSE 还是 HSI

    // 使用 HSE，配置系统时钟为 168M
    HSE_SetSysClock(25, 336, 2, 7);

    //系统时钟超频到 216M 爽一下，最高是 216M，别往死里整
    //HSE_SetSysClock(25, 432, 2, 9);

    // 使用 HSI，配置系统时钟为 168M
//  HSI_SetSysClock(16, 336, 2, 7);

    // LED 端口初始化
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();

    // MCO GPIO  初始化
    MCO1_GPIO_Config();
    MCO2_GPIO_Config();

    // MCO1 输出 PLLCLK
    HAL_RCC_MCOConfig(RCC_MCO1,RCC_MCO1SOURCE_PLLCLK, RCC_MCODIV_2);

    // MCO2 输出 SYSCLK
    HAL_RCC_MCOConfig(RCC_MCO2,RCC_MCO2SOURCE_SYSCLK, RCC_MCODIV_2);

    while (1) {
        LED1( ON );       // 亮
        Delay(0x0FFFFF);
        LED1( OFF );       // 灭
        Delay(0x0FFFFF);
    }
}

void Delay(__IO uint32_t nCount)    //简单的延时函数
{
    for (; nCount != 0; nCount--);
}
```

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。

15.3.5 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。

第 16 章　STM32 中断应用概览

本章参考资料《STM32F4xx 中文参考手册》第十章-中断和事件、《ARM Cortex™-M4F 技术参考手册》-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。

16.1 异常类型

F407 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 10 个，外部中断有 82 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 stm32f4xx.h 这个头文件查询到，在 IRQn_Type 这个结构体里面包含了 F4 系列全部的异常声明。

表格 16-1 F407 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 F407 外部中断清单

表 16-2 F407 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	-	窗口看门狗中断	0X0000 0040
1	8	可编程	PVD	连接 EXTI 线的可编程电压检测中断	0X0000 0044
2	9	可编程	TAMP_STAMP	连接 EXTI 线的入侵和时间戳中断	0X0000 0048
中间部分省略，详情请参考 STM32F4xx 中文参考手册》第十章-中断和事件-向量表部分					
79	86	可编程	CRYP	CRYP 加密全局中断	0X 0000 017C
80	87	可编程	HASH_RNG	哈希和随机数发生器全局中断	0X 0000 0180
81	88	可编程	FPU	FPU 全局中断	0X 0000 0184

16.2 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-M4 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-M4 的 NVIC 的一个子集。

16.2.1 NVIC 寄存器简介

在函数库中，NVIC 的结构体定义可谓是颇有远虑，给每个寄存器都预览了很多位，恐怕为的是日后扩展功能。不过 STM32F407 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《ARM Cortex™-M4F 技术参考手册》-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 {
    __IO uint32_t ISER[8];         // 中断使能寄存器
    uint32_t RESERVED0[24];
    __IO uint32_t ICER[8];         // 中断清除寄存器
    uint32_t RSERVED1[24];
    __IO uint32_t ISPR[8];         // 中断使能悬起寄存器
    uint32_t RESERVED2[24];
    __IO uint32_t ICPR[8];         // 中断清除悬起寄存器
    uint32_t RESERVED3[24];
    __IO uint32_t IABR[8];         // 中断有效位寄存器
    uint32_t RESERVED4[56];
    __IO uint8_t  IP[240];         // 中断优先级寄存器 (8Bit wide)
    uint32_t RESERVED5[644];
    __O  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。

16.2.2 NVIC 中断配置固件库

函数库文件 core_cm4.h 的最后，还提供了 NVIC 的一些函数，这些函数遵循 CMSI 规则，只要是 Cortex-M4 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

16.3 优先级的定义

16.3.1 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（在 F407 中，x=0…981）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 CM4 芯片都会精简设计，以致实际上支持的优先级数减少，在 F407 中，只使用了高 4bit，如下所示：

表 16-4 F407 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。

16.3.2 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 NVIC_PriorityGroupConfig() 实现，有关 NVIC 中断相关的库函数都在库文件 misc.c 和 misc.h 中。

列表 2: 代码 16-2 中断优先级分组库函数

```
/**
* 配置中断优先级分组：抢占优先级和子优先级
* 形参如下：
* @arg NVIC_PriorityGroup_0: 0bit for 抢占优先级
*                            4 bits for 子优先级
* @arg NVIC_PriorityGroup_1: 1 bit for 抢占优先级
*                            3 bits for 子优先级
* @arg NVIC_PriorityGroup_2: 2 bit for 抢占优先级
*                            2 bits for 子优先级
* @arg NVIC_PriorityGroup_3: 3 bit for 抢占优先级
*                            1 bits for 子优先级
* @arg NVIC_PriorityGroup_4: 4 bit for 抢占优先级
*                            0 bits for 子优先级
* @ 注意 如果优先级分组为 0，则抢占优先级就不存在，优先级就全部由子优先级控制
*/
void NVIC_PriorityGroupConfig(uint32_t PriorityGroup)
```

(下页继续)

(续上页)

```
{
    // 检查参数
    assert_param(IS_NVIC_PRIORITY_GROUP(PriorityGroup));
    // 设置优先级分组
    NVIC_SetPriorityGrouping(PriorityGroup);
}
```

表 16-6 优先级分组真值表

优先级分组	主优先级	子优先级	描述
NVIC_PriorityGroup_0	0	0-15	主-0bit，子-4bit
NVIC_PriorityGroup_1	0-1	0-7	主-1bit，子-3bit
NVIC_PriorityGroup_2	0-3	0-3	主-2bit，子-2bit
NVIC_PriorityGroup_3	0-7	0-1	主-3bit，子-1bit
NVIC_PriorityGroup_4	0-15	0	主-4bit，子-0bit

16.4 中断编程

在配置每个中断的时候一般有 3 个编程要点：

1、使用 HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup) 函数配置中断优先级分组。一般默认是 NVIC_PRIORITYGROUP_4 分组 4。

2、使用 HAL_NVIC_SetPriority (IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority) 函数配置具体外设中断通道的抢占优先级和子优先级。

3、使用 HAL_NVIC_EnableIRQ 函数使能中断请求。

列表 3: 代码 16-3 IRQn_Type 中断源结构体

```
typedef enum IRQn {
    //Cortex-M4 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 stm32f4xx.h

    CRYP_IRQn                 = 79,
    HASH_RNG_IRQn             = 80,
    FPU_IRQn                  = 81,
} IRQn_Type;
```

1）PreemptPriority：抢占优先级，具体的值要根据优先级分组来确定，具体参考表 16 6 优先级分组真值表。

2）SubPriority：子优先级，具体的值要根据优先级分组来确定，具体参考表 16 6 优先级分组真值表。

4、编写中断服务函数在启动文件 startup_stm32f40xx.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，中断服务函数我们统一写在 stm32f4xx_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 17 章　EXTI—外部中断/事件控制器

本章参考资料：《STM32F4xx 中文参考手册》系统配置控制器以及中断和事件章节。

上一章我们已经详细介绍了 NVIC，对 STM32F4xx 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 STM32F4xx 控制器非常重要的一个资源。学习本章时，配合《STM32F4xx 中文参考手册》系统配置控制器以及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 STM32F40xxx 系列控制器资源讲解。

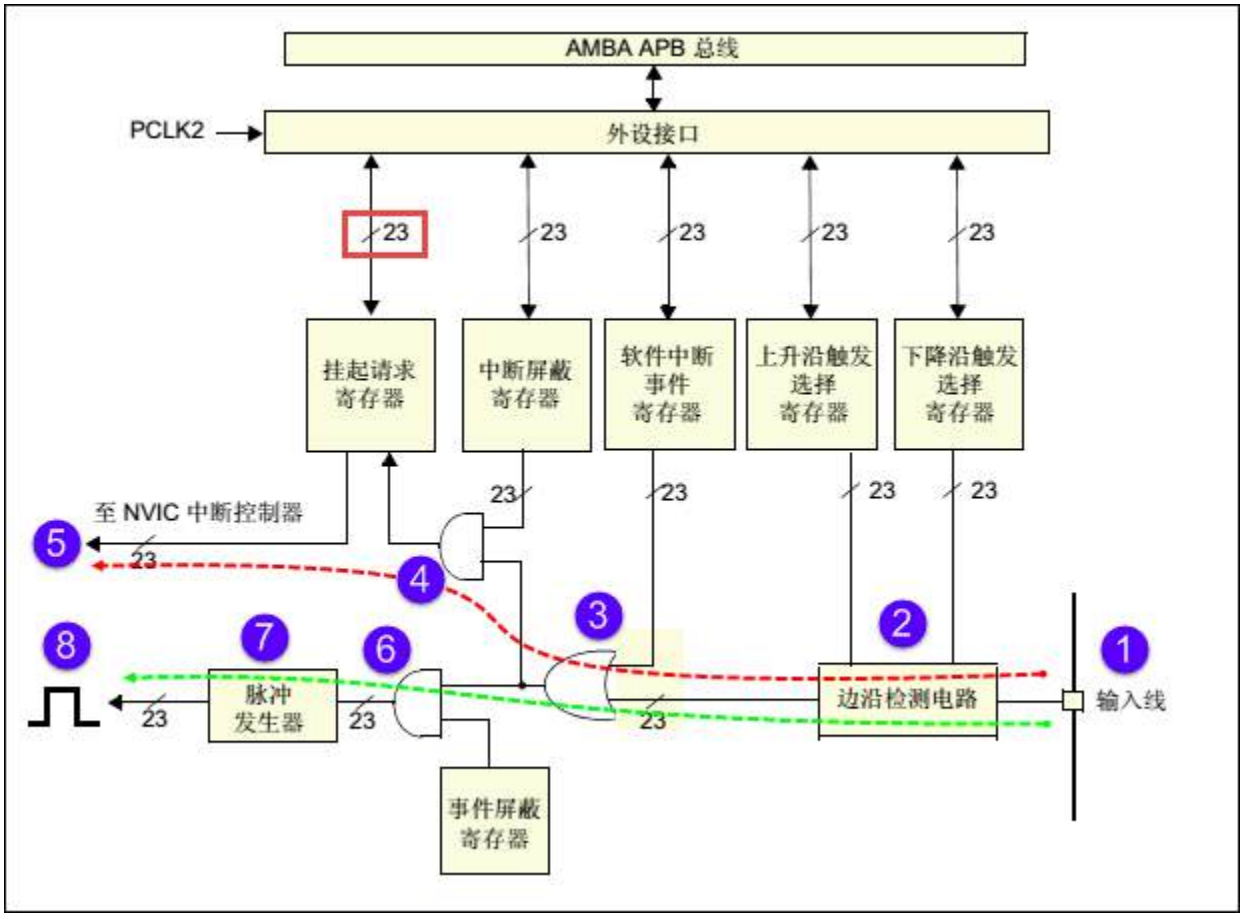
17.1 EXTI 简介

外部中断/事件控制器 (EXTI) 管理了控制器的 23 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。

17.2 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“23”字样，这个表示在控制器内部类似的信号线路有 23 个，这与 EXTI 总共有 23 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 22 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 23 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”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。

17.3 中断/事件线

EXTI 有 23 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* 。

七根特定外设中断/事件线由外设触发，具体用法参考《STM32F4xx 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）
EXTI20	USB OTG HS(在 FS 中配置) 唤醒事件
EXTI21	RTC 入侵和时间戳事件
EXTI22	RTC 唤醒事件

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。

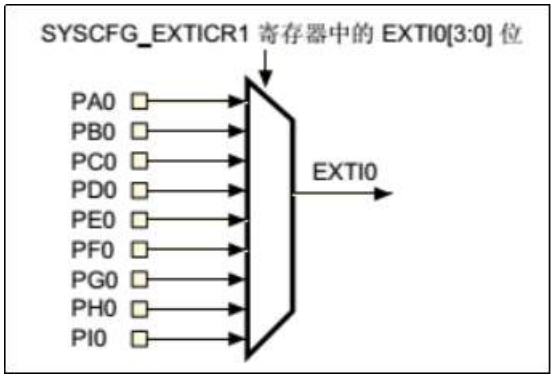


图 17-2 EXTI0 输入源选择

17.4 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。

17.4.1 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_3*。

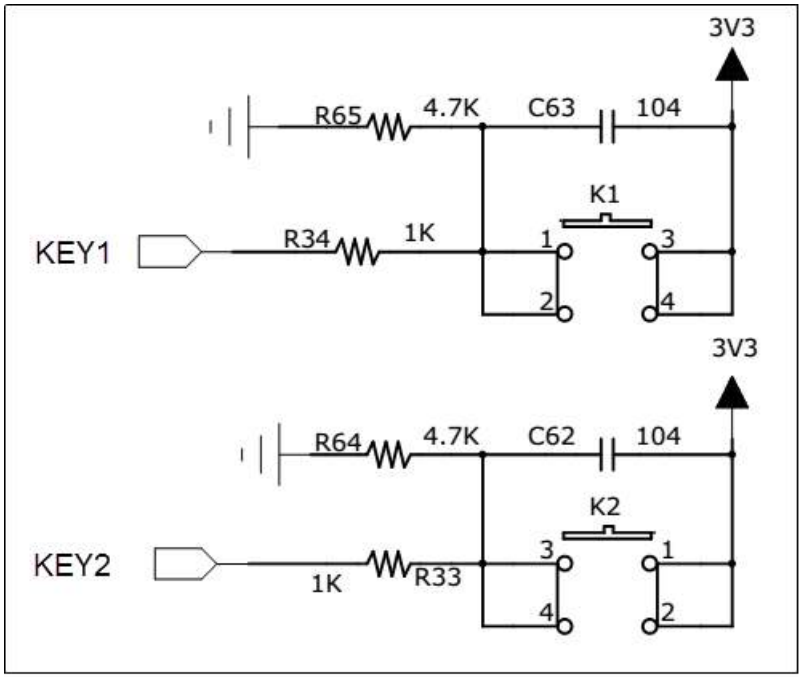


图 17-3 按键电路设计

17.4.2 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 stm32f4xx_it.h 文件中。

17.4.2.1 编程要点

1) 初始化 RGB 彩灯的 GPIO；

2) 开启按键 GPIO 时钟和 SYSCFG 时钟；

3) 配置 NVIC；

4) 配置按键 GPIO 为输入模式；

5) 将按键 GPIO 连接到 EXTI 源输入；

6) 配置按键 EXTI 中断/事件线；

7) 编写 EXTI 中断服务函数。

17.4.2.2 软件分析

17.4.2.2.1 按键和 EXTI 宏定义

列表 1: 代码清单 17-1 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
/*******************************************************/
```

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。

17.4.2.2 EXTI 中断配置

列表 2: 代码清单 17-2 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 */
    HAL_NVIC_EnableIRQ(KEY2_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。

17.4.2.2.3 EXTI 中断服务函数

列表 3: 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) {
```

(下页继续)

(续上页)

```
        // LED2 取反
        LED2_TOGGLE;
        //清除中断标志位
        __HAL_GPIO_EXTI_CLEAR_IT(KEY2_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。

17.4.2.2.4 主函数

列表 4: 代码清单 17-4 主函数

```
int main(void)
{
    /* 系统时钟初始化成 168 MHz */
    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 stm32f4xx_it.c 文件中的函数
```

(下页继续)

(续上页)

```
    *
    KEY1_IRQHandler 和 KEY2_IRQHandler，处理中断，反转 LED 灯。
    */
    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。

17.4.3 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。

第 18 章　SysTick—系统定时器

本章参考资料《ARM Cortex™-M4F 技术参考手册》-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 CM4 内核的外设，有关寄存器的定义和部分库函数都在 core_cm4.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。

18.1 SysTick 简介

SysTick—系统定时器是属于 CM4 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 168M。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 CM4 内核的外设，所以所有基于 CM4 内核的单片机都具有这个系统定时器，使得软件在 CM4 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。

18.2 SysTick 寄存器介绍

SysTick—系统定时有 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。

18.3 SysTick 定时实验

利用 SysTick 产生 1s 的时基，LED 以 1s 的频率闪烁。

18.3.1 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。

18.3.2 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.h 文件中。

18.3.2.1 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

18.3.2.2 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_cm4.h 中。

18.3.2.2.1 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=168M
    // 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为 168M，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。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 STM32F407 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 STM32F407 中，只有位 7:3 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);
```

18.3.2.2.2 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
```

(下页继续)

(续上页)

```
*/

void SysTick_Init(void)

{

    /* SystemFrequency / 1000 1ms 中断一次

    * SystemFrequency / 100000 10us 中断一次

    * 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。

18.3.2.2.3 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。

18.3.2.2.4 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。

18.3.2.2.5 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
```

(下页继续)

(续上页)

```
* @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。

18.3.2.2.6 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
* @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。

18.3.2.2.7 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 19 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。

19.1 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。

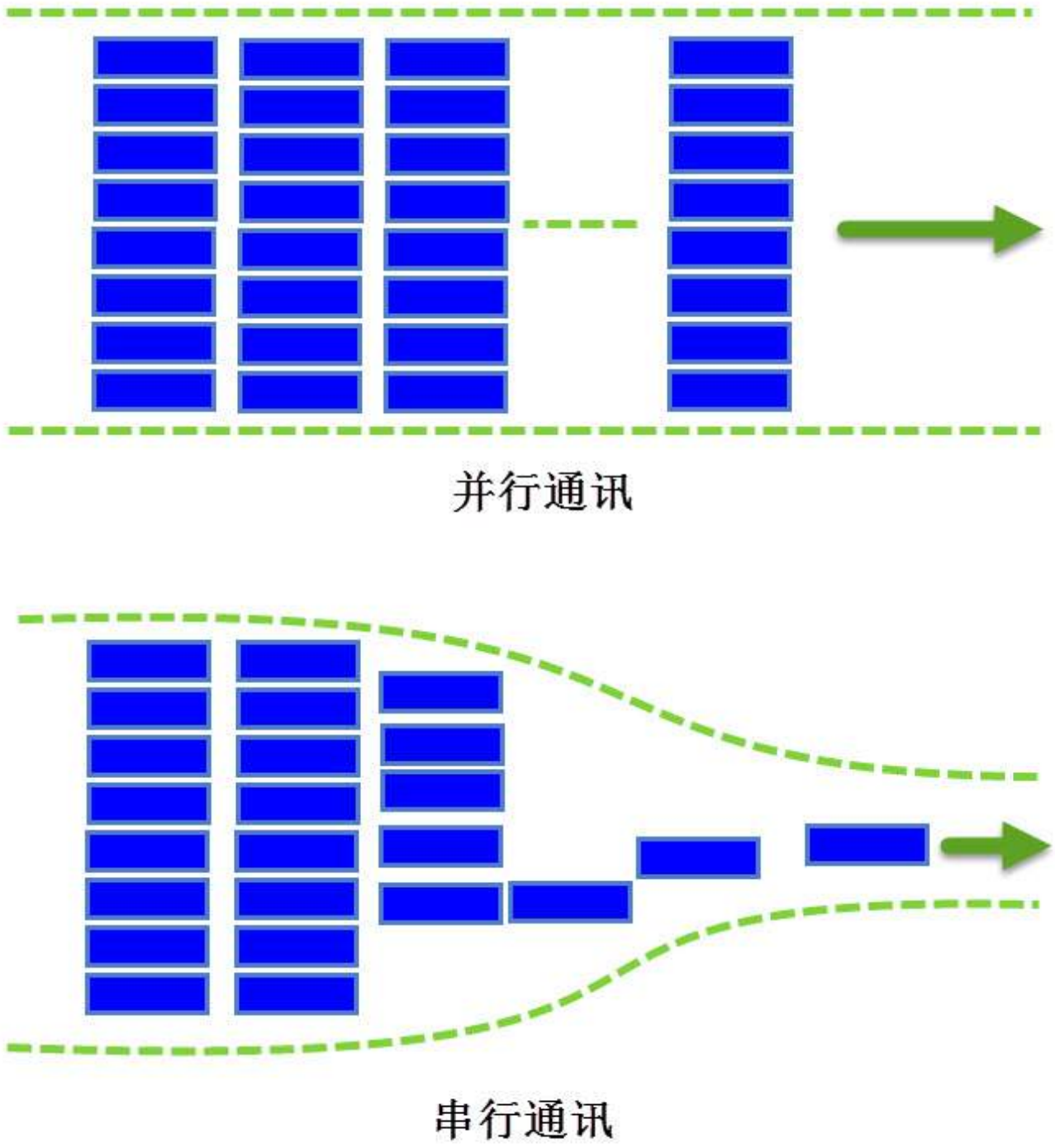


图 19-1 并行通讯与串行通讯的对比图

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。

19.2 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。

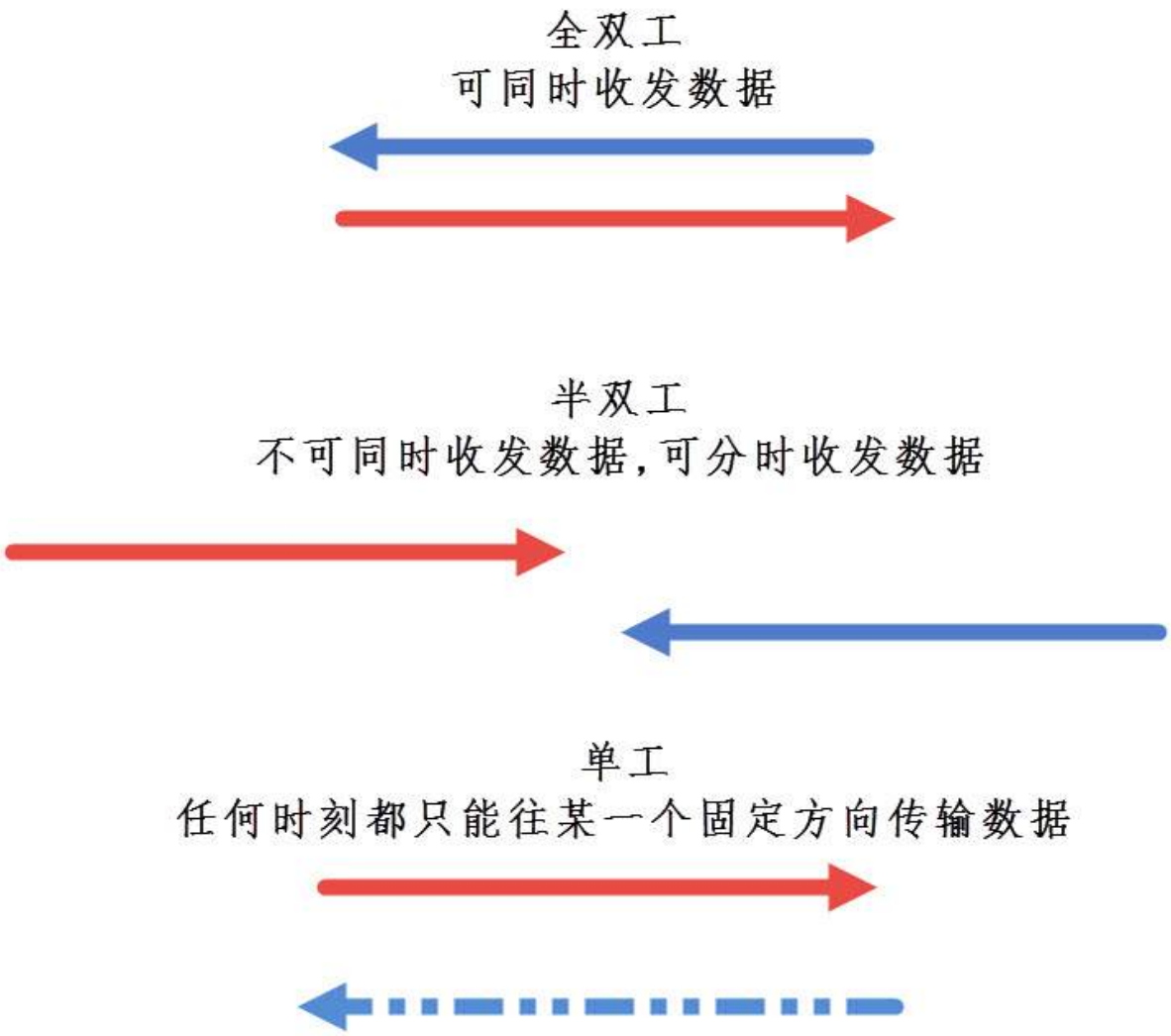


图 19-2 全双工、半双工及单工通讯

19.3 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。

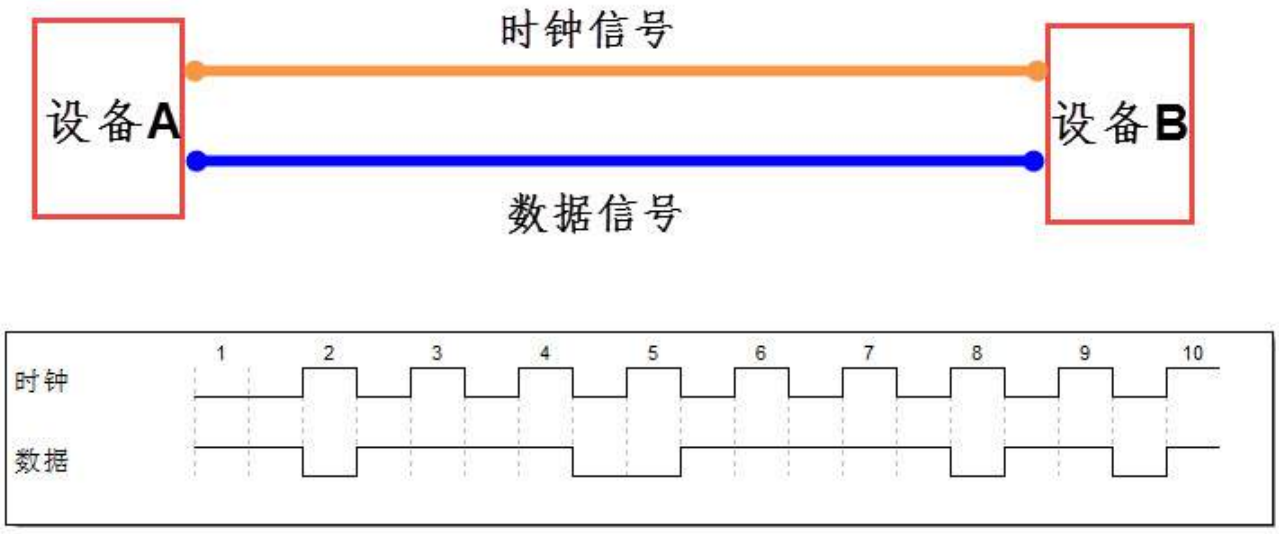


图 19-3 同步通讯

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。

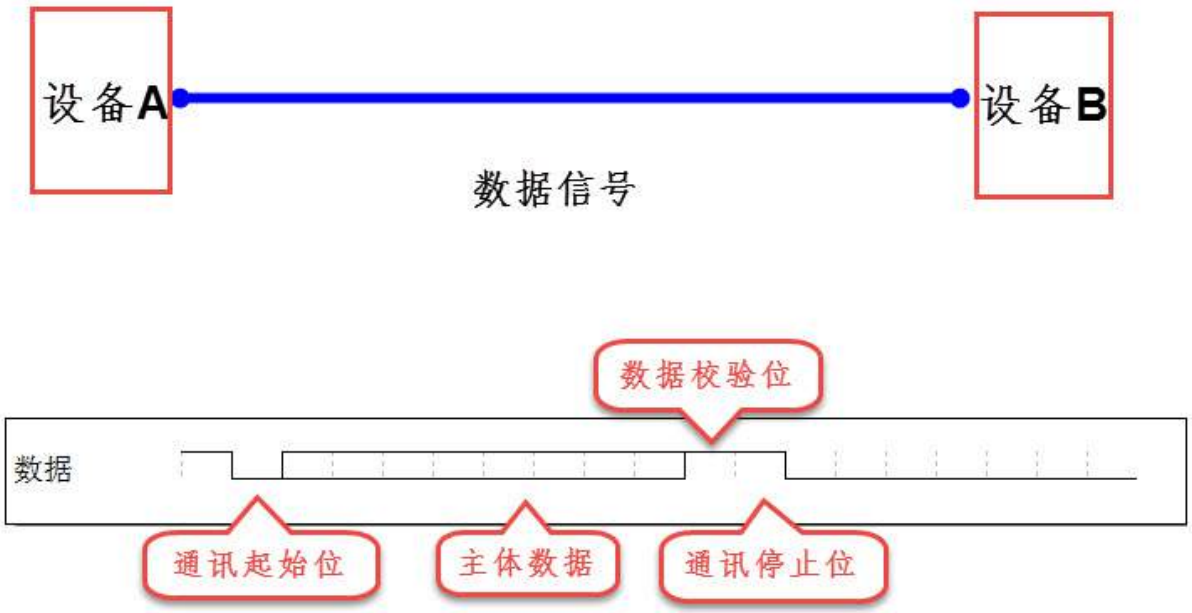


图 19-4 某种异步通讯

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。

19.4 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 20 章　USART—串口通讯

本章参考资料：《STM32F4xx 中文参考手册》USART 章节。

学习本章时，配合《STM32F4xx 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 STM32F4xxx 系列控制器资源讲解。

20.1 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。

20.1.1 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*。

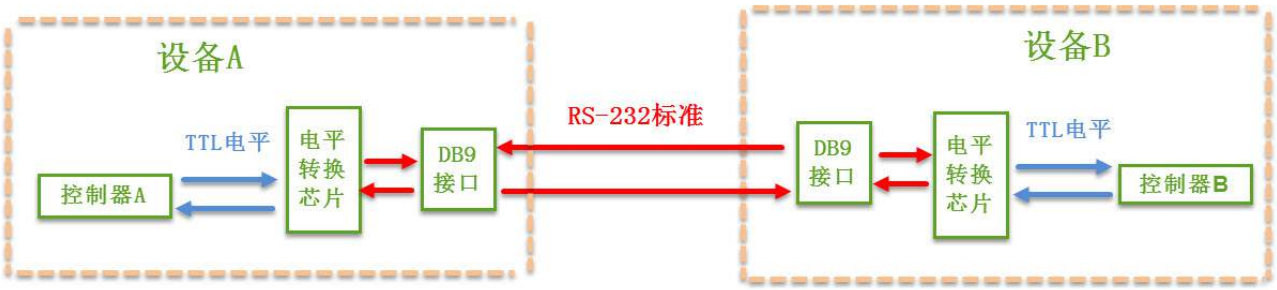


图 20-1 串口通讯结构图

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。

20.1.1.1 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*。

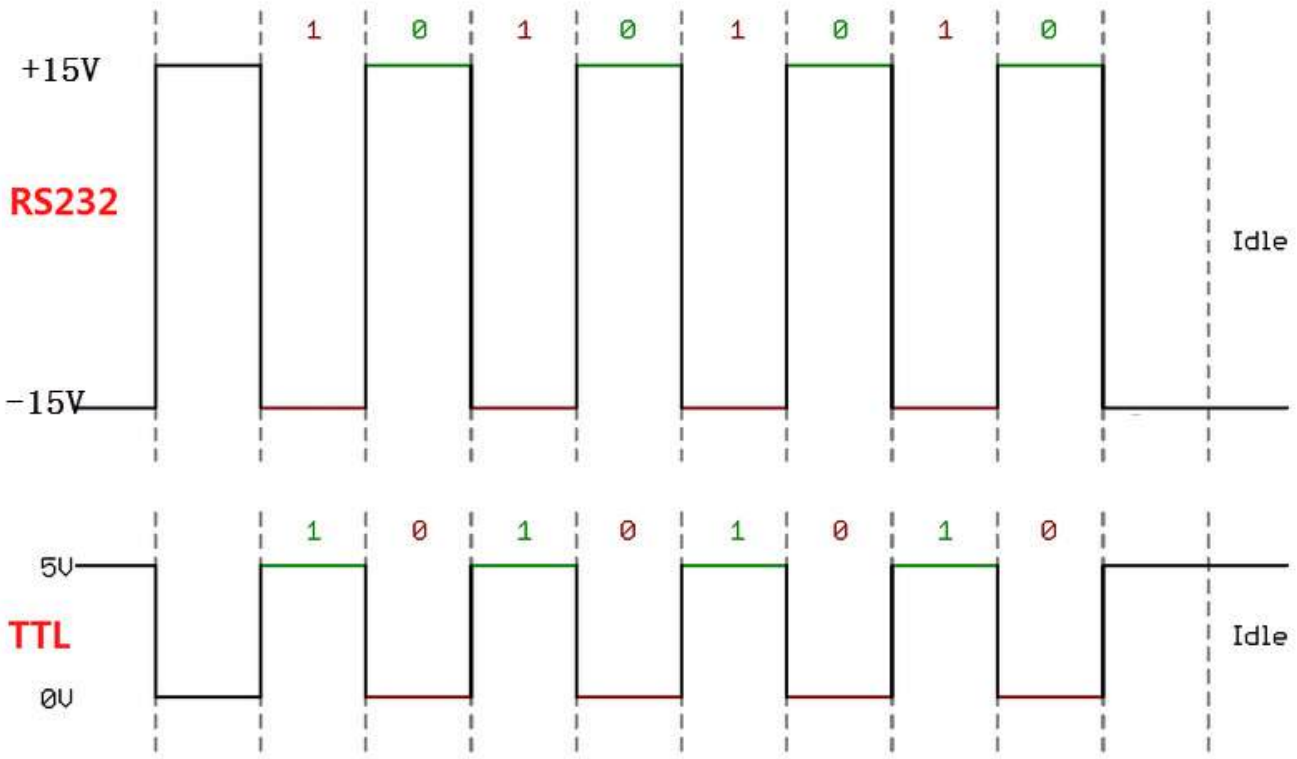


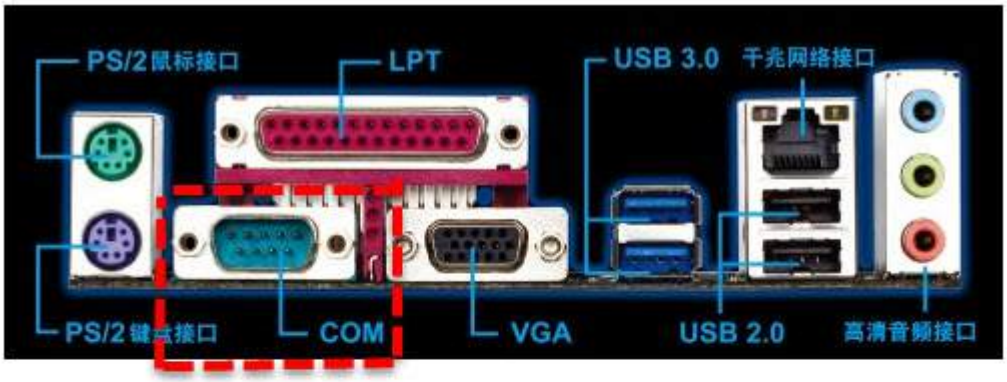
图 20-2 RS-232 与 TTL 电平标准下表示同一个信号

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。

20.1.1.2 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*。

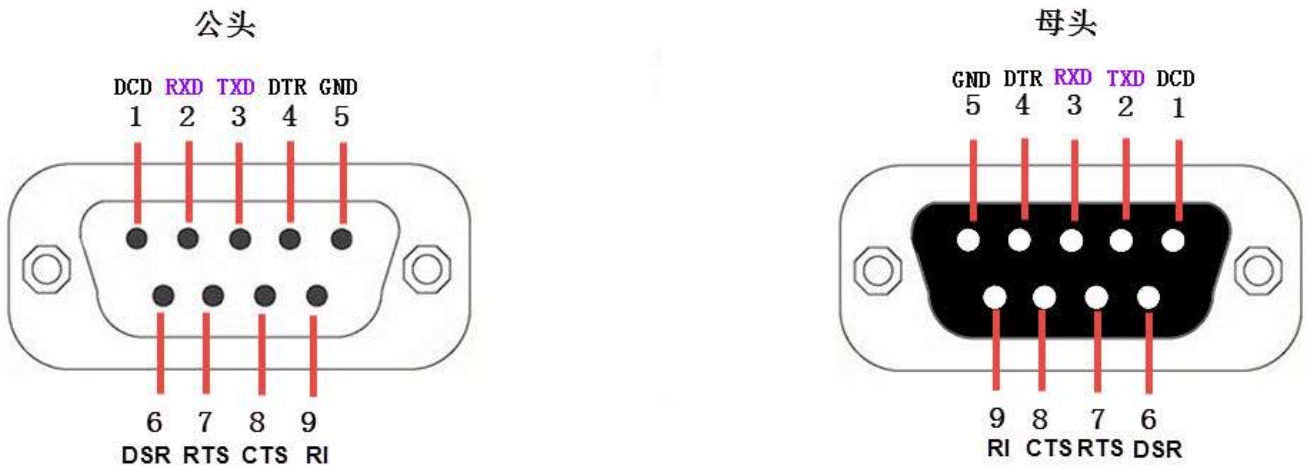


图 20-4 DB9 标准的公头及母头接法

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*。

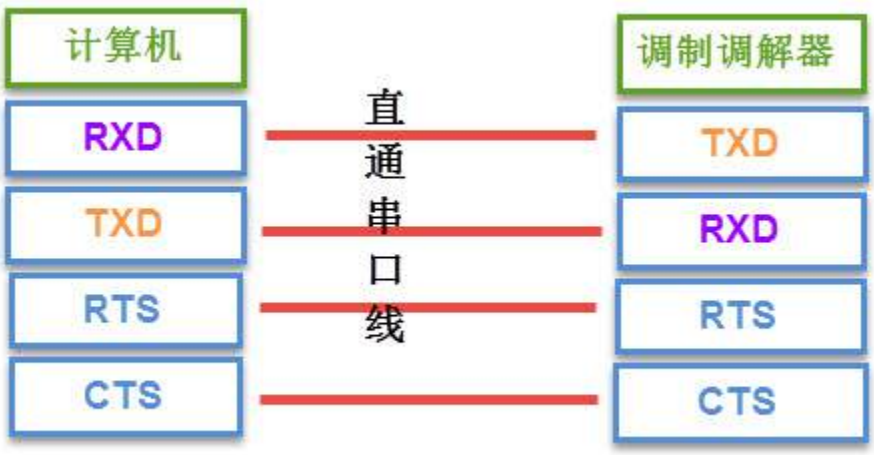


图 20-5 计算机与调制调解器的信号线连接

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。

20.1.2 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

20.1.2.1 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。

20.1.2.2 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。

20.1.2.3 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。

20.1.2.4 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)，它们介绍如下：

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”。

在无校验的情况下，数据包中不包含校验位。

20.2 STM32 的 USART 简介

STM32 芯片具有多个 USART 外设用于串口通讯，它是 Universal Synchronous Asynchronous Receiver and Transmitter 的缩写，即通用同步异步收发器可以灵活地与外部设备进行全双工数据交换。有别于 USART，它还有具有 UART 外设 (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。

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、指出运行出错位置等等。

STM32 的 USART 输出的是 TTL 电平信号，若需要 RS-232 标准的信号可使用 MAX3232 芯片进行转换。

20.3 USART 功能框图

STM32 的 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*。

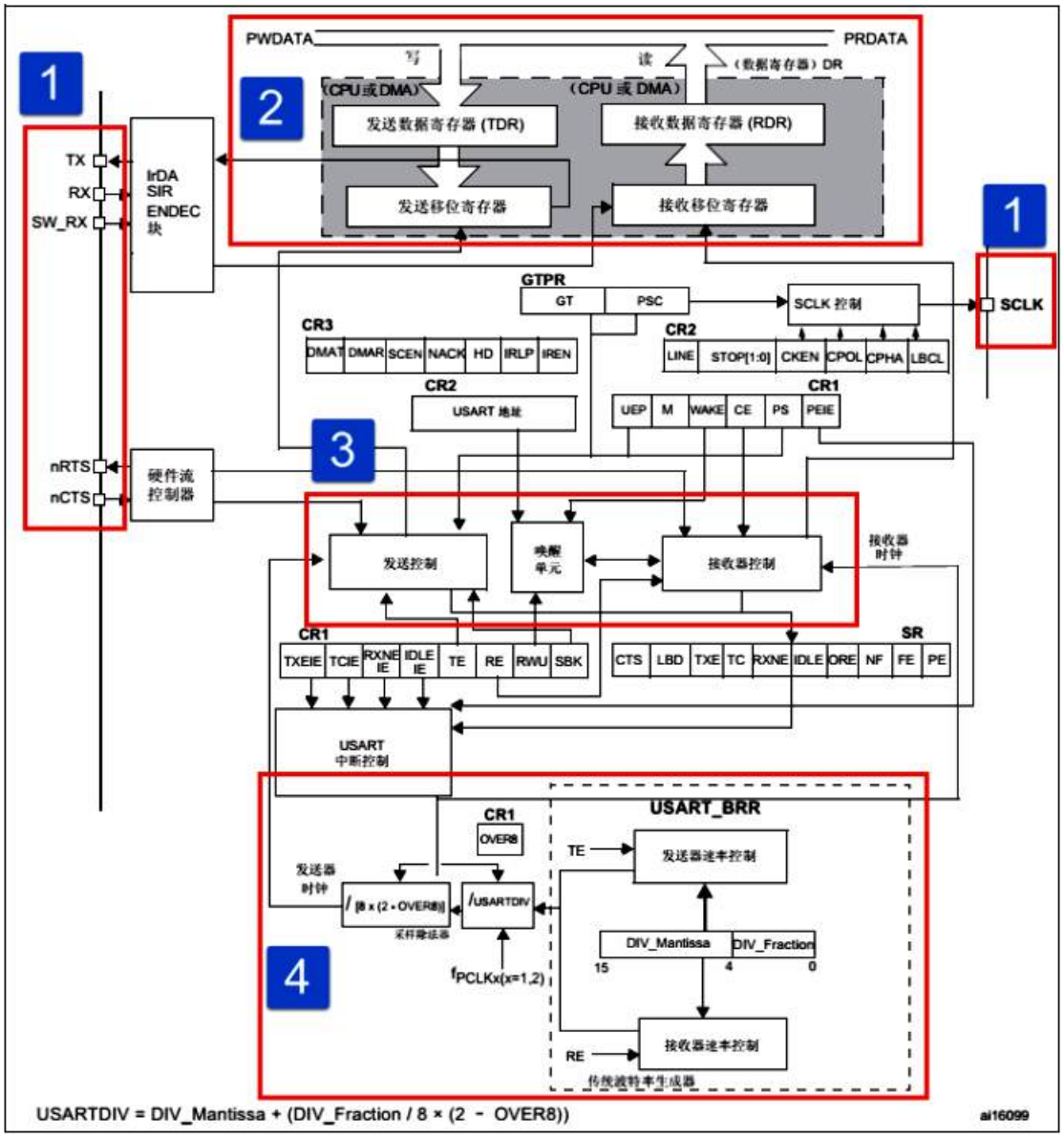


图 20-7 USART 功能框图

20.3.1 □ 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 STM32F407ZGT6 芯片具体发布见表 *20_3*。

表 20-3 STM32F407ZGT6 芯片的 USART 引脚

	APB2(最高 84MHz)		APB1(最高 42MHz)			
	USART1	USART6	USART2	USART3	UART4	UART5
TX	PA9/PB6	PC6/PG14	PA2/PD5	PB10/PD8 /PC10	PA0/PC10	PC12
RX	PA10/PB7	PC7/PG9	PA3/PD6	PB11/PD9 /PC11	PA1/PC11	PD2
SCLK	PA8	PG7/PC8	PA4/PD7	PB12/PD10 /PC12	-	-
nCTS	PA11	PG13/PG15	PA0/PD3	PB13/PD11	-	-
nRTS	PA12	PG8/PG12	PA1/PD4	PB14/PD12	-	-

STM32F407ZGT6 有四个 USART 和两个 UART，其中 USART1 和 USART6 的时钟来源于 APB2 总线时钟，其最大频率为 84MHz，其他四个的时钟来源于 APB1 总线时钟，其最大频率为 42MHz。

UART 只是异步传输功能，所以没有 SCLK、nCTS 和 nRTS 功能引脚。观察表 *20_3* 可发现很多 USART 的功能引脚有多个引脚可选，这非常方便硬件设计，只要在程序编程时软件绑定引脚即可。

20.3.2 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。

20.3.3 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*。

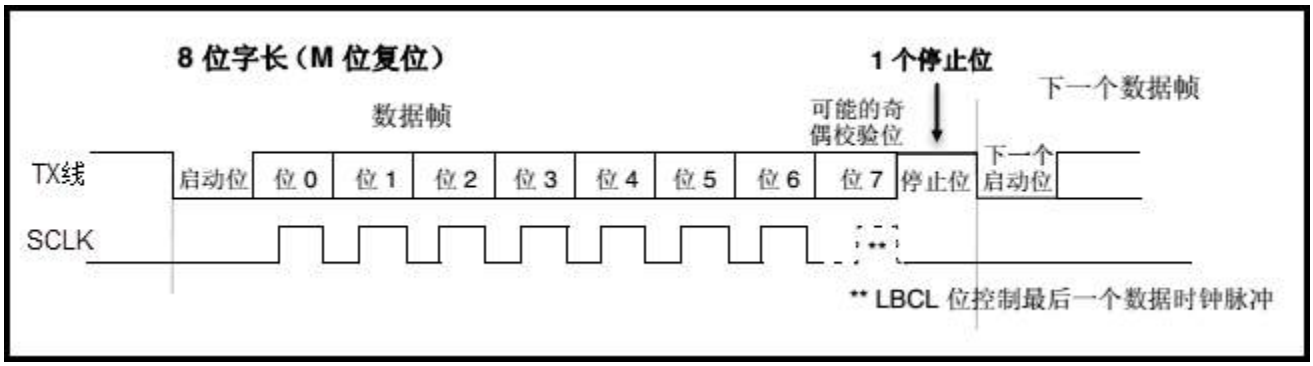


图 20-8 字符发送时序图

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。

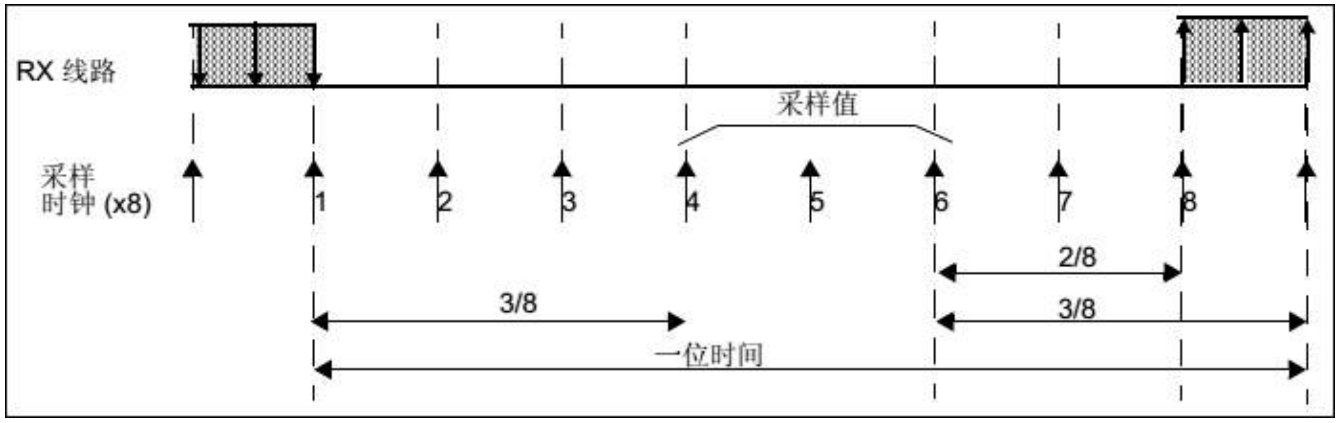
名称	描述
RE	接收使能
RXNE	读数据寄存器非空
RXNEIE	发送完成中断使能

为得到一个信号真实情况，需要用一个比这个信号频率高的采样信号去检测，称为过采样，这个采样信号的频率大小决定最后得到源信号准确度，一般频率越高得到的准确度越高，但为了得到越高频率采样信号越也困难，运算和功耗等等也会增加，所以一般选择合适就好。

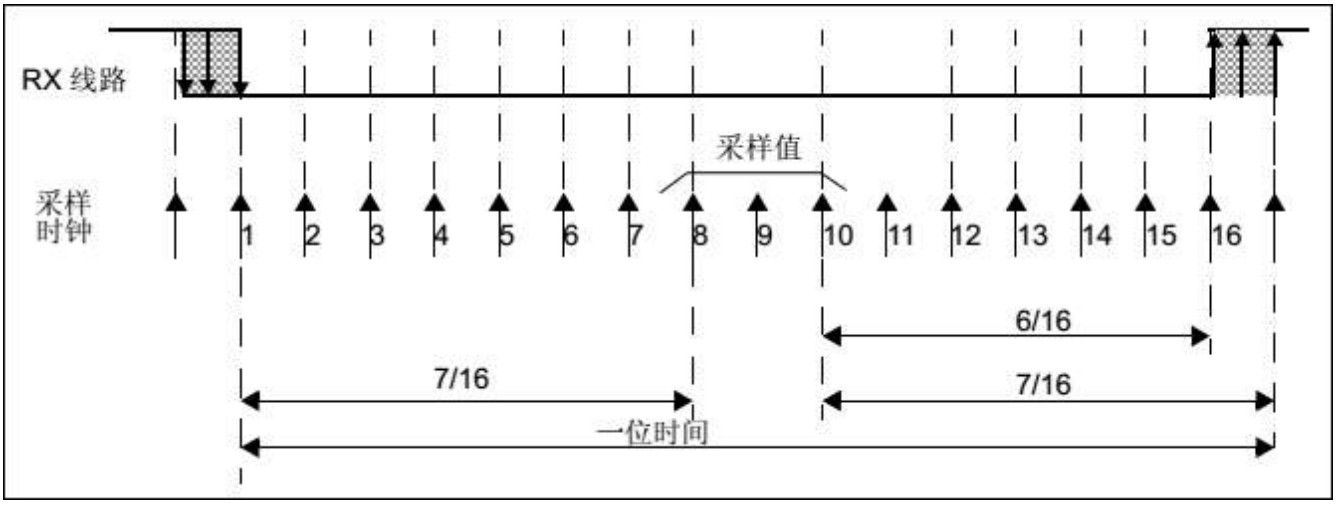
接收器可配置为不同过采样技术，以实现从噪声中提取有效的数据。USART_CR1 寄存器的 OVER8 位用来选择不同的采样采样方法，如果 OVER8 位设置为 1 采用 8 倍过采样，即用 8 个采样信号采样一位数据；如果 OVER8 位设置为 0 采用 16 倍过采样，即用 16 个采样信号采样一位数据。

USART 的起始位检测需要用到特定序列。如果在 RX 线识别到该特定序列就认为是检测到了起始位。起始位检测对使用 16 倍或 8 倍过采样的序列都是一样的。该特定序列为：1110X0X0X0000，其中 X 表示电平任意，1 或 0 皆可。

8 倍过采样速度更快，最高速度可达 fPCLK/8，fPCLK 为 USART 时钟，采样过程见图 *20_19*。使用第 4、5、6 次脉冲的值决定该位的电平状态。



16 倍过采样速度虽然没有 8 倍过采样那么快，但得到的数据更加精准，其最大速度为 fPCLK/16，采样过程见图 *20_20*。使用第 8、9、10 次脉冲的值决定该位的电平状态。



20.3.4 □ 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：

$$波特率=\frac{f_{PLCK}}{8\times(2-OVER8)\times USARTDIV}$$

公式 20-1 波特率计算

其中，fPLCK 为 USART 时钟，参考表 *20_3*；OVER8 为 USART_CR1 寄存器的 OVER8 位对应的值，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，DIV_Fraction[3] 位只有在 OVER8 位为 0 时有效，否则必须清零。

例如，如果 OVER8=0，DIV_Mantissa=24 且 DIV_Fraction=10，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 OVER8=0 并且知道 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，即位 0x1B。

如果 OVER8=1 情况类似，只是把计算用到的权值由 16 改为 8。

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。

由*表 20_3* 可知 USART1 和 USART6 使用 APB2 总线时钟，最高可达 84MHz，其他 USART 的最高频率为 42MHz。我们选取 USART1 作为实例讲解，即 fPLCK=84MHz。当我们使用 16 倍过采样时即 OVER8=0，为得到 115200bps 的波特率，此时：

$$115200 = \frac{84000000}{16 * \text{USARTDIV}}$$

解得 USARTDIV=45.57，可算得 DIV_Fraction=0x9（0.57*24=9.12 取整，在 BRR 寄存器中，表示小数位的有 4 位），DIV_Mantissa=0x2D，即应该设置 USART_BRR 的值为 0x2D9。

在计算 DIV_Fraction 时经常出现小数情况，经过我们取舍得到整数，这样会导致最终输出的波特率较目标值略有偏差。下面我们从 USART_BRR 的值为 0x2D9 开始计算得出实际输出的波特率大小。

由 USART_BRR 的值为 0x2D9，可得 DIV_Fraction=45，DIV_Mantissa=9，所以 USARTDIV=45+9/16=45.5625，所以实际波特率为：115226；这个值跟我们的目标波特率相差很小，这么小的误差在正常通信的允许范围内。

8 倍过采样时计算情况原理是一样的。

20.3.5 校验控制

STM32F4xx 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。

20.3.6 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

20.4 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 stm32f4xx_hal_usart.h 文件中，初始化库函数定义在 stm32f4xx_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。

20.4.1 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
    uint32_t CLKPolarity;
    uint32_t CLKPhase;
    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，见前面公式，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。

6)USART_LastBit：选择在发送最后一个数据位的时候时钟脉冲是否在 SCLK 引脚输出，可以是不输出脉冲 (USART_LASTBIT_DISABLE)、输出脉冲 (USART_LASTBIT_ENABLE)。它设定 USART_CR2 寄存器的 LBCL 位的值。

20.5 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。

20.5.1 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 IrDA 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 引脚通过跳帽 J11 与 USART1 的 RX 引脚连接，CH340G 的 RXD 引脚通过跳帽 J12 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。如果我们想使用串口 2，那么就可以把开发板中的默认连接的跳帽拔掉，然后用杜邦线连接 USART2 与 CH340G 的通信接口就可实现通信，这个就是用了跳帽的好处，不会把 CH340G 的通信引脚固定死。

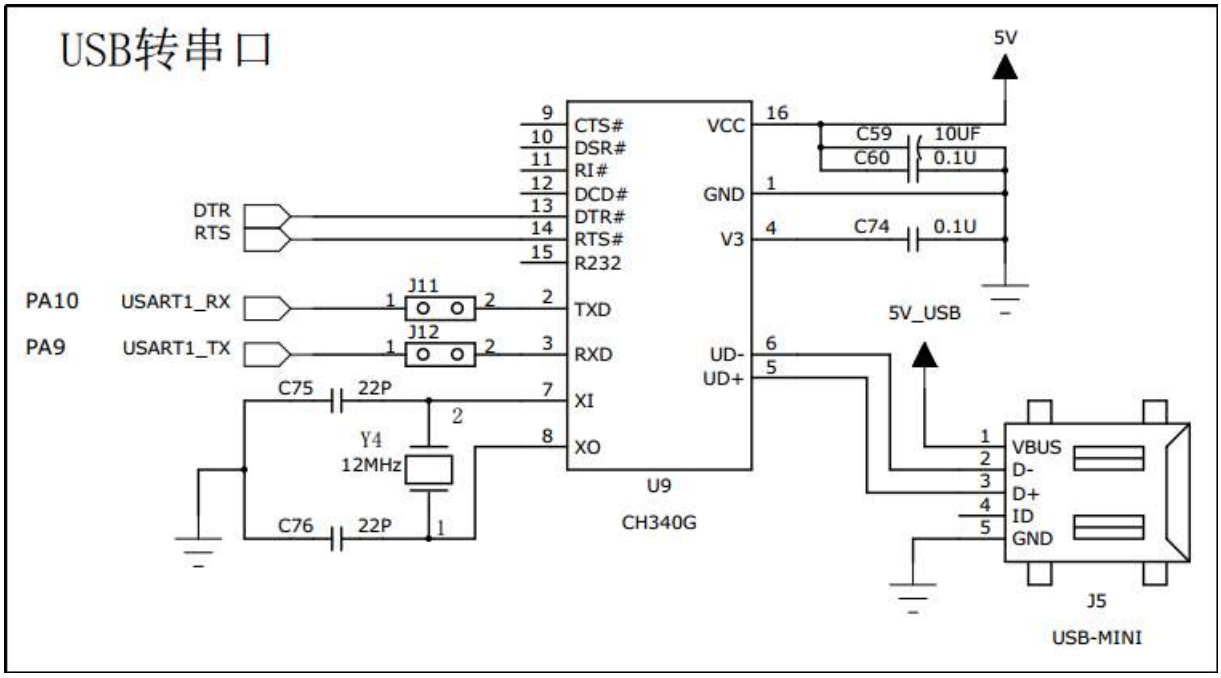


图 20-9 USB 转串口硬件设计

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。我们创建了两个文件：bsp_debug_usart.c 和 bsp_debug_usart.h 文件用来存放 USART 驱动程序及相关宏定义。

20.5.2.1 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。

20.5.2.2 代码分析

20.5.2.2.1 GPIO 和 USART 宏定义

列表 1: 代码清单 20-1 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
```

(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

/************************************************************/
```

使用宏定义方便程序移植和升级，根据图 *20_9* 电路，我们选择使用 USART1，设定波特率为 115200，一般我们会默认使用“8-N-1”参数，即 8 个数据位、不用校验、一位停止位。查阅表 *20_3* 可知 USART1 的 TX 线可对于 PA9 和 PB6 引脚，RX 线可对于 PA10 和 PB7 引脚，这里我们选择 PA9 以及 PA10 引脚。最后定义中断相关参数。

20.5.2.2.2 USART 初始化配置

列表 2: 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
```

(下页继续)

(续上页)

```
*/

void DEBUG_USART_Config(void)

{

    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 3: 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 4: 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

*/
```

(下页继续)

(续上页)

```
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

    */

    /* 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
```

(下页继续)

(续上页)

```
    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。

20.5.2.2.3 字符发送

列表 5: 代码清单 20-5 字符发送函数

```
/*************** 发送字符串 *\*********************/

void Usart_SendString(uint8_t *str)
```

(下页继续)

(续上页)

```
{

    unsigned int k=0;

    do {

        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。

20.5.2.2.4 USART 中断服务函数

列表 6: 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) {
```

(下页继续)

(续上页)

```
    ch=( uint16_t)READ_REG(UartHandle.Instance->DR);

    WRITE_REG ( UartHandle.Instance->DR,ch);

}

}
```

这段代码是存放在 stm32f4xx_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。

20.5.2.2.5 主函数

列表 7: 代码清单 20-7 主函数

```
int main(void)

{

    HAL_Init();

    /* 配置系统时钟 168MHz */

    SystemClock_Config();
```

(下页继续)

(续上页)

```
    /* 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。

20.5.3 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“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，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

20.6 USART1 指令控制 RGB 彩灯实验

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 RGB 彩灯的颜色。

20.6.1 硬件设计

硬件设计同第一个实验。

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。

20.6.2.1 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。

20.6.2.2 代码分析

20.6.2.2.1 GPIO 和 USART 宏定义

列表 8: 代码清单 20-8 GPIO 和 USART 宏定义

```
//串口波特率

#define DEBUG_USART_BAUDRATE 115200
```

(下页继续)

(续上页)

```
//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
```

(下页继续)

(续上页)

```
/**************************************************************/
```

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。

20.6.2.2.2 USART 初始化配置

列表 9: 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 */
```

(下页继续)

(续上页)

```
    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;
    UartHandle.Init.BaudRate = UARTx_BAUDRATE;
    UartHandle.Init.WordLength = UART_WORDLENGTH_8B;
    UartHandle.Init.StopBits = UART_STOPBITS_1;
    UartHandle.Init.Parity = UART_PARITY_NONE;
```

(下页继续)

(续上页)

```
    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，而恶是直接在 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。

20.6.2.2.3 重定向 prinft 和 scanf 函数

列表 10: 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);
```

(下页继续)

(续上页)

```
    return (ch);
}
//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。

20.6.2.2.4 输出提示信息

列表 11: 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);

    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");
```

(下页继续)

(续上页)

```
    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。

20.6.2.2.5 主函数

列表 12: 代码清单 20-12 主函数

```
int main(void)
{
    char ch;
    /* 配置系统时钟为 168 MHz */
    SystemClock_Config();

    /* 初始化 RGB 彩灯 */
    LED_GPIO_Config();

    /* 初始化 USART1 配置模式为 115200 8-N-1 */
    UARTx_Config();

    /* 打印指令输入提示信息 */
    Show_Message();
    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) {
```

(下页继续)

(续上页)

```
        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;
        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。

20.6.3 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 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，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 21 章　DMA—直接存储区访问

本章参考资料：《STM32F4xx 中文参考手册》DMA 控制器章节。

学习本章时，配合《STM32F4xx 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。本章内容专业名称较多，内容丰富也较难理解，但非常有必要细读研究。

特别说明，本章内容是以 STM32F4xx 系列资源讲解。

21.1 DMA 简介

DMA(Direct Memory Access, 直接存储区访问) 为实现数据高速在外设寄存器与存储器之间或者存储器与存储器之间传输提供了高效的方法。之所以称之为高效，是因为 DMA 传输实现高速数据移动过程无需任何 CPU 操作控制。从硬件层次上来说，DMA 控制器是独立于 Cortex-M4 内核的，有点类似 GPIO、USART 外设一般，只是 DMA 的功能是可以快速移动内存数据。

STM32F4xx 系列的 DMA 功能齐全，工作模式众多，适合不同编程环境要求。STM32F4xx 系列的 DMA 支持外设到存储器传输、存储器到外设传输和存储器到存储器传输三种传输模式。这里的外设一般指外设的数据寄存器，比如 ADC、SPI、I2C、DCMI 等等外设的数据寄存器，存储器一般是指片内 SRAM、外部存储器、片内 Flash 等等。

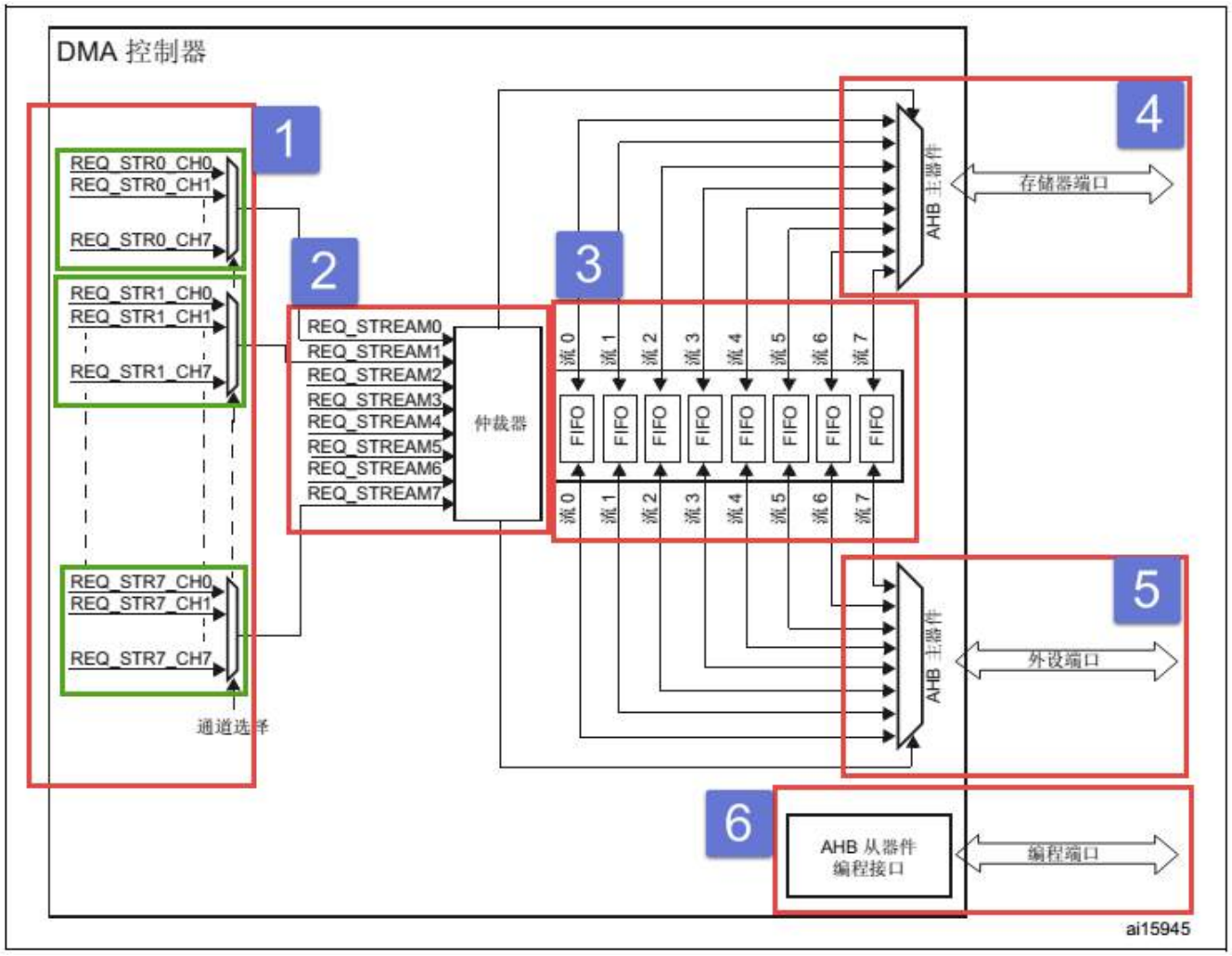
外设到存储器传输就是把外设数据寄存器内容转移到指定的内存空间。比如进行 ADC 采集时我们可以利用 DMA 传输把 AD 转换数据转移到我们定义的存储区中，这样对于多通道采集、采样频率高、连续输出数据的 AD 采集是非常高效的处理方法。

存储区到外设传输就是把特定存储区内容转移至外设的数据寄存器中，这种多用于外设的发送通信。

存储器到存储器传输就是把一个指定的存储区内容拷贝到另一个存储区空间。功能类似于 C 语言内存拷贝函数 memcpy，利用 DMA 传输可以达到更高的传输效率，特别是 DMA 传输是不占用 CPU 的，可以节省很多 CPU 资源。

21.2 DMA 功能框图

STM32F4xx 系列的 DMA 可以实现外设寄存器与存储器之间或者存储器与存储器之间传输三种模式，这要得益于 DMA 控制器是采样 AHB 主总线的，可以控制 AHB 总线矩阵来启动 AHB 事务。具体见图 *21_1*：DMA 控制器的框图。



21.2.1 外设通道选择

STM32F4xx 系列资源丰富，具有两个 DMA 控制器，同时外设繁多，为实现正常传输，DMA 需要通道选择控制。每个 DMA 控制器具有 8 个数据流，每个数据流对应 8 个外设请求。在实现 DMA 传输之前，DMA 控制器会通过 DMA 数据流 x 配置寄存器 DMA_SxCR(x 为 0~7，对应 8 个 DMA 数据流) 的 CHSEL[2:0] 位选择对应的通道作为该数据流的目标外设。

外设通道选择要解决的主要问题是决定哪一个外设作为该数据流的源地址或者目标地址。

DMA 请求映射情况参考表 21-1 和表 21-2。

表 21- 1 DMA1 请求映射

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
通道 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
通道 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
通道 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
通道 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
通道 5	UART8_TX	UART7_TX	TIM3_CH4 TIM3_UP	UART7_RX	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX	TIM3_CH3
通道 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
通道 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

图 21-2 DMA1 各个通道的请求映像

表 21- 2 DMA2 请求映射

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道 1		DCMI	ADC2	ADC2		SPI6_TX	SPI6_RX	DCMI
通道 2	ADC3	ADC3		SPI5_TX	SPI5_TX	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道 4	SPI4_RX	SPI4_TX	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道 5		USART6_RX	USART6_RX	SPI14_RX	SPI4_TX		USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_COM TIM1_TRIG	TIM1_UP	TIM1_CH3	
通道 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX	SPI5_TX	TIM8_CH4 TIM8_TRIG TIM8_COM

图 21-3 DMA2 各个通道的请求映像

每个外设请求都占用一个数据流通道，相同外设请求可以占用不同数据流通道。比如 SPI3_RX 请求，即 SPI3 数据发送请求，占用 DMA1 的数据流 0 的通道 0，因此当我们使用该请求时，我们需要在把 DMA_S0CR 寄存器的 CHSEL[2:0] 设置为 “000”，此时相同数据流的其他通道不被选择，处于不可用状态，比如此时不能使用数据流 0 的通道 1 即 I2C1_RX 请求等等。

查阅表 21-1 可以发现 SPI3_RX 请求不仅仅在数据流 0 的通道 0，同时数据流 2 的通道 0 也是 SPI3_RX 请求，实际上其他外设基本上都有两个对应数据流通道，这两个数据流通道都是可选

的，这样设计是尽可能提供多个数据流同时使用情况选择。

21.2.2 仲裁器

一个 DMA 控制器对应 8 个数据流，数据流包含要传输数据的源地址、目标地址、数据等等信息。如果我们需要同时使用同一个 DMA 控制器 (DMA1 或 DMA2) 多个外设请求时，那必然需要同时使用多个数据流，那究竟哪一个数据流具有优先传输的权利呢？这就需要仲裁器来管理判断了。

仲裁器管理数据流方法分为两个阶段。第一阶段属于软件阶段，我们在配置数据流时可以通过寄存器设定它的优先级别，具体配置 DMA_SxCR 寄存器 PL[1:0] 位，可以设置为非常高、高、中和低四个级别。第二阶段属于硬件阶段，如果两个或以上数据流软件设置优先级一样，则他们优先级取决于数据流编号，编号越低越具有优先权，比如数据流 2 优先级高于数据流 3。

21.2.3 FIFO

每个数据流都独立拥有四级 32 位 FIFO(先进先出存储器缓冲区)。DMA 传输具有 FIFO 模式和直接模式。

直接模式在每个外设请求都立即启动对存储器传输。在直接模式下，如果 DMA 配置为存储器到外设传输那 DMA 会见一个数据存放在 FIFO 内，如果外设启动 DMA 传输请求就可以马上将数据传输过去。

FIFO 用于在源数据传输到目标地址之前临时存放这些数据。可以通过 DMA 数据流 xFIFO 控制寄存器 DMA_SxFCR 的 FTH[1:0] 位来控制 FIFO 的阈值，分别为 1/4、1/2、3/4 和满。如果数据存储量达到阈值级别时，FIFO 内容将传输到目标中。

FIFO 对于要求源地址和目标地址数据宽度不同时非常有用，比如源数据是源源不断的字节数据，而目标地址要求输出字宽度的数据，即在实现数据传输时同时把原来 4 个 8 位字节的数据拼凑成一个 32 位字数据。此时使用 FIFO 功能先把数据缓存起来，分别根据需要输出数据。

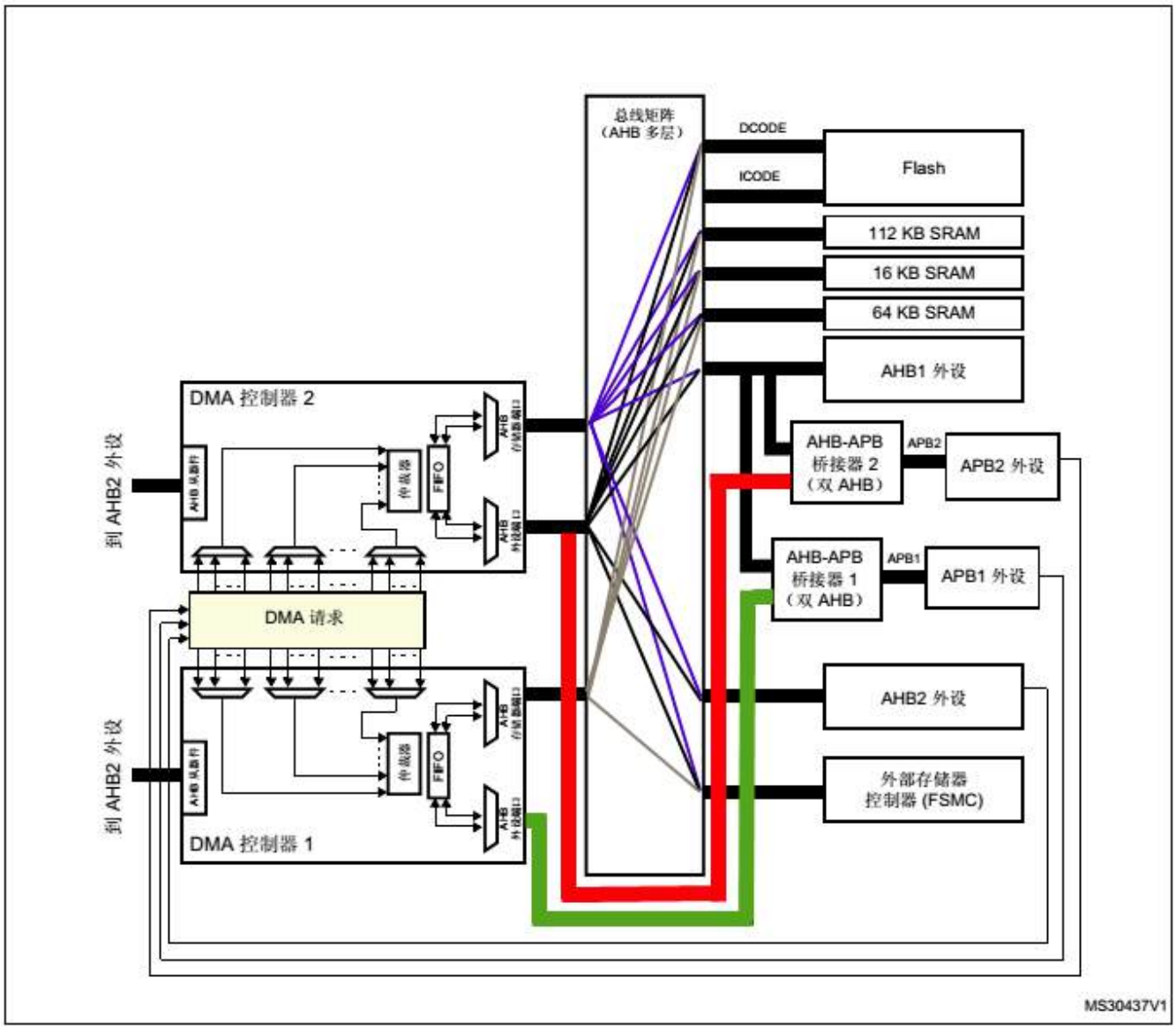
FIFO 另外一个作用使用于突发 (burst) 传输。

21.2.4 □ 存储器端口、□ 外设端口

DMA 控制器实现双 AHB 主接口，更好利用总线矩阵和并行传输。DMA 控制器通过存储器端口和外设端口与存储器和外设进行数据传输，关系见图 *21_4*。DMA 控制器的功能是快速转移内存数据，需要一个连接至源数据地址的端口和一个连接至目标地址的端口。

DMA2(DMA 控制器 2) 的存储器端口和外设端口都是连接到 AHB 总线矩阵，可以使用 AHB 总线矩阵功能。DMA2 存储器和外设端口可以访问相关的内存地址，包括有内部 Flash、内部 SRAM、AHB1 外设、AHB2 外设、APB2 外设和外部存储器空间。

DMA1 的存储区端口相比 DMA2 的要减少 AHB2 外设的访问权，同时 DMA1 外设端口是没有连接至总线矩阵的，只有连接到 APB1 外设，所以 DMA1 不能实现存储器到存储器传输。


总线矩阵
（AHB 多层）
DCODE
ICODE
Flash
112 KB SRAM
16 KB SRAM
64 KB SRAM
AHB1 外设
AHB-APB
桥接器 2
（双 AHB）
APB2
APB2 外设
AHB-APB
桥接器 1
（双 AHB）
APB1
APB1 外设
AHB2 外设
外部存储器
控制器 (FSMC)
DMA 控制器 2
DMA 请求
DMA 控制器 1
到 AHB2 外设
到 AHB2 外设
FIFO
MS30437V1

21.3 DMA 数据配置

DMA 工作模式多样，具有多种可能工作模式，具体可能配置见图 *21_5*。

DMA 传输模式	源	目标	流控制器	循环模式	传输类型	直接模式	双缓冲模式
外设到存储器	AHB 外设端口	AHB 存储器端口	DMA	允许	单次	允许	允许
					突发	禁止	
			外设	禁止	单次	允许	禁止
					突发	禁止	
存储器到外设	AHB 存储器端口	AHB 外设端口	DMA	允许	单次	允许	允许
					突发	禁止	
			外设	禁止	单次	允许	禁止
					突发	禁止	
存储器到存储器	AHB 存储器端口	AHB 存储器端口	仅 DMA	禁止	单次	禁止	禁止
					突发		

21.3.1 DMA 传输模式

DMA2 支持全部三种传输模式，而 DMA1 只有外设到存储器和存储器到外设两种模式。模式选择可以通过 DMA_SxCR 寄存器的 DIR[1:0] 位控制，进而将 DMA_SxCR 寄存器的 EN 位置 1 就可以使能 DMA 传输。

在 DMA_SxCR 寄存器的 PSIZE[1:0] 和 MSIZE[1:0] 位分别指定外设和存储器数据宽度大小，可以指定为字节 (8 位)、半字 (16 位) 和字 (32 位)，我们可以根据实际情况设置。直接模式要求外设和存储器数据宽度大小一样，实际上在这种模式下 DMA 数据流直接使用 PSIZE，MSIZE 不被使用。

21.3.2 源地址和目标地址

DMA 数据流 x 外设地址 DMA_SxPAR(x 为 0~7) 寄存器用来指定外设地址，它是一个 32 位数据有效寄存器。DMA 数据流 x 存储器 0 地址 DMA_SxM0AR(x 为 0~7) 寄存器和 DMA 数据流 x 存储器 1 地址 DMA_SxM1AR(x 为 0~7) 寄存器用来存放存储器地址，其中 DMA_SxM1AR 只用于双缓冲模式，DMA_SxM0AR 和 DMA_SxM1AR 都是 32 位数据有效的。

当选择外设到存储器模式时，即设置 DMA_SxCR 寄存器的 DIR[1:0] 位为“00”，DMA_SxPAR 寄存器为外设地址，也是传输的源地址，DMA_SxM0AR 寄存器为存储器地址，也是传输的目标地址。对于存储器到存储器传输模式，即设置 DIR[1:0] 位为“10”时，采用与外设到存储器模式相同配置。而对于存储器到外设，即设置 DIR[1:0] 位为“01”时，DMA_SxM0AR 寄存器作为为源地址，DMA_SxPAR 寄存器作为目标地址。

21.3.3 流控制器

流控制器主要涉及到一个控制 DMA 传输停止问题。DMA 传输在 DMA_SxCR 寄存器的 EN 位被置 1 后就进入准备传输状态，如果有外设请求 DMA 传输就可以进行数据传输。很多情况下，我们明确知道传输数据的数目，比如要传 1000 个或者 2000 个数据，这样我们就可以在传输之前设置 DMA_SxNDTR 寄存器为要传输数目值，DMA 控制器在传输完这么多数目数据后就可以控制 DMA 停止传输。

DMA 数据流 x 数据项数 DMA_SxNDTR(x 为 0~7) 寄存器用来记录当前仍需要传输数目，它是一个 16 位数据有效寄存器，即最大值为 65535，这个值在程序设计是非常有用也是需要注意的地方。我们在编程时一般都会明确指定一个传输数量，在完成一次数目传输后 DMA_SxNDTR 计数值就会自减，当达到零时就说明传输完成。

如果某些情况下在传输之前我们无法确定数据的数目，那 DMA 就无法自动控制传输停止了，此时需要外设通过硬件通信向 DMA 控制器发送停止传输信号。这里有一个大前提就是外设必须是可以发出这个停止传输信号，只有 SDIO 才有这个功能，其他外设不具备此功能。

21.3.4 循环模式

循环模式相对应于一次模式。一次模式就是传输一次就停止传输，下一次传输需要手动控制，而循环模式在传输一次后会自动按照相同配置重新传输，周而复始直至被控制停止或传输发生错误。

通过 DMA_SxCR 寄存器的 CIRC 位可以使能循环模式。

21.3.5 传输类型

DMA 传输类型有单次 (Single) 传输和突发 (Burst) 传输。突发传输就是用非常短时间结合非常高数据信号率传输数据，相对正常传输速度，突发传输就是在传输阶段把速度瞬间提高，实现高速传输，在数据传输完成后恢复正常速度，有点类似达到数据块“秒传”效果。为达到这个效果突发传输过程要占用 AHB 总线，保证要求每个数据项在传输过程不被分割，这样一次性把数据全部传输完才释放 AHB 总线；而单次传输时必须通过 AHB 的总线仲裁多次控制才传输完成。

单次和突发传输数据使用具体情况参考表 21 4。其中 PBURST[1:0] 和 MBURST[1:0] 位是位于 DMA_SxCR 寄存器中的，用于分别设置外设和存储器不同节拍数的突发传输，对应为单次传输、4 个节拍增量传输、8 个节拍增量传输和 16 个节拍增量传输。PINC 位和 MINC 位是寄存器 DMA_SxCR 寄存器的第 9 和第 10 位，如果位被置 1 则在每次数据传输后数据地址指针自动递增，其增量由 PSIZE 和 MSIZE 值决定，比如，设置 PSIZE 为半字大小，那么下一次传输地址将是前一次地址递增 2。

AHB 主端口	项目	单次传输	突发传输
外设	寄存器	PBURST[1:0]=00，PINC 无要求	PBURST[1:0]不为 0，PINC 必须为 1
	描述	每次 DMA 请求就传输一次字节/半字/字(取决于 PSIZE)数据	每次 DMA 请求就传输 4/8/16 个(取决于 PBURST[1:0])字节/半字/字(取决于 PSIZE)数据
存储器	寄存器	MBURST[1:0]=00，MINC 无要求	MBURST[1:0]不为 0，MINC 必须为 1
	描述	每次 DMA 请求就传输一次字节/半字/字(取决于 MSIZE)数据	每次 DMA 请求就传输 4/8/16 个(取决于 MBURST[1:0])字节/半字/字(取决于 MSIZE)数据

突发传输与 FIFO 密切相关，突发传输需要结合 FIFO 使用，具体要求 FIFO 阈值一定要是内存突发传输数据量的整数倍。FIFO 阈值选择和存储器突发大小必须配合使用，具体参考上表图 *21_6*。

MSIZE	FIFO 级别	MBURST=INCR4	MBURST=INCR8	MBURST=INCR16
字节	1/4	4 个节拍的 1 次突发	禁止	禁止
	1/2	4 个节拍的 2 次突发	8 个节拍的 1 次突发	
	3/4	4 个节拍的 3 次突发	禁止	
	满	4 个节拍的 4 次突发	8 个节拍的 2 次突发	16 个节拍的 1 次突发
半字	1/4	禁止	禁止	禁止
	1/2	4 个节拍的 1 次突发		
	3/4	禁止		
	满	4 个节拍的 2 次突发	8 个节拍的 1 次突发	
字	1/4	禁止	禁止	
	1/2			
	3/4			
	满	4 个节拍的 1 次突发		

21.3.6 直接模式

默认情况下，DMA 工作在直接模式，不使能 FIFO 阈值级别。

直接模式在每个外设请求都立即启动对存储器传输的单次传输。直接模式要求源地址和目标地址的数据宽度必须一致，所以只有 PSIZE 控制，而 MSIZE 值被忽略。突发传输是基于 FIFO 的所以直接模式不被支持。另外直接模式不能用于存储器到存储器传输。

在直接模式下，如果 DMA 配置为存储器到外设传输那 DMA 会见一个数据存放在 FIFO 内，如果外设启动 DMA 传输请求就可以马上将数据传输过去。

21.3.7 双缓冲模式

设置 DMA_SxCR 寄存器的 DBM 位为 1 可启动双缓冲传输模式，并自动激活循环模式。双缓冲不应用与存储器到存储器的传输。双缓冲模式下，两个存储器地址指针都有效，即 DMA_SxM1AR 寄存器将被激活使用。开始传输使用 DMA_SxM0AR 寄存器的地址指针所对应的存储区，当这个存储区数据传输完 DMA 控制器会自动切换至 DMA_SxM1AR 寄存器的地址指针所对应的另一块存储区，如果这一块也传输完成就再切换至 DMA_SxM0AR 寄存器的地址指针所对应的存储区，这样循环调用。

当其中一个存储区传输完成时都会把传输完成中断标志 TCIF 位置 1，如果我们使能了 DMA_SxCR 寄存器的传输完成中断，则可以产生中断信号，这个对我们编程非常有用。另外一个非常有用的信息是 DMA_SxCR 寄存器的 CT 位，当 DMA 控制器是在访问使用 DMA_SxM0AR 时 CT=0，此时 CPU 不能访问 DMA_SxM0AR，但可以向 DMA_SxM1AR 填充或者读取数据；当 DMA 控制器是在访问使用 DMA_SxM1AR 时 CT=1，此时 CPU 不能访问 DMA_SxM1AR，但可以向 DMA_SxM0AR 填充或者读取数据。另外在未使能 DMA 数据流传输时，可以直接写 CT 位，改变开始传输的目标存储区。

双缓冲模式应用在需要解码程序的地方是非常有效的。比如 MP3 格式音频解码播放，MP3 是被压缩的文件格式，我们需要特定的解码库程序来解码文件才能得到可以播放的 PCM 信号，解码需要一定的实际，按照常规方法是读取一段原始数据到缓冲区，然后对缓冲区内容进行解码，解码后才输出到音频播放电路，这种流程对 CPU 运算速度要求高，很容易出现播放不流畅现象。如果我们使用 DMA 双缓冲模式传输数据就可以非常好的解决这个问题，达到解码和输出音频数据到音频电路同步进行的效果。

21.3.8 DMA 中断

每个 DMA 数据流可以在发送以下事件时产生中断：

1) 达到半传输：DMA 数据传输达到一半时 HTIF 标志位被置 1，如果使能 HTIE 中断控制位将产生达到半传输中断；

2) 传输完成：DMA 数据传输完成时 TCIF 标志位被置 1，如果使能 TCIE 中断控制位将产生传输完成中断；

3) 传输错误：DMA 访问总线发生错误或者在双缓冲模式下试图访问“受限”存储器地址寄存器时 TEIF 标志位被置 1，如果使能 TEIE 中断控制位将产生传输错误中断；

4)FIFO 错误：发生 FIFO 下溢或者上溢时 FEIF 标志位被置 1，如果使能 FEIE 中断控制位将产生 FIFO 错误中断；

5) 直接模式错误：在外设到存储器的直接模式下，因为存储器总线没得到授权，使得先前数据没有完成被传输到存储器空间上，此时 DMEIF 标志位被置 1，如果使能 DMEIE 中断控制位将产生直接模式错误中断。

21.4 DMA 初始化结构体详解

HAL 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 stm32f4xx_hal_xxx.h(后面 xxx 为外设名称) 文件中，库函数 xxx_Init 定义在 stm32f4xx_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。

21.4.1 DMA_InitTypeDef 初始化结构体

```
typedef struct {
    uint32_t Channel;                //通道选择
    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
    uint32_t FIFOMode;               //FIFO 模式
    uint32_t FIFOThreshold;          //FIFO 阈值
    uint32_t MemBurst;               //存储器突发传输
    uint32_t PeriphBurst;            //外设突发传输
 } DMA_InitTypeDef;
```

1)Channel：DMA 请求通道选择，可选通道 0 至通道 7，每个外设对应固定的通道，具体设置值需要查表图 *21_2* 和表图 *21_3*；它设定 DMA_SxCR 寄存器的 CHSEL[2:0] 位的值。例如，我们使用模拟数字转换器 ADC3 规则采集 4 个输入通道的电压数据，查表图 *21_3* 可知使用通道 2。

2)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。

3)PeriphInc：如果配置为 PeriphInc_Enable，使能外设地址自动递增功能，它设定 DMA_SxCR 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。ADC3 的数据寄存器地址是固定并且只有一个所以不使能外设地址递增。

4)MemInc：如果配置为 MemInc_Enable，使能存储器地址自动递增功能，它设定 DMA_SxCR 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，所以使能存储器地址自动递增功能。我们之前已经定义了一个包含 4 个元素的数字用来存放数据，使能存储区地址递增功能，自动把每个通道数据存放到对应数组元素内。

5)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。

6)MemDataAlignment：存储器数据宽度，可选字节 (8 位)、半字 (16 位) 和字 (32 位)，它设定 DMA_SxCR 寄存器的 MSIZE[1:0] 位的值。保存 ADC 转换数据也要使用半字数据宽度，这跟我们定义的数组是相对应的。

7)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。

8)Priority：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。

9)FIFOMode：FIFO 模式使能，如果设置为 DMA_FIFOMode_Enable 表示使能 FIFO 模式功能；它设定 DMA_SxFCR 寄存器的 DMDIS 位。ADC 采集传输使用直接传输模式即可，不需要使用 FIFO 模式。

10)FIFOThreshold：FIFO 阈值选择，可选 4 种状态分别为 FIFO 容量的 1/4、1/2、3/4 和满；它设定 DMA_SxFCR 寄存器的 FTH[1:0] 位；DMA_FIFOMode 设置为 DMA_FIFOMode_Disable，那 DMA_FIFOThreshold 值无效。ADC 采集传输不使用 FIFO 模式，设置改值无效。

11)MemBurst：存储器突发模式选择，可选单次模式、4 节拍的增量突发模式、8 节拍的增量突发模式或 16 节拍的增量突发模式，它设定 DMA_SxCR 寄存器的 MBURST[1:0] 位的值。ADC 采集传输是直接模式，要求使用单次模式。

12)PeriphBurst：外设突发模式选择，可选单次模式、4 节拍的增量突发模式、8 节拍的增量突发模式或 16 节拍的增量突发模式，它设定 DMA_SxCR 寄存器的 PBURST[1:0] 位的值。ADC 采集传输是直接模式，要求使用单次模式。

21.4.2 DMA_HandleTypeDef 初始化结构体

```
typedef struct __DMA_HandleTypeDef {
    DMA_Stream_TypeDef          *Instance;    //注册基地址
    DMA_InitTypeDef             Init;         //DMA 通信参数
    HAL_LockTypeDef             Lock;         //DMA 锁定对象
```

(下页继续)

(续上页)

```
    __IO HAL_DMA_StateTypeDef  State;        //DMA 传输状态
    void                       *Parent;      //父类指针
    void (* XferCpltCallback)( struct __DMA_HandleTypeDef * hdma);
    //DMA 传输完成回调函数
    void (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
    //DMA 传输完成一半回调函数
    void (* XferM1CpltCallback)( struct __DMA_HandleTypeDef * hdma);
    //Memory1 DMA 传输完成回调函数
    void (* XferM1HalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
    //Memory1 DMA 传输完成一半回调函数
    void (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
    //DMA 传输错误回调函数
    void (* XferAbortCallback)( struct __DMA_HandleTypeDef * hdma);
    //DMA 传输中止回调函数
    __IO uint32_t              ErrorCode;          //DMA 错误码
    uint32_t                   StreamBaseAddress;  //DMA 数据流基地址
    uint32_t                   StreamIndex;            //DMA 数据流索引
} DMA_HandleTypeDef;
```

1)*Instance： 指向 DMA 数据流基地址的指针，即指定使用哪个 DMA 数据流。可选数据流 0 至数据流 7。例如，我们使用模拟数字转换器 ADC3 规则采集 4 个输入通道的电压数据，查表图 *21_3* 可知可以使用数据流 0 或者数据流 1，这里支持两个数据流是为了避免多个通道使用时发生冲突，提供备选数据流可选。

2)Init：这里包含上面介绍 DMA_InitTypeDef 结构体的所有参数的初始化。

3)Lock：DMA 锁定对象。DMA 进程锁，通常都在 DMA 传输设置开始前锁上进程锁，设置完毕后释放进程锁。

4)State：DMA 传输状态。它包含六种状态，1、复位状态，尚未初始化或者禁能。2、就绪状态，已经完成初始化，随时可以传输数据。3、传输忙，DMA 传输进程正在进行。4、传输超时状态。5、传输错误状态。6、传输中止状态。

5)*Parent：父类指针。只要将该指针指向一些 ADC、UART 等外设的 handle 类，就等于完成了继

承。

6)DMA 传输过程中的回调函数。包括传输完成，传输完成一半，传输错误，传输中止回调函数。这些回调函数中可以加入用户的处理代码。

7)ErrorCode：DMA 错误码，包含无错误：HAL_DMA_ERROR_NONE，传输错误 HAL_DMA_ERROR_TE，FIFO 错误 HAL_DMA_ERROR_FE，直接模式错误：HAL_DMA_ERROR_DME，超时错误：HAL_DMA_ERROR_TIMEOUT，参数错误：HAL_DMA_ERROR_PARAM，没有回调函数正在执行退出请求错误：HAL_DMA_ERROR_NO_XFER，不支持模式错误：HAL_DMA_ERROR_NOT_SUPPORTED。

8)StreamBaseAddress：DMA 数据流基地址，用来根据定义句柄计算数据流的基地址。

9)StreamIndex：DMA 数据流索引，根据数据流的序号来确定数据流的偏移地址。

21.5 DMA 存储器到存储器模式实验

DMA 工作模式多样，具体如何使用需要配合实际传输条件具体分析。接下来我们通过两个实验详细讲解 DMA 不同模式下的使用配置，加深我们对 DMA 功能的理解。

DMA 运行高效，使用方便，在很多测试实验都会用到，这里先详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会有很多使用到的情况，也会有相关的分析。

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，然后使用 DMA 传输把源数据拷贝到目标地址上，最后对比源数据和目标地址的数据，看看是否传输准确。

21.5.1 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，关于 RGB 彩色灯电路可以参考 GPIO 章节。

21.5.2 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。

21.5.2.1 编程要点

1) 使能 DMA 数据流时钟并复位初始化 DMA 数据流；
2) 配置 DMA 数据参数；
3) 使能 DMA 数据流，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。

21.5.2.2 代码分析

21.5.2.2.1 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]= {
```

(下页继续)

(续上页)

```
    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必须使用 DMA2，但对数据流编号以及通道选择就没有硬性要求，可以自由选择。

aSRC_Const_Buffer[BUFFER_SIZE] 是定义用来存放源数据的，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。

21.5.2.2.2 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;
    /* 使能 DMA 时钟 */
    DMA_STREAM_CLOCK();

    DMA_Handle.Instance = DMA_STREAM;
    /* DMA 数据流通道选择 */
```

(下页继续)

(续上页)

```
    DMA_Handle.Init .Channel = DMA_CHANNEL;
    /* 存储器到存储器模式 */
    DMA_Handle.Init.Direction = DMA_MEMORY_TO_MEMORY;
    /* 使能自动递增功能 */
    DMA_Handle.Init.PeriphInc = DMA_PINC_ENABLE;
    /* 使能自动递增功能 */
    DMA_Handle.Init.MemInc = DMA_MINC_ENABLE;
    /* 源数据是字大小 (32 位) */
    DMA_Handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
    /* 目标数据也是字大小 (32 位) */
    DMA_Handle.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
    /* 一次传输模式, 存储器到存储器模式不能使用循环传输 */
    DMA_Handle.Init.Mode = DMA_NORMAL;
    /* DMA 数据流优先级为高 */
    DMA_Handle.Init.Priority = DMA_PRIORITY_HIGH;
    /* 禁用 FIFO 模式 */
    DMA_Handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
    DMA_Handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
    /* 单次模式 */
    DMA_Handle.Init.MemBurst = DMA_MBURST_SINGLE;
    /* 单次模式 */
    DMA_Handle.Init.PeriphBurst = DMA_PBURST_SINGLE;
    /* 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 */
        while (1) {
            LED_RED;
```

(下页继续)

(续上页)

```
            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。

21.5.2.2.3 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 */
        if (*pBuffer != *pBuffer1) {
            /* 对应数据源不相等马上退出函数，并返回 0 */
```

(下页继续)

(续上页)

```
            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。

21.5.2.2.4 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 168 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 */
    Delay(0xFFFFFF);
```

(下页继续)

(续上页)

```
    /* DMA 传输配置 */
    DMA_Config();

    /* 等待 DMA 传输完成 */
    while (__HAL_DMA_GET_FLAG(&DMA_Handle,DMA_FLAG_TCIF0_4)==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。

21.5.2.3 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。

21.6 DMA 存储器到外设模式实验

DMA 存储器到外设传输模式非常方便把存储器数据传输外设数据寄存器中，这在 STM32 芯片向其他目标主机，比如电脑、另外一块开发板或者功能芯片，发送数据是非常有用的。RS-232 串口通信是我们常用开发板与 PC 端通信的方法。我们可以使用 DMA 传输把指定的存储器数据转移到 USART 数据寄存器内，并发送至 PC 端，在串口调试助手显示。

21.6.1 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。

21.6.2 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。

21.6.2.1 编程要点

1) 配置 USART 通信功能；
2) 设置 DMA 为存储器到外设模式，设置数据流通道，指定 USART 数据寄存器为目标地址，循环发送模式；
3) 使能 DMA 数据流；
4) 使能 USART 的 DMA 发送请求。
5) DMA 传输同时 CPU 可以运行其他任务。

21.6.2.2 代码分析

21.6.2.2.1 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
```

(下页继续)

(续上页)

```
#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

#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
```

(下页继续)

(续上页)

```
#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 USART1 对应 DMA2 的数据流 7 通道 4。

21.6.2.2.2 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)
{
  /* 开启 DMA 时钟 */
  DEBUG_USART_DMA_CLK_ENABLE();

  DMA_Handle.Instance = DEBUG_USART_DMA_STREAM;
  /*usart1 tx 对应 dma2, 通道 4, 数据流 7*/
  DMA_Handle.Init.Channel = DEBUG_USART_DMA_CHANNEL;
  /* 方向：从内存到外设 */
  DMA_Handle.Init.Direction= DMA_MEMORY_TO_PERIPH;
  /* 外设地址不增 */
  DMA_Handle.Init.PeriphInc = DMA_PINC_DISABLE;
  /* 内存地址自增 */
  DMA_Handle.Init.MemInc = DMA_MINC_ENABLE;
  /* 外设数据单位 */
  DMA_Handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
  /* 内存数据单位 8bit*/
  DMA_Handle.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
  /*DMA 模式：不断循环 */
```

(下页继续)

(续上页)

```
  DMA_Handle.Init.Mode = DMA_CIRCULAR;
  /* 优先级：中 */
  DMA_Handle.Init.Priority = DMA_PRIORITY_MEDIUM;
  /* 禁用 FIFO*/
  DMA_Handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
  DMA_Handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
  /* 存储器突发传输 16 个节拍 */
  DMA_Handle.Init.MemBurst = DMA_MBURST_SINGLE;
  /* 外设突发传输 1 个节拍 */
  DMA_Handle.Init.PeriphBurst = DMA_PBURST_SINGLE;
  /* 配置 DMA2 的数据流 7*/
  /* Deinitialize the stream for new transfer */
  HAL_DMA_DeInit(&DMA_Handle);
  /* Configure the DMA stream */
  HAL_DMA_Init(&DMA_Handle);

  /* Associate the DMA handle */
  __HAL_LINKDMA(&UartHandle, hdmatx, DMA_Handle);
}
```

使用 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 DEBUG_USART_DMA_CLK_ENABLE 宏开启 DMA 数据流时钟，使用 DMA 控制器之前必须开启对应的时钟。

USART 有固定的 DMA 通道，USART 数据寄存器地址也是固定的，外设地址不可以使用自动递增，源数据使用我们自定义的数组空间，存储器地址使用自动递增，采用循环发送模式，最后我调用 HAL_DMA_DeInit 函数复位到缺省配置状态，DMA_Init 函数完成 DMA 数据流的初始化配置。

__HAL_LINKDMA 函数用于链接 DMA 数据流及通道到串口外设通道上。

21.6.2.2.3 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;

    /* 系统时钟初始化成 168 MHz */
    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脑端串口调试助手会卡死，鼠标乱飞的情况，
    * 或把 DMA 配置中的循环模式改为单次模式 */

    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 */
```

(下页继续)

(续上页)

```
    //例如同时控制 LED
    while (1) {
        LED1_TOGGLE
        Delay(0xFFFFFF);
    }
}
```

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。

21.6.2.3 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“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 22 章　常用存储器介绍

22.1 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。

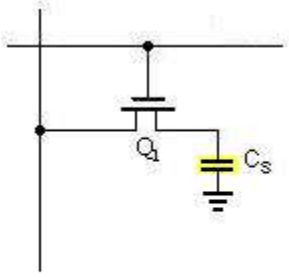
22.2 RAM 存储器

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。

22.2.1 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

22.2.1.1 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。

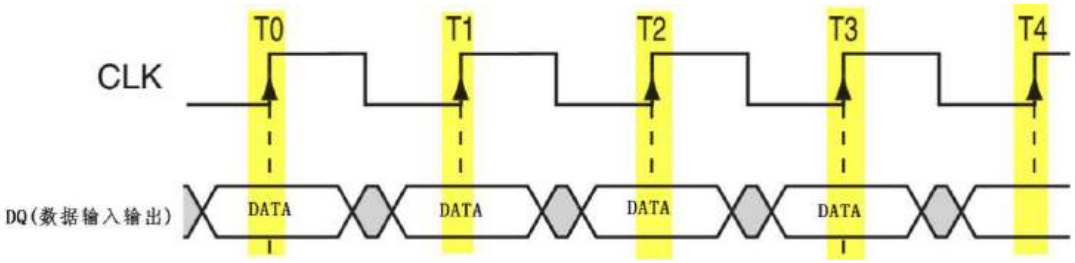


图 22-3 同步通讯时序图

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)。

22.2.1.2 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。

22.2.2 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。

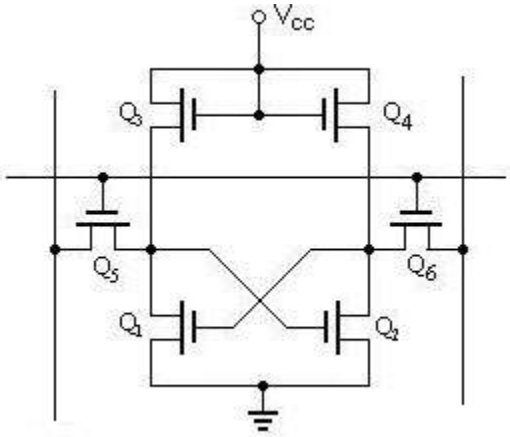


图 22-4 SRAM 存储单元

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。

22.2.3 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 STM32F407 等型号只能扩展 SRAM。

22.3 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。

22.3.1 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。

22.3.1.1 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。

22.3.1.2 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。

22.3.1.3 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。

22.3.1.4 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。

22.3.2 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 23 章　I2C—读写 EEPROM

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》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。若想了解 SMBUS，可阅读《smbus20》文档。

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。

23.1 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。

下面我们分别对 I2C 协议的物理层及协议层进行讲解。

23.1.1 I2C 物理层

I2C 通讯设备之间的常用连接方式见图 *23_1*。

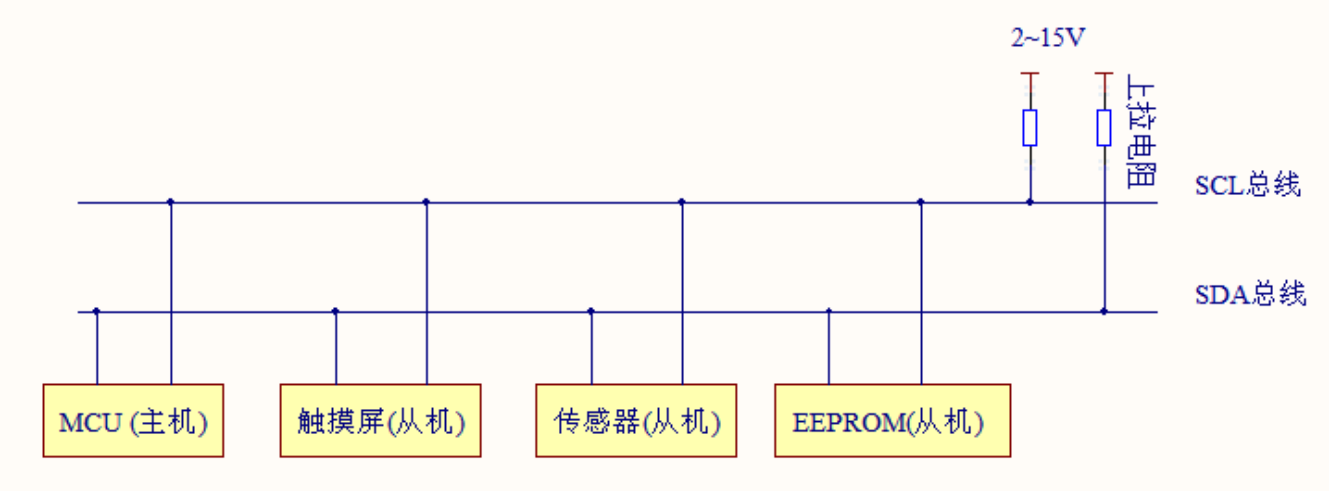


图 23-1 常见的 I2C 通讯系统

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。

23.1.2 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。

23.1.2.1 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*。

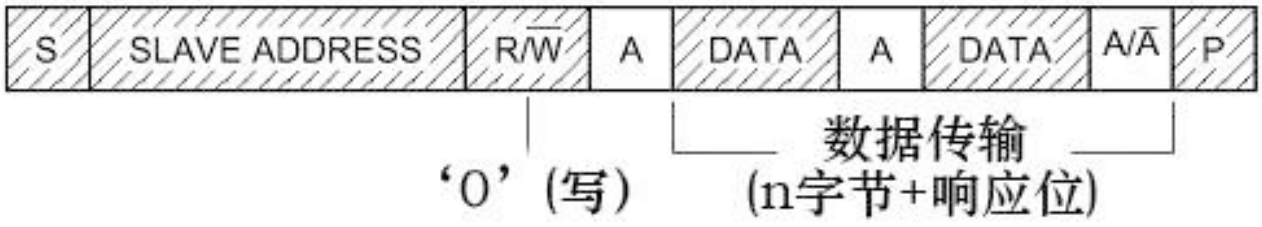


图 23-2 主机写数据到从机



图 23-3 主机由从机中读数据

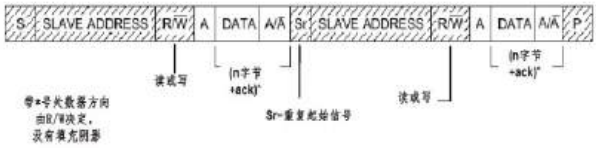


图 23-4 I2C 通讯复合格式

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：

23.1.2.2 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。

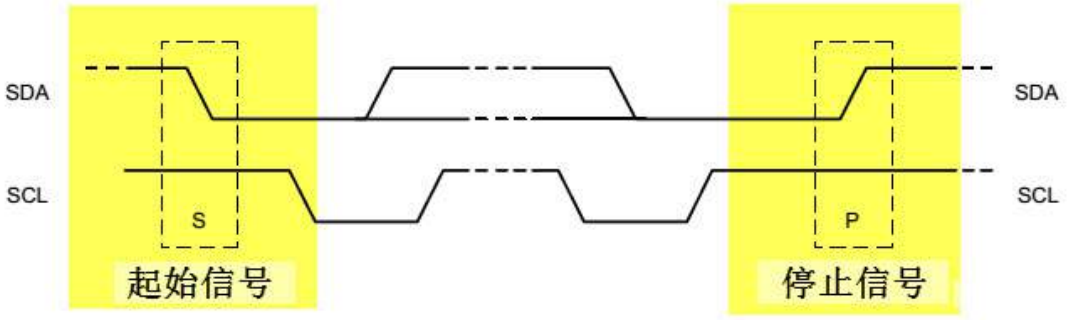


图 23-5 起始和停止信号

23.1.2.3 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。

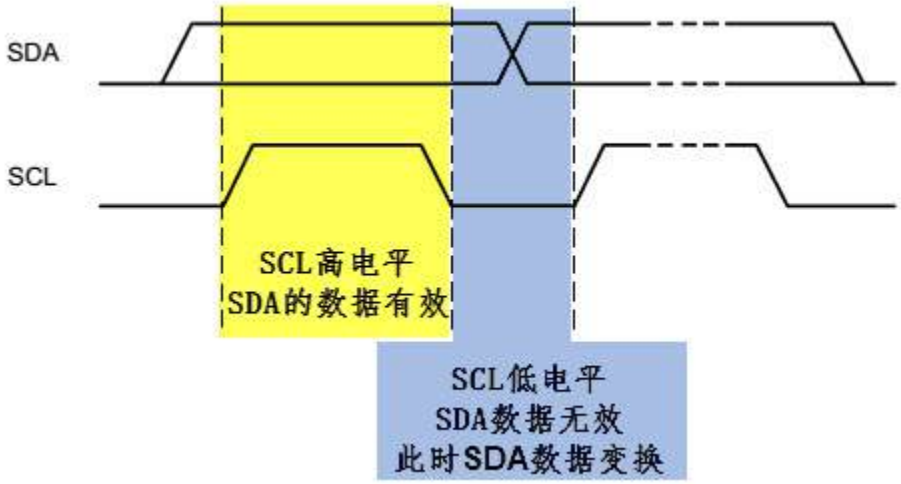


图 23-6 数据有效性

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。

23.1.2.4 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*。

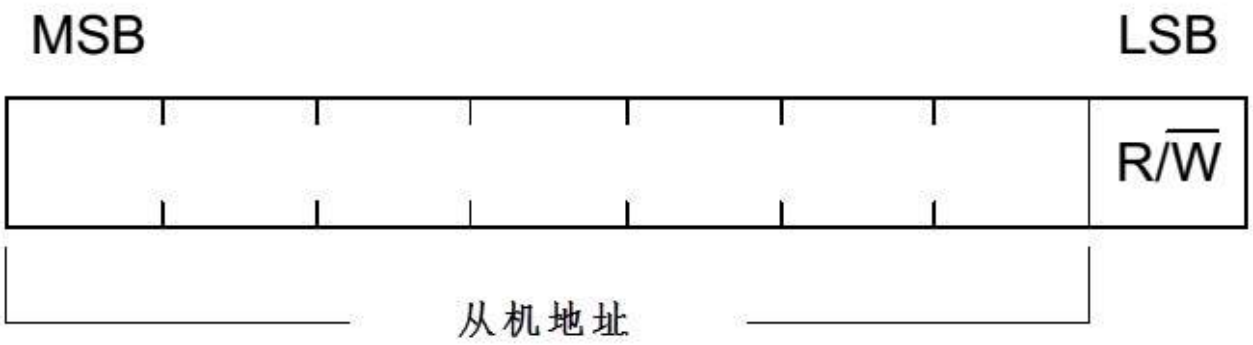


图 23-7 设备地址 (7 位) 及数据传输方向

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。

23.1.2.5 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*。

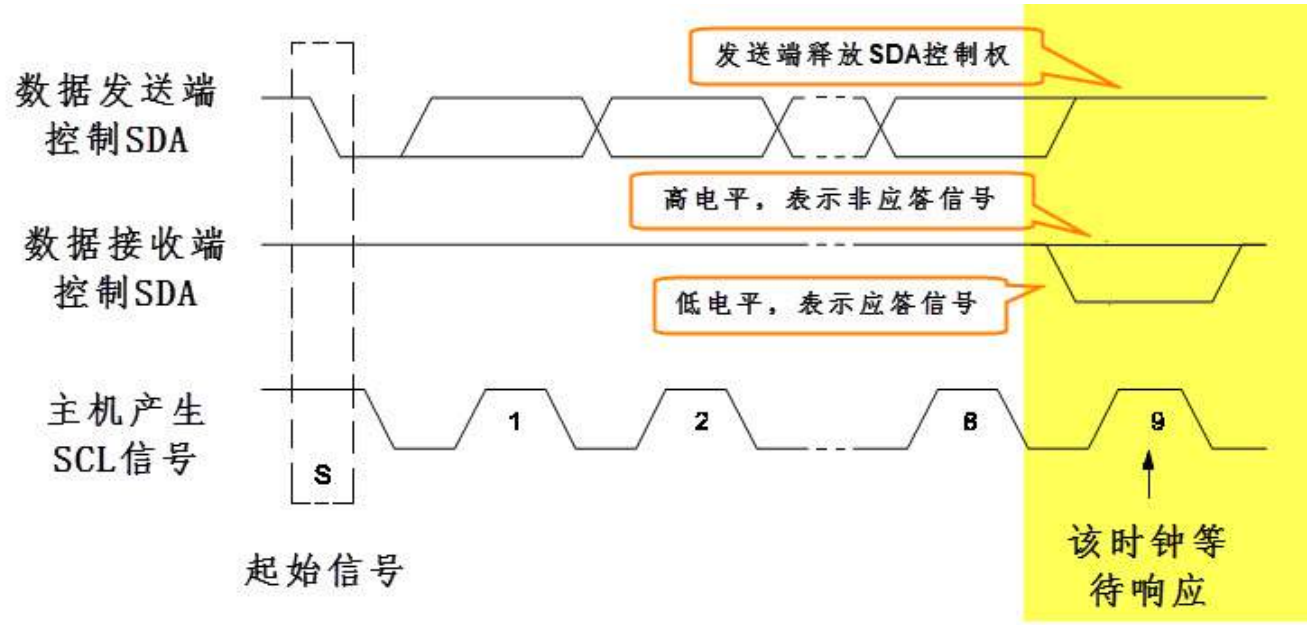


图 23-8 响应与非响应信号

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)。

23.2 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。

23.2.1 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。

23.2.2 STM32 的 I²C 架构剖析

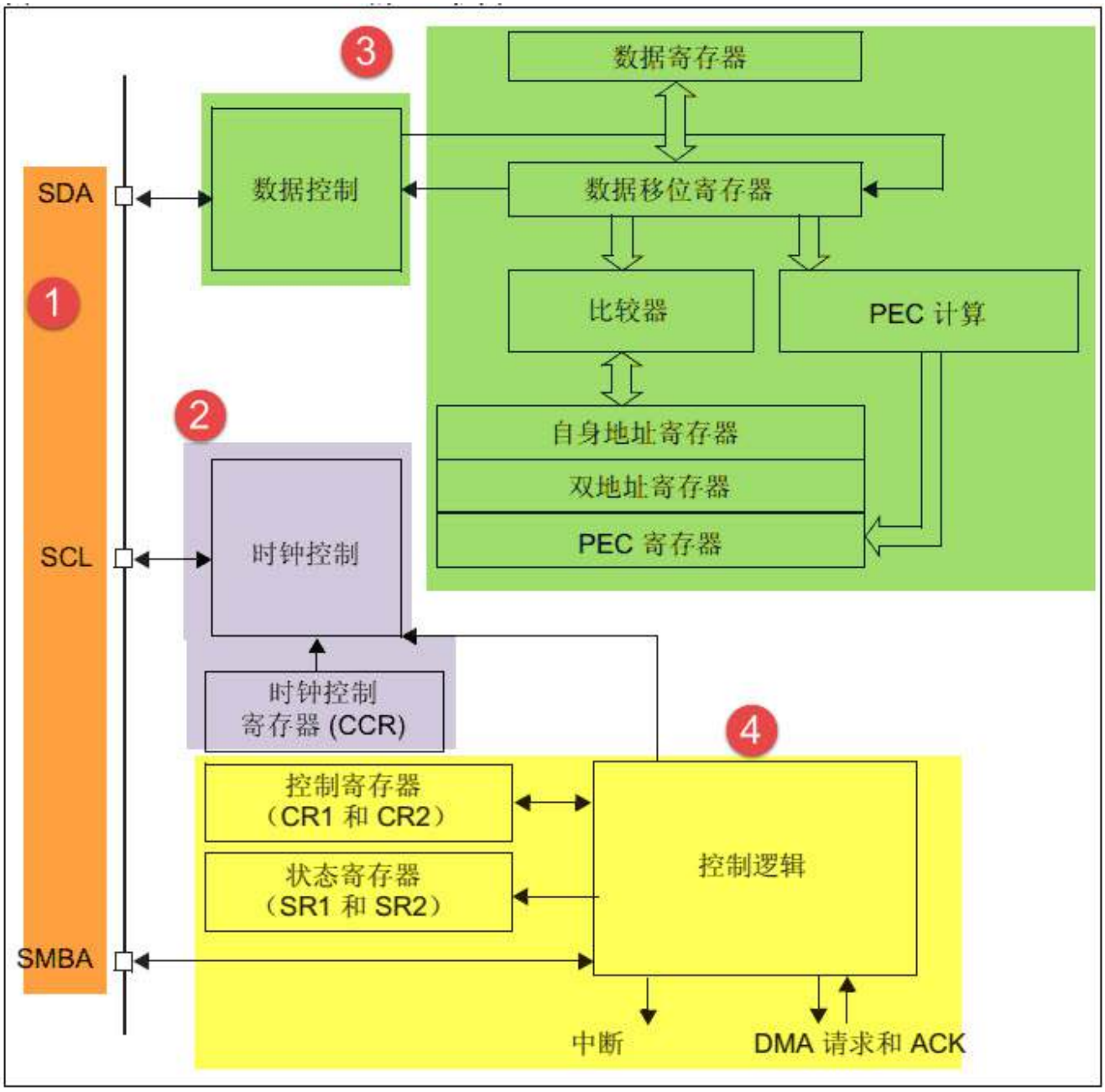


图 23-9 I2C 架构图

23.2.2.1 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《STM32F4xx 规格书》为准。

表 23-1 STM32F4xx 的 I2C 引脚 (整理自《STM32F4xx 规格书》)

引脚	I2C1	I2C2	I2C3
SCL	PB6/PB10(重映射)	PH4/PF1/PB10	PH7/PA8
SDA	PB7/PB9(重映射)	PH5/PF0/PB11	PH8/PC9

23.2.2.2 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:sub:*high*=2 时：

T_{high} = CCR*T_{PCKL1} T_{low} = 2*CCR*T_{PCKL1}

快速模式中 T_{low}/T:sub:*high*=16/9 时：

T_{high} = 9*CCR*T_{PCKL1} T_{low} = 16*CCR*T_{PCKL1}

例如，我们的 PCLK1=42MHz，想要配置 400Kbit/s 的速率，计算方式如下：

PCLK 时钟周期：TPCLK1 = 1/42000000

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 = 35

计算结果得出 CCR 为 35，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。

23.2.2.3 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。

23.2.2.4 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)。

23.2.3 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。

23.2.3.1 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。

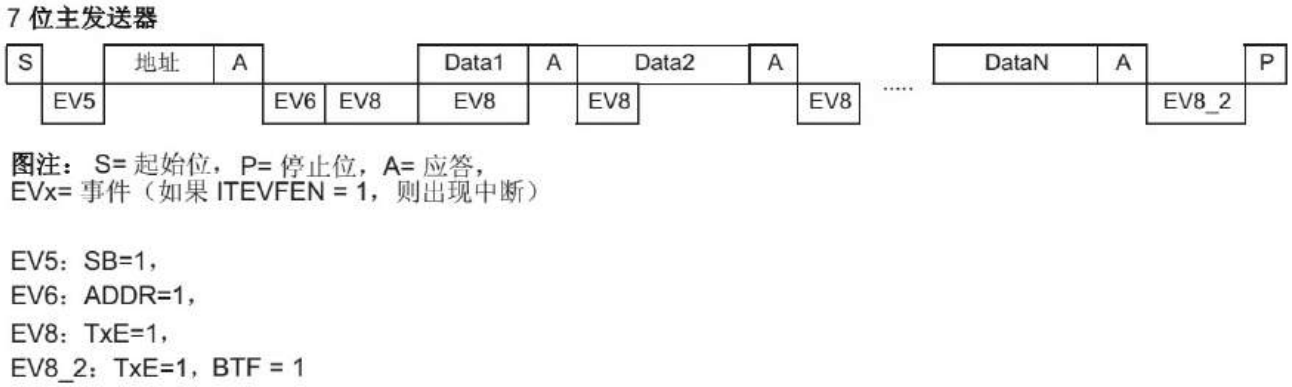


图 23-10 主发送器通讯过程

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。

23.2.3.2 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*。

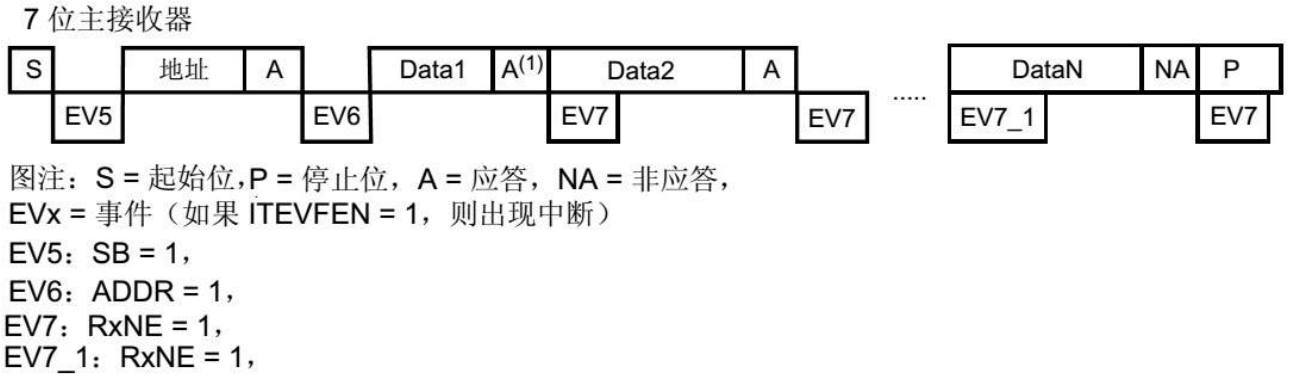


图 23-11 主接收器过程

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。

23.3 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 \
↪*/

```

(下页继续)

(续上页)

```
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，Ow-

nAddress1 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。

23.4 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。

23.4.1 硬件设计

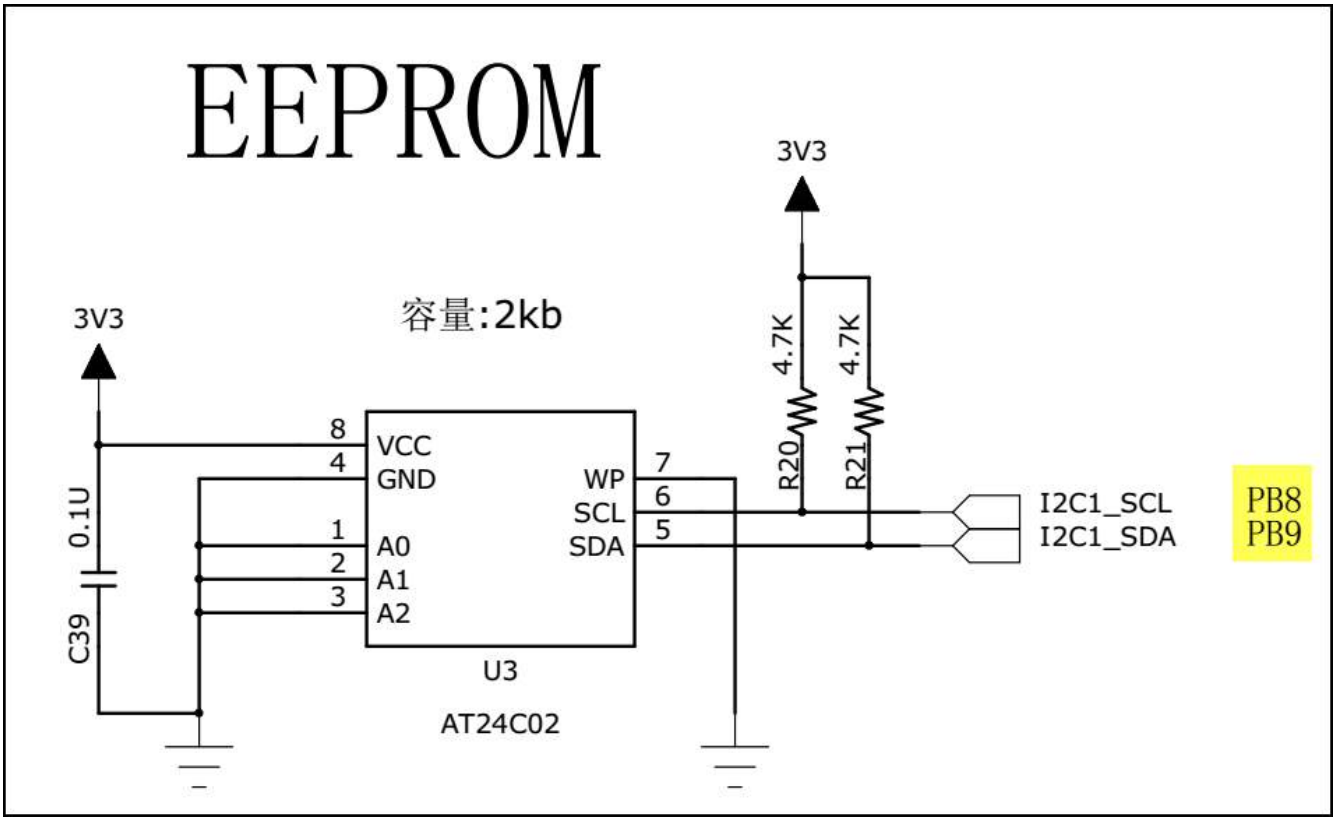


图 23-12 EEPROM 硬件连接图

本实验板中的 EEPROM 芯片 (型号：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。

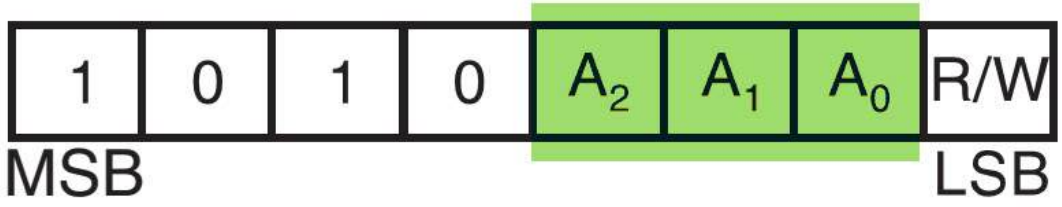


图 23-13 EEPROM 设备地址 (摘自《AT24C02》规格书)

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 0x50。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。

23.4.2 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。

23.4.2.1 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。

23.4.2.2 代码分析

23.4.2.2.1 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“bsp_i2c_ee.h”文件中，见*代码清单 23_2*。

列表 2: 代码清单 23-2 I2C 硬件配置相关的宏

```
/* 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。

23.4.2.2.2 初始化 I2C 的 GPIO

利用上面的宏，编写 I2C GPIO 引脚的初始化函数，见*代码清单 23_3*。

列表 3: 代码清单 23-3 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);
```

(下页继续)

(续上页)

```
    /* 强制 I2C 外设时钟复位 */
    I2Cx_FORCE_RESET() ;

    /* 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。

23.4.2.2.3 配置 I2C 的模式

以上只是配置了 I2C 使用的引脚，还不算对 I2C 模式的配置，见*代码清单 23_4*。

列表 4: 代码清单 23-4 配置 I2C 模式

```
/**
* @brief  I2C 工作模式配置
* @param  无
* @retval 无
*/
```

(下页继续)

(续上页)

```
static void I2C_Mode_Config(void)
{

    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。

23.4.2.2.4 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，见代码清单 *23_5*。

列表 5: 代码清单 23-5 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)
{
    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) {

    }
```

(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*.

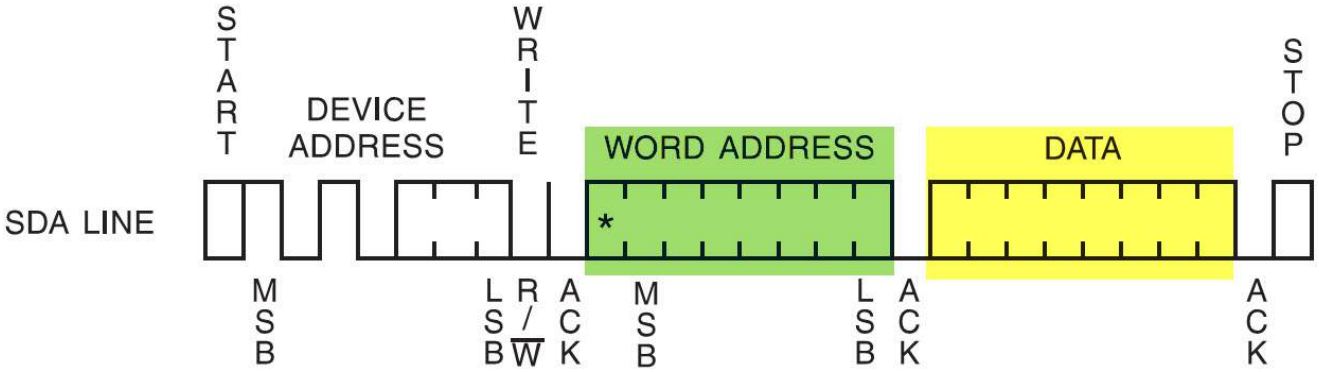


图 23-14 EEPROM 单字节写入时序 (摘自《AT24C02》规格书)

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。

23.4.2.2.5 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*。

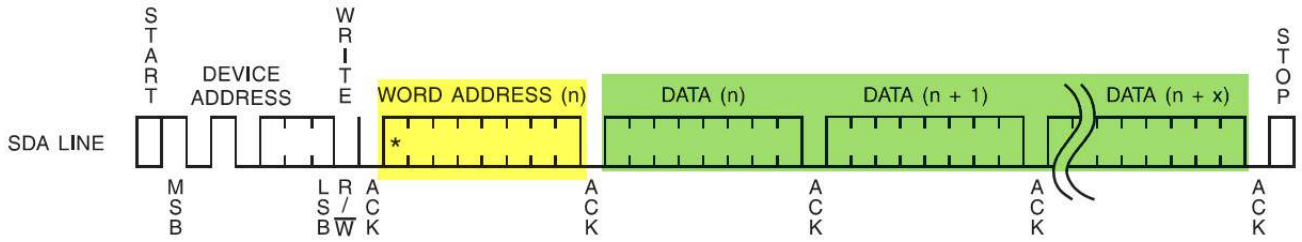


图 23-15 EEPROM 页写入时序 (摘自《AT24C02》规格书)

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 的页写入代码实现见*代码清单 23_6*。

列表 6: 代码清单 23-6 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)
```

(下页继续)

(续上页)

```
{
    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) {

    }

    /* 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。

23.4.2.2.6 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 7: 代码清单 23-7 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,
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 (NumOfPage == 0) {
            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;
```

(下页继续)

(续上页)

```
            }

            if (NumOfSingle!=0) {
                I2C_EE_PageWrite(pBuffer, WriteAddr, NumOfSingle);
            }
        }
    }
    /* If WriteAddr is not I2C_PageSize aligned  */
    else {
        /* If NumByteToWrite < I2C_PageSize */
        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);
```

(下页继续)

(续上页)

```
                }
            }
        }
    }
```

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。

23.4.2.2.7 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*。

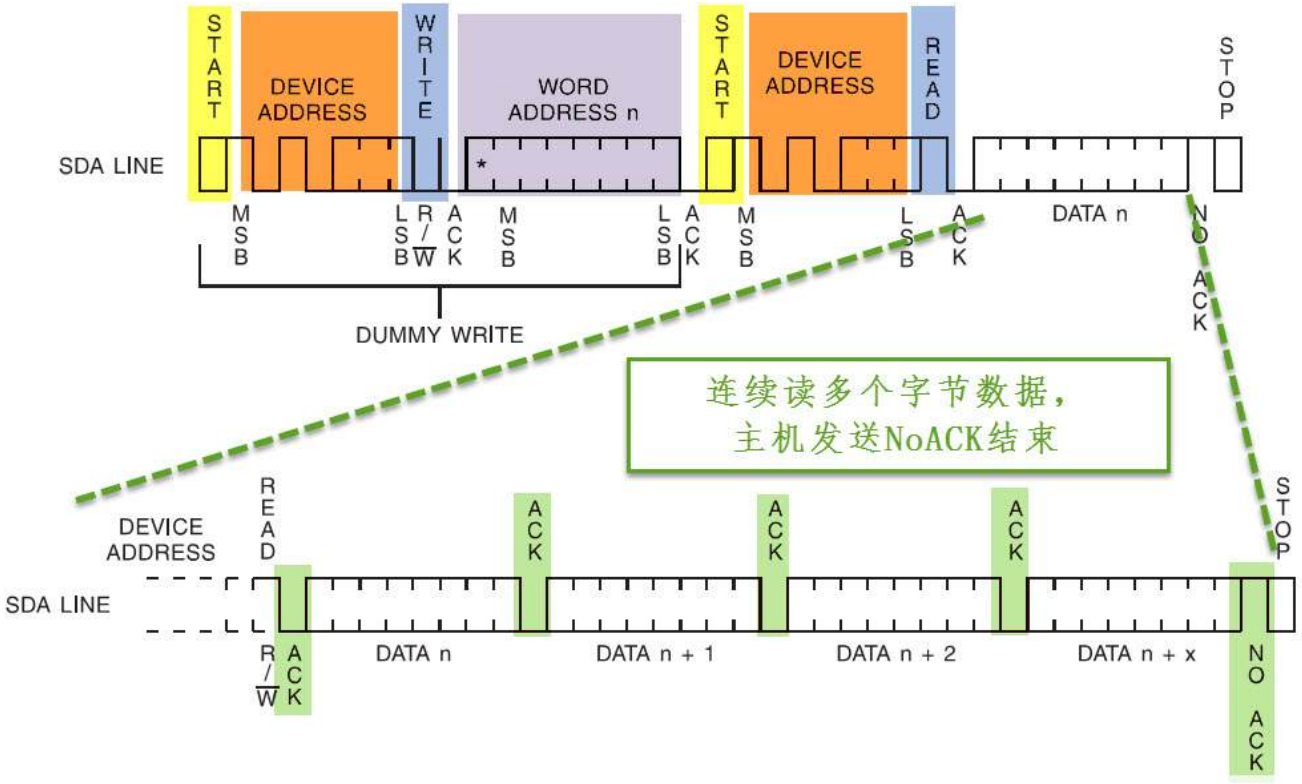


图 23-16 EEPROM 数据读取时序

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，实现代码见*代码清单 23_8*。

列表 8: 代码清单 23-8 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
```

(下页继续)

(续上页)

```
*   @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。

23.4.2.3 main 文件

23.4.2.3.1 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，见*代码清单 23_9*。

列表 9: 代码清单 23-9 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)
```

(下页继续)

(续上页)

```
{
    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");
            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;
```

(下页继续)

(续上页)

```
}
```

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。

23.4.2.3.2 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，见*代码清单 23_10*。

列表 10: 代码清单 23-10 main 函数

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /* 配置系统时钟为 168 MHz */
    SystemClock_Config();

    /* 初始化 RGB 彩灯 */
    LED_GPIO_Config();

    LED_BLUE;
    /* 初始化 USART1*/
```

(下页继续)

(续上页)

```
    UARTx_Config();

    printf("\r\n 欢迎使用野火  STM32 F407 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;
    }

    while (1) {

    }
}
```

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 EEPROM 测试的调试信息。

第 24 章　SPI—读写串行 FLASH

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》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《W25Q128》来了解。

24.1 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。

24.1.1 SPI 物理层

SPI 通讯设备之间的常用连接方式见图 *24_1*。

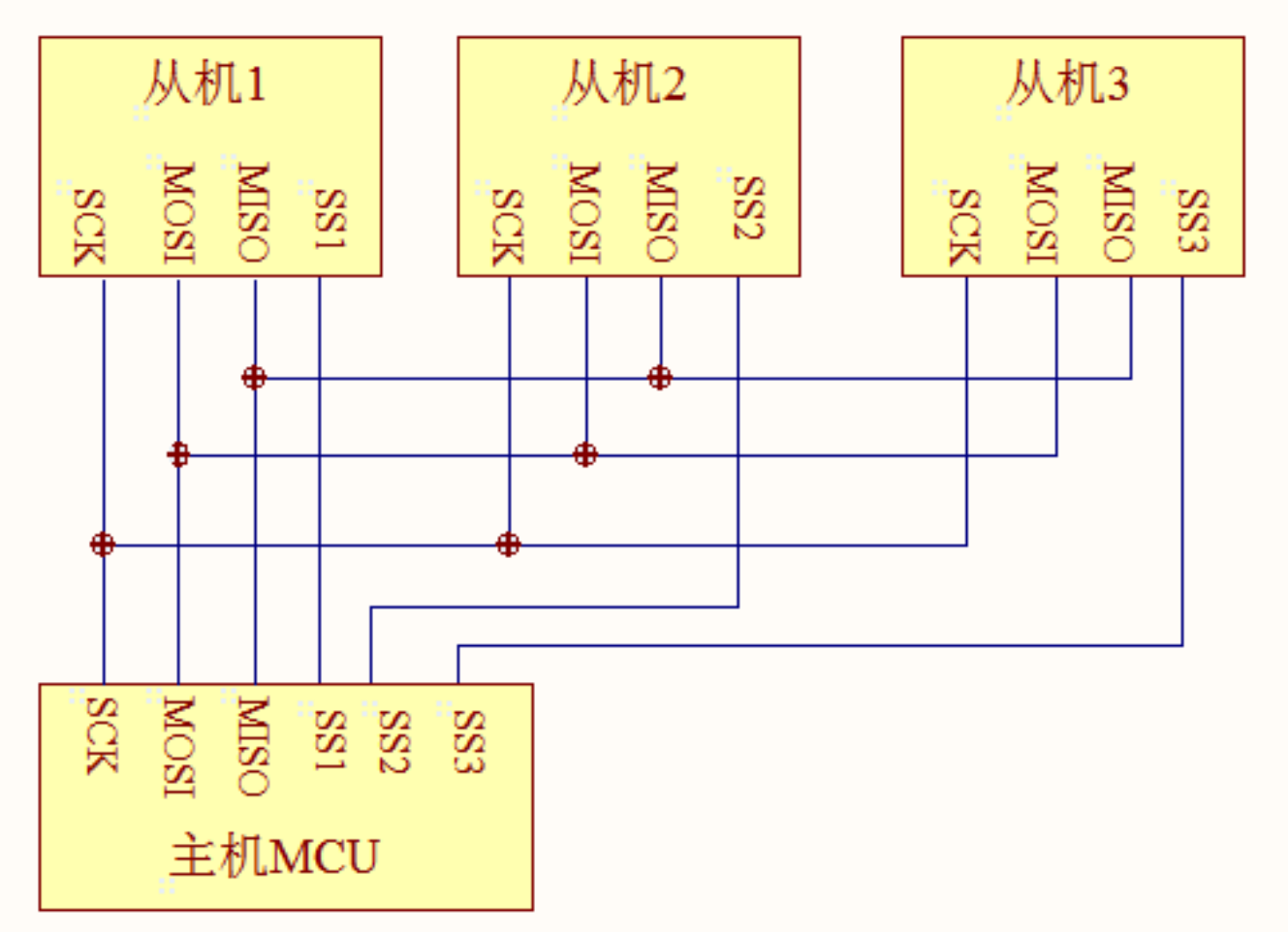


图 24-1 常见的 SPI 通讯系统

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 SS ，它们的作用介绍如下：

(1) SS*(* *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。

24.1.2 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。

24.1.2.1 SPI 基本通讯过程

先看看 SPI 通讯的通讯时序，见图 *24_2*。

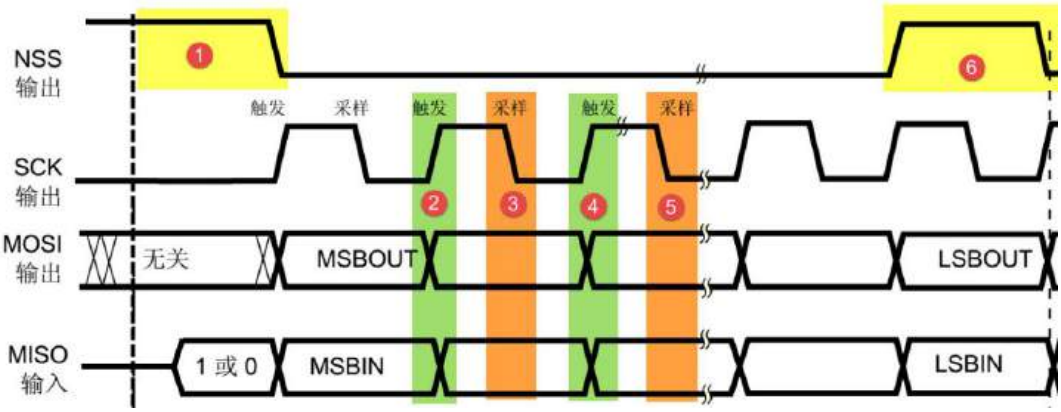


图 24-2 SPI 通讯时序

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：

24.1.2.2 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。

24.1.2.3 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。

24.1.2.4 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*。

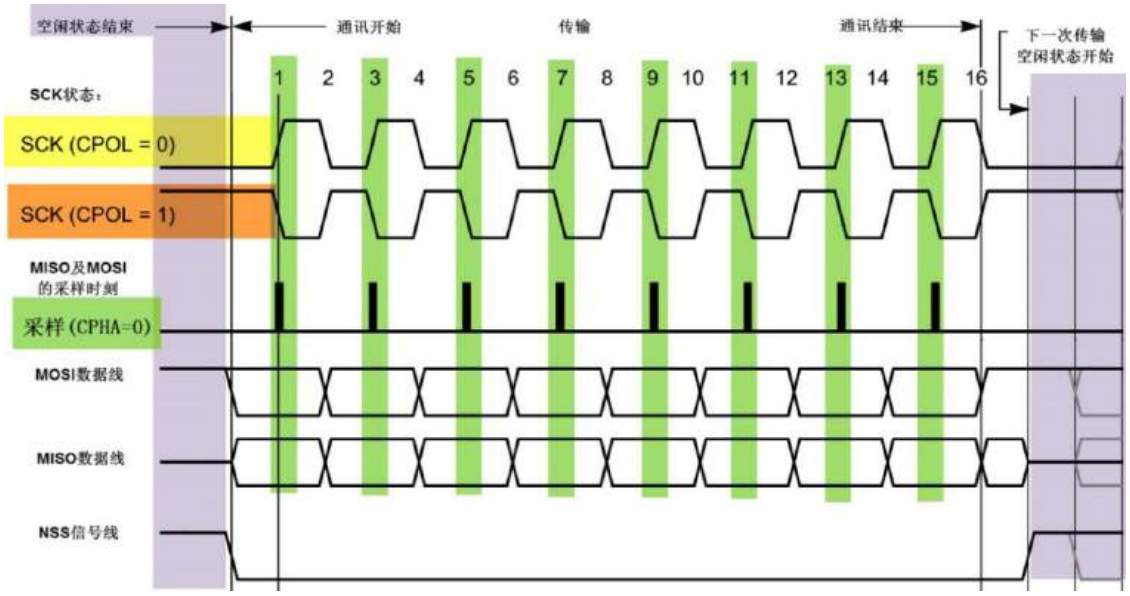


图 24-3 CPHA=0 时的 SPI 通讯模式

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*。

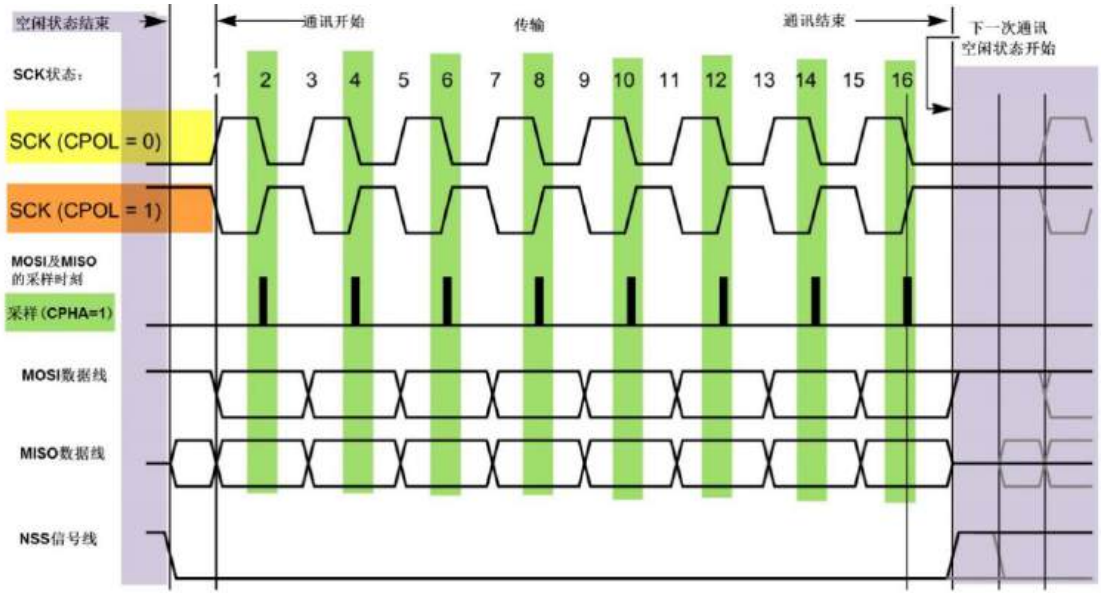


图 24-4 CPHA=1 时的 SPI 通讯模式

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

24.2 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。

24.2.1 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 72MHz，f_{pclk2} 为 36MHz)，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。

24.2.2 STM32 的 SPI 架构剖析

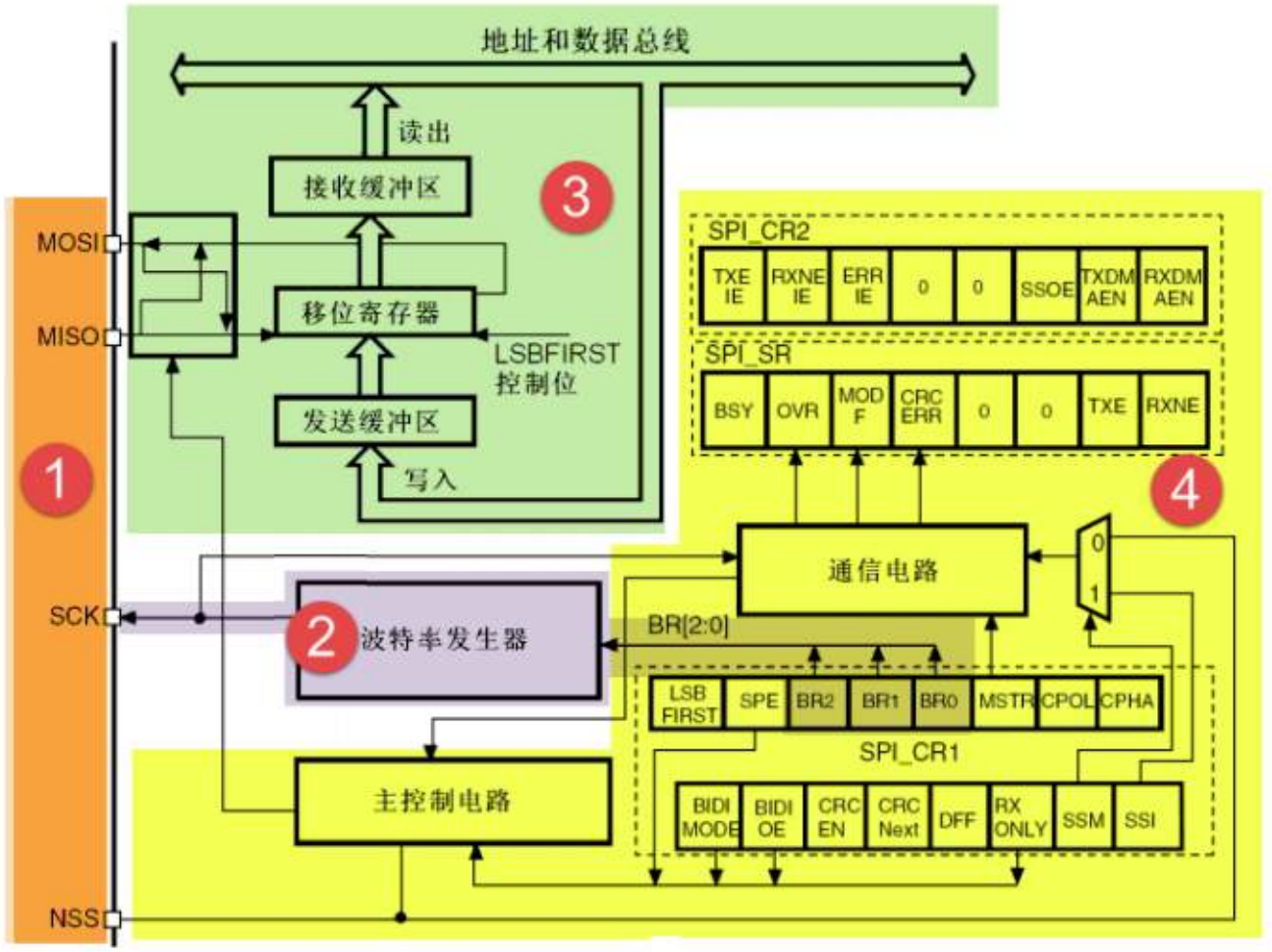


图 24-5 SPI 架构图

24.2.2.1 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《STM32F4xx 规格书》，以它为准。

表 24-2 STM32F4xx 的 SPI 引脚 (整理自《STM32F4xx 规格书》)

引脚	SPI 编号					
	SPI1	SPI2	SPI3	SPI4	SPI5	SPI6
NSS	PA7/PB5	PB15/PC3/PI3	PB5/PC12/PD6	PE6/PE14	PF9/PF11	PG14
CLK	PA6/PB4	PB14/PC2/PI2	PB4/PC11	PE5/PE13	PF8/PH7	PG12
MISO	PA5/PB3	PB10/PB13/PD3	PB3/PC10	PE2/PE12	PF7/PH6	PG13
MOSI	PA4/PA15	PB9/PB12/PI0	PA4/PA15	PE4/PE11	PF6/PH5	PG8

注：其中 PF、PH 端口在 176 引脚型号的芯片才有。

其中 SPI1、SPI4、SPI5、SPI6 是 APB2 上的设备，最高通信速率达 42Mbtis/s，SPI2、SPI3 是 APB1 上的设备，最高通信速率为 21Mbits/s。其它功能上没有差异。

24.2.2.2 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。

24.2.2.3 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 F 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。

24.2.2.4 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。

24.2.3 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。

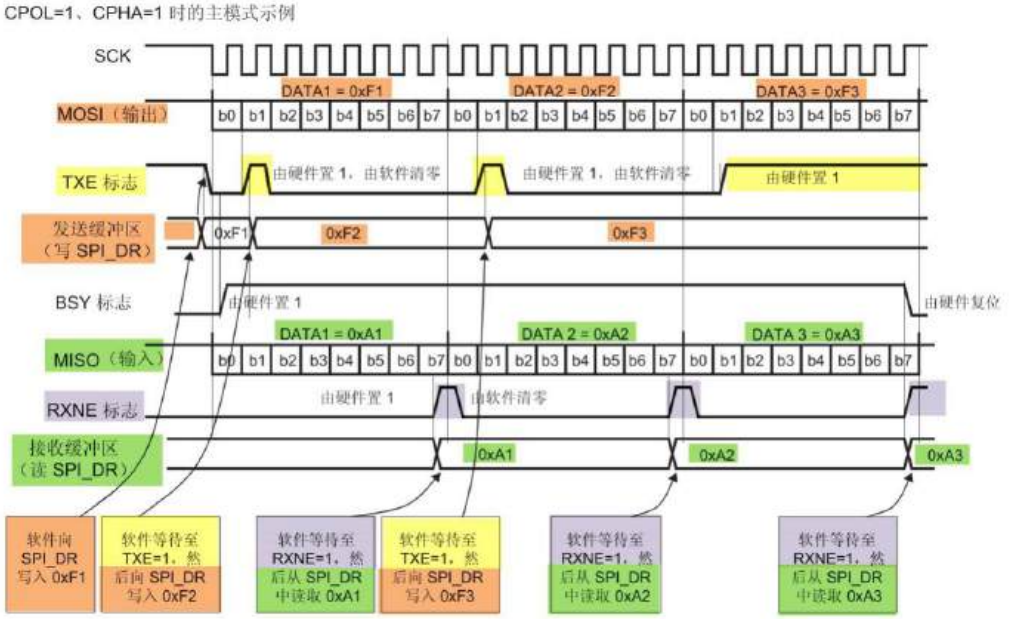


图 24-6 主发送器通讯过程

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。

24.3 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”及“stm32f4xx_hal_spi.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 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 */
```

(下页继续)

(续上页)

```
    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_PHASE_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。

24.4 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。

24.4.1 硬件设计

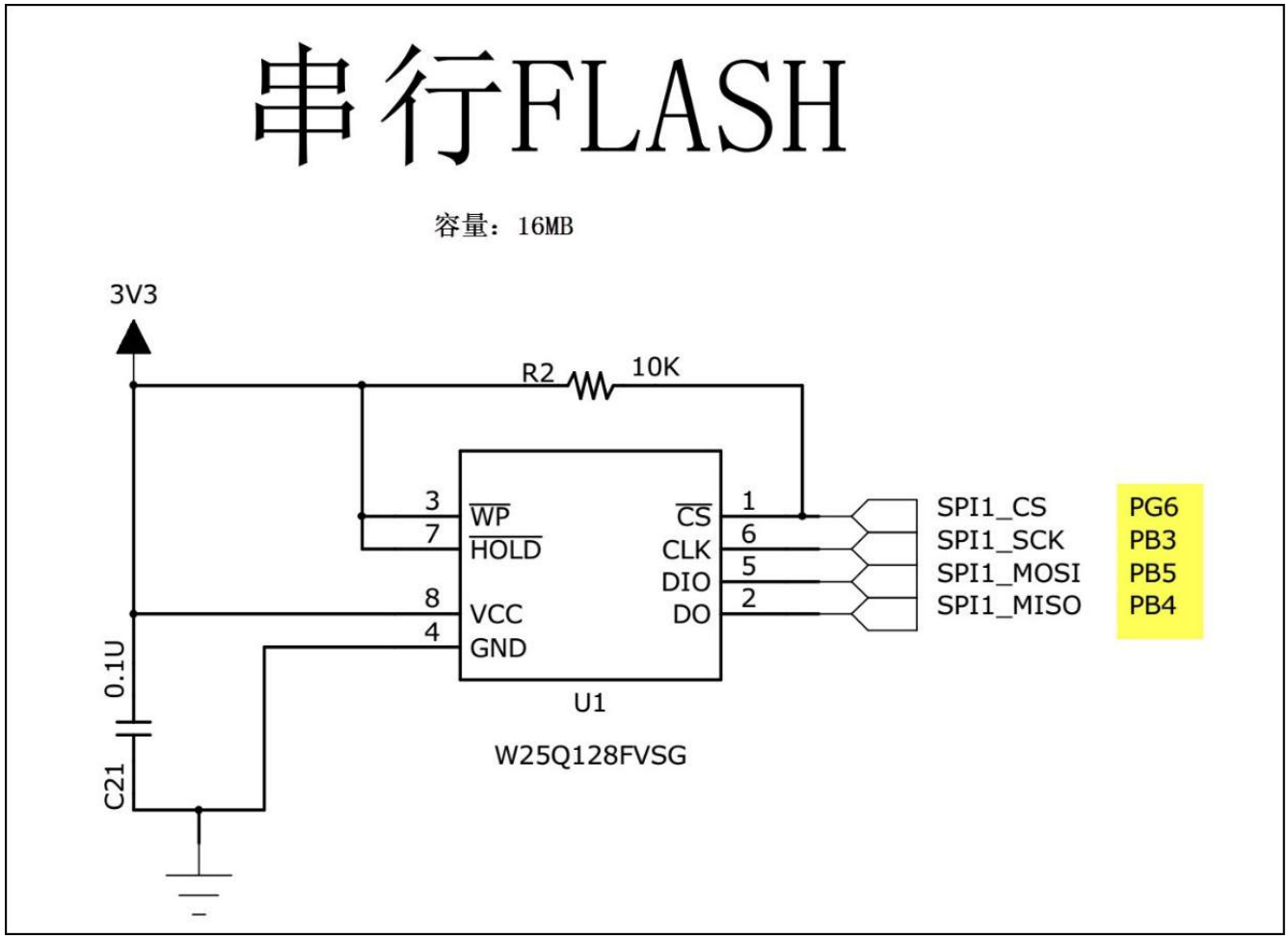


图 24-7 SPI 串行 FLASH 硬件连接图

本实验板中的 FLASH 芯片 (型号：W25Q128) 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 SDI 引脚 NSS/SCK/MOSI/MISO 上，其中 STM32 的 NSS 引脚是一个普通的 GPIO，不是 SPI 的专用 NSS 引脚，所以程序中我们要使用软件控制的方式。

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《W25Q128》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。

24.4.2 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。

24.4.2.1 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。

24.4.2.2 代码分析

24.4.2.2.1 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_2*。

列表 2: 代码清单 24-2 SPI 硬件配置相关的宏

```
//SPI 号及时钟初始化函数
#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_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_GPIOG_CLK_ENABLE()

#define SPIx_FORCE_RESET()                __HAL_RCC_SPI1_FORCE_RESET()
#define SPIx_RELEASE_RESET()              __HAL_RCC_SPI1_RELEASE_RESET()

//SCK 引脚
#define SPIx_SCK_PIN                      GPIO_PIN_3
#define SPIx_SCK_GPIO_PORT                GPIOB
#define SPIx_SCK_AF                       GPIO_AF5_SPI1
//MISO 引脚
#define SPIx_MISO_PIN                     GPIO_PIN_4
#define SPIx_MISO_GPIO_PORT               GPIOB
#define SPIx_MISO_AF                      GPIO_AF5_SPI1
//MOSI 引脚
#define SPIx_MOSI_PIN                     GPIO_PIN_5
#define SPIx_MOSI_GPIO_PORT               GPIOB
#define SPIx_MOSI_AF                      GPIO_AF5_SPI1
//CS(NSS) 引脚
#define FLASH_CS_PIN                      GPIO_PIN_6
#define FLASH_CS_GPIO_PORT                GPIOG
//设置为高电平
#define digitalHi(p,i)           {p->BSRR=i;}
//输出低电平
#define digitalLo(p,i)           {p->BSRR=(uint32_t)i << 16;}
#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。

24.4.2.2.2 初始化 SPI 的 GPIO

利用上面的宏，编写 SPI 的初始化函数，见*代码清单 24_3*。

列表 3: 代码清单 24-3 SPI 的初始化函数 (GPIO 初始化部分)

```
/**
  * @brief SPI MSP 初始化
  *        此函数配置此示例中使用的硬件资源:
  *            - 外设时钟使能
  *            - 外设引脚配置
  * @param hspi: SPI 句柄指针
  * @retval 无
  */
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;
```

(下页继续)

(续上页)

```
    GPIO_InitStruct.Speed     = GPIO_SPEED_FAST;
    GPIO_InitStruct.Alternate = SPIx_SCK_AF;

    HAL_GPIO_Init(SPIx_SCK_GPIO_PORT, &GPIO_InitStruct);

    /* SPI MISO GPIO pin configuration  */
    GPIO_InitStruct.Pin = SPIx_MISO_PIN;
    GPIO_InitStruct.Alternate = SPIx_MISO_AF;

    HAL_GPIO_Init(SPIx_MISO_GPIO_PORT, &GPIO_InitStruct);

    /* SPI MOSI GPIO pin configuration  */
    GPIO_InitStruct.Pin = SPIx_MOSI_PIN;
    GPIO_InitStruct.Alternate = SPIx_MOSI_AF;
    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。

24.4.2.2.3 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《W25Q128》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 4: 代码清单 24-4 配置 SPI 模式

```
void SPI_FLASH_Init(void)
{
    /* Set the SPI parameters */
    SpiHandle.Instance               = SPIx;
    SpiHandle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
    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。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。

24.4.2.2.4 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 5: 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 */
```

(下页继续)

(续上页)

```
    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);

    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);

}

/**
```

(下页继续)

(续上页)

```
* @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。

24.4.2.2.5 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《W25Q128》，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	(ID15-ID8) 存储器类	(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*。

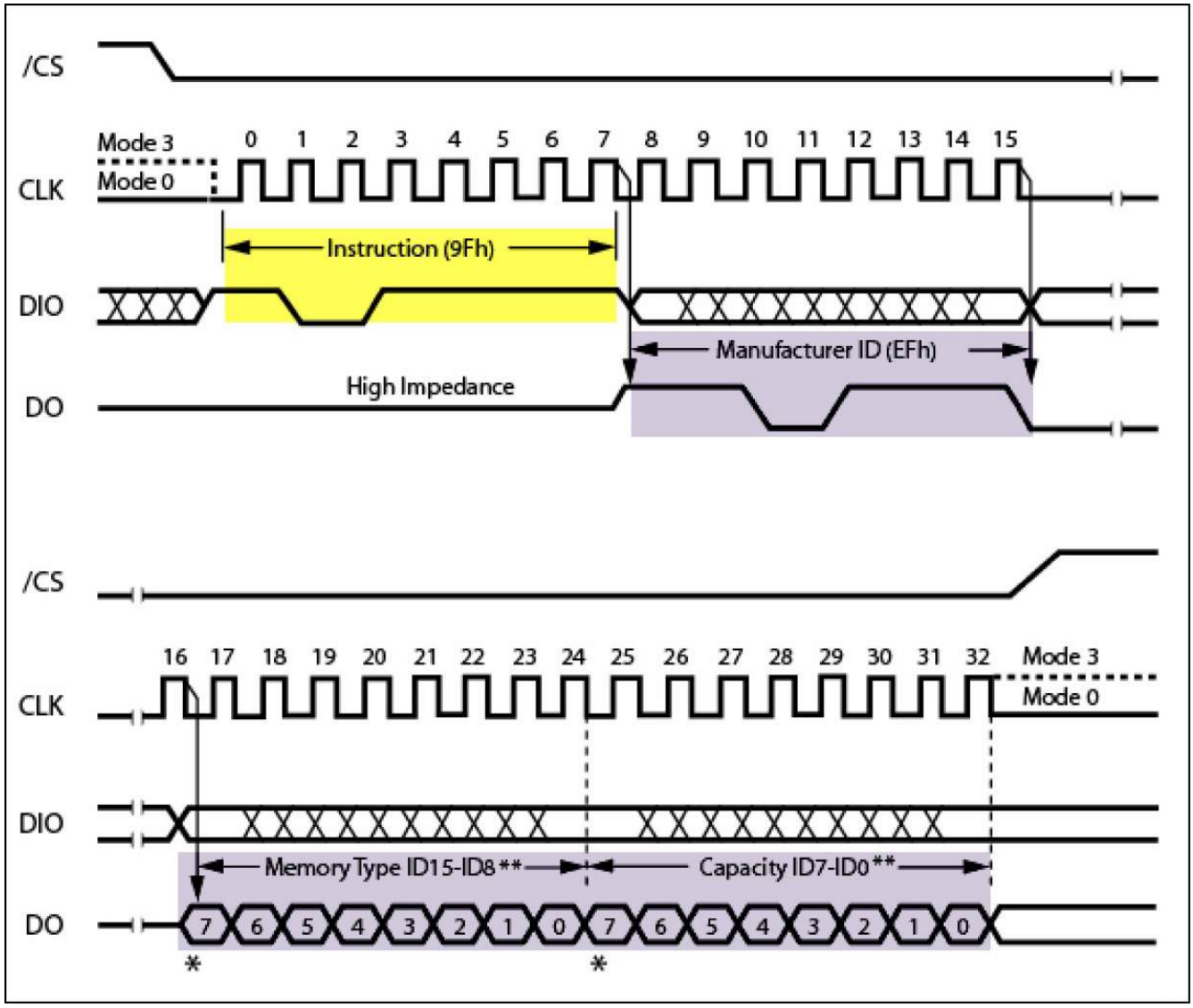


图 24-8 FLASH 读 ID 指令“JEDEC ID”的时序 (摘自规格书《W25Q64》)

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。

24.4.2.2.6 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 6: 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
#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
```

(下页继续)

(续上页)

```
/* 其它 */
#define sFLASH_ID 0XEF4018
#define Dummy_Byte 0xFF
```

24.4.2.2.7 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 7: 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 */
```

(下页继续)

(续上页)

```
    SPI_FLASH_SendByte(W25X_JedecDeviceID);

    /* 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。

24.4.2.2.8 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 8: 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();
```

(下页继续)

(续上页)

```
}
```

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*。

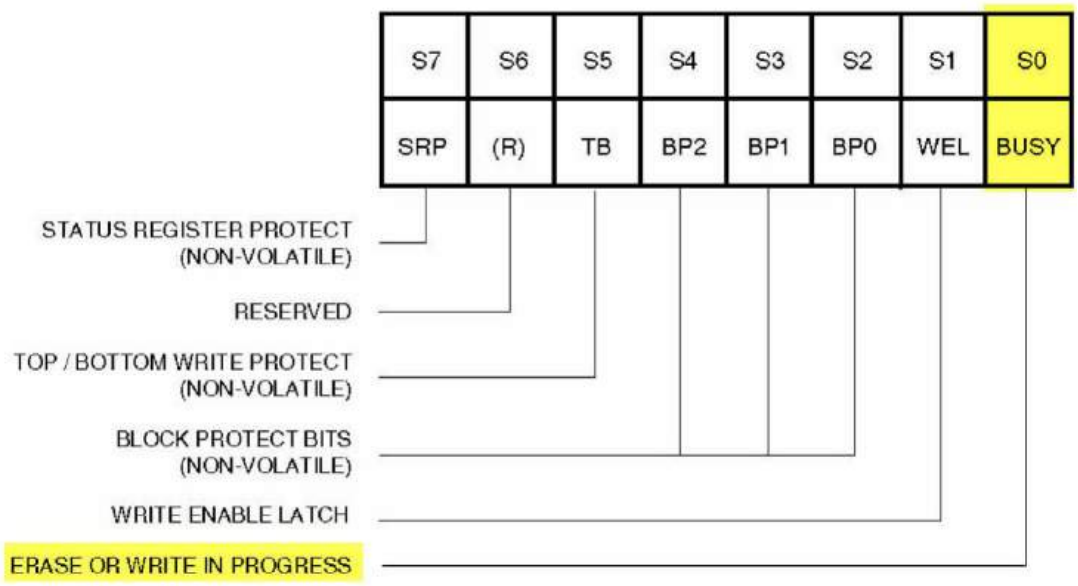


图 24-9 FLASH 芯片的状态寄存器

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*。

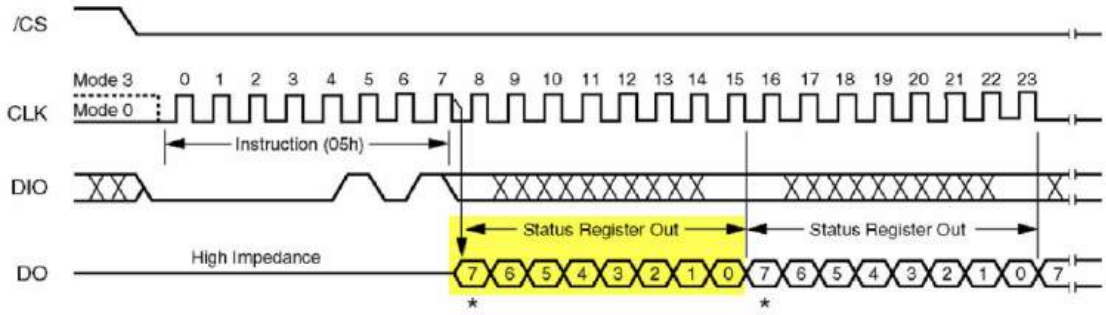


图 24-10 读取状态寄存器的时序

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 9: 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 */
```

(下页继续)

(续上页)

```
    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。

24.4.2.2.9 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*。

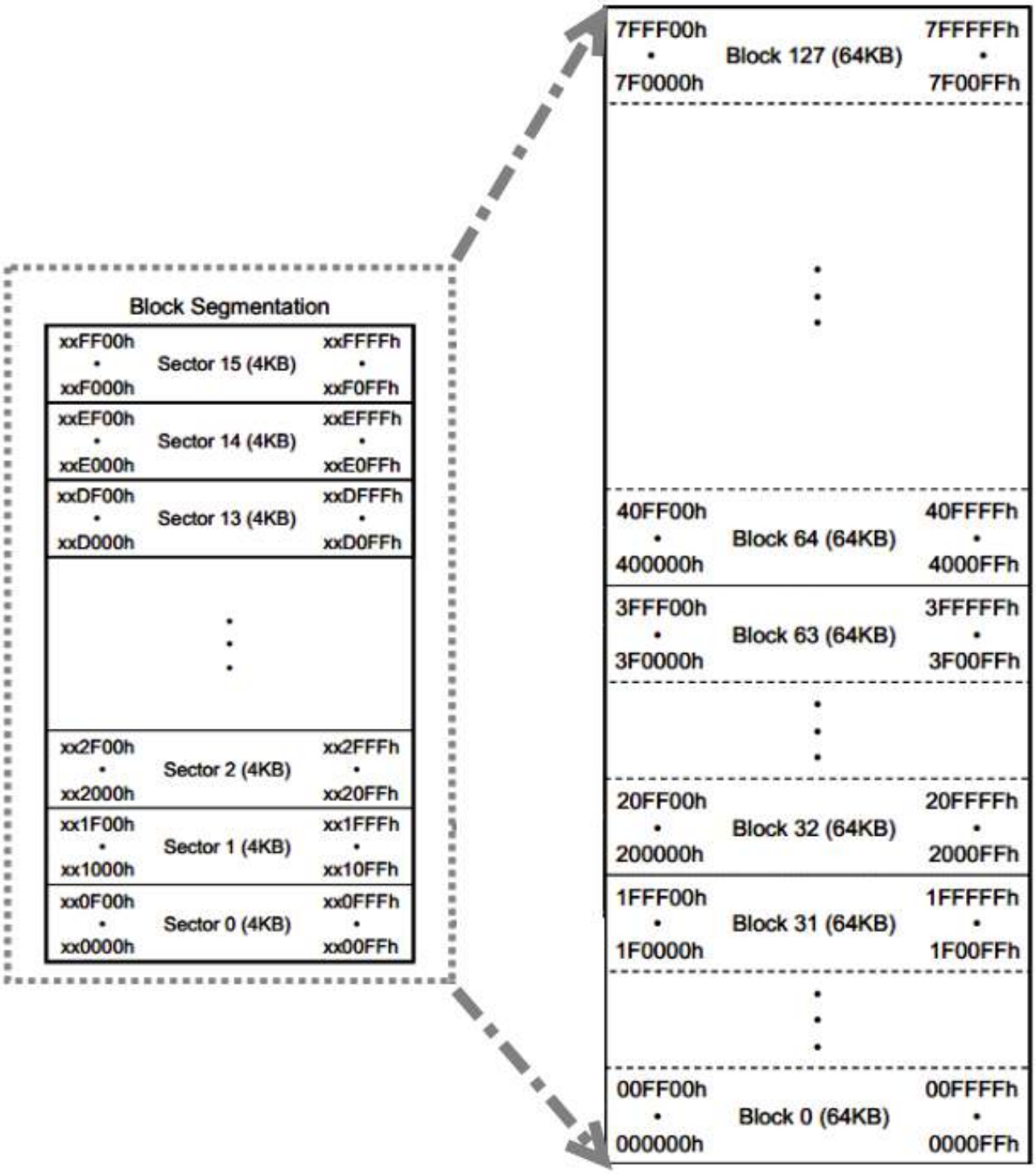


图 24-11 FLASH 芯片的存储矩阵

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*。

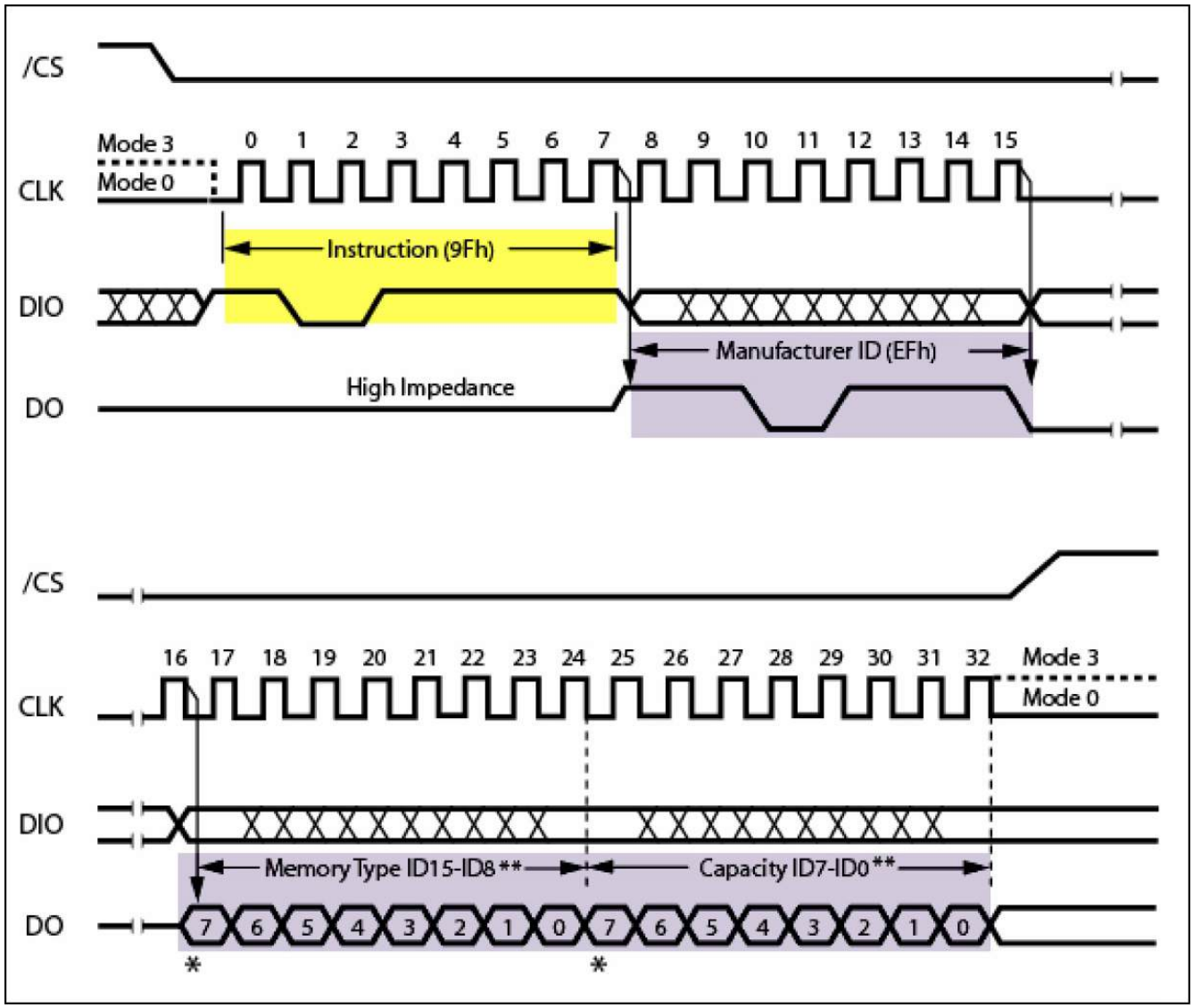


图 24-12 扇区擦除时序

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 10: 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 */
    SPI_FLASH_CS_LOW();
    /* 发送扇区擦除指令 */
    SPI_FLASH_SendByte(W25X_SectorErase);
```

(下页继续)

(续上页)

```
    /* 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。

24.4.2.2.10 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*。

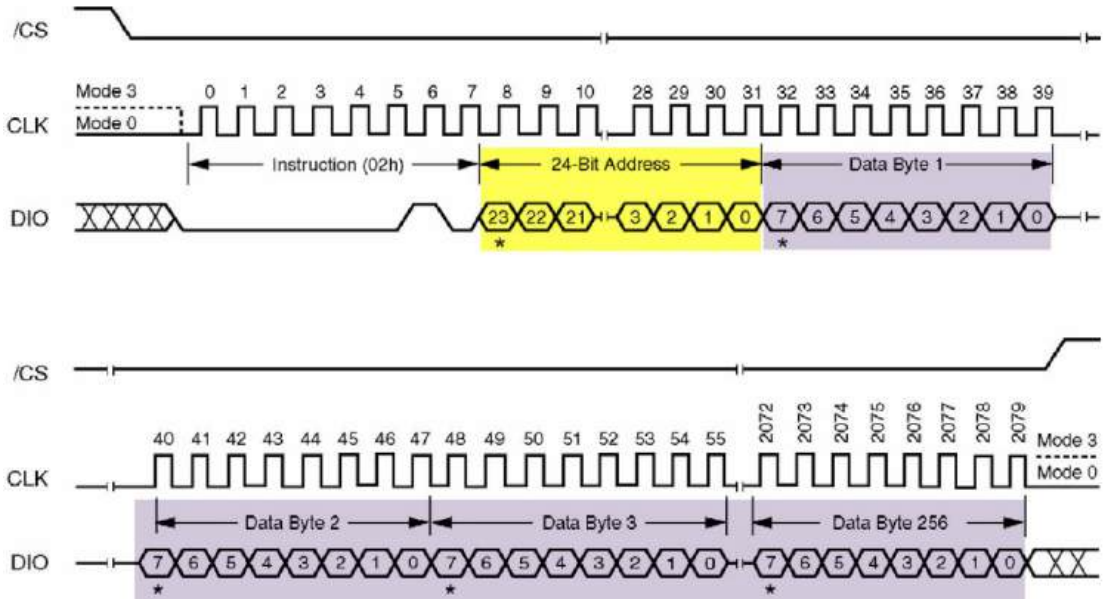


图 24-13 FLASH 芯片页写入

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 11: 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。

24.4.2.2.11 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 12: 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, NumOfSingle);
        }
    }
```

(下页继续)

(续上页)

```
    /* 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，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;
            NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;

            SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
```

(下页继续)

(续上页)

```
            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。

24.4.2.2.12 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*。

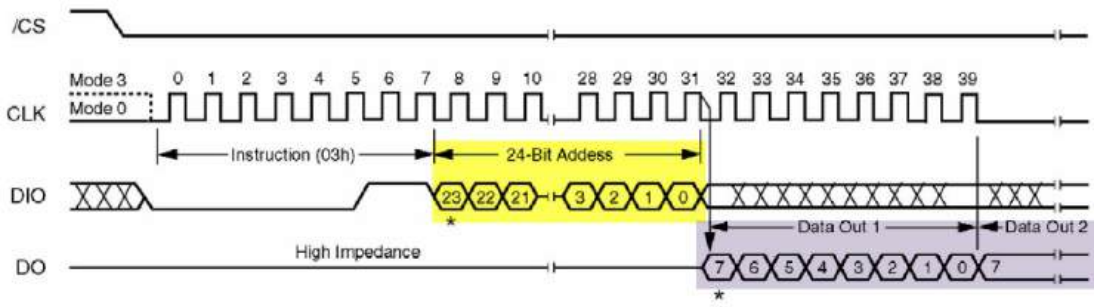


图 24-14 SPI FLASH 读取数据时序

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 13: 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 */
    SPI_FLASH_SendByte((ReadAddr & 0xFF0000) >> 16);
    /* 发送 读 地址中位 */
```

(下页继续)

(续上页)

```
    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。

24.4.2.3 main 函数

最后我们来编写 main 函数，进行 FLASH 芯片读写校验，见代码清单 *24_14*。

列表 14: 代码清单 24-14 main 函数

```
/* 获取缓冲区的长度 */
#define TxBufferSize1   (countof(TxBuffer1) - 1)
#define RxBufferSize1   (countof(TxBuffer1) - 1)
#define countof(a)      (sizeof(a) / sizeof(*(a)))
#define  BufferSize (countof(Tx_Buffer)-1)

#define  FLASH_WriteAddress     0x00000
```

(下页继续)

(续上页)

```
#define  FLASH_ReadAddress      FLASH_WriteAddress
#define  FLASH_SectorToErase    FLASH_WriteAddress

/* 发送缓冲区初始化 */
uint8_t Tx_Buffer[] = "感谢您选用野火 stm32 开发板\r\n";
uint8_t Rx_Buffer[BufferSize];

//读取的 ID 存储位置
__IO uint32_t DeviceID = 0;
__IO uint32_t FlashID = 0;
__IO TestStatus TransferStatus1 = FAILED;

// 函数原型声明
void Delay(__IO uint32_t nCount);

/*
* 函数名：main
* 描述  ：主函数
* 输入  ：无
* 输出  ：无
*/
int main(void)
{
    SystemClock_Config();
    LED_GPIO_Config();
    LED_BLUE;
    /* 配置串口 1 为：115200 8-N-1 */
    Debug_USART_Config();

    printf("\r\n这是一个 16M 串行 flash(W25Q128) 实验 \r\n");

    /* 16M 串行 flash W25Q128 初始化 */
```

(下页继续)

(续上页)

```
    SPI_FLASH_Init();

    Delay( 200 );

    /* 获取 SPI Flash ID */
    FlashID = SPI_FLASH_ReadID();

    /* 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 */
        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\n16M 串行 flash(W25Q128) 测试成功!\n\r");
        }
        else
```

(下页继续)

(续上页)

```
        {
            LED_RED;
            printf("\r\n16M 串行 flash(W25Q128) 测试失败!\n\r");
        }
    }// if (FlashID == sFLASH_ID)
    else
    {
        LED_RED;
        printf("\r\n获取不到 W25Q128 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。

24.4.3 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 25 章　串行 FLASH 文件系统 FatFs

25.1 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-F407 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。

25.2 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

25.2.1 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。

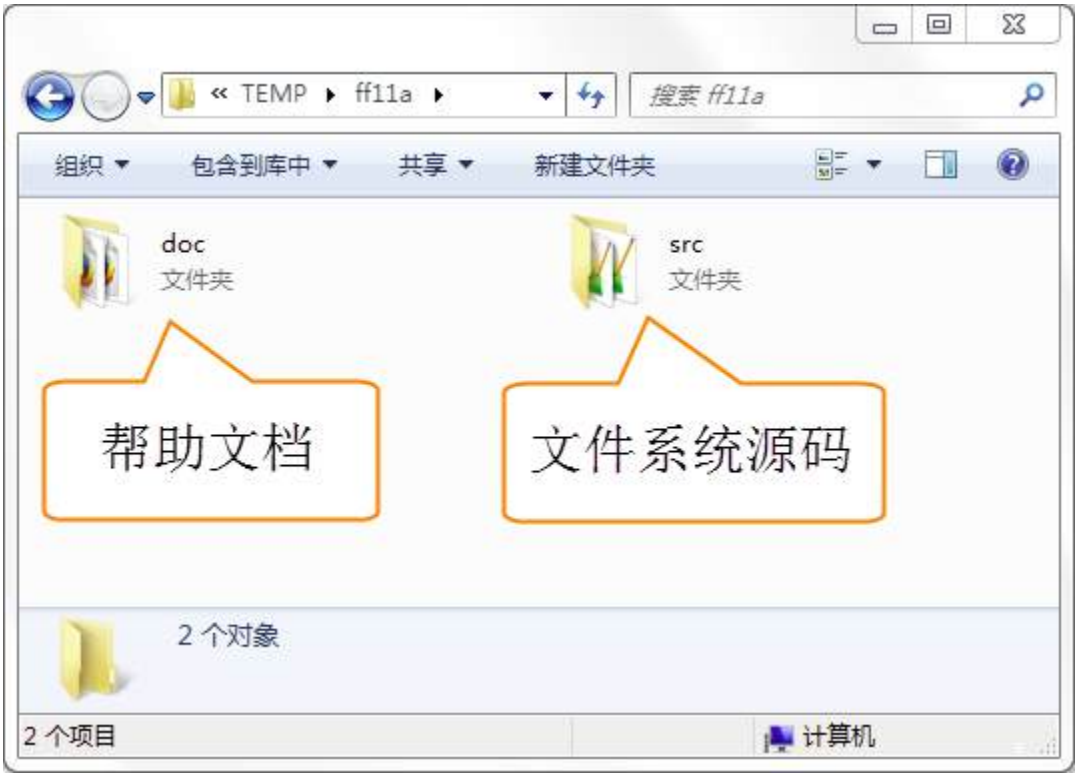


图 25-2 FatFs 文件目录

25.2.2 FatFs 帮助文档

打开 doc 文件夹，可看到如图 *25_3* 的文件目录：

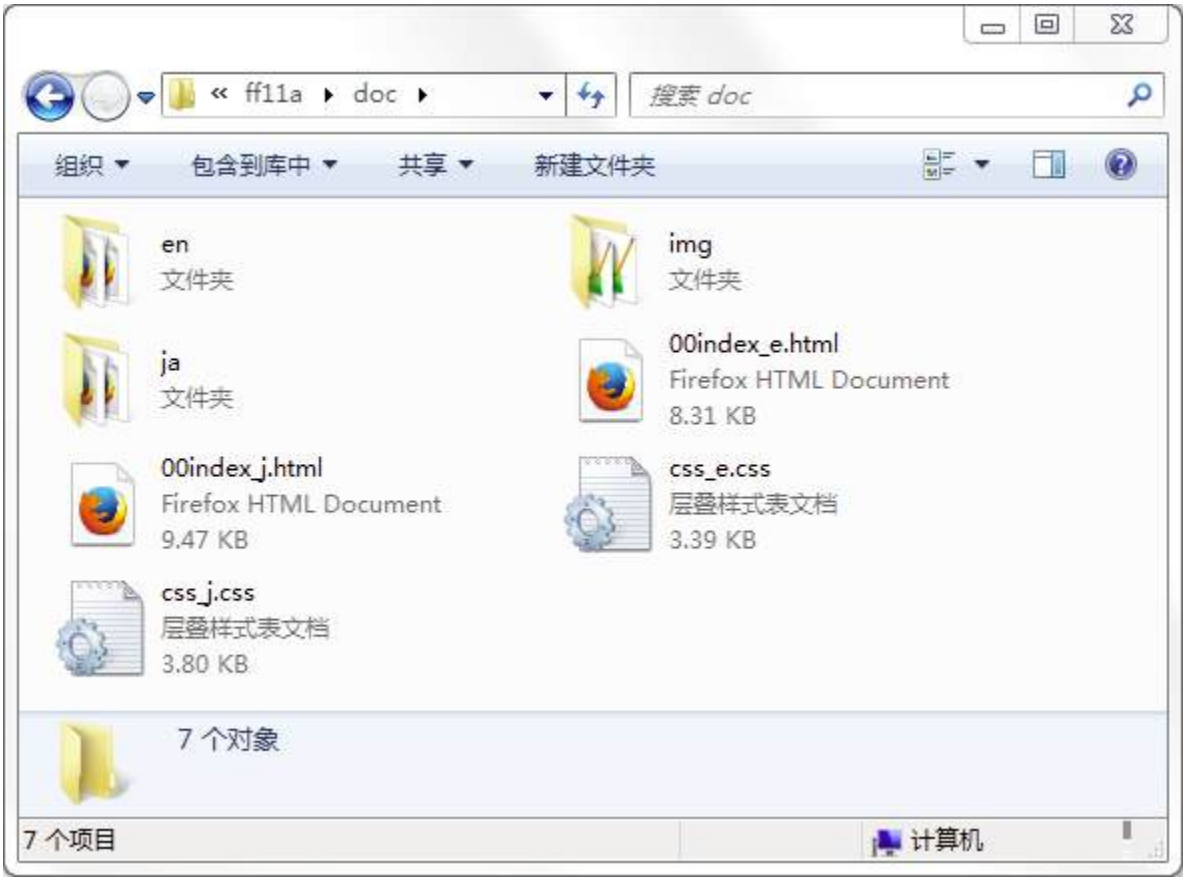


图 25-3 doc 文件夹的文件目录

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。

25.2.3 FATFS 源码

打开 src 文件夹，可看到如图 *25_4* 的文件目录：

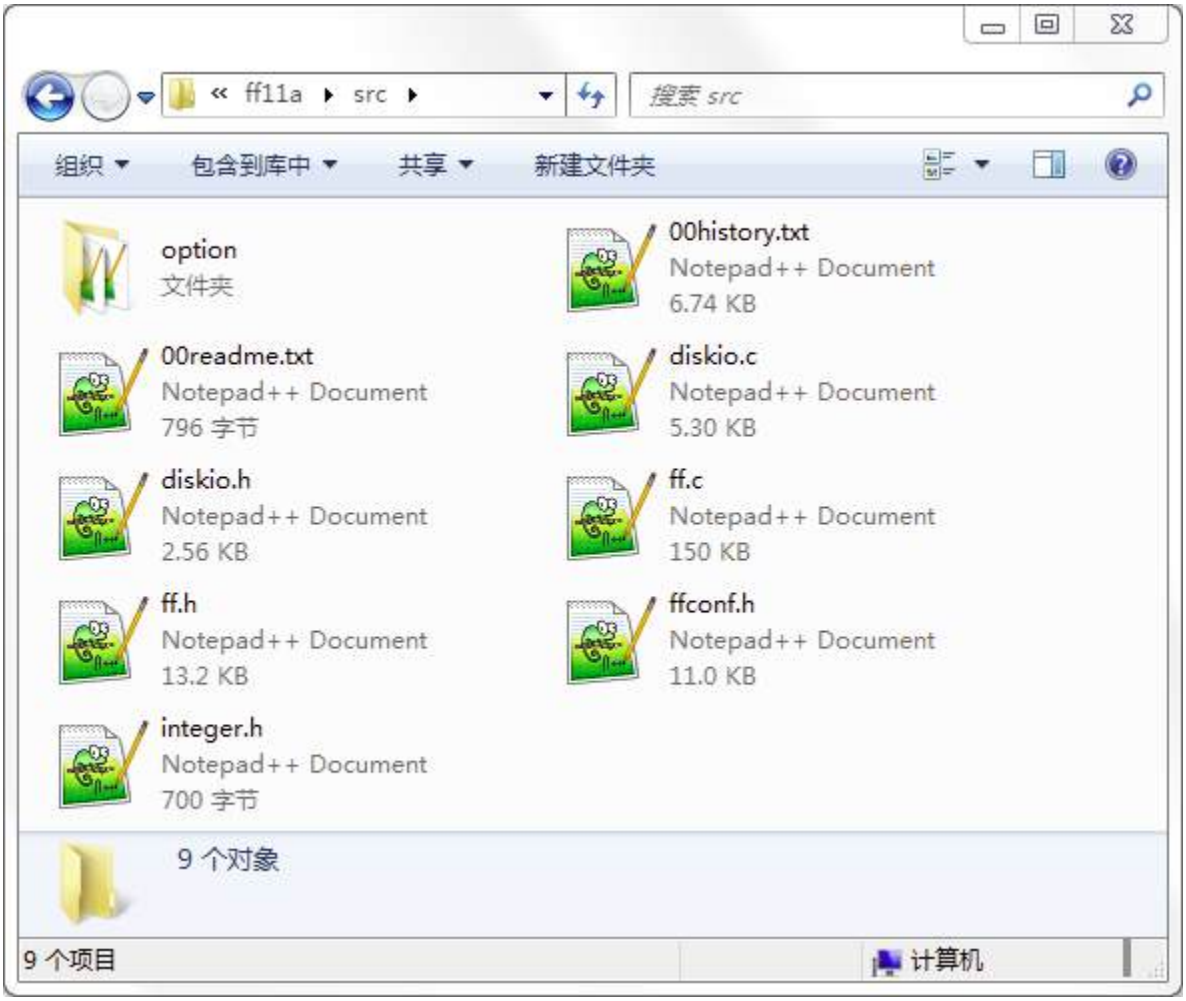


图 25-4 src 文件夹的文件目录

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。

25.3 FatFs 文件系统移植实验

25.3.1 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*。

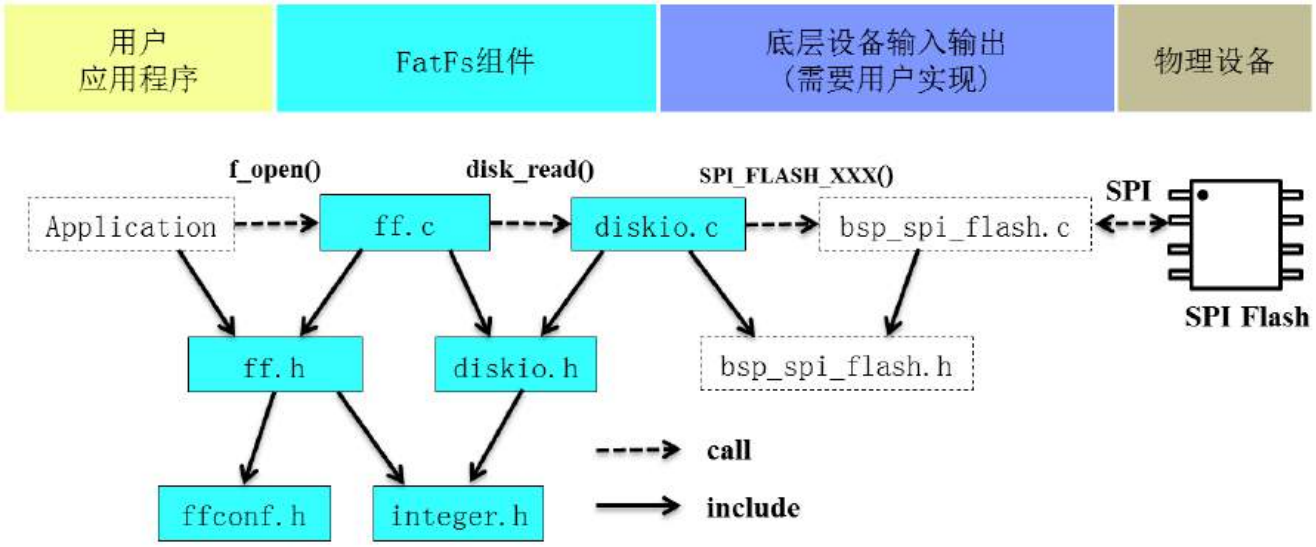


图 25-5 FatFs 程序结构图

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。

25.3.2 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。

25.3.3 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*。

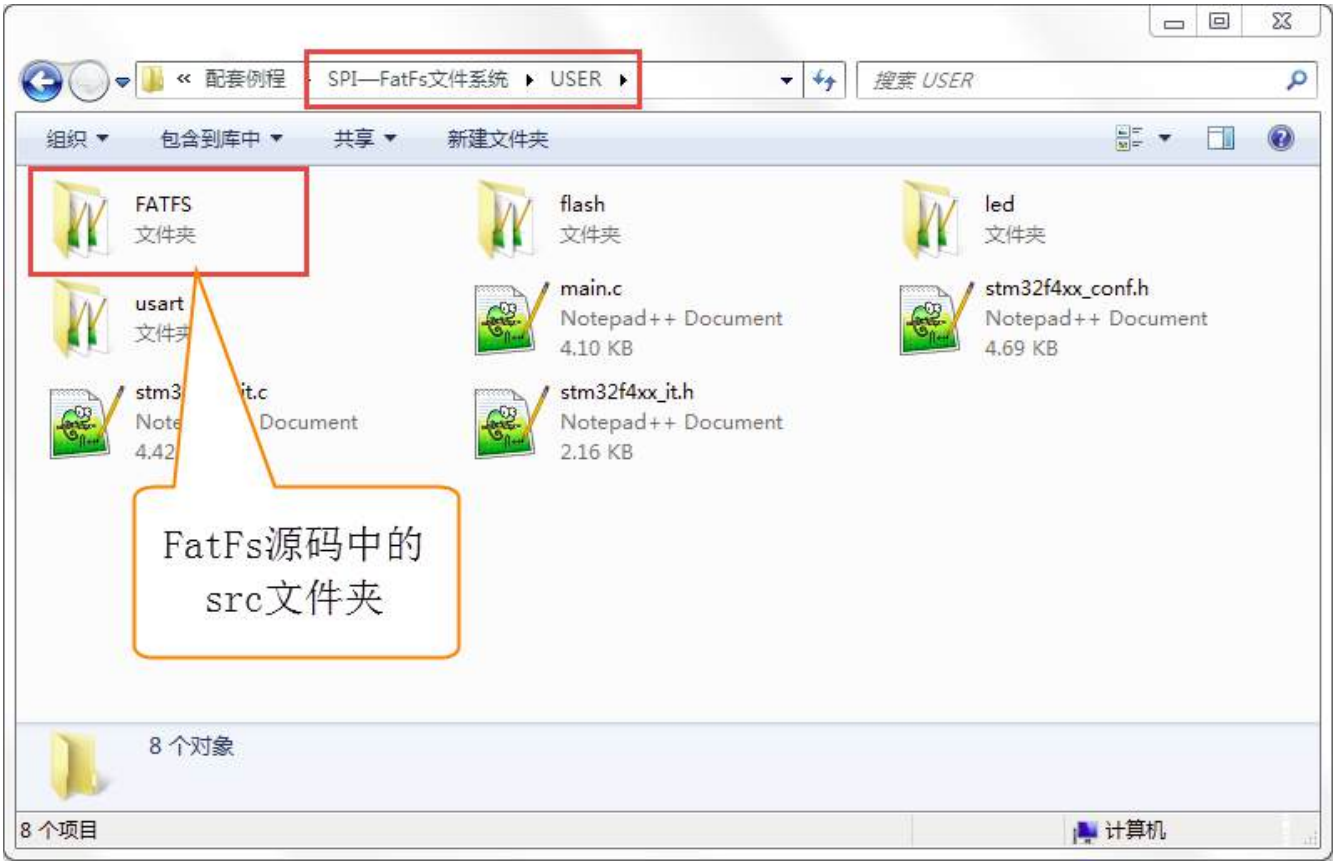


图 25-6 拷贝 FatFs 源码到工程

2) 使用 KEIL 软件打开工程文件 (..SPI—FatFs 文件系统 ProjectRVMDK(uv5)\ BH-F407.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*。

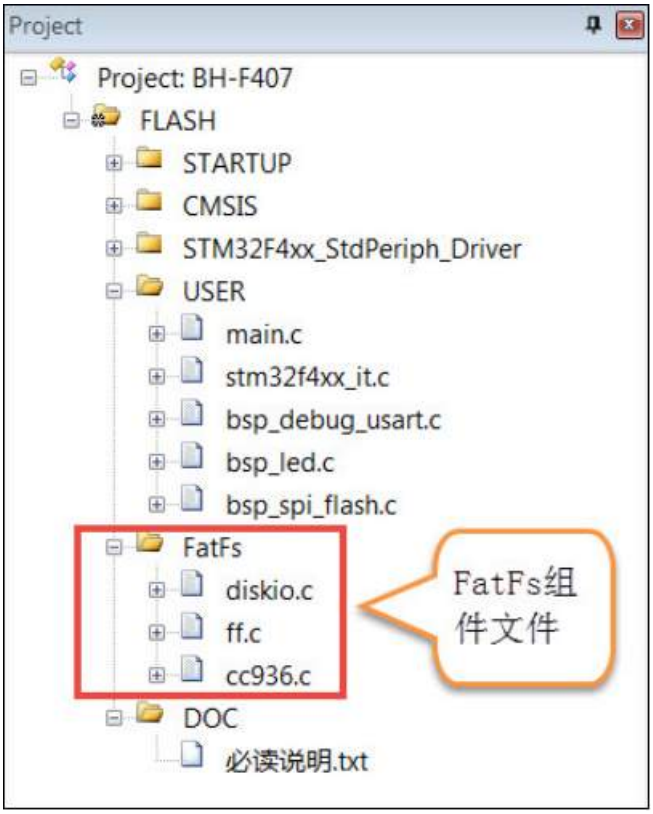


图 25-7 添加 FatFS 文件到工程

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*。

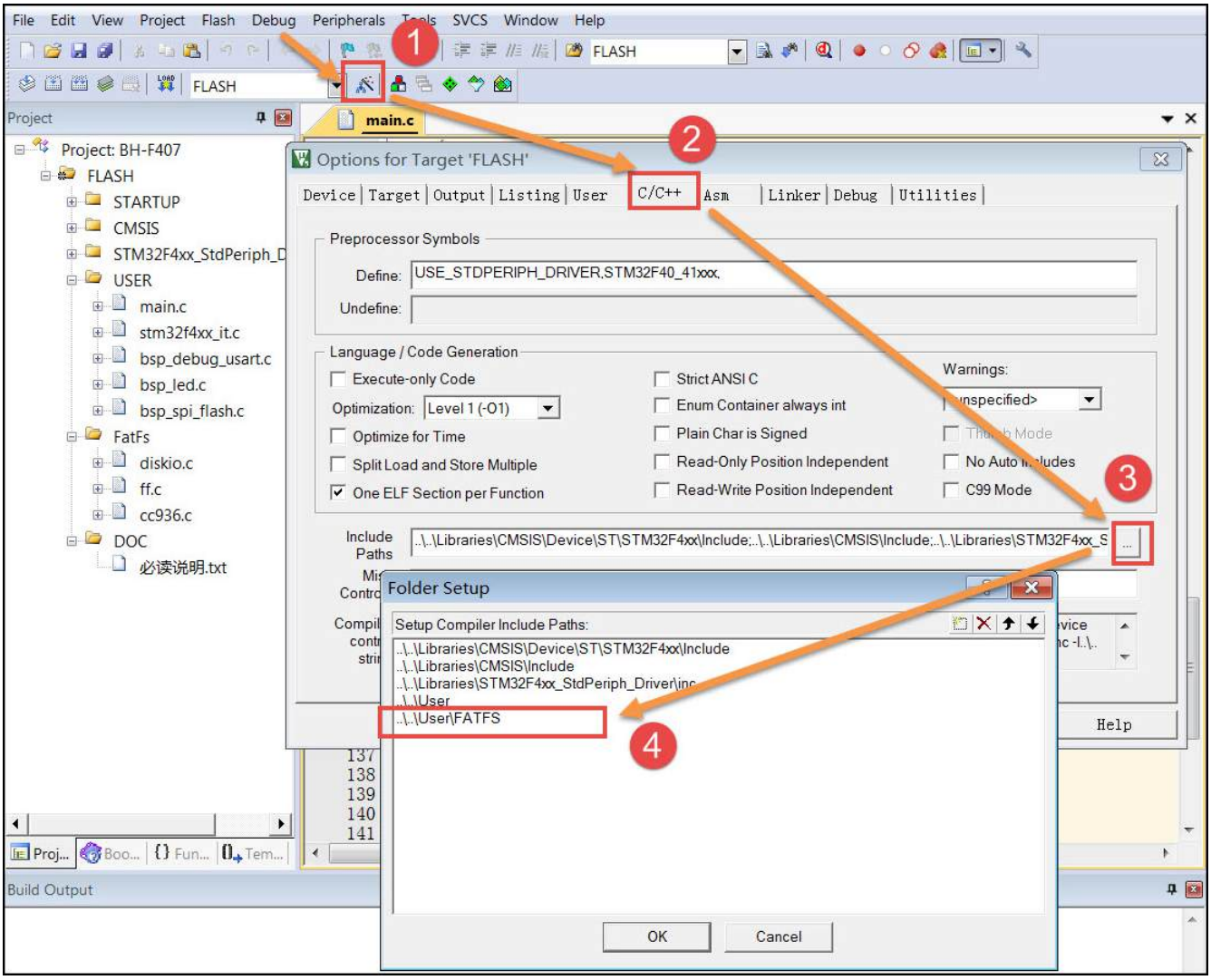


图 25-8 添加 FATFS 路径到工程选项

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。

25.3.4 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。

25.3.4.1 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
```

(下页继续)

(续上页)

```
#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。

25.3.4.2 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 {
            /* 设备 ID 读取结果错误 */
            status = STA_NOINIT;;
        }
        break;
```

(下页继续)

(续上页)

```
    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。

25.3.4.3 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;

    case SPI_FLASH:     /* SPI Flash */
        /* 初始化 SPI Flash */
        SPI_FLASH_Init();
        /* 延时一小段时间 */
```

(下页继续)

(续上页)

```
        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。

25.3.4.4 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 */
    DWORD sector,   /* 扇区首地址 */
    UINT count      /* 扇区个数(1..128) */
)
{
```

(下页继续)

(续上页)

```
    DRESULT status = RES_PARERR;
    switch (pdrv) {
    case ATA: /* SD CARD */
        break;

    case SPI_FLASH:
        /* 扇区偏移 6MB，外部 Flash 文件系统空间放在 SPI Flash 后面 10MB 空间 */
        sector+=1536;
        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 W25Q128FV，每个扇区大小为 4096 个字节 (4KB)，总共有 16M 字节空间，为兼容后面实验程序，我们只将后部分 10MB 空间分配给 FatFs 使用，前部分 6MB 空间用于其他实验需要，即 FatFs 是从 6MB 空间开始，为实现这个效果需要将所有的读写地址都偏移 1536 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。

25.3.4.5 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 6MB，外部 Flash 文件系统空间放在 SPI Flash 后面 10MB 空间 */
        sector+=1536;
        write_addr = sector<<12;
        SPI_FLASH_SectorErase(write_addr);
        SPI_FLASH_BufferWrite((u8 *)buff,write_addr,count<<12);
        status = RES_OK;
        break;
```

(下页继续)

(续上页)

```
    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。

25.3.4.6 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) {
        /* 扇区数量：2560*4096/1024/1024=10(MB) */
        case GET_SECTOR_COUNT:
            *(DWORD * )buff = 2560;
```

(下页继续)

(续上页)

```
            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。

25.3.4.7 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) 。

25.3.5 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。

25.3.6 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。

25.3.6.1 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 F407 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。

25.3.6.2 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。

25.3.7 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。

25.4 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。

25.4.1 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。

25.4.2 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。

25.4.2.1 FatFs 多项功能测试

列表 11: 代码清单 25-11 FatFs 多项功能测试

```
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 读写测试文件.txt",FA_OPEN_EXISTING|␣
↪FA_WRITE|FA_READ );
    if ( res_flash == FR_OK ) {
        /*  文件定位 */
        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 */
            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");
```

(下页继续)

(续上页)

```
        }
        if (res_flash==FR_OK) {
            /* 重命名并移动文件 */
            res_flash=f_rename("1:FatFs 读写测试文件.txt","1:TestDir/testdir.
↪txt");
        }
    } 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。

25.4.2.2 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)
{
    FILINFO fno;

    /* 获取文件信息 */
    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
```

(下页继续)

(续上页)

```
    }
    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。

25.4.2.3 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);
```

(下页继续)

(续上页)

```
        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
            //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。

25.4.2.4 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");
```

(下页继续)

(续上页)

```
    //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");
    }
    /* 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 fpaht 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。

25.4.3 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 26 章　FSMC—扩展外部 SRAM

本章参考资料：《STM32F4XX-参考手册》FSMC 章节。

关于 SRAM 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《IS62WV51216》来了解。

26.1 SRAM 控制原理

STM32 控制器芯片内部有一定大小的 SRAM 及 FLASH 作为内存和程序存储空间，但当程序较大，内存和程序空间不足时，就需要在 STM32 芯片的外部扩展存储器了。

扩展内存时一般使用 SRAM 和 SDRAM 存储器，但 STM32F407 系列的芯片不支持扩展 SDRAM(STM32F429 系列支持)，它仅支持使用 FSMC 外设扩展 SRAM，我们以 SRAM 为例讲解如何为 STM32 扩展内存。

给 STM32 芯片扩展内存与给 PC 扩展内存的原理是一样的，只是 PC 上一般以内存条的形式扩展，内存条实质是由多个内存颗粒 (即 SRAM 芯片) 组成的通用标准模块，而 STM32 直接与 SRAM 芯片连接。见图 *26_2*，这是我们实验板上使用的型号为 IS62WV51216 的 SRAM 芯片内部结构框图，以它为模型进行学习。

部分批次 SRAM 芯片替换为 AS6C8016 型号时序与 ISSI 的芯片兼容

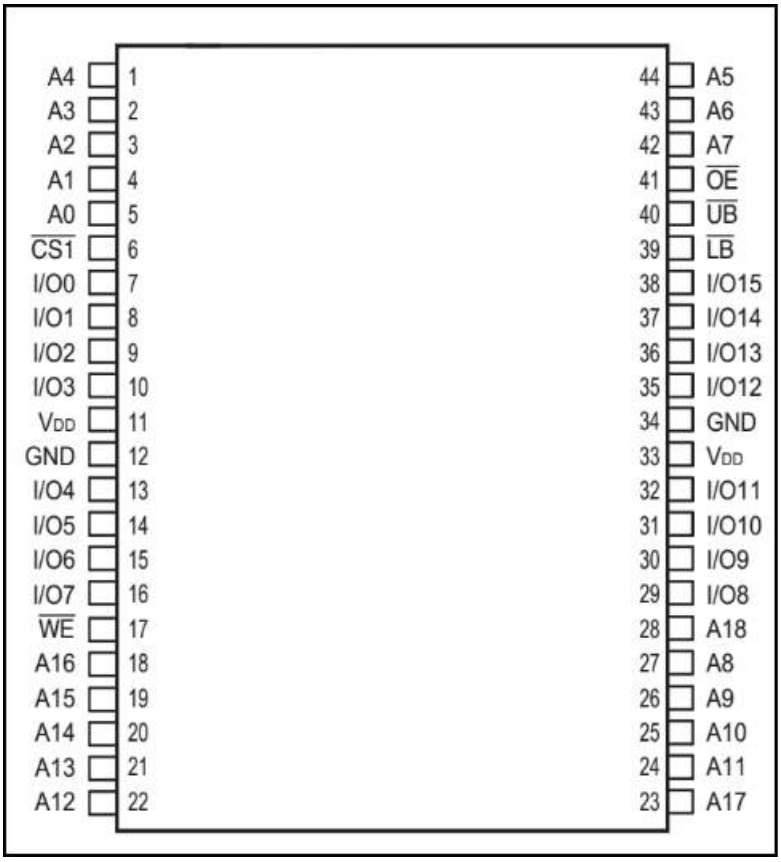


图 26-1 SRAM 芯片外观

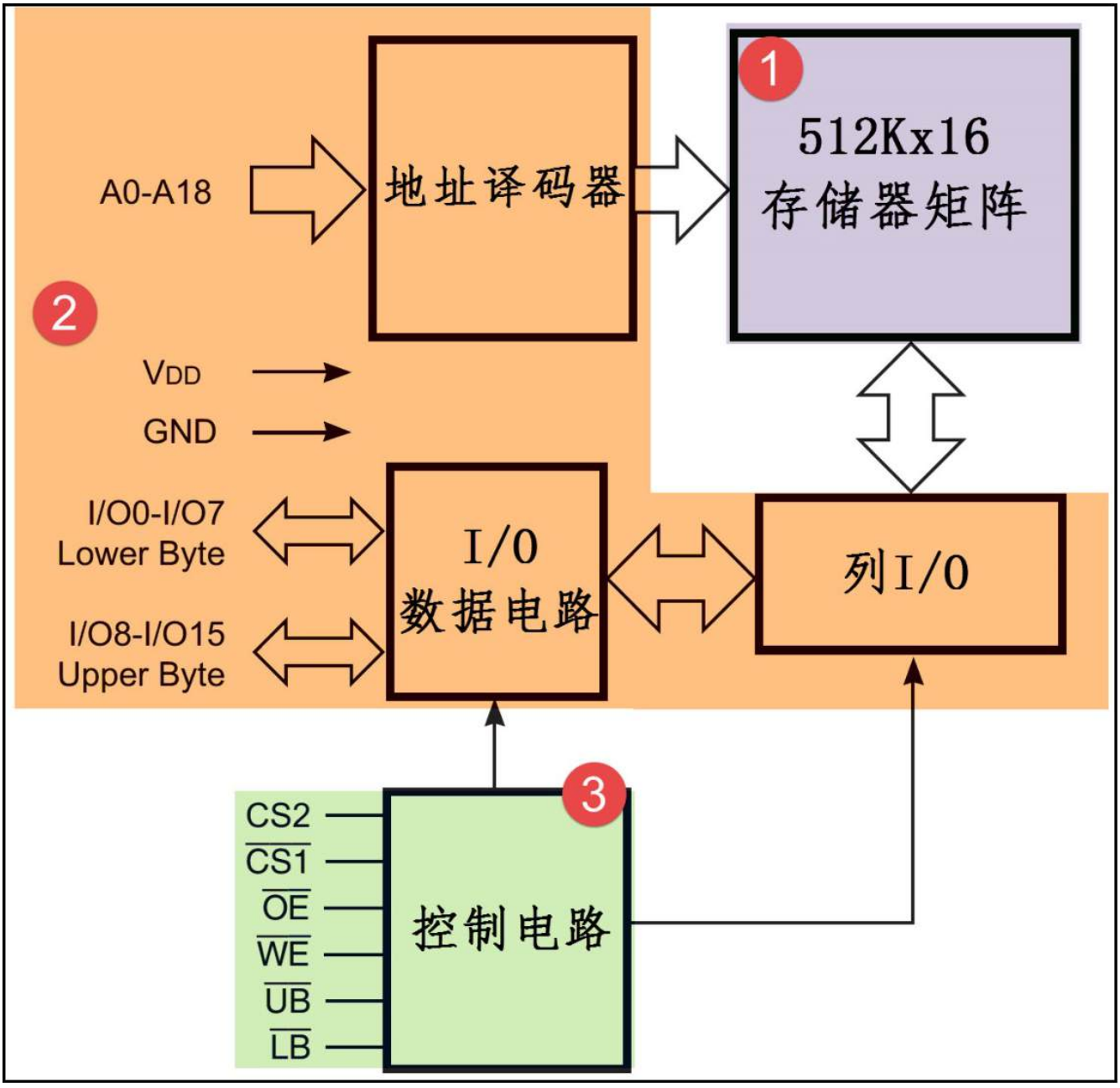


图 26-2 一种 SRAM 芯片的内部结构框图

26.1.1 SRAM 信号线

图 26-2 中左侧引出的是 SRAM 芯片的控制引脚，其说明见表 26-1。

表 26-1 SRAM 控制引脚说明

信号线	类型	说明
A0-A18	I	地址输入
I/O0-I/O7	I/O	数据输入输出信号，低字节
I/O8-I/O15	I/O	数据输入输出信号，高字节
CS 和 CS1#	I	片选信号，CS2 高电平有效，CS1# 低电平有效，部分芯片只有其中一个引脚
OE#	I	输出使能信号，低电平有效
WE#	I	写入使能，低电平有效
UB#	I	数据掩码信号 Upper Byte，高位字节允许访问，低电平有效
LB#	I	数据掩码信号 Lower Byte，低位字节允许访问，低电平有效

SRAM 的控制比较简单，只要控制信号线使能了访问，从地址线输入要访问的地址，即可从 I/O 数据线写入或读出数据。

26.1.2 存储器矩阵

框图中标号处表示的是存储器矩阵，这个 SRAM 芯片的空间大小为 512Kx16(bits)，见图 *26_3*。

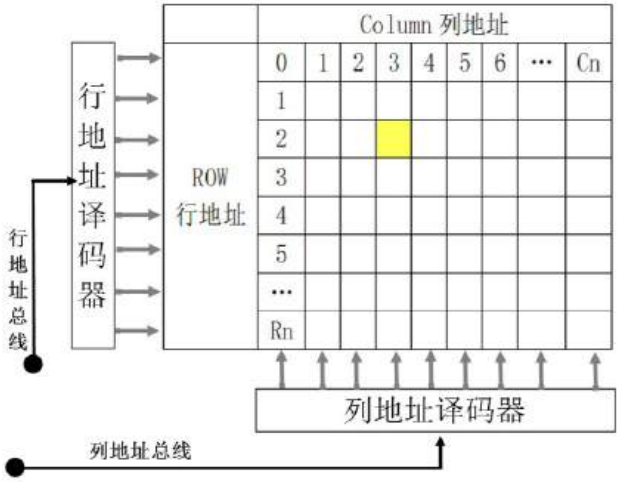


图 26-3 SRAM 存储阵列模型

SRAM 内部包含的存储阵列，可以把它理解成一张表格，数据就填在这张表格上。和表格查找一样，指定一个行地址和列地址，就可以精确地找到目标单元格，这是 SRAM 芯片寻址的基本原理。这样的每个单元格被称为存储单元，而这样的表则被称为存储矩阵。

26.1.3 地址译码器、列 I/O 及 I/O 数据电路

地址译码器把 N 根地址线转换成 2^N 根信号线，每根信号线对应一行或一列存储单元，通过地址线找到具体的存储单元，实现寻址。如果存储阵列比较大，地址线会分成行和列地址，或者行、列分时复用同一地址总线，访问数据寻址时先用地址线传输行地址再传输列地址。本实例中的 SRAM 比较小，没有列地址线，它的数据宽度为 16 位，即一个行地址对应 2 字节空间，框图中左侧的 A0-A18 是行址信号，18 根地址线一共可以表示 $2^{18}=2^8$x1024=512K 行存储单元，所以它一共能访问 512Kx16bits 大小的空间。访问时，使用 UB# 或 LB# 线控制数据宽度，例如，当要访问宽度为 16 位的数据时，使用行地址线指出地址，然后把 UB# 和 LB# 线都设置为低电平，那么 I/O0-I/O15 线都有效，它们一起输出该地址的 16 位数据 (或者接收 16 位数据到该地址)；当要访问宽度为 8 位的数据时，使用行地址线指出地址，然后把 UB# 或 LB# 其中一个设置为低电平，I/O 会对应输出该地址的高 8 位和低 8 位数据，因此它们被称为数据掩码信号。

26.1.4 控制电路

控制电路主要包含了片选、读写使能以及上面提到的宽度控制信号 UB# 和 LB#。利用 CS2 或 CS1# 片选信号，可以把多个 SRAM 芯片组成一个大容量的内存条。OE# 和 WE# 可以控制读写使能，防止误操作。

26.1.5 SRAM 的读写流程

对 SRAM 进行读写数据时，它各个信号线的时序流程见图 *26_4* 及图 *26_5*。

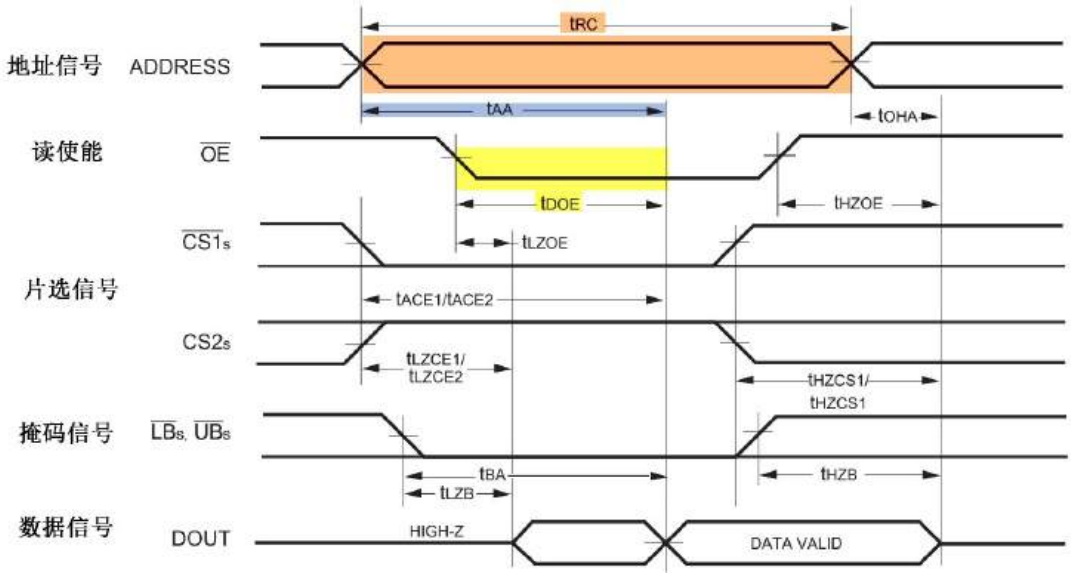


图 26-4 SRAM 的读时序

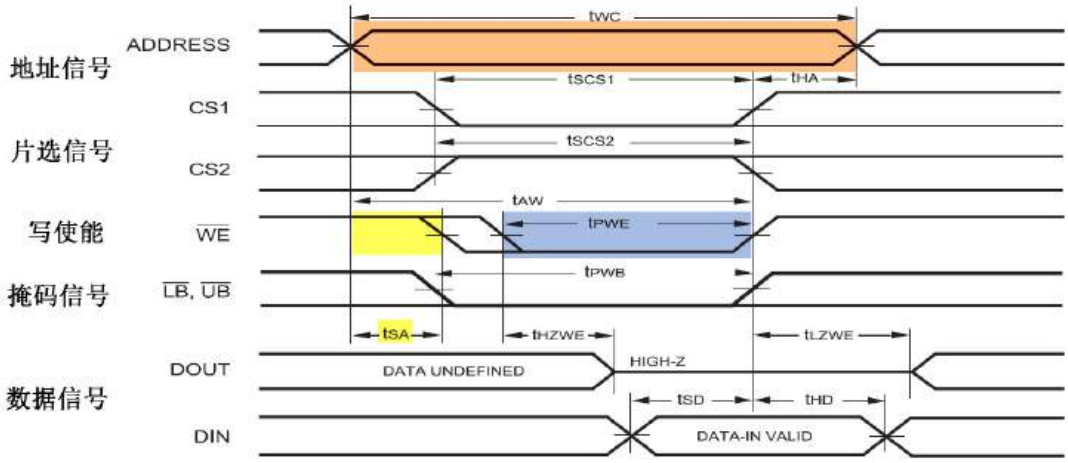


图 26-5 SRAM 的写时序

读写时序的流程很类似，下面我们统一解说：

(1) 主机使用地址信号线发出要访问的存储器目标地址；

(2) 控制片选信号 CS1# 及 CS2# 使能存储器芯片；

(3) 若是要进行读操作，则控制读使能信号 OE# 表示要读数据，若进行写操作则控制写使能信号 WE# 表示要写数据；

(4) 使用掩码信号 LB# 与 UB# 指示要访问目标地址的高、低字节部分；

(5) 若是读取过程，存储器会通过数据线向主机输出目标数据，若是写入过程，主要使用数据线向存储器传输目标数据。

在读写时序中，有几个比较重要的时间参数，在使用 STM32 控制的时候需要参考，它们的介绍见表 26-2。

表 26-2 IS62WV51216BLL-55ns 型号 SRAM 的时间参数

时间参数	IS62WV51216BLL-55ns 型号的时间要求	说明
t_{RC}	不小于 55ns	读操作的总时间
t_{AA}	最迟不大于 55ns	从接收到地址信号到给出有效数据的时间
t_{DOE}	最迟不大于 25ns	从接收到读使能信号到给出有效数据的时间
t_{WC}	不小于 55ns	写操作的总时间
t_{SA}	大于 0ns	从发送地址信号到给出写有使能信号的时间
t_{PWE}	不小于 40ns	从接收到写使能信号到数据采样的时间

26.2 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。

26.3 FSMC 框图剖析

STM32 的 FSMC 外设内部结构见图 *26_6*。

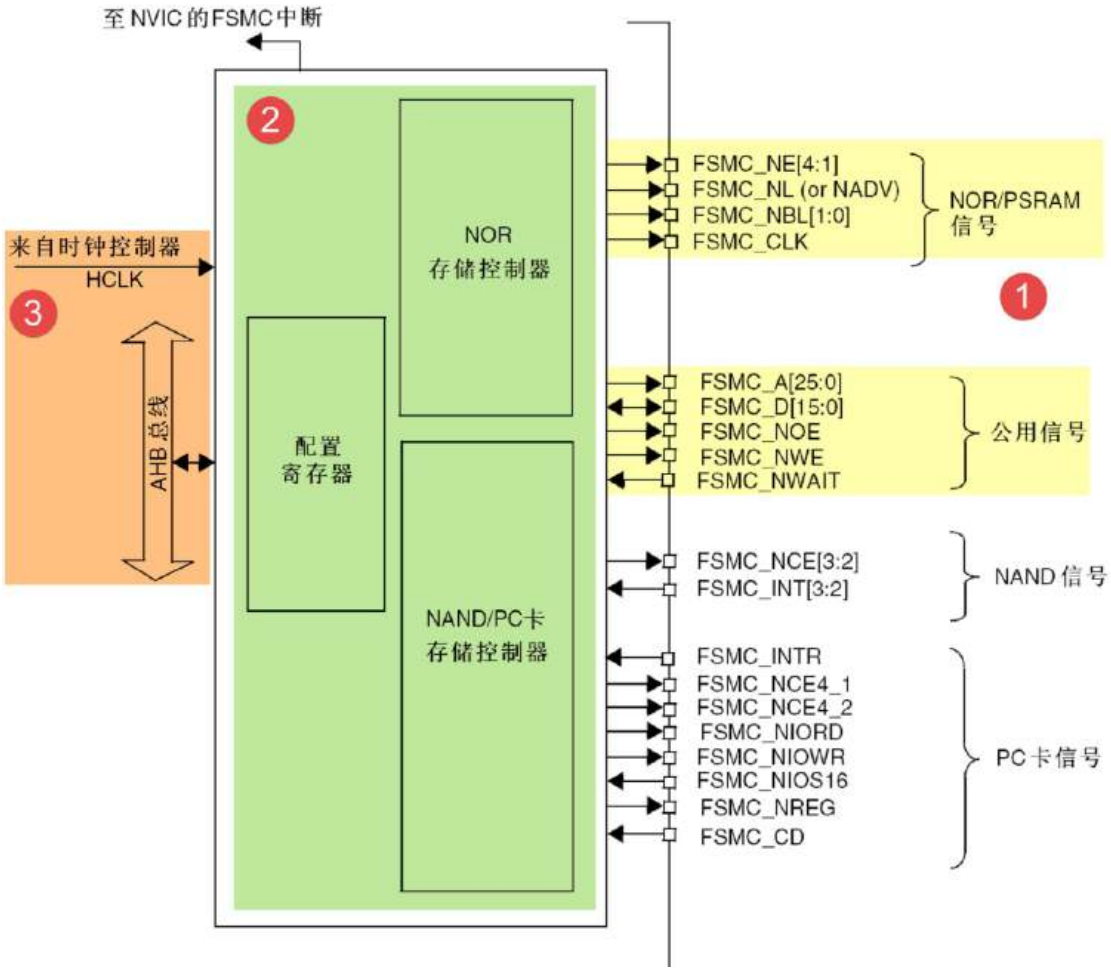


图 26-6 FSMC 控制器框图

26.3.1 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。针对本示例中的 SRAM 控制器，我们整理出以下的 FSMC 与 SRAM 引脚对照表 26-3。

表 26-3 FSMC 中的 SRAM 控制信号线

FSMC 引脚名称	对应 SRAM 引脚名	说明
FSMC_NBL[1:0]	LB#、UB#	数据掩码信号
FSMC_A[18:0]	A[18:0]	行地址线
FSMC_D[15:0]	I/O[15:0]	数据线
FSMC_NWE	WE#	写入使能
FSMC_NOE	OE#	输出使能 (读使能)
FSMC_NE[1:4]	CE#	片选信号

其中比较特殊的 FSMC_NE 是用于控制 SRAM 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 0x6C000000-0x6FFFFFFF 地址空间时，FSMC_NE3 引脚会自动设置为低电平，由于它连接到 SRAM 的 CE# 引脚，所以 SRAM 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 SRAM 的地址不一样，从而达到控制多块 SRAM 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。

26.3.2 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 SRAM 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。

26.3.3 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 168MHz)，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 SRAM 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 SRAM 通讯时，同步时钟最高频率为 84MHz。本示例中的 SRAM 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。

26.4 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 STM32 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。FSMC 的地址映射见见图 *26_7*。

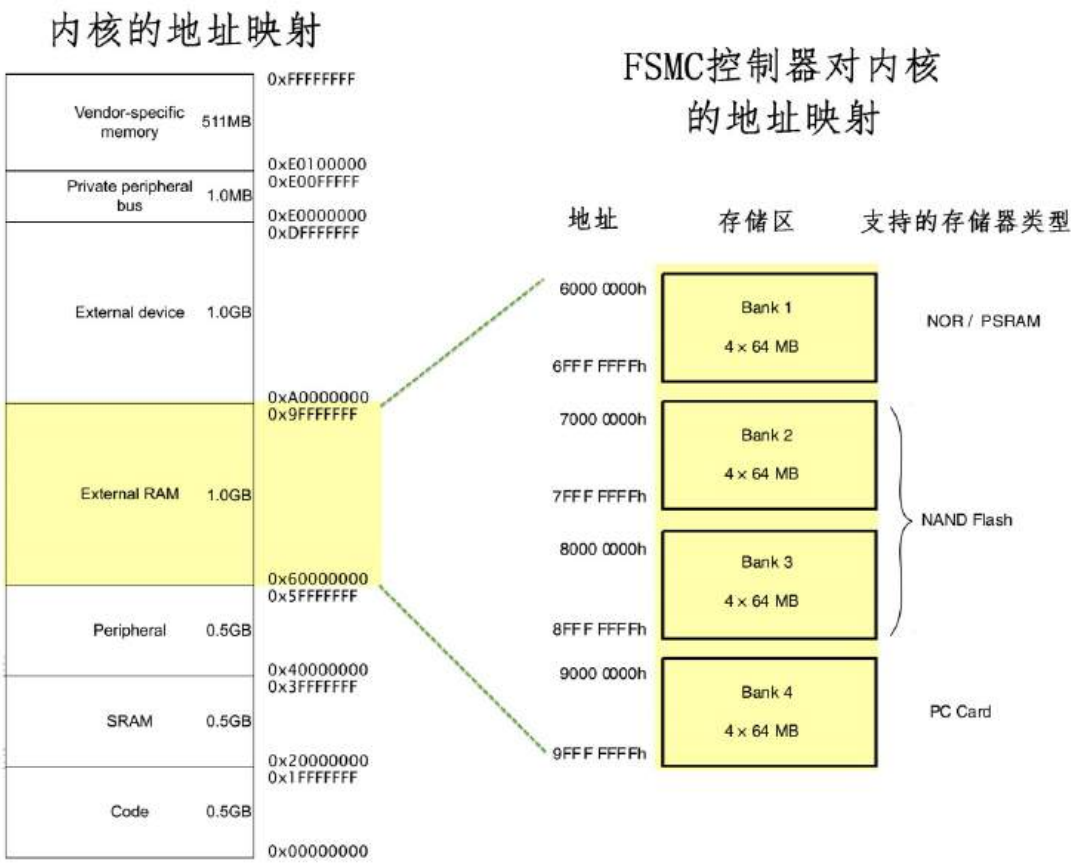


图 26-7 FSMC 的地址映射

图中左侧的是 Cortex-M4 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 *26_8*，当 STM32 访问 0x6C000000-0x6FFFFFFF 地址空间时，会访问到 Bank1 的第 3 小块区域，相应的 FSMC_NE3 信号线会输出控制信号。

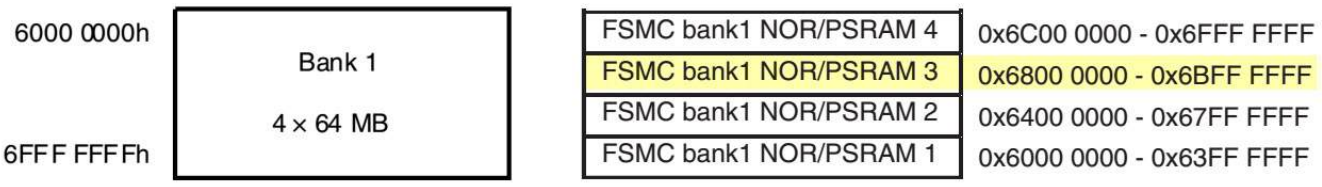


图 26-8 Bank1 内部的小块地址分配

26.5 FSMC 控制 SRAM 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 SRAM 使用的模式 A 进行讲解，见图 *26_9* 及图 *26_10*。

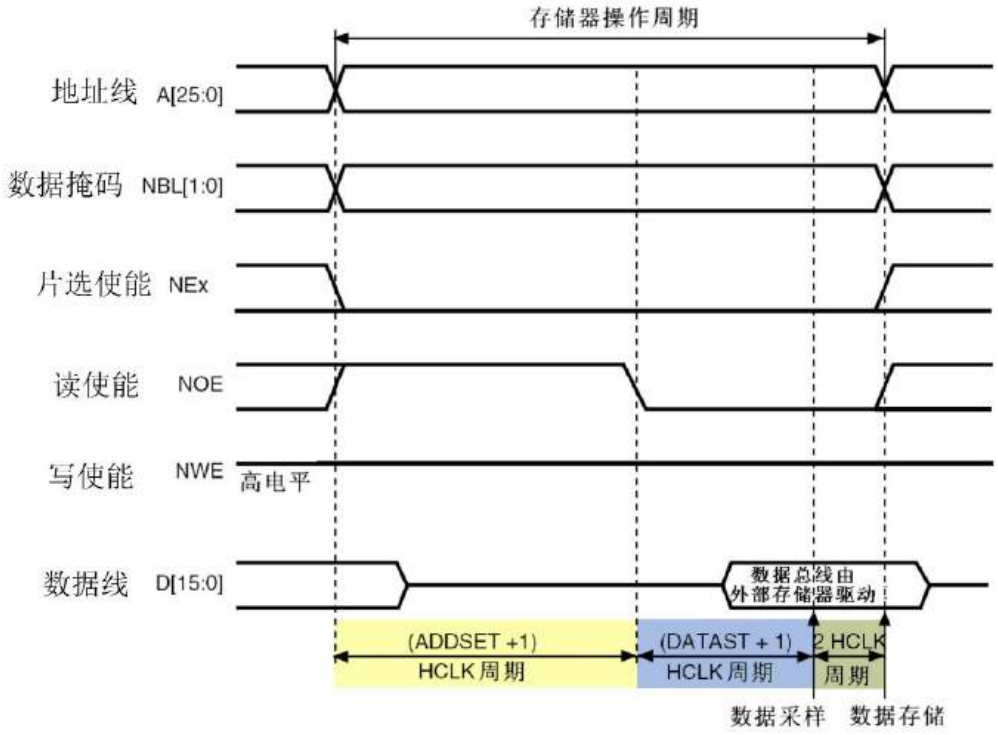


图 26-9 FSMC 模式 A 的读时序

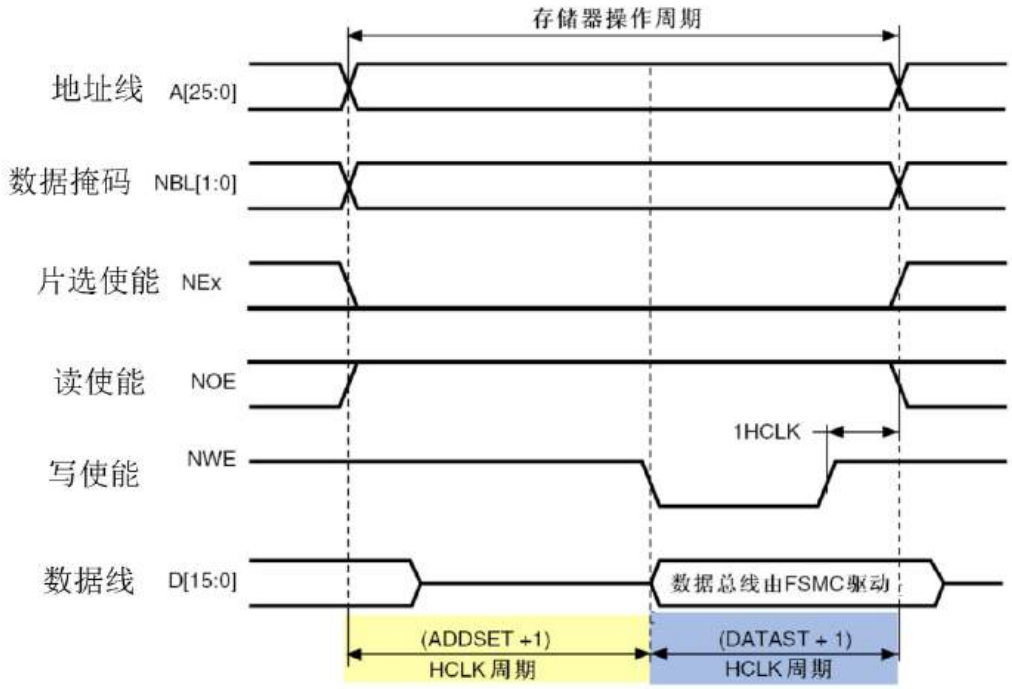


图 26-10 FSMC 模式 A 的写时序

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 SRAM 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。

26.6 SRAM 时序结构体

控制 FSMC 使用 SRAM 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 SRAM 时序结构体以及初始化结构体可以很方便地写入参数。

SRAM 时序结构体的成员见*代码清单 26_0_1*。

列表 1: 代码清单 26-1 SRAM 时序结构体 FSMC_NORSRAMTimingInitTypeDef

```
typedef struct
{
   uint32_t FSMC_AddressSetupTime;        /* 地址建立时间，0-0xF 个 HCLK 周期 */
   uint32_t FSMC_AddressHoldTime;         /* 地址保持时间，0-0xF 个 HCLK 周期 */
   uint32_t FSMC_DataSetupTime;            /* 地址建立时间，0-0xF 个 HCLK 周期
*/
   uint32_t FSMC_BusTurnAroundDuration;/* 总线转换周期,0-0xF 个 HCLK 周期，在
NOR FLASH */
   uint32_t FSMC_CLKDivision;/* 时钟分频因子,1-0xF，若控制异步存储器，本参数无效 */
   uint32_t FSMC_DataLatency;     /* 数据延迟时间，若控制异步存储器，本参数无效 */
   uint32_t FSMC_AccessMode;              /* 设置访问模式 */
}FSMC_NORSRAMTimingInitTypeDef;
```

这个结构体成员定义的都是 SRAM 读写时序中的各项时间参数，这些成员的的参数都与 FSMC_BRT 及 FSMC_BWTR 寄存器配置对应，各个成员介绍如下：

(1) FSMC_AddressSetupTime

本成员设置地址建立时间，即 FSMC 读写时序图 *26_9* 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 168MHz，即一个 HCLK 周期为 1/168 微秒。

(2) FSMC_AddressHoldTime

本成员设置地址保持时间，它可以被设置为 0-0xF 个 HCLK 周期数。

(3) FSMC_DataSetupTime

本成员设置数据建立时间，即 FSMC 读写时序图 *26_10* 中的 DATAST 值，它可以被设置为 0-0xF 个 HCLK 周期数。

(4) FSMC_BusTurnAroundDuration

本成员设置总线转换周期，在 NOR FLASH 存储器中，地址线与数据线可以分时复用，总线转换周期就是指总线在这两种状态间切换需要的延时，防止冲突。控制其它存储器时这个参数无效，配置为 0 即可。

(5) FSMC_CLKDivision

本成员用于设置时钟分频，它以 HCLK 时钟作为输入，经过 FSMC_CLKDivision 分频后输出到 FSMC_CLK 引脚作为通讯使用的同步时钟。控制其它异步通讯的存储器时这个参数无效，配置为 0 即可。

(6) FSMC_DataLatency

本成员设置数据保持时间，它表示在读取第一个数据之前要等待的周期数，该周期指同步时钟的周期，本参数仅用于同步 NOR FLASH 类型的存储器，控制其它类型的存储器时，本参数无效。

(7) FSMC_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。一般来说控制 SRAM 时使用 A 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 SRAM 初始化结构体的一个成员。

26.7 SRAM 初始化结构体

FSMC 的 SRAM 初始化结构体见代码清单 26_0_2。

列表 2: 代码清单 26-2 SRAM 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 */
   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;
```

(下页继续)

(续上页)

```
} FSMC_NORSRAM_InitTypeDef;
```

这个结构体，除最后两个成员是上一小节讲解的时序配置外，其它结构体成员的配置都对应到 FSMC_BCR 中的寄存器位。各个成员意义介绍如下，括号中的是 STM32HAL 库定义的宏：

(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表 26-4。

表 26-4 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) DataAddressMux

本成员用于设置地址总线与数据总线是否复用 (FSMC_DATA_ADDRESS_MUX_DISABLE/DISABLE)，在控制 NOR FLASH 时，可以地址总线与数据总线可以分时复用，以减少使用 STM32 信号线的数量。

(3) 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(FSMC_MEMORY_TYPE_SRAM/PSRAM/NOR)。

(4) 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 (FSMC_NORSRAM_MEM_BUS_WIDTH_8/16/32)。

(5) BurstAccessMode

本成员用于设置是否使用突发访问模式 (FSMC_BURST_ACCESS_MODE_DISABLE /DISABLE)，突发访问模式是指发送一个地址后连续访问多个数据，非突发模式下每访问一个数据都需要输入一个地址，仅在控制同步类型的存储器时才能使用突发模式。

(6) AsynchronousWait

本成员用于设置是否使能在同步传输时使用的等待信号 (FSMC_ASYNCHRONOUS_WAIT_DISABLE /DISABLE)，在控制同步类型的 NOR 或 PSRAM 时，存储器可以使用 FSMC_NWAIT 引脚通知 STM32 需要等待。

(7) WaitSignalPolarity

本成员用于设置等待信号的有效极性，即要求等待时，使用高电平还是低电平 (FSMC_WAIT_SIGNAL_POLARITY_LOW /HIGH)。

(8) FSMC_WrapMode

本成员用于设置是否支持把非对齐的 AHB 突发操作分割成 2 次线性操作 (FSMC_WRAP_MODE_DISABLE /ENABLE)，该配置仅在突发模式下有效。

(9) WaitSignalActive

本成员用于配置在突发传输模式时，决定存储器是在等待状态之前的一个数据周期有效还是在等待状态期间有效 (FSMC_WAIT_TIMING_BEFORE_WS / FSMC_WAIT_TIMING_DURING_WS)。

(10) WriteOperation

这个成员用于设置是否写使能 (FSMC_WRITE_OPERATION_DISABLE /ENABLE)，禁止写使能的话 FSMC 只能从存储器中读取数据，不能写入。

(11) WaitSignal

本成员用于设置当存储器处于突发传输模式时，是否允许通过 NWAIT 信号插入等待状态 (FSMC_WAIT_SIGNAL_ENABLE /DISABLE)。

(12) ExtendedMode

本成员用于设置是否使用扩展模式 (FSMC_EXTENDED_MODE_ENABLE /DISABLE)，

在非扩展模式下，对存储器读写的时序都只使用 FSMC_BCR 寄存器中的配置，即下面的 FSMC_ReadWriteTimingStruct 结构体成员；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，即下面的 FSMC_WriteTimingStruct 结构体。

对本结构体赋值完成后，调用 FSMC_NORSRAMInit 库函数即可把配置参数写入到 FSMC_BCR 及 FSMC_BTR/BWTR 寄存器中。

26.8 FSMC—扩展外部 SRAM 实验

本小节以型号为“IS62WV51216”的 SRAM 芯片为 STM32 扩展内存。它的地址线宽度为 19 位，数据线宽度为 16 位，容量大小为 1MB。

学习本小节内容时，请打开配套的“FSMC—外部 SRAM”工程配合阅读。本实验仅讲解基本的外部 SRAM 驱动，不涉及内存管理的内容，在本书的《MDK 编译过程及文件类型全解》章节将会讲解使用更简单的方法从外部 SRAM 中分配变量，以及使用 C 语言 HAL 库的 malloc 函数来分配外部 SRAM 的空间。

26.8.1 硬件设计

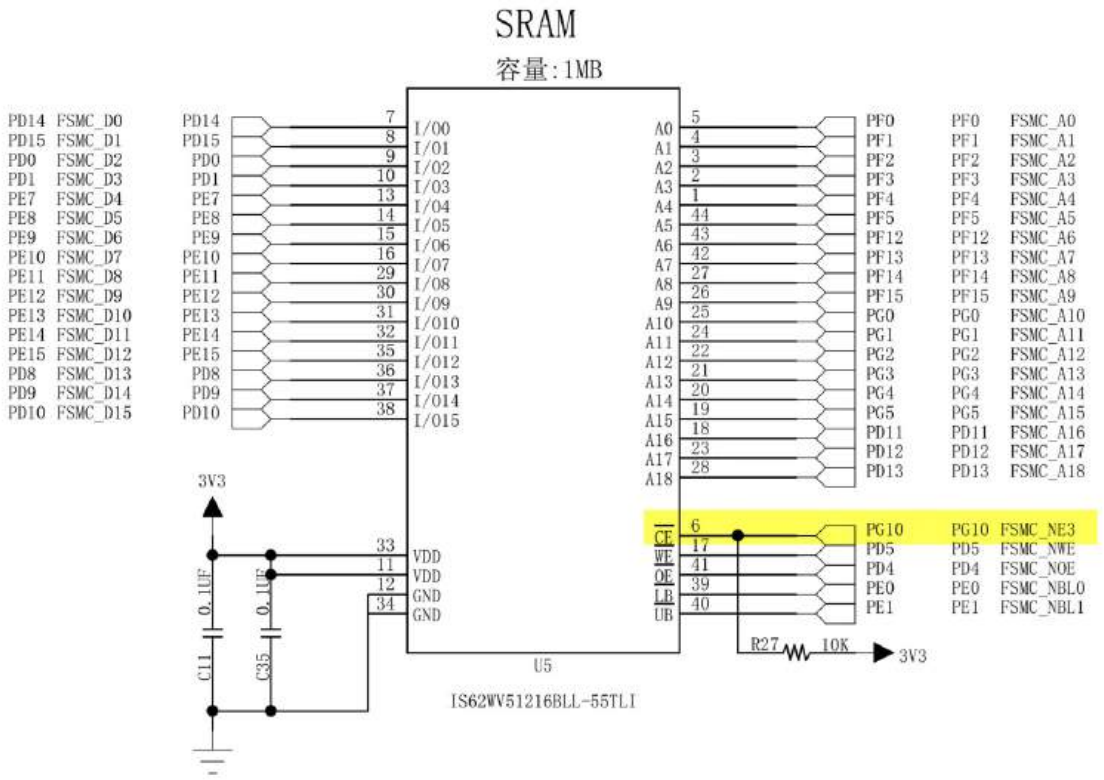


图 26-11 外部 SRAM 硬件连接图

外部 SRAM 芯片与 STM32 相连的引脚非常多，主要是地址线和数据线，这些具有特定 FSMC 功能的 GPIO 引脚可查询《STM32F4xx 规格书》中的说明来了解。

关于该 SRAM 芯片的更多信息，请参考其规格书《IS62WV51216》了解。若您使用的实验板 SRAM 的型号或控制引脚不一样，可在我们工程的基础上修改，程序的控制原理相同。

根据本硬件设计，SRAM 芯片的使能信号与 FSMC_NE4 连接，所以它会被映射到 STM32 中的 BANK1 NOR/SRAM 4 区域，该区域的地址范围为 0x6C000000-0x6FFFFFFF，因此，当内核访问从基地址 0x6C000000 开始的 1MB 空间时，FSMC 外设会自动控制原理图中的引脚产生访问时序，访问这个外部 SRAM 存储器。

26.8.2 软件设计

为了使工程更加有条理，我们把 SRAM 初始化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_sram.c”及“bsp_sram.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。

26.8.2.1 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 SRAM 的时序、工作模式；

(4) 建立机制访问外部 SRAM 存储器；

(5) 编写测试程序，对读写数据进行校验。

26.8.2.2 代码分析

26.8.2.2.1 FSMC 硬件相关宏定义

我们把 FSMC SRAM 硬件相关的配置都以宏的形式定义到“sram.h”文件中，见*代码清单 26_0_3*。

列表 3: 代码清单 26-3 SRAM 硬件配置相关的宏 (省略了大部分数据线，sram.h 文件)

```
/* 地址信号线 */
#define FMC_A0_GPIO_PORT          GPIOF
#define FMC_A0_GPIO_CLK()         __GPIOF_CLK_ENABLE()
#define FMC_A0_GPIO_PIN           GPIO_PIN_0
/* 省略一些引脚 */
#define FMC_A11_GPIO_PORT         GPIOG
#define FMC_A11_GPIO_CLK()        __GPIOG_CLK_ENABLE()
```

(下页继续)

(续上页)

```
#define FMC_A11_GPIO_PIN          GPIO_PIN_2

/* 数据信号线 */
#define FMC_D0_GPIO_PORT          GPIOD
#define FMC_D0_GPIO_CLK()         __GPIOD_CLK_ENABLE()
#define FMC_D0_GPIO_PIN           GPIO_PIN_14

/* 省略一些引脚 */
#define FMC_D15_GPIO_PORT         GPIOD
#define FMC_D15_GPIO_CLK()        __GPIOD_CLK_ENABLE()
#define FMC_D15_GPIO_PIN          GPIO_PIN_10

/* 控制信号线 */
#define FMC_CS_GPIO_PORT          GPIOH
#define FMC_CS_GPIO_CLK()         __GPIOH_CLK_ENABLE()
#define FMC_CS_GPIO_PIN           GPIO_PIN_6

#define FMC_BA0_GPIO_PORT         GPIOG
#define FMC_BA0_GPIO_CLK()        __GPIOG_CLK_ENABLE()
#define FMC_BA0_GPIO_PIN          GPIO_PIN_4

#define FMC_BA1_GPIO_PORT         GPIOG
#define FMC_BA1_GPIO_CLK()        __GPIOG_CLK_ENABLE()
#define FMC_BA1_GPIO_PIN          GPIO_PIN_5

#define FMC_WE_GPIO_PORT          GPIOC
#define FMC_WE_GPIO_CLK()         __GPIOC_CLK_ENABLE()
#define FMC_WE_GPIO_PIN           GPIO_PIN_0

#define FMC_RAS_GPIO_PORT         GPIOF
#define FMC_RAS_GPIO_CLK()        __GPIOF_CLK_ENABLE()
#define FMC_RAS_GPIO_PIN          GPIO_PIN_11
```

(下页继续)

(续上页)

```
#define FMC_CAS_GPIO_PORT          GPIOG
#define FMC_CAS_GPIO_CLK()         __GPIOG_CLK_ENABLE()
#define FMC_CAS_GPIO_PIN           GPIO_PIN_15

#define FMC_CLK_GPIO_PORT          GPIOG
#define FMC_CLK_GPIO_CLK()         __GPIOG_CLK_ENABLE()
#define FMC_CLK_GPIO_PIN           GPIO_PIN_8

#define FMC_CKE_GPIO_PORT          GPIOH
#define FMC_CKE_GPIO_CLK()         __GPIOH_CLK_ENABLE()
#define FMC_CKE_GPIO_PIN           GPIO_PIN_7

/*UDQM LDQM*/
#define FMC_UDQM_GPIO_PORT          GPIOE
#define FMC_UDQM_GPIO_CLK()         __GPIOE_CLK_ENABLE()
#define FMC_UDQM_GPIO_PIN           GPIO_PIN_1

#define FMC_LDQM_GPIO_PORT          GPIOE
#define FMC_LDQM_GPIO_CLK()         __GPIOE_CLK_ENABLE()
#define FMC_LDQM_GPIO_PIN           GPIO_PIN_0
```

以上代码根据硬件的连接，把与 SRAM 通讯使用的引脚端口、引脚号以及时钟都以宏封装起来。其中 FSMC_CS 作为片选引脚对应的是 FSMC_NE3，所以后面我们对 SDRAM 的寻址空间也是要指向存储区域 BANK1 NOR/SRAM 3 的。

26.8.2.2.2 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_0_4*。

列表 4: 代码清单 26-4 FSMC 的 GPIO 初始化函数 (省略了大部分数据线，sram.c 文件)

```
/**
* @brief  初始化控制 SRAM 的 IO
* @param  无
* @retval 无
*/
static void SRAM_GPIO_Config(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;

   /* 此处省略大量地址线、数据线以及控制信号线，
   它们的时钟配置都相同，具体请查看工程中的代码 */
   /* 使能 SDRAM 相关的 GPIO 时钟 */
   /* 地址信号线 */
   FMC_A0_GPIO_CLK();FMC_A1_GPIO_CLK(); FMC_A2_GPIO_CLK();
   /* 数据信号线 */  /* 控制信号线 */
   FMC_UDQM_GPIO_CLK();FMC_LDQM_GPIO_CLK();
   /*-- GPIO 配置 -----------------------------------------------------*/

   /* 通用 GPIO 配置 */
   /*--所有 GPIO 的配置都相同，此处省略大量引脚初始化，具体请查看工程中的代码 */
   GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;//配置为复用功能
   GPIO_InitStructure.Pull = GPIO_PULLUP;
   GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
   GPIO_InitStructure.Alternate = GPIO_AF12_FMC;

   /*A 行列地址信号线 针对引脚配置 */
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin = FMC_A0_GPIO_PIN;
    HAL_GPIO_Init(FMC_A0_GPIO_PORT, &GPIO_InitStructure);

    /*...*/
    /*DQ 数据信号线 针对引脚配置 */
    GPIO_InitStructure.Pin = FMC_D0_GPIO_PIN;
    HAL_GPIO_Init(FMC_D0_GPIO_PORT, &GPIO_InitStructure);

    /*...*/
    /* 控制信号线 */
    GPIO_InitStructure.Pin = FMC_CS_GPIO_PIN;
    HAL_GPIO_Init(FMC_CS_GPIO_PORT, &GPIO_InitStructure);
    /*...*/
}
```

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，以上代码把 FSMC SRAM 的所有信号线全都初始化为复用功能，所有引脚配置都是一样的。

26.8.2.2.3 配置 FSMC 的模式

接下来需要配置 FSMC SRAM 的工作模式，这个函数的主体是根据硬件连接的 SRAM 特性，对时序结构体以及初始化结构体进行赋值。见*代码清单 26_0_5*。

列表 5: 代码清单 26-5 配置 FSMC 的模式 (sram.c 文件)

```
void FSMC_SRAM_Init(void)
{
    SRAM_Handler.Instance = FMC_NORSRAM_DEVICE;
    SRAM_Handler.Extended = FMC_NORSRAM_EXTENDED_DEVICE;

    /* SRAM device configuration */
    Timing.AddressSetupTime      = 0x00;
```

(下页继续)

(续上页)

```
    Timing.AddressHoldTime       = 0x00;
    Timing.DataSetupTime         = 0x08;
    Timing.BusTurnAroundDuration = 0x00;
    Timing.CLKDivision           = 0x00;
    Timing.DataLatency           = 0x00;
    Timing.AccessMode            = FSMC_ACCESS_MODE_A;
    //使用 NE4
    SRAM_Handler.Init.NSBank=FSMC_NORSRAM_BANK4;
    //地址/数据线不复用
    SRAM_Handler.Init.DataAddressMux=FSMC_DATA_ADDRESS_MUX_DISABLE;
    //SRAM
    SRAM_Handler.Init.MemoryType=FSMC_MEMORY_TYPE_SRAM;
    //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周期期间使能 NWAIT
    SRAM_Handler.Init.WaitSignalActive=FSMC_WAIT_TIMING_BEFORE_WS;
    //存储器写使能
    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_Handler.Init.ContinuousClock=FSMC_CONTINUOUS_CLOCK_SYNC_ASYNC;
```

(下页继续)

(续上页)

```
    /* SRAM controller initialization */
    FSMC_SRAM_MspInit();
    HAL_SRAM_Init(& SRAM_Handler, &Timing, &Timing);
}
```

这个函数的执行流程如下：

(1) 初始化 GPIO 引脚以及 FSMC 时钟

函数开头调用了前面定义的 SDRAM_GPIO_Config 函数对 FMC 用到的 GPIO 进行初始化，并且使用库函数 __FMC_CLK_ENABLE 使能 FMC 外设的时钟。

(2) 时序结构体赋值

函数开头调用了前面定义的 SDRAM_GPIO_Config 函数对 FMC 用到的 GPIO 进行初始化，并且使用库函数 __FMC_CLK_ENABLE 使能 FMC 外设的时钟。

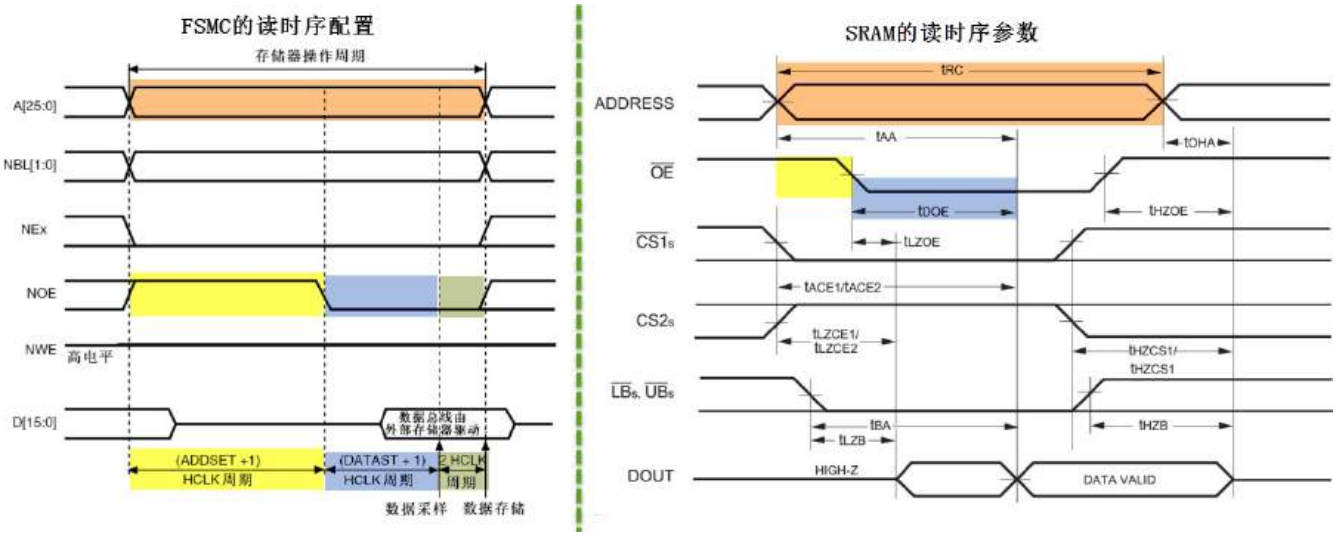


图 26-12 FSMC 时序配置与 SRAM 时序参数要求对比 (读)

表 26-5 SRAM 的读操作参数 (时间要求摘自《IS62WV51216》规格书)

时间参数	SRAM 要求	说明	FSMC 配置要求表达式
t_{RC}	不小于 55ns	读操作周期	ADDSET+1+DATAST+1+2 > 55ns
小于 t_{LZCE}	SRAM 无给出要求	从发出地址到给出读使能信号的时间	ADDSET+1 > 0 ns
t_{DOE}	最迟不大于 25ns	从接收到读使能信号至给出有效数据的时间	DATAST+1 > 25 ns

根据 FSMC 配置表达式的配置要求把时间单位 1/72 微秒 (即 1000/72 纳秒) 代入，可求得 ADDSET = 0，DATAST=1 时即可符合要求。如：

t_{RC}=ADDSET+1+DATAST+1+2 =(0+1+8+1+2)*1000/168 = 71.4 ns > 55 ns

t_{DOE}=DATAST+1 = (8+1)*1000/168 = 53.5 > 25 ns

可看出本实验中的配置有充足的裕量，裕量较大，可确保访问正确，但会导致访问速度变慢，可根据实际需要进行测试调整，保证访问正确的前提下可提高访问速度。不过还需要注意本实验的读时序配置与写时序是一致的，修改时还要确保写时序正常，下面再来列出写时序的计算过程：

由图 26-13 的 FSMC 时序和 SRAM 时序对比及 SRAM 时间参数要求可总结出表 26-5 最右侧的计算表达式。

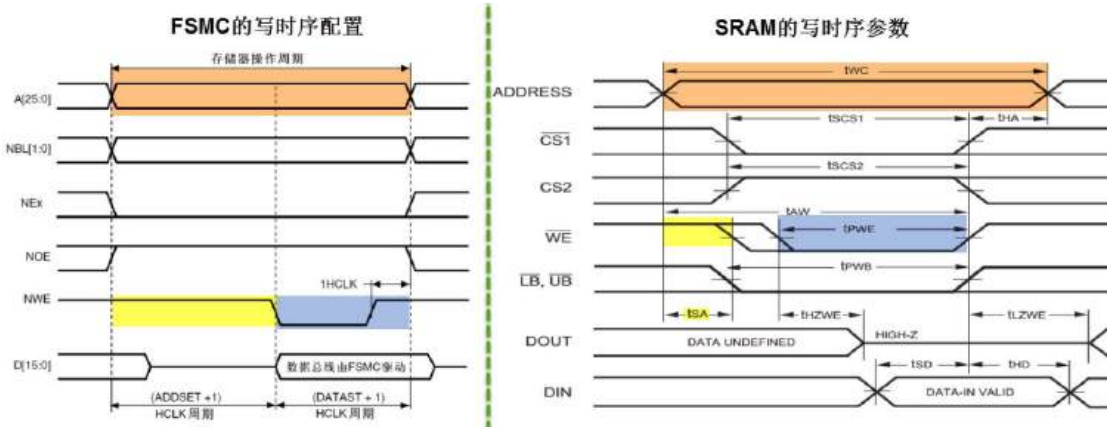


图 26-13 FSMC 时序配置与 SRAM 时序参数要求对比 (写)

表 26-6 SRAM 的写操作参数 (时间要求摘自《IS62WV51216》规格书)

时间参数	SRAM 要求	说明	FSMC 配置要求表达式
t_{WC}	大于 55ns	写操作周期	ADDSET+1+DATAST+1 > 55ns
t_{SA}	无要求	地址建立时间	ADDSET+1 > 0 ns
t_{PWB}	最早最不小于 40ns	从接收到写使能信号到对数据采样的时间	DATAST+1 > 40 ns

根据 FSMC 配置表达式的配置要求把时间单位 1/72 微秒 (即 1000/72 纳秒) 代入，可求得 ADDSET = 0，DATAST=2 时即可符合要求。如：

t_{WC} = ADDSET+1+DATAST+1 =(0+1+8+1)*1000/168 = 59.5 ns > 55 ns

t_{PWB} = DATAST+1 = (8+1) *1000 / 168 = 53.5 > 40 ns

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，然后再把时序结构体作为指针赋值到下面的 FSMC 初始化结构体中，作为读写的时序参数，最后再调用 FSMC_NORSRAMInit 函数即可把参数写入到相应的寄存器中。

(3) 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，这些是根据外接的 SRAM 的电路设置的。

- 设置存储区域 NSBank

  Bank 成员设置 FSMC 的 SRAM 存储区域映射选择为 FSMC_Bank1_NORSRAM4，这是由于我们的 SRAM 硬件连接到 FSMC_NE4 和 NOR/PSRAM 相关引脚，所以对应到存储区域 Bank1 SRAM4，对应的基地址为 0x6C000000；

- 存储器类型 MemoryType

  由于我们控制的是 SRAM 类型存储器，所以 MemoryType 成员要选择相应的 FSMC_MEMORY_TYPE_SRAM；

- 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 SRAM 控制不相关，都被设置为 Disable 了；

赋值完成后调用库函数 FSMC_NORSRAMInit 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。最后调用 FSMC_NORSRAMCmd 函数使能要控制的存储区域 FSMC_Bank1_NORSRAM4。

26.8.2.2.4 使用指针的方式访问 SRAM 存储器

完成初始化 SRAM 后，我们就可以利用它存储数据了，由于 SRAM 的存储空间是被映射到内核的寻址区域的，我们可以通过映射的地址直接访问 SRAM，访问这些地址时，FSMC 外设自动读写 SRAM，程序上无需额外操作。

通过地址访问内存，最直接的就是使用 C 语言的指针方式，见*代码清单 26_0_6*。

列表 6: 代码清单 26-6 使用指针的方式访问 SRAM

```
/*SRAM 起始地址 存储空间 4 的起始地址 */
#define Bank1_SRAM4_ADDR     ((uint32_t)0x6C000000)
```

(下页继续)

(续上页)

```
/*SRAM 大小，1M 字节 */
#define IS62WV51216_SIZE 0x100000

uint32_t temp;

/* 向 SRAM 写入 8 位数据 */
*( uint8_t*) (Bank1_SRAM4_ADDR ) = (uint8_t)0xAA;
/* 从 SRAM 读取数据 */
temp =  *( uint8_t*) (Bank1_SRAM4_ADDR );

/* 写/读 16 位数据 */
*( uint16_t*) (Bank1_SRAM4_ADDR +10 ) = (uint16_t)0xBBBB;
temp =  *( uint16_t*) (Bank1_SRAM4_ADDR+10 );

/* 写/读 32 位数据 */
*( uint32_t*) (Bank1_SRAM4_ADDR +20 ) = (uint32_t)0xCCCCCCCC;
temp =  *( uint32_t*) (Bank1_SRAM4_ADDR+20 );
```

为方便使用，代码中首先定义了宏 Bank1_SRAM4_ADDR 表示 SRAM 的起始地址，该地址即 FSMC 映射的存储区域 Bank SRAM4 的首地址；宏 IS62WV51216_SIZE 表示 SRAM 的大小，所以从地址 (Bank1_SRAM4_ADDR) 到 (Bank1_SRAM3_ADDR+ IS62WV51216_SIZE) 都表示在 SRAM 的存储空间，访问这些地址，直接就能访问 SRAM。

配合这些宏，使用指针的强制转换以及取指针操作即可读写 SRAM 的数据，使用上跟普通的变量无异。

26.8.2.2.5 直接指定变量存储到 SRAM 空间

每次存取数据都使用指针来访问太麻烦了，为了简化操作，可以直接指定变量存储到 SRAM 空间，见*代码清单 26_0_7*。

列表 7: 代码清单 26-7 直接指定变量地址的方式访问 SRAM

```
/* SRAM 起始地址 存储空间 2 的起始地址 */

#define Bank1_SRAM4_ADDR      ((uint32_t)0x6C000000)

/* 绝对定位方式访问 SRAM, 这种方式必须定义成全局变量 */

uint8_t testValue __attribute__((at(SDRAM_BANK_ADDR)));

testValue = 0xDD;
```

这种方式使用关键字“__attribute__((at()))”来指定变量的地址，代码中指定 testValue 存储到 SRAM 的起始地址，从而实现把变量存储到 SRAM 上。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。

更常见的是利用这种方法定义一个很大的数组，整个数组都指定到 SRAM 地址上，然后就像使用 malloc 函数一样，用户自定义一些内存管理函数，动态地使用 SRAM 的内存，我们在使用 emWin 写 GUI 应用的时候就是这样做的。参考我们配套的“FSMC—外部 SRAM（内存管理）”实验可以了解如何自行管理内存。。

然而，我们更推荐另一种方法，在本书的《MDK 编译过程及文件类型全解》章节将会讲解使用更简单的方法从 SRAM 中分配变量，以及使用 C 语言 HAL 库的 malloc 函数来分配 SRAM 的空间，更有效地进行内存管理。

26.8.2.3 main 函数

最后我们来编写 main 函数，进行 SRAM 芯片读写校验，见*代码清单 26_0_8*。

列表 8: 代码清单 26-8 main 函数

```
/* 绝对定位方式访问 SRAM, 这种方式必须定义成全局变量 */
uint8_t testValue __attribute__((at(Bank1_SRAM4_ADDR)));
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
   /* 系统时钟初始化成 168 MHz */
   SystemClock_Config();
   /* LED 端口初始化 */
   LED_GPIO_Config();
   /* 初始化串口 */
   DEBUG_USART_Config();
   //初始化外部 SRAM
   FSMC_SRAM_Init();
   /* 蓝灯亮, 表示正在读写 SRAM 测试 */
   LED_BLUE;
   /* 对 SRAM 进行读写测试, 检测 SRAM 是否正常 */
   printf ( "\r\n野火外部 SRAM 测试\r\n" );
   if (SRAM_Test()==1) {
   LED_GREEN;//测试正常 绿灯亮
   } else {
   LED_RED;//测试失败 红灯亮
   }

   /* 指针方式访问 SRAM*/
   {
   uint32_t temp;
```

(下页继续)

(续上页)

```
    printf("\r\n指针方式访问 SRAM\r\n");
    /* 向 SRAM 写入 8 位数据 */
    *( uint8_t*) (Bank1_SRAM4_ADDR ) = (uint8_t)0xAA;
    printf("\r\n指针访问 SRAM, 写入数据 0xAA \r\n");
    /* 从 SRAM 读取数据 */
    temp =  *( uint8_t*) (Bank1_SRAM4_ADDR );
    printf("读取数据: 0x%X \r\n",temp);
    /* 写/读 16 位数据 */
    *( uint16_t*) (Bank1_SRAM4_ADDR+10 ) = (uint16_t)0xBBBB;
    printf("指针访问 SRAM, 写入数据 0xBBBB \r\n");
    temp =  *( uint16_t*) (Bank1_SRAM4_ADDR+10 );
    printf("读取数据: 0x%X \r\n",temp);
    /* 写/读 32 位数据 */
    *( uint32_t*) (Bank1_SRAM4_ADDR+20 ) = (uint32_t)0xCCCCCCCC;
    printf("指针访问 SRAM, 写入数据 0xCCCCCCCC \r\n");
    temp =  *( uint32_t*) (Bank1_SRAM4_ADDR+20 );
    printf("读取数据: 0x%X \r\n",temp);
    }
    /* 绝对定位方式访问 SRAM, 这种方式必须定义成全局变量 */
    {
    testValue = 0xDD;
    printf("\r\n绝对定位访问 SRAM, 写入数据 0xDD, 读出数据 0x%X, 变量地址为%X\r\n",
↪testValue,(uint32_t )&testValue);
    }
    while (1) {
    }
}
```

函数中初始化了 LED、串口，接着调用前面定义好的 SRAM_Init 函数初始化 FSMC 及 SRAM，然后调用自定义的测试函数 SRAM_Test 尝试使用 SRAM 存取 8、16 及 32 位数据，并进行读写校验，它就是使用指针的方式存取数据并校验而已，此处不展开。

注意对 SRAM 存储空间的数据操作都要在 FSMC_SRAM_Init 初始化 FSMC 之后，否则数据是无

法正常存储的。

26.8.3 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 SRAM 测试的调试信息。

第 27 章　LCD—液晶显示

本章参考资料：《STM32F4XX-中文参考手册》、《STM32F4xx 数据手册》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

关于开发板配套的液晶屏控制器参数可查阅《NT35510-Data Sheet.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。

27.1 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。

27.1.1 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。

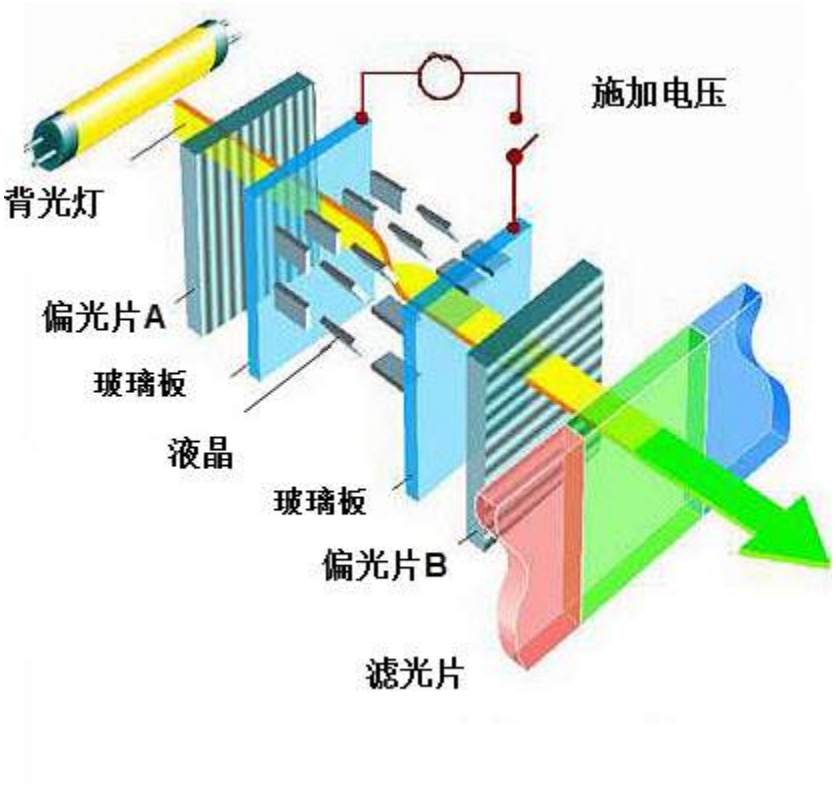


图 26-2 液晶屏的绿色显示结构

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。

27.1.2 LED 和 OLED 显示器

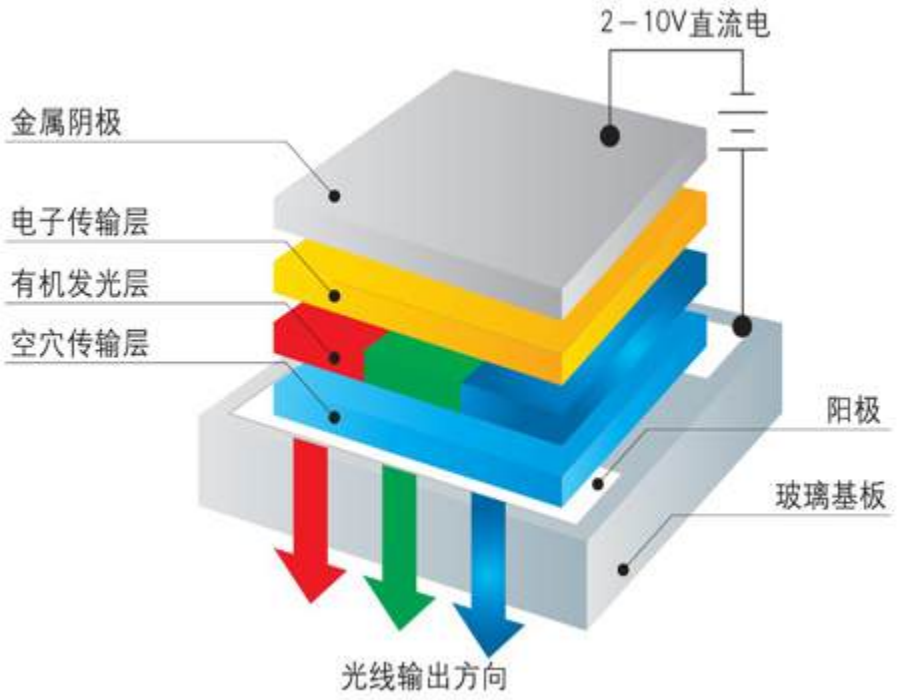
LED 点阵显示器不存在以上液晶显示器的问题，LED 点阵彩色显示器的单个像素点内包含红绿蓝三色 LED 灯，显示原理类似我们实验板上的 LED 彩灯，通过控制红绿蓝颜色的强度进行混色，实现全彩颜色输出，多个像素点构成一个屏幕。由于每个像素点都是 LED 灯自发光的，所以在户外白天也显示得非常清晰，但由于 LED 灯体积较大，导致屏幕的像素密度低，所以它一般只适合用于广场上的巨型显示器。相对来说，单色的 LED 点阵显示器应用得更广泛，如公交车上

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*。





27.1.3 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

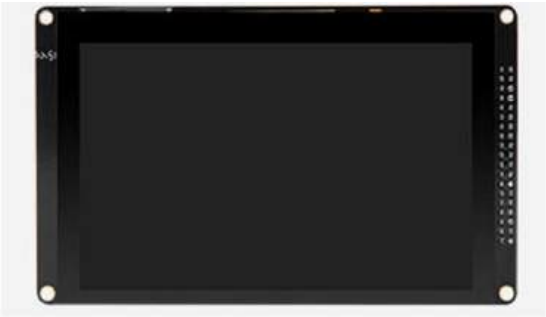
显示器的大小一般以英寸表示，如 5 英寸、21 英寸、24 英寸等，这个长度是指屏幕对角线的长度，通过显示器的对角线长度及长宽比可确定显示器的实际长宽尺寸。

(4) 点距

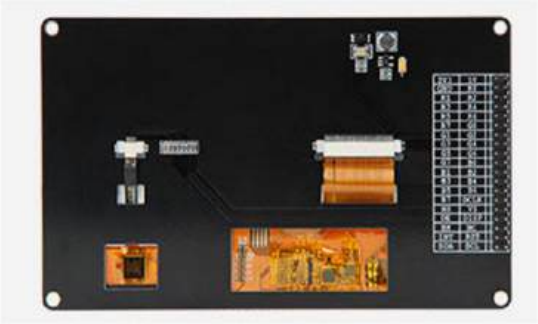
点距指两个相邻像素点之间的距离，它会影响画质的细腻度及观看距离，相同尺寸的屏幕，若分辨率越高，则点距越小，画质越细腻。如现在有些手机的屏幕分辨率比电脑显示器的还大，这是手机屏幕点距小的原因；LED 点阵显示屏的点距一般都比较大，所以适合远距离观看。

27.2 液晶控制原理

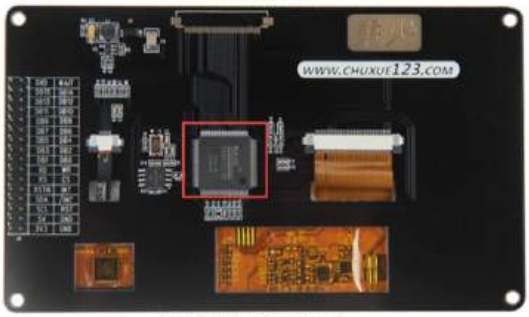
图 *26_6* 是两种适合于 STM32 芯片使用的显示屏，我们以它为例讲解控制液晶屏的基本原理。



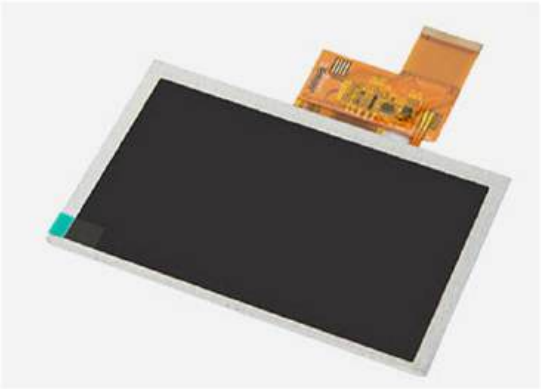
触摸显示屏正面



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

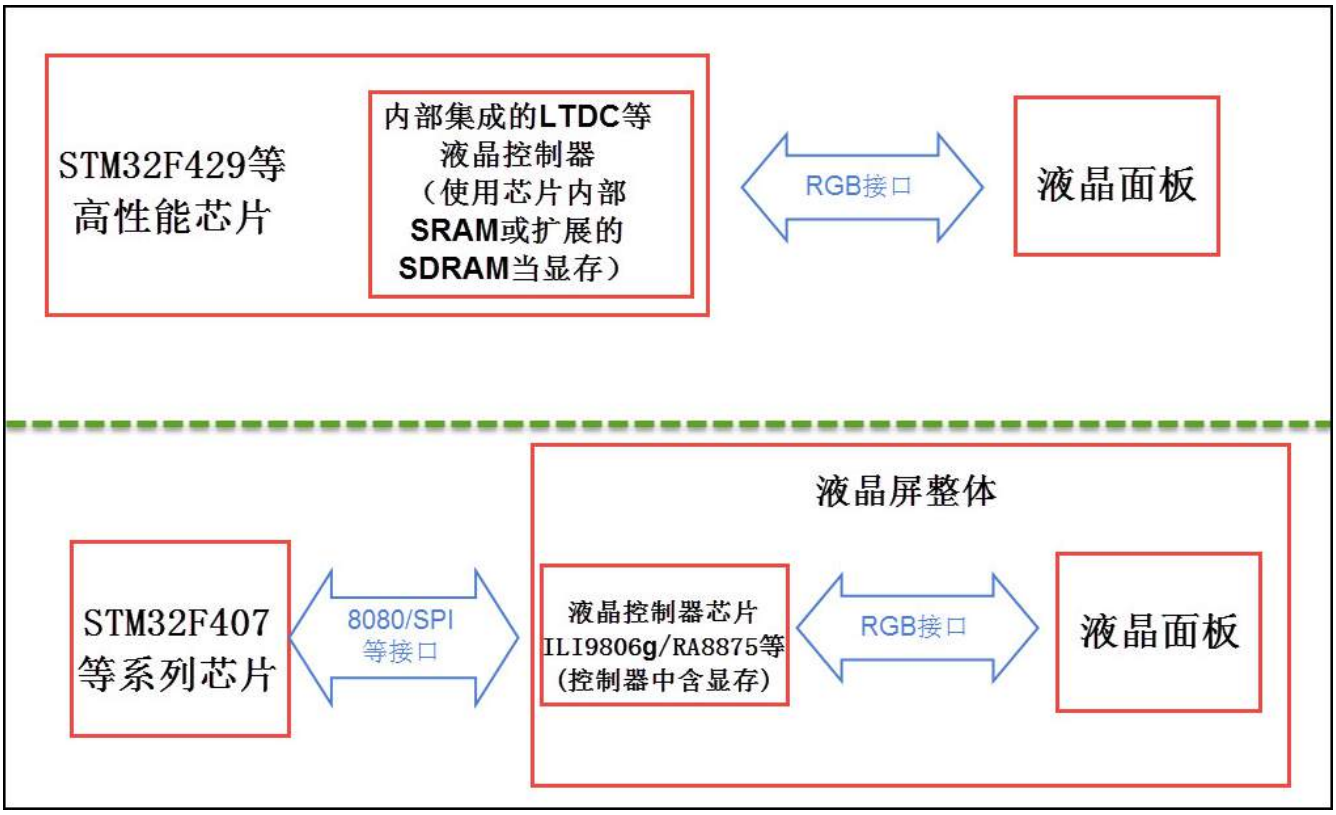


电容触摸屏面板
（带触摸控制器）

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 STM32F407 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。



27.2.1 液晶面板的控制信号

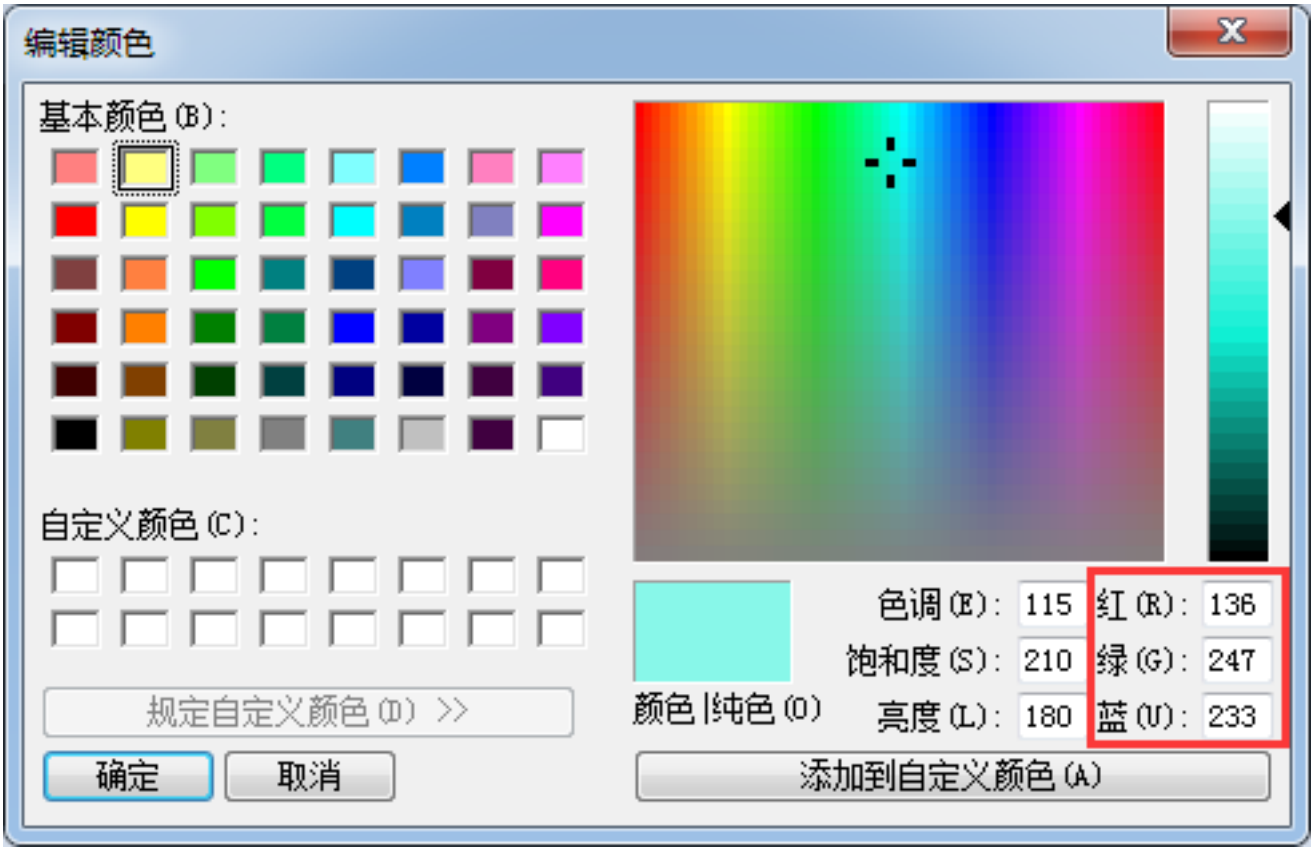
本章我们主要讲解如何控制液晶面板，液晶面板的控制信号线即图 *26_6* 中液晶面板引出的 FPC 排线，其说明见表 26-1，液晶面板通过这些信号线与液晶控制器通讯，使用这种通讯信号的被称为 RGB 接口 (RGB Interface)。

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

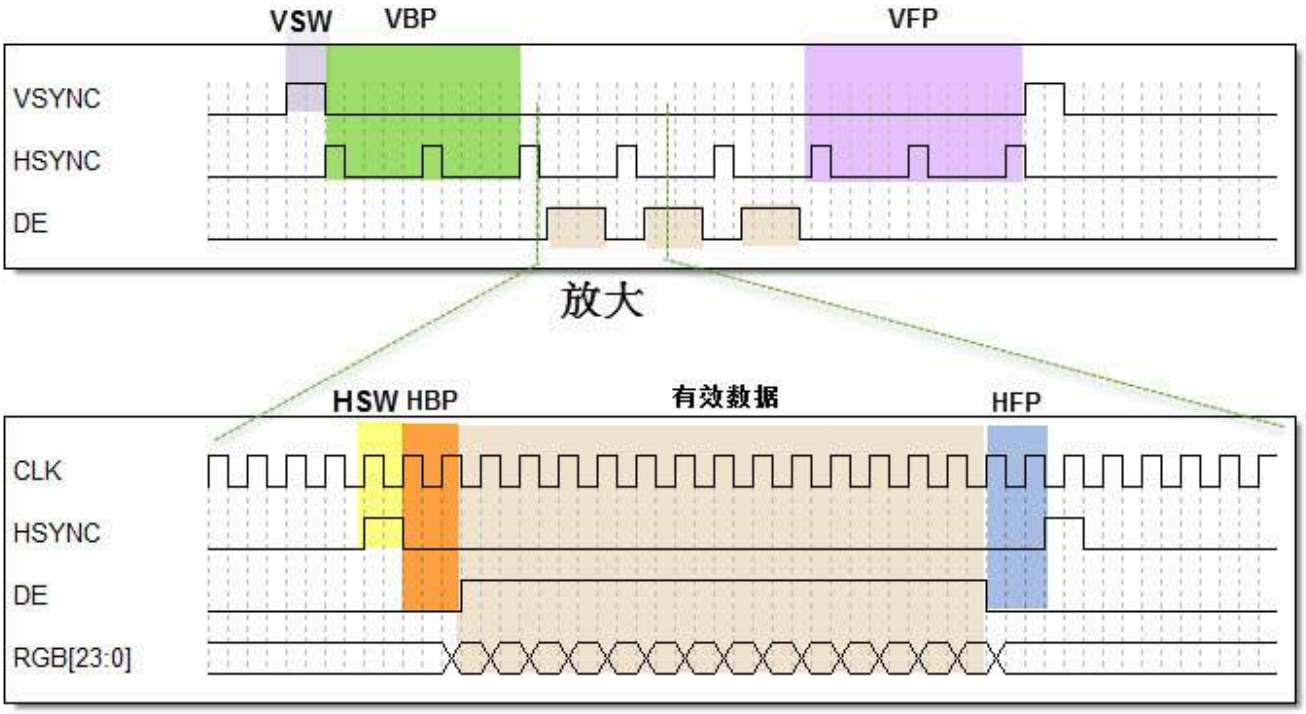
速率运行时，VSYNC 每秒钟电平会跳变 60 次。

(5) 数据使能信号 DE

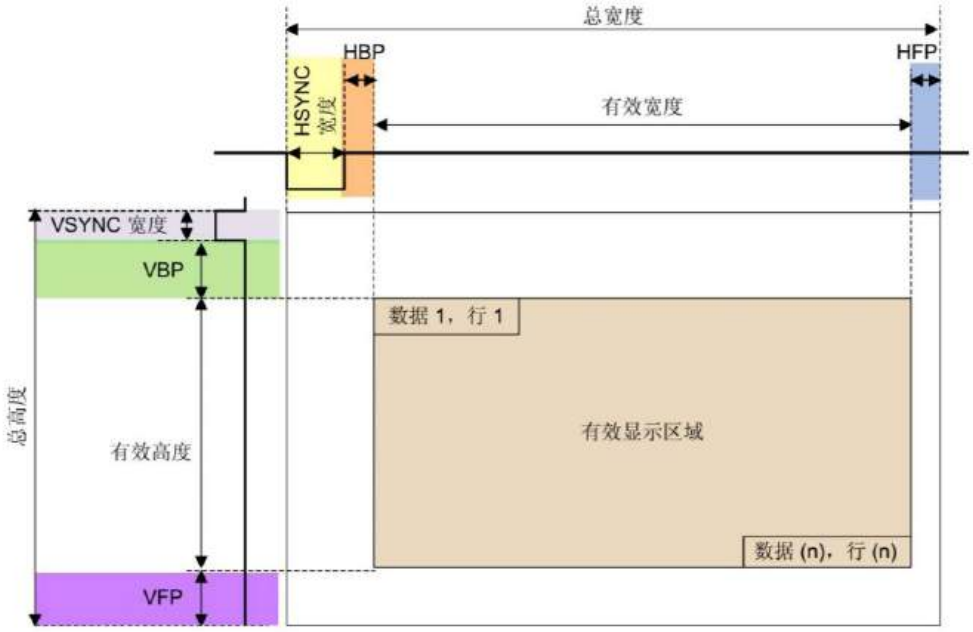
数据使能信号 DE(Data Enable) 用于表示数据的有效性，当 DE 信号线为高电平时，RGB 信号线表示的数据有效。

27.2.2 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。

27.2.3 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。

27.3 野火 4.3 寸液晶屏简介

27.3.1 4.3 寸电容触摸屏实物

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 854*480 的 4.3 寸电阻触摸液晶屏。图 *26_11*。

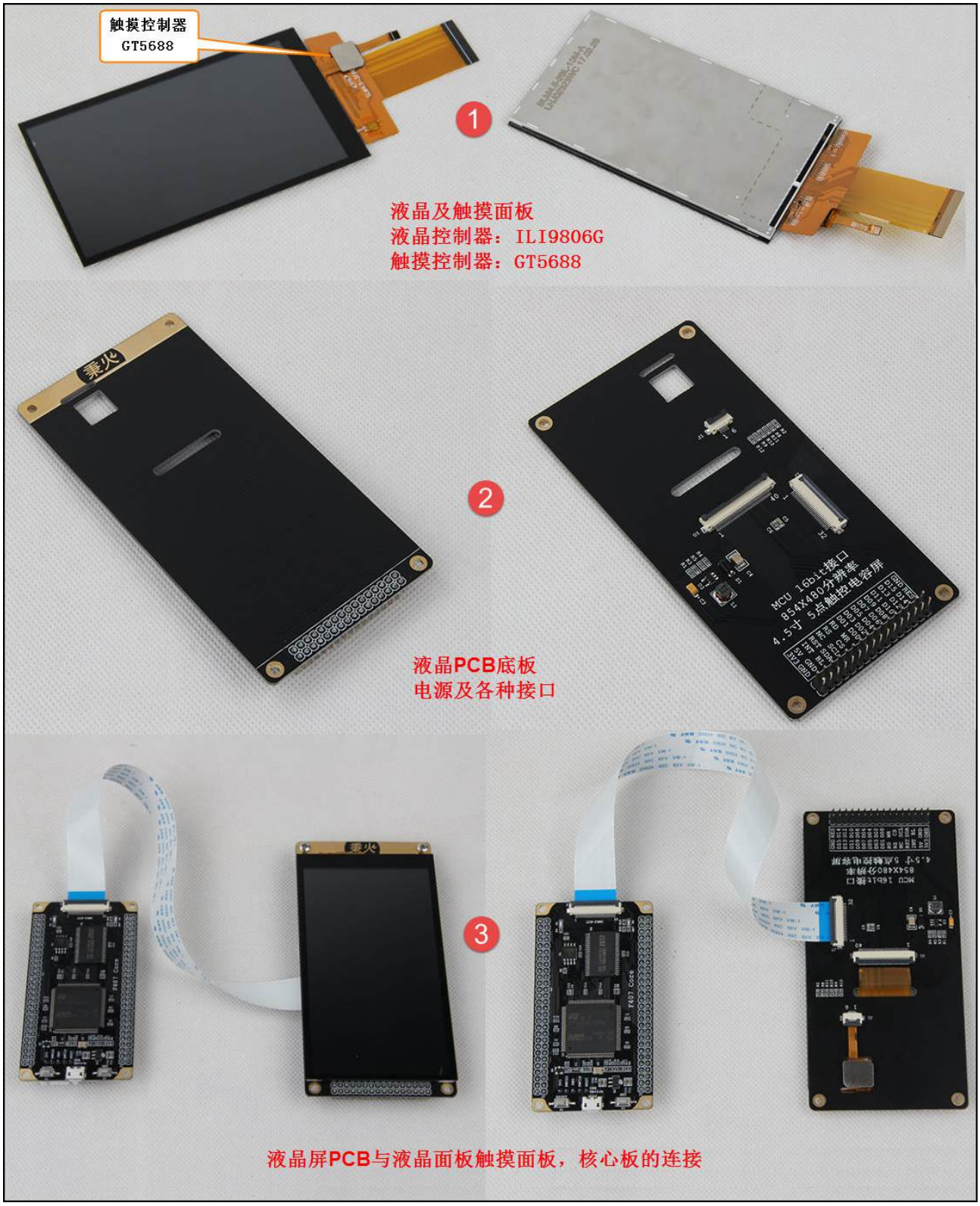

触摸控制器
GT5688
1
液晶及触摸面板
液晶控制器：ILI9806G
触摸控制器：GT5688
2
液晶PCB底板
电源及各种接口
3
液晶屏PCB与液晶面板触摸面板，核心板的连接

图 26-11 实验板标配的电容触摸屏

图中的标号 3 部分是液晶屏幕的整体，通过引出的排针或 FPC 排线接入到实验板上可对它进行控制，它分为标号 1 的液晶触摸面板和标号 3 的 PCB 底板两部分。

标号 1 处的液晶触摸面板由液晶屏和触摸面板组成，触摸面板和液晶面板是两个独立的部件，生产厂商把这两个部件帖在了一起，形成整体。

触摸面板引出的信号线引入到 GT917S 芯片，该芯片会对触摸信号进行采集并简化信号输出给外部控制器。

触摸面板的下方即为液晶面板，在它的内部包含了一个型号为 NT35510 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 NT35510 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 2 处的是 PCB 底板，它主要包含了液晶屏的电源控制电路，另外，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，也可以使用 FPC 接口与核心板连接。

27.3.2 NT35510 液晶控制器简介

本液晶屏内部包含有一个液晶控制芯片 NT35510，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 NT35510 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 8、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 NT35510 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。

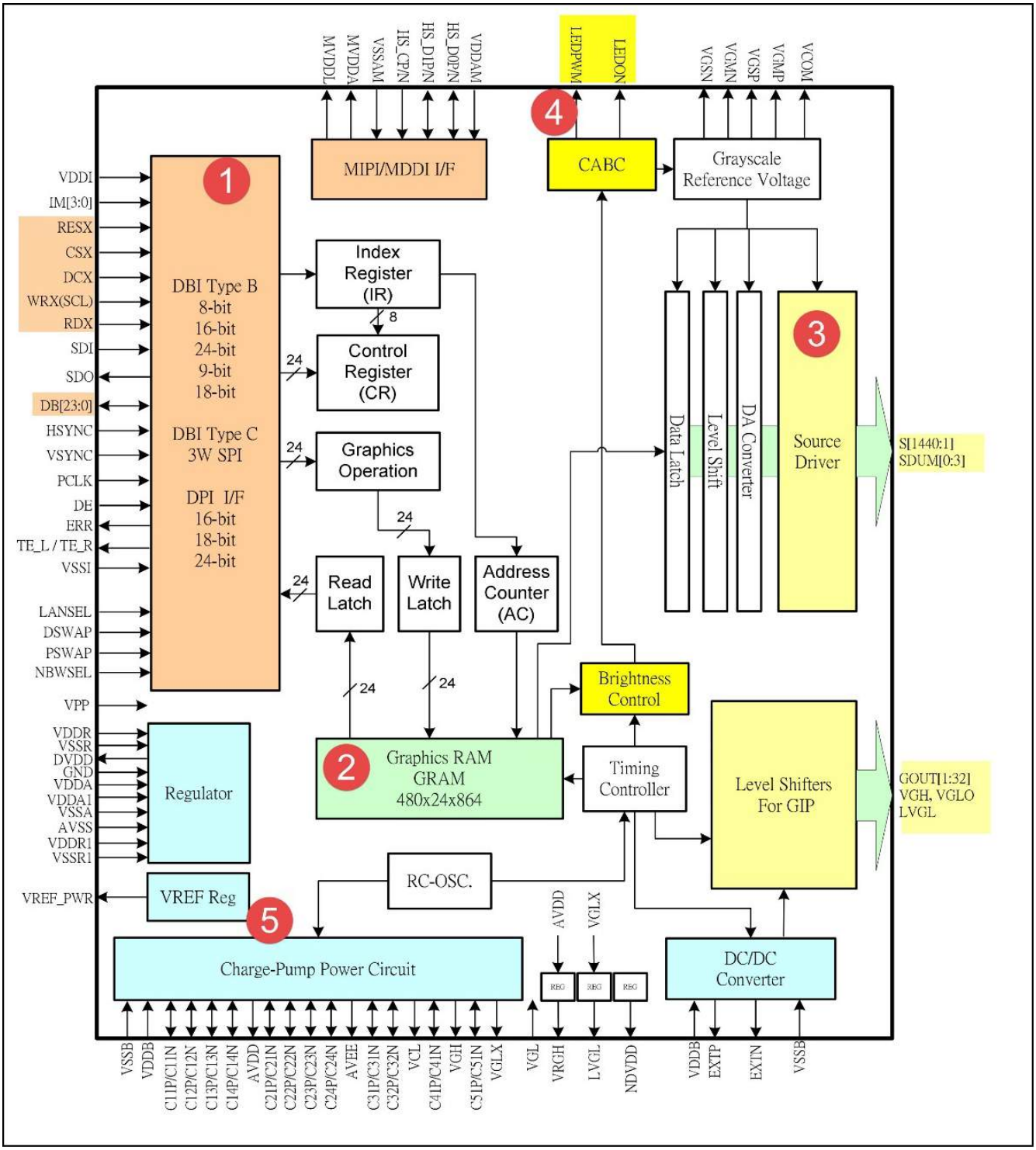


图 26-12 NT35510 控制器内部框图

27.3.3 液晶屏的信号线及 8080 时序

NT35510 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 NT35510 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，且背光 LED 引脚不与 NT35510 相连，而是直接引出到排针供外部控制器控制。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*。

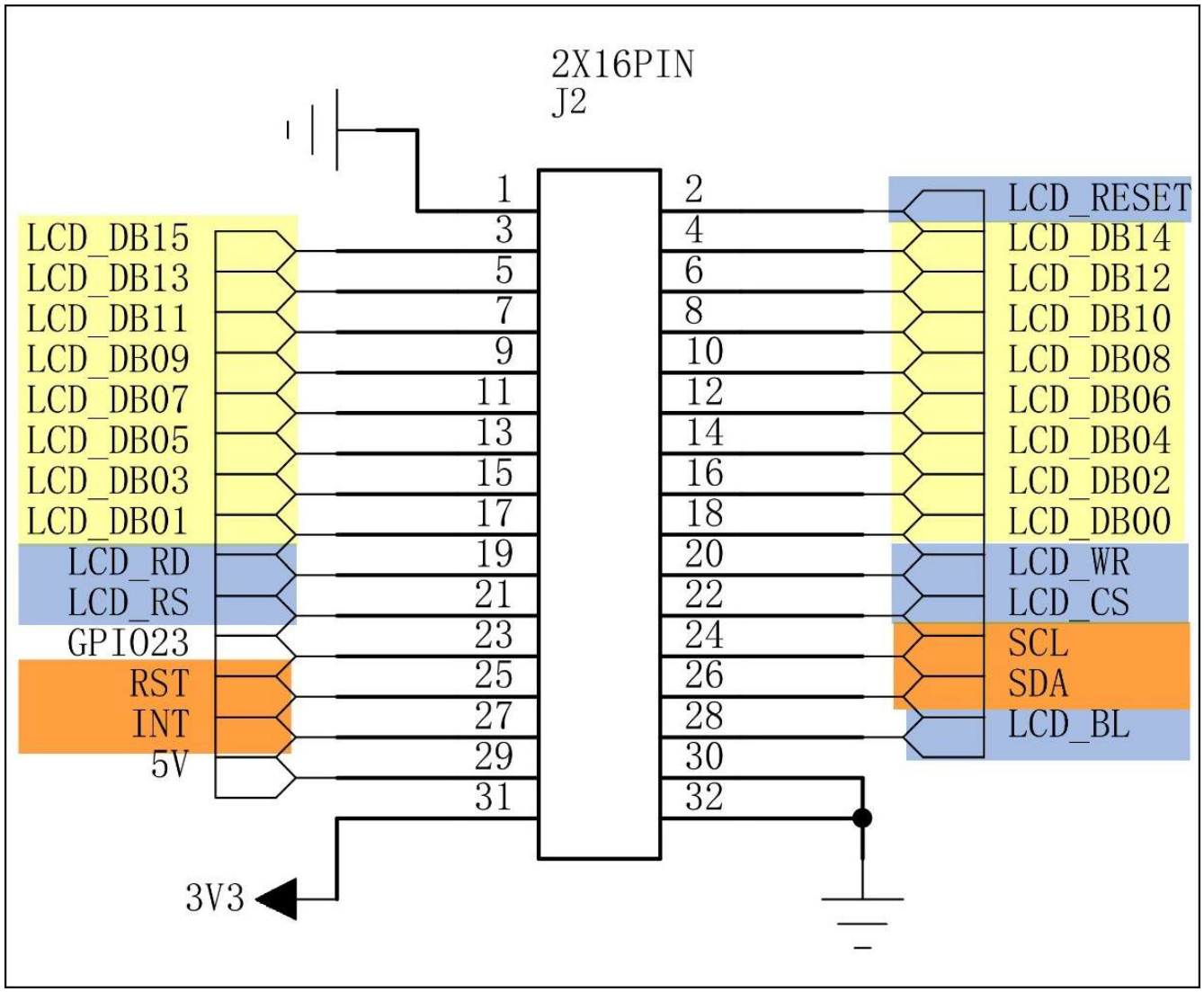


图 26-13 液晶屏引出的信号线

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	NT35510 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
GPIO23	•	空引脚，不需要连接
RST	与触摸 IC 相连	触摸 IC 的复位引脚
INT	与触摸 IC 相连	触摸 IC 的中断信号引脚
SCL	与触摸 IC 相连	触摸 IC 的 IIC 总线的时钟信号
SDA	与触摸 IC 相连	触摸 IC 的 IIC 总线的数据信号

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 NT35510 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 NT35510 的控制指令，MCU 可通过 8080 接口发送命令编码控制 NT35510 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《NT35510-Data Sheet.pdf》数据手册均有详细说明。写命令时序图见图 *26_14*。

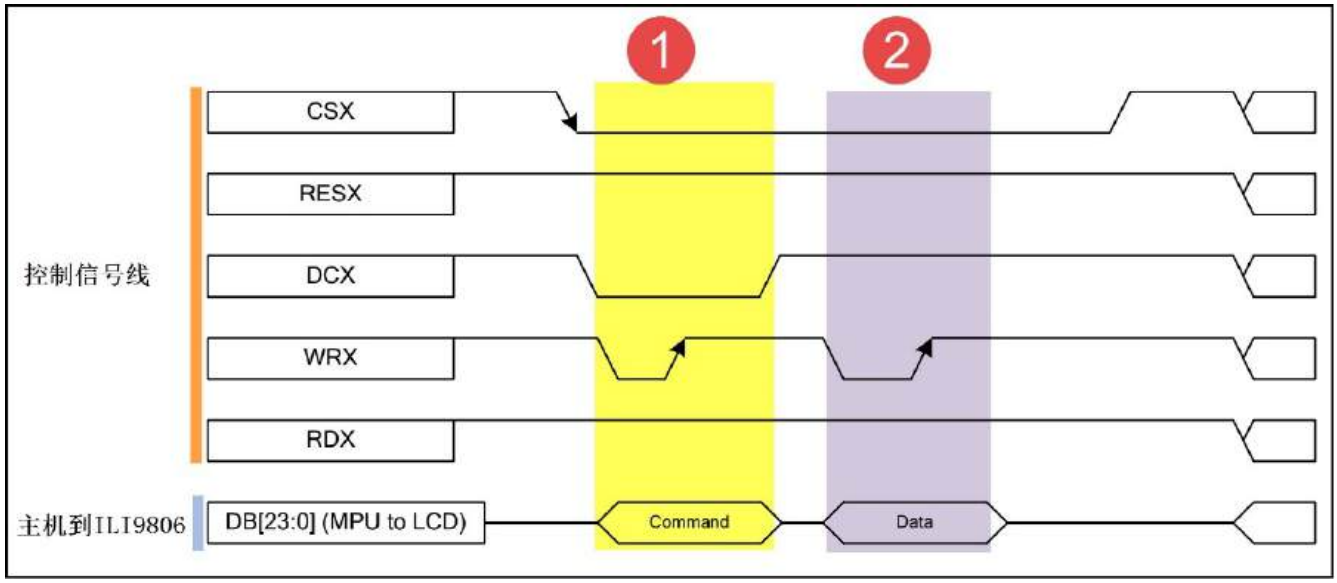


图 26-14 使用 8080 接口写命令时序

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[23:0](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[23:0] 传输的数据则会被 NT35510 保存至它的 GRAM 中。

27.4 使用 STM32 的 FSMC 模拟 8080 接口时序

NT35510 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。

27.4.1 FSMC 简介

在前面的《FSMC—扩展外部 SRAM》章节中了解到 STM32 的 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*。

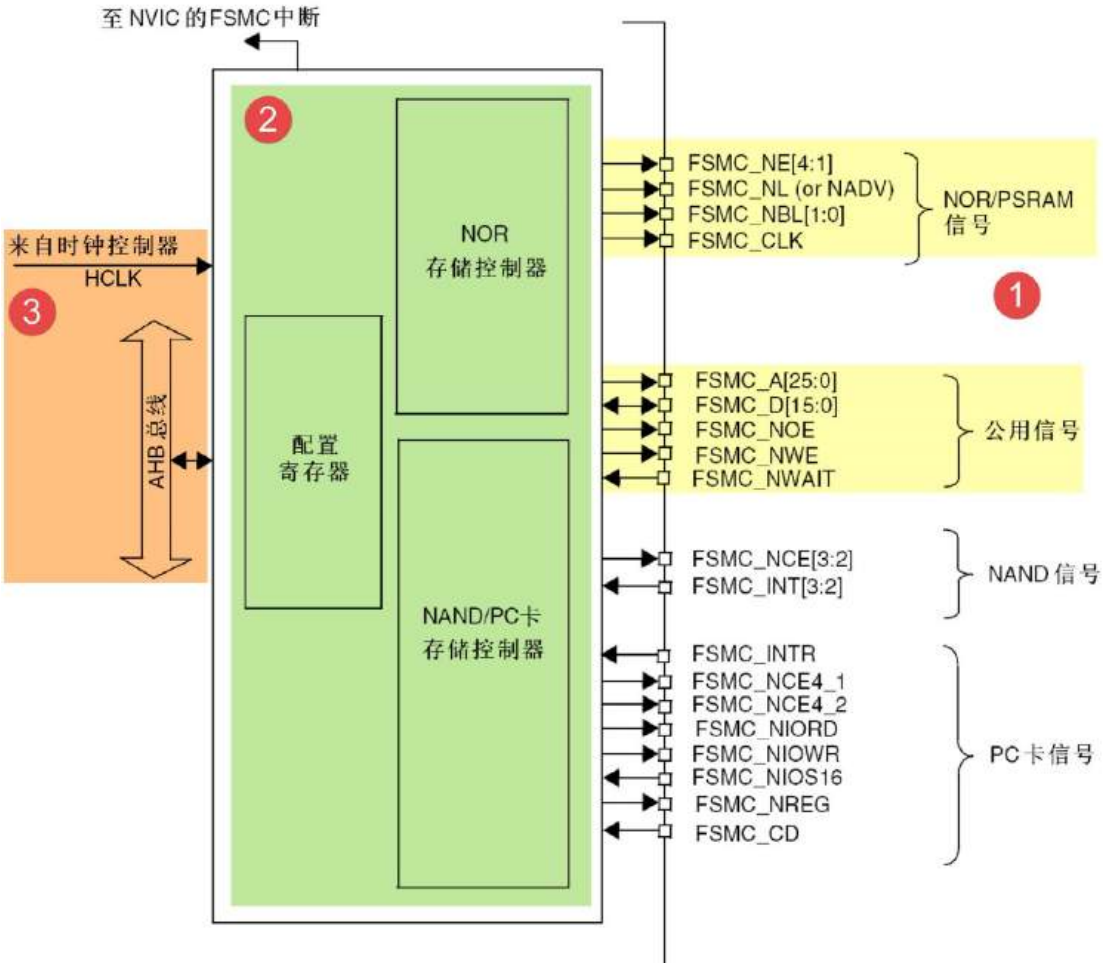


图 27-15 FSMC 结构图

控制 LCD 时，是使用 FSMC 的 NORPSRAM 模式的，且与前面使用 FSMC 控制 SRAM 的稍有不同，控制 SRAM 时使用的是模式 A，而控制 LCD 时使用的是与 NOR FLASH 一样的模式 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 主要使用到下表的信号线：

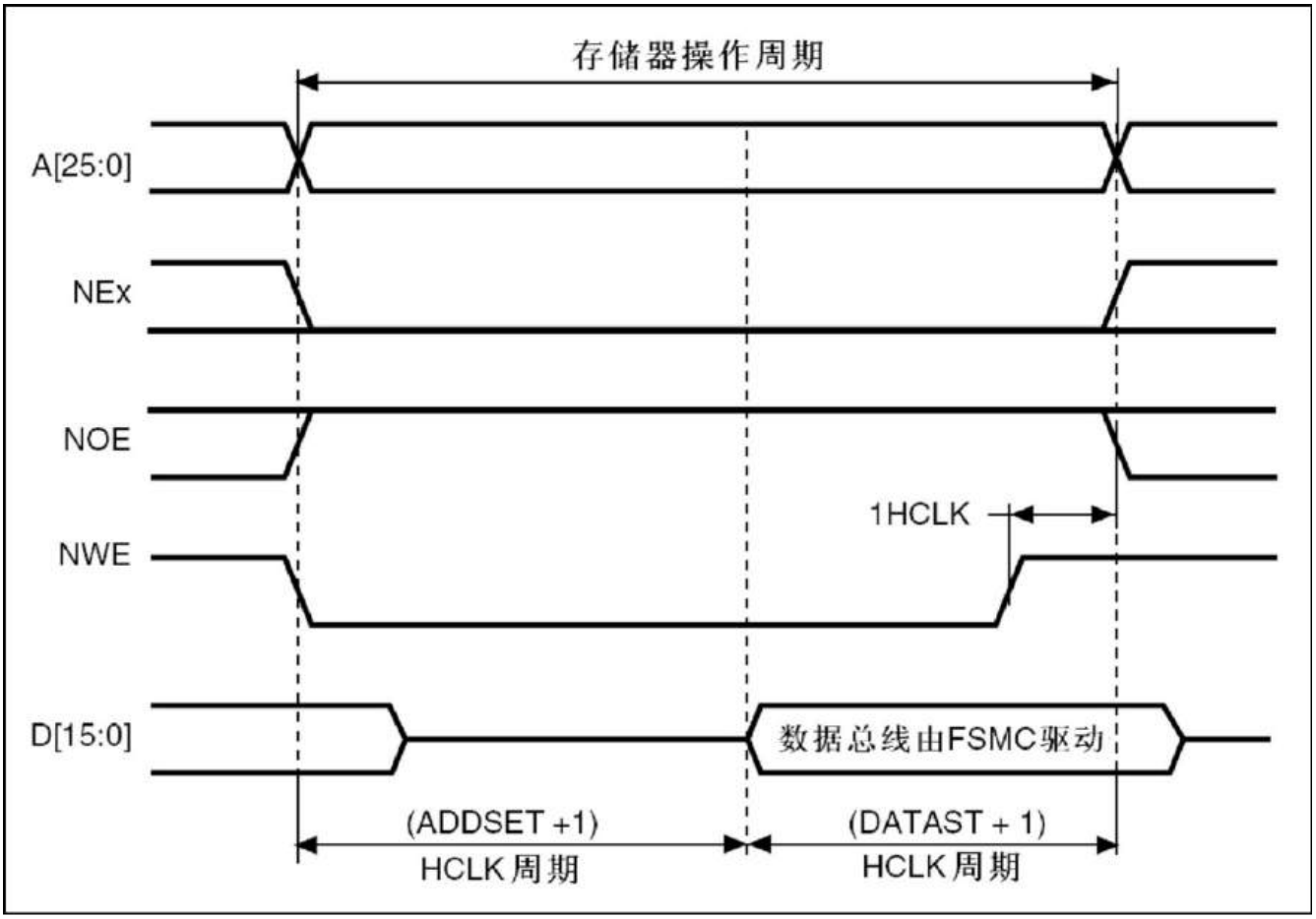
表 27-4 FSMC 控制 NOR FLASH 的信号线

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。

FSMC NOR/PSRAM 中的模式 B 的写时序见图



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。

27.4.2 用 FSMC 模拟 8080 时序

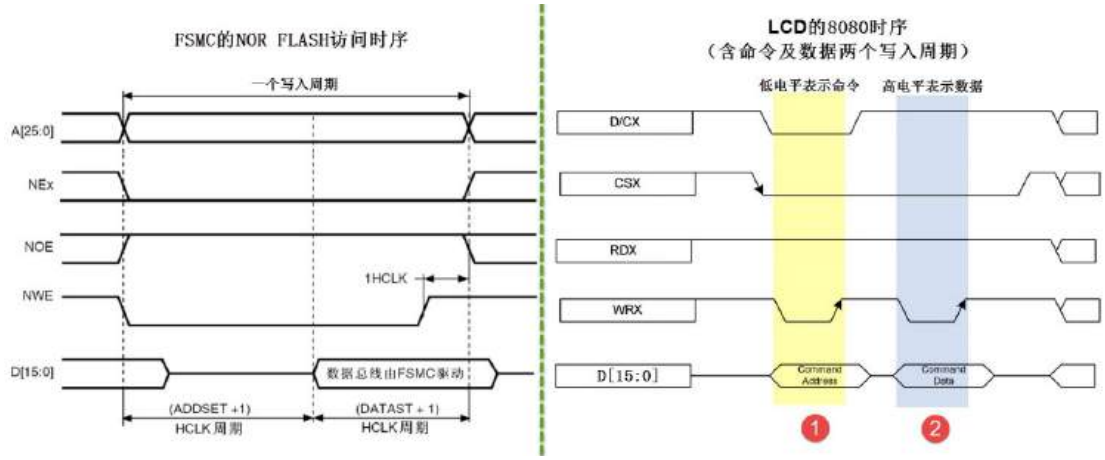


图 27-20FSMC 模式 B 时序与 8080 时序对比（写过程）

见表 27-5，对比 FSMC NOR/PSRAM 中的模式 B 时序与 NT35510 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

NT35510 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 NT35510 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 NT35510 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 NT35510 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。

27.5 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 1: 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;
```

这个结构体与 SRAM 中的时序结构体完全一样，以下仅列出控制 NOR FLASH 时使用模式 B 用到的结构体成员说明：

(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 168MHz，即一个 HCLK 周期为 1/168 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。

27.6 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 2: 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 */
```

(下页继续)

(续上页)

```
  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 */

  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 */
```

(下页继续)

(续上页)

```
    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下表。

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。

27.7 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 4.3 寸 NT35510 液晶屏，见图 *26_18*，该液晶屏的分辨率为 854x480，支持 RGB565 格式。

学习本小节内容时，请打开配套的“FSMC—液晶显示英文”工程配合阅读。

27.7.1 硬件设计

正面图



背面

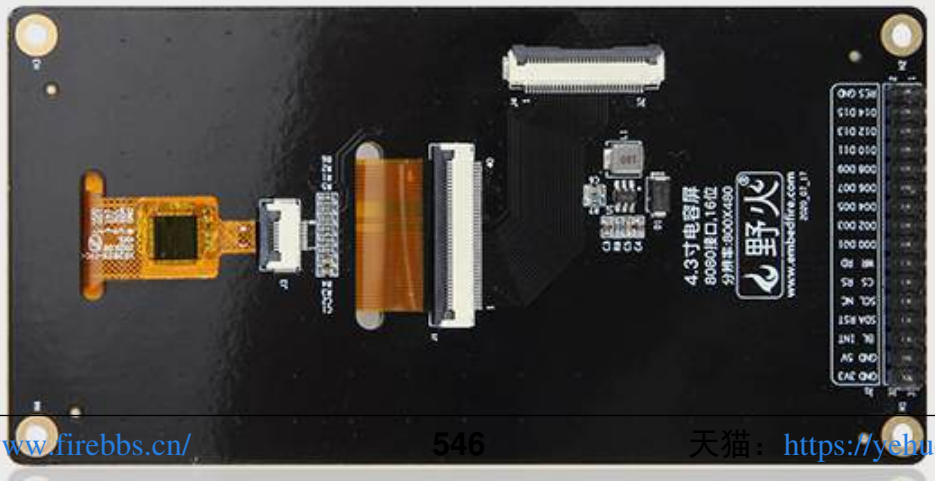


图 26-18 液晶屏实物图

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。

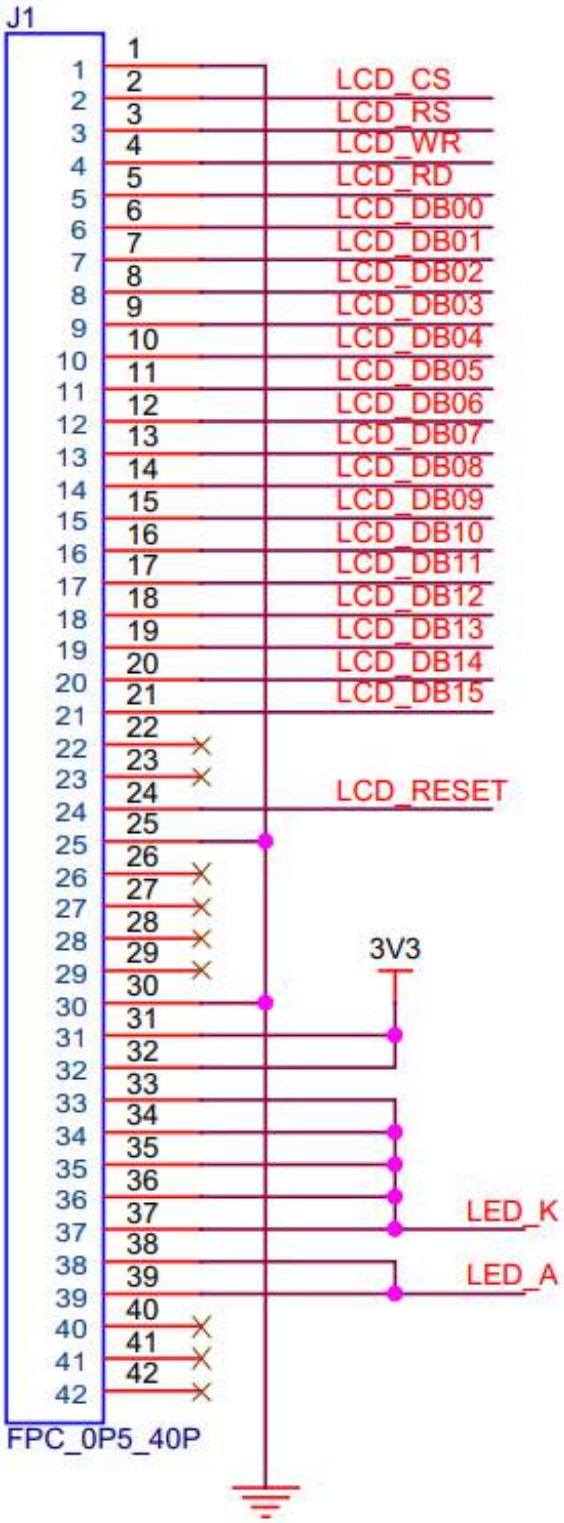

J1
FPC_0P5_40P
LCD_CS
LCD_RS
LCD_WR
LCD_RD
LCD_DB00
LCD_DB01
LCD_DB02
LCD_DB03
LCD_DB04
LCD_DB05
LCD_DB06
LCD_DB07
LCD_DB08
LCD_DB09
LCD_DB10
LCD_DB11
LCD_DB12
LCD_DB13
LCD_DB14
LCD_DB15
LCD_RESET
3V3
LED_K
LED_A

图 26-19 屏幕 PCB 底板原理图 (截图于《4.3 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_K 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。除液晶部分的 FPC 接口外，PCB 底板还有一个连接到电容触摸屏控制器的 FPC 接口，用于检测触摸信号。

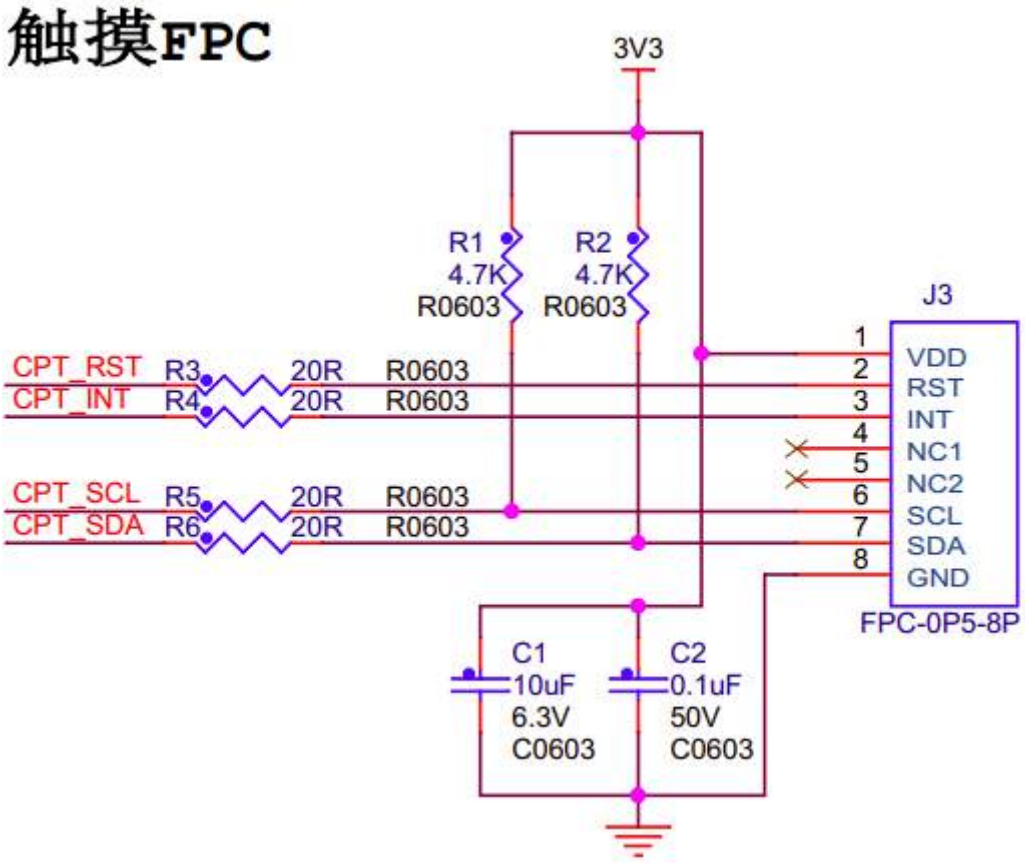


图 26-20 屏幕 PCB 底板的触摸部分原理图 (截图于《4.3 寸液晶原理图.pdf》)

触摸检测的主体是型号为 GT917S 的芯片，它接收触摸屏的信号进行处理，把触摸信息使用 I2C 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。

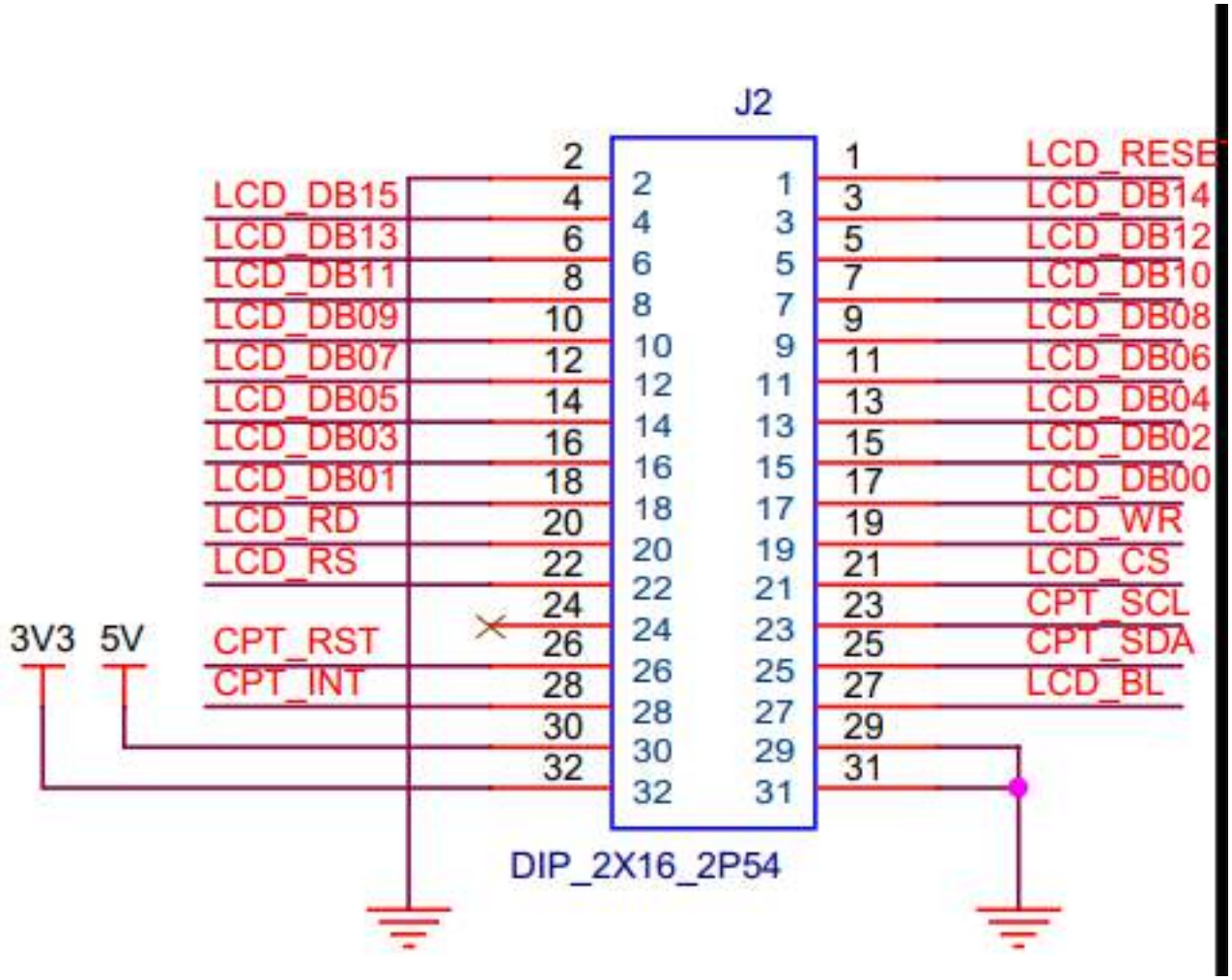


图 26-21 液晶屏接口 (截图于《4.3 寸液晶原理图.pdf》)

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。

液晶接口-FSMC-8080模式

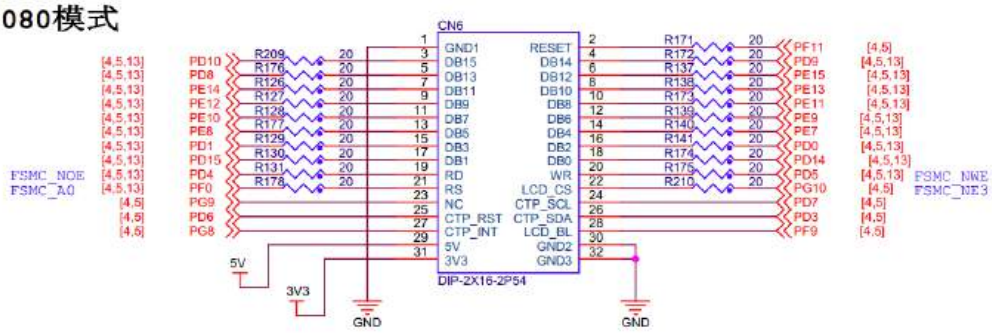


图 26-22 开发板与屏幕的连接的信号说明

图 *26_22* 是野火 F407 开发板上的液晶排母接口原理图，它说明了配套的 4.3 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《4.3 寸液晶原理图.pdf》及《F407 开发板底板原理图》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。

27.7.2 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_NT35510_lcd.c”及“bsp_NT35510_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。

27.7.2.1 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。

27.7.2.2 代码分析

27.7.2.2.1 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_NT35510_lcd.h”文件中，见*代码清单 26_3*。

列表 3: 代码清单 26-3 FSMC 硬件配置相关的宏 (省略了部分数据线)

```
//由片选引脚决定的 NOR/SRAM 块
#define      FSMC_Bank1_NORSRAMx           FSMC_Bank1_NORSRAM3
```

27.7.2.2.2 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 4: 代码清单 26-4 FSMC 的 GPIO 初始化函数 (省略了部分数据线)

```
SRAM_HandleTypeDef  SRAM_Handler;
FMC_NORSRAM_TimingTypeDef Timing;
/**
* @brief  初始化 NT35510 的 IO 引脚
* @param  无
* @retval 无
*/
static void NT35510_GPIO_Config ( void )
{
   GPIO_InitTypeDef  GPIO_Initure;
   /* Enable GPIOs clock */
__HAL_RCC_GPIOD_CLK_ENABLE();
```

(下页继续)

(续上页)

```
__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_HIGH;

GPIO_Initure.Pin=GPIO_PIN_10;
   HAL_GPIO_Init(GPIOG,&GPIO_Initure);

//初始化 PF11
   GPIO_Initure.Pin=GPIO_PIN_11;
   HAL_GPIO_Init(GPIOF,&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_PIN_8|\
                        GPIO_PIN_9|GPIO_PIN_10|GPIO_PIN_14|GPIO_PIN_15;
HAL_GPIO_Init(GPIOD, &GPIO_Initure);

   //初始化 PE7,8,9,10,11,12,13,14,15
   GPIO_Initure.Pin=GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10|GPIO_PIN_
↪11|\
                    GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
   HAL_GPIO_Init(GPIOE,&GPIO_Initure);

   //初始化 PF9
   GPIO_Initure.Pin=GPIO_PIN_0|GPIO_PIN_9;
```

(下页继续)

(续上页)

```
    HAL_GPIO_Init(GPIOF,&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。

27.7.2.2.3 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 5: 代码清单 26-5 配置 FSMC 的模式

```
static void NT35510_FSMC_Config ( void )
{
    SRAM_Handler.Instance = FMC_NORSRAM_DEVICE;
    SRAM_Handler.Extended = FMC_NORSRAM_EXTENDED_DEVICE;
    __HAL_RCC_FSMC_CLK_ENABLE();
    /* SRAM device configuration */
    Timing.AddressSetupTime      = 0x04;
    Timing.AddressHoldTime       = 0x00;
    Timing.DataSetupTime         = 0x04;
    Timing.BusTurnAroundDuration = 0x00;
    Timing.CLKDivision           = 0x00;
    Timing.DataLatency           = 0x00;
    Timing.AccessMode            = FSMC_ACCESS_MODE_B;
    //使用 NE3
    SRAM_Handler.Init.NSBank=FSMC_NORSRAM_BANK3;
    //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;
```

(下页继续)

(续上页)

```
    //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
    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_Handler.Init.ContinuousClock=FSMC_CONTINUOUS_CLOCK_SYNC_ASYNC;
    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：

(1) **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。

(2) **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 168MHz，对应每个时钟周期为 1/168 微秒。

由图 *26_23* 及图 *26_24* 中的 NT35510 时序参数说明及要求可大致得知 NT35510 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）

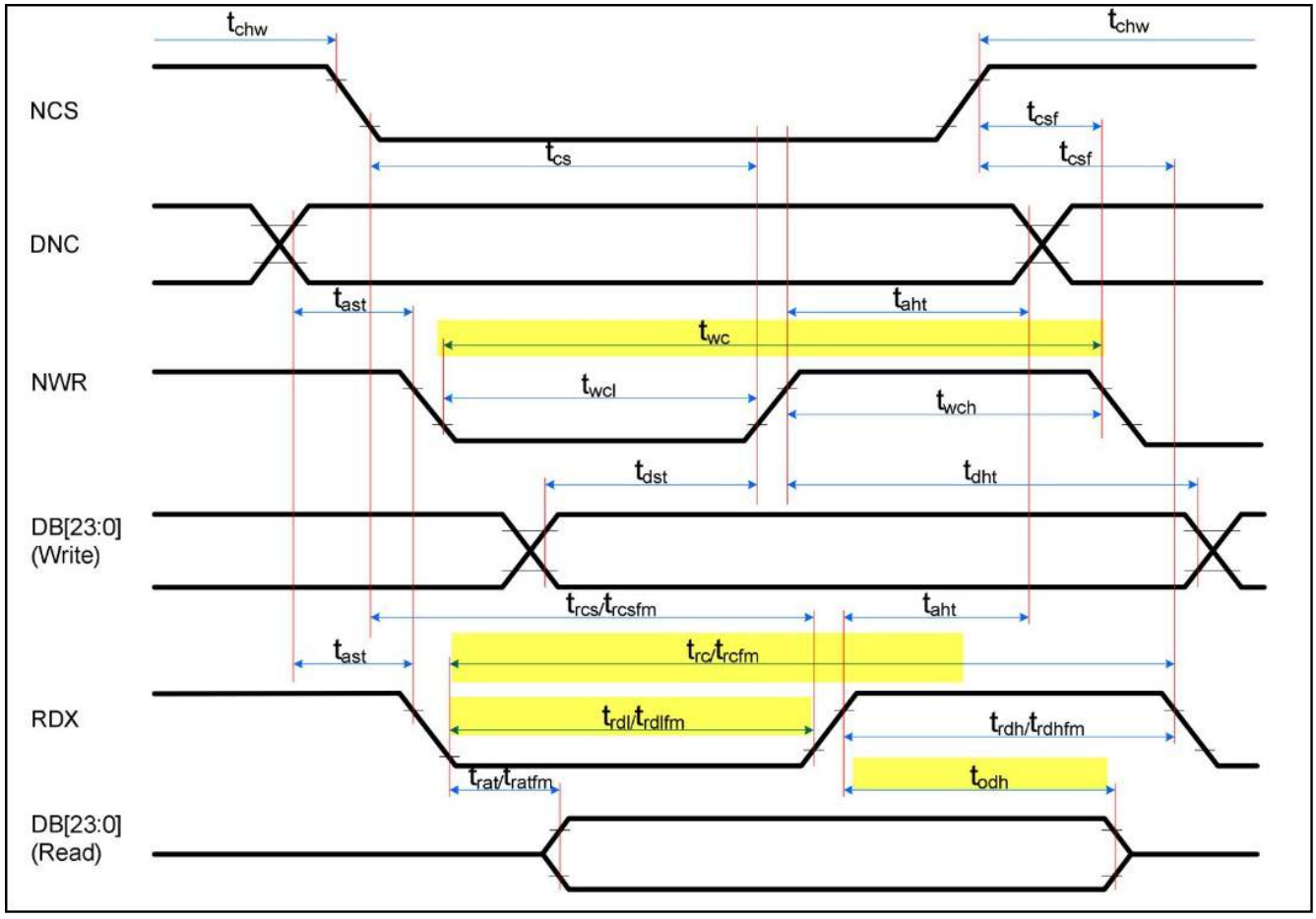


图 26-23 NT35510 时序参数说明图 (摘自 NT35510 数据手册)

Signal	Symbol	Parameter	min	max	Unit	Description
DCX	tast	Address setup time	0	-	ns	-
	taht	Address hold time (Write/Read)	10	-	ns	-
CSX	tchw	CSX "H" pulse width	0	-	ns	-
	tcs	Chip Select setup time (Write)	10	-	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	30	-	ns	-
	twrh	Write Control pulse H duration	10	-	ns	-
	twrl	Write Control pulse L duration	10	-	ns	-
RDX (FM)	trcfm	Read Cycle (FM)	450	-	ns	When read from the Frame Memory
	trdhfm	Read Control H duration (FM)	90	-	ns	
	trdlfm	Read Control L duration (FM)	355	-	ns	
RDX (ID)	trc	Read cycle (ID)	160	-	ns	When read ID data
	trdh	Read Control pulse H duration	90	-	ns	
	trdl	Read Control pulse L duration	45	-	ns	
DB[17:0], DB[15:0], DB[8:0] DB[7:0]	tdst	Write data setup time	10	-	ns	CL = 30pF (maximum) CL = 8pF (minimum)
	tdht	Write data hold time	10	-	ns	
	trat	Read access time	-	40	ns	
	tratfm	Read access time	-	340	ns	
	trodh	Read output disable time	20	80	ns	

图 26-24 NT35510 的时序参数要求 (摘自 NT35510 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*。

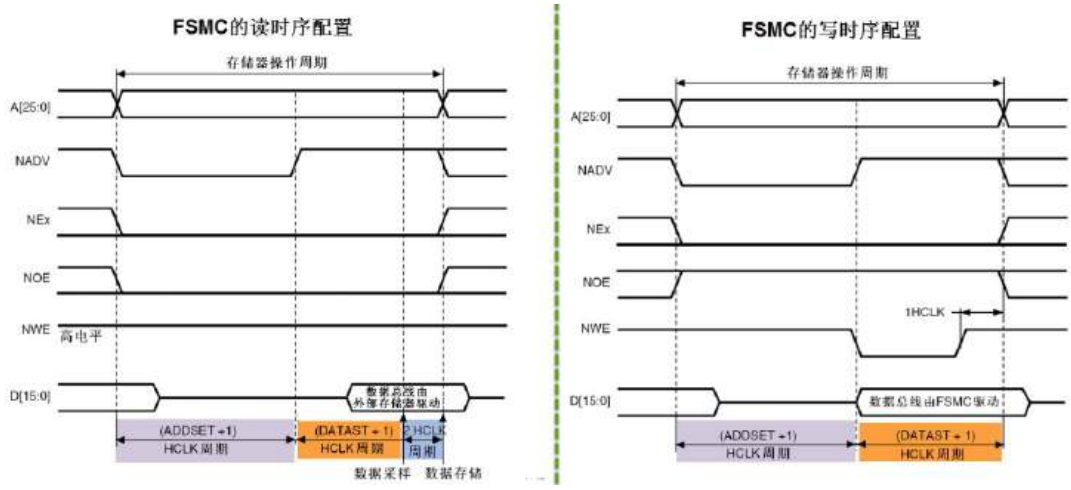


图 26-25FSMC 的读写时序

结合 NT35510 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 30ns 来计算，

配置结果为 ADDSET = 1 及 DATST = 4，把时间单位 1/168 微秒 (即 1000/168 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) *1000/168 = ((4+1)+(4+1)+2)*1000/168 = 71ns

写周期：t_{wc} =((ADDSET+1)+(DATST+1)) *1000/168 = ((1+4)+(4+1))*1000/168 = 60ns

所以把这两个参数值写入到 FSMC 后，它控制的读写周期都比 NT35510 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）DataSetupTime（即 DATAST 的值）中，然后再把时序结构体作为指针赋值到下面的 FSMC 初始化结构体中，作为读写的时序参数，最后再调用 FSMC_NORSRAMInit 函数即可把参数写入到相应的寄存器中。

(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

  NSBank 成员设置 FSMC 的 NOR FLASH 存储区域映射选择为宏 FSMC_Bank1_NORSRAMx (即 FSMC_Bank1_NORSRAM3)，这是由于我们的 SRAM 硬件连接到 FSMC_NE3 和 NOR/PSRAM 相关引脚，所以对应到存储区域 Bank1 SRAM3，对应的基地址为 0X6800 0000；

- 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 FSMC_NORSRAMInit 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。最后调用 FSMC_NORSRAMCmd 函数使能要控制的存储区域 FSMC_Bank1_NORSRAM3。

27.7.2.2.4 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 NT35510 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。

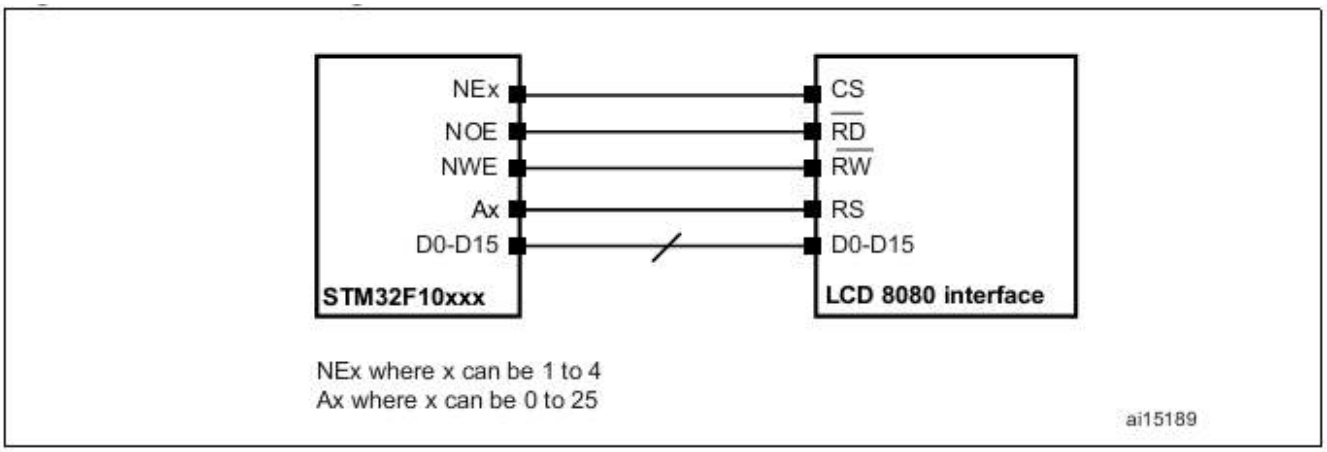


图 26-26 FSMC 与 8080 端口连接简图

计算地址的过程如下：

(1) 本工程中使用的是 FSMC_NE3 作为 8080_CS 片选信号，所以首先可以确认地址范围，当访问 0X6800 0000 ~ 0X6BFF 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_A0 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_A0 输出高电平的地址，即可控制表示数据，选择出使得 FSMC_A0 输出低电平的地址，即可控制表示命令。

- 要使 FSMC_A0 地址线为高电平，实质是输出地址信号的第 0 位为 1 即可，使用 0X6800 0000~0X6BFF FFFF 内的任意地址，作如下运算：

  设置地址的第 0 位为 1：0X6800 0000 |= (1«0) = 0x6800 0001

- 要使 FSMC_A0 地址线为低电平，实质是输出地址信号的第 0 位为 0 即可，使用 0X6800 0000~0X6BFF FFFF 内的任意地址，作如下运算：

  设置地址的第 0 位为 0：0X6800 0000 &= ~ (1«0) = 0x6800 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_A0 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_A0 地址线为高电平，实质是访问内部 HADDR 地址的第 (0+1) 位为 1 即可，使用 0X6800 0000~0X6BFF FFFF 内的任意地址，作如下运算：

  使 FSMC_A0 地址线为高电平：0X6800 0000 |= (1«(0+1)) = 0x6800 0002

- 要使 FSMC_A0 地址线为低电平，实质是访问内部 HADDR 地址的第 (0+1) 位为 0 即可，使用 0X6800 0000~0X6BFF FFFF 内的任意地址，作如下运算：

  使 FSMC_A0 地址线为低电平：0X6800 0000 &= ~ (1«(0+1)) = 0x6800 0000

根据最终的计算结果，总结如下：当 STM32 访问内部的 0x6800 0002 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_A0 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 0X6800 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_A0 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 6: 代码清单 26_6 使用 FSMC 访问数据及访问命令的地址（bsp_NT35510_lcd.h 文件）

```
/*****************************************************
^26 =0X0400 0000 = 64MB, 每个 BANK 有 4*64MB = 256MB
MB:FSMC_Bank1_NORSRAM1:0X6000 0000 ~ 0X63FF FFFF
```

(下页继续)

(续上页)

```
MB:FSMC_Bank1_NORSRAM2:0X6400 0000 ~ 0X67FF FFFF
MB:FSMC_Bank1_NORSRAM3:0X6800 0000 ~ 0X6BFF FFFF
MB:FSMC_Bank1_NORSRAM4:0X6C00 0000 ~ 0X6FFF FFFF

选择 BANK1-BORSRAM3 连接 TFT，地址范围为 0X6800 0000 ~ 0X6BFF FFFF
FSMC_A0 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+2^0*2 = 0X6800 0000 + 0X2 = 0X6800␣
↪0002
当选择不同的地址线时，地址要重新计算
*****************************************************************/

/********************** NT35510 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_NT35510_CMD         ( ( uint32_t ) 0x68000000 )

//FSMC_Bank1_NORSRAM 用于 LCD 数据操作的地址
#define      FSMC_Addr_NT35510_DATA        ( ( uint32_t ) 0x68000002 )
```

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 7: 代码清单 26-7 向液晶屏发送命令及发送数据的操作

```
/**
* @brief  向 NT35510 写入命令
* @param  usCmd : 要写入的命令(表寄存器地址)
* @retval 无
*/
__inline void NT35510_Write_Cmd ( uint16_t usCmd )
{
```

(下页继续)

(续上页)

```
    * ( __IO uint16_t * ) ( FSMC_Addr_NT35510_CMD ) = usCmd;

}

/**
* @brief  向 NT35510 写入数据
* @param  usData : 要写入的数据
* @retval 无
*/
__inline void NT35510_Write_Data ( uint16_t usData )
{
    * ( __IO uint16_t * ) ( FSMC_Addr_NT35510_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。

27.7.2.2.5 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 8: 代码清单 26-8 向液晶屏写入初始化配置

```
/**
* @brief  初始化 NT35510 寄存器
* @param  无
* @retval 无
*/
static void NT35510_REG_Config ( void )
{
    /* EXTC Command Set enable register */
```

(下页继续)

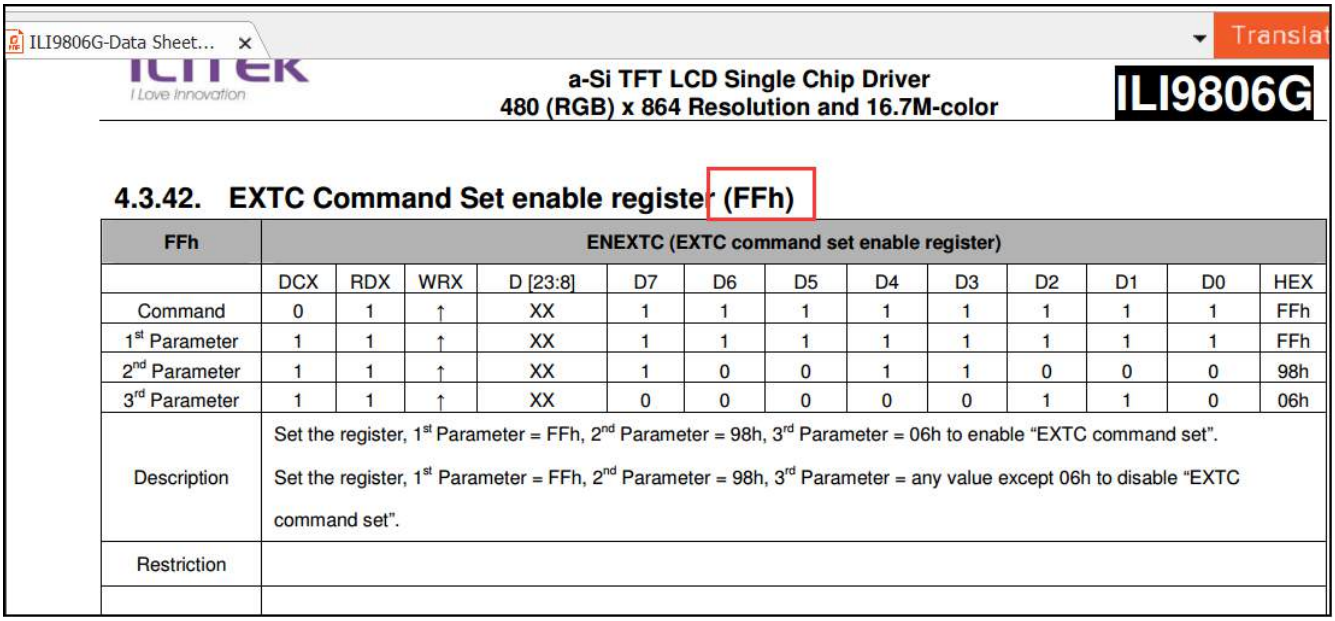
(续上页)

```
    DEBUG_DELAY  ();
    NT35510_Write_Cmd ( 0xFF  );
    NT35510_Write_Data ( 0xFF  );
    NT35510_Write_Data ( 0x98  );
    NT35510_Write_Data ( 0x06  );

    /* GIP 1(BCh)  */
    DEBUG_DELAY ();
    NT35510_Write_Cmd(0xBC);
    NT35510_Write_Data(0x01);
    NT35510_Write_Data(0x0E);
    NT35510_Write_Data(0x61);
    NT35510_Write_Data(0xFB);
    NT35510_Write_Data(0x10);
    NT35510_Write_Data(0x10);
    NT35510_Write_Data(0x0B);
    NT35510_Write_Data(0x0F);
    NT35510_Write_Data(0x2E);
    NT35510_Write_Data(0x73);
    NT35510_Write_Data(0xFF);
    /* 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 NT35510_Write_Cmd 发送代码，然后使用 NT35510_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《NT35510-Data Sheet》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*。



ILI9806G-Data Sheet... ×

Translat

ILITEK
I Love Innovation

a-Si TFT LCD Single Chip Driver
480 (RGB) x 864 Resolution and 16.7M-color

ILI9806G

4.3.42. EXTC Command Set enable register (FFh)

FFh	ENEXTC (EXTC command set enable register)												
	DCX	RDX	WRX	D [23:8]	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	1	1	1	1	1	1	1	1	FFh
1st Parameter	1	1	↑	XX	1	1	1	1	1	1	1	1	FFh
2nd Parameter	1	1	↑	XX	1	0	0	1	1	0	0	0	98h
3rd Parameter	1	1	↑	XX	0	0	0	0	0	1	1	0	06h
Description	Set the register, 1st Parameter = FFh, 2nd Parameter = 98h, 3rd Parameter = 06h to enable "EXTC command set". Set the register, 1st Parameter = FFh, 2nd Parameter = 98h, 3rd Parameter = any value except 06h to disable "EXTC command set".												
Restriction													

图 26-28 在《NT35510 数据手册》中 CFh 命令的部分说明

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 9: 代码清单 26-9 设置液晶显示窗口

```
/********** ILI9806 命令 **********************************/
#define      CMD_SetCoordinateX              0x2A        //设置 X 坐标
#define      CMD_SetCoordinateY              0x2B        //设置 Y 坐标

/**
* @brief  在 NT35510 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
```

(下页继续)

(续上页)

```
*/
void NT35510_OpenWindow ( uint16_t usX, uint16_t usY,
uint16_t usWidth, uint16_t usHeight )
{
   NT35510_Write_Cmd ( CMD_SetCoordinateX );           /* 设置 X 坐标 */
   NT35510_Write_Data ( usX >> 8  );  /* 先高 8 位，然后低 8 位 */
   NT35510_Write_Data ( usX & 0xff  );  /* 设置起始点和结束点 */
   NT35510_Write_Data ( ( usX + usWidth - 1 ) >> 8  );
   NT35510_Write_Data ( ( usX + usWidth - 1 ) & 0xff  );

   NT35510_Write_Cmd ( CMD_SetCoordinateY );            /* 设置 Y 坐标 */
   NT35510_Write_Data ( usY >> 8  );
   NT35510_Write_Data ( usY & 0xff  );
   NT35510_Write_Data ( ( usY + usHeight - 1 ) >> 8 );
   NT35510_Write_Data ( ( usY + usHeight - 1) & 0xff );
}
```

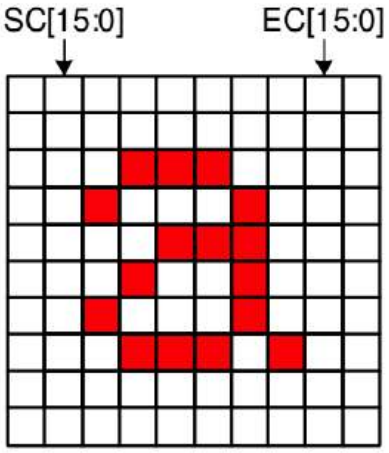
2Ah	CASET (Column Address Set)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	0	0	1	0	1	0	1	0	2Ah
1st Parameter	1	1	↑	XX	SC15	SC14	SC13	SC12	SC11	SC10	SC9	SC8	Note1
2ndParameter	1	1	↑	XX	SC7	SC6	SC5	SC4	SC3	SC2	SC1	SC0	
3rd Parameter	1	1	↑	XX	EC15	EC14	EC13	EC12	EC11	EC10	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《NT35510 数据手册》）

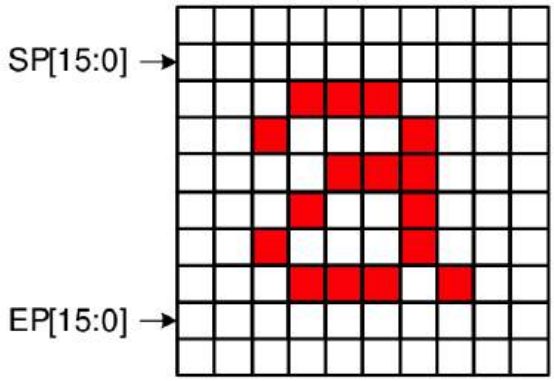
2Bh	PASET (Page Address Set)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	0	0	1	0	1	0	1	1	2Bh
1st Parameter	1	1	↑	XX	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	Note1
2nd Parameter	1	1	↑	XX	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	
3rd Parameter	1	1	↑	XX	EP15	EP14	EP13	EP12	EP11	EP10	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《NT35510 数据手册》）

代码中定义的 NT35510_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。NT35510_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。

27.7.2.2.6 发送像素数据

调用上面的 NT35510_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 10: 代码清单 26_10 发送像素数据

```
#define      CMD_SetPixel                    0x2C        //填充像素

/**
* @brief  在 NT35510 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 NT35510_FillColor ( uint32_t ulAmout_Point,
uint16_t usColor )
{
   uint32_t i = 0;

   /* memory write */
   NT35510_Write_Cmd ( CMD_SetPixel );

   for ( i = 0; i < ulAmout_Point; i ++ )
      NT35510_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《NT35510 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 NT35510_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 NT35510_Write_Cmd 发送一次命令代码，接着使用 for 循环调用 NT35510_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 NT35510_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。

27.7.2.2.7 绘制单个像素点

利用前面的 NT35510_OpenWindow 和 NT35510_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 11: 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 NT35510 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 NT35510_SetCursor ( uint16_t usX, uint16_t usY )
{
   NT35510_OpenWindow ( usX, usY, 1, 1 );
}
```

(下页继续)

(续上页)

```
/**
* @brief  对 NT35510 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 NT35510_SetPointPixel ( uint16_t usX, uint16_t usY )
{
   if ( ( usX < LCD_X_LENGTH ) && ( usY < LCD_Y_LENGTH ) ) {
      NT35510_SetCursor ( usX, usY );

      NT35510_FillColor ( 1, CurrentTextColor );
   }
}
```

NT35510_SetPointPixel 函数直接调用了 NT35510_SetCursor（实质上是 NT35510_OpenWindow 函数的封装）设置单个像素点的绘制窗口，然后调用 NT35510_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 NT35510_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 NT35510_OpenWindow 和 NT35510_FillColor 制作的绘制矩形操作见*代码清单 26_12*。

列表 12: 代码清单 26-12 绘制矩形

```
/**
* @brief  在 NT35510 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
```

(下页继续)

(续上页)

```
* @param  usWidth: 矩形的宽度（单位：像素）
* @param  usHeight: 矩形的高度（单位：像素）
* @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 NT35510_DrawRectangle ( uint16_t usX_Start, uint16_t usY_Start,
                uint16_t usWidth, uint16_t usHeight, uint8_t ucFilled )
{
   if ( ucFilled ) {
      NT35510_OpenWindow ( usX_Start, usY_Start, usWidth, usHeight );
      NT35510_FillColor ( usWidth * usHeight ,CurrentTextColor);
   } else {
      NT35510_DrawLine ( usX_Start, usY_Start,
                           usX_Start + usWidth - 1, usY_Start );
      NT35510_DrawLine ( usX_Start, usY_Start + usHeight - 1,
                     usX_Start + usWidth - 1, usY_Start + usHeight - 1 );
      NT35510_DrawLine ( usX_Start, usY_Start, usX_Start,
                           usY_Start + usHeight - 1 );
      NT35510_DrawLine ( usX_Start + usWidth - 1, usY_Start,
                     usX_Start + usWidth - 1, usY_Start + usHeight - 1 );
   }
}
```

NT35510_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 NT35510_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 NT35510_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 NT35510_DrawLine 函数绘制，NT35510_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 NT35510_SetPointPixel 函数一点一点地绘制成完整的直线。

关于 NT35510_DrawLine 画线函数、NT35510_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。

27.7.2.2.8 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 13: 代码清单 26-13 设置液晶的扫描方向

```
#define        NT35510_LESS_PIXEL          480      //液晶屏较短方向的像素宽度
#define        NT35510_MORE_PIXEL         854      //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 NT35510_GramScan 函数设置方向时会自动更改
uint16_t LCD_X_LENGTH = NT35510_LESS_PIXEL;
uint16_t LCD_Y_LENGTH = NT35510_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 NT35510_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 NT35510 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，
```

(下页继续)

(续上页)

```
*          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
*
*  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 NT35510_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 = NT35510_LESS_PIXEL;
       LCD_Y_LENGTH =  NT35510_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 = NT35510_MORE_PIXEL;
       LCD_Y_LENGTH =  NT35510_LESS_PIXEL;
    }

    //0x36 命令参数的高 3 位可用于设置 GRAM 扫描方向
```

(下页继续)

(续上页)

```
    NT35510_Write_Cmd ( 0x36 );
    NT35510_Write_Data ( 0x08 |(ucOption<<5));//根据 ucOption 的值设置 LCD 参数,
共 0-7 种模式
    NT35510_Write_Cmd ( CMD_SetCoordinateX );
    NT35510_Write_Data ( 0x00 );     /* x 起始坐标高 8 位 */
    NT35510_Write_Data ( 0x00 );     /* x 起始坐标低 8 位 */
    NT35510_Write_Data(((LCD_X_LENGTH-1)>>8)&0xFF );/* x 结束坐标高 8 位 */
    NT35510_Write_Data((LCD_X_LENGTH-1)&0xFF );     /* x 结束坐标低 8 位 */

    NT35510_Write_Cmd ( CMD_SetCoordinateY );
    NT35510_Write_Data ( 0x00 );     /* y 起始坐标高 8 位 */
    NT35510_Write_Data ( 0x00 );     /* y 起始坐标低 8 位 */
    NT35510_Write_Data(((LCD_Y_LENGTH-1)>>8)&0xFF );/*y 结束坐标高 8 位 */
    NT35510_Write_Data((LCD_Y_LENGTH-1)&0xFF );    /*y 结束坐标低 8 位 */

    /* write gram start */
    NT35510_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《NT35510 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。NT35510_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 NT35510_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。

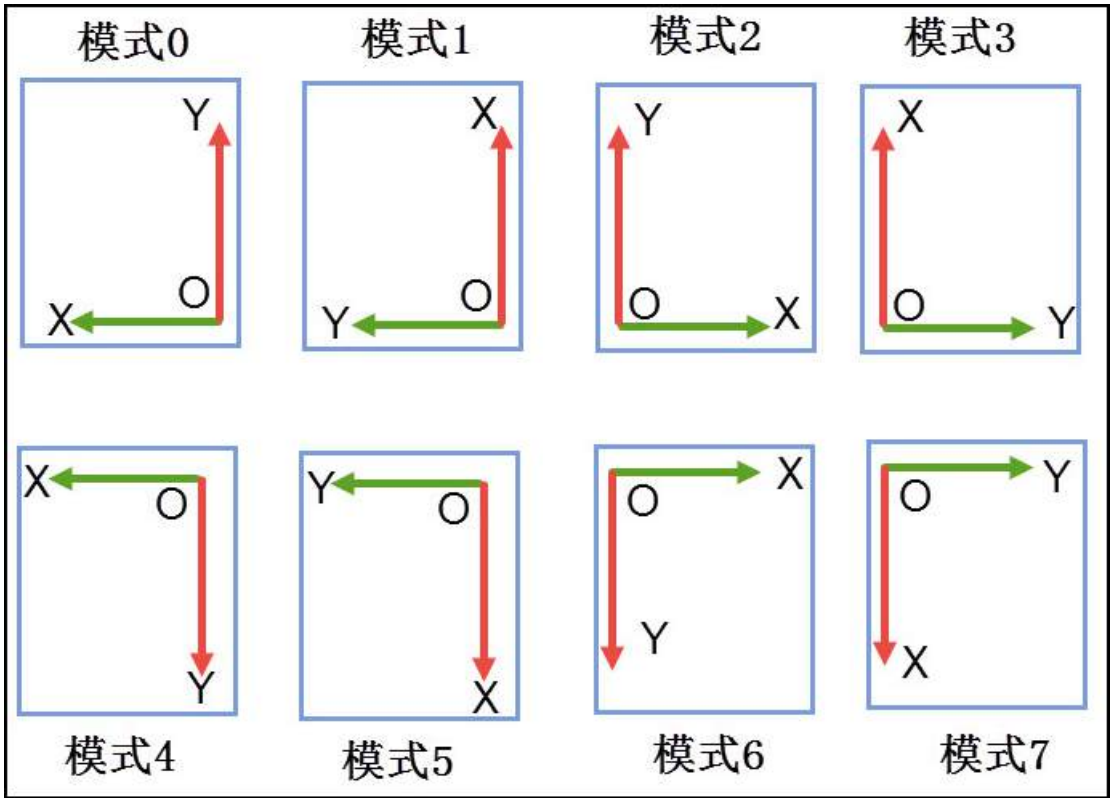


图 26-33 液晶屏的 8 种扫描模式

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。

27.7.2.2.9 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 NT35510_Init 函数，见*代码清单 26_14*。

列表 14: 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 NT35510_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  NT35510 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 NT35510_Init ( void )
{
   NT35510_GPIO_Config ();
   NT35510_FSMC_Config ();

   NT35510_Rst ();
   NT35510_REG_Config ();

   //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
   NT35510_GramScan(LCD_SCAN_MODE);

   NT35510_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */
   NT35510_BackLed_Control ( ENABLE );       //点亮 LCD 背光灯
}
```

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 NT35510_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。

27.7.2.3 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 15: 代码清单 26-15 main 函数

```
int main(void)
{
   /* 系统时钟初始化成 168MHz */
   SystemClock_Config();
   /* LED 端口初始化 */
   LED_GPIO_Config();
   /* 初始化串口 */
   DEBUG_USART_Config();
   NT35510_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
   NT35510_GramScan ( 6 );
   while ( 1 ) {
         LCD_Test();
   }
}
```

程序中，调用了 NT35510_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 16: 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(&Font16x32);
    LCD_SetColors(RED,BLACK);

    NT35510_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */
    /******** 显示字符串示例 *******/
    NT35510_DispStringLine_EN(LINE(0),"YH 4.3 inch LCD");
    NT35510_DispStringLine_EN(LINE(2),"resolution:480x800px");
    NT35510_DispStringLine_EN(LINE(3),"LCD driver:NT35510");
    NT35510_DispStringLine_EN(LINE(4),"Touch driver:GT917S");

    /******** 显示变量示例 *******/
    LCD_SetTextColor(GREEN);

    /* 使用 cHAL 库把变量转化成字符串 */
    sprintf(dispBuff,"Count : %d ",testCNT);
    NT35510_ClearLine(LINE(7));  /* 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 */
    NT35510_DispStringLine_EN(LINE(7),dispBuff);

    /******* 显示图形示例 ******/
```

(下页继续)

(续上页)

```
    /* 画直线 */

    NT35510_ClearLine(LINE(7));/* 清除单行文字 */
    LCD_SetTextColor(BLUE);

    NT35510_DispStringLine_EN(LINE(7),"Draw line:");

    LCD_SetTextColor(RED);
    NT35510_DrawLine(50,270,420,275);
    NT35510_DrawLine(50,300,420,375);

    LCD_SetTextColor(GREEN);
    NT35510_DrawLine(50,370,420,475);
    NT35510_DrawLine(50,400,420,475);

    LCD_SetTextColor(BLUE);
    NT35510_DrawLine(50,420,420,325);
    NT35510_DrawLine(50,450,420,395);

    Delay(0x2FFFFFF);

    NT35510_Clear(0,32*7,LCD_X_LENGTH,LCD_Y_LENGTH-32*7); /* 清屏，显示全黑 */
    /* 省略部分内容 */

    Delay(0x2FFFFFF);

    NT35510_Clear(0,32*7,LCD_X_LENGTH,LCD_Y_LENGTH-32*7); /* 清屏，显示全黑 */
}
```

27.7.2.4 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 17: 代码清单 26-17 液晶坐标方向演示例程的 main 函数

```
/**
   * @brief  主函数
   * @param  无
   * @retval 无
   */
int main ( void )
{   SystemClock_Config();
   LED_GPIO_Config();

   NT35510_Init ();          //LCD 初始化

   Debug_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");

   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;
```

(下页继续)

(续上页)

```
    char dispBuff[100];

    //轮流展示各个方向模式
    for (i=0; i<8; i++) {
        LCD_SetFont(&Font16x32);
        LCD_SetColors(RED,BLACK);

        NT35510_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
        NT35510_GramScan ( i );

        sprintf(dispBuff,"o%d. X --->",i);
        NT35510_DispStringLine_EN(LINE(0),dispBuff);//沿 X 方向显示文字

        sprintf(dispBuff,"o%d.Y|V",i);
        NT35510_DispString_EN_YDir(0,0,dispBuff);//沿 Y 方向显示文字

        Delay(0x2FFFFFF);

        //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。

27.7.3 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 28 章　LCD—液晶显示中英文

本章参考资料：《STM32F4xx-中文参考手册》、《STM32F4xx 数据手册》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

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。

28.1 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。

28.1.1 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

下页继续

表 1 – 续上页

1	1	SOH(start of headline)	标题开始
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

下页继续

表 1 – 续上页

30	1E	RS (record separator)	记录分离符
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	‘		

下页继续

表 2 – 续上页

49	31	1	97	61	a		
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	{		

下页继续

表 2 – 续上页

76	4C	L	124	7C			
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。

28.1.2 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。

28.1.2.1 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

28.1.2.1.1 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。

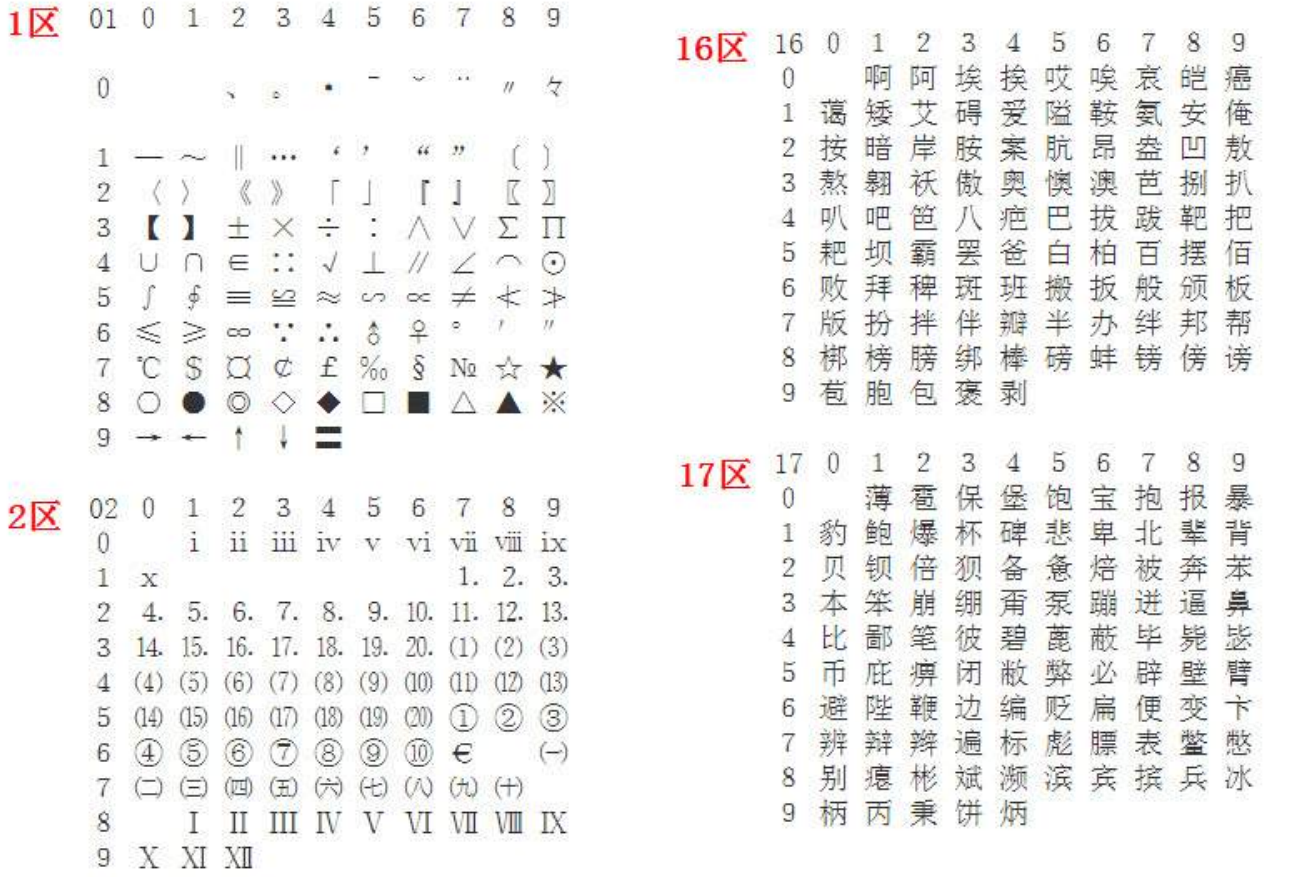


图 27-1 GB2312 的部分区位码

28.1.2.2 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

28.1.2.3 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		

28.1.2.4 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”。

28.1.3 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)。

28.1.4 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)。

28.1.5 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。

28.1.6 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。

28.1.7 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。

28.2 什么是字模？

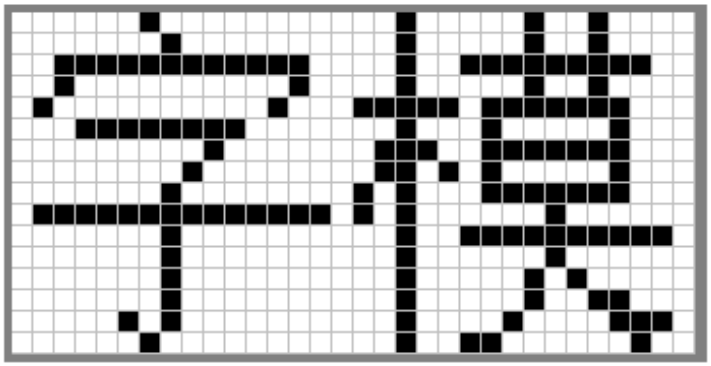
有了编码，我们就能在计算机中处理、存储字符了，但是如果计算机处理完字符后直接以编码的形式输出，人类将难以识别。来，在 2 秒内告诉我 ASCII 编码的“0x25”表示什么字符？不容易

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。

28.2.1 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,
};
```

28.2.2 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。



28.2.3 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。

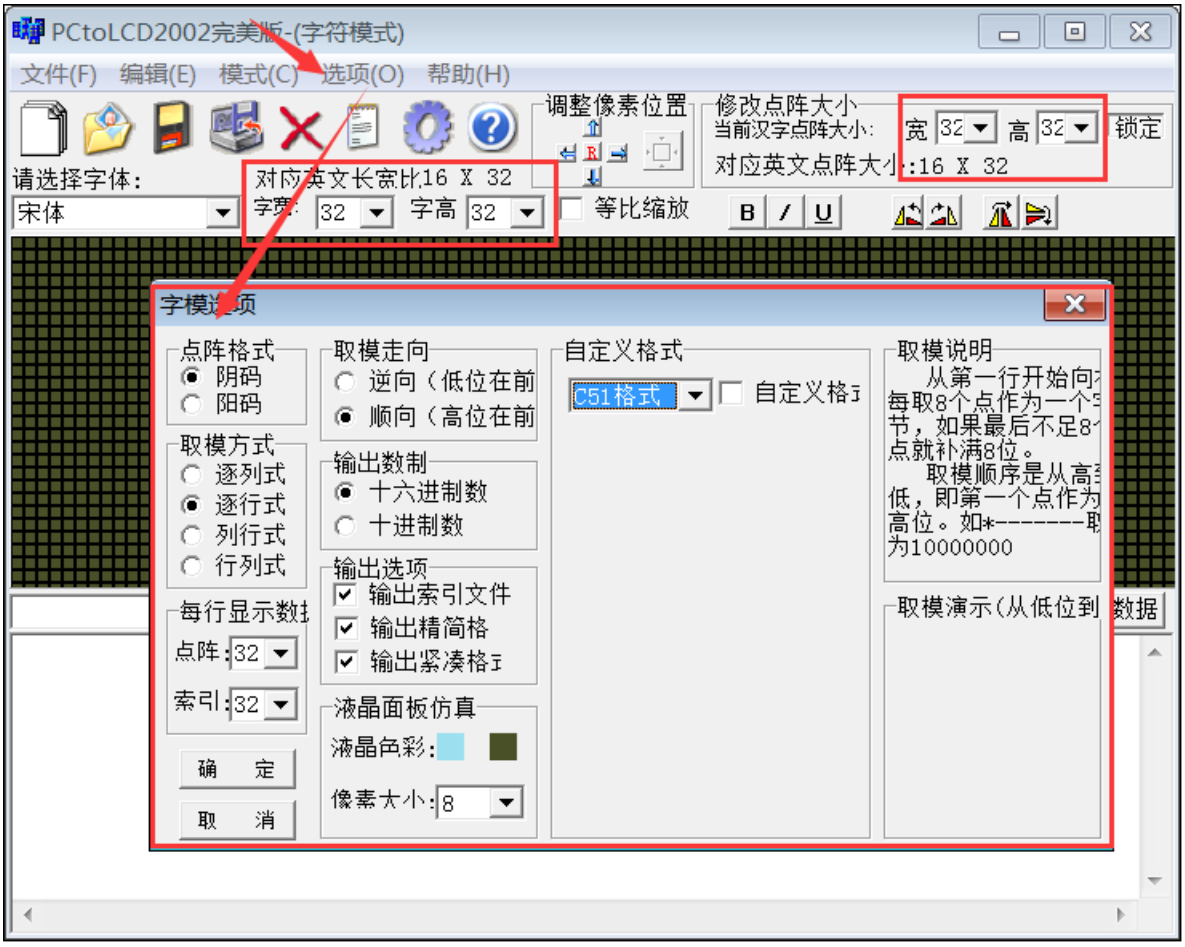
(1) 配置字模格式

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。

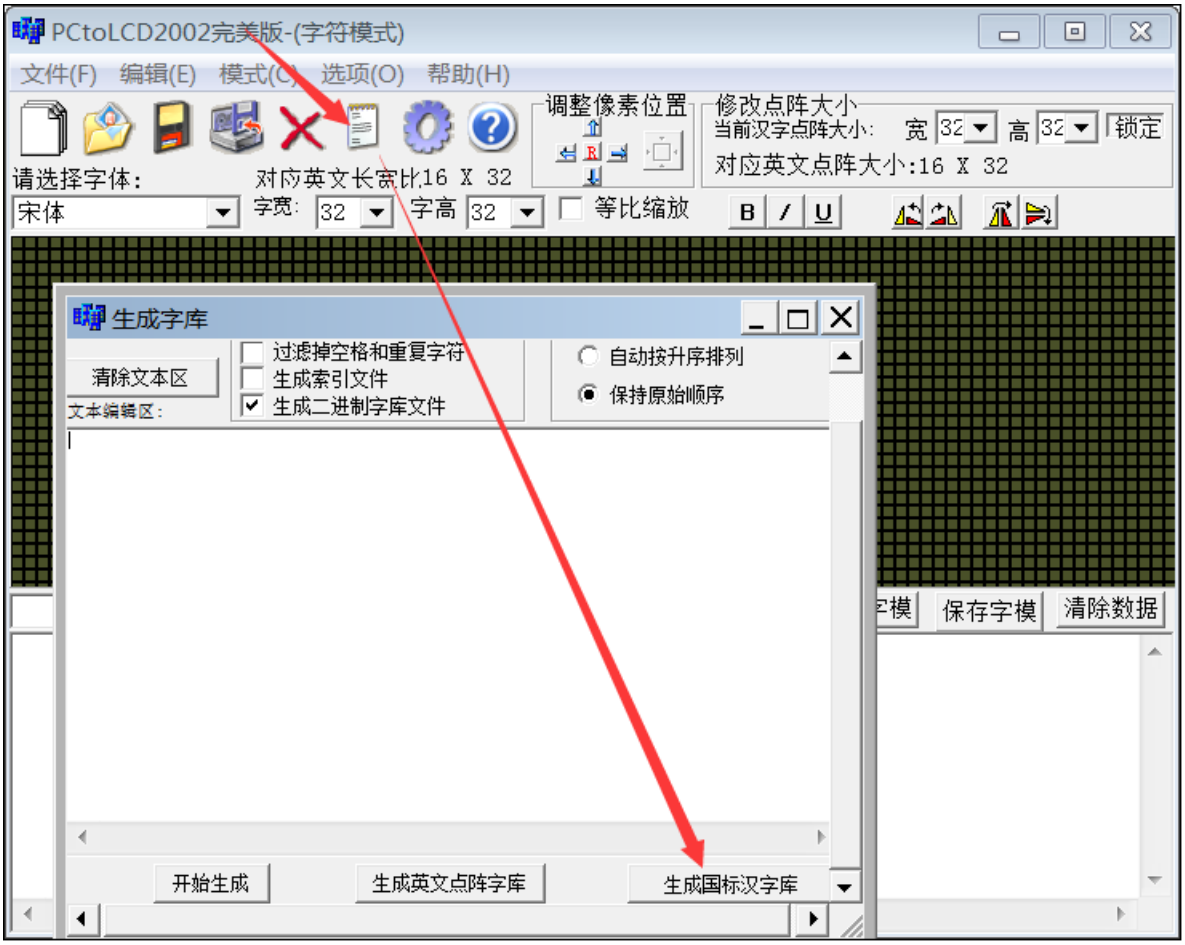
- 选项“取模方式”是指字模图形的扫描方向，修改这部分的设置后，选项框的右侧会有相应的说明及动画显示，这里我们依然按前文介绍的字模类型，把它配置成“逐行式”
- 选项“每行显示的数据”里我们把点阵和索引都配置成 32，设置这个点阵的像素大小为 32x32。

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_H3232.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。



28.2.4 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_H3232.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 32x32/8=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 128 字节存储的是它的字模数据；同理，128-256 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))*32*32/8

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。

28.2.5 存储字模文件

上面生成的《GB2312_H3232.FON》文件的大小为 1023KB，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。

28.3 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。

28.3.1 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_H3232.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_H3232.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。

28.3.2 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”（即上一章节的例程）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_NT35510_lcd.c”及“bsp_NT35510_lcd.h”中。

28.3.2.1 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。

28.3.2.2 代码分析

28.3.2.2.1 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 24x48、16x32、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 16x32 大小 (fonts.c 文件)

```
/*
* 常用 ASCII 表，偏移量 32，大小:32（高度）* 16 （宽度）
*/
//@conslons 字体，阴码点阵格式，逐行顺向取摸
const uint8_t ASCII16x32_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, 0x00,0x00, 0x00,0x00, 0x00,0x00,
    0x00,0x00, 0x00,0x00, 0x00,0x00, 0x00,0x00,
    0x00,0x00, 0x00,0x00, 0x03,0x80, 0x03,0x80,
    0x01,0x80, 0x01,0x80, 0x01,0x80, 0x01,0x80,
    0x01,0x80, 0x01,0x80, 0x01,0x80, 0x01,0x80,
    0x01,0x80, 0x01,0x80, 0x01,0x80, 0x01,0x80,
    0x00,0x00, 0x00,0x00, 0x03,0x80, 0x03,0xc0,

    /* 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 ASCII16x32_Table 数组是 16x32 大小的 ASCII 字库。

28.3.2.2.2 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 */
```

(下页继续)

(续上页)

```
} sFONT;

sFONT Font8x16 = {
   ASCII8x16_Table,
   8, /* 字模宽 */
   16, /* 字模高 */
};

sFONT Font16x32 = {
   ASCII16x32_Table,
   16, /* 字模宽 */
   32, /* 字模高 */
};

sFONT Font24x48 = {
   ASCII24x48_Table,
   24, /* 字模宽 */
   48, /* 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、Font16x32 之类的变量，方便显示时寻址。

28.3.2.2.3 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、Font16x32 之类的变量。

28.3.2.2.4 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 NT35510 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 NT35510_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
   NT35510_OpenWindow ( usX, usY, LCD_Currentfonts->Width, LCD_Currentfonts-
↪>Height);

   NT35510_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) )
                  NT35510_Write_Data ( CurrentTextColor );
```

(下页继续)

(续上页)

```
                else
                    NT35510_Write_Data ( CurrentBackColor );
            }
        }
}
```

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 NT35510_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 NT35510_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 NT35510_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*。

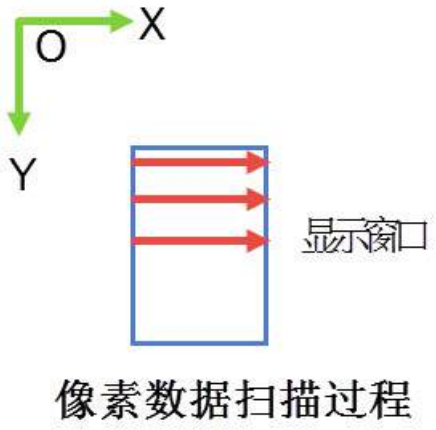


图 27-6 设置显示窗口后的像素数据扫描过程

(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 NT35510_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 NT35510_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。

28.3.2.2.5 显示字符串

对 NT35510_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 NT35510 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 NT35510_DispString_EN (  uint16_t usX ,uint16_t usY,  char * pStr )
{
   while ( * pStr != '\0' ) {
if ( ( usX - NT35510_DispWindow_X_Star + LCD_Currentfonts->Width )> LCD_X_
↪LENGTH) {
            usX = NT35510_DispWindow_X_Star;
            usY += LCD_Currentfonts->Height;
         }

if ( ( usY -NT35510_DispWindow_Y_Star + LCD_Currentfonts->Height )>LCD_Y_
↪LENGTH ) {
            usX = NT35510_DispWindow_X_Star;
            usY = NT35510_DispWindow_Y_Star;
         }

         NT35510_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 NT35510_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“NT35510_DispString_EN (10，20,”test”)”这样的格式在液晶屏上直接显示一串字符。

28.3.2.2.6 使用宏计算 Y 坐标

使用 NT35510_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 NT35510_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 NT35510 显示器上显示中英文字符串
* @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 NT35510_DispStringLine_EN_CH (  uint16_t line, char * pStr )
{
   uint16_t usCh;
   uint16_t usX = 0;

   while( * pStr != '\0' )
   {
      if ( * pStr <= 126 )                    //英文字符
      {
         if ( ( usX - NT35510_DispWindow_X_Star + LCD_Currentfonts->Width )␣
↪> LCD_X_LENGTH )
         {
            usX = NT35510_DispWindow_X_Star;
            line += LCD_Currentfonts->Height;
         }

         if ( ( line - NT35510_DispWindow_Y_Star + LCD_Currentfonts->Height␣
↪) > LCD_Y_LENGTH )
         {
            usX = NT35510_DispWindow_X_Star;
            line = NT35510_DispWindow_Y_Star;
         }

      NT35510_DispChar_EN ( usX, line, * pStr );

         usX +=  LCD_Currentfonts->Width;

      pStr ++;

      }
```

(下页继续)

(续上页)

```
      else                                    //汉字字符
      {
         if ( ( usX - NT35510_DispWindow_X_Star + WIDTH_CH_CHAR ) > LCD_X_
↪LENGTH )
         {
            usX = NT35510_DispWindow_X_Star;
            line += HEIGHT_CH_CHAR;
         }

         if ( ( line - NT35510_DispWindow_Y_Star + HEIGHT_CH_CHAR ) > LCD_Y_
↪LENGTH )
         {
            usX = NT35510_DispWindow_X_Star;
            line = NT35510_DispWindow_Y_Star;
         }

         usCh = * ( uint16_t * ) pStr;

         usCh = ( usCh << 8 ) + ( usCh >> 8 );

         NT35510_DispChar_CH ( usX, line, usCh );

         usX += WIDTH_CH_CHAR;

         pStr += 2;            //一个汉字两个字节

   }

}
}
```

本函数主体与 NT35510_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“NT35510_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 “NT35510_DispString_EN (10,LINE(1),” test”); ” 同样也是可以的，并且可指定 X 坐标。

28.3.2.2.7 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 NT35510 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 NT35510_Clear ( uint16_t usX, uint16_t usY, uint16_t usWidth, uint16_t␣
↪usHeight )
{
   NT35510_OpenWindow ( usX, usY, usWidth, usHeight );

   NT35510_FillColor ( usWidth * usHeight, CurrentBackColor );

}
/**
   * @brief  清除某行文字
```

(下页继续)

(续上页)

```
    * @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 NT35510_ClearLine(uint16_t Line)
{/* 清屏，显示背景颜色 */
   NT35510_Clear(0,Line,LCD_X_LENGTH,((sFONT *)LCD_GetFont())->Height);
}
```

代码中的 NT35510_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。

28.3.2.2.8 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 )
{
SystemClock_Config();

LED_GPIO_Config();
```

(下页继续)

(续上页)

```
    //LCD 初始化
    NT35510_Init ();

    /* USART config */
    DEBUG_USART_Config();


    printf("\r\n ********** 液晶屏中文显示程序（字库在外部 FLASH）********** \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
    NT35510_GramScan ( 6 );

    Printf_Charater();

    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(&Font16x32);
   LCD_SetColors(RED,BLACK);

   NT35510_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */
   /******** 显示字符串示例 *******/
   NT35510_DispStringLine_EN(LINE(0),"BH 4.5 inch LCD");
   NT35510_DispStringLine_EN(LINE(2),"resolution:480x854px");
   NT35510_DispStringLine_EN(LINE(3),"LCD driver:NT35510");
   NT35510_DispStringLine_EN(LINE(4),"Touch driver:GT5688");

   /******** 显示变量示例 *******/
   LCD_SetTextColor(GREEN);

   /* 使用 cHAL 库把变量转化成字符串 */
   sprintf(dispBuff,"Count : %d ",testCNT);
   NT35510_ClearLine(LINE(7));  /* 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 */
   NT35510_DispStringLine_EN(LINE(7),dispBuff);
   /* 以下省略其它液晶演示示例 */
```

(下页继续)

(续上页)

```
}
```

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。

28.3.3 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_NT35510_lcd.c”及“bsp_NT35510_lcd.h”中。

28.3.3.1 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。

28.3.3.2 代码分析

28.3.3.2.1 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_H3232.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_NT35510_ldc.c 文件)

```
#define      WIDTH_CH_CHAR                  32      //中文字符宽度
#define      HEIGHT_CH_CHAR                 32      //中文字符高度

//缓存读取回来的字模数据
static uint8_t ucBuffer [ WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8 ];

/**
* @brief  在 NT35510 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
* @retval 无
*/
void NT35510_DispChar_CH ( uint16_t usX, uint16_t usY, uint16_t usChar )
{
   uint8_t rowCount, bitCount;
   uint32_t usTemp;

   //  占用空间太大，改成全局变量
   //  uint8_t ucBuffer [ WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8 ];
```

(下页继续)

(续上页)

```
    //设置显示窗口
    NT35510_OpenWindow ( usX, usY, WIDTH_CH_CHAR, HEIGHT_CH_CHAR );

    NT35510_Write_Cmd ( CMD_SetPixel );

    //取字模数据
    GetGBKCode ( ucBuffer, usChar );

    for ( rowCount = 0; rowCount < HEIGHT_CH_CHAR; rowCount++ ) {
    /* 取出四个字节的数据，在 lcd 上即是一个汉字的一行 */
    usTemp = ucBuffer [ rowCount * 4 ];
    usTemp = ( usTemp << 8 );
    usTemp |= ucBuffer [ rowCount * 4 + 1 ];
    usTemp = ( usTemp << 8 );
    usTemp |= ucBuffer [ rowCount * 4 + 2 ];
    usTemp = ( usTemp << 8 );
    usTemp |= ucBuffer [ rowCount * 4 + 3 ];

    for ( bitCount = 0; bitCount < WIDTH_CH_CHAR; bitCount ++ ) {
    if ( usTemp & ( 0x80000000 >> bitCount ) )  //高位在前
     NT35510_Write_Data ( CurrentTextColor );
    else
     NT35510_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 NT35510_OpenWindow 和 NT35510_Write_Cmd 来设置显示窗口并发送显示像素命令。

(2) 使用量 macGetGBKCode 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。

28.3.3.2.2 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 NT35510 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 NT35510_DispString_EN_CH (uint16_t usX , uint16_t usY, char * pStr )
{
   uint16_t usCh;

   while ( * pStr != '\0' ) {
      if ( * pStr <= 126 ) {            //英文字符
if(( usX - NT35510_DispWindow_X_Star + LCD_Currentfonts->Width ) > LCD_X_
↪LENGTH ) {
               usX = NT35510_DispWindow_X_Star;
               usY += LCD_Currentfonts->Height;
```

(下页继续)

(续上页)

```
            }

if(( usY - NT35510_DispWindow_Y_Star + LCD_Currentfonts->Height )> LCD_Y_
↪LENGTH ) {
                usX = NT35510_DispWindow_X_Star;
                usY = NT35510_DispWindow_Y_Star;
            }

            NT35510_DispChar_EN ( usX, usY, * pStr );

            usX +=  LCD_Currentfonts->Width;
            pStr ++;
      }

      else {                                    //汉字字符
if ( ( usX - NT35510_DispWindow_X_Star + WIDTH_CH_CHAR ) > LCD_X_LENGTH ) {
                usX = NT35510_DispWindow_X_Star;
                usY += HEIGHT_CH_CHAR;
            }

if ( ( usY - NT35510_DispWindow_Y_Star + HEIGHT_CH_CHAR ) > LCD_Y_LENGTH ) {
                usX = NT35510_DispWindow_X_Star;
                usY = NT35510_DispWindow_Y_Star;
            }

            usCh = * ( uint16_t * ) pStr;

            usCh = ( usCh << 8 ) + ( usCh >> 8 );

            NT35510_DispChar_CH ( usX, usY, usCh );

            usX += WIDTH_CH_CHAR;
```

(下页继续)

(续上页)

```
            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。

28.3.3.2.3 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   1254*4096


/* 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）
```

(下页继续)

(续上页)

```
#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_H3232 配套的函数

//中文字库存储在 FLASH 的起始地址 :
//GBKCODE_START_ADDRESS 在 fonts.h 文件定义
/**
* @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;
 }

   High8bit= c >> 8;      /* 取高 8 位数据 */
   Low8bit= c & 0x00FF;   /* 取低 8 位数据 */
```

(下页继续)

(续上页)

```
    /*GB2312 公式 */
    pos = ((High8bit-0xa1)*94+Low8bit-0xa1)*WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8;
    SPI_FLASH_BufferRead(pBuffer,GBKCODE_START_ADDRESS+pos,
    WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8); //读取字库数据

    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。

28.3.3.2.4 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:/Font/GB2312_H3232.FON"

/* 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_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;                                /* file objects */
static FATFS fs;  /* Work area (file system object) for logical drives */
static FRESULT res_sd;
static UINT br;                         /* File R/W count */

    /*SD 卡字模路径 */
    #define GBKCODE_FILE_NAME                       "0:/srcdata/GB2312_
↪H3232.FON"
```

(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

         //32*32 大小的汉字 其字模 占用 32*32/8 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。

28.3.3.2.5 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

```
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 */
UINT fnum;                                              /* 文件成功读写数量 */
BYTE ReadBuffer[1024]={0};              /* 读缓冲区 */
```

(下页继续)

(续上页)

```
BYTE WriteBuffer[] =                /* 写缓冲区 */
"";

extern FATFS flash_fs;
extern Diskio_drvTypeDef  SD_Driver;
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
   SystemClock_Config();

   LED_GPIO_Config();

   //LCD 初始化
   NT35510_Init ();

   /* USART config */
   DEBUG_USART_Config();
   //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
   res_sd = f_mount(&fs,"0:",1);

   printf("\r\n ********** 液晶屏中文显示程序（字库在 SD 中）********** \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
   NT35510_GramScan ( 6 );
```

(下页继续)

(续上页)

```
    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(&Font16x32);
    LCD_SetColors(RED,BLACK);

    NT35510_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */
    /******** 显示字符串示例 *******/
    NT35510_DispStringLine_EN_CH(LINE(0),"野火 4.3 寸 LCD 参数：");
    NT35510_DispStringLine_EN_CH(LINE(2),"分辨率：480x800 px");
    NT35510_DispStringLine_EN_CH(LINE(3),"NT35510 液晶驱动");
```

(下页继续)

(续上页)

```
    NT35510_DispStringLine_EN_CH(LINE(4),"GT917S 触摸屏驱动");

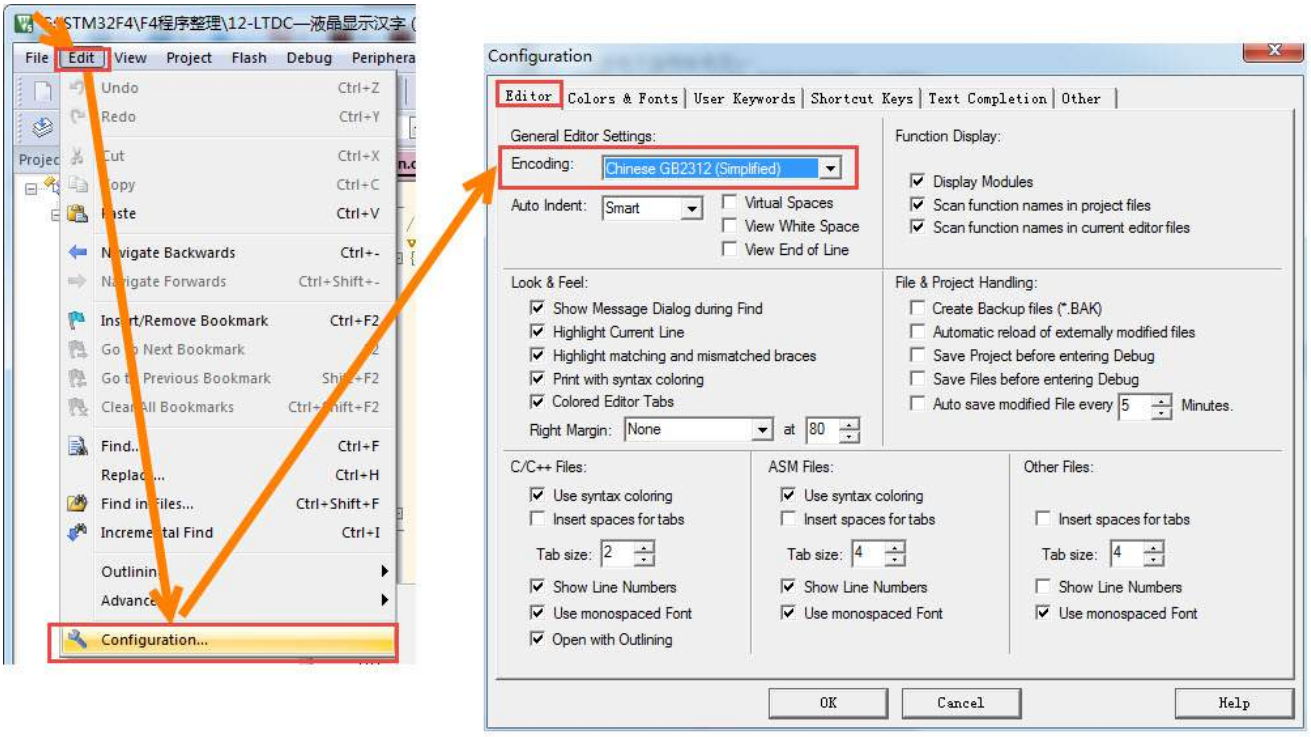
    /******** 显示变量示例 *******/
    LCD_SetTextColor(GREEN);

    /* 使用 cHAL 库把变量转化成字符串 */
    sprintf(dispBuff,"显示变量计数 : %d ",testCNT);
    NT35510_ClearLine(LINE(7));  /* 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 */
    NT35510_DispStringLine_EN(LINE(7),dispBuff);

    /* 以下省略其它液晶演示示例 */
}
```

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？

28.3.4 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。

28.3.4.1 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。

28.3.4.2 代码分析

28.3.4.2.1 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 NT35510_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 > 48) return;                    //字库不允许超过 48 像素
   if (in_width * in_heig == 0) return;
   if (in_width * in_heig > 48*48 ) return;  //限制输入最大 48*48

   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
```

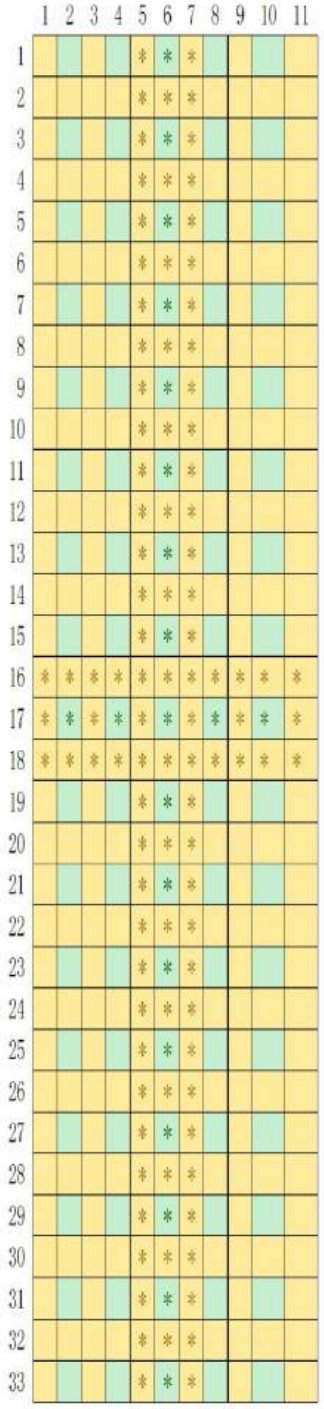
(下页继续)

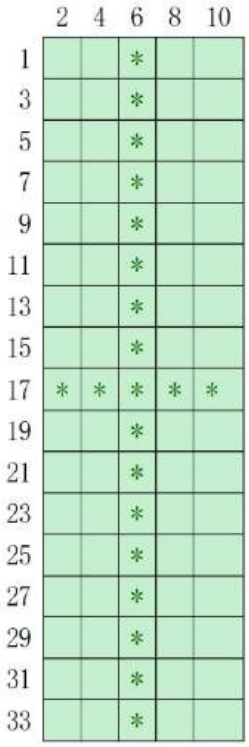
(续上页)

```
            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。

28.3.4.2.2 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 NT35510_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
   NT35510_OpenWindow ( usX, usY, Font_width, Font_Height);

   NT35510_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 )
                NT35510_Write_Data ( CurrentBackColor );
            else
                NT35510_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。

28.3.4.2.3 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 NT35510_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

   //占用空间太大，改成全局变量
   //  uint8_t ucBuffer [ WIDTH_CH_CHAR*HEIGHT_CH_CHAR/8 ];

   while ( *ptr != '\0') {
         /**** 处理换行 *****/
         if ( ( x - NT35510_DispWindow_X_Star + Charwidth ) > LCD_X_LENGTH␣
↪) {
            x = NT35510_DispWindow_X_Star;
            y += Font_Height;
         }

         if ( ( y - NT35510_DispWindow_Y_Star + Font_Height ) > LCD_Y_
↪LENGTH ) {
```

(下页继续)

(续上页)

```
            x = NT35510_DispWindow_X_Star;
            y = NT35510_DispWindow_Y_Star;
        }

        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 32*32
            NT35510_zoomChar(WIDTH_CH_CHAR,HEIGHT_CH_CHAR,Charwidth,
                             Font_Height,(uint8_t *)&ucBuffer,psr,1);
            //显示单个字符
            NT35510_DrawChar_Ex(x,y,Charwidth,Font_Height,
                                (uint8_t*)&zoomBuff,DrawModel);
            x+=Charwidth;
            ptr+=2;
        } else {
            Charwidth = Font_width / 2;
            Ascii = *ptr - 32;
            //使用 16*32 字体缩放字模数据
            NT35510_zoomChar(16,32,Charwidth,Font_Height,
   (uint8_t *)&Font16x32.table[Ascii * Font16x32.Height*Font16x32.Width/8],
↪psr,0);
            //显示单个字符
            NT35510_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 Font16x32 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。

28.3.4.2.4 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(&Font16x32);
    LCD_SetColors(RED,BLACK);

    NT35510_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */

    /******** 显示字符串示例 *******/
    NT35510_DispStringLine_EN_CH(LINE(0),"野火 BH");
    //显示指定大小的字符
    NT35510_DisplayStringEx(0,1*48,48,48,(uint8_t *)"野火 BH",0);
    NT35510_DisplayStringEx(0,2*56,56,56,(uint8_t *)"野火 BH",0);
```

(下页继续)

(续上页)

```
    /******** 显示变量示例 *******/
    LCD_SetTextColor(GREEN);

    /* 使用 cHAL 库把变量转化成字符串 */
    sprintf(dispBuff,"显示变量计数 : %d ",testCNT);
    NT35510_ClearLine(LINE(7));  /* 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 */
    NT35510_DispStringLine_EN(LINE(7),dispBuff);

    /*... 以下部分省略 */
}
```

28.3.5 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 29 章　电容触摸屏—触摸画板

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

关于开发板配套的结构尺寸参数可查阅《4.3 电容触摸屏结构尺寸》，触摸面板配套的触摸控制芯片可查阅《GT917S Datasheet》及《gt91x 编程指南》配套资料获知。。

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。

29.1 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。

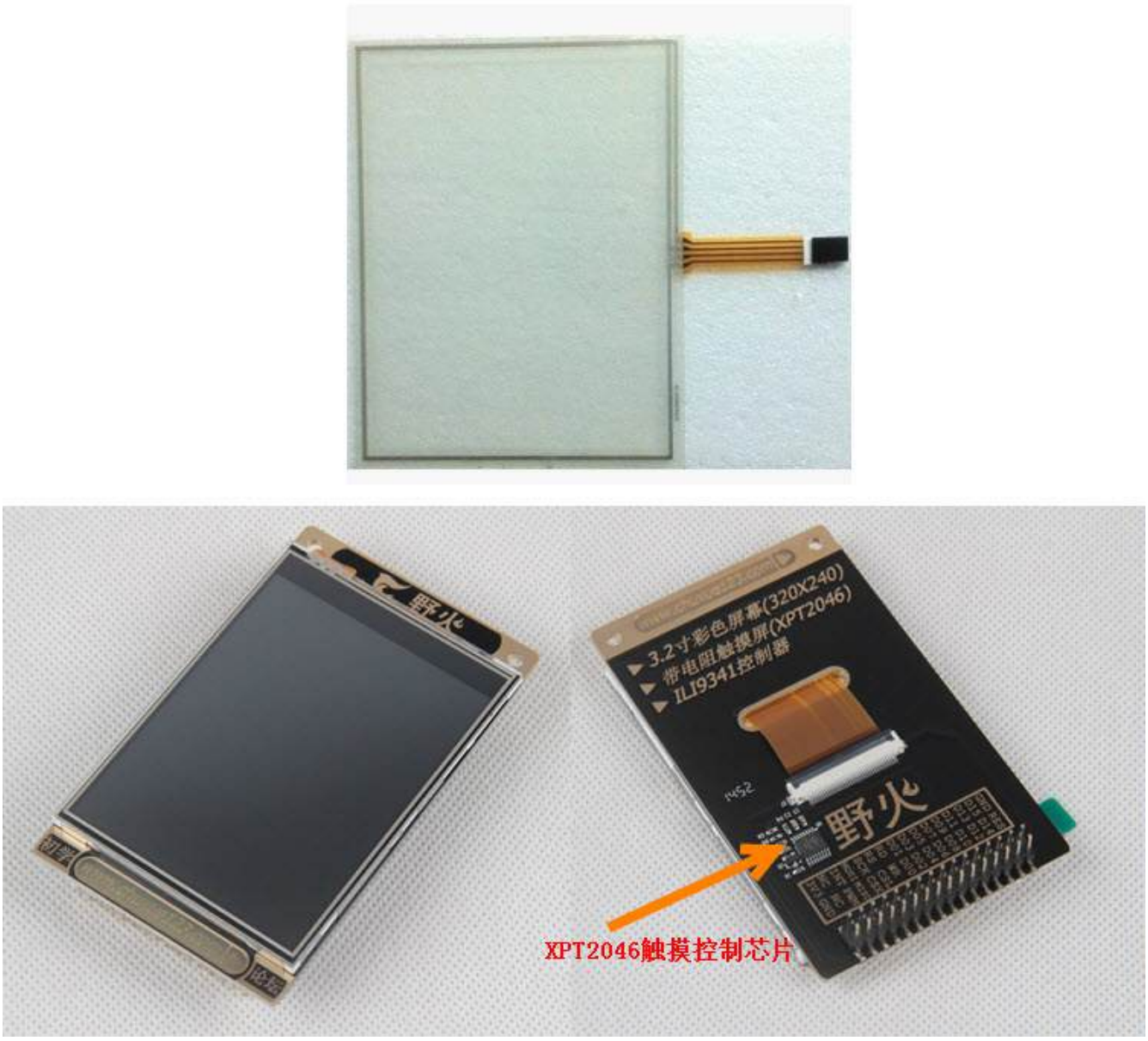


图 28-1 单电阻屏、电阻液晶屏（带触摸控制芯片）



图 28-2 单电容屏、电容液晶屏 (带触摸控制芯片)

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。

29.1.1 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。

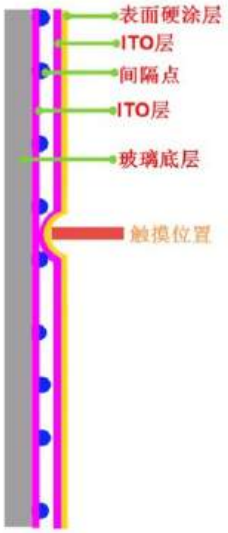


图 28-3 电阻式触摸屏结构

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。

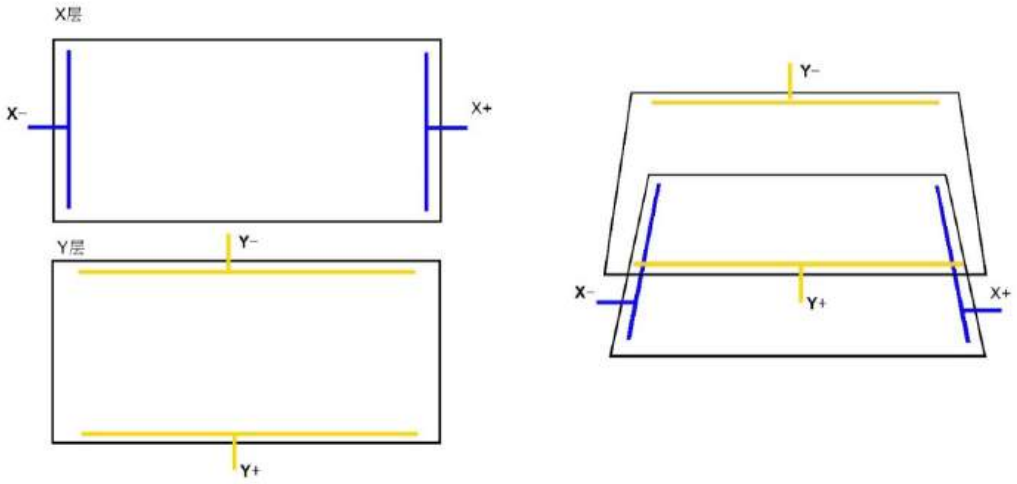


图 28-4 XY 的 ITO 层结构

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:sub:*ref* 之比等于触点 Y 坐标与屏高度之比，从而：

$$y = \frac{V_{Y+}}{V_{\text{ref}}} \times Height$$

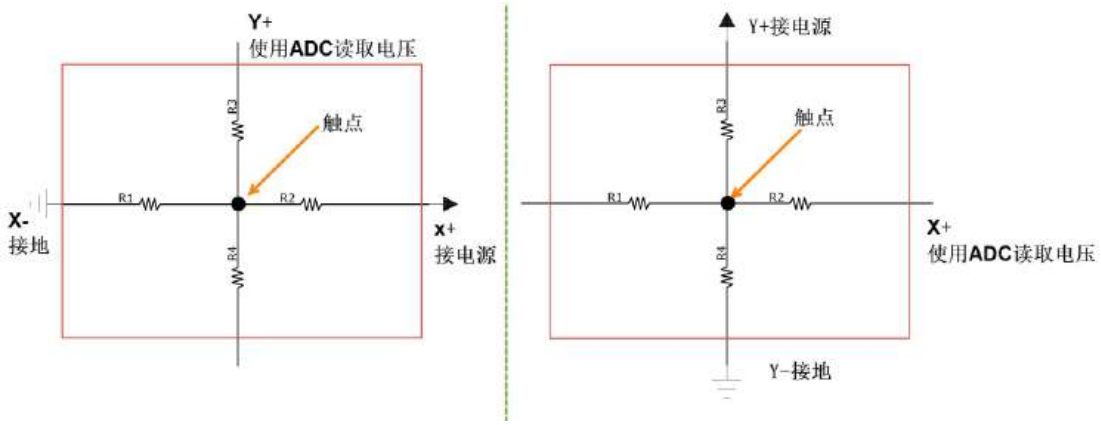


图 28-5 触摸检测等效电路

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。

29.1.2 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 29-6，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）。

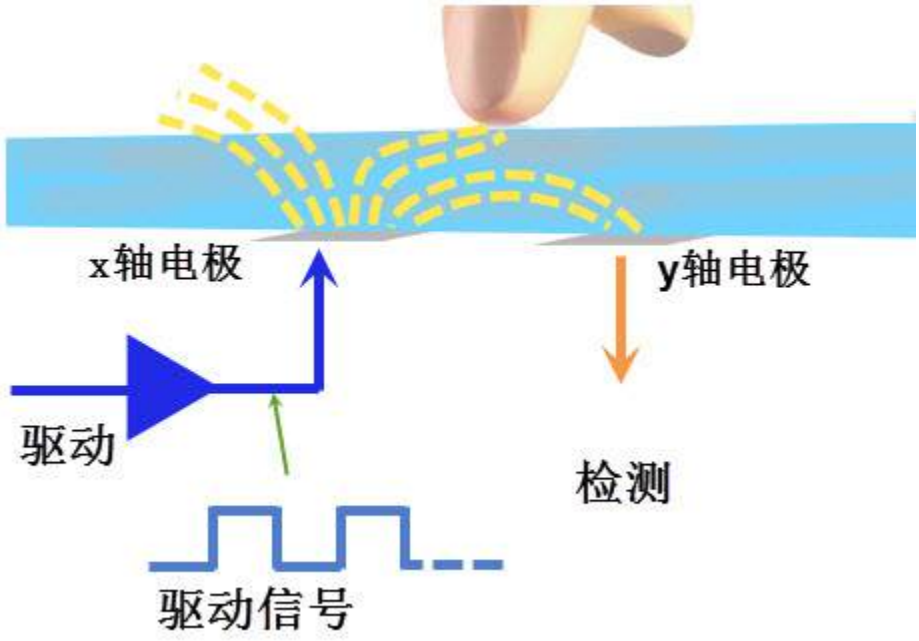


图 28-6 电容触摸屏基本原理

电容屏 ITO 层的结构见图 *28_7*，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。

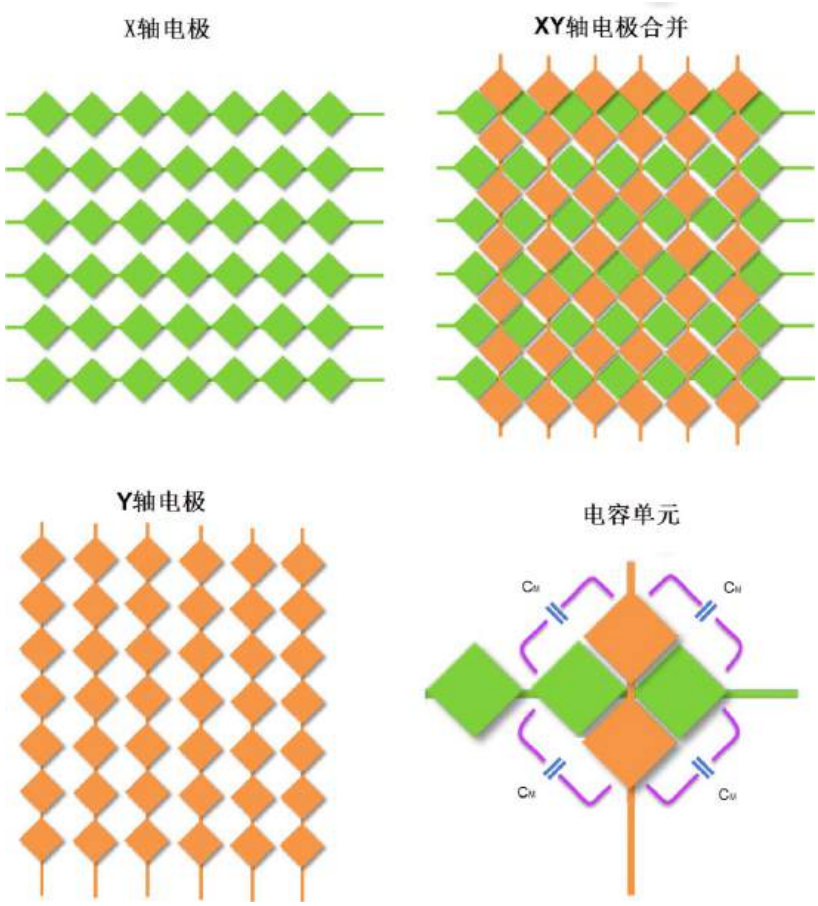


图 28-7 电容触摸屏的 ITO 层结构

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。

29.2 电容触摸屏控制芯片

相对来说，电容屏的坐标检测比电阻屏的要复杂，因而它也有专用芯片用于检测过程，下面我们以本章重点讲述的电容屏使用的触控芯片 GT917S 为例进行讲解，关于它的详细说明可从《gt91x 编程指南》和《GT917S Datasheet》文档了解。

29.2.1 GT917S 芯片的引脚

GT917S 芯片的外观可以图 *28_2* 中找到，其内部结构框图见图 *28_8*。

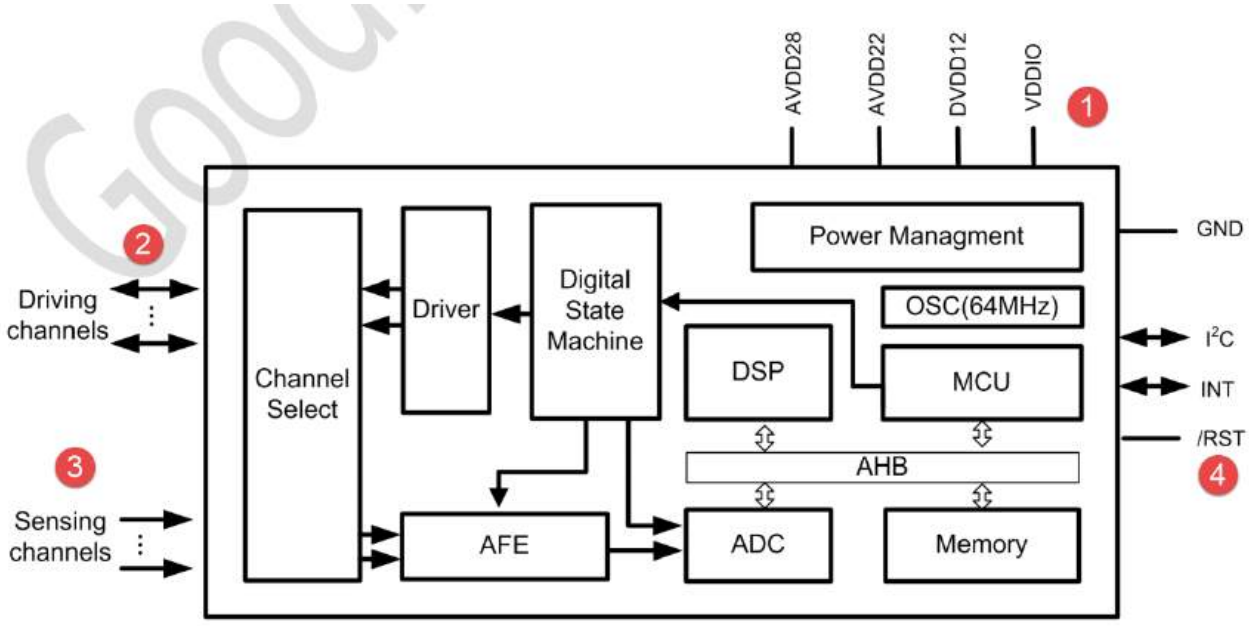


图 28-8 GT917S 结构框图

该芯片对外引出的信号线介绍如下：

表 28-1 GT917S 信号线说明

信号线	说明
AVDD、AVDD18、DVDD12、VDDDIO、GND	电源和地
Driving channels	激励信号输出的引脚，一共有 0-25 个引脚，它连接到电容屏 ITO 层引出的各个激励信号轴
Sensing channels	信号检测引脚，一共有 0-13 个引脚，它连接到电容屏 ITO 层引出的各个电容量检测信号轴
I2C	I2C 通信信号线，包含 SCL 与 SDA，外部控制器通过它与 GT917S 芯片通讯，配置 GT917S 的工作方式或获取坐标信号
INT	中断信号，GT917S 芯片通过它告诉外部控制器有新的触摸事件
/RSTB	复位引脚，用于复位 GT917S 芯片；在上电时还与 INT 引脚配合设置 IIC 通讯的设备地址

若您把电容触摸屏与液晶面板分离开来，在触摸面板的背面，可看到它的边框有一些电路走线，它们就是触摸屏 ITO 层引出的 XY 轴信号线，这些信号线分别引出到 GT917S 芯片的 Driving channels 及 Sensing channels 引脚中。也正是因为触摸屏有这些信号线的存在，所以手机厂商追求的屏幕无边框是比较难做到的。

29.2.2 上电时序与 I2C 设备地址

GT917S 触控芯片有两个备选的 I2C 通讯地址，这是由芯片的上电时序设定的，见图 29-9。上电时序有 Reset 引脚和 INT 引脚生成，若 Reset 引脚从低电电平转变到高电平期间，INT 引脚为高电平的时候，触控芯片使用的 I2C 设备地址为 0x28/0x29(8 位写、读地址)，7 位地址为 0x14；若 Reset 引脚从低电电平转变到高电平期间，INT 引脚一直为低电平，则触控芯片使用的 I2C 设备地址为 0xBA/0xBB(8 位写、读地址)，7 位地址为 0x5D。

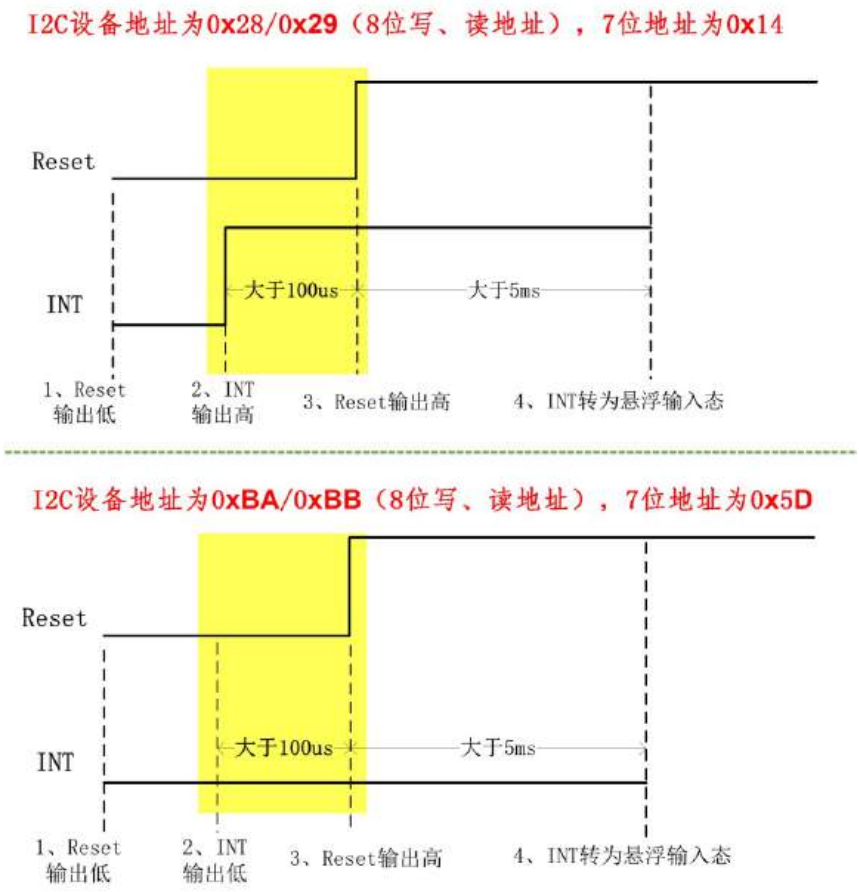


图 28-9 GT917S 的上电时序及 I2C 设备地址

29.2.3 寄存器配置

上电复位后，GT917S 芯片需要通过外部主控芯片加载寄存器配置，设定它的工作模式，这些配置通过 I2C 信号线传输到 GT917S，它的配置寄存器地址都由两个字节来表示，这些寄存器的地址从 0x8047- 0x8135，一般来说，我们实际配置的时候会按照 GT917S 生产厂商给的默认配置来控制芯片，仅修改部分关键寄存器，其中部分寄存器说明见图 *28_10*。（注意，由于芯片厂商并没有发布 GT917S 的寄存器列表说明文档，此处使用旧型号芯片 GT9157 的寄存器表截图来进行说明，这两个型号大部分兼容）

寄存器	Config Data	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0x8047	Config_ Version	配置文件的版本号(新下发的配置版本号大于原版本，或等于原版本号但配置内容有变化时保存，版本号版本正常范围：'A'~'Z'，发送 0x00 则将版本号初始化为'A')							
0x8048	X Output Max (Low Byte)	X 坐标输出最大值							
0x8049	X Output Max (High Byte)								
0x804A	Y Output Max (Low Byte)	Y 坐标输出最大值							
0x804B	Y Output Max (High Byte)								
0x804C	Touch Number	Reserved				输出触点个数上限：1~10			
0x804D	Module_ Switch1	Stylus_priority (预定义)		Stretch_rank		X2Y (X,Y 坐标交换)	Sito (软件降噪)	INT 触发方式 00：上升沿触发 01：下降沿触发 02：低电平查询 03：高电平查询	
0x804E-0x80FE寄存器省略									
0x80FF	Config_Chksum	配置信息校验(0x8047 到 0x80FE 之字节和的补码)							
0x8100	Config_Fresh	配置已更新标记(由主控写入标记)							

图 28-10 部分寄存器配置说明

这些寄存器介绍如下：

(1) 配置版本寄存器

0x8047 配置版本寄存器，它包含有配置文件的版本号，若新写入的版本号比原版本大，或者版本号相等，但配置不一样时，才会更新配置文件到寄存器中。其中配置文件是指记录了寄存器 0x8048-0x80FE 控制参数的一系列数据。

为了保证每次都更新配置，我们一般把配置版本寄存器设置为 “0x00”，这样版本号会默认初始化为 ‘A’，这样每次我们修改其它寄存器配置的时候，都会写入到 GT917S 中。

(2) X、Y 分辨率

0x8048-0x804B 寄存器用于配置触控芯片输出的 XY 坐标的最大值，为了方便使用，我们把它配置得跟液晶面板的分辨率一致，这样就能使触控芯片输出的坐标一一对应到液晶面板的每一个像素点了。

(3) 触点个数

0x804C 触点个数寄存器用于配置它最多可输出多少个同时按下的触点坐标，这个极限值跟触摸屏面板有关，如我们本章实验使用的触摸面板最多支持 5 点触控。

(4) 模式切换

0x804D 模式切换寄存器中的 X2Y 位可以用于交换 XY 坐标轴；而 INT 触发方式位可以配置不同的触发方式，当有触摸信号时，INT 引脚会根据这里的配置给出触发信号。

(5) 配置校验

0x80FF 配置校验寄存器用于写入前面 0x8047-0x80FE 寄存器控制参数字节之和的补码，GT917S 收到前面的寄存器配置时，会利用这个数据进行校验，若不匹配，就不会更新寄存器配置。

(6) 配置更新

0x8100 配置更新寄存器用于控制 GT917S 进行更新，传输了前面的寄存器配置并校验通过后，对这个寄存器写 1，GT917S 会更新配置。

29.2.4 读取坐标信息

29.2.4.1 坐标寄存器

上述寄存器主要是由外部主控芯片给 GT917S 写入配置的，而它则使用图 *28_11* 中的寄存器向主控器反馈信息。

Addr	Access	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0x8140	R	Product ID (Lowest Byte，ASCII 码)							
0x8141	R	Product ID (Third Byte，ASCII 码)							
0x8142	R	Product ID (Second Byte，ASCII 码)							
0x8143	R	Product ID (Highest Byte，ASCII 码)							
0x8144	R	Firmware version (16 进制数 LowByte)							
0x8145	R	Firmware version (16 进制数 HighByte)							
0x8146	R	x coordinate resolution (low byte)							
0x8147	R	x coordinate resolution (high byte)							
0x8148	R	y coordinate resolution (low byte)							
0x8149	R	y coordinate resolution (high byte)							
0x814A	R	Vendor_id (当前模组选项信息)							
0x814B	R	Reserved							
0x814C	R	Reserved							
0x814D	R	Reserved							
0x814E	R/W	buffer status	large detect	Proximity Valid	HaveKey	number of touch points			
0x814F	R	track id							
0x8150	R	point 1 x coordinate (low byte)							
0x8151	R	point 1 x coordinate (high byte)							
0x8152	R	point 1 y coordinate (low byte)							
0x8153	R	point 1 y coordinate (high byte)							
0x8154	R	Point 1 size (low byte)							
0x8155	R	point 1 size (high byte)							
0x8156	R	Reserved							
0x8157	R	track id							
0x8158	R	point 2 x coordinate (low byte)							
0x8159	R	point 2 x coordinate (high byte)							
0x815A	R	point 2 y coordinate (low byte)							
0x815B	R	point 2 y coordinate (high byte)							
0x815C	R	point 2 size (low byte)							
0x815D	R	point 2 size (high byte)							
0x815E	R	Reserved							

省略track id3-track ic10的寄存器

图 28-11 坐标信息寄存器

(1) 产品 ID 及版本

0x8140-0x8143 寄存器存储的是产品 ID，上电后我们可以利用 I2C 读取这些寄存器的

值来判断 I2C 是否正常通讯，这些寄存器中包含有 “5688” 字样; 而 0x8144-0x8145 则保存有固件版本号，不同版本可能不同。

(2) X/Y 分辨率

0x8146-0x8149 寄存器存储了控制触摸屏的分辨率，它们的值与我们前面在配置寄存器写入的 XY 控制参数一致。所以我们可以通过读取这两个寄存器的值来确认配置参数是否正确写入。

(3) 状态寄存器

0x814E 地址的是状态寄存器，它的 Buffer status 位存储了坐标状态，当它为 1 时，表示新的坐标数据已准备好，可以读取，0 表示未就绪，数据无效，外部控制器读取完坐标后，须对这个寄存器位写 0 。number of touch points 位表示当前有多少个触点。其余数据位我们不关心。

(4) 坐标数据

从地址 0x814F-0x8156 的是触摸点 1 的坐标数据，从 0x8157-0x815E 的是触摸点 2 的坐标数据，依次还有存储 3-10 触摸点坐标数据的寄存器。读取这些坐标信息时，我们通过它们的 track id 来区分笔迹，多次读取坐标数据时，同一个 track id 号里的数据属于同一个连续的笔划轨迹。

29.2.4.2 读坐标流程

上电、配置完寄存器后，GT917S 就会开监测触摸屏，若我们前面的配置使 INT 采用中断上升沿报告触摸信号的方式，整个读取坐标信息的过程如下：

(1) 待机时 INT 引脚输出低电平；

(2) 有坐标更新时，INT 引脚输出上升沿；

(3) INT 输出上升沿后，INT 脚会保持高直到下一个周期（该周期可由配置 Refresh_Rate 决定）。外部主控器在检测到 INT 的信号后，先读取状态寄存器 (0x814E) 中的 number of touch points 位获当前有多少个触摸点，然后读取各个点的坐标数据，读取完后将 buffer status 位写为 0。外部主控器的这些读取过程要在一周期内完成，该周期由 0x8056 地址的 Refresh_Rate 寄存器配置；

(4) 上一步骤中 INT 输出上升沿后，若主控未在一个周期内读走坐标，下次 GT917S 即使检测到坐标更新会再输出一个 INT 脉冲但不更新坐标；

(5) 若外部主控一直未读走坐标，则 GT917S 会一直输出 INT 脉冲。

29.3 电容触摸屏—触摸画板实验

本小节讲解如何驱动电容触摸屏，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“电容触摸屏—触摸画板”工程配合阅读。

29.3.1 硬件设计

正面图



背面

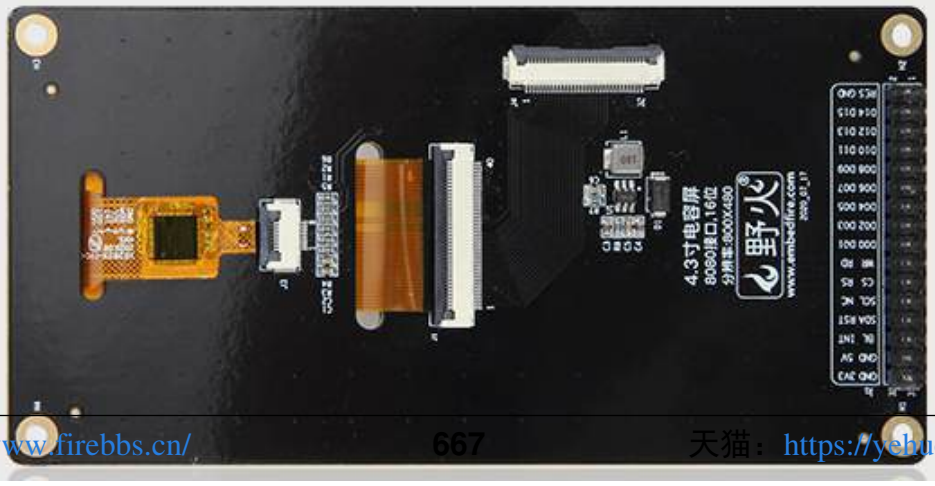


图 28-12 液晶屏实物图

本实验使用的液晶电容屏实物见图 *28_12*，屏幕背面的 PCB 电路对应图 *28_13*、图 *28_14* 中的原理图，分别是触摸屏接口及排针接口。

我们这个触摸屏出厂时就与 GT917S 芯片通过柔性电路板连接在一起了，柔性电路板从 GT917S 芯片引出 VCC、GND、SCL、SDA、RSTN 及 INT 引脚，再通过 FPC 座子引出到屏幕的 PCB 电路板中，PCB 电路板加了部分电路，如 I2C 的上拉电阻，然后把这些引脚引出到屏幕下方的排针处，方便整个屏幕与外部器件相连。

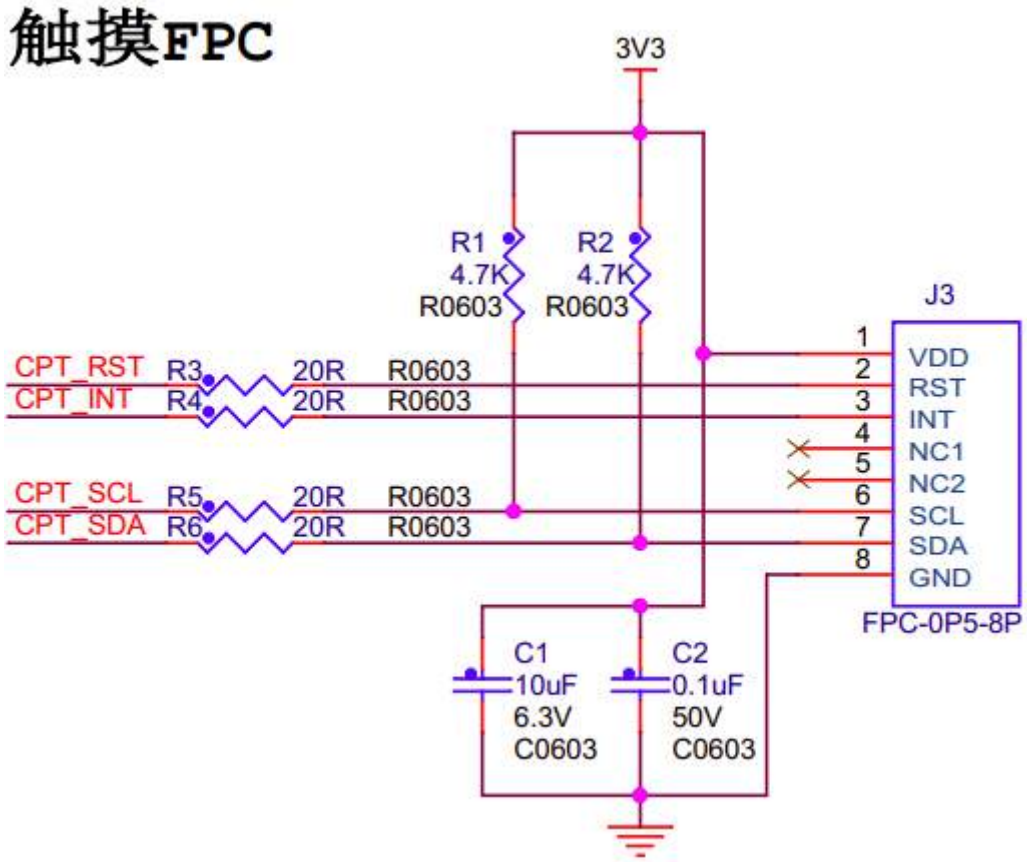


图 28-13 电容屏接口

以上是我们 STM32F407 实验板使用的 4.3 寸屏原理图，它通过屏幕上的排针接入到实验板的液晶排母接口，与 STM32 芯片的引脚相连，连接见图 *28_14*。

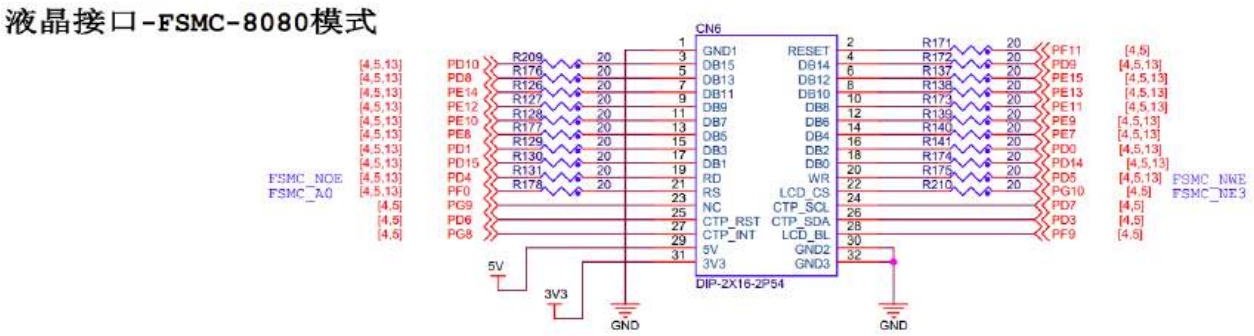


图 28-14 屏幕与实验板的引脚连接

图 *28_14* 中 23~27 号引脚即电容触摸屏相关的控制引脚。在本硬件设计中，没有使用 STM32 的硬件 I2C 引脚，只使用普通的 GPIO 与触摸芯片的 SCL、SDA 连接，在程序上使用软件模拟 I2C 与触摸芯片进行通讯。

以上原理图可查阅《LCD4.3 寸液晶原理图》及《野火 F407 开发板底板原理图》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。

29.3.2 软件设计

本工程中的 GT917S 芯片驱动主要是从官方提供的 Linux 驱动修改过来的，我们把这部分文件存储到“gt9xx.c”及“gt9xx.h”文件中，而这些驱动的底层 I2C 通讯接口我们存储到了“bsp_i2c_touch.c”及“bsp_i2c_touch.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。在我们提供的资料《GT5663 驱动.zip》压缩包里有官方的原 Linux 驱动，感兴趣的读者可以对比这些文件，了解如何移植驱动。

29.3.2.1 编程要点

(1) 分析官方的 gt9xx 驱动，了解需要提供哪些底层接口；

(2) 编写底层驱动接口；

(3) 利用 gt9xx 驱动，获取触摸坐标；

(4) 编写测试程序检验驱动。

29.3.2.2 代码分析

29.3.2.2.1 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_i2c_touch.h”文件中，见*代码清单 28_1* 触摸屏硬件配置相关的宏 (bsp_i2c_touch.h 文件)。

列表 1: 代码清单 28-1 触摸屏硬件配置相关的宏 (bsp_i2c_touch.h 文件)

```
/* 设定使用的电容屏 IIC 设备地址 */
#define GTP_ADDRESS             0xBA

#define I2CT_FLAG_TIMEOUT          ((uint32_t)0x1000)
#define I2CT_LONG_TIMEOUT          ((uint32_t)(10 * I2CT_F

#define GTP_I2C_SCL_CLK()           __GPIOD_CLK_ENABLE()
#define GTP_I2C_SDA_CLK()           __GPIOD_CLK_ENABLE()
#define GTP_I2C_RST_CLK()           __GPIOD_CLK_ENABLE()
#define GTP_I2C_GTP_CLK()           __GPIOG_CLK_ENABLE()

/*I2C 引脚 */
#define GTP_I2C_SCL_PIN                     GPIO_PIN_7
#define GTP_I2C_SCL_GPIO_PORT               GPIOD

#define GTP_I2C_SDA_PIN                     GPIO_PIN_3
#define GTP_I2C_SDA_GPIO_PORT               GPIOD

/* 复位引脚 */
#define GTP_RST_GPIO_PORT                   GPIOD
#define GTP_RST_GPIO_PIN                    GPIO_PIN_6
/* 中断引脚 */
#define GTP_INT_GPIO_PORT                   GPIOG
```

(下页继续)

(续上页)

```
#define GTP_INT_GPIO_PIN                    GPIO_PIN_8
#define GTP_INT_EXTI_IRQ                    EXTI9_5_IRQn
/* 中断服务函数 */
#define GTP_IRQHandler                      EXTI9_5_IRQHandler
```

以上代码根据硬件的连接，把与触摸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。在这里还定义了与 GT917S 芯片通讯的 I2C 设备地址，该地址是一个 8 位的写地址，它是由我们的上电时序决定的。

29.3.2.2.2 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* 触摸屏控制引脚的 GPIO 初始化函数 (bsp_i2c_touch.c 文件)。

列表 2: 代码清单 28-2 触摸屏控制引脚的 GPIO 初始化函数 (bsp_i2c_touch.c 文件)

```
static void I2C_GPIO_Config(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;

   /* 使能触摸屏使用的引脚的时钟 */
   RCC_AHB1PeriphClockCmd(GTP_I2C_SCL_GPIO_CLK|
                          GTP_I2C_SDA_GPIO_CLK|
                          GTP_RST_GPIO_CLK|GTP_INT_GPIO_CLK,
                          ENABLE);

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

      /* 配置 SCL 引脚 */
      GPIO_InitStructure.GPIO_Pin = GTP_I2C_SCL_PIN;
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
    GPIO_Init(GTP_I2C_SCL_GPIO_PORT, &GPIO_InitStructure);

    /* 配置 SDA 引脚 */
    GPIO_InitStructure.GPIO_Pin = GTP_I2C_SDA_PIN;
    GPIO_Init(GTP_I2C_SDA_GPIO_PORT, &GPIO_InitStructure);


  /* 配置 RST 引脚，下拉推挽输出 */
  GPIO_InitStructure.GPIO_Pin = GTP_RST_GPIO_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
  GPIO_Init(GTP_RST_GPIO_PORT, &GPIO_InitStructure);

  /* 配置 INT 引脚，下拉推挽输出，方便初始化 */
  GPIO_InitStructure.GPIO_Pin = GTP_INT_GPIO_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    //设置为下拉，方便初始化
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
  GPIO_Init(GTP_INT_GPIO_PORT, &GPIO_InitStructure);
}
```

以上函数初始化了触摸屏用到的 I2C 信号线，并且把 RST 及 INT 引脚也初始化成了下拉推挽输出模式，以便刚上电的时候输出上电时序，设置触摸屏的 I2C 设备地址。

29.3.2.2.3 配置 I2C 的模式

接下来需要配置 I2C 的工作模式，GT9157 芯片使用的是标准 7 位地址模式的 I2C 通讯，所以 I2C 这部分的配置跟我们在 EEPROM 实验中的是一样的，不了解这部分内容的请阅读 EEPROM 章节，见*代码清单 28_3* 软件 I2C 相关的部分函数 (bsp_i2c_touch.c 文件)。

列表 3: 代码清单 28-3 软件 I2C 相关的部分函数 (bsp_i2c_touch.c 文件)

```
//软件 IIC 使用的宏 (这部分宏在 bsp_i2c_touch.h 文件)
#define I2C_SCL_1()  GPIO_SetBits(GTP_I2C_SCL_GPIO_PORT, GTP_I2C_SCL_PIN)
↪          /* SCL = 1 */
#define I2C_SCL_0()  GPIO_ResetBits(GTP_I2C_SCL_GPIO_PORT, GTP_I2C_SCL_PIN)
↪          /* SCL = 0 */

#define I2C_SDA_1()  GPIO_SetBits(GTP_I2C_SDA_GPIO_PORT, GTP_I2C_SDA_PIN)
↪          /* SDA = 1 */
#define I2C_SDA_0()  GPIO_ResetBits(GTP_I2C_SDA_GPIO_PORT, GTP_I2C_SDA_PIN)
↪          /* SDA = 0 */

#define I2C_SDA_READ()  GPIO_ReadInputDataBit(GTP_I2C_SDA_GPIO_PORT, GTP_
↪I2C_SDA_PIN)          /* 读 SDA 口线状态 */

/*
*********************************************************
* 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
*********************************************************
*/
void i2c_Start(void)
{
```

(下页继续)

(续上页)

```
    /* 当 SCL 高电平时，SDA 出现一个下跳沿表示 I2C 总线启动信号 */
    I2C_SDA_1();
    I2C_SCL_1();
    i2c_Delay();
    I2C_SDA_0();
    i2c_Delay();
    I2C_SCL_0();
    i2c_Delay();
}

/*
*********************************************************************
* 函 数 名：i2c_Start
* 功能说明：CPU 发起 I2C 总线停止信号
* 形    参：无
* 返 回 值：无
*********************************************************************
*/
void i2c_Stop(void)
{
    /* 当 SCL 高电平时，SDA 出现一个上跳沿表示 I2C 总线停止信号 */
    I2C_SDA_0();
    I2C_SCL_1();
    i2c_Delay();
    I2C_SDA_1();
}

/*
*********************************************************************
* 函 数 名：i2c_ReadByte
* 功能说明：CPU 从 I2C 总线设备读取 8bit 数据
* 形    参：无
```

(下页继续)

(续上页)

```
* 返 回 值: 读到的数据
*************************************************************
*/
uint8_t i2c_ReadByte(void)
{
   uint8_t i;
   uint8_t value;

   /* 读到第 1 个 bit 为数据的 bit7 */
   value = 0;
   for (i = 0; i < 8; i++) {
         value <<= 1;
         I2C_SCL_1();
         i2c_Delay();
         if (I2C_SDA_READ()) {
            value++;
         }
         I2C_SCL_0();
         i2c_Delay();
   }
   return value;
}
```

考虑到硬件 I2C 在实际驱动时存在无法成功发送信号的情况，本工程中关于 I2C 的底层驱动使用软件 I2C。

所谓软件 I2C，即直接控制 STM32 的 IO 引脚的电平，根据 I2C 的时序产生对应的起始信号、结束信号、应答和非应答信号等。上述代码中使用 I2C_SCL_1、I2C_SCL_0 宏控制引脚产生高低电平逻辑，从而产生 I2C 需要的 SCL 时钟信号；I2C_SDA_1、I2C_SDA_0 宏控制 I2C 总线的 SDA 数据线，产生数据输出，当需要读取数据时，使用宏 I2C_SDA_READ。实际上这些宏都是对 GPIO 进行基本的读写，只是把它们按照 I2C 协议的要求来组织，就产生了 I2C 协议的通讯逻辑，实现了 I2C 通讯。（上述代码中省略了部分软件 I2C 的函数，感兴趣的请直接在配套代码中查看）

相对于硬件 I2C，软件 I2C 所有逻辑的产生均由 CPU 直接控制，不过这并不会比前面硬件 I2C 读写 EEPROM 的驱动消耗更多的 CPU 资源，因为在该硬件 I2C 的驱动中，CPU 需要轮询硬件 I2C 的事件标志，并不会省下 CPU 的时间。而软件 I2C 的访问速度可通过调整 i2c_Delay 函数的延时来修改，使用时不会比硬件 I2C 慢。

29.3.2.2.4 使用上电时序设置触摸屏的 I2C 地址

在上面配置完成 STM32 的引脚后，就可以开始控制这些引脚对触摸屏进行控制了，为了使用 I2C 通讯，首先要根据 GT917S 芯片的上电时序给它设置 I2C 设备地址，见*代码清单 28_4* 使用上电时序设置触摸屏的 I2C 地址 (bsp_i2c_touch.c 文件)。

列表 4: 代码清单 28-4 使用上电时序设置触摸屏的 I2C 地址 (bsp_i2c_touch.c 文件)

```
/**
   * @brief  对 GT91xx 芯片进行复位
   * @param  无
   * @retval 无
   */
void I2C_ResetChip(void)
{
      GPIO_InitTypeDef GPIO_InitStructure;

   /* 配置 INT 引脚，下拉推挽输出，方便初始化 */
      GPIO_InitStructure.GPIO_Pin = GTP_INT_GPIO_PIN;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
      GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;        //设置为下拉，方便
初始化
      GPIO_Init(GTP_INT_GPIO_PORT, &GPIO_InitStructure);
```

(下页继续)

(续上页)

```
    /* 初始化 GT5688,rst 为高电平，int 为低电平，则 gt5688 的设备地址被配置为
0xBA*/

    /* 复位为低电平，为初始化做准备 */
    GPIO_ResetBits (GTP_RST_GPIO_PORT,GTP_RST_GPIO_PIN);
    Delay(0x0FFFFF);

    /* 拉高一段时间，进行初始化 */
    GPIO_SetBits (GTP_RST_GPIO_PORT,GTP_RST_GPIO_PIN);
    Delay(0x0FFFFF);

    /* 把 INT 引脚设置为浮空输入模式，以便接收触摸中断信号 */
    GPIO_InitStructure.GPIO_Pin = GTP_INT_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
    GPIO_Init(GTP_INT_GPIO_PORT, &GPIO_InitStructure);
}
```

这段函数中控制 RST 引脚由低电平改变至高电平，且期间 INT 一直为低电平，这样的上电时序可以控制触控芯片的 I2C 写地址为 0xBA，读地址为 0xBB, 即 (0xBA|0x01)。输出完上电时序后，把 STM32 的 INT 引脚模式改成浮空输入模式，使它可以接收触控芯片输出的触摸中断信号。接下来我们在 I2C_GTP_IRQEnable 函数中使能 INT 中断，见代码清单 28_5。

列表 5: 代码清单 28-5 使能 INT 中断 (bsp_i2c_touch.c 文件)

```
/**
  * @brief  使能触摸屏中断
  * @param  无
  * @retval 无
  */
```

(下页继续)

(续上页)

```
void I2C_GTP_IRQEnable(void)
{
   EXTI_InitTypeDef EXTI_InitStructure;
   NVIC_InitTypeDef NVIC_InitStructure;
   GPIO_InitTypeDef GPIO_InitStructure;
   /* 配置 INT 为浮空输入 */
   GPIO_InitStructure.GPIO_Pin = GTP_INT_GPIO_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
   GPIO_Init(GTP_INT_GPIO_PORT, &GPIO_InitStructure);

   /* 连接 EXTI 中断源 到 INT 引脚 */
   SYSCFG_EXTILineConfig(GTP_INT_EXTI_PORTSOURCE, GTP_INT_EXTI_PINSOURCE);

   /* 选择 EXTI 中断源 */
   EXTI_InitStructure.EXTI_Line = GTP_INT_EXTI_LINE;
   EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
   EXTI_InitStructure.EXTI_LineCmd = ENABLE;
   EXTI_Init(&EXTI_InitStructure);

   /* 配置中断优先级 */
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

      /* 使能中断 */
   NVIC_InitStructure.NVIC_IRQChannel = GTP_INT_EXTI_IRQ;
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
   NVIC_Init(&NVIC_InitStructure);
```

(下页继续)

(续上页)

```
}
```

这个 INT 引脚我们配置为上升沿触发，是跟后面写入到触控芯片的配置参数一致的。

29.3.2.2.5 初始化封装

利用以上函数，我们把信号引脚及 I2C 设备地址初始化的过程都封装到函数 I2C_Touch_Init 中，见代码清单 28_6。

列表 6: 代码清单 28-6 封装引脚初始化及上电时序
(bsp_i2c_touch.c 文件)

```
/**
* @brief  I2C 外设 (gt9xx) 初始化
* @param  无
* @retval 无
*/
void I2C_Touch_Init(void)
{
   I2C_GPIO_Config();
   I2C_ResetChip();
   I2C_GTP_IRQEnable();
}
```

29.3.2.2.6 I2C 基本读写函数

为了与上层“gt9xx.c”驱动文件中的函数对接，本实验中的 I2C 读写函数与 EEPROM 实验中的有稍微不同，见代码清单 28_7。

列表 7: 代码清单 28-7 I2C 基本读写函数 (bsp_i2c_touch.c 文件)

```
#define I2C_DIR_WR  0   /* 写控制 bit */
#define I2C_DIR_RD  1   /* 读控制 bit */
/**
   * @brief   使用 IIC 读取数据
   * @param
   *   @arg ClientAddr: 从设备地址
   *   @arg pBuffer: 存放由从机读取的数据的缓冲区指针
   *   @arg NumByteToRead: 读取的数据长度
   * @retval  无
   */
  uint32_t I2C_ReadBytes(uint8_t ClientAddr,uint8_t* pBuffer, uint16_t
↪NumByteToRead)
   {

      /* 第 1 步: 发起 I2C 总线启动信号 */
      i2c_Start();

      /* 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
↪*/
      i2c_SendByte(ClientAddr | I2C_DIR_RD); /* 此处是读指令 */

      /* 第 3 步: 等待 ACK */
      if (i2c_WaitAck() != 0)
      {
         goto cmd_fail;       /* 器件无应答 */
      }

      while(NumByteToRead)
   {
```

(下页继续)

(续上页)

```
    if(NumByteToRead == 1)
    {
        i2c_NAck();       /* 最后 1 个字节读完后，CPU 产生 NACK 信号 (驱动
SDA = 1) */

        /* 发送 I2C 总线停止信号 */
        i2c_Stop();
    }

    *pBuffer = i2c_ReadByte();

    /* 读指针自增 */
    pBuffer++;

    /* 计数器自减 */
    NumByteToRead--;

    i2c_Ack();      /* 中间字节读完后，CPU 产生 ACK 信号 (驱动 SDA = 0) */
  }

    /* 发送 I2C 总线停止信号 */
    i2c_Stop();
    return 0;       /* 执行成功 */

  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 */
    /* 发送 I2C 总线停止信号 */
    i2c_Stop();
    return 1;
  }

/**
  * @brief   使用 IIC 写入数据
```

(下页继续)

(续上页)

```
    * @param
    *   @arg ClientAddr: 从设备地址
    *   @arg pBuffer: 缓冲区指针
    *     @arg NumByteToWrite: 写的字节数
    * @retval  无
    */
uint32_t I2C_WriteBytes(uint8_t ClientAddr,uint8_t* pBuffer,  uint8_t␣
↪NumByteToWrite)
{
    uint16_t m;

/*   第 0 步: 发停止信号, 启动内部写操作   */
i2c_Stop();

/* 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
    CLK 频率为 200KHz 时, 查询次数为 30 次左右
*/
for (m = 0; m < 1000; m++)
{
    /* 第 1 步: 发起 I2C 总线启动信号 */
    i2c_Start();

    /* 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 */
    i2c_SendByte(ClientAddr | I2C_DIR_WR);     /* 此处是写指令 */

    /* 第 3 步: 发送一个时钟, 判断器件是否正确应答 */
    if (i2c_WaitAck() == 0)
    {
        break;
    }
}
if (m  == 1000)
```

(下页继续)

(续上页)

```
{
    goto cmd_fail;      /* EEPROM 器件写超时 */
}

while(NumByteToWrite--)
{
/* 第 4 步：开始写入数据 */
i2c_SendByte(*pBuffer);

/* 第 5 步：检查 ACK */
if (i2c_WaitAck() != 0)
{
    goto cmd_fail;      /* 器件无应答 */
}

        pBuffer++;      /* 地址增 1 */
}

    /* 命令执行成功，发送 I2C 总线停止信号 */
    i2c_Stop();
    return 0;

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 */
    /* 发送 I2C 总线停止信号 */
    i2c_Stop();
    return 1;
}
```

为了与后面的 gt9xx.c 文件的读写接口兼容，这里的读写函数都是很纯粹的 I2C 通讯过程，即读函数只有读过程，不包含发送寄存器地址的过程，而写函数也是只有写过程，没有包含寄存器的地址，大家可以对比一下它们与前面 EEPROM 实验中的差别。这两个函数都只包含从 I2C 的设备地址、缓冲区指针以及数据量。

29.3.2.2.7 Linux 的 I2C 驱动接口

使用前面的基本读写函数，主要是为了对接原“gt9xx.c”驱动里使用的 Linux I2C 接口函数 I2C_Transfer，实现了这个函数后，移植时就可以减少“gt9xx.c”文件的修改量。I2C_Transfer 函数见*代码清单 28_8*。

列表 8: 代码清单 28-8 Linux 的 I2C 驱动接口 (gt9xx.c 文件)

```
/* 表示读数据 */
#define I2C_M_RD          0x0001
/*
* 存储 I2C 通讯的信息
* @addr:   从设备的 I2C 设备地址
* @flags: 控制标志
* @len:   读写数据的长度
* @buf:   存储读写数据的指针
**/
struct i2c_msg
{
   uint8_t addr;         /* 从设备的 I2C 设备地址 */
   uint16_t flags;       /* 控制标志 */
   uint16_t len;         /* 读写数据的长度      */
   uint8_t *buf;         /* 存储读写数据的指针    */
};
/**
* @brief   使用 IIC 进行数据传输
* @param
*     @arg i2c_msg: 数据传输结构体
*     @arg num: 数据传输结构体的个数
* @retval  正常完成的传输结构个数，若不正常，返回 0xff
*/
static int I2C_Transfer( struct i2c_msg *msgs,int num)
```

(下页继续)

(续上页)

```
{
    int im = 0;
    int ret = 0;
    //输出调试信息，可忽略
    GTP_DEBUG_FUNC();
    for (im = 0; ret == 0 && im != num; im++)
    {
        //根据 flag 判断是读数据还是写数据
        if ((msgs[im].flags&I2C_M_RD))
        {
                //IIC 读取数据
                ret = I2C_ReadBytes(msgs[im].addr, msgs[im].buf, msgs[im].len);
        }
        else
        {
                //IIC 写入数据
                ret = I2C_WriteBytes(msgs[im].addr,  msgs[im].buf, msgs[im].
↪len);
        }
    }
    if (ret)
        return ret;
    return im;                                          //正常完成的传输结构个数
}
```

I2C_Transfer 的主要输入参数是 i2c_msg 结构体的指针以及要传输多少个这样的结构体。i2c_msg 结构体包含以下几个成员：

(1) addr

这是从机的 I2C 设备地址，通讯时无论是读方向还是写方向，给这个成员赋值为写地址即可 (本实验中为 0xBA)。

(2) flags

这个成员存储了控制标志，它用于指示本 i2c_msg 结构体要求以什么方式来传输。在原 Linux 驱动中有很多种控制方式，在我们这个工程中，只支持读或写控制标志，flags 被赋值为 I2C_M_RD 宏的时候表示读方向，其余值表示写方向。

(3) len

本成员存储了要读写的数据长度。

(4) buf

本成员存储了指向读写数据缓冲区的指针。

利用这个结构体，我们再来看 I2C_Transfer 函数做了什么工作。

(1) 输入参数中可能包含有多个要传输的 i2c_msg 结构体，利用 for 循环把这些结构体一个个地传输出去；

(2) 传输的时候根据 i2c_msg 结构体中的 flags 标志，确定应该调用 I2C 读函数还是写函数，这些函数即前面定义的 I2C 基本读写函数。调用这些函数的时候，以 i2c_msg 结构体的成员作为参数。

29.3.2.2.8 I2C 复合读写函数

理解了 I2C_Transfer 函数的代码，我们发现它还是什么都没做，只是对 I2C 基本读写函数封装了比较特别的调用形式而已，而我们知道 GT917S 触控芯片都有很多不同的寄存器，如果我们仅用上面的函数，如何向特定寄存器写入参数或读取特定寄存器的内容呢？这就需要再利用 I2C_Transfer 函数编写具有 I2C 通讯复合时序的读写函数了。Linux 驱动进行这样的封装是为了让它的核心层与具体设备独立开来，对于这个巨型系统，这样写代码是很有必要的，上述的 I2C_Transfer 函数属于 Linux 内部的驱动层，它对外提供接口，而像 GT917S、EEPROM 等使用 I2C 的设备，都利用这个接口编写自己具体的驱动文件，GT917S 的这些 I2C 复合读写函数见代码清单 28_9。

列表 9: 代码清单 28-9 I2C 复合读写函数（gt9xx.c 文件）

```
//寄存器地址的长度
#define GTP_ADDR_LENGTH       2

/**
```

(下页继续)

(续上页)

```
    * @brief   从 IIC 设备中读取数据
    * @param
    *   @arg client_addr: 设备地址
    *   @arg  buf[0~1]: 读取数据寄存器的起始地址
    *   @arg buf[2~len-1]: 存储读出来数据的缓冲 buffer
    *   @arg len:    GTP_ADDR_LENGTH + read bytes count (寄存器地址长度 + 读取的数据字节数)
    * @retval  i2c_msgs 传输结构体的个数, 2 为成功, 其它为失败
    */
static int32_t GTP_I2C_Read(uint8_t client_addr, uint8_t *buf, int32_t len)
{
    struct i2c_msg msgs[2];
    int32_t ret=-1;
    int32_t retries = 0;

    GTP_DEBUG_FUNC();
    /* 一个读数据的过程可以分为两个传输过程:
       * 1. IIC  写入 要读取的寄存器地址
       * 2. IIC  读取  数据
       * */

    msgs[0].flags = !I2C_M_RD;          //写入
    msgs[0].addr  = client_addr;          //IIC 设备地址
    msgs[0].len   = GTP_ADDR_LENGTH;  //寄存器地址为 2 字节 (即写入两字节的数据)
    msgs[0].buf   = &buf[0];            //buf[0~1] 存储的是要读取的寄存器地址

    msgs[1].flags = I2C_M_RD;         //读取
    msgs[1].addr  = client_addr;          //IIC 设备地址
    msgs[1].len   = len - GTP_ADDR_LENGTH;  //要读取的数据长度
msgs[1].buf   = &buf[GTP_ADDR_LENGTH]; //buf[GTP_ADDR_LENGTH] 之后的缓冲区存储读出的数据
```

(下页继续)

(续上页)

```
    while (retries < 5) {
        ret = I2C_Transfer( msgs, 2); //调用 IIC 数据传输过程函数，有 2 个传输过程
            if (ret == 2)break;
            retries++;
    }
    if ((retries >= 5)) {
    GTP_ERROR("I2C Read: 0x%04X, %d bytes failed, errcode: %d! Process reset.
↪",
(((uint16_t)(buf[0] << 8)) | buf[1]), len-2, ret);

    }
    return ret;
}

/**
    * @brief    向 IIC 设备写入数据
    * @param
    *    @arg client_addr: 设备地址
    *    @arg  buf[0~1]: 要写入的数据寄存器的起始地址
    *    @arg buf[2~len-1]: 要写入的数据
    * @arg len:    GTP_ADDR_LENGTH + write bytes count (寄存器地址长度 + 写入的数
据字节数)
    * @retval  i2c_msgs 传输结构体的个数，1 为成功，其它为失败
    */
static int32_t GTP_I2C_Write(uint8_t client_addr,uint8_t *buf,int32_t len)
{
    struct i2c_msg msg;
    int32_t ret = -1;
    int32_t retries = 0;

    GTP_DEBUG_FUNC();
    /* 一个写数据的过程只需要一个传输过程:
```

(下页继续)

(续上页)

```
    * 1. IIC 连续 写入 数据寄存器地址及数据
    * */
   msg.flags = !I2C_M_RD;       //写入
   msg.addr  = client_addr;       //从设备地址
   msg.len   = len;        //长度直接等于 (寄存器地址长度 + 写入的数据字节数)
   msg.buf   = buf;              //直接连续写入缓冲区中的数据 (包括了寄存器地址)

   while (retries < 5) {
         ret = I2C_Transfer(&msg, 1);  //调用 IIC 数据传输过程函数, 1 个传输过程
         if (ret == 1)break;
         retries++;
   }
   if ((retries >= 5)) {

GTP_ERROR("I2C Write: 0x%04X, %d bytes failed, errcode: %d! Process reset.",
(((uint16_t)(buf[0] << 8)) | buf[1]) , len-2, ret);

   }
   return ret;
}
```

可以看到，复合读写函数都包含有 client_addr、buf 及 len 输入参数，其中 client_addr 表示 I2C 的设备地址，buf 存储了要读写的寄存器地址及数据，len 表示 buf 的长度。在函数的内部处理中，复合读写过程被分解成两个基本的读写过程，输入参数被转化存储到 i2c_msg 结构体中，每个基本读写过程使用一个 i2c_msg 结构体来表示，见表 29-2 和表 29-3。

表 29-29-2 复合读过程的步骤分解

复合读过程的步骤分解	说明
传输寄存器地址	这相当于一个 I2C 的基本写过程，写入一个 2 字节长度的寄存器地址，buf 指针的前两个字节内容被解释为寄存器地址。
从寄存器读取内容	这是一个 I2C 的基本读过程，读取到的数据存储到 buf 指针的第 3 个地址开始的空间中。

表 29-3 复合写过程的步骤分解

复合写过程的步骤分解	说明
传输寄存器地址	这相当于一个 I2C 的基本写过程，写入一个 2 字节长度的寄存器地址，buf 指针的前两个字节内容被解释为寄存器地址。
向寄存器写入内容	这也是一个 I2C 的基本写过程，写入的数据为 buf 指针的第 3 个地址开始的内容。

复合过程的分解主要是针对寄存器地址传输和实际数据传输来划分的，调用这两个复合读写过程的时候，我们需要注意 buf 的前两个字节为寄存器地址，且 len 的长度为 buf 的整体长度。

29.3.2.2.9 读取触控芯片的产品 ID 及版本号

利用上述复合读写函数，我们就可以使用 I2C 控制触控芯片了，首先是最简单的读取版本函数，见*代码清单 28_10*。

列表 10: 代码清单 28-10 读取触控芯片的产品 ID 及版本号（gt9xx.c 文件）

```
/*******************************************************
Function:
   Read chip version.
Input:
```

(下页继续)

(续上页)

```
    client:  i2c device
    version: buffer to keep ic firmware version
Output:
    read operation return.
       2: succeed, otherwise: failed
*******************************************************/
int32_t GTP_Read_Version(void)
{
    int32_t ret = -1;
    uint8_t buf[8] = {GTP_REG_VERSION >> 8, GTP_REG_VERSION & 0xff};     //寄
存器地址

    GTP_DEBUG_FUNC();

    ret = GTP_I2C_Read(GTP_ADDRESS, buf, sizeof(buf));
    if (ret < 0)
    {
        GTP_ERROR("GTP read version failed");
        return ret;
    }
    if (buf[2] == '5')
    {
        GTP_INFO("IC1 Version: %c%c%c%c_%02x%02x", buf[2], buf[3], buf[4],␣
↪buf[5], buf[7], buf[6]);

            //GT5688 芯片
            if(buf[2] == '5' && buf[3] == '6' && buf[4] == '8'&& buf[5] ==
↪'8')
                touchIC = GT5688;
    }
    else if (buf[5] == 0x00)
    {
```

(下页继续)

(续上页)

```
        GTP_INFO("IC2 Version: %c%c%c_%02x%02x", buf[2], buf[3], buf[4],␣
↪buf[7], buf[6]);

            //GT911 芯片
            if(buf[2] == '9' && buf[3] == '1' && buf[4] == '1')
                touchIC = GT911;
    }
    else if (buf[5] == '7')
    {
        GTP_INFO("IC3 Version: %c%c%c%c_%02x%02x", buf[2], buf[3], buf[4],␣
↪buf[5], buf[7], buf[6]);

            //GT9147 芯片
            if(buf[2] == '9' && buf[3] == '1' && buf[4] == '4' && buf[5] ==
↪'7')
                touchIC = GT9147;
        }
    else
    {
        GTP_INFO("IC3 Version: %c%c%c%c_%02x%02x", buf[2], buf[3], buf[4],␣
↪buf[5], buf[7], buf[6]);

            //GT9157 芯片
            if(buf[2] == '9' && buf[3] == '1' && buf[4] == '5' && buf[5] ==
↪'7')
                touchIC = GT9157;
        }
    return ret;
}
```

这个函数定义了一个 8 字节的 buf 数组，并且向它的第 0 和第 1 个元素写入产品 ID 寄存器的地址，然后调用复合读取函数，即可从芯片中读取这些寄存器的信息，结果使用宏 GTP_INFO 输

出。

29.3.2.2.10 向触控芯片写入配置参数

万事俱备，现在我们可以使用 I2C 向触摸芯片写入寄存器配置了，见*代码清单 28_11*。

列表 11: 代码清单 28-11 初始化并向触控芯片写入配置参数（gt9xx.c 文件）

```
// 4.3 寸屏 GT917S 驱动配置
const uint8_t CTP_CFG_GT917S[] =  {
0x84,0x20,0x03,0xE0,0x01,0x05,0x05,0x00,0x00,0x40,
0x00,0x0F,0x78,0x64,0x53,0x11,0x00,0x00,0x00,0x00,
0x23,0x17,0x19,0x1D,0x0F,0x04,0x00,0x00,0x00,0x00,
0x00,0x00,0x04,0x51,0x14,0x00,0x00,0x00,0x00,0x00,
0x32,0x00,0x00,0x50,0x38,0x28,0x8A,0x20,0x11,0x37,
0x39,0xA2,0x07,0x38,0x6D,0x28,0x11,0x03,0x24,0x00,
0x01,0x28,0x50,0xC0,0x94,0x02,0x00,0x00,0x53,0xB8,
0x2E,0xA2,0x35,0x8F,0x3B,0x80,0x42,0x75,0x49,0x6B,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xF0,0x4C,0x3C,
0xFF,0xFF,0x07,0x14,0x14,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x50,0x73,
0x50,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x20,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x1F,0x1D,0x1B,0x1A,0x19,0x18,0x17,0x16,0x15,0x09,
0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0xFF,0xFF,0xFF,
0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
0xFF,0xFF,0x1C,0x1B,0x1A,0x19,0x18,0x17,0x15,0x14,
0x13,0x12,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x05,0x00,0x00,0x0F,
0x00,0x00,0x00,0x80,0x46,0x08,0x96,0x50,0x32,0x0A,
0x0A,0x64,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

(下页继续)

(续上页)

```
0x32,0x03,0x0C,0x08,0x23,0x00,0x14,0x23,0x00,0x28,
0x46,0x30,0x3C,0xD0,0x07,0x50,0x70,0xB0,0x01
};
/*******************************************************
Function:
   Initialize gtp.
Input:
   ts: goodix private data
Output:
   Executive outcomes.
      0: succeed, otherwise: failed
*******************************************************/
int32_t GTP_Init_Panel(void)
{
   int32_t ret = -1;

   int32_t i = 0;
   uint16_t check_sum = 0;
   int32_t retry = 0;

   const uint8_t* cfg_info;
   uint8_t cfg_info_len  ;
      uint8_t* config;

   uint8_t cfg_num =0 ;                //需要配置的寄存器个数

   GTP_DEBUG_FUNC();

//uint8_t config[GTP_CONFIG_MAX_LENGTH + GTP_ADDR_LENGTH]
//                = {GTP_REG_CONFIG_DATA >> 8, GTP_REG_CONFIG_DATA & 0xff};
```

(下页继续)

(续上页)

```
    I2C_Touch_Init();

    ret = GTP_I2C_Test();
    if (ret < 0)
    {
        GTP_ERROR("I2C communication ERROR!");
              return ret;
    }

        //获取触摸 IC 的型号
    GTP_Read_Version();

#if UPDATE_CONFIG

    config = (uint8_t *)malloc (GTP_CONFIG_MAX_LENGTH + GTP_ADDR_LENGTH);

        config[0] = GTP_REG_CONFIG_DATA >> 8;
        config[1] =  GTP_REG_CONFIG_DATA & 0xff;

        //根据 IC 的型号指向不同的配置
        if(touchIC == GT5688)
        {
            cfg_info =  CTP_CFG_GT5688; //指向寄存器配置
            cfg_info_len = CFG_GROUP_LEN(CTP_CFG_GT5688);//计算配置表的大小
        }
        else if(touchIC == GT917S)
        {
            cfg_info =  CTP_CFG_GT917S; //指向寄存器配置
            cfg_info_len = CFG_GROUP_LEN(CTP_CFG_GT917S);//计算配置表的大小
        }
        else
```

(下页继续)

(续上页)

```
    {//默认配置为 GT917S
        cfg_info =  CTP_CFG_GT917S; //指向寄存器配置
        cfg_info_len = CFG_GROUP_LEN(CTP_CFG_GT917S);//计算配置表的大小
    }

memset(&config[GTP_ADDR_LENGTH], 0, GTP_CONFIG_MAX_LENGTH);
memcpy(&config[GTP_ADDR_LENGTH], cfg_info, cfg_info_len);


    cfg_num = cfg_info_len;

    GTP_DEBUG("cfg_info_len = %d ",cfg_info_len);
    GTP_DEBUG("cfg_num = %d ",cfg_num);
    GTP_DEBUG_ARRAY(config,6);

    /* 根据 LCD 的扫描方向设置分辨率 */
    config[GTP_ADDR_LENGTH+1] = LCD_X_LENGTH & 0xFF;
    config[GTP_ADDR_LENGTH+2] = LCD_X_LENGTH >> 8;
    config[GTP_ADDR_LENGTH+3] = LCD_Y_LENGTH & 0xFF;
    config[GTP_ADDR_LENGTH+4] = LCD_Y_LENGTH >> 8;

    /* 根据扫描模式设置 X2Y 交换 */
    switch(LCD_SCAN_MODE)
    {
        case 0:case 2:case 4: case 6:
            config[GTP_ADDR_LENGTH+6] |= (X2Y_LOC);
            break;

        case 1:case 3:case 5: case 7:
            config[GTP_ADDR_LENGTH+6] &= ~(X2Y_LOC);
            break;
    }
```

(下页继续)

(续上页)

```
    //计算要写入 checksum 寄存器的值
    check_sum = 0;

        for (i = GTP_ADDR_LENGTH; i < (cfg_num+GTP_ADDR_LENGTH -3); i += 2)
        {
            check_sum += (config[i] << 8) + config[i + 1];
        }

        check_sum = 0 - check_sum;
        GTP_DEBUG("Config checksum: 0x%04X", check_sum);
        //更新 checksum
        config[(cfg_num+GTP_ADDR_LENGTH -3)] = (check_sum >> 8) & 0xFF;
        config[(cfg_num+GTP_ADDR_LENGTH -2)] = check_sum & 0xFF;
        config[(cfg_num+GTP_ADDR_LENGTH -1)] = 0x01;

    //写入配置信息
    for (retry = 0; retry < 5; retry++)
    {
        ret = GTP_I2C_Write(GTP_ADDRESS, config , cfg_num + GTP_ADDR_
↪LENGTH+2);
        if (ret > 0)
        {
              break;
        }
    }
    Delay(0xfffff);                                    //延迟等待芯片更新



#if 1          //读出写入的数据，检查是否正常写入
    //检验读出的数据与写入的是否相同
```

(下页继续)

(续上页)

```
{
        uint16_t i;
        uint8_t buf[300];
        buf[0] = config[0];
        buf[1] =config[1];     //寄存器地址

        GTP_DEBUG_FUNC();

        ret = GTP_I2C_Read(GTP_ADDRESS, buf, sizeof(buf));

              GTP_DEBUG("read ");

              GTP_DEBUG_ARRAY(buf,cfg_num);

           GTP_DEBUG("write ");

              GTP_DEBUG_ARRAY(config,cfg_num);

              //不对比版本号
        for(i=1;i<cfg_num+GTP_ADDR_LENGTH-3;i++)
        {

           if(config[i] != buf[i])
           {
              GTP_ERROR("Config fail ! i = %d ",i);
                    free(config);
              return -1;
           }
        }
        if(i==cfg_num+GTP_ADDR_LENGTH-3)
           GTP_DEBUG("Config success ! i = %d ",i);
}
```

(下页继续)

(续上页)

```
#endif
    free(config);
#endif

    /* 使能中断，这样才能检测触摸数据 */
        I2C_GTP_IRQEnable();

    GTP_Get_Info();



    return 0;
}
```

这段代码调用 I2C_Touch_Init 初始化了 STM32 的 I2C 外设，设定触控芯片的 I2C 设备地址，然后调用了 GTP_Read_Version 尝试获取触控芯片的版本号。接下来是函数的主体，它使用 GTP_I2C_Write 函数通过 I2C 把配置参数表 CTP_CFG_GT917S 写入到触控芯片的的配置寄存器中，注意传输中包含有 checksum 寄存器的值。写入完参数后调用 I2C_GTP_IRQEnable 以使能 INT 引脚检测中断。

29.3.2.2.11 INT 中断服务函数

经过上面的函数初始化后，触摸屏就可以开始工作了，当触摸时，INT 引脚会产生触摸中断，会进入中断服务函数 GTP_IRQHandler，见*代码清单 28_12*。

列表 12: 代码清单 28-12 触摸屏的中断服务函数 (stm32f4xx_it.c 文件)

```
void GTP_IRQHandler(void)
{   //确保是否产生了 EXTI Line 中断
    if (__HAL_GPIO_EXTI_GET_IT(GTP_INT_GPIO_PIN) != RESET) {
```

(下页继续)

(续上页)

```
        LED2_TOGGLE;
        GTP_TouchProcess();
        __HAL_GPIO_EXTI_CLEAR_IT(GTP_INT_GPIO_PIN);       //清除中断标志位
    }
}
```

中断服务函数只是简单地调用了 GTP_TouchProcess 函数，它是读取触摸坐标的主体，而它又直接调用了函数 Goodix_TS_Work_Func。

29.3.2.2.12 读取坐标数据

Goodix_TS_Work_Func 函数的内容见*代码清单 28_13*。

列表 13: 代码清单 28-13 Goodix_TS_Work_Func 坐标读取函数

```
/* 状态寄存器地址 */
#define GTP_READ_COOR_ADDR    0x814E

/**
* @brief   触屏处理函数，轮询或者在触摸中断调用
* @param 无
* @retval 无
*/
static void Goodix_TS_Work_Func(void)
{
    uint8_t  end_cmd[3] = {GTP_READ_COOR_ADDR >> 8, GTP_READ_COOR_ADDR &␣
↪0xFF, 0};
    uint8_t  point_data[2 + 1 + 8 * GTP_MAX_TOUCH + 1]={GTP_READ_COOR_ADDR >>
↪ 8, GTP_READ_COOR_ADDR & 0xFF};
    uint8_t  touch_num = 0;
    uint8_t  finger = 0;
```

(下页继续)

(续上页)

```
static uint16_t pre_touch = 0;
static uint8_t pre_id[GTP_MAX_TOUCH] = {0};

uint8_t client_addr=GTP_ADDRESS;
uint8_t* coor_data = NULL;
int32_t input_x = 0;
int32_t input_y = 0;
int32_t input_w = 0;
uint8_t id = 0;

int32_t i  = 0;
int32_t ret = -1;

GTP_DEBUG_FUNC();

ret = GTP_I2C_Read(client_addr, point_data, 12);//10 字节寄存器加 2 字节地址
if (ret < 0)
{
   GTP_ERROR("I2C transfer error. errno:%d\n ", ret);

   return;
}

finger = point_data[GTP_ADDR_LENGTH];//状态寄存器数据

if (finger == 0x00)                //没有数据，退出
{
   return;
}

if((finger & 0x80) == 0)//判断 buffer status 位
{
```

(下页继续)

(续上页)

```
    goto exit_work_func;//坐标未就绪，数据无效
  }

  touch_num = finger & 0x0f;//坐标点数
  if (touch_num > GTP_MAX_TOUCH)
  {
    goto exit_work_func;//大于最大支持点数，错误退出
  }

  if (touch_num > 1)//不止一个点
  {
    uint8_t buf[8 * GTP_MAX_TOUCH] = {(GTP_READ_COOR_ADDR + 10) >> 8,
↪(GTP_READ_COOR_ADDR + 10) & 0xff};

    ret = GTP_I2C_Read(client_addr, buf, 2 + 8 * (touch_num - 1));
    memcpy(&point_data[12], &buf[2], 8 * (touch_num - 1));
↪ //复制其余点数的数据到 point_data
  }



  if (pre_touch>touch_num)                    //pre_touch>touch_num,
表示有的点释放了
  {
    for (i = 0; i < pre_touch; i++)
↪          //一个点一个点处理
      {
        uint8_t j;
      for(j=0; j<touch_num; j++)
      {
            coor_data = &point_data[j * 8 + 3];
            id = coor_data[0] & 0x0F;
↪                                   //track id
```

(下页继续)

(续上页)

```
            if(pre_id[i] == id)
               break;

            if(j >= touch_num-1)                                                ␣
↪                                          //遍历当前所有 id 都找不到 pre_
↪id[i], 表示已释放
            {
               GTP_Touch_Up( pre_id[i]);
            }
         }
      }
   }


   if (touch_num)
   {
      for (i = 0; i < touch_num; i++)                                           ␣
↪          //一个点一个点处理
      {
            coor_data = &point_data[i * 8 + 3];

            id = coor_data[0] & 0x0F;                                           ␣
↪                                  //track id
            pre_id[i] = id;

            input_x  = coor_data[1] | (coor_data[2] << 8);    //x 坐标
            input_y  = coor_data[3] | (coor_data[4] << 8);    //y 坐标
            input_w  = coor_data[5] | (coor_data[6] << 8);    //size

            {
#if 0
                              /* 根据扫描模式更正 X/Y 起始方向 */
```

(下页继续)

(续上页)

```
                        switch(LCD_SCAN_MODE)
                        {
                            case 0:case 7:
                                input_y  = LCD_Y_LENGTH - input_y;
                                break;

                            case 2:case 3:
                                input_x  = LCD_X_LENGTH - input_x;
                                input_y  = LCD_Y_LENGTH - input_y;
                                break;

                            case 1:case 6:
                                input_x  = LCD_X_LENGTH - input_x;
                                break;

                            default:
                            break;
                        }
#endif
                GTP_Touch_Down( id, input_x, input_y, input_w);//数据处理
            }
        }
    }
    else if (pre_touch)                 //touch_ num=0 且 pre_touch! =0
    {
        for(i=0;i<pre_touch;i++)
        {
            GTP_Touch_Up(pre_id[i]);
        }
    }
```

(下页继续)

(续上页)

```
    pre_touch = touch_num;


exit_work_func:
    {
        ret = GTP_I2C_Write(client_addr, end_cmd, 3);
        if (ret < 0)
        {
                GTP_INFO("I2C write end_cmd error!");
        }
    }

}
```

这个函数的内容比较长，它首先是读取了状态寄存器，获当前有多少个触点，然后根据触点数去读取各个点的数据，其中还有包含有 pre_touch 点的处理，pre_touch 保存了上一次的触点数据，利用这些数据和触点的 track id 号，可以确认同一条笔迹。这个读取过程完毕后，还对状态寄存器的 buffer status 位写 0，结束读取。在实际应用中，我们并不需要掌握这个 Goodix_TS_Work_Func 函数的所有细节，因为在这个函数中提供了两个坐标获取接口，我们只要在这两个接口中修改即可简单地得到坐标信息。

29.3.2.2.13 触点释放和触点按下的坐标接口

Goodix_TS_Work_Func 函数中获取到新的坐标数据时会调用触点释放和触点按下这两个函数，我们只要在这两个函数中添加自己的坐标处理过程即可，见*代码清单 28_14*。

列表 14: 代码清单 28-14 触点释放和触点按下的坐标接口（gt9xx.c 文件）

```
/**
* @brief   用于处理或报告触屏检测到按下
```

(下页继续)

(续上页)

```
* @param
*     @arg      id: 触摸顺序 trackID
*     @arg      x:  触摸的 x 坐标
*     @arg      y:  触摸的 y 坐标
*     @arg      w:  触摸的 大小
* @retval 无
*/
/* 用于记录连续触摸时 (长按) 的上一次触摸位置，负数值表示上一次无触摸按下 */
static int16_t pre_x[GTP_MAX_TOUCH] = {-1,-1,-1,-1,-1};
static int16_t pre_y[GTP_MAX_TOUCH] = {-1,-1,-1,-1,-1};
static void GTP_Touch_Down(int32_t id,int32_t x,int32_t y,int32_t w)
{
    GTP_DEBUG_FUNC();
    /* 取 x、y 初始值大于屏幕像素值 */
    GTP_DEBUG("ID:%d, X:%d, Y:%d, W:%d", id, x, y, w);
    /* 处理触摸按钮，用于触摸画板 */
    Touch_Button_Down(x,y);
    /* 处理描绘轨迹，用于触摸画板 */
    Draw_Trail(pre_x[id],pre_y[id],x,y,&brush);
    /**************************************/
    /* 在此处添加自己的触摸点按下时处理过程即可 */
    /* (x,y) 即为最新的触摸点 *************/
    /**************************************/
    /*prex,prey 数组存储上一次触摸的位置，id 为轨迹编号 (多点触控时有多轨迹)*/
    pre_x[id] = x;
    pre_y[id] =y;
}
/**
* @brief   用于处理或报告触屏释放
* @param 释放点的 id 号
* @retval 无
*/
```

(下页继续)

(续上页)

```
static void GTP_Touch_Up( int32_t id)
{
    /* 处理触摸释放，用于触摸画板 */
    Touch_Button_Up(pre_x[id],pre_y[id]);
    /****************************************/
    /* 在此处添加自己的触摸点释放时的处理过程即可 */
    /* pre_x[id],pre_y[id] 即为最新的释放点 ****/
    /****************************************/
    /***id 为轨迹编号 (多点触控时有多轨迹)********/
    /* 触笔释放，把 pre xy 重置为负 */
    pre_x[id] = -1;
    pre_y[id] = -1;
    GTP_DEBUG("Touch id[%2d] release!", id);
}
```

以上是我们工程中对这两个接口的应用，我们把触摸画板的坐标处理过程直接放到接口里了，大家可参考我们的演示，在函数的注释部分，根据自己的应用编写坐标处理过程。

注意这两个坐标接口都还是在中断服务函数里调用的 (中断服务函数调用 Goodix_TS_Work_Func 函数，该函数再调用这两个坐标接口)，实际应用中可以先把这些坐标信息存储起来，等待到系统空闲的时候再处理，就可以减轻中断服务程序的负担了。

29.3.2.3 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_15*。

列表 15: 代码清单 28-15 main 函数

```
int main ( void )
{
    LED_GPIO_Config();
    NT35510_Init ();          //LCD 初始化
```

(下页继续)

(续上页)

```
    Debug_USART_Config();


    //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
NT35510_GramScan ( 3 );

    /* 设定好液晶扫描方向后，再初始化触摸屏，触摸屏会根据液晶的扫描方向输出匹配的触摸坐标
    */
    /* 每次修改液晶扫描方向后，应重新调用一次 GTP_Init_Panel 函数更新触摸配置 */
    GTP_Init_Panel();

    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");


    //绘制触摸画板界面
    Palette_Init(LCD_SCAN_MODE);

    while ( 1 )
    {
    }


}
```

main 函数初始化触摸屏、液晶屏后，调用了 Palette_Init 函数初始化了触摸画板应用，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。

29.3.3 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 30 章　ADC—电压采集

本章参考资料：《STM32F4xx 中文参考手册》ADC 章节。

学习本章时，配合《STM32F4xx 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。

30.1 ADC 简介

STM32F407ZGT6 有 3 个 ADC，每个 ADC 有 12 位、10 位、8 位和 6 位可选，每个 ADC 有 16 个外部通道。另外还有两个内部 ADC 源和 VBAT 通道挂在 ADC1 上。ADC 具有独立模式、双重模式和三重模式，对于不同 AD 转换要求几乎都有合适的模式可选。ADC 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。

30.2 ADC 功能框图剖析

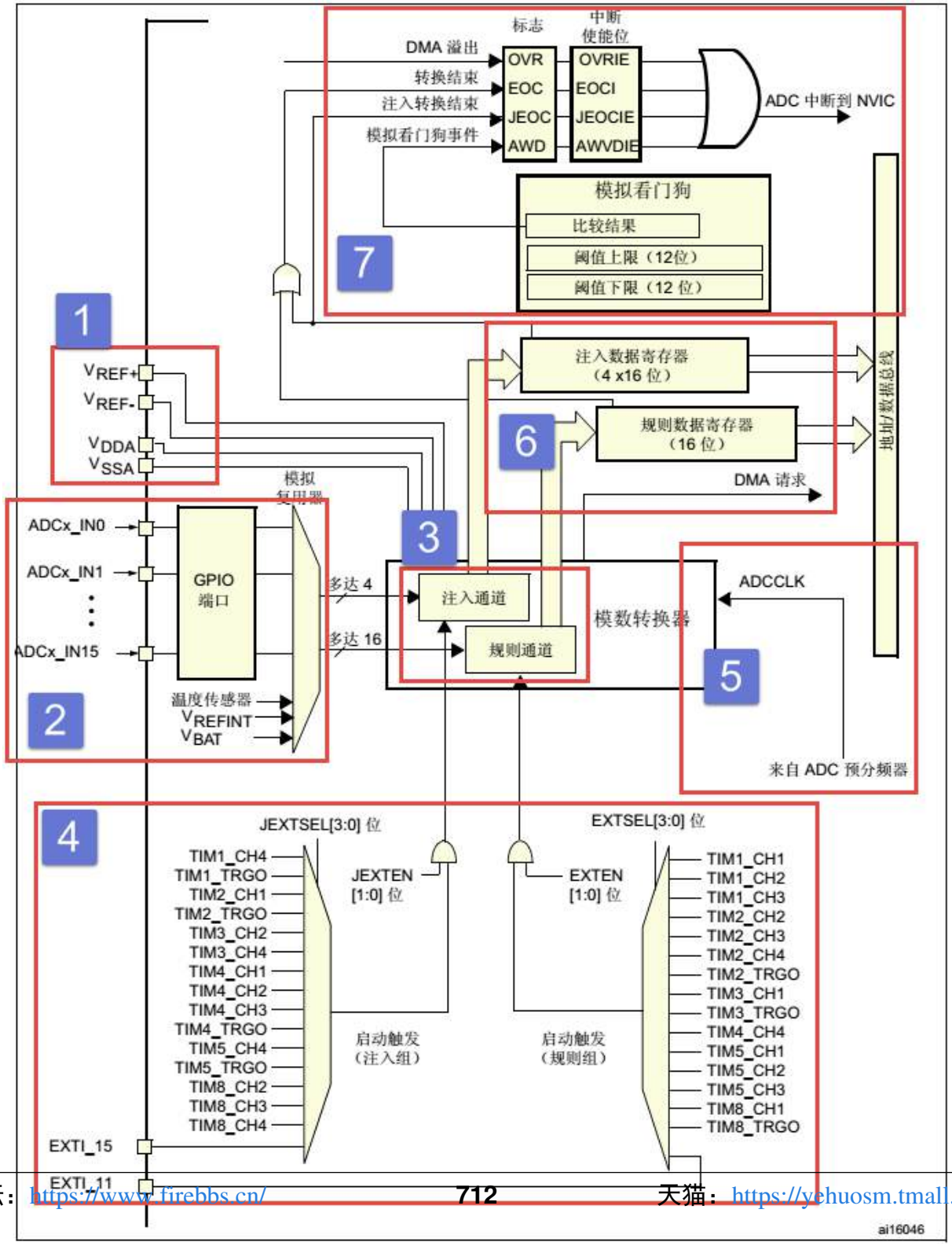


图 29-1 单个 ADC 功能框图

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。

30.2.1 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。

30.2.2 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 19 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 ADC1_IN16 连接到内部的 VSS，通道 ADC1_IN17 连接到了内部参考电压 VREFINT 连接，通道 ADC1_IN18 连接到了芯片内部的温度传感器或者备用电源 VBAT。ADC2 和 ADC3 的通道 16、17、18 全部连接到了内部的 VSS。

STM32F407ZGT6 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	PF3
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	PF4
通道15	PC5	通道15	PC5	通道15	PF5
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 STM32F407ZGT6 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？

30.2.2.1 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。

30.2.2.2 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。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。

30.2.3 □ 转换顺序

30.2.3.1 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

30.2.3.2 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

30.2.4 □ 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 还支持外部事件触发转换，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。

如果使能了外部触发事件，我们还可以通过设置 ADC 控制寄存器 2:ADC_CR2 的 EXTEN[1:0] 和 JEXTEN[1:0] 来控制触发极性，可以有 4 种状态，分别是：禁止触发检测、上升沿检测、下降沿检测以及上升沿和下降沿均检测。

30.2.5 转换时间

30.2.5.1 ADC 时钟

ADC 输入时钟 ADC_CLK 由 PCLK2 经过分频产生，最大值是 36MHz，典型值为 30MHz，分频因子由 ADC 通用控制寄存器 ADC_CCR 的 ADCPRE[1:0] 设置，可设置的分频系数有 2、4、6 和 8，注意这里没有 1 分频。对于 STM32F407ZGT6 我们一般设置 PCLK2=HCLK/2=84MHz。所以程序一般使用 4 分频或者 6 分频。

30.2.5.2 采样时间

ADC 需要若干个 ADC_CLK 周期完成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 3 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 3 个周期，这里说的周期就是 1/ADC_CLK。

ADC 的总转换时间跟 ADC 的输入时钟和采样时间有关，公式为：

Tconv = 采样时间 + 12 个周期

当 ADCCLK = 30MHz，即 PCLK2 为 60MHz，ADC 时钟为 2 分频，采样时间设置为 3 个周期，那么总的转换时为：Tconv = 3 + 12 = 15 个周期 =0.5us。

一般我们设置 PCLK2=84MHz，经过 ADC 预分频器能分频到最大的时钟只能是 21M，采样周期设置为 3 个周期，算出最短的转换时间为 0.7142us，这个才是最常用的。

30.2.6 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。如果是使用双重或者三重模式那规矩组的数据是存放在通用规矩寄存器 ADC_CDR 内的。

30.2.6.1 规则数据寄存器 ADC_DR

ADC 规则组数据寄存器 ADC_DR 只有一个，是一个 32 位的寄存器，只有低 16 位有效并且只是用于独立模式存放转换完成数据。因为 ADC 的最大精度是 12 位，ADC_DR 是 16 位有效，这样允许 ADC 存放数据时候选择左对齐或者右对齐，具体是以哪一种方式存放，由 ADC_CR2 的 11 位 ALIGN 设置。假如设置 ADC 精度为 12 位，如果设置数据为左对齐，那 AD 转换完成数据存放在 ADC_DR 寄存器的 [4:15] 位内；如果为右对齐，则存放在 ADC_DR 寄存器的 [0:11] 位内。

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。

如果没有使用 DMA 传输，我们一般都需要使用 ADC 状态寄存器 ADC_SR 获取当前 ADC 转换的进度状态，进而进行程序控制。

30.2.6.2 注入数据寄存器 ADC_JDRx

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。

30.2.6.3 通用规则数据寄存器 ADC_CDR

规则数据寄存器 ADC_DR 是仅适用于独立模式的，而通用规则数据寄存器 ADC_CDR 是适用于双重和三重模式的。独立模式就是仅仅适用三个 ADC 的其中一个，双重模式就是同时使用 ADC1 和 ADC2，而三重模式就是三个 ADC 同时使用。在双重或者三重模式下一般需要配合 DMA 数据传输使用。

30.2.7 □ 中断

30.2.7.1 转换结束中断

数据转换结束后，可以产生中断，中断分为四种：规则通道转换结束中断，注入转换通道转换结束中断，模拟看门狗中断和溢出中断。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。

30.2.7.2 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。

30.2.7.3 溢出中断

如果发生 DMA 传输数据丢失，会置位 ADC 状态寄存器 ADC_SR 的 OVR 位，如果同时使能了溢出中断，那在转换结束后会产生一个溢出中断。

30.2.7.4 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。

30.2.8 □ 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。

30.3 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 _HandleTypeDef (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 HAL_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__HandleTypeDef 和库函数 HAL_xxx_Init 配合使用是 HAL 库精髓所在，理解了结构体 xxx_HandleTypeDef 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_HandleTypeDef 定义在 stm32f4xx_hal_xxx.h 文件中，库函数 HAL_xxx_Init 定义在 stm32f4xx_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。

30.3.1 ADC_HandleTypeDef 结构体

ADC_HandleTypeDef 结构体定义在 stm32f4xx_adc.h 文件内，具体定义如下：

```
typedef struct {
   ADC_TypeDef           *Instance; /* 寄存器基地址指针 */
   ADC_InitTypeDef       Init; /*ADC 初始化参数结构体 */
   __IO uint32_t         NbrOfCurrentConversionRank;//正在转换序列的 ADC 数目
   DMA_HandleTypeDef     *DMA_Handle; /* DMA 处理程序指针 */
```

(下页继续)

(续上页)

```
    HAL_LockTypeDef       Lock;  /*ADC 锁定对象 */
    __IO uint32_t         State;  /*ADC 通信状态 */
    __IO uint32_t         ErrorCode;  /*ADC 错误码 */
} ADC_HandleTypeDef;
```

*Instance：ADC 寄存器基地址指针，所有参数都是指定基地址后才能正确写入寄存器。

Init：ADC 初始化结构体，下面会详细讲解每一个成员。

*DMA_Handle：DMA 处理程序指针。

Lock：ADC 锁定对象。

State：ADC 转换状态。

ErrorCode：ADC 错误码。

30.3.2 ADC_InitTypeDef 结构体

ADC_InitTypeDef 初始化结构体被 ADC_HandleTypeDef 结构体引用。

ADC_InitTypeDef 结构体定义在 stm32f4xx_hal_adc.h 文件内，具体定义如下：

```
typedef struct {
    uint32_t ClockPrescaler;         /*ADC 时钟分频系数 */
    uint32_t Resolution;             /*ADC 分辨率选择 */
    uint32_t DataAlign;              /* 输出数据对齐方式 */
    uint32_t ScanConvMode;           /* 扫描转换模式 */
    uint32_t EOCSelection;           /* 转换结束标志使用轮询或者中断 */
    uint32_t ContinuousConvMode;     /* 连续转换模式 */
    uint32_t NbrOfConversion;        /* 规格转换序列数目 */
    uint32_t DiscontinuousConvMode; /* 不连续采样模式 */
    uint32_t NbrOfDiscConversion;    /* 不连续采样通道 */
    uint32_t ExternalTrigConv;       /* 外部事件触发选择 */
```

(下页继续)

(续上页)

```
  uint32_t ExternalTrigConvEdge;   /* 外部事件触发极性 */
  uint32_t DMAContinuousRequests; /*DMA 连续请求转换 */
} ADC_InitTypeDef;
```

ADC_Prescaler：ADC 时钟分频系数选择，ADC 时钟是有 PCLK2 分频而来，分频系数决定 ADC 时钟频率，可选的分频系数为 2、4、6 和 8。ADC 最大时钟配置为 36MHz。

ADC_Resolution：配置 ADC 的分辨率，可选的分辨率有 12 位、10 位、8 位和 6 位。分辨率越高，AD 转换数据精度越高，转换时间也越长；分辨率越低，AD 转换数据精度越低，转换时间也越短。

ADC_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。

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。

EOCSelection：可选参数为 ENABLE 和 DISABLE，指定通过轮询和中断来使用 EOC（转换结束）标志进行转换。

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。

NbrOfConversion：AD 规则转换通道数目。

DiscontinuousConvMode：不连续采样模式。一般为禁止模式。

NbrOfDiscConversion：ADC 不连续转换通道数目。

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。

ExternalTrigConvEdge：外部触发极性选择，如果使用外部触发，可以选择触发的极性，可选有禁止触发检测、上升沿触发检测、下降沿触发检测以及上升沿和下降沿均可触发检测。

DMAContinuousRequests：DMA 请求连续转换，开启 DMA 传输时用到。

30.3.3 ADC_ChannelConfTypeDef 结构体

ADC_ChannelConfTypeDef 结构体定义在 stm32f4xx_adc.h 文件内，具体定义如下：

```
typedef struct {
      uint32_t        Channel;         /*ADC 转换通道 */
   uint32_t           Rank;            /*ADC 序列数目 */
   uint32_t           SamplingTime;     /*ADC 采样时间 */
   uint32_t           Offset;                  /* 预留未用到，设为 0 即可 */
} ADC_HandleTypeDef;
```

Channel：ADC 转换通道。

Rank：ADC 序列数目

SamplingTime：ADC 采样时间。

30.4 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。

30.4.1 硬件设计

开发板板载一个贴片滑动变阻器，电路设计见图 *29_5*。

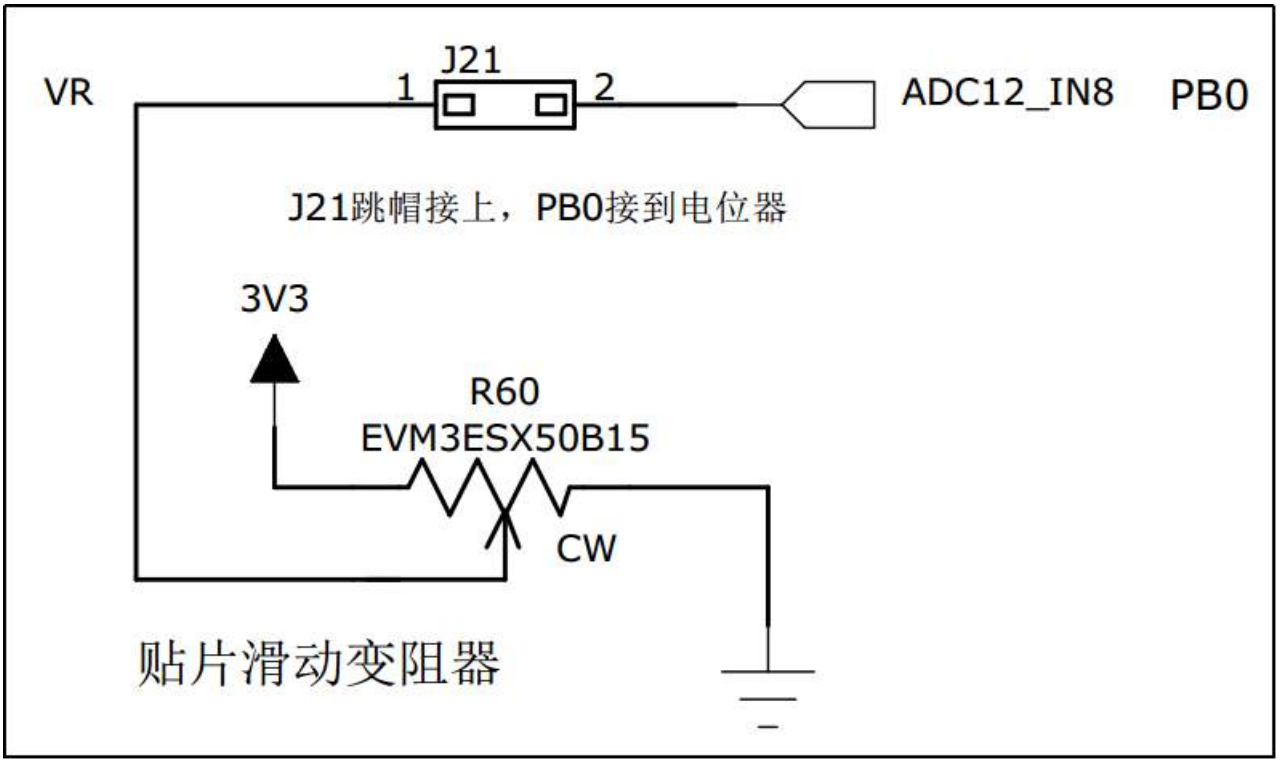


图 29-5 开发板电位器部分原理图

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。

30.4.2 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。

30.4.2.1 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。

30.4.2.2 代码分析

30.4.2.2.1 ADC 宏定义

代码清单 29-1 ADC 宏定义

```
//ADC GPIO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT                GPIOB
#define RHEOSTAT_ADC_GPIO_PIN                 GPIO_PIN_0
#define RHEOSTAT_ADC_GPIO_CLK_ENABLE()        __GPIOB_CLK_ENABLE()
//ADC 序号宏定义
#define RHEOSTAT_ADC                          ADC1
#define RHEOSTAT_ADC_CLK_ENABLE()             __ADC1_CLK_ENABLE()
#define RHEOSTAT_ADC_CHANNEL                  ADC_CHANNEL_8
//ADC DR 寄存器宏定义，ADC 转换后的数字值则存放在这里
#define RHEOSTAT_ADC_DR_ADDR                  ((uint32_t)ADC1+0x4c)
```

使用宏定义引脚信息方便硬件电路改动时程序移植。

30.4.2.2.2 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。

30.4.2.2.3 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
    ADC_Handle.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
```

(下页继续)

(续上页)

```
// 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
HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
```

(下页继续)

(续上页)

```
    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。

30.4.2.2.4 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);
    HAL_NVIC_EnableIRQ(Rheostat_ADC_IRQ);
}
```

在 Rheostat_ADC_NVIC_Config 函数中我们配置了 ADC 转换完成中断的优先级分组和优先级配置。

30.4.2.2.5 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 stm32f4xx_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。

30.4.2.2.6 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.

30.4.2.3 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。

30.5 独立模式多通道采集实验

30.5.1 硬件设计

开发板已通过排针接口把部分 ADC 通道引脚引出，我们可以根据需要选择使用。实际使用时候必须注意保存 ADC 引脚是单独使用的，不可能与其他模块电路共用同一引脚。

30.5.2 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。

30.5.2.1 编程要点

1) 初始化配置 ADC 目标引脚为模拟输入模式；

2) 使能 ADC 时钟和 DMA 时钟；

3) 配置 DMA 从 ADC 规矩数据寄存器传输数据到我们指定的存储区；

4) 配置通用 ADC 为独立模式，采样 4 分频；

5) 设置 ADC 为 12 位分辨率，启动扫描，连续转换，不需要外部触发；

6) 设置 ADC 转换通道顺序及采样时间；

7) 使能 DMA 请求，DMA 在 AD 转换完自动传输数据到指定的存储区；

8) 启动 ADC 转换；

9) 使能软件触发 ADC 转换。

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。

30.5.2.2 代码分析

30.5.2.2.1 ADC 宏定义

列表 6: 代码清单 29-7 多通道 ADC 相关宏定义

```
#define RHEOSTAT_NOFCHANEL      3

/*====================== 通道 1 IO=======================*/
// PC3 通过调帽接电位器
// ADC IO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT1             GPIOB
#define RHEOSTAT_ADC_GPIO_PIN1              GPIO_PIN_0
#define RHEOSTAT_ADC_GPIO_CLK1_ENABLE()     __GPIOB_CLK_ENABLE()
#define RHEOSTAT_ADC_CHANNEL1               ADC_CHANNEL_8
/*====================== 通道 2 IO =======================*/
// PA4 通过调帽接光敏电阻
// ADC IO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT2             GPIOB
#define RHEOSTAT_ADC_GPIO_PIN2              GPIO_PIN_1
```

(下页继续)

(续上页)

```
#define RHEOSTAT_ADC_GPIO_CLK2_ENABLE()      __GPIOB_CLK_ENABLE()
#define RHEOSTAT_ADC_CHANNEL2                ADC_CHANNEL_9
/*====================== 通道 3 IO ======================*/
// PA6 悬空，可用杜邦线接 3V3 或者 GND 来实验
// ADC IO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT3              GPIOA
#define RHEOSTAT_ADC_GPIO_PIN3               GPIO_PIN_6
#define RHEOSTAT_ADC_GPIO_CLK3_ENABLE()      __GPIOA_CLK_ENABLE()
#define RHEOSTAT_ADC_CHANNEL3                ADC_CHANNEL_6

// ADC 序号宏定义
#define RHEOSTAT_ADC                         ADC1
#define RHEOSTAT_ADC_CLK_ENABLE()            __ADC1_CLK_ENABLE()

// ADC DR 寄存器宏定义，ADC 转换后的数字值则存放在这里
#define RHEOSTAT_ADC_DR_ADDR                 ((uint32_t)ADC1+0x4c)

// ADC DMA 通道宏定义，这里我们使用 DMA 传输
#define RHEOSTAT_ADC_DMA_CLK_ENABLE()        __DMA2_CLK_ENABLE()
#define RHEOSTAT_ADC_DMA_CHANNEL             DMA_CHANNEL_0
#define RHEOSTAT_ADC_DMA_STREAM              DMA2_Stream0
```

定义多个通道进行多通道 ADC 实验，并且定义 DMA 相关配置。

30.5.2.2.2 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;
```

(下页继续)

(续上页)

```
    /*====================== 通道 1======================*/
    // 使能 GPIO 时钟
    RHEOSTAT_ADC_GPIO_CLK1_ENABLE();
    // 配置 IO
    GPIO_InitStructure.Pin = RHEOSTAT_ADC_GPIO_PIN1;
    GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStructure.Pull = GPIO_NOPULL ; //不上拉不下拉
    HAL_GPIO_Init(RHEOSTAT_ADC_GPIO_PORT1, &GPIO_InitStructure);
    /*====================== 通道 2======================*/
    // 使能 GPIO 时钟
    RHEOSTAT_ADC_GPIO_CLK2_ENABLE();
    // 配置 IO
    GPIO_InitStructure.Pin = RHEOSTAT_ADC_GPIO_PIN2;
    GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStructure.Pull = GPIO_NOPULL ; //不上拉不下拉
    HAL_GPIO_Init(RHEOSTAT_ADC_GPIO_PORT2, &GPIO_InitStructure);
    /*====================== 通道 3======================*/
    // 使能 GPIO 时钟
    RHEOSTAT_ADC_GPIO_CLK3_ENABLE();
    // 配置 IO
    GPIO_InitStructure.Pin = RHEOSTAT_ADC_GPIO_PIN3;
    GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStructure.Pull = GPIO_NOPULL ; //不上拉不下拉
    HAL_GPIO_Init(RHEOSTAT_ADC_GPIO_PORT3, &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。

30.5.2.2.3 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--------------------------
    // ADC1 使用 DMA2，数据流 0，通道 0，这个是手册固定死的
    // 开启 DMA 时钟
    RHEOSTAT_ADC_DMA_CLK_ENABLE();
    // 数据传输通道
    DMA_Init_Handle.Instance = RHEOSTAT_ADC_DMA_STREAM;
    // 数据传输方向为外设到存储器
    DMA_Init_Handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    // 外设寄存器只有一个，地址不用递增
    DMA_Init_Handle.Init.PeriphInc = DMA_PINC_DISABLE;
    // 存储器地址固定
    DMA_Init_Handle.Init.MemInc = DMA_MINC_ENABLE;
    // // 外设数据大小为半字，即两个字节
    DMA_Init_Handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    //  存储器数据大小也为半字，跟外设数据大小相同
    DMA_Init_Handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    // 循环传输模式
    DMA_Init_Handle.Init.Mode = DMA_CIRCULAR;
    // DMA 传输通道优先级为高，当使用一个 DMA 通道时，优先级设置不影响
    DMA_Init_Handle.Init.Priority = DMA_PRIORITY_HIGH;
    // 禁止 DMA FIFO ，使用直连模式
    DMA_Init_Handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
    // FIFO 大小，FIFO 模式禁止时，这个不用配置
    DMA_Init_Handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_HALFFULL;
    DMA_Init_Handle.Init.MemBurst = DMA_MBURST_SINGLE;
    DMA_Init_Handle.Init.PeriphBurst = DMA_PBURST_SINGLE;
    // 选择 DMA 通道，通道存在于流中
```

(下页继续)

(续上页)

```
  DMA_Init_Handle.Init.Channel = RHEOSTAT_ADC_DMA_CHANNEL;
  //初始化 DMA 流，流相当于一个大的管道，管道里面有很多通道
  HAL_DMA_Init(&DMA_Init_Handle);
  HAL_DMA_Start (&DMA_Init_Handle,RHEOSTAT_ADC_DR_ADDR,
  (uint32_t)&ADC_ConvertedValue,RHEOSTAT_NOFCHANEL);
  // 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
  ADC_Handle.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
  // ADC 分辨率
  ADC_Handle.Init.Resolution = ADC_RESOLUTION_12B;
  // 扫描模式，多通道采集才需要
  ADC_Handle.Init.ScanConvMode = ENABLE;
  // 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;
  //转换通道个数
  ADC_Handle.Init.NbrOfConversion = RHEOSTAT_NOFCHANEL;
  //使能连续转换请求
  ADC_Handle.Init.DMAContinuousRequests = ENABLE;
```

(下页继续)

(续上页)

```
    //转换完成标志
    ADC_Handle.Init.EOCSelection          = DISABLE;
    // 初始化 ADC
    HAL_ADC_Init(&ADC_Handle);
    //-------------------------------------------------------------------
    // 配置 ADC 通道 1 转换顺序为 1，第一个转换，采样时间为 3 个时钟周期
    ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL1;
    ADC_Config.Rank         = 1;
    ADC_Config.SamplingTime = ADC_SAMPLETIME_3CYCLES;// 采样时间间隔
    ADC_Config.Offset       = 0;
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    // 配置 ADC 通道 2 转换顺序为 2，第二个转换，采样时间为 3 个时钟周期
    ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL2;
    ADC_Config.Rank         = 2;
    ADC_Config.SamplingTime = ADC_SAMPLETIME_3CYCLES; // 采样时间间隔
    ADC_Config.Offset       = 0;
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    // 配置 ADC 通道 3 转换顺序为 3，第三个转换，采样时间为 3 个时钟周期
    ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL3;
    ADC_Config.Rank         = 3;
    ADC_Config.SamplingTime = ADC_SAMPLETIME_3CYCLES;// 采样时间间隔
    ADC_Config.Offset       = 0;
    HAL_ADC_ConfigChannel(&ADC_Handle, &ADC_Config);
    HAL_ADC_Start_DMA(&ADC_Handle, (uint32_t*)&ADC_ConvertedValue,1);
}
```

首先，我们使用了 DMA_HandleTypeDef 定义了一个 DMA 初始化类型变量，该结构体内容我们在 DMA 篇已经做了非常详细的讲解；另外还使用 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。

调用 RHEOSTAT_ADC_DMA_CLK_ENABLE() 和 RHEOSTAT_ADC_CLK_ENABLE() 函数开启 ADC 时钟以及开启 DMA 时钟。

我们需要对 DMA 进行必要的配置。首先设置外设基地址就是 ADC 的规则数据寄存器地址；存储器的地址就是我们指定的数据存储区空间，ADC_ConvertedValue 是我们定义的一个全局数组名，它是一个无符号 16 位含有 4 个元素的整数数组；ADC 规则转换对应只有一个数据寄存器所以地址不能递增，而我们定义的存储区是专门用来存放不同通道数据的，所以需要自动地址递增。ADC 的规则数据寄存器只有低 16 位有效，实际存放的数据只有 12 位而已，所以设置数据大小为半字大小。ADC 配置为连续转换模式 DMA 也设置为循环传输模式。设置好 DMA 相关参数后就使用 HAL_DMA_Init 函数初始化。

接下来我们使用 ADC_HandleTypeDef 和 ADC_ChannelConfTypeDef 来配置 ADC 为独立模式、分频系数为 4、不需要设置 DMA 模式、20 个周期的采样延迟，并调用 HAL_ADC_ConfigChannel 函数完成 ADC 通道的配置。

我们使用 ADC_InitTypeDef 结构体变量 ADC_InitStructure 来配置 ADC1 为 12 位分辨率、使能扫描模式、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 4，并调用 ADC_Init 函数完成 ADC3 工作环境配置。

ADC_ChannelConfTypeDef 函数用来绑定 ADC 通道转换顺序和采样时间。分别绑定四个 ADC 通道引脚并设置相应的转换顺序，控制是否使能 ADC 的 DMA 请求，如果使能请求，并调用 HAL_ADC_Start_DMA 函数控制 ADC 转换启动。在 ADC 转换完成后就请求 DMA 实现数据传输。

30.5.2.2.4 主函数

列表 9: 代码清单 29-10 主函数

```
int main(void)
{
   /* 配置系统时钟为 168 MHz */
   SystemClock_Config();
   /* 初始化 USART1 配置模式为 115200 8-N-1 */
   UARTx_Config();
   Rheostat_Init();
   while (1) {
```

(下页继续)

(续上页)

```
    ADC_ConvertedValueLocal[0] =(float) ADC_ConvertedValue[0]/4096*(float)3.
↪3;
    ADC_ConvertedValueLocal[1] =(float) ADC_ConvertedValue[1]/4096*(float)3.
↪3;
    ADC_ConvertedValueLocal[2] =(float) ADC_ConvertedValue[2]/4096*(float)3.
↪3;
    printf("\r\n CH1_PC3 value = %f V \r\n",ADC_ConvertedValueLocal[0]);
    printf("\r\n CH2_PA4 value = %f V \r\n",ADC_ConvertedValueLocal[1]);
    printf("\r\n CH3_PA6 value = %f V \r\n",ADC_ConvertedValueLocal[2]);
    printf("\r\n\r\n");
    Delay(0xffffff);
    }
}
```

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() 和 Rheostat_ADC_Mode_Config ()。

Delay 函数只是一个简单的延时函数。

我们配置了 DMA 数据传输所以它会自动把 ADC 转换完成后数据保存到数组 ADC_ConvertedValue 内，我们只要直接使用数组就可以了。经过简单地计算就可以得到每个通道对应的实际电压。

最后就是把相关数据打印至串口调试助手。

30.5.2.3 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。

30.6 三重 ADC 交替模式采集实验

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。双重或者三重 ADC 的机制使用两个或以上 ADC 同时采样两个或以上不同通道的数据或者使用两个或以上 ADC 交叉采集同一通道的数据。双重或者三重 ADC 模式较独立模式一个最大的优势就是转换速度快。

我们这里只介绍三重 ADC 交替模式，关于双重或者三重 ADC 的其他模式与之类似，可以参考三重 ADC 交替模式使用。三重 ADC 交替模式是针对同一通道的使用三个 ADC 交叉采集，就是在 ADC1 采样完等几个时钟周期后 ADC2 开始采样，此时 ADC1 处在转换阶段，当 ADC2 采样完成再等几个时钟周期后 ADC3 就进行采样此时 ADC1 和 ADC2 处在转换阶段，如果 ADC3 采样完成并且 ADC1 已经转换完成那么就可以准备下一轮的循环，这样充分利用转换阶段时间达到增快采样速度的效果。AD 转换过程见图 *29_6*，利用 ADC 的转换阶段时间另外一个 ADC 进行采样，而不用像独立模式必须等待采样和转换结束后才进行下一次采样及转换。

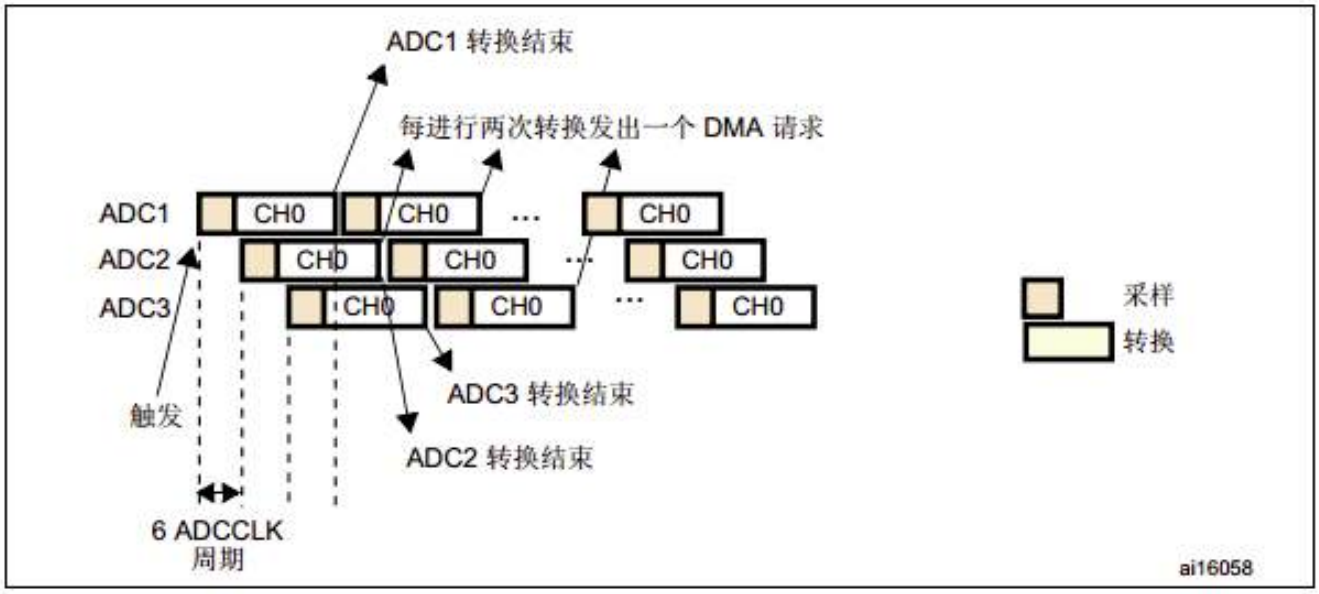


图 29-6 三重 ADC 同步规则模式

30.6.1 硬件设计

三重 ADC 交叉模式是针对同一个通道的 ADC 采集模式，这种情况跟前门小节的单通道实验非常类似，只是同时使用三个 ADC 对同一通道进行采集，所以电路设计与之相同即可，具体可参考图 *29_5*。

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。

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。

30.6.2.1 编程要点

1) 初始化配置 ADC 目标引脚为模拟输入模式；

2) 使能 ADC1、ADC2、ADC3 以及 DMA 时钟；

3) 配置 DMA 控制将 ADC 通用规矩数据寄存器数据转存到指定存储区；

4) 配置通用 ADC 为三重 ADC 交替模式，采样 4 分频，使用 DMA 模式 2；

5) 设置 ADC1、ADC2 和 ADC3 为 12 位分辨率，禁用扫描，连续转换，不需要外部触发；

6) 设置 ADC1、ADC2 和 ADC3 转换通道顺序及采样时间；

7) 使能 ADC1 的 DMA 请求，在 ADC 转换完后自动请求 DMA 进行数据传输；

8) 启动 ADC1、ADC2 和 ADC3 转换；

9) 使能软件触发 ADC 转换。

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。

30.6.2.2 代码分析

30.6.2.2.1 ADC 宏定义

列表 10: 代码清单 29-11 多通道 ADC 相关宏定义

```
#define RHEOSTAT_NOFCHANEL      3
// PC3 通过调帽接电位器
// ADC IO 宏定义
#define RHEOSTAT_ADC_GPIO_PORT              GPIOC
#define RHEOSTAT_ADC_GPIO_PIN               GPIO_PIN_2
#define RHEOSTAT_ADC_GPIO_CLK_ENABLE()      __GPIOC_CLK_ENABLE()
// ADC 序号宏定义
#define RHEOSTAT_ADC1                       ADC1
```

(下页继续)

(续上页)

```
#define RHEOSTAT_ADC2                        ADC2
#define RHEOSTAT_ADC3                        ADC3
#define RHEOSTAT_ADC1_CLK_ENABLE()           __ADC1_CLK_ENABLE()
#define RHEOSTAT_ADC2_CLK_ENABLE()           __ADC2_CLK_ENABLE()
#define RHEOSTAT_ADC3_CLK_ENABLE()           __ADC3_CLK_ENABLE()
#define RHEOSTAT_ADC_CHANNEL                 ADC_CHANNEL_13
// ADC DR 寄存器宏定义，ADC 转换后的数字值则存放在这里
#define RHEOSTAT_ADC_DR_ADDR                 ((uint32_t)0x40012308)
// ADC DMA 通道宏定义，这里我们使用 DMA 传输
#define RHEOSTAT_ADC_DMA_CLK_ENABLE()         __DMA2_CLK_ENABLE()
#define RHEOSTAT_ADC_DMA_CHANNEL              DMA_CHANNEL_0
#define RHEOSTAT_ADC_DMA_STREAM               DMA2_Stream0
```

双重或者三重 ADC 需要使用通用规则数据寄存器 ADC_CDR，这点跟独立模式不同。定义电位器动触点引脚作为三重 ADC 的模拟输入，三个 ADC 都是采集同一路信号。

30.6.2.2.2 ADC GPIO 初始化函数

列表 11: 代码清单 29-12 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。

30.6.2.2.3 配置三重 ADC 交替模式

列表 12: 代码清单 29-13 三重 ADC 交替模式配置

```
static void Rheostat_ADC_Mode_Config(void)
{
    ADC_MultiModeTypeDef    mode;
    // ------------------DMA Init 结构体参数 初始化--------------------------
    // ADC1 使用 DMA2，数据流 0，通道 0，这个是手册固定死的
    // 开启 DMA 时钟
    RHEOSTAT_ADC_DMA_CLK_ENABLE();
    // 数据传输通道
    DMA_Init_Handle.Instance = RHEOSTAT_ADC_DMA_STREAM;
    // 数据传输方向为外设到存储器
    DMA_Init_Handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    // 外设寄存器只有一个，地址不用递增
    DMA_Init_Handle.Init.PeriphInc = DMA_PINC_DISABLE;
    // 存储器地址固定
    DMA_Init_Handle.Init.MemInc = DMA_MINC_ENABLE;
    // // 外设数据大小为半字，即两个字节
    DMA_Init_Handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    //  存储器数据大小也为半字，跟外设数据大小相同
    DMA_Init_Handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    // 循环传输模式
    DMA_Init_Handle.Init.Mode = DMA_CIRCULAR;
    // DMA 传输通道优先级为高，当使用一个 DMA 通道时，优先级设置不影响
    DMA_Init_Handle.Init.Priority = DMA_PRIORITY_HIGH;
    // 禁止 DMA FIFO ，使用直连模式
    DMA_Init_Handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
```

(下页继续)

(续上页)

```
  // FIFO 大小，FIFO 模式禁止时，这个不用配置
  DMA_Init_Handle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_HALFFULL;
  DMA_Init_Handle.Init.MemBurst = DMA_MBURST_SINGLE;
  DMA_Init_Handle.Init.PeriphBurst = DMA_PBURST_SINGLE;
  // 选择 DMA 通道，通道存在于流中
  DMA_Init_Handle.Init.Channel = RHEOSTAT_ADC_DMA_CHANNEL;
  //初始化 DMA 流，流相当于一个大的管道，管道里面有很多通道
  HAL_DMA_Init(&DMA_Init_Handle);
  // 开启 ADC 时钟
  RHEOSTAT_ADC1_CLK_ENABLE();
  RHEOSTAT_ADC2_CLK_ENABLE();
  RHEOSTAT_ADC3_CLK_ENABLE();
  // -------------------ADC1 Init 结构体 参数 初始化----------------------
  // ADC1
  ADC_Handle1.Instance = RHEOSTAT_ADC1;
  // 时钟为 fpclk 4 分频
  ADC_Handle1.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
  // ADC 分辨率
  ADC_Handle1.Init.Resolution = ADC_RESOLUTION_12B;
  // 禁止扫描模式，多通道采集才需要
  ADC_Handle1.Init.ScanConvMode = DISABLE;
  // 连续转换
  ADC_Handle1.Init.ContinuousConvMode = ENABLE;
  // 非连续转换
  ADC_Handle1.Init.DiscontinuousConvMode = DISABLE;
  // 非连续转换个数
  ADC_Handle1.Init.NbrOfDiscConversion   = 0;
  //禁止外部边沿触发
  ADC_Handle1.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
```

(下页继续)

(续上页)

```
  ADC_Handle1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  //转换通道个数
  ADC_Handle1.Init.NbrOfConversion = 1;
  //使能连续转换请求
  ADC_Handle1.Init.DMAContinuousRequests = ENABLE;
  //转换完成标志
  ADC_Handle1.Init.EOCSelection          = DISABLE;
  // 初始化 ADC
  HAL_ADC_Init(&ADC_Handle1);
  //-------------------------------------------------------------------
  // 配置 ADC1 通道 13 转换顺序为 1，第一个转换，采样时间为 3 个时钟周期
  ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL;
  ADC_Config.Rank         = 1;
  ADC_Config.SamplingTime = ADC_SAMPLETIME_3CYCLES;// 采样时间间隔
  ADC_Config.Offset       = 0;
  HAL_ADC_ConfigChannel(&ADC_Handle1, &ADC_Config);
  // -------------------ADC2 Init 结构体 参数 初始化----------------------
  // ADC2
  ADC_Handle2.Instance = RHEOSTAT_ADC2;
  // 时钟为 fpclk 4 分频
  ADC_Handle2.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
  // ADC 分辨率
  ADC_Handle2.Init.Resolution = ADC_RESOLUTION_12B;
  // 禁止扫描模式，多通道采集才需要
  ADC_Handle1.Init.ScanConvMode = DISABLE;
  // 连续转换
  ADC_Handle2.Init.ContinuousConvMode = ENABLE;
  // 非连续转换
  ADC_Handle2.Init.DiscontinuousConvMode = DISABLE;
  // 非连续转换个数
  ADC_Handle2.Init.NbrOfDiscConversion   = 0;
  //禁止外部边沿触发
```

(下页继续)

(续上页)

```
  ADC_Handle2.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_Handle2.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  //转换通道个数
  ADC_Handle2.Init.NbrOfConversion = 1;
  //使能连续转换请求
  ADC_Handle2.Init.DMAContinuousRequests = ENABLE;
  //转换完成标志
  ADC_Handle2.Init.EOCSelection          = DISABLE;
  // 初始化 ADC
  HAL_ADC_Init(&ADC_Handle2);
  // 配置 ADC2 通道 13 转换顺序为 1，第一个转换，采样时间为 3 个时钟周期
  ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL;
  ADC_Config.Rank         = 1;
  ADC_Config.SamplingTime = ADC_SAMPLETIME_3CYCLES; // 采样时间间隔
  ADC_Config.Offset       = 0;
  HAL_ADC_ConfigChannel(&ADC_Handle2, &ADC_Config);

  // -------------------ADC33 Init 结构体 参数 初始化--------------------
  // ADC3
  ADC_Handle3.Instance = RHEOSTAT_ADC3;
  // 时钟为 fpclk 4 分频
  ADC_Handle3.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
  // ADC 分辨率
  ADC_Handle3.Init.Resolution = ADC_RESOLUTION_12B;
  // 禁止扫描模式，多通道采集才需要
  ADC_Handle1.Init.ScanConvMode = DISABLE;
  // 连续转换
  ADC_Handle3.Init.ContinuousConvMode = ENABLE;
  // 非连续转换
```

(下页继续)

(续上页)

```
  ADC_Handle3.Init.DiscontinuousConvMode = DISABLE;
  // 非连续转换个数
  ADC_Handle3.Init.NbrOfDiscConversion   = 0;
  //禁止外部边沿触发
  ADC_Handle3.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_Handle3.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  //转换通道个数
  ADC_Handle3.Init.NbrOfConversion = 1;
  //使能连续转换请求
  ADC_Handle3.Init.DMAContinuousRequests = ENABLE;
  //转换完成标志
  ADC_Handle3.Init.EOCSelection          = DISABLE;
  // 初始化 ADC
  HAL_ADC_Init(&ADC_Handle3);
  // 配置 ADC3 通道 13 转换顺序为 1，第一个转换，采样时间为 3 个时钟周期
  ADC_Config.Channel      = RHEOSTAT_ADC_CHANNEL;
  ADC_Config.Rank         = 1;
  ADC_Config.SamplingTime = ADC_SAMPLETIME_3CYCLES;// 采样时间间隔
  ADC_Config.Offset       = 0;
  HAL_ADC_ConfigChannel(&ADC_Handle3, &ADC_Config);

  /* 配置三重 AD 采样 */
  mode.Mode = ADC_TRIPLEMODE_INTERL;
  mode.DMAAccessMode = ADC_DMAACCESSMODE_2;
  mode.TwoSamplingDelay = ADC_TWOSAMPLINGDELAY_5CYCLES;

  HAL_ADCEx_MultiModeConfigChannel(&ADC_Handle1, &mode);

  HAL_ADC_Start(&ADC_Handle2);
```

(下页继续)

(续上页)

```
    HAL_ADC_Start(&ADC_Handle3);

    __HAL_LINKDMA(&ADC_Handle1, DMA_Handle, DMA_Init_Handle);
    __HAL_LINKDMA(&ADC_Handle2, DMA_Handle, DMA_Init_Handle);
    __HAL_LINKDMA(&ADC_Handle3, DMA_Handle, DMA_Init_Handle);
    HAL_ADCEx_MultiModeStart_DMA(&ADC_Handle1, (uint32_t *)ADC_
↪ConvertedValue, 3);
}
```

首先，我们使用了 DMA_HandleTypeDef 定义了一个 DMA 初始化类型变量，该结构体内容我们在 DMA 篇已经做了非常详细的讲解；另外还使用 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。

调用 RHEOSTAT_ADC_DMA_CLK_ENABLE() 和 RHEOSTAT_ADC_CLK_ENABLE() 函数开启 ADC 时钟以及开启 DMA 时钟。

我们需要对 DMA 进行必要的配置。首先设置外设基地址就是 ADC 的通用规则数据寄存器地址；存储器的地址就是我们指定的数据存储区空间，ADC_ConvertedValue 是我们定义的一个全局数组名，它是一个无符号 32 位有三个元素的整数数字；ADC 规则转换对应只有一个数据寄存器所以地址不能递增，我们指定的存储区也需要递增地址。ADC 的通用规则数据寄存器是 32 位有效，我们配置 ADC 为 DMA 模式 2，设置数据大小为字大小。ADC 配置为连续转换模式 DMA 也设置为循环传输模式。设置好 DMA 相关参数后就使能 DMA 的 ADC 通道。

接下来我们使用 ADC_InitTypeDef 结构体变量 ADC_InitStructure 来配置 ADC1 为 12 位分辨率、不使用扫描模式、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 ADC_Init 函数完成 ADC1 工作环境配置。ADC2 和 ADC3 使用与 ADC1 相同配置即可。

ADC_ChannelConfTypeDef 函数用来绑定 ADC 通道转换顺序和采样时间。绑定 ADC 通道引脚并设置相应的转换顺序。接下来我们使用 ADC_MultiModeTypeDef 结构体变量 mode 来配置 ADC 为三重 ADC 交替模式、分频系数为 4、需要设置 DMA 模式 2、10 个周期的采样延迟。

HAL_ADC_Start 函数控制 ADC 转换启动。

HAL_ADCEx_MultiModeConfigChannel 函数控制是否使能 ADC 的 DMA 请求，如果使能请求，并调用 HAL_ADCEx_MultiModeStart_DMA 函数使能 DMA，则在 ADC 转换完成后就请求 DMA 实现数据传输。三重模式只需使能 ADC1 的 DMA 通道。

30.6.2.3 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 168 MHz */
   SystemClock_Config();

   /* 初始化 USART1 配置模式为 115200 8-N-1 */
   DEBUG_USART_Config();

   Rheostat_Init();
   while (1) {
         Delay(0xffffee);
         ADC_ConvertedValueLocal[0] =(float)((uint16_t)
                                    ADC_ConvertedValue[0]*3.3/4096);
         ADC_ConvertedValueLocal[1] =(float)((uint16_t)
                                    ADC_ConvertedValue[1]*3.3/4096);
         ADC_ConvertedValueLocal[2] =(float)((uint16_t)
                                    ADC_ConvertedValue[2]*3.3/4096);
         printf("\r\n The current AD value = 0x%08X \r\n",
               ADC_ConvertedValue[0]);
         printf("\r\n The current AD value = 0x%08X \r\n",
               ADC_ConvertedValue[1]);
         printf("\r\n The current AD value = 0x%08X \r\n",
               ADC_ConvertedValue[2]);
         printf("\r\n The current ADC1 value = %f V \r\n",
```

(下页继续)

(续上页)

```
                ADC_ConvertedValueLocal[0]);
        printf("\r\n The current ADC2 value = %f V \r\n",
                ADC_ConvertedValueLocal[1]);
        printf("\r\n The current ADC3 value = %f V \r\n",
                ADC_ConvertedValueLocal[2]);
    }
}
```

主函数先初始化系统时钟再调用 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() 和 Rheostat_ADC_Mode_Config ()。

Delay 函数只是一个简单的延时函数。

我们配置了 DMA 数据传输所以它会自动把 ADC 转换完成后数据保存到数组变量 ADC_ConvertedValue 内，根据 DMA 模式 2 的数据存放规则，ADC_ConvertedValue[0] 的低 16 位存放 ADC1 数据、高 16 位存放 ADC2 数据，ADC_ConvertedValue[1] 的低 16 位存放 ADC3 数据、高 16 位存放 ADC1 数据，ADC_ConvertedValue[2] 的低 16 位存放 ADC2 数据、高 16 位存放 ADC3 数据，我们可以根据需要提取出对应 ADC 的转换结果数据。经过简单地计算就可以得到每个 ADC 对应的实际电压。

最后就是把相关数据打印至串口调试助手。

30.6.2.4 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 31 章　TIM—基本定时器

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

学习本章时，配合《STM32F4xx 参考手册》基本定时器章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。

特别说明，本书内容是以 STM32F4xx 系列控制器资源讲解。

31.1 TIM 简介

定时器 (Timer) 最基本的功能就是定时了，比如定时发送 USART 数据、定时采集 AD 数据等等。如果把定时器与 GPIO 结合起来使用的话可以实现非常丰富的功能，可以测量输入信号的脉冲宽度，可以生产输出波形。定时器生产 PWM 控制电机状态是工业控制普遍方法，这方面知识非常有必要深入了解。

STM32F4xx 系列控制器有 2 个高级控制定时器、10 个通用定时器和 2 个基本定时器，还有 2 个看门狗定时器。看门狗定时器不在本章讨论范围，有专门讲解的章节。控制器上所有定时器都是彼此独立的，不共享任何资源。各个定时器特性参考表 31-1。

表 31-1 各个定时器特性

表 31-1 各个定时器特性

定时器类型	Timer	计数器分辨率	计数器类型	预分频系数	DMA请求生成	捕获/比较通道	互补输出	最大接口时钟(MHz)	最大定时器时钟(MHz)
高级控制	TIM1和TIM8	16位	递增、递减、递增/递减	1~65536(整数)	有	4	有	84(APB2)	168
通用	TIM2，TIM5	32位	递增、递减、递增/递减	1~65536(整数)	有	4	无	42(APB1)	84/168
	TIM3，TIM4	16位	递增、递减、递增/递减	1~65536(整数)	有	4	无	42(APB1)	84/168
	TIM9	16位	递增	1~65536(整数)	无	2	无	84(APB2)	168
	TIM10，TIM11	16位	递增	1~65536(整数)	无	1	无	84(APB2)	168
	TIM12	16位	递增	1~65536(整数)	无	2	无	42(APB1)	84/168
	TIM13，TIM14	16位	递增	1~65536(整数)	无	1	无	42(APB1)	84/168
基本	TIM6和TIM7	16位	递增	1~65536(整数)	有	0	无	42(APB1)	84/168

其中最大定时器时钟可通过 RCC_DCKCFGR 寄存器配置为 84MHz 或者 168MHz。

定时器功能强大，这一点透过《STM32F4xx 中文参考手册》讲解定时器内容就有 160 多页就显而易见了。定时器篇幅长，内容多，对于新手想完全掌握确实有些难度，特别参考手册是先介绍高级控制定时器，然后介绍通用定时器，最后才介绍基本定时器。实际上，就功能上来说通用定时器包含所有基本定时器功能，而高级控制定时器包含通用定时器所有功能。所以高级控制定时器功能繁多，但也是最难理解的，本章我们先选择最简单的基本定时器进行讲解。

31.2 基本定时器

基本定时器比高级控制定时器和通用定时器功能少，结构简单，理解起来更容易，我们就开始先讲解基本定时器内容。基本定时器主要两个功能，第一就是基本定时功能，生成时基，第二就是专门用于驱动数模转换器 (DAC)。关于驱动 DAC 具体应用参考 DAC 章节。

控制器有两个基本定时器 TIM6 和 TIM7，功能完全一样，但所用资源彼此都完全独立，可以同时使用。在本章内容中，以 TIMx 统称基本定时器。

基本上定时器 TIM6 和 TIM7 是一个 16 位向上递增的定时器，当我在自动重载寄存器 (TIMx_ARR) 添加一个计数值后并使能 TIMx，计数寄存器 (TIMx_CNT) 就会从 0 开始递增，当 TIMx_CNT 的数值与 TIMx_ARR 值相同时就会生成事件并把 TIMx_CNT 寄存器清 0，完成一次循环过程。如果没有停止定时器就循环执行上述过程。这些只是大概的流程，希望大家有个感性认识，下面细讲整个过程。

31.3 基本定时器功能框图

基本定时器的功能框图包含了基本定时器最核心内容，掌握了功能框图，对基本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 31-1。

首先先看图 31-1 中绿色框内容，第一个是带有阴影的方框，方框内容一般是一个寄存器名称，比如图中主体部分的自动重载寄存器 (TIMx_ARR) 或 PSC 预分频器 (TIMx_PSC)，这里要特别突出的是阴影这个标志的作用，它表示这个寄存器还自带有影子寄存器，在硬件结构上实际是有两个寄存器，源寄存器是我们可以进行读写操作，而影子寄存器我们是完全无法操作的，有内部硬件使用。影子寄存器是在程序运行时真正起到作用的，源寄存器只是给我们读写用的，只有在特定时候 (特定事件发生时) 才把源寄存器的值拷贝给它的影子寄存器。多个影子寄存器一起使用可以到达同步更新多个寄存器内容的目的。

接下来是一个指向右下角的图标，它表示一个事件，而一个指向右上角的图标表示中断和 DMA 输出。这个我们把它放在图中主体更好理解。图中的自动重载寄存器有影子寄存器，它左边有一个带有“U”字母的事件图标，表示在更新事件生成时就把自动重载寄存器内容拷贝到影子寄存器内，这个与上面分析是一致。寄存器右边的事件图标、中断和 DMA 输出图标表示在自动重载寄存器值与计数器寄存器值相等时生成事件、中断和 DMA 输出。

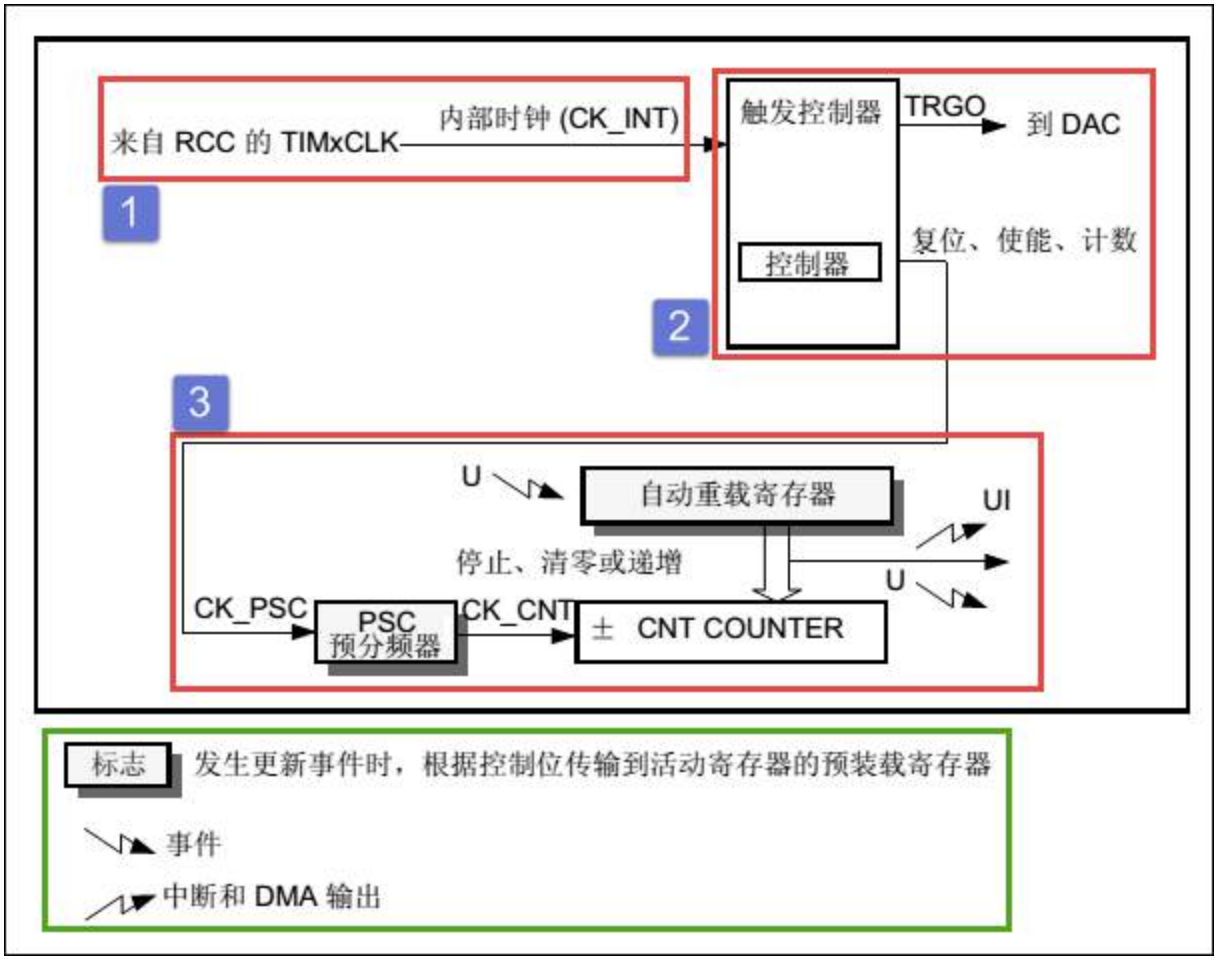


图 31-1 基本定时器功能框图

31.3.1 时钟源

定时器要实现计数必须有个时钟源，基本定时器时钟只能来自内部时钟，高级控制定时器和通用定时器还可以选择外部时钟源或者直接来自其他定时器等待模式。我们可以通过 RCC 专用时钟配置寄存器 (RCC_DCKCFGR) 的 TIMPRE 位设置所有定时器的时钟频率，我们一般设置该位为默认值 0，即 TIMxCLK 为总线时钟的两倍，使得表 31-1 中可选的最大定时器时钟为 84MHz，即基本定时器的内部时钟 (CK_INT) 频率为 84MHz。

基本定时器只能使用内部时钟，当 TIM6 和 TIM7 控制寄存器 1(TIMx_CR1) 的 CEN 位置 1 时，启动基本定时器，并且预分频器的时钟来源就是 CK_INT。对于高级控制定时器和通用定时器的时

钟源可以来找控制器外部时钟、其他定时器等等模式，较为复杂，我们在相关章节会详细介绍。

31.3.2 控制器

定时器控制器控制实现定时器功能，控制定时器复位、使能、计数是其基础功能，基本定时器还专门用于 DAC 转换触发。

31.3.3 计数器

基本定时器计数过程主要涉及到三个寄存器内容，分别是计数器寄存器 (TIMx_CNT)、预分频器寄存器 (TIMx_PSC)、自动重载寄存器 (TIMx_ARR)，这三个寄存器都是 16 位有效数字，即可设置值为 0 至 65535。

首先我们来看图 31-1 中预分频器 PSC，它有一个输入时钟 CK_PSC 和一个输出时钟 CK_CNT。输入时钟 CK_PSC 来源于控制器部分，基本定时器只有内部时钟源所以 CK_PSC 实际等于 CK_INT，即 84MHz。在不同应用场所，经常需要不同的定时频率，通过设置预分频器 PSC 的值可以非常方便得到不同的 CK_CNT，实际计算为：fCK_CNT 等于 fCK_PSC/(PSC[15:0]+1)。

图 31-2 是将预分频器 PSC 的值从 1 改为 4 时计数器时钟变化过程。原来是 1 分频，CK_PSC 和 CK_CNT 频率相同。向 TIMx_PSC 寄存器写入新值时，并不会马上更新 CK_CNT 输出频率，而是等到更新事件发生时，把 TIMx_PSC 寄存器值更新到影子寄存器中，使其真正产生效果。更新为 4 分频后，在 CK_PSC 连续出现 4 个脉冲后 CK_CNT 才产生一个脉冲。

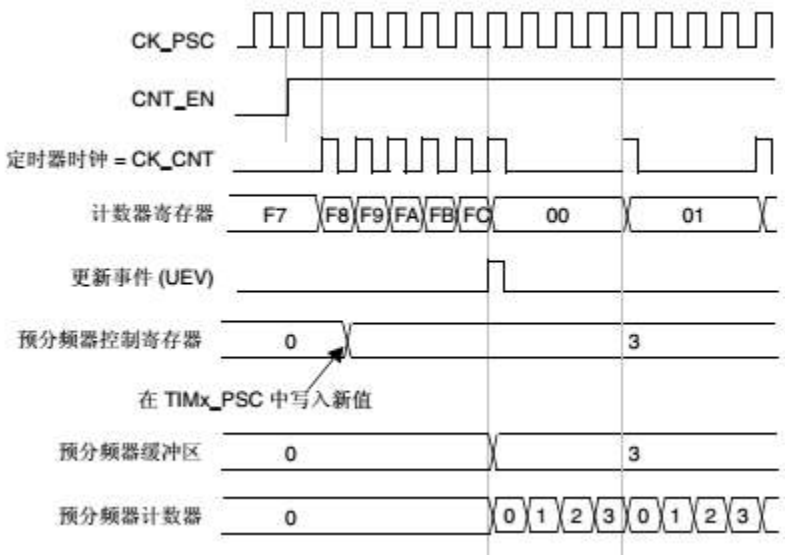


图 31-2 基本定时器时钟源分频

在定时器使能 (CEN 置 1) 时，计数器 COUNTER 根据 CK_CNT 频率向上计数，即每来一个 CK_CNT 脉冲，TIMx_CNT 值就加 1。当 TIMx_CNT 值与 TIMx_ARR 的设定值相等时就自动生成事件并 TIMx_CNT 自动清零，然后自动重新开始计数，如此重复以上过程。为此可见，我们只要设置 CK_PSC 和 TIMx_ARR 这两个寄存器的值就可以控制事件生成的时间，而我们一般的应用程序就是在事件生成的回调函数中运行的。在 TIMx_CNT 递增至与 TIMx_ARR 值相等，我们叫做为定时器上溢。

自动重载寄存器 TIMx_ARR 用来存放于计数器值比较的数值，如果两个数值相等就生成事件，将相关事件标志位置位，生成 DMA 和中断输出。TIMx_ARR 有影子寄存器，可以通过 TIMx_CR1 寄存器的 ARPE 位控制影子寄存器功能，如果 ARPE 位置 1，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，修改 TIMx_ARR 值马上有效。

31.3.4 定时器周期计算

经过上面分析，我们知道定时事件生成时间主要由 TIMx_PSC 和 TIMx_ARR 两个寄存器值决定，这个也就是定时器的周期。比如我们需要一个 1s 周期的定时器，具体这两个寄存器值该如何设置内。假设，我们先设置 TIMx_ARR 寄存器值为 9999，即当 TIMx_CNT 从 0 开始计算，刚好等于 9999 时生成事件，总共计数 10000 次，那么如果此时时钟源周期为 100us 即可得到刚好 1s 的定时周期。

接下来问题就是设置 TIMx_PSC 寄存器值使得 CK_CNT 输出为 100us 周期 (10000Hz) 的时钟。预分频器的输入时钟 CK_PSC 为 84MHz，所以设置预分频器值为 (8400-1) 即可满足。

31.4 定时器初始化结构体详解

HAL 库函数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，该结构体成员用于设置定时器基本工作参数，并由定时器基本初始化配置函数 TIM_TimeBaseInit 调用，这些设定参数将会设置定时器相应的寄存器，达到配置定时器工作环境的目的。这一章我们只介绍 TIM_TimeBaseInitTypeDef 结构体，其他结构体将在相关章节介绍。

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 stm32f4xx_hal_tim.h 文件中，初始化库函数定义在 stm32f4xx_hal_tim.c 文件中，编程时我们可以结合这两个文件内注释使用。

列表 1: 代码清单 31-1 定时器基本初始化结构体

```
typedef struct {
    uint16_t Prescaler;           // 预分频器
    uint16_t CounterMode;         // 计数模式
    uint32_t Period;              // 定时器周期
    uint16_t ClockDivision;       // 时钟分频
    uint8_t  RepetitionCounter;   // 重复计算器
} TIM_Base_InitTypeDef;
```

(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。

31.5 基本定时器定时实验

在 DAC 转换中几乎都用到基本定时器，使用有关基本定时器触发 DAC 转换内容在 DAC 章节讲解即可，这里就利用基本定时器实现简单的定时功能。

我们使用基本定时器循环定时 0.5s 并使能定时器中断，每到 0.5s 就在定时器中断服务函数翻转 RGB 彩灯，使得最终效果 RGB 彩灯暗 0.5s，亮 0.5s，如此循环。

31.5.1 硬件设计

基本定时器没有相关 GPIO，这里我们只用定时器的定时功能，无效其他外部引脚，至于 RGB 彩灯硬件可参考 GPIO 章节。

31.5.1.1 软件分析

31.5.1.1.1 宏定义

列表 2: 代码清单 31-2 宏定义

```
#define BASIC_TIM                   TIM6
#define BASIC_TIM_CLK_ENABLE()      __TIM6_CLK_ENABLE()
```

(下页继续)

(续上页)

```
#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。

31.5.1.1.2 NCIV 配置

列表 3: 代码清单 31-3 NVIC 配置

```
/**
* @brief  基本定时器 TIMx,x[6,7] 中断优先级配置
* @param  无
* @retval 无
*/
static void TIMx_NVIC_Configuration(void)
{
    //设置抢占优先级，子优先级
    HAL_NVIC_SetPriority(BASIC_TIM_IRQn, 0, 3);
    // 设置中断来源
    HAL_NVIC_EnableIRQ(BASIC_TIM_IRQn);
}
```

实验用到定时器更新中断，需要配置 NVIC，实验只有一个中断，对 NVIC 配置没什么具体要求。

31.5.1.1.3 基本定时器模式配置

列表 4: 代码清单 31-4 基本定时器模式配置

```
static void TIM_Mode_Config(void)
{
    // 开启 TIMx_CLK,x[6,7]
```

(下页继续)

(续上页)

```
    BASIC_TIM_CLK_ENABLE();

    TIM_TimeBaseStructure.Instance = BASIC_TIM;
    /* 累计 TIM_Period 个后产生一个更新或者中断 */
    //当定时器从 0 计数到 4999，即为 5000 次，为一个定时周期
    TIM_TimeBaseStructure.Init.Period = 5000-1;
    //定时器时钟源 TIMxCLK = 2 * PCLK1
    //        PCLK1 = HCLK / 4
    //        => TIMxCLK=HCLK/2=SystemCoreClock/2=84MHz
    // 设定定时器频率为 =TIMxCLK/(TIM_Prescaler+1)=10000Hz
    TIM_TimeBaseStructure.Init.Prescaler = 8400-1;

    // 初始化定时器 TIMx, x[2,3,4,5]
    HAL_TIM_Base_Init(&TIM_TimeBaseStructure);

    // 开启定时器更新中断
    HAL_TIM_Base_Start_IT(&TIM_TimeBaseStructure);
}
```

使用定时器之前都必须开启定时器时钟，基本定时器属于 APB1 总线外设。

接下来设置定时器周期数为 4999，即计数 5000 次生成事件。设置定时器预分频器为 (8400-1)，基本定时器使能内部时钟，频率为 84MHz，经过预分频器后得到 10KHz 的频率。然后就是调用 TIM_ HAL_TIM_Base_Init 函数完成定时器配置。

最后使用 HAL_TIM_Base_Start_IT 函数开启定时器和更新中断。

31.5.1.1.4 定时器中断服务函数

列表 5: 代码清单 31-5 定时器中断服务函数

```
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。

31.5.1.1.5 主函数

列表 6: 代码清单 31-6 主函数

```
int main(void)
{
    /* 初始化系统时钟为 168MHz */
    SystemClock_Config();
    /* 初始化 LED */
    LED_GPIO_Config();
    /* 初始化基本定时器定时，1s 产生一次中断 */
    TIMx_Configuration();
```

(下页继续)

(续上页)

```
    while (1) {
    }
}
```

实验中先初始化系统时钟，用到 RGB 彩灯，需要对其初始化配置。LED_GPIO_Config 函数是定义在 bsp_led.c 文件的完成 RGB 彩灯 GPIO 初始化配置的程序。

TIMx_Configuration 函数是定义在 bsp_basic_tim.c 文件的一个函数，它只是简单的先后调用 TIMx_NVIC_Configuration 和 TIM_Mode_Config 两个函数完成 NVIC 配置和基本定时器模式配置。

31.5.1.2 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。开始 RGB 彩灯是暗的，等一会 RGB 彩灯变为红色，再等一会又暗了，如此反复。如果我们使用表钟与 RGB 彩灯闪烁对比，可以发现它是每 0.5s 改变一次 RGB 彩灯状态的。

第 32 章　TIM—高级定时器

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。学习本章时，配合《STM32F4xx 参考手册》高级定时器章节一起阅读，效果会更佳，特别是涉及到寄存器说明的部分。特别说明，本书内容是以 STM32F40xx 系列控制器资源讲解。

上一章我们讲解了基本定时器功能，基本定时器功能简单，理解起来也容易。高级控制定时器包含了通用定时器的功能，再加上已经有了基本定时器基础的基础，如果再把通用定时器单独拿出来讲那内容有很多重复，实际效果不是很好，所以通用定时器不作为独立章节讲解，可以在理解了高级定时器后参考《STM32F4xx 中文参考手册》通用定时器章节内容理解即可。

32.1 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。

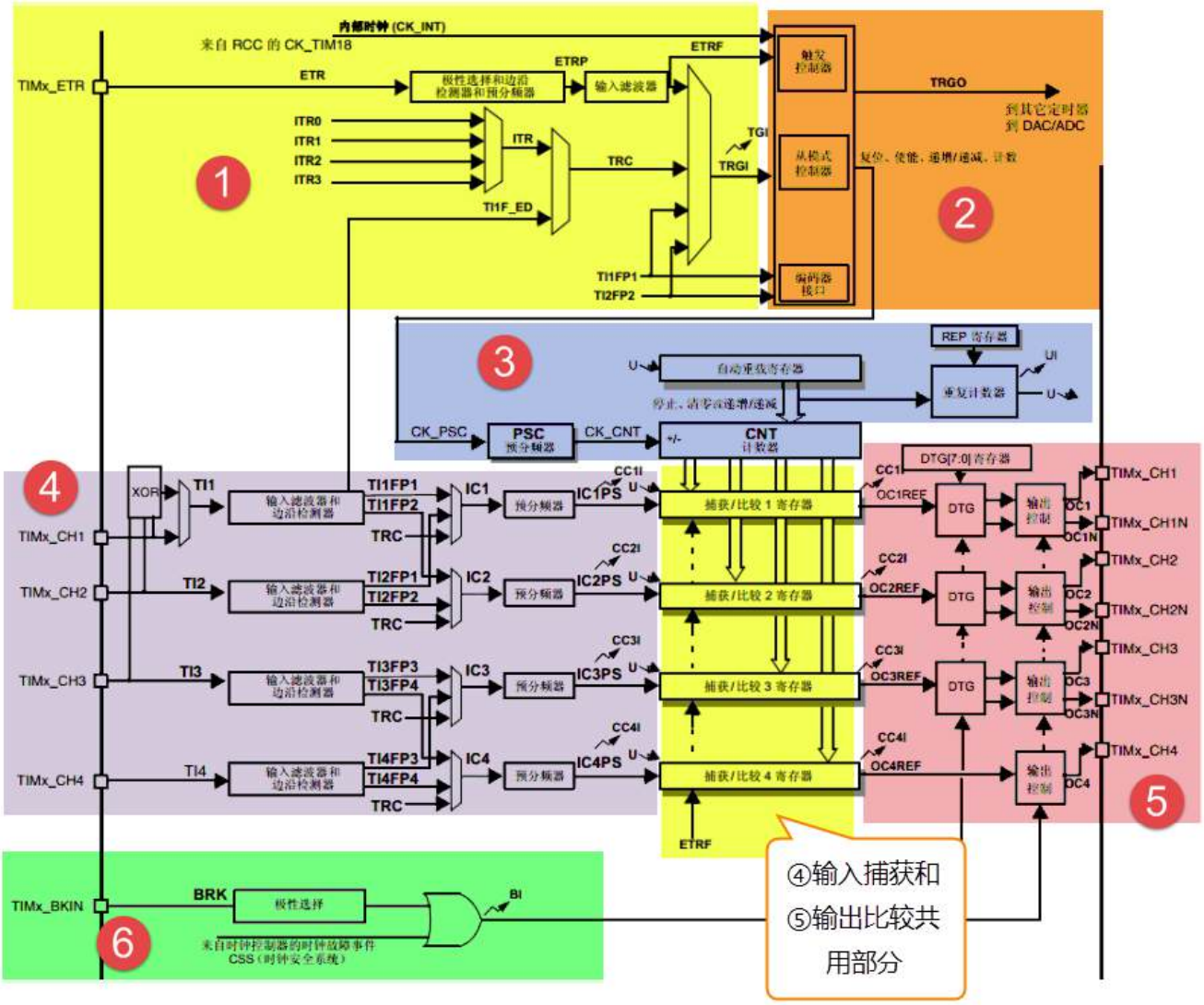
STM32F407ZGT6 的高级/通用定时器的 IO 分配具体见表 32-1。配套开发板因为 IO 资源紧缺，定时器的 IO 很多已经复用它途，故下表中的 IO 只有部分可用于定时器的实验。

表 32-1 高级控制和通用定时器通道引脚分布

	高级控制		通用定时器									
	TIM1	TIM8	TIM2	TIM5	TIM3	TIM4	TIM9	TIM10	TIM11	TIM12	TIM13	TIM14
CH1	PA8/PE9	PC6	PA0/PA5	PA0	PA6/PC6/PB4	PD12/PB6	PE5/PA2	PF6/PB8	PF7/PB9	PB14	PF8/PA6	PF9/PA7
CH1N	PA7/PE8/PB13	PA5/PA7										
CH2	PE11/PA9	PC7	PA1/PB3	PA1	PA7/PC7/PB5	PD13/PB7	PE6/PA3			PB15		
CH2N	PB0/PE10/PB14	PB0/PB14										
CH3	PE13/PA10	PC8	PA2/PB10	PA2	PB0/PC8	PD14/PB8						
CH3N	PB1/PE12/PB15	PB1/PB15										
CH4	PE14/PA11	PC9	PA3/PB11	PA3	PB1/PC9	PD15/PB9						
ETR	PE7/PA12	PA0/PI3	PA0/PA5/PA15		PD2	PE0						
BKIN	PA6/PE15/PB12	PA6/PI4										

32.2 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。



32.2.1 □ 时钟源

高级控制定时器有四个时钟源可选：

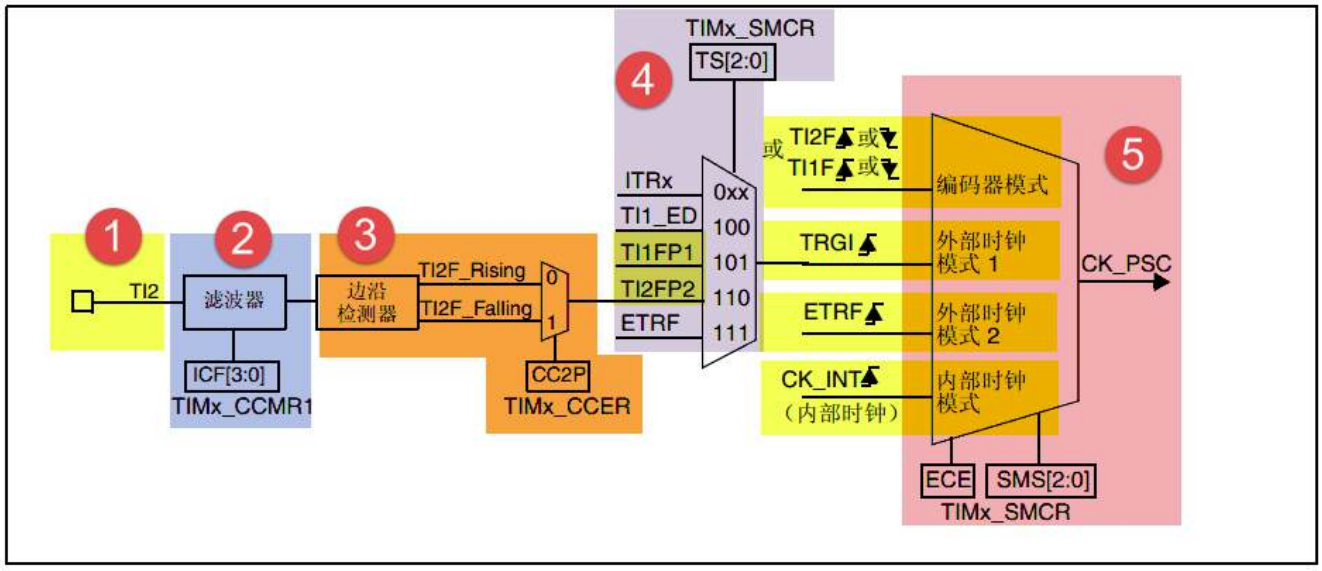
- 内部时钟源 CK_INT
- 外部时钟模式 1：外部输入引脚 TIx（x=1,2,3,4）
- 外部时钟模式 2：外部触发输入 ETR

- 内部触发输入 (ITRx)

32.2.1.1 内部时钟源 (CK_INT)

内部时钟 CK_INT 即来自于芯片内部，等于 168M，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。

32.2.1.2 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。

32.2.1.3 外部时钟模式 2

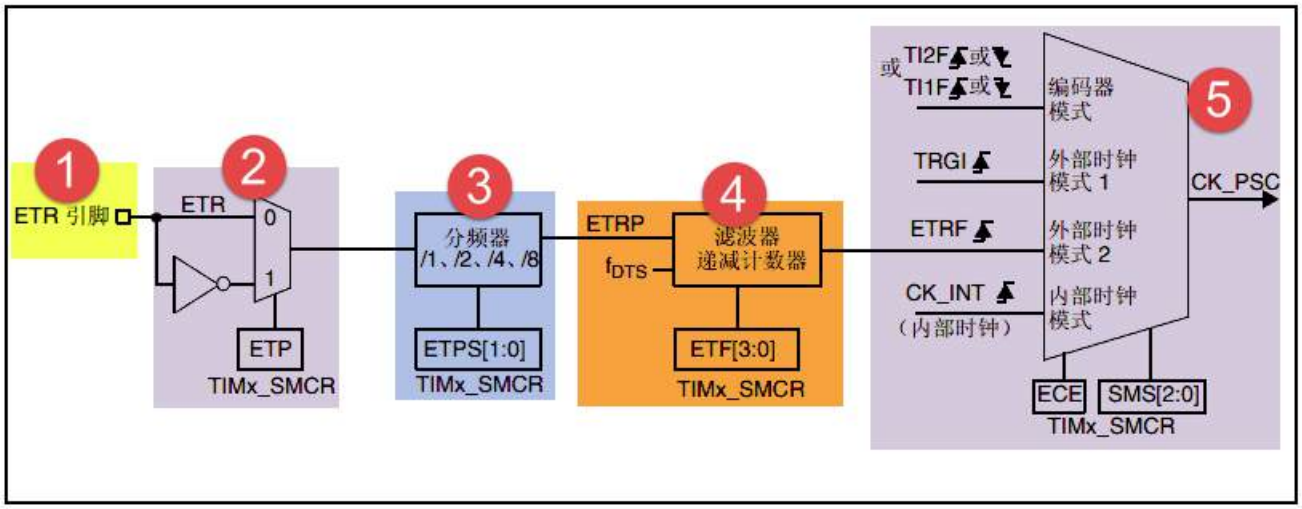


图 31-3 外部时钟模式 2 框图

□：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

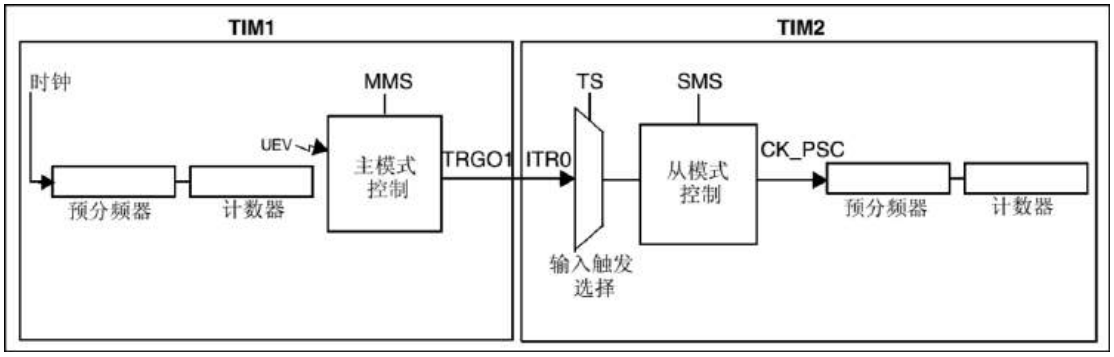
经过滤波器滤波的信号连接到 ETRF 引脚后，触发信号成为外部时钟模式 2 的输入，最终等于 CK_PSC，然后驱动计数器 CNT 计数。具体的配置 TIMx_SMCR 的位 ECE 为 1 即可选择外部时钟模式 2。

□：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。

32.2.1.4 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。高级控制定时器和部分通用定时器 (TIM2 至 TIM5) 可以设置为主模式或从模式，TIM9 和 TIM10 可设置为从模式。图 *32_21* 为主模式定时器 (TIM1) 为从模式定时器 (TIM2) 提供时钟，即 TIM1 用作 TIM2 的预分频器。



32.2.2 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。

32.2.3 时基单元

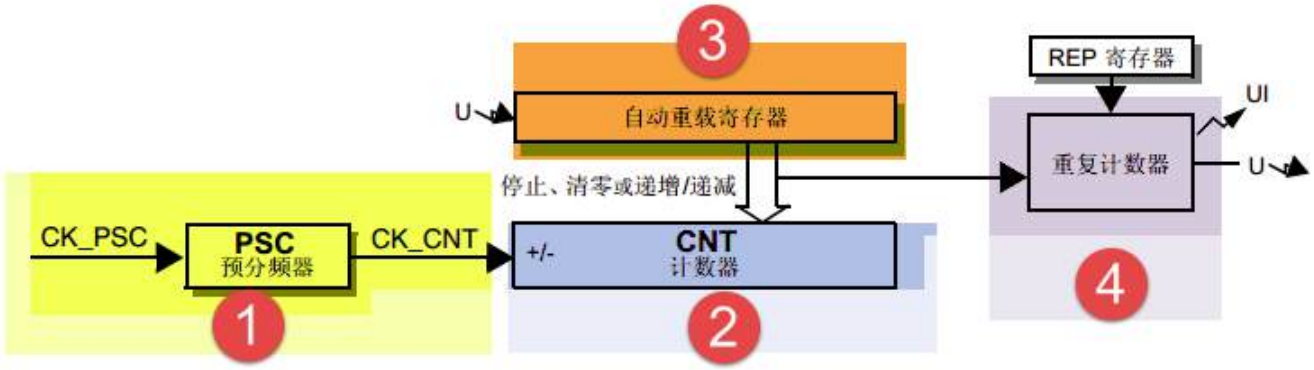


图 31-4 高级定时器时基单元

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。

32.2.3.1 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。

32.2.3.2 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。

32.2.3.3 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。

32.2.3.4 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。

32.2.4 □ 输入捕获

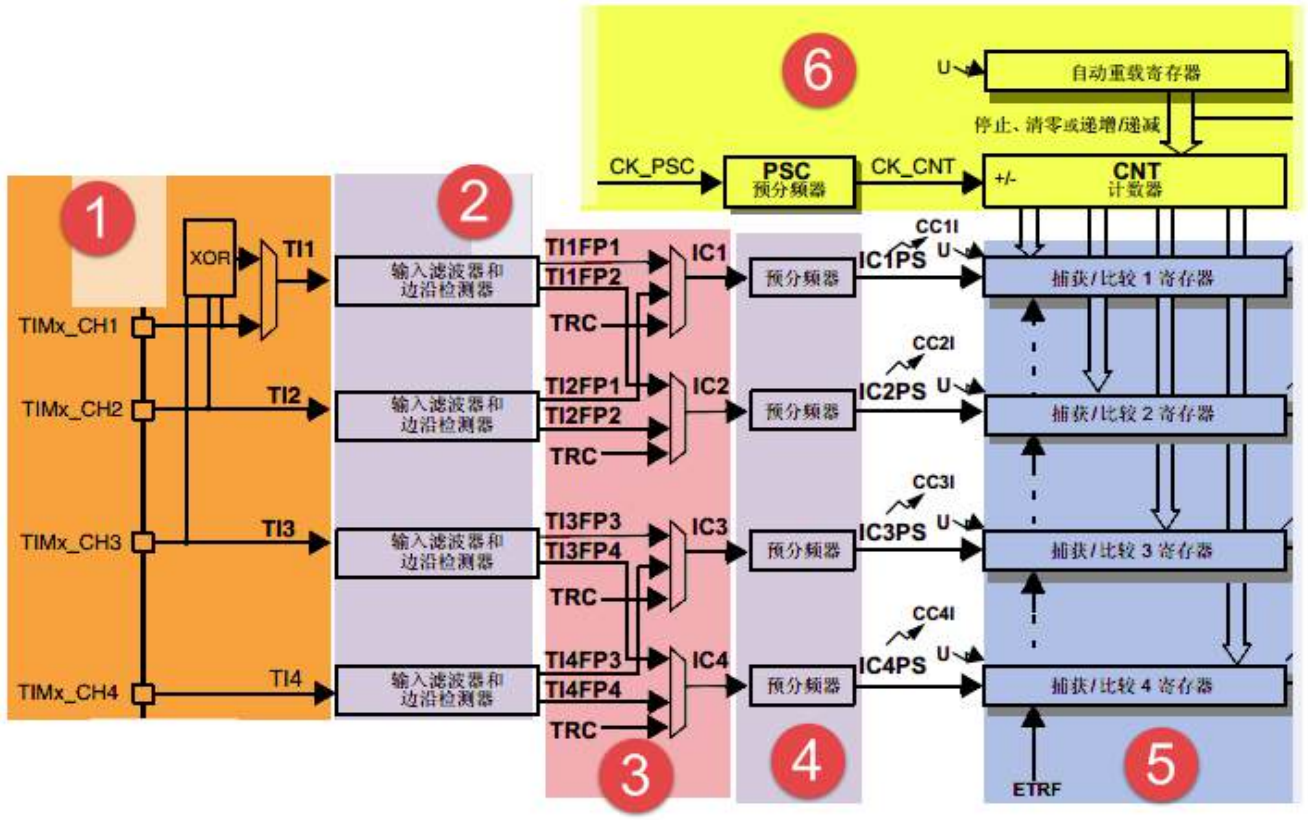


图 31-5 输入捕获功能框图

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。

32.2.4.1 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。

32.2.4.2 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。

32.2.4.3 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。

32.2.4.4 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。

32.2.4.5 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。

32.2.5 □ 输出比较

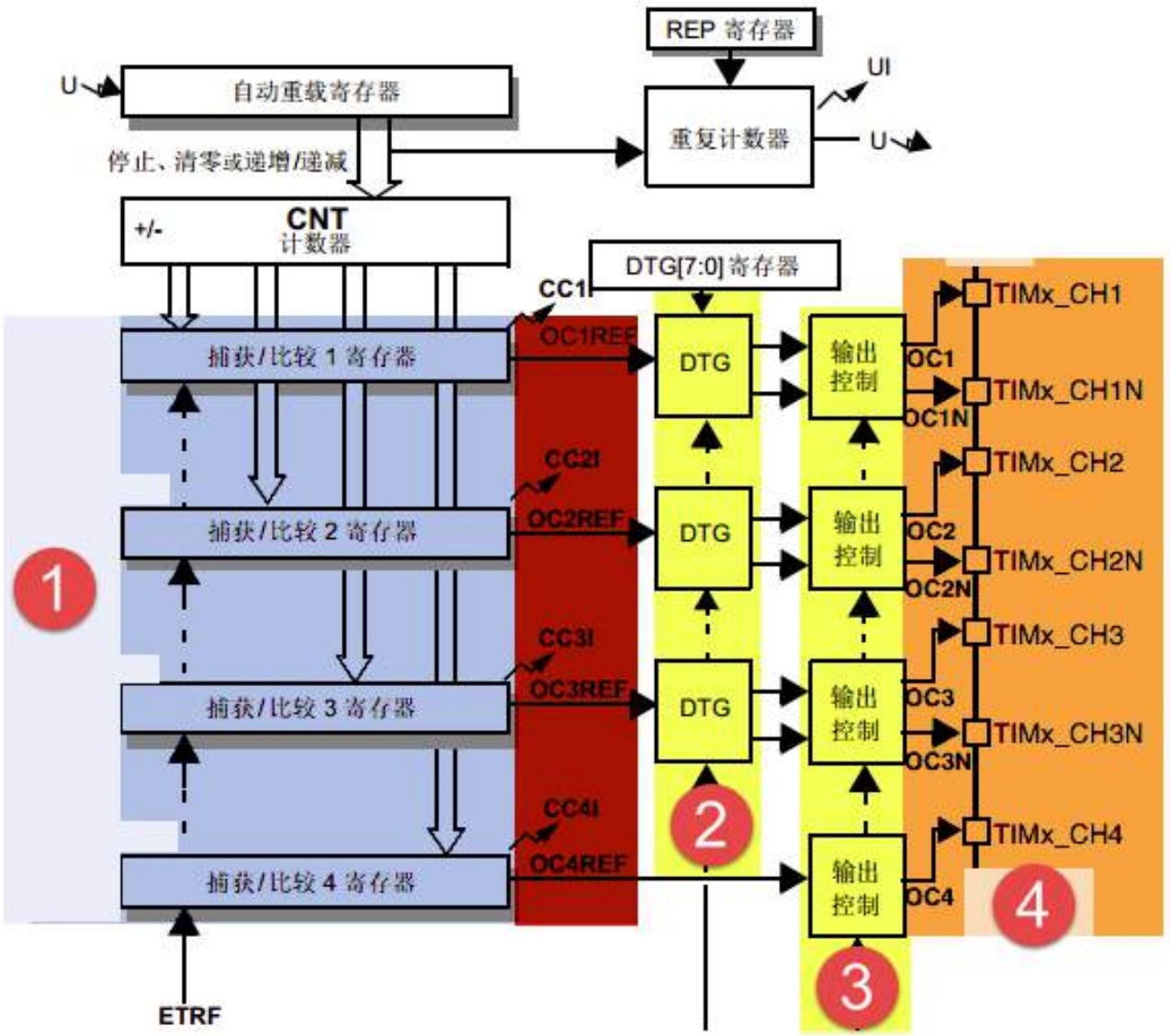


图 31-6 输出比较功能框图

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。

32.2.5.1 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。

32.2.5.2 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。

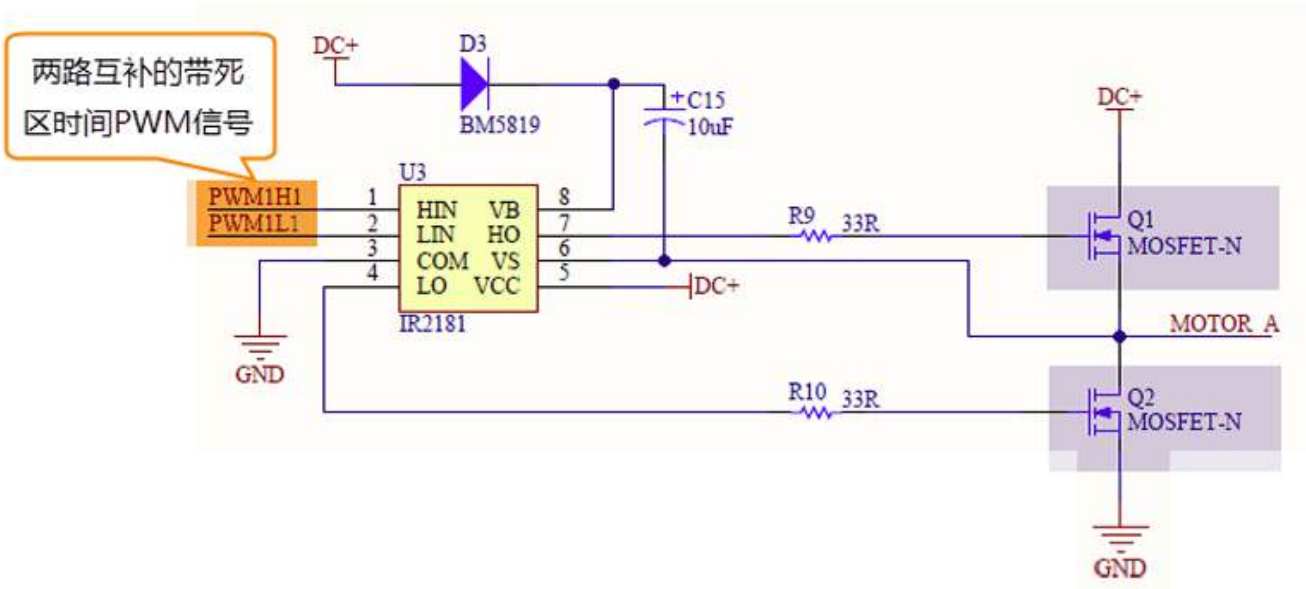


图 31-7 半桥驱动电路

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。

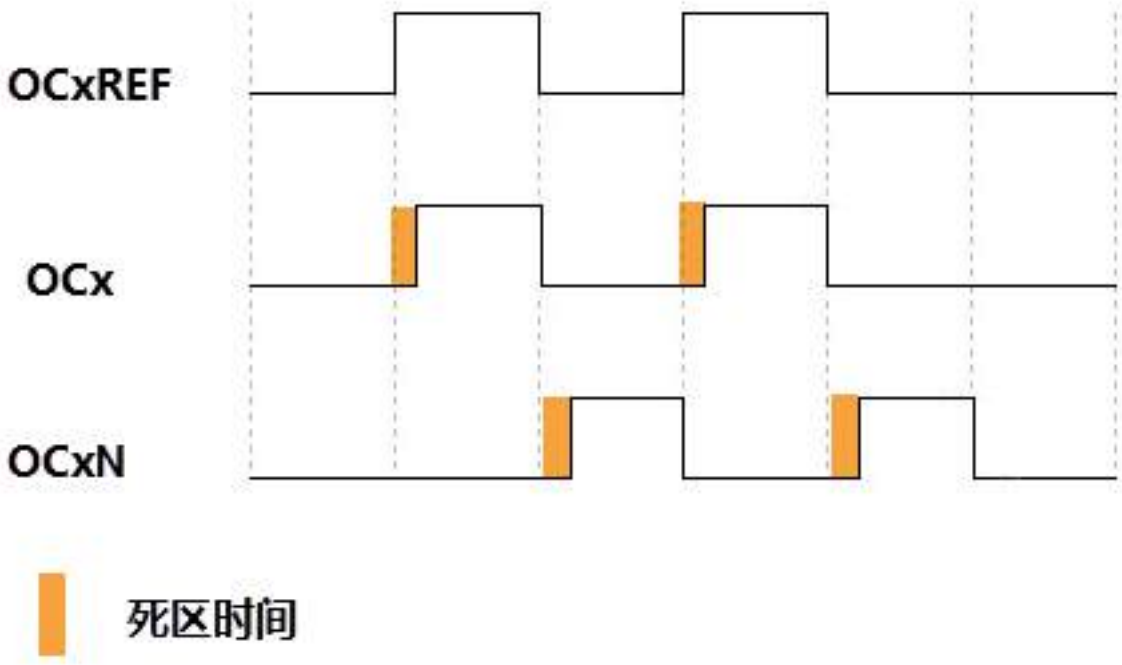


图 31-8 带死区插入的互补输出

32.2.5.3 □ 输出控制

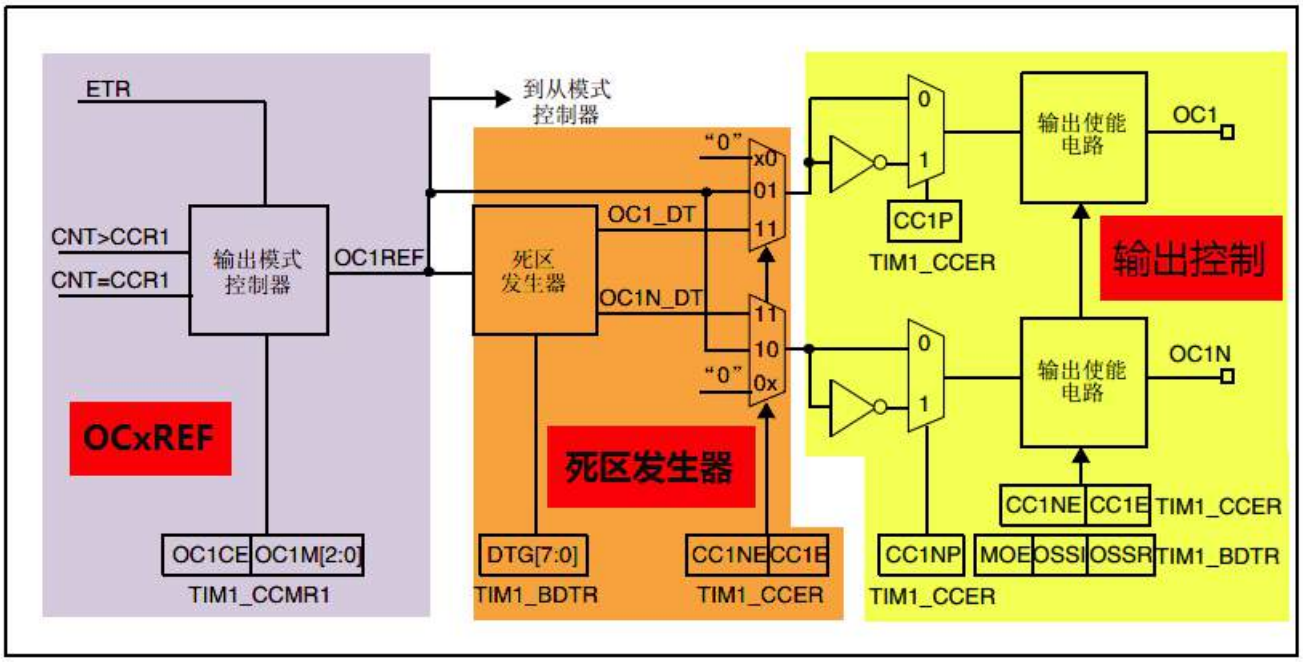


图 31-9 输出比较（通道 1~3）的输出控制框图

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。

32.2.5.4 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。

32.2.6 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。

32.3 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。

32.3.1 测量脉宽或者频率

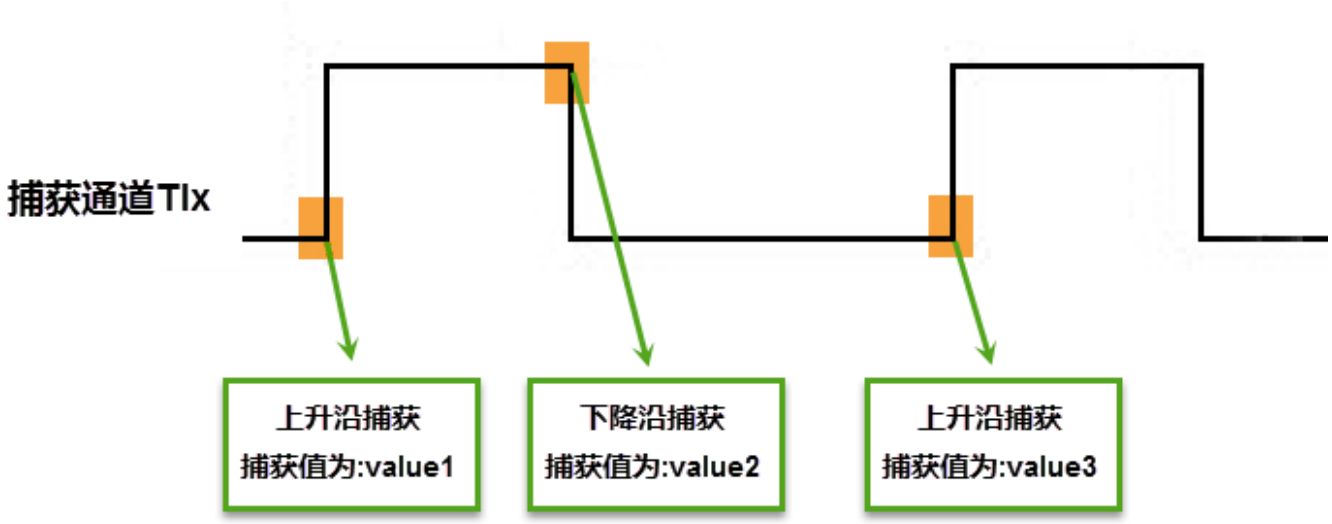


图 31-10 脉宽/频率测量示意图

32.3.1.1 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）。

32.3.1.2 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。

32.3.2 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。

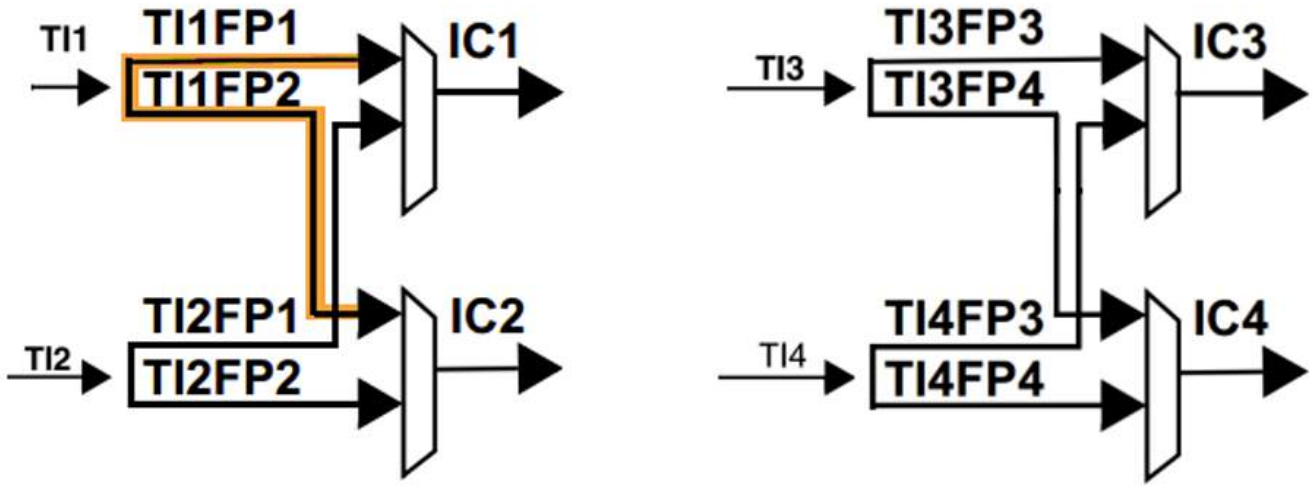


图 31-11 输入通道和捕获通道的关系映射图

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。

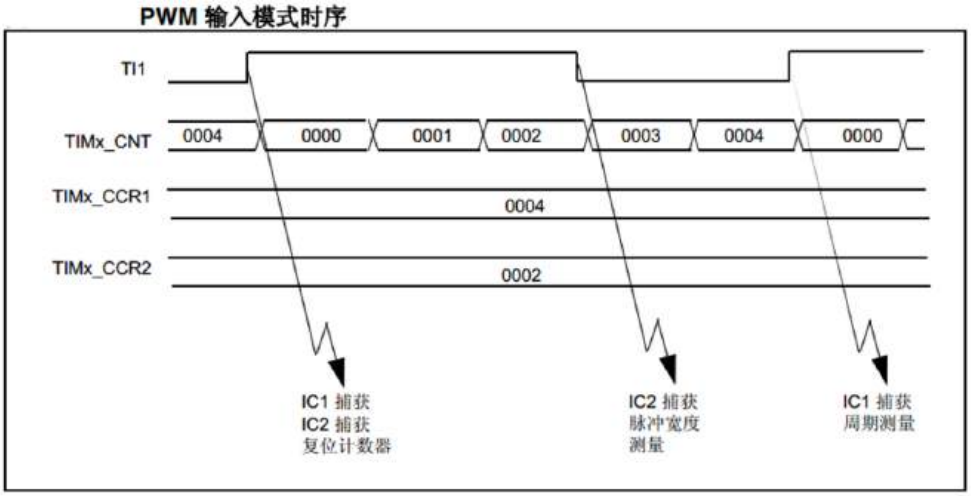


图 31-12 PWM 输入模式时序

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。

32.4 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。

32.4.1 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。

32.4.1.1 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

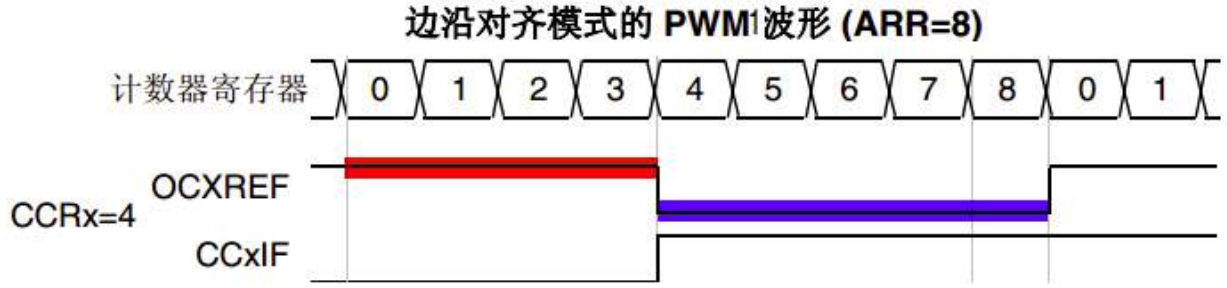


图 31-13 PWM1 模式的边沿对齐波形

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。

32.4.1.2 PWM 中心对齐模式

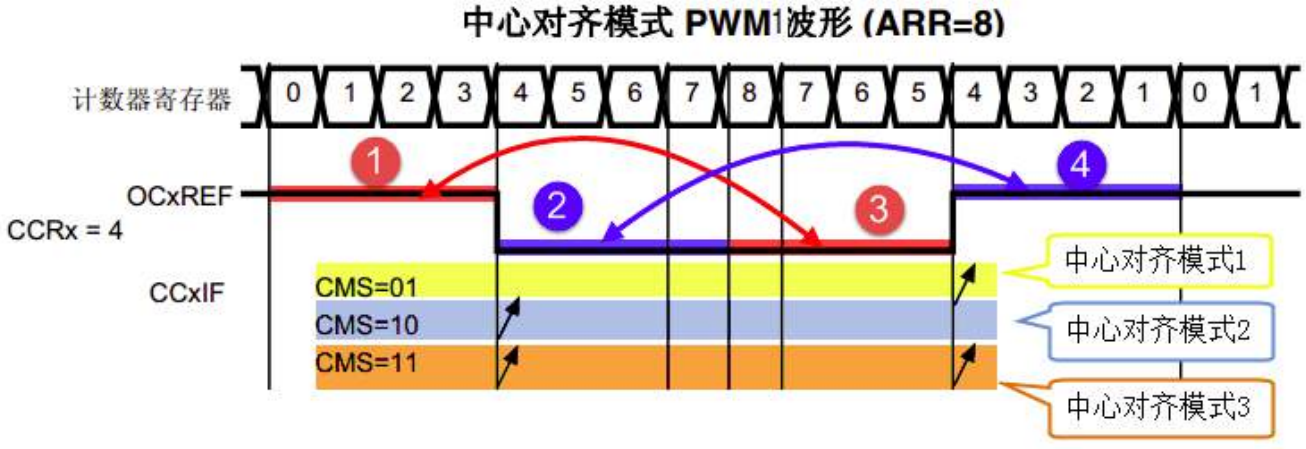


图 31-14 PWM1 模式的中心对齐波形

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。

32.5 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。

32.5.1 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 {
    uint16_t Prescaler;           // 预分频器
    uint16_t CounterMode;         // 计数模式
    uint32_t Period;              // 定时器周期
    uint16_t ClockDivision;       // 时钟分频
    uint8_t  RepetitionCounter;   // 重复计算器
} Time_Base_InitTypeDef;
```

(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。

32.5.2 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。

32.5.3 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。

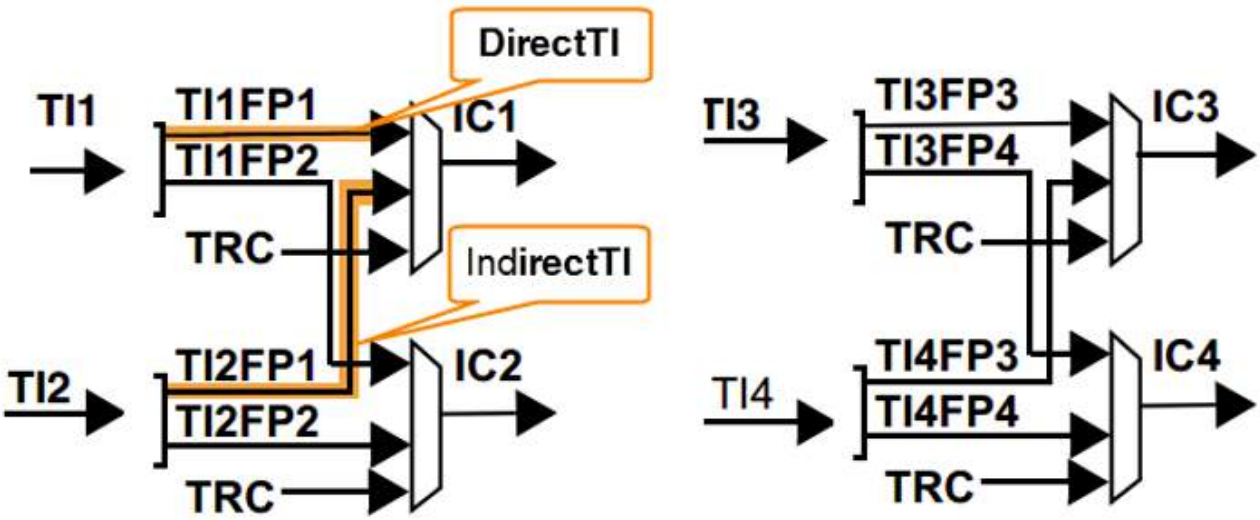


图 31-15 输入通道与捕获通道 IC 的映射图

(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。

32.5.4 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。

32.6 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。

32.6.1 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。

32.6.2 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。

32.6.2.1 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

32.6.2.2 软件分析

32.6.2.2.1 宏定义

列表 5: 代码清单 31-5 宏定义

```
/* 定时器 */
#define ADVANCE_TIM                         TIM8
#define ADVANCE_TIM_CLK_ENABLE()            __TIM8_CLK_ENABLE()

/* TIM8 通道 1 输出引脚 */
#define ADVANCE_OCPWM_PIN                   GPIO_PIN_6
#define ADVANCE_OCPWM_GPIO_PORT             GPIOC
#define ADVANCE_OCPWM_GPIO_CLK_ENABLE()     __GPIOC_CLK_ENABLE()
#define ADVANCE_OCPWM_AF                GPIO_AF3_TIM8

/* TIM8 通道 1 互补输出引脚 */
```

(下页继续)

(续上页)

```
#define ADVANCE_OCNPWM_PIN                  GPIO_PIN_5
#define ADVANCE_OCNPWM_GPIO_PORT            GPIOA
#define ADVANCE_OCNPWM_GPIO_CLK_ENABLE()    __GPIOA_CLK_ENABLE()
#define ADVANCE_OCNPWM_AF                GPIO_AF3_TIM8

/* TIM8 断路输入引脚 */
#define ADVANCE_BKIN_PIN                    GPIO_PIN_6
#define ADVANCE_BKIN_GPIO_PORT              GPIOA
#define ADVANCE_BKIN_GPIO_CLK_ENABLE()      __GPIOA_CLK_ENABLE()
#define ADVANCE_BKIN_AF                   GPIO_AF3_TIM8
```

使用宏定义非常方便程序升级、移植。如果使用不同的定时器 IO，修改这些宏即可。

32.6.2.2.2 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

```
static void TIMx_GPIO_Config(void)
{
    /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启定时器相关的 GPIO 外设时钟 */
    ADVANCE_OCPWM_GPIO_CLK_ENABLE();
    ADVANCE_OCNPWM_GPIO_CLK_ENABLE();
    ADVANCE_BKIN_GPIO_CLK_ENABLE();

    /* 定时器功能引脚初始化 */
    GPIO_InitStructure.Pin = ADVANCE_OCPWM_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    GPIO_InitStructure.Alternate = ADVANCE_OCPWM_AF;
    HAL_GPIO_Init(ADVANCE_OCPWM_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = ADVANCE_OCNPWM_PIN;
    GPIO_InitStructure.Alternate = ADVANCE_OCNPWM_AF;
    HAL_GPIO_Init(ADVANCE_OCNPWM_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = ADVANCE_BKIN_PIN;
    GPIO_InitStructure.Alternate = ADVANCE_BKIN_AF;
    HAL_GPIO_Init(ADVANCE_BKIN_GPIO_PORT, &GPIO_InitStructure);
}
```

定时器通道引脚使用之前必须设定相关参数，这选择复用功能，并指定到对应的定时器。使用 GPIO 之前都必须开启相应端口时钟。

32.6.2.2.3 定时器模式配置

列表 7: 代码清单 31-7 定时器模式配置

```
static void TIM_Mode_Config(void)
{
    TIM_BreakDeadTimeConfigTypeDef TIM_BDTRInitStructure;
    // 开启 TIMx_CLK,x[1,8]
    ADVANCE_TIM_CLK_ENABLE();
    /* 定义定时器的句柄即确定定时器寄存器的基地址 */
    TIM_TimeBaseStructure.Instance = ADVANCE_TIM;
    /* 累计 TIM_Period 个后产生一个更新或者中断 */
    //当定时器从 0 计数到 999，即为 1000 次，为一个定时周期
    TIM_TimeBaseStructure.Init.Period = 1000-1;
    // 高级控制定时器时钟源 TIMxCLK = HCLK=168MHz
```

(下页继续)

(续上页)

```
    // 设定定时器频率为 =TIMxCLK/(TIM_Prescaler+1)=1MHz
    TIM_TimeBaseStructure.Init.Prescaler = 168-1;
    // 采样时钟分频
    TIM_TimeBaseStructure.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    // 计数方式
    TIM_TimeBaseStructure.Init.CounterMode=TIM_COUNTERMODE_UP;
    // 重复计数器
    TIM_TimeBaseStructure.Init.RepetitionCounter=0;
    // 初始化定时器 TIMx, x[1,8]
    HAL_TIM_PWM_Init(&TIM_TimeBaseStructure);

    /*PWM 模式配置 */
    //配置为 PWM 模式 1
    TIM_OCInitStructure.OCMode = TIM_OCMODE_PWM1;
    TIM_OCInitStructure.Pulse = ChannelPulse;
    TIM_OCInitStructure.OCPolarity = TIM_OCPOLARITY_HIGH;
    TIM_OCInitStructure.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    TIM_OCInitStructure.OCIdleState = TIM_OCIDLESTATE_SET;
    TIM_OCInitStructure.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    //初始化通道 1 输出 PWM
    HAL_TIM_PWM_ConfigChannel(&TIM_TimeBaseStructure,&TIM_OCInitStructure,
↪TIM_CHANNEL_1);

    /* 自动输出使能，断路、死区时间和锁定配置 */
    TIM_BDTRInitStructure.OffStateRunMode = TIM_OSSR_ENABLE;
    TIM_BDTRInitStructure.OffStateIDLEMode = TIM_OSSI_ENABLE;
    TIM_BDTRInitStructure.LockLevel = TIM_LOCKLEVEL_1;
    TIM_BDTRInitStructure.DeadTime = 11;
    TIM_BDTRInitStructure.BreakState = TIM_BREAK_ENABLE;
    TIM_BDTRInitStructure.BreakPolarity = TIM_BREAKPOLARITY_LOW;
    TIM_BDTRInitStructure.AutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
    HAL_TIMEx_ConfigBreakDeadTime(&TIM_TimeBaseStructure,
```

(下页继续)

(续上页)

```
                                                &TIM_BDTRInitStructure);
    /* 定时器通道 1 输出 PWM */
    HAL_TIM_PWM_Start(&TIM_TimeBaseStructure,TIM_CHANNEL_1);
    /* 定时器通道 1 互补输出 PWM */
    HAL_TIMEx_PWMN_Start(&TIM_TimeBaseStructure,TIM_CHANNEL_1);
}
```

首先定义三个定时器初始化结构体，定时器模式配置函数主要就是对这三个结构体的成员进行初始化，然后通过相应的初始化函数把这些参数写入定时器的寄存器中。有关结构体的成员介绍请参考定时器初始化结构体详解小节。

不同的定时器可能对应不同的 APB 总线，在使能定时器时钟是必须特别注意。高级控制定时器属于 APB2，定时器内部时钟是 168MHz。

在时基结构体中我们设置定时器周期参数为 1000，频率为 1MHz，使用向上计数方式。因为我们使用的是内部时钟，所以外部时钟采样分频成员不需要设置，重复计数器我们没用到，也不需要设置。

在输出比较结构体中，设置输出模式为 PWM1 模式，主通道和互补通道输出高电平有效，设置脉宽为 ChannelPulse，ChannelPulse 是我们定义的一个无符号 16 位整形的全局变量，用来指定占空比大小，实际上脉宽就是设定比较寄存器 CCR 的值，用于跟计数器 CNT 的值比较。

断路和死区结构体中，使能断路功能，设定断路信号的有效极性，设定死区时间。

最后使用 HAL_TIM_PWM_Start 函数和 HAL_TIMEx_PWMN_Start 函数让计数器开始计数和通道输出。

32.6.2.2.4 主函数

列表 8: 代码清单 31-8 main 函数

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

(下页继续)

(续上页)

```
    /* 初始化系统时钟为 168MHz */
    SystemClock_Config();
    /* 初始化按键 GPIO */
    Key_GPIO_Config();
    /* 初始化基本定时器定时，1s 产生一次中断 */
    TIMx_Configuration();

    while (1) {
        /* 扫描 KEY1 */
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON  ) {
            /* 增大占空比 */
            if (ChannelPulse<950)
                ChannelPulse+=50;
            else
                ChannelPulse=1000;
            __HAL_TIM_SetCompare(&TIM_TimeBaseStructure,TIM_CHANNEL_1,
↪ChannelPulse);
        }
        /* 扫描 KEY2 */
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON  ) {
            /* 减小占空比 */
            if (ChannelPulse>=50)
                ChannelPulse-=50;
            else
                ChannelPulse=0;
            __HAL_TIM_SetCompare(&TIM_TimeBaseStructure,TIM_CHANNEL_1,
↪ChannelPulse);
        }
    }
}
```

首先，调用初始化系统时钟，Key_GPIO_Config 函数完成按键引脚初始化配置，该函数定义在

bsp_key.c 文件中。

接下来，调用 TIMx_Configuration 函数完成定时器参数配置，包括定时器复用引脚配置和定时器模式配置，该函数定义在 bsp_advance_tim.c 文件中它实际上只是简单的调用 TIMx_GPIO_Config 函数和 TIM_Mode_Config 函数。运行完该函数后通道引脚就已经有 PWM 波形输出，通过示波器可直观观察到。

最后，在无限循环函数中检测按键状态，如果是 KEY1 被按下，就增加 ChannelPulse 变量值，并调用 TIM_SetCompare1 函数完成增加占空比设置；如果是 KEY2 被按下，就减小 ChannelPulse 变量值，并调用 TIM_SetCompare1 函数完成减少占空比设置。TIM_SetCompare1 函数实际是设定 TIMx_CCR1 寄存器值。

32.6.2.3 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，并把断路输入引脚拉高。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 PWM 波形，参考图 *31_16*。此时，按下开发板上 KEY1 或 KEY2 可改变波形的占空比。

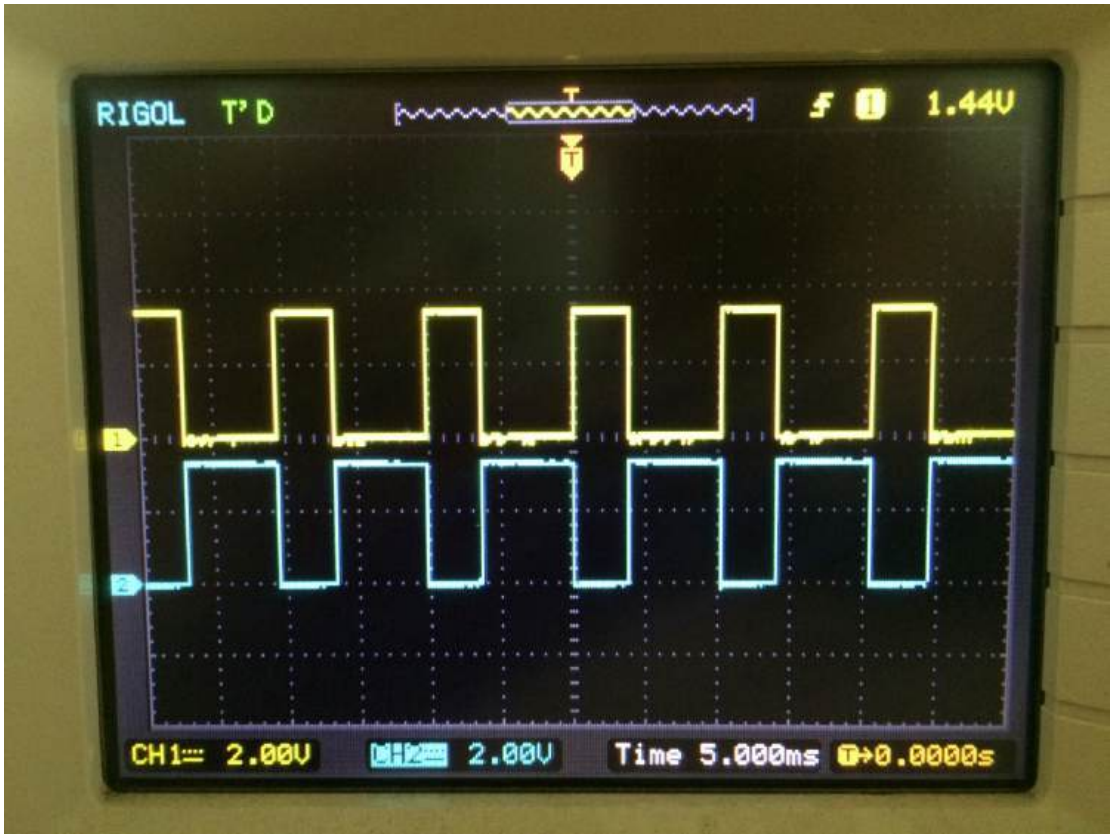


图 31-16 互补波形输出示波器图

32.7 PWM 输入捕获实验

实验中，我们用通用定时器产生已知频率和占空比的 PWM 信号，然后用高级定时器的 PWM 输入模式来测量这个已知的 PWM 信号的频率和占空比，通过两者的对比即可知道测量是否准确。

32.7.1 硬件设计

实验中用到两个引脚，一个是通用定时器通道用于波形输出，另一个是高级控制定时器通道用于输入捕获，实验中直接使用一根杜邦线短接即可。

32.7.2 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_advance_tim.c 和 bsp_advance_tim.h 文件用来存定时器驱动程序及相关宏定义。

32.7.2.1 编程要点

(1) 通用定时器产生 PWM 配置

(2) 高级定时器 PWM 输入配置

(3) 计算测量的频率和占空比，并打印出来比较

32.7.2.2 软件分析

32.7.2.2.1 宏定义

列表 9: 代码清单 31-9 宏定义

```
/* 通用定时器 */
#define GENERAL_TIM                         TIM2
#define GENERAL_TIM_CLK_ENABLE()            __TIM2_CLK_ENABLE()

/* 通用定时器 PWM 输出 */
/* PWM 输出引脚 */
#define GENERAL_OCPWM_PIN                   GPIO_PIN_5
#define GENERAL_OCPWM_GPIO_PORT             GPIOA
#define GENERAL_OCPWM_GPIO_CLK_ENABLE()     __GPIOA_CLK_ENABLE()
#define GENERAL_OCPWM_AF                GPIO_AF1_TIM2

/* 高级控制定时器 */
#define ADVANCE_TIM                         TIM8
```

(下页继续)

(续上页)

```
#define ADVANCE_TIM_CLK_ENABLE()          __TIM8_CLK_ENABLE()

/* 捕获/比较中断 */
#define ADVANCE_TIM_IRQn                  TIM8_CC_IRQn
#define ADVANCE_TIM_IRQHandler            TIM8_CC_IRQHandler
/* 高级控制定时器 PWM 输入捕获 */
/* PWM 输入捕获引脚 */
#define ADVANCE_ICPWM_PIN                 GPIO_PIN_6
#define ADVANCE_ICPWM_GPIO_PORT           GPIOC
#define ADVANCE_ICPWM_GPIO_CLK_ENABLE()   __GPIOC_CLK_ENABLE()
#define ADVANCE_ICPWM_AF                  GPIO_AF3_TIM8
#define ADVANCE_IC1PWM_CHANNEL            TIM_CHANNEL_1
#define ADVANCE_IC2PWM_CHANNEL            TIM_CHANNEL_2
```

使用宏定义非常方便程序升级、移植。如果使用不同的定时器 IO，修改这些宏即可。

32.7.2.2.2 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

```
static void TIMx_GPIO_Config(void)
{
    /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启定时器相关的 GPIO 外设时钟 */
    GENERAL_OCPWM_GPIO_CLK_ENABLE();
    ADVANCE_ICPWM_GPIO_CLK_ENABLE();

    /* 定时器功能引脚初始化 */
    /* 通用定时器 PWM 输出引脚 */
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin = GENERAL_OCPWM_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    GPIO_InitStructure.Alternate = GENERAL_OCPWM_AF;
    HAL_GPIO_Init(GENERAL_OCPWM_GPIO_PORT, &GPIO_InitStructure);

    /* 高级定时器输入捕获引脚 */
    GPIO_InitStructure.Pin = ADVANCE_ICPWM_PIN;
    GPIO_InitStructure.Alternate = ADVANCE_ICPWM_AF;
    HAL_GPIO_Init(ADVANCE_ICPWM_GPIO_PORT, &GPIO_InitStructure);
}
```

定时器通道引脚使用之前必须设定相关参数，这选择复用功能，并指定到对应的定时器。使用 GPIO 之前都必须开启相应端口时钟。

32.7.2.2.3 嵌套向量中断控制器组配置

列表 11: NVIC 配置

```
static void TIMx_NVIC_Configuration(void)
{
    //设置抢占优先级，子优先级
    HAL_NVIC_SetPriority(ADVANCE_TIM_IRQn, 0, 3);
    // 设置中断来源
    HAL_NVIC_EnableIRQ(ADVANCE_TIM_IRQn);
}
```

实验用到高级控制定时器捕获/比较中断，需要配置中断优先级，因为实验只用到一个中断，所以这里对优先级配置没具体要求，只要符合中断组参数要求即可。

32.7.2.2.4 通用定时器 PWM 输出

列表 12: 代码清单 31-11 定时器模式配置

```
static void TIM_PWMOUTPUT_Config(void)
{
    TIM_OC_InitTypeDef TIM_OCInitStructure;
    // 开启 TIMx_CLK,x[2,3,4,5,12,13,14]
    GENERAL_TIM_CLK_ENABLE();
    /* 定义定时器的句柄即确定定时器寄存器的基地址 */
    TIM_PWMOUTPUT_Handle.Instance = GENERAL_TIM;
    /* 累计 TIM_Period 个后产生一个更新或者中断 */
    //当定时器从 0 计数到 9999，即为 10000 次，为一个定时周期
    TIM_PWMOUTPUT_Handle.Init.Period = 10000-1;
    // 高级控制定时器时钟源 TIMxCLK = HCLK=84MHz
    // 设定定时器频率为 =TIMxCLK/(TIM_Prescaler+1)=100KHz
    TIM_PWMOUTPUT_Handle.Init.Prescaler = 84-1;
    // 采样时钟分频
    TIM_PWMOUTPUT_Handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
```

(下页继续)

(续上页)

```
    // 计数方式
    TIM_PWMOUTPUT_Handle.Init.CounterMode=TIM_COUNTERMODE_UP;
    // 重复计数器
    TIM_PWMOUTPUT_Handle.Init.RepetitionCounter=0;
    // 初始化定时器 TIMx, x[1,8]
    HAL_TIM_PWM_Init(&TIM_PWMOUTPUT_Handle);

    /*PWM 模式配置 */
    //配置为 PWM 模式 1
    TIM_OCInitStructure.OCMode = TIM_OCMODE_PWM1;
    TIM_OCInitStructure.Pulse = 5000;
    TIM_OCInitStructure.OCPolarity = TIM_OCPOLARITY_HIGH;
    TIM_OCInitStructure.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    TIM_OCInitStructure.OCIdleState = TIM_OCIDLESTATE_SET;
    TIM_OCInitStructure.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    //初始化通道 1 输出 PWM
    HAL_TIM_PWM_ConfigChannel(&TIM_PWMOUTPUT_Handle,
                              &TIM_OCInitStructure,TIM_CHANNEL_1);
    /* 定时器通道 1 输出 PWM */
    HAL_TIM_PWM_Start(&TIM_PWMOUTPUT_Handle,TIM_CHANNEL_1);

}
```

定时器 PWM 输出模式配置函数很简单，看代码注释即可。这里我们设置了 PWM 的频率为 100Hz，即周期为 10ms，占空比为：(Pulse+1)/(Period+1) = 50%。

32.7.2.2.5 高级控制定时 PWM 输入模式

列表 13: 代码清单 PWM 输入模式配置

```
static void TIM_PWMINPUT_Config(void)
{
    TIM_IC_InitTypeDef    TIM_ICInitStructure;
    TIM_SlaveConfigTypeDef  TIM_SlaveConfigStructure;
    // 开启 TIMx_CLK,x[1,8]
    ADVANCE_TIM_CLK_ENABLE();
    /* 定义定时器的句柄即确定定时器寄存器的基地址 */
    TIM_PWMINPUT_Handle.Instance = ADVANCE_TIM;
    TIM_PWMINPUT_Handle.Init.Period = 0xFFFF;
    // 高级控制定时器时钟源 TIMxCLK = HCLK=168MHz
    // 设定定时器频率为 =TIMxCLK/(TIM_Prescaler+1)=1MHz
    TIM_PWMINPUT_Handle.Init.Prescaler = 168-1;
    // 采样时钟分频
    TIM_PWMINPUT_Handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    // 计数方式
    TIM_PWMINPUT_Handle.Init.CounterMode=TIM_COUNTERMODE_UP;
    // 初始化定时器 TIMx, x[1,8]
    HAL_TIM_IC_Init(&TIM_PWMINPUT_Handle);

    /* IC1 捕获：上升沿触发 TI1FP1 */
    TIM_ICInitStructure.ICPolarity = TIM_ICPOLARITY_RISING;
    TIM_ICInitStructure.ICSelection = TIM_ICSELECTION_DIRECTTI;
    TIM_ICInitStructure.ICPrescaler = TIM_ICPSC_DIV1;
    TIM_ICInitStructure.ICFilter = 0x0;
    HAL_TIM_IC_ConfigChannel(&TIM_PWMINPUT_Handle,
                    &TIM_ICInitStructure,ADVANCE_IC1PWM_CHANNEL);
    /* IC2 捕获：下降沿触发 TI1FP2 */
    TIM_ICInitStructure.ICPolarity = TIM_ICPOLARITY_FALLING;
    TIM_ICInitStructure.ICSelection = TIM_ICSELECTION_INDIRECTTI;
    TIM_ICInitStructure.ICPrescaler = TIM_ICPSC_DIV1;
```

(下页继续)

(续上页)

```
    TIM_ICInitStructure.ICFilter = 0x0;

    HAL_TIM_IC_ConfigChannel(&TIM_PWMINPUT_Handle,
                    &TIM_ICInitStructure,ADVANCE_IC2PWM_CHANNEL);
    /* 选择从模式: 复位模式 */
    TIM_SlaveConfigStructure.SlaveMode = TIM_SLAVEMODE_RESET;
    /* 选择定时器输入触发: TI1FP1 */
    TIM_SlaveConfigStructure.InputTrigger = TIM_TS_TI1FP1;
    HAL_TIM_SlaveConfigSynchronization(&TIM_PWMINPUT_Handle,&TIM_
↪SlaveConfigStructure);
    /* 使能捕获/比较 2 中断请求 */
    HAL_TIM_IC_Start_IT(&TIM_PWMINPUT_Handle,TIM_CHANNEL_1);
    HAL_TIM_IC_Start_IT(&TIM_PWMINPUT_Handle,TIM_CHANNEL_2);
}
```

输入捕获配置中，主要初始化三个结构体，时基结构体部分很简单，看注释理解即可。关键部分是输入捕获结构体和从模式结构体的初始化。

首先，我们要选定捕获通道，这里我们用 IC1，然后设置捕获信号的极性，这里我们配置为上升沿，我们需要对捕获信号的每个有效边沿（即我们设置的上升沿）都捕获，所以我们不分频，滤波器我们也不需要用。那么捕获通道的信号来源于哪里呢? IC1 的信号可以是 TI1 输入的 TI1FP1，也可以是从 TI2 输入的 TI2FP1，我们这里选择直连（DIRECTTI），即 IC1 映射到 TI1FP1，即 PWM 信号从 TI1 输入。

我们知道，PWM 输入模式，需要使用两个捕获通道，占用两个捕获寄存器。由输入通道 TI1 输入的信号会分成 TI1FP1 和 TI1FP2，具体选择哪一路信号作为捕获触发信号决定着哪个捕获通道测量的是周期。这里我们选择 TI1FP1 作为捕获的触发信号，那 PWM 信号的周期则存储在 CCR1 寄存器中，剩下的另外一路信号 TI1FP2 则进入 IC2，CCR2 寄存器存储的是脉冲宽度。

测量脉冲宽度我们选择捕获通道 2，即 IC2，设置捕获信号的极性，这里我们配置为下降沿，我们需要对捕获信号的每个有效边沿（即我们设置的下降沿）都捕获，所以我们不分频，滤波器我们也不需要用。那么捕获通道的信号来源于 TI2 输入的 TI2FP1，这里选择间接（INDIRECTTI），PWM 信号从 IC1 输入再进入 IC2.

I2C 作为间接输入模式，我们需要配置他的从模式，即从模式复位模式，定时器触发源为 TIM_TS_TI1FP1，最后使用函数 HAL_TIM_SlaveConfigSynchronization 进行配置。

最后启动定时器的两个通道捕获。

32.7.2.2.6 高级控制定时器中断服务函数

列表 14: 代码清单高级控制定时器中断服务函数

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1) {
        /* 获取输入捕获值 */
        IC1Value = HAL_TIM_ReadCapturedValue(&TIM_PWMINPUT_Handle,
                                             ADVANCE_IC1PWM_CHANNEL);
        IC2Value = HAL_TIM_ReadCapturedValue(&TIM_PWMINPUT_Handle,
                                             ADVANCE_IC2PWM_CHANNEL);
        if (IC1Value != 0) {
            /* 占空比计算 */
            DutyCycle = (float)((IC2Value+1) * 100) / (IC1Value+1);

            /* 频率计算 */
            Frequency = 168000000/168/(float)(IC1Value+1);

        } else {
            DutyCycle = 0;
            Frequency = 0;
        }

    }
}
```

中断服务函数的回调函数中，我们获取 CCR1 和 CCR2 寄存器中的值，当 CCR1 的值不为 0 时，

说明有效捕获到了一个周期，然后计算出频率和占空比。

如果是第一个上升沿中断，计数器会被复位，锁存到 CCR1 寄存器的值是 0，CCR2 寄存器的值也是 0，无法计算频率和占空比。当第二次上升沿到来的时候，CCR1 和 CCR2 捕获到的才是有效的值。

32.7.2.2.7 主函数

列表 15: 代码清单 31-12 main 函数

```
int main(void)
{
    /* 初始化系统时钟为 168MHz */
    SystemClock_Config();
    /* 初始化串口 */
    UARTx_Config();
    /* 初始化基本定时器定时，1s 产生一次中断 */
    TIMx_Configuration();

    while (1) {
        HAL_Delay(500);
        printf("IC1Value = %d  IC2Value = %d ",IC1Value,IC2Value);
        printf("占空比：%0.2f%%   频率：%0.2fHz\n",DutyCycle,Frequency);
    }
}
```

主函数内容非常简单，首先初始化系统时钟、调用 UARTx_Config 函数完成串口初始化配置，该函定义在 bsp _usart.c 文件内。

接下来就是调用 TIMx_Configuration 函数完成定时器配置，该函数定义在 bsp_advance_tim.c 文件内，它只是简单的分别调用 TIMx_GPIO_Config()、TIMx_NVIC_Configuration()、TIM_PWMOUTPUT_Config() 和 TIM_PWMINPUT_Config() 四个函数，完成定时器引脚初始化配置，NVIC 配置，通用定时器输出 PWM 以及高级控制定时器 PWM 输入模式配置。

主函数的无限循环每隔 500ms 输出一次捕获结果。

32.7.3 下载验证

把编译好的程序烧写到开发板，用杜邦线把通用定时器的 PWM 输出引脚（PA5）连接到高级定时器的 PWM 输入引脚（PC6）。然后用 USB 线连接电脑与开发板的 USB 转串口，打开串口调试助手，即可看到捕获到的 PWM 信号的频率和占空比，具体见图 *31_19*。于此同时，可用示波器监控通用定时器输出的 PWM 信号，看下捕获到的信号是否正确，具体见图 *31_20*。

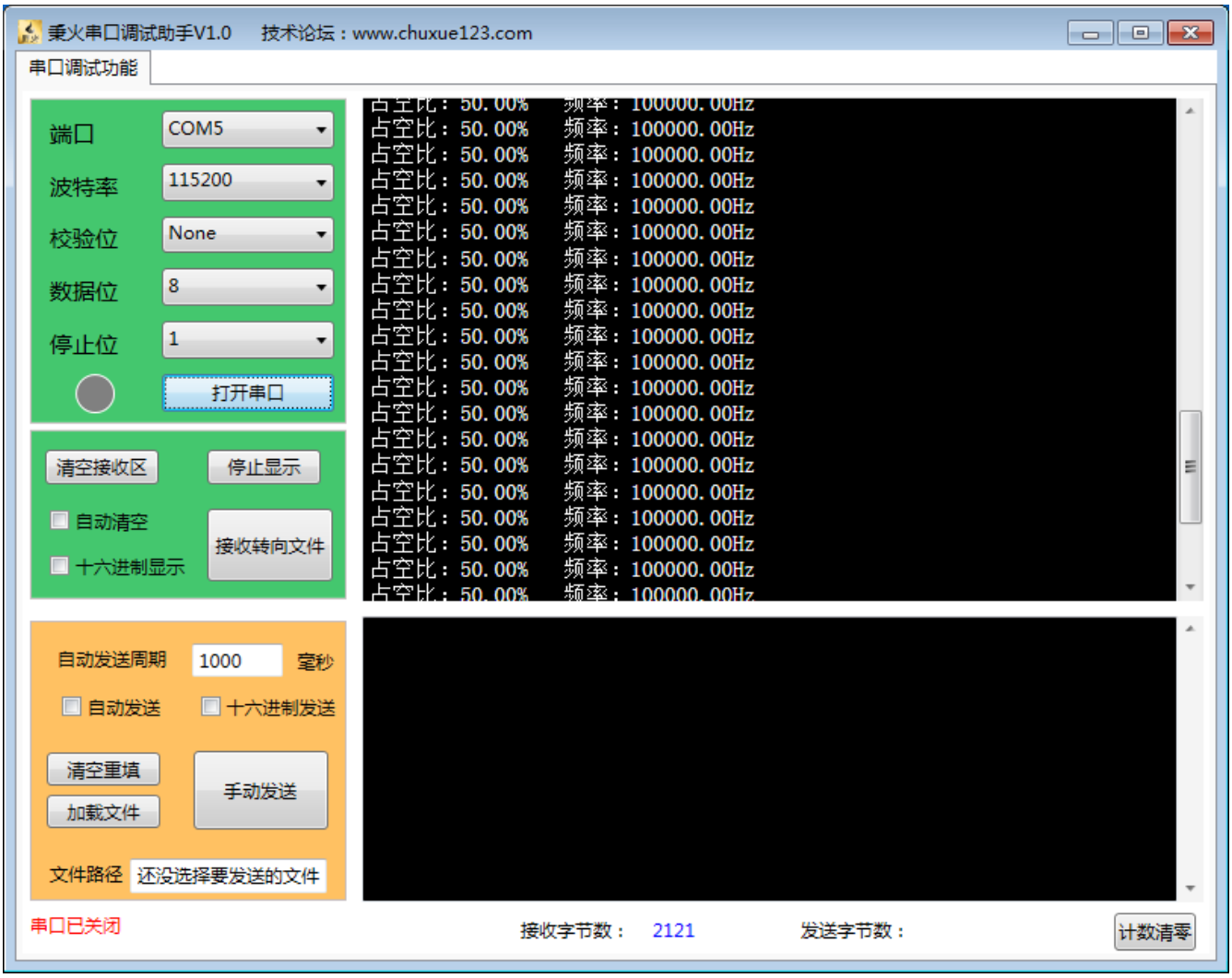


图 31-19 串口调试助手打印的捕获信息

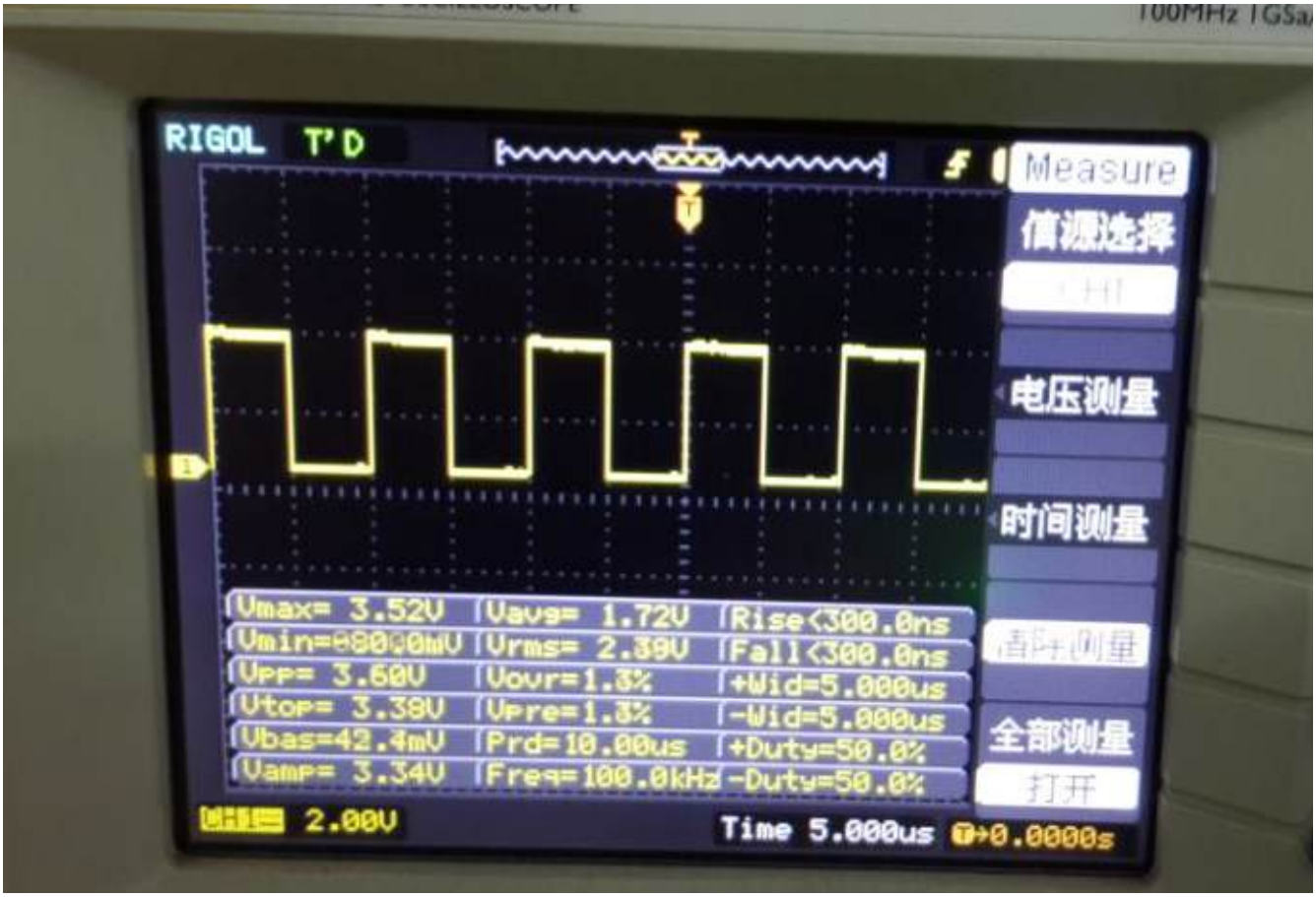


图 31-20 示波器监控的波形

从图 *31_19* 和图 *31_20* 我们可以看到，程序捕获计算出的频率和占空比和示波器监控到的波形的频率和占空比跟一致，所以我们的程序是正确的。

第 33 章　TIM—电容按键检测

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。前面章节我们讲解了基本定时器和高级控制定时器功能，这一章我们将介绍定时器输入捕获一个应用实例，帮助我们更加深入理解定时器。

33.1 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。

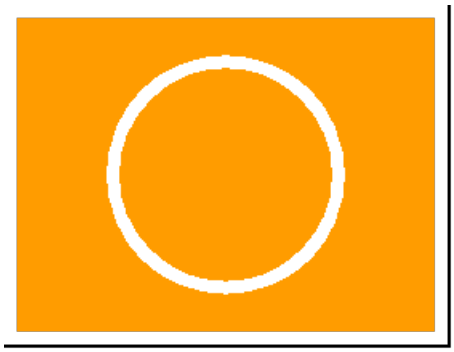


图 32-1 片状电容器

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。

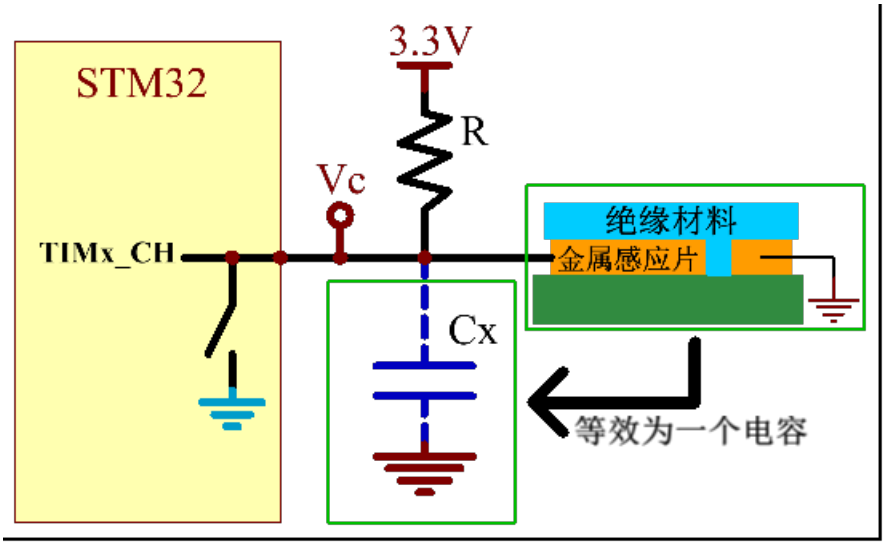


图 32-2 无手指触摸情况

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。

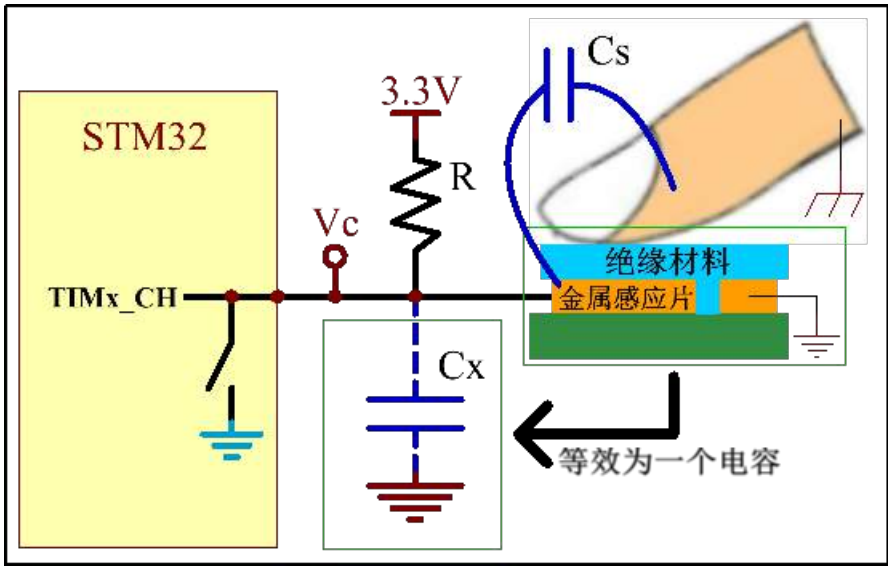


图 32-3 有手指触摸情况

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。

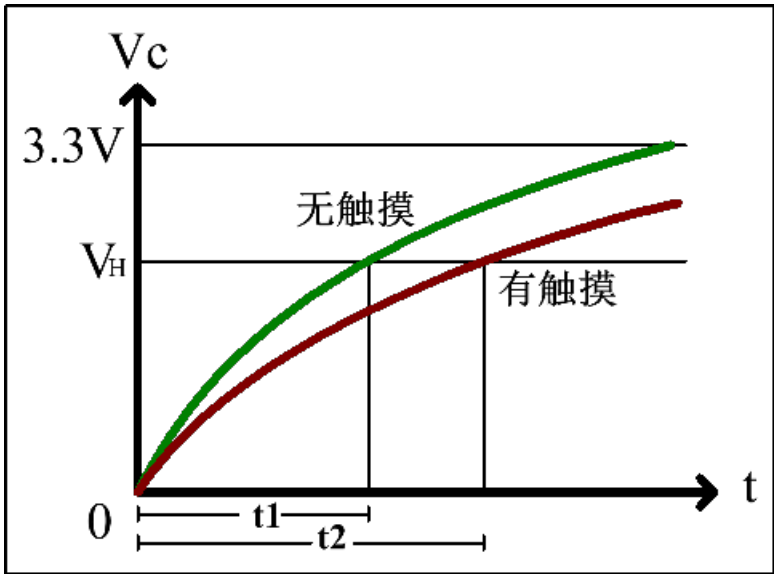


图 32-4 Vc 电压与充电时间关系

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。

33.2 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。

33.2.1 硬件设计

开发板板载一个电容按键，原理图设计参考图 32-5。

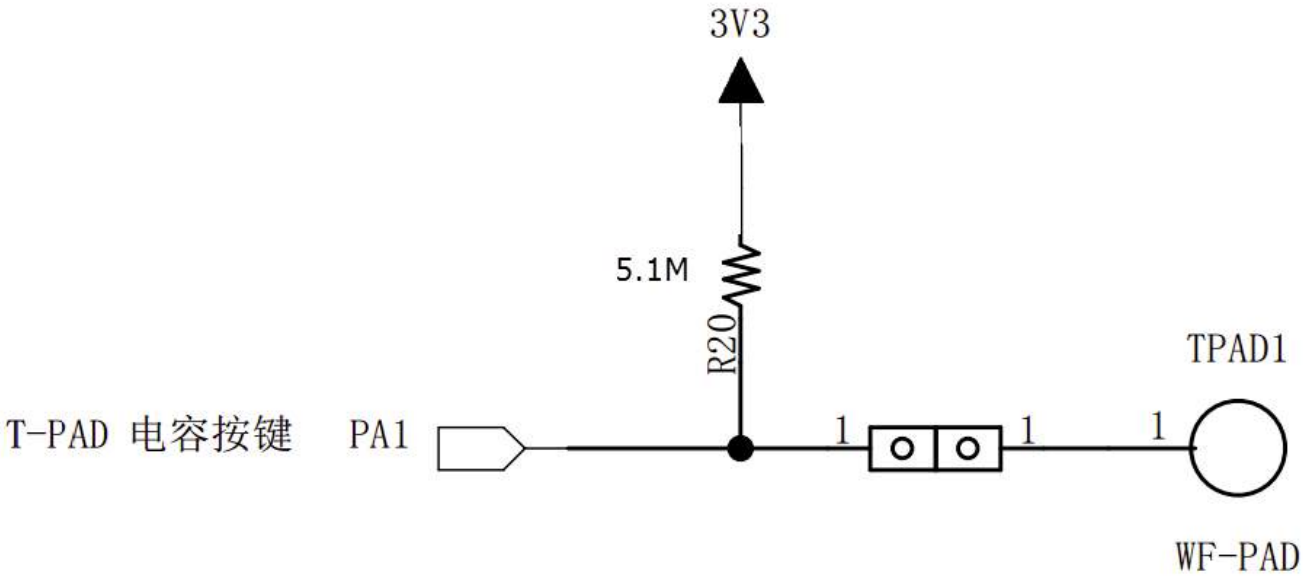


图 32-5 电容按键电路设计

开发板中标示“电容按键”的地方就是电容按键实体，实际上就是一块铜皮，然后在电路板子上通过一根导线连接至定时器通道引脚，并带一个 3.3V 的上拉电阻，电阻的大小和电容按键的大小决定了充电的时间长度。默认上电情况下电容按键会被 3.3V 通过电阻 R45 充满电。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。

33.2.2 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_touchpad.c 和 bsp_touchpad.h 文件用来存放电容按键检测相关函数和宏定义。

33.2.2.1 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

33.2.2.2 软件分析

33.2.2.2.1 电容按键宏定义

列表 1: 代码清单 32-1 宏定义

```
#define TPAD_TIMx                       TIM2
#define TPAD_TIM_CLK_ENABLE()       __TIM2_CLK_ENABLE()

#define TPAD_TIM_Channel_X              TIM_CHANNEL_1
#define TPAD_TIM_GetCaptureX            TIM_GetCapture1

#define TPAD_TIM_GPIO_CLK_ENABLE()   __GPIOA_CLK_ENABLE()
#define TPAD_TIM_CH_PORT                GPIOA
#define TPAD_TIM_CH_PIN                 GPIO_PIN_5
#define TPAD_TIM_AF                     GPIO_AF1_TIM2
```

使用宏定义非常方便程序升级、移植。

开发板选择使用通用定时器 2 的通道 1 连接到电容按键，对应的引脚为 PA5。

33.2.2.2.2 定时器初始化配置

列表 2: 定时器初始化配置

```
static void TIMx_CHx_Cap_Init(uint32_t arr,uint16_t psc)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    TIM_IC_InitTypeDef sConfigIC;
    //使能 TIM 时钟
    TPAD_TIM_CLK_ENABLE();
    //使能通道引脚时钟
    TPAD_TIM_GPIO_CLK_ENABLE();
    //端口配置
    GPIO_InitStructure.Pin = TPAD_TIM_CH_PIN;
    //复用功能
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Alternate = TPAD_TIM_AF;
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    //不带上下拉
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(TPAD_TIM_CH_PORT, &GPIO_InitStructure);
    //初始化 TIM
    //设定计数器自动重装值
    TIM_Handle.Instance = TPAD_TIMx;
    TIM_Handle.Init.Prescaler = psc;
    TIM_Handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    TIM_Handle.Init.RepetitionCounter = 0;
    TIM_Handle.Init.Period = arr;
    TIM_Handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_IC_Init(&TIM_Handle);
    //上升沿触发
    sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
    // 输入捕获选择
    sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
    //配置输入分频，不分频
    sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
    //配置输入滤波器 不滤波
    sConfigIC.ICFilter = 0;
    //初始化捕获通道
    HAL_TIM_IC_ConfigChannel(&TIM_Handle, &sConfigIC, TPAD_TIM_Channel_X);
    //启动 TIM
    HAL_TIM_IC_Start(&TIM_Handle, TPAD_TIM_Channel_X);
}
```

首先定义三个初始化结构体变量，这三个结构体之前都做了详细的介绍，可以参考相关章节理解。

使用外设之前都必须开启相关时钟，这里开启定时器时钟和定时器通道引脚对应端口时钟，并指

定定时器通道引脚复用功能。

接下来初始化配置定时器通道引脚为复用功能，无需上下拉。

然后，配置定时器功能。定时器周期和预分频器值由函数形参决定，采用向上计数方式。指定输入捕获通道，电容按键检测需要采用上升沿触发方式。

最后，启动定时器。

33.2.2.2.3 电容按键复位

列表 3: 电容按键复位

```
static void TPAD_Reset(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    //配置引脚为普通推挽输出
    GPIO_InitStructure.Pin = TPAD_TIM_CH_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    GPIO_InitStructure.Pull = GPIO_PULLDOWN;
    HAL_GPIO_Init(TPAD_TIM_CH_PORT, &GPIO_InitStructure);

    //输出低电平，放电
    HAL_GPIO_WritePin ( TPAD_TIM_CH_PORT,
    TPAD_TIM_CH_PIN ,GPIO_PIN_RESET);
    //保持一小段时间低电平，保证放电完全
    HAL_Delay(5);

    //清除更新标志
    __HAL_TIM_CLEAR_FLAG(&TIM_Handle,TIM_FLAG_CC1);
    __HAL_TIM_CLEAR_FLAG(&TIM_Handle,TIM_FLAG_UPDATE);
    //计数器归 0
    __HAL_TIM_SET_COUNTER(&TIM_Handle,0);
```

(下页继续)

(续上页)

```
    //引脚配置为复用功能，不上、下拉
    GPIO_InitStructure.Pin = TPAD_TIM_CH_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Alternate = TPAD_TIM_AF;
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(TPAD_TIM_CH_PORT,&GPIO_InitStructure);
}
```

该函数实现两个主要功能：控制电容按键放电和复位计数器。

首先，配置定时器通道引脚作为普通 GPIO，使其为下拉的推挽输出模式。然后调用 HAL_GPIO_WritePin 函数输出低电平，为保证放电完整，需要延时一小会时间，这里调用 HAL_Delay 函数完成 5 毫秒的延时。HAL_Delay 函数是定义在 stm32f4xx_hal.c 文件的一个延时函数，它利用系统滴答定时器功能实现毫秒级的精准延时。这个函数再初始化时钟的时候默认已经初始化，可以随时调用。

这里还需要一个注意的地方，在控制电容按键放电的整个过程定时器是没有停止的，计数器还是在不断向上计数的，只是现阶段计数值对我们来说没有意义而已。

然后，清除定时器捕获/比较标志位和更新标志位以及将定时器计数值赋值为 0，使其重新从 0 开始计数。

最后，配置定时器通道引脚为定时器复用功能，不上下拉。在执行完该 GPIO 初始化函数后，电容按键就马上开始充电，定时器通道引脚电压就上升，当达到 1.8V 时定时器就输入捕获成功。所以在执行完 TPAD_Reset 函数后应用程序需要不断查询定时器输入捕获标志，在发送输入捕获时马上读取 TIMx_CCRx 寄存器的值，作为该次电容按键捕获值。

33.2.2.2.4 获取输入捕获值

列表 4: 获取输入捕获值

```
//定时器最大计数值
#define TPAD_ARR_MAX_VAL  0XFFFF

static uint16_t TPAD_Get_Val(void)
{
    /* 先放电完全，并复位计数器 */
    TPAD_Reset();
    //等待捕获上升沿
    while (__HAL_TIM_GET_FLAG(&TIM_Handle,TIM_FLAG_CC1) == RESET) {
        //超时了，直接返回 CNT 的值
        if (__HAL_TIM_GET_COUNTER( &TIM_Handle)>TPAD_ARR_MAX_VAL-500)
            return __HAL_TIM_GET_COUNTER( &TIM_Handle);
    };
    /* 捕获到上升沿后输出 TIMx_CCRx 寄存器值 */
    return HAL_TIM_ReadCapturedValue(&TIM_Handle, TIM_CHANNEL_1);
}
```

开始是 TPAD_ARR_MAX_VAL 的宏定义，它指定定时器自动重载寄存器 (TIMx_ARR) 的值。

TPAD_Get_Val 函数用来获取一次电容按键捕获值，包括电容按键放电和输入捕获过程。

先调用 TPAD_Reset 函数完成电容按键放电过程，并复位计数器。

接下来，使用 __HAL_TIM_GET_FLAG 函数获取当前计数器的输入捕获状态，如果成功输入捕获就使用 HAL_TIM_ReadCapturedValue 函数获取此刻定时器捕获/比较寄存器的值并返回该值。如果还没有发生输入捕获，说明还处于充电过程，就进入等待状态。

为防止无限等待情况，加上超时处理函数，如果发生超时则直接返回计数器值。实际上，如果发生超时情况，很大可能是硬件出现问题。

33.2.2.2.5 获取最大输入捕获值

列表 5: 获取最大输入捕获值

```
static uint16_t TPAD_Get_MaxVal(uint8_t n)
{
    uint16_t temp=0;
    uint16_t res=0;
    while (n--) {
        temp=TPAD_Get_Val();//得到一次值
        if (temp>res)res=temp;
    };
    return res;
}
```

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 n 次发生捕获中最大的捕获值。

33.2.2.2.6 电容按键捕获初始化

列表 6: 电容按键捕获初始化

```
uint8_t TPAD_Init(void)
{
    uint16_t buf[10];
    uint16_t temp;
    uint8_t j,i;

    //设定定时器预分频器目标时钟为: 9MHz(168Mhz/24)
    TIMx_CHx_Cap_Init(TPAD_ARR_MAX_VAL,24-1);
    for (i=0; i<10; i++) { //连续读取 10 次
        buf[i]=TPAD_Get_Val();
```

(下页继续)

(续上页)

```
        Delay_ms(10);
    }
    for (i=0; i<9; i++) { //排序
        for (j=i+1; j<10; j++) {
            if (buf[i]>buf[j]) { //升序排列
                temp=buf[i];
                buf[i]=buf[j];
                buf[j]=temp;
            }
        }
    }
    temp=0;
    //取中间的 6 个数据进行平均
    for (i=2; i<8; i++) {
        temp+=buf[i];
    }
    tpad_default_val=temp/6;
    /* printf 打印函数调试使用，用来确定阈值 TPAD_GATE_VAL，在应用工程中应注释掉 */
    printf("tpad_default_val:%d\r\n",tpad_default_val);

    //初始化遇到超过 TPAD_ARR_MAX_VAL/2 的数值，不正常！
    if (tpad_default_val>TPAD_ARR_MAX_VAL/2) {
        return 1;
    }
    return 0;
}
```

该函数实现定时器初始化配置和无触摸时电容按键捕获值确定功能。它一般在 main 函数靠前位置调用完成电容按键初始化功能。

程序先调用 TIMx_CHx_Cap_Init 函数完成定时器基本初始化和输入捕获功能配置，两个参数用于设置定时器的自动重载计数和定时器时钟频率，这里自动重载计数被赋值为 TPAD_ARR_MAX_VAL，这里对该值没有具体要求，不要设置过低即可。定时器时钟配置设置

为 9MHz 为合适，实验中用到 TIM2，默认使用内部时钟为 168MHz，经过参数设置预分频器为 24 分频，使定时器时钟为 9MHz。

接下来，循环 10 次读取电容按键捕获值，并保存在数组内。TPAD_Init 函数一般在开机时被调用，所以认为 10 次读取到的捕获值都是无触摸状态下的捕获值。

然后，对 10 个捕获值从小到大排序，取中间 6 个的平均数作为无触摸状态下的参考捕获值，并保存在 tpad_default_val 变量中，该值对应图 *32_4* 中的时间 t1。

程序最后会检测 tpad_default_val 变量的合法性。

33.2.2.2.7 电容按键状态扫描

列表 7: 电容按键状态扫描

```
//阈值：捕获时间必须大于 (tpad_default_val + TPAD_GATE_VAL)，才认为是有效触摸.
#define TPAD_GATE_VAL   100

uint8_t TPAD_Scan(uint8_t mode)
{
    //0，可以开始检测;>0，还不能开始检测
    static uint8_t keyen=0;
    //扫描结果
    uint8_t res=0;
    //默认采样次数为 3 次
    uint8_t sample=3;
    //捕获值
    uint16_t rval;

    if (mode) {
        //支持连按的时候，设置采样次数为 6 次
        sample=6;
        //支持连按
        keyen=0;
```

(下页继续)

(续上页)

```
    }
    /* 获取当前捕获值 (返回 sample 次扫描的最大值) */
    rval=TPAD_Get_MaxVal(sample);
    /* printf 打印函数调试使用，用来确定阈值 TPAD_GATE_VAL，在应用工程中应注释掉 */
//  printf("scan_rval=%d\n",rval);

    //大于 tpad_default_val+TPAD_GATE_VAL，且小于 10 倍 tpad_default_val，则有效
    if (rval>(tpad_default_val+TPAD_GATE_VAL)&&rval<(10*tpad_default_val))
↪{
        //keyen==0，有效
        if (keyen==0) {
            res=1;
        }
        keyen=3;         //至少要再过 3 次之后才能按键有效
    }

    if (keyen) {
        keyen--;
    }
    return res;
}
```

TPAD_GATE_VAL 用于指定电容按键触摸阈值，当实时捕获值大于该阈值和无触摸捕获参考值 tpad_default_val 之和时就认为电容按键有触摸，否则认为没有触摸。阈值大小一般需要通过测试得到，一般做法是通过串口在 TPAD_Init 函数中把 tpad_default_val 值打印到串口调试助手并记录下来，在 TPAD_Scan 函数中也把实时捕获值打印出来，在运行时触摸电容按键，获取有触摸时的捕获值，这样两个值对比就可以大概确定 TPAD_GATE_VAL。

TPAD_Scan 函数用来扫描电容按键状态，需要被循环调用，类似独立按键的状态扫描函数。它有一个形参，用于指定电容按键的工作模式，当为赋值为 1 时，电容按键支持连续触发，即当一直触摸不松开时，每次运行 TPAD_Scan 函数都会返回电容按键被触摸状态，直到松开手指，才返

回无触摸状态。当参数赋值为 0 时，每次触摸函数只返回一次被触摸状态，之后就总是返回无触摸状态，除非松开手指再触摸。TPAD_Scan 函数有一个返回值，用于指示电容按键状态，返回值为 0 表示无触摸，为 1 表示有触摸。

TPAD_Scan 函数主要是调用 TPAD_Get_MaxVal 函数获取当前电容按键捕获值，该值这里指定在连续触发模式下取 6 次扫描的最大值为当前捕获值，如果是不连续触发只取三次扫描的最大值。正常情况下，如果无触摸，当前捕获值与捕获参考值相差很小；如果有触摸，当前捕获值比捕获参考值相差较大，此时捕获值对应图 *32_4* 的时间 t2。

接下来比较当前捕获值与无触摸捕获参考值和阈值之和的关系，以确定电容按键状态。这里为增强可靠性，还加了当前捕获值不能超过参考值的 10 倍的限制条件，因为超过 10 倍关系几乎可以认定为出错情况。

33.2.2.2.8 电容按键状态扫描

列表 8: main 函数

```
int main(void)
{
    /* 系统时钟初始化成 168 MHz */
    SystemClock_Config();
    /* 串口初始化 */
    UARTx_Config();
    /* 蜂鸣器端口初始化 */
    BEEP_GPIO_Config();
    /* 初始化电容按键 */
    TPAD_Init();
    /* 控制 IO */
    while (1) {
        if (TPAD_Scan(0)) {
            BEEP_ON;
            HAL_Delay(100);
            BEEP_OFF;
```

(下页继续)

(续上页)

```
            }
        }
    }
```

主函数分别调用 SystemClock_Config ()、UARTx_Config() 和 Beep_GPIO_Config() 完成系统时钟、串口和蜂鸣器的初始化。TPAD_Init 函数初始化配置定时器，并获取无触摸时的捕获参考值。

无限循环中调用 TPAD_Scan 函数完成电容按键状态扫描，指定为不连续触发方式。如果检测到有触摸就让蜂鸣器响 100ms，然后关掉

33.2.3 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 34 章　IWDG—独立看门狗

本章参考资料：《STM32F4XX 中文参考手册》IWDG 章节。

学习本章时，配合《STM32F4XX 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。

34.1 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。

34.2 IWDG 功能框图剖析

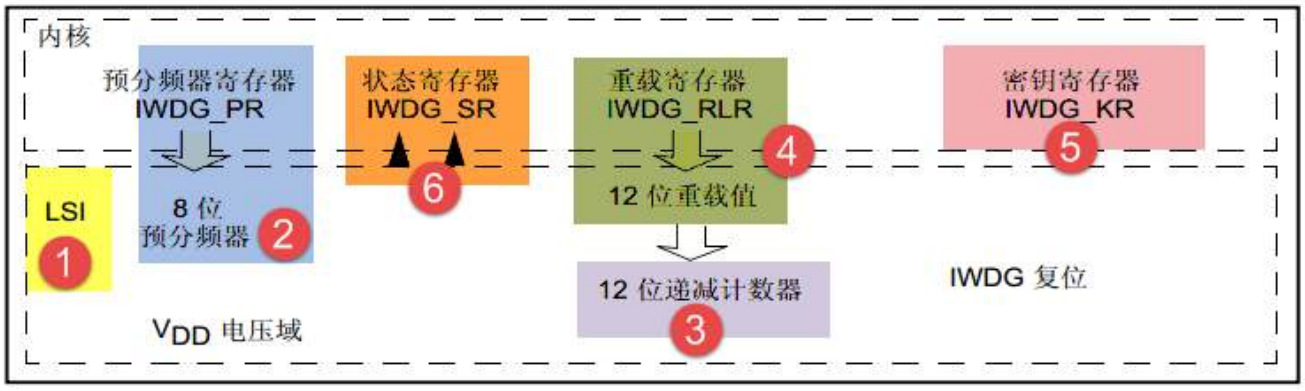


图 33-1 IWDG 功能框图

34.2.1 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一定非常精确，只适用于对时间精度
要求比较低的场合。

34.2.2 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。

34.2.3 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。

34.2.4 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。

34.2.5 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。

34.2.6 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。

34.3 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。

34.4 IWDG 超时实验

34.4.1 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。

34.4.2 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。

34.4.2.1 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。

34.4.2.1.1 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)
{
    IWDG_Handle.Instance = IWDG;
    // 设置预分频器值
    IWDG_Handle.Init.Prescaler = prv;
    // 设置重装载寄存器值
```

(下页继续)

(续上页)

```
    IWDG_Handle.Init.Reload = rlv;
    // 设置要与向下计数器进行比较的窗口值
    IWDG_Handle.Init.Window = IWDG_WINDOW_DISABLE;
    // 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。

34.4.2.1.2 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);
}
```

喂狗只是一个俗语，它的实际作用就是在递减计数器的值在没有递减到 0 的时候及时的刷新递减计数器的值，否则将产生系统复位，让程序重新执行。

34.4.2.1.3 主函数

列表 4: 代码清单 33-4

```
int main(void)
{
    /* 系统时钟初始化成 168 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
        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
```

(下页继续)

(续上页)

```
    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。

34.4.3 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 35 章　WWDG—窗口看门狗

本章参考资料：《STM32F4xx 中文参考手册》WWDG 章节。

学习本章时，配合《STM32F4xx 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。

35.1 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

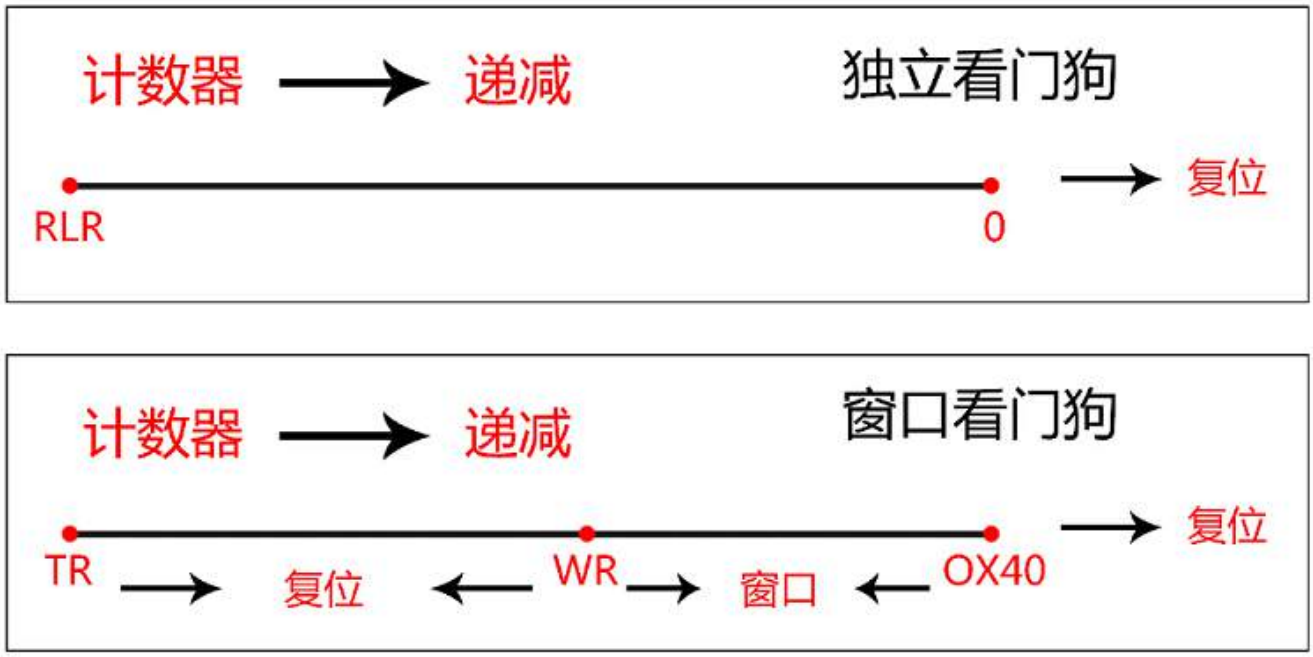


图 34-1 IWDG 与 WWDG 区别

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。

35.2 WWDG 功能框图剖析

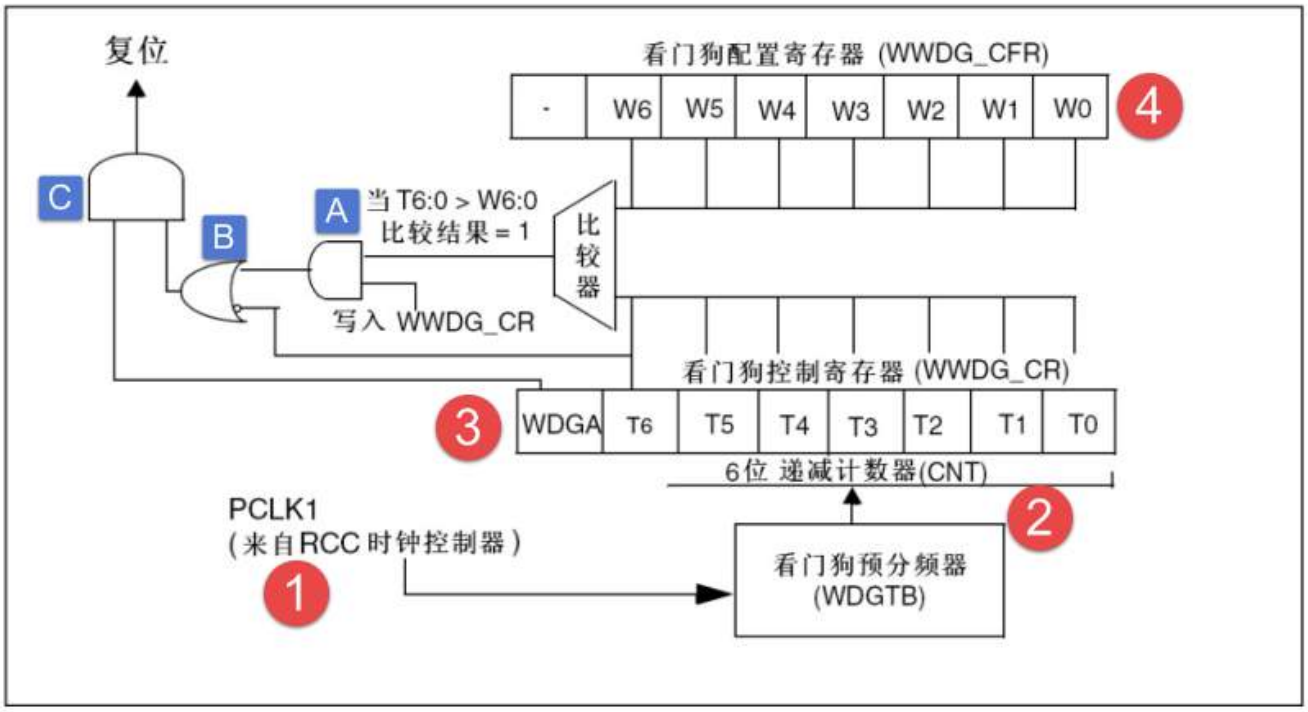


图 34-2 窗口看门狗功能框图

35.2.1 窗口看门狗时钟

窗口看门狗时钟来自 PCLK1，PCLK1 最大是 42M，由 RCC 时钟控制器开启。

35.2.2 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)。

35.2.3 □ 计数器

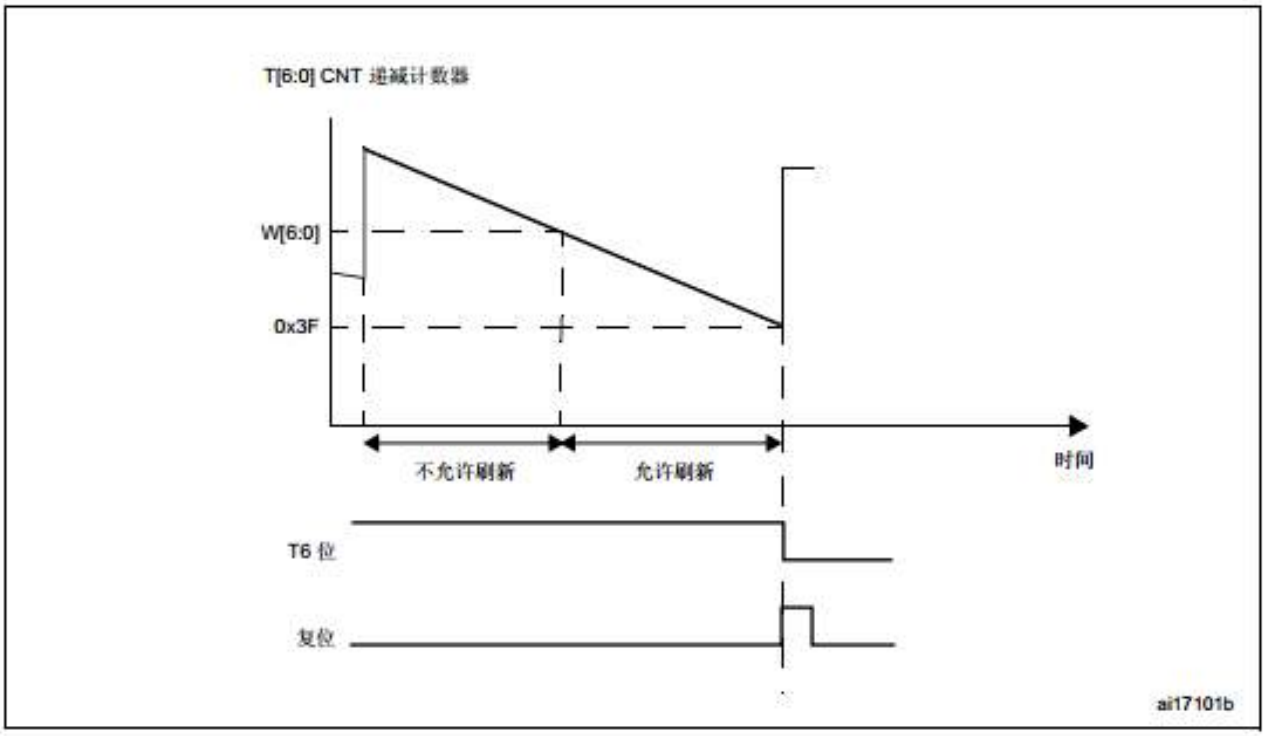
窗口看门狗的计数器是一个递减计数器，共有 7 位，其值存在控制寄存器 CR 的位 6:0，即 T[6:0]，当 7 个位全部为 1 时是 0X7F，这个是最大值，当递减到 T6 位变成 0 时，即从 0X40 变为 0X3F 时候，会产生看门狗复位。这个值 0X40 是看门狗能够递减到的最小值，所以计数器的值只能是：0X40~0X7F 之间，实际上真正用来计数的是 T[5:0]。当递减计数器递减到 0X40 的时候，还不会马上产生复位，如果使能了提前唤醒中断：CFR 位 9 EWI 置 1，则产生提前唤醒中断，如果真进入了这个中断的话，就说明程序肯定是出问题了，

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。

35.2.4 □ 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。

35.2.5 □ 计算看门狗超时时间



超时值的计算公式如下：

$$t_{WWDG} = t_{PCLK1} \times 4096 \times 2^{WDGTB} \times (t[5:0] + 1) \quad (ms)$$

其中：

t_{WWDG}：WWDG 超时

t_{PCLK1}：APB1 时钟周期，以 ms 为测量单位

有关 T_{WWDG} 的最小值和最大值，请参见下表。

表 87. 30 MHz (f_{PCLK1}) 时的超时值

预分频器	WDGTB	最小超时 (µs) T[5:0] = 0x00	最大超时 (ms) T[5:0] = 0x3F
1	0	136.53	8.74
2	1	273.07	17.48
4	2	546.13	34.95
8	3	1092.27	69.91

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 = 30MHZ 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 = 136.53us，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=8.74ms。同理，当 WDGTB 等于 1/2/3 时，代入公式即可。

35.3 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。

35.4 WWDG 喂狗实验

35.4.1 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。

35.4.2 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。

35.4.2.1 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。

35.4.2.2 WWDG 配置函数

列表 1: 代码 34-1 WWDG 配置函数

```
/* WWDG 配置函数
* tr : 递减计时器的值， 取值范围为: 0x7f~0x40, 超出范围会直接复位
* wr : 窗口值, 取值范围为: 0x7f~0x40
* prv: 预分频器值, 取值可以是
* @arg WWDG_PRESCALER_1: WWDG counter clock = (PCLK1(42MHz)/4096)/1
约 13184 76us
* @arg WWDG_PRESCALER_2: WWDG counter clock = (PCLK1(42MHz)/4096)/2
约 6592Hz 152us
* @arg WWDG_PRESCALER_4: WWDG counter clock = (PCLK1(42MHz)/4096)/4
约 3296Hz 304us
* @arg WWDG_PRESCALER_8: WWDG counter clock = (PCLK1(42MHz)/4096)/8
约 1648Hz  608us
*
* 例: tr = 127(0x7f, tr 的最大值)
*     wr = 80 (0x50, 0x40 为最小 wr 最小值)
*    prv = WWDG_PRESCALER_8
* 窗口时间为 608 * (127-80) = 28.6ms < 刷新窗口 < ~608 * 64 = 38.9ms
* 也就是说调用 WWDG_Config 进行这样的配置, 若在之后的 28.6ms 前喂狗,
```

(下页继续)

(续上页)

```
* 系统会复位，在 38.9ms 后没有喂狗，系统也会复位。
* 需要在刷新窗口的时间内喂狗，系统才不会复位。
*/
void WWDG_Config(uint8_t tr, uint8_t wr, uint32_t prv)
{
    // 开启 WWDG 时钟
    __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。

35.4.2.3 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。

35.4.2.4 窗口看门狗中断服务函数和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;
}
```

35.4.2.5 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_CNT );
}
```

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。

35.4.2.6 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 168 MHz */
    SystemClock_Config();
```

(下页继续)

(续上页)

```
    /* 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 0X50，分频系数为 8 分频，则计数器减 1 的时间约为 608us。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。

35.4.2.6.1 下载验证

把编译好的程序下载到开发板，LED3 被点亮，一段时间之后熄灭，之后 LED3 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 LED3 会再被点亮一次。中断服务程序中的 LED 也没被点亮过，说明喂狗正常。

第 36 章　SDIO—SD 卡读写测试

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》以及 SD 简易规格文件《Physical Layer Simplified Specification V2.0》(版本号：2.00)。

特别说明，本书内容是以 STM32F42x 系列控制器资源讲解。

阅读本章内容之前，建议先阅读 SD 简易规格文件。

36.1 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*。

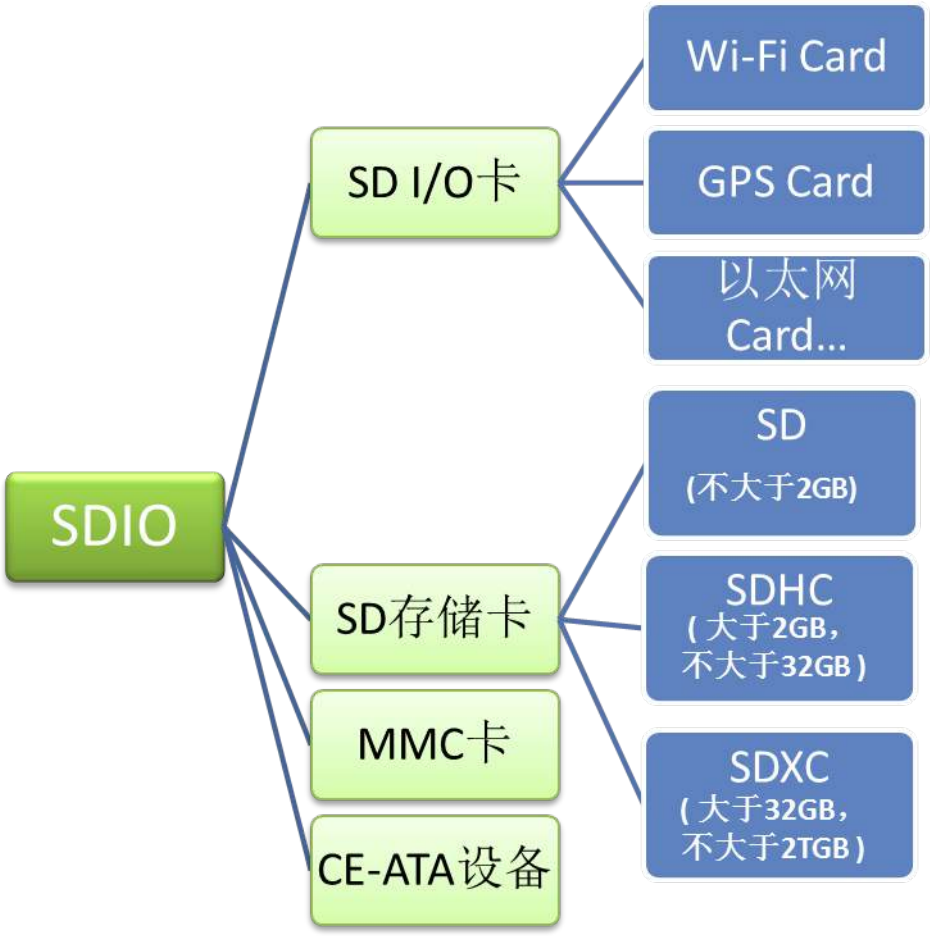


图 35-1 SDIO 接口的设备

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。

36.2 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。

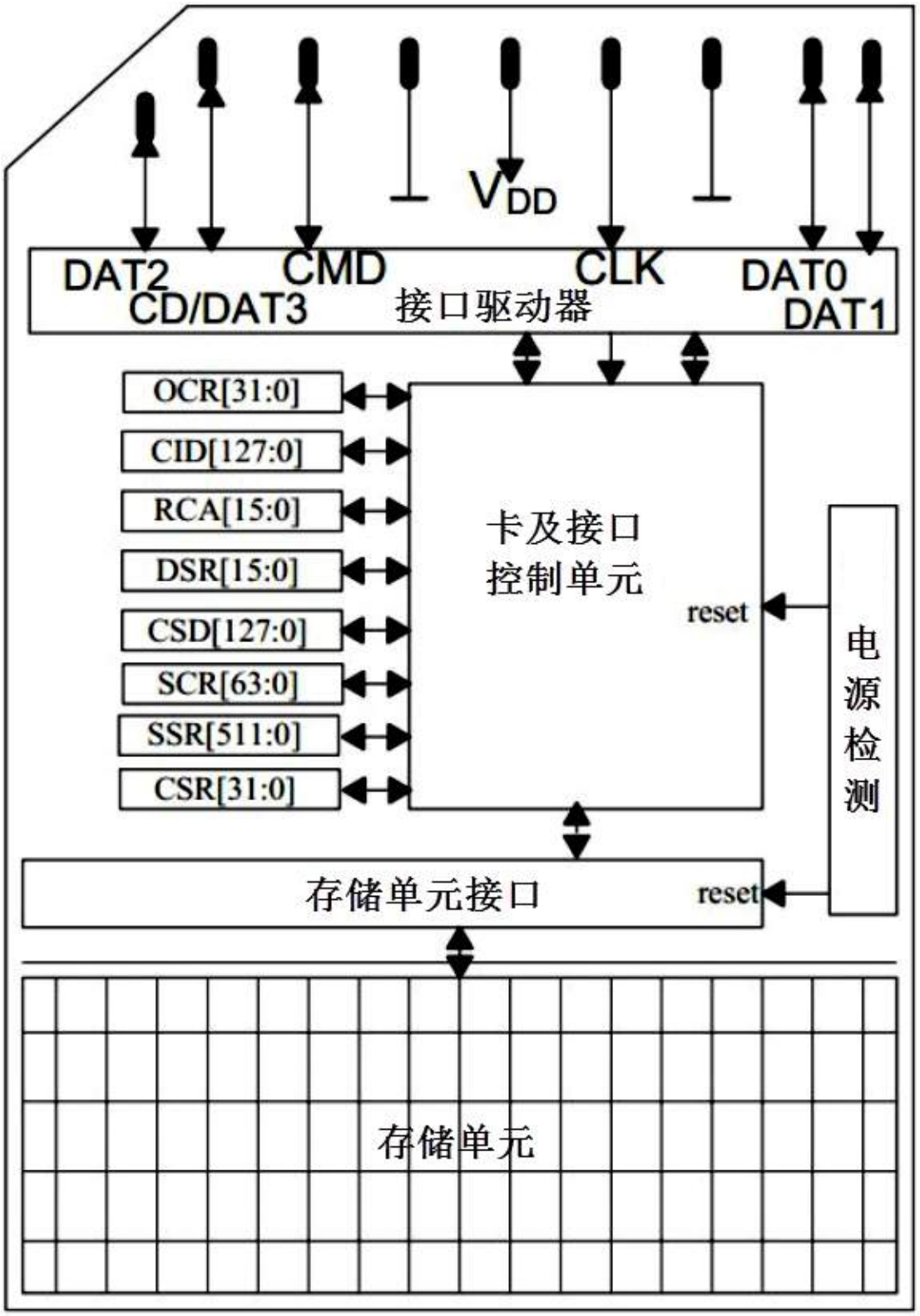

VDD
DAT2
CD/DAT3
CMD
CLK
DAT0
DAT1
接口驱动器
OCR[31:0]
CID[127:0]
RCA[15:0]
DSR[15:0]
CSD[127:0]
SCR[63:0]
SSR[511:0]
CSR[31:0]
卡及接口
控制单元
reset
电源检测
存储单元接口
reset
存储单元

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。

36.3 SDIO 总线

36.3.1 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。

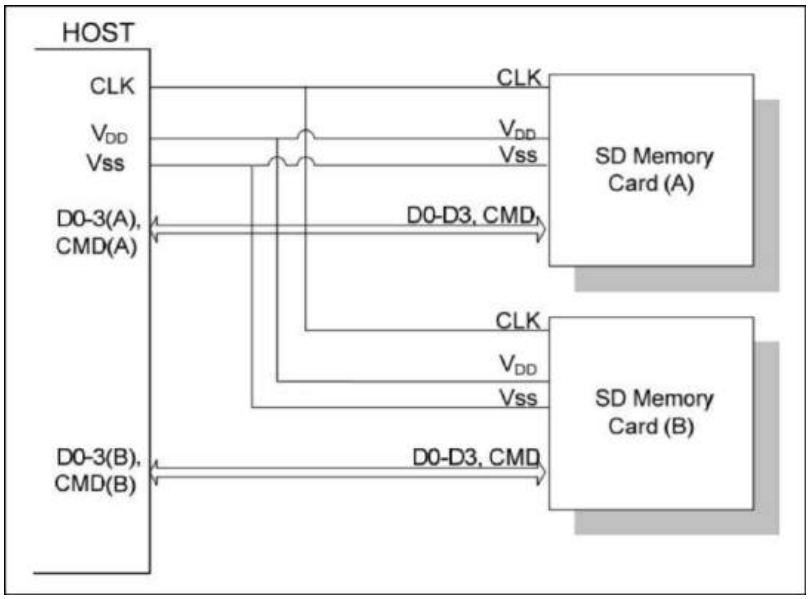


图 35-3 SD 卡总线拓扑

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。

36.3.2 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。

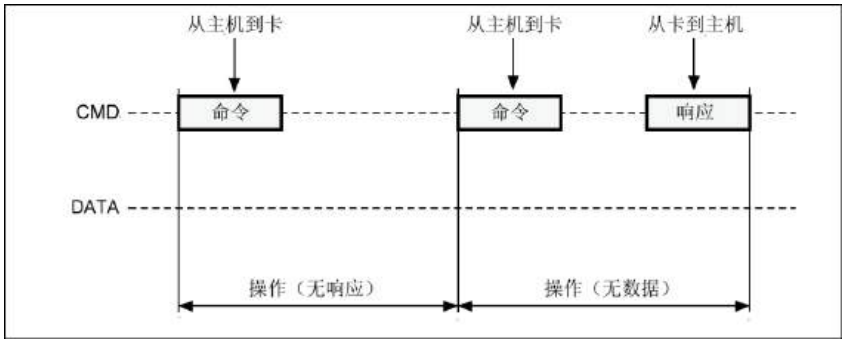


图 35-4 命令与响应交互

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。

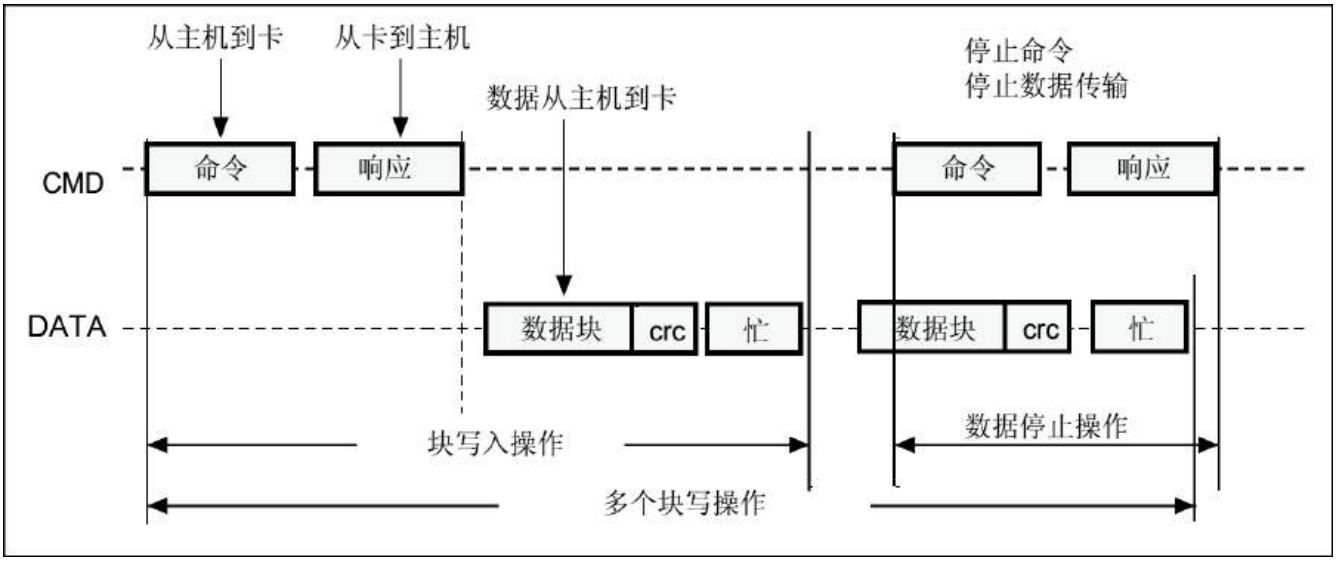


图 35-5 多块写入操作

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*。

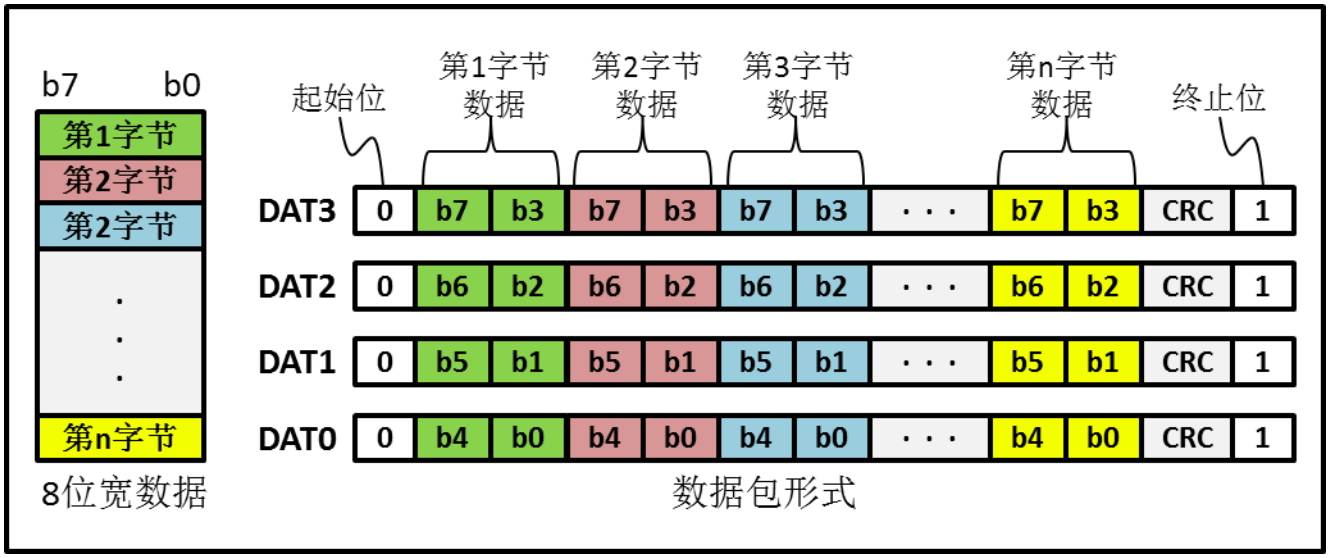


图 35-6 8 位宽数据包传输

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*。

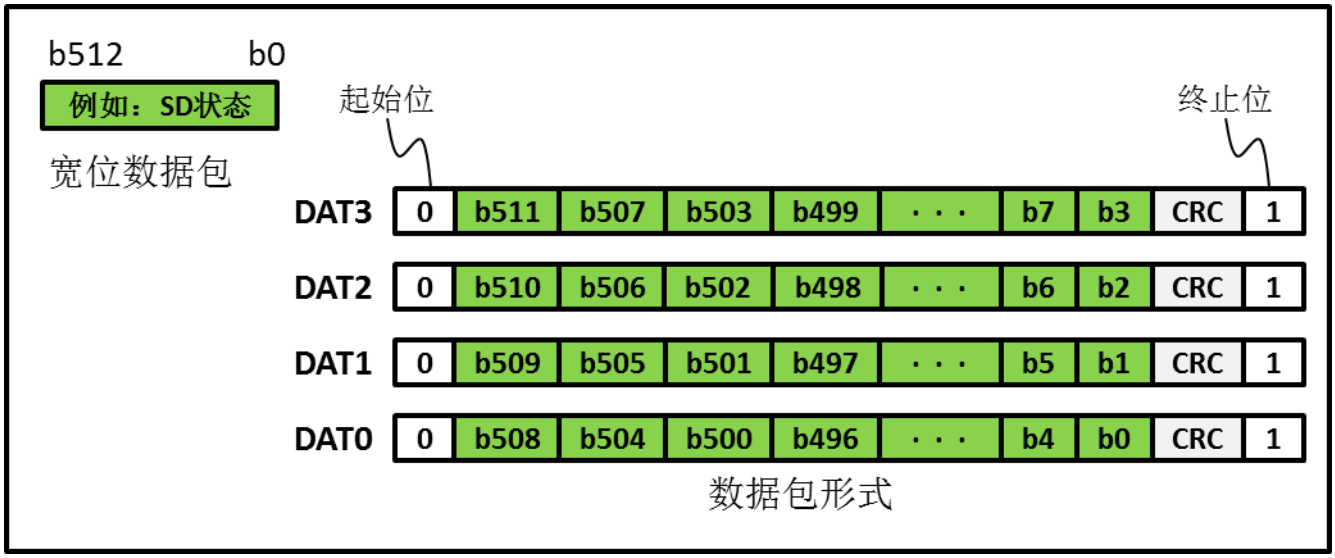


图 35-7 宽位数据包传输

36.3.3 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。

36.3.3.1 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*。

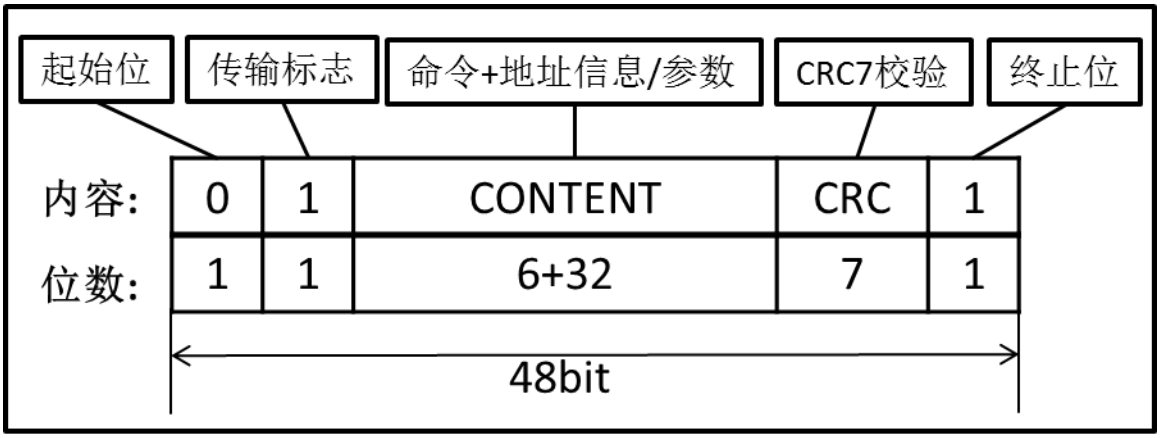


图 35-8 SD 命令格式

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。

36.3.3.2 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。

36.3.3.3 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
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。
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	强制卡停止传输
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)					
CMD16	ac	[31:0]块长度	R1	SET_BLOCK_LEN	对于标准SD卡，设置块命令的长度，对于SDHC卡块命令长度固定为512字节。
CMD17	adtc	[31:0]数据地址	R1	READ_SINGLE_BLOCK	对于标准卡，读取SEL_BLOCK_LEN长度字节的块；对于SDHC卡，读取512字节的块。
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)					
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_MULTIPLE_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
加锁命令(CLass 7)					
CMD42	adtc	[31:0]保留	R1	LOCK_UNLOCK	加锁/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）。
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

36.3.4 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		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寄存器响应）						
描述	起始位	传输位	CMD3	RCA寄存器	卡状态位	CRC7	终止位
Bit	47	46	[45:40]	[39:8]		[7:1]	0
位宽	1	1	6	16	16	7	1
值	"0"	"0"	"000011"	x	x	x	"1"
备注	专用于命令CMD3的响应						
R7（发布的RCA寄存器响应）							
描述	起始位	传输位	CMD8	保留	接收电压	检测模式	CRC7	终止位
Bit	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
位宽	1	1	6	20	4	8	7	1
值	"0"	"0"	"001000"	"00000h"	x	x	x	"1"
备注	专用于命令CMD8的响应，返回卡支持电压范围和检测模式							

36.4 SD 卡的操作模式及切换

36.4.1 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)

36.4.2 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*。

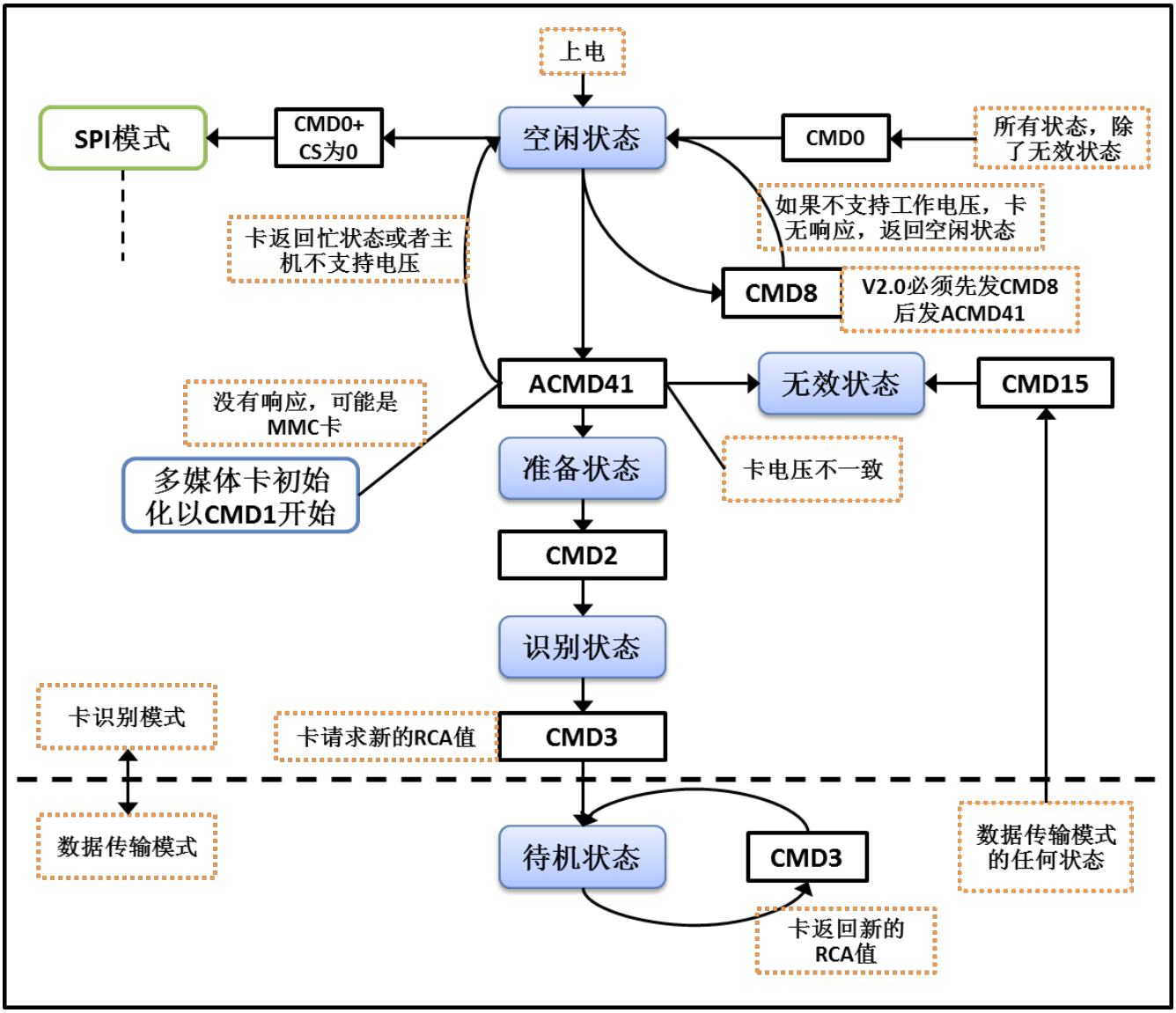


图 35-9 卡识别模式状态转换图

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。

36.4.3 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*。

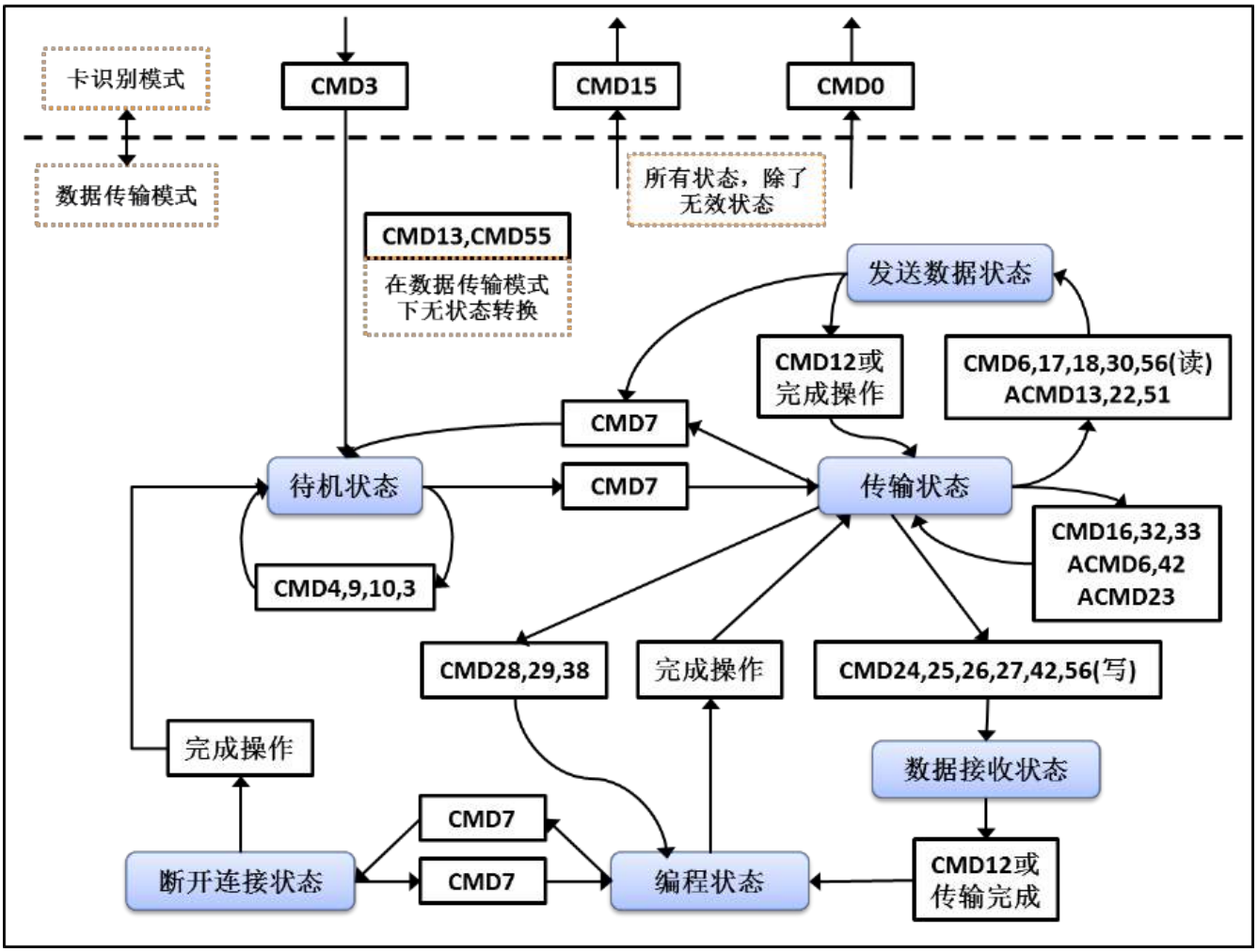


图 35-10 数据传输模式卡状态转换

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。

36.5 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。

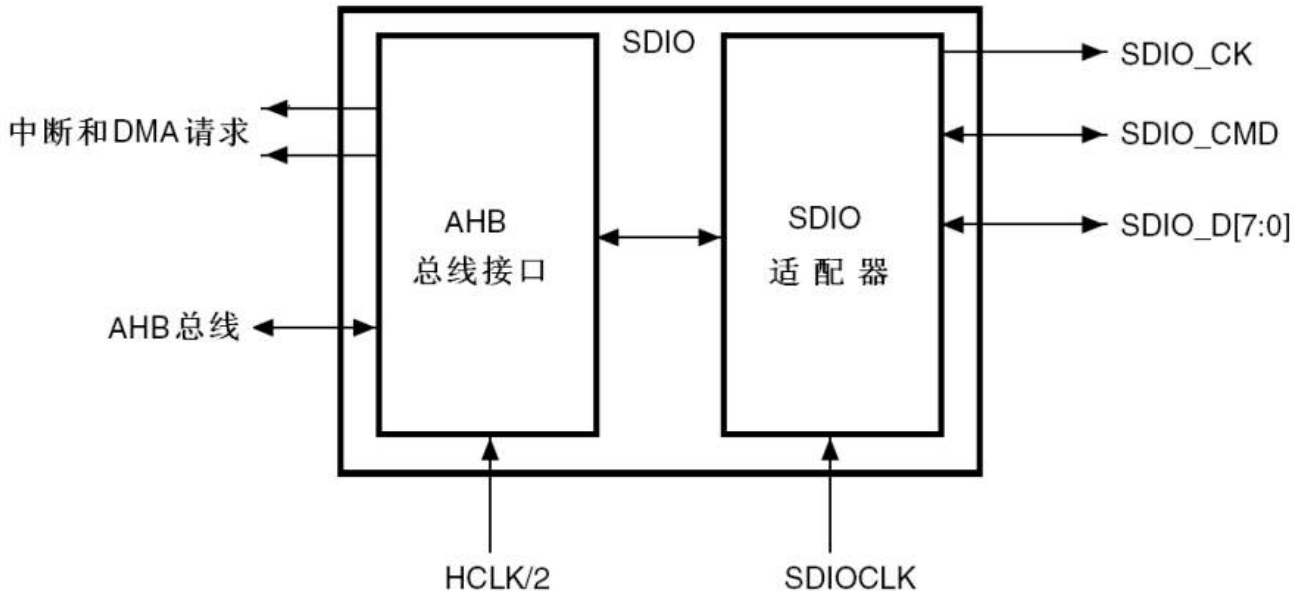


图 35-11 SDIO 功能框图

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。

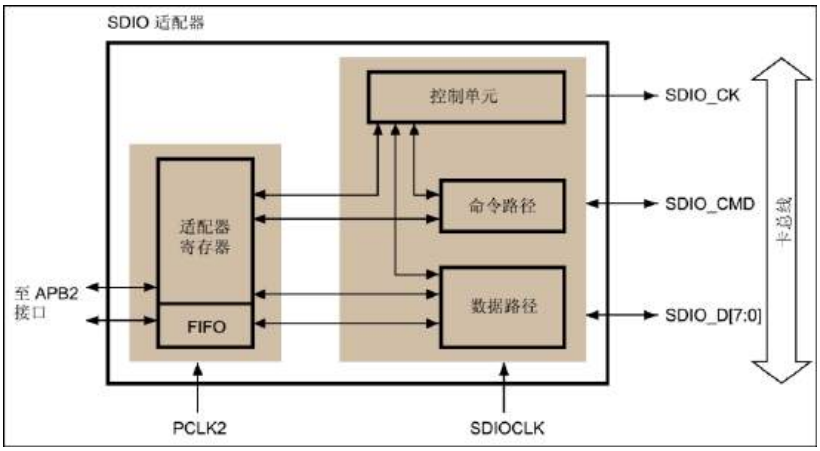


图 35-12 SDIO 适配器框图

36.5.1 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。

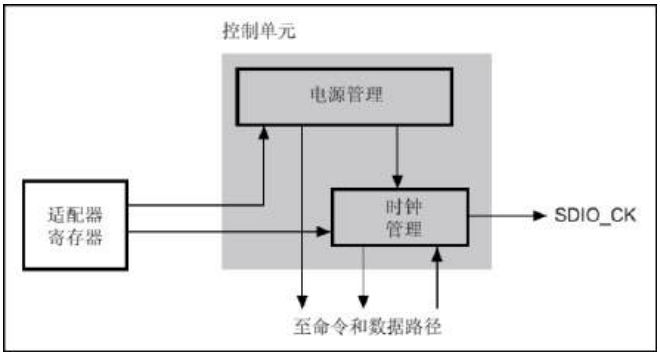


图 35-13 SDIO 适配器控制单元

36.5.2 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*。

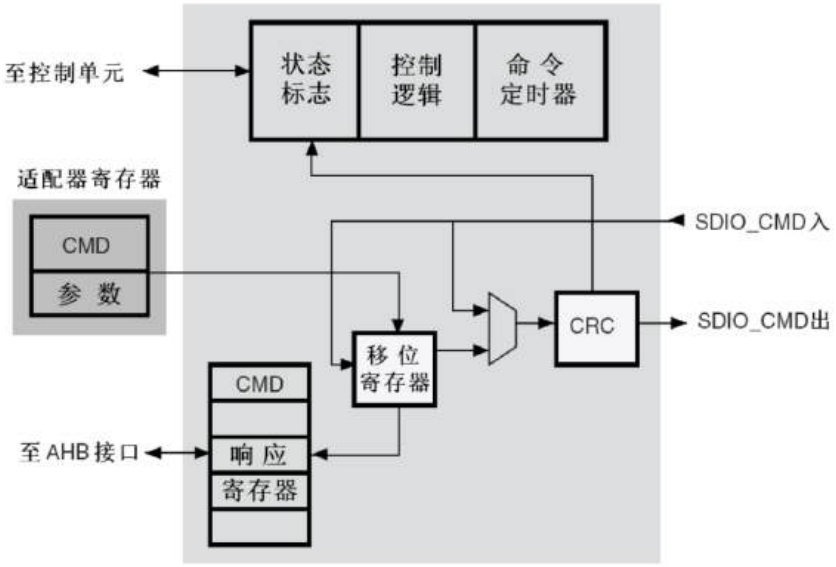


图 35-14 SDIO 适配器命令路径

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*。

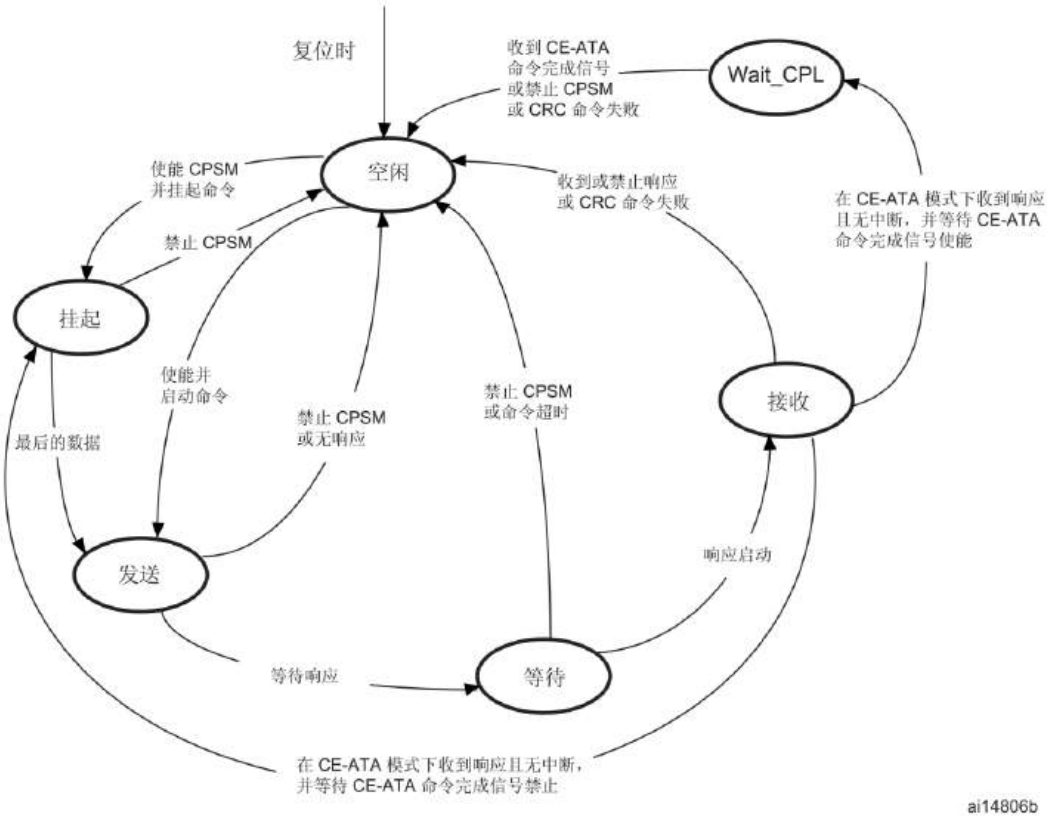


图 35-15 CPSM 状态机描述图

36.5.3 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*。

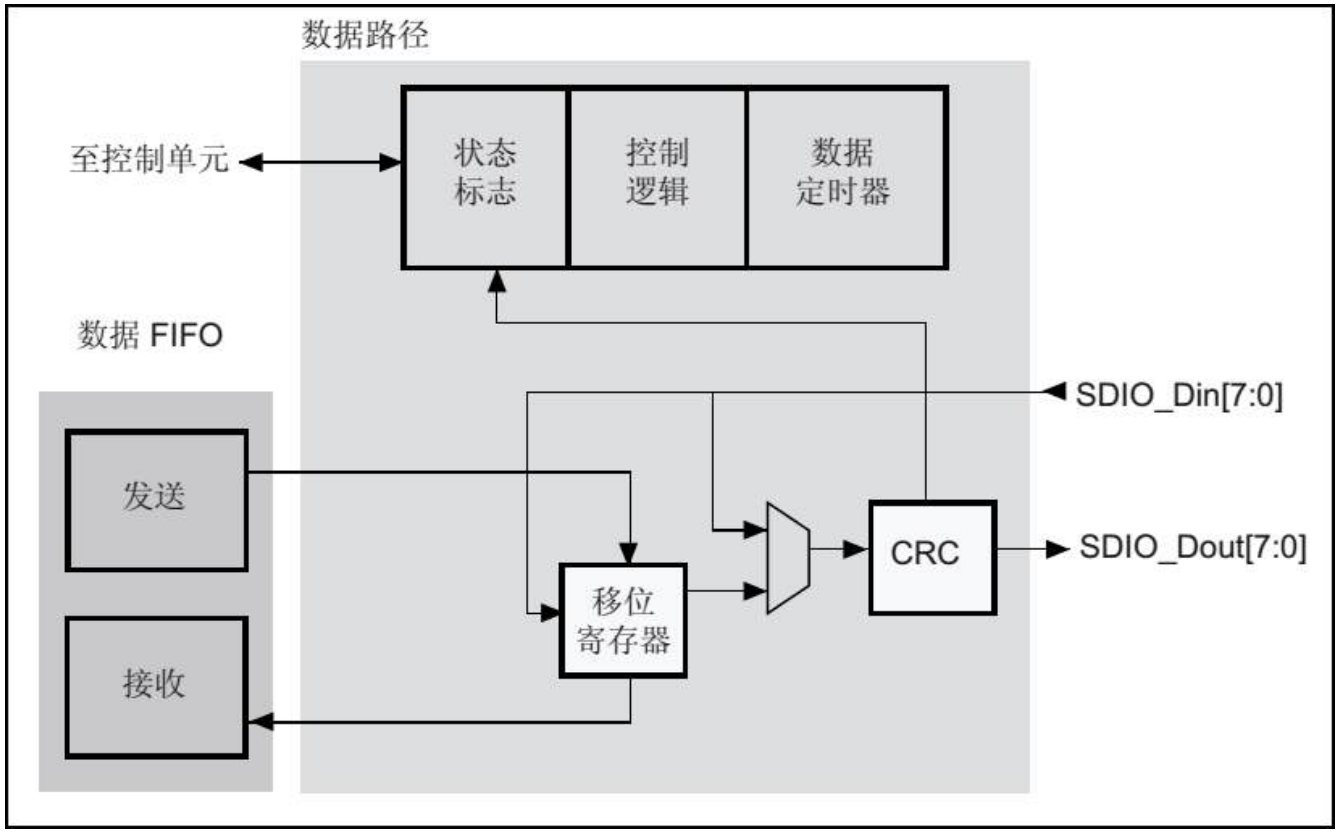


图 35-16 SDIO 适配器数据路径

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*。

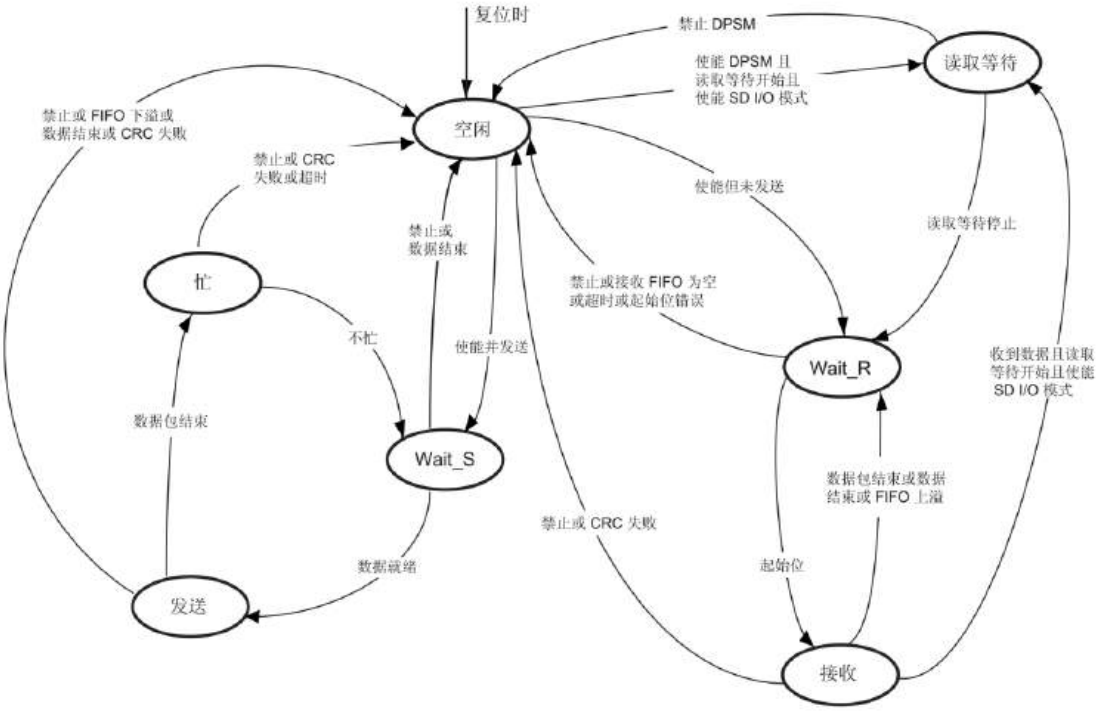


图 35-17 DPSM 状态机描述图

36.5.4 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。

36.5.5 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。

36.6 SDMMC 初始化结构体

HAL 库函数对 SDMMC 外设建立了三个初始化结构体，分别为 SDMMC 初始化结构体 SDMMC_InitTypeDef、SDMMC 命令初始化结构体 SDMMC_CmdInitTypeDef 和 SDMMC 数据初始化结构体 SDMMC_DataInitTypeDef。这些结构体成员用于设置 SDMMC 工作环境参数，并由 SDMMC 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 SDMMC 相应的寄存器，达到配置 SDMMC 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 stm32f4xx_ll_sdmmc.h 文件中，初始化库函数定义在 stm32f4xx_ll_sdmmc.c 文件中，编程时我们可以结合这两个文件内注释使用。

SDMMC 初始化结构体用于配置 SDMMC 基本工作环境，比如时钟分频、时钟沿、数据宽度等等。它被 HAL_SD_Init 函数使用。

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。

SDMMC 初始化结构体用于配置 SDMMC 基本工作环境，比如时钟分频、时钟沿、数据宽度等等。它被 HAL_SD_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
```

(下页继续)

(续上页)

```
    uint32_t BusWide;                    // 数据宽度
    uint32_t HardwareFlowControl;        // 硬件流控制
    uint32_t ClockDiv;                   // 时钟分频
} SDMMC_InitTypeDef;
```

各结构体成员的作用介绍如下：

(1) ClockEdge：主时钟 SDMMCCLK 产生 CLK 引脚时钟有效沿选择，可选上升沿或下降沿，它设定 SDMMC 时钟控制寄存器 (SDMMC_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 SDMMC_CLKCR 寄存器的 BYPASS 位。如果使能旁路，SDMMCCLK 直接驱动 CLK 线输出时钟；如果禁用，使用 SDMMC_CLKCR 寄存器的 CLKDIV 位值分频 SDMMCCLK，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 SDMMC_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 SDMMC_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 SDMMC_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 SDMMC_CLKCR 寄存器的 CLKDIV 位的值，设置 SDMMCCLK 与 CLK 线输出时钟分频系数：

CLK 线时钟频率 =SDIOCLK/([CLKDIV+2])。

36.7 SDMMC 命令初始化结构体

SDMMC 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。它被 SDMMC_SendCommand 函数使用。

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
    uint32_t CPSM;      // 命令路径状态机
} SDMMC_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 SDMMC 参数寄存器 (SDMMC_ARG) 的值。

(2) CmdIndex：命令号选择，它设定 SDMMC 命令寄存器 (SDMMC_CMD) 的 CMDINDEX 位的值。

(3) Response：响应类型，SDMMC 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。SDMMC 定义了四个 32 位的 SDMMC 响应寄存器 (SDMMC_RESPx,x=1..4)，短响应只用到 SDMMC_RESP1。

(4) WaitForInterrupt：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 SDMMC_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 SDMMC_CMD 寄存器的 CPSMEN 位的值。

36.8 SDMMC 数据初始化结构体

SDMMC 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。它被 SDMMC_DataConfig 函数使用。

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
    uint32_t TransferMode;    // 数据传输模式
    uint32_t DPSM;            // 数据路径状态机
} SDMMC_DataInitTypeDef;
```

各结构体成员介绍如下：

(1) 设置数据传输以卡总线时钟周期表示的超时周期，它设定 SDMMC 数据定时器寄存器 (SDMMC_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 SDMMC 数据长度寄存器 (SDMMC_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 SDMMC 数据控制寄存器 (SDMMC_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 SDMMC_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 SDMMC_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 SDMMC_DCTRL 寄存器的 DTEN 位的值。要实现数据传输都必须使能 SDMMC_DPSM。

36.9 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。类似与 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。

36.9.1 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。

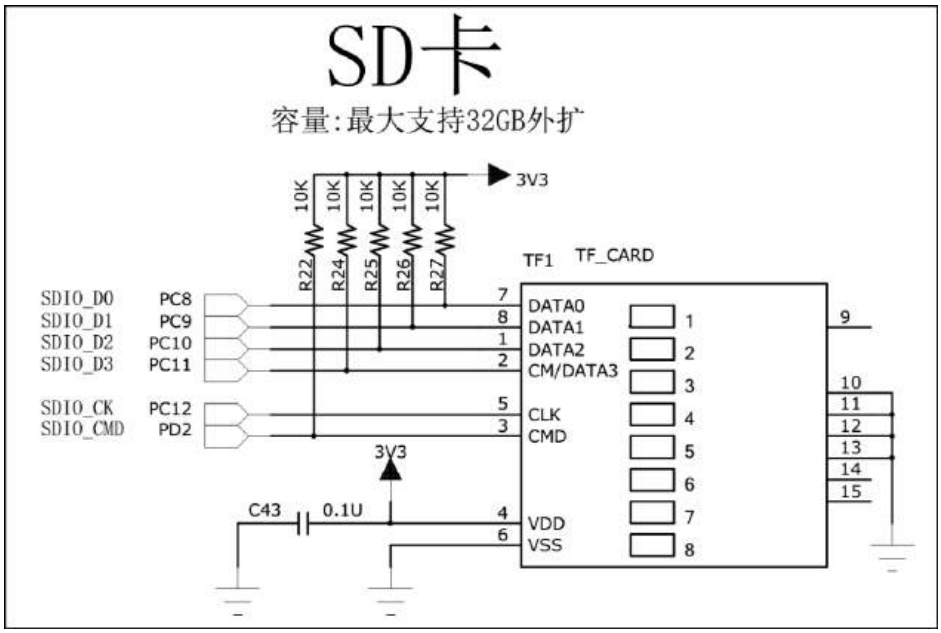


图 35-18 SD 卡硬件设计

36.9.2 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 SDMMC 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 SDMMC 外设；
- 配置 SDMMC 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 SDMMC、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。

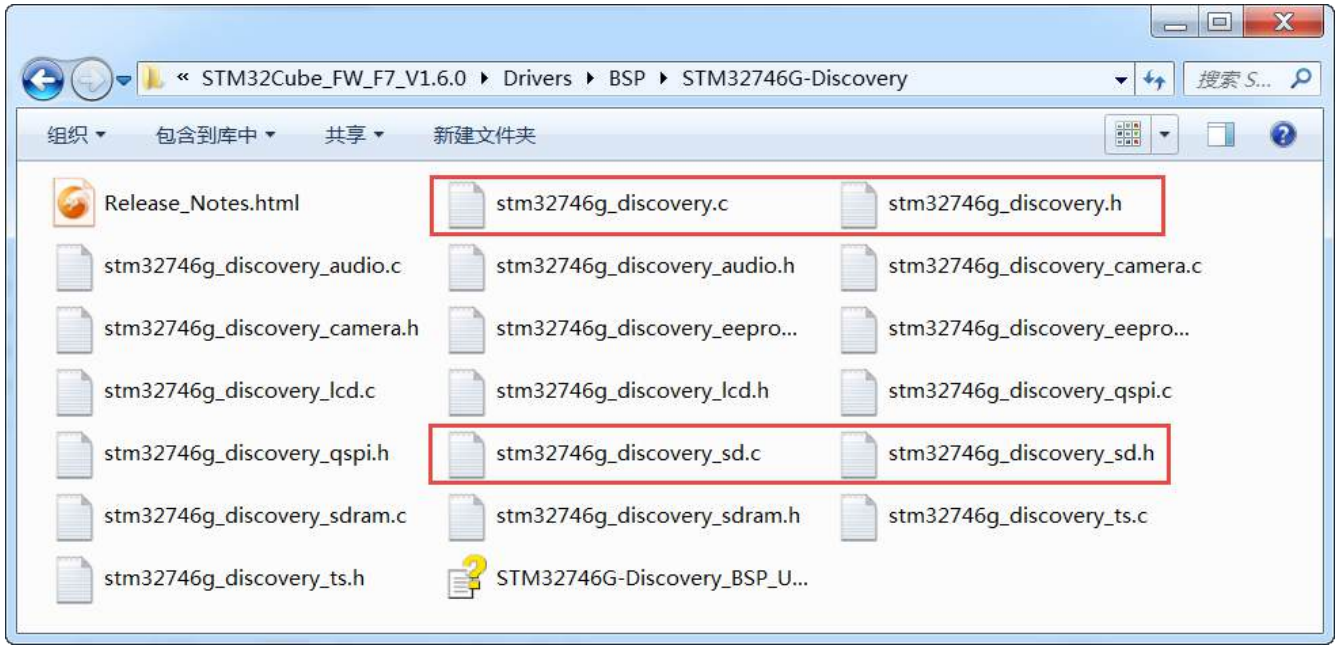


图 35-19 ST 官方实验板 SD 卡驱动文件

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。

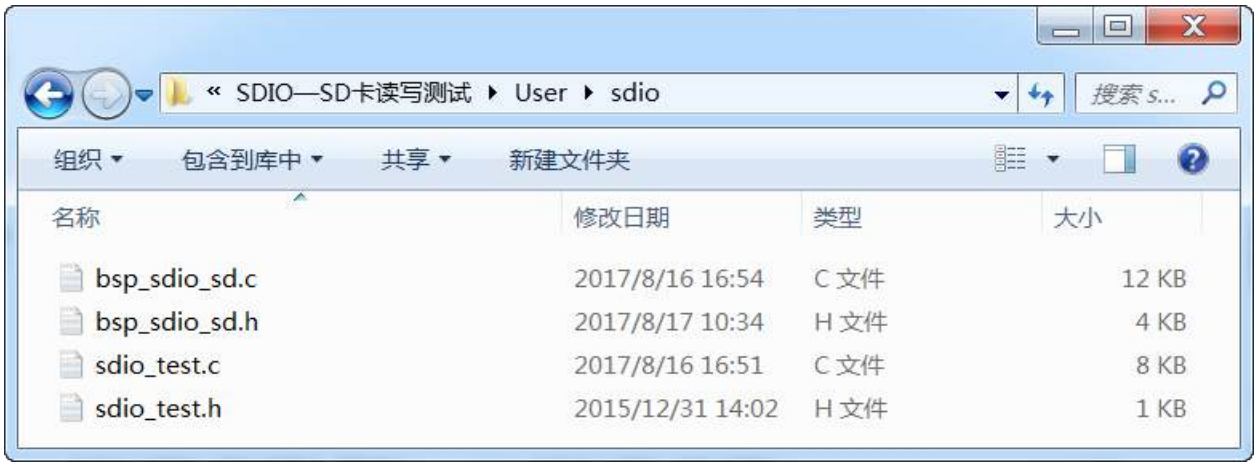


图 35-20 SD 卡驱动文件

36.9.2.1 GPIO 初始化和 DMA 配置

SDMMC 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 SDMMC 的类型。而 SDMMC 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 SDMMC 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 SDMMC 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。

GPIO 初始化和 DMA 配置这部分代码从 stm32746g_discovery_sd.c 和 stm32746g_discovery_sd.h 两个文件中移植而来。

36.9.2.1.1 DMA 及相关配置宏定义

列表 4: 代码清单 35-4 DMA 及相关配置宏定义

```
#define MSD_OK                                  ((uint8_t)0x00)
#define MSD_ERROR                               ((uint8_t)0x01)

#define SD_DATATIMEOUT                ((uint32_t)100000000)

/* DMA definitions for SD DMA transfer */
#define __DMAx_TxRx_CLK_ENABLE             __DMA2_CLK_ENABLE
#define SD_DMAx_Tx_CHANNEL                 DMA_CHANNEL_4
#define SD_DMAx_Rx_CHANNEL                 DMA_CHANNEL_4
#define SD_DMAx_Tx_STREAM                  DMA2_Stream6
#define SD_DMAx_Rx_STREAM                  DMA2_Stream3
#define SD_DMAx_Tx_IRQn                    DMA2_Stream6_IRQn
#define SD_DMAx_Rx_IRQn                    DMA2_Stream3_IRQn
#define SD_DMAx_Tx_IRQHandler              DMA2_Stream6_IRQHandler
#define SD_DMAx_Rx_IRQHandler              DMA2_Stream3_IRQHandler
```

使用宏定义编程对程序在同系列而不同型号主控芯片移植起到很好的帮助，同时简化程序代码。数据 FIFO 起始地址可用于 DMA 传输地址；SDIOCLK 在卡识别模式和数据传输模式下一般是不

同的，使用不同分频系数控制。SDMMC 使用 DMA2 外设，可选择 stream3 和 stream6。

36.9.2.1.2 SDMMC 底层驱动初始化

列表 5: SDMMC 底层驱动初始化

```
void BSP_SD_MspInit(SD_HandleTypeDef *hsd, void *Params)
{
    static DMA_HandleTypeDef dma_rx_handle;
    static DMA_HandleTypeDef dma_tx_handle;
    GPIO_InitTypeDef gpio_init_structure;

    /* 使能 SDMMC 时钟 */
    __HAL_RCC_SDMMC1_CLK_ENABLE();

    /* 使能 DMA2 时钟 */
    __DMAx_TxRx_CLK_ENABLE();

    /* 使能 GPIOs 时钟 */
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();

    /* 配置 GPIO 复用推挽、上拉、高速模式 */
    gpio_init_structure.Mode      = GPIO_MODE_AF_PP;
    gpio_init_structure.Pull      = GPIO_PULLUP;
    gpio_init_structure.Speed     = GPIO_SPEED_HIGH;
    gpio_init_structure.Alternate = GPIO_AF12_SDMMC1;
    /* GPIOC 配置 */
    gpio_init_structure.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10 | GPIO_
↪PIN_11 | GPIO_PIN_12;
    HAL_GPIO_Init(GPIOC, &gpio_init_structure);
```

(下页继续)

(续上页)

```
    /* GPIOD 配置 */
    gpio_init_structure.Pin = GPIO_PIN_2;
    HAL_GPIO_Init(GPIOD, &gpio_init_structure);

    /* SDMMC 中断配置 */
    HAL_NVIC_SetPriority(SDMMC1_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(SDMMC1_IRQn);

    /* 配置 DMA 接收参数 */
    dma_rx_handle.Init.Channel             = SD_DMAx_Rx_CHANNEL;
    dma_rx_handle.Init.Direction           = DMA_PERIPH_TO_MEMORY;
    dma_rx_handle.Init.PeriphInc           = DMA_PINC_DISABLE;
    dma_rx_handle.Init.MemInc              = DMA_MINC_ENABLE;
    dma_rx_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
    dma_rx_handle.Init.MemDataAlignment    = DMA_MDATAALIGN_WORD;
    dma_rx_handle.Init.Mode                = DMA_PFCTRL;
    dma_rx_handle.Init.Priority            = DMA_PRIORITY_VERY_HIGH;
    dma_rx_handle.Init.FIFOMode            = DMA_FIFOMODE_ENABLE;
    dma_rx_handle.Init.FIFOThreshold       = DMA_FIFO_THRESHOLD_FULL;
    dma_rx_handle.Init.MemBurst            = DMA_MBURST_INC4;
    dma_rx_handle.Init.PeriphBurst         = DMA_PBURST_INC4;

    dma_rx_handle.Instance = SD_DMAx_Rx_STREAM;

    /* 关联 DMA 句柄 */
    __HAL_LINKDMA(hsd, hdmarx, dma_rx_handle);

    /* 初始化传输数据流为默认值 */
    HAL_DMA_DeInit(&dma_rx_handle);

    /* 配置 DMA 接收数据流 */
    HAL_DMA_Init(&dma_rx_handle);
```

(下页继续)

(续上页)

```
/* 配置 DMA 发送参数 */
dma_tx_handle.Init.Channel             = SD_DMAx_Tx_CHANNEL;
dma_tx_handle.Init.Direction           = DMA_MEMORY_TO_PERIPH;
dma_tx_handle.Init.PeriphInc           = DMA_PINC_DISABLE;
dma_tx_handle.Init.MemInc              = DMA_MINC_ENABLE;
dma_tx_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
dma_tx_handle.Init.MemDataAlignment    = DMA_MDATAALIGN_WORD;
dma_tx_handle.Init.Mode                = DMA_PFCTRL;
dma_tx_handle.Init.Priority            = DMA_PRIORITY_VERY_HIGH;
dma_tx_handle.Init.FIFOMode            = DMA_FIFOMODE_ENABLE;
dma_tx_handle.Init.FIFOThreshold       = DMA_FIFO_THRESHOLD_FULL;
dma_tx_handle.Init.MemBurst            = DMA_MBURST_INC4;
dma_tx_handle.Init.PeriphBurst         = DMA_PBURST_INC4;

dma_tx_handle.Instance = SD_DMAx_Tx_STREAM;

/* 关联 DMA 句柄 */
__HAL_LINKDMA(hsd, hdmatx, dma_tx_handle);

/* 初始化传输数据流为默认值 */
HAL_DMA_DeInit(&dma_tx_handle);

/* 配置 DMA 发送数据流 */
HAL_DMA_Init(&dma_tx_handle);

/* 配置 DMA 接收传输完成中断 */
HAL_NVIC_SetPriority(SD_DMAx_Rx_IRQn, 6, 0);
HAL_NVIC_EnableIRQ(SD_DMAx_Rx_IRQn);

/* 配置 DMA 发送传输完成中断 */
HAL_NVIC_SetPriority(SD_DMAx_Tx_IRQn, 6, 0);
```

(下页继续)

(续上页)

```
    HAL_NVIC_EnableIRQ(SD_DMAx_Tx_IRQn);
}
```

由于 SDMMC 对应的 IO 引脚都是固定的，所以这里没有使用宏定义方式给出，直接使用 GPIO 引脚，该函数初始化引脚之后还使能了 SDMMC 和 DMA2 时钟。

接着分别配置 DMA 的 SDMMC 发送和接收数据流参数，关联 DMA 句柄，初始化 DMA 发送和接收数据流。对于 DMA 相关配置可以参考 DMA 章节内容。

36.9.2.2 相关类型定义

打开 bsp_sdio_sd.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 SD_ERROR、SDTransferState 和 SDCardState 三个。SD_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 SDMMC 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、SDMMC 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。SDMMC 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。

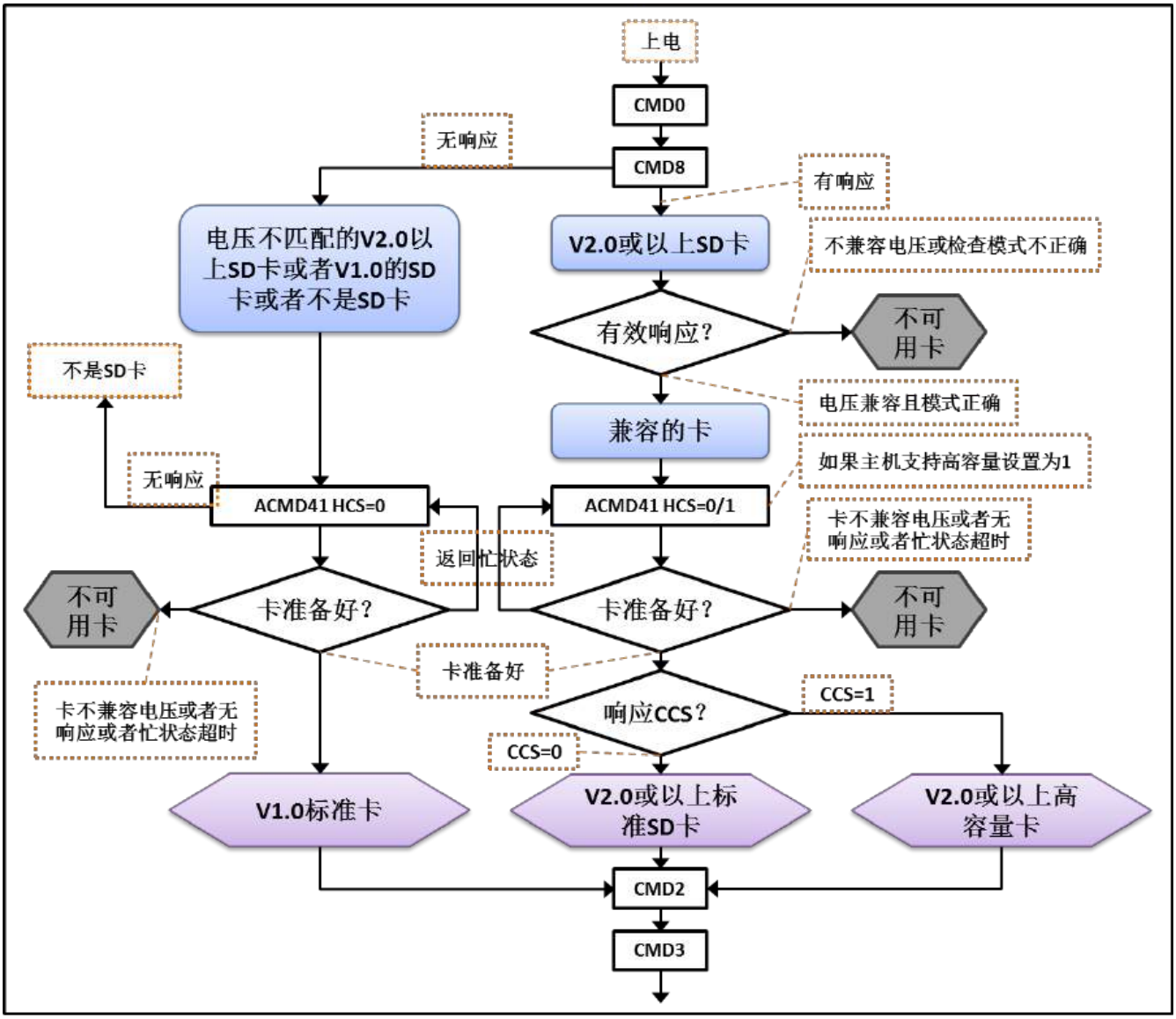
在 bsp_sdio_sd.c 文件也有部分宏定义，这部分宏定义只能在该文件中使用。这部分宏定义包括命令超时时间定义、OCR 寄存器位掩码、R6 响应位掩码等等，这些定义更多是为提取特定响应位内容而设计的掩码。

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_sd.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。

36.9.2.3 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。



36.9.2.3.1 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 sd_state = MSD_OK;

    /* 定义 SDMMC 句柄 */
    uSdHandle.Instance = SDMMC1;

    uSdHandle.Init.ClockEdge           = SDMMC_CLOCK_EDGE_RISING;
    uSdHandle.Init.ClockBypass         = SDMMC_CLOCK_BYPASS_DISABLE;
    uSdHandle.Init.ClockPowerSave      = SDMMC_CLOCK_POWER_SAVE_DISABLE;
    uSdHandle.Init.BusWide             = SDMMC_BUS_WIDE_1B;
    uSdHandle.Init.HardwareFlowControl =
    SDMMC_HARDWARE_FLOW_CONTROL_DISABLE;
    uSdHandle.Init.ClockDiv            = SDMMC_TRANSFER_CLK_DIV;

    /* 初始化 SD 底层驱动 */
    BSP_SD_MspInit(&uSdHandle, NULL);
    /* HAL SD 初始化 */
    if (HAL_SD_Init(&uSdHandle, &uSdCardInfo) != SD_OK) {
        sd_state = MSD_ERROR;
    }

    /* 配置 SD 总线位宽 */
    if (sd_state == MSD_OK) {
        /* 配置为 4bit 模式 */
        if (HAL_SD_WideBusOperation_Config(&uSdHandle,
            SDMMC_BUS_WIDE_4B) != SD_OK) {
            sd_state = MSD_ERROR;
        } else {
            sd_state = MSD_OK;
        }
    }
    return  sd_state;
}
```

该函数的部分执行流程如下：

(1) 配置 SD 外设参数，初始化 SD 外设。

(2) 执行 BSP_SD_MspInit 函数，其功能是对底层 SDMMC 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) HAL_SD_Init 函数用于初始化 SDMMC 外设接口，识别 SD 卡，流程包括初始化卡上外设接口的默认配置，识别卡工作电压，初始化当前的 SD 卡并将其置于空闲状态，读取 CSD/CID 寄存器获取 SD 卡信息，选择卡，配置 SDMMC 外设接口。

(4) 配置 SD 接口位宽为 4bit 用于数据传输。

36.9.2.3.2 SD_POWERON 函数

列表 7: 代码清单 35-7 SD_POWERON 函数

```
static HAL_SD_ErrorTypedef SD_PowerON(SD_HandleTypeDef *hsd)
{
    SDMMC_CmdInitTypeDef sdmmc_cmdinitstructure;
    __IO HAL_SD_ErrorTypedef errorstate = SD_OK;
    uint32_t response = 0, count = 0, validvoltage = 0;
    uint32_t sdtype = SD_STD_CAPACITY;

    /* Power ON Sequence -----------------------------------------------*/
    /* Disable SDMMC Clock */
    __HAL_SD_SDMMC_DISABLE(hsd);

    /* Set Power State to ON */
    SDMMC_PowerState_ON(hsd->Instance);

    /* 1ms: required power up waiting time before starting the SD␣
↪initialization sequence */
    HAL_Delay(1);
```

(下页继续)

(续上页)

```
/* Enable SDMMC Clock */
__HAL_SD_SDMMC_ENABLE(hsd);

/* CMD0: GO_IDLE_STATE ---------------------------------------------*/
/* No CMD response required */
sdmmc_cmdinitstructure.Argument         = 0;
sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_GO_IDLE_STATE;
sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_NO;
sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

/* Check for error conditions */
errorstate = SD_CmdError(hsd);

if (errorstate != SD_OK) {
    /* CMD Response Timeout (wait for CMDSENT flag) */
    return errorstate;
}

/* CMD8: SEND_IF_COND ----------------------------------------------*/
/* Send CMD8 to verify SD card interface operating condition */
/* Argument: - [31:12]: Reserved (shall be set to '0')
- [11:8]: Supply Voltage (VHS) 0x1 (Range: 2.7-3.6 V)
- [7:0]: Check Pattern (recommended 0xAA) */
/* CMD Response: R7 */
sdmmc_cmdinitstructure.Argument         = SD_CHECK_PATTERN;
sdmmc_cmdinitstructure.CmdIndex         = SD_SDMMC_SEND_IF_COND;
sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_SHORT;
SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);
```

(下页继续)

(续上页)

```
    /* Check for error conditions */
    errorstate = SD_CmdResp7Error(hsd);

    if (errorstate == SD_OK) {
        /* SD Card 2.0 */
        hsd->CardType = STD_CAPACITY_SD_CARD_V2_0;
        sdtype        = SD_HIGH_CAPACITY;
    }

    /* Send CMD55 */
    sdmmc_cmdinitstructure.Argument         = 0;
    sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_APP_CMD;
    SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

    /* Check for error conditions */
    errorstate = SD_CmdResp1Error(hsd, SD_CMD_APP_CMD);

    /* If errorstate is Command Timeout, it is a MMC card */
    /* If errorstate is SD_OK it is a SD card: SD card 2.0 (voltage range␣
↪mismatch)  or SD card 1.x */
    if (errorstate == SD_OK) {
        /* SD CARD */
        /* Send ACMD41 SD_APP_OP_COND with Argument 0x80100000 */
        while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL)) {

            /* SEND CMD55 APP_CMD with RCA as 0 */
            sdmmc_cmdinitstructure.Argument         = 0;
            sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_APP_CMD;
    sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_SHORT;
            sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
            sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
            SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);
```

(下页继续)

(续上页)

```
        /* Check for error conditions */
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_APP_CMD);

        if (errorstate != SD_OK) {
            return errorstate;
        }

        /* Send CMD41 */
    sdmmc_cmdinitstructure.Argument     = SD_VOLTAGE_WINDOW_SD | sdtype;
    sdmmc_cmdinitstructure.CmdIndex     = SD_CMD_SD_APP_OP_COND;
    sdmmc_cmdinitstructure.Response     = SDMMC_RESPONSE_SHORT;
        sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
        sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
        SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

        /* Check for error conditions */
        errorstate = SD_CmdResp3Error(hsd);

        if (errorstate != SD_OK) {
            return errorstate;
        }

        /* Get command response */
        response = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP1);

        /* Get operating voltage*/
        validvoltage = (((response >> 31) == 1) ? 1 : 0);

        count++;
    }
```

(下页继续)

(续上页)

```
        if (count >= SD_MAX_VOLT_TRIAL) {
            errorstate = SD_INVALID_VOLTRANGE;

            return errorstate;
        }

if ((response & SD_HIGH_CAPACITY) == SD_HIGH_CAPACITY) {
/* (response &= SD_HIGH_CAPACITY) */
            hsd->CardType = HIGH_CAPACITY_SD_CARD;
        }

    } /* else MMC Card */

    return errorstate;
}
```

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 SDMMC_PowerState_ON 函数控制 SDMMC 的电源状态，给 SDMMC 提供电源，并调用 __HAL_SD_SDMMC_DISABLE 库函数使能 SDMMC 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 SD_CmdError 函数检测错误即可。

SD_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 SDMMC_PowerState_ON 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 SD_CmdResp7Error 函数可获取得到 R7 响应结果，它是通过检测 SDMMC_STA 寄存器相关位完成的，并具有等待超时检测功能。如果 SD_CmdResp7Error 函数返回值为

SD_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。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，SD_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 SDMMC_GetResponse 函数才能获取响应的内容。实际上，在有响应时，SDMMC 外设会自动把响应存放在 SDMMC_RESPx 寄存器中，SDMMC_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_Power_ON 函数返回 HAL_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_Initialize_Cards 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。

36.9.2.3.3 SD_InitializeCards 函数

列表 8: 代码清单 35-8 SD_InitializeCards 函数

```
static HAL_SD_ErrorTypedef SD_Initialize_Cards(SD_HandleTypeDef *hsd)
{
    SDMMC_CmdInitTypeDef sdmmc_cmdinitstructure;
    HAL_SD_ErrorTypedef errorstate = SD_OK;
    uint16_t sd_rca = 1;

    if (SDMMC_GetPowerState(hsd->Instance) == 0) { /* Power off */
        errorstate = SD_REQUEST_NOT_APPLICABLE;

        return errorstate;
    }
```

(下页继续)

(续上页)

```
    if (hsd->CardType != SECURE_DIGITAL_IO_CARD) {
        /* Send CMD2 ALL_SEND_CID */
        sdmmc_cmdinitstructure.Argument         = 0;
        sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_ALL_SEND_CID;
        sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_LONG;
        sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
        sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
        SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

        /* Check for error conditions */
        errorstate = SD_CmdResp2Error(hsd);

        if (errorstate != SD_OK) {
            return errorstate;
        }

        /* Get Card identification number data */
        hsd->CID[0] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP1);
        hsd->CID[1] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP2);
        hsd->CID[2] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP3);
        hsd->CID[3] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP4);
    }

if ((hsd->CardType == STD_CAPACITY_SD_CARD_V1_1) ||
(hsd->CardType == STD_CAPACITY_SD_CARD_V2_0) ||\
(hsd->CardType == SECURE_DIGITAL_IO_COMBO_CARD) ||
(hsd->CardType == HIGH_CAPACITY_SD_CARD)) {
        /* Send CMD3 SET_REL_ADDR with argument 0 */
        /* SD Card publishes its RCA. */
        sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_SET_REL_ADDR;
        sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_SHORT;
        SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);
```

(下页继续)

(续上页)

```
        /* Check for error conditions */
        errorstate = SD_CmdResp6Error(hsd, SD_CMD_SET_REL_ADDR, &sd_rca);

        if (errorstate != SD_OK) {
            return errorstate;
        }
    }

    if (hsd->CardType != SECURE_DIGITAL_IO_CARD) {
        /* Get the SD card RCA */
        hsd->RCA = sd_rca;

        /* Send CMD9 SEND_CSD with argument as card's RCA */
    sdmmc_cmdinitstructure.Argument         = (uint32_t)(hsd->RCA << 16);
        sdmmc_cmdinitstructure.CmdIndex          = SD_CMD_SEND_CSD;
        sdmmc_cmdinitstructure.Response          = SDMMC_RESPONSE_LONG;
        SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

        /* Check for error conditions */
        errorstate = SD_CmdResp2Error(hsd);

        if (errorstate != SD_OK) {
            return errorstate;
        }

        /* Get Card Specific Data */
        hsd->CSD[0] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP1);
        hsd->CSD[1] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP2);
        hsd->CSD[2] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP3);
        hsd->CSD[3] = SDMMC_GetResponse(hsd->Instance, SDMMC_RESP4);
    }
```

(下页继续)

(续上页)

```
    /* All cards are initialized */
    return errorstate;
}
```

SD_InitializeCards 函数执行流程如下：

(1) 判断 SDMMC 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 SDMMC_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 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，SD_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，SD_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 SD_CmdResp6Error 函数之后返回到 SD_Initialize_Cards 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 SD_CmdResp6Error 函数获取得到的 rca。最后把响应内容存放在 CSD 数组中。

执行 SD_Initialize_Cards 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_Initialize_Cards 后会返回前面的 HAL_SD_Init 函数，执行接下来代码，以下是 HAL_SD_Init 函数的后续执行过程：

(1) 重新配置 SDMMC 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_Initialize_Cards 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 数组和 CSD 数组中，所以 HAL_SD_Get_CardInfo 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 HAL_SD_Get_CardInfo 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_Select_Deselect 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，函数有两个形参，一个指定句柄另外一个用于指定数据线宽度。在 HAL_SD_WideBusOperation_Config 函数中，调用了 SD_WideBus_Enable 函数使能使用宽数据线，然后传输 SDIO_InitTypeDef 类型变量并使用 SDMMC_Init 函数完成使用 4 根数据线配置。

至此，BSP_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 BSP_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。

36.9.2.4 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。

36.9.2.4.1 擦除函数

列表 9: 代码清单 35-9 SD_Erase 函数

```
HAL_SD_ErrorTypedef HAL_SD_Erase(SD_HandleTypeDef *hsd,
uint64_t startaddr, uint64_t endaddr)
{
    HAL_SD_ErrorTypedef errorstate = SD_OK;
    SDMMC_CmdInitTypeDef sdmmc_cmdinitstructure;
```

(下页继续)

(续上页)

```
    uint32_t delay         = 0;
    __IO uint32_t maxdelay = 0;
    uint8_t cardstate      = 0;

    /* Check if the card command class supports erase command */
    if (((hsd->CSD[1] >> 20) & SD_CCCC_ERASE) == 0) {
        errorstate = SD_REQUEST_NOT_APPLICABLE;

        return errorstate;
    }

    /* Get max delay value */
    maxdelay = 120000 / (((hsd->Instance->CLKCR) & 0xFF) + 2);

    if ((SDMMC_GetResponse(hsd->Instance, SDMMC_RESP1) &SD_CARD_LOCKED) ==␣
↪SD_CARD_LOCKED) {
        errorstate = SD_LOCK_UNLOCK_FAILED;

        return errorstate;
    }

    /* Get start and end block for high capacity cards */
    if (hsd->CardType == HIGH_CAPACITY_SD_CARD) {
        startaddr /= 512;
        endaddr   /= 512;
    }

    /* According to sd-card spec 1.0 ERASE_GROUP_START (CMD32) and erase_
↪group_end(CMD33) */
if ((hsd->CardType == STD_CAPACITY_SD_CARD_V1_1) ||
(hsd->CardType == STD_CAPACITY_SD_CARD_V2_0) ||\
    (hsd->CardType == HIGH_CAPACITY_SD_CARD)) {
```

(下页继续)

(续上页)

```
        /* Send CMD32 SD_ERASE_GRP_START with argument as addr  */
        sdmmc_cmdinitstructure.Argument         =(uint32_t)startaddr;
        sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_SD_ERASE_GRP_
↪START;
        sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_SHORT;
        sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
        sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
        SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

        /* Check for error conditions */
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_SD_ERASE_GRP_START);

        if (errorstate != SD_OK) {
            return errorstate;
        }

        /* Send CMD33 SD_ERASE_GRP_END with argument as addr  */
        sdmmc_cmdinitstructure.Argument         = (uint32_t)endaddr;
        sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_SD_ERASE_GRP_END;
        SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

        /* Check for error conditions */
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_SD_ERASE_GRP_END);

        if (errorstate != SD_OK) {
            return errorstate;
        }
    }

    /* Send CMD38 ERASE */
    sdmmc_cmdinitstructure.Argument         = 0;
    sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_ERASE;
```

(下页继续)

(续上页)

```
    SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

    /* Check for error conditions */
    errorstate = SD_CmdResp1Error(hsd, SD_CMD_ERASE);

    if (errorstate != SD_OK) {
        return errorstate;
    }

    for (; delay < maxdelay; delay++) {
    }

    /* Wait until the card is in programming state */
    errorstate = SD_IsCardProgramming(hsd, &cardstate);

    delay = SD_DATATIMEOUT;

while ((delay > 0) && (errorstate == SD_OK) && ((cardstate == SD_CARD_
↪PROGRAMMING) ||
        (cardstate == SD_CARD_RECEIVING))) {
        errorstate = SD_IsCardProgramming(hsd, &cardstate);
        delay--;
    }
    return errorstate;
}
```

HAL_SD_Erase 函数用于擦除 SD 卡指定地址范围内的数据。该函数接收三个参数，一个是 SD 外设的句柄，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。HAL_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) HAL_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 SD_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 HAL_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。

36.9.2.4.2 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

```
HAL_SD_ErrorTypedef HAL_SD_WriteBlocks_DMA(SD_HandleTypeDef *hsd, uint32_t
                        *pWriteBuffer, uint64_t WriteAddr, uint32_t␣
↪BlockSize,
                           uint32_t NumberOfBlocks)
{
    SDMMC_CmdInitTypeDef sdmmc_cmdinitstructure;
    SDMMC_DataInitTypeDef sdmmc_datainitstructure;
    HAL_SD_ErrorTypedef errorstate = SD_OK;

    /* Initialize data control register */
    hsd->Instance->DCTRL = 0;
```

(下页继续)

(续上页)

```
    /* Initialize handle flags */
    hsd->SdTransferCplt  = 0;
    hsd->DmaTransferCplt = 0;
    hsd->SdTransferErr   = SD_OK;

    /* Initialize SD Write operation */
    if (NumberOfBlocks > 1) {
        hsd->SdOperation = SD_WRITE_MULTIPLE_BLOCK;
    } else {
        hsd->SdOperation = SD_WRITE_SINGLE_BLOCK;
    }

    /* Enable transfer interrupts */
    __HAL_SD_SDMMC_ENABLE_IT(hsd, (SDMMC_IT_DCRCFAIL |\
                                   SDMMC_IT_DTIMEOUT |\
                                   SDMMC_IT_DATAEND  |\
                                   SDMMC_IT_TXUNDERR));

    /* Configure DMA user callbacks */
    hsd->hdmatx->XferCpltCallback  = SD_DMA_TxCplt;
    hsd->hdmatx->XferErrorCallback = SD_DMA_TxError;

    /* Enable the DMA Channel */
    HAL_DMA_Start_IT(hsd->hdmatx, (uint32_t)pWriteBuffer, (uint32_t)&hsd->
↪Instance->FIFO, (uint32_t)(BlockSize *  NumberOfBlocks)/4);

    /* Enable SDMMC DMA transfer */
    __HAL_SD_SDMMC_DMA_ENABLE(hsd);

    if (hsd->CardType == HIGH_CAPACITY_SD_CARD) {
        BlockSize = 512;
        WriteAddr /= 512;
```

(下页继续)

(续上页)

```
    }

    /* Set Block Size for Card */
    sdmmc_cmdinitstructure.Argument         = (uint32_t)BlockSize;
    sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_SET_BLOCKLEN;
    sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_SHORT;
    sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
    sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
    SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

    /* Check for error conditions */
    errorstate = SD_CmdResp1Error(hsd, SD_CMD_SET_BLOCKLEN);

    if (errorstate != SD_OK) {
        return errorstate;
    }

    /* Check number of blocks command */
    if (NumberOfBlocks <= 1) {
        /* Send CMD24 WRITE_SINGLE_BLOCK */
        sdmmc_cmdinitstructure.CmdIndex = SD_CMD_WRITE_SINGLE_BLOCK;
    } else {
        /* Send CMD25 WRITE_MULT_BLOCK with argument data address */
        sdmmc_cmdinitstructure.CmdIndex = SD_CMD_WRITE_MULT_BLOCK;
    }

    sdmmc_cmdinitstructure.Argument         = (uint32_t)WriteAddr;
    SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

    /* Check for error conditions */
    if (NumberOfBlocks > 1) {
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_WRITE_MULT_BLOCK);
```

(下页继续)

(续上页)

```
    } else {
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_WRITE_SINGLE_BLOCK);
    }

    if (errorstate != SD_OK) {
        return errorstate;
    }

    /* Configure the SD DPSM (Data Path State Machine) */
    sdmmc_datainitstructure.DataTimeOut    = SD_DATATIMEOUT;
    sdmmc_datainitstructure.DataLength     = BlockSize * NumberOfBlocks;
    sdmmc_datainitstructure.DataBlockSize = SDMMC_DATABLOCK_SIZE_512B;
    sdmmc_datainitstructure.TransferDir    = SDMMC_TRANSFER_DIR_TO_CARD;
    sdmmc_datainitstructure.TransferMode   = SDMMC_TRANSFER_MODE_BLOCK;
    sdmmc_datainitstructure.DPSM           = SDMMC_DPSM_ENABLE;
    SDMMC_DataConfig(hsd->Instance, &sdmmc_datainitstructure);

    hsd->SdTransferErr = errorstate;

    return errorstate;
}
```

HAL_SD_WriteBlocks_DMA 函数用于向指定的目标地址写入块的数据，它有五个形参，分别为指 SDMMC 外设句柄，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，块数量。块大小一般都设置为 512 字节。HAL_SD_WriteBlocks_DMA 写入函数的执行流程如下：

(1) HAL_SD_WriteBlocks_DMA 函数开始将 SDMMC 数据控制寄存器 (SDMMC_DCTRL) 清理，复位之前的传输设置。

(2) 来调用 __HAL_SD_SDMMC_ENABLE_IT 函数使能相关中断，包括数据 CRC 失败中断、数据超时中断、数据结束中断等等。

(3) 调用 HAL_DMA_Start_IT 函数，配置使能 SDMMC 数据向 SD 卡的数据传输的 DMA 请求。为使 SDMMC 发送 DMA 请求，需要调用 __HAL_SD_SDMMC_DMA_ENABLE 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。

(4) 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 BlockSize 长度字节的块；对于 SDHC 卡，写入 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。

(5) 利用 SDMMC_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 SDMMC_DataConfig 函数完成数据传输环境配置。

执行完以上代码后，SDMMC 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。

36.9.2.4.3 写入操作等待函数

HAL_SD_CheckWriteOperation 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，HAL_SD_CheckWriteOperation 函数不仅使用于单块写入函数也适用于多块写入函数。

列表 11: 代码清单 35-11 SD_WaitWriteOperation 函数

```
HAL_SD_ErrorTypedef HAL_SD_CheckWriteOperation
(SD_HandleTypeDef *hsd, uint32_t Timeout)
{
    HAL_SD_ErrorTypedef errorstate = SD_OK;
    uint32_t timeout = Timeout;
    uint32_t tmp1, tmp2;
    HAL_SD_ErrorTypedef tmp3;

    /* Wait for DMA/SD transfer end or SD error variables to be in SD␣
↪handle */
    tmp1 = hsd->DmaTransferCplt;
    tmp2 = hsd->SdTransferCplt;
    tmp3 = (HAL_SD_ErrorTypedef)hsd->SdTransferErr;
```

(下页继续)

(续上页)

```
    while (((tmp1 & tmp2) == 0) && (tmp3 == SD_OK) && (timeout > 0)) {
        tmp1 = hsd->DmaTransferCplt;
        tmp2 = hsd->SdTransferCplt;
        tmp3 = (HAL_SD_ErrorTypedef)hsd->SdTransferErr;
        timeout--;
    }

    timeout = Timeout;

    /* Wait until the Tx transfer is no longer active */
    while ((__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_FLAG_TXACT)) && (timeout >␣
↪0)) {
        timeout--;
    }

    /* Send stop command in multiblock write */
    if (hsd->SdOperation == SD_WRITE_MULTIPLE_BLOCK) {
        errorstate = HAL_SD_StopTransfer(hsd);
    }

    if ((timeout == 0) && (errorstate == SD_OK)) {
        errorstate = SD_DATA_TIMEOUT;
    }

    /* Clear all the static flags */
    __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_STATIC_FLAGS);

    /* Return error state */
    if (hsd->SdTransferErr != SD_OK) {
        return (HAL_SD_ErrorTypedef)(hsd->SdTransferErr);
    }
```

(下页继续)

(续上页)

```
    /* Wait until write is complete */
    while (HAL_SD_GetStatus(hsd) != SD_TRANSFER_OK) {
    }

    return errorstate;
}
```

该函数开始等待当前块数据正确传输完成，并添加了超时检测功能。然后不停监测 SDMMC_STA 寄存器的 TXACT 位，以等待数据写入完成。对于多块数据写入操作需要调用 HAL_SD_StopTransfer 函数停止数据传输，而单块写入则不需要。HAL_SD_StopTransfer 函数实际是向 SD 卡发送 CMD12，该命令专门用于停止数据传输，SD 卡系统保证在主机发送 CMD12 之后整块传输完后才停止数据传输。HAL_SD_CheckWriteOperation 函数最后是清除相关标志位并返回错误。

36.9.2.4.4 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。这里这介绍单块读操作函数，多块读操作类似理解即可。

列表 12: 代码清单 35-12 SD_ReadBlock 函数

```
HAL_SD_ErrorTypedef HAL_SD_ReadBlocks_DMA(SD_HandleTypeDef *hsd,
            uint32_t *pReadBuffer, uint64_t ReadAddr,
            uint32_t BlockSize, uint32_t NumberOfBlocks)
{
    SDMMC_CmdInitTypeDef sdmmc_cmdinitstructure;
    SDMMC_DataInitTypeDef sdmmc_datainitstructure;
    HAL_SD_ErrorTypedef errorstate = SD_OK;

    /* Initialize data control register */
```

(下页继续)

(续上页)

```
    hsd->Instance->DCTRL = 0;

    /* Initialize handle flags */
    hsd->SdTransferCplt  = 0;
    hsd->DmaTransferCplt = 0;
    hsd->SdTransferErr   = SD_OK;

    /* Initialize SD Read operation */
    if (NumberOfBlocks > 1) {
        hsd->SdOperation = SD_READ_MULTIPLE_BLOCK;
    } else {
        hsd->SdOperation = SD_READ_SINGLE_BLOCK;
    }

    /* Enable transfer interrupts */
    __HAL_SD_SDMMC_ENABLE_IT(hsd, (SDMMC_IT_DCRCFAIL |\
                                   SDMMC_IT_DTIMEOUT |\
                                   SDMMC_IT_DATAEND  |\
                                   SDMMC_IT_RXOVERR));

    /* Enable SDMMC DMA transfer */
    __HAL_SD_SDMMC_DMA_ENABLE(hsd);

    /* Configure DMA user callbacks */
    hsd->hdmarx->XferCpltCallback  = SD_DMA_RxCplt;
    hsd->hdmarx->XferErrorCallback = SD_DMA_RxError;

    /* Enable the DMA Channel */
    HAL_DMA_Start_IT(hsd->hdmarx, (uint32_t)&hsd->Instance->FIFO,
            (uint32_t)pReadBuffer,
        (uint32_t)(BlockSize * NumberOfBlocks)/4);
    if (hsd->CardType == HIGH_CAPACITY_SD_CARD) {
```

(下页继续)

(续上页)

```
        BlockSize = 512;
        ReadAddr /= 512;
    }

    /* Set Block Size for Card */
    sdmmc_cmdinitstructure.Argument         = (uint32_t)BlockSize;
    sdmmc_cmdinitstructure.CmdIndex         = SD_CMD_SET_BLOCKLEN;
    sdmmc_cmdinitstructure.Response         = SDMMC_RESPONSE_SHORT;
    sdmmc_cmdinitstructure.WaitForInterrupt = SDMMC_WAIT_NO;
    sdmmc_cmdinitstructure.CPSM             = SDMMC_CPSM_ENABLE;
    SDMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

    /* Check for error conditions */
    errorstate = SD_CmdResp1Error(hsd, SD_CMD_SET_BLOCKLEN);

    if (errorstate != SD_OK) {
        return errorstate;
    }

    /* Configure the SD DPSM (Data Path State Machine) */
    sdmmc_datainitstructure.DataTimeOut   = SD_DATATIMEOUT;
    sdmmc_datainitstructure.DataLength    = BlockSize * NumberOfBlocks;
    sdmmc_datainitstructure.DataBlockSize = SDMMC_DATABLOCK_SIZE_512B;
    sdmmc_datainitstructure.TransferDir   = SDMMC_TRANSFER_DIR_TO_SDMMC;
    sdmmc_datainitstructure.TransferMode  = SDMMC_TRANSFER_MODE_BLOCK;
    sdmmc_datainitstructure.DPSM          = SDMMC_DPSM_ENABLE;
    SDMMC_DataConfig(hsd->Instance, &sdmmc_datainitstructure);

    /* Check number of blocks command */
    if (NumberOfBlocks > 1) {
        /* Send CMD18 READ_MULT_BLOCK with argument data address */
        sdmmc_cmdinitstructure.CmdIndex = SD_CMD_READ_MULT_BLOCK;
```

(下页继续)

(续上页)

```
    } else {
        /* Send CMD17 READ_SINGLE_BLOCK */
        sdmmc_cmdinitstructure.CmdIndex = SD_CMD_READ_SINGLE_BLOCK;
    }

    sdmmc_cmdinitstructure.Argument         = (uint32_t)ReadAddr;
    SMMC_SendCommand(hsd->Instance, &sdmmc_cmdinitstructure);

    /* Check for error conditions */
    if (NumberOfBlocks > 1) {
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_READ_MULT_BLOCK);
    } else {
        errorstate = SD_CmdResp1Error(hsd, SD_CMD_READ_SINGLE_BLOCK);
    }

    /* Update the SD transfer error in SD handle */
    hsd->SdTransferErr = errorstate;

    return errorstate;
}
```

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。HAL_SD_ReadBlocks_DMA 函数有五个形参，分别为指 SDMMC 外设句柄，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，块数量。HAL_SD_ReadBlocks_DMA 函数执行流程如下：

(1) 将 SDMMC 外设的数据控制寄存器 (SDMMC_DCTRL) 清理，复位之前的传输设置。

(2) 调用 __HAL_SD_SDMMC_ENABLE_IT 函数使能相关中断，包括数据 CRC 失败中断、数据超时中断、数据结束中断等等。然后调用 HAL_DMA_Start_IT 函数，配置使能 SDMMC 从 SD 卡的读取数据的 DMA 请求，该函数可以参考前面章节。为使 SDMMC 发送 DMA 请求，需要调用 __HAL_SD_SDMMC_DMA_ENABLE 函数使能。对于高容量的 SD 卡要求块大小必须为 512 字节，程序员有责任保证目标读取地址与块大小的字节对齐问题。

(3) 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，读取 512 字节的块。

(4) 利用 SDMMC_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 SDMMC_DataConfig 函数完成数据传输环境配置。

(5) 最后控制器向 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控制器数据 FIFO 内，并自动生成 DMA 传输请求。

36.9.2.4.5 读取操作等待函数

HAL_SD_CheckReadOperation 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。HAL_SD_CheckReadOperation 函数也是适用于单块读取函数和多块读取函数的。

列表 13：代码清单 35-13 HAL_SD_CheckReadOperation 函数

```
HAL_SD_ErrorTypedef HAL_SD_CheckReadOperation(SD_HandleTypeDef *hsd,
uint32_t Timeout)
{
    HAL_SD_ErrorTypedef errorstate = SD_OK;
    uint32_t timeout = Timeout;
    uint32_t tmp1, tmp2;
    HAL_SD_ErrorTypedef tmp3;

    /* Wait for DMA/SD transfer end or SD error variables to be in SD␣
↪handle */
    tmp1 = hsd->DmaTransferCplt;
    tmp2 = hsd->SdTransferCplt;
    tmp3 = (HAL_SD_ErrorTypedef)hsd->SdTransferErr;

    while (((tmp1 & tmp2) == 0) && (tmp3 == SD_OK) && (timeout > 0)) {
```

(下页继续)

(续上页)

```
        tmp1 = hsd->DmaTransferCplt;
        tmp2 = hsd->SdTransferCplt;
        tmp3 = (HAL_SD_ErrorTypedef)hsd->SdTransferErr;
        timeout--;
    }

    timeout = Timeout;

    /* Wait until the Rx transfer is no longer active */
    while ((__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_FLAG_RXACT)) && (timeout >␣
↪0)) {
        timeout--;
    }

    /* Send stop command in multiblock read */
    if (hsd->SdOperation == SD_READ_MULTIPLE_BLOCK) {
        errorstate = HAL_SD_StopTransfer(hsd);
    }
    if ((timeout == 0) && (errorstate == SD_OK)) {
        errorstate = SD_DATA_TIMEOUT;
    }

    /* Clear all the static flags */
    __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_STATIC_FLAGS);

    /* Return error state */
    if (hsd->SdTransferErr != SD_OK) {
        return (HAL_SD_ErrorTypedef)(hsd->SdTransferErr);
    }

    return errorstate;
}
```

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。

36.9.2.5 SDMMC 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。如果是使用 DMA 传输，也会使能 DMA 相关中断。为简化代码，加之 SDMMC 中断服务函数内容一般不会修改，将中断服务函数放在 bsp_sdio_sd.c 文件中，而不是放在常用于存放中断服务函数的 stm32f4xx_it.c 文件。

列表 14: 代码清单 35-14 SDMMC 中断服务函数

```
void HAL_SD_IRQHandler(SD_HandleTypeDef *hsd)
{
    /* Check for SDMMC interrupt flags */
    if (__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_IT_DATAEND)) {
        __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_IT_DATAEND);

        /* SD transfer is complete */
        hsd->SdTransferCplt = 1;

        /* No transfer error */
        hsd->SdTransferErr  = SD_OK;

        HAL_SD_XferCpltCallback(hsd);
    } else if (__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_IT_DCRCFAIL)) {
        __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_FLAG_DCRCFAIL);

        hsd->SdTransferErr = SD_DATA_CRC_FAIL;

        HAL_SD_XferErrorCallback(hsd);
```

(下页继续)

(续上页)

```
    } else if (__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_IT_DTIMEOUT)) {
        __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_FLAG_DTIMEOUT);

        hsd->SdTransferErr = SD_DATA_TIMEOUT;

        HAL_SD_XferErrorCallback(hsd);
    } else if (__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_IT_RXOVERR)) {
        __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_FLAG_RXOVERR);

        hsd->SdTransferErr = SD_RX_OVERRUN;

        HAL_SD_XferErrorCallback(hsd);
    } else if (__HAL_SD_SDMMC_GET_FLAG(hsd, SDMMC_IT_TXUNDERR)) {
        __HAL_SD_SDMMC_CLEAR_FLAG(hsd, SDMMC_FLAG_TXUNDERR);

        hsd->SdTransferErr = SD_TX_UNDERRUN;

        HAL_SD_XferErrorCallback(hsd);
    } else {
        /* No error flag set */
    }

    /* Disable all SDMMC peripheral interrupt sources */
    __HAL_SD_SDMMC_DISABLE_IT(hsd, SDMMC_IT_DCRCFAIL | SDMMC_IT_DTIMEOUT
        | SDMMC_IT_DATAEND  | SDMMC_IT_TXFIFOHE | SDMMC_IT_RXFIFOHF |
SDMMC_IT_TXUNDERR |SDMMC_IT_RXOVERR);
}
```

SDMMC 中断服务函数 HAL_SD_IRQHandler 通过多个 if 判断语句分辨中断源，并对传输错误标志变量 TransferError 赋值以指示当前传输状态，每个状态都有一个中断回调函数供客户编写用户代码。最后禁用 SDMMC 中断。

列表 15: DMA 请求中断

```
/**
* @brief  Handles SD DMA Tx transfer interrupt request.
* @retval None
*/
void BSP_SD_DMA_Tx_IRQHandler(void)
{
    HAL_DMA_IRQHandler(uSdHandle.hdmatx);
}

/**
* @brief  Handles SD DMA Rx transfer interrupt request.
* @retval None
*/
void BSP_SD_DMA_Rx_IRQHandler(void)
{
    HAL_DMA_IRQHandler(uSdHandle.hdmarx);
}
```

BSP_SD_DMA_Tx_IRQHandler 和 BSP_SD_DMA_Rx_IRQHandler 函数是 DMA 传输中断服务函数，它直接调用 HAL_DMA_IRQHandler 函数执行。

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，很多时候这些函数已经足够我们使用了。接下来我们就编写一些简单的测试程序验证移植的正确性。

36.9.2.6 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。

36.9.2.6.1 SD 卡测试函数

列表 16: 代码清单 35-15 SD_Test

```
void SD_Test(void)
{

    LED_BLUE;
/*------------------------------ SD 初始化 ------------------------- */
/* 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 */
    if (BSP_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");

        LED_BLUE;
        /* 擦除测试 */
        SD_EraseTest();

        LED_BLUE;
        /*single block 读写测试 */
        SD_SingleBlockTest();

        LED_BLUE;
        /*muti block 读写测试 */
        SD_MultiBlockTest();
    }
```

(下页继续)

(续上页)

```
}
```

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 BSP_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_15*，如果初始化成功就可以进行数据操作测试。

36.9.2.6.2 SD 卡擦除测试

列表 17: 代码清单 35-16 SD_EraseTest

```
void SD_EraseTest(void)
{
    /*------------------- Block Erase ---------------------------------*/
    if (Status == SD_OK) {
        /* Erase NumberOfBlocks Blocks of WRITE_BL_LEN(512 Bytes) */
        Status = BSP_SD_Erase(0x00, (BLOCK_SIZE * NUMBER_OF_BLOCKS));
    }

if (Status == SD_OK) { Status = BSP_SD_ReadBlocks_DMA(
        Buffer_MultiBlock_Rx, 0x00, BLOCK_SIZE, NUMBER_OF_BLOCKS);
    }

    /* Check the correctness of erased blocks */
    if (Status == SD_OK) {
    EraseStatus = eBuffercmp(Buffer_MultiBlock_Rx, MULTI_BUFFER_SIZE/4);
    }

    if (EraseStatus == PASSED) {
        LED_GREEN;
        printf("SD 卡擦除测试成功! \n");
```

(下页继续)

(续上页)

```
    } else {
        LED_RED;
        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。

BSP_SD_Erase 函数用于擦除指定地址空间，源代码参考*代码清单 35_16*，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 BSP_SD_Erase 函数返回正确，表示擦除成功则执行数据块读取；如果 BSP_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 BSP_SD_ReadBlocks_DMA，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。

36.9.2.6.3 单块读写测试

列表 18: 代码清单 35-17 SD_SingleBlockTest 函数

```
void SD_SingleBlockTest(void)
{
    /*------------------- Block Read/Write ---------------------------*/
    /* Fill the buffer to send */
    Fill_Buffer(Buffer_Block_Tx, BLOCK_SIZE/4, 0);

    if (Status == SD_OK) {
```

(下页继续)

(续上页)

```
        /* Write block of 512 bytes on address 0 */
 Status = BSP_SD_WriteBlocks_DMA(Buffer_Block_Tx, 0x00, BLOCK_SIZE,1);
    }

    if (Status == SD_OK) {
        /* Read block of 512 bytes from address 0 */
 Status = BSP_SD_ReadBlocks_DMA(Buffer_Block_Rx, 0x00, BLOCK_SIZE,1);
    }

    /* Check the correctness of written data */
    if (Status == SD_OK) {
    TransferStatus1 = Buffercmp(Buffer_Block_Tx, Buffer_Block_Rx, BLOCK_
↪SIZE/4);
    }

    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。BSP_SD_WriteBlocks_DMA 函数和 BSP_SD_ReadBlocks_DMA 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。

36.9.2.6.4 主函数

列表 19: 代码清单 35-18 main 函数

```
int main(void)
{
    /* 配置系统时钟为 168 MHz */
    SystemClock_Config();
    /* 禁用 WiFi 模块 */
    WIFI_PDN_INIT();
    /* 初始化 RGB 彩灯 */
    LED_GPIO_Config();
    LED_BLUE;
    /* 初始化 USART1 配置模式为 115200 8-N-1 */
    UARTx_Config();
    /* 初始化独立按键 */
    Key_GPIO_Config();
    printf("\r\n欢迎使用野火  STM32 F407 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
↪");
```

(下页继续)

(续上页)

```
    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");

    while (1) {
        /* 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。

36.9.3 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 37 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。

37.1 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。

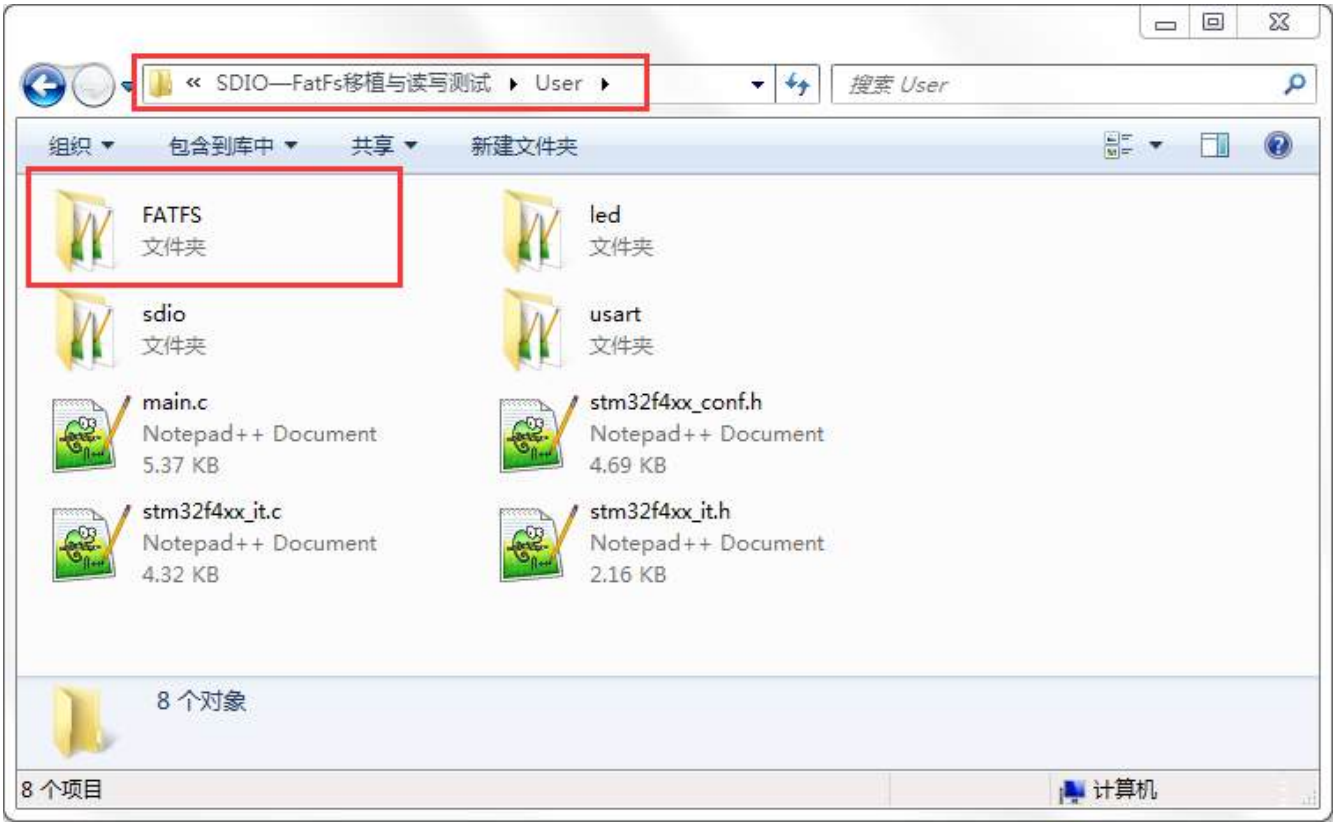
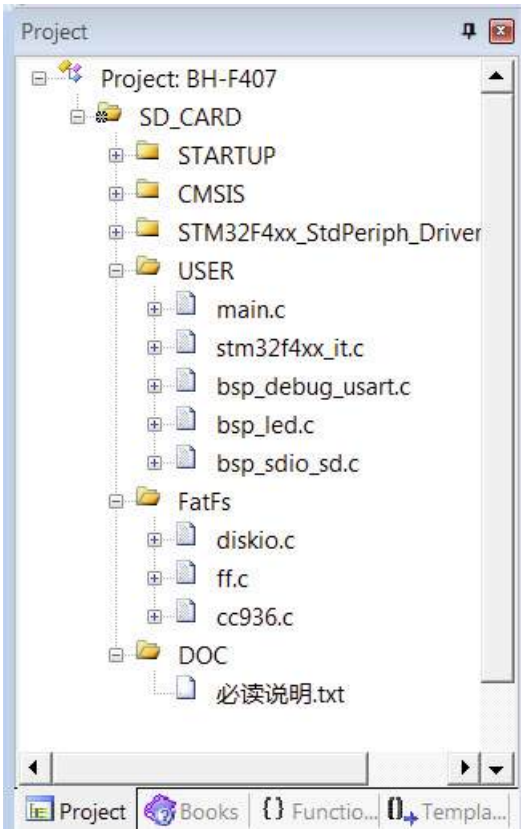
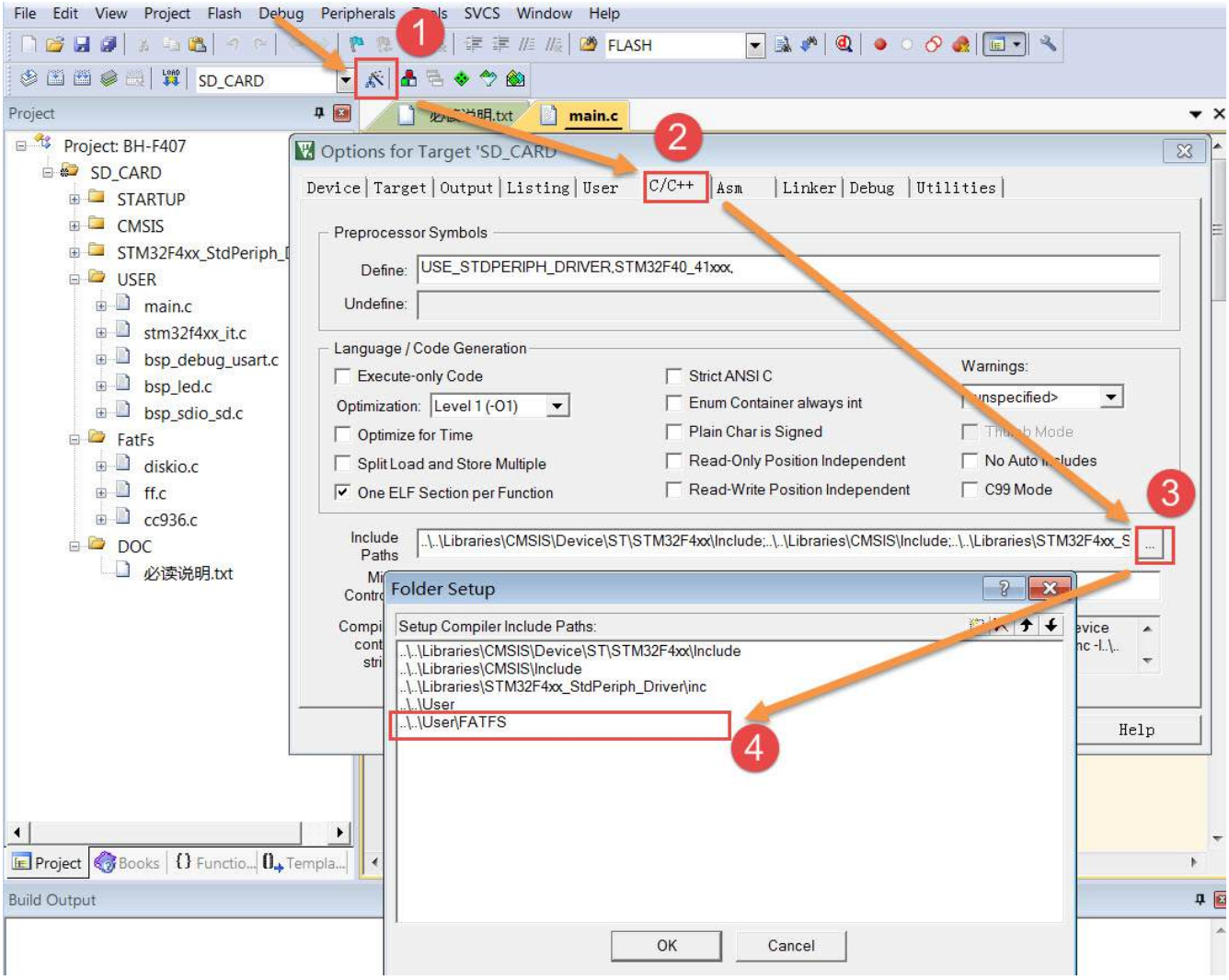


图 36-1 拷贝 FatFs 文件夹

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。

37.2 FatFs 接口函数

FatFs 文件系统与存储设备的连接函数在 sd_diskio.c 文件中，主要有 5 个函数需要我们编写的。

37.2.1 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。

37.2.2 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 */
```

(下页继续)

(续上页)

```
    if (BSP_SD_Init() == MSD_OK) {
        Stat &= ~STA_NOINIT;
    }

    return Stat;
}
```

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。

37.2.3 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;
        }
```

(下页继续)

(续上页)

```
        return (res);
    }
    if (BSP_SD_ReadBlocks_DMA((uint32_t*)buff,
                        (uint64_t) (sector * BLOCK_SIZE),
                        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。

37.2.4 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) {
        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。

37.2.5 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;
```

(下页继续)

(续上页)

```
    /* 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;

    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。

37.3 FatFs 功能测试

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。

37.3.1 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 */
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 F407 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。

37.3.2 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 168 MHz */
    SystemClock_Config();
    /* 禁用 WiFi 模块 */
    WIFI_PDN_INIT();
    /* 初始化 LED */
    LED_GPIO_Config();
    LED_BLUE;
    /* 初始化调试串口，一般为串口 1 */
    UARTx_Config();
    printf("****** 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
    res_sd = 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 (res_sd == 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 */
        res_sd=f_mkfs("0:",0,0);

        if (res_sd == FR_OK) {
            printf("》SD 卡已成功格式化文件系统。\r\n");
```

(下页继续)

(续上页)

```
            /* 格式化后，先取消挂载 */
            res_sd = f_mount(NULL,"0:",1);
            /* 重新挂载 */
            res_sd = f_mount(&fs,"0:",1);
        } else {
            LED_RED;
            printf("《《格式化失败。》》\r\n");
            while (1);
        }
    } else if (res_sd!=FR_OK) {
        printf("!! SD 卡挂载文件系统失败。(%d)\r\n",res_sd);
        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");
    res_sd = f_open(&fnew, "0:FatFs 读写测试文件.txt",FA_CREATE_ALWAYS | FA_
↪WRITE );
    if ( res_sd == 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_sd=f_write(&fnew,WriteBuffer,sizeof(WriteBuffer),&fnum);
        if (res_sd==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_sd);
```

(下页继续)

(续上页)

```
        }
        /* 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 ---------------------------*/
    printf("****** 即将进行文件读取测试... ******\r\n");
    res_sd = f_open(&fnew, "0:FatFs 读写测试文件.txt", FA_OPEN_EXISTING | FA_
↪READ);
    if (res_sd == FR_OK) {
        LED_GREEN;
        printf("》打开文件成功。\r\n");
        res_sd = f_read(&fnew, ReadBuffer, sizeof(ReadBuffer), &fnum);
        if (res_sd==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_sd);
        }
    } 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,"0:",1);
```

(下页继续)

(续上页)

```
    /* 操作完成，停机 */
    while (1) {
    }
}
```

首先，初始化系统时钟，调用 WIFI_PDN_INIT 函数禁用 WiFi 模块，接下来初始化 RGB 彩灯和调试串口，用来指示程序进程。

FatFs 的第一步工作是使用 FATFS_LinkDriver 函数创建盘符，然后就是使用 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 36_7* 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 36_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。

37.3.3 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 38 章　I2S—音频播放与录音输入

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》及《I2S BUS》。

若对 I2S 通讯协议不了解，可先阅读《I2S BUS》文档的内容学习。

关于音频编译码器 WM8978，请参考其规格书《WM8978_v4.5》来了解。

38.1 I2S 简介

Inter-IC Sount Bus(I2S) 是飞利浦半导体公司 (现为恩智浦半导体公司) 针对数字音频设备之间的音频数据传输而制定的一种总线标准。在飞利浦公司的 I2S 标准中，既规定了硬件接口规范，也规定了数字音频数据的格式。

38.1.1 数字音频技术

现实生活中的声音是通过一定介质传播的连续的波，它可以由周期和振幅两个重要指标描述。正常人可以听到的声音频率范围为 20Hz~20KHz。现实存在的声音是模拟量，这对声音保存和长距离传输造成很大的困难，一般的做法是把模拟量转成对应的数字量保存，在需要还原声音的地方再把数字量的转成模拟量输出，参考图 *36_0_1*。



图 36_0_1 音频转换过程

模拟量转成数字量过程，一般可以分为三个过程，分别为采样、量化、编码，参考图 *36_0_2* 。用一个比源声音频率高的采样信号去量化源声音，记录每个采样点的值，最后如果把所有采样点数值连接起来与源声音曲线是互相吻合的，只是它不是连续的。在图中两条蓝色虚线距离就是采样

信号的周期，即对应一个采样频率 (F_S)，可以想象得到采样频率越高最后得到的结果就与源声音越吻合，但此时采样数据量越越大，一般使用 44.1KHz 采样频率即可得到高保真的声音。每条蓝色虚线长度决定着该时刻源声音的量化值，该量化值有另外一个概念与之挂钩，就是量化位数。量化位数表示每个采样点用多少位表示数据范围，常用有 16bit、24bit 或 32bit，位数越高最后还原得到的音质越好，数据量也会越大。

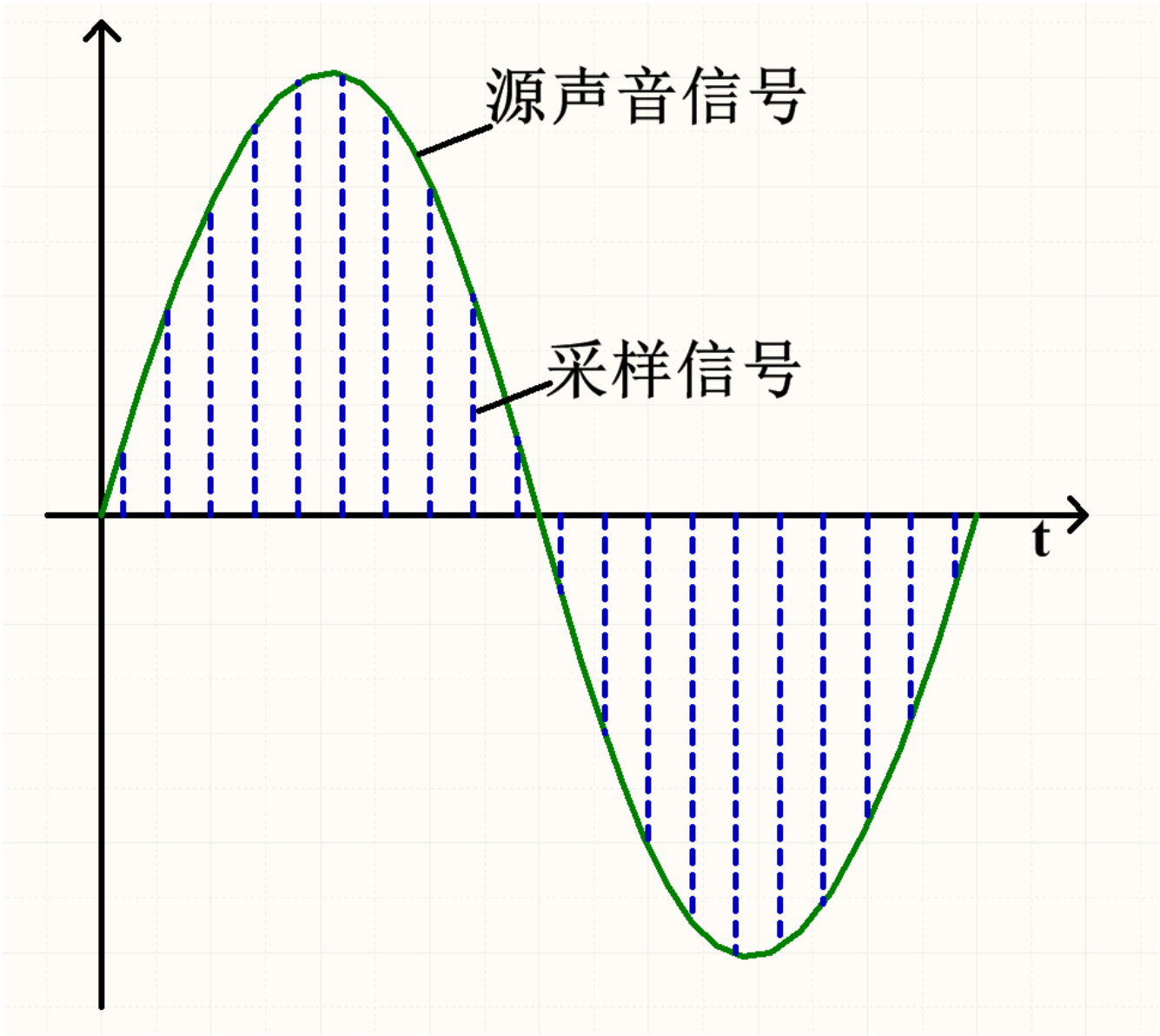


图 36_0_2 声音数字化过程

WM8978 是一个低功耗、高质量的立体声多媒体数字信号编译码器，集成 DAC 和 ADC，可以实现声音信号量化成数字量输出，也可以实现数字量音频数据转换为模拟量声音驱动扬声器。这样使用 WM8978 芯片解决了声音与数字量音频数据转换问题，并且通过配置 WM8978 芯片相关寄存器可以控制转换过程的参数，比如采样频率，量化位数，增益、滤波等等。

WM8978 芯片是一个音频编译码器，但本身没有保存音频数据功能，它只能接收其它设备传输过来的音频数据进行转换输出到扬声器，或者把采样到的音频数据输出到其它具有存储功能的设备保存下来。该芯片与其他设备进行音频数据传输接口就是 I2S 协议的音频接口。

38.1.2 I2S 总线接口

I2S 总线接口有 3 个主要信号，但只能实现数据半双工传输，后来为实现全双工传输有些设备增加了扩展数据引脚。STM32f4xx 系列控制器支持扩展的 I2S 总线接口。

(1) SD(Serial Data)：串行数据线，用于发送或接收两个时分复用的数据通道上的数据 (仅半双工模式)，如果是全双工模式，该信号仅用于发送数据。

(2) WS(Word Select)：字段选择线，也称帧时钟 (LRC) 线，表明当前传输数据的声道，不同标准有不同的定义。WS 线的频率等于采样频率 (F_S)。

(3) CK(Serial Clock)：串行时钟线，也称位时钟 (BCLK)，数字音频的每一位数据都对应有一个 CK 脉冲，它的频率为：2* 采样频率 * 量化位数，2 代表左右两个通道数据。

(4) ext_SD(extend Serial Data)：扩展串行数据线，用于全双工传输的数据接收。

另外，有时为使系统间更好地同步，还要传输一个主时钟 (MCK)，STM32F4xx 系列控制器固定输出为 256* F_S。

38.1.3 音频数据传输协议标准

随着技术的发展，在统一的 I2S 硬件接口下，出现了多种不同的数据格式，可分为左对齐 (MSB) 标准、右对齐 (LSB) 标准、I2S Philips 标准。另外，STM32F4xx 系列控制器还支持 PCM(脉冲编码调) 音频传输协议。下面以 STM32F4xx 系列控制器资源解释这四个传输协议。

STM32f4xx 系列控制器 I2S 的数据寄存器只有 16bit，并且左右声道数据一般是紧邻传输，为正确得到左右两个声道数据，需要软件控制数据对应通道数据写入或读取。另外，音频数据的量化位数可能不同，控制器支持 16bit、24bit 和 32bit 三种数据长度，因为数据寄存器是 16bit 的，所以对于 24bit 和 32bit 数据长度需要发送两个。为此，可以产生四种数据和帧格式组合：

- 将 16 位数据封装在 16 位帧中
- 将 16 位数据封装在 32 位帧中
- 将 24 位数据封装在 32 位帧中
- 将 32 位数据封装在 32 位帧中

当使用 32 位数据包中的 16 位数据时，前 16 位 (MSB) 为有效位，16 位 LSB 被强制清零，无需任何软件操作或 DMA 请求（只需一个读/写操作）。如果程序使用 DMA 传输 (一般都会用)，则 24 位和 32 位数据帧需要对数据寄存器执行两次 DMA 操作。24 位的数据帧，硬件会将 8 位非有效位扩展到带有 0 位的 32 位。对于所有数据格式和通信标准而言，始终会先发送最高有效位 (MSB 优先)。

38.1.3.1 I2S Philips 标准

使用 WS 信号来指示当前正在发送的数据所属的通道，为 0 时表示左通道数据。该信号从当前通道数据的第一个位 (MSB) 之前的一个时钟开始有效。发送方在时钟信号 (CK) 的下降沿改变数据，接收方在上升沿读取数据。WS 信号也在 SCK 的下降沿变化。参考图 *36_0_3*，为 24bit 数据封装在 32bit 帧传输波形。正如之前所说，WS 线频率对于采样频率 F_S，一个 WS 线周期包括发送左声道和右声道数据，在图中实际需要 64 个 CK 周期来完成一次传输。

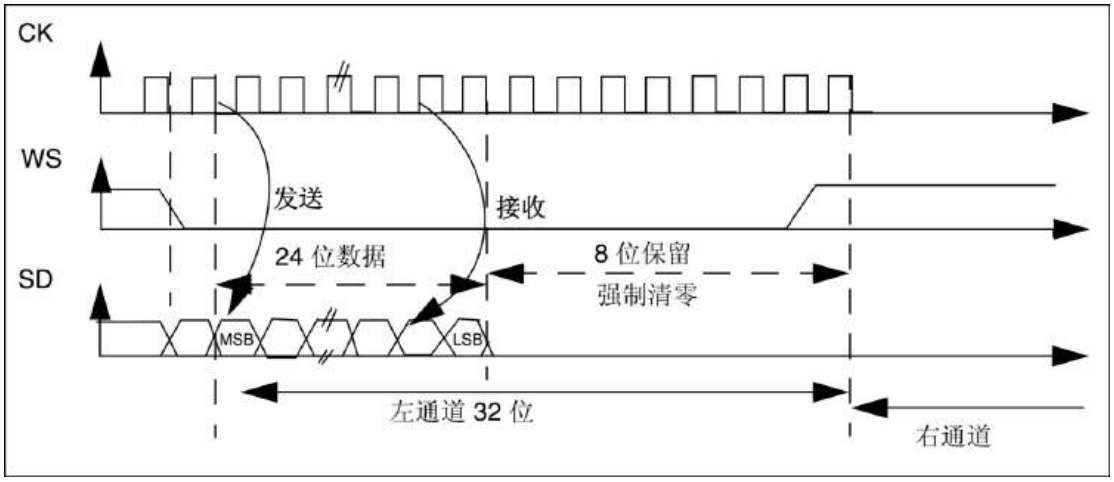


图 36_0_3 I2S Philips 标准 24bit 传输

38.1.3.2 左对齐标准

在 WS 发生翻转同时开始传输数据，参考图 *36_0_4*，为 24bit 数据封装在 32bit 帧传输波形。该标准较少使用。注意此时 WS 为 1 时，传输的是左声道数据，这刚好与 I2S Philips 标准相反。

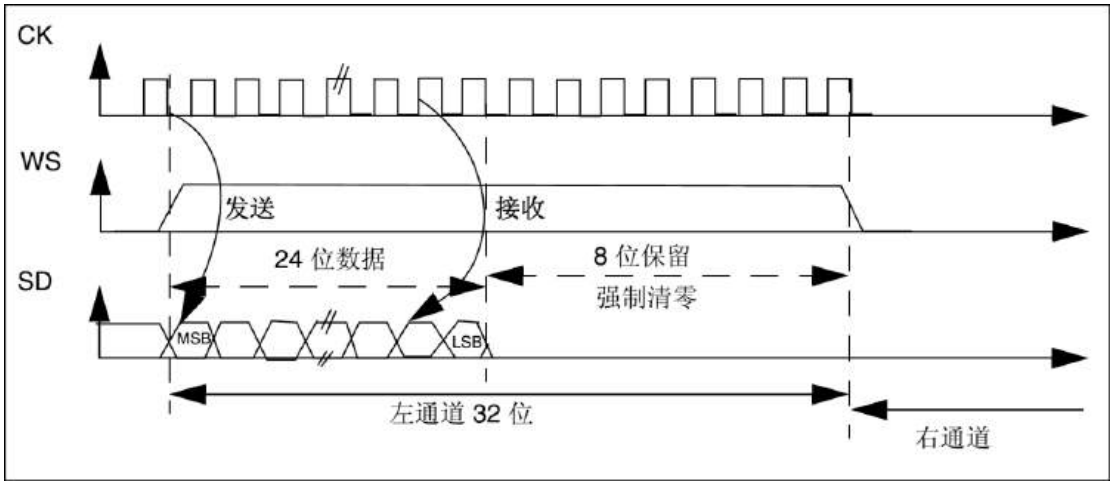


图 36_0_4 左对齐标准 24bit 传输

38.1.3.3 右对齐标准

与左对齐标准类似，参考图 *36_0_5*，为 24bit 数据封装在 32bit 帧传输波形。

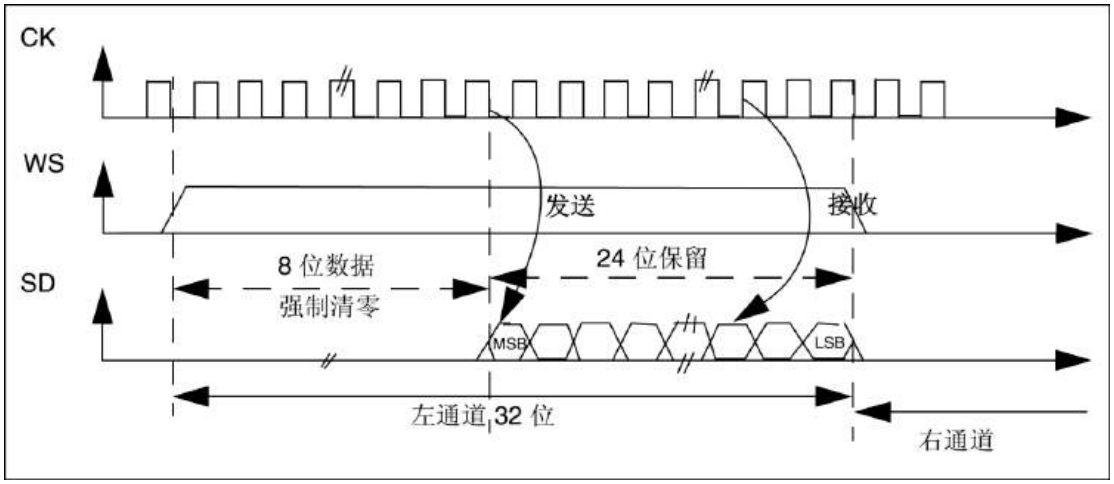


图 36_0_5 右对齐标准 24bit 传输

38.1.3.4 PCM 标准

PCM 即脉冲编码调制，模拟语音信号经过采样量化以及一定数据排列就是 PCM 了。WS 不再作为声道数据选择。它有两种模式，短帧模式和长帧模式，以 WS 信号高电平保持时间为判别依据，长帧模式保持 13 个 CK 周期，短帧模式只保持 1 个 CK 周期，可以通过相关寄存器位选择。如果有多通道数据是在一个 WS 周期内传输完成的，传完左声道数据就紧跟发送右声道数据。图 *36_0_6* 为单声道数据 16bit 扩展到 32bit 数据帧发送波形。

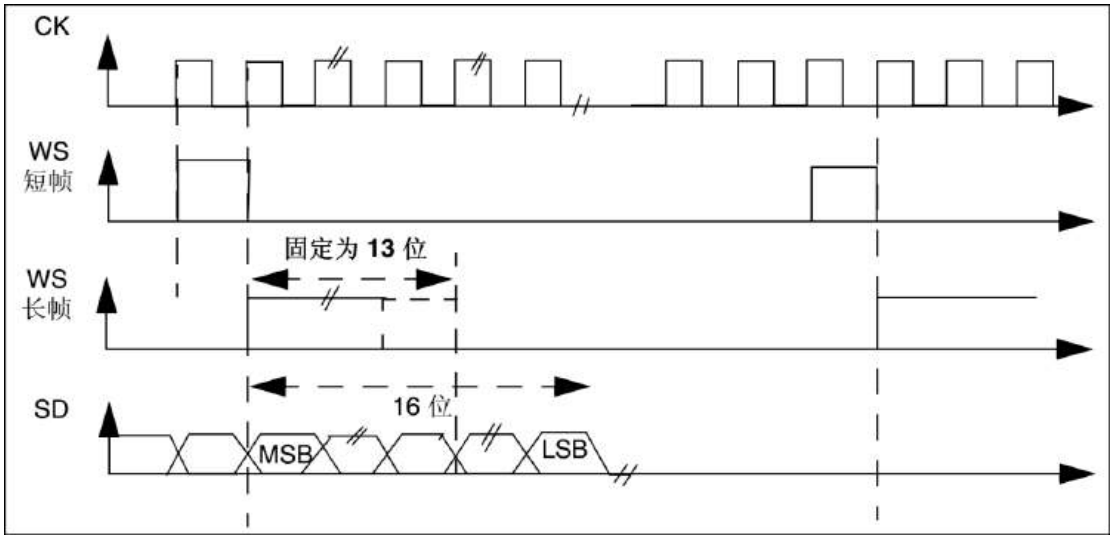


图 36_0_6 PCM 标准 16bit 传输

38.2 I2S 功能框图

STM32f4xx 系列控制器有两个 I2S，I2S2 和 I2S3，两个的资源是相互独立的，但分别与 SPI2 和 SPI3 共用大部分资源。这样 I2S2 和 SPI2 只能选择一个功能使用，I2S3 和 SPI3 相同道理。资源共用包括引脚共用和部分寄存器共用，当然也有部分是专用的。SPI 已经在之前相关章节做了详细讲解，建议先看懂 SPI 相关内容再学习 I2S。

控制器的 I2S 支持两种工作模式，主模式和从模式；主模式下使用自身时钟发生器生成通信时钟。I2S 功能框图参考图 *36_0_7*。

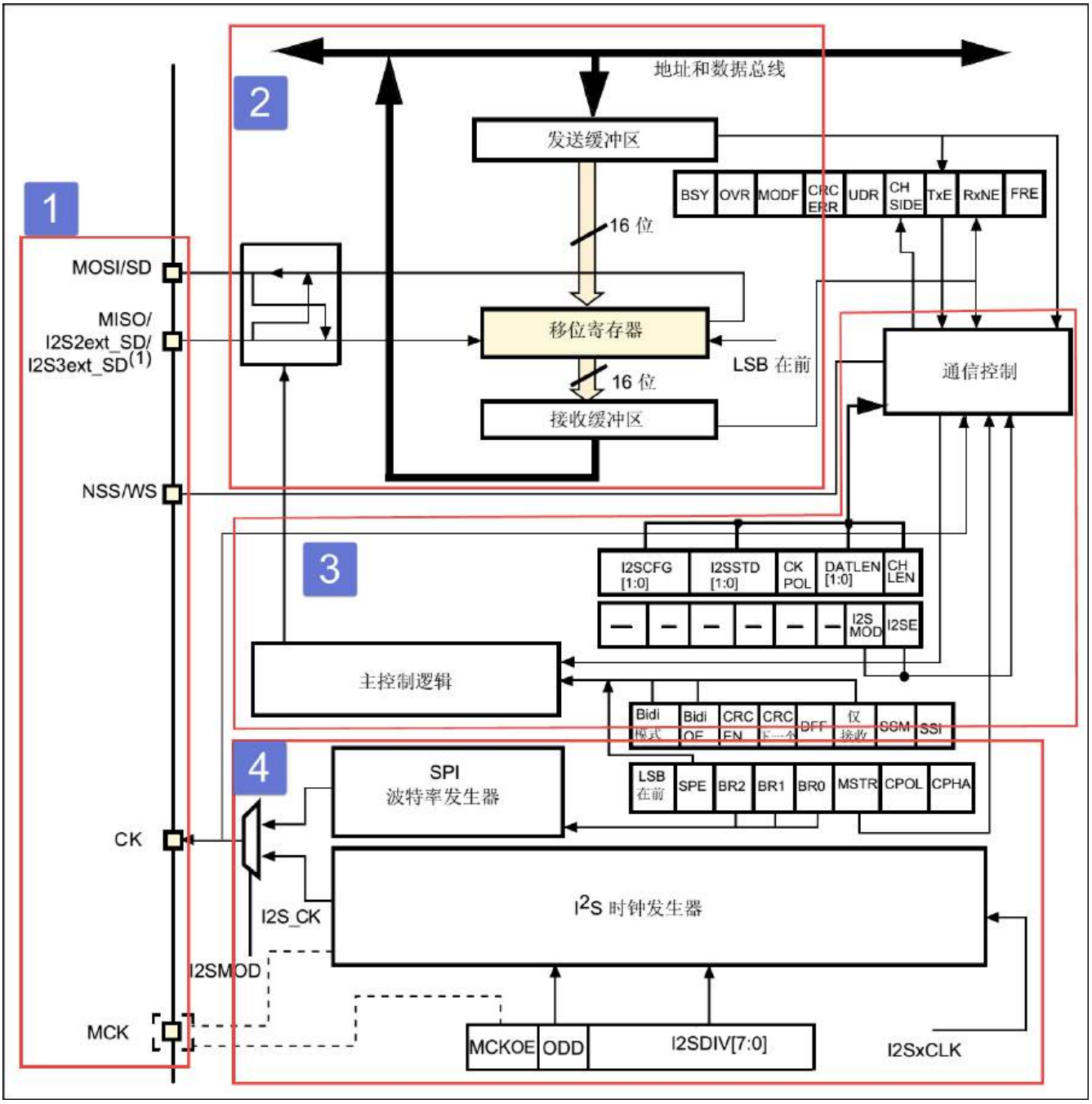


图 36_0_7 I2S 功能框图

38.2.1 功能引脚

I2S 的 SD 映射到 SPI 的 MOSI 引脚，ext_SD 映射到 SPI 的 MISO 引脚，WS 映射到 SPI 的 NSS 引脚，CK 映射到 SPI 的 SCK 引脚。MCK 是 I2S 专用引脚，用于主模式下输出时钟或在从模式下输入时钟。I2S 时钟发生器可以由控制器内部时钟源分频产生，亦可采用 CKIN 引脚输入时钟分频得到，一般使用内部时钟源即可。控制器 I2S 引脚分布参考表 36-0-1。

表 36-0-1 STM32f4xx 系列控制器 I2S 引脚分布

引脚	I2S2	I2S3
SD	PC3/PB15/PI3	PC12/PD6/PB5
ext_SD	PC2/PB14/PI2	PC11/PB4
WS	PB12/PB9	PA4/PA15
CK	PB10/PB13/PD3	PC10/PB3
MCK	PC6	PC7
CKIN	PC9	

38.2.2 数据寄存器

I2S 有一个与 SPI 共用的 SPI 数据寄存器 (SPI_DR)，有效长度为 16bit，用于 I2S 数据发送和接收，它实际由三个部分组成，一个移位寄存器、一个发送缓冲区和一个接收缓冲区，当处于发送模式时，向 SPI_DR 写入数据先保存在发送缓冲区，总线自动把发送缓冲区内容转入到移位寄存器中进行传输；在接收模式下，实际接收到的数据先填充移位寄存器，然后自动转入接收缓冲区，软件读取 SPI_DR 时自动从接收缓冲区内读取。I2S 是挂载在 APB1 总线上的。

38.2.3 逻辑控制

I2S 的逻辑控制通过设置相关寄存器位实现，比如通过配置 SPI_I2S 配置寄存器 (SPI_I2SCFGR) 的相关位可以实现选择 I2S 和 SPI 模式切换、选择 I2S 工作在主模式还是从模式并且选择是发送还是接收、选择 I2S 标准、传输数据长度等等。SPI 控制寄存器 2(SPI_CR2) 可用于设置相关中断

和 DMA 请求使能，I2S 有 5 个中断事件，分别为发送缓冲区为空、接收缓冲区非空、上溢错误、下溢错误和帧错误。SPI 状态寄存器 (SPI_SR) 用于指示当前 I2S 状态。

38.2.4 时钟发生器

I2S 比特率用来确定 I2S 数据线上的数据流和 I2S 时钟信号频率。I2S 比特率 = 每个通道的位数 × 通道数 × 音频采样频率。

图 *36_0_8* 为 I2S 时钟发生器内部结构图。I2SxCLK(x 可选 2 或 3) 可以通过 RCC_CFGR 寄存器的 I2SSRC 位选择使用 PLLI2S 时钟作为 I2S 时钟源或 I2S_CKIN 引脚输入时钟作为 I2S 时钟源。一般选择内部 PLLI2S(通过 R 分频系数) 作为时钟源。例程程序设置 PLLI2S 时钟为 258MHz，R 分频系数为 3，此时 I2SxCLK 时钟为 86MHz。

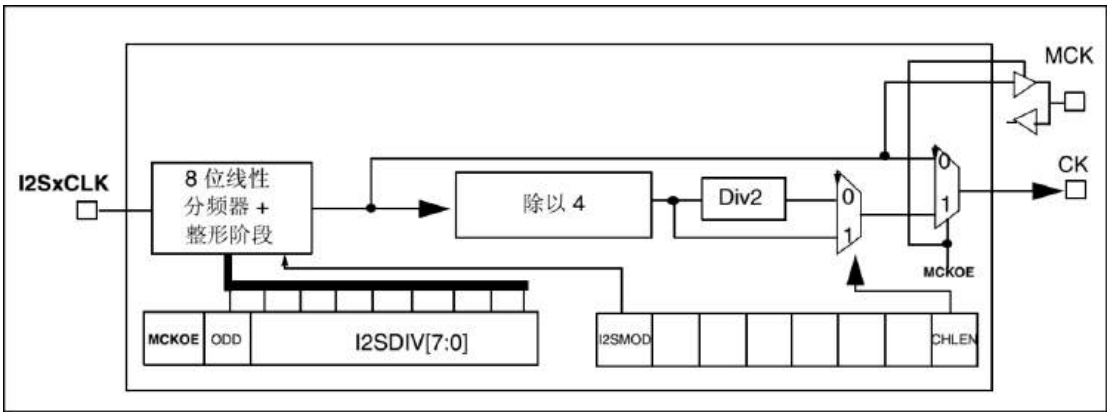


图 36-0-8 I2S 时钟发生器内部结构

SPI_I2S 预分频器寄存器 (SPI_I2SPR) 的 MCKOE 位用于设置 MCK 引脚时钟输出使能；ODD 位设置预分频器的奇数因子，实际分频值 =I2SDIV*2+ODD；I2SDIV 为 8 位线性分频器，不可设置为 0 或 1。

当使能 MCK 时钟输出，即 MCKOE=1 时，采样频率计算如下：

F_S = I2SxCLK/[(16*2)*((2*I2SDIV)+ODD)*8)]（通道帧宽度为 16bit 时）

F_S = I2SxCLK/[(32*2)*((2*I2SDIV)+ODD)*4)]（通道帧宽度为 32bit 时）

当禁止 MCK 时钟输出，即 MCKOE=0 时，采样频率计算如下：

F_S = I2SxCLK/[(16*2)*((2*I2SDIV)+ODD))]（通道帧宽度为 16bit 时）

F_S = I2SxCLK/[(32*2)*((2*I2SDIV)+ODD))]（通道帧宽度为 32bit 时）

38.3 WM8978 音频编译码器

WM8978 是一个低功耗、高质量的立体声多媒体数字信号编译码器。它主要应用于便携式应用。它结合了立体声差分麦克风的前置放大与扬声器、耳机和差分、立体声线输出的驱动，减少了应用时必需的外部组件，比如不需要单独的麦克风或者耳机的放大器。

高级的片上数字信号处理功能，包含一个 5 路均衡功能，一个用于 ADC 和麦克风或者线路输入之间的混合信号的电平自动控制功能，一个纯粹的录音或者重放的数字限幅功能。另外在 ADC 的线路上提供了一个数字滤波的功能，可以更好的应用滤波，比如“减少风噪声”。

WM8978 可以被应用为一个主机或者一个从机。基于共同的参考时钟频率，比如 12MHz 和 13MHz，内部的 PLL 可以为编译码器提供所有需要的音频时钟。与 STM32 控制器连接使用，STM32 一般作为主机，WM8978 作为从机。

图 *36_0_9* 为 WM8978 芯片内部结构示意图，参考来自《WM8978_v4.5》。该图给人的第一印象感觉就是很复杂，密密麻麻很多内容，特别有很多“开关”。实际上，每个开关对应着 WM8978 内部寄存器的一个位，通过控制寄存器的就可以控制开关的状态。

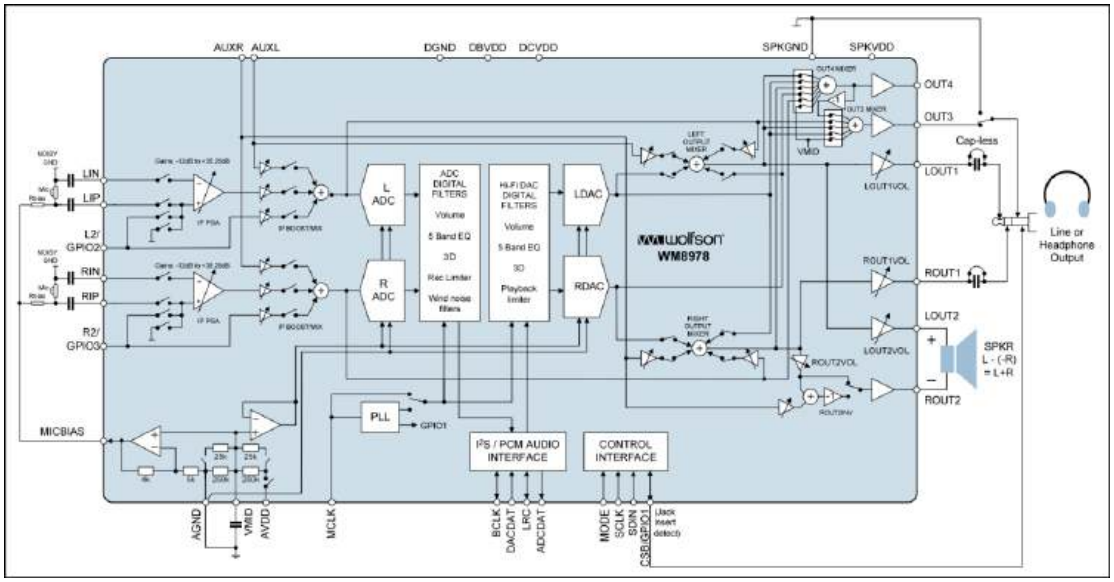


图 36-0-9 WM8978 内部结构

38.3.1 输入部分

WM8978 结构图的左边部分是输入部分，可用于模拟声音输入，即用于录音输入。有三个输入接口，一个是由 LIN 和 LIP、RIN 和 RIP 组合而成的伪差分立体声麦克风输入，一个是由 L2 和 R2 组合的立体声麦克风输入，还有一个是由 AUXL 和 AUXR 组合的线输入或用来传输告警声的输入。

38.3.2 输出部分

WM8978 结构图的右边部分是声音放大输出部分，LOUT1 和 ROUT1 用于耳机驱动，LOUT2 和 ROUT2 用于扬声器驱动，OUT3 和 OUT4 也可以配置成立体声线输出，OUT4 也可以用于提供一个左右声道的单声道混合。

38.3.3 ADC 和 DAC

WM8978 结构图的中边部分是芯片核心内容，处理声音的 AD 和 DA 转换。ADC 部分对声音输入进行处理，包括 ADC 滤波处理、音量控制、输入限幅器/电平自动控制等等。DAC 部分控制声音输出效果，包括 DAC5 路均衡器、DAC 3D 放大、DAC 输出限幅以及音量控制等等处理。

38.3.4 通信接口

WM8978 有两个通信接口，一个是数字音频通信接口，另外一个是控制接口。音频接口是采用 I2S 接口，支持左对齐、右对齐和 I2S 标准模式，以及 DSP 模式 A 和模拟 B。控制接口用于控制器发送控制命令配置 WM8978 运行状态，它提供 2 线或 3 线控制接口，对于 STM32 控制器，我们选择 2 线接口方式，它实际就是 I2C 总线方式，其芯片地址固定为 0011010。通过控制接口可以访问 WM8978 内部寄存器，实现芯片工作环境配置，总共有 58 个寄存器，表示为 R0 至 R57，限于篇幅问题这里不再深入探究，每个寄存器意义参考《WM8978_v4.5》了解。

WM8978 寄存器是 16bit 长度，高 7 位 ([15:9]bit) 用于表示寄存器地址，低 9 为有实际意义，比如对于图 38-9 中的某个开关。所以在控制器向芯片发送控制命令时，必须传输长度为 16bit 的指令，芯片会根据接收命令高 7 位值寻址。

38.3.5 其他部分

WM8978 作为主从机都必须对时钟进行管理，由内部 PLL 单元控制。另外还有电源管理单元。

38.4 WAV 格式文件

WAV 是微软公司开发的一种音频格式文件，用于保存 Windows 平台的音频信息资源，它符合资源互换文件格式 (Resource Interchange File Format，RIFF) 文件规范。标准格式化的 WAV 文件和 CD 格式一样，也是 44.1K 的取样频率，16 位量化数字，因此在声音文件质量和 CD 相差无几！WAVE 是录音时用的标准的 WINDOWS 文件格式，文件的扩展名为“WAV”，数据本身的格式为 PCM 或压缩型，属于无损音乐格式的一种。

38.4.1 RIFF 文件规范

RIFF 有不同数量的 chunk(区块) 组成，每个 chunk 由“标识符”、“数据大小”和“数据”三个部分组成，“标识符”和“数据大小”都是占用 4 个字节空间。简单 RIFF 格式文件结构参考图 *36_0_10*。最开始是 ID 为“RIFF”的 chunk，Size 为“RIFF”chunk 数据字节长度，所以总文件大小为 Size+8。一般来说，chunk 不允许内部再包含 chunk，但有两个例外，ID 为“RIFF”和“LIST”的 chunk 却是允许。对此“RIFF”在其“数据”首 4 个字节用来存放“格式标识码 (Form Type)”，“LIST”则对应“LIST Type”。

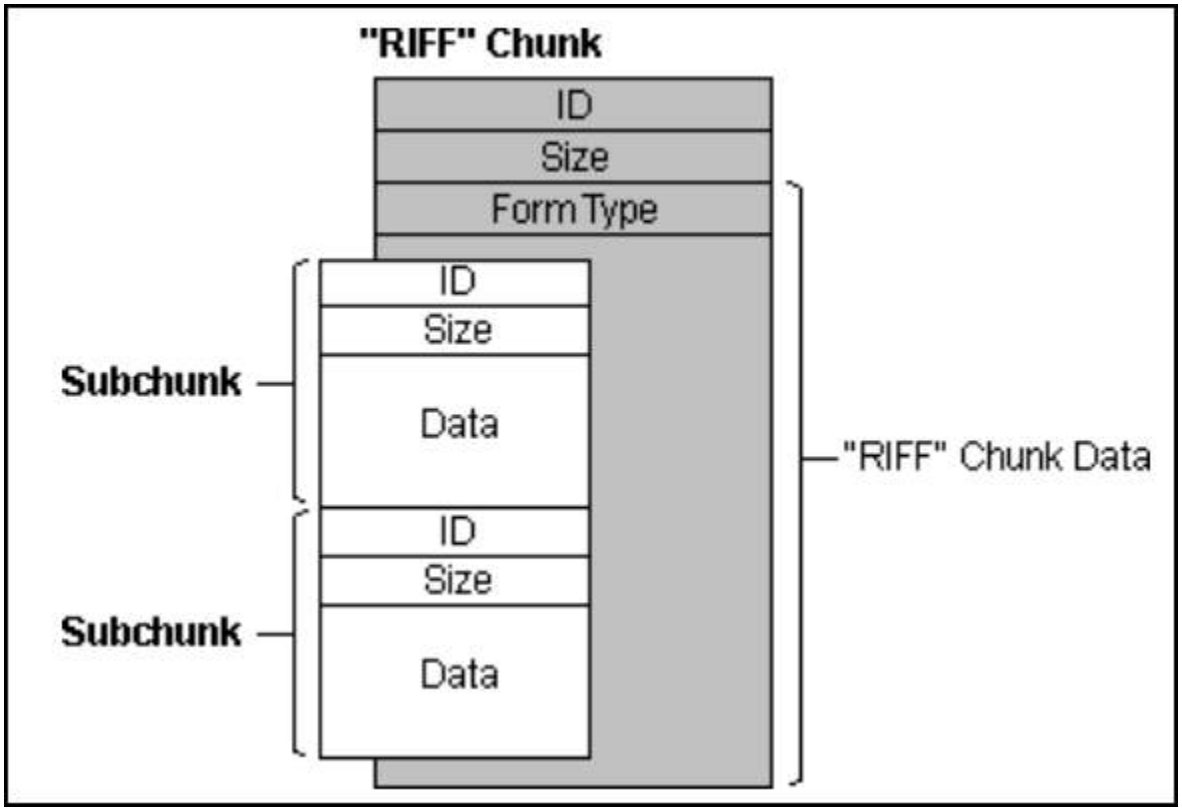


图 36_0_10 RIFF 文件格式结构

38.4.2 WAVE 文件

WAVE 文件是非常简单的一种 RIFF 文件，其“格式标识码”定义为 WAVE。RIFF chunk 包括两个子 chunk，ID 分别为 fmt 和 data，还有一个可选的 fact chunk。Fmt chunk 用于表示音频数据的属性，包括编码方式、声道数目、采样频率、每个采样需要的 bit 数等等信息。fact chunk 是一个可选 chunk，一般当 WAVE 文件由某些软件转化而成就包含 fact chunk。data chunk 包含 WAVE 文件的数字化波形声音数据。WAVE 整体结构如表 36-0-2。

表 36-0-2 WAVE 文件结构

标识码（“RIFF”）
数据大小
格式标识码（“WAVE”）
“fmt”
“fmt”块数据大小
“fmt”数据
“fact”(可选)
“fact”块数据大小
“fact”数据
“data”
声音数据大小
声音数据

data chunk 是 WAVE 文件主体部分，包含声音数据，一般有两个编码格式：PCM 和 ADPCM，ADPCM(自适应差分脉冲编码调制) 属于有损压缩，现在几乎不用，绝大部分 WAVE 文件是 PCM 编码。PCM 编码声音数据可以说是在“数字音频技术”介绍的源数据，主要参数是采样频率和量化位数。

表 36-0-3 为量化位数为 16bit 时不同声道数据在 data chunk 数据排列格式。

表 36-0-3 16bit 声音数据格式

单声道	采样一		采样二		……
	低字节	高字节	低字节	高字节	……
双声道	采样一				……
	左声道		右声道		……
	低字节	高字节	低字节	高字节	……

38.4.3 WAVE 文件实例分析

利用 winhex 工具软件可以非常方便以十六进制查看文件，图 *36_0_11* 为名为“张国荣-一盏小明灯.wav”文件使用 winhex 工具打开的部分界面截图。这部分截图是 WAVE 文件头部分，声音数据部分数据量非常大，有兴趣可以使用 winhex 查看。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	52	49	46	46	F4	FE	83	01	57	41	56	45	66	6D	74	20	RIFF酤? WAVEfmt
00000010	10	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00	........D?...?..
00000020	04	00	10	00	64	61	74	61	48	FE	83	01	00	00	00	00	....dataH?? ....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

图 36_0_11 WAV 文件头实例

下面对文件头进行解读，参考表 36-0-4。

表 36-0-4 WAVE 文件格式说明

	偏移地址	字节数	数据类型	十六进制源码	内容
文件头	00H	4	char	52 49 46 46	“RIFF”标识符
	04H	4	long int	F4 FE 83 01	文件长度：0x0 183FEF4(注意顺序)
	08H	4	char	57 41 56 45	“WAVE”标识符
	0CH	4	char	66 6D 74 20	“fmt ”，最后一位为空格
	10H	4	long int	10 00 00 00	fmt chunk 大小：0x10
	14H	2	int	01 00	编码格式：0x0 1 为 PCM。
	16H	2	int	02 00	声道数目：0x0 1 为单声道，0x 02 为双声道
	18H	4	int	44 AC 00 00	采样频率 (每秒样本数)：0xAC 44(44100)
	1CH	4	long int	10 B1 02 00	每秒字节数：0x 02B110，等于声道数 * 采样频率 * 量化位数/8
	20H	2	int	04 00	每个采样点字节数：0x04，等于声道数 * 量化位数 /8
	22H	2	int	10 00	量化位数：0x1 0
	24H	4	char	64 61 74 61	“data”数据标识符
	28H	4	long int	48 FE 83 01	声音数据量：0x 0183FE48

38.5 I2S 初始化结构体详解

HAL 库函数对 I2S 外设建立了一个初始化结构体 I2S_InitTypeDef。初始化结构体成员用于设置 I2S 工作环境参数，并由 I2S 相应初始化配置函数 HAL_I2S_Init 调用，这些设定参数将会设置 I2S 相应的寄存器，达到配置 I2S 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 stm32f4xx_hal_spi.h 文件中，初始化库函数定义在 stm32f4xx_hal_spi.c 文件中，编程时我们可以结合这两个文件内注释使用。

I2S 初始化结构体用于配置 I2S 基本工作环境，比如 I2S 工作模式、通信标准选择等等。它被 I2S_Init 函数调用。

列表 1: 代码清单 36_0_1 I2S_InitTypeDef 结构体

```
typedef struct {
    uint32_t Mode;          // I2S 模式选择
    uint32_t Standard;      // I2S 标准选择
    uint32_t DataFormat;    // 数据格式
    uint32_t MCLKOutput;    // 主时钟输出使能
    uint32_t AudioFreq;     // 采样频率
    uint32_t CPOL;          // 空闲电平选择
} I2S_InitTypeDef;
```

(1) Mode：I2S 模式选择，可选主机发送、主机接收、从机发送以及从机接收模式，它设定 SPI_I2SCFGR 寄存器 I2SCFG 位的值。一般设置 STM32 控制器为主机模式，当播放声音时选择发送模式；当录制声音时选择接收模式。

(2) Standard：通信标准格式选择，可选 I2S Philips 标准、左对齐标准、右对齐标准、PCM 短帧标准或 PCM 长帧标准，它设定 SPI_I2SCFGR 寄存器 I2SSTD 位和 PCMSYNC 位的值。一般设置为 I2S Philips 标准即可。

(3) DataFormat：数据格式选择，设定有效数据长度和帧长度，可选标准 16bit 格式、扩展 16bit(32bit 帧长度) 格式、24bit 格式和 32bit 格式，它设定 SPI_I2SCFGR 寄存器 DATLEN 位和 CHLEN 位的值。对应 16bit 数据长度可选 16bit 或 32bit 帧长度，其他都是 32bit 帧长度。

(4) MCLKOutput：主时钟输出使能控制，可选使能输出或禁止输出，它设定 SPI_I2SPR 寄存器 MCKOE 位的值。为提高系统性能一般使能主时钟输出。

(5) AudioFreq：采样频率设置，HAL 库提供采样采样频率选择，分别为 8kHz、11kHz、16kHz、22kHz、32kHz、44kHz、48kHz、96kHz、192kHz 以及默认 2Hz，它设定 SPI_I2SPR 寄存器的值。

(6) CPOL：空闲状态的 CK 线电平，可选高电平或低电平，它设定 SPI_I2SCFGR 寄存器 CKPOL 位的值。一般设置为电平即可。

38.6 录音与回放实验

WAV 格式文件在现阶段一般以无损音乐格式存在，音质可以达到 CD 格式标准。结合上一章 SD 卡操作内容，本实验通过 FatFS 文件系统函数从 SD 卡读取 WAV 格式文件数据，然后通过 I2S 接口将音频数据发送到 WM8978 芯片，这样在 WM8978 芯片的扬声器接口即可输出声音，整个系统构成一个简单的音频播放器。反过来的，我们可以实现录音功能，控制启动 WM8978 芯片的麦克风输入功能，音频数据从 WM8978 芯片的 I2S 接口传输到 STM32 控制器存储器中，利用 SD 卡文件读写函数，根据 WAV 格式文件的要求填充文件头，然后就把 WM8978 传输过来的音频数据写入到 WAV 格式文件中，这样就可以制成一个 WAV 格式文件，可以通过开发板回放也可以在电脑端回放。

38.6.1 硬件设计

开发板板载 WM8978 芯片，具体电路设计参考图 *36_0_12*。WM8978 与 STM32f4xx 有两个连接接口，I2S 音频接口和两线 I2C 控制接口，通过将 WM8978 芯片的 MODE 引脚拉低选择两线控制接口，符合 I2C 通信协议，这也导致 WM8978 是只写的，所以在程序上需要做一些处理。WM8978 输入部分有两种模式，一个是板载咪头输入，另外一个是通过 3.5mm 耳机插座引出。WM8978 输出部分通过 3.5mm 耳机插座引出，可直接接普通的耳机线或作为功放设备的输入源。

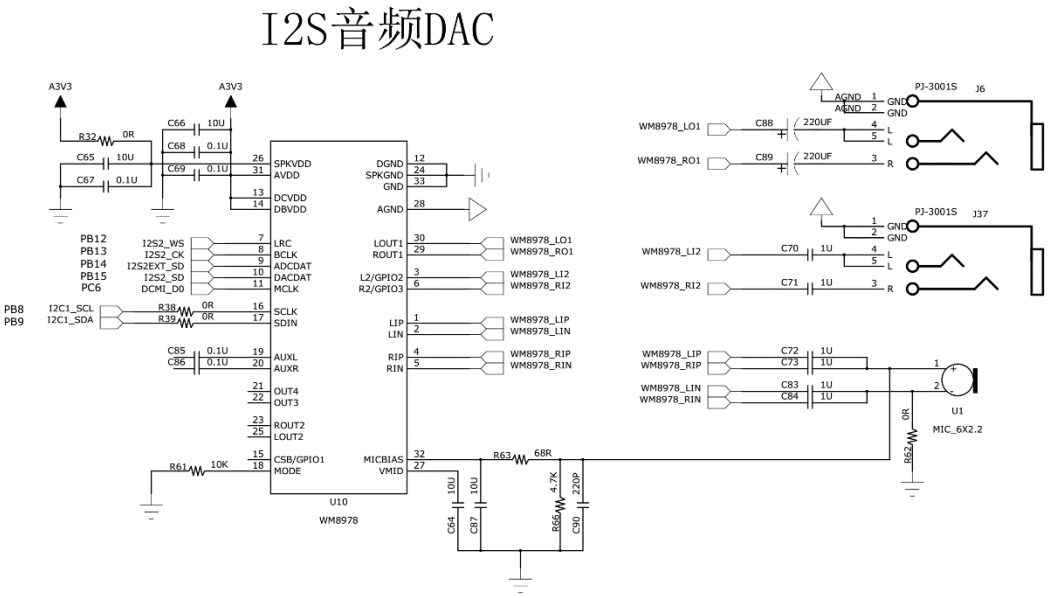


图 36-0-12 WM8978 电路设计

38.6.2 软件设计

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等没有全部罗列出来，完整的代码请参考本章配套的工程。

上一章我们已经介绍了基于 SD 卡的文件系统，认识读写 SD 卡内文件方法，前面已经介绍了 WAV 格式文件结构以及 WM8978 芯片相关内容，通过 WM8978 音频接口传输过来的音频数据可以直接作为 WAV 格式文件的音频数据部分，大致过程就是程序控制 WM8978 启动录音功能，通过 I2S 音频数据接口 WM8978 的录音输出传输到 STM32 控制器指定缓冲区内，然后利用 FatFs 的文件写入函数把缓冲区数据写入到 WAV 格式文件中，最终实现声音录制功能。同样的道理，WAV 格式文件中的音频数据可以直接传输给 WM8978 芯片实现音乐播放，整个过程与声音录制工程相反。

STM32 控制器与 WM8978 通信可分为两部分驱动函数，一部分是 I2C 控制接口，另一部分是 I2S 音频数据接口。

bsp_wm8978.c 和 bsp_wm8978.h 两个是专门创建用来存放 WM8978 芯片驱动代码。

38.6.2.1 I2C 控制接口

WM8978 要正常工作并且实现符合我们的要求，我们必须对芯片相关寄存器进行必须要配置，STM32 控制器通过 I2C 接口与 WM8978 芯片控制接口连接。I2C 接口内容也已经在以前做了详细介绍，这里主要讲解 WM8978 的功能函数。

bsp_wm8978.c 文件中的 I2C_GPIO_Config 函数、I2C_Mode_Configu 函数以及 wm8978_Init 函数用于 I2C 通信接口 GPIO 和 I2C 相关配置，属于常规配置可以参考 GPIO 和 I2C 章节理解，这里不再分析，代码具体见本章配套程序工程文件。

38.6.2.1.1 输入输出选择枚举

列表 2: 代码清单 36_0_2 输入输出选择枚举

```
/* WM8978 音频输入通道控制选项, 可以选择多路, 比如 MIC_LEFT_ON | LINE_ON */
typedef enum {
    IN_PATH_OFF    = 0x00, /* 无输入 */
    MIC_LEFT_ON    = 0x01, /* LIN,LIP 脚, MIC 左声道 (接板载咪头)   */
    MIC_RIGHT_ON   = 0x02, /* RIN,RIP 脚, MIC 右声道 (接板载咪头)   */
    LINE_ON        = 0x04, /* L2,R2 立体声输入 (接板载耳机插座) */
    AUX_ON         = 0x08, /* AUXL,AUXR 立体声输入 (开发板没用到)  */
    DAC_ON         = 0x10, /* I2S 数据 DAC (CPU 产生音频信号) */
    ADC_ON         = 0x20  /* 输入的音频馈入 WM8978 内部 ADC (I2S 录音) */
} IN_PATH_E;

/* WM8978 音频输出通道控制选项, 可以选择多路 */
typedef enum {
    OUT_PATH_OFF   = 0x00, /* 无输出 */
    EAR_LEFT_ON    = 0x01, /* LOUT1 耳机左声道 (接板载耳机插座) */
    EAR_RIGHT_ON   = 0x02, /* ROUT1 耳机右声道 (接板载耳机插座) */
    SPK_ON         = 0x04, /* LOUT2 和 ROUT2 反相输出单声道 (开发板没用到) */
    OUT3_4_ON      = 0x08, /* OUT3 和 OUT4 输出单声道音频 (开发板没用到) */
} OUT_PATH_E;
```

IN_PATH_E 和 OUT_PATH_E 枚举了 WM8978 芯片可用的声音输入源和输出端口，具体到开发板，如果进行录用功能，设置输入源为 (MIC_RIGHT_ON|ADC_ON) 或 (LINE_ON|ADC_ON)，设置输出端口为 OUT_PATH_OFF 或 (EAR_LEFT_ON | EAR_RIGHT_ON)；对于音乐播放功能，设置输入源为 DAC_ON，设置输出端口为 (EAR_LEFT_ON | EAR_RIGHT_ON)。

38.6.2.1.2 宏定义

列表 3: 代码清单 36_0_3 宏定义

```
/* 定义最大音量 */
#define VOLUME_MAX                              63     /* 最大音量 */
#define VOLUME_STEP                             1     /* 音量调节步长 */

/* 定义最大 MIC 增益 */
#define GAIN_MAX                                63     /* 最大增益 */
#define GAIN_STEP                               1     /* 增益步长 */

/* WM8978 I2C 从机地址 */
#define WM8978_SLAVE_ADDRESS                    0x34
/* 等待超时时间 */
#define WM8978_I2C_FLAG_TIMEOUT                 ((uint32_t)0x4000)
#define WM8978_I2C_LONG_TIMEOUT   ((uint32_t)(10*WM8978_I2C_FLAG_TIMEOUT))
```

WM8978 声音调节有一定的范围限制，比如 R52(LOUT1 Volume Control) 的 LOUT1VOL[5:0] 位用于设置 LOUT1 的音量大小，可赋值范围为 0~63。WM8978 包含可调节的输入麦克风 PGA 增益，可对每个外部输入端口可单独设置增益大小，比如 R45(Left Channel input PGA volume control) 的 INPPGAVOL[5:0] 位用于设置左通道输入增益音量，最大可设置值为 63。

WM8978 控制接口被设置为 I2C 模式，其地址固定为 0011010，为方便使用，直接定义为 0x34。

最后定义 I2C 通信超时等待时间。

38.6.2.1.3 WM8978 寄存器写入

列表 4: 代码清单 36_0_4 WM8978 寄存器写入

```
/**
* @brief  通过 I2C 将给定寄存器的字节写入音频编解码器
* @param  RegisterAddr: 待写入寄存器的地址
* @param  RegisterValue: 要写入目标寄存器的字节值
* @retval 通信成功返回 1，失败返回 0
*/
static uint8_t WM8978_I2C_WriteRegister(uint8_t RegisterAddr,
uint16_t RegisterValue)
{
    uint16_t tmp;

    tmp  = (RegisterValue&0xff) << 8;
    tmp |= ((RegisterAddr << 1) & 0xFE) | ((RegisterValue >> 8) & 0x1);
    if (HAL_I2C_Master_Transmit(&I2C_Handle,WM8978_SLAVE_ADDRESS,(uint8_t
    *)&tmp,2,WM8978_I2C_FLAG_TIMEOUT)==HAL_OK) {    return 1;
    } else
        return 0;
}
```

WM8978_I2C_WriteRegister 用于向 WM8978 芯片寄存器写入数值，达到配置芯片工作环境，函数有两个形参，一个是寄存器地址，可设置范围为 0~57；另外一个是寄存器值，WM8978 芯片寄存器总共有 16bit，前 7bit 用于寻址，后 9 位才是数据，这里寄存器值形参使用 uint16_t 类型，只有低 9 位有效。

使用 I2C 通信，首先使用中间变量提取正确的寄存器地址及数据值，然后调用 HAL_I2C_Master_Transmit 函数发送两次数据，因为 I2C 数据发送一次只能发送 8bit 数据，为此需要把 RegisterValue 变量的第 9bit 整合到 RegisterAddr 变量的第 0 位先发送，接下来再发送 RegisterValue 变量的低 8bit 数据。

HAL_I2C_Master_Transmit 函数中还有 I2C 通信超时等待功能，防止出错时卡死。

38.6.2.1.4 WM8978 寄存器读取

WM8978 芯片是从硬件上选择 I2C 通信模式，该模式是只写的，STM32 控制器无法读取 WM8978 寄存器内容，但程序有时需要用到寄存器内容，为此我们创建了一个存放 WM8978 所有寄存器值的数组，在系统复位是将数组内容设置为 WM8978 缺省值，然后在每次修改寄存器内容时同步更新该数组内容，这样可以达到该数组与 WM8978 寄存器内容相等的效果，参考代码清单 38-5。

列表 5: 代码清单 36_0_5 WM8978 寄存器值缓冲区和读取

```
/*
wm8978 寄存器缓存
由于 WM8978 的 I2C 两线接口不支持读取操作，因此寄存器值缓存在内存中，
当写寄存器时同步更新缓存，读寄存器时直接返回缓存中的值。
寄存器 MAP 在 WM8978(V4.5_2011).pdf 的第 89 页，寄存器地址是 7bit， 寄存器数据是 9bit
*/
static uint16_t wm8978_RegCash[] = {
    0x000, 0x000, 0x000, 0x000, 0x050, 0x000, 0x140, 0x000,
    0x000, 0x000, 0x000, 0x0FF, 0x0FF, 0x000, 0x100, 0x0FF,
    0x0FF, 0x000, 0x12C, 0x02C, 0x02C, 0x02C, 0x02C, 0x000,
    0x032, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
    0x038, 0x00B, 0x032, 0x000, 0x008, 0x00C, 0x093, 0x0E9,
    0x000, 0x000, 0x000, 0x000, 0x003, 0x010, 0x010, 0x100,
    0x100, 0x002, 0x001, 0x001, 0x039, 0x039, 0x039, 0x039,
    0x001, 0x001
};

/**
* @brief  从 cash 中读回读回 wm8978 寄存器
* @param  _ucRegAddr :  寄存器地址
* @retval 寄存器值
*/
```

(下页继续)

(续上页)

```
static uint16_t wm8978_ReadReg(uint8_t _ucRegAddr)
{
    return wm8978_RegCash[_ucRegAddr];
}

/**
* @brief  写 wm8978 寄存器
* @param  _ucRegAddr:  寄存器地址
* @param  _usValue:  寄存器值
* @retval 0: 写入失败
*         1: 写入成功
*/
static uint8_t wm8978_WriteReg(uint8_t _ucRegAddr, uint16_t _usValue)
{
    uint8_t res;
    res=WM8978_I2C_WriteRegister(_ucRegAddr,_usValue);
    wm8978_RegCash[_ucRegAddr] = _usValue;
    return res;
}
```

wm8978_WriteReg 实现向 WM8978 寄存器写入数据并修改缓冲区内容。

38.6.2.1.5 输出音量修改与读取

列表 6: 代码清单 36_0_6 音量修改与读取

```
/**
* @brief  修改输出通道 1 音量
* @param  _ucVolume : 音量值, 0-63
* @retval 无
*/
```

(下页继续)

(续上页)

```
void wm8978_SetOUT1Volume(uint8_t _ucVolume)
{
    uint16_t regL;
    uint16_t regR;

    if (_ucVolume > VOLUME_MAX) {
        _ucVolume = VOLUME_MAX;
    }
    regL = _ucVolume;
    regR = _ucVolume;
    /*
    R52 LOUT1 Volume control
    R53 ROUT1 Volume control
    */
    /* 先更新左声道缓存值 */
    wm8978_WriteReg(52, regL | 0x00);

    /* 再同步更新左右声道的音量 */
    /* 0x180 表示 在音量为 0 时再更新，避免调节音量出现的“嘎哒”声 */
    wm8978_WriteReg(53, regR | 0x100);
}

/**
* @brief  读取输出通道 1 音量
* @param  无
* @retval 当前音量值
*/
uint8_t wm8978_ReadOUT1Volume(void)
{
    return (uint8_t)(wm8978_ReadReg(52) & 0x3F );
}
```

(下页继续)

(续上页)

```
/**
* @brief  输出静音.
* @param  _ucMute: 模式选择
*          @arg 1: 静音
*          @arg 0: 取消静音
* @retval 无
*/
void wm8978_OutMute(uint8_t _ucMute)
{
    uint16_t usRegValue;
    if (_ucMute == 1) { /* 静音 */
        usRegValue = wm8978_ReadReg(52); /* Left Mixer Control */
        usRegValue |= (1u << 6);
        wm8978_WriteReg(52, usRegValue);

        usRegValue = wm8978_ReadReg(53); /* Left Mixer Control */
        usRegValue |= (1u << 6);
        wm8978_WriteReg(53, usRegValue);

        usRegValue = wm8978_ReadReg(45); /* Right Mixer Control */
        usRegValue |= (1u << 6);
        wm8978_WriteReg(45, usRegValue);

        usRegValue = wm8978_ReadReg(55); /* Right Mixer Control */
        usRegValue |= (1u << 6);
        wm8978_WriteReg(55, usRegValue);
    } else {  /* 取消静音 */
        usRegValue = wm8978_ReadReg(52);
        usRegValue &= ~(1u << 6);
        wm8978_WriteReg(52, usRegValue);

        usRegValue = wm8978_ReadReg(53); /* Left Mixer Control */
```

(下页继续)

(续上页)

```
        usRegValue &= ~(1u << 6);
        wm8978_WriteReg(53, usRegValue);

        usRegValue = wm8978_ReadReg(45);
        usRegValue &= ~(1u << 6);
        wm8978_WriteReg(45, usRegValue);

        usRegValue = wm8978_ReadReg(55); /* Left Mixer Control */
        usRegValue &= ~(1u << 6);
        wm8978_WriteReg(55, usRegValue);
    }
}
```

wm8978_SetOUT1Volume 函数用于修改 OUT1 通道的音量大小，有一个形参用于指示音量大小，要求范围为 0~63。这里同时更新 OUT1 的左右两个声道音量，WM8978 芯片的 R52 和 R53 分别用于设置 OUT1 的左声道和右声道音量，具体位段意义参考表 36-0-5。wm8978_SetOUT1Volume 函数会同时修改 WM8978 寄存器缓存区 wm8978_RegCash 数组内容。

表 36-0-5 OUT1 音量控制寄存器

寄存器地址	位	默认值	描述
R52(LOUT1 Volume Control)	8	不锁存	直到一个 1 写入到 HPVU 才更新 LOUT1 和 ROUT1 音量
	7	0	耳机音量零交叉使能：0= 仅仅在零交叉时改变增益；1= 立即改变增益
	6	0	左耳机输出消声：0= 正常操作；1= 消声
	5:0	11101	左耳机输出驱动： 000000=-57dB … 111001=0dB … 111111=+6dB
R53(ROUT1 Volume Control)	8	不锁存	直到一个 1 写入到 HPVU 才更新 LOUT1 和 ROUT1 音量
	7	0	耳机音量零交叉使能：0= 仅仅在零交叉时改变增益；1= 立即改变增益
	6	0	左耳机输出消声：0= 正常操作；1= 消声
	5:0	11101	右耳机输出驱动： 000000=-57dB … 111001=0dB … 111111=+6dB

另外，wm8978_SetOUT2Volume 用于设置 OUT2 的音量，程序结构与 wm8978_SetOUT1Volume 相同，只是对应修改 R45 和 R55。

wm8978_ReadOUT1Volume 函数用于读取 OUT1 的音量，它实际就是读取 wm8978_RegCash 数组对应元素内容。

wm8978_OutMute 用于静音控制，它有一个形参用于设置静音效果，如果为 1 则为开启静音，如

果为 0 则取消静音。静音控制是通过 R52 和 R53 的第 6 位实现的，在进入静音模式时需要先保存 OUT1 和 OUT2 的音量大小，然后在退出静音模式时就可以正确返回到静音前 OUT1 和 OUT2 的配置。

38.6.2.1.6 输入增益调整

列表 7: 代码清单 36_0_7 输入增益调整

```
/**
* @brief  设置增益
* @param  _ucGain : 增益值, 0-63
* @retval 无
*/
void wm8978_SetMicGain(uint8_t _ucGain)
{
    if (_ucGain > GAIN_MAX) {
        _ucGain = GAIN_MAX;
    }

    /* PGA 音量控制  R45,  R46
    Bit8  INPPGAUPDATE
    Bit7  INPPGAZCL   过零再更改
    Bit6  INPPGAMUTEL   PGA 静音
    Bit5:0  增益值, 010000 是 0dB
    */
    wm8978_WriteReg(45, _ucGain);
    wm8978_WriteReg(46, _ucGain | (1 << 8));
}

/**
* @brief  设置 Line 输入通道的增益
* @param  _ucGain : 音量值, 0-7. 7 最大, 0 最小。 可衰减可放大。
```

(下页继续)

(续上页)

```
* @retval 无
*/
void wm8978_SetLineGain(uint8_t _ucGain)
{
    uint16_t usRegValue;

    if (_ucGain > 7) {
        _ucGain = 7;
    }

    /*
    Mic 输入信道的增益由 PGABOOSTL 和 PGABOOSTR 控制
    Aux 输入信道的输入增益由 AUXL2BOOSTVO[2:0] 和 AUXR2BOOSTVO[2:0] 控制
    Line 输入信道的增益由 LIP2BOOSTVOL[2:0] 和 RIP2BOOSTVOL[2:0] 控制
    */
    /* R47 (左声道), R48 (右声道), MIC 增益控制寄存器
    R47 (R48 定义与此相同)
B8 PGABOOSTL=1,0 表示 MIC 信号直通无增益, 1 表示 MIC 信号 +20dB 增益(通过自举电路)
B7 = 0,  保留
B6:4  L2_2BOOSTVOL=x, 0 表示禁止, 1-7 表示增益-12dB ~ +6dB (可以衰减也可以放大)
B3 = 0,  保留
B2:0 AUXL2BOOSTVOL=x, 0 表示禁止, 1-7 表示增益-12dB~+6dB (可以衰减也可以放大)
    */

    usRegValue = wm8978_ReadReg(47);
    usRegValue &= 0x8F;/* 将 Bit6:4 清 0   1000 1111*/
    usRegValue |= (_ucGain << 4);
    wm8978_WriteReg(47, usRegValue);  /* 写左声道输入增益控制寄存器 */

    usRegValue = wm8978_ReadReg(48);
    usRegValue &= 0x8F;/* 将 Bit6:4 清 0   1000 1111*/
    usRegValue |= (_ucGain << 4);
```

(下页继续)

(续上页)

```
    wm8978_WriteReg(48, usRegValue);  /* 写右声道输入增益控制寄存器 */
}
```

wm8978_SetMicGain 用于设置麦克风输入的增益，可以设置增强或减弱输入效果，比如对于部分声音源本身就是比较微弱，我们就可以设置放大该信号，从而得到合适的录制效果，该函数主要通过设置 R45 和 R46 实现，可设置的范围为 0~63，默认值为 16，没有增益效果。

wm8978_SetLineGain 用于设置 LINE 输入的增益，对应芯片的 L2 和 R2 引脚组合的输入，开发板使用耳机插座引出拓展。它通过设置 R47 和 R48 寄存器实现，可设置范围为 0~7，默认值为 0，没有增益效果。

38.6.2.1.7 音频接口标准选择

列表 8: 代码清单 36_0_8 wm8978_CfgAudioIF 函数

```
/**
* @brief  配置 WM8978 的音频接口 (I2S)
* @param  _usStandard : 接口标准,
        I2S_Standard_Phillips, I2S_Standard_MSB 或 I2S_Standard_LSB
* @param  _ucWordLen : 字长, 16、24、32  (丢弃不常用的 20bit 格式)
* @retval 无
*/
void wm8978_CfgAudioIF(uint16_t _usStandard, uint8_t _ucWordLen)
{
    uint16_t usReg;

    /* WM8978(V4.5_2011).pdf 73 页, 寄存器列表 */
    /*  REG R4, 音频接口控制寄存器
    B8    BCP  = X, BCLK 极性, 0 表示正常, 1 表示反相
    B7    LRCP = x, LRC 时钟极性, 0 表示正常, 1 表示反相
    B6:5  WL = x,  字长, 00=16bit, 01=20bit, 10=24bit, 11=32bit
```

(下页继续)

(续上页)

```
        (右对齐模式只能操作在最大 24bit)
    B4:3  FMT = x, 音频数据格式, 00= 右对齐, 01= 左对齐, 10=I2S 格式, 11=PCM
    B2    DACLRSWAP = x, 控制 DAC 数据出现在 LRC 时钟的左边还是右边
    B1    ADCLRSWAP = x, 控制 ADC 数据出现在 LRC 时钟的左边还是右边
    B0    MONO  = 0, 0 表示立体声, 1 表示单声道, 仅左声道有效
    */
    usReg = 0;
    if (_usStandard == I2S_Standard_Phillips) { /* I2S 飞利浦标准 */
        usReg |= (2 << 3);
    } else if (_usStandard == I2S_Standard_MSB) { /* MSB 对齐标准 (左对齐) */
        usReg |= (1 << 3);
    } else if (_usStandard == I2S_Standard_LSB) { /* LSB 对齐标准 (右对齐) */
        usReg |= (0 << 3);
    } else {  /*PCM 标准 (16 位通道帧上带长或短帧同步或者 16 位数据帧扩展为 32 位通
道帧) */
        usReg |= (3 << 3);;
    }

    if (_ucWordLen == 24) {
        usReg |= (2 << 5);
    } else if (_ucWordLen == 32) {
        usReg |= (3 << 5);
    } else {
        usReg |= (0 << 5);    /* 16bit */
    }
    wm8978_WriteReg(4, usReg);

    /*
    R6, 时钟产生控制寄存器
    MS = 0,  WM8978 被动时钟, 由 MCU 提供 MCLK 时钟
    */
    wm8978_WriteReg(6, 0x000);
```

(下页继续)

(续上页)

```
}
```

wm8978_CfgAudioIF 函数用于设置 WM8978 芯片的音频接口标准，它有两个形参，第一个是标准选择，可选 I2S Philips 标准 (I2S_Standard_Phillips)、左对齐标准 (I2S_Standard_MSB) 以及右对齐标准 (I2S_Standard_LSB)；另外一个形参是字长设置，可选 16bit、24bit 以及 32bit，较常用 16bit。它函数通过控制 WM8978 芯片 R4 实现，最后还通过通用时钟控制寄存器 R6 设置芯片的 I2S 工作在从模式，时钟线为输入时钟。

38.6.2.1.8 输入输出通道设置

列表 9: 代码清单 36_0_9 wm8978_CfgAudioPath 函数

```
void wm8978_CfgAudioPath(uint16_t _InPath, uint16_t _OutPath)
{
    uint16_t usReg;
    if ((_InPath == IN_PATH_OFF) && (_OutPath == OUT_PATH_OFF)) {
        wm8978_PowerDown();
        return;
    }

    /*
    R1 寄存器 Power manage 1
    Bit8    BUFDCOPEN,  Output stage 1.5xAVDD/2 driver enable
    Bit7    OUT4MIXEN, OUT4 mixer enable
    Bit6    OUT3MIXEN, OUT3 mixer enable
    Bit5    PLLEN . 不用
    Bit4   MICBEN ,Microphone Bias Enable (MIC 偏置电路使能)
    Bit3   BIASEN ,Analogue amplifier bias control 必须设置为 1 模拟放大器才工
作
    Bit2    BUFIOEN , Unused input/output tie off buffer enable
    Bit1:0  VMIDSEL, 必须设置为非 00 值模拟放大器才工作
```

(下页继续)

(续上页)

```
    */
    usReg = (1 << 3) | (3 << 0);
    if (_OutPath & OUT3_4_ON) { /* OUT3 和 OUT4 使能输出 */
        usReg |= ((1 << 7) | (1 << 6));
    }
    if ((_InPath & MIC_LEFT_ON) || (_InPath & MIC_RIGHT_ON)) {
        usReg |= (1 << 4);
    }
    wm8978_WriteReg(1, usReg);  /* 写寄存器 */

    /*********************************************/
    /*          此处省略部分代码，具体参考工程文件          */
    /*********************************************/

    /*  R10 寄存器 DAC Control
    B8  0
    B7  0
    B6  SOFTMUTE, Softmute enable:
    B5  0
    B4  0
    B3  DACOSR128,  DAC oversampling rate: 0=64x (lowest power)
                                           1=128x (best performance)
    B2  AMUTE,    Automute enable
    B1  DACPOLR,  Right DAC output polarity
    B0  DACPOLL,  Left DAC output polarity:
    */
    if (_InPath & DAC_ON) {
        wm8978_WriteReg(10, 0);
    }
}
```

wm8978_CfgAudioPath 函数用于配置声音输入输出通道，有两个形参，第一个形参用于设置输

入源，可以使用 IN_PATH_E 枚举类型成员的一个或多个或运算结果；第二个形参用于设置输出通道，可以使用 OUT_PATH_E 枚举类型成员的一个或多个或运算结果。具体到开发板，如果进行录用功能，设置输入源为 (MIC_RIGHT_ON|ADC_ON) 或 (LINE_ON|ADC_ON)，设置输出端口为 OUT_PATH_OFF 或 (EAR_LEFT_ON | EAR_RIGHT_ON)；对于音乐播放功能，设置输入源为 DAC_ON，设置输出端口为 (EAR_LEFT_ON | EAR_RIGHT_ON)。

wm8978_CfgAudioPath 函数首先判断输入参数合法性，如果输入出错直接调用函数 wm8978_PowerDown 进入低功耗模式，并退出。

接下来使用 wm8978_WriteReg 配置相关寄存器值。大致可分三个部分，第一部分是电源管理部分，主要涉及到 R1、R2 和 R3 三个寄存器，使用输入输出通道之前必须开启相关电源。第二部分是输入通道选择及相关配置，配置 R44 控制选择输入通道，R14 设置输入的高通滤波器功能，R27、R28、R29 和 R30 设置输入的可调陷波滤波器功能，R32、R33 和 R34 控制输入限幅器/电平自动控制 (ALC)，R35 设置 ALC 噪声门限，R47 和 R48 设置通道增益参数，R15 和 R16 设置 ADC 数字音量，R43 设置 AUXR 功能。第三部分是输出通道选择及相关配置，控制 R49 选择输出通道，R50 和 R51 设置左右通道混合输出效果，R56 设置 OUT3 混合输出效果，R57 设置 OUT4 混合输出效果，R11 和 R12 设置左右 DAC 数字音量，R10 设置 DAC 参数。

38.6.2.1.9 录音放音设置

列表 10: 代码清单 36_0_10 wm8978_CtrlGPIO1 函数

```
/**
* @brief  控制 WM8978 的 GPIO1 引脚输出 0 或 1,
*     控制模拟开关来切换录音放音
*     1: 放音
*     0: 录音
* @param  _ucValue : GPIO1 输出值, 0 或 1
* @retval 无
*/
void wm8978_CtrlGPIO1(uint8_t _ucValue)
{
```

(下页继续)

(续上页)

```
    uint16_t usRegValue;

    /* R8,  pdf 62 页 */
    if (_ucValue == 0) { /* 输出 0 */
        usRegValue = 6; /* B2:0 = 110 */
    } else {
        usRegValue = 7; /* B2:0 = 111 */
    }
    wm8978_WriteReg(8, usRegValue);
}
```

由于 F407 的 I2S 是半双工，因此 wm8978_CtrlGPIO1 函数用于控制一个模拟开关来控制 WM8978 是录音还是放音，置高电平是放音，置低电平是录音。

38.6.2.1.10 软件复位

列表 11: 代码清单 36_0_11 wm8978_Reset 函数

```
uint8_t wm8978_Reset(void)
{
    /* wm8978 寄存器缺省值 */
    const uint16_t reg_default[] = {
        0x000, 0x000, 0x000, 0x000, 0x050, 0x000, 0x140, 0x000,
        0x000, 0x000, 0x000, 0x0FF, 0x0FF, 0x000, 0x100, 0x0FF,
        0x0FF, 0x000, 0x12C, 0x02C, 0x02C, 0x02C, 0x02C, 0x000,
        0x032, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
        0x038, 0x00B, 0x032, 0x000, 0x008, 0x00C, 0x093, 0x0E9,
        0x000, 0x000, 0x000, 0x000, 0x003, 0x010, 0x010, 0x100,
        0x100, 0x002, 0x001, 0x001, 0x039, 0x039, 0x039, 0x039,
        0x001, 0x001
    };
```

(下页继续)

(续上页)

```
    uint8_t res;
    uint8_t i;

    res=wm8978_WriteReg(0x00, 0);

    for (i = 0; i < sizeof(reg_default) / 2; i++) {
        wm8978_RegCash[i] = reg_default[i];
    }
    return res;
}
```

wm8978_Reset 函数用于软件复位 WM8978 芯片，通过写入 R0 完成，使其寄存器复位到缺省状态，同时会更新寄存器缓冲区数组 wm8978_RegCash 恢复到缺省状态。

38.6.2.2 I2S 控制接口

WM8978 集成 I2S 音频接口，用于与外部设备进行数字音频数据传输，芯片 I2S 接口属性通过 wm8978_CfgAudioIF 函数配置。STM32 控制器与 WM8978 进行音频数据传输，一般设置 STM32 控制器为主机模式，WM8978 作为从设备。

I2S_GPIO_Config 函数用于初始化 I2S 相关 GPIO，具体参考工程文件。

38.6.2.2.1 I2S 工作模式配置

列表 12: 代码清单 36_0_12 I2Sx_Mode_Config 函数

```
void I2Sx_Mode_Config(const uint16_t _usStandard,const uint16_t
                                _usWordLen,const uint32_t _usAudioFreq){

    /* PLL 时钟根据 AudioFreq 设置 (44.1khz vs 48khz groups) */
    BSP_AUDIO_OUT_ClockConfig(&I2S_InitStructure,_usAudioFreq, NULL);
```

(下页继续)

(续上页)

```
    /* Clock config is shared between AUDIO IN and OUT   */

    /* 打开 I2S2 APB1 时钟 */
    WM8978_CLK_ENABLE();

    /* 复位 SPI2 外设到缺省状态 */
    HAL_I2S_DeInit(&I2S_InitStructure);

    /* I2S2 外设配置 */
    I2S_InitStructure.Instance = WM8978_I2Sx_SPI;
    I2S_InitStructure.Init.ClockSource=RCC_I2SCLKSOURCE_PLLI2S;
    I2S_InitStructure.Init.Mode = I2S_MODE_MASTER_TX;/* 配置 I2S 工作模式 */
    I2S_InitStructure.Init.Standard = _usStandard;    /* 接口标准 */
    I2S_InitStructure.Init.DataFormat = _usWordLen; /* 数据格式，16bit */
    I2S_InitStructure.Init.MCLKOutput = I2S_MCLKOUTPUT_ENABLE;/* 主时钟模式␣
↪*/
    I2S_InitStructure.Init.AudioFreq = _usAudioFreq; /* 音频采样频率 */
    I2S_InitStructure.Init.CPOL = I2S_CPOL_LOW;
    HAL_I2S_Init(&I2S_InitStructure);

    /* 使能 SPI2/I2S2 外设 */
    __HAL_I2S_ENABLE(&I2S_InitStructure);
}
```

I2Sx_Mode_Config 函数用于配置 STM32 控制器的 I2S 接口工作模式，它有三个形参，第一个为指定 I2S 接口标准，一般设置为 I2S Philips 标准，第二个为字长设置，一般设置为 16bit，第三个为采样频率，一般设置为 44KHz 既可得到高音质效果。

首先是时钟配置，使用 BSP_AUDIO_OUT_ClockConfig 函数选择 I2S 时钟源，一般选择内部 PLLI2S 时钟，使能 PLLI2S 时钟，并等待时钟正常后使用 WM8978_CLK_ENABLE 函数开启 I2S 外设时钟。

接下来通过给 I2S_InitTypeDef 结构体类型变量赋值设置 I2S 工作模式，并由 HAL_I2S_Init 函数

完成 I2S 基本工作环境配置。

最后，__HAL_I2S_ENABLE 函数用于使能 I2S。

38.6.2.2.2 I2S 数据发送 (DMA 传输)

列表 13: 代码清单 36_0_13 I2Sx_TX_DMA_Init 函数

```
/**
* @brief  I2Sx TX DMA 配置，设置为双缓冲模式，并开启 DMA 传输完成中断
* @param  buf0:M0AR 地址.
* @param  buf1:M1AR 地址.
* @param  num: 每次传输数据量 (以两个字节算的一个传输数据量，因为数据长度为 HalfWord)
* @retval 无
*/
void I2Sx_TX_DMA_Init(const uint32_t buffer0,const uint32_t buffer1,const uint32_t num)
{
    DMA_HandleTypeDef  DMA_InitStructure;

    I2Sx_DMA_CLK_ENABLE();//DMA1 时钟使能

    //清空 DMA1_Stream4 上所有中断标志
    __HAL_DMA_CLEAR_FLAG(&DMA_InitStructure,DMA_FLAG_FEIF0_4 |
    DMA_FLAG_DMEIF0_4 |  DMA_FLAG_TEIF0_4 | DMA_FLAG_HTIF0_4 |
    DMA_FLAG_TCIF0_4);
    /* 配置 DMA Stream */
    hdma_spi2_tx.Instance =I2Sx_TX_DMA_STREAM;
    hdma_spi2_tx.Init.Channel = I2Sx_TX_DMA_CHANNEL;  //通道 0 SPIx_TX 通道
    hdma_spi2_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;//存储器到外设模式
    hdma_spi2_tx.Init.PeriphInc = DMA_PINC_DISABLE;//外设非增量模式
    hdma_spi2_tx.Init.MemInc = DMA_MINC_ENABLE;//存储器增量模式
    hdma_spi2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;  //外
设数据长度:16 位
```

(下页继续)

(续上页)

```
    hdma_spi2_tx.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;//存储器数
据长度：16 位
    hdma_spi2_tx.Init.Mode = DMA_CIRCULAR;// 使用循环模式
    hdma_spi2_tx.Init.Priority = DMA_PRIORITY_HIGH;//高优先级
    hdma_spi2_tx.Init.FIFOMode = DMA_FIFOMODE_DISABLE; //不使用 FIFO 模式
    hdma_spi2_tx.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
    hdma_spi2_tx.Init.MemBurst = DMA_MBURST_SINGLE;//外设突发单次传输
    hdma_spi2_tx.Init.PeriphBurst = DMA_PBURST_SINGLE;//存储器突发单次传输

    HAL_DMA_Init(&hdma_spi2_tx);//初始化 DMA Stream

    HAL_DMAEx_MultiBufferStart_IT(&hdma_spi2_tx,(uint32_t)buffer0,
        (uint32_t)&(WM8978_I2Sx_SPI->DR),(uint32_t)buffer1,num);

    __HAL_LINKDMA(&I2S_InitStructure,hdmatx,hdma_spi2_tx);

    /* I2S 中断的 NVIC 配置 */
    HAL_NVIC_SetPriority(SPI2_IRQn, 0, 3);
    HAL_NVIC_EnableIRQ(SPI2_IRQn);

    HAL_NVIC_SetPriority(I2Sx_TX_DMA_STREAM_IRQn,0,1);
    HAL_NVIC_EnableIRQ(I2Sx_TX_DMA_STREAM_IRQn);
}
```

I2Sx_TX_DMA_Init 函数用于初始化 I2S 数据发送 DMA 请求工作环境，并启动 DMA 传输。它有三个形参，第一个为缓冲区 1 地址，第二个为缓冲区 2 地址，第三为缓冲区大小。这里使用 DMA 的双缓冲区模式，就是开辟两个缓冲区空间，当第一个缓冲区用于 DMA 传输时 (不占用 CPU)，CPU 可以往第二个缓冲区填充数据，等到第一个缓冲区 DMA 传输完成后切换第二个缓冲区用于 DMA 传输，CPU 往第一个缓冲区填充数据，如此不断循环切换，可以达到 DMA 数据传输不间断效果，具体可参考 DMA 章节。这里为保证播放流畅性使用了 DMA 双缓冲区模式。

I2Sx_TX_DMA_Init 函数首先是使能 I2S 发送 DMA 流时钟，并复位 DMA 流配置和相关中断标志位。

通过对 DMA_HandleTypeDef 结构体类型的变量赋值配置 DMA 流工作环境并通过 HAL_DMA_Init 完成配置。

HAL_DMAEx_MultiBufferStart_IT 函数用于指定 DMA 双缓冲区模式下缓冲区地址。这里使能 DMA 传输完成中，用于指示其中一个缓冲区传输完成，需要切换缓冲区，可以开始往缓冲区填充数据。

__HAL_LINKDMA 用于 DMA 关联 I2S 外设。

最后配置 DMA 传输完成中断的优先级。

38.6.2.2.3 DMA 数据发送传输完成中断服务函数

列表 14: 代码清单 36_0_14 DMA 数据发送传输完成中断服务函数

```
/**
* @brief  SPIx_TX_DMA_STREAM 中断服务函数
* @param  无
* @retval 无
*/
void I2Sx_TX_DMA_STREAM_IRQFUN(void)
{
    //执行回调函数，读取数据等操作在这里面处理
    hdma_spi2_tx.XferCpltCallback = I2S_DMAConvCplt;
    hdma_spi2_tx.XferM1CpltCallback = I2S_DMAConvCplt;
    HAL_DMA_IRQHandler(&hdma_spi2_tx);
}
```

I2Sx_TX_DMA_STREAM_IRQFUN 函数是 I2S 的 DMA 传输中断服务函数，在判断是 DMA 传输完成中断后执行 I2S_DMAConvCplt 函数指针对应函数内容。

38.6.2.2.4 启动和停止播放控制

列表 15: 代码清单 36_0_15 启动和停止播放控制

```
/**
* @brief  I2S 开始播放
* @param  无
* @retval 无
*/
void I2S_Play_Start(void)
{
    //开启 DMA TX 发送请求，开始播放
    I2S_InitStructure.Instance->CR2 |= SPI_CR2_TXDMAEN;
```

(下页继续)

(续上页)

```
}


/**
* @brief  关闭 I2S 播放
* @param  无
* @retval 无
*/
void I2S_Play_Stop(void)
{
    //关闭 DMA TX 传输，结束播放
    HAL_I2S_DMAStop(&I2S_InitStructure);
}
```

I2S_Play_Start 用于开始播放，I2S_Play_Stop 用于停止播放，实际是通过控制 DMA 传输使能来实现。

38.6.2.2.5 I2S 录音功能模式配置

列表 16: 代码清单 36_0_16 I2Sxext_Mode_Config 函数

```
/**
* @brief  配置 STM32 的 I2S 外设工作模式
* @param  _usStandard : 接口标准，I2S_Standard_Phillips，I2S_Standard_MSB 或␣
↪I2S_Standard_LSB
* @param  _usWordlen : 数据格式，16bit 或者 24bit
* @param  _usAudioFreq : 采样频率，I2S_AudioFreq_8K、I2S_AudioFreq_16K、
I2S_AudioFreq_22K、
*                       I2S_AudioFreq_44K、I2S_AudioFreq_48
* @retval 无
*/
void I2Sxext_Mode_Config(const uint16_t _usStandard, const uint16_t
```

(下页继续)

(续上页)

```
_usWordLen,const uint32_t _usAudioFreq){

    /* 打开 I2S2 APB1 时钟 */
    WM8978_CLK_ENABLE();

    /* I2S2 外设配置 */
    /* 复位 SPI2 外设到缺省状态 */
    HAL_I2S_DeInit(&I2Sext_InitStructure);

    /* I2S2 外设配置 */
    I2Sext_InitStructure.Instance = WM8978_I2Sx_SPI;
    I2Sext_InitStructure.Init.ClockSource=RCC_I2SCLKSOURCE_PLLI2S;
    I2Sext_InitStructure.Init.Mode = I2S_MODE_MASTER_RX; /* 配置 I2S 工作模
式 */
    I2Sext_InitStructure.Init.Standard = _usStandard;/* 接口标准 */
    I2Sext_InitStructure.Init.DataFormat = _usWordLen; /* 数据格式，16bit */
    I2Sext_InitStructure.Init.MCLKOutput =I2S_MCLKOUTPUT_ENABLE;/* 主时钟模
式 */
    I2Sext_InitStructure.Init.AudioFreq = _usAudioFreq; /* 音频采样频率 */
    I2Sext_InitStructure.Init.CPOL = I2S_CPOL_LOW;
    HAL_I2S_Init(&I2Sext_InitStructure);

    /* PLL clock is set depending on the AudioFreq (44.1khz vs 48khz␣
↪groups) */
    /* Clock config is shared between AUDIO IN and OUT */
    BSP_AUDIO_OUT_ClockConfig(&I2Sext_InitStructure,_usAudioFreq, NULL);

    /* 使能 SPI2/I2S2 外设 */
    __HAL_I2S_ENABLE(&I2Sext_InitStructure);
}
```

I2Sx_Mode_Config 函数与 I2Sx_Mode_Config 函数类似，只不过一个用于输入即录音一个用于输

出即放音。这里用于配置 STM32 控制器的 I2S 接口工作模式为录音，它有三个形参，第一个为指定 I2S 接口标准，一般设置为 I2S Philips 标准，第二个为字长设置，一般设置为 16bit，第三个为采样频率，一般设置为 44KHz 既可得到高音质效果。

首先是时钟配置，使用 BSP_AUDIO_OUT_ClockConfig 函数选择 I2S 时钟源，一般选择内部 PLLI2S 时钟，使能 PLLI2S 时钟，并等待时钟正常后使用 WM8978_CLK_ENABLE 函数开启 I2S 外设时钟。

接下来通过给 I2S_InitTypeDef 结构体类型变量赋值设置 I2S 工作模式，并由 HAL_I2S_Init 函数完成 I2S 基本工作环境配置。

最后，__HAL_I2S_ENABLE 函数用于使能 I2S。

38.6.2.2.6 I2S 扩展数据接收 (DMA 传输)

列表 17: 代码清单 36_0_17 I2Sxext_RX_DMA_Init 函数

```
/**
* @brief  I2Sxext RX DMA 配置，设置为双缓冲模式，并开启 DMA 传输完成中断
* @param  buf0:M0AR 地址.
* @param  buf1:M1AR 地址.
* @param  num: 每次传输数据量
* @retval 无
*/
void I2Sxext_RX_DMA_Init(const uint16_t *buffer0,
const uint16_t *buffer1,const uint32_t num){
    DMA_HandleTypeDef  DMA_RXInitStructure;

    I2Sx_DMA_CLK_ENABLE();//DMA1 时钟使能

    /* 配置 DMA Stream */
    hdma_spi2_rx.Instance =I2Sxext_RX_DMA_STREAM;
    hdma_spi2_rx.Init.Channel=I2Sxext_RX_DMA_CHANNEL; //通道 3 SPIx_RX 通道
```

(下页继续)

(续上页)

```
    hdma_spi2_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;//存储器到外设模式
    hdma_spi2_rx.Init.PeriphInc = DMA_PINC_DISABLE;//外设非增量模式
    hdma_spi2_rx.Init.MemInc = DMA_MINC_ENABLE;//存储器增量模式
    hdma_spi2_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;//外设数
据长度:16 位
    hdma_spi2_rx.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;//存储器数
据长度: 16 位
    hdma_spi2_rx.Init.Mode = DMA_CIRCULAR;// 使用循环模式
    hdma_spi2_rx.Init.Priority = DMA_PRIORITY_LOW;//高优先级
    hdma_spi2_rx.Init.FIFOMode = DMA_FIFOMODE_DISABLE; //不使用 FIFO 模式
    hdma_spi2_rx.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
    hdma_spi2_rx.Init.MemBurst = DMA_MBURST_SINGLE;//外设突发单次传输
    hdma_spi2_rx.Init.PeriphBurst = DMA_PBURST_SINGLE;//存储器突发单次传输
    __HAL_DMA_CLEAR_FLAG(&DMA_RXInitStructure,DMA_FLAG_FEIF3_7 |
    DMA_FLAG_DMEIF3_7 | DMA_FLAG_TEIF3_7 |
                    DMA_FLAG_HTIF3_7 | DMA_FLAG_TCIF3_7);
    HAL_DMA_Init(&hdma_spi2_rx);//初始化 DMA Stream

    HAL_DMAEx_MultiBufferStart_IT(&hdma_spi2_rx,(uint32_t)&
    (WM8978_I2Sx_SPI->DR),(uint32_t)buffer0,(uint32_t)buffer1,num);

    __HAL_LINKDMA(&I2Sext_InitStructure,hdmarx,hdma_spi2_rx);

    /* NVIC configuration for I2S interrupts */
    HAL_NVIC_SetPriority(SPI2_IRQn, 0, 3);
    HAL_NVIC_EnableIRQ(SPI2_IRQn);

    HAL_NVIC_SetPriority(I2Sxext_RX_DMA_STREAM_IRQn,0,0);
    HAL_NVIC_EnableIRQ(I2Sxext_RX_DMA_STREAM_IRQn);
}
```

I2Sxext_RX_DMA_Init 函数配置 I2S 的数据接收功能，使用 DMA 传输方式接收数据，程序结构

与 I2Sx_TX_DMA_Init 函数一致，只是 DMA 传输方向不同，I2Sxext_RX_DMA_Init 函数是从外设到存储器传输，I2Sx_TX_DMA_Init 函数是存储器到外设传输。

I2Sxext_RX_DMA_Init 函数也是使用 DMA 的双缓冲区模式传输数据。最后使能了 DMA 传输完成中断，并使能 DMA 数据接收请求。

38.6.2.2.7 DMA 数据接收传输完成中断服务函数

列表 18: 代码清单 36_0_18 DMA 数据接收传输完成中断服务函数

```
/**
* @brief  I2Sxext_RX_DMA_STREAM 中断服务函数
* @param  无
* @retval 无
*/
void I2Sxext_RX_DMA_STREAM_IRQFUN(void)
{
    //执行回调函数，读取数据等操作在这里面处理
    hdma_spi2_rx.XferCpltCallback = I2Sxext_DMAConvCplt;
    hdma_spi2_rx.XferM1CpltCallback = I2Sxext_DMAConvCplt;
    HAL_DMA_IRQHandler(&hdma_spi2_rx);
}
```

与 DMA 数据发送传输完成中断服务函数类似，I2Sxext_RX_DMA_STREAM_IRQFUN 函数在判断得到是数据接收传输完成后执行 I2Sxext_DMAConvCplt 函数，I2Sxext_DMAConvCplt 实际也是一个函数指针。

38.6.2.2.8 启动和停止录音

列表 19: 代码清单 36_0_19 启动和停止录音

```
/**
* @brief  I2S 开始录音
* @param  无
* @retval 无
*/
void I2Sxext_Recorde_Start(void)
{
    //开启 DMA RX 传输, 开始录音
    I2Sext_InitStructure.Instance->CR2 |= SPI_CR2_RXDMAEN;
}

/**
* @brief  关闭 I2S 录音
* @param  无
* @retval 无
*/
void I2Sxext_Recorde_Stop(void)
{
    HAL_I2S_DMAStop(&I2Sext_InitStructure);
}
```

I2Sxext_Recorde_Start 函数用于启动录音，I2Sxext_Recorde_Stop 函数用于停止录音，实际是通过控制 DMA 传输使能来实现。

至此，关于 WM8978 芯片的驱动程序已经介绍完整了，该部分程序都是在 bsp_wm8978.c 文件中的，接下来我们就可以使用这些驱动程序实现录音和回放功能了。

38.6.2.3 录音和回放功能

录音和回放功能是在 WM8978 驱动函数基础上搭建而成的，实现代码存放在 Recorder.c 和 Recorder.h 文件中。启动录音功能后会在 SD 卡内创建一个 WAV 格式文件，把音频数据保存在该文件中，录音结束后既可得到一个完整的 WAV 格式文件。回放功能用于播放录音文件，实际上回放功能的实现函数也是适用于播放其他 WAV 格式文件的。

38.6.2.3.1 枚举和结构体类型定义

列表 20: 代码清单 36_0_20 枚举和结构体类型定义

```
/* 录音机状态 */
enum {
    STA_IDLE = 0, /* 待机状态 */
    STA_RECORDING,  /* 录音状态 */
    STA_PLAYING,  /* 放音状态 */
    STA_ERR,      /*  error  */
};

typedef struct {
    uint8_t ucInput;      /* 输入源: 0:MIC, 1: 线输入 */
    uint8_t ucFmtIdx;     /* 音频格式: 标准、位长、采样频率 */
    uint8_t ucVolume;     /* 当前放音音量 */
    uint8_t ucGain;       /* 当前增益 */
    uint8_t ucStatus;     /* 录音机状态, 0 表示待机, 1 表示录音中, 2 表示播放中 */
} REC_TYPE;

/* WAV 文件头格式 */
typedef __packed struct {
    uint32_t  riff;             /* = "RIFF" 0x46464952*/
    uint32_t  size_8;           /* 从下个地址开始到文件尾的总字节数 */
    uint32_t  wave;             /* = "WAVE" 0x45564157*/
```

(下页继续)

(续上页)

```
    uint32_t   fmt;              /* = "fmt " 0x20746d66*/
    uint32_t   fmtSize;          /* 下一个结构体的大小 (一般为 16) */
    uint16_t   wFormatTag;       /* 编码方式, 一般为 1  */
    uint16_t   wChannels;        /* 通道数, 单声道为 1, 立体声为 2 */
    uint32_t   dwSamplesPerSec;  /* 采样率 */
    uint32_t   dwAvgBytesPerSec; /* 每秒字节数 (= 采样率 × 每个采样点字节数) */
    uint16_t   wBlockAlign;      /* 每个采样点字节数 (= 量化比特数/8* 通道数) */
    uint16_t   wBitsPerSample;   /* 量化比特数 (每个采样需要的 bit 数) */

    uint32_t   data;             /* = "data" 0x61746164*/
    uint32_t   datasize;         /* 纯数据长度 */
} WavHead;
```

首先，定义一个枚举类型罗列录音和回放功能的状态，录音和回放功能是不能同时使用的，使用枚举类型区分非常有效。

REC_TYPE 结构体类型定义了录音和回放功能相关可控参数，包括 WM8978 声音源输入端，可选板载咪头或板载耳机插座的 LINE 线输入；音频格式选择，一般选择 I2S Philips 标准、16bit 字长、44KHz 采样频率；音频输出耳机音量控制；录音时声音增益；当前状态。

WavHead 结构体类型定义了 WAV 格式文件头，具体参考“WAV 格式文件”，这里没有用到 fact chunk。需要注意的是这里使用 __packed 关键字，它表示结构字节对齐。

38.6.2.3.2 启动播放 WAV 格式音频文件

列表 21: 代码清单 36_0_21 StartPlay 函数

```
static void StartPlay(const char *filename)
{
    printf("当前播放文件 -> %s\n",filename);
```

(下页继续)

(续上页)

```
    result=f_open(&file,filename,FA_READ);
    if (result!=FR_OK) {
        printf("打开音频文件失败!!!->%d\r\n",result);
        result = f_close (&file);
        Recorder.ucStatus = STA_ERR;
        return;
    }
    //读取 WAV 文件头
    result = f_read(&file,&rec_wav,sizeof(rec_wav),&bw);
    //先读取音频数据到缓冲区
    result = f_read(&file,(uint16_t *)buffer0,RECBUFFER_SIZE*2,&bw);
    result = f_read(&file,(uint16_t *)buffer1,RECBUFFER_SIZE*2,&bw);

    Delay_ms(10); /* 延迟一段时间，等待 I2S 中断结束 */
    I2S_Stop();      /* 停止 I2S 录音和放音 */
    wm8978_Reset();    /* 复位 WM8978 到复位状态 */

    Recorder.ucStatus = STA_PLAYING;     /* 放音状态 */

    /* 配置 WM8978 芯片，输入为 DAC，输出为耳机 */
    wm8978_CfgAudioPath(DAC_ON, EAR_LEFT_ON | EAR_RIGHT_ON);
    /* 调节音量，左右相同音量 */
    wm8978_SetOUT1Volume(Recorder.ucVolume);
    /* 配置 WM8978 音频接口为飞利浦标准 I2S 接口，16bit */
    wm8978_CfgAudioIF(I2S_Standard_Phillips, 16);

    I2Sx_Mode_Config(g_FmtList[Recorder.ucFmtIdx][0],
    g_FmtList[Recorder.ucFmtIdx][1],g_FmtList[Recorder.ucFmtIdx][2]);
    I2Sxext_Mode_Config(g_FmtList[Recorder.ucFmtIdx][0],
    g_FmtList[Recorder.ucFmtIdx][1],g_FmtList[Recorder.ucFmtIdx][2]);

    I2Sxext_RX_DMA_Init(&recplaybuf[0],&recplaybuf[1],1);
```

(下页继续)

(续上页)

```
    DMA_ITConfig(I2Sxext_RX_DMA_STREAM,DMA_IT_TC,DISABLE);//开启传输完成中断
    I2Sxext_Recorde_Stop();

    I2Sx_TX_DMA_Init(buffer0,buffer1,RECBUFFER_SIZE);
    I2S_Play_Start();
}
```

StartPlay 函数用于启动播放 WAV 格式音频文件，它有一个形参，用于指示待播放文件名称。函数首先检查待播放文件是否可以正常打开，如果打开失败则退出播放。如果可以正常打开文件则先读取 WAV 格式文件头，保存在 WavHead 结构体类型变量 rec_wav 中，同时先读取音频数据填充到两个缓冲区中，这两个缓冲区 buffer0 和 buffer1 是定义的全局数组变量，用于 DMA 双缓冲区模式。

接下来，配置 WM8978 的工作环境，首先停止 I2S 并复位 WM8978 芯片。这里是播放音频功能，所以设置 WM8978 的输入是 DAC，播放 I2S 接口接收到的音频数据，输出设置为耳机输出。STM32 控制器的 I2S 接口和 WM8978 的 I2S 接口都设置为 I2S Philips 标志、字长为 16bit。

然后，调用 I2Sx_TX_DMA_Init 配置 I2S 的 DMA 发送请求，并调用 I2S_Play_Start 函数使能 DMA 数据传输。

38.6.2.3.3 启动录音功能

列表 22: 代码清单 36_0_22 StartRecord 函数

```
static void StartRecord(const char *filename)
{
    printf("当前录音文件 -> %s\n",filename);
    result=f_open(&file,filename,FA_CREATE_ALWAYS|FA_WRITE);
    if (result!=FR_OK) {
        printf("Open wavfile fail!!!->%d\r\n",result);
        result = f_close (&file);
        Recorder.ucStatus = STA_ERR;
```

(下页继续)

(续上页)

```
        return;
    }

    // 写入 WAV 文件头，这里必须写入写入后文件指针自动偏移到 sizeof(rec_wav) 位置，
    // 接下来写入音频数据才符合格式要求。
    result=f_write(&file,(const void *)&rec_wav,sizeof(rec_wav),&bw);

    Delay_ms(10);    /* 延迟一段时间，等待 I2S 中断结束 */
    I2S_Stop();      /* 停止 I2S 录音和放音 */
    wm8978_Reset();    /* 复位 WM8978 到复位状态 */

    Recorder.ucStatus = STA_RECORDING;     /* 录音状态 */

    /* 调节放音音量，左右相同音量 */
    wm8978_SetOUT1Volume(Recorder.ucVolume);

    if (Recorder.ucInput == 1) { /* 线输入 */
        /* 配置 WM8978 芯片，输入为线输入，输出为耳机 */
        wm8978_CfgAudioPath(LINE_ON | ADC_ON, EAR_LEFT_ON | EAR_RIGHT_ON);
        wm8978_SetLineGain(Recorder.ucGain);
    } else { /* MIC 输入 */
        /* 配置 WM8978 芯片，输入为 Mic，输出为耳机 */
        wm8978_CfgAudioPath(MIC_RIGHT_ON|ADC_ON, EAR_LEFT_ON|EAR_RIGHT_ON);
        wm8978_SetMicGain(Recorder.ucGain);
    }

    /* 配置 WM8978 音频接口为飞利浦标准 I2S 接口，16bit */
    wm8978_CfgAudioIF(I2S_Standard_Phillips, 16);

    I2Sx_Mode_Config(g_FmtList[Recorder.ucFmtIdx][0],
        g_FmtList[Recorder.ucFmtIdx][1],g_FmtList[Recorder.ucFmtIdx][2]);
    I2Sxext_Mode_Config(g_FmtList[Recorder.ucFmtIdx][0],
```

(下页继续)

(续上页)

```
        g_FmtList[Recorder.ucFmtIdx][1],g_FmtList[Recorder.ucFmtIdx][2]);

    I2Sx_TX_DMA_Init(&recplaybuf[0],&recplaybuf[1],1);
    DMA_ITConfig(I2Sx_TX_DMA_STREAM,DMA_IT_TC,DISABLE);//开启传输完成中断

    I2S_DMA_RX_Callback=Recorder_I2S_DMA_RX_Callback;
    I2Sxext_RX_DMA_Init(buffer0,buffer1,RECBUFFER_SIZE);

    I2S_Play_Start();
    I2Sxext_Recorde_Start();
}
```

StartRecord 函数在结构上与 StartPlay 函数类似，它实现启动录音功能，它有一个形参，指示保存录音数据的文件名称。StartRecord 函数会首先创建录音文件，因为用到 FA_CREATE_ALWAYS 标志位，f_open 函数会总是创建新文件，如果已存在该文件则会覆盖原先的文件内容。

这些必须先写入 WAV 格式文件头数据，这样当前文件指针自动移动到文件头的下一个字节，即是存放音频数据的起始位置。

开发板支持 LINE 线输入和板载咪头输入，程序默认使用咪头输入，在录音同时使能耳机输出，这样录音时在耳机接口是有相同的声音输出的。

录音功能需要使能扩展 I2S。

38.6.2.3.4 录音和回放功能选择

列表 23: 代码清单 36_0_23 RecorderDemo 函数

```
void RecorderDemo(void)
{
    uint8_t i;
    uint8_t ucRefresh;  /* 通过串口打印相关信息标志 */
```

(下页继续)

(续上页)

```
    DIR dir;

    Recorder.ucStatus=STA_IDLE;     /* 开始设置为空闲状态  */
    Recorder.ucInput=0;             /* 缺省 MIC 输入  */
    Recorder.ucFmtIdx=3;    /* 缺省飞利浦 I2S 标准，16bit 数据长度，44K 采样率  */
    Recorder.ucVolume=35;           /* 缺省耳机音量  */
    if (Recorder.ucInput==0) { //MIC
        Recorder.ucGain=50;           /* 缺省 MIC 增益  */
        rec_wav.wChannels=1;          /* 缺省 MIC 单通道 */
    } else {                  //LINE
        Recorder.ucGain=6;            /* 缺省线路输入增益 */
        rec_wav.wChannels=2;          /* 缺省线路输入双声道 */
    }

    rec_wav.riff=0x46464952;        /*  “RIFF”; RIFF 标志 */
    rec_wav.size_8=0;               /* 文件长度，未确定 */
    rec_wav.wave=0x45564157;        /*  “WAVE”; WAVE 标志 */

    rec_wav.fmt=0x20746d66;         /*  “fmt ”; fmt 标志，最后一位为空 */
    rec_wav.fmtSize=16;             /* sizeof(PCMWAVEFORMAT) */
    rec_wav.wFormatTag=1;           /* 1 表示为 PCM 形式的声音数据 */
    /* 每样本的数据位数，表示每个声道中各个样本的数据位数。 */
    rec_wav.wBitsPerSample=16;
    /* 采样频率（每秒样本数） */
    rec_wav.dwSamplesPerSec=g_FmtList[Recorder.ucFmtIdx][2];
    /* 每秒数据量；其值为通道数 × 每秒数据位数 × 每样本的数据位数/ 8。 */
    rec_wav.dwAvgBytesPerSec=
    rec_wav.wChannels*rec_wav.dwSamplesPerSec*rec_wav.wBitsPerSample/8;
    /* 数据块的调整数（按字节算的），其值为通道数 × 每样本的数据位值/ 8。 */
    rec_wav.wBlockAlign=rec_wav.wChannels*rec_wav.wBitsPerSample/8;

    rec_wav.data=0x61746164;        /*  “data”; 数据标记符 */
```

(下页继续)

(续上页)

```
    rec_wav.datasize=0;                 /* 语音数据大小 目前未确定 */

    /*  如果路径不存在，创建文件夹  */
    result = f_opendir(&dir,RECORDERDIR);
    while (result != FR_OK) {
        f_mkdir(RECORDERDIR);
        result = f_opendir(&dir,RECORDERDIR);
    }

    /*  初始化并配置 I2S  */
    I2S_Stop();
    I2S_GPIO_Config();
    I2Sx_Mode_Config(g_FmtList[Recorder.ucFmtIdx][0],
    g_FmtList[Recorder.ucFmtIdx][1],g_FmtList[Recorder.ucFmtIdx][2]);
    I2Sxext_Mode_Config(g_FmtList[Recorder.ucFmtIdx][0],
    g_FmtList[Recorder.ucFmtIdx][1],g_FmtList[Recorder.ucFmtIdx][2]);

    I2S_DMA_TX_Callback=MusicPlayer_I2S_DMA_TX_Callback;
    I2S_Play_Stop();
    I2S_DMA_RX_Callback=Recorder_I2S_DMA_RX_Callback;
    I2Sxext_Recorde_Stop();

    ucRefresh = 1;
    bufflag=0;
    Isread=0;
    /* 进入主程序循环体 */
    while (1) {
        /* 如果使能串口打印标志则打印相关信息 */
        if (ucRefresh == 1) {
            DispStatus();    /* 显示当前状态，频率，音量等 */
            ucRefresh = 0;
        }
```

(下页继续)

(续上页)

```
        if (Recorder.ucStatus == STA_IDLE) {
            /*  KEY2 开始录音  */
            if (Key_Scan(KEY2_GPIO_PORT,KEY2_PIN)==KEY_ON) {
                /* 寻找合适文件名 */
                for (i=1; i<0xff; ++i) {
                    sprintf(recfilename,"0:/recorder/rec%03d.wav",i);
            result=f_open(&file,(const TCHAR *)recfilename,FA_READ);
                    if (result==FR_NO_FILE)break;
                }
                f_close(&file);

                if (i==0xff) {
                    Recorder.ucStatus =STA_ERR;
                    continue;
                }
                /* 开始录音 */
                StartRecord(recfilename);
                ucRefresh = 1;
            }
            /*  TouchPAD 开始回放录音  */
            if (TPAD_Scan(0)) {
                /* 开始回放 */
                StartPlay(recfilename);
                ucRefresh = 1;
            }
        } else {
            /*  KEY1 停止录音或回放  */
            if (Key_Scan(KEY1_GPIO_PORT,KEY1_PIN)==KEY_ON) {
                /* 对于录音，需要把 WAV 文件内容填充完整 */
                if (Recorder.ucStatus == STA_RECORDING) {
                        I2Sxext_Recorde_Stop();
                        I2S_Play_Stop();
```

(下页继续)

(续上页)

```
                        rec_wav.size_8=wavsize+36;
                        rec_wav.datasize=wavsize;
                        result=f_lseek(&file,0);
                        result=f_write(&file,(const void *)&rec_wav,
                                    sizeof(rec_wav),&bw);
                        result=f_close(&file);
                        printf("录音结束\r\n");
                    }
                    ucRefresh = 1;
                    Recorder.ucStatus = STA_IDLE;    /* 待机状态 */
                    I2S_Stop();    /* 停止 I2S 录音和放音 */
                    wm8978_Reset(); /* 复位 WM8978 到复位状态 */
                }
            }
            /* DMA 传输完成 */
            if (Isread==1) {
                Isread=0;
                switch (Recorder.ucStatus) {
                case STA_RECORDING:  // 录音功能，写入数据到文件
                    if (bufflag==0)
                    result=f_write(&file,buffer0,RECBUFFER_SIZE*2,(UINT*)&
↪bw);
                    else
                    result=f_write(&file,buffer1,RECBUFFER_SIZE*2,(UINT*)&
↪bw);
                    wavsize+=RECBUFFER_SIZE*2;
                    break;
                case STA_PLAYING:    // 回放功能，读取数据到播放缓冲区
                    if (bufflag==0)
                        result = f_read(&file,buffer0,RECBUFFER_SIZE*2,&
↪bw);
                    else
```

(下页继续)

(续上页)

```
                        result = f_read(&file,buffer1,RECBUFFER_SIZE*2,&
↪bw);
                    /* 播放完成或读取出错停止工作 */
                    if ((result!=FR_OK)||(file.fptr==file.fsize)) {
                        printf("播放完或者读取出错退出...\r\n");
                        I2S_Play_Stop();
                        file.fptr=0;
                        f_close(&file);
                        Recorder.ucStatus = STA_IDLE;    /* 待机状态 */
                        I2S_Stop();    /* 停止 I2S 录音和放音 */
                        wm8978_Reset(); /* 复位 WM8978 到复位状态 */
                    }
                    break;
                }
            }
        }
    }
```

RecorderDemo 函数实现录音和回放功能。Recorder 是一个 REC_TYPE 结构体类型变量，指示录音和回放功能相关参数，这里通过赋值缺省选择板载咪头输入、使用 I2S Philips 标准、16bit 字长、44KHz 采样频率，并设置了音量和增益，对于 LINE 输入增益范围为 0~7。rec_wav 是 WavHead 结构体类型变量，用于设置 WAV 格式文件头，很多成员赋值为缺省值即可，成员 size_8 和 datasize 变量表示数据大小，因为录音时间长度直接影响这两个变量大小，现在并无法确定它们大小，需要在停止录音后才可计算得到它们的值。

接下来是使用 FatFs 的功能函数 f_opendir 和 f_mkdir 组合判断 SD 卡内是否有名为“recorder”的文件夹，如果没有改文件夹就创建它，因为我们打算把录音文件存放在该文件夹内。

接下来就是调用 I2S 相关函数完成 I2S 工作环境配置。

MusicPlayer_I2S_DMA_TX_Callback 是我们定义的一个函数的函数名，把他赋值给函数指针 I2S_DMA_TX_Callback，这样可以实现在执行 DMA 数据发送完成中断服务函数时执行 MusicPlayer_I2S_DMA_TX_Callback 函数。Recorder_I2S_DMA_RX_Callback 情况与之类似。开始时停

止录音和回放功能。

ucRefresh 变量作为通过串口打印相关操作和状态信息到串口调试助手“刷新”标志。bufflag 变量用于指示当前空闲缓冲区，工程定义了两个缓冲区 buffer0 和 buffer1 用于 DMA 双缓冲区模式，对于录音功能，buffag 为 0 表示当前 DMA 使用 buffer1 填充，buffer0 已经填充完整，为 1 表示当前 DMA 使用 buffer0 填充，buffer1 已经填充完整；对于回放功能，buffag 为 0 表示 buffer1 用于当前播放，buffer0 已经播放完成需要读取新数据填充，为 1 表示 buffer0 用于当前播放，buffer1 已经播放完成需要读取新数据填充。Isread 变量用于指示 DMA 传输完成状态，为 1 时表示 DMA 传输完成，只有在 DMA 传输完成中断服务函数中才会被置 1。

接下来就是无限循环函数了，先判断 ucRefresh 变量状态，如果为 1 就执行 DispStatus 函数，该函数只是使用 printf 函数打印相关信息。开发板集成有两个独立按键 K1 和 K2，还有一个电容按键，程序设置按下 K2 按键开始录音功能，触摸电容按键开始回放功能，按下 K1 按键用于停止录音和回放功能，这里 K2 按键和电容按键是互斥的。

在空闲状态下，允许按下 K2 按键启动录音，录音功能是通过调用 StartRecord 函数启动的，该函数需要一个参数指示录音文件名称，使用在执行 StartRecord 函数之前会循环使用 f_open 函数获取 SD 卡内不存在的文件，防止覆盖已存在文件。

在空闲状态下，允许触摸电容按键启动回放功能，它直接使用 StartPlay 函数启动上一个录音文件播放。

不在空闲状态下，按下 K1 按键可以停在录音或回放功能，回放功能只需要停在 I2S 的 DMA 数据发送接口并复位 WM8978 即可，录音功能需要停在扩展 I2S 的 DMA 数据接收功能，并且需要填充完整 WAV 格式文件头并写入到录音文件中。

无限循环还要不懂检测 Isread 变量的状态，当它在 DMA 传输完成中断服务函数被置 1 时说明双缓冲区状态发生改变，对于录音功能，需要把以填充满的缓冲区数据通过 f_write 写入到文件中；对于回放功能，需要利用 f_read 函数读取新数据填充到已播缓冲区中，如果遇到读取出错或文件已经播放完全需要停止 I2S 的 DMA 传输并复位 WM8978。

38.6.2.3.5 DMA 传输完成中断回调函数

列表 24: 代码清单 36_0_24 DMA 传输完成中断回调函数

```
/* DMA 发送完成中断回调函数 */
/* 缓冲区内容已经播放完成，需要切换缓冲区，进行新缓冲区内容播放
同时读取 WAV 文件数据填充到已播缓冲区  */
void MusicPlayer_I2S_DMA_TX_Callback(void)
{
    if (Recorder.ucStatus == STA_PLAYING) {
        if (I2Sx_TX_DMA_STREAM->CR&(1<<19)) { //当前使用 Memory1 数据
            bufflag=0;                          //可以将数据读取到缓冲区 0
        } else {                               //当前使用 Memory0 数据
            bufflag=1;                          //可以将数据读取到缓冲区 1
        }
        Isread=1;                              // DMA 传输完成标志
    }
}

/* DMA 接收完成中断回调函数 */
/* 录音数据已经填充满了一个缓冲区，需要切换缓冲区，
同时可以把已满的缓冲区内容写入到文件中 */
void Recorder_I2S_DMA_RX_Callback(void)
{
    if (Recorder.ucStatus == STA_RECORDING) {
        if (I2Sxext_RX_DMA_STREAM->CR&(1<<19)) { //当前使用 Memory1 数据
            bufflag=0;
        } else {                                //当前使用 Memory0 数据
            bufflag=1;
        }
        Isread=1;                               // DMA 传输完成标志
    }
```

(下页继续)

(续上页)

```
}
```

这两个函数用于在 DMA 传输完成后切换缓冲区。DMA 数据流 x 配置寄存器的 CT 位用于指示当前目标缓冲区，如果为 1，当前目标缓冲区为存储器 1；如果为 0，则为存储器 0。

38.6.2.3.6 主函数

列表 25: 代码清单 36_0_25 main 函数

```
int main(void)
{
    /* 配置系统时钟为 180 MHz */
    SystemClock_Config();
    /* 禁用 WiFi 模块 */
    WIFI_PDN_INIT();
    /* 初始化 LED */
    LED_GPIO_Config();
    LED_BLUE;
    /* 初始化触摸按键 */
    TPAD_Init();
    /* 初始化调试串口，一般为串口 1 */
    UARTx_Config();
    printf("WM8978 录音和回放功能\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
    res_sd = f_mount(&fs,"0:",1);

    if (res_sd!=FR_OK) {
        printf("!! SD 卡挂载文件系统失败。(%d)\r\n",res_sd);
        printf("!! 可能原因：SD 卡初始化不成功。\r\n");
```

(下页继续)

(续上页)

```
        while (1);
    } else {
        printf("SD 卡挂载成功\r\n");
    }
    /* 检测 WM8978 芯片，此函数会自动配置 CPU 的 GPIO */
    if (wm8978_Init()==0) {
        printf("检测不到 WM8978 芯片!!!\n");
        while (1);  /* 停机 */
    }
    printf("初始化 WM8978 成功\n");

    /* 录音与回放功能 */
    RecorderDemo();

    while (1) {
    }
}
```

main 函数主要完成各个外设的初始化，包括独立按键初始化、电容按键初始化、调试串口初始化、SD 卡文件系统挂载还有系统定时器初始化。

wm8978_Init 初始化 I2C 接口用于控制 WM8978 芯片，并复位 WM8978 芯片，如果初始化成功则运行 RecorderDemo 函数进行录音和回放功能测试。

38.6.3 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，将耳机插入到开发板上边沿左侧的耳机插座，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 K2 按键，开始执行录音功能测试，不断对着咪头说话，就可以把声音录制下来，按下 K1 按键可以停止录音。然后触摸电容按键就可以在耳机接口听到之前录音内容了，按下 K1 按键可停止播放。录音完成后也可以在电脑端打开 SD 卡，找到其中的录音文件，可在电

脑端音频播放器播放录音文件。

38.7 MP3 播放器

MP3 格式音乐文件普遍存在我们生活中，实际上 MP3 本身是一种音频编码方式，全称为 Moving Picture Experts Group Audio Layer III(MPEG Audio Layer 3)。MPEG 音频文件是 MPEG 标准中的声音部分，根据压缩质量和编码复杂程度划分为三层，即 Layer-1、Layer2、Layer3，且分别对应 MP1、MP2、MP3 这三种声音文件，其中 MP3 压缩率可达到 10:1 至 12:1，可以大大减少文件占用存储空间大小。MPEG 音频编码的层次越高，编码器越复杂，压缩率也越高。MP3 是利用人耳对高频声音信号不敏感的特性，将时域波形信号转换成频域信号，并划分成多个频段，对不同的频段使用不同的压缩率，对高频加大压缩比（甚至忽略信号）对低频信号使用小压缩比，保证信号不失真。这样一来就相当于抛弃人耳基本听不到的高频声音，只保留能听到的低频部分，这样可得到很高的压缩率。

38.7.1 MP3 文件结构

MP3 文件大致分为 3 个部分：TAG_V2（ID3V2），音频数据，TAG_V1(ID3V1)。ID3 是 MP3 文件中附加关于该 MP3 文件的歌手、标题、专辑名称、年代、风格等等信息，有两个版本 ID3V1 和 ID3V2。ID3V1 固定存放在 MP3 文件末尾，固定长度为 128 字节，以 TAG 三个字符开头，后面跟上歌曲信息。因为 ID3V1 可存储信息量有限，有些 MP3 文件添加了 ID3V2，ID3V2 是可选的，如果存在 ID3V2 那它必然存在在 MP3 文件起始位置，它实际是 ID3V1 的补充。

38.7.1.1 ID3V2

ID3V2 以灵活管理方法 MP3 文件附件信息，比 ID3V1 可以存储更多的信息，同时也比 ID3V1 在结构上复杂得多。常用是 ID3V2.3 版本，一个 MP3 文件最多就一个 ID3V2.3 标签。ID3V2.3 标签由一个标签头和若干个标签帧或一个扩展标签头组成。关于曲目的信息如标题、作者等都存放在不同的标签帧中，扩展标签头和标签帧并不是必要的，但每个标签至少要有一个标签帧。标签头和标签帧一起顺序存放在 MP3 文件的首部。

标签头结构如表 38-6。

表 38-38-6 标签头

地址	标识	字节	描述
00H	Header	3	字符串“ID3”
03H	Ver	1	版本号，ID3V2.3 就记录为 3
04H	Revision	1	副版本号，一般记录为 0
05H	Flag	1	标志字节，一般不用设置
06H	Size	4	标签大小，整个 ID3V2 所占空间字节的大小

其中标签大小计算如下：

Total_size=(Size[0]&0x7F)*0x200000+(Size[1]&0x7F)*0x400

+(Size[2]&0x7F)*0x80+(Size[3]&0x7F)

每个标签帧都有一个 10 个字节的帧头和至少一个字节的不定长度内容，在文件中是连续存放的。标签帧头由三个部分组成，即 Frame[4]、Size[4] 和 Flags[2]。Frame 是用四个字符表示帧内容含义，比如 TIT2 为标题、TPE1 为作者、TALB 为专辑、TRCK 为音轨、TYER 为年代等等信息。Size 用四个字节组成 32bit 数表示帧大小。Flags 是标签帧的标志位，一般为 0 即可。

38.7.1.2 ID3V1

ID3V1 是早期的版本，可以存放的信息量有限，但编程比 ID3V2 简单很多，即使到现在使用还是很多。ID3V1 是固定存放在 MP3 文件末尾的 128 字节，组成结构见表 36-0-7。

表 36-0-7 ID3V1 结构

字节	字长	描述
1-3	3	标识符：“TAG”
4-33	30	歌名
34-63	30	作者
64-93	30	专辑名
94-97	4	年份
98-127	30	附注
128	1	MP3 音乐类别

其中，歌名是固定分配为 30 个字节，如果歌名太短则以 0 填充完整，太长则被截断，其他信息类似情况存储。MP3 音乐类别总共有 147 种，每一种对应一个数组，比如 0 对应“Blues”、1 对应“Classic Rock”、2 对应“Country”等等。

38.7.1.3 MP3 数据帧

音频数据是 MP3 文件的主体部分，它由一系列数据帧 (Frame) 组成，每个 Frame 包含一段音频的压缩数据，通过解码库解码即可得到对应 PCM 音频数据，就可以通过 I2S 发送到 WM8978 芯片播放音乐，按顺序解码所有帧就可以得到整个 MP3 文件的音轨。

每个 Frame 由两部分组成，帧头和数据实体，Frame 长度可能不同，由位率决定。帧头记录了 MP3 数据帧的位率、采样率、版本等等信息，总共有 4 个字节，见表 36-0-8。

表 36-0-8 数据帧头结构

名称	位长	描述	
Sync	第1、2字节	11	同步信息：每位固定为：1。
Ver-sion		2	版本：00-MPEG2.5；0 1-未定义；10-MPEG2；11-MPEG1。
Layer		2	层：00-未定义；01-Lay er3；10-Layer2；1 1-Layer1。
Error Pro-tection		1	CRC 校验：0-校验；1-不校验。
Bitrate incdex	第 3 字节	4	位率：参考表 42-9。
Sam-pling fre-quency		2	采样频率： 对于 MPEG-1：00-44. 1kHz；01-48kHz；1 0-32kHz；11-未定义。 对于 MPEG-2：00-22. 05kHz；01-24kHz；10-16kHz；11-未定义。 对于 MPEG-2.5：00-1 1.025kHz；01-12k Hz；10-8kHz；11-未定义。
Padding		1	帧长调整：用来调整文件头长度，0-无需调整；1-调整。
Pri-vate		1	保留字。
Mode	第 4 字节	2	声道：00-立体声 Stereo ；01-Joint Stereo；10-双声道；1 1-单声道。
Mode exten-sion		2	扩充模式：当声道模式为 01 时才使用。
Copy-right		1	版权：文件是否合法，0-不合法；1-合法。
Origi-nal		1	原版标志：0-非原版；1-原版。
Em-phasis		2	强调方式：用于声音经降噪压缩后再补偿的分类，几乎不用。

位率位在不同版本和层都有不同的定义，具体参考表 36-0-9，单位为 kbps。其中 V1 对应 MPEG-1，V2 对应 MPEG-2 和 MPEG-2.5；L1 对应 Layer1，L2 对应 Layer2，L3 对应 Layer3，free 表示位率可变，bad 表示该定义不合法。

表 36-0-9 位率选择

bits	V1,L1	V1,L2	V1,L3	V2,L1	V2,L2	V2,L3
0000	free	free	free	free	free	free
0001	32	32	32	32(32)	32(8)	8(8)
0010	64	48	40	64(48)	48(16)	16(16)
0011	96	56	48	96(56)	56(24)	24(24)
0100	128	64	56	128(64)	64(32)	32(32)
0101	160	80	64	160(80)	80(40)	64(40)
0110	192	96	80	192(96)	96(48)	80(48)
0111	224	112	96	224(112)	112(56)	56(56)
1000	256	128	112	256(128)	128(64)	64(64)
1001	288	160	128	288(144)	160(80)	128(80)
1010	320	192	160	320(160)	192(96)	160(96)
1011	352	224	192	352(176)	224(112)	112(112)
1100	384	256	224	384(192)	256(128)	128(128)
1101	416	320	256	416(224)	320(144)	256(144)
1110	448	384	320	448(256)	384(160)	320(160)
1111	bad	bad	bad	bad	bad	bad

例如，当 Version=11，Layer=01，Bitrate_incdex=0101 时，描述为 MPEG-1 Layer3(MP3) 位率为 64kbps。

数据帧长度取决于位率 (Bitrate) 和采样频率 (Sampling_freq)，具体计算如下：

对于 MPEG-1 标准，Layer1 时：

Size=(48000*Bitrate)/Sampling_freq + Padding；

Layer2 或 Layer3 时：

Size=(144000*Bitrate)/Sampling_freq + Padding；

对于 MPEG-2 或 MPEG-2.5 标准，Layer1 时：

Size=(24000*Bitrate)/Sampling_freq + Padding；

Layer2 或 Layer3 时：

Size=(72000*Bitrate)/Sampling_freq + Padding；

如果有 CRC 校验，则存放在在帧头后两个字节。接下来就是帧主数据 (MAIN_DATA)。一般来说一个 MP3 文件每个帧的 Bitrate 是固定不变的，所以每个帧都有相同的长度，称为 CBR，还有小部分 MP3 文件的 Bitrate 是可变，称之为 VBR，它是 XING 公司推出的算法。

MAIN_DATA 保存的是经过压缩算法压缩后得到的数据，为得到大的压缩率会损失部分源声音，属于失真压缩。

38.7.2 MP3 解码库

MP3 文件是经过压缩算法压缩而存在的，为得到 PCM 信号，需要对 MP3 文件进行解码，解码过程大致为：比特流分析、霍夫曼编码、逆量化处理、立体声处理、频谱重排列、抗锯齿处理、IMDCT 变换、子带合成、PCM 输出。整个过程涉及很多算法计算，要自己编程实现不是一件现实的事情，还好有很多公司经过长期努力实现了解码库编程。

现在合适在小型嵌入式控制器移植运行的有两个版本的开源 MP3 解码库，分别为 Libmad 解码库和 Helix 解码库，Libmad 是一个高精度 MPEG 音频解码库，支持 MPEG-1、MPEG-2 以及 MPEG-2.5 标准，它可以提供 24bitPCM 输出，完全是定点计算，更多信息可参考网站：http://www.underbit.com/。

Helix 解码库支持浮点和定点计算实现，将该算法移植到 STM32 控制器运行使用定点计算实现，它支持 MPEG-1、MPEG-2 以及 MPEG-2.5 标准的 Layer3 解码。Helix 解码库支持可变位速率、恒定位速率，以及立体声和单声道音频格式。更多信息可参考网站：https://datatype.helixcommunity.org/Mp3dec。

因为 Helix 解码库需要占用的资源比 Libmad 解码库更少，特别是 RAM 空间的使用，这对 STM32 控制器来说是比较重要的，所以在实验工程中我们选择 Helix 解码库实现 MP3 文件解码。这两个解码库都是一帧为解码单位的，一次解码一帧，这在应用解码库时是需要着重注意的。

Helix 解码库涉及算法计算，整个界面过程复杂，有兴趣可以深入探究，这里我们着重讲解 Helix 移植和使用方法。

Helix 网站有提供解码库代码，经过整理，移植 Helix 解码库需要用到的的文件如图 *36_0_13*。有优化解码速度，部分解码过程使用汇编实现。

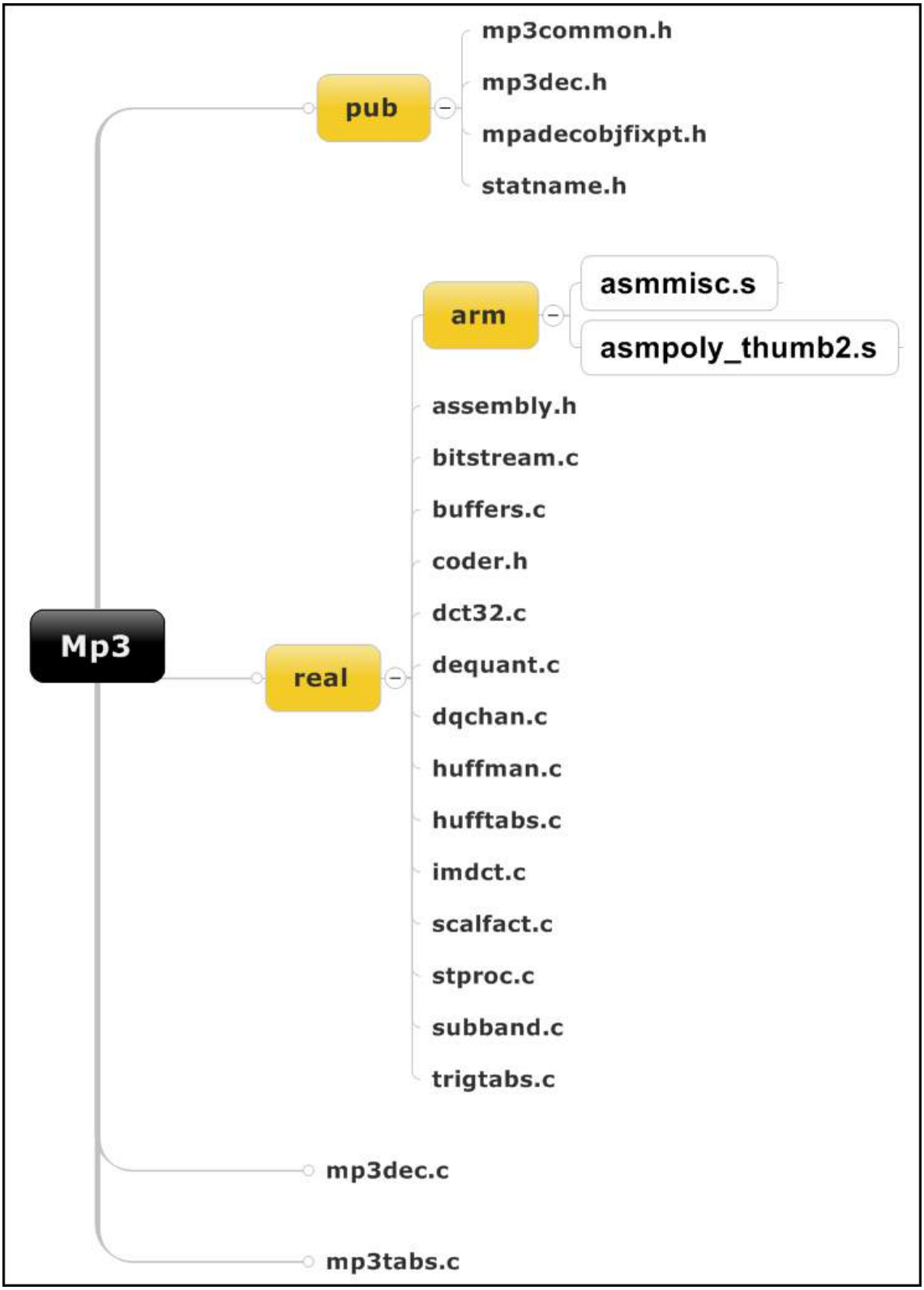

Mp3
pub
mp3common.h
mp3dec.h
mpadecobjfixpt.h
statname.h
real
arm
asmmisc.s
asmpoly_thumb2.s
assembly.h
bitstream.c
buffers.c
coder.h
dct32.c
dequant.c
dqchan.c
huffman.c
hufftabs.c
imdct.c
scalfact.c
stproc.c
subband.c
trigtabs.c
mp3dec.c
mp3tabs.c

图 36_0_13 Helix 解码库文件结构

38.7.3 Helix 解码库移植

在“录音与回放实验”已经实现了 WM8978 驱动代码，现在我们可以移植 Helix 解码库工程中，实现 MP3 文件解码，将解码输出的 PCM 数据通过 I2S 接口发送到 WM8978 芯片实现音乐播放。

我们在“录音与回放实验”工程文件基础上移植 Helix 解码，首先将需要用到的文件添加到工程中，如图 38-14。MP3 文件夹下文件是 Helix 解码库源码，工程移植中是不需要修改文件夹下代码的，我们只需直接调用相关解码函数即可。建议自己移植时直接使用例程中 mp3 文件夹内文件。我们是在 mp3Player.c 文件中调用 Helix 解码库相关函数实现 MP3 文件解码的，该文件是我们自己创建的。

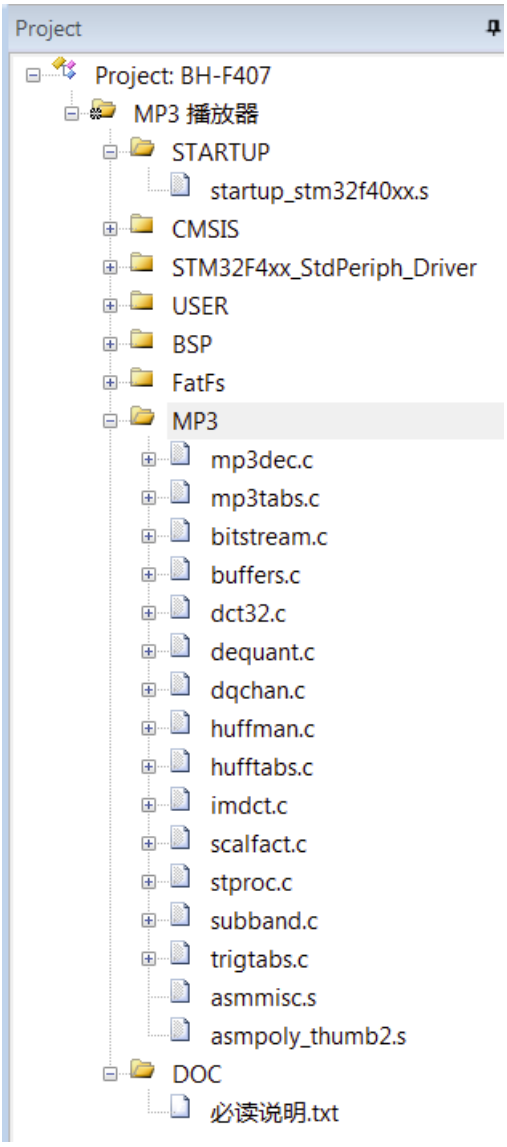


图 36_0_14 添加 Helix 解码库文件到工程

接下来还需要在工程选项中添加 Helix 解码库的文件夹路径，编译器可以寻找到相关头文件，见图 *36_0_15*。

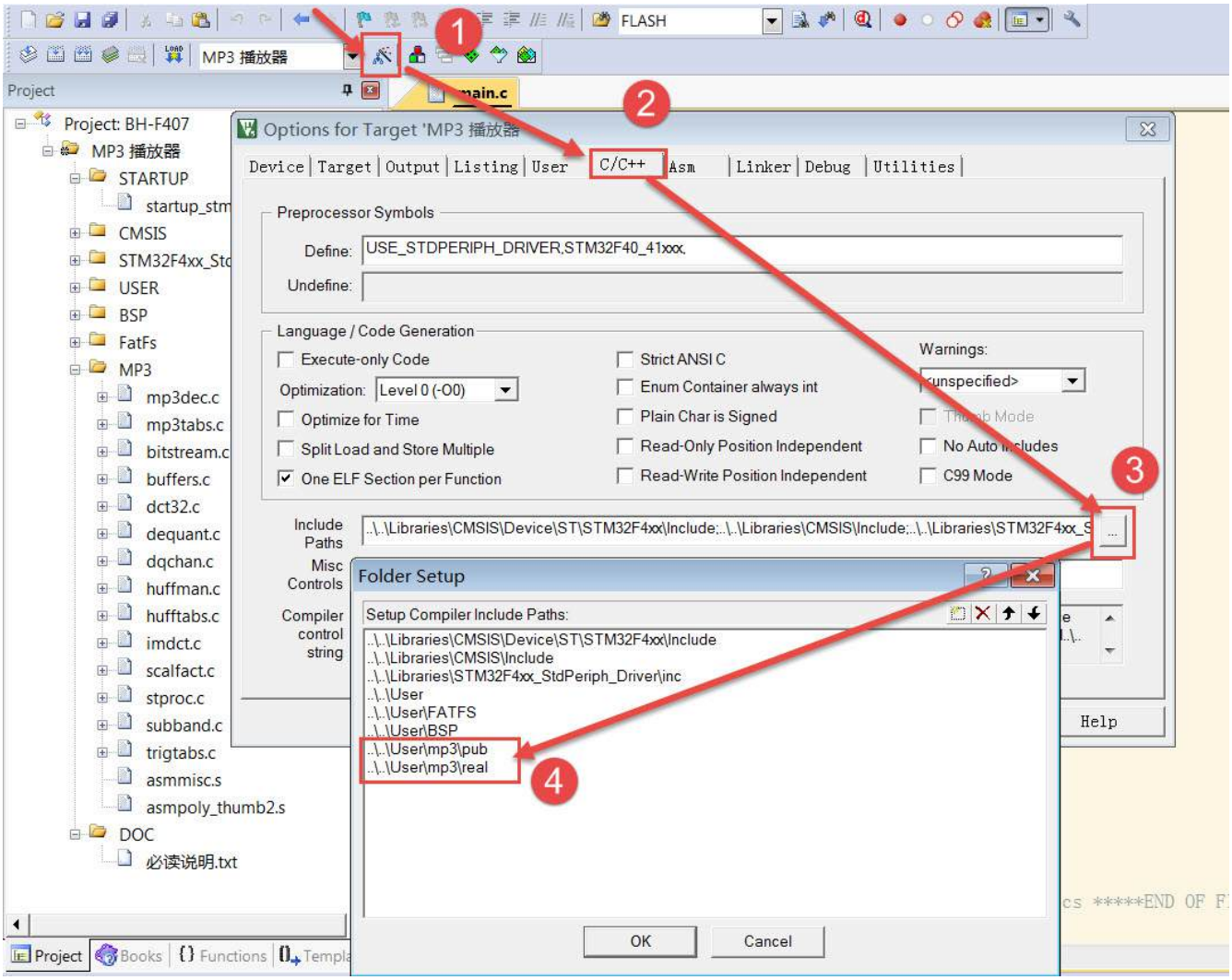


图 36_0_15 添加 Helix 解码库文件夹路径

38.7.4 MP3 播放器功能实现

“录音与回放实验”中的回放功能实际上就是从 SD 卡内读取 WAV 格式文件数据，然后提取里边音频数据通过 I2S 传输到 WM8978 芯片内实现声音播放。MP3 播放器的功能也是类似的，只不过现在音频数据提取方法不同，MP3 需要先经过解码库解码后才可得到“可直接”播放的音频数据。由此可以看到，MP3 播放器只是添加了 MP3 解码库实现代码，在硬件设计上并没有任何改变，即这里直接使用“录音与回放实验”中硬件设计即可。

实验工程代码中创建 mp3Player.c 和 mp3Player.h 两个文件存放 MP3 播放器实现代码。Helix 解码库是用来解码 MP3 数据帧，一次解码一帧，它是不能用来检索 ID3V1 和 ID3V2 标签的，如果需要获取歌名、作者等信息需要自己编程实现。解码过程可能用到的 Helix 解码库函数有：

- MP3InitDecoder：初始化解码器函数
- MP3FreeDecoder：关闭解码器函数
- MP3FindSyncWord：寻找帧同步函数
- MP3Decode：解码 MP3 帧函数
- MP3GetLastFrameInfo：获取帧信息函数

MP3InitDecoder 函数初始化解码器，它会申请分配一个存储空间用于存放解码器状态的一个数据结构并将其初始化，该数据结构由 MP3DecInfo 结构体定义，它封装了解码器内部运算数据信息。MP3InitDecoder 函数会返回指向该数据结构的指针。

MP3FreeDecoder 函数用于关闭解码器，释放由 MP3InitDecoder 函数申请的存储空间，所以一个 MP3InitDecoder 函数都需要有一个 MP3FreeDecoder 函数与之对应。它有一个形参，一般由 MP3InitDecoder 函数的返回指针赋值。

MP3FindSyncWord 函数用于寻址数据帧同步信息，实际上就是寻址数据帧开始的 11bit 都为“1”的同步信息。它有两个形参，第一个为源数据缓冲区指针，第二个为缓冲区大小，它会返回一个 int 类型变量，用于指示同步字较缓冲区起始地址的偏移量，如果在缓冲区中找不到同步字，则直接返回-1。

MP3Decode 函数用于解码数据帧，它有五个形参，第一个为解码器数据结构指针，一般由 MP3InitDecoder 函数返回值赋值；第二个参数为指向解码源数据缓冲区开始地址的一个指针，注意这里是地址的指针，即是指针的指针；第三个参数是一个指向存放解码源数据缓冲区有效数据量的变量指针；第四个参数是解码后输出 PCM 数据的指针，一般由我们定义的缓冲区地址赋值，对于双声道输出数据缓冲区以 LRLRLR…顺序排列；第五个参数是数据格式选择，一般设置为 0 表示标准的 MPEG 格式。函数还有一个返回值，用于返回解码错误，返回 ERR_MP3_NONE 说明解码正常。

MP3GetLastFrameInfo 函数用于获取数据帧信息，它有两个形参，第一个为解码器数据结构指针，一般由 MP3InitDecoder 函数返回值赋值；第二个参数为数据帧信息结构体指针，该结构体定义见*代码清单 36_0_26* MP3 数据帧信息结构体。

列表 26: 代码清单 36_0_26 MP3 数据帧信息结构体

```
typedef struct _MP3FrameInfo {
    int bitrate;          //位率
    int nChans;           //声道数
    int samprate;         //采样率
    int bitsPerSample;    //采样位数
    int outputSamps;      //PCM 数据数
    int layer;            //层级
    int version;          //版本
} MP3FrameInfo;
```

该结构体成员包括了该数据帧的位率、声道、采样频率等等信息，它实际上是从数据帧的帧头信息中提取的。

38.7.4.1 MP3 播放器实现

列表 27: 代码清单 36_0_27 mp3PlayerDemo 函数

```
void mp3PlayerDemo(const char *mp3file)
{
    uint8_t *read_ptr=inputbuf;
    uint32_t frames=0;
    int err=0, i=0, outputSamps=0;
    int read_offset = 0;          /* 读偏移指针 */
    int bytes_left = 0;           /* 剩余字节数 */

    mp3player.ucFreq=I2S_AudioFreq_Default;
    mp3player.ucStatus=STA_IDLE;
    mp3player.ucVolume=40;

    result=f_open(&file,mp3file,FA_READ);
```

(下页继续)

(续上页)

```
    if (result!=FR_OK) {
        printf("Open mp3file :%s fail!!!->%d\r\n",mp3file,result);
        result = f_close (&file);
        return; /* 停止播放 */
}
printf("当前播放文件 -> %s\n",mp3file);

//初始化 MP3 解码器
Mp3Decoder = MP3InitDecoder();
if (Mp3Decoder==0) {
    printf("初始化 helix 解码库设备\n");
    return; /* 停止播放 */
}
printf("初始化中...\n");

Delay_ms(10); /* 延迟一段时间，等待 I2S 中断结束 */
wm8978_Reset();   /* 复位 WM8978 到复位状态 */

/* 配置 WM8978 芯片，输入为 DAC，输出为耳机 */
wm8978_CfgAudioPath(DAC_ON, EAR_LEFT_ON | EAR_RIGHT_ON);

/* 调节音量，左右相同音量 */
wm8978_SetOUT1Volume(mp3player.ucVolume);

/* 配置 WM8978 音频接口为飞利浦标准 I2S 接口，16bit */
wm8978_CfgAudioIF(I2S_Standard_Phillips, 16);

/*  初始化并配置 I2S  */
I2S_Stop();
I2S_GPIO_Config();
I2Sx_Mode_Config(I2S_Standard_Phillips,
I2S_DataFormat_16b, mp3player.ucFreq);
```

(下页继续)

(续上页)

```
I2S_DMA_TX_Callback=MP3Player_I2S_DMA_TX_Callback;
Sx_TX_DMA_Init((uint16_t *)outbuffer[0],
             (uint16_t *)outbuffer[1], MP3BUFFER_SIZE);

bufflag=0;
Isread=0;

mp3player.ucStatus = STA_PLAYING;    /* 放音状态 */
result=f_read(&file,inputbuf,INPUTBUF_SIZE,&bw);
if (result!=FR_OK) {
    printf("读取%s 失败 -> %d\r\n",mp3file,result);
    MP3FreeDecoder(Mp3Decoder);
    return;
}
read_ptr=inputbuf;
bytes_left=bw;
/* 进入主程序循环体 */
while (mp3player.ucStatus == STA_PLAYING) {
    //寻找帧同步，返回第一个同步字的位置
    read_offset = MP3FindSyncWord(read_ptr, bytes_left);
    //没有找到同步字
    if (read_offset < 0) {
        result=f_read(&file,inputbuf,INPUTBUF_SIZE,&bw);
        if (result!=FR_OK) {
            printf("读取%s 失败 -> %d\r\n",mp3file,result);
            break;
        }
        read_ptr=inputbuf;
        bytes_left=bw;
        continue;
    }
```

(下页继续)

(续上页)

```
    read_ptr += read_offset;           //偏移至同步字的位置
    bytes_left -= read_offset;         //同步字之后的数据大小
    if (bytes_left < 1024) {             //补充数据
        /* 注意这个地方因为采用的是 DMA 读取，所以一定要 4 字节对齐  */
        i=(uint32_t)(bytes_left)&3;                    //判断多余的字节
        if (i) i=4-i;                                 //需要补充的字节
        memcpy(inputbuf+i, read_ptr, bytes_left); //从对齐位置开始复制
        read_ptr = inputbuf+i;                        //指向数据对齐位置
        //补充数据
        result=f_read(&file,inputbuf+bytes_left+i,
                                INPUTBUF_SIZE-bytes_left-i,&bw);
        bytes_left += bw;                        //有效数据流大小
    }
    //开始解码 参数：mp3 解码结构体、输入流指针、输入流大小、输出流指针、数据格式
    err = MP3Decode(Mp3Decoder, &read_ptr, &bytes_left,
                                outbuffer[bufflag], 0);
    frames++;
    //错误处理
    if (err != ERR_MP3_NONE) {
        switch (err) {
        case ERR_MP3_INDATA_UNDERFLOW:
            printf("ERR_MP3_INDATA_UNDERFLOW\r\n");
            result = f_read(&file, inputbuf, INPUTBUF_SIZE, &bw);
            read_ptr = inputbuf;
            bytes_left = bw;
            break;
        case ERR_MP3_MAINDATA_UNDERFLOW:
/* do nothing - next call to decode will provide more mainData */
            printf("ERR_MP3_MAINDATA_UNDERFLOW\r\n");
            break;
        default:
            printf("UNKNOWN ERROR:%d\r\n", err);
```

(下页继续)

(续上页)

```
            // 跳过此帧
            if (bytes_left > 0) {
                bytes_left --;
                read_ptr ++;
            }
            break;
        }
        Isread=1;
    } else {  //解码无错误，准备把数据输出到 PCM
    MP3GetLastFrameInfo(Mp3Decoder, &Mp3FrameInfo);    //获取解码信息
            /* 输出到 DAC */
    outputSamps = Mp3FrameInfo.outputSamps;             //PCM 数据个数
            if (outputSamps > 0) {
                if (Mp3FrameInfo.nChans == 1) { //单声道
                    //单声道数据需要复制一份到另一个声道
                    for (i = outputSamps - 1; i >= 0; i--) {
                outbuffer[bufflag][i * 2] = outbuffer[bufflag][i];
                outbuffer[bufflag][i * 2 + 1] = outbuffer[bufflag][i];
                    }
                    outputSamps *= 2;
                }//if (Mp3FrameInfo.nChans == 1)  //单声道
            }//if (outputSamps > 0)

            /* 根据解码信息设置采样率 */
            if (Mp3FrameInfo.samprate != mp3player.ucFreq) {  //采样率
            mp3player.ucFreq = Mp3FrameInfo.samprate;

printf(" \r\n Bitrate       %dKbps",Mp3FrameInfo.bitrate/1000);
            printf(" \r\n Samprate      %dHz", mp3player.ucFreq);
printf(" \r\n BitsPerSample %db", Mp3FrameInfo.bitsPerSample);
            printf(" \r\n nChans        %d", Mp3FrameInfo.nChans);
            printf(" \r\n Layer         %d", Mp3FrameInfo.layer);
```

(下页继续)

(续上页)

```
            printf(" \r\n Version        %d", Mp3FrameInfo.version);
            printf(" \r\n OutputSamps    %d", Mp3FrameInfo.outputSamps);
            printf("\r\n");
            //I2S_AudioFreq_Default = 2, 正常的帧, 每次都要改速率
            if (mp3player.ucFreq >= I2S_AudioFreq_Default) {
                    //根据采样率修改 I2S 速率
        I2Sx_Mode_Config(I2S_Standard_Phillips,I2S_DataFormat_16b,
                         mp3player.ucFreq);
        I2Sx_TX_DMA_Init((uint16_t *)outbuffer[0],
                        (uint16_t *)outbuffer[1],outputSamps);
            }
            I2S_Play_Start();
            }
        }//else 解码正常

        if (file.fptr==file.fsize) {  //mp3 文件读取完成, 退出
            printf("END\r\n");
            break;
        }

        while (Isread==0) {
        }
        Isread=0;
    }
    I2S_Stop();
    mp3player.ucStatus=STA_IDLE;
    MP3FreeDecoder(Mp3Decoder);
    f_close(&file);
}
```

mp3PlayerDemo 函数是 MP3 播放器的实现函数，篇幅很长，需要我们仔细分析。它有一个形参，用于指定待播放的 MP3 文件，需要用 MP3 文件的绝对路径加全名称赋值。

read_ptr 是定义的一个指针变量，它用于指示解码器源数据地址，把它初始化为用来存放解码器源数据缓冲区 (inputbuf 数组) 的首地址。read_offset 和 bytes_left 主要用于 MP3FindSyncWord 函数，read_offset 用来指示帧同步相对解码器源数据缓冲区首地址的偏移量，bytes_left 用于指示解码器源数据缓冲区有效数据量。

mp3player 是一个 MP3_TYPE 结构体类型变量，指示音量、状态和采样频率信息。

f_open 函数用于打开文件，如果文件打开失败则直接退出播放。MP3InitDecoder 函数用于初始化 Helix 解码器，分配解码器必须内存空间，如果初始化解码器失败直接退出播放。

接下来配置 WM8978 芯片功能，使能耳机输出，设置音量，使用 I2S Philips 标准和 16bit 数据长度。还要设置 I2S 外设工作环境，同样是 I2S Philips 标准和 16bit 数据长度，采样频率先使用 I2S_AudioFreq_Default，在运行 MP3GetLastFrameInfo 函数获取数据帧的采样频率后需要再次修改。MP3Player_I2S_DMA_TX_Callback 是一个函数名，该函数实现 DMA 双缓冲区标志位切换以及指示 DMA 传输完成，它在 I2S 的 DMA 发送传输完成中断服务函数中调用。I2Sx_TX_DMA_Init 用于初始化 I2S 的 DMA 发送请求，使用双缓冲区模式。bufflag 用于指示当前 DMA 传输的缓冲区号，Isread 用于指示 DMA 传输完成。

f_read 函数从 SD 卡读取 MP3 文件数据，存放在 inputbuf 缓冲区中，bw 变量保存实际读取到的数据的字节数。如果读取数据失败则运行 MP3FreeDecoder 函数关闭解码器后退出播放器。

接下来是循环解码帧数据并播放。MP3FindSyncWord 用于选择帧同步信息，如果在源数据缓冲区中找不到同步信息，read_offset 值为-1，需要读取新的 MP3 文件数据，重新寻找帧同步信息。一般 MP3 起始部分是 ID3V2 信息，所以可能需要循环几次才能寻找到帧同步信息。如果找到帧同步信息说明找到数据帧，接下来数据就是数据帧的帧头以及 MAIN_DATA。

有时找到了帧同步信息，但可能源数据缓冲区并没有包括整帧数据，这时需要把从帧同步信息开始的源数据，复制到源数据缓冲区起始地址上，再使用 f_read 函数读取新数据填充满整个源数据缓冲区，保证源数据缓冲区保存有整帧源数据。

MP3Decode 函数开始对源数据缓冲区中帧数据进行解码，通过函数返回值可判断得到解码状态，如果发生解码错误则执行对应的代码。

在解码无错误时，就可以使用 MP3GetLastFrameInfo 函数获取帧信息，如果有数据输出并且是单声道需要把数据复制成双声道数据格式。如果采样频率与上一次有所不同则执行通过串口打印帧信息到串口调试助手，可能还需要调整 I2S 的工作环境，还调用 I2S_Play_Start 启动播放。因

为 mp3player.ucFreq 缺省值为 I2S_AudioFreq_Default，一般都不会与 Mp3FrameInfo.samprate 相等，所以会至少进入 if 语句内，即会执行 I2S_Play_Start 函数。

如果文件读取已经都了文件末尾就退出循环，MP3 文件已经播放完整。循环中还需要等待 DMA 数据传输完成才进行下一帧的解码操作。

mp3PlayerDemo 函数最后停止 I2S，关闭解码器和 MP3 文件。

38.7.4.2 DMA 发送完成中断回调函数

列表 28: 代码清单 36_0_28 MP3Player_I2S_DMA_TX_Callback 函数

```
void MP3Player_I2S_DMA_TX_Callback(void)
{
    if (I2Sx_TX_DMA_STREAM->CR&(1<<19)) { //当前使用 Memory1 数据
        bufflag=0;                          //可以将数据读取到缓冲区 0
    } else {                               //当前使用 Memory0 数据
        bufflag=1;                          //可以将数据读取到缓冲区 1
    }
    Isread=1;                              // DMA 传输完成标志
}
```

38.7.4.3 主函数

列表 29: 代码清单 36_0_29 main 函数

```
int main(void)
{
    /* 配置系统时钟为 168 MHz */
    SystemClock_Config();
/* 初始化 LED */
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();
    LED_BLUE;
    /* 初始化触摸按键 */
    TPAD_Init();
    /* 初始化调试串口，一般为串口 1 */
    UARTx_Config();

    printf("Music Player\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
    res_sd = f_mount(&fs,"0:",1);

    if (res_sd!=FR_OK) {
        printf("!! SD 卡挂载文件系统失败。(%d)\r\n",res_sd);
        printf("!! 可能原因：SD 卡初始化不成功。\r\n");
        while (1);
    } else {
        printf("SD 卡挂载成功\r\n");
    }
    /* 检测 WM8978 芯片，此函数会自动配置 CPU 的 GPIO */
    if (wm8978_Init()==0) {
        printf("检测不到 WM8978 芯片!!!\n");
        while (1);  /* 停机 */
    }
    printf("初始化 WM8978 成功\n");


    while (1) {
        mp3PlayerDemo("0:/mp3/张国荣-玻璃之情.mp3");
        mp3PlayerDemo("0:/mp3/张国荣-全赖有你.mp3");
    }
```

(下页继续)

(续上页)

```
}
```

main 函数主要完成各个外设的初始化，包括初始化禁用 WIFI 模块、调试串口初始化、SD 卡文件系统挂载还有系统滴答定时器初始化。

wm8978_Init 初始化 I2C 接口用于控制 WM8978 芯片，并复位 WM8978 芯片，如果初始化成功则进入无限循环，执行 MP3 播放器实现函数 mp3PlayerDemo，它有一个形参，用于指定播放文件。

另外，为使程序正常运行还需要适当增加控制器的栈空间，见代码清单 38-30 栈空间大小调整，Helix 解码过程需要用到较多局部变量，需要调整栈空间，防止栈空间溢出。

列表 30: 代码清单 36_0_30 栈空间大小调整

```
Stack_Size      EQU     0x00001000

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

38.7.5 下载验证

将工程文件夹中的“音频文件放在 SD 卡根目录下”文件夹的内容拷贝到 Micro SD 卡根目录中，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，在开发板的上边沿的耳机插座 (左边那个) 插入耳机。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，如果没有提示错误信息则直接在耳机可听到音乐，播放完后自动切换下一首，如此循环。

实验主要展示 MP3 解码库移植过程和实现简单 MP3 文件播放，跟实际意义上的 MP3 播放器在功能上还有待完善，比如快进快退功能、声音调节、音效调节等等。

第 39 章　ETH—Lwip 以太网通信

互联网技术对人类社会的影响不言而喻。当今大部分电子设备都能以不同的方式接入互联网 (Internet)，在家庭中 PC 常见的互联网接入方式是使用路由器 (Router) 组建小型局域网 (LAN)，利用互联网专线或者调制调解器 (modem) 经过电话线网络，连接到互联网服务提供商 (ISP)，由互联网服务提供商把用户的局域网接入互联网。而企业或学校的局域网规模较大，常使用交换机组成局域网，经过路由以不同的方式接入到互联网中。

39.1 互联网模型

通信至少是两个设备的事，需要相互兼容的硬件和软件支持，我们称之为通信协议。以太网通信在结构比较复杂，国际标准组织将整个以太网通信结构制定了 OSI 模型，总共分层七个层，分别为应用层、表示层、会话层、传输层、网络层、数据链路层以及物理层，每个层功能不同，通信中各司其职，整个模型包括硬件和软件定义。OSI 模型是理想分层，一般的网络系统只是涉及其中几层。

TCP/IP 是互联网最基本的协议，是互联网通信使用的网络协议，由网络层的 IP 协议和传输层的 TCP 协议组成。TCP/IP 只有四个分层，分别为应用层、传输层、网络层以及网络访问层。虽然 TCP/IP 分层少了，但与 OSI 模型是不冲突的，它把 OSI 模型一些层次整合一起的，本质上可以实现相同功能。

实际上，还有一个 TCP/IP 混合模型，分为五个层，参考图 *36_1_1*，它实际与 TCP/IP 四层模型是相通的，只是把网络访问层拆成数据链路层和物理层。这种分层方法对我们学习理解更容易。

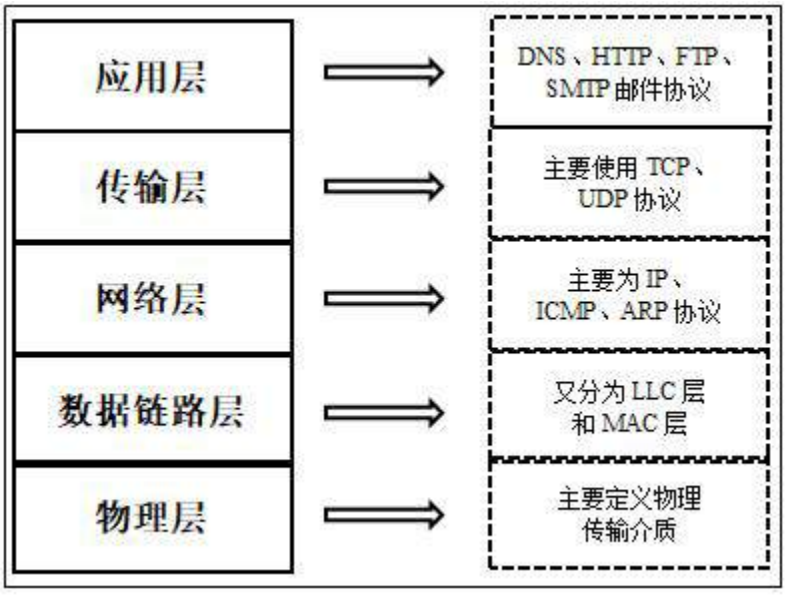


图 36_1_1 TCP/IP 混合参考模型

设计网络时，为了降低网络设计的复杂性，对组成网络的硬件、软件进行封装、分层，这些分层即构成了网络体系模型。在两个设备相同层之间的对话、通信约定，构成了层级协议。设备中使用的所有协议加起来统称协议栈。在这个网络模型中，每一层完成不同的任务，都提供接口供上一层访问。而在每层的内部，可以使用不同的方式来实现接口，因而内部的改变不会影响其它层。

在 TCP/IP 混合参考模型中，数据链路层又被分为 LLC 层 (逻辑链路层) 和 MAC 层 (媒体介质访问层)。目前，对于普通的接入网络终端的设备，LLC 层和 MAC 层是软、硬件的分界线。如 PC 的网卡主要负责实现参考模型中的 MAC 子层和物理层，在 PC 的软件系统中则有一套庞大程序实现了 LLC 层及以上的所有网络层次的协议。

由硬件实现的物理层和 MAC 子层在不同的网络形式有很大的区别，如以太网和 Wi-Fi，这是由物理传输方式决定的。但由软件实现的其它网络层次通常不会有太大区别，在 PC 上也许能实现完整的功能，一般支持所有协议，而在嵌入式领域则按需要进行裁剪。

39.2 以太网

以太网 (Ethernet) 是互联网技术的一种，由于它是在组网技术中占的比例最高，很多人直接把以太网理解为互联网。

以太网是指遵守 IEEE 802.3 标准组成的局域网，由 IEEE 802.3 标准规定的主要是位于参考模型的物理层 (PHY) 和数据链路层中的介质访问控制子层 (MAC)。在家庭、企业和学校所组建的 PC 局域网形式一般也是以太网，其标志是使用水晶头网线来连接 (当然还有其它形式)。IEEE 还有其它局域网标准，如 IEEE 802.11 是无线局域网，俗称 Wi-Fi。IEEE 802.15 是个人域网，即蓝牙技术，其中的 802.15.4 标准则是 ZigBee 技术。

现阶段，工业控制、环境监测、智能家居的嵌入式设备产生了接入互联网的需求，利用以太网技术，嵌入式设备可以非常容易地接入到现有的计算机网络中。

39.2.1 PHY 层

在物理层，由 IEEE 802.3 标准规定了以太网使用的传输介质、传输速度、数据编码方式和冲突检测机制，物理层一般是通过一个 PHY 芯片实现其功能的。

39.2.1.1 传输介质

传输介质包括同轴电缆、双绞线 (水晶头网线是一种双绞线)、光纤。根据不同的传输速度和距离要求，基于这三类介质的信号线又衍生出很多不同的种类。最常用的是“五类线”适用于 100BASE-T 和 10BASE-T 的网络，它们的网络速率分别为 100Mbps 和 10Mbps。

39.2.1.2 编码

为了让接收方在没有外部时钟参考的情况也能确定每一位的起始、结束和中间位置，在传输信号时不直接采用二进制编码。在 10BASE-T 的传输方式中采用曼彻斯特编码，在 100BASE-T 中则采用 4B/5B 编码。

曼彻斯特编码把每一个二进制位的周期分为两个间隔，在表示“1”时，以前半个周期为高电平，后半个周期为低电平。表示“0”时则相反，见图 *36_1_2*

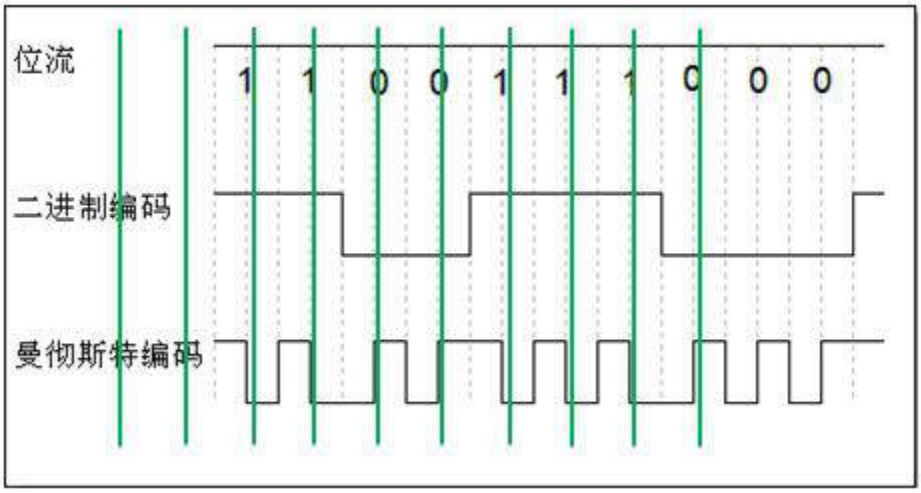


图 36_1_2 曼彻斯特编码

采用曼彻斯特码在每个位周期都有电压变化，便于同步。但这样的编码方式效率太低，只有 50%。

在 100BASE-T 采用的 4B/5B 编码是把待发送数据位流的每 4 位分为一组，以特定的 5 位编码来表示，这些特定的 5 位编码能使数据流有足够多的跳变，达到同步的目的，而且效率也从曼彻斯特编码的 50% 提高到了 80%。

39.2.1.3 CSMA/CD 冲突检测

早期的以太网大多是多个节点连接到同一条网络总线上 (总线型网络)，存在信道竞争问题，因而每个连接到以太网上的节点都必须具备冲突检测功能。以太网具备 CSMA/CD 冲突检测机制，如果多个节点同时利用同一条总线发送数据，则会产生冲突，总线上的节点可通过接收到的信号与原始发送的信号的比较检测是否存在冲突，若存在冲突则停止发送数据，随机等待一段时间再重传。

现在大多数局域网组建的时候很少采用总线型网络，大多是一个设备接入到一个独立的路由或交换机接口，组成星型网络，不会产生冲突。但为了兼容，新出的产品还是带有冲突检测机制。

39.2.2 MAC 子层

39.2.2.1 MAC 的功能

MAC 子层是属于数据链路层的下半部分，它主要负责与物理层进行数据交接，如是否可以发送数据，发送的数据是否正确，对数据流进行控制等。它自动对来自上层的数据包加上一些控制信号，交给物理层。接收方得到正常数据时，自动去除 MAC 控制信号，把该数据包交给上层。

39.2.2.2 MAC 数据包

IEEE 对以太网上传输的数据包格式也进行了统一规定，见图 *36_1_3*。该数据包被称为 MAC 数据包。

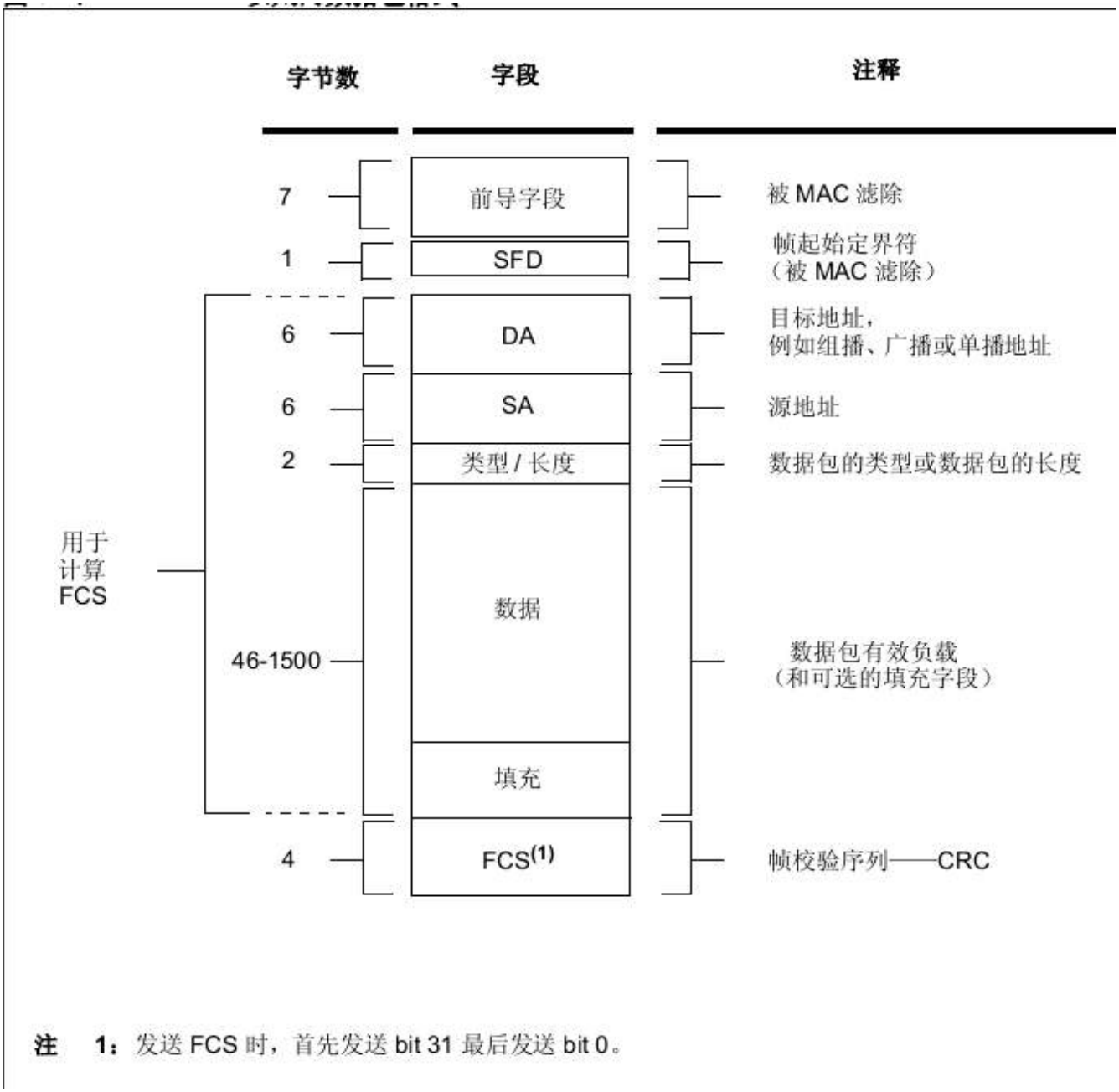


图 36_1_3 MAC 数据包格式

MAC 数据包由前导字段、帧起始定界符、目标地址、源地址、数据包类型、数据域、填充域、校验和域组成。

- 前导字段，也称报头，这是一段方波，用于使收发节点的时钟同步。内容为连续 7 个字节

的 0x55。字段和帧起始定界符在 MAC 收到数据包后会自动过滤掉。

- 帧起始定界符 (SFD)：用于区分前导段与数据段的，内容为 0xD5。
- MAC 地址：MAC 地址由 48 位数字组成，它是网卡的物理地址，在以太网传输的最底层，就是根据 MAC 地址来收发数据的。部分 MAC 地址用于广播和多播，在同一个网络里不能有两个相同的 MAC 地址。PC 的网卡在出厂时已经设置好了 MAC 地址，但也可以通过一些软件来进行修改，在嵌入式的以太网控制器中可由程序进行配置。数据包中的 DA 是目标地址，SA 是源地址。
- 数据包类型：本区域可以用来描述本 MAC 数据包是属于 TCP/IP 协议层的 IP 包、ARP 包还是 SNMP 包，也可以用来描述本 MAC 数据包数据段的长度。如果该值被设置大于 0x0600，不用于长度描述，而是用于类型描述功能，表示与以太网帧相关的 MAC 客户端协议的种类。
- 数据段：数据段是 MAC 包的核心内容，它包含的数据来自 MAC 的上层。其长度可以从 0~1500 字节间变化。
- 填充域：由于协议要求整个 MAC 数据包的长度至少为 64 字节 (接收到的数据包如果少于 64 字节会被认为发生冲突，数据包被自动丢弃)，当数据段的字节少于 46 字节时，在填充域会自动填上无效数据，以使数据包符合长度要求。
- 校验和域：MAC 数据包的尾部是校验和域，它保存了 CRC 校验序列，用于检错。

以上是标准的 MAC 数据包，IEEE 802.3 同时还规定了扩展的 MAC 数据包，它是在标准的 MAC 数据包的 SA 和数据包类型之间添加 4 个字节的 QTag 前缀字段，用于获取标志的 MAC 帧。前 2 个字节固定为 0x8100，用于识别 QTag 前缀的存在；后两个字节内容分别为 3 个位的用户优先级、1 个位的标准格式指示符 (CFI) 和一个 12 位的 VLAN 标识符。

39.3 TCP/IP 协议栈

标准 TCP/IP 协议是用于计算机通信的一组协议，通常称为 TCP/IP 协议栈，通俗讲就是符合以太网通信要求的代码集合，一般要求它可以实现图 39-1 中每个层对应的协议，比如应用层的 HTTP、FTP、DNS、SMTP 协议，传输层的 TCP、UDP 协议、网络层的 IP、ICMP 协议等等。关于 TCP/IP 协议详细内容推荐阅读《TCP-IP 详解》和《用 TCP/IP 进行网际互连》理解。

Windows 操作系统、UNIX 类操作系统都有自己的一套方法来实现 TCP/IP 通信协议，它们都提供非常完整的 TCP/IP 协议。对于一般的嵌入式设备，受制于硬件条件没办法支持使用在 Window 或 UNIX 类操作系统的运行的 TCP/IP 协议栈，一般只能使用简化版本的 TCP/IP 协议栈，目前开源的适合嵌入式的有 uIP、TinyTCP、uC/TCP-IP、LwIP 等等。其中 LwIP 是目前在嵌入式网络领域被讨论和使用广泛的协议栈。本章内容其中一个目的就是移植 LwIP 到开发板上运行。

39.3.1 为什么需要协议栈

物理层主要定义物理介质性质，MAC 子层负责与物理层进行数据交接，这两部分是与硬件紧密联系的，就嵌入式控制芯片来说，很多都内部集成了 MAC 控制器，完成 MAC 子层功能，所以依靠这部分功能是可以实现两个设备数据交换，而时间传输的数据就是 MAC 数据包，发送端封装好数据包，接收端则解封数据包得到可用数据，这样的一个模型与使用 USART 控制器实现数据传输是非常类似的。但如果将以太网运用在如此基础的功能上，完全是大材小用，因为以太网具有传输速度快、可传输距离远、支持星型拓扑设备连接等等强大功能。功能强大的东西一般都会用高级的应用，这也是设计者的初衷。

使用以太网接口的目的就是为了方便与其它设备互联，如果所有设备都约定使用一种互联方式，在软件上加一些层次来封装，这样不同系统、不同的设备通讯就变得相对容易了。而且只要新加入的设备也使用同一种方式，就可以直接与之前存在于网络上的其它设备通讯。这就是为什么产生了在 MAC 之上的其它层次的网络协议及为什么要使用协议栈的原因。又由于在各种协议栈中 TCP/IP 协议栈得到了最广泛使用，所有接入互联网的设备都遵守 TCP/IP 协议。所以，想方便地与其它设备互联通信，需要提供对 TCP/IP 协议的支持。

39.3.2 各网络层的功能

用以太网和 Wi-Fi 作例子，它们的 MAC 子层和物理层有较大的区别，但在 MAC 之上的 LLC 层、网络层、传输层和应用层的协议，是基本相同的，这几层协议由软件实现，并对各层进行封装。根据 TCP/IP 协议，各层的要实现的功能如下：

LLC 层：处理传输错误；调节数据流，协调收发数据双方速度，防止发送方发送得太快而接收方丢失数据。主要使用数据链路协议。

网络层：本层也被称为 IP 层。LLC 层负责把数据从线的一端传输到另一端，但很多时候不同的设备位于不同的网络中 (并不是简单的网线的两头)。此时就需要网络层来解决子网路由拓扑问题、路径选择问题。在这一层主要有 IP 协议、ICMP 协议。

传输层：由网络层处理好了网络传输的路径问题后，端到端的路径就建立起来了。传输层就负责处理端到端的通讯。在这一层中主要有 TCP、UDP 协议

应用层：经过前面三层的处理，通讯完全建立。应用层可以通过调用传输层的接口来编写特定的应用程序。而 TCP/IP 协议一般也会包含一些简单的应用程序如 Telnet 远程登录、FTP 文件传输、SMTP 邮件传输协议。

实际上，在发送数据时，经过网络协议栈的每一层，都会给来自上层的数据添加上一个数据包的头，再传递给下一层。在接收方收到数据时，一层层地把所在层的数据包的头去掉，向上层递交数据，参考图 39-4。

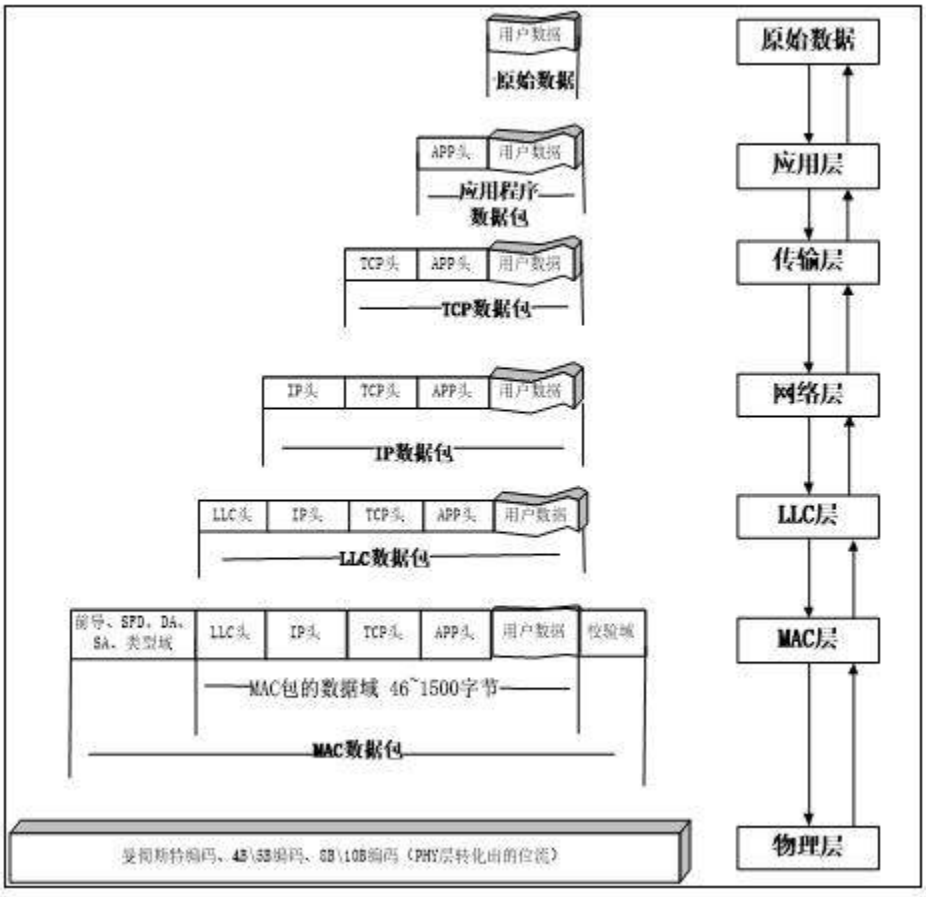


图 36_1_4 数据经过每一层的封装和还原

39.4 以太网外设 (ETH)

STM32F4xx 系列控制器内部集成了一个以太网外设，它实际是一个通过 DMA 控制器进行介质访问控制 (MAC)，它的功能就是实现 MAC 层的任务。借助以太网外设，STM32F4xx 控制器可以通过 ETH 外设按照 IEEE 802.3-2002 标准发送和接收 MAC 数据包。ETH 内部自带专用的 DMA 控制器用于 MAC，ETH 支持两个工业标准接口介质独立接口 (MII) 和简化介质独立接口 (RMII) 用于与外部 PHY 芯片连接。MII 和 RMII 接口用于 MAC 数据包传输，ETH 还集成了站管理接口 (SMI) 接口专门用于与外部 PHY 通信，用于访问 PHY 芯片寄存器。

物理层定义了以太网使用的传输介质、传输速度、数据编码方式和冲突检测机制，PHY 芯片是物理层功能实现的实体，生活中常用水晶头网线 + 水晶头插座 +PHY 组合构成了物理层。

ETH 有专用的 DMA 控制器，它通过 AHB 主从接口与内核和存储器相连，AHB 主接口用于控制数据传输，而 AHB 从接口用于访问 “控制与状态寄存器” (CSR) 空间。在进行数据发送是，先将数据有存储器以 DMA 传输到发送 TX FIFO 进行缓冲，然后由 MAC 内核发送；接收数据时，RX FIFO 先接收以太网数据帧，再由 DMA 传输至存储器。ETH 系统功能框图见图 *36_1_5*。

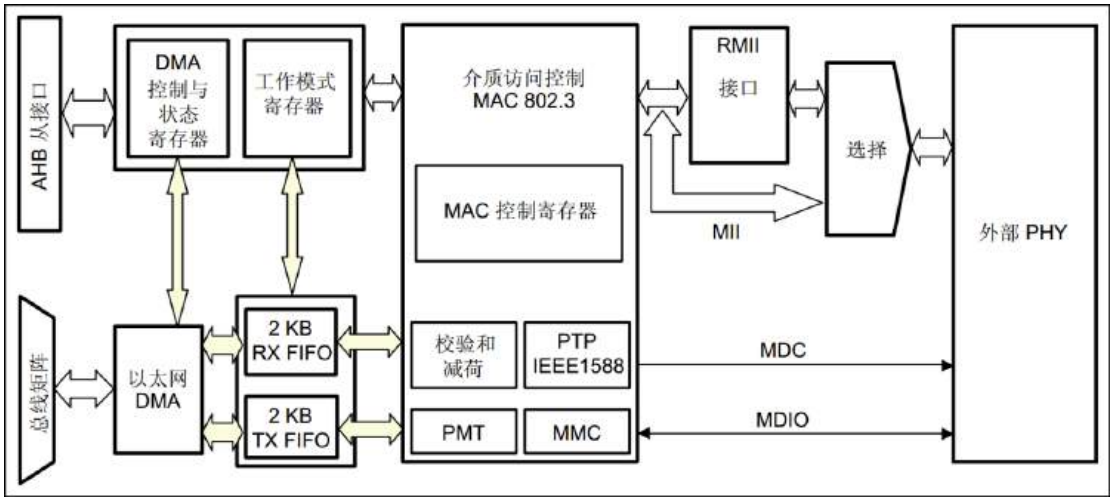


图 36_1_5 ETH 功能框图

39.4.1 SMI 接口

SMI 是 MAC 内核访问 PHY 寄存器标志接口，它由两根线组成，数据线 MDIO 和时钟线 MDC。SMI 支持访问 32 个 PHY，这在设备需要多个网口时非常有用，不过一般设备都只使用一个 PHY。PHY 芯片内部一般都有 32 个 16 位的寄存器，用于配置 PHY 芯片属性、工作环境、状态指示等等，当然很多 PHY 芯片并没有使用到所有寄存器位。MAC 内核就是通过 SMI 向 PHY 的寄存器写入数据或从 PHY 寄存器读取 PHY 状态，一次只能对一个 PHY 的其中一个寄存器进行访问。SMI 最大通信频率为 2.5MHz，通过控制以太网 MAC MII 地址寄存器 (ETH_MACMIIAR) 的 CR 位可选择时钟频率。

39.4.1.1 SMI 帧格式

SMI 是通过数据帧方式与 PHY 通信的，帧格式如表 36-1-1，数据位传输顺序从左到右。

表 36-1-1 SMI 帧格式

	管理帧字段							
	报头 (32bit)	起始	操作	PADDR	RADDR	TA	数据 (16bit)	空闲
读取	111…111	01	10	ppppp	rrrrr	Z0	ddd…ddd	Z
写入	111…111	01	01	ppppp	rrrrr	10	ddd…ddd	Z

PADDR 用于指定 PHY 地址，每个 PHY 都有一个地址，一般由 PHY 硬件设计决定，所以是固定不变的。RADDR 用于指定 PHY 寄存器地址。TA 为状态转换域，若为读操作，MAC 输出两个位高阻态，而 PHY 芯片则在第一位时输出高阻态，第二位时输出 “0”。若为写操作，MAC 输出 “10”，PHY 芯片则输出高阻态。数据段有 16 位，对应 PHY 寄存器每个位，先发送或接收到的位对应以太网 MAC MII 数据寄存器 (ETH_MACMIIDR) 寄存器的位 15。

39.4.1.2 SMI 读写操作

当以太网 MAC MII 地址寄存器 (ETH_MACMIIAR) 的写入位和繁忙位被置 1 时，SMI 将向指定的 PHY 芯片指定寄存器写入 ETH_MACMIIDR 中的数据。写操作时序图 *36_1_6*。

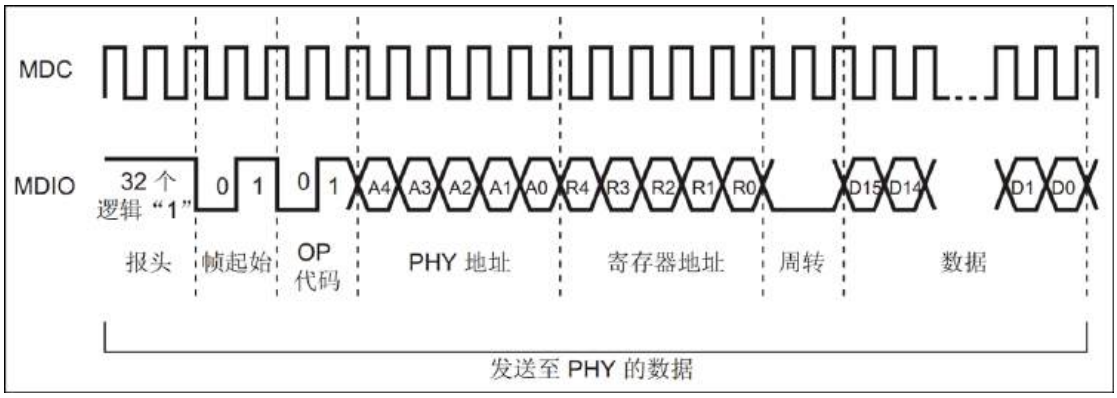


图 36_1_6 SMI 写操作

当以太网 MAC MII 地址寄存器 (ETH_MACMIIAR) 的写入位为 0 并且繁忙位被置 1 时，SMI 将从向指定的 PHY 芯片指定寄存器读取数据到 ETH_MACMIIDR 内。读操作时序见图 *36_1_7*。

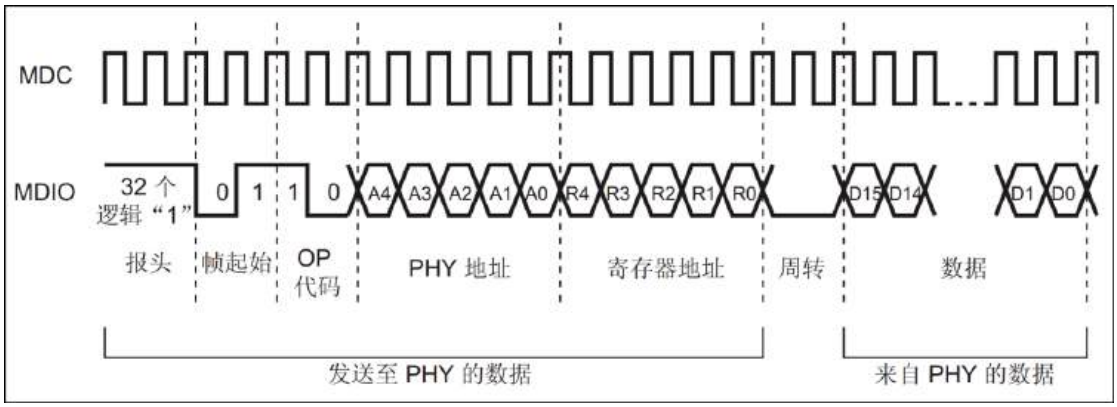


图 36_1_7 SMI 读操作

39.4.2 MII 和 RMII 接口

介质独立接口 (MII) 用于连接 MAC 控制器和 PHY 芯片，提供数据传输路径。RMII 接口是 MII 接口的简化版本，MII 需要 16 根通信线，RMII 只需 7 根通信，在功能上是相同的。图 *36_1_8* 为 MII 接口连接示意图，图 *36_1_9* 为 RMII 接口连接示意图。

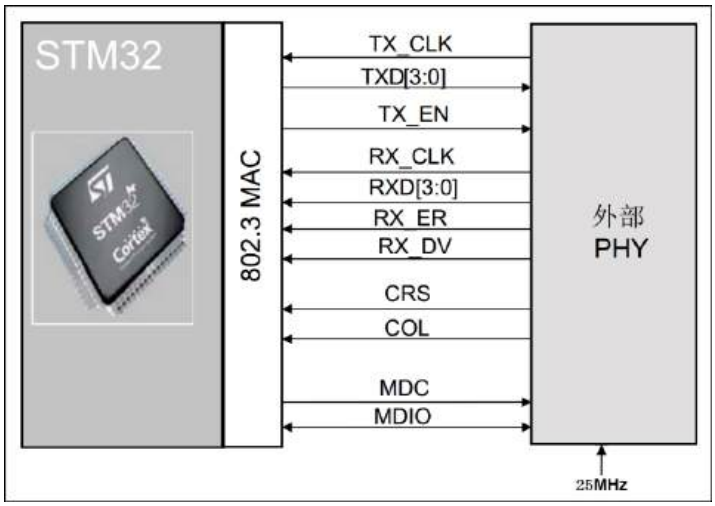


图 36_1_8 MII 接口连接

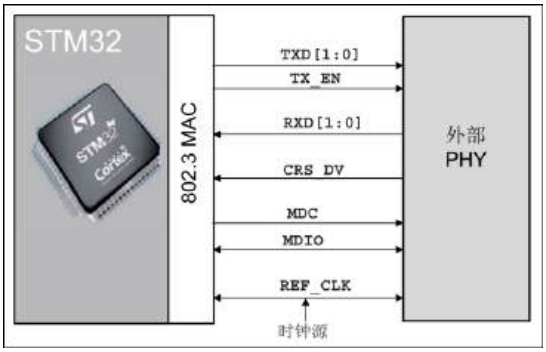


图 36_1_9 RMII 接口连接

- TX_CLK：数据发送时钟线。标称速率为 10Mbit/s 时为 2.5MHz；速率为 100Mbit/s 时为 25MHz。RMII 接口没有该线。
- RX_CLK：数据接收时钟线。标称速率为 10Mbit/s 时为 2.5MHz；速率为 100Mbit/s 时为 25MHz。RMII 接口没有该线。
- TX_EN：数据发送使能。在整个数据发送过程保存有效电平。

- TXD[3:0] 或 TXD[1:0]：数据发送数据线。对于 MII 有 4 位，RMII 只有 2 位。只有在 TX_EN 处于有效电平数据线才有效。
- CRS：载波侦听信号，由 PHY 芯片负责驱动，当发送或接收介质处于非空闲状态时使能该信号。在全双工模式该信号线无效。
- COL：冲突检测信号，由 PHY 芯片负责驱动，检测到介质上存在冲突后该线被使能，并且保持至冲突解除。在全双工模式该信号线无效。
- RXD[3:0] 或 RXD[1:0]：数据接收数据线，由 PHY 芯片负责驱动。对于 MII 有 4 位，RMII 只有 2 位。在 MII 模式，当 RX_DV 禁止、RX_ER 使能时，特定的 RXD[3:0] 值用于传输来自 PHY 的特定信息。
- RX_DV：接收数据有效信号，功能类似 TX_EN，只不过用于数据接收，由 PHY 芯片负责驱动。对于 RMII 接口，是把 CRS 和 RX_DV 整合成 CRS_DV 信号线，当介质处于不同状态时会自切换该信号状态。
- RX_ER：接收错误信号线，由 PHY 驱动，向 MAC 控制器报告在帧某处检测到错误。
- REF_CLK：仅用于 RMII 接口，由外部时钟源提供 50MHz 参考时钟。

因为要达到 100Mbit/s 传输速度，MII 和 RMII 数据线数量不同，使用 MII 和 RMII 在时钟线的设计是完全不同的。对于 MII 接口，一般是外部为 PHY 提供 25MHz 时钟源，再由 PHY 提供 TX_CLK 和 RX_CLK 时钟。对于 RMII 接口，一般需要外部直接提供 50MHz 时钟源，同时接入 MAC 和 PHY。

开发板板载的 PHY 芯片型号为 LAN8720A，该芯片只支持 RMII 接口，电路设计时参考图 *36_1_9*。

ETH 相关硬件在 STM32F4xx 控制器分布参考表 36-1-2。

表 36-1-2 ETH 复用引脚

ETH(AF11)	GPIO	
MII	MII_TX_CLK	PC3
	MII_TXD0	PB12/PG13
	MII_TXD1	PB13/PG14
	MII_TXD2	PC2
	MII_TXD3	PB8/PE2
	MII_TX_EN	PB11/PG11
	MII_RX_CLK	PA1
	MII_RXD0	PC4
	MII_RXD1	PC5
	MII_RXD2	PB0
	MII_RXD3	PB1
	MII_RX_ER	PB10
	MII_RX_DV	PA7
	MII_CRS	PA0
	MII_COL	PA3
RMII	RMII_TXD0	PB12/PG13
	RMII_TXD1	PB13/PG14
	RMII_TX_EN	PG11
	RMII_RXD0	PC4
	RMII_RXD1	PC5
	RMII_CRS_DV	PA7
	RMII_REF_CLK	PA1
SMI	MDIO	PA2
	MDC	PC1
其他	PPS_OUT	PB5/PG8

其中，PPS_OUT 是 IEEE 1588 定义的一个时钟同步机制。

39.4.3 MAC 数据包发送和接收

ETH 外设负责 MAC 数据包发送和接收。利用 DMA 从系统寄存器得到数据包数据内容，ETH 外设自动填充完成 MAC 数据包封装，然后通过 PHY 发送出去。在检测到有 MAC 数据包需要接收时，ETH 外设控制数据接收，并解封 MAC 数据包得到解封后数据通过 DMA 传输到系统寄存器内。

39.4.3.1 MAC 数据包发送

MAC 数据帧发送全部由 DMA 控制，从系统存储器读取的以太网帧由 DMA 推入 FIFO，然后将帧弹出并传输到 MAC 内核。帧传输结束后，从 MAC 内核获取发送状态并传回 DMA。在检测到 SOF(Start Of Frame) 时，MAC 接收数据并开始 MII 发送。在 EOF(End Of Frame) 传输到 MAC 内核后，内核将完成正常的发送，然后将发送状态返回给 DMA。如果在发送过程中发送常规冲突，MAC 内核将使发送状态有效，然后接受并丢弃所有后续数据，直至收到下一 SOF。检测到来自 MAC 的重试请求时，应从 SOF 重新发送同一帧。如果发送期间未连续提供数据，MAC 将发出下溢状态。在帧的正常传输期间，如果 MAC 在未获得前一帧的 EOF 的情况下接收到 SOF，则将忽略该 SOF 并将新的帧视为前一帧的延续。

MAC 控制 MAC 数据包的发送操作，它会自动生成前导字段和 SFD 以及发送帧状态返回给 DMA，在半双工模式下自动生成阻塞信号，控制 jabber(MAC 看门狗) 定时器用于在传输字节超过 2048 字节时切断数据包发送。在半双工模式下，MAC 使用延迟机制进行流量控制，程序通过将 ETH_MACFCR 寄存器的 BPA 位置 1 来请求流量控制。MAC 包含符合 IEEE 1588 的时间戳快照逻辑。MAC 数据包发送时序参考图 *36_1_10*。

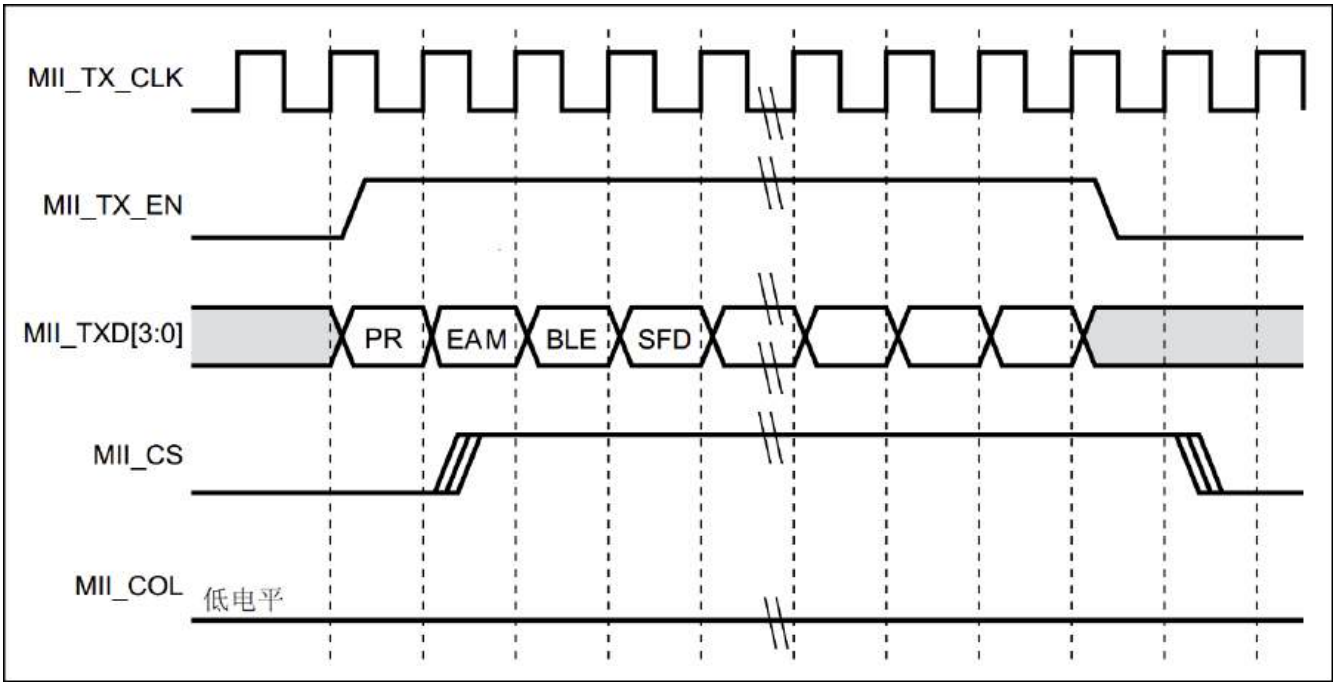


图 36_1_10 MAC 数据包发送时序 (无冲突)

39.4.3.2 MAC 数据包接收

MAC 接收到的数据包填充 RX FIFO，达到 FIFO 设定阈值后请求 DMA 传输。在默认直通模式下，当 FIFO 接收到 64 个字节 (使用 ETH_DMAOMR 寄存器中的 RTC 位配置) 或完整的数据包时，数据将弹出，其可用性将通知给 DMA。DMA 向 AHB 接口发起传输后，数据传输将从 FIFO 持续进行，直到传输完整个数据包。完成 EOF 帧的传输后，状态字将弹出并发送到 DMA 控制器。在 Rx FIFO 存储转发模式 (通过 ETH_DMAOMR 寄存器中的 RSF 位配置) 下，仅在帧完全写入 Rx FIFO 后才可读出帧。

当 MAC 在 MII 上检测到 SFD 时，将启动接收操作。MAC 内核将去除报头和 SFD，然后再继续处理帧。检查报头字段以进行过滤，FCS 字段用于验证帧的 CRC 如果帧未通过地址滤波器，则在内核中丢弃该帧。MAC 数据包接收时序参考图 *36_1_11*。

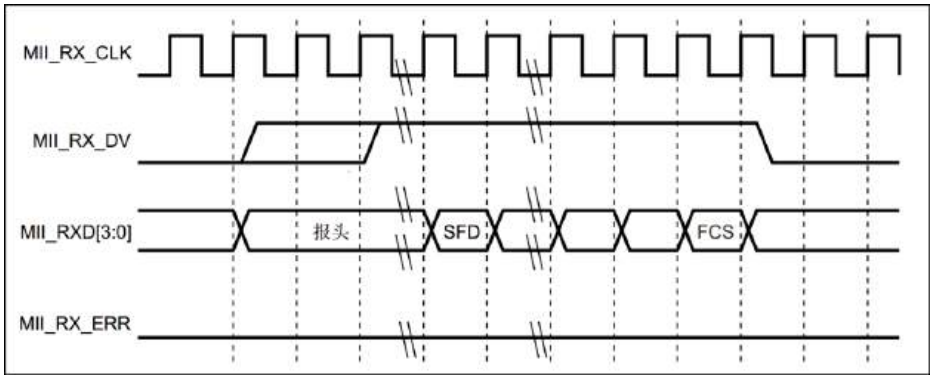


图 36_1_11 MAC 数据包接收时序 (无错误)

39.4.4 MAC 过滤

MAC 过滤功能可以选择性的过滤设定目标地址或源地址的 MAC 帧。它将检查所有接收到的数据帧的目标地址和源地址，根据过滤选择设定情况，检测后报告过滤状态。针对目标地址过滤可以有三种，分别是单播、多播和广播目标地址过滤；针对源地址过滤就只有单播源地址过滤。

单播目标地址过滤是将接收的相应 DA 字段与预设的以太网 MAC 地址寄存器内容比较，最高可预设 4 个过滤 MAC 地址。多播目标地址过滤是根据帧过滤寄存器中的 HM 位执行对多播地址的过滤，是对 MAC 地址寄存器进行比较来实现的。单播和多播目标地址过滤都还支持 Hash 过滤模式。广播目标地址过滤通过将帧过滤寄存器的 BFD 位置 1 使能，这使得 MAC 丢弃所有广播帧。

单播源地址过滤是将接收的 SA 字段与 SA 寄存器内容进行比较过滤。

MAC 过滤还具备反向过滤操作功能，即让过滤结构求补集。

39.5 PHY：LAN8720A

LAN8720A 是 SMSC 公司 (已被 Microchip 公司收购) 设计的一个体积小、功耗低、全能型 10/100Mbps 的以太网物理层收发器。它是针对消费类电子和企业应用而设计的。LAN8720A 总共只有 24Pin，仅支持 RMII 接口。由它组成的网络结构见图 *36_1_12*。

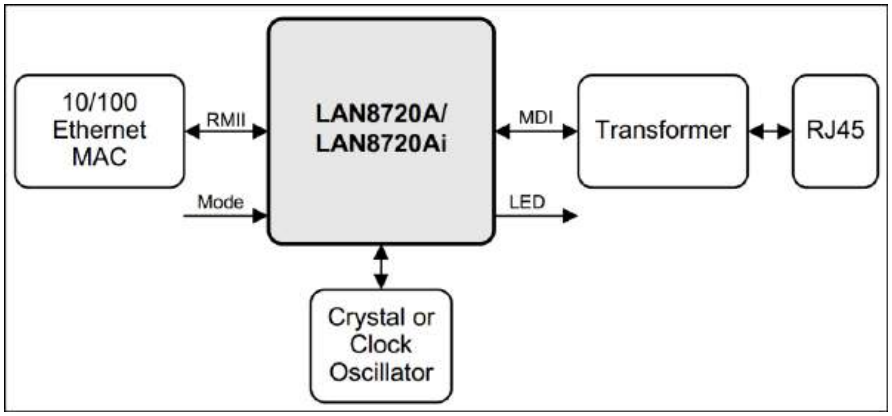


图 36_1_12 由 LAN8720A 组成的网络系统结构

LAN8720A 通过 RMII 与 MAC 连接。RJ45 是网络插座，在与 LAN8720A 连接之间还需要一个变压器，所以一般使用带电压转换和 LED 指示灯的 HY911105A 型号的插座。一般来说，必须为使用 RMII 接口的 PHY 提供 50MHz 的时钟源输入到 REF_CLK 引脚，不过 LAN8720A 内部集成 PLL，可以将 25MHz 的时钟源陪频到 50MHz 并在指定引脚输出该时钟，所以我们可以直接使其与 REF_CLK 连接达到提供 50MHz 时钟的效果。

LAN8720A 内部系统结构见图 *36_1_13*。

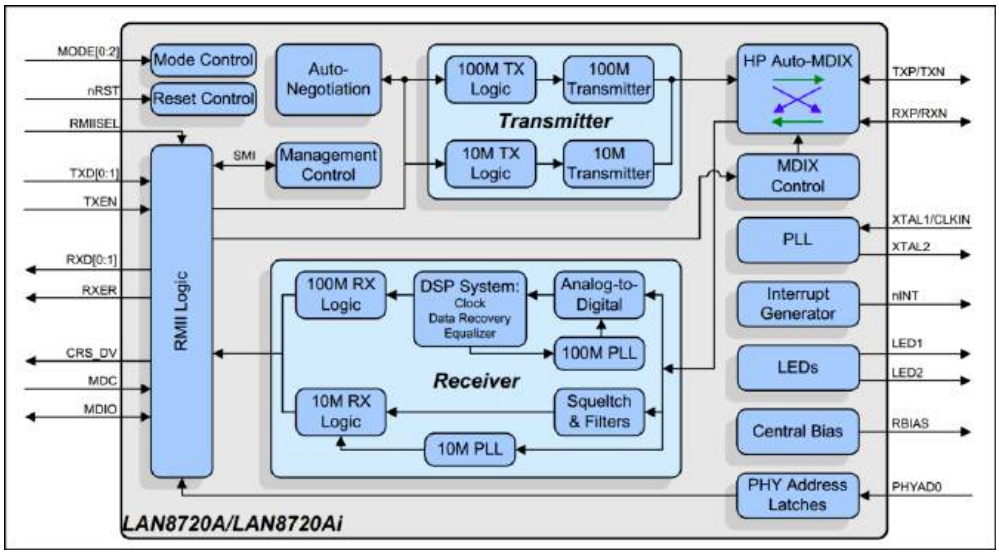


图 36_1_13 LAN8720A 内部系统结构

LAN8720A 有各个不同功能模块组成，最重要的要数接收控制器和发送控制器，其它的基本上都是与外部引脚挂钩，实现信号传输。部分引脚是具有双重功能的，比如 PHYAD0 与 RXER 引脚

是共用的，在系统上电后 LAN8720A 会马上读取这部分共用引脚的电平，以确定系统的状态并保存在相关寄存器内，之后则自动转入作为另一功能引脚。

PHYAD[0] 引脚用于配置 SMI 通信的 LAN8720A 地址，在芯片内部该引脚已经自带下拉电阻，默认认为 0(即使外部悬空不接)，在系统上电时会检测该引脚获取得到 LAN8720A 的地址为 0 或者 1，并保存在特殊模式寄存器 (R18) 的 PHYAD 位中，该寄存器的 PHYAD 有 5 个位，在需要超过 2 个 LAN8720A 时可以通过软件设置不同 SMI 通信地址。PHYAD[0] 是与 RXER 引脚共用。

MODE[2:0] 引脚用于选择 LAN8720A 网络通信速率和工作模式，可选 10Mbps 或 100Mbps 通信速度，半双工或全双工工作模式，另外 LAN8720A 支持 HP Auto-MDIX 自动翻转功能，即可自动识别直连或交叉网线并自适应。一般将 MODE 引脚都设置为 1，可以让 LAN8720A 启动自适应功能，它会自动寻找最优工作方式。MODE[0] 与 RXD0 引脚共用、MODE[1] 与 RXD1 引脚共用、MODE[2] 与 CRS_DV 引脚共用。

nINT/REFCLKO 引脚用于 RMII 接口中 REF_CLK 信号线，当 nINTSEL 引脚为低电平是，它也可以被设置成 50MHz 时钟输出，这样可以直接与 STM32F4xx 的 REF_CLK 引脚连接为其提供 50MHz 时钟源，这种模式要求为 XTAL1 与 XTAL2 之间或为 XTAL1/CLKIN 提供 25MHz 时钟，由 LAN8720A 内部 PLL 电路陪频得到 50MHz 时钟，此时 nIN/REFCLKO 引脚的中断功能不可用，用于 50MHz 时钟输出。当 nINTSEL 引脚为高电平时，LAN8720A 被设置为时钟输入，即外部时钟源直接提供 50MHz 时钟接入 STM32F4xx 的 REF_CLK 引脚和 LAN8720A 的 XTAL1/CLKIN 引脚，此时 nINT/REFCLKO 可用于中断功能。nINTSEL 与 LED2 引脚共用，一般使用下拉

REGOFF 引脚用于配置内部 +1.2V 电压源，LAN8720A 内部需要 +1.2V 电压，可以通过 VDDCR 引脚输入 +1.2V 电压提供，也可以直接利用 LAN8720A 内部 +1.2V 稳压器提供。当 REGOFF 引脚为低电平时选择内部 +1.2V 稳压器。REGOFF 与 LED1 引脚共用。

SMI 支持寻址 32 个寄存器，LAN8720A 只用到其中 14 个，参考表 36-1-3。

表 36-1-3 LAN8720A 寄存器列表

序号	寄存器名称	分组
0	Basic Control Register	Basic
1	Basic Status Register	Basic
2	PHY Identifier 1	Extended
3	PHY Identifier 2	Extended
4	Auto-Negotiation Advertisement Register	Extended
5	Auto-Negotiation Link Partner Ability Register	Extended
6	Auto-Negotiation Expansion Register	Extended
17	Mode Control/Status Register	Vendor-specific
18	Special Modes	Vendor-specific
26	Symbol Error Counter Register	Vendor-specific
27	Control / Status Indication Register	Vendor-specific
29	Interrupt Source Register	Vendor-specific
30	Interrupt Mask Register	Vendor-specific
31	PHY Special Control/Status Register	Vendor-specific

序号与 SMI 数据帧中的 RADDR 是对应的，这在编写驱动时非常重要，本文将它们标记为 R0~R31。寄存器可规划为三个组：Basic、Extended 和 Vendor-specific。Basic 是 IEEE 802.3 要求的，R0 是基本控制寄存器，其位 15 为 Soft Reset 位，向该位写 1 启动 LAN8720A 软件复位，还包括速度、自适应、低功耗等等功能设置。R1 是基本状态寄存器。Extended 是扩展寄存器，包括 LAN8720A 的 ID 号、制造商、版本号等等信息。Vendor-specific 是供应商自定义寄存器，R31 是特殊控制/状态寄存器，指示速度类型和自适应功能。

39.6 LwIP：轻型 TCP/IP 协议栈

LwIP 是 Light Weight Internet Protocol 的缩写，是由瑞士计算机科学院 Adam Dunkels 等开发的适用于嵌入式领域的开源轻量级 TCP/IP 协议栈。它可以移植到含有操作系统的平台中，也可以在无操作系统的平台下运行。由于它开源、占用的 RAM 和 ROM 比较少、支持较为完整的 TCP/IP 协议、且十分便于裁剪、调试，被广泛应用在中低端的 32 位控制器平台。可以访问网站：http://savannah.nongnu.org/projects/lwip/ 获取更多 LwIP 信息。

目前，LwIP 最新更新到 1.4.1 版本，我们在上述网站可找到相应的 LwIP 源码下载通道。我们下载两个压缩包：lwip-1.4.1.zip 和 contrib-1.4.1.zip，lwip-1.4.1.zip 包括了 LwIP 的实现代码，contrib-1.4.1.zip 包含了不同平台移植 LwIP 的驱动代码和使用 LwIP 实现的一些应用实例测试。

但是，遗憾的是 contrib-1.4.1.zip 并没有为 STM32 平台提供实例，这对于初学者想要移植 LwIP 来说难度还是非常大的。ST 公司也是认识到 LwIP 在嵌入式领域的重要性，所以他们针对 LwIP 应用开发了测试平台，其中有一个是在 STM32F4x7 系列控制器运行的 (文件编号为：STSW-STM32070)。为减少移植工作量，我们选择使用 ST 官方例程相关文件，特别是 ETH 底层驱动部分函数，这样我们也可以花更多精力在理解代码实现方法上。

本章的一个重点内容就是介绍 LwIP 移植至我们的开发平台，详细的操作步骤参考下文介绍。

39.7 ETH 初始化结构体详解

从 STM32 的 ETH 外设我们了解到它的功能非常多，控制涉及的寄存器也非常丰富，而使用 STM32 HAL 库提供的各种结构体及库函数可以简化这些控制过程。跟其它外设一样，STM32 HAL 库提供了初始化结构体成员用于设置 ETH 工作环境参数，并由 ETH 相应初始化配置函数或功能函数调用，这些设定参数将会设置 ETH 相应的寄存器，达到配置 ETH 工作环境的目的。这些内容都定义在库文件“stm32f4xx_hal_eth.h”及“stm32f4xx_ hal_eth.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。

列表 1: 代码清单 36_1_1 ETH_InitTypeDef

```
typedef struct {
    uint32_t              AutoNegotiation; // 自适应功能
    uint32_t              Speed;             // 以太网速度
    uint32_t              DuplexMode;      // 以太网工作模式选择
    uint16_t              PhyAddress;      // 以太网 PHY 地址
    uint8_t              *MACAddr;         // MAC 地址指针
    uint32_t              RxMode;          // 以太网接收模式
    uint32_t              ChecksumMode;    // 检查校验和模式
    uint32_t              MediaInterface;  // 以太网介质接口
} ETH_InitTypeDef;
```

- AutoNegotiation：自适应功能选择，可选使能或禁止，一般选择使能自适应功能，系统会自动寻找最优工作方式，包括选择 10Mbps 或者 100Mbps 的以太网速度以及全双工模式或半双工模式。
- Speed：以太网速度选择，可选 10Mbps 或 100Mbit/s，它设定 ETH_MACCR 寄存器的 FES 位的值，一般设置 100Mbit/s，但在使能自适应功能之后该位设置无效。
- DuplexMode：以太网工作模式选择，可选全双工模式或半双工模式，它设定 ETH_MACCR 寄存器 DM 位的值。一般选择全双工模式，在使能了自适应功能后该成员设置无效。
- PhyAddress：以太网 PHY 地址，取值范围为 0~32。该字段指示正在访问 32 个可能的 PHY 器件中的哪一个。
- *MACAddr：MAC 地址指针，必须是一个 6 个元素数组的指针。
- RxMode：以太网接收接收模式，可以是轮询模式或者中断模式。
- ChecksumMode：检查校验和模式，可以是硬件校验或者软件校验。
- MediaInterface：以太网介质接口，可以是 MII 介质接口或者 RMII 介质接口。

列表 2: 代码清单 36_1_2 ETH_MACInitTypeDef

```
typedef struct {
    uint32_t              Watchdog;          // 以太网看门狗
    uint32_t              Jabber;            // jabber 定时器功能
    uint32_t              InterFrameGap;     // 发送帧间间隙
    uint32_t              CarrierSense;      // 载波侦听
    uint32_t              ReceiveOwn;        // 接收自身
    uint32_t              LoopbackMode;      // 回送模式
    uint32_t              ChecksumOffload;     // 校验和减荷
    uint32_t              RetryTransmission;   // 传输重试
    uint32_t              AutomaticPadCRCStrip;  // 自动去除 PAD 和 FCS 字段
    uint32_t              BackOffLimit;      // 后退限制
    uint32_t              DeferralCheck;     // 检查延迟
    uint32_t              ReceiveAll;        // 接收所有 MAC 帧
    uint32_t              SourceAddrFilter;    // 源地址过滤
```

(下页继续)

(续上页)

```
    uint32_t               PassControlFrames;    // 传送控制帧
    uint32_t               BroadcastFramesReception;// 广播帧接收
    uint32_t               DestinationAddrFilter; // 目标地址过滤
    uint32_t               PromiscuousMode;      // 混合模式
    uint32_t               MulticastFramesFilter; // 多播源地址过滤
    uint32_t               UnicastFramesFilter;    // 单播源地址过滤
    uint32_t               HashTableHigh;      // 散列表高位
    uint32_t               HashTableLow;       // 散列表低位
    uint32_t               PauseTime;        // 暂停时间
    uint32_t               ZeroQuantaPause;      // 零时间片暂停
    uint32_t               PauseLowThreshold;        // 暂停阈值下限
    uint32_t               UnicastPauseFrameDetect; // 单播暂停帧检测
    uint32_t               ReceiveFlowControl;     // 接收流控制
    uint32_t               TransmitFlowControl;    // 发送流控制
    uint32_t               VLANTagComparison;    // VLAN 标记比较
    uint32_t               VLANTagIdentifier;    // VLAN 标记标识符
} ETH_MACInitTypeDef;
```

- Watchdog：以太网看门狗功能选择，可选使能或禁止，它设定以太网 MAC 配置寄存器 (ETH_MACCR) 的 WD 位的值。如果设置为 1，使能看门狗，在接收 MAC 帧超过 2048 字节时自动切断后面数据，一般选择使能看门狗。如果设置为 0，禁用看门狗，最长可接收 16384 字节的帧。
- Jabber：jabber 定时器功能选择，可选使能或禁止，与看门狗功能类似，只是看门狗用于接收 MAC 帧，jabber 定时器用于发送 MAC 帧，它设定 ETH_MACCR 寄存器的 JD 位的值。如果设置为 1，使能 jabber 定时器，在发送 MAC 帧超过 2048 字节时自动切断后面数据，一般选择使能 jabber 定时器。
- InterFrameGap：控制发送帧间的最小间隙，可选 96bit 时间、88bit 时间、…、40bit 时间，他设定 ETH_MACCR 寄存器的 IFG[2:0] 位的值，一般设置 96bit 时间。
- CarrierSense：载波侦听功能选择，可选使能或禁止，它设定 ETH_MACCR 寄存器的 CSD 位的值。当被设置为低电平时，MAC 发送器会生成载波侦听错误，一般使能载波侦听功能。

- ReceiveOwn：接收自身帧功能选择，可选使能或禁止，它设定 ETH_MACCR 寄存器的 ROD 位的值，当设置为 0 时，MAC 接收发送时 PHY 提供的所有 MAC 包，如果设置为 1，MAC 禁止在半双工模式下接收帧。一般使能接收。
- LoopbackMode：回送模式选择，可选使能或禁止，它设定 ETH_MACCR 寄存器的 LM 位的值，当设置为 1 时，使能 MAC 在 MII 回送模式下工作。
- ChecksumOffload：IPv4 校验和减荷功能选择，可选使能或禁止，它设定 ETH_MACCR 寄存器 IPCO 位的值，当该位被置 1 时使能接收的帧有效载荷的 TCP/UDP/ICMP 标头的 IPv4 校验和检查。一般选择禁用，此时 PCE 和 IPHCE 状态位总是为 0。
- RetryTransmission：传输重试功能，可选使能或禁止，它设定 ETH_MACCR 寄存器 RD 位的值，当被设置为 1 时，MAC 仅尝试发送一次，设置为 0 时，MAC 会尝试根据 BL 的设置进行重试。一般选择使能重试。
- AutomaticPadCRCStrip：自动去除 PAD 和 FCS 字段功能，可选使能或禁用，它设定 ETH_MACCR 寄存器 APCS 位的值。当设置为 1 时，MAC 在长度字段值小于或等于 1500 自己是去除传入帧上的 PAD 和 FCS 字段。一般禁止自动去除 PAD 和 FCS 字段功能。
- BackOffLimit：后退限制，在发送冲突后重新安排发送的延迟时间，可选 10、8、4、1，它设定 ETH_MACCR 寄存器 BL 位的值。一般设置为 10。
- DeferralCheck：检查延迟，可选使能或禁止，它设定 ETH_MACCR 寄存器 DC 位的值，当设置为 0 时，禁止延迟检查功能，MAC 发送延迟，直到 CRS 信号变成无效信号。
- ReceiveAll：接收所有 MAC 帧，可选使能或禁用，它设定以太网 MAC 帧过滤寄存器(ETH_MACFFR)RA 位的值。当设置为 1 时，MAC 接收器将所有接收的帧传送到应用程序，不过滤地址。当设置为 0 是，MAC 接收会自动过滤不与 SA/DA 匹配的帧。一般选择不接收所有。
- SourceAddrFilter：源地址过滤，可选源地址过滤、源地址反向过滤或禁用源地址过滤，它设定 ETH_MACFFR 寄存器 SAF 位和 SAIF 位的值。一般选择禁用源地址过滤。
- PassControlFrames：传送控制帧，控制所有控制帧的转发，可选阻止所有控制帧到达应用程序、转发所有控制帧、转发通过地址过滤的控制帧，它设定 ETH_MACFFR 寄存器 PCF 位的值。一般选择禁止转发控制帧。
- BroadcastFramesReception：广播帧接收，可选使能或禁止，它设定 ETH_MACFFR 寄存器

BFD 位的值。当设置为 0 时，使能广播帧接收，一般设置接收广播帧。

- DestinationAddrFilter：目标地址过滤功能选择，可选正常过滤或目标地址反向过滤，它设定 ETH_MACFFR 寄存器 DAIF 位的值。一般设置为正常过滤。
- PromiscuousMode：混合模式，可选使能或禁用，它设定 ETH_MACFFR 寄存器 PM 位的值。当设置为 1 时，不论目标或源地址，地址过滤器都传送所有传入的帧。一般禁用混合模式。
- MulticastFramesFilter：多播源地址过滤，可选完美散列表过滤、散列表过滤、完美过滤或禁用过滤，它设定 ETH_MACFFR 寄存器 HPF 位、PAM 位和 HM 位的值。一般选择完美过滤。
- UnicastFramesFilter：单播源地址过滤，可选完美散列表过滤、散列表过滤或完美过滤，它设定 ETH_MACFFR 寄存器 HPF 位和 HU 位的值。一般选择完美过滤。
- HashTableHigh：散列表高位，和 HashTableLow 组成 64 位散列表用于组地址过滤，它设定以太网 MAC 散列表高位寄存器 (ETH_MACHTHR) 的值。
- HashTableLow：散列表低位，和 HashTableHigh 组成 64 位散列表用于组地址过滤，它设定以太网 MAC 散列表低位寄存器 (ETH_MACHTLR) 的值。
- PauseTime：暂停时间，保留发送控制帧中暂停时间字段要使用的值，可设置 0 至 65535，它设定以太网 MAC 流控制寄存器 (ETH_MACFCR)PT 位的值。
- ZeroQuantaPause：零时间片暂停，可选使用或禁止，它设定 ETH_MACFCR 寄存器 ZQPD 位的值。当设置为 1 时，当来自 FIFO 层的流控制信号去断言后，此位会禁止自动生成零时间片暂停控制帧。一般选择禁止。
- PauseLowThreshold：暂停阈值下限，配置暂停定时器的阈值，达到该值值时，会自动程序传输暂停帧，可选暂停时间减去 4 个间隙、28 个间隙、144 个间隙或 256 个间隙，它设定 ETH_MACFCR 寄存器 PLT 位的值。一般选择暂停时间减去 4 个间隙。
- UnicastPauseFrameDetect：单播暂停帧检测，可选使能或禁止，它设定 ETH_MACFCR 寄存器 UPFD 位的值。当设置为 1 时，MAC 除了检测具有唯一多播地址的暂停帧外，还会检测具有 ETH_MACA0HR 和 ETH_MACA0LR 寄存器所指定的站单播地址的暂停帧。一般设置为禁止。
- ReceiveFlowControl：接收流控制，可选使能或禁止，它设定 ETH_MACFCR 寄存器 RFCE 位的值。当设定为 1 时，MAC 对接收到的暂停帧进行解码，并禁止其在指定时间（暂停时间）

内发送；当设置为 0 时，将禁止暂停帧的解码功能，一般设置为禁止。

- TransmitFlowControl：发送流控制，可选使能或禁止，它设定 ETH_MACFCR 寄存器 TFCE 位的值。在全双工模式下，当设置为 1 时，MAC 将使能流控制操作来发送暂停帧；为 0 时，将禁止 MAC 中的流控制操作，MAC 不会传送任何暂停帧。在半双工模式下，当设置为 1 时，MAC 将使能背压操作；为 0 时，将禁止背压功能。
- VLANTagComparison：VLAN 标记比较，可选 12 位或 16 位，它设定以太网 MAC VLAN 标记寄存器 (ETH_MACVLANTR)VLANTC 位的值。当设置为 1 时，使用 12 位 VLAN 标识符而不是完整的 16 位 VLAN 标记进行比较和过滤；为 0 时，使用全部 16 位进行比较，一般选择 16 位。
- VLANTagIdentifier：VLAN 标记标识符，包含用于标识 VLAN 帧的 802.1Q VLAN 标记，并与正在接收的 VLAN 帧的第十五和第十六字节进行比较。位 [15:13] 是用户优先级，位 [12] 是标准格式指示符 (CFI)，位 [11:0] 是 VLAN 标记的 VLAN 标识符 (VID) 字段。VLANTC 位置 1 时，仅使用 VID（位 [11:0]）进行比较。

列表 3: 代码清单 36_1_3 ETH_DMAInitTypeDef

```
typedef struct {
  uint32_t              DropTCPIPChecksumErrorFrame;//丢弃 TCP/IP 校验错误帧
  uint32_t              ReceiveStoreForward;      // 接收存储并转发
  uint32_t              FlushReceivedFrame;       // 刷新接收帧
  uint32_t              TransmitStoreForward;     // 发送存储并转发
  uint32_t              TransmitThresholdControl;  // 发送阈值控制
  uint32_t              ForwardErrorFrames;       // 转发错误帧
  uint32_t              ForwardUndersizedGoodFrames; // 转发过小的好帧
  uint32_t              ReceiveThresholdControl;    // 接收阈值控制
  uint32_t              SecondFrameOperate;       // 处理第二个帧
  uint32_t              AddressAlignedBeats;      // 地址对齐节拍
  uint32_t              FixedBurst;           // 固定突发
  uint32_t              RxDMABurstLength;       // DMA 突发接收长度
  uint32_t              TxDMABurstLength;       // DMA 突发发送长度
  uint32_t              EnhancedDescriptorFormat;  // 增强描述符格式
```

(下页继续)

(续上页)

```
  uint32_t               DescriptorSkipLength;     // 描述符跳过长度
  uint32_t               DMAArbitration;          // DMA 仲裁
} ETH_DMAInitTypeDef;
```

- DropTCPIPChecksumErrorFrame：丢弃 TCP/IP 校验错误帧，可选使能或禁止，它设定以太网 DMA 工作模式寄存器 (ETH_DMAOMR)DTCEFD 位的值，当设置为 1 时，如果帧中仅存在由接收校验和减荷引擎检测出来的错误，则内核不会丢弃它；为 0 时，如果 FEF 为进行了复位，则会丢弃所有错误帧。
- ReceiveStoreForward：接收存储并转发，可选使能或禁止，它设定以太网 DMA 工作模式寄存器 (ETH_DMAOMR)RSF 位的值，当设置为 1 时，向 RX FIFO 写入完整帧后可以从中读取一帧，同时忽略接收阈值控制 (RTC) 位；当设置为 0 时，RX FIFO 在直通模式下工作，取决于 RTC 位的阈值。一般选择使能。
- FlushReceivedFrame：刷新接收帧，可选使能或禁止，它设定 ETH_DMAOMR 寄存器 FTF 位的值，当设置为 1 时，发送 FIFO 控制器逻辑会恢复到缺省值，TX FIFO 中的所有数据均会丢失/刷新，刷新结束后改为自动清零。
- TransmitStoreForward：发送存储并并转发，可选使能或禁止，它设定 ETH_DMAOMR 寄存器 TSF 位的值，当设置为 1 时，如果 TX FIFO 有一个完整的帧则发送会启动，会忽略 TTC 值；为 0 时，TTC 值才会有效。一般选择使能。
- TransmitThresholdControl：发送阈值控制，有多个阈值可选，它设定 ETH_DMAOMR 寄存器 TTC 位的值，当 TX FIFO 中帧大小大于该阈值时发送会自动，对于小于阈值的全帧也会发送。
- ForwardErrorFrames：转发错误帧，可选使能或禁止，它设定 ETH_DMAOMR 寄存器 FEF 位的值，当设置为 1 时，除了段错误帧之外所有帧都会转发到 DMA；为 0 时，RX FIFO 会丢弃滴啊有错误状态的帧。一般选择禁止。
- ForwardUndersizedGoodFrames：转发过小的好帧，可选使能或禁止，它设定 ETH_DMAOMR 寄存器 FUGF 位的值，当设置为 1 时，RX FIFO 会转发包括 PAD 和 FCS 字段的过小帧；为 0 时，会丢弃小于 64 字节的帧，除非接收阈值被设置为更低。
- ReceiveThresholdControl：接收阈值控制，当 RX FIFO 中的帧大小大于阈值时启动 DMA 传

输请求，可选 64 字节、32 字节、96 字节或 128 字节，它设定 ETH_DMAOMR 寄存器 RTC 位的值。

- SecondFrameOperate：处理第二个帧，可选使能或禁止，它设定 ETH_DMAOMR 寄存器 OSF 位的值，当设置为 1 时会命令 DMA 处理第二个发送数据帧。
- AddressAlignedBeats：地址对齐节拍，可选使能或禁止，它设定以太网 DMA 总线模式寄存器 (ETH_DMABMR)AAB 位的值，当设置为 1 并且固定突发位 (FB) 也为 1 时，AHB 接口会生成与起始地址 LS 位对齐的所有突发；如果 FB 位为 0，则第一个突发不对齐，但后续的突发与地址对齐。一般选择使能。
- FixedBurst：固定突发，控制 AHB 主接口是否执行固定突发传输，可选使能或禁止，它设定 ETH_DMABMR 寄存器 FB 位的值，当设置为 1 时，AHB 在正常突发传输开始期间使用 SINGLE、INCR4、INCR8 或 INCR16；为 0 时，AHB 使用 SINGLE 和 INCR 突发传输操作。
- RxDMABurstLength：DMA 突发接收长度，有多个值可选，一般选择 32Beat，可实现 32*32bits 突发长度，它设定 ETH_DMABMR 寄存器 FPM 位和 RDP 位的值。
- TxDMABurstLength：DMA 突发发送长度，有多个值可选，一般选择 32Beat，可实现 32*32bits 突发长度，它设定 ETH_DMABMR 寄存器 FPM 位和 PBL 位的值。
- EnhancedDescriptorFormat：增强描述符格式，可以使能或者禁止。该位置 1 时，使能增强描述符格式，并将描述符大小增加至 32 字节（8 个 DWORD）。如果已激活时间戳功能（ETH_PTPTSCR 位 0 TSE=1）或 IPv4 校验和减荷（ETH_MACCR 位 10 IPCO=1），则必须使用此增强描述符。
- DescriptorSkipLength：描述符跳过长度，指定两个未链接描述符之间跳过的字数，地址从当前描述符结束处开始跳到下一个描述符起始处，可选 0~7，它设定 ETH_DMABMR 寄存器 DSL 位的值。
- DMAArbitration：DMA 仲裁，控制 RX 和 TX 优先级，可选 RX TX 优先级比为 1:1、2:1、3:1、4:1 或者 RX 优先于 TX，它设定 ETH_DMABMR 寄存器 PM 位和 DA 位的值，当设置为 1 时，RX 优先于 TX；为 0 时，循环调度，RX TX 优先级比由 PM 位给出。

39.8 以太网通信实验：无操作系统 LwIP 移植

LwIP 可以在带操作系统上运行，亦可在无操作系统上运行，这一实验我们讲解在无操作系统的移植步骤，并实现简单的传输代码，后续章节会讲解在带操作系统移植过程，一般都是在无操作系统基础上修改而来的。

39.8.1 硬件设计

在讲解移植步骤之前，有必须先介绍我们的实验硬件设计，主要是 LAN8720A 通过 RMII 和 SMI 接口与 STM32F4xx 控制器连接，见图 *36_1_14*。

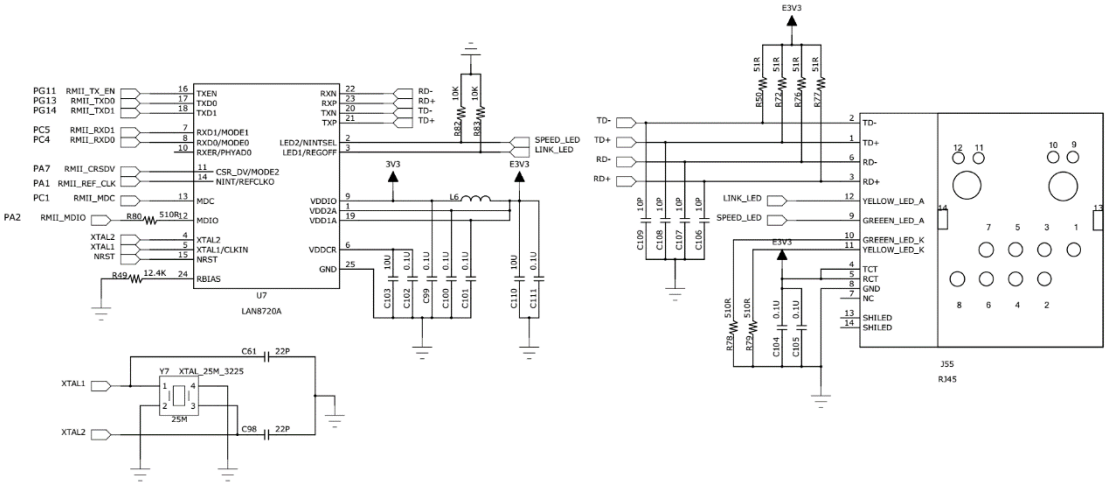


图 36_1_14 PHY 硬件设计

电路设计时，将 NINTSEL 引脚通过下拉电阻拉低，设置 NINT/FEFCLKO 为输出 50MHz 时钟，当然前提是在 XTAL1 和 XTAL2 接入了 25MHz 的时钟源。另外也把 REGOFF 引脚通过下拉电阻拉低，使能使用内部 +1.2V 稳压器。

39.8.2 移植步骤

之前已经介绍了 LwIP 源代码 (lwip-1.4.1.zip) 和 ST 官方 LwIP 测试平台资料 (stsw-stm32070.zip) 下载，我们移植步骤是基于这两份资料进行的。

无操作系统移植 LwIP 需要的文件参考图 *36_1_15*，图中只显示了 *.c 文件，还需要用到对应的 *.h 文件。

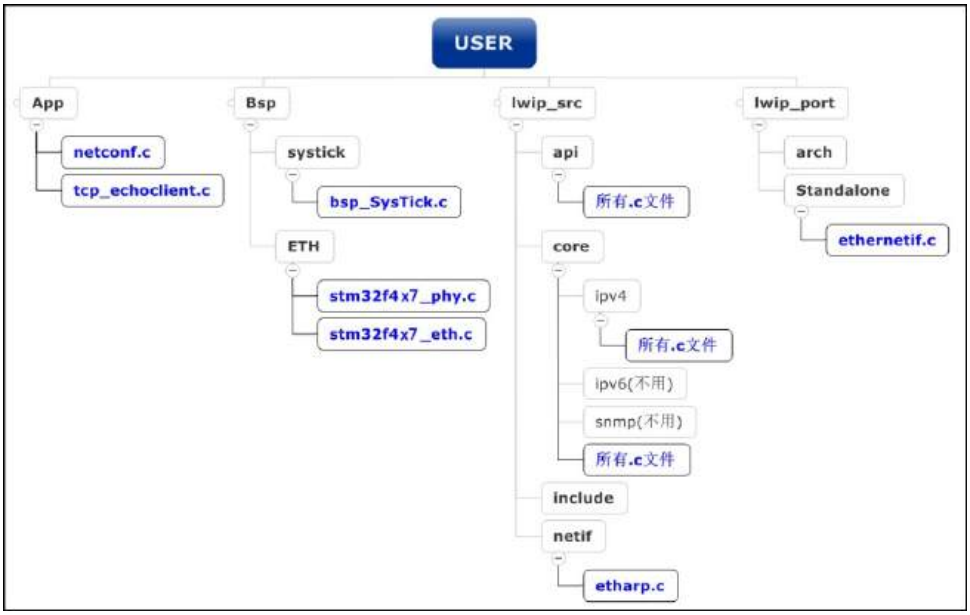


图 36_1_15 LwIP 移植实验文件结构

接下来，我们就根据图中文件结构详解移植过程。实验例程有需要用到系统滴答定时器 systick、调试串口 USART、独立按键 KEY、LED 灯功能，对这些功能实现不做具体介绍，可以参考相关章节理解。

39.8.2.1 第一步：相关文件拷贝

首先，解压 lwip-1.4.1.zip 和 stsw-stm32070.zip 两个压缩包，把整个 lwip-1.4.1 文件夹拷贝到 USER 文件夹下，特别说明，在整个移植过程中，不会对 lwip-1.4.1.zip 文件下的文件内容进行修改。然后，在 stsw-stm32070 文件夹找到 port 文件夹 (路径：…\UtilitiesThird_Partylwip-1.4.1port)，把整个 port 文件夹拷贝 lwip-1.4.1 文件夹中，在 port 文件夹下的 STM32F4x7 文件中把 arch 和 Standalone 两个文件夹直接剪切到 port 文件夹中，即此时 port 文件夹有三个 STM32F4x7、arch 和 Standalone

文件夹，最后把 STM32F4x7 文件夹删除，最终的文件结构见图 *36_1_16*，arch 存放与开发平台相关头文件，Standalone 文件夹是无操作系统移植时 ETH 外设与 LwIP 连接的底层驱动函数。

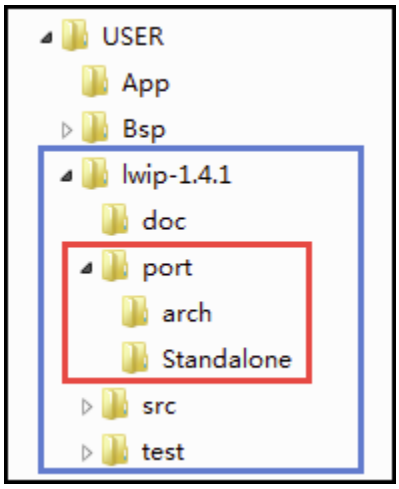


图 36_1_16 LwIP 相关文件拷贝

lwip-1.4.1 文件夹下的 doc 文件夹存放 LwIP 版权、移植、使用等等说明文件，移植之前有必须认真浏览一遍；src 文件夹存放 LwIP 的实现代码，也是我们工程代码真正需要的文件；test 文件夹存放 LwIP 部分功能测试例程；另外，还有一些无后缀名的文件，都是一些说明性文件，可用记事本直接打开浏览。port 文件夹存放 LwIP 与 STM32 平台连接的相关文件，正如上面所说 contrib-1.4.1.zip 包含了不同平台移植代码，不过遗憾地是没有 STM32 平台的，所以我们需要从 ST 官方提供的测试平台找到这部分连接代码，也就是 port 文件夹的内容。

接下来，在 Bsp 文件下新建一个 ETH 文件夹，用于存放与 ETH 相关驱动文件，包括两个部分文件，其中一个是 ETH 外设驱动文件，在 stsw-stm32070 文件夹中找到 stm32f4x7_eth.h 和 stm32f4x7_eth.c 两个文件 (路径：…LibrariesSTM32F4x7_ETH_Driver)，将这两个文件拷贝到 ETH 文件夹中，这两个文件是 ETH 驱动文件，类似 HAL 库中外设驱动代码实现文件，在移植过程中我们几乎不过文件的内容。这部分函数由 port 文件夹相关代码调用。另外一部分是相关 GPIO 初始化、ETH 外设初始化、PHY 状态获取等等函数的实现，在 stsw-stm32070 文件夹中找到 stm32f4x7_eth_bsp.c、stm32f4x7_eth_bsp.h 和 stm32f4x7_eth_conf.h 三个文件 (路径：…Project-Standalonetcp_echo_client)，将这三个文件拷贝到 ETH 文件夹中。因为 ST 官方 LwIP 测试平台使用的 PHY 型号不是使用 LAN8720A，所以这三个文件需要我们进行修改。

最后，是 LwIP 测试代码实现，为测试 LwIP 移植是否成功和检查 LwIP 功能，我们编写 TCP 通信实现代码，设置开发板为 TCP 从机，电脑端为 TCP 主机。在 stsw-stm32070 文件夹中找到

netconf.c、tcp_echoclient.c、lwipopts.h、netconf.h 和 tcp_echoclient.h 五个文件 (路径：…ProjectStandalonetcp_echo_client)，直接拷贝到 App 文件夹 (自己新建) 中，netconf.c 文件代码实现 LwIP 初始化函数、周期调用函数、DHCP 功能函数等等，tcp_echoclient.c 文件实现 TCP 通信参数代码，lwipopts.h 包含 LwIP 功能选项。

39.8.2.2 第二部：为工程添加文件

第一步已经把相关的文件拷贝到对应的文件夹中，接下来就可以把需要用到的文件添加到工程中。图 *36_1_15* 已经指示出来工程需要用到的 *.c 文件，所以最终工程文件结构见图 *36_1_17*，图中 api、ipv4 和 core 都包含了对应文件夹下的所有 *.c 文件。

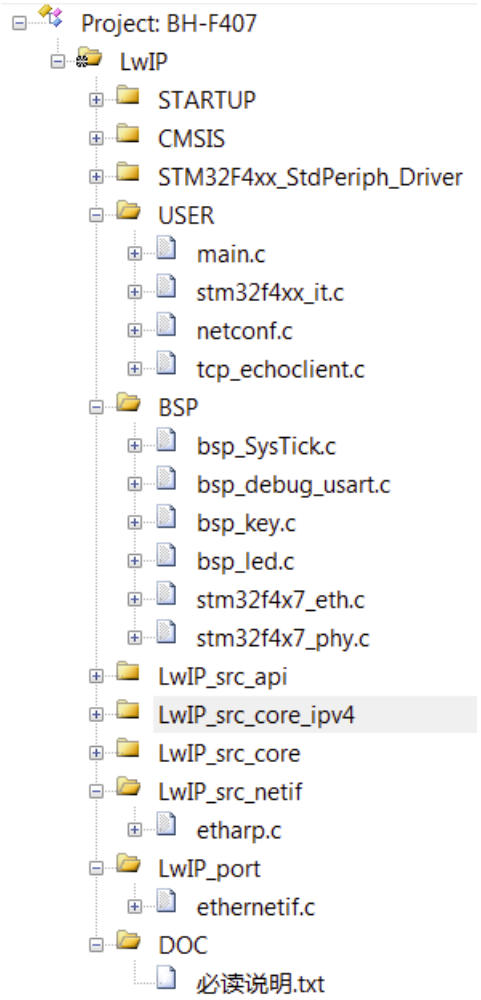


图 36_1_17 工程文件结构

接下来，还需要在工程选择中添加相关头文件路径，参考图 *36_1_18*。

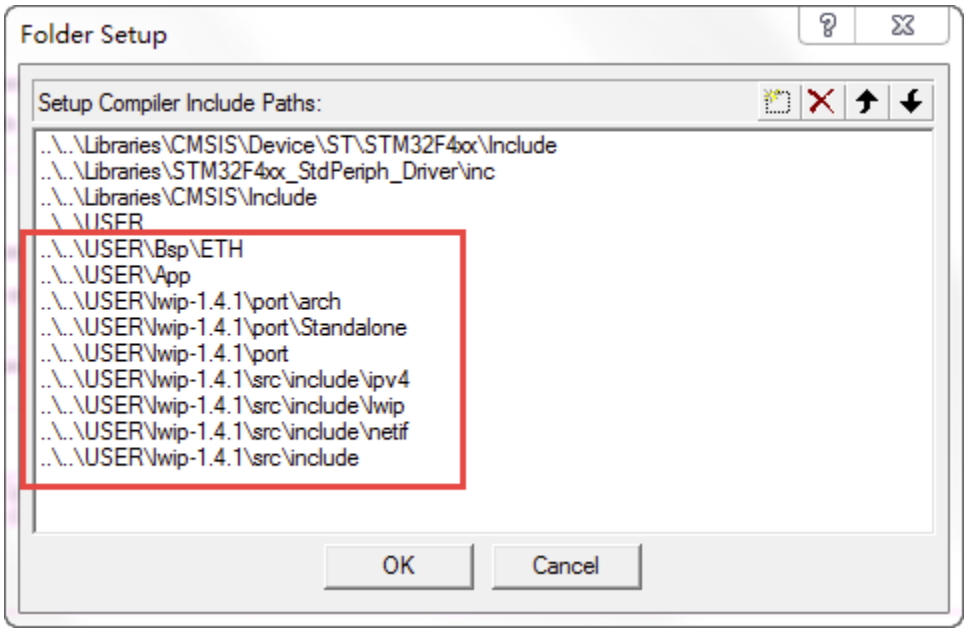


图 36_1_18 添加相关头文件路径

39.8.2.3 第三步：文件修改

ethernetif.c 文件是无操作系统时网络接口函数，该文件在移植时需要根据实际硬件初始化网络相关 IO 口，以及需要指定的 SRAM 空间作为缓存。该文件主要有三个部分函数，HAL_ETH_MspInit 函数用于初始化系统硬件接口；low_level_init 函数用于初始化 MAC 相关工作环境、初始化 DMA 描述符链表，并使能 MAC 和 DMA；low_level_output 函数是最底层发送一帧数据函数；low_level_input 函数是最底层接收一帧数据函数。sys_now 函数获取当前时间的一个函数；ethernetif_init 函数初始化网络接口结构（netif）并调用 low_level_init 以初始化以太网外设；ethernet_input 函数调用 low_level_input 接收包，然后将其提供给 LwIP 栈。

app_ethernet.c 文件主要是实际的网络初始化应用程序，这里包含两个函数，Netif_Config 函数是创建一个网络接口；User_notification 函数是指示当前网络连接的状态。

LAN8720A.h 和 LAN8720A.c 两个文件是 ETH 外设相关的底层配置，主要是 GPIO 初始化即相关时钟使能。

列表 4: 代码清单 36_1_4 ETH_GPIO_Config 函数

```
void ETH_GPIO_Config(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;
   /* 使能端口时钟 */
   ETH_MDIO_GPIO_CLK_ENABLE();
   ETH_MDC_GPIO_CLK_ENABLE();
   ETH_RMII_REF_CLK_GPIO_CLK_ENABLE();
   ETH_RMII_CRS_DV_GPIO_CLK_ENABLE();
   ETH_RMII_RXD0_GPIO_CLK_ENABLE();
   ETH_RMII_RXD1_GPIO_CLK_ENABLE();
   ETH_RMII_TX_EN_GPIO_CLK_ENABLE();
   ETH_RMII_TXD0_GPIO_CLK_ENABLE();
   ETH_RMII_TXD1_GPIO_CLK_ENABLE();

   /* 配置以太网引脚 */
   /*
   ETH_MDIO -------------------------> PA2
   ETH_MDC --------------------------> PC1
   ETH_MII_RX_CLK/ETH_RMII_REF_CLK---> PA1
   ETH_MII_RX_DV/ETH_RMII_CRS_DV ----> PA7
   ETH_MII_RXD0/ETH_RMII_RXD0 -------> PC4
   ETH_MII_RXD1/ETH_RMII_RXD1 -------> PC5
   ETH_MII_TX_EN/ETH_RMII_TX_EN -----> PB11
   ETH_MII_TXD0/ETH_RMII_TXD0 -------> PG13
   ETH_MII_TXD1/ETH_RMII_TXD1 -------> PG14
   */

   /* 配置 ETH_MDIO 引脚 */
   GPIO_InitStructure.Pin = ETH_MDIO_PIN;
   GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Alternate = ETH_MDIO_AF;
    HAL_GPIO_Init(ETH_MDIO_PORT, &GPIO_InitStructure);

    /* 配置 ETH_MDC 引脚 */
    GPIO_InitStructure.Pin = ETH_MDC_PIN;
    GPIO_InitStructure.Alternate = ETH_MDC_AF;
    HAL_GPIO_Init(ETH_MDC_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_REF_CLK 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_REF_CLK_PIN;
    GPIO_InitStructure.Alternate = ETH_RMII_REF_CLK_AF;
    HAL_GPIO_Init(ETH_RMII_REF_CLK_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_CRS_DV 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_CRS_DV_PIN;
    GPIO_InitStructure.Alternate = ETH_RMII_CRS_DV_AF;
    HAL_GPIO_Init(ETH_RMII_CRS_DV_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_RXD0 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_RXD0_PIN;
    GPIO_InitStructure.Alternate = ETH_RMII_RXD0_AF;
    HAL_GPIO_Init(ETH_RMII_RXD0_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_RXD1 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_RXD1_PIN;
    GPIO_InitStructure.Alternate = ETH_RMII_RXD1_AF;
    HAL_GPIO_Init(ETH_RMII_RXD1_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_TX_EN 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_TX_EN_PIN;
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Alternate = ETH_RMII_TX_EN_AF;
    HAL_GPIO_Init(ETH_RMII_TX_EN_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_TXD0 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_TXD0_PIN;
    GPIO_InitStructure.Alternate = ETH_RMII_TXD0_AF;
    HAL_GPIO_Init(ETH_RMII_TXD0_PORT, &GPIO_InitStructure);

    /* 配置 ETH_RMII_TXD1 引脚 */
    GPIO_InitStructure.Pin = ETH_RMII_TXD1_PIN;
    GPIO_InitStructure.Alternate = ETH_RMII_TXD1_AF;
    HAL_GPIO_Init(ETH_RMII_TXD1_PORT, &GPIO_InitStructure);
}
```

HAL_ETH_MspInit 函数调用 ETH_GPIO_Config 进行硬件初始化，并使能以太网时钟。

列表 5: 代码清单 36_1_5 HAL_ETH_MspInit 函数

```
/**
    * @brief  以太网硬件底层驱动
    * @param  heth: 以太网句柄
    * @retval None
    */
void HAL_ETH_MspInit(ETH_HandleTypeDef *heth)
{
    ETH_GPIO_Config();
    /* 使能以太网时钟  */
    __HAL_RCC_ETH_CLK_ENABLE();
}
```

low_level_init 主要是初始化硬件外设，最终被 ethernetif_init 函数调用。

列表 6: 代码清单 36_1_6 low_level_init 函数

```
/**
* @brief 在这个函数中初始化硬件
*       最终被 ethernetif_init 函数调用
*
* @param netif 已经初始化了这个以太网的 lwip 网络接口结构
*/
static void low_level_init(struct netif *netif)
{
uint8_t macaddress[6]= { MAC_ADDR0, MAC_ADDR1, MAC_ADDR2,
      MAC_ADDR3, MAC_ADDR4, MAC_ADDR5 };

   EthHandle.Instance = ETH;
   EthHandle.Init.MACAddr = macaddress;
   EthHandle.Init.AutoNegotiation = ETH_AUTONEGOTIATION_ENABLE;//使能自协商模
式
   EthHandle.Init.Speed = ETH_SPEED_100M;//网络速率 100M
   EthHandle.Init.DuplexMode = ETH_MODE_FULLDUPLEX;//全双工模式
   EthHandle.Init.MediaInterface = ETH_MEDIA_INTERFACE_RMII;//RMII 接口
   EthHandle.Init.RxMode = ETH_RXPOLLING_MODE;//轮询接收模式
   EthHandle.Init.ChecksumMode = ETH_CHECKSUM_BY_HARDWARE;//硬件帧校验
   EthHandle.Init.PhyAddress = LAN8720A_PHY_ADDRESS;//PHY 地址

   /* 配置以太网外设 (GPIOs, clocks, MAC, DMA) */
   if (HAL_ETH_Init(&EthHandle) == HAL_OK) {
      /* 设置 netif 链接标志 */
      netif->flags |= NETIF_FLAG_LINK_UP;
   }

   /* 初始化 Tx 描述符列表: 链接模式 */
   HAL_ETH_DMATxDescListInit(&EthHandle, DMATxDscrTab, &Tx_Buff[0][0], ETH_
↪TXBUFNB);
```

(下页继续)

(续上页)

```
    /* 初始化 Rx 描述符列表: 链接模式 */
    HAL_ETH_DMARxDescListInit(&EthHandle, DMARxDscrTab, &Rx_Buff[0][0], ETH_
↪RXBUFNB);

    /* 设置 netif MAC 硬件地址长度 */
    netif->hwaddr_len = ETHARP_HWADDR_LEN;

    /* 设置 netif MAC 硬件地址 */
    netif->hwaddr[0] =  MAC_ADDR0;
    netif->hwaddr[1] =  MAC_ADDR1;
    netif->hwaddr[2] =  MAC_ADDR2;
    netif->hwaddr[3] =  MAC_ADDR3;
    netif->hwaddr[4] =  MAC_ADDR4;
    netif->hwaddr[5] =  MAC_ADDR5;

    /* 设置 netif 最大传输单位 */
    netif->mtu = 1500;

    /* 接收广播地址和 ARP 流量 */
    netif->flags |= NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP;

    /* 使能 MAC 和 DMA 发送和接收 */
    HAL_ETH_Start(&EthHandle);
}
```

首先是 ETH_HandleTypeDef 结构体填充，关于结构体各个成员意义已在“ETH 初始化结构体详解”作了分析。然后调用系统函数 HAL_ETH_Init 初始化以太网外设。初始化相关描述符的列表，设置 MAC 地址，使能 MAC 和 DMA 发送和接收。

Netif_Config 函数一般在 main 函数中在 LwIP_Init 函数初始化完成后调用。

列表 7: 代码清单 36_1_7 Netif_Config 函数

```
#define DEST_IP_ADDR0    (uint8_t)192
#define DEST_IP_ADDR1    (uint8_t)168
#define DEST_IP_ADDR2    (uint8_t)31
#define DEST_IP_ADDR3    (uint8_t)198

#define DEST_PORT        (uint8_t)7

/*Static IP ADDRESS: IP_ADDR0.IP_ADDR1.IP_ADDR2.IP_ADDR3 */
#define IP_ADDR0    (uint8_t) 192
#define IP_ADDR1    (uint8_t) 168
#define IP_ADDR2    (uint8_t) 31
#define IP_ADDR3    (uint8_t) 122

/*NETMASK*/
#define NETMASK_ADDR0    (uint8_t) 255
#define NETMASK_ADDR1    (uint8_t) 255
#define NETMASK_ADDR2    (uint8_t) 255
#define NETMASK_ADDR3    (uint8_t) 0

/*Gateway Address*/
#define GW_ADDR0    (uint8_t) 192
#define GW_ADDR1    (uint8_t) 168
#define GW_ADDR2    (uint8_t) 31
#define GW_ADDR3    (uint8_t) 1
/**
* @brief  建立网络接口
* @param  None
* @retval None
*/
void Netif_Config(void)
```

(下页继续)

(续上页)

```
{
   ip_addr_t ipaddr;
   ip_addr_t netmask;
   ip_addr_t gw;

   IP_ADDR4(&ipaddr,IP_ADDR0,IP_ADDR1,IP_ADDR2,IP_ADDR3);
   IP_ADDR4(&netmask,NETMASK_ADDR0,NETMASK_ADDR1,NETMASK_ADDR2,NETMASK_
↪ADDR3);
   IP_ADDR4(&gw,GW_ADDR0,GW_ADDR1,GW_ADDR2,GW_ADDR3);

   /* 添加网络接口 */
   netif_add(&gnetif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &
↪ethernet_input);

   /* 注册默认网络接口 */
   netif_set_default(&gnetif);

   if (netif_is_link_up(&gnetif)) {
      /* 当 netif 完全配置时，必须调用此函数 */
      netif_set_up(&gnetif);
   } else {
      /* 当 netif 链接断开时，必须调用此函数 */
      netif_set_down(&gnetif);
   }
}
```

通过宏定义了远端 IP 和端口、MAC 地址、静态 IP 地址、子网掩码、网关相关宏，可以根据实际情况修改。netif_add 函数添加网络接口；netif_set_default 注册默认网络接口。

列表 8: 代码清单 36_1_8 User_notification 函数

```
/**
* @brief  通知用户有关网络接口配置状态
* @param  netif: 网络接口
* @retval None
*/
void User_notification(struct netif *netif)
{
   if (netif_is_up(netif)) {
      printf("Static IP: %d.%d.%d.%d\n",
IP_ADDR0,IP_ADDR1,IP_ADDR2,IP_ADDR3);
      printf("NETMASK  : %d.%d.%d.%d\n",
NETMASK_ADDR0,NETMASK_ADDR1,NETMASK_ADDR2,NETMASK_ADDR3);
      printf("Gateway  : %d.%d.%d.%d\n",
GW_ADDR0,GW_ADDR1,GW_ADDR2,GW_ADDR3);
      LED_GREEN;
   } else {
      printf ("The network cable is not connected \n");
      LED_RED;
   }
}
```

User_notification 函数在网络接口配置完成后调用，通知用户有关网络接口配置状态。打印接口连接状态，LED 指示连接状态。

列表 9: 代码清单 36_1_9 lwip_Init 函数

```
/**
* @ingroup lwip_nosys
* 初始化所有模块.
* 在 NO_SYS 模式下使用, 否则使用 tcpip_init ()。
*/
```

(下页继续)

(续上页)

```
void
lwip_init(void)
{
#ifndef LWIP_SKIP_CONST_CHECK
   int a = 0;
   LWIP_UNUSED_ARG(a);
   LWIP_ASSERT("LWIP_CONST_CAST not implemented correctly.
Check your lwIP port.", LWIP_CONST_CAST(void*, &a) == &a);
#endif
#ifndef LWIP_SKIP_PACKING_CHECK
   LWIP_ASSERT("Struct packing not implemented correctly.
Check your lwIP port.", sizeof(struct packed_struct_test) ==
      PACKED_STRUCT_TEST_EXPECTED_SIZE);
#endif

   /* 模块初始化 */
   stats_init();
#if !NO_SYS
   sys_init();
#endif /* !NO_SYS */
   mem_init();
   memp_init();
   pbuf_init();
   netif_init();
#if LWIP_IPV4
   ip_init();
#if LWIP_ARP
   etharp_init();
#endif /* LWIP_ARP */
#endif /* LWIP_IPV4 */
#if LWIP_RAW
   raw_init();
```

(下页继续)

(续上页)

```
#endif /* LWIP_RAW */
#if LWIP_UDP
   udp_init();
#endif /* LWIP_UDP */
#if LWIP_TCP
   tcp_init();
#endif /* LWIP_TCP */
#if LWIP_IGMP
   igmp_init();
#endif /* LWIP_IGMP */
#if LWIP_DNS
   dns_init();
#endif /* LWIP_DNS */
#if PPP_SUPPORT
   ppp_init();
#endif

#if LWIP_TIMERS
   sys_timeouts_init();
#endif /* LWIP_TIMERS */
}
```

lwip_Init 函数用于初始化 LwIP 协议栈，一般在 main 函数中调用。首先是内存相关初始化，mem_init 函数是动态内存堆初始化，memp_init 函数是存储池初始化，LwIP 是实现内存的高效利用，内部需要不同形式的内存管理模式。

pbuf 函数为预留的函数，目前是一个空操作。netif_init 函数多播的时候用到，本例没有用到。后面的功能都是通过 lwipopts.h 进行裁剪。

列表 10: 代码清单 36_1_10 ethernetif_input 函数

```
/**
* @brief 当数据包准备好从接口读取时，应该调用此函数。
* 它使用应该处理来自网络接口的字节的实际接收的函数 low_level_input。
* 然后确定接收到的分组的类型，并调用适当的输入功能。
*
* @param netif 以太网的 lwip 网络接口结构
*/
void ethernetif_input(struct netif *netif)
{
   err_t err;
   struct pbuf *p;

   /* 将接收到的数据包移动到新的 pbuf 中 */
   p = low_level_input(netif);

   /* 没有数据包可以读取，直接返回 */
   if (p == NULL) return;

   /* 到 LwIP 堆栈入口 */
   err = netif->input(p, netif);

   if (err != ERR_OK) {
      LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n"));
      pbuf_free(p);
      p = NULL;
   }
}
```

ethernetif_input 函数用于从以太网存储器读取一个以太网帧并将其发送给 LwIP，它在接收到以太网帧时被调用，它是直接调用 low_level_input 函数实现的，该函数定义在 ethernetif.c 文件中。

列表 11: 代码清单 36_1_11 sys_check_timeouts 函数

```
/**
* @ingroup lwip_nosys
* 处理 NO_SYS==1 超时 (即不使用 tcpip_thread/sys_timeouts_mbox_fetch())
* 使用 sys_now() 函数，当超时到期时调用超时处理函数。
* 必须定期从主循环中调用。
*/
#if !NO_SYS && !defined __DOXYGEN__
static
#endif /* !NO_SYS */
void
sys_check_timeouts(void)
{
   if (next_timeout) {
         struct sys_timeo *tmptimeout;
         u32_t diff;
         sys_timeout_handler handler;
         void *arg;
         u8_t had_one;
         u32_t now;

         now = sys_now();
         /* this cares for wraparounds */
         diff = now - timeouts_last_time;
         do {
            PBUF_CHECK_FREE_OOSEQ();
            had_one = 0;
            tmptimeout = next_timeout;
            if (tmptimeout && (tmptimeout->time <= diff)) {
               /* timeout has expired */
               had_one = 1;
```

(下页继续)

(续上页)

```
                timeouts_last_time += tmptimeout->time;
                diff -= tmptimeout->time;
                next_timeout = tmptimeout->next;
                handler = tmptimeout->h;
                arg = tmptimeout->arg;
#if LWIP_DEBUG_TIMERNAMES
                if (handler != NULL) {
                LWIP_DEBUGF(TIMERS_DEBUG, ("sct calling h=%s arg=%p\n",
                                                tmptimeout->handler_name,␣
↪arg));
                }
#endif /* LWIP_DEBUG_TIMERNAMES */
                memp_free(MEMP_SYS_TIMEOUT, tmptimeout);
                if (handler != NULL) {
#if !NO_SYS
          /* For LWIP_TCPIP_CORE_LOCKING, lock the core before calling the
                        timeout handler function. */
                    LOCK_TCPIP_CORE();
#endif /* !NO_SYS */
                    handler(arg);
#if !NO_SYS
                    UNLOCK_TCPIP_CORE();
#endif /* !NO_SYS */
                }
                LWIP_TCPIP_THREAD_ALIVE();
              }
              /* repeat until all expired timers have been called */
          } while (had_one);
    }
}
```

sys_check_timeouts 函数是一个必须被无限循环调用的 LwIP 支持函数，一般在 main 函数的无限

循环中调用，使用 sys_now() 函数，当超时到期时调用超时处理函数。

列表 12: 代码清单 36_1_12 LwIP_DHCP_Process_Handle 函数

```
void LwIP_DHCP_Process_Handle(void)
{
   struct ip_addr ipaddr;
   struct ip_addr netmask;
   struct ip_addr gw;

   switch (DHCP_state) {
   case DHCP_START: {
      DHCP_state = DHCP_WAIT_ADDRESS;
      dhcp_start(&gnetif);
      /* IP address should be set to 0
         every time we want to assign a new DHCP address */
      IPaddress = 0;
#ifdef SERIAL_DEBUG
      printf("\n     Looking for    \n");
      printf("     DHCP server    \n");
      printf("     please wait... \n");
#endif /* SERIAL_DEBUG */
   }
   break;

   case DHCP_WAIT_ADDRESS: {
      /* Read the new IP address */
      IPaddress = gnetif.ip_addr.addr;

      if (IPaddress!=0) {
            DHCP_state = DHCP_ADDRESS_ASSIGNED;
            /* Stop DHCP */
```

(下页继续)

(续上页)

```
            dhcp_stop(&gnetif);
#ifdef SERIAL_DEBUG
            printf("\n  IP address assigned \n");
            printf("    by a DHCP server   \n");
            printf("IP: %d.%d.%d.%d\n",(uint8_t)(IPaddress),
                        (uint8_t)(IPaddress >> 8),(uint8_t)(IPaddress >>␣
↪16),
                        (uint8_t)(IPaddress >> 24));
            printf("NETMASK: %d.%d.%d.%d\n",NETMASK_ADDR0,NETMASK_ADDR1,
                                                NETMASK_ADDR2,NETMASK_
↪ADDR3);
            printf("Gateway: %d.%d.%d.%d\n",GW_ADDR0,GW_ADDR1,
                                                    GW_ADDR2,GW_ADDR3);
            LED1_ON;
#endif /* SERIAL_DEBUG */
      } else {
            /* DHCP timeout */
            if (gnetif.dhcp->tries > MAX_DHCP_TRIES) {
               DHCP_state = DHCP_TIMEOUT;
               /* Stop DHCP */
               dhcp_stop(&gnetif);
               /* Static address used */
               IP4_ADDR(&ipaddr, IP_ADDR0 ,IP_ADDR1 , IP_ADDR2 , IP_ADDR3 );
               IP4_ADDR(&netmask, NETMASK_ADDR0, NETMASK_ADDR1,
                                        NETMASK_ADDR2, NETMASK_ADDR3);
               IP4_ADDR(&gw, GW_ADDR0, GW_ADDR1, GW_ADDR2, GW_ADDR3);
               netif_set_addr(&gnetif, &ipaddr , &netmask, &gw);
#ifdef SERIAL_DEBUG
               printf("\n    DHCP timeout    \n");
               printf("  Static IP address   \n");
               printf("IP: %d.%d.%d.%d\n",IP_ADDR0,IP_ADDR1,
                                                IP_ADDR2,IP_ADDR3);
```

(下页继续)

(续上页)

```
                printf("NETMASK: %d.%d.%d.%d\n",NETMASK_ADDR0,NETMASK_ADDR1,
                                                        NETMASK_ADDR2,NETMASK_
↪ADDR3);
                printf("Gateway: %d.%d.%d.%d\n",GW_ADDR0,GW_ADDR1,
                                                        GW_ADDR2,GW_ADDR3);
                LED1_ON;
#endif /* SERIAL_DEBUG */
              }
        }
    }
    break;
    default:
       break;
    }
}
```

LwIP_DHCP_Process_Handle 函数用于执行 DHCP 功能，当 DHCP 状态为 DHCP_START 时，执行 dhcp_start 函数启动 DHCP 功能，LwIP 会向 DHCP 服务器申请分配 IP 请求，并进入等待分配状态。当 DHCP 状态为 DHCP_WAIT_ADDRESS 时，先判断 IP 地址是否为 0，如果不为 0 说明已经有 IP 地址，DHCP 功能已经完成可以停止它；如果 IP 地址总是为 0，就需要判断是否超过最大等待时间，并提示出错。

lwipopts.h 文件存放一些宏定义，用于剪切 LwIP 功能，比如有无操作系统、内存空间分配、存储池分配、TCP 功能、DHCP 功能、UDP 功能选择等等。这里使用与 ST 官方例程相同配置即可。

列表 13: 代码清单 36_1_13 main 函数

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
```

(下页继续)

(续上页)

```
int main(void)
{

   /* 配置系统时钟为 168 MHz */
   SystemClock_Config();

   /* 初始化 RGB 彩灯 */
   LED_GPIO_Config();

   /* 初始化 USART1 配置模式为 115200 8-N-1 */
   UARTx_Config();

   /* 初始化 LwIP 协议栈 */
   lwip_init();

   printf("LAN8720A Ethernet Demo\n");
   printf("LwIP 版本: %s\n",LWIP_VERSION_STRING);

   printf("ping 实验例程\n");

   printf("使用同一个局域网中的电脑 ping 开发板的地址，可进行测试\n");

   //IP 地址和端口可在 main.h 文件修改
   printf("本地 IP 和端口: %d.%d.%d.%d\n",
IP_ADDR0,IP_ADDR1,IP_ADDR2,IP_ADDR3);
   /* 网络接口配置 */
   Netif_Config();
   /* 报告用户网络连接状态 */
   User_notification(&gnetif);

   while (1) {
      /* 从以太网缓冲区中读取数据包，交给 LwIP 处理 */
```

(下页继续)

(续上页)

```
        ethernetif_input(&gnetif);
        /* 处理 LwIP 超时 */
        sys_check_timeouts();
    }
}
```

首先是使能指令缓存、数据缓存，初始化系统时钟、LED 指示灯、按键、调试串口，lwip_init 函数初始化 LwIP 协议栈。通过 Netif_Config 函数配置网络接口；通过 User_notification 函数报告用户网络连接状态。进入无限循环函数，调用 ethernetif_input 函数从以太网缓存中读取数据包并交给 LwIP 处理；调用 sys_check_timeouts 函数处理 LwIP 超时。这两个函数必须在大循环中调用。

39.8.2.4 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并配置好相关参数；使用网线连接开发板网口跟路由器，这里要求电脑连接在同一个路由器上，之所以使用路由器是这样连接方便，电脑端无需更多操作步骤，并且路由器可以提供 DHCP 服务器功能，而电脑是不行的，最后在电脑端打开网络调试助手软件，并设置相关参数，见图 *36_1_19*，调试助手的设置与 netconf.h 文件中相关宏定义是对应的，不同电脑设置情况可能不同。把编译好的程序下载到开发板。

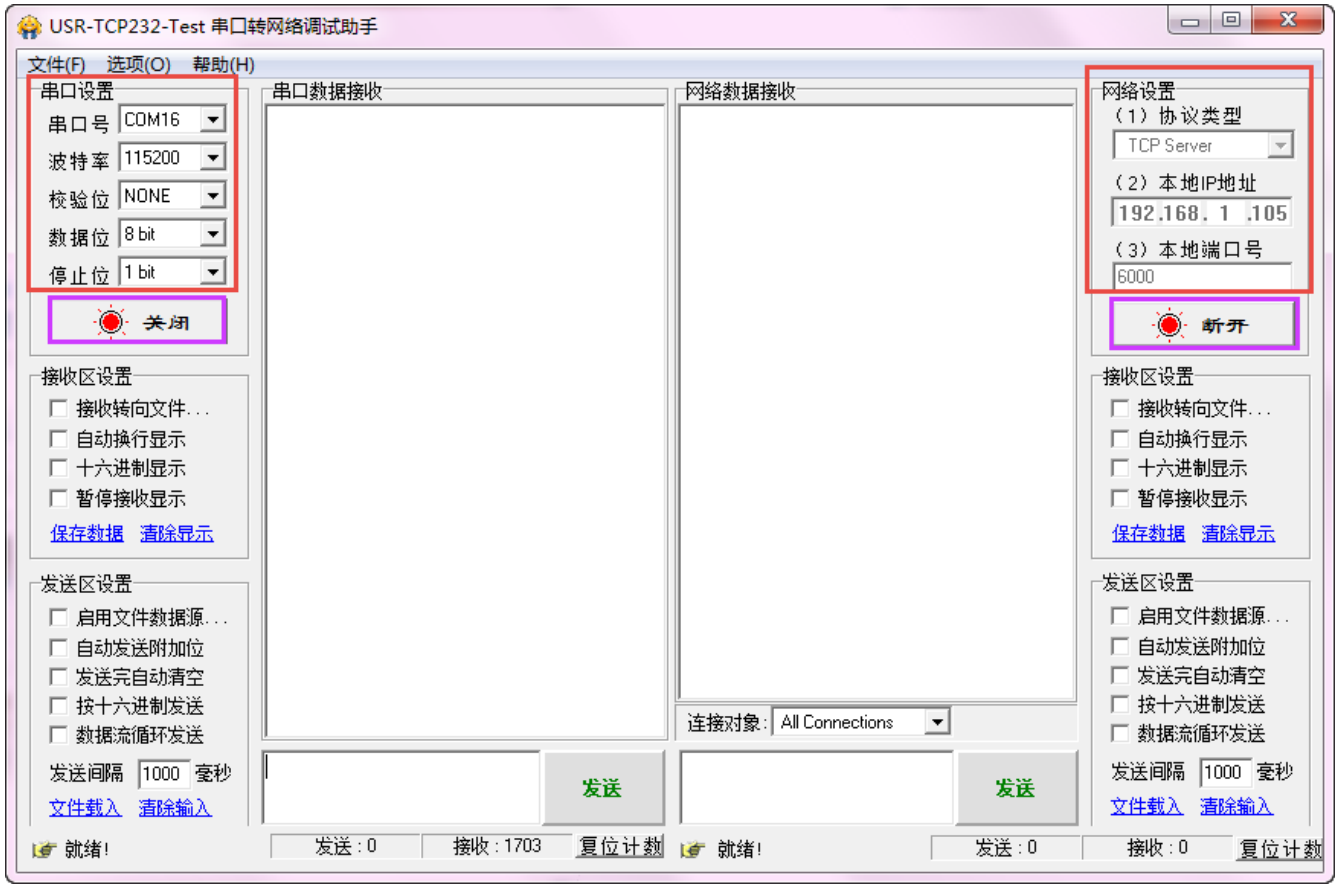


图 36_1_19 调试助手设置界面

在系统硬件初始化时串口调试助手会打印相关提示信息，等待初始化完成后可打开电脑端 CMD 窗口，输入 ping 命令测试开发板链路，图 *36_1_20* 为链路正常情况，如果出现 ping 不同情况，检查网线连接。

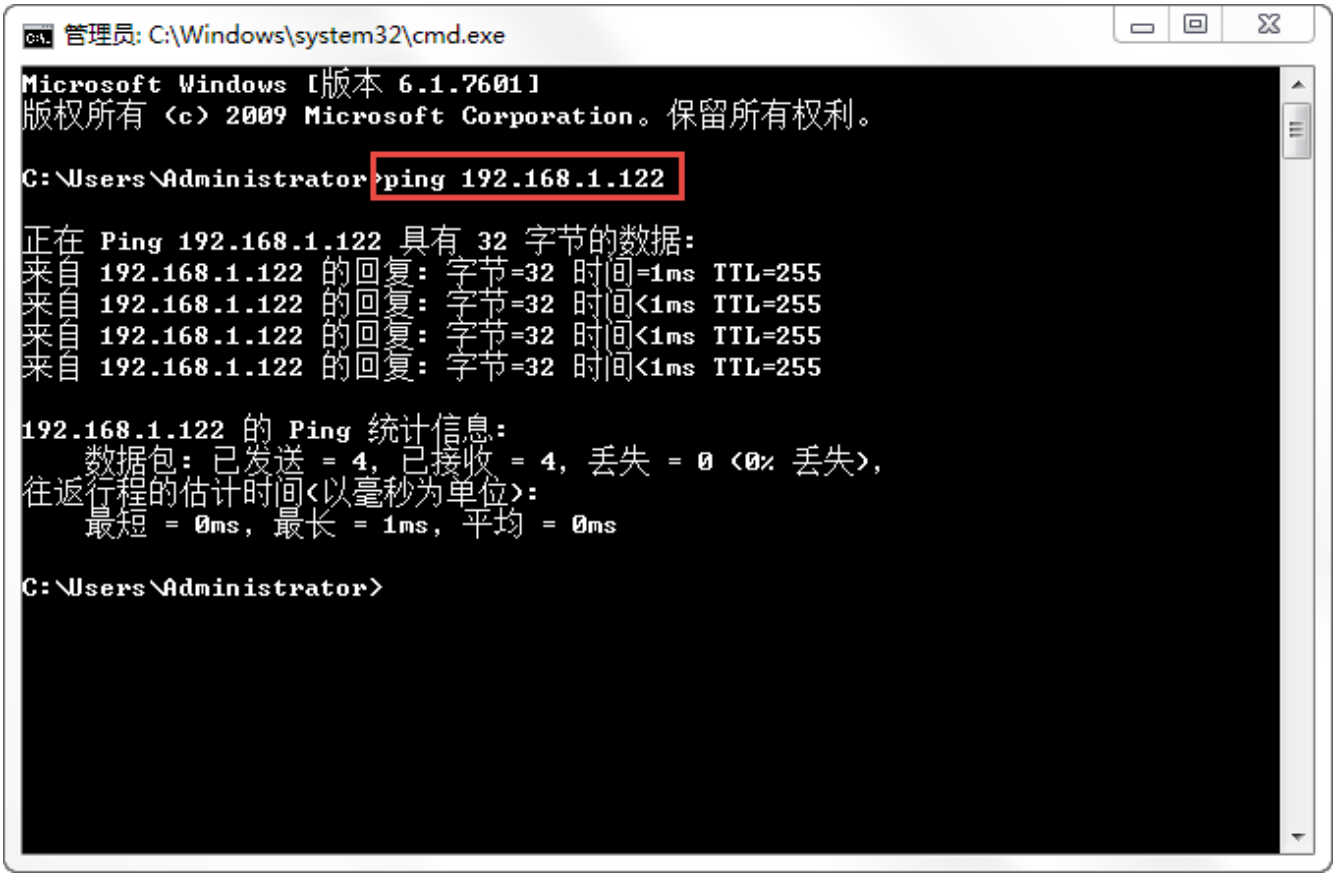


图 36_1_20 ping 窗口

ping 状态正常后，可按下开发板 KEY1 按键，使能开发板连接电脑端的 TCP 服务器，之后就可以进行数据传输，需要接收传输时可以按下开发板 KEY2 按键，实际操作调试助手界面见图 *36_1_21*。

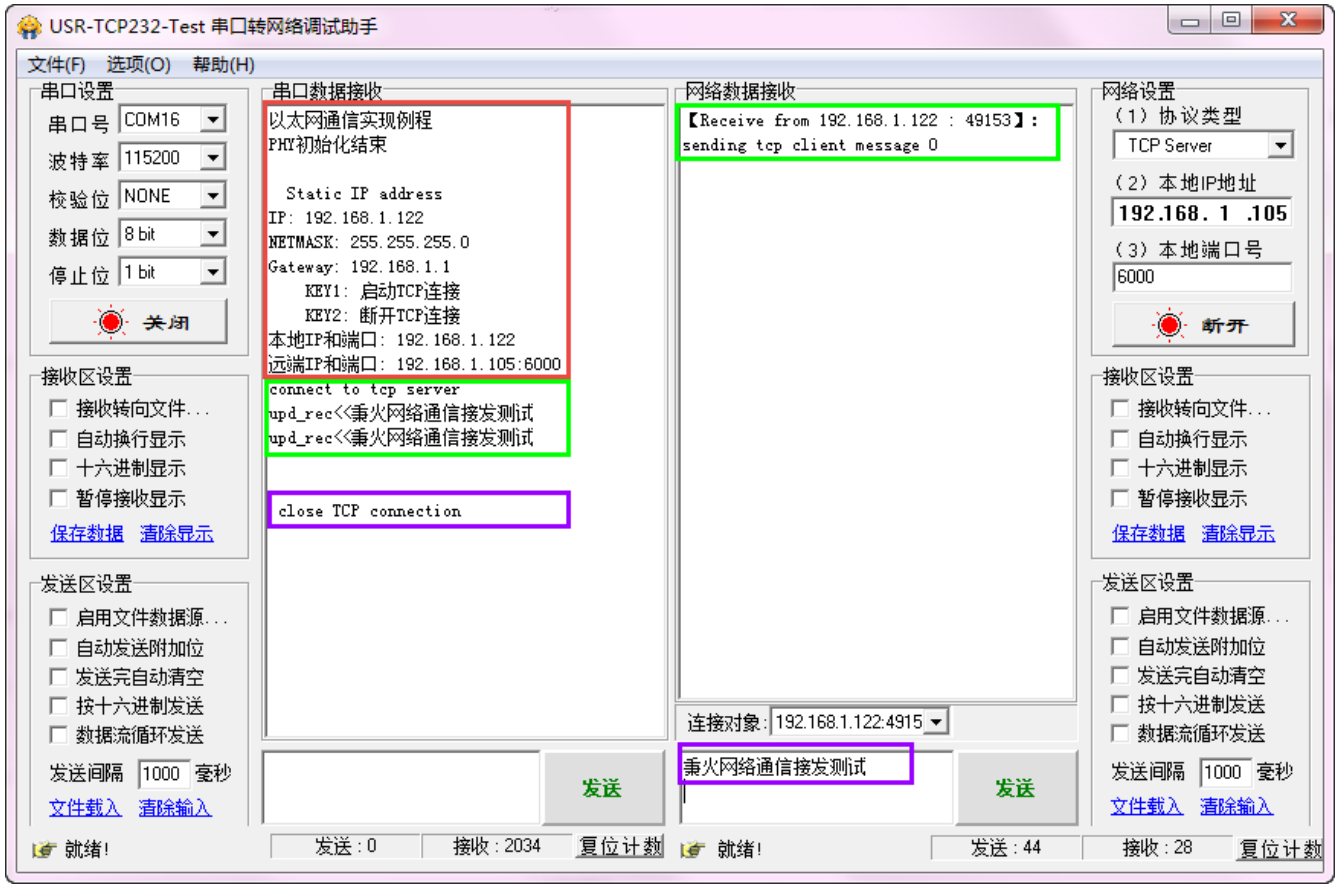


图 36_1_21 调试助手接发通信效果

39.9 基于 uCOS-III 移植 LwIP 实验

上面的实验是无操作系统移植 LwIP，LwIP 也确实是支持无操作系统移植运行，这对于芯片资源紧张、不合适运行操作系统环境还是有很大用处的。不过在很多应用中会采用操作系统上运行 LwIP，这有利于提高整体性能。这个实验我们主要讲解移植操作步骤，过程中直接使用上个实验 LwIP 底层驱动，除非有需要修改地方才指出，同时这里假设已有移植好的 uCOS-III 工程可参考使用，关于 uCOS-III 移植部分可参考我们相关文档，这里主要介绍 LwIP 使用 uCOS-III 信号量、消息队列、定时器函数等等函数接口。

这个实验最终实现在 uCOS-III 操作系统基础上移植 LwIP，使能 DHCP 功能，在动态获取 IP 之后即可 ping 通。运行 uCOS-III 操作系统之后一般会使用 Netconn 或 Socket 方法使用 LwIP，关于这

两个的应用方法限于篇幅问题这里不做深入探究。

UCOS-III 和 LwIP 都是属于软件编程层次，所以硬件设计部分并不需要做更改，直接使用上个实验的硬件设计即可。

接下来开始介绍移植步骤，为简化移植步骤，我们的思路是直接使用 uCOS-III 例程，在其基础上移植 LwIP 部分。

39.9.1 第一步：文件拷贝

拷贝整个 uCOS-III 工程，修改文件夹名称为 “ETH—基于 uCOS-III 的 LwIP 移植”，作为我们这个实验工程基础，我们在此基础上添加功能。拷贝上个实验工程中的 lwip-1.4.1 整个文件夹到 USER 文件夹 (路径：…ETH—基于 uCOS-III 的 LwIP 移植 USER) 中。

LwIP 源码部分，即 src 文件夹，内容是不用修改的，只有 port 文件夹内容需要修改。在 stsw-stm32070 文件夹找到 FreeRTOS 文件夹 (路径：…\UtilitiesThird_Partylwip-1.4.1port\STM32F4x7FreeRTOS)，该文件夹内容是 LwIP 与 FreeRTOS 操作系统连接的相关接口函数，虽然我们选择使用 uCOS-III 操作系统，当还是有很多可以借鉴的地方，移植过程我们采用修改这些文件方法实现而不是完全自己新建文件，把 FreeRTOS 整个文件夹拷贝到 port 文件夹 (路径：…ETH—基于 uCOS-III 的 LwIP 移植 USERlwip-1.4.1port) 内，并改名为 UCOS305，此时 port 文件夹内有三个文件夹，分别为：arch、Standalone、UCOS305，其中 Standalone 在本实验是不被使用的。

把上个实验工程中的 App 文件夹拷贝到本实验相同位置，其中 tcp_echoclient.c 和 tcp_echoclient.h 文件不是本实验需要的，将其删除。netconf.c、netconf.h 和 lwipopts.h 三个文件是必需的，但因为如果在本实验直接使用 lwipopts.h 文件需要修改较多地方，我们先将该文件删除，然后在 stsw-stm32070 文件夹找到 httpserver_socket 文件夹。(路径：…\UtilitiesThird_Partylwip-1.4.1port\STM32F4x7FreeRTOShttpserver_socket)，在该文件夹下 inc 文件夹中的 lwipopts.h 文件是更方便我们移植的文件，我们拷贝它到 App 文件夹中。

最后，把上个实验工程中的 ETH 文件夹拷贝到本实验相同位置，这个文件夹内容都是必需的，但我们不用进行修改。

39.9.2 第二步：为工程添加文件

与上个工程相比，LwIP 部分文件只有 port 文件夹文件有所修改，其他使用与上个实验相同文件结构皆可，最终工程文件结构参考图 *36_1_22*。

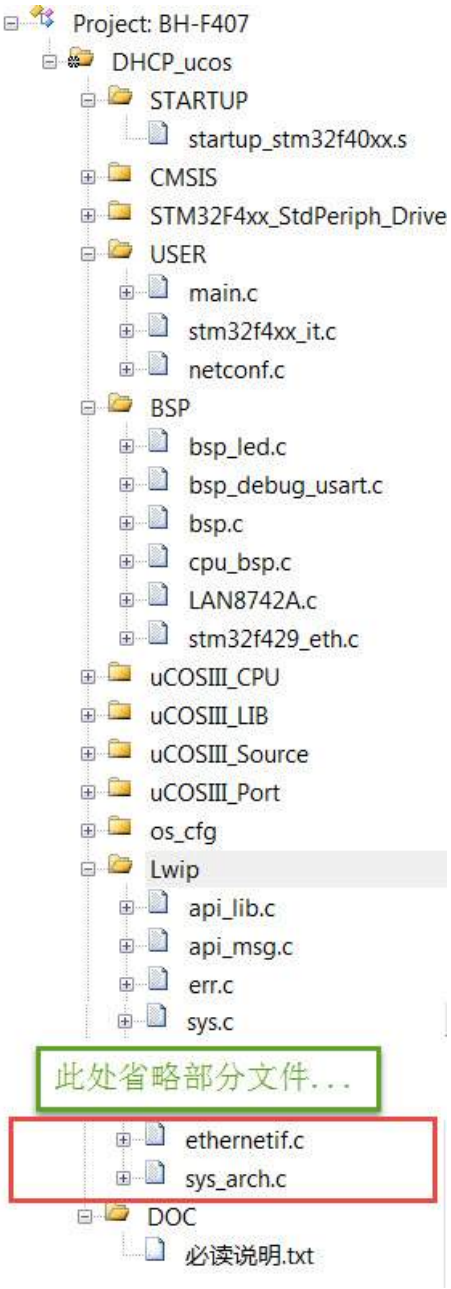


图 36_1_22 工程文件结构

添加完源文件后，还需要在工程选项中设置添加头文件路径，参考图 *36_1_23*。

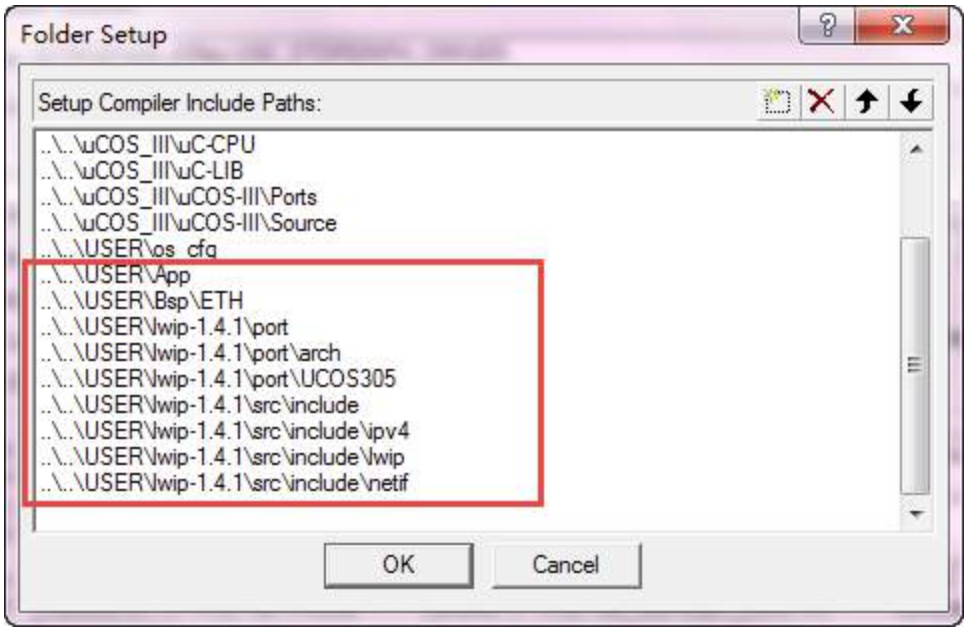


图 36_1_23 添加头文件路径

39.9.3 第三步：文件修改

ETH 文件夹内文件，stm32f4x7_eth.c、stm32f4x7_eth.h、stm32f4x7_phy.c 和 stm32f4x7_phy.h 四个文件是 ETH 外部和 PHY 相关驱动，本实验并无需修改硬件，所以这四个文件内容不用修改，stm32f4x7_eth_conf.h 文件是与 ETH 外设相关硬件宏定义，因为本实验使用操作系统，对延时函数定义与上个实验工程有所不同，需要稍作修改。

列表 14: 代码清单 36_1_14 延时函数定义

```
#ifdef USE_Delay
#include "Bsp/bsp.h"
#define _eth_delay_     Delay_10ms
#else
#define _eth_delay_
#endif
```

这里使用在 bsp.h 文件中定义的 Delay_10ms 延时函数。

sys_arch.h 和 sys_arch.c 两个文件是 LwIP 与 uCOS-III 连接的实现代码。sys_arch.h 存放相关宏定义和类型定义。

列表 15: 代码清单 36_1_15 宏定义

```
#define LWIP_STK_SIZE          512
#define LWIP_TASK_MAX          8

#define LWIP_TSK_PRIO          3
#define LWIP_TASK_START_PRIO   LWIP_TSK_PRIO
#define LWIP_TASK_END_PRIO     LWIP_TSK_PRIO +LWIP_TASK_MAX

#define MAX_QUEUES             10  // 消息邮箱的数量
#define MAX_QUEUE_ENTRIES      20  // 每个邮箱的大小

#define SYS_MBOX_NULL          (void *)0
#define SYS_SEM_NULL           (void *)0

#define sys_arch_mbox_tryfetch(mbox,msg)    sys_arch_mbox_fetch(mbox,msg,1)
```

宏 LWIP_STK_SIZE 定义 LwIP 任务栈空间大小，实际空间是 4*LWIP_STK_SIZE 个字节。宏 LWIP_TASK_MAX 定义预留给 LwIP 使用的最大任务数量。LWIP_TSK_PRIO、LWIP_TASK_START_PRIO 和 LWIP_TASK_END_PRIO 三个宏指定 LwIP 任务的优先级范围。宏 MAX_QUEUES 定义 LwIP 可以使用的最大邮箱数量，宏 MAX_QUEUE_ENTRIES 定义每个邮箱的大小。宏 SYS_MBOX_NULL 和 SYS_SEM_NULL 分别定义邮箱和信号量 NULL 对于的值。sys_arch_mbox_tryfetch 函数是尝试获取邮箱内容，这里直接调用 sys_arch_mbox_fetch 函数实现。

列表 16: 代码清单 36_1_16 类型定义

```
typedef OS_SEM      sys_sem_t; // type of semiphores
typedef OS_MUTEX    sys_mutex_t; // type of mutex
typedef OS_Q        sys_mbox_t; // type of mailboxes
```

(下页继续)

(续上页)

```
typedef CPU_INT08U sys_thread_t; // type of id of the new thread

typedef CPU_INT08U sys_prot_t;
```

不同操作系统有不同名称定义信号量、复合信号、邮箱、任务 ID 等等，这里使用 uCOS-III 操作系统需要使用对应的名称。

实际上，除了需要定于与操作系统对应的名称之外，还需要定于与编译器相关的名称，在 sys_arch.h 文件中有引用了 cc.h 头文件，因为我们使用 Windows 操作系统的 Keil 开发工具，需要对 cc.h 文件进行必须修改。

列表 17: 代码清单 36_1_17 编译器相关类型定于和宏定义

```
typedef u32_t mem_ptr_t;
//typedef int sys_prot_t;

//#define U16_F "hu"
//#define S16_F "d"
//#define X16_F "hx"
//#define U32_F "u"
//#define S32_F "d"
//#define X32_F "x"
//#define SZT_F "uz"

#define U16_F "4d"
#define S16_F "4d"
#define X16_F "4x"
#define U32_F "8ld"
#define S32_F "8ld"
#define X32_F "8lx"
```

sys_prot_t 类型已在 sys_arch.h 文件中定于，在 cc.h 文件必须注释掉不被使用。U16_F、S16_F、

X16_F 等等一系列名称用于 LwIP 的调试函数，这一系列宏定于用于调试信息输出格式化。

列表 18: 代码清单 36_1_18 调试信息输出定于

```
#define LWIP_PLATFORM_DIAG(x)  {printf x;}

#define LWIP_PLATFORM_ASSERT(x) do { printf("Assertion \"%s\" failed at  \
   line %d in %s\n",x, __LINE__, __FILE__);} while(0)

#define LWIP_ERROR(message, expression, handler) do { if (!(expression)) { \
printf("Assertion \"%s\" failed at line %d in %s\n", message, \
__LINE__, __FILE__); fflush(NULL);handler;} } while(0)
```

LwIP 实现代码已经添加了调试信息功能，我们只需要定于信息输出途径即可，这里直接使用 printf 函数，将调试信息打印到串口调试助手。

sys_arch.c 文件存放 uCOS-III 与 LwIP 连接函数，LwIP 为实现在操作系统上运行，预留了相关接口函数，不同操作系统使用不同方法实现要求的功能。该文件存放在 UCOS305 文件夹内。

列表 19: 代码清单 36_1_19 sys_now 函数

```
u32_t sys_now()
{
   OS_TICK os_tick_ctr;
   CPU_SR_ALLOC();

   CPU_CRITICAL_ENTER();
   os_tick_ctr = OSTickCtr;
   CPU_CRITICAL_EXIT();

   return os_tick_ctr;
}
```

sys_now 函数用于为 LwIP 提供系统时钟，这里直接的读取 OSTickCtr 变量值。CPU_CRITICAL_ENTER 和 CPU_CRITICAL_EXIT 分别是关闭总中断和开启总中断。

LwIP 的邮箱用于缓存和传递数据包。

列表 20: 代码清单 36_1_20 邮箱创建与删除

```
err_t sys_mbox_new(sys_mbox_t *mbox, int size)
{
   OS_ERR       ucErr;

   OSQCreate(mbox,"LWIP quiue", size, &ucErr);
   LWIP_ASSERT( "OSQCreate ", ucErr == OS_ERR_NONE );

   if ( ucErr == OS_ERR_NONE) {
      return 0;
   }
   return -1;
}

void sys_mbox_free(sys_mbox_t *mbox)
{
   OS_ERR     ucErr;
   LWIP_ASSERT( "sys_mbox_free ", mbox != SYS_MBOX_NULL );

   OSQFlush(mbox,& ucErr);

   OSQDel(mbox, OS_OPT_DEL_ALWAYS, &ucErr);
   LWIP_ASSERT( "OSQDel ", ucErr == OS_ERR_NONE );
}
```

sys_mbox_new 函数要求实现的功能是创建一个邮箱，这里使用 OSQCreate 函数创建一个队列。sys_mbox_free 函数要求实现的功能是释放一个邮箱，如果邮箱存在内容，会发生错误，这里先使用 OSQFlush 函数清除队列内容，然后再使用 OSQDel 函数删除队列。LWIP_ASSERT 函数是由 LwIP 定义的断言，用于调试错误。

列表 21: 代码清单 36_1_21 邮箱发送和获取

```
void sys_mbox_post(sys_mbox_t *mbox, void *data)
{
   OS_ERR      ucErr;
   CPU_INT08U  i=0;
   if ( data == NULL ) data = (void*)&pvNullPointer;
   /* try 10 times */
   while (i<10) {
         OSQPost(mbox, data,0,OS_OPT_POST_ALL,&ucErr);
         if (ucErr == OS_ERR_NONE)
            break;
         i++;
         OSTimeDly(5,OS_OPT_TIME_DLY,&ucErr);
   }
   LWIP_ASSERT( "sys_mbox_post error!\n", i !=10 );
}

err_t sys_mbox_trypost(sys_mbox_t *mbox, void *msg)
{
   OS_ERR      ucErr;
   if (msg == NULL ) msg = (void*)&pvNullPointer;
   OSQPost(mbox, msg,0,OS_OPT_POST_ALL,&ucErr);
   if (ucErr != OS_ERR_NONE) {
         return ERR_MEM;
   }
   return ERR_OK;
}

u32_t sys_arch_mbox_fetch(sys_mbox_t *mbox, void **msg, u32_t timeout)
{
   OS_ERR  ucErr;
```

(下页继续)

(续上页)

```
    OS_MSG_SIZE    msg_size;
    CPU_TS         ucos_timeout;
    CPU_TS         in_timeout = timeout/LWIP_ARCH_TICK_PER_MS;
    if (timeout && in_timeout == 0)
          in_timeout = 1;
    *msg  = OSQPend (mbox,in_timeout,OS_OPT_PEND_BLOCKING,&msg_size,
                     &ucos_timeout,&ucErr);

    if ( ucErr == OS_ERR_TIMEOUT )
          ucos_timeout = SYS_ARCH_TIMEOUT;
    return ucos_timeout;
}
```

sys_mbox_post 函数要求实现的功能是发送一个邮箱，这里主要调用 OSQPost 函数实现队列发送，为保证发送成功，最多尝试 10 次队列发送。sys_mbox_trypost 函数是尝试发送一个邮箱，这里我们直接使用 OSQPost 函数发送一次信号量，而不像 sys_mbox_post 函数在发送失败时可能尝试发送多次。sys_arch_mbox_fetch 函数用于获取邮箱内容，并指定等待超时时间，这里主要通过调用 OSQPend 函数实现队列获取。

列表 22: 代码清单 36_1_22 邮箱可用性检查和不可用设置

```
int sys_mbox_valid(sys_mbox_t *mbox)
{
    if (mbox->NamePtr)
          return (strcmp(mbox->NamePtr,"?Q"))? 1:0;
    else
          return 0;
}

void sys_mbox_set_invalid(sys_mbox_t *mbox)
{
```

(下页继续)

(续上页)

```
    if (sys_mbox_valid(mbox))
        sys_mbox_free(mbox);
}
```

sys_mbox_valid 函数要求实现的功能是检查指定的邮箱是否可用，对于 uCOSIII，直接调用 strcmp 函数检查队列名称是否存在“Q”字段，如果存在说明该邮箱可用，否则不可用。sys_mbox_set_invalid 函数要求实现的功能是将指定的邮箱设置为不可用 (无效)，这里先调用 sys_mbox_valid 函数判断邮箱是可用的，如果本身不可用就无需操作，确定邮箱可用后调用 sys_mbox_free 函数删除邮箱。

LwIP 的信号量用于进程间的通信。

列表 23: 代码清单 36_1_23 新建信号量

```
err_t sys_sem_new(sys_sem_t *sem, u8_t count)
{
    OS_ERR  ucErr;
    OSSemCreate (sem,"LWIP Sem",count,&ucErr);
    if (ucErr != OS_ERR_NONE ) {
        LWIP_ASSERT("OSSemCreate ",ucErr == OS_ERR_NONE );
        return -1;
    }
    return 0;
}
```

sys_sem_new 函数要求实现的功能是新建一个信号量，这里直接调用 OSSemCreate 函数新建一个信号量，count 参数用于指定信号量初始值。

列表 24: 代码清单 36_1_24 信号量相关函数

```
u32_t sys_arch_sem_wait(sys_sem_t *sem, u32_t timeout)
{
    OS_ERR  ucErr;
```

(下页继续)

(续上页)

```
    CPU_TS         ucos_timeout;
    CPU_TS         in_timeout = timeout/LWIP_ARCH_TICK_PER_MS;
    if (timeout && in_timeout == 0)
        in_timeout = 1;
    OSSemPend (sem,in_timeout,OS_OPT_PEND_BLOCKING,&ucos_timeout,&ucErr);
    /*  only when timeout! */
    if (ucErr == OS_ERR_TIMEOUT)
        ucos_timeout = SYS_ARCH_TIMEOUT;
    return ucos_timeout;
}

void sys_sem_signal(sys_sem_t *sem)
{
    OS_ERR  ucErr;
    OSSemPost(sem,OS_OPT_POST_ALL,&ucErr);
    LWIP_ASSERT("OSSemPost ",ucErr == OS_ERR_NONE );
}

void sys_sem_free(sys_sem_t *sem)
{
    OS_ERR     ucErr;
    OSSemDel(sem, OS_OPT_DEL_ALWAYS, &ucErr );
    LWIP_ASSERT( "OSSemDel ", ucErr == OS_ERR_NONE );
}

int sys_sem_valid(sys_sem_t *sem)
{
    if (sem->NamePtr)
        return (strcmp(sem->NamePtr,"?SEM"))? 1:0;
    else
        return 0;
}
```

(下页继续)

(续上页)

```
void sys_sem_set_invalid(sys_sem_t *sem)
{
   if (sys_sem_valid(sem))
      sys_sem_free(sem);
}
```

sys_arch_sem_wait 函数要求实现的功能是等待获取一个信号量，并具有超时等待检查功能，这里直接调用 OSSemPend 函数实现信号量获取。sys_sem_signal 函数要求实现的功能是发送一个信号量，这里直接调用 OSSemPost 函数发送一个信号量。sys_sem_free 函数要求实现的功能是释放一个信号量，这里直接调用 OSSemDel 函数删除信号量。sys_sem_valid 函数要求实现的功能是检查指定的信号量是否可用，这里调用 strcmp 函数判断信号量名称中是否存在 “SEM” 字段，如果存在说明是信号量，否则不是信号量。sys_sem_set_invalid 函数要求实现的功能是使指定的信号量不可用，这里先调用 sys_sem_valid 确保信号量可用，再调用 sys_sem_free 函数释放该信号量。

列表 25: 代码清单 36_1_25 系统初始化

```
void sys_init(void)
{
   OS_ERR ucErr;
   memset(LwIP_task_priority_stask,0,sizeof(LwIP_task_priority_stask));
   /* init mem used by sys_mbox_t, use ucosIII functions */
   OSMemCreate(&StackMem,"LWIP TASK STK",(void*)LwIP_Task_Stk,
               LWIP_TASK_MAX,LWIP_STK_SIZE*sizeof(CPU_STK),&ucErr);
   LWIP_ASSERT( "sys_init: failed OSMemCreate STK", ucErr == OS_ERR_NONE );
}
```

sys_init 函数在系统启动时被调用，可以用于初始化工作环境，这里先调用 memset 函数将 LwIP_task_priority_stask 数组内容清空，该数值用于存放任务优先级，接下来调用 OSMemCreate 函数初始化申请内存空间，用于 LwIP 任务栈空间。

列表 26: 代码清单 36_1_26 任务创建

```
sys_thread_t sys_thread_new(const char *name, lwip_thread_fn thread ,
                                  void *arg, int stacksize, int prio)
{
   CPU_INT08U  ubPrio = LWIP_TASK_START_PRIO;
   OS_ERR      ucErr;
   int i;
   int tsk_prio;
   CPU_STK * task_stk;
   if (prio) {
      ubPrio +=(prio-1);
      for (i=0; i<LWIP_TASK_MAX; ++i)
            if (LwIP_task_priority_stask[i] == ubPrio)
               break;
      if (i == LWIP_TASK_MAX) {
            for (i=0; i<LWIP_TASK_MAX; ++i)
               if (LwIP_task_priority_stask[i]==0) {
                  LwIP_task_priority_stask[i] = ubPrio;
                  break;
               }
            if (i == LWIP_TASK_MAX) {
LWIP_ASSERT("sys_thread_new: there is no space for priority",0);
               return (-1);
            }
      } else
            prio = 0;
   }
   /* Search for a suitable priority */
   if (!prio) {
      ubPrio = LWIP_TASK_START_PRIO;
      while (ubPrio < (LWIP_TASK_START_PRIO+LWIP_TASK_MAX)) {
```

(下页继续)

(续上页)

```
            for (i=0; i<LWIP_TASK_MAX; ++i)
               if (LwIP_task_priority_stask[i] == ubPrio) {
                  ++ubPrio;
                  break;
               }
            if (i == LWIP_TASK_MAX)
               break;
      }
      if (ubPrio < (LWIP_TASK_START_PRIO+LWIP_TASK_MAX))
            for (i=0; i<LWIP_TASK_MAX; ++i)
               if (LwIP_task_priority_stask[i]==0) {
                  LwIP_task_priority_stask[i] = ubPrio;
                  break;
               }
      if(ubPrio>=(LWIP_TASK_START_PRIO+LWIP_TASK_MAX)||i==LWIP_TASK_MAX){
         LWIP_ASSERT( "sys_thread_new: there is no free priority", 0 );
            return (-1);
      }
   }
   if (stacksize > LWIP_STK_SIZE || !stacksize)
      stacksize = LWIP_STK_SIZE;
   /* get Stack from pool */
   task_stk = OSMemGet( &StackMem, &ucErr );
   if (ucErr != OS_ERR_NONE) {
      LWIP_ASSERT( "sys_thread_new: impossible to get a stack", 0 );
      return (-1);
   }
   tsk_prio = ubPrio-LWIP_TASK_START_PRIO;
   OSTaskCreate(&LwIP_task_TCB[tsk_prio],
                (CPU_CHAR   *)name,
                (OS_TASK_PTR)thread,
                (void       *)0,
```

(下页继续)

(续上页)

```
                (OS_PRIO     )ubPrio,
                (CPU_STK    *)&task_stk[0],
                (CPU_STK_SIZE)stacksize/10,
                (CPU_STK_SIZE)stacksize,
                (OS_MSG_QTY )0,
                (OS_TICK     )0,
                (void       *)0,
             (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                (OS_ERR     *)&ucErr);

   return ubPrio;
}
```

sys_thread_new 函数要求实现的功能是新建一个任务，函数有五个形参，分别指定任务名称、任何函数、任务自定义参数、任务栈空间大小、任务优先级。对于 LwIP，系统有限制其最多可用任务数，对于优先级也指定一定的范围，sys_thread_new 函数先对优先级参数进行处理，获取合适的优先级。OSMemGet 函数用于从分配给 LwIP 任务栈使用的内存空间中申请一块空间用于本任务。OSTaskCreate 函数用于创建一个任务。

列表 27: 代码清单 36_1_27 临界区域保护

```
sys_prot_t sys_arch_protect(void)
{
   CPU_SR_ALLOC();

   CPU_CRITICAL_ENTER();
   return 1;
}

void sys_arch_unprotect(sys_prot_t pval)
{
   CPU_SR_ALLOC();
```

(下页继续)

(续上页)

```
    LWIP_UNUSED_ARG(pval);
    CPU_CRITICAL_EXIT();
}
```

sys_arch_protecth 函数要求实现的功能是完成临界区域保护并保存当前内容，这里调用 CPU_CRITICAL_ENTER 函数进入临界区域保护。sys_arch_unprotect 函数要求实现的功能是恢复受保护区域的先前状态，与 sys_arch_protecth 函数配套使用，这里直接调用 CPU_CRITICAL_EXIT 函数完成退出临界区域保护。

ethernetif.c 文件存放 LwIP 与 ETH 外设连接函数 (网络接口函数)，属于最底层驱动函数，与上个实验无操作系统移植的文件内容有所不同，这里在函数内部会调用相关系统操作函数。

列表 28: 代码清单 36_1_28 low_level_init 函数

```
static void low_level_init(struct netif *netif)
{
    uint32_t i;

    /* set netif MAC hardware address length */
    netif->hwaddr_len = ETHARP_HWADDR_LEN;
    /* set netif MAC hardware address */
    netif->hwaddr[0] =  MAC_ADDR0;
    netif->hwaddr[1] =  MAC_ADDR1;
    netif->hwaddr[2] =  MAC_ADDR2;
    netif->hwaddr[3] =  MAC_ADDR3;
    netif->hwaddr[4] =  MAC_ADDR4;
    netif->hwaddr[5] =  MAC_ADDR5;
    /* set netif maximum transfer unit */
    netif->mtu = 1500;
    /* Accept broadcast address and ARP traffic */
    netif->flags=NETIF_FLAG_BROADCAST|NETIF_FLAG_ETHARP|NETIF_FLAG_LINK_UP;
```

(下页继续)

(续上页)

```
    s_pxNetIf =netif;

    /* initialize MAC address in ethernet MAC */
    ETH_MACAddressConfig(ETH_MAC_Address0, netif->hwaddr);
    /* Initialize Tx Descriptors list: Chain Mode */
    ETH_DMATxDescChainInit(DMATxDscrTab, &Tx_Buff[0][0], ETH_TXBUFNB);
    /* Initialize Rx Descriptors list: Chain Mode  */
    ETH_DMARxDescChainInit(DMARxDscrTab, &Rx_Buff[0][0], ETH_RXBUFNB);

    /* Enable Ethernet Rx interrrupt */
    for (i=0; i<ETH_RXBUFNB; i++) {
        ETH_DMARxDescReceiveITConfig(&DMARxDscrTab[i], ENABLE);
    }

#ifdef CHECKSUM_BY_HARDWARE
    /* Enable the checksum insertion for the Tx frames */
    {
        for (i=0; i<ETH_TXBUFNB; i++) {
              ETH_DMATxDescChecksumInsertionConfig(&DMATxDscrTab[i],
                          ETH_DMATxDesc_ChecksumTCPUDPICMPFull);
        }
    }
#endif

    /* create the task that handles the ETH_MAC */
    sys_thread_new((const char*)"Eth_if",ethernetif_input,netif,
        netifINTERFACE_TASK_STACK_SIZE,netifINTERFACE_TASK_PRIORITY);
    /* Enable MAC and DMA transmission and reception */
    ETH_Start();
}
```

low_level_init 函数是网络接口初始化函数，在 ethernetif_init 函数被调用，在系统启动时被

运行一次，用于初始化与网络接口相关硬件。函数先是给 netif 结构体成员赋值配置网卡参数，调用 ETH_MACAddressConfig 函数绑定网卡 MAC 地址，ETH_DMATxDescChainInit 和 ETH_DMARxDescChainInit 初始化网络数据帧发送和接收描述符，设置为链模式。调用 ETH_DMARxDescReceiveITConfig 函数使能 DMA 数据接收相关中断。通过定义宏 CHECKSUM_BY_HARDWARE，可以使能发送数据硬件校验和，这个需要硬件支持，STM32F4xx 控制器是支持的。调用 sys_thread_new 函数创建一个任务，设置任务函数是 ethernetif_input，该函数用于讲接收到数据包转入到 LwIP 内部缓存区，这里还传递了 netif 结构体变量。最后，调用 ETH_Start 函数使能 ETH。

low_level_output 和 low_level_input 两个函数内容与上个实验工程同名函数几乎相同，这里不再讲解。

列表 29: 代码清单 36_1_29 ethernetif_input 函数

```
void ethernetif_input( void * pvParameters )
{
   struct pbuf *p;
   OS_ERR os_err;
   err_t err;

   /* move received packet into a new pbuf */
   while (1) {
      SYS_ARCH_DECL_PROTECT(sr);

      SYS_ARCH_PROTECT(sr);
      p = low_level_input(s_pxNetIf);
      SYS_ARCH_UNPROTECT(sr);
      if (p == NULL)
            continue;
      err = s_pxNetIf->input(p, s_pxNetIf);
      if (err != ERR_OK) {
   LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n"));
            pbuf_free(p);
```

(下页继续)

(续上页)

```
            p = NULL;
        }
        /*sleep 5 ms*/
        OSTimeDlyHMSM(0, 0, 0, 5, OS_OPT_TIME_DLY, (OS_ERR *)&os_err);
    }
}
```

ethernetif_input 函数作为 sys_thread_new 指定的任务函数，用于接收网络数据包并将其转入到 LwIP 内核。SYS_ARCH_DECL_PROTECT、SYS_ARCH_PROTECT 和 SYS_ARCH_UNPROTECT 三个函数是与临界区域保护相关代码，可用在 lwipopts.h 文件中的 SYS_LIGHTWEIGHT_PROT 配置相关功能。调用 low_level_input 函数获取接收到的数据包，如果判断没有接收到数据包则不执行本来循环后面内容。在判断接收到数据包后，将数据包内容转入 LwIP 内核。

相对与上个实验工程，那个时候我们需要在 main 函数中的无限循环中调用数据包接收查询，现在在这里我们创建一个这个数据包接收任务，让它执行数据包接收并将数据转入到 LwIP 内核。

列表 30: 代码清单 36_1_30 ethernetif_init 函数

```
err_t ethernetif_init(struct netif *netif)
{
    LWIP_ASSERT("netif != NULL", (netif != NULL));

#if LWIP_NETIF_HOSTNAME
    /* Initialize interface hostname */
    netif->hostname = "lwip";
#endif /* LWIP_NETIF_HOSTNAME */

    netif->name[0] = IFNAME0;
    netif->name[1] = IFNAME1;

    netif->output = etharp_output;
    netif->linkoutput = low_level_output;
```

(下页继续)

(续上页)

```
    /* initialize the hardware */
    low_level_init(netif);

    etharp_init();
    sys_timeout(ARP_TMR_INTERVAL, arp_timer, NULL);

    return ERR_OK;
}
```

ethernetif_init 函数用于初始化网卡，在系统启动时必须被运行一次，该函数在 LwIP_Init 函数中被调用。函数首先为 netif 结构体成员赋值，然后调用 low_level_init 函数完成 ETH 外设初始化，etharp_init 函数完成 ARP 协议初始化，sys_timeout 函数启动 ARP 超时并注册一个 ARP 超时回调函数。

netconf.c 和 netconf.h 用于存放 LwIP 配置相关代码。netconf.h 是相关的宏定义，具体参考*代码清单 36_1_8*，不过本实验定义了 USE_DHCP 宏，使能 DHCP 功能。不同于上个实验，现在 netconf.c 文件只要求实现两个函数，LwIP_Init 和 LwIP_DHCP_task 函数，把其他函数删除。其中 LwIP_Init 函数与上个实验工程使用相同配置即可，参考*代码清单 36_1_9*。

列表 31: 代码清单 36_1_31 LwIP_DHCP_task 函数

```
#ifdef USE_DHCP
void LwIP_DHCP_task(void * pvParameters)
{
    struct ip_addr ipaddr;
    struct ip_addr netmask;
    struct ip_addr gw;
    OS_ERR  os_err;

    while (1) {
```

(下页继续)

(续上页)

```
        switch (DHCP_state) {
                /*****************************************************/
                /*    与上个实验工程代码相同，参考代码清单 43 12            */
                /*****************************************************/
        }
        OSTimeDlyHMSM( 0u, 0u, 0u, 250u,OS_OPT_TIME_HMSM_STRICT,&os_err);
    }
}
#endif
```

在 netconf.h 文件中定义了 USE_DHCP 宏，即开启了 DHCP 功能，LwIP_DHCP_task 函数才有效。在上个实验工程中，我们使用 LwIP_DHCP_Process_Handle 函数 (参考代码清单 39-12) 完成 DHCP 功能实现，该函数是被周期调用执行的。现在，既然我们使用了操作系统，就可以直接创建一个任务执行 DHCP 功能，LwIP_DHCP_task 函数就是 DHCP 任务函数，函数需要实现的内容与代码清单 43-12 相同，在函数最后调用 OSTimeDlyHMSM 函数延时 250ms。

lwipopts.h 文件存放一些宏定义，用于剪切 LwIP 功能，该文件拷贝自 ST 官方带操作系统的工程文件，方便我们移植，该文件同时使能了 Netconn 和 Socket 编程支持。这里我们还需要对该文件一个宏定义进行修改，直接把 TCPIP_THREAD_PRIO 宏定义为 6，该宏定义了 TCPIP 任务的优先级。

至此，有关 LwIP 函数文件修改已经全部完成，接下来还需要实现就是调用相关初始化函数完成 LwIP 初始化，然后就可以直接使用 LwIP 函数完成用户任务。

首先是 ETH 外设硬件相关初始化函数调用，bsp.c 文件中 BSP_Init 函数用于放置系统启动时各模块硬件初始化函数。

列表 32: 代码清单 36_1_32 BSP_Init 函数

```
void  BSP_Init (void)
{
   /* 设置 NVIC 优先级分组为 Group2: 0-3 抢占式优先级，0-3 的响应式优先级 */
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
```

(下页继续)

(续上页)

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

    Key_GPIO_Config();
    /* 初始化调试串口，一般为串口 1 */
    Debug_USART_Config();

    printf("基于 uCOS-III 的 LwIP 网络通信测试\n");

    BSP_Tick_Init();

    /* Configure ethernet (GPIOs, clocks, MAC, DMA) */
    ETH_BSP_Config();
    printf("LAN8720A BSP INIT AND COMFIGURE SUCCESS\n");
}
```

在 BSP_Init 函数最后调用 ETH_BSP_Config 函数完成 ETH 外设相关硬件初始化，包含了 RMII 和 SMI 相关 GPIO 初始化，ETH 外设时钟使能、MAC 和 DMA 配置并获取 PHY 的状态。

列表 33: 代码清单 36_1_33 开始任务函数

```
static  void  AppTaskStart (void *p_arg)
{
    OS_ERR      err;
    (void)p_arg;

    BSP_Init();                     /* Initialize BSP functions*/
    CPU_Init();                    /* Initialize the uC/CPU services*/

#if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err); /*Compute CPU capacity with no task␣
 ↪running*/
```

(下页继续)

(续上页)

```
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN
   CPU_IntDisMeasMaxCurReset();
#endif

#if (APP_CFG_SERIAL_EN == DEF_ENABLED)
   APP_TRACE_DBG(("Creating Application kernel objects\n\r"));
#endif
   AppObjCreate();
   /* Create Applicaiton kernel objects                    */
#if (APP_CFG_SERIAL_EN == DEF_ENABLED)
   APP_TRACE_DBG(("Creating Application Tasks\n\r"));
#endif
   AppTaskCreate();                 /* Create Application tasks*/

   /* Initilaize the LwIP stack */
   LwIP_Init();

#ifdef USE_DHCP
   /* Start DHCPClient */
   OSTaskCreate(&AppTaskDHCPTCB,"DHCP",
                LwIP_DHCP_task,
                &gnetif,
                APP_CFG_TASK_DHCP_PRIO,
                &AppTaskDHCPStk[0],
                AppTaskDHCPStk[APP_CFG_TASK_DHCP_STK_SIZE / 10u],
                APP_CFG_TASK_DHCP_STK_SIZE,
                0u,
                0u,
                0,
                (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
```

(下页继续)

(续上页)

```
                &err);
#endif  //#ifdef USE_DHCP

   while (DEF_TRUE) {
      OSTimeDlyHMSM(0u, 0u, 1u, 0u,
                    OS_OPT_TIME_HMSM_STRICT,
                    &err);
   }
}
```

AppTaskStart 函数是系统运行的启动任务函数，先执行 BSP_Init 函数完成各个模块硬件初始化，接下来几个函数用于 uCOS-III 初始化，接下来调用 LwIP_Init 函数完成 LwIP 协议栈初始化。如果使能了 DHCP 功能，就创建 DHCP 任务，指定 LwIP_DHCP_task 函数为 DHCP 任务函数。

这个实验只是简单实现 LwIP 在 uCOS-III 操作系统基础上移植，并没有过多实现应用层方面代码，最后通过开发板是否 ping 通检验。

39.9.4 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并配置好相关参数；使用网线连接开发板网口跟路由器，这里要求电脑连接在同一个路由器上，这里要求使用路由器，可以提供 DHCP 服务器功能，而电脑不行的。编译工程文件下载到开发板上。在串口调试助手可以看到相关信息，参考图 *36_1_24*，可以看到在使能 DHCP 功能之后，开发板动态获取 IP 地址为：192.168.1.124，这与我们在 netconf.h 文件中设置的静态地址是不同的 (当然，也有可能刚好相同)。

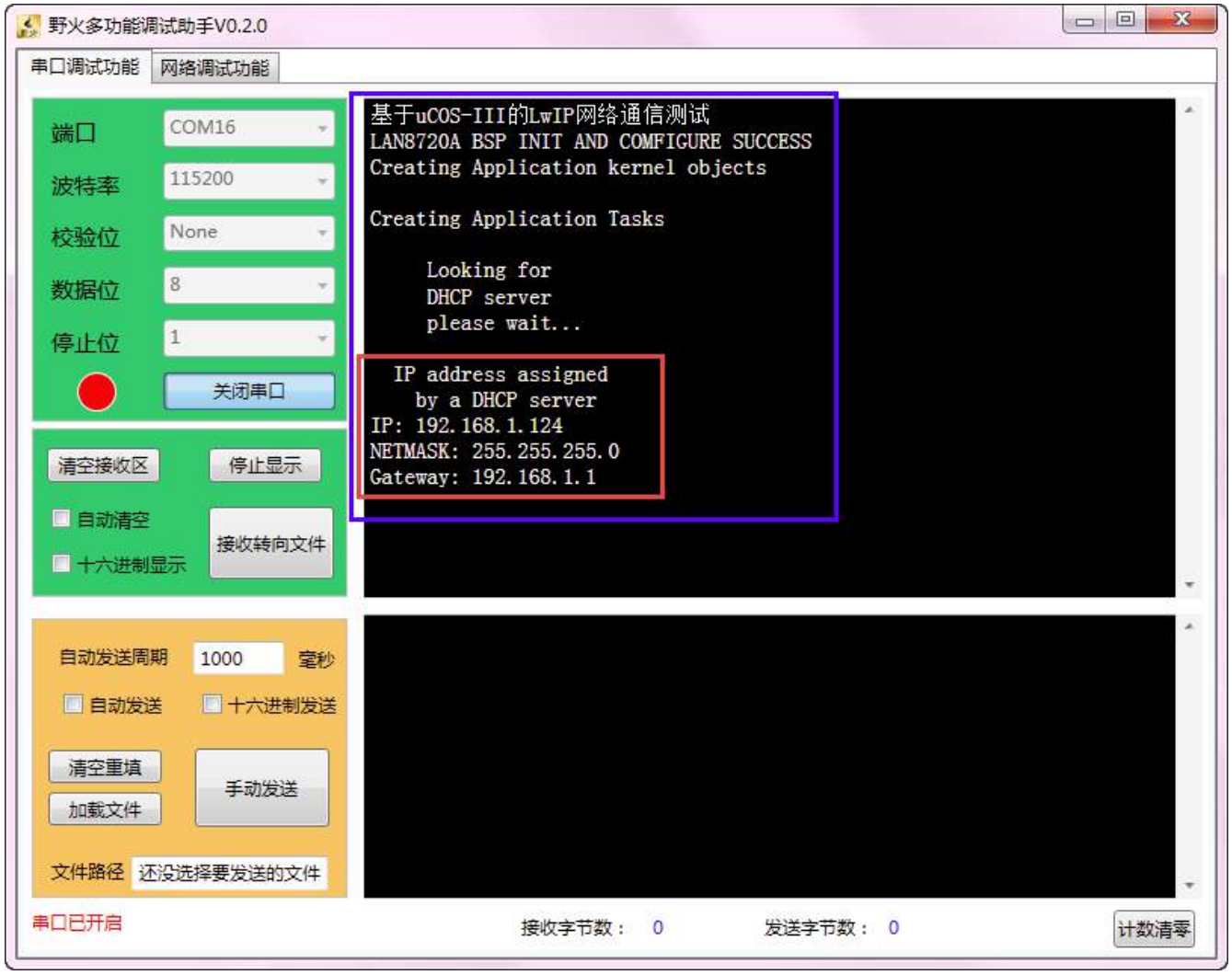


图 36_1_24 串口调试助手窗口

串口调试助手显示如图 *36_1_24* 信息是代码移植成功的最基本保证。如果没有代码没有移植成功或者网线没有接好，是无法通过 DHCP 获取动态 IP 的。保证移植成功之后，为进一步验证程序，我们可以在电脑端 ping 开发板网络。打开电脑端的 DOC 命令输入窗口，输入 “ping 192.168.1.124”，就可以测试网络链路，链路正常时 DOC 窗口截图如图 *36_1_25*。

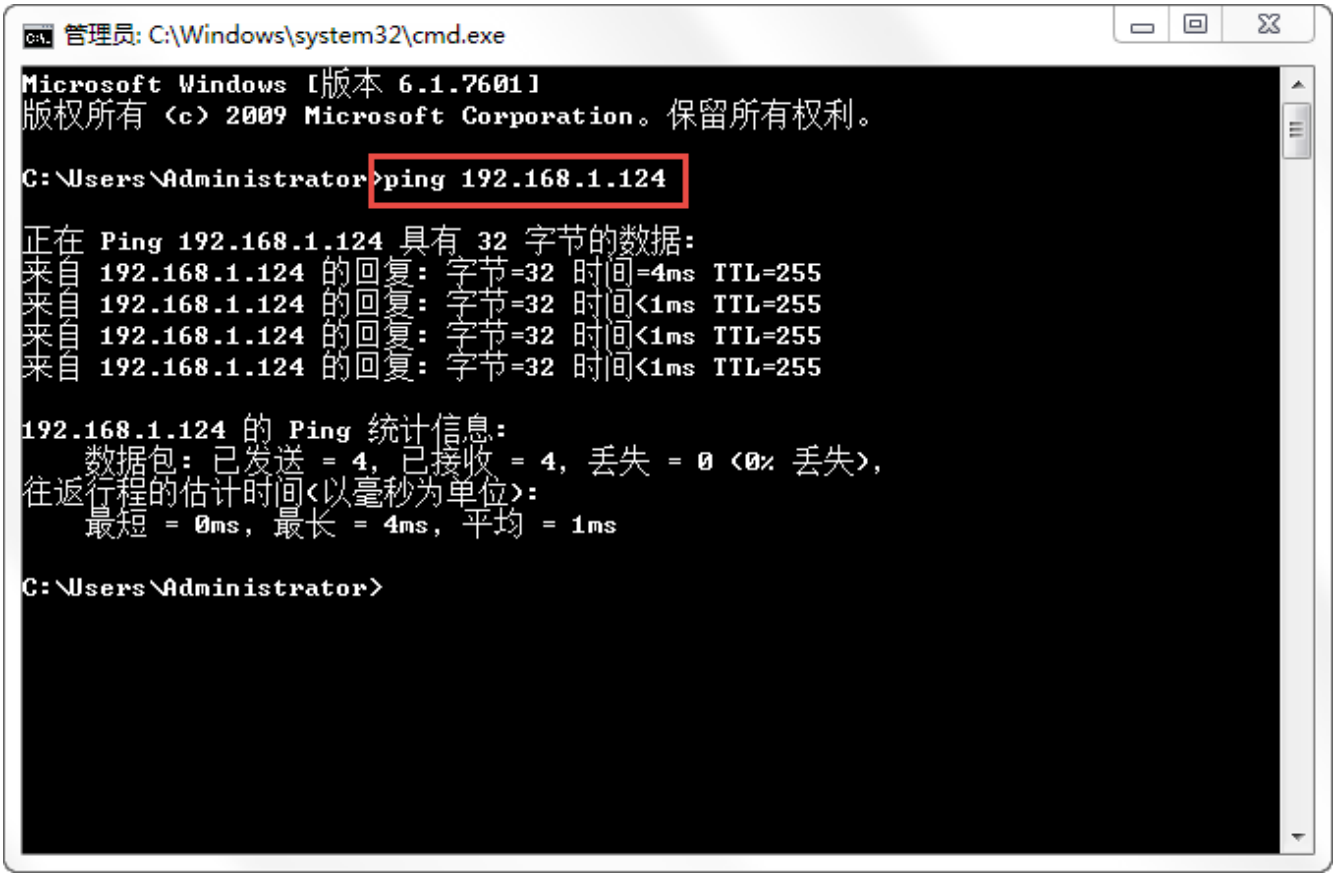


图 36_1_25 ping 命令窗口

第 40 章　CAN—通讯实验

本章参考资料：《STM32F4xx 参考手册 2》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

若对 CAN 通讯协议不了解，可先阅读《CAN 总线入门》、《CAN-bus 规范》文档内容学习。

关于实验板上的 CAN 收发器可查阅《TJA1050》文档了解。

40.1 CAN 协议简介

CAN 是控制器局域网络 (Controller Area Network) 的简称，它是由研发和生产汽车电子产品著称的德国 BOSCH 公司开发的，并最终成为国际标准（ISO11519），是国际上应用最广泛的现场总线之一。

CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。近年来，它具有的高可靠性和良好的错误检测能力受到重视，被广泛应用于汽车计算机控制系统和环境温度恶劣、电磁辐射强及振动大的工业环境。

40.1.1 CAN 物理层

与 I2C、SPI 等具有时钟信号的同步通讯方式不同，CAN 通讯并不是以时钟信号来进行同步的，它是一种异步通讯，只具有 CAN_High 和 CAN_Low 两条信号线，共同构成一组差分信号线，以差分信号的形式进行通讯。

40.1.1.1 闭环总线网络

CAN 物理层的形式主要有两种，图 *39_0_1* 中的 CAN 通讯网络是一种遵循 ISO11898 标准的高速、短距离“闭环网络”，它的总线最大长度为 40m，通信速度最高为 1Mbps，总线的两端各要求有一个“120 欧”的电阻。

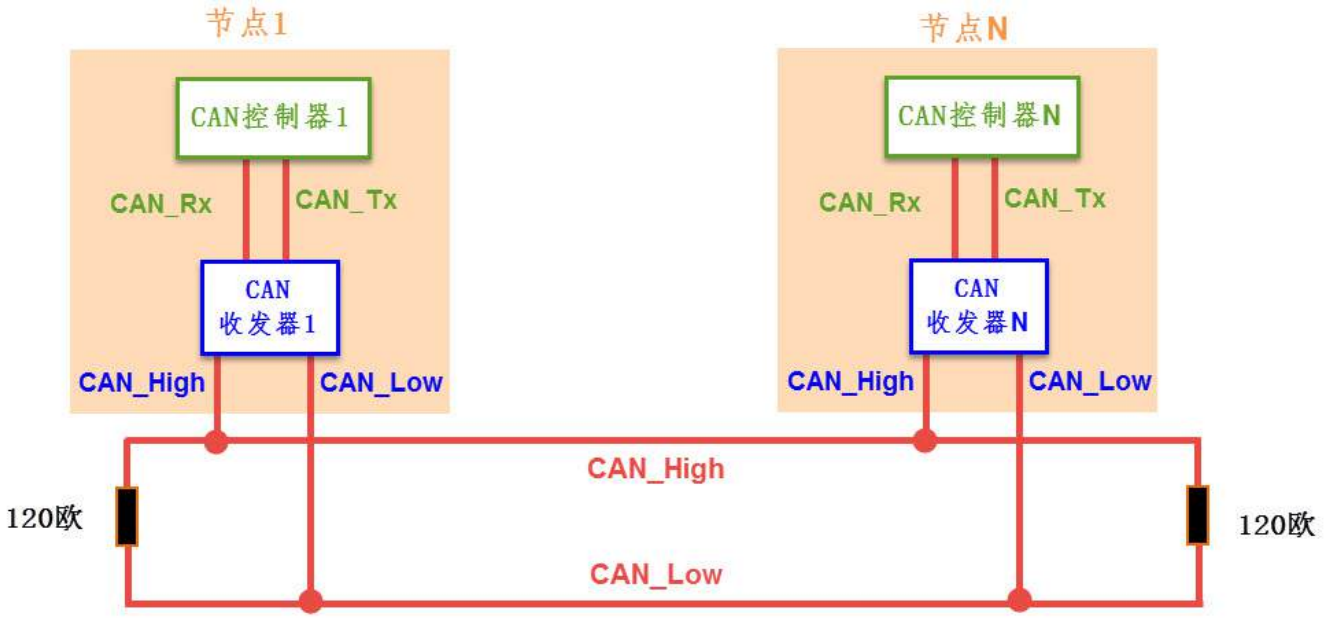


图 39-1 CAN 闭环总线通讯网络

40.1.1.2 开环总线网络

图 *39_0_2* 中的是遵循 ISO11519-2 标准的低速、远距离“开环网络”，它的最大传输距离为 1km，最高通讯速率为 125kbps，两根总线是独立的、不形成闭环，要求每根总线上各串联有一个“2.2 千欧”的电阻。

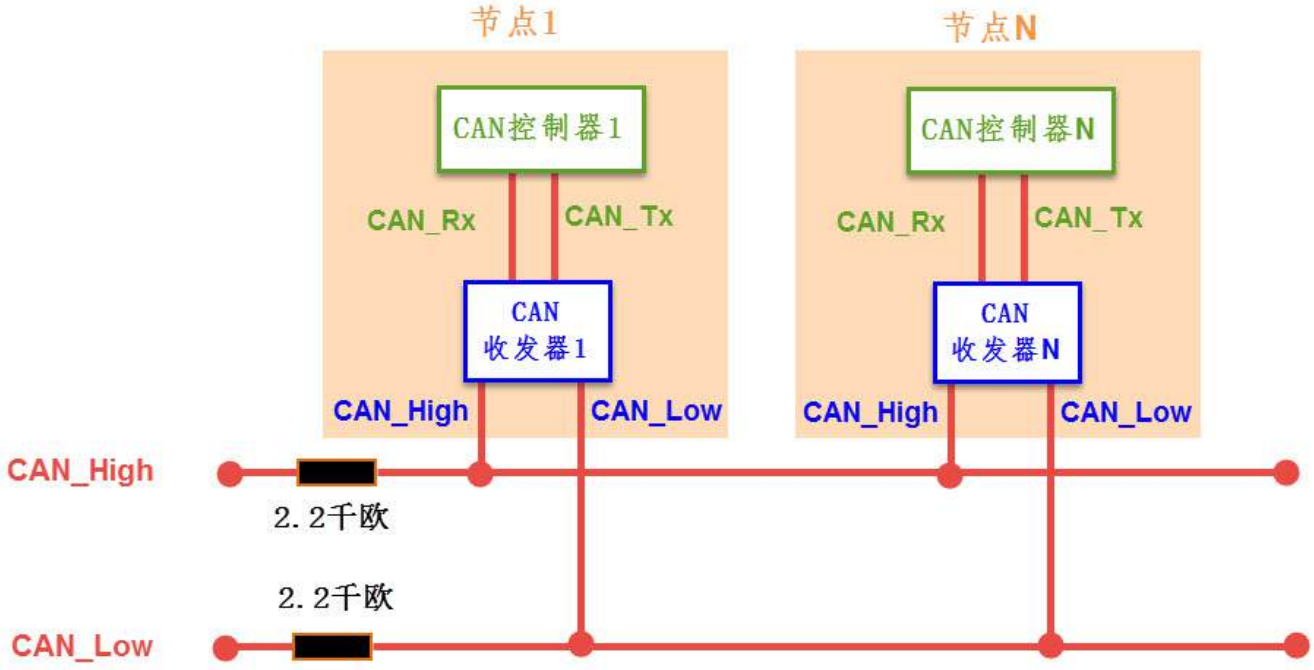


图 39-2 CAN 开环总线通讯网络

40.1.1.3 通讯节点

从 CAN 通讯网络图可了解到，CAN 总线上可以挂载多个通讯节点，节点之间的信号经过总线传输，实现节点间通讯。由于 CAN 通讯协议不对节点进行地址编码，而是对数据内容进行编码的，所以网络中的节点个数理论上不受限制，只要总线的负载足够即可，可以通过中继器增强负载。

CAN 通讯节点由一个 CAN 控制器及 CAN 收发器组成，控制器与收发器之间通过 CAN_Tx 及 CAN_Rx 信号线相连，收发器与 CAN 总线之间使用 CAN_High 及 CAN_Low 信号线相连。其中 CAN_Tx 及 CAN_Rx 使用普通的类似 TTL 逻辑信号，而 CAN_High 及 CAN_Low 是一对差分信号线，使用比较特别的差分信号，下一小节再详细说明。

当 CAN 节点需要发送数据时，控制器把要发送的二进制编码通过 CAN_Tx 线发送到收发器，然后由收发器把这个普通的逻辑电平信号转化成差分信号，通过差分线 CAN_High 和 CAN_Low 线输出到 CAN 总线网络。而通过收发器接收总线上的数据到控制器时，则是相反的过程，收发器把总线上收到的 CAN_High 及 CAN_Low 信号转化成普通的逻辑电平信号，通过 CAN_Rx 输出到控制器中。

例如，STM32 的 CAN 片上外设就是通讯节点中的控制器，为了构成完整的节点，还要给它外接一个收发器，在我们实验板中使用型号为 TJA1050 的芯片作为 CAN 收发器。CAN 控制器与 CAN 收发器的关系如同 TTL 串口与 MAX3232 电平转换芯片的关系，MAX3232 芯片把 TTL 电平的串口信号转换成 RS-232 电平的串口信号，CAN 收发器的作用则是把 CAN 控制器的 TTL 电平信号转换成差分信号 (或者相反) 。

40.1.1.4 差分信号

差分信号又称差模信号，与传统使用单根信号线电压表示逻辑的方式有区别，使用差分信号传输时，需要两根信号线，这两个信号线的振幅相等，相位相反，通过两根信号线的电压差值来表示逻辑 0 和逻辑 1。见图 *39_0_3*，它使用了 V+ 与 V-信号的差值表达出了图下方的信号。

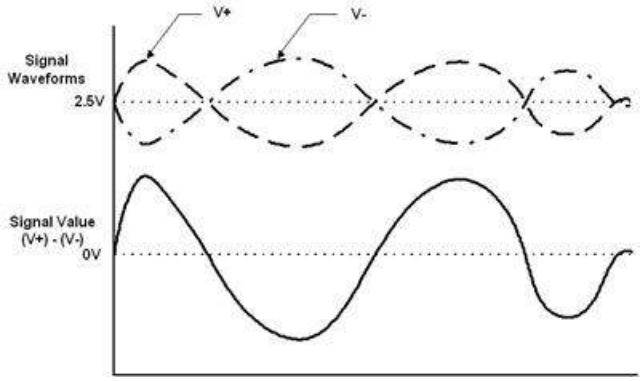


图 39-3 差分信号

相对于单信号线传输的方式，使用差分信号传输具有如下优点：

- 抗干扰能力强，当外界存在噪声干扰时，几乎会同时耦合到两条信号线上，而接收端只关心两个信号的差值，所以外界的共模噪声可以被完全抵消。
- 能有效抑制它对外部的电磁干扰，同样的道理，由于两根信号的极性相反，他们对外辐射的电磁场可以相互抵消，耦合的越紧密，泄放到外界的电磁能量越少。
- 时序定位精确，由于差分信号的开关变化是位于两个信号的交点，而不像普通单端信号依靠高低两个阈值电压判断，因而受工艺，温度的影响小，能降低时序上的误差，同时也更适合于低幅度信号的电路。

由于差分信号线具有这些优点，所以在 USB 协议、485 协议、以太网协议及 CAN 协议的物理层中，都使用了差分信号传输。

40.1.1.5 CAN 协议中的差分信号

CAN 协议中对它使用的 CAN_High 及 CAN_Low 表示的差分信号做了规定，见表 39-1 及图 *39_0_4*。以高速 CAN 协议为例，当表示逻辑 1 时 (隐性电平)，CAN_High 和 CAN_Low 线上的电压均为 2.5v，即它们的电压差 V_H-V:sub:*L*=0V；而表示逻辑 0 时 (显性电平)，CAN_High 的电平为 3.5V，CAN_Low 线的电平为 1.5V，即它们的电压差为 V_H-V:sub:*L*=2V。例如，当 CAN 收发器从 CAN_Tx 线接收到来自 CAN 控制器的低电平信号时 (逻辑 0)，它会使 CAN_High 输出 3.5V，同时 CAN_Low 输出 1.5V，从而输出显性电平表示逻辑 0 。

表 39-1 CAN 协议标准表示的信号逻辑

信号	ISO11898(高速)						ISO11519-2(低速)					
	隐性(逻辑 1)			显性(逻辑 0)			隐性(逻辑 1)			显性(逻辑 0)		
	最小值	典型值	最大值	最小值	典型值	最大值	最小值	典型值	最大值	最小值	典型值	最大值
CAN_High（V）	2.0	2.5	3.0	2.75	3.5	4.5	1.6	1.75	1.9	3.85	4.0	5.0
CAN_Low（V）	2.0	2.5	3.0	0.5	1.5	2.25	3.10	3.25	3.4	0	1.0	1.15
High-Low 电位差(V)	-0.5	0	0.05	1.5	2.0	3.0	-0.3	-1.5	-	0.3	3.0	-

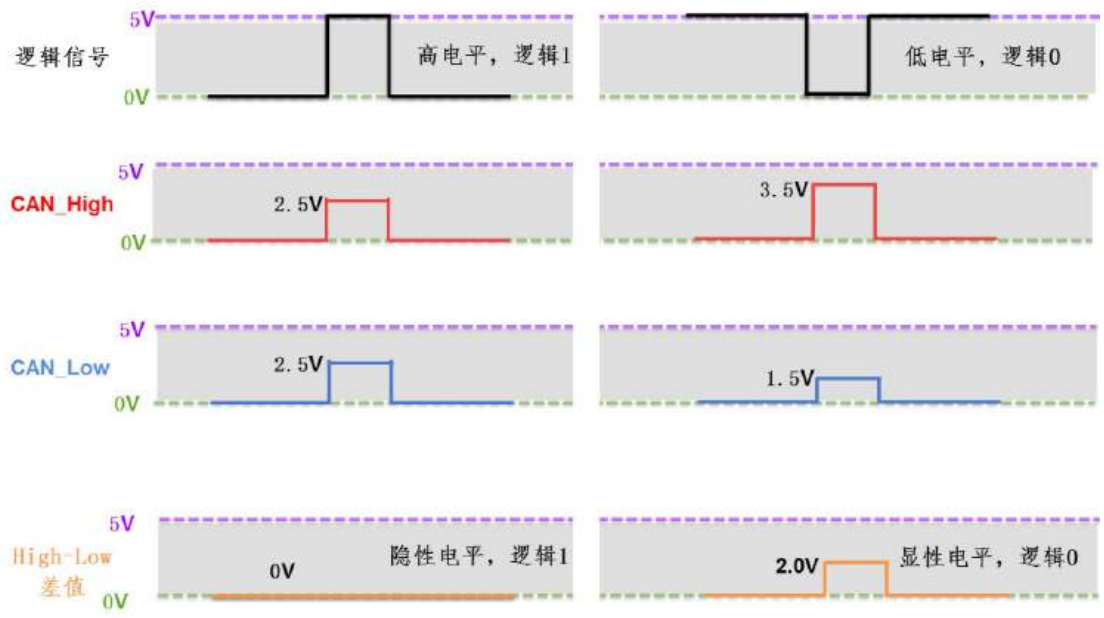


图 39-4 CAN 的差分信号（高速）

在 CAN 总线中，必须使它处于隐性电平 (逻辑 1) 或显性电平 (逻辑 0) 中的其中一个状态。假如有两个 CAN 通讯节点，在同一时间，一个输出隐性电平，另一个输出显性电平，类似 I2C 总线的“线与”特性将使它处于显性电平状态，显性电平的名字就是这样来的，即可以认为显性具有优先的意味。

由于 CAN 总线协议的物理层只有 1 对差分线，在一个时刻只能表示一个信号，所以对通讯节点来说，CAN 通讯是半双工的，收发数据需要分时进行。在 CAN 的通讯网络中，因为共用总线，在整个网络中同一时刻只能有一个通讯节点发送信号，其余的节点在该时刻都只能接收。

40.1.2 协议层

以上是 CAN 的物理层标准，约定了电气特性，以下介绍的协议层则规定了通讯逻辑。

40.1.2.1 CAN 的波特率及位同步

由于 CAN 属于异步通讯，没有时钟信号线，连接在同一个总线网络中的各个节点会像串口异步通讯那样，节点间使用约定好的波特率进行通讯，特别地，CAN 还会使用“位同步”的方式来抗干扰、吸收误差，实现对总线电平信号进行正确的采样，确保通讯正常。

40.1.2.1.1 位时序分解

为了实现位同步，CAN 协议把每一个数据位的时序分解成如图 *39_0_5* 所示的 SS 段、PTS 段、PBS1 段、PBS2 段，这四段的长度加起来即为一个 CAN 数据位的长度。分解后最小的时间单位是 Tq，而一个完整的位由 8~25 个 Tq 组成。为方便表示，图 *39_0_5* 中的高低电平直接代表信号逻辑 0 或逻辑 1(不是差分信号)。

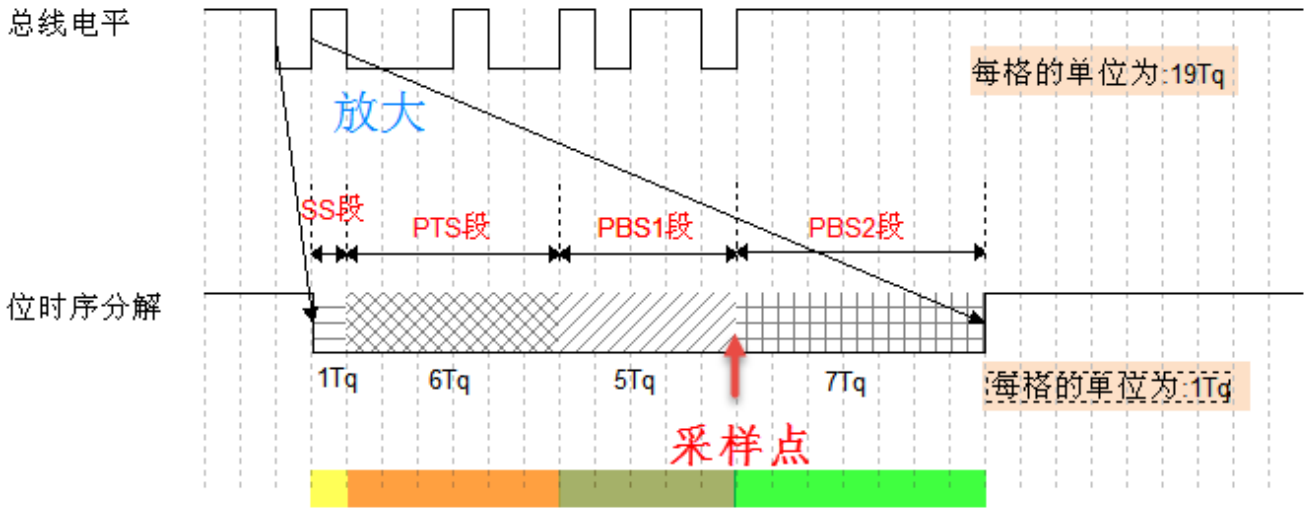


图 39-5 CAN 位时序分解图

该图中表示的 CAN 通讯信号每一个数据位的长度为 19Tq，其中 SS 段占 1Tq，PTS 段占 6Tq，PBS1 段占 5Tq，PBS2 段占 7Tq。信号的采样点位于 PBS1 段与 PBS2 段之间，通过控制各段的长度，可

以对采样点的位置进行偏移，以便准确地采样。

各段的作用如介绍下：

- SS 段 (SYNC SEG)

  SS 译为同步段，若通讯节点检测到总线上信号的跳变沿被包含在 SS 段的范围之内，则表示节点与总线的时序是同步的，当节点与总线同步时，采样点采集到的总线电平即可被确定为该位的电平。SS 段的大小固定为 1Tq。

- PTS 段 (PROP SEG)

  PTS 译为传播时间段，这个时间段是用于补偿网络的物理延时时间。是总线上输入比较器延时和输出驱动器延时总和的两倍。PTS 段的大小可以为 1~8Tq。

- PBS1 段 (PHASE SEG1)，

  PBS1 译为相位缓冲段，主要用来补偿边沿阶段的误差，它的时间长度在重新同步的时候可以加长。PBS1 段的初始大小可以为 1~8Tq。

- PBS2 段 (PHASE SEG2)

  PBS2 这是另一个相位缓冲段，也是用来补偿边沿阶段误差的，它的时间长度在重新同步时可以缩短。PBS2 段的初始大小可以为 2~8Tq。

40.1.2.1.2 通讯的波特率

总线上的各个通讯节点只要约定好 1 个 Tq 的时间长度以及每一个数据位占据多少个 Tq，就可以确定 CAN 通讯的波特率。

例如，假设上图中的 1Tq=1us，而每个数据位由 19 个 Tq 组成，则传输一位数据需要时间 T_{1bit} =19us，从而每秒可以传输的数据位个数为：

$1x10^6/19 = 52631.6$ (bps)

这个每秒可传输的数据位的个数即为通讯中的波特率。

40.1.2.1.3 同步过程分析

波特率只是约定了每个数据位的长度，数据同步还涉及到相位的细节，这个时候就需要用到数据位内的 SS、PTS、PBS1 及 PBS2 段了。

根据对段的应用方式差异，CAN 的数据同步分为硬同步和重新同步。其中硬同步只是当存在“帧起始信号”时起作用，无法确保后续一连串的位时序都是同步的，而重新同步方式可解决该问题，这两种方式具体介绍如下：

(1) 硬同步

若某个 CAN 节点通过总线发送数据时，它会发送一个表示通讯起始的信号 (即下一小节介绍的帧起始信号)，该信号是一个由高变低的下降沿。而挂载到 CAN 总线上的通讯节点在不发送数据时，会时刻检测总线上的信号。

见图 *39_0_6*，可以看到当总线出现帧起始信号时，某节点检测到总线的帧起始信号不在节点内部时序的 SS 段范围，所以判断它自己的内部时序与总线不同步，因而这个状态的采样点采集得的数据是不正确的。所以节点以硬同步的方式调整，把自己的位时序中的 SS 段平移至总线出现下降沿的部分，获得同步，同步后采样点就可以采集得正确数据了。

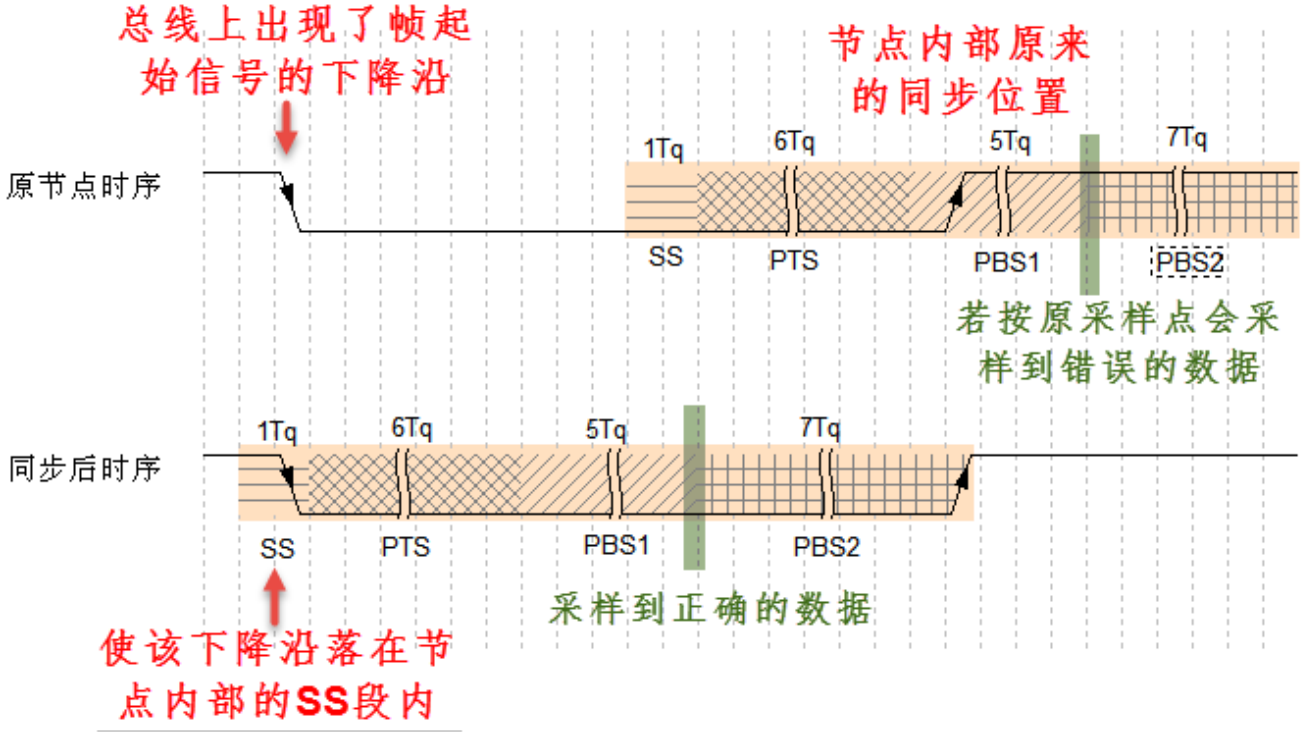


图 39-6 硬同步过程图

(2) 重新同步

前面的硬同步只是当存在帧起始信号时才起作用，如果在一帧很长的数据内，节点信号与总线信号相位有偏移时，这种同步方式就无能为力了。因而需要引入重新同步方式，它利用普通数据位的高至低电平的跳变沿来同步 (帧起始信号是特殊的跳变沿)。重新同步与硬同步方式相似的地方是它们都使用 SS 段来进行检测，同步的目的都是使节点内的 SS 段把跳变沿包含起来。

重新同步的方式分为超前和滞后两种情况，以总线跳变沿与 SS 段的相对位置进行区分。第一种相位超前的情况如图 *39_0_7*，节点从总线的边沿跳变中，检测到它内部的时序比总线的时序相对超前 2Tq，这时控制器在下一个位时序中的 PBS1 段增加 2Tq 的时间长度，使得节点与总线时序重新同步。

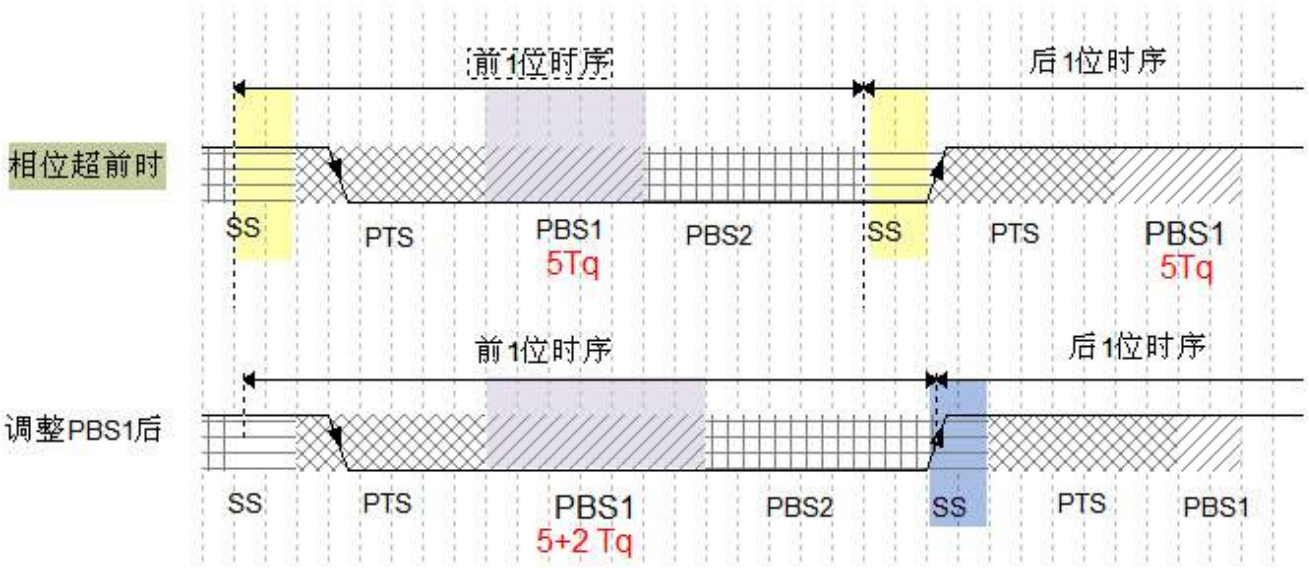


图 39-7 相位超前时的重新同步

第二种相位滞后的情况如图 *39_0_8* ，节点从总线的边沿跳变中，检测到它的时序比总线的时序相对滞后 2Tq，这时控制器在前一个位时序中的 PBS2 段减少 2Tq 的时间长度，获得同步。

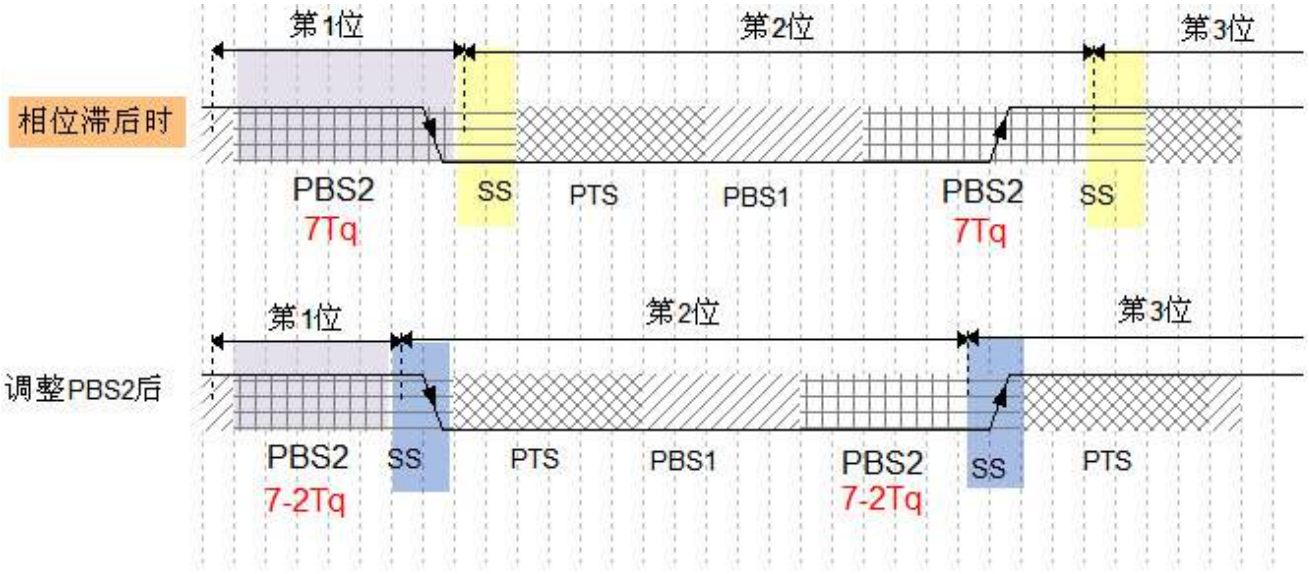


图 39-8 相位滞后时的重新同步

在重新同步的时候，PBS1 和 PBS2 中增加或减少的这段时间长度被定义为“重新同步补偿宽度 SJW* (reSynchronization Jump Width)”。一般来说 CAN 控制器会限定 SJW 的最大值，如限定了最大 SJW=3Tq 时，单次同步调整的时候不能增加或减少超过 3Tq 的时间长度，若有需要，控制器会通过多次小幅度调整来实现同步。当控制器设置的 SJW 极限值较大时，可以吸收的误差加大，但通讯的速度会下降。

40.1.2.2 CAN 的报文种类及结构

在 SPI 通讯中，片选、时钟信号、数据输入及数据输出这 4 个信号都有单独的信号线，I2C 协议包含有时钟信号及数据信号 2 条信号线，异步串口包含接收与发送 2 条信号线，这些协议包含的信号都比 CAN 协议要丰富，它们能轻易进行数据同步或区分数据传输方向。而 CAN 使用的是两条差分信号线，只能表达一个信号，简洁的物理层决定了 CAN 必然要配上一套更复杂的协议，如何用一个信号通道实现同样、甚至更强大的功能呢？CAN 协议给出的解决方案是对数据、操作命令 (如读/写) 以及同步信号进行打包，打包后的这些内容称为报文。

40.1.2.2.1 报文的种类

在原始数据段的前面加上传输起始标签、片选 (识别) 标签和控制标签，在数据的尾段加上 CRC 校验标签、应答标签和传输结束标签，把这些内容按特定的格式打包好，就可以用一个通道表达各种信号了，各种各样的标签就如同 SPI 中各种通道上的信号，起到了协同传输的作用。当整个数据包被传输到其它设备时，只要这些设备按格式去解读，就能还原出原始数据，这样的报文就被称为 CAN 的“数据帧”。

为了更有效地控制通讯，CAN 一共规定了 5 种类型的帧，它们的类型及用途说明如表 39-2。

表 39-2 帧的种类及其用途

帧	帧用途
数据帧	用于节点向外传送数据
遥控帧	用于向远端节点请求数据
错误帧	用于向远端节点通知校验错误，请求重新发送上一个数据
过载帧	用于通知远端节点：本节点尚未做好接收准备
帧间隔	用于将数据帧及遥控帧与前面的帧分离开来

40.1.2.2.2 数据帧的结构

数据帧是在 CAN 通讯中最主要、最复杂的报文，我们来了解它的结构，见图 *39_0_9*。

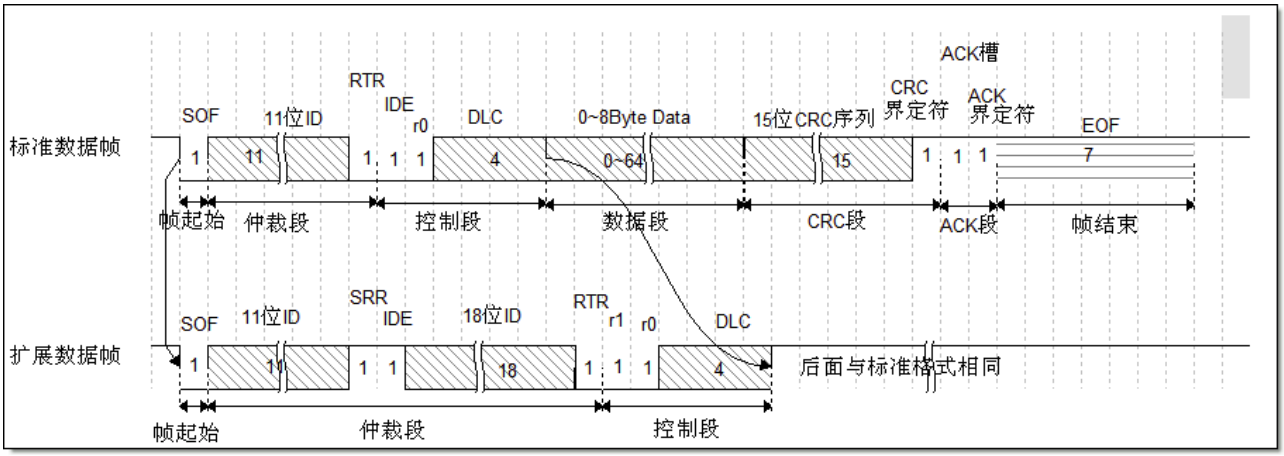


图 39-9 数据帧的结构

数据帧以一个显性位 (逻辑 0) 开始，以 7 个连续的隐性位 (逻辑 1) 结束，在它们之间，分别有仲裁段、控制段、数据段、CRC 段和 ACK 段。

- 帧起始

SOF 段 (Start OfFrame)，译为帧起始，帧起始信号只有一个数据位，是一个显性电平，它用于通知各个节点将有数据传输，其它节点通过帧起始信号的电平跳变沿来进行硬同步。

- 仲裁段

当同时有两个报文被发送时，总线会根据仲裁段的内容决定哪个数据包能被传输，这也是它名称的由来。

仲裁段的内容主要为本数据帧的 ID 信息 (标识符)，数据帧具有标准格式和扩展格式两种，区别就在于 ID 信息的长度，标准格式的 ID 为 11 位，扩展格式的 ID 为 29 位，它在标准 ID 的基础上多出 18 位。在 CAN 协议中，ID 起着重要的作用，它决定着数据帧发送的优先级，也决定着其它节点是否会接收这个数据帧。CAN 协议不对挂载在它之上的节点分配优先级和地址，对总线的占有权是由信息的重要性决定的，即对于重要的信息，我们会给它打包上一个优先级高的 ID，使它能够及时地发送出去。也正因为它这样的优先级分配原则，使得 CAN 的扩展性大大加强，在总线上增加或减少节点并不影响其它设备。

报文的优先级，是通过对 ID 的仲裁来确定的。根据前面对物理层的分析我们知道如果总线上同时出现显性电平和隐性电平，总线的状态会被置为显性电平，CAN 正是利用这个特性进行仲裁。

若两个节点同时竞争 CAN 总线的占有权，当它们发送报文时，若首先出现隐性电平，则会失去对总线的占有权，进入接收状态。见图 *39_0_10*，在开始阶段，两个设备发送的电平一样，所以它们一直继续发送数据。到了图中箭头所指的时序处，节点单元 1 发送的为隐性电平，而此时节点单元 2 发送的为显性电平，由于总线的“线与”特性使它表达出显示电平，因此单元 2 竞争总线成功，这个报文得以被继续发送出去。

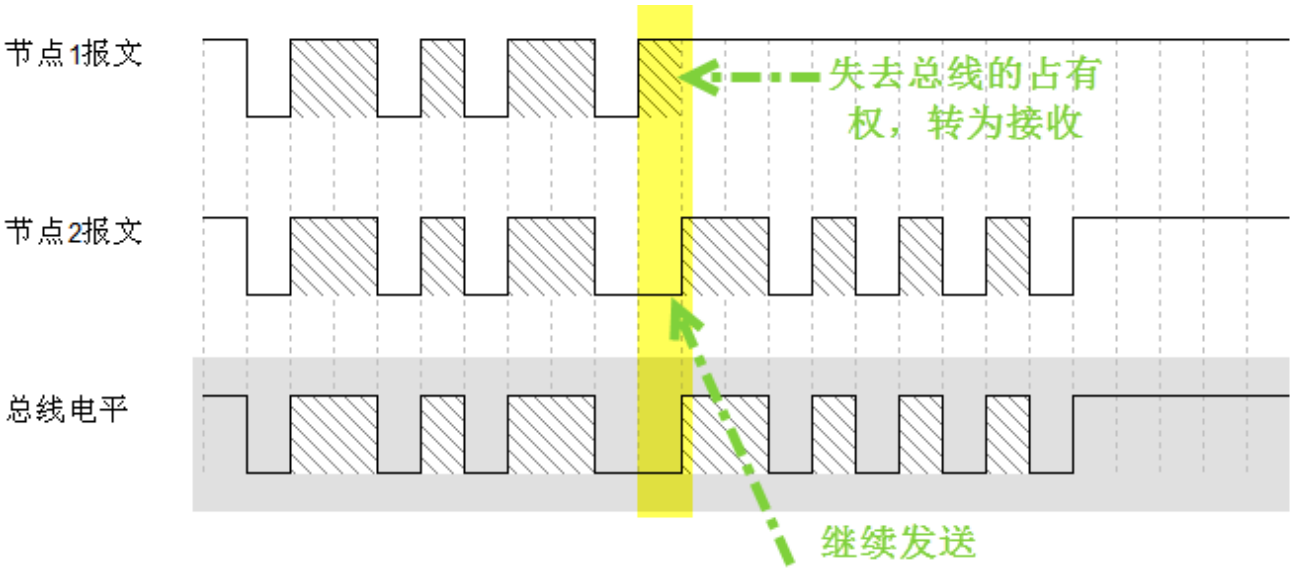


图 39-10 仲裁过程

仲裁段 ID 的优先级也影响着接收设备对报文的反应。因为在 CAN 总线上数据是以广播的形式发送的，所有连接在 CAN 总线的节点都会收到所有其它节点发出的有效数据，因而我们的 CAN 控制器大多具有根据 ID 过滤报文的功能，它可以控制自己只接收某些 ID 的报文。

回看图 *39_0_9* 中的数据帧格式，可看到仲裁段除了报文 ID 外，还有 RTR、IDE 和 SRR 位。

(1) RTR 位 (Remote Transmission Request Bit)，译作远程传输请求位，它是用于区分数据帧和遥控帧的，当它为显性电平时表示数据帧，隐性电平时表示遥控帧。

(2) IDE 位 (Identifier ExtensionBit)，译作标识符扩展位，它是用于区分标准格式与扩展格式，当它为显性电平时表示标准格式，隐性电平时表示扩展格式。

(3) SRR 位 (Substitute Remote Request Bit)，只存在于扩展格式，它用于替代标准格式中的 RTR 位。由于扩展帧中的 SRR 位为隐性位，RTR 在数据帧为显性位，所以在两个 ID 相同的标准格式报文与扩展格式报文中，标准格式的优先级较高。

- 控制段

在控制段中的 r1 和 r0 为保留位，默认设置为显性位。它最主要的是 DLC 段 (Data Length Code)，译为数据长度码，它由 4 个数据位组成，用于表示本报文中的数据段含有多少个字节，DLC 段表示的数字为 0~8。

- 数据段

数据段为数据帧的核心内容，它是节点要发送的原始信息，由 0~8 个字节组成，MSB 先行。

- CRC 段

为了保证报文的正确传输，CAN 的报文包含了一段 15 位的 CRC 校验码，一旦接收节点算出的 CRC 码跟接收到的 CRC 码不同，则它会向发送节点反馈出错信息，利用错误帧请求它重新发送。CRC 部分的计算一般由 CAN 控制器硬件完成，出错时的处理则由软件控制最大重发数。

在 CRC 校验码之后，有一个 CRC 界定符，它为隐性位，主要作用是把 CRC 校验码与后面的 ACK 段间隔起来。

- ACK 段

ACK 段包括一个 ACK 槽位，和 ACK 界定符位。类似 I2C 总线，在 ACK 槽位中，发送节点发送的是隐性位，而接收节点则在这一位中发送显性位以示应答。在 ACK 槽和帧结束之间由 ACK 界定符间隔开。

- 帧结束

EOF 段 (End Of Frame)，译为帧结束，帧结束段由发送节点发送的 7 个隐性位表示结束。

40.1.2.2.3 其它报文的结构

关于其它的 CAN 报文结构，不再展开讲解，其主要内容见图 *39_0_11*。

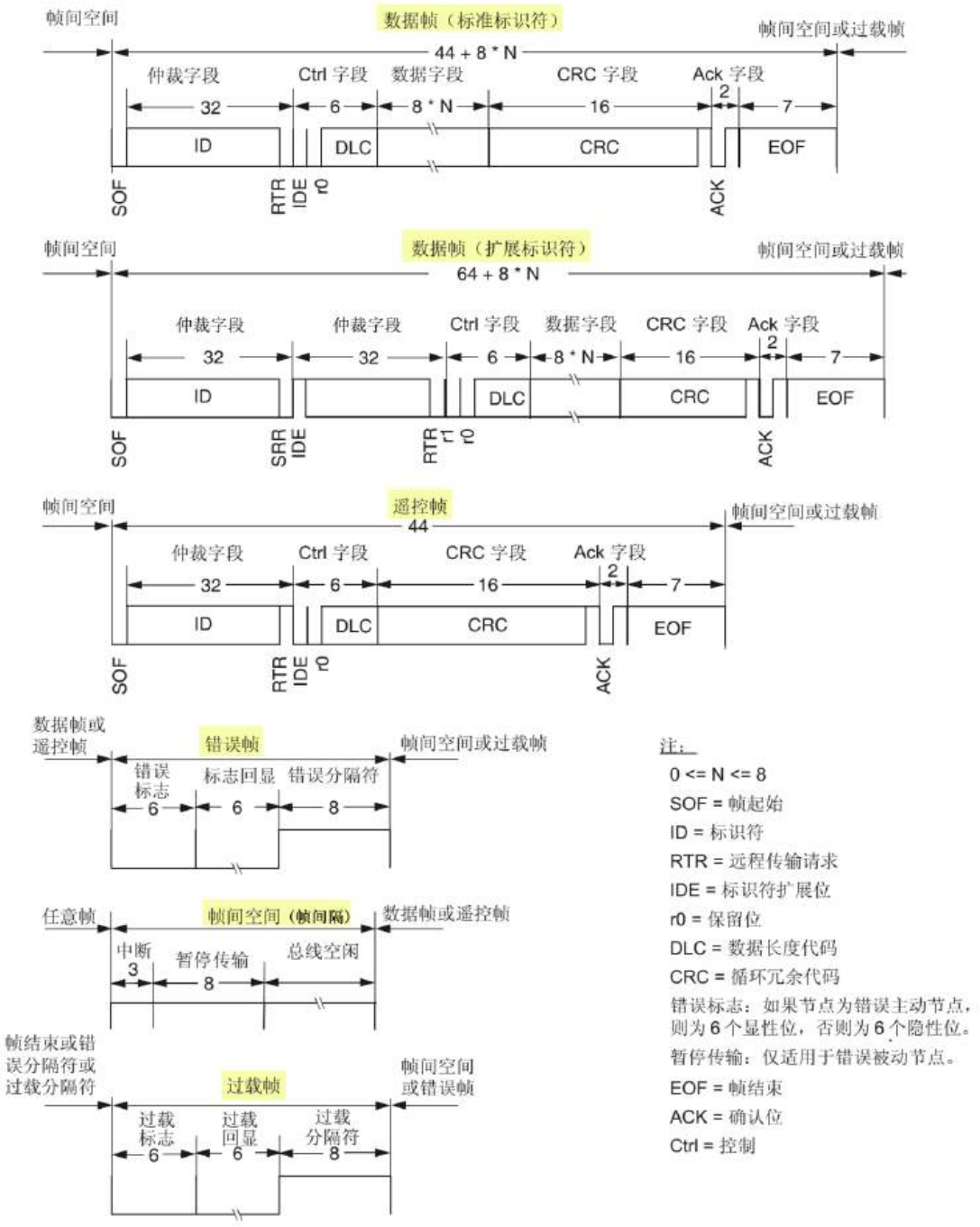


图 39-11 各种 CAN 报文的结构

40.2 STM32 的 CAN 外设简介

STM32 的芯片中具有 bxCAN 控制器 (Basic Extended CAN)，它支持 CAN 协议 2.0A 和 2.0B 标准。

该 CAN 控制器支持最高的通讯速率为 1Mb/s；可以自动地接收和发送 CAN 报文，支持使用标准 ID 和扩展 ID 的报文；外设中具有 3 个发送邮箱，发送报文的优先级可以使用软件控制，还可以记录发送的时间；具有 2 个 3 级深度的接收 FIFO，可使用过滤功能只接收或不接收某些 ID 号的报文；可配置成自动重发；不支持使用 DMA 进行数据收发。

40.2.1 STM32 的 CAN 架构剖析

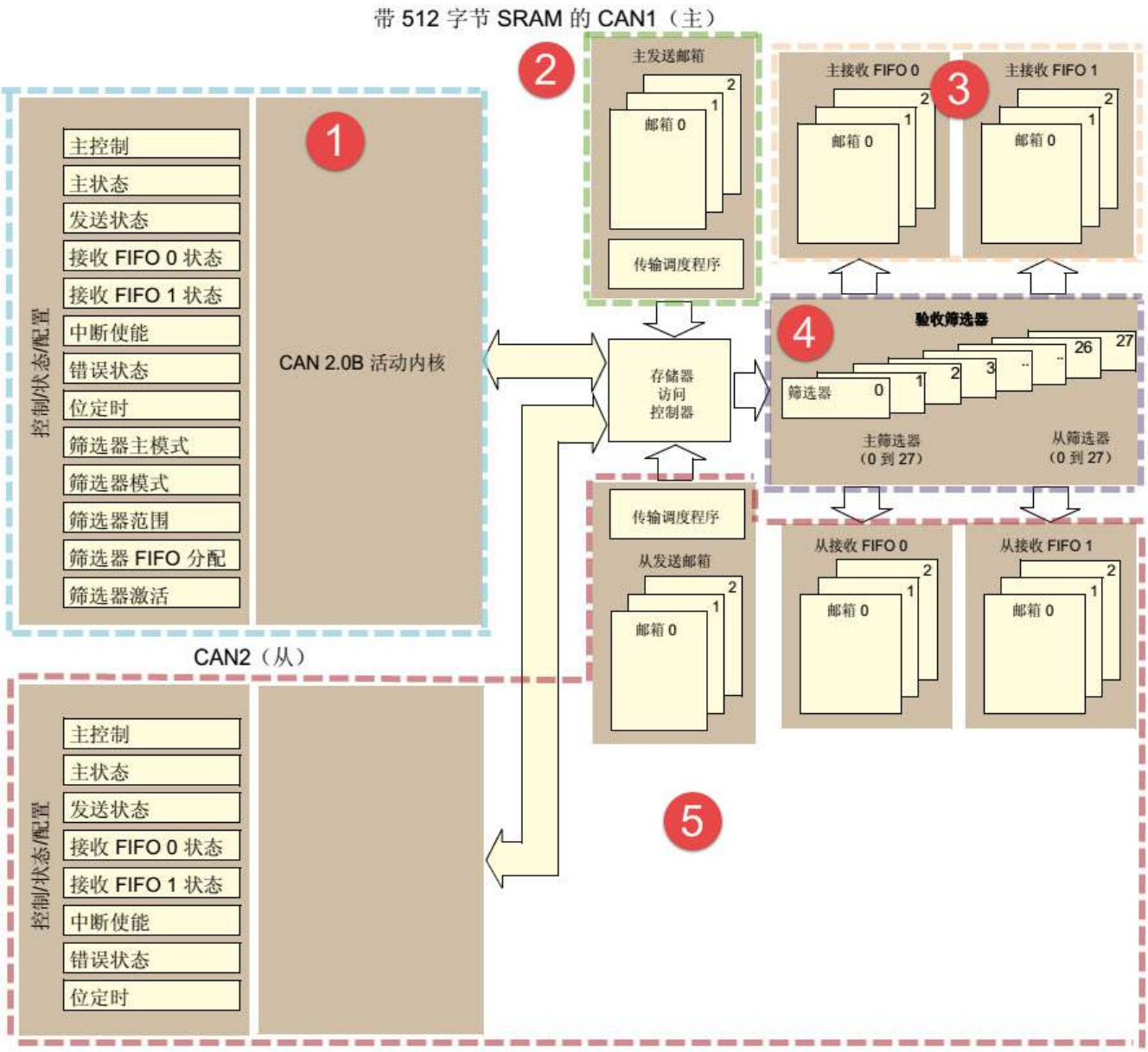


图 39-12 STM32 的 CAN 外设架构图（互联型产品）

图 *39_0_12* 是 STM32F105/107 系列互联型芯片的 CAN 外设架构图，图里具有 2 组 CAN 控制器，其中 CAN1 是主设备，框图中的“存储访问控制器”是由 CAN1 控制的，CAN2 无法直接访问存

储区域，所以使用 CAN2 的时候必须使能 CAN1 外设的时钟。框图中主要包含 CAN 控制内核、发送邮箱、接收 FIFO 以及验收筛选器。我们实验板中使用的 STM32F103 系列芯片跟上述框图类似，但该系列只包含 1 组 CAN 控制器，即它们不包含图中标号的部分。

下面对框图中的各个部分进行介绍。

40.2.1.1 CAN 控制内核

框图中标号处的 CAN 控制内核包含了各种控制寄存器及状态寄存器，我们主要讲解其中的主控制寄存器 CAN_MCR 及位时序寄存器 CAN_BTR。

40.2.1.1.1 主控制寄存器 CAN_MCR

主控制寄存器 CAN_MCR 负责管理 CAN 的工作模式，它使用以下寄存器位实现控制。

(1) DBF 调试冻结功能

DBF(Debug freeze) 调试冻结，使用它可设置 CAN 处于工作状态或禁止收发的状态，禁止收发时仍可访问接收 FIFO 中的数据。这两种状态是当 STM32 芯片处于程序调试模式时才使用的，平时使用并不影响。

(2) *TTCM* 时间触发模式

TTCM(Time triggered communication mode) 时间触发模式，它用于配置 CAN 的时间触发通信模式，在此模式下，CAN 使用它内部定时器产生时间戳，并把它保存在 CAN_RDTxR、CAN_TDTxR 寄存器中。内部定时器在每个 CAN 位时间累加，在接收和发送的帧起始位被采样，并生成时间戳。利用它可以实现 ISO 11898-4 CAN 标准的分时同步通信功能。

(3) ABOM 自动离线管理

ABOM (Automatic bus-off management) 自动离线管理，它用于设置是否使用自动离线管理功能。当节点检测到它发送错误或接收错误超过一定值时，会自动进入离线状态，在离线状态中，CAN 不能接收或发送报文。处于离线状态的时候，可以软件控制恢复或者直接使用这个自动离线管理功能，它会在适当的时候自动恢复。

(4) AWUM 自动唤醒

AWUM (Automatic bus-off management)，自动唤醒功能，CAN 外设可以使用软件进入低功耗的睡眠模式，如果使能了这个自动唤醒功能，当 CAN 检测到总线活动的时候，会自动唤醒。

(5) NART 自动重传

NART(No automatic retransmission) 报文自动重传功能，设置这个功能后，当报文发送失败时会自动重传至成功为止。若不使用这个功能，无论发送结果如何，消息只发送一次。

(6) RFLM 锁定模式

RFLM(Receive FIFO locked mode)FIFO 锁定模式，该功能用于锁定接收 FIFO 。锁定后，当接收 FIFO 溢出时，会丢弃下一个接收的报文。若不锁定，则下一个接收到的报文会覆盖原报文。

(7) TXFP 报文发送优先级的判定方法

TXFP(Transmit FIFO priority) 报文发送优先级的判定方法，当 CAN 外设的发送邮箱中有多个待发送报文时，本功能可以控制它是根据报文的 ID 优先级还是报文存进邮箱的顺序来发送。

40.2.1.1.2 位时序寄存器 (CAN_BTR) 及波特率

CAN 外设中的位时序寄存器 CAN_BTR 用于配置测试模式、波特率以及各种位内的段参数。

(1) 测试模式

为方便调试，STM32 的 CAN 提供了测试模式，配置位时序寄存器 CAN_BTR 的 SILM 及 LBKM 寄存器位可以控制使用正常模式、静默模式、回环模式及静默回环模式，见图 *39_0_13*。

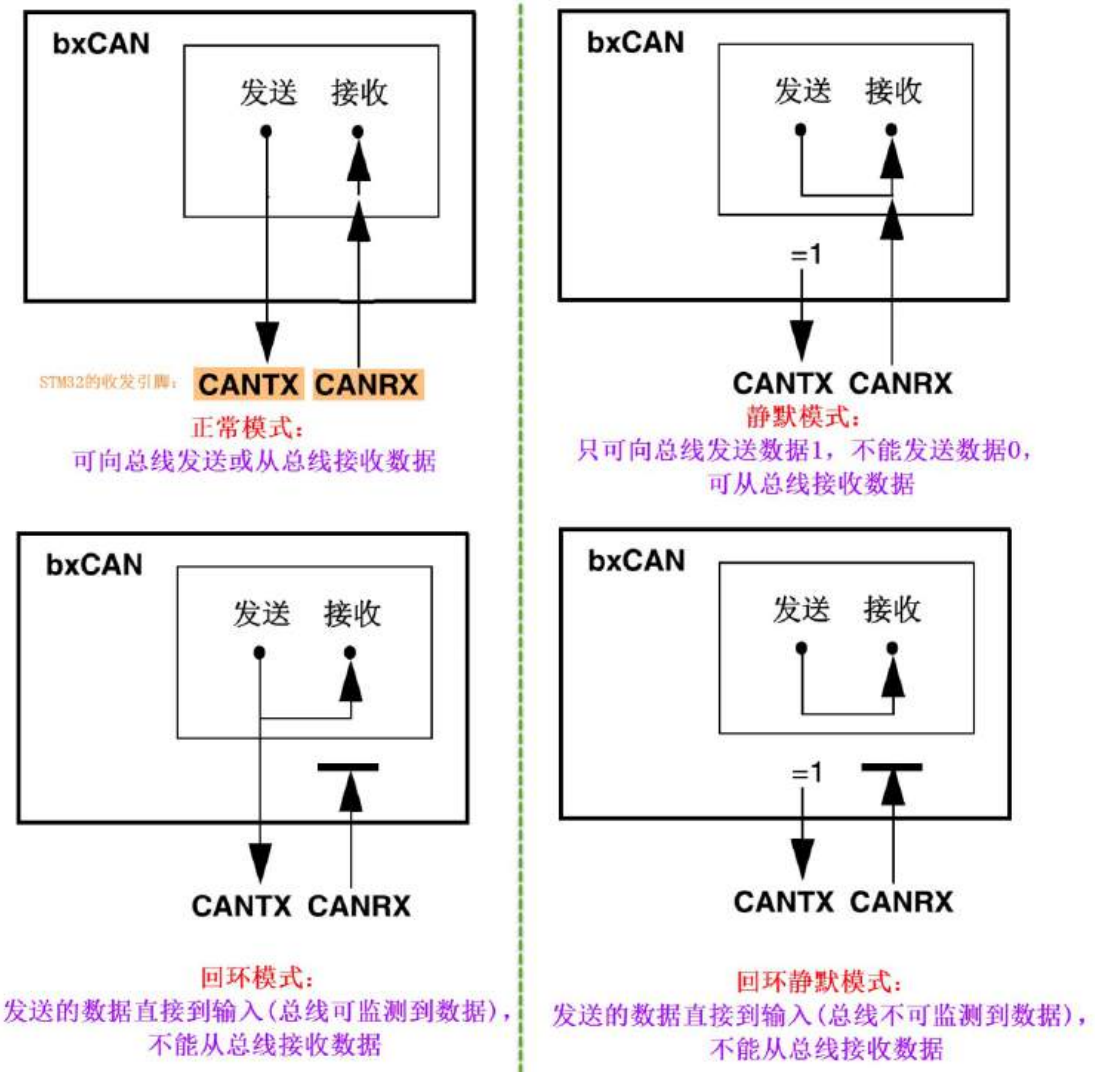


图 39-13 四种工作模式

各个工作模式介绍如下：

- 正常模式

正常模式下就是一个正常的 CAN 节点，可以向总线发送数据和接收数据。

- 静默模式

  静默模式下，它自己的输出端的逻辑 0 数据会直接传输到它自己的输入端，逻辑 1 可以被发送到总线，所以它不能向总线发送显性位 (逻辑 0)，只能发送隐性位 (逻辑 1)。

输入端可以从总线接收内容。由于它只可发送的隐性位不会强制影响总线的状态，所以把它称为静默模式。这种模式一般用于监测，它可以用于分析总线上的流量，但又不会因为发送显性位而影响总线。

- 回环模式

  回环模式下，它自己的输出端的所有内容都直接传输到自己的输入端，输出端的内容同时也会被传输到总线上，即也可使用总线监测它的发送内容。输入端只接收自己发送端的内容，不接收来自总线上的内容。使用回环模式可以进行自检。

- 回环静默模式

  回环静默模式是以上两种模式的结合，自己的输出端的所有内容都直接传输到自己的输入端，并且不会向总线发送显性位影响总线，不能通过总线监测它的发送内容。输入端只接收自己发送端的内容，不接收来自总线上的内容。这种方式可以在“热自检”时使用，即自我检查的时候，不会干扰总线。

以上说的各个模式，是不需要修改硬件接线的，例如，当输出直接连输入时，它是在 STM32 芯片内部连接的，传输路径不经过 STM32 的 CAN_Tx/Rx 引脚，更不经过外部连接的 CAN 收发器，只有输出数据到总线或从总线接收的情况下才会经过 CAN_Tx/Rx 引脚和收发器。

(2) 位时序及波特率

STM32 外设定义的位时序与我们前面解释的 CAN 标准时序有一点区别，见图 *39_0_14*。

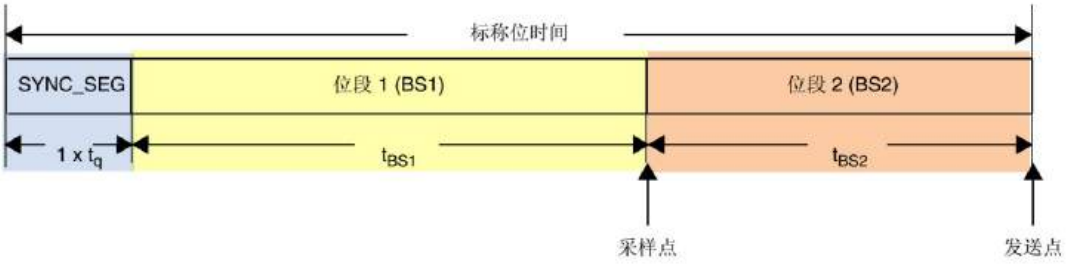


图 39-14 STM32 中 CAN 的位时序

STM32 的 CAN 外设位时序中只包含 3 段，分别是同步段 SYNC_SEG、位段 BS1 及位段 BS2，采样点位于 BS1 及 BS2 段的交界处。其中 SYNC_SEG 段固定长度为 1Tq，而 BS1 及 BS2 段可以在位时序寄存器 CAN_BTR 设置它们的时间长度，它们可以在重新同步期间增长或缩短，该长度 SJW 也可在位时序寄存器中配置。

理解 STM32 的 CAN 外设的位时序时，可以把它的 BS1 段理解为是由前面介绍的 CAN 标准协议中 PTS 段与 PBS1 段合在一起的，而 BS2 段就相当于 PBS2 段。

了解位时序后，我们就可以配置波特率了。通过配置位时序寄存器 CAN_BTR 的 TS1[3:0] 及 TS2[2:0] 寄存器位设定 BS1 及 BS2 段的长度后，我们就可以确定每个 CAN 数据位的时间：

BS1 段时间：

T_{S1}=Tq x (TS1[3:0] + 1)，

BS2 段时间：

T_{S2}= Tq x (TS2[2:0] + 1)，

一个数据位的时间：

T_{1bit} =1Tq+T_{S1}+T_{S2} =1+ (TS1[3:0] + 1)+ (TS2[2:0] + 1)= N Tq

其中单个时间片的长度 Tq 与 CAN 外设的所挂载的时钟总线及分频器配置有关，CAN1 和 CAN2 外设都是挂载在 APB1 总线上的，而位时序寄存器 CAN_BTR 中的 BRP[9:0] 寄存器位可以设置 CAN 外设时钟的分频值，所以：

Tq = (BRP[9:0]+1) x T_{PCLK}

其中的 PCLK 指 APB1 时钟，默认值为 42MHz。

最终可以计算出 CAN 通讯的波特率：

BaudRate = 1/N Tq

例如表 39-3 说明了一种把波特率配置为 1Mbps 的方式。

表 39-3 一种配置波特率为 1Mbps 的方式

参数	说明
SYNC_SE 段	固定为 1Tq
BS1 段	设置为 4Tq (实际写入 TS1[3:0] 的值为 3)
BS2 段	设置为 2Tq (实际写入 TS2[2:0] 的值为 1)
T_{PCLK}	APB1 按默认配置为 F=42MHz，T_{PCLK}=1/42M
CAN 外设时钟分频	设置为 6 分频 (实际写入 BRP[9:0] 的值为 4)
1Tq 时间长度	Tq = (BRP[9:0]+1) x T_{PCLK} = 6 x 1/42M=1/7M
1 位的时间长度	T_{1bit} =1Tq+T_{S1}+T_{S2} = 1+4+2 = 7Tq
波特率	BaudRate = 1/N Tq = 1/(1/7M x 7)=1Mbps

40.2.1.2 CAN 发送邮箱

回到图 24-5 中的 CAN 外设框图，在标号处的是 CAN 外设的发送邮箱，它一共有 3 个发送邮箱，即最多可以缓存 3 个待发送的报文。

每个发送邮箱中包含有标识符寄存器 CAN_TIxR、数据长度控制寄存器 CAN_TDTxR 及 2 个数据寄存器 CAN_TDLxR、CAN_TDHxR，它们的功能见表 39-5。

表 39-4 发送邮箱的寄存器

寄存器名	功能
标识符寄存器 CAN_TIxR	存储待发送报文的 ID、扩展 ID、IDE 位及 RTR 位
数据长度控制寄存器 CAN_TDTxR	存储待发送报文的 DLC 段
低位数据寄存器 CAN_TDLxR	存储待发送报文数据段的 Data0-Data3 这四个字节的内容
高位数据寄存器 CAN_TDHxR	存储待发送报文数据段的 Data4-Data7 这四个字节的内容

当我们要使用 CAN 外设发送报文时，把报文的各个段分解，按位置写入到这些寄存器中，并对标识符寄存器 CAN_TIxR 中的发送请求寄存器位 TMIDxR_TXRQ 置 1，即可把数据发送出去。

其中标识符寄存器 CAN_TIxR 中的 STDID 寄存器位比较特别。我们知道 CAN 的标准标识符的总位数为 11 位，而扩展标识符的总位数为 29 位的。当报文使用扩展标识符的时候，标识符寄存

器 CAN_TIxR 中的 STDID[10:0] 等效于 EXTID[18:28] 位，它与 EXTID[17:0] 共同组成完整的 29 位扩展标识符。

40.2.1.3 CAN 接收 FIFO

图 *39_0_12* 中的 CAN 外设框图，在标号处的是 CAN 外设的接收 FIFO，它一共有 2 个接收 FIFO，每个 FIFO 中有 3 个邮箱，即最多可以缓存 6 个接收到的报文。当接收到报文时，FIFO 的报文计数器会自增，而 STM32 内部读取 FIFO 数据之后，报文计数器会自减，我们通过状态寄存器可获知报文计数器的值，而通过前面主控制寄存器的 RFLM 位，可设置锁定模式，锁定模式下 FIFO 溢出时会丢弃新报文，非锁定模式下 FIFO 溢出时新报文会覆盖旧报文。

跟发送邮箱类似，每个接收 FIFO 中包含有标识符寄存器 CAN_RIxR、数据长度控制寄存器 CAN_RDTxR 及 2 个数据寄存器 CAN_RDLxR、CAN_RDHxR，它们的功能见表 39-5。

表 39-5 发送邮箱的寄存器

寄存器名	功能
标识符寄存器 CAN_RIxR	存储收到报文的 ID、扩展 ID、IDE 位及 RTR 位
数据长度控制寄存器 CAN_RDTxR	存储收到报文的 DLC 段
低位数据寄存器 CAN_RDLxR	存储收到报文数据段的 Data0-Data3 这四个字节的内容
高位数据寄存器 CAN_RDHxR	存储收到报文数据段的 Data4-Data7 这四个字节的内容

通过中断或状态寄存器知道接收 FIFO 有数据后，我们再读取这些寄存器的值即可把接收到的报文加载到 STM32 的内存中。

40.2.1.4 验收筛选器

图 *39_0_12* 中的 CAN 外设框图，在标号处的是 CAN 外设的验收筛选器，一共有 28 个筛选器组，每个筛选器组有 2 个寄存器，CAN1 和 CAN2 共用的筛选器的。其中 STM32F103 系列芯片仅有 14 个筛选器组：0-13 号。

在 CAN 协议中，消息的标识符与节点地址无关，但与消息内容有关。因此，发送节点将报文广播给所有接收器时，接收节点会根据报文标识符的值来确定软件是否需要该消息，为了简化软件

的工作，STM32 的 CAN 外设接收报文前会先使用验收筛选器检查，只接收需要的报文到 FIFO 中。

筛选器工作的时候，可以调整筛选 ID 的长度及过滤模式。根据筛选 ID 长度来分类有有以下两种：

(1) 检查 STDID[10:0]、EXTID[17:0]、IDE 和 RTR 位，一共 31 位。

(2) 检查 STDID[10:0]、RTR、IDE 和 EXTID[17:15]，一共 16 位。

通过配置筛选尺度寄存器 CAN_FS1R 的 FSCx 位可以设置筛选器工作在哪个尺度。

而根据过滤的方法分为以下两种模式：

(1) 标识符列表模式，它把要接收报文的 ID 列成一个表，要求报文 ID 与列表中的某一个标识符完全相同才可以接收，可以理解为白名单管理。

(2) 掩码模式，它把可接收报文 ID 的某几位作为列表，这几位被称为掩码，可以把它理解成关键字搜索，只要掩码 (关键字) 相同，就符合要求，报文就会被保存到接收 FIFO。

通过配置筛选模式寄存器 CAN_FM1R 的 FBMx 位可以设置筛选器工作在哪个模式。

不同的尺度和不同的过滤方法可使筛选器工作在图 39-15 的 4 种状态。

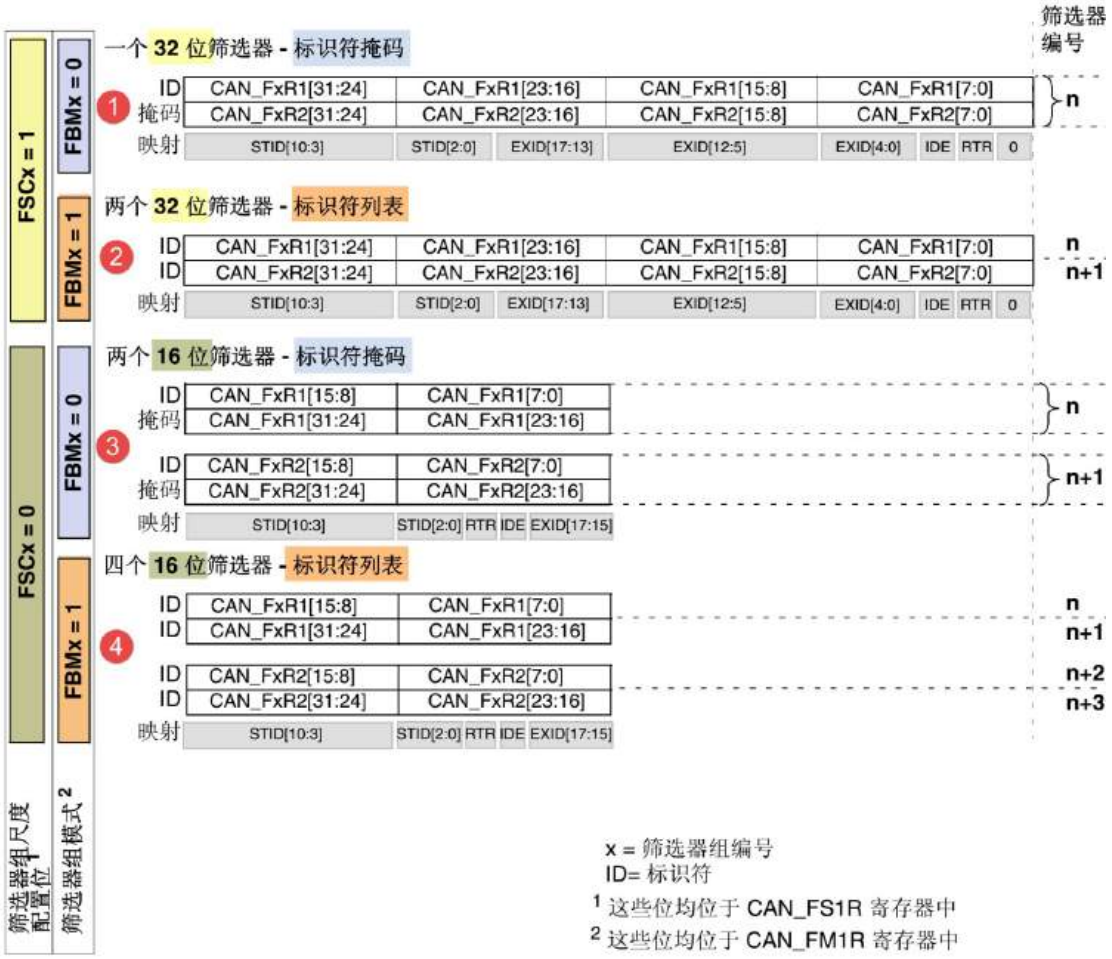


图 39-15 筛选器的 4 种工作状态

每组筛选器包含 2 个 32 位的寄存器，分别为 CAN_FxR1 和 CAN_FxR2，它们用来存储要筛选的 ID 或掩码，各个寄存器位代表的意义与图中两个寄存器下面“映射”的一栏一致，各个模式的说明见表 39-6。

表 39-6 筛选器的工作状态说明

模式	说明
32 位掩码模式	CAN_FxR1 存储 ID，CAN_FxR2 存储哪个位必须要与 CAN_FxR1 中的 ID 一致，2 个寄存器表示 1 组掩码。
32 位标识符模式	CAN_FxR1 和 CAN_FxR2 各存储 1 个 ID，2 个寄存器表示 2 个筛选的 ID
16 位掩码模式	CAN_FxR1 高 16 位存储 ID，低 16 位存储哪个位必须要与高 16 位的 ID 一致； CAN_FxR2 高 16 位存储 ID，低 16 位存储哪个位必须要与高 16 位的 ID 一致 2 个寄存器表示 2 组掩码。
16 位标识符模式	CAN_FxR1 和 CAN_FxR2 各存储 2 个 ID，2 个寄存器表示 4 个筛选的 ID

例如下面的表格所示，在掩码模式时，第一个寄存器存储要筛选的 ID，第二个寄存器存储掩码，掩码为 1 的部分表示该位必须与 ID 中的内容一致，筛选的结果为表中第三行的 ID 值，它是一组包含多个的 ID 值，其中 x 表示该位可以为 1 可以为 0。

ID	1	0	1	1	1	0	1	…
掩码	1	1	1	0	0	1	0	…
筛选的 ID	1	0	1	x	x	0	x	…

而工作在标识符模式时，2 个寄存器存储的都是要筛选的 ID，它只包含 2 个要筛选的 ID 值 (32 位模式时)。

如果使能了筛选器，且报文的 ID 与所有筛选器的配置都不匹配，CAN 外设会丢弃该报文，不存入接收 FIFO。

40.2.1.5 整体控制逻辑

回到图 39-12 结构框图，图中的标号处表示的是 CAN2 外设的结构，它与 CAN1 外设是一样的，他们共用筛选器且由于存储访问控制器由 CAN1 控制，所以要使用 CAN2 的时候必须要使能 CAN1 的时钟。其中 STM32F103 系列芯片不具有 CAN2 控制器。

40.3 CAN 初始化结构体

从 STM32 的 CAN 外设我们了解到它的功能非常多，控制涉及的寄存器也非常丰富，而使用 STM32 HAL 库提供的各种结构体及库函数可以简化这些控制过程。跟其它外设一样，STM32 HAL 库提供了 CAN 初始化结构体及初始化函数来控制 CAN 的工作方式，提供了收发报文使用的结构体及收发函数，还有配置控制筛选器模式及 ID 的结构体。这些内容都定义在库文件“stm32f4xx_hal_can.h”及“stm32f4xx_ hal_can.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。

首先我们来学习初始化结构体的内容，见*代码清单 39_0_1*。

列表 1: 代码清单 39-1 CAN 初始化结构体

```
/**
   * @brief  CAN 初始化结构体
   */
typedef struct {
   uint32_t Prescaler;         /* 配置 CAN 外设的时钟分频，可设置为 1-1024*/
   uint32_t  Mode;             /* 配置 CAN 的工作模式，回环或正常模式 */
   uint32_t  SJW;              /* 配置 SJW 极限值 */
   uint32_t  BS1;              /* 配置 BS1 段长度 */
   uint32_t  BS2;              /* 配置 BS2 段长度 */
   uint32_t  TTCM;            /* 是否使能 TTCM 时间触发功能 */
   uint32_t  ABOM;            /* 是否使能 ABOM 自动离线管理功能 */
   uint32_t  AWUM;            /* 是否使能 AWUM 自动唤醒功能 */
   uint32_t  NART;            /* 是否使能 NART 自动重传功能 */
```

(下页继续)

(续上页)

```
  uint32_t   RFLM;            /* 是否使能 RFLM 锁定 FIFO 功能 */
  uint32_t   TXFP;            /* 配置 TXFP 报文优先级的判定方法 */
} CAN_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，这些结构体成员都是“39.2.11CAN 控制内核”小节介绍的内容，可对比阅读：

(1) Prescaler

本成员设置 CAN 外设的时钟分频，它可控制时间片 Tq 的时间长度，这里设置的值最终会减 1 后再写入 BRP 寄存器位，即前面介绍的 Tq 计算公式：

$Tq = (BRP[9:0]+1) \times T_{PCLK}$

等效于：$Tq = CAN_Prescaler \times T_{PCLK}$

(2) Mode

本成员设置 CAN 的工作模式，可设置为正常模式 (CAN_MODE_NORMAL)、回环模式 (CAN_MODE_LOOPBACK)、静默模式 (CAN_MODE_SILENT) 以及回环静默模式 (CAN_MODE_SILENT_LOOPBACK)。

(3) SJW

本成员可以配置 SJW 的极限长度，即 CAN 重新同步时单次可增加或缩短的最大长度，它可以被配置为 1-4Tq(CAN_SJW_1/2/3/4tq)。

(4) BS1

本成员用于设置 CAN 位时序中的 BS1 段的长度，它可以被配置为 1-16 个 Tq 长度 (CAN_BS1_1/2/3…16tq)。

(5) BS2

本成员用于设置 CAN 位时序中的 BS2 段的长度，它可以被配置为 1-8 个 Tq 长度 (CAN_BS2_1/2/3…8tq)。

SYNC_SEG、BS1 段及 BS2 段的长度加起来即一个数据位的长度，即前面介绍的原来计算公式：

T_{1bit} =1Tq+T_{S1}+T_{S2} =1+ (TS1[3:0] + 1)+ (TS2[2:0] + 1)

等效于：T_{1bit} = 1Tq+CAN_BS1+CAN_BS2

(6) TTCM

本成员用于设置是否使用时间触发功能 (ENABLE/DISABLE)，时间触发功能在某些 CAN 标准中会使用到。

(7) ABOM

本成员用于设置是否使用自动离线管理 (ENABLE/DISABLE)，使用自动离线管理可以在节点出错离线后适时自动恢复，不需要软件干预。

(8) AWUM

本成员用于设置是否使用自动唤醒功能 (ENABLE/DISABLE)，使能自动唤醒功能后它会在监测到总线活动后自动唤醒。

(9) ABOM

本成员用于设置是否使用自动离线管理功能 (ENABLE/DISABLE)，使用自动离线管理可以在出错时离线后适时自动恢复，不需要软件干预。

(10) NART

本成员用于设置是否使用自动重传功能 (ENABLE/DISABLE)，使用自动重传功能时，会一直发送报文直到成功为止。

(11) RFLM

本成员用于设置是否使用锁定接收 FIFO(ENABLE/DISABLE)，锁定接收 FIFO 后，若 FIFO 溢出时会丢弃新数据，否则在 FIFO 溢出时以新数据覆盖旧数据。

(12) TXFP

本成员用于设置发送报文的优先级判定方法 (ENABLE/DISABLE)，使能时，以报文存入发送邮箱的先后顺序来发送，否则按照报文 ID 的优先级来发送。

配置完这些结构体成员后，我们调用库函数 HAL_CAN_Init 即可把这些参数写入到 CAN 控制寄存器中，实现 CAN 的初始化。

40.4 CAN 发送及接收结构体

在发送或接收报文时，需要往发送邮箱中写入报文信息或从接收 FIFO 中读取报文信息，利用 STM32HAL 库的发送及接收结构体可以方便地完成这样的工作，它们的定义见*代码清单 39_0_2*。

列表 2: 代码清单 39-2 CAN 发送及接收结构体

```
/**
  * @brief  CAN Tx message structure definition
  * 发送结构体
  */
typedef struct {
  uint32_t StdId;  /* 存储报文的标准标识符 11 位, 0-0x7FF. */
  uint32_t ExtId;  /* 存储报文的扩展标识符 29 位, 0-0x1FFFFFFF. */
  uint8_t IDE;     /* 存储 IDE 扩展标志 */
  uint8_t RTR;     /* 存储 RTR 远程帧标志 */
  uint8_t DLC;     /* 存储报文数据段的长度, 0-8 */
  uint8_t Data[8]; /* 存储报文数据段的内容 */
} CanTxMsgTypeDef;

/**
  * @brief  CAN Rx message structure definition
  * 接收结构体
  */
typedef struct {
  uint32_t StdId;  /* 存储了报文的标准标识符 11 位, 0-0x7FF. */
  uint32_t ExtId;  /* 存储了报文的扩展标识符 29 位, 0-0x1FFFFFFF. */
  uint8_t IDE;     /* 存储了 IDE 扩展标志 */
  uint8_t RTR;     /* 存储了 RTR 远程帧标志 */
  uint8_t DLC;     /* 存储了报文数据段的长度, 0-8 */
  uint8_t Data[8]; /* 存储了报文数据段的内容 */
  uint8_t FMI;     /* 存储了 本报文是由经过筛选器存储进 FIFO 的, 0-0xFF */
  uint8_t FIFONumber; /* 配置接收 FIFO 编号, 可以是 CAN_FIFO0 或者 CAN_FIFO1 */
```

(下页继续)

(续上页)

```
} CanRxMsgTypeDef;
```

这些结构体成员，说明如下：

(1) *StdId*

本成员存储的是报文的 11 位标准标识符，范围是 0-0x7FF。

(2) ExtId

本成员存储的是报文的 29 位扩展标识符，范围是 0-0x1FFFFFFF。ExtId 与 StdId 这两个成员根据下面的 IDE 位配置，只有一个是有效的。

(3) IDE

本成员存储的是扩展标志 IDE 位，当它的值为宏 CAN_ID_STD 时表示本报文是标准帧，使用 StdId 成员存储报文 ID；当它的值为宏 CAN_ID_EXT 时表示本报文是扩展帧，使用 ExtId 成员存储报文 ID。

(4) RTR

本成员存储的是报文类型标志 RTR 位，当它的值为宏 CAN_RTR_Data 时表示本报文是数据帧；当它的值为宏 CAN_RTR_Remote 时表示本报文是遥控帧，由于遥控帧没有数据段，所以当报文是遥控帧时，下面的 Data[8] 成员的内容是无效的。

(5) DLC

本成员存储的是数据帧数据段的长度，它的值的范围是 0-8，当报文是遥控帧时 DLC 值为 0。

(6) Data[8]

本成员存储的就是数据帧中数据段的数据。

(7) FMI

本成员只存在于接收结构体，它存储了筛选器的编号，表示本报文是经过哪个筛选器存储进接收 FIFO 的，可以用它简化软件处理。

当需要使用 CAN 发送报文时，先定义一个上面发送类型的结构体，然后把报文的内容按成员赋值到该结构体中，最后调用库函数 CAN_Transmit 把这些内容写入到发送邮箱即可把报文发送出去。

接收报文时，通过检测标志位获知接收 FIFO 的状态，若收到报文，可调用库函数 CAN_Receive 把接收 FIFO 中的内容读取到预先定义的接收类型结构体中，然后再访问该结构体即可利用报文了。

40.5 CAN 筛选器结构体

CAN 的筛选器有多种工作模式，利用筛选器结构体可方便配置，它的定义见*代码清单 39_0_3*。

列表 3: 代码清单 40-3 CAN 筛选器结构体

```
/**
  * @brief  CAN filter init structure definition
  * CAN 筛选器结构体
  */
typedef struct {
  uint32_t FilterIdHigh;          /*CAN_FxR1 寄存器的高 16 位 */
  uint32_t FilterIdLow;           /*CAN_FxR1 寄存器的低 16 位 */
  uint32_t FilterMaskIdHigh;      /*CAN_FxR2 寄存器的高 16 位 */
  uint32_t FilterMaskIdLow;       /*CAN_FxR2 寄存器的低 16 位 */
  uint32_t FilterFIFOAssignment; /* 设置经过筛选后数据存储到哪个接收 FIFO */
  uint32_t FilterNumber;          /* 筛选器编号，范围 0-27*/
  uint32_t FilterMode;            /* 筛选器模式 */
  uint32_t FilterScale;           /* 设置筛选器的尺度 */
  uint32_t FilterActivation;        /* 是否使能本筛选器 */
  uint32_t BankNumber;       /* 扇区序号 */
} CAN_FilterInitTypeDef;
```

这些结构体成员都是“41.2.14 验收筛选器”小节介绍的内容，可对比阅读，各个结构体成员的介绍如下：

(1) FilterIdHigh

FilterIdHigh 成员用于存储要筛选的 ID，若筛选器工作在 32 位模式，它存储的是所筛选 ID 的高 16 位；若筛选器工作在 16 位模式，它存储的就是一个完整的要筛选的 ID。

(2) FilterIdLow

类似地，FilterIdLow 成员也是用于存储要筛选的 ID，若筛选器工作在 32 位模式，它存储的是所筛选 ID 的低 16 位；若筛选器工作在 16 位模式，它存储的就是一个完整的要筛选的 ID。

(3) FilterMaskIdHigh

FilterMaskIdHigh 存储的内容分两种情况，当筛选器工作在标识符列表模式时，它的功能与 FilterIdHigh 相同，都是存储要筛选的 ID；而当筛选器工作在掩码模式时，它存储的是 FilterIdHigh 成员对应的掩码，与 FilterIdLow 组成一组筛选器。

(4) FilterMaskIdLow

类似地，FilterMaskIdLow 存储的内容也分两种情况，当筛选器工作在标识符列表模式时，它的功能与 FilterIdLow 相同，都是存储要筛选的 ID；而当筛选器工作在掩码模式时，它存储的是 FilterIdLow 成员对应的掩码，与 FilterIdLow 组成一组筛选器。

上面四个结构体的存储的内容很容易让人糊涂，请结合前面的图 *39_0_15* 和下面的表 39-7 理解，如果还搞不清楚，再结合库函数 FilterInit 的源码来分析。

表 39-7 不同模式下各结构体成员的内容

模式	FilterIdHigh	FilterIdLow	FilterMaskIdHigh	FilterMaskIdLow
32 位列表模式	ID1 的高 16 位	ID1 的低 16 位	ID2 的高 16 位	ID2 的低 16 位
16 位列表模式	ID1 的完整数值	ID2 的完整数值	ID3 的完整数值	ID4 的完整数值
32 位掩码模式	ID1 的高 16 位	ID1 的低 16 位	ID1 掩码的高 16 位	ID1 掩码的低 16 位
16 位掩码模式	ID1 的完整数值	ID2 的完整数值	ID1 掩码的完整数值	ID2 掩码完整数值

对这些结构体成员赋值的时候，还要注意寄存器位的映射，即注意哪部分代表 STID，

哪部分代表 EXID 以及 IDE、RTR 位。

(5) FilterFIFOAssignment

本成员用于设置当报文通过筛选器的匹配后，该报文会被存储到哪一个接收 FIFO，它的可选值为 FIFO0 或 FIFO1(宏 CAN_FILTER_FIFO0/1)。

(6) FilterNumber

本成员用于设置筛选器的编号，即本过滤器结构体配置的是哪一组筛选器，CAN 一共有 28 个筛选器，所以它的可输入参数范围为 0-27。

(7) FilterMode

本成员用于设置筛选器的工作模式，可以设置为列表模式 (宏 CAN_FILTERMODE_IDLIST) 及掩码模式 (宏 CAN_FILTERMODE_IDMASK)。

(8) FilterScale

本成员用于设置筛选器的尺度，可以设置为 32 位长 (宏 CAN_FILTERSCALE_32BIT) 及 16 位长 (宏 CAN_FILTERSCALE_16BIT)。

(9) FilterActivation

本成员用于设置是否激活这个筛选器 (宏 ENABLE/DISABLE)。

(10) BankNumber

本成员用于设置选择启动从设备的扇区滤波器，可以输入参数范围为 0-28，该设置只有 CAN2 适用。

配置完这些结构体成员后，我们调用库函数 HAL_CAN_ConfigFilter 即可把这些参数写入到筛选控制寄存器中，从而使用筛选器。我们前面说如果不理解那几个 ID 结构体成员存储的内容时，可以直接阅读库函数 HAL_CAN_ConfigFilter 的源代码理解，就是因为它直接对寄存器写入内容，代码的逻辑是非常清晰的。

40.6 CAN—双机通讯实验

本小节演示如何使用 STM32 的 CAN 外设实现两个设备之间的通讯，该实验中使用了两个实验板，如果您只有一个实验板，也可以使用 CAN 的回环模式进行测试，不影响学习的。为此，我们提供了“CAN—双机通讯”及“CAN—回环测试”两个工程，可根据自己的实验环境选择相应的工程来学习。这两个工程的主体都是一样的，本教程主要以“CAN—双机通讯”工程进行讲解。

40.6.1 硬件设计

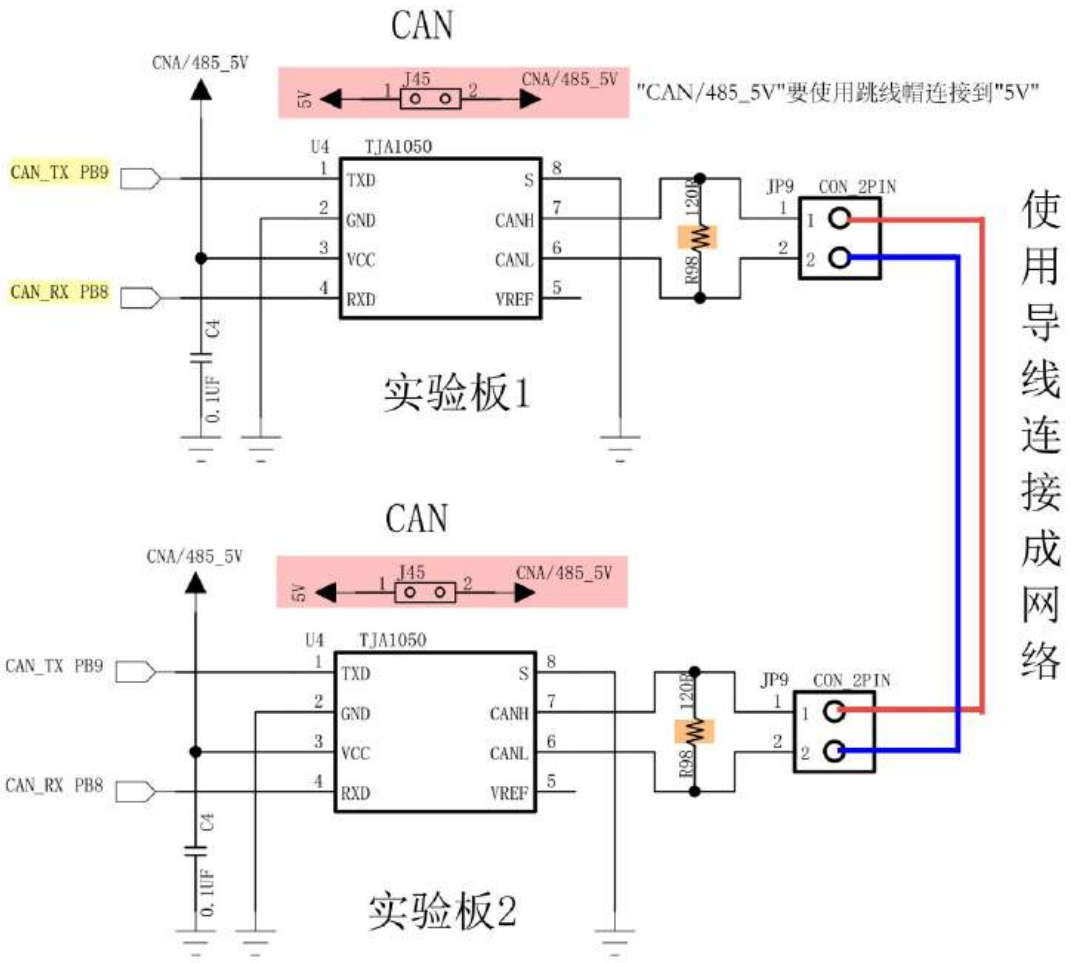


图 39-16 双 CAN 通讯实验硬件连接图

图 *39_0_16* 中的是两个实验板的硬件连接。在单个实验板中，作为 CAN 控制器的 STM32 引出 CAN_Tx 和 CAN_Rx 两个引脚与 CAN 收发器 TJA1050 相连，收发器使用 CANH 及 CANL 引脚连接到 CAN 总线网络中。为了方便使用，我们每个实验板引出的 CANH 及 CANL 都连接了 1 个 120 欧的电阻作为 CAN 总线的端电阻，所以要注意如果您要把实验板作为一个普通节点连接到现有的 CAN 总线时，是不应添加该电阻的！

要实现通讯，我们还要使用导线把实验板引出的 CANH 及 CANL 两条总线连接起来，才能构成完整的网络。实验板之间 CANH1 与 CANH2 连接，CANL1 与 CANL2 连接即可。

要注意的是，由于我们的实验板 CAN 使用的信号线与摄像头共用了，为防止干扰，平时我们默认是不给 CAN 收发器供电的，使用 CAN 的时候一定要把 CAN 接线端子旁边的“C/4-5V”排针使用跳线帽与“5V”排针连接起来进行供电，并且把摄像头从板子上拔下来。

如果您使用的是单机回环测试的工程实验，就不需要使用导线连接板子了，而且也不需要给收发器供电，因为回环模式的信号是不经过收发器的。

40.6.2 软件设计

为了使工程更加有条理，我们把 CAN 控制器相关的代码独立分开存储，方便以后移植。在“串口实验”之上新建“bsp_can.c”及“bsp_can.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。

40.6.2.1 编程要点

(1) 初始化 CAN 通讯使用的目标引脚及端口时钟；

(2) 使能 CAN 外设的时钟；

(3) 配置 CAN 外设的工作模式、位时序以及波特率；

(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。

40.6.2.2 代码分析

40.6.2.2.1 CAN 硬件相关宏定义

我们把 CAN 硬件相关的配置都以宏的形式定义到“bsp_can.h”文件中，见*代码清单 39_0_4*。

列表 4: 代码清单 39-3 CAN 硬件配置相关的宏 (bsp_can.h 文件)

```
#define CANx CAN1
#define CAN_CLK_ENABLE() __HAL_RCC_CAN1_CLK_ENABLE()
#define CAN_RX_IRQ CAN1_RX0_IRQn
#define CAN_RX_IRQHandler CAN1_RX0_IRQHandler

#define CAN_RX_PIN GPIO_PIN_8
#define CAN_TX_PIN GPIO_PIN_9
#define CAN_TX_GPIO_PORT GPIOB
#define CAN_RX_GPIO_PORT GPIOB

#define CAN_TX_GPIO_CLK_ENABLE() __HAL_RCC_GPIOB_CLK_ENABLE()
#define CAN_RX_GPIO_CLK_ENABLE() __HAL_RCC_GPIOB_CLK_ENABLE()
```

以上代码根据硬件连接，把与 CAN 通讯使用的 CAN 号、引脚号以及时钟都以宏封装起来，并且定义了接收中断的中断向量和中断服务函数，我们通过中断来获知接收 FIFO 的信息。注意在 GPIO 时钟部分我们还加入了 AFIO 时钟，这是为下面 CAN 进行复用功能重映射而设置的，当使用复用功能重映射时，必须开启 AFIO 时钟。

40.6.2.2.2 初始化 CAN 的 GPIO

利用上面的宏，编写 CAN 的初始化函数，见*代码清单 39_0_41*。

列表 5: 代码清单 39-4 CAN 的 GPIO 初始化函数
(bsp_can.c 文件)

```
static void CAN_GPIO_Config(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;

   /* 使能引脚时钟 */
   CAN_TX_GPIO_CLK_ENABLE();
   CAN_RX_GPIO_CLK_ENABLE();

   /* 配置 CAN 发送引脚 */
   GPIO_InitStructure.Pin = CAN_TX_PIN;
   GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
   GPIO_InitStructure.Speed = GPIO_SPEED_FAST;
   GPIO_InitStructure.Pull  = GPIO_PULLUP;
   GPIO_InitStructure.Alternate =  GPIO_AF9_CAN1;
   HAL_GPIO_Init(CAN_TX_GPIO_PORT, &GPIO_InitStructure);

   /* 配置 CAN 接收引脚 */
   GPIO_InitStructure.Pin = CAN_RX_PIN ;
   HAL_GPIO_Init(CAN_RX_GPIO_PORT, &GPIO_InitStructure);
};
```

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，CAN 的两个引脚都配置成通用推挽输出模式即可。

40.6.2.2.3 配置 CAN 的工作模式

接下来我们配置 CAN 的工作模式，由于我们是自己用的两个板子之间进行通讯，波特率之类的配置只要两个板子一致即可。如果您要使实验板与某个 CAN 总线网络的通讯的节点通讯，那么实验板的 CAN 配置必须要与该总线一致。我们实验中使用的配置见*代码清单 39_0_5*。

列表 6: 代码清单 39-5 配置 CAN 的工作模式 (bsp_can.c 文件)

```
static void CAN_Mode_Config(void)
{
    /*********************CAN 通信参数设置 ******************/
    /* 使能 CAN 时钟 */
    CAN_CLK_ENABLE();
    Can_Handle.Instance = CANx;
    Can_Handle.pTxMsg = &TxMessage;
    Can_Handle.pRxMsg = &RxMessage;
    /* CAN 单元初始化 */
    //MCR-TTCM  关闭时间触发通信模式使能
    Can_Handle.Init.TTCM=DISABLE;
    //MCR-ABOM  自动离线管理
    Can_Handle.Init.ABOM=ENABLE;
    //MCR-AWUM  使用自动唤醒模式
    Can_Handle.Init.AWUM=ENABLE;
    //MCR-NART  禁止报文自动重传    DISABLE-自动重传
    Can_Handle.Init.NART=DISABLE;
    //MCR-RFLM  接收 FIFO 锁定模式  DISABLE-溢出时新报文会覆盖原有报文
    Can_Handle.Init.RFLM=DISABLE;
    //MCR-TXFP  发送 FIFO 优先级 DISABLE-优先级取决于报文标示符
    Can_Handle.Init.TXFP=DISABLE;
    //回环模式
    Can_Handle.Init.Mode = CAN_MODE_LOOPBACK;
    //BTR-SJW 重新同步跳跃宽度 2 个时间单元
```

(下页继续)

(续上页)

```
    Can_Handle.Init.SJW=CAN_SJW_1TQ;

    /* ss=1 bs1=5 bs2=3 位时间宽度为 (1+5+3)
          波特率即为时钟周期 tq*(1+3+6)  */
    //BTR-TS1 时间段 1 占用了 6 个时间单元
    Can_Handle.Init.BS1=CAN_BS1_5TQ;
    //BTR-TS1 时间段 2 占用了 3 个时间单元
    Can_Handle.Init.BS2=CAN_BS2_3TQ;

    /* CAN Baudrate = 1 MBps (1MBps 已为 stm32 的 CAN 最高速率) (CAN
                      时钟频率为 APB 1 = 54 MHz) */
    Can_Handle.Init.Prescaler =6; //BTR-BRP 波特率分频器定义了时间单元的时间长度␣
↪54/(1+5+3)/5=1 Mbps
    HAL_CAN_Init(&Can_Handle);
}
```

这段代码主要是把 CAN 的模式设置成了正常工作模式，如果您阅读的是“CAN—回环测试”的工程，这里是被配置成回环模式的，除此之外，两个工程就没有其它差别了。

代码中还把位时序中的 BS1 和 BS2 段分别设置成了 4Tq 和 2Tq，再加上 SYNC_SEG 段，一个 CAN 数据位就是 9Tq 了，加上 CAN 外设的分频配置为 6 分频，CAN 所使用的总线时钟 f_{APB1} = 42MHz，于是我们可计算出它的波特率：

1Tq = 1/(42M/6)=1/7 us

T_{1bit}=(4+2+1) x Tq =1us

波特率 =1/T_{1bit} =1Mbps

40.6.2.2.4 配置筛选器

以上是配置 CAN 的工作模式，为了方便管理接收报文，我们还要把筛选器用起来，见*代码清单 39_0_6*。

列表 7: 代码清单 39-6 配置 CAN 的筛选器 (bsp_can.c 文件)

```
static void CAN_Filter_Config(void)
{
   CAN_FilterConfTypeDef  CAN_FilterInitStructure;

   /*CAN 筛选器初始化 */
   CAN_FilterInitStructure.FilterNumber=0; //筛选器组 0
   //工作在掩码模式
   CAN_FilterInitStructure.FilterMode=CAN_FILTERMODE_IDMASK;
   //筛选器位宽为单个 32 位。
   CAN_FilterInitStructure.FilterScale=CAN_FILTERSCALE_32BIT;
   /*
   使能筛选器，按照标志的内容进行比对筛选，扩展 I
   D 不是如下的就抛弃掉，是的话，会存入 FIFO0。 */
   //要筛选的 ID 高位
   CAN_FilterInitStructure.FilterIdHigh= ((((uint32_t)0x1314<<3)
                                          |CAN_ID_EXT|CAN_RTR_DATA)
                                          &0xFFFF0000)>>16;
   //要筛选的 ID 低位
   CAN_FilterInitStructure.FilterIdLow= (((uint32_t)0x1314<<3)
                                          |CAN_ID_EXT|CAN_RTR_DATA)
                                          &0xFFFF;
   //筛选器高 16 位每位必须匹配
   CAN_FilterInitStructure.FilterMaskIdHigh= 0xFFFF;
   //筛选器低 16 位每位必须匹配
   CAN_FilterInitStructure.FilterMaskIdLow= 0xFFFF;
```

(下页继续)

(续上页)

```
    //筛选器被关联到 FIFO0
    CAN_FilterInitStructure.FilterFIFOAssignment=CAN_FILTER_FIFO0 ;
    //使能筛选器
    CAN_FilterInitStructure.FilterActivation=ENABLE;
    HAL_CAN_ConfigFilter(&Can_Handle,&CAN_FilterInitStructure);
}
```

这段代码把筛选器第 0 组配置成了 32 位的掩码模式，并且把它的输出连接到接收 FIFO0，若通过了筛选器的匹配，报文会被存储到接收 FIFO0。

筛选器配置的重点是配置 ID 和掩码，根据我们的配置，这个筛选器工作在图 *39_0_17* 中的模式。

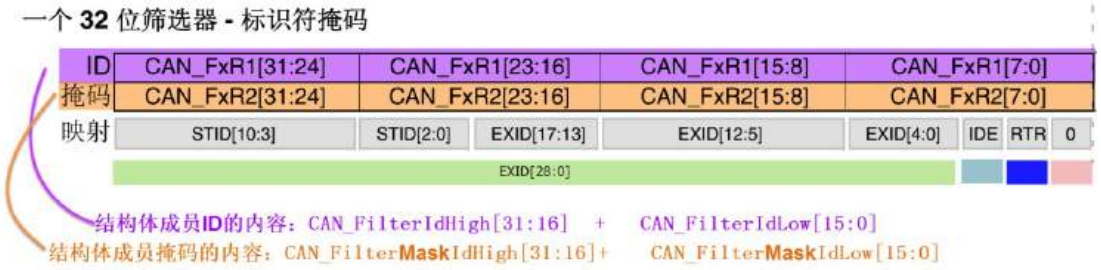


图 39-17 一个 32 位的掩码模式筛选器

在该配置中，结构体成员 FilterIdHigh 和 FilterIdLow 存储的是要筛选的 ID，而 FilterMaskIdHigh 和 FilterMaskIdLow 存储的是相应的掩码。在赋值时，要注意寄存器位的映射，在 32 位的 ID 中，第 0 位是保留位，第 1 位是 RTR 标志，第 2 位是 IDE 标志，从第 3 位起才是报文的 ID(扩展 ID)。

因此在上述代码中我们先把扩展 ID “0x1314”、IDE 位标志“宏 CAN_ID_EXT”以及 RTR 位标志“宏 CAN_RTR_DATA”根据寄存器位映射组成一个 32 位的数据，然后再把它的高 16 位和低 16 位分别赋值给结构体成员 FilterIdHigh 和 FilterIdLow。

而在掩码部分，为简单起见我们直接对所有位赋值为 1，表示上述所有标志都完全一样的报文才能经过筛选，所以我们这个配置相当于单个 ID 列表的模式，只筛选了一个 ID 号，而不是筛选一组 ID 号。这里只是为了演示方便，实际使用中一般会对不要求相等的数据位赋值为 0，从而过滤一组 ID，如果有需要，还可以继续配置多个筛选器组，最多可以配置 28 个，代码中只是配置了筛选器组 0。

对结构体赋值完毕后调用库函数 HAL_CAN_ConfigFilter 把个筛选器组的参数写入到寄存器中。

40.6.2.2.5 配置接收中断

当 FIFO0 接收到数据时会引起中断，该接收中断的优先级配置如下，见*代码清单 39_0_7*。

列表 8: 代码清单 39-7 配置 CAN 接收中断的优先级
(bsp_can.c 文件)

```
/*
* 函数名: CAN_NVIC_Config
* 描述  : CAN 的 NVIC 配置, 第 1 优先级组, 0, 0 优先级
* 输入  : 无
* 输出  : 无
* 调用  : 内部调用
*/
static void CAN_NVIC_Config(void)
{
   /* 配置抢占优先级的分组 */
   HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_1);
   /* 中断设置, 抢占优先级 0, 子优先级为 0*/
   HAL_NVIC_SetPriority(CAN_RX_IRQ, 0 ,0);
   HAL_NVIC_EnableIRQ(CAN_RX_IRQ);
}
```

这部分与我们配置其它中断的优先级无异，都是配置 NVIC 结构体，优先级可根据自己的需要配置，最主要的是中断向量，上述代码中把中断向量配置成了 CAN 的接收中断。

40.6.2.2.6 设置发送报文

要使用 CAN 发送报文时，我们需要先定义一个发送报文结构体并向它赋值，见*代码清单 39_0_8*。

列表 9: 代码清单 39-8 设置要发送的报文 (bsp_can.c 文件)

```
/*
* 函数名: CAN_SetMsg
* 描述  : CAN 通信报文内容设置, 设置一个数据内容为 0-7 的数据包
* 输入  : 发送报文结构体
* 输出  : 无
* 调用  : 外部调用
*/
void CAN_SetMsg(void)
{
   uint8_t ubCounter = 0;
   Can_Handle.pTxMsg->StdId=0x00;
   Can_Handle.pTxMsg->ExtId=0x1314;            //使用的扩展 ID
   Can_Handle.pTxMsg->IDE=CAN_ID_EXT;          //扩展模式
   Can_Handle.pTxMsg->RTR=CAN_RTR_DATA;        //发送的是数据
   Can_Handle.pTxMsg->DLC=8;                 //数据长度为 8 字节

   /* 设置要发送的数据 0-7*/
   for (ubCounter = 0; ubCounter < 8; ubCounter++) {
      Can_Handle.pTxMsg->Data[ubCounter] = ubCounter;
   }
}
```

这段代码是我们为了方便演示而自己定义的设置报文内容的函数，它把报文设置成了扩展模式的数据帧，扩展 ID 为 0x1314，数据段的长度为 8，且数据内容分别为 0-7，实际应用中您可根据自己的需求发设置报文内容。当我们设置好报文内容后，调用库函数 HAL_CAN_Transmit_IT 即可把该报文存储到发送邮箱，然后 CAN 外设会把它发送出去。

40.6.2.2.7 接收报文

由于我们设置了接收中断，所以接收报文的操作是在中断的服务函数中完成的，见*代码清单 39_0_9*。

列表 10: 代码清单 39-9 接收报文 (stm32f1xx_it.c)

```
/**
* @brief  CAN 接收完成中断 (非阻塞)
* @param  hcan: CAN 句柄指针
* @retval 无
*/
void HAL_CAN_RxCpltCallback(CAN_HandleTypeDef* hcan)
{
   /* 比较 ID 是否为 0x1314 */
   if ((hcan->pRxMsg->ExtId==0x1314) && (hcan->pRxMsg->IDE==CAN_ID_EXT)
   && (hcan->pRxMsg->DLC==8) ) {     flag = 1; //接收成功
   } else {
      flag = 0; //接收失败
   }
   /* 准备中断接收 */
   HAL_CAN_Receive_IT(&Can_Handle, CAN_FIFO0);
}
```

根据我们前面的配置，若 CAN 接收的报文经过筛选器匹配后会被存储到 FIFO0 中，并引起中断进入到这个中断服务函数中，在这个函数里我们调用了库函数 HAL_CAN_Receive_IT 把报文从 FIFO 复制到接收报文结构体 CanRxMsgTypeDef 中，并且比较了接收到的报文 ID 是否与我们希望接收的一致，若一致就设置标志 flag=1，否则为 0，通过 flag 标志通知主程序流程获知是否接收到数据。

要注意如果设置了接收报文中断，必须要在中断内调用 HAL_CAN_Receive_IT 函数读取接收 FIFO 的内容，因为只有这样才能清除该 FIFO 的接收中断标志，如果不在中断内调用它清除标志的话，一旦接收到报文，STM32 会不断进入中断服务函数，导致程序卡死。

40.6.2.3 main 函数

最后我们来阅读 main 函数，了解整个通讯流程，见*代码清单 39_0_10*。

列表 11: 代码清单 39-10 main 函数

```
int main(void)
{
   /* 配置系统时钟为 168 MHz */
   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 can，在中断接收 CAN 数据包 */
   CAN_Config();

   printf("\r\n 欢迎使用野火  STM32 F407 开发板。\r\n");
   printf("\r\n 野火 F407 CAN 通讯实验例程\r\n");
   printf("\r\n 实验步骤: \r\n");

   printf("\r\n 1. 使用导线连接好两个 CAN 讯设备\r\n");
   printf("\r\n 2. 使用跳线帽连接好:5v --- C/4-5V \r\n");
   printf("\r\n 3. 按下开发板的 KEY1 键，会使用 CAN 向外发送 0-7 的数据包，包的扩展
ID 为 0x1314 \r\n");
   printf("\r\n 4. 若开发板的 CAN 接收到扩展 ID 为 0x1314 的数据包，会把数据以打印到
串口。 \r\n");
   printf("\r\n 5. 本例中的 can 波特率为 1MBps，为 stm32 的 can 最高速速率。 \r\n
↪");
   while (1) {
         /* 按一次按键发送一次数据 */
         if (  Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON) {
            LED_BLUE;
```

(下页继续)

(续上页)

```
            /* 装载一帧数据 */
            CAN_SetMsg();
            /* 开始发送数据 */
            HAL_CAN_Transmit_IT(&Can_Handle);
            HAL_Delay(100);
            LED_RGBOFF;
        }
        if (flag==1) {
            printf("\r\nCAN 接收到数据: \r\n");
            CAN_DEBUG_ARRAY(Can_Handle.pRxMsg->Data,8);
            flag=0;
            HAL_Delay(100);
            LED_GREEN;
        }
    }
}
```

在 main 函数里，我们调用了 CAN_Config 函数初始化 CAN 外设，它包含我们前面解说的 GPIO 初始化函数 CAN_GPIO_Config、中断优先级设置函数 CAN_NVIC_Config、工作模式设置函数 CAN_Mode_Config 以及筛选器配置函数 CAN_Filter_Config。

初始化完成后，我们在 while 循环里检测按键，当按下实验板的按键 1 时，它就调用 CAN_SetMsg 函数设置要发送的报文，然后调用 HAL_CAN_Transmit_IT 函数把该报文存储到发送邮箱，等待 CAN 外设把它发送出去。代码中并没有检测发送状态，如果需要，您可以调用库函数 HAL_CAN_GetState 检查发送状态。

while 循环中在其它时间一直检查 flag 标志，当接收到报文时，我们的中断服务函数会把它置 1，所以我们可以通过它获知接收状态，当接收到报文时，我们把它使用宏 CAN_DEBUG_ARRAY 输出到串口。

40.6.3 下载验证

下载验证这个 CAN 实验时，我们建议您先使用“CAN—回环测试”的工程进行测试，它的环境配置比较简单，只需要一个实验板，用 USB 线使实验板“USB TO UART”接口跟电脑连接起来，在电脑端打开串口调试助手，并且把编译好的该工程下载到实验板，然后复位。这时在串口调试助手可看到 CAN 测试的调试信息，按一下实验板上的 KEY1 按键，实验板会使用回环模式向自己发送报文，在串口调试助手可以看到相应的发送和接收的信息。

使用回环测试成功后，如果您有两个实验板，需要按照“硬件设计”小节中的图例连接两个板子的 CAN 总线，并且一定要接上跳线帽给 CAN 收发器供电、把摄像头拔掉防止干扰。用 USB 线使实验板“USB TO UART”接口跟电脑连接起来，在电脑端打开串口调试助手，然后使用“CAN—双机通讯”工程编译，并给两个板子都下载该程序，然后复位。这时在串口调试助手可看到 CAN 测试的调试信息，按一下其中一个实验板上的 KEY1 按键，另一个实验板会接收到报文，在串口调试助手可以看到相应的发送和接收的信息。

第 41 章　RS-485 通讯实验

本章参考资料：《STM32F4xx 中文参考手册》USART 章节。

学习本章时，配合本书前面的《USART—串口通讯》及《CAN—通讯实验》章节进行对比学习，效果更佳。

关于实验板中使用的 MAX485 收发器资料可查阅《MAX485》规格书了解。

41.1 RS-485 通讯协议简介

与 CAN 类似，RS-485 是一种工业控制环境中常用的通讯协议，它具有抗干扰能力强、传输距离远的特点。RS-485 通讯协议由 RS-232 协议改进而来，协议层不变，只是改进了物理层，因而保留了串口通讯协议应用简单的特点。

41.2 RS-485 的物理层

从《CAN—通讯实验》章节中了解到，差分信号线具有很强的干扰能力，特别适合应用于电磁环境复杂的工业控制环境中，RS-485 协议主要是把 RS-232 的信号改进成差分信号，从而大大提高了抗干扰特性，它的通讯网络示意图见图 *40_0_1*。

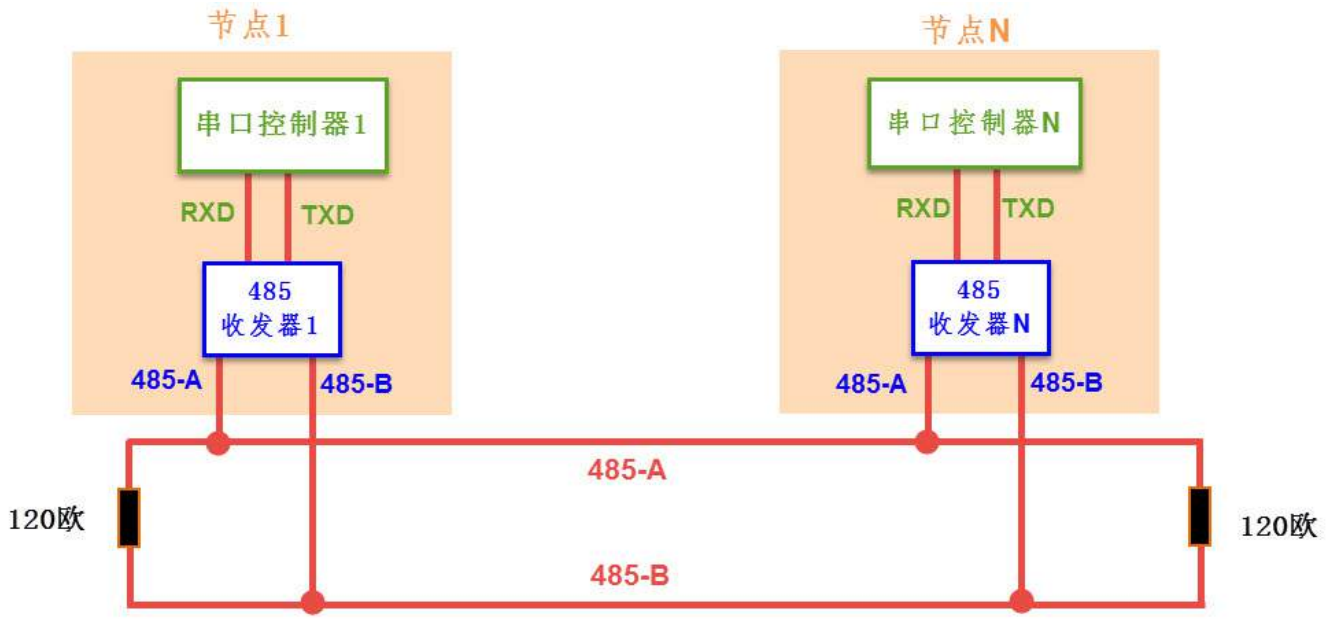


图 40-1 RS-485 通讯网络示意图

对比 CAN 通讯网络，可发现它们的网络结构组成是类似的，每个节点都是由一个通讯控制器和一个收发器组成，在 RS-485 通讯网络中，节点中的串口控制器使用 RX 与 TX 信号线连接到收发器上，而收发器通过差分线连接到网络总线，串口控制器与收发器之间一般使用 TTL 信号传输，收发器与总线则使用差分信号来传输。发送数据时，串口控制器的 TX 信号经过收发器转换成差分信号传输到总线上，而接收数据时，收发器把总线上的差分信号转化成 TTL 信号通过 RX 引脚传输到串口控制器中。

RS-485 通讯网络的最大传输距离可达 1200 米，总线上可挂载 128 个通讯节点，而由于 RS-485 网络只有一对差分信号线，它使用差分信号来表达逻辑，当 AB 两线间的电压差为-6V~-2V 时表示逻辑 1，当电压差为 +2V~+6V 表示逻辑 0，在同一时刻只能表达一个信号，所以它的通讯是半双工形式的，它与 RS-232 通讯协议的特性对比见图 40-1。

表 40-1 RS-232/422/485 标准对比

通讯标准	信号线	通讯方向	电平标准 (发送端)	通讯距离	通讯节点数
RS232	单端 TXD、RXD、GND	全双工	逻辑 1：-15V~-3V 逻辑 0：+3V~+15V	100 米以内	只有两个节点
RS485	差分线 AB	半双工	逻辑 1：+2V~+6V 逻辑 0：-6V~-2V	1200 米	支持多个节点。支持多个主设备，任意节点间可互相通讯

RS-485 与 RS-232 的差异只体现在物理层上，它们的协议层是相同的，也是使用串口数据包的形式传输数据。而由于 RS-485 具有强大的组网功能，人们在基础协议之上还制定了 MODBUS 协议，被广泛应用在工业控制网络中。此处说的基础协议是指前面串口章节中讲解的，仅封装了基本数据包格式的协议 (基于数据位)，而 MODBUS 协议是使用基本数据包组合成通讯帧格式的高层应用协议 (基于数据包或字节)。感兴趣的读者可查找 MODBUS 协议的相关资料了解。

由于 RS-485 与 RS-232 的协议层没有区别，进行通讯时，我们同样是使用 STM32 的 USART 外设作为通讯节点中的串口控制器，再外接一个 RS-485 收发器芯片把 USART 外设的 TTL 电平信号转化成 RS-485 的差分信号即可。

41.3 RS-485—双机通讯实验

本小节演示如何使用 STM32 的 USART 控制器与 MAX485 收发器，在两个设备之间使用 RS-485 协议进行通讯，本实验中使用了两个实验板，无法像 CAN 实验那样使用回环测试 (把 STM32 USART 外设的 TXD 引脚使用杜邦线连接到 RXD 引脚可进行自收发测试，不过这样的通讯不经过 RS-485 收发器，跟普通 TTL 串口实验没有区别)，本教程主要以 “USART—485 通讯” 工程进行讲解。

41.3.1 硬件设计

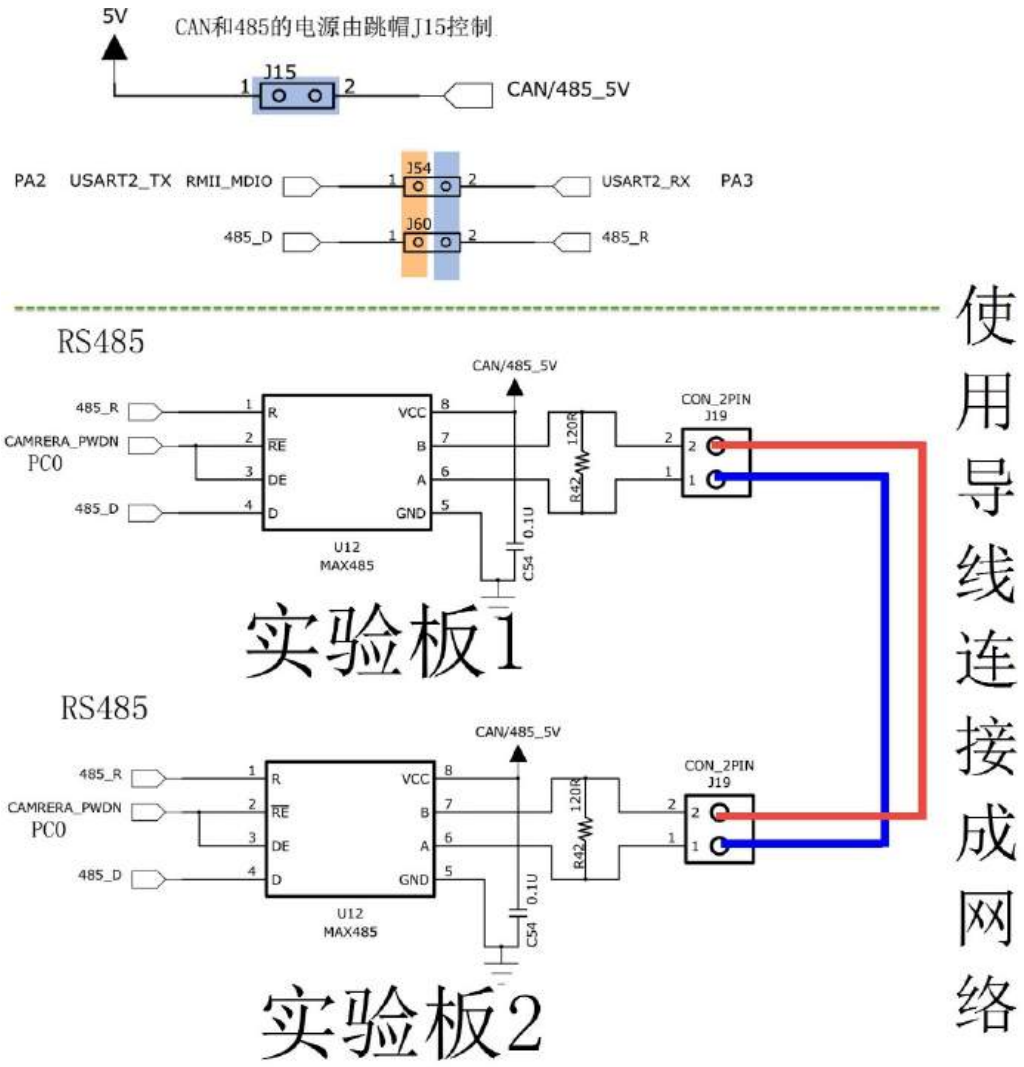


图 40-2 双 CAN 通讯实验硬件连接图

图 *40_0_2* 中的是两个实验板的硬件连接。在单个实验板中，作为串口控制器的 STM32 从 USART 外设引出 TX 和 RX 两个引脚与 RS-485 收发器 MAX485 相连，收发器使用它的 A 和 B 引脚连接到 RS-485 总线网络中。为了方便使用，我们每个实验板引出的 A 和 B 之间都连接了 1 个 120 欧的电阻作为 RS-485 总线的端电阻，所以要注意如果您要把实验板作为一个普通节点连接到现有的 RS-485 总线时，是不应添加该电阻的！

由于 485 只能以半双工的形式工作，所以需要切换状态，MAX485 芯片中有“RE”和“DE”两个引脚，用于控制 485 芯片的收发工作状态的，当 RE 引脚为低电平时，485 芯片处于接收状态，当 DE 引脚为高电平时芯片处于发送状态。实验板中使用了 STM32 的 PC0 直接连接到这两个引脚上，所以通过控制 PC0 的输出电平即可控制 485 的收发状态，在本开发板中，PC0 引脚与摄像头使用的引脚共用了，所以使用 485 时不要同时驱动摄像头。

还要注意的是，为防止干扰，平时我们默认是不给 485 收发器供电的，使用 485 的时候一定要把 485 接线端子旁边的“C/4-5V”排针使用跳线帽与“5V”排针连接起来进行供电；而又由于实验板的 RS-232 与 RS-485 通讯实验都使用 STM32 的同一个 USART 外设及收发引脚，实验时注意必须要把 STM32 的“PA2 引脚”与 MAX485 的“485_D”及“PA3”与“485_R”使用跳线帽连接起来 (这些信号都在 485 接线端子旁边的排针上)。

要实现通讯，我们还要使用导线把实验板引出的 A 和 B 两条总线连接起来，才能构成完整的网络。实验板之间 A 与 A 连接，B 与 B 连接即可。

41.3.2 软件设计

为了使工程更加有条理，我们把 RS485 控制相关的代码独立分开存储，方便以后移植。在“串口实验”之上新建“bsp_485.c”及“bsp_485.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。这个实验的底层 STM32 驱动与串口控制区别不大，上层实验功能上与 CAN 实验类似。

41.3.2.1 编程要点

(1) 初始化 485 通讯使用的 USART 外设及相关引脚；

(2) 编写控制 MAX485 芯片进行收发数据的函数；

(3) 编写测试程序，收发数据。

41.3.2.2 代码分析

41.3.2.2.1 485 硬件相关宏定义

我们把 485 硬件相关的配置都以宏的形式定义到“bsp_485.h”文件中，见代码清单 *40_0_1*。

列表 1: 代码清单 40-1 485 硬件配置相关的宏 (bsp_485.h 文件)

```
/*USART 号、时钟、波特率 */
#define _485_USART                              USART2
#define _485_USART_CLK_ENABLE()                 __USART2_CLK_ENABLE();
#define _485_USART_BAUDRATE                     115200

#define RCC_PERIPHCLK_485_USART                 RCC_PERIPHCLK_USART2
#define RCC_485_USARTCLKSOURCE_SYSCLK
RCC_USART2CLKSOURCE_SYSCLK
/*RX 引脚 */
#define _485_USART_RX_GPIO_PORT                 GPIOA
#define _485_USART_RX_GPIO_CLK_ENABLE()         __GPIOA_CLK_ENABLE()
#define _485_USART_RX_PIN                       GPIO_PIN_3
#define _485_USART_RX_AF                        GPIO_AF7_USART2
/*TX 引脚 */
#define _485_USART_TX_GPIO_PORT                 GPIOA
#define _485_USART_TX_GPIO_CLK_ENABLE()         __GPIOA_CLK_ENABLE()
#define _485_USART_TX_PIN                       GPIO_PIN_2
#define _485_USART_TX_AF                        GPIO_AF7_USART2
/*485 收发控制引脚 */
#define _485_RE_GPIO_PORT               GPIOC
#define _485_RE_GPIO_CLK_ENABLE()               __GPIOC_CLK_ENABLE()
#define _485_RE_PIN                 GPIO_PIN_0
/* 中断相关 */
#define _485_INT_IRQ                        USART2_IRQn
```

(下页继续)

(续上页)

```
#define bsp_485_IRQHandler                      USART2_IRQHandler
```

以上代码根据硬件连接，把与 485 通讯使用的 USART 外设号、引脚号、引脚源以及复用功能映射都以宏封装起来，并且定义了接收中断的中断向量和中断服务函数，我们通过中断来获知接收数据。

41.3.2.2.2 初始化 485 的 USART 配置

利用上面的宏，编写 485 的 USART 初始化函数，见*代码清单 40_0_2*。

列表 2: 代码清单 40-2 RS485 的初始化函数 (bsp_485.c 文件)

```
/*
* 函数名: _485_Config
* 描述  : USART GPIO 配置, 工作模式配置
* 输入  : 无
* 输出  : 无
* 调用  : 外部调用
*/
void _485_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    RCC_PeriphCLKInitTypeDef RCC_PeriphClkInit;

    _485_USART_RX_GPIO_CLK_ENABLE();
    _485_USART_TX_GPIO_CLK_ENABLE();
    _485_RE_GPIO_CLK_ENABLE();

    /* 配置 485 串口时钟源 */
```

(下页继续)

(续上页)

```
    RCC_PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_485_USART;
    RCC_PeriphClkInit.Usart2ClockSelection = RCC_485_USARTCLKSOURCE_SYSCLK;
    HAL_RCCEx_PeriphCLKConfig(&RCC_PeriphClkInit);

    /* 使能 UART 时钟 */
    _485_USART_CLK_ENABLE();

    /**USART2 GPIO Configuration
    PD5     ------> USART2_TX
    PD6     ------> USART2_RX
    */
    /* 配置 Tx 引脚为复用功能  */
    GPIO_InitStruct.Pin = _485_USART_TX_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    GPIO_InitStruct.Alternate = _485_USART_TX_AF;
    HAL_GPIO_Init(_485_USART_TX_GPIO_PORT, &GPIO_InitStruct);

    /* 配置 Rx 引脚为复用功能 */
    GPIO_InitStruct.Pin = _485_USART_RX_PIN;
    GPIO_InitStruct.Alternate = _485_USART_RX_AF;
    HAL_GPIO_Init(_485_USART_RX_GPIO_PORT, &GPIO_InitStruct);

    /* 485 收发控制管脚 */
    GPIO_InitStruct.Pin = _485_RE_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    HAL_GPIO_Init(_485_RE_GPIO_PORT, &GPIO_InitStruct);

    /* 配置串 485_USART 模式 */
```

(下页继续)

(续上页)

```
    Uart2_Handle.Instance = _485_USART;
    Uart2_Handle.Init.BaudRate = _485_USART_BAUDRATE;
    Uart2_Handle.Init.WordLength = UART_WORDLENGTH_8B;
    Uart2_Handle.Init.StopBits = UART_STOPBITS_1;
    Uart2_Handle.Init.Parity = UART_PARITY_NONE;
    Uart2_Handle.Init.Mode = UART_MODE_TX_RX;
    Uart2_Handle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    Uart2_Handle.Init.OverSampling = UART_OVERSAMPLING_16;
    Uart2_Handle.Init.OneBitSampling = UART_ONEBIT_SAMPLING_DISABLED;
    Uart2_Handle.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    HAL_UART_Init(&Uart2_Handle);

    /* 串口 1 中断初始化 */
    NVIC_Configuration();
    /* 配置串口接收中断 */
    __HAL_UART_ENABLE_IT(&Uart2_Handle,UART_IT_RXNE);
    //默认进入接收模式
    HAL_GPIO_WritePin(_485_RE_GPIO_PORT,_485_RE_PIN,GPIO_PIN_RESET);
}
```

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，其中用于控制 MAX485 芯片的收发状态的引脚被初始化成普通推挽输出模式，以便手动控制它的电平输出，切换状态。485 使用到的 USART 也需要配置好波特率、有效字长、停止位及校验位等基本参数，在通讯中，两个 485 节点的串口参数应一致，否则会导致通讯解包错误。在实验中还使能了串口的接收中断功能，当检测到新的数据时，进入中断服务函数中获取数据。

41.3.2.2.3 使用中断接收数据

接下来我们编写在 USART 中断服务函数中接收数据的相关过程，见*代码清单 40_0_3*，其中的 bsp_RS485_IRQHandler 函数直接被 bsp_stm32f4xx_it.c 文件的 USART 中断服务函数调用，不在此列出。

列表 3: 代码清单 40-3 中断接收数据的过程 (bsp_485.c 文件)

```
//中断缓存串口数据
#define UART_BUFF_SIZE      1024
volatile    uint16_t uart_p = 1;
uint8_t     uart_buff[UART_BUFF_SIZE];

void bsp_485_IRQHandler(void)
{
    if (uart_p<UART_BUFF_SIZE) {
        if (__HAL_UART_GET_IT( &Uart2_Handle, UART_IT_RXNE ) != RESET) {
            HAL_UART_Receive(&Uart2_Handle, (uint8_t *)(&uart_buff[uart_
↪p]),1 , 1000)
            uart_p++;
        }
    } else {
        clean_rebuff();
    }
    HAL_UART_IRQHandler(&Uart2_Handle);
}

//获取接收到的数据和长度
char *get_rebuff(uint16_t *len)
{
    *len = uart_p;
    return (char *)&uart_buff;
```

(下页继续)

(续上页)

```
}

//清空缓冲区
void clean_rebuff(void)
{

    uint16_t i=UART_BUFF_SIZE+1;
    uart_p = 0;
    while (i)
        uart_buff[--i]=0;
}
```

这个数据接收过程主要思路是使用了接收缓冲区，当 USART 有新的数据引起中断时，调用库函数 USART_ReceiveData 把新数据读取到缓冲区数组 uart_buff 中，其中 get_rebuff 函数可以用于获缓冲区中有效数据的长度，而 clean_rebuff 函数可以用于对缓冲区整体清 0，这些函数配合使用，实现了简单的串口接收缓冲机制。这部分串口数据接收的过程跟 485 收发器无关，是串口协议通用的。

41.3.2.2.4 切换收发状态

在前面我们了解到 RS-485 是半双工通讯协议，发送数据和接收数据需要分时进行，所以需要经常切换收发状态。而 MAX485 收发器根据其“RE”和“DE”引脚的外部电平信号切换收发状态，所以控制与其相连的 STM32 普通 IO 电平即可控制收尾，为简便起见，我们把收发状态切换定义成了宏，见*代码清单 40_0_4*。

列表 4: 代码清单 40-4 切换收发状态 (bsp_485.h 文件)

```
// 不精确的延时
static void _485_delay(__IO uint32_t nCount)
{
    for (; nCount != 0; nCount--);
```

(下页继续)

(续上页)

```
}
/* 控制收发引脚 */
//进入接收模式，必须要有延时等待 485 处理完数据
#define _485_RX_EN()       _485_delay(1000);\
HAL_GPIO_WritePin(_485_RE_GPIO_PORT,_485_RE_PIN,GPIO_PIN_RESET);
_485_delay(1000);
//进入发送模式，必须要有延时等待 485 处理完数据
#define _485_TX_EN()       _485_delay(1000);\
HAL_GPIO_WritePin(_485_RE_GPIO_PORT,_485_RE_PIN,GPIO_PIN_SET);
_485_delay(1000);
```

这两个宏中，主要是在控制电平输出前后加了一小段时间延时，这是为了给 MAX485 芯片预留响应时间，因为 STM32 的引脚状态电平变换后，MAX485 芯片可能存在响应延时。例如，当 STM32 控制自己的引脚电平输出高电平 (控制成发送状态)，然后立即通过 TX 信号线发送数据给 MAX485 芯片，而 MAX485 芯片由于状态不能马上切换，会导致丢失了部分 STM32 传送过来的数据，造成错误。

41.3.2.2.5 发送数据

STM32 使用 485 发送数据的过程也与普通的 USART 发送数据过程差不多，我们定义了一个 RS485_SendByte 函数来发送一个字节的数据内容，见*代码清单 40_0_5*。

列表 5: 代码清单 40-5 发送数据 (bsp_485.c 文件)

```
/****************** 发送一个字符  **********************/
//使用单字节数据发送前要使能发送引脚，发送后要使能接收引脚。
void _485_SendByte(  uint8_t ch )
{
    /* 发送一个字节数据到 USART1 */
    HAL_UART_Transmit(&Uart2_Handle, (uint8_t *)&ch, 1, 0xFFFF);
}
```

上述代码中就是直接调用了 STM32 库函数 HAL_UART_Transmit 把要发送的数据写入到 USART 的数据寄存器，然后检查标志位等待发送完成。

在调用 _485_SendByte 函数前，需要先使用前面提到的切换收发状态宏，把 MAX485 切换到发送模式，STM32 发出的数据才能正常传输到 485 网络总线上，当发送完数据的时候，应重新把 MAX485 切换回接收模式，以便获取网络总线上的数据。

41.3.2.3 main 函数

最后我们来阅读 main 函数，了解整个通讯过程，见*代码清单 40_0_6*。这个 main 函数的整体设计思路是，实验板检测自身的按键状态，若按键被按下，则通过 485 发送 256 个测试数据到网络总线上，若自身接收到总线上的 256 个数据，则把这些数据作为调试信息打印到电脑端。所以，如果把这样的程序分别应用到 485 总线上的两个通讯节点时，就可以通过按键控制互相发送数据了。

列表 6: 代码清单 40-6 main 函数

```
int main(void)
{
    char *pbuf;
    uint16_t len;
    /* 配置系统时钟为 168 MHz */
    SystemClock_Config();
    /* 初始化 RGB 彩灯 */
    LED_GPIO_Config();
    /* 初始化 USART1 配置模式为 115200 8-N-1 */
    DEBUG_USART_Config();
    /* 初始化 485 使用的串口，使用中断模式接收 */
    _485_Config();
    Key_GPIO_Config();
    printf("\r\n 欢迎使用野火  STM32 F407 开发板。\r\n");
    printf("\r\n 野火 F407 485 通讯实验例程\r\n");
    printf("\r\n 实验步骤：\r\n");
```

(下页继续)

(续上页)

```
    printf("\r\n 1. 使用导线连接好两个 485 通讯设备\r\n");
    printf("\r\n 2. 使用跳线帽连接好:5v --- C/4-5V,485-D --- PD5,485-R ---
↪PD6 \r\n");
    printf("\r\n 3. 若使用两个野火开发板进行实验，给两个开发板都下载本程序即可。\r\n
↪");
    printf("\r\n 4. 准备好后，按下其中一个开发板的 KEY1 键，会用 485 向外发送 0-255
的数字 \r\n");
    printf("\r\n 5. 若开发板的 485 接收到 256 个字节数据，会把数据以 16 进制形式打
印出来。 \r\n");

    while (1) {
        /* 按一次按键发送一次数据 */
        if (  Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON) {
            uint16_t i;

            LED_BLUE;

            _485_TX_EN();

            for (i=0; i<=0xff; i++) {
                _485_SendByte(i);   //发送数据
            }

            /* 加短暂延时，保证 485 发送数据完毕 */
            Delay(0xFFF);
            _485_RX_EN();

            LED_GREEN;

            printf("\r\n发送数据成功！\r\n"); //使用调试串口打印调试信息到终端
```

(下页继续)

(续上页)

```
        } else {
            LED_BLUE;

            pbuf = get_rebuff(&len);
            if (len>=256) {
                LED_GREEN;
                printf("\r\n接收到长度为%d 的数据\r\n",len);
                _485_DEBUG_ARRAY((uint8_t*)pbuf,len);
                clean_rebuff();
            }
        }
    }
}
```

在 main 函数中，首先初始化了 LED、按键以及调试使用的串口，再调用前面分析的 RS485_Config 函数初始化了 RS-485 通讯使用的串口工作模式。

初始化后 485 就进入了接收模式，当接收到数据的时候会进入中断并把数据存储到接收缓冲数组中，我们在 main 函数的 while 循环中 (else 部分) 调用 get_rebuff 来查看该缓冲区的状态，若接收到 256 个数据就把这些数据通过调试串口打印到电脑端，然后清空缓冲区。

在 while 循环中，还检测了按键的状态，若按键被按下，就把 MAX485 芯片切换到发送状态并调用 RS485_SendByte 函数发送测试数据 0x00-0xFF，发送完毕后切换回接收状态以检测总线的数据。

41.3.3 下载验证

下载验证这个 485 通讯实验需要您有两个实验板，操作步骤如下：

(1) 按照“硬件设计”小节中的图例连接两个板子的 485 总线；

(2) 使用跳线帽连接: 485_R<—>PA3、485_D<—>PA2、C/4-5V<—>5V；

(3) 用 USB 线使实验板“USB TO UART”接口跟电脑连接起来，在电脑端打开串口调试助手，

编译本章配套的程序，并给两个板子都下载该程序，然后复位。

(4) 复位后在串口调试助手应看到 485 测试的调试信息，按一下其中一个实验板上的 KEY1 按键，另一个实验板会接收到报文，在串口调试助手可以看到相应的发送和接收的信息。

第 42 章　电源管理—实现低功耗

本章参考数据：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库说明文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

42.1 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。

42.1.1 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。

42.1.1.1 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 *42_1*，在刚开始电压低于 VPOR 时 (约 1.72V)，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 1.68V)，会进入掉电复位状态 (PDR，Power Down Reset)。

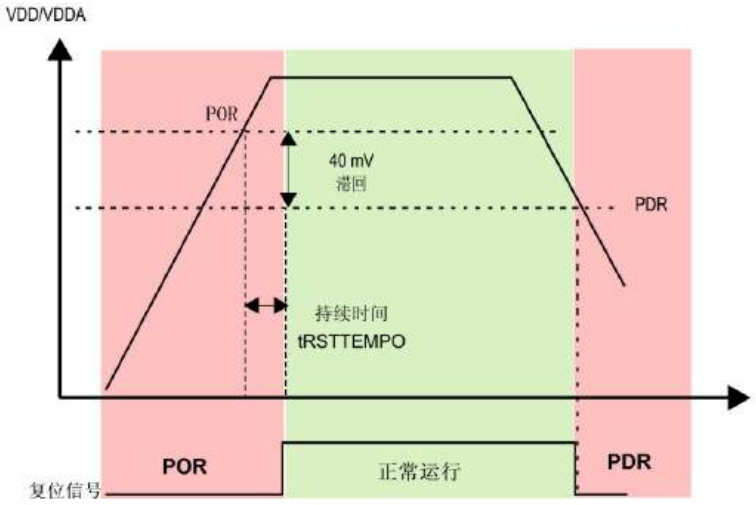


图 42-1 POR 与 PDR

42.1.1.2 欠压复位 (BOR)

POR 与 PDR 的复位电压阈值是固定的，如果用户想要自行设定复位阈值，可以使用 STM32 的 BOR 功能 (Brownout Reset)。它可以编程控制电压检测工作在表 42-1 中的阈值级别，通过修改“选项字节”(某些特殊寄存器) 中的 BOR_LEV 位即可控制阈值级别。其复位控制示意图见图 *42_2*。

表 42-42-1 BOR 欠压阈值等级

等级	条件	电压值
1 级欠压阈值	下降沿	2.19V
	上升沿	2.29V
2 级欠压阈值	下降沿	2.50V
	上升沿	2.59V
3 级欠压阈值	下降沿	2.83V
	上升沿	2.92V

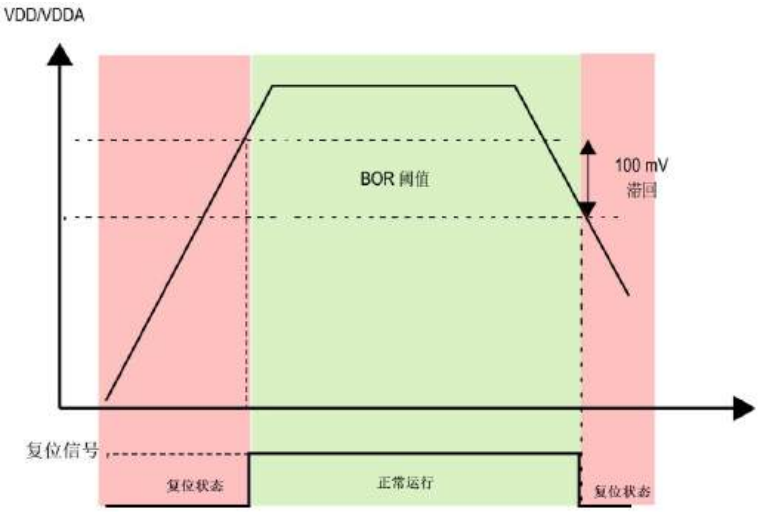


图 42-2 BOR 复位控制

42.1.1.3 可编程电压检测器 PVD

上述 POR、PDR 以及 BOR 功能都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 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 42-2。其中的上升沿和下降沿分别表示类似图 42-2 中 VDD 电压上升过程及下降过程的阈值。

表 42-2 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	2.09	2.14	2.19	V
	下降沿	1.98	2.04	2.08	V
级别 1	上升沿	2.23	2.3	2.37	V
	下降沿	2.13	2.19	2.25	V
级别 2	上升沿	2.39	2.45	2.51	V
	下降沿	2.29	2.35	2.39	V
级别 3	上升沿	2.54	2.6	2.65	V
	下降沿	2.44	2.51	2.56	V
级别 4	上升沿	2.7	2.76	2.82	V
	下降沿	2.59	2.66	2.71	V
级别 5	上升沿	2.86	2.93	2.99	V
	下降沿	2.65	2.84	3.02	V
级别 6	上升沿	2.96	3.03	3.1	V
	下降沿	2.85	2.93	2.99	V
级别 7	上升沿	3.07	3.14	3.21	V
	下降沿	2.95	3.03	3.09	V

42.1.2 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 *42_3*。

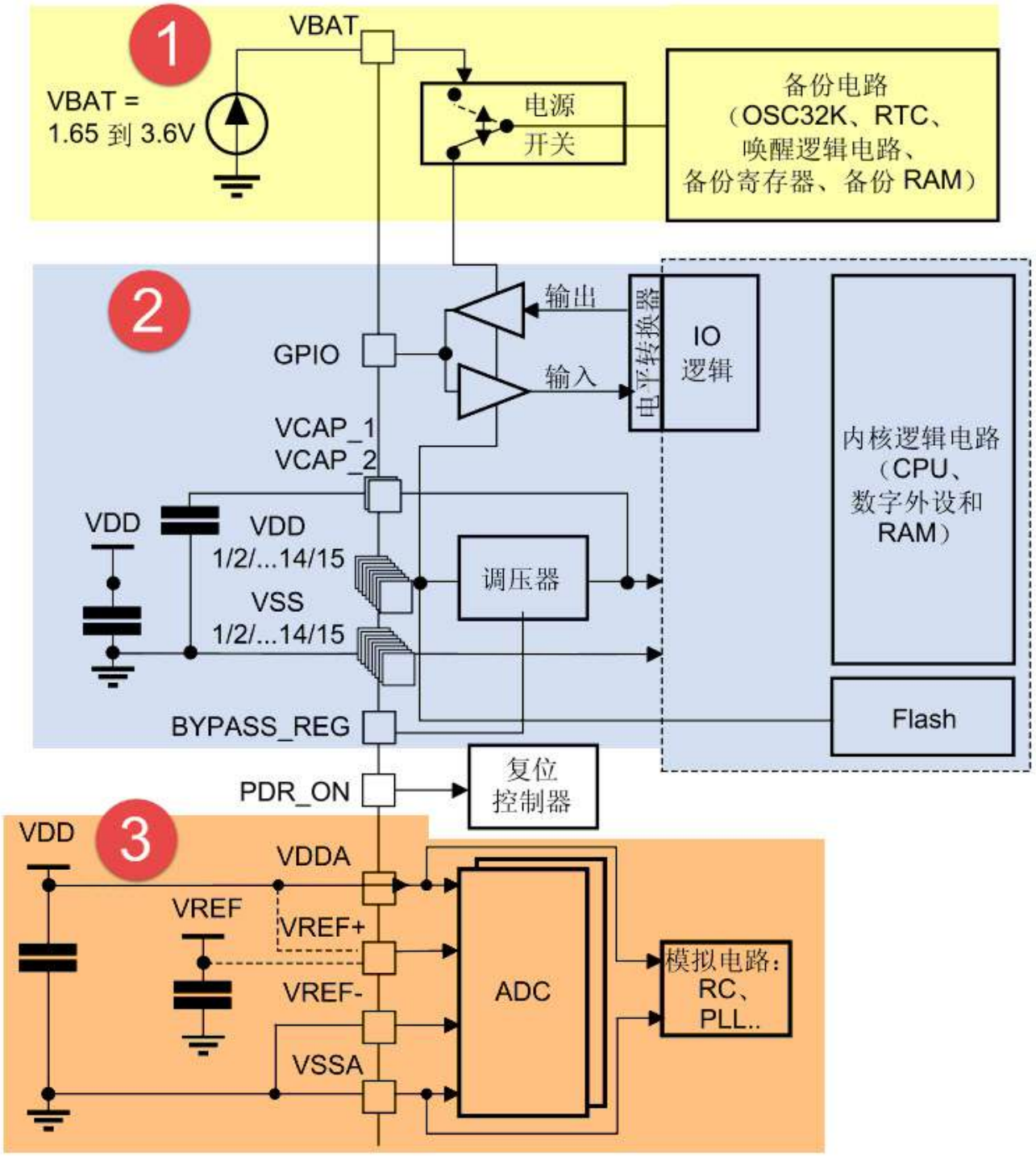


图 42-3 STM32 的电源系统

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：

- 备份域电路

  STM32 的 LSE 振荡器、RTC、备份寄存器及备份 SRAM 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 VBAT 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 *42_4* 中的双二极管，在它的上方连接了 VBAT 电源，下方连接了 VDD 主电源 (一般为 3.3V)，右侧引出到备份域电路中。当 VDD 主电源存在时，由于 VDD 电压较高，备份域电路通过 VDD 供电，当 VDD 掉电时，备份域电路由钮扣电池通过 VBAT 供电，保证电路能持续运行，从而可利用它保留关键数据。

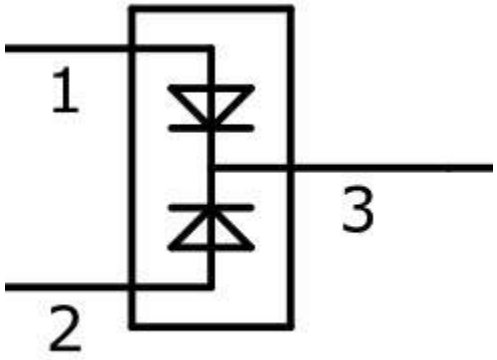


图 42-4 双二极管结构

- 调压器供电电路

  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 1.2V，因而使用调压器供电的这些电路区域被称为 1.2V 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，1.2V 域全功率运行；在停止模式下 1.2V 域运行在低功耗状态，1.2V 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 1.2V 域都断电，该区域的内核寄存器及 SRAM 内容都会丢失 (备份区域的寄存器及 SRAM 不受影响)。

- ADC 电源及参考电压

  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 VDDA 引脚输入，使用 VSSA 作为独立的地连接，VREF 引脚则为 ADC 提供测量使用的参考电压。

42.1.3 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 42-3。

表 42-3 STM32 的低功耗模式说明

模式	说明	进入方式	唤醒方式	对1.2V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M4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1.2V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	1.2V 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。

42.1.3.1 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，CM4 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 42-4。

表 42-4 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。

42.1.3.2 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 1.2V 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，可选择内部 FLASH 工作在正常模式或掉电模式。停止模式的各种特性见表 42-5。

表 42-5 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
FLASH 掉电模式	在停止模式下 FLASH 可工作在正常模式或掉电模式，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； PWR_CR 寄存器的 FPDS=0 时，FLASH 工作在正常模式，FPDS=1 时进入掉电模式。
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，若 FLASH 工作在掉电模式，还需要加上 FLASH 从掉电模式唤醒的时间。
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，STM3 2 会使用 HIS 作为系统时钟。

42.1.3.3 待机模式

待机模式，它除了关闭所有的时钟，还把 1.2V 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 42-6 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 NR ST 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、备份域内的寄存器及备份域内的 SRAM 数据会被保存，不受功耗模式影响。

42.2 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。

42.2.1 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 46-2 中说明的阈值等级。

42.2.2 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_cmInstr.h 中把这些指令封装成了函数。

列表 1: 代码清单 42-1 WFI 与 WFE 的指令定义 (core_cmInstr.h 文件)

```
/** \brief  Wait For Interrupt

    Wait For Interrupt is a hint instruction that suspends execution
    until one of a number of events occurs.
*/
#define __WFI                             __wfi

/** \brief  Wait For Event

Wait For Event is a hint instruction that permits the processor to enter
    a low-power state until one of a number of events occurs.
*/
#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，函数内部使用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。

42.2.3 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，STM32 HAL 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 42_2*。

列表 2: 代码清单 42-2 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
*        @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 */
```

(下页继续)

(续上页)

```
    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 {
        /* 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_CR1 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 (16MHz) 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，把系统时钟设置会原来的状态。

前面提到在停止模式中还可以控制内部 FLASH 的供电，控制 FLASH 是进入掉电状态还是正常供电状态，这可以使用库函数 HAL_PWREx_EnableFlashPowerDown 和 HAL_PWREx_DisableFlashPowerDown 配置，它其实只是封装了一个对 FPDS 寄存器位操作

的语句，见代码清单 42_3。这两个个函数需要在进入停止模式前被调用，即应用时需要把它放在上面的 HAL_PWR_EnterSTOPMode 之前。

列表 3: 代码清单 42-3 控制 FLASH 的供电状态

```
/**
* @brief 在停止模式时使能内部 flash 工作在掉电状态
* @retval None
*/
void HAL_PWREx_EnableFlashPowerDown(void)
{
    /* 使能 flash 掉电模式 */
    PWR->CR1 |= PWR_CR1_FPDS;
}

/**
* @brief 在停止模式时禁止内部 flash 工作在掉电状态，即正常工作
* @retval None
*/
void HAL_PWREx_DisableFlashPowerDown(void)
{
    /* 禁止 flash 掉电，即正常工作 */
    PWR->CR1 &= (uint32_t)~((uint32_t)PWR_CR1_FPDS);
}
```

42.2.4 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，其定义见代码清单 42_4。

列表 4: 代码清单 42-4 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)
* @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；另外，由于这个函数没有操作 WUF 寄存器位，所以在实际应用中，调用本函数前，还需要清空 WUF 寄存器位才能进入待机模式。

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。

42.3 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。

42.3.1 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。

42.3.2 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。

42.3.2.1 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；

42.3.2.2 代码分析

42.3.2.2.1 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，*代码清单 42_5*。

列表 5: 代码清单 42-5 睡眠模式的 main 函数 (main.c 文件)

```
int main(void)
{
    /* 初始化系统时钟为 168MHz */
    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 F407 开发板。\r\n");
    printf("\r\n 野火 F407 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) {
```

(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 *42_5*。



图 42-5 睡眠模式实验流程图

(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。

(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。

42.3.2.2.2 中断服务函数

系统刚被唤醒时会进入中断服务函数。

列表 6: 代码清单 42-6 按键中断的服务函数 (stm32f4xx_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。

42.3.3 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。

42.4 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。

42.4.1 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。

42.4.2 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。

42.4.2.1 程序设计要点

(1) 初始化用于唤醒的中断按键；

(2) 设置停止状态时的 FLASH 供电或掉电；

(3) 选择电压调节器的工作模式并进入停止状态；

(4) 使用按键中断唤醒芯片；

(5) 重启 HSE 时钟，使系统完全恢复停止前的状态。

42.4.2.2 代码分析

42.4.2.2.1 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 STM32F407 中，HSI 时钟一般为 16MHz，与我们常用的 168MHz 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。

列表 7: 代码清单 42-7 恢复系统时钟 (main.c 文件)

```
/*
* @brief  从停止模式唤醒后配置系统时钟：启用 HSE、PLL 并选择 PLL 作为系统时钟源。

* @param  无
* @retval 无
*/
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);
```

(下页继续)

(续上页)

```
    /* 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) {
            ;
        }
    }

    /* 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。

42.4.2.2.2 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。

列表 8: 代码清单 42-8 停止模式的 main 函数 (main.c 文件)

```
int main(void)
{
    uint32_t SYSCLK_Frequency=0;
    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 168MHz */
    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 F407 开发板。\r\n");
    printf("\r\n 野火 F407 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");
```

(下页继续)

(续上页)

```
    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) {
        /********** 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();
```

(下页继续)

(续上页)

```
        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");
        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");
```

(下页继续)

(续上页)

```
        //继续执行 while 循环
    }
}
```

这个 main 函数的执行流程见图 *42_5*。

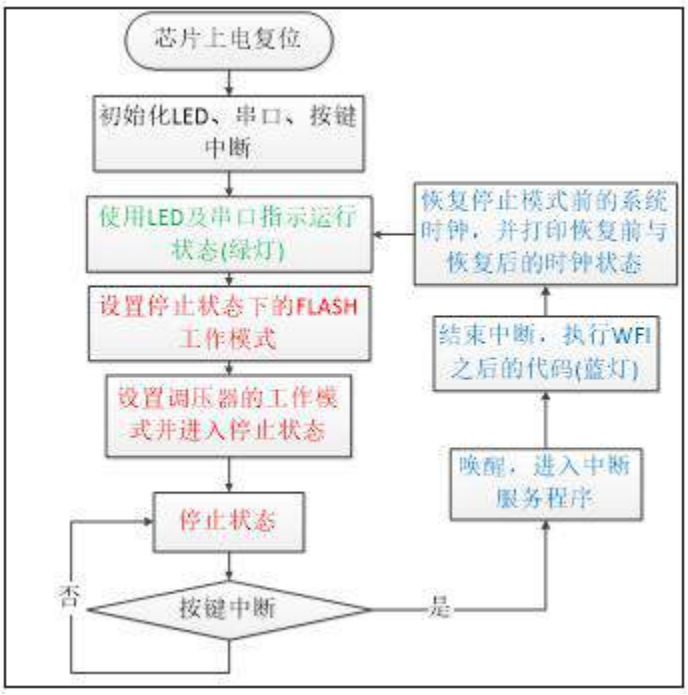


图 42-6 停止模式实验流程图

(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，这里串口的时钟源设定为 HSI 方便实验打印，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 SystemCoreClockUpdate、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 16MHz，恢复后的系统时钟采用 HSE 倍频后的 PLL 时钟，时钟频率为 168MHz。

42.4.3 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。

42.5 PWR—待机模式实验

最后我们来学习最低功耗的待机模式。

42.5.1 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。

42.5.2 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。

42.5.2.1 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。

42.5.2.2 代码分析

42.5.2.2.1 main 函数

待机模式实验的执行流程比较简单。

列表 9: 代码清单 42-9 停止模式的 main 函数 (main.c 文件)

```
int main(void)
{
    /* 初始化系统时钟为 168MHz */
    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 F407 开发板。\r\n");
    printf("\r\n 野火 F407 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 *42_5*。

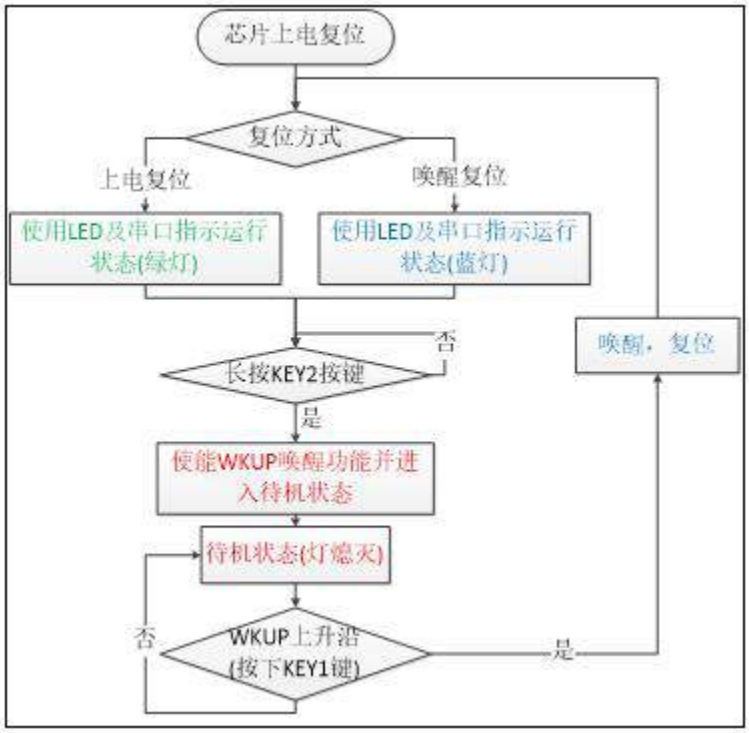


图 42-7 待机模式实验流程图

(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 42_10* 所示。

列表 10: 代码清单 42-10 Key_GPIO_Config 函数 (bsp_key.c 文件)

```
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 */
    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);
}
```

(1) 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。

(2) 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。

(3) 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。

(4) 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。

(5) 按下 KEY1 按键，会使 PA0 引脚产生一个上升沿，从而唤醒系统。

(6) 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。

42.5.3 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。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。

42.6 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。

42.6.1 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 *42_8*。

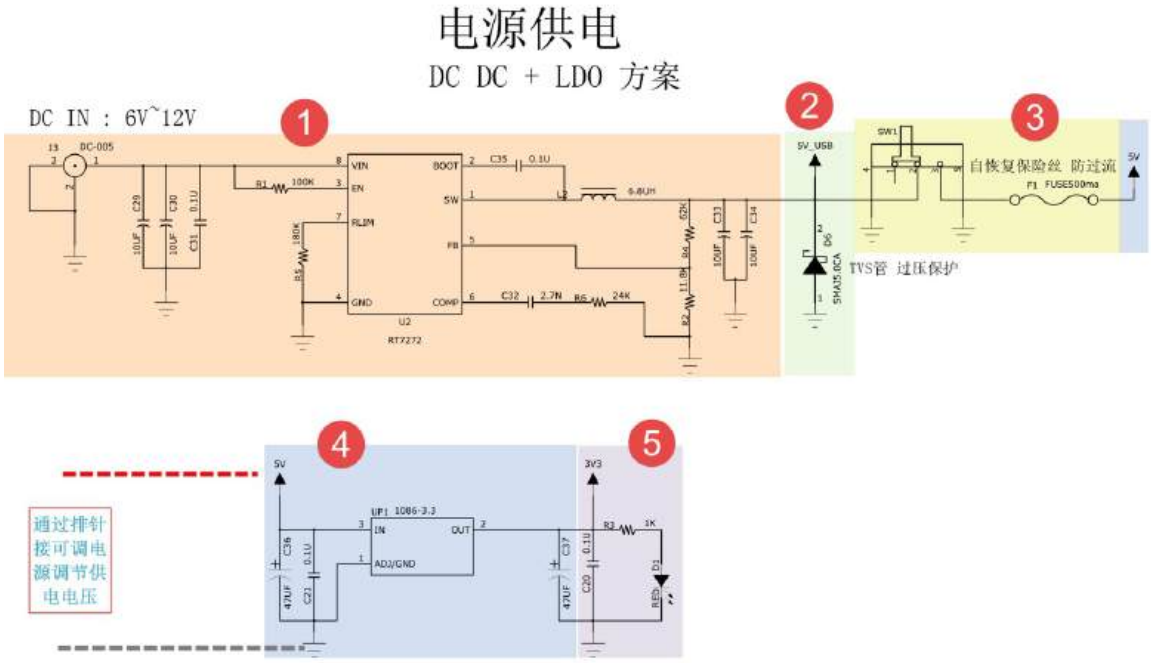


图 42-8 实验板的电源供电系统

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。

42.6.2 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。

42.6.2.1 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。

42.6.2.2 代码分析

42.6.2.2.1 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。

列表 11: 代码清单 42-11 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();
}
```

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 42-2 或 STM32 的数据手册。

(4) 最后使用库函数 HAL_PWR_EnablePVD 使能 PVD 功能。

42.6.2.2.2 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 42_12*。

列表 12: 代码清单 42-12 PVD 中断服务函数 (stm32f4xx_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。

42.6.2.2.3 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。

列表 13: 代码清单 42-13 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 */
    }

}
```

42.6.3 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_5”时，当可调电源电压降至 4V 时，板子亮红灯，此时的“3.3V”电源引脚的实测电压为 2.8V

注意：

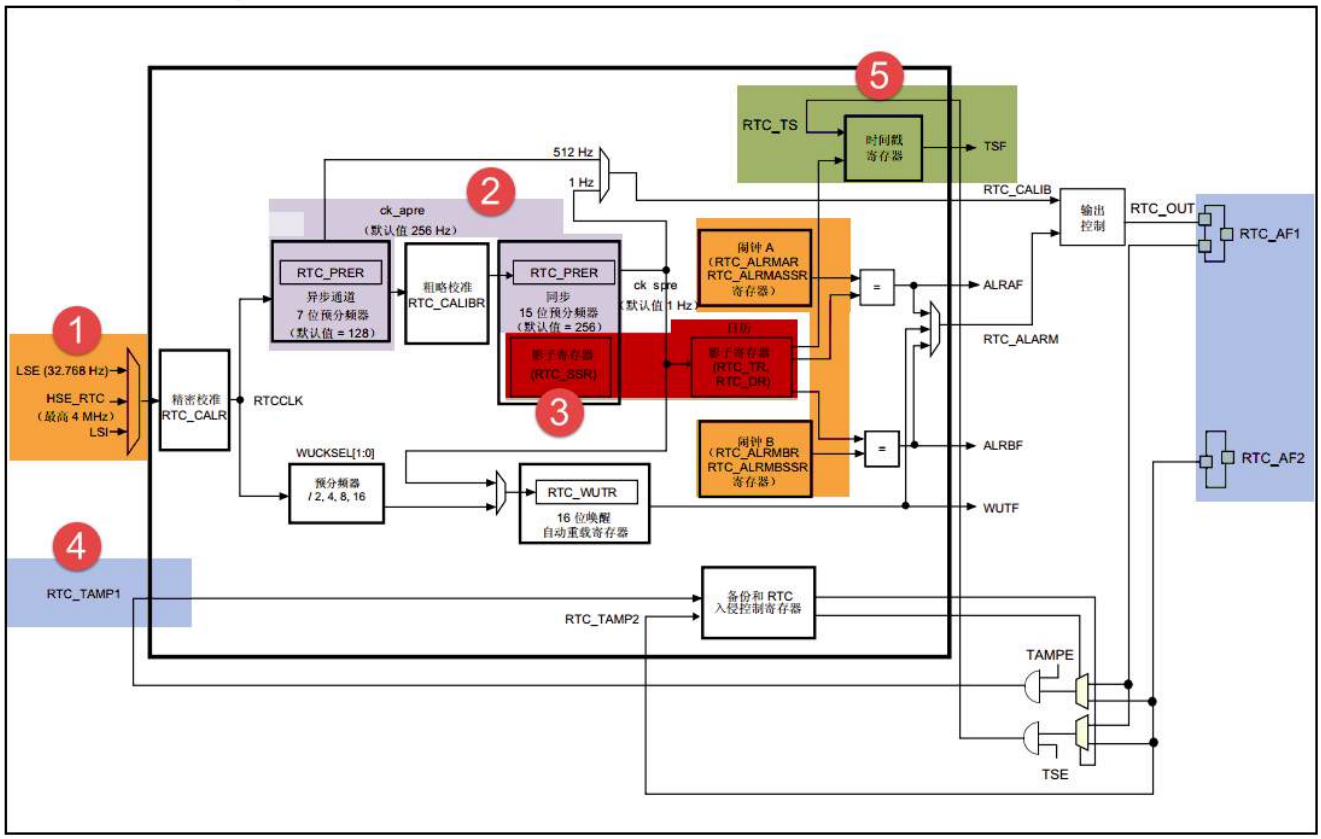
由于这样使用可调电源供电不经过保险丝，所以在调节电压的时候要小心，不要给它供电远高于 5V，否则可能会烧坏实验板上的芯片。

第 43 章　RTC—实时时钟

43.1 RTC 简介

RTC—real time clock，实时时钟，主要包含日历、闹钟和自动唤醒这三部分的功能，其中的日历功能我们使用的最多。日历包含两个 32bit 的时间寄存器，可直接输出时分秒，星期、月、日、年。比起 F103 系列的 RTC 只能输出秒中断，剩下的其他时间需要软件来实现，F407 的 RTC 可谓是脱胎换骨，让我们在软件编程时大大降低了难度。RTC 功能框图分析

43.2 RTC 功能框图解析



43.2.1 时钟源

RTC 时钟源—RTCCLK 可以从 LSE、LSI 和 HSE_RTC 这三者中得到。其中使用最多的是 LSE，LSE 由一个外部的 32.768KHZ（6PF 负载）的晶振提供，精度高，稳定，RTC 首选。LSI 是芯片内部的 30KHZ 晶体，精度较低，会有温漂，一般不建议使用。HSE_RTC 由 HSE 分频得到，最高是 4M，使用的也较少。

43.2.2 预分频器

预分频器 PRER 由 7 位的异步预分频器 APRE 和 15 位的同步预分频器 SPRE 组成。异步预分频器时钟 CK_APRE 用于为二进制 RTC_SSR 亚秒递减计数器提供时钟，同步预分频器时钟 CK_SPRE 用于更新日历。异步预分频器时钟 $f_{CK_APRE}=f_{RTC_CLK}/(PREDIV_A+1)$，同步预分频器时钟 f:sub:*CK_SPRE*=$f_{RTC_CLK}/(PREDIV_S+1)$，)。使用两个预分频器时，推荐将异步预分频器配置为较高的值，以最大程度降低功耗。一般我们会使用 LSE 生成 1HZ 的同步预分频器时钟。通常的情况下，我们会选择 LSE 作为 RTC 的时钟源，即 f:sub:*RTCCLK*=f_{LSE}=32.768KHZ。然后经过预分频器 PRER 分频生成 1HZ 的时钟用于更新日历。使用两个预分频器分频的时候，为了最大程度的降低功耗，我们一般把同步预分频器设置成较大的值，为了生成 1HZ 的同步预分频器时钟 CK_SPRE，最常用的配置是 PREDIV_A=127，PREDIV_S=255。计算公式为：$f_{CK_SPRE}=f_{RTCCLK}$/{（PREDIV_A+1）*（PREDIV_S+1）}= 32.768/{（127+1）*（255+1）}=1HZ。

43.2.3 实时时钟和日历

我们知道，实时时钟一般是这样表示的：时/分/秒/亚秒，其中时分秒可直接从 RTC 时间寄存器(RTC_TR) 中读取，有关时间寄存器的说明具体见图 *39_1* 和表 39-1 程序组件所属的区域。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PM	HT[1:0]		HU[3:0]			
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	MNT[2:0]			MNU[3:0]				Reserved	ST[2:0]			SU[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

图 39_1 RTC 时间寄存器（RTC_TR）

表 39-1 时间寄存器位功能说明

位名称	位说明
PM	AM/PM 符号，0:AM/24 小时制，1:PM
HT[1:0]	小时的十位
HU[3:0]	小时的个位
MNT[2:0]	分钟的十位
MNU[3:0]	分钟的个位
ST[2:0]	秒的十位
SU[3:0]	秒的个位

亚秒由 RTC 亚秒寄存器 (RTC_SSR) 的值计算得到，公式为：亚秒值 = (PREDIV_S –SS[15:0]) / (PREDIV_S + 1)，SS[15:0] 是同步预分频器计数器的值，PREDIV_S 是同步预分频器的值。

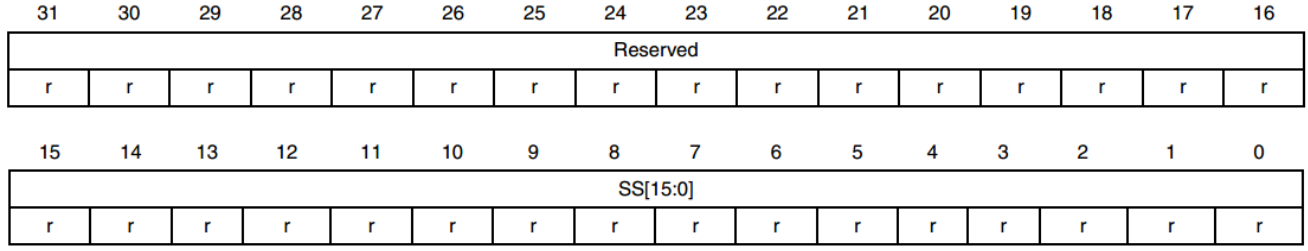


图 39_2 RTC 亚秒寄存器（RTC_SSR）

日期包含的年月日可直接从 RTC 日期寄存器 (RTC_DR) 中读取。

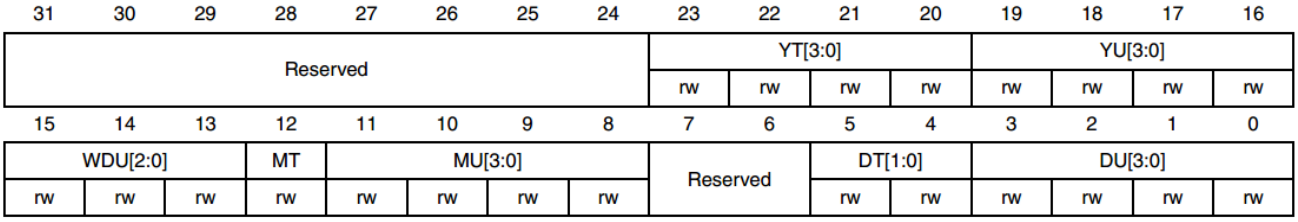


图 39_3 RTC 日期寄存器（RTC_DR）

表 39-2 RTC 日期寄存器位功能说明

位名称	位说明
YT[1:0]	年份的十位
YU[3:0]	年份的个位
WDU[2:0]	星期几的个位，000：禁止，001：星期一，…，111：星期日
MT	月份的十位
MU	月份的个位
DT[1:0]	日期的十位
DU[3:0]	日期的个位

当应用程序读取日历寄存器时，默认是读取影子寄存器的内容，每隔两个 RTCCLK 周期，便将当前日历值复制到影子寄存器。我们也可以通过将 RTC_CR 寄存器的 BYPSHAD 控制位置 1 来直接访问日历寄存器，这样可避免等待同步的持续时间。

RTC_CLK 经过预分频器后，有一个 512HZ 的 CK_APRE 和 1 个 1HZ 的 CK_SPRE，这两个时钟可以成为校准的时钟输出 RTC_CALIB，RTC_CALIB 最终要输出则需映射到 RTC_AF1 引脚，即 PC13 输出，用来对外部提供时钟。

43.2.4 闹钟

RTC 有两个闹钟，闹钟 A 和闹钟 B,，当 RTC 运行的时间跟预设的闹钟时间相同的时候，相应的标志位 ALRAF（在 RTC_ISR 寄存器中）和 ALRBF 会置 1。利用这个闹钟我们可以做一些备忘提醒功能。

如果使能了闹钟输出（由 RTC_CR 的 OSEL[0:1] 位控制），则 ALRAF 和 ALRBF 会连接到闹钟输出引脚 RTC_ALARM，RTC_ALARM 最终连接到 RTC 的外部引脚 RTC_AF1（即 PC13），输出的极性由 RTC_CR 寄存器的 POL 位配置，可以是高电平或者低电平。

43.2.5 时间戳

时间戳即时间点的意思，就是某一个时刻的时间。时间戳复用功能 (RTC_TS) 可映射到 RTC_AF1 或 RTC_AF2，当发生外部的入侵事件时，即发生时间戳事件时，RTC_ISR 寄存器中的时间戳标志位 (TSF) 将置 1，日历会保存到时间戳寄存器（RTC_TSSSR、RTC_TSTR 和 RTC_TSDR）中。时间戳往往用来记录危及时刻的时间，以供事后排查问题时查询。

43.2.6 入侵检测

RTC 自带两个入侵检测引脚 RTC_AF1（PC13）和 RTC_AF2（PI8，PI8 只有在 176pin 引脚的型号中才有），这两个输入既可配置为边沿检测，也可配置为带过滤的电平检测。当发生入侵检测时，备份寄存器将被复位。备份寄存器 (RTC_BKPxR) 包括 20 个 32 位寄存器，用于存储 80 字节的用户应用数据。这些寄存器在备份域中实现，可在 VDD 电源关闭时通过 VBAT 保持上电状态。备份寄存器不会在系统复位或电源复位时复位，也不会在器件从待机模式唤醒时复位。

43.3 RTC 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 stm32f4xx_hal_rtc.h 头文件中，初始化库函数定义在 stm32f4xx_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 HourFormat;       /* 配置 RTC 小时格式 */
```

(下页继续)

(续上页)

```
  uint32_t AsynchPrediv;     /* 配置 RTC_CLK 的异步分频因子 (0x00~0x7F ) */

  uint32_t SynchPrediv;      /* 配置 RTC_CLK 的同步分频因子 (0x00~0x7FFF ) */

  uint32_t OutPut;           /* 指定哪一路信号作为 RTC 的输出 */

  uint32_t OutPutPolarity;   /* 配置 RTC 输出信号的极性 */

  uint32_t OutPutType;       /* 配置 RTC 输出引脚的模式，开漏或者推挽 */
} RTC_InitTypeDef;
```

1) HourFormat：小时格式设置，有 RTC_HOURFORMAT_24 和 RTC_HOURFORMAT_12 两种格式，一般我们选择使用 24 小时制，具体由 RTC_CR 寄存器的 FMT 位配置。

2) 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。

3) SynchPrediv：RTC_CLK 同步分频因子设置，15 位有效，具体由 RTC 预分频器寄存器 RTC_PRER 的 PREDIV_S[14:0] 位配置。

4) 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。

5) OutPutPolarity：RTC 输出信号极性设置，可以设置为 RTC_OUTPUT_POLARITY_HIGH 和 RTC_OUTPUT_POLARITYLOW。

6) OutPutType：RTC 输出引脚的模式设置，可以是开漏或者推挽。

43.4 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。

```
typedef struct {
   uint8_t Hours;     /* 小时设置 */
   uint8_t Minutes;   /* 分钟设置 */
   uint8_t Seconds;   /* 秒设置 */
   uint32_t SubSeconds;       /* 亚秒设置 */
   uint32_t  SecondFraction;    /* 亚秒同步预分频系数 */
   uint8_t   TimeFormat;      /* AM/PM 符号设置 */
   uint32_t  DayLightSaving;  /* 夏令时日历时间设置 */
   uint32_t  StoreOperation;  /* AM/PM 符号设置 */
} 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。
4) SubSeconds：亚秒设置，取值范围 0~1(s)，1 Sec /(SecondFraction +1)。
5) SecondFraction：亚秒预分频系数，用于获取更加精确的 RTC 时间。
6) TimeFormat: AM/PM 设置，可取值 RTC_HOURFORMAT12_AM 和 RTC_HOURFORMAT12_PM，RTC_HOURFORMAT12_AM 时则是 24 小时制，RTC_HOURFORMAT12_PM 则是 12 小时制。
7) DayLightSaving：夏令时日历时间设置，可以增加一个小时，或者减一个小时，或者保持不变。
8) StoreOperation：用户对 RTC_CR 寄存器的 BKP 位执行写操作以记录是否已对夏令时进行更改。

43.5 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。

```
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。

43.6 RTC 闹钟结构体讲解

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。

```
typedef struct {
   RTC_TimeTypeDef AlarmTime;      /* 设定 RTC 时间寄存器的值：时/分/秒 */
   uint32_t AlarmMask;             /* RTC 闹钟 掩码字段选择 */
   uint32_t AlarmSubSecondMask;    /* RTC 闹钟 掩码字段选择 */
```

(下页继续)

(续上页)

```
  uint32_t AlarmDateWeekDaySel;   /* 闹钟的日期/星期选择 */

  uint8_t AlarmDateWeekDay;       /* 指定闹钟的日期/星期
                                     * 如果日期有效，则取值范围为 1~31
                                     * 如果星期有效，则取值为 1~7
                                     */
  uint32_t Alarm;                 /* RTC 闹钟选择 */
} RTC_AlarmTypeDef;
```

1. AlarmTime：闹钟时间设置，配置的是 RTC 时间初始化结构体，主要配置小时的制式，有 12 小时或者是 24 小时，配套具体的时、分、秒。

2. AlarmMask：闹钟掩码字段选择，即选择闹钟时间哪些字段无效，取值可为：RTC_ALARMMASK_NONE（全部有效）、RTC_ALARMMASK_DATEWEEKDAY（日期或者星期无效）、RTC_ALARMMASK_HOURS（小时无效）、RTC_ALARMMASK_MINUTES（分钟无效）、RTC_ALARMMASK_SECONDS（秒钟无效）、RTC_ALARMMASK_ALL（全部无效）。比如我们选择 RTC_ALARMMASK_DATEWEEKDAY，那么就是当 RTC 的时间的小时等于闹钟时间小时字段时，每天的这个小时都会产生闹钟中断。

3. AlarmSubSecondMask：闹钟亚秒掩码字段选择，即选择闹钟亚秒寄存器 RTC_TSSSR 哪些字段无效，取值可为 15 段。

4. AlarmDateWeekDaySel：闹钟日期或者星期选择，可选择 RTC_ALARMDATEWEEKDAYSEL_WEEKDAY 或者 RTC_ALARMDATEWEEKDAYSEL_DATE。要想这个配置有效，则 AlarmMask 不能配置为 RTC_ALARMMASK_DATEWEEKDAY，否则会被 MASK 掉。

5. AlarmDateWeekDay：具体的日期或者星期几，当 AlarmDateWeekDaySel 设置成 RTC_ALARMDATEWEEKDAYSEL_WEEKDAY 时，取值为 1~7，对应星期一 ~ 星期日，当设置成 RTC_ALARMDATEWEEKDAYSEL_DATE 时，取值为 1~31。

6. Alarm：RTC 闹钟选择，即选择闹钟 A 或者闹钟 B。

43.7 RTC—日历实验

利用 RTC 的日历功能制作一个日历，显示格式为：年-月-日-星期，时-分-秒。

43.7.1 硬件设计

该实验用到了片内外设 RTC，为了确保在 VDD 断电的情况下时间可以保存且继续运行，VBAT 引脚外接了一个 CR1220 电池座，用来放 CR1220 电池给 RTC 供电，电路图具体见图 *39_4*。

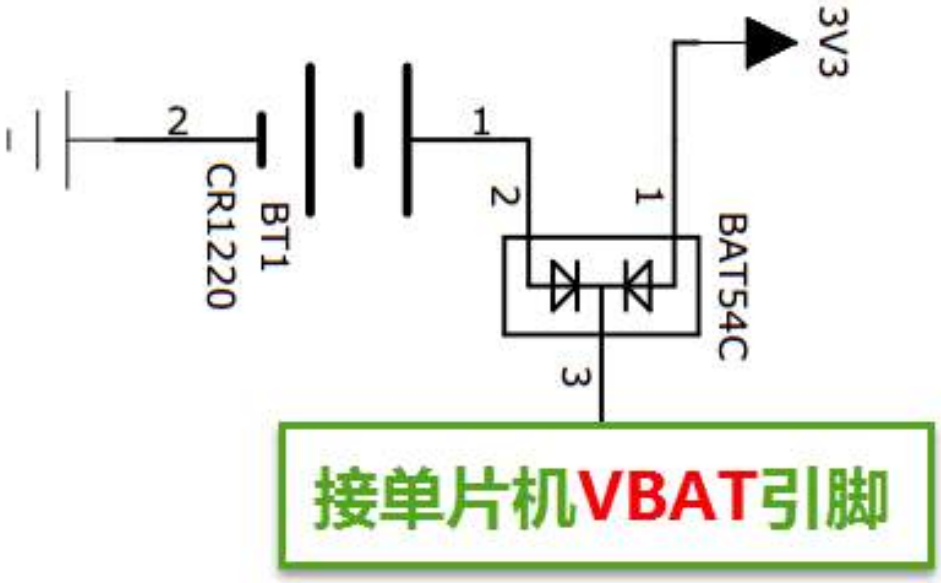


图 39_4 RTC 外接 CR1220 电池座子

43.7.2 软件设计

43.7.2.1 编程要点

1) 选择 RTC_CLK 的时钟源；

2) 配置 RTC_CLK 的分频系数，包括异步和同步两个；

3) 设置初始时间，包括日期；

4) 获取时间和日期，并显示；

43.7.2.2 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。

43.7.2.2.1 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
```

(下页继续)

(续上页)

```
#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
// 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。

43.7.2.2.2 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();
```

(下页继续)

(续上页)

```
#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;
   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);
```

(下页继续)

(续上页)

```
#endif /* RTC_CLOCK_SOURCE_LSI */

   /* 使能 RTC 时钟 */
   __HAL_RCC_RTC_ENABLE();

   /* 等待 RTC APB 寄存器同步 */
   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。

43.7.2.2.3 RTC 时间初始化函数

```
/**
* @brief  设置时间和日期
* @param  无
* @retval 无
*/
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。

43.7.2.2.4 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
```

(下页继续)

(续上页)

```
            //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",
                    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。

43.7.2.2.5 主函数

```
int main(void)
{
   /* 系统时钟初始化成 168 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
      *
```

(下页继续)

(续上页)

```
    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。
    */

/* 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);
```

(下页继续)

(续上页)

```
    }
    /* 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。

43.7.3 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。

43.8 RTC—闹钟实验

在日历实验的基础上，利用 RTC 的闹钟功能制作一个闹钟，在每天的 [XX 小时-XX 分钟-XX 秒钟] 产生闹钟，然后蜂鸣器响。

43.8.1 硬件设计

硬件设计跟日历实验部分的硬件设计一样。

43.8.2 软件设计

闹钟实验是在日历实验的基础上添加，相同部分的代码不再讲解，这里只讲解闹钟相关的代码，更加具体的请参考闹钟实验的工程源码。

43.8.2.1 闹钟相关宏定义

```
// 闹钟相关宏定义
#define ALARM_HOURS                 1           // 0~23
#define ALARM_MINUTES               1           // 0~59
#define ALARM_SECONDS               10           // 0~59

#define ALARM_MASK                  RTC_ALARMMASK_DATEWEEKDAY
#define ALARM_DATE_WEEKDAY_SEL      RTC_ALARMDATEWEEKDAYSEL_DATE
#define ALARM_DATE_WEEKDAY          2
#define RTC_Alarm_X                 RTC_ALARM_A
```

为了方便程序移植，我们把需要频繁修改的代码用宏封装起来。如果需要设置闹钟时间和闹钟的掩码字段，只需要修改这些宏即可。这些宏对应的是 RTC 闹钟结构体的成员，想知道每个宏的具体含义可参考“RTC 闹钟结构体讲解”小节。

闹钟时间字段掩码 ALARM_MASK 我们配置为 MASK 掉日期/星期，即忽略日期/星期，则闹钟时间只有时/分/秒有效，即每天到了这个时间闹钟都会响。掩码还有其他取值，用户可自行修改做实验。

43.8.2.2 编程要点

1) 初始化 RTC，设置 RTC 初始时间；

2) 编程闹钟，设置闹钟时间；

3) 编写闹钟中断服务函数；

43.8.3 代码分析

43.8.3.1 闹钟设置函数

```
/*
*     要使能 RTC 闹钟中断，需按照以下顺序操作：
* 1. 配置 NVIC 中的 RTC_Alarm IRQ 通道并将其使能。
* 2. 配置 RTC 以生成 RTC 闹钟（闹钟 A 或闹钟 B）。
*
*
*/
void RTC_AlarmSet(void)
{
   RTC_AlarmTypeDef  RTC_AlarmStructure;

   /* RTC 闹钟中断配置 */
   /* EXTI 配置 */
   HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 0, 0);
   /* 使能 RTC 闹钟中断 */
   HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);

   /* 设置闹钟时间 */
   RTC_AlarmStructure.Alarm = RTC_Alarm_X;
   RTC_AlarmStructure.AlarmTime.TimeFormat     = RTC_H12_AMorPM;
   RTC_AlarmStructure.AlarmTime.Hours    = ALARM_HOURS;
```

（下页继续）

(续上页)

```
    RTC_AlarmStructure.AlarmTime.Minutes = ALARM_MINUTES;
    RTC_AlarmStructure.AlarmTime.Seconds = ALARM_SECONDS;
    RTC_AlarmStructure.AlarmMask = ALARM_MASK;
    RTC_AlarmStructure.AlarmDateWeekDaySel = ALARM_DATE_WEEKDAY_SEL;
    RTC_AlarmStructure.AlarmDateWeekDay = ALARM_DATE_WEEKDAY;

    HAL_RTC_SetAlarm_IT(&Rtc_Handle,&RTC_AlarmStructure, RTC_FORMAT_BIN);
}
```

从参考手册知道，要使能 RTC 闹钟中断，必须按照三个步骤进行。RTC_AlarmSet() 函数则根据这三个步骤和代码中的注释阅读即可。

RTC 中断

所有 RTC 中断均与 EXTI 控制器相连。

要使能 RTC 闹钟中断，需按照以下顺序操作：

1. 将 EXTI 线 17 配置为中断模式并将其使能，然后选择上升沿有效。
2. 配置 NVIC 中的 RTC_Alarm IRQ 通道并将其使能。
3. 配置 RTC 以生成 RTC 闹钟（闹钟 A 或闹钟 B）。

图 39_5 RTC 闹钟中断编程步骤（摘自 STM32F4xx 参考手册 RTC 章节）

在第 3 步中，配置 RTC 以生成 RTC 闹钟中，在手册中也有详细的步骤说明，编程的时候必须按照这个步骤，具体见图 *39_6*。

编程闹钟

要对可编程的闹钟（闹钟 A 或闹钟 B）进行编程或更新，必须执行类似的步骤：

1. 将 RTC_CR 寄存器中的 ALRAE 或 ALRBE 位清零以禁止闹钟 A 或闹钟 B。
2. 轮询 RTC_ISR 寄存器中的 ALRAWF 或 ALRBWF 位，直到其中一个置 1，以确保闹钟寄存器可以访问。大约需要 2 个 RTCCLK 时钟周期（由于时钟同步）。
3. 编程闹钟 A 或闹钟 B 寄存器（RTC_ALRMASSR/RTC_ALRMAR 或 RTC_ALRMBSSR/RTC_ALRMBR）。
4. 将 RTC_CR 寄存器中的 ALRAE 或 ALRBE 位置 1 以再次使能闹钟 A 或闹钟 B。

图 39_6 RTC 闹钟编程步骤（摘自 STM32F4xx 参考手册 RTC 章节）

编程闹钟的步骤 1 和 2，由固件库函数 RTC_AlarmCmd(RTC_Alarm_X, DISABLE); 实现，即在编程闹钟寄存器设置闹钟时间的时候必须先失能闹钟。剩下的两个步骤配套代码的注释阅读即可。

43.8.3.2 闹钟中断服务函数

```
void RTC_Alarm_IRQHandler(void)
{
   HAL_RTC_AlarmIRQHandler(&Rtc_Handle);
}
/**
* @brief  Alarm callback
* @param  hrtc : RTC handle
* @retval None
*/
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
   /* 闹钟时间到，蜂鸣器标志位置 1 */
   Alarmflag = 1;
}
```

如果日历时间到了闹钟设置好的时间，则产生闹钟中断，在中断函数中把相应的标志位清 0。然后中断服务函数会调用闹钟回调函数，为了表示闹钟时间到，我们让蜂鸣器响。

43.8.3.3 main 函数

```
int main(void)
{
   /* 系统时钟初始化成 168 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 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) {
          /* 闹钟设置 */
          RTC_AlarmSet();
          /* 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，我们通过读取备份寄存器的值来判断 RTC 是否初始化过，如果没有则初识话 RTC，并设置闹钟时间，如果已经初始化过，则判断是电源还是外部引脚复位，并清除闹钟相关的中断标志位。

43.8.4 下载验证

把编译好的程序下载到开发板，当日历时间到了闹钟时间时，蜂鸣器一直响，但日历会继续运行。

第 44 章　MPU6050 传感器—姿态检测

本章参考数据：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库说明文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

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。

44.1 姿态检测

44.1.1 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。

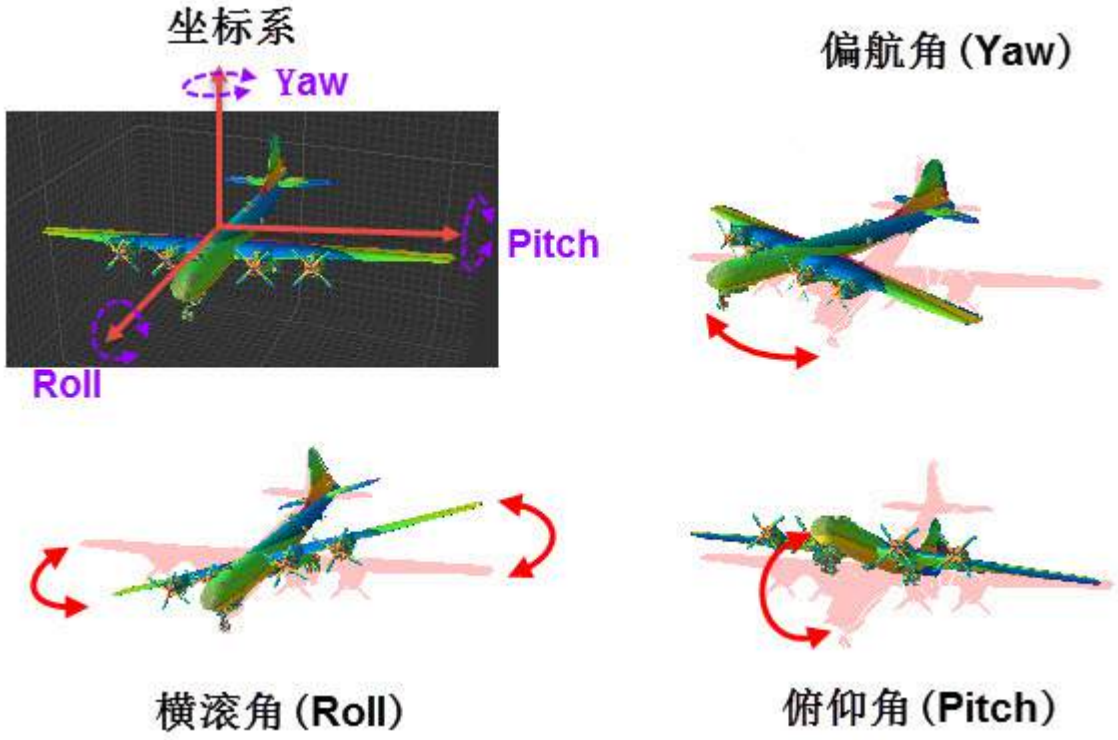


图 46-1 表示飞机姿态的偏航角、横滚角及俯仰角

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。

44.1.2 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。

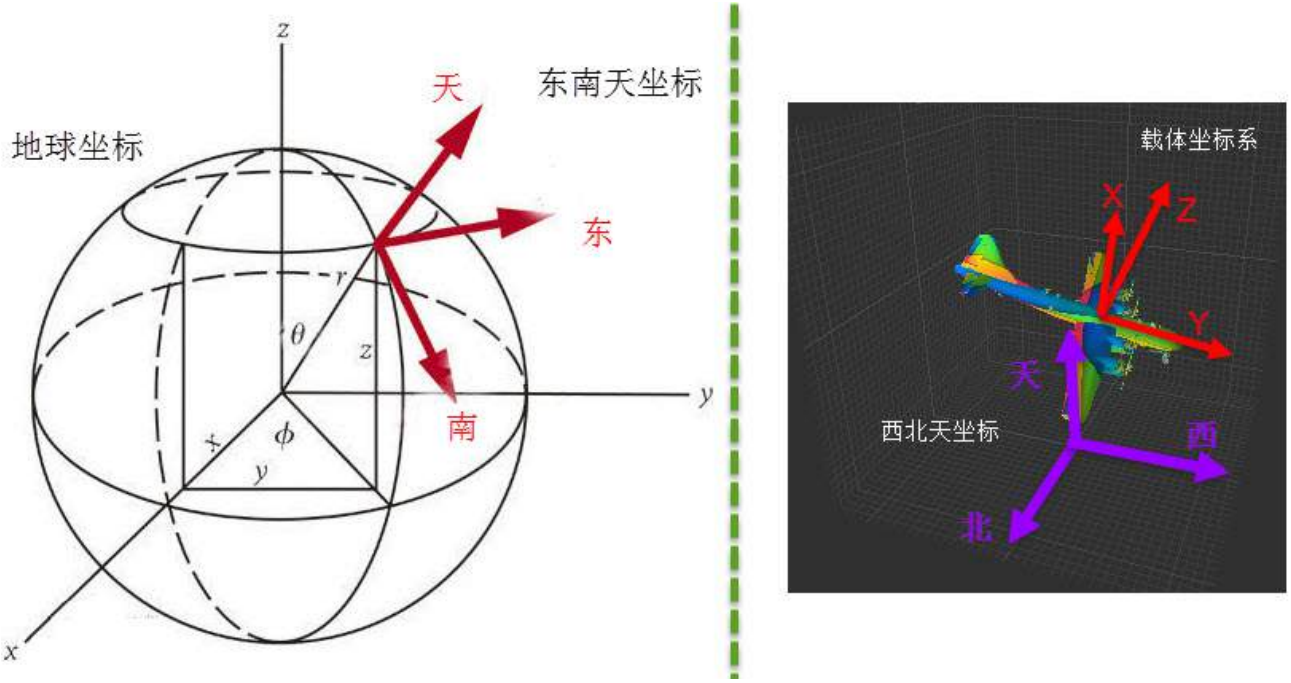


图 46-2 地球坐标系、地理坐标系与载体坐标系

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。

44.1.3 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”。

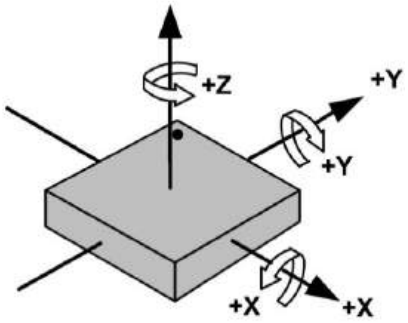


图 46-3 陀螺仪检测示意图

44.1.3.1 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。

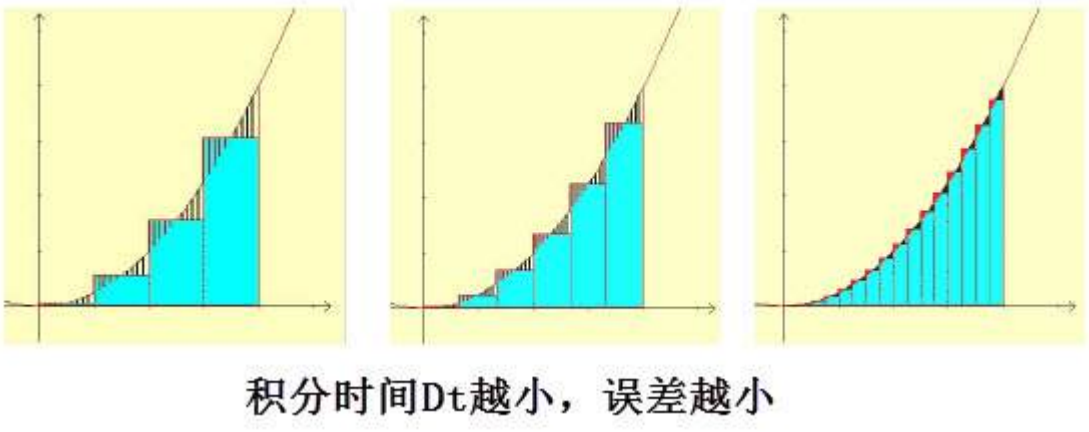


图 46-4 积分误差

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。

44.1.4 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*。

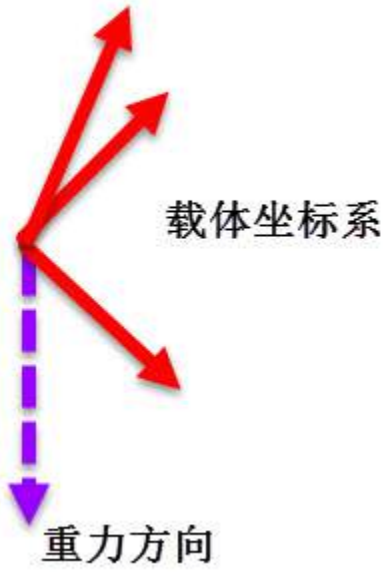


图 46-6 重力检测

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。

44.1.4.1 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。

44.1.5 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。

44.1.5.1 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。

44.1.6 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)。

44.1.7 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。

44.2 传感器

44.2.1 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。

44.2.2 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。

44.2.3 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。

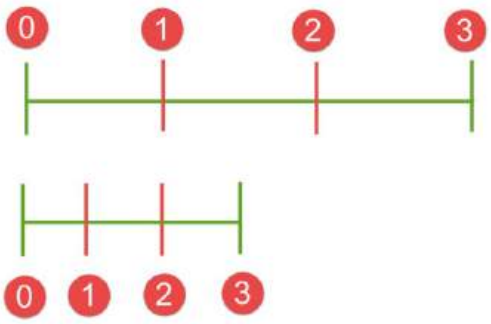


图 46-8 ADC 表示的物理量范围

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。

44.3 MPU6050 简介

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 STM32F4 实验板上有一个 MPU6050 芯片，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。

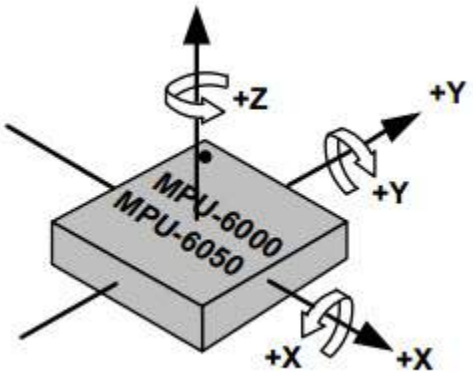


图 46-9 MPU6050 传感器的坐标及方向

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。

44.4 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。

44.5 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。

44.5.1 硬件设计

STM32 与 MPU6050 的硬件连接见图 *46_014* 。

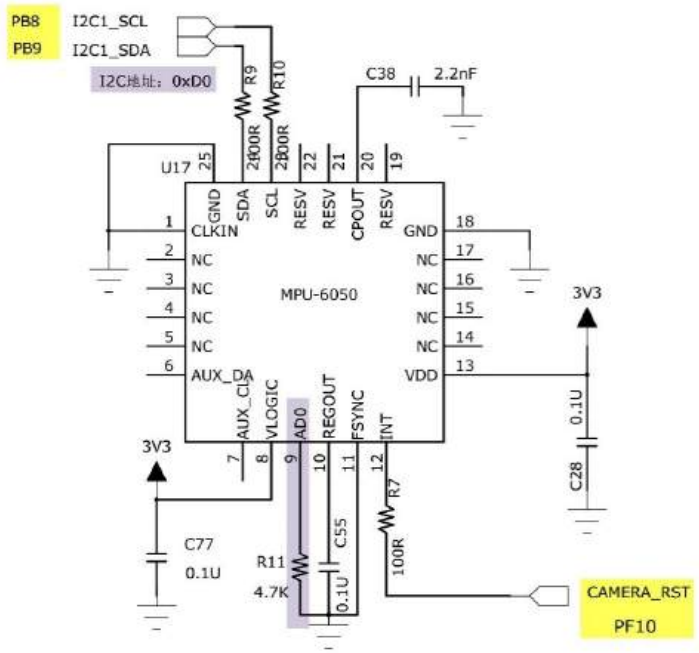


图 46-014 STM32 与 MPU6050 的硬件连接

它的硬件连接非常简单，SDA 与 SCL 引出到 STM32 的 I2C 引脚，注意图中的 I2C 没有画出上拉电阻，只是因为实验板中其它芯片也使用了同样的 I2C 总线，电阻画到了其它芯片的图里，没有出现在这个图中而已。传感器的 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，传感器的 INT

引脚接到了 STM32 的普通 IO 口，当传感器有新数据的时候会通过 INT 引脚通知 STM32。

由于 MPU6050 检测时是基于自己中心坐标系的，所以在自己设计硬件时，您需要考虑它与所在设备的坐标系统的关系。

44.5.2 软件设计

本小节讲解的是“MPU6050 基本数据读取”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“i2c.c”及“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。

44.5.2.1 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。

44.5.2.2 代码分析

44.5.2.2.1 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。

```
/* 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_8
```

(下页继续)

(续上页)

```
#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_9

#define SENSORS_I2C_AF                      GPIO_AF4_I2C1

#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。

44.5.2.2.2 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 1: 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();
```

(下页继续)

(续上页)

```
/* 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_HIGH;
GPIO_InitStructure.Pull= GPIO_NOPULL;
GPIO_InitStructure.Alternate=SENSORS_I2C_AF;
HAL_GPIO_Init(SENSORS_I2C_SCL_GPIO_PORT, &GPIO_InitStructure);

/* 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.Timing           = 0x60201E2B;//100KHz
      I2C_Handle.Init.OwnAddress1      = 0;
      I2C_Handle.Init.AddressingMode   = I2C_ADDRESSINGMODE_7BIT;
      I2C_Handle.Init.DualAddressMode  = I2C_DUALADDRESS_DISABLE;
      I2C_Handle.Init.OwnAddress2      = 0;
      I2C_Handle.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
      I2C_Handle.Init.GeneralCallMode  = I2C_GENERALCALL_DISABLE;
      I2C_Handle.Init.NoStretchMode    = I2C_NOSTRETCH_DISABLE;

      /* 初始化 I2C */
```

(下页继续)

(续上页)

```
        HAL_I2C_Init(&I2C_Handle);
        /* 使能模拟滤波器 */
        HAL_I2CEx_AnalogFilter_Config(&I2C_Handle, I2C_ANALOGFILTER_
↪ENABLE);
    }
}
```

44.5.2.2.3 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 2: 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,
```

(下页继续)

(续上页)

```
    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,
    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
```

(下页继续)

(续上页)

```
    ,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,
    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;
}
```

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。

44.5.2.2.4 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 3: 代码清单 46-4 MPU6050 的寄存器定义 (mpu6050.h)

```
// MPU6050, Standard address 0xD0
#define MPU6050_ADDRESS          0xD0
#define MPU6050_WHO_AM_I         0x75
#define MPU6050_SMPLRT_DIV       0   //8000Hz
#define MPU6050_DLPF_CFG         0
```

(下页继续)

(续上页)

```
#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] XG_OFFS_TC,
[0] OTP_BNK_VLD
#define MPU6050_RA_YG_OFFS_TC       0x01 //[7] PWR_MODE, [6:1] YG_OFFS_TC,
[0] OTP_BNK_VLD
#define MPU6050_RA_ZG_OFFS_TC       0x02 //[7] PWR_MODE, [6:1] ZG_OFFS_TC,
[0] OTP_BNK_VLD
#define MPU6050_RA_X_FINE_GAIN      0x03 //[7:0] X_FINE_GAIN
/*......... 以下部分省略 */
```

44.5.2.2.5 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 4: 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)
{
    //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
    Delay(100);
    //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
    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);
    Delay(200);
}
```

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)。

44.5.2.2.6 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 5: 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;
   }
}
```

44.5.2.2.7 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 6: 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
*/
```

(下页继续)

(续上页)

```
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];
    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
```

(下页继续)

(续上页)

```
    *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。

44.5.2.2.8 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 7: 代码清单 46-8 main 函数

```
/* 简单任务管理 */
uint32_t Task_Delay[NumOfTask]= {0};

/**
   * @brief  主函数
   * @param  无
   * @retval 无
   */
int main(void)
{

   short Acel[3];
   short Gyro[3];
   float Temp;

   /* 系统时钟初始化成 168MHz */
   SystemClock_Config();
   LED_GPIO_Config();
#ifdef USE_LCD_DISPLAY
   /* 初始化液晶屏 */
   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 );
#endif

   /* 初始化 USART1*/
   DEBUG_USART_Config();
```

(下页继续)

(续上页)

```
    //初始化 I2C
    I2cMaster_Init();
    printf("\r\n 欢迎使用野火  STM32 F407 开发板。\r\n");
    printf("\r\n 这是一个 I2C 外设 (AT24C02) 读写测试例程\r\n");
    //MPU6050 初始化
    MPU6050_Init();
    //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);
#ifdef USE_LCD_DISPLAY
                {
                      char cStr [ 70 ];
                      sprintf ( cStr, "Acc  :%6d%6d%6d",Acel[0],Acel[1],
                               Acel[2] );  //加速度原始数据

                      ILI9806G_DispStringLine_EN(LINE(7),cStr);

                      sprintf ( cStr, "Gyro :%6d%6d%6d",Gyro[0],Gyro[1],
                               Gyro[2] );  //角原始数据
```

(下页继续)

(续上页)

```
                    ILI9806G_DispStringLine_EN(LINE(8),cStr);

                    sprintf ( cStr, "Tem  :%6.2f",Temp ); //温度值
                    ILI9806G_DispStringLine_EN(LINE(9),cStr);
                }
#endif
                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;
#ifdef USE_LCD_DISPLAY
        /* 设置字体颜色及字体的背景颜色 */
        LCD_SetColors(BLUE,BLACK);
        //野火自带的 16*24 显示
        ILI9806G_DispStringLine_EN(LINE(4),"No MPU6050 detected! ");
        //野火自带的 16*24 显示
        ILI9806G_DispStringLine_EN(LINE(5),"Please check the hardware␣
↪connection! ");
#endif
        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。

44.5.2.3 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。

44.6 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。

44.6.1 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。

44.6.2 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。

44.6.2.1 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。

44.6.2.2 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。

44.6.2.2.1 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 8: 代码清单 46-9 I2C 读写接口 (inv_mpu.c 文件)

```
/* The following functions must be defined for this platform:
* 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,
```

(下页继续)

(续上页)

```
*       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。

44.6.2.2.2 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 9: 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
#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 10: 代码清单 46-11 使用 Systick 进行定时（bsp_SysTick.c）

```
/* @brief  获取当前毫秒值
* @param  存储最新毫秒值的变量
```

(下页继续)

(续上页)

```
* @retval 无
*/
int get_tick_count(unsigned long *count)
{
    count[0] = HAL_GetTick();
    return 0;
}
```

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 11: 代码清单 46-12 Systick 的中断服务函数 (stm32f1xx_it.c 文件)

```
void SysTick_Handler(void)
{
   uint8_t i;
   HAL_IncTick();
   for (i=0; i<NumOfTask; i++) {
      if (Task_Delay[i]) {
            Task_Delay[i]--;
      }
   }
}
```

44.6.2.2.3 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 12: 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 */
   HAL_UART_Transmit(&UartHandle, (uint8_t *)&ch, 1, 1000);
   return (ch);
}

/* 输出四元数数据 */
```

(下页继续)

(续上页)

```
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);
    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。

44.6.2.2.4 MPU6050 的中断接口

与我们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 13: 代码清单 46-14 中断接口 (stm32f1xx_it.c 文件)

```
#define MPU_IRQHandler                          EXTI9_5_IRQHandler

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 14: 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
```

(下页继续)

(续上页)

```
*/

void gyro_data_ready_cb(void)

{
    hal.new_gyro = 1;
}
```

44.6.2.2.5 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 15: 代码清单 46-16 使用 MPL 进行姿态解算的过程

```
/**
    * @brief main entry point.
    * @par Parameters None
    * @retval void None
    * @par Required preconditions: None
    */

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;
    /* 系统时钟初始化成 168 MHz */
    SysTick_Init();
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();
    /* 初始化 USART1*/
    Debug_USART_Config();
    EXTI_MPU_Config();
    // Configure I2C
    I2cMaster_Init();

    result = mpu_init(&int_param);
    if (result) {
          MPL_LOGE("Could not initialize gyro.\n");
          LED_RED;
    } else {
          LED_GREEN;
    }

    result = inv_init_mpl();
    if (result) {
          MPL_LOGE("Could not initialize MPL.\n");
    }

    /* Compute 6-axis and 9-axis quaternions. */
    inv_enable_quaternion();
    inv_enable_9x_sensor_fusion();

    /* Update gyro biases when not in motion.
       * WARNING: These algorithms are mutually exclusive.
       */
    inv_enable_fast_nomot();
    /* inv_enable_motion_no_motion(); */
    /* inv_set_no_motion_time(1000); */

    /* Update gyro biases when temperature changes. */
```

(下页继续)

(续上页)

```
    inv_enable_gyro_tc();

    /* Allows use of the MPL APIs in read_from_mpl. */
    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");
    }

    /* Get/set hardware configuration. Start gyro. */
    /* Wake up all sensors. */

    mpu_set_sensors(INV_XYZ_GYRO | INV_XYZ_ACCEL);
    /* Push both gyro and accel data into the FIFO. */
    mpu_configure_fifo(INV_XYZ_GYRO | INV_XYZ_ACCEL);
    mpu_set_sample_rate(DEFAULT_MPU_HZ);

    /* Read back configuration in case it was set improperly. */
    mpu_get_sample_rate(&gyro_rate);
    mpu_get_gyro_fsr(&gyro_fsr);
    mpu_get_accel_fsr(&accel_fsr);

    /* Sync driver configuration with MPL. */
    /* Sample rate expected in microseconds. */
    inv_set_gyro_sample_rate(1000000L / gyro_rate);
    inv_set_accel_sample_rate(1000000L / gyro_rate);
```

(下页继续)

(续上页)

```
/* Set chip-to-body orientation matrix.
* Set hardware units to dps/g's/degrees scaling factor.
*/
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);

/* Initialize HAL state variables. */

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;

/* Compass reads are handled by scheduler. */
get_tick_count(&timestamp);

/* To initialize the DMP:
* 1. Call dmp_load_motion_driver_firmware(). This pushes the DMP image in
   *    inv_mpu_dmp_motion_driver.h into the MPU memory.
* 2. Push the gyro and accel orientation matrix to the DMP.
   * 3. Register gesture callbacks. Don't worry, these callbacks won't be
   *    executed unless the corresponding feature is enabled.
   * 4. Call dmp_enable_feature(mask) to enable different features.
   * 5. Call dmp_set_fifo_rate(freq) to select a DMP output rate.
```

(下页继续)

(续上页)

```
     * 6. Call any feature-specific control functions.
     *
     * To enable the DMP, just call mpu_set_dmp_state(1). This function can
     * be called repeatedly to enable and disable the DMP at runtime.
     *
     * The following is a short summary of the features supported in the␣
↪DMP
     * image provided in inv_mpu_dmp_motion_driver.c:
     * DMP_FEATURE_LP_QUAT: Generate a gyro-only quaternion on the DMP at
     * 200Hz. Integrating the gyro data at higher rates reduces numerical
     * errors (compared to integration on the MCU at a lower sampling␣
↪rate).
   * DMP_FEATURE_6X_LP_QUAT: Generate a gyro/accel quaternion on the DMP at
     * 200Hz. Cannot be used in combination with DMP_FEATURE_LP_QUAT.
     * DMP_FEATURE_TAP: Detect taps along the X, Y, and Z axes.
     * DMP_FEATURE_ANDROID_ORIENT: Google's screen rotation algorithm.␣
↪Triggers
     * an event at the four orientations where the screen should rotate.
     * DMP_FEATURE_GYRO_CAL: Calibrates the gyro data after eight seconds␣
↪of
     * no motion.
     * DMP_FEATURE_SEND_RAW_ACCEL: Add raw accelerometer data to the FIFO.
     * DMP_FEATURE_SEND_RAW_GYRO: Add raw gyro data to the FIFO.
     * DMP_FEATURE_SEND_CAL_GYRO: Add calibrated gyro data to the FIFO.␣
↪Cannot
     * be used in combination with DMP_FEATURE_SEND_RAW_GYRO.
     */
  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);
```

(下页继续)

(续上页)

```
   /*
       * Known Bug -
       * DMP when enabled will sample sensor data at 200Hz and output to␣
↪FIFO at the rate
       * specified in the dmp_set_fifo_rate API. The DMP will then sent an␣
↪interrupt once
    * a sample has been put into the FIFO. Therefore if the dmp_set_fifo_
↪rate is at 25Hz
       * there will be a 25Hz interrupt from the MPU device.
       *
       * There is a known issue in which if you do not enable DMP_FEATURE_TAP
       * then the interrupts will be at 200Hz even if fifo rate
       * is set at a different rate. To avoid this issue include the DMP_
↪FEATURE_TAP
       *
       * DMP sensor fusion works only with gyro at +-2000dps and accel +-2G
       */
   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);
   hal.dmp_on = 1;

   while (1) {

         unsigned long sensor_timestamp;
         int new_data = 0;
         if (USART_GetFlagStatus(DEBUG_USART, USART_FLAG_RXNE)) {
      /* A byte has been received via USART. See handle_input for a list of
```

(下页继续)

(续上页)

```
            * valid commands.
            */
            USART_ClearFlag(DEBUG_USART, USART_FLAG_RXNE);
            handle_input();
        }
        get_tick_count(&timestamp);

        /* Temperature data doesn't need to be read with every gyro sample.
        * Let's make them timer-based like the compass reads.
        */
        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;
            /* This function gets new data from the FIFO when the DMP is in
            * use. The FIFO can contain any combination of gyro, accel,
            * quaternion, and gesture data. The sensors parameter tells the
            * caller which data fields were actually populated with new␣
↪data.
            * For example, if sensors == (INV_XYZ_GYRO | INV_WXYZ_QUAT),␣
↪then
            * the FIFO isn't being filled with accel data.
            * The driver parses the gesture data to determine if a gesture
            * event has occurred; on an event, the application will be␣
↪notified
            * via a callback (assuming that a callback function was properly
            * registered). The more parameter is non-zero if there are
```

(下页继续)

(续上页)

```
            * leftover packets in the FIFO.
            */
      dmp_read_fifo(gyro, accel_short, quat, &sensor_timestamp, &sensors, &
↪more);
            if (!more)
               hal.new_gyro = 0;
            if (sensors & INV_XYZ_GYRO) {
               /* Push the new data to the MPL. */
               inv_build_gyro(gyro, sensor_timestamp);
               new_data = 1;
               if (new_temp) {
                     new_temp = 0;
                     /* Temperature only used for gyro temp comp. */
                     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();
```

(下页继续)

(续上页)

```
            /* This function reads bias-compensated sensor data and sensor
            * fusion outputs from the MPL. The outputs are formatted as seen
            * in eMPL_outputs.c. This function only needs to be called at␣
↪the
            * rate requested by the host.
            */
            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。

44.6.2.2.6 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 16: 代码清单 46-17 MPL 的数据输出接口 (main.c)

```
/* Get data from MPL.
* TODO: Add return values to the inv_get_sensor_type_xxx APIs to␣
↪differentiate
* between new and stale data.
*/
static void read_from_mpl(void)
{
    long msg, data[9];
    int8_t accuracy;
    unsigned long timestamp;
    float float_data[3] = {0};

    if (inv_get_sensor_type_quat(data, &accuracy, (inv_time_t*)&timestamp))
```

(下页继续)

(续上页)

```
        /* Sends a quaternion packet to the PC. Since this is used by the␣
↪Python
         * test app to visually represent a 3D quaternion, it's sent each␣
↪time
         * the MPL has new data.
         */
        eMPL_send_quat(data);

/* Specific data packets can be sent or suppressed using USB commands. */
        if (hal.report & PRINT_QUAT)
            eMPL_send_data(PACKET_DATA_QUAT, data);
    }

    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);
    }
```

(下页继续)

(续上页)

```
   /******************* 使用液晶屏显示数据 ************************/
   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)) {

#ifdef USE_LCD_DISPLAY
            //inv_get_sensor_type_euler 读出的数据是 Q16 格式，所以左移 16 位.
            sprintf ( cStr, "Pitch :  %.4f  ", data[0]*1.0/(1<<16) );
            LCD_DisplayStringLine(7,(uint8_t* )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) );
            LCD_DisplayStringLine(8,(uint8_t* )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) );
            LCD_DisplayStringLine(9,(uint8_t* )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) );
            LCD_DisplayStringLine(10,(uint8_t* )cStr);
#endif

         }

         /* 获取步数 */
         get_tick_count(&timestamp);
```

(下页继续)

(续上页)

```
        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);

#ifdef USE_LCD_DISPLAY
   sprintf(cStr, "Walked steps :  %ld  steps over  %ld  milliseconds..",
↪step_count,walk_time);
            LCD_DisplayStringLine(11,(uint8_t* )cStr);
#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。

44.6.2.3 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。

44.6.2.3.1 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

44.6.2.3.2 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

44.6.2.3.3 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。)

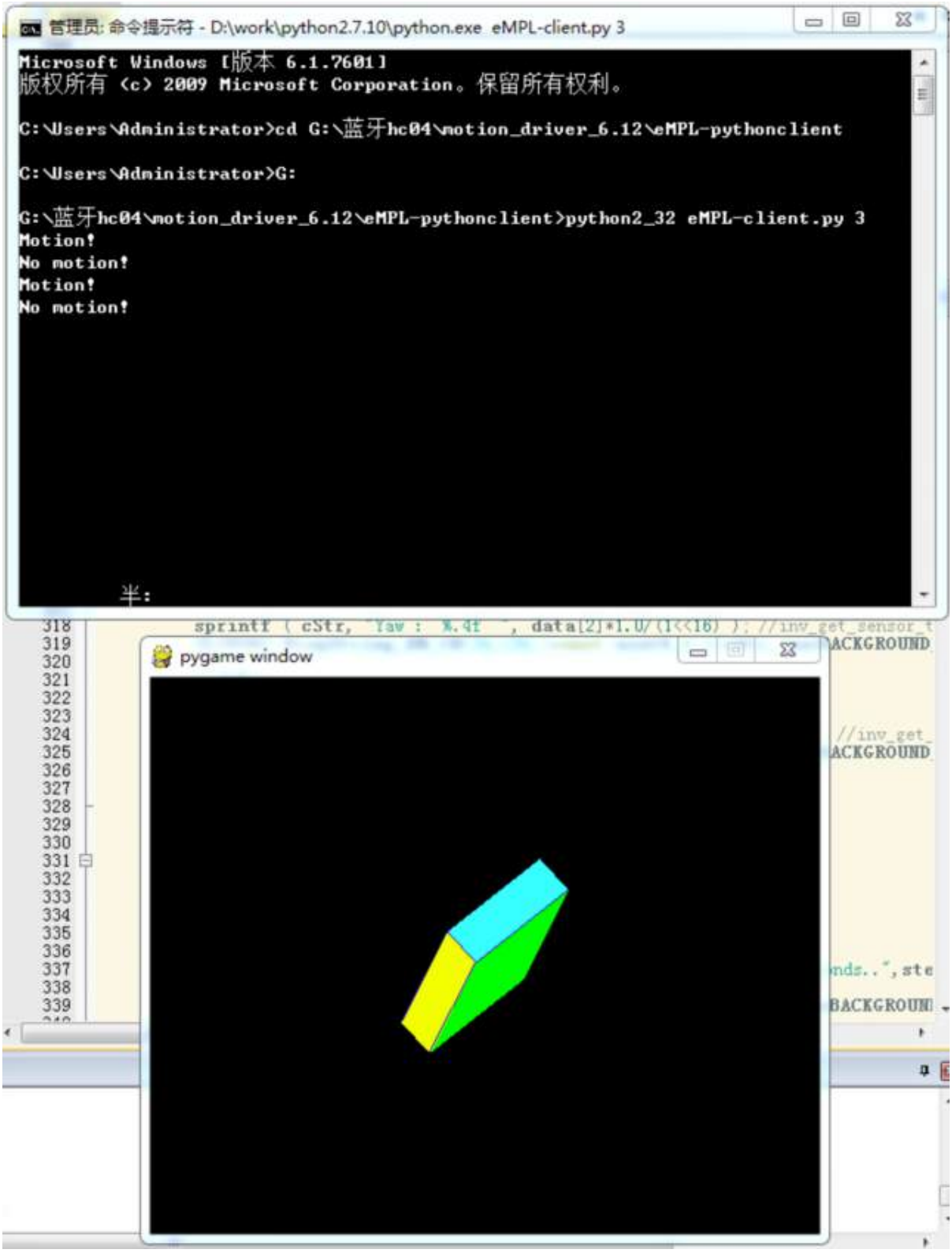

管理员: 命令提示符 - D:\work\python2.7.10\python.exe eMPL-client.py 3
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
C:\Users\Administrator>cd G:\蓝牙hc04\motion_driver_6.12\eMPL-pythonclient
C:\Users\Administrator>G:
G:\蓝牙hc04\motion_driver_6.12\eMPL-pythonclient>python2_32 eMPL-client.py 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

44.7 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。

44.7.1 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。

44.7.2 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。

44.7.2.1 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；

44.7.2.2 代码分析

44.7.2.2.1 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。

44.7.2.2.2 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 17: 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;
```

(下页继续)

(续上页)

```
u8 data_to_send[50];

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++)
```

(下页继续)

(续上页)

```
      usart_send_char(data_to_send[i]);
}

/* 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]);
```

(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[_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。

44.7.2.2.3 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 18: 代码清单 46-19 read_from_mpl 函数 (main.c 文件)

```
extern struct inv_sensor_cal_t sensors;
```

(下页继续)

(续上页)

```
/* Get data from MPL.
* TODO: Add return values to the inv_get_sensor_type_xxx APIs to␣
↪differentiate
* between new and stale data.
*/
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。

44.7.2.3 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。

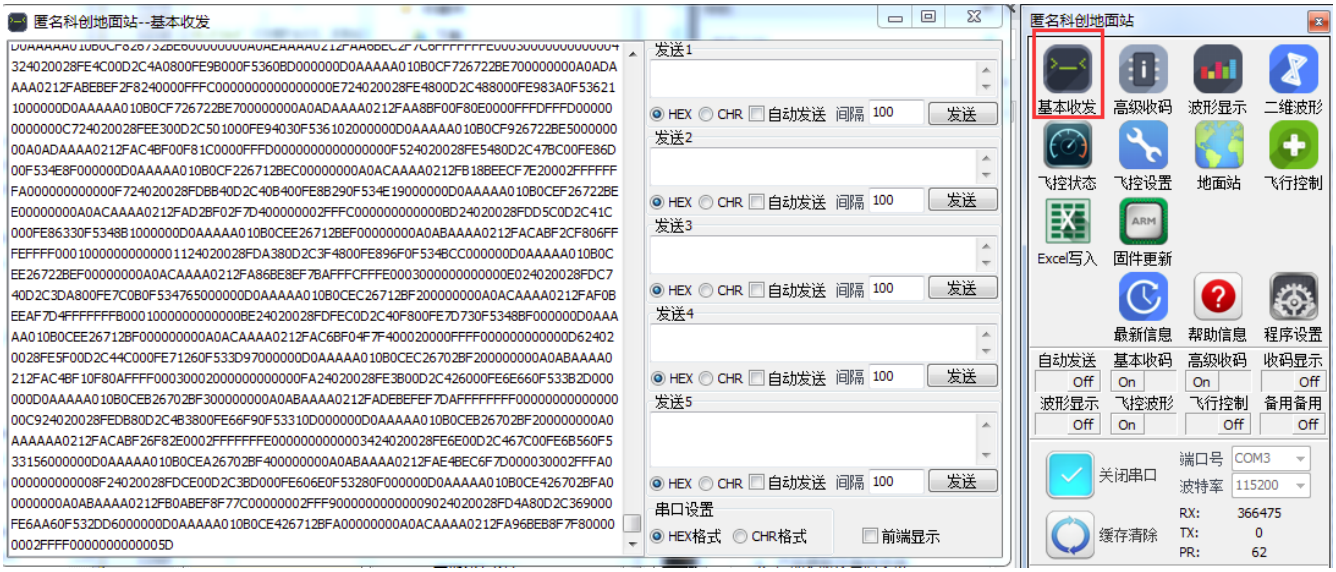
实验步骤如下：

(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
ARM
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
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
缓存清除
端口号 COM3
波特率 115200
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
缓存清除
端口号 COM3
波特率 115200
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 45 章　DCMI—OV2640 摄像头

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

关于开发板配套的 OV2640 摄像头参数可查阅《ov2640datasheet》配套资料获知。

STM32F4 芯片具有浮点运算单元，适合对图像信息使用 DSP 进行基本的图像处理，其处理速度比传统的 8、16 位机快得多，而且它还具有与摄像头通讯的专用 DCMI 接口，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。本章讲解如何使用 STM32 驱动 OV2640 型号的摄像头。

45.1 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。

45.1.1 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 41 万左右。现在的数字摄像头分辨率一般从数十万到数千万。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。

45.1.2 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。

45.2 OV2640 摄像头

本章主要讲解实验板配套的摄像头，它的实物见图 *45_0_1*，该摄像头主要由镜头、图像传感器、板载电路及下方的信号引脚组成。

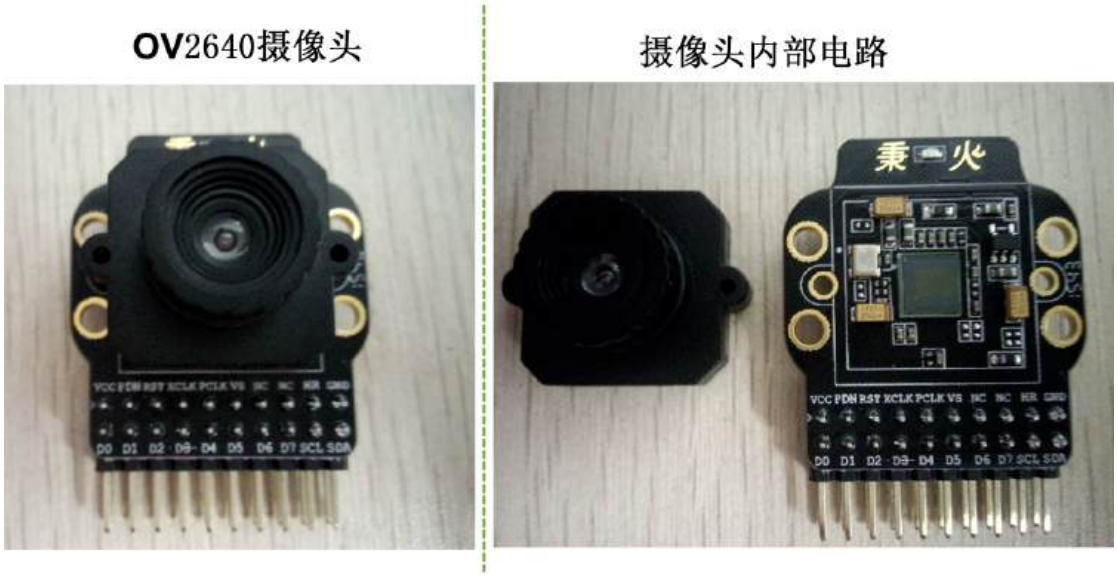


图 45-0-1 实验板配套的 OV2640 摄像头

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，然后把采集得的数据通过下方的信号引脚输出数据到外部器件。

45.2.1 OV2640 传感器简介

图像传感器是摄像头的核心部件，上述摄像头中的图像传感器是一款型号为 OV2640 的 CMOS 类型数字图像传感器。该传感器支持输出最大为 200 万像素的图像 (1600x1200 分辨率)，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、RGB565 以及 JPEG 格式，若直接输出 JPEG 格式的图像时可大大减少数据量，方便网络传输。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。根据不同的分辨率配置，传感器输出图像数据的帧率从 15-60 帧可调，工作时功率在 125mW-140mW 之间。

45.2.2 OV2640 引脚及功能框图

OV2640 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 *45_0_2*。

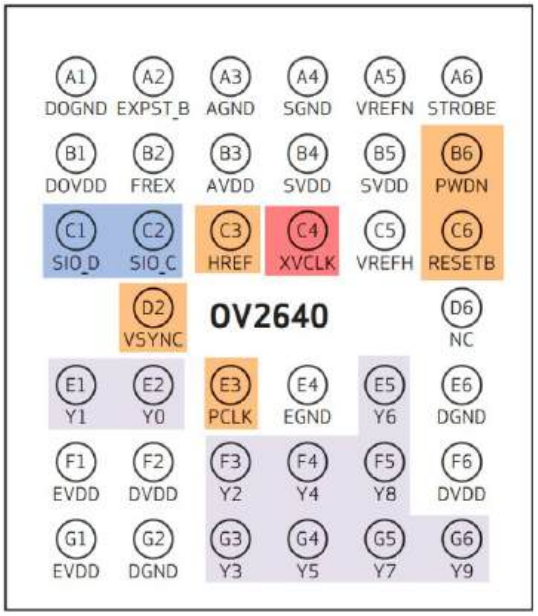


图 45-0-2 OV2640 传感器引脚分布图

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，介绍如下表 45-0-1。

表 45-0-1 OV2640 管脚

管脚名称	管脚类型	管脚描述
SIO_C	输入	SCCB 总线的时钟线，可类比 I2C 的 SCL
SIO_D	I/O	SCCB 总线的数据线，可类比 I2C 的 SDA
RESETB	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	帧同步信号
PCLK	输出	像素同步时钟输出信号
XCLK	输入	外部时钟输入端口，可接外部晶振
Y0…Y9	输出	像素数据输出端口

下面我们配合图 *45_0_3* 中的 OV2640 功能框图讲解这些信号引脚。

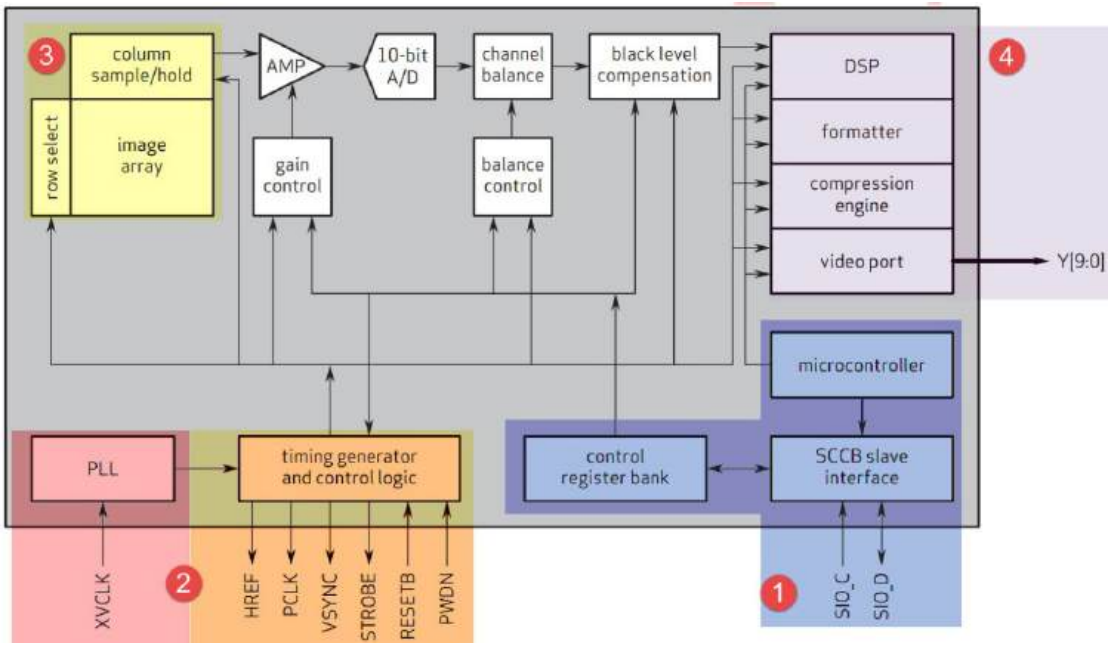


图 45-0-3 OV2640 功能框图

(1) 控制寄存器

标号处的是 OV2640 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 SIO_C 和 SIO_D 引脚写入的，SIO_C 与 SIO_D 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。

(1) 通信、控制信号及时钟

标号处包含了 OV2640 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信号是很类似的。RESETB 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 OV2640 的信号；而 PCLK 是 OV2640 输出数据时的同步信号，它是由 OV2640 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 OV2640 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。

(1) 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。

(1) 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 Y0-Y9 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 Y2-Y9 引脚，OV2640 与外部器件的连接方式见图 *45_0_4* 。

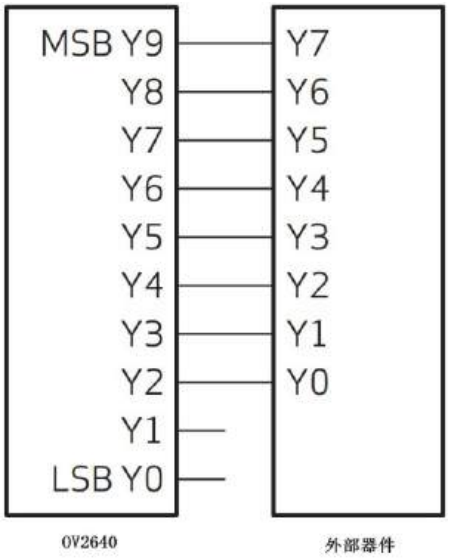


图 45-0-4 8 位数据线接法

45.2.3 SCCB 时序

外部控制器对 OV2640 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 驱动中我们直接使用片上 I2C 外设与它通讯。SCCB 与标准的 I2C 协议的区别是它每次传输只能写入或读取一个字节的数据，而 I2C 协议是支持突发读写的，即在一次传输中可以写入多个字节的数据 (EEPROM 中的页写入时序即突发写)。关于 SCCB 协议的完整内容可查看配套资料里的《SCCB 协议》文档，下面我们简单介绍下。

45.2.3.1 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 45-5 及图 45-6。

- 起始信号：在 SIO_C 为高电平时，SIO_D 出现一个下降沿，则 SCCB 开始传输。
- 停止信号：在 SIO_C 为高电平时，SIO_D 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 SIO_C 为高电平时，必须保证 SIO_D 上的数据稳定，也就是说，SIO_D 上的电平变换只能发生在 SIO_C 为低电平的时候，SIO_D 的信号在 SIO_C 为高电平时被采集。

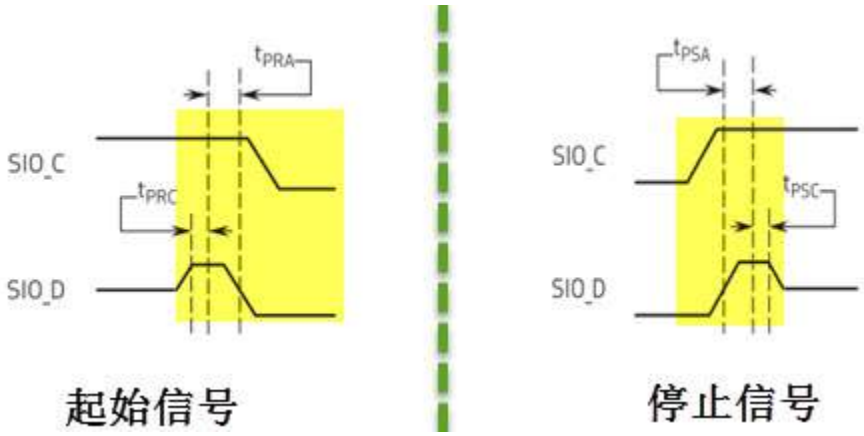


图 45-0-5 SCCB 停止信号

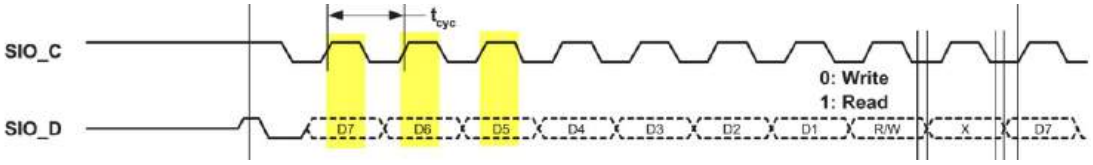


图 45-0-6 SCCB 的数据有效性

45.2.3.2 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 *45_0_7*。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。

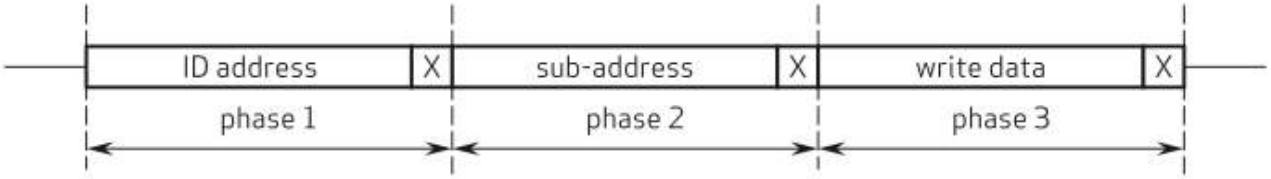


图 45-0-7 SCCB 的三步写操作

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 45-8。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。

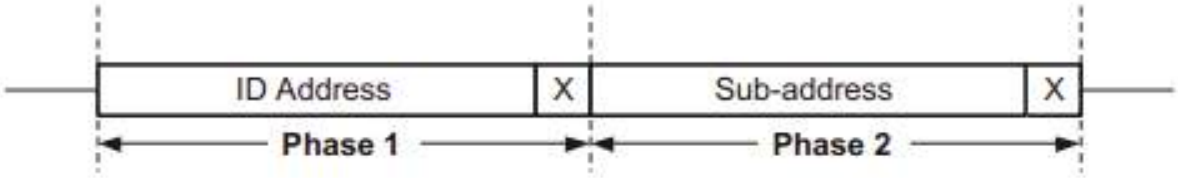


图 45-0-8 SCCB 的两步写操作

两步读操作，它用于读取从设备目的寄存器中的数据，见图 *45_0_9*。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。

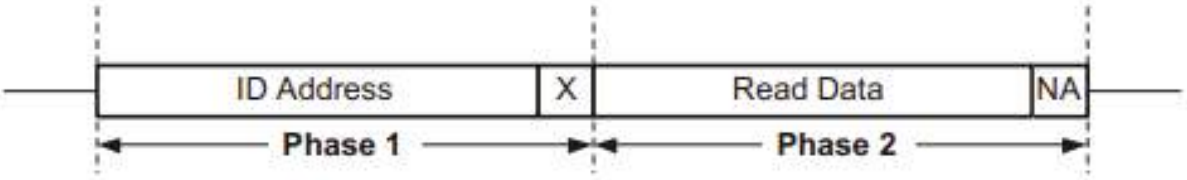


图 45-0-9 SCCB 的两步读操作

可以看到，以上介绍的 SCCB 特性都与 I2C 无区别，而 I2C 比 SCCB 还多出了突发读写的功能，所以 SCCB 可以看作是 I2C 的子集，我们完全可以使用 STM32 的 I2C 外设来与 OV2640 进行 SCCB 通讯。

45.2.4 OV2640 的寄存器

控制 OV2640 涉及到它很多的寄存器，可直接查询《ov2640datasheet》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、图像格式及图像方向等。要注意的是 OV2640 有两组寄存器，这两组寄存器有部分地址重合，通过设置地址为 0xFF 的 RA_DLMT 寄存器可以切换寄存器组，当 RA_DLMT 寄存器为 0 时，通过 SCCB 发送的寄存器地址在 DSP 相关的寄存器组寻址，见图 *45_0_10*；RA_DLMT 寄存器为 1 时，在 Sensor 相关的寄存器组寻址。

Table 12　Device Control Register List (when 0xFF = 00) (Sheet 1 of 4)

Address (Hex)	Register Name	Default (Hex)	R/W	Description
00-04	RSVD	XX	–	Reserved
05	R_BYPASS	0x1	RW	Bypass DSP Bit[7:1]: Reserved Bit[0]: Bypass DSP select 0: DSP 1: Bypass DSP, sensor out directly
06-43	RSVD	XX	–	Reserved
44	Qs	0C	RW	Quantization Scale Factor
45-4F	RSVD	XX	–	Reserved

图 45-0-10 0xFF=0 时的 DSP 相关寄存器说明 (部分)

Table 13　Device Control Register List (when 0xFF = 01) (Sheet 1 of 6)

Address (Hex)	Register Name	Default (Hex)	R/W	Description
00	GAIN	00	RW	AGC Gain Control LSBs Bit[7:0]: Gain setting • Range: 1x to 32x Gain = (Bit[7]+1) x (Bit[6]+1) x (Bit[5]+1) x (Bit[4]+1) x (1+Bit[3:0]/16) ***Note:*** *Set COM8[2] = 0 to disable AGC.*
01-02	RSVD	XX	–	Reserved
03	COM1	0F (UXGA) 0A (SVGA), 06 (CIF)	RW	Common Control 1 Bit[7:6]: Dummy frame control 00: Reserved 01: Allow 1 dummy frame 10: Allow 3 dummy frames 11: Allow 7 dummy frames Bit[5:4]: Reserved Bit[3:2]: Vertical window end line control 2 LSBs (8 MSBs in VEND[7:0] (0x1A)) Bit[1:0]: Vertical window start line control 2 LSBs (8 MSBs in VSTRT[7:0] (0x19))

图 45-0-11 0xFF=1 时的 Sensor 相关寄存器说明 (部分)

官方还提供了一个《OV2640_Camera_app》的文档，它针对不同的配置需求，提供了配置范例，见图 *45_0_12*。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。

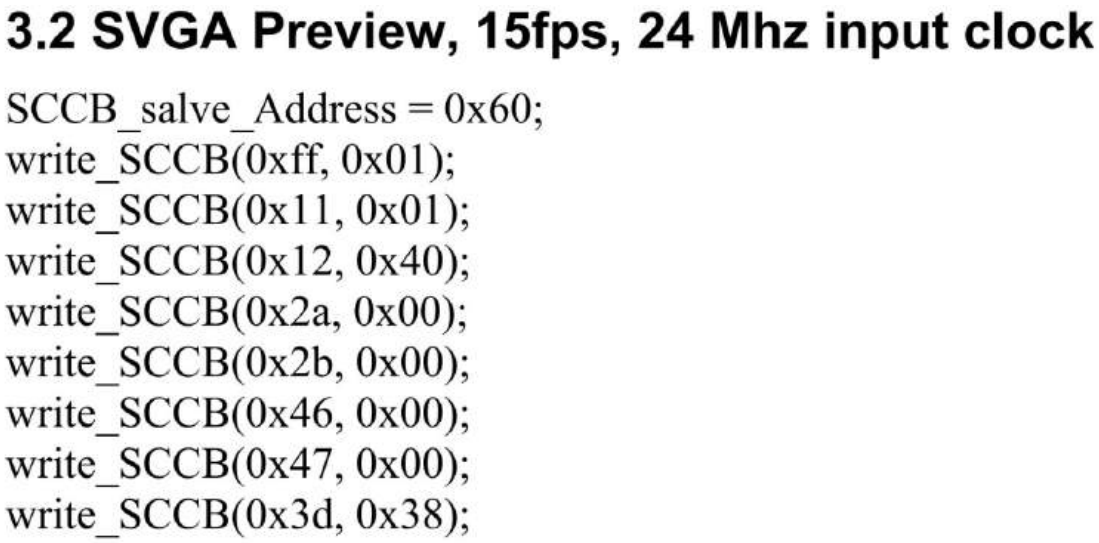


图 45-0-12 调节帧率的寄存器配置范例

45.2.5 像素数据输出时序

主控器控制 OV2640 时采用 SCCB 协议读写其寄存器，而它输出图像时则使用 VGA 时序 (还可用 SVGA、UXGA，这些时序都差不多)，这跟控制液晶屏输入图像时很类似。OV2640 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 *45_0_13*。

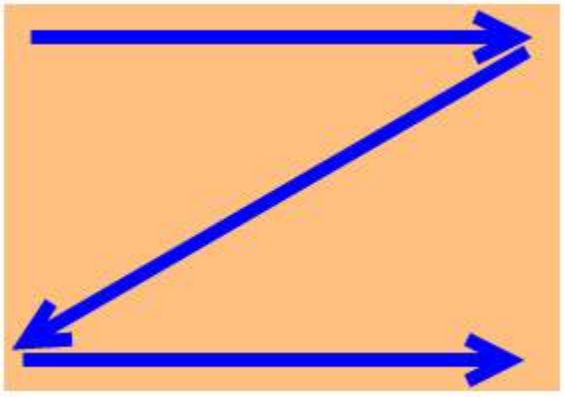


图 45-0-13 摄像头数据输出

例如，见图 *45_0_14* 和图 *45_0_15*，若我们使用 Y2-Y9 数据线，图像格式设置为 RGB565，进行数据输出时，Y2-Y9 数据线会在 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。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，每传输完一帧图像时，VSYNC 会输出一个电平跳变信号。

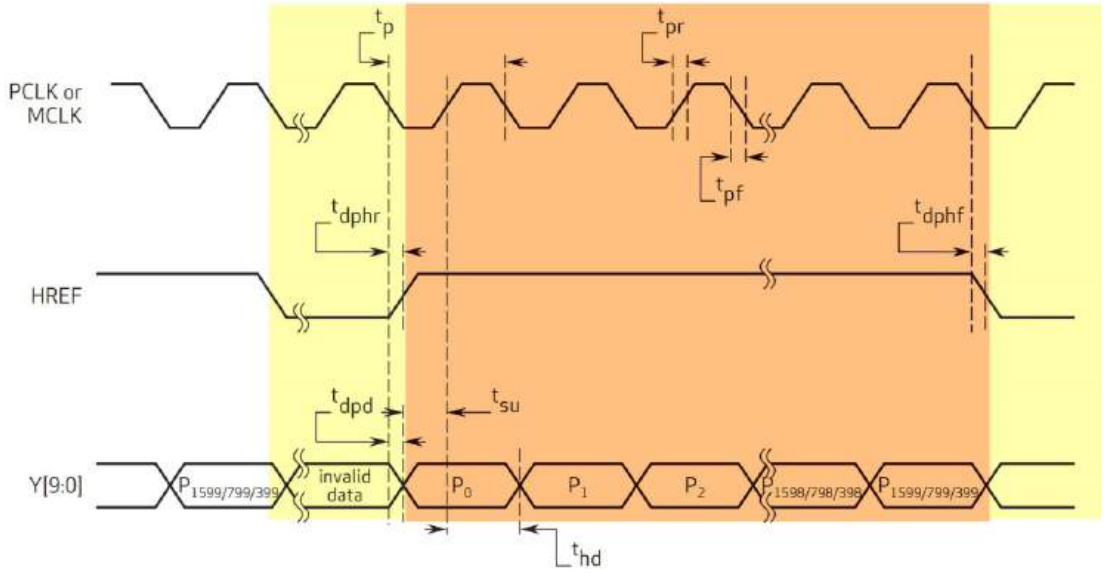


图 45-0-14 像素同步时序

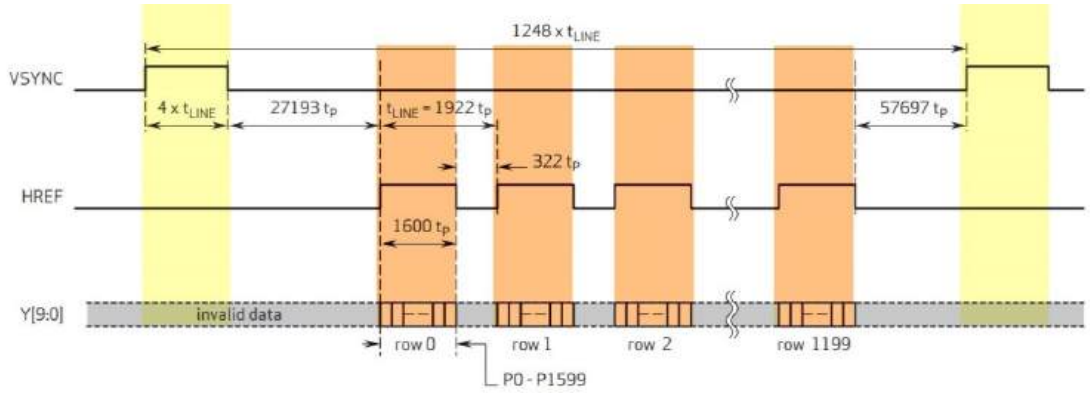


图 45-0-15 帧图像同步时序

45.3 STM32 的 DCMI 接口简介

STM32F4 系列的控制器包含了 DCMI 数字摄像头接口 (Digital camera Interface)，它支持使用上述类似 VGA 的时序获取图像数据流，支持原始的按行、帧格式来组织的图像数据，如 YUV、RGB，也支持接收 JPEG 格式压缩的数据流。接收数据时，主要使用 HSYNC 及 VSYNC 信号来同步。

45.3.1 DCMI 整体框图

STM32 的 DCMI 接口整体框图见图 *45_0_16*。

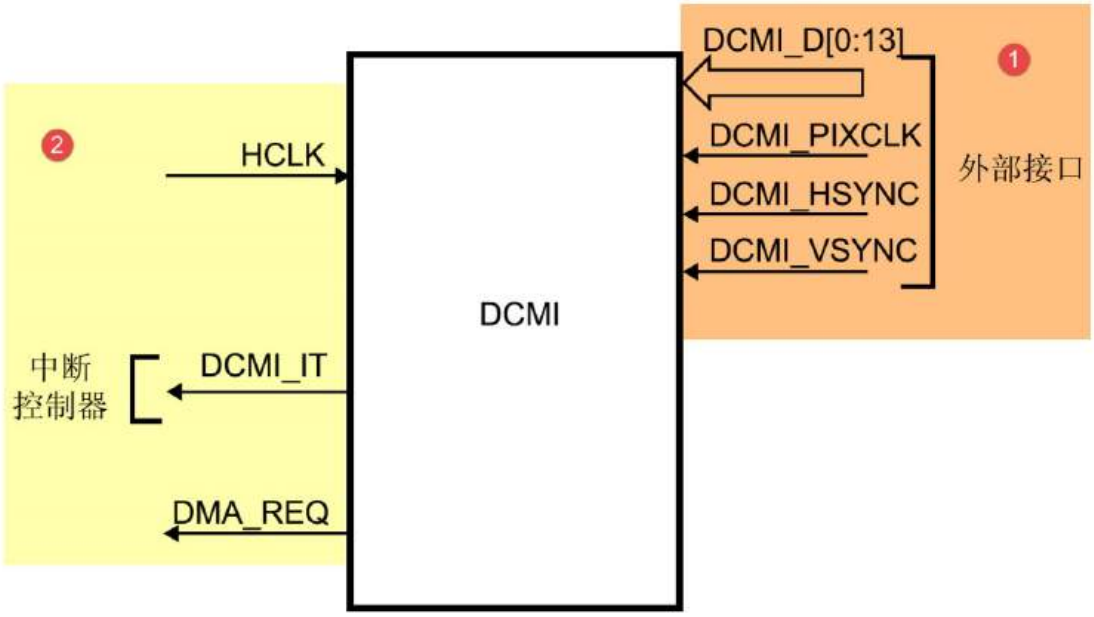


图 45-0-16 DCMI 接口整体框图

45.3.1.1 外部接口及时序

上图标号处的是 DCMI 向外部引出的信号线。DCMI 提供的外部接口的方向都是输入的，接口的各个信号线说明见表 45-0-2。

表 45-0-2 DCMI 的信号线说明

引脚名称	说明
DCMI_D[0:13]	数据线
DCMI_PIXCLK	像素同步时钟
DCMI_HSYNC	行同步信号 (水平同步信号)
DCMI_VSYNC	帧同步信号 (垂直同步信号)

其中 DCMI_D 数据线的数量可选 8、10、12 或 14 位，各个同步信号的有效极性都可编程控制。它使用的通讯时序与 OV2640 的图像数据输出接口时序一致，见图 *45_0_17*。

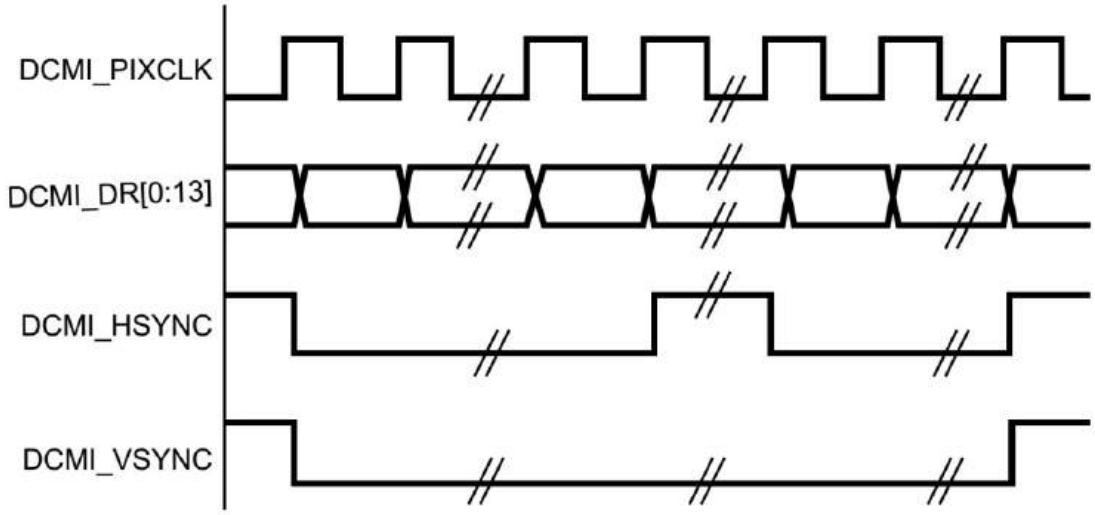


图 45-0-17 DCMI 时序图

45.3.1.2 内部信号及 PIXCLK 的时钟频率

图 *45_0_16* 的标号处表示 DCMI 与内部的信号线。在 STM32 的内部，使用 HCLK 作为时钟源提供给 DCMI 外设。从 DCMI 引出有 DCMI_IT 信号至中断控制器，并可通过 DMA_REQ 信号发送 DMA 请求。

DCMI 从外部接收数据时，在 HCLK 的上升沿时对 PIXCLK 同步的信号进行采样，它限制了 PIXCLK 的最小时钟周期要大于 2.5 个 HCLK 时钟周期，即最高频率为 HCLK 的 1/4。

45.3.2 DCMI 接口内部结构

DCMI 接口的内部结构见图 *45_0_18*。

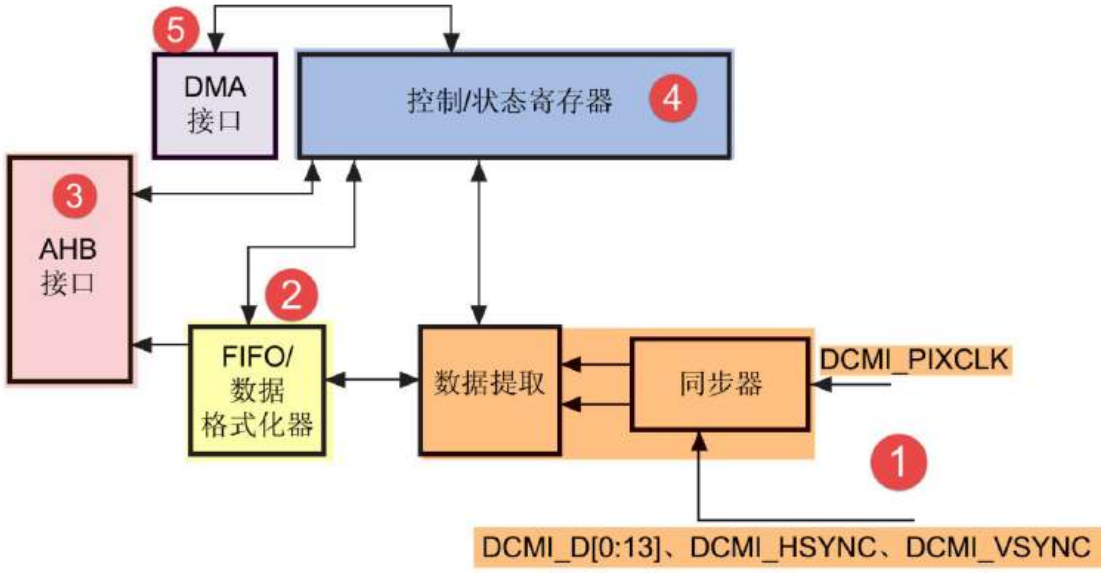


图 45-0-18 DCMI 接口内部结构

(1) 同步器

同步器主要用于管理 DCMI 接收数据的时序，它根据外部的信号提取输入的数据。

(1) FIFO/数据格式化器

为了对数据传输加以管理，STM32 在 DCMI 接口上实现了 4 个字 (32bit x4) 深度的 FIFO，用以缓冲接收到的数据。

(1) AHB 接口

DCMI 接口挂载在 AHB 总线上，在 AHB 总线中有一个 DCMI 接口的数据寄存器，当我们读取该寄存器时，它会从 FIFO 中获取数据，并且 FIFO 中的数据指针会自动进行偏移，使得我们每次读取该寄存器都可获得一个新的数据。

(1) 控制/状态寄存器

DCMI 的控制寄存器协调图中的各个结构运行，程序中可通过检测状态寄存器来获 DCMI 的当前运行状态。

(1) DMA 接口

由于 DCMI 采集的数据量很大，我们一般使用 DMA 来把采集得的数据搬运至内存。

45.3.3 同步方式

DCMI 接口支持硬件同步或内嵌码同步方式，硬件同步方式即使用 HSYNC 和 VSYNC 作为同步信号的方式，OV2640 就是使用这种同步时序。

而内嵌码同步的方式是使用数据信号线传输中的特定编码来表示同步信息，由于需要用 0x00 和 0xFF 来表示编码，所以表示图像的数据中不能包含有这两个值。利用这两个值，它扩展到 4 个字节，定义出了 2 种模式的同步码，每种模式包含 4 个编码，编码格式为 0xFF0000XY，其中 XY 的值可通过寄存器设置。当 DCMI 接收到这样的编码时，它不会把这些当成图像数据，而是按照表 45-0-3 中的编码来解释，作为同步信号。

表 45-0-3 两种模式的内嵌码

模式 2 的内嵌码	模式 1 的内嵌码
帧开始 (FS)	有效行开始 (SAV)
帧结束 (FE)	有效行结束 (EAV)
行开始 (LS)	帧间消隐期内的行开始 (SAV)，其中消隐期内的即为无效数据
行结束 (LS)	帧间消隐期内的行结束 (EAV)，其中消隐期内的即为无效数据

45.3.4 捕获模式及捕获率

DCMI 还支持两种数据捕获模式，分别为快照模式和连续采集模式。快照模式时只采集一帧的图像数据，连续采集模式会一直采集多个帧的数据，并且可以通过配置捕获率来控制采集多少数据，如可配置为采集所有数据或隔 1 帧采集一次数据或隔 3 帧采集一次数据。

45.4 DCMI 初始化结构体

与其它外设一样，STM32 的 DCMI 外设也可以使用库函数来控制，其中最主要的配置项都封装到了 DCMI_InitTypeDef 结构体，来这些内容都定义在库文件“stm32f4xx_hal_dcmi.h”及“stm32f4xx_hal_dcmi.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。

DCMI_InitTypeDef 初始化结构体的内容见代码清单 45-0-1。

```
typedef struct {
   uint32_t SynchroMode; /* 选择硬件同步模式还是内嵌码模式 */
   uint32_t PCKPolarity; /* 设置像素时钟的有效边沿 */
   uint32_t VSPolarity;  /* 设置 VSYNC 的有效电平 */
   uint32_t HSPolarity;  /* 设置 HSYNC 的有效边沿 */
   uint32_t CaptureRate; /* 设置图像的采集间隔 */
   uint32_t ExtendedDataMode; /* 设置数据线的宽度 */
   DCMI_CodesInitTypeDef SyncroCode;/* 分隔符设置 */
   uint32_t JPEGMode;    /*JPEG 模式选择 */
   uint32_t ByteSelectMode; /* 配置字节选项模式 */
   uint32_t ByteSelectStart; /* 字节选择开始 */
   uint32_t LineSelectMode;  /* 行选择模式 */
   uint32_t LineSelectStart; /* 行选择选择 */
} DCMI_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) SynchroMode

本成员设置 DCMI 数据的同步模式，可以选择为硬件同步方式 (DCMI_SYNCHRO_HARDWARE) 或内嵌码方式 (DCMI_SYNCHRO_EMBEDDED)。

(1) PCKPolarity

本成员用于配置 DCMI 接口像素时钟的有效边沿，即在该时钟边沿时，DCMI 会对数据线上的信号进行采样，它可以被设置为上升沿有效 (DCMI_PCKPOLARITY_RISING) 或下降沿有效 (DCMI_PCKPOLARITY_FALLING)。

(1) VSPolarity

本成员用于设置 VSYNC 的有效电平，当 VSYNC 信号线表示为有效电平时，表示新的一帧数据传输完成，它可以被设置为高电平有效 (DCMI_VSPolarity_High) 或低电平有效 (DCMI_VSPolarity_Low)。

(1) DCMI_HSPolarity

类似地，本成员用于设置 HSYNC 的有效电平，当 HSYNC 信号线表示为有效电平时，表示新的一行数据传输完成，它可以被设置为高电平有效 (DCMI_VSPOLARITY_HIGH) 或低电平有效 (DCMI_VSPOLARITY_LOW)。

(1) CaptureRate

本成员可以用于设置 DCMI 捕获数据的频率，可以设置为全采集、半采集或 1/4 采集 (DCMI_CR_ALL_FRAME/ DCMI_CR_ALTERNATE_2_FRAME/ DCMI_CR_ALTERNATE_4_FRAME)，在间隔采集的情况下，STM32 的 DCMI 外设会直接按间隔丢弃数据。

(1) ExtendedDataMode

本成员用于设置 DCMI 的数据线宽度，可配置为 8/10/12 及 14 位数据线宽 (DCMI_EXTEND_DATA_8B/10B/12B/14B)。

(1) ExtendedDataMode

本成员用于设置 DCMI 的数据线宽度，可配置为 8/10/12 及 14 位数据线宽 (DCMI_EXTEND_DATA_8B/10B/12B/14B)。

(1) SyncroCode

本成员用于设置 DCMI 的数据线指定行/帧开始分隔符和行/帧结束分隔符的代码。

(1) JPEGMode

本成员用于设置 DCMI 的数据输入模式，可配置为使能或者禁止 JPEG 模式。

(1) ByteSelectMode

本成员用于设置 DCMI 的数据字节的选择，可配置为全部接收（DCMI_BSM_ALL），每隔一个字节接收（DCMI_BSM_OTHER），每四个字节接收一个字节

（DCMI_BSM_ALTERNATE_4），每四个字节接收两个字节（DCMI_BSM_ALTERNATE_2）。

(1) ByteSelectStart

本成员用于设置 DCMI 的数据字节开始选择，可配置为奇数或者偶数。

(1) LineSelectMode

本成员用于设置 DCMI 的行数据的采集，可配置全部采集或者隔行采集。

(1) LineSelectStart

本成员用于设置 DCMI 的行数据字节开始选择，可配置为奇数或者偶数。

配置完这些结构体成员后，我们调用库函数 HAL_DCMI_Init 即可把这些参数写入到 DCMI 的控制寄存器中，实现 DCMI 的初始化。

45.5 DCMI—OV2640 摄像头实验

本小节讲解如何使用 DCMI 接口从 OV2640 摄像头输出的 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“DCMI—OV2640 摄像头”工程配合阅读。

45.5.1 硬件设计

45.5.1.1 摄像头原理图

本实验采用的 OV2640 摄像头实物见图 *45_0_1*，其原理图见图 *45_0_19*。

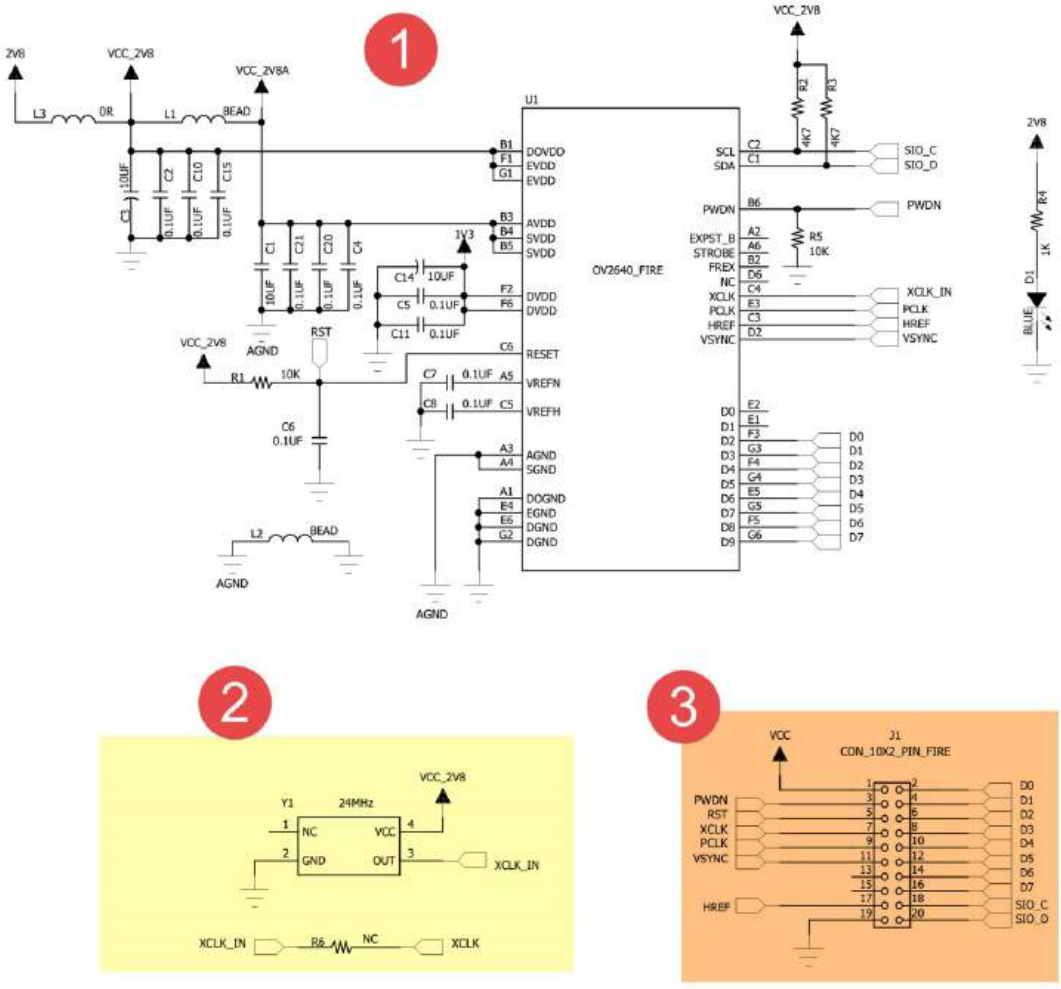


图 45-0-19 OV2640 摄像头原理图

图 *45_0_19* 标号 1 处的是 OV2640 芯片的主电路，在这部分中已对 SCCB 使用的信号线接了上拉电阻，外部电路可以省略上拉；标号处的是一个 24MHz 的有源晶振，它为 OV2640 提供系统时钟，如果不想使用外部晶振提供时钟源，可以参考图中的 R6 处贴上 0 欧电阻，XCLK 引脚引出至外部，由外部控制器提供时钟；标号处的是摄像头引脚集中引出的排针接口，使用它可以方便地与 STM32 实验板中的排母连接。

45.5.1.2 摄像头与实验板的连接

通过排母，OV2640 与 STM32 引脚的连接关系见图 *45_0_20*。

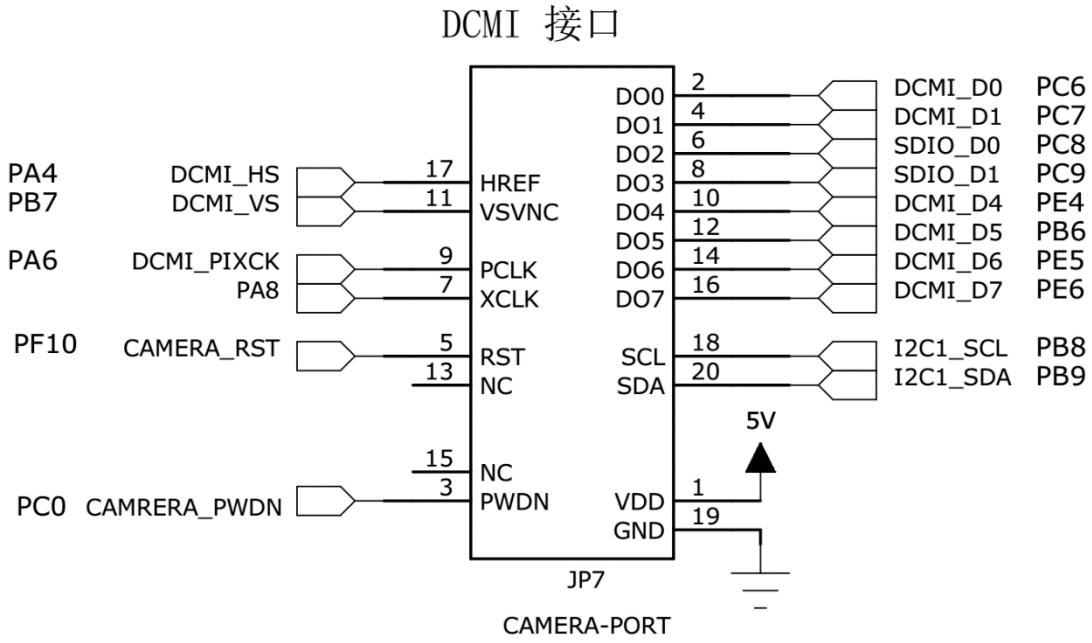


图 45-0-20 STM32 实验板引出的 DCMI 接口

以上原理图可查阅《ov2640—黑白原理图》及《野火 F407 开发板底板原理图》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。

45.5.2 软件设计

为了使工程更加有条理，我们把摄像头控制相关的代码独立分开存储，方便以后移植。在“液晶显示”工程的基础上新建“bsp_ov2640.c”及“bsp_ov2640.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。

45.5.2.1 编程要点

(1) 初始化 DCMI 时钟，I2C 时钟；

(2) 使用 I2C 接口向 OV2640 写入寄存器配置；

(3) 初始化 DCMI 工作模式；

(4) 初始化 DMA，用于搬运 DCMI 的数据到显存空间进行显示；

(5) 编写测试程序，控制采集图像数据并显示到液晶屏。

45.5.2.2 代码分析

45.5.2.2.1 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都以宏的形式定义到“bsp_ov2640.h”文件中，其中包括 I2C 及 DCMI 接口的，见代码清单 45-0-2。

```
/* 摄像头接口 */
//IIC SCCB
#define CAMERA_I2C                          I2C1
#define CAMERA_I2C_CLK_ENABLE()             __HAL_RCC_I2C1_CLK_ENABLE()

#define CAMERA_I2C_SCL_PIN                  GPIO_PIN_8
#define CAMERA_I2C_SCL_GPIO_PORT            GPIOB
#define CAMERA_I2C_SCL_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
#define CAMERA_I2C_SCL_AF                   GPIO_AF4_I2C1

#define CAMERA_I2C_SDA_PIN                  GPIO_PIN_9
#define CAMERA_I2C_SDA_GPIO_PORT            GPIOB
#define CAMERA_I2C_SDA_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
#define CAMERA_I2C_SDA_AF                   GPIO_AF4_I2C1

//VSYNC
```

(下页继续)

(续上页)

```
#define DCMI_VSYNC_GPIO_PORT                GPIOB
#define DCMI_VSYNC_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOB_CLK_ENABLE()
#define DCMI_VSYNC_GPIO_PIN                 GPIO_PIN_7
#define DCMI_VSYNC_AF                       GPIO_AF13_DCMI
// HSYNC
#define DCMI_HSYNC_GPIO_PORT                GPIOA
#define DCMI_HSYNC_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define DCMI_HSYNC_GPIO_PIN                 GPIO_PIN_4
#define DCMI_HSYNC_AF                       GPIO_AF13_DCMI
//PIXCLK
#define DCMI_PIXCLK_GPIO_PORT               GPIOA
#define DCMI_PIXCLK_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOA_CLK_ENABLE()
#define DCMI_PIXCLK_GPIO_PIN                GPIO_PIN_6
#define DCMI_PIXCLK_AF                      GPIO_AF13_DCMI
//PWDN
#define DCMI_PWDN_GPIO_PORT                 GPIOC
#define DCMI_PWDN_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_PWDN_GPIO_PIN                  GPIO_PIN_0

//RST
#define DCMI_RST_GPIO_PORT                  GPIOF
#define DCMI_RST_GPIO_CLK                  __HAL_RCC_GPIOF_CLK_ENABLE()
#define DCMI_RST_GPIO_PIN                   GPIO_PIN_10

//数据信号线
#define DCMI_D0_GPIO_PORT                 GPIOC
#define DCMI_D0_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_D0_GPIO_PIN                  GPIO_PIN_6
#define DCMI_D0_AF                        GPIO_AF13_DCMI

#define DCMI_D1_GPIO_PORT                 GPIOC
#define DCMI_D1_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()
```

(下页继续)

(续上页)

```
#define DCMI_D1_GPIO_PIN                GPIO_PIN_7
#define DCMI_D1_AF                      GPIO_AF13_DCMI

#define DCMI_D2_GPIO_PORT               GPIOC
#define DCMI_D2_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_D2_GPIO_PIN                GPIO_PIN_8
#define DCMI_D2_AF                      GPIO_AF13_DCMI

#define DCMI_D3_GPIO_PORT               GPIOC
#define DCMI_D3_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_D3_GPIO_PIN                GPIO_PIN_9
#define DCMI_D3_AF                      GPIO_AF13_DCMI

#define DCMI_D4_GPIO_PORT               GPIOE
#define DCMI_D4_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOE_CLK_ENABLE()
#define DCMI_D4_GPIO_PIN                GPIO_PIN_4
#define DCMI_D4_AF                      GPIO_AF13_DCMI

#define DCMI_D5_GPIO_PORT               GPIOB
#define DCMI_D5_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOB_CLK_ENABLE()
#define DCMI_D5_GPIO_PIN                GPIO_PIN_6
#define DCMI_D5_AF                      GPIO_AF13_DCMI

#define DCMI_D6_GPIO_PORT               GPIOE
#define DCMI_D6_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOE_CLK_ENABLE()
#define DCMI_D6_GPIO_PIN                GPIO_PIN_5
#define DCMI_D6_AF                      GPIO_AF13_DCMI
#define DCMI_D7_GPIO_PORT               GPIOE
#define DCMI_D7_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOE_CLK_ENABLE()
#define DCMI_D7_GPIO_PIN                GPIO_PIN_6
#define DCMI_D7_AF                      GPIO_AF13_DCMI
```

以上代码根据硬件的连接，把与 DCMI、I2C 接口与摄像头通讯使用的引脚号、引脚源以及复用功能映射都以宏封装起来。

45.5.2.2.2 初始化 DCMI 的 GPIO 及 I2C

利用上面的宏，初始化 DCMI 的 GPIO 引脚及 I2C。

```
/**
* @brief  初始化 I2C 总线，使用 I2C 前需要调用
* @param  无
* @retval 无
*/
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_HIGH;
   GPIO_InitStructure.Pull= GPIO_NOPULL;
   GPIO_InitStructure.Alternate=SENSORS_I2C_AF;
   HAL_GPIO_Init(SENSORS_I2C_SCL_GPIO_PORT, &GPIO_InitStructure);

   /* 配置 I2Cx 引脚: SDA ----------------------------------------*/
```

(下页继续)

(续上页)

```
    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.Timing           = 0x60201E2B;//100KHz
        I2C_Handle.Init.OwnAddress1      = 0;
        I2C_Handle.Init.AddressingMode   = I2C_ADDRESSINGMODE_7BIT;
        I2C_Handle.Init.DualAddressMode  = I2C_DUALADDRESS_DISABLE;
        I2C_Handle.Init.OwnAddress2      = 0;
        I2C_Handle.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
        I2C_Handle.Init.GeneralCallMode  = I2C_GENERALCALL_DISABLE;
        I2C_Handle.Init.NoStretchMode    = I2C_NOSTRETCH_DISABLE;

        /* 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);
    }
}
/**
* @brief  初始化控制摄像头使用的 GPIO(I2C/DCMI)
* @param  None
* @retval None
```

(下页继续)

(续上页)

```
*/
void OV2640_HW_Init(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;

   /***DCMI 引脚配置 ***/
   /* 使能 DCMI 时钟 */
   DCMI_PWDN_GPIO_CLK_ENABLE();
   DCMI_VSYNC_GPIO_CLK_ENABLE();
   DCMI_HSYNC_GPIO_CLK_ENABLE();
   DCMI_PIXCLK_GPIO_CLK_ENABLE();
   DCMI_D0_GPIO_CLK_ENABLE();
   DCMI_D1_GPIO_CLK_ENABLE();
   DCMI_D2_GPIO_CLK_ENABLE();
   DCMI_D3_GPIO_CLK_ENABLE();
   DCMI_D4_GPIO_CLK_ENABLE();
   DCMI_D5_GPIO_CLK_ENABLE();
   DCMI_D6_GPIO_CLK_ENABLE();
   DCMI_D7_GPIO_CLK_ENABLE();

   /* 控制/同步信号线 */
   GPIO_InitStructure.Pin = DCMI_VSYNC_GPIO_PIN;
   GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
   GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
   GPIO_InitStructure.Pull = GPIO_PULLUP ;
   GPIO_InitStructure.Alternate = DCMI_VSYNC_AF;
   HAL_GPIO_Init(DCMI_VSYNC_GPIO_PORT, &GPIO_InitStructure);

   GPIO_InitStructure.Pin = DCMI_HSYNC_GPIO_PIN;
   GPIO_InitStructure.Alternate = DCMI_VSYNC_AF;
   HAL_GPIO_Init(DCMI_HSYNC_GPIO_PORT, &GPIO_InitStructure);
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin = DCMI_PIXCLK_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_PIXCLK_AF;
    HAL_GPIO_Init(DCMI_PIXCLK_GPIO_PORT, &GPIO_InitStructure);

    /* 数据信号 */
    GPIO_InitStructure.Pin = DCMI_D0_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D0_AF;
    HAL_GPIO_Init(DCMI_D0_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D1_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D1_AF;
    HAL_GPIO_Init(DCMI_D1_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D2_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D2_AF;
    HAL_GPIO_Init(DCMI_D2_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D3_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D3_AF;
    HAL_GPIO_Init(DCMI_D3_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D4_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D4_AF;
    HAL_GPIO_Init(DCMI_D4_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D5_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D5_AF;
    HAL_GPIO_Init(DCMI_D5_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D6_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D6_AF;
```

(下页继续)

(续上页)

```
    HAL_GPIO_Init(DCMI_D6_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_D7_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D7_AF;
    HAL_GPIO_Init(DCMI_D7_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_PWDN_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    HAL_GPIO_Init(DCMI_PWDN_GPIO_PORT, &GPIO_InitStructure);
    /*PWDN 引脚，高电平关闭电源，低电平供电 */
    HAL_GPIO_WritePin(DCMI_PWDN_GPIO_PORT,
    DCMI_PWDN_GPIO_PIN,GPIO_PIN_RESET);

}
```

函数中 I2C 的初始化配置，使用 I2C 与 OV2640 的 SCCB 接口通讯，这里的 I2C 模式配置与标准的 I2C 无异。

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，以上代码把 DCMI 接口的信号线全都初始化为 DCMI 复用功能，而 PWDN 则被初始化成普通的推挽输出模式，并且在初始化完毕后直接控制它为低电平，使能给摄像头供电。

45.5.2.2.3 配置 DCMI 的模式

接下来需要配置 DCMI 的工作模式，我们通过编写 OV2640_Init 函数完成该功能。

```
/**
* @brief  配置 DCMI/DMA 以捕获摄像头数据
* @param  None
* @retval None
*/
void OV2640_Init(void)
```

(下页继续)

(续上页)

```
{
    /*** 配置 DCMI 接口 ***/
    /* 使能 DCMI 时钟 */
    __HAL_RCC_DCMI_CLK_ENABLE();

    /* DCMI 配置 */
    DCMI_Handle.Instance              = DCMI;
    DCMI_Handle.Init.SynchroMode      = DCMI_MODE_CONTINUOUS;
    DCMI_Handle.Init.SynchroMode      = DCMI_SYNCHRO_HARDWARE;
    DCMI_Handle.Init.PCKPolarity      = DCMI_PCKPOLARITY_RISING;
    DCMI_Handle.Init.VSPolarity       = DCMI_VSPOLARITY_LOW;
    DCMI_Handle.Init.HSPolarity       = DCMI_HSPOLARITY_LOW;
    DCMI_Handle.Init.CaptureRate      = DCMI_CR_ALL_FRAME;
    DCMI_Handle.Init.ExtendedDataMode = DCMI_EXTEND_DATA_8B;
    HAL_DCMI_Init(&DCMI_Handle);

    /* 配置中断 */
    HAL_NVIC_SetPriority(DCMI_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(DCMI_IRQn);

    //开始传输，数据大小以 32 位数据为单位 (即像素个数/4，LCD_GetXSize()*LCD_
↪GetYSize()*2/4)
    OV2640_DMA_Config(LCD_FB_START_ADDRESS,LCD_GetXSize()*LCD_GetYSize()/2);
}
```

该函数的执行流程如下：

(1) 使能 DCMI 外设的时钟，它是挂载在 AHB2 总线上的；

(2) 根据摄像头的时序和硬件连接的要求，配置 DCMI 工作模式为：使用硬件同步，连续采集所有帧数据，采集时使用 8 根数据线，PIXCLK 被设置为上升沿有效，VSYNC 和 HSYNC 都被设置成低电平有效；

(3) 调用 OV2640_DMA_Config 函数开始 DMA 数据传输，每传输完一帧数据需要调用一次，它

包含本次传输的目的首地址及传输的数据量，后面我们再详细解释；

(4) 配置 DMA 中断，DMA 每次传输完毕会引起中断，以便我们在中断服务函数配置 DMA 传输下一帧数据；

(5) 配置 DCMI 的帧传输中断，为了防止有时 DMA 出现传输错误或传输速度跟不上导致数据错位、偏移等问题，每次 DCMI 接收到摄像头的一帧数据，得到新的帧同步信号后 (VSYNC)，就进入中断，复位 DMA，使它重新开始一帧的数据传输。

45.5.2.2.4 配置 DMA 数据传输

上面的 DCMI 配置函数中调用了 OV2640_DMA_Config 函数开始了 DMA 传输。

```
/**
* @brief  配置 DCMI/DMA 以捕获摄像头数据
* @param  DMA_Memory0BaseAddr: 本次传输的目的首地址
* @param DMA_BufferSize: 本次传输的数据量 (单位为字, 即 4 字节)
*/
void OV2640_DMA_Config(uint32_t DMA_Memory0BaseAddr,uint32_t DMA_
↪BufferSize)
{
    /* 配置 DMA 从 DCMI 中获取数据 */
    /* 使能 DMA*/
    __HAL_RCC_DMA2_CLK_ENABLE();
    DMA_Handle_dcmi.Instance = DMA2_Stream1;
    DMA_Handle_dcmi.Init.Channel = DMA_CHANNEL_1;
    DMA_Handle_dcmi.Init.Direction = DMA_PERIPH_TO_MEMORY;
    DMA_Handle_dcmi.Init.PeriphInc = DMA_PINC_DISABLE;
    DMA_Handle_dcmi.Init.MemInc = DMA_MINC_ENABLE;       //寄存器地址自增
    DMA_Handle_dcmi.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
    DMA_Handle_dcmi.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    DMA_Handle_dcmi.Init.Mode = DMA_CIRCULAR;                 //循环模式
    DMA_Handle_dcmi.Init.Priority = DMA_PRIORITY_HIGH;
```

(下页继续)

(续上页)

```
    DMA_Handle_dcmi.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
    DMA_Handle_dcmi.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
    DMA_Handle_dcmi.Init.MemBurst = DMA_MBURST_SINGLE;
    DMA_Handle_dcmi.Init.PeriphBurst = DMA_PBURST_SINGLE;

    /*DMA 中断配置 */
    __HAL_LINKDMA(&DCMI_Handle, DMA_Handle, DMA_Handle_dcmi);

    HAL_NVIC_SetPriority(DMA2_Stream1_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(DMA2_Stream1_IRQn);

    HAL_DMA_Init(&DMA_Handle_dcmi);
    //使能 DCMI 采集数据
    HAL_DCMI_Start_DMA(&DCMI_Handle, DCMI_MODE_CONTINUOUS,
    (uint32_t)DMA_Memory0BaseAddr,DMA_BufferSize);
}
```

该函数跟普通的 DMA 配置无异，它把 DCMI 接收到的数据从它的数据寄存器搬运到 SDRAM 显存中，从而直接使用液晶屏显示摄像头采集得的图像。它包含 2 个输入参数 DMA_Memory0BaseAddr 和 DMA_BufferSize，其中 DMA_Memory0BaseAddr 用于设置本次 DMA 传输的目的首地址，DMA_BufferSize 则用于指示本次 DMA 传输的数据量，要注意它的单位是一个字，即 4 字节，如我们要传输 60 字节的数据时，它应配置为 15。这两参数会被传递到库函数 HAL_DCMI_Start_DMA 中作为形参。在前面的 OV2640_Init 函数中，对这个函数有如下调用：

```
/* 摄像头采集图像的大小，改变这两个值可以改变数据量，
但不会加快采集速度，要加快采集速度需要改成 SVGA 械 */
#define img_width  LCD_GetXSize()
#define img_height LCD_GetYSize()

//开始传输，数据大小以 32 位数据为单位 (即像素个数/4，LCD_GetXSize()*LCD_
↪GetYSize()*2/4)
OV2640_DMA_Config(LCD_FB_START_ADDRESS,LCD_GetXSize()*LCD_GetYSize()/2);
```

其中的 LCD_GetXSize 和 LCD_GetYSize 获取液晶屏的分辨率，img_width 和 img_heigh 表示摄像头输出的图像的分辨率，LCD_FB_START_ADDRESS 是液晶层的首个显存地址。另外，本工程中显示摄像头数据的这个液晶层采用 RGB565 的像素格式，每个像素点占据 2 个字节。把摄像头输出的每一帧数据显示到液晶屏上，不需要额外的处理这样最简单直接。

45.5.2.2.5 DMA 传输完成中断及帧中断

OV2640_Init 函数初始化了 DCMI，使能了帧中断、DMA 传输完成中断，并使能了第一次 DMA 传输，当这一行数据传输完成时，会进入 DMA 中断服务函数，见 45-0-7 中的 DMA2_Stream1_IRQHandler。

```
/**
* @brief  DMA 中断服务函数
* @param  None
* @retval None
*/
void DMA2_Stream1_IRQHandler(void)
{
    HAL_DMA_IRQHandler(&DMA_Handle_dcmi);
}

/**
* @brief  DCMI 中断服务函数
* @param  None
* @retval None
*/
void DCMI_IRQHandler(void)
{
    HAL_DCMI_IRQHandler(&DCMI_Handle);
}
/**
* @brief  帧同步回调函数.
```

(下页继续)

(续上页)

```
* @param  None
* @retval None
*/
void HAL_DCMI_VsyncEventCallback(DCMI_HandleTypeDef *hdcmi)
{
    fps++; //帧率计数
    OV2640_DMA_Config(LCD_FB_START_ADDRESS,LCD_GetXSize()*LCD_GetYSize()/
↪2);
}
```

DMA 中断服务函数中直接调用库函数进行处理。

当 DCMI 接口检测到摄像头传输的帧同步信号时，会进入 DCMI_IRQHandler 中断服务函数，DCMI 中断服务函数中直接调用库函数进行处理。每次帧同步来临是重新设置一次 DMA 传输数据，液晶的显存就会收到摄像头采集的数据然后显示在液晶上。

45.5.2.2.6 读取 OV2640 芯片 ID

配置完了 STM32 的 DCMI，还需要控制摄像头，它有很多寄存器用于配置工作模式。利用 STM32 的 I2C 接口，可向 OV2640 的寄存器写入控制参数，我们先写个读取芯片 ID 的函数测试一下，见代码清单 45-0-8。

```
//存储摄像头 ID 的结构体
typedef struct
{
    uint8_t Manufacturer_ID1;
    uint8_t Manufacturer_ID2;
    uint8_t PIDH;
    uint8_t PIDL;
} OV2640_IDTypeDef;
/* 寄存器地址 */
#define OV2640_DSP_RA_DLMT        0xFF
```

(下页继续)

(续上页)

```
#define OV2640_SENSOR_MIDH       0x1C
#define OV2640_SENSOR_MIDL       0x1D
#define OV2640_SENSOR_PIDH       0x0A
#define OV2640_SENSOR_PIDL       0x0B
/**
* @brief  读取摄像头的 ID.
* @param  OV2640ID: 存储 ID 的结构体
* @retval None
*/
void OV2640_ReadID(OV2640_IDTypeDef *OV2640ID)
{
    /*OV2640 有两组寄存器，设置 0xFF 寄存器的值为 0 或为 1 时可选择使用不同组的寄存器
*/
    OV2640_WriteReg(OV2640_DSP_RA_DLMT, 0x01);

    /* 读取寄存芯片 ID*/
    OV2640ID->Manufacturer_ID1 = OV2640_ReadReg(OV2640_SENSOR_MIDH);
    OV2640ID->Manufacturer_ID2 = OV2640_ReadReg(OV2640_SENSOR_MIDL);
    OV2640ID->PIDH = OV2640_ReadReg(OV2640_SENSOR_PIDH);
    OV2640ID->PIDL = OV2640_ReadReg(OV2640_SENSOR_PIDL);
}
```

在 OV2640 的 MIDH 及 MIDL 寄存器中存储了它的厂商 ID，默认值为 0x7F 和 0xA2；而 PIDH 及 PIDL 寄存器存储了产品 ID，PIDH 的默认值为 0x26，PIDL 的默认值不定，可能的值为 0x40、0x41 及 0x42。在代码中我们定义了一个结构体 OV2640_IDTypeDef 专门存储这些读取得的 ID 信息。

由于这些寄存器都是属于 Sensor 组的，所以在读取这些寄存器的内容前，需要先把 OV2640 中 0xFF 地址的 DLMT 寄存器值设置为 1。

OV2640_ReadID 函数中使用的 OV2640_ReadReg 及 OV2640_WriteReg 函数是使用 STM32 的 I2C 外设向某寄存器读写单个字节数据的底层函数，它与我们前面章节中用到的 I2C 函数大同小异，就不展开分析了。

45.5.2.2.7 向 OV2640 写入寄存器配置

检测到 OV2640 的存在后，向它写入配置参数。

```
/* OV2640 的 SVGA 是 600 列 *800 行的，在 800 列 *480 行的液晶屏上不能全屏 */
/* 所以直接用 UXGA 模式，再根据所需的图像窗口裁剪 */
const unsigned char OV2640_UXGA[][2]=
{
    0xff, 0x00,
    0x2c, 0xff,
    0x2e, 0xdf,
    0xff, 0x01,
    0x3c, 0x32,
    0x11, 0x00,
    0x09, 0x02,
    0x04, 0x20|0x80,     //水平翻转
    0x13, 0xe5,

    /*.... 以下省略 */
}
/**
* @brief  配置 OV2640 为 UXGA 模式，并设置输出图像大小
* @param  None
* @retval None
*/
void OV2640_UXGAConfig(void)
{
    uint32_t i;

    /* 摄像头复位 */
    OV2640_Reset();

    /* 进行三次寄存器写入，确保配置写入正常
```

(下页继续)

(续上页)

```
    (在使用摄像头长排线时，IIC 数据线干扰较大，必须多次写入来保证正常)*/
    /* 写入寄存器配置 */
    for (i=0; i<(sizeof(OV2640_UXGA)/2); i++) {
        OV2640_WriteReg(OV2640_UXGA[i][0], OV2640_UXGA[i][1]);

    }
    /* Initialize OV2640 */
    for (i=0; i<(sizeof(OV2640_UXGA)/2); i++) {
        OV2640_WriteReg(OV2640_UXGA[i][0], OV2640_UXGA[i][1]);

    }
    /* Initialize OV2640 */
    for (i=0; i<(sizeof(OV2640_UXGA)/2); i++) {
        OV2640_WriteReg(OV2640_UXGA[i][0], OV2640_UXGA[i][1]);

    }
    /* 设置输出的图像大小 */
    OV2640_OutSize_Set(img_width,img_height);
}
```

这个 OV2640_UXGAConfig 函数简单粗暴，它直接把一个二维数组 OV2640_UXGA 使用 I2C 传输到 OV2640 中，该二维数组的第一维存储的是寄存器地址，第二维存储的是对应寄存器要写入的控制参数。

如果您对这些寄存器配置感兴趣，可以一个个对着 OV2640 的寄存器说明来阅读，阅读时要注意区分 DLMT 寄存器 (地址 0xFF) 为 1 或为 0 时的寄存器组。总的来部，这些配置主要是把 OV2640 配置成了 UXGA 时序模式，并使用 8 根数据线输出格式为 RGB565 的图像数据。

其中 UXGA 时序是指它最大可输出 1600x1200 分辨率的图像，OV2640 还支持使用 SVGA 时序输出最大分辨率为 800x600 的图像，相对于 UXGA，它可使用更高的帧率输出，但由于它规定好了数据是 800 行、600 列，而我们的液晶屏在横屏状态下，无法直接全屏显示 (800 列、480 行)，所以就把 OV2640 配置成 UXGA 模式了。通过修改 COM4 寄存器 (地址 0x12) 可改变时序模式。

45.5.2.2.8 main 函数

最后我们来编写 main 函数，利用前面讲解的函数，控制采集图像。

```
int main(void)
{
    OV2640_IDTypeDef OV2640_Camera_ID;
    /* 系统时钟初始化成 168MHz */
    SystemClock_Config();
    /* LED 端口初始化 */
    LED_GPIO_Config();
    /* 初始化 USART1*/
    DEBUG_USART_Config();

    //LCD 初始化
    ILI9806G_Init ();

    LCD_SetFont(&Font16x32);
    LCD_SetColors(RED,BLACK);

    ILI9806G_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
    ILI9806G_GramScan ( 5 );

    CAMERA_DEBUG("STM32F407 DCMI 驱动 OV2640 例程");
    //初始化 I2C
    I2CMaster_Init();

    OV2640_HW_Init();
```

(下页继续)

(续上页)

```
    HAL_Delay(100);

    /* 读取摄像头芯片 ID，确定摄像头正常连接 */
    OV2640_ReadID(&OV2640_Camera_ID);

    if (OV2640_Camera_ID.PIDH  == 0x26) {
        CAMERA_DEBUG("%x%x",OV2640_Camera_ID.PIDH ,OV2640_Camera_ID.PIDL);
    } else {
        LCD_SetTextColor(RED);
        ILI9806G_DispString_EN(10,10,"Can not detect OV2640 module,please␣
↪check the connection!");
        CAMERA_DEBUG("没有检测到 OV2640 摄像头，请重新检查连接。");

        while (1);
    }
    OV2640_UXGAConfig();

    OV2640_Init();

    //重置
    fps =0;
    Task_Delay[0]=1000;

    //扫描模式，横屏
    ILI9806G_GramScan(5);
    LCD_SetFont(&Font16x32);
    LCD_SetColors(RED,BLACK);
    ILI9806G_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */
    ILI9806G_DispStringLine_EN(LINE(0),"BH 4.8 inch LCD + OV2640");

    /*DMA 会把数据传输到液晶屏，开窗后数据按窗口排排列 */
```

(下页继续)

(续上页)

```
    ILI9806G_OpenWindow(0,0,img_width,img_height);

    while (1) {
        //使用串口输出帧率
        if (Task_Delay[0]==0) {
//      HAL_DCMI_Start_DMA(&DCMI_Handle, DCMI_MODE_CONTINUOUS,
                           FSMC_Addr_ILI9806G_DATA,1);

            /* 输出帧率 */
            CAMERA_DEBUG("\r\n帧率:%.1f/s \r\n", (double)fps/5.0);
            //重置
            fps =0;
            Task_Delay[0]=5000;//此值每 1ms 会减 1，减到 0 才可以重新进来这里
        }
    }
}
```

在 main 函数中，首先初始化了系统时钟，串口、液晶屏，注意它是把摄像头使用的液晶层初始化成 RGB565 格式了，可直接在工程的液晶底层驱动解这方面的内容。

摄像头控制部分，首先调用了 OV2640_HW_Init 函数初始化 DCMI 及 I2C，然后调用 OV2640_ReadID 函数检测摄像头与实验板是否正常连接，若连接正常则调用 OV2640_Init 函数初始化 DCMI 的工作模式及 DMA，再调用 OV2640_UXGAConfig 函数向 OV2640 写入寄存器配置，DCMI 开始捕获数据，摄像头数据显示在液晶屏上面。

45.5.3 下载验证

把 OV2640 接到实验板的摄像头接口中，用 USB 线连接开发板，编译程序下载到实验板，并上电复位，液晶屏会显示摄像头采集得的图像，通过旋转镜头可以调焦。

第 46 章　DCMI—OV5640 摄像头

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

关于开发板配套的 OV5640 摄像头参数可查阅《ov5640datasheet》配套资料获知。

STM32F4 芯片具有浮点运算单元，适合对图像信息使用 DSP 进行基本的图像处理，其处理速度比传统的 8、16 位机快得多，而且它还具有与摄像头通讯的专用 DCMI 接口，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。本章讲解如何使用 STM32 驱动 OV5640 型号的摄像头。

46.1 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。

46.1.1 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 41 万左右。现在的数字摄像头分辨率一般从数十万到数千万。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。

46.1.2 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。

46.2 OV5640 摄像头

本章主要讲解实验板配套的摄像头，它的实物见图 46-0-1，该摄像头主要由镜头、图像传感器、板载电路及下方的信号引脚组成。

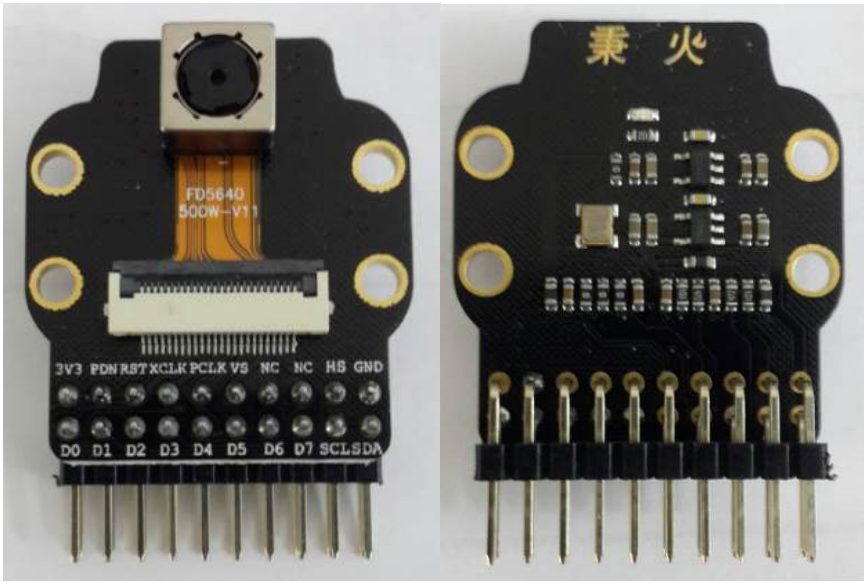


图 46-0-1 实验板配套的 OV5640 摄像头

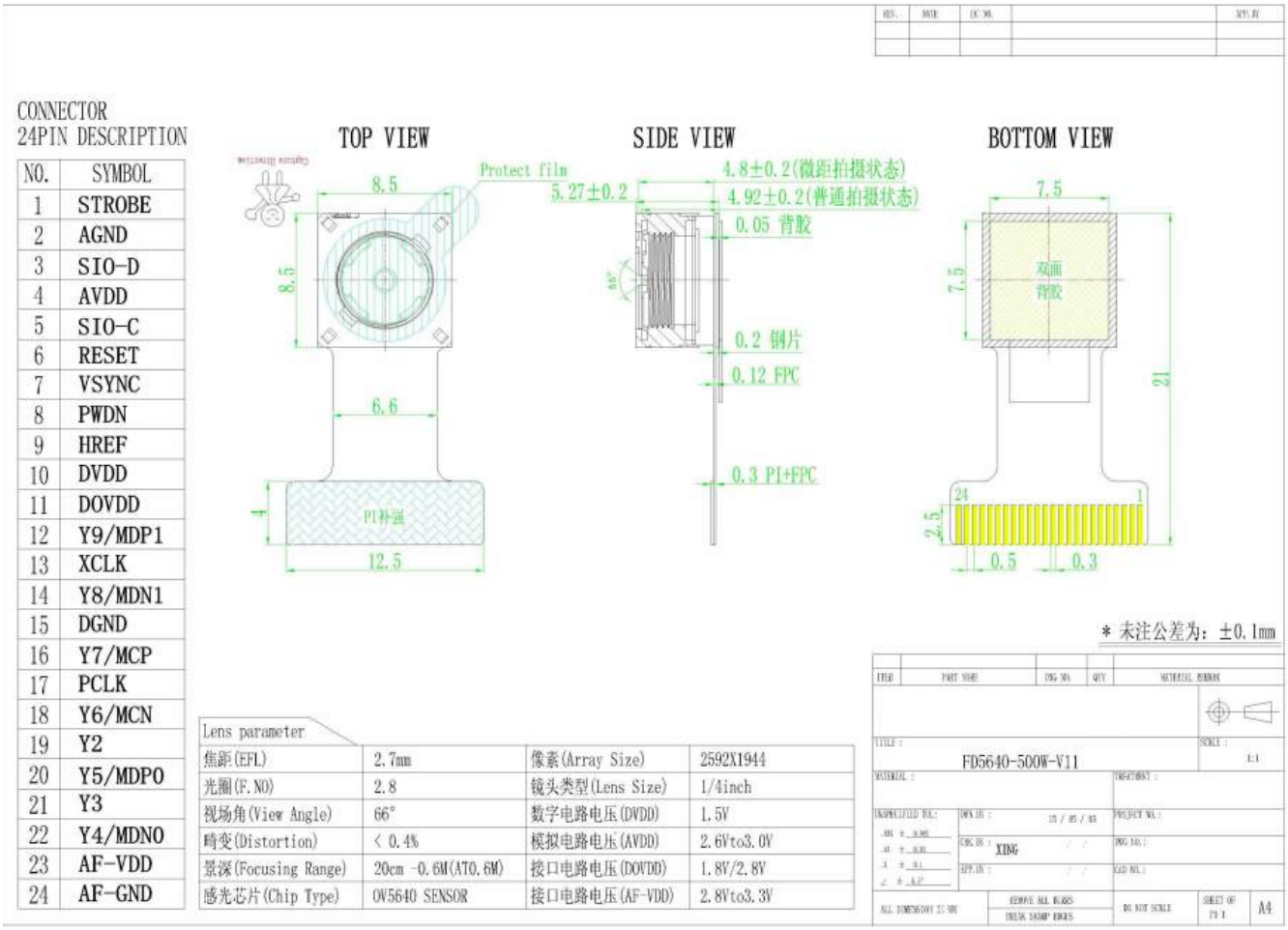
镜头部件包含一个镜头座和一个可旋转调节距离的凸透镜，通过旋转可以调节焦距，正常使用时，镜头座覆盖在电路板上遮光，光线只能经过镜头传输到正中央的图像传感器，它采集光线信号，然后把采集得的数据通过下方的信号引脚输出数据到外部器件。

46.2.1 OV5640 传感器简介

图像传感器是摄像头的核心部件，上述摄像头中的图像传感器是一款型号为 OV5640 的 CMOS 类型数字图像传感器。该传感器支持输出最大为 500 万像素的图像 (2592x1944 分辨率)，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、RGB565 以及 JPEG 格式，若直接输出 JPEG 格式的图像时可大大减少数据量，方便网络传输。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。根据不同的分辨率配置，传感器输出图像数据的帧率从 15-60 帧可调，工作时功率在 150mW-200mW 之间。

46.2.2 OV5640 引脚及功能框图

OV5640 模组带有自动对焦功能，引脚的定义见图 *46_0_2*。



NO.	SYMBOL
1	STROBE
2	AGND
3	SIO-D
4	AVDD
5	SIO-C
6	RESET
7	VSYNC
8	PWDN
9	HREF
10	DVDD
11	DOVDD
12	Y9/MDP1
13	XCLK
14	Y8/MDN1
15	DGND
16	Y7/MCP
17	PCLK
18	Y6/MCN
19	Y2
20	Y5/MDP0
21	Y3
22	Y4/MDN0
23	AF-VDD
24	AF-GND

Lens parameter

焦距(EFL)	2.7mm	像素(Array Size)	2592X1944
光圈(F.NO)	2.8	镜头类型(Lens Size)	1/4inch
视场角(View Angle)	66°	数字电路电压(DVDD)	1.5V
畸变(Distortion)	< 0.4%	模拟电路电压(AVDD)	2.6Vto3.0V
景深(Focusing Range)	20cm ~0.6M(AT0.6M)	接口电路电压(DOVDD)	1.8V/2.8V
感光芯片(Chip Type)	OV5640 SENSOR	接口电路电压(AF-VDD)	2.8Vto3.3V

图 46-0-2 OV5640 传感器引脚分布图

信号引脚功能介绍如下，介绍如下表 46-0-1。

表 46-0-1 OV5640 管脚

管脚名称	管脚类型	管脚描述
SIO_C	输入	SCCB 总线的时钟线，可类比 I2C 的 SCL
SIO_D	I/O	SCCB 总线的数据线，可类比 I2C 的 SDA
RESET	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	帧同步信号
PCLK	输出	像素同步时钟输出信号
XCLK	输入	外部时钟输入端口，可接外部晶振
Y2…Y9	输出	像素数据输出端口

下面我们配合图 *46_0_3* 中的 OV5640 功能框图讲解这些信号引脚。

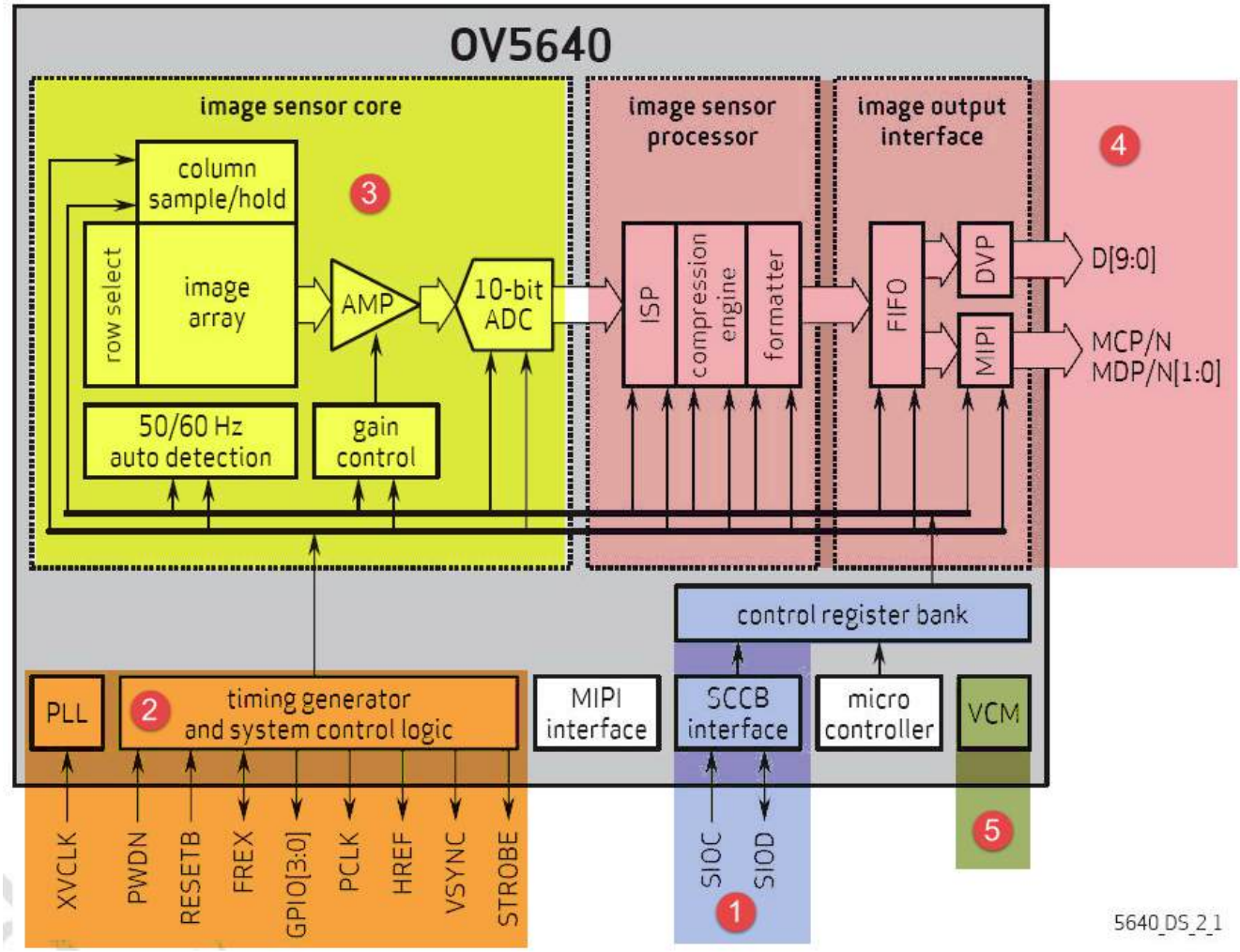


图 46-0-3 OV5640 功能框图

(1) 控制寄存器

标号处的是 OV5640 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 SIO_C 和 SIO_D 引脚写入的，SIO_C 与 SIO_D 使用的通讯协议跟 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 OV5640 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信号是很类似的。RESETB 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 OV5640 的信号；而 PCLK 是 OV5640 输出数据时的同步信号，它是由 OV5640 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 OV5640 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 Y0-Y9 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 Y2-Y9 引脚，OV5640 与外部器件的连接方式见图图 *46_0_4*。

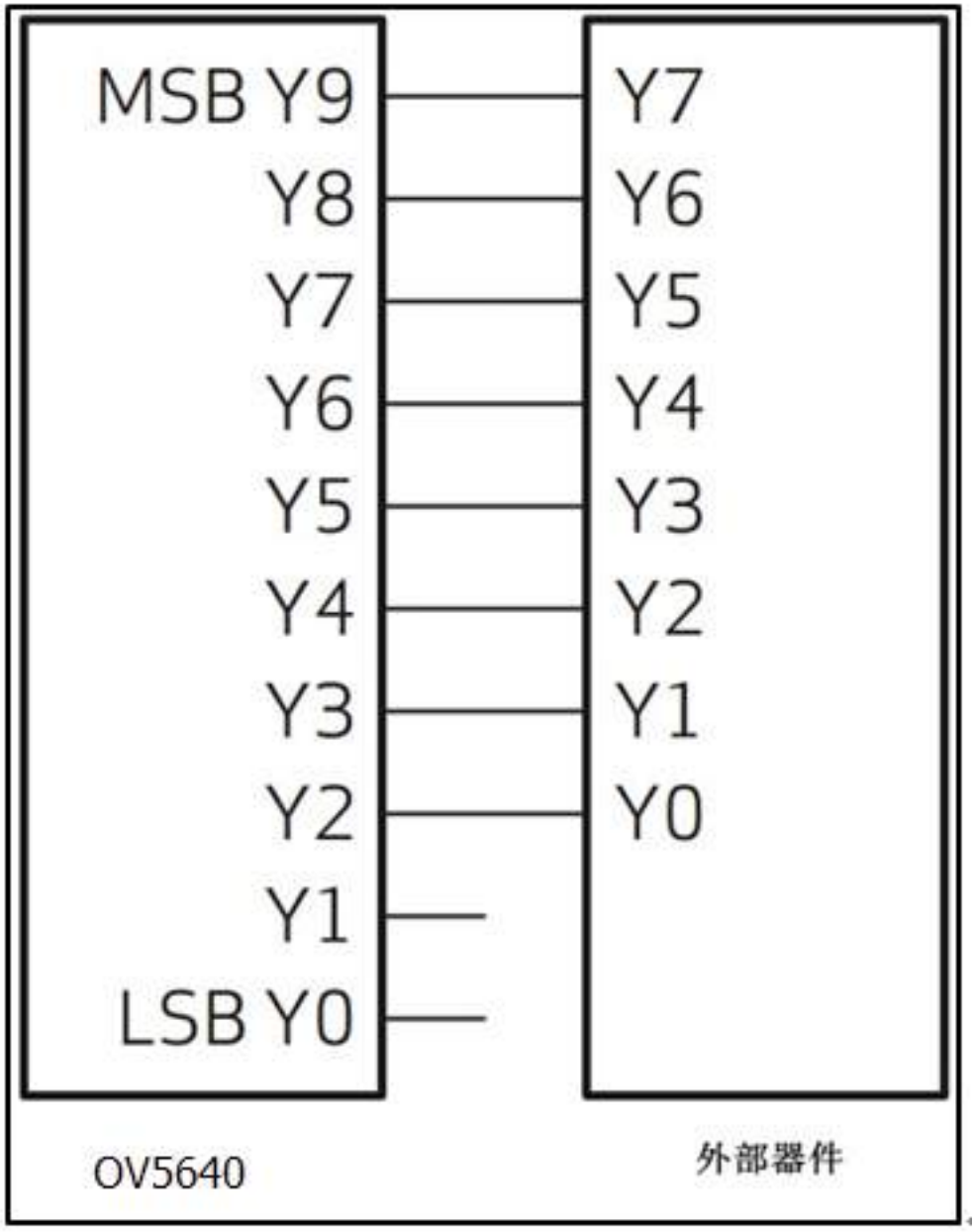


图 46-0-4 8 位数据线接法

(5) 数据输出信号

标号 ⑤ 处为 VCM 处理单元，他会通过图像分析来实现图像的自动对焦功能。要实现自动对焦还需要下载自动对焦固件到模组，后面摄像头实验详细介绍这个功能。

46.2.3 SCCB 时序

外部控制器对 OV5640 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 驱动中我们直接使用片上 I2C 外设与它通讯。SCCB 与标准的 I2C 协议的区别是它每次传输只能写入或读取一个字节的数据，而 I2C 协议是支持突发读写的，即在一次传输中可以写入多个字节的数据 (EEPROM 中的页写入时序即突发写)。关于 SCCB 协议的完整内容可查看配套资料里的《SCCB 协议》文档，下面我们简单介绍下。

46.2.3.1 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 46-5 及图 46-6。

- 起始信号：在 SIO_C 为高电平时，SIO_D 出现一个下降沿，则 SCCB 开始传输。
- 停止信号：在 SIO_C 为高电平时，SIO_D 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 SIO_C 为高电平时，必须保证 SIO_D 上的数据稳定，也就是说，SIO_D 上的电平变换只能发生在 SIO_C 为低电平的时候，SIO_D 的信号在 SIO_C 为高电平时被采集。

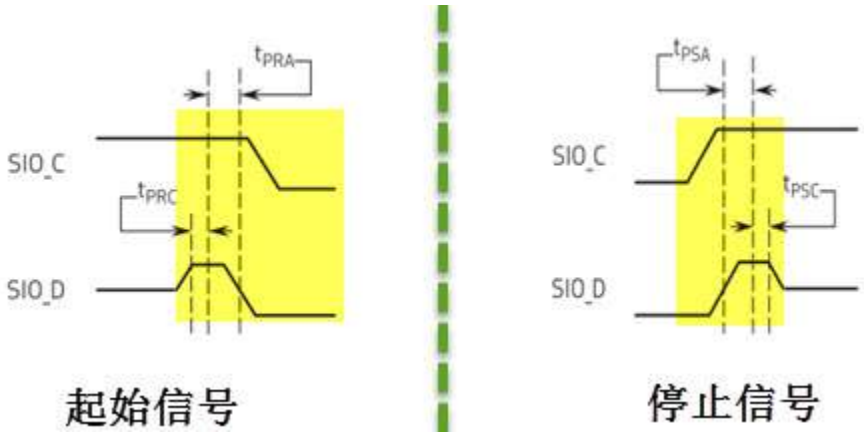


图 46-0-5 SCCB 停止信号

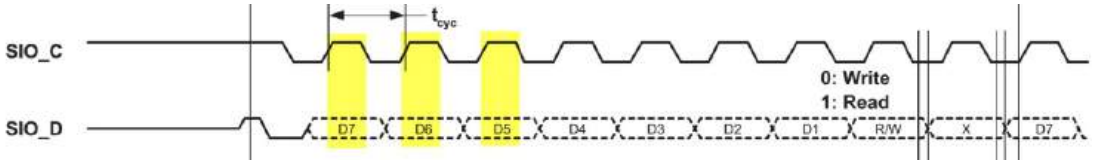


图 46-0-6 SCCB 的数据有效性

46.2.3.2 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 *46_0_7*。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 16 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。

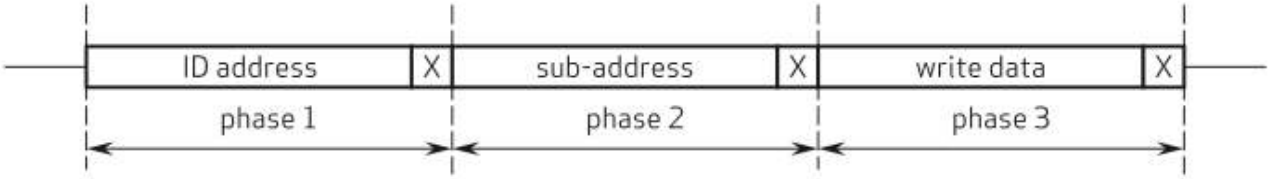


图 46-0-7 SCCB 的三步写操作

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 *46_0_8*。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。

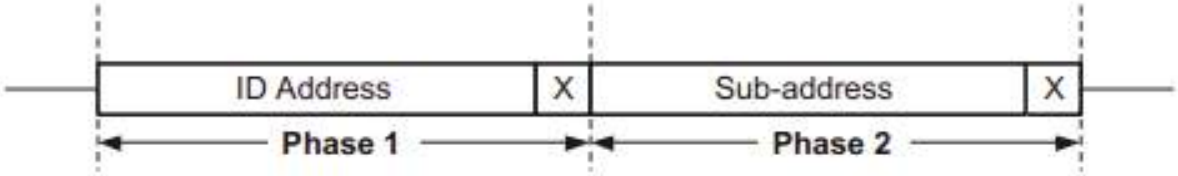


图 46-0-8 SCCB 的两步写操作

两步读操作，它用于读取从设备目的寄存器中的数据，见图 *46_0_9*。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。

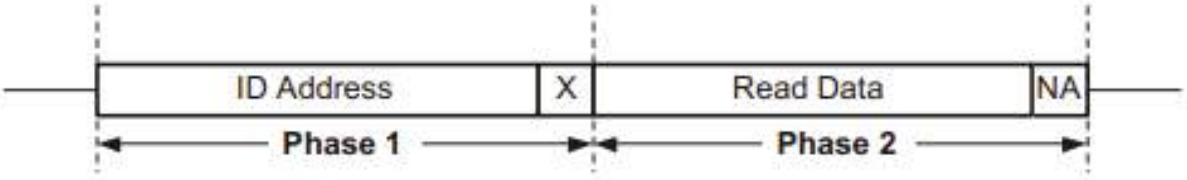


图 46-0-9 SCCB 的两步读操作

可以看到，以上介绍的 SCCB 特性都与 I2C 无区别，而 I2C 比 SCCB 还多出了突发读写的功能，所以 SCCB 可以看作是 I2C 的子集，我们完全可以使用 STM32 的 I2C 外设来与 OV5640 进行 SCCB 通讯。

46.2.4 OV5640 的寄存器

控制 OV5640 涉及到它很多的寄存器，可直接查询《ov5640datasheet》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、图像格式及图像方向等。要注意的是 OV5640 寄存器地址为 16 位。

官方还提供了一个《OV5640_ 自动对焦照相模组应用指南 (DVP_ 接口)__R2.13C.pdf》的文档，它针对不同的配置需求，提供了配置范例，见图 *46_0_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。

4.1.2 VGA 预览

VGA 30 帧/秒

```
// YUV VGA 30fps, night mode 5fps
// Input Clock = 24Mhz, PCLK = 56MHz
write_i2c(0x3035, 0x11);        // PLL
write_i2c(0x3036, 0x46);        // PLL
write_i2c(0x3c07, 0x08);        // light meter 1 threshold [7:0]
write_i2c(0x3820, 0x41);        // Sensor flip off, ISP flip on
write_i2c(0x3821, 0x07);        // Sensor mirror on, ISP mirror on, H binning on
```

图 46-0-10 调节帧率的寄存器配置范例

46.2.5 像素数据输出时序

对 OV5640 采用 SCCB 协议进行控制，而它输出图像时则使用 VGA 时序 (还可用 SVGA、UXGA，这些时序都差不多)，这跟控制液晶屏输入图像时很类似。OV5640 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 *46_0_11*。

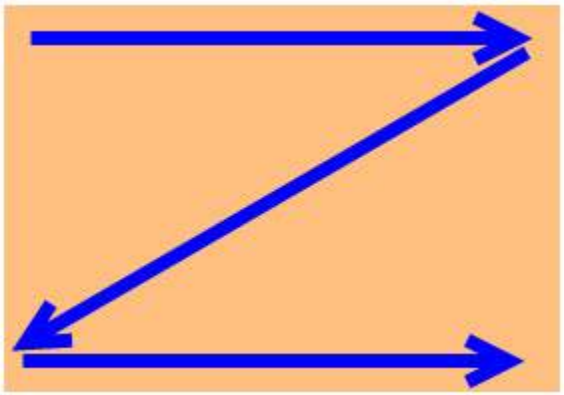


图 46-0-11 摄像头数据输出

例如，图 *46_0_12*，若我们使用 Y2-Y9 数据线，图像格式设置为 RGB565，进行数据输出时，Y2-Y9 数据线会在 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。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，每传输完一帧图像时，VSYNC 会输出一个电平跳变信号。

6.6.2 DVP timing

figure 6-7 DVP timing diagram

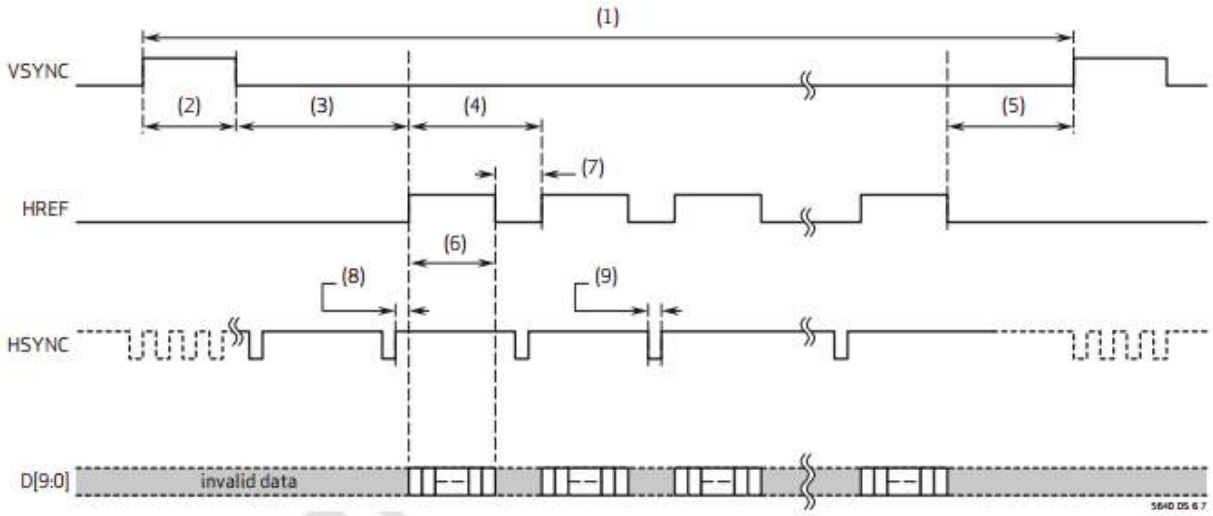


table 6-7 DVP timing specifications (sheet 1 of 2)

mode	timing
5 megapixel 2592x1944	(1) 5596992 tp (2) 5688 tp (3) 48276 tp (4) 2844 tp (5) 14544 tp (6) 2592 tp (7) 252 tp (8) 0 tp (9) 252 tp

图 46-0-12 DVP 接口时序

46.3 STM32 的 DCMI 接口简介

STM32F4 系列的控制器包含了 DCMI 数字摄像头接口 (Digital camera Interface)，它支持使用上述类似 VGA 的时序获取图像数据流，支持原始的按行、帧格式来组织的图像数据，如 YUV、RGB，也支持接收 JPEG 格式压缩的数据流。接收数据时，主要使用 HSYNC 及 VSYNC 信号来同步。

46.3.1 DCMI 整体框图

STM32 的 DCMI 接口整体框图见图 *46_0_13*。

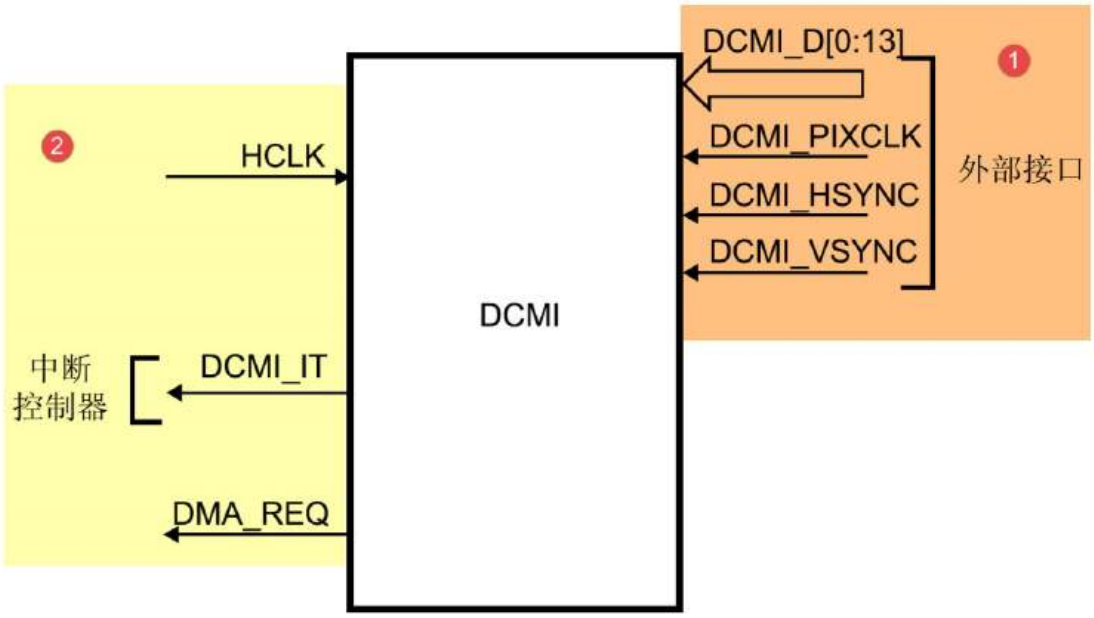


图 46-0-13 DCMI 接口整体框图

46.3.1.1 外部接口及时序

上图标号处的是 DCMI 向外部引出的信号线。DCMI 提供的外部接口的方向都是输入的，接口的各个信号线说明见表 46-0-2。

表 46-0-2 DCMI 的信号线说明

引脚名称	说明
DCMI_D[0:13]	数据线
DCMI_PIXCLK	像素同步时钟
DCMI_HSYNC	行同步信号 (水平同步信号)
DCMI_VSYNC	帧同步信号 (垂直同步信号)

其中 DCMI_D 数据线的数量可选 8、10、12 或 14 位，各个同步信号的有效极性都可编程控制。它使用的通讯时序与 OV5640 的图像数据输出接口时序一致，见图 *46_0_14*。

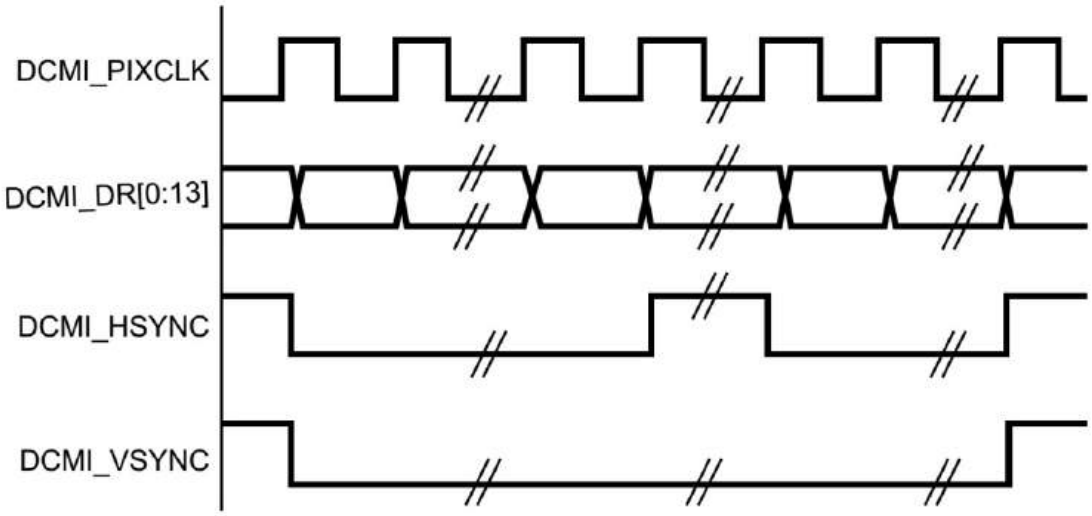


图 46-0-14 DCMI 时序图

46.3.1.2 内部信号及 PIXCLK 的时钟频率

图 *46_0_13* 的标号 1 处表示 DCMI 与内部的信号线。在 STM32 的内部，使用 HCLK 作为时钟源提供给 DCMI 外设。从 DCMI 引出有 DCMI_IT 信号至中断控制器，并可通过 DMA_REQ 信号发送 DMA 请求。

DCMI 从外部接收数据时，在 HCLK 的上升沿时对 PIXCLK 同步的信号进行采样，它限制了 PIXCLK 的最小时钟周期要大于 2.5 个 HCLK 时钟周期，即最高频率为 HCLK 的 1/4。

46.3.2 DCMI 接口内部结构

DCMI 接口的内部结构见图 *46_0_15*。

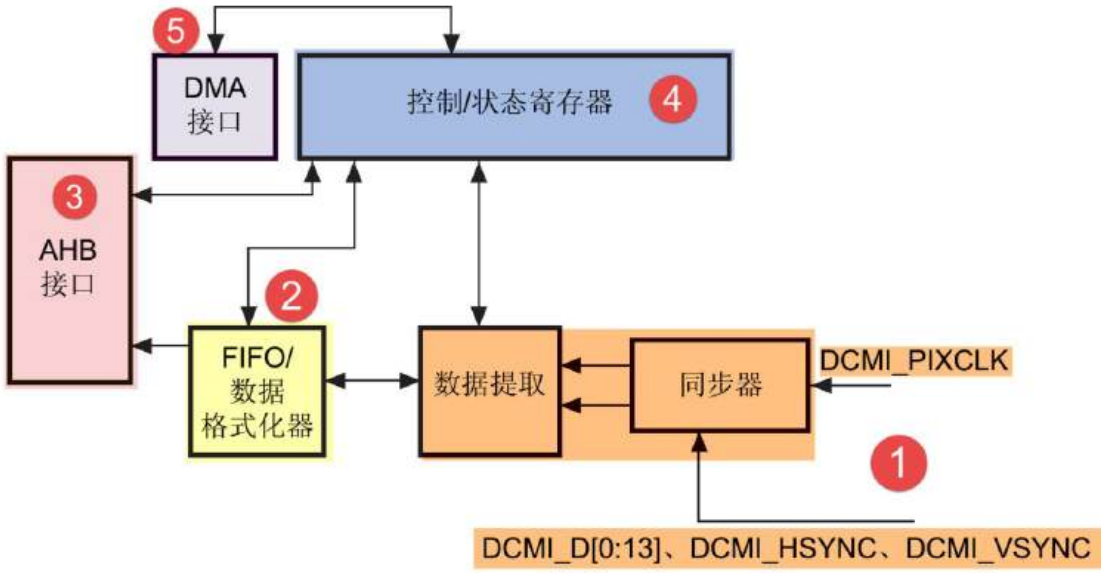


图 46-0-15 DCMI 接口内部结构

(1) 同步器

同步器主要用于管理 DCMI 接收数据的时序，它根据外部的信号提取输入的数据。

(2) FIFO/数据格式化器

为了对数据传输加以管理，STM32 在 DCMI 接口上实现了 4 个字 (32bit x4) 深度的 FIFO，用以缓冲接收到的数据。

(3) AHB 接口

DCMI 接口挂载在 AHB 总线上，在 AHB 总线中有一个 DCMI 接口的数据寄存器，当我们读取该寄存器时，它会从 FIFO 中获取数据，并且 FIFO 中的数据指针会自动进行偏移，使得我们每次读取该寄存器都可获得一个新的数据。

(4) 控制/状态寄存器

DCMI 的控制寄存器协调图中的各个结构运行，程序中可通过检测状态寄存器来获 DCMI 的当前运行状态。

(5) DMA 接口

由于 DCMI 采集的数据量很大，我们一般使用 DMA 来把采集得的数据搬运至内存。

46.3.3 同步方式

DCMI 接口支持硬件同步或内嵌码同步方式，硬件同步方式即使用 HSYNC 和 VSYNC 作为同步信号的方式，OV5640 就是使用这种同步时序。

而内嵌码同步的方式是使用数据信号线传输中的特定编码来表示同步信息，由于需要用 0x00 和 0xFF 来表示编码，所以表示图像的数据中不能包含有这两个值。利用这两个值，它扩展到 4 个字节，定义出了 2 种模式的同步码，每种模式包含 4 个编码，编码格式为 0xFF0000XY，其中 XY 的值可通过寄存器设置。当 DCMI 接收到这样的编码时，它不会把这些当成图像数据，而是按照表 46-0-3 中的编码来解释，作为同步信号。

表 46-0-3 两种模式的内嵌码

模式 2 的内嵌码	模式 1 的内嵌码
帧开始 (FS)	有效行开始 (SAV)
帧结束 (FE)	有效行结束 (EAV)
行开始 (LS)	帧间消隐期内的行开始 (SAV)，其中消隐期内的即为无效数据
行结束 (LS)	帧间消隐期内的行结束 (EAV)，其中消隐期内的即为无效数据

46.3.4 捕获模式及捕获率

DCMI 还支持两种数据捕获模式，分别为快照模式和连续采集模式。快照模式时只采集一帧的图像数据，连续采集模式会一直采集多个帧的数据，并且可以通过配置捕获率来控制采集多少数据，如可配置为采集所有数据或隔 1 帧采集一次数据或隔 3 帧采集一次数据。

46.4 DCMI 初始化结构体

与其它外设一样，STM32 的 DCMI 外设也可以使用库函数来控制，其中最主要的配置项都封装到了 DCMI_InitTypeDef 结构体，来这些内容都定义在库文件“stm32f4xx_hal_dcmi.h”及“stm32f4xx_hal_dcmi.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。

DCMI_InitTypeDef 初始化结构体的内容见代码清单 46-0-1。

```
typedef struct {
   uint32_t SynchroMode; /* 选择硬件同步模式还是内嵌码模式 */
   uint32_t PCKPolarity; /* 设置像素时钟的有效边沿 */
   uint32_t VSPolarity;  /* 设置 VSYNC 的有效电平 */
   uint32_t HSPolarity;  /* 设置 HSYNC 的有效边沿 */
   uint32_t CaptureRate; /* 设置图像的采集间隔 */
   uint32_t ExtendedDataMode; /* 设置数据线的宽度 */
   DCMI_CodesInitTypeDef SyncroCode;/* 分隔符设置 */
   uint32_t JPEGMode;     /*JPEG 模式选择 */
   uint32_t ByteSelectMode; /* 配置字节选项模式 */
   uint32_t ByteSelectStart; /* 字节选择开始 */
   uint32_t LineSelectMode;  /* 行选择模式 */
   uint32_t LineSelectStart; /* 行选择选择 */
} DCMI_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) SynchroMode

本成员设置 DCMI 数据的同步模式，可以选择为硬件同步方式 (DCMI_SYNCHRO_HARDWARE) 或内嵌码方式 (DCMI_SYNCHRO_EMBEDDED)。

(2) PCKPolarity

本成员用于配置 DCMI 接口像素时钟的有效边沿，即在该时钟边沿时，DCMI 会对数据线上的信号进行采样，它可以被设置为上升沿有效 (DCMI_PCKPOLARITY_RISING) 或下降沿有效 (DCMI_PCKPOLARITY_FALLING)。

(3) VSPolarity

本成员用于设置 VSYNC 的有效电平，当 VSYNC 信号线表示为有效电平时，表示新的一帧数据传输完成，它可以被设置为高电平有效 (DCMI_VSPolarity_High) 或低电平有效 (DCMI_VSPolarity_Low)。

(4) DCMI_HSPolarity

类似地，本成员用于设置 HSYNC 的有效电平，当 HSYNC 信号线表示为有效电平时，表示

新的一行数据传输完成，它可以被设置为高电平有效 (DCMI_VSPOLARITY_HIGH) 或低电平有效 (DCMI_VSPOLARITY_LOW)。

(5) CaptureRate

本成员可以用于设置 DCMI 捕获数据的频率，可以设置为全采集、半采集或 1/4 采集 (DCMI_CR_ALL_FRAME/ DCMI_CR_ALTERNATE_2_FRAME/ DCMI_CR_ALTERNATE_4_FRAME)，在间隔采集的情况下，STM32 的 DCMI 外设会直接按间隔丢弃数据。

(6) ExtendedDataMode

本成员用于设置 DCMI 的数据线宽度，可配置为 8/10/12 及 14 位数据线宽 (DCMI_EXTEND_DATA_8B/10B/12B/14B)。

(7) ExtendedDataMode

本成员用于设置 DCMI 的数据线宽度，可配置为 8/10/12 及 14 位数据线宽 (DCMI_EXTEND_DATA_8B/10B/12B/14B)。

(8) SyncroCode

本成员用于设置 DCMI 的数据线指定行/帧开始分隔符和行/帧结束分隔符的代码。

(9) JPEGMode

本成员用于设置 DCMI 的数据输入模式，可配置为使能或者禁止 JPEG 模式。

(10) ByteSelectMode

本成员用于设置 DCMI 的数据字节的选择，可配置为全部接收（DCMI_BSM_ALL），每隔一个字节接收（DCMI_BSM_OTHER），每四个字节接收一个字节（DCMI_BSM_ALTERNATE_4），每四个字节接收两个字节（DCMI_BSM_ALTERNATE_2）。

(11) ByteSelectStart

本成员用于设置 DCMI 的数据字节开始选择，可配置为奇数或者偶数。

(12) LineSelectMode

本成员用于设置 DCMI 的行数据的采集，可配置全部采集或者隔行采集。

(13) LineSelectStart

本成员用于设置 DCMI 的行数据字节开始选择，可配置为奇数或者偶数。

配置完这些结构体成员后，我们调用库函数 HAL_DCMI_Init 即可把这些参数写入到 DCMI 的控制寄存器中，实现 DCMI 的初始化。

46.5 DCMI—OV5640 摄像头实验

本小节讲解如何使用 DCMI 接口从 OV5640 摄像头输出的 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“DCMI—OV5640 摄像头”工程配合阅读。

46.5.1 硬件设计

46.5.1.1 摄像头原理图

本实验采用的 OV5640 摄像头实物见实物图，其原理图见图 *46_0_16*。

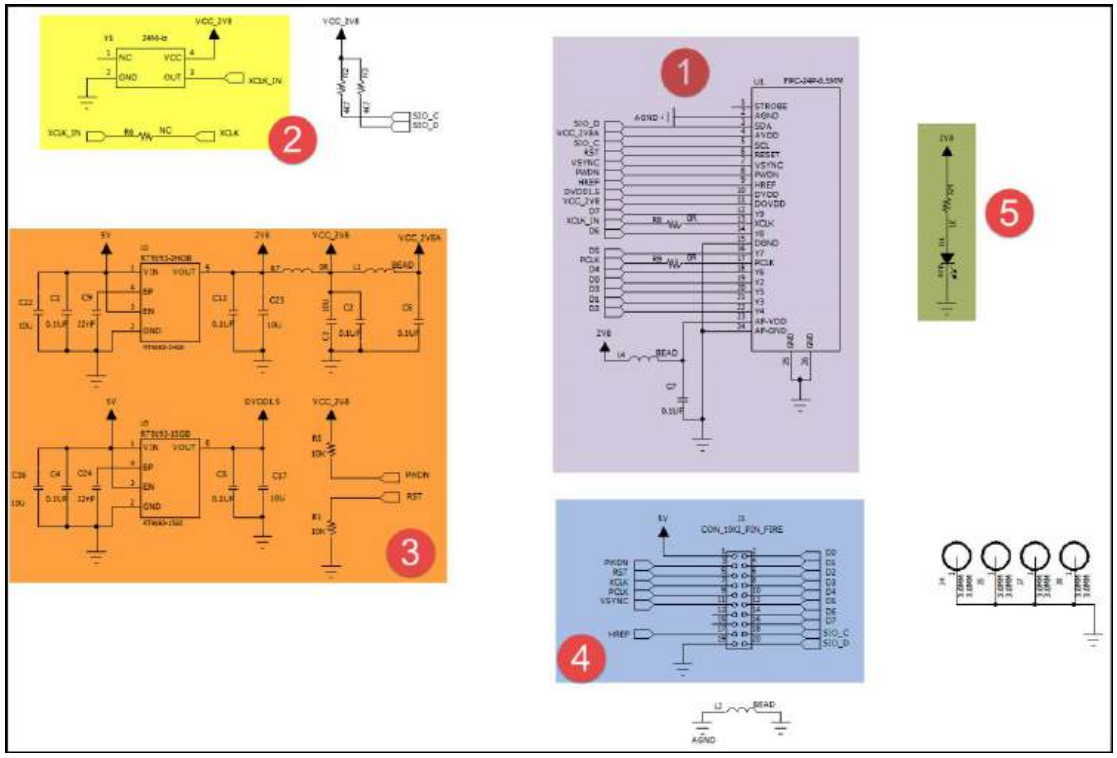


图 46-0-16 OV5640 摄像头原理图

图 *46_0_16* 标号 1 处的是 OV5640 模组接口电路，在这部分中已对 SCCB 使用的信号线接了上拉电阻，外部电路可以省略上拉；标号处的是一个 24MHz 的有源晶振，它为 OV5640 提供系统时钟，如果不想使用外部晶振提供时钟源，可以参考图中的 R6 处贴上 0 欧电阻，XCLK 引脚引出至外部，由外部控制器提供时钟；标号处的是电源转换模块，可以从 5V 转 2.8V 和 1.5V 供给模组使用；标号 ④ 处的是摄像头引脚集中引出的排针接口，使用它可以方便地与 STM32 实验板中的排母连接。标号 ⑤ 处的是电源指示灯。

46.5.1.2 摄像头与实验板的连接

通过排母，OV5640 与 STM32 引脚的连接关系见图 *46_0_17*。

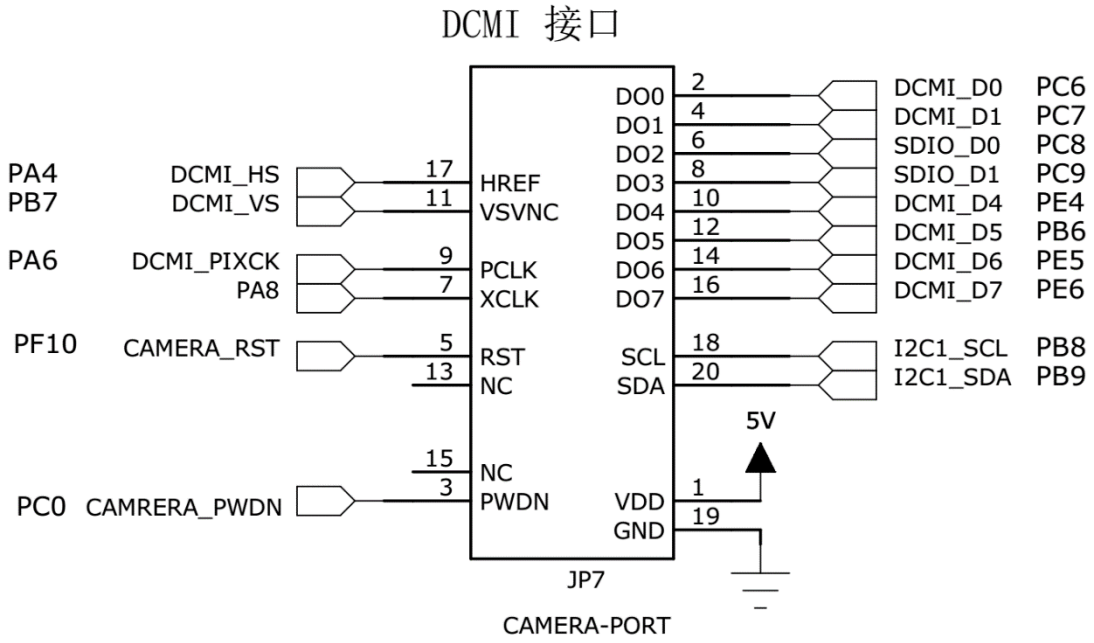


图 46-0-17 STM32 实验板引出的 DCMI 接口

以上原理图可查阅《ov5640—黑白原理图》及《野火 F407 开发板底板原理图》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。

46.5.2 软件设计

为了使工程更加有条理，我们把摄像头控制相关的代码独立分开存储，方便以后移植。在“液晶显示”工程的基础上新建“bsp_ov5640.c”，“ov5640_AF.c”，“bsp_ov5640.h”，“ov5640_AF.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。

46.5.2.1 编程要点

(1) 初始化 DCMI 时钟，I2C 时钟；

(2) 使用 I2C 接口向 OV5640 写入寄存器配置；

(3) 初始化 DCMI 工作模式；

(4) 初始化 DMA，用于搬运 DCMI 的数据到显存空间进行显示；

(5) 编写测试程序，控制采集图像数据并显示到液晶屏。

46.5.2.2 代码分析

46.5.2.2.1 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都以宏的形式定义到“bsp_ov5640.h”文件中，其中包括 I2C 及 DCMI 接口的，见代码清单 46-0-2。

```
/* 摄像头接口 */
//IIC SCCB
#define CAMERA_I2C                          I2C1
#define CAMERA_I2C_CLK_ENABLE()             __HAL_RCC_I2C1_CLK_ENABLE()

#define CAMERA_I2C_SCL_PIN                  GPIO_PIN_8
#define CAMERA_I2C_SCL_GPIO_PORT            GPIOB
#define CAMERA_I2C_SCL_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
#define CAMERA_I2C_SCL_AF                   GPIO_AF4_I2C1

#define CAMERA_I2C_SDA_PIN                  GPIO_PIN_9
#define CAMERA_I2C_SDA_GPIO_PORT            GPIOB
#define CAMERA_I2C_SDA_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
#define CAMERA_I2C_SDA_AF                   GPIO_AF4_I2C1

//VSYNC
```

(下页继续)

(续上页)

```
#define DCMI_VSYNC_GPIO_PORT                GPIOB
#define DCMI_VSYNC_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOB_CLK_ENABLE()
#define DCMI_VSYNC_GPIO_PIN                 GPIO_PIN_7
#define DCMI_VSYNC_AF                       GPIO_AF13_DCMI
// HSYNC
#define DCMI_HSYNC_GPIO_PORT                GPIOA
#define DCMI_HSYNC_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define DCMI_HSYNC_GPIO_PIN                 GPIO_PIN_4
#define DCMI_HSYNC_AF                       GPIO_AF13_DCMI
//PIXCLK
#define DCMI_PIXCLK_GPIO_PORT               GPIOA
#define DCMI_PIXCLK_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOA_CLK_ENABLE()
#define DCMI_PIXCLK_GPIO_PIN                GPIO_PIN_6
#define DCMI_PIXCLK_AF                      GPIO_AF13_DCMI
//PWDN
#define DCMI_PWDN_GPIO_PORT                 GPIOC
#define DCMI_PWDN_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_PWDN_GPIO_PIN                  GPIO_PIN_0

//RST
#define DCMI_RST_GPIO_PORT                  GPIOF
#define DCMI_RST_GPIO_CLK                  __HAL_RCC_GPIOF_CLK_ENABLE()
#define DCMI_RST_GPIO_PIN                   GPIO_PIN_10

//数据信号线
#define DCMI_D0_GPIO_PORT                 GPIOC
#define DCMI_D0_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_D0_GPIO_PIN                  GPIO_PIN_6
#define DCMI_D0_AF                        GPIO_AF13_DCMI

#define DCMI_D1_GPIO_PORT                 GPIOC
#define DCMI_D1_GPIO_CLK_ENABLE()         __HAL_RCC_GPIOC_CLK_ENABLE()
```

(下页继续)

(续上页)

```
#define DCMI_D1_GPIO_PIN                GPIO_PIN_7
#define DCMI_D1_AF                      GPIO_AF13_DCMI

#define DCMI_D2_GPIO_PORT               GPIOC
#define DCMI_D2_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_D2_GPIO_PIN                GPIO_PIN_8
#define DCMI_D2_AF                      GPIO_AF13_DCMI

#define DCMI_D3_GPIO_PORT               GPIOC
#define DCMI_D3_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOC_CLK_ENABLE()
#define DCMI_D3_GPIO_PIN                GPIO_PIN_9
#define DCMI_D3_AF                      GPIO_AF13_DCMI

#define DCMI_D4_GPIO_PORT               GPIOE
#define DCMI_D4_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOE_CLK_ENABLE()
#define DCMI_D4_GPIO_PIN                GPIO_PIN_4
#define DCMI_D4_AF                      GPIO_AF13_DCMI

#define DCMI_D5_GPIO_PORT               GPIOB
#define DCMI_D5_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOB_CLK_ENABLE()
#define DCMI_D5_GPIO_PIN                GPIO_PIN_6
#define DCMI_D5_AF                      GPIO_AF13_DCMI

#define DCMI_D6_GPIO_PORT               GPIOE
#define DCMI_D6_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOE_CLK_ENABLE()
#define DCMI_D6_GPIO_PIN                GPIO_PIN_5
#define DCMI_D6_AF                      GPIO_AF13_DCMI

#define DCMI_D7_GPIO_PORT               GPIOE
#define DCMI_D7_GPIO_CLK_ENABLE()       __HAL_RCC_GPIOE_CLK_ENABLE()
#define DCMI_D7_GPIO_PIN                GPIO_PIN_6
#define DCMI_D7_AF                      GPIO_AF13_DCMI
```

以上代码根据硬件的连接，把与 DCMI、I2C 接口与摄像头通讯使用的引脚号、引脚源以及复用功能映射都以宏封装起来。

46.5.2.2.2 初始化 DCMI 的 GPIO 及 I2C

利用上面的宏，初始化 DCMI 的 GPIO 引脚及 I2C，见代码清单 46-0-3。

```
/**
* @brief  初始化 I2C 总线，使用 I2C 前需要调用
* @param  无
* @retval 无
*/
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_HIGH;
   GPIO_InitStructure.Pull= GPIO_NOPULL;
   GPIO_InitStructure.Alternate=SENSORS_I2C_AF;
   HAL_GPIO_Init(SENSORS_I2C_SCL_GPIO_PORT, &GPIO_InitStructure);

   /* 配置 I2Cx 引脚: SDA ----------------------------------------*/
```

(下页继续)

(续上页)

```
    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.Timing           = 0x60201E2B;//100KHz
        I2C_Handle.Init.OwnAddress1      = 0;
        I2C_Handle.Init.AddressingMode   = I2C_ADDRESSINGMODE_7BIT;
        I2C_Handle.Init.DualAddressMode  = I2C_DUALADDRESS_DISABLE;
        I2C_Handle.Init.OwnAddress2      = 0;
        I2C_Handle.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
        I2C_Handle.Init.GeneralCallMode  = I2C_GENERALCALL_DISABLE;
        I2C_Handle.Init.NoStretchMode    = I2C_NOSTRETCH_DISABLE;

        /* 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);
    }
}
/**
* @brief  初始化控制摄像头使用的 GPIO(I2C/DCMI)
* @param  None
* @retval None
*/
```

(下页继续)

(续上页)

```
void OV5640_HW_Init(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;

   /***DCMI 引脚配置 ***/
   /* 使能 DCMI 时钟 */
   DCMI_PWDN_GPIO_CLK_ENABLE();
   DCMI_VSYNC_GPIO_CLK_ENABLE();
   DCMI_HSYNC_GPIO_CLK_ENABLE();
   DCMI_PIXCLK_GPIO_CLK_ENABLE();
   DCMI_D0_GPIO_CLK_ENABLE();
   DCMI_D1_GPIO_CLK_ENABLE();
   DCMI_D2_GPIO_CLK_ENABLE();
   DCMI_D3_GPIO_CLK_ENABLE();
   DCMI_D4_GPIO_CLK_ENABLE();
   DCMI_D5_GPIO_CLK_ENABLE();
   DCMI_D6_GPIO_CLK_ENABLE();
   DCMI_D7_GPIO_CLK_ENABLE();
   /* 控制/同步信号线 */
   GPIO_InitStructure.Pin = DCMI_VSYNC_GPIO_PIN;
   GPIO_InitStructure.Mode = GPIO_MODE_AF_PP;
   GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
   GPIO_InitStructure.Pull = GPIO_PULLUP ;
   GPIO_InitStructure.Alternate = DCMI_VSYNC_AF;
   HAL_GPIO_Init(DCMI_VSYNC_GPIO_PORT, &GPIO_InitStructure);

   GPIO_InitStructure.Pin = DCMI_HSYNC_GPIO_PIN;
   GPIO_InitStructure.Alternate = DCMI_VSYNC_AF;
   HAL_GPIO_Init(DCMI_HSYNC_GPIO_PORT, &GPIO_InitStructure);


   GPIO_InitStructure.Pin = DCMI_PIXCLK_GPIO_PIN;
```

(下页继续)

(续上页)

```
  GPIO_InitStructure.Alternate = DCMI_PIXCLK_AF;
  HAL_GPIO_Init(DCMI_PIXCLK_GPIO_PORT, &GPIO_InitStructure);

  /* 数据信号 */
  GPIO_InitStructure.Pin = DCMI_D0_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D0_AF;
  HAL_GPIO_Init(DCMI_D0_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.Pin = DCMI_D1_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D1_AF;
  HAL_GPIO_Init(DCMI_D1_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.Pin = DCMI_D2_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D2_AF;
  HAL_GPIO_Init(DCMI_D2_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.Pin = DCMI_D3_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D3_AF;
  HAL_GPIO_Init(DCMI_D3_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.Pin = DCMI_D4_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D4_AF;
  HAL_GPIO_Init(DCMI_D4_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.Pin = DCMI_D5_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D5_AF;
  HAL_GPIO_Init(DCMI_D5_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.Pin = DCMI_D6_GPIO_PIN;
  GPIO_InitStructure.Alternate = DCMI_D6_AF;
  HAL_GPIO_Init(DCMI_D6_GPIO_PORT, &GPIO_InitStructure);
```

(下页继续)

(续上页)

```
    GPIO_InitStructure.Pin = DCMI_D7_GPIO_PIN;
    GPIO_InitStructure.Alternate = DCMI_D7_AF;
    HAL_GPIO_Init(DCMI_D7_GPIO_PORT, &GPIO_InitStructure);

    GPIO_InitStructure.Pin = DCMI_PWDN_GPIO_PIN;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    HAL_GPIO_Init(DCMI_PWDN_GPIO_PORT, &GPIO_InitStructure);
    /*PWDN 引脚，高电平关闭电源，低电平供电 */
    HAL_GPIO_WritePin(DCMI_PWDN_GPIO_PORT,DCMI_PWDN_GPIO_PIN,GPIO_PIN_RESET);

}
```

函数中 I2C 的初始化配置，使用 I2C 与 OV2640 的 SCCB 接口通讯，这里的 I2C 模式配置与标准的 I2C 无异。

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，以上代码把 DCMI 接口的信号线全都初始化为 DCMI 复用功能，这里需要特别注意的地方是：OV5640 的上电时序比较特殊，我们初始化 PWDN 和 RST 应该特别小心，先初始化成普通的推挽输出模式，并且在初始化完毕后直接控制 RST 为低电平，PWDN 为高电平，使摄像头处于待机模式，延时 10ms 后控制 PWDN 为低电平，再延时 10ms 后控制 RST 为高电平，OV5640 模组启动。特别注意：IO 初始化完必须延时 50ms，再进行对 OV5640 寄存器的读写操作。

46.5.2.2.3 配置 DCMI 的模式

接下来需要配置 DCMI 的工作模式，我们通过编写 OV2640_Init 函数完成该功能，见代码清单 46-0-4。

```
/**
* @brief  配置 DCMI/DMA 以捕获摄像头数据
* @param  None
* @retval None
```

(下页继续)

(续上页)

```
*/
void OV5640_Init(void)
{
   /*** 配置 DCMI 接口 ***/
   /* 使能 DCMI 时钟 */
   __HAL_RCC_DCMI_CLK_ENABLE();

   /* DCMI 配置 */
   DCMI_Handle.Instance              = DCMI;
   DCMI_Handle.Init.SynchroMode      = DCMI_MODE_CONTINUOUS;
   DCMI_Handle.Init.SynchroMode      = DCMI_SYNCHRO_HARDWARE;
   DCMI_Handle.Init.PCKPolarity      = DCMI_PCKPOLARITY_RISING;
   DCMI_Handle.Init.VSPolarity       = DCMI_VSPOLARITY_LOW;
   DCMI_Handle.Init.HSPolarity       = DCMI_HSPOLARITY_LOW;
   DCMI_Handle.Init.CaptureRate      = DCMI_CR_ALL_FRAME;
   DCMI_Handle.Init.ExtendedDataMode = DCMI_EXTEND_DATA_8B;
   HAL_DCMI_Init(&DCMI_Handle);

   /* 配置中断 */
   HAL_NVIC_SetPriority(DCMI_IRQn, 5, 0);
   HAL_NVIC_EnableIRQ(DCMI_IRQn);

   //开始传输，数据大小以 32 位数据为单位 (即像素个数/4，LCD_GetXSize()*LCD_
↪GetYSize()*2/4)
   OV5640_DMA_Config(LCD_FB_START_ADDRESS,LCD_GetXSize()*LCD_GetYSize()/2);
}
```

该函数的执行流程如下：

(1) 使能 DCMI 外设的时钟，它是挂载在 AHB2 总线上的；

(2) 根据摄像头的时序和硬件连接的要求，配置 DCMI 工作模式为：使用硬件同步，连续采集所有帧数据，采集时使用 8 根数据线，PIXCLK 被设置为上升沿有效，VSYNC 和 HSYNC

都被设置成低电平有效；

(3) 调用 OV2640_DMA_Config 函数开始 DMA 数据传输，每传输完一帧数据需要调用一次，它包含本次传输的目的首地址及传输的数据量，后面我们再详细解释；

(4) 配置 DMA 中断，DMA 每次传输完毕会引起中断，以便我们在中断服务函数配置 DMA 传输下一帧数据；

(5) 配置 DCMI 的帧传输中断，为了防止有时 DMA 出现传输错误或传输速度跟不上导致数据错位、偏移等问题，每次 DCMI 接收到摄像头的一帧数据，得到新的帧同步信号后 (VSYNC)，就进入中断，复位 DMA，使它重新开始一帧的数据传输。

46.5.2.2.4 配置 DMA 数据传输

上面的 DCMI 配置函数中调用了 OV5640_DMA_Config 函数开始了 DMA 传输，该函数的定义见代码清单 46-0-5。

```
/**
* @brief  配置 DCMI/DMA 以捕获摄像头数据
* @param  DMA_Memory0BaseAddr: 本次传输的目的首地址
* @param DMA_BufferSize: 本次传输的数据量 (单位为字, 即 4 字节)
*/
void OV5640_DMA_Config(uint32_t DMA_Memory0BaseAddr,uint32_t DMA_
↪BufferSize)
{
    /* 配置 DMA 从 DCMI 中获取数据 */
    /* 使能 DMA*/
    __HAL_RCC_DMA2_CLK_ENABLE();
    DMA_Handle_dcmi.Instance = DMA2_Stream1;
    DMA_Handle_dcmi.Init.Channel = DMA_CHANNEL_1;
    DMA_Handle_dcmi.Init.Direction = DMA_PERIPH_TO_MEMORY;
    DMA_Handle_dcmi.Init.PeriphInc = DMA_PINC_DISABLE;
    DMA_Handle_dcmi.Init.MemInc = DMA_MINC_ENABLE;       //寄存器地址自增
    DMA_Handle_dcmi.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
```

(下页继续)

(续上页)

```
    DMA_Handle_dcmi.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    DMA_Handle_dcmi.Init.Mode = DMA_CIRCULAR;                    //循环模式
    DMA_Handle_dcmi.Init.Priority = DMA_PRIORITY_HIGH;
    DMA_Handle_dcmi.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
    DMA_Handle_dcmi.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
    DMA_Handle_dcmi.Init.MemBurst = DMA_MBURST_SINGLE;
    DMA_Handle_dcmi.Init.PeriphBurst = DMA_PBURST_SINGLE;

    /*DMA 中断配置 */
    __HAL_LINKDMA(&DCMI_Handle, DMA_Handle, DMA_Handle_dcmi);

    HAL_NVIC_SetPriority(DMA2_Stream1_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(DMA2_Stream1_IRQn);

    HAL_DMA_Init(&DMA_Handle_dcmi);
    //使能 DCMI 采集数据
    HAL_DCMI_Start_DMA(&DCMI_Handle, DCMI_MODE_CONTINUOUS,
    (uint32_t)DMA_Memory0BaseAddr,DMA_BufferSize);
}
```

该函数跟普通的 DMA 配置无异，它把 DCMI 接收到的数据从它的数据寄存器搬运到 SDRAM 显存中，从而直接使用液晶屏显示摄像头采集得的图像。它包含 2 个输入参数 DMA_Memory0BaseAddr 和 DMA_BufferSize，其中 DMA_Memory0BaseAddr 用于设置本次 DMA 传输的目的首地址，DMA_BufferSize 则用于指示本次 DMA 传输的数据量，要注意它的单位是一个字，即 4 字节，如我们要传输 60 字节的数据时，它应配置为 15。这两参数会被传递到库函数 HAL_DCMI_Start_DMA 中作为形参。在前面的 OV5640_Init 函数中，对这个函数有如下调用：

```
/* 摄像头采集图像的大小，改变这两个值可以改变数据量，
但不会加快采集速度，要加快采集速度需要改成 SVGA 械 */
#define img_width  LCD_GetXSize()
#define img_height LCD_GetYSize()
```

(下页继续)

(续上页)

```
//开始传输，数据大小以 32 位数据为单位 (即像素个数/4,
//LCD_GetXSize()*LCD_GetYSize()*2/4)
OV5640_DMA_Config(LCD_FB_START_ADDRESS,LCD_GetXSize()*LCD_GetYSize()/2);
```

其中的 LCD_GetXSize 和 LCD_GetYSize 获取液晶屏的分辨率，img_width 和 img_heigh 表示摄像头输出的图像的分辨率，LCD_FB_START_ADDRESS 是液晶层的首个显存地址。另外，本工程中显示摄像头数据的这个液晶层采用 RGB565 的像素格式，每个像素点占据 2 个字节。把摄像头输出的每一帧数据显示到液晶屏上，不需要额外的处理这样最简单直接。

46.5.2.2.5 DMA 传输完成中断及帧中断

OV5640_Init 函数初始化了 DCMI，使能了帧中断、DMA 传输完成中断，并使能了第一次 DMA 传输，当这一行数据传输完成时，会进入 DMA 中断服务函数，见代码清单 46-6 中的 DMA2_Stream1_IRQHandler。

```
/**
* @brief  DMA 中断服务函数
* @param  None
* @retval None
*/
void DMA2_Stream1_IRQHandler(void)
{
    HAL_DMA_IRQHandler(&DMA_Handle_dcmi);
}

/**
* @brief  DCMI 中断服务函数
* @param  None
* @retval None
*/
void DCMI_IRQHandler(void)
```

(下页继续)

(续上页)

```
{
    HAL_DCMI_IRQHandler(&DCMI_Handle);
}
/**
* @brief  帧同步回调函数.
* @param  None
* @retval None
*/
void HAL_DCMI_VsyncEventCallback(DCMI_HandleTypeDef *hdcmi)
{
    fps++; //帧率计数
    OV5640_DMA_Config(LCD_FB_START_ADDRESS,
LCD_GetXSize()*LCD_GetYSize()/2);
}
```

DMA 中断服务函数中直接调用库函数进行处理。当 DCMI 接口检测到摄像头传输的帧同步信号时，会进入 DCMI_IRQHandler 中断服务函数，DCMI 中断服务函数中直接调用库函数进行处理。每次帧同步来临是重新设置一次 DMA 传输数据，液晶的显存就会收到摄像头采集的数据然后显示在液晶上。

46.5.2.2.6 读取 OV5640 芯片 ID

配置完了 STM32 的 DCMI，还需要控制摄像头，它有很多寄存器用于配置工作模式。利用 STM32 的 I2C 接口，可向 OV5640 的寄存器写入控制参数，我们先写个读取芯片 ID 的函数测试一下，见代码清单 46-0-7。

```
//存储摄像头 ID 的结构体
typedef struct {
    uint8_t PIDH;
    uint8_t PIDL;
} OV5640_IDTypeDef;
```

(下页继续)

(续上页)

```
#define OV5640_SENSOR_PIDH       0x300A
#define OV5640_SENSOR_PIDL       0x300B
/**
* @brief  读取摄像头的 ID.
* @param  OV5640ID: 存储 ID 的结构体
* @retval None
*/
void OV5640_ReadID(OV5640_IDTypeDef *OV5640ID)
{

    /* 读取寄存芯片 ID*/
    OV5640ID->PIDH = OV5640_ReadReg(OV5640_SENSOR_PIDH);
    OV5640ID->PIDL = OV5640_ReadReg(OV5640_SENSOR_PIDL);
}
/**
* @brief  从 OV5640 寄存器中读取一个字节的数据
* @param  Addr: 寄存器地址
* @retval 返回读取得的数据
*/
uint8_t OV5640_ReadReg(uint16_t Addr)
{
    uint8_t Data = 0;

    HAL_StatusTypeDef status = HAL_OK;

    status = HAL_I2C_Mem_Read(&I2C_Handle, OV5640_DEVICE_ADDRESS,
        (uint16_t)Addr, I2C_MEMADD_SIZE_16BIT, (uint8_t*)&Data,1, 1000);

    /* 检查通信状态 */
    if (status != HAL_OK) {
        /* 发生错误重新初始化 I2C */
        I2Cx_Error();
```

(下页继续)

(续上页)

```
    }
    /* 返回读到的数据 */
    return Data;
}
```

在 OV5640 的 PIDH 及 PIDL 寄存器存储了产品 ID，PIDH 的默认值为 0x56，PIDL 的默认值为 0x40。在代码中我们定义了一个结构体 OV5640_IDTypeDef 专门存储这些读取得的 ID 信息。

OV5640_ReadID 函数中使用的 OV5640_ReadReg 函数是使用 STM32 的 I2C 外设向某寄存器读写单个字节数据的底层函数，它与我们前面章节中用到的 I2C 函数差异是 OV5640 的寄存器地址是 16 位的所以要先设置为 I2C_MEMADD_SIZE_16BIT 然后再读取寄存器的值。

46.5.2.2.7 向 OV5640 写入寄存器配置

检测到 OV5640 的存在后，向它写入配置参数，见代码清单 46-0-8。

```
/**
* @brief  Configures the OV5640 camera in BMP mode.
* @param  BMP ImageSize: BMP image size
* @retval None
*/
void OV5640_RGB565Config(void)
{
    uint32_t i;

    /* 摄像头复位 */
    OV5640_Reset();
    /* 写入寄存器配置 */
    /* Initialize OV5640   Set to output RGB565 */
    for (i=0; i<(sizeof(RGB565_Init)/4); i++) {
        OV5640_WriteReg(RGB565_Init[i][0], RGB565_Init[i][1]);
            Delay(5);
```

(下页继续)

(续上页)

```
    }

    if (img_width == 320)

        ImageFormat=BMP_320x240;

    else if (img_width == 640)

        ImageFormat=BMP_640x480;

    else if (img_width == 800)

        ImageFormat=BMP_800x480;

    switch (ImageFormat) {
    case BMP_320x240: {
        for (i=0; i<(sizeof(RGB565_QVGA)/4); i++) {
            OV5640_WriteReg(RGB565_QVGA[i][0], RGB565_QVGA[i][1]);
        }
        break;
    }
    case BMP_640x480: {
        for (i=0; i<(sizeof(RGB565_VGA)/4); i++) {
            OV5640_WriteReg(RGB565_VGA[i][0], RGB565_VGA[i][1]);
        }
        break;
    }
    case BMP_800x480: {
        for (i=0; i<(sizeof(RGB565_WVGA)/4); i++) {
            OV5640_WriteReg(RGB565_WVGA[i][0], RGB565_WVGA[i][1]);
        }
        break;
```

(下页继续)

(续上页)

```
    }
    default: {
        for (i=0; i<(sizeof(RGB565_WVGA)/4); i++) {
            OV5640_WriteReg(RGB565_WVGA[i][0], RGB565_WVGA[i][1]);
        }
        break;
    }
    }
}
/**
* @brief  写一字节数据到 OV5640 寄存器
* @param  Addr: OV5640 的寄存器地址
* @param  Data: 要写入的数据
* @retval 返回 0 表示写入正常，0xFF 表示错误
*/
uint8_t OV5640_WriteReg(uint16_t Addr, uint8_t Data)
{
    HAL_StatusTypeDef status = HAL_OK;

    status = HAL_I2C_Mem_Write(&I2C_Handle, OV5640_DEVICE_ADDRESS,
(uint16_t)Addr, I2C_MEMADD_SIZE_16BIT, (uint8_t*) &Data, 1, 1000);

    /* Check the communication status */
    if (status != HAL_OK) {
        /* Re-Initiaize the I2C Bus */
        I2Cx_Error();
    }
    return status;
}
```

这个 OV5640_RGB565Config 函数直接把一个初始化的二维数组 RGB565_Init 和一个分辨率设置的二维数组 RGB565_WVGA(分辨率决定) 使用 I2C 传输到 OV5640 中，二维数组的第一维存储的

是寄存器地址，第二维存储的是对应寄存器要写入的控制参数。OV5640_WriteReg 函数中，因为 OV5640 的寄存器地址为 16 位，所以要先设置为 I2C_MEMADD_SIZE_16BIT 然后再写入寄存器的值，这个是有别于普通的 I2C 设备的写入方式，需要特别注意。

如果您对这些寄存器配置感兴趣，可以一个个对着 OV5640 的寄存器说明来阅读，这些配置主要是把 OV5640 配置成了 WVGA 时序模式，并使用 8 根数据线输出格式为 RGB565 的图像数据。我们参考《OV5640_ 自动对焦照相模组应用指南 (DVP_ 接口)__R2.13C.pdf》文档中第 20 页 4.1.3 节的 800x480 预览的寄存器参数进行配置。使摄像头输出为 WVGA 模式。

4.1.3 800x480 预览

800x480 15 帧/秒

```
// 800x480 15fps, night mode 5fps
// input clock 24Mhz, PCLK 45.6Mhz
write_i2c(0x3035, 0x41);      // PLL
write_i2c(0x3036, 0x72);      // PLL
write_i2c(0x3c07, 0x08);      // light meter 1 threshold[7:0]
write_i2c(0x3820, 0x41);      // flip
write_i2c(0x3821, 0x07);      // mirror
write_i2c(0x3814, 0x31);      // timing X inc
```

46.5.2.2.8 初始化 OV5640 自动对焦功能

写入 OV5640 的配置参数后，需要向它写入自动对焦固件，初始化自动对焦功能，才能使用自动对焦功能，见代码清单 46-0-9。

```
void OV5640_AUTO_FOCUS(void)
{
    OV5640_FOCUS_AD5820_Init();
    OV5640_FOCUS_AD5820_Constant_Focus();
```

(下页继续)

(续上页)

```
}
static void OV5640_FOCUS_AD5820_Init(void)
{
    u8  state=0x8F;
    u32 iteration = 100;
    u16 totalCnt = 0;

    CAMERA_DEBUG("OV5640_FOCUS_AD5820_Init\n");

    OV5640_WriteReg(0x3000, 0x20);
    totalCnt = sizeof(OV5640_AF_FW);
    CAMERA_DEBUG("Total Count = %d\n", totalCnt);

//  写入自动对焦固件 Brust mode
    OV5640_WriteFW(OV5640_AF_FW,totalCnt);

    OV5640_WriteReg(0x3022, 0x00);
    OV5640_WriteReg(0x3023, 0x00);
    OV5640_WriteReg(0x3024, 0x00);
    OV5640_WriteReg(0x3025, 0x00);
    OV5640_WriteReg(0x3026, 0x00);
    OV5640_WriteReg(0x3027, 0x00);
    OV5640_WriteReg(0x3028, 0x00);
    OV5640_WriteReg(0x3029, 0xFF);
    OV5640_WriteReg(0x3000, 0x00);
    OV5640_WriteReg(0x3004, 0xFF);
    OV5640_WriteReg(0x0000, 0x00);
    OV5640_WriteReg(0x0000, 0x00);
    OV5640_WriteReg(0x0000, 0x00);
    OV5640_WriteReg(0x0000, 0x00);

    do {
```

(下页继续)

(续上页)

```
        state = (u8)OV5640_ReadReg(0x3029);
        CAMERA_DEBUG("when init af, state=0x%x\n",state);

        Delay(10);
        if (iteration-- == 0) {
            CAMERA_DEBUG("[OV5640]STA_FOCUS state check ERROR!!,state=0x%x\
↪n",state);
            break;
        }
    } while (state!=0x70);

    OV5640_FOCUS_AD5820_Check_MCU();
    return;
}   /*  OV5640_FOCUS_AD5820_Init  */

//set constant focus
void OV5640_FOCUS_AD5820_Constant_Focus(void)
{
    u8 state = 0x8F;
    u32 iteration = 300;
    //send constant focus mode command to firmware
    OV5640_WriteReg(0x3023,0x01);
    OV5640_WriteReg(0x3022,0x04);

    iteration = 5000;
    do {
        state = (u8)OV5640_ReadReg(0x3023);
        if (iteration-- == 0) {
            CAMERA_DEBUG("[OV5640]AD5820_Single_Focus time out !!%x\n",
↪state);
            return ;
        }
```

(下页继续)

(续上页)

```
        Delay(10);
    } while (state!=0x00); //0x0 : focused 0x01: is focusing
    return;
}
```

OV5640_AUTO_FOCUS 函数调用了 OV5640_FOCUS_AD5820_Init 函数和 OV5640_FOCUS_

AD5820_Constant_Focus 函数，我们先来介绍 OV5640_FOCUS_AD5820_Init 函数，首先复位 OV5640 内部的 MCU，然后通过 I2C 的突发模式写入自动对焦固件，突发模式就是只需要写入首地址，接着就一直写数据，这个过程地址会自增，直接写完数据位置，对于连续地址写入相当方便。写入固件之后 OV5640 内部 MCU 开始初始化，最后检查初始化完成的状态是否为 0x70，如果是就代表固件已经写入成功，并初始化成功。接着，我们需要 OV5640_FOCUS_AD5820_Constant_Focus 函数来调用自动对焦固件中的持续对焦指令，完成以上步骤后，摄像头就已经初始化完毕。

46.5.2.2.9 main 函数

最后我们来编写 main 函数，利用前面讲解的函数，控制采集图像。

```
//修改 cam_mode 参数，修改后需要重新调用配置函数才生效
void Camera_Mode_Reconfig(void)
{
    cam_mode.frame_rate = FRAME_RATE_30FPS,

            //ISP 窗口
            cam_mode.cam_isp_sx = 0;
    cam_mode.cam_isp_sy = 0;

    cam_mode.cam_isp_width = 1920;
    cam_mode.cam_isp_height = 1080;

    //输出窗口
    cam_mode.scaling = 1;       //使能自动缩放
```

(下页继续)

(续上页)

```
    cam_mode.cam_out_sx = 16; //使能自动缩放后，一般配置成 16 即可
    cam_mode.cam_out_sy = 4;     //使能自动缩放后，一般配置成 4 即可
    cam_mode.cam_out_width = 320;
    cam_mode.cam_out_height = 240;

    //LCD 位置
    cam_mode.lcd_sx = 270;
    cam_mode.lcd_sy = 120;
    cam_mode.lcd_scan = 5; //LCD 扫描模式，

    //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 = 0;     //正常模式
    cam_mode.exposure = 0;

    cam_mode.auto_focus = 1;//自动对焦
}


//【*】注意事项:
//本程序暂时不支持摄像头延长线!!!
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    uint8_t focus_status = 0;
```

(下页继续)

(续上页)

```
    ILI9806G_Init ();          //LCD 初始化

    LCD_SetFont(&Font16x32);
    LCD_SetColors(RED,BLACK);

    ILI9806G_Clear(0,0,LCD_X_LENGTH,LCD_Y_LENGTH);  /* 清屏，显示全黑 */

    Debug_USART_Config();

    /* 配置 SysTick 为 1ms 中断一次，时间到后触发定时中断，
    * 进入 stm32fxx_it.c 文件的 SysTick_Handler 处理，通过数中断次数计时
    */
    SysTick_Init();

    Key_GPIO_Config();

    //液晶扫描方向
    ILI9806G_GramScan(5);

    CAMERA_DEBUG("STM32F407 DCMI 驱动 OV5640 例程");

    /* 初始化摄像头 GPIO 及 IIC */
    OV5640_HW_Init();

    /* 读取摄像头芯片 ID，确定摄像头正常连接 */
    OV5640_ReadID(&OV5640_Camera_ID);

    if (OV5640_Camera_ID.PIDH  == 0x56) {
        sprintf((char*)dispBuf, "OV5640 Camera ID:0x%x,initializing... ",
OV5640_Camera_ID.PIDH);
        ILI9806G_DispStringLine_EN(LINE(0),dispBuf);
```

(下页继续)

(续上页)

```
        CAMERA_DEBUG("检测到摄像头 %x %x",
OV5640_Camera_ID.PIDH ,OV5640_Camera_ID.PIDL);

    } else {
        LCD_SetTextColor(RED);
        ILI9806G_DispString_EN(10,10, "Can not detect OV5640 module,
please check the connection!");
        CAMERA_DEBUG("没有检测到 OV5640 摄像头，请重新检查连接。");

        while (1);
    }


    OV5640_Init();

    OV5640_RGB565_Default_Config();

    OV5640_USER_Config();

    OV5640_FOCUS_AD5820_Init();


    if (cam_mode.auto_focus ==1) {
        OV5640_FOCUS_AD5820_Constant_Focus();
        focus_status = 1;
    }

    /*DMA 直接传输摄像头数据到 LCD 屏幕显示 */
    ImagDisp();

    while (1) {
        //运行时动态修改摄像头参数示例
```

(下页继续)

(续上页)

```
    if ( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON  ) {
        //关闭采集
        OV5640_Capture_Control(DISABLE);

        //修改 Cam mode 参数
        Camera_Mode_Reconfig();

        OV5640_USER_Config();

        if (cam_mode.auto_focus ==1) {
            OV5640_FOCUS_AD5820_Constant_Focus();
            focus_status = 1;
        }

        /*DMA 直接传输摄像头数据到 LCD 屏幕显示 */
        ImagDisp();

    }

    //反转对焦状态，若原来在持续对焦则暂停，否则开启
    if ( Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON  ) {
        if (focus_status == 1) {
            //暂停对焦
            OV5640_FOCUS_AD5820_Pause_Focus();
            focus_status = 0 ;
        } else {
            //自动对焦
            OV5640_FOCUS_AD5820_Constant_Focus();
            focus_status = 1 ;
        }
    }
```

(下页继续)

(续上页)

```
        //使用串口输出帧率
        if (Task_Delay[0]==0) {
            /* 输出帧率 */
            CAMERA_DEBUG("\r\n帧率:%.1f/s \r\n", (double)fps/5.0);
            //重置
            fps =0;

            Task_Delay[0]=5000; //此值每 1ms 会减 1，减到 0 才可以重新进来这里

        }
    }
}
```

阅读本代码时，先跳过前面的 Camera_Mode_Reconfig 函数。先来看在 main 函数，它初始化了液晶屏、串口、按键和定时器等外设。

摄像头控制部分，首先调用了 OV5640_HW_Init 函数初始化 DCMI 的 GPIO 及 I2C，然后调用 OV5640_ReadID 函数检测摄像头与实验板是否正常连接，若连接正常则调用 OV5640_Init 函数初始化 DCMI 的工作模式及 DMA，再调用 OV5640_RGB565_Default_Config 函数向 OV5640 写入默认的寄存器配置，调用 OV5640_USER_Config 把全局变量 cam_mode 的值写入摄像头，调用 OV5640_FOCUS_AD5820_Init 函数设置自动对焦，最后，一定要记得调用 ImagDisp 使能 DCMI 开始捕获数据，这样才能正常开始工作。

在 main 函数的 while 循环里，循环检测按键，若按下 key1 按键，它会调用开头部分的 Camera_Mode_Reconfig 函数设置新的 cam_mode 值，然后调用 OV5640_USER_Config 更新摄像头配置，它演示了如何在程序运行时利用 cam_mode 更新摄像头配置。

在 while 循环中，还有一个条件判断，它使用定时器每 5 秒统计一次帧率，并使用串口输出。

46.5.3 下载验证

把 OV5640 接到实验板的摄像头接口中，用 USB 线连接开发板，编译程序下载到实验板，并上电复位，液晶屏会显示摄像头采集得的图像，摄像头会自动调焦。

第 47 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。

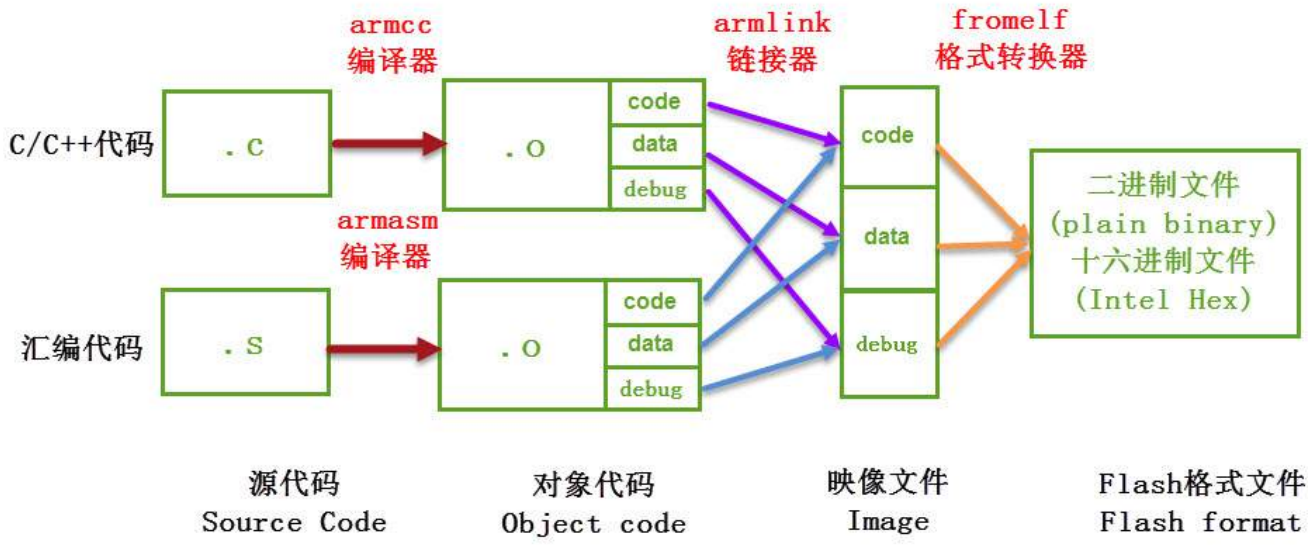
在本章中讲解了非常多的文件类型，学习时请跟着教程的节奏，打开实际工程中的文件来了解。

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。

47.1 编译过程

47.1.1 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。

47.1.2 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*。

```
*** Using Compiler 'V5.05 update 2 (build 169)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target '多彩流水灯'
assembling startup_stm32f40xx.s...
compiling system_stm32f4xx.c...
compiling misc.c...
compiling stm32f4xx_adc.c...
compiling stm32f4xx_can.c...
compiling stm32f4xx_cec.c...
compiling stm32f4xx_crc.c...
compiling stm32f4xx_cryp.c...
compiling stm32f4xx_usart.c...
compiling stm32f4xx_wwdg.c...
compiling main.c...
compiling stm32f4xx_it.c...
compiling bsp_led.c...
linking...
Program Size: Code=920 RO-data=408 RW-data=0 ZI-data=1024
FromELF: creating hex file...
"..\..\Output\多彩流水灯.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:25
```

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。

armar.exe	2015/7/27 18:03	应用程序	1,549 KB
armasm.exe	2015/7/27 18:03	应用程序	5,871 KB
armcc.exe	2015/7/27 18:03	应用程序	15,571 KB
armcompiler_libFNP.dll	2015/7/27 18:03	DLL 文件	4,656 KB
armlink.exe	2015/7/27 18:03	应用程序	6,379 KB
fromelf.exe	2015/7/27 18:03	应用程序	5,307 KB

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。

47.2 程序的组成、存储与运行

47.2.1 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)。

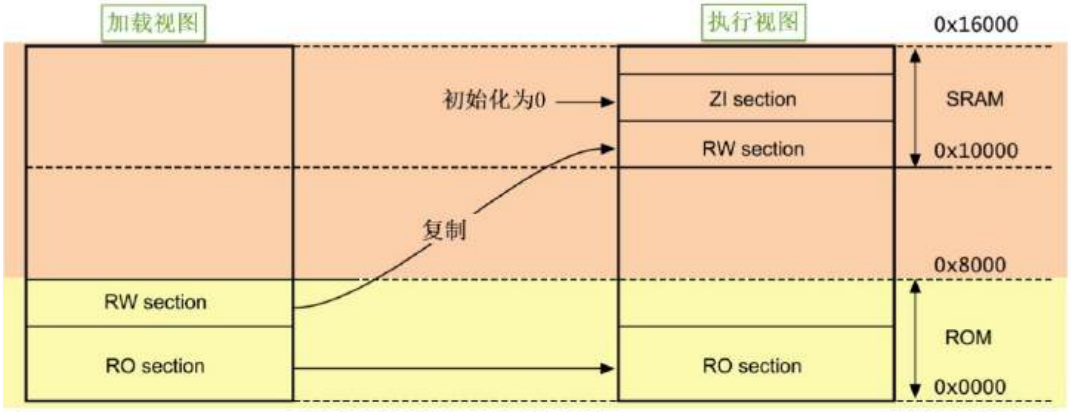
  综上所述，以程序的组成构件为例，它们所属的区域类别见表 40-1。

  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

47.2.2 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。

47.3 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。

47.3.1 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:\work\keil5\ARM\ARMCC\bin”。

47.3.1.1 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*；

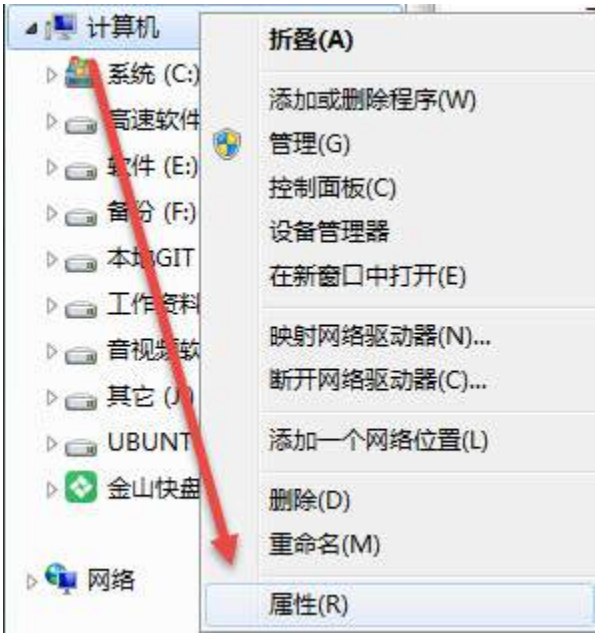


图 40-6 计算机属性页面

(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*；

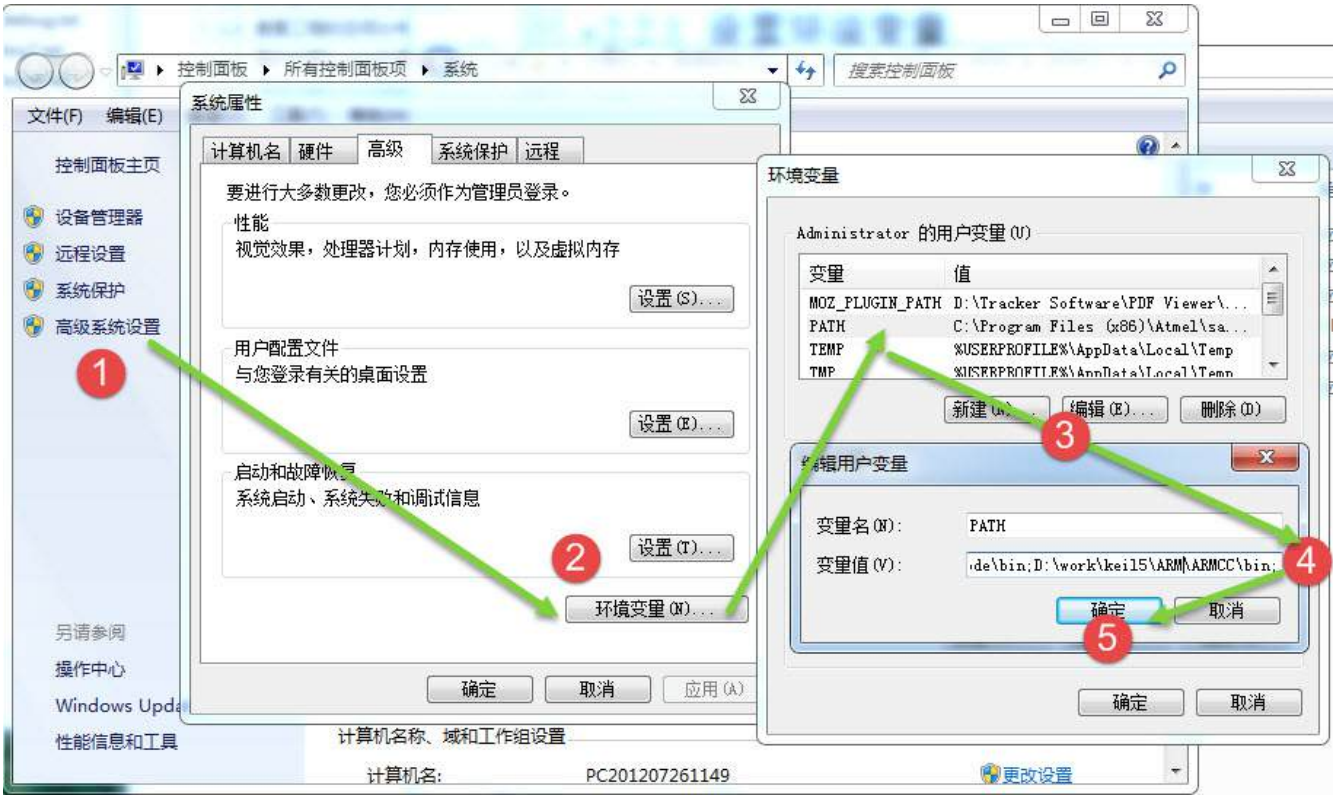


图 40-7 添加工具链路径到 PATH 变量

(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。

47.3.2 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。

47.3.2.1 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*

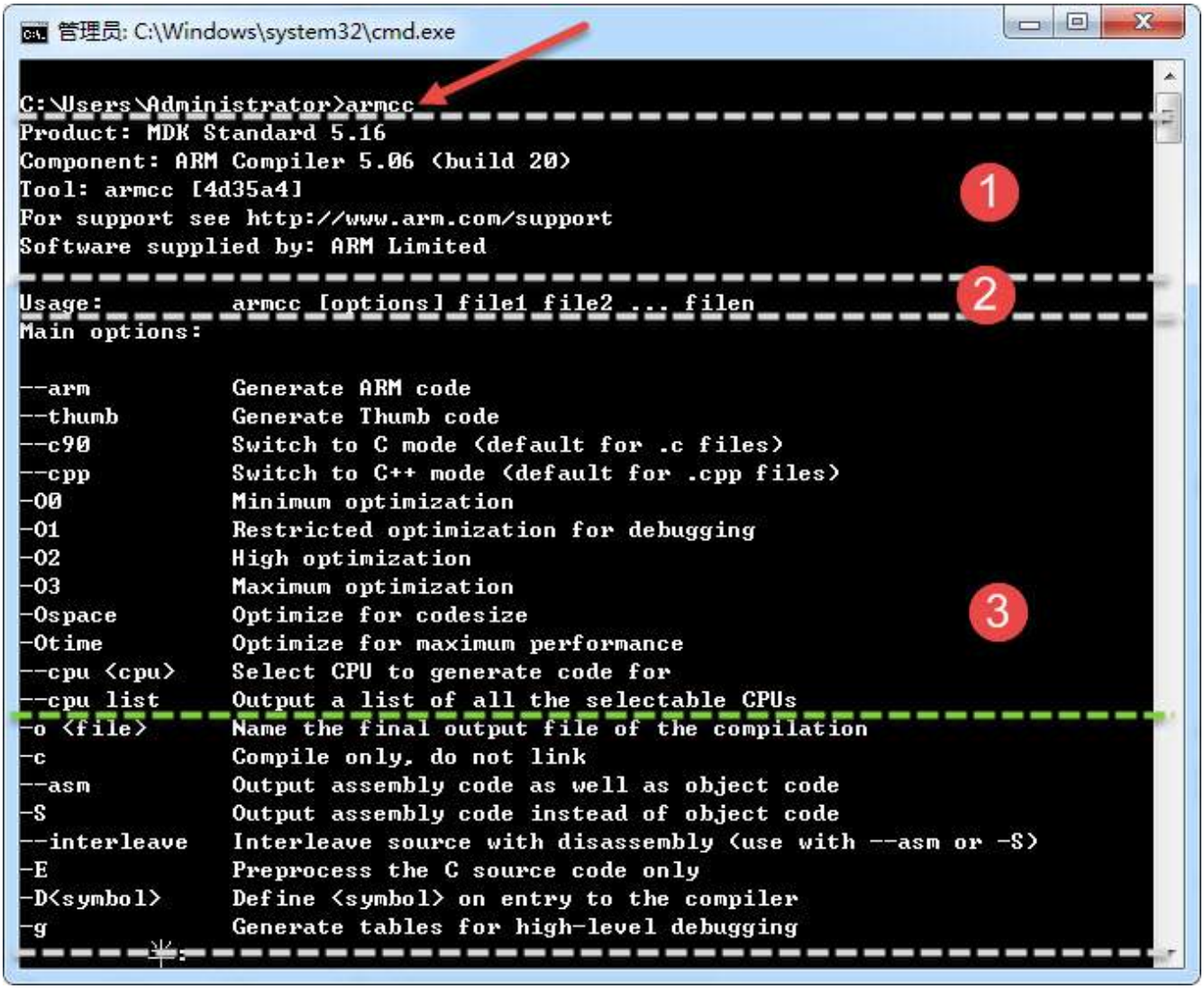


图 40-9 armcc 的帮助提示

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。

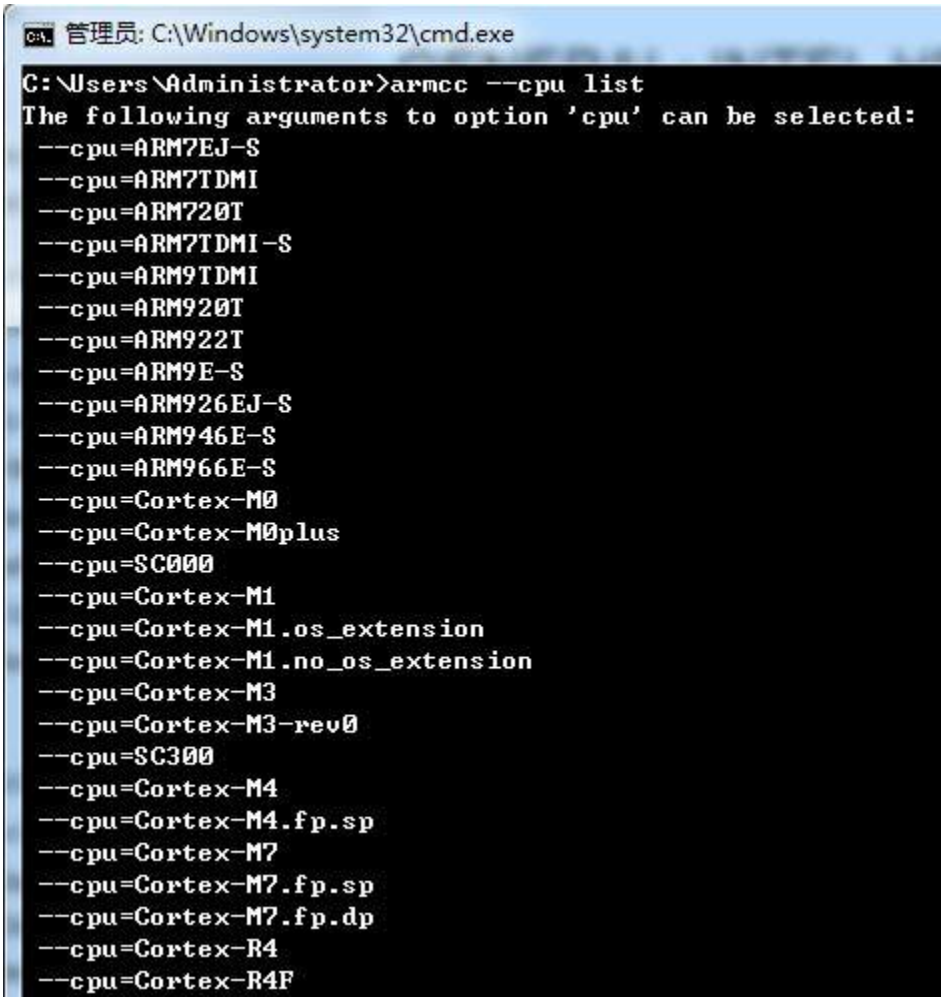


图 40-10 cpulist

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*。

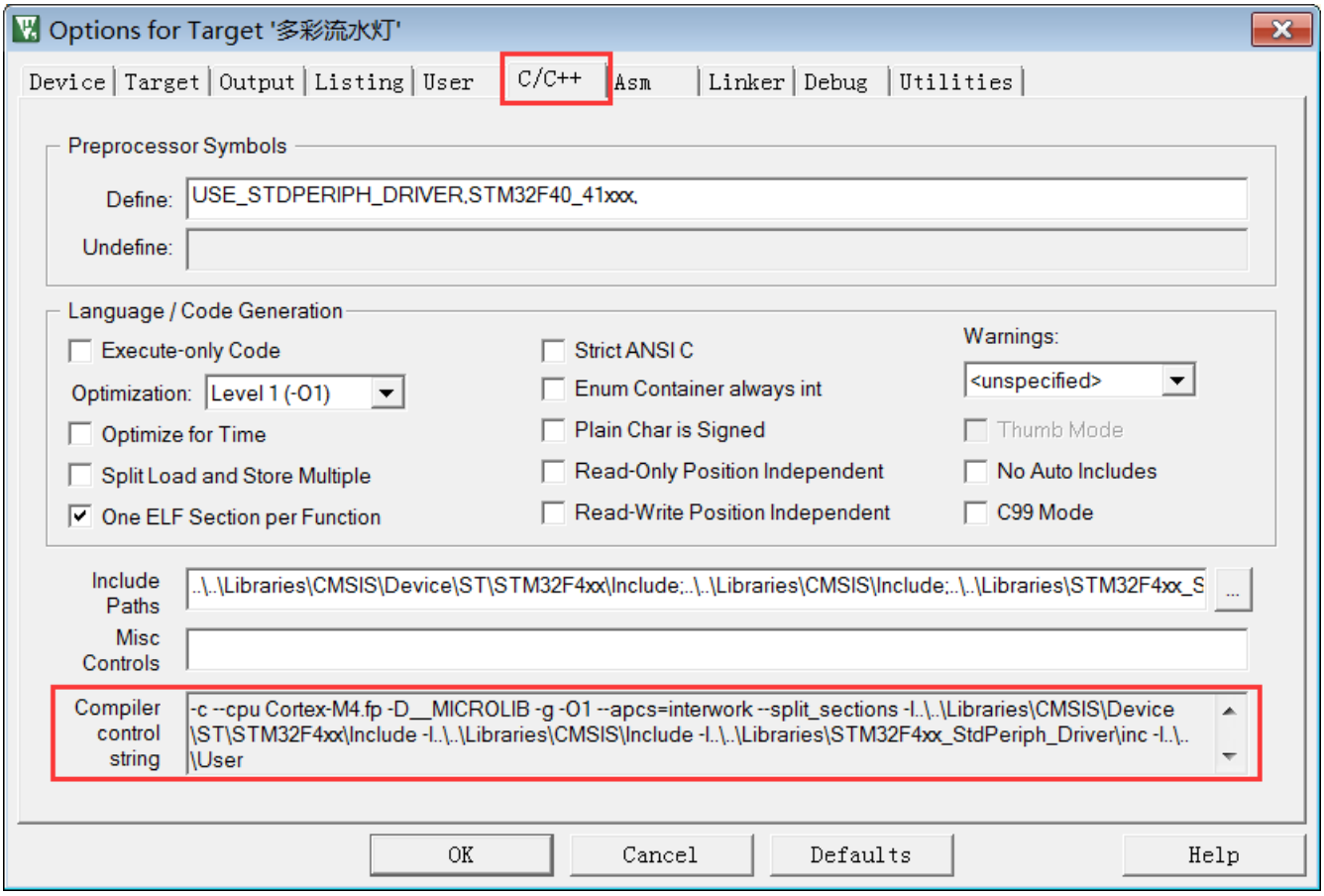


图 40-11 MDK 的 ARMCC 编译选项

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*。

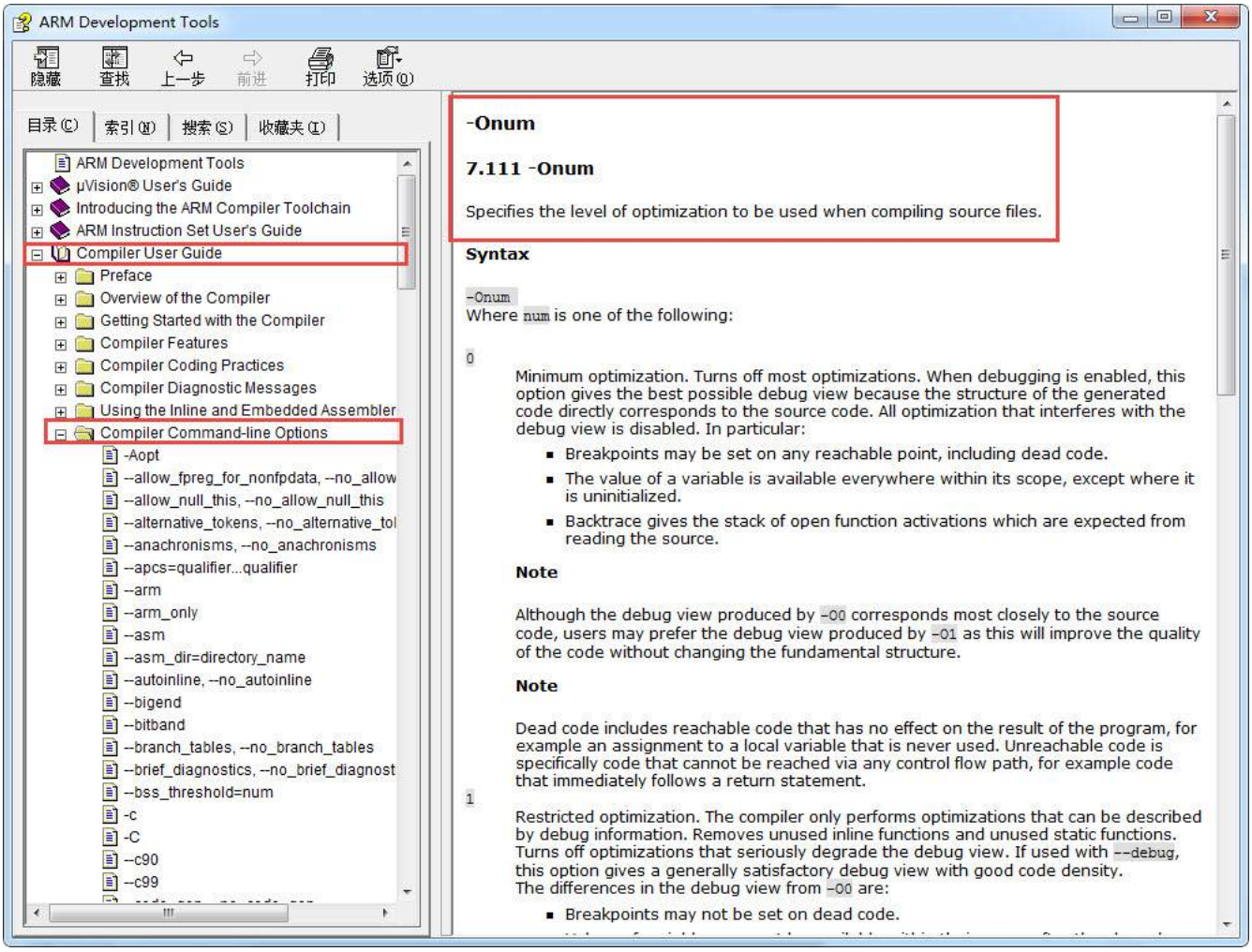


图 40-12 编译器选项说明

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。

47.3.2.2 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*。

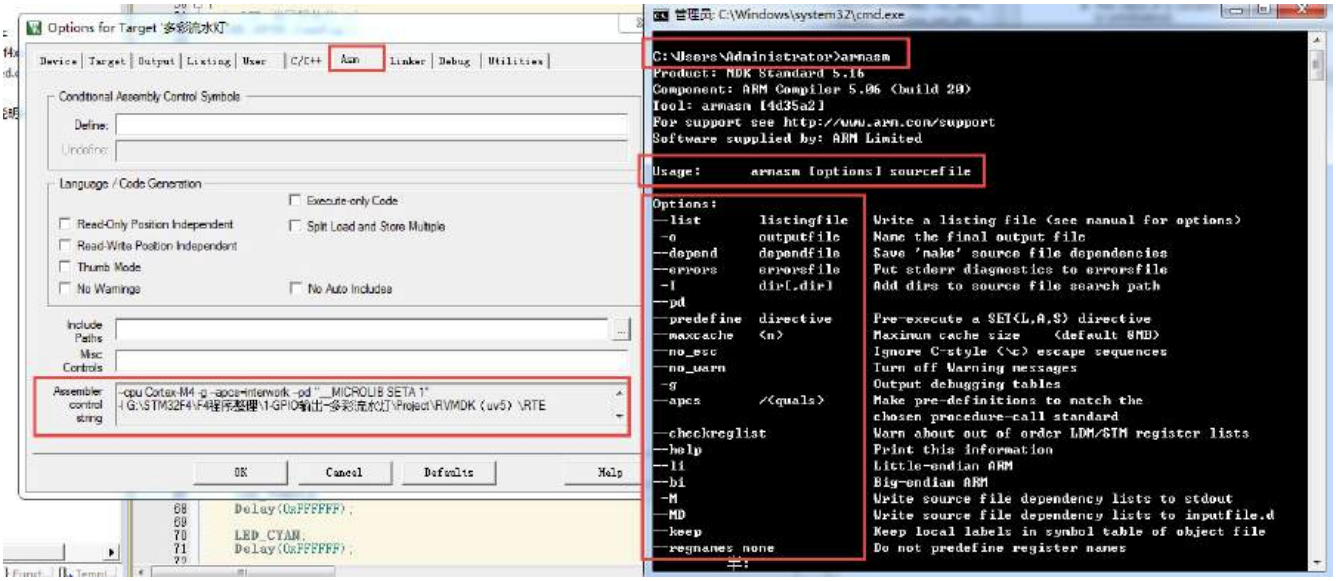


图 40-13 armasm 与 MDK 的编译选项

47.3.2.3 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*。

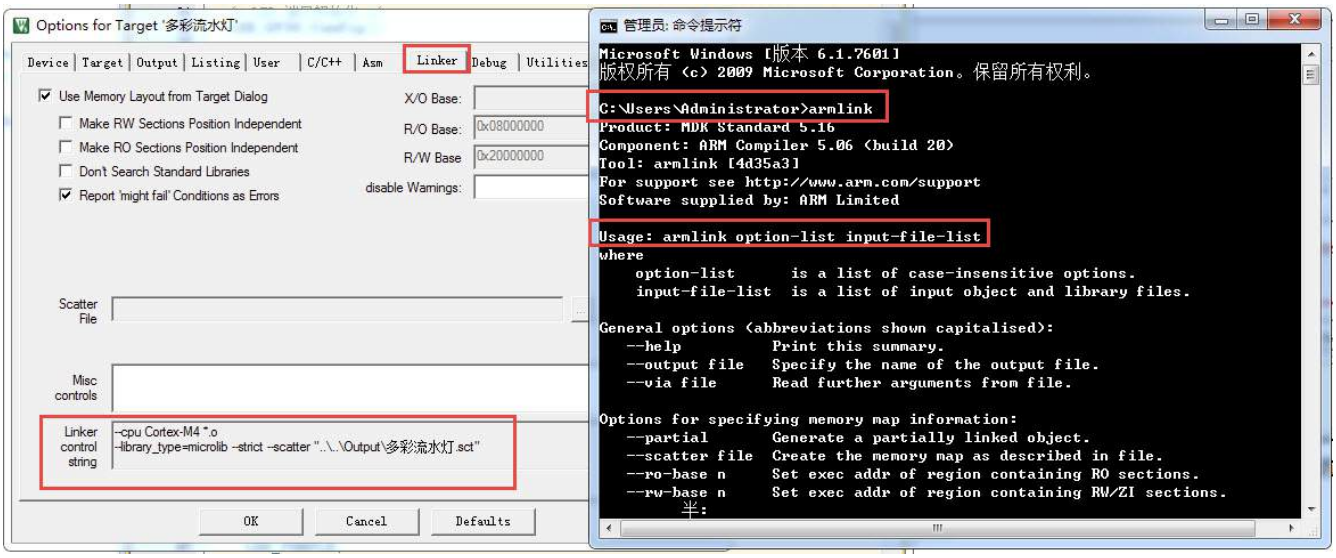


图 40-14 armlink 与 MDK 的配置选项

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。

47.3.3 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*。

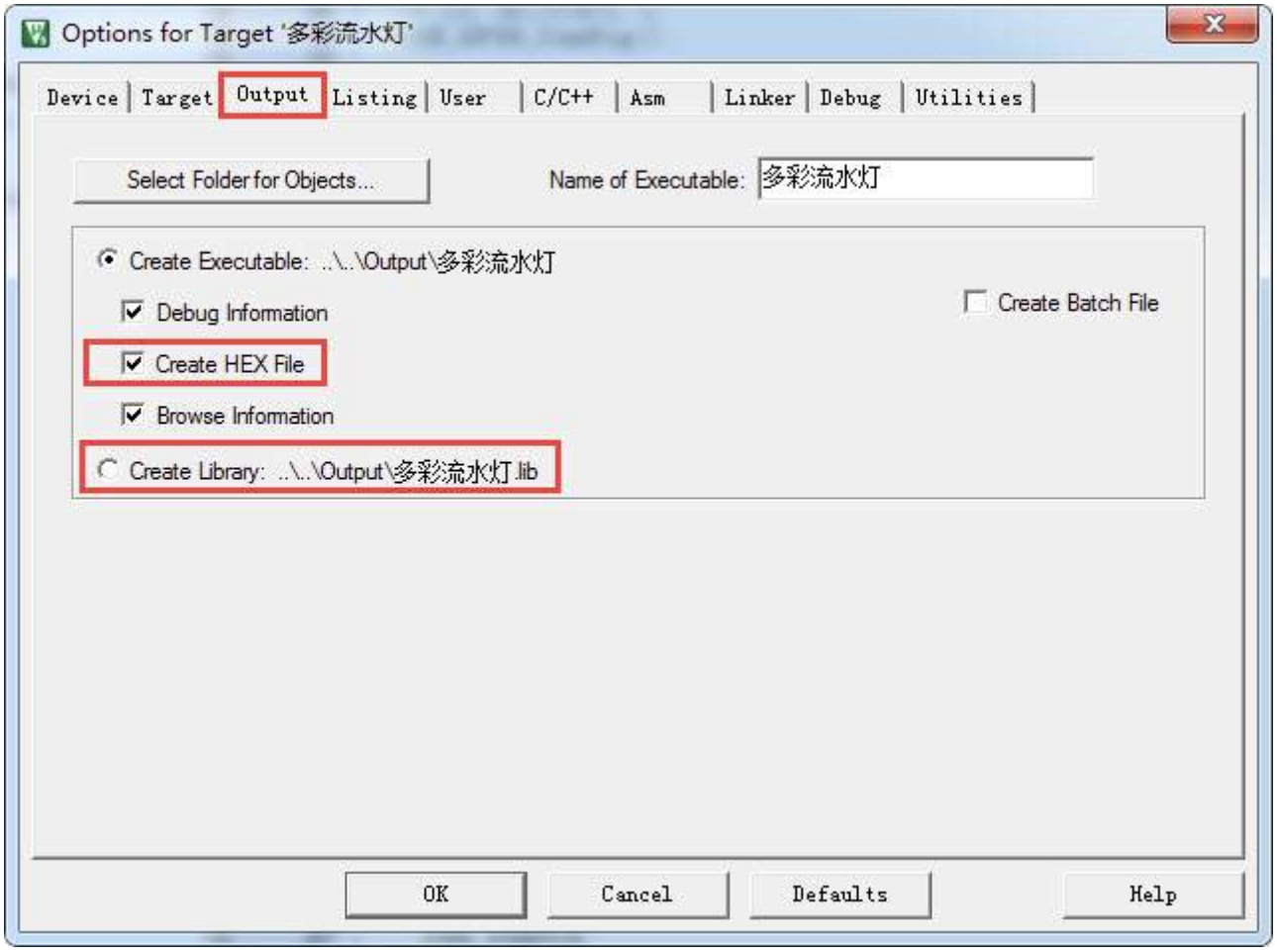


图 40-15 MDK 中，控制 fromelf 生成 hex 及控制 armar 生成 lib 的配置

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*。

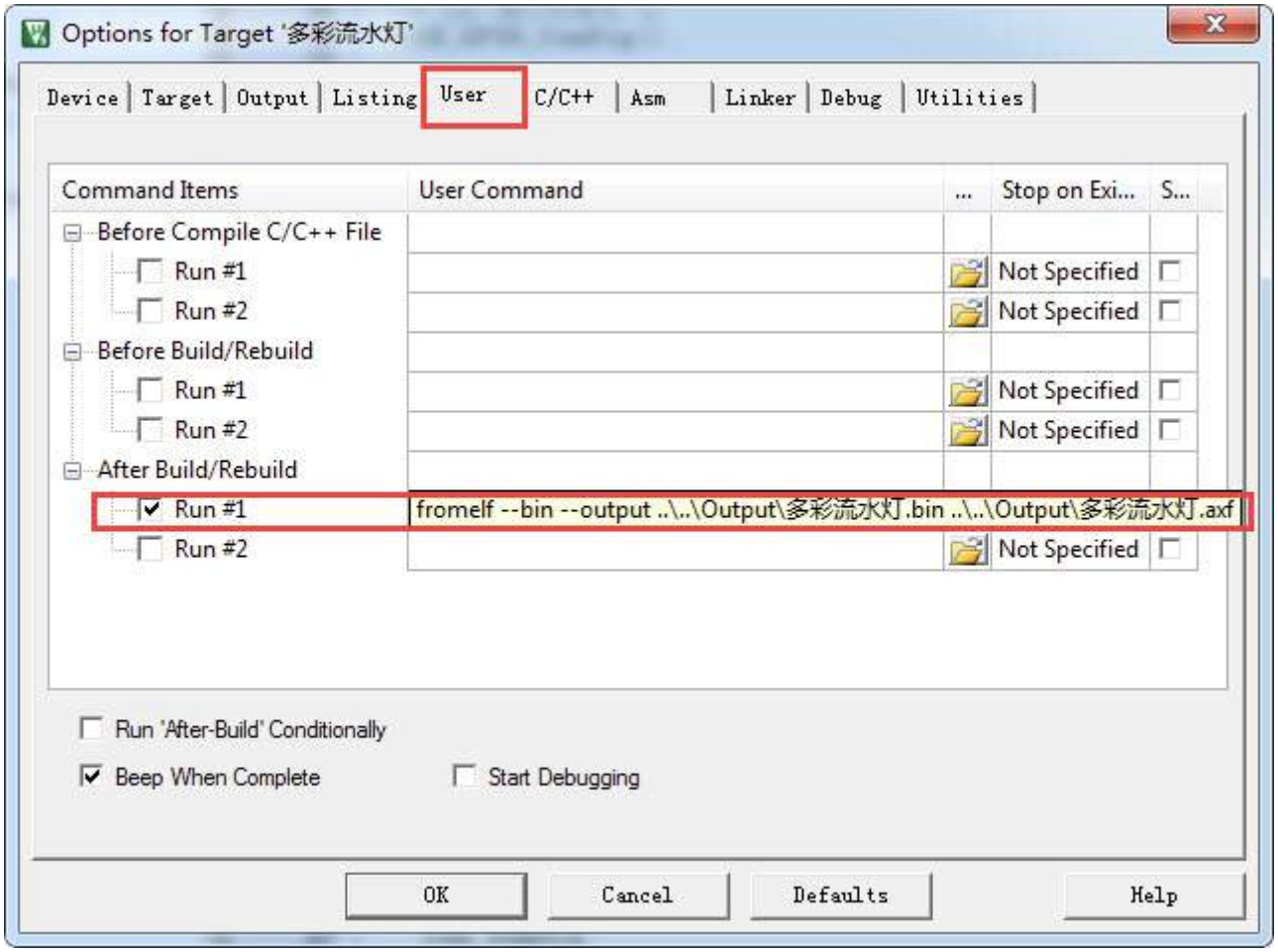


图 40-16 在 MDK 中添加指令

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。

47.4 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

表 1 – 续上页

*.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)
*.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。

47.4.1 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

47.4.1.1 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*。

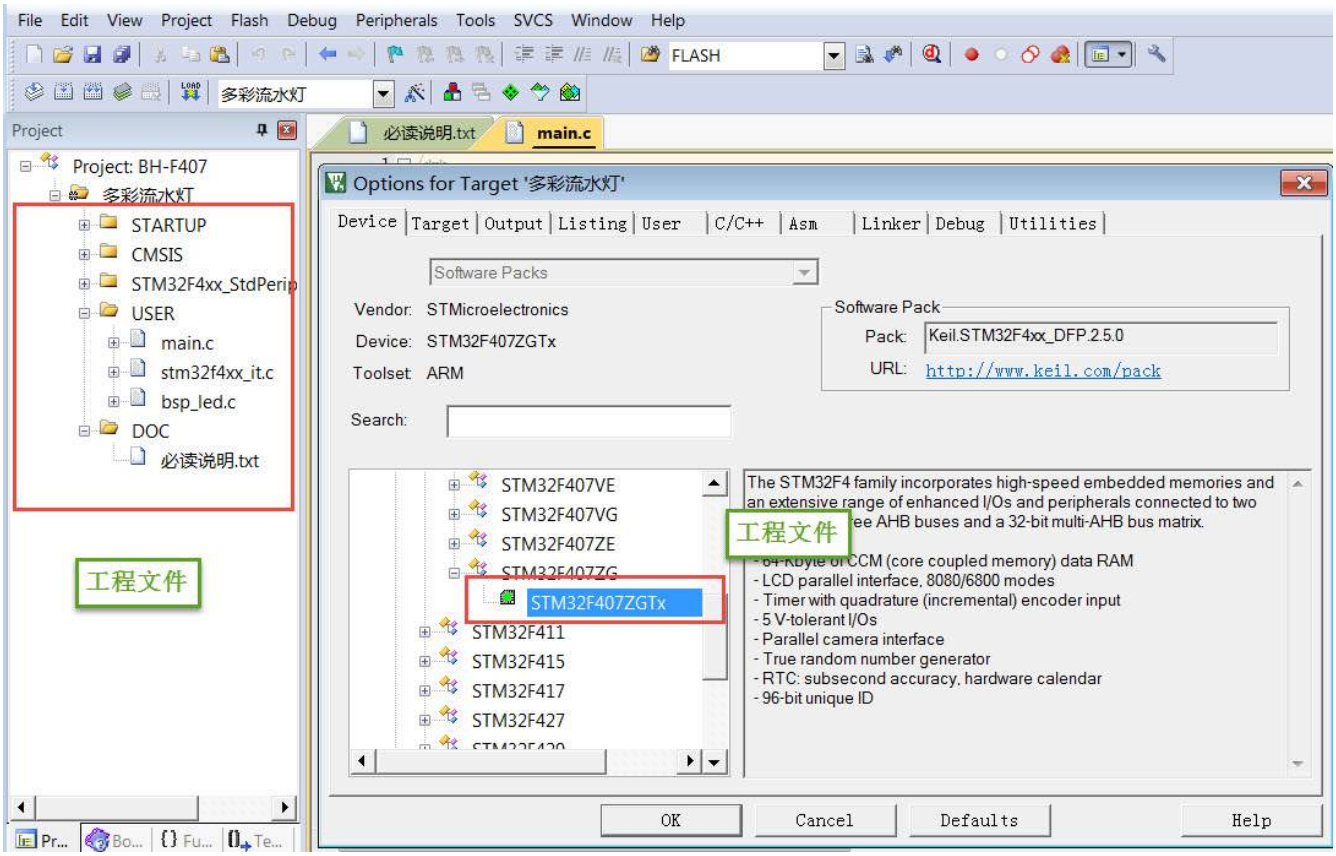


图 40-18 工程包含的文件、芯片类型等内容

47.4.1.2 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

47.4.1.3 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。

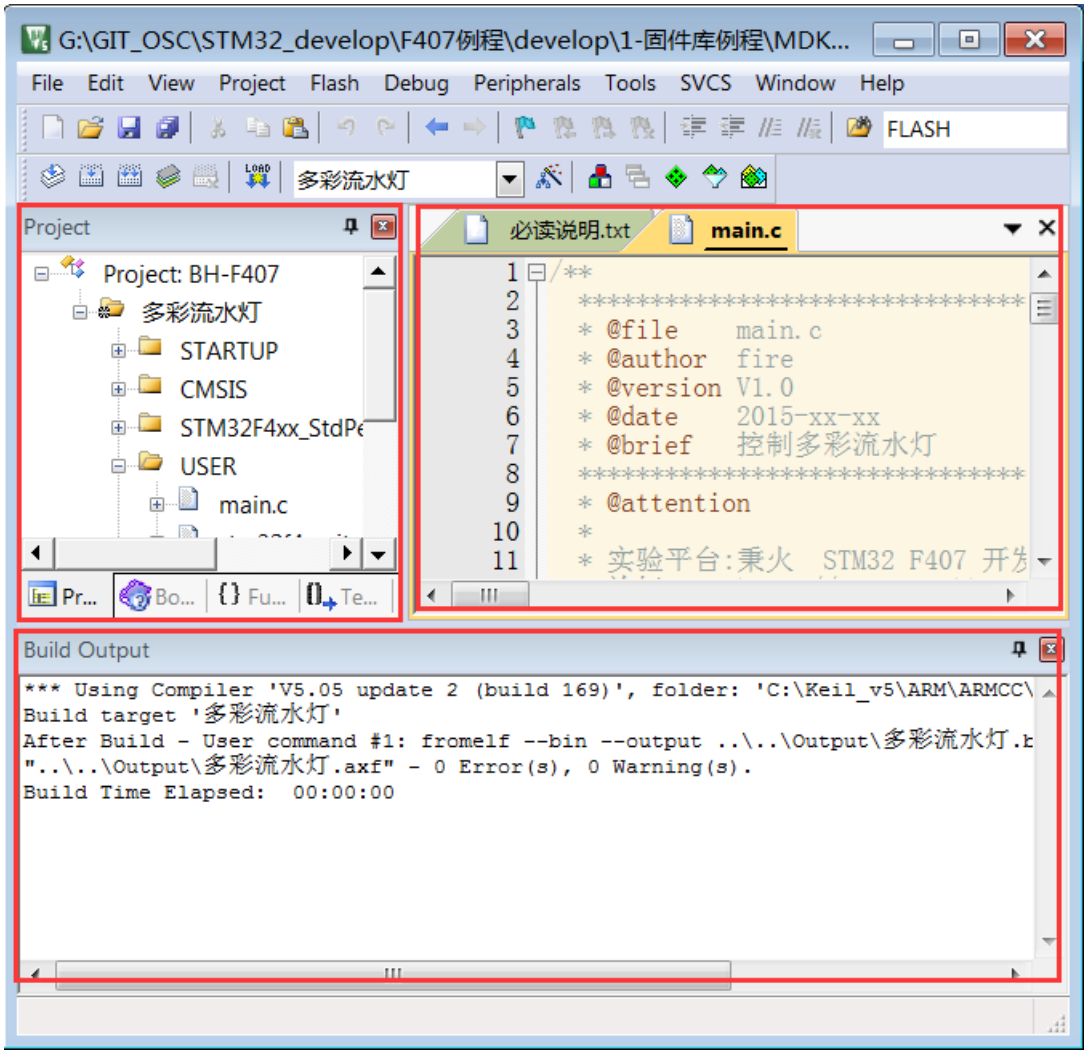


图 40-20 记录 MDK 工作环境中各个窗口的大小

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。

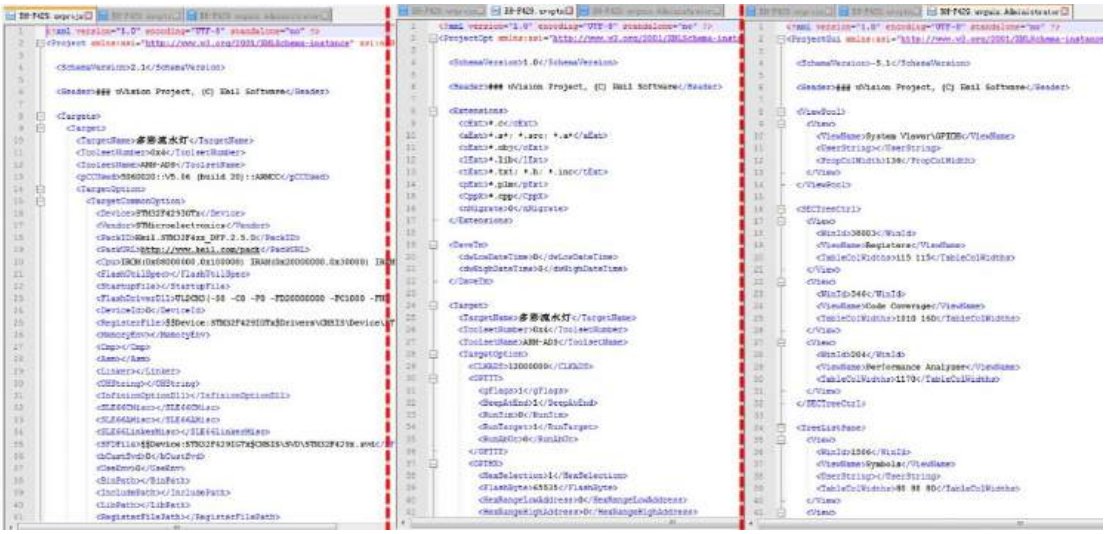


图 40-21 使用记事本打开 uvprojx、uvoptx 及 uvguix 文件可看到 XML 格式的记录

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)

47.4.2 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。

47.4.3 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*。

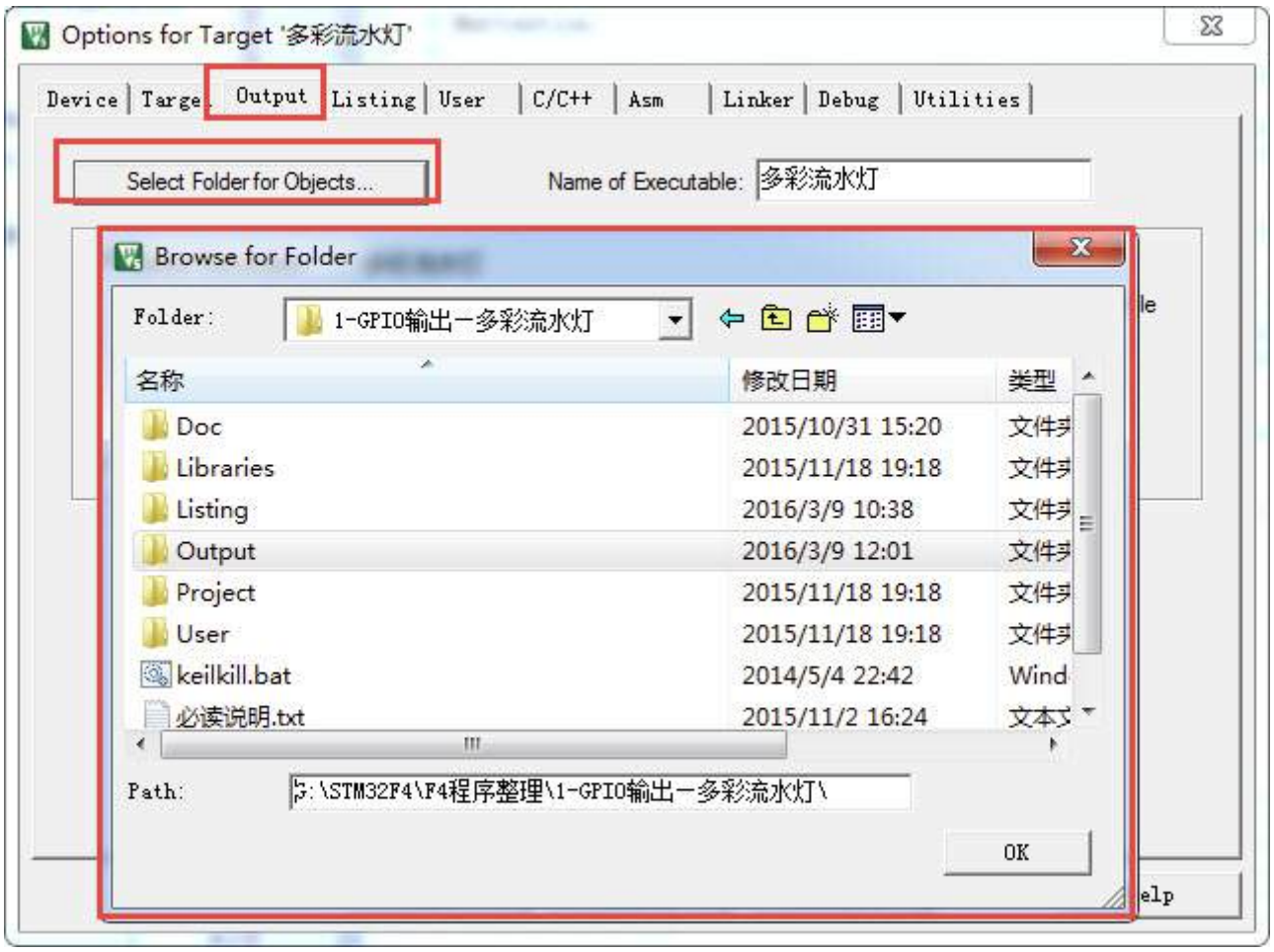


图 40-22 设置 Output 输出路径

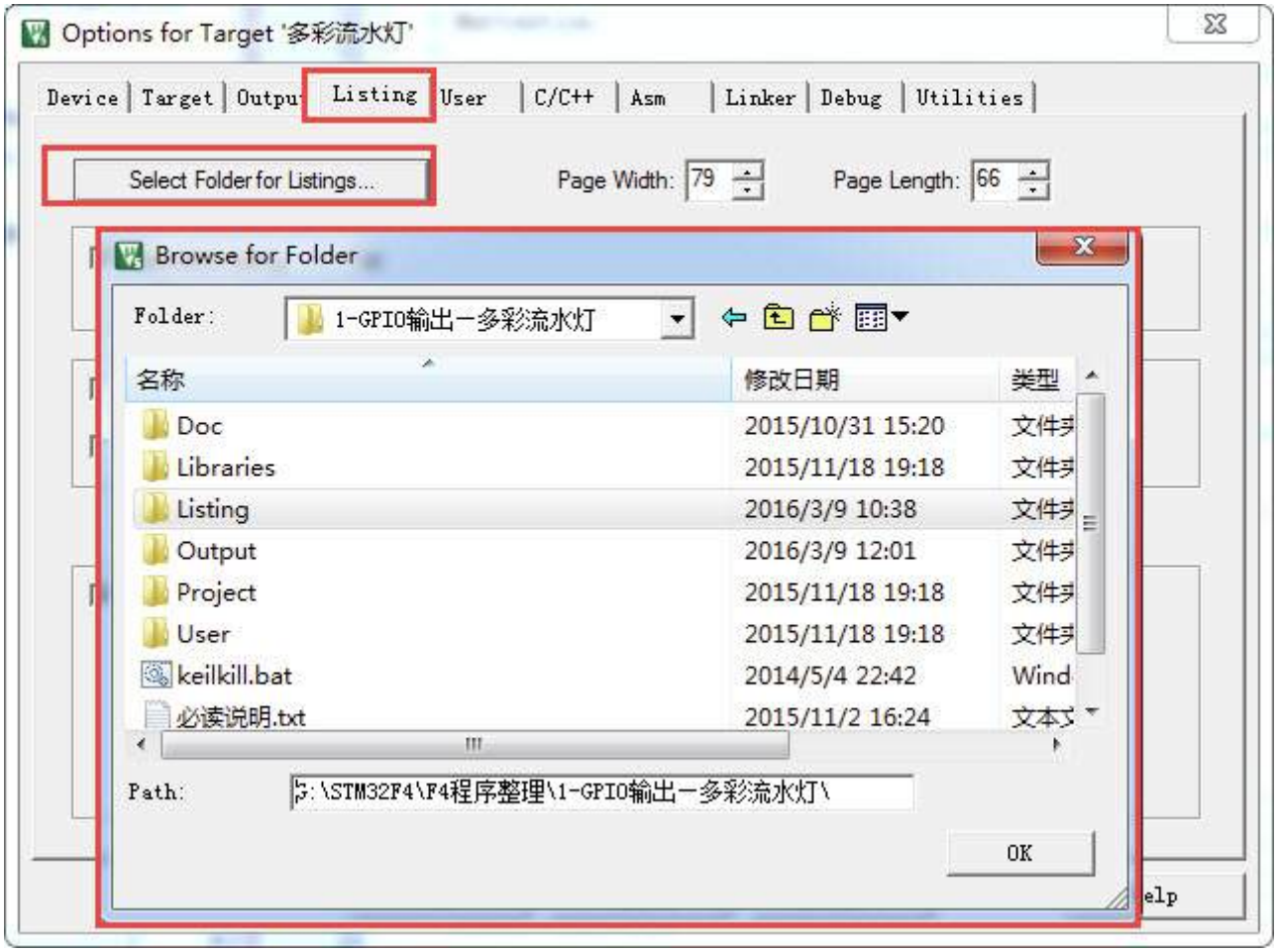


图 40-23 设置 Listing 输出路径

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。

47.4.3.1 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*。

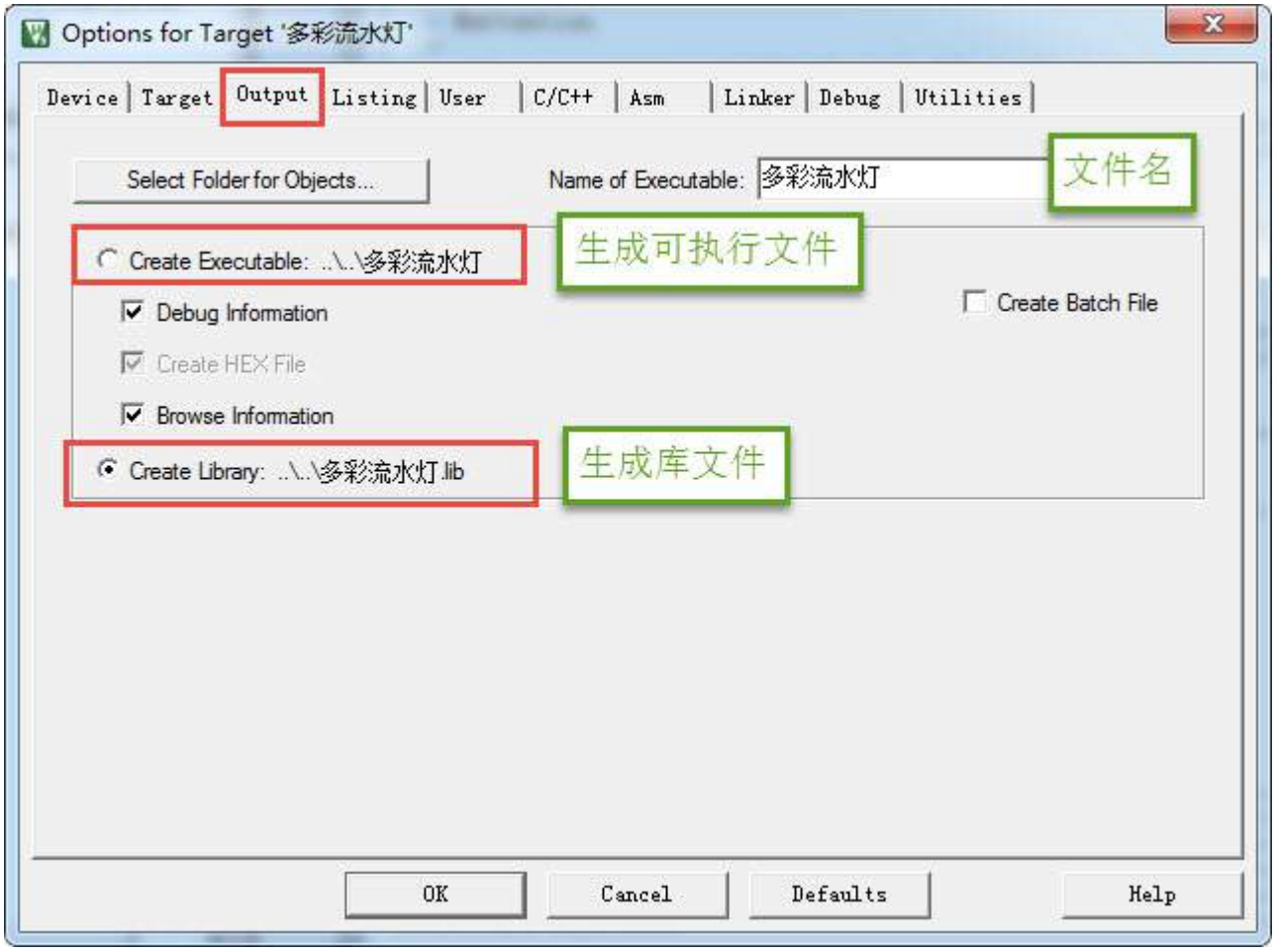


图 40-25 生成库文件或可执行文件

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。

47.4.3.2 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-F407_多彩流水灯.dep

```
1  Dependencies for Project 'BH-F407', Target '多彩流水灯': (DO NOT MODIFY !)
2  F (..\..\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm\startup_stm32f40xx.
3
4  -I G:\GIT_OSC\STM32_develop\F407例程\develop\1-固件库例程\MDK编译过程及文件全解\MDK文
5
6  -I C:\Keil_v5\ARM\PACK\Keil\STM32F4xx_DFP\2.5.0\Drivers\CMSIS\Device\ST\STM32F4xx\Inc
7
8  -I C:\Keil_v5\ARM\CMSIS\Include
9
10 --pd "__UVISION_VERSION SETA 515" --pd "STM32F407xx SETA 1"
11
12 --list ..\..\listing\startup_stm32f40xx.lst --xref -o ..\..\output\startup_stm32f40xx
13 F (..\..\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\system_stm32f4xx.c)(0x5
```

图 40-26 工程的 dep 文件内容

bsp_led.d | BH-F407_多彩流水灯.dep

```
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\Device\ST\STM32F4xx\Include\stm32f4xx.
4  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\Include\core_cm4.h
5  ..\..\output\bsp_led.o: C:\Keil_v5\ARM\ARMCC\Bin\..\include\stdint.h
6  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\Include\core_cmInstr.h
7  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\Include\core_cmFunc.h
8  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\Include\core_cmSimd.h
9  ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\Device\ST\STM32F4xx\Include\system_str
10 ..\..\output\bsp_led.o: ..\..\User\stm32f4xx_conf.h
11 ..\..\output\bsp_led.o: ..\..\Libraries\STM32F4xx_StdPeriph_Driver\inc\stm32f4xx_ad
12 ..\..\output\bsp_led.o: ..\..\Libraries\CMSIS\Device\ST\STM32F4xx\Include\stm32f4xx.
13 ..\..\output\bsp_led.o: ..\..\Libraries\STM32F4xx_StdPeriph_Driver\inc\stm32f4xx_cr
```

图 40-27 bsp_led.d 文件的内容

47.4.3.3 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。

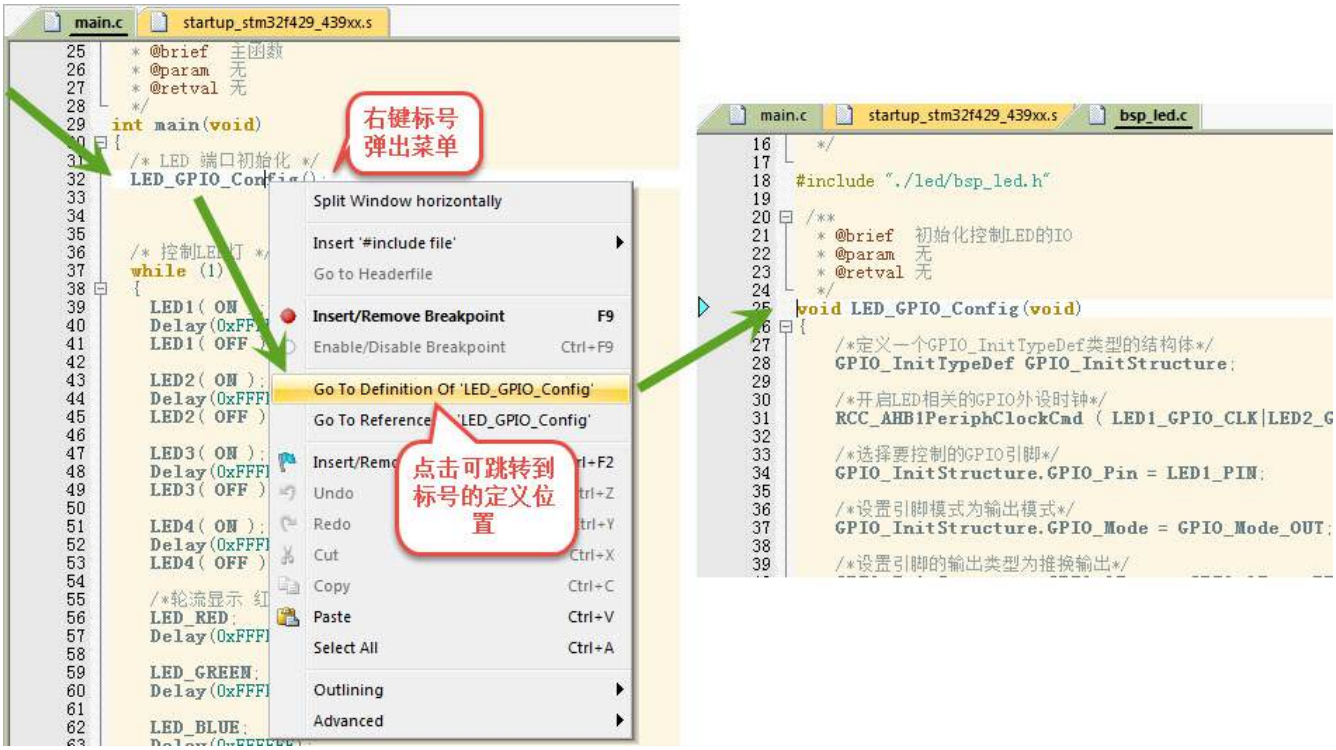


图 40-28 浏览信息

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。

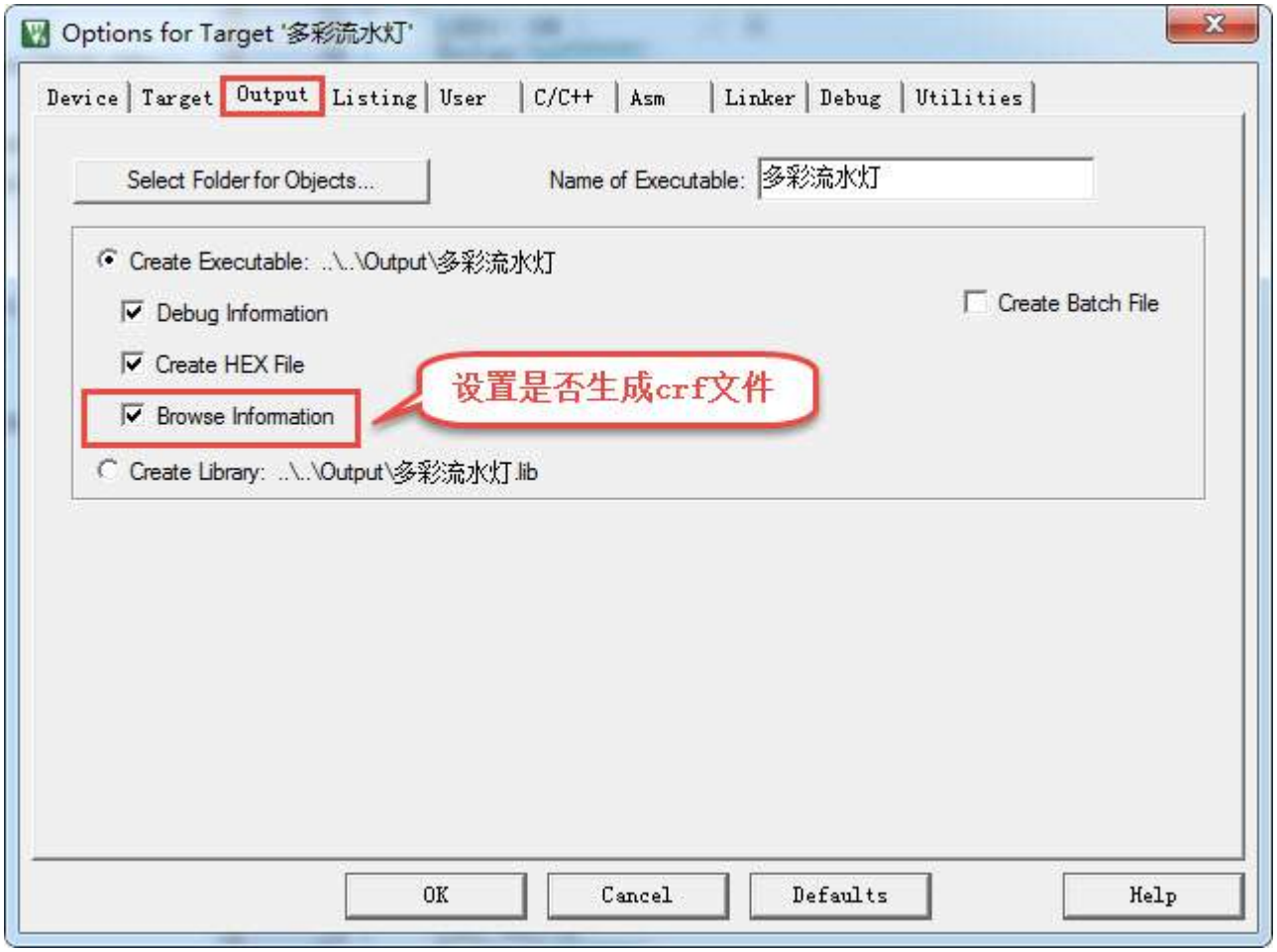


图 40-29 在 Options forTarget 中设置是否生成浏览信息

*.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。

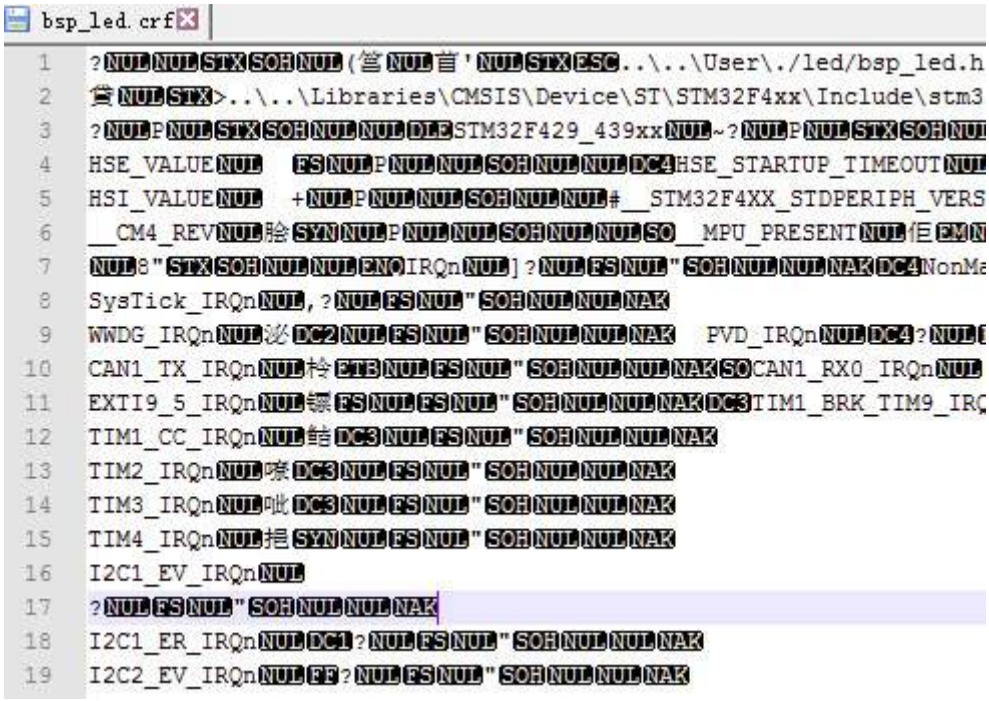


图 40-30 crf 文件内容

47.4.3.4 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。

47.4.3.4.1 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。

47.4.3.4.2 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*。

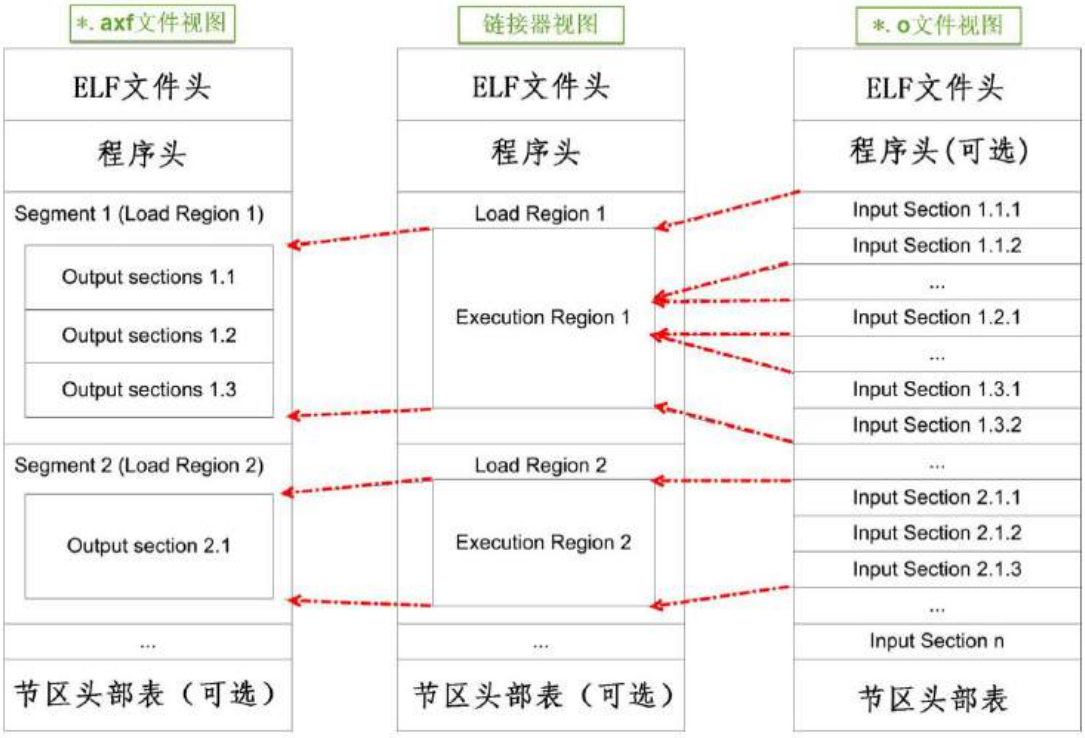


图 40-31*.axf 文件与 *.o 文件的关系

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“stm32f4xx_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 stm32f4xx_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。

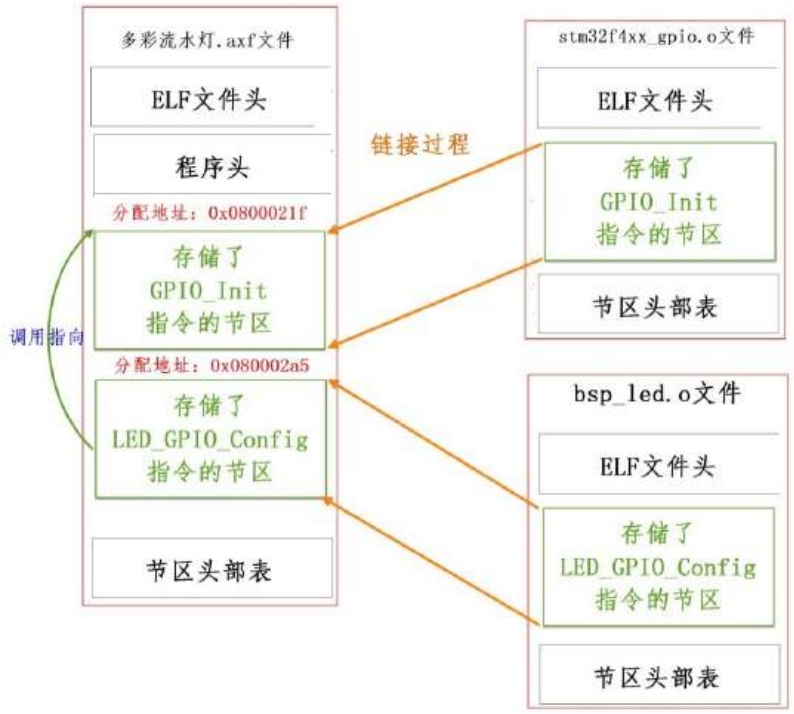


图 40-32 具体的链接过程

47.4.3.4.3 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。

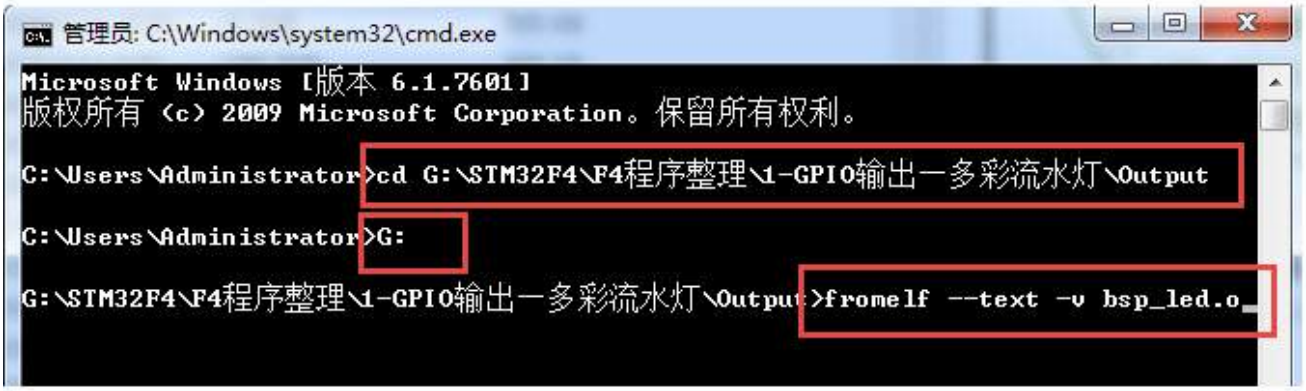


图 40-33 使用 fromelf 查看 o 文件信息

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)      //可重定位文件类型
   Machine: EM_ARM (ARM)

   Entry offset (in SHF_ENTRYSECT section): 0x00000000
   Flags: None (0x05000000)

   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: 0 bytes (0x0)  //程序头大小
   Section header entry size: 40 bytes (0x28)

   Program header entries: 0
```

(下页继续)

(续上页)

```
   Section header entries: 231

   Program header offset: 0 (0x00000000)    //程序头在文件中的位置 (没有程序头)
   Section header offset: 478264 (0x00074c38)         //节区头在文件中的位置

   Section header string table index: 228

   ============================================================================
```

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。

47.4.3.4.4 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:
.\多彩流水灯.axf                          //多彩流水灯.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)
```

(下页继续)

(续上页)

```
   Operating System ABI: none
   ABI Version: 0
   File Type: ET_EXEC (Executable) (2)          //可执行文件类型
   Machine: EM_ARM (ARM)

   Image Entry point: 0x08000189
   Flags: EF_ARM_HASENTRY (0x05000002)

   ARM ELF revision: 5 (ABI version 2)

   Conforms to Soft float procedure-call standard

   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)
   Section header entry size: 40 bytes (0x28)

   Program header entries: 1
   Section header entries: 15

   Program header offset: 334956 (0x00051c6c)       //程序头在文件中的位置
   Section header offset: 334988 (0x00051c8c)       //节区头在文件中的位置

   Section header string table index: 14

   ================================================================

   ** Program header #0
```

(下页继续)

(续上页)

```
  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   : 1316 bytes (0x524)           //程序在文件中占据的大小
  Size in memory: 2340 bytes (0x924)            //若程序加载到内存，占据的内存空间
  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。

47.4.3.4.5 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 #4

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。
```

(下页继续)

(续上页)

```
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  : 68 (0x44)                 //在文件中的偏移
Size         : 88 bytes (0x58)                //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*。

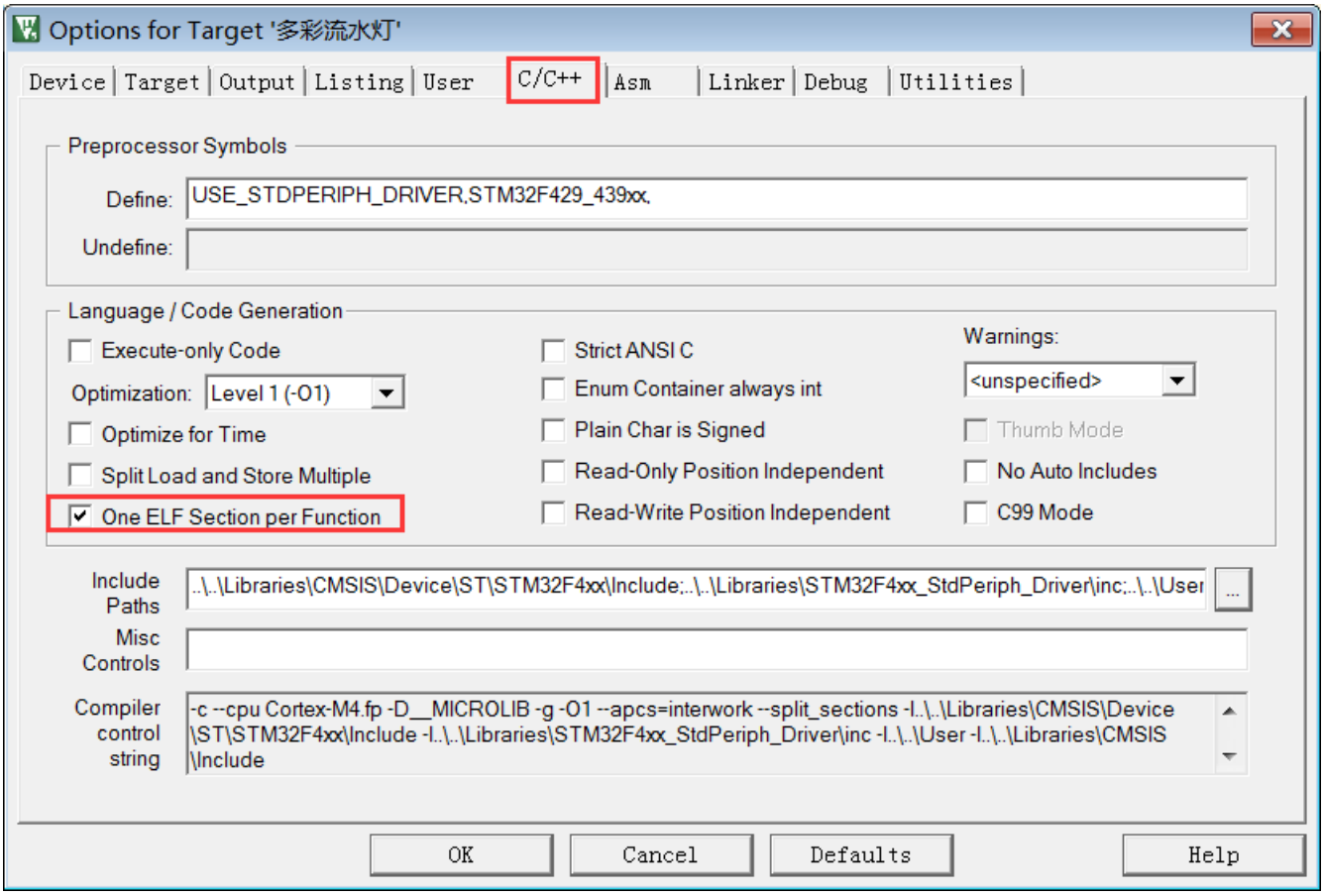


图 40-34 勾选 One ELF Section per Function

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         : 1316 bytes (0x524)          //大小
   Link         : SHN_UNDEF
   Info         : 0
   Alignment    : 4
   Entry Size   : 0

====================================
** Section #2

   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_NOBITS (0x00000008)

   //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。
```

(下页继续)

(续上页)

```
   Flags        : SHF_ALLOC + SHF_WRITE (0x00000003)
   Addr         : 0x20000000            //地址
   File Offset  : 1368 (0x558)
   Size         : 1024 bytes (0x400)    //大小
   Link         : SHN_UNDEF
   Info         : 0
   Alignment    : 8
   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。

47.4.3.4.6 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 #4 'i.LED_GPIO_Config' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
   Size   : 88 bytes (alignment 4)
   Address: 0x00000000
```

(下页继续)

(续上页)

```
    $t
    i.LED_GPIO_Config
    LED_GPIO_Config
    //  地址          内容       (ASCII 码)    内容对应的代码
    //                             (无意义)
       0x00000000:    e92d41fc    -..A    PUSH     {r2-r8,lr}
       0x00000004:    2101        .!      MOVS     r1,#1
       0x00000006:    2020                MOVS     r0,#0x20
0x00000008:    f7fffffe    ....    BL       RCC_AHB1PeriphClockCmd
       0x0000000c:    2440        @$      MOVS     r4,#0x40
       0x0000000e:    9400        ..      STR      r4,[sp,#0]
       0x00000010:    2001        .       MOVS     r0,#1
       0x00000012:    f88d0004    ....    STRB     r0,[sp,#4]
          0x00000016:    2100        .!      MOVS     r1,#0
          0x00000018:    f88d1006    ....    STRB     r1,[sp,#6]
          0x0000001c:    f88d0007    ....    STRB     r0,[sp,#7]
          0x00000022:    f88d0005    ....    STRB     r0,[sp,#5]
          0x00000024:    4f0b        .O      LDR      r7,[pc,#44] ; ;
          0x00000026:    4669        iF      MOV      r1,sp
          0x00000028:    4638        8F      MOV      r0,r7
          0x0000002a:    f7fffffe    ....    BL       GPIO_Init
          0x0000002e:    2680        .&      MOVS     r6,#0x80
          /*.... 以下省略 **/
```

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
      0x08000280:    e92d41fc    -..A    PUSH     {r2-r8,lr}
      0x08000284:    2101        .!      MOVS     r1,#1
      0x08000286:    2020                MOVS     r0,#0x20
0x08000288:    f000f82a    ..*.    BL       RCC_AHB1PeriphClockCmd ;␣
↪0x80002e0
      0x0800028c:    2440        @$      MOVS     r4,#0x40
      0x0800028e:    9400        ..      STR      r4,[sp,#0]
      0x08000290:    2001        .       MOVS     r0,#1
      0x08000292:    f88d0004    ....    STRB     r0,[sp,#4]
      0x08000296:    2100        .!      MOVS     r1,#0
      0x08000298:    f88d1006    ....    STRB     r1,[sp,#6]
      0x0800029c:    f88d0007    ....    STRB     r0,[sp,#7]
      0x080002a0:    f88d1005    ....    STRB     r1,[sp,#5]
0x080002a4:    4f0b        .O      LDR      r7,[pc,#44] ; [0x80002d4] =␣
↪0x40021400
      0x080002a6:    4669        iF      MOV      r1,sp
      0x080002a8:    4638        8F      MOV      r0,r7
      0x080002aa:    f7ffffa6    ....    BL       GPIO_Init ; 0x80001fa
      0x080002ae:    2680        .&      MOVS     r6,#0x80
      /*.... 以下省略 **/
```

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。

47.4.3.4.7 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)

```
  $t
  .text
  __scatterload
  __scatterload_rt2
        0x080001c0:    4c06        .L      LDR      r4,[pc,#24] ;␣
↪[0x80001dc] = 0x8000514
        0x080001c2:    4d07        .M      LDR      r5,[pc,#28] ;␣
↪[0x80001e0] = 0x8000524
        0x080001c4:    e006        ..      B        0x80001d4 ; __
↪scatterload + 20
        0x080001c6:    68e0        .h      LDR      r0,[r4,#0xc]
        0x080001c8:    f0400301    @...    ORR      r3,r0,#1
        0x080001cc:    e8940007    ....    LDM      r4,{r0-r2}
        0x080001d0:    4798        .G      BLX      r3
        0x080001d2:    3410        .4      ADDS     r4,r4,#0x10
        0x080001d4:    42ac        .B      CMP      r4,r5
        0x080001d6:    d3f6        ..      BCC      0x80001c6 ; __
↪scatterload + 6
```

(下页继续)

(续上页)

```
        0x080001d8:    f7ffffda    ....    BL        __main_after_
↪scatterload ; 0x8000190
  $d
        0x080001dc:    08000514    ....    DCD    134219028
        0x080001e0:    08000524    $...    DCD    134219044
```

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）

```
__main
_main_stk
   0x08000188:    f8dfd00c    ....    LDR       sp,__lit__00000000 ;␣
↪[0x8000198] = 0x20000400
.ARM.Collect$$$$00000004
_main_scatterload
   0x0800018c:    f000f818    ....    BL        __scatterload ; 0x80001c0
```

47.4.3.5 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。

47.4.3.5.1 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

47.4.3.5.2 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。

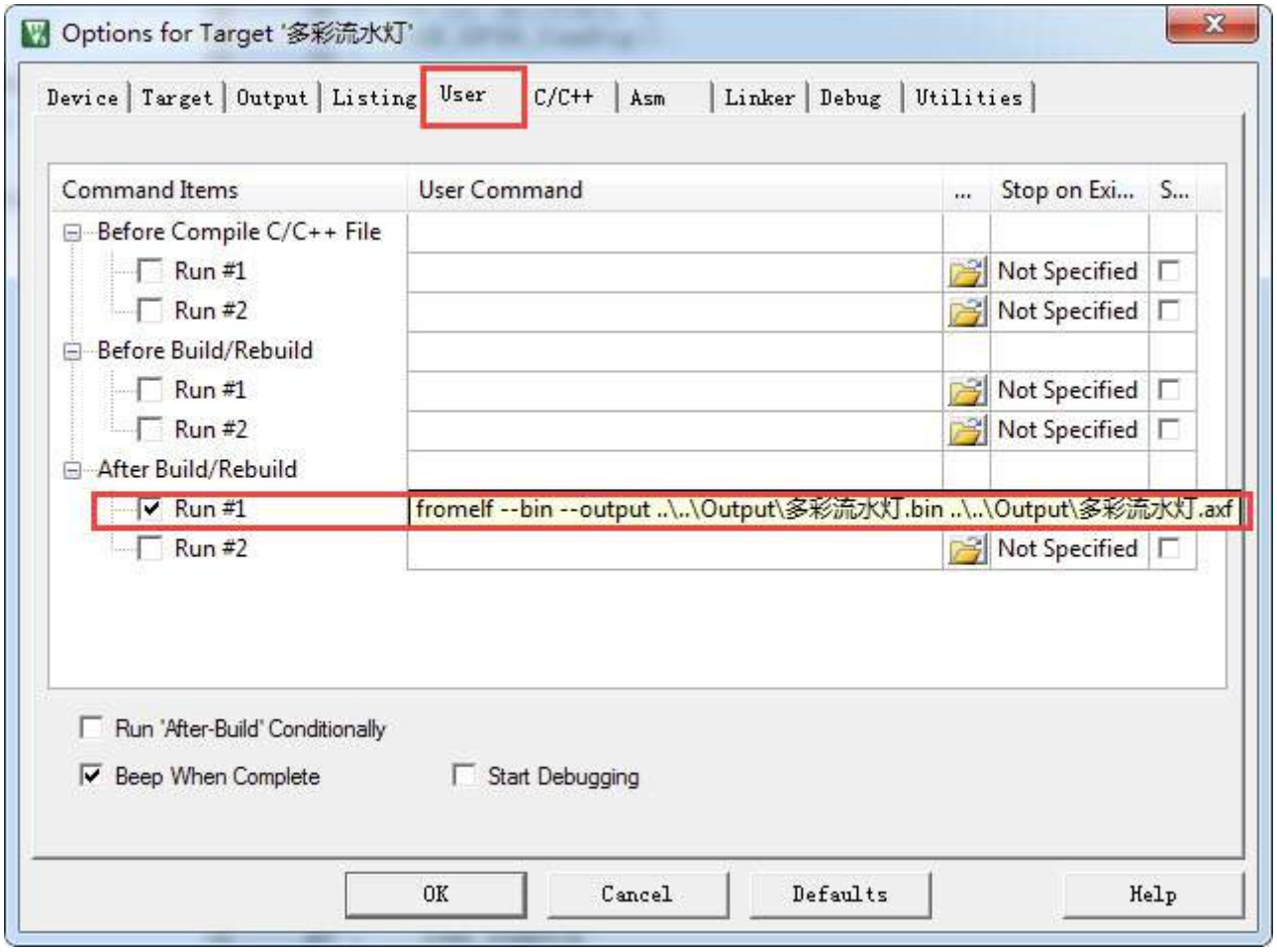


图 40-36 使用 fromelf 指令生成 bin 文件

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*。

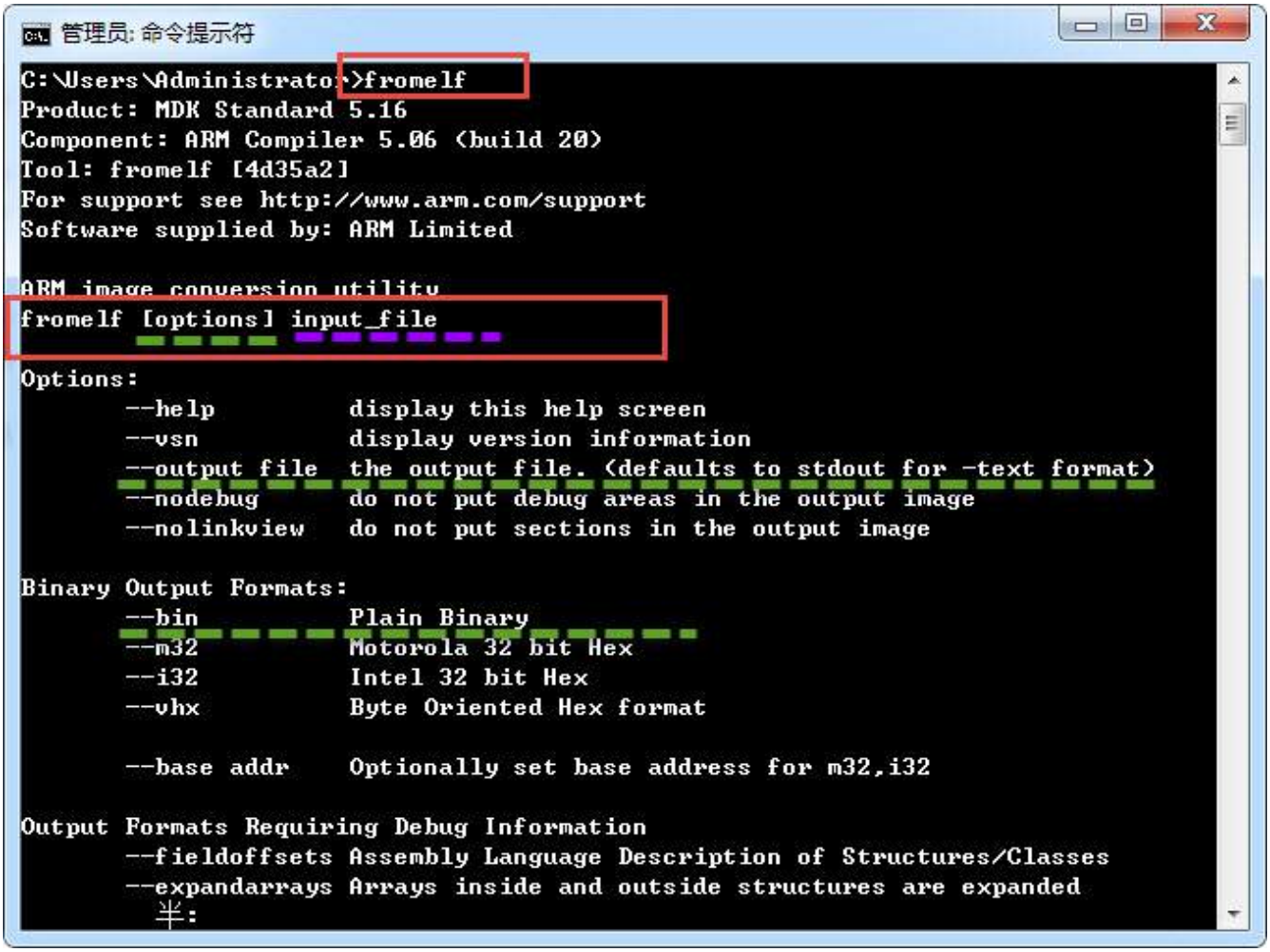


图 40-37 fromelf 的帮助

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*。

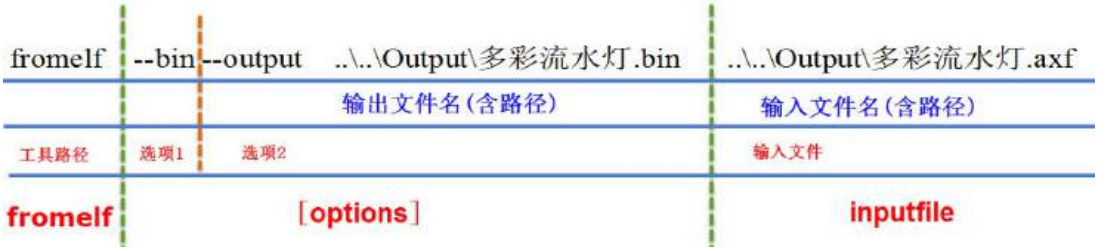


图 40-38 fromelf 命令格式分解

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 'V5.06 (build 20)', folder: 'D:\work\keil5\ARM\ARMCC\Bin'
Build target '多彩流水灯'
linking...
Program Size: Code=1012 RO-data=444 RW-data=0 ZI-data=1024
FromELF: creating hex file...
After Build - User command #1: fromelf --bin --output ..\..\Output\多彩流水灯.bin ..\..\Output\多彩流水灯.axf
"..\..\Output\多彩流水灯.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

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。

47.4.3.5.3 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
:10000000000400209D010008DB0200087F020008B8
:10001000D9020008E50100080D04000800000000F6
:1000200000000000000000000000000000FD020008C9
:10003000E701000800000000DD020008B903000825
:10004000B7010008B7010008B7010008B7010008B0
```

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) B8: 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。

47.4.3.5.4 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*。

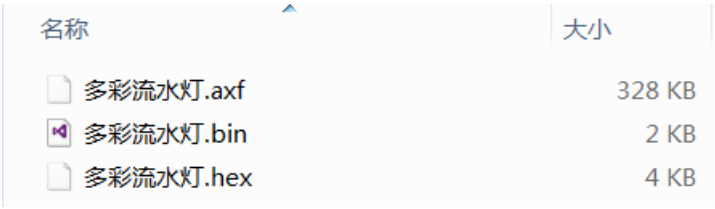


图 40-40 同一个工程的 bin、bex 及 axf 文件大小

实际上，这个工程要烧写到 FLASH 的内容总大小为 1316 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*。

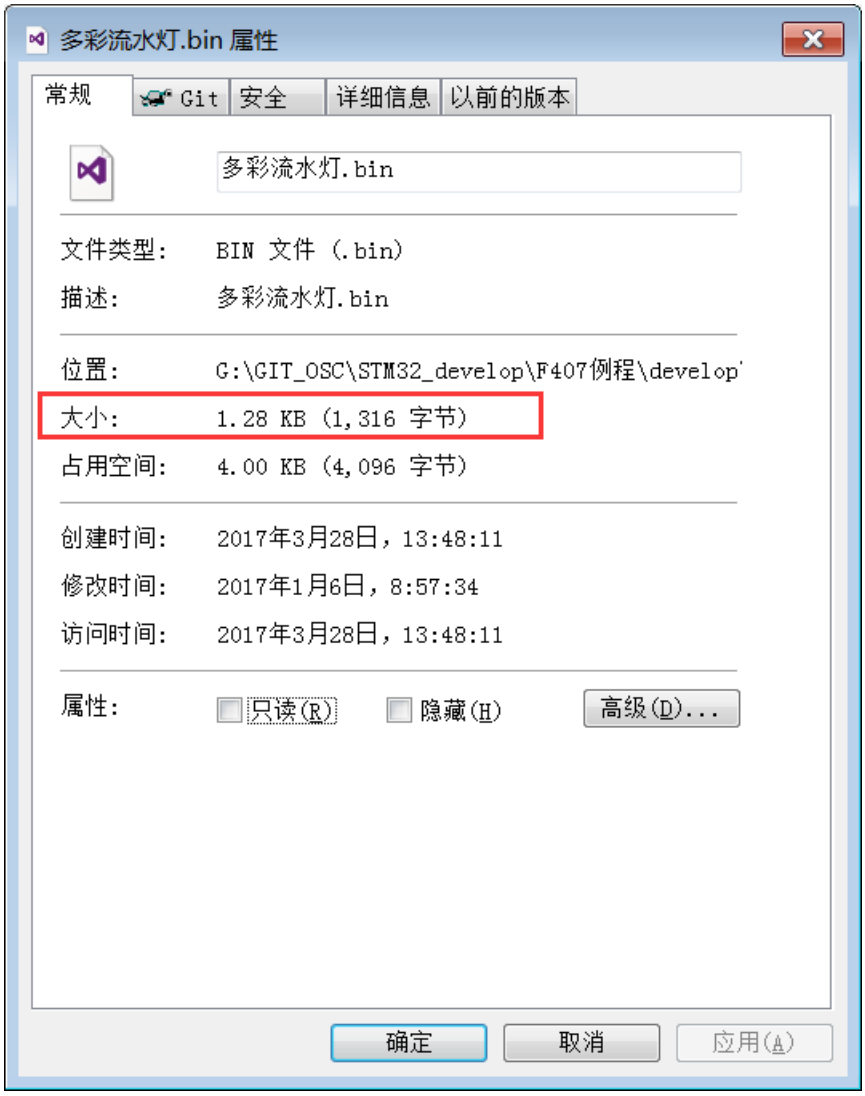


图 40-41 bin 文件大小

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。

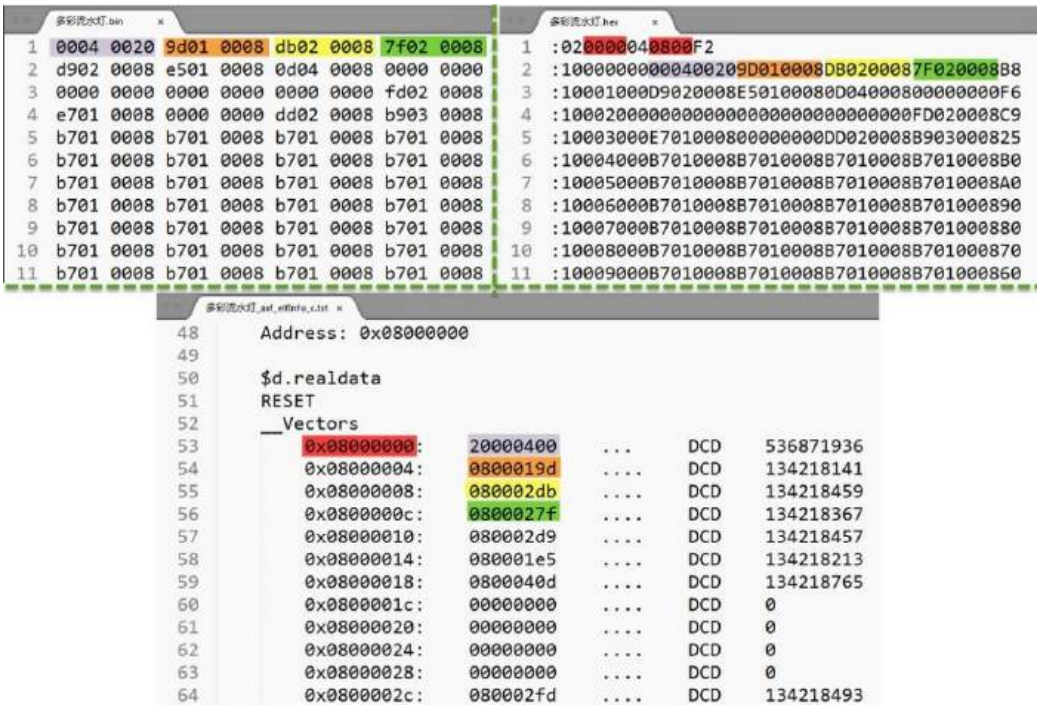


图 40-42 同一个工程的 bin、hex 及 axf 文件对代码的记录

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。

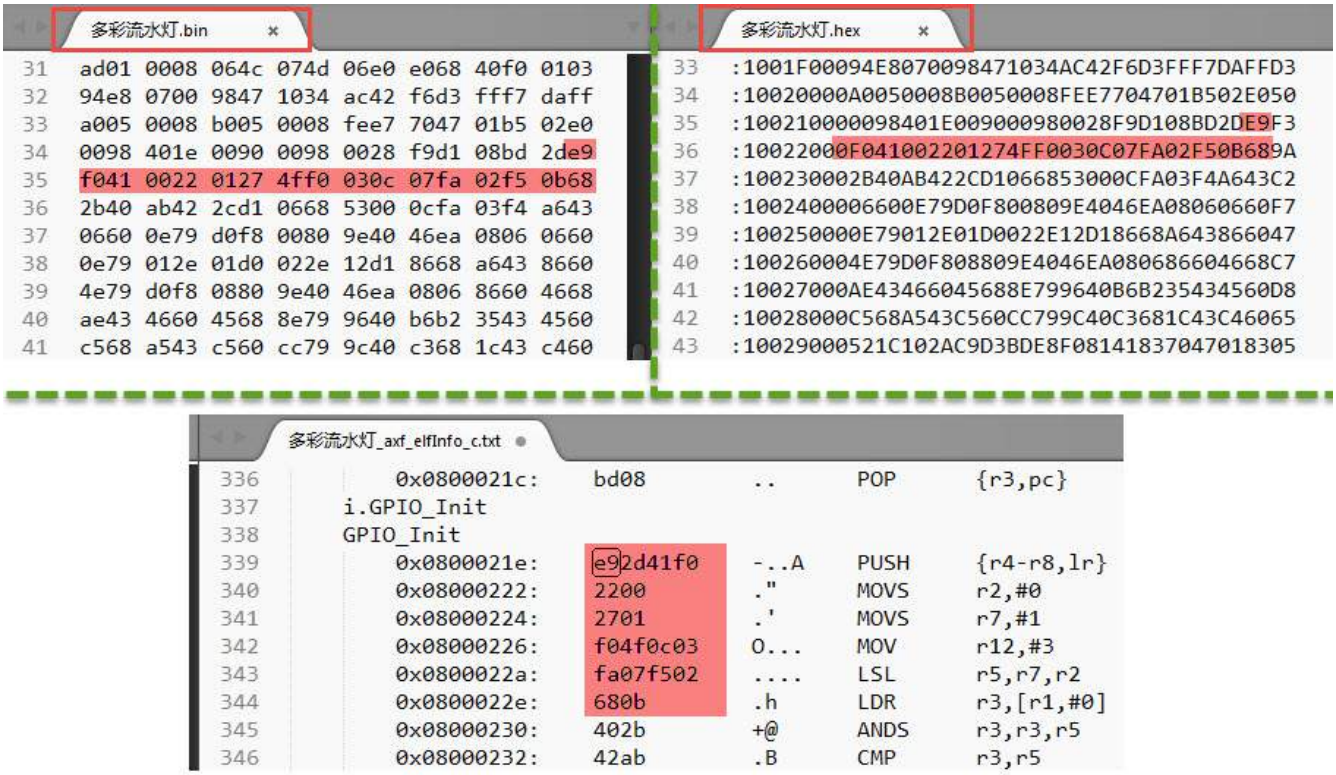


图 40-43 GPIO_Init 函数的代码数据在三个文件中的表示

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。

47.4.3.6 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\多彩流水灯.axf

```
#<CALLGRAPH># ARM Linker, 5050169: Last Updated: Mon Mar 27 16:30:14 2017

Maximum Stack Usage = 56 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。

47.4.4 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。

47.4.4.1 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：

47.4.4.1.1 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_stm32f40xx.o(RESET) refers to startup_stm32f40xx.o(STACK) for __
↪initial_sp
   startup_stm32f40xx.o(RESET) refers to startup_stm32f40xx.o(.text) for␣
↪Reset_Handler
   startup_stm32f40xx.o(RESET) refers to stm32f4xx_it.o(i.NMI_Handler) for␣
↪NMI_Handler
   startup_stm32f40xx.o(RESET) refers to stm32f4xx_it.o(i.HardFault_
↪Handler) for HardFault_Handler
   /**... 以下部分省略 ****/
```

(下页继续)

(续上页)

```
  main.o(i.main) refers to bsp_led.o(i.LED_GPIO_Config) for LED_GPIO_Config
  main.o(i.main) refers to stm32f4xx_gpio.o(i.GPIO_ResetBits) for GPIO_
↪ResetBits
  main.o(i.main) refers to main.o(i.Delay) for Delay
  main.o(i.main) refers to stm32f4xx_gpio.o(i.GPIO_SetBits) for GPIO_
↪SetBits
  bsp_led.o(i.LED_GPIO_Config) refers to stm32f4xx_rcc.o(i.RCC_
↪AHB1PeriphClockCmd) for RCC_AHB1PeriphClockCmd
  bsp_led.o(i.LED_GPIO_Config) refers to stm32f4xx_gpio.o(i.GPIO_Init) for␣
↪GPIO_Init
  /**... 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

47.4.4.1.2 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)

```
==============================================================================

Removing Unused input sections from the image.

Removing startup_stm32f40xx.o(HEAP), (512 bytes).
```

(下页继续)

(续上页)

```
Removing system_stm32f4xx.o(.rev16_text), (4 bytes).
Removing system_stm32f4xx.o(.revsh_text), (4 bytes).
Removing system_stm32f4xx.o(.rrx_text), (6 bytes).
Removing system_stm32f4xx.o(i.SystemCoreClockUpdate), (136 bytes).
Removing system_stm32f4xx.o(.data), (20 bytes).
Removing misc.o(.rev16_text), (4 bytes).
Removing misc.o(.revsh_text), (4 bytes).
Removing misc.o(.rrx_text), (6 bytes).
Removing misc.o(i.NVIC_Init), (104 bytes).
Removing misc.o(i.NVIC_PriorityGroupConfig), (20 bytes).
Removing misc.o(i.NVIC_SetVectorTable), (20 bytes).
Removing misc.o(i.NVIC_SystemLPConfig), (28 bytes).
Removing misc.o(i.SysTick_CLKSourceConfig), (28 bytes).
Removing stm32f4xx_adc.o(.rev16_text), (4 bytes).
Removing stm32f4xx_adc.o(.revsh_text), (4 bytes).
Removing stm32f4xx_adc.o(.rrx_text), (6 bytes).
Removing stm32f4xx_adc.o(i.ADC_AnalogWatchdogCmd), (16 bytes).
Removing stm32f4xx_adc.o(i.ADC_AnalogWatchdogSingleChannelConfig), (12␣
↪bytes).
Removing stm32f4xx_adc.o(i.ADC_AnalogWatchdogThresholdsConfig), (6 bytes).
Removing stm32f4xx_adc.o(i.ADC_AutoInjectedConvCmd), (24 bytes).
Removing stm32f4xx_adc.o(i.ADC_ClearFlag), (6 bytes).
Removing stm32f4xx_adc.o(i.ADC_ClearITPendingBit), (8 bytes).
Removing stm32f4xx_adc.o(i.ADC_Cmd), (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_stm32f407xx.o 中的 HEAP(在启动文件中定义的用于动态分配的“堆”区) 以及 stm32f4xx_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 stm32f4xx_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。

47.4.4.1.3 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)

../clib/microlib/init/entry.s              0x00000000   Number          0 ␣
 ↪entry8a.o ABSOLUTE
../clib/microlib/init/entry.s              0x00000000   Number          0 ␣
 ↪entry7b.o ABSOLUTE
../clib/microlib/init/entry.s              0x00000000   Number          0 ␣
 ↪entry7a.o ABSOLUTE
/*... 省略部分 */
LED_GPIO_Config                            0x08000281   Thumb Code     84  bsp_
 ↪led.o(i.LED_GPIO_Config)
MemManage_Handler                          0x080002d9   Thumb Code      2 ␣
 ↪stm32f4xx_it.o(i.MemManage_Handler)
NMI_Handler                                0x080002db   Thumb Code      2 ␣
 ↪stm32f4xx_it.o(i.NMI_Handler)
PendSV_Handler                             0x080002dd   Thumb Code      2 ␣
 ↪stm32f4xx_it.o(i.PendSV_Handler)
```

(下页继续)

(续上页)

```
RCC_AHB1PeriphClockCmd                  0x080002e1   Thumb Code    22 ␣
 ↪stm32f4xx_rcc.o(i.RCC_AHB1PeriphClockCmd)
SVC_Handler                             0x080002fd   Thumb Code     2 ␣
 ↪stm32f4xx_it.o(i.SVC_Handler)
SysTick_Handler                         0x080003b9   Thumb Code     2 ␣
 ↪stm32f4xx_it.o(i.SysTick_Handler)
SystemInit                              0x080003bd   Thumb Code    74 ␣
 ↪system_stm32f4xx.o(i.SystemInit)
UsageFault_Handler                      0x08000419   Thumb Code     2 ␣
 ↪stm32f4xx_it.o(i.UsageFault_Handler)
__scatterload_copy                      0x0800041b   Thumb Code    14 ␣
 ↪handlers.o(i.__scatterload_copy)
__scatterload_null                      0x08000429   Thumb Code     2 ␣
 ↪handlers.o(i.__scatterload_null)
__scatterload_zeroinit                  0x0800042b   Thumb Code    14 ␣
 ↪handlers.o(i.__scatterload_zeroinit)
main                                    0x08000439   Thumb Code   226 ␣
 ↪main.o(i.main)
Region$$Table$$Base                     0x08000520   Number         0  anon
 ↪$$obj.o(Region$$Table)
Region$$Table$$Limit                    0x08000530   Number         0  anon
 ↪$$obj.o(Region$$Table)
__initial_sp                            0x20000400   Data           0 ␣
 ↪startup_stm32f40xx.o(STACK)

/*... 省略部分 */

==============================================================================
```

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 0x08000281 地址，它属于 Thumb Code 类型，大小为 84 字节，它所在的节区为 bsp_led.o 文件的 i.LED_GPIO_Config 节区。

47.4.4.1.4 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 : 0x08000189

Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00000530, Max: 0x00100000,␣
↪ABSOLUTE)

Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x00000530, Max:␣
↪0x00100000, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        ␣
↪Object

0x08000000   0x00000188   Data   RO            3    RESET               ␣
↪startup_stm32f40xx.o
/*.. 省略部分 */
0x080001e8   0x00000012   Code   RO         4765    i.Delay             ␣
↪main.o
0x080001fa   0x0000007c   Code   RO         1857    i.GPIO_Init         ␣
↪stm32f4xx_gpio.o
0x08000276   0x00000004   Code   RO         1864    i.GPIO_ResetBits    ␣
↪stm32f4xx_gpio.o
0x0800027a   0x00000004   Code   RO         1865    i.GPIO_SetBits      ␣
↪stm32f4xx_gpio.o
```

(下页继续)

(续上页)

```
0x0800027e   0x00000002   Code   RO          4800    i.HardFault_Handler ␣
↪stm32f4xx_it.o
0x08000280   0x00000058   Code   RO          4873    i.LED_GPIO_Config   bsp_
↪led.o
0x080002d8   0x00000002   Code   RO          4801    i.MemManage_Handler ␣
↪stm32f4xx_it.o
/*.. 省略部分 */
0x08000438   0x000000e8   Code   RO          4766    i.main              ␣
↪main.o
0x08000520   0x00000010   Data   RO          4913    Region$$Table       anon
↪$$obj.o

Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x00000400, Max:␣
↪0x00020000, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name       ␣
↪Object

0x20000000   0x00000400   Zero   RW            1    STACK               ␣
↪startup_stm32f40xx.o


==============================================================================
```

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 0x08000280 地址，大小为 0x00000058，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）。

47.4.4.1.5 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

88           4          0          0          0        702   bsp_led.o
250          6          0          0          0       1787    main.o
36           8        392          0       1024       1048   startup_
 ↪stm32f40xx.o
132          0          0          0          0       2940    stm32f4xx_gpio.
 ↪o
18           0          0          0          0       5198   stm32f4xx_it.o
28           6          0          0          0        785   stm32f4xx_rcc.o
276         34          0          0          0     250921    system_
 ↪stm32f4xx.o

------------------------------------------------------------------------------
834          58        408          0       1024     263381   Object Totals
0            0         16          0          0          0   (incl. Generated)
6            0          0          0          0          0   (incl. Padding)

/*... 省略部分 */
==============================================================================
```

(下页继续)

(续上页)

```
Code (inc. data)   RO Data    RW Data    ZI Data      Debug

920         74         408          0       1024     262453   Grand Totals
920         74         408          0       1024     262453   ELF Image␣
↪Totals
920         74         408          0          0          0   ROM Totals

================================================================================

Total RO  Size (Code + RO Data)                 1328 (   1.30kB)
Total RW  Size (RW Data + ZI Data)              1024 (   1.00kB)
Total ROM Size (Code + RO Data + RW Data)       1328 (   1.30kB)

================================================================================
```

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 920 字节，这部分包含了 74 字节的指令数据 (inc .data) 已算在内，另外 RO-data 占 408 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 1328 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 1328 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。

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 1328 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。

47.4.5 sct 分散加载文件的格式与应用

47.4.5.1 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。

47.4.5.2 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 0x00020000  {  ; 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*。

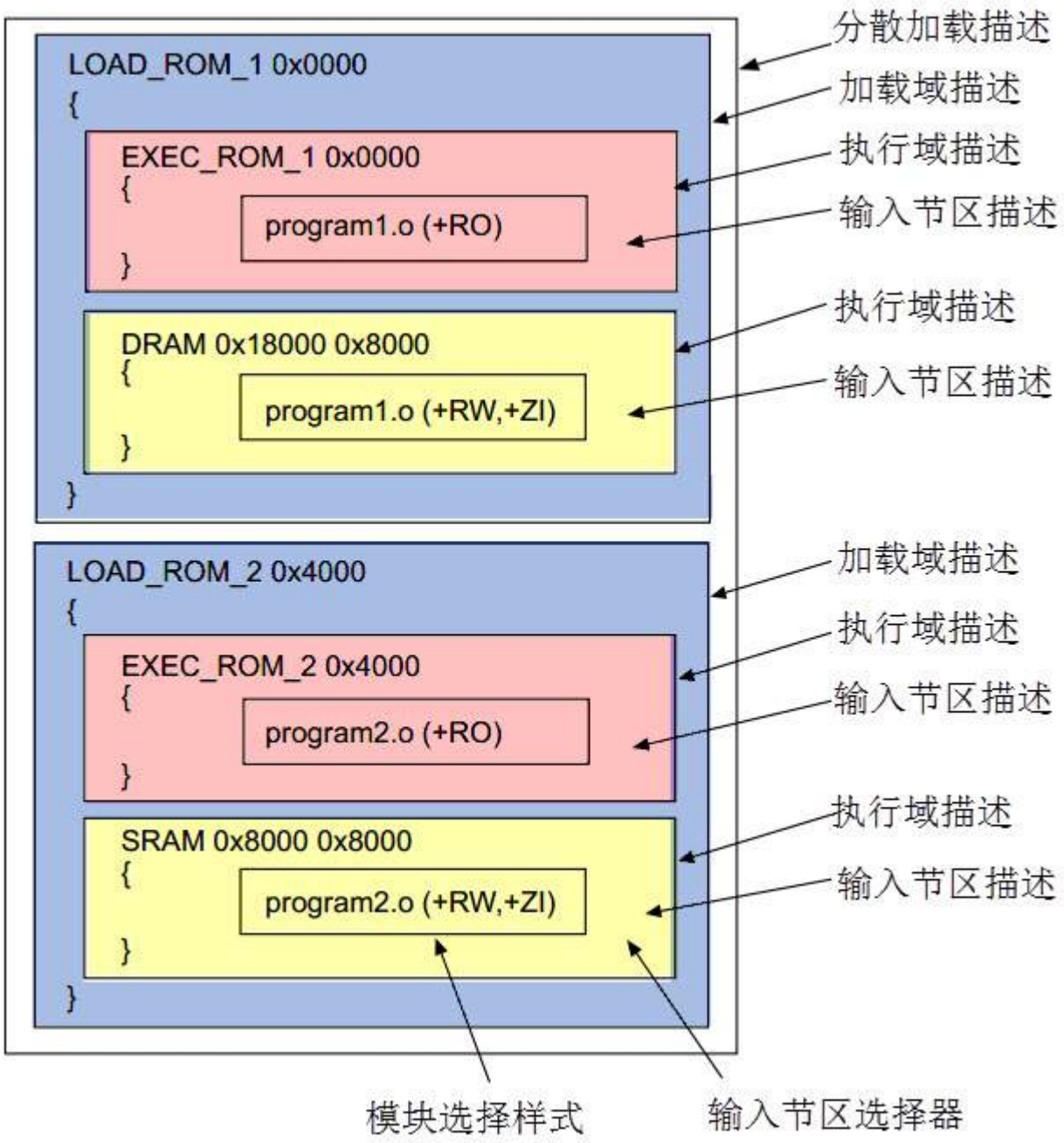


图 40-46 分散加载文件的整体结构

47.4.5.2.1 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 0x00100000，即 1MB，正是本型号 STM32 内部 FLASH 的空间大小。

47.4.5.2.2 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。

47.4.5.2.3 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 0x00100000；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 0x00020000 的空间 (128KB)，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。

47.4.5.3 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。

47.4.5.3.1 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*。

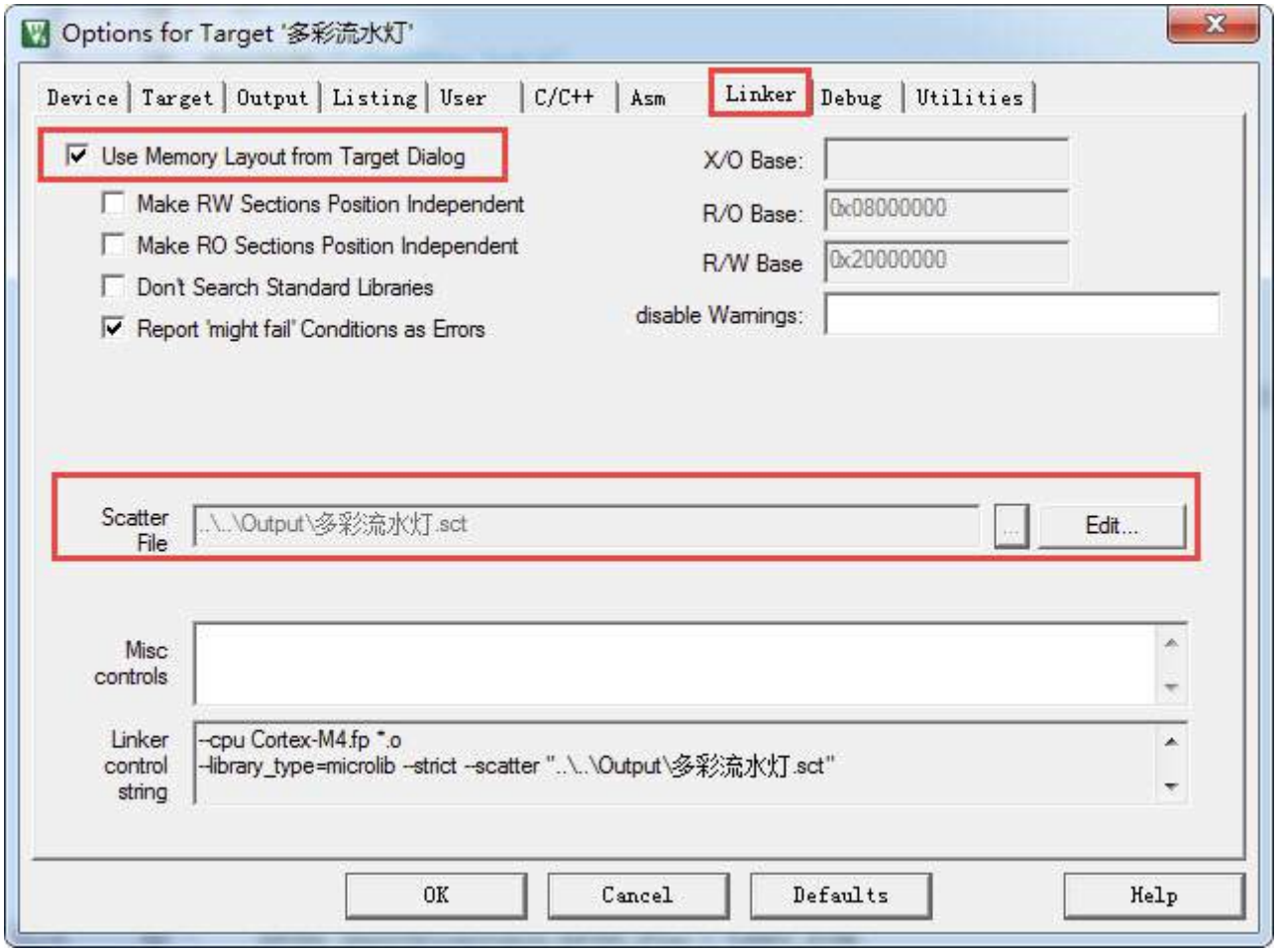


图 40-47 选择使用 MDK 生成的 sct 文件

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*。

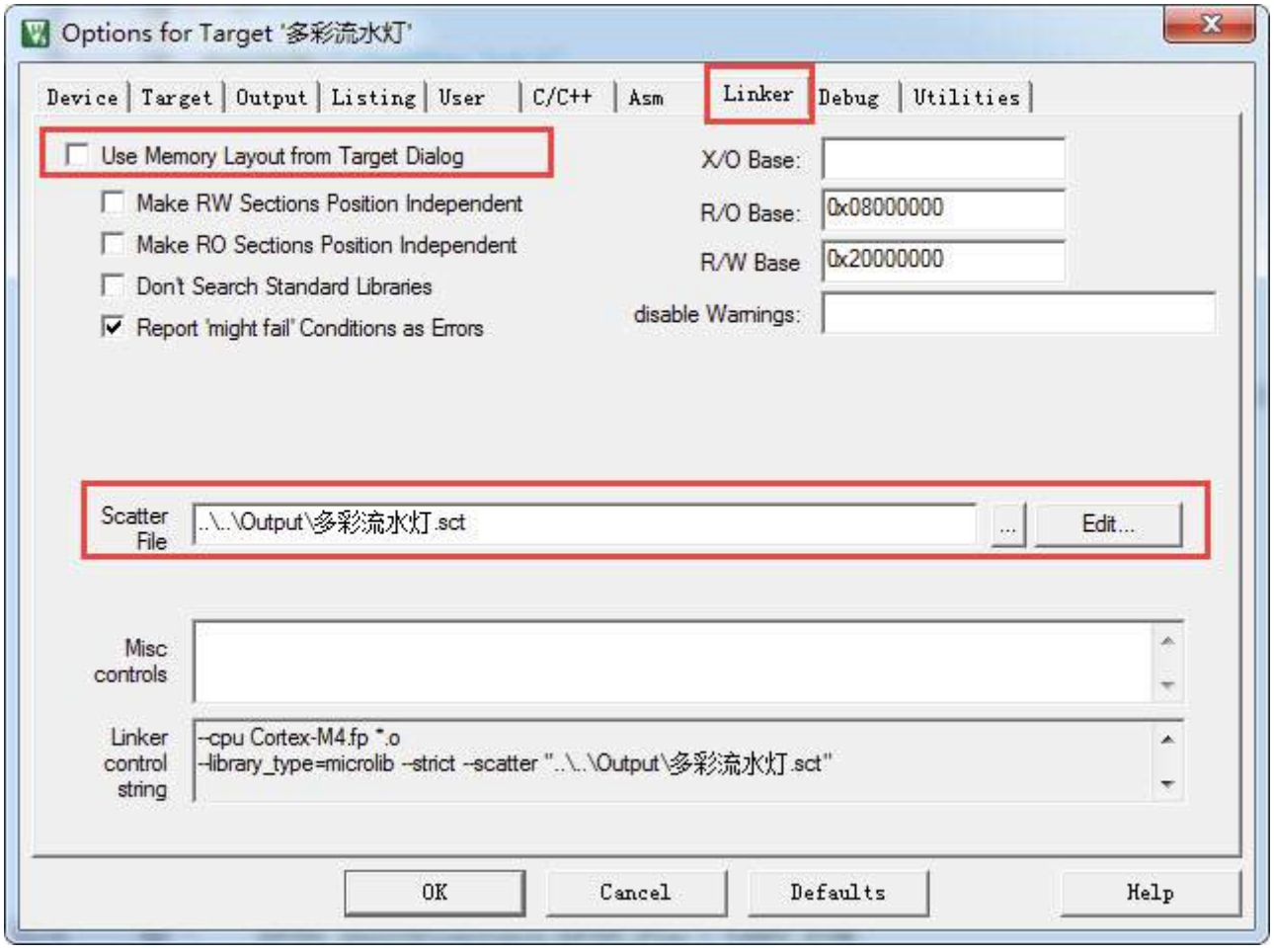


图 40-48 使用指定的 sct 文件构建工程

47.4.5.3.2 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*。

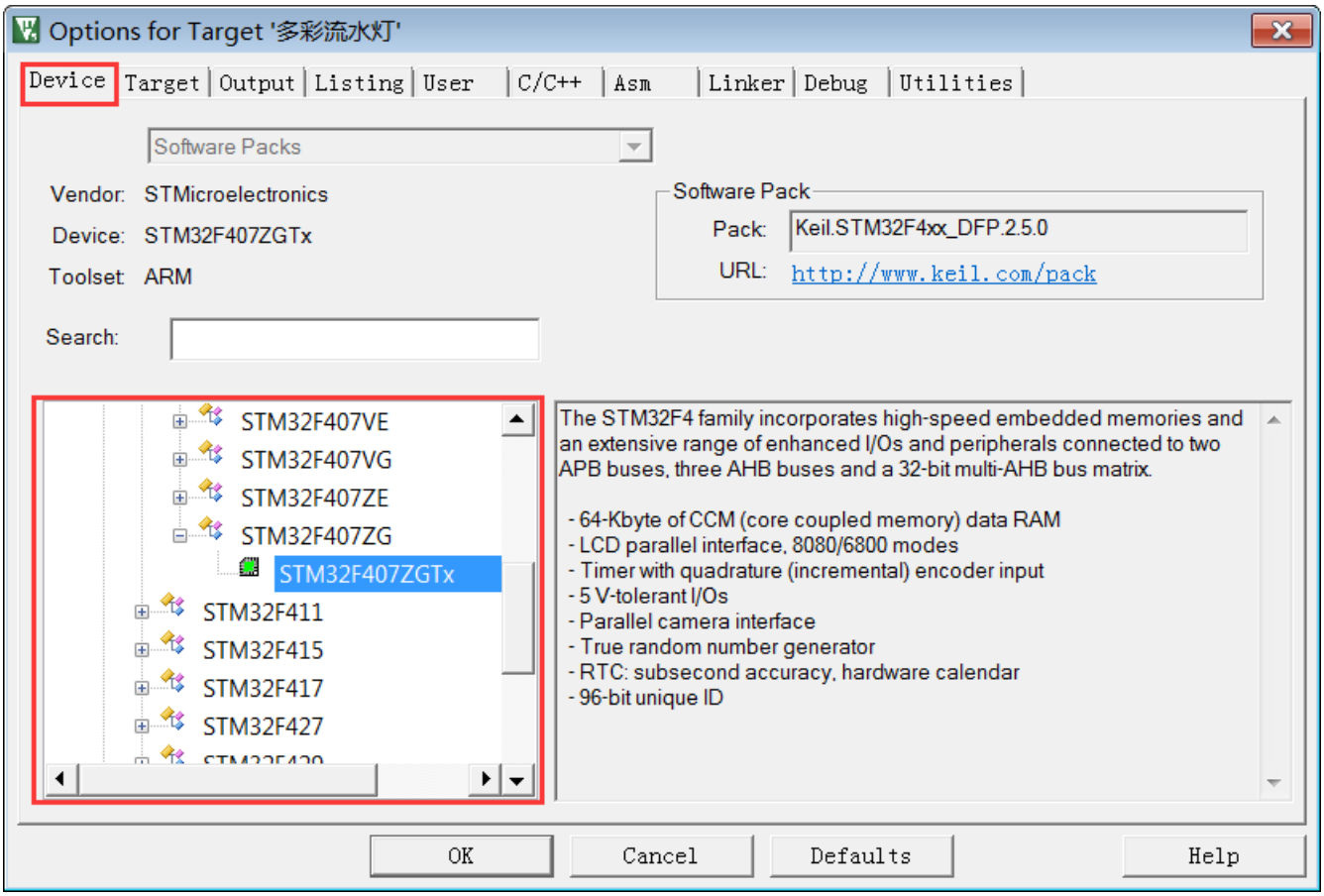


图 40-49 选择芯片类型

图中 Device 标签页中选定了芯片的型号为 STM32F407ZGTx，选中后，在 Target 标签页中的存储器信息会根据芯片更新。

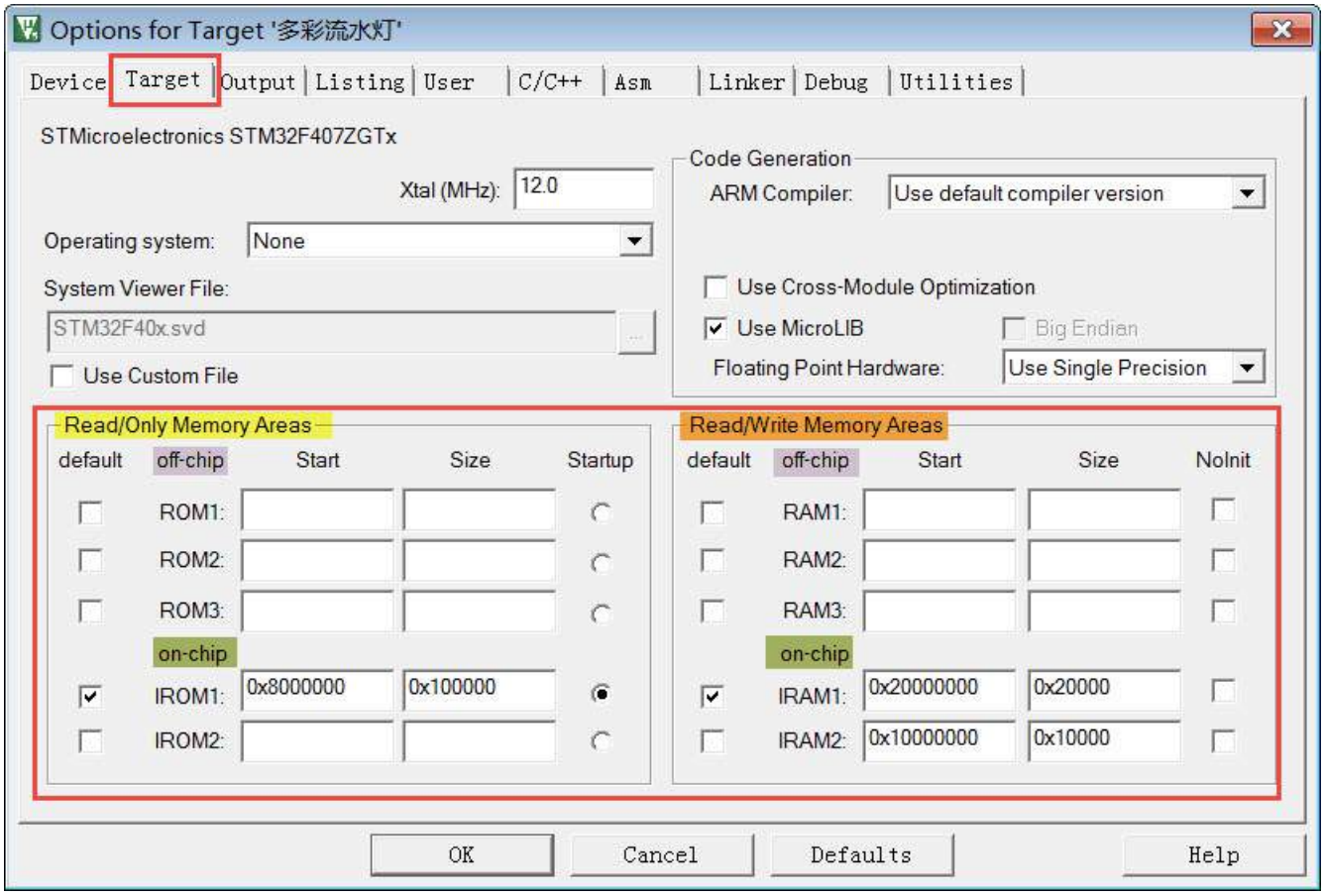


图 40-50 Target 对话框中的存储器分配

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 0x100000，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 0x20000，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。

在标签页中的 IRAM2 一栏默认也填写了配置信息，它的地址为 0x10000000，大小为 0x10000，这是 STM32F4 系列特有的内部 64KB 高速 SRAM(被称为 CCM)。当我们希望使用这部分存储空间的时候需要勾选该配置，另外要注意这部分高速 SRAM 仅支持 CPU 总线的访问，不能通过外设

访问。

下面我们尝试修改 Target 标签页中的这些存储信息，例如，按照图 *40_51* 中的 1 配置，把 IRAM1 的基地址改为 0x20001000，然后编译工程，查看到工程的 sct 文件如代码清单 *40_20* 所示；当按照图 *40_51* 中的 2 配置时，同时使用 IRAM1 和 IRAM2，然后编译工程，可查看到工程的 sct 文件如代码清单 47 21 所示。

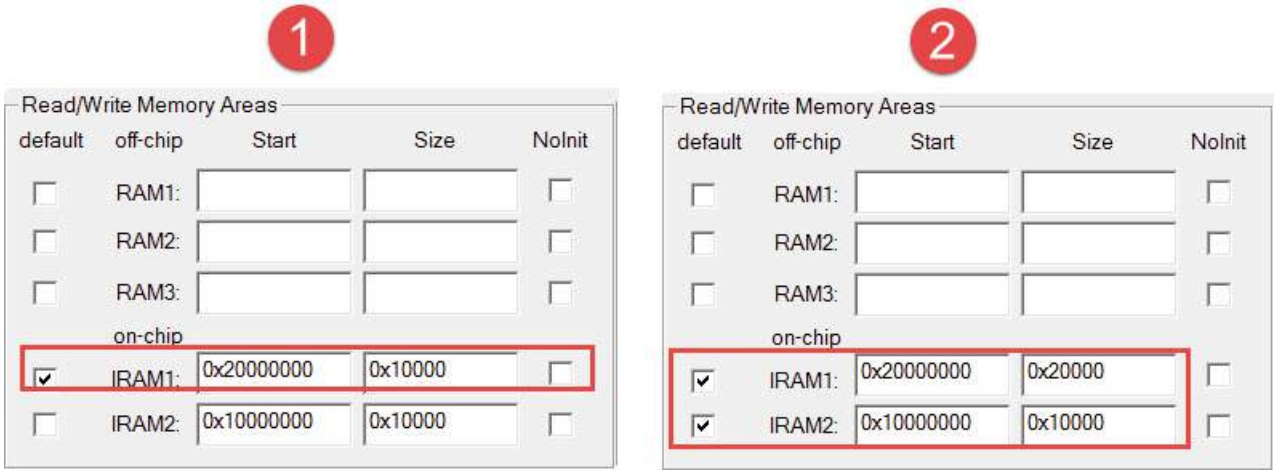


图 40-51 修改 IRAM1 的基地址及仅使用 IRAM2 的配置

列表 20: 代码清单 40-20 修改了 IRAM1 基地址后的 sct 文件内容

```
LR_IROM1 0x08000000 0x00100000  {    ; load region size_region
ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20001000 0x00010000  {  ; RW data
   .ANY (+RW +ZI)
}
}
```

列表 21: 代码清单仅使用 IRAM2 时的 sct 文件内容

```
LR_IROM1 0x08000000 0x00100000  {    ; load region size_region
ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00020000  {  ; RW data
   .ANY (+RW +ZI)
}
RW_IRAM2 0x10000000 0x00010000  {
   .ANY (+RW +ZI)
}
}
```

可以发现，sct 文件都根据 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。

47.4.5.3.3 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*。

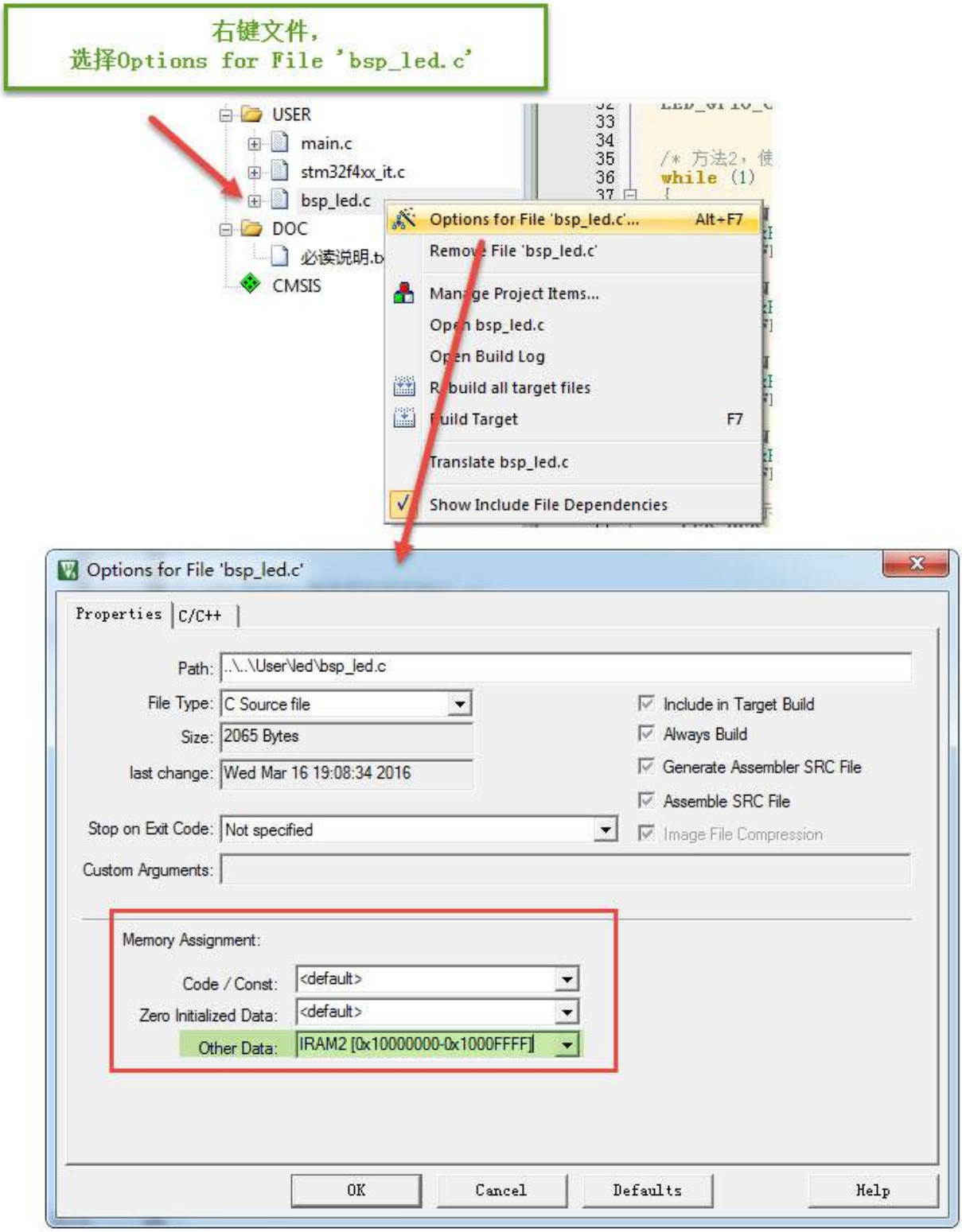

右键文件，
选择Options for File 'bsp_led.c'
USER
main.c
stm32f4xx_it.c
bsp_led.c
DOC
必读说明.b
CMSIS
Options for File 'bsp_led.c'... Alt+F7
Remove File 'bsp_led.c'
Manage Project Items...
Open bsp_led.c
Open Build Log
Rebuild all target files
Build Target F7
Translate bsp_led.c
Show Include File Dependencies
Options for File 'bsp_led.c'
Properties
C/C++
Path: ..\..\User\led\bsp_led.c
File Type: C Source file
Size: 2065 Bytes
last change: Wed Mar 16 19:08:34 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 [0x10000000-0x1000FFFF]
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 22: 代码清单 40-21 修改 bsp_led.c 配置后的 sct 文件

```
LR_IROM1 0x08000000 0x00100000  {    ; load region size_region
ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00020000  {  ; RW data
   .ANY (+RW +ZI)
}
RW_IRAM2 0x10000000 0x00010000  {
   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。

47.5 实验：自动分配变量到外部 SRAM 空间

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存储器定制。

在本书前面的 SRAM 实验中，当我们需要读写 SRAM 存储的内容时，需要使用指针或者 __attribute__((at(具体地址))) 来指定变量的位置，当有多个这样的变量时，就需要手动计算地址空间了，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外部 SRAM 的地址并进行管理，使得利用外部 SRAM 的空间就跟内部 SRAM 一样简单。

47.5.1 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 23: 代码清单 40_22 直接指定变量地址

```
1 /* 定义一个要指定的地址 */

#define USER_ADDR ((uint32_t)0x20005000)

3 /* 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)

```
4 uint8_t testValue __attribute__((at(USER_ADDR)));

5 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。

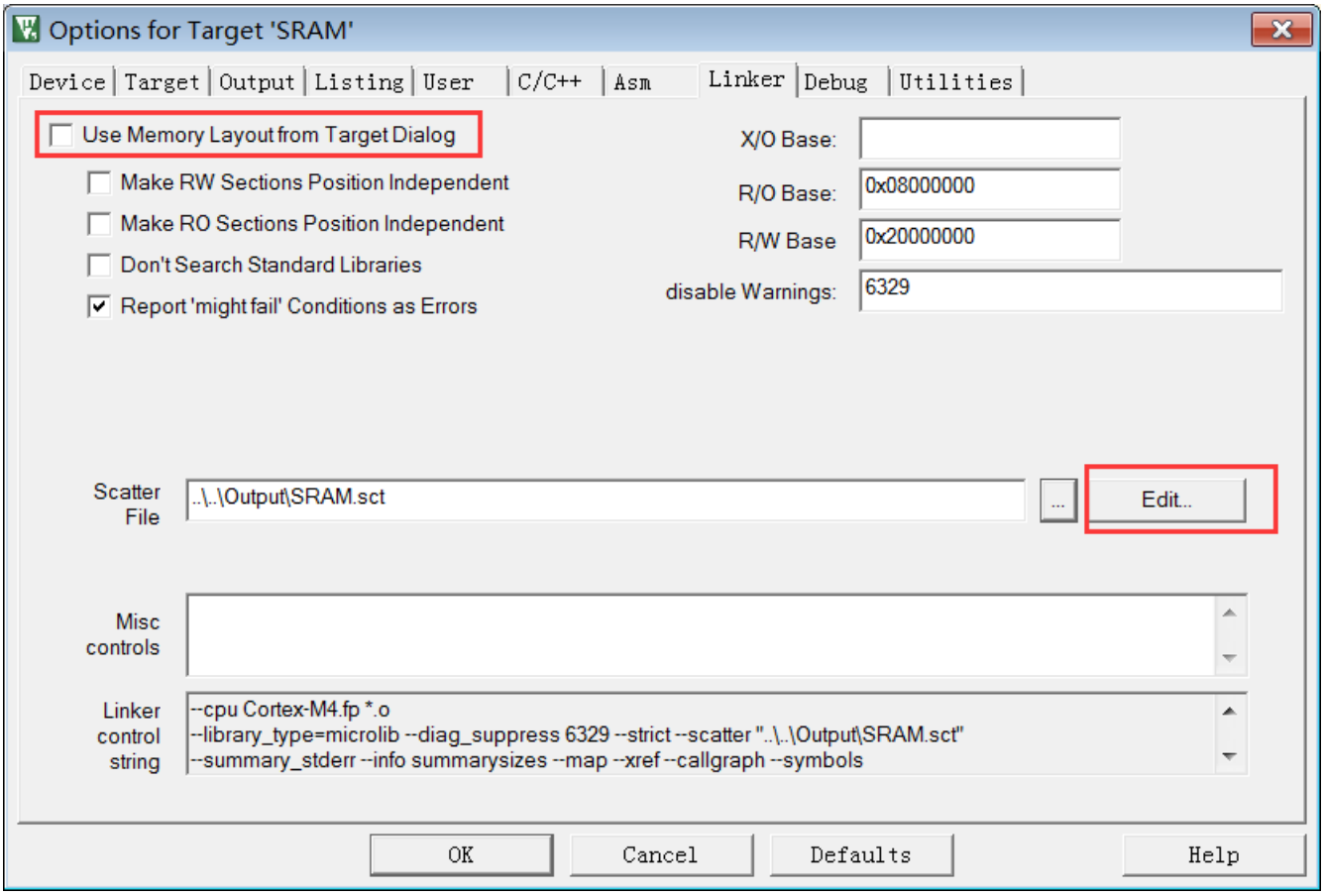
47.5.2 硬件设计

本小节中使用到的硬件跟“扩展外部 SRAM”实验中的一致，若不了解，请参考该章节的原理图说明。

47.5.3 软件设计

本小节中提供的例程名为“SCT 文件应用—自动分配变量到外部 SRAM”，学习时请打开该工程来理解，该工程是基于“扩展外部 SRAM”实验改写而来的。

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。

47.5.3.1 编程要点

(1) 修改启动文件，在 __main 执行之前初始化外部 SRAM；

(2) 在 sct 文件中增加外部 SRAM 空间对应的执行域；

(3) 使用节区选择语句选择要分配到外部 SRAM 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。

47.5.3.2 代码分析

47.5.3.2.1 在 __main 之前初始化外部 SRAM

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外部 SRAM 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外部 SRAM 存储器正常运转，使它能够正常保存数据。

在本来的“扩展外部 SRAM”工程中，我们使用 FSMC_SRAM_Init 函数初始化外部 SRAM，且该函数在 main 函数里才被调用，所以在外部 SRAM 正常运转之前，分散加载过程复制到外部 SRAM 中的数据都丢失了，因而需要在初始化外部 SRAM 之后，需要重新给变量赋值才能正常使用 (即定义变量时的初值无效，在调用外部 SRAM_Init 函数之后的赋值才有效)。

为了解决这个问题，可修改工程的 startup_stm32f40xx.s 启动文件，见*代码清单 40_23*。

列表 24: 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
Reset_Handler    PROC
                  EXPORT  Reset_Handler             [WEAK]
            IMPORT  SystemInit
            IMPORT  __main

         ; 从外部文件引入声明
         IMPORT FSMC_SRAM_Init

                  LDR     R0, =SystemInit
                  BLX     R0

            ; 在__main 之前调用 SDRAM_Init 进行初始化
            LDR   R0, =FSMC_SRAM_Init
```

(下页继续)

(续上页)

```
            BLX    R0

                   LDR      R0, =__main
                   BX       R0
                   ENDP
```

在原来的启动文件中我们增加了上述加粗表示的代码，增加的代码中使用到汇编语法的 IMPOR 引入在 bsp_sram.c 文件中定义的 FSMC_SRAM_Init 函数，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。

以上代码实现了 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。

47.5.3.2.2 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外部 SRAM 中，见*代码清单 40_24*。

列表 25: 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 0x00100000  {    ; 加载域
ER_IROM1 0x08000000 0x00100000  {  ; 加载地址 = 执行地址
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}

RW_IRAM1 0x20000000 0x00020000  {  ; 内部 SRAM
```

(下页继续)

(续上页)

```
*.o(STACK)        ; 选择 STACK 节区，栈
stm32f4xx_rcc.o(+RW)              ; 选择 stm32f4xx_rcc 的 RW 内容
.ANY (+RW +ZI)                    ; 其余的 RW/ZI-data 都分配到这里
}

RW_ERAM1 0x6C000000 0x00100000 {  ; 外部 SRAM

   .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 0x6C000000 0x100000{}

  RW_ERAM1 是我们配置的外部 SRAM 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，这两个值与我们实验板配置的外部 SRAM 基地址及空间大小一致，所以该执行域会被映射到外部 SRAM 的空间。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外部 SRAM 区域，所以我们在工程中的 C 文件定义全局变量时，它都会被分配到这个外部 SRAM 区域。

- RW_IRAM1 执行域

  RW_IRAM1 是 STM32 内部 SRAM 的执行域。我们在默认配置中增加了 “*.o(STACK) 及 stm32f4xx_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?

  这是由于我们在 __main 之前调用的 FSMC_SRAM_Init 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)，见图 *40_0_54*，查看静态调用图文件 “SRAM.htm” 可了解它使用了多少栈空间以及调用了哪些函数。

```
FSMC_SRAM_Init (Thumb, 90 bytes, Stack size 96 bytes, bsp_sram.o(i.FSMC_SRAM_Init))

[Stack]

  • Max Depth = 168
  • Call Chain = FSMC_SRAM_Init ⇒ SRAM_GPIO_Config ⇒ GPIO_Init

[Calls]

  • >>   FSMC_NORSRAMInit
  • >>   FSMC_NORSRAMCmd
  • >>   RCC_AHB3PeriphClockCmd
  • >>   SRAM_GPIO_Config

[Address Reference Count : 1]

  • startup_stm32f40xx.o(.text)
```

从文件中可了解到 FSMC_SRAM_Init 使用的 STACK 的深度为 168 字节，它调用了 FMC_NORSRAMInit、FMC_NORSRAMCmd、RCC_AHB3PeriphClockCmd 及 SRAM_GPIO_Config 函数。由于它使用了栈空间，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 SRAM 芯片却并未正常工作，这样的矛盾导致栈空间不能被分配到外部 SRAM。

虽然内部 SRAM 的执行域 RW_IRAM1 及 SDRAM 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 SDRAM 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，栈节区会被分配到外部 SRAM，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“stm32f4xx_rcc.o(+RW)”语句是因为 FSMC_SRAM_Init 函数调用了 stm32f4xx_rcc.c 文件中的 RCC_AHB3PeriphClockCmd 函数，而查看 map 文件后了解到 stm32f4xx_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_Initt 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

```
1852
1853  Image component sizes
1854
1855
1856       Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
1857
1858        188         16          0          0          0       1409   bsp_debug_usart.o
1859         88          4          0          0          0        658   bsp_led.o
1860       1026         18          0          0          0       1761   bsp_sram.o
1861        816        578        227        108        100       3041   main.o
1862         44         12        392          0       1536       1008   startup_stm32f40xx.o
1863        268         16          0          0          0       2557   stm32f4xx_fsmc.o
1864        156          0          0          0          0       2247   stm32f4xx_gpio.o
1865         18          0          0          0          0       4810   stm32f4xx_it.o
1866        248         34          0         16          0       5751   stm32f4xx_rcc.o
1867        234          8          0          0          0       4164   stm32f4xx_usart.o
1868        264         34          0          0          0     280793   system_stm32f4xx.o
1869
```

图 40_54 SRAM.map 文件中查看到 stm32f4xx_rcc.o 文件的 RW-data 使用统计信息

47.5.3.2.3 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 26: 代码清单 40-25 main 文件

```
//定义变量到外部 SRAM
uint32_t testValue  =7 ;
//定义变量到外部 SRAM
uint32_t testValue2  =0;

//定义数组到外部 SRAM
uint8_t testGrup[100]  = {0};
//定义数组到外部 SRAM
uint8_t testGrup2[100] = {1,2,3};

/* 本实验中的 sct 配置，堆区可能工作不正常，
```

(下页继续)

(续上页)

```
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 */

/*
* 函数名：main
* 描述  ：主函数
* 输入  ：无
* 输出  ：无
*/
int main(void)
{
   uint32_t inerTestValue =10;

   LED_GPIO_Config();
   LED_BLUE;

   /* 配置串口 1 为：115200 8-N-1 */
   Debug_USART_Config();

   //FSMC_SRAM_Init 已经在启动文件的 Reset_handler 中调用，进入 main 之前已经完成初
始化 */
//    FSMC_SRAM_Init();

   printf("\r\nSCT 文件应用——自动分配变量到外部 SRAM 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  =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);
```

(下页继续)

(续上页)

```
  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, 堆区可能工作不正常,
  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");
        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--));
```

(下页继续)

(续上页)

```
        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*。

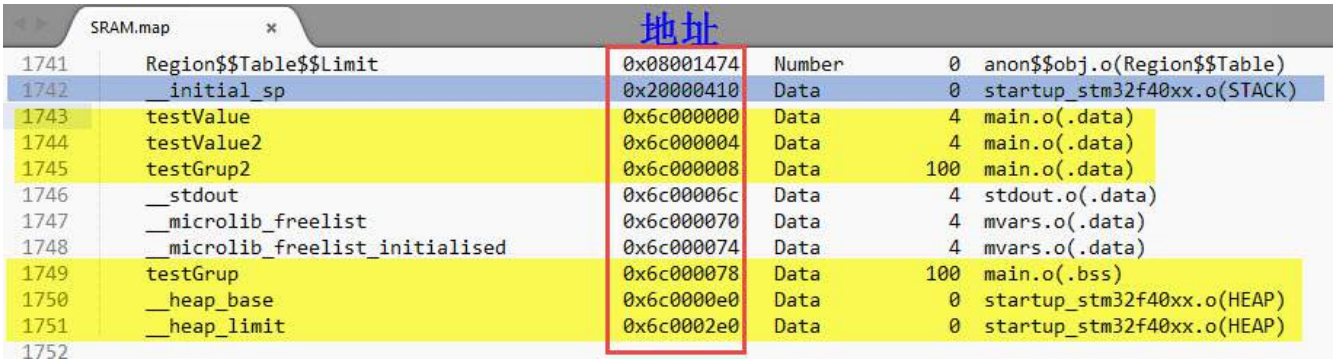


图 40-55 在 map 文件中查看工程的存储分布 1(SRAM.map 文件)

SRAM.map

```
1831    Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x00000410, Max: 0x00020000, ABSOLUTE)
1832
1833    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
1834
1835    0x20000000   0x00000010   Data   RW         2614    .data               stm32f4xx_rcc.o
1836    0x20000010   0x00000400   Zero   RW            1    STACK               startup_stm32f40xx.o
1837
1838
1839    Execution Region RW_ERAM1 (Base: 0x6c000000, Size: 0x000002e0, Max: 0x00100000, ABSOLUTE)
1840
1841    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
1842
1843    0x6c000000   0x0000006c   Data   RW         4768    .data               main.o
1844    0x6c00006c   0x00000004   Data   RW         5295    .data               mc_w.l(stdout.o)
1845    0x6c000070   0x00000004   Data   RW         5304    .data               mc_w.l(mvars.o)
1846    0x6c000074   0x00000004   Data   RW         5305    .data               mc_w.l(mvars.o)
1847    0x6c000078   0x00000064   Zero   RW         4766    .bss                main.o
1848    0x6c0000dc   0x00000004   PAD
1849    0x6c0000e0   0x00000200   Zero   RW            2    HEAP                startup_stm32f40xx.o
1850
1851
1852  ===============================================================================
1853
1854  Image component sizes
1855
1856
1857       Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
1858
1859        188         16          0          0          0       1505   bsp_debug_usart.o
1860         88          4          0          0          0        706   bsp_led.o
1861       1026         18          0          0          0       1857   bsp_sram.o
1862        820        582        227        108        100       3089   main.o
1863         44         12        392          0       1536       1056   startup_stm32f40xx.o
1864        268         16          0          0          0       2653   stm32f4xx_fsmc.o
1865        156          0          0          0          0       2343   stm32f4xx_gpio.o
```

图 40-56 在 map 文件中查看工程的存储分布 2(SDRAM.map 文件)

从 map 文件中，可看到 stm32f4xx_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，即我们扩展的外部 SRAM 空间中，看起来一切都与我们的 sct 文件配置一致了。(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外部 SRAM 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*。

```
SCT文件应用——自动分配变量到外部SRAM实验

使用“  uint32_t inerTestValue =10; ”语句定义的局部变量:
结果：它的地址为：0x20000404,变量值为：10

使用“uint32_t testValue  =7 ;”语句定义的全局变量:
结果：它的地址为：0x6c000000,变量值为：7

使用“uint32_t testValue2  =0 ; ”语句定义的全局变量:
结果：它的地址为：0x6c000004,变量值为：0

使用“uint8_t testGrup[100]  ={0};”语句定义的全局数组:
结果：它的地址为：0x6c000078,变量值为：0,0,0

使用“uint8_t testGrup2[100] ={1,2,3};”语句定义的全局数组:
结果：它的地址为：0x6c000008,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：0x6c0000f0,查看变量值操作:
*(pointer--)=3,
*(pointer--)=2,
*(pointer)=1,
```

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，全局变量定义到了外部 SRAM 的区域。

经过我的测试，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 SRAM 区域，都无法正常使用 malloc 分配空间。另外，由于外部 SRAM 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 SRAM。

在下一小节中我们将改进 sct 的文件配置，解决这两个问题。

47.6 实验：优先使用内部 SRAM 并把堆区分配到外部 SRAM

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外部 SRAM，另外，我们还把系统默认的堆空间 (HEAP) 映射到外部 SRAM，从而可以使用 C 语言 HAL 库的 malloc 函数动态从外部 SRAM 中分配空间，利用 HAL 库进行外部 SRAM 的空间内存管理。

47.6.1 硬件设计

本小节中使用到的硬件跟“扩展外部 SRAM”实验中的一致，若不了解，请参考该章节的原理图说明。

47.6.2 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 并把堆分配到外部 SRAM 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*。

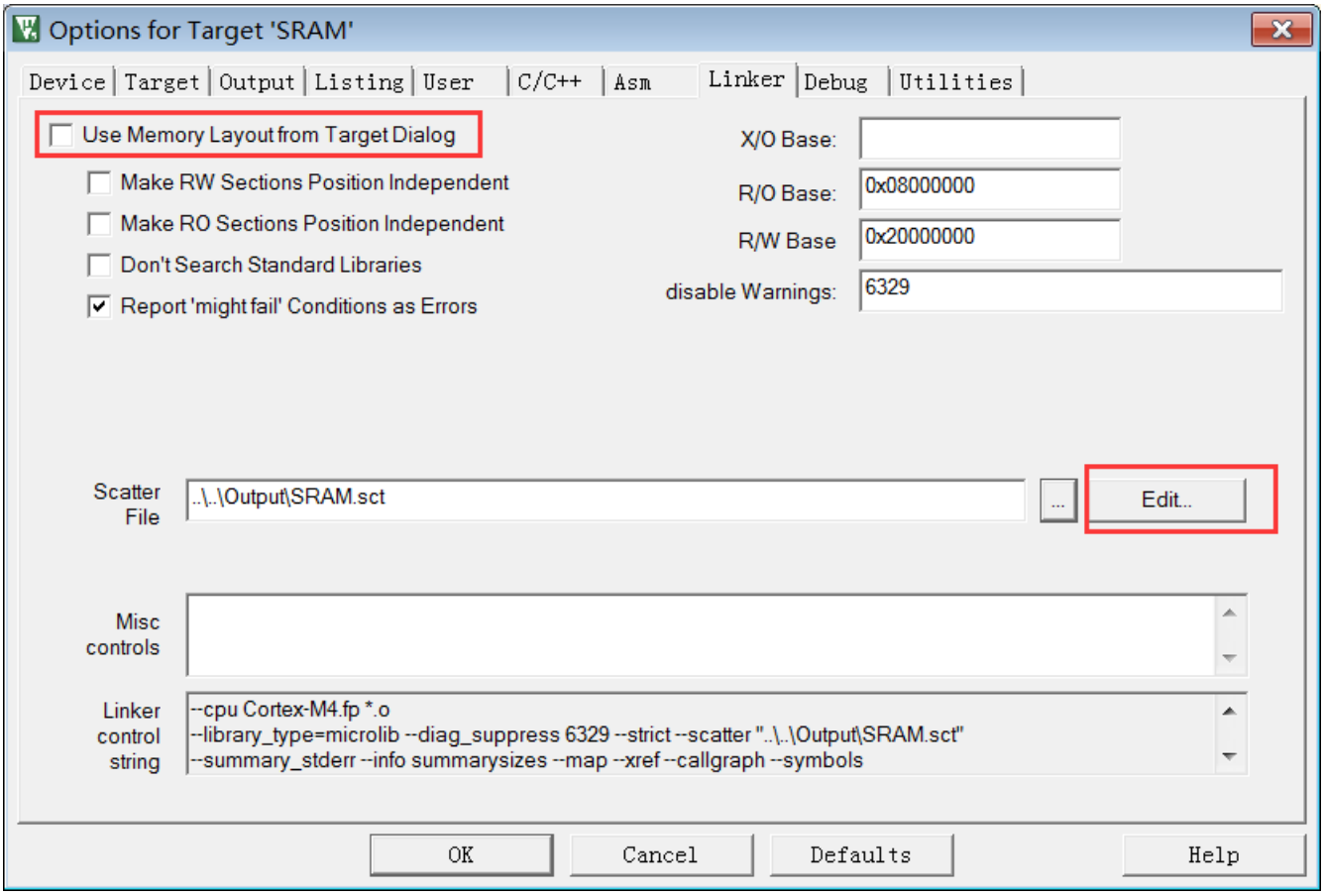


图 40-58 使用手动编写的 sct 文件

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。

47.6.2.1 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。

47.6.2.2 代码分析

47.6.2.2.1 在 __main 之前初始化外部“指定的存储空间”的硬件

同样地，为了使定义到外部 SRAM 的变量能被正常初始化，需要修改工程 startup_stm32f40xx.s 启动文件中的 Reset_handler 函数，在 __main 函数之前调用 FSMC_SRAM_Init 函数使外部 SRAM 硬件正常运转，见*代码清单 40_26*。

列表 27: 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
          IMPORT FSMC_SRAM_Init

                 LDR     R0, =SystemInit
                 BLX     R0

             ; 在__main 之前调用 SDRAM_Init 进行初始化
             LDR   R0, = FSMC_SRAM_Init
             BLX   R0

                LDR     R0, =__main
```

(下页继续)

(续上页)

```
                BX      R0
                ENDP
```

它与上一小节中的改动一样，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。

47.6.2.2.2 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 28: 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 0x00100000  {    ; load region size_region
ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}


RW_IRAM1 0x20000000 0x00030000  {  ; 内部 SRAM
.ANY (+RW +ZI)       ; 其余的 RW/ZI-data 都分配到这里
}

RW_ERAM1 0x6C000000 0x00100000 {  ; 外部 SRAM
*.o(HEAP)             ; 选择堆区
   .ANY (EXRAM)        ; 选择 EXRAM 节区
}
```

(下页继续)

(续上页)

```
}
```

本实验的 sct 文件中对内部 SRAM 的执行域保留了默认配置，没有作任何改动，新增了一个外部 SRAM 的执行域，并且使用了“*.o(HEAP)”语句把堆区分配到了外部 SRAM 空间，使用“.ANY(EXRAM)”语句把名为“EXRAM”的节区分配到外部 SRAM 空间。

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外部 SRAM 时，只需要指定该变量分配到该节区，它就会被分配到外部 SRAM 空间。

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。

47.6.2.2.3 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 29: 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外部 SRAM
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。

47.6.2.2.4 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_29*。

列表 30: 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")))

//定义变量到外部 SRAM
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 ;
```

(下页继续)

(续上页)

```
//定义数组到外部 SRAM
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;
   /*FSMC_SRAM_Init 已经在启动文件的 Reset_handler 中调用,
   进入 main 之前已经完成初始化 */
//    FSMC_SRAM_Init();

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

   /* 初始化串口 */
   Debug_USART_Config();

   printf("\r\nSCT 文件应用——自动分配变量到外部 SRAM 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",
```

(下页继续)

(续上页)

```
        (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;
     *(++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));
```

(下页继续)

(续上页)

```
    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

1741	Region$$Table$$Base	0x08001430	Number	0	anon$$obj.o(Region$$Table)
1742	Region$$Table$$Limit	0x08001470	Number	0	anon$$obj.o(Region$$Table)
1743	testGrup2	0x20000010	Data	3	main.o(.data)
1744	testValue2	0x20000014	Data	4	main.o(.data)
1745	__stdout	0x20000018	Data	4	stdout.o(.data)
1746	__microlib_freelist	0x2000001c	Data	4	mvars.o(.data)
1747	__microlib_freelist_initialised	0x20000020	Data	4	mvars.o(.data)
1748	__initial_sp	0x20000428	Data	0	startup_stm32f40xx.o(STACK)
1749	testGrup	0x6c000000	Data	3	main.o(EXRAM)
1750	testValue	0x6c000004	Data	4	main.o(EXRAM)
1751	__heap_base	0x6c000008	Data	0	startup_stm32f40xx.o(HEAP)
1752	__heap_limit	0x6c000208	Data	0	startup_stm32f40xx.o(HEAP)
1753					

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外部 SRAM 的地址区域，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——自动分配变量到外部SRAM实验

使用“  uint32_t inerTestValue =10; ”语句定义的局部变量:
结果: 它的地址为: 0x20000404,变量值为: 10

使用“uint32_t testValue  =7 ;”语句定义的全局变量:
结果: 它的地址为: 0x6c000000,变量值为: 7

使用“uint32_t testValue2  =0 ; ”语句定义的全局变量:
结果: 它的地址为: 0x6c000004,变量值为: 0

使用“uint8_t testGrup[100]  ={0};”语句定义的全局数组:
结果: 它的地址为: 0x6c000078,变量值为: 0,0,0

使用“uint8_t testGrup2[100] ={1,2,3};”语句定义的全局数组:
结果: 它的地址为: 0x6c000008,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: 0x6c0000f0,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。

本实验中的 sct 文件配置方案完全可以应用到您的实际工程项目中，下面再进一步强调其它应用细节。

47.6.2.2.5 使用 malloc 和 free 管理外部 SRAM 的空间

外部 SRAM 的内存空间非常大，为了管理这些空间，一些工程师会自己定义内存分配函数来管理外部 SRAM 空间，这些分配过程本质上就是从外部 SRAM 中动态分配内存。从本实验中可了解到我们完全可以直接使用 CHAL 库的 malloc 从外部 SRAM 中分配空间，只要在前面配置的基础上修改启动文件中的堆顶地址限制即可，见代码清单。

```
Heap_Size        EQU      0x00000200

                 AREA     HEAP, NOINIT, READWRITE, ALIGN=3



__heap_base
Heap_Mem         SPACE    Heap_Size
__heap_limit     EQU    0x6C100000     ; 设置堆空间的极限地址 (外部 SRAM),
;0x6c000000+0x00100000

                 PRESERVE8
                 THUMB
```

CHAL 库的 malloc 函数是根据 __heap_base 及 __heap_limit 地址限制分配空间的，在以上的代码定义中，堆基地址 __heap_base 的由链接器自动分配未使用的基地址，而堆顶地址 __heap_limit 则被定义为外部 SRAM 的最高地址 0x6c000000+0x100000 (使用这种定义方式定义的 __heap_limit 值与 Heap_Size 定义的大小无关)，经过这样配置之后，外部 SRAM 内除 EXRAM 节区外的空间都被分配为堆区，所以 malloc 函数可以管理剩余的所有外部 SRAM 空间。修改后，它生成的 map 文件信息见图 *40_70*。

SRAM.map

```
1745    __stdout                                0x20000018   Data          4  stdout.o(.data)
1746    __microlib_freelist                     0x2000001c   Data          4  mvars.o(.data)
1747    __microlib_freelist_initialised         0x20000020   Data          4  mvars.o(.data)
1748    __initial_sp                            0x20000428   Data          0  startup_stm32f40xx.o(STACK)
1749    testGrup                                0x6c000000   Data          3  main.o(EXRAM)
1750    testValue                               0x6c000004   Data          4  main.o(EXRAM)
1751    __heap_base                             0x6c000008   Data          0  startup_stm32f40xx.o(HEAP)
1752    __heap_limit                            0x6c100000   Number        0  startup_stm32f40xx.o ABSOLUTE
1753
```

SRAM.map

```
1844
1845    Execution Region RW_ERAM1 (Base: 0x6c000000, Size: 0x00000208, Max: 0x00100000, ABSOLUTE)
1846
1847    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
1848
1849    0x6c000000   0x00000008   Data   RW         4769    EXRAM               main.o
1850    0x6c000008   0x00000200   Zero   RW            2    HEAP                startup_stm32f40xx.o
1851
```

图 40_70 使用 malloc 管理剩余外部 SRAM 空间

可看到 __heap_base 的地址紧跟在 EXRAM 之后，__heap_limit 指向了外部 SRAM 的最高地址，因此 malloc 函数就能利用所有外部 SRAM 的剩余空间了。注意图中显示的 HEAP 节区大小为 0x00000200 字节，修改启动文件中的 Heap_Size 大小可以改变该值，它的大小是不会影响 malloc 函数使用的，malloc 实际可用的就是 __heap_base 与 __heap_limit 之间的空间。至于如何使 Heap_Size 的值能自动根据 __heap_limit 与 __heap_base 的值自动生成，我还没找到方法，若您了解，请告知。

47.6.2.2.6 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)” 选择语句从外部 SRAM 的执行域删除掉即可，堆节区就会默认分配到内部 SRAM，外部 SRAM 仅选择 EXRAM 节区的内容进行分配，见*代码清单 40_30*，若您更改了启动文件中堆的默认配置，主注意空间地址的匹配。

列表 31：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 0x00100000  {    ; load region size_region
ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
```

(下页继续)

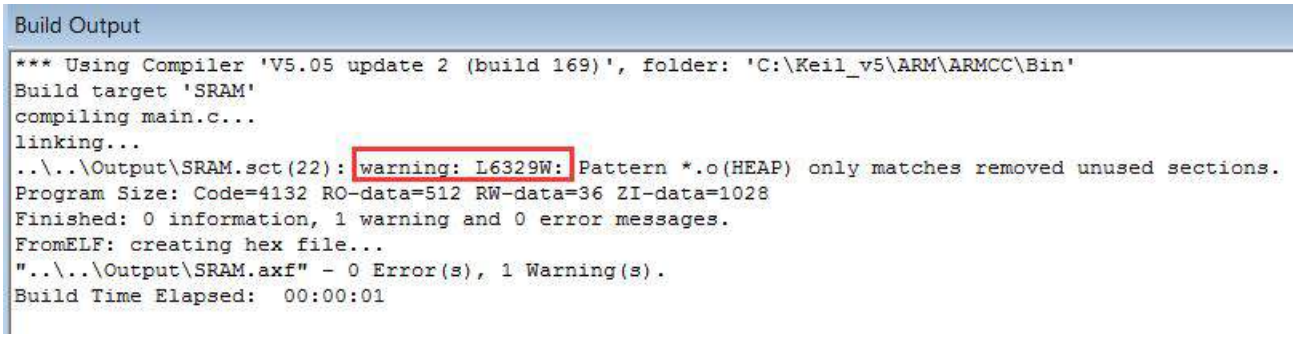
(续上页)

```
    *.o (RESET, +First)
    *(InRoot$$Sections)
    .ANY (+RO)
}
RW_IRAM1 0x20000000 0x00030000  {  ; 内部 SRAM
.ANY (+RW +ZI)       ; 其余的 RW/ZI-data 都分配到这里
}

RW_ERAM1 0x6C000000 0x00100000 {  ; 外部 SDRAM
    .ANY (EXRAM)        ; 选择 EXRAM 节区
}
}
```

47.6.2.2.7 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*。



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。

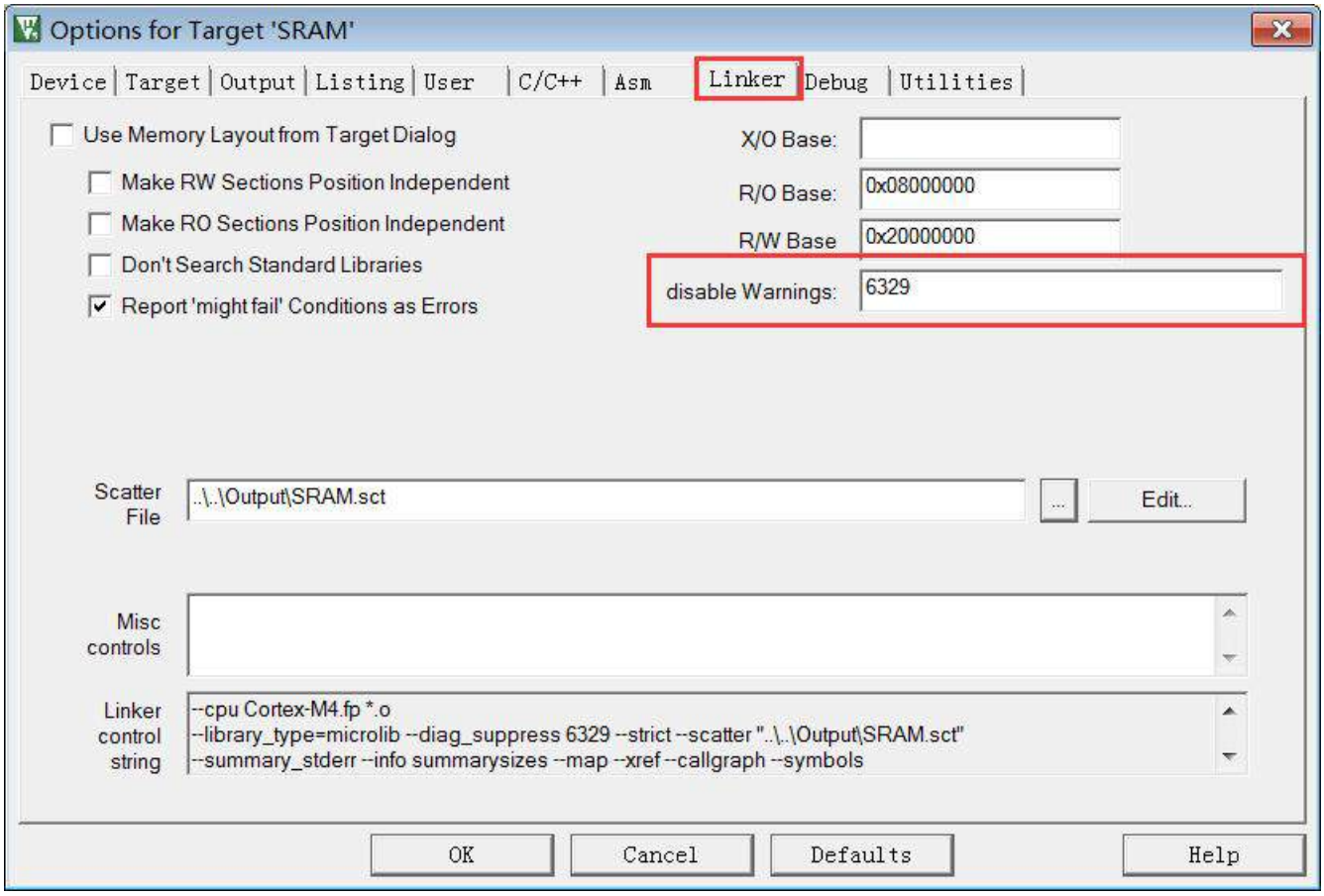


图 40-62 屏蔽链接过程的 warning

47.6.2.2.8 如何把栈空间也分配到外部 SRAM

前面提到因为 FSMC_SRAM_Init 初始化函数本身使用了栈空间 (STACK)，而在执行 FSMC_SRAM_Init 函数之前外部 SRAM 并未正常工作，这样的矛盾导致无法把栈分配到外部 SRAM。其实换一个思路，只要我们的外部 SRAM 初始化过程不使用栈空间，外部 SRAM 正常运行后栈才被分配到外部 SRAM 空间，这样就没有问题了。

由于原来的 FSMC_SRAM_Init 实现的外部 SRAM 初始化过程使用了 STM32HAL 库函数，它不可避免地使用了栈空间 (定义了局部变量)，要完全不使用栈空间完成外部 SRAM 的初始化，只能使用纯粹的寄存器方式配置。在 STM32HAL 库的“system_stm32f4xx.c”文件已经给出了类似的解决方案，SystemInit_ExtMemCtl 函数，见*代码清单 40_31*。

```
#elif defined (DATA_IN_ExtSRAM)
/**
* @brief  Setup the external memory controller.
*         Called in startup_stm32f4xx.s before jump to main.
* This function configures the external SRAM mounted on STM324xG_EVAL/
↪STM324x7I boards
*   This SRAM will be used as program data memory (including heap and␣
↪stack).
* @param  None
* @retval None
*/
void SystemInit_ExtMemCtl(void)
{
   /*-- GPIOs Configuration -----------------------------------------------
↪-----


   /* Enable GPIOD, GPIOE, GPIOF and GPIOG interface clock */
   RCC->AHB1ENR   |= 0x00000078;

   /* Connect PDx pins to FMC Alternate function */
   GPIOD->AFR[0]  = 0x00cc00cc;
   GPIOD->AFR[1]  = 0xcccccccc;
   /* Configure PDx pins in Alternate function mode */
   GPIOD->MODER   = 0xaaaa0a0a;
   /* Configure PDx pins speed to 100 MHz */
   GPIOD->OSPEEDR = 0xffff0f0f;
   /* Configure PDx pins Output type to push-pull */
   GPIOD->OTYPER  = 0x00000000;
   /* No pull-up, pull-down for PDx pins */
   GPIOD->PUPDR   = 0x00000000;
   /* 具体配置省略 */
```

(下页继续)

(续上页)

```
}
```

该函数没有使用栈空间，仅使用 register 关键字定义了两个分配到内核寄存器的变量，其余配置均通过直接操作寄存器完成。这个函数针对 ST 的一个官方评估编写的，在其它硬件平台直接使用可能有错误，若有需要可仔细分析它的代码再根据自己的硬件平台进行修改。

这个函数是使用条件编译语句“#elif defined (DATA_IN_ExtSRAM)”包装起来的，默认情况下这个函数并不会被编译，需要使用这个函数时只要定义这个宏即可。

定义了 DATA_IN_ExtSRAM 宏之后，SystemInit_ExtMemCtl 函数会被 SystemInit 函数调用，见*代码清单 40_32*，而我们又知道 SystemInit 会在启动文件中的 Reset_handler 函数中执行。

```
/**
* @brief  Setup the microcontroller system
*   Initialize the Embedded Flash Interface, the PLL and update the
*         SystemFrequency variable.
* @param  None
* @retval None
*/
void SystemInit(void)
{

/****** 部分代码省略 ******/
#if defined(DATA_IN_ExtSRAM) || defined(DATA_IN_ExtSDRAM)
SystemInit_ExtMemCtl();
#endif /* DATA_IN_ExtSRAM || DATA_IN_ExtSDRAM */
/****** 部分代码省略 ******/
}
```

所以，如果希望把栈空间分配到外部 SRAM，可按以下步骤操作：

- 修改 sct 文件，使用“*.o(STACK)”语句把栈空间分配到外部 SRAM 的执行域；
- 根据自己的硬件平台，修改 SystemInit_ExtMemCtl 函数，该函数要实现外部 SRAM 的初始化过

程，且该函数不能使用栈空间；

□ 定义 DATA_IN_ExtSDRAM 宏，从而使得 SystemInit_ExtMemCtl 函数被加进编译，并被 SystemInit 调用；

□ 由于 Reset_handler 默认会调用 SystemInit 函数执行，所以不需要修改启动文件。

47.6.2.3 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 48 章　在 SRAM 中调试代码

本章参考资料：《STM32F4xx 中文参考手册》、《STM32F4xx 规格书》、《Cortex-M3 权威指南》、《Cortex-M4 Technical Reference Manual》(跟 M3 大部分是相同的，读英文不习惯可先参考《Cortex-M3 权威指南》)。

学习本章时，配合《STM32F4xx 中文参考手册》“存储器和总线结构”及“嵌入式 FLASH 接口”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。

48.1 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。

48.2 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 STM32F407 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	0x1FFF0000	0x1FFF0004

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_stm32f407xx.s 决定了 0x00000000 和 0x00000004 地址存储什么内容，链接时，由分散加载文件 (sct) 决定这些内容的绝对地址，即分配到内部 FLASH 还是内部 SRAM。（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。

48.3 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 STM32F4 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。

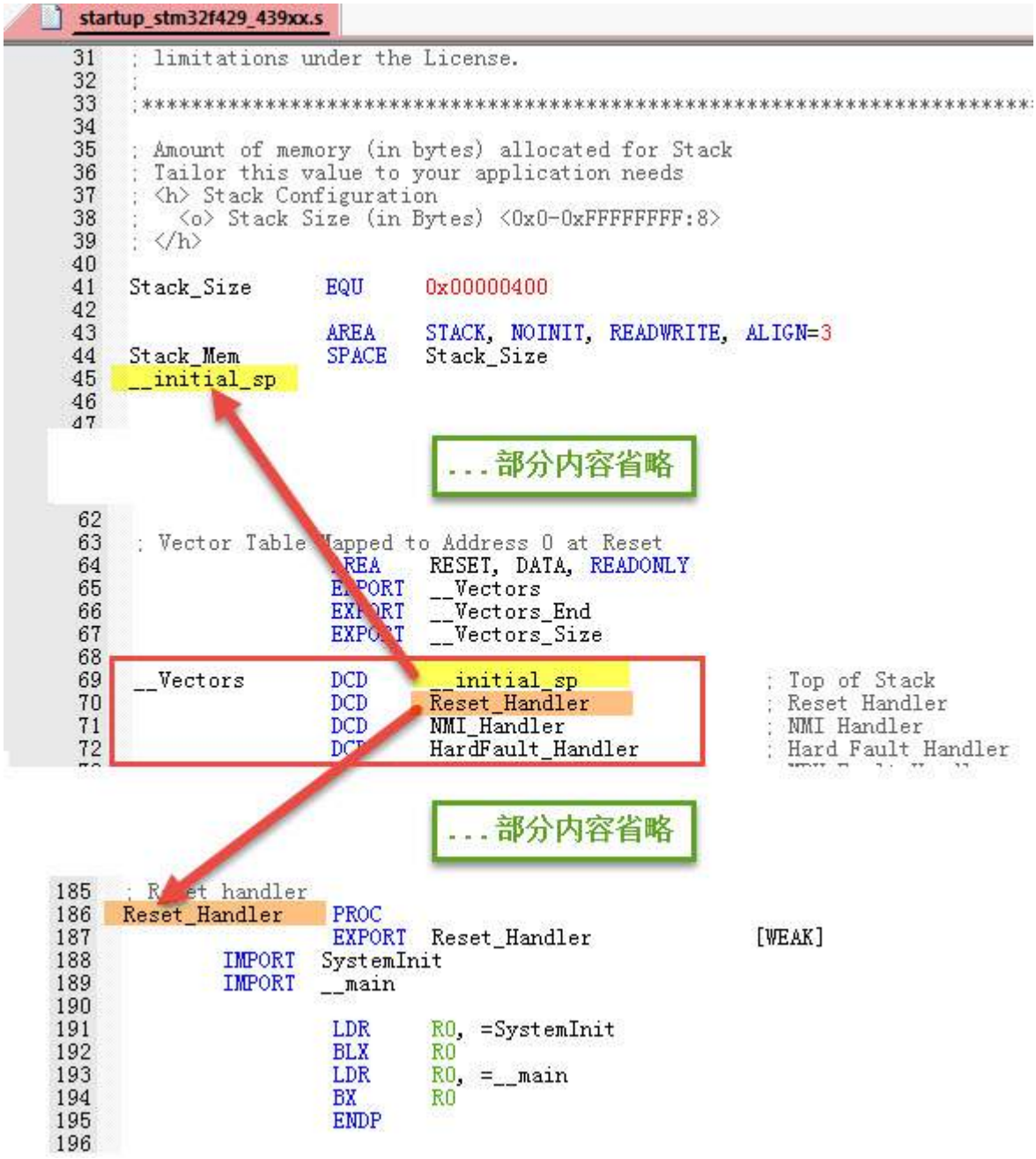


图 43-2 启动代码中存储的 MSP 及 PC 指针内容

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，STM32F407ZGT6 型号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 0x00100000  {    ; load region size_region
ER_IROM1 0x08000000 0x00100000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20000000 UNINIT 0x00020000  {  ; 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。

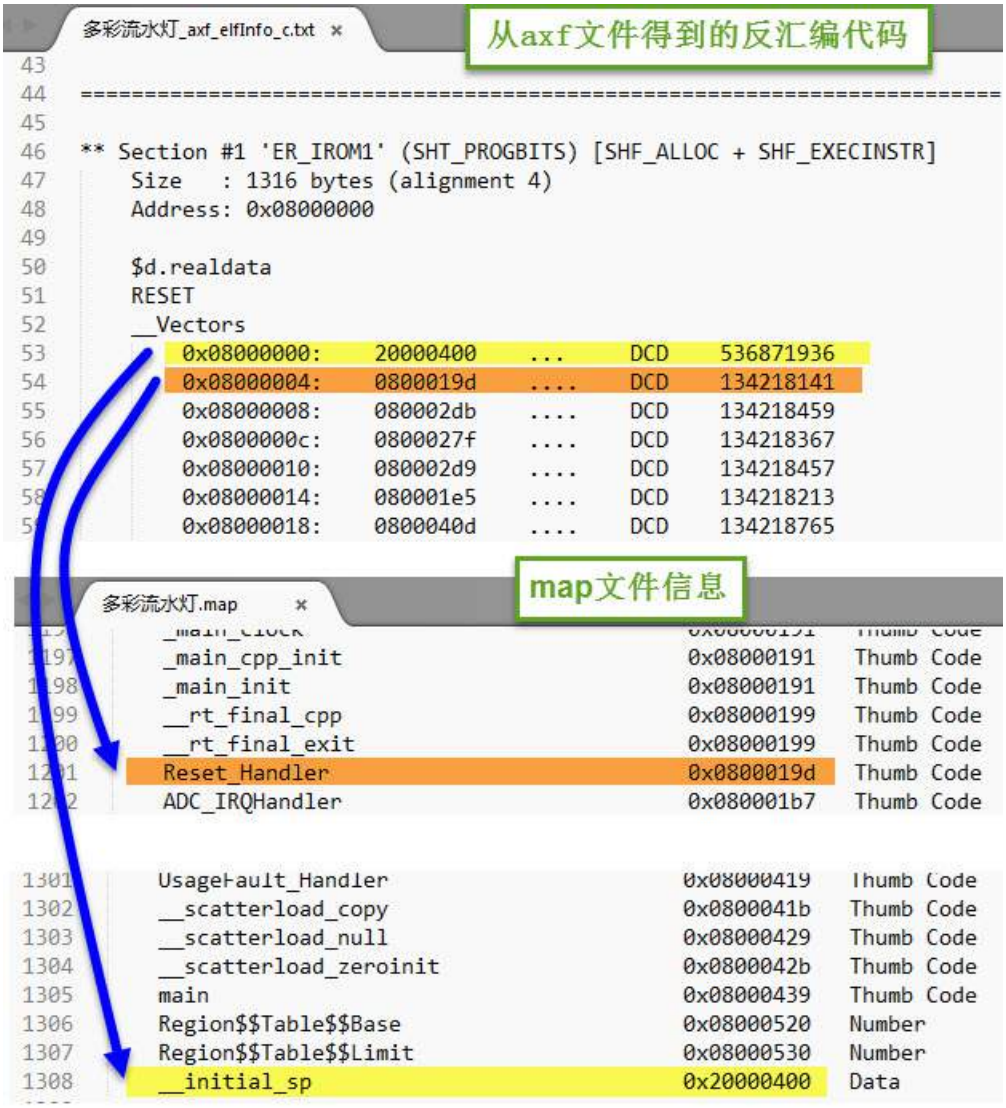


图 43-3 从反汇编代码及 map 文件查看存储器的内容

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 0x0800019d，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 0x0800019d，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。

48.4 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。

48.4.1 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*。

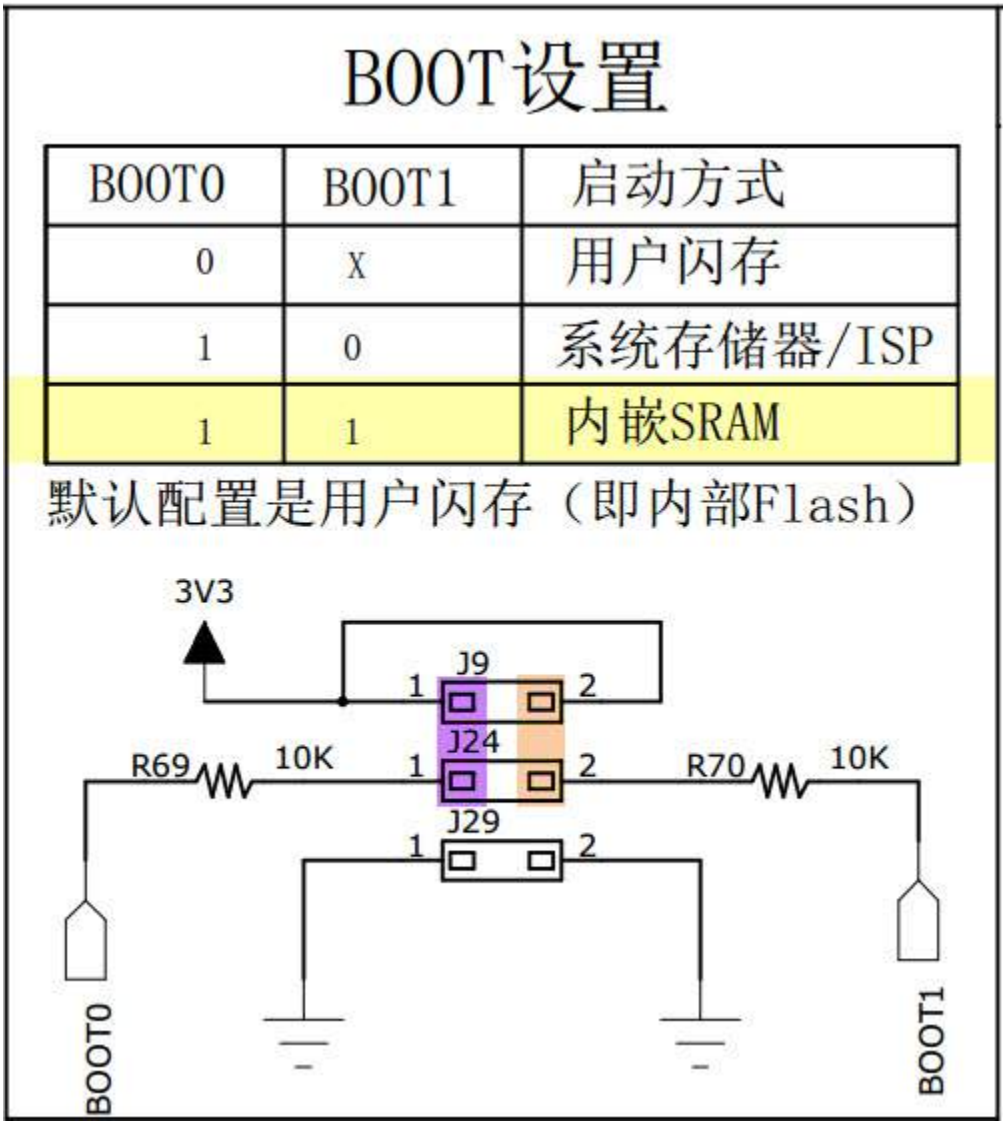


图 43-4 实验板的 boot 引脚配置

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。

48.4.2 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。

48.4.2.1 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。

48.4.2.2 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。

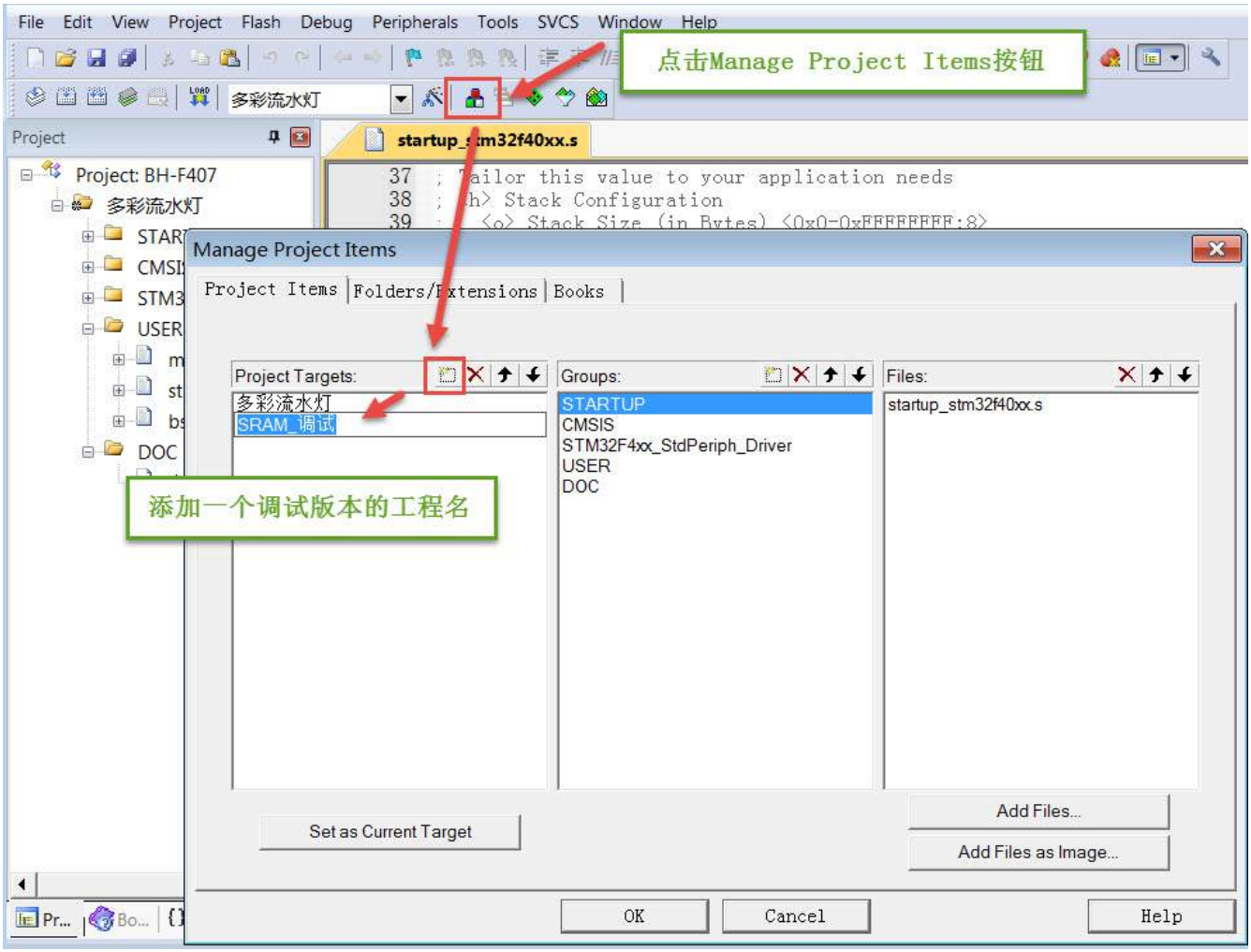


图 43-5 使用 Manage Project Items 添加一个工程配置

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。

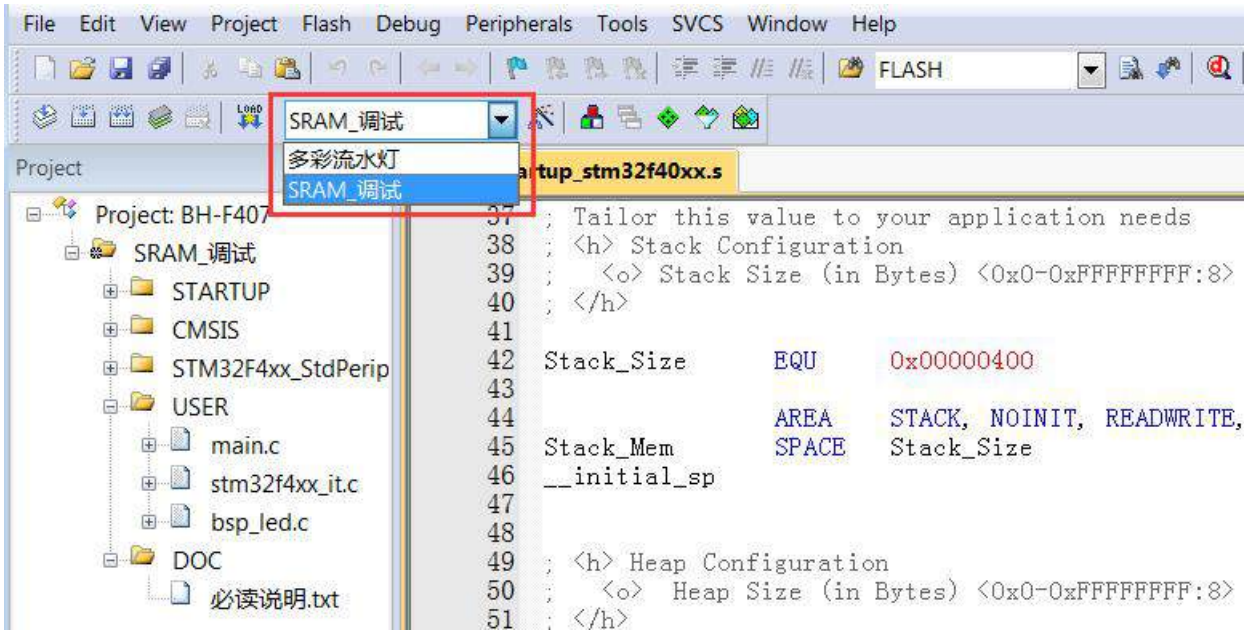


图 43-6 切换工程

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。

48.4.2.3 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*。

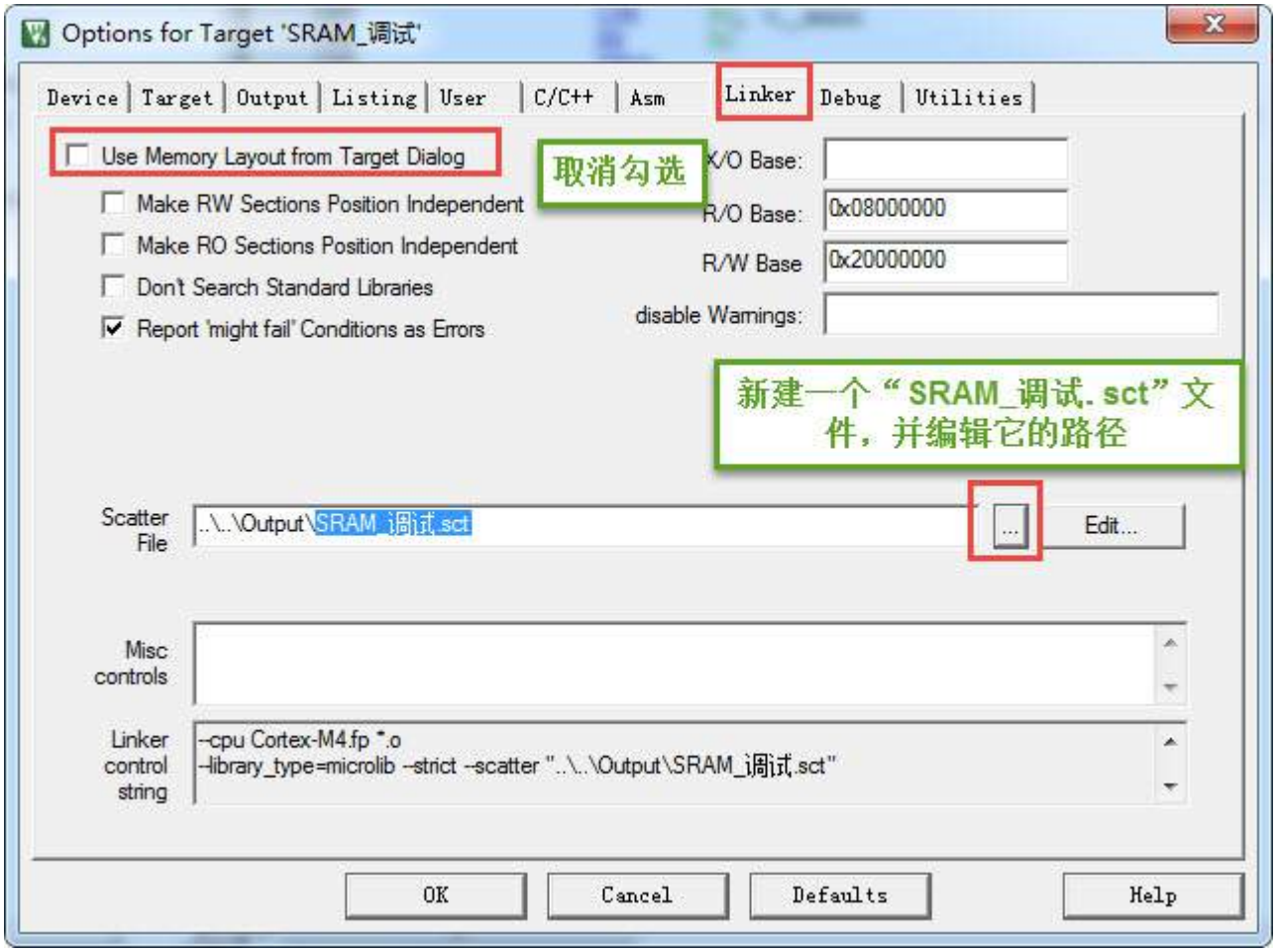


图 43-7 使用新建的“SRAM_ 调试.sct”文件

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 ***
; *************************************************************
```

(下页继续)

(续上页)

```
LR_IROM1 0x20000000 0x00010000  {    ; load region size_region
ER_IROM1 0x20000000 0x00010000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
}
RW_IRAM1 0x20010000 0x00010000  {  ; 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 64KB(0x00010000) 空间，而 RW data 空间改到了以地址 0x20010000 开始的 64KB 空间 (0x00010000)。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。

48.4.2.4 配置中断向量表

由于 startup_stm32f40xx.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_stm32f4xx.c 文件)

```
/**
* @brief  Setup the microcontroller system
*   Initialize the Embedded Flash Interface, the PLL and update the
*         SystemFrequency variable.
* @param  None
* @retval None
*/
void SystemInit(void)
{
/* .. 其它代码部分省略 */

/* Configure the Vector Table location add offset address ----*/
#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。

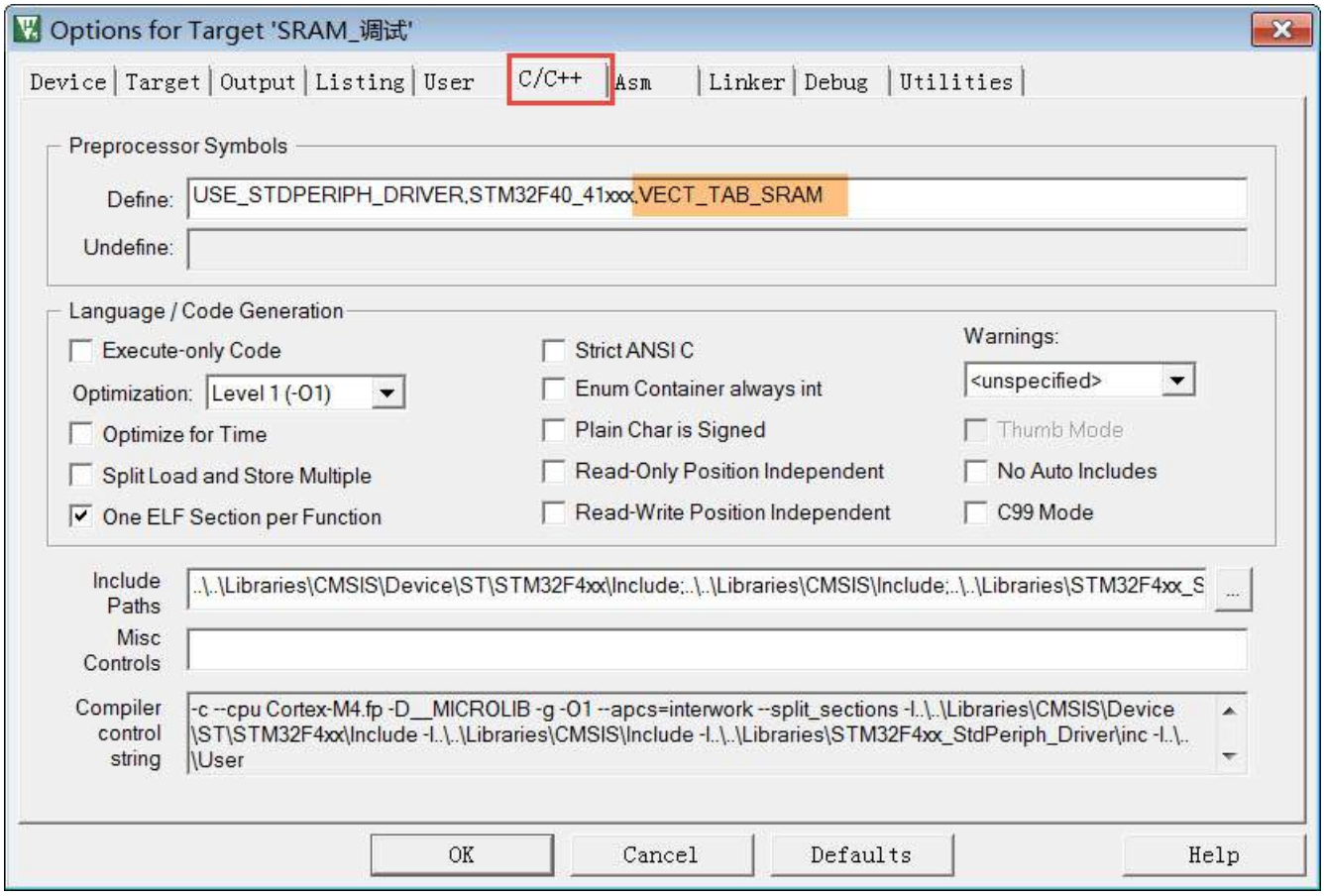


图 43-8 在 c/c++ 编译选项中加入宏 VECT_TAB_SRAM

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。

48.4.2.5 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。

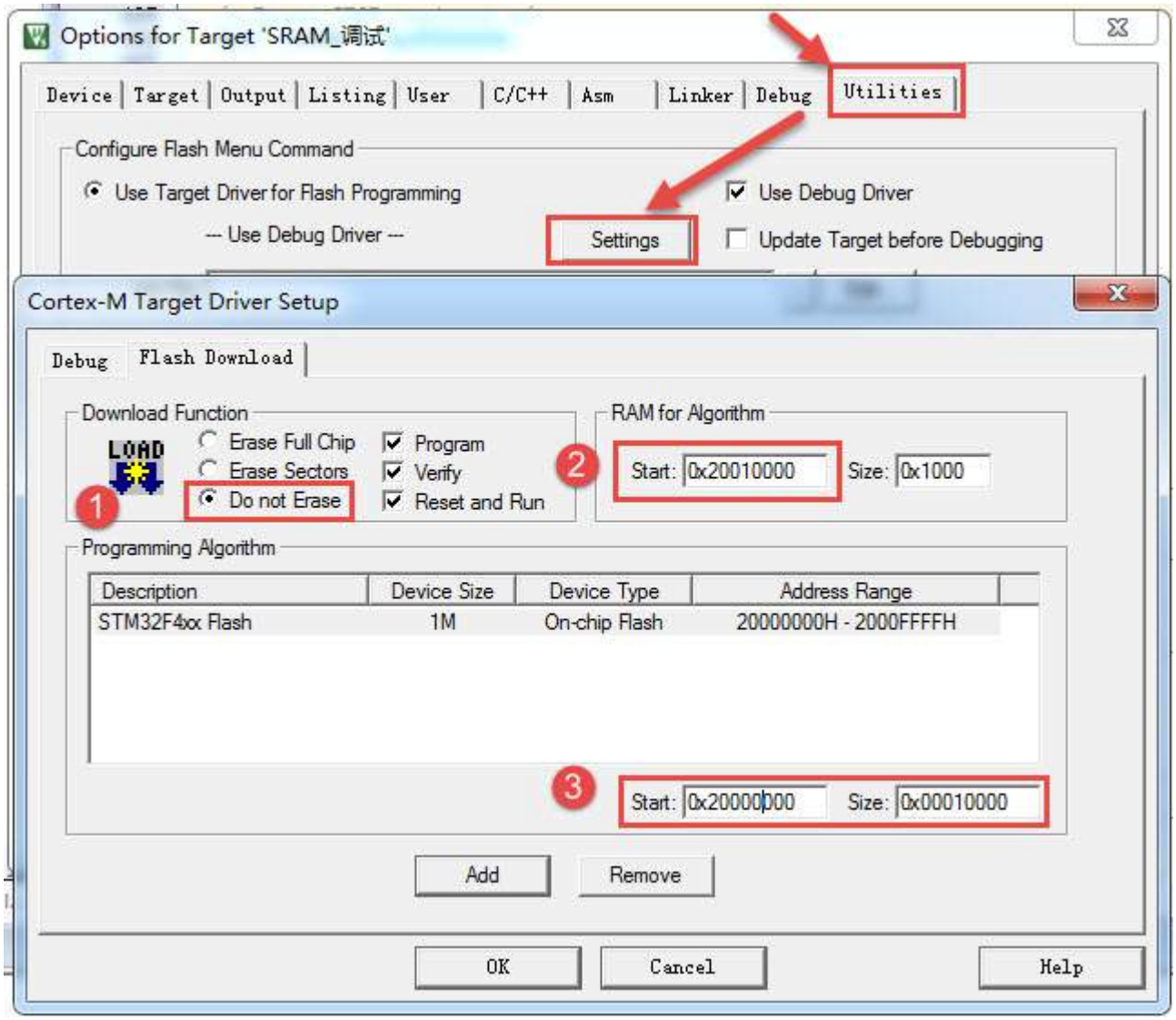


图 43-9 下载配置

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 64KB 实际为虚拟 ROM 空间，实际的 RAM 空间是从地址 0x20010000 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。复位后程序会正常运行 (前提是 BOOT0 和 BOOT 要被设置为 SRAM 启动) 。芯片掉电后这个存储在 SRAM 的程序会丢失，想恢复的话必须要重新下载程序。

48.4.2.6 仿真器的配置

上面的下载配置使得程序能够加载到 SRAM 中全速运行，但作为 SRAM 版本的程序，其功能更着重于调试，也就是说我们希望它能支持平时使用  按钮 (调试、debug) 时进行的硬件在线调试、单步运行等功能。

要实现调试功能，还要在“Options for Target->Debug->Settings”中进行配置，见图 *43_10*。

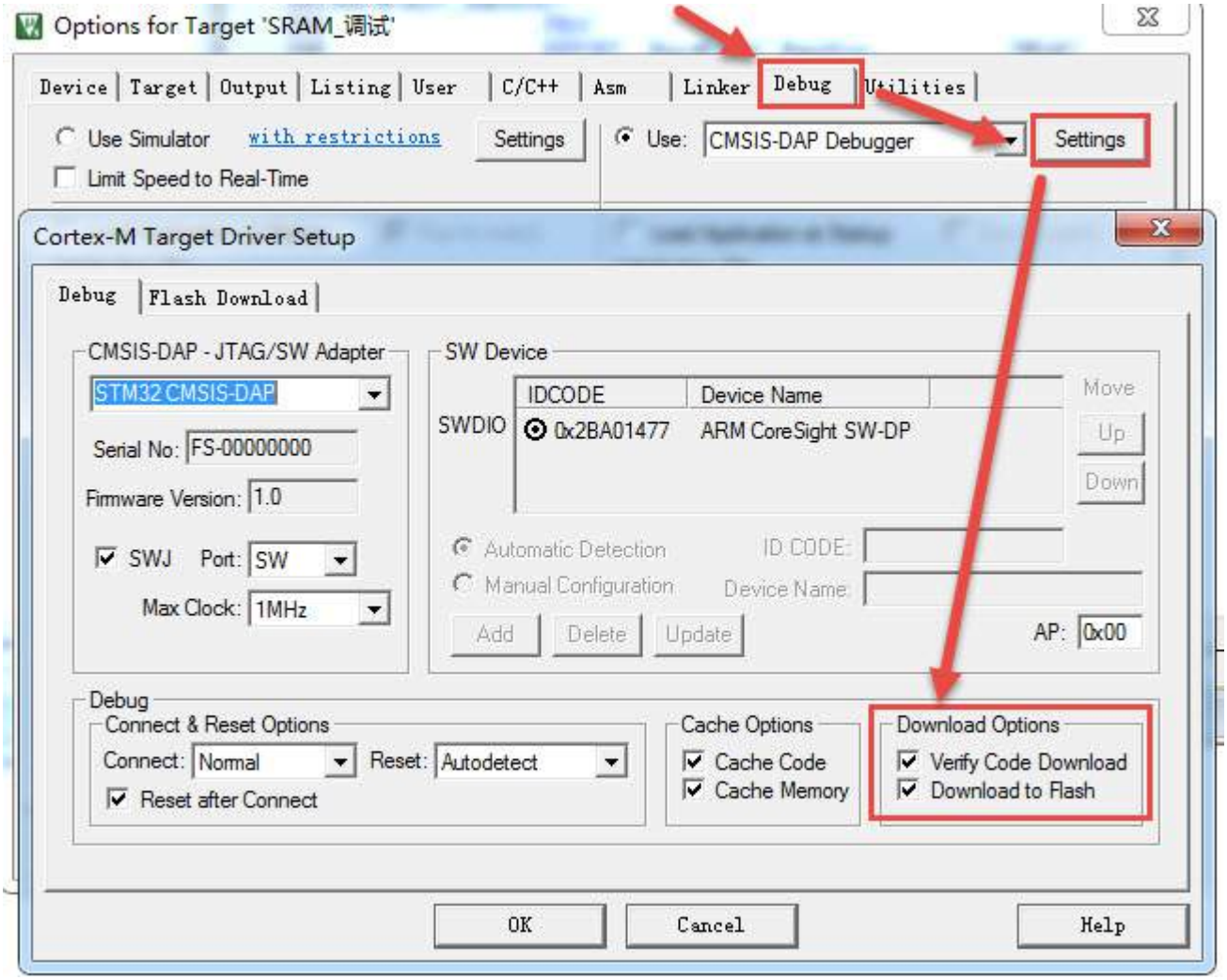


图 43-10 设置仿真前检查代码并下载程序到 FLASH 中

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。

经过这样的配置后，硬件仿真时与平时内部 FLASH 版本的程序无异，支持软件复位、单步运行、全速运行以及查看各种变量值等 (同样地，前提是 BOOT0 和 BOOT 要被设置为 SRAM 启动) 。

48.4.2.7 不需要修改 BOOT 引脚的仿真配置

假如您使用的硬件平台中 BOOT0 和 BOOT1 引脚电平已被固定，设置为内部 FLASH 启动，不方便改成 SRAM 方式，可以使用如下方法配置调试选项实现在 SRAM 调试：

(1) 与上述步骤一样，勾选“Verify Code Download”及“Download to FLASH”选项；

(2) 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；

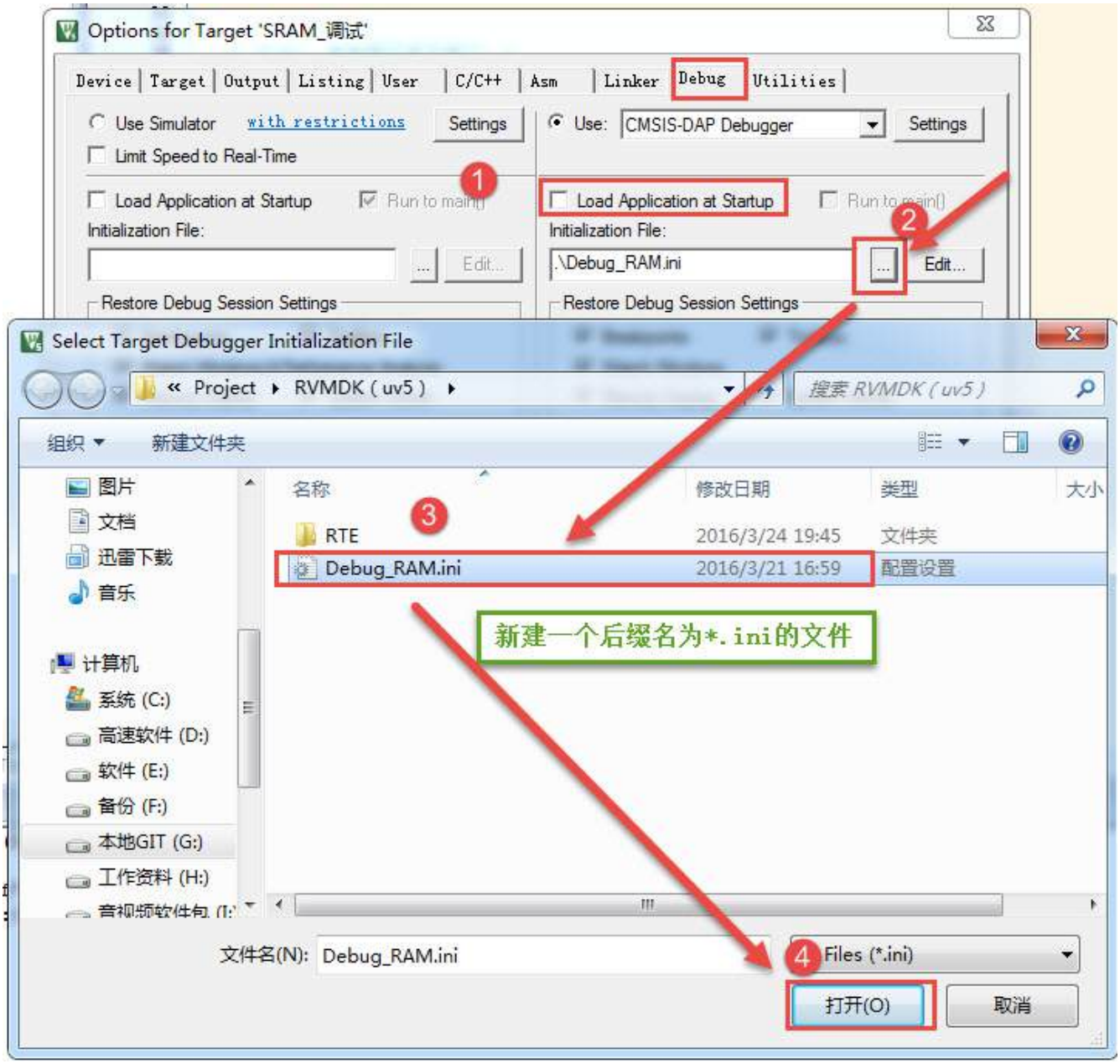


图 43-11 新建一个 ini 文件

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。

XPSR = 0x01000000;                          // 设置状态寄存器指针 xPSR
_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，这样的操作等效于从 SRAM 复位。

有了这样的配置，即使 BOOT0 和 BOOT1 引脚不设置为 SRAM 启动也能正常仿真了，但点击下载按钮把程序下载到 SRAM 然后按复位是不能全速运行的 (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 SRAM)。

上述 Debug_RAM.ini 文件是从 STM32F4 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。

第 49 章　读写内部 FLASH

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库说明文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

49.1 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*。

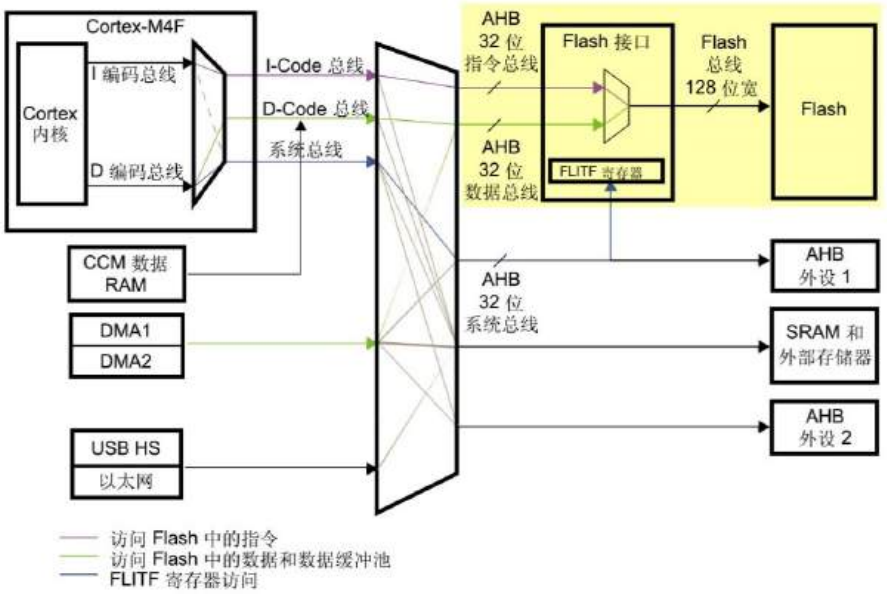


图 44-1 STM32 的内部框架图

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。

49.1.1 内部 FLASH 的构成

STM32 的内部 FLASH 包含主存储器、系统存储器、OTP 区域以及选项字节区域，它们的地址分布及大小见表 44-1

区域	块	名称	块地址	大小
主存储器	块1	扇区0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
		扇区1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		扇区2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
		扇区3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
		扇区4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
		扇区5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
		扇区6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		扇区7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
		扇区8	0x0808 0000 - 0x0809 FFFF	128 Kbytes
		扇区9	0x080A 0000 - 0x080B FFFF	128 Kbytes
		扇区10	0x080C 0000 - 0x080D FFFF	128 Kbytes
		扇区11	0x080E 0000 - 0x080F FFFF	128 Kbytes
	块2	扇区12	0x0810 0000 - 0x0810 3FFF	16 Kbytes
		扇区13	0x0810 4000 - 0x0810 7FFF	16 Kbytes
		扇区14	0x0810 8000 - 0x0810 BFFF	16 Kbytes
		扇区15	0x0810 C000 - 0x0810 FFFF	16 Kbyte
		扇区16	0x0811 0000 - 0x0811 FFFF	64 Kbytes
		扇区17	0x0812 0000 - 0x0813 FFFF	128 Kbytes
		扇区18	0x0814 0000 - 0x0815 FFFF	128 Kbytes
		扇区19	0x0816 0000 - 0x0817 FFFF	128 Kbytes
		扇区20	0x0818 0000 - 0x0819 FFFF	128 Kbytes
		扇区21	0x081A 0000 - 0x081B FFFF	128 Kbytes
		扇区22	0x081C 0000 - 0x081D FFFF	128 Kbytes
		扇区23	0x081E 0000 - 0x081F FFFF	128 Kbytes
系统存储区			0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP区域			0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
选项字节	块1		0x1FFF C000 - 0x1FFF C00F	16 bytes
	块2		0x1FFE C000 - 0x1FFE C00F	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 1M FLASH、2M FLASH 都是指这个区域的大小。主存储器分为两块，共 2MB，每块内分 12 个扇区，其中包含 4 个 16KB 扇区、1 个 64KB 扇区和 7 个 128KB 的扇区。如我们实验板中使用的 STM32F407ZGT6 型号芯片，它的主存储区域大小为 1MB，所以它只包含有表中的扇区 0-扇区 11。

与其它 FLASH 一样，在写入数据前，要先按扇区擦除，而有的时候我们希望能以小规格操纵存储单元，所以 STM32F42x/43x 针对 1MB FLASH 的产品还提供了一种双块的存储格式，见表 44-2。

1M字节单块存储器的扇区分配			1M字节双块存储器的扇区分配		
DB1M=0			DB1M=1		
主存储器	扇区号	扇区大小	主存储器	扇区号	扇区大小
1MB	扇区0	16 Kbytes	Bank 1 512KB	扇区0	16 Kbytes
	扇区1	16 Kbytes		扇区1	16 Kbytes
	扇区2	16 Kbytes		扇区2	16 Kbytes
	扇区3	16 Kbytes		扇区3	16 Kbytes
	扇区4	64 Kbytes		扇区4	64 Kbytes
	扇区5	128 Kbytes		扇区5	128 Kbytes
	扇区6	128 Kbytes		扇区6	128 Kbytes
	扇区7	128 Kbytes		扇区7	128 Kbytes
	扇区8	128 Kbytes	Bank 2 512KB	扇区12	16 Kbytes
	扇区9	128 Kbytes		扇区13	16 Kbytes
	扇区10	128 Kbytes		扇区14	16 Kbytes
	扇区11	128 Kbytes		扇区15	16 Kbytes
	-	-		扇区16	64 Kbytes
	-	-		扇区17	128 Kbytes
	-	-		扇区18	128 Kbytes
	-	-		扇区19	128 Kbytes

通过配置 FLASH 选项控制寄存器 FLASH_OPTCR 的 DB1M 位，可以切换这两种格式，切换成双块模式后，扇区 8-11 的空间被转移到扇区 12-19 中，扇区细分了，总容量不变。

要强调的是：本实验板采用的是 STM32F40x 系列的芯片，它没有双块存储格式，也不存在扇区 12-23，仅 STM32F42x/43x 系列产品才支持扇区 12-23。

- 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。

- OTP 区域

  OTP(One Time Program)，指的是只能写入一次的存储区域，容量为 512 字节，写入后数据就无法再更改，OTP 常用于存储应用程序的加密密钥。

- 选项字节

  选项字节用于配置 FLASH 的读写保护、电源管理中的 BOR 级别、软件/硬件看门狗等功能，这部分共 32 字节。可以通过修改 FLASH 的选项控制寄存器修改。

49.2 对内部 FLASH 的写入过程

49.2.1 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，芯片复位后默认会结 FLASH 上锁，这个时候不允许设置 FLASH 的控制寄存器，并且不能对修改 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 Flash 密钥寄存器 FLASH_KEYR 中写入 KEY1 = 0x45670123

(2) 再往 Flash 密钥寄存器 FLASH_KEYR 中写入 KEY2 = 0xCDEF89AB

49.2.2 数据操作位数

在内部 FLASH 进行擦除及写入操作时，电源电压会影响数据的最大操作位数，该电源电压可通过配置 FLASH_CR 寄存器中的 PSIZE 位改变，见下表

电压范围	2.7-3.6V(使用外部 Vpp)	2.7-3.6V	2.1–2.7V	1.8–2.1V
位数	64	32	16	8
PSIZE(1:0) 配置	11b	10b	01b	00b

最大操作位数会影响擦除和写入的速度，其中 64 位宽度的操作除了配置寄存器位外，还需要在 Vpp 引脚外加一个 8-9V 的电压源，且其供电时间不得超过一小时，否则 FLASH 可能损坏，所以 64 位宽度的操作一般是在量产时对 FLASH 写入应用程序时才使用，大部分应用场合都是用 32 位的宽度。

49.2.3 擦除扇区

在写入新的数据前，需要先擦除存储区域，STM32 提供了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“激活扇区擦除寄存器位 SER ”置 1，并设置“扇区编号寄存器位 SNB”，选择要擦除的扇区；

(3) 将 FLASH_CR 寄存器中的“开始擦除寄存器位 STRT ”置 1，开始擦除；

(4) 等待 BSY 位被清零时，表示擦除完成。

49.2.4 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) 针对所需存储器地址（主存储器块或 OTP 区域内）执行数据写入操作；

(4) 等待 BSY 位被清零时，表示写入完成。

49.3 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。

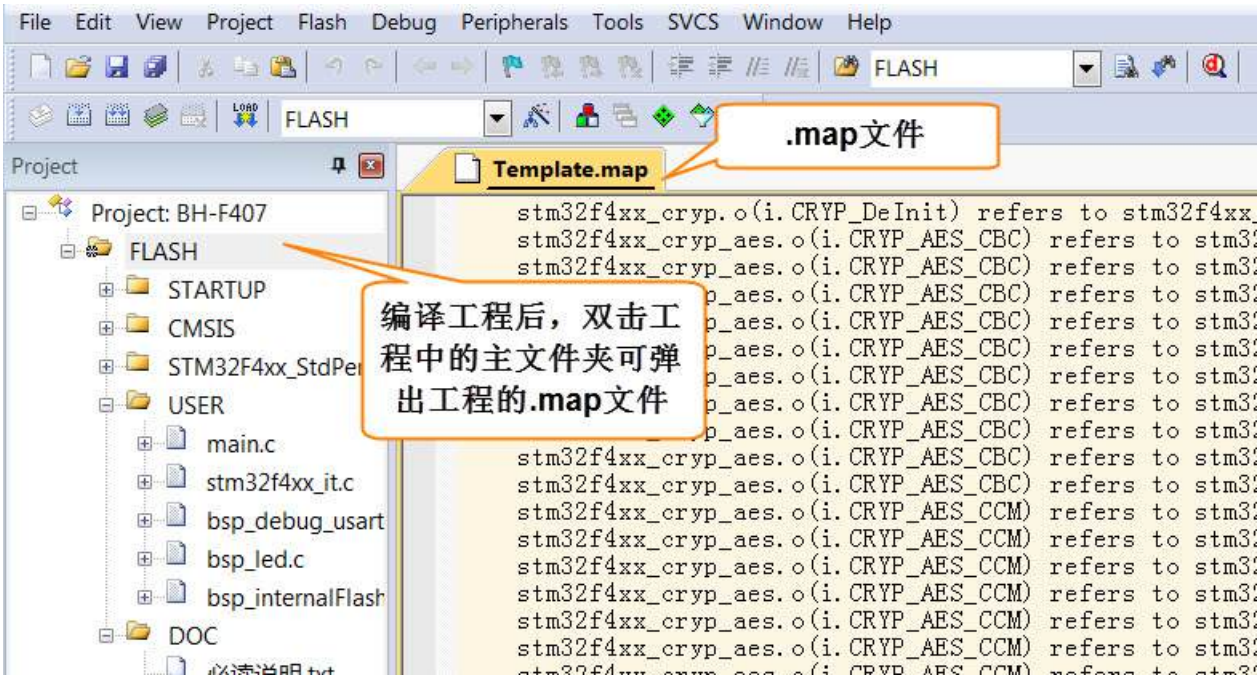


图 44-2 打开工程的.map 文件

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 : 0x08000189
/* 程序 ROM 加载空间 */
Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00000aa4, Max: 0x00100000,␣
↪ABSOLUTE)
/* 程序 ROM 执行空间 */
Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x00000a90, Max:␣
↪0x00100000, ABSOLUTE)
/* 地址分布列表 */
```

(下页继续)

(续上页)

```
Base Addr    Size         Type   Attr      Idx    E Section Name       ␣
 ↪Object

0x08000000   0x00000188   Data   RO            3    RESET              ␣
 ↪startup_stm32f40xx.o
0x08000188   0x00000000   Code   RO         4963  * .ARM.Collect$$$
 ↪$00000000  mc_w.l(entry.o)
0x08000188   0x00000004   Code   RO         5226    .ARM.Collect$$$
 ↪$00000001  mc_w.l(entry2.o)
0x0800018c   0x00000004   Code   RO         5229    .ARM.Collect$$$
 ↪$00000004  mc_w.l(entry5.o)
0x08000190   0x00000000   Code   RO         5231    .ARM.Collect$$$
 ↪$00000008  mc_w.l(entry7b.o)
0x08000190   0x00000000   Code   RO         5233    .ARM.Collect$$$
 ↪$0000000A  mc_w.l(entry8b.o)
0x08000190   0x00000008   Code   RO         5234    .ARM.Collect$$$
 ↪$0000000B  mc_w.l(entry9a.o)
0x08000198   0x00000000   Code   RO         5236    .ARM.Collect$$$
 ↪$0000000D  mc_w.l(entry10a.o)
/*... 此处省略大部分内容 */
0x08000902   0x00000002   PAD
0x08000904   0x00000010   Code   RO         4967    i.__0printf$bare    mc_
 ↪w.l(printfb.o)
0x08000914   0x0000000e   Code   RO         5268    i.__scatterload_copy ␣
 ↪mc_w.l(handlers.o)
0x08000922   0x00000002   Code   RO         5269    i.__scatterload_null ␣
 ↪mc_w.l(handlers.o)
0x08000924   0x0000000e   Code   RO         5270    i.__scatterload_
 ↪zeroinit  mc_w.l(handlers.o)
0x08000932   0x00000022   Code   RO         4974    i._printf_core      mc_
 ↪w.l(printfb.o)
0x08000954   0x00000024   Code   RO         4879    i.fputc             bsp_
 ↪debug_usart.o
```

(下页继续)

(续上页)

```
0x08000978   0x000000f8   Code   RO          4765    i.main                ␣
↪main.o
0x08000a70   0x00000020   Data   RO          5266    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。

49.3.1 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 0x00000aa4 及 0x00000a90，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 0x00100000，即 1M 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+0x00000aa4) 地址的空间区域。

49.3.2 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 0x08000a70，大小为 0x00000020，可知它占用的最高的地

址空间为 0x08000a90，跟执行区域的最高地址 0x00000a90 一样，但它们比加载区域说明中的最高地址 0x8000aa4 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 扇区地址分布表，可知仅使用扇区 0 就可以完全存储本应用程序，所以从扇区 1(地址 0x08004000) 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。

49.4 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。

49.4.1 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
* @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;
```

(下页继续)

(续上页)

```
    } 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。

49.4.2 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  Perform a mass erase or erase the specified FLASH memory sectors
* @param[in]  pEraseInit: pointer to an FLASH_EraseInitTypeDef structure␣
↪that
```

(下页继续)

(续上页)

```
*         contains the configuration information for the erasing.
*
* @param[out]  SectorError: pointer to variable  that
*contains the configuration information on faulty sector in case of error
*         (0xFFFFFFFF means that all the sectors have been correctly erased)
*
* @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*/
      *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
```

(下页继续)

(续上页)

```
            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*/
or (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) {
* In case of error, stop erase procedure and return the faulty sector*/
                  *SectorError = index;
                  break;
               }
            }
      }
```

(下页继续)

(续上页)

```
    }

    /* 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。

49.4.3 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

```
/**
   * @brief  Program byte, halfword, word or double word at a specified␣
↪address
   * @param  TypeProgram:  Indicate the way to program at a specified␣
↪address.
   *This parameter can be a value of @ref FLASH_Type_Program
   * @param  Address:  specifies the address to be programmed.
   * @param  Data: specifies the data to be programmed
```

(下页继续)

(续上页)

```
   *
   * @retval HAL_StatusTypeDef HAL Status
   */
HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t Address,␣
↪uint64_t Data)
{
   HAL_StatusTypeDef status = HAL_ERROR;

   /* Process Locked */
   __HAL_LOCK(&pFlash);

   /* Check the parameters */
   assert_param(IS_FLASH_TYPEPROGRAM(TypeProgram));

   /* Wait for last operation to be completed */
   status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

   if (status == HAL_OK) {
         switch (TypeProgram) {
         case FLASH_TYPEPROGRAM_BYTE : {
            /*Program byte (8-bit) at a specified address.*/
            FLASH_Program_Byte(Address, (uint8_t) Data);
            break;
         }

         case FLASH_TYPEPROGRAM_HALFWORD : {
            /*Program halfword (16-bit) at a specified address.*/
            FLASH_Program_HalfWord(Address, (uint16_t) Data);
            break;
         }

         case FLASH_TYPEPROGRAM_WORD : {
```

(下页继续)

(续上页)

```
            /*Program word (32-bit) at a specified address.*/
            FLASH_Program_Word(Address, (uint32_t) Data);
            break;
          }

          case FLASH_TYPEPROGRAM_DOUBLEWORD : {
            /*Program double word (64-bit) at a specified address.*/
            FLASH_Program_DoubleWord(Address, Data);
            break;
          }
          default :
            break;
          }
          /* Wait for last operation to be completed */
        status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

          /* If the program operation is completed, disable the PG Bit */
          FLASH->CR &= (~FLASH_CR_PG);
    }

    /* 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。

49.5 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。

49.5.1 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。

49.5.2 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。

49.5.2.1 程序设计要点

(1) 对内部 FLASH 解锁；

(2) 找出空闲扇区，擦除目标扇区；

(3) 进行读写测试。

49.5.2.2 代码分析

49.5.2.2.1 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)

```
/* Base address of the Flash sectors */
#define ADDR_FLASH_SECTOR_0     ((uint32_t)0x08000000) /* Base address
                                 of Sector 0, 16 Kbytes   */
#define ADDR_FLASH_SECTOR_1     ((uint32_t)0x08004000) /* Base address
                                 of Sector 1, 16 Kbytes   */
#define ADDR_FLASH_SECTOR_2     ((uint32_t)0x08008000) /* Base address
                                 of Sector 2, 16 Kbytes   */
#define ADDR_FLASH_SECTOR_3     ((uint32_t)0x0800C000) /* Base address
                                 of Sector 3, 16 Kbytes   */
#define ADDR_FLASH_SECTOR_4     ((uint32_t)0x08010000) /* Base address
                                 of Sector 4, 64 Kbytes   */
#define ADDR_FLASH_SECTOR_5     ((uint32_t)0x08020000) /* Base address
                                 of Sector 5, 128 Kbytes  */
#define ADDR_FLASH_SECTOR_6     ((uint32_t)0x08040000) /* Base address
                                 of Sector 6, 128 Kbytes  */
#define ADDR_FLASH_SECTOR_7     ((uint32_t)0x08060000) /* Base address
                                 of Sector 7, 128 Kbytes  */
```

这些宏跟表 44-1 中的地址说明一致。

49.5.2.2.2 根据扇区地址计算 SNB 寄存器的值

在擦除操作时，需要向 FLASH 控制寄存器 FLASH_CR 的 SNB 位写入要擦除的扇区号，固件库把各个扇区对应的寄存器值使用宏定义到了 stm32f4xx_hal_flash.h 文件。为了便于使用，我们自定义了一个 GetSector 函数，根据输入的内部 FLASH 地址，找出其所在的扇区，并返回该扇区对应的 SNB 位寄存器值，见*代码清单 44_0_6*。

列表 6: 写入到 SNB 寄存器位的值(stm32f4xx_hal_flash.h 及 bsp_internalFlash.c 文件)

```
/** 固件库定义的用于扇区写入到 SNB 寄存器位的宏 (stm32f4xx_hal_flash.h 文件)
   * @{
   */
#define FLASH_SECTOR_0           ((uint32_t)0U) /*!< Sector Number 0   */
#define FLASH_SECTOR_1           ((uint32_t)1U) /*!< Sector Number 1   */
#define FLASH_SECTOR_2           ((uint32_t)2U) /*!< Sector Number 2   */
#define FLASH_SECTOR_3           ((uint32_t)3U) /*!< Sector Number 3   */
#define FLASH_SECTOR_4           ((uint32_t)4U) /*!< Sector Number 4   */
#define FLASH_SECTOR_5           ((uint32_t)5U) /*!< Sector Number 5   */
#define FLASH_SECTOR_6           ((uint32_t)6U) /*!< Sector Number 6   */
#define FLASH_SECTOR_7           ((uint32_t)7U) /*!< Sector Number 7   */
/**
   * @brief  根据输入的地址给出它所在的 sector
   *         例如:
           uwStartSector = GetSector(FLASH_USER_START_ADDR);
           uwEndSector = GetSector(FLASH_USER_END_ADDR);
   * @param  Address: 地址
   * @retval 地址所在的 sector
   */
static uint32_t GetSector(uint32_t Address)
{
   uint32_t sector = 0;

if ((Address < ADDR_FLASH_SECTOR_1) && (Address >= ADDR_FLASH_SECTOR_0)) {
         sector = FLASH_SECTOR_0;
} else if((Address < ADDR_FLASH_SECTOR_2) &&
(Address >= ADDR_FLASH_SECTOR_1)) {
         sector = FLASH_SECTOR_1;
   } else if ((Address < ADDR_FLASH_SECTOR_3) &&
```

(下页继续)

(续上页)

```
(Address >= ADDR_FLASH_SECTOR_2)) {
          sector = FLASH_SECTOR_2;
   } else if ((Address < ADDR_FLASH_SECTOR_4) &&
(Address >= ADDR_FLASH_SECTOR_3)) {
          sector = FLASH_SECTOR_3;
   } else if ((Address < ADDR_FLASH_SECTOR_5) &&
(Address >= ADDR_FLASH_SECTOR_4)) {
          sector = FLASH_SECTOR_4;
   } else if ((Address < ADDR_FLASH_SECTOR_6) &&
(Address >= ADDR_FLASH_SECTOR_5)) {
          sector = FLASH_SECTOR_5;
   } else if ((Address < ADDR_FLASH_SECTOR_7) &&
(Address >= ADDR_FLASH_SECTOR_6)) {
          sector = FLASH_SECTOR_6;
   } else { /*(Address < FLASH_END_ADDR) &&
(Address >= ADDR_FLASH_SECTOR_23))*/
          sector = FLASH_SECTOR_7;
   }
   return sector;
}
```

代码中固件库定义的宏 FLASH_Sector_0-7 对应的值是跟寄存器说明一致的，见图 *44_3*

Bits 7:3 **SNB[4:0]:** Sector number
if nDBANK=1 in single bank mode These bits select the sector to erase.
00000 sector 0
00001 sector 1
...
01011 sector 11
Others not allowed

if nDBANK=0 in dual bank mode These bits select the sector to erase from bank 1 or bank 2, where MSB bit selects the bank
00000 bank 1 sector 0
00001 bank 1 sector 1
...
01011 bank 1 sector 11
01100: not allowed
01101: not allowed
01110: not allowed
01111: not allowed

10000 bank 2 sector 0
10001 bank 2 sector 1
...
11011 bank 2 sector 11
11100: not allowed
11101: not allowed
11110: not allowed
11111: not allowed

GetSector 函数根据输入的地址与各个扇区的基地址进行比较，找出它所在的扇区，并使用 FLASH_EraseInitTypeDef 擦除 flash 初始化结构体，最终计算出 NbSectors（扇区个数）。

49.5.2.2.3 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 7: 代码清单 44-6 对内部地 FLASH 进行读写测试
(bsp_internal_Flash.c 文件)

```
/* 准备写入的测试数据 */
#define DATA_32                 ((uint32_t)0x87645321)
```

(下页继续)

(续上页)

```
/* Exported types ------------------------------------------------------------*/
/* Exported constants --------------------------------------------------------*/
/* 要擦除内部 FLASH 的起始地址 */
#define FLASH_USER_START_ADDR   ADDR_FLASH_SECTOR_5
/* 要擦除内部 FLASH 的结束地址 */
#define FLASH_USER_END_ADDR     ADDR_FLASH_SECTOR_7

static uint32_t GetSector(uint32_t Address);

/**
* @brief  InternalFlash_Test, 对内部 FLASH 进行读写测试
* @param  None
* @retval None
*/
int InternalFlash_Test(void)
{
   /* 要擦除的起始扇区 (包含) 及结束扇区 (不包含), 如 8-12, 表示擦除 8、9、10、11 扇区
*/
   uint32_t FirstSector = 0;
   uint32_t NbOfSectors = 0;

   uint32_t SECTORError = 0;

   uint32_t Address = 0;

   __IO uint32_t Data32 = 0;
   __IO uint32_t MemoryProgramStatus = 0;
   static FLASH_EraseInitTypeDef EraseInitStruct;

   /* FLASH 解锁 ********************************/
   /* 使能访问 FLASH 控制寄存器 */
```

(下页继续)

(续上页)

```
    HAL_FLASH_Unlock();

    FirstSector = GetSector(FLASH_USER_START_ADDR);
    NbOfSectors = GetSector(FLASH_USER_END_ADDR)- FirstSector + 1;

    /* 擦除用户区域 (用户区域指程序本身没有使用的空间，可以自定义)**/
    /* Fill EraseInit structure*/
    EraseInitStruct.TypeErase     = FLASH_TYPEERASE_SECTORS;
/* 以“字”的大小进行操作 */
    EraseInitStruct.VoltageRange  = FLASH_VOLTAGE_RANGE_3;
    EraseInitStruct.Sector        = FirstSector;
    EraseInitStruct.NbSectors     = NbOfSectors;
    /* 开始擦除操作 */
    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;
    }

    /* 以“字”的大小为单位写入数据 ***********************************/
    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;
        } else {
            /* 写入出错，返回，实际应用中可加入处理 */
            return -1;
        }
    }

    /* 给 FLASH 上锁，防止内容被篡改 */
```

(下页继续)

(续上页)

```
    HAL_FLASH_Lock();

    /* 从 FLASH 中读取出数据进行校验 ****************************************/
    /*  MemoryProgramStatus = 0: 写入的数据正确
        MemoryProgramStatus != 0: 写入的数据错误，其值为错误的个数 */
    Address = FLASH_USER_START_ADDR;
    MemoryProgramStatus = 0;

    while (Address < FLASH_USER_END_ADDR) {
        Data32 = *(__IO uint32_t*)Address;

        if (Data32 != DATA_32) {
              MemoryProgramStatus++;
        }

        Address = Address + 4;
    }
    /* 数据校验不正确 */
    if (MemoryProgramStatus) {
        return -1;
    } else { /* 数据校验正确 */
        return 0;
    }
}
```

该函数的执行过程如下：

(1) 调用 HAL_FLASH_Unlock 解锁；

(2) 调用 GetSector 根据起始地址及结束地址计算要擦除的扇区；

(3) 配置 FLASH_EraseInitTypeDef 擦除 flash 初始化结构体；

(4) 调用 HAL_FLASHEx_Erase 擦除扇区，擦除时按字为单位进行操作；

(5) 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数值“DATA_32”;

(6) 调用 HAL_FLASH_Lock 上锁;

(7) 使用指针读取数据内容并校验。

49.5.2.2.4 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 8: 代码清单 44-7 main 函数 (main.c 文件)

```
int main(void)
{
   /* 配置系统时钟为 168 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();

   printf("\r\n 欢迎使用野火  STM32 F407 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()==0) {
         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) {
   }
```

(下页继续)

(续上页)

```
}
```

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。

49.5.3 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 50 章　设置 FLASH 的读写保护及解除

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》以及《Proprietary code read-out protection on microcontrollers》。

50.1 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。

50.1.1 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	[63:16]	[15:0]
0x1FFFC000	保留	ROP 和用户选项字节 (RDP&USER)
0x1FFFC008	保留	扇区 0 到 11 的写保护 nWRP 位
0x1FFEC000	保留	保留
0x1FFEC008	保留	扇区 12 到 23 的写保护 nWRP 位

选项字节具体的数据位配置说明见表 45-2。

选项字节（字，地址 0x1FFF C000）	
RDP：读保护选项字节。	
读保护用于保护 Flash 中存储的软件代码。	
位 15:8	0xAA：级别 0，无保护 其它值：级别 1，存储器读保护（调试功能受限） 0xCC：级别 2，芯片保护（禁止调试和从 RAM 启动）
USER：用户选项字节	
此字节用于配置以下功能：	
选择看门狗事件：硬件或软件	
进入停止模式时产生复位事件	
进入待机模式时产生复位事件	
位 7	nRST_STDBY 0：进入待机模式时产生复位 1：不产生复位
位 6	nRST_STOP 0：进入停止模式时产生复位 1：不产生复位
位 5	WDG_SW 0：硬件看门狗 1：软件看门狗
位 4	0x1：未使用
位 3:2	BOR_LEV：BOR 复位级别 这些位包含释放复位信号所需达到的供电电压阈值。 通过对这些位执行写操作，可将新的 BOR 级别值编程到 Flash。 00：BOR 级别 3 (VBOR3)，复位阈值电压为 2.70 V 到 3.60 V 01：BOR 级别 2 (VBOR2)，复位阈值电压为 2.40 V 到 2.70 V 10：BOR 级别 1 (VBOR1)，复位阈值电压为 2.10 V 到 2.40 V 11：BOR 关闭 (VBOR0)，复位阈值电压为 1.8 V 到 2.10 V
位 1:0	0x1：未使用
选项字节（字，地址 0x1FFF C008）	
位15	SPMOD：选择nWPRi位的模式 0：nWPRi位用于写保护(默认) 1：nWPRi位用于代码读出保护(Proprietary code readout protection ，PCROP)
位14	DB1M：设置内部FLASH为1MB的产品的双Bank模式 0：单个Bank的模式 1：使用两个Bank的模式
位 13:12	0xF：未使用
nWRP：Flash 写保护选项字节。	
扇区 0 到 11 可采用写保护。	
位 11:0	nWRPi (i值为0-11，对应0-11扇区的保护设置)： 0：开启所选扇区的写保护 1：关闭所选扇区的写保护
选项字节（字，地址 0x1FFE C000）	
位 15:0	0xFF：未使用
选项字节（字，地址 0x1FFE C008）	
位 15:12	0xF：未使用
nWRP：Flash 写保护选项字节。	
扇区 12 到 23 可采用写保护。	
位 11:0	nWRPi： 0：开启扇区 i 的写保护。 1：关闭扇区 i 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。

50.1.2 RDP 读保护级别

修改选项字节的 RDP 位的值可设置内部 FLASH 为以下保护级别：

- 0xAA：级别 0，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 及“备份 SRAM”的内容都没有任何限制。(注意这里说的“备份 SRAM”是指 STM32 备份域的 SRAM 空间，不是指主 SRAM，下同)

- 其它值：级别 1，使能读保护

  把 RDP 配置成除 0xAA 或 0xCC 外的任意数值，都会使能级别 1 的读保护。在这种保护下，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 及备份 SRAM 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 及备份 SRAM 的任意访问。

  也就是说，在级别 1 模式下，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，若该程序读取内部 FLASH，会被禁止。而如果是芯片自己访问内部 FLASH，是完全没有问题的，例如前面的“读写内部 FLASH”实验中的代码自己擦写内部 FLASH 空间的内容，即使处于级别 1 的读保护，也能正常擦写。

  当芯片处于级别 1 的时候，可以把选项字节的 RDP 位重新设置为 0xAA，恢复级别 0。在恢复到级别 0 前，芯片会自动擦除内部 FLASH 及备份 SRAM 的内容，即降级后原内部 FLASH 的代码会丢失。在级别 1 时使用 SRAM 自举的程序也可以访问选项字节进行修改，所以如果原内部 FLASH 的代码没有解除读保护的操作时，可以给它加载一个 SRAM 自举的程序进行保护降级，后面我们将会进行这样的实验。

- 0xCC：级别 2，禁止调试

  把 RDP 配置成 0xCC 值时，会进入最高级别的读保护，且设置后无法再降级，它会永久禁止用于调试的 JTAG 接口 (相当于熔断)。在该级别中，除了具有级别 1 的所有保护功能外，进一步禁止了从 SRAM 或系统存储器的自举 (即平时使用的串口 ISP 下载

功能也失效)，JTAG 调试相关的功能被禁止，选项字节也不能被修改。它仅支持从内部 FLASH 自举时对内部 FLASH 及 SRAM 的访问 (读写、擦除)。

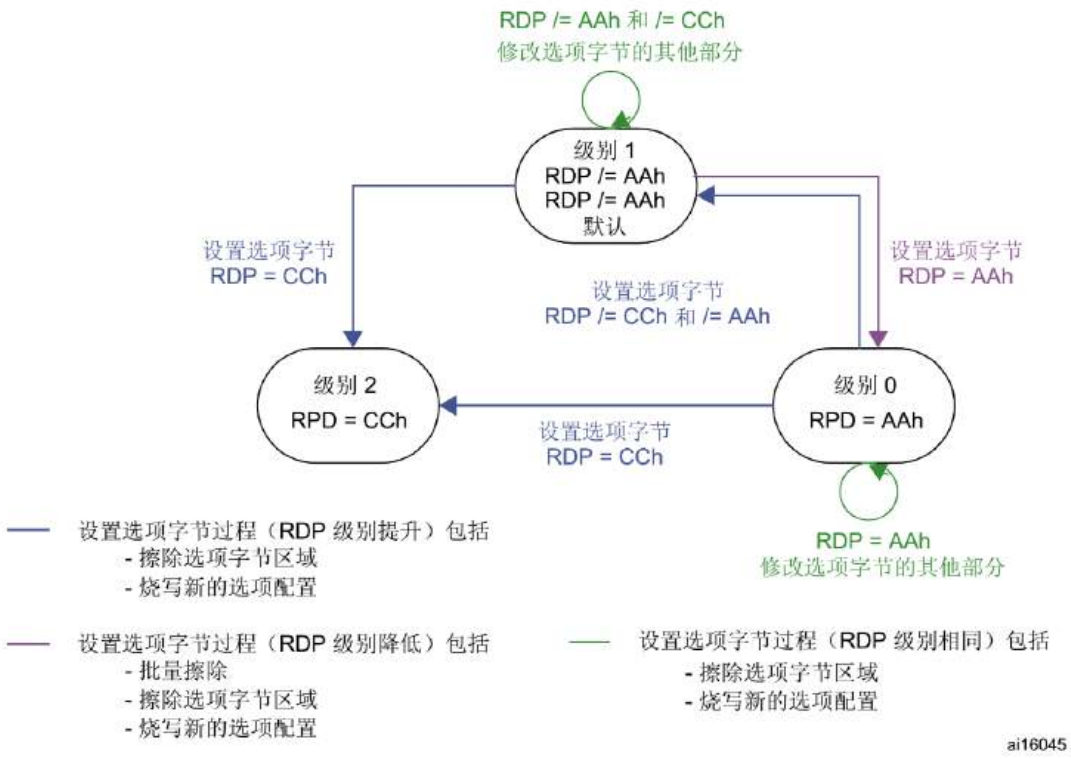
由于设置了级别 2 后无法降级，也无法通过 JTAG、串口 ISP 等方式更新程序，所以使用这个级别的保护时一般会在程序中预留“后门”以更新应用程序，若程序中没有预留后门，芯片就无法再更新应用程序了。所谓的“后门”是一种 IAP 程序 (In Application Program)，它通过某个通讯接口获取将要更新的程序内容，然后利用内部 FLASH 擦写操作把这些内容烧录到自己的内部 FLASH 中，实现应用程序的更新。

不同级别下的访问限制见图 *45_2*。

存储区	保护级别	调试功能，从 RAM 或系统存储器自举			从 Flash 自举		
		读	写	擦除	读	写	擦除
主 Flash 和备份 SRAM	级别 1	否		否(1)	是		
	级别 2	否			是		
选项字节	级别 1	是			是		
	级别 2	否			否		
OTP	级别 1	否		NA	是		NA
	级别 2	否		NA	是		NA

1. 只有在 RDP 从级别 1 更改为级别 0 时，才会擦除主 Flash 和备份 SRAM。OTP 区域保持不变。

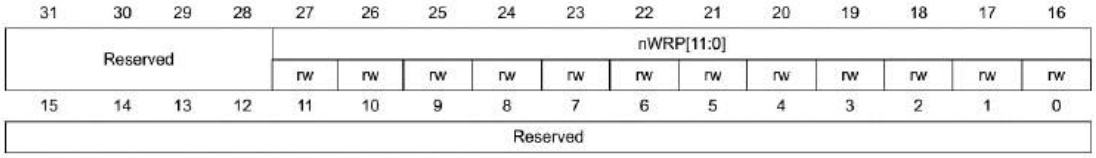
不同保护级别之间的状态转换见图 *45_3*



50.2 修改选项字节的过程

修改选项字节的内容可修改各种配置，但是，当应用程序运行时，无法直接通过选项字节的地址改写它们的内容，例如，接使用指针操作地址 0x1FFFC0000 的修改是无效的。要改写其内容时必须设置寄存器 FLASH_OPTCR 及 FLASH_OPTCR1 中的对应数据位，寄存器的与选项字节对应位置见图 50 3 及图 50 4，详细说明请查阅《STM32 参考手册》。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SPR MOD	DB1M	Reserved		nWRP[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RDP[7:0]								nRST_ STDBY	nRST_ STOP	WDG_ SW	BFB2	BOR_LEV		OPTST RT	OPTLO CK
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rs	rs



默认情况下，FLASH_OPTCR 寄存器中的第 0 位 OPTLOCK 值为 1，它表示选项字节被上锁，需要解锁后才能进行修改，当寄存器的值设置完成后，对 FLASH_OPTCR 寄存器中的第 1 位 OPTSTRT 值设置为 1，硬件就会擦除选项字节扇区的内容，并把 FLASH_OPTCR/1 寄存器中包含的值写入到选项字节。

所以，修改选项字节的配置步骤如下：

(1) 解锁，在 Flash 选项密钥寄存器 (FLASH_OPTKEYR) 中写入 OPTKEY1 = 0x0819 2A3B；接着在 Flash 选项密钥寄存器 (FLASH_OPTKEYR) 中写入 OPTKEY2 = 0x4C5D 6E7F。

(2) 检查 FLASH_SR 寄存器中的 BSY 位，以确认当前未执行其它 Flash 操作。

(3) 在 FLASH_OPTCR 和/或 FLASH_OPTCR1 寄存器中写入选项字节值。

(4) 将 FLASH_OPTCR 寄存器中的选项启动位 (OPTSTRT) 置 1。

(5) 等待 BSY 位清零，即写入完成。

50.3 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。

50.3.1 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 */
   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。

50.3.2 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 */
   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*/
```

(下页继续)

(续上页)

```
            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,
                                              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 */
    }
```

(下页继续)

(续上页)

```
    /* 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。

50.3.3 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。

50.4 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，更改读保护级别、设置 PCROP 或写保护，最后把选项字节恢复默认值。

本实验要进行的操作比较特殊，在开发和调试的过程中都是在 SRAM 上进行的（使用 SRAM 启动方式）。例如，直接使用 FLASH 版本的程序进行调试时，如果该程序在运行后对扇区进行了写保护而没有解除的操作或者该解除操作不正常，此时将无法再给芯片的内部 FLASH 下载新程序，最终还是要使用 SRAM 自举的方式进行解除操作。所以在本实验中为便于修改选项字节的参数，我们统一使用 SRAM 版本的程序进行开发和学习，当 SRAM 版本调试正常后再改为 FLASH 版本。

关于在 SRAM 中调试代码的相关配置，请参考前面的章节。

注意：

若您在学习的过程中想亲自修改代码进行测试，请注意备份原工程代码。当芯片的 FLASH 被保护导致无法下载程序到 FLASH 时，可以下载本工程到芯片，并使用 SRAM 启动运行，即可恢复芯片至默认配置。但如果修改了读保护为级别 2，采用任何方法都无法恢复！(除了这个配置，其它选项都可以大胆地修改测试。)

50.4.1 硬件设计

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。

50.4.2 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，它是从“RAM 调试—多彩流水灯”工程改写而来的。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“internalFlash_reset.c”及“internalFlash_reset.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。

50.4.2.1 主要实验

(1) 学习配置扇区写保护；

(2) 学习配置读保护级别；

(3) 学习如何恢复选项字节到默认配置；

50.4.2.2 代码分析

50.4.2.2.1 设置写保护及解除

我们先以*代码清单 45_3* 中的设置与解除写保护过程来学习如何配置选项字节。

列表 4: 代码清单 45-3 配

```
#define FLASH_WRP_SECTORS    (OB_WRP_SECTOR_0|OB_WRP_SECTOR_1)
__IO uint32_t SectorsWRPStatus = 0xFFF;

/**
* @brief  WriteProtect_Test, 普通的写保护配置
* @param  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
* @retval None
*/
void WriteProtect_Test(void)
{
   /* 获取扇区的写保护状态 */
   HAL_FLASHEx_OBGetConfig(&OBInit);
   SectorsWRPStatus = OBInit.WRPSector & FLASH_WRP_SECTORS;

   if (SectorsWRPStatus == 0x00) {
      /* 扇区已被写保护，执行解保护过程 */

      /* 使能访问 OPTCR 寄存器 */
      HAL_FLASH_OB_Unlock();

      HAL_FLASH_Unlock();
      /* 设置对应的 nWRP 位，解除写保护 */
      OBInit.OptionType = OPTIONBYTE_WRP;
      OBInit.WRPState   = OB_WRPSTATE_DISABLE;
      OBInit.WRPSector  = FLASH_WRP_SECTORS;
```

(下页继续)

(续上页)

```
    HAL_FLASHEx_OBProgram(&OBInit);
    /* 开始对选项字节进行编程 */
    if (HAL_FLASH_OB_Launch() != HAL_OK) {
          FLASH_ERROR("对选项字节编程出错，解除写保护失败");
          while (1) {
          }
    }
    /* 禁止访问 OPTCR 寄存器 */
    HAL_FLASH_OB_Lock();
    HAL_FLASH_Lock();
    /* 获取扇区的写保护状态 */
    HAL_FLASHEx_OBGetConfig(&OBInit);
    SectorsWRPStatus = OBInit.WRPSector & FLASH_WRP_SECTORS;

    /* 检查是否配置成功 */
    if (SectorsWRPStatus == FLASH_WRP_SECTORS) {
          FLASH_INFO("解除写保护成功！");
    } else {
          FLASH_ERROR("未解除写保护！");
    }
} else {
    /* 若扇区未被写保护，开启写保护配置 */

    /* 使能访问 OPTCR 寄存器 */
    HAL_FLASH_OB_Unlock();

    HAL_FLASH_Unlock();
    /* 使能 FLASH_WRP_SECTORS 扇区写保护 */
    OBInit.OptionType = OPTIONBYTE_WRP;
    OBInit.WRPState   = OB_WRPSTATE_ENABLE;
       OBInit.WRPSector  = FLASH_WRP_SECTORS;
       HAL_FLASHEx_OBProgram(&OBInit);
```

(下页继续)

(续上页)

```
        /* 开始对选项字节进行编程 */
        if (HAL_FLASH_OB_Launch() != HAL_OK) {
            FLASH_ERROR("对选项字节编程出错，解除写保护失败");
            while (1) {
            }
        }

        /* 禁止访问 OPTCR 寄存器 */
        HAL_FLASH_OB_Lock();

        HAL_FLASH_Lock();

        /* 获取扇区的写保护状态 */
        HAL_FLASHEx_OBGetConfig(&OBInit);
        SectorsWRPStatus = OBInit.WRPSector & FLASH_WRP_SECTORS;

        /* 检查是否配置成功 */
        if (SectorsWRPStatus == 0x00) {
            FLASH_INFO("设置写保护成功！");
        } else {
            FLASH_ERROR("设置写保护失败！");
        }
    }
}
```

本函数分成了两个部分，它根据目标扇区的状态进行操作，若原来扇区为非保护状态时就进行写保护，若为保护状态就解除保护。其主要操作过程如下：

1、调用 HAL_FLASHEx_OBGetConfig 函数获取目标扇区的保护状态若扇区被写保护，则开始解除保护过程，否则开始设置写保护过程；

2、调用 HAL_FLASH_OB_Unlock 解锁选项字节的编程；

3、调用 HAL_FLASHEx_OBProgram 函数配置目标扇区关闭或打开写保护；

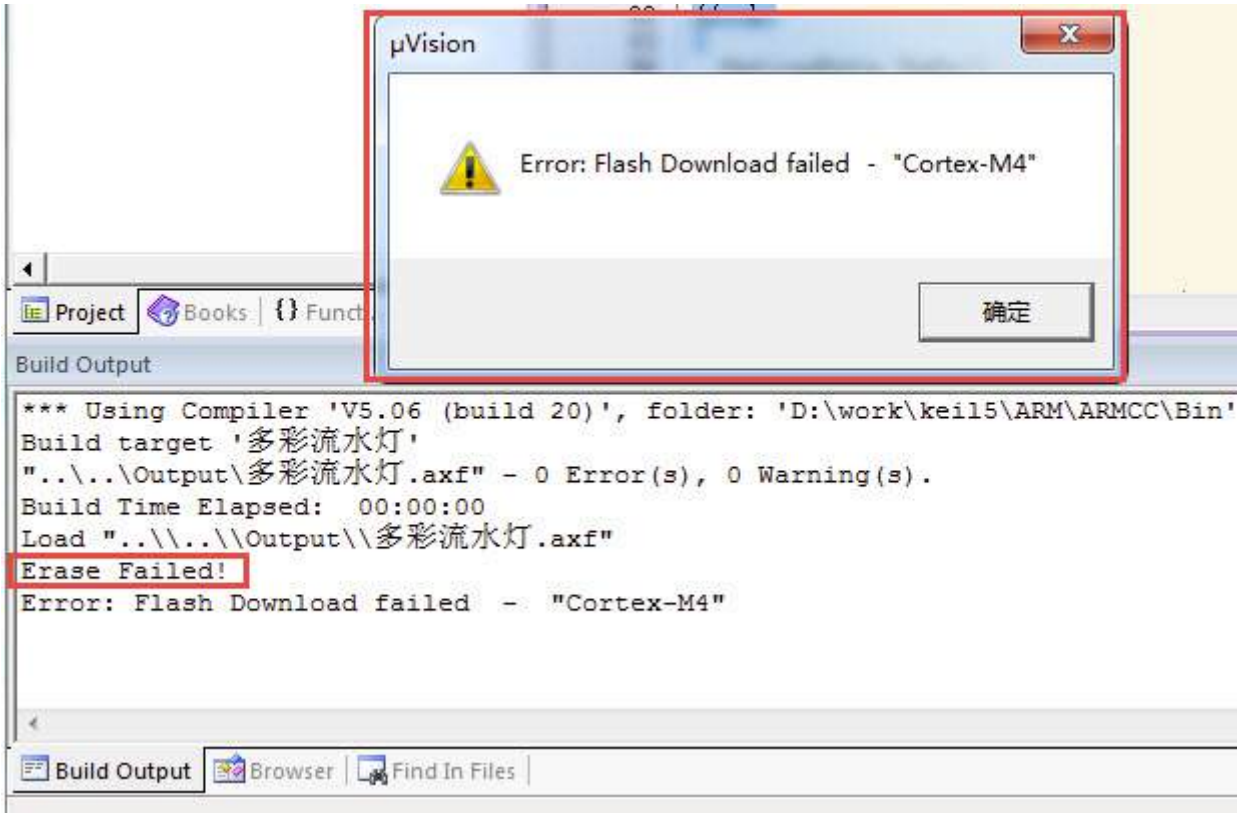
4、调用 HAL_FLASH_OB_Launch 函数把寄存器的配置写入到选项字节；

5、调用 HAL_FLASHEx_OBGetConfig 函数检查是否配置成功；

6、调用 HAL_FLASH_OB_Lock 禁止修改选项字节。

50.4.2.2.2 恢复选项字节为默认值

当芯片被设置为读写保护或 PCROP 保护时，这时给芯片的内部 FLASH 下载程序时，可能会出现图 *45_6* 的擦除 FLASH 失败的错误提示。



只要不是把读保护配置成了级别 2 保护，都可以使用 SRAM 启动运行*代码清单 45_7* 中的函数恢复选项字节为默认状态，使得 FLASH 下载能正常进行。

列表 5: 代码清单 45-7 恢复选项字节为默认值

```
/**
* @brief  InternalFlash_Reset, 恢复内部 FLASH 的默认配置
* @param  None
* @retval None
*/
HAL_StatusTypeDef InternalFlash_Reset(void)
{

   /* 使能访问选项字节寄存器 */
   HAL_FLASH_OB_Unlock();

   FLASH_INFO("\r\n");
   FLASH_INFO("正在准备恢复的条件，请耐心等待...");
   //选项字节全部恢复默认值
   OBInit.OptionType = OPTIONBYTE_WRP|OPTIONBYTE_RDP|OPTIONBYTE_
↪USER|OPTIONBYTE_BOR|\
                       OPTIONBYTE_BOOTADDR_0|OPTIONBYTE_BOOTADDR_1;
   OBInit.WRPSector  = OB_WRP_SECTOR_All;
   OBInit.RDPLevel   = OB_RDP_LEVEL_0;
   OBInit.USERConfig  = OB_WWDG_SW|OB_IWDG_SW|OB_STOP_NO_RST|OB_STDBY_NO_
↪RST|\
                     OB_IWDG_STOP_ACTIVE|OB_IWDG_STDBY_ACTIVE|OB_DUAL_BOOT_
↪DISABLE|\
                       OB_NDBANK_SINGLE_BANK;
   OBInit.BORLevel = OB_BOR_OFF;
   OBInit.BootAddr0 = OB_BOOTADDR_ITCM_FLASH;
   OBInit.BootAddr1 = OB_BOOTADDR_SYSTEM;
   HAL_FLASHEx_OBProgram(&OBInit);

   if (HAL_FLASH_OB_Launch() != HAL_OK) {
```

(下页继续)

(续上页)

```
        FLASH_ERROR("对选项字节编程出错，恢复失败");
        return HAL_ERROR;
    }
    FLASH_INFO("恢复选项字节默认值成功！");
    //禁止访问
    HAL_FLASH_OB_Lock();

    return HAL_OK;
}
```

这个函数进行了如下操作：

1、调用 HAL_FLASH_OB_Unlock 解锁选项字节的编程；

2、初始化 FLASH_OBProgramInitTypeDef 结构体，并使用 HAL_FLASHEx_OBProgram 函数将选项字节相关的位都恢复默认值；

3、调用 HAL_FLASH_OB_Launch 定稿选项字节并等待设置完毕，由于这个过程需要擦除内部 FLASH 的内容，等待的时间会比较长；

4、恢复选项字节为默认值操作完毕。

50.4.2.2.3 main 函数

最后来看看本实验的 main 函数，见*代码清单 45_4*。

列表 6: 代码清单 45-4 main 函数

```
int main(void)
{
    /* 系统时钟初始化成 168 MHz */
    SystemClock_Config();

    /* LED 端口初始化 */
```

(下页继续)

(续上页)

```
    LED_GPIO_Config();
    /* 串口初始化 */
    DEBUG_USART_Config();
    LED_BLUE;

    FLASH_INFO("本程序将会被下载到 STM32 的内部 SRAM 运行。");

    FLASH_INFO("\r\n");
    FLASH_INFO("----这是一个 STM32 芯片内部 FLASH 解锁程序----");
    FLASH_INFO("程序会把芯片的内部 FLASH 选项字节恢复为默认值");

#if 0
//工程调试、演示时使用，正常解除时不需要运行此函数
    WriteProtect_Test();
                        //修改写保护位，仿真芯片扇区被设置成写保护的的环境
#endif

    OptionByte_Info();

    /* 恢复选项字节到默认值，解除保护 */
    if (InternalFlash_Reset()==HAL_OK) {
          FLASH_INFO("选项字节恢复成功");
          FLASH_INFO("然后随便找一个普通的程序，下载程序到芯片的内部 FLASH 进行测试");
          LED_GREEN;
    } else {
          FLASH_INFO("选项字节恢复成功失败，请复位重试");
          LED_RED;
    }

    OptionByte_Info();

    while (1) {
```

(下页继续)

(续上页)

```
    }
}
```

在 main 函数中，主要是调用了 InternalFlash_Reset 函数把选项字节恢复成默认值，程序默认时没有调用 WriteProtect_Test 函数设置写保护，若您想观察实验现象，可修改条件编译的宏，使它加入到编译中。

50.4.3 下载测试

把开发板的 BOOT0 和 BOOT1 引脚都使用跳线帽连接到 3.3V 电源处，使它以 SRAM 方式启动，然后用 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并复位运行，在串口调试助手可看到调试信息。程序运行后，请耐心等待至开发板亮绿灯或串口调试信息提示恢复完毕再给开发板断电，否则由于恢复过程被中断，芯片内部 FLASH 会处于保护状态。

芯片内部 FLASH 处于保护状态时，可重新下载本程序到开发板以 SRAM 运行恢复默认配置。

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

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/Embdefire/embed_linux_tutorial
- gitee 仓库地址：https://gitee.com/wildfireteam/embed_linux_tutorial

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。